**Tìm hiểu về Thread Pool trong java**

# 1. Mở đầu

Thread được sinh ra để thực hiện một nhiệm vụ cụ thể, nhiều Thread cùng xử lý công việc giúp chúng ta giải quyết được bài toán thời gian và hiệu năng khi xử lý một tác vụ nào đó.

Bài toán đặt ra ở đây là có phải cứ sinh ra nhiều Thread thì tác vụ của chúng ta sẽ nhanh hơn mượt mà hơn?

Câu trả lời là không phải cứ tạo nhiều Thread cùng hoạt động thì sẽ đem lại hiệu năng cao vì mỗi khi có một Thread mới được tạo ra và được cấp phát bộ nhớ bằng từ hóa new thì sẽ có vấn đề bộ nhớ và hiệu suất -> có thể dẫn tới crash chương trình.

* Để giải quyết bài toán đó **ThreadPool** ra đời để giới hạn số lượng Thread được chạy bên trong ứng dụng chúng ta cùng một thời điểm.

**Ví dụ:** Khi chúng ta viết chương trình tải các tập tin từ Internet, mỗi tập tin cần 1 Thread để thực hiện quá trình tải, giả sử cần tải 10,000 tệp âm thanh thì chúng ta phải cần tới 10,000 Thread hoạt động cùng một thời điểm trong cùng một chương trình. Điều này sẽ dễ dẫn đến lỗi quá tải của chương trình, làm ảnh hưởng đến bộ nhớ và hiệu suất của chương trình sẽ rất dễ dẫn đến bị crash vì khó kiểm soát.

Vì vậy, để khắc phục hiện tượng này, Java cho phép chúng ta thay vì phải tạo mới Thread cho mỗi nhiệm vụ, quá trình được thực hiện trong cùng một thời điểm thì các nhiệm vụ, quá trình đó có thể được đưa vào trong một ThreadPool để khi trong ThreadPool có bất kỳ Thread nào đang không phải thực hiện một nhiệm vụ nào thì sẽ có nhiệm vụ gán vào một trong số các Thread đó để thực thi. Điều này sẽ giúp khắc phục được sự tắc nghẽn và chương trình sẽ kiểm soát được các luồng thực thi.

Bên trong **ThreadPool**, các nhiệm vụ sẽ được chèn vào trong một Blocking Queue. Blocking Queue có thể hiểu là nơi chứa các nhiệm vụ mà các Thread sẽ lấy chúng ra và thực thi lần lượt. Mỗi khi có một nhiệm vụ mới được thêm vào Queue và sau đó sẽ chỉ có một Thread đang không phải thực hiện một nhiệm vụ nào vào Queue lấy nhiệm vụ đó ra, còn các Thread còn lại phải chờ đợi cho đến khi Thread đó lấy nhiệm vụ ra thành công.

Nhưng cũng thật là may, bắt đầu từ\*\* Java 5\*\*, chúng ta đã được cung cấp một thư viên “Executor framework” trong gói java.util.concurrent giúp cho lập trình viên tạo và quản lý các “ThreadPool” và “Thread Factories” đơn giản hơn bao giờ hết. Java cung cấp cho chúng ta lớp Executor, sub-interface của nó là **ExecutorService** và lớp **ThreadPoolExecutor** kế thừa từ interface ExecutorService trên.

# 2. ThreadPoolExecutor

Diagram

Description automatically generated

Ở đây chúng ta có thể thấy rằng các đối tượng **ThreadPool Executor** chấp nhận Runnable và đặt nó vào một **Runnable Queue**. Hàng đợi này đại diện cho tất cả các **nhiệm vụ** (Các Task) được gửi để được thực thi bởi Threadpool. Bản thân **ThreadPool** nó là một chuỗi các luồng đang chờ để kéo Runnables ra khỏi hàng đợi và thực hiện chúng theo các phương thức run() riêng của chúng . Khi ThreadPool running, hay nói cách khác, các luồng trong ThreadPool vẫn còn sống và sẵn sàng thực thi runnables. Khi có một Runnable mới trong hàng đợi, một trong các luồng sẽ kéo nó ra và gọi phương thức run() của Runnable.

## 2.1. Ví dụ

Ví dụ dưới đây sẽ minh họa cách tạo ThreadPool bằng cách sử dụng ThreadPoolExecutor:

**Run.java**

public class Run implements Runnable{

int id;

public Run(int id) {

this.id = id;

}

@Override

public void run() {

System.out.println("Tiến trình đang được thực thi " + id);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Tiến trình đã được thực thi" + id);

}

}

**TestThreadPool.java**

public class TestThreadPool {

public static void main(String[] args) {

int corePoolSize = 5;

int maximumPoolSize = 10;

long keepAliveTime = 500;

TimeUnit unit = TimeUnit.SECONDS;

ArrayBlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(100);

RejectedExecutionHandler handler = new ThreadPoolExecutor.CallerRunsPolicy();

ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(corePoolSize,

maximumPoolSize, keepAliveTime, unit, workQueue, handler);

for (int i = 0; i < 10; i++) {

threadPoolExecutor.execute(new Run(i));

}

}

}

## 2.2. Cách hoạt động của ThreadPoolExecutor

Giải thích hoạt động của chương trình trên

Trong dòng code khởi tạo ThreadPoolExecutor:

ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(corePoolSize, maxPoolSize, keepAlive, unit, workQueue, handler);

chúng ta có 6 tham số:

* Đối số 1: (corePoolSize) là số lượng Thread tối thiểu trong ThreadPool. Khi khởi tạo, số lượng Thread có thể là 0. Khi nhiệm vụ được thêm vào thì Thread mới được tạo ra và kể từ đây, nếu số lượng Thread ít hơn corePoolSize thì những Thread mới sẽ được tạo ra đến khi số Thread bằng giá trị của corePoolSize.
* Đối số 2: (maximumPoolSize) là số lượng tối đa các Thread trong ThreadPool. Điều kiện corePoolSize < maximumPoolSize
* Đối số 3: (keepAliveTime): khi số Thread lớn hơn corePoolSize thì keepAliveTime là thời gian tối đa mà 1 Thread "nhàn rỗi" chờ nhiệm vụ. Khi hết thời gian chờ mà Thread đó chưa có nhiệm vụ thì nó sẽ bị hủy.
* Đối số 4: (unit) là đơn vị thời gian của keepAliveTime. Trong ví dụ này thì unit của tôi là TimeUnit.SECONDS.
* Đối số 5: (workQueue) là hàng đợi dùng để chứa các nhiệm vụ mà các Thread sẽ lấy chúng ra và thực thi lần lượt, ở đây tôi dùng ArrayBlockingQueue.
* Đối số 6: (handler): Hành động khi một request (task) bị từ chối (rejected)

Ở đây bạn có thể thấy mình sử dụng ThreadPoolExecutor.CallerRunsPolicy, đây là một trong số các error handler rất hay của ThreadPoolExecutor, chúng ta hãy cùng tìm hiểu: **CallerRunsPolicy**: Đây là một error handler được gọi đến khi một task vì lý do gì đó bị từ chối (có thể là do queue trong Thread pool đã đầy …), nó sẽ được chạy lại bởi một thread mới khác (khi ThreadPool có một thread đang rảnh rỗi), trừ khi executor đã bị shutdown, hoặc task đó đã bị hủy. Có nghĩa là khi sử dụng CallerRunsPolicy bạn sẽ không bao giờ sợ chương trình lỡ một một task được gửi đi mà không được xử lý. Ngoài ra ta cũng có thể dùng các “policy” cho RejectedExecutionHandler như sau:

* ThreadPoolExecutor.AbortPolicy: Khi một task bị từ chối chương trình sẽ throw ra một runtime RejectedExecutionException.
* ThreadPoolExecutor.DiscardPolicy: Khi một task bị từ chối nó đơn gian là sẽ bị “bỏ qua” (discard), lỗi lầm gì đó cũng sẽ không bị throw ra.
* ThreadPoolExecutor.DiscardOldestPolicy: Khi một task bị từ chối, chương trình sẽ hủy task “cũ nhất” (oldest) trong queue mà chưa được sử lý, sau đó gửi task vừa bị từ chối vô queue và cố gắng sử lý lại task đó.

# 3. ExecutorService

Kể từ Java 5 trở đi, ThreadPool đã được xây dựng sẵn trong gói java.util.concurrent, vì vậy chúng ta không cần phải tạo một **ThreadPool** mà thay vào đó chúng ta sẽ sử dụng các lớp có sẵn của gói này. Java cung cấp cho chúng ta lớp Executor, interface của lớp Executor là **ExecutorService**. Interface ExecutorService đại diện cho cơ chế thực thi bất đồng bộ có khả năng thực thi các nhiệm vụ trong background. ExecutorService là một đối tượng chịu trách nhiệm quản lý các luồng và thực hiện các tác vụ Runnable được yêu cầu xử lý. Nó tách riêng các chi tiết của việc tạo Thread, lập kế hoạch (scheduling), … để chúng ta có thể tập trung phát triển logic của tác vụ mà không quan tâm đến các chi tiết quản lý Thread.

A picture containing diagram

Description automatically generated

## 3.1. Ví dụ

**Run.java**

public class Run implements Runnable{

int id;

public Run(int id) {

this.id = id;

}

@Override

public void run() {

System.out.println("Tiến trình đang được thực thi " + id);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Tiến trình đã được thực thi" + id);

}

}

**TestThreadPool.java**

public class TestThreadPool {

public static void main(String[] args) {

ExecutorService pool = Executors.newFixedThreadPool(5);

for (int i = 0; i < 10; i++) {

pool.submit(new Run(i));

}

try {

pool.awaitTermination(1, TimeUnit.DAYS);

} catch (InterruptedException e) {

e.printStackTrace();

}

pool.shutdown();

}

}

## 3.2. Cách tạo ExecutorService

Ta có thể tạo các thread pool thông qua ExecutorService, các “tác vụ” (task) sẽ gửi vào pool và sẽ được sử lý bằng một trong những phương thức mà Executor cung cấp như sau:

* Single Thread Executor : Trong ThreadPool chỉ có 1 Thread và các task sẽ được sử lý một cách tuần tự. Tên method “newSingleThreadExecutor()”
* Cached ThreadPool : Trong ThreadPool sẽ có rất nhiều Thread, và các task sẽ được sử lý một cách song song. Các Thread cũ sau khi sử lý xong sẽ được sử dụng lại cho tác vụ mới. Mặc định nếu một Thread không được sử dụng trong vào 60 giây thì Thread đó sẽ được hủy (shut down). Tên method “newCachedThreadPool()”
* Fixed Thread Pool : Trong ThreadPool sẽ được cố định (fixed) số lượng các Thread. Nếu một task mới được đưa vào mà các thread đều đang “bận rộn” thì task đó sẽ được gửi vào Blocking Queue và ngay sau khi một Thread đã thực thi xong nhiệm vụ của nó thì nhiệm vụ đang ở trong Queue đó sẽ được push ra khỏi Queue và được Thread đó xử lý tiếp. Method “newFixedThreadPool()”
* Scheduled Thread Pool : Tương tự như “Cached Thread Pool” nhưng sẽ có khoảng delay giữa các Thread. Method “newScheduledThreadPool()”
* Single Thread Scheduled Pool : Tương tự như “Single Thread Executor” nhưng sẽ có khoảng delay giữa các Thread. Method “newSingleThreadScheduledExecutor()”

## 3.3. Cách sử dụng ExecutorService

Có một vài cách khác nhau để giao nhiệm vụ tới một ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...) Sau đây là cách sử dụng các phương thức trên và tác dụng của chúng:
* execute(Runnable) Phươngthức execute(Runnable) đưa vào một đối tượng java.lang.Runnable và thực thi chúng bất đồng bộ. Với việc sử dụng phương thức này không có cách nào để thu được kết quả của việc thực hiện Runnable (k có callback hoặc giá trị trả về khi thực hiện xong nhiệm vụ).

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

* submit(Runnable) Phương thức submit(Runnable) cũng đưa vào 1 Runnable nhưng nó trả về một đối tượng Future. Đối tượng Future có thể được sử dụng để kiểm tra nếu Runnable đã hoàn tất việc thực thi.

Future<T> future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

}, T.class);

future.get(); // Trả về đối tượng T mà bạn truyền vào, dựa vào đây xác đây nhiệm đã hòan tất

* submit(Callable) Phương thức submit(Callable) tương tự như submit(Runnable) ngoại trừ việc hàm call() của nó cần 1 giá trị trả về để xác định kết quả thu được sau khi hòan thành nhiệm vụ còn phương thức Runnable.run()không thể trả lại kết quả. Kết quả của Callable có thể thu được thông qua đối tượng Future được trả về bởi phương thức submit(Callable)

Future<T> future = executorService.submit(new Callable(){

public Object call() throws Exception {

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

return T;

}

});

System.out.println("future.get() = " + future.get()); // Return T object (callable result)

Sử dụng phương thức future.get() để thu được kết quả. Chú ý phương thực này được thực thi đồng bộ (Asynchronous - tức là sau khi callable hòan thành nhiệm vụ kết quả được trả về nó mới được thực thi).

* invokeAny(Collection<?> extends Callable<T> tasks) Phương thức invokeAny() nhận một tập hợp (collection) của các đối tượng Callable hoặc các lớp được kế thừa từ Callable chúng. Gọi phương thức này không trả về một Future, nhưng trả về kết quả của một trong những đối tượng Callables. Bạn không đảm bảo về kết quả nào bạn sẽ nhận được từ callable. Chỉ cần một trong số chúng hòan thành (Tức là ko cần tất cả các thread hòan thành, chỉ cần 1 task hòan thành phương thức get() sẽ nhận được kết quả.

Nếu 1 trong số task hòan thành (hoặc ném ra 1 ngoại lệ), phần còn lại của Callable sẽ được hủy bỏ (cancelled).

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

int b = 3 / 0;

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

Đoạn mã trên sẽ in ra các kết quả được trả về từ 1 trong những Callable trong tập hợp. Chạy nó vài lần bạn sẽ nhận được những kết quả khác nhau.

* invokeAll(Collection<?> extends Callable<T> tasks) Phương thức invokeAll() gọi tất cả đối tượng Callable bạn đẩy vào trong tập hợp. Phương thức này trả về 1 danh sách các đối tượng Future (list of Future) mà được trả về từ việc thực thi các Callables.

Hãy nhớ rằng một công việc có thể hòan thành do một ngoại lệ, vì vậy có nghĩa nó có thể không “thành công” nhiệm vụ. Không có cách nào để biết sự khác biệt trên đối tượng Future.

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

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

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

## 3.4. Cách kết thúc ExecutorService

* shutdown() Khi bạn đã thêm vào các nhiệm vụ cần thiết bạn nên tắt ExcutorService bằng phương thức shutdown(). Khi bạn gọi phương thức này có nghĩa ExcutorService sẽ từ chối nhận thêm các nhiệm vụ (Task), và một khi tất cả các nhiệm vụ đã được thêm vào trước đó đã hòan thành. Sau đó Executor sẽ được tắt (Có nghĩa tất cả các task được thêm vào trước khi gọi shutdown() đều sẽ được thực thi).

Dưới đây là một ví dụ về cách gọi ExecutorService shutdow():

executorService.shutdown();

* ShutdownNow() Nếu bạn muốn tắt ExecutorService ngay lập tức, bạn có thể gọi phương thức shutdownNow(). Điều này sẽ cố gắng ngắn chặn tất cả các nhiệm vụ ngay lập tức và loại bỏ các nhiệm vụ đã được đưa vào Queue nhưng chưa được thực thi. Không có gì đảm bảo về việc tắt các nhiệm vụ đang chạy hòan tòan, nhưng phương thức này là nỗ lực tốt nhất để tắt chúng.

Dưới đây là một ví dụ về cách gọi ExecutorService shutdowNow():

executorService.shutdownNow();

* awaitTermination() Phương thức ExecutorService awaitTermination () sẽ chặn luồng gọi nó cho đến khi ExecutorService tắt hoàn toàn hoặc cho đến khi hết thời gian nhất định. Phương thức awaitTermination () thường được gọi sau khi gọi shutdown () hoặc shutdownNow ().

Dưới đây là một ví dụ về cách gọi ExecutorService awaitTermination ():

executorService.shutdown();

executorService.awaitTermination();

Lưu ý: Bạn nên shutdown một ThreadPool bằng cách gọi phương thức shutdown() bởi vì ta không thể chắc chắn được rằng máy ảo Java có thể tự động làm điều đó

# 4. Tổng kết

Trên đây là những hướng dẫn cơ bản của mình về tạo và sử dụng ThreadPool trong Java. Nếu ứng dụng của bạn cần thực thi nhiều luồng cùng lúc thì bạn nên cố gắng tìm hiểu và sử dụng ThreadPool để hiệu năng tốt hơn, tiết kiệm thời gian vì không cần phải tạo Thread mới cho mỗi task.

Nguồn tham khảo: <http://tutorials.jenkov.com/java-util-concurrent/executorservice.html> <https://caffinc.github.io/2016/03/simple-threadpool/>