Let's assume that we have a hypothetical pair of methods called `releaseCopy` and `acquireCopy`. The `releaseCopy` method assumes that the `Entry` class is in the `Copying` state and changes the state from `Copying` to another `Copying` or `Idle` state. It then returns the old `Copying` object associated with the previous state:

```
def releaseCopy(e: Entry): Copying = e.state.get match {
  case c: Copying =>
    val nstate = if (c.n == 1) new Idle else new Copying(c.n - 1)
    if (e.state.compareAndSet(c, nstate)) c
    else releaseCopy(e)
}
```

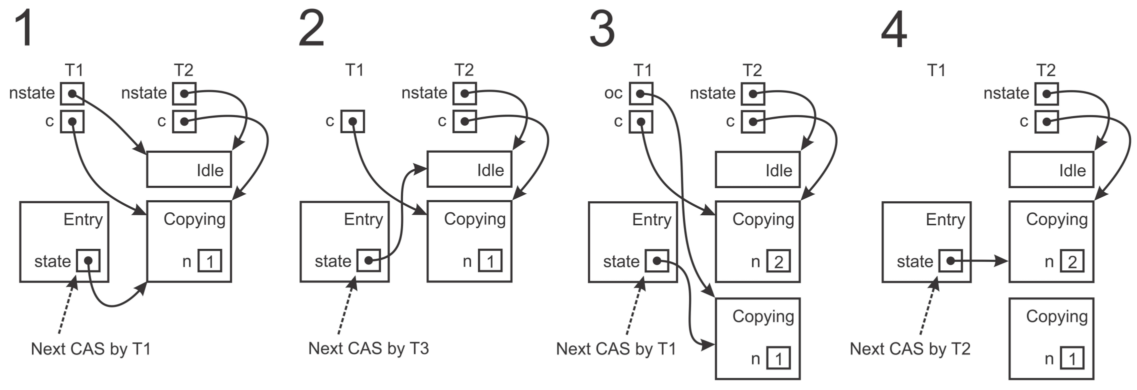
The `acquireCopy` method takes a currently unused `Copying` object and attempts to replace the old state with the previously used `Copying` object:

```
def acquireCopy(e: Entry, c: Copying) = e.state.get match {
  case i: Idle =>
    c.n = 1
    if (!e.state.compareAndSet(i, c)) acquire(e, c)
  case oc: Copying =>
    c.n = oc.n + 1
    if (!e.state.compareAndSet(oc, c)) acquire(e, c)
}
```

Upon calling the `releaseCopy` method, a thread might store the old `Copying` object. Later, the same thread can reuse the old `Copying` object in the call to the `acquireCopy` method. Here, the programmer's intent could be to reduce the pressure on the garbage collector by allocating fewer `Copying` objects. However, this leads to the ABA problem, as we will describe further.

We consider two threads **T1** and **T2**, which call the `releaseCopy` method. They both read the state of the `Entry` object and create a new state object `nstate`, which is `Idle`. Let's assume that the thread **T1** executes the `compareAndSet` operation first and returns the old `Copying` object `c` from the `releaseCopy` method. Next, let's assume that a third thread **T3** calls the `acquireCopy` method and changes the state of the `Entry` object to `Copying(1)`. If the thread **T1** now calls the `acquireCopy` method with the old `Copying` object `c`, the state of the `Entry` object becomes `Copying(2)`. Note that, at this point, the old `Copying` object `c` is once again stored inside the atomic variable `state`. If the thread **T1** now attempts to call `compareAndSet`, it will succeed and set the state of the `Entry` object to `Idle`. Effectively, the last `compareAndSet` operation changes the state from `Copying(2)` to `Idle`, so one acquire is lost.

This scenario is shown in the following figure:



In the preceding example, the ABA problem manifests itself in the execution of thread **T2**. Having first read the value of the `state` field in the `Entry` object with the `get` method and with the `compareAndSet` method later, thread **T2** assumes that the value of the `state` field has not changed between these two writes. In this case, this leads to a program error.

There is no general technique to avoid the ABA problem, so we need to guard the program against it on a per-problem basis. Still, the following guidelines are useful when avoiding the ABA problem in a managed runtime, such as JVM:

- Create new objects before assigning them to the `AtomicReference` objects
- Store immutable objects inside the `AtomicReference` objects
- Avoid assigning a value that was previously already assigned to an atomic variable
- If possible, make updates to numeric atomic variables monotonic, that is, either strictly decreasing or strictly increasing with respect to the previous value

There are other techniques in order to avoid the ABA problem, such as pointer masking and hazard pointers, but these are not applicable to JVM.

In some cases, the ABA problem does not affect the correctness of the algorithm; for example, if we change the `Idle` class to a singleton object, the `prepareForDelete` method will continue to work correctly. Still, it is a good practice to follow the preceding guidelines, because they simplify the reasoning about lock-free algorithms.

Lazy values

You should be familiar with lazy values from sequential programming in Scala. Lazy values are the value declarations that are initialized with their right-hand side expression when the lazy value is read for the first time. This is unlike regular values, which are initialized the moment they are created. If a lazy value is never read inside the program, it is never initialized and it is not necessary to pay the cost of its initialization. Lazy values allow you to implement data structures such as lazy streams; they improve complexities of persistent data structures, can boost the program's performance, and help avoid initialization order problems in Scala's mix-in composition.

Lazy values are extremely useful in practice, and you will often deal with them in Scala. However, using them in concurrent programs can have some unexpected interactions, and this is the topic of this section. Note that lazy values must retain the same semantics in a multithreaded program; a lazy value is initialized only when a thread accesses it, and it is initialized at most once. Consider the following motivating example in which two threads access two lazy values, which are `obj` and `non`:

```
object LazyValsCreate extends App {
  lazy val obj = new AnyRef
  lazy val non = s"made by ${Thread.currentThread.getName}"
  execute {
    log(s"EC sees obj = $obj")
    log(s"EC sees non = $non")
  }
  log(s"Main sees obj = $obj")
  log(s"Main sees non = $non")
  Thread.sleep(500)
}
```

You know from sequential Scala programming that it is a good practice to initialize the lazy value with an expression that does not depend on the current state of the program. The lazy value `obj` follows this practice, but the lazy value `non` does not. If you run this program once, you might notice that `non` lazy value is initialized with the name of the main thread:

```
[info] main: Main sees non = made by main
[info] FJPool-1-worker-13: EC sees non = made by main
```

Running the program again shows you that `non` is initialized by the worker thread:

```
[info] main: Main sees non = made by FJPool-1-worker-13
[info] FJPool-1-worker-13: EC sees non = made by FJPool-1-worker-13
```

As the previous example shows you, lazy values are affected by non-determinism. Non-deterministic lazy values are a recipe for trouble, but we cannot always avoid them. Lazy values are deeply tied into Scala, because singleton objects are implemented as lazy values under the hood:

```
object LazyValsObject extends App {
  object Lazy { log("Running Lazy constructor.") }
  log("Main thread is about to reference Lazy.")
  Lazy
  log("Main thread completed.")
}
```

Running this program reveals that the `Lazy` initializer runs when the object is first referenced in the third line and not when it is declared. Getting rid of singleton objects in your Scala code would be too restrictive, and singleton objects are often large; they can contain all kinds of potentially non-deterministic code.

You might think that a little bit of non-determinism is something we can live with. However, this non-determinism can be dangerous. In the existing Scala versions, lazy values and singleton objects are implemented with the so-called *double-checked locking idiom* under the hood. This concurrent programming pattern ensures that a lazy value is initialized by at most one thread when it is first accessed. Thanks to this pattern, upon initializing the lazy value, its subsequent reads are cheap and do not need to acquire any locks. Using this idiom, a single lazy value declaration, which is `obj` from the previous example, is translated by the Scala compiler as follows:

```
object LazyValsUnderTheHood extends App {
  @volatile private var _bitmap = false
  private var _obj: AnyRef = _
  def obj = if (_bitmap) _obj else this.synchronized {
    if (!_bitmap) {
      _obj = new AnyRef
      _bitmap = true
    }
    _obj
  }
  log(s"$obj")
  log(s"$obj")
}
```

The Scala compiler introduces an additional volatile field, `_bitmap`, when a class contains lazy fields. The private `_obj` field that holds the value is uninitialized at first. After the `obj` getter assigns a value to the `_obj` field, it sets the `_bitmap` field to `true` to indicate that the lazy value was initialized. Other subsequent invocations of the getter know whether they can read the lazy value from the `_obj` field by checking the `_bitmap` field.

The getter `obj` starts by checking whether the `_bitmap` field is `true`. If `_bitmap` field is `true`, then the lazy value was already initialized and the getter returns `_obj`. Otherwise, the getter `obj` attempts to obtain the intrinsic lock of the enclosing object, in this case, `LazyValsUnderTheHood`. If the `_bitmap` field is still not set from within the `synchronized` block, the getter evaluates the `new AnyRef` expression, assigns it to `_obj`, and sets `_bitmap` to `true`. After this point, the lazy value is considered initialized. Note that the `synchronized` statement, together with the check that the `_bitmap` field is `false`, ensure that a single thread at most initializes the lazy value.

The double-checked locking idiom ensures that every lazy value is initialized by at most a single thread.

This mechanism is robust and ensures that lazy values are both thread-safe and efficient. However, synchronization on the enclosing object can cause problems. Consider the following example in which two threads attempt to initialize lazy values `A.x` and `B.y` at the same time:

```
object LazyValsDeadlock extends App {
  object A { lazy val x: Int = B.y }
  object B { lazy val y: Int = A.x }
  execute { B.y }
  A.x
}
```

In a sequential setting, accessing either `A.x` or `B.y` results in a stack overflow. Initializing `A.x` requires calling the getter for `B.y`, which is not initialized. Initialization of `B.y` calls the getter for `A.x` and continues in infinite recursion. However, this example was carefully tuned to access both `A.x` and `B.y` at the same time by both the main thread and the worker thread. Prepare to restart SBT. After both `A` and `B` are initialized, their monitors are acquired simultaneously by two different threads. Each of these threads needs to acquire a monitor owned by the other thread. Neither thread lets go of its own monitor, and this results in a deadlock.

Cyclic dependencies between lazy values are unsupported in both sequential and concurrent Scala programs. The difference is that they potentially manifest themselves as deadlocks instead of stack overflows in concurrent programming.

Avoid cyclic dependencies between lazy values, as they can cause deadlocks.

The previous example is not something you are likely to do in your code, but cyclic dependencies between lazy values and singleton objects can be much more devious and harder to spot. In fact, there are other ways to create dependencies between lazy values besides directly accessing them. A lazy value initialization expression can block a thread until some other value becomes available. In the following example, the initialization expression uses the `thread` statement from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, to start a new thread and join it:

```
object LazyValsAndBlocking extends App {
  lazy val x: Int = {
    val t = ch2.thread { println(s"Initializing $x.") }
    t.join()
    1
  }
  x
}
```

Although there is only a single lazy value in this example, running it inevitably results in a deadlock. The new thread attempts to access `x`, which is not initialized, so it attempts to call the `synchronized` statement on the `LazyValsAndBlocking` object and blocks, because the main thread already holds this lock. On the other hand, the main thread waits for the other thread to terminate, so neither thread can progress.

While the deadlock is relatively obvious in this example, in a larger code base, circular dependencies can easily sneak past your guard. In some cases, they might even be non-deterministic and occur only in particular system states. To guard against them, avoid blocking in the lazy value expression altogether.

Never invoke blocking operations inside lazy value initialization expressions or singleton object constructors.

Lazy values cause deadlocks even when they do not block themselves. In the following example, the main thread calls the `synchronized` statement on the enclosing object, starts a new thread, and waits for its termination. The new thread attempts to initialize the lazy value `x`, but it cannot acquire the monitor until the main thread releases it:

```
object LazyValsAndMonitors extends App {
  lazy val x = 1
  this.synchronized {
    val t = ch2.thread { x }
    t.join()
  }
}
```

This kind of deadlock is not inherent to lazy values and can happen with arbitrary code that uses `synchronized` statements. The problem is that the `LazyValsAndMonitors` lock is used in two different contexts: as a lazy value initialization lock and as the lock for some custom logic in the main thread. To prevent two unrelated software components from using the same lock, always call `synchronized` on separate private objects that exist solely for this purpose.

Never call `synchronized` on publicly available objects; always use a dedicated, private dummy object for synchronization.

Although we rarely use separate objects for synchronization in this book, to keep the examples concise, you should strongly consider doing this in your programs. This tip is useful outside the context of lazy values; keeping your locks private reduces the possibility of deadlocks.

Concurrent collections

As you can conclude from the discussion on the Java Memory Model in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, modifying the Scala standard library collections from different threads can result in arbitrary data corruption. Standard collection implementations do not use any synchronization. Data structures underlying mutable collections can be quite complex; predicting how multiple threads affect the collection state in the absence of synchronization is neither recommended nor possible. We will demonstrate this by letting two threads add numbers to the `mutable.ArrayBuffer` collection:

```
import scala.collection._
object CollectionsBad extends App {
  val buffer = mutable.ArrayBuffer[Int]()
  def asyncAdd(numbers: Seq[Int]) = execute {
    buffer ++= numbers
    log(s"buffer = $buffer")
  }
  asyncAdd(0 until 10)
  asyncAdd(10 until 20)
  Thread.sleep(500)
}
```

Instead of printing an array buffer with 20 different elements, this example arbitrarily prints different results or throws exceptions each time it runs. The two threads modify the internal array buffer state simultaneously and cause data corruption.

Never use mutable collections from several different threads without applying proper synchronization.

We can restore synchronization in several ways. First, we can use **immutable collections** along with synchronization to share them between threads. For example, we can store immutable data structures inside atomic reference variables. In the following code snippet, we introduce an `AtomicBuffer` class that allows concurrent `+=` operations. Appending reads the current immutable `List` value from the atomic reference buffer and creates a new `List` object containing `x`. It then invokes a CAS operation to atomically update the buffer, retrying the operation if the CAS operation is not successful:

```
class AtomicBuffer[T] {
  private val buffer = new AtomicReference[List[T]](Nil)
  @tailrec def +=(x: T): Unit = {
    val xs = buffer.get
```

```
      val nxs = x :: xs
      if (!buffer.compareAndSet(xs, nxs)) this += x
    }
  }
```

While using atomic variables or the `synchronized` statements with immutable collections is simple, it can lead to scalability problems when many threads access an atomic variable at once.

If we intend to continue using mutable collections, we need to add `synchronized` statements around calls to collection operations:

```
  def asyncAdd(numbers: Seq[Int]) = execute {
    buffer.synchronized {
      buffer ++= numbers
      log(s"buffer = $buffer")
    }
  }
```

This approach can be satisfactory, provided that collection operations do not block inside `synchronized`. In fact, this approach allows you to implement guarded blocks around collection operations, as we saw in the `SynchronizedPool` example in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. However, using the `synchronized` statement can also lead to scalability problems when many threads attempt to acquire the lock at once.

Finally, concurrent collections are collection implementations with operations that can be safely invoked from different threads without synchronization. In addition to the thread-safe versions of basic collection operations, some concurrent collections provide more expressive operations. Conceptually, the same operations can be achieved using atomic primitives, `synchronized` statements, and guarded blocks, but concurrent collections ensure far better performance and scalability.

Concurrent queues

A common pattern used in concurrent programming is the **producer-consumer pattern**. In this pattern, the responsibility for different parts of the computational workload is divided across several threads. In an FTP server, one or more threads can be responsible for reading chunks of a large file from the disk. Such threads are called producers. Another dedicated set of threads can bear the responsibility of sending file chunks through the network. We call these threads consumers. In their relationship, consumers must react to work elements created by the producers. Often, the two are not perfectly synchronized, so work elements need to be buffered somewhere.

The concurrent collection that supports this kind of buffering is called a **concurrent queue**. There are three main operations we expect from a concurrent queue. The enqueue operation allows producers to add work elements to the queue, and the dequeue operation allows consumers to remove them. Finally, sometimes we want to check whether the queue is empty or inspect the value of the next item without changing the queue's contents. Concurrent queues can be **bounded**, which means that they can only contain a maximum number of elements, or they can be **unbounded**, which means that they can grow indefinitely. When a bounded queue contains the maximum number of elements, we say it is full. The semantics of the various versions of enqueue and dequeue operations differ with respect to what happens when we try to enqueue to a full queue or dequeue from an empty queue. This special case needs to be handled differently by the concurrent queue. In single-threaded programming, sequential queues usually return a special value such as `null` or `false` when they are full or empty, or simply throw an exception. In concurrent programming, the absence of elements in the queue can indicate that the producer has not yet enqueued an element, although it might enqueue it in the future. Similarly, a full queue means that the consumer did not yet remove elements but will do so later. For this reason, some concurrent queues have *blocking* enqueue and dequeue implementations, which block the caller until the queue is non-full or non-empty, respectively.

JDK represents multiple efficient concurrent queue implementations in the `java.util.concurrent` package with the `BlockingQueue` interface. Rather than reinventing the wheel with its own concurrent queue implementations, Scala adopts these concurrent queues as part of its concurrency utilities and it does not currently have a dedicated trait for blocking queues.

The `BlockingQueue` interface contains several versions of the basic concurrent queue operations, each with slightly different semantics. Different variants of their enqueue, dequeue, and inspect-next methods are summarized in the following table. The inspect, dequeue, and enqueue versions are called `element`, `remove`, and `add` in the first column; they throw an exception when the queue is empty or full. Methods such as `poll` and `offer` return special values such as `null` or `false`. Timed versions of these methods block the caller for a specified duration before returning an element or a special value, and blocking methods block the calling thread until the queue becomes non-empty or non-full.

Operation	**Exception**	**Special value**	**Timed**	**Blocking**
Dequeue	remove(): T	poll(): T	poll(t: Long, u: TimeUnit): T	take(): T
Enqueue	add(x: T)	offer(x: T): Boolean	offer(x: T, t: Long, u: TimeUnit)	put(x: T)
Inspect	element: T	peek: T	N/A	N/A

The `ArrayBlockingQueue` class is a concrete implementation of a bounded blocking queue. When creating the `ArrayBlockingQueue` class, we need to specify its capacity, which is the number of elements in the queue when it is full. If producers can potentially create elements faster than the consumers can process them, we need to use bounded queues. Otherwise, the queue size can potentially grow to the point where it consumes all the available memory in the program.

Another concurrent queue implementation is called `LinkedBlockingQueue`. This queue is unbounded, and we can use it when we are sure that the consumers work much faster than the producers. This queue is an ideal candidate for the logging component of our filesystem's API. Logging must return feedback about the execution to the user. In a file manager, logging produces messages to the user inside the UI, while in an FTP server it sends feedback over the network. To keep the example simple, we just print the messages to the standard output.

We use the `LinkedBlockingQueue` collection to buffer various messages from different components of the filesystem API. We declare the queue to a private variable called `messages`. A separate daemon thread, called `logger`, repetitively calls the `take` method on messages. Recall from the previous table that the `take` method is blocking; it blocks the `logger` thread until there is a message in the queue. The `logger` thread then calls `log` to print the message. The `logMessage` method, which we used in the `prepareForDelete` method earlier, simply calls the `offer` method on the `messages` queue. We could have alternatively called `add` or `put`. We know that the queue is unbounded, so these methods never throw or block:

```
private val messages = new LinkedBlockingQueue[String]
val logger = new Thread {
  setDaemon(true)
  override def run() = while (true) log(messages.take())
}
logger.start()
def logMessage(msg: String): Unit = messages.offer(msg)
```

We place these methods and the previously defined `prepareForDelete` method into the `FileSystem` class. To test this, we can simply instantiate our `FileSystem` class and call the `logMessage` method. Once the main thread terminates, the `logger` thread automatically stops:

```
val fileSystem = new FileSystem(".")
fileSystem.logMessage("Testing log!")
```

An important difference between sequential queues and concurrent queues is that concurrent queues have **weakly consistent iterators**. An iterator created with the `iterator` method traverses the elements that were in the queue at the moment the `iterator` method was created. However, if there is an enqueue or dequeue operation before the traversal is over, all bets are off, and the iterator might or might not reflect any modifications. Consider the following example, in which one thread traverses the concurrent queue while another thread dequeues its elements:

```
object CollectionsIterators extends App {
  val queue = new LinkedBlockingQueue[String]
  for (i <- 1 to 5500) queue.offer(i.toString)
  execute {
    val it = queue.iterator
    while (it.hasNext) log(it.next())
  }
  for (i <- 1 to 5500) queue.poll()
  Thread.sleep(1000)
}
```

The main thread creates a queue with 5,500 elements. It then starts a concurrent task that creates an iterator and prints the elements one by one. At the same time, the main thread starts removing all the elements from the queue in the same order. In one of our thread runs, the iterator returns `1`, `4,779`, and `5,442`. This does not make sense, because the queue never contained these three elements alone; we would expect to see a suffix that has the range of `1` to `5,500`. We say that the iterator is not consistent. It is never corrupt and does not throw exceptions, but it fails to return a consistent set of elements that were in the queue at some point. With a few notable exceptions, this effect can happen when using any concurrent data structure.

Use iterators on concurrent data structures only when you can ensure that no other thread will modify the data structure from the point where the iterator was created until the point where the iterator's `hasNext` method returns `false`.

The `CopyOnWriteArrayList` and `CopyOnWriteArraySet` collections in JDK are exceptions to this rule, but they copy the underlying data whenever the collection is mutated and can be slow. Later in this section, we will see a concurrent collection from the `scala.collection.concurrent` package called `TrieMap`, which creates consistent iterators without copying the underlying dataset and allows arbitrary modifications during the traversal.

Concurrent sets and maps

Concurrent API designers strive to provide programmers with interfaces that resemble those from sequential programming. We have seen that this is the case with concurrent queues. As the main use case for concurrent queues is the producer-consumer pattern, the `BlockingQueue` interface additionally provides blocking versions of methods that are already known from sequential queues. Concurrent maps and concurrent sets are map and set collections, respectively, that can be safely accessed and modified by multiple threads. Like concurrent queues, they retain the API from the corresponding sequential collections. Unlike concurrent queues, they do not have blocking operations. The reason is that their principal use case is not the producer-consumer pattern, but encoding the program state.

The `concurrent.Map` trait in the `scala.collection` package represents different concurrent map implementations. In our filesystem API, we use it to track the files that exist in the filesystem as follows:

```
val files: concurrent.Map[String, Entry]
```

This concurrent map contains paths and their corresponding `Entry` objects. These are the same `Entry` objects that `prepareForDelete` used earlier. The concurrent `files` map is populated when the `FileSystem` object is created.

For the examples in this section, we add the following dependency to our `build.sbt` file. This will allow us to use the Apache `Commons IO` library in order to handle files:

```
libraryDependencies += "commons-io" % "commons-io" % "2.4"
```

We will allow `FileSystem` objects to only track files in a certain directory called `root`. By instantiating the `FileSystem` object with the `"."` string, we set the `root` directory to the root of our project with the example code. This way, the worst thing that can happen is that you delete all your examples by accident and have to rewrite them once more. However, that's okay, as practice makes perfect! The `FileSystem` class is shown in the following snippet:

```
import scala.collection.convert.decorateAsScala._
import java.io.File
import org.apache.commons.io.FileUtils
class FileSystem(val root: String) {
  val rootDir = new File(root)
  val files: concurrent.Map[String, Entry] =
    new ConcurrentHashMap().asScala
  for (f <- FileUtils.iterateFiles(rootDir, null, false).asScala)
  files.put(f.getName, new Entry(false))
}
```

We first create a new `ConcurrentHashMap` method from the `java.util.concurrent` package and wrap it to a Scala `concurrent.Map` trait by calling `asScala`. This method can be called to wrap most Java collections, provided the contents of the `decorateAsScala` object are imported like they are in our example. The `asScala` method ensures that Java collections obtain the Scala collection API. The `iterateFiles` method in the `FileUtils` class returns a Java iterator that traverses the files in a specific folder; we can only use Scala iterators in `for` comprehensions, so we call `asScala` again. The first argument for the `iterateFiles` method specifies the `root` folder, and the second method specifies an optional filter for the files. The final `false` argument for the `iterateFiles` method denotes that we do not scan files recursively in the subdirectories of `root`. We play it safe and expose only files in our `root` project directory to the `FileSystem` class. We place each `f` file along with a fresh `Entry` object into `files` by calling `put` on the concurrent map. There is no need to use a `synchronized` statement around `put`, as the concurrent map takes care of synchronization and thread-safety. The `put` operation is atomic, and it establishes a happens-before relationship with subsequent `get` operations.

The same is true for the other methods such as `remove`, which removes key-value pairs from a concurrent map. We can now use the `prepareForDelete` method implemented earlier to atomically lock a file for deletion and then remove it from the `files` map. We implement the `deleteFile` method for this purpose:

```
def deleteFile(filename: String): Unit = {
  files.get(filename) match {
    case None =>
      logMessage(s"Path '$filename' does not exist!")
    case Some(entry) if entry.isDir =>
      logMessage(s"Path '$filename' is a directory!")
    case Some(entry) => execute {
      if (prepareForDelete(entry))
        if (FileUtils.deleteQuietly(new File(filename)))
          files.remove(filename)
    }
  }
}
```

If the `deleteFile` method finds that the concurrent map contains the file with the given name, it calls the `execute` method to asynchronously delete it, as we prefer not to block the caller thread. The concurrent task, started by the `execute` invocation, calls the `prepareForDelete` method. If the `prepareForDelete` method returns `true`, then it is safe to call the `deleteQuietly` method from the `Commons IO` library. This method physically removes the file from the disk. If the deletion is successful, the file entry is removed from the `files` map. We create a new file called `test.txt` and use it to test the `deleteFile` method. We prefer not to experiment with the build definition file. The following code shows the deletion of the file:

```
fileSystem.deleteFile("test.txt")
```

The second time we run this line, our logger thread from before complains that the file does not exist. A quick check in our file manager reveals that the `test.txt` file is no longer there.

The `concurrent.Map` trait also defines several complex linearizable methods. Recall that complex linearizable operations involve multiple reads and writes. In the context of concurrent maps, methods are complex linearizable operations if they involve multiple instances of the `get` and `put` methods, but appear to get executed at a single point in time. Such methods are a powerful tool in our concurrency arsenal. We have already seen that volatile reads and writes do not allow us to implement the `getUniqueId` method; we need the `compareAndSet` method for that. Similar methods on concurrent maps have comparable advantages. Different atomic methods on atomic maps are summarized in the following table. Note that, unlike the CAS instruction, these methods use structural equality to compare keys and values, and they call the `equals` method.

Signature	**Description**
putIfAbsent (k: K, v: V): Option[V]	**This atomically assigns the value `v` to the key `k` if `k` is not in the map. Otherwise, it returns the value associated with `k`.**
remove (k: K, v: V): Boolean	This atomically removes the key `k` if it is associated to the value equal to `v` and returns `true` if successful.
replace (k: K, v: V): Option[V]	This atomically assigns the value `v` to the key `k` and returns the value previously associated with `k`.
replace (k: K, ov: V, nv: V): Boolean	This atomically assigns the key `k` to the value `nv` if `k` was previously associated with `ov` and returns `true` if successful.

Coming back to our filesystem API, let's see how these methods work to our advantage. We will now implement the `copyFile` method in the `FileSystem` class. Recall the diagram from the section on atomic variables. A copy operation can start only if the file is either in the `Idle` state or already in the `Copying` state, so we need to atomically switch the file state from `Idle` to `Copying` or from the `Copying` state to another `Copying` state with the value `n` incremented. We do this with the `acquire` method:

```
@tailrec private def acquire(entry: Entry): Boolean = {
  val s0 = entry.state.get
  s0 match {
    case _: Creating | _: Deleting =>
      logMessage("File inaccessible, cannot copy."); false
    case i: Idle =>
      if (entry.state.compareAndSet(s0, new Copying(1))) true
      else acquire(entry)
    case c: Copying =>
      if (entry.state.compareAndSet(s0, new Copying(c.n+1))) true
      else acquire(entry)
  }
}
```

After a thread completes copying a file, it needs to release the `Copying` lock. This is done by a similar `release` method, which decreases the `Copying` count or changes the state to `Idle`. Importantly, this method must be called after files are newly created in order to switch from the `Creating` state to the `Idle` state. By now, the retry pattern following unsuccessful CAS operations should be child's play for you. The following code shows this:

```
@tailrec private def release(entry: Entry): Unit = {
  Val s0 = entry.state.get
  s0 match {
    case c: Creating =>
      if (!entry.state.compareAndSet(s0, new Idle)) release(entry)
    case c: Copying =>
      val nstate = if (c.n == 1) new Idle else new Copying(c.n-1)
      if (!entry.state.compareAndSet(s0, nstate)) release(entry)
  }
}
```

We now have all the machinery required to implement the `copyFile` method. This method checks whether an `src` entry exists in the `files` map. If the entry exists, the `copyFile` method starts a concurrent task to copy the file. The concurrent task attempts to acquire the file for copying and creates a new `destEntry` file entry in the `Creating` state. It then calls the `putIfAbsent` method, which atomically checks whether the file path `dest` is a key in the map and adds the `dest` and `destEntry` pair if it is not. Both the `srcEntry` and `destEntry` value pair are locked at this point, so the `FileUtils.copyFile` method from the `Commons IO` library is called to copy the file on the disk. Once the copying is complete, both the `srcEntry` and `destEntry` value pair are released:

```
def copyFile(src: String, dest: String): Unit = {
  files.get(src) match {
    case Some(srcEntry) if !srcEntry.isDir => execute {
      if (acquire(srcEntry)) try {
        val destEntry = new Entry(isDir = false)
        destEntry.state.set(new Creating)
        if (files.putIfAbsent(dest, destEntry) == None) try {
          FileUtils.copyFile(new File(src), new File(dest))
        } finally release(destEntry)
      } finally release(srcEntry)
    }
  }
}
```

You should convince yourself that the `copyFile` method would be incorrect if it first called `get` to check whether `dest` is in the map and then called `put` to place `dest` in the map. This would allow another thread's `get` and `put` steps to interleave and potentially overwrite an entry in the `files` map. This demonstrates the importance of the `putIfAbsent` method.

There are some methods that the `concurrent.Map` trait inherits from the `mutable.Map` trait and that are not atomic. An example is the `getOrElseUpdate` method, which retrieves an element if it is present in the map and updates it with a different element otherwise. This method is not atomic, while its individual steps are atomic; they can be interleaved arbitrarily with concurrent calls to the `getOrElseUpdate` method. Another example is `clear`, which does not have to be atomic on concurrent collections in general and can behave like the concurrent data structure iterators we studied before.

The`+=`, `-=`, `put`, `update`, `get`, `apply`, and `remove` methods in the `concurrent.Map` trait are linearizable methods. The `putIfAbsent`, conditional `remove`, and `replace` methods in the `concurrent.Map` trait are the only complex methods guaranteed to be linearizable.

Just like the Java concurrency library, Scala currently does not have a dedicated trait for concurrent sets. A concurrent set of the `Set[T]` type can be emulated with a concurrent map with the `ConcurrentMap[T, Unit]` type, which ignores the values assigned to keys. This is the reason why concrete concurrent set implementations appear less often in concurrency frameworks. In rare situations, where a Java concurrent set, such as the `ConcurrentSkipListSet[T]` class, needs to be converted to a Scala concurrent set, we can use the `asScala` method, which converts it to a `mutable.Set[T]` class.

As a final note, you should never use `null` as a key or value in a concurrent map or a concurrent set. Many concurrent data structure implementations on JVM rely on using `null` as a special indicator of the absence of an element.

Avoid using the `null` value as a key or a value in a concurrent data structure.

Some implementations are defensive and will throw an exception; for others, the results might be undefined. Even when a concurrent collection specifies that `null` is allowed, you should avoid coupling `null` with your program logic in order to make potential refactoring easier.

Concurrent traversals

As you had a chance to witness, Scala directly inherits many of its basic concurrency utilities from the Java concurrency packages. After all, these facilities were implemented by JVM's concurrency experts. Apart from providing conversions that make Java's traditional concurrency utilities feel Scala-idiomatic, there is no need to reinvent what's already there. When it comes to concurrent collections, a particularly bothersome limitation is that you cannot safely traverse most concurrent collections and modify them in the same time. This is not so problematic for sequential collections where we control the thread that calls the `foreach` loop or uses iterators. In a concurrent system where threads are not perfectly synchronized with each other, it is much harder to guarantee that there will be no modifications during the traversal.

Fortunately, Scala has an answer for concurrent collection traversals. The `TrieMap` collection from the `scala.collection.concurrent` package, which is based on the concurrent **Ctrie** data structure, is a concurrent map implementation that produces consistent iterators. When its `iterator` method is called, the `TrieMap` collection atomically takes a snapshot of all the elements. A **snapshot** is complete information about the state of a data structure. The iterator then uses this snapshot to traverse the elements. If the `TrieMap` collection is later modified during the traversal, the modifications are not visible in the snapshot and the iterator does not reflect them. You might conclude that taking a snapshot is expensive and requires copying all the elements, but this is not the case. The `snapshot` method of the `TrieMap` class incrementally rebuilds parts of the `TrieMap` collection when they are first accessed by some thread. The `readOnlySnapshot` method, internally used by the `iterator` method, is even more efficient. It ensures that only the modified parts of the `TrieMap` collection are lazily copied. If there are no subsequent concurrent modifications, then no part of the `TrieMap` collection is ever copied.

Let's study the difference between the Java `ConcurrentHashMap` and the Scala `concurrent.TrieMap` collections in an example. Assume that we have a concurrent map that maps names to numerals in these names. For example, `"Jane"` will be mapped to `0`, but `"John"` will be mapped to `4`, and so on. In one concurrent task, we add different names for John in the order of `0` to `10` to the `ConcurrentHashMap` collection. We concurrently traverse the map and output these names:

```
object CollectionsConcurrentMapBulk extends App {
  val names = new ConcurrentHashMap[String, Int]().asScala
  names("Johnny") = 0; names("Jane") = 0; names("Jack") = 0
  execute {
    for (n <- 0 until 10) names(s"John $n") = n }
  execute {
    for (n <- names) log(s"name: $n") }
  Thread.sleep(1000)
}
```

If the iterator was consistent, we would expect to see the three names Johnny, Jane, and Jack that were initially in the map and the name John in the interval from 0 to an n value, depending on how many names the first task added; this could be John 1, John 2, or John 3. Instead, the output shows you random nonconsecutive names such as John 8 and John 5, which does not make sense. John 8 should never appear in the map without John 7, and other entries inserted earlier by the other task. This never happens in a concurrent TrieMap collection. We can run the same experiment with the TrieMap collection and sort the names lexicographically before outputting them. Running the following program always prints all the John names in the interval of 0 and some value n:

```
object CollectionsTrieMapBulk extends App {
  val names = new concurrent.TrieMap[String, Int]
  names("Janice") = 0; names("Jackie") = 0; names("Jill") = 0
  execute {for (n <- 10 until 100) names(s"John $n") = n}
  execute {
    log("snapshot time!")
    for (n <- names.map(_._1).toSeq.sorted) log(s"name: $n")
  }
}
```

How is this useful in practice? Imagine that we need to return a consistent snapshot of the filesystem; all the files are as seen by the file manager or an FTP server at a point in time. A TrieMap collection ensures that other threads that delete or copy files cannot interfere with the thread that is extracting the files. We thus use the TrieMap collection to store files in our filesystem API and define a simple allFiles method that returns all the files. At the point where we start using the files map in a for comprehension, a snapshot with the filesystem contents is created:

```
val files: concurrent.Map[String, Entry] = new concurrent.TrieMap()
def allFiles(): Iterable[String] = for ((name, state) <- files) yield name
```

We use the allFiles method to display all the files in the root directory:

```
val rootFiles = fileSystem.allFiles()
log("All files in the root dir: " + rootFiles.mkString(", "))
```

After having seen both these concurrent maps, you might be wondering about which one to use. This mainly depends on the use case. If the application requires consistent iterators, then you should definitely use the `TrieMap` collections. On the other hand, if the application does not require consistent iterators and rarely modifies the concurrent map, you can consider using `ConcurrentHashMap` collections, as their lookup operations are slightly faster.

Use `TrieMap` collections if you require consistent iterators and `ConcurrentHashMap` collections when the `get` and `apply` operations are the bottlenecks in your program.

From a performance point of view, this tip is only applicable if your application is exclusively accessing a concurrent map all the time and doing nothing else. In practice, this is rarely the case, and in most situations, you can use either of these collections.

Custom concurrent data structures

In this section, we will show how to design a concurrent data structure. The data structure we will use as a running example will be simple, but sufficient to demonstrate the general approach. You will be able to apply the same principles to more complex data structures.

Before we start, there is a disclaimer. Designing a concurrent data structure is hard, and, as a rule of the thumb, you should almost never do it. Even if you manage to implement a correct and efficient concurrent data structure, the cost of doing so is usually high.

There are several reasons why designing a concurrent data structure is hard. The first is achieving correctness: errors are much harder to notice, reproduce, or analyze due to inherent non-determinism. Then, operations must not slow down when more processors use the data structure. In other words, the data structure must be scalable. Finally, a concurrent data structure must be efficient in absolute terms, and it must not be much slower than its sequential counterpart when used with a single processor.

That said, we proceed to designing a concrete data structure: a concurrent pool.

Implementing a lock-free concurrent pool

In this section, we will implement a concurrent lock-free pool as an example of how to design a concurrent data structure. A **pool** is one of the simplest data structure abstractions, and only has two methods–the `add` and the `remove` operations. The `add` operation simply adds an element into the pool, but the `remove` operation is more limited than in a set or a map of elements. Instead of removing a specific element from the pool, the `remove` operation removes any element, as long as the pool is non-empty. A lock-free pool is a pool whose operations are lock-free.

Although simple, the pool abstraction is very useful, as it allows temporarily storing expensive objects (for example, worker threads or database connectors). For this use-case, we do not care about which exact element the `remove` operation returns, as long as it returns some element.

Determining its operations is the first step in designing a concurrent data structure. Knowing the operations and their exact semantics drives the rest of the design, and adding supplementary operations later is likely to break the invariants of the data structure. It is usually hard to correctly extend a concurrent data structure once it has already been implemented.

Having determined the operations that a concurrent data structure must support, the next step is to think about data representation. Since we decided that the operations must be lock-free, one seemingly reasonable choice is to encode the state as an `AtomicReference` object holding a pointer to an immutable list:

```
val pool = new AtomicReference[List[T]]
```

Both, the `add` and `remove` operations follow naturally from this choice. To add an element, we read the old list, use it to append the element at the head of the list, and then invoke a `compareAndSet` operation to replace the old list, retrying if necessary. Elements would be removed in a similar fashion.

However, such an implementation would not be very scalable. Multiple processors would need to access the same memory location, and retrying would occur frequently. The expected time to complete the operation would then be *O(P)*, where *P* is the number of processors that are concurrently executing `add` and `remove` operations.

To improve this, we will need to allow different processors to pick different memory locations in the data structure when updating it. The fact that we are implementing a pool mitigates this decision, since the `remove` operation will not have to search for specific elements, and just needs to return any element. Therefore, the `add` operation can append the element to any location in the data structure.

With this in mind, we choose an array of atomic references, each holding an immutable list, as our internal representation. Having many atomic references allows each processor to pick an arbitrary slot to perform the update. This is shown in the following snippet:

```
class Pool[T] {
  val parallelism = Runtime.getRuntime.availableProcessors * 32
  val buckets =
    new Array[AtomicReference[(List[T], Long)]](parallelism)
  for (i <- 0 until buckets.length)
    buckets(i) = new AtomicReference((Nil, 0L))
```

Note that each atomic reference holds not only a list of values in the respective bucket, but also a `Long` value. This unique numeric value will serve as a timestamp that must be incremented each time the bucket is modified. Before we see why having the timestamp is important, we will implement the `add` operation.

The `add` operation must pick one of the atomic references in the `buckets` array, create a new version of the list that contains the new element, and then invoke the CAS instruction until the respective atomic reference is updated. When picking a bucket, the processor must aim for a bucket that no other processor is currently using, to prevent contention and retries. There are many ways to achieve this, but we will settle for a relatively simple strategy–we compute the bucket from the thread ID, and the hash code of the element. Once the bucket is picked, the `add` operation follows the standard retry pattern that we saw earlier. This is shown in the following snippet:

```
  def add(x: T): Unit = {
    val i =
      (Thread.currentThread.getId ^ x.## % buckets.length).toInt
    @tailrec def retry() {
      val bucket = buckets(i)
      val v = bucket.get
      val (lst, stamp) = v
      val nlst = x :: lst
      val nstamp = stamp + 1
      val nv = (nlst, nstamp)
      if (!bucket.compareAndSet(v, nv)) retry()
    }
    retry()
  }
```

The `remove` operation is more complex. Unlike the `add` operation, which can pick any bucket when inserting an element, the `remove` operation must pick a non-empty bucket. The current design of the data structure offers no apriori way of knowing which bucket is non-empty, so the best we can do is pick some bucket, and scan the other buckets linearly until finding a non-empty bucket. This has two consequences. First, if our concurrent pool is nearly empty, we will need to scan all the buckets in the worst case scenario. The `remove` operation is only scalable if the pool is relatively full. Second, when the pool is almost empty, it is impossible to atomically scan all the entries. It can happen that, during the scan, one thread inserts an element to a bucket we already traversed, and another thread removes an element from a non-traversed bucket. In this case, the `remove` operation could falsely conclude that the pool is empty, which was never the case.

To address the second issue, we use the timestamps associated with each bucket. Recall that each timestamp is incremented when the respective bucket is modified. Therefore, if the sum of the timestamps remains constant, then no operation was executed on the pool. We can use this fact as follows. If we scan the bucket array twice, and see that the timestamp sum did not change, we can conclude that there have been no updates to the pool. This is crucial for the `remove` operation, which will use this information to know when to terminate.

The `remove` operation starts by picking a bucket based on the current thread ID, and then starting a tail-recursive `scan` method. The `scan` method traverses the array, searching for non-empty buckets. When an empty bucket is observed, its timestamp is added to the `sum` local variable. When a non-empty bucket is found, the standard CAS pattern attempts to remove an element from the bucket in the `retry` method. If successful, the element is immediately removed from the `remove` operation. Otherwise, if upon traversing the array the previous timestamp sum is equal to the current sum, the `scan` method terminates. This is shown in the following snippet:

```
def remove(): Option[T] = {
  val start =
    (Thread.currentThread.getId % buckets.length).toInt
  @tailrec def scan(witness: Long): Option[T] = {
    var i = (start + 1) % buckets.length
    var sum = 0L
    while (i != start) {
      val bucket = buckets(i)

      @tailrec def retry(): Option[T] = {
        bucket.get match {
          case (Nil, stamp) =>
            sum += stamp
            None
          case v @ (lst, stamp) =>
```

```
                val nv = (lst.tail, stamp + 1)
                if (bucket.compareAndSet(v, nv)) Some(lst.head)
                else retry()
              }
            }
            retry() match {
              case Some(v) => return Some(v)
              case None =>
            }

            i = (i + 1) % buckets.length
          }
          if (sum == witness) None
          else scan(sum)
        }
        scan(-1L)
      }
    }
```

We test the concurrent pool as follows. First, we instantiate a concurrent hash map that will track the elements we removed. Then, we create a concurrent pool, and set the number of threads `p` and the number of elements `num`:

```
    val check = new ConcurrentHashMap[Int, Unit]()
    val pool = new Pool[Int]
    val p = 8
    val num = 1000000
```

We first start `p` inserter threads, which insert non-overlapping ranges of integers into the pool. We then wait for the threads to complete:

```
    val inserters = for (i <- 0 until p) yield ch2.thread {
      for (j <- 0 until num) pool.add(i * num + j)
    }
    inserters.foreach(_.join())
```

We similarly start `p` remover threads, which remove the elements from the pool, and store the removed elements to the `check` hash map we created earlier. Each thread removes `num` elements, so the pool should never be empty until all the threads complete:

```
    val removers = for (i <- 0 until p) yield ch2.thread {
      for (j <- 0 until num) {
        pool.remove() match {
          case Some(v) => check.put(v, ())
          case None => sys.error("Should be non-empty.")
        }
      }
```

```
}
removers.foreach(_.join())
```

At the end, we sequentially traverse the elements we expect to see in the `check` hash map, and assert that they are contained, as shown in the following snippet:

```
for (i <- 0 until (num * p)) assert(check.containsKey(i))
```

And this is it! We have verified that our concurrent pool implementation works correctly. Although we will not prove this, we loosely claim that the `add` operation runs in the expected *O(1)* time, the `remove` operation runs in the expected *O(1)* time when the pool has enough elements, and in the expected *O(P)* time when the queue is nearly empty. As an exercise, you can try to improve the `remove` operation, so that it always runs in the expected *O(1)* time.

Creating and handling processes

So far, we focused on concurrency within a Scala program running in a single JVM process. Whenever we wanted to allow multiple computations to proceed concurrently, we created new threads or sent `Runnable` objects to `Executor` threads. Another route to concurrency is to create separate processes. As explained in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, separate processes have separate memory spaces and cannot share the memory directly.

There are several reasons why we occasionally want to do this. First, while JVM has a very rich ecosystem with thousands of software libraries for all kinds of tasks, sometimes the only available implementation of a certain software component is a command-line utility or prepackaged program. Running it in a new process could be the only way to harvest its functionality. Second, sometimes we want to put Scala or Java code that we do not trust in a sandbox. A third-party plugin might have to run with a reduced set of permissions. Third, sometimes we just don't want to run in the same JVM process for performance reasons. Garbage collection or JIT compilation in a separate process should not affect the execution of our process, given that the machine has sufficient CPUs.

The `scala.sys.process` package contains a concise API for dealing with other processes. We can run the child process synchronously, in which case the thread from the parent process that runs it waits until the child process terminates, or asynchronously, in which case, the child process runs concurrently with the calling thread from the parent process. We will first show you how to run a new process synchronously:

```
import scala.sys.process._
object ProcessRun extends App {
  val command = "ls"
  val exitcode = command.!
  log(s"command exited with status $exitcode")
}
```

Importing the contents of the `scala.sys.process` package allows us to call the `!` method on any string. The shell command represented by the string is then run from the working directory of the current process. The return value is the exit code of the new process–zero when the process exits successfully and a nonzero error code otherwise.

Sometimes, we are interested in the standard output of a process rather than its exit code. In this case, we start the process with the`!!` method. Let's assume that we want a `lineCount` method for text files in `FileSystem`, but are too lazy to implement it from scratch:

```
def lineCount(filename: String): Int = {
  val output = s"wc $filename".!!
  output.trim.split(" ").head.toInt
}
```

After removing the white space from the output with the `trim` method on `String` type and converting the first part of the output to an integer, we obtain the word count of a file.

To start the process asynchronously, we call the `run` method on a string that represents the command. This method returns a `Process` object with the `exitValue` method, which is blocked until the process terminates, and the `destroy` method, which stops the process immediately. Assume that we have a potentially long-running process that lists all the files in our filesystem. After one second, we might wish to stop it by calling the `destroy` method on the `Process` object:

```
object ProcessAsync extends App {
  val lsProcess = "ls -R /".run()
  Thread.sleep(1000)
  log("Timeout - killing ls!")
  lsProcess.destroy()
}
```

Overloads of the `run` method allow you to communicate with the process by hooking the custom input and output streams or providing a custom `logger` object that is called each time the new process outputs a line.

The `scala.sys.process` API has additional features such as starting multiple processes and piping their outputs together, running a different process if the current process fails, or redirecting the output to a file. It strives to mimic much of the functionality provided by the Unix shells. For complete information, we refer the reader to the Scala standard library's documentation of the `scala.sys.process` package.

Summary

This chapter presented the traditional building blocks of concurrent programs in Scala. We saw how to use `Executor` objects to run concurrent computations. We learned how to use atomic primitives to atomically switch between different states in the program and implement locks and lock-free algorithms. We studied the implementation of lazy values and their impact on concurrent programs. We then showed you important classes of concurrent collections and learned how to apply them in practice, and we concluded by visiting the `scala.sys.process` package. These insights are not only specific to Scala; but most languages and platforms also have concurrency utilities that are similar to the ones presented in this chapter.

Many other Java concurrency APIs are thoroughly explained in the book *Java Concurrency in Practice*, by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. To learn more about concepts such as lock-freedom, atomic variables, various types of locks, or concurrent data structures, we recommend the book *The Art of Multiprocessor Programming* by Maurice Herlihy and Nir Shavit, Morgan Kaufmann.

Although the concurrency building blocks in this chapter are more high level than the basic concurrency primitives of `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, there are still culprits lurking at every corner. We had to be careful not to block when running on the execution context, to steer clear from the ABA problem, avoid synchronizing on objects that use lazy values, and ensure that concurrent collections are not modified while using their iterators. All this imposes quite a burden on the programmer. Couldn't concurrent programming be simpler? Fortunately, the answer is yes, as Scala supports styles of expressing concurrency that are more high level and declarative; less prone to effects such as deadlocks, starvation, or non-determinism; and generally easier to reason about. In the following chapters, we will dive into Scala-specific concurrency APIs that are safer and more intuitive to use. We will start by studying futures and promises in the next chapter, which allow you to compose asynchronous computations in a thread-safe and intuitive way.

Exercises

The following exercises cover the various topics from this chapter. Most of the exercises require implementing new concurrent data structures using atomic variables and the CAS instruction. These data structures can also be solved using the `synchronized` statement, so it is helpful to contrast the advantages of the two approaches:

1. Implement a custom `ExecutionContext` class called `PiggybackContext`, which executes `Runnable` objects on the same thread that calls the `execute` method. Ensure that a `Runnable` object executing on the `PiggybackContext` can also call the `execute` method and that exceptions are properly reported.
2. Implement a `TreiberStack` class, which implements a concurrent stack abstraction:

   ```
   class TreiberStack[T] {
     def push(x: T): Unit = ???
     def pop(): T = ???
   }
   ```

   Use an atomic reference variable that points to a linked list of nodes that were previously pushed to the stack. Make sure that your implementation is lock-free and not susceptible to the ABA problem.

3. Implement a `ConcurrentSortedList` class, which implements a concurrent sorted list abstraction:

   ```
   class ConcurrentSortedList[T](implicit val ord: Ordering[T]) {
      def add(x: T): Unit = ???
      def iterator: Iterator[T] = ???
   }
   ```

   Under the hood, the `ConcurrentSortedList` class should use a linked list of atomic references. Ensure that your implementation is lock-free and avoids ABA problems.

   The `Iterator` object returned by the `iterator` method must correctly traverse the elements of the list in ascending order under the assumption that there are no concurrent invocations of the `add` method.

4. If required, modify the `ConcurrentSortedList` class from the previous example so that calling the `add` method has the running time linear to the length of the list and creates a constant number of new objects when there are no retries due to concurrent `add` invocations.

5. Implement a `LazyCell` class with the following interface:

   ```
   class LazyCell[T](initialization: =>T) {
     def apply(): T = ???
   }
   ```

   Creating a `LazyCell` object and calling the `apply` method must have the same semantics as declaring a lazy value and reading it, respectively.

   You are not allowed to use lazy values in your implementation.

6. Implement a `PureLazyCell` class with the same interface and semantics as the `LazyCell` class from the previous exercise. The `PureLazyCell` class assumes that the initialization parameter does not cause side effects, so it can be evaluated more than once.

   The `apply` method must be lock-free and should call the initialization as little as possible.

7. Implement a `SyncConcurrentMap` class that extends the `Map` interface from the `scala.collection.concurrent` package. Use the `synchronized` statement to protect the state of the concurrent map.
8. Implement a method `spawn` that, given a block of Scala code, starts a new JVM process and runs the specified block in the new process:

   ```
   def spawn[T](block: =>T): T = ???
   ```

   Once the block returns a value, the `spawn` method should return the value from the child process. If the block throws an exception, the `spawn` method should throw the same exception.

TIP

Use Java serialization to transfer the block of code, its return value, and the potential exceptions between the parent and the child JVM processes.

9. Augment the lock-free pool implementation from this chapter with a `foreach` operation, used to traverse all the elements in the pool. Then make another version of `foreach` that is both lock-free and linearizable.
10. Prove that the lock-free pool implementation from this chapter is correct.
11. Currently, the `remove` operation of the lock-free pool implementation from this chapter runs in *O(P)* worst-case time, where *P* is the number of processors on the machine. Improve the lock-free pool implementation so that the operations run in *O(1)* expected time, both in terms of the number of stored elements and the number of processors.

4
Asynchronous Programming with Futures and Promises

Programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible.

– John Carmack

In the examples of the previous chapters, we often dealt with blocking computations. We have seen that blocking synchronization can have negative effects: it can cause deadlocks, starve thread pools, or break lazy value initialization. While, in some cases, blocking is the right tool for the job, in many cases we can avoid it. Asynchronous programming refers to the programming style in which executions occur independently of the main program flow. Asynchronous programming helps you to eliminate blocking instead of suspending the thread whenever a resource is not available; a separate computation is scheduled to proceed once the resource becomes available.

In a way, many of the concurrency patterns seen so far support asynchronous programming; thread creation and scheduling execution context tasks can be used to start executing a computation concurrent to the main program flow. Still, using these facilities directly when avoiding blocking or composing asynchronous computations is not straightforward. In this chapter, we will focus on two abstractions in Scala that are specifically tailored for this task: futures and promises.

More specifically, we will study the following topics:

- Starting asynchronous computations, and using `Future` objects
- Installing callbacks that handle the results of asynchronous computations
- Exception semantics of `Future` objects, and using the type `Try`
- Functional composition of `Future` objects
- Using `Promise` objects to interface with callback-based APIs, implement future combinators, and support cancellation
- Blocking threads inside asynchronous computations
- Using the Scala `Async` library

In the next section, we will start by introducing the try `Future` , and show why it is useful.

Futures

In earlier chapters, we learned that parallel executions in a concurrent program proceed on entities called **threads**. At any point, the execution of a thread can be temporarily suspended, until a specific condition is fulfilled. When this happens, we say that the thread is blocked. Why do we block threads in the first place in concurrent programming? One of the reasons is that we have a finite amount of resources; multiple computations that share these resources sometimes need to wait. In other situations, a computation needs specific data to proceed, and if that data is not yet available, threads responsible for producing the data could be slow or the source of the data could be external to the program. A classic example is waiting for the data to arrive over the network. Let's assume that we have a `getWebpage` method, that given a `url` string with the location of the webpage, returns that webpage's contents:

```
def getWebpage(url: String): String
```

The return type of the `getWebpage` method is `String`; the method must return a string with the webpage's contents. Upon sending an HTTP request, though, the webpage's contents are not available immediately. It takes some time for the request to travel over the network to the server and back before the program can access the document. The only way for the method to return the contents of the webpage as a string value is to wait for the HTTP response to arrive. However, this can take a relatively long time from the program's point of view; even with a high-speed Internet connection, the `getWebpage` method needs to wait. Since the thread that called the `getWebpage` method cannot proceed without the contents of the webpage, it needs to pause its execution; therefore, the only way to correctly implement the `getWebpage` method is to block.

We already know that blocking can have negative side-effects, so can we change the return value of the `getWebpage` method to some special value that can be returned immediately? The answer is yes. In Scala, this special value is called a **future**. A future is a placeholder, that is, a memory location for the value. This placeholder does not need to contain a value when the future is created; the value can be placed into the future eventually by the `getWebpage` method. We can change the signature of the `getWebpage` method to return a future as follows:

```
    def getWebpage(url: String): Future[String]
```

Here, the `Future[String]` type means that the future object can eventually contain a `String` value. We can now implement the `getWebpage` method without blocking-we can start the HTTP request asynchronously and place the webpage's contents into the future when they become available. When this happens, we say that the `getWebpage` method completes the future. Importantly, after the future is completed with some value, that value can no longer change.

The `Future[T]` type encodes latency in the program; use it to encode values that will become available later during execution.

This removes blocking from the `getWebpage` method, but it is not clear how the calling thread can extract the content of the future. Polling is one non-blocking way of extracting the content. In the polling approach, the calling thread calls a special method to block until the value becomes available. While this approach does not eliminate blocking, it transfers the responsibility of blocking from the `getWebpage` method to the caller thread. Java defines its own `Future` type to encode values that will become available later. However, as a Scala developer, you should use Scala's futures instead; they allow additional ways of handling future values and avoid blocking, as we will soon see.

When programming with futures in Scala, we need to distinguish between **future values** and **future computations**. A future value of the type Future[T] denotes some value of type T in the program that might not be currently available, but could become available later. Usually, when we say a future, we really mean a future value. In the scala.concurrent package, futures are represented with the Future[T] trait:

```
trait Future[T]
```

By contrast, a future computation is an asynchronous computation that produces a future value. A future computation can be started by calling the apply method on the Future companion object. This method has the following signature in the scala.concurrent package:

```
def apply[T](b: =>T)(implicit e: ExecutionContext): Future[T]
```

This method takes a by-name parameter of the type T . This is the body of the asynchronous computation that results in some value of type T. It also takes an implicit ExecutionContext parameter, which abstracts over where and when the thread gets executed, as we learned in Chapter 3, *Traditional Building Blocks of Concurrency*. Recall that Scala's implicit parameters can either be specified when calling a method, in the same way as normal parameters, or they can be left out-in this case, the Scala compiler searches for a value of the ExecutionContext type in the surrounding scope. Most Future methods take an implicit execution context. Finally, the Future.apply method returns a future of the type T. This future is completed with the value resulting from the asynchronous computation, b.

Starting future computations

Let's see how to start a future computation in an example. We first import the contents of the scala.concurrent package. We then import the global execution context from the Implicits object. This makes sure that future computations execute on global, the default execution context you can use in most cases:

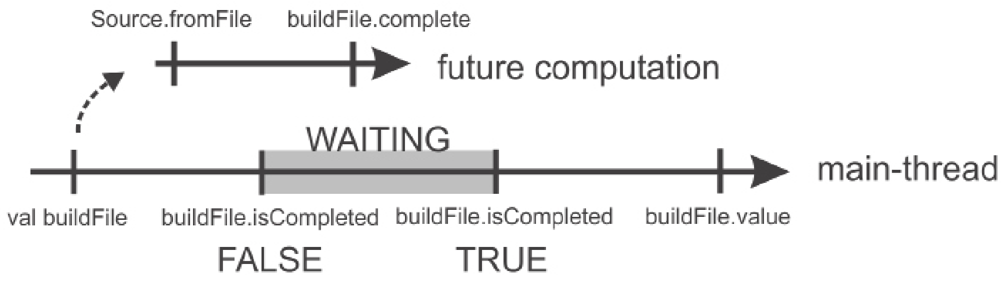
```
import scala.concurrent._
import ExecutionContext.Implicits.global
object FuturesCreate extends App {
  Future { log("the future is here") }
  log("the future is coming")
  Thread.sleep(1000)
}
```

The order in which the `log` method calls (in the future computation and the main thread) execute is nondeterministic. The `Future` singleton object followed by a block is syntactic sugar for calling the `Future.apply` method. The `Future.apply` method acts similarly to the `execute` statement from `Chapter 3`, *Traditional Building Blocks of Concurrency*. The difference is that the `Future.apply` method returns a future value. We can poll this future value until it is completed. In the following example, we can use the `scala.io.Source` object to read the contents of our `build.sbt` file in a future computation. The main thread calls the `isCompleted` method on the `buildFile` future value, returned from the future computation. Chances are that the build file was not read so rapid, so `isCompleted` returns `false`. After 250 milliseconds, the main thread calls `isCompleted` again, and this time `isCompleted` returns `true`. Finally, the main thread calls the `value` method, which returns the contents of the build file:

```
import scala.io.Source
object FuturesDataType extends App {
  val buildFile: Future[String] = Future {
    val f = Source.fromFile("build.sbt")
    try f.getLines.mkString("\n") finally f.close()
  }
  log(s"started reading the build file asynchronously")
  log(s"status: ${buildFile.isCompleted}")
  Thread.sleep(250)
  log(s"status: ${buildFile.isCompleted}")
  log(s"value: ${buildFile.value}")
}
```

In this example, we used polling to obtain the value of the future. The `Future` singleton object's polling methods are non-blocking, but they are also nondeterministic; `isCompleted` will repeatedly return `false` until the future is completed. Importantly, completion of the future is in a happens-before relationship with the polling calls. If the future completes before the invocation of the polling method, then its effects are visible to the thread after polling completes.

Shown graphically, polling looks like the following figure:



Polling diagram

Polling is like calling your potential employer every five minutes to ask if you're hired. What you really want to do is hand in a job application and then apply for other jobs, instead of busy-waiting for the employer's response. Once your employer decides to hire you, they will give you a call on the phone number you left them. We want futures to do the same; when they are completed, they should call a specific function we left for them. This is the topic of the next section.

Future callbacks

A callback is a function that is called once its arguments become available. When a Scala future takes a callback, it eventually calls that callback. However, the future does not call the callback before this future is completed with some value.

Let's assume that we need to look up details of the URL specification from the W3 consortium. We are interested in all the occurrences of the `telnet` keyword. The URL specification is available as a text document at `https://www.w3.org/`. We can use the `scala.io.Source` object to fetch the contents of the specification, and use futures in the `getUrlSpec` method to asynchronously execute the HTTP request. The `getUrlSpec` method first calls the `fromURL` method to obtain a `Source` object with the text document. It then calls `getLines` to get a list of separate lines in the document:

```
object FuturesCallbacks extends App {
  def getUrlSpec(): Future[List[String]] = Future {
    val url = "http://www.w3.org/Addressing/URL/url-spec.txt"
    val f = Source.fromURL(url)
    try f.getLines.toList finally f.close()
  }
  val urlSpec: Future[List[String]] = getUrlSpec()
```

To find the lines in the `urlSpec` future that contains the `telnet` keyword, we use the `find` method which takes a list of lines and a keyword and returns a string containing the matches:

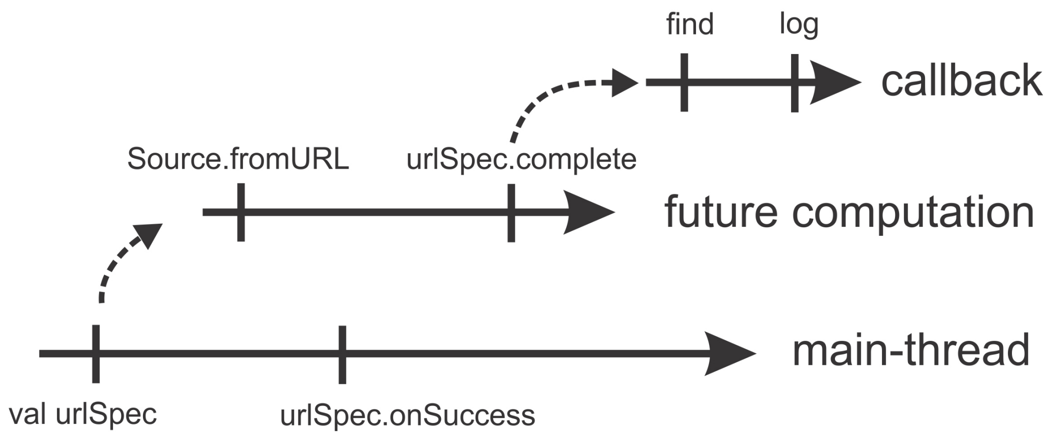
```
    def find(lines: List[String], keyword: String): String =
  lines.zipWithIndex collect {
        case (line, n) if line.contains(keyword) => (n, line)
      } mkString("\n")
```

The `find` method takes a `List[String]` parameter, but `urlSpec` is of the `Future[List[String]]` type. We cannot pass the `urlSpec` future directly to the `find` method; and for a good reason, the value might not be available at the time when we call the `find` method.

Instead, we install a callback to the future using the `foreach` method. Note that the equivalent of the `foreach` method is the `onSuccess` method, but it might be deprecated after Scala 2.11. This method takes a partial function that, given a value of the future, performs some action, as follows:

```
    urlSpec foreach {
      case lines => log(find(lines, "telnet"))
    }
    log("callback registered, continuing with other work")
    Thread.sleep(2000)
```

Importantly, installing a callback is a non-blocking operation. The `log` statement in the main thread immediately executes after the callback is registered, but the `log` statement in the callback can be called much later. This is illustrated in the following figure:



Callback diagram

Note that the callback is not necessarily invoked immediately after the future is completed. Most execution contexts schedule a task to asynchronously process the callbacks. The same is true if the future is already completed when we try to install a callback.

After the future is completed, the callback is called *eventually* and independently from other callbacks on the same future. The specified execution context decides when and on which thread the callback gets executed.
There is a happens-before relationship between completing the future and starting the callback.

We are not limited to installing a single callback to the future. If we additionally want to find all the occurrences of the `password` keyword, we can install another callback:

```
    urlSpec foreach {
      case lines => log(find(lines, "password"))
    }
    Thread.sleep(1000)
  }
```

As an experienced Scala programmer, you might have heard about referential transparency. Roughly speaking, a function is referentially transparent if it does not execute any side effects such as variable assignment, modifying mutable collections, or writing to the standard output. Callbacks on futures have one very useful property. Programs using only the `Future.apply` and `foreach` calls with referentially transparent callbacks are deterministic. For the same inputs, such programs will always compute the same results.

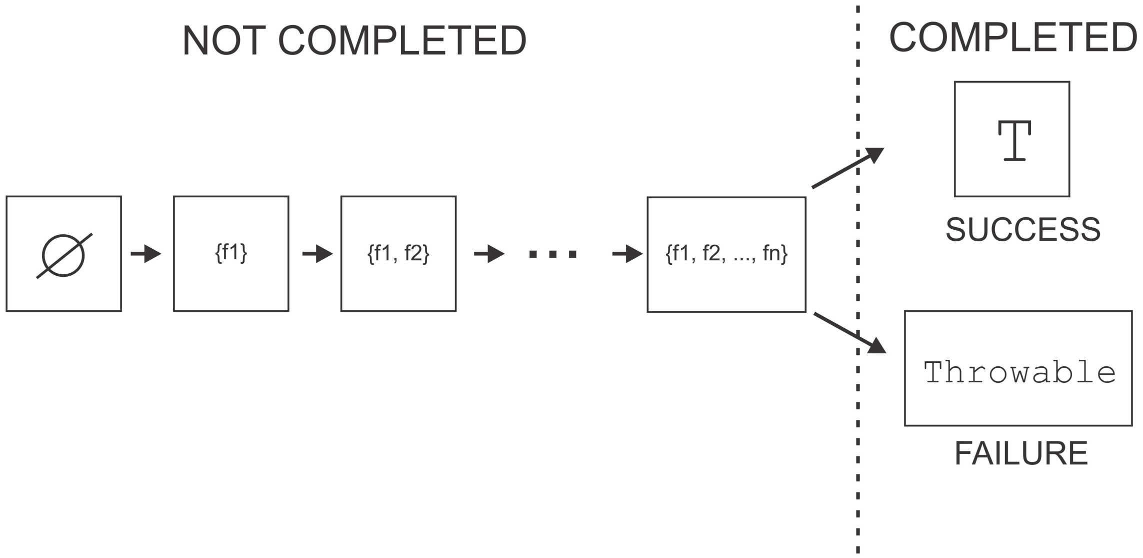
Programs composed from referentially transparent future computations and callbacks are deterministic.

In the examples so far, we assumed that an asynchronous computation yielding a future always succeeds. However, computations occasionally fail and throw exceptions. We will study how to handle failures in asynchronous computations next.

Futures and exceptions

If a future computation throws an exception, then its corresponding future object cannot be completed with a value. Ideally, we would like to be notified when this happens. If you apply for a job and the employer decides to hire someone else, you would still like to receive a phone call. Otherwise, you might spend days sitting idly in front of your phone, waiting for the call from the recruiter.

When a Scala future is completed, it can either be completed *successfully* or with a failure. When a future is completed with a failure, we also say that a future has *failed*. To summarize all the different states of a future, we show the following state diagram. A future is created without any associated callbacks. Then, any number of callbacks `f1`, `f2`, ..., `fn` can be assigned to it. When the future is completed, it has either completed successfully or has failed. After that, the future's state no longer changes, and registering a callback immediately schedules it for execution.



We now take a closer look at handling the failure case. The `foreach` method only accepts callbacks that handle values from a successfully completed future, so we need another method to install failure callbacks. This method is called `failed`. It returns a `Future[Throwable]` object that contains the exception that the current object has failed with, and can be used with the `foreach` statement to access the exception:

```
object FuturesFailure extends App {
  val urlSpec: Future[String] = Future {
    val invalidUrl = "http://www.w3.org/non-existent-url-spec.txt"
    Source.fromURL(invalidUrl).mkString
  }
  urlSpec.failed foreach {
    case t => log(s"exception occurred - $t")
  }
  Thread.sleep(1000)
}
```

In this example, our asynchronous computation sends an HTTP request to an invalid URL. As a result, the `fromURL` method throws an exception, and the `urlSpec` future fails. The program then prints the exception name and message with the `log` statement.

Using the Try type

For conciseness, sometimes we want to subscribe to both successes and failures in the same callback. To do this, we need to use the type `Try[T]`. The type `Try[T]` is very similar to the type `Option[T]` . Recall from your experience with sequential Scala programming that the type `Option[T]` is used to encode a value of the type `T` or its absence. A value of `Option[T]` type can either be an object of a type `Some[T]` , which holds some value, or `None`, which does not hold anything. We use pattern matching to determine whether an `Option[T]` type is `Some[T]` or `None`. Optional types are an alternative to using `null` values, which is what one typically does in Java. However, the `Option[T]` type does not allow encoding failures in its `None` subtype. The `None` subtype tells us nothing about the exception in the computation. For this, we use the `Try[T]` type.

The type Try[T] has two implementations–the type Success[T], which encodes the results of the successful computations, and the Failure[T] type, which encodes the Throwable objects that failed the computation. We use pattern matching to determine which of the two a Try[T] object is:

```
def handleMessage(t: Try[String]) = t match {
  case Success(msg) => log(msg)
  case Failure(error) => log(s"unexpected failure - $error")
}
```

The Try[T] objects are immutable objects used synchronously; unlike futures, they contain a value or an exception from the moment they are created. They are more akin to collections than to futures. We can even compose Try[T] values in for-comprehensions. In the following code snippet, we will compose the name of the current thread with some custom text:

```
import scala.util.{Try, Success, Failure}
object FuturesTry extends App {
  val threadName: Try[String] = Try(Thread.currentThread.getName)
  val someText: Try[String] = Try("Try objects are synchronous")
  val message: Try[String] = for {
    tn <- threadName
    st <- someText
  } yield s"Message $st was created on t = $tn"
  handleMessage(message)
}
```

We will first create two Try[String] values, threadName and someText, using the Try.apply factory method. The for comprehension extracts the thread name, tn, from the threadName value, and then the st text from the someText value. These values are then used to yield another string. If any of the Try values in the for comprehension fail, then the resulting Try value fails with the Throwable object from the first failed Try value. However, if all the Try values are Success, then the resulting Try value is Success with the value of the expression after the yield keyword. If this expression throws an exception, the resulting Try value fails with that exception.

Note that the preceding example always prints the name of the main thread. Creating Try objects and using them in for comprehensions always occurs on the caller thread.

Unlike `Future[T]` values, `Try[T]` values are manipulated synchronously.

In most cases, we use the `Try` values in pattern matching. When calling the `onComplete` callback, we will provide a partial function that matches the `Success` and `Failure` values. Our example for fetching the URL specification is as follows:

```
urlSpec onComplete {
  case Success(txt) => log(find(txt))
  case Failure(err) => log(s"exception occurred - $err")
}
```

Fatal exceptions

We have seen futures storing exceptions that caused them to fail. However, there are some `Throwable` objects that a future computation does not catch. In the following short program, the callback on the `f` future is never invoked. Instead, the stack trace of `InterruptedException` exception is printed on the standard error output:

```
object FuturesNonFatal extends App {
  val f = Future { throw new InterruptedException }
  val g = Future { throw new IllegalArgumentException }
  f.failed foreach { case t => log(s"error - $t") }
  g.failed foreach { case t => log(s"error - $t") }
}
```

The `InterruptedException` exception and some severe program errors such as `LinkageError`, `VirtualMachineError`, `ThreadDeath`, and Scala's `ControlThrowable` error are forwarded to the execution context's `reportFailure` method introduced in `Chapter 3`, *Traditional Building Blocks of Concurrency*. These types of `Throwable` object are called **fatal errors**. To find out if a `Throwable` object will be stored in a `Future` instance, you can pattern match the `Throwable` object with the `NonFatal` extractor:

```
f.failed foreach {
  case NonFatal(t) => log(s"$t is non-fatal!")
}
```

Note that you never need to manually match in order to see whether errors in your futures are nonfatal. Fatal errors are automatically forwarded to the execution context.

Future computations do not catch fatal errors. Use the `NonFatal` extractor to pattern match against nonfatal errors.

Functional composition on futures

Callbacks are useful, but they can make reasoning about control flow difficult when programs become larger. They also disallow certain patterns in asynchronous programming in particular, it is cumbersome to use a callback to subscribe to multiple futures at once. Luckily, Scala futures have an answer to these problems called **functional composition**. Functional composition on futures allows using futures inside `for` comprehensions, and is often more intuitive to use than callbacks.

Introducing futures transfers the responsibility for blocking from the API to the caller. The `foreach` method helps you to avoid blocking altogether. It also eliminates non-determinism inherent to polling methods such as `isCompleted` and `value`. Still, there are some situations when the `foreach` statement is not the best solution.

Let's say that we want to implement some of the functionality from the Git version control system; we want to use the `.gitignore` file to find files in our project tree that should not be versioned. We simplify our task by assuming that the `.gitignore` file only contains a list of prefixes for blacklisted file paths, and no regular expressions.

We perform two asynchronous actions. First, we fetch the contents of our `.gitignore` file in a future computation. Then, using its contents, we will asynchronously scan all the files in our project directory and match them. We will start by importing the packages necessary for file handling. In addition to the `scala.io.Source` object, we use the `java.io` package and the `apache.commons.io.FileUtils` class, and import them as follows:

```
import java.io._
import org.apache.commons.io.FileUtils._
import scala.collection.convert.decorateAsScala._
```

If you haven't already added the dependency on Commons IO to your `build.sbt` file in the previous chapters, now is a good time to introduce the following line:

```
libraryDependencies += "commons-io" % "commons-io" % "2.4"
```

We will first create a future using the `blacklistFile` method, which reads the contents of the `.gitignore` file. Given the pace at which technology is evolving these days, we never know when a different version control system will become more popular; so we add the `name` parameter for the name of the blacklist file. We filter out the empty lines and all the comment lines starting with a `#` sign. We then convert them to a list, as shown in the following code snippet:

```
object FuturesClumsyCallback extends App {
  def blacklistFile(name: String): Future[List[String]] = Future {
    val lines = Source.fromFile(name).getLines
    lines.filter(x => !x.startsWith("#") && !x.isEmpty).toList
  }
```

In our case, the future returned by the `blacklistFile` method eventually contains a list with a single string, the `target` directory is where SBT stores files created by the Scala compiler. Then, we implement another method named `findFiles` that, given a list of patterns, finds all the files in the current directory containing these patterns. The `iterateFiles` method from the Commons IO library returns a Java iterator over the project files, so we can convert it to a Scala iterator by calling `asScala`. We then yield all the matching file paths:

```
  def findFiles(patterns: List[String]): List[String] = {
    val root = new File(".")
    for {
      f <- iterateFiles(root, null, true).asScala.toList
      pat <- patterns
      abspat = root.getCanonicalPath + File.separator + pat
      if f.getCanonicalPath.contains(abspat)
    } yield f.getCanonicalPath
  }
```

If we now want to list blacklisted files, we first need to call `foreach` on the `blacklistFile` future, and call `findPatterns` from inside the callback, as follows:

```
  blacklistFile(".gitignore") foreach {
    case lines =>
      val files = findFiles(lines)
      log(s"matches: ${files.mkString("\n")}")
  }
  Thread.sleep(1000)
}
```

Assume your fellow developer now asks you to implement another `blacklisted` method that takes the name of the blacklist file and returns a future with a list of blacklisted files. This allows us to specify the callback independently in the program; instead of printing the files to the standard output, another part of the program can, for example, create a safety backup of the blacklisted files using the following method:

```
def blacklisted(name: String): Future[List[String]]
```

Being an experienced object-oriented developer, you'd like to reuse the `blacklistFile` future and the `findFiles` method. After all, the functionality is already there. We challenge you to reuse the existing methods to implement the new `blacklisted` method. Try to use the `foreach` statement. You will find this task extremely difficult.

So far, we haven't seen methods that produce new futures using the values in existing futures. The `Future` trait has a `map` method that maps the value in one future to a value in another future:

```
def map[S](f: T => S)(implicit e: ExecutionContext): Future[S]
```

This method is non blocking–it returns the `Future[S]` object immediately. After the original future completes with some value `x`, the returned `Future[S]` object is eventually completed with `f(x)`. With the `map` method, our task is trivial: we transform the patterns into a list of matching files by calling the `findFiles` method:

```
def blacklisted(name: String): Future[List[String]] =
  blacklistFile(name).map(patterns => findFiles(patterns))
```

As a Scala developer, you know that a `map` operation on a collection transforms many elements into a new collection. To more easily comprehend operations such as the `map` operation on futures, you can consider a future as a specific form of collection that contains at most one element.

Functional composition is a programming pattern in which simpler values are composed into more complex ones by means of higher-order functions called **combinators**. Functional composition on Scala collections should be familiar to you from sequential Scala programming. For example, the `map` method on a collection produces a new collection containing elements from the original collection, mapped with a specified function.

Functional composition on futures is similar; we can produce new futures by transforming or merging existing futures, as in the preceding example. Callbacks are useful, but they do not directly allow functional composition in the way combinators such as `map` do. Just as with callbacks, a function passed to a combinator is never invoked before the corresponding future completes.

There is a happens-before relationship between completing the future and invoking the function in any of its combinators.

Choosing between alternative ways to handle futures can be confusing. When should we use functional composition in place of callbacks? A good rule of thumb is to use callbacks for side-effecting actions that depend on a single future. In all other situations, we can use functional composition.

When an action in the program depends on the value of a single future, use callbacks on futures. When subsequent actions in the program depend on values of multiple futures or produce new futures, use functional composition on futures.

Let us consider several crucial combinators for functional composition. The `map` method on a `Future[T]` takes an `f` function and returns a new `Future[S]` future. After the `Future[T]` is completed, the `Future[S]` is completed by applying `f` to the value in `Future[T]`. If `Future[T]` fails with an exception `e`, or the mapping function `f` throws an exception `e`, then `Future[S]` also fails with that exception `e`.

Recall that Scala allows using for-comprehensions on objects that have a `map` method, so we can use futures in for-comprehensions. Let's assume that we want to get the future with the longest line from our `build.sbt` file. The computation proceeds in two steps. First, we read in the lines from the disk, and then we call the `maxBy` method to get the longest line:

```
val buildFile = Future {
  Source.fromFile("build.sbt").getLines
}

val longest = for (ls <- buildFile) yield ls.maxBy(_.length)
longest foreach {
  case line => log(s"longest line: $line")
}
```

The `longest` declaration is desugared by the Scala compiler into the following line:

```
val longest = buildFile.map(ls => ls.maxBy(_.length))
```

The real advantage of `for` comprehensions becomes apparent when we use the `flatMap` combinator, which has the following signature:

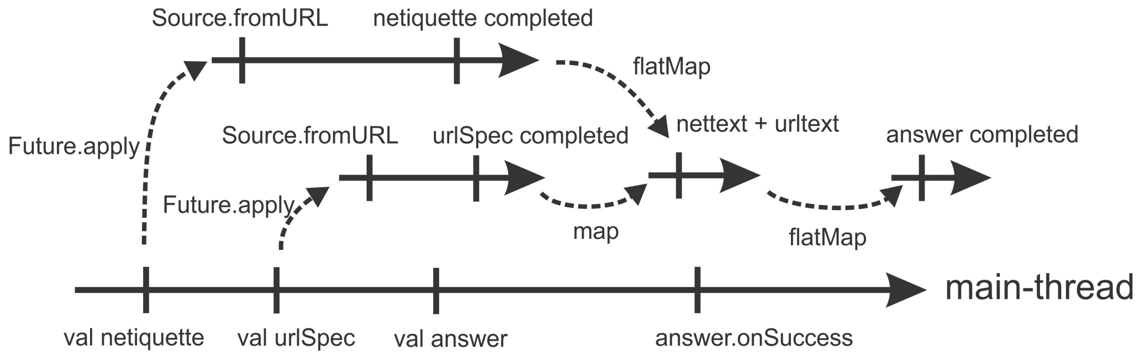
```
def flatMap[S](f: T => Future[S])(implicit e: ExecutionContext):
  Future[S]
```

The `flatMap` combinator uses the current future with the `Future[T]` type to produce another future with the type `Future[S]` . The resulting `Future[S]` is completed by taking the value `x` of the type `T` from the current future, and mapping that value to another future `f(x)`. While the future resulting from a `map` method completes when the mapping function `f` completes, the future resulting from a `flatMap` method completes when both `f` and the future returned by `f` complete.

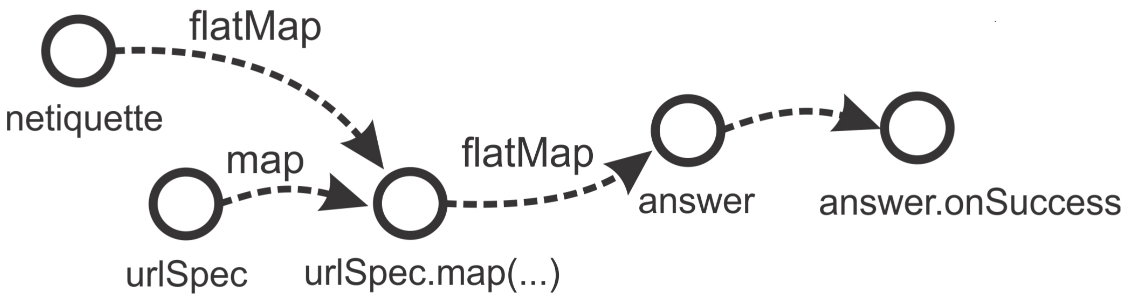
To understand how this combinator is useful, let's consider the following example. Assume that your job application went well and you got that new job you were hoping for. On the first day of work, you receive a chain e-mail from your secretary. The chain e-mail claims that you should never open URLs starting with `ftp://`, because all of them contain viruses. As a skillful techie with a lot of experience, you quickly recognize the chain letter for what it is–a scam. You, therefore, decide to enlighten your secretary by sending her instructions on how to communicate using e-mails, and an explanation of what FTP links are. You write a short program that replies asynchronously. You've got better things to do than to spend your day writing e-mails:

```
val netiquetteUrl = "http://www.ietf.org/rfc/rfc1855.txt"
val netiquette = Future { Source.fromURL(netiquetteUrl).mkString }
val urlSpecUrl = "http://www.w3.org/Addressing/URL/url-spec.txt"
val urlSpec = Future { Source.fromURL(urlSpecUrl).mkString }
val answer = netiquette.flatMap { nettext =>
  urlSpec.map { urltext =>
    "Check this out: " + nettext + ". And check out: " + urltext
  }
}
answer foreach { case contents => log(contents) }
```

This program asynchronously fetches the good old **RFC 1855**–the guidelines for e-mail communication or netiquette. It then asynchronously fetches the URL specification with information on the `ftp` schema. The program attempts to concatenate the two texts. It calls `flatMap` on the `netiquette` future. Based on the `nettext` value in the `netiquette` future, the `flatMap` future needs to return another future. It could return the `urlSpec` future directly, but the resulting future, `answer`, would then be completed with just the URL specification. Instead, we can call the `map` combinator on the `urlSpec` future; we map its value, `urltext`, into the concatenation of the `nettext` and `urltext` values. This results in another intermediate future holding the concatenation; once this future is completed, the `answer` future is completed as well. Graphically, this looks as follows:



If you look at this execution diagram from a distance, you will notice that there is an inherent ordering between asynchronous computations. We can capture these relationships in a graph, as shown in the following figure:



This graph is called the **dataflow graph**, because it describes how the data flows from one future to another. Futures are represented with vertices and asynchronous computations are directed edges between them. An edge points from one vertex to another if the value of future in the first vertex is used to compute the value of future in the second vertex. In this graph, futures produced by `Future.apply` are source vertices-they have only outward edges. Various future combinators such as `map` and `flatMap` connect different vertices. Callback functions such as `foreach` lead to sink vertices-they have no outward edges. Some combinators, such as `flatMap`, can use values from multiple vertices.

The `flatMap` combinator combines two futures into one: the one on which the `flatMap` combinator is invoked and the one that is returned by the argument function.

There are two issues with our e-mail example. First, we should be nicer to our new secretary; she's not a techie like we are. Second, using `flatMap` directly makes the program hard to understand. There are not many developers in the Scala community that use `flatMap` like this. Instead, `flatMap` should be used implicitly in `for` comprehensions:

```
val answer = for {
  nettext <- netiquette
  urltext <- urlSpec
} yield {
  "First, read this: " + nettext + ". Now, try this: " + urltext
}
```

After desugaring, this `for` comprehension is identical to what we had before. This is much simpler; the program now almost reads itself. For the `nettext` value of the `netiquette` future and the `urltext` value of the `urlSpec` future, the `answer` future is a new future with the concatenation of `nettext` and `urltext`.

You should prefer for-comprehensions to using `flatMap` directly to make programs more concise and understandable.

Note that the following for-comprehension looks very similar to what we had before, but it is not equivalent:

```
val answer = for {
  nettext <- Future { Source.fromURL(netiquetteUrl).mkString }
  urltext <- Future { Source.fromURL(urlSpecUrl).mkString }
} yield {
  "First, read this: " + nettext + ". Now, try this: " + urltext
}
```

In the preceding code, the `nettext` value is extracted from the first future. Only after the first future is completed, the second future computation start. This is useful when the second asynchronous computation uses `nettext`, but in our case fetching the `netiquette` document and the URL specification can proceed concurrently.

So far, we have only considered future combinators that work with successful futures. When any of the input futures fail or the computation in the combinator throws an exception, the resulting future fails with the same exception. In some situations, we want to handle the exception in the future in the same way as we handle exceptions with a `try-catch` block in sequential programming. A combinator that is helpful in these situations is called `recover`. Its simplified signature is as follows:

```
def recover(pf: PartialFunction[Throwable, T])
  (implicit e: ExecutionContext): Future[T]
```

When this combinator is called on a future, which is successfully completed with some value `x` of the type `T`, the resulting future is completed with the same value `x`. On the other hand, if a future fails, then the `pf` partial function is applied to the `Throwable` object that failed it. If the `pf` partial function is not defined for the `Throwable` object, then the resulting future is failed with the same `Throwable` object. Otherwise, the resulting future is completed with the result of applying `pf` to the `Throwable` object. If the `pf` partial function itself throws an exception, the resulting future is completed with that exception.

Let's assume you're worried about misspelling the URL for the `netiquette` document. You can use the `recover` combinator on the `netiquette` future to provide a reasonable default message if anything fails, as follows:

```
val netiquetteUrl = "http://www.ietf.org/rfc/rfc1855.doc"
val netiquette = Future { Source.fromURL(netiquetteUrl).mkString }
val answer = netiquette recover {
  case e: java.io.FileNotFoundException =>
    "Dear secretary, thank you for your e-mail." +
    "You might be interested to know that ftp links " +
    "can also point to regular files we keep on our servers."
}
answer foreach { case contents => log(contents) }
Thread.sleep(2000)
```

Futures come with other combinators such as `filter`, `fallbackTo`, or `zip`, but we will not cover all of them here; an understanding of the basic combinators should be sufficient. You might wish to study the remaining combinators in the API documentation.

Promises

In `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we implemented an `asynchronous` method that used a worker thread and a task queue to receive and execute asynchronous computations. That example should have left you with a basic intuition about how the `execute` method is implemented in execution contexts. You might be wondering how the `Future.apply` method can return and complete a `Future` object. We will study promises in this section to answer this question. **Promises** are objects that can be assigned a value or an exception only once. This is why promises are sometimes also called single-assignment variables. A promise is represented with the `Promise[T]` type in Scala. To create a promise instance, we use the `Promise.apply` method on the `Promise` companion object:

```
def apply[T](): Promise[T]
```

This method returns a new promise instance. Like the `Future.apply` method, the `Promise.apply` method returns immediately; it is non-blocking. However, the `Promise.apply` method does not start an asynchronous computation; it just creates a fresh promise object. When the promise object is created, it does not contain a value or an exception. To assign a value or an exception to a promise, we use the `success` or `failure` method, respectively.

Perhaps you have noticed that promises are very similar to futures. Both futures and promises are initially empty and can be completed with either a value or an exception. This is intentional; every promise object corresponds to exactly one future object. To obtain the future associated with a promise, we can call the `future` method on the promise. Calling this method multiple times always returns the same future object.

A promise and a future represent two aspects of a single–assignment variable–the promise allows you to assign a value to the future object, whereas the future allows you to read that value.

In the following code snippet, we create two promises, `p` and `q`, that can hold string values. We then install a `foreach` callback on the future associated with the `p` promise and wait for one second. The callback is not invoked until the `p` promise is completed by calling the `success` method. We then fail the `q` promise in the same way and install a `failed.foreach` callback:

```
object PromisesCreate extends App {
  val p = Promise[String]
  val q = Promise[String]
  p.future foreach { case x => log(s"p succeeded with '$x'") }
  Thread.sleep(1000)
  p success "assigned"
  q failure new Exception("not kept")
  q.future.failed foreach { case t => log(s"q failed with $t") }
  Thread.sleep(1000)
}
```

Alternatively, we can use the `complete` method and specify a `Try[T]` object to complete the promise. Depending on whether the `Try[T]` object is a success or a failure, the promise is successfully completed or failed. Importantly, after a promise is either successfully completed or failed, it cannot be assigned an exception or a value again in any way. Trying to do so results in an exception. Note that this is true even when there are multiple threads simultaneously calling `success` or `complete`. Only one thread completes the promise, and the rest throw an exception.

Assigning a value or an exception to an already completed promise is not allowed and throws an exception.

We can also use the `trySuccess`, `tryFailure`, and `tryComplete` methods that correspond to `success`, `failure`, and `complete`, respectively, but return a Boolean value to indicate whether the assignment was successful. Recall that using `Future.apply` and callback methods with referentially transparent functions results in deterministic concurrent programs. As long as we do not use the `trySuccess`, `tryFailure`, and `tryComplete` methods, and none of the `success`, `failure`, and `complete` methods ever throw an exception, we can use promises and retain determinism in our programs.

We now have everything we need to implement our custom `Future.apply` method. We call it `myFuture` in the following example. The `myFuture` method takes a `b` by-name parameter that is the asynchronous computation. First, it creates a `p` promise. Then, it starts an asynchronous computation on the `global` execution context. This computation tries to evaluate `b` and complete the promise. However, if the `b` body throws a nonfatal exception, the asynchronous computation fails the promise with that exception. In the meanwhile, the `myFuture` method returns the future immediately after starting the asynchronous computation:

```
import scala.util.control.NonFatal
object PromisesCustomAsync extends App {
  def myFuture[T](b: =>T): Future[T] = {
    val p = Promise[T]
    global.execute(new Runnable {
      def run() = try {
        p.success(b)
      } catch {
        case NonFatal(e) => p.failure(e)
      }
    })
    p.future
  }
  val f = myFuture { "naa" + "na" * 8 + " Katamari Damacy!" }
  f foreach { case text => log(text) }
}
```

This is a common pattern when producing futures. We create a promise, let some other computation complete that promise, and return the corresponding future. However, promises were not invented just for our custom future computation method, `myFuture`. In the following sections, we will study use cases in which promises are useful.

Converting callback-based APIs

Scala futures are great. We already saw how they can be used to avoid blocking. We have learned that callbacks help us to avoid polling and busy-waiting. We witnessed that futures compose well with functional combinators and `for` comprehensions. In some cases, futures and promises even guarantee deterministic programs. But, we have to face the truth-not all legacy APIs were created using Scala futures. Although futures are now the right way to do asynchronous computing, various third-party libraries have different approaches to encoding latency.

Legacy frameworks deal with latency in the program with raw callbacks. Methods that take an unbounded amount of time to complete do not return the result; instead, they take a callback argument, which is invoked with the result later. JavaScript libraries and frameworks are a good example for this–there is a single thread executing a JavaScript program and it is unacceptable to block that thread every time we call a blocking method.

Such legacy systems have issues in large-scale development. First, they do not nicely compose, as we already saw. Second, they are hard to understand and reason about; a bunch of unstructured callbacks feels almost like spaghetti code. The control flow of the program is not apparent from the code, but is dictated by the internals of the library. This is called **inversion of control**. We would like to somehow create a bridge between legacy callback-based APIs and futures, and avoid this inversion of control. This is where promises come in handy.

Use promises to bridge the gap between callback-based APIs and futures.

Let's consider the `org.apache.commons.io.monitor` package from the **Commons IO** library. This package allows subscribing to filesystem events such as file and directory creation and deletion. Having become well versed in the use of futures, we do not want to deal with this API directly anymore. We, therefore, implement a `fileCreated` method that takes a directory name and returns a future with the name of the first file in that freshly created directory:

```
import org.apache.commons.io.monitor._
```

To subscribe to a filesystem event using this package, we first need to instantiate a `FileAlterationMonitor` object. This object periodically scans the filesystem for changes. After that, we need to create a `FileAlterationObserver` object, which observes a specific directory for changes. Finally, we create a `FileAlterationListenerAdaptor` object, which represents the callback. Its `onFileCreate` method is called when a file is created in the filesystem; we use it to complete the promise with the name of the file that was changed:

```
def fileCreated(directory: String): Future[String] = {
  val p = Promise[String]
  val fileMonitor = new FileAlterationMonitor(1000)
  val observer = new FileAlterationObserver(directory)
  val listener = new FileAlterationListenerAdaptor {
    override def onFileCreate(file: File): Unit =
      try p.trySuccess(file.getName) finally fileMonitor.stop()
  }
  observer.addListener(listener)
  fileMonitor.addObserver(observer)
  fileMonitor.start()
  p.future
}
```

Notice that the structure of this method is the same as the structure of the `myFuture` method. We first create a promise and defer the completion of the promise to some other computation. Then, we return the future associated with the promise. This recurring pattern is called the future-callback bridge.

We can now use the future to subscribe to the first file change in the filesystem. We add a `foreach` call to the future returned by the `fileCreated` method, create a new file in the editor, and witness how the program detects a new file:

```
fileCreated(".") foreach {
  case filename => log(s"Detected new file '$filename'")
}
```

A useful utility that is not available on futures is the `timeout` method. We want to call a `timeout` method that takes some number of `t` milliseconds and returns a future that is completed after at least `t` milliseconds. We apply the callback-future bridge to the `Timer` class from the `java.util` package. We use a single timer object for all the `timeout` calls:

```
import java.util._
private val timer = new Timer(true)
```

Again, we first create a promise `p`. This promise holds no useful information other than the fact that it is completed, so we give it the type `Promise[Unit]` . We then call the `Timer` class's `schedule` method with a `TimerTask` object that completes the `p` promise after `t` milliseconds:

```
def timeout(t: Long): Future[Unit] = {
  val p = Promise[Unit]
  timer.schedule(new TimerTask {
    def run() = {
      p success ()
      timer.cancel()
    }
  }, t)
  p.future
}
timeout(1000) foreach { case _ => log("Timed out!") }
Thread.sleep(2000)
```

The future returned by the `timeout` method can be used to install a callback, or it can be combined with other futures using combinators. In the next section, we will introduce new combinators for this purpose.

Extending the future API

Usually, the existing future combinators are sufficient for most tasks, but occasionally we want to define new ones. This is another use case for promises. Assume that we want to add a combinator to futures, as follows:

```
def or(that: Future[T]): Future[T]
```

This method returns a new future of the same type that is assigned the value of the `this` future or the `that` future, whichever is completed first. We cannot add this method directly to the `Future` trait because futures are defined in the Scala standard library, but we can create an implicit conversion that adds this method. Recall that, if you call a nonexistent `xyz` method on an object of some type `A`, the Scala compiler will search for all implicit conversions from type `A` to some other type that has the `xyz` method. One way to define such an implicit conversion is to use Scala's implicit classes:

```
implicit class FutureOps[T](val self: Future[T]) {
  def or(that: Future[T]): Future[T] = {
    val p = Promise[T]
    self onComplete { case x => p tryComplete x }
    that onComplete { case y => p tryComplete y }
    p.future
```

```
      }
    }
```

The implicit `FutureOps` class converts a future of type `Future[T]` to an object with an additional `or` method. Inside the `FutureOps` object, we refer to the original future with the name `self`; we cannot use `this` word, because `this` is a reserved keyword that refers to the `FutureOps` object. The `or` method installs callbacks on `self` and `that` future. Each of these callbacks calls the `tryComplete` method on the `p` promise; the callback that executes first successfully completes the promise. The `tryComplete` method in the other callback returns `false` and does not change the state of the promise.

Use promises to extend futures with additional functional combinators.

Note that we used the `tryComplete` method in this example, and the `or` combinator is nondeterministic as a result. The resulting future is completed with the value of one of the input futures depending on the execution schedule. In this particular case, this is exactly what we want.

Cancellation of asynchronous computations

In some cases, we want to cancel a future computation. This might be because a future computation takes more than the allotted amount of time, or because the user clicks on the **Cancel** button in the UI. In either case, we need to provide some alternative value for the canceled future.

Futures come without built-in support for cancellation. Once a future computation starts, it is not possible to cancel it directly. Recall from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, that violently stopping concurrent computations can be harmful, and this is why the `Thread` methods such as `stop` were deprecated in the early JDK releases.

One approach to cancel a future is to compose it with another future called the **cancellation future**. The `cancellation` future provides a default value when a future is canceled. We can use the `or` combinator, discussed in the previous section, along with the `timeout` method, to compose a future with its `cancellation` future:

```
val f = timeout(1000).map(_ => "timeout!") or Future {
  Thread.sleep(999)
  "work completed!"
}
```

The nondeterminism of the `or` combinator is apparent when running this program. The `timeout` and `sleep` statements are precisely tuned to occur approximately at the same time. Another thing worth noting is that the computation started by the `Future.apply` method does not actually stop if a timeout occurs. The `f` future is completed with the value `"timeout!"`, but the future computation proceeds concurrently. Eventually, it fails to set the value of the promise when calling `tryComplete` in the `or` combinator. In many cases, this is not a problem. An HTTP request that needs to complete a future does not occupy any computational resources, and will eventually timeout anyway. A keyboard event that completes a future only consumes a small amount of CPU time when it triggers. Callback-based futures can usually be canceled, as in the preceding example. On the other hand, a future that performs an asynchronous computation can use a lot of CPU power or other resources. We might want to ensure that actions such as scanning the filesystem or downloading a huge file really terminate.

A future computation cannot be forcefully stopped. Instead, there should exist some form of cooperation between the future computation and the client of the future. In the examples seen so far, asynchronous computations always use futures to communicate a value to the client. In this case, the client also communicates in the opposite direction to let the asynchronous computation know that it should stop. Naturally, we use futures and promises to accomplish this two-way communication.

First, we define a type `Cancellable[T]` as a pair of `Promise[Unit]` and `Future[T]` objects. The client will use the `Promise[Unit]` part to request a cancellation, and the `Future[T]` part to subscribe to the result of the computation:

```
object PromisesCancellation extends App {
  type Cancellable[T] = (Promise[Unit], Future[T])
```

The `cancellable` method takes the `b` body of the asynchronous computation. This time, the `b` body takes a single parameter, `Future[Unit]`, to check if the cancellation was requested. The `cancellable` method creates a `cancel` promise of the type `Promise[Unit]` and forwards its corresponding future to the asynchronous computation. We call this promise the **cancellation promise**. The `cancel` promise will be used to signal that the asynchronous computation `b` should end. After the asynchronous computation `b` returns some value `r`, the `cancel` promise needs to fail. This ensures that, if the type `Future[T]` is completed, then the client cannot successfully cancel the computation using the `cancel` promise:

```
  def cancellable[T](b: Future[Unit] => T): Cancellable[T] = {
    val cancel = Promise[Unit]
    val f = Future {
      val r = b(cancel.future)
      if (!cancel.tryFailure(new Exception))
```

```
          throw new CancellationException
        r
      }
      (cancel, f)
    }
```

If calling `tryFailure` on the `cancel` promise returns `false`, then the client must have already completed the `cancel` promise. In this case, we cannot fail the client's attempt to cancel the computation, so we throw a `CancellationException`. Note that we cannot omit this check, as it exists to avoid the race in which the client successfully requests the cancellation, and the future computation simultaneously completes the future.

The asynchronous computation must occasionally check if the future was canceled using the `isCompleted` method on the `cancel` future. If it detects that it was canceled, it must cease execution by throwing a `CancellationException` value:

```
    val (cancel, value) = cancellable { cancel =>
      var i = 0
      while (i < 5) {
        if (cancel.isCompleted) throw new CancellationException
        Thread.sleep(500)
        log(s"$i: working")
        i += 1
      }
      "resulting value"
    }
```

After the `cancellable` computation starts, the main thread waits for 1,500 milliseconds and then calls `trySuccess` to complete the cancellation promise. By this time, the cancellation promise could have already failed; in this case, calling `success` instead of the `trySuccess` method would result in an exception:

```
    Thread.sleep(1500)
    cancel trySuccess ()
    log("computation cancelled!")
    Thread.sleep(2000)
  }
```

We expect to see the final `working` message printed after the `"computation cancelled!"` message from the main thread. This is because the asynchronous computation uses polling and does not immediately detect that it was cancelled.

Use promises to implement cancellation or any other form of two-way communication between the client and the asynchronous computation.

Note that calling the `trySuccess` method on the `cancel` promise does not guarantee that the computation will really be canceled. It is entirely possible that the asynchronous computation fails the `cancel` promise before the client has a chance to cancel it. Thus, the client, such as the main thread in our example, should in general use the return value from the `trySuccess` method to check if the cancellation succeeded.

Futures and blocking

Examples in this book should have shed the light into why blocking is sometimes considered an anti-pattern. Futures and asynchronous computations mainly exist to avoid blocking, but in some cases, we cannot live without it. It is, therefore, valid to ask how blocking interacts with futures.

There are two ways to block with futures. The first is waiting until a future is completed. The second is blocking from within an asynchronous computation. We will study both the topics in this section.

Awaiting futures

In rare situations, we cannot use callbacks or future combinators to avoid blocking. For example, the main thread that starts multiple asynchronous computations has to wait for these computations to finish. If an execution context uses daemon threads, as is the case with the `global` execution context, the main thread needs to block to prevent the JVM process from terminating.

In these exceptional circumstances, we use the `ready` and `result` methods on the `Await` object from the `scala.concurrent` package. The `ready` method blocks the caller thread until the specified future is completed. The `result` method also blocks the caller thread, but returns the value of the future if it was completed successfully, or throws the exception in the future if the future was failed.

Both the methods require specifying a timeout parameter-the longest duration that the caller should wait for the completion of the future before a `TimeoutException` is thrown. To specify a timeout, we import the `scala.concurrent.duration` package. This allows us to write expressions such as `10.seconds`:

```
import scala.concurrent.duration._
object BlockingAwait extends App {
  val urlSpecSizeFuture = Future {
    val specUrl = "http://www.w3.org/Addressing/URL/url-spec.txt"
    Source.fromURL(specUrl).size
  }
  val urlSpecSize = Await.result(urlSpecSizeFuture, 10.seconds)
  log(s"url spec contains $urlSpecSize characters")
}
```

In this example, the main thread starts a computation that retrieves the URL specification and then awaits. By this time, the World Wide Web Consortium is worried that a DOS attack is under way, so this is the last time we download the URL specification.

Blocking in asynchronous computations

Waiting for the completion of a future is not the only way to block. Some legacy APIs do not use callbacks to asynchronously return results. Instead, such APIs expose the blocking methods. After we call a blocking method, we lose control over the thread; it is up to the blocking method to unblock the thread and return the control back.

Execution contexts are often implemented using thread pools. As we saw in `Chapter 3`, *Traditional Building Blocks of Concurrency*, blocking worker threads can lead to thread starvation. Thus, by starting future computations that block, it is possible to reduce parallelism and even cause deadlocks. This is illustrated in the following example, in which 16 separate future computations call the `sleep` method, and the main thread waits until they complete for an unbounded amount of time:

```
val startTime = System.nanoTime
val futures = for (_ <- 0 until 16) yield Future {
  Thread.sleep(1000)
}
for (f <- futures) Await.ready(f, Duration.Inf)
val endTime = System.nanoTime
log(s"Total time = ${(endTime - startTime) / 1000000} ms")
log(s"Total CPUs = ${Runtime.getRuntime.availableProcessors}")
```

Assume that you have eight cores in your processor. This program does not end in one second. Instead, the first batch of eight futures started by the `Future.apply` method will block all the worker threads for one second, and then another batch of eight futures will block for another second. As a result, none of our eight processor cores can do any useful work for one second.

Avoid blocking in asynchronous computations, as it can cause thread starvation.

If you absolutely must block, then the part of the code that blocks should be enclosed within the `blocking` call. This signals to the execution context that the worker thread is blocked and allows it to temporarily spawn additional worker threads if necessary:

```
val futures = for (_ <- 0 until 16) yield Future {
  blocking {
    Thread.sleep(1000)
  }
}
```

With the `blocking` call around the `sleep` call, the `global` execution context spawns additional threads when it detects that there is more work than the worker threads. All 16 future computations can execute concurrently, and the program terminates after one second.

The `Await.ready` and `Await.result` statements block the caller thread until the future is completed, and are in most cases used outside the asynchronous computations. They are blocking operations. The `blocking` statement is used inside asynchronous code to designate that the enclosed block of code contains a blocking call. It is not a blocking operation by itself.

The Scala Async library

In the final section of this chapter, we turn to the Scala**Async** library. You should understand that the Scala Async library does not add anything conceptually new to futures and promises. If you got this far in this chapter, you already know everything that you need to know about asynchronous programming, callbacks, future composition, promises, and blocking. You can start building asynchronous applications right away.

Having said that, the Scala Async library is a convenient library for futures and promises that allow expressing chains of asynchronous computations more conveniently. Every program that you express using the Scala Async library can also be expressed using futures and promises. Often, the Scala Async library allows writing shorter, more concise, and understandable programs.

The Scala Async library introduces two new method calls–the `async` and `await` methods. The `async` method is conceptually equivalent to the `Future.apply` method; it starts an asynchronous computation and returns a future object. The `await` method should not be confused with the `Await` object used to block on futures. The `await` method takes a future and returns that future's value. However, unlike the methods on the `Await` object, the `await` method does not block the underlying thread; we will soon see how this is possible.

The Scala Async library is currently not part of the Scala standard library. To use it, we need to add the following line to our build definition file:

```
libraryDependencies +=
  "org.scala-lang.modules" %% "scala-async" % "0.9.1"
```

As a simple example, consider the `delay` method, which returns a future that is completed after `n` seconds. We use the `async` method to start an asynchronous computation that calls the `sleep` method. When the `sleep` call returns, the future is completed:

```
def delay(n: Int): Future[Unit] = async {
  blocking { Thread.sleep(n * 1000) }
}
```

The `await` method must be statically enclosed within an `async` block in the same method; it is a compile-time error to invoke `await` outside of the `async` block. Whenever the execution inside the `async` block reaches an `await` statement, it stops until the value from the future in the `await` statement becomes available. Consider the following example:

```
async {
  log("T-minus 1 second")
  await { delay(1) }
  log("done!")
}
```

Here, the asynchronous computation in the `async` block prints `"T-minus 1 second"`. It then calls `delay` to obtain a future that is completed after one second. The `await` call designates that the computation can proceed only after the future returned by `delay` completes. After that happens, the `async` block prints the text `done`.

The natural question is: How can the Scala Async library execute the preceding example without blocking? The answer is that the Scala Async library uses Scala Macros to transform the code inside the `async` statement. The code is transformed in such a way that the code after every `await` statement becomes a callback registered to the future inside `await`. Immensely simplifying how this transformation works under the hood, the preceding code is equivalent to the following computation:

```
Future {
  log("T-minus 1 second")
  delay(1) foreach {
    case x => log("done!")
  }
}
```

As you can see, the equivalent code produced by the Scala Async library is completely non-blocking. The advantage of the `async`/`await` style code is that it is much more understandable. For example, it allows defining a custom `countdown` method that takes a number of seconds and an `n` and a `f` function to execute every second. We use a `while` loop for the `countdown` method inside the `async` block: each time an `await` instance is invoked, the execution is postponed for one second. The implementation using the Scala Async library feels like regular procedural code, but it does not incur the cost of blocking:

```
def countdown(n: Int)(f: Int => Unit): Future[Unit] = async {
  var i = n
  while (i > 0) {
    f(i)
    await { delay(1) }
    i -= 1
  }
}
```

The `countdown` method can be used from the main thread to print to the standard output every second. Since the `countdown` method returns a future, we can additionally install a `foreach` callback to execute after the `countdown` method is over:

```
countdown(10) { n => log(s"T-minus $n seconds") } foreach {
  case _ => log(s"This program is over!")
}
```

Having seen how expressive the Async library is in practice, the question is: When to use it in place of callbacks, future combinators, and for-comprehensions? In most cases, whenever you can express a chain of asynchronous computations inside a single method, you are free to use Async. You should use your best judgment when applying it; always choose the programming style that results in concise, more understandable, and more maintainable programs.

Use the Scala Async library when a chain of asynchronous computations can be expressed more intuitively as procedural code using the `async` and `await` statements.

Alternative future frameworks

Scala futures and promises API resulted from an attempt to consolidate several different APIs for asynchronous programming, among them, legacy Scala futures, Akka futures, Scalaz futures, and Twitter's Finagle futures. Legacy Scala futures and Akka futures have already converged to the futures and promises APIs that you've learned about so far in this chapter. Finagle's `com.twitter.util.Future` type is planned to eventually implement the same interface as `scala.concurrent.Future`, while the Scalaz `scalaz.concurrent.Future` type implements a slightly different interface. In this section, we give a brief description of Scalaz futures.

To use Scalaz, we add the following dependency to the `build.sbt` file:

```
libraryDependencies +=
  "org.scalaz" %% "scalaz-concurrent" % "7.0.6"
```

We now encode an asynchronous tombola program using Scalaz. The `Future` type in Scalaz does not have the `foreach` method. Instead, we use its `runAsync` method, which asynchronously runs the future computation to obtain its value, and then calls the specified callback:

```
import scalaz.concurrent._
object Scalaz extends App {
  val tombola = Future {
    scala.util.Random.shuffle((0 until 10000).toVector)
  }
  tombola.runAsync { numbers =>
    log(s"And the winner is: ${numbers.head}")
  }
  tombola.runAsync { numbers =>
    log(s"... ahem, winner is: ${numbers.head}")
  }
}
```

Unless you are terribly lucky and draw the same permutation twice, running this program reveals that the two `runAsync` calls print different numbers. Each `runAsync` call separately computes the permutation of the random numbers. This is not surprising, as Scalaz futures have the pull semantics, in which the value is computed each time some callback requests it, in contrast to the push semantics of Finagle and Scala futures, in which the callback is stored, and applied if and when the asynchronously computed value becomes available.

To achieve the same semantics as we would have with Scala futures, we need to use the `start` combinator that runs the asynchronous computation once, and caches its result:

```
val tombola = Future {
  scala.util.Random.shuffle((0 until 10000).toVector)
} start
```

With this change, the two `runAsync` calls use the same permutation of random numbers `tombola`, and print the same values.

We will not delve further into the internals of alternate frameworks. The fundamentals of futures and promises that you learned about in this chapter should be sufficient to easily familiarize yourself with other asynchronous programming libraries, should the need arise.

Summary

This chapter presented some powerful abstractions for asynchronous programming. We have seen how to encode latency with the `Future` type, how to avoid blocking with callbacks on futures, and how to compose values from multiple futures. We have learned that futures and promises are closely tied together and that promises allow interfacing with legacy callback-based systems. In cases where blocking was unavoidable, we learned how to use the `Await` object and the `blocking` statement. Finally, we learned that the Scala Async library is a powerful alternative for expressing future computations more concisely.

Futures and promises only allow dealing with a single value at a time. What if an asynchronous computation produces more than a single value before completing? Similarly, how do we efficiently execute thousands of asynchronous operations on different elements of large datasets? Should we use futures in such cases? In the next chapter, we will explore Scala's support for data-parallelism, a form of concurrency where similar asynchronous computations execute in parallel on different collection elements. We will see that using data-parallel collections is preferable to using futures when collections are large, as it results in a better performance.

Exercises

The following exercises summarize what we have learned about futures and promises in this chapter, and require implementing custom future factory methods and combinators. Several exercises also deal with several deterministic programming abstractions that were not covered in this chapter, such as single-assignment variables and maps:

1. Implement a command-line program that asks the user to input a URL of some website, and displays the HTML of that website. Between the time that the user hits ENTER and the time that the HTML is retrieved, the program should repetitively print a `.` to the standard output every 50 milliseconds, with a 2 second timeout. Use only futures and promises, and avoid synchronization primitives from the previous chapters. You may reuse the `timeout` method defined in this chapter.
2. Implement an abstraction called a single-assignment variable, represented by the `IVar` class:

```
class IVar[T] {
  def apply(): T = ???
  def :=(x: T): Unit = ???
}
```

When created, the `IVar` class does not contain a value, and calling `apply` results in an exception. After a value is assigned using the `:=` method, subsequent calls to `:=` throw an exception, and the `apply` method returns the previously assigned value. Use only futures and promises, and avoid the synchronization primitives from the previous chapters.

3. Extend the `Future[T]` type with the `exists` method, which takes a predicate and returns a `Future[Boolean]` object:

   ```
   def exists(p: T => Boolean): Future[Boolean]
   ```

   The resulting future is completed with `true` if and only if the original future is completed and the predicate returns `true`, and `false` otherwise. You can use future combinators, but you are not allowed to create any `Promise` objects in the implementation.

4. Repeat the previous exercise, but use `Promise` objects instead of future combinators.
5. Repeat the previous exercise, but use the Scala Async framework.
6. Implement the `spawn` method, which takes a command-line string, asynchronously executes it as a child process, and returns a future with the exit code of the child process:

   ```
   def spawn(command: String): Future[Int]
   ```

   Make sure that your implementation does not cause thread starvation.

7. Implement the `IMap` class, which represents a single-assignment map:

   ```
   class IMap[K, V] {
     def update(k: K, v: V): Unit
     def apply(k: K): Future[V]
   }
   ```

   Pairs of keys and values can be added to the `IMap` object, but they can never be removed or modified. A specific key can be assigned only once, and subsequent calls to `update` with that key result in an exception. Calling `apply` with a specific key returns a future, which is completed after that key is inserted into the map. In addition to futures and promises, you may use the `scala.collection.concurrent.Map` class.

8. Extend the type `Promise[T]` with the `compose` method, which takes a function of the type `S => T`, and returns a `Promise[S]` object:

   ```
   def compose[S](f: S => T): Promise[S]
   ```

   Whenever the resulting promise is completed with some value `x` of the type `S` (or failed), the original promise must be completed with the value `f(x)` asynchronously (or failed), unless the original promise is already completed.

9. Implement the `scatterGather` method, which given a sequence of tasks, runs those tasks as parallel asynchronous computations, then combines the results, and returns a future that contains the sequence of results from different tasks. The `scatterGather` method has the following interface:

   ```
   def scatterGather[T](tasks: Seq[() => T]): Future[Seq[T]]
   ```

10. Implement another version of the `timeout` method shown in this chapter, but without using the `blocking` construct or `Thread.sleep`. Instead use the `java.util.Timer` class from the JDK. What are the advantages of this new implementation?
11. A directed graph is a data structure composed from a finite set of nodes, where each node has a finite number of directed edges that connect it with other nodes in the graph. A directed acyclic graph, or shorter, DAG, is a directed graph data structure in which, starting from any node N and following any path along the directed edges, we cannot arrive back at N. In other words, directed edges of a DAG never form a cycle.

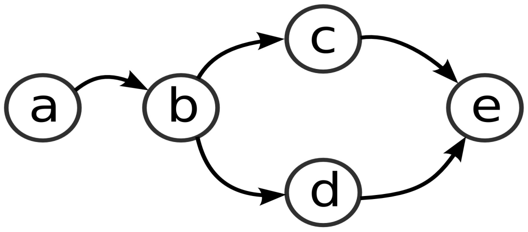
    One way to represent the nodes of the DAG data structure is as follows:

    ```
    class DAG[T](val value: T) {
      val edges = scala.collection.mutable.Set[DAG[T]]
    }
    ```

Here is an example of a DAG declaration:

```
val a = new DAG("a")
val b = new DAG("b")
val c = new DAG("c")
val d = new DAG("d")
val e = new DAG("e")
a.edges += b
b.edges += c
b.edges += d
c.edges += e
d.edges += e
```

The preceding DAG declaration is shown graphically in the following figure:



DAGs are often used to declare dependencies between different items, for example, build tasks in a project build tool or an IDE. Your task is to implement the `fold` method that takes a DAG node and a function that maps each item and its inputs into some value, and then returns the future with the resulting value of the input node:

```
def fold[T, S](g: DAG[T], f: (T, Seq[S]) => S): Future[S]
```

The `fold` method runs an asynchronous task for each item in the DAG to map the item and its inputs to a new value. Dependencies between DAG items must be respected: an item can only run after all of its dependencies have been computed. For example, in the previous figure, task `b` can only run after both `c` and `d` have produced a result.

5
Data-Parallel Collections

"Premature optimization is the root of all evil."
– Donald Knuth

So far, we have been composing multiple threads of computation into safe concurrent programs. In doing so, we focused on ensuring their correctness. We saw how to avoid blocking in concurrent programs, react to the completion of asynchronous computations, and how to use concurrent data structures to communicate information between threads. All these tools made organizing the structure of concurrent programs easier. In this chapter, we will focus mainly on achieving good performance. We require minimal or no changes in the organization of existing programs, but we will study how to reduce their running time using multiple processors. Futures from the previous chapter allowed doing this to a certain extent, but they are relatively heavyweight and inefficient when the asynchronous computation in each future is short.

Data parallelism is a form of computation where the same computation proceeds in parallel on different data elements. Rather than having concurrent computation tasks that communicate through the use of synchronization, in data-parallel programming, independent computations produce values that are eventually merged together in some way. An input to a data-parallel operation is usually a dataset such as a collection, and the output can be a value or another dataset.

In this chapter, we will study the following topics:

- Starting a data-parallel operation
- Configuring the parallelism level of a data-parallel collection
- Measuring performance and why it is important
- Differences between using sequential and parallel collections
- Using parallel collections together with concurrent collections
- Implementing a custom parallel collection, such as a parallel string
- Alternative data-parallel frameworks

In Scala, data-parallel programming was applied to the standard collection framework to accelerate bulk operations that are, by their nature, declarative and fit data parallelism well. Before studying data-parallel collections, we will present a brief overview of the Scala collection framework.

Scala collections in a nutshell

The Scala collections module is a package in the Scala standard library that contains a variety of general-purpose collection types. Scala collections provide a general and easy-to-use way of declaratively manipulating data using functional combinators. For example, in the following program, we use the `filter` combinator on a range of numbers to return a sequence of palindromes between 0 and 100,000; that is, numbers that are read in the same way in both the forward and reverse direction:

```
(0 until 100000).filter(x => x.toString == x.toString.reverse)
```

Scala collections define three basic types of collections: **sequences**, **maps**, and **sets**. Elements stored in sequences are ordered and can be retrieved using the `apply` method and an integer index. Maps store key-value pairs and can be used to retrieve a value associated with a specific key. Sets can be used to test the element membership with the `apply` method.

The Scala collection library makes a distinction between immutable collections, which cannot be modified after they are created, and mutable collections which can be updated after they are created. Commonly used immutable sequences are `List` and `Vector`, while `ArrayBuffer` is the mutable sequence of choice in most situations. Mutable `HashMap` and `HashSet` collections are maps and sets implemented using hash tables, while immutable `HashMap` and `HashSet` collections are based on the less widely known hash trie data structure.

Scala collections can be transformed into their parallel counterparts by calling the `par` method. The resulting collection is called a **parallel collection**, and its operations are accelerated by using multiple processors simultaneously. The previous example can run in parallel, as follows:

```
(0 until 100000).par.filter(x => x.toString == x.toString.reverse)
```

In the preceding code line, the filter combinator is a data-parallel operation. In this chapter, we will study parallel collections in more detail. We will see when and how to create parallel collections, study how they can be used together with sequential collections, and conclude by implementing a custom parallel collection class.

Using parallel collections

Most of the concurrent programming utilities we have studied so far are used in order to enable different threads of computation to exchange information. Atomic variables, the `synchronized` statement, concurrent queues, futures, and promises are focused on ensuring the correctness of a concurrent program. On the other hand, the parallel collection programming model is designed to be largely identical to that of sequential Scala collections; parallel collections exist solely in order to improve the running time of the program. In this chapter, we will measure the relative speedup of programs using parallel collections. To make this task easier, we will introduce the `timed` method to the package object used for the examples in this chapter. This method takes a block of code body, and returns the running time of the executing block of code `body`. It starts by recording the current time with the `nanoTime` method from the JDK `System` class. It then runs the body, records the time after the body executes, and computes the time difference:

```
@volatile var dummy: Any = _
def timed[T](body: =>T): Double = {
  val start = System.nanoTime
  dummy = body
  val end = System.nanoTime
  ((end - start) / 1000) / 1000.0
}
```

Certain runtime optimizations in the JVM, such as dead-code elimination, can potentially remove the invocation of the `body` block, causing us to measure an incorrect running time. To prevent this, we assign the return value of the `body` block to a volatile field named `dummy`.

Program performance is subject to many factors, and it is very hard to predict in practice. Whenever you can, you should validate your performance assumptions with measurements. In the following example, we use the Scala `Vector` class to create a vector with five million numbers and then shuffle that vector using the `Random` class from the `scala.util` package. We then compare the running time of the sequential and parallel `max` methods, which both find the greatest integer in the `numbers` collection:

```
import scala.collection._
import scala.util.Random
object ParBasic extends App {
  val numbers = Random.shuffle(Vector.tabulate(5000000)(i => i))
  val seqtime = timed { numbers.max }
  log(s"Sequential time $seqtime ms")
  val partime = timed { numbers.par.max }
  log(s"Parallel time $partime ms")
}
```

Running this program on an Intel i7-4900MQ quad-core processor with hyper-threading and Oracle JVM Version 1.7.0_51, we find that the sequential `max` method takes 244 milliseconds, while its parallel version takes 35 milliseconds. This is partly because parallel collections are optimized better than their sequential counterparts, and partly because they use multiple processors. However, on different processors and JVM implementations, results will vary.

Always validate assumptions about performance by measuring the execution time.

The `max` method is particularly well-suited for parallelization. Worker threads can independently scan subsets of the collection, such as `numbers`. When a worker thread finds the greatest integer in its own subset, it notifies the other processors and they agree on the greatest result. This final step takes much less time than searching for the greatest integer in a collection subset. We say that the `max` method is **trivially parallelizable**.

In general, data-parallel operations require more inter-processor communication than the `max` method. Consider the `incrementAndGet` method on atomic variables from `Chapter 3`, *Traditional Building Blocks of Concurrency*. We can use this method once again to compute unique identifiers. This time, we will use parallel collections to compute a large number of unique identifiers:

```
import java.util.concurrent.atomic._
object ParUid extends App {
  private val uid = new AtomicLong(0L)
  val seqtime = timed {
    for (i <- 0 until 10000000) uid.incrementAndGet()
  }
  log(s"Sequential time $seqtime ms")
  val partime = timed {
    for (i <- (0 until 10000000).par) uid.incrementAndGet()
  }
  log(s"Parallel time $partime ms")
}
```

This time, we use parallel collections in a `for` loop; recall that every occurrence of a `for` loop is desugared into the `foreach` call by the compiler. The parallel `for` loop from the preceding code is equivalent to the following:

```
(0 until 10000000).par.foreach(i => uid.incrementAndGet())
```

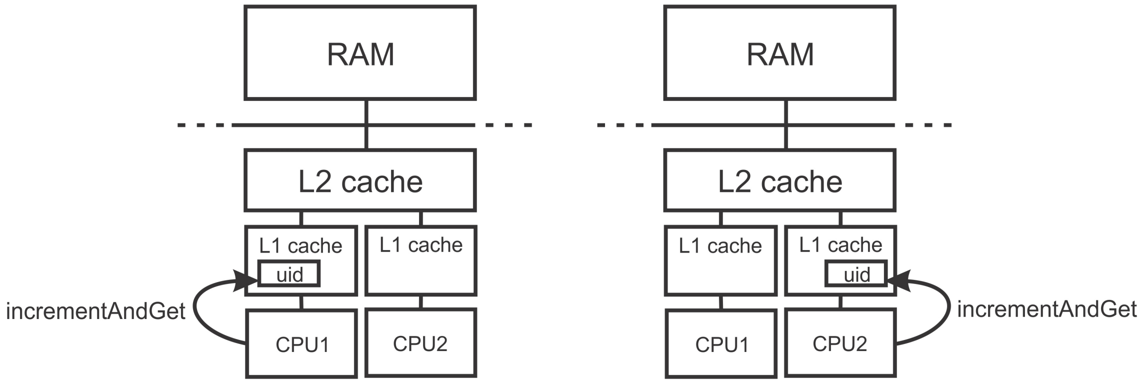
When the `foreach` method is called on a parallel collection, collection elements are processed concurrently. This means that separate worker threads simultaneously invoke the specified function, so proper synchronization must be applied. In our case, this synchronization is ensured by the atomic variable, as explained in `Chapter 3`, *Traditional Building Blocks of Concurrency*.

Running this program on our machine reveals that there is no increase in speed. In fact, the parallel version of the program is even slower; our program prints 320 milliseconds for the sequential `foreach` call, and 1,041 milliseconds for the parallel `foreach` call.

You might be surprised to see this; shouldn't a program be running at least four times faster on a quad-core processor with hyper-threading? As shown by the preceding example, this is not always the case. The parallel `foreach` call is slower because the worker threads simultaneously invoke the `incrementAndGet` method on the atomic variable, `uid`, and write to the same memory location at once.

Memory writes do not go directly to **Random Access Memory** (**RAM**) in modern architectures, as this would be too slow. Instead, modern computer architectures separate the CPU from the RAM with multiple levels of caches: smaller, more expensive, and much faster memory units that hold copies of parts of the RAM that the processor is currently using. The cache level closest to the CPU is called the L1 cache. The L1 cache is divided into short contiguous parts called **cache lines**. Typically, a cache-line size is 64 bytes. Although multiple cores can read the same cache line simultaneously, in standard multicore processors, the cache line needs to be in exclusive ownership when a core writes to it. When another core requests to write to the same cache line, the cache line needs to be copied to that core's L1 cache. The cache coherence protocol that enables this is called **Modified Exclusive Shared Invalid** (**MESI**), and its specifics are beyond the scope of this book. All you need to know is that exchanging cache-line ownership can be relatively expensive in terms of the processor's time scale.

Since the `uid` variable is atomic, the JVM needs to ensure a happens-before relationship between the writes and reads of the `uid` variable, as we know from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. To ensure the happens-before relationship, memory writes have to be visible to other processors. The only way to ensure this is to obtain the cache line in exclusive mode before writing to it. In our example, different processor cores repetitively exchange the ownership of the cache line in which the `uid` variable is allocated, and the resulting program becomes much slower than its sequential version. This is shown in the following diagram:



If different processors only read a shared memory location, then there is no slowdown. Writing to the same memory location is, on the other hand, an obstacle to scalability.

Writing to the same memory location with proper synchronization leads to performance bottlenecks and contention; avoid this in data-parallel operations.

Parallel programs share other resources in addition to computing power. When different parallel computations request more resources than are currently available, an effect known as **resource contention** occurs. The specific kind of resource contention that occurs in our example is called a **memory contention**, a conflict over exclusive rights to write to a specific part of memory.

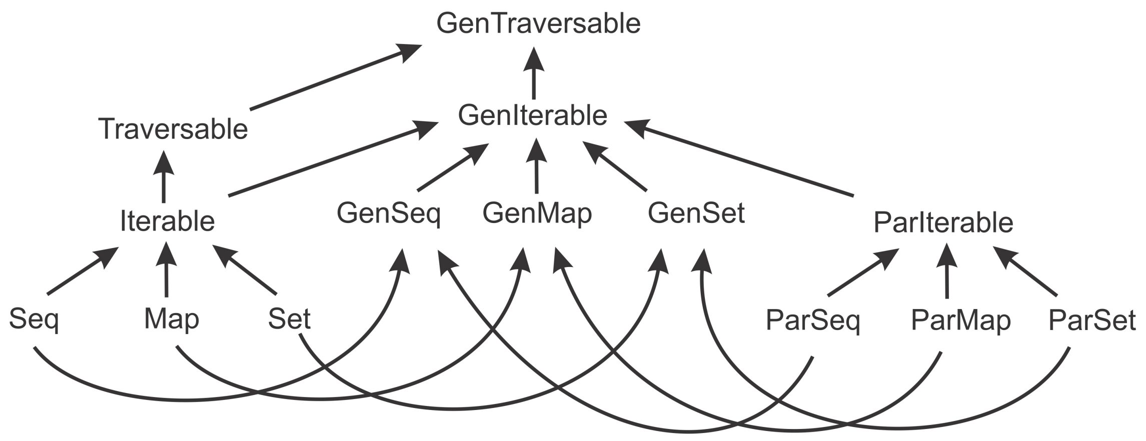
We can expect the same kind of performance degradation when using multiple threads to concurrently start the `synchronized` statement on the same object, repetitively modifying the same key in a concurrent map or simultaneously enqueueing elements to a concurrent queue; all these actions require writes to the same memory location. Nonetheless, this does not mean that threads should never write to the same memory locations. In some applications, concurrent writes occur very infrequently; the ratio between the time spent writing to contended memory locations and the time spent doing other work determines whether parallelization is beneficial or not. It is difficult to predict this ratio by just looking at the program; the `ParUid` example serves to illustrate that we should always measure in order to see the impact of contention.

Parallel collection class hierarchy

As we saw, parallel collection operations execute on different worker threads simultaneously. At any point during the execution of a parallel operation, an element in a parallel collection can be processed by at most one worker thread executing that operation. The block of code associated with the parallel operation is executed on each of the elements separately; in the `ParUid` example, the `incrementAndGet` method is called concurrently many times. Whenever a parallel operation executes any side-effects, it must take care to use proper synchronization; the naive approach of using `var` to store `uid` causes data races as it did in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. This is not the case with sequential Scala collections.

The consequence is that a parallel collection cannot be a subtype of a sequential collection. If it were, then the *Liskov substitution principle* would be violated. The Liskov substitution principle states that if type `S` is a subtype of `T`, then the object of type `T` can be replaced with objects of type `S` without affecting the correctness of the program.

In our case, if parallel collections are subtypes of sequential collections, then some methods can return a sequential sequence collection with the static type `Seq[Int]`, where the sequence object is a parallel sequence collection at runtime. Clients can call methods such as `foreach` on the collection without knowing that the body of the `foreach` method needs synchronization, and their programs would not work correctly. For these reasons, parallel collections form a hierarchy that is separate from the sequential collections, as shown in the following diagram:



Scala collection hierarchy

The preceding diagram shows the simplified Scala collection hierarchy with sequential collections on the left. The most general collection type is called `Traversable`. Different collection operations such as `find`, `map`, `filter`, or `reduceLeft` are implemented in terms of its abstract `foreach` method. Its `Iterable[T]` subtype offers additional operations such as `zip`, `grouped`, `sliding`, and `sameElements`, implemented using its `iterator` method. The `Seq`, `Map`, and `Set` traits are iterable collections that represent Scala sequences, maps, and sets, respectively. These traits are used to write code that is generic in the type of the concrete Scala collection. The following `nonNull` method copies elements from an `xs` collection that are different from `null`. Here, the `xs` collection can be a `Vector[T]`, `List[T]`, or some other sequence:

```
def nonNull(xs: Seq[T]): Seq[T] = xs.filter(_ != null)
```

Parallel collections form a separate hierarchy. The most general parallel collection type is called `ParIterable`. Methods such as `foreach`, `map`, or `reduce` on a `ParIterable` object execute in parallel. The `ParSeq`, `ParMap`, and `ParSet` collections are parallel collections that correspond to `Seq`, `Map`, and `Set`, but are not their subtypes. We can rewrite the `nonNull` method to use parallel collections:

```
def nonNull(xs: ParSeq[T]): ParSeq[T] = xs.filter(_ != null)
```

Although the implementation is identical, we can no longer pass a sequential collection to the `nonNull` method. We can call `.par` on the sequential `xs` collection before passing it to the `nonNull` method, but then the `filter` method will execute in parallel. Can we instead write code that is agnostic in the type of the collection? The generic collection types: `GenTraversable`, `GenIterable`, `GenSeq`, `GenMap`, and `GenSet` exist for this purpose. Each of them represents a supertype of the corresponding sequential or parallel collection type. For example, the `GenSeq` generic sequence type allows us to rewrite the `nonNull` method as follows:

```
def nonNull(xs: GenSeq[T]): GenSeq[T] = xs.filter(_ != null)
```

When using generic collection types, we need to remember that they might be implemented either as a sequential collection or as a parallel collection. Thus, as a precaution, if operations invoked on a generic collection execute any side effects, you should use synchronization.

Treat operations invoked on a generic collection type as if they are parallel.

Configuring the parallelism level

Parallel collections use all the processors by default; their underlying executor has as many workers as there are processors. We can change this default behavior by changing the `TaskSupport` object of the parallel collection. The basic `TaskSupport` implementation is the `ForkJoinTaskSupport` class. It takes a `ForkJoinPool` collection and uses it to schedule parallel operations.

Therefore, to change the parallelism level of a parallel collection, we instantiate a `ForkJoinPool` collection with the desired parallelism level:

```
import scala.concurrent.forkjoin.ForkJoinPool
object ParConfig extends App {
  val fjpool = new ForkJoinPool(2)
  val customTaskSupport = new parallel.ForkJoinTaskSupport(fjpool)
  val numbers = Random.shuffle(Vector.tabulate(5000000)(i => i))
  val partime = timed {
    val parnumbers = numbers.par
    parnumbers.tasksupport = customTaskSupport
    val n = parnumbers.max
    println(s"largest number $n")
  }
  log(s"Parallel time $partime ms")
}
```

Once a `TaskSupport` object is created, we can use it with different parallel collections. Every parallel collection has a `tasksupport` field that we use to assign the `TaskSupport` object to it.

Measuring the performance on the JVM

To correctly measure the running time on the JVM is not an easy task. Under the hood, the JVM does a lot more than meets the eye. The Scala compiler does not produce machine code directly runnable on the CPU. Instead, the Scala compiler produces a special intermediate instruction code called **Java bytecode**. When bytecode from the Scala compiler gets run inside the JVM, at first it executes in so-called **interpreted mode**; the JVM interprets each bytecode instruction and simulates the execution of the program. Only when the JVM decides that the bytecode in a certain method was run often enough does it compile the bytecode to machine code, which can be executed directly on the processor. This process is called **just-in-time compilation**.

The JVM needs standardized bytecode to be cross-platform; the same bytecode can be run on any processor or operating system that supports the JVM. However, the entire bytecode of a program cannot be translated to the machine code as soon as the program runs; this would be too slow. Instead, the JVM translates parts of the programs, such as specific methods, incrementally, in short compiler runs. In addition, the JVM can decide to additionally optimize certain parts of the program that execute very frequently. As a result, programs running on the JVM are usually slow immediately after they start, and eventually reach their optimal performance. Once this happens, we say that the JVM reached its steady state. When evaluating the performance on the JVM, we are usually interested in the **steady state**; most programs run long enough to achieve it.

To witness this effect, assume that you want to find out what the TEXTAREA tag means in HTML. You write the program that downloads the HTML specification and searches for the first occurrence of the TEXTAREA string. Having mastered asynchronous programming in Chapter 4, *Asynchronous Programming with Futures and Promises*, you can implement the getHtmlSpec method, which starts an asynchronous computation to download the HTML specification and returns a future value with the lines of the HTML specification. You then install a callback; once the HTML specification is available, you can call the indexWhere method on the lines to find the line that matches the regular expression .*TEXTAREA.*:

```
object ParHtmlSearch extends App {
  def getHtmlSpec() = Future {
    val url = "http://www.w3.org/MarkUp/html-spec/html-spec.txt"
    val specSrc = Source.fromURL(url)
    try specSrc.getLines.toArray finally specSrc.close()
  }
  getHtmlSpec() foreach { case specDoc =>
    def search(d: GenSeq[String]): Double =
      timed { d.indexWhere(line => line.matches(".*TEXTAREA.*")) }
    val seqtime = search(specDoc)
    log(s"Sequential time $seqtime ms")
    val partime = search(specDoc.par)
    log(s"Parallel time $partime ms")
  }
}
```

Running this example several times from SBT shows that the times vary. At first, the sequential and parallel versions execute for 45 and 16 milliseconds, respectively. Next time, they take 36 and 10 milliseconds, and subsequently 10 and 4 milliseconds. Note that we only observe this effect when running the examples inside the same JVM process as SBT itself.

We can draw a false conclusion that the steady state was reached at this point. In truth, we should run this program many more times before the JVM properly optimizes it. Therefore, we add the warmedTimed method to our package object. This method runs the block of code n times before measuring its running time. We set the default value for the n variable to 200; although there is no way to be sure that the JVM will reach a steady state after executing the block of code 200 times, this is a reasonable default:

```
def warmedTimed[T](n: Int = 200)(body: =>T): Double = {
  for (_ <- 0 until n) body
  timed(body)
}
```

We can now call the `warmedTimed` method instead of the `timed` method in the `ParHtmlSearch` example:

```
def search(d: GenSeq[String]) = warmedTimed() {
  d.indexWhere(line => line.matches(".*TEXTAREA.*"))
}
```

Doing so changes the running times on our machine to 1.5 and 0.5 milliseconds for the sequential and parallel versions of the program, respectively.

Make sure that the JVM is in the steady state before drawing any premature conclusions about the running time of a program.

There are other reasons why measuring performance on the JVM is hard. Even if the JVM reached a steady state for the part of the program we measure, the **Just-In-Time** (**JIT**) compiler can at any point pause the execution and translate some other part of the program, effectively slowing down our measurement. Then, the JVM provides automatic memory management. In languages such as C++, an invocation of the `new` keyword, which is used to allocate an object, must be accompanied by the corresponding `delete` call that frees the memory occupied by the object so that it can be reused later. In languages such as Scala and Java, however, there is no `delete` statement; objects are eventually freed automatically during the process called **Garbage Collection** (**GC**). Periodically, the JVM stops the execution, scans the heap for all objects no longer used in the program, and frees the memory they occupy. If we measure the running time of code that frequently causes GC cycles, the chances are that GC will skew the measurements. In some cases, the performance of the same program can vary from one JVM process to the other because the objects get allocated in a way that causes a particular memory access pattern, impacting the program's performance.

To get really reliable running time values, we need to run the code many times by starting separate JVM processes, making sure that the JVM reached a steady state, and taking the average of all the measurements. Frameworks such as **ScalaMeter**, introduced in `Chapter 9`, *Concurrency in Practice*, go a long way toward automating this process.

Caveats with parallel collections

Parallel collections were designed to provide a programming API similar to sequential Scala collections. Every sequential collection has a parallel counterpart and most operations have the same signature in both sequential and parallel collections. Still, there are some caveats when using parallel collections, and we will study them in this section.

Non-parallelizable collections

Parallel collections use **splitters**, represented with the `Splitter[T]` type, in order to provide parallel operations. A splitter is a more advanced form of an iterator; in addition to the iterator's `next` and `hasNext` methods, splitters define the `split` method, which divides the splitter `S` into a sequence of splitters that traverse parts of the `S` splitter:

```
def split: Seq[Splitter[T]]
```

This method allows separate processors to traverse separate parts of the input collection. The `split` method must be implemented efficiently, as this method is invoked many times during the execution of a parallel operation. In the vocabulary of computational complexity theory, the allowed asymptotic running time of the `split` method is **O**(log (*N*)), where *N* is the number of elements in the splitter. Splitters can be implemented for flat data structures such as arrays and hash tables, and tree-like data structures such as immutable hash maps and vectors. Linear data structures such as the Scala `List` and `Stream` collections cannot efficiently implement the `split` method. Dividing a long linked list of nodes into two parts requires traversing these nodes, which takes a time that is proportionate to the size of the collection.

Operations on Scala collections such as `Array`, `ArrayBuffer`, mutable `HashMap` and `HashSet`, `Range`, `Vector`, immutable `HashMap` and `HashSet`, and concurrent `TrieMap` can be parallelized. We call these collections *parallelizable*. Calling the `par` method on these collections creates a parallel collection that shares the same underlying dataset as the original collection. No elements are copied and the conversion is fast.

Other Scala collections need to be converted to their parallel counterparts upon calling `par`. We can refer to them as *non-parallelizable collections*. Calling the `par` method on non-parallelizable collections entails copying their elements into a new collection. For example, the `List` collection needs to be copied to a `Vector` collection when the `par` method is called, as shown in the following code snippet:

```
object ParNonParallelizableCollections extends App {
  val list = List.fill(1000000)("")
  val vector = Vector.fill(1000000)("")
  log(s"list conversion time: ${timed(list.par)} ms")
  log(s"vector conversion time: ${timed(vector.par)} ms")
}
```

Calling `par` on `List` takes 55 milliseconds on our machine, whereas calling `par` on `Vector` takes 0.025 milliseconds. Importantly, the conversion from a sequential collection to a parallel one is not itself parallelized, and is a possible sequential bottleneck.

Converting a non-parallelizable sequential collection to a parallel collection is not a parallel operation; it executes on the caller thread.

Sometimes, the cost of converting a non-parallelizable collection to a parallel one is acceptable. If the amount of work in the parallel operation far exceeds the cost of converting the collection, then we can bite the bullet and pay the cost of the conversion. Otherwise, it is more prudent to keep the program data in parallelizable collections and benefit from fast conversions. When in doubt, measure!

Non-parallelizable operations

While most parallel collection operations achieve superior performance by executing on several processors, some operations are inherently sequential, and their semantics do not allow them to execute in parallel. Consider the `foldLeft` method from the Scala collections API:

```
def foldLeft[S](z: S)(f: (S, T) => S): S
```

This method visits elements of the collection going from left to right and adds them to the accumulator of type `S`. The accumulator is initially equal to the zero value `z`, and is updated with the function `f` that uses the accumulator and a collection element of type `T` to produce a new accumulator. For example, given a list of integers `List(1, 2, 3)`, we can compute the sum of its integers with the following expression:

```
List(1, 2, 3).foldLeft(0)((acc, x) => acc + x)
```

This `foldLeft` method starts by assigning `0` to `acc`. It then takes the first element in the list `1` and calls the function `f` to evaluate `0 + 1`. The `acc` accumulator then becomes `1`. This process continues until the entire list of elements is visited, and the `foldLeft` method eventually returns the result `6`. In this example, the `S` type of the accumulator is set to the `Int` type. In general, the accumulator can have any type. When converting a list of elements to a string, the zero value is an empty string and the function `f` concatenates a string and a number.

The crucial property of the `foldLeft` operation is that it traverses the elements of the list by going from left to right. This is reflected in the type of the function `f`; it accepts an accumulator of type `S` and a list value of type `T`. The function `f` cannot take two values of the accumulator type `S` and merge them into a new accumulator of type `S`. As a consequence, computing the accumulator cannot be implemented in parallel; the `foldLeft` method cannot merge two accumulators from two different processors. We can confirm this by running the following program:

```
object ParNonParallelizableOperations extends App {
  ParHtmlSearch.getHtmlSpec() foreach { case specDoc =>
    def allMatches(d: GenSeq[String]) = warmedTimed() {
      val results = d.foldLeft("") { (acc, line) =>
        if (line.matches(".*TEXTAREA.*")) s"$acc\n$line" else acc
      }
    }
    val seqtime = allMatches(specDoc)
    log(s"Sequential time - $seqtime ms")
    val partime = allMatches(specDoc.par)
    log(s"Parallel time   - $partime ms")
  }
  Thread.sleep(2000)
}
```

In the preceding program, we use the `getHtmlSpec` method introduced earlier to obtain the lines of the HTML specification. We install a callback using the `foreach` call to process the HTML specification once it arrives; the `allMatches` method calls the `foldLeft` operation to accumulate the lines of the specification that contain the `TEXTAREA` string. Running the program reveals that both the sequential and parallel `foldLeft` operations take 5.6 milliseconds.

To specify how the accumulators produced by different processors should be merged together, we need to use the `aggregate` method. The `aggregate` method is similar to the `foldLeft` operation, but it does not specify that the elements are traversed from left to right. Instead, it only specifies that subsets of elements are visited going from left to right; each of these subsets can produce a separate accumulator. The `aggregate` method takes an additional function of type `(S, S) => S`, which is used to merge multiple accumulators:

```
d.aggregate("")(
  (acc, line) =>
  if (line.matches(".*TEXTAREA.*")) s"$acc\n$line" else acc,
  (acc1, acc2) => acc1 + acc2
)
```

Running the example again shows the difference between the sequential and parallel versions of the program; the parallel `aggregate` method takes 1.4 milliseconds to complete on our machine.

When doing these kinds of reduction operation in parallel, we can alternatively use the `reduce` or `fold` methods, which do not guarantee going from left to right. The `aggregate` method is more expressive, as it allows the accumulator type to be different from the type of the elements in the collection.

Use the `aggregate` method to execute parallel reduction operations.

Other inherently sequential operations include `foldRight`, `reduceLeft`, `reduceRight`, `reduceLeftOption`, `reduceRightOption`, `scanLeft`, `scanRight`, and methods that produce non-parallelizable collections such as the `toList` method.

Side effects in parallel operations

As their name implies, parallel collections execute on multiple threads concurrently. We have already learned in Chapter 2, *Concurrency on the JVM and the Java Memory Model*, that multiple threads cannot correctly modify shared memory locations without the use of synchronization. Assigning to a mutable variable from a parallel collection operation may be tempting, but it is almost certainly incorrect. This is best illustrated by the following example, in which we construct two sets, a and b, where b is the subset of the elements in a, and then uses the total mutable variable to count the size of the intersection:

```
object ParSideEffectsIncorrect extends App {
  def intersectionSize(a: GenSet[Int], b: GenSet[Int]): Int = {
    var total = 0
    for (x <- a) if (b contains x) total += 1
    total
  }
  val a = (0 until 1000).toSet
  val b = (0 until 1000 by 4).toSet
  val seqres = intersectionSize(a, b)
  val parres = intersectionSize(a.par, b.par)
  log(s"Sequential result - $seqres")
  log(s"Parallel result   - $parres")
}
```

Instead of returning 250, the parallel version nondeterministically returns various wrong results. Note that you might have to change the sizes of the sets a and b to witness this:

```
    run-main-32: Sequential result - 250
    run-main-32: Parallel result   - 244
```

To ensure that the parallel version returns the correct results, we can use an atomic variable and its incrementAndGet method. However, this leads to the same scalability problems we had before. A better alternative is to use the parallel count method:

```
a.count(x => b contains x)
```

If the amount of work executed per element is low and the matches are frequent, the parallel count method will result in better performance than the foreach method with an atomic variable.

To avoid the need for synchronization and ensure better scalability, favor declarative-style parallel operations instead of the side effects in parallel `for` loops.

Similarly, we must ensure that the memory locations read by a parallel operation are protected from concurrent writes. In the last example, the `b` set should not be concurrently mutated by some thread while the parallel operation is executing; this leads to the same incorrect results as using mutable variables from within the parallel operation.

Nondeterministic parallel operations

In `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we saw that multithreaded programs can be nondeterministic; given the same inputs, they can produce different outputs depending on the execution schedule. The `find` collection operation returns an element matching a given predicate. The parallel `find` operation returns whichever element was found first by some processor. In the following example, we use `find` to search the HTML specification for occurrences of the `TEXTAREA` string; running the example several times gives different results, because the `TEXTAREA` string occurs in many different places in the HTML specification:

```
object ParNonDeterministicOperation extends App {
  ParHtmlSearch.getHtmlSpec() foreach { case specDoc =>
    val patt = ".*TEXTAREA.*"
    val seqresult = specDoc.find(_.matches(patt))
    val parresult = specDoc.par.find(_.matches(patt))
    log(s"Sequential result - $seqresult")
    log(s"Parallel result   - $parresult")
  }
  Thread.sleep(3000)
}
```

If we want to retrieve the first occurrence of the `TEXTAREA` string, we need to use `indexWhere` instead:

```
val index = specDoc.par.indexWhere(_.matches(patt))
val parresult = if (index != -1) Some(specDoc(index)) else None
```

Parallel collection operations other than `find` are deterministic as long as their operators are **pure functions**. A pure function is always evaluated to the same value, given the same inputs, and does not have any side effects. For example, the function `(x: Int) => x + 1` is a pure function. By contrast, the following function `f` is not pure, because it changes the state of the `uid` value:

```
val uid = new AtomicInteger(0)
val f = (x: Int) => (x, uid.incrementAndGet())
```

Even if a function does not modify any memory locations, it is not pure if it reads memory locations that might change. For example, the following `g` function is not pure:

```
val g = (x: Int) => (x, uid.get)
```

When used with a non-pure function, any parallel operation can become nondeterministic. Mapping the range of values to unique identifiers in parallel gives a nondeterministic result, as illustrated by the following call:

```
val uids: GenSeq[(Int, Int)] = (0 until 10000).par.map(f)
```

The resulting sequence, `uids`, is different in separate executions. The parallel `map` operation retains the relative order of elements from the range `0 until 10000`, so the tuples in `uids` are ordered by their first elements from 0 until 10,000. On the other hand, the second element in each tuple is assigned nondeterministically; in one execution, the `uids` sequence can start with the `(0, 0), (1, 2), (2, 3), ...` and in another, with `(0, 0), (1, 4), (2, 9), ....`

Commutative and associative operators

Parallel collection operations such as `reduce`, `fold`, `aggregate`, and `scan` take binary operators as part of their input. A binary operator is a function `op` that takes two arguments, `a` and `b`. We can say that the binary operator `op` is **commutative** if changing the order of its arguments returns the same result, that is, `op(a, b) == op(b, a)`. For example, adding two numbers together is a commutative operation. Concatenating two strings is not a commutative operation; we get different strings depending on the concatenation order.

Binary operators for the parallel `reduce`, `fold`, `aggregate`, and `scan` operations never need to be commutative. Parallel collection operations always respect the relative order of the elements when applying binary operators, provided that the underlying collections have any ordering. Elements in sequence collections, such as `ArrayBuffer` collections, are always ordered. Other collection types can order their elements but are not required to do so.

In the following example, we can concatenate the strings inside an `ArrayBuffer` collection into one long string by using the sequential `reduceLeft` operation and the parallel `reduce` operation. We then convert the `ArrayBuffer` collection into a set, which does not have an ordering:

```
object ParNonCommutativeOperator extends App {
  val doc = mutable.ArrayBuffer.tabulate(20)(i => s"Page $i, ")
  def test(doc: GenIterable[String]) {
    val seqtext = doc.seq.reduceLeft(_ + _)
    val partext = doc.par.reduce(_ + _)
    log(s"Sequential result - $seqtext\n")
    log(s"Parallel result   - $partext\n")
  }
  test(doc)
  test(doc.toSet)
}
```

We can see that the string is concatenated correctly when the parallel `reduce` operation is invoked on a parallel array, but the order of the pages is mangled both for the `reduceLeft` and `reduce` operations when invoked on a set; the default Scala set implementation does not order the elements.

Binary operators used in parallel operations do not need to be commutative.

An `op` binary operator is **associative** if applying `op` consecutively to a sequence of values `a`, `b`, and `c` gives the same result regardless of the order in which the operator is applied, that is, `op(a, op(b, c)) == op(op(a, b), c)`. Adding two numbers together or computing the larger of the two numbers is an associative operation. Subtraction is not associative, as `1 - (2 - 3)` is different from `(1 - 2) - 3`.

Parallel collection operations usually require associative binary operators. While using subtraction with the `reduceLeft` operation means that all the numbers in the collection should be subtracted from the first number, using subtraction in the `reduce`, `fold`, or `scan` methods gives nondeterministic and incorrect results, as illustrated by the following code snippet:

```
object ParNonAssociativeOperator extends App {
  def test(doc: GenIterable[Int]) {
    val seqtext = doc.seq.reduceLeft(_ - _)
    val partext = doc.par.reduce(_ - _)
    log(s"Sequential result - $seqtext\n")
    log(s"Parallel result   - $partext\n")
  }
  test(0 until 30)
}
```

While the `reduceLeft` operation consistently returns `-435`, the `reduce` operation returns meaningless results at random.

Make sure that binary operators used in parallel operations are associative.

Parallel operations such as `aggregate` require the multiple binary operators, `sop` and `cop`:

```
def aggregate[S](z: S)(sop: (S, T) => S, cop: (S, S) => S): S
```

The `sop` operator is of the same type as the operator required by the `reduceLeft` operation. It takes an accumulator and the collection element. The `sop` operator is used to fold elements within a subset assigned to a specific processor. The `cop` operator is used to merge the subsets together and is of the same type as the operators for `reduce` and `fold`. The `aggregate` operation requires that `cop` is associative and that `z` is the **zero element** for the accumulator, that is, `cop(z, a) == a`. Additionally, the `sop` and `cop` operators must give the same result irrespective of the order in which element subsets are assigned to processors, that is, `cop(sop(z, a), sop(z, b)) == cop(z, sop(sop(z, a), b))`.

Using parallel and concurrent collections together

We have already seen that parallel collection operations are not allowed to access mutable states without the use of synchronization. This includes modifying sequential Scala collections from within a parallel operation. Recall that we used a mutable variable in the section on side effects to count the size of the intersection. In the following example, we will download the URL and HTML specifications, convert them to sets of words, and try to find an intersection of their words. In the `intersection` method, we use a `HashSet` collection and update it in parallel. Collections in the `scala.collection.mutable` package are not thread-safe. The following example nondeterministically drops elements, corrupts the buffer state, or throws exceptions:

```
object ConcurrentWrong extends App {
  import ParHtmlSearch.getHtmlSpec
  import ch4.FuturesCallbacks.getUrlSpec
  def intersection(a: GenSet[String], b: GenSet[String]) = {
    val result = new mutable.HashSet[String]
    for (x <- a.par) if (b contains x) result.add(x)
    result
  }
  val ifut = for {
    htmlSpec <- getHtmlSpec()
    urlSpec <- getUrlSpec()
  } yield {
    val htmlWords = htmlSpec.mkString.split("\\s+").toSet
    val urlWords = urlSpec.mkString.split("\\s+").toSet
    intersection(htmlWords, urlWords)
  }
  ifut onComplete { case t => log(s"Result: $t") }
  Thread.sleep(3000)
}
```

We learned in `Chapter 3`, *Traditional Building Blocks of Concurrency*, that concurrent collections can be safely modified by multiple threads without the risk of data corruption. We use the concurrent skip list collection from the JDK to accumulate words that appear in both specifications. The `decorateAsScala` object is used to add the `asScala` method to Java collections:

```
import java.util.concurrent.ConcurrentSkipListSet
import scala.collection.convert.decorateAsScala._
def intersection(a: GenSet[String], b: GenSet[String]) = {
  val skiplist = new ConcurrentSkipListSet[String]
  for (x <- a.par) if (b contains x) skiplist.add(x)
  val result: Set[String] = skiplist.asScala
  result
}
```

Weakly consistent iterators

As we saw in `Chapter 3`, *Traditional Building Blocks of Concurrency*, iterators on most concurrent collections are weakly consistent. This means that they are not guaranteed to correctly traverse the data structure if some thread concurrently updates the collection during traversal.

When executing a parallel operation on a concurrent collection, the same limitation applies; the traversal is weakly consistent and might not reflect the state of the data structure at the point when the operation started. The Scala `TrieMap` collection is an exception to this rule. In the following example, we will create a `TrieMap` collection called `cache` containing numbers between 0 and 100, mapped to their string representation. We will then start a parallel operation that traverses these numbers and adds the mappings for their negative values to the map:

```
object ConcurrentTrieMap extends App {
  val cache = new concurrent.TrieMap[Int, String]()
  for (i <- 0 until 100) cache(i) = i.toString
  for ((number, string) <- cache.par) cache(-number) = s"-$string"
  log(s"cache - ${cache.keys.toList.sorted}")
}
```

The parallel `foreach` operation does not traverse entries added after the parallel operation started; only positive numbers are reflected in the traversal. The `TrieMap` collection is implemented using the Ctrie concurrent data structure, which atomically creates a snapshot of the collection when the parallel operation starts. Snapshot creation is efficient and does not require you to copy the elements; subsequent update operations incrementally rebuild parts of the `TrieMap` collection.

TIP

Whenever program data needs to be simultaneously modified and traversed in parallel, use the `TrieMap` collection.

Implementing custom parallel collections

Parallel collections in the Scala standard library are sufficient for most tasks, but in some cases we want to add parallel operations to our own collections. The Java `String` class does not have a direct parallel counterpart in the parallel collections framework. In this section, we will study how to implement a custom `ParString` class that supports parallel operations. We will then use our custom parallel collection class in several example programs.

The first step in implementing a custom parallel collection is to extend the correct parallel collection trait. A parallel string is a sequence of characters, so we need to extend the `ParSeq` trait with the `Char` type argument. Once a string is created, it can no longer be modified; we say that the string is an immutable collection. For this reason, we extend a subtype of the `scala.collection.parallel.ParSeq` trait, the `ParSeq` trait from the `scala.collection.parallel.immutable` package:

```
class ParString(val str: String) extends immutable.ParSeq[Char] {
  def apply(i: Int) = str.charAt(i)
  def length = str.length
  def splitter = new ParStringSplitter(str, 0, str.length)
  def seq = new collection.immutable.WrappedString(str)
}
```

When we extend a parallel collection, we need to implement its `apply`, `length`, `splitter`, and `seq` methods. The `apply` method returns an element at position `i` in the sequence, and the `length` method returns the total number of elements in the sequence. These methods are equivalent to the methods on sequential collections, so we use the `String` class's `charAt` and `length` methods to implement them. Where defining a custom regular sequence requires implementing its `iterator` method, custom parallel collections need a `splitter` method. Calling `splitter` returns an object of the `Splitter[T]` type, a special iterator that can be split into subsets. We implement the `splitter` method to return a `ParStringSplitter` object, which we will show you shortly. Finally, parallel collections need a `seq` method, which returns a sequential Scala collection. Since `String` itself comes from Java and is not a Scala collection, we will use its `WrappedString` wrapper class from the Scala collections library.

Our custom parallel collection class is almost complete; we only need to provide the implementation for the `ParStringSplitter` object. We will study how to do this next.

Splitters

A splitter is an iterator that can be efficiently split into disjoint subsets. Here, efficient means that the splitter's `split` method must have **O**(*log*(*N*)) running time, where *N* is the number of elements in the splitter. Stated informally, a splitter is not allowed to copy large parts of the collection when split; if it did, the computational overhead from splitting would overcome any benefits from parallelization and become a serial bottleneck.

The easiest way to define a new `Splitter` class for the Scala parallel collection framework is to extend the `IterableSplitter[T]` trait, which has the following simplified interface:

```
trait IterableSplitter[T] extends Iterator[T] {
  def dup: IterableSplitter[T]
  def remaining: Int
  def split: Seq[IterableSplitter[T]]
}
```

The splitter interface declares the `dup` method which duplicates the current splitter. This method simply returns a new splitter pointing to the same subset of the collection. Splitters also define the `remaining` method, which returns the number of elements that the splitter can traverse by calling `next` before the `hasNext` method returns `false`. The `remaining` method does not change the state of the splitter and can be called as many times as necessary.

However, the `split` method can be called only once and it invalidates the splitter; none of the splitter's methods should be called after calling the `split` method. The `split` method returns a sequence of splitters that iterate over the disjoint subsets of the original splitter. If the original splitter has two or more elements remaining, then none of the resulting splitters should be empty, and the `split` method should return at least two splitters. If the original splitter has a single element or no elements remaining, then `split` is allowed to return empty splitters. Importantly, the splitters returned by `split` should be approximately equal in size; this helps the parallel collection scheduler achieve good performance.

To allow sequence-specific operations such as `zip`, `sameElements`, and `corresponds`, parallel sequence collections use a more refined subtype of the `IterableSplitter` trait, called the `SeqSplitter` trait:

```
trait SeqSplitter[T] extends IterableSplitter[T] {
  def psplit(sizes: Int*): Seq[SeqSplitter[T]]
}
```

Sequence splitters declare an additional method, `psplit`, which takes the list of sizes for the splitter partitions and returns as many splitters and elements as specified by the `sizes` parameter. If `sizes` specifies more elements than there are available in the splitter, additional empty splitters are returned at the end of the resulting sequence. For example, calling `s.psplit(10, 20, 15)` on a splitter with only 15 elements yields three splitters with sizes 10, five, and zero.

Similarly, if the `sizes` parameter specifies fewer elements than there are in the splitter, an additional splitter with the remaining elements is appended at the end.

Our parallel string class is a parallel sequence, so we need to implement a sequence splitter. We can start by extending the `SeqSplitter` class with the `Char` type parameter:

```
class ParStringSplitter
  (val s: String, var i: Int, val limit: Int)
extends SeqSplitter[Char] {
```

We add the `s` field pointing to the underlying `String` object in the `ParStringSplitter` constructor. A parallel string splitter must represent a subset of the elements in the string, so we add an `i` field to represent the position of the next character that will be traversed by the splitter. Note that `i` does not need to be synchronized; the splitter is only used by one processor at a time. The `limit` field contains the position after the last character in the splitter. This way, our splitter class represents substrings of the original string.

Implementing methods inherited from the `Iterator` trait is easy. As long as `i` is less than `limit`, `hasNext` must return `true`. The `next` method uses `i` to read the character at that position, increment `i`, and return the character:

```
  final def hasNext = i < limit
  final def next = {
    val r = s.charAt(i)
    i += 1
    r
  }
```

The `dup` and `remaining` methods are straightforward; the `dup` method creates a new parallel string splitter using the state of the current splitter, and the `remaining` method uses `limit` and `i` to compute the number of remaining elements:

```
def dup = new ParStringSplitter(s, i, limit)
def remaining = limit - i
```

The main parts of a splitter are its `split` and `psplit` methods. Luckily, `split` can be implemented in terms of `psplit`. If there is more than one element remaining, we call the `psplit` method. Otherwise, if there are no elements to split, we return the `this` splitter:

```
def split = {
  val rem = remaining
  if (rem >= 2) psplit(rem / 2, rem - rem / 2)
  else Seq(this)
}
```

The `psplit` method uses `sizes` to peel off parts of the original splitter. It does so by incrementing the `i` variable and creating a new splitter for each size `sz` in the `sizes` parameter. Recall that the current splitter is considered invalidated after calling the `split` or `psplit` method, so we are allowed to mutate its `i` field:

```
    def psplit(sizes: Int*): Seq[ParStringSplitter] = {
      val ss = for (sz <- sizes) yield {
        val nlimit = (i + sz) min limit
        val ps = new ParStringSplitter(s, i, nlimit)
        i = nlimit
        ps
      }
      if (i == limit) ss
      else ss :+ new ParStringSplitter(s, i, limit)
    }
  }
```

Note that we never copy the string underlying the splitter; instead, we update the indices that mark the beginning and the end of the splitter.

We have now completed our `ParString` class; we can use it to execute parallel operations on strings. We can also use it to count the number of uppercase characters in the string as follows:

```
object CustomCharCount extends App {
  val txt = "A custom text " * 250000
  val partxt = new ParString(txt)
  val seqtime = warmedTimed(50) {
    txt.foldLeft(0) { (n, c) =>
      if (Character.isUpperCase(c)) n + 1 else n
    }
  }
  log(s"Sequential time - $seqtime ms")
  val partime = warmedTimed(50) {
    partxt.aggregate(0)(
      (n, c) => if (Character.isUpperCase(c)) n + 1 else n,
      _ + _)
  }
  log(s"Parallel time   - $partime ms")
}
```

On our machine, the sequential `foldLeft` call takes 57 milliseconds, and the parallel `aggregate` call takes 19 milliseconds. This is a good indication that we have implemented parallel strings efficiently.

Combiners

Collection methods in the Scala standard library are divided into two major groups: **accessor** and **transformer** methods. Accessor methods, such as `foldLeft`, `find`, or `exists`, return a single value from the collection. By contrast, transformer methods, such as `map`, `filter`, or `groupBy`, create new collections and return them as results.

To generically implement transformer operations, the Scala collection framework uses an abstraction called a **builder**, which has roughly the following interface:

```
trait Builder[T, Repr] { // simplified interface
  def +=(x: T): Builder[T, Repr]
  def result: Repr
  def clear(): Unit
}
```

Here, the Repr type is of a collection that a specific builder can produce, and T is the type of its elements. A builder is used by repetitively calling its += method to add more elements, and eventually calling the result method to obtain the collection. After the result method is called, the contents of the builder are undefined. The clear method can be used to reset the state of the builder.

Every collection defines a custom builder used in various transformer operations. For example, the filter operation is defined in the Traversable trait, roughly as follows:

```
def newBuilder: Builder[T, Traversable[T]]
def filter(p: T => Boolean): Traversable[T] = {
  val b = newBuilder
  for (x <- this) if (p(x)) b += x
  b.result
}
```

In the preceding example, the filter implementation relies on the abstract newBuilder method, which is implemented in subclasses of the Traversable trait. This design allows defining all collection methods once, and only provide the foreach method (or the iterator) and the newBuilder method when declaring a new collection type.

Combiners are a parallel counterpart of standard builders, and are represented with the Combiner[T, Repr] type, which subtypes the Builder[T, Repr] type:

```
trait Combiner[T, Repr] extends Builder[T, Repr] {
  def size: Int
  def combine[N <: T, NewRepr >: Repr]
    (that: Combiner[N, NewRepr]): Combiner[N, NewRepr]
}
```

The size method is self-explanatory. The combine method takes another combiner called that, and produces a third combiner that contains the elements of the this and that combiners. After the combine method returns, the contents of both the this and that combiners are undefined, and should not be used again. This constraint allows reusing the this or that combiner object as the resulting combiner. Importantly, if that combiner is the same runtime object as the this combiner, the combine method should just return the this combiner.