# Giới thiệu

## Nhằm giúp các bạn có một cái nhìn tổng quan về vấn đề đồng bộ, đa luồng, xử lý song song phục vụ cho môn học Lập trình mạng. Mình xin được chia sẻ bài dịch liên quan vì cảm thấy hay sau khi đã xin phép tác giả.

## Bài viết gốc bạn tham khảo tại đây: <https://www.callicoder.com/java-concurrency-multithreading-basics/>

## Tự giới thiệu của tác giả blog

Tên tôi là Rajeev Kumar Singh. Tôi là người sáng lập, nhà thiết kế cũng như là người chỉnh sửa nội dung cho blog (blog tên CalliCoder – người dịch).

Tôi làm việc tại vị trí kĩ sư phần mềm cho công ty [Directi](https://www.directi.com/), công ty này đã xây dựng nhiều sản phẩm đa nền tảng như [Media.net](https://www.media.net/), [Flock](https://flock.com/), [Radix](https://radix.website/), [Zeta](https://www.zeta.in/), và phi lợi nhuận với [CodeChef](https://www.codechef.com/).

## Về CalliCoder

CalliCoder là blog giáo dục về khoa học máy tính viết về ngôn ngữ lập trình, lập trình web, lập trình ứng dụng desktop và thiết kế hệ thống.

Tôi bắt đầu blog khi tôi muốn chia sẽ về những thứ tôi học được qua từng ngày đi làm cùng với những lập trình viên khác. Tôi hi vọng người khác có thể đọc được nó và cảm thấy nó hữu ích với họ.

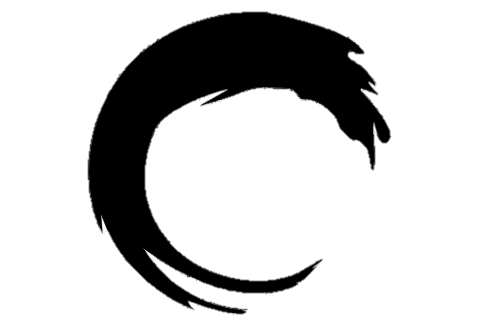
Hơn nữa, blog còn giúp tôi hệ thống lại và cải thiện cái nhìn tổng quát về kiến thức của chính mình vì tôi đã thật sự cố gắng tìm tòi trước khi viết bất kì nội dung nào.

## Callicoder nghĩa là gì?

Bất kì ai nghe về tên callicoder đều hỏi tôi về ý nghĩa của nó?

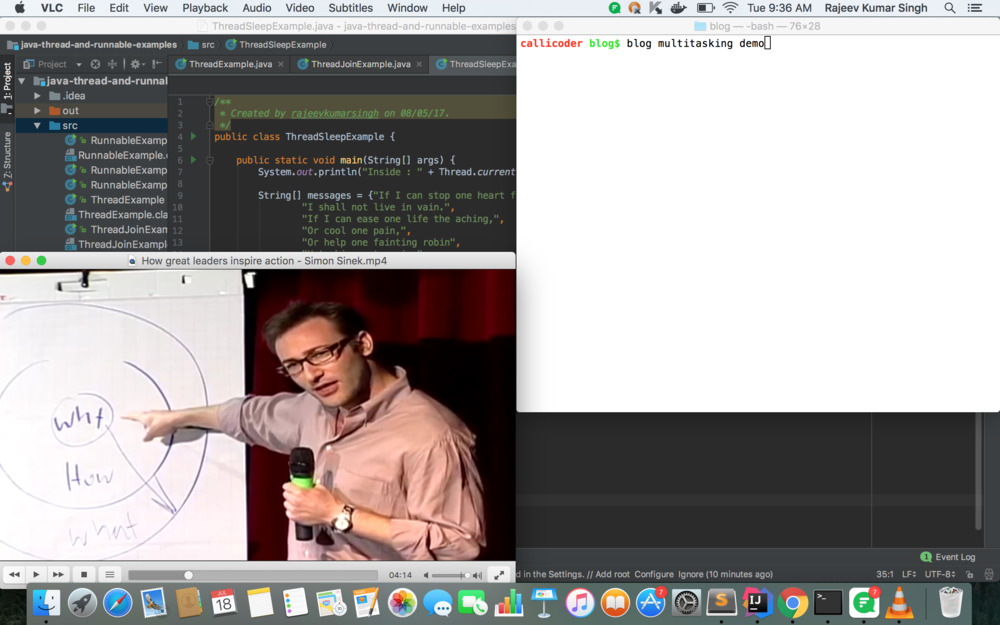
Thật ra thì, Callicoder có nguyên bản là từ **Calligraphy**. **Calli** trong Calligraphy có nghĩa là đẹp. Vì vậy, từ callicoder là có ý ám chỉ đến việc viết code đẹp (hay thậm chí là một coder đẹp :p).

Logo của blog lấy ý tưởng một kí hiệu trong Zen (Thiền).



**Trong Thiền tông, ensō là một vòng tròn được vẽ bằng tay trong một hoặc hai nét vẽ không ngắt quãng để biểu hiện khoảnh khắc khi tâm trí được tự do để cho cơ thể được sáng tạo.** (Trích từ [*wikipedia*](https://vi.wikipedia.org/wiki/Ens%C5%8D))

Kí hiệu trên được thiết kế bởi [Nick Bluth](https://thenounproject.com/nickbluth/) trong [noun project](https://thenounproject.com/). Tôi đã chỉnh sửa lại để được logo blog.

**Java Concurrency / Multithreading Basics**

*Concurrency là khả năng làm nhiều hơn một việc trong một thời điểm.*

Trước đây, máy tính chỉ có thể thực thi một chương trình vào một thời điểm. Nhưng hiện nay, những chiếc máy tính hiện đại có thể làm việc với nhiều chương trình một lúc. Ví dụ -

* Bạn có thể vừa lướt web vừa nghe nhạc từ trình phát nhạc trên máy tính, *cùng lúc.*
* Bạn có thể đang sửa văn bản trong MS Word, trong khi đó một ứng dụng khác đang tải file từ internet về máy, *cùng lúc.*

Concurrency không nhất thiết phải là sự có mặt của nhiều ứng dụng. Chạy nhiều phần của một ứng dụng cũng là một cách thể hiện của concurrency. Ví dụ -

* Bộ xử lý của Word vừa định dạng văn bản vừa lắng người dùng nhập từ bàn phím, *cùng lúc*.
* Một trình nghe nhạc trực tuyến download nhạc từ mạng, giải nén nó và hiển thị nó ra dưới dạng âm thanh, *cùng lúc*.
* Web server, một ứng dụng trên một máy chủ nào đó, có thể giao tiếp với hàng ngàn máy trạm (client), *cùng lúc*.

Phần mềm có thể thực hiện nhiều việc cùng lúc như vậy gọi là *concurrent software.*

Ảnh chụp màn hình của tôi phía đầu bài viết cũng là một ví dụ cho vấn đề. Nó vừa phát video, vừa nhận lệnh từ bàn phím vào cửa sổ terminal, vừa build một chương trình trong IntelliJ.

**Concurrency: Chi tiết hơn**

Được! Tôi hiểu là máy tính có thể làm được nhiều việc cùng lúc, nhưng nó làm như thế nào?

Tôi cũng biết là ngày nay máy tính có khái niệm đa xử lý (multiple processors), nhưng như vậy có nghĩa những con chip đơn không thể có tính năng concurrency? Hơn nữa, một máy tính có khả năng xử lý nhiều tác vụ hơn số lõi nó có cơ mà.

Nhiều tác vụ thực hiện cùng lúc trên một CPU như thế nào?

Vâng! Rõ ràng rằng, không thực sự có một xử lý cùng lúc nào trong một đơn vị thời gian vật lý. Concurrency cũng không làm việc song song.

Khi chúng ta nói - “nhiều việc đang làm cùng lúc”, thì có nghĩa rằng “nhiều việc đang được diễn ra trong một khoảng thời gian.”

Chỉ duy nhất một tác vụ được thực hiện tại một thời điểm. Chính hệ điều hành chuyển qua lại một cách thường xuyên giữa các tác vụ làm cho người dùng cảm giác rằng các việc này diễn ra cùng lúc.

Vì vậy, *Concurrency* không có nghĩa là *Parallelism* (Song song). Sự thật thì, Parallelism là không thể trên những CPU đơn nhân.

**Đơn vị của Concurrency**

Concurrency là một vấn đề lớn, được sử dụng rộng rãi trên nhiều cấp độ. Ví dụ -

* **Multiprocessing –** Nhiều Processor/CPU thực hiện đồng thời. Đơn vị của concurrency ở đây là CPU.
* **Multitasking –** Nhiều task/process (tác vụ/tiến trình) cùng chạy trên một CPU. Hệ điều hành thực hiện chúng bằng cách chuyển qua lại rất thường xuyên. Đơn vị của concurrency trong trường hợp này là Process.
* **Multithreading –** Nhiều phần của một chương trình chạy đồng thời. Trong trường hợp này, chúng ta tiến xa hơn và chia một process thành nhiều part/thread (phần/luồng) và chạy chúng đồng thời.

**Process và Thread**

Hãy cùng nói về hai đơn vị cơ bản của concurrency : Process (Tiến trình) và Threads (Luồng).

**Process**

Process là một chương trình đang thực thi. Nó có địa chỉ không gian, tập lệnh thực thi và liên kết đến các tài nguyên như là các file được mở.

Một máy tính bình thường có nhiều process chạy cùng lúc. Hệ điều hành giữ thông tin về chúng và giúp chúng thực hiện được bằng cách chia sẻ processing time (thời gian thực thi) của CPU.

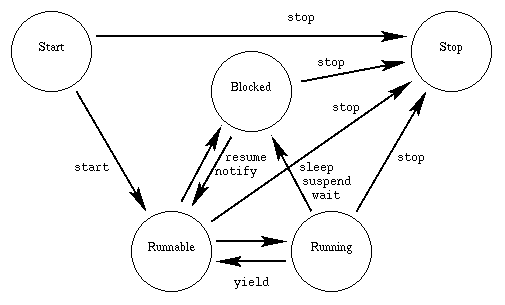
**Thread**

Thread là một phần thực thi của một process. Mỗi process có tối thiểu một thread – được gọi là main thread. Main thread có thể tạo ra thêm những thread trong process.

Những thread trong cùng process chia sẻ cho nhau các tài nguyên như bộ nhớ và file đang mở. Tuy nhiên, mỗi thread có một tệp lệnh thực thi khác nhau.

Cũng chính vì các thread chia sẻ cho nhau địa chỉ không gian của process, việc tạo thread mới và giao tiếp giữa chúng trở nên dễ dàng hơn.

Các trạng thái của một thread.



Dead

**Vấn đề Thường gặp với Concurrency**

Concurrency cải thiện một cách đáng kể hiệu năng của máy tính bằng cách tăng khả năng CPU. Nhưng hiệu suất tăng cũng mang lại vài vấn đề -

* **Lỗi Thread interference (Race Conditions) :** Lỗi Thread interference xảy ra khi nhiều thread cố gắng đọc và ghi một biến được dùng chung vào cùng thời điểm, việc đọc và ghi này có thể chồng lên nhau.

Trường hợp này xảy ra thì kết quả phụ thuộc vào việc cái nào được đọc/ghi trước. Lỗi này không thể lường trước nên khó phát hiện và sửa.

Lỗi Thread interference có thể được ngăn ngừa bằng cách chỉ cho phép một thread truy cập vào biến một thời điểm. Cách này thường được triển khai theo cách khóa tài nguyên trước khi truy cập.

Nhưng cách trên cũng dẫn đến vài vấn đề khác như **deadlock** và **starvation**. Chúng ta sẽ tìm hiểu thêm trong những bài sau.

* **Lỗi Memory consistency:** Lỗi Memory consistency xảy ra khi nhiều thread “nhìn” một dữ liệu không giống nhau. Chẳng hạn, khi một thread cập nhật một biến, những thread khác không biết và vẫn dùng dữ liệu cũ của biến đó trong chương trình của mình.

# Java Thread và RunnableJava Thread and Runnable Tutorial

Đây là phần hai trong series Java Concurrency. Trong phần trước, ta đã học cơ bản về concurrency, process và thread. Trong bài này, chúng ta sẽ học cách để tạo những thread mới và thực thi task (tác vụ) trong những thread đó.

## Khởi tạo và chạy một Thread

Có hai cách để tạo một thread trong Java -

### 1. Bằng cách kế thừa lớp Thread

Bạn có thể tạo một thread đơn giản bằng cách kế thừa class Thread từ class của bạn và override lại phương thức run()của nó.

Phương thức run() chứa code thực thi bên trong thread mới. Mỗi khi một thread được tạo ra, bạn có thể chạy nó bằng cách gọi hàm start().

public class ThreadExample extends Thread {

// run() method contains the code that is executed by the thread.

@Override

public void run() {

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

}

public static void main(String[] args) {

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

System.out.println("Creating thread...");

Thread thread = new ThreadExample();

System.out.println("Starting thread...");

thread.start();

}

}

# Output

Inside : main

Creating thread...

Starting thread...

Inside : Thread-0

Thread.currentThread() trả về thread đang được thực thi hiện tại. Trong ví dụ trên, tôi sử dụng hàm getName() để in ra tên của thread hiện tại.

Mỗi thread đều có một tên. Bạn có thể tạo một thread với một tên tùy chọn bằng cách sử dụng constructor Thread(String name). Nếu không có tên nào được chỉ định thì một cái tên sẽ tự động được chọn cho thread.

### 2. Bằng cách cung cấp một đối tượng Runnable

Interface Runnable là mẫu nguyên thủy cho bất kì đối tượng nào cần được thực thi bởi một thread. Nó định nghĩa một hàm run() chứa code được thực thi bởi thread.

Bất cứ class nào muốn được thực thi bởi thread thì phải implement interface Runnable.

**Thread vốn đã implement interface Runnable với hàm run() rỗng.**

Để tạo một thread mới, ta tạo một thể hiện của class (tức là object) đã implement interface Runnable và sau đó truyền đối tượng đó vào constructor Thread(Runnable target).

public class RunnableExample implements Runnable {

public static void main(String[] args) {

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

System.out.println("Creating Runnable...");

Runnable runnable = new RunnableExample();

System.out.println("Creating Thread...");

Thread thread = new Thread(runnable);

System.out.println("Starting Thread...");

thread.start();

}

@Override

public void run() {

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

}

}

# Output

Inside : main

Creating Runnable...

Creating Thread...

Starting Thread...

Inside : Thread-0

Nên nhớ rằng, thay vì tạo một lớp implements interface Runnable và sau đó khởi tạo lớp đó để có đối tượng runnable, thì bạn có thể tạo một Runnable ẩn danh bằng cách dùng cú pháp [lớp ẩn danh](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html) của Java.

Các lớp ẩn danh cho phép bạn làm cho code ngắn gọn hơn. Chúng cho phép bạn khai báo class và khởi tạo đối tượng của class đó cùng một thời điểm. - Trích từ Java doc.

public class RunnableExampleAnonymousClass {

public static void main(String[] args) {

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

System.out.println("Creating Runnable...");

Runnable runnable = new Runnable() {

@Override

public void run() {

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

}

};

System.out.println("Creating Thread...");

Thread thread = new Thread(runnable);

System.out.println("Starting Thread...");

thread.start();

}

}

Ví dụ trên còn có thể ngắn gọn hơn nữa khi sử dụng [biểu thức lambda](https://www.callicoder.com/java-lambda-expression-tutorial/) của Java 8 -

public class RunnableExampleLambdaExpression {

public static void main(String[] args) {

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

System.out.println("Creating Runnable...");

Runnable runnable = () -> {

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

};

System.out.println("Creating Thread...");

Thread thread = new Thread(runnable);

System.out.println("Starting Thread...");

thread.start();

}

}

## Runnable hay Thread, dùng cái nào?

Cách đầu tiên là kế thừa từ class Thread thì rất hạn chế vì mỗi lớp chỉ có thể kế thừa một lớp mà thôi, bạn không thể kế thừa thêm class khác vì Java không hỗ trợ đa kế thừa.

Ngoài ra, nếu theo nghĩa trong thiết kế, tính kế thừa có nghĩa là cho phép mở rộng thuộc tính, hàm của class cha. Nhưng khi bạn tạo một thread, bạn không mở rộng lớp Thread, bạn chỉ đơn giản là hiện thực hóa phương thức run().

Nói chung, bạn nên luôn sử dụng đối tượng Runnable để tạo một thread. Cách này linh hoạt hơn. Nó cho phép class của bạn kế thừa từ bất kì class nào khác. Ngoài ra bạn cũng có thể dùng cú pháp của class ẩn danh và [biểu thức lambda](https://www.callicoder.com/java-lambda-expression-tutorial/) vào Runnable để làm cho code của bạn gọn gàng hơn.

## Tạm dừng thực thi một thread bằng sleep()

Hàm sleep() được cung cấp bởi class Thread cho phép bạn tạm dừng thực thi thread đang chạy trong một khoảng thời gian được chỉ định (đơn vị milli second).

public class ThreadSleepExample {

public static void main(String[] args) {

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

String[] messages = {"If I can stop one heart from breaking,",

"I shall not live in vain.",

"If I can ease one life the aching,",

"Or cool one pain,",

"Or help one fainting robin",

"Unto his nest again,",

"I shall not live in vain"};

Runnable runnable = () -> {

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

for(String message: messages) {

System.out.println(message);

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

}

};

Thread thread = new Thread(runnable);

thread.start();

}

}

# Output

Inside : main

Inside : Thread-0

If I can stop one heart from breaking,

I shall not live in vain.

If I can ease one life the aching,

Or cool one pain,

Or help one fainting robin

Unto his nest again,

I shall not live in vain

Ví dụ trên bao gồm một vòng lặp for duyệt tuần tự trên một mảng các message (tin nhắn), in ra message hiện tại, chờ trong 2 giây bằng cách gọi hàm Thread.sleep(), và sau đó tiếp tục vòng lặp.

Phương thức sleep() quăng ngoại lệ InterruptedException khi có bất kì thread nào interrupts thread hiện tại. InterruptedException là một ngoại lệ kiểm tra và cần được kiểm soát.

## Chờ một thread khác hoàn thành bằng cách dùng join()

Hàm join() cho phép một thread chờ một thread khác hoàn thành trong một khoảng thời gian. Trong ví dụ dưới đây, Thread 2 chờ Thread 1 hoàn thành trong 1000 milli giây (1 giây) bằng cách gọi hàm Thread.join(1000), và sau đó mới bắt đầu thực thi -

public class ThreadJoinExample {

public static void main(String[] args) {

// Create Thread 1

Thread thread1 = new Thread(() -> {

System.out.println("Entered Thread 1");

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("Exiting Thread 1");

});

// Create Thread 2

Thread thread2 = new Thread(() -> {

System.out.println("Entered Thread 2");

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("Exiting Thread 2");

});

System.out.println("Starting Thread 1");

thread1.start();

System.out.println("Waiting for Thread 1 to complete");

try {

thread1.join(1000);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("Waited enough! Starting Thread 2 now");

thread2.start();

}

}

# Output

Starting Thread 1

Waiting for Thread 1 to complete

Entered Thread 1

Waited enough! Starting Thread 2 now

Entered Thread 2

Exiting Thread 1

Exiting Thread 2

Thời gian chờ của Thread.join() bằng giá trị nhỏ hơn khi so sánh giữa thời gian để thread kết thúc với số milli giây được chỉ định trong tham số của phương thức.

Hàm join() cũng có thể được gọi mà không có tham số. Trong trường hợp này, nó sẽ chờ cho đến khi thread “chết” (thực thi xong hoặc bị tắt).

# Java ExecutorService và Thread PoolsJava ExecutorService and Thread Pools Tutorial

Chào mừng đến phần ba của series Java Concurrency. Trong phần này, chúng ta cùng học về cách quản lý thread trong ứng dụng của mình sử dụng Executors và Thread Pools.

## Executors Framework

Trong phần trước, ta đã học cách tạo một thread trong Java bằng cách kế thừa lớp Thread hoặc implement interface Runnable.

Để tạo một hoặc hai thread thì vô cùng dễ dàng, cho đến khi ứng dụng của chúng ta có vài chục thread đang chạy cùng lúc.

Cũng như vậy, không có gì là nói quá khi một chương trình lớn có hàng trăm, thậm chí hàng nghìn thread. Do đó, việc tạo một thread và quản lý nó ở những thời điểm tiếp theo của chương trình là cần thiết.

**Đến với Executors, một framework cho phép tạo và quản lý thread. Executors framework sẽ giúp bạn -**

1. **Thread Creation**: Cung cấp một số phương thức giúp bạn tạo thread, một pool (hồ) của thread – cái mà chương trình của bạn có thể lấy ra bất kì thread nào có sẵn trong đó để chạy tác vụ đồng thời.
2. **Thread Management**: Quản lý vòng đời của các thread trong thread pool. Bạn không cần bận tâm rằng một thread đang sẵn sang, đang chạy hay đã chết cho đến khi bạn submit nó.
3. **Task submission and execution**: Executors framework hỗ trợ những phương thức để submit tác vụ cho việc thực thi trong thread pool, và cũng cung cấp cho bạn khả năng quyết định khi nào tác vụ được thực thi. Ví dụ, Bạn có thể submit một task để nó thực thi ngay hoặc đặt lịch cho nó thực thi sau hoặc thực thi một cách định kì.

[Java Concurrency API](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) định nghĩa ra ba interface hỗ trợ mọi thứ về tạo và quản lý thread -

* **Executor** – Một interface đơn giản chỉ chứa phương thức execute() để chạy một đối tượng Runnable.
* **ExecutorService** – Một interface con của Executor đã thêm vài phương thức để quản lý vòng đời của thread. Nó có một phương thức tên submit() có thể chấp nhận cả Runnable hoặc Callable. Callable là gần giống với Runnable ngoại trừ việc đối tượng Callable có thể trả về giá trị. Chúng ta sẽ tìm hiểu ở chương sau.
* **ScheduledExecutorService** - Một interface con của ExecutorService. Nó thêm phương thức để đặt lịch thực thi cho tác vụ.

Ngoài ba interface trên, API còn cung cấp lớp [Executors](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html) chứa phương thức theo pattern factory để tạo các Executor Service khác nhau.

## Ví dụ ExecutorService

Ổn rồi! Chúng ta cùng đến với ví dụ để hiểu rõ hơn. Trong ví dụ này, chúng ta sẽ tạo ra ExecutorService với một thread đơn, và sau đó chúng ta submit một tác vụ cho nó.

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorsExample {

public static void main(String[] args) {

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

System.out.println("Creating Executor Service...");

ExecutorService executorService = Executors.newSingleThreadExecutor();

System.out.println("Creating a Runnable...");

Runnable runnable = () -> {

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

};

System.out.println("Submit the task specified by the runnable to the executor service.");

executorService.submit(runnable);

}

}

# Output

Inside : main

Creating Executor Service...

Creating a Runnable...

Submit the task specified by the runnable to the executor service.

Inside : pool-1-thread-1

Ví dụ trên đã chỉ cách tạo một executor service và thực thi một tác vụ bên trong executor. Chúng ta dùng phương thức Executors.newSingleThreadExecutor()để tạo một ExecutorService với một thread đơn. Nếu một task được submit vào để thực thi trong khi thread của ExecutorService lại đang thực hiện một task khác thì task mới đó phải chờ đến khi thread sẵn sang thực thi trở lại.

Nếu bạn để ý, chương trình trên chưa hề kết thúc, đó là vì, executor service vẫn tiếp tục ở đó, chờ các task mới để thực thi. Để tắt thì chúng ta cần chỉ ra một cách rõ ràng.

## Tắt ExecutorService

ExecutorService hỗ trợ hai phương thức để tắt executor -

* **shutdown()** – khi phương thức shutdown() của executor service được gọi, service này ngừng nhận submit mới, chờ cho tác vụ đang tiến hành thực thi đến khi xong, và sau đó tắt executor.
* **shutdownNow()** – phương thức này interrupt tác vụ đang thực hiện và tắt executor ngay lập tức.

Hãy thêm hai dòng code sau đây vào chương trình trên để tắt executor -

System.out.println("Shutting down the executor");

executorService.shutdown();

## Ví dụ ExecutorService với nhiều thread và nhiều task (tác vụ)

Trong ví dụ trên, chúng ta đã tạo một ExecutorService với một thread duy nhất. Nhưng sự thật là sức mạnh của ExecutorService thể hiện qua việc ta có thể tạo một pool của thread và thực thi nhiều task đồng thời trong cái pool đó.

Ví dụ sau cho ta cách tạo một ExecutorService với nhiều thread và chạy nhiều task -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsExample {

public static void main(String[] args) {

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

System.out.println("Creating Executor Service with a thread pool of Size 2");

ExecutorService executorService = Executors.newFixedThreadPool(2);

Runnable task1 = () -> {

System.out.println("Executing Task1 inside : " + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException ex) {

throw new IllegalStateException(ex);

}

};

Runnable task2 = () -> {

System.out.println("Executing Task2 inside : " + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(4);

} catch (InterruptedException ex) {

throw new IllegalStateException(ex);

}

};

Runnable task3 = () -> {

System.out.println("Executing Task3 inside : " + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(3);

} catch (InterruptedException ex) {

throw new IllegalStateException(ex);

}

};

System.out.println("Submitting the tasks for execution...");

executorService.submit(task1);

executorService.submit(task2);

executorService.submit(task3);

executorService.shutdown();

}

}

# Output

Inside : main

Creating Executor Service with a thread pool of Size 2

Submitting the tasks for execution...

Executing Task2 inside : pool-1-thread-2

Executing Task1 inside : pool-1-thread-1

Executing Task3 inside : pool-1-thread-1

Trong ví dụ trên chúng ta đã tạo một executor service với kích cỡ thread pool cố định là 2. Thread pool có cỡ cố định là một đang rất hay gặp trong ứng dụng đa luồng.

Trong thread pool cỡ cố định, executor service sẽ đảm bảo rằng luôn có một số lượng được chỉ định các thread đang chạy và sẵn sàng nhận task mới. Nếu vì bất kì lý do nào một trong số chúng “chết”, nó sẽ được thay thế bởi cái mới.

Khi một task được submit, executor service sẽ chọn một thread bất kì đang sẵn sàng (còn “sống”, không đang chạy task nào khác) từ pool và tiến hành thực thi. Nếu chúng ta submit số task nhiều hơn số lượng thread sẵn sàng thì những task mới dư ra này phải chờ đến lượt mình để được thực thi.

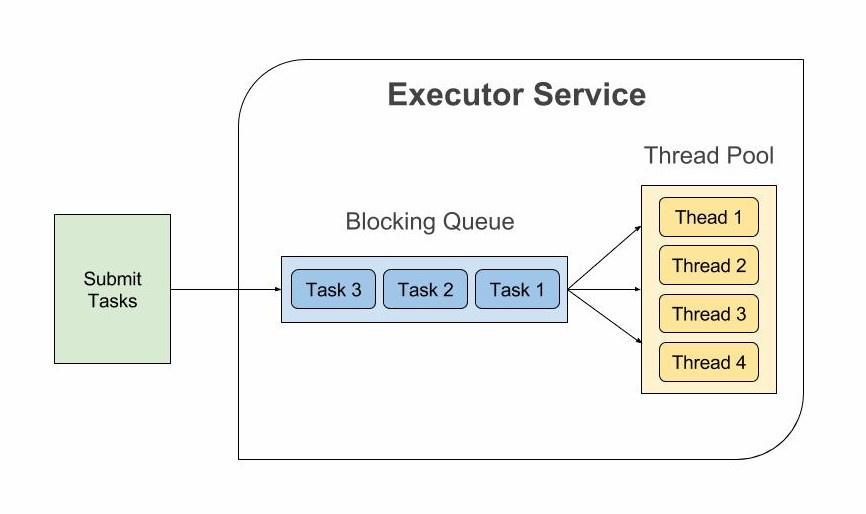
## Thread Pool

Hầu hết các hiện thực của executor sử dụng *thread pool* để thực thi task. Một thread pool ichẳng qua là một nhóm các thread tồn tại riêng biệt với Runnable hoặc Callable và được quản lý bởi executor.

Tạo một thread là việc làm tốn tài nguyên cho nên nó cần được tối ưu hóa. Có các thread được tạo duy nhất một lần và sử dụng lại sau đó để thực hiện các task là cách để tối ưu.

Chúng ta đã lấy một ví dụ trong phần trước để tạo ra một fixed thread-pool (thread pool kích cỡ cố định).

Những tác vụ được submit vào thread pool thông qua một hàng chờ nội bộ tên **Blocking Queue**. Nếu có nhiều task hơn số lượng thread sẵn sàng thì chúng sẽ được cho vào blocking queue và chờ cho đến khi có thread sẵn sàng. Nếu blocking queue đầy thì task mới sẽ bị từ chối.



## Ví dụ ScheduledExecutorService

ScheduledExecutorService được dùng để thực hiện task một cách chu kì hoặc sau một khoảng thời gian.

Trong ví dụ sau, ta sẽ lên lịch cho tác vụ thực hiện sau 5 giây -

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsExample {

public static void main(String[] args) {

ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);

Runnable task = () -> {

System.out.println("Executing Task At " + System.nanoTime());

};

System.out.println("Submitting task at " + System.nanoTime() + " to be executed after 5 seconds.");

scheduledExecutorService.schedule(task, 5, TimeUnit.SECONDS);

scheduledExecutorService.shutdown();

}

}

# Output

Submitting task at 2909896838099 to be executed after 5 seconds.

Executing Task At 2914898174612

Phương thức scheduledExecutorService.schedule() nhận một Runnable, một giá trị trễ, và đơn vị thời gian. Chương trình trên thực hiện sau khoảng thời gian trễ là 5 giây kể từ thời điểm submit.

Bây giờ ta đến với ví dụ thực hiện task một cách chu kỳ -

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsPeriodicExample {

public static void main(String[] args) {

ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);

Runnable task = () -> {

System.out.println("Executing Task At " + System.nanoTime());

};

System.out.println("scheduling task to be executed every 2 seconds with an initial delay of 0 seconds");

scheduledExecutorService.scheduleAtFixedRate(task, 0, 2, TimeUnit.SECONDS);

}

}

# Output

scheduling task to be executed every 2 seconds with an initial delay of 0 seconds

Executing Task At 2996678636683

Executing Task At 2998680789041

Executing Task At 3000679706326

Executing Task At 3002679224212

...

scheduledExecutorService.scheduleAtFixedRate() nhận một Runnable, một thời gian trễ lúc đầu, khoảng thời gian lặp lại, và đơn vị thời gian. Nó sẽ bắt đầu chạy chương trình sau một khoảng thời gian trễ lúc đầu, lặp lại nó sau mỗi khoảng thời gian lặp.

Lưu ý rằng khi một tác vụ xảy ra exception, sẽ không có bất kì lần lặp nào nữa. Nếu không, task sẽ chỉ kết thúc khi bạn tắt nó hoặc đóng chương trình.

# Java Callable và FutureJava Callable and Future Tutorial

Chào mừng bạn đến với phần thứ tư trong series Java Concurrency. Trong những phần trước, chúng ta đã tìm hiểu vềcơ bản của concurrency, thread, runnable và executor service.Trong phần này, ta sẽ học về Callable và Future.

**Callable**

Trong bài trước, chúng ta dùng một đối tượng Runnable để tạo một tác vụ được thực thi bằng thread. Mặc dù tạo ra Runnable khá dễ dàng nhưng có một hạn chế là nó không thể trả về một giá trị.

**Chuyện gì nếu bạn muốn một kết quả từ một task?**

Well, Java hỗ trợ một interface tên Callable để làm điều này. Callable giống với Runnable ngoại trừ việc nó có thể trả về một giá trị (nguyên thủy hoặc không) và throw exception (hàm run() thì không thể).

Interface Callable có một phương thức call() duy nhất chứa code sẽ được thực thi bởi thread. Sau đây là ví dụ đơn giản về Callable -

Callable<String> callable = new Callable<String>() {

@Override

public String call() throws Exception {

// Perform some computation

Thread.sleep(2000);

return "Return some result";

}

};

Ghi nhớ rằng với Callable, không nhất thiết bạn phải để Thread.sleep() trong try/catch, vì không giống như Runnable, Callable có thể throw exception.

Bạn cũng có thể dùng [biểu thức lambda](https://www.callicoder.com/java-lambda-expression-tutorial/) để rút gọn code -

Callable<String> callable = () -> {

// Perform some computation

Thread.sleep(2000);

return "Return some result";

};

**Thực thi Callable dùng ExecutorService và nhận kết quả dùng Future**

Giống Runnable, bạn có thể submit Callable vào một executor service để thực thi. Nhưng còn kết quả của Callable thì sao? Làm sao lấy ra được mà dùng?

Phương thức submit() của executor service sẽ submit task vào thread để chạy. Nhưng mà, nó không thể biết được khi nào kết quả của task đó sẵn sàng cả. (*Thật thế, bình thường chương trình của ta chạy một mạch vào phương thức nào đó rồi trả về kết quả, nhưng phương thức* submit() *chỉ nói rằng sẽ thực thi phương thức đã được submit vào một lúc nào đó trong tương lai, không ai biết bao giờ có kết quả.* – người dịch). Do đó, nó trả về một kiểu dữ liệu đặc biệt tên là Future chứa kết quả của một task khi nó sẵn sàng.

Khái niệm về Future giống với [Promise](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise) trong những ngôn ngữ khác như Javascript. Nó thể hiện kết quả tính toán, cái mà sẽ được hoàn thành trong tương lai.

Sau đây là ví dụ về Future và Callable -

import java.util.concurrent.\*;

public class FutureAndCallableExample {

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

ExecutorService executorService = Executors.newSingleThreadExecutor();

Callable<String> callable = () -> {

// Perform some computation

System.out.println("Entered Callable");

Thread.sleep(2000);

return "Hello from Callable";

};

System.out.println("Submitting Callable");

Future<String> future = executorService.submit(callable);

// This line executes immediately

System.out.println("Do something else while callable is getting executed");

System.out.println("Retrieve the result of the future");

// Future.get() blocks until the result is available

String result = future.get();

System.out.println(result);

executorService.shutdown();

}

}

# Output

Submitting Callable

Do something else while callable is getting executed

Retrieve the result of the future

Entered Callable

Hello from Callable

Phương thức ExecutorService.submit() ngay lập tức trả về cho ta một Future. Kể từ lúc bạn có Future, bạn có thể làm những việc khác trong khi chờ cho task được thực thi, sau đó dùng future.get() để lấy ra kết quả.

Cần nhớ rằng, phương thức get() khi được gọi mà chưa có sẵn kết quả, thread gọi get() sẽ bị dừng lại chờ đến khi có kết quả. Future có cung cấp phương thức isDone() để kiểm tra xem task đã xong (kết quả để get() sẵn sàng) chưa -

import java.util.concurrent.\*;

public class FutureIsDoneExample {

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

ExecutorService executorService = Executors.newSingleThreadExecutor();

Future<String> future = executorService.submit(() -> {

Thread.sleep(2000);

return "Hello from Callable";

});

while(!future.isDone()) {

System.out.println("Task is still not done...");

Thread.sleep(200);

}

}

System.out.println("Task completed! Retrieving the result");

String result = future.get();

System.out.println(result);

executorService.shutdown();

}

}

# Output

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task completed! Retrieving the result

Hello from Callable

**Hủy Future**

Bạn có thể dùng Future.cancel() để hủy một Future. Phương thức này sẽ cố gắng hủy task tạo ra future, trả về true nếu hủy được task, các trường hợp khác trả về false.

Phương thức cancel() cũng có thể nhận vào một tham số kiểu boolean - mayInterruptIfRunning. Nếu nó là true thì thread đang thực thi task sẽ bị interrupt, nếu không task đang được xử lý sẽ được chờ đến khi hoàn tất.

Chúng ta có thể dùng hàm isCancelled() để kiểm tra xem một Future có bị hủy chưa. Sau khi bị hủy thì hàm isDone() luôn true.

import java.util.concurrent.\*;

public class FutureCancelExample {

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

ExecutorService executorService = Executors.newSingleThreadExecutor();

long startTime = System.nanoTime();

Future<String> future = executorService.submit(() -> {

Thread.sleep(2000);

return "Hello from Callable";

});

while(!future.isDone()) {

System.out.println("Task is still not done...");

Thread.sleep(200);

double elapsedTimeInSec = (System.nanoTime() - startTime)/1000000000.0;

if(elapsedTimeInSec > 1) {

future.cancel(true);

}

}

System.out.println("Task completed! Retrieving the result");

String result = future.get();

System.out.println(result);

executorService.shutdown();

}

}

# Output

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task is still not done...

Task completed! Retrieving the result

Exception in thread "main" java.util.concurrent.CancellationException

at java.util.concurrent.FutureTask.report(FutureTask.java:121)

at java.util.concurrent.FutureTask.get(FutureTask.java:192)

at FutureCancelExample.main(FutureCancelExample.java:34)

Chạy chương trình trên bạn sẽ thấy future.get() quăng ra một CancellationException vì task đã bị hủy. Chúng ta có thể xử lý bằng cách kiểm tra nó có bị hủy chưa trước khi lấy giá trị -

if(!future.isCancelled()) {

System.out.println("Task completed! Retrieving the result");

String result = future.get();

System.out.println(result);

} else {

System.out.println("Task was cancelled");

}

**Thêm Timeout (Thời gian chờ)**

Như bạn biết, phương thức future.get() sẽ khóa cho đến khi task hoàn thành. Giả sử bạn gọi hàm này của một máy chủ, máy chủ sập, và phương thức bị khóa vĩnh viễn, chương trình của bạn sẽ rơi vào trạng thái “không phản hồi”.

Để hạn chế điều này bạn cần thêm timeout vào hàm get() -

future.get(1, TimeUnit.SECONDS);

Lúc này phương thức future.get() sẽ cho ra một TimeoutException nếu sau thời gian chỉ định task vẫn chưa hoàn thành.

**invokeAll**

*Submit nhiều task và chờ đến khi tất cả bọn chúng hoàn thành.*

Bạn có thể thực thi nhiều task bằng cách cho một collection các Callable vô hàm invokeAll(). invokeAll() sẽ trả về một List các Future. Hàm future.get() của bất kì future nào sẽ bị khóa cho đến khi tất cả Future sẵn sàng.

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.\*;

public class InvokeAllExample {

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

ExecutorService executorService = Executors.newFixedThreadPool(5);

Callable<String> task1 = () -> {

Thread.sleep(2000);

return "Result of Task1";

};

Callable<String> task2 = () -> {

Thread.sleep(1000);

return "Result of Task2";

};

Callable<String> task3 = () -> {

Thread.sleep(5000);

return "Result of Task3";

};

List<Callable<String>> taskList = Arrays.asList(task1, task2, task3);

List<Future<String>> futures = executorService.invokeAll(taskList);

for(Future<String> future: futures) {

// The result is printed only after all the futures are complete. (i.e. after 5 seconds)

System.out.println(future.get());

}

executorService.shutdown();

}

}

# Output

Result of Task1

Result of Task2

Result of Task3

Trong chương trình trên, câu lệnh future.get() lần đầu thực thi sẽ bị khóa cho đến khi tất cả future sẵn sàng. v.d. các kết quả sẽ được in sau 5 giây.

**invokeAny**

*Submit nhiều task và chờ đến khi một trong số chúng hoàn thành.*

Phương thức invokeAny() nhận một collection các Callable và trả về giá trị của Callable hoàn thành sớm nhất. Lưu ý rằng, giá trị trả về không phải một Future.

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.\*;

public class InvokeAnyExample {

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

ExecutorService executorService = Executors.newFixedThreadPool(5);

Callable<String> task1 = () -> {

Thread.sleep(2000);

return "Result of Task1";

};

Callable<String> task2 = () -> {

Thread.sleep(1000);

return "Result of Task2";

};

Callable<String> task3 = () -> {

Thread.sleep(5000);

return "Result of Task3";

};

// Returns the result of the fastest callable. (task2 in this case)

String result = executorService.invokeAny(Arrays.asList(task1, task2, task3));

System.out.println(result);

executorService.shutdown();

}

}

# Output

Result of Task2

# Java Concurrency issues và Thread SynchronizationJava Concurrency issues and Thread Synchronization

Chào mừng đến phần năm của series Java Concurrency. Trong phần này chúng ta sẽ tìm hiểu các hỗi hay gặp phải trong một chương trình concurrent/multithread, và cách ngăn chặn chúng.

## Vấn đề trong Concurrency

Đa luồng (multithreading) là một công cụ mạnh mẽ cho phép ta nâng cao hiệu suất của hệ thống, nhưng đổi lại ta cũng cần lưu ý khi đọc và ghi những dữ liệu dùng chung giữa các thread.

Hai vấn đề gặp phải là -

1. Lỗi Thread interference (Race conditions)
2. Lỗi Memory consistency

Giờ ta cùng tìm hiểu từng cái một.

### Lỗi Thread Interference (Race Conditions)

Class Counter chứa phương thức increment() tăng biến count lên 1 đơn vị mỗi khi nó được gọi -

class Counter {

int count = 0;

public void increment() {

count = count + 1;

}

public int getCount() {

return count;

}

}

Bây giờ, giả sử ta có vài thread cố gắng tăng giá trị biến count bằng cách gọi hàm increment() -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class RaceConditionExample {

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

ExecutorService executorService = Executors.newFixedThreadPool(10);

Counter counter = new Counter();

for(int i = 0; i < 1000; i++) {

executorService.submit(() -> counter.increment());

}

executorService.shutdown();

executorService.awaitTermination(60, TimeUnit.SECONDS);

System.out.println("Final count is : " + counter.getCount());

}

}

Đoán xem kết quả chương trình trên là bao nhiêu? Biến count sẽ tăng lên 1000 vì hàm increment() được gọi 1000 lần?

Không, câu trả lời là không! Bạn hãy chạy chương trình để thấy kết quả. Thay vì là 1000, mỗi lần chạy lại ra một kết quả khác nhau. Tôi chạy ba lần và được kết quả 992, 996 và 993.

**Cùng xem xét rõ hơn để hiểu tại sao lại có sự phi lý như vậy -**

Khi một thread thực thi phương thức increment() sẽ thực hiện ba bước: 1. Lấy dữ liệu biến count; 2. Tăng giá trị lấy được lên 1; 3. Lưu giá trị đã tăng trở lại biến count

Bây giờ giả sử ta có hai thread - ThreadA và ThreadB, thực thi những việc sau đây -

1. **ThreadA** : Lấy dữ liệu biến count, lúc này giá trị là = 0
2. **ThreadB** : Lấy dữ liệu biến count, lúc này giá trị là = 0
3. **ThreadA** : Tăng giá trị lên 1, kết quả = 1
4. **ThreadB** : Tăng giá trị lên 1, kết quả = 1
5. **ThreadA** : Lưu lại giá trị đã tăng, biến count bây giờ = 1
6. **ThreadB** : Lưu lại giá trị đã tăng, biến count bây giờ = 1

Cả hai thread đều tăng giá trị count lên 1, nhưng kết quả vẫn là 1 thay cho 2 vì hai thread làm việc không quan tâm nhau. Trong trường hợp trên, sự cập nhật của ThreadA đã bị mất và bị ghi đè bởi ThreadB.

Trình tự các bước như trên chỉ là một trong số nhiều cách sắp xếp khiến cho kết quả dữ liệu của chúng ta không nhất quán.

Nhiều thread cùng đọc và ghi một biến (biến này gọi là gọi là shared variable) đồng thời, và những việc này chồng lặp lên nhau, thì kết quả sẽ dựa trên thứ tự của các hành động ghi đọc ấy, một chuyện không thể kiểm soát. Thuật ngữ cho việc này là [Race condition](https://en.wikipedia.org/wiki/Race_condition).

Phần (Section) code mà shared variable được truy cập thì gọi là [Critical Section](https://en.wikipedia.org/wiki/Critical_section).

Lỗi Thread interference có thể được ngăn chặn bằng cách đồng bộ sự truy cập vào shared variables. Ta sẽ hiện thực cách này bằng Java code trong phần sau.

Giờ chúng ta tiếp tục tìm hiểu dạng lỗi thứ hai hai gặp trong multithreading – Lỗi Memory Consistency.

### Lỗi Memory Consistency

Lỗi Memory inconsistency xảy ra khi nhiều thread có những cái nhìn khác nhau về (giá trị) một biến. Việc này xảy ra khi một thread cập nhật giá trị một biến, trong khi đó sự cập nhật này không được các thread khác, do vậy chúng vẫn xài giá trị cũ.

**Tại sao chuyện như thế xảy ra được?** Oh, có nhiều lý do cho việc này. Trình biên dịch (compiler) đã thực hiện vài động tác tối ưu hóa để nâng cao hiệu suất của bạn. Có thể nó đã sắp xếp lại trình tự các công việc. Bộ vi xử lý (processer) cũng cố gắng nâng cao hiệu suất theo cách của nó: đọc biến từ register (nơi chứa giá trị được đọc lần cuối từ bộ nhớ chính), thay vì từ bộ nhớ chính (thường là RAM – nơi chứa giá trị mới nhất của biến).

Cùng theo dõi ví dụ sau gây lỗi Memory Consistency -

public class MemoryConsistencyErrorExample {

private static boolean sayHello = false;

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

Thread threadA = new Thread(() -> {

while(!sayHello) {

}

System.out.println("Hello World!");

while(sayHello) {

}

System.out.println("Good Bye!");

});

threadA.start();

Thread.sleep(1000);

System.out.println("Say Hello..");

sayHello = true;

Thread.sleep(1000);

System.out.println("Say Bye..");

sayHello = false;

}

}

Theo ý tưởng thì chương trình sẽ -

1. Chờ một giây và in ra Hello World! sau khi sayHello trở thành true.
2. Chờ một giây nữa và in ra Good Bye! sau khi sayHello trở thành false.

# Ideal Output

Say Hello..

Hello World!

Say Bye..

Good Bye!

Nhưng có thật chương trình sẽ chạy ra như vậy? Haizz, nếu bạn chạy thử bạn sẽ thấy chương trình in ra như thế này -

# Actual Output

Say Hello..

Say Bye..

Đồng thời, chương trình vẫn còn chạy.

Chờ một chút. What? Sao có thể?

Haha! Đó chính là cái mà ta gọi là Lỗi Memory Consistency. ThreadA không biến đến sự thay đổi của biến sayHello trong main thread.

Bạn hãy dùng từ khóa volatile để tránh lỗi này. Chúng ta sẽ học về từ khóa trên ngay sau đây.

## Synchronization (Sự đồng bộ)

Lỗi Thread interference và memory consistency có thể tránh được khi ta đảm bảo hai chuyện -

1. Chỉ một biến được đọc và ghi một shared variable vào một thời điểm. Khi một thread đang truy cập một shared variable, những thread khác phải chờ. Điều này nhằm đảm bảo việc truy cập một shared variable là Atomic (đơn nhất), các thread không được gây nhiễu loạn cho nhau.
2. Bất kể khi nào có thread thay đổi một shared variable, nó sẽ khởi động một mối quan hệ happens-before đối với các thread khác có hành vi đọc/ghi biến này để thread đó cập nhật giá trị mới trước khi sử dụng. Việc này đảm bảo sự thay đổi biến của thread này được thread khác biết đến.

May mắn hay, Java hỗ trợ ta từ khóa synchronized để giúp đồng bộ sự truy cập đến bất kỳ tài nguyên nào, ngăn ngừa cả hai lỗi trên.

#### ****Synchronized Methods (Phương thức được Synchronize)****

Phiên bản Synchronized của lớp Counter. Chúng ta dùng từ khóa synchronized của Java cho phương thức increment() để ngăn nhiều thread truy cập đồng thời -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

class SynchronizedCounter {

private int count = 0;

// Synchronized Method

public synchronized void increment() {

count = count + 1;

}

public int getCount() {

return count;

}

}

public class SynchronizedMethodExample {

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

ExecutorService executorService = Executors.newFixedThreadPool(10);

SynchronizedCounter synchronizedCounter = new SynchronizedCounter();

for(int i = 0; i < 1000; i++) {

executorService.submit(() -> synchronizedCounter.increment());

}

executorService.shutdown();

executorService.awaitTermination(60, TimeUnit.SECONDS);

System.out.println("Final count is : " + synchronizedCounter.getCount());

}

}

Nếu chạy chương trình trên ta được kết quả kì vọng là 1000. Không có race conditions xảy ra và kết quả là bất biến. **Từ khóa synchronized đảm bảo rằng chỉ một thread thực thi phương thức increment() tại một thời điểm.**

Ghhi nhớ rằng khái niệm Synchronization luôn liên quan đến cùng đối tượng (object). Trong ví dụ trên, nhiều lần gọi hàm increment() của một đối tượng SynchonizedCounter có thể dẫn đến race condition. Và ta phải ngăn chặn bằng cách dùng từ khóa synchronized. Tuy nhiên nhiều thread có thể gọi hàm increment() trên nhiều object khác nhau của SynchronizedCounter cùng một lúc mà chẳng gây ra race condition.

Trường hợp phương thức là static, synchronization liên quan đến đối tượng của lớp [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html).

#### ****Synchronized Blocks (Khối được Synchronize)****

Bản thân Java dùng một cái có tên là intrinsic lock hay monitor lock để quản lý sự đồng bộ các thread. Mỗi object có một intrinsic lock.

Khi một thread gọi một synchronized method của một object, nó sẽ tự động yêu cầu intrinsic lock của đối tượng đó và trả lại khi kết thúc phương thức. Việc trả lại này diễn ra kể cả khi phương thức throw (quăng ra) một ngoại lệ (exception).

Trường hợp phương thức static, không giống như phương thức thường, thread yêu cầu intrinsic lock của một object của lớp Class mà object này liên quan đến lớp chứa phương thức static.

Từ khóa synchronized cũng được dùng như một khai báo block, nhưng không giống synchronized method, synchronized block phải chỉ định đối tượng nào sẽ được yêu cầu intrinsic lock -

public void increment() {

// Synchronized Block -

// Acquire Lock

synchronized (this) {

count = count + 1;

}

// Release Lock

}

Khi một thread yêu cầu intrinsic lock của một object, những thread khá phải chờ đến khi intrinsic lock được trao trả. Tuy nhiên, thread đang sở hữu lock có thể yêu cầu nó nhiều lần mà chả sao cả.

Ý tưởng của việc một thread có thể yêu cầu lock nhiều lần gọi là Reentrant Synchronization.

## Từ khóa Volatile

Từ khóa Volatile được dùng để ngăn chặn lỗi memory. Nó nói với trình biên dịch đừng tối ưu hóa gì biến này cả.

Như vậy, giá trị của biến luôn được chương trình lấy từ bộ nhớ chính thay vì register.

Sau đây là ví dụ MemoryConsistencyError giống như cái trước đây chúng ta đã xem ngoại trừ việc khai báo biến sayHello với từ khóa volatile.

public class VolatileKeywordExample {

private static volatile boolean sayHello = false;

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

Thread thread = new Thread(() -> {

while(!sayHello) {

}

System.out.println("Hello World!");

while(sayHello) {

}

System.out.println("Good Bye!");

});

thread.start();

Thread.sleep(1000);

System.out.println("Say Hello..");

sayHello = true;

Thread.sleep(1000);

System.out.println("Say Bye..");

sayHello = false;

}

}

Chạy chương trình trên và nhận được kết quả như kì vọng -

# Output

Say Hello..

Hello World!

Say Bye..

Good Bye!

# Java Locks và Atomic Variables Java Locks and Atomic Variables Tutorial

Trong chương trình đa luồng, một shared variables cần được synchronize để ngăn chặn race conditions.

Trong phần trước, chúng ta đã học cách dùng synchronized methods và synchronized blocks để bảo đảm sự truy xuất đến shared variable nhằm ngăn chặn race conditions.

Từ khóa synchronized trong Java bản thân nó sử dụng một intrinsic lock liên quan đến object đặc quyền truy cập đến các thành phần của object.

Thay vì dùng intrinsic lock thông qua từ khóa synchronized, bạn có thể dùng vài kiểu lock khác nhau được cung cấp bởi Java trong những class Lock để kiểm soát tốt hơn bằng các cơ chế khóa.

Trong phần này, chúng ta sẽ tìm hiểu cách sử dụng những class Lock được hỗ trợ bởi Java này để đồng bộ sự truy cập đến những shared variable.

Cuối cùng, chúng ta sẽ xem xét một cách mới của đồng bộ thread thông qua các kiểu lớp Atomic hỗ trợ bởi Java.

## Locks

### 1. ReentrantLock

ReentrantLock là một lock hành xử như intrinsic/implicit lock thông qua từ khóa synchronized.

ReentrantLock, như tên gọi, có khả năng tái xâm nhập. Nghĩa là, một thread hiện đang giữ lock có thể yêu cầu lock một lần nữa mà không xảy ra vấn đề gì.

Chương trình sau hướng dẫn sử dụng ReentrantLock-

import java.util.concurrent.locks.ReentrantLock;

class ReentrantLockCounter {

private final ReentrantLock lock = new ReentrantLock();

private int count = 0;

// Thread Safe Increment

public void increment() {

lock.lock();

try {

count = count + 1;

} finally {

lock.unlock();

}

}

}

Ý tưởng vô cùng đơn giản – Bất kì thread nào gọi hàm increment() sẽ yêu cầu lock rồi tăng biến count. Khi nó làm xong thì sẽ trao trả lock và thread đang chờ có thể yêu cầu lock.

Lưu ý đến đoạn try/finally trong ví dụ trên. Khối finally để đảm bảo lock sẽ được trả lại kể cả khi có ngoại lệ xảy ra.

ReentrantLock cũng hỗ trợ những phương thức khác để hỗ trợ kiểm soát dễ hơn -

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.locks.ReentrantLock;

class ReentrantLockMethodsCounter {

private final ReentrantLock lock = new ReentrantLock();

private int count = 0;

public int incrementAndGet() {

// Check if the lock is currently acquired by any thread

System.out.println("IsLocked : " + lock.isLocked());

// Check if the lock is acquired by the current thread itself.

System.out.println("IsHeldByCurrentThread : " + lock.isHeldByCurrentThread());

// Try to acquire the lock

boolean isAcquired = lock.tryLock();

System.out.println("Lock Acquired : " + isAcquired + "\n");

if(isAcquired) {

try {

Thread.sleep(2000);

count = count + 1;

} catch (InterruptedException e) {

throw new IllegalStateException(e);

} finally {

lock.unlock();

}

}

return count;

}

}

public class ReentrantLockMethodsExample {

public static void main(String[] args) {

ExecutorService executorService = Executors.newFixedThreadPool(2);

ReentrantLockMethodsCounter lockMethodsCounter = new ReentrantLockMethodsCounter();

executorService.submit(() -> {

System.out.println("IncrementCount (First Thread) : " +

lockMethodsCounter.incrementAndGet() + "\n");

});

executorService.submit(() -> {

System.out.println("IncrementCount (Second Thread) : " +

lockMethodsCounter.incrementAndGet() + "\n");

});

executorService.shutdown();

}

}

# Output

IsLocked : false

IsHeldByCurrentThread : false

Lock Acquired : true

IsLocked : true

IsHeldByCurrentThread : false

Lock Acquired : false

IncrementCount (Second Thread) : 0

IncrementCount (First Thread) : 1

Phương thức tryLock() sẽ yêu cầu khóa mà không dừng chương trình lại. Nghĩa là nếu một thread yêu cầu khóa nhưng không được (thread khác đang giữ), thì thread tiếp tục ngay lập tức chứ không chờ đến khi lock được trao trả.

Bạn cũng có thể chỉ định một khoảng thời gian cho tryLock() để nó chờ khóa trong khoảng thời gian đó -

lock.tryLock(1, TimeUnit.SECONDS);

Bây giờ thread sẽ dừng lại 1 giây. Sau 1 giây đó, có nhận được khóa hay không chương trình vẫn chạy tiếp.

### 2. ReadWriteLock

ReadWriteLock chứa một cặp khóa – một cho quyền đọc và một cho quyền ghi. Khóa đọc có thể được giữ bởi nhiều thread cùng lúc với điều kiện khóa ghi không bị chiếm bởi thread nào.

ReadWriteLock chấp nhận một mức cao hơn của đồng bộ hóa. Nó tốt hơn khi quan niệm rằng có nhiều hành vi đọc hơn là ghi.

import java.util.concurrent.locks.ReadWriteLock;

import java.util.concurrent.locks.ReentrantReadWriteLock;

class ReadWriteCounter {

ReadWriteLock lock = new ReentrantReadWriteLock();

private int count = 0;

public int incrementAndGetCount() {

lock.writeLock().lock();

try {

count = count + 1;

return count;

} finally {

lock.writeLock().unlock();

}

}

public int getCount() {

lock.readLock().lock();

try {

return count;

} finally {

lock.readLock().unlock();

}

}

}

Trong ví dụ trên, nhiều thread có thể thực thi hàm getCount() chỉ khi hàm incrementAndGetCount() không được thực thi. Nếu bất kì thread nào gọi hàm incrementAndGetCount() và yêu cầu khóa ghi, thì tất cả thread đọc sẽ tạm dừng chờ đến ghi hàm ghi hoàn tất.

## Biến Atomic

Java có định nghĩa ra vài class trong package [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) hỗ trợ những thao tác Atomic (đơn nhất) trên biến.

Những class Atomic sử dụng cơ chế [compare-and-swap](https://en.wikipedia.org/wiki/Compare-and-swap) được hỗ trợ bởi những CPU hiện đại để thực hiện việc đồng bộ hóa. Cơ chế này thì nhanh hơn lock.

Cùng xem cách dùng lớp AtomicInteger để đảm bảo tăng biến count là atomic (đơn nhất).

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {

private AtomicInteger count = new AtomicInteger(0);

public int incrementAndGet() {

return count.incrementAndGet();

}

public int getCount() {

return count.get();

}

}

public class AtomicIntegerExample {

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

ExecutorService executorService = Executors.newFixedThreadPool(2);

AtomicCounter atomicCounter = new AtomicCounter();

for(int i = 0; i < 1000; i++) {

executorService.submit(() -> atomicCounter.incrementAndGet());

}

executorService.shutdown();

executorService.awaitTermination(60, TimeUnit.SECONDS);

System.out.println("Final Count is : " + atomicCounter.getCount());

}

}

# Output

Final Count is : 1000

Phương thức AtomicInteger.incrementAndGet() là atomic, vì vậy bạn có thể gọi nó trong nhiều thread đồng thời mà vẫn đảm bảo tính đồng bộ của nó.

Vài class khác được định nghĩa trong package [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) -

* [AtomicBoolean](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html)
* [AtomicLong](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html)
* [AtomicReference](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html)

Bạn có thể dùng những class Atomic này thay vì dùng từ khóa synchronized hay lock bởi vì nó nhanh hơn, dễ dùng hơn, dễ đọc hơn và dễ thay đổi hơn.

## Tổng kết

Chúc mừng bạn đã hoàn thành series Java concurency. Trong phần này, chúng ta đã học cách để sử dụng Lock và biến Atomic cho việc đồng bộ hóa. Bạn có thể tải các source code trong bài học ở [github repository](https://github.com/callicoder/java-concurrency-examples).

# Bài tập vui

Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

The average value is 82

The minimum value is 72

The maximum value is 95

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)