***Thợ lành nghề #9: Những thread nguy hiểm (Dịch vụ Socket 4)***

*Câu chuyện tay học việc trẻ tuổi của chúng ta học được bài nằm lòng: Không để các thread đeo lủng lẳng – phải nắm chắc bạn kiểm soát bước kết thúc cũng như điểm khởi tạo của chúng.*

*Bạn có thể tải mã nguồn mà chúng ta có được ở* [*phần trước*](http://tapchilaptrinh.vn/2013/05/07/tho-lanh-nghe-8-kiem-thu-la-mot-dang-tai-lieu-dich-vu-socket-3/) *ở* [*đây*](http://www.objectmentor.com/resources/craftman/SDSocketServiceR3_SendAndReceive.zip)*.*

Ngày 20 tháng 12 năm 2002.

Sáng nay chiếc PDA khe khẽ đánh thức tôi dậy. Cố rũ cơn ngái ngủ ra khỏi đầu, tôi tắt báo thức và mò vào phòng tắm. Trong khi vòi phun kỳ cọ và xoa bóp thân thể, tâm trí tôi vẩn vơ lạc vào những sự kiện ngày hôm trước.

…Tôi trở lại phòng làm việc lại sau thời gian giải lao, trong đầu vẫn nghĩ ngợi về giá trị tài liệu của kiểm thử. Jerry đang đợi tôi, gã nói: “Tao mừng là mày trở lại. Tao đang hoàn tất trường hợp kiểm thử kế tiếp đây. Hãy nhìn qua và thử đoán mục đích của nó là gì.”

public void testMultiThreaded() throws Exception {

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

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

BufferedReader br = SocketService.getBufferedReader(s1);

PrintStream ps = SocketService.getPrintStream(s1);

Socket s2 = new Socket(&quot;localhost&quot;, 999);

BufferedReader br2 = SocketService.getBufferedReader(s2);

PrintStream ps2 = SocketService.getPrintStream(s2);

ps2.println(&quot;MyMessage&quot;);

String answer2 = br2.readLine();

s2.close();

ps.println(&quot;MyMessage&quot;);

String answer = br.readLine();

s1.close();

assertEquals(&quot;MyMessage&quot;, answer2);

assertEquals(&quot;MyMessage&quot;, answer);

}

“Nó hơi phức tạp một chút nhưng hình như ông muốn chứng minh là SocketService có thể làm việc với hai kết nối cùng lúc.”

“Ðúng vậy,” Jerry trả lời. “Mày có nhận ra là kết nối thứ nhất lại đóng sau cùng không?”

“Không, nhưng khi ông nói thì tôi thấy rồi. Ông làm thế để làm chi vậy?”

“Tao muốn hai phiên truy cập cùng mở liên tục,” Jerry đáp.

“Tại sao?”

Jerry tò mò nhìn tôi và nói: “Bởi khi ấy phương thức serve trong lớp SocketService sẽ phải đi vào hai lần trong hai thread khác nhau, trước khi một trong hai có cơ hội kết thúc,” Jerry tiếp tục. “Khi một hàm được gọi vào hơn một lần trước khi nó kết thúc, cái này gọi là *reentrant* (kẻ vào lại).”

“Nhưng sao ông lại muốn kiểm thử nó làm gì?”

“Bởi vì các hàm reentrant thường đem lại cho mình những sự cố rất lý thú.”

Tôi không hiểu nổi điều này nhưng tôi biết chắc rốt cuộc Jerry sẽ giải thích vấn đề.

“OK” gã nói. “Hãy chạy cái kiểm thử đi.”

Tôi biên dịch và chạy cái kiểm thử. Thanh chỉ định màu xanh lá chuyển động nhẹ nhàng xuyên qua khung kiểm thử cho chúng tôi biết rằng trọn bộ những kiểm thử trước đây vẫn làm việc ngon lành. Thế rồi, trước khi kết thúc, chương trình bị khựng lại. Tôi đợi vài giây xem thử nó có “tỉnh lại” và hoàn tất hay không, nhưng nó bị treo hoàn toàn luôn.

“Hừm…” Tôi thốt lên một cách thông minh.

“Ừ, phải rồi”, Jerry nói. “Mày nghĩ xem có gì sai ở đây?”

Sau khi nghiên cứu mã nguồn ở đoạn SocketService.serve chừng một phút, tôi nói, “Ồ, thật dễ. Hãy xem đoạn lặp này.”

while (running) {

try {

Socket s = serverSocket.accept();

itsServer.serve(s);

s.close();

} catch (IOException e) {

}

}

“itsServer.serve không trở lại để bắt lấy kết nối thứ nhì. Kết nối thứ nhất bị treo trong khi đoạn EchoServer đợi mình gửi đến một thông điệp. Bởi thế chúng ta không bao giờ đi hết vòng lặp để gọi accept cho socket thứ hai kết nối tới.”

Jerry cười rạng rỡ. “Khá lắm! bây giờ mình làm sao với nó đây?”

“Chúng ta cần đưa itsServer.serve vào trong thread riêng của nó để vòng lặp đó có cơ hội trở lại mà không phải đợi nó.”

“Lại đúng lần nữa! Dám chọc nó một phát không?”

Tôi vớ lấy bàn phím và đổi phương thức SocketService.serve như sau:

while (running) {

try {

Socket s = serverSocket.accept();

new Thread(new ServiceRunnable(s)).start();

} catch (IOException e) {

}

}

Kế tiếp tôi thêm một một lớp lồng nhau mới bên trong SocketService gọi là ServiceRunnable:

class ServiceRunnable implements Runnable {

private Socket itsSocket;

ServiceRunnable(Socket s) {

itsSocket = s;

}

public void run() {

try {

itsServer.serve(itsSocket);

itsSocket.close();

} catch (IOException e) {

}

}

}

“Vậy là đủ rồi,” tôi nói. Tôi nhấn nút kiểm thử và được trả công bằng kết quả mĩ mãn.

“Nhấn nút thêm vài lần xem sao,” Jerry đề nghị.

“Ôi không, đừng chơi mấy trò này,” tôi phàn nàn, nhớ đến cái trục trặc ở lần đầu tiên chúng tôi bắt đầu. Tôi miễn cưỡng chạy phần kiểm thử thêm vài lần. Hiển nhiên tôi thấy ngay vấn đề phát sinh:

testMultiThreaded(TestSocketServer)

java.lang.NullPointerException

at SocketService.close(SocketService.java:32)

at TestSocketServer.tearDown(TestSocketServer.java:30)

“Quỷ tha ma bắt gì đây?” tôi nhăn nhó nhìn dòng 32 của lớp SocketService.java

public void close() throws Exception {

running = false;

serverSocket.close();

}

“Đợi một phút,” tôi chống chế. “Làm sao có thể bị **null pointer exception** chỗ đó được cơ chứ?” Tôi kéo lên phần TestSocketServer ở dòng 30:

public void tearDown() throws Exception {

ss.close();

}

“Bây giờ hãy dừng một phút ở chỗ đáng nguyền rủa này”. Tôi nhắc lại “Vô lý. TearDown đóng SockerService như giả định nhưng cái serverSocket lại null là thế nào? Nếu serverSocket là null thì mình đã dính ngay lỗi từ đoạn testMultiThreaded chớ không phải trong đoạn tearDown.”

Jerry hẳn cảm thấy có lý bởi gã nói, “Ừa.”

“Jerry, cái quỷ gì đây? chẳng nghĩa lý gì cả”

“Máy đã nói thế mà”

“Nhưng nó đúng mà. Biến serverSocket không thể là null được.”

“Alphonse. Hãy suy nghĩ một chút đi. Trạng thái các thread thế nào?”

“Hở?”

“Các thread. Chúng làm gì khi tearDown được gọi?”

Tôi suy nghĩ vấn đề này chừng một phút. Rõ ràng các kiểm thử đơn vị đã đạt; không thì tearDown đã không được gọi. Ðiều này có nghĩa là cả hai kết nối socket được tiếp nhận và serverThread đã đi xuyên vòng lặp hai lần. serverThread có thể chặn cuộc gọi tiếp nhận lần thứ ba hoặc nó chưa trở lại hàm khởi động dùng để kích hoạt thread ServiceRunnable thứ hai.

Thread đầu của ServiceRunnable đã vào EchoServer, thread này đã đọc và viết thông điệp nhưng nó có thể chưa kết thúc. Nó có thể đợi println gửi thông điệp ngược lại từ trường hợp kiểm thử, nhưng thread thứ hai của ServiceRunnable hẳn có đủ thời gian để kết thúc. Nó đã nhận và gửi thông điệp của nó đã lâu.

Tôi giảng giải tất cả mọi điều với Jerry và gã lặng lẽ gật đầu.

“Ừ đúng,” gã nói. “Tao cũng phân tích như thế.”

“Vậy thì sao lại có null pointer exception?” tôi hỏi.

“Tao chả biết,” gã nói. “Nhưng sự thật là nó bị hỏng khi mình đóng cái serverSocket, do đó tao nghĩ là mình để cho một vài thread nào đó chạy làm ảnh hưởng đến thư viện socket.”

“Ý ông là có bug trong bộ thư viện socket?”.

Jerry chỉ dán mắt vào màn hình và nói, “Tao không chắc. Hãy làm thêm một vài thử nghiệm xem.”

**Bài tiếp:** [Thợ lành nghề #10: Những thread lửng lơ (Vòng lặp không hạn chế)](https://codegym.vn/blog/2018/08/08/tho-lanh-nghe-10-nhung-thread-lung-lo-vong-lap-khong-han-che/)

**Bài trước:** [Thợ lành nghề #8: Kiểm thử là một dạng tài liệu (Dịch vụ Socket 3)](https://codegym.vn/blog/2018/08/06/tho-lanh-nghe-8-kiem-thu-la-mot-dang-tai-lieu-dich-vu-socket-3/)

**Tác giả:** Robert C. Martin

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