**Thợ lành nghề #10: Những thread lửng lơ (Vòng lặp không hạn chế)**

Bạn có thể tải mã nguồn của bài viết trước ở [đây](http://www.objectmentor.com/resources/articles/CraftsmanCode/SDSocketServiceR4_TestMultiThreaded.zip).

Ngày 14 tháng 1 năm 2003.

Hàng tháng tôi dùng điểm tâm một lần ở đài quan sát. Ðây là điều hoang phí với túi tiền của một tay học việc như tôi, nhưng tôi khoái ăn dưới vòm trời mở rộng.

Trong lúc ăn, tôi ngẫm nghĩ về những thread treo lủng lẳng mà chúng tôi đã giải quyết ngày hôm qua. Chúng tôi sửa cái serverThread nhưng lại để toàn bộ các thread thuộc serviceRunnable bị treo lửng lơ. Tôi biết thế nào Jerry cũng muốn sửa mấy cái ấy cho sớm.

Ðúng như vậy, ngay khi tôi bước vào phòng làm việc, Jerry đã đưa ra mấy cái kiểm thử trên màn hình như sau:

public void testAllServersClosed() throws Exception {

ss.serve(999, new WaitThenClose());

Socket s1 = new Socket(&quot;localhost&quot;, 999);

Thread.sleep(20);

assertEquals(1, WaitThenClose.threadsActive);

ss.close();

assertEquals(0, WaitThenClose.threadsActive);

}

“À, để tôi xem ông đang làm gì. Ông đang đảm bảo rằng tất cả những SocketServer được đóng hết ngay khi trở lại từ bước đóng SockeService,” tôi nói.

“Tao muốn chắc ăn là mình không để cho mấy cái servers đó bị treo lủng lẳng như thế,” Jerry trả lời.

“Nhưng ông chỉ kiểm tra nó với một server thôi mà. Bộ mình không cần kiểm tra với nhiều server hay sao?”

“Ðúng thế!” Jerry mỉm cười. “Nhưng hãy làm xong ngon lành cái kiểm thử này cái đã.”

“Được thôi,” tôi trả lời. “Tôi biết cách viết WaitThenClose ra sao rồi.”

class WaitThenClose implements SocketServer {

public static int threadsActive = 0;

public void serve(Socket s) {

threadsActive++;

delay();

threadsActive--;

}

private void delay() {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

}

}

}

Jerry gật gù trong lúc mã nguồn của tôi hiện ra trên màn hình; cái WaitThenClose của tôi đúng y như gã dự tưởng. Tôi biên dịch mã nguồn này và chạy các kiểm thử, chúng thất bại như dự đoán:

1) testAllServersClosed AssertionFailedError: expected: but was:

Jerry xoa tay và nói, “bây giờ hãy làm cho nó đạt đi.” Gã với lấy bàn phím nhưng tôi cản gã lại. “Tôi nghĩ là tôi có một ý kiến”. Thế nên, trong khi Jerry quan sát, tôi thay đổi đoạn mã như sau:

private LinkedList serverThreads = new LinkedList();

public void serve(int port, SocketServer server) throws Exception {

itsServer = server;

serverSocket = new ServerSocket(port);

serverThread = new Thread(new Runnable() {

public void run() {

running = true;

while (running) {

try {

Socket s = serverSocket.accept();

Thread serverThread = new Thread(new ServiceRunnable(s));

serverThreads.add(serverThread);

serverThread.start();

} catch (IOException e) {

}

}

}

});

serverThread.start();

}

public void close() throws Exception {

if (running) {

running = false;

serverSocket.close();

serverThread.join();

for (Iterator i = serverThreads.iterator(); i.hasNext();) {

Thread thread = (Thread) i.next();

serverThreads.remove(thread);

thread.join();

}

} else {

serverSocket.close();

}

}

Khi mã nguồn được biên dịch, Jerry nhăn nhó.

“Vậy được không?” tôi hỏi.

“Hãy xem nào,” gã trả lời. “Chạy thử cái kiểm thử xem sao.”

Khi chạy kiểm thử, bị ngay một lỗi khác thường:

1) testOneConnection java.util.ConcurrentModificationException

“Cái gì vậy?” tôi hỏi.

“Mày đã phạm quy đó, Alphonse,” Jerry nói. “Không bao giờ thêm hoặc bớt từ một list trong khi mày chạy trong một iterator (biến lặp) hoạt động.”

“Tất nhiên rồi!” tôi nói một cách ngượng ngùng. “Được thôi, nhưng chuyện này dễ sửa thôi, bởi vì tôi không cần phải tháo bỏ cái cái thread từ list.” Tôi bỏ dòng remove và chạy lại kiểm thử.

“À! bây giờ thì nó chạy.”

Jerry gật đầu nhưng bắt đầu nhìn tôi chờ đợi một điều gì đó.

“Gì hở?” tôi gào lên sau nửa phút chịu đựng kiểu nhìn của gã.

“Mày vẫn đang thay đổi cái list trong khi iterator hoạt động,” gã phán.

“Vậy sao?” tôi quả thật bối rối. “Chỉ có một nơi duy nhất cái list được thay đổi, và đó là nơi thread được thêm vào trong vòng lặp running. Làm sao nó được gọi trong khi iterator đang hoạt động?”

“Có thể được,” Jerry nói. “Lời gọi để tiếp nhận có thể ở tình trạng sẵn sàng trở lại ngay khi mày đi vào iterator. Khi iterator chặn một cú nối (join), phần tiếp nhận sẽ trở lại và thêm một thread nữa vào list.”

“OK, nhưng mình kiểm tra chuyện đó được không?” tôi hỏi.

“Mình có thể làm được chuyện đó nhưng chẳng ích gì,” Jerry trả lời. “Hoá ra ở một nơi khác nơi mày sẽ thay đổi cái list trong khi iterator mở ra.”

“Có à?”

“Ừ, mày sắp sửa thêm nó vào đó. Có bao nhiêu thread trong list đó vậy?”

“Tất cả chúng… ôi!” tôi vỗ trán. “Tôi nên bỏ cái thread ra khỏi list khi nó đã hoàn thành! không thì, các thread đã hoàn tất sẽ bị treo trong list.” Tôi vớ lấy bàn phím và thay đổi như sau:

class ServiceRunnable implements Runnable {

private Socket itsSocket;

ServiceRunnable(Socket s) {

itsSocket = s;

}

public void run() {

try {

itsServer.serve(itsSocket);

serverThreads.remove(Thread.currentThread());

itsSocket.close();

} catch (IOException e) {

}

}

}

“À hà, bây giờ nó lại hỏng tiếp,” tôi nói. “Ông nói đúng – Một vài thread hoàn tất trước khi iterator kết thúc. Cha chả, iterator “ý kiến” với các cập nhật liên đới quả là điều thật hay!”

“Ðúng thế,” Jerry gật đầu. “Bây giờ để tao chỉ mày cách tao trị nó như thế nào.”

Lại một lần nữa Jerry dành lấy bàn phím, và cũng lại một lần nữa tôi không phản đổi. Jerry tiếp tục thay đổi như sau:

public void close() throws Exception {

if (running) {

running = false;

serverSocket.close();

serverThread.join();

while (serverThreads.size() &gt; 0) {

Thread t = (Thread) serverThreads.get(0);

serverThreads.remove(t);

t.join();

}

} else {

serverSocket.close();

}

}

“Đó!” Jerry nói. “Bây giờ thì mấy kiểm thử hẳn phải đạt.”

“Tôi biết rồi,” tôi thốt ra. “Thay vì dùng iterator, ông chỉ kéo phần tử thứ nhất ra khỏi list và tiếp tục lặp lại cho đến khi list trống rỗng.”

“Ðúng đó,” Jerry trả lời. “Bằng cách đó, không có iterator nào mở ra quá lâu. Các cú nối (joins) có thể mất thời gian, cho nên để vòng lặp mở quá lâu khi các thread khác thay đổi list là điều không hay.”

“Thế, mình xong việc rồi sao?”

Jerry lắc đầu. “Không, vẫn còn hiểm nguy,” gã cảnh báo.

“Ý ông thế nào vậy? Thế còn có sự cố gì nữa đây?”

“Alphonse, mỗi khi mày có một bộ chứa (container) bị nhiều thread thay đổi, rất có khả năng hai thread xung đột với nhau bên trong bộ chứa. Một thread có thể thêm một phần tử trong khi một thread khác lại xoá phần tử khác. Khi trường hợp này xảy ra, bộ chứa có thể bị hỏng và những chuyện kỳ quái có thể xảy ra.”

“Vậy ý ông là mình nên đồng bộ hoá truy cập đến bộ chứa?” tôi hỏi.

“Chính xác,” Jerry trả lời. “Chúng ta cần nắm chắc không có thread nào khác có thể truy cập bộ chứa trong khi nó bị thay đổi.”

“Ðơn giản thôi,” tôi nói trong khi gom lại đoạn add và hai đoạn remove với lệnh sychronized (serverThreads) {…}. Tôi chạy mấy kiểm thử và chúng đạt hết.

“Ðó là một cách,” Jerry nói với nụ cười trên mặt, “nhưng nó hơi bị dễ dính lỗi. Nếu có ai chỉnh sửa mã nguồn và đặt vào một cái add hay remove, họ phải nhớ đặt phần đồng bộ hoá vào. Nếu họ quên, những chuyện tồi tệ có thể xảy ra.”

Tôi ngẫm nghĩ vấn đề ấy vài phút và xác định gã nói đúng – nếu chúng ta không cần phải gom các lệnh điều khiển list với lệnh synchronized thì có lẽ tốt hơn. “Thế cách nào tốt hơn vậy?”

“Tao chỉ cho mày xem.” Gã lấy bàn phím và gỡ bỏ các dòng synchronized của tôi. Sau đó gã thay đổi thêm một dòng mã nữa – dòng tạo LinkedList ngay lúc đầu:

private List serverThreads = Collections.synchronizedList(new LinkedList());

Jerry biên dịch mã nguồn và chạy toàn bộ các cái kiểm thử. Mọi sự ổn cả. Sau rồi gã hỏi, “Mày biết gì về mẫu thiết kế hả Alphonse? Có bao giờ mày nghe đến mẫu Decorator (trang trí) chưa?”

“Tất nhiên là tôi nghe về chúng rồi, và tôi cũng thấy sách nói về chuyện này trên giá sách của thiên hạ, nhưng tôi không biết nhiều lắm về chúng.”

Jerry nhìn tôi nghiêm khắc nói, “vậy thì đến lúc nên bắt đầu học về chúng một cách nghiêm chỉnh đi. Mày có thể mượn sách của tao và nghiên cứu nó nếu thích. Ðầu tiên tao muốn mày đọc chương nói về mẫu Decorator. Hàm synchronizedList mình vừa gọi để gói cái LinkedList trong một Decorator. Mọi lời gọi đến LinkedList đều được nó đồng bộ hoá cả.”

“Nghe đúng là một giải pháp hay,” tôi đáp.

“Ừ, mà mày cũng phải nhớ đồng bộ hoá cụ thể những nơi dùng iterator.” Jerry cau mày.

“Vậy sao?” Tôi hỏi. “Ý ông vòng lặp không được đồng bộ hoá trong danh sách đồng bộ sao?”

“TANSTAAFL,” gã trả lời.

“Hở?” tôi hỏi, thộn người ra. Không biết có phải gã nói tiếng Clangrish hay gì đây.

“TANSTAAFL,” gã lặp lại theo kiểu khống chế; rồi gã mỉm cười. “There Ain’t No Such Thing As A Free Lunch” (Không hề có cái gọi là buổi ăn trưa miễn phí).

“Tôi biết,” tôi mỉm cười trong khi rảo bước về phòng của mình.

**Bài tiếp:** [Thợ lành nghề #11: Dùng hàm main để làm gì? (SMCRemote – phần 1)](https://codegym.vn/blog/2018/08/09/tho-lanh-nghe-11-dung-ham-main-de-lam-gi-smcremote-phan-1/)

**Bài trước:** [Thợ lành nghề #9: Những thread nguy hiểm (Dịch vụ Socket 4)](https://codegym.vn/blog/2018/08/07/tho-lanh-nghe-9-nhung-thread-nguy-hiem-dich-vu-socket-4/)

**Tác giả:** Robert C. Martin

**Người dịch:** Hoàng Ngọc Diêu (conmale)