

# セプターニ・オリジナル 技術読本

## 目次

[はじめに](#)

[セプターニ・オリジナル社とは？](#)

[PYXIS事業でのデータパーティション設計事例](#)

[IntelliJのショートカット＋便利機能集](#)

[Incoming WebHooksを使ってScalaでSlackに投稿してみる](#)

[Beanstalk for Dockerを使ったPlay Framework実行環境構築](#)

[Python学生からプロScalaエンジニアに](#)

[GANMA!でのCache実装例](#)

[Play Framework入門者向けプロジェクト分割](#)

[ヘキサゴナルアーキテクチャを採用したプロジェクトの実装例](#)

[Akka Schedulerの定期実行を使って自動的にInstagramから画像を取得し、Tumblrへ投稿](#)

[scala DDD 依存性逆転の原則を採用してみた](#)

[今すぐ使える！Android開発に役立つKotlinの拡張関数10選](#)

[おわりに](#)

## はじめに

こんにちは！

この本は株式会社セプテーニ・オリジナル所属のエンジニア有志で作った技術同人誌です。

Scalaネタを中心としつつ、様々な技術ネタを一冊に詰め込んだごった煮本です。

どうぞ一部だけでも良いので読んでみて戴けると幸いです。

## セプターニ・オリジナルとは？

こんにちは、株式会社セプターニ・オリジナルの杉谷と申します。前職は株式会社ドワンゴで、ニコニコ生放送の初代リーダーを務めていました。現在はGANMA!というオリジナルマンガ配信サービスの開発を担当しています。

"セプターニ"という会社をご存じでしょうか？

おそらく普通のエンジニアの方にはあまり知られていないとおもいますが、実はネット広告業界ではトップクラスで、その界隈では名のしれた会社だったりいたします。

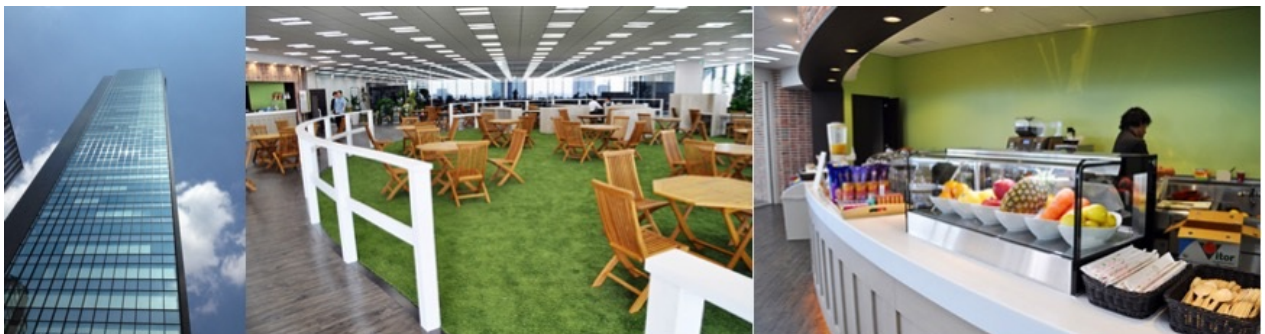
本章では我々と、我々が所属するセプターニグループについてご紹介させていただきます。

## セプターニグループとセプターニ・オリジナル

セプターニグループはネット広告会社である株式会社セプターニをはじめとして、様々な事業をおこなっているグループです。

ネット広告では、インターネットで効果的にプロモーションを行いたいお客様に代わってクリエイティブの制作や、広告の運用ノウハウを活かして、広告効果の最大化をはかっています。

2015年度の売上は645億円で、オフィスがユニークだったり社内にカフェがあったりします。



広告以外にもマンガ配信サービスである"GANMA!"やファッションコーディネートアプリ"MANT"など、広告以外の事業も手広くやっています。

我々セプターニ・オリジナルはグループの"開発部"のようなもので 広告に関するシステムや、先述のGANMA!など、グループに関わる様々なシステム開発を担当しています。

## システム紹介

主な担当システムの概要とアーキテクチャを紹介します。

### PYXIS

FacebookやTwitterなどに対する広告運用をサポートするための支援ツールです。

- Playframework + Scala / AngularJS + TypeScript
- DDD採用（基本レイヤードアーキテクチャ、一部ヘキサゴナルアーキテクチャ）

最近はYahoo!JAPANにも対応し、日々の運用効率化に貢献しています。

## GANMA!

<http://ganma.jp>

オリジナルのマンガを配信するサービスです。セプテーニグループの代表である佐藤が自ら編集長となり、作家さんと共に作品作りをしています。（体裁ではなく、本当に編集業務をやっている）

- Playframework + Scala / AngularJS + TypeScript / iOS - Swift / Android - Scala
- DDD採用（レイヤードアーキテクチャ）

アプリは200万DLを突破し、単行本も2015/12月現在で12作品が単行本化され、アニメ化された作品もあります。超絶魔球が飛び交う超絶野球漫画「デッド・オア・ストライク」はTwitterで話題になりました。



作品抜粋：猫はまたたび／デッド・オア・ストライク／武蔵くんと村山さんは付き合ってみた／ミリオンドール／サークルクラッシュ／乙女哲学／リセットゲーム／ブレイブフロンティア ハルトの召喚日記／ごー・れむ／リヒト／…

## MANT

<http://mant.jp>

10代の女性に人気のファッションコーディネートアプリです。

- LAMP / Android - Kotlin,Java併用 / iOS - Swift,Objective-C併用
- ユニットテストだいぶ整備

ファッションライフをサポートする様々なオリジナル記事も配信しており、女子中高生から絶大な支持を集める人気モデルの"ちいばぽ"さんが編集長を務めています。

## 特徴と文化

私たちは現在、ほとんどのシステムでScalaを採用し、ドメイン駆動設計を活用し、スクラムを適切に回し、レビューを行い、ユニットテストを整備しCIを回す、など 正しく動き、手も入れやすいシステムを素早く届け続ける努力をしています。

また、私たちはネット広告業界大手であるセプテーニグループの一員ですので 単一の事業のみを手がけている企業ではありませんし、巨大なプロダクトを保持する大企業でもない、という特殊な立ち位置にあります。

プロジェクトの幅が広く、資金にも余裕がある、という特徴がある一方、保守しなくてはならない資産が少なく、変化に柔軟で新しいことにも挑戦できます。

整った環境で、様々なプロジェクトに、エンジニアとして有るべき姿勢でシステム作りに取り組めるとするのは、なかなか希有な職場ではないでしょうか。

## 終わりに

様々なサービスを手掛ける中で多くの失敗を重ね、改善を繰り返しながら今に至っています。今後もより良い職場にしていけたらな、と思っています。

もし本書を読んで、我々に興味を持って戴けたとしたら幸いです。

読んでくださいますとありがとうございます。《杉谷保幸》

# PYXIS事業でのデータパーティション設計事例

## はじめに

こんにちは、@kimutyam(※1)です。  
私はPYXIS(※2)事業の開発リーダーを勤めております。PYXISでは大量の広告データを扱うプロダクトで、その時のデータの負荷分散設計について述べます。

## シャーディングとパーティションについての考察

シャーディングとは、リソース(主にデータ)をshardに分解して、複数のサーバに分散させる手法のことを指します。シャーディングはデータベース固有の手法ではないので、一概にOracleやMySQLが提供しているパーティション機能のことを指しているわけではないです。

シャーディングの設計に関しては、アプリケーションレイヤで解決するのか、ストレージに委譲するのかは様々な方法が存在します。Gizzard(※3)ではシャーディングは多くの場合パーティションとレプリケーションの2つの技術を用いると記述されています。

Sharding strategies often involve two techniques: partitioning and replication.

また、パーティションはWikipedia(※4)では次のように説明されています。

コンピュータのハードディスクのパーティション (Partition) とは、ハードディスクの記憶領域を論理的に分割すること、あるいは分割された個々の領域を指す

今回はレプリケーションについては取り扱いませんので、パーティションの話がメインテーマとなります。

## ユースケースの説明

今回設計をしたユースケースは次のようになります。

- 1日あたり約35万レコードがDBに保存される
- 対象のデータベースのテーブルには日付のフィールドが存在する
- 1年後のデータは不要
- 最大30日分のレコードを取得するクエリを発行する。
- 月内で完結するクエリが多い

## 技術

- MySQL 5.6
- InnoDB ストレージエンジン

以下のようなDDLを想定をします。



```
CREATE TABLE `Sample` (  
  `sampleId` BIGINT NOT NULL COMMENT 'サンプルID',  
  `date` DATE NOT NULL COMMENT '日付',  
  `name` VARCHAR(255) NOT NULL COMMENT '名前',  
  `hogeId` BIGINT NOT NULL COMMENT 'HogeID',  
  PRIMARY KEY (`sampleId`, `date`),  
  CONSTRAINT `sample_hogeId` FOREIGN KEY (`hogeId`) REFERENCES `Hoge` (`hogeId`)  
  ON DELETE CASCADE ON UPDATE CASCADE  
) ENGINE = InnoDB DEFAULT CHARACTER SET = utf8mb4 COMMENT = 'サンプル';
```

## パーティション設計

- 分散系の処理をアプリケーションで実装する
  - メリット
    - 特定のストレージやサーバー、ライブラリに依存しない
  - デメリット
    - 保守、運用が大変
    - ストレージが提供しているパーティション機能のようなものをアプリケーションだけで補おうとすると開発が大掛かりになりがち
- MySQLパーティションに任せる
  - メリット
    - パーティションプルーニング(※5)によるクエリー最適化
    - 不要なデータの削除が高速
  - デメリット
    - 特定のストレージに依存する
    - InnoDB外部キーが利用できない(※6)

InnoDB 外部キーと MySQL パーティショニングは互換性を持ちません。パーティション化された InnoDB テーブルは、外部キー参照を持つことができず、外部キーによって参照されるカラムを持つこともできません。外部キーを持っていたり、外部キーによって参照されたりする InnoDB テーブルはパーティション化できません。
    - 動的にパーティションが追加、削除する機構なし

[結論]今回はMySQLパーティションを採用しました。

[理由]

- 削除が高速で他のパーティションに干渉しない
- パーティションプルーニングが強力であり、アプリケーションで同じようなものを実装するのは大変
- 外部キーは外す(ストレージでデータ整合性を担保せずにアプリケーションでカバーした。既にしていた(※7))
- 動的にパーティションを追加、削除する機構を作成した(自動化)
- 要件的にRDBMS以外で持つことは今後考えづらく、MySQL以外の選択もしばらくないためストレージ依存にしてよいという判断になった



また、Gizzardも検討しましたが、使い方を理解するのに時間がかかりそうだったので今回は見送りました。

## MySQLのパーティション設計

- パーティションタイプの選択

RANGEパーティション(※8)を選択し、Sample.dateのRANGEで月毎にパーティションでを切ります。1年経ったパーティションは削除する。

- パーティションの最大数

パーティショニングの制約と制限(※9)

パーティションの最大数MySQL5.6.7より前は、NDB ストレージエンジンを使用しないテーブルで可能な最大パーティション数は1024でした。MySQL 5.6.7以降は、この制限は8192パーティションに増えています。MySQL Server バージョンにかかわらず、この最大数にはサブパーティションが含まれます。

パーティションは1年の月分しか発行しないので、月毎にパーティションを切って、1年後にパーティションを消す

- パーティションの定義と追加

パーティションの大枠となる定義を追加し、バッチで追加・削除を行う方針をとります。

```
CREATE TABLE `Sample` (  
  `sampleId` BIGINT NOT NULL COMMENT 'サンプルID',  
  `date` DATE NOT NULL COMMENT '日付',  
  `name` VARCHAR(255) NOT NULL COMMENT '名前',  
  `hogId` BIGINT NOT NULL COMMENT 'HogeID',  
  PRIMARY KEY (`sampleId`, `date`),  
) ENGINE = InnoDB DEFAULT CHARACTER SET = utf8mb4 COMMENT = 'サンプル'  
PARTITION BY RANGE (`date`)  
(  
  PARTITION p20151001 VALUES LESS THAN ('2015-11-01')  
  COMMENT = '~ 2015/10/31' ENGINE = InnoDB  
)
```

具体的な日時パーティションでDDLを汚したく気持ちはあります。しかし、MySQLではパーティションを定義しないと追加文が実行できません。

以下のように自動化バッチで1年分のパーティション定義するSQL実行を実行してやれば、ここに定義を書く必要がなくなります。

```
ALTER TABLE AdReport PARTITION BY RANGE (`date`) (
PARTITION p20151001 VALUES LESS THAN ('2015-10-01') ENGINE =InnoDB,
PARTITION p20151101 VALUES LESS THAN ('2015-11-01') ENGINE =InnoDB,
PARTITION p20151201 VALUES LESS THAN ('2015-12-01') ENGINE =InnoDB,
PARTITION p20160101 VALUES LESS THAN ('2016-01-01') ENGINE =InnoDB,
PARTITION p20160201 VALUES LESS THAN ('2016-02-01') ENGINE = InnoDB
....
);
```

しかし、パーティションを定義するALTER TABLE文はテーブルのデータ量が多くなると実行速度が遅くなります。これはALTER TABLEによって既存データがコピーされるためです。

一方、RANGEパーティションにおいてALTER TABLE ADD PARTITIONはデータ量に依存せずに高速に処理が完了します。

```
ALTER TABLE AdReport ADD PARTITION (
PARTITION p20151202 VALUES LESS THAN ('2020-12-02'));
```

理由は以下。

パーティション化された InnoDB テーブルに対するオンライン DDL(※10)

RANGE または LIST によってパーティション化されたテーブルに対する ADD PARTITION および DROP PARTITION では、どの既存のデータもコピーされません。

また、以下のようにMAXVALUEに設定したパーティション定義をし直すアプローチもあります。

```
CREATE TABLE `Sample` (
`sampleId` BIGINT NOT NULL COMMENT 'サンプルID',
`date` DATE NOT NULL COMMENT '日付',
`name` VARCHAR(255) NOT NULL COMMENT '名前',
`hogelId` BIGINT NOT NULL COMMENT 'HogeID',
PRIMARY KEY (`sampleId`, `date`),
) ENGINE = InnoDB DEFAULT CHARACTER SET = utf8mb4 COMMENT = 'サンプル'
PARTITION BY RANGE (`date`)
(
PARTITION p20151001 VALUES LESS THAN ('2015-11-01')
COMMENT = '~ 2015/10/31' ENGINE = InnoDB,
PARTITION pmax VALUES LESS THAN MAXVALUE ENGINE=InnoDB
)

ALTER TABLE AdReport REORGANIZE PARTITION pmax INTO
(
PARTITION p20151101 VALUES LESS THAN ('2015-12-01')
COMMENT = '~ 2015/11/30' ENGINE=InnoDB,
PARTITION pmax VALUES LESS THAN MAXVALUE ENGINE=InnoDB
)
```

しかし、これも既存データをコピーするのでALTER TABLEなので実装速度が遅いです。

以上の理由から、パーティションを追加する方針にしました。

[方針まとめ]

- RANGEパーティションを選択
- 月毎にパーティションを追加
- 年毎にパーティションを(自動バッチによって)削除
- 未来のパーティションは自動バッチによって追加

## パーティション取得・追加・削除のサンプルコード

せっかくなので、Scalaで。

ScalikeJDBC(※11)とJoda-Time(※12)を使っています。

### RANGEパーティションVO

```
package db.partition

import org.joda.time.LocalDate
import org.joda.time.format.ISODateTimeFormat

trait PartitionRange {
  val partitionName: PartitionRangeName
  val value: Any
}

private[partition] case class PartitionRangeImpl(
  partitionName: PartitionRangeName,
  value: Any) extends PartitionRange

object PartitionRange {

  /**
   * Date型で1ヶ月単位でパーティションする時に利用するファクトリ
   *
   * 引数の整合性はjodaTime側で担保されている
   */
  def ofDateRangePerMonth(year: Int, month: Int): PartitionRange = {
    val date = new LocalDate(year, month, PartitionRangeConstant.BeginningOfMonthDay)

    ofDateRangePerMonth(date)
  }

  /**
   * Date型で1ヶ月単位でパーティションする時に利用するファクトリ
   */
  def ofDateRangePerMonth(date: LocalDate): PartitionRange = {

    // 月始まりでなければならない
    require(date.getDayOfMonth == PartitionRangeConstant.BeginningOfMonthDay)

    PartitionRangeImpl(
      PartitionRangeName.ofDateRangePerMonth(date),
      date.plusMonths(1).toString(ISODateTimeFormat.date)
    )
  }
}
```

## RANGEパーティションの名前VO

```
package db.partition

import org.joda.time.LocalDate
import org.joda.time.format.ISODateTimeFormat

trait PartitionRangeName {
  val value: String
}

private[partition] case class PartitionRangeColumnsNameImpl(value: String)
  extends PartitionRangeName

object PartitionRangeName {
  private val partitionNamePrefix = "p"

  def apply(value: String): PartitionRangeName = {
    PartitionRangeColumnsNameImpl(value)
  }

  private[partition] def ofDateRangePerMonth(date: LocalDate): PartitionRangeName = {
    //月始まりでなければならない
    require(date.getDayOfMonth == PartitionRangeConstant.BeginningOfMonthDay)

    PartitionRangeColumnsNameImpl(
      partitionNamePrefix + date.toString(ISODateTimeFormat.basicDate)
    )
  }
}
```

## パーティションクエリを実行するプロバイダー

```
package db.partition

import scalikejdbc._

trait PartitionProviderOnMySQL
  extends FindFeaturePartitionOnMySQL
  with AddFeaturePartitionOnMySQL
  with DropFeaturePartitionOnMySQL

/**
 * パーティション名を取得
 * パーティション名が存在する場合に追加をしたり、
 * パーティション名が存在しない場合に削除するとエラーになるためtraitを提供している
 */
trait FindFeaturePartitionOnMySQL extends BasicPartitionHelper {
```

```

def findPartitionNameBy(
  partitionRangeName: PartitionRangeName
)(implicit s: DBSession): Option[PartitionRangeName] = {
  SQL(
    s"""
    |SELECT PARTITION_NAME
    |FROM INFORMATION_SCHEMA.PARTITIONS
    |WHERE TABLE_NAME = '$tableName' AND PARTITION_NAME = '${partitionRangeName.value}';
    """).stripMargin
  ).map { resultSet =>
    PartitionRangeName(resultSet.string("PARTITION_NAME"))
  }.single().apply()
}

trait AddFeaturePartitionOnMySQL extends BasicPartitionHelper {

  def addRangeColumnsPartition(range: PartitionRange)(implicit s: DBSession): Int = {
    SQL(
      s"""
      |ALTER TABLE ${toBackQuoteString(tableName)} ADD PARTITION (
      | PARTITION ${range.partitionName.value} VALUES LESS THAN (?)
      |)
      """).stripMargin
    ).bind(
      range.value
    ).update().apply()
  }
}

trait DropFeaturePartitionOnMySQL extends BasicPartitionHelper {

  def dropRangeColumnsPartition(
    partitionRangeName: PartitionRangeName
  )(implicit s: DBSession): Int = {
    dropPartition(partitionRangeName.value)
  }

  def dropPartition(partitionName: String)(implicit s: DBSession): Int = {
    SQL(
      s"""
      |ALTER TABLE ${toBackQuoteString(tableName)} DROP PARTITION $partitionName
      """).stripMargin
    ).update().apply()
  }
}

trait BasicPartitionHelper {
  protected def tableName: String

  protected def toBackQuoteString(self: String): String = s"$self"

```

```
}
```

### パーティション関連で仕様する定数

```
package db.partition

object PartitionRangeConstant {
  /**
   * 月初の日にち
   */
  val BeginningOfMonthDay = 1
}
```

### サンプルテーブルのパーティションプロバイダー

```
package db.partition.sample

import db.partition.PartitionProviderOnMySQL

class SamplePartitionProvider extends PartitionProviderOnMySQL {
  protected def tableName = "Sample"
}
```



## 具体的なクライアント実装イメージ

### 追加バッチ

```
package batch

import org.joda.time.LocalDate
import db.partition.PartitionRange
import db.partition.sample.SamplePartitionProvider
import scalikejdbc.{ AutoSession, DBSession }

object SamplePartitionAddBatch extends App {
  implicit val s: DBSession = AutoSession

  val nowDate = LocalDate.now
  val partitionRange = PartitionRange.ofDateRangePerMonth(
    nowDate.getYear,
    nowDate.getMonthOfYear
  )

  val provider = new SamplePartitionProvider

  if (provider.findPartitionNameBy(partitionRange.partitionName).isEmpty) {
    provider.addRangeColumnsPartition(partitionRange)
  }
}
```

## 削除バッチ

```
package batch

import org.joda.time.LocalDate
import db.partition.PartitionRange
import db.partition.sample.SamplePartitionProvider
import scalikejdbc.{ AutoSession, DBSession }

object SamplePartitionDeleteBatch extends App {
  implicit val s: DBSession = AutoSession

  val targetDate = LocalDate.now.minusYears(1)

  val partitionRange = PartitionRange.ofDateRangePerMonth(
    targetDate.getYear,
    targetDate.getMonthOfYear
  )

  val provider = new SamplePartitionProvider

  provider.findPartitionNameBy(partitionRange.partitionName).map { _ =>
    provider.dropRangeColumnsPartition(partitionRange.partitionName)
  }
}
```

## 終わりに

ご読了ありがとうございました。

※1 著者のTwitterアカウントです。 <https://twitter.com/kimutyam>

※2 PYXISの紹介サイトです。 <http://pyxis-social.com/> この他にもPYXIS事業では様々なメディアのツールを提供しています。

※3 Twitter社のシャーディングフレームワーク <https://github.com/twitter/gizzard>

※4

<https://ja.wikipedia.org/wiki/%E3%83%91%E3%83%BC%E3%83%86%E3%82%A3%E3%82%B7%E3%83%A7%E3%83%B3>

※5 <https://dev.mysql.com/doc/refman/5.6/ja/partitioning-pruning.html>

※6 <https://dev.mysql.com/doc/refman/5.6/ja/partitioning-limitations-storage-engines.html>

※7 「我々(主語が大きい)は何故MySQLで外部キーを使わないのか」を参考にしました。

<https://songmu.github.io/slides/fk-night/#0>

※8 <https://dev.mysql.com/doc/refman/5.6/ja/partitioning-range.html>

※9 <https://dev.mysql.com/doc/refman/5.6/ja/partitioning-limitations.html>

※10 <https://dev.mysql.com/doc/refman/5.6/ja/online-ddl-partitioning.html>

※11 <http://scalikejdbc.org/>

※12 <http://www.joda.org/joda-time/>

# IntelliJのショートカット＋便利機能集

## はじめに

Scalaを書く時、多くの人がIntelliJ IDEAを使っているのではないのでしょうか？ 型チェックや定義ジャンプをする上でIDEの機能は欠かせないと思います。その点、IntelliJは開発の補助をしてくれる豊富な機能があり、デフォルトでも多くのショートカットが設定されています。しかし、IntelliJの機能がいくら豊富でも、把握していたり活用できてる機能はあまり多くないのではないのでしょうか。私もIntelliJを使い始めてもうすぐ1年になるのですが、未だに新しい機能を発見したりしてIntelliJの機能を使いこなせていないなと感じます。そこで、本項ではIntelliJの便利なショートカットとGitをはじめとする便利機能をまとめてみました。個人的に使用頻度の高いものから載せています。Scala開発者の助けになれば幸いです。

サンプルで使用している言語：Scala

対象環境OS：Mac

IntelliJのバージョン：14.1.5

## ショートカットキー

使用頻度	機能	ショートカットキー	説明	備考
高	前に開いていたファイルに戻る	Command + Alt + ←	前に開いていたファイルのカーソルのあった行に戻ることができます。	
高	マルチカーソル	Ctrl + G	現在カーソルのある箇所の文字列と同じ文字列を選択状態(マルチカーソル)にしてくれます。そのマルチカーソルになっている状態で修正することで複数行を同時に書き換えることができます。	
高	選択範囲をコメントアウト	Command + /	コメントアウトしたい箇所を選択した状態でこのショートカットを使うことでその使用言語に応じた形で一括コメントアウトしてくれます。	
高	エラーや警告の出ている箇所でクイック・フィックスする	Alt + Enter	タイプミスや型エラーによってIntelliJ上で黄色や赤の波線が入ってるところで、このショートカットを使うことで解決策のメニューを表示してくれます。また、そのメニューを選択することで容易に問題箇所の修正ができます。	警告の表示されてる文字列上でショートカットキーをタイプすることで以下のようなメニューが表示されます。 
				以下のようなウィンドウが表示されます。

高	コピー履歴から貼り付け	Command + Shift + V	過去のコピー履歴（5件分）からペーストしたい文字列を選択できます。※ただし、履歴が残っているのはIntelliJ上でコピーしたもののみ。	
高	選択行を上下に移動	Alt + Shift + ↑ / ↓	現在選択している行をそのまま上下に移動することができます。	
高	ファイル横断テキスト検索	Command + Shift + F	ファイルを横断したテキスト検索ができます。	<p>以下のようなウィンドウが表示されます。</p> 
高	なんでも検索	Shift 2回連打	クラス名やメソッド名、ファイル名などからインクリメント検索できます。	<p>以下のようなウィンドウが表示され、即時検索できます。</p> 
高	タブを閉じる	Command + F4	現在開いているタブを閉じることができます。	
高	宣言ヘジャンプ	Command + B	カーソルを置いてる変数やクラスの宣言部分に移動できます。	
高	現在の行を複製	Command + D	現在カーソルのある行を複製することができます。	
高	現在の行を削除	Command + Y	現在カーソルのある行を削除することができます。	
中	コードをフォーマットする	Command + Alt + L	各言語のフォーマットに合わせて現在開いているファイルの簡易的なコード整形を実行できます。	
中	最近開いたファイルを表示	Command + E	直近で触っていたファイルを開いていた順に表示 & オープンできます。	<p>以下のようなウィンドウが表示されます。</p> 
			カーソルを置いてるクラスやメソッドの実装箇所	

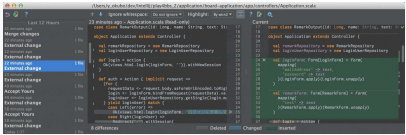
中	実装ヘジャンプ	Command + Alt + B	所に移動できます。もし、実装が複数ある場合は、実装すべてが表示された選択ウィンドウが表示されます。	
中	型の継承階層を表示する	Ctrl + H	カーソルを置いているクラスを継承しているクラスや逆に継承されているクラスの階層形式で表示できます。	IntelliJの右に以下のようなHierarchyメニューが表示されます。 
中	呼び出し先を調べる	Alt + Shift + H	現在カーソルのある関数を呼び出して使っている箇所を階層表示してくれます。	IntelliJの右に以下のようなHierarchyメニューが表示されます。 
中	選択状態ブロックの拡大／縮小	Command + W / Command + Shift + W	現在カーソルのある箇所の単語を選択状態にできます。さらにこのショートカットを使うことで、括弧やダブルクォーテーションを基準に選択の範囲を拡大／縮小することが可能です。	
中	行番号を指定してジャンプ	Command + G	行番号を指定して対象行にジャンプできます。また、コロン(:)で区切ることで行だけでなく文字数も指定できます。	以下のようなウィンドウが表示されます。 
低	選択範囲をブロックコメントアウト	Command + Shift + /	コメントアウトしたい箇所を選択した状態でこのショートカットを使うことでその使用言語に応じた形で一括コメントアウトしてくれます。一行一行コメントアウトするのではなく、一つの大きなブロックコメントにしてくれます。	
低	選択行を他の行と入れ替える	Command + Shift + ↑ / ↓	現在選択している行をインデントやコードブロックを考慮して上下に移動することができます。	
低	プロジェクトツールウィンドウの表示/非表示	Command + 1	プロジェクトツールウィンドウの表示・非表示を切り替えることができます。	
低	ストラクチャーツールウィンドウの表示/非表示	Command + 7	ストラクチャーツールウィンドウの表示・非表示を切り替えることができます。	
低	変数抽出	Command +	変数化したい式や関数を選択してからこのショートカットを使うことで適切な変数を自動生成	複雑な式の最終的な型を知りたいときに使ったり

			することができます。	もしています。
低	選択箇所をif、while、forなどで囲む	Command + Alt + T	囲みたい箇所を選択した状態でこのショートカットを使うことでメニューが表示され、そこから選択することで対象箇所を囲んだfor式やif式を自動生成できます。	<p>以下のようなメニューが表示されます。</p> 
低	新しい行を追加	Shift + Enter	現在カーソルのある行の下に改行を追加し、そこにカーソルを移動してくれます。	
低	選択箇所のインラインを展開する	Command + Alt + N	カーソルを置いている変数上でこのショートカットを実行することで、変数の中身をその場に展開してくれます。	[変数抽出]の逆バージョンです。
低	変数が使われている箇所へジャンプ	Command + Alt + F7	カーソルを置いている変数でこのショートカットを実行することで、その変数が使われている箇所の一覧表示&ジャンプができます。	<p>以下のようなメニューが表示されます。</p> 
低	メソッドに切り出す	Command + Alt + M	メソッド化したい式を選択している状態でこのショートカットを実行することでメソッドに切り出すことができます。実行後、メニューが表示され作成するメソッドのスキップの指定やメソッド名を設定できます。	
低	フィールドとして宣言に切り出す	Command + Alt + F	現在カーソルのある変数や関数をこのショートカットを実行することでフィールドに切り出すことができます。	
低	UML図の表示	Command + Alt + U	現在カーソルのあるクラスの継承関係をUML図にして見やすく表示してくれます。	<p>以下のようなUML図が表示されます。</p> 
低	リファクタリングメニュー表示	Command + Alt + Shift + T	現在カーソルのある箇所で行うことのできるリファクタリングメニューを表示してくれます。そのメニューから変数化や関数化などのリファクタリングを実行することができます。	<p>以下のようなリファクタリングメニューが表示されます。</p> 

## その他の便利機能

使用頻度	機能	使い方	説明	備考
高	型の簡易表示	Command + マウスでコードをホバー	マウスでホバーしている関数や変数の型や参照元を簡易表示できます。	
高	複数行の一括選択	Alt + マウスでコードドラッグ	マウス操作で直感的に複数行を選択状態にすることができ、一行ごとにカーソルがある状態（マルチカーソル）になります。このまま修正することで、複数行を同時に書き換えることができます。	
中	現在表示されているファイルの各行の最終更新日と更新者表示	行番号が表示されてるところを右クリック -> [Annotate] を選択	ファイルの行番号とかの横にGitのコミット情報を表示できます。	<p>行番号表示などの下に[Annotate]を選択します。</p>  <p>行番号の横にコミット番号・編集日・編集者・コミット回数が表示されます。</p> 
低	変更行を最終コミット状態に戻す	変更行の青色が変わっている箇所をクリック -> [Rollback] をクリック	部分的な変更箇所を変更前の状態と比較して、元に戻したりDiffを表示させたりできます。	<p>以下のようにメニューが表示されます。</p> 
低	現在のファイルの過去の変更点の差分を表示	ファイル上で右クリック -> [Git]選択 -> [Show History]選択 -> リビジョンをダブルクリック	現在のファイルのGitログから過去の変更点とその変更内容を差分表示できます。	<p>変更の差分は以下のように表示されます。</p> 
低	ファイルの変更	ファイル上で右クリック -> [Local	ローカル環境の変更履歴とそれぞれの変更での差分を一覧表示でき	<p>変更の差分は以下のように表示されます。</p> 



低	更履歴 の表示	History]選択 -> [Show History]選択	れの変更での差分を一覧表示で きます。	
---	------------	--------------------------------------	------------------------	--

## 参考・引用

- 忙しい人のためのIntelliJ IDEAショートカット集 -  
<http://qiita.com/yoppe/items/f7cbeb825c071691d3f2>
- 意外と知らないIntelliJ IDEAのGit管理機能いろいろ -  
<http://qiita.com/yoppe/items/fd03607d4d4f191d32dd>

# Incoming WebHooksを使ってScalaでSlackに投稿してみる

新卒の杉山です。

先日、RedmineとBitbucketのWebHookを使った連携に失敗してしまい、WebHook = 難しい というイメージを持ってました。しかし、ScalaからSlackに投稿する機能を作っていた際に使ったIncoming WebHooksという機能はめちゃめちゃ簡単だったので、実装したコードと一緒にご紹介したいと思います。

## Incoming WebHooks

Incoming WebHooksは、決められたフォーマットのJsonを決められたURLに送信するだけでSlackに投稿できるというものです。この機能を利用するにはあらかじめSlack側で設定しておく必要があります。

### Slack側の設定

Slack側の設定は大きく分けて2ステップあります。

1. 登録したいチャンネルにIncoming WebHooksを追加する
  - 以下のページに行く ([https://\[ドメイン\].slack.com/services/new/incoming-webhook](https://[ドメイン].slack.com/services/new/incoming-webhook))
  - 投稿したいチャンネルを選択する
2. Incoming WebHooksに名前やアイコンを設定する
  - 以下のページに行く ([https://\[ドメイン\].slack.com/services#service](https://[ドメイン].slack.com/services#service))
  - 先ほど設定したIncoming WebHooksを選択し、名前とアイコンを設定
  - (webhookURLが表示されるので控えておくといいかもしれません)

以上でSlack側で行う設定は終了です。

### 試しに投稿する

Slack側で設定さえしてしまえばたったこれだけで投稿できます。

#### コンソール

```
curl -X POST --data-urlencode 'payload={"text": "hello world"}' https://(webhook url)
```

## ScalaでSlackに投稿する

Slack側の設定が終わったので、Scalaで実装していきたいと思います。今回のサンプルでは主題・本文・@メンションを有効化・色・デスクトップ通知の内容を設定したメッセージを作ってSlackに投稿します。環境はsbtプロジェクトを想定しています。

### 必要なライブラリのインストール

指定されたURLにJsonで送信する必要があるので、build.sbtにhttpクライアントライブラリとJsonライブラリの依存性を追加しておきます。

#### build.sbt

```
libraryDependencies += "com.typesafe.play" % "play-ws_2.11" % "2.4.3"
libraryDependencies += "com.typesafe.play" % "play-json_2.11" % "2.4.3"
```

## 投稿するメッセージのフォーマットを定義

では、実際にSlackに投稿するメッセージを定義していきます。投稿するメッセージのフォーマットは決められているので、それに沿うように引数名を付けておけば後は勝手にいい感じのJsonに変換できるようにしています。

```
import play.api.libs.json._

/**
 * メッセージ
 * @param link_names @メンション有効化の設定（1だと有効）
 * @param attachments メッセージの内容
 */
case class Message(
  link_names: Int,
  attachments: Seq[Attachment]
)

/**
 * メッセージの内容
 * @param fallback デスクトップ通知に表示されるメッセージ
 * @param color 投稿に表示する棒(?)の色。good, warning, dangerを指定可能
 * @param title 主題（強調表示される）
 * @param text 本文
 */
case class Attachment(
  fallback: String,
  color: String,
  title: String,
  text: String
)

/**
 * Json変換用
 */
object Message {
  implicit val attachmentWrite = Json.writes[Attachment]
  implicit val messageWrite = Json.writes[Message]
}
```

これで今回使うメッセージフォーマットの定義は終了です。link\_namesとcolorは入る値がほぼ決まっているので、以下のような感じでEnumを定義してくるといいかもしれません。

```
/**
 * @Mentionの有効無効を表すEnum
 */
sealed abstract class Mention(val value: Int)
object Mention {
  case object True extends Mention(1)
  case object False extends Mention(0)
}

/**
 * 投稿の種類（色）を表すEnum
 */
sealed abstract class Color(val value: String)
object Color {
  case object Good extends Color("good")
  case object Warning extends Color("warning")
  case object Danger extends Color("danger")
}
```

今回記載したフォーマット以外にも2列に表示したり、3色以外にもカラーコードを指定して色をつけたりできます。詳しくは参考URLにある公式ページをご参照ください。

## メッセージをSlackに投稿する

最後に、定義したメッセージを使ってSlackに投稿します。webHookURLはSlack設定の手順2で操作したページから取得できます。

```
import play.api.libs.json._
import play.api.libs.ws._
import play.api.libs.ws.ning.NingWSClient
import scala.concurrent.duration._
import scala.concurrent.{Await, Future}

// Slackの設定ページから取得してください
val webHookURL = "https://hooks.slack.com/services/?????"

val message = Message(
  link_names = Mention.True.value,
  attachments = Seq(
    Attachment(
      fallback = "@hoge ゆっくりしてってね",
      color = Color.Good.value,
      title = "どうぞごゆっくり",
      text = "@hoge ゆっくりしてってね"
    )
  )
)

val data = Json.toJson(message)

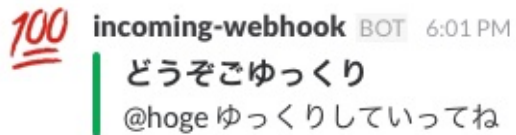
def using[A](client: NingWSClient)(f: NingWSClient => A): A = {
  try {
    f(client)
  } finally {
    client.close()
  }
}

using[WSResponse](NingWSClient()) { client =>
  val futureResponse: Future[WSResponse] = client.url(webHookURL).post(data)
  Await.result(futureResponse, Duration.Inf)
}
```

実装は以上となります。

## 投稿してみる

実際に投稿されるメッセージはこんな感じになります。



## 終わりに

長くなってしまいましたが、今回はIncoming WebHooksを使ったメッセージ投稿サンプルをご紹介させていただきました。Slack側の設定もシンプルで導入しやすいので、是非一度お試しください！

読んでくださいましてありがとうございました。《杉山》

## 参考URL

- <https://api.slack.com/incoming-webhooks>
- <https://api.slack.com/docs/formatting>
- <https://api.slack.com/docs/attachments>

# Beanstalk for Dockerを使ったPlay Framework実行環境構築

## 概要

弊社のプロジェクトでBeanstalk for DockerでPlay Frameworkアプリケーションを動かす機会がありましたので、ご紹介したいと思います。

## 環境

- Docker 1.8
- AWS Cloudformation
  - [sfn](#)<sup>1</sup>
  - [sparkle\\_formation](#)<sup>2</sup>
- AWS Elastic Beanstalk for Docker(単一コンテナ)<sup>3</sup>
  - 今回はDocker Hubは使わないです(コンテナはEC2インスタンス上でbuildします)
- Play Framework 2.4

## 構築方法

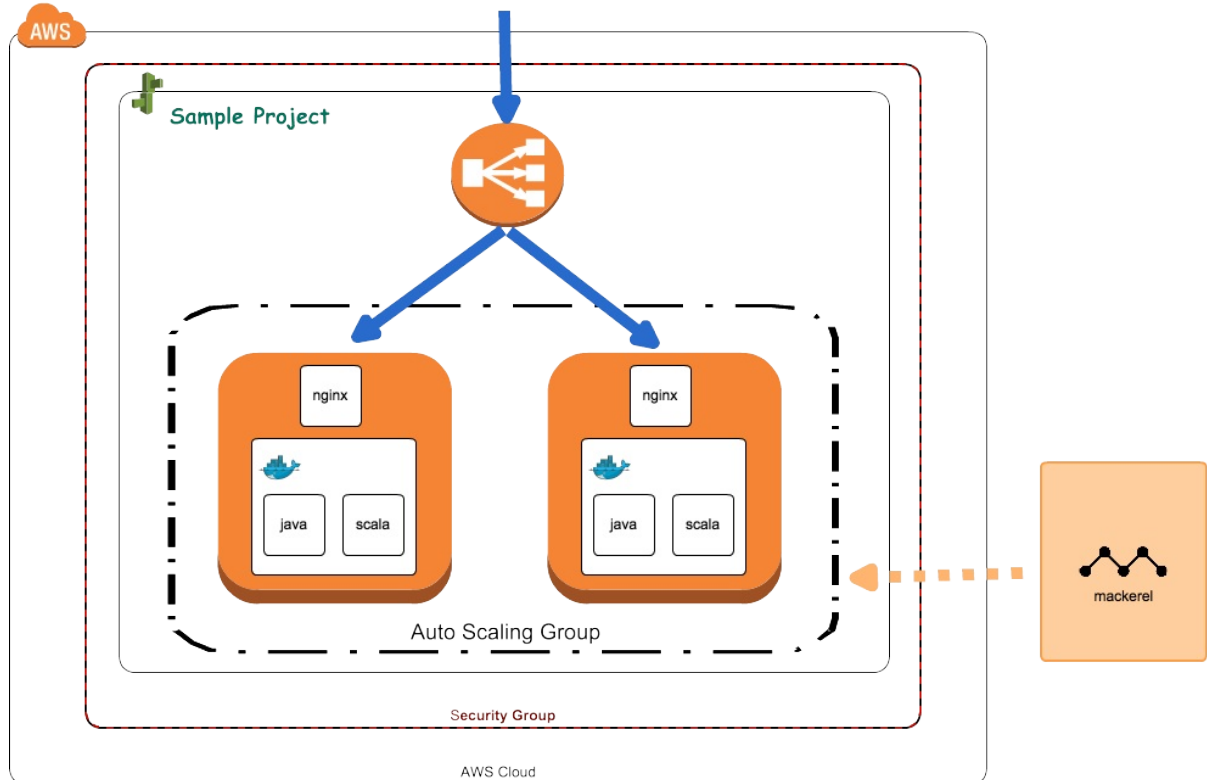
### ■ 構築する前に

- `key_pair`だけ先に作成しておきます。今回は`sample_project`としています。作成した秘密鍵はリネームせず、`~/.``ssh` に移動し、パーミッションを変更しておきます。※ 秘密鍵をリネームすると `eb ssh` した時に怒られます



## ■ 構成図

- 以下のイメージの環境を構築していきます。



## ■ 構築からデプロイまでの流れ

1. Beanstalk構築 with Cloudformation
2. PlayFrameworkでアプリケーション作成
3. Docker環境設定
4. ローカルでDockerコンテナ起動
5. デプロイ環境設定(awsebcliインストール)
6. Beanstalkにデプロイ

## ■ CloudformationでBeanstalk構築

CloudformationでBeanstalk for Dockerを構築していきます。

### 1. テンプレート作成

まずはテンプレートを作成します。直接jsonを編集するのは大変なので、sfnやsparkle\_formationという便利なgemパッケージを使って、Rubyで作成しました。

#### sparkle\_formationを利用したテンプレート作成例

```
# 以下のリソースが構築されます
# VPC
# InternetGateway
```

```
# RouteTable
# Subnet
# SecurityGroup
# Beanstalk for Docker

SparkleFormation.new(:SampleProjectTemplate) do
  set!('AWSTemplateFormatVersion', '2010-09-09')
  description "Sample Project"

  parameters do
    project do
      description 'Project Name'
      type 'String'
      default 'sample-project'
    end
  end
end

#####
# VPC
#####
resources(:Vpc) do
  type 'AWS::EC2::VPC'
  properties do
    cidr_block '10.0.0.0/16'
  end
end

#####
# Internet Gateway
#####
resources(:InternetGateway) do
  type 'AWS::EC2::InternetGateway'
  properties do
    tags _array(
      -> {
        key 'Project'
        value ref!(:Project)
      }
    )
  end
end

resources(:AttachGateway) do
  type 'AWS::EC2::VPCGatewayAttachment'
  properties do
    internet_gateway_id ref!(:internet_gateway)
    vpc_id ref!(:vpc)
  end
end
```

```
#####  
# RouteTable  
#####  
resources(:PublicRouteTable) do  
  type 'AWS::EC2::RouteTable'  
  properties do  
    vpc_id ref!(:Vpc)  
  end  
end  
  
# # Route for Internet  
resources(:PublicRoute) do  
  type 'AWS::EC2::Route'  
  properties do  
    destination_cidr_block '0.0.0.0/0'  
    gateway_id ref!(:InternetGateway)  
    route_table_id ref!(:PublicRouteTable)  
  end  
end  
  
#####  
# Subnet  
#####  
resources(:PublicSubnet) do  
  type 'AWS::EC2::Subnet'  
  properties do  
    availability_zone 'ap-northeast-1b'  
    cidr_block '10.0.0.0/24'  
    map_public_ip_on_launch 'true'  
    vpc_id ref!(:Vpc)  
  end  
end  
  
resources(:PublicSubnetRouteTableAssociation) do  
  type 'AWS::EC2::SubnetRouteTableAssociation'  
  properties do  
    route_table_id ref!(:PublicRouteTable)  
    subnet_id ref!(:PublicSubnet)  
  end  
end  
  
#####  
# Security Group  
#####  
resources(:PublicSecurityGroup) do  
  type 'AWS::EC2::SecurityGroup'  
  properties do  
    group_description 'Public Security Group'
```

```
    vpc_id ref!(:Vpc)
  end
end

resources(:PublicSecurityGroupIngress1) do
  type 'AWS::EC2::SecurityGroupIngress'
  properties do
    source_security_group_id ref!(:PublicSecurityGroup)
    from_port 0
    to_port 65535
    ip_protocol -1
    group_id ref!(:PublicSecurityGroup)
  end
end

resources(:PublicSecurityGroupEgress1) do
  type 'AWS::EC2::SecurityGroupEgress'
  properties do
    cidr_ip '0.0.0.0/0'
    from_port 0
    to_port 65535
    ip_protocol -1
    group_id ref!(:PublicSecurityGroup)
  end
end

resources(:ServerRole) do
  type 'AWS::IAM::Role'
  properties do
    assume_role_policy_document do
      statement _array(
        -> {
          effect 'Allow'
          principal do
            service _array(
              'ec2.amazonaws.com'
            )
          end
          action ['sts:AssumeRole']
        }
      )
    end
    path '/'
  end
end

resources(:ServerPolicy) do
  type 'AWS::IAM::Policy'
  depends_on "ServerRole"
```

```

    properties do
      policy_name 'ServerRole'
      policy_document do
        statement _array(
          -> {
            effect 'Allow'
            not_action 'iam:*'
            resource '*'
          }
        )
      end
      roles _array(
        ref!(:ServerRole)
      )
    end
  end
end

resources(:ServerInstanceProfile) do
  type 'AWS::IAM::InstanceProfile'
  depends_on "ServerRole"
  properties do
    path '/'
    roles _array(
      ref!(:ServerRole)
    )
  end
end

resources(:SampleProjectApplication) do
  type 'AWS::ElasticBeanstalk::Application'
  properties do
    description 'Sample Project Application'
    application_name 'SampleProject'
  end
end

resources(:SampleProjectApplicationEnvironment) do
  type 'AWS::ElasticBeanstalk::Environment'
  depends_on ["SampleProjectApplication", "ServerRole"]
  properties do
    application_name ref!(:SampleProjectApplication)
    description "Sample Project for Staging"
    solution_stack_name '64bit Amazon Linux 2015.03 v2.0.2 running Docker 1.7.1'
    environment_name 'SampleProjectStaging'
    CNAMEPrefix 'sample-projet-staging'
    tier do
      name 'WebServer'
      type 'Standard'
    end
  end
end

```

```
option_settings _array(  
  -> {  
    namespace 'aws:autoscaling:launchconfiguration'  
    option_name 'SSHSourceRestriction'  
    value "tcp,22,22,113.34.78.168/32"  
  },  
  -> {  
    namespace 'aws:autoscaling:launchconfiguration'  
    option_name 'SecurityGroups'  
    value ref!(:PublicSecurityGroup)  
  },  
  -> {  
    namespace 'aws:autoscaling:launchconfiguration'  
    option_name 'EC2KeyName'  
    value 'sample_project'  
  },  
  -> {  
    namespace 'aws:ec2:vpc'  
    option_name 'VPCId'  
    value ref!(:Vpc)  
  },  
  -> {  
    namespace 'aws:ec2:vpc'  
    option_name 'AssociatePublicIpAddress'  
    value "true"  
  },  
  -> {  
    namespace 'aws:ec2:vpc'  
    option_name 'Subnets'  
    value ref!(:PublicSubnet)  
  },  
  -> {  
    namespace 'aws:ec2:vpc'  
    option_name 'ELBSubnets'  
    value ref!(:PublicSubnet)  
  },  
  -> {  
    namespace 'aws:autoscaling:launchconfiguration'  
    option_name 'InstanceType'  
    value 't2.micro'  
  }  
)  
end  
end  
end
```

## 2. stack作成

sfnコマンドでstackを作成します。

```
$ bundle exec sfn create sample-project-Beanstalk -b templates/ -f templates/Beanstalks.json -P
```

```
[Sfn]: SparkleFormation: create
[Sfn]:   -> Name: sample-project
[Sfn]: Stack runtime parameters:
[Sfn]: Project [sample-project]:
[Sfn]: Events for Stack: sample-project
```

Time	Resource	Logical Id	Resource Status
2015-11-27 03:01:36 UTC	sample-project		CREATE_IN_PROGRESS
User Initiated			
2015-11-27 03:01:40 UTC	Vpc		CREATE_IN_PROGRESS
2015-11-27 03:01:40 UTC	InternetGateway		CREATE_IN_PROGRESS
2015-11-27 03:01:40 UTC	SampleProjectApplication		CREATE_IN_PROGRESS

```
(snip)
```

2015-11-27 03:04:08 UTC	ServerInstanceProfile		CREATE_COMPLETE
2015-11-27 03:08:32 UTC	SampleProjectApplicationEnvironment		CREATE_COMPLETE
2015-11-27 03:08:34 UTC	sample-project		CREATE_COMPLETE

```
[Sfn]: Stack create complete: SUCCESS
[Sfn]: Stack description of sample-project:
[Sfn]: Outputs for stack: sample-project
[Sfn]:   No outputs found
```

## 3. Beanstalkの確認

Beanstalk構築後はサンプルのアプリケーションが動いてますので、URLにアクセスしてページが表示されるか確認します。今回はcname\_prefixをsample-projet-stagingとしたので、<http://sample-projet-staging.elasticbeanstalk.com/> がURLになります



## ■ アプリケーションの作成

### 1. アプリケーションの作成

Play Frameworkで簡単なサンプルを準備します。

```
# 新規アプリケーション作成
$ ./activator new sample-project play-scala
$ cd sample-project/

# コンテンツ作成
$ vim app/views/main.scala.html
@(title: String)(content: Html)
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>@title</title>
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png" href="@routes.Assets.versioned("images/favicon.png")">
    <script src="@routes.Assets.versioned("javascripts/hello.js")" type="text/javascript"></script>
  </head>
  <body>
    <H1>2016 Scala Maturi</H1>
    <H2>Sample Project</H2>
  </body>
</html>

# 確認
$ ./activator run
```

### 2. buildしてjarを生成

SBT assembly plugin<sup>4</sup> を使ってjarを生成します。公式サイト<sup>5</sup>の手順に沿って対応していきます。

```
$ vim project/plugins.sbt
# 追加
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.1")

$ vim build.sbt
# 追加
import AssemblyKeys._

assemblySettings

mainClass in assembly := Some("play.core.server.NettyServer")

fullClasspath in assembly += Attributed.blank(PlayKeys.playPackageAssets.value)

# java.lang.RuntimeException: deduplicate: different file contents found in the
# following:とエラーが起きたので追加
mergeStrategy in assembly <=< (mergeStrategy in assembly) {
  (old) => {
    case "log4j.properties" => MergeStrategy.first
    case x =>
      old(x)
  }
}

# build
$ ./activator assembly
[info] Loading project definition from /Users/t_saeki/Develop/sample-project/pro
ject
[info] Set current project to sample-project (in build file:/Users/t_saeki/Devel
op/sample-project/)
[info] Including from cache: config-1.3.0.jar
[info] Including from cache: bonecp-0.8.0.RELEASE.jar

(snip)

[info] Assembly up to date: /Users/t_saeki/Develop/sample-project/target/scala-2
.11/sample-project-assembly-1.0-SNAPSHOT.jar
[success] Total time: 6 s, completed 2015/11/26 18:24:05

# 実行
# ブラウザでhttp://localhost:9000にアクセスして、ページが表示されることを確認します
$ java -jar target/scala-2.11/sample-project-assembly-1.0-SNAPSHOT.jar
NettyServer.main is deprecated. Please start your Play server with the ${ProdSer
verStart.getClass.getName}.main.
[info] - play.api.Play - Application started (Prod)
[info] - play.core.server.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:9
000
```

## ■ Docker作成

作成したアプリケーションをDockerで動かすようにします。

### 1. 事前にDockerをインストールしておく

公式サイト<sup>6</sup>の手順でインストールしていきます。 `docker-machine` と `docker-compose` もインストールされますが今回は使いません

### 2. Dockerfile作成

- Dockerfile<sup>7</sup>はアプリケーションと同じディレクトリに作成しておきます
- 今回は公式のjavaイメージ<sup>8</sup>をベースにイメージを作成します

```
$ vim Dockerfile
FROM java:8

RUN mkdir /opt/sample-project
COPY target/scala-2.11/sample-project-assembly-1.0-SNAPSHOT.jar /opt/sample-project/
EXPOSE 9000
CMD ["java", "-jar", "/opt/sample-project/sample-project-assembly-1.0-SNAPSHOT.jar"]
```

### 3. Docker buildとrun

- Dockerをbuildして、実際にローカルで動かしてみます。事前にDockerを起動しておきます
- Dockerコマンドの詳細はリファレンス<sup>9</sup>を参考にしてください

```
$ docker build -t tsaeki/sample-project .
Sending build context to Docker daemon 162 MB
Step 0 : FROM java:8
---> 36621d4ab8e3
Step 1 : RUN mkdir /opt/sample-project
---> Using cache
---> b5e800d8653f
Step 2 : COPY target/scala-2.11/sample-project-assembly-1.0-SNAPSHOT.jar /opt/sample-project/
---> Using cache
---> 74ec351a9034
Step 3 : EXPOSE 9000
---> Using cache
---> fd1947bdc2e5
Step 4 : CMD java -jar /opt/sample-project/sample-project-assembly-1.0-SNAPSHOT.jar
---> Using cache
---> 0a8f13d586ce
Successfully built 0a8f13d586ce

# 実行
# ブラウザでhttp://192.168.99.100:9000にアクセスして、ページが表示されることを確認します
$ Docker run -it --rm -p 9000:9000 tsaeki/sample-project
NettyServer.main is deprecated. Please start your Play server with the ${ProdServerStart.getClass.getName}.main.
[info] - play.api.Play - Application started (Prod)
[info] - play.core.server.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:9000
```

## 4. Dockerrun.aws.json作成

BeanstalkにDockerをデプロイするにはDockerrun.aws.json<sup>10</sup>が必要になりますので、作成しておきます

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "tsaeki/sample-project",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": "9000"
    }
  ],
  "Volumes": [],
  "Logging": ""
}
```

## ■ 実際にデプロイしてみる

実際に構築したBeanstalk環境にデプロイしてみます。

### 1. EB CLIインストール for Mac

事前にEB CLI<sup>11</sup>というツールをインストールしておきます。公式サイト<sup>12</sup>の手順でインストールしていきます。 ※ brewだと2.x系になってしまうようなので、pipからインストールしましょう。

```
$ sudo pip install awsebcli
```

### 2. EB CLIの初期設定

`eb init` で初期設定をします。RegionやApplicationを選択していきます。

```
$ eb init

Select a default region
1) us-east-1 : US East (N. Virginia)
2) us-west-1 : US West (N. California)
3) us-west-2 : US West (Oregon)
4) eu-west-1 : EU (Ireland)
5) eu-central-1 : EU (Frankfurt)
6) ap-southeast-1 : Asia Pacific (Singapore)
7) ap-southeast-2 : Asia Pacific (Sydney)
8) ap-northeast-1 : Asia Pacific (Tokyo)
9) sa-east-1 : South America (Sao Paulo)
10) cn-north-1 : China (Beijing)
(default is 3): 8

Select an application to use
1) SampleProject
2) [ Create new Application ]
(default is 2): 1

It appears you are using Docker. Is this correct?
(y/n): y

Select a platform version.
1) Docker 1.7.1
2) Docker 1.6.2
(default is 1): 1

Do you want to set up SSH for your instances?
(y/n): y

Select a keypair.
1) sample_project
2) [ Create new KeyPair ]
(default is 2): sample_project
```

※ やり直したい場合は、 `eb init -i` で

### 3. デプロイをする

```
# Environment list確認
$ eb list
* SampleProjectStaging

$ eb deploy SampleProjectStaging
Creating application version archive "app-151127_133042".
Uploading: [#####] 100% Done...
INFO: Environment update is starting.
INFO: The environment does not have an IAM instance profile associated with it.
To improve deployment speed please associate an IAM instance profile with the environment.
INFO: Deploying new version to instance(s).
INFO: Successfully pulled java:8
INFO: Successfully built aws_Beanstalk/staging-app
INFO: Docker container d72d6432f5f8 is running aws_Beanstalk/current-app.
INFO: New application version was deployed to running EC2 instances.
INFO: Environment update completed successfully.

# 実際はSampleProjectStagingはデフォルト設定(*印)になっているので省略が可能です。
```

### 4. コンテンツを確認する

ブラウザで<http://sample-projet-staging.elasticbeanstalk.com/> にアクセスして、先ほどローカルで確認した内容と同じか確認してみます。同じ内容ならデプロイに成功しています。おめでとうございます！

## ■ 監視(mackerel)について

監視についてもちっと書いておこうかと思います。監視はmackerel<sup>13</sup>を使っています。mackerelを使ってどのように監視しているかを説明したいと思います。

### 概要

- ホストサーバにmackerelを導入する際はebextensions<sup>14</sup>でカスタマイズしてきます。
- Docker内で稼働してるjvmのリソース情報をhttp経由で取得するためjolokia<sup>15</sup>を使っています。
- 事前にmackerelに登録して、api keyを取得しておきます。

### 1. コンテナ内でjolokiaを実行する

ホストサーバからDocker内のJVMの情報を取得するため、jolokiaを使い、ホストサーバからhttp経由でJVMの情報を取得します<sup>16</sup>

```
# jolokiaのjarをダウンロード
$ curl -O https://repo1.maven.org/maven2/org/jolokia/jolokia-jvm/1.3.2/jolokia-jvm-1.3.2-agent.jar
```

```
# jar実行時にjolokia agentも起動させるように修正
$ vim Dockerfile
FROM java:8

RUN mkdir /opt/sample-project
COPY target/scala-2.11/sample-project-assembly-1.0-SNAPSHOT.jar /opt/sample-project/
# ダウンロードしたjarファイルをコピー
COPY jolokia-jvm-1.3.2-agent.jar /opt/sample-project/

# 8778ポートはホストからコンテナにアクセスするために設定しています
EXPOSE 9000 8778

# "-javaagent:/opt/sample-project/jolokia-jvm-1.3.2-agent.jar=port=8778,host=0.0.0.0"を追加する
CMD ["java", "-javaagent:/opt/sample-project/jolokia-jvm-1.3.2-agent.jar=port=8778,host=0.0.0.0", "-jar", "/opt/sample-project/sample-project-assembly-1.0-SNAPSHOT.jar"]

# Docker build, run
$ Docker build -t tsaeki/sample-project .

# 8778ポートにアクセスするために、-p 8778:8778を追加
$ Docker run -it --rm -p 9000:9000 -p 8778:8778 tsaseki/sample-project

# 情報が取得できるか確認する
$ curl -s http://192.168.99.100:8778/jolokia/ | jq .
{
  "request": {
    "type": "version"
  },
  "value": {
    "agent": "1.3.2",
    "protocol": "7.2",
    "config": {
      "maxDepth": "15",
      "discoveryEnabled": "true",
      "maxCollectionSize": "0",
      "agentId": "172.17.0.3-1-5e2de80c-jvm",
      "debug": "false",
      "agentType": "jvm",
      "historyMaxEntries": "10",
      "agentContext": "/jolokia",
      "maxObjects": "0",
      "debugMaxEntries": "100"
    },
    "info": {}
  },
  "timestamp": 1448603006,
```



```
"status": 200
}
```

## 2. .ebextensionsでホストサーバにmackerelをインストール、pluginの設定する

```
$ mkdir .ebextensions
$ vim .ebextensions/01_install_mackerel.config
files:
  /etc/mackerel-agent/mackerel-agent.conf:
    mode: "00644"
    owner: root
    group: root
    encoding: plain
    content: |
      apikey = "<mackerel api key>"
      include = "/etc/mackerel-agent/conf.d/*.conf"
      [plugin.metrics.Docker]
      command = "/usr/local/bin/mackerel-plugin-Docker"

# コンテナのIPアドレス
/opt/elasticBeanstalk/hooks/appdeploy/post/99_setup-jolokia-config-for-mackere
l.sh:
  mode: "00755"
  owner: root
  group: root
  encoding: plain
  content: |
    #!/bin/sh
    # setup jolokia monitoring
    Docker_IP=$(docker inspect --format '{{.NetworkSettings.IPAddress}}' $(d
ocker ps -q -l))
    MACKEREL_HOSTID=$(cat /var/lib/mackerel-agent/id)
    cat > /etc/mackerel-agent/conf.d/jolokia.conf <<EOF
    [plugin.metrics.jolokia]
    command = "/usr/local/bin/mackerel-plugin-jolokia.rb ${Docker_IP} ${MACKER
EL_HOSTID}"
    EOF

    # restart mackerel
    /etc/init.d/mackerel-agent restart

/usr/local/bin/mackerel-plugin-jolokia.rb:
  mode: "00755"
  owner: root
  group: root
  encoding: plain
  content: |
    #!/usr/bin/env ruby
```

```

require 'mackerel/client'
require 'socket'
require 'uri'
require 'net/http'
require 'json'

MACKEREL_KEY = 'etEwwFKkZJW6goWf8Tqmx7awjz5aPucZC46xQm92WDHH'

monitoring_ip = ARGV[0]
host_id = ARGV[1]
base_url = "https://mackerel.io//api/v0/tsdb"
end_time = Time.now

uri = URI.parse("http://#{monitoring_ip}:8778/jolokia/read/java.lang:type
=Memory/HeapMemoryUsage")
get_json_memory = Net::HTTP.get(uri)
get_heap_memory = JSON.load(get_json_memory)

metrics = [
  {'hostId' => host_id, 'name' => "HeapMemoryUsage.init", 'time' => end_time.to_i, 'value' => get_heap_memory['value']['init'] },
  {'hostId' => host_id, 'name' => "HeapMemoryUsage.committed", 'time' => end_time.to_i, 'value' => get_heap_memory['value']['committed'] },
  {'hostId' => host_id, 'name' => "HeapMemoryUsage.max", 'time' => end_time.to_i, 'value' => get_heap_memory['value']['max'] },
  {'hostId' => host_id, 'name' => "HeapMemoryUsage.used", 'time' => end_time.to_i, 'value' => get_heap_memory['value']['used'] },
]

@mackerel = Mackerel::Client.new(:mackerel_api_key => MACKEREL_KEY)
ret = @mackerel.post_metrics(metrics)

commands:
00_enable_sudo:
  command: sed -i 's/requiretty/!requiretty/' /etc/sudoers
01_setup_yumrepo_for_mackerel:
  command: curl -fsSL https://mackerel.io/assets/files/scripts/setup-yum.sh | sh
02_install_mackerel_rpm_packages:
  command: yum install -y mackerel-agent mackerel-agent-plugins
03_install_mackerel_gem_packages:
  command: /usr/bin/gem install mackerel-client
04_create_mackerel_dir:
  command: mkdir -p /etc/mackerel-agent/conf.d

container_commands:
start_mackerel_agent:
  command: "/etc/init.d/mackerel-agent restart"

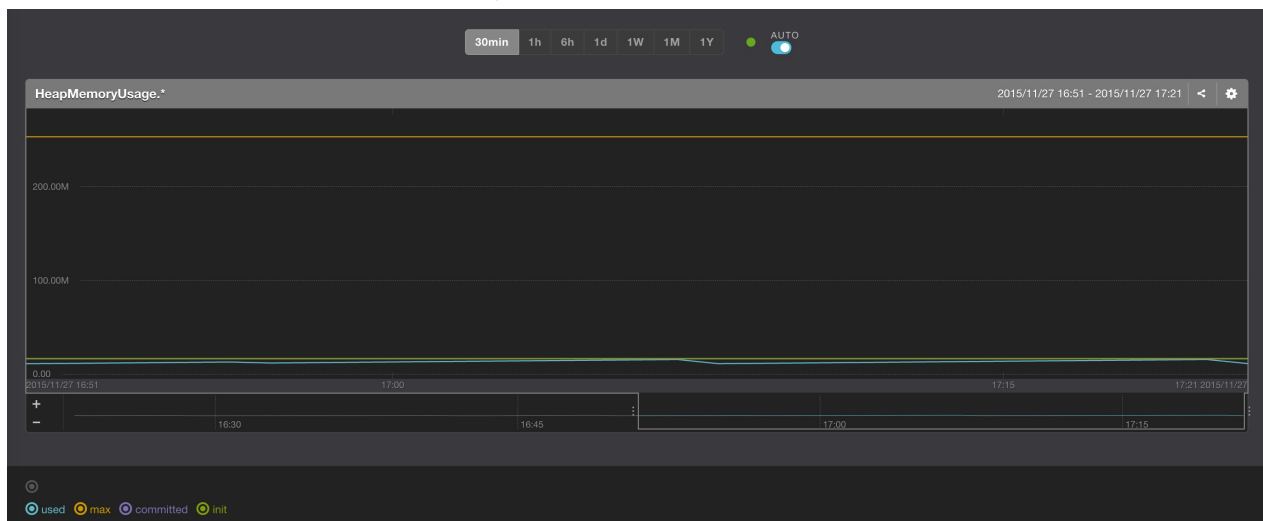
```

### 3. デプロイして、.ebextensionsを反映させる

```
$ eb deploy SampleProjectStaging
$ eb deploy SampleProjectStaging
Creating application version archive "app-151127_162030".
Uploading: [#####] 100% Done...
INFO: Environment update is starting.
INFO: Deploying new version to instance(s).
INFO: Successfully pulled java:8
INFO: Successfully built aws_Beanstalk/staging-app
INFO: Docker container 1c4d16bd9b12 is running aws_Beanstalk/current-app.
INFO: New application version was deployed to running EC2 instances.
INFO: Environment update completed successfully.
```

### 4. mackerelにグラフが表示されているか確認する

無事グラフが表示されていれば成功です。



### 注意点

- 実行の順番は上からではなく、アルファベット順なので、意図的に01\_というprefixをつけておきましょう
- アプリ実行後に処理をしたい場合は/opt/elasticBeanstalk/hooks/appdeploy/post/にスクリプトを置くといひようです

### 最後に

Dockerを本番環境で使うのはまだ早いな、と思ってましたが、Beanstalk使うことで敷居が低くなりました。環境は比較的簡単に構築できましたが、mackerelのインストールなどBeanstalkのカスタマイズが必要になると、結構面倒な感じでした。今後はそこらへんを改善できればと思ってます。今後Dockerを本番環境で使っていこう、という方に参考になれば。

### 参考

- <sup>1</sup> <https://github.com/sparkleformation/sfn>

- <sup>2</sup> [https://github.com/sparkleformation/sparkle\\_formation](https://github.com/sparkleformation/sparkle_formation)
- <sup>3</sup> [http://docs.aws.amazon.com/ja\\_jp/elasticbeanstalk/latest/dg/docker-singlecontainer-deploy.html](http://docs.aws.amazon.com/ja_jp/elasticbeanstalk/latest/dg/docker-singlecontainer-deploy.html)
- <sup>4</sup> <https://github.com/sbt/sbt-assembly>
- <sup>5</sup> <https://www.playframework.com/documentation/ja/2.3.x/ProductionDist>
- <sup>6</sup> <https://docs.docker.com/v1.8/installation/mac/>
- <sup>7</sup> <https://docs.docker.com/v1.8/reference/builder/>
- <sup>8</sup> [https://hub.docker.com/\\_/java/](https://hub.docker.com/_/java/)
- <sup>9</sup> <https://docs.docker.com/v1.8/reference/commandline/cli/>
- <sup>10</sup>  
[http://docs.aws.amazon.com/ja\\_jp/elasticbeanstalk/latest/dg/create\\_deploy\\_docker\\_image.html#create\\_deploy\\_docker\\_image\\_dockerrun](http://docs.aws.amazon.com/ja_jp/elasticbeanstalk/latest/dg/create_deploy_docker_image.html#create_deploy_docker_image_dockerrun)
- <sup>11</sup> [http://docs.aws.amazon.com/ja\\_jp/elasticbeanstalk/latest/dg/eb-cli3.html](http://docs.aws.amazon.com/ja_jp/elasticbeanstalk/latest/dg/eb-cli3.html)
- <sup>12</sup> [http://docs.aws.amazon.com/ja\\_jp/elasticbeanstalk/latest/dg/eb-cli3-install.html](http://docs.aws.amazon.com/ja_jp/elasticbeanstalk/latest/dg/eb-cli3-install.html)
- <sup>13</sup> <https://mackerel.io/ja/>
- <sup>14</sup> [http://docs.aws.amazon.com/ja\\_jp/elasticbeanstalk/latest/dg/ebextensions.html](http://docs.aws.amazon.com/ja_jp/elasticbeanstalk/latest/dg/ebextensions.html)
- <sup>15</sup> <https://jolokia.org/>
- <sup>16</sup> 参考「Play2 が動いている JVM を Jolokia で監視する」  
<http://blog.cloudpack.jp/2014/11/20/monitor-jvm-running-play-framework-2-with-jolokia/>

# Python学生からプロScalaエンジニアに

## はじめに

こんにちは。早瀬と申します。2015年入社の新卒です。  
学生時代はPythonでツイート取得したりWebのクローラを作成して遊んでいました。  
現在はPYXIS For Facebookという広告最適化ツールの開発を担当しています。  
弊社は全システムをScalaで開発しており、新人研修もScalaで行っています。  
今回は、私が受けたScala研修の内容を紹介したいと思います。

## 新卒研修

3ヶ月の研修のトピックは以下です。

- ① 外部講師の方(@OE\_uiaさん)からのScalaレクチャー
- ② 掲示板をScala & Playで作成
- ③ 掲示板をScala & Play & ドメイン駆動設計(DDD)で作成

### ① 外部講師(@OE\_uiaさん)からのScalaレクチャー

一番初めに、@OE\_uiaさんから約1ヶ月のScalaのレクチャーを受けました。  
ここでは、Scalaとsbt（ビルドツール）specs2（テストフレームワーク）についてについて学びました。  
レクチャーは、「Scalaスケーラブルプログラミング（通称 コップ本）」を元に進めていきました。

#### レクチャーの進め方

1. 本日の研修の目的を明確にする
2. 教材(コップ本や公式ドキュメントなど)を読みつつREPLでコードを書く
3. 演習問題を行う
4. 演習問題を@OE\_uiaさんにレビューしてもらう

## Python学生のScalaレビュー

今までPythonを使っていた学生が実際にScalaを触ってみて戸惑った点やいいなと思った点について書いていきます。

初めてScalaを触ってみて戸惑った点は、変数の再代入についてです。  
私に変数をPythonで扱うときは、再代入を行います。Scalaの場合 val 使用するので再代入は基本的にしないと言われた時には少々戸惑いました。  
しかし、課題をやったり関数型のソースなどを見ていると次第になれることができました。  
また、valを使うメリットなども理解できるようになりました。

Pythonで抽象クラスを記述する場合は

```

from abc import ABCMeta, abstractmethod
class AbstractClass(object):
    __metaclass__ = ABCMeta      # 抽象基底クラスを定義します

    @abstractmethod             # 抽象メソッドを示すデコレーター
    def sample_method(self):
        print "abstract"

```

抽象基底クラスを呼び出してやって、クラスが抽象基底であると定義しなければなりません。  
抽象メソッドの場合はメソッドの上に `@abstractmethod` というデコレータを付けます。

```

abstract class AbstractClass() {
    def sampleMethod(): Unit = {
        println("abstract")
    }
}

```

一方Scalaだとabstract class を宣言してやればいいだけなので何も考えずに実装できる点がいい点だと思いました。

mapなどの記法も違います。Python の場合だと、関数の呼び出しが入れ子型になります。

```

>>> result = range(0, 10)
>>> result = map(lambda x: x + 1, result)
>>> result
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

Scalaだとメソッドチェーンを利用して書くことができます。

```

scala> (0 until 10).map(f => f + 1).toList
res4: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

```

メソッドチェーンだとScalaだと左から右へ目を動かしていったという処理をしているかと言うのが、読みやすいなと思いました。

## Scalaレクチャーを受けて

本研修は日々の目的が明示的なので、学習自体も進めやすかったです。  
重要な部分は用途とメリットを聞いて進めたので実装のイメージを持つことができました。  
演習問題のレビューでは、書き方の問題から実行時内部の内容まで細かく教えていただきました。



taisukeoe commented on 29 Apr

#課題5ハノイ

良いと思います。ただ、Scalaの習慣として、towerOfHanoiのようにメソッド名は小文字から始めましょう。

#P7

パターンマッチで `case e:List[A] => flatten(e)` 書いても実行時には型消去(イレイジャ)により `case e:List[_]` として解釈されて、Listの型パラメーターはチェックされません。List[A]と書くにあたかもList[A]にしかマッチしないように誤解を生み、バグの元にもなるので、ここでは `List[_]` と書くほうが良いです。

それ以外はよく書けています。

#8

よく書けています。

非推奨な古い内容については他の教材（Scala公式ドキュメントなど）を使い、最新の情報を教えてもらうことができました。

自分一人でコップ本を読むよりも実際にScalaを使っている人に最新の情報を聞き、用途を踏まえた上で解説していただいて理解と定着がよりスムーズに進みました。

## ② 掲示板をScala & Playで作成

Scalaレクチャーの後には、掲示板をScala & Playで作りました。  
ここでは、WebフレームワークのPlayFrameworkと単体テストに使われるSpecs2について学びました。

研修は、実際の開発と同じScrumで進められます。  
弊社では開発手法にScrumを利用しています。  
実際の開発ではプロダクトオーナー（通称 PO）と開発チームに分かれています。  
POは機能を実装するかどうかを判断し、開発チームは機能を実装します。  
開発チームの中から一人システムリーダー（通称 SL）を立てます。  
SLは開発チームのメンバーから相談を受けたり、それに対するアドバイスを行います。  
今回の研修ではPOとSLが一人づつ立ち、私が一人で開発を担当します。  
研修のルールにしたがって、1つずつストーリーを進めていきます。

研修のルールとストーリーは以下になります。

### 掲示板作成のルール

1. ストーリーが完了しないと次のストーリーに進むことはできない（どのストーリーから進めてもよい）
2. 完了条件はPOにデモをすることと、SLからソースコードレビューの承認

### ストーリー

1. ユーザーは掲示板の投稿を一覧できる。
2. 投稿者のメアドと投稿内容が閲覧できる。
3. ユーザーは掲示板に文字列を投稿できる。
4. ユーザーは、メアドとパスワードを入力することでログインできる。
5. ログイン後であればユーザーは自分のメアドを入力せずに投稿できる。
6. 新規ユーザー登録を行うことができる。

完成した掲示板がこちらです。

detail	email
aaaaa	hoge@example.com
hogehogeo	hoge@example.com
やっほー	hoge@example.com
ほげほげ	hoge@example.com
ふがふが	fuga@example.com

## Post Message

detail

addMessage

この掲示板研修で困ったところは、メソッドの責務を考えずにコードを書いていった結果、見にくくテストがしづらいコードになりました。

これは、コントローラーにレスポンスを返す処理の部分にSQLの処理を記述したことで、コントローラーの責務が大きくなりました。

```
// 投稿の一覧のレスポンスを返す処理
def index = Action {
  implicit val connection = DB.getConnection()
  val sql = SQL(
    """SELECT Message.id, detail, email FROM User JOIN Message
    |ON User.id = user_id ORDER BY Message.id ASC"""
  ).stripMargin

  val userSeq = sql().map(
    row => row[String]("detail") -> row[String]("email")
  ).toSeq

  Ok(views.html.index(userSeq))
}
```

今回はメソッドが何を行っているかを把握するためにも、これからは各メソッドの責務を減らしてより単体テストが容易な状態で実装しようと思いました。

PlayFrameworkとSpecs2を学んだことによって、Webアプリケーションの開発ができるようになりました。

## ③ 掲示板をScala & Play & ドメイン駆動設計(DDD)で作ってみた

2度目の掲示板研修ではドメイン駆動設計について学びました。

ドメイン層の実装にはscala-dddbaseというライブラリを使っています。この研修は1度目とストーリーは同じなのですが、新たなルールが加わりました。



## 掲示板作成の追加ルール

1. ストーリーの番号順にやる
2. 各ストーリーごとにユビキタス言語とコンテキストマップとドメイン図をシステムリーダーがレビューをする
3. 2.の承認がないと実装に入ることができない

このルールに則って、以下のように掲示板研修を進めていきました。

1. ストーリーを改変とユビキタス言語の定義
2. コンテキストマップの作成
3. ドメイン図を作成
4. 開発

### 1.ストーリーの改変とユビキタス言語の定義

ここでは「ユーザーは、メールアドレスとパスワードを入力することでログインできる。」を例に出して説明していきます。

私はこのストーリーを行うまでにユーザーが掲示板に文字列を投稿&閲覧できる仕組みを作りました。

このストーリーではユーザーがログインできる仕組みを作ります。

しかしこのストーリーには、ログインする目的が書いてありませんでした。

そこで、POにこのストーリーの目的を聞きました。

私：「このストーリーでユーザーがログインしたら、何ができるようになるんですか？」

PO：「ユーザーはログインしたら、掲示板に投稿ができるようになるよ。」

ログインする目的がわかったのでストーリーを書き換えました。

「ユーザーは、メールアドレスとパスワードを入力することでログインできる。ログインできたユーザーは投稿ができるようになる。」

これで手段の内容と目的がわかるようになりました。

先ほどの改変したストーリーの中で、ログインできたユーザーが出てきました。

このユーザーは掲示板への投稿ができるようになります。

ということは、ログインしていない or できないユーザーは掲示板への投稿ができないということになります。

ここで使われるユーザーという言葉はとても曖昧なので、ユビキタス言語を決めます。

このストーリーで定義が必要なのはログインしたユーザーです。

私はログインユーザーと名づけました。

そして、ここでもストーリーが改変できます。

「ユーザーは、メールアドレスとパスワードを入力することでログインできる。ログインできたユーザーはログインユーザーになる。ログインユーザーは掲示板へのポスティングが可能になる。」

ユビキタス言語はこちらです

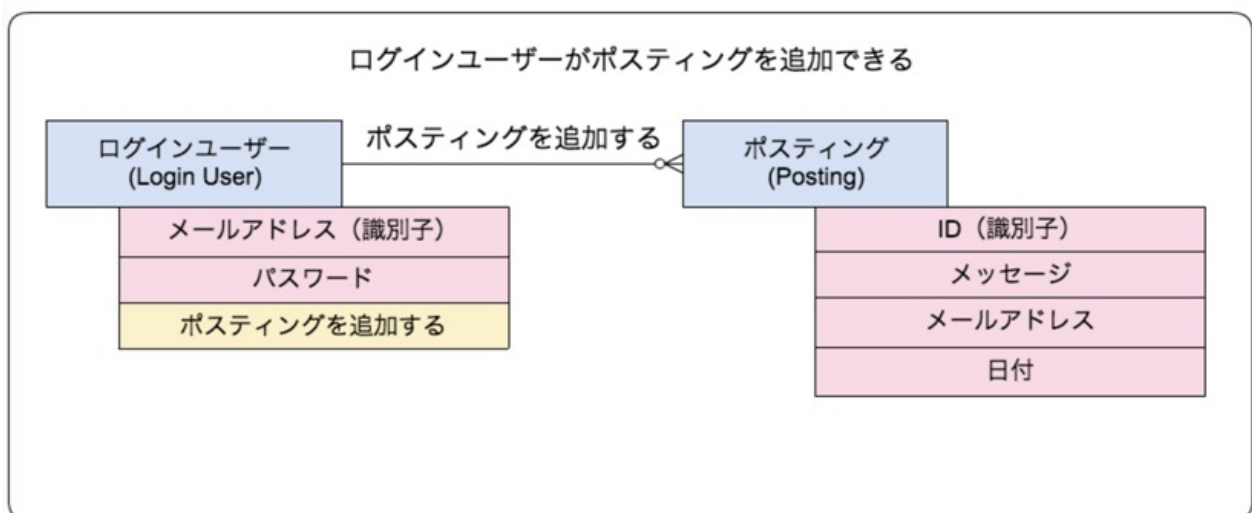
言葉	意味
ポスティング	掲示板上の投稿
ログインユーザー	掲示板サービスにメールアドレスとパスワードが登録されている人。この人はポスティングの追加ができる。
メールアドレス	メールアドレス
パスワード	パスワード
メッセージ	ポスティングの内容
日付	ポスティングが行われた日時

## 2. コンテキストマップの作成

改変したストーリーと定義したユビキタス言語を元にコンテキストマップに落とし込みました。

ポスティングは識別子（ID）とポスティングの内容を示すメッセージ、ポスティングの内容メールアドレスとメッセージを持っています。

ログインユーザーは識別子（メールアドレス）と属性（パスワード）と、振る舞い（ポスティングを掲示板に追加する）を持っています。



## 3. ドメイン図の作成

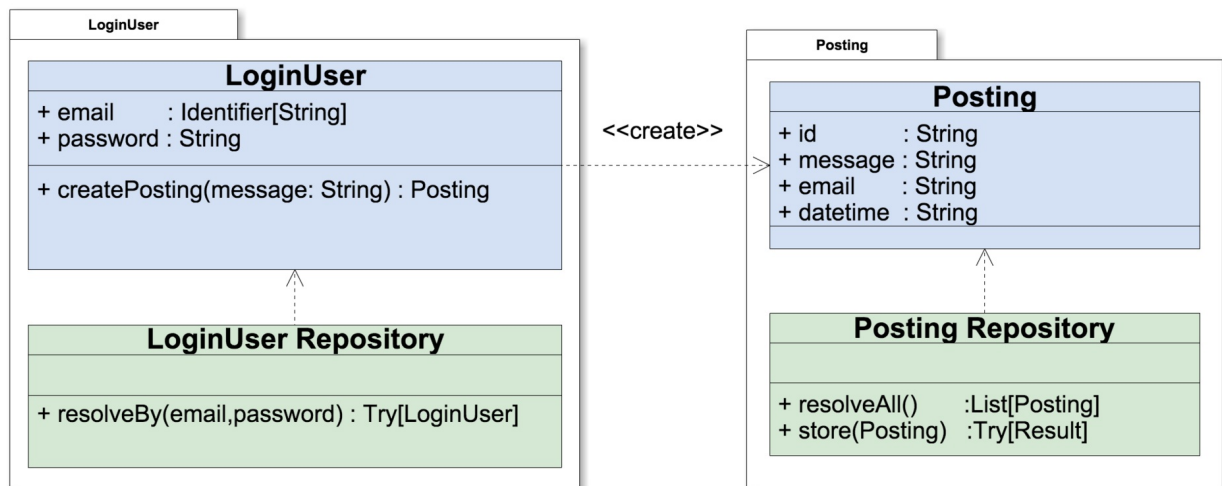
作成したコンテキストマップを元にドメイン図の作成をおこないます。ドメイン図とは弊社のユビキタス言語で、コンテキストマップをクラス図のように起こしたものです。

コンテキストマップで定義したエンティティと識別子と属性と振る舞いを実装をイメージして考えていきます。

定義した2つのエンティティは永続化が必要になるため、リポジトリを定義しました。

ログインユーザーのリポジトリはログインユーザーを取得するだけなので、resolveByというメソッドを定義します。

作成したドメイン図が以下になります。



## 4.開発

ドメイン図を用いてドメイン層の実装をしていきました。

```

/**
 * ポスティング
 * id: ID (識別子)
 * message: メッセージ
 * email: メールアドレス
 * dateTime: ポスティングが作成された日時
 */
case class PostingID(value: String) extends Identifier[String]

trait Posting extends Entity[PostingID] {

  val identifier : PostingID
  val message    : String
  val email      : String
  val dateTime   : String

}

/**
 * ログインユーザー
 * email: メールアドレス (識別子)
 * password: パスワード
 * createPosting: ポスティングを書き込む振る舞い
 */

case class LoginUserEmail(value: String) extends Identifier[String]

trait LoginUser extends Entity[LoginUserEmail] {

  val identifier : LoginUserEmail
  val password   : String

}

```

```
// ログインユーザーの実装部分
private[loginUser] case class LoginUserImp(
  identifier: LoginUserEmail,
  password: String) extends LoginUser {

  def createPostingEntity(message: String): Posting = {
    Posting(
      PostingID(java.util.UUID.randomUUID.toString),
      message,
      identifier.value,
      DateTime.now.toString
    )
  }
}

/**
 * AbstractRepository
 * store(entity: E): エンティティを保存する
 * resolveBy(identifier: ID): 識別子に対応するエンティティを探すメソッド
 */
abstract class AbstractRepositoryOnAnorm[ID <: Identifier[_], E <: Entity[ID]]
  extends SyncRepository[ID, E] with Helpers{

  def store(entity: E)(implicit ctx: Ctx): Try[Result] = Try {
    implicit val session = getDBSession(ctx)

    if (sqlProvider.update(convertToMap(entity)).executeUpdate == 0)
      sqlProvider.insert(convertToMap(entity)).execute

    SyncResultWithEntity[This, ID, E](this.asInstanceOf[This], entity)
  }

  def resolveBy(identifier: ID)(implicit ctx: Ctx): Try[E] = Try {
    implicit val session = getDBSession(ctx)
    val result = sqlProvider.select(identifier).single()
    convertToEntity(result)
  }
}

/**
 * PostingRepositoryOnAnorm
 * resolveAll(): 全てのポスティングを取得するメソッド
 */
class PostingRepositoryOnAnorm extends AbstractRepositoryOnAnorm[PostingID, Posting] {

  def resolveAll()(implicit ctx: Ctx): Try[List[Posting]] = Try {
    implicit val session = getDBSession(ctx)
    val result = sqlProvider.selectAll()
    result().map(convertToEntity).toList
  }
}
```

```
}

/**
 * LoginUserRepositoryOnAnorm
 */
class LoginUserRepositoryOnAnorm extends AbstractRepositoryOnAnorm[LoginUserEmail, LoginUser]
```

レイヤードアーキテクチャで処理をそれぞれの層に分けたことで、ドメインが見やすくなりました。また、処理内容がおおまかに把握できるので実装がスムーズに進みました。

## まとめ

ドメイン駆動設計を用いた掲示板研修では、ユビキタス言語の定義からドメイン図の作成までの手順を行いました。

その結果、開発段階で出戻りがなくなり実装がスムーズに進みました。

## 新卒研修から学んだこと

ここまでの研修で言語・開発・設計の3点について学んできました。

Scalaのレクチャーでは、トレイトやオブジェクトなどオブジェクト指向についてのレクチャーが多いため、ざっくりとオブジェクト指向の概要を掴んでおけば、レクチャーの内容がより充実すると思いました。

私はオブジェクト指向の知識がなかったので「オブジェクト脳の作り方」を読んでオブジェクト指向の概要を覚えました。

1度目の掲示板研修では一つのメソッドに複数の実装した結果、単体テストを書く上でものすごく苦労しました。

単体テストしづらいコードを書かないために、各メソッドの目的を明確にして実装することが大事だと学びました。

各メソッドの目的を明確に書くことができれば、内容も理解しやすくなり、テストの内容も明確になります。

DDDを使った掲示板研修ではストーリーの改変とユビキタス言語の定義のところで詰まってしまいました。

ストーリーには曖昧なところがあった場合、曖昧なところを明示することで、設計のイメージがよりわかりやすくなりました。

私は曖昧なところを明示的にするために、目的について考えることを徹底しました。

目的について考えることで、曖昧なストーリーの場合POに質問ができ、ストーリーの曖昧な部分を明示的にできました。

## まとめ

私は研修の後、業務にジョインしました。

新卒研修で学んだことは業務にも生かされています。

大量の広告データを扱う上で、Scalaの豊富なコレクションメソッドはとても役に立っています。

実装するにあたって、メソッドの目的を明確にすることで、実装内容も理解しやすくテストが容易なコードを書くことができました。

業務のストーリーの目的を考えることで、実装に必要なものが理解できるようになり、無駄な実装を抑えることができるようになりました。

その結果、スムーズに実装を行えるようになりました。

私はこの3ヶ月の研修を通して言語・開発・設計を身につけることができ、プロScalaエンジニアになることができました。

弊社では一律で品質面とスピード面の両立するためにこのような教育プログラムを組んで、Scalaに取り組んでいます。

ぜひ、Scalaに興味があって品質もスピードもよりよいものを追求したい方はぜひともお越しください。

《早瀬峻介》

# GANMA!でのCache実装例

こんにちは、杉谷です。

GANMA!は負荷対策として、Memcacheを利用したオーソドックスなCache機構をDDDレイヤードアーキテクチャとDI機構にうまく組み込み、富豪的な実装をやっても勝手に高速になるような機構を開発しました。実装的には大変原始的なのですが、使い勝手は悪くないため、本セクションではこれの詳細をご紹介します。どなたかのご参考になれば幸いです。

## Cache方針

Cache機構は

- 使う側はCache制御を気にしなくて良い（明示的なCacheクリアは不要、等）
- 簡単に使える
- 富豪的に呼んで良い

といった性質を備え、透過的に使えるのが理想です。全てを一つで実現するのは難しいので、GANMA!では特性の違う複数のCache機構を作って使い分ける実装にしました。現時点では以下の3機構が存在します。

- インフラ層<sup>\*</sup>Cacheと呼んでいる物 《副作用ほぼ無し・結構早い》
- インスタンスCacheと呼んでいる物 《副作用すこし有り・もうちょい早い》
- アプリ層<sup>\*</sup>Cacheと呼んでいる物 《副作用大・ウルトラ早い》

個別に紹介していきます。

<sup>\*</sup>GANMA!はドメイン駆動設計(DDD)・レイヤードアーキテクチャを採用しています。DDDをご存じない方は"インフラ層 ≡ Model"、"アプリケーション層 ≡ Controller"のようなものとして適宜読み替えてください。

## インフラ層Cache 《副作用無し》

DBアクセスクラスなど、インフラ層の存在に実装したCache機構です。

- 読み取り系のMethodはCacheを確認し、あればそれを取得。なければ実データを取得しCache更新。
- 更新系はデータの更新をした後Cacheの消去。

を行う原始的な実装です。複数スレッド・複数機材で同一オブジェクトを同時更新する可能性が無ければ平和に動作します。

## 実装イメージ

```
// テーブル読み書き
class CatsTable {
    def get(catName: String): Option[Row] = ??? //※1
```

```
def list(offset: Int = 0): Seq[Row] = ???

def store(cat: RawData): Unit = ???

// ~
}

// Cache読み書き
object Cache {

  def makeKey(tag: String, args: Any*): String = ???

  def getOrElseUpdate[A: ClassTag](key: String, op: => A, expire: Int = 300): A = ???

  def clear(key: String): Unit = ???

  // ~
}

// Cache付きテーブル読み書き
class CachedCatsTable extends CatsTable {

  private val className = getClass.getName

  override def get(catName: String): Option[Row] = {
    val key = Cache.makeKey(className + ".get", catName)
    Cache.getOrElseUpdate(key, super.get(catName))
  }

  override def list(offset: Int = 0): Seq[Row] = {
    offset match {
      case 0 => // ※2
        val key = Cache.makeKey(className + ".list")
        Cache.getOrElseUpdate(key, super.list(offset))

      case _ =>
        super.list(offset)
    }
  }

  override def store(cat: RawData) = {
    super.store(cat)
    clearCache(cat)
  }

  private def clearCache(cat: RawData): Unit = { // ※3
    Cache.makeKey(className + ".get", cat.name)
    Cache.makeKey(className + ".list")
  }
}
```



- ※1 - 実際に組むときはFutureで返しましょう。
- ※2 - 引数の組み合わせが膨大になる物に関しては、offset=0のときだけ、などよく使われるものだけに限定してCacheすると結構効率よいです。
- ※3 - この消す処理が膨らむ場合はアプローチを変える必要があります。Memcacheの代わりにRedisを使えば、Hash型が使えるので、もうすこし融通が利くようになります。

## インスタンスCache 《場合によっては気をつける必要あり》

DDDで実際の実装ではEntityIDだけを引き回し、実データは都度Repositoryから取得する。という場面が多くでてきます。何も対策をしないと同一オブジェクトをインフラ層Cacheから何度も取得する事になりますが、データ量が多くなると遅くなります。

GANMA!では、一度取得したオブジェクトは同一リクエストの間再利用する機構を作り、オブジェクトの再取得を抑制しています。具体的な仕組みは以下の通りです。

1. DI機構をつくる (GANMA!では <http://qiita.com/takezoux2@github/items/a2b607cdfedd21974687> の記事を参考にさせていただいたものを使っています)
2. DI機構は1HTTPリクエスト毎に必要なリポジトリをインスタンス化する
3. リポジトリは一度取得したデータを忘れない。ただし自分が更新系の処理を行ったらCacheを忘れる。
4. リクエストが終わったら全部開放

## 実装イメージ

### インスタンスCacheの実装例

```
class InstanceCachedCatsRepository extends CatsRepository {  
  
  import scala.collection._  
  import scala.collection.convert.decorateAsScala._  
  
  private object cache {  
  
    val get: concurrent.Map[String, Option[Cat]] =  
      new ConcurrentHashMap[String, Option[Cat]]().asScala  
    val list: concurrent.Map[Int, List[Cat]] =  
      new ConcurrentHashMap[Int, List[Cat]].asScala  
  
    def clear(): Unit = {  
      get.clear()  
      list.clear()  
    }  
  }  
  
  override def get(catName: String): Option[Cat] =  
    cache.get.getOrElseUpdate(catName, super.get(catName))  
  
  override def list(offset: Int = 0): List[Cat] =  
    cache.list.getOrElseUpdate(offset, super.list(offset))  
  
  override def store(cat: Cat): Unit = {  
    super.store(cat)  
    cache.clear()  
  }  
}
```

## DIとコントローラーの実装例

```
trait CatMomiatureServiceDepends {  
  implicit lazy val catRepository = new InstanceCachedCatsRepository  
}  
  
class DomainInjector extends CatMomiatureServiceDepends  
  
class CatMomiatureService(implicit depends: CatMomiatureServiceDepends) {  
  
  import depends._  
  
  def momi(catName: String): Unit = {  
    catRepository.get(catName) foreach doMomiature  
  }  
  
  private def doMomiature(cat: Cat) = ???  
  
}  
  
class CatsController(implicit injector: DomainInjector =  
  new DomainInjector) extends Controller {  
  
  import injector._  
  
  def momi(catName:String) = Action {  
  
    catRepository.get(catName) foreach doSomething  
  
    val service = new CatMomiatureService  
    service.momi(catName)  
  
    Ok("やったぜ")  
  }  
}
```

インスタンスキャッシュはサーバによる更新が入る前提の処理には向かないため、ドメイン層として実装し、アプリ層の都合によってCacheの有無を選べるようにしています。

## アプリケーション層Cache 《副作用大》

トップページなど誰が見ても内容が固定で、高速に値を返す価値があるものであればアプリケーション層だけでレスポンス自体(htmlとかjson)のキャッシュを検討する価値があります。

応答速度は最大化されますが、管理ツールなど他のサーバによる更新が合った場合、何も対策しなければ古い情報が表示されてしまう等の問題が発生しますので、対策が許容できる箇所にだけピンポイントで導入します。

実装はMapなどに突っ込んでしまうだけですが、スレッドセーフである必要がある場合が多いので注意が必要です。Googleが開発しているライブラリ[Guava](#)のCacheBuilderを使うと、Expire機能付きMapなどが簡単に作れて便利です。

## 終わりに

ご紹介は以上です。如何でしたでしょうか？ GANMA!では開発者を募集しています。楽しい職場ですので、ご興味有りましたら是非ともお問い合わせください！

読んでくださいましてありがとうございました。《杉谷保幸》

# Play Framework入門者向けプロジェクト分割

こんにちは、廣幡と申します。

私はまだ社会人2年目ですが、新規プロジェクトの立ち上げや既存プロジェクトのリビルドなど、非常に刺激的な経験をさせていただいております。

新規プロジェクトの立ち上げではバックエンドをレイヤードアーキテクチャで設計し、各レイヤ毎にプロジェクトを分割したので今回はその方法をご紹介します。

これからPlayScala+DDDで開発する方などのご参考になれば幸いです。

## レイヤードアーキテクチャとは

レイヤードアーキテクチャとは、複雑なプログラムを複数レイヤに分割し、各レイヤで設計を進めることです。基本的には、ユーザーインターフェース、アプリケーション、ドメイン、インフラストラクチャの4つに分類され、下位のレイヤは上位のレイヤに対して疎結合、つまり上位層は下位層のみに依存するようであればなりません。そうすることで、各レイヤ（特にドメイン層）は本来の責務に集中できるようになります。

## 方針

今回作成するプロジェクトはレイヤードアーキテクチャで設計するため、以下の様な構造に分割するのが目的です。

```
projectRoot
├─ application
├─ domain
└─ infrastructure
```

あれ？ユーザインターフェース層は分離しないの？と思う方もいらっしゃるかもしれません。

今回作成するプロジェクトでは、アプリケーション層はPlayアプリケーションとなっており、ControllerとViewが分かれていますのでそれで妥協します。妥協せずにやるのであれば、アプリケーション層はただJSONを返すだけのRESTAPIサーバとして実装し、UI層をPlayのViewではなく別で実装する方法が良いと思います。そうすることによって、Android、iOS、WebなどのUIを別々に作ることができ、ドメインを使いまわせることになります。

また、

- アプリケーション層はPlayアプリケーションでいいけど、ドメイン層・インフラストラクチャ層はPlayに依存させたくない・・・
- 上層から下層は呼び出せるけど、下層から上層は呼び出せないようにしたい・・・

とレイヤ間での依存関係を制限するため、以下の方法で実現しました。

## プロジェクト分割

### build.sbt

```
/**
 * アプリケーション層、ドメイン層、インフラストラクチャ層の集合
 */
lazy val root = (
  project in file(".")
).aggregate(
  application,
  domain,
  infrastructure
)

/**
 * アプリケーション層
 *
 * ドメイン層・インフラ層に依存、Playアプリケーション
 */
lazy val application = Project(
  id = "application",
  base = file("application")
).dependsOn(
  domain,
  infrastructure
).enablePlugins(
  PlayScala
)

/**
 * ドメイン層
 *
 * インフラストラクチャ層に依存
 */
lazy val domain = Project(
  id = "domain",
  base = file("domain")
).dependsOn(
  infrastructure
).settings(
  scalaSource in Compile := baseDirectory.value / "src" / "main" / "scala",
  scalaSource in Test := baseDirectory.value / "src" / "test" / "scala"
)

/**
 * インフラストラクチャ層
 */
lazy val infrastructure = Project(
```

```

    id = "infrastructure",
    base = file("infrastructure")
  ).settings(
    scalaSource in Compile := baseDirectory.value / "src" / "main" / "scala",
    scalaSource in Test := baseDirectory.value / "src" / "test" / "scala"
  )

```

これでビルドしてみると以下の様な構造になります。

```

projectRoot
├─ application
│   ├─ app/
│   ├─ conf/
│   ├─ public/
│   └─ target/
├─ domain
│   ├─ src/
│   └─ target/
├─ infrastructure
│   ├─ src/
│   └─ target/
├─ project/
├─ target/
├─ activator
└─ build.sbt

```

これで今回作成したかったものを実現できました。

実行方法は以下の通りです。

```
$ ./activator "project application" run # ./activator "application/run" でも可
```

上記は、下記と同じです。

```

$ ./activator
[projectRoot] project application
[application] run

```

## おまけ

ここまで説明したものとは少し違う、以下のような考え方もあります。

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.(上位のモジュールは、下位のモジュールに依存してはならない。どちらのモジュールも「抽象」に依存すべきである。)
2. Abstractions should not depend on details. Details should depend on abstractions.(「抽象」は実装の詳細に依存してはならない。実装の詳細が「抽象」に依存すべきである。)

これは、DIP(Dependency Inversion Principle)、すなわち依存関係逆転の原則と呼ばれるものです。

このDIPを実現するために有効だとされているアーキテクチャの1つに、ヘキサゴナルアーキテクチャと呼ばれるものがあります。そちらについては今回説明しませんが、DDDにおいてかなり効果的なので気になる方は是非調べてみてください。

## 最後に

ご紹介は以上です。長々と書いてしまいましたが、Playでプロジェクトの分割、ついでに依存関係の強制の方法についてご紹介させていただきました。

ScalaやPlayFrameworkに興味がある方、DDDでの設計に興味がある方などのご参考になれば幸いです。

読んでくださり、ありがとうございました。《廣幡 俊樹》



## ヘキサゴナルアーキテクチャを採用したプロジェクトの実装例

こんにちは、湯浅と申します。

今関わっているプロジェクトでヘキサゴナルアーキテクチャを採用しているのですが、今回はその実装例を簡単なサンプルでご参考までに紹介したいと思います。

### ヘキサゴナルアーキテクチャとは

ヘキサゴナルアーキテクチャとは別名「ポートとアダプター(Ports & Adapters)」とも呼ばれるアーキテクチャで、システムを内部と外部の2つの領域に分け、システムのコアである内部を**アプリケーション**、外部を**ポート**と大きく2つの層に分離することを特徴としています。

詳しくは下記、考案者公式ページ参照

#### Hexagonal architecture

<http://alistair.cockburn.us/Hexagonal+architecture>

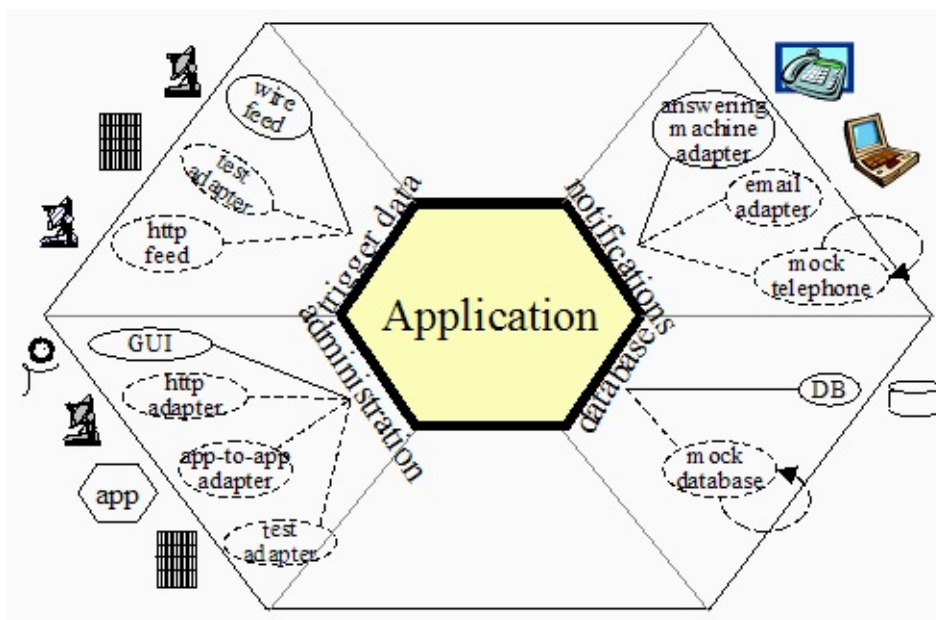


図1 気象情報通知システムの例 (上記公式ページから引用)

(例のシステムでは4つのポートがあり、それぞれのポートの中に複数のアダプターがある)

## 今回のサンプルシステム

今回は簡略化のため下記のようなポートを2だけ持ったREST APIベースのシンプルなWEBアプリの開発を考えます。

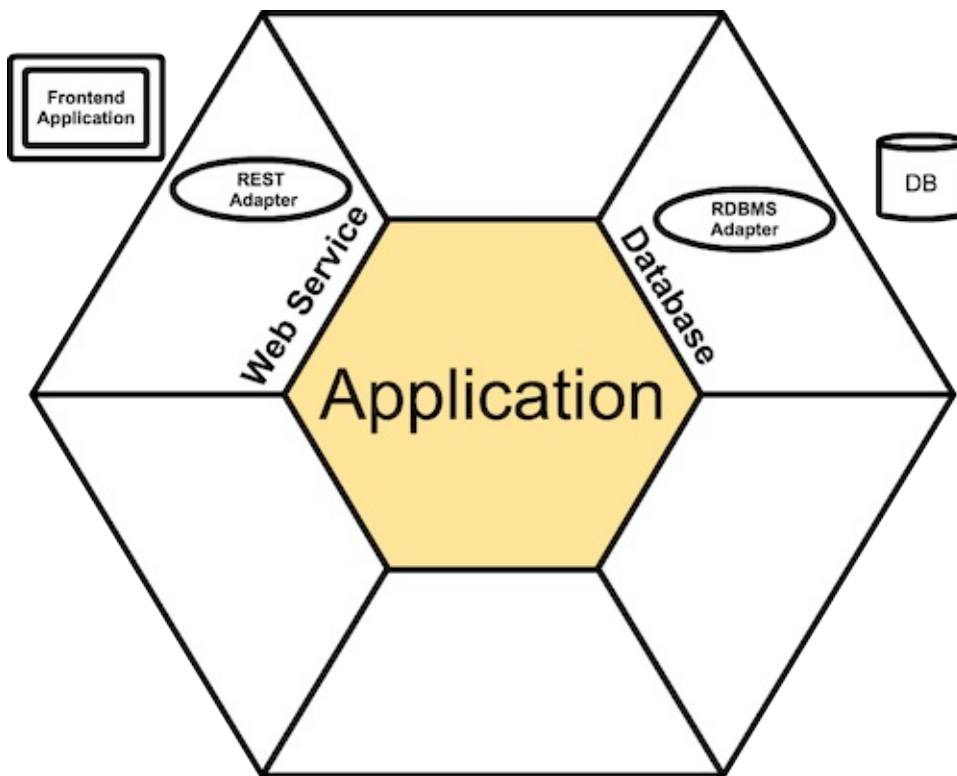


図2 サンプルのシステム

以降の実装イメージでは Web ServiceポートはPlay framework 2.4で実装、Databaseポートは具体的な実装は省略する形で解説していきます。

また今回はプロジェクト構成等の実装イメージを解説することを主目的としているため、実装しているコード自体は大幅に簡略化したものとなっています。

## プロジェクト構成

まずSBTのプロジェクト構成ですがアプリケーションと各ポートの依存関係を強制するため、それぞれを個別のSBTプロジェクトとして分割していきます。

また、各ポート自体も他のポートへの依存を排除しているため、今回のサンプルでは1番上の階層にそれぞれのポートへ依存した**root**というプロジェクトをシステムの起動用に配置しました。

この**root**プロジェクト内で各ポートで実装したクラスをインスタンス化しシステムを起動できるようにしています。

build.sbtの実装イメージ

```
name := "hexagonal_sample"

lazy val commonSettings = Seq(
  scalaVersion := "2.11.7"
)

lazy val root =
  (project in file("root"))
    .enablePlugins(PlayScala)
    .dependsOn(
      portWebService,
      portDatabase
    )
    .settings(commonSettings)
    .settings(
      routesGenerator := InjectedRoutesGenerator
    )

lazy val portWebService =
  Project(id = "port-webservice", base = file("port/webservice"))
    .dependsOn(application)
    .settings(commonSettings)
    .settings(/* Port固有の設定 */)

lazy val portDatabase =
  Project(id = "port-database", base = file("port/database"))
    .dependsOn(application)
    .settings(commonSettings)
    .settings(/* Port固有の設定 */)

lazy val application =
  Project(id = "application", base = file("application"))
    .enablePlugins(PlayScala)
    .settings(commonSettings)
    .settings(/* Application固有の設定 */)
```

## アプリケーションの実装イメージ

ここにはシステムのコアとなるエンティティ、リポジトリ、サービスなどを実装していきます。リポジトリやサービスが外部に依存する場合、インターフェースだけここに定義し、具象クラスは各ポートに実装されることになります。

```
package sample.application.user

case class UserId(value: Long)

class User(
  val id: UserId,
  val name: String,
  val password: String
)

trait UserRepository {

  def resolveBy(id: UserId): Option[User]

  def store(user: User): Unit
}

class UserRegistrationService(userRepository: UserRepository) {

  def register(name: String, password: String): User = {
    val user = createNewUser(name, password)
    userRepository.store(user)
    user
  }

  private def createNewUser(name: String, password: String): User = {
    // 新規ユーザ作成のために必要な何らかの処理
    ???
  }
}
```

## Databaseポートの実装イメージ

ここには先程インターフェースだけ定義したりポジトリの具象クラスを実装していきます。

```
package sample.port.database.rdbmsadapter.user

import sample.application.user.{UserRepository, User, UserId}

class UserRepositoryOnRDBMS extends UserRepository {

  override def resolveBy(id: UserId): Option[User] = {
    // DBライブラリ等に依存した参照処理
    ???
  }

  override def store(user: User): Unit = {
    // DBライブラリ等に依存した保存処理
    ???
  }
}
```

## Web Serviceポートの実装イメージ

今回はREST APIを提供するシステムを仮定しているので、このポートにはアプリケーションのインターフェースにだけ依存する形でコントローラ等の実装をしていきます。

```
package sample.port.webservice.restadapter.user

import com.google.inject.Inject
import play.api.data.Forms
import play.api.data.Form
import play.api.libs.json._
import play.api.mvc._
import sample.application.user._

case class UserDTO(id: Long, name: String)

object UserDTO {

  def fromModel(user: User): UserDTO = UserDTO(user.id.value, user.name)

  implicit val jsonWrites = Json.writes[UserDTO]
}

case class UsercreateForm(name: String, password: String)

object UsercreateForm {
```

```
val form: Form[UserCreateForm] = Form(
  Forms.mapping(
    "name" -> Forms.nonEmptyText,
    "password" -> Forms.nonEmptyText
  )(apply)(unapply)
)
}

class UserController @Inject()(
  userRepository: UserRepository,
  userRegistrationService: UserRegistrationService
) extends Controller {

  def get(id: Long) = Action {
    val user: Option[User] = userRepository.resolveBy(UserId(id))
    user.fold[Result](
      NotFound
    ) { u =>
      Ok(Json.toJson(UserDTO.fromModel(u)))
    }
  }

  def create = Action { implicit request =>
    UserCreateForm.form.bindFromRequest().fold(
      errors => BadRequest,
      form => {
        val user = userRegistrationService.register(form.name, form.password)
        Ok(Json.toJson(UserDTO.fromModel(user)))
      }
    )
  }
}
```

## Rootプロジェクトの実装イメージ

ここにはシステムの起動に必要な設定ファイルや、各ポートで実装したクラスのインスタンス化をするための最低限の実装だけを置きます。

インジェクション用の実装

```
package sample

import com.google.inject.AbstractModule
import sample.application.user.{UserRegistrationService, UserRepository}
import sample.port.database.rdbmsadapter.user.UserRepositoryOnRDBMS

class SampleInjector extends AbstractModule {

  lazy val userRepository: UserRepository =
    new UserRepositoryOnRDBMS()

  lazy val userRegistrationService: UserRegistrationService =
    new UserRegistrationService(userRepository)

  override def configure(): Unit = {
    bind(classOf[UserRepository])
      .toInstance(userRepository)

    bind(classOf[UserRegistrationService])
      .toInstance(userRegistrationService)
  }
}
```

application.confに上記のクラスを追加

```
play.modules.enabled += "sample.SampleInjector"
```

routesファイル

```
GET    /users/:id @sample.port.webservice.restadapter.user.UserController.get(id: Long)
POST   /users     @sample.port.webservice.restadapter.user.UserController.create
```

## おわりに

簡単ではありますが、以上のような構成で最低限システムが起動するようなベースができました。今回はシンプルなREST APIベースのWEBアプリをサンプルとして、我々のプロジェクトでどのようにしてヘキサゴナルアーキテクチャを実装をしたのかをご紹介します。実際のシステムでは今回のサンプルのようにポートの数が2つのシンプルなものになることは少ないかと思いますが、その場合でも基本は今回と同じような形でポートやアダプターの追加をしていけるはずです。

ヘキサゴナルアーキテクチャは具体的な実装をどのようにするのかまでは規定されていないため、今回のサンプルは実装のほんの一例とはなりますがご参考になれば幸いです。

《湯浅孝司》



# Akka Schedulerの定期実行を使って自動的にInstagramから画像を取得し、Tumblrへ投稿

こんにちは！村井と申します。

見出しの通りですが、Akka Schedulerの定期実行を使ってInstagramから画像を取得し、Tumblrへ自動投稿をしたいと思います。

## Akkaの設定

### Akkaとは

そもそもAkkaですが、Scala／Javaで非同期処理を実現するためのライブラリで、Scalaの開発元であるTypesafe社が中心となって開発しています。Akkaは軽量な並列処理のためのActorモデルをベースとしており、分散／並列／耐障害性を持ったイベント駆動型アプリケーションを構築できます。

akka-quartz-schedulerというライブラリを使用すれば、cronのような記述で指定時間に処理を実行することが出来ます。今回、playframeworkにこのライブラリを入れて、自動的にブログ（Tumblr）にInsta画像を貼っていきたいと思います。

#### 1. build.sbtに設定

```
libraryDependencies += "com.enragedginger" %% "akka-quartz-scheduler" % "1.4.0-akka-2.3.x"
```

#### 2. application.confに時間を設定

```
akka {
  quartz {
    defaultTimezone = "Asia/Tokyo"
    schedules {
      AutomationBlog {
        description = "3時間ごとに実行"
        expression = "0 0 0-23/3 ? * *"
      }
    }
  }
}
```

そんなに頻繁に動くものを考えていないので、3時間に1回のスケジュールにしています。設定方法はほぼcronと一緒ですが、以下のリンクを参考にどうぞ。

※ <http://quartz-scheduler.org/api/2.1.7/org/quartz/CronExpression.html>

### 3. Actorを用意

簡単に言えば、定期的に行いたい処理の部分ですね。

今回はInstagramから画像を取得して、Tumblrに投稿することがそれに当たります。

```
class GetInstagramImageCreateTumblr extends Actor {  
  
  def receive = {  
    case msg: String =>  
      val instaImgUrl = GetHashTagImage  
      val tumblrResponse = CreateBlog(instaImgUrl)  
  }  
}
```

長ったらしいファイル名にしていますが、スルーでw

GetHashTagImageはInstagramから画像を取得しており、

CreateBlog(instaImgUrl)で画像URLをTumblrに投稿しています。

### 4. Globalオブジェクトを用意

```
object Global extends GlobalSettings {  
  
  val system = ActorSystem("SampleSystem")  
  val actor = system.actorOf(Props(classOf[GetInstagramImageCreateTumblr]))  
  
  override def onStart(app: Application) = {  
    QuartzSchedulerExtension(system).schedule("AutomationBlog", actor, "")  
  }  
  
  override def onStop(app: Application) = {  
    system.shutdown()  
  }  
}
```

GlobalオブジェクトのonStartで定期実行の処理を開始させます。

**AutomationBlog**とありますが、これはapplication.confで指定したスケジュール名と一致させる必要があります。

これで設定は完了です。

activator runをすれば、GlobalのonStartでスケジューラーが動作します。

## InstagramAPI

### 1. アプリ登録とアクセストークンの取得

さて、Instagramから画像を取得します。

それにはまずAPIを叩きますが、そのためにアプリ登録してアクセストークンが必要になります。

その辺は説明が長くなりすぎるので、以下のリンクを参考にしてください。

※ アプリ登録：<https://syncer.jp/instagram-api-matome#sec-1>

※ トークン取得：<https://syncer.jp/instagram-api-matome#sec-2>

## 2. 画像の取得

Instagramからの画像取得は指定したハッシュタグ画像を1つ取得しようと思います。

今回使用したAPIは以下のリンクにあるものです。

※ [https://www.instagram.com/developer/endpoints/tags/#get\\_tags\\_media\\_recent](https://www.instagram.com/developer/endpoints/tags/#get_tags_media_recent)

```
private def GetHashTagImage = {  
  // 取得したいハッシュタグを指定  
  val hashTag = "cat"  
  
  // 画像取得用のAPI(countで取得したい数を指定)  
  val apiUrl = "https://api.instagram.com/v1/" +  
    "tags/" + hashTag + "/media/recent?count=1&access_token=" +  
    "*****"  
  
  // WSを使用してリクエストを投げてjsonで受け取る  
  val apiResult = WS.url(apiUrl).get().map {  
    response => response.json  
  }  
  val json = Await.result(apiResult, Duration.Inf)  
  
  // jsonツリーを解析して画像URLだけ取得  
  val imageUrl = json \ "data" \ "low_resolution" map (_ \ "url")  
  val imageUrlStr = imageUrl.map { a =>  
    a.get.toString()  
  }  
  // ダブルクォーテーションが含まれてしまうので置換で削除しておく  
  imageUrlStr.head.replace("\"", "")  
}
```

一部のみですが、Instaからは以下ような感じでjsonが返ってきます。

今回は画像URLさえあればいいので、imagesのlow\_resolutionのurlを取得しています。

```
"images": {
  "low_resolution": {
    "height": 320,
    "url": "https://scontent.cdninstagram.com/**.jpg",
    "width": 320
  },
  "standard_resolution": {
    "height": 640,
    "url": "https://scontent.cdninstagram.com/**.jpg",
    "width": 640
  },
  "thumbnail": {
    "height": 150,
    "url": "https://scontent.cdninstagram.com/**.jpg",
    "width": 150
  }
},
```

## TumblrAPI

Instagramから画像は取得できました。  
あとはTumblrに投稿するだけです。

### 1. アプリ登録

Instagram同様アプリ登録が必要です。

※ <https://syncer.jp/tumblr-api-matome#sec-1>

### 2. OAuth認証

今回使用するTumblrAPIは以下のURLにあるPhoto Postsです。

投稿はGETメソッドではなくPOSTですね。OAuth認証が必要です。

※ <https://www.tumblr.com/docs/en/api/v2#posting>

### 3. OAuth token取得

TumblrはInstagramやTwitterみたいにブラウザなどからtokenを確認する方法はなく、token取得用プログラムを叩いて取得してね。となってるそうです。

めんどくさいことこの上ないです。

取得用プログラムを公開してる方もいたのですが、API仕様が変わったようで使えなくなっていました。

ですが！

Tumblrのにログインしてみると、サンプルプログラムが書かれている箇所にtokenが載っています！

なので、これをコピーして使います。

### 4. Jumblr

Api consoleのサンプルプログラムの中に、Javaがあります。

JavaではTumblrのOAuth認証のために**Jumblr**が提供されています。

このライブラリを使用せずにscalaだけでやっちゃおうかと思ったのですが、せっかくなのでJumblrを使用することにしました。

### 5. build.sbtにJumblrを設定

```
libraryDependencies += "com.tumblr" % "jumblr" % "0.0.11"
```

## 1. 投稿する

```
private def CreateBlog(imageUrl: String) = {  
  val consumerKey = "*****"  
  val consumerSecret = "*****"  
  
  val client: JumblrClient = new JumblrClient(  
    consumerKey,  
    consumerSecret  
  )  
  client.setToken(  
    // oauthToken  
    "*****",  
    // oauthTokenSecret  
    "*****"  
  )  
  
  // postCreateのparamはjava.util.Mapなので、scalaのMapではなくこちらを使用する  
  val params: java.util.Map[String, String] = new HashMap[String, String]()  
  params.put("type", "photo")  
  params.put("source", imageUrl)  
  client.postCreate("instagramtagimage.tumblr.com", params)  
}
```

Jumblrのおかげでめっちゃ簡単ですね。

imageUrlはInstaから取得した画像URLを渡しています。

consumerKey

consumerSecret

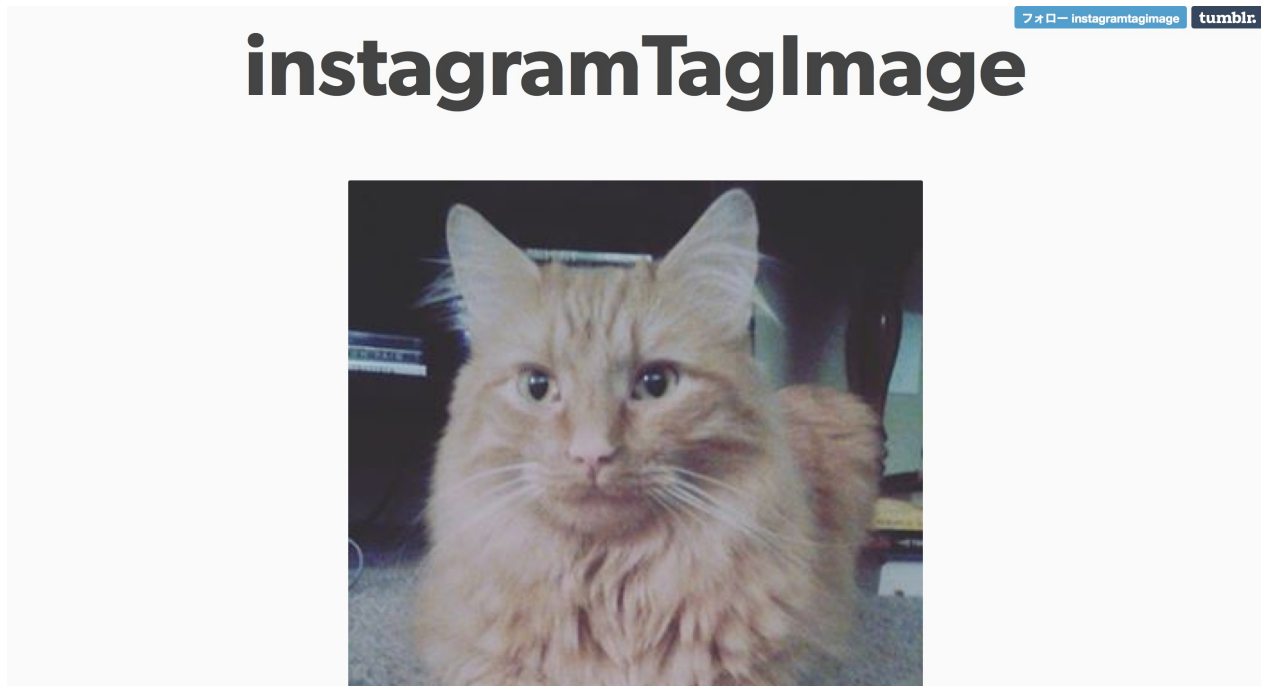
oauthToken

oauthTokenSecret

は4に記載している通り、TumblrのApi consoleに記載されているのでコピペしてください。

## 完成

Tumblrアプリ登録時に作っておいたブログを見てみましょう。



ちゃんと投稿されてますね。

## まとめ

Akka schedulerを使わなくてもcronでもいいやん。って思われますが、プログラム含め、Playひとつで完結出来るんで管理もしやすいですね。

他にもTwitterに自動的につぶやいたり、画像をクローリングして、Pinterestにピンしておいたり、など色々出来ますね。もちろん、SNS以外にも使えますw

長くなりましたが以上です。  
最後まで読んでいただきありがとうございました。  
《村井》

## scala DDD 依存性逆転の原則を採用してみた

### はじめに

こんにちわ、下村と申します。

GANMA!のプロジェクトで採用した 依存関係逆転の原則 on DDD の実装例を簡単な例を元に紹介させていただきます。

実践ドメイン駆動設計にも、レイヤードアーキテクチャやヘキサゴナルアーキテクチャと共に 依存性逆転の原則を採用する例などができます。

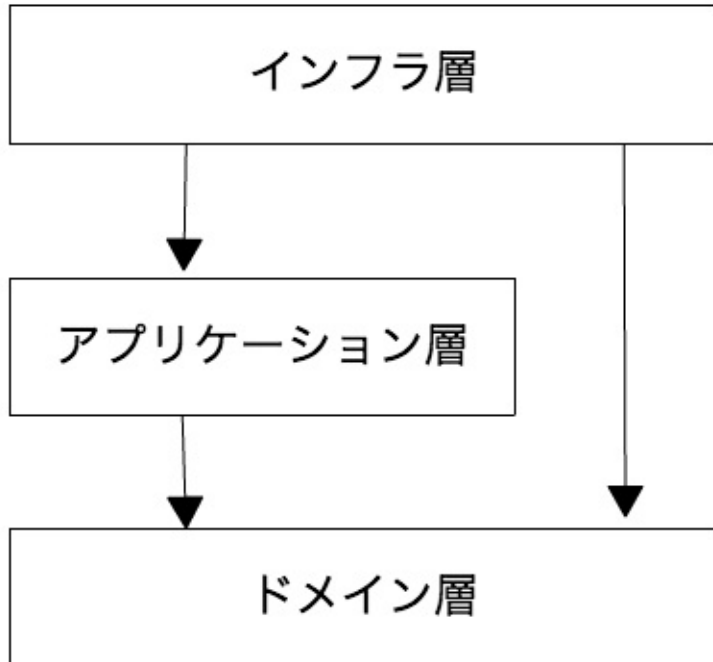
### 依存関係逆転の原則（DIP:the Dependency Inversion Principle）について

アジャイルソフトウェア開発の奥義に記載されている「クラスに関する5つの設計原則」の一つであり下記のように説明されています。

上位のモジュールは下位のモジュールに依存してはならない。どちらのモジュールも「抽象」に依存すべきである。

「抽象」は実装の詳細に依存してはならない。実装の詳細が「抽象」に依存すべきである。

抽象的で捉えにくいので、DDDでの実装にそって説明していきます。

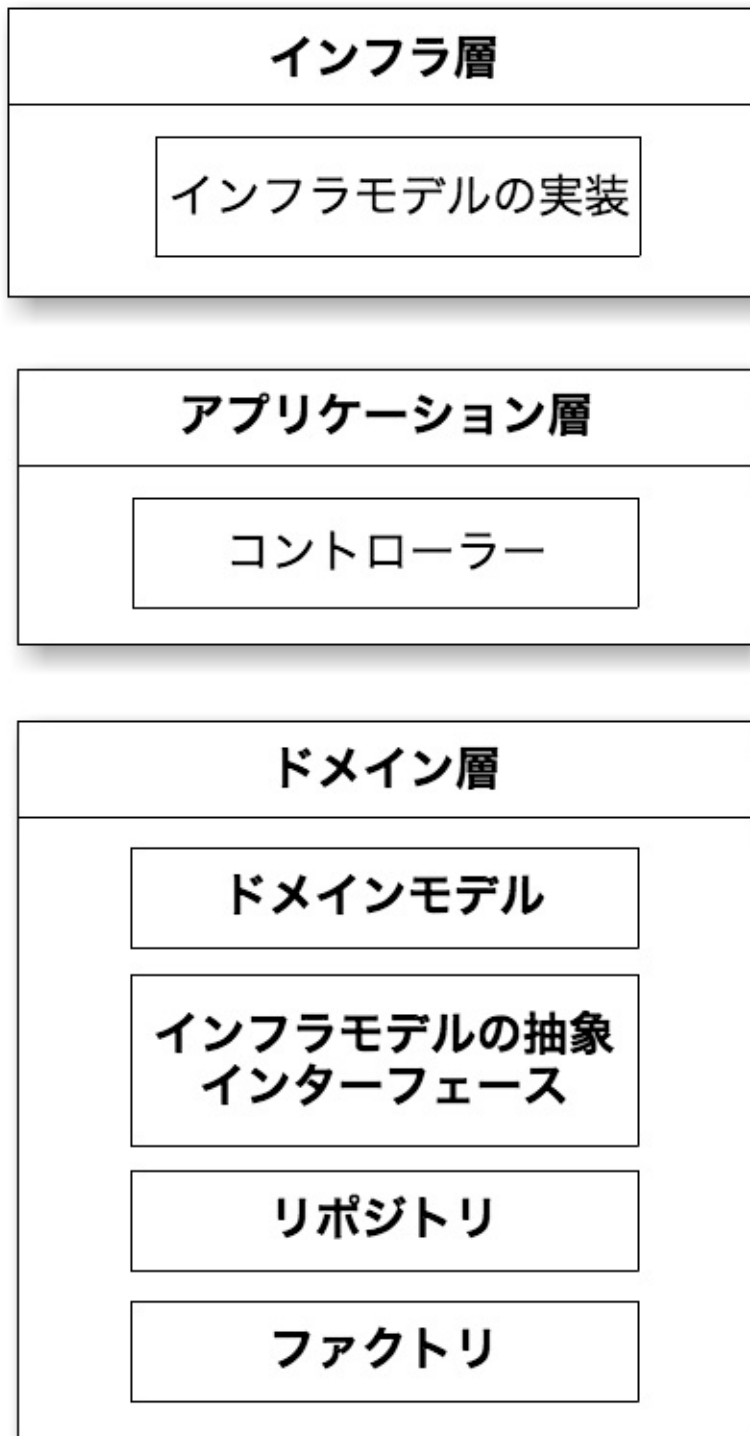


レイヤードアーキテクチャではドメインより下位に位置していたインフラ層を すべてのレイヤーの上位に位置させることで、下位のドメインモデルへの単方向参照を持ち上位のレイヤーが下位のレイヤーに依存することがなくなります。つまり、ドメインモデル最下位にすることで他のレイヤーに依存することを防ぐことができます。



## 実装例

GANMA!の Androidアプリで実装した例をもとに解説していきます。今回は実装の全体像を主眼としているので、実装は簡略化させていただきます。



## インフラモデルの抽象インターフェース

これまで下位のインフラモジュールに置いていたのを、DIPに則りインターフェースの所有権も逆転させ、ドメイン層におきます。

```
package com.sample.domain.magazin.trait

trait MagazineIdSource {
  val rawId: String
}

trait MagazineSource {
  val id: MagazineIdSource
  val title: String
}
```

## ドメインモデル実装

ここではドメインモデルの定義と上位からドメインモデルを取得するためのファクトリとなるリポジトリを実装します。

```
package com.sample.domain.magazine

import com.COMICSMART.GANMA.domain.magazine.traits.{MagazineIdSource, MagazineSource}
import com.sample.infra.magazine.MagazineAPI
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

case class MagazineId(rawid: String) extends MagazineIdSource

//ドメインモデルを抽象インターフェースに依存
case class Magazine(
  id: MagazineId,
  title: String
) extends MagazineSource

// Magazineモデルのファクトリ
object Magazine {
  def apply(src: MagazineSource): Magazine = {
    Magazine(
      MagazineId(src.id.rawId),
      src.title
    )
  }
}

class MagazineRepository(api: MagazineApi = MagazineAPI()) {

  def get(magazineId: MagazineId): Future[Magazine] =
    api.get(magazineId).map(Magazine(_))

}
```

## インフラ層の実装

インフラ層ではドメイン層で定義したインフラモデルの抽象インターフェースの実装を行います。

```
package com.sample.infra.magazine

import com.COMICSMART.GANMA.domain.magazine.traits.{MagazineIdSource, MagazineSource}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

class MagazineAPI() {

  def get(magazineId: MagazineIdSource): Future[MagazineSource] = {
    //http通信ライブラリ、json変換ライブラリに依存した抽象インターフェースの実装
  }
}
```

## アプリケーション層

GANMA!のAndroidアプリの実装を前提としているので、ここではコントローラーの実装と表示するためになどに 取得したデータをViewに渡して表示する等の実装をします。

```
package com.sample.application.magazine

import android.app.Activity
import android.os.Bundle
import com.sample.domain.magazine.{ MagazineId, Magazine }
import com.sample.domain.magazine.MagazineRepository
import scala.concurrent.ExecutionContext.Implicits.global

class MagazineActivity extends Activity {

  val magazineRepository = MagazineRepository()

  override def onCreate(savedInstanceState: Bundle): Unit = {
    super.onCreate(savedInstanceState)

    val magazineId = MagazineId(getIntent.getStringExtra("magazineId"))
    magazineRepository.get(magazineId).onSuccess {
      case m: Magazine => //取得したデータをviewモデルにセット
      case _ =>
    }

  }

  --- 省略 ---

}
```

## 終わりに

最後まで読んでいただきありがとうございました。DDDに依存性逆転の原則を取り入れることでドメインモデルが他のレイヤーに依存することを防ぐ例を紹介させていただきました。依存性逆転の原則を取り入れた一例となりますが、ご参考になれば幸いです。

### 参考文献

- ・ Vaughn Vernon (2015) 『実践ドメイン駆動設計 エリック・エヴァンズが確立した理論を実際の設計に応用する』 (高木正弘訳) 株式会社翔泳社
- ・ <http://d.hatena.ne.jp/asakichy/20090128/1233144989>
- ・ [http://www.atmarkit.co.jp/fdotnet/designptn/designptn07/designptn07\\_01.html](http://www.atmarkit.co.jp/fdotnet/designptn/designptn07/designptn07_01.html)

# 今すぐ使える！Android開発に役立つKotlinの拡張関数10選

湯上と申します。ゆのうえと読みます。10代女子向けファッションコーディネートアプリ MANTのAndroidアプリを担当しています。蔵書は技術本よりレシピ本が多い、作って食べれるエンジニアです。

2015年のAndroidの話題で個人的に最もホットだったのはKotlinでした。MANTでも早速プロジェクトに導入して楽しく開発を進めています。

さて、Kotlinの機能の中でも特徴的なものの一つに拡張関数があります。既存クラスにお手軽に関数を追加できるので、随所で儀式的な構文が必要とされるAndroidの開発に役立つ機能です。この記事では、そんなAndroid開発の手助けになりそうな10種類の拡張関数を紹介します。

## 1. setOnClickListenerをさらに短く

```
inline fun <T : View> T.onClick(crossinline f: (T) -> Unit) =
    setOnClickListener { f(this) }

// 使い方
button.onClick { /* do something */ }
```

## 2. カラーリソースの取得

```
fun Context.loadColor(@ColorRes resId: Int) = ContextCompat.getColor(this, resId)

// 使い方
view.setBackgroundColor(loadColor(R.color.orange))
```

## 3. ソフトウェアキーボードを非表示に

```
fun Activity.hideKeyboard() =
    (getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager)
        .hideSoftInputFromWindow(currentFocus.windowToken, 0)

// 使い方
hideKeyboard()
```

## 4. Fragmentのreplace

```
fun FragmentManager.replace(
    resId: Int = R.id.default_content,
    fragment: Fragment,
    tag: String? = null,
    stack: String? = null) {
    beginTransaction()
        .replace(resId, fragment, tag)
        .addToBackStack(stack)
        .commit()
}

// 使い方
supportFragmentManager.replace(fragment = MyFragment())
```

## 5. JSONObjectからキーと型を指定してオブジェクトを引き出す

```
fun <T: Any> JSONObject.pick(key: String): T? =
    try { get(key) as? T } catch (e: JSONException) { null }

// 使い方
val firstName = json.pick<String>("firstName")
```

## 6. LoaderCallbackの簡略化 initよりrestartのほうが使い勝手が良いかもしれません。

```
fun <T> LoaderManager.init(
    id: Int = 0,
    args: Bundle? = null,
    onCreate: (Int, Bundle?) -> Loader<T>,
    onLoadFinish: (Loader<T>?, T) -> Unit,
    onLoadReset: (Loader<T>?) -> Unit = { }) =
    initLoader(id, args, object : LoaderManager.LoaderCallbacks<T> {
        override fun onCreateLoader(id: Int, args: Bundle?): Loader<T>? = onCreate(id, args)
        override fun onLoadFinished(loader: Loader<T>?, data: T) = onLoadFinish(loader, data)
        override fun onLoadReset(loader: Loader<T>?) = onLoadReset(loader)
    })

// 使い方
supportLoaderManager.init<Cursor>(
    onCreate = { id, args -> createNewLoader() }
    onLoadFinish = { loader, c -> bindData(c) }
)
```

## 7. ScrollListenerの簡略化

```
fun <T: AbsListView> T.onScroll(
    onStateChange: (T?, Int) -> Unit = { view, scrollState -> }
    onScroll: (T?, Int, Int, Int) -> Unit = { view, f, v, t -> }) {
    setOnScrollListener(object: AbsListView.OnScrollListener {
        override fun onScrollStateChanged(view: AbsListView?, scrollState: Int) {
            onStateChange(view as T, scrollState)
        }

        override fun onScroll(view: AbsListView?,
            firstVisibleItem: Int, visibleItemCount: Int, totalItemCount: Int) {
            onScroll(view as T,
                firstVisibleItem, visibleItemCount, totalItemCount)
        }
    })
}

// 使い方
listView.onScroll(onStateChange = { view, scrollState -> /*do something*/ })
```

## 8. ImageViewに画像リソースを渡す

```
fun ImageView.setImage(source: Any?) {
    source?.javaClass?.name
    when (source) {
        null -> setImageBitmap(null)
        is Int -> setImageResource(source)
        is Bitmap -> setImageBitmap(source)
        is Drawable -> setImageDrawable(source)
        is Uri -> setImageURI(source)
        else ->
            throw IllegalArgumentException(
                "This source type of ${source.javaClass.name} is not supported")
    }
}

// 使い方
imageView1.setImage(bitmap)
imageView2.setImage(R.drawable.icon)
```

## 9. Animator.AnimationListenerを短く

```
fun ViewPropertyAnimator.listener(
    onStart: (Animator?) -> Unit = {},
    onCancel: (Animator?) -> Unit = {},
    onRepeat: (Animator?) -> Unit = {},
    onEnd: (Animator?) -> Unit = {}) {
    setListener(object : Animator.AnimationListener {
        override fun onAnimationStart(a: Animator?) = onStart(a)
        override fun onAnimationCancel(a: Animator?) = onCancel(a)
        override fun onAnimationRepeat(a: Animator?) = onRepeat(a)
        override fun onAnimationEnd(a: Animator?) = onEnd(a)
    })
}

// 使い方
view.animate().x(80f).listener(
    onStart = { view.alpha = 0.5f },
    onEnd = { view.alpha = 1.0f })
```

## 10. ViewTreeObserver.OnGlobalLayoutListenerを簡潔に

```
inline fun View.newGlobalLayoutListener(
    crossinline f: (ViewTreeObserver.OnGlobalLayoutListener) -> Unit):
    ViewTreeObserver.OnGlobalLayoutListener {
    return object: ViewTreeObserver.OnGlobalLayoutListener {
        override fun onGlobalLayout() = f(this)
    }.apply {
        viewTreeObserver.addOnGlobalLayoutListener(this)
    }
}

fun View.removeGlobalLayoutListener(
    listener: ViewTreeObserver.OnGlobalLayoutListener) =
    viewTreeObserver.addOnGlobalLayoutListener(listener)

// 使い方
view.newGlobalLayoutListener {
    textView.visibility = View.VISIBLE
    view.removeGlobalLayoutListener(it)
}
```

一気に紹介させていただきましたが、以上になります。拡張関数はコードを簡潔にする上で非常に強力なので、どんどん活用していきたいものです。こちらのサンプルが自分好みの拡張関数を作る手助けになれば幸いです。



## あとがき

当社ではいつでもScalaをやってみたい方を募集しております。この本を読んで、もし我々に興味をもっていただけなのであれば

<http://septeni-original.co.jp>

よりエントリーいただけると幸いです。

社内見学イベントも毎月行っています

<http://tour.septeni.net/>

まずは様子を見たい、と言うときはこちらもご検討ください。

## Septeni x Scala 勉強会

connpassにて定期的に情報共有を兼ねたScala勉強会を開催しています。

<http://septeni-scala.connpass.com/>

発表例

- 弊社がScalaに至るまでの物語 - 杉谷保幸
- 新卒で始めて学ぶ言語がScalaで良かったこと/大変だったこと - 寺坂郁也
- Scala × DDD × 弊社実践例 - 原田侑亮
- GANMA!でDDDをやってみてから1年くらい経った - 杉谷保幸
- DDD × 新人教育 - 原田侑亮

ご興味をもたれた場合、上記URLにてconnpassにご登録の上グループにご参加していただくと イベント予定日が決まりますと通知が届き便利です。

最後まで読んで戴きましてありがとうございました、心よりの御礼を申し上げます。

次号も発行できるとよいなとおもいます、またお会いしましょう！《セプテーニ・オリジナルー同》