Nayankumar Dhome
nayankumardhome@gmail.com

Concurrent HashMap in Java

A Deep Dive into Thread-Safe HashMaps

Swipe To Know More

Nayankumar Dhome
nayankumardhome@gmail.com

01

Introduction

In modern multi-threaded applications, efficiently managing shared data is crucial. ConcurrentHashMap is a thread-safe, high-performance version of HashMap that enables concurrent read/write operations without blocking the entire map.

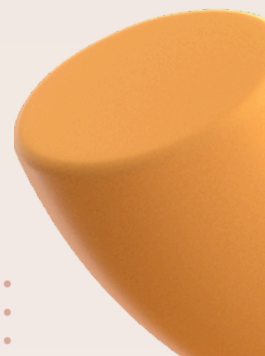
Nayankumar Dhome
nayankumardhome@gmail.com

02

What is ConcurrentHashMap?

A ConcurrentHashMap is a thread-safe, high-performance version of HashMap, designed for scenarios where multiple threads read/write concurrently.

- **Thread-safe:** Multiple threads can modify the map safely.
- **High performance:** Uses fine-grained locks instead of global synchronization.
- **Non-blocking reads:** Read operations execute without locks, improving speed.



Nayankumar Dhome
nayankumardhome@gmail.com

03

Why Use ConcurrentHashMap?

- **Faster than synchronized HashMap** – Avoids global synchronization.
- **Ideal for multi-threaded environments** – Prevents ConcurrentModificationException.
- **Allows high read throughput** – Read operations run without blocking.
- **Better scalability** – Threads can operate on different parts of the map simultaneously.
- **Safe in concurrent scenarios** – Unlike HashMap, which can enter an infinite loop due to concurrent modifications.

Nayankumar Dhome
nayankumardhome@gmail.com

04

Key Features of ConcurrentHashMap

- Thread-safe & lock-free reads – Reads don't require synchronization.
- Segmented locking (pre-Java 8) → Fine-grained locking (Java 8+).
- Supports parallel execution using forEach(), search(), reduce().
- Does not allow null keys or values to prevent ambiguity in concurrent settings.

Nayankumar Dhome
nayankumardhome@gmail.com

05

Internal Working of ConcurrentHashMap

Pre-Java 8 (**Segment-based locking**)

- The map was divided into segments, each acting as a smaller HashMap with its own lock.
- This allowed multiple threads to access different segments simultaneously.

Nayankumar Dhome
nayankumardhome@gmail.com

06

Internal Working of ConcurrentHashMap

Java 8+ (Fine-grained locking with CAS & Bins)

- The segment-based approach was replaced with a fine-grained locking mechanism using Compare-And-Swap (CAS).
- Nodes are stored in bins, and updates occur using synchronized blocks per bin instead of locking the entire map.
- Improves performance by allowing concurrent writes to different keys without blocking.

Nayankumar Dhome
nayankumardhome@gmail.com

07

ConcurrentHashMap Methods & Explanation

- **put(K key, V value):** Inserts a key-value pair without locking the entire map.
- **get(K key):** Retrieves a value with lock-free read.
- **remove(K key):** Removes a key-value pair atomically.
- **computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction):** Computes a value only if absent, preventing race conditions.

Nayankumar Dhome
nayankumardhome@gmail.com

08

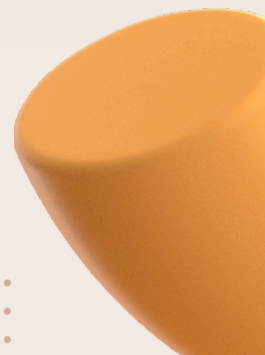
ConcurrentHashMap Methods & Explanation

- **computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction):** Updates value only if present, ensuring atomic updates.
- **forEach(long parallelismThreshold, BiConsumer<? super K, ? super V> action):** Performs parallel execution for better performance.
- **merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction):** Merges values atomically, ensuring safe updates.

Nayankumar Dhome
nayankumardhome@gmail.com

09

Real-World Use Cases of ConcurrentHashMap



Nayankumar Dhome
nayankumardhome@gmail.com

10

High-Concurrency Caching System

- Use ConcurrentHashMap for real-time caching, ensuring fast, thread-safe access.

```java
import java.util.concurrent.*;

public class CacheService {
    private final ConcurrentHashMap<String, String> cache = new ConcurrentHashMap<>();

    public void addToCache(String key, String value) {
        cache.put(key, value);
    }

    public String getFromCache(String key) {
        return cache.getOrDefault(key, "Not Found");
    }

    public static void main(String[] args) {
        CacheService cacheService = new CacheService();
        cacheService.addToCache("user1", "Nayankumar Dhome");
        System.out.println(cacheService.getFromCache("user1")); // Output: Nayankumar Dhome
    }
}
```

Nayankumar Dhome
nayankumardhome@gmail.com

11

Preventing Race Conditions in Multi-Threaded Counters

- Use computeIfAbsent() to initialize and safely update counters in concurrent environments.

```java
import java.util.concurrent.*;

public class ConcurrentCounter {
    private static ConcurrentHashMap<String, Integer> counter = new ConcurrentHashMap<>();

    public static void increment(String key) {
        counter.compute(key, (k, v) -> (v == null) ? 1 : v + 1);
    }

    public static void main(String[] args) {
        increment("profilePageView");
        increment("profilePageView");
        System.out.println(counter.get("profilePageView")); // Output: 2
    }
}
```

Nayankumar Dhome
nayankumardhome@gmail.com

12

Parallel Processing using forEach()

- Use forEach() for parallel iteration over large datasets.

```java
import java.util.concurrent.*;

public class ParallelProcessing {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> tasks = new ConcurrentHashMap<>();
        tasks.put(101, "Implement login functionality");
        tasks.put(102, "Build API endpoints");
        tasks.put(103, "Fix UI bugs");
        tasks.put(104, "Optimize database queries");
        tasks.put(105, "Write unit tests");

        // Using forEach to process the tasks in parallel
        tasks.forEach(1, (developerId, task) -> {
            System.out.println("Developer #" + developerId + " is assigned to: " + task);
        });
    }
}
```

Nayankumar Dhome
nayankumardhome@gmail.com

13

Safe Merging of Values using merge()

- Use merge() for atomic updates, ensuring safe aggregation.

```java
import java.util.concurrent.*;

public class SafeMerging {
    public static void main(String[] args) {
        // Simulating diamond collection for players in an online game
        ConcurrentHashMap<String, Integer> diamondCollection = new ConcurrentHashMap<>();

        // Initial diamond count for players
        diamondCollection.put("Nayankumar", 5);  // Nayankumar starts with 5 diamonds
        diamondCollection.put("Kartik", 3);    // Kartik starts with 3 diamonds

        // Nayankumar collects 7 more diamonds
        diamondCollection.merge("Nayankumar", 7, Integer::sum);  // Nayankumar's diamonds become 12

        // Kartik collects 5 more diamonds
        diamondCollection.merge("Kartik", 5, Integer::sum);    // Kartik's diamonds become 8

        // Tushar is a new player, and collects 10 diamonds
        diamondCollection.merge("Tushar", 10, Integer::sum); // Tushar's diamonds become 10

        // Output the final diamond count for all players
        System.out.println(diamondCollection); // Output: {Nayankumar=12, Kartik=8, Tushar=10}
    }
}
```

Nayankumar Dhome
nayankumardhome@gmail.com

14

Pros & Cons of ConcurrentHashMap

- ✅ **Advantages**
- **Thread-Safe Reads & Writes** – Uses fine-grained locks for efficiency.
- **Lock-Free Reads** – Read operations do not block other threads.
- **Parallel Execution** – Uses forEach(), search(), reduce().
- **Avoids Deadlocks** – Does not block entire map like synchronizedMap().

- ❌ **Disadvantages**
- **Higher Memory Usage** – Stores additional metadata for concurrency control.
- **Slower than HashMap** (**Single-threaded**) – Slight overhead due to locking.
- **No Null Keys**/**Values** – Unlike HashMap, does not allow null keys/values.

Nayankumar Dhome
nayankumardhome@gmail.com

15

Did You Know?

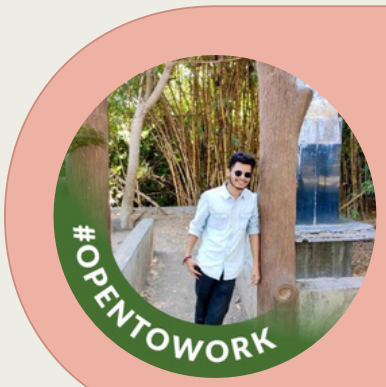
- ConcurrentHashMap was inspired by Doug Lea's work on concurrent data structures.
- In Java 8+, compute() operations use atomic updates, avoiding locks in many cases.
- Google's Guava Library extends ConcurrentHashMap for even more advanced use cases!

Nayankumar Dhome
nayankumardhome@gmail.com

16

Final Thoughts

- **Best Choice for Multi-Threaded Applications** – ConcurrentNavigableMap provides a thread-safe alternative to TreeMap, ensuring sorted key-value storage without explicit locking.
- **Efficient Concurrent Access** – With its Skip List structure, it allows fast read/write operations in multi-threaded environments without performance bottlenecks.
- **Ideal for Range-Based Queries** – Methods like subMap(), headMap(), and tailMap() make it perfect for time-series databases, real-time analytics, and financial applications.
- **Balances Performance & Scalability** – Unlike TreeMap, which needs external synchronization, ConcurrentNavigableMap ensures high concurrency while maintaining sorted order.



If you found this helpful?

please like, share with your network, and follow us for more insightful content

Nayankumar Dhome
nayankumardhome@gmail.com