AKKA

동시성 Actor 모델의 현대적 구현체

- Actor 모델 사용의 이점

Scala 기본 문법을 안다는 전제하에 실용적인 측면으로 기술.

Akka Cookbook을 기반으로 서술.

Framework, CQRS, Concurrency

기초

Reactive System

- Resilient$_{회복력}$: 오류/예외 같은 장애가 발생하더라도 항상 응답하여 회복
- Elastic$_{탄력성}$: Traffic 크기에 상관없이 할당된 자원을 조절해 부하를 서비스할 수 있게 항상 응답성을 유지
- Message Driven$_{메시지 기반}$: 시스템의 구성 요소가 서로 **느슨하게 결합**되어 있으며 **비동기** 메시지 전달을 사용해 통신하고, 행동을 취함으로써 메시지에 반응.
- Responsive$_{응답성}$: 위 세 속성을 만족하는 시스템을 "응답성이 있는 시스템"이라 한다.

Actor의 속성

- State : Actor는 내부 상태가 있고, 이 상태는 Message를 처리함에 따라 순차적으로 변한다. (mutable)
- Behavior : Actor는 송신된 Message에 행위를 적용함으로써 반응한다.
- Communication : Actor는 다른 Actor에게 Message를 송신하거나 수신함으로써 서로 통신한다.
- Mailbox : Mailbox는 Actor가 메시지를 가져오고 처리하는 **Message Queue**다.

기본 Actor System Project 생성

https://doc.akka.io/docs/akka/current/actors.html?language=scala

환경 설정

- JDK 1.8 이상 설치

2018/06/09 14:37 · ledyx

Project 생성

아래 과정은 console에서 수행할 경우 필요한 작업. Intellij로 sbt 프로젝트 생성하면 더 쉽게 만들 수 있다.

1. Project를 생성할 Directory 생성
2. 생성한 Directory로 이동해서 “sbt” 실행
3. sbt 콘솔에서 아래 과정을 수행하면 “build.sbt” Build 설정 파일과 Class 파일이 놓일 “target” Directory가 생성된다.

```
set name := "hello-akka"
set version := "0.1"
set scalaVersion := "2.12.6"
session save
exit
```

1. build.sbt를 열고 아래 내용을 추가한다.

```
// https://mvnrepository.com/artifact/com.typesafe.akka/akka-actor
libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.5.14"
```

1. src/main/scala/ 밑에 프로젝트 이름의 패키지와 클래스를 작성.

```
package com.practice.ex01

import akka.actor.ActorSystem

object HelloAkkaActorSystem extends App{
  val actorSystem = ActorSystem("HelloAkka")
  println(actorSystem)
  // akka://HelloAkka
}
```

1. console에서

```
sbt "runMain com.practice.ex01.HelloAkkaActorSystem"
```

기본적인 **Ask-Tell 예제**

scala에서 ask는 “?”, tell은 “!” method를 사용한다.

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
```

```
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.Await
import scala.concurrent.duration._

import akka.actor.Actor

object HelloAkkaActorSystem extends App {
  val actorSystem = ActorSystem("HelloAkka")
  val actor = actorSystem.actorOf(Props(classOf[SumActor]), "sumActor")


  // ask를 사용하려면 timeout을 반드시 명시해야 함.
  implicit val timeout = Timeout(10 seconds)
  val future = actor ? (1 to 10).toArray
  val result = Await.result(future, 10 seconds)
  println(s"result : $result")
}

class SumActor extends Actor {
  override def receive: Receive = {
    case arr: Array[Int] => sender ! arr.sum
    case _ => sender ! "Error!"
  }
}
```

Actor간 통신 (tell을 이용한 결과 수신)

ask는 기본적으로 blocking. 그러므로 비슷하게 tell을 이용하여 서로 주고 받는 callback 형식으로 구현하므로 더 경제적.

1. QueryActor에서 Start로 연산 요청. 연산할 Actor인 SumActor와 값을 전달 받음.
2. SumActor에서 전달 받은 값을 연산하고 sender(QueryActor)의 Complete로 전달
3. QueryActor의 Complete에서 연산 결과 수신.

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
import com.practice.ex01.Message.Start

object HelloAkkaActorSystem extends App {
  val actorSystem = ActorSystem("HelloAkka")

  val queryActor = actorSystem.actorOf(Props(classOf[QueryActor]),
"queryActor")
  val sumActor = actorSystem.actorOf(Props(classOf[SumActor]), "sumActor")

  queryActor ! Start(sumActor, (1 to 10).toArray)
```

```
}
```

```
package com.practice.ex01

import akka.actor.ActorRef

object Message {
  case class Start(actorRef: ActorRef, arr: Array[Int])
  case class Progress(arr: Array[Int])
  case class Complete(result: Int)
}
```

```
package com.practice.ex01

import akka.actor.Actor
import com.practice.ex01.Message._

class QueryActor extends Actor{
  override def receive: Receive = {
    case Start(actorRef, arr) =>
      println("Start!")
      actorRef ! Progress(arr)

    case Complete(result) =>
      println(s"Complete! $result")

    case _ => println("QueryActor - WTHIGO?")
  }
}
```

```
package com.practice.ex01

import akka.actor.Actor
import com.practice.ex01.Message._

class SumActor extends Actor {
  override def receive: Receive = {
    case Progress(arr) =>
      println("Progress!")
      sender ! Complete(arr.sum)
    case _ => sender ! "SumActor - WTHIGO?"
  }
```

```
}
```

사용자 정의 Mailbox 만들기

Actor가 메시지를 가져오는 방법을 제어할 때 사용.

Mailbox 정의

```
package com.practice.ex01

import akka.actor.{ActorRef, ActorSystem}
import akka.dispatch.{MailboxType, MessageQueue, ProducesMessageQueue}
import com.typesafe.config.Config

class MyCustomMailbox extends MailboxType with
ProducesMessageQueue[MyMessageQueue] {
  def this(setting: ActorSystem.Settings, config: Config) = this()
  override def create(owner: Option[ActorRef], system: Option[ActorSystem]):
MessageQueue = {
    println("MyCustomMailbox is created!")
    new MyMessageQueue()
  }
}
```

```
package com.practice.ex01

import java.util.concurrent.ConcurrentLinkedQueue

import akka.actor.ActorRef
import akka.dispatch.{Envelope, MessageQueue}

class MyMessageQueue extends MessageQueue{
  private final val queue = new ConcurrentLinkedQueue[Envelope]()

  override def enqueue(receiver: ActorRef, handle: Envelope): Unit = {
    println("enqueue!")

    if(handle.sender.path.name == "MyActor") {
      handle.sender ! "I know you!"
      queue.offer(handle)
    }
    else
      handle.sender ! "Who are you?"
```

```
  }

  override def dequeue(): Envelope = queue.poll

  override def numberOfMessages: Int = queue.size

  override def hasMessages: Boolean = !queue.isEmpty

  override def cleanUp(owner: ActorRef, deadLetters: MessageQueue): Unit = {
    while (hasMessages) {
      deadLetters.enqueue(owner, dequeue())
    }
  }
}
```

```
custom-dispatcher {
  mailbox-requirement = "com.practice.ex01.MyMessageQueue"
}
akka.actor.mailbox.requirements {
  "com.practice.ex01.MyMessageQueue" = custom-dispatcher-mailbox
}
custom-dispatcher-mailbox {
  mailbox-type = "com.practice.ex01.MyCustomMailbox"
}
```

Actor 정의

```
package com.practice.ex01

import akka.actor.Actor

class MySpecialActor extends Actor {
  override def receive: Receive = {
    case msg: String => println(s"MySpecialActor's msg : $msg")
  }
}
```

```
package com.practice.ex01

import akka.actor.{Actor, ActorRef}

class MyActor extends Actor {
```

```
  override def receive: Receive = {
    case (msg: String, actorRef: ActorRef) => actorRef ! msg
    case msg => println(s"MyActor's msg? $msg")
  }
}
```

Client

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}

object Client extends App {
  val actorSystem = ActorSystem("HelloAkka")

  // application.conf에 기술된 custom-dispatcher을 읽고
 // 사용자 정의 Mailbox(MyCustomMailBox)가 먼저 생성되고,
 // 그 안에 create() method로 message queue가 생성된다.
  val mySpecialActor = actorSystem.actorOf(Props[MySpecialActor]
                       .withDispatcher("custom-dispatcher"))

  val unknownActor = actorSystem.actorOf(Props[MyActor], "abracadabra")
  val knownActor = actorSystem.actorOf(Props[MyActor], "MyActor")

  unknownActor ! ("hello", mySpecialActor)
  knownActor ! ("hello", mySpecialActor)
 // 비동기적이므로 아래 결과는 바뀔 수 있음.
  /*
  MyCustomMailbox is created!
  enqueue!
  enqueue!
  MyActor's msg? Who are you?
  MyActor's msg? I know you!
  MySpecialActor's msg : hello
  */
}
```

수신 메시지의 우선순위 정의

우선순위에 따라 특정 메시지를 먼저 처리하고 싶을 때 사용. 설정은 위와 비슷하므로 상세한 설명은 생략.

Mailbox 정의

```
package com.practice.ex01

import akka.actor.ActorSystem
import akka.dispatch.{PriorityGenerator, UnboundedPriorityMailbox}
import com.typesafe.config.Config

class MyPriorityMailbox(settings: ActorSystem.Settings, config: Config)
  extends UnboundedPriorityMailbox (
  PriorityGenerator {
    // int 값이 작을수록 우선순위가 높다.

    case x: Int => 1
    case x: String => 0
    case _ => 3
  }
)
```

```
my-priority-mailbox {
  mailbox-type = "com.practice.ex01.MyPriorityMailbox"
}
```

Actor 정의

```
package com.practice.ex01

import akka.actor.Actor

class MyPriorityActor extends Actor{
  override def receive: Receive = {
    case x: Int => println(x)
    case x: String => println(x)
    case x => println(x)
  }
}
```

Client

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
```

```
object Client extends App {
  val actorSystem = ActorSystem("HelloAkka")

  val myPriorityActor = actorSystem.actorOf(Props[MyPriorityActor]
                        .withDispatcher("my-priority-mailbox"))

  Array('LowestPriority, 123, "Top Priority!", 1.23).foreach(myPriorityActor
! _)
 /* 결과적으로 String, int 순으로 우선순위가 높음.*/
  /*
  Top Priority!
  123
  1.23
  'LowestPriority
  */
}
```

때와 상관없이 항상 우선권을 갖는 메일박스 만들기

항상 우선권을 가질 Message 정의

```
package com.practice.ex01

import akka.dispatch.ControlMessage

case object MyControlMessage extends ControlMessage
```

Mailbox 정의

```
control-aware-mailbox {
  mailbox-type = "akka.dispatch.UnboundedControlAwareMailbox"
}
```

Actor 정의

```
package com.practice.ex01

import akka.actor.Actor
```

```
class Logger extends Actor{
  override def receive: Receive = {
    case MyControlMessage => println("Control message first!")
    case x => println(x)
  }
}
```

Client

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}

object Client extends App {
  val actorSystem = ActorSystem("HelloAkka")

  val loggerActor = actorSystem.actorOf(Props[Logger]
                        .withDispatcher("control-aware-mailbox"))

  loggerActor ! "Hello,"
  loggerActor ! "world!"
  loggerActor ! MyControlMessage // 우선적으로 처리

  /*
  Control message first!
  Hello,
  world!
  */
}
```

Runtime중에 Actor 행위 바꾸기

become/unbecome으로 구현.

Message 정의

```
package com.practice.ex01

object StateMessage {
  case class IntState()
  case class StringState()
}
```

Actor 정의

```
package com.practice.ex01

import akka.actor.Actor
import com.practice.ex01.StateMessage.{IntState, StringState}

class StateChangingActor extends Actor {
  override def receive: Receive = {
    case StringState => context.become(isStateString)
    case IntState => context.become(isStateInt)
  }

  def isStateString: Receive = {
    case msg : String => println(s"$msg")
    case IntState => context.become(isStateInt)
  }

  def isStateInt: Receive = {
    case msg : Int => println(s"$msg")
    case StringState => context.become(isStateString)
  }
}
```

Client

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
import com.practice.ex01.StateMessage.{IntState, StringState}

object Client extends App {
  val actorSystem = ActorSystem("HelloAkka")

  val stateChangingActor = actorSystem.actorOf(Props[StateChangingActor])

  stateChangingActor ! StringState
  stateChangingActor ! "Hello, world!"

  stateChangingActor ! IntState
  stateChangingActor ! 123
```

```
  stateChangingActor ! "abracadabra" // 현재 state가 IntState이기 때문에 무시됨.

  /*
  Hello, world!
  123
  */
}
```

Actor 중지시키기

https://stackoverflow.com/questions/13847963/akka-kill-vs-stop-vs-poison-pill

- PoisonPill : Mailbox에 대기중인 메시지를 모두 처리 정지
- Kill : ActorKilledException를 발생시킴. 이에 대한 구체적인 동작은 supervisor mechanism을 따른다. 기본값은 Actor를 중지. 그리고 Mailbox가 유지되므로 실패를 발생시킨 메시지를 제외한 나머지 메시지는 그대로 남는다.
- context.stop(actorRef) : Actor를 순차적으로 정지시킬때 사용. Bottom-up(자식, 부모, 액터 시스템) 순서로 정지. 현재 처리중인 Message 완료후 다른 메시지는 모두 무시하고 정지시킨다.

```
package com.practice.ex01

import akka.actor.Actor

class ShutdownActor extends Actor {
  override def receive: Receive = {
    case msg: String => println(s"$msg")
    case Stop => context.stop(self)
  }
}
```

```
package com.practice.ex01

import akka.actor.{ActorSystem, Kill, PoisonPill, Props}

object Client extends App {
  val actorSystem = ActorSystem("HelloAkka")

  val shutdownActor1 = actorSystem.actorOf(Props[ShutdownActor])
  shutdownActor1 ! "hello1"
  shutdownActor1 ! PoisonPill
  shutdownActor1 ! "Are you breathing?"

  val shutdownActor2 = actorSystem.actorOf(Props[ShutdownActor])
  shutdownActor2 ! "hello2"
  shutdownActor2 ! Kill
```

```
  shutdownActor2 ! "Are you breathing?"

  val shutdownActor3 = actorSystem.actorOf(Props[ShutdownActor])
  shutdownActor3 ! "hello3"
  shutdownActor3 ! Stop
  shutdownActor3 ! "Are you breathing?"
/*
hello2
hello1
hello3

scala> [INFO] [08/06/2018 00:25:30.860] [HelloAkka-akka.actor.default-
dispatcher-2] [akka://HelloAkka/user/$c] Message [java.lang.String] without
sender to Actor[akka://HelloAkka/user/$c#303943664] was not delivered. [1]
dead letters encountered. If this is not an expected behavior, then
[Actor[akka://HelloAkka/user/$c#303943664]] may have terminated
unexpectedly, This logging can be turned off or adjusted with configuration
settings 'akka.log-dead-letters' and 'akka.log-dead-letters-during-
shutdown'.
[INFO] [08/06/2018 00:25:30.861] [HelloAkka-akka.actor.default-dispatcher-2]
[akka://HelloAkka/user/$a] Message [java.lang.String] without sender to
Actor[akka://HelloAkka/user/$a#-1903145472] was not delivered. [2] dead
letters encountered. If this is not an expected behavior, then
[Actor[akka://HelloAkka/user/$a#-1903145472]] may have terminated
unexpectedly, This logging can be turned off or adjusted with configuration
settings 'akka.log-dead-letters' and 'akka.log-dead-letters-during-
shutdown'.
[ERROR] [08/06/2018 00:25:30.862] [HelloAkka-akka.actor.default-
dispatcher-5] [akka://HelloAkka/user/$b] Kill
(akka.actor.ActorKilledException: Kill)
[INFO] [08/06/2018 00:25:30.863] [HelloAkka-akka.actor.default-dispatcher-2]
[akka://HelloAkka/user/$b] Message [java.lang.String] without sender to
Actor[akka://HelloAkka/user/$b#14529698] was not delivered. [3] dead letters
encountered. If this is not an expected behavior, then
[Actor[akka://HelloAkka/user/$b#14529698]] may have terminated unexpectedly,
This logging can be turned off or adjusted with configuration settings
'akka.log-dead-letters' and 'akka.log-dead-letters-during-shutdown'.
*/
}
```

Fault tolerant system

장애 발생시 정지되는 것이 아닌 처리량이 감소한 가동으로 항상 반응성을 유지하며, 완전 가동중과 비교하여 정책적으로 더 혹은 덜 가동되는 시스템.

System design

염두할 사항

- 시스템을 컴포넌트로 나누기 : "어떤 기능에 대한 책임" 단위로 나눈다. 이 때, 연쇄적 장애 발생시 각 컴포넌트는 간섭이 없는 독립적인 형태로 서로 영향이 없어야 한다. 특히 Core 컴포넌트는 더더욱 장애 발생으로 부터 독립적이어야 한다.
- Backup : 장애 발생에 대비. 고가용성.

구축 방법

- Duplication : 컴포넌트 여러개 실행
- Replication : 같은 컴포넌트를 여러개 준비, 모두에게 요청 후 송신하고, 그 중 하나 선택
- Isolation : Multi process. 다른 컴포넌트의 실패에 영향을 받지 않게 하려는 목적
- Delegation : 장애 발생시 정상 실행중인 컴포넌트에게 넘기려는 목적.

Actor Life cycle

https://doc.akka.io/docs/akka/2.5/actors.html#actor-lifecycle

Supervison and Monitoring

https://doc.akka.io/docs/akka/2.5/general/supervision.html

Strategy

- OneForOneStrategy : Default. 고장난 자식 Actor만 Restart/Resume 혹은 상위 Actor에게 보고. 이는 각 자식 Actor들이 각각 영향을 주지 않은, 독립적일 때 유용.
- AllForOneStrategy : 한 Supervisor 액터 시스템 아래에 있는 모든 자식 Actor 전략 적용. 이는 특정 자식 Actor들이 종속적인 관계일 때 유용.

Monitoring

특정 서비스가 살아있는 지 지속적으로 확인할 때 필요.

```
// 감시
context.watch(childActor: ActorRef)

// 감시 해제
context.unwatch(childActor: ActorRef)
```

기본적인 예제

아래와 같은 Tree 구조로 운영한 뒤 Strategy, watch(감시, 모니터링)를 이용한다.

부모 액터의 자식 액터 생성

Master - Slave. 자식 액터의 장애를 부모가 처리하는 구조.

Message 정의

```
package com.practice.ex01

object Message {
  case object CreateChild
  case class Greet(msg: String)
}
```

자식, 부모 Actor 정의

```
package com.practice.ex01

import akka.actor.Actor
import com.practice.ex01.Message.Greet

class ChildActor extends Actor {
  override def receive: Receive = {
    case Greet(msg) => println(s"parent : ${self.path.parent} // me :
${self.path} // msg : ${msg}")
  }
}
```

```
package com.practice.ex01

import akka.actor.{Actor, Props}
import com.practice.ex01.Message.{CreateChild, Greet}

class ParentActor extends Actor{
  override def receive: Receive = {
    case CreateChild =>
      val child = context.actorOf(Props[ChildActor], "child")
      child ! Greet("Hello, child!")
```

```
  }
}
```

Client

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
import com.practice.ex01.Message.CreateChild

object Client extends App {
  val actorSystem = ActorSystem("Supervision")
  val parent = actorSystem.actorOf(Props[ParentActor], "parent")
  parent ! CreateChild
  // parent : akka://Supervision/user/parent // me :
akka://Supervision/user/parent/child // msg : Hello, child!
}
```

Routing

https://doc.akka.io/docs/akka/2.5/routing.html

언제 사용하는가?

- 같은 형태의 액터들 중에서 가장 부하가 적은 액터에 메시지를 보내고 싶을 때
- 한 액터에 메시지를 Round-robin으로, 즉, Loop로 모든 액터에 메시지를 하나씩 보내고 싶을 때
- 그룹 내 모든 액터에 하나의 메시지를 보내고 싶을 때
- 액터 사이의 작업을 어떤 메커니즘으로 재분배하고 싶을 때

Pool의 종류

SmallestMailboxPool

메시지 수가 가장 적은 액터에 메시지 전달. 즉, 가장 덜 바쁜 액터에게 메시지 전달.

```
class RoutingActor extends Actor {
  override def receive: Receive = {
    case msg: String => sender ! s"I am ${self.path.name}, I received $msg"
    case _ => println(s"I don't understand the message")
  }
}
```

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
import akka.routing.SmallestMailboxPool

object Client extends App {
  val actorSystem = ActorSystem("Supervision");
  val router =
actorSystem.actorOf(SmallestMailboxPool(5).props(Props[SmallestMailboxActor]
))

  1 to 5 foreach(i => router ! s"Hello $i")

/*
I am $a
I am $a
I am $d
I am $b
I am $c
*/
}
```

BalancingPool

액터 사이의 작업을 재분배.

BalancingPool vs SmallestMailboxPool

https://groups.google.com/forum/#!msg/akka-user/pymtrCZBduk/rwXKIdGUGvEJ
https://doc.akka.io/docs/akka/2.5/routing.html#balancing-pool

- BalancingPool은 각 라우팅되는 대상이 완전히 구분된 정체성을 가진 속성을 가지지 않는다. 대상들은 다름 이름들을 가지지만, 대부분 그 대상들은 결국 올바른 액터들에게 말하지 않는다.
    - 그렇기 때문에 BalacingPool은 Broadcast Messages에 사용하면 안된다.
- SmallestMailboxPool은 BancingPool과 다른 이점으로 메시지 큐 구현에 제한을 두지 않는다.
- SmallestMailboxPool이 더 낫다. 가장 덜 바쁜 액터에게 보내지는 것을 항상 알기 때문.

RoundRobinPool

각 액터에게 하나씩 메시지 전달.

RandomPool

말그대로 무작위.

BroadcastPool

하나의 같은 메시지를 모든 액터에게 전달. 모든 액터에게 같은 작업을 하도록 일반적인 명령을 보내고 싶을 때 사용.

Specially Handled Messages

https://doc.akka.io/docs/akka/2.5/routing.html#specially-handled-messages

- Broadcast
- PoisonPill
- Kill
- GetRoutees, Addroutee, RemoveRoutee, AdjustPoolSize

액터를 관리하는 데 사용.

ScatterGatherFirstCompletedPool

같은 라우터를 공유하는 액터들 중에게 같은 메시지를 모두 보내고(Broadcast), 가장 먼저 작업을 마친 액터를 **기다려 응답을 송신**(ask) 후 나머지 액터 응답을 버림. 여러 서버 중 가장 빠르게 반응하는 서버에 작업을 보내는 상황에서 사용.

```
package com.practice.ex01

import akka.actor.Actor

class RoutingActor extends Actor {
  override def receive: Receive = {
    case msg: String => sender ! s"I am ${self.path.name}, I received $msg"
    case _ => println(s"I don't understand the message")
  }
}
```

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
import akka.pattern.ask
import akka.routing.ScatterGatherFirstCompletedPool
import akka.util.Timeout

import scala.concurrent.duration._
import scala.concurrent.{Await, Future}
```

```
object Client extends App {
  val duration = 10 seconds
  implicit val timeout = Timeout(duration)

  val actorSystem = ActorSystem("Hello-Akka");
  val router = actorSystem
    .actorOf(ScatterGatherFirstCompletedPool(5, within =  duration)
      .props(Props[RoutingActor]))

  val future: Future[Any] = router ? "hello"
  val response: String = Await.result(future.mapTo[String], duration)

  println(response)
  // I am $a, I received hello
}
```

TailChoppingPool

무작위로 고른 액터에게 메시지를 보내고, interval에서 지정한 약간의 지연 후 남은 액터들 중에 무작위로 고른 액터에게 송신하는 일을 계속한다. 첫 번째 응답이 수신되기를 기다리고, 이를 본래 송신자에게 전달, 나머지 응답은 버림.

```
package com.practice.ex01

import akka.actor.Actor

class RoutingActor extends Actor {
  override def receive: Receive = {
    case msg: String => sender ! s"I am ${self.path.name}, I received $msg"
    case _ => println(s"I don't understand the message")
  }
}
```

```
package com.practice.ex01

import akka.actor.{ActorSystem, Props}
import akka.pattern.ask
import akka.routing.TailChoppingPool
import akka.util.Timeout

import scala.concurrent.duration._
import scala.concurrent.{Await, Future}

object Client extends App {
  val duration = 10 seconds
```

```
  implicit val timeout = Timeout(duration)

  val actorSystem = ActorSystem("Hello-Akka");
  val router = actorSystem
    .actorOf(TailChoppingPool(5, within = duration, interval = 20 millis)
      .props(Props[RoutingActor]))

  val future: Future[Any] = router ? "hello!"
  val response: String = Await.result(future.mapTo[String], duration)

  println(response)
}
```

ConsistentHashingPool

말그대로 Hashing. Consistent Hashing 사용. 같은 키를 가지는 메시지를 항상 같은 액터에게 전달.

```
package com.practice.ex01

import akka.routing.ConsistentHashingRouter.ConsistentHashable

object Message {
  final case class Create(key: String, value: String)
  final case class Get(key: String) extends ConsistentHashable {
    override def consistentHashKey: Any = key
  }
  final case class Remove(key: String)
}
```

```
package com.practice.ex01

import akka.actor.Actor
import com.practice.ex01.Message._

class Cache extends Actor {
  var cache = Map.empty[String, String]

  override def receive: Receive = {
    case Create(key, value) =>
      println(s"${self.path.name} : Create $key/$value")
      cache += (key -> value)

    case Get(key) =>
      println(s"${self.path.name} : Get $key")
      context.sender ! cache.get(key)
```

```
    case Remove(key) =>
      println(s"${self.path.name} : Remove $key")
      cache -= key
  }
}
```

```
package com.practice.ex01

import akka.actor.{ActorRef, ActorSystem, Props}
import akka.pattern.ask
import akka.routing.ConsistentHashingPool
import akka.routing.ConsistentHashingRouter.{ConsistentHashMapping,
ConsistentHashableEnvelope}
import akka.util.Timeout
import com.practice.ex01.Message._

import scala.concurrent.Await
import scala.concurrent.duration._

object Client extends App {
  def hashMapping: ConsistentHashMapping = {
    case Remove(key) ⇒ key
  }

  val actorSystem = ActorSystem("Hello-Akka")
  val cache: ActorRef =
    actorSystem.actorOf(
      ConsistentHashingPool(10, hashMapping = hashMapping).
      props(Props[Cache]), name = "cache")
 // 생성
  cache ! ConsistentHashableEnvelope(message = Create("hello", "HELLO"),
hashKey = "hello")
  cache ! ConsistentHashableEnvelope(message = Create("hi", "HI"), hashKey =
"hi")

  // Get 응답 받기, 삭제등 시험
  val duration = 10 seconds
  implicit val timeout = Timeout(duration)

  val future1 = cache ? Get("hello")
  val response1 = Await.result(future1.mapTo[Any], duration)
  println(response1)

  val future2 = cache ? Get("hi")
  val response2 = Await.result(future2.mapTo[Any], duration)
  println(response2)

  cache ! Remove("hi")
```

```
  val future3 = cache ? Get("hi")
  val response3 = Await.result(future3.mapTo[Any], duration)
  println(response3)
}
```

```
$d : Create hello/HELLO
$e : Create hi/HI
$d : Get hello
Some(HELLO)
$e : Get hi
Some(HI)
$e : Remove hi
$e : Get hi
None
```

Dynamically Resizable Pool

https://doc.akka.io/docs/akka/2.5/routing.html#specially-handled-messages
https://doc.akka.io/docs/akka/2.1.2/scala/routing.html

```
package com.practice.ex01

import akka.actor.{Actor, ActorSystem, Props}
import akka.pattern.ask
import akka.routing.{DefaultResizer, ScatterGatherFirstCompletedPool}
import akka.util.Timeout

import scala.annotation.tailrec
import scala.concurrent.duration._
import scala.concurrent.{Await, Future}

// Message
case class FibonacciNumber(nbr: Int)

// Actor
class FibonacciActor extends Actor {
  def receive = {
    case FibonacciNumber(nbr) => sender ! fibonacci(nbr)
    case _ => new IllegalArgumentException
  }

  private def fibonacci(n: Int): Int = {
    @tailrec
    def fib(n: Int, b: Int, a: Int): Int = n match {
      case 0 => a
```

```
      case _ => fib(n - 1, a + b, b)
    }

    fib(n, 1, 0)
  }
}

// Client
object Client extends App {
  val duration = 10 seconds
  implicit val timeout = Timeout(duration)

  val akkaSystem = ActorSystem("Hello-Akka")
  val resizer = DefaultResizer(lowerBound = 4, upperBound = 15)

  val router = akkaSystem.actorOf(
    ScatterGatherFirstCompletedPool(5, within = duration, resizer =
Some(resizer))
      .props(Props[FibonacciActor]))

  val future: Future[Any] = router ? FibonacciNumber(20)
  val response: Int = Await.result(future.mapTo[Int], duration)

  println(response)
  //6765
}
```

Futures and Agents

- Future : **Concurrency** 혹은 **Pararell** 실행을 위해 다른 Thread에서 작업할 때 사용. Callback 및 다른 작업을 실행하기 위해 Thread Pool이 지원하는 ExcutionContext를 필요.
    - 단, Actor를 이용할 경우 ActorSystem 자체에서 제공하므로 명시적으로 필요하지 않다.

From:
https://ledyx.xyz/ - **Ledyx WIKI**

Permanent link:
https://ledyx.xyz/doku.php?id=back-end:akka&rev=1612676902

Last update: **2026/10/24 07:03**