Đồng bộ hóa các luồng trong Java

Đăng vào **20/02/2018** *Được đăng bởi* **GP Coder** 17314 *Lượt xem*

Trong bài trước tôi đã giới thiệu với các bạn các kiến thức cơ bản về Lập trình đa luồng trong Java. Trong bài này chúng ta tiếp tục tìm hiểu về vấn đề đồng bộ hóa giữa các luồng trong Java, cơ chế hoạt động và cách thức giao tiếp giữa các luồng.

Nội dung [Ẩn]

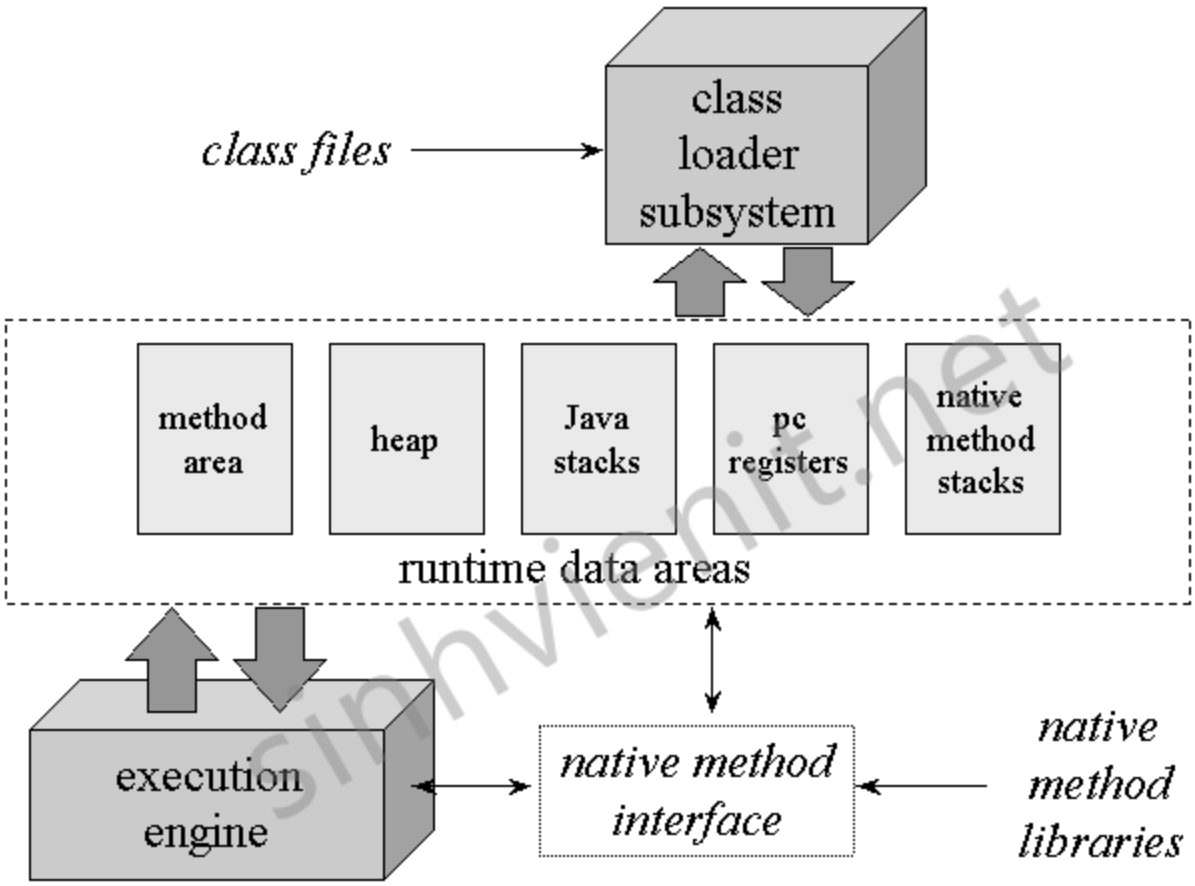
1 Đối tượng khóa
2 Đồng bộ là gì? Tại sao lại cần đồng bộ?
  2.1 Tại sao cần đồng bộ?
  2.2 Đồng bộ hóa là gì?
3 Java monitor là gì? Cách hoạt động của java monitor?
  3.1 Java monitor là gì ?
  3.2 Cách hoạt động của java monitor
4 Các cách để đồng bộ trong Java
  4.1 Synchronized methods
  4.2 Synchronized statements/ Synchronized Block
  4.3 Static synchronized method
5 So sánh các cách synchronized
6 Phương thức wait(), notify(), notifyall()
  6.1 wait()
  6.2 notify() và notifyall()
  6.3 Ví dụ minh họa
7 Deadlock (Khoá chết) là gì?
8 Tại sao không nên dùng phương thức Thread.stop()
9 Tại sao không nên dùng phương thức Thread.suspend(), Thread.resume()

10 Tại sao không nên dùng phương thức Thread.Destroy()?

1. Đối tượng khóa

Khi lập trình đa luồng, người ta đã chọn kĩ thuật này vì 1 phần lý do là có thể dùng chung dữ liệu của chương trình dễ dàng. Tuy nhiên, việc dùng chung này sẽ có thể xảy ra điều không mong muốn (sẽ đề cập ở phần tại sao cần đồng bộ hóa).

Đối tượng khóa (**Object Locking**) là đối tượng nằm trong danh sách các đối tượng mà nhiều luồng muốn cùng sử dụng đồng thời như đã nói ở trên. Để tiếp tục nói về vấn đề này, chúng ta cần xem xét về kỹ thuật của máy ảo Java. Hình dưới mô tả sơ lược về kỹ thuật này:



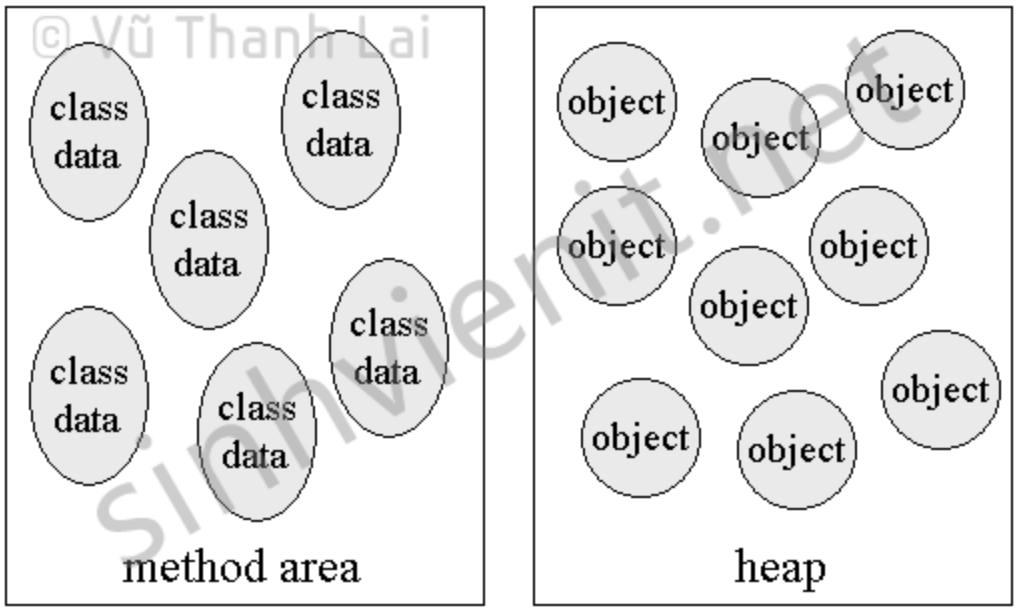
*(*) Ở đây chúng ta không tìm hiểu sâu về kiến trúc của JVM nên mình chỉ giải thích sơ về* ***class loader subsystem*** *là 1 hệ thống tìm nạp các class, kiểm tra tính đúng đắn của các class và cấp phát bộ nhớ cho các biến đối tượng khi chạy chương trình*
*(*) Native method là phương thức được viết bởi 1 ngôn ngữ khác như: C,C++, Assembly. Java sử dụng các method này rất nhiều trong việc tương tác với hệ điều hành và phần cứng*

Khi **máy ảo java** chạy chương trình, nó cần bộ nhớ để lưu trữ nhiều thứ bao gồm: Mã byte code và các thông tin khác khi tải các class file lên như: các đối tượng, các tham số của

phương thức, giá trị trả về, biến local, và các kết quả tính toán trung gian. Máy ảo java sẽ tổ chức lưu trữ dữ liệu cần thiết trong thời gian chạy chương trình vào 1 nơi gọi là "runtime data areas".

Runtime data areas được chia sẻ cho các luồng của ứng dụng và các luồng riêng biệt khác trong cùng 1 máy ảo java. Mỗi phiên bản của máy ảo java có 1 method area và 1 heap. Những vùng này sẽ chia sẻ cho tất cà các luồng đang chạy trong máy ảo java hiện tại. Khi máy ảo java load các file class lên, nó sẽ phân tích tất cả các thông tin về lớp từ dạng nhị phân trong các file class. Sau đó, nó sẽ đặt các thông tin này vào method area. Như vậy, **method area sẽ chứa tất cả các thông tin về biến, thuộc tính, phương thức** ...

Tương tự như vậy, khi chương trình chạy, nó cũng sẽ đặt tất cả các thể hiện của đối tượng (hay các tham chiếu đến đối tượng) vào vùng heap. Xem tiếp hình dưới để hiểu về vùng bộ nhớ này:



Vì 2 vùng này được chia sẻ, do đó các luồng thuộc cùng 1 máy ảo java hoàn toàn có thể dùng chung các tham chiếu đến đối tượng ở heap và dùng chung các thuộc tính, phương thức của đối tượng ở method area, điều này làm cho các luồng có thể dùng chung dữ liệu của chương trình. Chính việc dùng chung là là 1 điểm hấp dẫn của đa luồng nhưng sẽ gây ra 1 số phiền toái mà ta sẽ đề cập ở phần tiếp theo và cách giải quyết phiền toái đó.

> Xem thêm Quản lý bộ nhớ trong Java với Heap Space vs Stack

2. Đồng bộ là gì? Tại sao lại cần đồng bộ?

2.1. Tại sao cần đồng bộ?

Trong kỹ thuật đa luồng, nếu các luồng sử dụng dữ liệu độc lập thì ta không có gì phải tranh luận. Nhưng nếu trên hệ thống nhiều CPU hoặc CPU đa nhân hay CPU hỗ trợ siêu phân luồng, các luồng sẽ thục sự hoạt động song song tại cùng 1 thời điểm. Như vậy, nếu các luồng này cùng truy xuất đến 1 biến dữ liệu hoặc 1 phương thức nhờ vào lý do đã nói ở phần trên, điều này có thể gây ra việc sai lệch dữ liệu.

```
1  public class Counter {
2      int count=0;
3
4      public void tang() {
5          count=count+1;
6      }
7  }
```

Giả sử rằng, tại cùng 1 thời điểm có 2 luồng cùng lúc gọi phương thức tang() trên 1 đối tượng thuộc lớp Counter.

Như vậy, cùng 1 lúc, 2 luồng cùng lấy ra được giá trị count hiện tại là 0, và cùng lúc cộng thêm 1 vào giá trị count này thành 1, sau đó cùng ghi giá trị mới cộng lại được là 1 lên RAM.

Nếu thực sự như vậy, sau khi cả 2 luồng thực hiện công việc thì count có giá trị 1. Tuy nhiên, 2 luồng cùng tăng giá trị count thì count phải có giá trị 2 mới đúng là việc chúng ta mong muốn

=> Việc sắp xếp thứ tự truy xuất đối tượng lúc này là thật sự cần thiết khi các luồng có dùng chung dữ liệu.

2.2. Đồng bộ hóa là gì?

Như đã nói ở trên, việc sắp xếp thứ tự các luồng truy xuất đối tượng thật sự cần thiết trong kỹ thuật đa luồng. **Đồng bộ hóa (synchronized) chính là việc xắp xếp thứ tự các luồng khi truy xuất vào cùng đối tượng sao cho không có sự xung đột dữ liệu.** Nói cách khác, đồng bộ hóa tức là thứ tự hóa.

3. Java monitor là gì? Cách hoạt động của java monitor?

3.1. Java monitor là gì ?

JVM Monitor (Java Virtual Machine) hay java monitor là 1 công cụ giám sát hỗ trợ cho việc đồng bộ hóa các luồng.

3.2. Cách hoạt động của java monitor

Bạn có thể tưởng tượng monitor như 1 công trình xây dựng có 1 phòng đặc biệt. Phòng này chỉ được sử dụng bởi 1 luồng duy nhất vào 1 thời điểm, trong phòng có chứa các dữ liệu cần thiết mà luồng đang ở trong phòng cần cho việc thực hiện công việc của mình và dữ liệu này cũng là dữ liệu mà các luồng khác cũng đang "thèm khát". Từ khi luồng được vào phòng này đến khi luồng rời khỏi phòng, luồng được phép truy cập đến bất kỳ dữ liệu nào trong phòng đang có, còn các luồng khác không được phép "đụng" đến các dữ liệu này.

Khi luồng đến được tới cửa phòng. Ta gọi là: luồng "Dành được monitor" (**acquiring** the monitor). Luồng đang ở trong phòng và sử dụng các dữ liệu trong đó ta gọi luồng đang "chiếm giữ monitor" (**owning** the monitor). Khi luồng rời khỏi phòng ta gọi: luồng "trả monitor" (**releasing** the monitor). Khi luồng rời khỏi hoàn toàn công trình xây dựng ta gọi: "luồng thoát khỏi monitor" (**exiting** the monitor).

Một luồng không thể chỉ làm 1 công việc nhỏ mà có thể nhiều công việc. Khi ta chia nhỏ các công việc này ra tới mức toán tử, không thể chia nhỏ hơn nữa, ta gọi công việc không thể chia nhỏ này là w, khi đó w của 1 luồng thực hiện trong 1 monitor riêng biệt gọi là **Monitor Region**.

Khi 1 luồng đến để bắt đầu monitor region thì nó được bảo vệ bởi monitor, mà monitor này đang bị chiếm giữ bởi 1 luồng khác, Còn các luồng mới hơn luồng đang được bảo vệ đến sẽ bị đưa vào entry set. Sau khi luồng đang chiếm giữ monitor này kết thúc công việc và trả monitor thì luồng đang được monitor bảo vệ sẽ được đưa vào làm luồng chiếm giữ monitor. Còn các luồng đang ở entry set sẽ phải tranh đấu để dành được monitor bào vệ làm luồng tiếp theo. Quá trình tranh đấu được thực hiện bởi máy ảo java dựa trên độ ưu tiên hoặc cơ chế **FIFO (First In First Out), LIFO (Last In First Out)**.

Để mô tả về java monitor, ta có thể xem hình dưới đây:

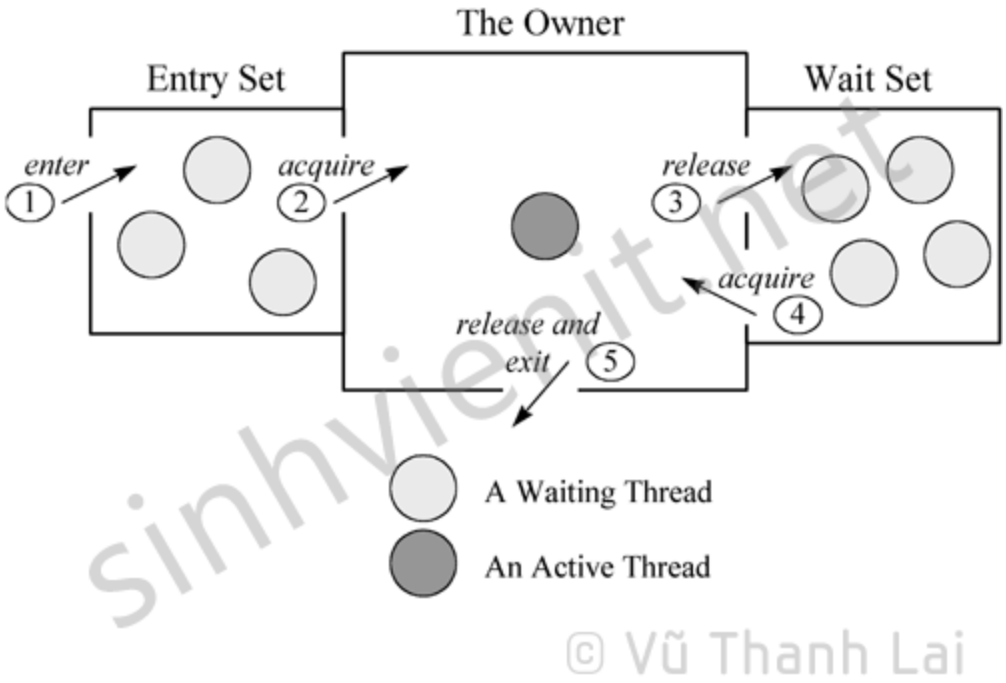


Figure 20-1. A Java monitor.

Ở hình trên, ta thấy có 5 cửa mà luồng phải đi qua trong quá trình dành và nắm giữ monitor. Khi 1 luồng đến để bắt đầu chiếm monitor nó phải đi vào khu vực entry set thông qua cửa 1. Nếu không có luồng nào đang sở hữu monitor và không có luồng nào khác đang chờ trong entry set thì nó mặc nhiên sẽ được vượt qua cửa số 2 và là luồng chiếm giữ monitor. Nếu như khi nó đến có luồng khác đang nắm giữ monitor thì luồng phải chờ trong entry set, và có thể có sẵn các luồng khác cũng đang nằm trờ trong entry set với nó.

Cũng ở hình trên, ta có thể thấy hiện có 3 luồng đang chờ ở entry set và 4 luồng đang chờ ở wait set, các luồng này vẫn sẽ phải chờ cho đến khi luồng sở hữu monitor trả lại monitor. Luồng đang sở hữu monitor có thể trả monitor bằng 2 cách: Hoàn thành công việc cần làm trong monitor region và thoát, gửi lệnh **wait()**. Nếu hoàn tất công việc, luồng sẽ rời khỏi monitor theo cửa số 5. Nếu nó gọi lệnh wait(), nó sẽ vào cửa sổ thứ 3 để chờ ở vùng wait set.

Nếu luồng chiếm giữ cũng trả monitor mà không thông báo bằng phương thức **notify()** và không có luồng nào trước đây đang chờ sự thông báo này (Tức là không có luồng nào trong wait set) thì các luồng ở vùng entry set sẽ cạnh tranh để chiếm giữ monitor. Còn nếu trước đó có thực hiện lệnh notify() và trong wait set có luồng đang chờ thì các luồng ở vùng entry set và wait set sẽ cạnh tranh với nhau để dành monitor, nếu luồng bên entry set thắng nó sẽ vào bằng cửa số 2, nếu luồng bên wait set thắng, nó sẽ vào bằng cửa số 4. Một luồng chỉ có thể thục hiện lệnh wait() khi nó đang là luồng sở hữu monitor, và khi nó đã vào wait set do lệnh wait() thì nó không thể tự ý quay lại làm chủ sở hữu monitor được.

Trong máy ảo java, luồng có thể tùy chọn chỉ định 1 thời gian chờ đợi khi gọi lệnh **wait(long milis)**. Nếu hết thời gian này, máy ảo java sẽ tự gửi lệnh notify() cho luồng cho dù luồng khác không gửi.

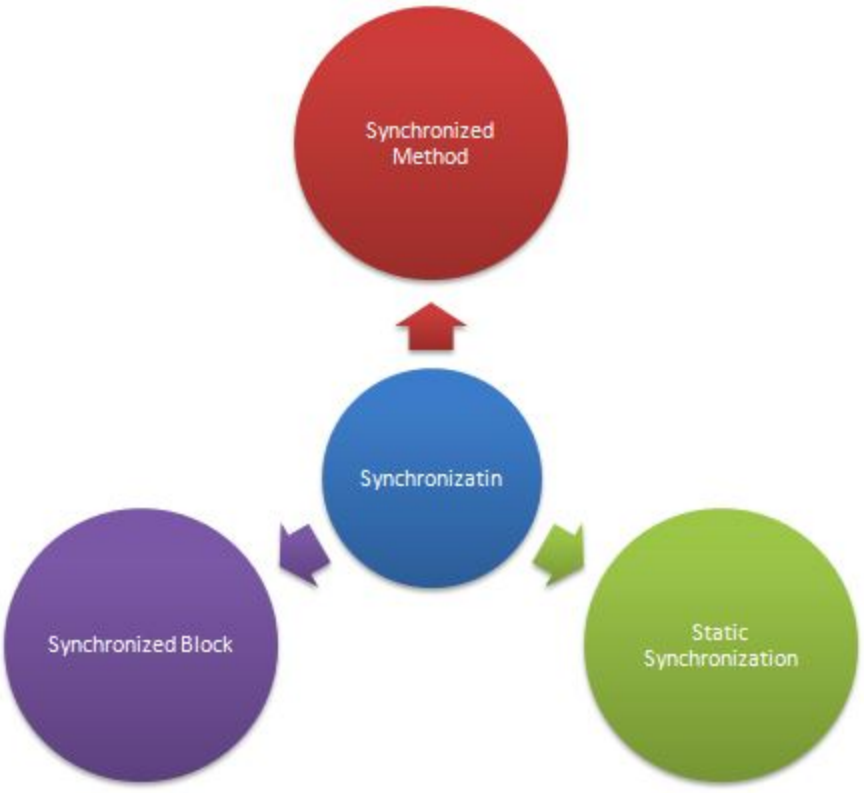
Máy ảo java cung cấp 2 loại lệnh thông báo cho luồng đang ở wait set là: **notify()** và **notifyall()**. notify() sẽ chọn 1 luồng tùy ý trong wait set để luồng này phục hồi lại, còn notifyall() sẽ gửi cho tất cả các luồng trong wait set.

*(*) Lưu ý: Phương thức wait(),notify(),notifyall() là phương thức của đối tượng mà luồng đang sử dụng chứ không phải phương thức của luồng. Xem chi tiết ở phần dưới*

4. Các cách để đồng bộ trong Java

Như đã đề cập về nguyên nhân dẫn đến cần phải đồng bộ và cách máy ảo java đồng bộ. Ở phần này, ta tiếp tục cách đồng bộ các luồng trong ứng dụng của chúng ta khi lập trình.

Java cung cấp cho ta 3 cách để đồng bộ là: synchronized methods và synchronized statements, static synchronized method.



4.1. Synchronized methods

Hãy xem ví dụ sau:

ShareMemory.java

```java
package com.gpcoder.sync;

public class ShareMemory {
    public synchronized void printData(String threadName) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + ": " + i);
        }
    }
}
```

WorkingThread.java

```java
package com.gpcoder.sync;

public class WorkingThread extends Thread {
    private ShareMemory mShareMemory;
    private String mThreadName;

    public WorkingThread(ShareMemory sm, String threadName) {
        this.mShareMemory = sm;
        this.mThreadName = threadName;
    }

    @Override
    public void run() {
        mShareMemory.printData(mThreadName);
    }
}
```

ShareMemoryTest.java

```java
package com.gpcoder.sync;

public class ShareMemoryTest {

    public static void main(String[] args) {

        ShareMemory sm = new ShareMemory();
        WorkingThread thread1 = new WorkingThread(sm, "Thread1");
        WorkingThread thread2 = new WorkingThread(sm, "Thread2");
        WorkingThread thread3 = new WorkingThread(sm, "Thread3");

        thread1.start();
        thread2.start();
        thread3.start();
    }

}
```

Thực thi chương trình trên:

```
Thread2: 1
Thread2: 2
```

```
Thread2: 3
Thread2: 4
Thread2: 5
Thread1: 1
Thread1: 2
Thread1: 3
Thread1: 4
Thread1: 5
Thread3: 1
Thread3: 2
Thread3: 3
Thread3: 4
Thread3: 5
```

Trong ví dụ trên, ngoài từ khóa mô tả phạm vi truy xuất của phương thức là public và từ khóa mô tả kiểu dữ liệu trả về là void, ta có thêm từ khóa **synchronized**, từ khóa này sẽ ngăn các luồng gọi đồng thời. Về cơ chế ngăn, phương thức này sẽ được cấp duy nhất 1 "java monitor" cho phương thức này như đã đề cập ở phần trước, chỉ có luồng nào đang nắm giữ monitor mới có quyền gọi phương thức này.

4.2. Synchronized statements/ Synchronized Block

Với Synchronized methods thời gian chờ giữa các luồng khá lớn. Trong một vài trường hợp cụ thể, bạn cần hiệu suất và đồng bộ hóa cả hai cùng một lúc trong một ứng dụng. Java cung cấp cơ chế đồng bộ hóa một phần của code trong một phương thức (Synchronized statements/ Synchronized Block). Vì vậy, nhiều thread có thể truy cập vào các thông tin mà không cần phải được đồng bộ.

Ví dụ, giả sử đọc và viết cả hai hoạt động đang được thực hiện trong phương pháp cụ thể của bạn sau đó bạn có thể đồng bộ hóa các hoạt động viết và để cho thread đọc thông tin.

Khi sử dụng Synchronized Block, chúng ta cần phải cho biết khóa của đối tượng nào bạn muốn sử dụng làm khóa trong đoạn code cần đồng bộ. Bằng cách đó, chúng ta cũng có thể sử dụng đối tượng khác làm khóa.

Hãy xem ví dụ trên được viết lại bằng cách sử dụng Synchronized Block:

```
package com.gpcoder.sync.block;

public class ShareMemory {
    public void printData(String threadName) {
        // Do Something before synchronized ...
        synchronized (this) {
            for (int i = 1; i <= 5; i++) {
                System.out.println(threadName + ": " + i);
            }
        }
```

```
    }
}
```

4.3. Static synchronized method

Ngoài 2 cách trên, chúng ta có thể synchronized một static method.

Mỗi lớp được load trong Java có một thể hiện tương ứng của java.lang.Object và được sử dụng để đại diện cho lớp đó. Một khóa của java.lang.Object được sử dụng để bảo vệ các phương thức static synchronized của lớp đó.

Hãy xem ví dụ sau:

ShareMemory.java

```
package com.gpcoder.sync.staticmethod;

public class ShareMemory {
    public static synchronized void printData(String threadName) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + ": " + i);
        }
    }
}
```

WorkingThread.java

```
package com.gpcoder.sync.staticmethod;

public class WorkingThread extends Thread {
    private String mThreadName;

    public WorkingThread(String threadName) {
        this.mThreadName = threadName;
    }

    @Override
    public void run() {
        ShareMemory.printData(mThreadName);
    }
}
```

ShareMemoryTest.java

```
package com.gpcoder.sync.staticmethod;

public class ShareMemoryTest {

    public static void main(String[] args) {

```

```
        WorkingThread thread1 = new WorkingThread("Thread1");
        WorkingThread thread2 = new WorkingThread("Thread2");
        WorkingThread thread3 = new WorkingThread("Thread3");

        thread1.start();
        thread2.start();
        thread3.start();
    }

}
```

5. So sánh các cách synchronized

Phạm vi khóa:

- Synchronized method: khóa toàn bộ phương thức của một instance của object (**this**).
- Synchronized block: khóa một phần của code trong một phương thức.
- Synchronized static method: khóa toàn bộ phương thức của một lớp (**class**).

Synchronized block giảm phạm vi khóa, cãi thiện performance (các luồng khác không phải chờ nếu truy cập vào các tài nguyên khác).

Ví dụ bên dưới tổng hợp lại các cách thực hiện đồng bộ hóa trong Java:

```
package com.gpcoder.sync;

/**
 * Java class to demonstrate use of synchronization method and block
 *
 * If you make any static method as synchronized, the lock will be o
 * not on object.
 *
 * Points to remember for Synchronized block: Synchronized block is
 * an object for any shared resource. Scope of synchronized block is
 * than the method.
 *
 */
public class SynchronizationExample {
    private static SynchronizationExample instance;

    // Synchronized Method: None-staic method
    public synchronized void lockedByThis() {
        System.out.println(" This synchronized method is locked by c
    }

    // Synchronized Method: Static method
    public static synchronized SynchronizationExample lockedByClassL
        System.out.println("This static synchronized method is locke
                + "i.e. SychronizationExample.class");
        if (instance == null) {
            instance = new SynchronizationExample();
```

```
28          }
29
30          return instance;
31      }
32
33      // Synchronized block
34      public static synchronized SynchronizationExample lockedBySynchr
35          System.out.println("This line is executed without locking");
36          if (instance == null) {
37              synchronized (instance) { // synchronized (Synchronizati
38                  // Thread Safe. Might be costly operation in some ca
39                  if (instance == null) {
40                      instance = new SynchronizationExample();
41                  }
42              }
43          }
44          return instance;
45      }
46
47  }
```

6. Phương thức wait(), notify(), notifyall()

Như đã nói ở phần Synchronized methods và Synchronized block, 2 cách này sẽ khóa đối tượng lại và cùng lúc chỉ có 1 luồng được làm việc với đối tượng, và khi có nhiều luồng muốn cùng làm việc thì các luồng phải làm việc theo thứ tự, khi luồng đang làm trả monitor đối tượng thì luồng tiếp theo nắm giữ monitor mới được sở hữu đối tượng và tiếp tục làm...

Vấn đề đặt ra rằng, giả sử, nếu trong quá trình luồng 1 khóa đối tượng b, sau khi thực hiện được 1 đoạn công việc, luồng 1 cần luồng 2 làm 1 việc gì đó trên đối tượng b này thì luồng 1 mới có đủ dữ liệu để làm tiếp thì sao?

Để giải quyết vấn đề này, Java cung cấp cho ta 3 phương thức: **wait()**, **notify()**, **notifyall()**.

Lưu ý: Các phương thức wait(), notify() and notifyAll() chỉ được gọi từ bên trong một phương thức được đồng bộ hóa (synchronized method).

6.1. wait()

Phương thức này sẽ làm cho luồng đang sở hữu monitor của đối tượng b (hay luồng đang khóa đối tượng b và nắm giữ đối tượng này) tạm thời ngưng hoạt động và trả monitor của b cho luồng khác. Sau khi trả monitor luồng 1 sẽ về trạng thái đợi (nằm ở vùng wait set. Trạng thái này java định nghĩa là **Thread.State.WAITING**).

6.2. notify() và notifyall()

Sau khi luồng 2 nắm giữ monitor của b và xử lý xong những gì luồng 1 cần, thì luồng 2 sẽ gọi phương thức notify() hoặc notifyall() trên đối tượng b để đánh thức các luồng đang chờ monitor của b và ngay sau đó luồng 2 sẽ trả lại monitor của b.

Điểm khác nhau giữa notify() và notifyall() là: notify() sẽ gửi thông điệp đánh thức cho 1 luồng ngẫu nhiên trong các luồng đang chờ, còn notifyall() sẽ gửi cho tất cả các luồng đang chờ b. Tuy nhiên, thông điệp gửi bởi notify() như đã nói, nó sẽ đánh thức 1 luồng bất kỳ chứ không chắc chắn là luồng 1 nên Oracle khuyến cáo nên dùng notifyall().

6.3. Ví dụ minh họa

Ví dụ mô phỏng hệ thống rút tiền ATM: khách hàng chỉ được rút tiền nếu số tiền rút nhỏ hơn số tiền hiện có trong tài khoản. Nếu số tiền rút lớn hơn số tiền hiện có thì phải chờ (wait) nạp đủ tiền mới được rút, sau khi nạp tiền thì thực hiện thông báo (notify) có thể tiếp tục xử lý rút tiền.

Hãy xem đoạn code sau:

Customer.java

```
1  package com.gpcoder.sync.atm;
2
3  public class Customer {
4      private int balance = 1000;
5
6      public Customer() {
7          System.out.println("Tài khoản của bạn là " + balance);
8      }
9
10     public synchronized void withdraw(int amount) {
11         System.out.println("Đang thực hiện giao dịch rút tiền " + am
12         while (balance < amount) {
13             System.out.println("Không đủ tiền rút!!!");
14             try {
15                 wait(); // Chờ nạp tiền
16             } catch (InterruptedException ie) {
17                 System.out.println(ie.toString());
18             }
19         }
20         balance -= amount;
21         System.out.println("Rút tiền thành công. Tài khoản của bạn h
22     }
23
24     public synchronized void deposit(int amount) {
25         System.out.println("Đang thực hiện giao dịch nạp tiền " + am
26         balance += amount;
27         System.out.println("Nạp tiền thành công. Tài khoản của bạn h
```

```
28            notify(); // Thông báo đã nạp tiền
29        }
30
31    }
```

CustomerOperationExample.java

```
1   package com.gpcoder.sync.atm;
2
3   public class CustomerOperationExample {
4       public static void main(String[] args) {
5           final Customer c = new Customer();
6           Thread t1 = new Thread() {
7               public void run() {
8                   c.withdraw(2000);
9               }
10          };
11          t1.start();
12
13          Thread t2 = new Thread() {
14              public void run() {
15                  c.deposit(500);
16                  try {
17                      Thread.sleep(2000);
18                  } catch (InterruptedException e) {
19                      e.printStackTrace();
20                  }
21                  c.deposit(3000);
22              }
23          };
24          t2.start();
25      }
26  }
```

Thực thi chương trình trên:

```
1  Tài khoản của bạn là 1000
2  Đang thực hiện giao dịch rút tiền 2000...
3  Không đủ tiền rút!!!
4  Đang thực hiện giao dịch nạp tiền 500...
5  Nạp tiền thành công. Tài khoản của bạn hiện tại là 1500
6  Không đủ tiền rút!!!
7  Đang thực hiện giao dịch nạp tiền 3000...
8  Nạp tiền thành công. Tài khoản của bạn hiện tại là 4500
9  Rút tiền thành công. Tài khoản của bạn hiện tại là 2500
```

7. Deadlock (Khoá chết) là gì?

Deadlock (hay khóa chết) xảy ra khi 2 tiến trình đợi nhau hoàn thành, trước khi chạy. Kết quả của quá trình là cả 2 tiến trình không bao giờ kết thúc.

Ví dụ:

DeadlockThread.java

```
1  package com.gpcoder.sync.deadlock;
2
3  public class DeadlockThread implements Runnable {
4      private Object obj1;
5      private Object obj2;
6
7      public DeadlockThread(Object o1, Object o2) {
8          this.obj1 = o1;
9          this.obj2 = o2;
10     }
11
12     @Override
13     public void run() {
14         String name = Thread.currentThread().getName();
15         System.out.println(name + " acquiring lock on " + obj1);
16
17         synchronized (obj1) {
18             System.out.println(name + " acquired lock on " + obj1);
19             work();
20
21             System.out.println(name + " acquiring lock on " + obj2);
22             synchronized (obj2) {//Avoid nested lock
23                 System.out.println(name + " acquired lock on " + obj
24                 work();
25             }
26
27             System.out.println(name + " released lock on " + obj2);
28         }
29
30         System.out.println(name + " released lock on " + obj1);
31         System.out.println(name + " finished execution.");
32     }
```

```
33
34      private void work() {
35          try {
36              Thread.sleep(6000);
37          } catch (InterruptedException e) {
38              e.printStackTrace();
39          }
40      }
41  }
```

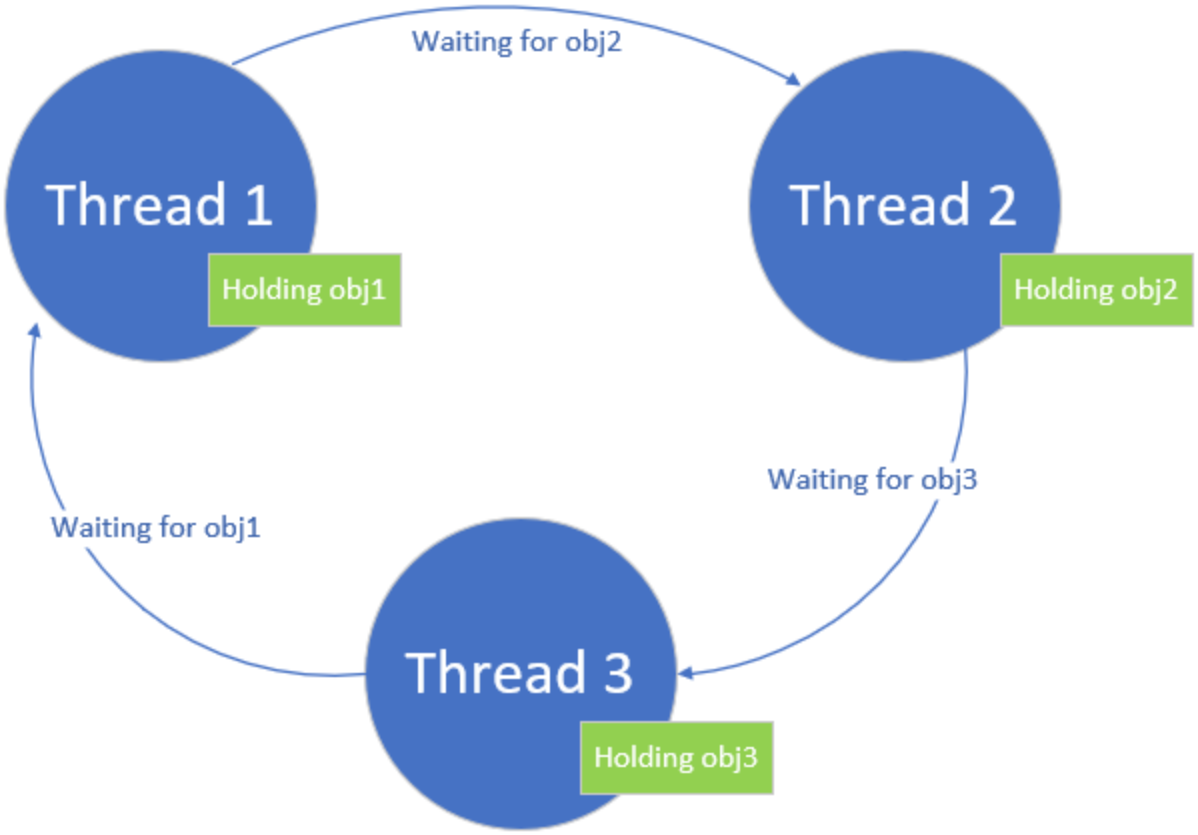
DeadLockTest.java

```
1   package com.gpcoder.sync.deadlock;
2
3   public class DeadLockTest {
4
5       public static void main(String[] args) throws InterruptedExcepti
6           Object obj1 = new String("obj1");
7           Object obj2 = new String("obj2");
8           Object obj3 = new String("obj3");
9
10          Thread t1 = new Thread(new DeadlockThread(obj1, obj2), "t1")
11          Thread t2 = new Thread(new DeadlockThread(obj2, obj3), "t2")
12          Thread t3 = new Thread(new DeadlockThread(obj3, obj1), "t3")
13
14          t1.start();
15          Thread.sleep(1000);
16
17          t2.start();
18          Thread.sleep(1000);
19
20          t3.start();
21      }
22  }
```

Thực thi chương trình trên, do cả 3 luồng đang chờ lẫn nhau nên chương trình không bao giờ kết thúc (deadlock).

- Luồng t1 đang khóa obj1 và yêu cầu sử dụng obj2.
- Luồng t2 đang khóa obj2 và yêu cầu sử dụng obj3.
- Luồng t3 đang khóa obj3 và yêu cầu sử dụng obj1.

```
1   t1 acquiring lock on obj1
2   t1 acquired lock on obj1
3   t2 acquiring lock on obj2
4   t2 acquired lock on obj2
5   t3 acquiring lock on obj3
6   t3 acquired lock on obj3
7   t1 acquiring lock on obj2
8   t2 acquiring lock on obj3
9   t3 acquiring lock on obj1
```



8. Tại sao không nên dùng phương thức Thread.stop()

Kết thúc luồng bằng phương thức stop() nó sẽ ném ra 1 ngoại lệ ThreadDeath, ngoại lệ này sẽ làm hàm run() bị cắt ngang khiến luồng kết thúc.

Sau khi luồng kết thúc, tất cả các monitor (Đã đề cập monitor ở phần trước) sẽ được gỡ bỏ 1 cách bất thường, nếu bất kỳ các đối tượng nào trước đó đang được monitor bảo vệ trước sự tranh chấp của các luồng đang cùng muốn sử dụng sẽ bị rơi vào trạng thái không thể kiểm soát sự đồng bộ, đối tượng này là 1 đối tượng "hư (hỏng)" . Lúc này, các luồng sẽ tùy tiện sử dụng đối tượng mà không hề có 1 sự kiểm soát nào. Do đó, Oracle cho rằng phương thức stop() thực sự không an toàn trong kỹ thuật đa luồng.

Không chỉ như vậy, khi ThreadDeath được ném ra bởi phương thức stop(), ta không thể bắt được ngoại lệ này, chương trình sẽ kết thúc 1 cách đột ngột trong âm thầm mà người dùng không hề nhận được bất kỳ thông báo nào cả. Theo Oracle, việc xử lý bắt ngoại lệ ThreadDeath chỉ có thể nói trên lý thuyết và không khả thi với thực tế vì rất phức tạp.

9. Tại sao không nên dùng phương thức Thread.suspend(), Thread.resume()

Khi 1 luồng bị suspend(), nó sẽ ngưng hoạt động và ngưng được cấp CPU, điều này dẫn đến việc các monitor mà luồng này đang nắm giữ cũng sẽ bị ngưng theo luồng đang nắm giữ nó. Như vậy, trong suốt quá trình luồng này ngưng hoạt động, không có 1 luồng nào khác được quyền truy cập tới các dữ liệu đang bị luồng bị suspend khóa cho đến khi luồng này được 1 luồng khác gọi phương thức resume()

=> Dẫn đến việc các luồng đang cần dữ liệu mà luồng bị suspend khóa cũng bị dừng theo, nếu luồng đang chờ lại là luồng sẽ làm công việc resume() luồng đang khóa thì Deadlock sảy ra.

10. Tại sao không nên dùng phương thức Thread.Destroy()?

So với suspend() thì Destroy() giống gần như hoàn toàn, Khi Destroy() luồng cũng sẽ không được cấp CPU nữa, có 1 điều cần lưu ý là: Sau khi luồng bị suspend() thì có thể cho luồng tiếp tục chạy bằng phương thức resume() nhưng với destroy() thì sẽ không có phương thức nào để luồng được cấp CPU lại cả. Như vậy, Destroy còn nguy hiểm hơn cả stop() và suspend() vì:
+ Tài nguyên của luồng không được giải phóng
+ Nếu luồng đang giữ 1 monitor nào đó của 1 đối tượng, mà đối tượng này đang được chờ bởi 1 luồng khác => Phải chờ vô thời hạn vì luồng giữ monitor sẽ không bao giờ chạy lại => deadlock sảy ra.

Tài liệu tham khảo:

- https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html
- http://www.java67.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html

Chuyên mục: **Multi-Thread**
Được gắn thẻ: **Multithreading**

« Luồng Daemon (Daemon Thread) trong Java

Hướng dẫn tạo và sử dụng ThreadPool trong Java »

10.1. Có thể bạn muốn xem:

- Sử dụng Fork/Join Framework với ForkJoinPool trong Java **(06/03/2018)**
- Semaphore trong Java **(18/09/2019)**
- Sử dụng CyclicBarrier trong Java **(13/03/2018)**
- Lập trình đa luồng với CompletableFuture trong Java 8 **(19/07/2018)**
- Lập trình đa luồng trong Java (Java Multi-threading) **(12/02/2018)**

10.2. Bình luận

1 bình luận

1 bình luận Sắp xếp theo **Mới nhất**

Thêm bình luận...

Vũ Đạt

Thấy đồng bộ là em nghĩ đây là kiến thức cơ bản ngại đọc nhưng vì các bài luồng khác hay quá nên theo dõi

Đọc rồi thấy còn nhiều cái mình chưa biết đặc biệt nhất là cái Synchronized block nó hữu ích với em

Một lần nữa mong anh đừng bỏ chủ đề luồng,làm thêm nhiều về nó ví dụ lỗi hay mắc phải,cách chữa ...vvv...

Thích · Phản hồi · 3 · 3 năm

Tìm kiếm

Từ khóa …

Bài viết mới

- Giới thiệu CloudAMQP – Một RabbitMQ server trên Cloud 02/10/2020
- Kết nối RabbitMQ sử dụng Web STOMP Plugin 19/06/2020
- Sử dụng publisher confirm trong RabbitMQ 16/06/2020
- Sử dụng Dead Letter Exchange trong RabbitMQ 13/06/2020
- Sử dụng Alternate Exchange trong RabbitMQ 10/06/2020

Xem nhiều

- Hướng dẫn Java Design Pattern – Factory Method (52289 *lượt xem*)
- Lập trình đa luồng trong Java (Java Multi-threading) (51678 *lượt xem*)
- Hướng dẫn Java Design Pattern – Singleton (50723 *lượt xem*)
- Xây dựng ứng dụng Client-Server với Socket trong Java (47856 *lượt xem*)
- Giới thiệu Design Patterns (46196 *lượt xem*)

Nội dung bài viết

- 1 Đối tượng khóa

- 2 Đồng bộ là gì? Tại sao lại cần đồng bộ?
  - 2.1 Tại sao cần đồng bộ?
  - 2.2 Đồng bộ hóa là gì?
- 3 Java monitor là gì? Cách hoạt động của java monitor?
  - 3.1 Java monitor là gì ?
  - 3.2 Cách hoạt động của java monitor
- 4 Các cách để đồng bộ trong Java
  - 4.1 Synchronized methods
  - 4.2 Synchronized statements/ Synchronized Block
  - 4.3 Static synchronized method
- 5 So sánh các cách synchronized
- 6 Phương thức wait(), notify(), notifyall()
  - 6.1 wait()
  - 6.2 notify() và notifyall()
  - 6.3 Ví dụ minh họa
- 7 Deadlock (Khoá chết) là gì?
- 8 Tại sao không nên dùng phương thức Thread.stop()
- 9 Tại sao không nên dùng phương thức Thread.suspend(), Thread.resume()
- 10 Tại sao không nên dùng phương thức Thread.Destroy()?

Lưu trữ

Chọn tháng

Thẻ đánh dấu

Annotation Authentication Basic Java Behavior Pattern Collection Creational Design Pattern Cấu trúc điều khiển Database Dependency Injection Design pattern Eclipse Exception Executor Service Google Guice Gson Hibernate How to Interceptor IO Jackson Java 8 Java Core JDBC JDK Jersey JMS JPA json JUnit JWT Message Queue Mockito Multithreading OOP Performance PowerMockito RabbitMQ Reflection Report REST SOAP Structuaral Pattern Thread Pool Unit Test Webservice

Liên kết website

Design Pattern

- Refactoring Guru
- Source Making

Lập trình Java

Giới thiệu

GP Coder là trang web cá nhân, được thành lập với mục đích lưu trữ, chia sẽ kiến thức đã học và làm việc của tôi. Các bài viết trên trang này chủ yếu về ngôn ngữ Java và các công nghệ có liên quan đến Java như: Spring, JSF, Web Services, Unit Test, Hibernate, SQL, ...
Hi vọng góp được chút ít công sức cho sự phát triển cộng đồng Coder Việt.

- JavaTpoint
- JavaWorld
- Journaldev
- TutorialsPoint
- W3Schools Online Web Tutorials

Tìm kiếm các bài viết của GP Coder với Google Search

Liên hệ

Các bạn có thể liên hệ với tôi thông qua:

- Trang liên hệ
- Linkedin: gpcoder
- Email: contact@gpcoder.com
- Skype: ptgiang56it

Follow me