Assignment 3 FAQ

Help

Hello all,

This is the FAQ document for week 3 - we're listing answers to common questions here. Make sure you read this before posting a question - maybe the answer's already here!

2. What is a `Promise`? What is a `Future`? What does the `Future` type have to do with doing an asynchronous computation like this `Future { ... }`? And how is this different from `async`?

First of all, there is a difference between the **type** `Future[T]` and the **computation** started with the `Future` block.

The type `Future[T]` designates a value of type `T` that might not be available now, but will become available at some point in the future. This type captures the concept of time in your programs - the future is initially **not completed**, and then becomes **completed** at some time. And because the value of the future is not always available, you cannot always get the value immediately by calling some method on instances of the type `Future[T]`. Instead, you must install callbacks to this type using `onComplete`, or, more conveniently, you should use `map`, `flatMap` or `filter` to transform one future in some other future - see lectures for more details on that.

Question is - how can I even obtain values of this type `Future[T]`? There are several ways. First of all, you can start of an asynchronous computation that will potentially execute on a different thread and that will complete the future. This is done with the `Future { ... }` block - but it is not to be confused with the `Future[T]` type! The `Future[T]` might not really involve no asynchronous computation, it just designates the possibility of it.

Next question is - if `Future[T]` can be completed differently than by starting an asynchronous computation, then how? This is

2. Can we use `Thread.sleep?`

Yes you can when defining the `delay`, but it should not be your general practice. It's only necessary for some future combinators, but very few of them. In fact, you can use any construct from `scala.concurrent` that results in delaying for that amount of time.

Just make sure that when you read the note about `blocking`!

3. Should I use `async` or future combinators and promises?

Depends on that task, but use whatever you like. Sometimes one solves the task better, sometimes the other. You should strive for concise and understandable, but correct code.

We suggest that you solve the server part using `async` as the code should be more readable and clear, but if you really want to call future combinators, that is fine too.

But avoid blocking as it's a bad practice here - `Await.result` you may only use for `delay` or with an argument `0 seconds`.

4. Why does `async` not work in a value class - what are all these errors with nested classes not allowed in value classes?

The Scala Async feature is nearing its first release and is partly experimental.

As a workaround for using Scala Async in value classes, consider moving the `async` invocation outside of the value class using the following trick:

```
class FutureOps[T](f: Future[T]) extends AnyVal {
  def foo: Future[T] = fooImpl(f)
}

def fooImpl(f: Future[T]) = async {
  val x = await { f }
  throw new Exception
}
```

If that or other code juggling and refactoring does not solve the problem, consider using regular functional combinators like `map` , `flatMap` , `continueWith` and the rest of their friends.

5. What's a good reference for futures and promises?

This is: http://docs.scala-lang.org/overviews/core/futures.html Also the API docs can help: http://www.scala-lang.org/api/current/index.html\#scala.concurrent.Future

And don't forget the lectures! :)

6. Semantics of `continueWith` and friends - how should it handle exceptions?

All `Future` combinators should handle exceptions in a reasonable way. This means that if the user gives them arbitrary blocks of code to execute, they should be aware that those blocks of code also potentially throw. When in doubt, consult the implementation of `flatMap` in the Scala standard library:

https://github.com/scala/scala/blob/master/src/library/scala/concurrent/Future.scala#L239

7. What should be semantics of `now`

It should throw an exception if the future is not completed, and return the value if the future is completed. And if the future is completed with an exception - well, the only thing to do is to throw the exception back at the user.

8. What should be semantics of `always`

It should return a future that is always completed with the specified value, i.e. as soon as the future is constructed and returned it should already be completed.

9. Why can't I mix `Future`s with `List`s in my `for`-comprehensions?

Because the `flatMap` on the `List` is defined to accept only functions that return other `List`s, roughly speaking, and `flatMap` on the `Future` is defined to only accept functions that return futures.

The reason that is done this way is because `List` would have to know the semantics of futures and vice versa for this to be implemented correctly. These two abstractions model very different things -- a list has no idea that a future is asynchronous.

But, to solve this issue, you can `always` convert a list to a future of that list. Get it? ;)

10. What is cancellation useful for? Why is `CancellationTokenSource` recommended to use a `Promise`?

`CancellationToken`s and `CancellationTokenSource`s are used to communicate in the opposite direction than the one you do when you use `Future`s. Clients ask for `Future`s and wait until they get some result back from the computation, but they cannot write to the `Future`. On the other hand, clients can try to unsubscribe from the computation using `CancellationTokenSource`s and in this way write back to the computation. The computation itself cannot write to the `CancellationToken`, only check if they are cancelled.

On the JVM you're strongly advised to use proper synchronization when sharing mutable state between the threads, to learn more about this you can try to find resources on the Java Memory Model.

You could either use the `@volatile` annotation or the `synchronized` keyword to ensure that the inner state of the `CancellationTokenSource` is properly accessed, but the simplest thing is to reuse the logic you already have in `Promise`. Promises are single-assignment variables that are already thread-safe.

11. What does the `blocking` call do? When do I have to use it?

The explanation is fairly technical. It has to do with how thread pools work on the JVM. When a typical thread pool is created on the JVM, it contains some fixed number of threads. These threads are used to work on client-submitted *tasks*, for example, computations that have to execute asynchronously when you call the `Future` block - for this reason, these threads are called the *worker threads.* Each worker thread has a queue of tasks and continually polls this queue to see if there are new tasks to execute. In some cases, worker threads may even share this queue(s) by taking each other's tasks. This all works out nicely until it happens that all the worker threads execute tasks with code which blocks on some condition. At that point, even if new tasks arrive, no worker thread will be able to execute them, because it is blocking. Assume that all the threads are blocking for user input, for example. If you call `userInput` 4 times on a 4 processor machine (there are as many worker threads as processors, by default), all the worker threads will wait for user input. If you start additional futures after that, they will not be executed until the user enters some input.

All this shows that blocking is generally not a good idea with the JVM and concurrent reactive programming. But sometimes you need it! There is a way out of this, however - typically, thread pools can add new worker threads to execute the tasks if they can detect that all the threads are blocked. Various thread pools do this in different ways, but in general they need a hint. This is where the `blocking` construct comes in. Any time a future executes code inside `blocking`, it will notify the thread pool to watch out - potentially new threads should be created to prevent thread starvation.

You have to watch out when implementing `delay` and `userInput` for this reason.

See more information here: http://stackoverflow.com/questions/19681389/use-case-of-scala-concurrent-blocking

---

Created Tue 19 Nov 2013 4:55 AM PST

Last Modified Thu 21 Nov 2013 12:51 AM PST