**Chapter 16**

**Thread và Locks**

Trong cuộc phỏng vấn trên Microsoft, Google hoặc Amazon, nó không quá khủng khiếp khi được hỏi về việc thực hiện một thuật toán với Threads (trừ khi bạn làm việc trong một nhóm mà đây là một kĩ năng cực kì quan trọng). Tuy nhiên nó tương đối phổ biến trong các cuộc phỏng vấn khác tại bất kì công ty nào để truy cập vào sự hiểu biết chung về các Thread của bạn, đặc biệt hiểu biết bế tắc của bạn.

Chương này sẽ giới thiệu về đề tài này.

**Threads in Java**

Mỗi thread trong Java được tạo và điều khiển bởi đối tượng độc nhất của Java.lang. Lớp Thread. Khi một ứng dụng độc lập được chạy, một thread người dùng được tạo tự động để thực thi phương thức chính. Thread này được gọi là thread chính.

Trong java, chúng ta có thể triển khai các thread theo 2 cách:

* Thực hiện bởi java.lang. Giao diện Runnable.
* Mở rộng java.lang, lớp Thread

Chúng ta sẽ cover cả hai ở bên dưới.

*Giao diện triển khai Runnable Interface*

Theo dõi và khai triển một cách đơn giản.

1. public interface Runnable {
2. void run();
3. }

Để tạo và sử dụng một thread sử dụng giao diện này, chúng ta làm theo:

1. Tạo một lớp mà thực thi giao diện Runnable. Một đối tượng của lớp này là một đối tượng Runnable.
2. Tạo một đối tượng của loại Thread bằng cách vượt qua một đối tượng Runnable như tạo đối số cho hàm tạo chuỗi. Đối tượng Thread bây giờ có một đối tượng Runnable để thực thi phương thức run().
3. Phương thức start( ) được gọi trên đối tượng Thread được tạo trong bước trước.

Ví dụ :

public class RunnableThreadExample implements Runnable {

public int count = 0;

public void run() {

System.out.println("RunnableThread starting.");

try {

while (count < 5) {

Thread.sleep(500);

count++ ;

}

} catch (InterruptedException exc) {

System.out.println("RunnableThread interrupted.");

}

System.out.println("RunnableThread terminating.");

}

}

public static void main(String[] args){

RunnableThreadExample instance = new RunnableThreadExample();

Thread thread = new Thread(instance);

thread.start();

/ \* waits until above thread counts to 5 (slowly) \*/

while (instance.count != 5 ) {

try {

Thread.sleep(250);

} catch (InterruptedException exc) {

exc.printStackTrace();

}

}

}

Trong code ở trên, thấy rằng tất cả những gì chúng ta thực sự cần làm là thực hiện phương thức run( ) lớp của chúng ta (dòng 4). Phương thức khác có thể vượt qua một thể hiện của lớp để Thread (obj) mới (dòng 19-20) và gọi start( ) on the thread (dòng 21).

*Mở rộng lớp Thread*

Thay vào đó, chúng ta có thể tạo một luồng bằng cách mở rộng lớp thread. điều này hầu như luôn luôn có nghĩa là chúng ta ghi đè lên phương thức run (), và lớp con cũng có thể gọi hàm tạo một cách rõ ràng trong hàm tạo của nó.

Phía dưới là code ví dụ về điều này:

public class ThreadExample extends Thread {

int count = 0;

public void run( ) {

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

try {

while (count < 5) {

Thread.sleep(500);

System.out.println("In Thread, count is " + count);

count++;

}

} catch (InterruptedException exc ) {

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

} System.out.println("Thread terminating.");

}

}

public class ExampleB {

public static void main(String args[]) {

ThreadExample instance = new ThreadExample();

instance.start();

while (instance.count != 5) {

try {

Thread.sleep(250);

} catch (InterruptedException exc) {

exc.printStackTrace() ;

}

}

}

}

Code này rất giống với phương pháp tiếp cận thông minh. Sự khác biệt là vì chúng ta đang mở rộng lớp Thread, thay vì chỉ thực hiện một giao diện, chúng ta có thể gọi start( ) trên cá thể của lớp đó.

Mở rộng Thread Class so với thực hiện Runnable Interface

Khi tạo các chủ đề, có hai lý do tại sao việc thực hiện giao diện Runnable có thể thích hợp hơn để mở rộng lớp Thread:

* Java không hỗ trợ đa kế thừa. Do đó, việc mở rộng các lớp Thread có nghĩa là lớp con không thể mở rộng bất kỳ lớp nào khác. Một lớp thực hiện giao diện Runnable sẽ có thể mở rộng một lớp khác.
* Một lớp chỉ có thể quan tâm đến việc runnable, và do đó, thừa hưởng toàn bộ chi phí của lớp Thread sẽ là quá mức.

**Synchronization and Lock**

Các Thread trong một tiến trình đã cho chia sẻ cùng một không gian bộ nhớ, cả hai đều là số dương và âm. Nó cho phép các Thread chia sẻ dữ liệu, có thể có giá trị. Tuy nhiên, nó cũng tạo ra cơ hội cho các vấn đề khi hai Thread sửa đổi một tài nguyên ở cùng một lúc. Java cung cấp đồng bộ hóa để kiểm soát quyền truy cập vào các tài nguyên được chia sẻ. Từ khóa được đồng bộ hóa và khóa tạo thành cơ sở để triển khai thực hiện đồng bộ mã.

*Synchronized Methods*

Thông thường, chúng tôi hạn chế quyền truy cập vào tài nguyên được chia sẻ thông qua việc sử dụng từ khóa được đồng bộ hóa. Tôi có thể được áp dụng cho các phương pháp và khối mã, và hạn chế nhiều Thread từ việc thực thi code đồng thời trên cùng một đối tượng

Để làm rõ điểm cuối cùng, xem xét mã sau:

public class MyClass extends Thread {

private String name;

private MyObject myObj;

public MyClass(MyObject obj , String n) {

name = n ;

myObj = obj;

}

public void run( ) {

myObj.foo(name);

}

}

public class MyObject {

public synchronized void foo(String name ) {

try {

System.out.println("Thread " + name + ".foo(): starting");

Thread.sleep(3000);

System.out.println("Thread " + name + ".fooQ: ending");

} catch (InterruptedException exc) {

System.out.println("Thread " + name + ": interrupted.");

}

}

}

Có thể hai trường hợp Myclass gọi foo cùng một lúc không? Nó phụ thuộc. Nếu họ có cùng một trường hợp MyObject, thì không. Nhưng, nếu họ giữ các tài liệu tham khảo khác nhau, thì câu trả lời là có.

/\* Difference references - both threads can call MyObject.foo( ) \*/

MyObject objl = new MyObject();

MyObject obj2 = new MyObject();

MyClass threadl = new MyClass(obj1, "1");

MyClass thread2 = new MyClass(obj2, "2");

thread1.start();

thread2.start( );

/\* Same reference to obj. Only one will be allowed to call foo, and the other will be forced to wait. \*/

MyObject obj = new MyObject();

MyClass thread1 = new MyClass(obj, "1");

MyClass thread2 = new MyClass(obj, "2");

thread1.start()

thread2.start()

Các phương thức tĩnh đồng bộ hóa trên lớp class. Hai luồng trên không thể đồng thời thực hiện các phương thức tĩnh đồng bộ trên cùng một lớp, ngay cả khi một phương thức đang gọi foo và lệnh kia là thanh gọi.

public class MyClass extends Thread {

...

public void run() {

if (name.equals("1") ) MyObject.foo(name);

else if (name.equals("2") ) MyObject.bar(name) ;

}

}

public class MyObject {

public static synchronized void foo(String name ) {

/\* same as before \*/

}

public static synchronized void bar(String name) {

/\* same as foo \*/

}

}

Nếu bạn chạy code này, bạn sẽ thấy in ra:

Thread l.foo(): starting

Thread l.foo() : ending

Thread 2.bar() : starting

Thread 2.bar() : ending

*Synchronized Blocks*

Tương tự, một khối mã có thể được đồng bộ hóa. Điều này hoạt động rất giống với việc đồng bộ hóa một phương thức.

public class MyClass extends Thread {

...

public void run() { myObj.foo(name); }

}

public class MyObject {

public void foo(String name) {

synchronized(this) {

...

}

}

}

Giống như việc đồng bộ một phương thức, chỉ một Thread cho mỗi thể hiện của MyObject có thể thực thi mã trong khối đồng bộ. Điều đó có nghĩa là, nếu threadl và thread2 có cùng một thể hiện MyObject, chỉ một cái sẽ được phép thực thi khối mã tại một thời điểm.

*Locks*

Để kiểm soát chi tiết hơn, chúng tôi có thể sử dụng khóa. Một khóa (hoặc màn hình)được sử dụng để đồng bộ hóa quyền truy cập vào tài nguyên được chia sẻ bằng cách liên kết tài nguyên với khóa. Một luồng được truy cập vào tài nguyên được chia sẻ bằng cách đầu tiên có được khóa được liên kết với tài nguyên. Tại bất kỳ thời điểm nào, tối đa một luồng có thể giữ khóa và do đó, chỉ có một luồng có thể truy cập vào phần chia sẻ được chia sẻ.

Trường hợp sử dụng phổ biến cho khóa là khi một tài nguyên được truy cập từ nhiều địa điểm, chỉ nên truy cập một chuỗi tại một thời điểm. Trường hợp này được thể hiện trong mã bên dưới:

public class LockedATM {

private Lock lock;

private int balance = 100;

public LockedATMQ {

lock = new ReentrantLockQ;

}

public int withdraw(int value) {

lock.lock();

int temp = balance;

try {

Thread.sleep(100);

temp = temp - value;

Thread.sleep(108);

balanc e = temp;

} catch (interruptedException e) { }

lock.unlock();

return temp;

}

public int deposit(int value) {

lock.lock();

int temp = balance;

try {

Thread.sleep(100);

temp = temp + value;

Thread.sleep(300);

balance = temp;

} catch (InterruptedException e) { }

lock.unlock();

return temp;

}

}

Tất nhiên, chúng tôi đã thêm mã để cố ý làm chậm việc thực hiện rút tiền và gửi tiền, vì nó giúp minh họa các vấn đề tiềm năng có thể xảy ra. Bạn có thể không viết mã chính xác như thế này, nhưng tình huống mà nó phản chiếu là rất, rất thực tế. Sử dụng khóa sẽ giúp bảo vệ tài nguyên được chia sẻ khỏi bị sửa đổi theo các cách không mong muốn.

**Deadlocks and Deadlock Prevention**

Một bế tắc là một tình huống mà một Thread đang chờ một khóa đối tượng mà một thread khác giữ, và thread thứ hai này đang chờ một khóa đối tượng mà thread đầu tiên giữ (hoặc một tình huống tương đương với một số thread). Vì mỗi thread đang chờ các thread khác từ bỏ khóa, cả hai đều chờ đợi mãi mãi. Các chủ đề được cho là bế tắc.

Để bế tắc xảy ra, bạn phải có tất cả bốn điều kiện sau đây:

1. Loại trừ lẫn nhau: Chỉ một quy trình có thể truy cập tài nguyên tại một thời điểm nhất định. (Hoặc, chính xác hơn, có giới hạn truy cập vào một tài nguyên. Một bế tắc cũng có thể xảy ra nếu một tài nguyên có số lượng hạn chế.)
2. Giữ và chờ: Các tiến trình đã nắm giữ tài nguyên có thể yêu cầu các tài nguyên bổ sung, mà không từ bỏ các tài nguyên hiện tại của chúng.
3. Không ưu tiên: Một quy trình không thể loại bỏ cưỡng bức một tài nguyên quy trình khác.
4. Vòng chờ đợi: Hai quá trình nữa sẽ hình thành một chuỗi vòng tròn, trong đó mỗi quá trình đang chờ đợi một tài nguyên khác trong chuỗi.

Việc ngăn chặn deadlock đòi hỏi loại bỏ bất kỳ điều kiện nào ở trên, nhưng nó trở nên phức tạp vì nhiều điều kiện này khó đáp ứng. Ví dụ: việc xóa số 1 rất khó bởi vì nhiều tài nguyên chỉ có thể được sử dụng bởi một quy trình tại một thời điểm (ví dụ: máy in). Hầu hết các thuật toán ngăn chặn bế tắc tập trung vào việc tránh điều kiện # 4: vòng đợi.

**Interview Questions**

1. Sự khác nhau giữa một thread vả một process?
2. Bạn làm thế nào để đo thời gian dùng cho việc chuyển đổi một phạm vi?
3. Trong vấn đề triết lý ăn uống nổi tiếng, một loạt các triết học đang ngồi quanh một chiếc bàn tròn với một chiếc đũa giữa mỗi người. Một nhà triết học cần cả hai chiếc đũa để ăn, và luôn nhấc chiếc đũa bên trái trước chiếc đũa bên phải. Một deadlock có thể có khả năng xảy ra nếu tất cả nhà triết học với lấy chiếc đũa bên trái cùng một lúc. Sử dụng threads và locks, thực hiện một mô phỏng của vấn đề triết lý ăn uống để ngăn ngừa deadlocks.
4. Thiết kế một lớp chỉ cung cấp một lock nếu không có deadlocks khả thi.
5. Giả sử chúng ta có đoạn mã sau:

public class Foo {

public Foo( ) { ... }

public void first() { ... }

public void second() { ... }

public void third() { ... }

}

Ví dụ tương tự của Foo sẽ được chuyển đến ba luồng khác nhau. ThreadA sẽ gọi đầu tiên, threads sẽ gọi thứ hai, và threadC sẽ gọi thứ ba. Thiết kế một cơ chế để đảm bảo rằng đầu tiên được gọi trước thứ hai và thứ hai được gọi trước thứ ba.

1. Bạn được nhận được một lớp với phương thức A đồng bộ và phương thức B bình thường. Nếu bạn có hai threads trong một cá thể của chương trình. Họ có thể thực thi cả hai cùng một lúc không? Họ có thể thực thi A và B cùng một lúc?