Scala Future API

Mikhail Mutcianko, Alexey Otts
СПБгУ, СП

Future - что это?

Модель данных, описывающая три состояния процесса

- Незавершенный
- Завершенный успешно
- Завершенный с ошибкой

Future - основные методы

```
def Future.apply(f: => A): Future[A]
```

➔ принимает блок кода

➔ начинает выполнение жадно (в момент вызова)

```
def onComplete(cb: Try[A] => B): Unit
```

➔ Выполнение функции после завершения процесса

Future - основные методы

```
def Future.apply(f: => A)(implicit ec: ExecutionContext): Future[A]
```

➔ принимает блок кода

➔ начинает выполнение жадно (в момент вызова)

```
def onComplete(cb: Try[A] => B)(implicit ec: ExecutionContext): Unit
```

➔ Выполнение функции после завершения процесса

Future - Execution Context

- интерфейс инструмента для выполнения задач
- может основываться на пуле потоков, и распределять задачи между ними
  - java.util.concurrent.ForkJoinPool
- готовый экземпляр:
  - scala.concurrent.ExecutionContext.Implicits.global

Future - блокировки

Блокировка снаружи:

```
Await.result(future, duration): A
Await.ready(future, duration): future.type
```

Выполнение блокирующих методов внутри Future:

```
Future { blocking { action() } }
```

Future - последовательные комбинаторы

```
def map[B](f: A => B): Future[B]
def flatMap[B](f: A => Future[B]): Future[B]
def withFilter(f: A => Boolean): Future[A]
```

- позволяют писать for comprehension

Future - for comprehension

```
def m1(): Future[Int] = Future { 1 }
def m2(): Future[Int] = Future { 2 }
def m3(arg: Int): Future[Boolean] = Future { arg > 0 }


def action(): Future[Boolean] = for {
 a <- m1()
 b <- m2()
 res <- m3(a + b)
} yield !res
```

Future - for comprehension

```
def m1(): Future[Int] = Future { 1 }
def m2(): Future[Int] = Future { 2 }
def m3(arg: Int): Future[Boolean] = Future { arg > 0 }

def action(): Future[Boolean] = {
 val aF: Future[Int] = m1()
 val bF: Future[Int] = m2()
 for {
   a <- aF()
   b <- bF()
   res <- m3(a + b)
 } yield !res
}
```

Future - конкурентные комбинаторы

```
def Future.traverse(in: M[B])(f: A => Future[B]): Future[M[B]]
def Future.sequence(in: M[Future[B]]): Future[M[B]]


def zip(other: Future[B]): Future[(A, B)]
```

Future - for comprehension

```
def m1(): Future[Int] = Future { 1 }
def m2(): Future[Int] = Future { 2 }
def m3(arg: Int): Future[Boolean] = Future { arg > 0 }

def action(): Future[Boolean] = for {
  (a, b) <- m1() zip m2()
  res <- m3(a + b)
} yield !res
```

Future - обработка ошибок

```
type PF[+A] = PartialFunction[A]

def recover(pf: PF[Throwable, A]): Future[A]
def recoverWith[B](pf: PF[Throwable, Future[A]]): Future[A]
def transform[B](f: Try[A] => Try[B]): Future[B]
def transformWith[B](f: Try[A] => Future[B]): Future[B]
def fallbackTo(f: Future[A]): Future[A]
```

Future - побочные эффекты

```
def andThen[B](pf: PF[Try[A], B]): Future[A]
```

Future - Promise

API для получения Future с управляемым состоянием:

future - метод получения Future

Методы управления состоянием:

complete, completeWith
tryComplete, tryCompleteWith
success, trySuccess, failure, tryFailure

Future - Promise

```
val p = Promise[Int]()
val f = p.future

Future { p.trySuccess(1) }
Future { p.trySuccess(2) }

f.map(print) // может быть 1 или 2
```