Scala Threads and Futures

Sergei Winitzki

Workday, Inc.

September 26, 2017

JVM threads

Java `Thread` API: low-level and underpowered

- user code must create/maintain thread, start, interrupt
- cannot reuse one thread for different tasks
- "heavy": cannot create more than about 2,000 JVM threads
- difficult to synchronize across threads (wait / notify / synchronize)
- code is error-prone, hard to debug
- exceptions on a thread are invisible

Plain C multithreading was essentially just as hard

ThreadPoolExecutor = thread pool + task queue

Java ThreadPoolExecutor:

- has a queue of pending tasks
- runs tasks on a dynamically managed thread pool
- reuses threads for different tasks
- Fork/Join executor: additional facilities for synchronization
- cannot be restarted after shutdown

Main pattern of usage:

- Run a large number of short tasks on a small, fixed number of threads

Usages of thread pools

- `scala.concurrent.ExecutionContext` is a wrapper over a thread pool
- `akka.actor.ActorSystem` contains a thread pool
- Apache's `HttpAsyncClient` contains a thread pool
- Scala's `Future[T]` operations (`map`, `flatMap`) use implicit `ExecutionContext`

First look at `scala.concurrent.Future`

Main features of `scala.concurrent.Future`:

- "value semantics"

  `val f:  Future[Boolean]`

  – represents a Boolean value that *will become available* in the future
- "container semantics": has `map` and `flatMap`
- error handling and failure recovery
- use `scala.concurrent.Promise[T]` to convert callback APIs to `Future[T]`

Note: `java.util.concurrent.Future` is just a callback wrapper class

How does a Future run?

```
implicit ec = somebody.giveMeExecContextPlease()
val f:  Future[Array[Float]] = Future { long_computation() }
logger.info(‘‘Long computation started’’)
```

Getting a `Future[T]` value means:

- a task was queued on that *somebody's* thread pool
- we could get the result value when it becomes available
- we have no way of knowing when that will happen

How does `map` work on a `Future`?

```
implicit ec = somebody.giveMeExecContextPlease()
val f: Future[Array[Float]] = Future { long_computation() }
logger.info(‘‘Long computation started’’)
val s: Future[Int] = f.map(_.length)
```

- using `map` requires an `ExecutionContext`
- the new computation (_.length) will run once the array is ready
- the new computation may run on another thread!
- `flatMap` also works: `Future[Future[T]]` can be “flattened” to `Future[T]`

How to work with APIs returning a `Future`?

- get an `ExecutionContext` (e.g. `actorSystem.dispatcher`)
- use `map` or `flatMap` to specify computations to be done in the future
- return another `Future[T]` value
- avoid using `Await.result` if you can

With modern libraries, no need to wait!
Examples:

- Akka-Streaming accepts a `Future[T]` value inside `mapAsync`
- Akka-HTTP accepts a `Future[T]` value inside an HTTP route

How to convert callback APIs into a `Future`?

- create a `Promise[T]` value
  `val p = Promise[Int]()`
- in the callback, resolve the promise
  `p.succeed(123)`
- return the future value
  `p.future`

Example:

- converting Apache's `HttpAsyncClient` into a `Future`-based API

“Mistakes were made”

Typical “gotchas” when using `scala.concurrent.Future`:

- unnecessarily waiting for futures to complete – blocked threads
- forgetting to wait for futures to complete – race conditions
- expecting to see a stack trace from exceptions inside a `Future`
- using one thread pool for everything – thread starvation
- forgetting to shut down the thread pool – application never quits

What is “thread-safe”

Thread-safe methods:

- work correctly even if called from many threads in parallel
- either have no mutable state, or manage it with great care

Examples of thread-safe methods:

- `AtomicInteger.incrementAndGet()`
- `ConcurrentLinkedQueue.add()`

Example of non-thread-safe method:

- `MockitoSugar.mock[T]()` – deadlocks when called in parallel

What is "non-blocking"

Non-blocking methods:

- return quickly, although they may *schedule* long-running calculations
- do not perform an idle wait
  - `Thread.sleep()` or `Await.result()`
- do not perform a busy wait
  - `while( ! isReady() ) { doNothingLoop() }`
- do not perform slow I/O (e.g. HTTP with high latency)

Thread pools perform best when most tasks are non-blocking calls

Blocked threads cause suboptimal CPU utilization, a.k.a. "slowness"

Can we avoid "blocking"?

We could avoid blocking if arriving events could wake up our threads... but:

- A thread cannot be "woken up" if it isn't already blocked ("sleeping")
- Someone, somewhere has to keep a blocked thread waiting for events
- However, it does not have to be within *our* thread pools!

If we never `Await.result()`, we would never get any non-`Future` values...

- With the right libraries (Akka, etc.), our code never needs to do `Await.result()`
- (except in some unit tests)

How and when to convert "blocking" to "non-blocking"

Given a 3rd party blocking call `doHttpWork()`, our options are:

- Wrap it in a `Future` using the `scala.concurrent.blocking()` instruction
- Create a thread pool dedicated to scheduling `doHttpWork()` tasks

It's OK to block if we are not within concurrent code

Summary

- The backbone of Java concurrency: thread pool executors
- How and when do `Futures` run?
- What does it mean to be "thread safe" and "nonblocking", and when do we need that?
- Some typical "gotchas" when using `Futures` in the real world
- Converting other async APIs to `Futures` and back