JVM concurrency: Asynchronous event handling in Scala

Learn blocking and nonblocking techniques, including easy nonblocking code with the async macro

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

02 September 2014

Scala `Future` and `Promise` classes give you powerful ways to handle asynchronous processing, including nonblocking ways to organize and sequence events. The `async` / `await` constructs, implemented as a macro, make it even easier to write nonblocking event handling in Scala. This tutorial shows you the basic `Future` and `Promise` techniques in action, then demonstrates how `async` / `await` can convert what looks like simple blocking code to nonblocking execution.

View more content in this series

Asynchronous event handling is crucial in concurrent applications. Whatever the events' source — separate computational tasks, I/O operations, or interactions with external systems — your code must keep track of the events and coordinate actions taken in response to them. Applications can take one of two basic approaches to asynchronous event handling:

- **Blocking**: A coordination thread waits for the event.
- **Nonblocking**: The event generates some form of notification to the application without a thread explicitly waiting for it.

In "*JVM concurrency*: To block, or not to block?" you read about ways to handle asynchronous events in Java™ 8, using both blocking and nonblocking techniques based on the `CompletableFuture` class. This tutorial shows some of the options for asynchronous event handling in Scala, starting with a simple blocking version and then moving on to some of the nonblocking options. Finally, you'll see how the `async` / `await` constructs can convert what looks like simple blocking code to nonblocking execution. (Get the full sample code from the author's GitHub repository.)

© Copyright IBM Corporation 2014

About this series

Now that multicore systems are ubiquitous, concurrent programming must be applied more widely than ever before. But concurrency can be difficult to implement correctly, and you need new tools to help you use it. Many of the JVM-based languages are developing tools of this type, and Scala has been particularly active in this area. This series gives you a look at some of the newer approaches to concurrent programming for the Java and Scala languages.

Composing events

The `scala.concurrent.Promise` and `scala.concurrent.Future` classes give Scala developers a similar range of options to what Java 8 developers have with `CompletableFuture`. In particular, `Future` offers both blocking and nonblocking ways of working with event completions. Despite the similarity at this level, however, the techniques used to work with the two types of futures differ.

In this section, you'll see examples of both blocking and nonblocking approaches to working with events represented by `Future`s. This tutorial uses the same concurrent tasks setup as the last one. I'll quickly review that setup before digging into the code.

Tasks and sequencing

Applications often must perform multiple processing steps in the course of a particular operation. For example, before returning a response to the user, a web application might need to:

1. Look up information for the user in a database.
2. Use the looked-up information for a web service call and perhaps another database query.
3. Perform a database update based on the results from the first two operations.

Figure 1 illustrates this type of structure.

Figure 1. Application task flow

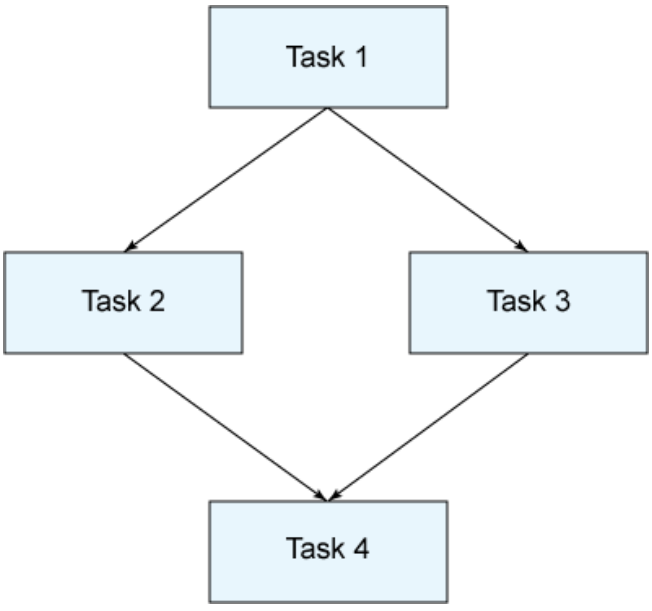

Figure 1 breaks the processing down into four separate tasks, connected by arrows representing order dependencies. Task 1 can execute directly, Task 2 and Task 3 both execute after Task 1 completes, and Task 4 executes after both Task 2 and Task 3 complete.

Modeling asynchronous events

In a real system, the sources of asynchronous events are generally either parallel computations or a form of I/O operation. It's easier to model a system of this type using simple time delays, though, and that's the approach I'm taking here. Listing 1 shows the basic timed-event code used to generate events, in the form of completed `Future`s.

Listing 1. Timed-event code

```
import java.util.Timer
import java.util.TimerTask

import scala.concurrent._

object TimedEvent {
  val timer = new Timer

  /** Return a Future which completes successfully with the supplied value after secs seconds. */
  def delayedSuccess[T](secs: Int, value: T): Future[T] = {
    val result = Promise[T]
    timer.schedule(new TimerTask() {
      def run() = {
        result.success(value)
      }
    }, secs * 1000)
    result.future
  }

  /** Return a Future which completes failing with an IllegalArgumentException after secs
    * seconds. */
  def delayedFailure(secs: Int, msg: String): Future[Int] = {
    val result = Promise[Int]
    timer.schedule(new TimerTask() {
      def run() = {
        result.failure(new IllegalArgumentException(msg))
      }
    }, secs * 1000)
    result.future
  }
```

Like the Java code from the last installment, the Listing 1 Scala code uses a `java.util.Timer` to schedule `java.util.TimerTask`s for execution after a delay. Each `TimerTask` completes an associated future when it runs. The `delayedSuccess` function schedules a task to successfully complete a Scala `Future[T]` when it runs, and it returns the future to the caller. The `delayedSuccess` function returns the same type of future but uses a task that completes the future with an `IllegalArgumentException` failure.

Listing 2 shows how to use the Listing 1 code to create events, in the form of `Future[Int]`s, matching the four tasks in Figure 1. (This code is from the `AsyncHappy` class in the sample code.)

Listing 2. Events for sample tasks

```
// task definitions
def task1(input: Int) = TimedEvent.delayedSuccess(1, input + 1)
def task2(input: Int) = TimedEvent.delayedSuccess(2, input + 2)
def task3(input: Int) = TimedEvent.delayedSuccess(3, input + 3)
def task4(input: Int) = TimedEvent.delayedSuccess(1, input + 4)
```

Each of the four task methods in Listing 2 uses particular delay values for when the task will complete: 1 second for `task1`, 2 seconds for `task2`, 3 seconds for `task3`, and back down to 1 second for `task4`. Each also takes an input value and uses that input plus the task number as the (eventual) result value for the future. These methods all use the success form of the future; later, you'll see examples that use the failure form.

The intention with these tasks is that you run them in the order shown in Figure 1 and pass each task the result value returned by the preceding task (or the sum of the two preceding task results, in the case of `task4`). The total execution time should be approximately 5 seconds (1 second + max(2 seconds, 3 seconds) + 1 second) if the two middle tasks execute concurrently. If 1 is the input to `task1`, the result is 2. If that result is passed to `task2` and `task3`, the results are 4 and 5. And if the sum of these two results (9) is passed as the input to `task4`, the final result is 13.

Blocking waits

Now that the stage is set, it's time to see how Scala handles event completions. As in the Java code from the last tutorial, the simplest way to coordinate the execution of the four tasks is to use blocking waits: The main thread waits for each task to complete in turn. Listing 3 (again, from the `AsyncHappy` class in the sample code) shows this approach.

Listing 3. Blocking waits for tasks

```
def runBlocking() = {
  val v1 = Await.result(task1(1), Duration.Inf)
  val future2 = task2(v1)
  val future3 = task3(v1)
  val v2 = Await.result(future2, Duration.Inf)
  val v3 = Await.result(future3, Duration.Inf)
  val v4 = Await.result(task4(v2 + v3), Duration.Inf)
  val result = Promise[Int]
  result.success(v4)
  result.future
}
```

Listing 3 uses the Scala `scala.concurrent.Await` object `result()` method to do the blocking waits. The code first waits for the `task1` result, then creates both the `task2` and `task3` futures before waiting for each in turn, and finally waits for the `task4` result. The last three lines (creating and setting `result`) enable the method to return a `Future[Int]`. Returning the future makes this method consistent with the nonblocking forms I'll show next, but the future will actually be complete before the method returns.

Combining futures

Listing 4 (again from the `AsyncHappy` class in the sample code) shows one way of linking futures together to execute the tasks in the correct order and with the correct dependencies, all without blocking.

Listing 4. Handling completions with `onSuccess()`

```
def runOnSuccess() = {
  val result = Promise[Int]
  task1(1).onSuccess(v => v match {
    case v1 => {
      val a = task2(v1)
      val b = task3(v1)
      a.onSuccess(v => v match {
        case v2 =>
          b.onSuccess(v => v match {
            case v3 => task4(v2 + v3).onSuccess(v4 => v4 match {
              case x => result.success(x)
            })
          })
      })
    }
  })
  result.future
}
```

The Listing 4 code uses the `onSuccess()` method to set a function (technically a partial function, because it handles only the successful completion case) to be executed when each future completes. Because the `onSuccess()` calls are nested, they'll execute in order (even if the futures are not all completed in order).

The Listing 4 code is reasonably easy to understand but verbose. Listing 5 shows a simpler way of handling this case, using the `flatMap()` method.

Listing 5. Handling completions with `flatMap()`

```
def runFlatMap() = {
  task1(1) flatMap {v1 =>
    val a = task2(v1)
    val b = task3(v1)
    a flatMap { v2 =>
      b flatMap { v3 => task4(v2 + v3) }}
  }
}
```

The Listing 5 code is effectively doing the same thing as the Listing 4 code, but Listing 5 uses the `flatMap()` method to extract the single result value from each future. Using `flatMap()` eliminates the `match` / `case` constructs needed in Listing 4, giving a more concise form but keeping the same step-by-step execution path.

Trying out the example

The sample code uses a Scala `App` to run each version of the events code in turn and ensure that the time to completion (about 5 seconds) and the result (13) are correct. You can run this code from the command line by using Maven, as shown in Listing 6 (with extraneous Maven output removed):

Listing 6. Running the events code

```
dennis@linux-9qea:~/devworks/scala4/code> mvn scala:run -Dlauncher=happypath
...
[INFO] launcher 'happypath' selected => com.sosnoski.concur.article4.AsyncHappy
Starting runBlocking
runBlocking returned 13 in 5029 ms.
Starting runOnSuccess
runOnSuccess returned 13 in 5011 ms.
Starting runFlatMap
runFlatMap returned 13 in 5002 ms.
#
```

The unhappy path

So far, you've seen code to coordinate events in the form of futures that always complete successfully. In real applications, you can't depend on always staying on this happy path. Problems with processing tasks will occur, and in JVM language terms, these problems are normally represented by `Throwable`s.

It's easy to change the Listing 2 task definitions to use `delayedFailure()` in place of the `delayedSuccess()` method, as shown here for `task4`:

```
def task4(input: Int) = TimedEvent.delayedFailure(1, "This won't work!")
```

If you just run the Listing 3 code with `task4` modified to complete with an exception, you get the expected `IllegalArgumentException` thrown by the `Await.result()` call on `task4`. If the problem isn't caught in the `runBlocking()` method, the exception is passed up the call chain until it's finally caught (terminating the thread if it's not caught). Fortunately, it's easy to modify the code so that if any of the tasks completes exceptionally, the exception is passed on to the caller for handling through the returned future. Listing 7 shows this change.

Listing 7. Blocking waits with exceptions

```
def runBlocking() = {
  val result = Promise[Int]
  try {
    val v1 = Await.result(task1(1), Duration.Inf)
    val future2 = task2(v1)
    val future3 = task3(v1)
    val v2 = Await.result(future2, Duration.Inf)
    val v3 = Await.result(future3, Duration.Inf)
    val v4 = Await.result(task4(v2 + v3), Duration.Inf)
    result.success(v4)
  } catch {
    case t: Throwable => result.failure(t)
  }
  result.future
}
```

In Listing 7, the original code is wrapped in a `try`/`catch`, and the `catch` passes the exception back as the completion of the returned future. This approach adds a little complexity but should still be easy for any Scala developer to understand.

What about the nonblocking variations of the event handling code, in Listings 4 and 5? As its name implies, the `onSuccess()` method used in Listing 4 *only* works with successful completions of a

future. If you want to handle both successful and failure completions, you must instead use the `onComplete()` method and check to see which type of completion applies. Listing 8 shows how this technique would work for the event handling code.

Listing 8. `onComplete()` handling of both successes and failures

```
def runOnComplete() = {
  val result = Promise[Int]
  task1(1).onComplete(v => v match {
    case Success(v1) => {
      val a = task2(v1)
      val b = task3(v1)
      a.onComplete(v => v match {
        case Success(v2) =>
          b.onComplete(v => v match {
            case Success(v3) => task4(v2 + v3).onComplete(v4 => v4 match {
              case Success(x) => result.success(x)
              case Failure(t) => result.failure(t)
            })
            case Failure(t) => result.failure(t)
          })
        case Failure(t) => result.failure(t)
      })
    }
    case Failure(t) => result.failure(t)
  })
  result.future
}
```

Listing 8 looks messy, and fortunately you have a much easier alternative: Use the Listing 5`flatMap()` code instead. The `flatMap()` approach handles both success and failure completions without the need for any changes.

Using `async`

Recent versions of Scala include the ability to transform code during compilation by using *macros*. One of the most useful macros implemented so far is `async`, which transforms apparently sequential code that uses futures into asynchronous code during compilation. Listing 9 shows how `async` can simplify the task code used in this tutorial.

Listing 9. Combining futures with `async {}`

```
def runAsync(): Future[Int] = {
  async {
    val v1 = await(task1(1))
    val a = task2(v1)
    val b = task3(v1)
    await(task4(await(a) + await(b)))
  }
}
```

The enclosing `async {...}` block in Listing 9 invokes the `async` macro. This invocation declares the block to be asynchronous and, by default, executed asynchronously, and it returns a future for the result of the block. Within the block, the `await()` method (actually a keyword for the macro, rather than a true method) shows where the result of a future is needed. The async macro modifies the abstract syntax tree (AST) of the Scala program during compilation to convert the block into code that uses callbacks, roughly equivalent to the Listing 4 code.

Aside from the `async {...}` wrapper, the Listing 9 code looks much like the original blocking code shown in Listing 3. That's quite an accomplishment for the macro — abstracting away all the complexity of asynchronous events and making it look like you're writing simple linear code. Behind the scenes, a considerable amount of complexity *is* involved.

`async` uncovered

If you look at the classes generated from the source code by the Scala compiler, you'll see several inner classes with names like `AsyncHappy$$anonfun$1.class`. As you might guess from the name, these are generated by the compiler for anonymous functions (such as the statements passed to the `onSuccess()` or `flatMap()` methods).

With the Scala 2.11.1 compiler and Async 0.9.2 implementation, you'll also see a class named `AsyncUnhappy$stateMachine$macro$1$1.class`. This is the actual implementation code generated by the `async` macro, in the form of a state machine to handle the asynchronous tasks. Listing 10 shows a partial decompiled view of this class.

Listing 10. Decompiled `AsyncUnhappy$stateMachine$macro$1$1.class`

```
public class AsyncUnhappy$stateMachine$macro$1$1
  implements Function1<Try<Object>, BoxedUnit>, Function0.mcV.sp
{
  private int state;
  private final Promise<Object> result;
  private int await$macro$3$macro$13;
  private int await$macro$7$macro$14;
  private int await$macro$5$macro$15;
  private int await$macro$11$macro$16;
  ...
  public void resume() {
    ...
  }

  public void apply(Try<Object> tr) {
    int i = this.state;
    switch (i) {
      default:
        throw new MatchError(BoxesRunTime.boxToInteger(i));
      case 3:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$11$macro$16 = BoxesRunTime.unboxToInt(tr.get());
          this.state = 4;
          resume();
        }
        break;
      case 2:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$7$macro$14 = BoxesRunTime.unboxToInt(tr.get());
          this.state = 3;
          resume();
        }
        break;
      case 1:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$5$macro$15 = BoxesRunTime.unboxToInt(tr.get());
```

```
          this.state = 2;
          resume();
        }
        break;
      case 0:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$3$macro$13 = BoxesRunTime.unboxToInt(tr.get());
          this.state = 1;
          resume();
        }
        break;
    }
  }
  ...
}
```

The Listing 10`apply()` method handles the actual state changes, evaluating the result of a future and changing the output state to match. The input state tells the code which future is being evaluated; each state value corresponds to one particular future within the async block. It's hard to tell this from the partial Listing 10 code, but from looking at some of the other bytecode, you can see that the state codes match up with the tasks, so state `0` means the result of `task1` is expected, state `1` means the result of `task2` is expected, and so on.

The `resume()` method is not shown in Listing 10 because the decompiler wasn't able to figure out how to convert it to Java code. I'm not going to go through this exercise either, but from looking at the bytecode I can tell you that the `resume()` method does the equivalent of a Java `switch` on the state code. For each nonterminal state, `resume()` executes the appropriate fragment of code to set up the next expected future, ending by setting the `AsyncUnhappy$stateMachine$macro$1$1` instance as the target of the future's `onComplete()` method. For the terminal state, `resume()` sets the result value and completes the promise for the final result.

You don't actually need to dig through the generated code to understand async (though it can be interesting). The full description of how async works is included in the SIP-22 - Async proposal.

`async` limitations

Use of the `async` macro has some limitations because of the way the macro converts code into a state-machine class. The most significant restriction is that you can't nest `await()` inside another object or closure (including a function definition) within the `async` block. You also can't nest the `await()` inside a `try` or `catch`.

These usage limitations aside, the biggest problem with `async` is that when it comes to debugging, you're right back to the callback-hell experience often associated with asynchronous code — in this case, trying to make sense of call stacks that don't reflect your apparent code structure. Unfortunately, there's no way around these problems with current debugger designs. This is an area seeing new work in Scala (see Rethinking the debugger.) In the meantime, you can disable asynchronous execution of `async` blocks to make debugging easier (assuming the problem you're trying to fix still occurs when you execute sequentially).

Finally, Scala macros are still a work in progress. The intention is that `async` will become an official part of the Scala language in an upcoming release, but that will only happen when the Scala language team is satisfied with the way macros work. Until then, there are no guarantees that the form of `async` won't change.

Conclusion

Some Scala approaches to handling asynchronous events diverge significantly from the Java code discussed in "*JVM concurrency*: To block, or not to block?" With both the `flatMap()` and the `async` macro, Scala offers techniques that are clean and easy to understand. `async` is especially interesting in that you can write what looks like normal sequential code, but the compiled code executes concurrently. Scala isn't the only language to offer this type of approach, but the macro-based implementation provides superior flexibility to other approaches.

Resources

- Scalable Scala: Series author Dennis Sosnoski shares insights and behind-the-scenes information on the content in this series and Scala development in general.
- Sample code for this tutorial: Get this tutorial's full sample code from the author's repository on GitHub.
- Scala: Scala is a modern, functional language on the JVM.
- "SIP-22 - Async" (Philipp Haller and Jason Zaugg, Scala Improvement Process): This Scala Improvement Process (SIP) proposal describes the intent behind the `async` and `await` constructs and gives details of the code transformation used to build the state-machine class.
- "An asynchronous programming facility for Scala" (Jason Zaugg et al, Github): Get the source code and latest documentation for the `async` implementation.
- "Rethinking the debugger" (Iulian Dragos, Scala Days 2014): Learn about some of the exciting ideas being explored in Scala to making debugging concurrent programs easier.

About the author

Dennis Sosnoski

Dennis Sosnoski is a Java and Scala developer with extensive experience developing scalable systems. He's well-known in the XML and web services areas, where his background includes the development of JiBX XML data binding and work on several open source web services frameworks (most recently, Apache CXF). Dennis is a frequent presenter at Java user groups and conferences and has written many articles for developerWorks, including the popular *Java web services* series. Learn more about his web services training and consulting work at his Sosnoski Software Associates Ltd site, and follow his ongoing explorations of concurrent programming on the JVM at his Scalable Scala site.

© Copyright IBM Corporation 2014
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)