Khái niệm ThreadPool và Executor trong Java

Java 156 spring 65 spring boot 65 spring mvc 50 spring security 57 spring core 49

https://loda.me viết ngày 31/05/2019

`Thread Pool` Là một trong những yếu tố chính tác động tới hiệu năng của các chương trình lớn, đòi hỏi xử lý đồng thời nhiều nhiệm vụ cùng lúc.

Nếu bạn chưa rõ Thread trong Java, thì hãy đọc bài này trước:

1. Thread và xử lý đa luồng trong Java
2. Kiến thức nâng cao, sau khi đọc xong bài này: ThreadPoolExecutor

Một ví dụ đơn giản nhé (Trong thực tế sẽ khác, hãy coi đây là ví dụ nha):

Bây giờ, giả sử bạn có một Server Web. Nếu chúng ta nhận 1 request từ client, chúng ta sẽ xử lý mất 0.5s và trả về kết quả cho người dùng.

Thế nếu có 2 người request cùng lúc? => giải quyết bằng cách mỗi một request sẽ xử lý ở 1 thread, đơn giản.

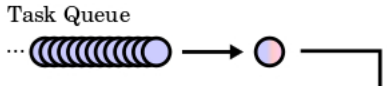
Thế nếu có 100 người request cùng lúc? => mỗi người tạo một thre... wait a minute.... (nếu 1 tháng có 10M lượt request => tạo ra 10M thread)

Nếu bạn tạo 1-2 thread mới, chả ai trách gì bạn cả. Nhưng nếu bạn tạo liên tục và tới hàng trăm cái mới mỗi lần nhưng lại giải quyết cùng 1 vấn đề thì có lỗ hổng đấy. Vì chi phí của việc tạo 1 thread là tương đối lớn, thường dẫn tới các vấn đề về hiệu năng và cấp phát dữ liệu.

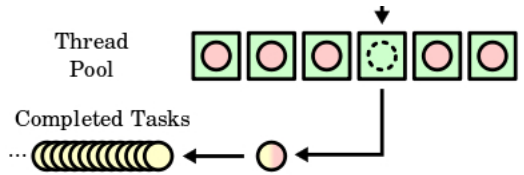
Với việc xử lý các tác vụ liên tục như vậy, có một giải pháp là sử dụng `Thread Pool`.

Ở ví dụ trên, Bây giờ tôi sẽ chỉ sử dụng 30 thread thôi! Và đặt 30 thread này ở trạng thái không làm gì và vứt vào 1 cái `Pool` (1 cái bể chứa, kiểu vậy). Với mỗi request đến, tôi sẽ lấy trong `Pool` ra 1 thread và xử lý công việc, xử lý xong, thì cất thread vào ngược trở lại pool. Đơn giản vậy thôi, như thế chúng ta sẽ không phải tạo mới Thread nữa. Tránh tình tốn chi phí và hiệu năng.

Vấn đề là giả sử có hơn 31 request tới cùng lúc thì sao? rất đúng, trường hợp này là chắc chắn có. Lúc này `Pool` sẽ không còn thread nào sẵn có nữa. Nên 1 request còn lại sẽ bị đẩy vào 1 hàng đợi `BlockingQueue`. Nó sẽ đợi ở đó, bao giờ `Pool` có 1 thread rảnh rỗi thì sẽ quay lại xử lý nốt =))) Chịu thôi, cứ ví dụ vậy hah.



Đầu mục bài viết



Đó là concept của `ThreadPool` đó các bạn.

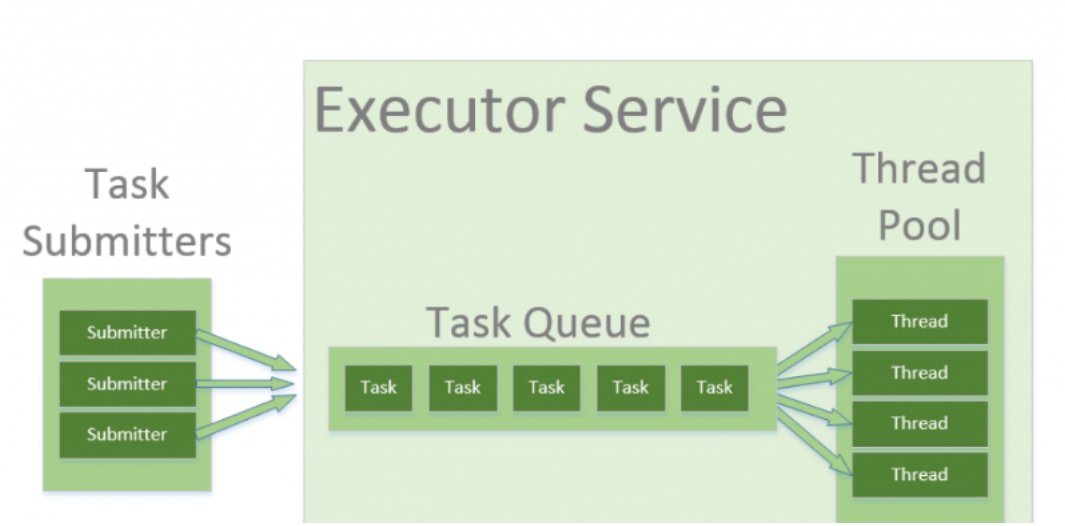
Cách tạo ThreadPool trong Java

Java Concurrency API hỗ trợ một vài loại `ThreadPool` sau:

- **Cached thread pool**: Mỗi nhiệm vụ sẽ tạo ra thread mới nếu cần, nhưng sẽ tái sử dụng lại các thread cũ. (Cái này vẫn nguy hiểm nhé, nên áp dụng với các task nhỏ, tốn ít tính toán)
- **Fixed thread pool**: giới hạn số lượng tối đa của các Thread được tạo ra. Các task khác đến sau phải chờ trong hàng đợi (BlockingQueue). (Ví dụ đầu bài)
- **Single-threaded pool**: chỉ giữ một Thread thực thi một nhiệm vụ một lúc.
- **Fork/Join pool**: một Thread đặc biệt sử dụng Fork/ Join Framework bằng cách tự động chia nhỏ công việc tính toán cho các core xử lý. (Tính toán song song)

Executor

`Executor` là một class đi kèm trong gói `java.util.concurrent`, 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.



Nói chung nó là thằng wrapper các các bước mình nói ở trên, và quản lý hộ chúng ta.

Chúng có thể tạo một Executor bằng cách sử dụng một trong các phương thức được cung cấp bởi lớp tiện ích `Executors` như sau:

- **newSingleThreadExecutor()**: trong ThreadPool chỉ có 1 Thread và các task (nhiệm vụ) sẽ được xử lý một cách tuần tự.
- **newCachedThreadPool()**: như giải thích ở trên, nó sẽ có 1 số lượng nhất định thread để sử dụng lại, nhưng vẫn sẽ tạo mới thread nếu cần. Mặc định nếu một Thread không được sử dụng trong vòng 60 giây thì Thread đó sẽ bị tắt.
- **newFixedThreadPool(int n)**: trong Pool chỉ có n Thread để xử lý nhiệm vụ, các yêu cầu tới sau bị đẩy vào hàng đợi
- **newScheduledThreadPool(int corePoolSize)**: tương tự như `newCachedThreadPool()` nhưng sẽ có thời gian delay giữa các Thread.
- **newSingleThreadScheduledExecutor()**: tương tự như `newSingleThreadExecutor()` nhưng sẽ có khoảng thời gian delay giữa các Thread.

Code chạy thử

Chúng ta sẽ lấy ví dụ đầu bài để code luôn nhé.

Tạo một class implement `Runnable` để xử lý request đến. (phân biệt `Runnable` và `Thread` nhé các bạn)

```
public class RequestHandler implements Runnable {
    String name;
    public RequestHandler(String name){
        this.name = name;
    }

    @Override
    public void run() {
        try {
            // Bắt đầu xử lý request đến
            System.out.println(Thread.currentThread().getName() + " Starting process " + name);
            // cho ngủ 500 milis để ví dụ là quá trình xử lý mất 0,5 s
            Thread.sleep(500);
            // Kết thúc xử lý request
            System.out.println(Thread.currentThread().getName() + " Finished process " + name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
}
```

newSingleThreadExecutor

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SingleThreadPoolExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // Có 100 request tới cùng lúc
        for (int i = 0; i < 100; i++) {
            executor.execute(new RequestHandler("request-" + i));
        }
        executor.shutdown(); // Không cho threadpool nhận thêm nhiệm vụ nào nữa


        while (!executor.isTerminated()) {
            // Chờ xử lý hết các request còn chờ trong Queue ...
        }
    }
}
// OUTPUT:
/*
..
..
pool-1-thread-1 Starting process request-98
pool-1-thread-1 Finished process request-98
pool-1-thread-1 Starting process request-99
pool-1-thread-1 Finished process request-99
*/
```

Cả chương trình chỉ có 1 pool, 1 thread duy nhất, xử lý toàn bộ request đến. Cái nào đến sau thì đợi thôi.

newFixedThreadPool()

```
public class FixedThreadPoolExample {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(5);

        // Có 100 request tới cùng lúc

        for (int i = 0; i < 100; i++) {
            executor.execute(new RequestHandler("request-" + i));
        }
        executor.shutdown(); // Không cho threadpool nhận thêm nhiệm vụ nào nữa

        while (!executor.isTerminated()) {
            // Chờ xử lý hết các request còn chờ trong Queue ...
        }
    }
}
```

```
// OUTPUT:
/*
..
..
pool-1-thread-2 Finished process request-96
pool-1-thread-5 Starting process request-99
pool-1-thread-3 Finished process request-97
pool-1-thread-4 Finished process request-98
pool-1-thread-5 Finished process request-99
*/
```

Loại này thì chúng ta cố định 5 thread, và nó cứ mặc định như vậy mà xài thôi, thiếu thread thì phải chờ tới khi có

newCachedThreadPool()

```
public class CachedThreadPoolExample {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newCachedThreadPool();

        // Có 100 request tới cùng lúc

        for (int i = 0; i < 100; i++) {
            executor.execute(new RequestHandler("request-" + i));
            Thread.sleep(200);
        }
        executor.shutdown(); // Không cho threadpool nhận thêm nhiệm vụ nào nữa


        while (!executor.isTerminated()) {
            // Chờ xử lý hết các request còn chờ trong Queue ...
        }
    }
}

//OUTPUT:
/*
..
..
pool-1-thread-3 Starting process request-98
pool-1-thread-1 Finished process request-96
pool-1-thread-1 Starting process request-99
pool-1-thread-2 Finished process request-97
pool-1-thread-3 Finished process request-98
pool-1-thread-1 Finished process request-99
*/
```

Có chút khởi sắc, chương trình chạy nhanh hơn hẳn. Vì nó được tạo số thread thoải mái nếu cần :)))) Rất nguy hiểm. Nhưng bạn sẽ thấy là có chỗ nó sử dụng lại các thread đã xong trước đó.

Bài viết tới đây kết thúc, Đây là một kiến thức rất quan trọng nên các bạn cố gắng nắm vững nhé.

toàn bộ code mình để tại GITHUB loda-kun: CODE

chúc các bạn học tập tốt! ohoho

Cùng một tác giả

Series hướng dẫn Spring Boot căn bản, Zero to Hero 6 0

Java spring boot spring mvc spring core spring security

Vì sao bạn nên học Java? Trước khi nói Spring Boot, chúng ta nói về nền tảng của nó, chính là Java. Java ra đời năm 1991, tới nay thì đã gần 30 n...

https://loda.me viết 3 tháng trước

Tại sao nên học Java và Spring? 6 0

Java spring spring boot spring mvc spring core spring security

Vì sao bạn nên học Java? Trước khi nói Spring Boot, chúng ta nói về nền tảng của nó, chính là Java. Java ra đời năm 1991, tới nay thì đã gần 30 n...

https://loda.me viết 3 tháng trước

「Spring」Hướng dẫn lập trình Spring căn bản cho người mới 4 0

Java spring boot spring security spring mvc spring

Xin chào tất cả các bạn, trước khi đi vào chi tiết bài hôm nay, các bạn cần đọc cho mình các khái niệm sau: 1. Khái niệm tightcoupling (liên k...

https://loda.me viết 3 tháng trước

Bài viết liên quan

Spring JPA fetch test TIL spring spring boot spring jpa fetch 2 0

I used Spring boot, Hibernate few times back then at University, I'v started using it again recently. In this (Link), I want to check how Spring J...

Rey viết 8 tháng trước