Programação_Concorrente-Studies/ Aula08: Count Down e Latch , usando contador para coordenar Threads

- Vamos utilizar a classe CountDownLatch, ela é basicamente serve para o mesmo principio da cyclicBarrier, porém elas tem algumas diferenças, Aqui estão as principais diferenças entre elas:

- **Funcionalidade:**
  - CyclicBarrier: Permite que um grupo de threads aguarde umas às outras em um ponto de sincronização. Quando um thread atinge o ponto de sincronização, ele aguarda até que todos os outros threads do grupo também cheguem lá antes de continuar a execução.
  - CountDownLatch: Permite que uma ou mais threads aguardem até que um conjunto de operações seja concluído. Ele é inicializado com uma contagem de inicialização e decrementa essa contagem à medida que as operações são concluídas. As threads aguardam até que a contagem chegue a zero.
- **Reutilização:**
  - CyclicBarrier: Pode ser reutilizado após a barreira ser alcançada e liberada.
  - CountDownLatch: Não pode ser reutilizado. Uma vez que a contagem chega a zero, ela não pode ser reiniciada.
- **Número de Operações:**
  - CyclicBarrier: Funciona bem quando o número de threads e operações é fixo e conhecido.
  - CountDownLatch: É útil quando o número de operações pode variar e precisa ser especificado somente uma vez.
- **Comportamento:**
  - CyclicBarrier: Todas as threads devem esperar até que todas alcancem o ponto de sincronização. Depois disso, a barreira é liberada e as threads podem continuar.
  - CountDownLatch: As threads podem continuar assim que a contagem chegar a zero, não sendo necessário que todas as threads cheguem a um ponto comum.
- **Uso Típico:**
  - CyclicBarrier: É útil para dividir um problema em subproblemas independentes que podem ser resolvidos em paralelo e, em seguida, aguardar que todas as soluções parciais estejam prontas antes de prosseguir.
  - CountDownLatch: É útil para aguardar a conclusão de várias operações antes de prosseguir, como iniciar vários serviços e aguardar até que todos estejam prontos.

- Essas diferenças refletem a natureza dos problemas que essas classes são projetadas para resolver e ajudam os desenvolvedores a escolher a mais apropriada para suas necessidades específicas.

- A maior diferença é que a count down serve para executarmos certas tarefas com threads e após um certo número de execuções ela pare e realize outra tarefa

- Primeiro declaramos o nosso countDownLatch e determinamos o quanto de execuções da nossa thread deve acontecer ate o countDown chamar a outra tarefa:

```
private static CountDownLatch latch = new
CountDownLatch(count:3);
```

- Já na thread:

```
Runnable r1 = () -> {
    int j = new Random().nextInt(bound:1000);
    int x = i * j;
    System.out.println(i + " x " + j + " = " +
    x);
    latch.countDown();
};
```

- Chamamos o método countDown, então toda vez que ela executar ela adiciona 1 ou countDown até ele chegar no valor que desejamos

- Agora chamamos o método await em nossa execução que fará com que o nosso código espere até o countdown atinja o valor que estipulamos e assim o programa continue.

```
    executor.scheduleAtFixedRate(r1,
    initialDelay:0, period:1, TimeUnit.SEC

    while (true) {
        await();
        i = new Random().nextInt(bound:100);
        latch = new CountDownLatch(count:3);
    }
}
```

- Após o await, trocamos a variável i do nosso programa por uma aleatória e instanciamos novamente o nosso latch , para o código continuar a execução e criar mais um limitador de 3 execuções da minha Thread, e assim tudo se repetindo por conta do while.
- Isso demonstra um dos pontos negativos do countDownLatch, que é o fato dele n ser reutilizável.

- Agora vamos ver outra implementação do countDownLatch:

```java
public class CountDownLatch_2 {

    private static volatile int i = 0;
    private static CountDownLatch latch = new
    CountDownLatch(count:3);

    Run | Debug
    public static void main(String[] args) {
        ScheduledExecutorService executor =
                Executors.newScheduledThreadPool
                (corePoolSize:4);

        Runnable r1 = () -> {
            int j = new Random().nextInt(bound:1000);
            int x = i * j;
            System.out.println(i + " x " + j + " = " +
            x);
            latch.countDown();
        };
        Runnable r2 = () -> {
            await();
            i = new Random().nextInt(bound:100);
        };
        Runnable r3 = () -> {
            await();
            latch = new CountDownLatch(count:3);
        };
```

```
    Runnable r4 = () -> {
        await();
        System.out.println("Terminou! "
                + "Vamos começar de novo! "
                + "Increva-se no canal!");
    };

    executor.scheduleAtFixedRate(r1,
    initialDelay:0, period:1, TimeUnit.SECONDS);
    executor.scheduleWithFixedDelay(r2,
    initialDelay:0, delay:1, TimeUnit.SECONDS);
    executor.scheduleWithFixedDelay(r3,
    initialDelay:0, delay:1, TimeUnit.SECONDS);
    executor.scheduleWithFixedDelay(r4,
    initialDelay:0, delay:1, TimeUnit.SECONDS);

  }

  private static void await() {
    try {
      latch.await();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

}
```

- Aqui utilizamos várias Threads e ambas esperam 3 vezes a execução do countDownLatch para continuar a execução, a primeira faz as 3 contas e a contagem do latch, a segunda espera até a contagem do latch atingir 3 para modificar a minha variável i, a terceira espera o latch atinigir a contagem de 3 e após isso instancia novamente o meu latch para a repetição do código, e a 4

espera a contagem e após a contagem do latch ela imprime uma mensagem no console, dessa forma temos várias Threads, que esperam a execução de uma para prosseguir.

#Concorrent