Difference between CyclicBarrier and CountDownLatch in Java?

JAVINPAUL

MAR 31, 2024 • PAID

Hello guys, multithreading and concurrency are one of the most important topics for Java interviews and a good knowledge of different concurrent classes and features goes a long way in receiving a good offer.

In the past few posts, I have shared popular Java interview questions like,, How ConcurrentHashMap work in Java? Why String is Immutable, why wait() and notify() is called from synchronized context, and What is the difference between List, List<Object> and today, I am going to share another popular Java Concurrency interview question, **difference between CyclicBarrier and CountDownLatch in Java.**

One of the most important task in concurrent application is to coordinate between threads like when one thread completes is job the next thread start his job and that's where CountDownLatch and CyclicBarrier helps.

While I understand that many Java developer don't get chance to write concurrent code and many of them have not even used wait, notify, CountDownLatch, and CyclicBarrier but you cannot make this excuse on interview. It doesn't goes well with the interviewer and give them a reason to reject you.

Even if you have not used this classes in real application, having good knowledge of what they are and when to use them is critical to answer this question. So let's start with that.

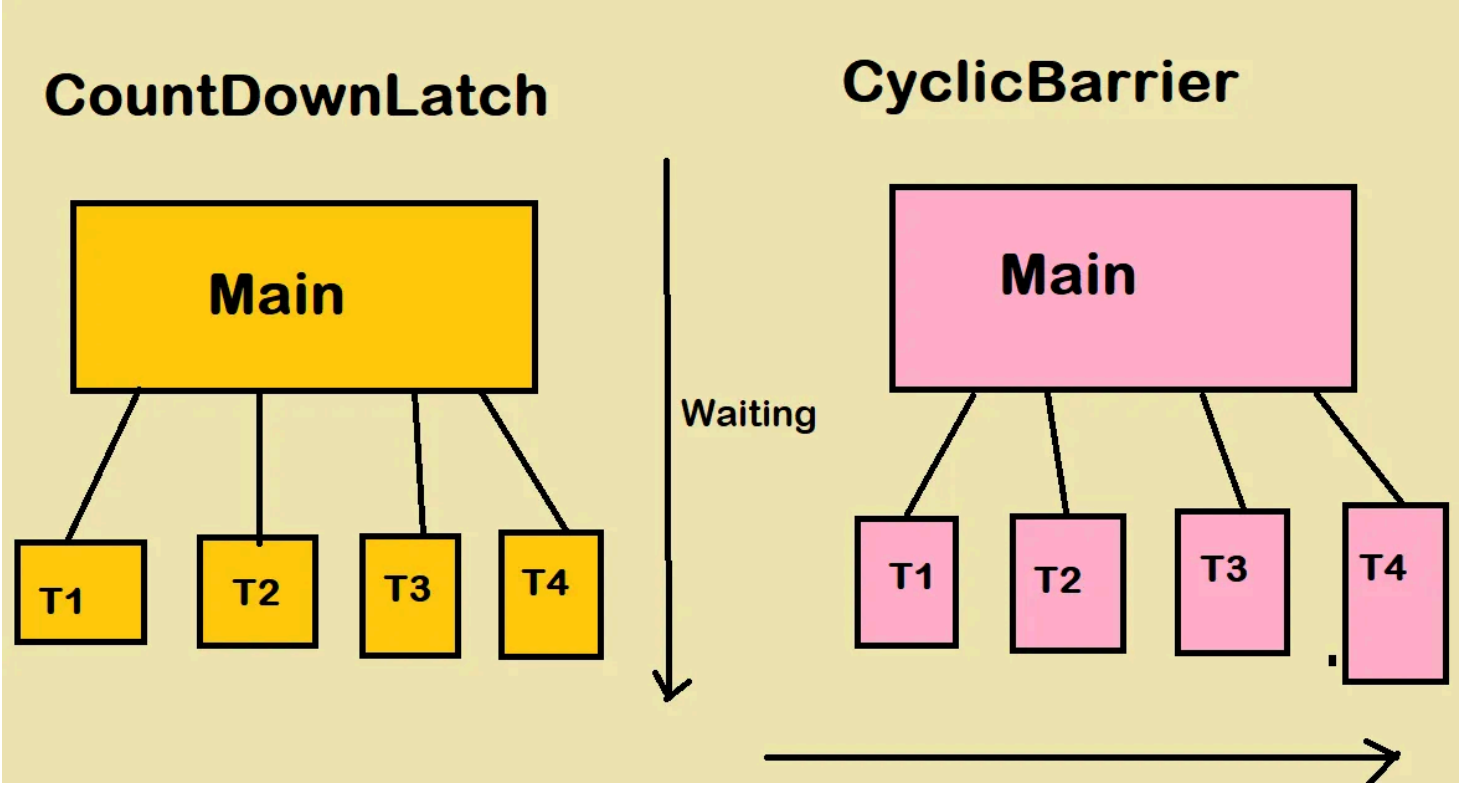
What is difference between `CyclicBarrier` and `CountDownLatch` in Java

`CyclicBarrier` and `CountDownLatch` are both synchronization constructs provided by the Java concurrency utilities (`java.util.concurrent` package) **to coordinate threads in a concurrent application**. While they have similar use cases, they have some fundamental differences in their behavior and usage:

1. **Usage:**

- `CountDownLatch`: It allows **one or more threads to wait until a set of operations being performed in other threads completes**. It is used to synchronize the progress of threads. A `CountDownLatch` is initialized with a count, and threads call `await()` on it to wait until the count reaches zero.
- `CyclicBarrier`: **It allows a fixed number of threads to wait for each other at a barrier point.** It is used to make threads wait for each other until all threads have reached a common execution point. A `CyclicBarrier` is initialized with a count and an optional `Runnable` action to be executed once the barrier is reached. Threads call `await()` on it to wait at the barrier.

2. **Reusability:**
   - `CountDownLatch`: **Once the count reaches zero, the `CountDownLatch` cannot be reset**, so it cannot be reused for another synchronization task.
   - `CyclicBarrier`: **After all threads have reached the barrier point and the barrier is broken, it can be reused for subsequent synchronization tasks.** The count can be reset by calling `reset()`.
3. **Behavior:**
   - `CountDownLatch`: Threads wait for a fixed number of operations to complete before being released. Once the count reaches zero, all waiting threads are released simultaneously.
   - `CyclicBarrier`: Threads wait for each other to reach a common execution point (barrier). Once the required number of threads have reached the barrier, the barrier is broken, and all threads are allowed to continue execution.
4. **Use Cases:**
   - `CountDownLatch`
     This is useful in scenarios where a certain operation cannot proceed until all other operations have completed, such as coordinating the startup of multiple dependent services or waiting for initialization tasks to complete.
     For example, in one of the application I world, we used to load data for multiple caches like product, orders, customers before we can process any request. I have used CountDownLatch to coordinate between different thread loading those caches.
   - `CyclicBarrier`
     This is useful in scenarios where a group of threads need to wait for each other to reach a particular state before proceeding together, such as **parallel computation tasks** where subtasks need to synchronize at certain points.



Real-world example for `CountDownLatch`

Let's see some code and real world scenario where you could use CountDownLatch. Imagine a distributed system where multiple microservices need to be initialized before the system can start processing requests.

Solution with `CountDownLatch`:

```
import java.util.concurrent.CountDownLatch;

public class DistributedSystem {

private static final int NUM_SERVICES = 3;

private static final CountDownLatch latch = new
CountDownLatch(NUM_SERVICES);

public static void main(String[] args) throws InterruptedException {

// Start services

for (int i = 0; i < NUM_SERVICES; i++) {

new Thread(new Service()).start();

}
```

```
// Wait for all services to initialize

latch.await();

System.out.println("All services initialized. System ready.");

}

static class Service implements Runnable {

@Override

public void run() {

// Simulate initialization task

try {

Thread.sleep((long) (Math.random() * 1000));

} catch (InterruptedException e) {

e.printStackTrace();

}

// Signal that this service is initialized

latch.countDown();

System.out.println(Thread.currentThread().getName() + " initialized.");

}

}

}
```

In this example, each service is represented by a thread. The main thread waits for all services to initialize using a `CountDownLatch`. Each service thread decrements the latch count once it completes initialization. When all services are initialized, the main thread proceeds.

Now, let's see a real-world example of CountDownLatch in Java:

Real-world example for `CyclicBarrier`:

Scenario: Suppose you have a parallel computing task where multiple threads perform independent calculations, and you want them to synchronize at certain points to combine their results.

Solution with `CyclicBarrier`:

```
import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class ParallelComputing {

private static final int NUM_THREADS = 4;

private static final CyclicBarrier barrier = new
CyclicBarrier(NUM_THREADS, new MergeResults());

public static void main(String[] args) throws InterruptedException {

// Start computation threads

for (int i = 0; i < NUM_THREADS; i++) {

new Thread(new ComputationTask()).start();

}

}

static class ComputationTask implements Runnable {

@Override

public void run() {

// Perform computation

System.out.println(Thread.currentThread().getName() + " starting
computation");

// Simulate computation

try {

Thread.sleep((long) (Math.random() * 1000));
```

```
} catch (InterruptedException e) {

e.printStackTrace();

}

try {

// Wait for other threads to complete computation

barrier.await();

System.out.println(Thread.currentThread().getName() + " continuing
after synchronization");

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

static class MergeResults implements Runnable {

@Override

public void run() {

// Merge results from all computation threads

System.out.println("Merging results from all computation threads");

}

}

}
```

In this example, each computation task is represented by a thread. The `CyclicBarrier` is used to synchronize all computation threads at the barrier point. After all threads reach the barrier, a `MergeResults` action is executed, and then all threads continue with their computation.

That's all about **difference between CyclicBarrier and CountDownLatch in Java**. In summary, while both `CyclicBarrier` and `CountDownLatch` serve the purpose of synchronizing threads in a concurrent application, they have different semantics and are suited for different synchronization scenarios based on their behavior and reusability.

Let me know how did you find this article

And, All the best for your interviews!!

And, if need help with your interview preparation, you can also check my Java + Spring Interview + SQL Bundle on Gumroad, use code ***friends20*** *to get a 20% discount also*

8 Likes · 2 Restacks

← Previous

Next →

Comments

Write a comment...