Java’s Collections.synchronizedList() Explained

Alexander Obregon

12 min read

Sep 26, 2024

Image Source

Introduction

The `Collections.synchronizedList()` method from the `java.util.Collections` class is a key tool for managing concurrent access to a list in Java. This method wraps a list with synchronized access, ensuring thread safety. In this article, we will explore how `synchronizedList()` works, why it is important in concurrent applications, and compare it to other synchronization strategies. We will also cover real-world examples to demonstrate its use and limitations.

Thread Safety and Synchronization

In Java, when developing multi-threaded applications, making sure that shared resources are accessed safely by multiple threads is a critical concern. One of the common challenges is managing access to collections, such as lists, where multiple threads may be adding, removing, or modifying elements concurrently. If access to these collections is not synchronized, unpredictable behavior, including data corruption and runtime exceptions, can occur.

The `synchronizedList()` method from the `java.util.Collections` class addresses this by providing synchronized access to lists, ensuring thread safety. However, before we dive into the specifics of how this method works, it is important to understand the basics of thread safety and why synchronization is essential in multi-threaded applications.

What Is Thread Safety?

Thread safety refers to the ability of a program to function correctly when multiple threads access shared resources concurrently. In the context of collections like lists, thread safety makes sure that the list remains in a consistent state even when multiple threads are performing operations on it simultaneously.

Consider the following scenario in a multi-threaded application where multiple threads are adding elements to a shared list:

```
List<String> list = new ArrayList<>();
list.add("Thread1");
list.add("Thread2");
```

In this simple example, the list is accessed by two different threads. If the threads attempt to modify the list at the same time, unpredictable behavior can occur. For instance, the internal state of the `ArrayList` might become corrupted, or some elements might be lost, leading to data inconsistency. This is because `ArrayList` is not thread-safe by default, meaning it does not provide built-in mechanisms to manage concurrent modifications by multiple threads.

Without thread safety mechanisms in place, operations like adding, removing, or accessing elements from the list could interleave in such a way that the internal structure of the list becomes inconsistent, resulting in unpredictable or erroneous outcomes. In extreme cases, such issues could lead to application crashes.

Why Is Synchronization Necessary?

Synchronization is a mechanism used to control the access of multiple threads to a shared resource. In the context of collections, synchronization makes sure that only one thread can access or modify the collection at a time, preventing concurrent threads from interfering with each other's operations. This prevents data corruption, race conditions, and other concurrency-related issues.

A **race condition** occurs when two or more threads try to change shared data at the same time, and the final result depends on the timing of the threads' execution. Without proper synchronization, the behavior of the program may vary unpredictably based on which thread completes its operation first. Here's a simple example of a race condition:

```
List<Integer> list = new ArrayList<>();
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 1000; i++) {
        list.add(i);
    }
});

Thread t2 = new Thread(() -> {
    for (int i = 1000; i < 2000; i++) {
        list.add(i);
    }
});

t1.start();
t2.start();
```

In this scenario, `t1` and `t2` both attempt to add numbers to the shared list. Without synchronization, the final contents of the list are uncertain. It is possible that some values will be lost, duplicated, or even cause the list to enter an invalid state, because `ArrayList` is not designed to handle concurrent modifications from multiple threads. The order in which the threads execute and access the shared list is non-deterministic, which can lead to inconsistencies in the data.

This is where **synchronization** comes in. By synchronizing access to the list, you make sure that only one thread can modify the list at a time, avoiding such issues.

Common Synchronization Techniques in Java

In Java, synchronization is typically achieved using either **synchronized blocks** or **concurrent collections**. Let's take a closer look at these two approaches:

The most common way to achieve thread safety is by using a synchronized block. This makes sure that a section of code can only be executed by one thread at a time. Here's an example:

```
List<Integer> list = new ArrayList<>();
Thread t1 = new Thread(() -> {
    synchronized (list) {
        for (int i = 0; i < 1000; i++) {
            list.add(i);
        }
    }
});

Thread t2 = new Thread(() -> {
    synchronized (list) {
        for (int i = 1000; i < 2000; i++) {
            list.add(i);
        }
    }
```

});

t1.start();
t2.start();

In this case, the `synchronized` block makes sure that only one thread can add elements to the list at a time. When a thread enters the synchronized block, it locks the list, preventing other threads from accessing it until the lock is released. This effectively eliminates race conditions, but it also introduces a performance overhead because threads must wait for the lock to be released.

Java also provides a set of **concurrent collections** that are designed to handle multi-threaded environments more efficiently than synchronized blocks. For example, `ConcurrentHashMap`, `ConcurrentLinkedQueue`, and `CopyOnWriteArrayList` are all thread-safe collections that provide better performance by allowing more fine-grained control over synchronization.

However, the `Collections.synchronizedList()` method offers a simpler alternative for developers who need a thread-safe list without having to switch to more complex concurrent collections. It wraps an existing list and synchronizes all of its operations, providing basic thread safety while allowing developers to continue using familiar list implementations like `ArrayList` and `LinkedList`.

How Collections.synchronizedList() Works

The `Collections.synchronizedList()` method is a utility provided by the `java.util.Collections` class that allows developers to wrap a list with synchronized access. This method makes sure that all operations on the list are thread-safe, meaning that only one thread can access the list at any given time. In this section, we will explore how this method works and the best practices for using it in concurrent environments.

Creating a Synchronized List

The `synchronizedList()` method takes any existing `List` implementation, such as `ArrayList` or `LinkedList`, and returns a synchronized (thread-safe) version of it. Here's an example of how to create a synchronized list:

List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>());

In this example, `synchronizedList` is a thread-safe version of `ArrayList`. Any operations on this list, such as adding or removing elements, will now be synchronized, making sure that only one thread can modify the list at a time. This prevents concurrent access issues, such as race conditions or data inconsistencies.

Synchronized Access to the List

When you use `Collections.synchronizedList()`, all modifications to the list are automatically synchronized. This means that operations such as adding, removing, or updating elements are wrapped in a synchronized block, making sure that only one thread can perform these operations at a time. Here's a closer look at what happens under the hood:

```
public static <T> List<T> synchronizedList(List<T> list) {
    return new SynchronizedList<>(list);
}
private static class SynchronizedList<E> implements List<E>, Serializable {
    private final List<E> list;


    SynchronizedList(List<E> list) {
        this.list = list;
    }
    public synchronized boolean add(E e) {
        return list.add(e);
    }
    public synchronized E get(int index) {
        return list.get(index);
    }


}
```

In the code snippet above, the `SynchronizedList` class wraps the original list and synchronizes each method. For example, the `add()` method is synchronized, meaning that if one thread is adding an element to the list, any other thread attempting to modify the list will be blocked until the `add()` operation is complete. This makes sure that modifications are performed atomically, preventing race conditions.

It is important to note that this synchronization applies to individual methods of the list. For example, if two threads try to call `add()` at the same time, one will be blocked until the other finishes. However, this does not automatically protect the list from more complex operations that span multiple method calls, such as iteration.

Synchronizing Iteration Over a List

One key limitation of `Collections.synchronizedList()` is that it does not automatically synchronize iteration over the list. This means that if you are iterating over the list in a multi-threaded environment, you need to manually synchronize the iteration block to avoid `ConcurrentModificationException`. Here's how you can safely iterate over a synchronized list:

```
List<String> list = Collections.synchronizedList(new ArrayList<>());
list.add("A");
list.add("B");
list.add("C");
synchronized (list) {
    for (String item : list) {
        System.out.println(item);
    }
}
```

In this example, the `synchronized` block makes sure that no other thread can modify the list while it is being iterated over. Without this manual synchronization, modifying the list during iteration would throw a `ConcurrentModificationException`.

The need for manual synchronization when iterating is one of the most important considerations when using `Collections.synchronizedList()`. Although the list itself is thread-safe for individual operations, more complex scenarios (like iterating and modifying the list at the same time) still require careful synchronization.

Performance Considerations

While `Collections.synchronizedList()` is a simple and effective way to ensure thread-safe access to a list, it does come with performance overhead. Since every operation on the list is synchronized, this introduces additional latency due to the locking mechanism. For example, if multiple threads frequently access or modify the list, the locking and unlocking process can slow down performance, especially in high-concurrency environments.

In some cases, this performance cost may be acceptable, but in high-performance systems, other alternatives like `CopyOnWriteArrayList` or concurrent collections (such as those provided by the `java.util.concurrent` package) may offer better performance.

It’s important to weigh the trade-offs between simplicity and performance when deciding whether to use `Collections.synchronizedList()`. For applications where the number of writes is low, and most operations are reads, `Collections.synchronizedList()` can be an excellent solution. However, if your application involves frequent writes or requires high throughput, consider using more specialized concurrent collections.

Example: Using Collections.synchronizedList()

Let’s look at a practical example of how `Collections.synchronizedList()` can be used in a multi-threaded application. Imagine you have a shared list that multiple threads are modifying:

```
List<String> sharedList = Collections.synchronizedList(new ArrayList<>());
Thread t1 = new Thread(() -> {
  for (int i = 0; i < 100; i++) {
    sharedList.add("Thread1 - " + i);
  }
});

Thread t2 = new Thread(() -> {
  for (int i = 0; i < 100; i++) {
    sharedList.add("Thread2 - " + i);
  }
});

t1.start();
t2.start();

t1.join();
t2.join();

System.out.println("Size of list: " + sharedList.size());
```

In this example, two threads (`t1` and `t2`) are concurrently adding elements to the shared list. By using `Collections.synchronizedList()`, we make sure that the list remains thread-safe, and no data is lost or corrupted during the concurrent modifications. The `synchronizedList()` method wraps the list and synchronizes all operations, preventing race conditions and making sure that the list is modified safely.

Comparing Collections.synchronizedList() to Other Synchronization Strategies

While `Collections.synchronizedList()` provides a simple and convenient way to make a list thread-safe, it's important to understand that it's not the only option available for synchronization in Java. Depending on the specific requirements of your application, other synchronization strategies might offer better performance, scalability, or flexibility. Here, we'll compare `Collections.synchronizedList()` to other popular synchronization strategies, such as `CopyOnWriteArrayList`, concurrent collections, and manual synchronization using explicit locks.

CopyOnWriteArrayList

One common alternative to `Collections.synchronizedList()` is the `CopyOnWriteArrayList` class, which is part of the `java.util.concurrent` package. Unlike

`Collections.synchronizedList()`, which locks the entire list during every modification, `CopyOnWriteArrayList` takes a different approach: it creates a new copy of the list every time an element is added, removed, or modified.

In a `CopyOnWriteArrayList`, all modifications result in a new copy of the underlying array. This means that read operations can be performed without synchronization, as readers are working with a consistent snapshot of the list. Writes, however, are more expensive, as they involve creating a new copy of the list.

Here's an example of how to use `CopyOnWriteArrayList`:

```
List<String> list = new CopyOnWriteArrayList<>();
list.add("Item1");
list.add("Item2");
for (String item : list) {
   System.out.println(item);
}
```

In this example, reads are fast and do not block other threads. However, adding elements to the list triggers a new copy of the list, which incurs a performance cost, particularly when modifications are frequent.

`CopyOnWriteArrayList` is best suited for situations where the list is frequently read but infrequently modified. Because reads do not require locking, `CopyOnWriteArrayList` can provide better performance in scenarios where the list is accessed by multiple threads for reading far more often than for writing.

However, if your application involves frequent modifications to the list, the overhead of copying the list for every write operation can degrade performance. In such cases, `Collections.synchronizedList()` or other concurrent collections might be more appropriate.

Concurrent Collections

Java provides a rich set of concurrent collections in the `java.util.concurrent` package, designed specifically for multi-threaded environments. These collections, such as `ConcurrentHashMap`, `ConcurrentLinkedQueue`, and `ConcurrentSkipListSet`, offer more fine-grained control over synchronization, often allowing multiple threads to perform non-conflicting operations concurrently without the need for explicit synchronization.

For lists, the closest equivalent to `Collections.synchronizedList()` is `ConcurrentLinkedDeque`, which is a thread-safe, non-blocking double-ended queue that can function as a list. Unlike `Collections.synchronizedList()`, which locks the entire list for each operation, `ConcurrentLinkedDeque` uses non-blocking synchronization, allowing multiple threads to access the collection concurrently.

Example of Using ConcurrentLinkedDeque

```
Deque<String> deque = new ConcurrentLinkedDeque<>();
deque.add("Item1");
deque.add("Item2");
for (String item : deque) {
    System.out.println(item);
}
```

In this example, multiple threads can add or remove elements from the deque without locking the entire collection, resulting in better performance for highly concurrent applications.

Concurrent collections like `ConcurrentLinkedDeque` are ideal for high-concurrency environments where multiple threads frequently modify the collection. These collections are designed to minimize locking and maximize parallelism, making them a good choice for applications that require both high throughput and thread safety.

However, they are more complex than `Collections.synchronizedList()` and may introduce additional overhead in terms of memory usage or implementation complexity. If your application does not require high concurrency or fine-grained control, `Collections.synchronizedList()` may be simpler to implement.

Manual Synchronization Using Explicit Locks

Another alternative to `Collections.synchronizedList()` is manual synchronization using explicit locks, such as those provided by the `java.util.concurrent.locks` package. With explicit locks, you can synchronize only the critical sections of your code, providing more control over which operations need to be synchronized and which do not.

The `ReentrantLock` class is a flexible alternative to Java's built-in `synchronized` keyword. It allows you to manually acquire and release locks around specific code blocks, giving you more fine-grained control over synchronization.

Here's an example of how to use `ReentrantLock` to synchronize access to a list:

```
ReentrantLock lock = new ReentrantLock();
List<String> list = new ArrayList<>();
Thread t1 = new Thread(() -> {
    lock.lock();
    try {
        list.add("Thread1");
    } finally {
        lock.unlock();
    }
});
```

```
Thread t2 = new Thread(() -> {
    lock.lock();
    try {
        list.add("Thread2");
    } finally {
        lock.unlock();
    }
});

t1.start();
t2.start();
```

In this example, the `ReentrantLock` makes sure that only one thread can modify the list at a time. By manually locking and unlocking the list, you can control precisely which operations need to be synchronized, avoiding unnecessary synchronization for read-only operations or less critical sections of code.

Explicit locks are useful when you need more flexibility or control over synchronization than `Collections.synchronizedList()` or concurrent collections provide. For example, you might only need to synchronize certain operations on the list, or you might want to implement a custom locking strategy to improve performance.

However, manual synchronization can be error-prone, as it requires careful handling of lock acquisition and release. Failing to release a lock (for example, due to an exception) can result in deadlocks, where two or more threads are waiting indefinitely for each other to release locks. For most applications, `Collections.synchronizedList()` or concurrent collections provide safer and simpler solutions.

Comparing Performance and Use Cases

When deciding which synchronization strategy to use, it’s important to consider the specific requirements of your application. Here’s a quick comparison of the strategies we’ve discussed:

Collections.synchronizedList(): Simple to implement and ensures thread safety for basic list operations. Best for applications with moderate concurrency where performance is not a critical concern. It may introduce performance overhead due to locking every operation.

CopyOnWriteArrayList: Ideal for situations where reads are frequent and writes are infrequent. Provides excellent performance for read-heavy applications but can be costly when modifications are frequent due to the need to copy the list on every write.

Concurrent Collections (e.g., ConcurrentLinkedDeque): Best for high-concurrency applications where multiple threads frequently modify the collection. Provides better scalability and performance in highly concurrent environments but introduces more complexity than `Collections.synchronizedList()`.

Manual Synchronization (e.g., ReentrantLock): Offers the most flexibility and control over synchronization. Useful for applications that need custom synchronization strategies but can be error-prone and harder to maintain.

Summary of the Comparison

Simple thread safety: Use `Collections.synchronizedList()` for basic thread-safe lists where simplicity is more important than performance.

Read-heavy workloads: Use `CopyOnWriteArrayList` if your application involves frequent reads and infrequent writes.

High concurrency: Use concurrent collections like `ConcurrentLinkedDeque` for high-concurrency applications requiring efficient multi-threaded access.

Custom synchronization: Use explicit locks like `ReentrantLock` when you need precise control over which operations are synchronized.

Conclusion

In multi-threaded Java applications, ensuring thread-safe access to shared resources like lists is essential to prevent data corruption and race conditions. `Collections.synchronizedList()` offers a straightforward solution by wrapping a list with synchronized access. While it's simple to use, it may not always be the most efficient choice in high-concurrency environments. Alternatives such as `CopyOnWriteArrayList`, concurrent collections, and explicit locks provide varying levels of performance and flexibility, depending on the specific needs of your application. Choosing the right synchronization strategy will help you balance simplicity, safety, and performance effectively.

Java Collections Framework

Collections.synchronizedList() Documentation

Java Concurrency Documentation

CopyOnWriteArrayList Documentation

Thank you for reading! If you find this article helpful, please consider highlighting, clapping, responding or connecting with me on Twitter/X as it's very appreciated and helps keeps content like this free!