Chapter 32

Futures and Concurrency

One consequence of the proliferation of multicore processors has been an increased interest in concurrency. Java provides concurrency support built around shared memory and locking. Although this support is sufficient, this approach turns out to be quite difficult to get right in practice. Scala's standard library offers an alternative that avoids these difficulties by focusing on asynchronous transformations of immutable state: the Future.

Although Java also offers a Future, it is very different from Scala's. Both represent the result of an asynchronous computation, but Java's Future requires that you access the result via a blocking get method. Although you can call isDone to find out if a Java Future has completed before calling get, thereby avoiding any blocking, you must wait until the Java Future has completed before proceeding with any computation that uses the result.

By contrast, you can specify transformations on a Scala Future whether it has completed or not. Each transformation results in a new Future representing the asynchronous result of the original Future transformed by the function. The thread that performs the computation is determined by an implicitly provided execution context. This allows you to describe asynchronous computations as a series of transformations of immutable values, with no need to reason about shared memory and locks.

32.1 TROUBLE IN PARADISE

On the Java platform, each object is associated with a logical *monitor*, which can be used to control multi-threaded access to data. To use this model, you decide what data will be shared by multiple threads and mark as "synchronized" sections of the code that access, or control access to, the shared data. The Java runtime employs a locking mechanism to ensure that only one thread at a time enters synchronized sections guarded by the same lock, thereby enabling you to orchestrate multi-threaded access to the shared data.

For compatibility's sake, Scala provides access to Java's concurrency primitives. The wait,notify, and notifyAll methods can be called in Scala, and they have the same meaning as in Java. Scala doesn't technically have a synchronized keyword, but it includes a predefinedsynchronized method that can be called as follows:

```
var counter = 0
synchronized {
  // One thread in here at a time
  counter = counter + 1
}
```

Unfortunately, programmers have found it very difficult to reliably build robust multi-threaded applications using the shared data and locks model, especially as applications grow in size and complexity. The problem is that at each point in the program, you must reason about what data you are

modifying or accessing that might be modified or accessed by other threads, and what locks are being held. At each method call, you must reason about what locks it will try to hold and convince yourself that it will not deadlock while trying to obtain them. Compounding the problem, the locks you reason about are not fixed at compile time, because the program is free to create new locks at run time as it progresses.

Making things worse, testing is not reliable with multi-threaded code. Since threads are non-deterministic, you might successfully test a program one thousand times—and the program could still go wrong the first time it runs on a customer's machine. With shared data and locks, you must get the program correct through reason alone.

Moreover, you can't solve the problem by over-synchronizing either. It can be just as problematic to synchronize everything as it is to synchronize nothing. Although new lock operations may remove possibilities for race conditions, they simultaneously add possibilities for deadlocks. A correct lock-using program must have neither race conditions nor deadlocks, so you cannot play it safe by overdoing it in either direction.

The java.util.concurrent library provides higher level abstractions for concurrent programming. Using the concurrency utilities makes multi-threaded programming far less error prone than rolling your own abstractions with Java's low-level synchronization primitives. Nevertheless, the concurrent utilities are also based on the shared data and locks model, and as a result, do not solve the fundamental difficulties of using that model.

32.2 ASYNCHRONOUS EXECUTION AND TRYS

Although not a silver bullet, Scala's Future offers one way to deal with concurrency that can reduce, and often eliminate, the need to reason about shared data and locks. When you invoke a Scala method, it performs a computation "while you wait" and returns a result. If that result is a Future, the Future represents another computation to be performed asynchronously, often by a completely different thread. As a result, many operations on Futurerequire an implicit execution context that provides a strategy for executing functions asynchronously. For example, if you try to create a future via the Future.apply factory methodwithout providing an implicit execution context, an instance ofscala.concurrent.ExecutionContext, you'll get a compiler error:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
<console>:11: error: Cannot find an implicit ExecutionContext.
    You might pass an (implicit ec: ExecutionContext)
    parameter to your method or import
    scala.concurrent.ExecutionContext.Implicits.global.
       val fut = Future { Thread.sleep(10000); 21 + 21 }
                          ^
```

The error message gives you one way to solve the problem: importing a global execution context provided by Scala itself. On the JVM, the global execution context uses a thread pool.[1] Once you bring an implicit execution context into scope, you can create a future:

```
scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...
```

The future created in the previous example asynchronously executes the block of code, using the global execution context, then completes with the value 42. Once it starts execution, that thread will sleep for ten seconds. Thus this future will take at least ten seconds to complete.

Two methods on Future allow you to poll: isCompleted and value. When invoked on a future that has not yet completed, isCompleted will return false and value will return None.

```
scala> fut.isCompleted
res0: Boolean = false

scala> fut.value
res1: Option[scala.util.Try[Int]] = None
```

Once the future completes (in this case, after at least ten seconds has gone by), isCompleted will return true and value will return a Some:

```
scala> fut.isCompleted
res2: Boolean = true

scala> fut.value
res3: Option[scala.util.Try[Int]] = Some(Success(42))
```

The option returned by value contains a Try. As shown in Figure 32.1, a Try is either a Success, which contains a value of type T, or a Failure, which contains an exception (an instance ofjava.lang.Throwable). The purpose of Try is to provide for asynchronous computations what thetry expression provides for synchronous computations: It allows you to deal with the possibility that the computation will complete abruptly with an exception rather than return a result.[2]

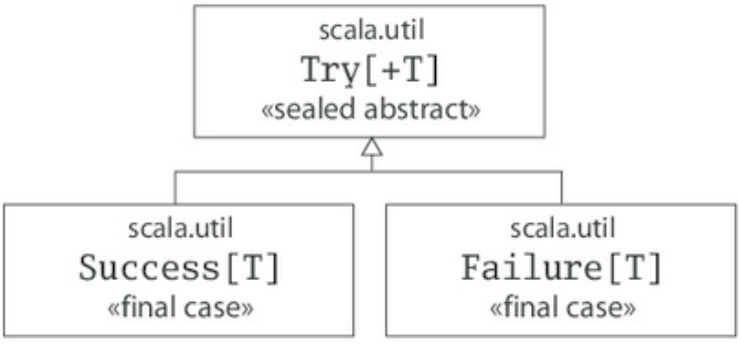

Figure 32.1 - Class hierarchy for Try.

For synchronous computations you can use try/catch to ensure that a thread that invokes a method catches and handles exceptions thrown by the method. For asynchronous computations, however, the thread that initiates the computation often moves on to other tasks. Later if that asynchronous computation fails with an exception, the original thread is no longer able to handle the exception in a catch clause. Thus when working with a Futurerepresenting an asynchronous activity, you use Try to deal with the possibility that the activity fails to yield a value and instead completes abruptly an exception. Here's an example that shows what happens when an asynchronous activity fails:

```
scala> val fut = Future { Thread.sleep(10000); 21 / 0 }
fut: scala.concurrent.Future[Int] = ...

scala> fut.value
res4: Option[scala.util.Try[Int]] = None
```

Then, after ten seconds:

```
scala> fut.value
res5: Option[scala.util.Try[Int]] =
    Some(Failure(java.lang.ArithmeticException: / by zero))
```

32.3 WORKING WITH FUTURES

Scala's Future allows you to specify transformations on Future results and obtain a new futurethat represents the composition of the two asynchronous computations: the original and the transformation.

Transforming Futures with map

The most fundamental such operation is map. Instead of blocking then continuing with another computation, you can just map the next computation onto the future. The result will be a new future that represents the original asynchronously computed result transformed asynchronously by the function passed to map.

For example, the following future will complete after ten seconds:

```
scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...
```

Mapping this future with a function that increments by one will yield another future. This new future will represent a computation consisting of the original addition followed by the subsequent increment:

```
scala> val result = fut.map(x => x + 1)
result: scala.concurrent.Future[Int] = ...

scala> result.value
res5: Option[scala.util.Try[Int]] = None
```

Once the original future completes and the function has been applied to its result, the future returned by map will complete:

```
scala> result.value
```

```
res6: Option[scala.util.Try[Int]] = Some(Success(43))
```

Note that the operations performed in this example—the future creation, the 21 + 21 sum calculation, and the 42 + 1 increment—may be performed by three different threads.

Transforming Futures with for expressions

Because Scala's future also declares a flatMap method, you can transform futures using a forexpression. For example, consider the following two futures that will, after ten seconds, produce 42 and 46:

```
scala> val fut1 = Future { Thread.sleep(10000); 21 + 21 }
fut1: scala.concurrent.Future[Int] = ...

scala> val fut2 = Future { Thread.sleep(10000); 23 + 23 }
fut2: scala.concurrent.Future[Int] = ...
```

Given these two futures, you can obtain a new future representing the asynchronous sum of their results like this:

```
scala> for {
         x <- fut1
         y <- fut2
       } yield x + y
res7: scala.concurrent.Future[Int] = ...
```

Once the original futures have completed, and the subsequent sum completes, you'll be able to see the result:

```
scala> res7.value
res8: Option[scala.util.Try[Int]] = Some(Success(88))
```

Because for expressions serialize their transformations,[3] if you don't create the futures before the for expression, they won't run in parallel. For example, although the previous forexpression requires around ten seconds to complete, the following for expression requires at least twenty seconds:

```
scala> for {
         x <- Future { Thread.sleep(10000); 21 + 21 }
         y <- Future { Thread.sleep(10000); 23 + 23 }
       } yield x + y
res9: scala.concurrent.Future[Int] = ...

scala> res9.value
res27: Option[scala.util.Try[Int]] = None

scala> // Will need at least 20 seconds to complete

scala> res9.value
res28: Option[scala.util.Try[Int]] = Some(Success(88))
```

Creating the Future: Future.failed, Future.successful, Future.fromTry, and Promises

Besides the apply method, used in earlier examples to create futures, the Future companion object also includes three factory methods for creating already-completed futures: successful,failed, and fromTry. These factory methods do not require an ExecutionContext.

The successful factory method creates a future that has already succeeded:

```
scala> Future.successful { 21 + 21 }
res2: scala.concurrent.Future[Int] = ...
```

The failed method creates a future that has already failed:

```
scala> Future.failed(new Exception("bummer!"))
res3: scala.concurrent.Future[Nothing] = ...
```

The fromTry method creates an already completed future from a Try:

```
scala> import scala.util.{Success,Failure}
import scala.util.{Success, Failure}

scala> Future.fromTry(Success { 21 + 21 })
res4: scala.concurrent.Future[Int] = ...

scala> Future.fromTry(Failure(new Exception("bummer!")))
res5: scala.concurrent.Future[Nothing] = ...
```

The most general way to create a future is to use a Promise. Given a promise you can obtain a future that is controlled by the promise. The future will complete when you complete the promise. Here's an example:

```
scala> val pro = Promise[Int]
pro: scala.concurrent.Promise[Int] = ...

scala> val fut = pro.future
fut: scala.concurrent.Future[Int] = ...

scala> fut.value
res8: Option[scala.util.Try[Int]] = None
```

You can complete the promise with methods named success, failure, and complete. These methods on Promise are similar to those described previously for constructing already completed futures. For example, the success method will complete the future successfully:

```
scala> pro.success(42)
res9: pro.type = ...

scala> fut.value
res10: Option[scala.util.Try[Int]] = Some(Success(42))
```

The failure method takes an exception that will cause the future to fail with that exception.
The complete method takes a Try. A completeWith method, which takes a future, also exists; the promise's future will thereafter mirror the completion status of the future you passed tocompleteWith.

Filtering: filter and collect

Scala's future offers two methods, filter and collect, that allow you to ensure a property holds true about a future value. The filter method validates the future result, leaving it the same if it is valid. Here's an example that ensures an Int is positive:

```
scala> val fut = Future { 42 }
fut: scala.concurrent.Future[Int] = ...

scala> val valid = fut.filter(res => res > 0)
valid: scala.concurrent.Future[Int] = ...

scala> valid.value
res0: Option[scala.util.Try[Int]] = Some(Success(42))
```

If the future value is not valid, the future returned by filter will fail with aNoSuchElementException:

```
scala> val invalid = fut.filter(res => res < 0)
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res1: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.filter predicate is not satisfied))
```

Because Future also offers a withFilter method, you can perform the same operation with forexpression filters:

```
scala> val valid = for (res <- fut if res > 0) yield res
valid: scala.concurrent.Future[Int] = ...

scala> valid.value
res2: Option[scala.util.Try[Int]] = Some(Success(42))

scala> val invalid = for (res <- fut if res < 0) yield res
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res3: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.filter predicate is not satisfied))
```

Future's collect method allows you to validate the future value and transform it in one operation. If the partial function passed to collect is defined at the future result, the future returned by collect will succeed with that value transformed by the function:

```
scala> val valid =
         fut collect { case res if res > 0 => res + 46 }
valid: scala.concurrent.Future[Int] = ...

scala> valid.value
res17: Option[scala.util.Try[Int]] = Some(Success(88))
```

Otherwise, the future will fail with NoSuchElementException:

```
scala> val invalid =
```

```
          fut collect { case res if res < 0 => res + 46 }
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res18: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.collect partial function is not defined at: 42))
```

Dealing with failure: failed, fallBackTo, recover, and recoverWith

Scala's future provides ways to work with futures that fail, including failed, fallBackTo, recover, and recoverWith. The failed method will transform a failed future of any type into a successfulFuture[Throwable] that holds onto the exception that caused the failure. Here's an example:

```
scala> val failure = Future { 42 / 0 }
failure: scala.concurrent.Future[Int] = ...

scala> failure.value
res23: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))

scala> val expectedFailure = failure.failed
expectedFailure: scala.concurrent.Future[Throwable] = ...

scala> expectedFailure.value
res25: Option[scala.util.Try[Throwable]] =
  Some(Success(java.lang.ArithmeticException: / by zero))
```

If the future on which the failed method is called ultimately succeeds, the future returned byfailed will itself fail with a NoSuchElementException. The failed method is appropriate, therefore, only when you expect that the future will fail. Here's an example:

```
scala> val success = Future { 42 / 1 }
success: scala.concurrent.Future[Int] = ...

scala> success.value
res21: Option[scala.util.Try[Int]] = Some(Success(42))

scala> val unexpectedSuccess = success.failed
unexpectedSuccess: scala.concurrent.Future[Throwable] = ...

scala> unexpectedSuccess.value
res26: Option[scala.util.Try[Throwable]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.failed not completed with a throwable.))
```

The fallbackTo method allows you to provide an alternate future to use in case the future on which you invoke fallbackTo fails. Here's an example in which a failed future falls back to a successful future:

```
scala> val fallback = failure.fallbackTo(success)
fallback: scala.concurrent.Future[Int] = ...

scala> fallback.value
res27: Option[scala.util.Try[Int]] = Some(Success(42))
```

If the original future on which fallbackTo is invoked fails, a failure of the future passed tofallbackTo is essentially ignored. The future returned by fallbackTo will fail with the initial exception. Here's an example:

```
scala> val failedFallback = failure.fallbackTo(
         Future { val res = 42; require(res < 0); res }
        )
failedFallback: scala.concurrent.Future[Int] = ...

scala> failedFallback.value
res28: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))
```

The recover method allows you to transform a failed future into a successful one, allowing a successful future's result to pass through unchanged. For example, on a future that fails withArithmeticException, you can use the recover method to transform the failure into a success, like this:

```
 scala> val recovered = failedFallback recover {
          case ex: ArithmeticException => -1
        }
 recovered: scala.concurrent.Future[Int] = ...

 scala> recovered.value
res32: Option[scala.util.Try[Int]] = Some(Success(-1))
```

If the original future doesn't fail, the future returned by recover will complete with the same value:

```
scala> val unrecovered = fallback recover {
         case ex: ArithmeticException => -1
       }
unrecovered: scala.concurrent.Future[Int] = ...

scala> unrecovered.value
res33: Option[scala.util.Try[Int]] = Some(Success(42))
```

Similarly, if the partial function passed to recover isn't defined at the exception with which the original future ultimately fails, that original failure will pass through:

```
scala> val alsoUnrecovered = failedFallback recover {
         case ex: IllegalArgumentException => -2
       }
alsoUnrecovered: scala.concurrent.Future[Int] = ...

scala> alsoUnrecovered.value
res34: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))
```

The recoverWith method is similar to recover, except instead of recovering to a value like recover, the recoverWith method allows you to recover to a future value. Here's an example:

```
scala> val alsoRecovered = failedFallback recoverWith {
         case ex: ArithmeticException => Future { 42 + 46 }
       }
alsoRecovered: scala.concurrent.Future[Int] = ...
```

```
scala> alsoRecovered.value
res35: Option[scala.util.Try[Int]] = Some(Success(88))
```

As with recover, if either the original future doesn't fail, or the partial function passed torecoverWith isn't defined at the exception the original future ultimately fails with, the original success (or failure) will pass through to the future returned by recoverWith.

Mapping both possibilities: transform

Future's transform method accepts two functions with which to transform a future: one to use in case of success and the other in case of failure:

```
scala> val first = success.transform(
         res => res * -1,
         ex => new Exception("see cause", ex)
       )
first: scala.concurrent.Future[Int] = ...
```

If the future succeeds, the first function is used:

```
scala> first.value
res42: Option[scala.util.Try[Int]] = Some(Success(-42))
```

If the future fails, the second function is used:

```
scala> val second = failure.transform(
         res => res * -1,
         ex => new Exception("see cause", ex)
       )
second: scala.concurrent.Future[Int] = ...

scala> second.value
res43: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.Exception: see cause))
```

Note that with the transform method shown in the previous examples, you can't change a successful future into a failed one, nor can you change a failed future into a successful one. To make this kind of transformation easier, Scala 2.12 introduced an alternate overloaded form of transform that takes a function from Try to Try. Here are some examples:

```
scala> val firstCase = success.transform { // Scala 2.12
         case Success(res) => Success(res * -1)
         case Failure(ex) =>
           Failure(new Exception("see cause", ex))
       }
first: scala.concurrent.Future[Int] = ...

scala> firstCase.value
res6: Option[scala.util.Try[Int]] = Some(Success(-42))

scala> val secondCase = failure.transform {
         case Success(res) => Success(res * -1)
         case Failure(ex) =>
           Failure(new Exception("see cause", ex))
       }
```

```
secondCase: scala.concurrent.Future[Int] = ...

scala> secondCase.value
res8: Option[scala.util.Try[Int]] =
   Some(Failure(java.lang.Exception: see cause))
```

Here's an example of using the new transform method to transform a failure into a success:

```
scala> val nonNegative = failure.transform { // Scala 2.12
         case Success(res) => Success(res.abs + 1)
         case Failure(_) => Success(0)
       }
nonNegative: scala.concurrent.Future[Int] = ...

scala> nonNegative.value
res11: Option[scala.util.Try[Int]] = Some(Success(0))
```

Combining futures: zip, Future.fold, Future.reduce, Future.sequence, and Future.traverse

Future and its companion object offer methods that allow you to combine multiple futures.
The zip method will transform two successful futures into a future tuple of both values. Here's an example:

```
scala> val zippedSuccess = success zip recovered
zippedSuccess: scala.concurrent.Future[(Int, Int)] = ...

scala> zippedSuccess.value
res46: Option[scala.util.Try[(Int, Int)]] =
     Some(Success((42,-1)))
```

If either of the futures fail, however, the future returned by zip will also fail with the same exception:

```
scala> val zippedFailure = success zip failure
zippedFailure: scala.concurrent.Future[(Int, Int)] = ...

scala> zippedFailure.value
res48: Option[scala.util.Try[(Int, Int)]] =
  Some(Failure(java.lang.ArithmeticException: / by zero))
```

If both futures fail, the failed future that results will contain the exception stored in the initial future, the one on which zip was invoked.

Future's companion object offers a fold method that allows you to accumulate a result across aTraversableOnce collection of futures, yielding a future result. If all futures in the collection succeed, the resulting future will succeed with the accumulated result. If any future in the collection fails, the resulting future will fail. If multiple futures fail, the result will fail with the same exception with which the first future (earliest in the TraversableOnce collection) fails. Here's an example:

```
scala> val fortyTwo = Future { 21 + 21 }
fortyTwo: scala.concurrent.Future[Int] = ...

scala> val fortySix = Future { 23 + 23 }
fortySix: scala.concurrent.Future[Int] = ...
```

```
scala> val futureNums = List(fortyTwo, fortySix)
futureNums: List[scala.concurrent.Future[Int]] = ...

scala> val folded =
        Future.fold(futureNums)(0) { (acc, num) =>
          acc + num
        }
folded: scala.concurrent.Future[Int] = ...

scala> folded.value
res53: Option[scala.util.Try[Int]] = Some(Success(88))
```

The Future.reduce method performs a fold without a zero, using the initial future result as the start value. Here's an example:

```
scala> val reduced =
        Future.reduce(futureNums) { (acc, num) =>
          acc + num
        }
reduced: scala.concurrent.Future[Int] = ...

scala> reduced.value
res54: Option[scala.util.Try[Int]] = Some(Success(88))
```

If you pass an empty collection to reduce, the resulting future will fail with aNoSuchElementException.

The Future.sequence method transforms a TraversableOnce collection of futures into a futureTraversableOnce of values. For instance, in the following example, sequence is used to transform a List[Future[Int]] to a Future[List[Int]]:

```
scala> val futureList = Future.sequence(futureNums)
futureList: scala.concurrent.Future[List[Int]] = ...

scala> futureList.value
res55: Option[scala.util.Try[List[Int]]] =
  Some(Success(List(42, 46)))
```

The Future.traverse method will change a TraversableOnce of any element type into aTraversableOnce of futures and "sequence" that into a future TraversableOnce of values. For example, here a List[Int] is transformed into a Future[List[Int]] by Future.traverse:

```
scala> val traversed =
        Future.traverse(List(1, 2, 3)) { i => Future(i) }
traversed: scala.concurrent.Future[List[Int]] = ...

scala> traversed.value
res58: Option[scala.util.Try[List[Int]]] =
  Some(Success(List(1, 2, 3)))
```

Performing side-effects: foreach, onComplete, and andThen

Sometimes you may need to perform a side effect after a future completes. Future provides several methods for this purpose. The most basic method is foreach, which will perform a side effect if a future

completes successfully. For instance, in the following example a println is not executed in the case of a failed future, just a successful future:

```
scala> failure.foreach(ex => println(ex))

scala> success.foreach(res => println(res))
42
```

Since for without yield will rewrite to an invocation of foreach, you can also accomplish the same effect using for expressions:

```
scala> for (res <- failure) println(res)

scala> for (res <- success) println(res)
42
```

Future also offers two methods for registering "callback" functions. The onComplete method will be executed whether the future ultimately succeeds or fails. The function will be passed a Try—a Success holding the result if the future succeeded, else a Failure holding the exception that caused the future to fail. Here's an example:

```
scala> import scala.util.{Success, Failure}
import scala.util.{Success, Failure}

scala> success onComplete {
        case Success(res) => println(res)
        case Failure(ex) => println(ex)
      }
42

scala> failure onComplete {
        case Success(res) => println(res)
        case Failure(ex) => println(ex)
      }
java.lang.ArithmeticException: / by zero
```

Future does not guarantee any order of execution for callback functions registered withonComplete. If you want to enforce an order for callback functions, you must use andThen instead.The andThen method returns a new future that mirrors (succeeds or fails in the same way as) the original future on which you invoke andThen, but it does not complete until the callback function has been fully executed:

```
scala> val newFuture = success andThen {
        case Success(res) => println(res)
        case Failure(ex) => println(ex)
      }
42
newFuture: scala.concurrent.Future[Int] = ...

scala> newFuture.value
res76: Option[scala.util.Try[Int]] = Some(Success(42))
```

Note that if a callback function passed to andThen throws an exception when executed, that exception will not be propagated to subsequent callbacks or reported via the resulting future.

Other methods added in 2.12: flatten, zipWith, and transformWith

The flatten method, added in 2.12, transforms a Future nested inside another Future into aFuture of the nested type. For example, flatten can transform a Future[Future[Int]] into aFuture[Int]:

```
scala> val nestedFuture = Future { Future { 42 } }
nestedFuture: Future[Future[Int]] = ...

scala> val flattened = nestedFuture.flatten // Scala 2.12
flattened: scala.concurrent.Future[Int] = Future(Success(42))
```

The zipWith method, added in 2.12, essentially zips two Futures together, then performs a mapon the resulting tuple. Here's an example of the two-step process, a zip followed by a map:

```
scala> val futNum = Future { 21 + 21 }
futNum: scala.concurrent.Future[Int] = ...

scala> val futStr = Future { "ans" + "wer" }
futStr: scala.concurrent.Future[String] = ...

scala> val zipped = futNum zip futStr
zipped: scala.concurrent.Future[(Int, String)] = ...

scala> val mapped = zipped map {
         case (num, str) => s"$num is the $str"
       }
mapped: scala.concurrent.Future[String] = ...

scala> mapped.value
res2: Option[scala.util.Try[String]] =
    Some(Success(42 is the answer))
```

The zipWith method allows you to perform the same operation in one step:

```
scala> val fut = futNum.zipWith(futStr) { // Scala 2.12
         case (num, str) => s"$num is the $str"
       }
zipWithed: scala.concurrent.Future[String] = ...

scala> fut.value
res3: Option[scala.util.Try[String]] =
    Some(Success(42 is the answer))
```

Future also gained a transformWith method in Scala 2.12, which allows you to transform a future using a function from Try to Future. Here's an example:

```
scala> val flipped = success.transformWith { // Scala 2.12
         case Success(res) =>
           Future { throw new Exception(res.toString) }
         case Failure(ex) => Future { 21 + 21 }
       }
flipped: scala.concurrent.Future[Int] = ...
```

```
scala> flipped.value
res5: Option[scala.util.Try[Int]] =
     Some(Failure(java.lang.Exception: 42))
```

The transformWith method is similar to the new, overloaded transform method added in Scala 2.12, except instead of yielding a Try in your passed function as in transform, transformWithallows you to yield a future.

32.4 TESTING WITH FUTURES

One advantage of Scala's futures is that they help you avoid blocking. On most JVM implementations, after creating just a few thousand threads, the cost of context switching between threads will degrade performance to an unnacceptable level. By avoiding blocking, you can keep the finite number of threads you decide to work with busy. Nevertheless, Scala does allow you to block on a future result when you need to. Scala's Await object facilitates blocking to wait for future results. Here's an example:

```
scala> import scala.concurrent.Await
import scala.concurrent.Await

scala> import scala.concurrent.duration._
import scala.concurrent.duration._

scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...

scala> val x = Await.result(fut, 15.seconds) // blocks
x: Int = 42
```

Await.result takes a Future and a Duration. The Duration indicates how long Await.result should wait for a Future to complete before timing out. In this example, fifteen seconds was specified for the Duration. Thus the Await.result method should not time out before the future completes with its eventual value, 42.

One place where blocking has been generally accepted is in tests of asynchronous code. Now that the Await.result has returned, you can perform a computation using that result, such as an assertion in a test:

```
scala> import org.scalatest.Matchers._
import org.scalatest.Matchers._

scala> x should be (42)
res0: org.scalatest.Assertion = Succeeded
```

Alternatively, you can use blocking constructs provided by ScalaTest's trait ScalaFutures. For example, the futureValue method, implicitly added to Future by ScalaFutures, will block until the future completes. If the future fails, futureValue will throw a TestFailedException describing the problem. If the future succeeds, futureValue will return the successful result of the future so you can perform asssertions on that value:

```
scala> import org.scalatest.concurrent.ScalaFutures._
import org.scalatest.concurrent.ScalaFutures._
```

```
scala> val fut = Future { Thread.sleep(10000); 21 + 21 }
fut: scala.concurrent.Future[Int] = ...

scala> fut.futureValue should be (42) // futureValue blocks
res1: org.scalatest.Assertion = Succeeded
```

While blocking in tests is often fine, ScalaTest 3.0 adds "async" testing styles that allow you to test futures without blocking. Given a future, instead of blocking and performing assertions on the result, you can map assertions directly onto that future and return the resultingFuture[Assertion] to ScalaTest. An example is shown in Listing 32.1. When the future assertion completes, ScalaTest will fire events (test succeeded, test failed, *etc.*) to the test reporter asynchronously.

```
import org.scalatest.AsyncFunSpec
import scala.concurrent.Future

class AddSpec extends AsyncFunSpec {

  def addSoon(addends: Int*): Future[Int] =
        Future { addends.sum }

  describe("addSoon") {
    it("will eventually compute a sum of passed Ints") {
      val futureSum: Future[Int] = addSoon(1, 2)
      // You can map assertions onto a Future, then return
      // the resulting Future[Assertion] to ScalaTest:
      futureSum map { sum => assert(sum == 3) }
    }
  }
}
```

Listing 32.1 - Returning a future assertion to ScalaTest.

The async testing use case illustrates a general principle for working with futures: Once in "future space," try to stay in future space. Don't block on a future then continue the computation with the result. Stay asynchronous by performing a series of transformations, each of which returns a new future to transform. To get results out of future space, register side effects to be performed asynchronously once futures complete. This approach will help you make maximum use of your threads.

32.5 CONCLUSION

Concurrent programming gives you great power. It lets you simplify your code and take advantage of multiple processors. It's unfortunate that the most widely used concurrency primitives, threads, locks, and monitors, are such a minefield of deadlocks and race conditions. Futures provide a way out of that minefield, letting you write concurrent programs without as great a risk of deadlocks and race conditions. This chapter has introduced several fundamental constructs for working with futures in Scala, including how to create futures, how to transform them, and how to test them, among other nuts and bolts. It then showed you how to use these constructs as part of a general futures style.

Footnotes for Chapter 32:

[1] On Scala.js, the global execution context places tasks on the JavaScript event queue.

[2] Note that the Java Future also has a way to deal with the potential of an exception being thrown by the asynchronous computation: its get method will throw that exception wrapped in an ExecutionException.

[3] The for expression shown in this example will be rewritten as a call to fut1.flatMap passing in a function that calls fut2.map: fut1.flatMap(x => fut2.map(y => x + y)).