07: Q41 – Q48 Scala Async and Actor System Interview Q&As

Posted on September 10, 2016 by Arulkumaran Kumaraswamipillai

Q41. What is the purpose of the “async” macro in Scala?
A41. The 2.11 and later versions of Scala include the ability to transform code during compilation by using macros. The “**async**” macro is one of them, which transforms sequential code that uses “Futures” into asynchronous code during compilation by modifying the AST (Abstract Syntax Tree). Here is an example of Future chaining using the “async” macro in the scala module

```
1
2       <dependency>
3           <groupId>org.scala-lang.modules</groupId>
4           <artifactId>scala-async_2.11</artifactId>
5           <version>0.9.5</version>
```

600+ Full Stack Java/JEE Interview Q&As ♥Free ♦FAQs

open all | close all

- Ice Breaker Interview
- Core Java Interview C
- JEE Interview Q&A (3
- Pressed for time? Jav
- SQL, XML, UML, JSC
- Hadoop & BigData Int
- Java Architecture Inte
- Scala Interview Q&As
  - Scala way of codin
    - 01: Coding Scala
    - 01: ♥ Q1 – Q6 Scal
    - 02: Q6 – Q12 Scala
    - 03: Q13 – Q18 Sca
    - 04: Q19 – Q26 Sca
    - 05: Q27 – Q32 Sca
    - 06: Q33 – Q40 Sca
    - 07: Q41 – Q48 Sca
    - 08: Q49 – Q58 Sca
    - 09: Q59 – Q65 Hig
    - 10: Q66 – Q70 Pat
    - 11: Q71 – Q77 – S

```
6       </dependency>
7
```

The revised code from the “SimpleFuture” Scala class.

```
1
2  import scala.concurrent.Await
3  import scala.concurrent.ExecutionContext
4  import scala.concurrent.ExecutionContext.Implici
5  import scala.concurrent.Future
6  import scala.concurrent.duration.DurationInt
7  import scala.async.Async._
8
9  object SimpleFuture extends App {
10
11    val x = 23;
12    val y = 12;
13
14    val a = 3;
15    val b = 2;
16
17    val sumFuture: Future[Int] = Future {
18      Thread.sleep(3000); // 3 seconds
19      x + y
20    }
21
22    val multiplyFuture: Future[Int] = Future {
23      Thread.sleep(4000); // 4 seconds
24      a * b
25    }
26
27    println("I am not blocked");
28
29    val resultFuture = async {
30        await(sumFuture) +  await(multiplyFuture)
31    }
32
33    val result = Await.result(resultFuture, Durati
34    println("After 4 seconds ....");
35    println("result = " + result); //41
36
37 }
38
```

Output:

```
1
2 I am not blocked
3 After 4 seconds ....
4 result = 41
5
```

Q42. What is the purpose of “Actors” in Scala?

A42. Scala **Actors** are basically concurrent processes that communicate by exchanging messages. They can exchange

12: Q78 – Q80 Rec
Spring, Hibernate, & M
Testing & Profiling/Sa
Other Interview Q&A f
Free Java Interviev

16 Technical Key Areas

open all | close all
Best Practice (6)
Coding (26)
Concurrency (6)
Design Concepts (7)
Design Patterns (11)
Exception Handling (3
Java Debugging (21)
Judging Experience In
Low Latency (7)
Memory Managemen
Performance (13)
QoS (8)
Scalability (4)
SDLC (6)
Security (13)
Transaction Managen

80+ step by step Java Tutorials

open all | close all
Setting up Tutorial (6)
Tutorial - Diagnosis (2
Akka Tutorial (9)
Core Java Tutorials (2
Hadoop & Spark Tuto
JEE Tutorials (19)
Scala Tutorials (1)

messages both synchronously and asynchronously. Actors may use **Futures** to handle requests asynchronously.

Q43. Can you write a producer & consumer scenario in Scala using Actors?
A43. Here is a very simple example where actors concurrently send messages (i.e. **multiple producers**) and the main thread receive those messages (i.e. **single consumer**).

You need the following dependency in your pom.xml

```
1
2  <dependency>
3      <groupId>org.scala-lang</groupId>
4      <artifactId>scala-actors</artifactId>
5      <version>2.11.5</version>
6  </dependency>
7
```

```
1
2  import scala.actors.Actor._
3
4  object ActorsInScala extends App {
5
6    val listOfNumbers: List[Int] = (1 to 3).toList
7    val consumer = self; //the main thread itself
8
9    //sender sending
10   listOfNumbers.foreach { element =>
11     actor {
12       val thread = Thread.currentThread().getNam
13       println(thread + " sending " + element);
14       consumer ! element //! means tell the send
15     }
16   }
17
18   //main consumer consuming
19   listOfNumbers.foreach { msg =>
20     val thread = Thread.currentThread().getName
21     receive {
22       case _ => println(thread  + " received " +
23     }
24   }
25
26 }
27
```

Output:

```
1
```

Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

100+ Java pre-interview coding tests

open all | close all
Can you write code? (
♦ Complete the given
Converting from A to I
Designing your classe
Java Data Structures
Passing the unit tests
What is wrong with th
Writing Code Home A
Written Test Core Jav
Written Test JEE (1)

How good are your .....?

open all | close all
Career Making Know-
Job Hunting & Resum

```
2 ForkJoinPool-1-worker-13 sending 1
3 ForkJoinPool-1-worker-9 sending 3
4 ForkJoinPool-1-worker-11 sending 2
5 main received 1
6 main received 2
7 main received 3
8
```

Q44. How will you go about modifying the above code so that the consumer runs on a separate worker thread instead of the main thread?

A44. As shown below.

```
1
2  import scala.actors.Actor._
3
4  object ActorsInScala extends App {
5
6    val listOfNumbers: List[Int] = (1 to 3).toList
7
8    val consumer = actor {
9      val thread = Thread.currentThread().getName
10
11     listOfNumbers.foreach { x =>
12       receive {
13         case msg => println(thread + " received
14       }
15     }
16   }
17
18   consumer.start();
19
20   //start sending
21   listOfNumbers.foreach { element =>
22     actor {
23       val thread = Thread.currentThread().getNam
24       println(thread + " sending " + element);
25       consumer ! element //! means tell the send
26     }
27   }
28
29 }
30
```

Output:

```
1
2 ForkJoinPool-1-worker-9 sending 3
3 ForkJoinPool-1-worker-11 sending 1
4 ForkJoinPool-1-worker-7 sending 2
5 ForkJoinPool-1-worker-13 received 2
6 ForkJoinPool-1-worker-13 received 1
7 ForkJoinPool-1-worker-13 received 3
8
```

Q45. Isn't the Scala actors implementation has been replaced by the **Akka Actor system** from Scala version 2.11?

A45. Yes. The **Akka toolkit** has become the de-facto standard for actor based development.

You need to import the Akka toolkit first in your pom.xml file.

```
1
2 <dependency>
3     <groupId>com.typesafe.akka</groupId>
4     <artifactId>akka-actor_2.11</artifactId>
5     <version>2.3.15</version>
6 </dependency>
7
```

Using the Akka ActorSystem

```
1
2  import akka.actor.Actor._
3  import akka.actor.ActorSystem
4  import akka.actor.Props
5
6  object ActorsInScala extends App {
7    val listOfNumbers: List[Int] = (1 to 3).toList
8
9    val system = ActorSystem("Print-Numbers")
10   val consumer = system.actorOf(Props(classOf[Co
11
12   //start sending
13   listOfNumbers.foreach { element =>
14      {
15       val thread = Thread.currentThread().getNam
16       println(thread + " sending " + element);
17       consumer ! element //! means tell the send
18     }
19   }
20
21   Thread.sleep(5000)
22   System.exit(0)
23 }
24
```

Consumer.scala

```
1
2  import akka.actor.Actor
3
4  class Consumer extends Actor {
5
6    def receive = {
7      case value : Int  => {
8        Thread.sleep(1000); //emulate some process
9        val thread = Thread.currentThread().getNam
```

```
10        println(thread + " received " + value)
11      }
12      case _ =>  //do nothing
13    }
14  }
15
```

Output:

```
1
2  main sending 1
3  main sending 2
4  main sending 3
5  Print-Numbers-akka.actor.default-dispatcher-2 rec
6  Print-Numbers-akka.actor.default-dispatcher-2 rec
7  Print-Numbers-akka.actor.default-dispatcher-2 rec
8
```

Q46. How will you create a number of actors to consume the message? In the above example, you only have a single thread “Print-Numbers-akka.actor.default-dispatcher-2” consuming the messages.

A46. Messages can be sent via a **router** to route them to relevant destination actors, known as the **routees**. A Router can be used inside or outside of an actor, and you can manage the routees with configuration.

Here is the revised “**ActorsInScala.scala**” with a router and routees.

```
1
2  import akka.actor.Actor._
3  import akka.actor.ActorSystem
4  import akka.actor.Props
5  import akka.routing.ActorRefRoutee
6  import akka.routing.RoundRobinRoutingLogic
7  import akka.routing.Router
8  import akka.actor.Actor
9
10 object ActorsInScala extends App {
11   val listOfNumbers: List[Int] = (1 to 3).toList
12
13   val system = ActorSystem("Print-Numbers")
14
15   val router = {
16      val routees = Vector.fill(3) {
17         val consumer = system.actorOf(Props(clas
18         ActorRefRoutee(consumer)
19      }
20
21      Router(RoundRobinRoutingLogic(), routees)
22   }
23
```

```
24    //start sending
25    listOfNumbers.foreach { element =>
26       {
27        val thread = Thread.currentThread().getNam
28        println(thread + " sending " + element);
29        router.route(element, Actor.noSender)
30      }
31    }
32
33    Thread.sleep(5000)
34    System.exit(0)
35  }
36
```

Output:

```
1
2  main sending 1
3  main sending 2
4  main sending 3
5  Print-Numbers-akka.actor.default-dispatcher-2 rec
6  Print-Numbers-akka.actor.default-dispatcher-3 rec
7  Print-Numbers-akka.actor.default-dispatcher-4 rec
8
```

As you can see, there are 3 different consumer threads (i.e. actors) consume the messages.

Q47. Can you explain the following statement?

"Don't use actors for concurrency. Instead, use actors for state and use futures for concurrency."

A47. It is an anti-pattern to use the **actor system** as a tool for flow control or concurrency. An "Actor system" is useful for maintaining state and providing a messaging endpoint.

Spinning up multiple actors with a router as shown in the above trivial example without any states can give concurrency, but this approach can over complicate the coding, when it can be implemented using the **Futures**.

Actors need to be used for **managing state**. Here is a simple "**Counter.scala**" example. The state is maintained with the "count" variable.

```
1
```

```
import akka.actor.Actor

class Counter extends Actor{

  var count: Long = 0

  def receive = {
    case Request => { count += 1}
    case Response => { sender ! count }
  }
}

case class Request(body:String) {}
case class Response(body:String) {}


```

Another important use for actors is to be a **message endpoint**. For example, an HTTP endpoint, a tcp endpoint, etc.The **Akka remoting** is a separate jar file.

```

"com.typesafe.akka" %% "akka-remote" % "2.4-SNAPS

```

Q48. Can you explain the following statement?

Akka is the Ideal Runtime for Building Reactive Applications on the JVM

A48. A **reactive programming** is programming with asynchronous data streams. The reactive programming paradigm is **message-driven**. A reactive programming is about writing code that define how to react to changes like user input, data coming from a stream, a change in the state of a system, etc. Actors are a good technology for reactive systems that need to process and link many different concurrent outside events.

It is easier to implement reactive programming in functional programming languages like Scala. You can create, combine, and filter streams via functional programming. A stream can be used as an input to another one. You can merge streams to create a new stream or filter streams to get the messages you are interested in.

Popular Member Posts

♦ 11 Spring boot interview questions & answers

850 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

769 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

399 views

18 Java scenarios based interview Questions and Answers

387 views

♦ 7 Java debugging interview questions & answers

308 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

305 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

297 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

294 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

246 views

001B: ♦ Java architecture & design concepts interview questions & answers

204 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription

based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ 06: Spark Streaming with Flume Avro Sink Tutorial

08: Q49 – Q58 Scala Implicits Interview Q&As ›

Posted in member-paid, Scala Interview Q&As

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

© Disclaimer

The contents in this Java-Success are copy righted. The author has the right to correct or enhance the current content without any prior notice.

These are general advice only, and one needs to take his/her own circumstances into consideration. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. No guarantees are made regarding the accuracy or usefulness of content, though I do make an effort to be accurate. Links to external sites do not imply endorsement of the linked-to sites.

↑