



리액티브 프로그래밍

(with Scala, Akka, Play)



nextree.namoosori.2017

목차 (Table of Contents)

1. 리액티브 웹 애플리케이션
2. Play로 리액티브 프로그래밍 시작하기
3. 스칼라
4. 함수형 프로그래밍
5. Future
6. Actor



리액티브 웹 애플리케이션

최근의 컴퓨팅 추세

- ✓ 멀티코어 프로세스
- ✓ 분산 처리
- ✓ 수많은 디바이스: 유비쿼터스 환경으로 인한 기하급수적 증가
- ✓ 전통적인 접근방법의 어려움

Reactive Web Application

- ✓ 웹 Application은 지난 몇년간 비약적인 발전을 이루어 왔고 우리 생활에서 중요한 일부분이 되었다.
- ✓ 소셜 네트워크, 인터넷 뱅킹,..
- ✓ 리액티브 웹 Application은 이러한 고가용성, 고 효율을 만족하는 대안으로 등장하였다.
- ✓ Play Framework는 리액티브 웹 Application이 요구하는 고가용성, 실시간성을 구현한 풀 스택 프레임워크이다.

리액티브 프로그램은...

1. 동작하는 환경에 대해서 끊임이 없이 가용하여야 한다.
2. 환경에 속도를 맞추어 동작한다. 프로그램 혼자 앞서나가지 않는다.
3. 외부 요청에 즉각 응답한다.

Reactive Manifesto

- ✓ 리액티브 선언문, 2013년 6월 발행 , Reactive Application의 아키텍처에 대한 기술
- ✓ Reactive Application이 가지는 특성을 정의함.
- ✓ <http://www.reactivemanifesto.org/>

Reactive Manifesto에서 정의한 리액티브 애플리케이션의 특징

1. 응답성 - 사용자에게 반응한다.
2. 확장성 - 사용량에 반응한다.
3. 탄력성 - 실패에 반응한다.
4. 이벤트 드리븐 - 이벤트에 반응한다.

Reactive Programming

- ✓ 리액티브 프로그래밍은 데이터 흐름을 기반으로 한 프로그래밍의 한 방식이다.
- ✓ 엑셀 프로그램 또는 구글 워드프로세서 등에서 이러한 방식의 프로그래밍을 엿볼 수 있다.
- ✓ 엑셀과 같이 계산식이 들어있는 셀에서 다른 셀의 값이 변함에 따라 자동으로 계산되어 지는 프로그램을 구현하는 방법.
- ✓ 비동기, 이벤트 기반의 기술이며 오늘날 많은 종류의 기술들이 발생하고 있다. (Node.js, Netty, Play)

Microsoft Reactive Extension - <https://rx.codeplex.com/>

Node.js - <http://nodejs.org>

Netty - <http://netty.io>

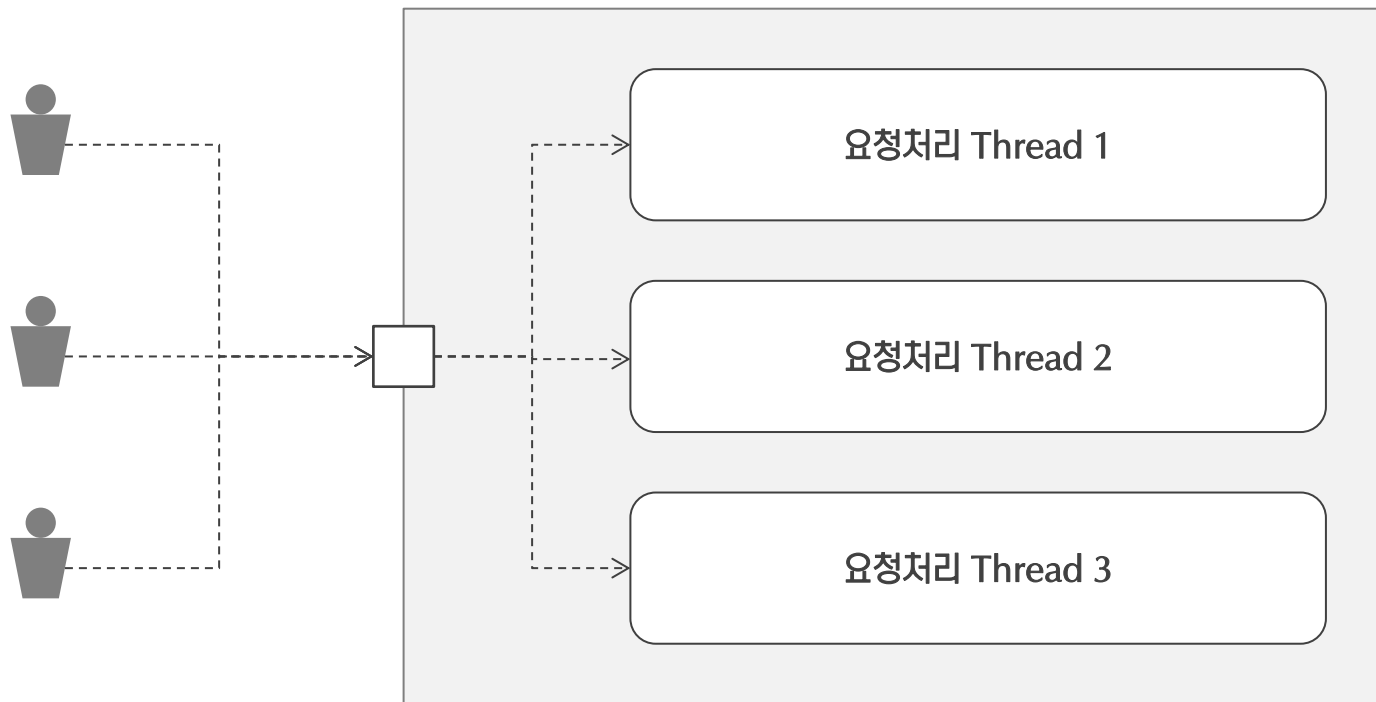
Play Framework

- ✓ 오늘날 리액티브를 지원하는 많은 기술들이 등장함.(Rx 계열, Node.js, Netty, 등)
- ✓ 그러나 이러한 대부분의 기술들은 리액티브 애플리케이션을 구현하기 위한 도구에 불과함.
- ✓ Play Framework은 리액티브 애플리케이션을 구현하기 위한 풀 스택 프레임워크임.
- ✓ 오직 Play Framework 만이 리액티브 기술과 풀 스택을 지원함. 그리고 그 내부도 Reactive 원칙을 따라 설계됨.



스레드 기반 처리

- ✓ 요청처리를 동시에 하기 위하여 일반적으로 사용하는 방식임.
- ✓ 메모리 자원이 한계가 있으므로 일정 수 이상의 동시 처리는 어려움.
- ✓ 스레드를 발생시키기 위한 자원 소모가 많으며 컨텍스트 스위칭 비용이 비쌈.
- ✓ 공유자원에 대한 처리가 어려움.



이벤트 방식 처리

- ✓ 요청 작업을 최대한 작은 단위의 이벤트로 분리한 후 처리하는 방식
- ✓ 제한된 수의 스레드가 발생되며 이벤트의 처리는 단일 스레드에서 처리됨.
- ✓ 비동기로 처리되므로 요청 이벤트 전송 후 기다리지 않는다.
- ✓ 자원을 효율적으로 사용할 수 있음. 스레드 방식에 비하여 메모리 사용량이 적고 빠른 처리가 가능하다.



스레드 방식과 이벤트 방식의 구현

- ✓ 전통적인 요청 처리의 구현 방식은 스레드 방식임. 변수, 순차적 명령의 나열 등 구현방식이 평이함.
- ✓ 개발자 코드는 스레드 내에서 동작하며 동기 방식의 순차적 나열로 표현 됨.
- ✓ 이벤트 방식은 비동기 처리를 기반으로 하므로 콜백 패턴을 사용함.
- ✓ 이벤트 방식은 함수형 언어가 구현하기에 유리함. 따라서 자바스크립트 또는 스칼라 같은 언어를 선호함.

* 스레드간의 공유자원 처리의 어려움

- 공유자원은 언제든지 변경가능함.
- 한 자원에 대하여 여러 스레드가 경쟁상태에 놓임.
- 데드락이 발생할 수 있음.
- 정확하게 테스트한 코드도 스레드 경쟁상황에 놓이면 예측하지 못한 결과가 발생하며 이를 해결하기 위하여 방어코드가 발생하고 결국 복잡하고 어려운 프로그램으로 변질됨.

* 함수형 언어의 특징

- 함수중심이며 함수를 전달하는 방식을 사용함.
- 값은 immutable 함. 따라서 사이트 이펙트가 발생하지 않음.
- 선언형 언어

이벤트 방식 서버 Vs 스레드 방식 서버

- ✓ 스레스 수의 차이. 스레드 방식 서버는 동시 요청된 수만큼 스레드가 발생하나 이벤트 방식은 그렇지 않다.
- ✓ 스레드 방식은 하나의 처리 프로세스가 하나의 스레드를 독점하는 반면
- ✓ 이벤트 방식은 하나의 처리 프로세스가 작은 function 단위로 나뉘어져 이벤트 call 방식으로 처리됨. 따라서 스레드 자원을 여러 처리 프로세스가 공유함.
- ✓ 이벤트 방식은 수많은 비동기 처리가 발생되며 각 처리 function은 서로 값을 공유하지 않음
- ✓ 이벤트 방식의 서버를 구현하기 위해서는 함수형 프로그래밍이 적합함.

* 스레드 방식 서버

- 메모리 사용량이 높을 수 있음
- 하나의 비즈니스 처리 로직이 스레드를 독점함
- 개발자는 비즈니스 처리 로직을 순차적으로 어려움 없이 작성함.

* 이벤트 방식 서버

- 제한된 스레드 발생
- 하나의 비즈니스 처리 프로세스는 수많은 비동기 function으로 분리됨.
- 비동기이므로 순차처리 구현이 까다로움.

스레드 방식의 문제점

- ✓ 여러 스레드가 하나의 자원 공유할 경우 문제가 발생함.
- ✓ 동시 접근시 스레드는 경쟁 상태가 되며 이때 Thread-safe 여부가 중요한 이슈가 됨.
- ✓ 개발자 입장에서는 구현이 쉽고 명확하더라도 현실 세계에서는 스레드 경쟁으로 인해 예기치 못한 결과가 나올 수 있음.
- ✓ 이러한 스레드 방식 문제의 근본 원인은 mutable(변하는) state 의 공유임.

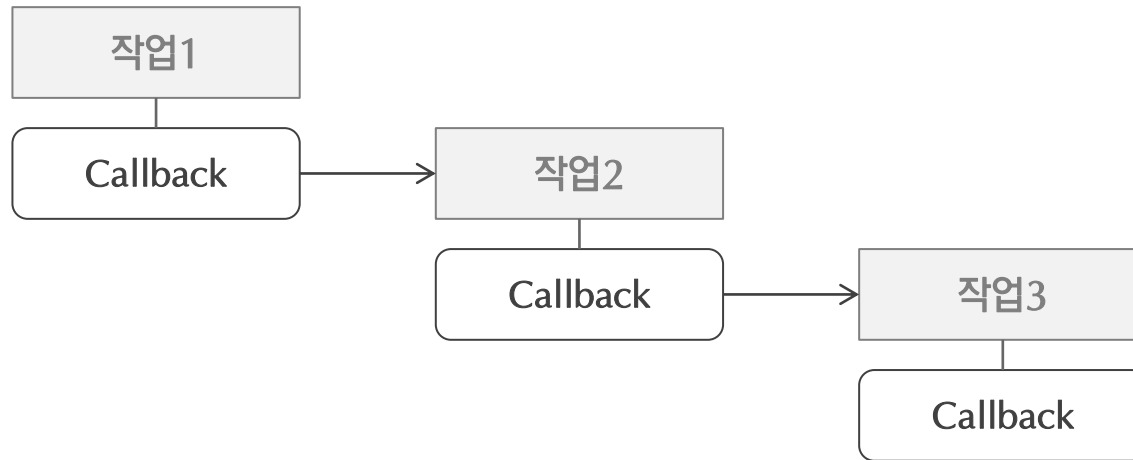
스레드 방식은 mutable 한 변수 값을 예측할 수 없으므로 각종 방어코드(locking, 상태 체크)가 필요하며 프로그램은 복잡해 지고 어려워짐.

이 문제를 해결하려면 프로세스가 수행되는 동안 관련된 값을 고정(immutable) 시켜야 하고 프로세스는 격리 시켜야 함.

함수형 언어, 프로그래밍이 필요한 이유임.

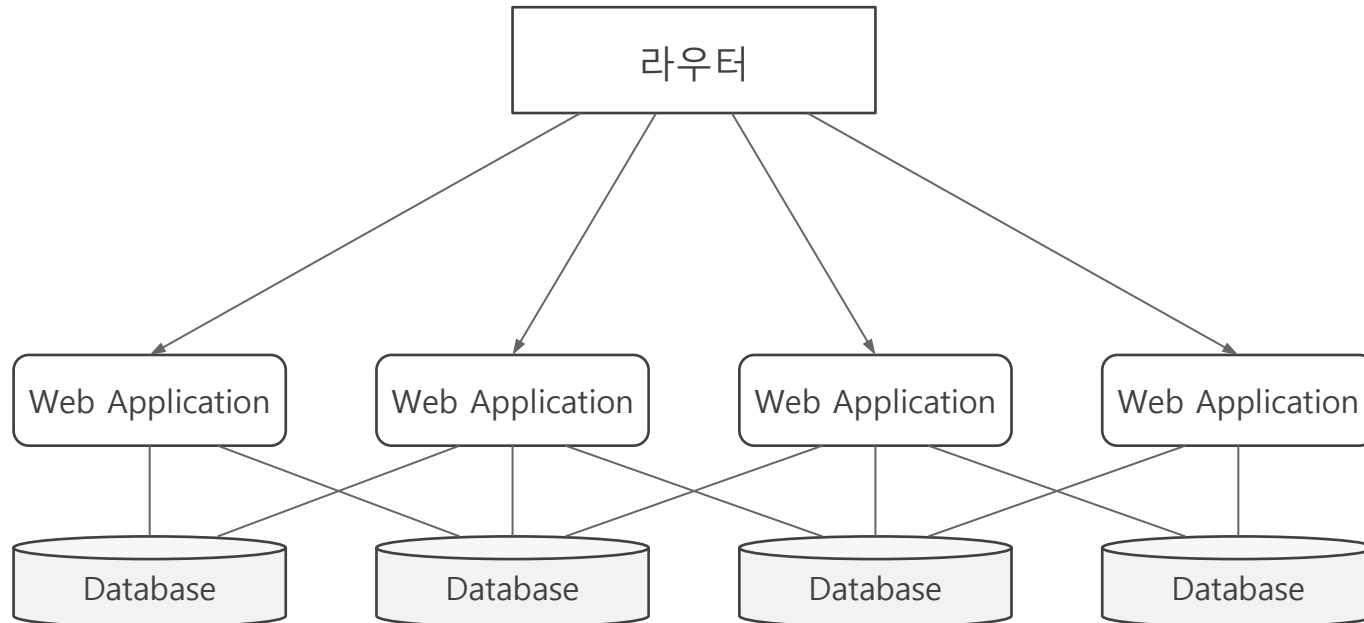
이벤트 기반 서버 구현

- ✓ 모든 작업을 비동기로 처리해야 하므로 콜백 패턴을 사용하여 구현함.
- ✓ 비동기와 순차처리를 동반한 코드는 수많은 콜백을 사용하게 되어 코드의 가독성이 떨어짐(콜백 헬, <http://callbackhell.com>)
- ✓ 비동기 처리와 동시에 사람이 이해 가능한 코드를 만들기 위해서는 함수형 프로그래밍 과 더불어 Scala와 같은 추상화 된 언어를 사용하여야 함.



Stateless 아키텍처

- ✓ Scale-out 확장 시스템은 현대 서버 시스템 분산 처리를 위한 방법이다.
- ✓ 데이터베이스는 분산처리를 효과적으로 수행하는 NoSql DB를 사용함(MongoDB)
- ✓ Web Application 분산 처리시 가장 문제가 되는 부분은 Session과 같은 상태값의 처리이다.
- ✓ Play Framework은 무상태(Stateless) 아키텍처를 사용하므로 Scale-out이 쉽다.

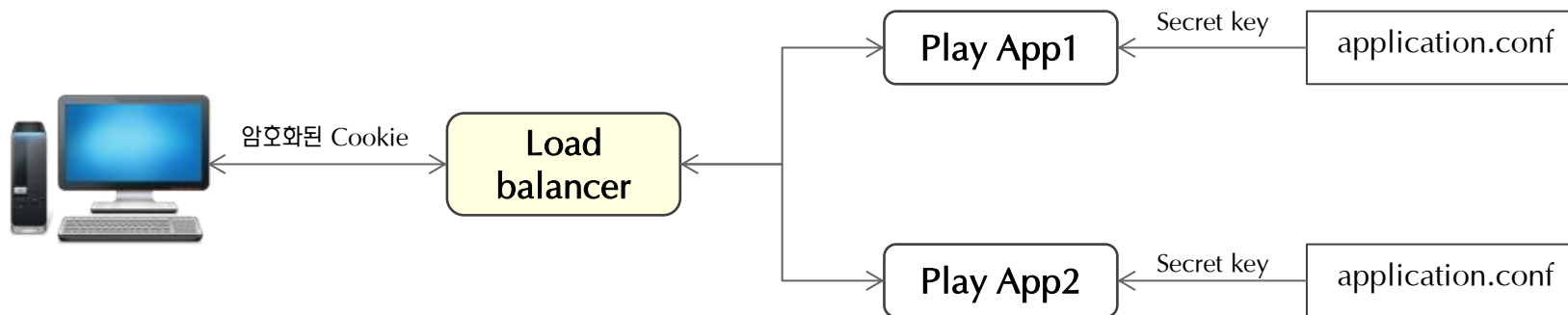


Play Framework의 클라이언트 세션

- ✓ Session 또는 Flash Scope의 데이터를 서버가 아닌 클라이언트에 저장하는 것은 보안 문제를 야기함.
- ✓ Play는 cookie를 암호화하고 서버에서 secret key로 풀수 있게 처리함.
- ✓ secret key는 application.conf에 존재함.
- ✓ Play Application을 여러 인스턴스로 분산처리하는 경우 이러한 secret key를 반드시 동일하게 유지해야 함.

```
...  
application.secret="1/jnx^4=7:gFa`3CdruOa2rGVNo?9l0a3S>RM:CCTySSstj;]NL9dDZVs=fXm:t/"  
...
```

application.conf



리액티브 시스템의 탄력성

- ✓ 단일 혹은 네트워크 구간에서 분산 처리되는 시스템은 언젠가는 내/외부적인 요인에 의해서 실패하기 마련임.
- ✓ 일부 모듈 혹은 일부 네트워크 구간이 실패한다고 하더라도 전체 시스템이 멈추어서는 안된다.
- ✓ 이를 위해서 리액티브 시스템은 실패에 대해서 복구할 수 있는 탄력성을 가져야 한다.
- ✓ 예를 들어 클라이언트 시스템에서 서버와의 접속이 단절되었을 때 일반 시스템은 서비스를 멈추지만 리액티브 시스템은 수차례 복구를 시도하고 그래도 접속 불가하면 메시지를 보낸다. 그리고 문제가 되는 일부 기능만 멈춘다.

실패를 관리하는 방안

- ✓ 리액티브 시스템은 시스템을 무결점으로 만들기 보다는 실패를 용인하며 실패에 대해서 적절하게 반응하는 방법에 중점을 둔다.
- ✓ 실패는 언제든지 일어날 수 있으며 예측할 수 없다.
- ✓ 리액티브 시스템은 실패를 관리하기 위해 Supervision과 Actor 개념을 도입하였다.
- ✓ JVM Akka 에서 동작하는 Actor는 이러한 Supervision 아래에 놓여 있으며 Actor의 실패에 대해서 관리한다.

Supervision 개념

1. 수퍼바이저는 자식 노드를 관리하며 자식들의 실패를 관리할 책임이 있다.
2. 수퍼바이저는 자식 노드가 실패할 경우 실패의 이유를 알고 있다.
3. 수퍼바이저는 실패 원인에 따라 적절히 반응한다. (retry, 재시작, 메시지 전달 등)

Load에 대한 반응

- ✓ 리액티브 시스템은 고유한 서비스를 제공하는 것 외에도 Load에 대한 반응을 해야 한다.
- ✓ 현재 이 시스템에 대한 상태를 알아야만 Load에 대한 대처를 할 수 있는데 모니터링 도구를 사용하여 알 수 있다.
- ✓ 배압(Back pressure)은 시스템에서 처리 속도가 늦는 경우 발생하는데 이를 이용하여 Scale-out 할 수 있다.
- ✓ 서킷브레이크(Circuit Breaker)는 오히려 시스템 처리 속도를 늦추기 위해 사용되는데 주로 레거시 시스템 보호를 위해 사용하는 장치이다.

현대적 요구에 부응하는 컴퓨팅 환경

- ✓ 함수형 프로그래밍(Functional Programming) => Scala
- ✓ 액터 모델(The Actor Model) => Akka
- ✓ 이벤트 서버 모델(Evented Server Model) => Play
- ✓ 무상태 아키텍처(Stateless Architecture) => Play
- ✓ 이벤트 소싱(Event Sourcing) => Akka Persistence
- ✓ 리액티브 스트림(Reactive Stream) => Akka Streams

함수형 프로그래밍 핵심

- ✓ immutability
- ✓ Higher- order function
- ✓ 함수 조합

immutability

✓ 변수보다는 변하지 않는 스냅샷을 사용

```
case class Car(brand: String, position: Int)

val car = Car(brand = "DeLorean", position = 0)
val movedCar = car.copy(position = 10)
val movedCarLater = car.copy(position = 30)
```


Higher-order function

- ✓ 다른 함수를 인자로 받거나 그 결과로 함수를 반환하는 함수
- ✓ 데이터를 넘기는 대신 행동을 넘긴다.

```
val users = List(  
  User("Bob", "Marley", 19),  
  User("Jimmy", "Hendrix", 16)  
)  
  
val (minors, majors) = users.partition(_.age < 18)
```



Play Framework으로 Reactive Programming 시작하기

왜 Play Framework 인가 (1/5)

- ✓ 현대의 엔터프라이즈 시장에서 요구되는 리액티브 애플리케이션 특성을 잘 반영함.
- ✓ Java, Scala 언어를 지원하며 full-stack 프레임워크임.
- ✓ Play는 개발자 생산성과 Mobility를 많이 배려한 프레임워크이다.
- ✓ 빠른 개발이 필요할 경우, 초보자의 Learning Curve를 줄이기 위한 대안이 될 수 있음.

Play는 가볍고 Stateless하며 웹에 친숙한 아키텍처를 기반으로 구성되어 있다.



왜 Play Framework 인가 (2/5) – Google 트렌드 분석

✓ Play Framework는 최근 몇 년간 관심이 증가한 사실을 알 수 있다.



왜 Play Framework 인가 (3/5) – 적용 기업

Linked ®

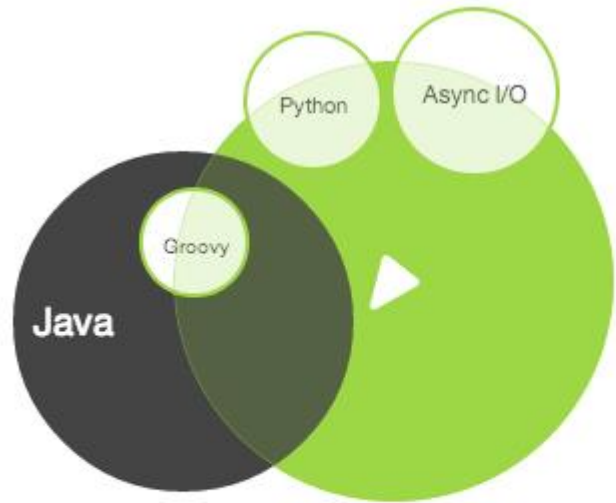
GILT

 **KLOUT**

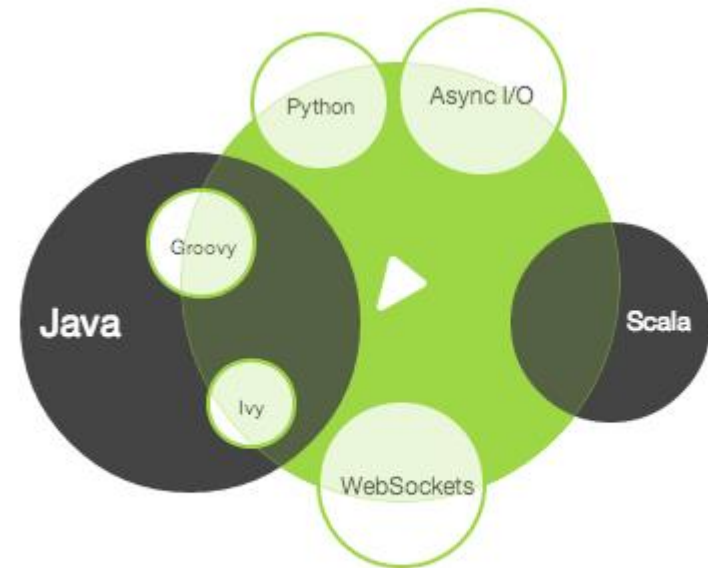
theguardian

왜 Play Framework 인가 (4/5) – Play 1.x

- ✓ Java 기반으로 동작함.
- ✓ Java Convention을 따르지 않음.
- ✓ Java Enterprise Edition API 기반이 아님.
- ✓ 웹 개발자를 위한 손쉬운 프레임워크로 만들어짐.
- ✓ 개발자의 높은 생산성을 위한 프레임워크(Ruby on Rails, Django)



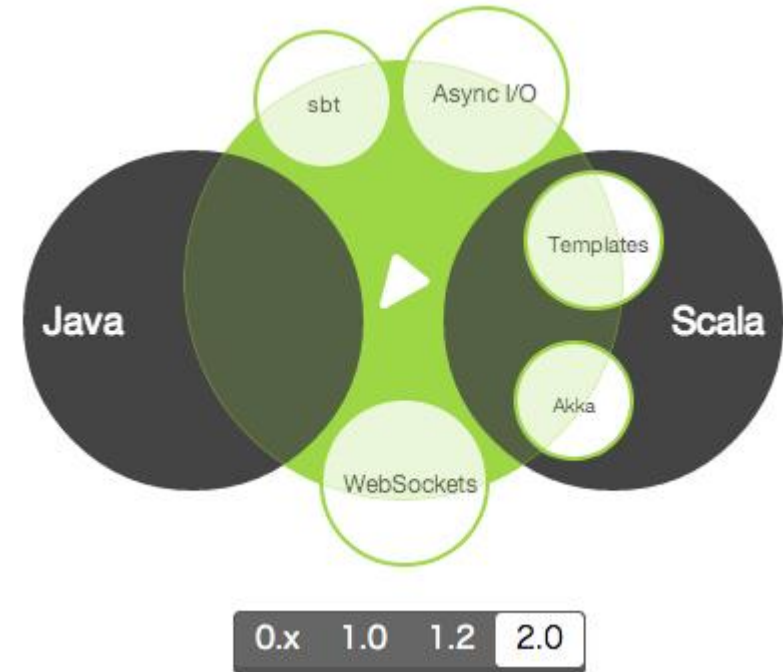
0.x 1.0 1.2 2.0



0.x 1.0 1.2 2.0

왜 Play Framework 인가 (5/5) – Play 2.x

- ✓ Scala 언어로 만들어짐.
- ✓ 그러나 Scala 또는 Java 두가지 버전으로 동작 가능함.
- ✓ 이전 버전에 비하여 Type-safe가 강화됨
- ✓ 무엇보다 개발자 편의를 위한다는 점은 이전 버전과 동일함. (Developer Experience)



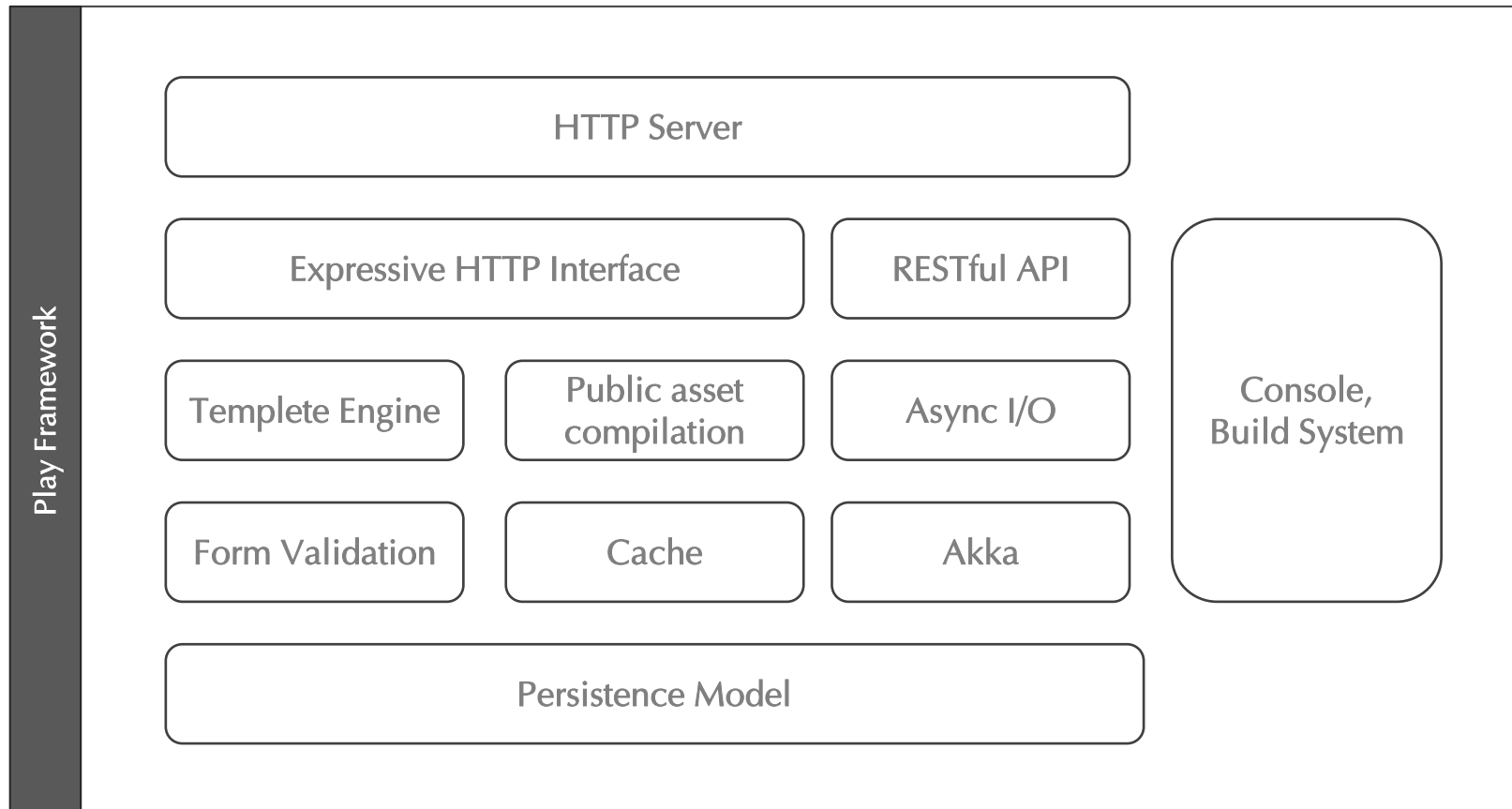
Play Framework 특징 (1/3)

- ✓ 단순함.
- ✓ 직관적인 URL 설정
- ✓ Type-safe
- ✓ HTML5 기술 포함
- ✓ 라이브 코딩
- ✓ Full-stack : 데이터 저장, 보안, 국제화 등
- ✓ 이벤트 기반(Event-driven), 탄력성, 확장성(Scalable)

Play는 Developer-frendly 하다.	
단순함 (Simplicity)	기업형 애플리케이션의 복잡도가 증가될 수록 단순함은 빛을 발한다.프레임워크가 단순하다는 의미가 아니라 단순한 개발을 지원한다는 의미이다.단순함을 지원함으로써 개발자로 하여금 비즈니스 로직에 집중할 수 있게 한다.
Type-safe	일반적으로 Java 코드를 제외한 나머지 Configuration 파일들은 그 오류를 실행중에서야 확인이 가능하다. Play는 이러한 Configuration까지 Type-safe하다.
Full-stack	Full-stack은 All-in-one의 의미로서 하나의 프레임워크로 HTTP부터 데이터 저장 및 빌드까지 가능하다는 특징을 가지고 있다. 그만큼 개발이 빠르고 배포까지의 시간을 단축시켜 준다.

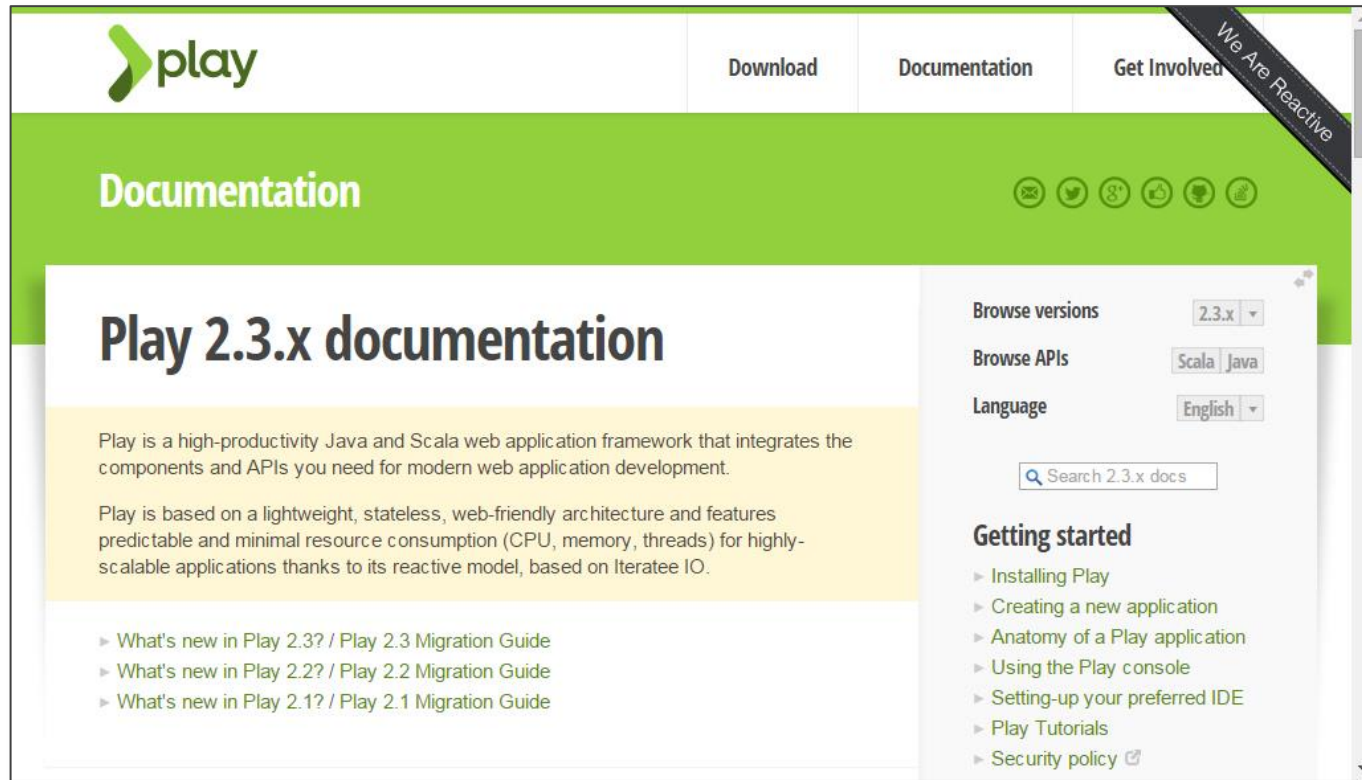
Play Framework 특징 (2/3) – stack

- ✓ Play Framework는 HTTP Server에서 부터 Data Access까지 지원함.
- ✓ 개발을 위한 Console API를 지원하며 자체 빌드 시스템도 갖추.
- ✓ 일반적인 HTTP 인터페이스뿐만 아니라 RESTful API 구성이 가능함.



Play Framework 특징 (3/3) – Full-stack Framework

- ✓ Stack의 각 기능들이 웹 개발에 최적화 됨.
- ✓ 웹 개발에 필요한 모든 기능들이 이미 문서화 되어 있음(가이드를 제공함)
- ✓ 분리된 기능을 제공하는 각 라이브러리 또는 프레임워크들을 익히고 엮는 시간을 줄일 수 있음.



<https://playframework.com/documentation/2.3.x/Home>

리액티브 프로그래밍 환경 설정(Play, Scala)

- ✓ jdk 1.8.x 설치 -> <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- ✓ sbt 설치 -> <http://www.scala-sbt.org/download.html>
- ✓ scala-ide 설치 -> <http://scala-ide.org/download/sdk.html>
- ✓ play framework -> <http://playframework.com>

Scala는 JVM에서 동작하므로 jdk 1.8 이상 버전이 필요함

sbt는 Scala 기반의 빌드 및 의존성 관리 도구임.

scala-ide 는 이클립스 기반의 스칼라 언어를 개발할 수 있는 IDE임.

JDK 1.8 설치

- ✓ <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- ✓ 설치 후 환경변수 설정(JAVA_HOME)

```
java -version
```

```
javac -version
```

```
@echo %JAVA_HOME%
```

SBT 설치

- ✓ <http://www.scala-sbt.org/download.html> 설치버전 실행
- ✓ sbt 초기명령 실행시 의존성 다운로드로 시간이 다소 걸림.
- ✓ Scala repl : sbt console

```
Welcome to Scala version 2.10.6 (Java HotSpot(TM) 64-Bit Server
VM, Java 1.8.0_92).
Type in expressions to have them evaluated.
Type :help for more information.
```

```
scala>
```



Activator 설치(play 2.4의 경우)

- ✓ <https://www.lightbend.com/activator/download>
- ✓ 미니멀 버전을 다운받고(1M 내외) 적당한 위치에 압축해제
- ✓ bin 폴더를 환경변수 path에 추가함.
- ✓ Lightband에서는 activator 대신 sbt를 권장 하나 playframework 2.4 이전 버전의 프로젝트를 생성을 위해서는 activator를 사용할 경우가 있음

Scala-IDE 설치

- ✓ Scala 및 Java 언어를 편집할 수 있는 도구임, Eclipse 기반
- ✓ 해당 폴더에 압축해제
- ✓ 소스 변경 인식 가능 옵션 : Preference > General > Workspace > Refresh using native hooks or pooling 옵션 체크
- ✓ 하이라이트 기능 옵션: Preference > Scala > Editor > Mark occurrence of the selected element... 체크

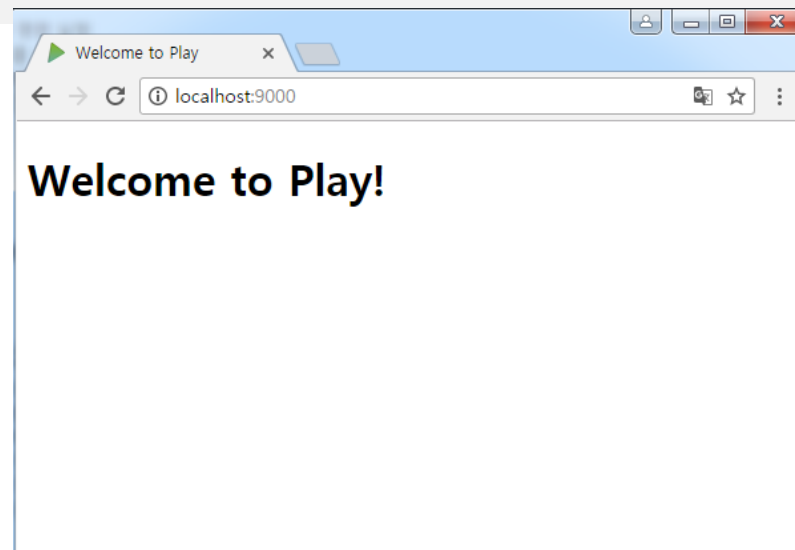


twitter-stream 프로젝트 생성

- ✓ sbt에서 제공하는 템플릿을 사용하여 Play Framework 기반 프로젝트를 생성한다.
- ✓ <https://www.playframework.com/documentation/2.5.x/NewApplication>
- ✓ 프로젝트 생성 후 sbt 명령으로 애플리케이션을 실행한다.
- ✓ <http://localhost:9000>

```
sbt new playframework/play-scala-seed.g8  
or  
activator new twitter-stream play-scala-2.4
```

```
cd twitter-stream  
sbt  
twitter-stream> run
```



Scala-IDE импорт

- ✓ sbteclipse 플러그인 설치 : project/plugins.sbt 에 플러그인 추가
- ✓ sbt 명령모드에서 eclipse with-source=true 명령어 수행
- ✓ Scala-IDE를 열고 프로젝트 Imports를 수행

플러그인 추가

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "5.1.0")
```

의존성 추가

```
libraryDependencies += "com.ning" % "async-http-client" % "1.9.29"
```

프로젝트 импорт 후 스칼라 버전 체크 오류가 나는 경우
Preference > Scala > Compiler > Build Manager >
withVersionClasspathValidator 옵션을 끈다.

컨트롤러 및 라우터

✓ 새로운 tweets 액션을 추가한다.

컨트롤러 추가

```
def tweets = Action {  
  Ok  
}
```

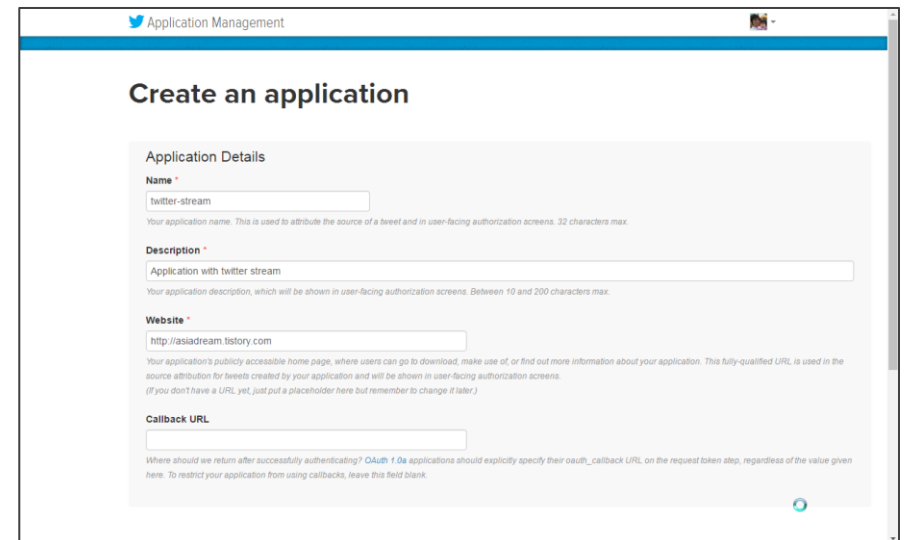
라우터 추가

GET	/tweets	controllers.HomeController.tweets
-----	---------	-----------------------------------

twitter app 등록

- ✓ apps.twitter.com 으로 가서 twitter api를 사용할 수 있도록 twitter app을 등록한다.(개인 정보에 전화번호가 등록되어 있어야 함.)
- ✓ 등록 후 OAUTH 정보를 확인하여 conf/application.conf 에 추가함.

```
# Twitter
twitter.apiKey="<your api key>"
twitter.apiSecret="<your api secret>"
twitter.token="<your access token>"
twitter.tokenSecret="<your access token secret>"
```



The screenshot shows the 'Create an application' page on the Twitter Application Management interface. The page has a blue header with the Twitter logo and 'Application Management' text. The main content area is titled 'Create an application' and contains a form with the following fields:

- Name ***: A text input field containing 'twitter-stream'. Below it is a small note: 'Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.'
- Description ***: A text input field containing 'Application with twitter stream'. Below it is a small note: 'Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.'
- Website ***: A text input field containing 'http://asiadream.tistory.com'. Below it is a small note: 'Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens. (If you don't have a URL, yet, just put a placeholder here but remember to change it later.)'
- Callback URL**: A text input field that is currently empty. Below it is a small note: 'Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.'

At the bottom right of the form, there is a blue circular button with a white plus sign.

Configuration 가져오기

- ✓ 비동기로 설정파일을 읽기 위해 Action.async 를 사용한다.
- ✓ credentials는 ConsumerKey, RequestToken 두가지를 가진 Option이며 설정파일에서 내용을 읽어온다.
- ✓ 설정파일을 잘 가져온 경우 Future(Ok), 문제가 생긴 경우 Future(예외)를 담아 리턴한다.

```
def tweets = Action.async {  
  val credentials: Option[(ConsumerKey, RequestToken)] = for {  
    apiKey <- Play.configuration.getString("twitter.apiKey")  
    apiSecret <- Play.configuration.getString("twitter.apiSecret")  
    token <- Play.configuration.getString("twitter.token")  
    tokenSecret <- Play.configuration.getString("twitter.tokenSecret")  
  } yield(ConsumerKey(apiKey, apiSecret), RequestToken(token, tokenSecret))  
  
  credentials.map { case (consumerKey, requestToken) =>  
    Future.successful(Ok)  
  } getOrElse {  
    Future.successful(InternalServerError("Twitter credentials missing"))  
  }  
}
```

twitter API에 접속하기

- ✓ WS API 는 OAuth 표준으로 Twitter 에 접속할 수 있게 도와줌.
- ✓ "reactive" 라는 단어로 현재 트윗 검색을 시도함.
- ✓ 현재 코드를 실행하면 아무런 결과도 나타나지 않음. 스트림으로 처리하지 않았기 때문임.

```
def tweets = Action.async {  
  credentials.map { case (consumerKey, requestToken) =>  
    WS  
    .url("https://stream.twitter.com/1.1/statuses/filter.json")  
    .sign(OAuthCalculator(consumerKey, requestToken))  
    .withQueryString("track" -> "reactive")  
    .get()  
    .map { response =>  
      Ok(response.body)  
    }  
  } getOrElse {  
    Future.successful(InternalServerError("Twitter credentials missing"))  
  }  
}
```

스트림 출력하기

- ✓ 스트림을 소비하는 `loggingIteratee`를 정의하여 `get` 요청을 보낸다.
- ✓ `get()` 을 사용하여 이전과 다르게 응답을 스트림으로 받는다.
- ✓ `loggingIteratee`는 바이트 배열로 된 스트림을 소비하며 청크를 받아 출력하는 역할을 담당한다.

```
def tweets = Action.async {  
  val loggingIteratee = Iteratee.foreach[Array[Byte]] { array =>  
    Logger.info(array.map(_.toChar).mkString)  
  }  
  
  credentials.map { case (consumerKey, requestToken) =>  
    WS  
    .url("https://stream.twitter.com/1.1/statuses/filter.json")  
    .sign(OAuthCalculator(consumerKey, requestToken))  
    .withQueryString("track" -> "reactive")  
    .get { response =>  
      Logger.info("Status:" + response.status)  
      loggingIteratee  
    }  
    .map { _ =>  
      Ok("Stream Closed")  
    }  
  }  
  .getOrElse {  
    Future.successful(InternalServerError("Twitter credentials missing"))  
  }  
}
```

Play 스트림 API

- ✓ Iteratee는 비동기 스트림 데이터를 소비하기 위한 API임. Iteratee[E, A]는 A를 만들기 위해 청크 E를 소비함.
- ✓ Enumerator는 Iteratee와 반대로 스트림 데이터를 제공함. Enumerator[E] 는 청크 E를 생산함.
- ✓ 스트림 데이터를 변환하는 API는 Enumeratee임. Enumeratee[From, To] 는 From 타입의 청크를 To타입의 청크로 변환함.

추가 Api 사용 위한 환경 설정

- ✓ JSON 관련 API를 사용하기 위해 프로젝트에 의존성을 추가한다.
- ✓ build.sbt에 의존성과 리파지토리 위치를 추가한다.
- ✓ 콘솔에서 update, eclipse 명령을 실행하여 클래스패스가 추가되도록 한다.

build.sbt

```
resolvers += "Typesafe private" at "https://private-repo.typesafe.com/typesafe/maven-releases"  
  
libraryDependencies += "com.typesafe.play.extras" %% "iteratees-extras" % "1.5.0"
```

```
[twitter-stream] $ update
```

```
...
```

```
[twitter-stream] $ eclipse with-source=true
```

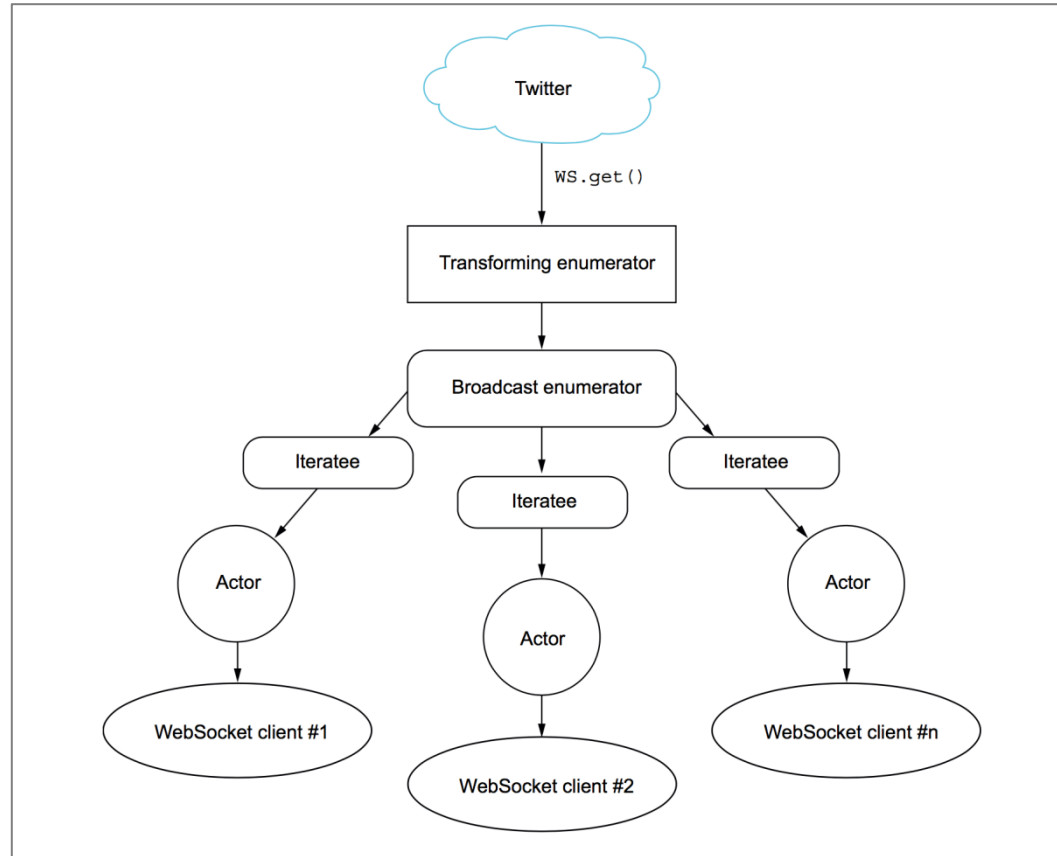
스트리밍 파이프라인 구성

- ✓ 결합된 iteratee, enumerator 을 만든 다음 인코딩 및 json 파싱을 하는 enumeratee 를 차례로 결합하여 jsonStream 을 만든다.
- ✓ 바이트 배열을 소비하는 iteratee는 결국 simpleJson을 만들어내는 파이프라인이 구성되며 iteratee를 응답처리에 추가.

```
def tweets = Action.async {  
  val loggingIteratee = Iteratee.foreach[JsonObject] { value =>  
    Logger.info(value.toString)  
  }  
  
  credentials.map { case (consumerKey, requestToken) =>  
    val (iteratee, enumerator) = Concurrent.joined(Array[Byte])  
  
    val jsonStream: Enumerator[JsonObject] =  
      enumerator &>  
        Encoding.decode() &>  
        Enumeratee.grouped(JsonIteratees.jsSimpleObject)  
  
    jsonStream run loggingIteratee  
  
WS  
  .url("https://stream.twitter.com/1.1/statuses/filter.json")  
  .sign(OAuthCalculator(consumerKey, requestToken))  
  .withQueryString("track" -> "reactive")  
  .get { response =>  
    Logger.info("Status:" + response.status)  
    iteratee  
  }  
  .map { _ =>  
    Ok("Stream Closed")  
  }  
  ...  
}
```

웹소켓을 사용하여 스트리밍 하기

- ✓ 클라이언트의 브라우저는 웹소켓 프로토콜을 사용할 수 있고 Play 프레임워크는 웹소켓을 다루는 API가 준비되어 있음.
- ✓ 그림과 같이 클라이언트의 웹소켓 접속시 Akka의 Actor를 생성하고 지속적으로 이 접속 채널을 통해 스트리밍 한다.
- ✓ Actor는 스레드가 아니며 서로 메시지를 주고 받고 처리하는 가벼운 객체이다.
- ✓ 주로 Actor 간의 메시지 전달이 일어나며 메시지 처리는 비동기로 처리된다.



새로운 Actor 만들기

- ✓ TwitterStreamer는 Actor를 상속받아 생성하며 receive 메소드를 구현하여야 한다.
- ✓ TwitterStreamer Actor는 "subscribe" 메시지를 받는 경우 ActorRef에게 Json 형태의 "Hello, World!" 메시지를 전달한다.

```
import akka.actor.{Actor, ActorRef, Props}
import play.api.Logger
import play.api.libs.json.Json

class TwitterStreamer(out: ActorRef) extends Actor {
  def receive = {
    case "subscribe" =>
      Logger.info("Received subscription from a client")
      out ! Json.obj("text" -> "Hello, world!")
  }
}

object TwitterStreamer {
  def props(out: ActorRef) = Props(new TwitterStreamer(out))
}
```

'!' 기호는 연산자가 아닌 ActorRef 메소드임. 따라서 의미는 메시지를 전달하라는 뜻임.

WebSocket 점점 구성

- ✓ acceptWithActor를 통해서 TwitterStreamer 액터를 생성하는 핸들러를 등록함. out은 접속요청한 클라이언트임.
- ✓ 클라이언트는 웹브라우저에서 WebSocket 접속 요청을 보낸다.
- ✓ 메시지를 받으면 웹브라우저 화면에 메시지를 출력한다.

```
def tweets = WebSocket.acceptWithActor[String, JsValue] { request =>
  out => TwitterStreamer.props(out)
}
```

Application.scala

```
@(message: String)(implicit request: RequestHeader)
```

```
@main(message) {
  <div id="tweets"></div>
  <script type="text/javascript">
    var url = "@routes.Application.tweets().websocketURL()";
    var tweetSocket = new WebSocket(url);

    tweetSocket.onmessage = function (event) {
      console.log(event);
      var data = JSON.parse(event.data);
      var tweet = document.createElement("p");
      var text = document.createTextNode(data.text);
      tweet.appendChild(text);
      document.getElementById("tweets" ).appendChild(tweet);
      document.body.scrollTop = document.body.scrollHeight;
    };

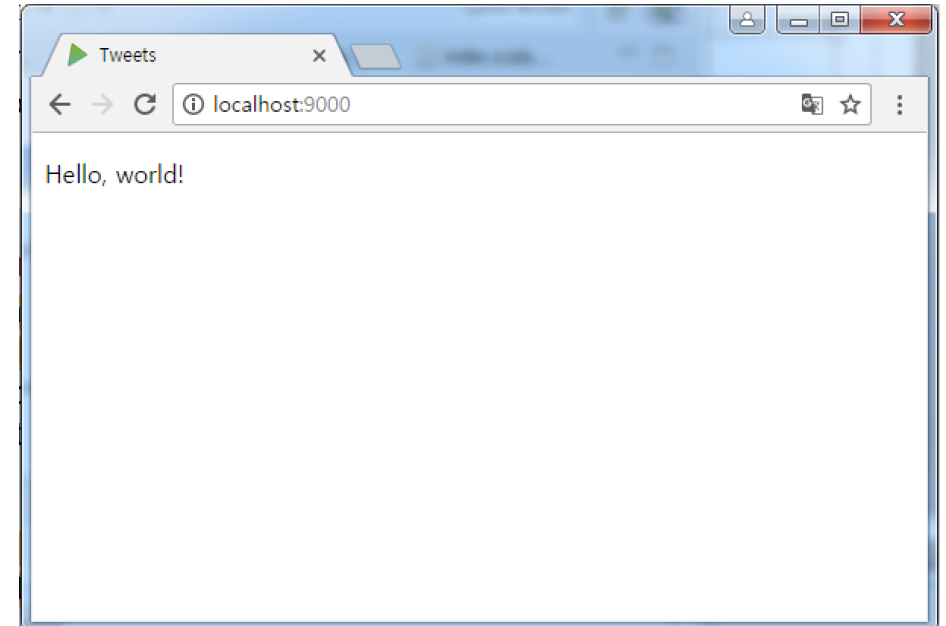
    tweetSocket.onopen = function() {
      tweetSocket.send("subscribe");
    };
  </script>
}
```

index.scala.html

WebSocket 점점 구성(계속)

- ✓ index.scala.html에서 암시적 파라미터 RequestHeader가 필요하므로 index Action에서 암시적 파라미터 선언을 한다.
- ✓ 클라이언트에서 localhost:9000 요청을 보내면 웹소켓 접속을 다시 시도하고 액터가 생성된 후 응답 메시지를 받는다.

```
def index = Action { implicit request =>
  Ok(views.html.index("Tweets"))
}
```



웹소켓에 트윗 메시지 보내기

```
object TwitterStreamer {
  def props(out: ActorRef) = Props(new TwitterStreamer(out))
  private var broadcastEnumerator: Option[Enumerator[JsonObject]] = None

  def connect(): Unit = {
    credentials.map { case (consumerKey, requestToken) =>
      val (iteratee, enumerator) = Concurrent.joined(Array[Byte])

      val jsonStream: Enumerator[JsonObject] = enumerator &>
        Encoding.decode() &>
        Enumeratee.grouped(JsonIteratees.jsSimpleObject)

      val (be, _) = Concurrent.broadcast(jsonStream)
      broadcastEnumerator = Some(be)

      val url = "https://stream.twitter.com/1.1/statuses/filter.json"
      WS
        .url(url)
        .sign(OAuthCalculator(consumerKey, requestToken))
        .withQueryString("track" -> "reactive")
        .get { response =>
          Logger.info("Status: " + response.status)
          iteratee
        }.map { _ =>
          Logger.info("Twitter stream closed")
        }
    } getOrElse {
      Logger.error("Twitter credentials missing")
    }
  }

  def credentials: ...
}
```


웹소켓에 트윗 메시지 보내기(계속)

- ✓ 웹브라우저에 localhost:9000 로 호출해 보기
- ✓ 여러 웹브라우저에도 동일하게 호출해 보기

```
def subscribe(out: ActorRef): Unit = {  
  if (broadcastEnumerator.isEmpty) {  
    connect()  
  }  
  val twitterClient = Iteratee.foreach[JsonObject] { t => out ! t }  
  broadcastEnumerator.foreach { enumerator =>  
    enumerator run twitterClient  
  }  
}
```

object TweetStreamer

```
class TwitterStreamer(out: ActorRef) extends Actor {  
  def receive = {  
    case "subscribe" =>  
      Logger.info("Received subscription from a client")  
      TwitterStreamer.subscribe(out)  
  }  
}
```

Scala 소개

- ✓ 마틴 오더스키 (Martin Odersky) 창시 (2003년)
- ✓ LightBand
- ✓ 함수형 프로그래밍



스칼라 특징

- ✓ 객체지향이면서 함수형 언어임
- ✓ 모든 값은 객체이며 클래스의 인스턴스임
- ✓ 스칼라는 정적 타입이며 확장성이 높음.

인터프리터

✓ 스칼라 시작

인터프리터 시작

```
$ sbt console
Welcome to Scala version 2.10.6 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_92).
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

식

```
scala> 1 + 1
res0: Int = 2

scala> val three = 1 + 2
three: Int = 3
```

변수

✓ 스칼라 변수는 변경 가능한 것(mutable)과 변경 가능하지 않는 것(immutable)이 있다.

변수

```
scala> var name = "han"
name: String = han

scala> name = "kim"
name: String = kim

scala> val age = 5
age: Int = 5

scala> age = 6
<console>:8: error: reassignment to val
    age = 6
      ^
```

함수

✓ 함수는 def를 사용하여 정의할 수 있다.

함수 정의

```
scala> def add(m: Int): Int = m + 1  
add: (m: Int)Int
```

```
scala> val added = add(1)  
added: Int = 2
```

인자없는 함수

```
scala> def three() = 1 + 2  
three: ()Int
```

```
scala> three  
res0: Int = 3
```

```
scala> three()  
res1: Int = 3
```

함수

✓ 함수는 def를 사용하여 정의할 수 있다.

이름없는 함수

```
scala> (x: Int) => x + 1  
res0: Int => Int = <function1>  
  
scala> res0(1)  
res1: Int = 2
```

함수 할당 및 전달

```
scala> val addOne = (x: Int) => x + 1  
addOne: Int => Int = <function1>  
  
scala> addOne(1)  
res0: Int = 2  
  
scala> def format(adder: Int => Int) = adder(1) + " km"  
format: (adder: Int => Int)String  
  
scala> val result = format(addOne)  
result: String = 2 km
```


함수

✓ 커리 함수

커리 함수

```
scala> def multiply(m: Int)(n: Int): Int = m * n  
multiply: (m: Int)(n: Int)Int
```

```
scala> multiply(2)_  
res2: Int => Int = <function1>
```

```
scala> res2(3)  
res3: Int = 6
```

클래스

✓ 클래스

클래스

```
scala> class Calculator {  
    |   val brand: String = "HP"  
    |   def add(m: Int, n: Int): Int = m + n  
    | }
```

```
defined class Calculator
```

```
scala> val calc = new Calculator  
calc: Calculator = Calculator@59aefb8c
```

```
scala> calc.add(1, 2)  
res4: Int = 3
```

```
scala> calc.brand  
res5: String = HP
```

트레이트

✓ 트레이트

트레이트

```
scala> trait Shiny {  
  |   val shineRefraction: Int  
  | }  
defined trait Shiny  
  
scala> class BMW extends Car {  
  |   val brand = "BMW"  
  | }  
defined class BMW  
  
scala> class BMW extends Car with Shiny {  
  |   val brand = "BMW"  
  |   val shineRefraction = 12  
  | }  
defined class BMW
```

타입

✓ 타입은 제너릭과 같은 모든 타입의 값을 처리 할 수 있는 일반적인 함수를 만들 때 사용한다

타입

```
trait Cache[K, V] {  
  def get(key: K): V  
  def put(key: K, value: V)  
  def delete(key: K)  
}  
  
def remove[K](key: K)
```

apply 메소드

✓ apply

```
scala> class Foo {}  
defined class Foo  
  
scala> object FooMaker {  
    |   def apply() = new Foo  
    | }  
defined module FooMaker  
  
scala> val newFoo = FooMaker()  
newFoo: Foo = Foo@3271fa4e
```

```
scala> class Bar {  
    |   def apply() = 0  
    | }  
defined class Bar  
  
scala> val bar = new Bar  
bar: Bar = Bar@48847272  
  
scala> bar()  
res0: Int = 0
```

apply를 정의하면 객체를 마치 함수 호출하듯이 사용할 수 있다. 객체를 호출하면 객체 내의 apply함수가 호출된다.

객체

✓ 객체

객체

```
object Timer {  
    var count = 0  
  
    def currentCount(): Long = {  
        count += 1  
        count  
    }  
}
```

```
Timer.currentCount()
```

```
class Bar(foo: String)
```

```
object Bar {  
    def apply(foo: String) = new Bar(foo)  
}
```

객체는 object 키워드로 선언하며 보통 유일한 객체를 만들기 위해 사용한다. 객체명과 클래스명은 동일해도 된다.

함수

✓ 함수의 실체

함수

```
scala> object addOne extends Function1[Int, Int] {  
  |   def apply(m: Int): Int = m + 1  
  | }  
defined module addOne  
  
scala> addOne(1)  
res1: Int = 2
```

스칼라에서 함수는 객체이다. 다시 말해 Function1~22 의 트레잇의 인스턴스이다.

함수 표현식으로 함수를 정의하는 것은 스칼라에서는 위의 예제와 같이 apply 를 구현한 객체와 같다.

Function1 트레잇은 인자가 하나인 함수를 의미한다.

```
scala> class AddOne extends (Int => Int) {  
  |   def apply(m: Int): Int = m + 1  
  | }  
defined class AddOne  
  
scala> val addOne = new AddOne  
addOne: AddOne = <function1>  
  
scala> addOne(2)  
res2: Int = 3
```


패턴 매치

✓ 패턴 매치

패턴 매치

```
val times = 1

val a = times match {
  case 1 => "one"
  case 2 => "two"
  case _ => "some other number"
}

println(a) // one
```

'_' 키워드는 와일드 카드와 같음. 여기서는 모든 수가 해당됨

조건문을 사용한 경우

```
times match {
  case i if i == 1 => "one"
  case i if i == 2 => "two"
  case _ => "some other number"
}
```

패턴 매치

✓ 패턴 매치

타입 매치

```
def bigger(o: Any): Any = {  
  o match {  
    case i: Int if i < 0 => i - 1  
    case i: Int => i + 1  
    case d: Double if d < 0.0 => d - 0.1  
    case d: Double => d + 0.1  
    case text: String => text + "s"  
  }  
}  
  
println(bigger(5))           // 6  
println(bigger(-5))          // -6  
println(bigger(2.5))         // 2.6  
println(bigger(-2.5))        // -2.6  
println(bigger("hello"))    // hellos
```

서로 다른 타입에 대해서도 매치가 가능함

케이스 클래스

✓ 케이스 클래스는 어떤 값을 손쉽게 저장하고 이용하는 방법을 제공한다. new 를 사용하지 않고 인스턴스 생성이 가능하다.

케이스 클래스

```
scala> case class Calculator(brand: String, model: String)
defined class Calculator

scala> val hp20b = Calculator("HP", "20b")
hp20b: Calculator = Calculator(HP,20b)

scala> val hp20B = Calculator("HP", "20b")
hp20B: Calculator = Calculator(HP,20b)

scala> hp20b == hp20B
```

케이스 클래스는 클래스 정의가 단순하며 객체 생성도 간단하다.
생성 인자에 따른 동등성 검사가 가능하다.

케이스 클래스

✓ 케이스 클래스는 패턴 매칭을 쉽게 하기 위하여 설계되었으며 패턴매칭과의 조합으로 많이 사용된다.

케이스 클래스와 패턴 매칭

```
val hp20b = Calculator("HP", "20B")
val hp30b = Calculator("HP", "30B")

def calcType(calc: Calculator) = calc match {
  case Calculator("HP", "20B") => "financial"
  case Calculator("HP", "48G") => "scientific"
  case Calculator("HP", "30B") => "business"
  case Calculator(ourBrand, ourModel) => "Calculator: %s %s is of unknown type".format(ourBrand,
    ourModel)
}
```

```
case c@Calculator(_, _) => "Calculator: %s of unknown type".format(c)
```

'_' 키워드는 마지막 매치에 주로 사용되며 아무것도 아닌 케이스를 의미한다.

'@' 키워드 앞의 'c'는 패턴 매치에 사용되는 객체를 의미한다.

자료구조

✓ 스칼라 자료구조

리스트, Set, 튜플

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)

scala> Set(1, 1, 2)
res5: scala.collection.immutable.Set[Int] = Set(1, 2)

scala> val hostPort = ("localhost", 80)
hostPort: (String, Int) = (localhost,80)
```

List는 데이터를 나열하는 자료구조임.

Set은 중복 없는 데이터 구조를 가짐.

튜플은 두가지의 값을 묶는 용도로 사용됨. 클래스를 정의하지 않고도 간단히 아이টে을 묶을 수 있다.

✓ 튜플

튜플의 접근

```
scala> val hostPort = ("localhost", 80)
hostPort: (String, Int) = (localhost,80)
```

```
scala> hostPort._1
res6: String = localhost
```

```
scala> hostPort._2
res7: Int = 80
```

튜플의 패턴 매칭

```
def tutType(hostPort: (String, Int)) = hostPort match {
  case ("localhost", 8080) => "localhost:8080"
  case ("127.0.0.1", 90) => "127.0.0.1:90"
  case _ => "Not match"
}
```

```
println(tutType(("localhost", 8080)))
```

튜플을 만드는 다른 방법

```
scala> 1 -> 2
res8: (Int, Int) = (1,2)
```

✓ Map

Map

```
scala> Map(1 -> 2)
res4: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2)

scala> Map("foo" -> "bar")
res5: scala.collection.immutable.Map[String,String] = Map(foo -> bar)

scala> Map(1 -> Map("foo" -> "bar"))
res6: scala.collection.immutable.Map[Int,scala.collection.immutable.Map[String,String]] = Map(1
-> Map(foo -> bar))
```

옵션

✓ 옵션

Option

```
scala> val numbers = Map("one" -> 1, "two" -> 2)
numbers: scala.collection.immutable.Map[String,Int] = Map(one -> 1, two -> 2)

scala> numbers.get("two")
res8: Option[Int] = Some(2)

scala> numbers.get("three")
res9: Option[Int] = None

scala> val result = res9.getOrElse(0) * 2
result: Int = 0
```

값이 존재하거나 존재하지 않을 경우가 있으면 Option을 사용함.
옵션을 클래스이며 하위 클래스로는 Some[T], None이 있음.
옵션을 사용할 때 getOrElse나 패턴 매칭을 사용할 것

함수 조합

map

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)

scala> numbers.map((i: Int) => i * 2)
res10: List[Int] = List(2, 4, 6, 8)
```

리스트의 모든 원소에 대하여 입력된 함수를 적용하고 그 결과를 새로운 리스트로 반환한다.

foreach

```
scala> numbers.foreach((i: Int) => i * 2)

scala> numbers
res12: List[Int] = List(1, 2, 3, 4)
```

map과 비슷하나 새로운 결과를 반환하지 않는다.

함수 조합

filter

```
scala> numbers.filter((i: Int) => i % 2 == 0)
res13: List[Int] = List(2, 4)
```

참/거짓을 반환하는 함수를 입력해야 하며 함수가 수행된 결과가 참인 경우 필터링 된 새로운 리스트를 리턴한다.

zip

```
scala> List(1, 2, 3).zip(List("a", "b", "c"))
res15: List[(Int, String)] = List((1,a), (2,b), (3,c))
```

또 다른 리스트를 입력 받아 튜플을 원소로 하는 리스트를 반환한다.

partition

```
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> numbers.partition(_ % 2 == 0)
res16: (List[Int], List[Int]) = (List(2, 4, 6, 8, 10), List(1, 3, 5, 7, 9))
```

입력된 함수의 조건에 따라 리스트를 둘로 나누어 튜플로 리턴한다.

함수 조합

find

```
scala> numbers.find(_ > 5)
res17: Option[Int] = Some(6)
```

입력된 함수를 만족하는 원소 중 첫번째 원소만 리턴한다.

flatten

```
scala> List(List(1, 2), List(3, 4)).flatten
res20: List[Int] = List(1, 2, 3, 4)
```

리스트 타입이 겹친 경우 하나로 풀어준다.

flatMap

```
scala> nestedNumbers.flatMap(x => x.map(_ * 2))
res21: List[Int] = List(2, 4, 6, 8)

scala> nestedNumbers.map((x: List[Int]) => x.map(_ * 2))
res23: List[List[Int]] = List(List(2, 4), List(6, 8))

scala> res23.flatten
res24: List[Int] = List(2, 4, 6, 8)
```

flatMap은 map과 flatten을 합친 것이다. 우선 내부적으로 map을 수행 후 그 결과 중첩된 리스트를 하나로 풀어준다.



함수형 프로그래밍 핵심

함수형 프로그래밍 정의

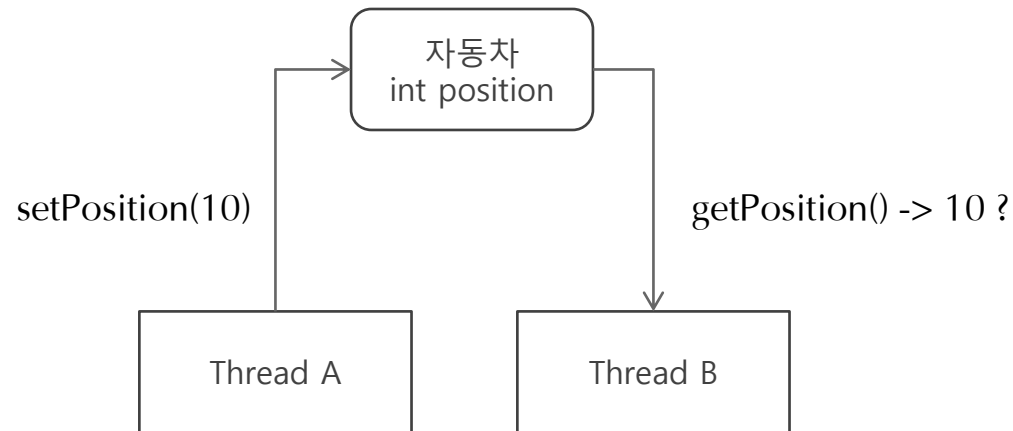
- ✓ 단어 그대로 함수를 다루는 프로그래밍이나 아주 폭넓은 의미를 가지고 있다.
- ✓ 로버트 C. 마틴 에 의하면 함수형 프로그래밍이란 할당문이 없이 프로그래밍을 하는 것이다.
- ✓ 이것은 어떤 변수의 immutability 특성을 의미하며 함수 내의 코드가 실행되는 동안 상태가 변하지 않는다는 의미이다.
- ✓ 상태가 변하지 않는다는 것은 상태가 변함으로서 발생하는 사이드 이펙트가 없다는 의미이다.
- ✓ 즉 함수형 프로그래밍이라는 것은 상태를 간직하지 않으며 사이드 이펙트가 존재하지 않은 함수기반의 프로그래밍이라는 의미이다.

Functional programming is programming without assignment statements.

Robert C. Martin, Functional Programming Basics

변하는 값의 불확실성 극복

- ✓ 지속적으로 변하는 자원에 대해 두 스레드가 공유한다면 불확실성으로 프로그래밍이 어려워 짐.
- ✓또는 객체지향 프로그래밍에서 객체의 상태도 지속적으로 변한다면 마찬가지로 상황이 발생함.
- ✓이러한 불확실성을 극복하기 위해 함수형 프로그래밍에서는 immutable 값을 사용하는데 명령형 프로그래밍 스타일에서는 적용하기 어려움
- ✓immutable을 사용한다는 의미는 값의 참조 대신 값을 직접 전달한다는 의미임.
- ✓함수는 immutable 값을 다루는데 최적화 되어 있음.



Expression-oriented Programming(표현식 프로그래밍)

- ✓ 명령형 프로그래밍은 코드 수행시 아무것도 리턴하지 않은 반면 표현식 프로그래밍은 값을 리턴한다.
- ✓ 이것은 다른 말로 명령형은 자신의 코드 수행 범위 외의 어떤 상태를 변경시킬 수 있다는 의미이며
- ✓ 표현식 프로그래밍은 외부의 어떠한 값도 변경하지 않는다는 의미이다.
- ✓ 표현식 프로그래밍은 외부 값의 수정에 의한 사이드 이펙트를 발생시키지 않는다.

```
public static void removeElement(List<String> list, String toRemove) {  
    int index = 0;  
    for (String s : list) {  
        if (s.equals(toRemove)) {  
            list.remove(index);  
        }  
        index++;  
    }  
}
```

Statement(명령형)

```
public static List<String> filterNot(List<String> list, String toRemove) {  
    List<String> filtered = new LinkedList<String>();  
    for (String s : list) {  
        if (!s.equals(toRemove)) {  
            filtered.add(s);  
        }  
    }  
    return filtered;  
}
```

Expression(표현형)

스칼라의 Expression-oriented programming

- ✓ 스칼라 언어는 표현식 기반으로 설계됨.
- ✓ 표현식 프로그래밍은 immutable 값으로도 충분히 값을 조작할 수 있다.
- ✓ 스칼라는 이러한 표현식을 쉽게 구사하기 위한 장치를 제공한다.(match)

```
def spellOut(number: Int): String = number match {  
  case 1 => "one"  
  case 2 => "two"  
  case 42 => "forty-two"  
  case _ => "unknown number"  
}  
  
val fortyTwo = spellOut(42)
```

지금까지 명령형 프로그래밍에서 수많은 if-else 및 데이터 변환, 변수 수정, 변수 범위 밖의 데이터 조작 등 복잡한 코드들을 만들어 왔다. 표현식 프로그래밍은 명령형 프로그래밍에서 보이고 있는 이러한 복잡성과 오류가능성을 줄이기 위한 방법이다. 표현식 프로그래밍은 논리 흐름을 되도록 작게 유지하고 이러한 단위 논리블록을 결합하여 복잡한 메커니즘을 구현한다.

우리는 이러한 단위 논리블록을 함수라고 한다.

Object Oriented 언어에서 함수

- ✓ OO 프로그래밍에서 캡슐화는 전체 코드의 복잡도를 줄이고 유지 보수성을 높이는데 기여하였다.
- ✓ 그러나 캡슐화는 데이터 가변성(mutable) 문제를 해결하지 못하고 있다. 가변성은 객체의 상태 추론을 어렵게 만들고 있다.
- ✓ 객체 내의 메소드는 객체가 독점하고 있어 로직의 재사용에 방해가 된다.
- ✓ Java언어로 진행하는 프로젝트에서는 항상 utils 패키지가 발견되는데 이는 객체 상태와 상관없는 재사용 가능한 로직들이 모여있다. 이들은 1차적으로 함수의 후보가 된다.

함수의 정의

- ✓ 함수는 언어에서 일차 시티즌이므로 변수에 할당할 수 있고 다른 함수의 파라미터로 넘길 수 있다.
- ✓ 객체지향 언어에서의 메소드와는 다르게 함수는 행위를 캡슐화 하고 당장 실행되지 않아도 된다.
- ✓ 스칼라에서 함수의 정의는 다양하게 할 수 있다.

def 키워드 사용(메소드로 선언)

```
def square(x: Int): Int = x * x
```

함수 리터럴 사용

```
val square = (x: Int) => x * x
```

함수 리터럴은 Function1, Function2, ... 타입의 객체이다.
따라서 실제 타입은 다음과 같다.

```
val square: Function1[Int, Int] = (x: Int) => x * x
```

함수의 전달

- ✓ 비동기 프로그래밍에서 함수의 실행을 연기하고 파라미터도 전달할 수 있는 기능은 아주 중요하다.
- ✓ 스칼라에서는 메소드 형태로 정의된 것을 함수처럼 변수에 할당할 수 있다.
- ✓ 한편 자바 언어에는 함수 전달 기능이 없으므로 Runnable 인터페이스 처럼 인터페이스를 구현하는 객체를 넘기는 방식을 사용하다.

```
scala> def square(x: Int): Int = x * x
square: (x: Int)Int

scala> val squareLiteral = square
<console>:8: error: missing arguments for method square;
follow this method with `_' if you want to treat it as a partially applied function
    val squareLiteral = square
                        ^

scala> val squareLiteral = square _
squareLiteral: Int => Int = <function1>
```

함수의 조합

- ✓ 고차함수(higher-order function)는 함수를 파라미터로 갖는 함수를 의미하며 기능을 추상화 하는 좋은 방법이다.
- ✓ 함수를 추상화하면 응용범위가 넓어진다.

고차함수 사용

```
scala> val square: Function1[Int, Int] = (x: Int) => x * x
square: Int => Int = <function1>

scala> def fourth(x: Int, squarer: Function1[Int, Int]): Int = squarer(squarer(x))
fourth: (x: Int, squarer: Int => Int)Int

scala> val twoToThePowerOfFour = fourth(2, square)
twoToThePowerOfFour: Int = 16
```

함수의 추상화

```
def applyTwice(x: Int, f: Int => Int): Int = f(f(x))

def fourth(x: Int) = applyTwice(x, y => y * y)
def quadruple(x: Int) = applyTwice(x, y => 2 * y)
```

함수의 크기

- ✓ 기능이 복잡할 수록 함수의 길이가 길어진다. 긴 함수는 프로그램 실행에 영향을 주지 않지만 가독성에 영향을 준다.
- ✓ 함수나 코드 블록이 여러 단계로 겹쳐지면 이해하기가 어려워 지지만 작은 함수 단위로 나누면 읽기가 편해진다.

```
def computeYearlyAggregates(clickRepository: ClickRepository): Map[Long, Seq[(Month, Int)]] = {  
  val pastClicks = clickRepository.getClicksSince(DateTime.now.minusYears(1))  
  pastClicks.groupBy(_.advertisementId).mapValues {  
    case clicks =>  
      val monthlyClicks = clicks  
        .groupBy(click => Month(click.timestamp.getYear, click.timestamp.getMonthOfYear))  
        .map { case (month, groupedClicks) =>  
          month -> groupedClicks.length  
        }.toSeq  
      monthlyClicks  
    }  
  }  
}
```



```
def computeYearlyAggregatesRefactored(clickRepository: ClickRepository): Map[Long, Seq[(Month, Int)]] = {  
  def monthOfClick(click: Click) = Month(click.timestamp.getYear, click.timestamp.getMonthOfYear)  
  def countMonthlyClicks(monthlyClicks: (Month, Seq[Click])) = monthlyClicks match {  
    case (month, clicks) =>  
      month -> clicks.length  
  }  
  
  def computeMonthlyAggregates(clicks: Seq[Click]) = clicks.groupBy(monthOfClick).map(countMonthlyClicks).toSeq  
  
  val pastClicks = clickRepository.getClicksSince(DateTime.now.minusYears(1))  
  pastClicks.groupBy(_.advertisementId).mapValues(computeMonthlyAggregates)  
}
```

함수 조합의 원칙

- ✓ 하나의 함수는 하나의 기능만 수행한다.
- ✓ 함수의 이름은 그 함수의 기능을 대변한다. 만일 함수의 이름이 길어지면 그 함수는 하나 이상의 기능을 수행할 가능성이 있다.
- ✓ 함수 하나가 여러가지 기능을 수행하기 보다는 단위기능을 수행하는 함수를 조합한다.
- ✓ 함수가 또 다른 함수를 전달하는 방법으로 함수를 조합할 수 있다.

핵심 기능을 하는 단위 함수를 정의하고 이를 조합하여 쌓아 올라가는 함수 조합 방식이 함수형 프로그래밍에서 중요하다.

Immutable 컬렉션 – 반복 보다 변환으로

- ✓ 어떤 컬렉션에서 두가지로 분류하는 기능을 구현할 때 아래와 같은 두가지 방법을 생각해 볼 수 있다.
- ✓ 두번째 방법은 우선 루프가 존재하지 않는다.
- ✓ 두번째 방법에서 함수가 중요한 역할을 담당한다. 비교하는 함수를 인자로 넘긴다.
- ✓ 두번째 방법은 상태를 변경하지 않는다.

```
List<User> minors = new ArrayList<User>();  
List<User> majors = new ArrayList<User>();  
  
for(int i = 0; i < users.size(); i++) {  
    User u = users.get(i);  
    if(u.getAge() < 18) {  
        minors.add(u);  
    } else {  
        majors.add(u);  
    }  
}
```



```
val (minors, majors) = users.partition(_.age < 18)
```

Immutable 컬렉션 – 고차함수 활용

- ✓ 고차함수는 컬렉션은 어떻게 변환할 것인지 선언적으로 지정할 때 매우 유용함.
- ✓ map은 함수를 입력받아 컬렉션의 각 Element에게 모두 적용하여 그 결과를 컬렉션으로 리턴한다. 그러나 원본 컬렉션은 변경하지 않는다.
- ✓ flatMap은 map과 다르게 결과에 컬렉션 타입이 겹친 경우 flattens 단계를 거쳐 단일 컬렉션으로 가공하여 리턴한다.

map

```
scala> val list1 = List(1, 2, 3)
list1: List[Int] = List(1, 2, 3)

scala> val doubles = list1.map(_ * 2)
doubles: List[Int] = List(2, 4, 6)

scala> list1
res0: List[Int] = List(1, 2, 3)
```

flatMap

```
scala> def f(i: Int) = List(i * 2, i * i)
f: (i: Int)List[Int]

scala> val flatMapped = list1.flatMap(f)
flatMapped: List[Int] = List(2, 1, 4, 4, 6, 9)

scala> val mapped = list1.map(f)
mapped: List[List[Int]] = List(List(2, 1), List(4, 4), List(6, 9))
```


Immutable 컬렉션 – for의 활용

- ✓ for를 사용하면 여러 컬렉션을 결합하고 다양한 방법으로 필터링 할 수 있음.
- ✓ map, flatMap, withFilter를 결합한 기능과 같음.
- ✓ 간단한 방법으로 효과적으로 결과를 가공할 수 있으므로 잘 사용되며 특히 컬렉션 조합의 용도로 쓰인다.

```
scala> val aList = List(1, 2, 3)
aList: List[Int] = List(1, 2, 3)

scala> val bList = List(4, 5, 6)
bList: List[Int] = List(4, 5, 6)

scala> val result = for {
  |   a <- aList
  |   if a > 1
  |   b <- bList
  |   if b < 6
  | } yield a + b
result: List[Int] = List(6, 7, 7, 8)
```

Option

- ✓ Option은 어떤 값을 감싼 형태이며 그 값이 있을 수도 있고 없을 수도 있다.
- ✓ NullPointerException에 지친 자바개발자들을 스칼라 영역으로 끌고 오는 동기가 되기도 한다.
- ✓ Option은 개발자로 하여금 null 체크를 항상 각인시킨다. 적어도 NullPointerException으로 고생하지 않아도 된다.

```
scala> val box = Option("Cat")
box: Option[String] = Some(Cat)
```

```
scala> box.isEmpty
res0: Boolean = false
```

```
scala> box.isDefined
res1: Boolean = true
```

```
scala> box == None
res2: Boolean = false
```

```
if (box != None) {
  val contents = box.get
  process(contents)
} else {
  reportError()
}
```

```
val user: Option[User] = User.findById(42)
val fullName: Option[String] = user.map { u =>
  u.fullName
}
```

```
-----
val user: Option[User] = User.findById(42)
val fullName: String = user.map { u =>
  u.fullName
} getOrElse {
  "Unknown user"
}
```

```
val user: Option[User] = User.findById(42)
val address: Option[Option[Address]] = user.map { u
=>
  Address.findById(u.addressId)
}
```

```
-----
val user: Option[User] = User.findById(42)
val address: Option[Address] = user.flatMap { u =>
  Address.findById(u.addressId)
}
```

프로그래밍 스타일의 전환

- ✓ Immutability, 고차함수, 함수 조합은 함수형 프로그래밍의 핵심이다.
- ✓ 이러한 프로그래밍 스타일은 선언적인 프로그래밍 스타일을 보여주므로 기존 명령형에 익숙해진 개발자들의 접근이 어렵다.
- ✓ 함수형 프로그래밍으로 전환하는데 가장 빠른 방법은 스칼라와 같은 함수형 언어로 프로젝트를 진행하는 것임.

1. Option 내의 값을 꺼내지 말 것.
2. Immutable 값 또는 데이터 구조만 사용할 것.
3. 함수를 작게 유지하고 그 목적을 명확히 할 것.
 - 함수 네이밍은 데이터 구조 보다 어떤 일을 하는지 그 의미가 포함되어야 함.
4. 반복적이고 점진적으로 함수형 스타일로 변경할 것



Future

Future

- ✓ Future는 스칼라 리액티브 프로그래밍에서 기본으로 사용된다.
- ✓ 의미 그대로 아직 일어나지 않은 행위에 대한 결과를 의미한다. 즉 미래의 값을 가지고 있다.
- ✓ 실패에 대해 효율적으로 다룬다.

Future의 이해

- ✓ Option과 같이 Future는 모다드 데이터 구조를 나타낸다. 모나드이란 어떤 타입을 감싼 또 다른 타입을 의미한다.
- ✓ 여러 Future들을 쉽게 조합할 수 있다. (여러 비동기 함수에서 실행된 결과를 조합할 수 있다.)
- ✓ Future는 콜백 스타일을 쉬운 형태로 추상화 한 타입이다.

* 콜백 지옥 (callback hell)

- 비동기 api 호출시 순차적인 처리를 위해 콜백 스타일의 코딩을 한다.
- 그러나 이러한 콜백은 순차처리를 위해 겹쳐지면 코드 가독성이 급격히 떨어진다.
- 이를 콜백 지옥이라 하며 이러한 단점을 극복하기 위한 여러가지 방안들이 제시되었는데 그 중 Future는 자연스럽게 이를 극복해 준다.

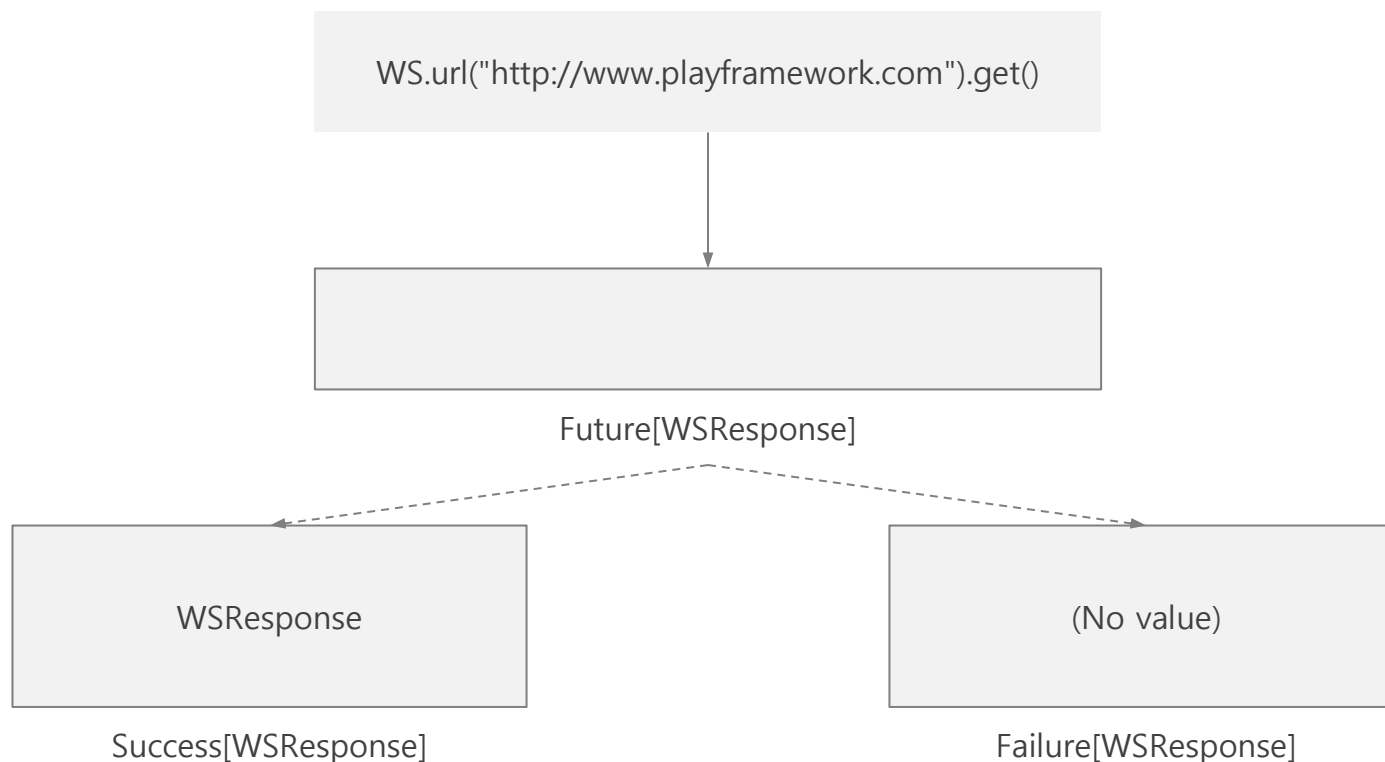
모나드 이해 : <http://tech.kakao.com/2016/03/03/monad-programming-with-scala-future/>

비동기 코딩 스타일의 단점 극복

- ✓ Future는 비동기 프로그래밍 스타일의 단점을 앞으로 일어날 미래의 값이라는 부분을 추상화 하여 극복하였다.
- ✓ 비동기 수행 결과를 캡슐화 하였으며 조합이 가능하다.
- ✓ 실패에 대한 처리를 투명하게 할 수 있으며 예외는 Future 체인을 따라 전파된다.
- ✓ 비동기 작업에 대한 스레드 풀 설정이 가능하다.

Future의 기본

- ✓ `scala.concurrent.Future[T]` 의 스칼라 유형으로 정의됨. T타입의 값이 미래에 저장되는 Box 개념임.
- ✓ Future가 정의 되는 순간 비동기 작업이 시작됨.
- ✓ Future를 받는 순간에는 비동기 작업이 시작되었으나 아직 결과는 나오지 않음. 그러나 바로 다음 코드라인으로 넘어감.
- ✓ Future가 완료되면 성공 또는 실패 상태로 변경된다. 성공인 경우 해당 타입의 값이 존재하고 실패면 예외를 가지게 됨.



Future의 변환

- ✓ Future의 장점 중의 하나는 값이 아직 나오지 않았는데도 미리 변환 작업이 가능하다는 점이다.
- ✓ 따라서 이의 조합을 통해 복잡한 비동기적 연산 파이프라인을 구축할 수 있다.
- ✓ Expression-oriented Programming 에 따라 Future는 소규모의 많은 단계를 조합하는 수단으로 사용됨.

```
val response: Future[WSResponse] = WS.url("http://www.playframework.com").get()

val siteOnline: Future[Boolean] = response.map { r =>
  r.status == 200
}

siteOnline.foreach { isOnline =>
  if(isOnline) {
    println("The Play site is up")
  } else {
    println("The Play site is down")
  }
}
```

1. Future[WSResponse] 타입 생성.
2. Future[Boolean] 타입으로 변경
3. 테스크 완료 후 처리

실패한 Future 처리

- ✓ Future는 항상 성공하지 않는다. 실패의 경우 Future는 실패의 이유를 기억하고 있다.
- ✓ 일반적인 명령형 프로그래밍에서 예외나 실패가 발생한 경우 코드 실행이 중단되고 바로 exception을 던진다.
- ✓ Future는 일반적인 try... catch 블록 대신 실패에 대한 유연한 처리를 가능하게 한다.
- ✓ recover 또는 recoverWith 함수를 사용하며 실패에 대한 처리를 정의할 수 있다.

```
val response: Future[WSResponse] = WS.url("http://www.playframework.com").get()

val siteAvailable: Future[Option[Boolean]] = response.map { r =>
  Some(r.status == 200)
} recover {
  case ce: java.net.ConnectException => None
}
```

recoverWith : 현재 코드에서 예외 상황을 처리 할 수 있을 경우 사용
recover : 현재 코드에서 처리할 수 없고 다른 지점에서 처리할 경우 사용

Future의 조합

- ✓ 스칼라 언어의 가장 강력한 기능 중의 하나가 바로 Future의 조합이다.
- ✓ 기존 명령형 프로그램은 두가지 작업 결과를 조합할 경우 순차적으로 두가지 작업이 모두 종료된 후 가능하지만
- ✓ Future는 비동기 작업 두가지를 병행하며 작업결과를 조합할 수 있다.
- ✓ 또한 조합 후 실패에 대한 처리도 가능하다.

```
def siteAvailable(url: String): Future[Boolean] =  
  WS.url(url).get().map { r =>  
    r.status == 200  
  }  
  
val playSiteAvailable = siteAvailable("http://www.playframework.com")  
  
val playGithubAvailable = siteAvailable("https://github.com/playframework")  
  
val allSitesAvailable: Future[Boolean] = for {  
  siteAvailable <- playSiteAvailable  
  githubAvailable <- playGithubAvailable  
} yield (siteAvailable && githubAvailable)
```

두 사이트의 상태를 비동기로 병행하여 체크하며 그 체크 결과를 조합한다.

Future의 실행

- ✓ Future가 실제로 실행되기 위해서는 ExecutionContext가 필요하다. 이는 실행을 위한 스레드 풀을 의미한다.
- ✓ Scala concurrent 라이브러리는 기본 실행 컨텍스트를 제공한다.
- ✓ 기본 실행 컨텍스트 외에 커스텀 실행 컨텍스트를 설정 할 수도 있다.
- ✓ 커스텀 실행 컨텍스트를 object 속성으로 정의하면 다른 코드에서 임포트 후 사용 가능하다.

```
import scala.concurrent._
import java.util.concurrent.Executors

implicit val ec = ExecutionContext.fromExecutor(
  Executors.newFixedThreadPool(2)
)

val sum: Future[Int] = Future { 1 + 1 }
sum.foreach { s => println(s) }
```

2개의 스레드를 가지는 고정 스레드풀을 생성한다.

실행컨텍스트를 implicit로 지정하여 Future가 이를 따르도록 한다.

언제 Future를 사용하는가...

- ✓ 주로 블록킹이 존재하는 api 또는 라이브러리를 호출할 때 Future를 사용한다.
- ✓ 블록킹은 주로 IO를 다루는 API에서 발행한다.(파일, 네트워크...)
- ✓ 블록킹 코드를 Future로 감싸면 현재 실행 컨텍스트에서는 비동기로 호출하겠지만 Future 블록은 여전히 블록킹 상태임.
- ✓ 이는 단지 실행 컨텍스트만 바뀔 뿐이며 오히려 컨텍스트 스위칭 비용이 더 들어감.

```
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.Global
import java.io.File

def fileExists(path: String): Future[Boolean] = Future {
  blocking {
    new java.io.File(path).exists
  }
}
```

블록킹을 가지는 자바 File Api를 Future로 감싼다.

그러나 여전히 API 호출은 블록킹이므로 blocking 블록으로 명시하거나 실행 컨텍스트를 명시하여 스레드 독점 상황을 조절하여야 한다.

Play와 Future

- ✓ Play는 컨트롤러 작업을 비동기로 수행하는 방법을 제공한다. 그 결과는 Future이다.
- ✓ Action.async 빌더는 {request => Future[Result]} 유형의 함수가 필요하다.
- ✓ Play는 Action.async 가 비동기임을 이미 알고 있기 때문에 Future 블록을 사용하지 않았다.
- ✓ 이 컨트롤러는 비동기로 수행될 것이다.└

```
import play.api.libs.ws._
import scala.concurrent._
import play.api.libs.concurrent.Execution.Implicits._
import play.api.Play.current
def availability = Action.async {
  val response: Future[WSResponse] =
    WS.url("http://www.playframework.com").get()
  val siteAvailable: Future[Boolean] = response.map { r =>
    r.status == 200
  }
  siteAvailable.map { isAvailable =>
    if(isAvailable) {
      Ok("The Play site is up.")
    } else {
      Ok("The Play site is down!")
    }
  }
}
```

블록킹, 논블록킹 컨트롤러

- ✓ Action.async 빌더는 블록킹 I/O를 사용하거나 CPU 를 소모하는 작업을 처리할 때 유용함.
- ✓ Action 빌더로 생성된 컨트롤러는 내부 작업이 논블록킹인 경우 사용하여야 한다.
- ✓ Action 인 경우 Play는 디폴트 작업자 풀에서 수행하려고 한다.
- ✓ 리액티브 웹 애플리케이션 작업에서 중요한 점은 어떤 지점에서 블록킹이 일어나는 지 식별하여야 하는 점이다.

```
def listFiles = Action { implicit request =>
  val files = new java.io.File(".").listFiles
  Ok(files.map(_.getName).mkString(", "))
}
```

이 경우 컨트롤러 내부작업이 블록킹이므로 서버 전체 성능을 떨어뜨린다.
왜냐하면 이 작업이 디폴트 스래드를 점유하기 때문이다.

Play의 탄력성 – 사용자 정의 에러 핸들링

- ✓ Play는 디폴트 에러 핸들링 기능이 있음.
- ✓ 사용자 정의 에러 핸들러를 만들 수도 있는데 REST API를 구축할 경우 유리하다.
- ✓ 예제는 여러 유형의 에러를 처리 할 수 있는 핸들러임.

```
def authenticationErrorHandler: PartialFunction[Throwable, Result] = {  
  case UserNotFoundException(userId) =>  
    NotFound(  
      Json.obj("error" -> s"User with ID $userId was not found")  
    )  
  case UserDisabledException(userId) =>  
    Unauthorized(  
      Json.obj("error" -> s"User with ID $userId is disabled")  
    )  
  case ce: ConnectionException =>  
    ServiceUnavailable(  
      Json.obj("error" -> "Authentication backend broken")  
    )  
}  
val authentication: Future[Result] = ???  
val recoveredAuthentication: Future[Result] = authentication.recover(authenticationErrorHandler)
```


Play의 탄력성 – 타임아웃 핸들링

- ✓ 외부 서비스를 호출하는 경우 사용자가 너무 오래 기다리지 않도록 타임아웃 시간을 설정하는 것이 좋음.
- ✓ 인터넷은 신뢰할 수 없는 구간이며 언제든지 원격 접속이 지연되거나 단절될 수 있음.
- ✓ Play는 응답 시간을 설정하여 시간이 초과되면 적절히 retry를 한다든지 대체 응답을 한다든지 대응을 할 수 있음.

```
import play.api.libs.concurrent.Promise
import scala.concurrent.duration._

case class AuthenticationResult(success: Boolean, error: String)

def authenticate(username: String, password: String) = Action.async {
  implicit request =>
    val authentication: Future[AuthenticationResult] =
      authenticationService.authenticate(username, password)
    val timeoutFuture = Promise.timeout(
      "Authentication service unresponsive", 2.seconds
    )
    Future.firstCompletedOf(
      Seq(authentication, timeoutFuture)
    ).map {
      case AuthenticationResult(success, _) if success =>
        Ok("You can pass")
      case AuthenticationResult(success, error) if !success =>
        Unauthorized(s"You shall not pass: $error")
      case timeoutReason: String =>
        ServiceUnavailable(timeoutReason)
    }
}
```

실행 컨텍스트 설정

- ✓ Play는 임포트 문 `import play.api.libs.concurrent.Execution.Implicits._` 을 사용하여 가져올 수 있는 기본 실행 컨텍스트가 존재함.
- ✓ Play의 기본 실행 컨텍스트는 Akka 디스패처에 의해 지원되며 Play 자체 설정에서 조정 가능함.
- ✓ Akka는 동시 프로그래밍을 위한 도구이며 Akka 디스패처는 스레드 실행 전략을 세부적으로 구성할 수 있는 방법을 제공함

```
akka {  
  actor {  
    default-dispatcher {  
      fork-join-executor {  
        parallelism-factor = 1.0  
        parallelism-max = 24  
        task-peeking-mode = LIFO  
      }  
    }  
  }  
}
```

기본 실행 컨텍스트, CPU 코어 당 1의 스레드 생성, 최대 24개까지, 작업 선택 방법: LIFO
이 설정은 블록킹 I/O 나 CPU 소모 작업이 없이 진정으로 비동기 방식으로 구축된 경우 적합함.

스레드 모델로 돌아가기

- ✓ 코드가 상당부분 동기적이고 리소스가 제한적이어서 많은 작업을 수행할 수 없는 경우
- ✓ 이벤트 방식을 포기하고 스레드 모델로 돌아가는 것도 한 방법이다.
- ✓ 스레드 모델로 돌아가는 것은 효율적이지 않지만 비동기를 고려하지 않는 기존 코드를 동작시킬 때 유용하다.
- ✓ 설정에서 스레드 수를 늘리면 됨.

```
akka {  
  akka.loggers = ["akka.event.slf4j.Slf4jLogger"]  
  loglevel = WARNING  
  actor {  
    default-dispatcher = {  
      fork-join-executor {  
        parallelism-min = 300  
        parallelism-max = 300  
      }  
    }  
  }  
}
```

300개의 스레드 풀을 구성함. 톰캣의 경우 200개의 스레드 풀을 기본으로 가지고 있으므로 동기적인 코드를 처리하는데 충분함. 그러나 이러한 설정은 이벤트 모델에 비하여 전체 성능은 떨어짐. 또한 필요한 메모리의 양이 늘어남.



Actor

Actor 소개

- ✓ 액터 기반의 동시성 모델은 Erlang 프로그래밍 언어에서 대중화 되었음.(얼랭: 병렬프로그래밍 언어, 1986)
- ✓ 액터는 Akka 동시성 툴킷에 의해 JVM 상에서 구현됨.(<http://akka.io>)
- ✓ 액터는 탄력적이고 확장가능한 응용 프로그램을 구축하는데 매우 효과적인 도구임
- ✓ Akka는 Play에서 바로 사용 가능하다.

* Akka

JVM상의 동시성과 분산 애플리케이션을 구현하기 위한 런타임

Erlang의 영향으로 Actor기반의 동시성 구현

자바/ 스칼라 모두 가능

2010년 1월 부터 릴리즈

현재는 Play 프레임워크, 스칼라 언어와 함께 Lightbend 플랫폼의 일부를 구성함.

<http://akka.io/docs>

Kuhn and Allen's Reactive Design Patterns (Manning, 2016)

Akka in Action by Roestenburg, Bakker, and Williams (Manning, 2016)

Actor의 행위

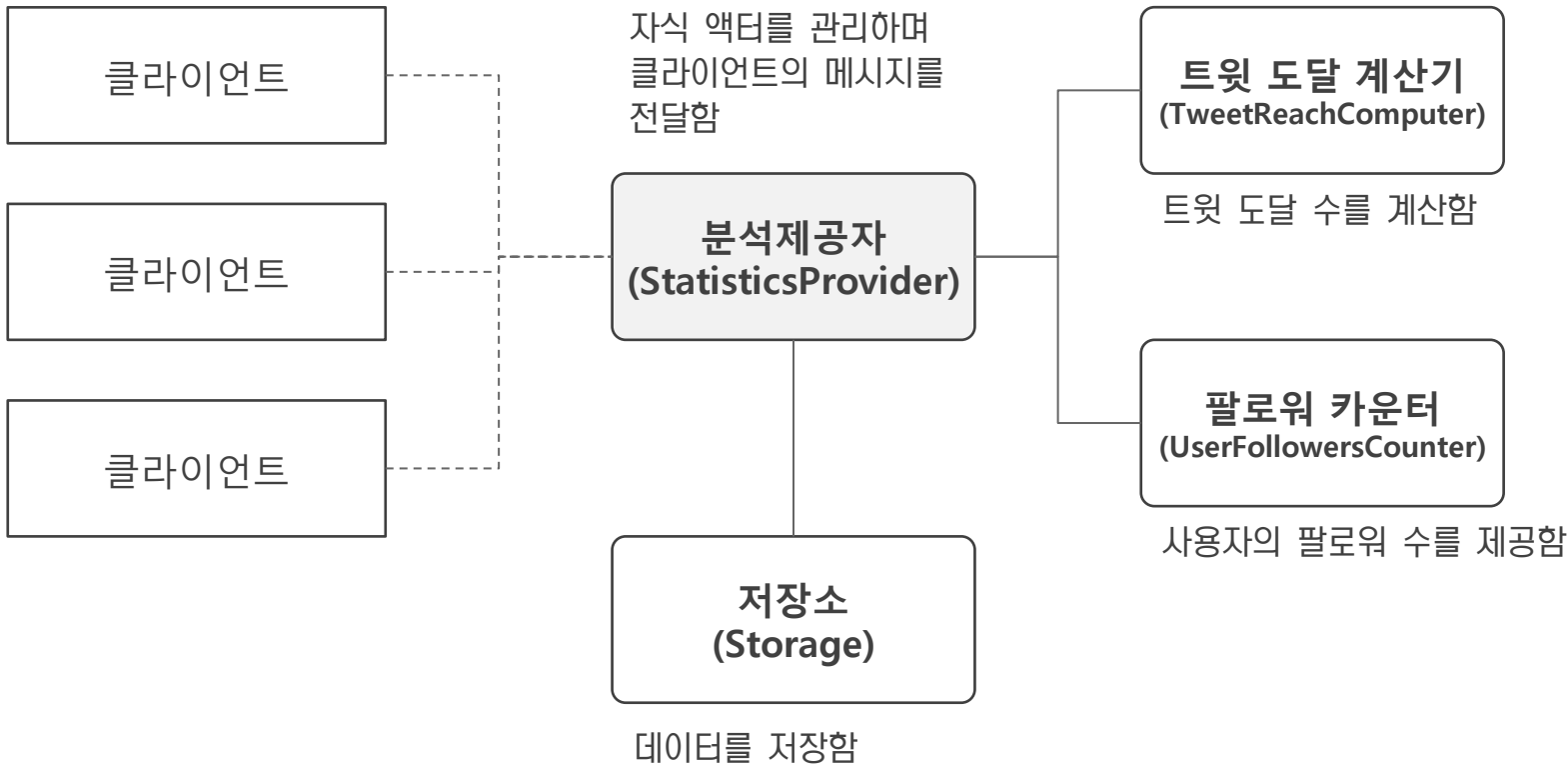
- ✓ 액터는 비동기 메시지 전달을 통해 서로 통신하며 메시지는 immutable 함.
- ✓ 액터는 공유를 위한 준비가되었는지 또는 언제 변경할지를 결정함으로써 스레드 모델의 고질적인 공유자원 문제를 해결한다.
- ✓ 액터는 Object-Oriented 개념을 올바르게 수행하면서도 비동기로 수행을 함.

* 액터의 행위

1. 메시지 송신 및 수신
2. 메시지에 대한 응답으로 어떤 행위를 하거나 상태를 변경함.
3. 새로운 자식 액터 시작

트위터 서비스 소개

- ✓ 트위터 활동에 대한 분석을 제공하는 서비스 구현
- ✓ 트윗의 리트윗 횟수와 도달 범위를 계산
- ✓ 각 액터는 하나의 책임을 가져야 함. 따라서 이 서비스는 4가지 종류의 액터가 식별되었음



프로젝트 생성 및 설정

✓ play 2.4 버전 사용시 activator로 프로젝트 생성 가능. 그 이후에는 sbt 사용 가능

콘솔

```
activator new twitter-service play-scala-2.4  
또는  
sbt new playframework/play-scala-seed.g8
```

build.sbt

```
...  
  
libraryDependencies += Seq(  
  ws,  
  "org.reactivemongo" %% "play2-reactivemongo" % "0.11.7.play24",  
  "com.typesafe.akka" %% "akka-actor" % "2.4.0",  
  "com.typesafe.akka" %% "akka-slf4j" % "2.4.0"  
)  
  
...  
  
libraryDependencies += "com.ning" % "async-http-client" % "1.9.29"
```


StatisticsProvider(분석제공자) 뼈대 구성

✓ Actor와 ActorLogging 트레이트를 구현함.

StatisticsProvider 액터

```
package actors
import akka.actor.{Actor, ActorLogging, Props}

class StatisticsProvider extends Actor with ActorLogging {
  def receive = {
    case message => // do nothing
  }
}

object StatisticsProvider {
  def props = Props[StatisticsProvider]
}
```

로깅 바인딩 설정

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "DEBUG"
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"
}
```

Actor의 기본개념

- ✓ 메시지는 받는 순서대로 처리되며 처리될때 까지 메일박스 대기열에 보관됨.
- ✓ Actor가 메시지를 처리하려면 실행을 위한 공간이 필요한데 이를 디스패처라 한다.
- ✓ 디스패처는 ExecutionContext 와 같음. 따라서 Actor내의 Future를 실행할 수 있음.
- ✓ 기본적으로 동일한 디스패처가 전체 ActorSystem에서 사용되며 fork-join excutor 에 의해 지원됨.

Actor와 스레드, 액터참조

- ✓ 액터는 스레드가 아님. 그러나 액터는 작업을 수행하기 위하여 스레드가 필요함.
- ✓ 액터는 공유된 스레드 풀에서 제공되는 스레드를 이용하여 작업을 수행하며 하는 일이 없는 경우 스레드를 점유하지 않음.
- ✓ 이러한 액터와 스레드의 분리 개념은 자원을 효율적으로 사용할 수 있도록 함.
- ✓ Actor와 Actor간의 통신이 필요한 경우 직접 참조하지 않고 ActorRef를 통하여 참조함.
- ✓ 이를 통해 얻는 잇점은 물리적으로 Actor의 위치가 변하거나 Actor가 재시작 되는 경우에도 항상 참조가 유효함.

Actor 생성

- ✓ Actors.scala는 ActorSystem으로 부터 Actor들을 생성하는 모듈임.
- ✓ 이 모듈을 등록하면 애플리케이션이 초기화 될 때 실행된다.
- ✓ 디펜던시 인젝션으로 ActorSystem이 인젝션 되며 이를 통해 액터를 생성할 수 있다.

```
package modules

import javax.inject._
import actors.StatisticsProvider
import akka.actor.ActorSystem
import com.google.inject.AbstractModule

class Actors @Inject() (system: ActorSystem) extends ApplicationActors {
  // create root actor
  system.actorOf(props = StatisticsProvider.props, name = "statisticsProvider")
}

trait ApplicationActors // marker trait

class ActorsModule extends AbstractModule {
  override def configure(): Unit = {
    bind(classOf[ApplicationActors]).to(classOf[Actors]).asEagerSingleton
  }
}
```

Actors.scala

Actor 생성

- ✓ 액터의 생성은 ActorSystem에 의해 만들어지거나 또다른 액터의 자식으로 만들어짐.
- ✓ 액터는 단독으로 존재할 수 없으며 항상 부모가 존재하고 계층 구조를 이루고 있다.
- ✓ 부모 액터는 자식 액터들을 감독하며 자식 액터의 실패를 다루는 방법을 알고 있다.
- ✓ 액터가 직접 인스턴스화 되지는 않으며 ActorSystem에 Props를 전달하는 방법으로 인스턴스 생성 지시를 할 수 있다.

액터모듈 등록

```
play.modules.enabled += "modules.ActorsModule"
```

application.conf

가급적이면 최상위 액터를 많이 만들지 말 것. 최상위 액터는 생성하는데 비용이 많이 들고 동기화가 필요한 부분이기 때문이다. ActorSystem 내의 액터들은 서로 통신이 가능하며 별도의 JVM에서 구동하는 액터도 서로 통신이 가능하도록 할 수 있다.

자식 Actor 생성

- ✓ StatisticsProvider의 context를 이용하여 자식 Actor 생성함.(UserFollowersCounter, Storage, TweetReachComputer)
- ✓ 여기서 생성된 액터들은 StatisticsProvider 액터의 자식 액터가 됨.
- ✓ context.child(childName), 또는 context.children 컬렉션에서 자식 액터를 찾을 수 있음.
- ✓ 자식 액터는 반드시 preStart() 메소드 내에서 만들어야만 부모 액터가 재시작 되었을 때 자식들도 같이 생성될 수 있음.

```
var reachComputer: ActorRef = _
var storage: ActorRef = _
var followersCounter: ActorRef = _

override def preStart(): Unit = {
  log.info("Starting StatisticsProvider.")

  followersCounter = context.actorOf(Props[UserFollowersCounter], name = "userFollowersCounter")
  storage = context.actorOf(Props[Storage], name = "storage")
  reachComputer = context.actorOf(TweetReachComputer.props(followersCounter, storage), name =
    "tweetReachComputer")
}
```

StatisticsProvider class

메시지 전달

- ✓ 액터는 자신의 목적에 부합하는 메시지만 처리하고 나머지 메시지는 무시한다.
- ✓ 액터 시스템을 구축할 때 가장 중요한 작업은 상호간의 메시지 프로토콜을 구축하는 것이다.
- ✓ 각 액터간의 상호작용을 분석하여 만든 메시지 유형은 아래와 같이 케이스 클래스로 정의할 수 있다.
- ✓ 각 액터 간을 넘나드는 메시지는 immutable 하여야 한다.

```
package messages

case class ComputeReach(tweetId: BigInt)
case class TweetReach(tweetId: BigInt, score: Int)

case class FetchFollowerCount(tweetId: BigInt, userId: BigInt)
case class FollowerCount(tweetId: BigInt, userId: BigInt, followersCount: Int)

case class StoreReach(tweetId: BigInt, score: Int)
case class ReachStored(tweetId: BigInt)
```

Messages.scala

액터 상호작용

- ✓ ComputeReach 메시지를 받으면 트위터에 접속하여 해당 트윗에 대한 리트윗을 가져옴.
- ✓ 리트윗에 따라 UserFollowersCounter에 리트윗 작성자가 몇 명의 팔로워가 있는지 물어봄.
- ✓ 모든 사용자에게 대해 해당 정보를 받으면 클라이언트에 응답하고 리트윗을 저장함.

```
class TweetReachComputer(
  userFollowersCounter: ActorRef, storage: ActorRef) extends Actor with ActorLogging {

  implicit val executionContext = context.dispatcher

  var followerCountsByRetweet = Map.empty[FetchedRetweet, List[FollowerCount]]

  def receive = {
    case ComputeReach(tweetId) =>
      fetchRetweets(tweetId, sender()).map { fetchedRetweets =>
        followerCountsByRetweet = followerCountsByRetweet + (fetchedRetweets -> List.empty)
        fetchedRetweets.retweeters.foreach { rt =>
          userFollowersCounter ! FetchFollowerCount(tweetId, rt)
        }
      }
    case count @ FollowerCount(tweetId, _, _) =>
      log.info("Received followers count for tweet {}", tweetId)
      fetchedRetweetsFor(tweetId).foreach { fetchedRetweets =>
        updateFollowersCount(tweetId, fetchedRetweets, count)
      }
    case ReachStored(tweetId) =>
      followerCountsByRetweet.keys.find(_.tweetId == tweetId)
        .foreach { key =>
          followerCountsByRetweet = followerCountsByRetweet.filterNot(_. _1 == key)
        }
  }
}
```

계속...

액터 상호작용

```
case class FetchedRetweet(tweetId: BigInt, retweeters: List[BigInt], client: ActorRef)

def fetchedRetweetsFor(tweetId: BigInt) = followerCountsByRetweet.keys.find(_.tweetId == tweetId)

def updateFollowersCount(tweetId: BigInt, fetchedRetweets: FetchedRetweet, count: FollowerCount) = {
  val existingCounts = followerCountsByRetweet(fetchedRetweets)
  followerCountsByRetweet = followerCountsByRetweet.updated(fetchedRetweets, count :: existingCounts)
  val newCounts = followerCountsByRetweet(fetchedRetweets)

  if (newCounts.length == fetchedRetweets.retweeters.length) {
    log.info("Received all retweeters followers count for tweet {}" + ", computing sum", tweetId)
    val score = newCounts.map(_.followersCount).sum

    fetchedRetweets.client ! TweetReach(tweetId, score)
    storage ! StoreReach(tweetId, score)
  }
}

def fetchRetweets(tweetId: BigInt, client: ActorRef): Future[FetchedRetweet] = ???
}
```

액터 상호작용 - mutable 값에 대한 경쟁

- ✓ 앞의 예제를 자세히 살펴보면 변경가능한 변수(followerCountsByRetweet)에 대해 경쟁상태임을 알 수 있다.
- ✓ ComputeReach 메시지가 도착하면 관련 트윗을 가져온 후 패치된 트윗에 해당하는 카운트 정보를 초기화 한다.
- ✓ 한편 FollowerCount 메시지가 도착하면 팔로워 수를 가져온 후 카운트 정보를 업데이트 한다.
- ✓ 여기서 fetchRetweets 메소드는 비동기로 동작하며 Future를 리턴한다. 따라서 fetchRetweets 가 완료되기 전에(카운트 정보가 초기화 되기 전에) FollowerCount 메시지 처리가 일어날 수 있다.

액터 상호작용 - 파이프 패턴 사용

- ✓ 파이프 패턴을 사용하면 Future의 결과가 완료되면 액터로 자동 전송할 수 있음.
- ✓ 앞의 예제에서 fetchRetweets 실행 결과가 Future이고 이 결과를 다시 자기 자신에게 파이프 함.
- ✓ fetchedRetweets 메시지를 받았을 때에는 이미 앞의 작업이 완료된 이후이므로 여기서 패치된 트윗에 해당하는 카운터를 초기화 시켜도 정보의 경쟁상태가 발생하지 않음.

```
import akka.pattern.pipe

def receive = {
  case ComputeReach(tweetId) =>
    fetchRetweets(tweetId, sender()) pipeTo self

  case fetchedRetweets: FetchedRetweet =>
    followerCountsByRetweet += (fetchedRetweets -> List.empty)
    fetchedRetweets.retweeters.foreach { rt =>
      userFollowersCounter ! FetchFollowerCount(fetchedRetweets.tweetId, rt)
    }
  ...
}
```

Future의 실패

- ✓ 파이프 패턴에는 다음 액터에게 Future를 보내지만 받는 쪽에서는 실제 결과를 받음.
- ✓ 그러나 Future가 실패할 경우 이에대한 처리 책임은 보내는 쪽에 있음.(상위 액터가 받아보긴 하나 효율적 대응을 할 수 없음.)
- ✓ 따라서 Future를 처리할 때에는 실패에 대한 처리도 함께 하는 것이 좋다.

```
case class RetweetFetchingFailed(tweetId: BigInt, cause: Throwable, client: ActorRef)

def receive = {
  case ComputeReach(tweetId) =>
    val originalSender = sender()
    fetchRetweets(tweetId, sender()).recover {
      case NonFatal(t) => RetweetFetchingFailed(tweetId, t, originalSender)
    } pipeTo self
  ...
}
```

Future, Actor 차이

- ✓ Future와 Actor 모두 비동기 컨텍스트에서 수행되는 점은 똑같다.
- ✓ 그러나 Future는 일회성이며 단일 결과를 계산하기 위한 목적으로 사용되며, Actor는 고급 프로세스 수행에 사용된다
- ✓ Future는 완료되면 끝나지만, Actor는 완료되면 다음 테스트 수행까지 대기한다.
- ✓ Actor는 상태를 외부에 노출시키지 않는 조건으로 상태를 유지할 수 있다.

오류 처리에 대한 패러다임 전환

- ✓ 지금까지 시스템에 대한 생각은 하나의 조직화 되고 물샐틈 없이 잘 짜여진 시스템을 구성하는 것이었다.(시스템은 오류 없이 완벽해야 한다.)
- ✓ 그러나 시스템이 거대해 지고 잘 짜여 질 수록 변화에 대해 매우 민감하게 반응한다.(일부 코드의 수정이 예측할 수 없는 전혀 다른 부분의 문제를 야기한다.)
- ✓ 시스템에서 실패할 수 있는 모든 방법을 예측하는 것은 불가능에 가까움.
- ✓ Actor 기반의 디자인 아이디어는 각 구성요소는 충돌할 수 밖에 없으며 시스템 스스로 치료하는 방법을 알게 된다는 점임.
- ✓ Actor 시스템처럼 실패를 피하는 데 집중하는 대신 대신 실패로 부터 가장 효과적인 방법으로 회복하려고 노력해야 함.
- ✓ Akka 커뮤니티의 블로그 제목이 "Let it crash"임 (<http://letitcrash.com>)

저장소 액터

- ✓ 저장소는 MongoDB와의 연결을 위해 ReactiveMongo 의존성을 사용함.
- ✓ 액터 내에서 외부로 노출시키지 않는 한 내부에서 mutable 변수를 사용 가능함. 그러나 Future와 더불어 경쟁상태에 두지 않도록 주의하여야 함.
- ✓ <http://reactive-mongo.org>

```
class Storage extends Actor with ActorLogging {  
  
  val Database = "twitterService"  
  val ReachCollection = "ComputedReach"  
  
  implicit val executionContext = context.dispatcher  
  
  val driver: MongoDriver = new MongoDriver  
  var connection: MongoConnection = _  
  var db: DefaultDB = _  
  var collection: BSONCollection = _  
  obtainConnection()  
  
  // overrides the postRestart handler to reinitialize the connection after restart if necessary  
  override def postRestart(reason: Throwable): Unit = {  
    reason match {  
      case ce: ConnectionException =>  
        // try to obtain a brand new connection  
        obtainConnection()  
    }  
    super.postRestart(reason)  
  }  
}
```

저장소 액터

```
// Tears down connection and driver instances when the actor is stoped.
override def postStop(): Unit = {
  connection.close()
  driver.close()
}

var currentWrites = Set.empty[BigInt]

def receive = {
  case StoreReach(tweetId, score) =>
}

private def obtainConnection(): Unit = {
  connection = driver.connection(List("localhost"))
  db = connection.db(Database)
  collection = db.collection[BSONCollection](ReachCollection)
}

case class StoredReach(when: DateTime, tweetId: BigInt, score: Int)
```


실패에 대한 대응 전략

- ✓ 부모액터는 자식 액터의 실패에 대해서 대응하는 전략을 정 할 수 있다.(OneForAll, OneForOne)
- ✓ StatisticsProvider의 자식 액터는 대부분 stateless 로 동작하므로 OneForOne Supervision 전략을 선택하였다.
- ✓ 아래 코드의 OneForOne 전략은 액터 종료 전에 2분 내에 3번까지 재시도한다.
- ✓ ConnectionException인 경우 재시작하고 그 이외의 경우 디폴트 전략을 사용하며 처리 못하면 전략을 다시 상위 액터로 확대한다.

```
override def supervisorStrategy: SupervisorStrategy =  
  
  OneForOneStrategy(maxNrOfRetries = 3, withinTimeRange = 2.minutes) {  
    case _: ConnectionException => Restart  
    case t: Throwable => super.supervisorStrategy.decider.applyOrElse(t, (_: Any) => Escalate)  
  }
```

StatisticsProvider.scala

OneForAll 전략은 문제가 발생한 하위 액터를 기준으로 형제 액터를 모두 재시작하는 것을 의미하고
OneForOne 전략은 해당 하위 액터만 재시작한다는 의미이다.

액터의 감시

- ✓ 부모는 자식 액터의 상태를 감지하기 위해 일종의 모니터링 도구를 사용한다.
- ✓ watch를 사용하면 자식 액터는 자신의 상태를 부모에게 통지한다.
- ✓ 만일 Storage 액터가 종료되면 Terminated 메시지를 보낸다.
- ✓ Terminated 메시지로 Storage가 중단되었다는 메시지를 받으면 서비스를 유지하기 어려우므로 모드를 변경한다.

```
class StatisticsProvider extends Actor with ActorLogging {  
  ...  
  override def preStart(): Unit = {  
    storage = context.actorOf(Props[Storage], name = "storage")  
    context.watch(storage)  
  }  
  
  def receive = {  
    case reach: ComputeReach =>  
      reachComputer forward reach  
    case Terminated(terminatedStorageRef) =>  
      context.system.scheduler.scheduleOnce(1.minute, self, ReviveStorage)  
      context.become(storageUnavailable) // switch mode  
  }  
  
  def storageUnavailable: Receive = {  
    case ComputeReach(_) =>  
      sender() ! ServiceUnavailable  
    case ReviveStorage =>  
      storage = context.actorOf(Props[Storage], name = "storage")  
      context.unbecome() // switch back to the original  
  }  
}
```

```
object StatisticsProvider {  
  case object ServiceUnavailable  
  case object ReviveStorage  
}
```

보다 정교한 회복 전략

- ✓ 앞의 예제와 같이 `context.watch()` 도구로 정교한 회복 프로세스를 구현할 수도 있음.
- ✓ 하위 액터가 스스로 회복하도록 일단 기다린 후 그래도 회복할 수 없으면 하위 액터를 다시 시작할 수 있음.
- ✓ 보다 정교하게 하자면 해당 데이터베이스 접속 문제라면 다른 접속 URL로 변경하여 시도하거나
- ✓ 아니면 로컬 DB로 전환하였다가 회복되면 원래 DB로 전환하는 방법도 생각해 볼 수 있다.

감사합니다...

- ❖ 넥스트리컨설팅(주)
- ❖ 한성영 (syhan@nextree.co.kr)
- ❖ www.nextree.co.kr