Welcome

Java Concurrency Collections

Copyright Notice

Java Concurrency Collections

Prakash Badhe

prakash.badhe@vishwasoft.in

Concurrent Collections

- The java.util.concurrent package includes a number of additions to the Java Collections Framework.
- 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

- 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.

ConcurrentNavigableMap

- 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.

ACID in Operations

- **Atomicity**, **Consistency**, **isolation** and **durability**.
- **Atomicity** is either all successful or none.
- **Consistency** ensures bringing the database from one consistent state to another consistent state.
- **Isolation** ensures that transaction is isolated from other transaction.
- **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Atomicity

- Atomicity is one of the key concepts in multi-threaded programs.
- A set of actions is atomic if they all execute as a single operation, in an indivisible manner.
- Assuming that a set of actions in a multi-threaded program will be executed serially may lead to incorrect results.
- The reason is due to thread interference, which means that if two threads execute several steps on the same data, they may overlap.

Race Condition

- When shared the state of an object (its data) without synchronization across the threads , it leads to the presence of race conditions.
- The code will have a race condition if there's a possibility to produce incorrect results due to thread interleaving.
- Two types of race conditions:
  - Check-then-act
  - Read-modify-write

Check-then-act race condition

- To remove race conditions and enforce thread safety, we must make these actions atomic by using synchronization.
- This race condition appears when you have a shared field and expect to serially execute the following steps
  - Get a value from a field.
  - Do something based on the result of the previous check.

Effects of Race Condition

- The problem here is that when the first thread is going to act after the previous check, another thread may have interleaved and changed the value of the field.
- Now, the first thread will act based on a value that is no longer valid.

Read-modify-write race condition

- Type of race condition which appears when executing the following set of actions:
  - Fetch a value from a field.
  - Modify the value.
  - Store the new value to the field.
- Here's another dangerous possibility which consists in the loss of some updates to the field.

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.
- The **Java** language specification guarantees that readin**g** or writing a variable is an **atomic operation**(unless the variable is of type long or double)

Atomic Variables

- ***The Atomic Variables*** provided by the Java Concurrency API in the **java.util.concurrent.atomic** package.
- **AtomicBoolean**
- **AtomicInteger**
- **AtomicLong**
- These are wrapper of primitive types boolean, integer and long, with the difference: they are designed to be safely used in multi-threaded context.

Atomic Operations

- These are called atomic variables because they provide some operations that cannot be interfered by multiple threads
- **incrementAndGet()**: Atomically increments by one the current value.
- **decrementAndGet()**: Atomically decrements by one the current value.

Atomic Operations

- 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 across threads

- These operations are guaranteed to execute atomically using machine-level instructions on modern processors.
- Using atomic variables help avoiding the overhead of synchronization on a single primitive variable, so it is more efficient than using synchronization/locking mechanism.

Atomic vs. Synchronization

- synchronization/locking comes at the cost of slow performance as it requires resources and thread scheduler to monitor the lock.
- Therefore, atomic variable is a good alternative to synchronization on a single primitive type as mentioned earlier, atomic variable uses machine-level instructions to guarantee atomicity.

More atomic

- The AtomicInteger and AtomicLong classes provide other atomic methods such as:
- addAndGet(int delta): Atomically adds the given value to the current value.
- compareAndSet(int expect, int update): Atomically sets the value to the given updated value if the current value == the expected value.
- getAndAdd(int delta): Atomically adds the given value to the current value.
- set(int newValue): Sets to the given value.

Atomic Arrays

- Java Concurrency API also provides atomic arrays and atomic reference type
- AtomicIntegerArray
- AtomicLongArray
- AtomicReference
- AtomicReferenceArray

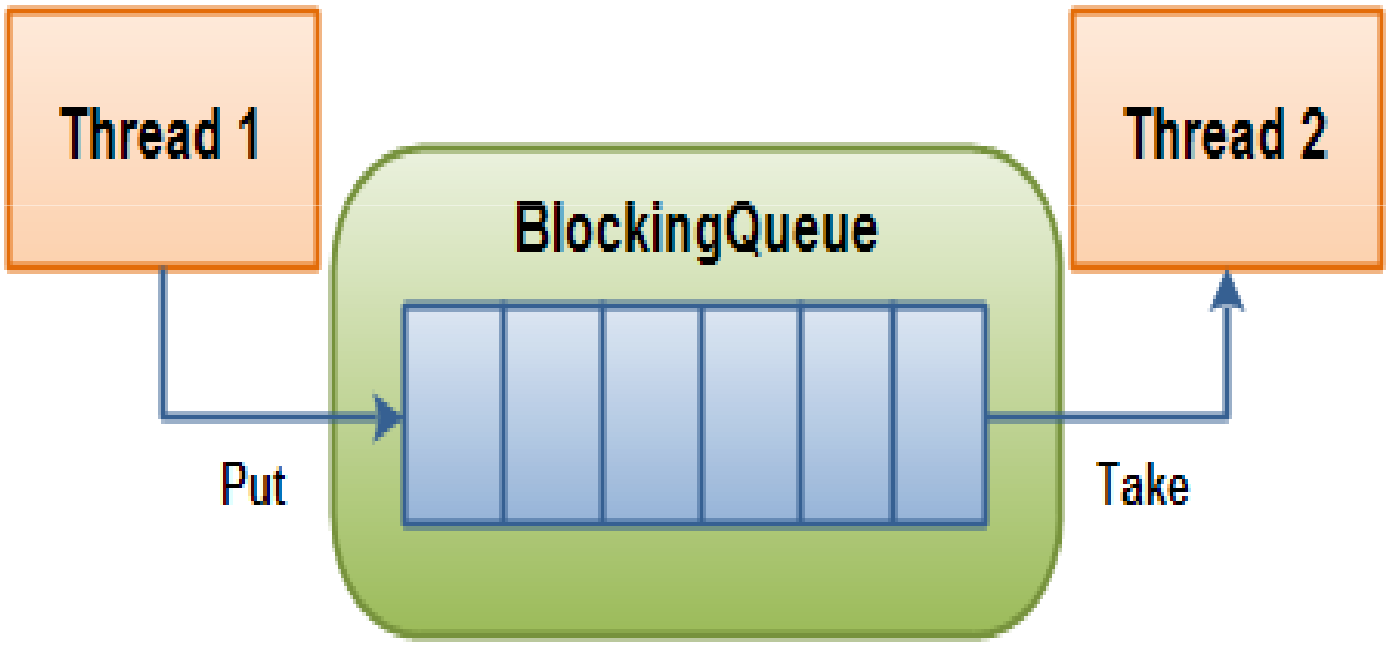
BlockingQueue

- The java.util.concurrent.BlockingQueue represents a queue which is thread safe to put elements into, and take elements out of from.
- The multiple threads can be inserting and taking elements concurrently from a BlockingQueue without any concurrency issues arising.

BlockingQueue in Threads

- BlockingQueue is capable of blocking the threads that try to insert or take elements from the queue.
- For instance, if a thread tries to take an element and there are none left in the queue, the thread can be blocked until there is an element to take.
- Whether or not the calling thread is blocked depends on what methods you call on the BlockingQueue.

Across threads



A BlockingQueue with one thread putting into it, and another thread taking from it.

BlockingQueue Queue

- A Queue that supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
- BlockingQueue methods come in four forms.
- Handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future.
- One throws an exception
- The second returns a special value (either null or false, depending on the operation),
- The third blocks the current thread indefinitely until the operation can succeed.
- Te fourth blocks for only a given maximum time limit before giving up.

Queue Operations

Operation	*Throws exception*	*Special value*	*Blocks*	*Times out*
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	*not applicable*	*not applicable*

Producer and Consumer

- The producing thread will keep producing new objects and insert them into the BlockingQueue, until the queue reaches some upper bound on what it can contain.
- If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.
- The consuming thread keeps taking objects out of the BlockingQueue to processes them.
- If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

BlockingQueue Specials

- It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw a NullPointerException.
- It is possible to access all the elements inside a BlockingQueue, and not just the elements at the start and end

BlockingQueue Implementations

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue