

Hanbit
RealTime
117

**Early
Release**

UNFINISHED

임백준의 아카 시작하기

Akka 개념 잡기

임백준 지음



임백준의
아카 시작하기

Akka 개념 잡기

임백준 지음





표지 사진 신영호

이 책의 표지는 신영호님이 보내 주신 풍경사진을 담았습니다.
리얼타임은 독자의 시선을 담은 풍경사진을 책 표지로 보여주고자 합니다.

사진 보내기 ebookwriter@hanbit.co.kr

임백준의 아카 시작하기 | Akka 개념 잡기

초판발행 2015년 10월 29일

지은이 임백준 / 펴낸이 김태현

펴낸곳 한빛미디어(주) / 주소 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 9월 30일 제10-1779호

ISBN 978-89-6848-789-7 15000 / 정가 11,000원

총괄 배용석 / 책임편집 김창수

디자인 표지/내지 여동일, 조판 최승실

마케팅 박상용 / 영업 김형진, 김진불, 조유미

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 [한빛미디어\(주\)](#)의 홈페이지나 아래 이메일로 알려주십시오.

한빛미디어 홈페이지 www.hanbit.co.kr / 이메일 ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2015 임백준 & HANBIT Media, Inc.

이 책의 저작권은 임백준과 [한빛미디어\(주\)](#)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.

[한빛미디어\(주\)](#)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

한빛미디어에서 『나는 프로그래마다 2015』, 『폴리글랏 프로그래밍』, 『누워서 읽는 퍼즐북』, 『프로그래밍은 상상이다』, 『뉴욕의 프로그래머』, 『임백준의 소프트웨어 산책』, 『나는 프로그래마다』, 『누워서 읽는 알고리즘』, 『행복한 프로그래밍』을 출간했고, 로드북에서 『프로그래머 그 다음 이야기』를 출간했다. 현재 월스트리트에서 자바, C#, 스칼라 언어를 이용해 금융 관련 소프트웨어를 개발하고 있으며, 뉴저지에서 아내, 두 딸과 함께 살고 있다. 개발자들을 위한 유쾌한 팟캐스트 <나는 프로그래마다>(<http://iamprogrammer.io>)의 MC로 활동 중이다.

이 책은 아카 입문서다. 아카를 한 번도 사용해본 적이 없는 사람이 최대한 빠른 속도로 아카를 이용하여 코딩을 시작하게 하는 것이 목적이다. 그런 의미에서 매우 실전적인 책이라고 말할 수 있다. 이 책에는 아카가 제공하는 개념과 API가 모두 포함되어 있지 않다. 이론적인 설명은 최대한 줄이고 코드에 집중했기 때문이다. 책에 포함된 코드를 돌려보면 아카의 핵심적인 개념이 저절로 익숙해지도록 책을 구성했다.

이 책을 쓰는 시점에서 나는 맨해튼에 있는 작은 스타트업 회사에서 일하고 있다. 구글 애드워즈 AdWords와 비슷한 서비스를 제공하는 회사인데, 하루 평균 10억 개 정도의 HTTP 요청을 받는 서버 클러스터를 담당하고 있다. 서버 클러스터는 원래 톰캣 Tomcat 기반이었는데, 그것을 아카 기반으로 바꾸는 프로젝트를 진행했다. 프로젝트를 처음 시작하던 무렵에 기존 팀원들을 대상으로 몇 차례 아카를 강의했고, 각자 노트북을 들고 오라고 해서 코딩실습 시간도 가졌다.

그때 강의를 들은 팀원들이 간단한 코드를 통해서 연습하니 아카에 대한 문서만 읽는 것에 비해서 훨씬 빠르고 정확하게 개념을 이해할 수 있었다는 피드백을 주었다. 강의를 진행한 입장에서 반갑고 고무적인 이야기였다. 이 책에서 사용한 소스코드는 대부분 그때 사용한 것이다. 한국에서도 최근 아카에 대한 관심이 높아지고 있다는 이야기를 듣고 이 경험을 책으로 만들어서 공유하고 싶다는 생각을 하게 되었다.

아카는 현재 타입세이프 Typesafe의 공동창업자이자 CTO인 요나스 보네어 Jonas Boner 가 스칼라 언어를 이용해 개발했다. 스칼라가 초기에는 액터 모델을 구현한 별도의 라이브러리를 가지고 있었는데, 보네어가 개발한 아카의 성능과 안정성이 더 뛰어난 것을 확인하고 아카를 스칼라 언어의 공식 라이브러리로 채택했다. 또한, JVM 환경에서는 자바를 사용하는 개발자가 다수이기 때문에 스칼라로 작성된 아카 위에 얹은 자바 API를 입혀놓기도 했다. 나 역시 이 책을 쓰면서 자바 개발자를 염두에 두었기 때문에 책에 포함된 소스코드는 모두 자바를 사용했다.



아카가 구현한 액터 모델^{actor model}은 오래전인 1973년에 칼 휴이트^{Carl Hewitt}가 제안한 수학적 모델을 기초로 삼고 있다. 멀티스레딩 환경에서 동시성 프로그램을 작성하는 일이 점점 어려워지고, 한 대의 컴퓨터가 사용하는 CPU 코어의 수가 빠른 속도로 늘어나는 요즘에, 아카는 동시성 코드를 작성하기 위한 직관적이고 편리한 프로그래밍 모델을 제공한다.

액터 모델이 설명하는 내용은 그 자체로는 간단하기 때문에 누구나 쉽게 이해할 수 있다. 하지만 원리를 실제 코드에 적용시키는 것은 차원이 다른 문제다. 아카를 사용하는 프로젝트를 진행하면서 머리로 이해한 개념을 실전에 적용하지 못해서 애를 먹는 사람을 많이 보았다. 그럴 수밖에 없는 것이, 아카는 몇몇 개념을 이해하면 익힐 수 있는 게 아니라 근본적인 패러다임의 전환을 받아들일 때 비로소 자기 것으로 만들 수 있기 때문이다. 그런 면에서 아카 프로그래밍은 언뜻 생각하기보다 쉽지 않다.

이러한 패러다임 전환의 내용은 다음과 같다.

- 객체의 메서드를 직접 호출할 수 없고, 오직 메시지를 전달할 수 있을 뿐이다.
- 모든 것이 비동기적^{asynchronous}이다.
- 블로킹^{blocking}이 일어나면 안 된다.
- 모든 것이 동시적^{concurrent}이다.

객체지향 개념의 진정한 출발점이라고 일컬어지는 스몰토크^{Smalltalk}를 개발할 때 앤런 케이는 오늘날 우리가 아카를 통해서 보는 방식을 꿈꾸었다. 모든 것이 독립적인 객체고, 객체들이 메시지를 주고받는 방식으로 상호작용하는 세계가 그것이다. 하지만 우리에게 친숙하게 다가온 객체지향 언어는 C++였고, 이후에 자바나 C# 같은 언어가 뒤를 이었다. C++, 자바, C# 등의 언어에서 객체가 상호작용을 하는 방식은 서로의 메서드를 직접 호출하는 것이다. 메서드를 호출하면 이것이 리턴할 때까지 기다려야 하므로 이는 동기적^{synchronous}이고 블로킹을 야기한다.

아카를 사용해 프로그래밍하는 것은 이러한 동기적 세계를 떠나서 비동기적 세계로 들어가는 것을 의미한다. 아카에서는 모든 것이 철저하게 비동기적이다. 동기적인 메서드 호출에 익숙한 사람들에게 이것은 심오한 패러다임 전환이다. 순차적으로 질서 정연하게 이루어지던 작업이 모두 해체되어 각각의 유닛이 동시에 별적으로 동작하는 세계는 기존의 방식으로 사고 reasoning하기 어렵다. 이러한 패러다임 전환에 도전하는 사람은 코딩을 할 때 자기가 알고 있던 접근방식이 뿌리부터 흔들리는 전율을 맛보게 된다. 지적욕구가 충만한 프로그래머에게는 물론 즐겁고 유쾌한 전율이다.

아카는 강력하다. 아카가 동작하는 원리를 알지 못하는 사람에게 그것은 때로 마법처럼 보인다. 아카는 이미 수많은 회사에서 사용하고 있는 중이며, 스파크 Apache Spark를 비롯한 수많은 오픈소스 라이브러리에서 활용되고 있다. 아카는 모든 문제를 해결해 주지 않지만, 분산 컴퓨팅, 확장성, 동시성, 반응형 reactive과 같은 요소를 고려할 때 훌륭한 선택이 되고 있다. 타입세이프에서 주장하는 반응형 플랫폼 reactive platform에서 아카는 스파크, 플레이 Play와 함께 3대 요소의 자리를 차지하고 있으며, 액터 모델은 STM Software Transaction Memroy과 함께 차세대 동시성 프로그래밍 구조물로 손꼽히는 방법이기도 하다.

한국에서도 아카에 대한 관심이 많이 일어나고 있다. 아직은 아카를 실전 코드에 활용하기보다는 실험적으로 접근하는 곳이 많지만, 가까운 장래에 아카에 대한 관심이 폭발적으로 일어날 거라고 확신한다. 아직은 초기라서인지 국내에서는 아카에 대한 좋은 입문서를 찾아보기 어렵다. akka.io에 있는 문서가 자세한 내용을 담고 있기는 하지만 영문으로 되어 있고, 개념을 중심으로 설명을 하기 때문에 아카를 처음 접하는 사람들이 활용하기에는 불편하다.

그런 의미에서 이 책은 아카를 처음 공부하려는 한국의 개발자에게 점프스타트 입문서의 역할을 수행할 수 있을 것이다. 아카를 사용하는 사람이 아직 충분히 많지 않은



지금, 이 책을 읽은 독자들이 열심히 아카를 공부해서 새로운 패러다임의 선구자가 되기를 희망한다.

아카를 소개하는 책을 출간할 수 있도록 선뜻 동의해준 한빛미디어 김태현 사장님과 김창수 팀장께 고마움을 전한다. 그리고 아카의 세계에 발을 들여놓은 여러분을 진심으로 환영한다. 원래 아카는 라포니아^{Laponia}라는 북부 스웨덴 지역에 있는 아름다운 산의 이름이다. 이제 아카에 발을 들여놓은 그대의 눈앞에 지금까지 경험해보지 못한 환상적인 절경이 펼쳐질 거라는 점을 의심치 않는다.

이 책을 읽기 전에 준비해야 할 것들

소프트웨어

이 책에서 사용하는 예제를 실행하려면 다음 소프트웨어가 필요하다.

- 자바 JDK 1.8 버전 이상
- 메이븐 최근 버전
- 아카 2.3.9
- IDE(이클립스나 인텔리제이)

이 책을 쓰는 동안 다음 소프트웨어와 버전을 사용했다. 이러한 소프트웨어를 설치하는 방법에 대해서는 별도로 설명하지 않겠다.

- 윈도우 7
- 자바 JDK 1.8.0_25
- 메이븐 3.1.0
- 아카 2.3.9
- 이클립스 루나 4.4.1

소스코드

이 책에서 설명하는 예제 코드는 모두 깃헙에 공개되어 있다. 다음 사이트를 방문하면 실행 가능한 소스코드를 다운로드할 수 있다. 코드의 자세한 내용은 책의 본문에서 설명한다.

- <https://github.com/baekjunlim/AkkaStarting>



아카 문서

아카에 대한 책은 다수가 출간되어 있지만 <http://akka.io/>에 있는 문서를 읽는 것이 가장 좋다. 아카를 사용하는 개발자라면 반드시 akka.io에 있는 문서를 읽고 내용을 숙지해야 한다. 자바 API와 스칼라 API를 모두 포함하고 있으므로 본인이 사용하는 언어에 맞는 내용을 선택해서 읽으면 된다.

아카는 스칼라 언어를 이용해 작성되었다. 하지만 자바 언어를 사용하는 개발자의 수가 훨씬 많아서 자바 개발자들도 아카를 사용할 수 있게 하기 위해 스칼라 코드 위에 자바 API를 입혀놓았다. 이 책에서 사용한 언어는 자바고, 따라서 아카가 제공하는 자바 API를 사용했다.

이 책의 내용이 아카 문서가 설명하고 있는 기능을 모두 포함하지 않는다는 사실을 기억할 필요가 있다. 이 책의 목적은 아카에 포함되어 있는 모든 기능과 개념을 설명하는 것이 아니다. 이 책은 여러분의 손이 지금 당장 키보드를 두드리도록 만들고, 그렇게 하는 동안 자연스럽게 추상적인 개념을 익히도록 의도되었다.

한빛 리얼타임

한빛 리얼타임은 IT 개발자를 위한 전자책입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾기도 쉽지 않습니다. 또한, 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1 eBook First –

빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한, 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2 무료로 업데이트되는 전자책 전용 서비스입니다

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3 독자의 편의를 위해 DRM-Free로 제공합니다

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 어려운 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 총고에 귀 기울이며 지속해서 발전시켜 나가겠습니다.

지금 보시는 전자책에 소유 권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 큽니다. 이 경우 저작권법에 따라 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한, 한빛미디어 사이트에서 구매한 후에는 횟수와 관계없이 내려받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 불이기가 가능합니다.

전자책은 오픈자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려 드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구매하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

chapter 1 아카에 대하여 ————— 015

처리율	016
스케일 아웃	018
모듈화	021
차세대 동시성 모델	022

chapter 2 평퐁 게임 ————— 025

평퐁액터	025
액터시스템 만들기	028
액터 만들기	029
ActorRef	031
장소 투명성	034
메시지 전송	034
메일박스	036
onReceive	037
액터 라이프사이클	040

chapter 3 아카 계층구조 ————— 043

아카 계층구조	043
보내고 잊기	049
작업의 완료	050
메시지 순서	052
테크닉	054
액터의 내부 상태와 스레드	057



chapter 4 고장 나도록 허용하라 —— 059

고장 나도록 허용하라	059
SupervisorStrategy	068
Resume	070
Restart	070
Stop	072
Escalate	073
기본 감시전략	073
재귀와 연대책임	073

chapter 5 액터와 상태기계 —— 075

액터의 상태	075
상태기계	077

chapter 6 라우터 —— 087

라우터	087
라우터와 감시전략	096
풀과 그룹	097
라우팅 알고리즘	098

chapter 7 퓨처와 에이전트 —— 101

퓨처	101
예제코드	103
블로킹 호출	109
난블로킹 호출	110
에이전트	113



chapter 8 클러스터 —— 117

클러스터	117
코드	120
구성파일	123
actor.provider	125
actor.remote	126
actor.cluster	127
deployment	128

아카에 대하여

내가 아카를 처음 사용한 것은 2013년 봄이었다. 모건스탠리에서 헤지펀드를 비롯한 클라이언트들에게 투자기술을 제공하는 부서에서 근무하던 때였다. 수백 개의 헤지펀드 중에서 어느 펀드가 갑자기 거래를 중단하는 일이 발생하면 그들이 맡겨놓은 돈을 돌려주기 위한 자금을 어떻게 충당할 것인가를 미리 예측하는 프로그램이 있었다. 2008년 금융 위기 이후에 미국 정부는 모든 은행에게 이러한 계산을 수행한 결과를 매일 제출하도록 요구했다.

정부가 요구하는 리포트를 만들기 위해서 프로그램은 다양한 방식으로 펀드를 선택했다. 일정한 규칙에 따라서 1개, 10개, 100개 등 여러 펀드를 선택해 그들이 당장 거래를 중단했을 때 (혹은 파산했을 때) 그것이 은행의 자금사정에 어떤 영향을 끼치는지를 정밀하게 분석해서 계산했다. 한 번 계산하고 마는 것이 아니라 선택된 모형마다 시나리오를 1,000,000번씩 실행해 정규분포를 만들어내는 몬테카를로 프로그램이었다.

누군가 자바를 이용해 만들어놓은 이 프로그램은 자바 스레드와 Executor Service를 이용하는 방식으로 작성되어 있었다. 한 번 동작을 시작하면 결과를 내놓기까지 대략 6시간이 걸렸다. 시간을 단축할 필요가 있었다. 비즈니스 요구사항에 따라서 다양한 시나리오를 적용해보고 싶은데 시간이 장애물이었기 때문이다. 이런 경우에 가장 손쉽게 떠올릴 수 있는 방법은 계산해야 하는 시나리오를 분할해 여러 대의 컴퓨터에 분산되어 있는 프로그램에 할당하는 것이다.

하지만 프로그램이 스케일업^{scale-up}은 가능해도 스케일아웃^{scale-out}은 가능하지 않은 방식으로 작성된 게 문제였다. 스케일업은 컴퓨터에 CPU와 메모리를 추가해서 성능을 높이는 방식을 의미하고, 스케일아웃은 별도의 컴퓨터를 추가해 병렬처리를 수행하는 것을 의미한다. 아무튼, 프로그램의 실행속도를 줄이기 위해 자원을 추가하는 것이 상식적인 접근방법인데, 그렇게 투입된 자원을 소프트웨어 자체가 제대로 활용하지 않으면 가망이 없다.

당시는 금융 위기 이후 회복세를 타기 시작한 월스트리트에 헤지펀드의 수가 빠르게 늘어나는 시점이었기 때문에 더 늦기 전에 프로그램을 대폭으로 리팩토링하기로 결정했다. 이때 염두에 둔 요구사항은 다음과 같다.

- 컴퓨터 1대 위에서 돌아가는 속도가 6시간보다 빠를 것
- 컴퓨터를 추가해 병렬처리가 가능하게 할 것
- 동작하는 방식이 이해하기 편하고 새로운 시나리오 추가가 쉽도록 코드를 만들 것

개념을 중심으로 정리하면 첫째는 일반적인 처리율^{throughput}에 대한 것이고, 둘째는 스케일아웃에 대한 것이며, 셋째는 코드의 모듈화^{modularity}에 대한 것이다. 이러한 세 가지 요구사항을 충족시키기 위해서 나는 아카를 선택했다.

처리율(throughput)

아카를 이용한 리팩토링을 끝마쳤을 때, 똑같은 컴퓨터 위에서 전과 동일한 몬테카를로 시나리오를 수행하는데 걸리는 시간이 6시간에서 2시간으로 단축되었다. 66%의 시간이 절약된 것이다. 결과를 확인한 사람들은 깜짝 놀랐다. 단순히 자바스레드에서 아카로 라이브러리를 바꾸었을 뿐인데 그렇게 엄청난 차이가 있을 수 있느냐며 고개를 갸웃거렸다.

물론 이런 차이를 일반화할 수는 없다. 이런 결과 하나를 가지고 아카가 자바 스레드보다 3배 빠르다고 말하는 어리석은 사람은 없을 것이다. 아카도 내부적으로 자

바 스레드를 사용하기 때문에 그런 비교 자체가 성립하지 않는다. 하지만 일반적인 차원에서 짚고 넘어갈만한 부분도 있다. 이렇게 커다란 차이가 어디에서 비롯되었는지 이해하려면 우선 암달의 법칙Amdahl's law을 생각해볼 필요가 있다. 암달의 법칙은 이렇다.

“멀티코어를 사용하는 프로그램의 속도는 프로그램 내부에 존재하는 순차적sequential 부분이 사용하는 시간에 의해서 제한된다.”

Thread나 Task를 만들어서 ExecutorService에게 제출하는 식으로 동시성 코드를 작성하면 여러 개의 스레드가 동시에 작업을 수행한다. 하지만 프로그램 안에는 Thread나 Task가 포함하지 않는 코드가 존재한다. 여러 개의 스레드가 동시에 작업을 수행하더라도 synchronized 블록이나 데이터베이스, 네트워크 API 호출 등을 만날 때 다른 스레드와 나란히 줄을 서서 순차적으로 작업을 수행해야 하는 경우도 있다. 암달의 법칙은 프로그램이 낼 수 있는 속도의 상한이 이런 순차적 코드가 사용하는 시간에 의해서 제한된다고 말하는 것이다.

이러한 순차적 코드의 또 다른 이름은 블로킹blocking 콜이다. 문제는 스레드 자체가 아니라 스레드를 사용하면서 자기도 모르게 만들어내는 블로킹 콜이다. 조금 과장해서 말하자면 자바 개발자가 스레드를 이용해서 만들어내는 ‘동시성’ 코드는 일종의 신기루다. 사실은 코드 곳곳에 존재하는 블로킹 콜, 순차적 코드 때문에 전체적인 프로그램의 처리율은 이미 상한이 정해져 있지만 여러 개의 스레드가 ‘동시에’ 동작한다는 사실로부터 위안을 받을 뿐이다.

아카 내부에 숨겨진 마법 같은 것은 없다. 아카는 스칼라 언어로 작성되었지만 더 아래로 내려가면 자바의 동시성 패키지를 사용하기 때문에 아카를 사용하는 것은 궁극적으로 자바의 Thread나 Task를 사용하는 것과 마찬가지다. 하지만 아카를 사용하면 프로그램 곳곳에 존재하는 순차적 부분, 블로킹 콜을 전부 없애거나 적어도 최소한으로 만드는 것이 가능해진다. 6시간이 2시간으로 줄어드는 ‘마법’은

여기에서 비롯된 것이다.

아카는 물리적으로 가벼운 라이브러리지만, 어떤 의미에서는 하나의 패러다임이다. 블로킹 blocking 혹은 동기적 synchronous 방식의 프로그래밍에 익숙한 우리의 사고 방식을 난블로킹 non-blocking 혹은 비동기적 asynchronous 방식으로 탈바꿈시킨다는 점에서 패러다임 전환을 요구한다. 아카를 이용해서 프로그램을 설계한다는 것은 블로킹 호출이 일어나는 지점을 난블로킹 호출로 전환하는 작업을 수행하는 것을 의미한다. 그렇기 때문에 암달의 법칙에서 이야기하는 순차적 부분이 차지하는 면적이 최소한으로 줄어들게 되고, 프로그램의 전체적인 처리율은 그와 반비례해서 급등하게 된다.

이런 이야기가 아직 구체적으로 감이 잡히지 않아도 걱정할 필요는 없다. 이 책에 담긴 예제코드를 공부하고 아카를 이용한 실전코드를 작성해보면 금방 이해가 될 것이다. 다만 블로킹/동기적 호출은 낡은 방식이고, 난블로킹/비동기적 호출은 현대적 방식이라는 점은 기억할 필요가 있다. 그게 핵심이다. 그리하여 낡은 방식을 고집할 것인가 아니면 현대적 방식을 받아들일 것인가는 결국 자신의 선택임을 잘 생각해보기 바란다.

스케일 아웃(scale out)

아카가 가진 가장 탁월한 장점은 스케일 아웃을 자동적으로 보장해준다는 점이다. 스케일 아웃을 위해서 해야 하는 일이 실제로 거의 없다. 구성파일의 내용을 약간 수정하는 게 전부다. 간단한 예를 생각해보자.

수많은 사람이 방문하는 웹사이트가 있다고 가정하자. 웹서버로 1대의 컴퓨터 위에서 돌아가는 1개의 톰캣 JVM을 사용한다. 그런데 방문자의 수가 늘어서 1대의 서버로는 더 이상 트래픽을 감당할 수 없게 되었다. 이런 경우에 우리는 전통적으로 로드 밸런서 load-balancer 뒤에 여러 대의 서버를 설치하는 방식을 사용한다. 로드

밸런싱 클러스터 구축, 혹은 로드 밸런서를 이용한 스케일 아웃이다.

이렇게 하면 어느 정도 수준의 스케일은 충분히 감당할 수 있다. 하지만 최선은 아니다. 무엇보다도 여러 대의 컴퓨터 위에서 돌아가는 톰캣 서버가 서로의 존재를 알지 못한다. 서로 완전히 독립되어 있다. 그러한 독립성이 장점일 수도 있지만 단점일 수도 있다. 예를 들어서 여러 대의 톰캣 서버에게 각각 특정한 작업을 수행하는 역할을 부여할 필요가 있다고 하자. 그렇게 하는 것이 가능하긴 하지만 복잡한 알고리즘을 구현해야 한다. 불편하다.

더 큰 문제는 자원이용의 효율성이다. 예를 들어서 톰캣 위에서 돌아가는 프로그램 내부에 A부터 D까지 개의 컴포넌트가 존재한다고 하자. 이들은 모두 2만큼의 자원을 사용하는데 유독 C만 4만큼의 자원을 사용한다. 여기에서 자원은 CPU가 될 수도 있고 메모리가 될 수도 있다. 그래서 프로그램 전체가 사용하는 자원은 $2+2+4+2=10$ 이다. 우리가 사용하는 컴퓨터는 자원을 15까지 지원할 수 있다고 하자. 우리는 컴포넌트 C를 스케일하고 싶다. C의 용량만 두 배로 키우면 시스템의 처리율을 두 배로 올릴 수 있음을 안다(고정하자).

하지만 톰캣에서 동작하는 프로그램은 대부분 하나의 큰 덩어리, 영어로 ‘monolithic’하다고 표현하는 방식으로 작성되어서 하나의 컴포넌트만 떼어서 독립적으로 실행하기가 어렵다. 설명 그렇게 할 수 있다고 해도 그 컴포넌트가 다른 컴포넌트와 커뮤니케이션 하는 것이 문제다. 그래서 보통은 프로그램 전체를 복제한다. 즉, 자원소모가 $2+2+4+2=10$ 에 달하는 프로그램 전체를 하나 더 생성해서 실행시키는 것이다.

그런데 우리는 앞에서 컴퓨터가 처리할 수 있는 용량이 15라고 말했다. 따라서 추가적으로 생성된 프로그램은 별도의 컴퓨터 위에서 동작해야 한다. 10만큼의 자원을 사용하는 프로그램을 동시에 2개 실행시킬 수 없기 때문이다. 이 시점에서 생각해보면, 자원 사용량이 4에 불과한 C라는 컴포넌트를 스케일하기 위해 우리

가 자원 사용량이 10에 달하는 프로그램 전체를 복제했음을 알 수 있다. 심지어 새로운 컴퓨터까지 필요하다. 단순히 4를 복제해 8로 만들려고 10을 20으로 복제하는 것은 받아들이기 힘든 비효율성이다.

아카의 경우라면 이야기가 다르다. 아카 세계에서 컴포넌트는 액터다. 어느 액터를 하나만 콕 집어내어 기존의 JVM에서 독립시켜 독자적인 JVM으로 실행하는 것은 거의 아무런 노력이 들지 않는다. 굳이 별도의 JVM으로 독립시키지 않고 동일한 JVM 내부에서 인스턴스의 수만 늘어나게 하는 방법도 가능하다. 본문에서 설명하겠지만 장소 투명성(location transparency)라는 기능이 제공되기 때문에 이러한 스케일을 위해서 코드를 수정할 필요가 전혀 없다.

이 예에서 우리가 스케일 하려는 대상은 C라는 컴포넌트다. 아카를 사용하면 다른 부분은 건드리지 않고 C만 두 배로 만들어주는 것이 가능하다. C가 두 배가 되면 자원 사용량이 $2+2+8+2=14$ 인데, 컴퓨터가 15까지는 지원할 수 있으므로 별도의 컴퓨터가 필요하지 않다. 이런 유연한 스케일은 시스템 전체의 처리율을 적정한 수준으로 유지하는 작업을 수월하게 만들어준다. 반응형 선언(reactive manifesto)에서는 이렇게 스케일 업/다운, 아웃/인을 자유자재로 지원하는 기능을 일컬어서 탄력성(elasticity)이라고 말한다. 이런 특성을 가장 직관적으로, 풍부하게 지원해주는 라이브러리가 바로 아카다.

아카에서 제공하는 클러스터(clustering) 기능은 매우 강력하고 편리하다. 내가 재작성한 몬테카를로 프로그램은 컴퓨터를 1대만 사용해도 이미 우리가 원했던 것보다 빠른 시간을 보여주었기 때문에 클러스터를 사용할 필요가 없었다. 하지만 내가 지금 다니고 있는 회사에서는 아카 클러스터 기능을 이용해서 여러 가지 흥미로운 패턴을 만들어내고 있는 중이다. 아카 라이브러리가 아니었으면 쉽게 생각하지 못했을 일들을 구현하는 작업이 즐겁다.

모듈화(modularity)

일반적으로 아카 라이브러리에 관심을 갖는 사람들은 아카를 주로 자바의 Thread나 Task를 대신하여 사용할 수 있는 동시성 라이브러리라는 측면에서 접근한다. 이렇게 접근하면 한 가지 중요한 사실을 간과하는 경우가 많다. 아카의 액터 모델이 단순히 동시성 코드를 지원하기 위한 모델이 아니라 오히려 진정한 의미에서의 객체지향을 구현하기 위한 패러다임이라는 사실을 잊는 것이다.

액터가 제공하는 가장 강력한 기능의 하나는 사실 모듈화다. 아카를 이용해서 시스템을 설계하면 ‘클래스’나 ‘객체’를 중심으로 생각하던 사고방식이 ‘액터’를 중심으로 생각하는 방식으로 변하게 된다. 우리는 자바나 C# 같은 언어에서 사용하는 객체가 객체지향의 원리를 충실히 구현하고 있다고 생각하지만, 사실 그렇지 않다. 객체들은 서로의 메서드를 동기적인 방식으로 호출하면서 관련을 맺기 때문에 다른 객체의 내부에서 발생하는 사건(예를 들어, 예외 exception가 발생하는 것)으로부터 직접적인 영향을 받는다. 서로의 삶이 직접적으로 맞닿아 있기 때문이다. 다른 말로 하면, 우리가 사용하는 객체는 서로 밀접하게 결합 tightly coupled 되어 있어 나쁘다.

그에 비해 액터는 서로 완벽하게 독립적이며, 오로지 메시지를 주고받는 방식으로만 커뮤니케이션하므로 코드의 응집성 coherence, 느슨한 결합 loosely coupled, 캡슐화 encapsulation과 같은 프로그래밍 원리를 완벽하게 구현한다. 이런 원리가 성립하는 방식으로 코드를 설계하다 보면 코드 조각들이 특정한 기능을 중심으로 자연스럽게 모여든다. 그 조각이 하나의 독립적인 액터, 혹은 컴포넌트를 형성하는 것이다.

이런 컴포넌트를 뚝 떼어내서 독립적인 JVM 위에서 실행하려면 클러스터를 구축 할 수도 있고, 아예 독자적인 서비스로 만들려면 마이크로서비스 아키텍처를 구현 할 수도 있다. 이렇게 아카를 이용해 코드를 작성하다 보면 진정한 의미에서의 객체지향적인 사고방식을 체화하게 되기 때문에 아카를 사용하는지와 상관없이 프로그래밍 실력 일반에 좋은 영향을 받는다. 엄청난 보너스다.

차세대 동시성 모델

소프트웨어 세계는 빠르게 변한다. 하지만 시류와 상관없이 지속적으로 관철되는 법칙도 있다. 그 중에서 하나는 ‘추상수준 상승의 법칙’이다. 멀리 갈 것 없이 프로그래밍 언어만 생각해도 충분하다. 0과 1로 이루어진 기계어는 사람이 쉽게 이해 할 수 있는 기호를 사용하는 어셈블리어로 추상수준이 높아졌고, 그것은 훗날 C 와 같은 범용 언어로 추상수준이 한 단계 더 높아졌다. 바이트코드와 가상기계를 사용하는 자바에 이르러서 추상수준이 또 한 계단 상승한 것은 물론이다. 그 자바 는 이제 함수 패러다임이 요구하는 추상수준으로 더 올라가야 한다는 압력을 심하게 받고 있는 중이다.

소프트웨어 세계의 모든 문제는 코드와 코드 사이에 하나의 계층을 도입하는 방법, 영어로 ‘one more indirection’이라고 부르는 방법으로 풀 수 있다는 이야 기도 있다. 구체적인 예를 들자면 디자인 패턴에서 사용하는 어댑터adapter 패턴이 대표적이다. 이것도 넓은 의미에서는 추상수준 상승의 법칙과 일맥상통한다. 계속 추상수준이 상승하고 코드와 코드 사이에 또 하나의 계층이 도입된다는 법칙은 모든 분야에 적용되기 때문에 스레드와 관련된 부분도 예외가 아니다.

폴 부처Paul Butcher가 쓴 『7주 동안에 익히는 7개의 동시성 모델 Seven Concurrency Models in Seven Weeks』(2014, Pragmatic)은 동시성 코딩과 관련해서 자바에서 사용하는 스레 드와 잠금장치lock 정도만 알고 있는 개발자에게 현대 프로그래밍이 어떤 새로운 기법들을 추구하고 있는지 잘 설명해준다. 액터 모델은 물론 7개의 모델 중에서 하나다. 모든 개발자가 반드시 알고 넘어가야 하는 차세대 동시성 모델의 하나라는 이야기다.

액터 모델은 동시성 코드를 작성하기 위해서 지금까지 사용해온 구조물과 완전히 다른 차원의 추상수준을 제공한다. 여기에서는 잠금장치lock와 같은 구조물이 없다. 심지어 스레드라는 개념조차 초월해야 한다. (물론 실전에서는 액터와 스레드의 관

계를 제대로 이해할 필요가 있다.) 액터 모델에서는 오직 액터만 존재한다. 그들은 서로 전적으로 독립적인 구조물이며 서로 메서드를 호출할 수도 없고 new를 통해서 새로운 인스턴스를 만들 수도 없다.

풀 부처의 경우에는 동시성 코드를 돋기 위한 도구를 구체적인 코딩의 수준에서 설명하고 있는데 비해서, 요즘 자주 회자되는 반응형 선언은 상당히 추상적인 수준에서 현대 소프트웨어 시스템이 갖춰야 하는 덕목을 4개로 요약하고 있다. 반응성 responsiveness, 탄력성 elasticity, 유연성 resilience, 메시지 중심 event-driven이 그들이다.

우선 반응성은 최종사용자의 입장에서 보았을 때 가장 중요한 요소다. 사용자의 필요성에 대해서 즉각적으로 반응하지 않는 소프트웨어는 존재할 이유가 없다. 아무리 다른 요소가 훌륭해도 상관없다. 모든 소프트웨어는 사용자가 필요로 할 때 빠르게 응답하는 것을 목적으로 삼아야 한다. 그것은 곧 소프트웨어 내부에 어떤 문제가 발생했을 때, 그것을 감지하고 수정하는 작업이 전체적인 반응성에 영향을 주지 않아야 함을 뜻한다. 당연한 이야기처럼 들리지만 소프트웨어가 사용량이 폭증했을 때, 데이터베이스나 네트워크에 문제가 발생했을 때, 코드에 버그가 있었을 때, 이런 모든 상황에 구애받지 않고 일관성 있는 반응성을 유지하게 만드는 것은 쉬운 일이 아니다.

탄력성은 앞에서 이미 보았다. 원래 처음에는 확장성 scalability이라는 표현이 많이 사용되었는데, 확장성은 스케일업만 포함하고 스케일다운을 포함하지 않는 개념이라고 해서 요즘에는 탄력성 elasticity이라는 표현을 더 많이 사용한다. 소프트웨어 시스템은 필요에 따라서 더 많은 (CPU나 메모리 같은) 자원을 이용해서 처리율을 높이는 스케일업이 가능해야 하고, 때에 따라서는 더 적은 자원을 이용하는 스케일다운도 용이해야 함을 뜻한다. 자유롭게 늘어났다 줄어들었다 할 수 있어야 한다는 의미에서 '탄력'이라는 표현이 사용된다.

유연성은 장애허용 fault tolerance과 밀접한 관련이 있는 개념이다. 앞에서 반응성을

이야기할 때 현대 소프트웨어 시스템은 어떤 장애^{fault}나 에러가 발생했을 때 그런 사실에 구애받지 않고 일관성 있는 반응성을 유지해야 한다고 이야기했다. 유연성은 그런 일을 가능하게 만들어주는 구체적인 방법을 포괄한다. 예를 들어서 이러한 유연성은 중복^{replication}, 격리^{isolation}, 대리^{delegation}와 같은 기법을 통해서 확보될 수 있다. 여기에서 중복은 낯선 개념이 아니다. 데이터베이스 시스템이 클러스터로 구성되어 있을 때 똑같은 데이터 조각을 여러 노드에 중복해서 저장하는 것은 하나의 노드가 동작을 멈추었을 때 데이터 손실이 발생하지 않도록 만들기 위한 기법이다. 격리는 소프트웨어를 구성하는 여러 컴포넌트 중에서 어느 한 컴포넌트에서 문제가 발생했을 때, 그것이 나머지 컴포넌트에 영향을 미치지 않도록 일정한 경계를 설정해주는 기법을 의미한다.

메시지 중심은 원래 사건 중심^{event-driven}이라는 개념으로 표현되었는데, 사건^{event}보다 메시지^{message}가 더 적확한 표현이라는 이야기가 나오면서 최근에는 메시지 중심이라는 말을 더 자주 쓴다. 사건과 메시지는 비슷하지만 서로 다른 개념이다. 차이를 하나만 이야기하자면 사건은 목적지^{destination}가 없는데 비해서 메시지는 반드시 목적지를 갖는다. 메시지 중심이라는 개념은 비동기적^{asynchronous} 메시지 전달과 장소 투명성^{location transparency}를 포괄한다.

아카를 학습하다 보면 반응성, 탄력성, 유연성, 메시지 중심이라는 4개의 덕목이 어떻게 아카에 구현되어 있는지 목격하게 된다. 즉 액터 모델은 현대 소프트웨어 시스템이 갖춰야 하는 덕목을 종합적으로 지원해주는 거의 유일한 모델이다. 액터 모델을 구현한 라이브러리가 아카만 있는 것은 아니다. 엘랭^{Erlang}도 액터 모델을 구현해서 사용해왔고, 최근에는 엘랭을 계승한 엘리서^{Elixir}도 액터 모델을 사용하고 있다. 하지만 JVM 언어를 사용하는 개발자에게는 실질적으로 아카가 유일한 라이브러리다.

이야기를 많이 했으니 이제 코드를 보도록 하자.

핑퐁 게임

핑퐁액터

이번 장에서 공부하는 내용은 액터를 만들고 실행하기 위한 가장 기본적인 방법이다. 그렇게 하기 위해서 우리는 ‘핑’이라는 이름의 액터와 ‘퐁’이라는 이름의 액터를 만들 것이다. 핑은 퐁이라는 메시지를 받으면 핑이라는 응답을 보내고, 퐁은 핑이라는 메시지를 받으면 퐁이라는 응답을 보낸다. 그런 식으로 핑액터와 퐁액터가 핑과 퐁이라는 메시지를 주고받으면서 계속 동작을 수행하는 예다.

이런 핑퐁액터는 액터시스템을 공부하기 위한 첫걸음으로 널리 활용되고 있으므로 액터를 위한 “헬로우 월드”라고 생각해도 좋다. 우선 필요한 코드의 내용부터 살펴보도록 하자.

다음은 핑퐁 예제를 실행하는 메인 클래스 전문이다.

```
package org.study;
import org.study.actor.PingActor;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 핑퐁액터 데모를 위한 메인 클래스
 * @author Baekjun Lim
 */
public class Main {
```

```
public static void main(String[] args) {
    ActorSystem actorSystem = ActorSystem.create("TestSystem");
    ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class),
"pingActor");
    ping.tell("start", ActorRef.noSender());
}
}
```

다음은 핑액터의 전문이다.

```
package org.study.actor;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 임의의 문자열 혹은 "pong" 메시지를 받으면 "ping" 메시지를 보내는 핑액터
 * @author Baekjun Lim
 */
public class PingActor extends UntypedActor {
    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef pong;

    @Override
    public void preStart() {
        this.pong = context().actorOf(Props.create(PongActor.class, getSelf()),
"pongActor");
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            String msg = (String)message;
            log.info("Ping received {}", msg);
            pong.tell("ping", getSelf());
        }
    }
}
```

그리고 다음은 풍액터의 전문이다.

```
package org.study.actor;
import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 임의의 문자열 혹은 "ping" 메시지를 받으면 "pong" 메시지를 보내는 풍액터
 * @author Baekjun Lim
 */
public class PongActor extends UntypedActor {
    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef ping;

    public PongActor(ActorRef ping) {
        this.ping = ping;
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            String msg = (String)message;
            log.info("Pong received {}", msg);
            ping.tell("pong", getSelf());
            Thread.sleep(1000); // 실전에서는 절대 금물!!!
        }
    }
}
```

메인 클래스를 실행하면 다음과 같이 “Pong received ping”, “Ping received pong”이라는 메시지가 반복해 화면에 출력되는 것을 확인할 수 있다.

```
[INFO] [08/30/2015 07:06:21.079] [TestSystem-akka.actor.default-dispatcher-5]
[akka://TestSystem/user/pingActor] Ping received start
[INFO] [08/30/2015 07:06:21.080] [TestSystem-akka.actor.default-dispatcher-5]
[akka://TestSystem/user/pingActor/pongActor] Pong received ping
[INFO] [08/30/2015 07:06:21.080] [TestSystem-akka.actor.default-dispatcher-3]
[akka://TestSystem/user/pingActor] Ping received pong
```

```
[INFO] [08/30/2015 07:06:22.080] [TestSystem-akka.actor.default-dispatcher-5]
[akka://TestSystem/user/pingActor/pongActor] Pong received ping
[INFO] [08/30/2015 07:06:22.080] [TestSystem-akka.actor.default-dispatcher-6]
[akka://TestSystem/user/pingActor] Ping received pong
[INFO] [08/30/2015 07:06:23.081] [TestSystem-akka.actor.default-dispatcher-5]
[akka://TestSystem/user/pingActor/pongActor] Pong received ping
[INFO] [08/30/2015 07:06:23.081] [TestSystem-akka.actor.default-dispatcher-4]
[akka://TestSystem/user/pingActor] Ping received pong
[INFO] [08/30/2015 07:06:24.081] [TestSystem-akka.actor.default-dispatcher-5]
[akka://TestSystem/user/pingActor/pongActor] Pong received ping
[INFO] [08/30/2015 07:06:24.081] [TestSystem-akka.actor.default-dispatcher-7]
[akka://TestSystem/user/pingActor] Ping received pong
[INFO] [08/30/2015 07:06:25.081] [TestSystem-akka.actor.default-dispatcher-5]
[akka://TestSystem/user/pingActor/pongActor] Pong received ping
[INFO] [08/30/2015 07:06:25.081] [TestSystem-akka.actor.default-dispatcher-7]
[akka://TestSystem/user/pingActor] Ping received pong
```

여기까지 확인한 사람은 이제 액터시스템을 만들어서 실행해보는 최초의 경험을 하게 되었다. 흥미진진하고 재미있는 일로 가득한 액터 세상에 입문한 것을 환영 한다. 지금까지 코드에서 한 일이 무엇인지 하나씩 자세히 살펴보자.

액터시스템 만들기

액터를 사용하기 위해서는 우선 `ActorSystem`이라는 객체를 만들어야 한다. 모든 액터는 어떤 액터시스템 내부에서 동작을 수행한다. 그런 면에서 액터시스템은 액터를 담는 컨테이너라고 생각해도 좋다. 동일한 액터시스템 안에서 동작하는 액터들은 아카 라이브러리가 제공하는 스퍼드 스케줄링, 구성파일, 로그 서비스 등을 공유한다.

액터시스템은 다음과 같이 `ActorSystem` 클래스에 정의되어 있는 정적 메서드 `create`를 호출함으로써 생성한다. (이렇게 어떤 객체를 생성하기 위한 목적으로 사용되는 정적 메서드를 ‘팩토리’ 메서드라고 부르기도 한다.)

```
ActorSystem actorSystem = ActorSystem.create("TestSystem");
```

`create` 메서드에 전달하는 문자열은 사용하려는 액터시스템의 이름이다. 자기가 원하는 문자열을 아무거나 전달해도 좋다. 액터시스템은 가벼운 구조물이 아니기 때문에 하나의 애플리케이션에서 하나의 액터시스템을 사용하는 것이 일반적이다. 하지만 필요하면 하나의 애플리케이션에서 여러 개의 액터시스템을 만들어서 사용하는 것도 가능하다.

애플리케이션 내부에 액터시스템이 하나만 존재하는 경우에는 액터시스템의 이름이 별다른 의미를 갖지 않는다. 하지만 애플리케이션 안에 여러 개의 액터시스템이 존재하거나 해당 액터시스템이 원격 호스트를 포함하는 클러스터에 참여하는 경우에는 이름이 중요하다. 액터시스템 자체에 대해서는 일단 이 정도만 알아도 충분하다. 더 자세한 내용은 필요에 따라서 조금씩 공부해도 늦지 않다.

액터 만들기

액터를 만드는 방법은 다음과 같다.

```
ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class),  
"pingActor");
```

한 줄에 불과하지만 아카를 처음 접하는 사람에게는 복잡해 보이는 내용이다. 하나씩 살펴보자. 액터시스템이 액터를 만들기 위해서는 기본적으로 다음 두 개의 정보가 필요하다.

- `Props`
- 액터의 이름

프롭스는 액터를 만들기 위한 조리법^{recipe}이다. 액터를 만드는 데 필요한 구성요

소를 담는 일종의 구성파일 같은 클래스다. 한 번 만들어지면 값을 변경할 수 없는 불변^{immutable} 객체이기 때문에 필요한 곳에서 자유롭게 공유할 수 있다.

만들려는 액터 클래스가 인수가 없는 기본 생성자^{constructor}를 사용하는 경우에는 `Props.create(PingActor.class)`처럼 액터의 타입(즉, 클래스) 정보만 전달하면 충분하다. 만약 만들려는 액터 클래스가 인수를 받아들이는 생성자를 사용하면 `Props.create(PingActor.class, "arg1", "arg2")`같이 써서 인수를 전달할 수도 있다.

조리법 다음에는 액터의 이름이다. 액터시스템에서 액터의 이름은 반드시 고유한 문자열이어야 한다. 똑같은 이름을 사용하는 액터를 한 개 이상 만들 수 없다는 뜻이다. 예를 들어, 다음과 같이 'pingActor'라는 동일한 이름을 사용하는 액터를 두 번 만들면 에러가 발생한다.

```
ActorRef ping1 = actorSystem.actorOf(Props.create(PingActor.class),
"pingActor");
ActorRef ping2 = actorSystem.actorOf(Props.create(PingActor.class),
"pingActor");
```

이때 발생하는 에러의 내용은 다음과 같다.

```
Exception in thread "main" akka.actor.InvalidActorNameException: actor name [pingActor] is not unique!
  at akka.actor.dungeon.ChildrenContainer$NormalChildrenContainer.reserve(ChildrenContainer.scala:130)
  at akka.actor.dungeon.Children$class.reserveChild(Children.scala:76)
  at akka.actor.ActorCell.reserveChild(ActorCell.scala:369)
  at akka.actor.dungeon.Children$class.makeChild(Children.scala:201)
  at akka.actor.dungeon.Children$class.attachChild(Children.scala:41)
  at akka.actor.ActorCell.attachChild(ActorCell.scala:369)
  at akka.actor.ActorSystemImpl.actorOf(ActorSystem.scala:553)
  at org.study.AkkaStudyWithBaekjunLimApplication.main(AkkaStudyWithBaekjunLimApplication.java:19)
```

PingActor라는 동일한 클래스를 이용해 하나 이상의 액터를 만들고 싶으면 다음과 같이 이름을 서로 다르게 만들어야 한다.

```
ActorRef ping1 = actorSystem.actorOf(Props.create(PingActor.class), "pingActor1");
ActorRef ping2 = actorSystem.actorOf(Props.create(PingActor.class), "pingActor2");
```

뒤에서 액터의 계층구조를 설명할 때 자세한 내용을 공부하겠지만, 액터들은 트리 tree와 비슷한 계층구조를 형성한다. A라는 액터가 B라는 액터를 만들면, A는 B의 부모가 되고 B는 A의 자식이 되는 식이다. 그렇기 때문에 경로^{path}라는 개념이 존재하게 되는데, 액터의 이름은 이러한 경로에서 사용될 때 의미를 갖는다.

실전 코드에서는 액터의 이름이 실질적인 의미가 없는 경우도 있다. 그럴 때는 고유성^{uniqueness}만 확보하기 위해서 UUID.randomUUID()와 같은 무작위 이름을 사용하기도 한다. 그렇지만 지금은 아카운트 처음 공부하는 입장이므로 액터의 이름이 의미가 있다고 생각하고 접근하는 편이 나을 것이다.

actorOf라는 것은 액터를 만들기 위한 ‘팩토리’ 메서드다. 조리법(즉, Props)과 문자열(즉, 액터의 이름)을 인수로 받아들인다. ActorSystem이나 ActorContext 객체로부터 호출할 수 있다. 이제 앞에서 보았던 이 코드를 다시 보면 어떤 일이 일어나고 있는지 더 많이 이해할 수 있게 되었을 것이다.

```
ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class),
"pingActor");
```

ActorRef

한 가지 흥미로운 점은 ActorSystem의 actorOf 메서드가 생성하는 객체의 타입이 우리가 정의한 PingActor가 아니라 ActorRef 타입이라는 사실이다.

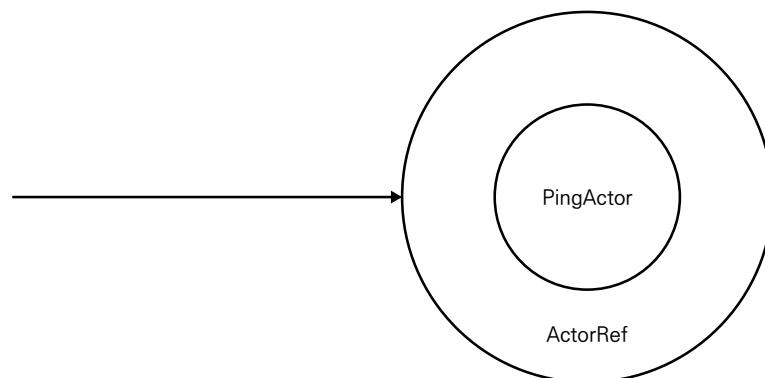
ActorRef는 액터시스템, 혹은 아카를 공부하는 데 있어 가장 중요한 개념의 하나다. 액터시스템 내에서 사용하는 액터는 모두 ActorRef다. PingActor와 같이 만든 타입은 (적어도 겉으로 보기에는) 존재하지 않는다.

ActorRef는 이름에서 알 수 있듯이 액터를 가리키는 참조다. 모든 액터는 오직 ActorRef라는 타입에 의해서만 접근될 수 있다는 뜻이다. ActorRef는 액터 객체를 둘러싸고 있는 껍질이라고 볼 수도 있다. 예를 들어, 앞의 PingActor 안에 다음과 같은 퍼블릭 메서드를 정의했다고 해보자.

```
public int getCount() {  
    return count;  
}
```

이것은 퍼블릭 메서드이므로 PingActor 객체 p가 있다고 했을 때 p.getCount()과 같은 방식으로 메서드를 호출하는 데 익숙하다. 그런데 액터시스템에서는 이와 같은 접근이 전적으로 불가능하다. 이는 PingActor라는 타입을 직접 사용하는 것이 원천적으로 봉쇄되어 있기 때문이다. PingActor에 아무리 많은 퍼블릭 메서드를 정의해도 우리가 가지고 있는 것은 PingActor가 아니라 ActorRef이기 때문에 그런 메서드에 접근할 수 있는 방법이 없다.

그림 2-1



앞에서 우리는 평액터를 만들기 위해서 액터시스템의 actorOf 메서드에 ‘조리법’과 ‘이름’을 전달했다. 액터시스템은 조리법에 따라서 액터를 만들고, 그것을 ActorRef로 둘러싼 다음 되돌려 준다. PingActor는 ActorRef의 내부 어딘가에 저장되어 사라지고 우리 손에 남는 것은 오직 ActorRef일 뿐이다.

PingActor에 정의된 getCount 메서드를 얹지로라도 호출하고 싶으면, 우선 PingActor 객체를 만들어야 한다. 하지만 다음과 같이 PingActor를 전통적인 방식으로 생성하려고 시도하면 에러가 발생한다.

```
PingActor p = new PingActor();
```

에러는 다음과 같은 메시지를 출력한다.

```
Exception in thread "main" akka.actor.ActorInitializationException: You cannot create an instance of [org.study.ch2.PingActor] explicitly using the constructor (new). You have to use one of the 'actorOf' factory methods to create a new actor. See the documentation.
```

```
at akka.actor.ActorInitializationException$.apply(Actor.scala:165)
at akka.actor.Actor$class.$init$(Actor.scala:421)
at akka.actor.UntypedActor.<init>(UntypedActor.scala:97)
at org.study.ch2.PingActor.<init>(PingActor.java:9)
at org.study.AkkaStudyWithBaekjunLimApplication.main(AkkaStudyWithBaekjunLimApplication.java:20)
```

PingActor는 UntypedActor라는 아카의 부모 클래스를 상속하고 있어서 일반적인 자바 클래스처럼 생성자를 호출하는 것이 금지되어 있다. 액터는 오직 Props라는 조리법을 통해서만 만들 수 있고, ActorRef라는 참조를 통해서만 접근할 수 있다. 아무 때나 new를 호출해서 객체를 만들고, 객체의 메서드에 마음껏 접근하던 ‘과거’는 잊어야 한다. 일단 액터 세계에 발을 담근 이상, 액터의 규칙을 준수해야 함을 명심하라.

장소 투명성

장소 투명성(Location Transparency)은 아카에서 가장 중요한 개념의 하나다. 장소 투명성은 내가 사용하는 액터가 물리적으로 어디에 존재하는지 알 필요가 없음을 가리키는 개념이다. 앞에서 본 것처럼 PingActor는 ActorRef에 둘러싸여서 우리 눈 앞에서 사라졌다. 우리는 오직 ActorRef를 가지고 작업을 수행할 뿐이기 때문에 PingActor가 어디에 있는지 알 필요가 없다.

ActorRef로 둘러싸인 액터는 클라이언트 코드와 동일한 JVM에 존재할 수도 있고, 동일한 컴퓨터의 다른 JVM에 존재할 수도 있고, 아예 다른 컴퓨터의 JVM에 존재할 수도 있다. 액터가 어디에 존재하든 그것은 ActorRef를 소비하는 우리의 방식에 아무런 영향을 주지 않는다.

모든 액터를 하나의 JVM 위에서 실행하는 간단한 애플리케이션이나 데모 코드의 경우에는 장소 투명성이 큰 의미를 갖지 않는다. 하지만 여러 개의 JVM이 클러스터를 형성하는 클러스트링의 경우에는 위력을 나타낸다. 모든 코드가 ActorRef를 대상으로 작성되어 있기 때문에 실제 액터의 물리적인 장소를 (클라이언트 코드에 아무런 영향을 주지 않으면서) 자유롭게 옮길 수 있는 것이다. 이런 부분은 뒤에서 클러스터를 공부할 때 더 자세히 살펴볼 것이다.

메시지 전송

main 메서드의 마지막 줄은 다음과 같은 모습을 가지고 있다.

```
ping.tell("start", ActorRef.noSender());
```

이것은 ping이라는 ActorRef 객체에 또는 pingActor라는 이름을 가진 액터에게 'start'라는 문자열 객체를 메시지로 전송하는 코드다. 일반적인 객체지향 프로그래밍에서는 '객체를 만든 다음 그 객체가 가지고 있는 메서드를 호출'한다. 이

것이 가장 기본적인 프로그래밍 방법이다. 그에 비해 액터 세계에서는 ‘액터를 만든 다음 그 액터에 메시지를 전송’하는 것이 가장 기본적인 프로그래밍 방법이다.

ActorRef는 tell과 ask라는 두 개의 API를 제공한다. ask는 퓨처Future와 관련된 개념인데, 뒤에서 살펴보고 여기에서는 액터 프로그래밍에서 가장 핵심적인 API인 tell 메서드에 대해서 살펴보도록 한다.

ActorRef의 tell 메서드는 다음과 같은 두 개의 값을 인수로 받아들인다.

- 메시지(message)
- 발신자(sender)

편지와 비슷하다. 편지를 전송할 때 편지봉투 안에 메시지를 담고, 곁면에 보내는 사람의 주소를 적는다. 그것과 개념적으로 다를 것이 없다. 이 코드에서 메시지는 ‘start’라는 문자열 객체고, 발신자는 ActorRef.noSender()다.

첫 번째 인수인 메시지는 임의의 자바 객체(즉, Object)가 될 수 있다. 우리가 PingActor와 PongActor 클래스를 정의할 때 ‘extends UntypedActor’라고 쓴 것을 확인하기 바란다. 이것은 곧 액터가 받아들이는 메시지의 타입이 미리 정해져 있지 않음을 의미한다. 그렇기 때문에 ActorRef 객체가 있을 때 tell 메서드를 이용해 어떤 타입의 객체라도 보낼 수 있다.

이 책에서는 주로 문자열을 메시지로 주고받는데, 실전 코드에서는 문자열 대신 애플리케이션에 적합한 메시지 타입을 별도로 정의해 사용하는 것이 일반적이다. 문자열을 메시지로 사용하는 것은 실전에서는 권장할 만한 방법이 아니며 단지 예제 코드를 간단하게 만들기 위해서 사용한다.

메시지가 임의의 객체일 수 있는 데 비해 발신자의 주소는 반드시 ActorRef 타입을 가져야 한다. 즉, 발신자는 어떤 액터다. 발신자 주소를 위해서는 보통 다음과 같은 네 가지 형태의 값이 사용된다.

- **getSelf()** 이 메서드는 액터 자신의 ActorRef 객체를 리턴한다. 따라서 메시지를 보내는 액터 자신을 발신자로 정할 때 사용한다. 가장 일반적인 형태라고 볼 수 있다.
- **getSender()** 이것은 현재 처리 중인 메시지를 보내 온 발신인의 주소를 다음 액터에게 그대로 포워드 forward 할 때 사용한다. 설명만으로는 이해하기 어려울 수도 있는데, 코드를 보면 쉽게 이해할 수 있을 것이다. 액터 사이에 질의-응답 관계를 형성하려 할 때 발신자의 주소를 포워드 하는 경우가 많은데, 고급 기법에 속하므로 지금은 몰라도 상관없다.
- 어떤 특정 액터를 가리키는 ActorRef 객체가 있다면 그것을 발신자로 정할 수도 있다. 실제 발신인의 주소를 다른 액터의 주소로 변경한다는 점에서 앞에서 본 getSender() 와 개념적으로 비슷하다.
- **ActorRef.noSender()** 이 메서드는 개념적으로 null에 해당하는 액터 주소를 리턴한다. 발신인 주소가 아무 의미가 없는 경우에 사용한다. main 클래스의 코드에서 우리는 ActorRef.noSender() 를 사용했다.

지금까지 살펴 본 내용을 바탕으로 main 메서드를 다시 읽어보면 전체적인 의미가 어렵지 않게 이해될 것이다.

```
ActorSystem actorSystem = ActorSystem.create("TestSystem");
ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class), "pingActor");
ping.tell("start", ActorRef.noSender());
```

이 코드는 'TestSystem'이라는 이름을 사용하는 액터시스템과 PingActor 클래스를 이용하는 액터를 만들고, 이 액터에 'start'라는 메시지를 전송한다. 이제 PingActor가 이 메시지를 전달받았을 때 어떤 일이 일어나는지 살펴보자.

메일박스

액터에 어떤 메시지를 전송하면 그것은 액터마다 가지고 있는 메일박스, 즉 우편함에 저장된다. 때에 따라서 여러 개의 액터가 하나의 메일박스를 공유하는 경우도 있는데, 지금은 일단 액터가 자기 자신의 고유한 메일박스를 가진다고 생각하는 게 좋다.

아카 라이브러리를 사용할 때 액터의 메일박스를 염두에 두는 것은 메모리 사용량이나 역류^{back-pressure} 등을 고민할 때뿐이다. 일반 애플리케이션에서는 액터의 메일박스에 접근할 방법이 없으므로 메일박스의 존재를 신경 쓰는 경우가 거의 없다. 즉, 메일박스는 애플리케이션 코드에서는 보이지 않는 투명한 존재다.

아카가 기본으로 제공하는 메일박스는 자바의 ConcurrentLinkedQueue를 이용해 구현되었는데 메일박스의 종류에 따라서 다른 클래스를 이용하기도 한다. 대부분의 경우에는 기본적인 메일박스를 이용해도 충분하지만, 실전 코드에서는 애플리케이션의 쓰임새에 맞는 최적의 메일박스 타입을 골라서 다양한 속성을 정의하는 것이 필요한 경우도 있다.

액터 하나를 떼어놓고 생각해보면 그것은 리액터^{reactor} 패턴 혹은 Node.js의 이벤트 처리 방식과 비슷하다. 자바 스윙 같은 GUI 시스템에 익숙한 사람이라면 이벤트 큐^{event queue} 혹은 디스패치 큐^{dispatch queue}를 생각해도 좋다. 원리적으로 이 모든 패턴은 비슷한 철학을 공유한다. 전달되는 메시지가 메일박스에 차곡차곡 쌓이고, 액터는 그 메시지를 한 번에 하나씩 처리하는 것이다.

onReceive

액터시스템은 내부에 스레드풀을 보유하고 있다. 디스패처^{dispatcher}라고 부르기도 하는데, 액터시스템은 일정한 규칙에 따라 스레드를 액터에게 할당한다. 이렇게 스레드가 할당된 액터는 (예를 들어, 메일박스에 있는 메시지를 100개 처리하거나 메일박스에 메시지가 남아 있지 않을 때까지) 메시지를 처리하고 스레드를 풀로 반납한다. 액터 시스템은 반납된 스레드를 다른 액터에게 할당하는 방식으로 작업을 이어나간다.

액터에 스레드가 할당되고, 메시지를 처리하고, 스레드를 풀에 반납하고, 스레드가 다른 액터에 할당되는 일련의 리듬은 액터시스템의 심장박동에 해당한다. 액터는 기본적으로 메모리 사용량이 300~400바이트에 불과한 가벼운 객체이기 때

문에 하나의 JVM 안에서 (메모리만 충분하면) 수천, 수만, 수백만 개가 생성되어도 상관없다. 하지만 액터의 수가 아무리 많아도 동시에 실행될 수 있는 스레드의 수에는 한계가 있다.

자바 스레드는 보통 512KB에서 2MB까지의 메모리를 사용하기 때문에 하나의 JVM 안에서 생성될 수 있는 스레드의 수는 수천 개 정도가 한계다. ‘동시에’ 실행되는 스레드의 수는 CPU 코어의 숫자에 의해서 더욱 심각하게 제약된다. 그렇기 때문에 액터의 수와 스레드의 수 사이에는 불균형이 존재한다. 일반적으로 액터의 수가 훨씬 많다고 보면 맞을 것이다.

액터는 디스패처에 의해서 스레드가 할당되어야만 동작을 수행할 수 있다. 스레드가 할당되지 않은 상태에서는 (당연히) 액터가 아무런 일도 수행하지 않는다. 그래서 아카의 디스패처는 수많은 액터를 대상으로 풀 내부에 존재하는 스레드를 최대한 공정한 방식으로 할당하기 위해 애를 쓴다.

이러한 과정을 거쳐서 어느 액터에 스레드가 할당되면 다음 같은 일이 일어난다.

- 메일박스에서 메시지를 하나 꺼낸다.
- 액터의 `onReceive` 메서드를 호출하면서 메시지를 인수로 전달한다.
- `onReceive`의 동작이 완료된다.
- 스레드를 반납할 시간이 되었는지를 확인한다.
- 시간이 되었으면 스레드를 반납하고 다음 순서를 기다린다.
- 아직 시간이 되지 않았으면 메일박스에서 다음 메시지를 꺼내고 위 과정을 반복한다.

이러한 메시지 처리 방식 자체는 이해하는 것이 어렵지 않다. 하지만 여기에서 반드시 짚고 넘어가야 하는 매우 중요한 사항이 있다. 그것은 다음 두 가지다.

- 어느 특정한 시점에서 보았을 때 액터의 `onReceive` 메서드를 호출하는 스레드는 반드시 1개로 국한된다. 디스패처가 하나의 액터에 2개 또는 그 이상의 스레드를 동시에 할당하지 않기 때문이다. 아카 혹은 액터시스템에서 전통적인 멀티스레드 환경문제를 고민하지 않아도 되는 이유는 바로 이러한 보장 때문이다.

- 하지만 액터의 onReceive 메서드를 호출하는 스레드가 영원히 똑같은 스레드인 것은 아니다. 하나의 시점에서 onReceive를 호출하는 스레드는 반드시 1개로 국한되지만, 시점이 달라지면 똑같은 액터의 onReceive를 실행하는 스레드가 얼마든지 다른 스레드로 달라질 수 있다.

다시 코드로 돌아가 보자. 앞에서 ping.tell("start", ActorRef.noSender())라는 코드를 이용해 평액터에 'start'라는 메시지를 전송했다. 이 메시지는 앞에서 살펴본 일련의 과정을 거쳐 PingActor 클래스가 정의하는 onReceive 메서드에 전달된다. 그리고 onReceive 메서드는 다음과 같은 동작을 수행한다.

```
@Override  
public void onReceive(Object message) throws Exception {  
    if (message instanceof String) {  
        String msg = (String)message;  
        log.info("Ping received {}", msg);  
        pong.tell("ping", getSelf());  
    }  
}
```

평액터는 전송된 메시지가 String 타입인지를 검사하고, String 타입이면 메시지를 화면에 출력한다. 그 다음 pong이라는 액터에 'ping'이라는 메시지를 전송한다. 이때 getSelf()라는 ActorRef를 발신인의 주소로 포함한다. 평액터 내부에서 getSelf()를 호출했으므로 getSelf() 메서드가 리턴하는 ActorRef는 평액터 자신을 가리키는 객체다.

한편 풍액터의 onReceive 메서드는 다음과 같은 동작을 수행한다.

```
@Override  
public void onReceive(Object message) throws Exception {  
    if (message instanceof String) {  
        String msg = (String)message;  
        log.info("Pong received {}", msg);  
    }  
}
```

```
ping.tell("pong", getSelf());
Thread.sleep(1000);
}
}
```

핑액터가 보낸 ‘ping’이라는 메시지는 퐁액터의 `onReceive` 메서드에 전송된다. 퐁액터는 메시지가 `String` 타입인지 확인하고, `String` 타입이면 화면에 메시지를 출력한 후 다시 평액터에 ‘pong’이라는 메시지를 전송한다. 마찬가지로 발신인은 `getSelf()`, 즉 퐁액터 자신을 가리키는 `ActorRef`다.

한 가지 주의할 점이 있다. 퐁액터의 코드 마지막 부분에 `Thread.sleep(1000)`이라는 코드가 포함되어 있는데, 이것은 프로그램을 실행했을 때 화면에 출력되는 내용을 눈으로 쫓아갈 수 있도록 만들기 위한 것이다. 실제 액터 코드에서 `Thread.sleep`을 호출하는 것은 상상조차 할 수 없는 금기다. 액터의 철학은 기본적으로 모든 것이 비동기적이고, 어느 것도 중단^{blocking}되지 않는다는 데 있다. `Thread.sleep`과 같은 메서드는 그것을 호출하는 현재 스레드를 중단되도록 만들기 때문에 액터의 철학에 정면으로 위배한다. 이 책의 예제들이 `Thread.sleep`을 포함하고 있다고 해서 실제 아카 코드에서 그래도 괜찮다고 착각하는 일이 없기를 바란다.

액터 라이프사이클

핑액터는 퐁액터와 달리 `preStart()`라는 메서드를 정의하고 있다. 이 메서드는 `PingActor`의 생성자가 완료된 다음, 하지만 첫 번째 `onReceive` 호출이 일어나기 전에 실행되는 코드다.

```
@Override
public void preStart() {
```

```
this.pong = context().actorOf(  
    Props.create(PongActor.class, getSelf()), "pongActor");  
}
```

`preStart()` 내부에 PongActor를 만들었다. 이 코드는 앞에서 PingActor를 만들 때 보았던 내용과 비교해 두 가지 다른 점을 가지고 있다.

- `actorOf`를 `systemActor`가 아니라 `context()`로부터 호출하고 있다.
- `Props.create()` 메서드가 인수 두 개를 받아들이고 있다.

우선 `context()`가 무엇인지 알아보자. 모든 액터는 자신이 실행되는 액터시스템의 환경이나 문맥, 즉 `ActorContext`에 접근할 수 있는 `context()` 메서드를 포함하고 있다. `ActorContext`는 해당 액터의 관점에서 보이는 `ActorSystem`의 모습을 나타내는 객체라고 생각하면 충분하다.

`Props.create` 메서드를 호출할 때 첫 번째 인수는 언제나 만들려는 액터를 위한 타입(예, `PongActor.class`)이어야 한다. 그 다음에 나열되는 인수는 `PongActor` 클래스의 생성자를 호출할 때 전달되는 인수를 의미한다. 액터 클래스의 생성자가 아무런 인수를 받아들이지 않는 경우에는 `Props.create`가 받아들이는 인수는 하나로 국한된다. 예를 들어, `PingActor`의 경우에 `Props.create(PingActor.class)`처럼 `create` 메서드를 호출할 때 인수를 하나만 전달했다.

하지만 `PongActor` 클래스는 생성자가 `ActorRef` 타입을 인수로 받아들인다. 따라서 `Props.create(PongActor.class, getSelf())`에서 보는 바와 같이 `create` 메서드를 호출할 때 인수를 두 개 전달했다. 여기에서 `getSelf()`는 평액터 자신을 가리키는 `ActorRef`고, 이것은 `PongActor`가 만들어질 때 `PongActor`의 생성자에 인수로 전달된다. `PongActor`의 생성자는 다음과 같이 `ActorRef`를 인수로 받아들인다.

```
public PongActor(ActorRef ping) {  
    this.ping = ping;  
}
```

이쯤에서 다시 PingActor.java와 PongActor.java의 내용을 읽어보면 코드의 의미가 잘 이해될 것이다.

다시 `preStart()` 메서드로 돌아가서 이야기하면, 모든 액터는 다음과 같은 라이프사이클을 가지고 있다.

- 생성 (created)
- 재시작 (restarted)
- 멈춤 (stopped)

더 자세한 라이프사이클은 나중에 살펴보기로 하고, 여기에서는 일단 세 가지 기본적인 단계에 관해서만 이야기하겠다. 액터가 재시작하는 경우는 액터시스템의 어딘가에서 에러가 발생해 액터 인스턴스가 새로 만들어지는 것을 의미한다. 다시 말해, `ActorRef`가 내부에 저장된 액터 객체를 버리고, 동일한 객체를 클래스의 생성자를 호출함으로써 액터를 새로 만드는 경우다. 액터를 소비하는 클라이언트 코드는 해당 액터를 둘러싸고 있는 `ActorRef`만 상대하므로 이러한 재시작 과정이 일어난다는 사실을 알 수 없고, 알 필요도 없다.

멈춤은 액터 라이프사이클의 최종적인 종료를 의미한다. 모든 액터는 라이프사이클과 관련된 메서드로서 다음과 같은 메서드를 가지고 있다.

- `preStart` 생성자가 완료된 직후에 실행된다.
- `preStop` 액터의 라이프사이클이 완전히 종료하기 직전에 실행된다.
- `preRestart` 재시작이 이루어지기 직전에 실행된다.
- `postRestart` 재시작이 이루어진 직후에 실행된다.

아카 계층구조

아카 계층구조

칼 휴이트 Carl Hewitt가 1973년에 이야기한 액터의 정의에 따르면, 어떤 액터는 다음과 같은 세 가지 동작을 수행할 수 있다.

- 메시지를 수신한다.
- 메시지를 송신한다.
- 다른 액터를 만든다.

앞 장에서 PingActor와 PongActor의 예를 통해 액터가 ActorRef의 tell 메서드를 이용해 메시지를 전송하는 것, onReceive 메서드를 통해서 메시지를 수신하는 것, PingActor가 PongActor를 만드는 것까지 확인했다. 휴이트가 정의한 내용을 모두 확인한 셈이다.

핑액터가 풍액터의 부모 액터고, 풍액터가 핑액터의 자식 액터라는 사실도 언급했다. 이미 설명한 대로 어떤 액터가 다른 액터를 만들면 두 액터 사이에는 부모-자식 관계가 형성된다. 그래서 하나의 액터시스템 안에서 만들어진 수많은 액터는 거대한 트리구조를 형성한다. 즉, 계층구조를 만드는 것이다. 이번 장에서는 액터가 다른 액터를 만들면서 형성하는 계층구조에 초점을 맞추고 살펴보겠다.

우선 3장의 내용을 공부하기 위한 소스코드 예제부터 살펴보자. 먼저 메인 함수를 담고 있는 메인 클래스의 전문이다.

```
package org.study;

import org.study.actor.PingActor;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 아카의 계층구조를 보여주기 위한 메인 클래스
 * @author Baekjun Lim
 */
public class Main {
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("TestSystem");
        ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class),
        "pingActor");
        ping.tell("work", ActorRef.noSender());
    }
}
```

다음은 계층구조에서 가장 상위에 위치한 PingActor의 내용이다. Ping1Actor라는 자식 액터를 만들고 있음에 주목하기 바란다.

```
package org.study.actor;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 계층구조 트리의 루트에 해당하는 최상위 부모 액터
 * @author Baekjun Lim
 */
public class PingActor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef child;
    private int count = 0;
```

```

public PingActor() {
    child = context().actorOf(Props.create(Ping1Actor.class), "ping1Actor");
}

@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if ("work".equals(msg)) {
            child.tell(msg, getSelf());
        }
        else if ("done".equals(msg)) {
            if (count == 0) {
                count++;
            } else {
                log.info("all works are completed.");
                count = 0;
            }
        }
    }
}
}

```

그 다음에는 평액터의 자식 액터인 Ping1Actor의 코드다. Ping2Actor와 Ping3Actor라는 두 개의 자식 액터를 생성하고 있음에 주목하라.

```

package org.study.actor;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 루트(PingActor)의 자식노드에 해당하는 액터.
 * Ping2Actor와 Ping3Actor라는 두 개의 자식노드 액터를 생성.
 * @author Baekjun Lim
 */
public class Ping1Actor extends UntypedActor {

```

```

private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
private ActorRef child1;
private ActorRef child2;
public Ping1Actor() {
    child1 = context().actorOf(Props.create(Ping2Actor.class), "ping2Actor");
    child2 = context().actorOf(Props.create(Ping3Actor.class), "ping3Actor");
}

@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if ("work".equals(msg)) {
            log.info("Ping1 received {}", msg);
            child1.tell("work", getSender());
            child2.tell("work", getSender());
        }
    }
}
}

```

끝으로 Ping2Actor와 Ping3Actor는 더 이상 자식 액터를 생성하지 않고 어떤 업무를 수행하는 액터다. 업무의 내용은 실전에서는 결코 사용되지 않아야 하는 Thread.sleep로 시뮬레이션 되고 있다. Ping2Actor와 Ping3Actor의 내용은 거의 동일하므로 Ping2Actor의 코드만 포함시켰다.

```

package org.study.actor;

import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 계층구조 트리의 최하위 leaf에 해당하는 액터
 * @author Baekjun Lim
 */
public class Ping2Actor extends UntypedActor {
    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    @Override

```

```

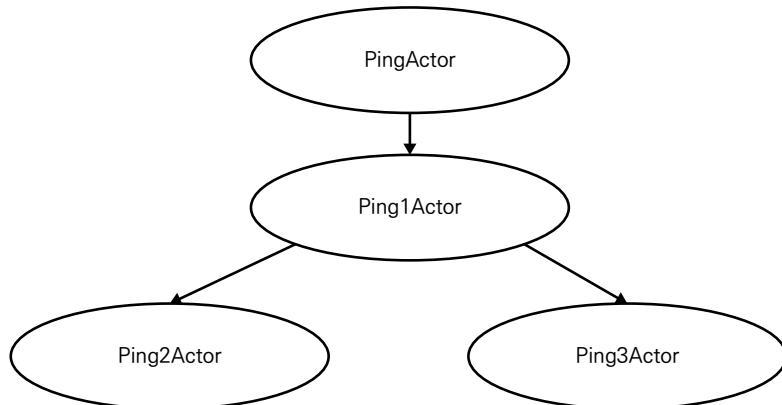
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if ("work".equals(msg)) {
            log.info("Ping2 received {}", msg);
            work();
            getSender().tell("done", getSelf());
        }
    }
}

private void work() throws Exception {
    Thread.sleep(1000); // 실전에서는 절대 금물!!!
    log.info("Ping2 working...");
}
}

```

지금까지 만든 액터를 액터시스템 전체에서 바라보면 그림처럼 간단한 트리를 형성한다. 이 구조는 뒤에서도 사용되니까 머릿속에 잘 기억해두기 바란다. 이것은 의도적으로 이해하기 쉽게 만든 예이기 때문에 트리전체가 4개의 액터만으로 구성되어 있다. 하지만 실전 코드에서는 수 천, 수 만개의 액터가 한 눈에 파악하기 어려운 보잡한 계층구조를 만들어내는 경우가 흔하다는 점을 기억하기 바란다.

그림 3-1



모든 코드가 준비되었으면 Main 객체를 실행하라. 그러면 다음과 같은 결과가 화면에 출력될 것이다.

```
[akka://TestSystem/user/pingActor/ping1Actor] Ping1 received work  
[akka://TestSystem/user/pingActor/ping1Actor/ping2Actor] Ping2 received work  
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3 received work  
[akka://TestSystem/user/pingActor/ping1Actor/ping2Actor] Ping2 working...  
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3 working...  
[akka://TestSystem/user/pingActor] all works are completed.
```

main 메서드에서 핑액터를 향해서 보낸 ‘work’라는 메시지가 핑액터에 전달되면, 핑액터는 자신의 자식인 Ping1Actor에 메시지를 전송하고, 그것은 다시 Ping1Actor의 두 자식인 Ping2Actor와 Ping3Actor에 전달된다. 화면에 출력된 내용은 그러한 내용을 순서대로 보여주고 있다. (나중에 살펴보겠지만 이러한 순서는 결코 보장된 순서가 아니다. 즉, 언제나 이와 동일한 순서로 출력된다는 보장이 없다.)

Ping2Actor는 ‘업무’가 완료되면 ‘work’ 메시지를 보낸 sender를 향해서 ‘done’이라는 메시지를 보낸다. Ping3Actor도 마찬가지다. 이러한 완료 메시지는 다시 PingActor에게 전달되어 최종적으로 화면에 ‘all works are completed.’라는 메시지가 출력된다.

연습문제

Ping2Actor와 Ping3Actor의 부모인 Ping1Actor는 ‘done’이라는 메시지를 처리하지 않는다. 그런데 어떻게 ‘done’ 메시지가 Ping2Actor와 Ping3Actor의 ‘할아버지’ 액터인 PingActor에게 전달될 수 있었을까?

힌트 Ping2Actor와 Ping3Actor가 보낸 ‘done’ 메시지는 누구에게 전달되는 것일까?

보내고 잊기

액터시스템에서 tell 메서드를 이용해서 메시지를 전달하는 것은 철저하게 비동기적^{asynchronous}이다. tell 메서드를 호출하면 해당 메시지가 실제로 목적지에 도달하는가 여부와 무관하게 즉시 리턴되기 때문에 ‘보내고 잊기(fire and forget)’라고 일컬어진다. 보내고 잊기는 액터시스템을 구성하는 핵심적인 원리의 하나다.

이것은 실제로 편지를 보내는 과정과 비슷하다. 편지를 보낼 때 우체통에 편지를 집어넣고 자리를 떠난다. 편지가 실제로 전송되는지를 생각하지 않고 다른 일을 시작한다. 즉, 편지를 보내고 잊어버린다.

편지를 보내는 과정이 만약 동기적^{synchronous}이라면 우리는 우체통에 편지를 넣은 다음, 우편배달부 아저씨가 도착해서 편지를 수거하고, 편지가 우체국에서 정상적으로 분류되고, 분류된 편지가 최종적으로 수신인의 손에 들어가기까지 다른 일을 할 수 없다. 예를 들어, obj.foo()처럼 우리가 일반적으로 사용하는 메서드 호출은 동기적이다. 메서드를 호출한 스레드는 해당 메서드가 리턴할 때까지 아무 일도 수행하지 못하고 기다려야 한다. 하지만 아카에서 메시지를 전송하는 동작은 철저하게 비동기적이다.

아카를 사용할 때 초급이나 중급 개발자들이 개념적으로는 이해하지만 실제로 제대로 구현하지 못하거나 자주 버그를 발생시키는 부분이 주로 이러한 비동기적인 작동 방식과 관련되어 있다. 지금까지 사용해온 동기적인 메서드 호출에 익숙해져 있기 때문에 모든 것이 비동기적으로 동작하는 커다란 패러다임의 변화를 몸으로 받아들이기 어려운 탓이다.

그런 사례를 보여주는 예를 두 개만 살펴보도록 하자. 비동기적인 동작을 동기적인 방식의 프로그래밍과 혼동하는 사례는 많지만 이 두 가지는 대표적이다. 하나는 ‘작업완료 확인’의 어려움이고 다른 하나는 ‘메시지 순서 확인’의 어려움이다. 하나씩 살펴보도록 하자.

작업의 완료

우리가 이번 장에서 생성한 액터의 계층구조에 의하면 main이 “work”라는 메시지를 PingActor에 전달하면, 핑액터는 그것을 자신의 자식인 Ping1Actor에 전달한다. 그러면 Ping1Actor는 자신의 자식인 Ping2Actor과 Ping3Actor에 메시지를 전달한다. Ping2Actor와 Ping3Actor가 수행하는 작업이 완료되면 각각은 “done”이라는 메시지를 sender에 전달한다.

PingActor는 “done” 메시지가 두 번 전달되면 모든 작업이 완료된 것으로 간주하고 화면에 최종 결과를 출력한다. 이러한 알고리즘을 따르면 PingActor가 최종결과를 출력하는 시점에서 Ping2Actor와 Ping3Actor의 작업이 실제로 완전히 완료되었음을 확신할 수 있다. 수학적인 증명도 가능할 것이다.

하지만 액터시스템을 처음 접하는 사람에게 이와 동일한 계층구조를 주고 업무가 종료되었을 때 최종결과를 화면에 출력해 보라고 시키면 조금 다른 코드를 작성하는 경우가 많다.

[Ping1Actor의 코드]

```
@Override  
public void onReceive(Object message) throws Exception {  
    if (message instanceof String) {  
        String msg = (String)message;  
        if ("work".equals(msg)) {  
            child1.tell("work", getSelf());  
            child2.tell("work", getSelf());  
            getSender().tell("done", getSelf());  
        }  
    }  
}
```

코드에서 볼 수 있는 것처럼 Ping1Actor의 onReceive에서 곧바로 sender(즉, PingActor)에 “done” 메시지를 전송하는 것이다.

코드를 이렇게 작성하면, PingActor가 Ping1Actor로부터 “done” 메시지를 전달 받았을 때 Ping2Actor와 Ping3Actor가 수행하는 작업이 모두 완료되었음을 확신할 수 있을까? 천만의 말씀이다. Ping1Actor가 보낸 “done” 메시지는 “work” 메시지를 Ping2Actor와 Ping3Actor에게 전달하는 작업이 수행되었음을 의미할 뿐, 그들이 실제로 수행하는 작업의 상태와는 아무런 상관이 없다. 이것이 모두 동기적인 메서드 호출이었다면 다르다.

```
class Ping {  
    public void foo() {  
        Ping1 p1 = new Ping1();  
        p1.work();  
        System.out.println("all works are completed.");  
    }  
}  
  
class Ping1 {  
    public void work() {  
        Ping2 p2 = new Ping2();  
        Ping3 p3 = new Ping3();  
        p2.work();  
        p3.work();  
    }  
}
```

이와 같은 동기적인 세계에서는 p1.work()라는 메서드 호출이 리턴하는 것은 p2.work()와 p3.work()가 모두 완료된 다음이다. 그렇기 때문에 우리는 내재되어^{nested} 있는 메서드 호출에 대해서 별도의 고민을 하지 않는다. 우리는 오랫동안 그런 방식에 길들여져 왔다. 하지만 비동기적인 세계에서는 이런 가정이 붕괴되어 해체된다. Ping1Actor가 자기 할 일을 다 했다고 말하는 것과, Ping2Actor와 Ping3Actor가 어떤 업무를 수행하는 것 사이에는 아무런 함수 관계가 없다. 모든 액터가 독립적이고 비동기적인 동작으로 분해되기 때문이다.

이제 메시지 순서 ordering에 대해 이야기할 차례다.

메시지 순서

앞에서 본 예제에서 메시지가 전송되고 수신되는 과정을 조금 수학적으로 정의해 보자.

P: PingActor

P1: Ping1Actor

P2: Ping2Actor

P3: Ping3Actor

w: “work” 메시지

d: “done” 메시지

(X m Y): 액터 X가 액터 Y에게 메시지 m을 보내는 동작

이렇게 기호를 정의하고 나면 예에서 등장하는 동작을 다음과 같이 설명할 수 있다. 여전히 Ping1Actor가 PingActor에 “done” 메시지를 보내는 코드를 염두에 두고 있다. (예제 코드에서는 Ping1Actor가 PingActor에 메시지를 보내지 않는다.)

- a. (P w P1) PingActor가 Ping1Actor에게 “work”를 전송
- b. (P1 w P2) Ping1Actor가 Ping2Actor에게 “work”를 전송
- c. (P1 w P3) Ping1Actor가 Ping3Actor에게 “work”를 전송
- d. (P1 d P) Ping1Actor가 PingActor에게 “done”을 전송
- e. (P2 d P) Ping2Actor가 PingActor에게 “done”을 전송
- f. (P3 d P) Ping3Actor가 PingActor에게 “done”을 전송

전체적으로 6번의 메시지 전송이 일어나고 있다. 간단하게 그림을 그려놓고 메시지 전송의 흐름을 손으로 따라가 보면 이해하기가 더 쉬울 것이다. 액터시스템의 초보자들이 가장 잘 저지르는 실수의 하나는 이렇게 (a)에서 (f)에 이르는 메시지의 흐름이 언제나 이런 순서로 진행될 거라고 믿는 것이다. 전혀 그렇지 않다. 대부분의 경우 메시지가 전달되는 순서는 비결정적(nondeterministic)이다.

예를 들어, (d)와 (e)와 (f) 사이에는 어떤 순서도 보장되어 있지 않다. 메시지가 (f), (e), (d)의 순서로 전달된다고 해도 전혀 놀라운 일이 아니다. 그렇기 때문에 방금 앞에서 보았던 것처럼 (d)가 일어났을 때 Ping2Actor와 Ping3Actor가 수행하는 작업이 완료되었음을 가정할 수 없는 것이다.

이렇게 생각하면 쉽다. 아카에서 보장하는 메시지의 순서는 오직 발신자와 수신자가 동일한 경우로 국한된다. 예를 들어서 A가 B에게 m1을 보내고, 잠시 후 m2를 보낸다고 하자.

1. (A m1 B)
2. (A m2 B)

이 경우에는 B의 입장에서 보았을 때 m1이 항상 m2보다 먼저 도착한다고 말할 수 있다. 하지만 이것을 제외한 모든 경우에는 순서가 보장되지 않는다. 다음 예를 보자.

1. (A m1 B)
2. (A m2 C)

A가 B에게 m1을 보내고 10초 뒤에 m2를 보냈다고 하자. 10초면 지나치게 극단적이긴 하지만 아무튼 논리적으로 따졌을 때 B가 m1을 수신하는 것이 항상 C가 m2를 수신하는 것보다 먼저 일어나는 일이라고 말할 수 있는 근거는 없다. 그럼 이것은 어떨까?

1. (A m1 B)
2. (B m1 C)
3. (A m2 B)
4. (B m2 C)

B는 A로부터 m1을 받아서 C로 전송하고, 다시 A로부터 m2를 받아서 C로 전송한다. A는 분명히 m1를 m2보다 먼저 전송했다. 이 경우에 C의 입장에서 m1이

항상 m2보다 먼저 도착할 수 있다고 말할 수 있을까? 당연히 아니다. A가 메시지를 직접 C에게 보내는 경우라면 그렇게 말할 수 있지만 이 경우에는 중간에 B라는 액터가 있기 때문에 어떤 순서도 보장되지 않는다.

akka.io에 있는 문서는 이것을 다음과 같이 설명하고 있다.

“어떤 액터의 짹(pair)이 있다고 했을 때, 첫 번째 액터가 두 번째 액터에게 직접 메시지를 보내는 경우에는 순서가 보장된다. 여기에서 ‘직접’이라는 말은 첫 번째 액터가 tell 메서드를 이용해서 최종 도착지인 두 번째 액터에게 메시지를 전송하는 경우만 그렇다는 사실을 강조하기 위함이다. 중간에 다른 액터가 끼어드는 경우에는 보장이 성립되지 않는다.”

액터 A1이 A2에게 m1, m2, m3라는 메시지를 보냈다.

액터 A3이 A2에게 m4, m5, m6라는 메시지를 보냈다.

이 경우에 보장되는 순서는 다음과 같은 경우로 국한된다.

1. m1이 도착한다면 그것은 반드시 m2와 m3보다 먼저 도착해야 한다.
2. m2가 도착한다면 그것은 반드시 m3보다 먼저 도착해야 한다.
3. m4가 도착한다면 그것은 반드시 m5와 m6보다 먼저 도착해야 한다.
4. m5가 도착한다면 그것은 반드시 m6보다 먼저 도착해야 한다.
5. A2는 A1이 보낸 메시지와 A3이 보낸 메시지가 섞여서 도착하는 것을 볼 수도 있다.
6. 메시지 전달이 보장되지 않기 때문에, 이러한 메시지 중에서 어떤 것이라도 중간에 사라질 수 있다. 즉, A2에게 전달되지 않을 수도 있다.

테크닉

메시지의 순서가 보장되지 않기 때문에 발생하는 혼란을 줄이기 위해서 사용할 수 있는 방법은 여러 가지가 있다. 어떤 방법이 최선인지는 주어진 문제에 따라서 다

르다. 여기에서는 우리가 이미 PingActor를 작성하면서 사용한 방법을 짚고 넘어가도록 하겠다.

다음 코드를 보자.

```
@Override  
public void onReceive(Object message) throws Exception {  
    if (message instanceof String) {  
        String msg = (String)message;  
        if ("work".equals(msg)) {  
            child.tell(msg, getSelf());  
        }  
        else if ("done".equals(msg)) {  
            if (count == 0) {  
                count++;  
            } else {  
                log.info("all works are completed.");  
                count = 0;  
            }  
        }  
    }  
}
```

PingActor는 내부에 count라는 변수를 선언한다. 실전 코드에서는 좀 더 정교한 데이터 구조를 사용할 수도 있는데, 아무튼 이것은 액터의 내부에 일정한 상태를 유지하는 방법이다. 즉, “work”라는 메시지를 아래로 downstream 내려 보냈으면 count의 값을 0으로 설정하고, “done”이라는 메시지가 전달될 때마다 count의 값을 증가시키는 방법이다. 물론 실전 코드에서 사용할 수 있는 완전한 방법은 아니다.

예를 들어서 “done” 메시지가 도착해서 count의 값을 +1만큼 증가시켰는데, 다시 “work”라는 메시지가 도착한다면 어떻게 해야 하는가? 실전에서는 이런 모든 상황을 염두에 둔 더 정교한 구조를 사용해야 할 것이다. 이 책에서 보여주는 코드

는 어디까지나 예제를 위해서 일부러 만든 간단한 내용이라는 사실을 잊지 않기 바란다.

두 번째로 생각할 부분은 Ping1Actor의 onReceive 안에 있다. 코드를 보자.

```
@Override  
public void onReceive(Object message) throws Exception {  
    if (message instanceof String) {  
        String msg = (String)message;  
        if ("work".equals(msg)) {  
            log.info("Ping1 received {}", msg);  
            child1.tell("work", getSender());  
            child2.tell("work", getSender());  
        }  
    }  
}
```

자식 액터에게 메시지를 보낼 때 발신자의 주소를 자기 자신(즉, getSelf())으로 정하지 않고, 자신의 입장에서 보았을 때 메시지 발신자에 해당하는 getSender()로 설정하고 있다. 이렇게 되면 Ping2Actor와 Ping3Actor의 입장에서 보았을 때 “work” 메시지를 보낸 액터가 부모인 Ping1Actor가 아니라 할아버지인 PingActor로 설정된다.

메시지를 단순히 다른 액터에게 전달하려면 사실 tell 메서드가 아니라 forward 메서드를 이용하는 편이 더 좋다. 하지만 여기에서 핵심은 그게 아니다. 어떤 액터의 입장에서 보았을 때 메시지를 보낸 발신자가 ‘물리적으로’ 메시지를 전송한 액터가 아닐 수도 있음을 깨닫는 것이 중요하다.

우리가 실제 편지를 받을 때도 그렇다. 나한테 편지를 전해주는 사람은 우편배달부 아저씨지만 편지 겉봉에 적혀있는 사람은 멀리 떨어져 있는 부모님이나 애인일 수 있다. 마찬가지로 tell 메서드의 두 번째 인수인 ActorRef에 들어가는 주소는 메시지를 물리적으로 전달해 주는 액터가 아닌 경우가 많다.

우리는 예제에서 PingActor의 주소를 직접 Ping2Actor와 Ping3Actor에게 전달해줌으로써 모든 작업이 완료되는 시점을 PingActor가 정확하게 인지할 수 있도록 만들었다. 무척 간단한 장난감 같은 코드지만 액터시스템을 이용해서 실전 코드를 작성하기 전에 이런 코드 안에서 이루어지고 있는 핵심적인 내용을 정확히 숙지하는 것이 대단히 중요하다.

액터의 내부 상태와 스레드

앞의 예에서 우리는 PingActor 내부에 count라는 정수형 변수를 선언했다. “done”이라는 메시지가 전달되었을 때 count의 값이 0이면 count를 1만큼 증가시키고, “done”이 전달되었을 때 count의 값이 0이 아니라면 모든 업무가 완료되었다고 판단하여 최종적인 결과를 화면에 출력하고 count의 값을 다시 0으로 설정한다.

count와 같은 객체 내부의 변수는 액터의 행위를 조절하는 상태^{state}에 해당한다. 실전 프로그래밍에서 액터는 대부분 이와 같은 내부 상태를 보유한다. 이 예에서는 count와 같은 정수형 변수를 사용하는 것보다 (예컨대 뒤에서 보게 되는 become, unbecome 메서드를 활용하는 식의) 더 효과적인 방법이 있기도 하지만 쉬운 설명을 위해서 count를 사용했다.

액터 프로그래밍을 처음 접하는 사람들은 count와 같은 변수의 타입을 선언할 때 단순히 int 타입이 아니라 AtomicInteger로 정하는 경우가 있다. Map을 만들 때는 HashMap이 아니라 ConcurrentHashMap을 사용한다. 멀티스레딩 환경에서의 안전성을 고려하는 것이다.

그렇게 한다고 해도 기능적으로 문제가 되지는 않지만, 액터에서 그러한 보호 장치는 불필요하다. 액터의 내부 상태에 접근하는 것은 (어떤 하나의 시점에서 보았을 때) 하나의 스레드로 국한되어 있기 때문에 액터의 내부 상태는 구태여 멀티스레딩을

위한 안전장치로 보호하지 않아도 상관없다.

액터와 스레드의 관계는 앞에서 이미 설명한 바 있다. 그 관계를 잘 이해한다면, 액터 내부의 상태에 접근하는 스레드가 언제나 하나의 스레드로 국한된다는 사실을 이해할 수 있을 것이다. 사람들이 동시성 코드를 작성하기 위한 차세대 방법으로 아카를 추천할 때 염두에 두는 내용이 바로 이 부분이다. 액터의 내부 상태에 접근하는 스레드는 언제나 1개로 국한되기 때문에 멀티스레드와 관련된 고민을 할 필요가 없어지는 것이다.

그렇다고 해서 아카가 멀티스레드와 관련된 문제를 모두 해결해주는 것은 전혀 아니다. 액터의 `onReceive` 메서드 내부에서 어떤 외부의 객체에 접근하는 경우가 얼마든지 있을 수 있다. 이런 경우에는 모든 것이 원점으로 되돌아온다. 그 객체가 멀티스레딩 문제로부터 보호되어 있다는 사실을 반드시 확인해야 하고, 보호되어 있지 않으면 스스로 보호방법을 강구해야 한다. 액터는 그런 경우에 대해서까지 문제를 해결해 주는 것이 아님을 잘 기억할 필요가 있다.

고장 나도록 허용하라

고장 나도록 허용하라

자바나 C#과 같은 언어에서 기본적인 에러 처리 전략은 try-catch 구문을 사용하는 것이다. 자바의 경우에는 실행타임^{runtime} 예외와 검사되는^{checked} 예외를 구분해서 사용하기도 하는데, 실전 코드는 대개 catch (Exception ex)와 같은 방식으로 작성된다. 모든 예외를 잡아서 처리하기 때문에 예외의 종류를 구분하는 것이 무색해진다. 처리도 공통적이다. 예외가 저장하고 있는 스택 정보를 로그파일에 적어 넣는 게 전부다. catch 구문 내부에서 실제로 ‘처리’라고 부를 만한 일을 수행하는 코드를 만나는 일은 드물다. 예를 들어, 다음 코드를 생각해 보자.

```
int successCount;
public void foo() {
    try {
        successCount++;
        doSomethingStupid();
    } catch (Exception ex) {
        logger.error("에러발생", ex);
    }
}
```

foo()라는 메서드는 우선 successCount의 값을 1만큼 증가시키고 doSomethingStupid()라는 메서드를 호출한다. 이 메서드가 어떤 예외를 발생시키면 프로그

램 콘트롤은 catch 구문 내부로 진입한다. catch 구문이 로그파일에 에러가 발생했다는 사실을 적어 넣으면 메서드는 리턴된다. 뭐가 잘못되었다는 느낌이 들지 않는가?

foo() 메서드는 예외가 발생하여 비정상적인 방식으로 리턴되었으므로 successCount의 값을 증가시키지 말아야 한다. 그렇다면 catch 구문은 단순히 예외의 내용을 로그파일에 기록하는 것으로 그치는 것이 아니라 try 구문 내부에서 변경시켰을지도 모르는 내부 상태를 원상복귀 시키는 작업을 반드시 수행해 주어야 한다. 그것이 '처리'다.

이뿐만 아니다. 자바 프로그래밍을 하다 보면 예외를 계속 스택 위로 (혹은 아래로?) 보내는 코드를 쉽게 만나게 된다. 이런 식이다.

```
public void foo() throws Exception {
    try {
        successCount++;
        doSomethingStupid();
    } catch (Exception ex) {
        logger.error("에러발생", ex);
        throw ex;
    }
}
```

이런 식으로 작성한 코드의 결과는 어딘가 멀리 떨어져 있는 코드가 어디선가 전달된 예외를 '처리'해야 하는 사태에 직면하게 된다는 사실이다. 그 코드는 예외가 발생한 소스로부터 멀리 떨어져 있기 때문에 예외가 발생한 이유를 알기 어렵고, 그것을 어떻게 '처리'해야 하는지는 더욱 알기 어렵다. 그렇기 때문에 할 수 있는 일이 예외가 발생했다는 사실을 로그파일에 기록하는 것으로 국한된다.

칼 휴이트의 액터 모델을 최초로 구현한 얼랭은 “고장 나도록 허용하라”는 철학, 영어로 “Let It Crash”라고 부르는 전략을 취했다. 이것은 자바나 C#이 사용하

는 try-catch와 구별된다. 수상쩍은 코드를 try-catch 블록으로 일일이 감싸는 것이 아니라 예외가 발생하도록 그냥 내버려 두라는 전략이다. 예외를 처리하지 않겠다는 이야기가 아니다. 게으른 프로그래머가 예외를 100km 떨어진 곳으로 던져버릴 수 있는 자바의 try-catch와 달리 어느 액터가 발생시킨 예외는 그 액터의 감시자(supervisor)가 곧바로 처리하도록 만들겠다는 전략이다. 어떤 의미에서는 자바의 try-catch보다 더 공격적인 방법이다.

우선 이 장에서 살펴볼 예제 코드를 확인하도록 하자. 이번에는 메인 클래스가 두 개다. 하나는 BadMain이라는 클래스고, 다른 하나는 GoodMain이라는 클래스다. 다른 점은 핑액터에게 “good”을 보내는가 아니면 “bad”를 보내는가로 국한된다. 따라서 여기에는 GoodMain만 포함시켰다.

```
package org.study;

import org.study.actor.PingActor;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 아카의 Let It Crash 철학을 보여주기 위한 메인 클래스
 * @author Baekjun Lim
 */
public class GoodMain {
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("TestSystem");
        ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class),
        "pingActor");
        ping.tell("good", ActorRef.noSender());
    }
}
```

다음은 우리가 만드는 액터 계층구조에서 언제나 루트에 해당하는 PingActor다.

```
package org.study.actor;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * "good"이나 "bad" 메시지를 받으면 Ping1Actor라는 자식 액터에게 전달.
 * "done" 메시지를 받으면 화면에 결과를 출력.
 * @author Baekjun Lim
 */
public class PingActor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef child;

    public PingActor() {
        child = context().actorOf(Props.create(Ping1Actor.class), "ping1Actor");
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            String msg = (String)message;
            if ("good".equals(msg) || "bad".equals(msg)) {
                child.tell(msg, getSelf());
            }
            else if ("done".equals(msg)) {
                log.info("all works are successfully completed.");
            }
            else {
                unhandled(message);
            }
        }
    }
}
```

다음으로는 감시전략과 관련해서 중요한 내용을 담고 있는 Ping1Actor의 코드다. SupervisorStrategy라는 객체가 정의되는 부분을 주의해서 읽기 바란다. 자식 액터가 ArithmeticException을 발생시키면 resume()이라는 메서드를 호출하고, NullPointerException을 발생시키면 restart()라는 메서드를 호출한다.

```
package org.study.actor;

import static akka.actor.SupervisorStrategy.escalate;
import static akka.actor.SupervisorStrategy.restart;
import static akka.actor.SupervisorStrategy.resume;
import static akka.actor.SupervisorStrategy.stop;
import scala.concurrent.duration.Duration;
import akka.actor.ActorRef;
import akka.actor.OneForOneStrategy;
import akka.actor.Props;
import akka.actor.SupervisorStrategy;
import akka.actor.SupervisorStrategy.Directive;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.Function;

/**
 * 자식 액터들을 감시하기 위한 전략을 선언하는 액터
 * @author Baekjun Lim
 */
public class Ping1Actor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef child1;
    private ActorRef child2;

    public Ping1Actor() {
        child1 = context().actorOf(Props.create(Ping2Actor.class), "ping2Actor");
        child2 = context().actorOf(Props.create(Ping3Actor.class), "ping3Actor");
    }
}
```

```

@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if ("good".equals(msg) || "bad".equals(msg)) {
            log.info("Ping1Actor received {}", msg);
            child1.tell(msg, getSender());
            child2.tell(msg, getSender());
        }
    } else {
        unhandled(message);
    }
}

private static SupervisorStrategy strategy =
new OneForOneStrategy(10, Duration.create("1 minute"),
new Function<Throwable, Directive>() {
    @Override
    public Directive apply(Throwable t) throws Exception {
        if (t instanceof ArithmeticException) {
            // Ping2Actor는 "bad" 메시지를 받으면 ArithmeticException을 발생
            return resume();
        } else if (t instanceof NullPointerException) {
            // Ping3Actor는 "bad" 메시지를 받으면 NullPointerException을 발생
            return restart();
        } else if (t instanceof IllegalArgumentException) {
            return stop();
        } else {
            return escalate();
        }
    }
});

@Override
public SupervisorStrategy supervisorStrategy() {
    return strategy;
}

```

다음은 “bad”라는 메시지를 전달받았을 때 일부러 ArithmeticException을 발생시키는 Ping2Actor의 모습이다. 이 액터는 다시 시작^{restarted}될 때 화면에 메시지를 출력시킨다.

```
package org.study.actor;

import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

public class Ping2Actor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public Ping2Actor() {
        log.info("Ping2Actor constructgor..");
    }

    @Override
    public void preRestart(Throwable reason, scala.Option<Object> message) {
        log.info("Ping2Actor preRestart..");
    }

    @Override
    public void postRestart(Throwable reason) {
        log.info("Ping2Actor postRestart..");
    }

    @Override
    public void postStop() {
        log.info("Ping2Actor postStop..");
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            String msg = (String)message;
            if ("good".equals(msg)) {
                goodWork();
            }
            else if ("bad".equals(msg)) {
                badWork();
            }
        }
    }
}
```

```

        getSender().tell("done", getSelf());
    }
    else if ("bad".equals(msg)) {
        badWork();
    }
}
else {
    unhandled(message);
}
}

private void goodWork() throws Exception {
    log.info("Ping2Actor is good.");
}

/** 일부러 ArithmeticException을 발생시킨다 */
private void badWork() throws Exception {
    int a = 1 / 0;
}
}

```

끝으로 “bad”라는 메시지를 전달받았을 때 일부러 NullPointerException을 발생시키는 Ping3Actor의 모습이다. 전체적인 모습은 Ping2Actor와 거의 비슷하다. 이 액터도 다시 시작될 때 화면에 메시지를 출력시킨다.

```

package org.study.actor;

import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

public class Ping3Actor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public Ping3Actor() {
        log.info("Ping3Actor constructgor..");
    }
}

```

```

@Override
public void preRestart(Throwable reason, scala.Option<Object> message) {
    log.info("Ping3Actor preRestart..");
}

@Override
public void postRestart(Throwable reason) {
    log.info("Ping3Actor postRestart..");
}

@Override
public void postStop() {
    log.info("Ping3Actor postStop..");
}

@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if ("good".equals(msg)) {
            goodWork();
            getSender().tell("done", getSelf());
        }
        else if ("bad".equals(msg)) {
            badWork();
        }
    }
    else {
        unhandled(message);
    }
}

private void goodWork() throws Exception {
    log.info("Ping3Actor is good.");
}

/** 일부러 NullPointerException을 발생시킨다 */
private void badWork() throws Exception {
    throw new NullPointerException();
}

```

SupervisorStrategy

코드에서 보다시피 우리는 Ping2Actor와 Ping3Actor의 부모인 Ping1Actor에서 감시전략^{supervisor strategy}을 정의했다. 정의한 내용은 매우 간단하다.

- ArithmeticException이 발생하면 resume()을 호출하라.
- NullPointerException이 발생하면 restart()를 호출하라.
- IllegalArgumentException이 발생하면 stop()을 호출하라.
- 나머지 모든 예외에서 대해서는 escalate()를 호출하라.

onReceive 메서드의 시그너처를 보면 throws Exception이라는 자바의 문법과 함께 정의되어 있다. 그것은 onReceive에서 발생하는 예외가 onReceive를 호출하는 코드에게 던져진다는 뜻이다. 그러한 예외를 처리하는 코드는 물론 아카내부의 코드다. 이러한 아카내부의 코드는 예외를 포착했을 때 그 액터의 감시자supervisor가 감시전략을 가지고 있는지를 확인한다. 만약 (Ping1Actor의 경우처럼) 별도의 감시전략을 가지고 있으면 그 전략을 사용하고 그렇지 않으면 아카가 기본적으로 제공하는 전략을 사용한다.

어떤 액터가 자신만의 감시전략을 수립하려면 내부에 SupervisorStrategy 객체를 정의하고 다음 메서드를 통해서 기본전략을 오버라이드 override하면 된다.

```
@Override  
public SupervisorStrategy supervisorStrategy() {  
    return strategy;  
}
```

이러한 메커니즘이 존재하기 때문에 액터 코드를 작성하는 사람은 onReceive 내부에서 별도의 try-catch 구문을 선언하는데 시간을 할애하지 않아도 된다. 액터를 벗어난 예외는 반드시 감시자에게 전달이 되어 어떤 식으로든 ‘처리’가 되기 때문이다. 고장 나도록 허용하라는 철학의 배후에는 이렇게 “내가 알아서 처리해

줄께”라는 의도가 숨어있는 것이다.

아카에서 감시자는 해당 액터를 생성한 부모 액터다. 따라서 부모 액터는 모든 자식 액터의 감시자다. 우리가 반복적으로 사용하고 있는 계층구조를 다시 생각해보자. 아카가 제공하는 감시전략 메커니즘에 의하면 PingActor는 Ping1Actor의 감시자고, Ping1Actor는 Ping2Actor와 Ping3Actor의 감시자다. 따라서 우리가 Ping1Actor에서 정의한 감시전략은 Ping2Actor와 Ping3Actor를 감시하기 위해서 사용된다.

이제 BadMain을 실행해보라. 화면에 다음과 같은 내용이 출력될 것이다. “bad”라는 메시지가 전달되었을 때 Ping2Actor는 일부러 ArithmeticException을 발생시키고, Ping3Actor는 NullPointerException을 발생시킨다. 그리고 Ping1Actor에 정의한 감시전략에 따르면 ArithmeticException은 resume()으로 처리되고, NullPointerException은 restart()로 처리되어야 한다. 화면에 출력된 내용은 그러한 처리가 이루어진 결과다.

```
[akka://TestSystem/user/pingActor/ping1Actor/ping2Actor] Ping2Actor
constructgor..
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3Actor
constructgor..
[akka://TestSystem/user/pingActor/ping1Actor] Ping1Actor received bad
[akka://TestSystem/user/pingActor/ping1Actor/ping2Actor] / by zero
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] null
java.lang.NullPointerException
    at org.study.actor.Ping3Actor.badWork(Ping3Actor.java:53)
    at org.study.actor.Ping3Actor.onReceive(Ping3Actor.java:39)
    at akka.actor.UntypedActor$$anonfun$receive$1.applyOrElse(UntypedActor.
scala:167)
    at akka.actor.Actor$class.aroundReceive(Actor.scala:465)
    at akka.actor.UntypedActor.aroundReceive(UntypedActor.scala:97)
    at akka.actor.ActorCell.receiveMessage(ActorCell.scala:516)
    at akka.actor.ActorCell.invoke(ActorCell.scala:487)
    at akka.dispatch.Mailbox.processMailbox(Mailbox.scala:254)
    at akka.dispatch.Mailbox.run(Mailbox.scala:221)
```

```
at akka.dispatch.Mailbox.exec(Mailbox.scala:231)
at scala.concurrent.forkjoin.ForkJoinTask.doExec(ForkJoinTask.java:260)
at scala.concurrent.forkjoin.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.
java:1339)
at scala.concurrent.forkjoin.ForkJoinPool.runWorker(ForkJoinPool.
java:1979)
at scala.concurrent.forkjoin.ForkJoinWorkerThread.
run(ForkJoinWorkerThread.java:107)

[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3Actor
preRestart..
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3Actor
constructgor..
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3Actor
postRestart..
```

Resume

Resume은 예외를 발생시킨 메시지를 무시하고 메일박스에 있는 다음 메시지를 처리하라는 뜻이다. 즉, ArithmeticException이 발생하면 그 예외를 발생시킨 메시지(이 예에서는 “bad”라는 메시지)를 버리고, 메일박스에 있는 다음 메시지를 처리하라는 뜻이 된다. 그렇기 때문에 Ping2Actor의 경우에는 “/ by zero”라는 메시지가 화면에 출력되었을 뿐, 다른 일은 일어나지 않았다. 이제 Ping2Actor에게 다른 메시지가 전달되면 아무 일도 없었다는 듯이 onReceive 가 호출될 것이다.

Restart

하지만 Ping3Actor는 다르다. NullPointerException과 관련된 스택 정보를 화면에 출력하고 다음과 같은 메시지를 보여준다.

```
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3Actor
preRestart..
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3Actor
constructgor..
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3Actor
postRestart..
```

“Ping3Actor constructor”라는 메시지에서 알 수 있듯이 Ping3Actor 클래스의 생성자가 다시 호출되었다. 생성자가 호출되기 바로 직전에 preRestart가 호출되고, 직후에 postRestart가 호출되는 것도 알 수 있다. 이러한 메서드 호출 순서를 잘 숙지하는 것은 실전코드에서 액터의 라이프사이클을 관리할 때 도움이 된다.

생성자가 호출되었다는 사실로부터 유추할 수 있듯이 Ping3Actor의 객체는 감시자의 판단에 의해서 완전히 새로 만들어졌다. Ping3Actor 내부에서 관리되는 (맵이나 리스트 같은) 데이터 구조가 있었다면 그들도 모두 초기화가 될 것이다. 만약 그런 내부 데이터를 반드시 유지할 필요가 있다면 preRestart 메서드에서 데이터를 외부에 저장하고 생성자나 postRestart 메서드에서 불러들여야 할 것이다. 한 가지 기억할 점은 액터가 다시 시작된다고 해도 메일박스에 있던 메시지들은 영향을 받지 않는다는 점이다.

Ping1Actor는 Ping3Actor를 child2라는 변수를 이용해서 참조하고 있다. 변수의 타입은 물론 ActorRef다. NullPointerException이 발생했을 때 ActorRef 객체 내부에서 Ping3Actor라는 클래스의 인스턴스가 새로 만들어져서 낡은 인스턴스를 교체했다. 하지만 Ping1Actor는 여전히 그 액터를 child2라는 변수로 참조한다. 다음과 같은 코드는 Ping3Actor가 다시 시작되었다는 사실로부터 아무런 영향을 받지 않는다.

```
child2.tell("work", getSender());
```

새로운 인스턴스에게 자리를 빼앗긴 Ping3Actor의 낡은 인스턴스는 조금 뒤에 가비지 컬렉터에 의해서 수집되어 사라질 것이다.

이제 감이 오기 시작했을 것이다. 액터시스템을 사용하면서 단순히 resume()이 아니라 restart()를 사용하는 경우는 해당 액터 내부에서 관리되는 데이터가 예외가 발생하는 상황에 의해서 오염되었다고 판단될 때다. 앞에서 보았던 예를 생각해보기 바란다. try-catch 구문을 이용해서 예외를 잡아내고 자세한 내용을 로그 파일에 기록한다고 해도 successCount의 값이 오염되는 경우에는 추가적인 작업이 필요하다. 즉, successCount의 값을 다시 초기화하는 작업이 수반되어야 하는 것이다.

resume()이 아니라 restart()를 사용하는 것은 그러니까 일종의 ‘초기화’를 위한 전략이다. A라는 액터에서 X라는 예외가 발생하는 경우에는 A 내부의 값이 오염될 수 있다고 판단될 때, SupervisorStrategy를 이용해서 X라는 예외에 restart()라는 동작을 매핑해 준다.

Stop

stop()은 감시전략이 취할 수 있는 형별 중에서 가장 가혹한 극형에 해당한다. 말하자면 사형이다. stop에 의해서 동작이 중지된 액터는 다른 액터에 의해서 다시 생성되기 전까지 액터시스템 안에 존재할 수 없다. stop이라는 것은 액터가 갖는 라이프사이클에서 최후의 단계에 해당한다. 죽음이다.

액터는 preStop() 메서드를 구현함으로써 완전히 생명을 잃기 전에 필요한 일을 수행할 수 있다.

Escalate

액터의 감시전략은 resume(), restart(), 그리고 stop()에게 매핑되지 않은 예외가 발생하면 자신의 감시자, 즉 부모 액터에게 알린다. 그것이 escalate()가 수행하는 일이다.

기본 감시전략

액터를 구현하면서 감시전략을 별도로 정의하지 않으면 아카가 기본적으로 제공하는 전략이 사용된다. 기본적인 전략은 다음과 같이 정의되어 있다.

- ActorInitializationException stop()을 호출한다.
- ActorKilledException stop()을 호출한다.
- Exception restart()를 호출한다.
- Throwable escalate()를 호출한다.

일반적으로 발생하는 예외는 세 번째 Exception에 의해서 포착되므로, 감시전략을 정의하지 않은 액터의 자식은 대부분 restart()에 의해서 처리된다. 액터를 새로 만드는 것은 가볍고 빠르기 때문에 restart() 전략을 기본적으로 사용하는 것이 나쁘지 않은데, 빠른 성능에 대한 요구가 있거나 아니면 액터의 내부 상태를 굳이 초기화할 필요가 없다고 판단될 때는 resume()을 사용하는 것이 더 효율적이다.

재귀와 연대책임

감시전략과 관련해서 알아두어야 하는 내용이 더 있지만 여기에서는 두 가지 정도만 기억해두도록 하자. 하나는 감시전략이 가지고 있는 재귀의 측면이고, 또 다른 하나는 연대책임이라는 측면이다.

우선 어떤 액터가 예외를 발생시켜서 그 액터의 부모가 restart()를 호출했다고 하자. 새롭게 생성되는 액터는 그 자신이 다른 액터들의 부모일 수도 있다. 이런 액터가 생성된다면 그 액터가 거느리고 있는 자식 액터들도 모두 자동적으로 새로 생성된다. 즉, restart()는 재귀적인 방식으로 자식과 후손들에게 적용이 되는 것이다.

우리의 예제에서처럼 트리 전체가 4개의 액터로 이루어져 있으면 별로 생각할 것이 없다. 하지만 트리구조가 만 개, 십만 개의 액터로 이루어져 있다고 생각해보자. 이 경우에 트리의 상위에 위치하는 액터가 다시 시작하게 되면 그 아래에 있는 수 천, 수 만 개의 액터가 재귀적인 방식으로 다시 시작하게 된다. 아카는 해당 액터의 후손 액터들을 모두 중단^{stop}시키고 그들을 구성하는 객체를 새로 만들어서 저마다의 ActorRef에 집어넣는다.

그렇기 때문에 restart()는 단순히 액터 하나의 restart()가 아니다. 이러한 재귀적인 동작방식 때문에 그것은 미처 생각하지 못한 커다란 비용을 수반하는 동작이 될 수도 있다. 그래서 액터시스템을 이용해서 실전코드를 작성할 때는 이런 면들을 모두 고려할 수 있는 능력을 갖추어야 한다.

감시전략과 관련해서 알아두어야 하는 또 한 가지 사항은 감시전략이 어떤 예외를 포착해서 매픽되어 있는 메서드를 호출할 때 연대책임을 물을 수도 있다는 점이다. resume(), restart(), stop() 등의 메서드를 예외를 발생시킨 액터에게 국한해서 적용시킬 수도 있고 모든 자식 액터에게 (즉, 말썽을 부린 액터의 형제/자매 액터들에게) 한꺼번에 적용시킬 수도 있다. 우리가 예제에서 본 OneForOneStrategy는 말썽을 부린 액터에게만 별을 주는 방식이고, AllForOneStrategy는 연대책임을 묻는 방식이다.

커다란 트리구조에서 AllForOneStrategy가 restart()와 결합되어 사용된다면, 눈에 보이지 않는 비용이 더 커질 수도 있다는 점에 주의할 필요가 있다.

액터와 상태기계

액터의 상태

앞에서 보았던 코드를 다시 생각해보자. PingActor가 모든 작업이 완료되는 시점을 파악하기 위해서 사용한 방법이다. 단순한 예제를 위해서 정수형 변수가 사용되고 있다는 점을 빼면 실전에서도 종종 사용되는 구조를 구현하고 있다. 아카에 사용되는 일종의 패턴이다.

```
private int count = 0;

@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if ("work".equals(msg)) {
            child.tell(msg, getSelf());
        }
        else if ("done".equals(msg)) {
            if (count == 0) {
                count++;
            } else {
                log.info("all works are completed.");
                count = 0;
            }
        }
    }
}
```

기본 가정은 이렇다. 아래에 존재하는 액터에게 “work”라는 메시지를 전달하면 2개의 “done” 메시지가 전달될 것이다. 순서는 상관없다. “done” 메시지가 2개 도착하면 필요한 작업이 모두 완료된 것이다. 그래서 “done”이 첫 번째로 도착하면 count의 값을 1만큼 증가시키고, 두 번째로 도착하면 “all works are completed”라는 메시지를 출력한다. count 값은 다시 0으로 설정한다.

이런 방식으로 작성한 코드는 우리가 원하는 동작을 수행한다. 하지만 코드가 좀 지저분하다. 단순히 코드만 지저분한 것이 아니다. 기능적으로도 문제가 있다. 다음과 같은 상황을 생각해보자.

- **Main** PingActor에 “work”라는 메시지를 보낸다.
- **PingActor** Main으로부터 “work”라는 메시지가 전달되면 Ping1Actor에 전달한다.
- **Ping1Actor** PingActor로부터 “work”라는 메시지가 전달되면 Ping2Actor와 Ping3Actor에 전달한다.
- **Ping2Actor** 비즈니스 업무를 수행한다. 완료되면 PingActor에 “done”을 전달한다.
- **Ping3Actor** 비즈니스 업무를 수행한다. 완료되면 PingActor에 “done”을 전달한다.
- **PingActor** “done”을 두 번 전달받으면 Main으로부터 전달된 “work”라는 작업이 완성되었으므로 화면에 결과를 출력한다.

여기까지는 지금까지 보았던 상황과 별로 다를 것이 없다. 이 책의 내용을 계속 읽어온 사람이라면 이해하는데 아무런 어려움이 없을 것이다. 그런데 우리가 작성한 PingActor는 작업이 완성되는 과정을 모니터 하기 위해서 count라는 정수형 변수를 사용하고 있다. 이제 질문이다. 일련의 메시지가 다음과 같은 순서로 발생했다고 생각해보자. 모든 것을 PingActor의 관점에서 서술하고 있다.

1. PingActor가 Main으로부터 “work” 메시지를 수신했다.
2. PingActor가 Ping1Actor에게 “work”를 전송했다.
3. PingActor가 “done” 메시지를 수신했다.
4. PingActor가 Main으로부터 “work” 메시지를 수신했다.

5. PingActor가 “done” 메시지를 수신했다.

우리가 지금까지 생각해오던 깔끔한 시나리오에 4번 단계가 끼어들었다. 그럴 수 밖에 없는 것이 Main은 PingActor의 내부 상태에 대해서 아무런 관심이 없기 때문이다. 자기가 필요하면 아무 때나 “work” 메시지를 보낸다. 즉, PingActor에게 전달되는 두 가지 메시지 타입인 “work”와 “done”은 서로에 대해서 무관심한 채로 순서에 상관없이 도착한다.

문제는 다음과 같은 두 가지 요구사항의 충돌이다.

- “work”가 도착하면 Ping1Actor에게 전달한다. 이때 count의 값은 0이어야 한다.
- “done”이 도착했을 때, 만약 count의 값이 0이면 값을 1만큼 증가시킨다. 만약 1이면 최종 결과를 화면에 출력하고 count를 다시 0으로 설정한다.

PingActor의 코드는 “work”를 Ping1Actor에게 보냈을 때 count의 값이 0이어야 한다고 가정한다. 그게 문제다. 앞의 4번 단계에서 “work” 메시지를 수신했을 때 count의 값은 이미 1인 상태다. 그렇다고 해서 4번 단계에서 “work”를 수신했을 때 count를 0으로 설정하면 5번 단계에서 “done”을 수신했을 때 정상적인 상태의 추적이 불가능하게 된다. 그렇다고 해서 Ping1Actor에게 “work”를 보내고 나서 count의 값이 1인 상태로 내버려 두면 지금 보내는 “work”的 진행상황을 정상적으로 추적할 수 없게 된다. 이러한 충돌을 어떻게 해결해야 할까?

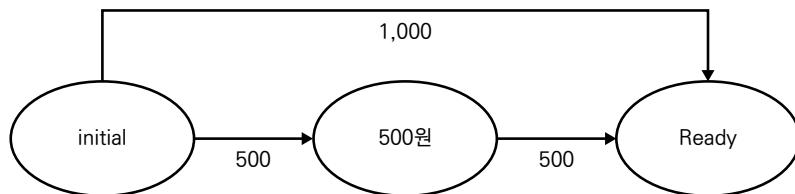
이럴 때 사용할 수 있는 방법이 바로 상태기계다.

상태기계

컴퓨터 과학에서 상태기계는 자주 등장하는 낯익은 개념이다. 쉽게 말해서 상태기계는 한 번에 한 개의 상태를 반영하는 기계를 의미한다. 흔히 사용하는 커피자동판매기는 대표적인 상태기계다. 예를 들어서 커피 한잔의 가격이 1,000원이라고 하자. 아직 동전을 집어넣지 않은 자판기는 초기상태^{initial state}를 반영하고 있다.

이제 500원짜리 동전을 하나 집어 넣으면 자판기는 500원이 입력된 상태를 반영 한다. 그 상태에서 500원짜리를 하나 더 넣으면 자판기는 이제 커피를 내보낼 준비가 되었다. 준비상태^{ready state}가 되는 것이다. 혹은 초기상태에서 1,000원을 넣으면 중간단계를 생략하고 곧바로 준비상태가 된다. 이러한 과정을 간략하게 그림으로 나타내면 이렇다.

그림 5-1

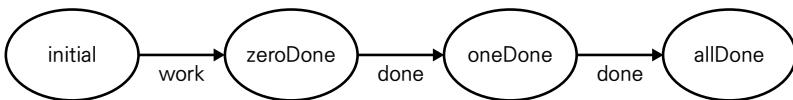


타원은 각각의 상태를 나타내고, 직선은 사건을 의미한다. 이것이 상태기계다. 우리가 앞에서 보았던 PingActor는 지저분한 if-else 구문 대신 상태기계를 이용하면 보다 깔끔하게 해결할 수 있는 문제다. 메서드를 동기적인 방식으로 직접 호출하는 것이 아니라 메시지를 전달하는 비동기적인 방식으로만 통제할 수 있는 액터 세계에서는 상태기계를 이용해서 해결할 수 있는 문제가 많다. 그래서 아카는 상태기계를 구현할 수 있는 편리한 API를 제공한다.

앞에서 보았던 PingActor의 상태는 대략 네 개의 상태로 나누어 볼 수 있다. 아직 아무 메시지도 전달되지 않은 초기상태, “work” 메시지가 전달되어 초기화가 시작된 상태, “done” 메시지가 한 번 전달된 상태, “done” 두 번 전달되어 작업이 완료된 상태, 이렇게 네 개의 상태가 존재하는 것이다. 이런 상태들에게 initial, zeroDone, oneDone, allDone이라는 이름을 붙이면 그림과 같은 상태기계로 표현할 수 있다.

개념적으로 보았을 때 초기상태(initial)와 최종상태(allDone)는 동일한 상태로 보아도 무방하지만 쉬운 이해를 위해서 분리해 놓았다.

그림 5-2



이제 원래 문제로 되돌아가 보자.

1. PingActor가 Main으로부터 “work” 메시지를 수신했다.
2. PingActor가 Ping1Actor에게 “work”를 전송했다.
3. PingActor가 “done” 메시지를 수신했다.
4. PingActor가 Main으로부터 “work” 메시지를 수신했다.
5. PingActor가 “done” 메시지를 수신했다.

각 단계별로 상태가 어떻게 변하는지 설명하면 다음과 같다.

1. 아직 변화 없음.
2. initial --> zeroDone
3. zeroDone --> oneDone
4. ??
5. oneDone --> allDone

문제는 현재 상태가 “oneDone”일 때 기다리고 있는 “done”이 아니라 “work”를 수신한다면 어떤 동작을 취해야 하는가이다. 간단하다. PingActor가 oneDone인 상태에서 “work”를 수신하면 그 메시지를 메일박스 어딘가에 저장해 두고 아무런 일도 하지 않으면 된다. 이것은 커피자동판매기에서 500을 넣은 상태에서 커피를 내보내라고 버튼을 눌렸을 때 아무런 일이 일어나지 않는 것과 동일한 이치다.

500원이 입력된 자동판매기는 커피를 내보내라는 버튼이 아니라 (합계 1,000원을 만들기 위한) 500원이라는 메시지를 기다린다. 따라서 다른 메시지에 대해서는 반

응을 보이지 않는다. “done” 메시지가 한 번 도착한 PingActor는 oneDone인 상태에서 추가적인 “done” 메시지를 기다린다. 따라서 “work”라는 메시지에 대해서는 반응을 보이지 않는다. 다만 그 메시지는 상태가 바뀌었을 때 처리해야 하므로 잊지 않고 메일박스에 잘 보관해둔다.

이번 장에서 예제 코드는 아카가 제공하는 become, unbecome API를 이용해서 이와 같은 동작을 수행하는 상태기계를 구현한다. 코드의 내용을 살펴보도록 하자. 우선 메인 클래스다.

```
package org.study;

import org.study.actor.PingActor;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 아카의 상태기계를 보여주기 위한 메인 클래스
 * @author Baekjun Lim
 */
public class Main {
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("TestSystem");
        ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class),
                "pingActor");
        ping.tell("work", ActorRef.noSender());
        ping.tell("reset", ActorRef.noSender());
    }
}
```

다음은 become/unbecome API를 이용해서 상태기계를 구현하는 Ping Actor 클래스다.

```
package org.study.actor;
```

```

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActorWithStash;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.Procedure;

/**
 * 아카의 상태기계를 구현한 액터
 * @author Baekjun Lim
 */
public class PingActor extends UntypedActorWithStash {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef child;

    public PingActor() {
        child = context().actorOf(Props.create(Ping1Actor.class), "ping1Actor");
        getContext().become(initial);
    }

    @Override
    public void onReceive(Object message) throws Exception {
    }

    /** 맨 처음 상태에서 "work" 메시지를 받으면 zeroDone 상태가 된다. */
    Procedure<Object> initial = new Procedure<Object>() {
        @Override
        public void apply(Object message) {
            if ("work".equals(message)) {
                child.tell("work", getSelf());
                getContext().become(zeroDone);
            } else {
                stash();
            }
        }
    };
}

/** zeroDone 상태에서 "done" 메시지를 받으면 oneDone 상태가 된다. */
Procedure<Object> zeroDone = new Procedure<Object>() {
    @Override

```

```

public void apply(Object message) {
    if ("done".equals(message)) {
        log.info("Received the first done");
        getContext().become(oneDone);
    } else {
        stash();
    }
}
};

/** oneDone 상태에서 "done" 메시지를 받으면 allDone 상태가 된다. */
Procedure<Object> oneDone = new Procedure<Object>() {
    @Override
    public void apply(Object message) {
        if ("done".equals(message)) {
            log.info("Received the second done");
            unstashAll();
            getContext().become(allDone);
        } else {
            stash();
        }
    }
};

/** allDone 상태에서 "reset" 메시지를 받으면 다시 initial 상태가 된다. */
Procedure<Object> allDone = new Procedure<Object>() {
    @Override
    public void apply(Object message) {
        if ("reset".equals(message)) {
            log.info("Received a reset");
            getContext().become(initial);
        }
    }
};
}

```

다음은 “work”라는 메시지를 받으면 Ping2Actor와 Ping3Actor에게 전송해 주는 역할을 수행하는 Ping1Actor 클래스다. 특별한 내용은 없다.

```
package org.study.actor;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

public class Ping1Actor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef child1;
    private ActorRef child2;

    public Ping1Actor() {
        child1 = context().actorOf(Props.create(Ping2Actor.class), "ping2Actor");
        child2 = context().actorOf(Props.create(Ping3Actor.class), "ping3Actor");
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            String msg = (String)message;
            log.info("Ping1 received {}", msg);
            child1.tell(msg, getSender());
            child2.tell(msg, getSender());
        }
    }
}
```

다음은 비즈니스 업무를 시뮬레이션 하는 Ping2Actor 클래스다. Ping3Actor의 내용은 거의 동일하므로 생략한다.

```
package org.study.actor;

import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
```

```

public class Ping2Actor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            String msg = (String)message;
            log.info("Ping2 received {}", msg);
            work();
            getSender().tell("done", getSelf());
        }
    }

    private void work() throws Exception {
        Thread.sleep(1000); // 실전에서는 절대 금물!!!
        log.info("Ping2 working...");
    }
}

```

액터는 메시지를 담는 메일박스와 별도로 상태에 따른 동작을 담아두기 위한 스택도 포함하고 있다. 이러한 동작스택에 저장되는 객체는 PingActor 코드에서 확인할 수 있는 Procedure 객체다. 스택에 Procedure 객체를 저장하는 방법은 context()를 이용해서 become() 메서드를 호출하면 된다. 액터의 어느 장소에서 context().become(initial)와 같이 호출하면 인수로 전달되는 initial이라는 이름의 Procedure 객체가 스택에 저장된다. 만약 unbecome이라는 메서드를 호출하면 스택의 꼭대기에 있던 객체가 꺼내어지고 현재 상태가 되기 직전의 상태로 되돌아간다.

become 메서드가 호출되지 않았거나 become 후에 unbecome이 호출되어서 동작스택이 비어 있으면 액터는 onReceive 메서드에 정의된 동작을 수행한다. 만약 스택에 Procedure 객체가 존재하면 그 객체의 apply 메서드가 정의하는 동작을 수행한다. 즉, Procedure의 동작이 onReceive 메서드의 동작을 대

신하는 것이다. 이러한 경우에 onReceive 메서드에 정의되어 있는 내용은 실행되지 않는다.

PingActor의 코드를 보면 initial, zeroDone, oneDone, allDone이라는 네 개의 Procedure 객체가 정의되고 있다. PingActor는 생성자에서 become(initial)을 호출하면서 초기상태가 된다. 이로써 PingActor에서 정의된 onReceive 메서드는 의미를 상실했다.

initial 상태는 오직 “work” 메시지만 기다린다. 나머지 메시지는 모두 stash()에 의해서 저장된다. stash() 메서드는 자기가 기다리고 있는 메시지가 아닌 다른 모든 메시지를 메일박스로 되돌려 보내는 동작을 수행한다. 기다리던 메시지가 전달되면 필요한 동작을 수행하고 그 다음 단계인 zeroDone 상태로 넘어간다. 이제 스택의 맨 윗자는 initial 상태가 아니라 zeroDone 상태가 차지한다.

zeroDone 상태는 더 간단하다. 오직 “done” 메시지를 기다리고, 만약 기다리던 메시지가 전달되면 oneDone 상태로 넘어간다. 여기에서도 마찬가지로 기다리는 메시지가 아닌 다른 메시지는 모두 stash()에 의해서 메일박스에 저장된다.

oneDone 상태도 마찬가지다. 오직 “done” 메시지를 기다린다. 기다리던 메시지가 전달되면 allDone 상태로 넘어간다. 기다린 메시지가 아닌 다른 메시지는 모두 stash()에 의해서 메일박스에 저장된다. 여기에서 한 가지 주목할 부분이 있다. oneDone 상태는 그 다음 단계인 allDone으로 넘어가기 전에 unstashAll()을 호출한다. 지금까지 메일박스에 저장해둔 메시지를 하나씩 다시 처리하라는 명령이다.

상태기계라는 개념이 익숙하지 않은 사람에게는 불필요하게 복잡한 방식으로 느껴질 수도 있다. 아, 너무 복잡해. 그냥 count 변수를 사용할래. 이렇게 생각하는 것이다. 하지만 상태기계라는 개념은 강력하다. 상태기계를 이용해서 문제를 해결하는 것이 익숙해지면 간결하고 단순하게 작성된 각각의 Procedure를 이용하는

것이 얼마나 편하고 강력한지 깨닫게 될 것이다.

이제 메인 클래스를 실행해보라. 다음과 같은 결과가 화면에 출력될 것이다.

```
[akka://TestSystem/user/pingActor/ping1Actor] Ping1 received work
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3 received work
[akka://TestSystem/user/pingActor/ping1Actor/ping2Actor] Ping2 received work
[akka://TestSystem/user/pingActor/ping1Actor/ping3Actor] Ping3 working...
[akka://TestSystem/user/pingActor/ping1Actor/ping2Actor] Ping2 working...
[akka://TestSystem/user/pingActor] Received the first done
[akka://TestSystem/user/pingActor] Received the second done
[akka://TestSystem/user/pingActor] Received a reset
```

메인 클래스를 수정해서 “work” 메시지를 연속해서 보내고 어떤 일이 일어나는지 확인해보라. PingActor가 “done” 메시지를 기다리고 있을 때 전달된 “work” 메시지가 어떻게 처리되는지 이해하는 것이 이번 장에서 다룬 내용의 핵심이다.

라우터

라우터

지금까지 아카 라이브러리가 제공하는 여러 가지 기능에 대해 알아보았다. 메시지를 주고받으면서 비동기적인 커뮤니케이션을 수행하는 방법, 감시전략을 이용해서 try-catch 구문을 사용하지 않고 예외를 처리하는 방법, 상태기계를 이용해 if-else 구문을 사용하지 않고 깔끔하고 명확하게 코드를 작성하는 방법 등을 공부했다.

하지만 액터시스템에 관심을 갖는 사람들은 비동기적 커뮤니케이션이나 예외 처리보다 병렬처리에 더 관심이 많다. 코딩하다 보면 ‘황당할 정도로 병렬적인’ 혹은 영어로 ‘embarrassingly parallel’이라고 표현하는 동작을 발견할 때가 있다. 순차적인 방식 대신 병렬적인 처리를 하면 훨씬 빠르게 작업을 마칠 수 있는 경우를 이런 식으로 표현하는데, 특히 병렬처리되는 대상이 서로 완벽하게 독립적인 방식으로 동작할 수 있으면 두말할 필요도 없이 ‘embarrassingly parallel’이다. 그런 코드를 병렬적으로 만들지 않고 쓰는 것은 부끄러운 일이라는 뜻이다.

자바에서는 JDK 7에 포함된 더그 리 Doug Lea의 fork and join 포크조인 라이브러리가 코드의 특정 부분을 병렬적 또는 동시적으로 처리하도록 만드는 데 많은 도움을 주었다. 자바 8에 람다가 포함되면서 특정한 동작을 병렬적으로 처리하는 것은 더욱 손쉬워졌다.

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

자바 8에서는 예를 들어서 이렇게 stream()이라는 메서드 대신 parallel Stream()이라고 써주면 라이브러리나 JVM이 내부적으로 알아서 병렬적인 처리를 수행해준다. 자바 8에서의 병렬처리 메커니즘이 parallelStream이라면 아카 라이브러리에서 사용하는 병렬처리 메커니즘은 라우터다.

먼저 예제 코드를 살펴보자. 우선 메인 클래스다.

```
package org.study;

import org.study.actor.PingActor;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 아카의 라우터를 보여주기 위한 메인 클래스
 * @author Baekjun Lim
 */
public class Main {
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("TestSystem");
        ActorRef ping = actorSystem.actorOf(Props.create(PingActor.class),
        "pingActor");
        ping.tell("start", ActorRef.noSender());
    }
}
```

다음은 PingActor 클래스다.

```
package org.study.actor;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.routing.RoundRobinPool;
import akka.routing.RoundRobinRouter;

/**
 * 아카이브 라우터를 이용해서 자식 액터를 만들고 1부터 10까지의 정수를 보내는 액터
 * @author Baekjun Lim
 */
public class PingActor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(),
this);
    private ActorRef childRouter;

    public PingActor() {
        childRouter = getContext().actorOf(
            new RoundRobinPool(5).props(Props.create(Ping1Actor.class)),
"ping1Actor");
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            for(int i = 0; i < 10; i++) {
                childRouter.tell(i, getSelf());
            }
            log.info("PingActor sent 10 messages to the router.");
        } else {
            unhandled(message);
        }
    }
}
```

마지막으로 Ping1Actor 클래스다.

```
package org.study.actor;

import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 정수를 받으면 자신의 해시코드 값과 함께 화면에 출력하는 간단한 액터
 * @author Baekjun Lim
 */
public class Ping1Actor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof Integer) {
            Integer msg = (Integer)message;
            log.info("Ping1Actor({}) received {}", hashCode(), msg);
            work(msg);
        }
    }

    private void work(Integer n) throws Exception {
        log.info("Ping1Actor({}) working on {}", hashCode(), n);
        Thread.sleep(1000); // 실전에서는 절대 금물!!!
        log.info("Ping1Actor({}) completed ", hashCode());
    }
}
```

PingActor가 자식 액터를 만들 때 평범하게 만들지 않고 아카의 라우터를 만들고 있다는 점을 제외하면 어려운 것은 없다. PingActor는 자식 액터를 다음과 같은 방식으로 생성한다.

```
public PingActor() {
    childRouter = getContext().actorOf(
        new RoundRobinPool(5).props(Props.create(Ping1Actor.class)), "ping1Actor");
}
```

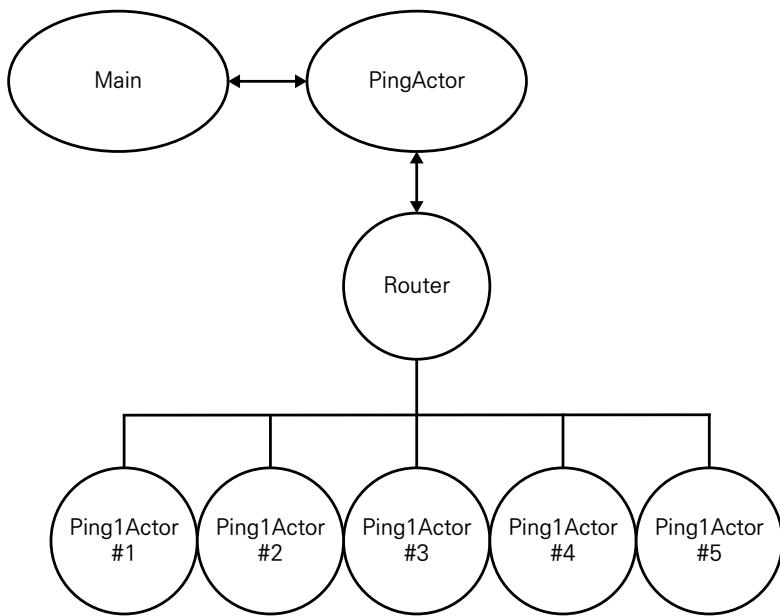
보통의 경우라면 Props.create(Ping1Actor.class)이 들어갈 자리에 new RoundRobinPool(5).props(Props.create(Ping1Actor.class))라는 코드가 들어가 있다. 우리가 만든 액터 조리법을 RoundRobinPool이라는 객체의 props 메서드에 인수로 전달하는 것이다. 이것은 아카에서 라우터를 만들기 위해서 사용하는 특정한 API다. RoundRobinPool 말고도 라우터의 종류에 따라서 사용할 수 있는 다른 타입이 여러 가지 존재한다.

라우터는 동일한 클래스를 이용해서 만들어진 여러 개의 액터 인스턴스 앞에 존재하는 가상의 액터라고 생각할 수 있다. new RoundRobinPool(5)라고 하면 Ping1Actor.class를 이용하는 액터를 5개까지 포함하는 풀(pool)을 만들어서 사용하겠다는 뜻이다. 스페드풀과 비슷한 개념이다. 우리는 이때 풀에 존재하는 구체적인 액터 인스턴스에 직접 접근하지 않는다. 대신 라우터를 가리키는 ActorRef를 사용한다. PingActor 코드에서 childRouter라는 이름을 가진 객체의 타입이 ActorRef인데 이것이 라우터다.

일반적인 액터나 라우터나 모두 ActorRef 타입으로 표현된다는 점에 있어서는 구별이 없다. 즉 액터를 사용하는 입장에서는 동일하다. 하지만 일반적인 액터는 ActorRef가 액터의 인스턴스 자체를 가리키고, 라우터의 경우에는 ActorRef가 액터의 인스턴스로 이루어진 풀 앞에 놓여있는 임시적인 액터를 가리킨다는 점에서 차이가 있다.

예제 코드가 구현하고 있는 액터와 라우터를 그림으로 표현해보면 다음과 같다. PingActor가 보기에 Ping1Actor는 평소와 다름없이 하나의 액터로 인식된다. 그림에서 Router 아래에 있는 Ping1Actor #1에서 #5까지의 인스턴스는 액터 풀에 존재하는 객체들로서 라우터 바깥에서는 보이지 않는다. 그들에게 접근하는 방법은 오직 라우터에게 메시지를 전달하는 것뿐이다. 이렇게 라우터 뒤에 숨어서 동작하는 액터들을 가르켜 “라우터^{routee}”라고 부르기도 한다.

그림 6-1



PingActor는 이렇게 만든 라우터에게 다음과 같은 for 루프를 통해서 10개의 메시지를 연속적으로 전송한다.

```
for(int i = 0; i < 10; i++) {
    childRouter.tell(i, getSelf());
}
```

라우터에게 전달된 10개의 메시지는 백그라운드 풀에 존재하는 5개의 Ping1Actor 인스턴스에게 하나씩 번갈아가면서 (round robin 방식으로) 전달된다. 이러한 Ping1Actor 액터들은 컴퓨터에 CPU 코어가 충분히 존재하면 최대한 병렬적인 방식으로 동작을 수행한다.

Ping1Actor는 메시지가 전달되면 자신의 해시코드 값과 함께 자기가 처리하고 있는 메시지의 값을 화면에 출력한다. 어떤 액터 인스턴스가 어떤 값을 처리하는

지 확인하기 위해서다. 이제 메인 클래스를 실행해보라. 다음과 같은 결과가 화면에 나타날 것이다.

```
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) received 1
[akka://TestSystem/user/pingActor/ping1Actor/$c] Ping1Actor(509523757) received 2
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) working on 1
[akka://TestSystem/user/pingActor/ping1Actor/$c] Ping1Actor(509523757) working on 2
[akka://TestSystem/user/pingActor/ping1Actor/$a] Ping1Actor(1556762159) received 0
[akka://TestSystem/user/pingActor] PingActor sent 10 messages to the router.
[akka://TestSystem/user/pingActor/ping1Actor/$a] Ping1Actor(1556762159) working
on 0
[akka://TestSystem/user/pingActor/ping1Actor/$e] Ping1Actor(198616476) received 4
[akka://TestSystem/user/pingActor/ping1Actor/$d] Ping1Actor(1881249507) received 3
[akka://TestSystem/user/pingActor/ping1Actor/$e] Ping1Actor(198616476) working on 4
[akka://TestSystem/user/pingActor/ping1Actor/$d] Ping1Actor(1881249507) working
on 3
[akka://TestSystem/user/pingActor/ping1Actor/$c] Ping1Actor(509523757)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$e] Ping1Actor(198616476)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$d] Ping1Actor(1881249507)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$a] Ping1Actor(1556762159)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$c] Ping1Actor(509523757) received 7
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) received 6
[akka://TestSystem/user/pingActor/ping1Actor/$c] Ping1Actor(509523757) working
on 7
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) working
on 6
[akka://TestSystem/user/pingActor/ping1Actor/$e] Ping1Actor(198616476) received 9
[akka://TestSystem/user/pingActor/ping1Actor/$d] Ping1Actor(1881249507)
received 8
[akka://TestSystem/user/pingActor/ping1Actor/$e] Ping1Actor(198616476) working
on 9
[akka://TestSystem/user/pingActor/ping1Actor/$d] Ping1Actor(1881249507) working
on 8
[akka://TestSystem/user/pingActor/ping1Actor/$a] Ping1Actor(1556762159)
```

```
received 5
[akka://TestSystem/user/pingActor/ping1Actor/$a] Ping1Actor(1556762159) working
on 5
[akka://TestSystem/user/pingActor/ping1Actor/$c] Ping1Actor(509523757)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$e] Ping1Actor(198616476)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$a] Ping1Actor(1556762159)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$d] Ping1Actor(1881249507)
completed
```

복잡하게 보이긴 하지만 단순한 내용이다. 예를 들어서 메시지 1이 어떻게 처리되고 있는지 살펴보자. 첫 번째 줄의 내용은 다음과 같다.

```
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) received 1
```

이 사실로부터 우리는 해시코드 값이 788026366인 객체가 (즉, Ping1Actor 액터의 인스턴스가) 값이 1인 메시지를 전달받았음을 알 수 있다. 여기서부터는 동일한 해시코드를 사용하는 객체가 출력한 내용을 추려서 정리하면 전체적으로 어떤 일이 일어나는지 쉽게 확인할 수 있다.

```
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) received 1
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) working
on 1
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366)
completed
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) received 6
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366) working
on 6
[akka://TestSystem/user/pingActor/ping1Actor/$b] Ping1Actor(788026366)
completed
```

이렇게 추려진 로그 내용으로부터 이 객체가 메시지 1과 6을 처리했음을 확인할 수 있다. 하나의 액터를 중심으로 살펴보면 이러한 처리가 우리가 기대하는 순차적인 방식대로 일어났음을 알 수 있다. 하지만 화면에 출력된 내용을 전체적으로 바라보면 메시지가 뒤죽박죽이다. 5개의 액터가 병렬적으로 작업을 수행하기 때문이다.

좀 더 자세한 내용을 공부해보고 싶은 사람은 PingActor의 childRouter를 라우터 대신 일반적인 액터로 바꾸고 전체적인 실행시간을 측정해보기 바란다. 그 다음 다시 라우터를 이용해서 실행시간을 측정해보면 (사용하는 컴퓨터가 여러 개의 코어를 가지고 있다고 가정했을 때) 실행시간이 눈에 띄게 단축되는 것을 확인할 수 있을 것이다. (그런 차이는 물론 Ping1Actor 안에 일부러 집어넣은 Thread.sleep(1000) 때문에 증폭될 것이다. 실제 아카코드에서 1초씩 잠을 자는 스레드는 어떤 경우에도 허용되지 않는다.)

아카의 라우터가 제공하는 기능은 대단히 강력하고 편리하기 때문에 실전코드에서 상당히 꼭넓게 사용된다. 병렬성이나 확장성을 고려할 때 라우터는 언제나 훌륭한 선택이 된다. 라우터는 뒤에서 살펴보게 될 클러스터링과도 밀접한 관련이 있다. 어려운 개념이 아니므로 라우터의 개념과 API를 확실하게 이해하고 넘어가는 것이 매우 중요하다.

예제에서 우리는 Ping1Actor 객체가 여러 개 생성된다는 점, 그리고 그들이 동시에 동작을 수행한다는 사실을 강조하기 위해서 일부러 화면에 객체의 해시값을 출력했다. 하지만 사실 해시값은 불필요하다. 출력된 내용을 자세히 들여다보면 우리가 살펴보았던 액터가 다음과 같은 고유한 경로^{path}를 가지고 있음을 알 수 있다.

```
/user/pingActor/ping1Actor/$b
```

아카에서 모든 액터는 트리구조에 포함된 노드이기 때문에 루트에서 시작해서 자신에게 이르는 ‘경로’라는 개념을 갖는다. 파일시스템의 디렉토리와 흡사한 개념

이다. 우리가 다루고 있는 액터의 경우에는 “pingActor”라는 이름의 액터에서 출발해서 “ping1Actor”라는 이름의 라우터 액터를 지나 “\$b”라는 이름의 액터에 이르는 경로를 가지고 있다. 다시 말해서 이 액터의 이름은 “\$b”다. 이 액터는 아카의 라우터 기능에 의해서 풀 내부에서 동적으로 생성되기 때문에 우리(사람)가 붙여주는 이름이 없다.

이런 지식을 갖고 출력된 내용을 다시 살펴보면 라우터에 의해서 동적으로 생성된 아카의 이름이 “\$a”에서 “\$e”에 이르는 5개임을 알 수 있다.

라우터와 감시전략

실전코드에서 라우터를 사용하다 보면 초보자들이 제대로 이해하지 못하는 부분이 종종 눈에 띈다. 그것은 바로 풀 내부에 존재하는 액터 인스턴스들의 부모 액터가 누구인가 하는 점이다. 예제 코드에서 “ping1Actor”라는 이름의 라우터를 만든 부모 액터는 “pingActor”였다. 하지만 “ping1Actor”로 지칭되는 액터는 실제 Ping1Actor 클래스의 인스턴스가 아니라 ‘라우터’를 가리키는 액터다.

그렇다면 풀 내부에 존재하는 다섯 개의 Ping1Actor 인스턴스들의 부모는 “pingActor”인가 아니면 “ping1Actor”인가.

앞에서 본 경로에서도 쉽게 확인할 수 있듯 Ping1Actor 인스턴스들의 부모는 당연히 “ping1Actor”, 즉 라우터 자신이다. 그것은 곧 Ping1Actor의 인스턴스들(\$a, \$b, \$c, \$d, \$d라는 이름을 갖는 액터 인스턴스들)의 감시자도 “ping1Actor”라는 이름을 가진 라우터라는 사실을 뜻하기도 한다.

일반적으로 라우터는 모든 예외를 자신의 부모에게 보고 escalate하는 기본적인 전략을 가지고 있다. 하지만 다음 코드에서 보는 것처럼 라우터 자신에게 감시전략을 부여할 수도 있다.

```
final SupervisorStrategy strategy =  
    new OneForOneStrategy(5, Duration.create(1, TimeUnit.MINUTES),  
        Collections.<Class<? extends Throwable>>singletonList(Exception.class));  
final ActorRef router = system.actorOf(new RoundRobinPool(5),  
    withSupervisorStrategy(strategy).props(Props.create(Echo.class)));
```

라우터를 이러한 방식으로 생성하면 풀 내부에 존재하는 액터 인스턴스가 예외를 발생시켰을 때 애플리케이션의 요구사항에 맞는 필요한 동작을 수행하도록 만들 수 있다.

풀과 그룹

이것은 다소 고급에 속하는 내용이라서 여기에서는 간단하게 설명을 하겠다. 아카 라우터에서 풀^{pool}은 라우터에게 내부의 액터 인스턴스를 생성할 권리를 부여하는 것을 의미한다. 그래서 라우터가 액터 인스턴스들의 감시자 역할도 담당하고, 부모로서의 라이프사이클을 관리한다. 우리가 지금까지 이야기 한 라우터는 이러한 풀 개념을 사용했다.

하지만 경우에 따라서는 우리가 직접 생성한 액터를 그룹으로 묶은 다음, 어떤 라우터에게 그 그룹에 존재하는 액터들을 라우터^{routee}로 사용하라고 말하고 싶을 때도 있다. 이런 경우에는 라우터가 액터 인스턴스를 생성할 권리를 갖지 않고, 따라서 감시자나 부모의 역할도 수행하지 않는다. 단순히 자기에게 전달된 메시지를 정해진 알고리즘에 따라서 그룹에 존재하는 다른 액터에게 전달하는 역할만 담당 할 뿐이다.

라우팅 알고리즘

라우터는 자기에게 전달된 메시지를 특정한 라우팅 알고리즘에 따라서 자신의 풀(혹은 그룹)에 존재하는 액터 인스턴스에게 전달한다. PingActor의 경우에는 우리가 RoundRobin 알고리즘을 선택했기 때문에 전달되는 메시지가 공평하게 순서대로 전달된다.

아카 라이브러리는 단순한 라운드로빈 이외에도 다음과 같이 다양한 라우팅 알고리즘을 지원한다.

RandomRouter

이름 그대로 라우터 배후에서 동작하고 있는 액터 인스턴스 중에서 하나를 무작위로 골라서 메시지를 전송하는 알고리즘이다.

SmallestMailboxRouter

액터 인스턴스 중에서 메일박스에 처리되지 않고 저장되어 있는 메시지의 수가 가장 적은 것을 골라서 보내는 알고리즘이다. 인스턴스 사이에서 업무량의 균형을 맞춰주고 싶을 때 사용하면 된다.

BroadcastRouter

라우터에 전달된 메시지를 액터 인스턴스 모두에게 전송하고자 할 때 사용한다.

ScatterGatherFirstCompletedRouter

scatter라는 영어단어는 흩뿌린다는 뜻을 가지고 있다. gather first completed라는 표현은 첫 번째로 완료된 결과를 수집한다는 뜻이다. 두 표현을 연결해서 생각해보면 라우터에 전달된 메시지를 모든 액터 인스턴스에 전달한 다음, 가장 먼저 계산된 결과를 리턴 하는 것을 취하고 나머지는 무시하는 알고리즘이다.

액터 인스턴스가 모두 동일한 JVM 혹은 컴퓨터 위에서 실행되고 있는 경우에는

사용할만한 방법이 아니지만, 인스턴스가 모두 각각 별도의 컴퓨터 위에서 실행되고 있다면 가장 빠른 응답을 보장받기 위해서 사용할 수 있는 방법이다.

ConsistentHashingRouter

해시에서 키가 매핑되는 슬롯의 위치에 액터가 있다고 생각하면 이해하기 쉽다. 라우터는 전달된 메시지가 가진 키의 해시 값을 계산한 다음, 그 해시 값에 매핑되어 있는 액터 인스턴스에게 메시지를 전송한다. 따라서 해시 값이 변하지 않는 한 동일한 키를 가진 메시지는 언제나 동일한 액터 인스턴스에게 전달된다. 아카이브에서 지원하는 ConsistentHashingRouter는 단순히 해시 알고리즘만 사용하는 것이 아니라 분산 시스템에서 널리 사용되는 알고리즘인 일관적 해시^{consistent hashing}를 사용해서 가변성을 최소화 시킨 것이다.

이 알고리즘이 동작하는 방식을 숙지해 두는 것은 분산시스템을 개발할 때 많은 도움이 되지만, 이 책의 범위를 넘어서므로 더 이상 설명하지 않겠다.

퓨처와 에이전트

퓨처

액터시스템은 비동기적 메시지 전송, 즉 ‘보내고 잊기’ 철학을 바탕으로 구축되어 있다. 이러한 보내고 잊기 동작을 구현하는 방법으로는 액터처럼 메시지를 주고받는 것이 가장 직관적이다. 하지만 실전코드를 작성하다 보면 보내고 잊기만으로 모든 동작을 구현하기는 어렵다. 경우에 따라서는 ‘보내고 잊지 않기’가 필요한 때가 있다.

여기에서 이야기하는 ‘보내고 잊지 않기’는 소포를 보내고 트래킹 번호를 받는 것과 비슷하다. 우체국에서 받은 트래킹 번호를 웹사이트 같은 곳에서 등록해서 수신인이 소포를 받았을 때 나에게 문자메시지가 오도록 만드는 것이다. 이렇게 하면 나는 여전히 소포를 보내자마자 다른 일을 시작할 수 있지만, 내가 원하는 일이 완료되었을 때 어떤 알람을 받을 수 있다. 퓨처는 그런 상황에서 사용할 수 있는 구조물이다. 퓨처는 일종의 트래킹 번호다.

퓨처는 휴이트가 이야기한 액터모델의 일부는 아니지만 기능의 편리함 때문에 아카 라이브러리에 포함되었다. Future라는 클래스 자체는 스칼라 언어의 `scala.concurrent` 패키지에 존재하는 기본적인 클래스이지만 아카는 Future를 이용한 통신방법을 다양한 API를 통해서 지원한다. 오래전부터 자바에도 퓨처라는 개념이 존재해왔다. 하지만 아카/스칼라에 포함된 퓨처에 비하면 자바

의 동시성 패키지에서 사용하는 퓨처는 기능이 제한적이다. 자바 8에 이르러서 CompletableFuture라는 것이 포함되었는데 이것은 아카/스칼라의 퓨처에 상당히 근접한 기능을 포함하고 있다.

아카에서 퓨처를 사용하는 방법은 ActorRef가 제공하는 tell 메서드를 사용하는 것이 아니라 Patterns라는 유ти리티 객체가 제공하는 ask 메서드를 사용하는 것이다. 예를 들면 이런 식이다.

```
Timeout timeout = new Timeout(Duration.create(5, "seconds"));
Future<Object> future = Patterns.ask(actor, msg, timeout);
String result = (String) Await.result(future, timeout.duration());
```

이때 우리가 메시지를 보내고자 하는 목적지에 해당하는 액터는 ask 메서드에게 인수로 전달된다. 보내려는 메시지와 타임아웃 객체도 인수로 전달된다. 이렇게 Patterns.ask라는 메서드를 호출하면 퓨처 객체가 리턴된다. ask 메서드 자체는 비동기적이므로 즉각적으로 리턴된다. 문제는 그 다음 줄에 있는 Await.result다. Await라는 단어 자체가 기다림을 뜻하는 것처럼, 이 메서드 호출은 현재 스레드가 그 자리에서 답변을 기다리게 만든다. 블로킹 호출이다. 퓨처를 이용하면 이렇게 동기적인 방식의 코드, 혹은 블로킹 코드를 만드는 것이 가능하다.

실전코드를 작성할 때 ‘모든 것이 비동기적인 세계’라는 새로운 패러다임을 온전히 받아들이지 못한 개발자가 이런 퓨처의 블로킹 기능을 남용하는 경우를 본 적이 있다. ‘퓨처’라는 말을 들으면 Await.result처럼 곧바로 블로킹 호출을 떠올리는 개발자들이 의외로 많다. 아마 자바 퓨처의 영향이 클 것이다. 다음과 같은 전형적인 자바 퓨처 사용례를 생각해보자.

```
Future<String> f = superHeavyOperation();
String result = f.get();
```

이 코드에서 Future는 아카/스칼라 퓨처가 아니라 자바 퓨처다. superHeavyOperation()이라는 메서드는 호출이 일어나면 바로 퓨처를 리턴하고 백그라운드 스레드를 이용해서 무거운 작업을 수행한다. 따라서 superHeavyOperation()이라는 호출 자체는 비동기적이고 난블로킹이다. 하지만 우리는 무거운 작업이 수행된 다음에 도출되는 결과가 필요하다. 그래서 두 번째 줄에서 f.get()이라는 식으로 퓨처로부터 결과 값을 읽으려고 시도한다. 이게 문제다. f.get()은 동기적인 블로킹 호출이다. 현재 스레드는 바로 그 지점에서 퓨처가 어떤 값을 리턴할 때까지 명청하게 기다릴 수밖에 없다. 이것이 지금까지 우리가 (자바 8 이전의) 자바에서 퓨처를 사용하는 방식이었다.

우리가 액터시스템을 이용하는 이유의 하나는 바로 이와 같은 블로킹 지점을 아예 없애버리기 위해서다. 그런 면에서 아카가 f.get()과 개념적으로 동일한 Await.result라는 API를 포함시킨 것은 다소 유감이다. 자바 퓨처에 익숙한 개발자들이 Await.result를 남용하면서 액터시스템의 기본전제를 희생시키는 길을 열어주었기 때문이다. 거칠게 말하자면 Await.result는 Thread.sleep하고 다를 게 없다. 현재 스레드가 하는 일 없이 무언가를 기다리게 만든다는 점에 있어서 그렇다.

퓨처를 쓰되 Await.result에 기대지 않아도 되는 방법은 물론 존재한다. 연속^{continuation}이라는 개념을 사용하는 것이다. 아카에서 퓨처를 사용하는 것은 Await.result라는 블로킹 호출을 사용하는 것이 아니라 '연속'이라는 난블로킹 방식을 사용하는 것을 의미한다.

예제코드

예제 코드를 통해서 우리는 퓨처를 블로킹 방식으로 사용하는 모습과 난블로킹 방식으로 사용하는 것을 모두 살펴볼 것이다. 우선 메인 클래스다. 편의를 위해서 BlockingMain과 NonblockingMain으로 나누어 놓았다.

우선 블로킹 메인 클래스다.

```
package org.study;

import org.study.actor.BlockingActor;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 아카의 Future를 이용해서 blocking 동작을 보여주는 메인 클래스
 * @author Baekjun Lim
 */
public class BlockingMain {

    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("TestSystem");
        ActorRef blockingActor =
            actorSystem.actorOf(Props.create(BlockingActor.class), "blockingActor");
        blockingActor.tell(10, ActorRef.noSender());
        blockingActor.tell("hello", ActorRef.noSender());
    }
}
```

다음은 난블로킹 메인 클래스다.

```
package org.study;

import org.study.actor.NonblockingActor;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 아카의 Future를 이용해서 non-blocking 동작을 보여주는 메인 클래스
 * @author Baekjun Lim
 */
public class NonblockingMain {
```

```
public static void main(String[] args) {
    ActorSystem actorSystem = ActorSystem.create("TestSystem");
    ActorRef nonblockingActor =
        actorSystem.actorOf(Props.create(NonblockingActor.class),
    "nonblockingActor");
    nonblockingActor.tell(10, ActorRef.noSender());
    nonblockingActor.tell("hello", ActorRef.noSender());
}
}
```

다음은 퓨처를 블로킹 방식으로 사용하는 액터인 BlockingActor의 내용이다.

```
package org.study.actor;

import scala.concurrent.Await;
import scala.concurrent.ExecutionContext;
import scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.dispatch.OnComplete;
import akka.dispatch.OnFailure;
import akka.dispatch.OnSuccess;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.pattern.Patterns;
import akka.util.Timeout;

/**
 * 아카의 퓨처를 이용해서 블로킹 동작을 보여주는 액터
 * @author Baekjun Lim
 */
public class BlockingActor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef child;
    private Timeout timeout = new Timeout(Duration.create(5, "seconds"));
    private final ExecutionContext ec;
```

```
public BlockingActor() {
    child = context().actorOf(Props.create(CalculationActor.class),
"calculationActor");
    ec = context().system().dispatcher();
}

@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof Integer) {
        Future<Object> future = Patterns.ask(child, message, timeout);
        Integer result = (Integer) Await.result(future, timeout.
duration()); // 블로킹!!!!
        log.info("Final result is " + (result + 1));
    } else if (message instanceof String) {
        log.info("BlockingActor received a messasge: " + message);
    }
}
}
```

다음은 퓨처를 난블로킹 방식으로 사용하는 액터인 NonblockingActor의 내용이다.

```
package org.study.actor;

import scala.concurrent.ExecutionContext;
import scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.dispatch.OnComplete;
import akka.dispatch.OnFailure;
import akka.dispatch.OnSuccess;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.pattern.Patterns;
import akka.util.Timeout;

/**
```

```

* 아카의 퓨처를 이용해서 non-blocking 동작을 보여주는 액터
* @author Baekjun Lim
*/
public class NonblockingActor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private ActorRef child;
    private Timeout timeout = new Timeout(Duration.create(5, "seconds"));
    private final ExecutionContext ec;

    public NonblockingActor() {
        child = context().actorOf(Props.create(CalculationActor.class),
        "calculationActor");
        ec = context().system().dispatcher();
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof Integer) {
            Future<Object> future = Patterns.ask(child, message, timeout);

            // onSuccess, onComplete, onFailure 등은 blocking 동작이 아니다.
            future.onSuccess(new SaySuccess<Object>(), ec);
            future.onComplete(new SayComplete<Object>(), ec);
            future.onFailure(new SayFailure<Object>(), ec);
        } else if (message instanceof String) {
            log.info("NonblockingActor received a messasge: " + message);
        }
    }

    public final class SaySuccess<T> extends OnSuccess<T> {
        @Override public final void onSuccess(T result) {
            log.info("Succeeded with " + result);
        }
    }

    public final class SayFailure<T> extends OnFailure {
        @Override public final void onFailure(Throwable t) {
            log.info("Failed with " + t);
        }
    }
}

```

```
public final class SayComplete<T> extends OnComplete<T> {
    @Override public final void onComplete(Throwable t, T result) {
        log.info("Completed.");
    }
}
```

끝으로 백그라운드에서 수행되는 계산을 시뮬레이션 하는데 사용되는 CalculationActor의 내용이다.

```
package org.study.ch7;

import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

public class CalculationActor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof Integer) {
            Integer msg = (Integer)message;
            log.info("CalculationActor received {}", msg);
            work(msg);
            getSender().tell(msg * 2, getSelf());
        }
    }

    private void work(Integer n) throws Exception {
        log.info("CalculationActor working on " + n);
        Thread.sleep(1000); // 실전에서는 절대 금물!!!
        log.info("CalculationActor completed " + n);
    }
}
```

블로킹 호출

코드가 수행하는 일은 간단하다. 우선 BlockingMain을 실행하면 메인 메서드는 BlockingActor에게 10이라는 메시지를 전송한다. BlockingActor가 메시지를 받으면 다음과 같이 Patterns.ask와 Await.result를 이용해서 CalculationActor에게 메시지를 보내고 계산된 결과 값을 돌려받는다. 여기에 서 Await.result는 물론 블로킹 호출이다.

```
@Override  
public void onReceive(Object message) throws Exception {  
    if (message instanceof Integer) {  
        Future<Object> future = Patterns.ask(child, message, timeout);  
        Integer result = (Integer) Await.result(future, timeout.duration()); //  
        블로킹!!!!  
        log.info("Final result is " + (result + 1));  
    } else if (message instanceof String) {  
        log.info("BlockingActor received a messasge: " + message);  
    }  
}
```

CalculationActor는 정수형 메시지 값이 전달되면 거기에 2를 곱한 다음 되돌려주는 간단한 연산을 수행한다. 중간에 일부러 Thread.sleep(1000)을 호출하여 시간이 오래 걸리는 연산을 시뮬레이션 한다.

BlockingMain을 실행해보라. 그럼 다음과 같은 결과가 화면에 출력될 것이다. 핵심에 집중하기 위해서 불필요한 부분은 수작업으로 제거했다.

```
[12:04:03.718] CalculationActor received 10  
[12:04:03.719] CalculationActor working on 10  
[12:04:04.719] CalculationActor completed 10  
[12:04:04.719] Final result is 21  
[12:04:04.719] BlockingActor received a messasge: hello
```

이 결과에서 맨 마지막 줄이 출력된 시간에 주목하기 바란다. 12:04:04.719다. CalculationActor가 작업을 시작한 시점으로부터 정확히 1초 뒤다. 다시 BlockingMain의 메인 메서드를 보자.

```
blockingActor.tell(10, ActorRef.noSender());  
blockingActor.tell("hello", ActorRef.noSender());
```

메인 메서드는 10이라는 메시지를 전송한 다음에 곧바로 “hello”라는 메시지를 전송했다. 그렇기 때문에 아마 “hello” 메시지는 BlockingActor의 메일박스에 10이라는 메시지와 거의 동시에 (물론 1 나노초라도 뒤에) 도착했을 것이다. 아무리 양보해도 그들이 도착한 시간의 차이가 1초에 이르리라고는 생각하기 어렵다. 하지만 “hello” 메시지가 BlockingActor에 의해서 받아들여진 시점은 무려 1초 후다.

1초라는 억겁의 시간 동안 BlockingActor가 CalculationActor의 퓨처가 값을 리턴할 때까지 기다렸기 때문이다. 바로 이와 같은 비효율성이 블로킹 호출의 효과다. 이상적인 코드였다면 “hello” 메시지는 메일박스에 도착하자마자 처리가 되었어야 한다.

난블로킹 호출

이번에는 NonblockingMain을 실행해보라. NonblockingMain과 NonblockingActor가 수행하는 일은 앞의 경우와 거의 동일하다. 여기에서는 블로킹 호출이 일어나지 않는다는 것만 다르다. 코드를 실행하면 다음과 같은 결과가 화면에 나타날 것이다. 여기에서도 불필요한 내용을 수작업으로 제거했다.

```
[12:30:36.023] CalculationActor received 10  
[12:30:36.023] CalculationActor working on 10
```

```
[12:30:36.025] NonblockingActor received a messasge: hello  
[12:30:37.023] CalculationActor completed 10  
[12:30:37.026] Completed.  
[12:30:37.027] Succeeded with 20
```

“hello” 메시지가 어디에 있는지 확인해보라. 세 번째 줄에 있다. Calculation Actor가 작업을 시작한지 겨우 2밀리초 이후에 “hello” 메시지가 처리되었다. 이것이 난블로킹 호출의 위력이다.

NonblockingActor는 다음과 같은 코드를 통해서 퓨처를 난블로킹 방식으로 소비한다. Patterns.ask는 퓨처를 얻기 위해서 사용했지만, Await.result는 사용하지 않는다. 대신 onSuccess, onComplete, onFailure 메서드에게 백그라운드 작업이 성공했을 때, 완료되었을 때, 그리고 실패했을 때 실행되어야 하는 코드를 (일종의 콜백callback처럼) 전달한다.

```
Future<Object> future = Patterns.ask(child, message, timeout);  
future.onSuccess(new SaySuccess<Object>(), ec);  
future.onComplete(new SayComplete<Object>(), ec);  
future.onFailure(new SayFailure<Object>(), ec);
```

이 4줄의 코드 중 어디에도 블로킹 동작은 없다. NonblockingActor는 이렇게 퓨처에게 필요한 동작을 설정해놓고, 곧바로 메일박스에 있는 다음 메시지를 처리한다. 그렇기 때문에 “hello” 메시지가 전에 비해서 훨씬 빠르게 처리될 수 있었던 것이다.

아카에서 사용하는 스칼라 퓨처가 강력한 이유는 이것이 모나드^{Monad}이기 때문이다. 모나드에 대한 이야기는 이 책의 범위를 뛰어넘으므로 생략하겠다. 하지만 스칼라 퓨처는 서로 조합^{compose}하는 것이 가능하고, 결과 값을 다른 액터에게 메시지로 전달하는^{pipe} 것도 가능하다. 퓨처를 이용해서 map, flatmap, filter와 같은 여러 가지 함수형 연산을 수행하는 것도 가능하다. 이런 기능의 풍요로움 때문

에 퓨처를 애용하는 사람도 있는데, 이런 모든 API의 배후에서 동작하는 것이 액터라는 사실만 잊지 않으면 그것도 좋은 방법이다.

퓨처에 대한 자세한 내용은 akka.io의 문서에 있기 때문에 더 이상 다루지 않겠다. 다만 퓨처를 이용해서 조합, 파이프, map, filter 등의 연산을 수행하고자 한다면 자바보다 스칼라를 이용하는 것이 더 낫다는 말을 해두고 싶다. 예를 들어서 똑같은 기능을 수행하는 다음 두 개의 코드를 비교해보라.

우선 자바 코드다.

```
final ExecutionContext ec = system.dispatcher();

Future<String> f1 = future(new Callable<String>() {
    public String call() {
        return "Hello" + "World";
    }
}, ec);

Future<Integer> f2 = f1.map(new Mapper<String, Integer>() {
    public Integer apply(String s) {
        return s.length();
    }
}, ec);

f2.onSuccess(new PrintResult<Integer>(), system.dispatcher());
```

다음은 스칼라 코드다.

```
val f1 = Future {
    "Hello" + "World"
}

val f2 = f1 map { x =>
    x.length
}

f2 foreach println
```

자바 8 문법을 사용하면 자바 코드가 더 간결해질 수 있는 여지가 있긴 하지만, 이런 차이를 불러일으키는 것이 단순히 람다 표현 때문인 것만은 아니다. 아카 자체가 스칼라를 이용해서 만들어진 라이브러리이기 때문에 아카의 내부에 들어가다 보면 언젠가 스칼라를 만날 수밖에 없다. 자바와 스칼라의 차이는 표면적인 문법의 차이가 아니다. 그것은 함수 ‘스타일’을 일부 도입한 언어와 함수 언어의 차이다. 여기에서는 더 자세히 설명하지 않지만 아카를 공부하는 개발자는 스칼라를 완전히 외면하기 어렵다는 점을 말해두고 싶다.

에이전트

JVM에서 사용되는 함수 언어의 양대 산맥은 스칼라와 클로저다. 에이전트는 그 클로저 언어에서 차용된 개념이다. 앞에서 살펴본 퓨처와 마찬가지로 원래부터 액터 모델의 일부는 아니지만 기능의 편리함 때문에 아카 라이브러리에 포함되었다. 그렇기 때문에 에이전트를 이해하기 위해서는 구체적인 구조물로서의 에이전트가 아니라 하나의 추상적인 개념으로서의 에이전트를 먼저 이해하는 것이 중요하다.

앞에서 이야기한 것처럼 액터는 70년대에 정의된 수학적 추상이다. (1) 메시지를 받아들이고, (2) 메시지를 전송하고, (3) 다른 액터를 만드는 것. 이렇게 세 가지 기능을 가지고 있는 추상적 개념이다. 우리가 아카에서 사용하는 액터는 요나스 보네어가 스칼라를 이용해서 이렇게 추상적인 액터 모델을 구현해놓은 것이다. 에이전트도 개념에서 출발한다. 에이전트는 (1) 어떤 하나의 값을 포함하고, (2) 값을 변경하기 위한 함수를 받아들이고, (3) 값을 읽는 동작을 지원한다. 액터가 내부에 여러 가지 값(즉, 상태)과 동작을 동시에 포함할 수 있는데 비해서 에이전트는 값만 포함한다.

비유를 하자면 액터는 객체고 에이전트는 하나의 변수다. 액터는 내부에 on

Receive나 Procedure를 통해서 동작을 정의한다. 내부에 자기만의 데이터 구조를 가지고 상태를 유지할 수도 있다. 그런 면에서 전통적인 프로그래밍에서 등장하는 객체를 닮았다. 이에 비해서 에이전트는 내부에 코드가 없다. 값만 존재한다. 하지만 에이전트를 이용하는 외부의 코드는 이러한 값에 직접 접근할 수 없고, 에이전트가 제공하는 API를 통해서만 접근할 수 있다.

코드를 보자. 에이전트를 사용하는 예를 보여주고 있는 AgentActor의 내용이다.

```
package org.study.actor;

import scala.concurrent.ExecutionContext;
import akka.actor.UntypedActor;
import akka.agent.Agent;
import akka.dispatch.ExecutionContexts;
import akka.dispatch.Mapper;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 아카이의 에이전트가 동작하는 방식을 보여주는 액터
 * @author Baekjun Lim
 */
public class AgentActor extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    @Override
    public void onReceive(Object message) throws Exception {
        ExecutionContext ec = ExecutionContexts.global();
        Agent<Integer> agent = Agent.create(5, ec);

        agent.send(new Mapper<Integer, Integer>() {
            public Integer apply(Integer i) {
                return i * 2;
            }
        });

        // 에이전트의 값이 여전히 5로 출력될 것이다 (아닐 수도)
    }
}
```

```
        log.info("Current agent value = " + agent.get());
        Thread.sleep(100); // 일부러 조금 기다린다 - 실전에서는 절대 금물!!!
        // 에이전트의 값이 10으로 출력될 것이다 (아닐 수도)
        log.info("Current agent value = " + agent.get());
    }
}
```

AgentMain을 실행해보라. 코드를 실행시키면 다음과 같은 결과가 화면에 나타날 것이다. 불필요한 부분은 손으로 제거했다

```
[16:45:33.351] Current agent value = 5
[16:45:33.451] Current agent value = 10
```

AgentActor는 다음과 같은 코드를 이용해서 (1) 5라는 초기 값을 담은 에이전트 하나를 만들고 (2)값에 2를 곱하라는 변경을 해당 에이전트에 요청하고 있다.

```
Agent<Integer> agent = Agent.create(5, ec); // 5를 담은 에이전트를 생성

agent.send(new Mapper<Integer, Integer>() {
    public Integer apply(Integer i) {
        return i * 2; // 내부의 값에 2를 곱하는 함수
    }
});
```

agent.send라는 메서드는 에이전트에게 내부의 값을 변경하라고 요청할 때 사용하는 API다. 보다시피 에이전트에 포함되어 있는 값의 타입에 상응하는 apply라는 메서드를 정의해서 전달한다. 이렇게 전달된 코드는 나중에 에이전트에 의해 실행되고, 그 결과로 에이전트 내부의 상태가 변경된다.

에이전트 바깥에 있는 클라이언트 코드는 전달된 코드가 정확히 언제 실행되는지에 대해서 아무런 결정권이 없다. 그것은 전적으로 에이전트 내부의 결정에 달려 있는 문제다. 그런 면에서 agent.send는 actor.tell과 마찬가지로 비동기적인

동작이다. AgentActor의 코드는 그런 비동기성을 드러내기 위해서 작성되었다.

agent.send가 호출되자마자 에이전트의 현재 값을 화면에 출력하면 그 값은 전과 마찬가지로 여전히 5다. 하지만 100밀리초 동안 일부러 기다린 다음, 다시 에이전트의 현재 값을 화면에 출력하면 값이 10으로 바뀌었다. 에이전트의 내부 상태가 정확히 언제 바뀌었는지는 알 수 없지만 아무튼 값에 2를 곱하라는 함수를 전달한 agent.send 호출은 정상적으로 실행이 되었다.

액터시스템에서 에이전트를 사용하는 이유는 그것이 제공하는 비동기적인 API가 액터와 거의 동일한 일관성을 제공하기 때문이다. 또한 퓨처와 마찬가지로 에이전트도 모나드이기 때문에 함수 패러다임의 다양한 연산이 가능하다. 전통적인 자바 프로그래밍에서, 특히 멀티스레딩 환경 안에서 어떤 값을 읽거나 변경하려면 반드시 어떤 보호 장치를 사용해야 한다. 예컨대 synchronized 키워드나 ReentrantLock 같은 객체를 이용해서 일정한 코드블록을 감싸주거나 최소한 AtomicInteger와 같은 유ти리티 클래스를 이용해서 값 자체를 감싸주어야 한다. 에이전트는 AtomicInteger 같은 유ти리티와 비슷한 역할을 수행하지만, 훨씬 더 풍부한 기능을 제공해주는 구조물이라고 생각하면 된다.

다음 장에서 우리는 마지막으로 아카의 클러스터에 대해서 살펴볼 것이다. 클러스터 환경에서 한 가지 주의할 점은 에이전트는 로컬 JVM 안에서만 정상적으로 사용될 수 있다는 사실이다. 원격 컴퓨터에서 동작하는 액터들이 서로 다양한 객체를 메시지로 주고받을 수 있는데, 그러한 메시지 안에 에이전트가 포함되면 안 된다는 뜻이다.

클러스터

클러스터

클러스터 기능은 아카의 꽃이다. 많은 개발자들이 확장성 있는 시스템을 개발하기 위해서 아카를 선택하는 이유는 무엇보다도 아카가 제공하는 클러스터 기능의 편리함과 강력함 때문이다. 아카 클러스터를 이해하기 위해서는 약간의 이론적인 내용을 공부할 필요가 있는데 그 내용은 뒤에서 보기로 하고 우선 예제부터 살펴보도록 하자.

지금까지와 달리 이번 장에서는 예제가 두 개의 프로젝트로 나뉘어 있다. 깃헙을 보면 ch8-node1과 ch8-node2라는 두 개의 프로젝트가 있을 것이다. 두 개의 독립된 노드로 이루어진 ‘클러스터’라는 개념을 이해할 수 있도록 하기 위해서 일부러 각각의 노드를 별도의 프로젝트로 만들었다. 이번 장을 위한 예제는 약간 복잡하므로 우선 코드를 실행하는 것부터 해보자.

- ch8-node1 프로젝트의 Main을 실행하라.
- ch8-node2 프로젝트의 Main을 실행하라.
- 동일한 컴퓨터에서 웹브라우저를 실행한 다음, 주소창에 `http://localhost:8877/akkaStudy`를 입력하고 엔터키를 누른다.
- 화면에 “PING 0”라는 결과가 나올 것이다. F5키를 눌러서 페이지를 다시 로드하면 그 때마다 PING 뒤에 나오는 정수 값이 1씩 증가할 것이다.

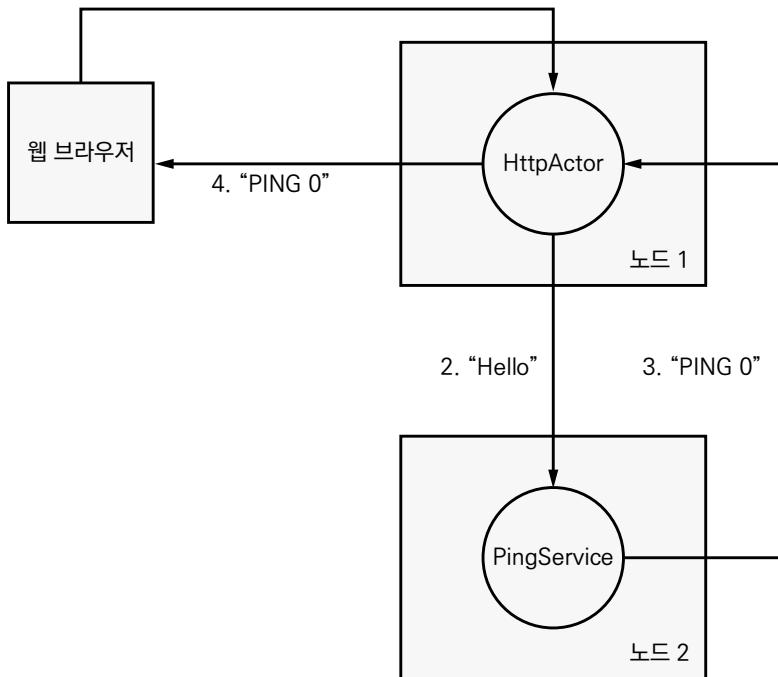
이번 예제가 동작하는 방식을 이해하려면 우선 전체적인 그림을 이해할 필요가 있

다. 단순하게 말하자면 노드1은 웹서버고 노드2는 백엔드다. 노드1을 실행하면 HTTP 요청을 받아들이는 액터가 실행되고, 노드2를 실행하면 어떤 메시지가 전달되었을 때 “PING N”이라는 메시지를 응답으로 보내는 액터가 실행된다. 여기에서 N은 그 액터의 내부에서 관리되고 있는 정수형 변수로 요청이 이루어질 때마다 1만큼 증가한다.

여기에서 노드1에서 동작하는 액터와 노드2에서 동작하는 액터는 아카 클러스터 기능을 통해서 서로 메시지를 주고받는다. 그게 예제의 핵심이다. 나머지는 다 군더더기다. 코드를 보기 전에 그림을 보고 예제가 동작하는 과정 전체를 이해해 두기 바란다.

그림 8-1

1. `http://localhost:8877/akkaStudy` 요청



코드를 보기 전에 다음 설명을 읽고 각 클래스가 수행하는 역할이 무엇인지 이해해 두는 것도 필요하다.

Node1

Main

- ClusterSystem이라는 이름을 사용하는 액터시스템을 생성한다.
- PingService라는 액터클래스로 이루어진 라우터^{router}를 만든다. (라우터의 뒤에서 동작하는 액터는 PingService 인스턴스다.)
- 간단한 웹서버 역할을 수행할 HttpActor 액터를 만든다. 앞에서 만든 라우터를 가리키는 ActorRef 객체를 HttpActor의 생성자에 인수로 전달한다. HttpActor가 라우터에게 메시지를 전달할 필요가 있기 때문이다.

HttpActor

- 생성될 때 ActorRef 객체를 (즉, PingService 액터로 이루어진 라우터를) 전달받는다.
- 아카-캐멀 통합 기능을 이용해서 http://localhost:8877/akkaStudy라는 주소를 사용하는 서비스를 시작한다.
- http://localhost:8877/akkaStudy에 HTTP 요청이 들어오면 CamelMessage라는 타입의 객체를 전달받는다. (아카-캐멀 통합 기능이 제공해주는 기능이므로 자세한 내용은 몰라도 상관없다.)
- CamelMessage가 전달되면 (즉, HTTP 요청이 들어오면) 메시지를 보내온 sender를 변수에 기억해 둔 다음, 라우터에게 “hello”라는 문자열을 전송한다.
- 어디에선가 문자열^{String} 메시지가 도착하면 변수에 저장해 두었던 sender에게 답변을 전송한다.

Node2

Main

- ClusterSystem이라는 이름을 사용하는 액터시스템을 생성한다.
- PingService 액터를 만든다.

PingService

- 문자열메시지가 전달되면 sender에게 “PING”이라는 답변을 보낸다. 내부에 저장된 정수형 변수를 이용해서 그것이 몇 번째 답변인지도 나타낸다. 예를 들어서 액터가 시작된 다음에 처음으로 전달된 메시지면 답변이 “PING 0”이고 두 번째 메시지면 답변은 “PING 1”이다.

이제 코드를 보자.

코드

먼저 Node1에서 실행되는 Main 클래스와 HttpActor 클래스의 모습이다.

```
package org.study.ch8;

import org.study.ch8.http.HttpActor;
import org.study.ch8.service.PingService;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;
import akka.routing.FromConfig;

/**
 * 아카이 클러스터링을 보여주기 위한 메인 클래스
 * @author Baekjun Lim
 */
public class Main {

    /** 웹서버 역할을 하는 HttpActor와 라우터를 생성한다 */
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("ClusterSystem");
        ActorRef router = actorSystem.actorOf(
            Props.create(PingService.class).withRouter(new FromConfig()),
            "serviceRouter");
        ActorRef httpActor = actorSystem.actorOf(
            Props.create(HttpActor.class, router), "httpActor");
    }
}
```

다음은 아카-캐멀 통합 라이브러리를 이용해서 웹서버 역할을 수행하는 HttpActor의 내용이다. HTTP나 캐멀과 관련된 내용은 예제의 핵심적인 내용이 아니므로 자세하게 이해하려고 애쓸 필요는 없다. 생성자에 인수로 전달되는 ActorRef 객체를 onReceive 메서드 안에서 서비스로 사용하고 있는 점에 주목하기 바란다.

```
package org.study.ch8.http;

import akka.actor.ActorRef;
import akka.camel.CamelMessage;
import akka.camel.javaapi.UntypedConsumerActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 웹서버 역할을 수행하기 위해서 캐멀과 제티를 사용하는 액터
 * @author Baekjun Lim
 */
public class HttpActor extends UntypedConsumerActor{

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private String uri;
    private ActorRef service;
    private ActorRef sender;

    public HttpActor(ActorRef service) {
        this.service = service;
        this.uri = "jetty:http://localhost:8877/akkaStudy";
    }

    public void onReceive(Object message) throws Exception {
        if (message instanceof CamelMessage) {
            this.sender = getSender();
            service.tell("Hello", getSelf());
        } else if (message instanceof String) {
            sender.tell(message, getSelf());
        } else {
            unhandled(message);
        }
    }
}
```

```
        }
    }

    public String getEndpointUri() {
        return uri;
    }
}
```

다음은 웹서버 뒤에서 백엔드 서비스의 역할을 수행하는 Node2의 Main 클래스와 PingService의 코드를 살펴보도록 하자. 우선 메인 클래스다.

```
package org.study.ch8;

import org.study.ch8.service.PingService;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

/**
 * 아카이 클러스터링을 보여주기 위한 메인 클래스
 * @author Baekjun Lim
 */
public class Main {

    /** 백엔드 서비스 역할을 수행하는 PingService 액터를 생성한다. */
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("ClusterSystem");
        ActorRef pingService = actorSystem.actorOf(
            Props.create(PingService.class), "pingService");
    }
}
```

다음은 PingService 액터의 내용이다.

```
package org.study.ch8.service;

import akka.actor.UntypedActor;
```

```

import akka.event.Logging;
import akka.event.LoggingAdapter;

/**
 * 백엔드 서비스의 역할을 흉내내는 액터.
 * 메시지가 전달되면 단순히 "PING N" 메시지를 리턴한다.
 * N은 PING이 리턴될 때마다 1씩 증가하는 정수.
 * @author Baekjun Lim
 */
public class PingService extends UntypedActor {

    private LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private int count = 0;

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String) {
            String msg = (String)message;
            getSender().tell("PING: " + count++, getSelf());
        } else {
            unhandled(message);
        }
    }
}

```

이상으로 이번 장의 예제를 구성하는 클래스의 내용을 모두 살펴보았다. 아카 클러스터를 처음 접하는 사람이라면 노드1의 액터와 노드2의 액터가 서로 어떻게 연결되는지 아직 구체적으로 이해가 되지 않을 것이다. 그런 연결의 접착제 역할을 해주는 것이 바로 아카의 구성파일이다. 그 중에서도 deployment라는 이름의 속성인데, 구성파일의 내용을 확인해 보도록 하자.

구성파일

아카 클러스터를 제대로 구현하려면 코드 이상으로 구성파일이 중요하다. Node1과 Node2의 구성파일을 살펴보도록 하자. 먼저 ch8-node1 프로젝트

의 src/main/resources 디렉토리에 존재하는 application.conf 파일의 전문이다.

```
akka {  
    actor {  
        provider = "akka.cluster.ClusterActorRefProvider"  
  
        deployment {  
            /serviceRouter {  
                router = round-robin-pool  
                routees.paths = ["/user/pingService"]  
                cluster {  
                    enabled = on  
                    allow-local-routees = off  
                    #use-role = compute  
                }  
            }  
        }  
    }  
    remote {  
        log-remote-lifecycle-events = off  
        netty.tcp {  
            hostname = "127.0.0.1"  
            port = 2551  
        }  
    }  
  
    cluster {  
        seed-nodes = [  
            "akka.tcp://ClusterSystem@127.0.0.1:2551"]  
  
        auto-down-unreachable-after = 10s  
    }  
}
```

다음은 ch8-node2 프로젝트의 src/main/resources 디렉토리에 있는 application.conf의 전문이다.

```
akka {
    actor {
        provider = "akka.cluster.ClusterActorRefProvider"
    }
    remote {
        log-remote-lifecycle-events = off
        netty.tcp {
            hostname = "127.0.0.1"
            port = 0
        }
    }
    cluster {
        seed-nodes = [
            "akka.tcp://ClusterSystem@127.0.0.1:2551"]
        auto-down-unreachable-after = 10s
    }
}
```

두 구성파일 사이에 존재하는 차이를 확인하기 바란다. 예를 들어서 Node1의 구성파일에는 deployment라는 섹션이 존재하는데 Node2의 구성파일에는 없다. 이유는 곧 설명할 것이다.

actor.provider

먼저 액터 제공자provider의 내용을 보자.

```
actor {
    provider = "akka.cluster.ClusterActorRefProvider"
}
```

이것은 액터시스템을 시작했을 때 액터시스템이 ActorRef 객체를 만들 때 아카에 존재하는 3개의 제공자 중에서 어느 것을 사용할지를 정해주는 내용이다. 아카

의 액터시스템이 사용할 수 있는 제공자로는 다음과 같은 3가지가 존재한다.

- LocalActorRefProvider
- RemoteActorRefProvider
- ClusterActorRefProvider

클래스 이름이 스스로의 의미를 잘 드러내고 있으므로 별도의 설명은 하지 않겠다. 기본 값은 LocalActorRefProvider이므로 클러스터를 구축하고 싶은 경우에는 제공자를 ClusterActorRefProvider로 바꿔줘야 한다. 구성파일에서 이 제공자를 사용하면 액터시스템을 만들 때 자동으로 클러스터가 시작된다.

actor.remote

다음은 액터 리모트 속성이다.

```
remote {
    log-remote-lifecycle-events = off
    netty.tcp {
        hostname = "127.0.0.1"
        port = 2551
    }
}
```

어떤 액터가 자신이 속한 JVM이 아닌 다른 JVM에서 동작하는 액터와 메시지를 주고받으려면 반드시 리모트 속성을 선언해 주어야 한다. 다른 JVM이나 컴퓨터에서 실행되고 있는 액터가 해당 액터에게 메시지를 보내려면, 그 액터가 동작하고 있는 컴퓨터의 IP 주소와 포트 값을 알아야 하기 때문이다.

혹은 어떤 액터시스템이 클러스터의 일부로 (즉 하나의 노드로) 참여하려면 반드시 자신이 동작하고 있는 컴퓨터의 IP 주소와 포트를 공개해야 한다고 생각해도 좋다. netty.tcp가 정의하고 있는 hostname은 액터시스템이 동작하고 있는 컴퓨

터의 주소를 의미하고, port는 포트 값을 의미한다.

이 예제에서는 두 개의 JVM을 (즉, 노드1과 노드2를) 하나의 컴퓨터 안에서 실행할 것이기 때문에 로컬호스트localhost를 의미하는 “127.0.0.1”를 주소로 사용했다. 만약 노드1과 노드2를 별도의 컴퓨터에서 실행하고 싶은 사람은 hostname의 값을 실제 IP 주소로 바꾸면 된다.

여기에서 port의 값은 2551로 설정이 되어 있는데, 이것을 0으로 설정하면 아카에게 임의의 포트 값을 동적으로 생성해서 할당하라는 의미다. 씨앗노드seed node의 개념을 설명할 때 이러한 동적 포트 값에 대한 이야기가 다시 등장할 것이다.

actor.cluster

그 다음은 actor.cluster다.

```
cluster {  
    seed-nodes = [  
        "akka.tcp://ClusterSystem@127.0.0.1:2551"]  
  
    auto-down-unreachable-after = 10s  
}
```

actor.provider의 값이 ClusterActorRefProvider로 설정된 경우에는 반드시 cluster 속성을 통해서 씨앗노드의 주소와 포트를 설정해 주어야 한다.

씨앗노드는 어떤 액터시스템이 클러스터에 참여하기 위해서 참여joining 메시지를 보내는 컴퓨터를 의미한다. seed-nodes라는 값에 설정되는 주소는 하나의 값이 아니라 여러 개의 값을 갖는 리스트다. 이 예제에서는 127.0.0.1:2551이라는 하나의 주소만 사용하고 있지만 예를 들어서 10개의 노드로 구성된 클러스터라면 적어도 3개의 컴퓨터 정도를 씨앗노드로 설정할 것이다.

동일한 클러스터에 참여하는 액티시스템들은 이 씨앗노드의 주소를 공유해야 한다. 그래야 같은 클러스터에 참여해서 서로 메시지를 주고받을 수 있기 때문이다. ch8-node1 프로젝트와 ch8-node2 프로젝트도 동일한 씨앗노드 주소를 공유하고 있음을 확인하기 바란다.

auto-down-unreachable-after라는 속성은 아카 클러스터에서 노드가 갖는 라이프사이클과 관련이 있다. 아카 클러스터에 참여한 노드들은 서로 수시로 일종의 확인(health-check) 메시지를 주고받는다. 만약 어느 노드가 확인 메시지에 대해서 제 때 응답하지 않으면 그 노드의 상태가 응답 없음(unreachable) 상태로 진입한다. auto-down-unreachable-after=10s이라는 값은 이러한 응답 없음 상태가 10초간 유지되면 그 노드의 상태를 자동적으로 다운(down) 상태로 설정하라는 뜻이다. 일단 다운 상태가 된 노드는 반드시 클러스터에서 제거가 되어야 하고, 다시 처음부터 새롭게 참여 메시지를 보내서 클러스터에 들어와야 한다.

이러한 이야기가 아직은 별로 구체적으로 이해되지 않을 것이다. 하지만 아카 클러스터를 활용한 코드를 다루다보면 이러한 속성이 왜 의미를 갖는지 스스로 터득 할 수 있게 된다.

deployment

마지막으로 deployment를 알아보자.

```
deployment {
    /serviceRouter {
        router = round-robin-pool
        routees.paths = ["/user/pingService"]
        cluster {
            enabled = on
            allow-local-routees = off
            #use-role = compute
```

```
        }  
    }  
}
```

이 속성이 아카 클러스터 구성에서 가장 핵심이다. 이 부분을 잘 이해하지 않으면 아카 클러스터를 실전에 적용하는 것은 불가능하다.

우선 이 속성이 ch8-node1에는 정의되어 있는데, ch8-node2에는 정의되어 있지 않음에 주목하기 바란다. 노드1의 HttpActor가 사용하는 라우터, 더 구체적으로 그 라우터의 배후에서 동작하는 PingService 액터 인스턴스들을 원격 컴퓨터에 ‘전개(deployment)’하기 위한 속성이기 때문이다. 그러한 전개가 필요한 것은 노드1이지 노드2가 아니다. 그렇게 전개된 액터를 사용하는 주체는 노드1의 HttpActor인 것이다.

우선 /serviceRouter라는 속성의 이름은 우리가 노드1의 메인 클래스에서 라우터를 만들 때 사용한 이름과 정확히 일치해야 한다. 코드를 보면 우리가 라우터의 이름을 serviceRouter라고 정한 것을 알 수 있다. 만약 코드에서 이 이름을 “test”라고 바꾸면 구성파일에서도 속성의 이름을 /test로 바꿔야 한다.

```
ActorRef router = actorSystem.actorOf(  
    Props.create(PingService.class).withRouter(new FromConfig()),  
    "serviceRouter");
```

router=round-robin-pool은 앞에서 라우터를 공부할 때 보았던 라우팅 알고리즘을 의미한다. 라우드로빈 방식이 선택되고 있다.

routees.paths=["/user/pingService"]은 원격 컴퓨터에서 동작하는 액터시스템에서 이 라우터의 라우터, 즉 PingService 액터를 생성할 때 그 액터들이 가져야 하는 경로를 정의한다.

아카의 클러스터 기능을 사용하기 위해서 숙지할 필요가 있는 속성들은 이밖에도 많다. 자세한 내용은 akka.io에 있는 문서를 통해서 확인하기 바란다.

이제 한 가지 재미있는 실험을 해보자. 노드2의 메인 클래스가 가진 main 메서드의 내용은 다음과 같다.

```
/** 백엔드 서비스 역할을 수행하는 PingService 액터를 생성한다. */
public static void main(String[] args) {
    ActorSystem actorSystem = ActorSystem.create("ClusterSystem");  ActorRef
pingService = actorSystem.actorOf( Props.create(PingService.class),
"pingService");
}
```

이 코드에서 마지막 줄, 즉 pingService를 생성하고 있는 줄을 지운다면 어떤 일이 벌어질까? 그래도 예제 코드가 정상적인 기능을 수행할 수 있을까? 직접 실험을 해보기 바란다. 언뜻 생각하기로는 백엔드 역할을 수행하는 액터가 존재하지 않으므로 더 이상 웹브라우저의 화면에 “PING 0”과 같은 메시지가 출력될 것 같지 않다.

하지만 이 마지막 줄은 사실 불필요하다. 이유를 잘 생각해보기 바란다. 지금까지 이 책의 내용을 공부한 사람이라면 스스로 답을 찾을 수 있을 것이다.