Futures. Reactive Streams

Вадим Цесько

ТЕХНОПОЛИС

одноклассники

Futures and Promises

David Ross

So a Promise can be broken, but you can't change the Future. Love the metaphors.

Зачем

- `Future` возникают, когда появляется асинхронный IO
- Т. е. повсеместно

Происхождение

- Henry Baker and Carl Hewitt. The Incremental Garbage Collection of Processes[1]. 1977.

Future

Структура данных для **синхронного** (блокирующегося) или **асинхронного** (неблокирующегося) извлечения результата параллельной операции.

[1] https://en.wikipedia.org/wiki/Futures_and_promises

Реализации

- Java (`j.u.c.CompletableFuture`)
- Scala[2] (**Akka**[3])
- C#
- Python
- Haskell
- Clojure
- Node.js
- ...

[2] `https://docs.scala-lang.org/overviews/core/futures.html`

[3] `https://doc.akka.io/docs/akka/current/futures.html`

Future

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

```scala
val future = Future {
  "Hello" + "World"
}
val result = Await.result(future, 1 second)
```

```scala
val future = Future.successful("Yay!")
val otherFuture =
  Future.failed[String](
      new IllegalArgumentException("Bang!"))
```

Promise

```
1  import scala.concurrent.{Future, Promise}
2  import scala.concurrent.ExecutionContext.Implicits.global
3
4  val p = Promise[T]()
5  val f = p.future
6  val producer = Future {
7    val r = produceSomething()
8    p success r
9    continueDoingSomethingUnrelated()
10 }
11 val consumer = Future {
12   startDoingSomething()
13   f onSuccess {
14     case r => doSomethingWithResult()
15   }
```

Futures and Actors

Общение с акторами вне Actor System:

```
1 implicit val timeout = Timeout(5 seconds)
2 val future = actor ? msg
3 val result =
4   Await.result(future, timeout.duration)
5     .asInstanceOf[String]
```

Или так:

```
1 val future: Future[String] = ask(actor, msg).mapTo[String]
```

И обратно:

```
1 import akka.pattern.pipe
2 future pipeTo actor
```

map (1)

```
1  val f1 = Future {
2    "Hello" + "World"
3  }
4
5  val f2 = f1 map { x =>
6    x.length
7  }
8
9  val result = Await.result(f2, 1 second)
10
11 result must be(10)
12
13 f1.value must be(Some(Success("HelloWorld")))
```

map (2)

```scala
val f1 = Future {
  "Hello" + "World"
}
val f2 = Future.successful(3)
val f3 = f1 map { x =>
  f2 map { y =>
    x.length * y
  }
}
```

Тип `f3`?

- `Future[Future[Int]]`
- А хотели: `Future[Int]`

flatMap

```
1  val f1 = Future {
2    "Hello" + "World"
3  }
4  val f2 = Future.successful(3)
5  val f3 = f1 flatMap { x =>
6    f2 map { y =>
7      x.length * y
8    }
9  }
10
11 val result = Await.result(f3, 1 second)
12
13 result must be(30)
```

filter

```
1  val future1 = Future.successful(4)
2  val future2 = future1.filter(_ % 2 == 0)
3  val result = Await.result(future2, 1 second)
4  result must be(4)
5
6  val failedFilter = future1.filter(_ % 2 == 1).recover {
7    case _: NoSuchElementException => 0
8  }
9
10 val result2 = Await.result(failedFilter, 1 second)
11 result2 must be(0)
```

Await.result()

```
1 val f1 = ask(actor1, msg1)
2 val f2 = ask(actor2, msg2)
3
4 val a = Await.result(f1, 1 second).asInstanceOf[Int]
5 val b = Await.result(f2, 1 second).asInstanceOf[Int]
6
7 val f3 = ask(actor3, (a + b))
8
9 val result = Await.result(f3, 1 second).asInstanceOf[Int]
```

Блокировки

Каждый вызов `Await.result()` — блокировка.

For-comprehension

```
1  val f1 = ask(actor1, msg1)
2  val f2 = ask(actor2, msg2)
3
4  val f3 = for {
5    a <- f1.mapTo[Int]
6    b <- f2.mapTo[Int]
7    c <- ask(actor3, (a + b)).mapTo[Int]
8  } yield c
9
10 val result = Await.result(f3, 1 second)
```

Блокировки

Уже лучше, но усложняется с ростом числа акторов.

Комбинаторы

```
Future.sequence:
  M[Future[A]] => Future[M[A]]
```

```
Future.traverse:
  M[A] => (A => Future[B]) => Future[M[B]]
```

```
Future.fold:
  Traversable[Future[T]] => R => ((R, T) => R) => Future[R]
Future.reduce[T, R >: T]:
  Traversable[Future[T]] => ((R, T) => R) => Future[R]
```

Базовые

```
future onSuccess {
  case "bar"    => println("Got my bar alright!")
  case x: String => println("Got some random string: " + x)
}
future onFailure {
  case ise: IllegalStateException
    if ise.getMessage == "OHNOES" =>
      // OHNOES! We are in deep trouble, do something!
  case e: Exception =>
    // Do something else
}
future onComplete {
  case Success(result) => doSomethingOnSuccess(result)
  case Failure(error) => doSomethingOnFailure(error)
}
```

Порядок

Исполнение Callback'ов

Обработчики исполняются в **неопределенном порядке** и/или **параллельно**.

```
1 val result = Future { loadPage(url) } andThen {
2   case Left(exception) => log(exception)
3 } andThen {
4   case _ => watchSomeTV
5 }
```

Полезное

Future.fallbackTo():

```
1 val future4 = future1 fallbackTo future2 fallbackTo future3
```

Future.zip():

```
1 val future3 =
2   future1 zip future2 map { case (a, b) => a + " " + b }
```

Exceptions

Исключения доставляются **вместо** результата.

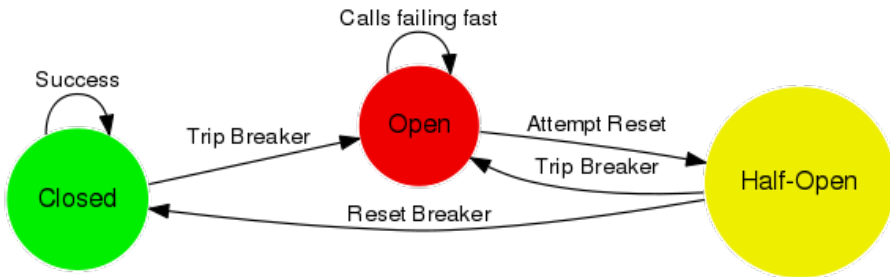
Но можно делать так:

```
1 val future = akka.pattern.ask(actor, msg1) recover {
2   case e: ArithmeticException => 0
3 }
```

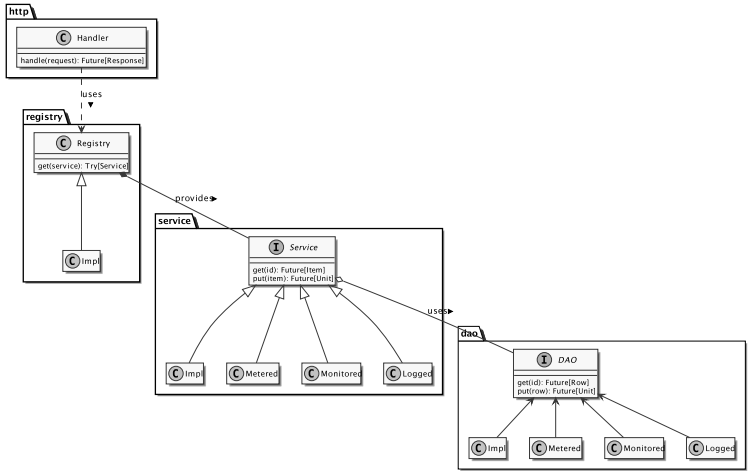
Или так:

```
1 val future = akka.pattern.ask(actor, msg1) recoverWith {
2   case e: ArithmeticException =>
3     Future.successful(0)
4   case foo: IllegalArgumentException =>
5     Future.failed[Int](
6         new IllegalStateException("All br0ken!"))
7 }
```

Circuit Breaker: States



Типичный CRUD Web-сервис



Внимание

Проблема

Можно получить OOM из-за неограниченного накопления Future (FJP).

Решение

- Throttling на верхнем уровне
- Но легко пропустить callback и перестать отвечать на запросы

Reactive Streams[4]

Ever-newer waters flow on those who step into the same rivers.

Heraclitus

- Real-time источники
- Массовое копирование
- Мониторинг/аналитика

[4] Roland Kuhn. Reactive Streams: Handling Data-Flows the Reactive Way: https://www.infoq.com/presentations/reactive-steams-akka/

Проблема

Back Pressure

Consumer не справляется с потоком от producer.

Демаркация:

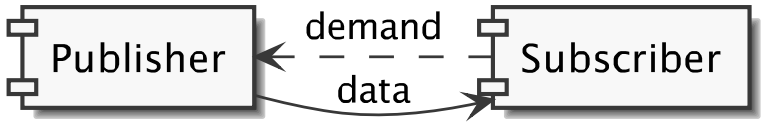
- Сервера
- Приложения
- Потоки
- Акторы
- ...

Традиционные решения

- Pull: блокирующие вызовы
- Push: буферизация и/или отбрасывание
- **Reactive way**[5]

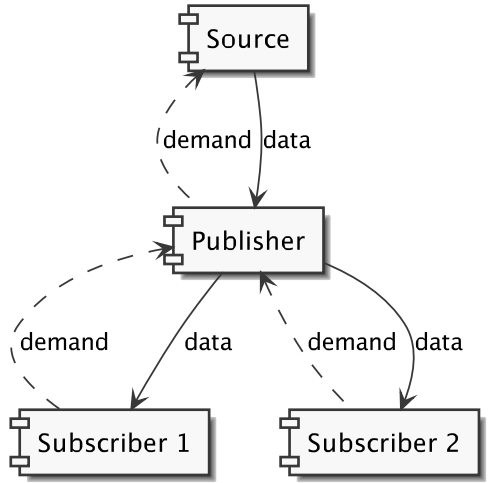
[5] https://www.reactive-streams.org

Основная идея

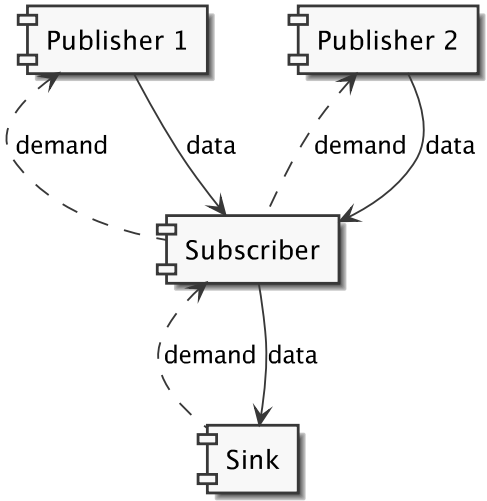


- Получатель контролирует скорость потока
- Потребление памяти зависит от demand
- Consumer быстрее — push
- Producer быстрее — pull
- Все вызовы неблокирующие

Split Data = Merge Demand



Merge Data = Split Demand



Секрет успеха

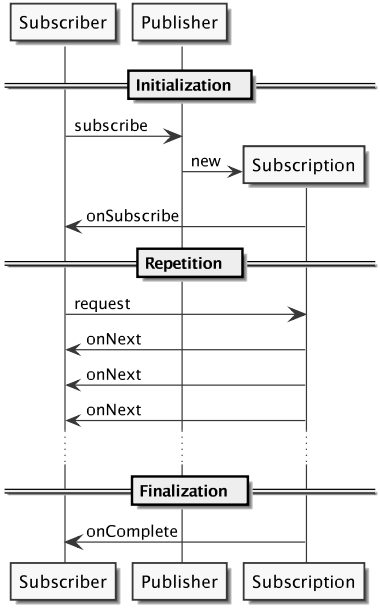
- Минимальные интерфейсы
- Строгая спецификация семантики[6]
- TCK
- Свобода для идиоматичных API

[6] https://www.reactive-streams.org/

```java
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}

public interface Subscription {
    void request(long n);
    void cancel();
}
```


Subscriber
Publisher
Initialization
subscribe
new
Subscription
onSubscribe
Repetition
request
onNext
onNext
onNext
Finalization
onComplete
Subscriber
Publisher
Subscription

Реализации

- RxJava
- Project Reactor
- Vert.x
- Akka Streams
- Slick
- **Java 9**
- etc.[7]

[7] https://en.wikipedia.org/wiki/Reactive_Streams

Ссылки

- www.reactive-streams.org
- github.com/reactive-streams/reactive-streams-jvm
- JEP 266[8] in Java 9 by Doug Lea
- Site Reliability Engineering (Антон Иванов)[9]
- Реактивный раздатчик ok.ru/music[10]

[8] https://openjdk.java.net/jeps/266

[9] https://ok.ru/video/1379739765425

[10] https://incubos.org/posts/2018/10/19/ok-music/

Conclusion

- Параллельные программы — это **неизбежно**
- Нужны простые способы создания **корректных** параллельных систем
- Существуют **альтернативы** низкоуровневым примитивам[11]:
  - Message-Passing Concurrency (Actor Model)
  - Futures
  - Reactive Streams

[11] ScalaDays 2013. Concurrency — The good, the bad, the ugly: `https://www.youtube.com/watch?v=0JfS7K-Vkgk`

Вопросы?

Вадим Цесько
Одноклассники
https://tt.ru/incubos
https://t.ru/incubos
mail@incubos.org