Tutorial of Gradle

Hydroxide

Not Broccoli, Abstraction of triangular pyramid.

Abstract

本書では Gradle の使い方について説明します。動作を検証した環境は Linux (Ubuntu 14.04) と Cygwin (mintty 2.1.5, Windows8.1) ですが、他の環境にも応用できると思います。

Contents

1 Gradleのセットアップ

1.1 Javaの確認

Gradle には Java JDK が必要なので、まずはこれが入っているか確認します。確認のためにターミナルで

Code 1: Java のバージョン確認

```
$ java -version
```

と打ち込みます。Java のバージョンは 1.8.0（Java8）以降であることが前提なので、Java が新しくなければ更新します。

1.1.1 Ubuntu14.04 の場合

本節は Ubuntu14.04 での jdk のセットアップ方法です。手軽に入手できる OpenJDK(version 11.0) を導入します。以下のコマンドは ubuntu のパッケージマネージャー apt を用いた方法です。RedHat 系、Cygwin、MinGW、および Mac の場合は異なります。情報提供求む。

Code 2: OpenJDK11.0 のインストール (Ubuntu14.04 の場合)

```
$ sudo apt-add-repository ppa:openjdk-r/ppa
$ sudo apt-get update
$ sudo apt-get install openjdk-11-jdk
```

再度バージョン確認して新しくなっていれば完了です。Gradle のインストール作業に移ります。

1.1.2 Cygwin その他 Windows 上のエミュレータの場合

OpenJDK をダウンロードしてきて好きな場所に解凍してください。ただし、Java のパスの設定は

Code 3: JDK の PATH の設定

```
$ export JAVA_HOME="/full/path/to/jdk-*.*.*"
$ export PATH="${JAVA_HOME}/bin:${PATH}"
```

としてください。指定した JDK が最優先になるように、PATH の先頭に追記します。また JAVA_HOME の末尾に/を入れると Gradle がエラーを返すので、入れないよう注意してください。JAVA_HOME はフルパスでないと正常に参照されません。

1.2 Gradle の手動インストール

1.2.1 Ubuntu14.04 の場合

Gradle のホームページ（https://gradle.org/install/）にアクセス。Installing manually 節の、’・Complete, with docs and sources’ から zip ファイルをダウンロードし、任意の場所に解凍します。その後以下の通り PATH を設定します。

Code 4: PATH の設定

```
$ unzip gradle-*.*.*-all.zip -d "/arbitrary/path/"
$ export GRADLE_HOME="path/to/gradle-*.*.*/"
$ export PATH="${PATH}:${GRADLE_HOME}bin"
```

またホームディレクトリ下にある’.bash_profile’ あるいは’.profile’ の末尾に上の export 文２つを書き足すと次回のログインから自動的に環境変数が設定されるので、登録しておくと便利です。PATH を通したら

Code 5: セットアップの確認

```
$ gradle -v
```

と打ち込みます。筆者の環境では

Code 6: Gradle のバージョン表示 (例)

```
------------------------------------------------------------
Gradle 5.4.1
------------------------------------------------------------

Build time:   2019-MM-dd hh:mm:ss UTC
Revision:     **********************

Kotlin:       1.3.21
Groovy:       2.5.4
Ant:          Apache Ant(TM) version 1.9.13 compiled on July 10 2018
JVM:          11.0.1 (Oracle Corporation 11.0.1+13-Ubuntu-3ubuntu114.04ppa1)
OS:           Linux 3.19.0-32-generic amd64
```

となります。似たようなものが表示されたらインストール完了です。

次に、練習用のプロジェクトを格納する作業フォルダを作ります。名前と場所は自由でいいです。筆者は Documents の中に gradle_study を作ったので

Code 7: 作業フォルダの作成

```
$ cd ~/Documents
$ mkdir gradle_study
$ cd gradle_study
```

となります。本書で作成するプロジェクトは皆この中に格納します。プロジェクトの作成に関しては、自動で作成する方法

Code 8: 楽な方法

```
$ gradle init --type java-application
```

がありますが、ここでは gradle の理解のために手動でプロジェクトを作っていきます。

1.2.2 Cygwin の場合

本節では Windows で Gitbash を用いた場合の、Gradle のインストール手順を解説します。なので Cygwin は入っていることが前提となります。また本節の手順は Gitbash(MINGW) でも使えると思います。Cygwin で Gradle を動かすと若干重たいです。

まずはダウンロードして解凍までした (open)JDK と gradle（ZIP ファイルは前節、Ubuntu の場合と同様に https://gradle.org/install/からダウンロード）を好きなところに置きます。本書ではホームディレクトリ下に JavaTools というフォルダを作り、その中に入れておきます。あとは前節と同じです。上で設定した JAVA_HOME と GRADLE_HOME、PATH については、ホームディレクトリの.bash_profile に export 文を追記しておけば次回起動時から自動で環境変数がセットされます。

2 Gradle の練習

2.1 練習 1：compileJava

ここから先は、前章で作った作業フォルダ内での操作です。まずプロジェクト一号を格納するフォルダを作ります。名前は自由ですが、本書では case1 とでもしておきます。この中に Java のソースコードと build.gradle を配置します。ソースコードの配置場所は、自動生成した時のディレクトリ構造に従います。すなわち、src/main/java/下にソースコードを置きます。メインクラスを Case1 とすると、

Code 9: プロジェクトの作成

```
$ mkdir case1
$ cd case1
$ mkdir -p src/main/java
$ touch build.gradle src/main/java/Case1.java
```

となります。Case1.Java と build.gradle を作成したら、それぞれに内容を書き込んでいきます。Case1.java については今回は重要ではないので、とりあえず

Code 10: Case1.java

```
public class Case1{
    public static void main(String[] args){
        System.out.println("Hello Tortoise!");
    }
}
```

とでも書き込んでおきます。一方の build.gradle には、

Code 11: build.gradle

```
plugins {
    id 'java'
}

compileJava {
    options.encoding = 'UTF-8'
}
```

と書き込みます。これが、Java のコンパイルのための必要最小限の記述です。gradle で Java をコンパイルすること、文字は UTF-8 でエンコードすること以外の指示はしていません。これでコンパイルはできます。プロジェクトフォルダの中で（build.gradle と同じ階層で）

Code 12: gradle compilejava

```
$ gradle compileJava
```

と入力すれば Gradle が動き始めます。最初に起動したときのみ、初期化処理のため少々時間がかかります。「BUILD SUCCESSFUL」と出れば成功です。フォルダ内に build フォルダができ、build/classes/java/main 下にコンパイルされたクラスファイルができているはずです。以下のようにして実行ができます。

Code 13: 起動確認

```
$ java -cp build/classes/java/main/ Case1
```

2.2 練習 2：build

前節では Gradle を使って Java のコンパイルをしてみました。つぎに、Gradle のビルドを試してみます。case1 フォルダから抜けて作業フォルダ直下へ戻り、プロジェクト第二号を作ります。名前はもちろん自由ですが、本書では case2 とします。Gradle build の勉強のため、今度は成果物を複雑にしてみます。

Code 14: プロジェクト 2 の作成

```
$ mkdir case2
$ cd case2
$ mkdir -p src/main/java/controller
$ mkdir -p src/main/java/textloader
$ mkdir -p src/main/resources
$ touch build.gradle
$ touch src/main/resources/tortoise.txt
$ touch src/main/java/controller/Controller.java
$ touch src/main/java/textloader/TextLoader.java
```

成果物の機能は単純です。controller パッケージの Controller クラスに main メソッドを置き、main メソッドから textloader パッケージの TextLoader クラスを呼び出します。TextLoader クラスには、テキストファイル (.txt) の内容を読み取ってコンソールに出力する機能を実装し、アクセスしたいテキストファイルを src/main/resources 下に置きます。今回の例では、読み出したいテキストファイルを以下のようにします。

Code 15: src/main/resources/tortoise.txt

```
Testudo graeca         : Greek tortoise       : ギリシャリクガメ
Testudo hermanni       : Hermann's tortoise   : ヘルマンリクガメ
Agrionemys horsfieldii: Russian tortoise      : ヨツユビリクガメ
Geochelone elegans     : Indian star tortoise : インドホシガメ
Geochelone platynota   : Burmese star tortoise: ビルマホシガメ
Pyxis arachnoides      : spider tortoise      : クモノスガメ
```

controller.Controller.java は以下のとおりです。

Code 16: src/main/java/controller/Controller.java

```
package controller;

import textloader.*;

import java.io.*;

public class Controller{
    public static void main(String[] args) throws IOException{
        TextLoader loader = new TextLoader("tortoise.txt");

        loader.Show();
        loader.Close();
    }
}
```

textloader.TextLoader.java は以下のとおりです。

Code 17: src/main/java/textloader/TextLoader.java

```
package textloader;

import java.io.*;
import java.net.*;

public class TextLoader{
    BufferedReader buff;

    public TextLoader(String arg) throws IOException{
        this.buff = this.Connect(arg);
    }

    private BufferedReader Connect(String path) throws IOException{
        BufferedReader result;
        URL url = TextLoader.class.getClassLoader().getResource(path);
        if (url == null){
            System.out.println("error: cannot open "+path);
            return null;
        }

        try{
            result = new BufferedReader(
                new InputStreamReader(url.openStream())
            );

            return result;

        }catch(FileNotFoundException e){
            System.out.println("File "+path+" is not found.");

            return null;
        }
    }

    public void Show() throws IOException{
        String entry;

        if (this.buff == null){
            return;
        }

        while( (entry=this.buff.readLine()) != null ){
            System.out.println(entry);
        }
    }

    public void Close() throws IOException{
        if (this.buff != null){
            this.buff.close();
        }
    }
}
```

さて問題の build.gradle は

Code 18: build.gradle

```
plugins {
    id 'java'
}

compileJava {
    options.encoding = 'UTF-8'
}

sourceSets {
    main {
        java {
            srcDirs = ["src/main/java"]
        }

        resources {
            srcDirs = ["src/main/resources"]
        }
    }
}
```

となります。前節から増えた部分は

Code 19: build.gradle の追加項目

```
sourceSets {
    main {
        java {
            srcDirs = ["src/main/java"]
        }

        resources {
            srcDirs = ["src/main/resources"]
        }
    }
}
```

です。main{. . .} は src/main/. . . に対する処理を指定します。後々テストと本番を分けるために src/test/. . . が作られることになるので、このような括りが存在します。srcDirs=. . . の行では、Java ソースコードとリソース（前述の仕様で言えば TextLoader に渡すテキストファイル）の参照先をそれぞれ指定します。前節ではこれを書きませんでしたが、参照先がデフォルトの src/main/java、src/main/resources となっていれば明記は必要ないようです。でも明記しておいたほうがいいと思います。

仕込みは以上です。Gradle を動かしてみます。前節と同じく build.gradle と同じ階層で

```
$ gradle compileJava
```

と打ってコンパイルします。うまく行けば build フォルダが作られ、build/classes/java/main 下に各パッケージの.class ファイルが出来上がっています。複数のパッケージがある場合でも、Gradle はまとめてコンパイルしてくれます。しかし、このままでは不完全です。試しに

```
$ java -cp build/classes/java/main/ controller.Controller
```

と打っても、テキストの読み込みに失敗します。この段階ではまだリソースファイル (テキストファイル) は組み込まれていないからです。

そこで、gradle compileJava の代わりに

Code 20: gradle build

```
$ gradle build
```

と打ってみます。すると、build 下にフォルダ libs が作られ、更にその中に jar ファイルが作られます。この jar ファイルには、出力されたリソースファイルと.class ファイルがまとめられています。なので、

```
$ gradle build
$ java -cp build/libs/case2.jar controller.Controller
```

と打てば正常に動作します。この jar ファイルが成果物になります。

この jar ファイルの生成についても build.gradle で設定できますが、それは次節で説明します。

2.3 練習 3: jar

次はビルドの設定、特に jar 作成のオプションをいじります。例によって case2 を抜けて作業フォルダ直下に戻ってください。3 つ目のプロジェクトは前節のプロジェクト 2 号を修正したものになります。なので

Code 21: プロジェクト 3 の作成

```
$ cp -r case2 case3
$ cd case3
$ gradle clean
```

とします。gradle clean は build フォルダを消去するだけのコマンドです。加筆は build.gradle のみで、

Code 22: case3/build.gradle

```
plugins {
    id 'java'
}

def mClass = 'controller.Controller'
def projectName = 'case3'

compileJava {
    options.encoding = 'UTF-8'
}

sourceSets {
    main {
        java {
            srcDirs = ["src/main/java"]
        }

        resources {
            srcDirs = ["src/main/resources"]
        }
    }
}

jar {
    manifest {
        attributes('Main-Class':mClass)
        baseName projectName
    }
}
```

とします。加筆部分は

Code 23: build.gradle の加筆部分 1

```
def mClass = 'controller.Controller'
def projectName = 'case3'
```

と

Code 24: build.gradle の加筆部分 2

```
jar {
    manifest {
        attributes('Main-Class':mClass)
        baseName projectName
    }
}
```

になります。一つめの def 文は変数の定義です。メインクラスとプロジェクト名を定義しています。二つ目は jar のマニフェストファイルの作成です。メインクラスの居場所を jar 内の MANIFEST.MF に書き込んで、jar ファイル単体で実行可能な状態にします。この際、先ほど定義した変数を使用しています。これで jar の設定ができたので、

```
$ gradle build
$ java -jar build/libs/case3.jar
```

で動作確認ができます。これで java コマンドにクラスパスとメインクラス名を書く必要がなくなります。そのため出来上がった jar ファイルを任意の場所に移動させても、java -jar {name}.jar で簡単に実行できます。ゲームを作りたいときにもおすすめ。

なお、コンパイルと jar の生成を別々に行いたいときは、gradle build を使わずに

```
$ gradle jar
```

を使うといいです。

2.4 練習 4: run

これまではプロジェクトをビルドした後、java コマンドで動作確認をしていました。しかしもっと楽な方法があります。それが gradle run です。これでプロジェクトを実行できます。そのためにまた build.gradle を修正します。例によって作業ディレクトリ直下に戻って

Code 25: プロジェクト 4 の作成

```
$ cp -r case3 case4
$ cd case4
$ gradle clean
```

を入力します。

また今回の build.gradle は

Code 26: case4/build.gradle

```
plugins {
    id 'java'
    id 'application'
}

def mClass = 'controller.Controller'
def projectName = 'case4'

compileJava {
    options.encoding = 'UTF-8'
}

sourceSets {
    main {
        java {
            srcDirs = ["src/main/java"]
        }

        resources {
            srcDirs = ["src/main/resources"]
        }
    }
}

jar {
    manifest {
        attributes('Main-Class':mClass)
        baseName projectName
    }
}

run {
    standardInput = System.in
}

mainClassName = mClass
```

となります。加筆部分は最初の plugins 中の

Code 27: build.gradle の加筆部分 1

```
id 'application'
```

と末尾の

Code 28: build.gradle の加筆部分 2

```
run {
    standardInput = System.in
}

mainClassName = mClass
```

です。id 'application' の追加で gradle が実行のための準備を行ってくれます。最終行の mainClassName でメインクラス名を指定し（jar の設定とは別です）、run {...} で細かい設定を書きます。standardInput = System.in はコマンドライン入力をしたいときに付け加えます。今は使っていませんがこれから使いたい人のために書いておきます。

Gradle の基本的な操作は以上です。

3 Junit の利用

3.1 簡単な単体テスト

Gradle などのビルドツールを使えば、単体テストも比較的楽に行えます。というわけで、本章ではプロジェクトのテストについて解説します。Gradle で JUnit を利用する際、JUnit のバージョンによって扱いが変わります。Gradle4.6 以降では、JUnit が標準でサポートされているようです。そのため、最新版の Gradle では扱いが少し楽になります。

例によってプロジェクト case5 を作ります。作業フォルダ直下で

```
$ cp case4 case5
$ cd case5
$ gradle clean
```

とします。また今回から、src/以下に test フォルダを作ります。

```
$ mkdir -p src/test/java
```

この src/test/java 以下にテストコードを格納します。試しに

Code 29: src/test/java/textloader/TextLoaderTest.java

```
package textloader;

import java.io.*;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class TextLoaderTest{
    @Test
    public void test() throws IOException{
        TextLoader loader = new TextLoader("");
        assertNotNull(loader);
    }
}
```

を作ってみます。テストコードの命名規則については、'ClassName.java' という名前のクラスファイルに対しては'ClassNameTest.java' として、パッケージ名や java/以下のフォルダ構成もテスト対象と一致させます。JUnit については、JUnit5 以降は JUnit jupiter なるものを使います。そのためインポートするパッケージは org.junit.jupiter となります。

今回の build.gradle は

Code 30: case5/build.gradle

```
plugins {
    id 'java'
    id 'application'
}

def mClass = 'controller.Controller'
def projectName = 'case5'

compileJava {
    options.encoding = 'UTF-8'
}

sourceSets {
    main {
        java {
            srcDirs = ["src/main/java"]
        }

        resources {
            srcDirs = ["src/main/resources"]
        }
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.junit.jupiter:junit-jupiter-api:5.4.2'
    implementation 'org.junit.jupiter:junit-jupiter-engine:5.4.2'
}

test {
    useJUnitPlatform {
        includeEngines 'junit-jupiter'
    }
}

jar {
    manifest {
        attributes('Main-Class':mClass)
        baseName projectName
    }
}

run {
    standardInput = System.in
}

mainClassName = mClass
```

です。

追加部分は

Code 31: build.gradle の追加部分

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.junit.jupiter:junit-jupiter-api:5.4.2'
    implementation 'org.junit.jupiter:junit-jupiter-engine:5.4.2'
}

test {
    useJUnitPlatform {
        includeEngines 'junit-jupiter'
    }
}
```

となります。便宜上、下から見ていきます。まず test の項目で、JUnit platform と jupiter を使用することを宣言します。その際依存するパッケージは dependencies の項目に記載します。ここでは implementation 文を使っています。また、記載した依存パッケージの入手先が repositories の項目に記述されます。Gradle が参照する Java パッケージのリポジトリは主に google、mavenCentral、jcenter の 3 つです。今回は mavenCentral を選びます。特に理由はないです。junit-jupiter のバージョンはここでは 5.4.2 ですが、適宜最新版を指定して下さい。

これで準備は完了です。case5 直下で

```
$ gradle test
```

と打てば単体テストを実行してくれます。BUILD SUCCESSFUL と出れば成功です。build/reports/tests/test 以下に html に出力されたレポートが生成されます。ブラウザから閲覧できます。

以上が単体テストの簡単な例です。今回は junit.jupiter.api のうち Test と Assertions 中の assertNotNull しか使っていませんが、テストに用いる機能はまだまだ大量にあります。以降で少しずつ紹介していくつもりです。