

concurrency lock concurrent linked list in java

Published: September 27, 2026 | Index ID: #-

Introduction

In modern Java applications, data structures that can safely handle multiple threads are essential. Whether you're building a real time messaging system, a high frequency trading engine, or a web server that processes thousands of concurrent requests, you need collections that maintain integrity under parallel access. One of the most common patterns is a **concurrent linked list in Java**, a thread safe, non blocking data structure that supports high throughput. This article takes a deep dive into the mechanics of concurrency locks, the design of concurrent linked lists, and how you can leverage these structures to write robust, scalable code.

Understanding Concurrency in Java

What Is Concurrency?

Concurrency refers to the execution of multiple threads or processes that may interleave or run in parallel. In Java, the `java.lang.Thread` class and the `java.util.concurrent` package provide the building blocks for creating concurrent applications. The challenge lies in ensuring that shared data remains consistent when accessed by many threads simultaneously.

Thread Safety and the Role of Locks

Thread safety is the guarantee that a piece of code will behave correctly when executed by multiple threads at the same time. Locks are the primary mechanism for achieving thread safety. They control access to shared resources by allowing only one thread to hold the lock at a time, effectively serializing critical sections of code.

Locks: The Backbone of Thread Safety

ReentrantLock and the synchronized Keyword

The classic way to protect shared data in Java is the `synchronized` keyword. It implicitly uses a monitor lock on the object, making the code block mutually exclusive. `ReentrantLock`, part of the `java.util.concurrent.locks` package, offers similar semantics but with additional features such as fairness policies, interruptible lock acquisition, and the ability to try acquiring a lock without blocking.

StampedLock and Read Write Semantics

For scenarios that require many readers but few writers, `StampedLock` provides a lightweight read write lock. It allows multiple threads to hold a read stamp simultaneously while still ensuring exclusive access for writers. The stamped lock is particularly useful when implementing data structures that are read heavy, like a concurrent linked list used as a cache.

Lock-Free Techniques and Atomic Variables

Lock-free programming avoids blocking entirely. Instead, it relies on atomic operations provided by classes such as `AtomicReference` and `AtomicIntegerFieldUpdater`. These operations use low-level CPU instructions like compare and swap (CAS) to achieve thread safety without acquiring a lock. Lock-free algorithms are highly desirable for high throughput concurrent collections.

Why Linked Lists in Concurrent Environments?

Traditional LinkedList vs. ConcurrentLinkedList

The `java.util.LinkedList` is not thread-safe. If multiple threads modify it concurrently, you risk data corruption or unpredictable behavior. A **concurrent linked list in Java** addresses these concerns by using fine grained locking or lock free techniques to allow safe concurrent modifications.

Use Cases for Concurrent Linked Lists

Concurrent linked lists are ideal when you need:

- Fast insertions and deletions at both ends.
- Unbounded size without pre allocation.
- Non-blocking reads for high read to write ratios.
- Ordered traversal that preserves insertion order.

Java's Concurrent Linked List Implementations

ConcurrentLinkedQueue

Part of the `java.util.concurrent` package, `ConcurrentLinkedQueue` is a lock free, unbounded FIFO queue. Internally, it uses a singly linked list with atomic references to nodes. It's designed for scenarios where you need a queue that can handle many concurrent producers and consumers without blocking.

ConcurrentLinkedDeque

Extending the queue concept, `ConcurrentLinkedDeque` provides a double ended queue (deque). It supports insertion and removal at both ends and is also lock free. The deque is particularly useful for implementing LRU caches or double ended pipelines.

CopyOnWriteArrayList

While not a linked list, `CopyOnWriteArrayList` is a thread safe list that achieves safety by copying the underlying array on each write. It's efficient for read heavy workloads but unsuitable for high write scenarios.

LinkedBlockingDeque

This implementation uses explicit locks (`ReentrantLock`) and a bounded capacity. It's blocking, meaning that producers will wait if the deque is full and consumers will wait if it's empty. It's a good fit for producer consumer patterns where you need back pressure.

Deep Dive: ConcurrentLinkedQueue

Internal Node Structure

Each node in the queue holds a value and an `AtomicReference` to the next node. The head and tail of the queue are also `AtomicReference` objects. The use of atomic references ensures that updates to the list are visible across threads without explicit synchronization.

Enqueue Operation (Offer)

1. Read the current tail node.
2. Attempt to set the next reference of the tail node to the new node using CAS.
3. If CAS succeeds, update the tail reference to the new node.
4. If CAS fails, retry until successful.

Dequeue Operation (Poll)

1. Read the current head node.
2. Read the value from the head node's next reference.
3. Attempt to move the head reference to the next node using CAS.
4. If CAS succeeds, return the value.
5. If CAS fails, retry.

Because the operations use CAS, they do not block. If contention is high, threads may retry several times, but the algorithm guarantees progress.

Performance and Memory Considerations

Lock Contention vs. Throughput

Fine grained locking reduces contention but adds overhead due to acquiring and releasing many locks. Lock free algorithms eliminate this overhead but can suffer from high CPU usage due to retries. Choosing the right strategy depends on the workload: high read to write ratios favor lock free structures, whereas mixed workloads may benefit from fine grained locks.

Garbage Collection Impact

Linked lists create many small node objects. In a long running application, this can lead to frequent garbage collections. Using `ConcurrentLinkedQueue` or `ConcurrentLinkedDeque` mitigates this by reusing node objects when possible. However, for extremely high throughput, consider pooling node objects or using array based structures.

Cache Locality

Linked lists suffer from poor cache locality because nodes can be scattered in memory. This can degrade performance on modern CPUs. Techniques such as node pooling or using segmented structures can improve locality.

Locking Strategies for Linked Lists

Coarse Grained Locking

Acquiring a single lock for the entire list is simple but can become a bottleneck. It's suitable for low concurrency scenarios or when the list is rarely accessed.

Fine Grained Locking

Locking individual nodes or segments allows multiple threads to operate on different parts of the list concurrently. This approach increases complexity but can dramatically improve throughput.

Lock Free Approaches

Using atomic operations and CAS, lock free linked lists avoid blocking. The classic Michael Scott queue is a prime example. While lock free structures are faster under contention, they require careful design to avoid subtle bugs such as the ABA problem.

Implementing Your Own Concurrent Linked List

Example: A Simple Lock Free Linked List

The following code demonstrates a minimal lock free singly linked list using `AtomicReference`. It supports insertion at the head and traversal.

```

import java.util.concurrent.atomic.AtomicReference;

public class SimpleLockFreeList<T> {
    private static class Node<T> {
        final T value;
        final AtomicReference<Node<T>> next;
        Node(T value, Node<T> next) {
            this.value = value;
            this.next = new AtomicReference<>(next);
        }
    }

    private final AtomicReference<Node<T>> head = new AtomicReference<>(null);

    public void push(T value) {
        Node<T> newNode = new Node<>(value, null);
        while (true) {
            Node<T> currentHead = head.get();
            newNode.next.set(currentHead);
            if (head.compareAndSet(currentHead, newNode)) {
                return;
            }
        }
    }

    public void forEach(java.util.function.Consumer<T> action) {
        Node<T> node = head.get();
        while (node != null) {
            action.accept(node.value);
            node = node.next.get();
        }
    }
}

```

Note that this example does not handle removal or concurrent traversal updates. In practice, you would need to address the ABA problem and memory reclamation.

Best Practices and Pitfalls

Deadlocks and Ordering

When using multiple locks, always acquire them in a consistent order to avoid deadlocks. In a concurrent linked list that uses fine grained locks, this means locking nodes in a predictable sequence.

ABA Problem

In lock free algorithms, a node may be removed and later reinserted, causing a CAS to succeed incorrectly. Using

version tags or hazard pointers can mitigate this risk.

Memory Consistency Errors

Java's memory model guarantees that actions before a successful CAS

Baca Juga (Artikel Terkait)

- [composite nonwoven materials structure properties and applications](http://basicgravity.com/composite-nonwoven-materials-structure-properties-and-applications.pdf)

URL: <http://basicgravity.com/composite-nonwoven-materials-structure-properties-and-applications.pdf>

- [composite materials science and engineering chawla](http://basicgravity.com/composite-materials-science-and-engineering-chawla.pdf)

URL: <http://basicgravity.com/composite-materials-science-and-engineering-chawla.pdf>

- [composite materials in aerospace applications ijsrp](http://basicgravity.com/composite-materials-in-aerospace-applications-ijsrp.pdf)

URL: <http://basicgravity.com/composite-materials-in-aerospace-applications-ijsrp.pdf>

- [composite materials for aircraft structures](http://basicgravity.com/composite-materials-for-aircraft-structures.pdf)

URL: <http://basicgravity.com/composite-materials-for-aircraft-structures.pdf>

Tags: [#concurrency](#) [#lock](#) [#concurrent](#) [#linked](#) [#list](#) [#java](#)

