ENGINEERS to ADVANCE the WORLD

世界を前に進めるエンジニア集団

We attract all users with advanced technology.

セプテーニ・オリジナル

技術読本

当社の開発サービス ①

GANMA!

論より証拠

ユーザー評価が示す技術力
アプリストア最高峰の
評価を獲得する漫画アプリ
GANMA!

当社の開発サービス ②

PYXIS

機能仕様よりも提供価値に拘る

インフィード広告市場を作り
変化の激しい市場を
今も牽引する広告システム
PYXIS

Septeni Original

©COMICSMART INC.

目次

はじめに	1.1
セプテーニ・オリジナル社とは？	1.2
ScalaでAndroidアプリ開発	1.3
単一責任の原則(SRP)についての見解と方法論	1.4
末尾再帰の呼び出し最適化の有無によるScalaコンパイラの挙動について	1.5
Akka HTTP で LINE bot を作ってみました	1.6
新卒scalaエンジニアが書くISUCONの歩き方	1.7
おわりに	1.8
付録：セプテーニ・オリジナル技術指針	1.9

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。

セプテーニ・オリジナルとは？

こんにちは、株式会社セプテーニ・オリジナルCTOの杉谷と申します。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。

"セプテーニ" という会社をご存じでしょうか？

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、そこそこ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テーニグループについてご紹介させていただきます。

セプテーニグループとセプテーニ・オリジナル

セプテーニグループはネット広告会社である株式会社セプテ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。

オフィスが結構立派で社内にカフェがあったりします。

我々セプテーニ・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開発が主な作業ですが、先述のGANMA!のように広告以外のシステムも担当しています。

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

我々セプテーニグループは、広告を出したい企業様からご予算を預かり、 企業様に変わって画像や動画やテキストを作り、様々な場所に広告を出稿する「広告代理業」が主な業務です。

広告代理業をスムーズに行うための様々なツールの集合体が「PYXIS」です。 FacebookやTwitterなど様々なサイトに対する広告運用やレポート業務を支援します。

Scalaを採用し、ドメイン駆動設計(DDD)で開発をしています。

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、作家様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）

こちらもScala + DDDで開発を行っているほか、 Android版もScalaで開発しています。（※iOS版はSwift）

Androidアプリをscalaで開発する方法については本書で詳しく解説しています。

アプリは400万DLを超え、単行本も50冊以上出版、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。

文化

気持ちよく高速で高品質な開発を行い続けられるよう、保守性を重視した開発体制を敷いています。

ほとんどのシステムでScalaとDDDで開発を進めています。 主要プロダクトのScala/DDD率は100%です。 レビューを行う、ユニットテストを整備する、などももちろん行っています。

チーム運営はスクラムが主な手段で、 なんちゃってスクラムにならないよう研修を受けたり、 スクラムマスター同士で情報共有を行い、健全にスクラムを回せているか確認できるような体制にしています。

これらをより高い水準の仕事を実現し、維持するために行っている二つの施策をご紹介します。

全レビュー

良い開発を行うためにコードレビューは欠かせないのですが、 経験の浅いチームや人数が少ないチームではレビューの質や、視点の豊富さに不安が残ります。 誰かの思い込みによる主張で良くない方向にコードが曲がってしまう、などもありがちですが避けたい現象です。

これらを防ぐため、チーム外からもレビューが行える仕組みを整えました。

我々はコード管理にAtlassian社Bitbucketを利用しているのですが、そのプラグインに グループ単位でレビュアーを自動で追加できるプラグインWorkzoneというものがあります。

これを利用して、社内の全てのレビューに参加するグループを作成、 視点が偏らないように社外レビュアーとして Scala・DDD伝道師の加藤潤一氏(@j5ik2o)と、ScalaMatsuri座長・Japan Scala Association常務理事・BONX INC所属である麻植泰輔氏(@OE_uia) に参加していただき、全てのコードをレビューしていただいています。

技術指針

上記全レビューを行う際、どういった基準で指摘を行うかを定めておかないと無用な混乱が発生するので、 会社としての品質に対する考え方、採用する技術やその理由と例外、などをまとめた技術指針を作成しました。

内容は以下のような物です

- 我々は"高速高品質"を目指す
- 我々は保守性のあるコードを高速に生産する
- Scalaを使う理由は気軽に品質を維持するため / Scalaを利用しない場合
- スクラムを採用する理由はプロダクトオーナーがハンドルを握るため / スクラム以外を利用する場合
- DDDを採用するのはチーム全員で要件の芯を捉え続けるため / DDDを採用しない場合
- ユニットテストを書く理由はリファクタリングをし続けるため
- テストを書く＝遅いという思い込みを止めよう / テストを書かない場合
- レビューを行うのは意識を保ち続けるため / レビューを行い続けるため属人性を高めない / レビューを行わない場合

本書の最後に付録として収録しています。

その他制度・環境

他にも

- 裁量労働制。早めに出てくる朝方の人もいればお昼頃から稼働開始する方もいる
- 主なコミュニケーション手段はSlack
- 全員がScala利用者なので、MacbookPro 15インチ(not touchId)のメモリ・CPU増強版が標準機
- マウスやキーボードは会社持ちで好きな物が買える
- 有料カンファレンスも、会社業務に直結するものであれば会社持ち。平日開催でも出勤扱いで参加可能

などの特徴があります。

求人しています！

セプテーニ・オリジナルでは、いつでもScalaを**やってみたい方**を募集しております。

最近入社される方は、次のような方が多いです。

- Scala未経験だが、Scalaには興味があり、興味がある理由も分かっているが、現状の会社では経験できない
- テストや設計などをちゃんとやりたいし、やる方法論も知っているが、現状の会社では重視されない

セプテーニ・オリジナルでは**ScalaやDDD未経験でも問題ありません。** 入社された方のために以下のカリキュラムを通してScala・DDDのトレーニングを積むことが出来ます。

1. Scalaに慣れるため、@yattom氏が公開されている研修問題 https://gist.github.com/yattom/567dfa484016012e21a0 をやってみる
2. Play Frameworkに慣れるため、簡単な掲示板を作成してみる
3. DDDに慣れるため、掲示板をDDDで作り直してみる

レビュアーに選出された5名がつくので、適宜アドバイスを受けながら進められます。

最近は他社様とのコラボ企画が多かったので、間が空いていますが 勉強会イベント http://septeni-scala.connpass.com/ やもくもく会 https://sep-ori-mokumoku-day.connpass.com/ も不定期に行っています。 ご興味をもたれた場合、connpassにご登録の上、上記グループにご参加していただくとイベント予定の通知が届き便利です。

もし我々に興味をもっていただけたのであれば http://septeni-original.co.jp よりエントリーいただだけると幸いです。

社内見学イベント http://tour.septeni.net/ も毎月実施しています。 まずは様子を見たい、というときはこちらもご検討ください。

本書はまだまだ続きます、どうぞ引き続きお楽しみください《杉谷保幸》

ScalaでAndroidアプリ開発

はじめに

GANMA!のAndroid版はScalaを使って開発をしています。

皆様がまず気になるのは「ぶっちゃけ（Scalaで作って）どう？」だとおもいますが、 結論からいうと**「動くし快適だが、途中が険しくてオススメしづらい」**といった代物です。

GANMMA!のダウンロード数は400万を超え、アクティブユーザーもかなりの人数となっています。 安定性も高い状態を保てており、今のところGoogle Play上でも高評価を得られています。 Scalaでの開発実績としては十分なものかと思われます。

本稿では、ScalaでAndroid開発を行ってみたい方のお役に立てるよう、 Android Studioを使った開発環境の構築方法をステップバイステップでご紹介いたします。

なぜScalaを使うのか？

繰り返しになりますが、ScalaでのAndroid開発は導入は面倒だし、将来性には難があります。 それでもその辛みを補って余りある嬉しさがあります。

- **楽に書ける** Scalaの豊富な記述性が醸し出す書き味は大変に素晴らしい物です。 標準CollectionAPIの充実、traitによる型表現の広がり、Scalaエコシステムの様々なライブラリ、どの要素も日々の開発を楽にしてくれます。
- **Future美味しい** Android開発では通信をしてからUI更新、といった非同期処理をよく書きますが、 ScalaのFutureは快適に非同期処理が記述できます。Java/Kotlinでもライブラリを使えばFuture/Promiseは扱えますが、Scalaでの書きやすさにはかなわない印象です。Akkaも利用すれば、非同期処理を扱う手段はさらに広がるでしょう。

我々の場合、サーバはScalaで開発しているのと、1エンジニアがサーバもクライアントも区別無くいじる文化なので クライアント側もScalaにしておきたかった、という都合からScalaでのAndroid開発を始めたのですが、とても楽に開発できています。

Scalaと比較するならKotlinでしょうか？ 実際の所Kotlinは深くは触っていないため、比較はできないのですが Kotlin公式サイトによる "Scalaとの比較" https://kotlinlang.org/docs/reference/comparison-to-scala.html には

> The main goal of the Kotlin team is to create a pragmatic and productive programming language, rather than to advance the state of the art in programming language research. Taking this into account, if you are happy with Scala, you most likely do not need Kotlin. (訳:Kotlinチームの主な目標は、プログラミング言語の研究における最先端技術を進歩させるのではなく、実用的で生産的なプログラミング言語を作成することです。これを考慮すると、あなたがScalaに満足しているのであれば、Kotlinはおそらく必要ないでしょう。)

と書かれています。

Javaから移行する場合、Scalaが未経験であればKotlinでも十分な幸福さを得られると思いますが、 Scalaに慣れている場合は、導入の面倒さや将来性の不安を押してでもScalaで開発をする価値があるでしょう。

sbt-androidとgradle-android-scala-pluginの比較

Androidの標準ビルドシステムはGradleですが、Scalaでよく使われるビルドシステムはsbtです。 ScalaでAndroid開発を行うアプローチは

- sbtをAndroidに対応させるアプローチ = pfn氏作 sbt-android を利用する
- GradleをScalaに対応させるアプローチ = saturday06氏作 gradle-android-scala-plugin を利用する

の二つが主な方法となります。

sbt-androidは開発が活発で動作が安定していますし、ドキュメントも充実しています。Instant Runも利用できるようです。

ただし、ビルドシステムがAndroid標準ではないので、プロジェクトをAndroid StudioやIDEAで扱うには一手間あるほか（実行設定をsbtを通した物にする必要がある等） テストやデバッガのIDE連携が出来ない[1]、などAndroid統合機能との連携に難があります。

gradle-android-scala-pluginはIDEAやAndroidStudioで特別な設定無しに扱えるのが大きなメリットです。 プロジェクトをインポートするだけで全てのAndroid統合機能が利用できます。デバッガやInstantRunも通常通り使えます。 特に動かしたいユニットテストをGUIから選んで単独実行できる点が大変に便利です。

ただし開発が活発ではない、**というか止まっている**という強烈な問題があります。

MavenのCentral Repositoryに登録されているバージョンでは 新しいGradleが利用できず、最新のAndroid Build Toolも使えない、ビルドも遅い、という問題があるため、 対応が進んでいるmasterブランチのHEADを取得し**自力でビルドを行う必要があります**[2]。 Android界の進展は早いので、将来にわたって利用できるかどうかが懸念となります。

我々はそれでも使い勝手が良く、テストを実行しやすいgradle-android-scala-pluginのほうを利用しています。

> [1]. 我々が動かす方法が見つけられなかっただけなので、方法はあるのかもしれません。

> [2]. メンテナとして名乗り出ようかな、と考えています。

プロジェクト作成

それでは実際にAndroid Studio 2.2.3を使って簡単なAndroidアプリを作れるようになるまで、 gradle-android-scala-pluginのビルドからステップバイステップで紹介いたします。

gradle-android-scala-pluginのビルド

最初にGradleのインストールします。 インストールにはJVM系のインストールマネージャであるSDKMANを利用します。

以下のコマンドでインストールされます。

```
$ curl -s get.sdkman.io | bash
```

次のコマンドを実行し初期設定を行います。

```
$ source "$HOME/.sdkman/bin/sdkman-init.sh"
```

次にgradleをインストールしますが gradle-android-scala-pluginは2017年の1月のmaster:HEADでは2.14.0,2.14.1ではビルドできなかったので、 対応とされている2.12をインストールします。[3]

```
$ sdk install gradle 2.12
```

gradle-android-scala-pluginをcloneします。

```
$ git clone https://github.com/saturday06/gradle-android-scala-plugin.git
```

ビルドします。

```
$ cd gradle-android-scala-plugin/
$ gradle assemble
```

`./build/libs/gradle-android-scala-plugin-1.5-SNAPSHOT.jar` にプラグインのパッケージが作成されます。 後ほどAndroidプロジェクト作成時に、このJARを利用します。

> 3. この場面以外なら新しいバージョンのGradleを利用できます。動作速度が向上しているのでより新しいバージョンを使った方が良いでしょう。

Android Studioのセットアップとプロジェクト作成

Android Studioをインストールし実行します。 初回起動時にインストール方法（Install Type)を聞かれますが、Standardで進めます。

Welcome to Android Studioの画面が表示されたら、メニューの Configure > Plugins を選び Install JetBrains plugin画面から、Scalaプラグインをインストールします。

インストールが終わったら、 Android Studioを再起動し、再びWelcome画面から Start a new Android Studio project を選び新規プロジェクトを作成します。 プロジェクト名はお好みで大丈夫です。本稿では `ScalaHelloWorld` という名前で `~/AndroidStudioProjects/ScalaHelloWorld/` に作成するとします。企業ドメインは `septeni-original.co.jp` (=Package Name が `jp.co.septeni_original`)とします。

Minimum SDKはScala自体はAndroid 2.x系が対象でも利用できなくはないですが、苦痛がとても大きいので4.0.3以上を推奨します。

Add an Activity to Mobileは Empty Activity を選択します。

Customize the Activityは初期値のままで進めます。

これで `~/AndroidStudioProjects/ScalaHelloWorld/` にプロジェクトが作成されました。

プロジェクトにgradle-android-scala-pluginを導入

プロジェクトのディレクトリ(`~/AndroidStudioProjects/ScalaHelloWorld/`)に 先ほど作成したgradle-android-scala-plugin-1.5-SNAPSHOT.jarをコピーし `build.gradle` で利用指定をします。（以降、ファイル名を表すとき `~/AndroidStudioProjects/ScalaHelloWorld/` を省略）

`build.gradle`

```
buildscript {
    …
    dependencies {
        classpath 'com.android.tools.build:gradle:2.2.3'
        classpath files("gradle-android-scala-plugin-1.5-SNAPSHOT.jar")
        …
    }
    …
}
```

次に `app/build.gradle` でプラグイン利用指定を行います。

```
…
apply plugin: 'com.android.application'
apply plugin: "jp.leafytree.android-scala"
…
```

依存ライブラリにScala 2.11.8を追加します。[4]

`app/build.gradle`

```
dependencies {
    …
    compile 'org.scala-lang:scala-library:2.11.8'
    …
}
```

これで最低限のプロジェクト設定はできました。

> [4]. Android SDKは現時点ではJava7バイトコードまでしか対応していないため、Scala2.12はまだ利用できません。

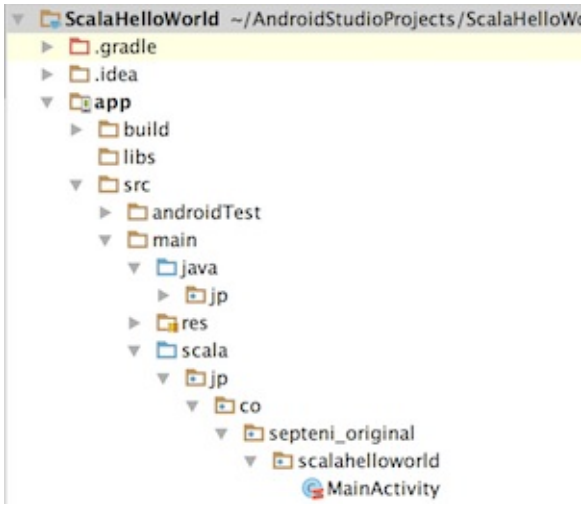
HelloWorld

Android Studioによって作られた `MainActivity.java` をScalaで書き直してみましょう。

`app/src/main` 以下にディレクトリ `scala` を作成します。

`app/src/main/scala` 以下にパッケージ `jp.co.septeni_original.scalahelloworld` を作成(= `jp/co/septeni_original/` でディレクトリを作成)します。

`app/src/main/scala/jp/co/septeni_original/` に `MainActiviy.scala` を作成します。



`MainActiviy.scala` の編集画面上部にNo Scala SDK in moduleと出てくるので、Setup Scala SDKを押下し、Scala 2.11.8を追加します。

`MainActivity.scala` に `MainActivity.java` と同等の記述をします。

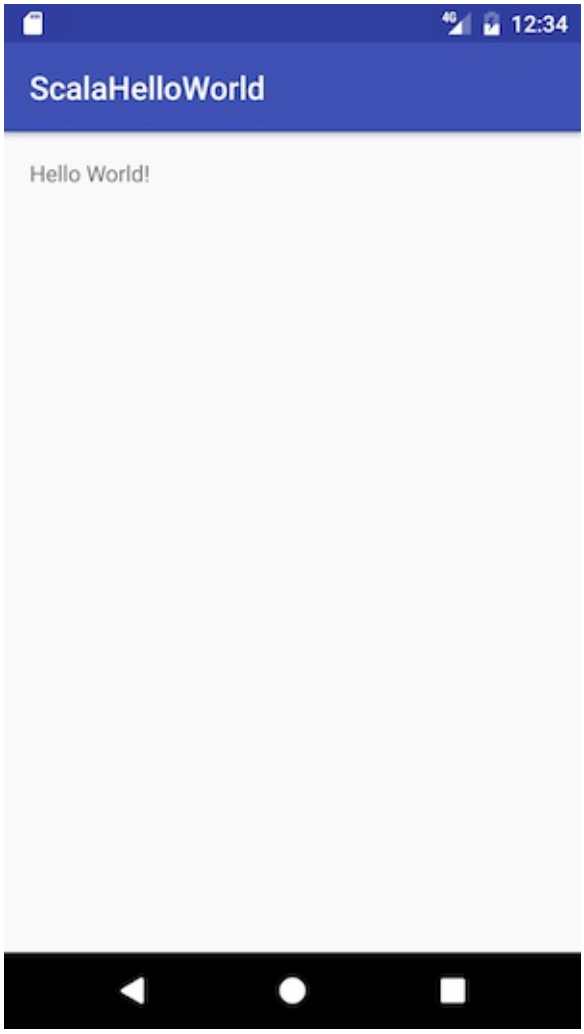
`MainActivity.scala`

```
package jp.co.septeni_original.scalahelloworld

import android.os.Bundle
import android.support.v7.app.AppCompatActivity

class MainActivity extends AppCompatActivity {

  override def onCreate(savedInstanceState: Bundle): Unit = {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
  }

}
```

Java側(`app/src/main/java` 以下）にある `MainActiviy.java` を削除します。

これで実装は完了です。

メニューのRun > Run app、もしくはメニューバーのRunボタンを押し、 Android5.0(API Level 20)以上[5]のエミュレーターを作成し、実行します。



デバッグ実行を行えばデバッガも利用できます。

> [5]. Scalaを導入するだけでクラス数は1dex上限の65535を超えてしまうので、Android5.0(API Level 20)未満ではまだ動かせません。後述するMultiDex設定が終わるまではAndroid5.0以上を利用します。

テスト

次に、 `MainActivity` と同じように、Android Studioにより作成された実機で実行する単体テスト
(`ExampleInstrumentedTest.java`)のほうもScalaで書き直してみましょう。残念ながらspecs2等は使えずJunit4となります。

`app/src/androidTest` 以下にディレクトリ `scala` を作成します。

`app/src/androidTest/scala` 以下にパッケージ `jp.co.septeni_original.scalahelloworld` を作成します。

`app/src/androidTest/scala/jp/co/septeni_original/` に `ExampleInstrumentedTest.scala` を作成します。

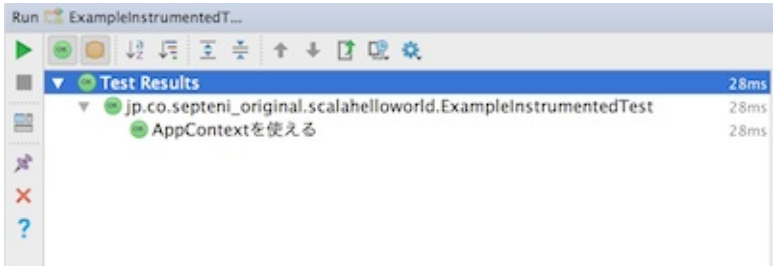
`ExampleInstrumentedTest.scala` に `ExampleInstrumentedTest.java` と同等の記述をします。

`ExampleInstrumentedTest.scala`

```
package jp.co.septeni_original.scalahelloworld

import android.content.Context
import android.support.test.InstrumentationRegistry
import android.support.test.runner.AndroidJUnit4
import org.junit.Assert._
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(classOf[AndroidJUnit4])
class ExampleInstrumentedTest {

  @Test
  def AppContextを使える(): Unit = {
    val appContext: Context = InstrumentationRegistry.getTargetContext
    assertEquals("jp.co.septeni_original.scalahelloworld", appContext.getPackageName)
  }
}
```

Java側(`app/src/androidTest/java` 以下）にある `ExampleInstrumentedTest.java` を削除します。

プロジェクトナビゲーター側からExampleInstrumentedTestを右クリックし、 Run 'ExampleInstrumentedTest'からテストを実行します。



MultiDex

次にAPI Level 20未満でも実行できるようにMultiDex[6]の設定を行います。

`app/build.gradle` に以下の設定を追記します。

```
android {
    defaultConfig {
      …
            multiDexEnabled true
      …
    }
}
```

```
dependencies {
    …
    compile "com.android.support:multidex:1.0.1"
    …
}
```

MultiDex処理を実行するApplicationクラスを新規作成します。[7]

`app/src/main/scala/jp/co/septeni_original/scalahelloworld/ScalaHelloWorldApplication.scala`

```
package jp.co.septeni_original.scalahelloworld

import android.app.Application
import android.content.Context
import android.support.multidex.MultiDex

class ScalaHelloWorldApplication extends Application {
  override protected def attachBaseContext(base: Context): Unit = {
    super.attachBaseContext(base)
    MultiDex.install(this)
  }
}
```

このApplicationクラスが実行されるようにManifestに追記します。

`app/src/main/AndroidManifest.xml`

```
    …
    <application
     android:name=".ScalaHelloWorldApplication"
     …
```

一番最初のdexファイル(`classes.dex`)には `ScalaHelloWorldApplication` クラス自身と、 `ScalaHelloWorldApplication` クラスが利用するクラスが含まれる必要があります。

次にこれを指定するファイルを `app/multidex.keep` に記述、設定します。

```
android/support/multidex/BuildConfig.class
android/support/multidex/MultiDex$V14.class
android/support/multidex/MultiDex$V19.class
android/support/multidex/MultiDex$V4.class
android/support/multidex/MultiDex.class
android/support/multidex/MultiDexApplication.class
android/support/multidex/MultiDexExtractor$1.class
android/support/multidex/MultiDexExtractor.class
android/support/multidex/ZipUtil$CentralDirectory.class
android/support/multidex/ZipUtil.class
jp/co/septeni_original/scalahelloworld/ScalaHelloWorldApplication.class
```

8

以降、`ScalaHelloWorldApplication`クラスが利用するクラスが増えたときはこのファイルに追記する必要があります[8]。

`multidex.keep`が読み込まれるよう、build設定を変更します。

`app/build.gradle`

```
android {
    …
    dexOptions {
        additionalParameters = [
                '--multi-dex',
                "--main-dex-list=$projectDir/multidex.keep".toString(),
                '--set-max-idx-number=45000'
        ]
    }
    …
}
```

additionalParametersの`--set-max-idx-number=45000`と`--multi-dex`は本来は不要なのですが、 Android4.0.x系では1dexに65535まで詰め込むと読み込めない事があるので、4.0.x系を動作対象に入れる場合は`--set-max-idx-number=45000`(1dex45000までに制限する)を指定したほうが安全です。 また我々の環境では`--multi-dex`も設定しておかないとInstant Runが正しく動作しなかったので指定しています。

以上でMultiDexの設定は完了です。API Level 15あたりのエミュレーターを作成し、動作確認を行いましょう[9]。

> 6. MultiDexに関する詳細説明は割愛します。

> 7. 実際にはここまで単機能であれば、`android.support.multidex.MultiDexApplication`を使うか継承するで十分なのですが、後ほど拡張するので本稿では自分で定義をしました。

> 8. 追記する場合、ビルド時に生成される`build/intermediates/multi-dex/debug/multidexlist.txt`を参照すると探しやすいです。

> 9. 最初のdexに格納されているclassを確認したい場合、後述の付録: dexに格納されているクラス一覧を確認する方法をご参照ください。

独自Applicationクラスの注意点

gradle-android-scala-pluginや元々のMultiDexのREADMEでは、独自のApplicationクラスを作る場合は以下に注意するよう案内されています。

- static field(Scalaであればclassやobjectのコンストラクタ)は`MultiDex#install`より先に呼ばれるので最初のclasses.dexに含まれる(multidex.keepに記述する）ようにしましょう。
- Applicationクラスのメソッドが実行されるとき、Applicationクラスより後に読まれる他のクラスのへのアクセス権がない可能性があります。以下のようにRunnableなど、別のクラスを噛ませると問題を回避できます。

```
override def onCreate = {
  super.onCreate

  val context = this
  new Runnable {
    override def run = {
      // 独自の処理をここに記述。
      // thisの代わりにcontextを使うこと
    }
  }.run
}
```

この場合、multidex.keepにいちいち利用するクラスを記述する必要がありません。

通しで記述すると以下のようになります。

app/src/main/scala/jp/co/septeni_original/scalahelloworld/ScalaHelloWorldApplication.scala

```
package jp.co.septeni_original.scalahelloworld

import android.app.Application
import android.content.Context
import android.support.multidex.MultiDex

class ScalaHelloWorldApplication extends Application {

  override def onCreate(): Unit = {

    super.onCreate()

    val context = this
    new Runnable {
      override def run(): Unit = {
        // 独自の処理をここに記述。
        // thisの代わりにcontextを使うこと
      }
    }.run()
  }

  override protected def attachBaseContext(base: Context): Unit = {
    super.attachBaseContext(base)
    MultiDex.install(this)
  }

}
```

Futureを利用する

ScalaでAndroidを開発する大きな理由の一つにFutureとPromiseがあります。 もちろんJava/Kotlinでもライブラリを使うなり作るなりすれば可能ですが、Scalaでの書き味の良さには及ばない印象があります。

通信などでFutureを利用する場合、大体の場合「通信をするFutureを作る→.andThenや.onCompleteでUIを更新する」といった処理になるのですが、AndroidではUI操作はUIスレッドで行う必要があるので、 最後のUI更新はUIスレッドで行う必要があります。

この処理をExecutionContextでラップし使い勝手を良くするコードを ScalaMatsuri座長の麻植さん(@OE_uia)が公開されています。

https://gist.github.com/taisukeoe/7812514

これを利用すると、Futureコールバックは UIスレッドで実行する処理は、以下のように記述できます。

```
val tv = new TextView(this)
val x: Future[Long] = future { … }
…
val mine = new UIExecutionContext(this)
x.onComplete { // onSuccessはScala2.12からdeprecatedなのでご注意
  case Success(count) => tv.setText(tv.getText + " onSuccess!:"+count.toString )
  case Failure(t) => …
}(mine)
```

また、場合によっては、次のようなサポートクラスを利用しています。

```
object ThreadUtil {

  def runOnUiThread(exec: => Unit): Future[Unit] = {
    val result = Promise[Unit]()

    def execute(): Unit = {
      Try {
        exec
      }.map { _ =>
        result.success(Unit)
      }.recover {
        case e: Throwable => result.failure(e)
      }
    }

    if (Looper.myLooper() == Looper.getMainLooper) {
      execute()
    } else {
      val handler = new Handler(Looper.getMainLooper)
      handler.post(new Runnable {
        override def run(): Unit = execute()
      })
    }

    result.future
  }
}
```

次のように利用します。

```
ThreadUtil.runOnUiThread(tv.setText("ほげ"))
```

付録: ProGuardについて

アプリをリリースするときProGuardをかける場合があります。

設定方法は通常と変わりませんが、Scalaのライブラリを多数除外指定する必要があります。

pfn氏作の sbt-androidのほうで容易されている https://github.com/scala-android/sbt-android/blob/master/resources/android-proguard.config をたたき台に調整していくと便利です。

付録: dexに格納されているクラス一覧を確認する方法

apkファイルをunzipし、出てきたdexファイルに対してdexdumpをする事によって確認できます。

```
$ cd app/build/outputs/apk
$ unzip app-debug.apk
$ dexdump *.dex | grep 'Class descriptor'
  Class descriptor  : 'Landroid/support/annotation/AnimRes;'
  Class descriptor  : 'Landroid/support/annotation/AnimatorRes;'
  Class descriptor  : 'Landroid/support/annotation/AnyRes;'
  Class descriptor  : 'Landroid/support/annotation/AnyThread;'
  Class descriptor  : 'Landroid/support/annotation/ArrayRes;'
  Class descriptor  : 'Landroid/support/annotation/AttrRes;'
  Class descriptor  : 'Landroid/support/annotation/BinderThread;'
  Class descriptor  : 'Landroid/support/annotation/BoolRes;'
  …
```

終わりに

如何でしたでしょうか？

ご紹介させていただいたとおり、使えるようになるまでがの道が険しく、将来の不安も残る方法です。大体の場合はKotlinを選ぶ方が良いでしょう。

しかしそれでもScalaを使う価値はあります。 本稿がScala愛のある方のお役に立てると幸いです。

《杉谷保幸》

単一責任の原則(SRP)についての見解と方法論

前説

飲み屋で小一時間議論が白熱した。 木村と今泉の中で単一責任の原則の考え方が違ったからだ。

木村がシステムリーダーを担っているプロジェクトでは、「あー。単一責任の原則はちゃんと守ろうね。一般的なOOPの考え方だよ。」というテンションでプロジェクトを進めていく内に、ドツボにハマった。

スタートアップさせるプロジェクトで初期設計者は木村含めた2人で行った。 最初の方はうまくいった。4人くらいまでチームがスケールしても初期設計はたいしてぶれなかった。 そんなこんなでマーケットが変わり、組織構造が変わり、チームメンバーが増減する中で気付いたら設計違反しているコードが目立つようになった。 設計を抜本的に改善している過渡期に今泉がチームにジョインしてきた。「単一責任の原則は一般的だからこそ明文化する必要があるんだよ」と発言してくれて、木村も納得感を得たので明文化してみた。 そしたらメンバーによって単一責任の原則の考え方が違うことが発覚したのだ。

我々は議論を重ね、単一責任の原則を考えるにあたって考慮すべきことをいくつか発見することができた。 本稿ではそれについての見解や方法論を紹介する。

過去の見解

木村の過去の見解

単一責任の原則は凝集度を強め、結合度を弱めて内部品質を高めるための戦術の1つくらいに捉えていた。 モデルの正当性はクライアントクラスを実装する時に明らかになると考えていた。 つまり、変更理由はクライアントクラスとその実装者によって発生するものだとしていた。 "ある程度"経験則と対話によって要件の変更を予測して、設計していた。

ただ、この予測が属人化しており明文化されていることはなかった。 可能性が低いクラスやモジュールに対しても構造分割をしているように映り、 不必要な複雑さ[1]と捉えるメンバーもいたようだった。

ビジネスの核はドメイン駆動開発(DDD)で捉えたつもりになっていたが、 要件レベルでの変更理由との同期を都度都度にとれてなかったため、 要件に沿わないモデルができあがってしまった。

今泉の過去の見解

1つのクラスが色んな事をできるのを防ぐためのものくらいに捉えていた。つまりそもそも誤った認識をしていた。 所謂神クラス的な物を避けるために、このクラスの役割はこれ1つだと定めるもの。 そのため、クラスの役割の粒度をプロジェクト単位で定めなければ、プロジェクトでの意思統一ができないものだと思い 上記のような発言になった。

余談となるが『オブジェクト指向入門 第2版方法論・実践』の「22.2.2 私のクラスは・・・を実行する」では、クラスに対して以下のような事を記載している。

> クラスは何か1つのことをするものではなく、特定の型のオブジェクトに対していくつかのサービスを提供するものである。 クラスが本当に1つのことだけをするのならば、おそらく偉大なる過ちのケースの1つであろう。 すなわち、ほかの何らかのクラスの1ルーチンにすべきものに対してクラスを作ったのである。

話を戻し、その議論の後、そもそも単一責任の原則って何?というところで原典を漁っていき、『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』にたどり着く。 会社で本を購入し、読んでみた所、責務 = 変更理由となっており、上記議論の際にもそもそも変更理由という単語が登場しない事に違和感を感じた。

今までコードレビューや会話等で責務という単語は多数聞いていたが、その全てが本当に変更理由を指していたとは到底思えないくらい変更理由という言葉の出て来る回数は責務に比べ少ない印象があった。

上記の誤解の後に、モジュールを変更する要因とはなんだろうと考えていった際に プロジェクトのステークホルダー、ステークホルダーを動かす組織体制というビジネス要因の部分。 プロジェクトのコード規約や言語、ライブラリ等のアップデート等の技術的要因の部分。 この2点があり、どちらかというと私は前者であるビジネス要因の部分、すなわち要件を強く考慮する必要があると感じた。

原理原則的にはどうなのか？

『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』の見解

単一責任の原則(SRP)

> クラスを変更する理由は1つ以上存在してはならない 単一責任の原則（SRP）では、「役割（責任） = 変更理由」と定義している。クラスを変更するのに2つ以上の理由がある場合、そのクラスには2つ以上の役割があることになる。

この「変更理由」こそが立場によって解釈が異なるのだ。『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』ではSOLIDが提唱されており、他の原則からも変更理由に関して考えを強化できる。

オープン・クローズドの原則(OCP: Open-Closed Principle)

> ソフトウェアの構成要素（クラス、モジュール、関数など）は拡張に対して開いて（オープン: Open）いて、修正に対して閉じて（クローズド: Closed）いなければならない
>
> どんなに「閉じた(Closed)」モジュールであっても、閉じることのできない変更が必ずあるのだ。つまり、すべてのケースに適用できる自然モデルなど存在しないのだ！ あらゆる変更に対して完璧に閉じることが不可能なら、戦略的に閉じるしかない。
>
> (中略)
>
> では、どうすれば発生しそうな変更を推測できるだろうか？　適切なリサーチを行い、適切な質問 を投げかけ、その上で経験と常識を使う。それ以外にできることは何もない。あとは実際に変更が起 きるのを待つしかない。

完璧なモデルが存在しないのであれば、先を見越しすぎても却って「不要な複雑性」につながる。 一方で先を全然見越さなければ、拡張性と修正性が損なわれる可能性がある。 モデリングする上でこのバランスを保つために変更理由となりうる登場人物との対話は欠かせないだろう。

リスコフの置換原則(LSP: Liskov Substitution Principle)

> 派生型はその基本型と置換可能でなければならない
>
> リスコフの置換原則は私たちを非常に重要な結論に導いてくれる。それは、「モデルの正当性は立場によって異なり、立場抜きに正当性を証明することは意味をなさない」ということだ。そのモデルの正当性を明確にできるのはクライアントだけなのだ。 ある特定の設計の正当性を検討するときは、他の立場を無視してそれ単独で考えてはいけない。その設計を利用するユーザーの視点で考え、その立場で合理的な仮定をしなければならない。

この結論の前には正方形と長方形の設計の話があった。 正方形は長方形の派生型と定義して最もらしいモデルができあがったが、 長方形の抽象を使うクライアントの使い方を考慮していなかったためクライアントにとって不備のある設計となってしまった例だった。 このように、立場が異なるとモデルの組み方も変わるので、変更理由となりうる立場の人は誰なのか、 その立場の人の考えを捉え、対話する必要がありそうだ。

『オブジェクト指向入門 第2版原則・コンセプト』の見解

開放/閉鎖の原則

> モジュールは開いていると同時にとじているべきである。
>
> モジュールが拡張を受け入れられる状態にある場合、そのモジュールは開放されているという。 モジュールがほかのモジュールから使用できる状態にある場合、そのモジュールは閉鎖されているという。
>
> どうすればモジュールを開放されているのと同時に閉鎖されている状態にすることができるのだろうか。 オブジェクト指向ならば、継承のおかげで特にエレガントにこの問題に対処することができる。

あるモジュールAを利用しているモジュールBが存在し、その後Aのモジュールを拡張、改修したモジュールを要求する別のモジュールCが登場した場合にモジュールAをどうすればいいのか。 こういう時にAを継承したA'を使えば、開放と閉鎖を満たせる。というもの。

『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』でのオープン・クローズドの原則は『オブジェクト指向入門 第2版原則・コンセプト』の開放/閉鎖の原則を参考元と置いているが、『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』でのオープン・クローズドの原則と『オブジェクト指向入門 第2版原則・コンセプト』の開放/閉鎖の原則はかなり違った印象を与える説明がなされている。『オブジェクト指向入門 第2版原則・コンセプト』ではオブジェクト指向におけるモジュール性に関する原理原則の文脈で登場しており、解決の手法に上記の様に継承を用いており、継承はエレガントに開放と閉鎖を達成すると記載されている。一方『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』でのオープン・クローズドの原則では抽象が鍵であると記載しており、さらにこの抽象はやりすぎると余計な複雑性を生むことの指摘をしている。

原典を漁っている時、同じ原則でさえこのように異なる視点で記述されている点を考えると、自分の中で当たり前と思っている原理原則達が、本当に正しく認識できているのか疑う視点を持つことの重要さというものがよくわかる。

単一責任選択の原理

> ソフトウェアシステムが選択肢を提供しなければならないとき、そのシステムの中の1つのモジュールだけがその選択肢のすべてを把握すべきである。

同じif文の分岐が多数のモジュールにある必要はない、1つにあれば多相性と動的束縛で解決出来る。というもの。

まとめの部分では下記のように記載されている。

> 単一責任選択の原則により、特定の概念の変化表明の完全な一覧の情報が分散されるのを防ぐことができる。

この記載を見た時に、「特定の概念の変化表明の完全な一覧の情報」と単一責任の原則における「変更理由」が 近しい概念を示しているように感じられた。

単一責任の原則で考慮する戦略と戦術

戦略について

戦略は顧客の要求やシステムの外部要因(組織構造や外部システム)、システム境界(ユースケースなど)、ドメインを変数として立てる。 これらの前提が変更するとシステムに対して大きな変更理由になる。 戦略面が明確になっていないとクラスやモジュール粒度での変更理由の統一感を持つことは困難だ。 戦略が整っていない(ないしは浸透していない)場合は設計の関心事が技術よりに向かう傾向がある。 この場合、単一責任の原則を適用する上でも技術よりな視点になってしまう。 システムによってもたらされる価値が重要だという本質から逸脱してしまう。 従って、戦略を先に立てる必要がある。

戦略の立て方

戦略の立て方で有用だったこと、もしくは有用だと思われることを紹介する。

ドメイン駆動設計(DDD)

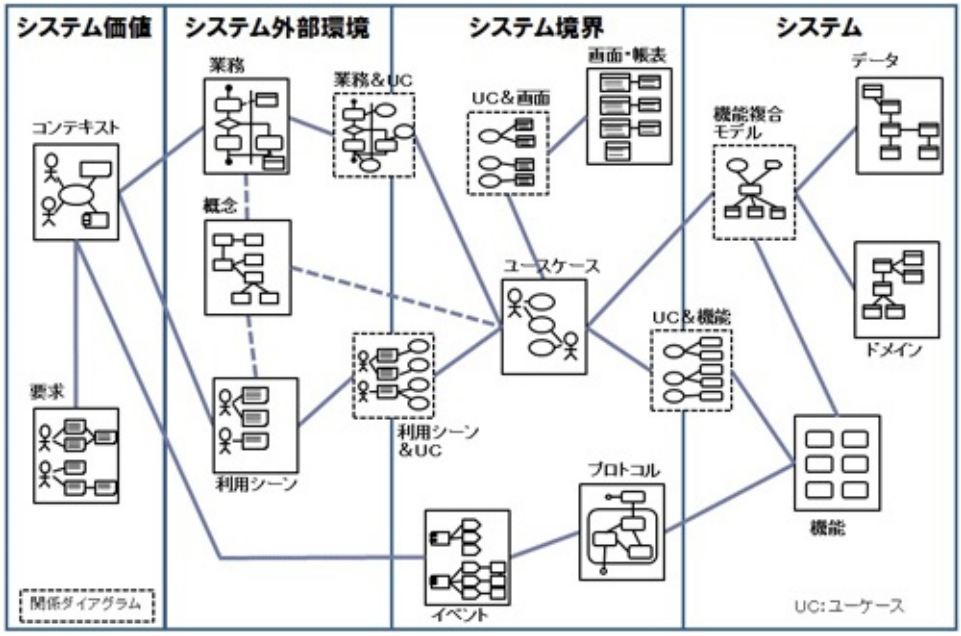
ドメイン駆動設計(DDD)では戦略的設計を扱うパターンがあり、単一責任の原則を扱う上で重要な視点となる。 戦略的設計はシステムの全体像を捉え、コンテキストの関係性を明らかできる。 コンテキストとはドメインモデルの解決領域のことを指す。 このコンテキスト単位でシステムやモジュール、チームを分割していく。

リレーションシップ駆動要件分析(RDRA)

リレーションシップ駆動要件分析(RDRA)では要件の構成要素を4つの領域に整理し、それらを4つのステップとして捉えている。

1.システム価値(価値を生み出す対象)
2.システム外部環境(システムを取り巻く環境)
3.システム境界(システムとの接点)
4.システム(システムそのもの)

以下は『構造に沿って要件をUMLで具体的に定義する』より引用



コンウェイの法則

「システムを設計する組織は、その構造をそっくりまねた構造の設計を生み出してしまう」というとても有名な法則。 開発組織とアーキテクチャが一致していない状態で、技術的な変更要請が発生した場合、開発組織のコミュニケーションの複雑さ、 修正難易度の高さ等は安易に想像ができるのではないだろうか。

組織はマーケットに従う

『組織パターン チームの成長によりアジャイルソフトウェア開発の変革を促す』という本の組織構築パターンの中にあるパターン言語の一つである。 マーケットの構造を開発組織に反映させる事を説いているパターンであり、上記のコンウェイの法則と連動し、アーキテクチャはマーケットに従わなければならないという事を説くパターンでもある。 つまり顧客要求のさらに先にあるマーケットの変動というものが変更理由の大きな要因となる。 変動が大きいマーケットに関わるアーキテクチャは、その変動の大きさに耐えうる柔軟性の高いものを要求される事となる。

戦術について

戦術は戦略を満たすためのプロセスを指す。 戦略から予想できる変更理由からクラス設計を駆動する。 クラス設計で迷ったり、議論に収拾がつかない場合に戦略に立ち返るとよいだろう。

チーム方針やコード方針を立てて戦略からぶれないようにすることや、 テスト駆動開発によって変更理由のフィードバックサイクルを早めることは1つの戦術となりうるだろう。 コード方針を整える上でパターンを利用することは有用だ。 DDDの戦術的設計のパターン[2]やGoFを始めとしたデザインパターンを元に議論するのもよいだろう。

結論

我々の見解の違いは要件とクライアントクラスだったがどちらも間違っていないと思う。 しかしオブジェクト指向は設計と開発の境界をシームレスに行っていく技法であることから、どちらの面からも原則を捉えるべき事が本来必要であった。 すなわち、要件の変更に対し柔軟に対応するための戦略、クライアントクラスからの要請に対し柔軟に対応する戦術(この対応が逆転することも往々にしてあるのだろう)、 その両方が揃うことで初めて単一責任の原則が目指す所に辿りつけるのだろう。

《木村 彰宏 / 今泉 光一》

[1]不必要な複雑さ

『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』の引用

> 「不必要な複雑さ」を内包している設計とは、設計している時点で不必要な要素が含まれている設計である。 こういった問題は、開発者が私用の変更を先取り師、あとで仕様が変更になったときに対処しやすい仕組みをあらかじめソフトウェアに仕込んでおく時に発生する。

[2]DDDの戦術的設計のパターン

戦術的設計としてDDDは7つのパターンを提供している。

- エンティティ
- 値オブジェクト
- モジュール
- サービス
- ファクトリ
- リポジトリ

参考文献
1) Bertrand Meyer『オブジェクト指向入門 第2版原則・コンセプト』 翔泳社
2) Bertrand Meyer『オブジェクト指向入門 第2版方法論・実践』 翔泳社
3) Robert C. Martin『アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技』SBクリエイティブ
3) 神崎善司『モデルベース要件定義テクニック』秀和システム
4) Eric Evans『エリック・エヴァンスのドメイン駆動設計 ソフトウェアの核心にある複雑さに立ち向かう』翔泳社
5) 神崎 善司 『構造に沿って要件をUMLで具体的に定義する』(https://codezine.jp/article/detail/5689 閲覧日: 2017/01/13)
6) James O.Coplien、Neil B.Harrison『組織パターン チームの成長によりアジャイルソフトウェア開発の変革を促す』

末尾再帰の呼び出し最適化の有無によるScalaコンパイラの挙動について

はじめに

セプテーニ・オリジナルに2016年8月に入社した嶽(だけ)@masayadk1229です。

突然ですが、末尾再帰の呼び出し最適化と聞いて、すぐにピンと頭に浮かぶでしょうか。 私はよくワードは耳にしていたのですが、ずっと理解が浅いまま過ごしていました。 ただ、社内のFP in Scala輪読会の発表する機会をいただき、理解することができました。 今回はその内容を書籍向けに修正したものをご紹介致します。

検証環境

今回はMac上でScala 2.12.1、jdk1.8.0_102の環境でコードの検証を行いました。

末尾再帰の呼び出し最適化とは

まず、末尾再帰の呼び出し最適化を理解するために、用語をそれぞれを説明致します。

- **末尾再帰** 再帰的な関数やプロシージャにおいて、自身の再帰呼び出しがその計算における最後のステップになっているような再帰のパターンのことです。 この後のサンプルコードで末尾再帰の有無のサンプルコードを記載致します。
- **呼び出し最適化** 末尾呼出しのコードを、戻り先を保存しないジャンプに変換することによって、スタックの累積を無くし、効率の向上などを図る手法です。 末尾呼出しの除去（tail call elimination）などとも言います。
- **コールスタック(スタック)** プログラムの実行中にサブルーチンに関する情報を格納するスタックです。単に「スタック」と言ったときにコールスタックを指していることが多いです。 再帰処理を行う際に、リターンアドレス(戻り先)は呼び出される度に記憶しておかなければならないので、コールスタックを消費します。
- **StackOverFlowError** 再帰の呼び出し階層が深くなると、コールスタックが溢れてしまいStackOverFlowErrorが発生します。
- **呼び出し最適化されると何が違うのか** コールスタックを消費しない実装に変換され、再帰の呼び出しが深くなってもStackOverFlowErrorが発生しなくります。
- **tailrecアノテーション** Scalaでは「@tailrec」を再帰関数の前につけると、末尾再帰でない場合にコンパイルエラーが発生します。

末尾再帰ではない場合

サンプルコード1

```
import scala.annotation._

object FibonacchiNotOptimized {
  def main(args: Array[String]) {
    println(fib(1000000))
  }

  // 末尾再帰ではありません。(関数を呼び出す以外のこと(足し算)をしています)
  // ローカルな末尾再帰関数を使用していません。
  // 大きな値を与えるとStackOverFlowErrorが発生します。
  // @tailrec
  def fib(n: Int): Int = {
    (n: @switch) match {
      case 0 => 0
      case 1 => 1
      case _ => fib(n - 1) + fib(n - 2)
    }
  }
}
```

呼び出し最適化がされていないことの確認

前述の通り、末尾再帰でなく呼び出し最適化されていないかは、tailrecアノテーションをつけてコンパイルエラーが出ることで判断できます。 サンプルコード1の11行目にある「@tailrec」行のコメントアウトを消してコンパイルしてみます。 想定通り、コンパイルエラーが発生しました。

```
$  scalac FibonacchiNotOptimized.scala
FibonacchiNotOptimized.scala:16: error:
could not optimize @tailrec annotated method fib:
it contains a recursive call not in tail position
      case _ => fib(n - 1) + fib(n - 2)
                           ^
one error found
```

StackOverFlowErrorが発生することを確認

前述の通り、再帰関数に大きな値を与えているので、実行するとStackOverFlowErrorが発生します。 想定通り、StackOverFlowErrorが発生しました。

```
$ scalac FibonacchiNotOptimized.scala
$ scala FibonacchiNotOptimized
java.lang.StackOverflowError
    at FibonacchiNotOptimized$.fib(FibonacchiNotOptimized.scala:16)
    at FibonacchiNotOptimized$.fib(FibonacchiNotOptimized.scala:16)
   .
   .
   .
  以下略
```

末尾再帰の場合

サンプルコード2

```
import scala.annotation._

object FibonacchiOptimized {
  def main(args: Array[String]) {
    println(fib(1000000))
  }

  // ローカルな末尾再帰関数(loop)を使用しています。
  // 大きな値を与えてもStackOverFlowErrorが発生しません。
  def fib(n: Int): Int = {
    @tailrec
    def loop(n: Int, prev: Int, cur: Int): Int = {
      n match {
        case 0 => prev
        case _ => loop(n - 1, cur, prev + cur)
      }
    }

    loop(n, 0, 1)
  }
}
```

呼び出し最適化がされていることの確認

末尾再帰で呼び出し最適化がされているとコンパイルエラーは発生しないはずです。 想定通り、コンパイルエラーはしませんでした。

```
$ scalac FibonacchiNotOptimized.scala
$
```

StackOverFlowErrorが発生しないことを確認

再帰関数に大きな値を与えていますが、実行しても呼び出し最適化によりStackOverFlowErrorが発生しないはずです。想定通り、StackOverFlowErrorが発生しませんでした。

```
$ scalac FibonacchiOptimized.scala
$ scala FibonacchiOptimized
1884755131
```

両者のJVMのバイトコード命令を比較してみる

JVMのバイトコード命令の作成方法

コンパイル時に作成されるclassファイルを下記のコマンドで指定すると、逆アセンブルされJVMのバイトコード命令が出力されます。

```
javap -c クラスファイル
```

JVMのバイトコード命令のgoto

今回、出力されるJVMのバイトコード命令について特に着目したいgotoについて、説明致します。 gotoとは特定の処理番号に命令を移し、処理が終わった後に戻るための命令のアドレス情報をコールスタックに格納します。

端的にいうと、このgoto命令が発生するのは再帰処理 or 分岐処理ということになります。 今回はgoto命令は再帰処理という認識で読み進めて下さい。 また、前述の通りこのコールスタックに情報を格納しきれなくなると、StackOverFlowErrorが発生します。

末尾再帰ではない場合(サンプルコード1)

コードを逆アセンブルすると、以下のJVMのバイトコード命令が出力されます。

```
$ scalac FibonacchiNotOptimized.scala
$ javap -c FibonacchiNotOptimized$.class
Compiled from "FibonacchiNotOptimized.scala"
public final class FibonacchiNotOptimized$ {
  public static FibonacchiNotOptimized$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class FibonacchiNotOptimized$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
       // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: ldc           #21                 // int 1000000
       6: invokevirtual #25                 // Method fib:(I)I
       9: invokestatic  #31                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      12: invokevirtual #35                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      15: return

  public int fib(int);
    Code:
       0: iload_1
       1: istore_2
       2: iload_2
       3: tableswitch   { // 0 to 1
                     0: 24
                     1: 28
                default: 32
          }
      24: iconst_0
      25: goto          50
      28: iconst_1
      29: goto          50
      32: aload_0
      33: iload_1
      34: iconst_1
      35: isub
      36: invokevirtual #25                 // Method fib:(I)I
      39: aload_0
      40: iload_1
      41: iconst_2
      42: isub
      43: invokevirtual #25                 // Method fib:(I)I
      46: iadd
      47: goto          50
      50: ireturn
}
```

goto命令があるので、再帰処理がされていると言えます。 再帰処理がされているということは、コールスタックにどんどん情報が積まれているということでもあります。

末尾再帰の場合(サンプルコード2)

コードを逆アセンブルすると、以下のJVMのバイトコード命令が出力されます。

```
$ scalac FibonacchiOptimized.scala
$ javap -c FibonacchiOptimized$.class
Compiled from "FibonacchiOptimized.scala"
public final class FibonacchiOptimized$ {
  public static FibonacchiOptimized$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class FibonacchiOptimized$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
       // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: ldc           #21                 // int 1000000
       6: invokevirtual #25                 // Method fib:(I)I
       9: invokestatic  #31                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      12: invokevirtual #35                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      15: return

  public int fib(int);
    Code:
       0: aload_0
       1: iload_1
       2: iconst_0
       3: iconst_1
       4: invokespecial #42                 // Method loop$1:(III)I
       7: ireturn
}
```

上記の結果を見るに、最適化されていなかったものに比べてgoto文が存在しなくなっていて、 コールスタックを消費しない実装になっています。 コールスタックを消費しないので、もちろんStackOverFlowErrorも発生しません。

比較結果

末尾再帰による呼び出し最適化の有無でJVMのバイトコード命令が大きく変わることが証明できました。 ただ、JVMのバイトコード命令での違いは証明できたが、Scalaコンパイラがどんな変換をして classファイルを出力したかは気になるところだと思います。

今度はclassファイルを逆コンパイルして、Javaコードにして違いをわかりやすく説明致します。

両者のclassファイルを逆コンパイルし、Javaコードで比較

Jadを用いて、クラスファイルを逆コンパイルして比較します。

以下のコマンドでJadをMacでインストールします。 brewは入っている前提とします。

```
brew tap homebrew/binary
brew install jad
```

以下のコマンドでclassファイルを逆コンパイルできます。

```
jad -r クラスファイル
```

末尾再帰ではない場合(サンプルコード1)

```
$ scalac FibonacchiNotOptimized.scala
$ jad -r FibonacchiNotOptimized$.class
```

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   FibonacchiNotOptimized.scala

import scala.Predef$;
import scala.runtime.BoxesRunTime;

public final class FibonacchiNotOptimized$
{

    public void main(String args[])
    {
        Predef$.MODULE$.println(BoxesRunTime.boxToInteger(fib(0xf4240)));
    }

    public int fib(int n)
    {
        int i = n;
        int j;
        switch(i)
        {
        case 0: // '\0'
            j = 0;
            break;

        case 1: // '\001'
            j = 1;
            break;

        default:
            j = fib(n - 1) + fib(n - 2);
            break;
        }
        return j;
    }

    private FibonacchiNotOptimized$()
    {
        MODULE$ = this;
    }

    public static FibonacchiNotOptimized$ MODULE$;

    static
    {
        new FibonacchiNotOptimized$();
    }
}
```

末尾再帰の場合(サンプルコード2)

```
$ scalac FibonacchiOptimized.scala
$ jad -r FibonacchiOptimized$.class
```

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   FibonacchiOptimized.scala

import scala.Predef$;
import scala.runtime.BoxesRunTime;

public final class FibonacchiOptimized$
{

    public void main(String args[])
    {
        Predef$.MODULE$.println(BoxesRunTime.boxToInteger(fib(0xf4240)));
    }

    public int fib(int n)
    {
        return loop$1(n, 0, 1);
    }

    private final int loop$1(int n, int prev, int cur)
    {
        do
        {
            int i = n;
            switch(i)
            {
            case 0: // '\0'
                return prev;
            }
            cur = prev + cur;
            prev = cur;
            n = n - 1;
        } while(true);
    }

    private FibonacchiOptimized$()
    {
        MODULE$ = this;
    }

    public static FibonacchiOptimized$ MODULE$;

    static
    {
        new FibonacchiOptimized$();
    }
}
```

比較結果

着目したい点としては、末尾再帰による呼び出し最適化の有無で出力されるJavaコードに違いがあることです。 呼び出し最適化がされていない場合は再帰で処理を行うことは変わっていません。 それに比べて呼び出し最適化がされている場合は再帰ではなく、do-while文のループになっていることです。 do-while文のループになっているため、コールスタックを消費しません

Scalaコンパイラで呼び出し最適化がされていれば、このような変換処理が行われてclassファイルが出力されることがわかりました。

【参考】switchアノテーション、lookupswitchとtableswitchについて

末尾再帰ではない以下のサンプルコードではswitchアノテーションをつけていました。

こちらについても、参考までに説明します。

サンプルコード1

```
import scala.annotation._

object FibonacchiNotOptimized {
  def main(args: Array[String]) {
    println(fib(1000000))
  }

  // 末尾再帰ではない。(関数を呼び出す以外のこと(足し算)をしている)
  // ローカルな末尾再帰関数を使用していない
  // 大きな値を与えるとスタックオーバーフローになる。
  // @tailrec
  def fib(n: Int): Int = {
    (n: @switch) match {
      case 0 => 0
      case 1 => 1
      case _ => fib(n - 1) + fib(n - 2)
    }
  }
}
```

switchアノテーションの説明

switchアノテーションをつけると、即値整数のみのmatch式は内部でJavaのswitch文に展開しますが、 このような最適化がされない場合はコンパイルエラーにする挙動となります。 では、なぜこのようなアノテーションが存在するのでしょうか。

caseの条件が即値整数の比較のみの場合だとswitch文に置き換えられます。 即値整数の比較以外だと、if文に置き換えられます。 このScalaのコンパイラの挙動を考えると、分岐が多いケースなどswitch文の方が高速であると考えられます。 ただし、caseの条件が即値整数の比較かどうかは少しの変更で変わってしまうため、 動作を保証するためにこのアノテーションが存在していると考えられます。

コンパイルエラーになる例

```
object Foo{
    final val a = 2
    val b = 3
 }
 def goo(x:Int) = (x: @switch) match {
   case 1 => true
   case Foo.a => true
   case Foo.b => true // コンパイルエラー
   case _ => false
 }
```

JVMのバイトコード命令

Scalaコンパイラで変換されるJavaのswitch文ですが、JVMのバイトコード命令では2種類の命令のどちらかに分類されます。

- **tableswitch** switch文のcase式の値が連続である場合に（キーがインデクス値、値が飛び越し先番地の）表を使って

飛び越しを行います。（高速です）
- **lookupswitch** switch文のcase式の値が不連続である場合に、値を探しながら飛び越し先を探し、飛び越しを行います。

同じくJavaのSwitch文に展開される場合でも、なるべくtableswitch命令が使われるように実装する方が高速に実行できます。

ちなみに今回の末尾再帰ではないコードは逆アセンブルすると、tableswitch命令が使われる形に実装しました。

サンプルコード1のJVMのバイトコード命令

```
$ scalac FibonacchiNotOptimized.scala
$ javap -c FibonacchiNotOptimized$.class
Compiled from "FibonacchiNotOptimized.scala"
public final class FibonacchiNotOptimized$ {
  public static FibonacchiNotOptimized$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class FibonacchiNotOptimized$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
        // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: ldc           #21                 // int 1000000
       6: invokevirtual #25                 // Method fib:(I)I
       9: invokestatic  #31                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      12: invokevirtual #35                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      15: return

  public int fib(int);
    Code:
       0: iload_1
       1: istore_2
       2: iload_2
       3: tableswitch   { // 0 to 1
                     0: 24
                     1: 28
               default: 32
          }
      24: iconst_0
      25: goto          50
      28: iconst_1
      29: goto          50
      32: aload_0
      33: iload_1
      34: iconst_1
      35: isub
      36: invokevirtual #25                 // Method fib:(I)I
      39: aload_0
      40: iload_1
      41: iconst_2
      42: isub
      43: invokevirtual #25                 // Method fib:(I)I
      46: iadd
      47: goto          50
      50: ireturn
}
```

ではどのようなコードだと、lookupswitch、tableswitchになるかはサンプルコードを参考に記載していきます。

lookupswitchの例

以下に、lookupswitchになる複数のサンプルコード、JVMのバイトコード命令を記載します。

ここ以降のgoto命令は再帰処理で使われてはおらず、switchの値が決まったらリターンするために使われています。

サンプルコード3

```
import scala.annotation._

object Lookupswitch {
  def main(args: Array[String]) {
    println(lookupswitch(0))
  }
  def lookupswitch(n: Int): Int = {
    (n: @switch) match {
      case 0 => 0
      case 1 => 1
      case 4 => 4 // 2 or 3 ならtableswitchとして動作します。
    }
  }
}
```

サンプルコード3のJVMのバイトコード命令

```
$ scalac Lookupswitch.scala
$ javap -c Lookupswitch\$.class
Compiled from "Lookupswitch.scala"
public final class Lookupswitch$ {
  public static Lookupswitch$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class Lookupswitch$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
       // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: iconst_0
       5: invokevirtual #24                 // Method lookupswitch:(I)I
       8: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      11: invokevirtual #34                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      14: return

  public int lookupswitch(int);
    Code:
       0: iload_1
       1: istore_2
       2: iload_2
       3: lookupswitch  { // 3
                     0: 36
                     1: 40
                     4: 44
                default: 48
          }
      36: iconst_0
      37: goto          60
      40: iconst_1
      41: goto          60
      44: iconst_4
      45: goto          60
      48: new           #39                 // class scala/MatchError
      51: dup
      52: iload_2
      53: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      56: invokespecial #41                 // Method scala/MatchError."<init>":(Ljava/lang/Object;)V
      59: athrow
      60: ireturn
}
```

サンプルコード4

```
import scala.annotation._

object Lookupswitch2 {
  def main(args: Array[String]) {
    println(lookupswitch(0))
  }
  def lookupswitch(n: Int): Int = {
    (n: @switch) match {
      case 0 => 0
      case 2 => 2 //1なら、tableswitchとして動作します。
    }
  }
}
```

サンプルコード4のJVMのバイトコード命令

```
$ scalac Lookupswitch2.scala
$ javap -c Lookupswitch2\$.class
Compiled from "Lookupswitch2.scala"
public final class Lookupswitch2$ {
  public static Lookupswitch2$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class Lookupswitch2$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
       // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: iconst_0
       5: invokevirtual #24                 // Method lookupswitch:(I)I
       8: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      11: invokevirtual #34                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      14: return

  public int lookupswitch(int);
    Code:
       0: iload_1
       1: istore_2
       2: iload_2
       3: lookupswitch  { // 2
                     0: 28
                     2: 32
                default: 36
          }
      28: iconst_0
      29: goto          48
      32: iconst_2
      33: goto          48
      36: new           #39                 // class scala/MatchError
      39: dup
      40: iload_2
      41: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      44: invokespecial #41                 // Method scala/MatchError."<init>":(Ljava/lang/Object;)V
      47: athrow
      48: ireturn
}
```

tableswitchの例

以下にtableswitchになる複数のサンプルコード、JVMのバイトコード命令を記載します。

サンプルコード5

```
import scala.annotation._

object Lookupswitch {
  def main(args: Array[String]) {
    println(lookupswitch(0))
  }
  def lookupswitch(n: Int): Int = {
    (n: @switch) match {
      case 0 => 0
      case 1 => 1
      case 3 => 3 // 2でもtableswitchとして動作します。
    }
  }
}
```

サンプルコード5のJVMのバイトコード命令

```
$ scalac Tableswitch.scala
$ javap -c Tableswitch\$.class
Compiled from "Tableswitch.scala"
public final class Tableswitch$ {
  public static Tableswitch$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class Tableswitch$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
       // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: iconst_0
       5: invokevirtual #24                 // Method tableswitch:(I)I
       8: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      11: invokevirtual #34                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      14: return

  public int tableswitch(int);
    Code:
       0: iload_1
       1: istore_2
       2: iload_2
       3: tableswitch   { // 0 to 3
                     0: 32
                     1: 36
                     2: 44
                     3: 40
                default: 44
           }
      32: iconst_0
      33: goto          56
      36: iconst_1
      37: goto          56
      40: iconst_3
      41: goto          56
      44: new           #39                 // class scala/MatchError
      47: dup
      48: iload_2
      49: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      52: invokespecial #41                 // Method scala/MatchError."<init>":(Ljava/lang/Object;)V
      55: athrow
      56: ireturn
}
```

サンプルコード6

```
import scala.annotation._

object Tableswitch2 {
  def main(args: Array[String]) {
    println(tableswitch(0))
  }
  def tableswitch(n: Int): Int = {
    (n: @switch) match {
      case 0 => 0
      case 1 => 1 //2だと、lookupswitchとして動作します。
    }
  }
}
```

サンプルコード6のJVMのバイトコード命令

```
$ scalac Tableswitch2.scala
$ javap -c Tableswitch2\$.class
Compiled from "Tableswitch2.scala"
public final class Tableswitch2$ {
  public static Tableswitch2$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class Tableswitch2$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
       // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: iconst_0
       5: invokevirtual #24                 // Method tableswitch:(I)I
       8: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      11: invokevirtual #34                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      14: return

  public int tableswitch(int);
    Code:
       0: iload_1
       1: istore_2
       2: iload_2
       3: tableswitch   { // 0 to 1
                     0: 24
                     1: 28
                default: 32
          }
      24: iconst_0
      25: goto          44
      28: iconst_1
      29: goto          44
      32: new           #39                 // class scala/MatchError
      35: dup
      36: iload_2
      37: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      40: invokespecial #41                 // Method scala/MatchError."<init>":(Ljava/lang/Object;)V
      43: athrow
      44: ireturn
}
```

サンプルコード7

```
import scala.annotation._

object Tableswitch3 {
  def main(args: Array[String]) {
    println(tableswitch(0))
  }
  def tableswitch(n: Int): Int = {
    (n: @switch) match {
      case 0 => 0
      case _ if (n == 3) => 3
    }
  }
}
```

サンプルコード7のJVMのバイトコード命令

```
$ scalac Tableswitch3.scala
$ javap -c Tableswitch3\$.class
Compiled from "Tableswitch3.scala"
public final class Tableswitch3$ {
  public static Tableswitch3$ MODULE$;

  public static {};
    Code:
       0: new           #2                  // class Tableswitch3$
       3: invokespecial #12                 // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #20
       // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: aload_0
       4: iconst_0
       5: invokevirtual #24                 // Method tableswitch:(I)I
       8: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      11: invokevirtual #34                 // Method scala/Predef$.println:(Ljava/lang/Object;)V
      14: return

  public int tableswitch(int);
    Code:
       0: iload_1
       1: istore_2
       2: iload_2
       3: tableswitch   { // 0 to 0
                     0: 20
                default: 24
           }
      20: iconst_0
      21: goto          48
      24: iload_1
      25: iconst_3
      26: if_icmpne     33
      29: iconst_3
      30: goto          45
      33: new           #39                 // class scala/MatchError
      36: dup
      37: iload_2
      38: invokestatic  #30                 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      41: invokespecial #41                 // Method scala/MatchError."<init>":(Ljava/lang/Object;)V
      44: athrow
      45: goto          48
      48: ireturn
}
```

以上、簡単な例を紹介しました。

最後に

如何でしたでしょうか？

今回、末尾再帰、呼び出し最適化、switchアノテーション、lookupswitch/tableswitchについて説明しました。 逆コンパイルや逆アセンブルを普段使わない方にとっては、目新しい内容だったかもしれません。 今後、最適化を実施する際には、導入の内容としてぜひご活用下さい。

また、世の中のプログラマーは末尾再帰がわかるプログラマーとわからないプログラマーで二分されると誰かが言ってました。 末尾再帰がわかるプログラマーになるためにも、本稿が少しでも役に立てると幸いです。

《嶽 雅也》

参考にさせていただいたブログ等

https://ja.wikipedia.org/wiki/%E6%9C%AB%E5%B0%BE%E5%86%8D%E5%B8%B0

https://ja.wikipedia.org/wiki/%E3%82%B3%E3%83%BC%E3%83%AB%E3%82%B9%E3%82%BF%E3%83%83%E3%82%AF

https://ja.wikipedia.org/wiki/Java%E4%BB%AE%E6%83%B3%E3%83%9E%E3%82%B7%E3%83%B3

https://groups.google.com/forum/#!topic/scala--tohoku/rD2jT4Be5sI

http://mikamikuh.hatenablog.com/entry/2013/07/03/230712

http://taka-2.hatenablog.jp/entry/20110922/p1

http://www.ne.jp/asahi/hishidama/home/tech/scala/annotation/index.html#switch

http://hacknote.jp/archives/10259/

http://www.ne.jp/asahi/hishidama/home/tech/scala/annotation/

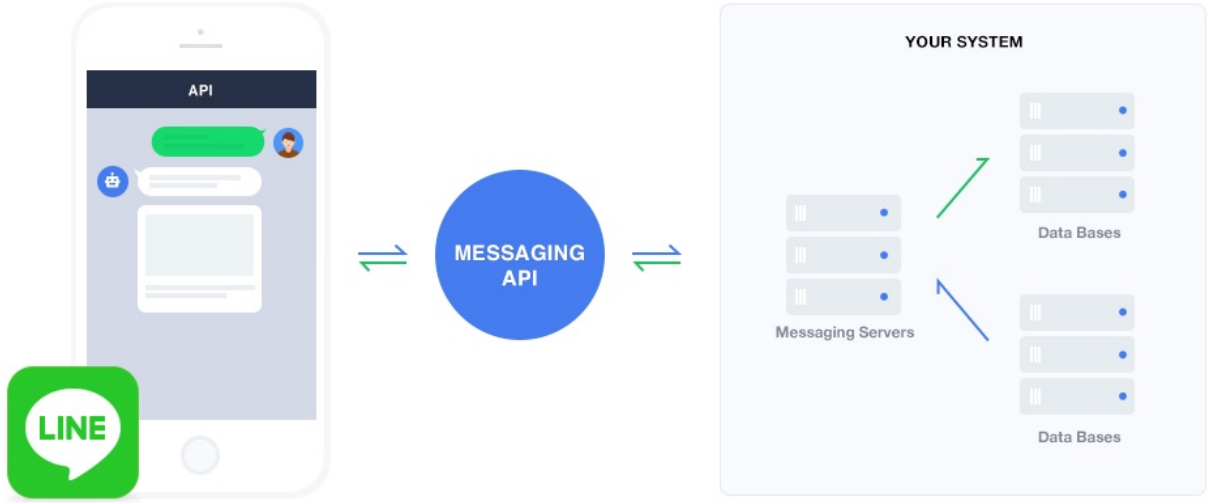
Akka HTTP で LINE bot を作ってみました

こんにちは、去年年末で父親になった肖と申します。

当時のScala界隈の大きな出来事の一つに、Akka HTTPの正式版 X のリリースがありました。 Akka HTTPを詳しく勉強するため、簡単なWebサービスを作ってみました。 今回のScalaMatsuri2017ではAkkaやストリーム処理関連の話が多いので、そのブームに乗りつつご紹介しようと思います。

LINE Bot とは

> LINE が提供する Messaging API により、LINEのトーク画面を使った対話型Botアプリケーションの開発が可能になります。



※ LINE公式ドキュメント から引用

この図を説明すると、

- サーバのURLを LINE Bot アカウント[1]と繋いでいます。
- ユーザがBotに対して、友たち追加、メッセージ送信などを行う際に、設定されたサーバにイベント情報が送られます。
- サーバがイベント情報を基づいて、LINE Messaging API経由でユーザに返信できます。

> [1]. LINE アカウントの設定について、LINEの公式ドキュメントに記載されているので、割愛します。

Akka HTTPについて

Akka HTTP は Akka Actor と Akka Stream をベースに作られた、HTTPサーバ／クライエント処理をサポートするためのツール、ライブラリです（フレームワークではありません！[2]）。

主な特徴は、HTTP処理（HttpRequest => HttpResponse）をストリーム処理で実現することによって、高いパフォーマンス[3]を出せるようになっています。 そしてもう一つの特徴は、分かりやすいRoute DSLです。LINE botを作るには、Akka HTTP みたいなツールが丁度良いと思います。

> [2]. Akka HTTPの哲学: http://doc.akka.io/docs/akka-http/current/scala/http/introduction.html#philosophy

> [3]. Akka HTTPのパフォーマンスについて: http://akka.io/news/2016/08/02/akka-2.4.9-RC1-released.html

今回で作る Bot

今回は Akka HTTP の特性の紹介するため、一番シンプルな「受けたメッセージをそのまま返す」Botを例にして、Akka HTTP の特性を幾つか紹介したいと思います。

- JSON を変換する ((un)Marshalling)
- HTTP リクエストを処理する(Server side)
- HTTP リクエストを送る(Client side)
- Route をテストする

ソースコードはこちらでたどり着けます: https://github.com/xoyo24/akka-http-line-bot

TL;DR

JSON を変換する ((un)Marshalling)

LINE Messaging API とのやりとりは全部JSONで行います。

Akka HTTP の特徴の一つである(un)Marshalling機構を使うとJSON、あるいはXMLやバイナリデータなどを簡単に扱うことできるので、活用します。

JSONに関してはspray-json ベースの変換機構が用意されています。使い方は以下になります。

1. spray-jsonモジュールを使うため、Dependencyに"akka-http-spray-json"を追加。
2. スコープ内に RootJsonFormat[T] を用意する。
3. Marshalling と Unmarshalling 機構を使うため、SprayJsonSupport を導入する。

RootJsonFormat を用意する方法

場合に応じた二つの方法があります。

1. Case Class を直接変換する場合、要素の数に応じてjsonFormatX メソッドを利用する。

Reply Message API の Send Message Object を JSON に変換するため、以下のようなjsonFormatを用意します。

```
case class TextMessage(`type`: String = "text", text: String)

case class Messages(replyToken: String, messages: List[TextMessage])

trait MessagesJsonSupport extends SprayJsonSupport with DefaultJsonProtocol {
  implicit val textMessageFormat = jsonFormat2(TextMessage)
  implicit val messageFormat = jsonFormat2(Messages)
}
```

2. それ以外の場合、RootJsonFormat[T] を実装する

Callback Event の Source の継承関係を表したい場合、以下のような自前の実装が必要になります[4]。

```
trait Source {
  val `type`: String
}

case class UserSource(id: String) extends Source {
  override val `type`: String = "user"
}

case class GroupSource(id: String) extends Source {
  override val `type`: String = "group"
}

case class RoomSource(id: String) extends Source {
  override val `type`: String = "room"
}

trait SourceJsonSupport extends SprayJsonSupport with DefaultJsonProtocol {
  implicit object SourceFormat extends RootJsonFormat[Source] {
    def write(s: Source): JsValue = s match {
      case UserSource(id) => JsObject(
        "type" -> JsString(s.`type`),
        "userId" -> JsString(id)
      )
      case GroupSource(id) => JsObject(
        "type" -> JsString(s.`type`),
        "groupId" -> JsString(id)
      )
      case RoomSource(id) => JsObject(
        "type" -> JsString(s.`type`),
        "roomId" -> JsString(id)
      )
      case _ => serializationError(s"Unsupported Source Type '${s.`type`}' !")
    }

    def read(value: JsValue): Source =
      value.asJsObject.getFields("type", "userId", "groupId", "roomId") match {
        case Seq(JsString("user"), JsString(id)) => UserSource(id)
        case Seq(JsString("group"), JsString(id)) => GroupSource(id)
        case Seq(JsString("room"), JsString(id)) => RoomSource(id)
        case _ => deserializationError(s"Source expected")
      }
  }
}
```

4

> [4]. 詳しい説明は spray-json のドキュメントを参照してください。

spray-json 以外のJSONライブラリを使いたい場合

(un)Marshalling 機構の抽象化によって、JSON ライブラリの切り替えは簡単に出来るようになっています。 例えば jackson を使いたい場合、以下の用にUnmarshallerとMarshallerを定義します[5]。

```
trait JacksonSupport {
  import JacksonSupport._

  private val jsonStringUnmarshaller =
    Unmarshaller.byteStringUnmarshaller
      .forContentTypes(`application/json`)
      .mapWithCharset {
        case (ByteString.empty, _) => throw Unmarshaller.NoContentException
        case (data, charset)       => data.decodeString(charset.nioCharset.name)
      }

  /**
   * HTTP entity => `A`
   */
  implicit def jacksonUnmarshaller[A](
      implicit ct: ClassTag[A],
      objectMapper: ObjectMapper = defaultObjectMapper
  ): FromEntityUnmarshaller[A] = {
    jsonStringUnmarshaller.map(
      data => objectMapper.readValue(data, ct.runtimeClass).asInstanceOf[A]
    )
  }

  /**
   * `A` => HTTP entity
   */
  implicit def jacksonToEntityMarshaller[Object](
      implicit objectMapper: ObjectMapper = defaultObjectMapper
  ): ToEntityMarshaller[Object] = {
    Jackson.marshaller[Object](objectMapper)
  }
}
```

> [5]. 詳細はこちらのライブラリhseeberger/akka-http-json を参照してください。

HTTP リクエストを処理する(Server side)

ユーザがLINE Botに対して友たち追加やメッセージ送信などを行った際に、設定されたサーバにイベント情報がPUSHされます。

HTTP サービスを実装するため、Akka HTTP は Low Level と High Level 二種類の API を提供しています。

Low Level API

Low Level API は **akka-http-core** モジュールによって提供されて、ストリーム処理にみたいにHTTP 処理を実装出来るAPIです。 主に以下のスコープにフォーカスしています。

- コネクション管理
- メッセージやヘッダの変換処理
- リクエストとコネクションの Timeout 管理
- HTTP pipelining の Response ordering

今回は利用しませんが、実際に使うとこんな感じなります。

```
import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.model.HttpMethods._
import akka.http.scaladsl.model._
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.Sink

implicit val system = ActorSystem()
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

val serverSource = Http().bind(interface = "localhost", port = 8080)

val requestHandler: HttpRequest => HttpResponse = {
  case HttpRequest(GET, Uri.Path("/"), _, _, _) =>
    HttpResponse(entity = HttpEntity(
      ContentTypes.`text/html(UTF-8)`,
      "<html><body>Hello world!</body></html>"))

  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
    HttpResponse(entity = "PONG!")

  case HttpRequest(GET, Uri.Path("/crash"), _, _, _) =>
    sys.error("BOOM!")

  case r: HttpRequest =>
    r.discardEntityBytes() // important to drain incoming HTTP Entity stream
    HttpResponse(404, entity = "Unknown resource!")
}

val bindingFuture: Future[Http.ServerBinding] =
  serverSource.to(Sink.foreach { connection =>
    println("Accepted new connection from " + connection.remoteAddress)

    connection handleWithSyncHandler requestHandler
    // this is equivalent to
    // connection handleWith { Flow[HttpRequest] map requestHandler }
  }).run()
```

High Level API

High Level API は **akka-http** モジュールによって提供されて、Low Level API では実装されてないルーティング機能を独自のDSLによってサポートし、分かりやすくHTTP 処理を実装出来る APIです。

今回のBotはこんなイメージで実装しています。

```
implicit val system = ActorSystem("LINE-bot")
implicit val materializer = ActorMaterializer()
implicit val ec = system.dispatcher

def routes: Route = {
  path("line" / "callback") {
    (post & entity(as[Events])) { request =>

      ...

      complete {
        "OK"
      }
    }
  }
}

Http().bindAndHandle(routes, config.getString("http.interface"), config.getInt("http.port"))
```

Akka HTTPでは、Reoute を作成するためのDSLを Directive[6] と呼んでいます。 今回は以下のDirectiveを使いました。

- **path("line" / "callback")** はサービスのエンドポイントURLをフィルターします。
- **(post & entity(as[Events]))** は POST メソッドをマッチし、**Events** に変換出来るリクエストをフィルターします。
- **complete** は処理成功（200コード）を表します。
- なお、フィルターされたリクエストは Rejection になって、エラーとしてかえします。

> [6]. 他のDirectiveについては Akka HTTP公式ドキュメントのHigh-level Server-Side API を参照してください。

独自の Directive を作って、Signatureチェックする

LINE Bot を実装する際に、セキュリティのため、Signatureをチェックする必要があります。LINEの公式ドキュメントによると、

> リクエストの送信元がLINEであることを確認するために署名検証を行わなくてはなりません。 各リクエストには X-Line-Signature ヘッダが付与されています。 X-Line-Signature ヘッダの値と、Request Body と Channel Secret から計算した Signature が同じものであることをリクエストごとに 必ず検証してください。
>
> 検証は以下の手順で行います。
>
> 1. Channel Secretを秘密鍵として、HMAC-SHA256アルゴリズムによりRequest Bodyのダイジェスト値を得る。
> 2. ダイジェスト値をBASE64エンコードした文字列が、Request Headerに付与されたSignatureと一致することを確認する。

となります。

Akka HTTP に既存の Directive を組み合わせることによって、チェック処理をシンプルに実装できます。

```
val channelSecret: String
val signatureVerifier: SignatureVerifier

def verifySignature: Directive0 =
  (headerValueByName("X-Line-Signature") & entity(as[String])).tflatMap {
    case (signature, body) if signatureVerifier.isValid(channelSecret, body, signature) => pass
    case _ => reject(ValidationRejection("Invalid signature"))
  }

def routes: Route = {
  (path("line" / "callback") & post & verifySignature & entity(as[Events])) { entity =>
    ...
    complete {
      "OK"
    }
  }
}
```

HTTP リクエストを送る(Client side)

ユーザに返信する場合、LINE の Reply Message API を利用します。

他の HTTP API を利用する場合、Akka HTTPでは Connection-Level、Host-Level、Request-Levelの三種類のAPIが用意されています。

リクエストを作成する

まずはリクエストを作成します。これを各種APIを用いてサーバに送信します。

今回は以下のように、Reply Message のリクエストを作っています。

```
val auth = headers.Authorization(OAuth2BearerToken(accessToken))
val content = Messages(
  replyToken = token,
  messages = List(TextMessage(text = message))
)

val request = RequestBuilding.Post(
  uri = "https://api.line.me/v2/bot/message/reply",
  content = content
).withHeaders(auth)
```

- Akka HTTPは、**Authorization** 機構を通じて、OAuth認証などをサポートしています。
- Akka HTTPでは、HTTP の body の型を HTTPEntity と呼びます。
- **HttpRequest** を直接実装するのもできますが、content を HTTPEntity に自動変換させるため、**RequestBuilding** を使いました。

- **Connection-Level Client-Side API**

HTTPコネクションを完全管理し、ストリーム方式で処理を実装できる、最も低レベルかつ柔軟性をもつAPIです。

```
val connectionFlow: Flow[HttpRequest, HttpResponse, Future[Http.OutgoingConnection]] =
  Http().outgoingConnectionHttps("api.line.me")
val responseFuture: Future[HttpResponse] =
  Source.single(request)
    .via(connectionFlow)
    .runWith(Sink.head)
```

こちらは基本一つの Request に対して一つの HTTP コネクションになっています、ストリーム処理が終わると、コネクションはクローズされます。

Host-Level Client-Side API

HTTPコネクションの管理が不要、エンドポイント(url:port)毎にコネクションプールを作るAPIです。

```
val poolClientFlow = Http().cachedHostConnectionPoolHttps[Int]("api.line.me")
val responseFuture: Future[(Try[HttpResponse], Int)] =
  Source.single(request -> 42)
    .via(poolClientFlow)
    .runWith(Sink.head)
```

書き方は **Connection-Level Client-Side API** とほぼ同じです、変わったのは以下の二箇所のみです、

- **Http().cachedHostConnectionPoolHttps()** を使うことで、"api.line.me"のアクセスする場合は、同じプールからコネクションを取得します。
- コネクションプールの使用によって、レスポンスとリクエストの順番が一致しない可能性が有るため、もう一つパラメーターが必要になります。今回の場合、順番は重要ではないので固定で書いています。
- 設定ファイルから **akka.http.host-connection-pool** でコネクションプールを調整できます。

Request-Level Client-Side API

もっともシンプルなAPIです。二種類の書き方があります。

Flow-base 方式

```
val poolClientFlow = Http().superPool[Int]()
val responseFuture: Future[(Try[HttpResponse], Int)] =
  Source.single(request -> 42)
    .via(poolClientFlow)
    .runWith(Sink.head)
```

Host-Level APIと似たような書き方ですが、superPoolでエンドポイントを指定出来ないので、リクエストに完全なURIが必要な点が異なります。

Futrue-base 方式

こちらはより簡単なやり方です。

```
val responseFuture: Future[HttpResponse] = Http().singleRequest(request)
```

Route をテストする

今回のプログラムはHerokuにデプロイしていますが、変更するたびに確認のためデプロイし直さないといけないのは、ちょっと手間がかかります。 そこで、テストを追加して手間を省きます。

Akka HTTPは、**akka-http-testkit** モジュールによって、HTTP サービスを簡単にテスト出来るようになっています。

今回のBotは以下のようなやり方でテスト出来ます。

```
class EchoBotSpec
  extends FlatSpec
    with Matchers
    with ScalatestRouteTest
    with EventsJsonSupport
    with MockFactory {

  it should "reply text message as reveived" in {
    ...

    Post("/line/callback", body).withHeaders(header) ~> bot.routes ~> check {
      status shouldBe StatusCodes.OK
      responseAs[String] shouldBe "OK"
    }
  }
}
```

REQUEST ~> ROUTE ~> check { ASSERTIONS } の方式でレスポンス[7]をチェック出来ます。

> [7]. チェック出来る項目については、Akka HTTP公式ドキュメントのRoute TestKit を参照してください。

おわり

今回はテキストベースのBotに必要のものを紹介しました。 DBやAkka Presistenceを使えば情報を保持出来るBotを作れますし、他の 3rd Party API に繋げばいろんなサービスを提供することが出来ます。

参考

- 前回の記事: http://labs.septeni.co.jp/entry/2016/12/26/120625
- Akka HTTP リリースノート: http://akka.io/news/2016/11/22/akka-http-10.0.0-released.html
- Akka HTTP 公式ドキュメント: http://doc.akka.io/docs/akka-http/current/scala/http/
- LINE公式ドキュメント: https://developers.line.me/messaging-api/overview
- LINE API リファレンス: https://devdocs.line.me/ja/#reply-message
- spray-json: https://github.com/spray/spray-json
- akka-http-json: https://github.com/hseeberger/akka-http-json

《肖楊》

新卒scalaエンジニアが書くISUCONの歩き方

新卒1年目の東です。

個人的にハッカソンやコンテストに出るのが好きで、去年のISUCON6予選[1]にも挑戦しました。

ISUCON

> LINE社主催の"Iikanjini Speed Up Contest"の略称です。 予選と本選の2部構成で開催され、優勝賞金はなんと**100万円**です。 数人でチームを組み、与えられたWebアプリケーションに対して、8時間以内でありとあらゆる高速化を施し、ベンチマークテストのスコアを競います。 短い時間の中でアプリケーションの要件把握・ボトルネック特定・処理改善を行う必要があり、事前に"秘伝のタレ"を作って準備したり、当日のチームの動き方を決めて練習を重ねることが重要になります。

結果は惨敗だったのですが、ISUCON6予選はアプリケーションにボトルネックが仕掛けられており、非常に勉強になるものでした。 また、ごく短期間の開発におけるチームの動き方について、得るものが多かったように感じます。

ところで、ISUCON6予選の言語比率[2]をながめてみると、scalaは**2組だけ**です。 そこで今回は、毎年増えるISUCON参加者へ向けてISUCON初心者向けの事前準備の進め方を、 scalaでISUCONに挑戦する人が増えることを願い、scalaでISUCON6予選問題を初期状態から少し改善するまでを記事にしてみます。

ISUCONの事前準備の進め方

チームを組む

ISUCONは2名もしくは3名での参加が認められています。**単独での参加は不可です。** 予選突破を目指す場合、チームメンバーそれぞれに担当領域を割り当てるよりも、全員が複数あるいは全領域を触れる状態にすることをお勧めします。 というのも、ここ最近の予選問題は1つのテーマに基づいたボトルネックが設定されており、そこにチームの全リソースを割いて取り組むことでスコアが大きく跳ね上がるためです。

自分が解いた過去のISUCON予選内容を振り返ってみます。

開催会	テーマ	ボトルネック	主な解決手法
ISUCON4	銀行のログインに関するセキュリティ	ログイン禁止制御	ログインの失敗回数をメモリに載せてしまう
ISUCON5	SNSのユーザー同士のコミュニケーション	N+1問題	クエリ修正, テーブル再設計
ISUCON6	キーワードに関するページの相互リンク	膨大な正規表現マッチング	正規表現をメモリに載せてしまう, Aho Corastick法を使う

もちろん予選トップ通過や優勝を狙うのであれば、アプリケーション・データベース・インフラそれぞれ高度なチューニングが必要となります。

当日の進め方の相談

少なくとも最初の1時間の動き方について、チームで計画を組むことをお勧めします。

- 初期状態をremoteのリポジトリに保存する
- ベンチマークの初期スコアを計測する

といった作業は分割して個人に割り振り、1時間以内にどこまでできているかの目処を立てておくと、 想定外に時間がかかった作業をみつけつつ、実際に改善着手するタイミングとのズレを意識できます。 例として、自分のチームで使ったざっくり計画表を載せます。

東	done	Aさん	done	Bさん	done
インスタンス作成		インスタンス作成		インスタンス作成	
残り2人のip確認		インスタンスのipを東へ		インスタンスのipを東へ	
ansible流す		-		-	
設定ファイルのバージョン管理		DBのdump		githubに初期状態push	
言語切り替え&ベンチマーク初回		ブラウザで動作確認		ブラウザで動作確認	
当日マニュアル確認		当日マニュアル確認		当日マニュアル確認	
アクセス解析		スキーマ確認		プロファイラ	
ベンチマーク中のプロセス調査		スロークエリ解析		アプリケーションログ確認	
結果報告・相談		結果報告・相談		結果報告・相談	

最後の結果報告・相談まで1時間の見積もりだったのですが、ISUCON6予選時にはプロファイラの導入がうまくいかないまま全体的に遅れ、1時間半ほどかかる結果となってしまいました。 ですが、作業分割して決まった報告タイミングを設置することで自分の作業に集中し、うまくいかなかった部分に対して全員で妥協案を探すことができたように思います。

秘伝のタレを作る

秘伝のタレというと想像しづらいかもしれませんが、要はある程度設定を書き込んだ `nginx.conf` や `my.cnf` を用意したり、アクセスログやスロークエリの解析ツールの導入を自動化しておく、といった作業のことを指します。 終了間際になると、直接本番のソースコードを編集してベンチマーク実行する場合も少なくないので、 `.vimrc` や `.zshrc` の配置自動化も行っておくと良いかもしれません。 ただ、対象インスタンスのスペックやアプリケーションの仕様によって最適な設定は変化するため、基本的には自分は以下の範囲に収めています。

- nginx.conf
  - アクセスログを出力し、出力先を指定
  - 少しでも速く処理するため、その他のログの出力はしない
  - 静的ファイル配信の設定をコメントアウトで作っておく
- my.cnf
  - スロークエリを出力し、出力先を指定
  - クエリキャッシュの設定をコメントアウトで作っておく
- .zshrc, .vimrc
  - チームメンバー個人の好みな設定を受け取っておき、それぞれのインスタンスへの配置自動化
- その他
  - アクセスログ解析, クエリ解析, プロファイラ導入をansibleで自動化

- 各種設定ファイルのそれぞれのインスタンスへの配置をansibleで自動化

チームにインフラエンジニアがいない場合は `nginx.conf` や `my.cnf` は普段から慣れ親しんでいるものではないと思うので、いくつかテンプレ的な設定をコメントアウトで用意しておき、 必要に応じて編集して使うようにすると時間もかからずお勧めです。

ISUCONの開始前にできる準備は以上になります。 それでは、ISUCON6予選問題をscalaで解いてみたいと思います。

ISUCON6予選問題をscalaで解く

Vagrantで初期環境構築

これまでのISUCON問題をVagrantで構築できるようまとめてくださっています[3]。 ISUCON6は本来azureでVMインスタンスを用意し、ansibleを流して構築する必要があるのですが、今回はこちらを利用して進めていきます。 今回はisucon6-qualifier-standaloneを利用します。

計測用ツールの導入

ISUCONといえばスロークエリやアクセスログの解析、プロファイラを利用したりCPU・メモリ状況を観測してボトルネックを発見する、という部分に面白さがあります。 このツール群については色々な方が触れているので、参考程度に自分が利用しているツールを挙げておきます。

- pt-query-digest
  - スロークエリ解析
  - https://www.percona.com/doc/percona-toolkit/2.2/pt-query-digest.html
- alp
  - アクセスログ解析
  - https://github.com/tkuchiki/alp

その他、 `vmstat` や `top` などを適宜使っています。

scala実装への切り替えとスコア計測

今回の初期実装はperlで動作しており、systemdで管理されています。 scala実装に切り替えた上で初期スコアを計測してみます。

```
$ systemctl disable isuda.perl.service isutar.perl.service
$ systemctl enable isuda.scala.service isutar.scala.service
```

今回はisuda, isutarという2種類のサービスによって構成されているようです。 一旦 `vagrant reload` をかけた上で、ベンチマークをかけてみます。

```
$ ./ISUCON6q-bench -target http://127.0.0.1
2017/01/03 12:07:57 start pre-checking
2017/01/03 12:08:20 pre-check finished and start main benchmarking
2017/01/03 12:08:57 benchmarking finished
{"pass":true,"score":0,"success":121,"fail":57,"messages":["リクエストがタイムアウトしました (GET /)","リクエストがタイムアウトしました (GET /keyword/2089年)","リクエストがタイムアウトしました (GET /keyword/684年)","リクエストがタイムアウトしました (GET /keyword/イギリス政府)","リクエストがタイムアウトしました (GET /keyword/ウーズ)","リクエストがタイムアウトしました (GET /keyword/ジェイアールバス東北大湊営業所)","リクエストがタイムアウトしました (GET /keyword/ナンバーディスプレイ)","リクエストがタイムアウトしました (GET /keyword/フーリエ)","リクエストがタイムアウトしました (GET /keyword/印旛村)","リクエストがタイムアウトしました (GET /keyword/巨大基数)","リクエストがタイムアウトしました (GET /keyword/神戸国際会館)","リクエストがタイムアウトしました (POST /keyword)","リクエストがタイムアウトしました (POST /login)"]}
```

初期スコアは0のようです。ここから改善して0からの脱出を目指します。

ボトルネックを特定する準備

ベンチマーク実行中の `top` をみてみます。

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2607 ISUCON    20   0 2990956 504256  19876 S 169.7 24.6   3:40.39 java
 2606 ISUCON    20   0 2942064 274508  18500 S  52.2 13.4   0:40.19 java
 1282 ISUCON    20   0  118136  10372   5752 S   2.3  0.5   0:02.61 isupam
 2793 root      20   0  139580  87804   7184 S   1.0  4.3   0:01.60 ISUCON6q-bench
 1348 mysql     20   0 1543880 230616  16944 S   0.7 11.3   0:05.30 mysqld
```

`top` の時点ではざっくりとしかわからないのですが、javaプロセスのCPU使用率が高いことからアプリケーションにボトルネックがあることが推測されます。

次に `alp` を用いてアクセス解析を行っていきます。

```
$ alp -f /var/log/access.log --aggregates "/keyword/*" -r
+-------+-------+--------+---------+--------+-------+--------+--------+--------+--------+-----------------------------------+
| COUNT |  MIN  |  MAX   |   SUM   |  AVG   |  P1   |  P50   |  P99   | STDDEV | METHOD |                URI                |
+-------+-------+--------+---------+--------+-------+--------+--------+--------+--------+-----------------------------------+
|    18 | 1.421 | 15.005 | 187.433 | 10.413 | 1.421 | 15.000 | 15.005 |  6.163 | GET    | /keyword/*                        |
|    21 | 2.895 | 15.005 | 265.020 | 12.620 | 2.895 | 15.000 | 15.005 |  3.679 | GET    | /                                 |
|    71 | 0.139 |  3.006 | 167.744 |  2.363 | 0.139 |  3.000 |  3.005 |  1.065 | POST   | /login                            |
|    11 | 0.473 |  3.002 |  18.487 |  1.681 | 0.473 |  0.769 |  3.002 |  1.124 | POST   | /keyword/*                        |
|    19 | 0.207 |  0.616 |   7.287 |  0.384 | 0.207 |  0.321 |  0.606 |  0.139 | GET    | /css/bootstrap.min.css            |
|    10 | 0.115 |  0.600 |   3.258 |  0.326 | 0.115 |  0.289 |  0.445 |  0.145 | GET    | /css/bootstrap-responsive.min.css |
|    10 | 0.151 |  0.583 |   3.504 |  0.350 | 0.151 |  0.311 |  0.546 |  0.132 | GET    | /js/jquery.min.js                 |
|    10 | 0.212 |  0.558 |   3.429 |  0.343 | 0.212 |  0.319 |  0.437 |  0.102 | GET    | /js/bootstrap.min.js              |
|    10 | 0.109 |  0.548 |   3.361 |  0.336 | 0.109 |  0.285 |  0.547 |  0.141 | GET    | /favicon.ico                      |
|    10 | 0.195 |  0.536 |   3.350 |  0.335 | 0.195 |  0.296 |  0.465 |  0.109 | GET    | /img/star.gif                     |
|     1 | 0.330 |  0.330 |   0.330 |  0.330 | 0.330 |  0.330 |  0.330 |  0.000 | GET    | /css/main.css                     |
|     1 | 0.213 |  0.213 |   0.213 |  0.213 | 0.213 |  0.213 |  0.213 |  0.000 | GET    | /js/star.js                       |
|     1 | 0.131 |  0.131 |   0.131 |  0.131 | 0.131 |  0.131 |  0.131 |  0.000 | GET    | /logout                           |
|     1 | 0.121 |  0.121 |   0.121 |  0.121 | 0.121 |  0.121 |  0.121 |  0.000 | GET    | /initialize                       |
+-------+-------+--------+---------+--------+-------+--------+--------+--------+--------+-----------------------------------+
```

どうやら `/` と `/keyword/*` に時間がかかっているようです。 `/keyword/*` はともかく、 `/` の最大時間と最小時間の差が大きい点が気になります。

ボトルネックを特定する

では、実際にコードをみていきます。 上記までの流れで、アプリケーションのエンドポイント2つにボトルネックがある予想を立てているので、コード内のその他の部分については考えずに読んでみます。

```
get("/")(withUserName { maybeUserName =>
  val page = params.get("page").flatMap(_.asInt).getOrElse(1)
  val offset = PerPage * (page - 1)
  val entries = DB.readOnly { implicit session =>
    sql"""
      SELECT * FROM entry
      ORDER BY updated_at DESC
      LIMIT $PerPage
      OFFSET $offset
    """.map(asEntry).list.apply()
  }

  val Some(totalEntries) = DB.readOnly { implicit session =>
    sql"SELECT COUNT(*) FROM entry".map(_.int(1)).single.apply()
  }
  val lastPage = totalEntries / PerPage
  val pages = Seq(1, page - 5).max to Seq(lastPage, page + 5).min

  render("index",
    "user" -> maybeUserName,
    "entries" -> entries.map(e => Map("entry" -> e.toHash)),
    "pages" -> pages.map(p => Map(
      "page" -> p,
      "active" -> (p == page)
    )),
    "prev" -> Option(page - 1).filter(_ > 0),
    "next" -> Option(page + 1).filter(_ < lastPage)
    )
  }
)

get("/keyword/:keyword")(withUserName { maybeUserName =>
  val result = for {
    keyword <- params.get("keyword").toRight(BadRequest()).right
    entry <- DB.readOnly { implicit session =>
      sql"""
        SELECT * FROM entry
        WHERE keyword = $keyword
      """.map(asEntry).single.apply()
    }.toRight(NotFound()).right
  } yield render("keyword",
    "user" -> maybeUserName,
    "entry" -> entry.toHash
  )
  result.merge
})
```

entryの取得が様々な場所で行なわれているのが気になりますが、javaプロセスのCPU負荷が高いのでSQLは今ボトルネックになっていないはずです。 上記の処理で気になるのは、標準ライブラリの処理でなく且つ両方のエンドポイントの処理で使われている `.toHash` の部分です。

```
implicit class EntryOps(private val entry: Model.Entry) extends AnyVal {
  def toHash: Map[String, Any] =
    toJSON(entry).extract[Map[String, Any]] ++ Map(
      "url" -> s"/keyword/${entry.keyword.uriEncoded}",
      "html" -> htmlify(entry.description),
      "stars" -> loadStars(entry.keyword).map(s => Map("star" -> s))
    )
}
```

`.uriEncoded`, `htmlify`, `loadStars`が改善対象になりそうです。

これらについて改善の優先度をつけるには、プロファイラを用いて処理時間の測定を行う必要があるのですが、プロファイラの導入がどうしてもうまくいかなかったので諦める判断を下しました。 繰り返しになりますがISUCONは全体で8時間しかないため、うまくいかない部分について早めに見切りをつけることも必要です。 この3つの処理について、コードから読み取れることを列挙してみることにします。

- .uriEncoded
  - 文字列をUTF-8でURLエンコードする
  - RFC3986に基づいて`+`と`%7E`をエスケープする
- htmlify
  - DB上の全entryのキーワードのいずれかにマッチさせられる正規表現を作る
  - 与えられた文字列の中から、キーワードにマッチした部分を対応キーワードのページへのリンクに置き換える
- loadStars
  - `isutar/stars`に対して、キーワードをパラメータに含めてHTTP#GETのリクエストを送信する
  - `isutar/stars`はDBからキーワード指定でstarを取得して返す
  - 返ったstarをOptionで受け取り、安全な形で返す

`loadStars`がHTTPリクエスト越しに処理を行っているのが気になりますが、リクエスト先ではstarを取得しているだけなのでjavaプロセスのCPU負荷には関係がないはずです。 `uriEncoded`も比較的短い文字列であるキーワードに文字列置換を行うだけなので、さほど負荷はないと思われます。 残ったのは`htmlify`ですが、長い文字列である説明文に対して膨大な正規表現でマッチをかけているので、いかにもCPU負荷が高そうです。

実際ISUCON6出題者側の意図したボトルネックとして`htmlify`が用意されており、これをどう改善するかというのが肝になっていました。今回は`htmlify`を改善してみます。

ボトルネックを改善する

`htmlify`のコードは以下の通りです。

```
def htmlify(content: String): String = {
  val entries = DB.readOnly { implicit session =>
    sql"""
      SELECT * FROM entry
    """.map(_.toEntry).list.apply()
  }
  val regex =
    entries.map(e => Pattern.quote(e.keyword)).mkString("(", "|", ")").r
  val hashBuilder = Map.newBuilder[String, String]
  val escaped = regex.replaceAllIn(content, m => {
  val kw = m.group(1)
  val hash = s"isuda_${sha1Hex(kw)}"
  hashBuilder += kw -> hash
    hash
  }).htmlEscaped
  hashBuilder.result.foldLeft(escaped) { case (content, (kw, hash)) =>
    val url = s"/keyword/${kw.uriEncoded}"
    val link = s"""<a href="$url">${kw.htmlEscaped}</a>"""
    content.replaceAllLiterally(hash, link)
  }.replaceAllLiterally("\n", "<br />\n")
}
```

この処理の問題点を考えてみます。

1. 処理の都度、DBのentryテーブルから全取得して正規表現を生成している
2. 正規表現にマッチするキーワードの抽出とリンクへの置き換えを別々の反復で行っている

1についてはメモリ上に載せてしまうのが良さそうです。scalaにはシングルトンを保証する`object`が用意されているので、以下のような`EntryRegex`オブジェクトを実装します。

```
object EntryRegex {
  import Model.WrappedSetConvert

  private var value: String = ""

  def initialize(): Unit = {
    val entries = DB.readOnly { implicit session =>
      sql"""
        SELECT * FROM entry
      """.map(_.toEntry).list.apply()
    }
    value =
      entries.map(e => Pattern.quote(e.keyword)).mkString("(", "|", ")")
  }

  def regex: Regex = value.r
}
```

キーワードの追加・削除・更新があった時にどうするか？という問題が出てきますが、既存のコード上にあるDBへの処理の後に都度 `initialize` を実行することにします。 実行タイミングによっては整合性の取れていない値を返すことがありそうですが、ISUCONなので時間もないですし、問題が起きてから考えれば良いです。

1, 2の改善後の `htmlify` は以下の通りです。

```
def htmlify(content: String): String = {
  val regex = EntryRegex.regex
  val hashBuilder = Map.newBuilder[String, String]
  regex.replaceAllIn(content.htmlEscaped, m => {
    val kw = m.group(1)
    val url = s"/keyword/${kw.uriEncoded}"
    val link = s"""<a href="$url">${kw.htmlEscaped}</a>"""
    link
  }).replaceAllLiterally("\n", "<br />\n")
}
```

だいぶすっきりした気がしますが、実際の処理は改善されたでしょうか？ 再度、計測してみることにします。

```
$ ./ISUCON6q-bench -target http://127.0.0.1
2017/01/03 16:28:45 start pre-checking
2017/01/03 16:28:52 pre-check finished and start main benchmarking
2017/01/03 16:29:45 benchmarking finished
{"pass":true,"score":3154,"success":1198,"fail":1,"messages":["荒川道路 に 中部地方の道路一覧 へのリンクがありません (GET /)"]}
```

エラーが発生しているようですが、**無事スコア0から脱出できました。** 本番のインスタンスはVagrant環境より性能が良いので、この内容でももっとスコア出るのではと思います。

あとがき

2015年, 2016年とISUCON予選に挑戦してきましたが、ベンチマークを回すまでミスに気がつきにくいのが辛い部分でした。 特に本番はベンチマークを外部から実行する形式のため、長いベンチマーク実行待ちの後にエラーが吐き出されるとそれだけで気力が吸われます。 単体テストが用意されていないISUCONに対して、コンパイル時にある程度ミスを発見しつつガシガシ書けるのは嬉しさがあります。

2015年時点ではISUCONはマイナーなコンテストなのかな、と思っていたのですが、最近はよく名前も聞くようになり参加者も大幅に増え、 社内ISUCONを開催する会社もでてきたそうです。

2017年のISUCONはscalaで決勝進出者が出てくるといいですね。

参考

[1]ISUCON6予選について: http://isucon.net/archives/48396740.html

[2]ISUCON6予選言語比率: http://isucon.net/archives/48501097.html

[3]VagrantでのISUCON環境の提供: https://github.com/matsuu/vagrant-isucon

《東 優太》

付録：セプテーニ・オリジナル 技術指針

序文

この文章は、セプテーニ・オリジナルに所属する企画者・エンジニアの指針となるよう、2016年5月当時の我々の目指すところ／我々が取っている行動の理由／我々が求める姿勢、を文章化したものです。

我々は”高速高品質”を目指す

システム開発は機を逸さないために開発速度が重要です。かといって品質を顧みなさすぎても不幸を生み出します。我々は高速に高品質なシステムを開発できる状態を目指します。

初回リリース最速を目指すことは意味しません。初期に作成する部分は最も下地となる部分であり、土台です。土台が腐ると、その上位レイヤーも腐ります。取り返しも付かないことが多いので丁寧に作成しましょう。

我々は保守性のあるコードを高速に生産する

“品質”と一口に言っても国際規格が存在するくらい複雑ですが、我々は品質の中でも”保守性”を重要視します。

保守性の低いコードを保守しようとすると、開発者の肉体や精神の不健康を招き、酷くなると休職・離職に繋がります。ビジネス面においてもフットワークは悪くなり、徐々に貢献できなくなってくるどころか、障害による損失・運用でカバーの多発によりマイナス影響を与えることも起こりえます。

低い保守性のコードは追記された部分も悪くなりがちなうえ、前例に従うつもりでコピペされることも多いので、いつの間にか増殖します。後から改善することも困難です。早めに対応しましょう。

採用活動にも影響があります。良いコードを書いていればPRになりますが、悪いコードが蔓延していると好んで劣悪な環境に入りたがる人は居ないため、人材の確保が難しくなります。

いずれも会社の存続にも影響する問題であることから、我々はまずは保守性があり、それを保てるコードを高速に生産できる状態を目指します。

具体的に何をするのか

高速高品質を目指すため、効率が良い道具とプロセスを選び、良い状態を作れるよう努力をして、良い状態を作り、それを保ちます。

Scalaを使う理由は気軽に品質を維持するため

我々はScalaを積極的に使用します。

Scalaは表現力に富み、手慣れた状態であれば労力少なく意図する機構を記述することができます。また型がありIDEも発展していることから、リファクタリングを行いやすく品質維持の労力が少ない言語です。

学習コストは小さくありませんが、何ヶ月もかかるものではありません。Javaの資産を活かせる点も実用的です。現時点では意識の高い人材との出会いを増やす効能もあります。

Scalaを利用しない場合

バッチなど、小さなプログラム、もしくは完全な使い捨てなプログラムを作成する場合、Scalaは利用しません。BashなりPythonなりPHPなり、状況に応じて使いやすい物を利用します。

またユーザーに配布する必要がある、といった事情でフットプリントを大きく出来ない場合も利用しません。GoやC#、Excel VBAなどユーザー利便性と開発効率・保守性を天秤にかけて利用言語を選定します。

iOS開発や古いシステムの保守、目的とする環境でScalaが利用できない場合も他の言語を利用します。ただし、目的のライブラリが特定言語の物しかない、といった場合であればマイクロサービス化して分離を試みる、ラッパーを開発する、といった手法を検討する余地があります。

マイクロサービスを導入する場合、試行のしやすさを目的にレンダリング部分だけPHPに担当させるといったことも選択に入ります。

学習コストや人材調達の都合を言い訳にして、普及した言語を選択することは行いません。

なお目的を鑑み、より優れた言語が登場した場合は、それに乗り換えることが推奨されます。

スクラムを採用する理由はプロダクトオーナーがハンドルを握るため

我々はスクラムを積極的に採用します。

開発においてリソースが潤沢であることは希で、殆どの場合、限られたリソースの振り分け判断が重要となります。

スクラムはバックログを通してプロダクトオーナーの意図を表せますし、スプリントとストーリーポイントにより、コストも根拠を持って可視化することができ判断に活かせます。

これ以外にも、顧客意見反映を手続き化・迅速化できる／進捗確認や異常の検出できるタイミングが多い／ベロシティにより完成予想に根拠を持たせられる、といったメリットがあります。

スクラム以外を使う場合

不具合や割り込みが多すぎる場合、スプリントが成り立たないためカンバンを利用します。プロダクトが大きい場合は保守チーム・開発チームといったようにチーム別けをして制度を変える手法も検討できます。

削りようが無い最小要件の物を最速で作るといったように、プロダクトオーナーの意図を入れようが無い場合も、検討会・反省会の時間が無駄であるためカンバンを利用します。この場合、事態が落ち着き次第、速やかにスクラムに移行することが推奨されます。

開発チームの外に予定を説明する場合はガントチャートが便利です。内部管理には利用すべきではありません。目安以上の精度は出しづらく、スケジュールに合わせて実装を曲げる事態を招きます。

DDDを採用するのはチーム全員で要件の芯を捉え続けるため

我々はドメイン駆動設計を積極的に採用します。

ドメイン駆動設計はユビキタス言語という言葉と実装を一致させる言語を用いてビジネス要件の芯を設計の中核におく手法です。「エンジニア以外にも翻訳不要で伝えることができる」「設計と実装に対する価値観が共有できる」という特徴があり、チーム全員で要件の芯を捉える助けになることを期待しています。

具体的には、やらねばならないことのコストをプロダクトオーナーが認識しやすくなる／レビューや仕様検討時に、言葉と実装が一致しているか、というチェックを行えるようになる／設計や概念がまずい場合、言葉にすると自然と大仰に成り危険な感じになる、といった効能を期待しています。

なお、”レイヤードアーキテクチャを採用する”ではないため、上記効能を維持出来るアーキテクチャであればどの手法でも構いません。

DDDを採用しない場合・崩す場合

小さなプログラム、もしくは完全な使い捨てのプログラムの場合、DDDは採用しません。

マイクロサービスにおいては、特定サービスの内部だけ採用しない、といった選択もあり得ます。

既存コードへの改修や、フレームワークなど開発環境の仕様が特殊である、など導入に無理が出過ぎる場合も採用しないか、崩すことを検討します。

プロトタイプ作成時にも採用しません。プロトタイプは全てを割り切って高速に具現化することに価値があり、丁寧に設計することは目的と反します。調査が終わったら本番コードに流用せず、必ず破棄してください。

なお、この文章でのプロトタイプとは中身は仮実装だが発注者が実装するものの具体的イメージを掴む助けになり、使い勝手も検証できる程度のもの、を差します。

ユニットテストを書く理由はリファクタリングをし続けるため

我々はユニットテストを書きます。

改修を続けるシステムにおいて、仕様変更に伴い既存コードがベストな設計で無くなる場面が発生します。この際、保守性を維持するためには既存コードも理想と思える設計に書き換える—リファクタリングをする必要が有ります。

テストがあれば”書き換えたが動かしたことが無い”領域が減り、安心度が高まります。テストが無い場合、只の作り直しであるためリスクが大きく既存コードの改修は憚られる場合が発生します。これが積み重なると設計の歪みが大きくなり、保守性の大幅な低下を招きます。

またユニットテストは頻繁に動作確認ができる作業台としての役割も期待します。E2EテストやF5(再読込)駆動開発では全てを結合しなければ動作確認ができませんし、場合分けをカバーしきるのは難しいですが、ユニットテストであればメソッド単位で確認を行うことができます。細かな場合分けにも対応できます。

我々はTDDを標準とはしませんが、その心は参考にし、ある程度小さいまとまりでテストを動かしましょう。テストを書くことで自分の書いたコードの最初の利用者が自分となりましょう。使い勝手の改善や、不具合検出の助けとなります。

テストを書く＝遅い、という思い込みを止めよう

テストもコードを書く以上、時間を消費するのは確かですが、作業台として活用をすれば確認効率も高まるので、慣れ次第ではF5駆動開発より素早く開発できるかもしれません。

仕様変更を行ったときの確認項目の削減効果も大きいので、長期的に見て速度を落とさないための手段、と考えても良いでしょう。

カバレッジ100%を目指す、といったテストを書きすぎる方向に陥ると本当に遅くなるので注意しましょう。テストの密度や品質は保守性の維持を念頭に置きつつ、適度に省きましょう。他のテストにより安心できる部分は省くことが推奨されます。テストはよりカバー率の高くなる下位層の方—レイヤードアーキテクチャであればDomain層／Infra層、の方を充実させましょう。

テストを書かない場合

バッチなど小さなプログラム、もしくは完全に使い捨てであることが確定しているプログラム、にはテストを書かなくても十分な場合が多いでしょう。

またプロトタイプに対しても前述の理由でテストは書くべきではありません。

レビューを行うのは意識を保ち続けるため

我々はコードレビューを行います。

一人で綺麗な設計・コードを書き続けるのは難しく、他人の目と意見があってこそ行うことが出来ます。鉄の自制心を持ったメンバーでチームを固めているわけでもなければ、レビュー無しに品質を保ち続けるのは困難でしょう。

レビューを行い続けるため、属人性を高めない

一人開発、もしくはサーバをいじれる人一人・クライアントをいじれる人一人、というようにスキルセットで担当分けをすると相互にレビューをすることが難しくなります。このような属人性を高める体制は避けるべきです。

可能であればサーバ側2名・クライアント2名、もしくはサーバとクライアントを両方扱える者1名を混ぜる体制にしてレビュー品質の確保を組織レベルで心がけましょう。

それでも1名〜2名で始めざるを得ない場合は発生しますが、この状態を長引かせると属人性が高まりすぎてしまい、健全なレビュー体制に戻せなくなります。早めにタスクローテーションを行う、もしくは二人でサーバを作ってから二人でクライアントを作成する、等の工夫で属人性が高まらないよう注意をしましょう。

レビューを行わない場合

よほどの緊急事態でなければ、レビューを行わない理由はありません。

そもそも一人開発である、専門性が高く自分しか見ることが出来ない、等の問題でレビューを行えない場合は有ります。これは組織の問題なので、組織（あるいはチーム）で解決を図ることになります。具体的には単独開発にならないアサインをする・専門家を育てる、といった策になります。

プロダクトオーナーに求めることは採配

プロダクトオーナーには市場で生き残れる製品を作る責任がありますし、チームが生存できる衛生を保つ義務があります。

バックログ＝限られたリソースの振り分け、を通して義務の全うを目指すことになりますが、ステークホルダーに引っ張られ、チームが破綻する状況を招いてはいけませんし、チームを大事にしすぎてステークホルダーが満足しない状況を招いてもいけません。

限られたリソースの中で膨大なストーリーをこなすことになります。それぞれのストーリーの目的を把握し、優先度を割り出し取捨選択し、自信と根拠のある選択をチームに示す姿勢が求められます。

とくに技術的負債が残る不完全な工法を選択する、といった危険な選択を行う場合、それを本当に行うべき状況なのかの確認、関係者全員へのリスクの共有と覚悟の取りつけ、残る影響の解消計画の立案、などの行動が求められます。

衛生を保つため、開発メンバーから上がってくる技術的負債の返済やライブラリバージョンアップ等のストーリーも状況を鑑み適切にバックログに盛り込む必要があります。

ステークホルダーから受注をする、というスタイルの場合、ステークホルダーが何をやりたいのかを具体的に想像できているとは限りません。話を分析し、より効率的なストーリーを編み出す手腕も求められます。

スクラムマスターに求める事は円滑

スクラムマスターとはスクラムを円滑に回すための様々な事を行う役割です。

具体的には、各種会議の司会を行う／進捗が不穏であればヘルプを行ったり都合したりする／チームが緩んでいれば発破を掛ける／プロダクトオーナーを教育する／実装時間をより確保できるよう工夫をする／検討会・反省会が長くなりすぎないよう注意する／会議が増えすぎないよう注意する／属人性が高まらないアサインを考える／技術的負債を解消するス

トーリーの提案／組織的問題があれば上長と相談する、などの行動が求められます。

プロダクトオーナーとは利益が相反するため、兼任はできません。

メンバーに求めることはチームでの達成

スクラムに参加するエンジニアには、チームで達成できるように振る舞うことが求められます。チームがスプリントを達成できるよう自分の分を頑張る（ただし抱えすぎ厳禁）のは当然として、チームを助ける・頼る視点が重要です。

スクラムでは自己組織化を目指します。スクラムルールの原点である"スクラムガイド"では"自己組織化チームは作業を成し遂げるための最善の策を，チーム外部からの指示ではなく，自らが選択する"と記述されています。

チームで進捗の厳しい仲間が居た場合、アドバイス等を行うほか、落ちそうなストーリーがチームにとって重要であれば、自分の分を落としてでも巻き取りに行くことがより良い結果に繋がります。逆に自分の頑張りが及ばす、達成できなさそうであれば早めにチームに助けを求めることがより良い結果に繋がります。

おわりに

この文章に記載されている具体的技術（言語・設計手法等）はすぐに陳腐化すると予想されるため、拘り続ける意味はありません。しかし、この文章に記載されている考え方は時間が経っても大きくは変わらないと思われます。常に心に留め、研鑽を積みましょう。

PHILOSOPHY 企業理念

行動規範
Code of conduct

コードは**やさしくあたたかく**
今日の仕事が**人生の自己ベスト**
失敗に学び**成功に飢えろ**
怠けたいなら**教育を怠けるな**
勝ちたいなら**価値あるものをつくれ**
期待値を知り**期待値を超えろ**
勉強しよう**そうしよう**
最適な技術が**最高の選択肢**
計画的に休み**計画的に作る**
Thank God Let's Code!

コア・バリュー
Core Value

匠 Excellent
私たちは、
洗練された匠の技術で
優れた価値を
提供します。

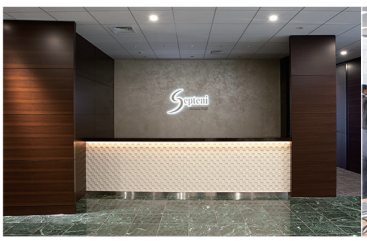
先進 Advance
私たちは、
先進的な技術に
挑戦します。

充実 Life
私たちは、
公私ともに人生を
充実させることを
大切にします。

RECRUITMENT 募集要項

ポジション	アプリケーションエンジニア　フロントエンドエンジニア　インフラエンジニア
仕事内容	インフィード広告を中心としたアドテクノロジーサービス《PYXIS》の設計および開発。 アプリストア最高峰のオリジナル新作マンガ配信サービス《GANMA!(ガンマ)》の設計および開発。
応募要件	ソフトウェア品質維持に対する意欲 新規の言語・技術習得に対する意欲（特にScalaでの実務経験がある方、または経験したい方） ユニットテストが書ける方、または書きたい方 スクラム開発の経験がある方、または経験したい方
勤務地	東京都新宿区西新宿 8-17-1　住友不動産新宿グランドタワー 27F
勤務体系／休暇	裁量労働制／年間休日122日　※2年目以降 132 日（夏季休暇10日）
お問い合わせ先	career-recruitment@septeni.co.jp　※どんなことでもお気軽にご連絡ください！

OFFICE TOURS 定期開催中のセプテーニオフィスツアー！ ▶▶▶ http://tour.septeni.net/



Septeni Original

セプテーニ・オリジナルの採用に興味がある方はこちらからエントリー

http://www.septeni-holdings.co.jp/career/

【HP】http://www.septeni-original.co.jp/　【Twitter】@SEP_Engineer　【FB】SepteniOriginal,inc.