◆CygwinでDockerを使う
◆Unity開発でのXcodeのバージョンアップ
◆Re:VIEWのCIをAWS CodeBuildで
◆WebAssemblyで行列演算
◆Pythonインスタンスの属性の実装について

Vol.2

KLab Tech Book

KLab Tech Book

Vol.2

◆CygwinでDockerを使う
◆Unity開発でのXcodeのバージョンアップ
◆Re:VIEWのCIをAWS CodeBuildで
◆WebAssemblyで行列演算
◆Pythonインスタンスの属性の実装について


Lab
Lab

はじめに

このたびは本書をお手に取っていただきありがとうございます。

本書は、KLab 株式会社の有志にて作成された KLab Tech Book の第 2 弾です。今回もさまざまな内容が集まりました。Cygwin 環境で Docker を使う方法・Unity プロジェクトでの Xcode のバージョンアップ・AWS CodeBuild・WebAssembly で行列計算・CPython のインスタンス属性となっています。

内容は各章ごとに独立していますので、どの順番からお読みくださっても問題ありません。

楽しんでいただければさいわいです。

岡本和樹

お問い合わせ先

本書に関するお問い合わせは tech-book@support.klab.com まで。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第 1 章

Cygwin 環境で Docker を快適に使うために

1.1 はじめに

Cygwin は、Unix ライクな環境に慣れきっている人々が Windows を利用するにあたって救いとなるソフトウェアなのは皆さんご存知のところと思います[*1]。また、モダンな開発シーンでは Docker を使うことも増えてきました。しかし、Cygwin 環境から Docker を使おうとすると後述するいくつかの問題にぶつかります。

この章では、どうしても Cygwin から離れられない筆者のような人が、Cygwin 環境のまま Docker を快適に使えるようにする方法についてまとめます。また、試行錯誤の過程を Qiita に投稿してありますので、合わせてご覧ください[*2]。

環境について

動作確認は Windows 7 上の Docker Toolbox で行っています。バージョンに制限は特に無いはずです。Windows 10 Pro と Docker for Windows の組み合わせでは確認をしていませんが、動作原理的には同じように使えると思われます。

1.2 Cygwin で Docker を使うときの問題点

Docker Toolbox にはデフォルトのターミナルとして Docker Quickstart Termnal（GitBash）が付属しています。しかし普段 Cygwin で生活しているのであれば、Cygwin の環境のまま `docker`コマンドを使いたくなってきます。

ドキュメント[*3]を参考に環境変数やパスの設定をすることで Docker Toolbox の `docke`

[*1] 現在では Cygwin 以外にも MSYS2 や Windows Subsystem for Linux など複数の選択肢があります。よい時代になりましたね。

[*2] `https://qiita.com/makiuchi-d/items/7f177cc90c75af40e5ad`

[*3] `https://docs.docker.com/machine/reference/env/`

rコマンドを呼び出すことはできるようになります。ただ、このままでは次の 2 つの問題点があり、実用できるとはとてもいえません。

TTY オプションが使えない

mintty（Cygwin の標準ターミナル）を使っている場合に docker runや docker execで疑似 TTY オプション（--ttyあるいは-t）を指定したとき、TTY モードを有効にできないというエラーが発生します。

```
$ docker run -it ubuntu
cannot enable tty mode on non tty input
```

これは他のターミナルエミュレータを使うことで回避することもできますが、minttyのままで回避する方法を「1.3 TTY モードを有効にできない問題を回避する」で解説します。

そのままのパスをパラメータに指定できない

Cygwin 環境は独自のルートディレクトリを持ち、さらに Windows の各ドライブを/cygdrive 以下に配置するという独特のディレクトリ構成になっています。しかし dockerコマンドは Cygwin のアプリケーションでは無いため、Cygwin 環境のファイルパスをそのまま dockerのパラメータに指定しても、それは存在しないパスとして解釈されてしまいます。

この問題の回避方法について「1.4 Cygwin 環境のパスをそのまま指定できるようにする」で解説します。

1.3 TTY モードを有効にできない問題を回避する

この問題は、mintty のサイト[*4]にも書いてあるように、mintty の疑似 TTY が Windows のネイティブコンソールと互換を持たないためです。Windows のネイティブコンソールアプリケーションを mintty で動かすには、winpty[*5]を使って疑似 TTY とネイティブコンソールの橋渡しをする方法もありますが、ここでは筆者が最終的にたどり着いた sshコマンドを利用する方法について解説します。

[*4] https://mintty.github.io/#compatibility
[*5] https://github.com/rprichard/winpty

SSH コマンドによるネイティブコンソールの回避

さて、Docker は Linux カーネルの機能に依存しているため、Windows 上で直接動作することはできません。Docker Toolbox や Docker for Windows では Linux の仮想マシン（VM）を動かし、その中で Docker が立ち上がっています。そして VM 内には `docker`コマンドも用意されており、VM にログインして `docker`コマンドを叩くことができます。

この VM では `sshd`も立ち上がっているため、一般的な SSH クライアントで接続することができます。そして Cygwin の `ssh`コマンドには`-t`（仮想 TTY）オプションがあるため、これで mintty の TTY と VM 内の `docker`コマンドの TTY を直接つなぐことができます。

つまり、`ssh`コマンドで直接 VM 内の `docker`コマンドを叩くことにより、Windows ネイティブの `docker.exe`を使う必要がなくなり、ネイティブコンソールを完全に回避できるのです[*6]。

ラッパースクリプトによる実装例

SSH を使って VM 内の `docker`コマンドを叩くものを、ラッパースクリプトとして実装してみます。このとき、`docker run`や `docker exec`で疑似 TTY オプションが指定されている場合は、`ssh`にも`-t`オプションをつけて呼び出すようにします。

接続する VM の IP アドレスやユーザ名、使用する鍵といった情報は、`docker-machine inspect`コマンドで取得することができます。ここに含まれるファイルパスは Windows の形式になっているため、Cygwin のパスに忘れずに変換しておきましょう。

▼リスト 1.1　SSH を使って docker コマンドを叩く bash スクリプト

```
#!/bin/bash -e

cmd=(docker "$@")
ttymode=false

# exec/run で -t/--tty があるときのみ ttymode=true
case "$1" in
    "exec" | "run")
        shift
        while [[ $# -gt 0 && $1 =~ ^- ]]; do
            if [[ "$1" =~ ^-[^-=]*t || "$1" =~ ^--tty(=true)? ]]; then
                ttymode=true
            fi
            if [[ "$1" =~ ^-[^-=]*t=false || "$1" == "--tty=false" ]]; then
                ttymode=false
            fi
            shift
        done
```

[*6] Linux の仮想マシンが動いてさえいえれば、VirtualBox だろうと他の仮想化技術だろうと使える方法です。Docker Toolbox とはなんだったのか。

```
        ;;
esac

# SSH 接続に必要な情報を集める
INSPECT=$(docker-machine inspect)
SSHUSER=$(echo "$INSPECT" | grep -Po '"SSHUser":\s*"\K[^"]*(?=",)')
SSHKEY=$(echo "$INSPECT" | grep -Po '"SSHKeyPath":\s*"\K[^"]*(?=",)' |\
    sed -E 's|^(\w+):|/cygdrive/\L\1|;s|\\\\|/|g')
VMADDR=$(echo "$INSPECT" | grep -Po '"IPAddress":\s*"\K[^"]*(?=",)')

# ttymode 有効のときだけ ssh -t にする
sshopt=""
if $ttymode; then
    sshopt="-t"
fi

ssh $sshopt -q -i "$SSHKEY" "$SSHUSER"@"$VMADDR" "$(printf "%q " "${cmd[@]}")"
```

このスクリプトを dockerという名前で PATH の通った場所においておけば、普段とまったく同じ感覚で dockerコマンドを利用できるようになります。winptyを使った場合と違い、標準入出力をパイプにしても問題なく動きます。

もしかしたら docker-machineコマンドの応答待ち時間が気になるかもしれません。複数の VM を切り替えるようなことが無いのであれば、接続先情報を固定値にしてしまうことでさらに快適にすることもできます。ぜひお試しください。

1.4 Cygwin 環境のパスをそのまま指定できるようにする

docker run時のボリュームのマウント（--volumeまたは-v）などで Windows 上のファイルやディレクトリを dockerコマンドに渡すことはよくあります。VM 内で動作している Docker は、ホストである Windows のファイルやディレクトリにどうやってアクセスしているのでしょうか。

答えは単純で、共有フォルダとして VM 内にマウントしているのです。Docker Toolbox の場合、初期設定で Windows の C:\Users を VM 内の/c/Users にマウントしています。ということは、C:\Users 以外の場所を渡すことは初期設定のままではできません。

一方、特にボリュームのマウントではフルパスでの指定を求められるのですが、これが VM 内でのフルパスになっていないと Docker は解釈できません。

Cygwin 環境でのフルパスは、C:\Users は/cygdrive/c/Users となってしまい、VM 内のフルパスとは異なります。加えて Cygwin の/home 以下などのパスについても、/c/Users 以下の対応するパスに書き換える必要があります。

ここでもまたラッパースクリプトで、dockerコマンドに渡す前にパスを変換することで回避していきます。

Cygwin 環境のパスを VM 内のパスに変換する

Cygwin に付属している cygpathコマンドで、Cygwin 環境のパスをさまざまな形式のパスに変換することができます。特に cygpath -amとすると、ドライブレターから始まる“/”区切りのフルパスが得られ、都合がよいです。

```
$ file WinHome
WinHome: symbolic link to /cygdrive/c/Users/makiuchi-d

$ cygpath -am WinHome/Projects
C:/Users/makiuchi-d/Projects
```

このようにシンボリックリンクも解決してくれるので、先頭のドライブレター部分を小文字にして“/”で始まるように置換するだけで済みます。これを dmpathという関数として定義したものをリスト 1.2 に示します。

▼リスト 1.2　cygwin 環境のパスを VM 内のパスに変換するコマンド

```
function dmpath ()
{
    cygpath -am "$1" | sed -E 's|^(\w*):|/\L\1|'
}
```

パスが与えられるパラメータ部分だけパスを変換する

コマンドラインのパラメータを列挙しただけでは、どれが変換すべきパスなのか判別がつきません。それどころか、docker cpでは 2 つのパラメータのうち“:”の無い方だけをパス変換しなければならない一方で、docker run -vの場合は“:”の前半のみパス変換しなければならなず、さらに、コンテナ内で実行するコマンドのオプションについては変換してはならないなど、対応方法が千差万別です。

正しく動作させるためには、知りうる限りのオプションを片っ端からチェックしていく以外になさそうです。

ここでは docker cpと docker runの一部のオプションについてパス変換したものをリスト 1.3 に示します。

▼リスト 1.3　適切なパラメータのパスを変換して docker を呼び出す bash スクリプト

```
#!/bin/bash -e
# dmpath() の定義は省略

cmd=(docker "$@")

case "$1" in
```

```
    "cp")
        cmd=(docker cp)
        shift
        while [[ $# -gt 0 ]]; do
            case "$1" in
                -* | *:*)  # オプションやコンテナ内のパスは変換しない
                    cmd=("${cmd[@]}" "$1")
                    ;;
                *) # それ以外は変換する
                    cmd=("${cmd[@]}" "$(dmpath "$1")")
                    ;;
            esac
            shift
        done
        ;;
    "run")
        cmd=(docker run)
        shift
        while [[ $# -gt 0 && $1 =~ ^- ]]; do
            # volume オプションはホスト側のパス部分を取り出して置換
            if [[ "${1%=*}" =~ ^-[^-]*v$ || "${1%=*}" = "--volume" ]]; then
                case "$1" in
                    *=*)
                        opt="${1%=*}"
                        paths="${1#*=}"
                        ;;
                    *)
                        opt="$1"
                        paths="$2"
                        shift
                        ;;
                esac
                cmd=("${cmd[@]}" "$opt" "$(dmpath "${paths%:*}"):${paths#*:}")
            # 引数を取るオプションは次のパラメータも一緒に処理（一部）
            elif [[ "$1" =~ ^-[^-]*[aelpuw]$ ||
                    "$1" = "--attach" ||
                    "$1" = "--env" ||
                    "$1" = "--label" ||
                    "$1" = "--link" ||
                    "$1" = "--mount" ||
                    "$1" = "--name" ||
                    "$1" = "--publish" ||
                    "$1" = "--user" ||
                    "$1" = "--workdir" ]]; then
                cmd=("${cmd[@]}" "$1" "$2")
                shift
            # その他のパラメータはそのまま追加
            else
                cmd=("${cmd[@]}" "$1")
            fi
            shift
        done
        # パラメータ以外（image 名以降）は置換せず追加
        cmd=("${cmd[@]}" "$@")
        ;;
esac

$(printf "%q " "${cmd[@]}")
```

このスクリプトでは、docker runのパラメータを網羅できていません。たとえば、デタッチキーを変更する--detach-keysは追加のパラメータを要求しますが、このままで

は単独で使われるパラメータとして判定されてしまい、キー文字列をイメージ名と判定してしまいます。

少なくとも自身で使うパラメータはすべて網羅しなければなりません。もしエレガントな解決策などありましたらご教示ください。

また、ここに示した`docker cp`、`docker run`以外にもパスを受け取るコマンドがあります。それらについても同様の方法でパス変換することで、この問題は解決できます。

このようなラッパースクリプトを用いることで、パス名に関して Cygwin 環境であることを意識することなく`docker`コマンドが使えるようになります。

1.5 まとめ

Cygwin 環境で`docker`コマンドを使うときにぶつかる、疑似 TTY の問題とパス名の問題について回避策を示しました。

紙面と解説の都合上、それぞれの回避策を別々のスクリプトとして紹介しましたが、この 2 つの回避策を統合し、さらにコマンドパラメータをより多く網羅したものを GitHub にて公開しています[*7]。

これにより Cygwin 環境でも快適に Docker を利用できるようになりました。お困りの方はぜひご利用ください。

最後に強く言いたいことは、Docker を使うならホストも Linux にしましょう。

Makiuchi Daisuke / @makki_d

[*7] `https://github.com/makiuchi-d/docker-in-cygwin`

第 2 章

Unity で開発しているスマホゲームで Xcode のバージョンアップをする

KLab 入社から約三年が過ぎようとしているクライアントエンジニアです。KLab ではこれまで、Unity を使ったいくつかのプロジェクトでビルド環境、フローの改善に取り組んできています。

本章では、これまで Xcode のバージョンアップを行ってきた経験を活かし、同じようにビルド環境、フローの改善に取り組んでいる方々に向けた情報をお伝えしたいと思います。KLab で筆者が参加してきたプロジェクトの事例を通し、どのようなアプローチを取るべきか？ なるべく詳細に記載することで、読者の皆様の取り組みに役立てていただけたらと思います。

本章を読むことで、次のような知見の獲得が期待できると思います。

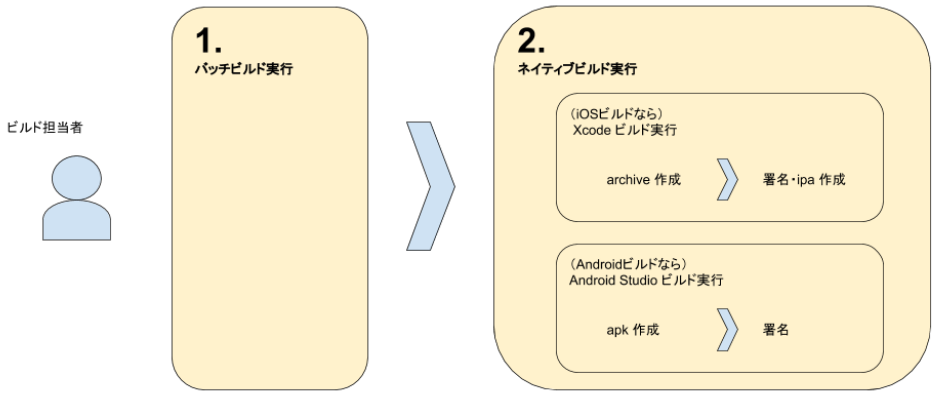
- Unity で開発している iOS/Android アプリのビルドの知見
- iOS アプリのビルド手法、注意すべき点、対応ノウハウ
- Xcode バージョンアップ対応の TODO やスケジューリングの留意点

それでは、早速見ていきましょう。

2.1 Unity で開発しているスマホゲームのビルド手法

Unity で開発しているアプリのビルド手法

Unity でアプリのバイナリを生成する流れは大まかに図 2.1 のようになっています。



▲図 2.1　Unity で開発しているアプリのバイナリ生成フロー

Unity でビルドする際にはバッチビルドの手法が用いられます。バッチビルドとは、ビルドスクリプトを C#で記述して、コマンドラインで実行するものです。

最小限のビルドスクリプトの例をリスト 2.1 で紹介します。

▼リスト 2.1　最小限の Unity バッチビルドスクリプト

```
using UnityEditor;
class MyEditorScript
{
     static void PerformBuild ()
     {
         string[] scenes = { "Assets/MyScene.unity" };
         BuildPipeline.BuildPlayer(scenes, ...);
     }
}
```

このスクリプトをコマンドラインで実行する際はリスト 2.2 のようにして実行します。

▼リスト 2.2　最小限の Unity バッチビルドスクリプト

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity \
    -quit \
    -batchmode \
    -executeMethod MyEditorScript.PerformBuild
```

以上のスクリプトは Unity の公式ドキュメント[*1] で紹介されている内容です。

実際のプロジェクトでは、前述の PerformBuild のパラメータに「どのサーバー環境に接続するか」「In-App Purchase や Game Center のサンドボックス課金用バイナリか」

[*1] Unity 公式ドキュメント
https://docs.unity3d.com/ja/current/Manual/CommandLineArguments.html

「本番バイナリか」といった内容の情報を含めているところが多いと思います。

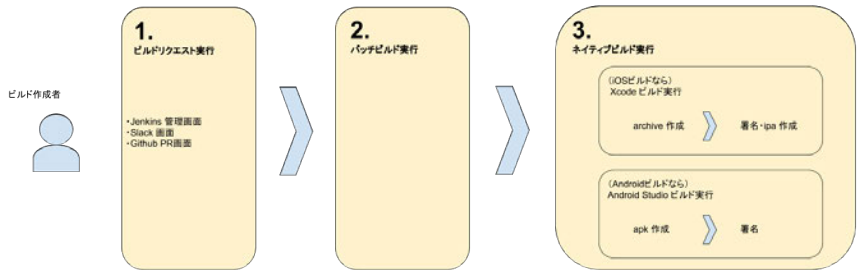
参考までに、筆者が関わってきたタイトルで使っていたスクリプトを紹介します。

▼リスト 2.3　リズムゲームプロジェクトの Unity バッチビルドスクリプト

```
${UNITY_PATH} \
-batchmode \
-quit \
-executeMethod BatchBuild.BuildProcess \
-projectPath ${WORKDIR}/rhythm-prj-client-server/client \
-buildTarget=ios \
-productName=${PRODUCT_NAME} \
-bundleVersion=${BUNDLE_VERSION} \
-bundleIdentifier=${BUNDLE_IDENTIFIER} \
-developmentBuild=${DEVELOPMENT_BUILD} \
-il2cppBuild=${IL2CPP} \
-enableAssetbundle=${ENABLE_ASSET_BUNDLE} \
-firebaseEnv=${FIREBASE_ENV} \
-buildNumber=${BUNDLE_VERSION_INTERNAL} \
-logFile ${WORKSPACE}/${BUILD_NUMBER}.log
```

Slack や GitHub と連携したビルドリクエスト、バイナリ配布による業務改善

バイナリビルドは、Slack や GitHub と連携して bot へのメッセージ・PR コメントによってリクエストを行い、サーバーなどにアップロードして配布することで業務改善に繋げることもできます（図 2.2）。



▲図 2.2　Slack・GitHub と連携したアプリのバイナリ生成フロー

KLab では上記のような仕組みを導入しているプロジェクトも複数ありますが、詳細は前著『KLab Tech Book Vol. 1』でも紹介しています。pdf 版は次の URL でも配布していますので、ご興味あれば参照してみてください。

KLab Tech Book Vol. 1
`http://klabgames.tech.blog.jp.klab.com/techbook/KLab-Tech-Book-Vol-1-1.2-ebook.pdf`

2.2　iOS アプリのビルド手法

前節では Unity でアプリのバイナリをバッチビルドで生成する流れを紹介しました。本節では iOS アプリのビルドを具体的に見ていきましょう。

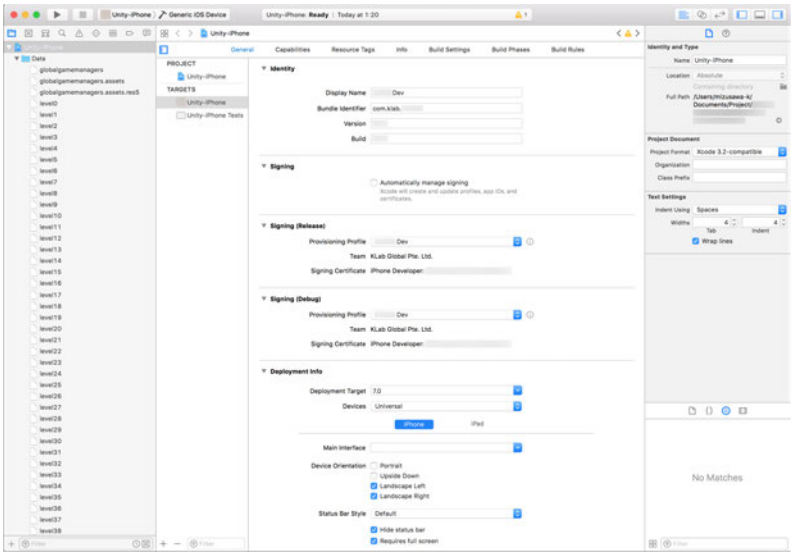
iOS アプリのビルドのしかたには大きくふたつあります。ひとつは「Xcode の GUI 画面を用いたビルド（以降 **GUI ビルド**と呼びます）」です。そしてもうひとつは「Xcode Command Line Tool を用いたビルド（以降 **CLI ビルド**と呼びます）」です。

ipa の GUI ビルドの流れ

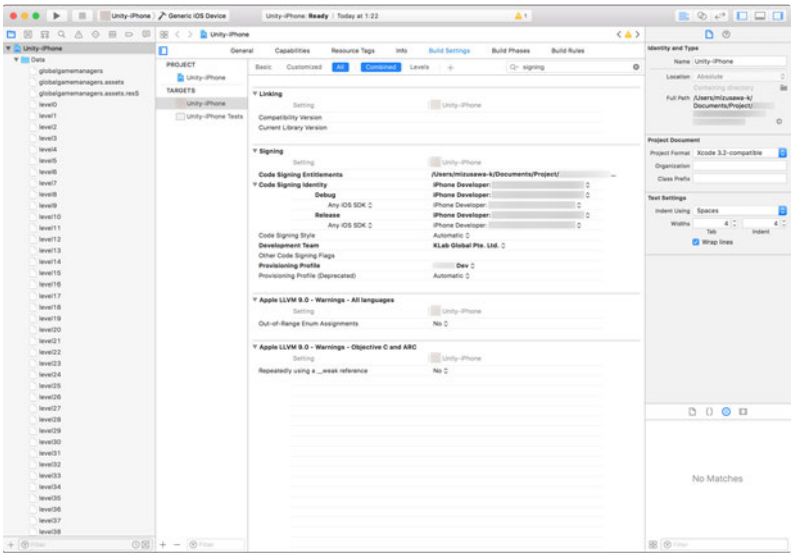
ipa の GUI ビルドの流れは次のようになります（Xcode 9.2 の場合)。

1. Xcode 上でプロジェクトを開く（図 2.3）
2. Build Settings にて、ipa のビルドに使用するプロビジョニングプロファイル[*2]、証明書を指定する（図 2.4）
3. Xcode > Run > Archive メニューを選択する（図 2.5）
4. Xcode > Organizer メニューを選択する
5. 画面に表示されたアプリの一覧から、archive を選択し、Export ボタンを選択する（図 2.6）
6. アプリの配布方式の中から、2. で指定したプロビジョニングと合致するものを選択し、Next ボタンを選択する（図 2.7）
7. App Thining の設定をして Next ボタンを選択する（図 2.8）
8. アプリのプロビジョニングプロファイル、証明書の中から、2. で指定した内容と合致するものを選択し、Next ボタンを選択する（図 2.9）
9. 内容を確認して、Next ボタンを選択する（図 2.10）
10. ipa や plist を export するディレクトリ階層や、ディレクトリ名を指定して Export ボタンを選択する（図 2.11）
11. ipa が生成されたことを確認する

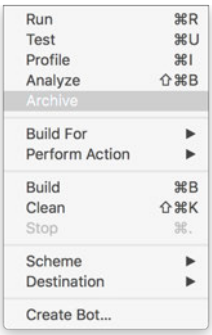
[*2] バイナリと開発端末、証明書を紐づけて管理するためのファイルで、拡張子は.provisioningprofile です。エディタで開くことで端末リストや証明書の内容を確認することができます



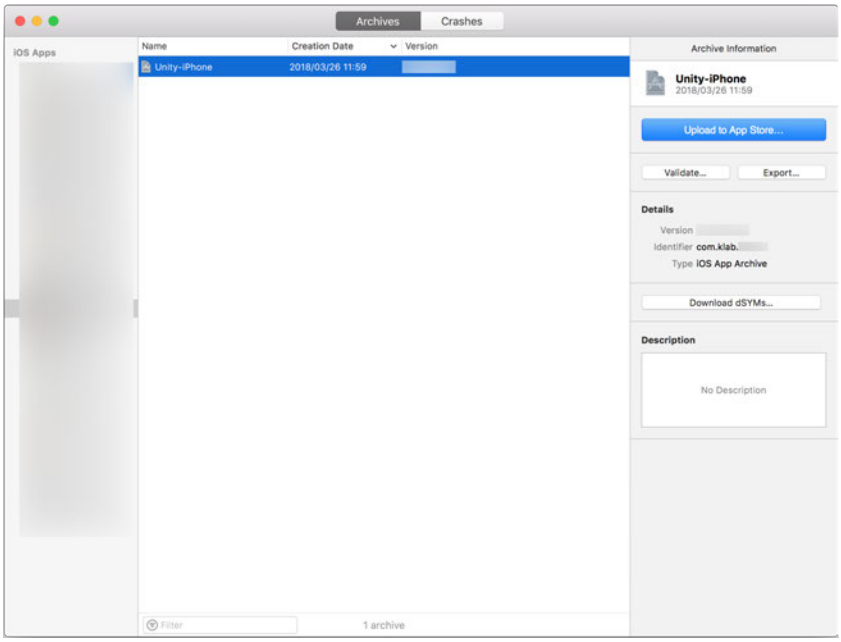
▲図 2.3　Xcode 上でプロジェクトを開く



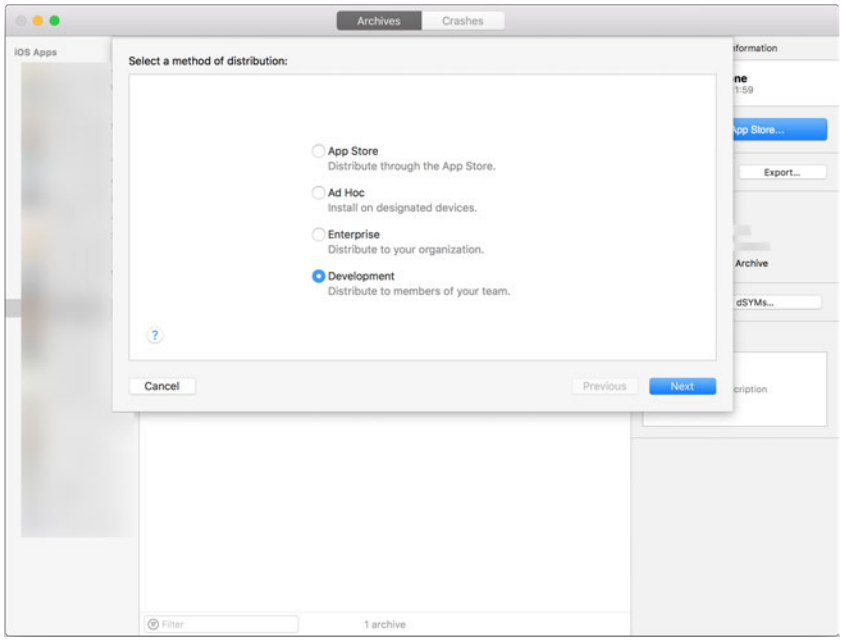
▲図 2.4　Xcode 上でプロビジョニングプロファイル、証明書を指定する



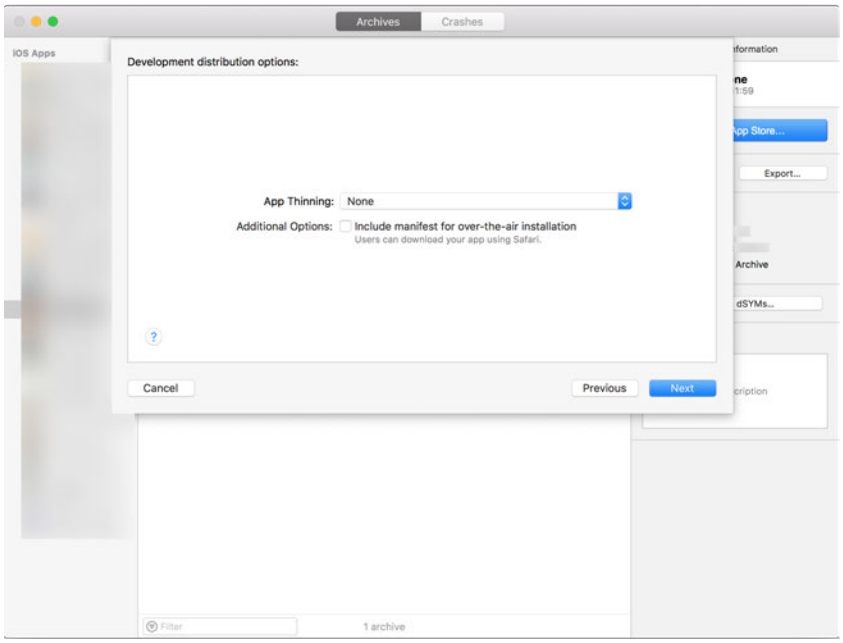
▲図 2.5　Xcode 上で Archive メニューを実行する



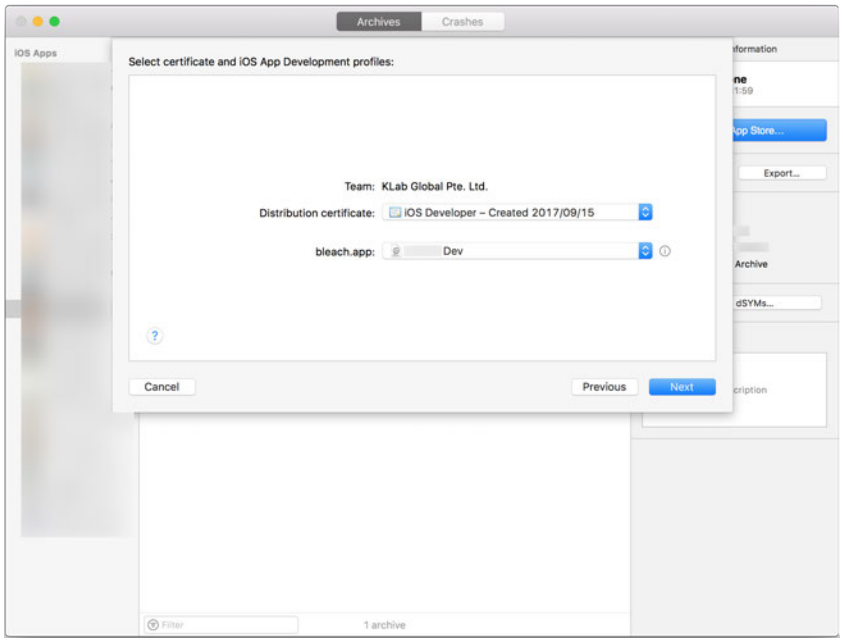
▲図 2.6　Xcode 上で ipa を作成する



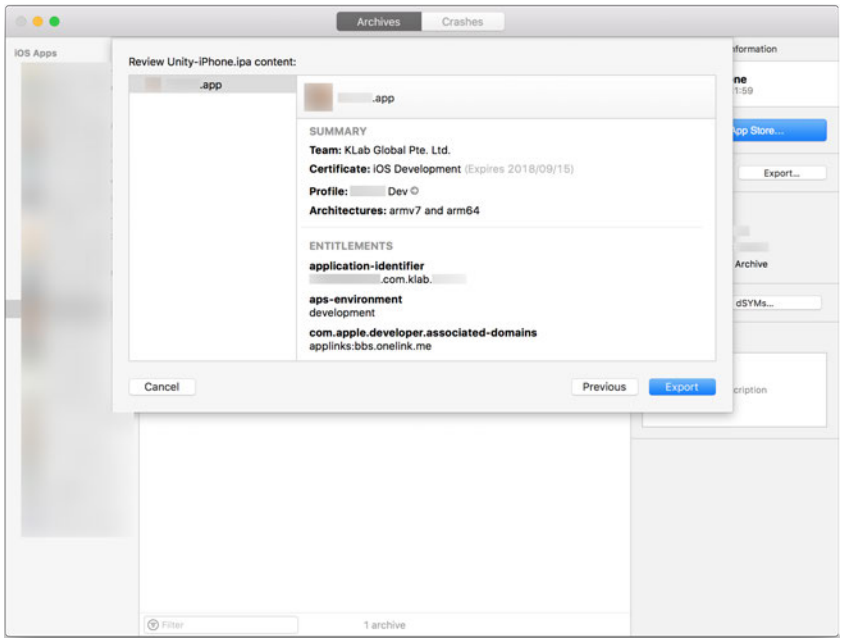
▲図 2.7　Xcode 上で ipa を作成する



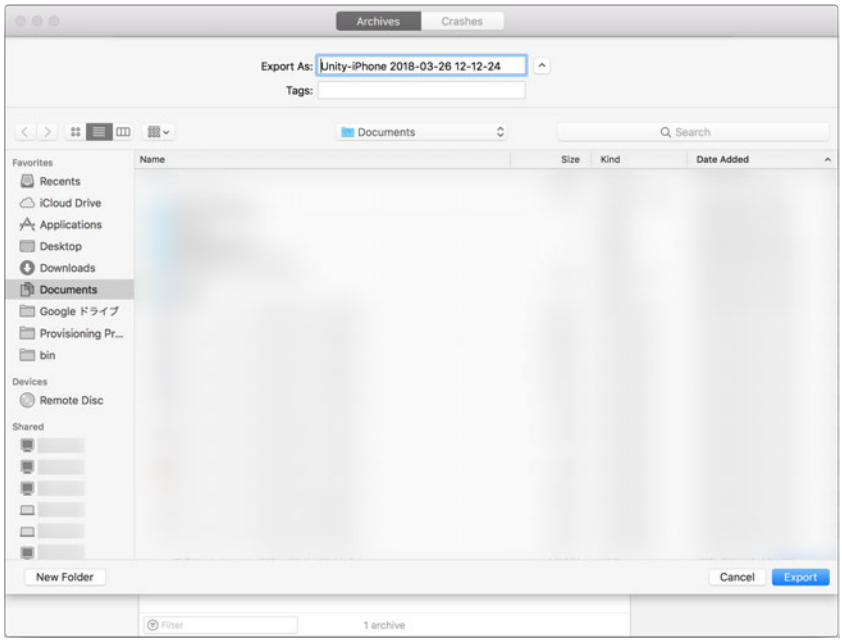
▲図 2.8　Xcode 上で ipa を作成する



▲図 2.9　Xcode 上で ipa を作成する



▲図 2.10　Xcode 上で ipa を作成する



▲図 2.11　Xcode 上で ipa を作成する

ipa の CLI ビルドの流れ

ipa の CLI ビルドの流れは次のようになります。

1. Xcode のプロジェクトをマシンに用意する
2. プロジェクトに対して、archive 生成コマンドを実行する
3. archive が生成されたら、archive に対して ipa エクスポートコマンドを実行する
4. ipa が生成されたことを確認する

Xcode のプロジェクトをマシンに用意する

Git からクローンする、Unity からプロジェクトを出力するなどして、Xcode プロジェクトを用意します。

プロジェクトに対して、archive 生成コマンドを実行する

用意した Xcode プロジェクトに対して、リスト 2.4 のような archive 生成コマンドを実行します。

▼リスト 2.4　Xcode CLI での archive ビルドスクリプト

```
xcodebuild \
-project "${XCODE_PROJECT_CONFIG_PATH}" \
-configuration "${CONFIGURATION}" \
-scheme Unity-iPhone \
-archivePath $ARCHIVE_PATH \
-sdk iphoneos11.2 \
CODE_SIGN_IDENTITY="${CODE_SIGN_IDENTITY}" \
PROVISIONING_PROFILE="${PROFILE_UUID}" \
archive 2>&1)
```

archive に対して、ipa エクスポートコマンドを実行する

生成した archive に対して、ipa エクスポートコマンドを実行します。

▼リスト 2.5　Xcode CLI での ipa ビルドスクリプト

```
xcodebuild -exportArchive \
-archivePath $ARCHIVE_PATH -exportPath $UNITY_BUILD_PATH \
-exportOptionsPlist $EXPORT_OPTIONS_PLIST_PATH
```

ipa が生成されたことを確認する

生成された ipa を確認します。

2.3　Xcode バージョンアップのケーススタディ

本節では、バージョンアップする時の流れに沿って、どういう点に留意すべきか？ 具体的に何を確認しながら進めるべきか？ 示していきたいと思います。

Xcode バージョンアップの進め方（一例）

1. スケジュール作成
2. 検証用ビルドマシン、ジョブの確保
3. 検証用ビルドマシンの環境構築（Git・Unity・Xcode など）
4. ビルド環境検証（ローカルでの GUI/CLI ビルド・Jenkins ビルド）
5. 検証環境で作成したビルドのテスト
6. 運用ビルドマシン、ジョブへの反映

スケジュール作成

経験的に、おおよそ 1 週間から 2 週間程度見ておくと大体収まる印象です。前述した進め方にのっとると、各項目の工数は次のようになります。

1. スケジュール作成：—
2. 検証用ビルドマシン、ジョブの確保：0.5 日
3. 検証用ビルドマシンの環境構築（Git・Unity・Xcode など）：0.5 日
4. ビルド環境検証（ローカルでの GUI/CLI ビルド・Jenkins ビルド）：2.0 日
5. 検証環境で作成したビルドのテスト：1.5 日
6. 運用ビルドマシン、ジョブへの反映：0.5 日

工数を見る時に参考にしたいのは、一回のビルドにかかる時間です。Xcode をアップデートした場合、ビルド手順や設定の確認と修正のために何度も試行錯誤する必要があります。しかし、プロジェクトごとにビルド時間は異なるとは思いますが、一度頭からビルドするとなると全体で 30〜60 分くらいかかります。場合によっては 1 日に 8 本程度しかビルドを試せません。このため、比較的余裕を持ったスケジュールとしては、作業者が 1 人の場合は 2 週間くらい見ておいた方が安心、という印象です。

検証用ビルドマシン、ジョブの確保

Xcode でのバイナリビルドについては、「異なるバージョンの Xcode で同時にビルドさせようとすると片方のビルドが失敗してしまう」現象が知られています。このため、デイリービルドなどを行なっているプロジェクトであれば、なるべく検証用のマシンを別途確保したいところです。

また、Jenkins を利用しているプロジェクトの場合は、既存のジョブをコピーしてテスト用のジョブを作成すると簡単に検証に取り掛かることができます。既存のジョブをそのまま実行させながらジョブの設定を試行錯誤することができますので、テスト用ジョブを確保しておくことはオススメです。

検証用ビルドマシンの環境構築

Xcode・Unity・Git など、必要なソフトをインストールします。このとき、macOS のバージョンアップも必要になる点は意識しておきたいポイントです。Xcode のメジャーバージョンアップではほとんどと言っていいほど macOS のバージョンアップが必要になりますので、macOS の変更点の確認もしておいた方がよいでしょう[*3]。

ビルド環境検証

GUI でのビルドは比較的すんなり行きやすいことと、CLI でのビルドに活かせる部分があるので先に GUI でのビルドを試します。

[*3] Xcode を 8 系から 9 系にアップデートした際、「当時の最新 macOS マシンで、Unity がプロジェクトのファイルを認識できなくなる」現象が起きたのですが、その時は社内の空いているマシンの中から正常動作する macOS バージョンのマシンを探して利用するという事例がありました

iOS ビルドでは、あるある話なのですが、大体コード署名エラーで失敗するのでその辺りを見ていくことになります。このとき、エラーログの内容だけではわからないことが多く、何だかんだで Xcode のリリースノート、デベロッパーフォーラムなどの力を借りることが多いです。また、プロビジョニングプロファイルや証明書、秘密鍵のインストール方法によってビルドの成否が変わることもちょくちょくあります。

このときチェックしたいポイントを見ていきます。

プロビジョニングプロファイル

まず、プロビジョニングプロファイルの確認です。

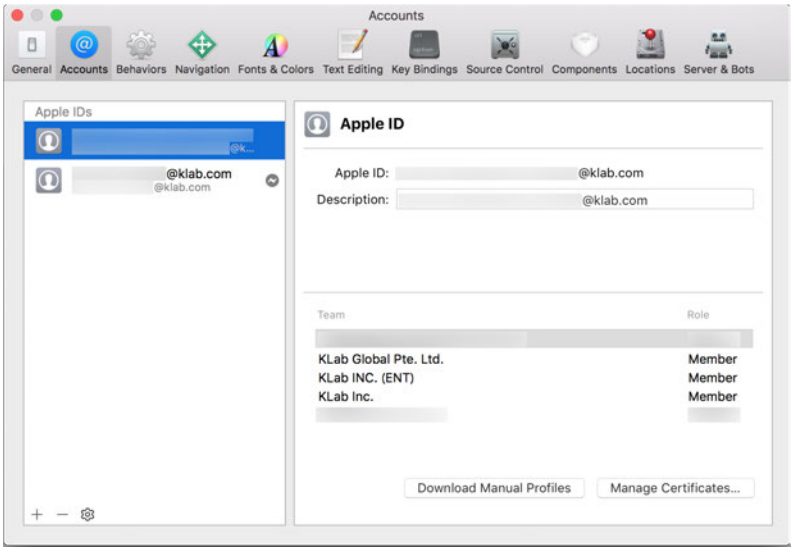
Xcode が認識しているプロビジョニングプロファイル群は、/Users/{user-name}/Library/Mobile Device/ProvisioningProfile/以下にインストールされています。ls /Users/{user-name}/Library/Mobile Device/ProvisioningProfile/というコマンドをターミナルで実行すればプロビジョニングプロファイルの一覧が確認可能です。

no ProvisioningProfile といった「コード署名でプロビジョニングプロファイルが存在しない」という旨のエラーが出てビルドに失敗する時は、ここにプロビジョニングプロファイルが入っているか確認しましょう。

また、まれに古いプロビジョニングプロファイルが残っていて悪さをしている場合もあるので、一度一式削除して入れ直すのもよい手です。プロビジョニングプロファイルは、次の方法でインストールすることができます。

- Xcode > Preference > Account に Apple Developer アカウントを設定し、Xcode で直接ダウンロード、インストールする
- Apple Developer Center[*4]にログインし、必要なプロビジョニングプロファイルをダウンロードし、ファイルをダブルクリックしてインストールする
- Apple Developer Center にログインし、必要なプロビジョニングプロファイルを CLI 上でインポートしてインストールする

[*4] https://developer.apple.com/account/



▲図 2.12　Xcode でのプロビジョニングプロファイル一括ダウンロード画面

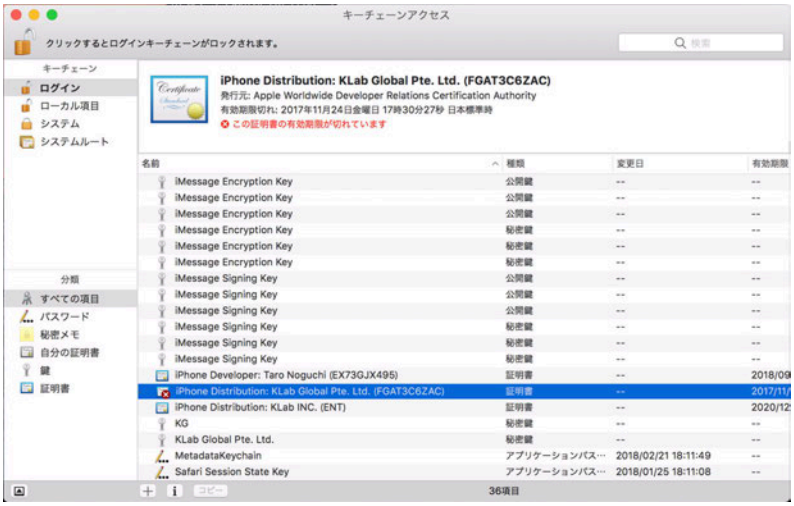
証明書、秘密鍵

次に、証明書、秘密鍵の確認です。macOS が認識している証明書、秘密鍵はキーチェーンアクセスで確認できます。

証明書で確認すべき点は次です。いずれかの状態に反している場合、署名に失敗するケースがあります。

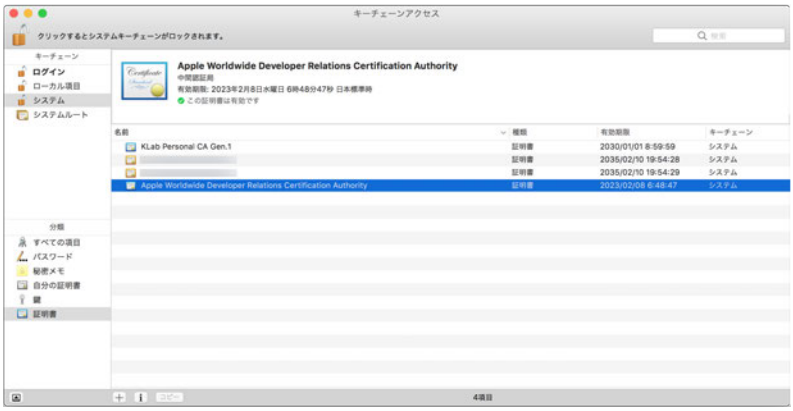
- コード署名証明書、中間認証局の証明書それぞれの有効期限が切れていないこと（図 2.13・図 2.14）
- Apple 中間認証局の証明書（Apple World Developer Relations Certificate Authority, Developer ID Certification Authority）がログインキーチェーンに登録されておらず、システムに表示されていること（Jenkins で CLI ビルドするマシンの場合のみ[*5]。図 2.15）

[*5] GUI ビルドする場合は Apple 中間認証局の証明書がログインキーチェーンに登録されていても問題なくビルドできます



▲図 2.13　有効期限が切れた証明書をキーチェーンアクセスで確認している画面

▲図 2.14　ログインキーチェーンに登録されている証明書を確認している画面



▲図 2.15　システムキーチェーンに登録されている証明書を確認している画面

プロビジョニングプロファイル同様、古い証明書がまれに悪さをしている場合もあるので、一度一式削除して入れ直すのもよい手です。証明書は、次の方法でインストールすることができます[*6]。

- Apple Developer Center にログインし、必要な証明書をダウンロードし、ファイルをダブルクリックしてインストールする
- Apple Developer Center にログインし、必要な証明書を CLI 上でインポートしてインストールする（リスト 2.6）

秘密鍵で確認すべき点は次です。

- 秘密鍵の　情報を見る > アクセス制御　項目で、アクセスすることを許可したアプリケーションとして**/usr/bin/codesign** が追加されていること（図 2.16）

[*6] 一般的に、iOS アプリの証明書は秘密鍵を含む p12 ファイルで扱います

▲図 2.16　証明書に紐づけられた秘密鍵の情報確認画面で codesign にアクセスを許可しているか確認している画面

▼リスト 2.6　証明書を CLI 上でインポートする

```
# キーチェーンのアンロック
security unlock-keychain -p $OSX_ADMIN_PASSWORD "${KEYCHAIN_LOCATION}"
security set-keychain-settings -t 3600 -l "${KEYCHAIN_LOCATION}"

# インポート
security import "${IOS_P12_FILE_PATH}" \
-f pkcs12 \
-P "${IOS_P12_PASSWORD}" \
-k "${KEYCHAIN_LOCATION}" \
-T /usr/bin/codesign
```

iOS アプリ開発に必要になる証明書

一般的に、iOS のコード署名証明書は、「証明書」の情報に「CSR[a]を生成した PC の秘密鍵」の情報が付加された p12 ファイルで扱うことが多いと思いますが、この p12 ファイルは以下のような手順で作ることができます。

1. Mac のキーチェーンアクセスで certSigningRequest ファイルを生成
2. 1. で生成した certSigningRequest ファイルを Apple Developer Center にアップロードして cer ファイルを生成
3. 2. で生成した cer ファイルを 1. の作業を行った PC（certSigningRequest を生成した PC）でダウンロード
4. 3. でダウンロードした cer ファイルを実行して、キーチェーンアクセスに登録
5. 4. で登録した証明書をキーチェーンアクセスで選択して、p12 ファイルとして書き出し生成

手順こそ少々複雑ですが、このように p12 ファイルの形で扱うことによって certSigningRequest ファイルを生成した PC 以外でも証明書を利用できるようになり、その結果ビルドマシンごとに証明書を生成する必要がなくなるメリットがあります。

[a] Certificate Signing Request。証明書作成を認証機関に依頼するために作成が必要になるファイルです

2.4 Xcode バージョンアップによるビルド手法の変更点、キャッチアップ方法

ここまで、駆け足ですが Unity で開発している iOS アプリのビルド手法、バージョンアップ時の注意点などを示してきました。

Xcode のバージョンアップでは、それぞれ微妙にビルド方法に変更が入り、ビルドスクリプトなどに手を入れる必要が生じます。

大きくは中間成果物（app・xcarchive など）を作成して、署名して ipa を生成する流れですが、コマンドの必須オプションが増えたり、オプションで指定するファイルを別途用意する必要が出て来たりとバージョンにより違いがあります。

筆者の今までの経験的には、たとえば「app を生成して ipa を作る方法から xcarchive を生成して ipa を作る方法に変更した（Xcode 6.3 → 7.1）」「ipa 生成時のオプションで署名の種別などを定義する plist（ExportOptions.plist）を指定するように変更した（Xcode

7.1 → 8.2)」「ExportOptions.plist の必須定義を追記した、ストア用のアプリアイコンをバイナリに追加した（Xcode 8.2 → 9.2)」といったものがあります。

これらは Xcode のリリースノートなどのドキュメントにも記述されているものですが、普通にビルドを試して出てきたエラーログを眺めるだけでは分かりにくいものです。公式ドキュメントは読んでおきましょう。

Xcode のバージョンアップが必要になる状況とは

そもそも、Xcode のバージョンアップは何のために行うのでしょうか。
Xcode のバージョンアップが必要になる状況というのはプロジェクトによってまちまちです。基本的には新しい API を使えるようにする、App Store で継続的にアプリをリリースし続けられるようにする、ということだと思いますが、昨今のスマフォゲームでは多くの外部 SDK を利用していますので、この SDK の動作状況に引っ張られてやむなくアップデート、ということもあります。
以下は筆者が携わったプロジェクトのバージョンアップ経緯です。

- Xcode 6.3 → 7.1：パズルプロジェクトの場合
  - パズルプロジェクトでは、新しい iOS のベータ版で、広告 SDK の初期化でアプリが落ちる現象が起きていたため Xcode バージョンアップを行う必要がありました。
  - SDK のバージョンアップでの対応が期間的にできず、また Xcode のバージョンアップにより現象が発生しないことがわかっていたため、Xcode バージョンアップを行うことは必須でした。
- Xcode 7.1 → 8.2：リズムアクションゲームプロジェクトの場合
  - リズムアクションゲームプロジェクトの場合、新 iOS の API 利用の需要が強くあったため、新 iOS の SDK が同梱されているより新しい Xcode にバージョンアップを行うことが望ましい状況でした。
- Xcode 8.2 → 9.2：アクションゲームプロジェクトの場合
  - iPhone X でのフルスクリーン表示のため。
  - レターボックス表示の場合、将来的にヒューマンインターフェイスガイドラインに沿っていないことによりアプリの審査に落ちてしまうことが想定されたため。

2.5 おわりに

Xcode のバージョンアップは、6 系から 7 系、7 系から 8 系、そして 8 系から 9 系と三度に渡り経験してきました。何だかんだでいまだにつまづく時はつまづくし、ハマって時間を費やす時は費やします。

iOS ではアプリの審査があり、動作確認用だけでなく iTunes Connect にアップロードできるバイナリも作る必要もありますので、さらに時間がかかります。また、アプリが利用しているサードパーティの SDK についても、動作に問題がないかなどのテストをしておくことも必要です。Xcode のアップデートは必要になってから対応を進めるよりは、一年に一度のメジャーアップデートのタイミングで早めに済ませておくのもひとつの手かもしれません。

最後になりますが、実際のところ、こうした記録としてまとめておくことで自身の Xcode バージョンアップ作業の備忘録、助けになることを期待したりもします。しかしながら、本章の内容が、自分自身だけではなく読者の方々の役に立つことができれば嬉しいなと思います。

Kinuko MIZUSAWA

第 3 章

Re:VIEW で書いた原稿の CI を AWS CodeBuild で回す

KLab でビルド、Jenkins 周りも見ているクライアントエンジニアです。この章では、この本（本書「KLab Tech Book Vol. 2」）の pdf 生成 CI 環境を作ったよ。という話をしたいと思います。

3.1　ことのはじまり

発起人岡本から、「技術書典 3 のときはできなかったけど今度こそ CI 環境ちゃんと作りたい！ 誰かやりたい人いない？？」という声かけがあったので、ちょうど「クラウド CI サービス、そういえばちょっと興味あるな〜」と思いやってみることになりました。

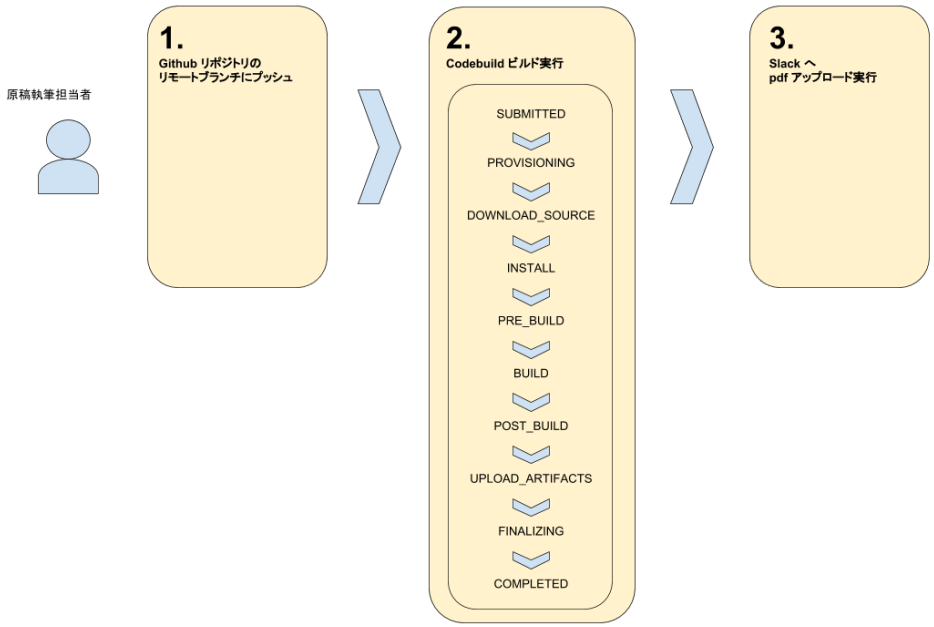
今回は、CI 環境を支えるサービスとして AWS CodeBuild[*1] を使っています。当初は Travis CI を利用する話も挙がったのですが、社内では AWS を積極的に使っているので、社内実績のない CodeBuild も使ってみれると良さそうということや、Travis CI よりも費用が安くできるのではということがあり、検証して問題なければ採用してみるという流れでそのまま採用となりました。

3.2　CI 環境、フロー

CI フロー図

CI のフローは図 3.1 のようになります。

[*1] AWS CodeBuild `https://aws.amazon.com/jp/`

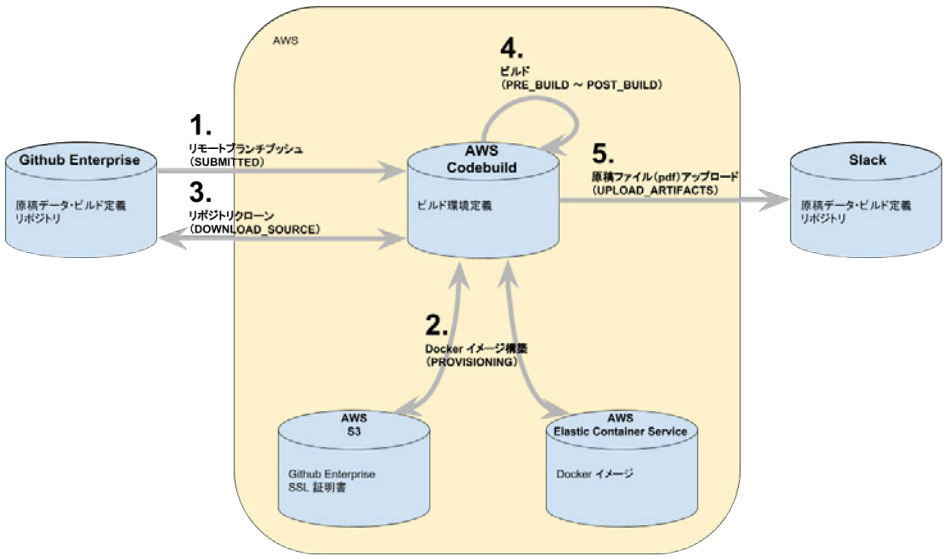


▲図 3.1　AWS, GitHub Enterprise を利用した Re:VIEW 原稿の CI フロー

- GitHub Enterprise の Webhook でコミットがプッシュされるたびに pdf 生成
- 生成された pdf は Slack で通知

システム構成図

システムの構成は図 3.2 のようになります。



▲図 3.2　AWS, GitHub Enterprise を利用した Re:VIEW 原稿の CI システム構成

- GitHub Enterprise[*2]
  - Re:VIEW の原稿の置き場所
- AWS CodeBuild
  - 原稿の pdf 化を行う場所
- AWS S3[*3]
  - 証明書、成果物の置き場所
- AWS Elastric Container Service（ECS）[*4]
  - pdf 化を行うマシンの Docker[*5]イメージの置き場所
- Slack[*6]
  - 技術書典 pdf のアップロード先

AWS では、CodeBuild で作られた成果物は同じ AWS の S3 に保存することができ、ECS に Docker イメージを置いておくことで、独自の Docker イメージで CodeBuild のビルド処理を走らせることもできます。2018 年 1 月末より GitHub Enterprise もサポートされ、Webhook によるビルドができるようになっていたため、今回はそれらを組み合わせた形で構成しています。

[*2] GitHub Enterprise `https://enterprise.github.com/home`
[*3] AWS S3 `https://aws.amazon.com/jp/`
[*4] AWS Elastric Container Service `https://aws.amazon.com/jp/`
[*5] Docker `https://www.docker.com/`
[*6] Slack `https://slack.com/`

3.3 AWS CodeBuild とは

▲図 3.3　AWS CodeBuild トップ画面

Amazon が提供している AWS の中に含まれている CI サービスです。

特徴

ビルドにかかった時間によって請求金額が決まり、ビルドしなければ費用が発生しない点が大きな特徴です。また、AWS の他サービスとの連携がサポートされていて扱いやすい点も特徴といえます。また、日本語のドキュメント、チュートリアルも比較的充実している印象です。

料金体系、用意されているマシンスペック

利用するコンピュータのスペックによって、分単位で決まります。ビルド時間は分ごとに切り上げで扱われます。

インスタンスタイプ	メモリ	CPU 数	1 分あたりの料金 (USD)
build.general1.small	3 GB	2	$ 0.005
build.general1.medium	7 GB	4	$ 0.010
build.general1.large	15 GB	8	$ 0.020

また、プログラミング言語環境としては、予め必要なツールがインストールされた Docker イメージとして、言語ごとに Ubuntu 14.04 系で用意されています。CodeBuild では AWS の他サービスを利用することで自前のカスタム Docker イメージを利用することが可能ですが、特に必要がなければまず、CodeBuild 標準の Docker イメージを利用できないか検討するとよさそうです。

CodeBuild では、以下の言語をインストールした Docker イメージをバージョンごとに個別に用意しており、言語環境を選ぶ形で Docker イメージを選択することができます。

- Android Java 8 24.4.1
- Docker 17.09.0
- Go 1.7.3
- Java 8
- Node.js 6.3.1, 4.4.7, 4.3.2
- Python 3.5.2, 3.4.5, 3.3.6, 2.7.12
- Ruby 2.3.1, 2.2.5
- .NET Core 1.1, 2.0

3.4 GHE / AWS（S3, CodeBuild, ECS）を利用した Re:VIEW CI 環境構築

GHE / AWS（S3, CodeBuild, ECS）を利用した Re:VIEW CI 環境構築の流れ

CI 環境構築の手順はざっくり次のようになります。

1. Re:VIEW プロジェクトを追加（GHE）
2. 証明書・成果物保管用ストレージを AWS 上に作成（S3）
3. CI の実行環境となる Docker イメージを AWS 上に構築（ECS）
4. CI プロジェクトを作成（CodeBuild）
5. CI ビルドスクリプトを作成
6. CI を試行
7. リポジトリの Webhook 設定
8. 流れの確認

手順 1 Re:VIEW プロジェクトを追加（GHE）

Re:VIEW で init して作っておきます。ルートに buildspec.yml というファイル名で、ビルド定義ファイルを追加しておきます。このビルド定義ファイルの内容は、Codebuild のプロジェクト作成画面のコピーで作成可能です。参考までに、リスト 3.1 に buildspec.yml の初期内容を紹介します。

▼リスト 3.1　参考：buildspec.yml の初期内容

```
version: 0.2

#env:
  #variables:
     # key: "value"
     # key: "value"
  #parameter-store:
     # key: "value"
     # key: "value"

phases:
  #install:
    #commands:
      # - command
      # - command
  #pre_build:
    #commands:
      # - command
      # - command
  build:
    commands:
      # - command
      # - command
  #post_build:
    #commands:
      # - command
      # - command
#artifacts:
  #files:
    # - location
    # - location
  #discard-paths: yes
  #base-directory: location
#cache:
  #paths:
    # - paths
}
```

手順 2 証明書・成果物保管用ストレージを AWS 上に作成（S3）

今回は、GitHub Enterprise の Webhook を利用するため、CodeBuild のビルドを実行するイメージに証明書をインストールできるようにする必要があります。CodeBuild では、この証明書を S3 のバケットと呼ばれるストレージに置き、CodeBuild 側で参照するように設定することで、証明書のインストールを CodeBuild のプロセスの中で自動的に

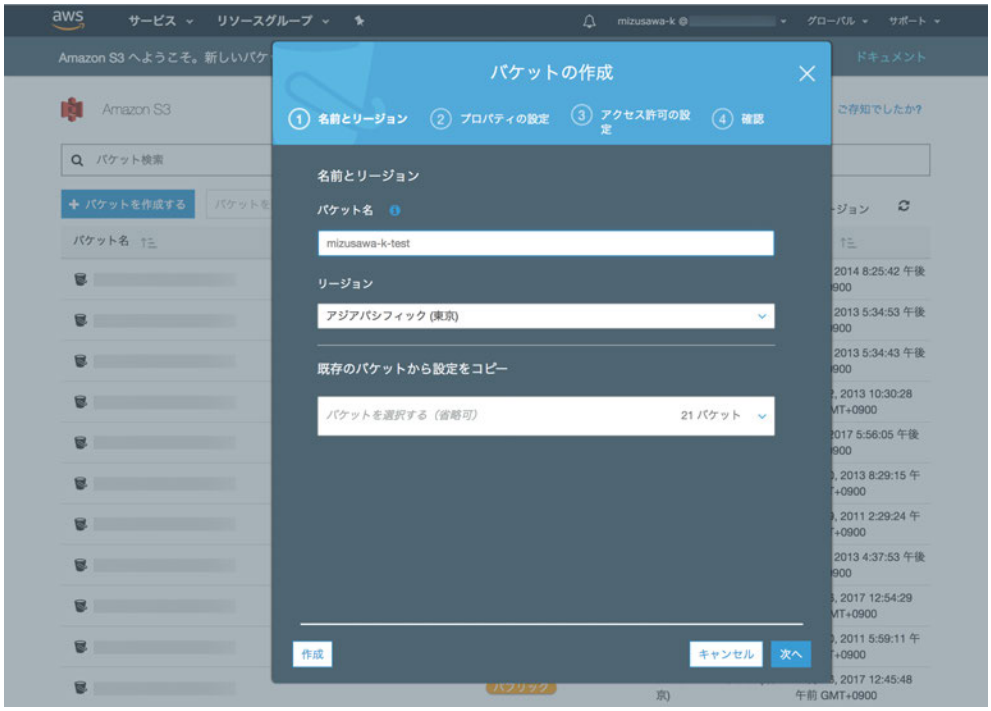
行ってくれます[*7]。

ここでは S3 バケットの作成フロー、手順について紹介します。

S3 バケットの作成フロー

■S3 のトップ画面を開く　AWS トップ画面から S3 のリンクを選択し、S3 のトップ画面を開きます。

■バケットを作成する　S3 のトップ画面で「バケットを作成する」を選択すると、バケットの作成をすることができます。バケットの作成では「バケット名」「リージョン名」を入力する必要があるため、順番に入れていきます。



▲図 3.4　AWS S3 バケット作成画面

「バケット名」は、今回は **mizusawa-k-test** としました。
「リージョン」は、今回は**アジアパシフィック（東京）**としました。
設定を進めていき、確認して問題なければ「バケットを作成」を選択します。

[*7] 今回利用した用途以外にも、S3 はビルドの成果物となる原稿の pdf データをクラウドサーバー上に保管しておきたい場合にも活用できます

▲図 3.5　AWS S3 バケット作成確認画面

手順 3 CI の実行環境となる Docker イメージを AWS 上に構築（ECS）

Re:VIEW をビルドする Docker イメージは、vvakame さんが提供している物[*8]を利用させてもらいました。

ECS リポジトリの作成フロー

まずは ECS のリポジトリを作成します。ECS のリポジトリ作成には、次のコマンドラインツールが必要なので、インストールしておきましょう。

- AWS CLI[*9]
- Docker
- pip, python

インストールしたら、順に進めていきます。

[*8] Re:VIEW image for Docker `https://hub.docker.com/r/vvakame/review/`
[*9] AWS CLI `https://aws.amazon.com/jp/cli/`

■ECS のトップ画面を開く　トップ画面を開きます。

■リポジトリの作成を選択する　リポジトリの作成を選択します。

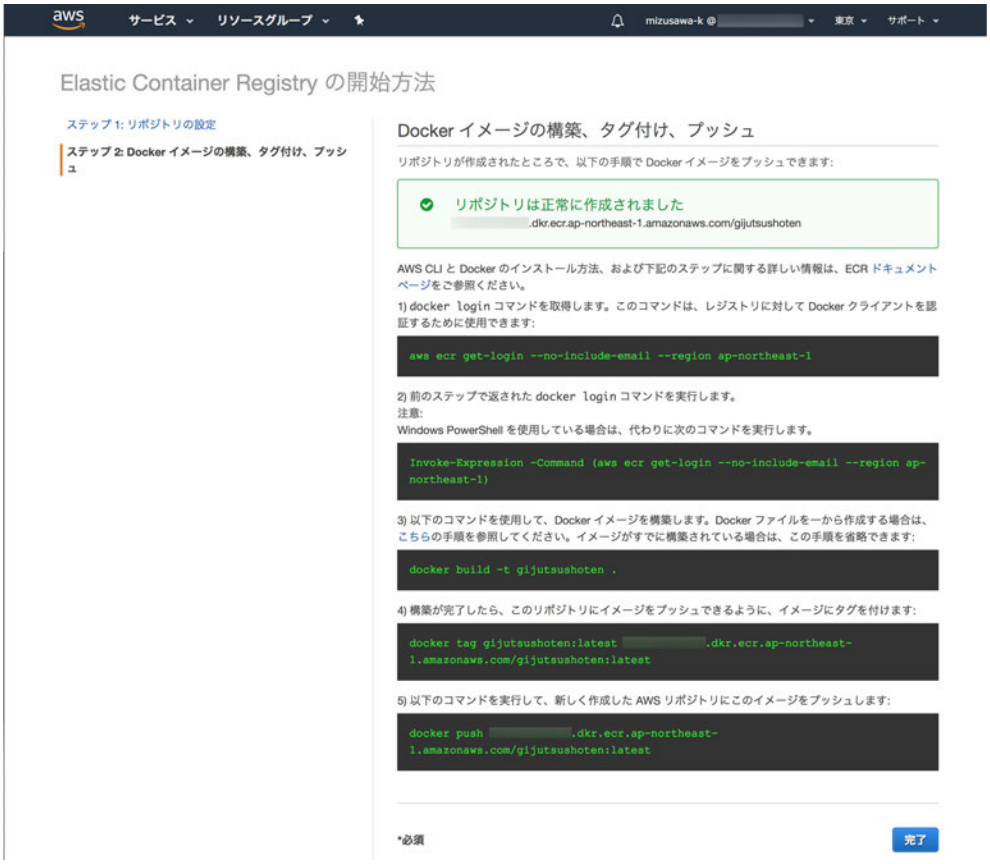
■リポジトリの設定　リポジトリの設定をして作成します。項目は「リポジトリ名」ひとつだけです。

▲図 3.6　AWS ECS リポジトリ作成画面

「リポジトリ名」には、識別できる名前を入れておきます。

■次へを選択する　次のステップを選択すると、AWS CLI のコマンドが表示されます。また、コマンドと合わせて、Docker イメージのプッシュ方法のヘルプページへのリンクも表示されています。リンク先のページは「Amazon ECR の使用開始[*10]」というヘルプページです。

[*10] Amazon ECR の使用開始
https://docs.aws.amazon.com/ja_jp/AmazonECR/latest/userguide/ECR_GetStarted.html



▲図 3.7　AWS ECS リポジトリ作成コマンド確認画面

■コマンドを実行していく　コマンドを順番に、ターミナルで実行し、リポジトリにイメージをプッシュしていきます。ひととおりコマンド作業が終わると、AWS ECS のコンソール画面でイメージが確認できるようになります。

■CodeBuild で利用できるようにアクセス許可を設定する　初期状態だと CodeBuild で利用できないアクセス状態になっているので、許可するように設定を追加していきます。

ECS のリポジトリを作成すると、AWS CLI で実行するプッシュコマンドが表示されるので、コマンドの実行を進めます。ちなみに、macOS で Docker イメージを作成する場合は VM の起動が必要なので、要注意です。

手順 4 CI プロジェクトを作成（CodeBuild）

CodeBuild の設定はプロジェクトという単位で行います。プロジェクトの設定を順番に進めていきます。

▲図 3.8　AWS CodeBuild プロジェクト作成画面

プロジェクトの設定

■プロジェクト名　識別しやすい名前を入れておきます。今回は、**mizusawa-k-test** としました。

■説明　プロジェクトの用途、目的などを入れておきます。今回は、検証用プロジェクトということを明記しておきました。

ソース: ビルドの対象

■ソースプロバイダ　次のバージョン管理システムの中から、どのシステムでプロジェクトファイルを取得するか選択することができます。

- Amazon S3
- AWS CodeCommit
- Bitbucket
- GitHub
- GitHub Enterprise

今回は、**GitHub Enterprise** を選択しました。

環境: ビルド方法

■環境イメージ　次のイメージを選択することが出来ます。

- AWS CodeBuild によって管理されたイメージの使用
- Docker イメージの指定

今回は、すでに用意した Docker イメージを利用するため、**Docker イメージの指定**を選択しました。

■環境タイプ　Linux のみ選択可能なので、**Linux** を選択しました。

■カスタムイメージタイプ　次のイメージタイプを選択することができます。

- Amazon ECR
- その他

今回はすでに作成してあるリポジトリを選択するため、**Amazon ECR** を選択します。

■Amazon ECR リポジトリ　すでに作成してあるリポジトリを選択します。今回は **gijutsushoten-4** を選択します。

■特権付与　インストールなどで必要な場合があるので、チェックを入れておきます。

■現在のビルド仕様　ソースコードのルートディレクトリに buildspec.yml を置くので、そのままにしておきます。

■証明書、証明書のバケット、証明書のオブジェクトキー　「証明書」項目にて、証明書が必要かどうかで次の選択肢を選びます。

- 証明書をインストールしない
- S3 から証明書をインストールする

今回は Webhook で使用するため、**S3 から証明書をインストールする**を選択します。「証明書のバケット」項目では、S3 のバケットを選択します。今回は **mizusawa-k-test** を選択しました。「証明書のオブジェクトキー」項目では、S3 のバケットにアップロードしておいた GHE の SSL 証明書である pem ファイルの名前を拡張子付きで入力します。今回は、**github-enterprise.pem** と入力しました。

アーティファクト: このビルドプロジェクトからアーティファクトを配置する場所

■タイプ　AWS CodeBuild では、ビルドの成果物をアーティファクトという名前で扱っています。「タイプ」項目ではこのアーティファクトを S3 に保存するかどうかを選択することができます。

- Amazon S3
- アーティファクトなし

今回は、ビルドの度に pdf を Slack に送信することから、S3 での保存が必要なかったため、**アーティファクトなし**を選択しました。

キャッシュ

■タイプ　AWS CodeBuild では、キャッシュの仕組みも持っているようです。次の選択肢からキャッシュの保存場所を選択できます。

- Amazon S3
- キャッシュなし

今回は、S3 をあまり使わないようにする為、**キャッシュなし**を選択しました。

VPC

■VPC　特に使用しない為、**No VPC** を選択します。

詳細設定

その他、細かい設定が可能です。

■タイムアウト　デフォルト（1 時間）のまま

■暗号化キー　デフォルトのまま

■コンピューティングタイプ

- 3 GB メモリ、2vCPU
- 7 GB メモリ、4vCPU
- 15 GB メモリ、8vCPU

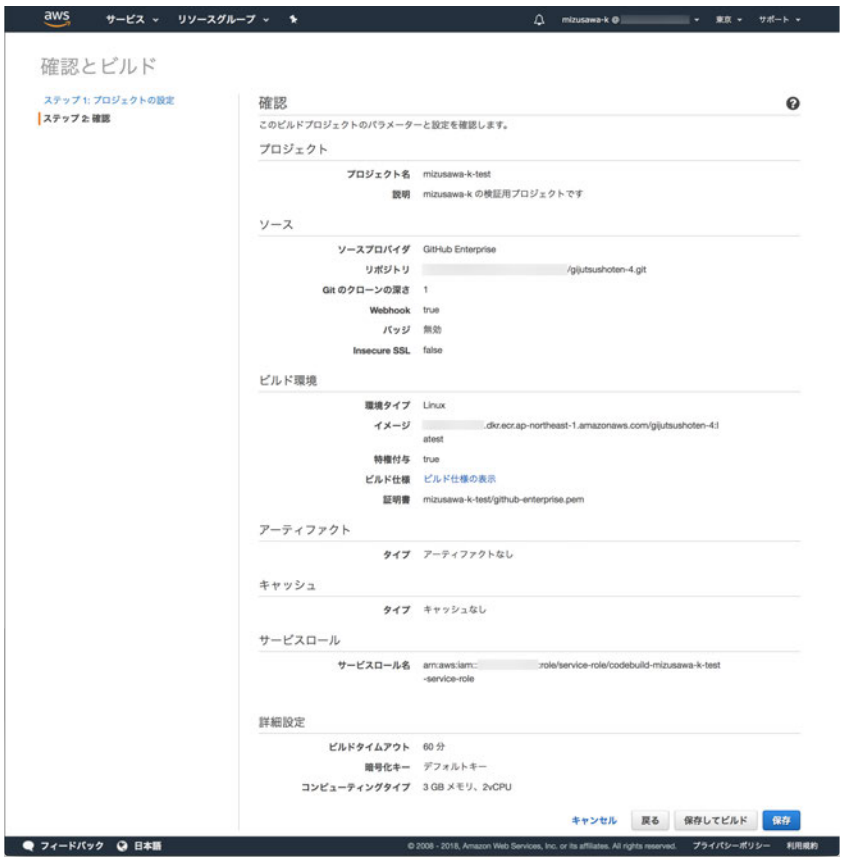
15 GB メモリ、8vCPU を選択

■環境変数　特に設定なし

■タグ　特に設定なし

確認、作成

以上をまとめると、今回は図 3.9 のように設定しました。



▲図 3.9　AWS CodeBuild プロジェクト作成画面

確認して、問題なければ保存を押すとプロジェクトが作成されます。

手順 5 CI ビルドスクリプトを作成

今回 AWS 上に構築した Docker マシンでは、必要なツールはすでに Docker イメージに入れてあります。また、原稿データを保管しているリポジトリのクローンは AWS CodeBuild 側で、ビルドスクリプトを実行する前にすでに行ってくれています。この為、ビルドスクリプトの中で書かなければならないことは次のみになります。

- pdf 化
- pdf を Slack へアップロード

結果として、次のようなビルドスクリプトリスト 3.2 になりました。それぞれ、順番に説明していきます。

▼リスト 3.2　参考：buildspec.yml の最終内容

```
version: 0.2

#env:
  #variables:
     # key: "value"
     # key: "value"
  #parameter-store:
     # key: "value"
     # key: "value"

phases:
  #install:
    #commands:
      # - command
  #pre_build:
    #commands:
      # - command
  build:
    commands:
      # build
      - . ./export-pdf.sh
  post_build:
    commands:
      # upload Slack
      - . ./upload-pdf-to-slack.sh
#artifacts:
  #files:
    # - locations
  #discard-paths: yes
  #base-directory: location
#cache:
  #paths:
    # - paths
```

pdf 化

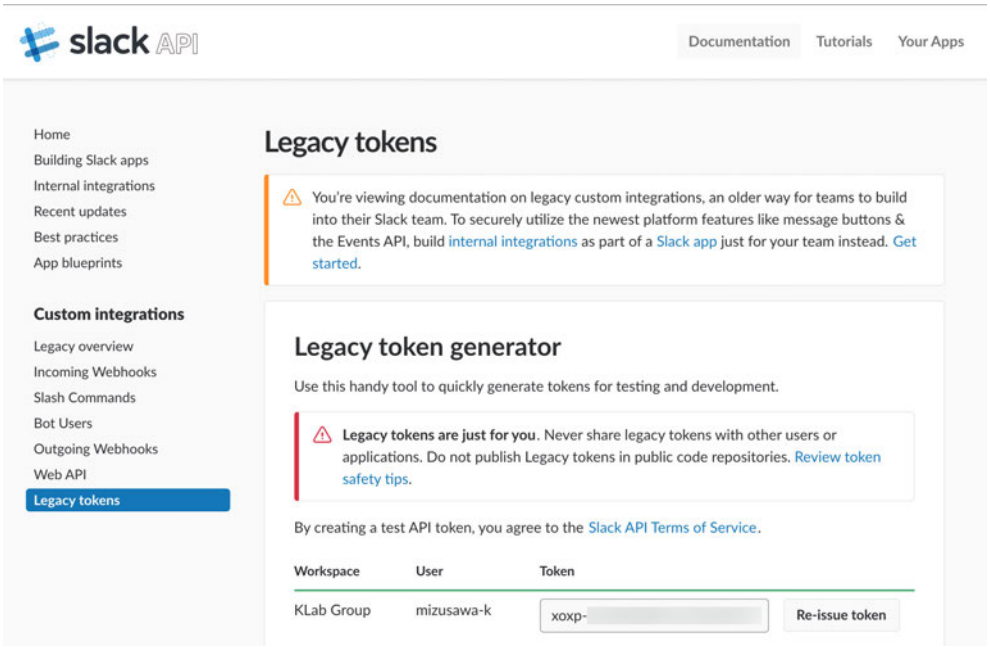
Re:VIEW で.re ファイルなどをまとめて、pdf 化するコマンドを記述します。今回は、.sh ファイルにコマンドをまとめておきました。

▼リスト 3.3　参考：pdf 化するシェルスクリプト

```
npm run pdf
```

pdf を Slack へアップロード

Slack への pdf アップロードには Slack の files.upload API[*11] を利用します。この API の利用には token が必要ですが、token の生成は Slack のサイト内でできます[*12]。



▲図 3.10　Slack Token 取得・再生成画面

今回は、pdf 化コマンドと同様、.sh ファイルにコマンドをまとめておきました。

[*11] Slack files.upload API `https://api.slack.com/methods/files.upload`
[*12] Slack Legacy Tokens `https://api.slack.com/custom-integrations/legacy-tokens`

▼リスト 3.4　参考：pdf を Slack にアップロードするシェルスクリプト

```
# ディレクトリ移動
cd articles

# 版情報作成
HISTORY=$(grep -C 0 history: config.yml)
HISTORY=$(echo "$HISTORY" | sed 's/^.*"\(.*\)".*$/\1/' #)
echo ${HISTORY}

# Slack 投稿用コメント作成
SLACK_COMMENT="【"${HISTORY}"】 の原稿が出来ました"

# Slack 投稿コメント確認
echo "Slack 投稿コメント確認"
echo ${SLACK_COMMENT}

# Slack 投稿
curl -F file=@gijutsushoten-4-klab.pdf \
     -F channels=gr-kgtechblog \
     -F token=xoxp---- \
     -F initial_comment=${SLACK_COMMENT} \
     https://slack.com/api/files.upload
```

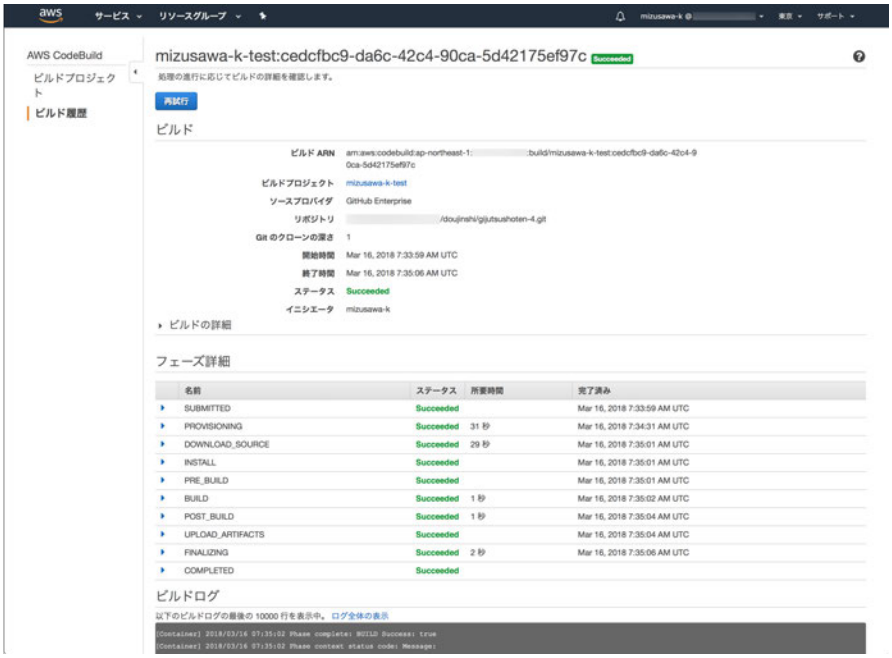
実際にポストしているのは curl のコマンドです。curl コマンドのパラメータにある token は、前述した Slack のサイトで生成した文字列を入れることになります。file で指定しているファイル名に@をつける必要がある点に注意です。channels で指定するチャンネル名には#をつける必要はないようです。

今回は、Slack 投稿時にファイルごとに版情報と Git のコミット ID が分かるように、initial_comment に含めるようにしておきました。

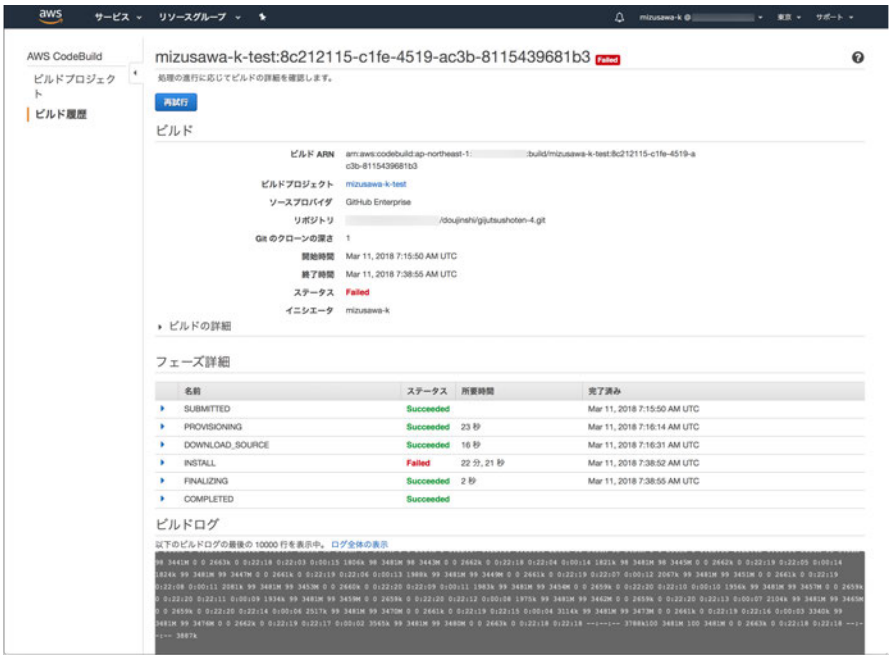
手順 6 CI を試行

ここまで来たら、一度ビルドを試行してみます。

ビルドの実行はボタンでできます。成功すると、SUCCESS と表示されます。途中、エラーが起きると即時にビルドが中止されて FAILED と表示されます。



▲図 3.11　CodeBuild で成功したビルド画面

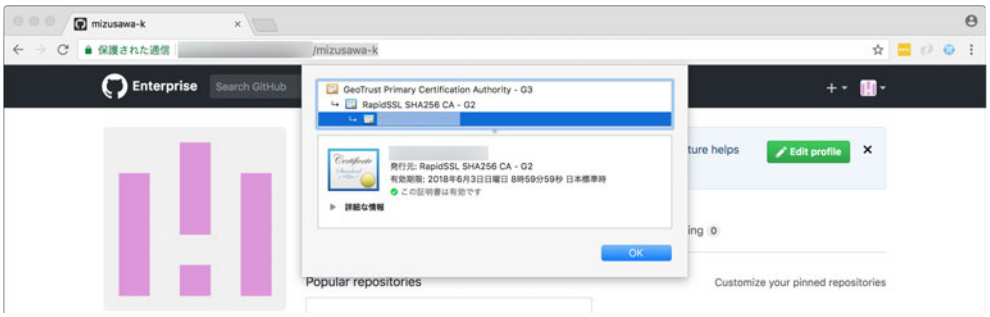


▲図 3.12　CodeBuild で失敗したビルド画面

手順 7 リポジトリの Webhook 設定

SSL 証明書設定

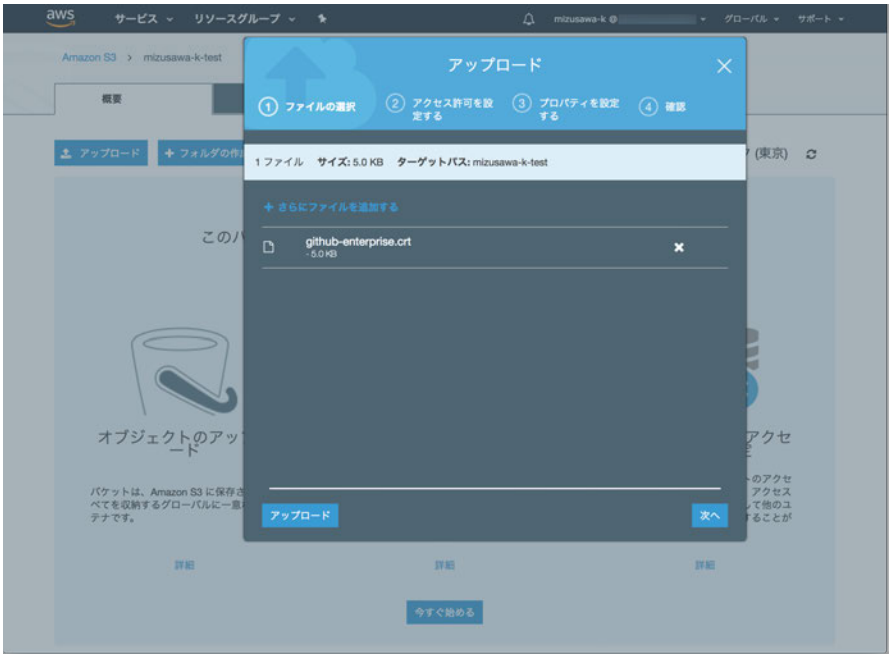
SSL 証明書（.pem ファイル）を、ブラウザで GitHub Enterprise のリポジトリサイトにアクセスして取得します。取得するために必要な操作は、OS, ブラウザによって異なるので注意が必要です。



▲図 3.13　ブラウザで GitHub Enterprise 画面を開き SSL 証明書をダウンロード

取得後は、作成した S3 のバケットにアップロードします。

▲図 3.14　SSL 証明書を S3 バケットにアップロード

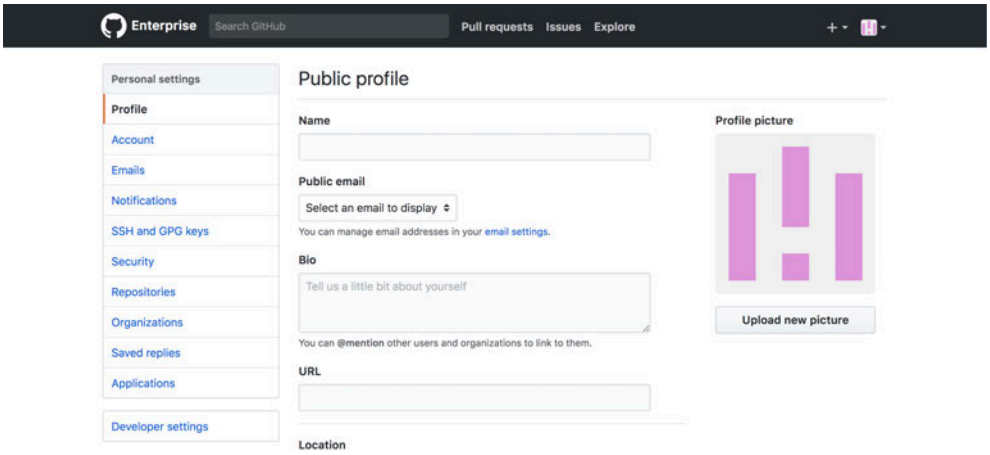


▲図 3.15　SSL 証明書を S3 バケットにアップロード

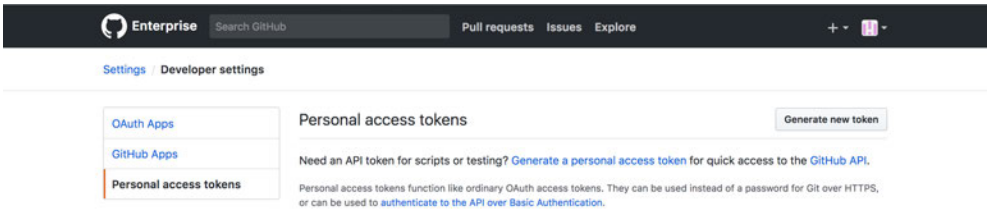
▲図 3.16　SSL 証明書を S3 バケットにアップロード

アクセストークン生成

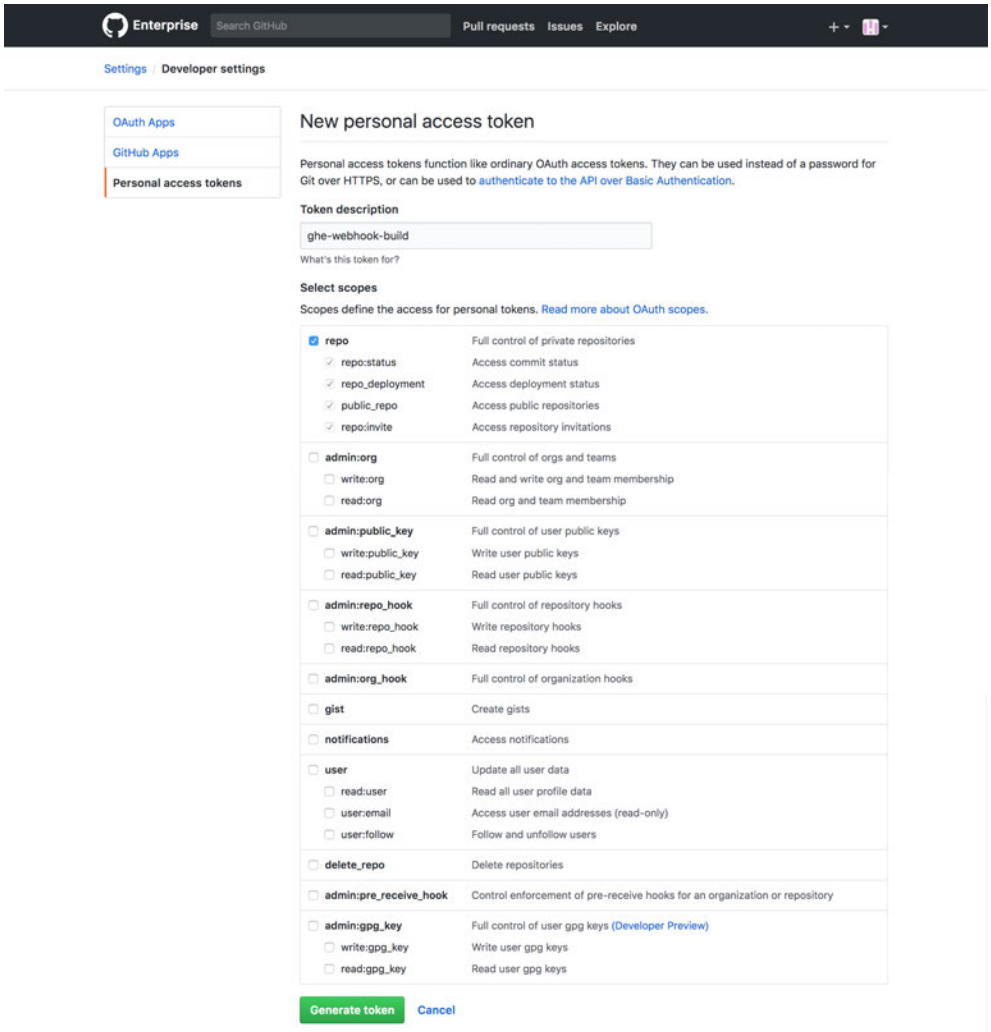
CodeBuild と GitHub のプッシュアクションを紐づけるために、GitHub のプロフィール画面、アクセストークンを生成します。



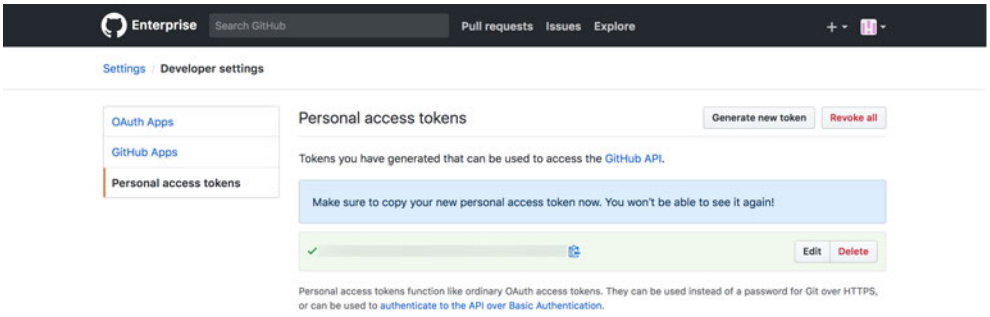
▲図 3.17　GitHub アクセストークン生成：Personal setting > Profile 画面で Developer setting を選択



▲図 3.18　GitHub アクセストークン生成：Developer settings > Personal access tokens 画面で Generate new token を選択



▲図 3.19　GitHub アクセストークン生成：Developer settings > Personal access tokens 画面で repo にチェックを入れて Generate token を選択

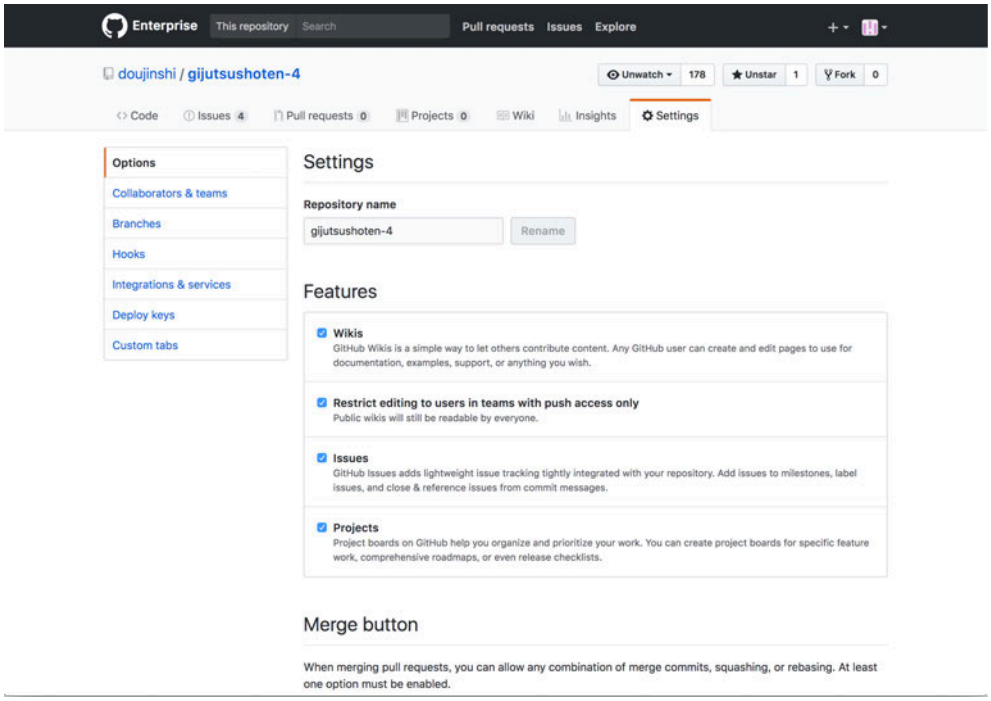


▲図 3.20　GitHub アクセストークン生成：Developer settings > Personal access tokens 画面で生成された token を確認

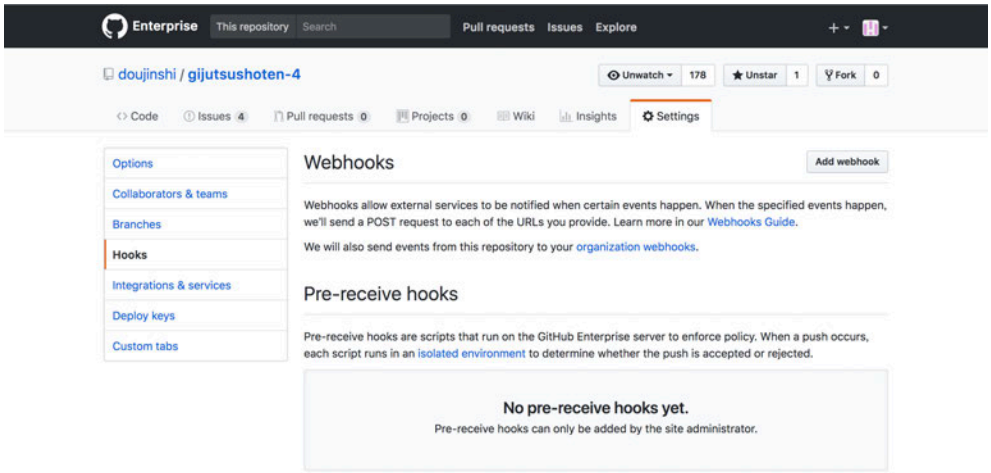
リポジトリの Webhook 有効化設定

リポジトリ自体に、Webhook の有効化設定を行います。GitHub のリポジトリの Webhook 設定画面を開き、CodeBuild で表示された「ペイロードの URL」と「シークレット」を入力して設定します。

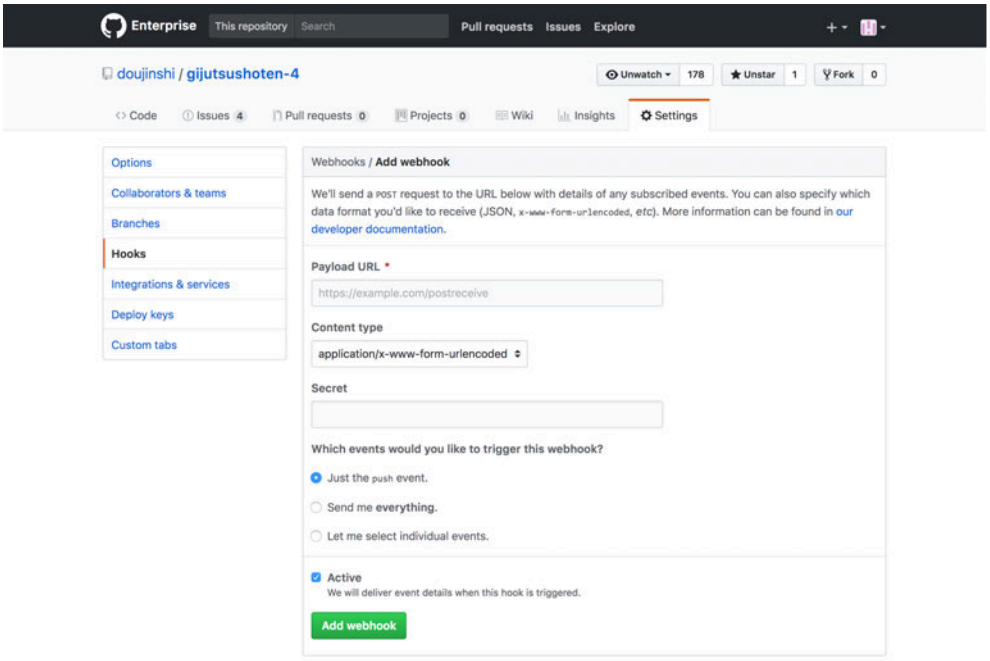
▲図 3.21　GitHub Webhook 設定画面



▲図 3.22　GitHub Webhook 設定画面



▲図 3.23　GitHub Webhook 設定画面



▲図 3.24　GitHub Webhook 設定画面

手順 8 流れの確認

ここまで来たらひととおりの作業は完了です。コミットをプッシュして、CodeBuild プロジェクトのビルドが実行され、Slack に pdf が共有されるまでの流れを通してみましょう。

3.5　CI 環境構築でハマったところ

ここまで CI 環境構築を進めていく中で、個人的にこれはハマったなというところをまとめておきます。

GitHub Enterprise の証明書のエクスポートができるブラウザが限定されていた

GitHub Enterprise の証明書をエクスポートで少々手こずりました。Mac マシンで作業をしていたところ、ブラウザで証明書をエクスポートする際は、Safari, Chrome ではエクスポートする UI が見当たらず、できませんでした。最終的には Firefox でエクスポートして対応しました。

AWS CodeBuild プロジェクトの作成でアカウントの権限追加が必要だった

プロジェクトの作成時、サービスロールを追加する必要があり、IAM アカウントの管理者権限が必要だったので付与するように対応しました。

GitHub Enterprise のソース URL が https じゃないと失敗した

SSH の URL 指定だと、DOWNLOAD_SOURCE のフェーズで cannot run ssh エラーと出てビルドが失敗していたので、https の URL を指定して暫定対応しました。公式ドキュメントでは例示の URL が https にはなっていましたが、特に注意書きがなかったためつまづいてしまいました。

AWS CodeBuild で用意している Docker（Ubuntu）に、日本語を含めた TeX のインストールができなかった

「パッケージ管理ソフトでインストールする」「ミラーサイトからダウンロードして展開、インストールする」いくつかの方法を試したものの、日本語フォントなどを含めたパッケージのインストールができない、そもそもパッケージ検索で出てこない問題がありました。また、提供されている Docker の Ubuntu のバージョンも 14.04 のみとなっていたことから、TeX 環境を自力で構築することは難しいと思いカスタムの Docker イメージによる環境構築で対応しました。

Slack のアップロードを Slack App（Bot）から行うことができなかった

当初は自分自身の Slack ユーザーアカウントではなく、Slack App（Bot）からアップロードしてもらう形で進めていたのですが、時間の都合上できませんでした。Bot で投稿する際の注意点としては、Bot 用の Slack アプリを作成して、ファイルのアップロード権限をつけ、Bot 用の Slack App の token で API を叩くようにする点だと思いますが、私が API 実行を試した時には token エラーが返って来ていました。token の取得場所を間違っていたか、どこかで見落としがあったのかなと思います。

今回はとりあえず pdf を Slack のチャンネルにアップロードできればいいかなということで、ユーザーアカウントからのアップロードのままにしましたが、この場合にちょっと怖いのは、アップロードしたユーザーアカウント自身が、自分がファイルをアップロードしている事に気付きにくいところです。

▲図 3.25　Slack ファイルアップロードおよび通常の発言ログの表示画面

Slack ではチャンネルに未読の発言が投稿された際、チャンネル名がハイライト表示されるのですが、今回のようにファイルアップロードを API で実行した際にはチャンネル名のハイライト表示はされません。API 実行時に指定している token 自体も、ファイルアップロード以外にも通常の発言にも使えるようですので、やろうと思えば、同じリポジトリを見ている人がいたずらでなりすますこともできてしまいます[*13]。Slack 画面の見かけ上では、本人が発言したのか API で発言したのかわからないので、Slack のユーザーアカウントの token は Git のリポジトリに保管しない方が安全そうです。

3.6　おわりに

意外と細かいところでつまづいてしまい、真顔になったケースも多かったですが、なんとかみんなの CI 環境が整ってよかったです。AWS CodeBuild はビルド時間で費用が決まる従量課金制なので、環境を作ってもビルドしなければそのまま放置しても費用が発生しないという点は環境構築のコストを考えると地味に嬉しいところなんじゃないかなと思います。

本章の内容が、技術書執筆、CI に関わる方々の御役に立てば幸いです。

Kinuko MIZUSAWA

[*13] token の更新をすればひとまずなりすまし投稿を止めることはできます:)

第 4 章

WebAssembly で行列の演算をする

4.1 はじめに

皆さんは WebAssembly にどんな夢を抱いていますか？ C/C++ 資産の活用、難読化、高速化といった話を私はよく耳にします。

C/C++ 資産の活用は特にわかりやすい用途です。たとえば、ゲームエンジンの Unity[*1]の WebGL 出力は IL2CPP[*2]で出力された C/C++ ソースコードを Emscripten[*3]を使用して WebAssembly に変換しています。特にゲームではディスプレイの更新頻度を維持するため 1 フレームの処理をを 16ms 以内に終えることが求められますが、WebAssembly を使用すればそれを目指すこともできます。

一方で WebAssembly を難読化目的に使うのはどうでしょうか。JavaScript はそもそもとして解析しやすく、モダンなブラウザには優秀な JavaScript デバッグツールが搭載されています。ゲームを実装するにあたって、ルールが解析されたり変数が書き換えられることは非常に大きな問題なのですが、JavaScript ではそれが簡単にできてしまいます。そのため、JavaScript ゲーム開発者には解析されることや改ざんされることを想定した設計が要求されます。WebAssembly はその点バイナリフォーマットなので JavaScript よりは解析されにくいようにも思えます。ですが、WebAssembly のデコンパイルツールが公式で公開されており、バイナリフォーマットから C 言語のソースコードへと簡単に戻せてしまいます。バイナリだから安全とはいえません。

では、高速化という点はどうでしょうか。WebAssembly を使ったとしても、すべての処理が WebAssembly バイナリの中で行われるわけではありません。たとえば、Emscripten を使用すれば OpenGL を使用したソースコードもコンパイルできます。しかし、Emscripten において OpenGL の API の大部分は JavaScript にて実装されていま

[*1] https://unity3d.com/

[*2] https://docs.unity3d.com/Manual/IL2CPP.html

[*3] https://github.com/kripken/emscripten

す[*4]。つまりは、WebAssembly を使用していたとしても JavaScript の API を呼ぶことを避けられません。

実際に WebAssembly を使用すると JavaScript より高速に処理できるのでしょうか、ちょっと試してみたいと思います。

4.2 4 行 4 列行列の乗算

ゲームグラフィクスでは 4 行 4 列の行列演算を多用します。

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix}$$

ベクトルを回転させたり、移動させたりといった処理はこの行列演算を用いれば簡単に行えます。回転する行列と移動する行列を乗算すれば回転した後に移動する行列を作成できます。GPU においてもこの 4 行 4 列行列は使用可能で、大量の行列演算には大きな需要があります。この行列の乗算をを JavaScript と Emscripten で実装し、両者のパフォーマンスを比較します。

4.3 JavaScript による実装

行列は WebGL でも使用する長さが 16 の `Float32Array`を使用します。これに列単位に行列要素を配置していきます。先の行列の場合だと配列には `[m00, m10, m20, m30, ..., m03, m13, m23, m33]`のように配置します。

次の行列であればリスト 4.1 のように表現します。ちょうどソースコードと実際の行列が転置の関係になっています。

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

▼リスト 4.1　Float32Array による 4 × 4 行列

```
const mat1 = new Float32Array([
  1, 5, 1, 5,
  2, 6, 2, 6,
  3, 7, 3, 7,
  4, 8, 4, 8
]);
```

[*4] https://github.com/kripken/emscripten/blob/master/src/library_gl.js

次に乗算を実装します。行列の乗算の実装は行と列をそれぞれ乗算したものを行列の成分とします。これを JavaScript で実装するとリスト 4.2 となります。

▼リスト 4.2　JavaScript による 4 行 4 列行列の積

```
const mat1 = new Float32Array([
  1, 5, 1, 5,
  2, 6, 2, 6,
  3, 7, 3, 7,
  4, 8, 4, 8
]);

const mat2 = new Float32Array([
  1, 5, 1, 5,
  2, 6, 2, 6,
  3, 7, 3, 7,
  4, 8, 4, 8
]);

const matResult = new Float32Array([
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1
]);

MultiplyMatrixJS(mat1, mat2, matResult);

// [34, 82, 34, 82, 44, 108, 44, 108, 54, 134, 54, 134, 64, 160, 64, 160]
console.log(matResult);
```

4.4　Emscripten による実装

まずは Emscripten をインストールしてください[*5]。

Windows 10 で動作確認しましたが、Windows、Mac、Linux どの OS でも構いません。

Emscripten で JavaScript から呼び出す関数を定義する場合、マングリングされてしまうと呼び出しが難しくなるため、`extern "C"`で関数をマングリングしないようにします。

行列は JavaScript での実装と同様に列単位で要素を格納します。バッファは WebAssembly のヒープから取得するため、C++ 側で `malloc`を呼び出します。それぞれの要素を 16 個の引数に取るようにしました。

また、C++ 側で確保したバッファは C++ 側で解放する必要があります。これらをまとめてリスト 4.3 に示します。

[*5] `https://kripken.github.io/emscripten-site/docs/getting_started/downloads.html`

▼リスト 4.3　行列の作成と解放

```
// 行列の作成
extern "C" float *CreateMatrix(
    float m00, float m10, float m20, float m30,
    float m01, float m11, float m21, float m31,
    float m02, float m12, float m22, float m32,
    float m03, float m13, float m23, float m33)
{
  auto p = static_cast<float *>(malloc(sizeof(float) * 16));

  p[0] = m00; p[1] = m10; p[2] = m20; p[3] = m30;
  p[4] = m01; p[5] = m11; p[6] = m21; p[7] = m31;
  p[8] = m02; p[9] = m12; p[10] = m22; p[11] = m32;
  p[12] = m03; p[13] = m13; p[14] = m23; p[15] = m33;

  return p;
}

// 行列の解放
extern "C" void ReleaseMatrix(float *p)
{
  free(p);
}
```

行列の乗算も C++ で実装します（リスト 4.4)。左辺行列 a と右辺行列 b を受け取って作成済みの行列 dst に値をセットします。この実装はリスト 4.2 とほぼ同じ実装となっています。引数は float 型へのポインタとなっています。

▼リスト 4.4　行列の乗算

```
extern "C" void MultiplyMatrix(float *a, float *b, float *dst)
{
  for (auto x = 0; x < 4; x++)
  {
    for (auto y = 0; y < 4; y++)
    {
      auto i = x * 4;
      dst[i + y] =
          a[0 + y] * b[i + 0] +
          a[4 + y] * b[i + 1] +
          a[8 + y] * b[i + 2] +
          a[12 + y] * b[i + 3];
    }
  }
}
```

これら WebAssembly の関数を呼び出す JavaScript で、Module オブジェクトを定義します。WebAssembly の実行後に呼び出される `postRun`関数にて WebAssembly の関数を取得します。JavaScript から WebAssembly の関数を呼び出すための関数を `cwrap` 関数で取得します。`cwrap`関数は関数名、リターン型、引数型を引数にとります。

▼リスト 4.5　JavaScript からのモジュール呼び出し

```
var Module = {
  postRun: () => {
    // CreateMatrix は number 型を 16 個とる。
    const createMatrixArgs = [
        'number', 'number', 'number', 'number',
        'number', 'number', 'number', 'number',
        'number', 'number', 'number', 'number',
        'number', 'number', 'number', 'number'
    ];

    const CreateMatrix = Module.cwrap('CreateMatrix',
      'number', createMatrixArgs);

    const MultiplyMatrix = Module.cwrap('MultiplyMatrix',
      '', ['number', 'number', 'number']);
  }
};
```

CreateMatrix関数は引数に行列要素を number型で取り、作成した行列をポインタを number型で返します。MultiplyMatrix関数の引数には CreateMatrixで取得したポインタを渡します。このポインタは Module.bufferからのバイトオフセットとなっているため、Float32Arrayを通じて配列アクセスができます（リスト 4.6）。

▼リスト 4.6　CreateMatrix から Float32Array を取得

```
const pMat1 = CreateMatrix(
  1, 5, 1, 5,
  2, 6, 2, 6,
  3, 7, 3, 7,
  4, 8, 4, 8
);

// [1, 5, 1, 5, 2, 6, 2, 6, 3, 7, 3, 7, 4, 8, 4, 8]
const mat1 = new Float32Array(Module.buffer, pMat1, 16);
```

ビルド時はリスト 4.7 のコマンドでビルドします。

EXPORTED_FUNCTIONSにエクスポートする関数を列挙するのですが、この際に関数名の先頭にアンダースコアをつけるのを忘れないようにします。また、JavaScript からランタイムの cwrap関数を呼び出すために EXTRA_EXPORTED_RUNTIME_METHODSに cwrapを列挙します。

▼リスト 4.7　Emscripten でのビルド

```
em++ m4mul.cpp -o m4mul.js \
  -Wall \
  -std=c++11 \
  -O3 \
  -g0 \
  -s WASM=1 \
  -s EXPORTED_FUNCTIONS="[ \
    '_CreateMatrix', \
    '_MultiplyMatrix', \
    '_ReleaseMatrix' \
  ]" \
  -s EXTRA_EXPORTED_RUNTIME_METHODS="[ \
    'cwrap' \
  ]"
```

ビルドした WebAssembly を使用して行列の乗算を行ってみました（リスト 4.8）。リスト 4.2 と同じ結果を得ることができました。

▼リスト 4.8　Emscripten での演算結果

```
const pMat1 = CreateMatrix(
  1, 5, 1, 5,
  2, 6, 2, 6,
  3, 7, 3, 7,
  4, 8, 4, 8
);

const pMat2 = CreateMatrix(
  1, 5, 1, 5,
  2, 6, 2, 6,
  3, 7, 3, 7,
  4, 8, 4, 8
);

const pMatResult = CreateMatrix(
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1
);

MultiplyMatrix(pMat1, pMat2, pMatResult)

// [1, 5, 1, 5, 2, 6, 2, 6, 3, 7, 3, 7, 4, 8, 4, 8]
console.log(new Float32Array(Module.buffer, pMatResult, 16));
```

4.5　ベンチマーク

それでは JavaScript と WebAssembly のパフォーマンスを計測してみます[*6]。

それぞれ 3 秒間行列の演算を繰り返し、行列の乗算関数を何度呼び出すことができたか

[*6] https://nyamadan.github.io/m4mul/

を計測してみます。

▼リスト 4.9　JavaScript による実装のベンチマーク

```
const ms = 3000;
let count = 0;
for (const t = Date.now() + ms; Date.now() < t; count++) {
  MultiplyMatrixJS(mat1, mat2, matResult);
}

const score = Math.floor(count / ms);
```

▼リスト 4.10　WebAssembly による実装のベンチマーク

```
const ms = 3000;
let count = 0;
for (const t = Date.now() + ms; Date.now() < t; count++) {
  MultiplyMatrix(pMat1, pMat2, pMatResult);
}

const score = Math.floor(count / ms);
console.log(score);
```

リスト 4.9 とリスト 4.10 の結果は、筆者環境の Google Chrome 65 にて JavaScript は 3380、WebAssembly は 4507 のスコアとなりました。Edge や Firefox も試してみましたが同様に WebAssembly のほうがよいスコアとなりました。

4.6　まとめ

今回の実験で行列演算のようなちょっとした演算でも WebAssembly を使用したほうがパフォーマンスとしてよいということがわかりました。一方で WebAssembly で確保したメモリは `free`関数で解放する必要があるため、その点は注意が必要です。

WebAssembly を使って Web の世界をもっともっと高速化していきたいですね。

やまだ

第 5 章

Python インスタンスの属性は辞書（dict）で管理されているのか調べてみた

聞いたことはあったものの、日常に影響はそうないため確認せずに済ませていたことがありました。Python プログラミング言語において、クラスインスタンスの属性の管理には組み込み型である辞書（dict）そのものが使われているらしい、と。

公式ドキュメント、言語リファレンスのデータモデルには次のように書かれています[*1]。

> 属性アクセスのデフォルトの動作はオブジェクトの辞書から値を取り出したり、値を設定したり、削除したりするというものです。たとえば、a.xによる属性の検索では、まず a.__dict__['x']、次に type(a).__dict__['x']、そして type(a) の基底クラスでメタクラスでないものに続く、といった具合に連鎖が起こります。

「デフォルトの」という但し書きがあるように、__slots__をもつクラスは辞書をもたなくなる、__getattribute__をもつクラスはこれを用いて属性の解決をおこなうようになる、など動作が書き換わることはありますが、通常は辞書で管理されているのだと読み取れます。

そう、ここまでの理解でした。とはいえ、疑問は残ります。インスタンスの__dict__属性が「Python からみると辞書にみえる別のなにか」かもしれないという可能性。であるならば、どのように実装されているものか直接見てみよう、と思い立ちました。Python の一実装、C 言語で作られた CPython のコードは公開されているのですから。

今回は Python 3.6.5[*2]のコードを追うことにします。

[*1] https://docs.python.jp/3/reference/datamodel.html

[*2] https://github.com/python/cpython/tree/v3.6.5

5.1 dis - バイトコードの逆アセンブラ

インスタンス属性がどこにどのように保持されているのかを探すのも良さそうですが、今回はインスタンス属性に触れるコードがどのようなバイトコードにコンパイルされるのかから追ってみることにしました。

そのためのライブラリは標準で用意されています。dis モジュールです。dis.dis関数にコード片を与えるとコンパイル、そして逆コンパイルからの解析が行われます。

```
In [1]: import dis

In [2]: dis.dis('spam.ham')
  1           0 LOAD_NAME                0 (spam)
              2 LOAD_ATTR                1 (ham)
              4 RETURN_VALUE
```

これにより spam.hamというコードは LOAD_NAME・LOAD_ATTR・RETURN_VALUEという3つのバイトコード命令になることがわかりました。LOAD_ATTR命令の説明をマニュアルで調べると次のように書かれています[*3]。

> TOSを getattr(TOS, co_names[namei])と入れ替えます。

TOSが何なのかは置いておいて[*4] 。getattrとありこれが属性を取り出す命令であることがわかりました。そしてこれらの命令名でコードを探るとPython/ceval.c というファイルにたどり着きます。

5.2 ceval.c - バイトコードの実行処理

早速 ceval.c ファイルを開きました。先頭には“Execute compiled code”とのコメントがあります。バイトコードをどのように実行するのかが書かれたコードと踏んで読みすすめます（リスト 5.1)。

ここで PyObject_GetAttr関数が見つかりました。C 言語で Python 拡張を作る際、オブジェクトから属性を取り出すために使う関数と同じのものです。普段のバイトコードの実行処理で使われる関数と C 製拡張でつかう関数が同じであるということにたどりつき、Python を使うだけでなく少しは中身が読めるようになった楽しさを味わいつつ、読み進めます。

▼リスト 5.1　ceval.c - LOAD_ATTR

[*3] https://docs.python.org/ja/3/library/dis.html#opcode-LOAD_ATTR

[*4] TOS は現在実行中の VM におけるスタックの先頭を指すものだそうです。つまり、事前に積まれているオブジェクトから属性を取り出して同じ位置に置き直すのですね。

```
TARGET(LOAD_ATTR) {
  PyObject *name = GETITEM(names, oparg);
  PyObject *owner = TOP();
  PyObject *res = PyObject_GetAttr(owner, name);
  Py_DECREF(owner);
  SET_TOP(res);
  if (res == NULL)
      goto error;
  DISPATCH();
}
```

5.3 object.c - PyObject_GetAttr の実装

PyObject_GetAttrの実装の一部です（リスト 5.2)。

▼リスト 5.2 object.c - PyObject_GetAttr

```
PyObject *
PyObject_GetAttr(PyObject *v, PyObject *name)
{
  PyTypeObject *tp = Py_TYPE(v);
  // 変数宣言を一部省略
  if (tp->tp_getattro != NULL)
    return (*tp->tp_getattro)(v, name);
```

PyTypeObjectの tp_getattroが指している関数を呼び出しています。NULLチェックを行っていることから可変、かつ NULLになることもあるのでしょうか。公式ドキュメントを調べると PyTypeObject.tp_getattroそのものの説明が見つかりました[*5]。

> オプションのポインタで、get-attributeを実装している関数を指します。シグネチャは PyObject_GetAttr()と同じです。通常の属性検索を実装している PyObject_GenericGetAttr()をこのフィールドに設定しておくとたいていの場合は便利です。

ここに独自の関数を入れることで属性検索の方法を変えることが可能であるものの、通常は PyObject_GenericGetAttrが収められていると推測されます。ではこれも読んでみます（リスト 5.3)。

▼リスト 5.3 object.c - PyObject_GenericGetAttr

```
PyObject *
PyObject_GenericGetAttr(PyObject *obj, PyObject *name)
{
    return _PyObject_GenericGetAttrWithDict(obj, name, NULL);
}
```

[*5] https://docs.python.org/ja/3/c-api/typeobj.html#c.PyTypeObject.tp_getattro

なるほど、_PyObject_GenericGetAttrWithDictの dict引数に NULLを指定して実行しています。ではこちらも（リスト 5.4)。

▼リスト 5.4　object.c - _PyObject_GenericGetAttrWithDict

```
PyObject *
_PyObject_GenericGetAttrWithDict(PyObject *obj, PyObject *name, PyObject *dict)
{
    PyTypeObject *tp = Py_TYPE(obj);
    PyObject *res = NULL;
    Py_ssize_t dictoffset;
    PyObject **dictptr;
    // 中略 tp_descr_get を探し、あればこれを使って値を取得。失敗したら次へ
    if (dict == NULL) {
        dictoffset = tp->tp_dictoffset;
        if (dictoffset != 0) {
            dictptr = (PyObject **) ((char *)obj + dictoffset);
            dict = *dictptr;
        }
    }
    if (dict != NULL) {
        Py_INCREF(dict);
        res = PyDict_GetItem(dict, name);
```

属性が記録されているオブジェクトの位置を算出し、そこから PyDict_GetItemで値を取り出していました。事前にオプションのデスクリプタ tp_descr_getの実行があり、属性アクセスのデフォルト処理が上書きされているケースへの対処が見られます。また、tp_dictoffsetが 0 でないことの確認もあり、属性用辞書を持たないケースへの対処も見られました。

では最後にこの dict変数の中身が本当に組み込み型の辞書なのかどうかを調べていきます。dict変数は PyDict_GetItem関数に渡され、そこではまず PyDict_Checkマクロが呼ばれています（リスト 5.5)。

▼リスト 5.5　dictobject.c - PyDict_GetItem

```
PyObject *
PyDict_GetItem(PyObject *op, PyObject *key)
{
    if (!PyDict_Check(op))
        return NULL;
```

このマクロで辞書かそのサブクラスであることの型確認が行われていました（リスト 5.6)。

▼リスト 5.6　dictobject.h - PyDict_Check

```
#define PyDict_Check(op) \
                PyType_FastSubclass(Py_TYPE(op), Py_TPFLAGS_DICT_SUBCLASS)
```

つまり、dict変数の中身は組み込みの辞書またはそのサブクラスであることがわかりました。

5.4 おわりに

というわけで、インスタンス属性の管理には組み込み型の辞書（dict）またはそのサブクラスが使われるというところまでコードを追いかけることができました。今回読んだ範囲では辞書そのものであるかどうかの確認はとれなかったのですが、人から聞いた、ドキュメントで読んだという段階から一歩進むことができたのではないかと思います。もう一歩踏み込むにはインスタンスを作成する際のメモリ確保と初期化の処理を追う必要がありそうです。

Shunsuke Ito / @fgshun

第 6 章

リアルの街を Unity に取り込む ～GIS へのいざない～

実際の街を広範囲にゲームに取り込みたいという要望はさまざまなところにあります。当然まともにゼロから取り組めば、すべてのモデルを作成するという途方もない作業量が必要になります。そのため多くの場合地図データなどから自動生成することになります。

しかしながら地図データは有料のものが多く、またゲームエンジンに簡単に取り込んで無償で使えるものにはさまざまな制約がついていたりします。

本稿では、実際の街を国土地理院などが公開しているオープンなデータセットから、フリーウェアや OSS を活用してさまざまなデータ処理を行い、Unity にデータとしてインポートするまでを解説します。（一部でシェアウェアなども使いますが、その工程については OSS の同等なソフトウェアなどで代替可能です。）その過程で地理情報と呼ばれる位置や場所を表すデータの扱い方や勘所と、地理情報を扱うための GIS（Geographic Information System）と呼ばれるソフトウェアの概念や使い方などを解説していきたいと思います。

対象として東京近辺を例に進めますが、本稿の手法は日本全国どこでも適用できると思われます。海外の街に関しては、同様なオープンデータセットが存在するかにもよりますが、工夫次第で同様に適用できると思われます。

本稿が皆さんの創造力の一助になれば幸いです。

6.1　商用、先行事例など

はじめに、商用などですでに実際の街をゲームに取り込むことのできるソリューションを紹介します。もし、これらのうちどれかを使うことができる（用途的にも経済的にも）なら、迷わず使いましょう。地理情報の処理は専門的な知識が必要になり、また大きなデータセットを扱うことになります。PC のリソース要求も高いし、処理時間も長時間にわたります。本稿の執筆中も処理が終わらなくて数日かけっぱなしにしてそれでもうまくいかなかったこともありました。とにかく、使えるものがあるなら使った方がよいです。

株式会社ゼンリン　3D 都市モデルデータ

株式会社ゼンリンが提供している 3D 都市モデルデータです[*1][*2]。
サンプルとしていくつかの街のデータが Unity のアセットストアから利用可能です。

ESRI 株式会社　CityEngine

ESRI 株式会社が提供している、3D 都市景観モデリングソフトウェアです[*3]。
ルールに基づいて、自動的に都市構造を生成する機能などがあります。実際の地理情報データに基づいてルールを設定すれば、実際の街並みに近いモデルを作成できます。

mapbox

mapbox[*4]は、カスタムマップがメインのサービスですが、幅広く地理情報を扱っています。Unity 用の Asset[*5]があり、mapbox のデータをもとにして、Unity 内に実在の街を取り込むことができます。見た目を変えることもでき、さまざまな応用が考えられます。

WRLD

WRLD[*6]も、同様に実際のカスタムマップなど地理情報サービスを提供しています。こちらの UnitySDK[*7]で、実在の街を Unity 内に取り込むことができます。見た目を変えることもでき、こちらもさまざまな応用が考えられます。

Google Maps API for Gaming

先日（2018 年 3 月 14 日）発表されたサービス[*8]です。Google Maps のデータをゲーム内で使えるようにするものです。もちろんゲームに合わせたカスタマイズなどもできるようです。期待が高まります。

[*1] 株式会社ゼンリン 3D 都市モデルデータのページ：http://www.zenrin.co.jp/product/service/3d/index.html
[*2] 株式会社ゼンリン ZENRIN City Asset Series：http://www.zenrin.co.jp/product/service/3d/asset/index.html
[*3] ESRI ジャパン株式会社 CityEngine：https://www.esrij.com/products/esri-cityengine/
[*4] mapbox サイト：https://www.mapbox.com/
[*5] mapbox UnityAsset のページ：https://www.mapbox.com/unity/
[*6] WRLD サイト：https://www.wrld3d.com/
[*7] WRLD UnityAsset のページ：https://docs.wrld3d.com/unity/latest/docs/api/
[*8] Google Maps APIs for games のページ：https://developers.google.com/maps/gaming/

その他

その他にも、衛星画像の販売や地図データの販売など、ニッチな領域ではありますが、老舗から新興ベンチャーまで多くの会社があります。目的に合わせて検討してみてください。

6.2 自前でデータを用意する意味

これだけさまざまなサービスがあるのに、それでも自前で用意しようと思った物好きなあなた。考え直すなら今のうちですよ。

自前でデータを用意するのはとにかく大変ですし、やはり精度や詳細さで商用のものにかなわない点は多くあります。しかし、自前で用意したデータを基データに「東京と同じ配置の中世風の街並み」だったり、「必要なところだけ簡素化して軽量化し、アプリバイナリに含める」などのような、必要に応じてさまざまな調整や修正が可能なところは、大きな利点だと思います。また、こうした地理空間情報の扱い方を理解しておくことは、商用のソリューションを使う際にも大きな助けになるでしょう。

6.3 地理情報の基礎知識

作業に入る前に、地理情報を扱う上での基本的なところを確認しておきましょう。

座標・緯度経度・測地系・座標系

地物（地上にあるすべてのものを表す地理学用語です）が、地球のどこにあるかを表すには、座標が要ります。もちろん皆さんは、緯度経度という表し方をご存知だと思いますが、実は同じ緯度経度でも違う場所を表す場合があることをご存知でしょうか。

緯度経度は、地球を球体（正確には回転楕円体）としたときの、球面座標（これも正確な言い方ではないですが）のふたつの角度で地物を表す座標系です。しかし、緯度経度には重要なパラメーターとして測地系というものが付随します。測地系は、その緯度経度がどの回転楕円体を基にしたものなのかを規定したものです。測地系には、回転楕円体のパラメータと、座標系の原点と軸の方向、ジオイド面の三つが含まれています。これによりある地点を正確に表すことができます。そのため、もとにする測地系が違えば同じ緯度経度でも違う場所を表すことになるのです。

複数の測地系がある理由としては、地球は正確な回転楕円体ではなくいびつな形をしているため、各国が自分の国の地域都合や測量の歴史的経緯により、都合のよい回転楕円体を設定していました。

しかし、GPS の発展で全地球的な座標系が必要になり、WGS84 と呼ばれる GPS で用いる回転楕円体の設定に合わせた形での測地系に、各国で切り替えるようになりました。

日本でも旧日本測地系という日本ローカルの測地系だったものが、2002 年に日本測地系 2000 と呼ばれる GPS の用いる測地系に合わせたものに変更されました。ちなみに、旧日本測地系と日本測地系 2000 では、東京近辺では同一緯度経度で 450m ほどのずれがあります。

日本測地系 2011

東日本大震災による地殻変動から、2011 年に国土地理院は新しい測地系、「日本測地系 2011」を公開しました。測地系はさまざまな要因で現実に合わせて変わります。旧日本測地系から日本測地系 2000 への変更は、技術の発展で GPS などにより地球上の位置を知ることができるようになったためでした。一方で、日本測地系 2000 から日本測地系 2011 への変更は、東日本大震災による地殻変動で測量の基準となる三角点や水準点が移動してしまったのを補正するため行われました。

地図の投影法・図法

今回は三次元で取り込むことを考えていますし、特に地球の球面の影響が大きく出るような広い範囲を取り込みませんが、「地図」という二次元の形で表現するときに気を付けなければならないのが、投影法です。地球という三次元球面を二次元に投影するため、そのままでは大きなゆがみが生じるものを、地域や用途に応じて投影方法を工夫し、実用的な地図に表現するための図法が各種あります。代表的なものにメルカトル図法があります。

Google マップの投影法

Google マップでは、メルカトル図法を使っています。[a] ただし、緯度 +/-85 度で切り取られています。メルカトル図法では、円筒に投影しているため緯度方向が無限の長さになってしまうため、日常では使われることのない北極と南極の領域が切り取られています。メルカトル図法は角度が正しく描画されるため、十分狭い範囲だけを見ると正しく形を描画できます。そのため、小さな範囲を切り取って使われることの多い Web 地図やスマフォの地図アプリなどでは便利な投影法として使われています。

[a] `https://developers.google.com/maps/documentation/javascript/maptypes?hl=ja#WorldCoordinates`

GPS

GPS は、上空を周回する人工衛星からの電波を複数比較して、三角測量の原理で自分の絶対位置を計測します。このときに取得できる緯度経度は、特殊な設定がなされていなければ WGS84 という測地系を用いています。WGS84 は実用上は、日本測地系 2000 および日本測地系 2011 とほぼ同等です。

ジオイドと標高

GPS の計測した「高度」の値と測量に基づく地図の「標高」にはずれがあります。これは、GPS での高度は測地系で定義される楕円体表面からの高さであるのに対して、地図の標高は平均海水面からの高さとなるからです。この平均海水面にもっとも近い「重力の等ポテンシャル面」のことを「ジオイド」といい、標高はこのジオイド面からの高さとなります。地図の標高と GPS の高度を合わせて処理するときは気を付けなければならない点です。

GIS

地理的な位置に関する情報を持ったデータを、総合的に編集・加工したり分析したりなどをするソフトウェアやシステムのことを指します。今回は、OSS の GIS ソフトウェアである QGIS を使っていきます。他にも、老舗の OSSGIS で分析などに長けた GRASS や、Java の GIS ツールキットライブラリの GeoTools などもあります。また、商用のものでは ESRI 株式会社の ArcGIS が有名です。

ラスタとベクタ

地理情報にも画像のようなラスタとベクタの二種類があります。航空写真、衛星画像のような、ピクセルごとに値をもつタイプと、道路のネットワークや、市区町村境界のような、幾何学形状をデータとしてあらわしたものです。それぞれラスタとベクタと分類します。

ベクタデータの構成

ベクタデータは、幾何学的形状を表すデータと、各形状に紐付く属性データとになっています。たとえば、都道府県のデータがあったとして、都道府県それぞれの形が幾何学的形状のデータとして格納されます。この場合はポリゴンやエリアといった形で格納されます。そして、それぞれの都道府県の名前、人口、面積といった情報が、属性データとしてデータベースのような形で紐付けられます。このふたつのデータが紐付けられていることが GIS の大きな価値となっています。

ファイルフォーマット

ラスタの地理情報を格納するのには、GeoTIFF というファイルフォーマットがよく使われます。TIFF 画像フォーマットにメタデータとして地理情報の諸元を格納したものです。一方、ベクタの地理情報は Shape ファイルという ESRI 社の開発したフォーマットが事実上の標準となっています。データ交換の際は、これらのファイルフォーマットに変換するとよいでしょう。

またデータベース関連では、PostgreSQL に PostGIS[*9] という地理情報を扱うための定評のある拡張があります。MySQL でも 5.6 あたりから地理情報を扱うまともな仕組みが実装され始めています。[*10]

Shape ファイルについて

Shape ファイルは、実際には複数のファイル郡から成り立つファイル形式です。地理情報の幾何学的情報を格納しているのが、.shp ファイルで、.dbf が属性値を格納しています。この.dbf ファイルは、dBASE というデータベースシステムのファイルフォーマットです。そして、.shx ファイルが.shp と.dbf のデータを紐付ける役割をしています。他にもいくつかの補助的なファイルがあり、それらをまとめて Shape ファイルとして扱います。基本的にはファイル名で紐付けられるので、これらのファイル群のファイル名を変えることは、思わぬ問題の原因となるため注意が必要です。また、Shape ファイルは地理情報データの交換用フォーマットとしては事実上の標準といえますが、かなり古い形式でもあることからさまざまな点で注意が必要です。場合によっては無理に Shape にせず、使うソフトウェアの標準形式や、PostGIS などの地理情報を正しく扱えるデータベースを用いてデータ交換をするのがよいでしょう。

数値精度

地球の周囲長は約 4 万 km です。メートルに直すとおよそ 40000000m となり 10 進数で 8 桁のオーダーです。正確な位置情報の処理を行うためには、小数点以下数桁までの数値計算の精度が保証されてないと難しいでしょう。現在一般的なコンピュータの単精度浮動小数点数は、IEEE 754 で規定される 23 bit の仮数部をもつ binary32 という形式です。

[*9] PostGIS のページ：`https://postgis.net/`

[*10] MySQL の地理情報データ型のドキュメント：`https://dev.mysql.com/doc/refman/5.7/en/spatial-types.html`

仮数部が 23 bit なので、10 進数でおよそ 7 桁の精度しか表すことができず、1m 単位の精度は持てません。これでは、計算のたびに誤差が蓄積してしまいます。地理座標の計算では、原則として倍精度浮動小数点数を用いるようにしたほうがよいです。一方で、一般的なゲームエンジンの座標値は単精度で表現されていたりもするので、データ処理時は倍精度、最終データでは単精度など、工夫をしなければなりません。また、データ処理時にも、倍精度を使っていても、計算誤差を減らすための注意はしましょう。

地理空間インデックス

大量の地理情報データの処理には、マシンパワーが必要になります。データベース処理では、適切に構築されたインデックスを用いて高速化することが一般的ですが、地理情報処理でも同様に、空間インデックスを使います。四分木や、R 木などを用いて、空間内での位置や範囲の検索を高速化します。

6.4　QGIS のインストール

さて、座学はここまでです。早速 OSS の GIS ソフトウェアである QGIS をインストールして、データを処理していきましょう。QGIS のサイト[*11]から、最新版のスタンドアローンインストーラーをダウンロードします。自分の使っている OS に合わせて選んでください。本稿執筆時点での最新が 3.0.0 なので、本稿では「Windows 版 QGIS スタンドアローンインストーラ Version 3.0（64bit)」を用います。

▲図 6.1　QGIS のダウンロード

ダウンロードしたら実行して、インストーラの指示にしたがってインストールしてください。なお、このとき指定するインストール先フォルダには多バイト文字やスペースを入れないようにしてください。後のデータ処理でうまくいかず、余計なトラブルシュートをする羽目になります。

[*11] QGIS のページ：https://www.qgis.org/

6.5　データのダウンロード

次に GIS データをダウンロードしましょう。今回使用するのは次の 2 つです。

- 国土地理院の基盤地図情報
- NTT データと RESTEC 提供の AW3D30

基盤地図情報

基盤地図情報のサイトを表示し[*12]、その中の基盤地図情報のダウンロードをクリックします。

▲図 6.2　基盤地図情報トップページ

データのダウンロードにはユーザー登録が必要なので、新規登録からユーザー登録をします。そして、基盤地図情報の基本項目のファイル選択をクリックします。

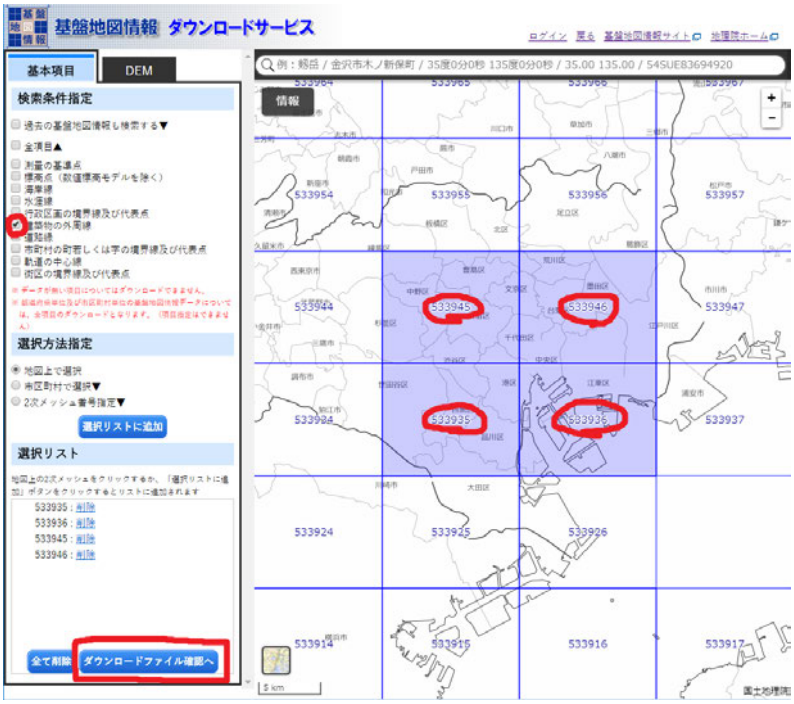
[*12] 基盤地図情報サイト：`http://www.gsi.go.jp/kiban/index.html`

▲図 6.3　基本項目ファイル選択

日本地図が表示されました。ここから必要な項目と地域を選択します。今回は「建築物の外周線」のみにチェックを入れます。

また、地域は自分の馴染みのある地域の方が作業がしやすいですが、適当に選びましょう。例として、港区を含む東京の 533945,533946,533935,533936 の 4 区画を選びます。

「ダウンロードファイル確認へ」をクリックします。



▲図 6.4　基本項目データ選択

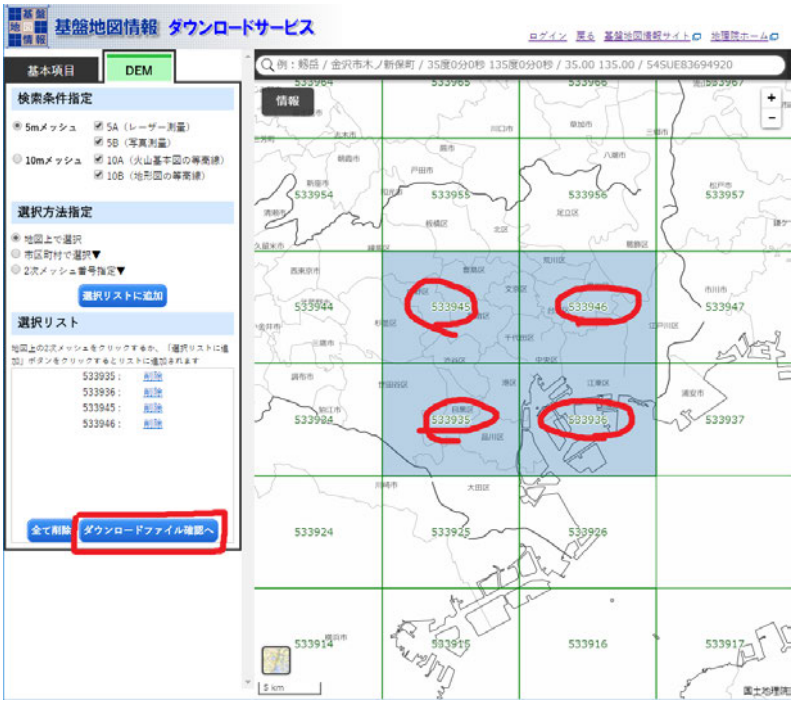
「全てにチェック」して、「まとめてダウンロード」します。

▲図 6.5　基本項目ダウンロード確認ページ

これで、選択したデータの格納された Zip ファイルがダウンロードされます。
ダウンロードのページまで戻り、同様に数値標高モデルをダウンロードします。

▲図 6.6　数値標高モデル選択

こちらも同じ地域で 5m メッシュを選択してください。同様に地図で選択して、ダウンロードします。



▲図 6.7　数値標高モデルデータ選択

▲図 6.8　数値標高モデルダウンロード確認ページ

最後に、もう一度ダウンロードのページに戻り、「符号化規則、ファイル仕様書、表示ソフトウェア等」のリンクから、

▲図 6.9　表示ソフトウェア等リンク

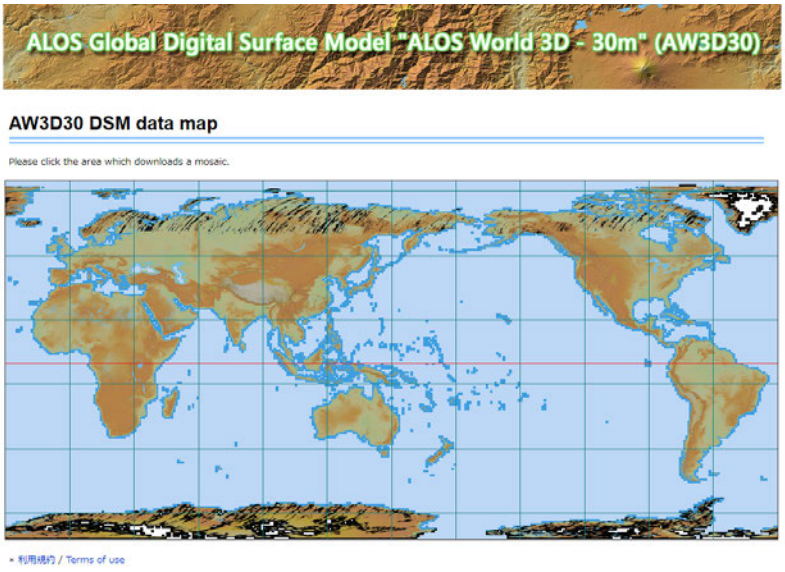
「基盤地図情報ビューア」をダウンロードします。

▲図 6.10　表示ソフトウェアダウンロードリンク

AW3D30

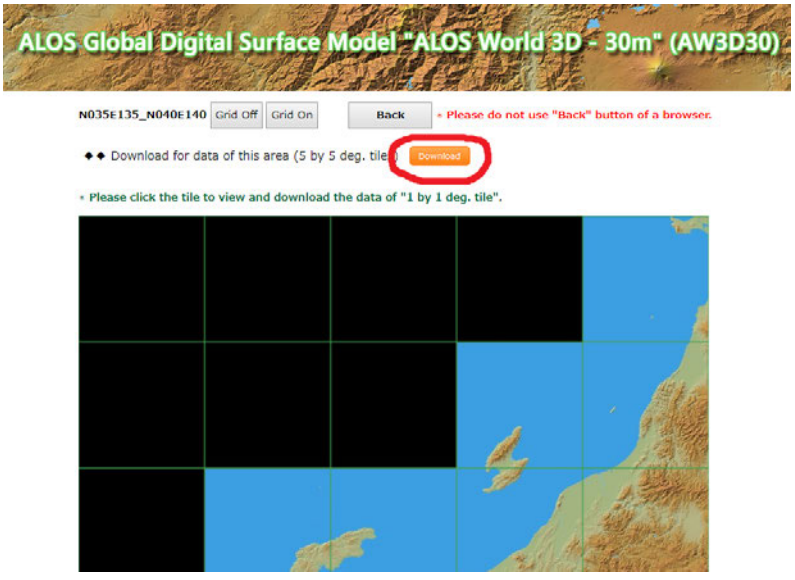
AW3D30 のサイトを表示します[*13]。こちらでもユーザー登録が必要です。上記ページの「4. Download」の項目から、ユーザー登録とダウンロード用のサイトへのリンクがあります。世界地図から何度か目的の地域にドリルダウンします。

[*13] AW3D30 のサイト：`http://www.eorc.jaxa.jp/ALOS/en/aw3d30/`



▲図 6.11　世界地図表示画面

地図から必要な地域を選び「Download」します。



▲図 6.12　AW3D30 ダウンロード画面

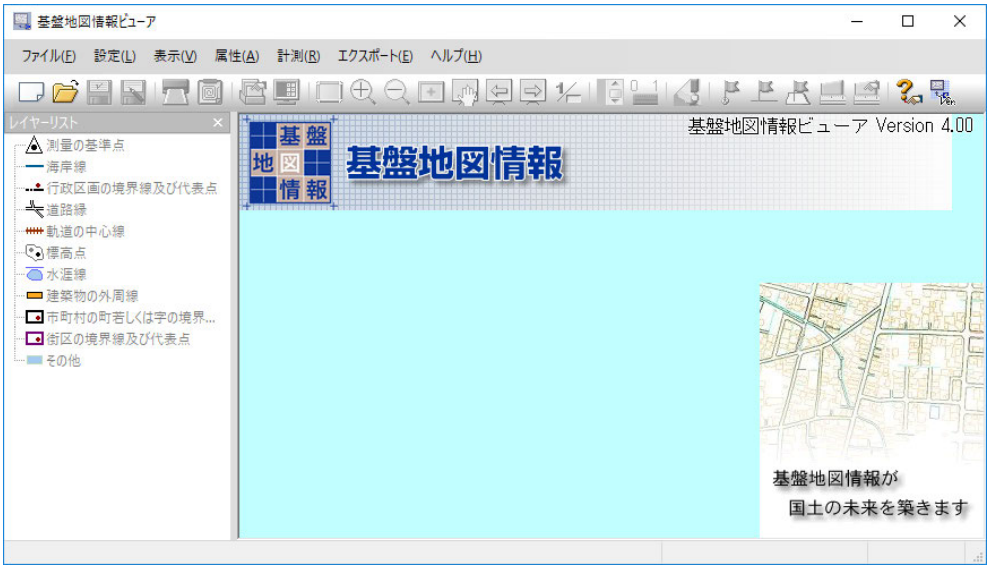
5 × 5 の領域を一括でダウンロードすることもできるので活用します。今回の例として、N035E139 の名前のついているデータを使用しますので、例にしたがって試してみる

方はこのデータをダウンロードしてください。なお注意点として、ここでは FTP でダウンロードするため、ファイヤーウォールなどの設定を見直す必要があるかもしれません。

ダウンロードしたファイルは tar.gz 形式の圧縮ファイルで、GeoTiff 形式の DSM が収められています。DSM とは Digital Surface Model の略で、建物や樹木などの地物を含めた地表面の高さのデータです。

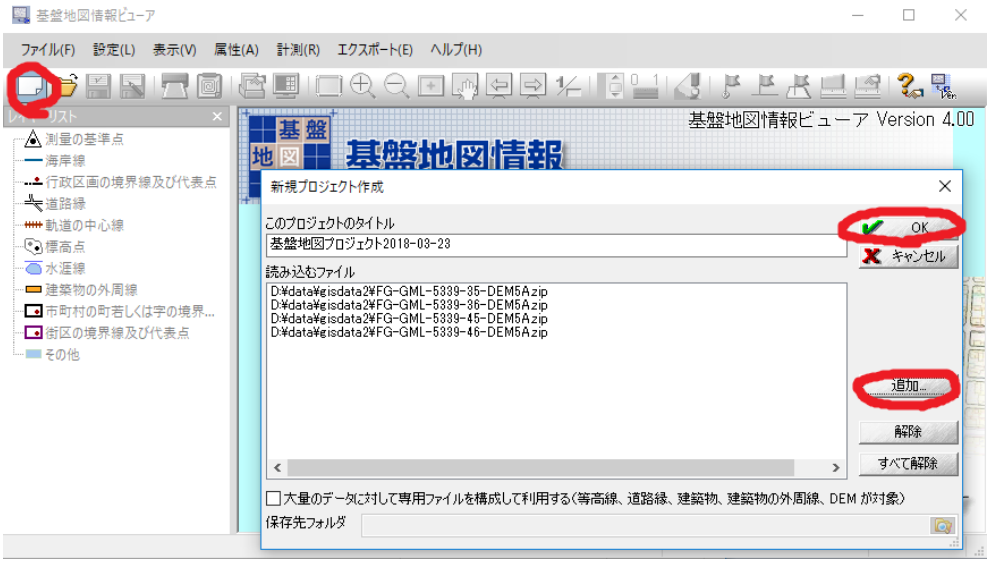
6.6 データの変換

基盤地図ビューア FGDV によるデータの確認と Shape ファイルへの変換をします。ダウンロードした基盤地図情報ビューアを展開すると、FGDV.exe という実行ファイルが得られます。これを実行して図 6.13 のようなソフトウェアを起動します。



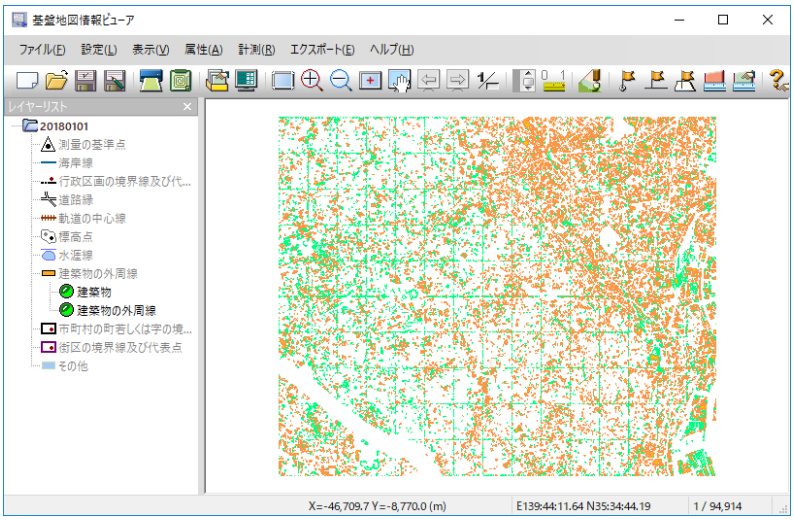
▲図 6.13　基盤地図情報ビューア

次に、左上の白い四角のアイコンまたは「ファイル」メニューから、「新規プロジェクト作成」を選びます。新規プロジェクト作成ダイアログが出てくるので、「追加」で先ほどダウンロードしたファイルを選択します。基盤地図情報ビューアは、Zip ファイルのままで読み込めるので、Zip ファイルを指定します。

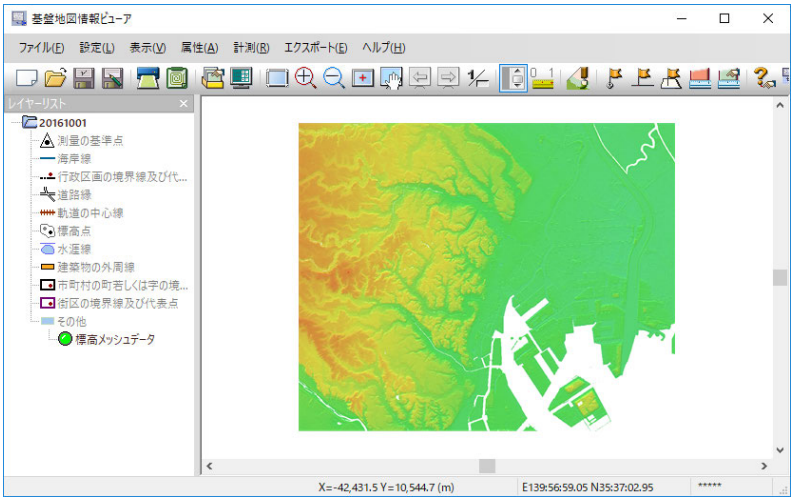


▲図 6.14　新規プロジェクト作成

それなりに時間がかかりますが、うまくいけば、次のような画面が表示されます。

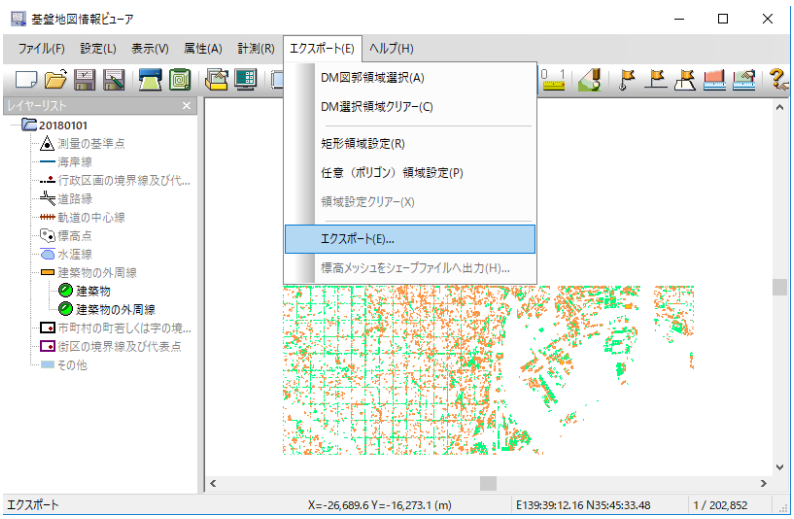


▲図 6.15　建築物の外周線データの表示



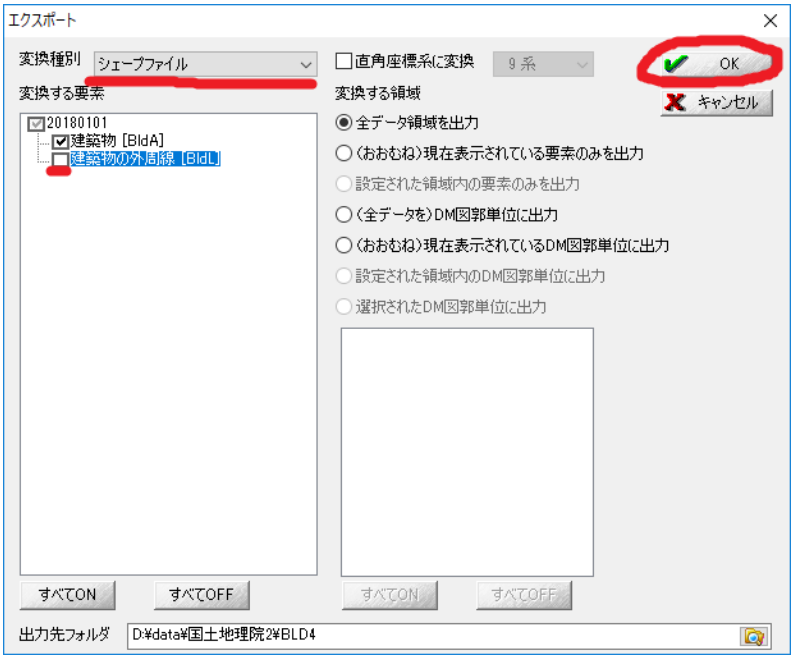
▲図 6.16　数値標高モデルデータの表示

次に、データを Shape ファイルにエクスポートします。
建築物の外周線のデータは「エクスポート」メニューから「エクスポート」を選びます。



▲図 6.17　建築物の外周線データをエクスポート

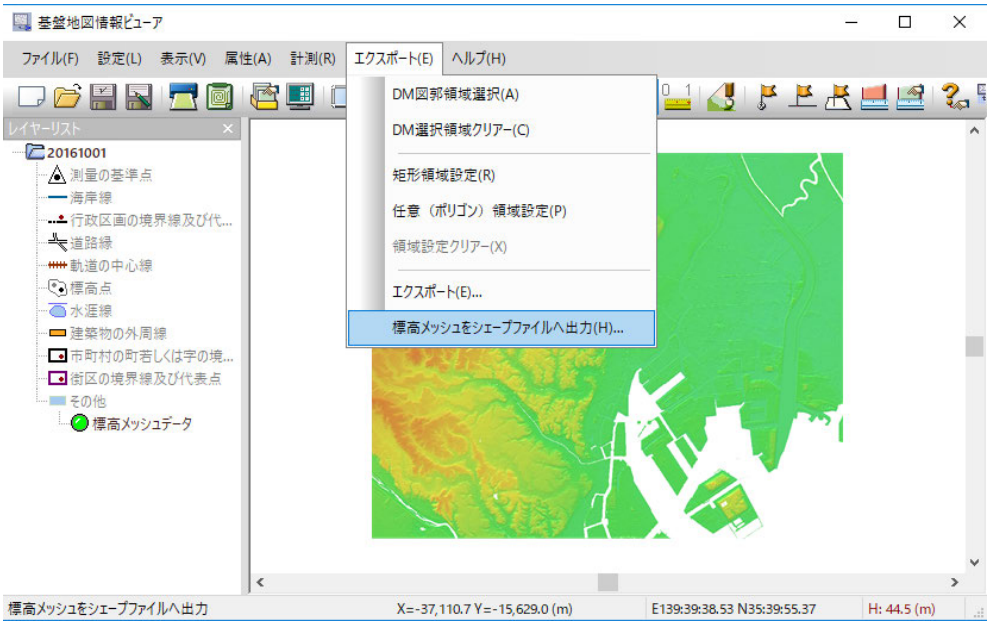
出てきたダイアログの「変換種別」を「シェープファイル」に、「建築物の外周線」のチェックを外し、出力先フォルダを適切に設定して「OK」を押します。



▲図 6.18　データエクスポートダイアログ

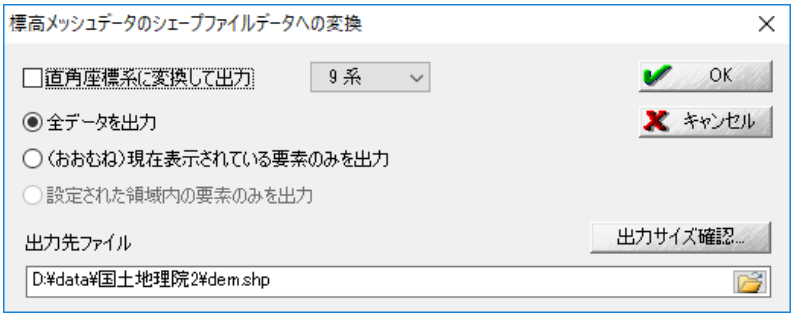
特に大都市の中心部では建物の数が多く、「Out of Memory」などのエラーが起こる可能性がありますので、適切に一区画ごとに読み込んでエクスポートするなどしてください。

数値標高モデルは「エクスポート」メニューから「標高メッシュをシェープファイルへ出力」を選びます。



▲図 6.19　標高メッシュをエクスポート

出てきたダイアログの出力先ファイルだけ適切に調整して「OK」を押します。なお、ファイルのパスにはマルチバイト文字やスペースを含まない方が余計な問題が起きなくて楽です。



▲図 6.20　標高メッシュエクスポートダイアログ

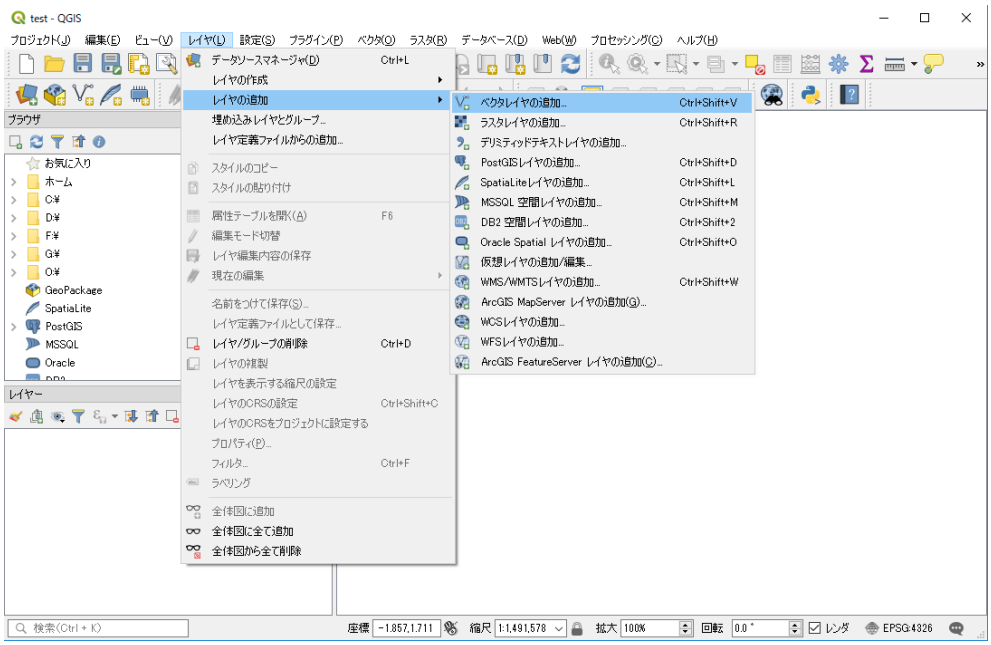
もし「Out of Memory」のエラーが発生したら、一区画ごとにエクスポートしてください。

以上の手順で、建築物の外周線の Shape ファイルと数値標高メッシュの Shape ファイルが作成できました。

6.7　データの読み込みと QGIS の基本操作

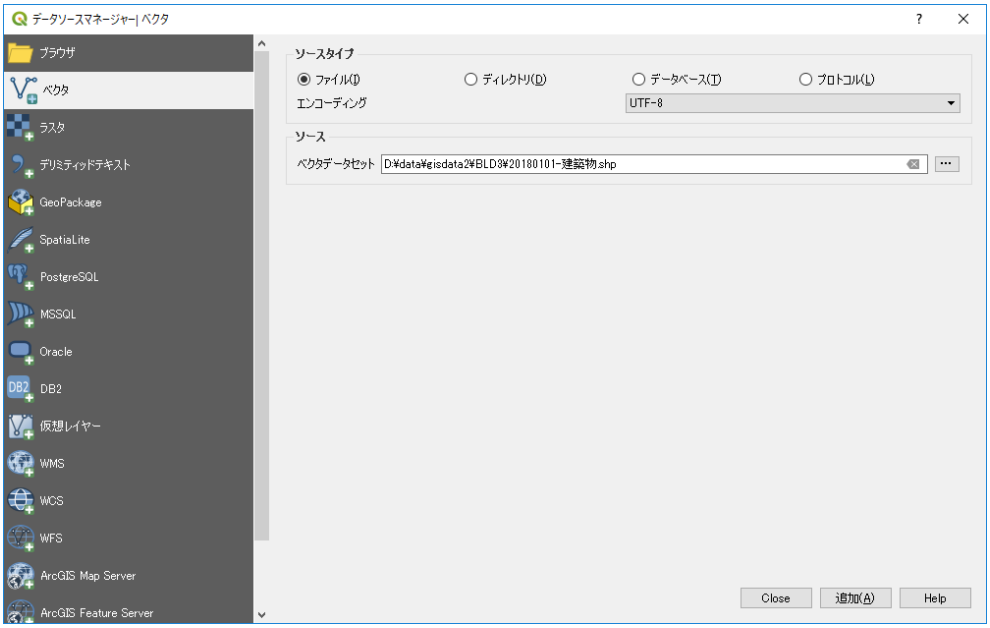
「QGIS Desktop 3.0.0 with GRASS 7.4.0」を起動してください。起動したらまず、プロジェクトを作ります。「プロジェクト」-「新規作成」を選んでください。空のプロジェクトが作成されるので、メニューの「プロジェクト」-「保存」によりファイル名等を指定して保存します。

次に、「レイヤ」-「レイヤの追加」-「ベクタレイヤの追加」を選択します。



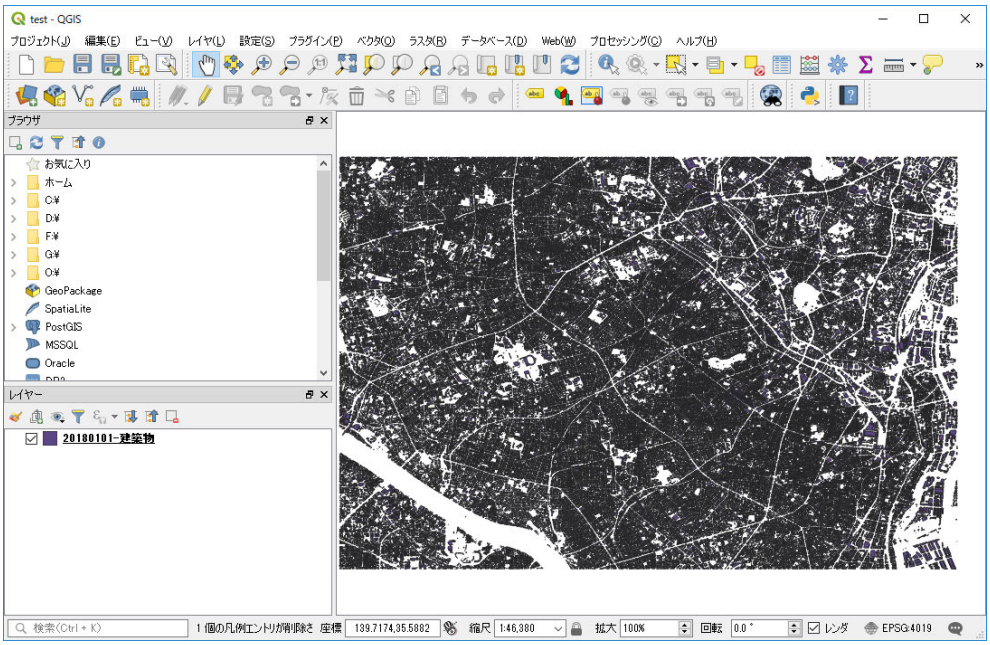
▲図 6.21　ベクタレイヤの追加

表示されるダイアログで、先ほど基盤地図情報ビューアでエクスポートした建築物の外周線データの Shape ファイルを指定して「追加」を押します。



▲図 6.22　ベクタレイヤの追加ダイアログ

図 6.23 のように読み込まれます。

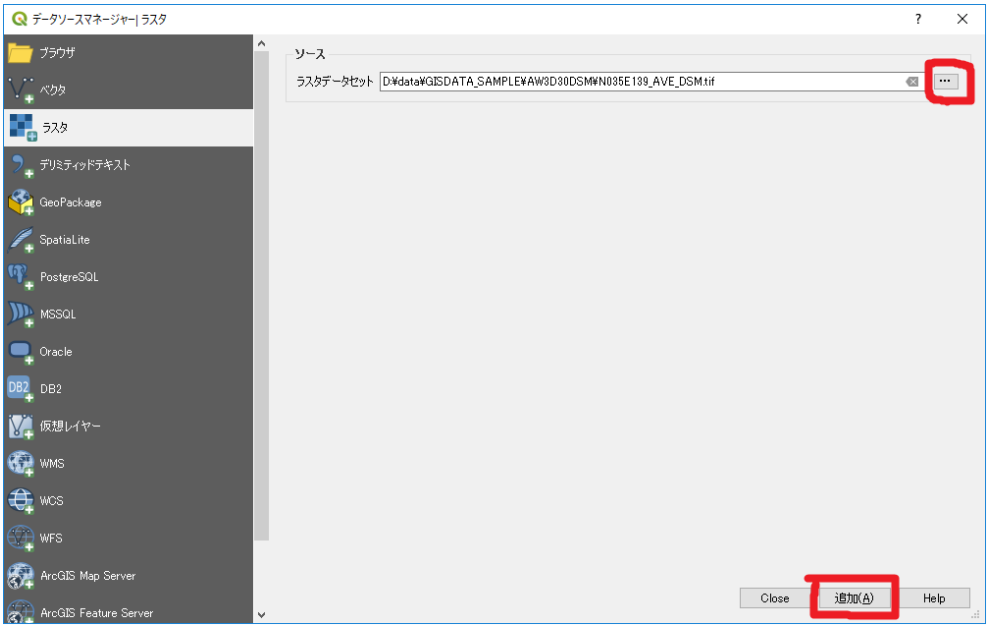


▲図 6.23　ベクタレイヤが読み込まれたところ

こうして建築物のデータを読み込みました。同様の手順で数値標高のデータも読み込みます。数値標高のデータは 5m 間隔の格子状に配置された点のデータになるので、かなり描画の量が増え、重くなるので気を付けてください。

次に、「レイヤ」-「レイヤの追加」-「ラスタレイヤの追加」を選択します。

表示されるダイアログで、ファイル名を設定して「追加」を押して、AW3D30 のラスタデータを読み込みます。



▲図 6.24　ラスタレイヤの追加ダイアログ

最後に、QGIS の基本操作ですが、手のアイコンはドラッグで地図の移動になります。スペースキーを押しながらドラッグすることで、地図の移動をすることもできます。また、マウスのホイールまたは「＋」「－」の虫眼鏡アイコンで拡大縮小ができます。

6.8　データの処理　その 1

一般的な GIS ソフトウェアには、単に地理情報データを表示するツールだけでなく、さまざまなアルゴリズムで地理情報データを処理したり分析するツール群も含まれています。ここからはこれらのツール群を駆使して、用意したデータから建物の高さを計算して 3D にすることをやってみましょう。

最初に、ここで行う操作の簡単な解説をします。今回用意したデータは、測量等で作成された基盤地図情報の建築物の外周線のポリゴンデータ、レーザー測量や写真測量で作成された基盤地図情報の数値標高モデル（以降 DEM）、衛星画像から作成された AW3D30

のデータ（以降 DSM）の 3 点です。DEM は、地物を除いた「地表面」の高さデータ、DSM は、地物込みの表面の高さデータなので、その引き算で、地物の高さを求めることができます。これを利用して、高さを持たない建築物の外周線のポリゴンに高さ情報を付加します。DEM は、5m メッシュ、DSM は 30m メッシュなので、それなりに誤差が生じますが、「当たらずも遠からず」のレベルでのリアルさを目指します。

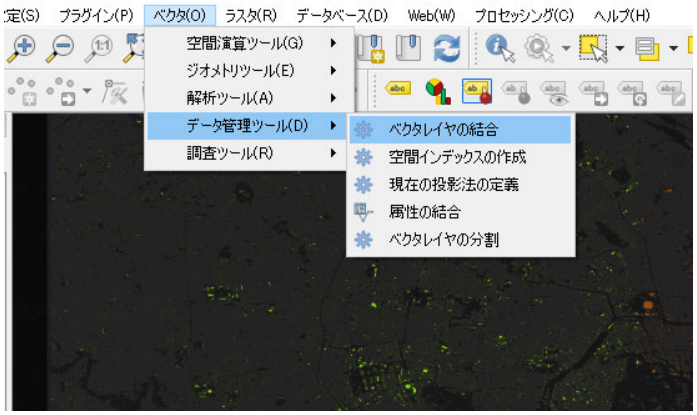
準備として、複数に分かれているデータを結合します。そして最初に、ベクタデータの DEM とラスタデータの DSM を処理して高さデータを作成しましょう。

今回の処理のその他の誤差要因

さらに細かい話ですが、DSM データは ALOS という人工衛星で撮影した画像をもとに、ステレオ視の原理を用いて複数の画像から高さを測定しています。対して、DEM データは、航空機からのレーザー測量で標高データを測定しています。また、仕様書を確認すると DSM データは EGM96 というジオイドモデルを使用しています。DEM 側には国土地理院の測地成果であることから、水準測量での標高値および日本のジオイドモデルを使用していると思われます。このため、このような計測方法や基にするジオイドモデルの違いでも誤差が生じる可能性があります。

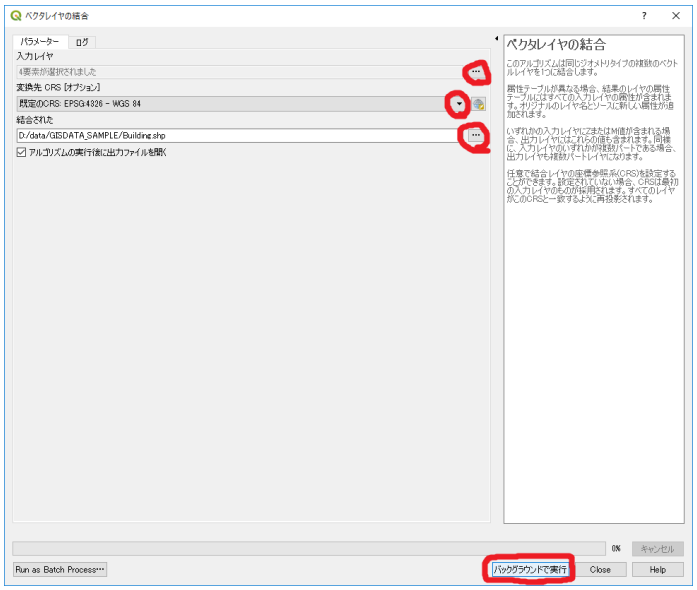
建築物の外周線データの結合

基盤地図ビューア FGDV によるエクスポート時に、4 つに分けて書き出した Shape ファイルを結合します。メニューから、「ベクタ」-「データ管理ツール」-「ベクタレイヤの結合」を選択します。



▲図 6.25　ベクタレイヤの結合

ベクタレイヤの結合ダイアログでは、入力レイヤに結合したい建築物の外周線データのレイヤをチェックを付けます。変換先 CRS は、測地系です。これは、EPSG:4326 を選んでおきましょう。結合されたデータの保存先には、適当な名前の Shape ファイルを指定します。そして、「バックグラウンドで実行」を押します。



▲図 6.26　ベクタレイヤの結合ダイアログ

処理が終わると新規レイヤとしてプロジェクトに追加されます。他の建築物の外周線データのレイヤは今後必要ないので、「レイヤ」ペインで右クリックして削除しておきま

しょう。

もし、DEM データも分割して書き出していた場合、同様の手順で結合しておいてください。

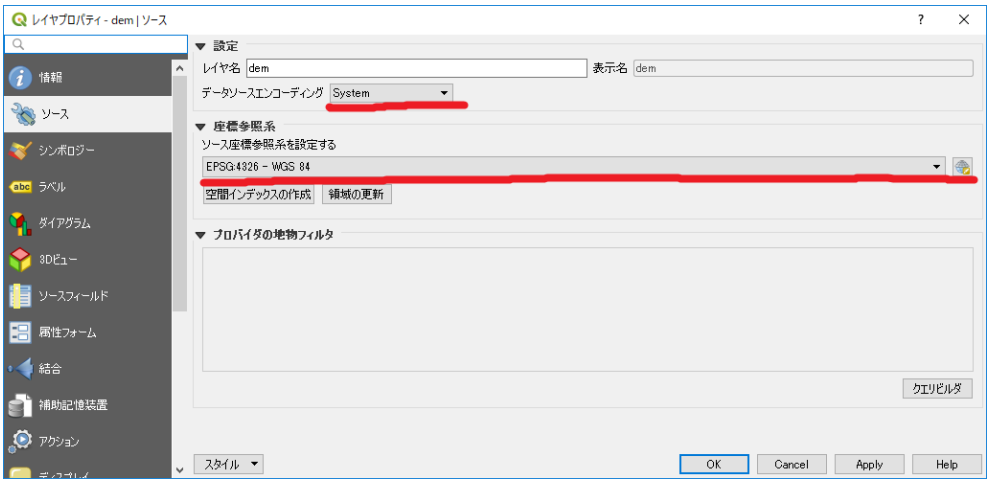
ポイントデータにラスタからの値を取り込む

DEM と DSM は、ベクタとラスタで種類が異なるので、そのままでは演算できません。そこで、どちらかに合わせることになりますが、ここではより詳細な DEM の方に、DSM のデータを合わせましょう。

まず、ラスタの値を読み込むためのカラムを作成します。DEM のレイヤをダブルクリックしてレイヤプロパティのダイアログを開きます。

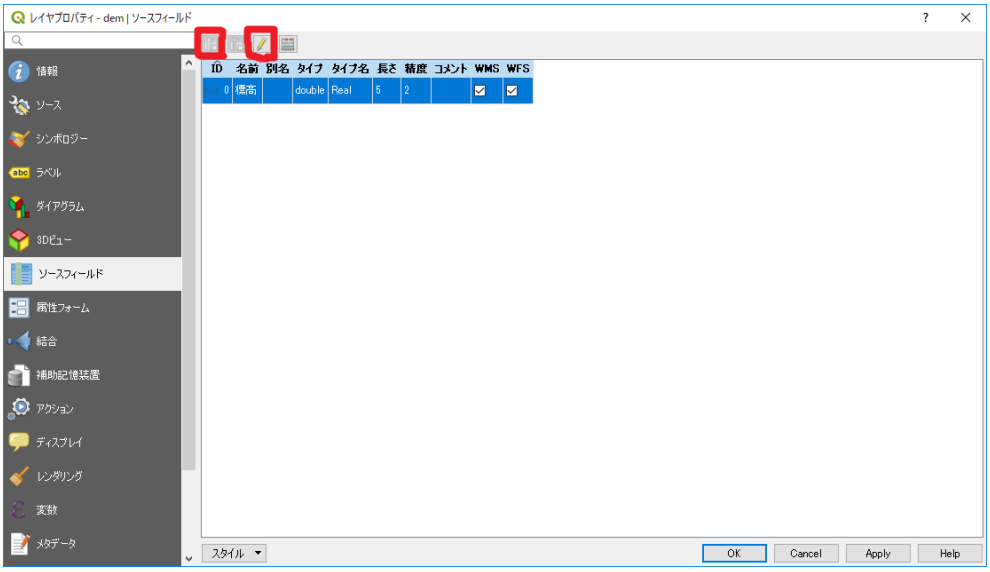
せっかくなので、ここで測地系の設定も行っておきましょう。左側のペインから「ソース」を選んでください。表示された画面で、座標参照系を設定します。ここでは、EPSG:4326 を設定しましょう。これは、WGS84 を意味します。

また、データソースエンコーディングで文字列のエンコーディングを選べます。今回は System にするとよいでしょう。もし、カラム名などが文字化けしているときはここの設定を見直してみます。



▲図 6.27　レイヤプロパティ・ソース

次に左側のペインから「ソースフィールド」を選んでください。上に並ぶアイコンの鉛筆アイコンをクリックして編集モードにします。



▲図 6.28　レイヤプロパティ・ソースフィールド

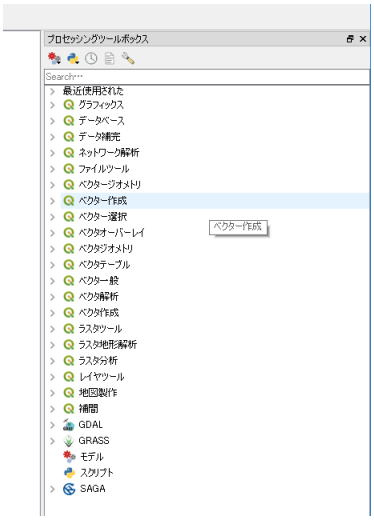
編集モードにしないとカラムの変更やジオメトリの修正といったデータ編集ができないようになっています。編集モードになると「新規フィールド」ボタンが押下可能になるのでクリックし、フィールド追加ダイアログで必要な項目を入力し、「OK」ボタンを押します。長さ・精度は、浮動小数点のデータの桁数設定となります。今回は高さの値なのでそれに合わせて十分な長さを確保しておきます。

▲図 6.29　フィールドの追加ダイアログ

ついでに、DEM データの「標高」フィールドがマルチバイト文字のフィールド名なので、「dem」などと変えておいてもよいかもしれません。

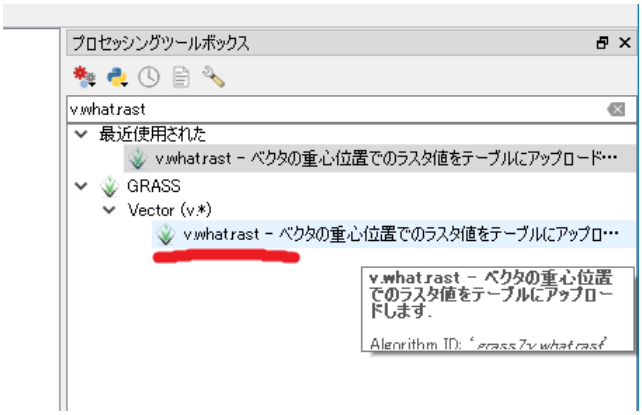
最後に、再び鉛筆アイコンをクリックしてデータの変更を保存して「OK」ボタンを押してダイアログを閉じます。

さて、準備が長くなりましたが、ラスタの値を読み込みます。「プロセッシング」メニューの「ツールボックス」を選択します。右側に新しいペインが表示されました。



▲図 6.30　ツールボックス

「Search」と表示されているフィールドに「v.what.rast」と入れてください。「ベクタの重心位置でのラスタ値をテーブルにアップロードします.」という項目が見つかりますので、これを選択します。



▲図 6.31　ベクタの重心位置でのラスタ値をテーブルにアップロードする

これは、ベクタデータの属性カラムにその座標でのラスタの値を取り込むツールです。ダブルクリックして実行してみましょう。

開いたダイアログで必要項目を設定していきます。「Name of vector points map for

which to edit attributes」には、ラスタからデータを読み込むレイヤを選択します。今回の場合は、DEM のレイヤを選択します。

「Raster map to be sampled」には、読み込むラスタのレイヤを指定します。「Input feature type」では、今回の DEM データは点の集合なので、「point」を指定します。「Name of attribute column to be updated with the query result」には、先ほど新規で作成したフィールドの名前を設定します。このフィールドにラスタからの値が読み込まれます。「v.out.ogr 出力タイプ」は auto でもよいですが、一応明示的に point を指定しておきます。「Sampled」は出力結果をどうするかの指定です。入力フィールド横の「…」アイコンで「Save to File...」を選んで、適切なファイル名を設定します。

すべての設定が終わったら「実行」ボタンをクリックします。処理が終わるまでにかなりの時間がかかるので、ゆっくり待ちましょう。

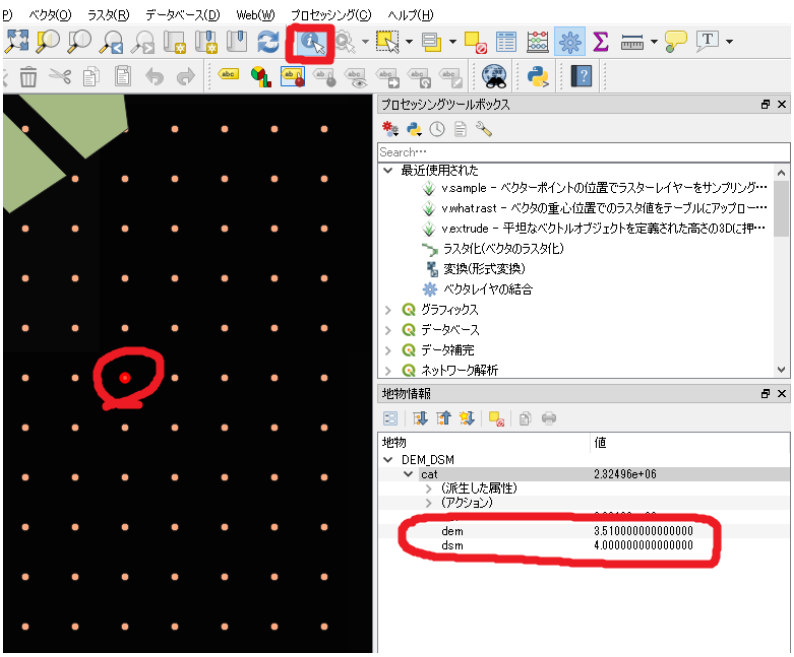
なお、処理画面でコマンドラインらしき表示がたくさん出てきます。これは、GRASS という OSS の GIS ソフトウェアを呼び出しています。

GRASS は、定評のある老舗の機能豊富な GIS ソフトウェアですが、コマンドライン中心のインターフェースから難易度が高くなっています。QGIS は、それ自体でも多くの GIS 処理用の機能を持っていますが、このようにフロントエンドとして GRASS と連携することで、より多くの機能をより簡単に使えるように構成されています。

実行が終了すると、作成されたベクタレイヤが追加されます。レイヤプロパティから空間インデックスを作成しておくと、今後の作業が早くなるかもしれません。

最後に実行結果を確認して次の作業に移りましょう。i の文字を矢印が指しているアイコンをクリックします。これは、地物の情報表示機能です。

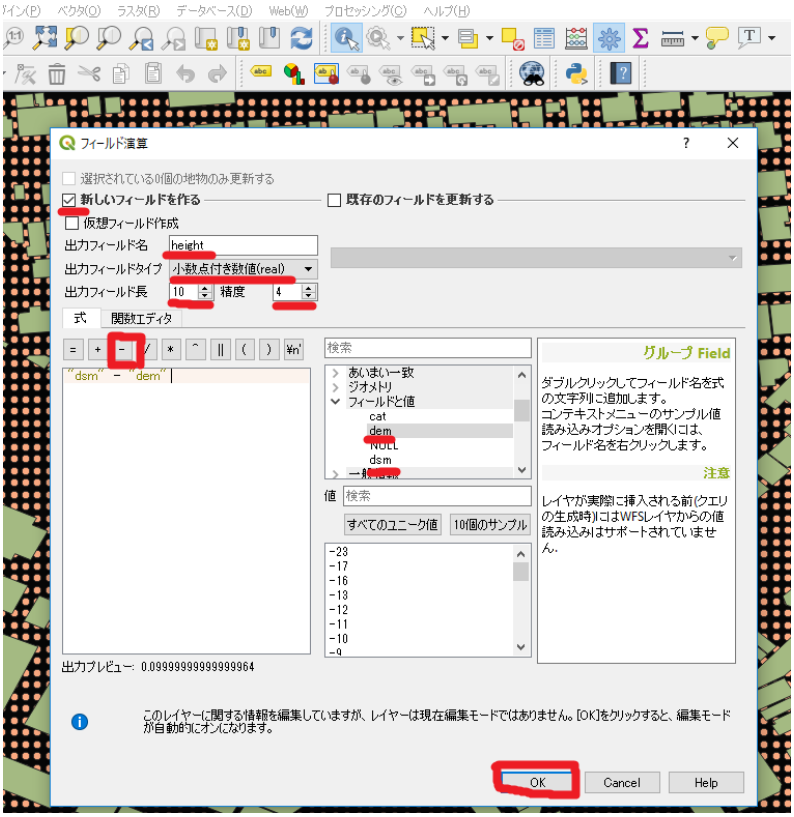
DSM の値を取り込んだ結果のベクタレイヤを選択し、地図上の任意のポイントデータをクリックします。すると、右側に地物情報のペインが開き、もともと持っていた dem のデータ値と取り込んだ dsm ラスタデータの値が格納されているのが分かると思います。



▲図 6.32　地物情報表示

フィールド計算機を使う

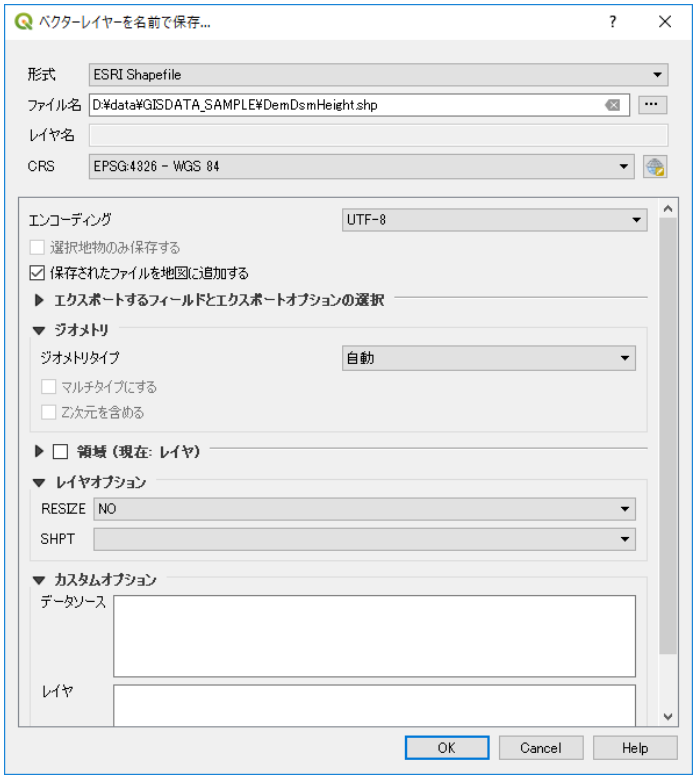
DSM の高さの値を属性に取り込んだので、元々もっていた DEM の高さと引き算をしてみましょう。この操作にはフィールド計算機を使います。アイコンバーの中のそろばんのアイコンをクリックします。でてきたダイアログで、DSM-DEM の計算を実行します。「新しいフィールドを作る」にチェックを入れ、出力フィールド名に「height」出力フィールドタイプを「小数点付き数値」にします。出力フィールド長は、10、精度を 4 くらいにしておきましょう。「フィールドと値」の dsm をダブルクリックし、式テキストフォームに入れます。演算子ボタンの「-」をクリックし、dem をダブルクリックして式「"dem" - "dsm"」を作ります。最後に、「OK」ボタンを押して実行します。これで、DSM-DEM で計算した高さ情報の演算結果が新規フィールドに保存されました。



▲図 6.33　フィールド計算機を使う

地物の情報表示機能を使って、計算した結果が入っているか確認してみましょう。

最後に、フィールド計算機の操作は編集操作となるので、編集モードに自動的になっています。メニューの下のアイコンの 2 行目の鉛筆アイコンをクリックして編集モードを保存して終わりましょう。と言いたいところなのですが、なぜか保存処理がとても時間がかかるようなので、編集モードを解除せずに、レイヤペインで右クリックして、「Save as...」を選択し、図 6.34 のように形式を Shape ファイル、ファイル名に適切なもの指定して「OK」ボタンを押して保存してください。



▲図 6.34　フィールド計算機結果をファイルに保存

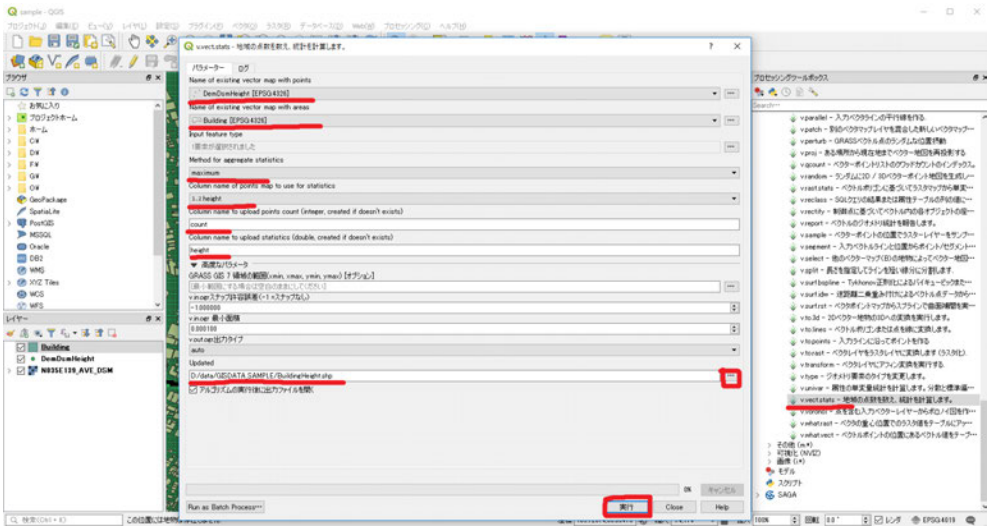
元の DEM ファイルは、編集モード終了時に、保存しない「Discard」の方を選び、レイヤリストから削除しておきましょう。

ポリゴンデータにポイントデータの値を取り込む

ここまでで、DSM と DEM から計算した地物の高さの値が各ポイントに格納されました。次に、建物外周線ポリゴンのデータに、地物の高さの値のポリゴン内での最大値を取り込んでみましょう。

ツールボックスペインで、「v.vect.stats」を検索します。見つかったらダブルクリックしてツールを起動します。設定する項目は、「Name of existing vector map with points」に、地物の高さまで計算した DEM レイヤ、「Name of existing vector map with areas」に、建物の外周線データのレイヤを、「Method for aggregate statistics」には、領域内に複数の点が含まれる際に集約する時の計算を指定しますが、maximum を設定します。「Column name of points map to use for statistics」には、地物の高さが入っているフィールドの名前を指定します。「Column name to upload points count」には、領域に含まれる点数を格納するフィールド名、「Column name to upload statistics」には、集約

した結果の値（この場合は、領域に含まれる点の height の最大値）を格納するフィールド名を指定します。最後に、「Updated」の入力フィールドに、「...」ボタンから、「Save to File...」を選んで、保存するファイル名を指定し、「実行」ボタンを押してください。



▲図 6.35　v.vect.stats の実行

この処理も非常に時間がかかりますが、終わったら地物の情報表示機能で正しく集計が行われたか確認してみましょう。

6.9　データの処理　その 2

ここまでの作業で、もともと高さデータを持っていなかった建築物の外周線データに、DSM と DEM から計算した高さのデータを持たせることができました。ここから、Unity に取り込む方法を考えてみましょう。

そもそも、ベクタデータは仕組みとしてはゲームエンジンで描画のために扱うジオメトリの仕組みと似たようなものです。そのため、原理上はベクタデータの座標を何らかの形で Unity 上の座標に変換し、点同士の接続データなどを追加して読み込めるような形式にコンバートするとよいように思われます。

しかし、たとえば QGIS で出力可能なベクタデータ形式として DXF に出力し、試しに CAD ソフトやモデリングソフトに読み込んでみると、本稿で例として扱っているデータなどはそのままではデータ量が多すぎるため操作できない状態になります。扱えるようにするには、より小さい領域で分割するなどの方法が必要そうです。

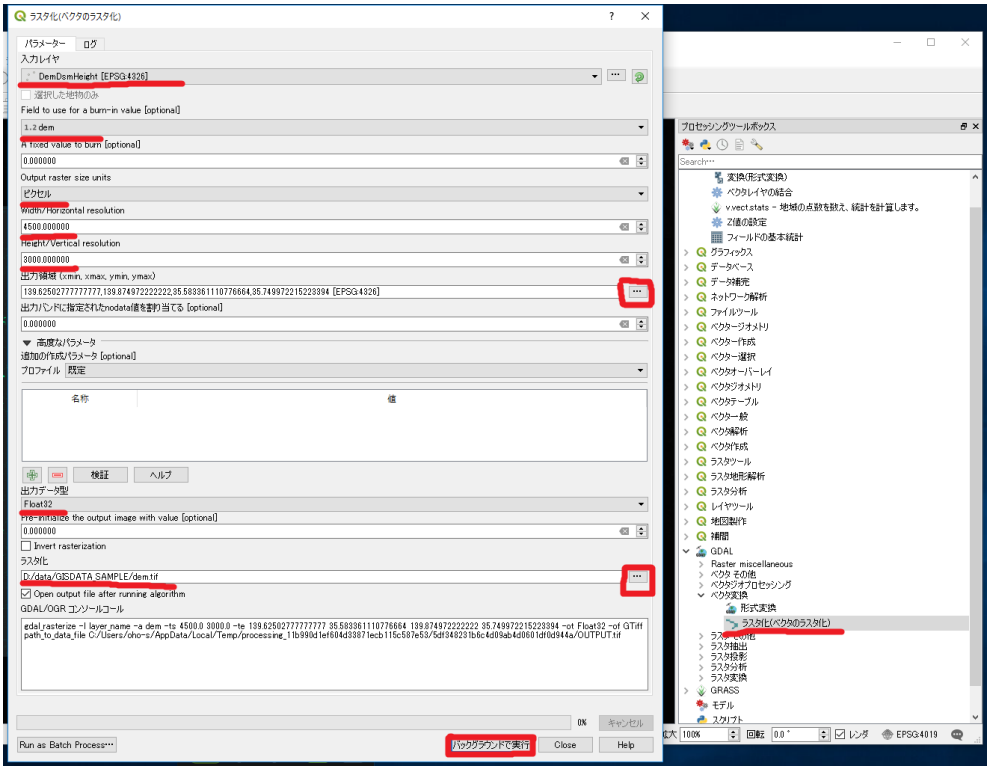
そこで、本稿では Unity 上で目で見える形にするために、ベクタデータを一度ラスタデータに変換し画像データとしたのちに、その画像をテクスチャとして取り込み、Unity

の頂点シェーダにハイトマップとして扱わせることにします。試してみたところこの方法であれば分割せずに表示することができました。

Unity に取り込むためにエクスポートする

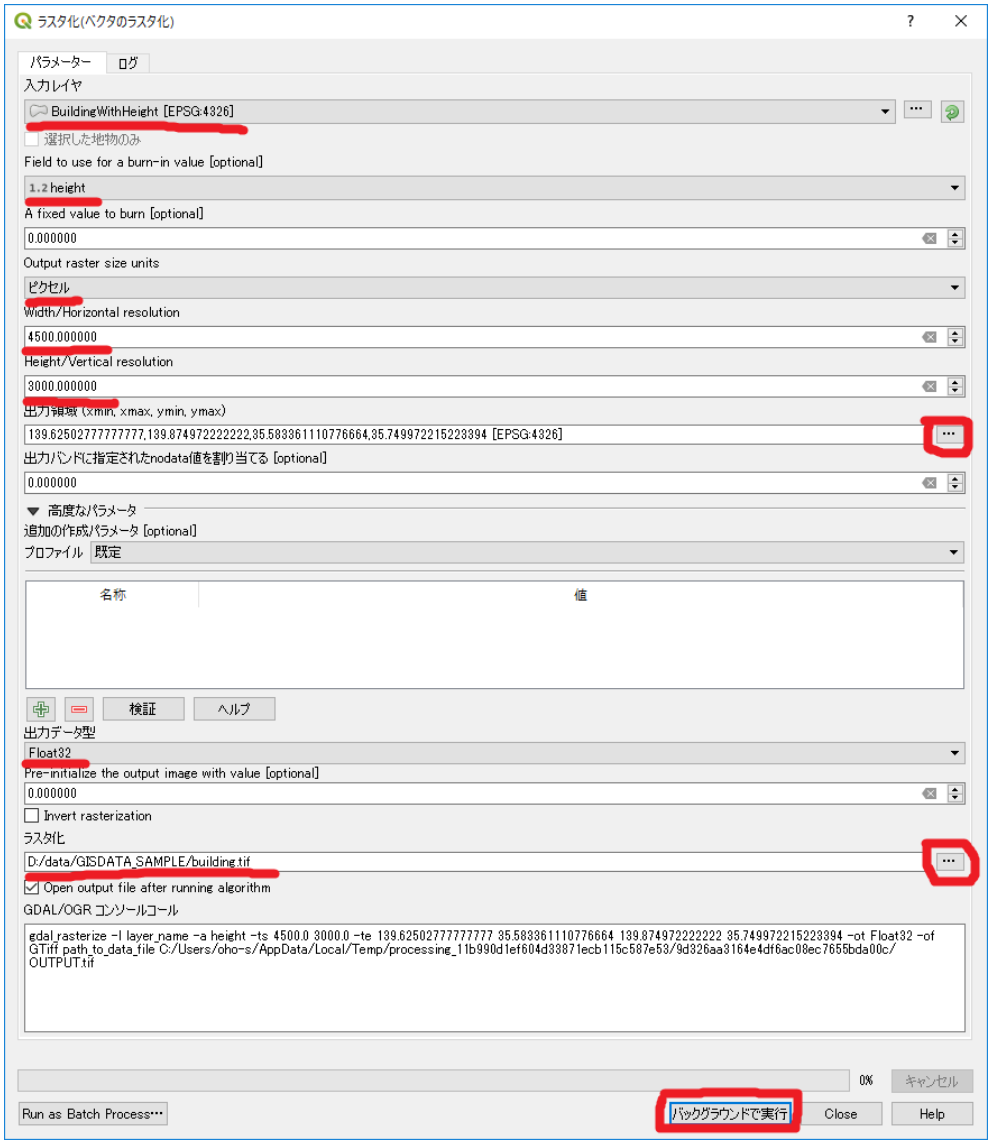
DEM のベクタデータをラスタ化します。ラスタ化する際には縦横のピクセル数を決めなければいけません。国土地理院の基盤地図情報の仕様書を見ると、1 メッシュが 225 × 150 のデータ点で構成され、10 × 10 のファイルがひとつの Zip ファイルにまとめられています。今回の例ではそれを 2 × 2 並べているので、縦横のデータ点の数は、4500 × 3000 となります。

ツールボックスの「ラスタ化」を用いて DEM レイヤを GeoTIFF に変換します。「ラスタ化（ベクタのラスタ化)」をツールボックスから探しダブルクリックして実行します。パラメータとして「入力レイヤ」には DEM のベクタレイヤを、画素値とするフィールドを指定する「Field to use for a burn-in value」には標高値のフィールド名を、出力するラスタサイズの単位を指定する「Output raster size units」には、ピクセルを設定します。「Width」「Height」には先ほど計算した 4500 と 3000 を設定します。「出力領域」には「…」をクリックして「レイヤー/キャンパス範囲を利用する」を選び DEM レイヤを選択します。「出力データ型」は Float32 を設定し「ラスタ化」には「…」から保存するファイル名を設定します。そして「バックグラウンドで実行」をします。



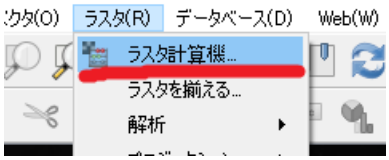
▲図 6.36　DEM のラスタ化

同様に建築物の外周線データレイヤもラスタ化します。「ラスタ化（ベクタのラスタ化）」を実行します。パラメータとして「入力レイヤ」には建築物の外周線データのベクタレイヤを、画素値とするフィールドを指定する「Field to use for a burn-in value」には建築物の高さのフィールド名を、出力するラスタサイズの単位を指定する「Output raster size units」には、ピクセルを設定します。「Width」「Height」には 4500 と 3000 を設定します。「出力領域」には「…」をクリックして「レイヤー/キャンパス範囲を利用する」を選び、先ほどラスタ化した DEM レイヤと合わせるために DEM レイヤを選択します。「出力データ型」は Float32 を設定し「ラスタ化」には「…」から保存するファイル名を設定します。そして「バックグラウンドで実行」をします。



▲図 6.37　建築物の外周線データのラスタ化

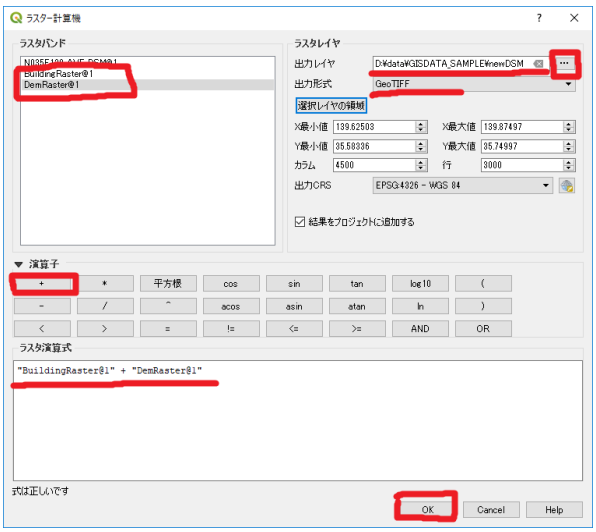
次に、ふたつのラスタレイヤの加算を行います。メニューの「ラスタ」-「ラスタ計算機」を選択します。



▲図 6.38　メニュー：ラスタ計算機

ラスタ演算式に行いたい演算の式を作成します。Dem のラスタレイヤ名を「ラスタバンド」フィールドで選びダブルクリックすると演算式に追加されます。演算子から「＋」を選びクリックして演算式に追加し、建築物の外周線データのラスタレイヤ名をダブルクリックして「"BuildingRaster@1" ＋ "DemRaster@1"」のように演算式を完成させてください。「ラスタ演算式」のフィールドの下部に「式は正しいです」という表示があれば、正しく解釈されています。なお、ラスタバンドフィールドのレイヤ名の後の「@1」という表記はバンド 1 という意味です。バンドは RGB 画像でいう R や B や G のカラーチャネルのような意味で、今回使っているラスタデータはシングルバンドのデータですが、バンドが複数あるマルチバンドのラスタデータでは@以降の数字が異なる複数のレイヤ名が表示されます。

「出力レイヤ」で「…」から保存するファイルを選択し、「出力形式」を GeoTIFF に設定します。「選択レイヤの領域」をクリックすれば自動的に計算する領域を利用するレイヤの領域に合わせてくれます。「OK」で実行します。

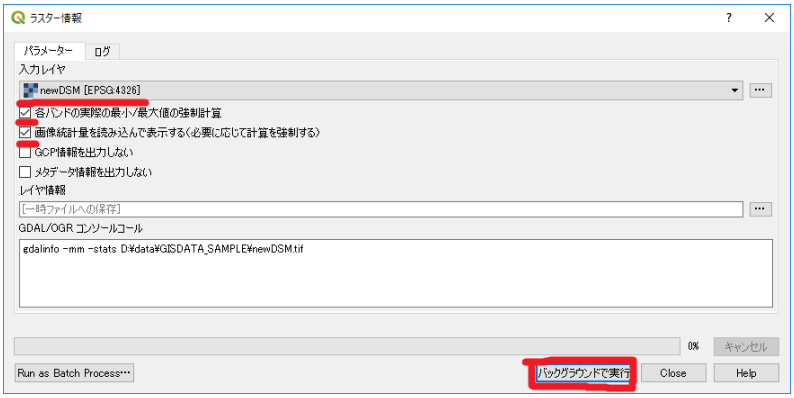


▲図 6.39　ラスタ計算機ダイアログ

最後に DEM と建築物の高さデータを合わせたラスタの数値の型を UInt16 に変換して

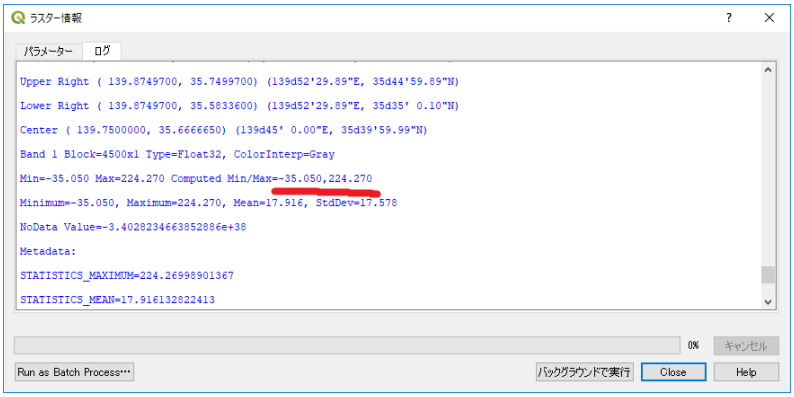
保存します。符号なし整数にしないと Unity ではテクスチャとして扱いにくいためです。そのためには、最小値を計算しておかなければなりません。

メニューから「ラスタ」-「その他」-「ラスター情報」を選びます。「入力レイヤ」に対象のレイヤを、「各バンドの実際の最小/最大値の強制計算」と「画像統計量を読み込んで表示する（必要に応じて計算を強制する）」にチェックを入れ、「バックグラウンドで実行」します。



▲図 6.40　ラスタ情報ダイアログ

計算が終わり「ログ」タブに結果が出力されます。これを見ると最小値は-35.05 と分かります。

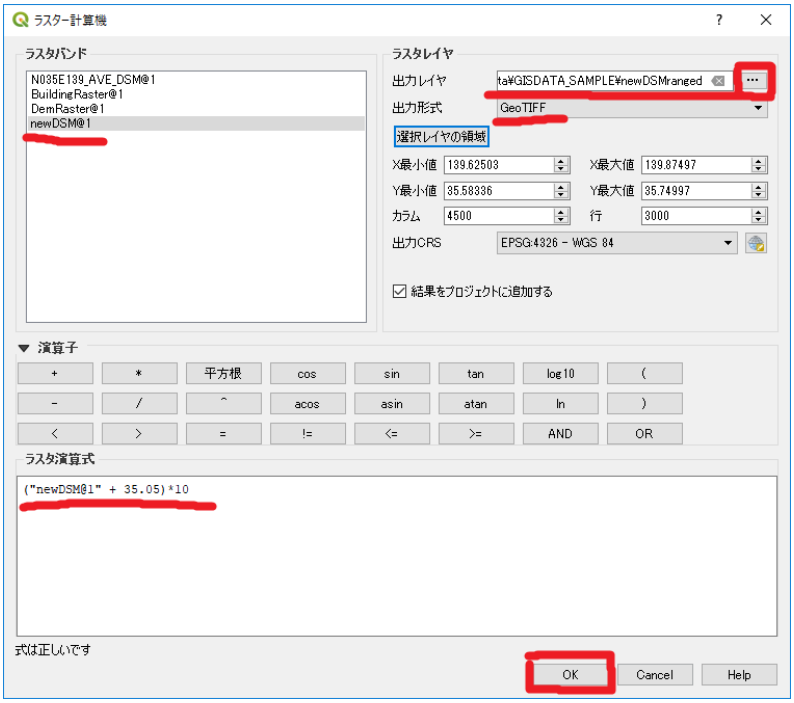


▲図 6.41　ラスタ情報結果

なお、ベクタデータ用にも同様な統計量計算ツールが用意されています。活用する場面があるかと思います。

次に UInt16 に収まるようにラスタ計算をします。得られた最小値をもとにして最小値

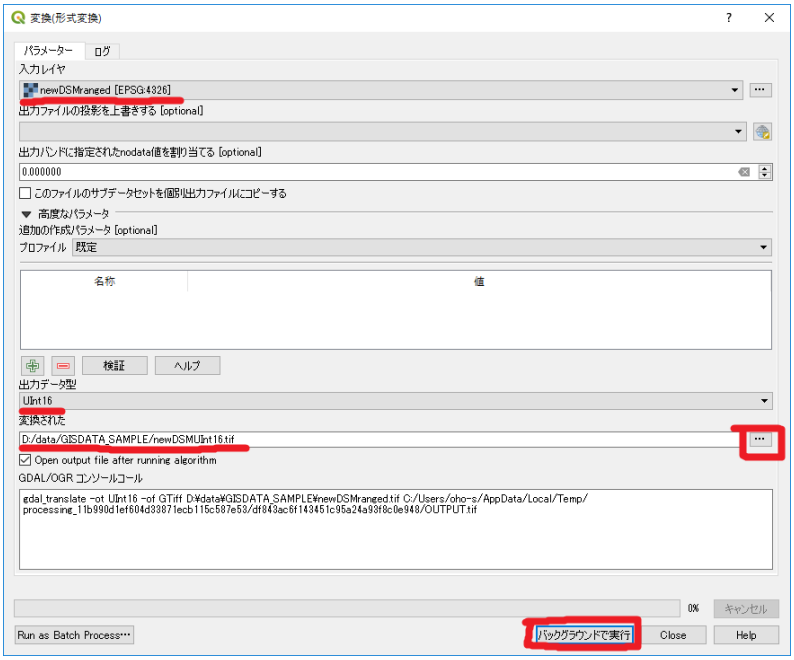
分を足すことにより 0 以上の値域に調整します。また、UInt16 では 0～65535 までの数値を扱えます。富士山の標高が 3776m なので、日本国内のデータを扱うならば 10 倍しても UInt16 の値範囲に収まります。10 倍することで小数点以下 1 桁の 10 ㎝オーダーまでの精度を持たせることができます。ラスタ計算機を用いてこの計算を行います。計算式は`("newDSM@1" + 35.05)*10`となります。



▲図 6.42　ラスタ計算機での値域の調整

作成されたラスタデータはまだ Float32 形式なので、これを UInt16 形式の GeoTIFF に変換して保存します。

メニューから「ラスタ」-「変換」-「変換（形式変換）」を実行します。「入力レイヤ」に値域を調整したラスタレイヤ名を、「出力データ型」に UInt16 を、「変換された」に「…」から保存先ファイル名を指定して入力して、「バックグラウンドで実行」します。



▲図 6.43　UInt16 型で保存

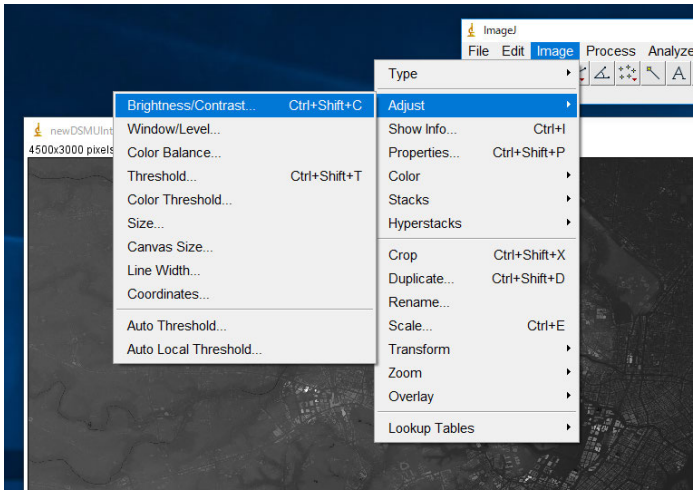
これでハイトマップ用の画像の処理とエクスポートができました。

書き出した画像の処理

16 bit TIFF でもまだ Unity では扱いにくいので、ImageJ を利用してカラーマップと法線マップを計算するための 8 bit 画像を作成します。ImageJ は科学研究での画像解析などに広く利用される画像処理ソフトウェアです。今回は 16 bit TIFF を読み込めることや正確な処理が可能であることなどから使用します。ImageJ の Web ページ[*14]からインストーラをダウンロードしてインストールしてください。

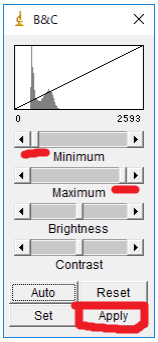
メニューの「File」-「Open」から先ほど UInt16 で保存した GeoTIFF 画像を開きます。8 bit 画像にするまえに明るさ補正をします。メニューの「Image」-「Adjust」-「Brightness/Contrast」を選択します。

[*14] ImageJ の Web ページ：`https://imagej.nih.gov/ij/`



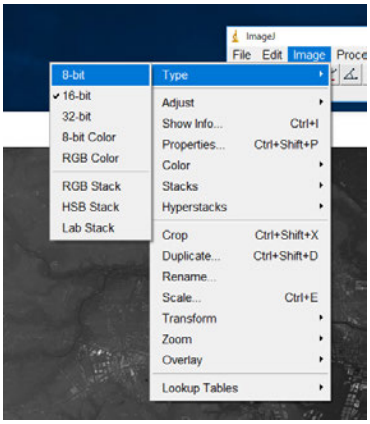
▲図 6.44　ImageJ で明るさ補正をする

明るさ補正のダイアログで、Minimum をスライドバーの最小に Maximum をスライドバーの最大にします。



▲図 6.45　明るさ補正ダイアログ

そして、メニューの「Image」-「Type」-「8-bit」を選択し、色深度を 8 bit に変換します。



▲図 6.46　色深度を 8 bit に変換

最後に別名で保存してください。この画像ファイルがカラーマップとなります。

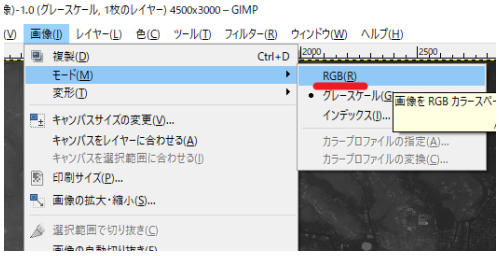
次に法線マップを作成します。法線マップは GIMP のプラグインを利用します。

まず GIMP の Web ページ[*15]からインストーラをダウンロードしてインストールしてください。

そして、法線マップを作成するプラグインを Web サイト[*16]からダウンロードしてインストールします。ダウンロードした Zip アーカイブの readme.txt にあるように、アーカイブ内のファイルを配置してください。

GIMP で画像を開きます。メニューの「ファイル」-「開く/インポート」や、直接 GIMP のウインドウに画像ファイルをドラッグ＆ドロップすることで開きます。

画像を開いたら、メニューの「画像」-「モード」-「RGB」を選択して、グレースケール画像から RGB 画像に変換します。

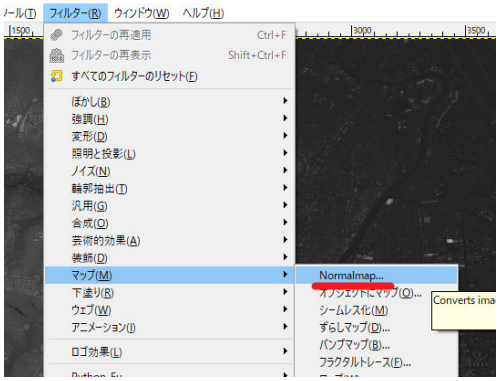


▲図 6.47　RGB 画像に変換

そして、メニューの「フィルター」-「マップ」-「Normalmap…」を選択します。

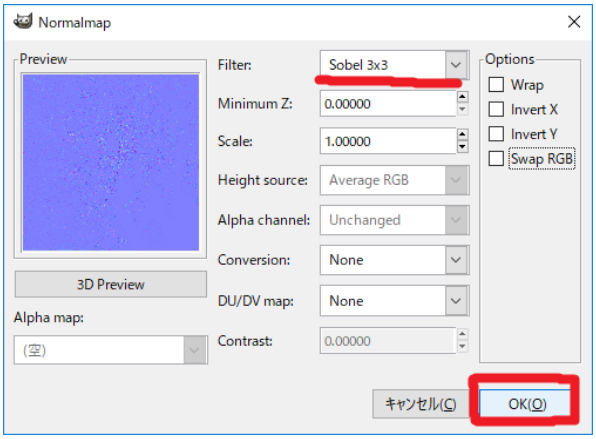
[*15] GIMP Web ページ：https://www.gimp.org/

[*16] GIMP 法線マッププラグインの URL：https://code.google.com/archive/p/gimp-normalmap/



▲図 6.48　法線マッププラグインの実行

ダイアログでは、Filter に試しに「Sobel 3x3」を選んで実行してみます。



▲図 6.49　法線マッププラグインのダイアログ

青い法線マップ画像が作成されました。これも別名で TIFF などにエクスポートします。この画像が、法線マップとなります。

最後に元の 16 bit の GeoTIFF 画像を加工しましょう。このままでは 16 bit の値域をそのまま Unity に取り込むことができないので工夫します。まず、ファイルのプロパティでサイズを確認します。例で用いている画像の場合は 27,024,402 バイトでした。画像データ自体のデータ量は 4500 × 3000 × 2 で 27,000,000 バイトなので、24,402 バイトがヘッダなどの画像データ以外のデータとなります。そこで、この画像ファイルをバイナリエディタで読み込み、先頭から 24,402 バイトまでを削除して、別名で保存します。筆

者はバイナリエディタとして、Stirling[*17]を使っていますが、同等の操作が可能であれば何を使っても構いません。

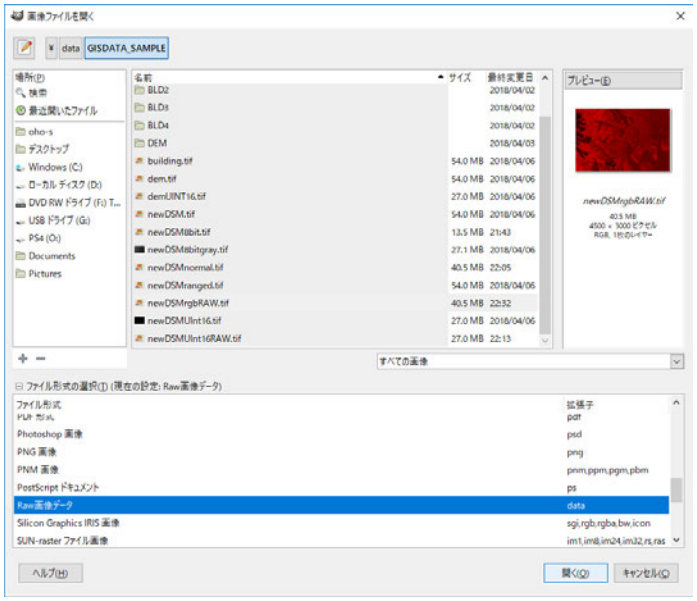
次に、リスト 6.1 を用いて 16 bit で格納されていたデータを 24 bit にして RGB のデータ構造に合わせます。リスト 6.1 は Visual Studio などで適切にコンパイルして実行してください。また、ファイル名などは適宜変更してください。

▼リスト 6.1　16toRGB.c

```
#include <stdio.h>
int main() {
        unsigned char buf[3];
        FILE *fp;
        FILE *fp2;

        // 入力ファイル
        fopen_s(&fp, "dsmraster_uint16.raw", "rb");
        // 出力ファイル
        fopen_s(&fp2, "dsmraster_uint16_rgb.raw", "wb");

        for (int i = 0; i < 4500 * 3000; ++i) {
                fread(buf, 2, 1, fp);
                buf[2] = 0x00;
                fwrite(buf, 3, 1, fp2);
        }
        fclose(fp);
        fclose(fp2);
        return 0;
}
```

最後に、GIMP でこのバイナリファイルを RAW 画像データとして読み込み、TIFF に書き出します。GIMP のメニューから「ファイル」-「開く/インポート」を選択します。対象のバイナリファイルを選択したら、ファイル形式の選択で「Raw 画像データ」を選択して開きます。

[*17] Stirling ダウンロードページ：https://www.vector.co.jp/soft/win95/util/se079072.html



▲図 6.50　GIMP で Raw 画像データとして開く

「Raw データから画像読み込み」のダイアログで、画像の種類に「RGB」幅を「4500」高さを「3000」として開きます。

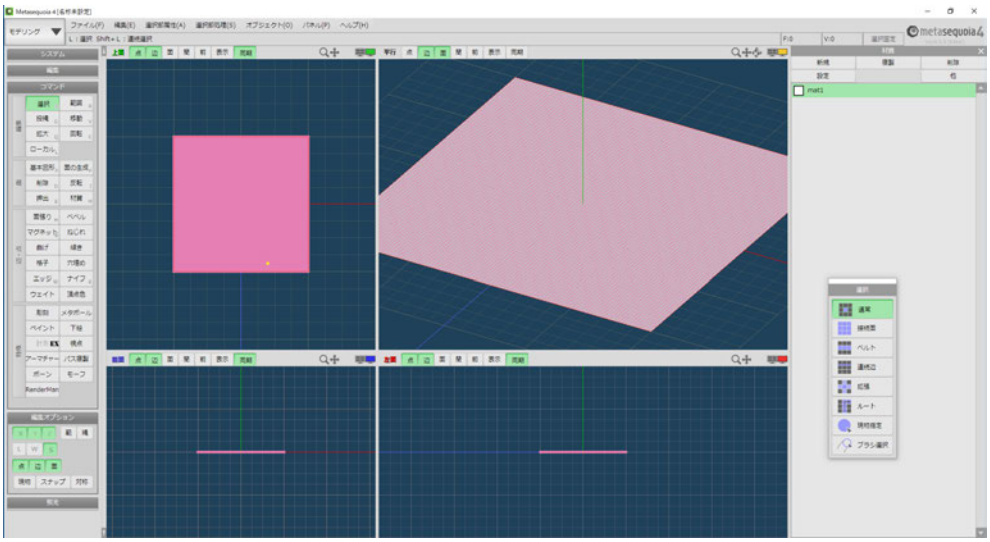
▲図 6.51　Raw 画像データとして読み込むパラメータ設定

画像が表示されたのを確認して RGB の TIFF としてエクスポートします。この画像をハイトマップとして使用します。

6.10　Unity から使う

最初にベースとするための 100 × 100 分割した四角い平面ポリゴンモデルを作成します。今回は使い慣れている Metasequoia 4 Standard[*18]を使いました。

同等の事ができれば使うソフトウェアは何でも構いません。作成したポリゴンモデルは OBJ などの Unity に読み込める形式で保存してください。

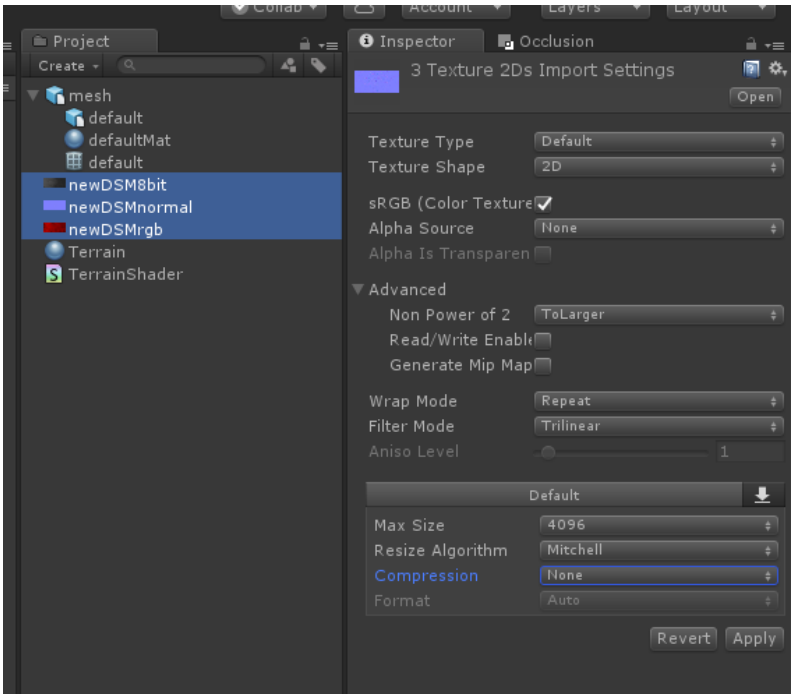


▲図 6.52　Metasequoia でのモデル作成

さて、最後の工程です。Unity に取り込んで表示してみましょう。Unity のバージョンは Unity 2017.4.1f1 を用いました。最近のバージョン（2017 系以降）であれば問題ないと思います。新規プロジェクトを作成します。

[*18] Metasequoia のページ：`http://www.metaseq.net/jp/`

最初にハイトマップ、カラーマップ、法線マップとポリゴンモデルを読み込みます。Project ペインにドラッグ＆ドロップするなどしてプロジェクトにインポートしてください。テクスチャの設定は図 6.53 のようにしてください。法線マップのみ、「Texture Type」を Normal Map にしておきます。



▲図 6.53　テクスチャの設定

次に、適当なシェーダーファイルを新規作成して、リスト 6.2 のシェーダを入力します。

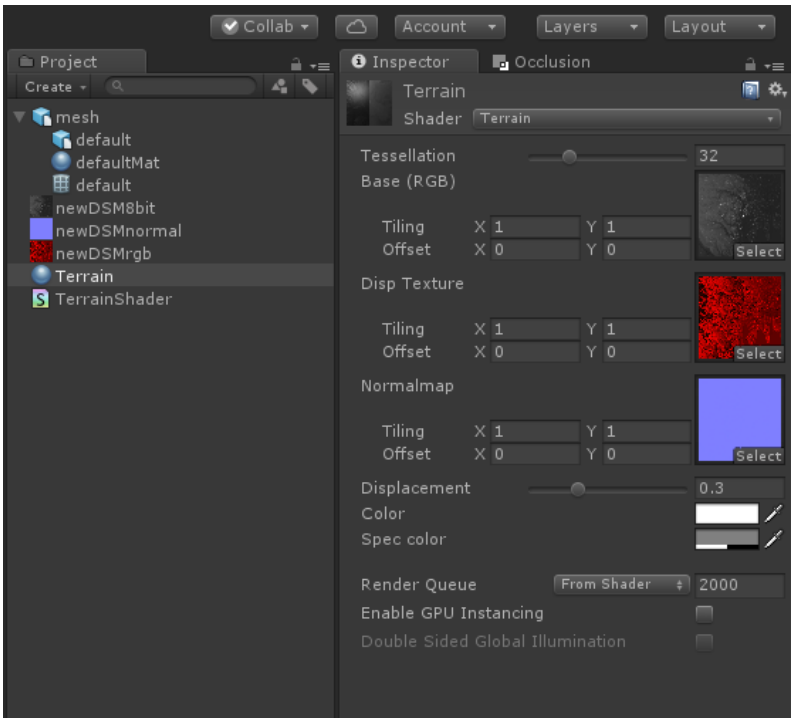
▼リスト 6.2　Terrain.shader

```
Shader "Terrain" {
        Properties{
                _Tess("Tessellation", Range(1,128)) = 4
                _MainTex("Base (RGB)", 2D) = "white" {}
        _DispTex("Disp Texture", 2D) = "gray" {}
            _NormalMap("Normalmap", 2D) = "bump" {}
            _Displacement("Displacement", Range(0, 1.0)) = 0.3
                _Color("Color", color) = (1,1,1,0)
                _SpecColor("Spec color", color) = (0.5,0.5,0.5,0.5)
        }
        SubShader{
                Tags{ "RenderType" = "Opaque" }
                LOD 300
                CGPROGRAM
#pragma surface surf BlinnPhong addshadow fullforwardshadows
        vertex:disp tessellate:tessFixed nolightmap
// ↑紙面の都合上二行に分けていますが一行です。
#pragma target 4.6
                struct appdata {
                float4 vertex : POSITION;
                float4 tangent : TANGENT;
                float3 normal : NORMAL;
                float2 texcoord : TEXCOORD0;
        };

        float _Tess;
        float4 tessFixed()
        {
            return _Tess;
        }

        sampler2D _DispTex;
        float _Displacement;
        void disp(inout appdata v)
        {
            float d = (tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r
                        + tex2Dlod(_DispTex, float4(v.texcoord.xy, 0, 0)).g * 256)
                        * _Displacement;
            v.vertex.xyz += v.normal * d;
        }

        struct Input {
            float2 uv_MainTex;
            float2 uv_NormalMap;
        };
        sampler2D _MainTex;
        sampler2D _NormalMap;
        fixed4 _Color;
        void surf(Input IN, inout SurfaceOutput o) {
            half4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = 0.2;
            o.Gloss = 1.0;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_NormalMap));
        }
        ENDCG
        }
        FallBack "Diffuse"
}
```

シェーダの処理としては、固定値のテッセレーションでポリゴンを分割し、頂点シェー

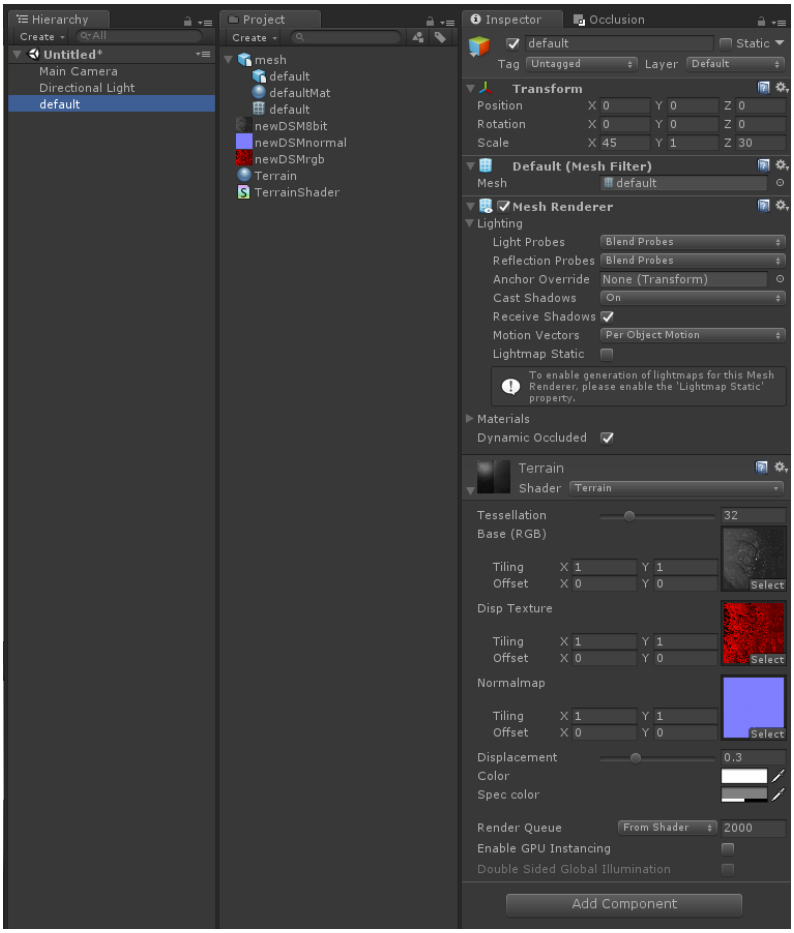
ダでハイトマップ分を法線方向に押し出し、フラグメントシェーダでカラーマップと法線マップをもとに描画しています。頂点シェーダでの計算は、ハイトマップの R 値を下位 8 ビットに、G 値を上位 8 ビットに復元する処理となっています。

そして、マテリアルを新規作成して Shader をリスト 6.2 のシェーダに設定し、用意した 3 枚のテクスチャと各パラメータをそれぞれ設定します。Tessellation の値は 32 くらいに設定します。正確な高さが必要な場合は計算する必要がありますが、Displacement の値は様子をみながら調整してください。



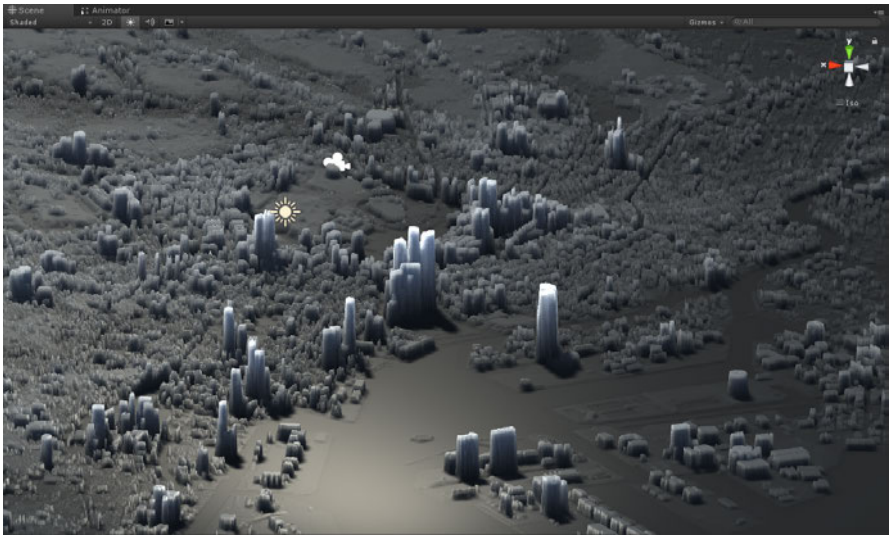
▲図 6.54　マテリアルの設定

最後に Hierarchy にポリゴンモデルを追加し、テクスチャのアスペクト比に合わせて Scale を (45,1,30) などに設定し、先ほど用意したマテリアルを設定します。



▲図 6.55　メッシュオブジェクトの設定

いかがでしょうか、図 6.56 のように GIS から作成した都市が表示されたと思います。



▲図 6.56　結果の表示

6.11　おわりに

今回はデータを画像に落とし込み Unity と連携させましたが、ベクタデータのまま Unity の 3D オブジェクトに変換できると、より表示精度が高くなります。また、Mesh を加工することでゲームなどに使いやすくできると思います。そのためには GIS データを自分の使いたい形式にする独自コンバータの作成などが必要になってきます。さらに、より広い範囲を扱いたい場合はこの巨大なデータを扱うための空間分割をしなければなりません。

機会があれば次はそういった発展的な内容に挑戦したいと思います。

地理情報は背景となる知識やノウハウが専門的であり、またそもそもあまり情報がないこともあって、かなりとっつきにくい分野ではあると思います。しかし、このように実際に操作しつつ扱ってみると、意外と何とかなりそうと感じられた方もいるかと思います。

今回は目標として、Unity に実際の都市を取り込むというゴールを設定してその流れを説明しましたが、GIS はそのほかにもさまざまな処理や分析ができます。災害対策や商圏分析、環境保護、資源探査、そしてもちろんゲーム、さまざまな分野で有効な仕組みです。本稿で少しでも興味を持ち面白いと思っていただけるのであれば、著者として望外の喜びです。

Suguru Oho / @ohomagic

著者

第 1 章 Daisuke MAKIUCHI / @makki_d

眼鏡っ娘が好きです。

第 2 章・第 3 章 Kinuko MIZUSAWA

ふと思ったのですが、最近やっとターミナルとお近づきになれてきた気がします。
個人的にはもうちょっとデレてくれてもいいと思います。

第 4 章 やまだ

花粉症だと認めたくない。

第 5 章 Shunsuke Ito / @fgshun

RimWorld プレイ中。さて明日はどこに墜落しようかな。

第 6 章 Suguru Oho / @ohomagic

最近の楽しみは 4 歳の息子と GT SPORT をプレイすることです。

表紙 たなか

コメント書くのは苦手です。

裏表紙 いしい

久々に仕事で絵描いた

ちびキャラ あべ

最近ガチャの引きが悪い

ダウンロードカード こにし

太眉はいいぞ

ダウンロードカード あきやま

久しぶりに女の子描きました

編集 岡本和樹 / @kakkun61

QoL 爆上げとの噂のドラム式洗濯乾燥機を買った。まだ宅配来てない。

2018 年 4 月 22 日　技術書典 4 電子版（2.1）
著　者　KLab 技術書サークル
編　集　岡本和樹
発行所　KLab 技術書サークル