Futures and Async: When to Use Which?

Philipp Haller
@philippkhallер
The leading forum for research & development related to the Scala programming language.
Overview

• A brief guide to the Future of Scala
• What is a Promise good for?
• What is Async?
• Guidelines on when to use which
Future

Creation

object Future {
    // [use case]
    def apply[T](body: => T): Future[T]
    // ..
}

Example

val fstGoodDeal =
    Future {
        usedCars.find(car => isGoodDeal(car))
    }
Future

Type

trait Future[+T] extends Awaitable[T] {
  // [use case]
  def map[S](f: T => S): Future[S]
  // [use case]
  def flatMap[S](f: T => Future[S]): Future[S]
  // ..
}

Example

val fstGoodDeal: Future[Option[Car]] = ..
val fstPrice: Future[Int] =
  fstGoodDeal.map(opt => opt.get.price)
val lowestPrice: Future[Int] =
  fstPrice.flatMap { p1 =>
    sndPrice.map { p2 =>
      math.min(p1, p2)
    }
  }
Collections of Futures

val goodDeals: List[Future[Option[Car]]] = ..

val bestDeal: Future[Option[Car]] = 
  Future.sequence(goodDeals).map(
    deals => deals.sorted.head
  )
Promise

Main purpose: create futures for non-lexically-scoped asynchronous code

Example

Function for creating a Future that is completed with value after delay milliseconds

def after[T](delay: Long, value: T): Future[T]
"after", Version 1

def after1[T](delay: Long, value: T) = Future {
    Thread.sleep(delay)
    value
}
"after", Version 1

How does it behave?

assert(Runtime.getRuntime()
    .availableProcessors() == 8)

for ( _ <- 1 to 8 ) yield
    after1(1000, true)

val later = after1(1000, true)

Quiz: when is “later” completed?

Answer: after either ~1 s or ~2 s (most often)
object Promise {
    def apply[T](): Promise[T]
}

trait Promise[T] {
    def success(value: T): this.type
    def failure(cause: Throwable): this.type

    def future: Future[T]
}
“after”, Version 2

def after2[T](delay: Long, value: T) = {
    val promise = Promise[T]()

    timer.schedule(new TimerTask {
        def run(): Unit = promise.success(value)
    }, delay)

    promise.future
}

Much better behaved!
Managing Blocking

What if there is no asynchronous variant of a required API?

```
import scala.concurrent.blocking

def after3[T](delay: Long, value: T) = 
  Future {
    blocking {
      Thread.sleep(delay)
    }
    value
  }
```
public static interface ManagedBlocker {
    boolean block() throws InterruptedException;
    boolean isReleasable();
}
What is Async?

- New Scala module
  - "org.scala-lang.modules" %% " scala-async"
- Purpose: *simplify non-blocking concurrency*
- SIP-22 (June 2013)
- Releases for Scala 2.10 and 2.11
What Async Provides

• Future and Promise provide **types** and operations for managing **data flow**

• There is very little support for control flow
  
  • For-comprehensions, ..?

• Async complements Future and Promise with new constructs to manage **control flow**
Programming Model

Basis: **suspendible computations**

- **async { ... }** — *delimit* suspendible computation
- **await(obj)** — *suspend* computation until an event is signaled to *obj*

Example: Future
object Async {
    // [use case]
    def async[T](body: => T): Future[T]

    def await[T](future: Future[T]): T
}

Async
Example

val fstGoodDeal: Future[Option[Car]] = ..
val sndGoodDeal: Future[Option[Car]] = ..

val goodCar = async {
  val car1 = await(fstGoodDeal).get
  val car2 = await(sndGoodDeal).get
  if (car1.price < car2.price) car1 else car2
}

Guidelines
Item #1

**Use async/await instead of (deeply-)nested map/flatMap calls**

```scala
val goodCar = fstGoodDeal.flatMap { fstDeal =>
  val car1 = fstDeal.get
  sndGoodDeal.map { sndDeal =>
    val car2 = sndDeal.get
    if (car1.price < car2.price) car1
    else car2
  }
}
```

BAD!
Item #1

*Use async/await instead of (deeply-)nested map/flapMap calls*

```scala
val goodCar = async {
  val car1 = await(fstGoodDeal).get
  val car2 = await(sndGoodDeal).get
  if (car1.price < car2.price) car1 else car2
}
```
Item #2

**Use async/await instead of complex for-comprehensions**

```scala
def nameOfMonth(num: Int): Future[String] = ...
val date = "\d+/(\d+)".r

for { doyResponse <- futureDOY
  dayOfYear = doyResponse.body
  response <- dayOfYear match {
    case date(month, day) =>
      for (name <- nameOfMonth(month.toInt))
        yield Ok(s"It’s $name!")
    case _ =>
      Future.successful(NotFound("Not a..."))
  }
} yield response
```
Item #2

**Use async/await instead of complex for-comprehensions**

```scala
def nameOfMonth(num: Int): Future[String] = ...

val date = "/(\d+)/(\d+)".r

async {
    await(futureDOY).body match {
        case date(month, day) =>
            Ok(s"It’s ${await(nameOfMonth(month.toInt))}!")
        case _ =>
            NotFound("Not a date, mate!")
    }
}
```
Item #3

Use combinators for collections of futures instead of async/await and imperative while-loops

val futures = ..

async {
  var results = List[T]()
  var i = 0
  while (i < futures.size) {
    results = results :+: await(futures(i))
    i += 1
  }
  results
}
Item #3

*Use combinators for collections of futures instead of async/await and imperative while-loops*

```scala
val futures = ..

Future.sequence(futures)
```
Item #4

Do not accidentally sequentialize futures

```scala
for {
    x <- Future { .. }
    y <- Future { .. }
} yield {
    ..
}
```

BAD!
Item #4

Do not accidentally sequentialize futures

```scala
futX = Future { .. }
futY = Future { .. }

async {
    val x = await(futX)
    val y = await(futY)
    ..
}
```
Item #5

*Use Async to improve performance*

- Async *reduces closure allocations* compared to code using higher-order functions like map, flatMap, etc.

- Async *reduces boxing* of primitive values in some cases
Use Async because of future extensions

Since Async is a macro library, we will be able to do useful rewritings in the future:

- **Automatic parallelization** of Future-returning calls if no dependencies

- Optionally configure await calls to be blocking to maintain intact thread stack
Conclusion

- Focus on Future, not Promise
- Use Promise only when necessary
- Async is there to simplify Future-based code
- Async is production-ready