

KLab Tech Book Vol.8

クラブテックブック

2021 SUMMER



- 01 音響におけるディレイエフェクトの応用範囲
- 02 文章生成モデルGPT-2で遊ぶ
- 03 Unity Build Serverの入門と小技の紹介
- 04 CPython itertoolsを読んでみた
- 05 ARマーカーを使った画像処理でアナログ値の読み取り
- 06 TypeScriptで低レベルメモリ操作入門
- 07 ミニマルGitを自作しよう
- 08 Makefileに秘められた真の力を開放する

Sound Effect

Colaboratory

Transformer

Unity License

PyObject

OpenCV

Memory Layout

Rust

Haskell

GNU Make

KLab Tech Book Vol. 8

2021-07-10 版 KLab 技術書サークル 発行

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 8 弾です。

KLab 株式会社では主にスマートフォン向けのゲームを開発していますが、本書ではこれまでどおり、社内のエンジニアが興味を持っていることや語りたいことについて記事を執筆し、社内のデザイナーの方にも協力していただき一冊の同人誌に仕上げました。

業務への関わりによらず、自分が好きなことを調べるのは楽しい作業です。記事の分野はバラバラですが、それぞれがその分野で面白いと感じたことなどを伝えるために執筆されています。

このような発信活動の場のひとつである本書を通して、読者のみなさまにも好きなことに接する楽しさを感じてもらえると幸いです。

梅澤 寿史

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

はじめに	2
お問い合わせ先	2
免責事項	2
第 1 章 音響におけるディレイエフェクトの応用範囲	5
1.1 基本発音の表記とシングルディレイ	5
1.2 フィードバックディレイ	7
1.3 ディレイエフェクト回路を応用したシミュレーションリバーブ	8
1.4 音響特性の収録による畳み込みリバーブ	8
1.5 簡易フィルタの実現	9
1.6 4000Hz カットオフのフィルタへの応用	10
1.7 他にも	11
1.8 最後に	11
第 2 章 文章生成モデル GPT-2 で遊ぶ	12
2.1 GPT-2 とは	12
2.2 サンプルコード	15
2.3 GPT-2 にいろいろやらせてみる	17
2.4 まとめ	21
第 3 章 Unity Build Server の入門と小技の紹介	22
3.1 Unity Build Server とは	22
3.2 入門	23
3.3 小技 (コンテナ化)	28
3.4 おわりに	31
第 4 章 CPython itertools を読んでみた	32
4.1 はじめに	32
4.2 PyObject 構造体	32
4.3 PyTypeObject 構造体	33

4.4	repeat にみるイテレータプロトコル	34
4.5	その他のイテレータ	35
4.6	終わりに	39
第 5 章	AR マーカーを使った画像処理でアナログ値の読み取り	40
5.1	はじめに	40
5.2	状況設定と戦略	40
5.3	AR マーカーを使った処理	41
5.4	画像処理	47
5.5	値の検証	51
5.6	考察	55
5.7	おわりに	56
第 6 章	TypeScript で低レベルメモリ操作入門	57
6.1	前提	57
6.2	単純な表現: 白、黒、または石なしの 3 ステート	58
6.3	Rust との連携	60
6.4	少し複雑な表現: フリップを GPU で行う	62
6.5	おわり	65
第 7 章	ミニマル Git を自作しよう	67
7.1	Git のしくみと構造	68
7.2	実装	77
7.3	終わりに	86
第 8 章	Makefile に秘められた真の力を開放する	90
8.1	Makefile の基本	90
8.2	ファイル名のパターンを使ったルール	91
8.3	ファイルの内容からルールを生成する	92
8.4	まとめ	95
	執筆者・スタッフコメント	96

第1章

音響におけるディレイエフェクトの 応用範囲

Mitsuteru Furukawa / @ttt_333

音響における**ディレイエフェクト**とは、ソースとなる音源が遅れて発音される効果です。これをたくさん組み合わせることで、それ以上の効果を得ることができます。

その効果は、フィードバックディレイ、リバーブのようにディレイエフェクトの効果をリッチにしたもの、フィルタ、イコライザ、フランジャ、フェイズシフタ、ワウワウのようにディレイエフェクトの効果からは想像が出来ないものと多様です。

今回は、これらの中からディレイエフェクトのバリエーション、応用例としてリバーブ、フィルタを例に説明します。

また、Windows 環境で動作するサンプルアプリケーションを次の URL に用意しましたので、実際の音もお聴きいただけます。

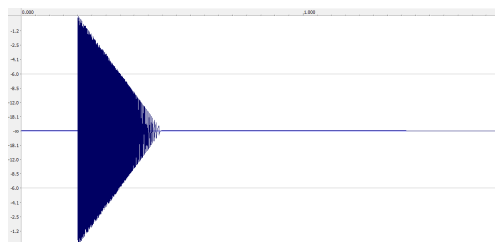
Windows 環境以外の方には render フォルダ以下にノコギリ波のスイープ DOWN のワンショット波形へ実際に効果を適用した波形ファイルを用意してありますので、そちらをご試聴ください。

- <https://github.com/ttt-333/ConvolutionPlayerSample>

1.1 基本発音の表記とシングルディレイ

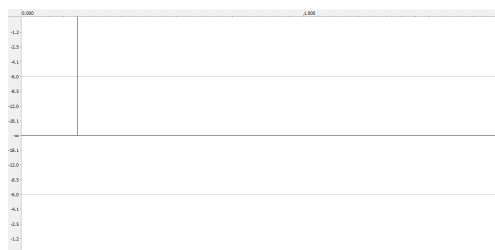
以降の図は元の音声を発音したものを時間領域で表記します。縦軸は振幅、横軸は時間をあらわしています。

図 1.1 は実際に任意のワンショット波形の再生したものを時間領域でモノラル、48KHz で表した図です。(波形ファイル:001_impulse.wav)



▲図 1.1 実際のワンショット波形の時間領域表記図

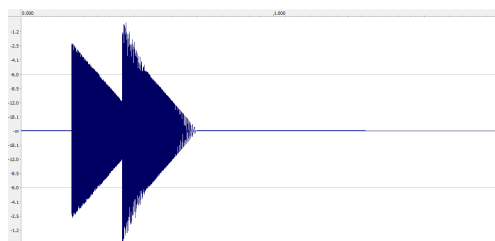
図 1.1 のワンショット波形の時間軸上の発音開始時点を取り出すと、一瞬だけ正相に発生する音源の図となります。



▲図 1.2 基本発音の時間領域表記図

この波形は**インパルス**と表記されます。また、この図 1.2 はインパルスを入力したときの発音システムの出力で**インパルス応答**と呼ばれるものを時間領域で図解したものです。

次に、元のワンショット波形を発音時間をずらして発音させます。図 1.3 だと 200ms 後に元と同じ音源が同じ音量で発音されます。(波形ファイル:002_delay_200ms.wav)



▲図 1.3 シングルディレイの時間領域表記図

この単発エコーのエフェクトは**シングルディレイ**と表現されます。図 1.3 のワンショット波形の時間軸上の発音開始時点を取り出すと、200ms 間隔で 2 回分、正相に発生する音源の図となります。



▲図 1.4 シングルディレイの時間領域表記図

以降の説明はインパルス応答を時間領域で図解したものを元に進めます。

1.2 フィードバックディレイ

シングルディレイの後、200ms 間隔で連続的に音量を下げながら音声を発音させます。

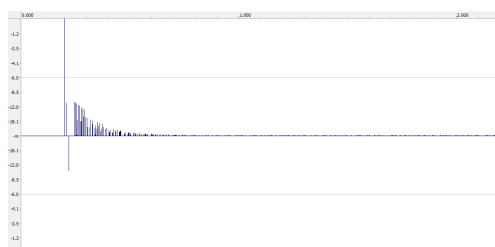


▲図 1.5 フィードバックディレイの時間領域表記図

この等時間の間隔で繰り返すディレイは**フィードバックディレイ**と表現されます。同じ音が繰り返しながら消えていきます。(波形ファイル:003_fb_delay_200ms.wav)

1.3 ディレイエフェクト回路を応用したシミュレーションリバーブ

これらのディレイの効果を応用して部屋の残響効果等も疑似的に再現できます。



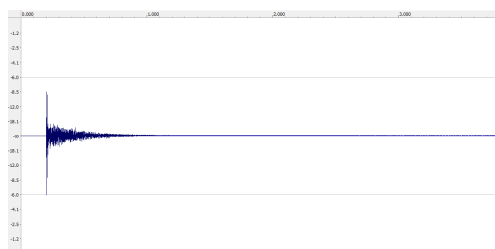
▲図 1.6 ディレイエフェクト回路を応用したシミュレーションリバーブの時間領域表記図

この音響エフェクトは**ルームリバーブ**と表現されます。その名のとおり部屋の中の残響音のような、複雑に音が重なり合う豊かな響きになります。

位相が反転されている部分がありますが、壁に低周波が吸収されていく過程を再現しています。(波形ファイル:004_digital_reverb.wav)

1.4 音響特性の収録による畳み込みリバーブ

図 1.6 でディレイの位相が反転されていた箇所がありましたが、さらに音の位相を反転したディレイを増やしてみます。さらに、全体的なディレイの密度も増やします。



▲図 1.7 音響特性の収録による畳み込みリバーブの時間領域表記図

こうすることで音源の残響が複雑なものになっていきます。

これは、コンサートホールのように残響時間の長い部屋の音響特性を収録により再現したものになります。(波形ファイル:005_impulse_record_reverb.wav)

1.5 簡易フィルタの実現

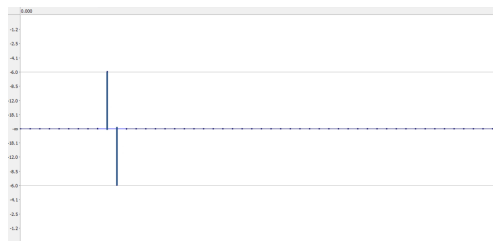
また、音量を半分にして極端な短時間（目安として 10 マイクロ秒、1～数サンプル等）でエコーを発生させると高周波成分を削ることができます。



▲ 図 1.8 簡易ローパスフィルタの時間領域表記図

図 1.8 では、2 サンプル分の連続ディレイでは高周波成分を削る効果が分かりにくいので、4 サンプル分の連続ディレイを発生させることで効果を強くしています。

そして、このエコーの位相を反転させると音の低周波成分を削ることができるのです。（波形ファイル:006_posi_posi_4sample.wav）



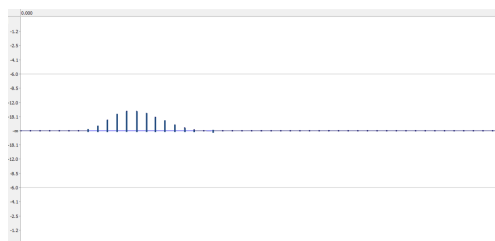
▲ 図 1.9 簡易ハイパスフィルタの時間領域表記図

図 1.9 では 1 サンプル単位でエコーを発生させた場合になります。（波形ファイル:007_posi_neg_2sample.wav）

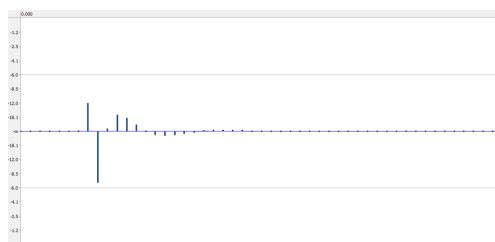
1.6 4000Hz カットオフのフィルタへの応用

これらの性質を応用すると、音の好きな周波数を操作することが可能になります。

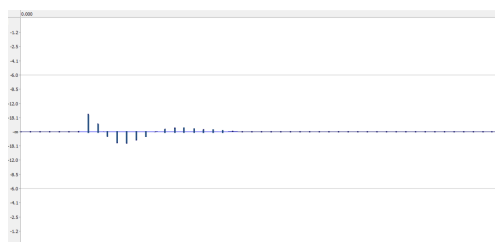
以下、4000Hz をカットオフにしたローパスフィルタ、ハイパスフィルタ、バンドパスフィルタのインパルス応答の図となります。(波形ファイル:008_lo_pass_4000hz.wav、009_hi_pass_4000hz.wav、010_band_pass_4000hz.wav)



▲ 図 1.10 4000Hz をカットオフにしたローパスフィルタの時間領域表記図



▲ 図 1.11 4000Hz をカットオフにしたハイパスフィルタの時間領域表記図



▲ 図 1.12 4000Hz をカットオフにしたバンドフィルタの時間領域表記図

1.7 他にも

今回は取り上げませんでしたが、音がうねる効果（フランジヤ、フェイズシフタ、ワウワウ等）についてもディレイエフェクトのパラメータを応用したものになります。

1.8 最後に

実は音のエフェクトプラグインの効果の多くはディレイエフェクトの応用で成り立っているということを説明させていただきました。

ディレイエフェクト自体の動作原理については実は単純なのですが、欲しい効果を得るパラメータとなる、**発生させるタイミングや大きさ、位相の正逆、を発生させる回数**を求める所が大変です。

また、DAW、波形編集アプリケーション等でも今回解説したディレイエフェクト応用で成り立っているプラグインエフェクトがあります。

これらのアプリケーション上でインパルスの波形をプラグインエフェクトを適用して再生、録音することで、プラグインエフェクトのインパルス応答の波形を取り出すことができます。

興味があれば、試してみると面白いかもしれません。



第2章

文章生成モデル GPT-2 で遊ぶ

Atsushi Takada

この記事で紹介するのは OpenAI が開発した文章生成用の機械学習モデル GPT-2 です。GPT-2 は 2019 年に公開された、簡単に自然な文章が生成できるモデルです。このため、当初よりフェイクニュースなどへの悪用が懸念され、モデルは慎重に公開されました。

その後、2020 年には GPT-2 をさらに改良した GPT-3 が発表され、大きな話題を呼びました。こちらはかなり話題になったので耳にしたことがある方も多いでしょう。GPT-3 は自然な文章を作成できるのはおろか、指示にしたがって簡単なソースコードを書いたり、翻訳なども実現できるそうです。

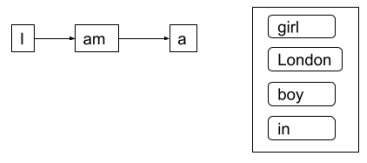
悪用防止のため、GPT-3 の学習済みモデルは現在でも公開されておらず、OpenAI の API からのみアクセスできる状態です。一方 GPT-2 の方は学習済みのモデルもいくつか公開され、2021 年 4 月には、rinna 株式会社が日本語で学習済みのモデルを公開しました。このモデルは、transformers などのライブラリを使って、非常に簡単に使用することができます。

本章では GPT-2 の紹介とともに、この日本語学習済み GPT-2 モデルを使うサンプルコードを紹介し、いろいろ文章を生成して遊んでみたいと思います。

2.1 GPT-2 とは

言語モデルとは

GPT-2 は「言語モデル」と呼ばれるモデルの一種なので、まず言語モデルについて説明します。簡単にいえば、言語モデルとは、文の途中までを見て次に来る単語を予測するモデルです。



▲図 2.1 次に来る単語を予測

たとえば図 2.1 のように、英語の文で、“I am a”の次に来る単語を予測するケースを考えてみましょう。次に来る単語として、“boy”や“girl”はありえるでしょうが、“London”、“in”、“I”などがつづく確率は非常に低そうです。

自然言語のテキストを大量に学習させることで、こうした確率分布を機械学習モデルに学習させ、次に来る単語を予測できるようにしたものが言語モデルです。

次に来る単語を予測するタスクを繰り返し実行し、確率的にありそうな単語列を作っていけば文章生成も実現できます。当然ながら、この方法では文章の意味を考慮しているわけではないので、よく読むとおかしな文章になっていることも多いですが、意外に自然な文章が生成されることも多いです。

ここで、GPT-2 で生成した日本語の文章のサンプルをいくつかあげておきましょう。次のサンプルでは太字の部分を GPT-2 に与え、そのつづきをモデルに生成させています。

都内で発見された dna の解析結果によると、この女性は 1 年ほど前に 30 歳になったばかりで、身長は 178 cm、体重は 2 kg でした。この女性には、2 人の子どもがいましたが、そのうちの 1 人は 40 歳で、もう 1 人には 50 歳の男性がいます。この男性は 3 年前に死亡しています。dna 解析の結果では、男性の死因は心臓病

進捗やばいんですがもうちょっと待っててくださいね (´; ω;) ウウウ まあ、そんなこんなで、今年もよろしくお願ひします m(__ __)m

文章が生成されるといっても javascript で書かれているので html や css の知識がなくても web ページを簡単に作成することができます。今回は wordpress をインストールしてみようと思います。まずは、インストールの手順をご紹介します。インストール手順は簡単ですが、注意点もあります。この記事を読めば、初心者でも簡単にインストールできるはずですよ。ぜひ、参考にしてみてください。1. ダウンロードしたファイル

どうでしょうか。まあまあ自然に見えますね。それぞれ「ニュース」「SNS」「技術的なチュートリアル文書」の体裁になっているようです。

また、たとえば、最初の文章などは、一見すると自然な日本語に見えますが、よく読むと身長 178 cm で体重が 2kg だったり、30 歳になったばかりの女性に 40 歳と 50 歳の子どもがいたり、設定がめちゃくちゃです。このように、言語モデルで生成された文章は、数字部分が適当だったり、論理的な整合性がおかしいことは多々あります。

背景となる技術: 埋め込みベクトルとトランスフォーマー

「言語モデル」という用語は、あくまで「次に来る単語を予測するモデル」という程度の意味なので、実装はさまざまです。

たとえば、素朴な実装としては、単純に学習するテキストの二単語、三単語の連鎖をすべて集計し、そこから確率を算出するという手法も考えられます。こうした実装は昔から使われており、同じような仕組みでチャットボットなどを作ったことがある人も多いのではないのでしょうか（わたしも作ったことがあります）。

ただし、単純にすべての連鎖を集計するような素朴な実装だと、長い連鎖を考慮すると組み合わせ量が膨大になってしまいます。そのうえ、はじめて見る連鎖には対処しようがないという問題もあります。

一方、GPT-2 など、近年の機械学習モデルでは、ニューラルネットワークに自然言語のテキストを学習させることで、ある程度長い連鎖でも扱うことができるようになっていきます。

こうした機械学習モデルの構成要素をすべて説明するのは難しいので、ここでは重要な部分のさわりだけ説明します。具体的には、(1) 埋め込みベクトルと、(2) トランスフォーマーというふたつについて簡単に解説します。なお、次の解説については、後の部分を読む上で必須というわけではないので、それほど興味がなければ読み飛ばしてもらってもかまいません。

(1) 埋め込みベクトルについて

埋め込みベクトルは、機械学習による自然言語処理で現在きわめて汎用的に使用されている技術です。これは一言でいえば、自然言語の単語を数値のベクトルに変換する手法です。

ニューラルネットワークで日本語や英語の単語を扱うためには、単語を何らかの数値データに置換する必要があります。この数値データへの置換方法として現在よく使用されているのは、各単語を数百次元の数値ベクトルに変換し、変換処理自体も学習可能なネットワークとして、データから学習させるという手法です。

埋め込みベクトルの学習が理想的に進んだ場合、似た役割をもった単語は似たベクトルに置換されるようになります。ここで、「似た役割をもった単語」というのは、互いに置換可能な単語を指しています。たとえば、「千葉」と「埼玉」という単語はどちらも県名であり、おそらく文中の似た箇所に登場するはずです。「千葉に住んでいる」という文章の「千葉」の部分「埼玉」に置き換えても、「埼玉に住んでいる」という自然な文章になるでしょう。埋め込みベクトルを使った機械学習モデルは、この「千葉」と「埼玉」のように、似た役割をもった単語を近いベクトルに変換します。

おそらく、以上の説明だけではイメージしづらいと思いますが、ここでは単純に、自然言語の単語をベクトルに変換しており、意味の近い単語はベクトルの距離が近くなる、ということだけ抑えておけば十分です。

(2) トランスフォーマーについて

GPT-2 や GPT-3 で使用されている技術の中でも、特に重要なのがトランスフォーマーです。トランスフォーマーについて詳しく解説するとそれだけで一章分（へたしたら本一冊分）くらいになってしまうので、ここでは概要だけにとどめておきます。

トランスフォーマーは、ニューラルネットワークの中の特殊な層の一種です。「アテンション」と呼ばれる処理を組み合わせることで、自然言語の文のように順序をもった情報を扱えるようにしたものがトランスフォーマーです。

少し前までは、この種のタスクには再帰型ニューラルネットワーク（RNN）と呼ばれる層が使用されていましたが、現在ではトランスフォーマーが覇権をにぎっています。機械翻訳、文章生成、テキスト分類など、自然言語処理におけるあらゆるタスクでトランスフォーマーが使用されています。GPT-2 以外では、自然言語処理で注目されている BERT などのモデルもトランスフォーマーベースです。また自然言語だけではなく、画像認識などの分野でも、トランスフォーマーを使ったモデルが注目されはじめています。

トランスフォーマーの重要な特徴は、いくらでも層を重ねることができる点です。この特徴があるため、学習データの量を増やし、層を重ねてモデルのサイズを大きくしていくことで、際限なく性能を上げることができるといわれています。

このため、トランスフォーマーを使ったモデルは巨大なモデルになりがちです。GPT-3 はパラメーター数が 1750 億あり、学習にも膨大な費用と時間がかかっています。GPT-2 は GPT-3 に比べれば規模はずっと小さいですが、それでも、それなりに大きなモデルであることには変わりありません。手元の環境^{*1}では、200 語程度の文章を生成するのに、CPU で 2 分程度、GPU を使用した場合には 4 秒程度かかりました。

2.2 サンプルコード

GPT-2 日本語学習済みモデルを使って、日本語の文章を生成するサンプルコードを解説していきます。次のサンプルコードは、Google の Colaboratory というサービスを使って動かすことを想定しています。Colaboratory は Python のノートブックを実行できるサービスで、機械学習関連のライブラリがあらかじめインストールされた環境を無料で使用することができます (<https://colab.research.google.com/>)。ただし、Colaboratory 固有の部分はそれほどないので、Jupyter Notebook やコマンドラインの Python でも同様のコードを動かすことができます。

また、以下では Hugging Face 社の transformers ライブラリを使用します。transformers はトランスフォーマーベースのモデル群を統一的に使用できるようにしたライブラリで、これを使うことで簡単に GPT-2 日本語学習済みモデルを使用することができます。

^{*1} ゲーミング PC。CPU ひとつのみ使用時。GPU は GeForce GTX 1080Ti をひとつのみ使用時。

第 2 章 文章生成モデル GPT-2 で遊ぶ

まず関連ライブラリをインストールします。Colaboratory で新規ノートブックを作成し、次のコードを最初のセルに貼りつけて実行してください。これによって、pip で transformers と sentencepiece ライブラリをインストールすることができます。

```
! pip install transformers
! pip install sentencepiece
```

rinna 株式会社の日本語学習済みモデルを使用するには、モデル名を指定して、トークナイザーと、モデルのインスタンスを作成するだけです。以下を新しいセルに貼りつけて実行してください。

```
from transformers import T5Tokenizer, AutoModelForCausalLM

tokenizer = T5Tokenizer.from_pretrained("rinna/japanese-gpt2-medium")
model = AutoModelForCausalLM.from_pretrained("rinna/japanese-gpt2-medium")
```

なお、トークナイザーは、日本語の文章を単語に分かち書きし、各単語を GPT-2 が理解できる数値 ID に変換するものです。モデルの学習時に使用されたトークナイザーと違うものを使ってしまうと結果もおかしくなる可能性があるため、トークナイザーはモデルに対応したものを使用するように注意してください。モデルを使って文章を生成するには、トークナイザーを使ってテキストを変換し、モデルの generate メソッドを呼び出すだけです。

```
import torch

def generate(input_str, max_length=200):
    input_ids = tokenizer(input_str, return_tensors="pt").input_ids.to(device)
    beam_output = model.generate(
        input_ids,
        max_length=max_length,
        num_beams=5,
        no_repeat_ngram_size=2,
        num_return_sequences=5,
        early_stopping=True
    )
    print("Output:\n" + 100 * '-')
    print(tokenizer.decode(beam_output[0], skip_special_tokens=True))

device = "cpu" # GPUを使用する場合は cuda に変更
model = model.to(device)
generate("こんにちは-", 100)
```

以上を実行すると、input_str に与えた文章（ここでは「こんにちは-」）のつづきが生成され表示されます。わたしが実行した際には、次の文章が作られました。

こんにちは- 今日、お客様から頂いたお花をご紹介します

あとは、generate 関数に入力文字と長さを渡せば任意の文章が生成できます。

2.3 GPT-2 にいろいろやらせてみる

GPT-2 の可能性を探るため、いろいろな方向の文章を書かせてみます。なお、文章生成を試す際の注意点ですが、GPT-2 は入力された文章から文脈を読みとるので、できれば入力する文字列も一語や二語ではなく、ある程度長めの文章を与えた方が生成される文章の質もよくなります。

社内メールを書かせてみる

社内向けのメールの文章を考えるのがめんどろなときに使えないかと思って試してみました。

お疲れ様です。開発推進部高田です今回は、前回のブログでご紹介させていただいた新商品のご案内をさせていただきます。今回ご紹介するのは、「おにぎりせんべい」と「おむすびまんじゅう」の2商品です! どちらも、お子様からご年配の方まで、幅広い年齢層の方々にお楽しみいただける商品となっておりますので、ぜひ一度お試しくださいませ。また、この度、弊社ではお客様により便利にご利用いただくため、ホームページをリニューアルいたしました。お客様にはご不便をおかけいたしますが、何卒ご了承くださいませよう、よろしく願い申し上げます m(_ _)m さて、本日は弊社のお盆休みについてお知らせしたいと思います。8月11日(土)~8月15日(水)までお

謎の新商品が開発されていてウケました。

名前を変えてみます。

お疲れ様です。開発推進部佐藤です今回は、前回の続きで、開発の進め方についてお話したいと思います。前回の記事はこちらからご覧いただけます。[...] [続きを読む] こんにちは! 開発部 佐藤でございます(*´ω*) さて、本日は前回に引き続き、弊社で開発を進めておりますアプリのご紹介をさせていただきます(^^)/ 今回のアプリは「おにぎり屋さん」です! お客様がお店にいらっしゃった時に、お客様にお弁当をお届けするアプリです(^ω^) このアプリを使えば、ご自宅に居ながらにして、美味しいお米やお惣菜を手軽に買うことができます! また、このアプリでお買い物をする、ポイントが貯まり、貯まったポイントで商品と交換

名前を変えると内容も変わりましたが、なぜかおにぎりにはこだわりがあるようです。

今度は所属を少し変えてみます。

お疲れ様です。機械学習グループの高田です今回は、前回の続きで、ディープラーニング(深層学習)について書きたいと思います。前回の記事でも書きましたが、deep learning(以下、cnn)とは、ニューラルネットワーク(神経細胞の集合体であるニューロン)を用いて、人間の脳の神経回路をモデル化し、そのモデルをコンピュータに学習させる学習手法のことを指します。この手法を用いることにより、人間は学習したモデルをもとに、機械に命令を実行させることができ、その結果として、学習されたデータから、最適な学習結果を導き出すことができますようになります(図1)。また、この手法を用いると、画像認識や音声認識など、様々な分野での応用が期待されています。例えば、人工知能(ai)の分野に応用すると、大量の画像データの中から、適切な画像を抽出し、それを機械が学習することで、より精度の高い学習を行うことができます(図2)。このように、

所属をグループ名に変えると、機械学習という言葉が入っているせいか、ディープラーニングの解説がはじまりました。これはちょっと修正すればそのまま使えそうです。内容もだいたい合ってます。

技術文書を書かせてみる

技術書典なので技術解説も書かせてみたいと思います。GPT-2は技術解説はわりと得意なようですが、GPT-2のトークナイザーが用語に対応しないケースが見られるのがネックです。

まずトランスフォーマーの解説です。先ほど書いた紹介の一部を渡して、続きを書かせてみます。

transformerはニューラルネットワークの中の特殊な層の一種です。「アテンション」と呼ばれる処理を組み合わせることで、自然言語の文のように、複雑な文を処理することができます。今回は、この「ニューロン」について、その特徴や仕組みについて解説します。アテンションの仕組みを理解するために、まずは「文」と「文章」の違いについて理解しておきましょう。まず、文とは、文章を構成する要素の集合のことです。文章には、主語と述語があります。主語は文章の先頭に置かれ、述語は文の最後に置かれます(文末)。文は文章を区切る役割を担っていますが、文末には改行やスペースがありません。また、句読点を打つこともできません。このように文章は文と文の間に「区切り」があるので、複数の文が組み合わさって構成されています。また、同じ文の中に複数の文章が混ざっている場合は、1つの文

方向性はまちがってはいませんが、どちらかというと「文」と「文章」の説明になっています。

次に python の紹介。

python とは ruby on rails の略で、ruby で書かれた web アプリケーション開発のための言語です。今回は、この javascript を html に埋め込む方法をご紹介します。まず、ブラウザで `http://localhost:8080/` へアクセスしてみましょう。この url にアクセスすると、以下のような画面が表示されます。「hello world!」と表示されれば ok ですが、もし表示されない場合はブラウザのキャッシュをクリアしてみてください。次に、html の `header.php` と `body.css` の 2 つのファイルを編集し、以下のコードを追記してブラウザに表示させます

Python にも Ruby にもケンカを売っている感じの文章です。ただし、これは入力が短かすぎたかもしれません。

こんどは IP アドレスの説明。

ip アドレスとは、インターネット上の住所のようなものですインターネットに接続されているパソコンやスマートフォンなどの機器を識別するための番号です。インターネットに接続するためには、プロバイダー (isp) と契約する必要があります。isp はインターネット接続サービスを提供する会社ですが、そのサービスを提供しているのは「インターネットサービスプロバイダ」と呼ばれています。このサービス提供会社には、大きく分けて 2 つの種類があります。「isdn」と「光ファイバー」の 2 つですね。この 2 種類の回線を総称して「ipv6」と呼んでいます。ipv6 のサービスを利用するには、「pppoe」という技術が必要になりますので、まずはこちらをご覧ください。まずは、「フレッツ・adsl」でインターネット

合ってる部分もありますが、雰囲気だけです。

物語を書かせてみる

次は、もっとクリエイティブなタスクに挑戦させてみましょう。物語のはじまりっぽい書き出しを与え、つづきを作らせてみます。

むかしむかし遠い銀河の誰も知らない国にひとりの少女がいました。その少女はとても美しい少女でしたが、ある日 突然 姿を消してしまいました。それから何年も 時が経ちましたが、少女の消息を知る者は 誰もいません。ある日 一人の少年が ふと 空を見上げると、そこには美しい星空が広がっていたのです。少年はその星に興味を持ち、その星を調べてみると、なんとその星には たくさんの星が輝いていて、とてもきれいな星だったのです。そして、少年はその星の美しさに魅了され、いつしか 星の虜になってしまっていたのでした...。「星の王子さま」の作者である アントワーヌ・ド・サン＝テグジュペリは、この星の美しさを「星の輝き」と表現しています。この星の美しい輝きを、あなたもぜひ 見つけてみてください

よく読むと変なことも言ってますが、何となくそれっぽいです。

もっといろいろなジャンルに挑戦してみましょう。

まずはファンタジー。

ある国にひとりの魔法使いがいました。ある日魔法使いはドラゴンに連れて行かれてしまいます。そしてドラゴンは人間に化けて人間界にやってきました。しかし、そのドラゴンの正体はなんと人間ではありませんでした…。人間とドラゴンが共存する世界。その世界では、人間には見えないものが見えるという不思議な力があるのです。その力を使って、ドラゴンを人間の姿に戻してあげるのが、魔法使いの役目なのです。この世界には、たくさんの不思議な力があります。そんな不思議な力を、あなたも使ってみませんか？あなたは、この世界の秘密を知っているのでしょうか。それとも、知らないのですか？この世界の謎を解き明かして、

この世界の秘密を知っているのか知らないのか。

つぎにミステリー。

平凡な生活を送る会社員の山田。だが、ある日山田の周囲をかきまわるあやしい人物が気になって仕方ない。そんなある日、山田は謎の男に拉致されてしまう。山田が拉致されたのは、東京の下町。そこには山田を誘拐した男と、その男を拉致した女がいるという噂が流れている。その噂の真偽を確かめるため、男は山田に会いに行くが、そこで男の姿が消えてしまう。そして、再び山田の前に姿を現した男は、なんと山田の妻だったのだ。そして、この物語の鍵となるのが、「山田」と「あや」という2人の女性の存在だ。山田とあやは結婚し、子供を産み、幸せな家庭を築いていた。しかし、ある事件をきっかけに、2人の間に亀裂が入ってしまう。2人は、お互いの気持ちに気づかないまま、結婚生活に終止符を打つことになるのだが、そこに現れたの

山田が拉致された後、山田を誘拐した男も拉致され、かなり複雑な状況のようです。

こんどはSF。

銀河帝国で平和な生活を送るロボットと少年が宇宙の平和を守るために戦う姿を描いたsfアクション映画『スター・ウォーズ/フォースの覚醒』が、いよいよ2017年12月15日(金)に公開されます。今回は、そんな本作の魅力や見どころ、キャストやあらすじなどについてご紹介します。『スター・ウォーズ』シリーズでおなじみのルーク・スカイウォーカーとレイア・オーガナの2人を中心に、反乱軍と反乱軍の対立が激化する中、ルークとオビ=ワン・ケノービ、そしてルークの親友であるカイロ・レンの3人が中心となり、帝国軍と反乱軍が激突する壮大な物語が描かれています。また、本作の監督を務めるのは

SF というか『スター・ウォーズ』になってしまいました。『フォースの覚醒』そのままかと思いきや、ルークとカイロ・レンが親友であるといった設定変更も追加されているようです。

2.4 まとめ

この記事では文章生成用の機械学習モデル GPT-2 を紹介しました。まだまだ変な出力もたくさんありますが、意外なほどきちんとした文章が作られることもあります。GPT-2 は簡単に使えますし、他にもいろいろな使い方が考えられると思うのでぜひ試してみてください。

余談ですが、まとめの文章も GPT-2 に書かせてみようと思って試してみたのが次になります。

以上では、文章生成用の機械学習モデルを紹介してきました。いかがだったでしょうか？今回は、前回の続きです。前回の記事はこちらからご覧いただけます。さて、前置きが長くなってしまいましたが、本題に入りたいと思います。まず、今回紹介するのは、「文章を生成して、その文章から文章を生成する」という方法です。この方法を使えば、簡単に文章が生成できますので、ぜひ試してみてください。この記事を読んでくださっている方は、きっとプログラミングに興味がある方だと思うのですが、「プログラミングって難しそう」と思っている方もいらっしゃるかもしれません。私もそうでした。しかし、プログラミングは決して難しいものではありません。プログラミングの本を読んだり、ネットで調べたりすれば、ある程度は理解できるでしょう。しかし、実際にプログラミングをやろうとすると、何から手をつけていいかわからないという方も多いのではない

第3章

Unity Build Server の入門と小技の紹介

Keita Yamamoto

業務で自動化していますか？

はじめまして、Yamamoto を担当している KLab の雑用兼便利屋です。今回は KLab の業務で実際に使用されているシステムの一部と小技を紹介をしたいと思います。

KLab ではほとんどのプロジェクトが Unity を使用してゲームを作成しており、そのすべてのプロジェクトで成果物生成の自動化が行われています。ここでいう成果物とは画像、音声、3D モデル、プログラム等のアセットデータをまとめた物のことを指していて、Android でいえば「apk」、iOS でいえば「ipa」等が当てはまります。

成果物はアセットデータに変更を加える毎に確認できる開発フローを組んでいるプロジェクトが多く、その成果物をビルドするマシンはプロジェクトごとに平均 4-5 台で運用しています。成果物をビルドするマシンすべてに Unity をインストールするのですが、マシンが増えることで Unity のライセンスの管理が大変になるので、KLab の比較的新しいプロジェクトでは Unity Build Server を使った運用をはじめました。

3.1 Unity Build Server とは

今回紹介する Unity Build Server^{*1}とは、Unity 社が開発提供しているライセンス管理サーバーの名称です。これは、Unity のライセンス認証サーバーをローカルネットワーク内に構築して、UnityEditor を Pro ライセンスで Batch Mode 起動させることができるものになります。

また価格も通常の UnityPro よりもお安く提供されている^{*2}ところが魅力です。

^{*1} Unity Build Server 製品ページ

<https://unity.com/ja/products/unity-build-server>

^{*2} 1 ライセンスあたりの価格は、Unity Build Server は¥ 72,000/年

3.2 入門

それでは実際に構築してみましょう。

今回は購入時のマニュアルを参考に CentOS7 の VM 上にライセンスサーバーを構築してみたいと思います。

動作環境

ライセンスサーバーの動作環境は次のとおりになります。

- OS
 - Windows(64-bit versions)
 - * Windows 8
 - * Windows 10
 - * Windows Server 2012
 - * Windows Server 2016
 - Linux(64-bit versions)
 - * Ubuntu 16.04+
 - * Red Hat Enterprise Linux 7.x
 - * CentOS 7.x
- .NET Core 2.1

Unity クライアントは 2019.4.3 LTS 以降のバージョンに対応しています。

サーバーの準備

VM に CentOS7 をインストールする手順は今回は割愛させていただきます。準備した CentOS7 に .NET Core 2.1 をインストールします。標準のパッケージマネージャーでは .NET Core はインストールされないので、追加のプラグインをリスト 3.1 の手順でインストールします。

続いてライセンスサーバーを準備します。Unity Build Server を購入すると `Unity.Licensing.Server.linux-x64-v1.X.X.zip` や `Unity.Licensing.Server.win-x64-v1.X.X.zip` をダウンロードできるようになると思います。執筆時の最新は「1.8.0」ですので、`Unity.Licensing.Server.linux-x64-v1.8.0.zip` をダウンロードして解説を進めます。

ダウンロードしてきた `Unity.Licensing.Server.linux-x64-v1.8.0.zip` を解凍し、マニュアルにしたがって `/opt/UnityLicensingServer` に配置します。リスト 3.2 の手順を進めることで配置します。

▼リスト 3.1 .NET Core 2.1 のインストール

```
# rpm -Uvh \  
https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm  
  
# yum install -y expect dotnet-sdk-2.1 dotnet-runtime-2.1  
読み込んだプラグイン:fastestmirror  
Loading mirror speeds from cached hostfile  
* base: ftp.riken.jp  
* epel: ftp.riken.jp  
  
...(省略)  
  
依存性関連をインストールしました:  
aspnetcore-runtime-2.1.x86_64 0:2.1.28-1      dotnet-host.x86_64 0:5.0.6-1  
dotnet-hostfxr-2.1.x86_64 0:2.1.28-1  
dotnet-runtime-deps-2.1.x86_64 0:2.1.28-1  libicu.x86_64 0:50.2-4.e17_7  
tcl.x86_64 1:8.5.13-8.e17  
完了しました!
```

▼リスト 3.2 UnityLicensingServer の準備

```
# ls  
Unity.Licensing.Server.linux-x64-v1.8.0.zip  
  
# unzip Unity.Licensing.Server.linux-x64-v1.8.0.zip  
Unity.Licensing.Server.linux-x64-v1.8.0.zip  
  
# ls  
Unity.Licensing.Server.linux-x64-v1.8.0  
Unity.Licensing.Server.linux-x64-v1.8.0.zip  
  
# mv Unity.Licensing.Server.linux-x64-v1.8.0 /opt/UnityLicensingServer  
# cd /opt/UnityLicensingServer  
# pwd  
/opt/UnityLicensingServer
```

Unity.Licensing.Server の設定

それでは、ライセンスサーバーの設定を行います。

ターミナルで `./Unity.Licensing.Server setup` とコマンドを入力してリスト 3.3 のように設定をしていきます。項目は全部で 4 項目あり設定内容は次のとおりです。

- サーバー名の設定
- 通信プロトコルの指定 (http or https) (optional)
- ライセンスサーバーの待受ポート
- 管理者アクセス用のホワイトリスト (admin IP whitelist)

セットアップが完了すると `services-config.jsonserver-registration-request.xml` が出力されます。

▼リスト 3.3 UnityLicensingServer のセットアップ

```

# ./Unity.Licensing.Server setup
- - - -
Welcome to Unity Licensing Server setup command line interface.
This setup will help you configure your license server and generate
server registration request file.
- - - -

Enter the server name (e.g. LicenseServer): [sandbox.local]
Do you want the licensing server to use HTTPS? [Y/n] n
List of available network interfaces on this host

- [1] eth1 (00:15:5D:15:92:10) 192.168.1.254
Enter the index number of the network interface which server will operate on: 1
Enter server's listening port number (between 1025 and 65535): [8080]
Add default addresses to the Admin IP Whitelist
(127.0.0.1, ::1, 192.168.1.254)? [Y/n] Y
List of current white-listed admin IP addresses:
- 127.0.0.1
- ::1
- 192.168.1.254
Add an additional admin IP address to the white list? [y/N] N

Generating signing key... Done

Generating server configuration ...Done
Generating services configuration ...Done
Reloading configuration... Done
Generating server registration request file... Done

- - - -
Setup Completed!
- - - -
> Server Registration Request
Please go to Unity ID portal -> Organizations -> [Organization Name]
-> License servers, upload this file and follow the instructions to
receive a compressed license archive file.

File to upload:
/opt/UnityLicensingServer/server-registration-request.xml

> Services Configuration File
Generated services-config.json file must be copied to the following
location on all end user computer in order to enable floating licensing:
- Windows: %PROGRAMDATA%\Unity\config\
- MacOS: /Library/Application Support/Unity/config/
- Linux: /usr/share/unity3d/config/

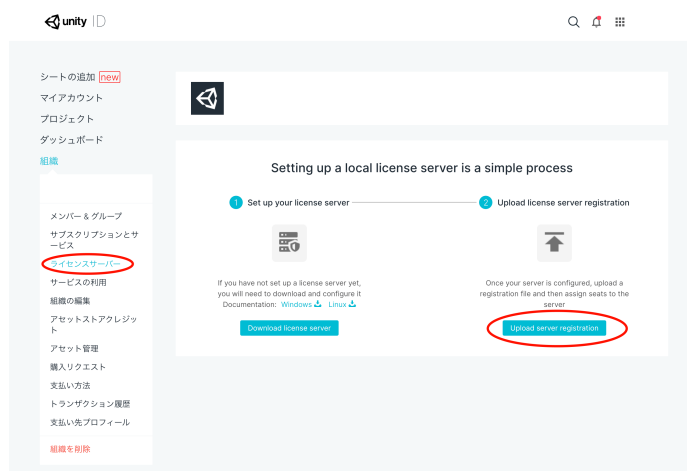
File to copy:
/opt/UnityLicensingServer/services-config.json

WARNING: Please make sure to backup the following directory as it contains
essential data for operating the licensing server:
/root/.config/unity3d/Unity/LicensingServer

```

ライセンスサーバーに Unity ポータルからライセンスの割当

ライセンスサーバーをダウンロードしたページの「Upload server registration」を選択して、セットアップ時に書き出された `server-registration-request.xml` をアップロードします (図 3.1)。



▲図 3.1 unity-portal

ライセンスサーバーが登録できると、ライセンスの割当数が設定できるので必要数を設定します。^{*3}

設定完了後に署名された zip ファイルをダウンロードしてサーバーに読み込ませます。正常に読み込まれるとリスト 3.4 のように表示されます。

▼リスト 3.4 署名されたアーカイブのインポート

```
# ./Unity.Licensing.Server import sandbox.zip
Extracting files...Done
Importing access control file...Done
Validating archive content...Done
  LSD type import detected
Importing server license files...Done
Importing delegation file...Done
List of available toolsets on this server

- [1] sandbox_15668316334479_1 [Unity Pro Build Server]
Enter the index number of the toolset that should be used by default: 1

Successfully imported licensing files. You may run the server.
Please note that if the server is already running, you will have to restart it.
```

Unity Build Server を実行して設定は以上になります。

```
# ./Unity.Licensing.Server
Server Version: 1.8.0.6d5c611
... 以下略
```

^{*3} 割当数を増やすことはできますが、減らすことはできないので注意

ライセンスの要求と確認

それではライセンスサーバーの準備ができたのでライセンスが割り振られるか試してみしましょう。「Unity.Licensing.Server の設定」で出力された `services-config.json` を UnityEditor がインストールされている PC に配置します。配置先は表 3.1 のディレクトリに配置させます。^{*4}

▼表 3.1 services-config.json 設置場所

OS	Path
Windows	%PROGRAMDATA%\Unity\config\
macOS	/Library/Application Support/Unity/config/
Linux	/usr/share/unity3d/config/

ライセンスクライアントから直接ライセンスの要求を行い場合は、表 3.2 に配置されている `Unity.Licensing.Client` を使用して要求を行います。

引数に `--acquire-floating` を使用することでリスト 3.5 のように手動要求を行います。

リスト 3.6 のように引数に `--return-floating` を設定して、取得した TOKEN を入力することでライセンスの返却も行えます。

▼表 3.2 ライセンスクライアント Path

OS	Path
Windows	[UnityEditorDir]\Data\Resources\Licensing\Client\Unity.Licensing.Client.exe
macOS	[UnityEditorDir]/Contents/Frameworks/Unity.Licensing.Client.app/Contents/Resources/Unity.Licensing.Client
Linux	[UnityEditorDir]/Data/Resources/Licensing/Client/Unity.Licensing.Client

▼リスト 3.5 ライセンスを要求

```
C:\>Unity.Licensing.Client --acquire-floating
Client product version: 1.6.2+1.91f8005
Trying to acquire floating license from: 192.168.1.254 ...
License lease Created with token
d36b704c-ba53-41f1-b645-941817494239. Expires: 2021年5月25日 4:22:33
```

^{*4} ディレクトリが存在しない場合は自身で作成してください。

▼リスト 3.6 ライセンスの返却

```
C:\>Unity.Licensing.Client --return-floating d36b704c-ba53-41f1-b645-941817494239
Client product version: 1.6.2+1.91f8005
Trying to return floating license lease
(d36b704c-ba53-41f1-b645-941817494239) to: 192.168.1.254 ...
License lease Returned successfully
```

ライセンスの要求と返却ができることを確認できたら、今後はリスト 3.7 のように Unity を batchmode で起動させることで自動的に認証するようになります。

▼リスト 3.7 ライセンスの返却

```
C:\>Unity.exe -batchmode -executeMethod SampleBuild.Build -projectPath unityproject
```

3.3 小技（コンテナ化）

ここまで紹介したのはマニュアルに記載されているやり方ですが、ここからは記載されていない方法です。これはサポート外の方法になるので、ご利用の際は自己責任でお願いします。

Unity Build Server で重要なのは次の 2 つです。

- ホスト名
- Mac アドレス

この「ホスト名」と「Mac アドレス」を固定することができれば、ライセンスサーバーを別ホストに移動することができます。今回は Docker を使用した移植性の高いライセンスサーバーを構築する小技を紹介します。

ではまず Docker をインストールしたマシンを準備して、DockerImage の作成を行います。

リスト 3.8 に記載している内容の Dockerfile を作成します。

▼リスト 3.8 Dockerfile

```
FROM centos:7
RUN rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm
RUN yum install -y expect dotnet-sdk-2.1 dotnet-runtime-2.1
WORKDIR /opt/UnityLicensingServer
```

続いて Dockerfile をビルドします。

```
$ docker build -t centos:UnityLicenceServer .
[+] Building 48.1s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 241B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/centos:7
=> [auth] library/centos:pull token for registry-1.docker.io
=> [1/3] FROM docker.io/library/centos:7@sha256:0f4ec88e21daf75124b8a9e5ca03c37a5e937e0e108a255
=> => resolve docker.io/library/centos:7@sha256:0f4ec88e21daf75124b8a9e5ca03c37a5e937e0e108a255
=> => sha256:0f4ec88e21daf75124b8a9e5ca03c37a5e937e0e108a255d890492430789b60e 1.20kB / 1.20kB
=> => sha256:e4ca2ed0202e76be184e75fb26d14bf974193579039d5573fb2348664deef76e 529B / 529B
=> => sha256:8652b9f0cb4c0599575e5a003f5906876e10c1ceb2ab9fe1786712dac14a50cf 2.75kB / 2.75kB
=> => sha256:2d473b07cdd5f0912cd6f1a703352c82b512407db6b05b43f2553732b55df3bc 76.10MB / 76.10MB
=> => extracting sha256:2d473b07cdd5f0912cd6f1a703352c82b512407db6b05b43f2553732b55df3bc
=> [2/3] RUN rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.r
=> [3/3] WORKDIR /opt/UnityLicensingServer
=> exporting to image
=> => exporting layers
=> => writing image sha256:62d85178c78479f32fc1150376a02bb53af2d929c890b736b1905f93cbc956f3
=> => naming to docker.io/library/centos:UnityLicenceServer

$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos               UnityLicenceServer 62d85178c784      18 minutes ago    1.83GB
```

image の準備ができたので、リスト 3.9 のような docker-compose.yml を用意します。セットアップ作業はコンテナ内に入り行わなければならないので、sh の実行が終わらないようにコマンドを記述しています。

▼リスト 3.9 docker-compose.yml

```
version: "3.8"
services:
  UnityLicenceServer:
    image: centos:UnityLicenceServer
    restart: always
    ports:
      - 8080:8080
    volumes:
      - ../volumes/.config:/root/.config
      - ../volumes/Unity.Licensing.Server.linux-x64-v1.8.0:/opt/UnityLicensingServer
      - ../volumes/data:/data

    #command: ./Unity.Licensing.Server
    command: /bin/sh -c "while ;; do sleep 10; done"
    hostname: sandbox
    # mac_address: XX:XX:XX:XX:XX:XX
    networks:
      app_net:
        ipv4_address: 172.22.0.2

networks:
  app_net:
    name: app_net
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.22.0.0/24
```

第3章 Unity Build Server の入門と小技の紹介

docker-composeでバックグラウンド起動させて確認します。

```
$ docker-compose up -d
Starting unity-licenceserver_UnityLicenceServer_1 ... done
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                                CREATED
334edeace09   centos:UnityLicenceServer          "/bin/sh -c 'while :...'"           About a
```

docker-compose execコマンドで起動中のコンテナ内に入りセットアップ実行していきます。セットアップ方法は入門編と同じ内容になるので、署名済ファイルをインポートするところまで説明は省略します。

```
$ docker-compose exec UnityLicenceServer bash
[root@sandbox UnityLicensingServer]# ./Unity.Licensing.Server setup
- - - -
Welcome to Unity Licensing Server setup command line interface.
This setup will help you configure your license server and generate
server registration request file.
- - - -

Enter the server name (e.g. LicenseServer): [sandbox.local]
Do you want the licensing server to use HTTPS? [Y/n] n
List of available network interfaces on this host

- [1] eth1 (XX:XX:XX:XX:XX:XX) 172.22.0.2
.....
省略
.....

WARNING: Please make sure to backup the following directory as
it contains essential data for operating the licensing server:
    /root/.config/unity3d/Unity/LicensingServer

[root@sandbox UnityLicensingServer]# ./Unity.Licensing.Server import sandbox.zip
Successfully imported licensing files. You may run the server.
Please note that if the server is already running, you will have to restart it.
[root@sandbox UnityLicensingServer]# exit
```

インポート完了後、セットアップ中に記載されていた NIC の MAC アドレスは今後変更されないように docker-compose.yml のコメントアウトしている # mac_address : XX:XX:XX:XX:XX:XX の部分のコメントアウトを解除してリスト 3.10 のように変更します。

command をリスト 3.10 のように無限ループから Unity Build Server が実行されるように変更してコンテナを起動します。

```
$ docker-compose up -d
Starting unity-licenceserver_UnityLicenceServer_1 ... done
```

▼リスト 3.10 docker-compose.yml

```
version: "3.8"
services:
  UnityLicenceServer:
    image: centos:UnityLicenceServer
    restart: always
    ports:
      - 8080:8080
    volumes:
      - ./volumes/.config:/root/.config
      - ./volumes/Unity.Licensing.Server.linux-x64-v1.8.0:/opt/UnityLicensingServer
      - ./volumes/data:/data

    command: ./Unity.Licensing.Server
    hostname: sandbox
    mac_address: XX:XX:XX:XX:XX:XX
    networks:
      app_net:
        ipv4_address: 172.22.0.2

networks:
  app_net:
    name: app_net
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.22.0.0/24
```

以上で設定は終わりですが、`docker-compose.yml`と `volumes`ディレクトリのバックアップを取ることで、簡単に復元やホストの移動ができるようになります。

3.4 おわりに

UnityPro のライセンスを使用しても `batchmode` で起動はできますが、Unity Build Server の方がコストが抑えられ、さらにライセンスの管理が簡略化されるのでぜひ導入を検討してみたいかがでしょうか。

今回 Unity Build Server について解説することで、自身も理解を深めることができました。また、この記事でコンテナでも運用ができるんですよ、ということを紹介できたと思います。

それではまた機会があれば他の内容も紹介したいと思います。

第4章

CPython itertools を読んでみた

Shunsuke Ito / @fgshun

4.1 はじめに

Python の標準ライブラリの中でもひととき強力なモジュールである `itertools`。Python の C 言語実装である CPython ではどのように実装されているのか、そのコードを読みます。

4.2 PyObject 構造体

PyObject_HEAD マクロ

Python におけるイテレータとは、イテレータプロトコルに対応したクラスのインスタンスです。そしてイテレータプロトコルへの対応とは、反復処理における次の値を返す `__next__` 関数と自身を返す `__iter__` 関数を備えることです。

このようなクラスはこの2つのメソッドをもつという特徴こそありますが、クラスには違いありません。また、このようなクラスのインスタンスもまたインスタンスに違いありません。加えて、Python におけるインスタンスとはオブジェクトです。そして、オブジェクトを表すために CPython で用いられている構造体が PyObject 構造体です。

たとえば、あらたなインスタンス `SpamObject` を作るにあたって PyObject 構造体と互換性のある構造体を定義するには PyObject_HEAD マクロを用いて次のように記述します。

▼リスト 4.1 あらたな PyObject 構造体を作る

```
typedef struct {
    PyObject_HEAD
    /* 追加のメンバーが欲しいのであればこの後ろに加える */
} SpamObject;
```

PyObject_HEAD は Python 3.9.5 では単に PyObject ob_base; と展開されるマクロです。CPython ではすべての Python オブジェクトを表現する構造体は同様の実装をなされており PyObject へキャストすることが可能です。

repeat にみるインスタンスオブジェクト

さっそく CPython の itertools モジュールのコードにてインスタンスの実装の実例を見てみます。対象は itertools の中でも、もっとも単純な動きをする repeat^{*1}を選びました。同じ値を返し続けるイテレータです。times オプション引数が与えられた場合はその回数だけを繰り返し、与えられない時は無限に繰り返し続けます。

その実装は Modules/itertoolsmodule.c を repeatobject で検索することで見つかります。

▼リスト 4.2 repeatobject 構造体

```
typedef struct {
    PyObject_HEAD
    PyObject *element;
    Py_ssize_t cnt;
} repeatobject;
```

特別なことは行われていないことがわかります。PyObject_HEAD から始まり、繰り返す値を保持するためのメンバー element があり、繰り返す回数を保持するメンバー cnt がある、という作りになっています。

4.3 PyTypeObject 構造体

PyObject 構造体と PyTypeObject 構造体

では、クラスの方はどのような実装になっているのでしょうか？ Python ではクラスもオブジェクトです。しかし、クラスにはクラス名、インスタンスのバイトサイズ、コンストラクタ、デストラクタ、メソッド、デスク립タなどなど、多様な追加情報があります。これらの追加情報をもつオブジェクトを表すためにに用いられる構造体が PyObject 構造体の一種である PyTypeObject 構造体です。

repeat にみるクラスオブジェクト

itertools.repeat のクラスの実装を見てみます。

^{*1} <https://docs.python.org/ja/3/library/itertools.html#itertools.repeat>

▼リスト 4.3 repeat_type 構造体

```
static PyTypeObject repeat_type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "itertools.repeat",          /* tp_name */
    // 以下、イテレータ作りに関連するメンバーを抜粋
    PyObject_SelfIter,          /* tp_iter */
    (iternextfunc)repeat_next,  /* tp_iternext */
};
```

PyVarObject_HEAD_INIT マクロから始まり、さまざまなメンバーが埋められています。イテレータのための処理に注目するなら `__iter__` に相当する `tp_iter` と `__next__` に相当する `iternext` を見ることになります*2。

4.4 repeat にみるイテレータプロトコル

PyObject_SelfIter

では、CPython では `__iter__` に相当する処理をどのように実装しているのかについて読んでいきます。イテレータプロトコルではイテレータは `__iter__` から自身を返すこと、と定められています。そして `repeat_type.tp_iter` メンバーには `PyObject_SelfIter` といういかにもそれらしい名の関数がおさまっています。実装は `Objects/object.c` を探すと見つかります。

▼リスト 4.4 PyObject_SelfIter 関数

```
PyObject *
PyObject_SelfIter(PyObject *obj)
{
    Py_INCREF(obj);
    return obj;
}
```

`def __iter__(self): return self` 相当の関数であり、Python で記述したものと同様に簡素な作りをしています。それでも CPython ならではの要素がないわけではありません。それが参照カウンタの操作です。CPython では参照カウンタによるガベージコレクションが採用されています。そのため CPython 用のコードを書く時はカウンタを適切に操作しなければなりません。`__iter__` を呼び出してイテレータを受け取った側の立場で考えてみると、操作中のうけとったイテレータの参照カウンタが外部要因によって 0 になってオブジェクトの開放が突如始まるのは不都合です。そのためにイテレータを渡す側は参照カウンタを 1 増やしてカウンタを減らす義務とともに自身への参照を返しています。

*2 その他、他のオブジェクトへの参照をもつことやその対処方法をガベージコレクションに伝えるため `tp_flags`, `tp_traverse`, `tp_dealloc`, `tp_free` に手が入っています。イテレータに限った話ではないためここでは触れません。

repeat_next

`__next__` に相当する処理はどのようになっているのでしょうか。読み進めます。

▼リスト 4.5 repeat_next 関数

```
static PyObject *
repeat_next(repeatobject *ro)
{
    if (ro->cnt == 0)
        return NULL;
    if (ro->cnt > 0)
        ro->cnt--;
    Py_INCREF(ro->element);
    return ro->element;
}
```

`cnt` はコンストラクタで指定した繰り返し回数で初期化されますが、指定されなかった場合は-1で初期化されます。このためここでの処理は次のようになります。まず繰り返し回数に残り0である場合、NULLを返すことでこのイテレータから返す値がもうないことを示します。それ以外の場合、残回数が正の値であれば減らし、-1であれば何もせず、繰り返し対象オブジェクトの参照カウンタを増やして返します。

無限回繰り返すフラグとして負の値をうまく用いている工夫と、参照カウンタ操作が目を引きます。

4.5 その他のイテレータ

chain

他の `itertools` の処理も見ていきます。chain^{*3}です。

`chain('AB', 'CD')` のようにイテレート可能オブジェクトを複数渡すと A B C D とつなげて返してくるイテレータを作成するものです。chain のための `chainobject` 構造体はリスト 4.6 のようになっています。

▼リスト 4.6 chainobject

```
typedef struct {
    PyObject_HEAD
    PyObject *source; /* Iterator over input iterables */
    PyObject *active; /* Currently running input iterator */
} chainobject;
```

繋ぐイテレート可能オブジェクトたちをひとつずつ返すイテレータ `source` と、そこから取り出されて今操作中であるイテレータ `active` を保持するようになっていることがわかります。そして次の値を取り出す `__iter__` に相当する処理の前半はリスト 4.7 です。

*3 <https://docs.python.org/ja/3/library/itertools.html#itertools.tee>

▼リスト 4.7 chain_next 関数 前半

```
static PyObject *
chain_next(chainobject *lz)
{
    PyObject *item;
    while (lz->source != NULL) {
        if (lz->active == NULL) {
            PyObject *iterable = PyIter_Next(lz->source);
            if (iterable == NULL) {
                Py_CLEAR(lz->source);
                return NULL; /* no more input sources */
            }
            lz->active = PyObject_GetIter(iterable);
            Py_DECREF(iterable);
            if (lz->active == NULL) {
                Py_CLEAR(lz->source);
                return NULL; /* input not iterable */
            }
        }
    }
}
```

前半は今操作中の active イテレータがない時に source から取り出す処理です。source の内容が尽きているときには PyIter_Next が NULL を返すので、source の参照カウントを減らしつつ NULL とする Py_CLEAR が呼び出されます。このため、ふたたび chain_next が呼び出された時には、この while ループには入ってこない作りになっています。source の内容がまだ残っているときに取り出された値は次に繋ぐイテレート可能オブジェクトであるはずなので、確認するために PyObject_GetIter が使われています。

このタイミングで iterable 変数にあるオブジェクトの出番は終わりであるため Py_DECREF で参照カウントが減らされています*4。

▼リスト 4.8 chain_next 関数 後半

```
    item = (*Py_TYPE(lz->active)->tp_iternext)(lz->active);
    if (item != NULL)
        return item;
    if (PyErr_Occurred()) {
        if (PyErr_ExceptionMatches(PyExc_StopIteration))
            PyErr_Clear();
        else
            return NULL; /* input raised an exception */
    }
    Py_CLEAR(lz->active);
}
/* Everything had been consumed already. */
return NULL;
}
```

後半は active イテレータから次の値を取り出す処理です。PyIter_Next を使わずに直接 tp_iternext を呼んでいるのは、速度を限界まで出すために済んでいるエラーチェックをはぶくためでしょうか。ともかく tp_iternext により次の値が得られます。次の値がない場合、すなわち active イテレータの要素が尽きている場合は前半の source から次の繋ぐイテレータを取り出す処理に戻ります。

*4 PyObject_GetIter は参照カウントが 1 増えた対象そのものを返すことがあります。その場合でも参照カウントを 1 減らせばつじつまが合うことには変わりありません。

tee

より複雑なことをしているイテレータである `tee`^{*5} を読んでいきます。ひとつの大元となるイテレータから同じ結果を返す独立した複数のイテレータを作り出すものです。公式ドキュメントでは説明のため同等の動きをする `collections.deque` を用いた Python コードリスト 4.9 を掲示しつつ、「実際の実装はより複雑で、下層の FIFO キューをひとつしか使いません」とあります。

▼リスト 4.9 tee と同等の Python コード

```
def tee(iterable, n=2):
    it = iter(iterable)
    deque = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:          # when the local deque is empty
                try:
                    newval = next(it) # fetch a new value and
                except StopIteration:
                    return
            for d in deque:          # load it to all the deque
                d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deque)
```

さて、実際にはどのようなになっているのでしょうか？

▼リスト 4.10 teedataobject および teeobject

```
#define LINKCELLS 57

typedef struct {
    PyObject_HEAD
    PyObject *it;
    int numread;          /* 0 <= numread <= LINKCELLS */
    int running;
    PyObject *nextlink;
    PyObject *(values[LINKCELLS]);
} teedataobject;

typedef struct {
    PyObject_HEAD
    teedataobject *dataobj;
    int index;           /* 0 <= index <= LINKCELLS */
    PyObject *weakreflist;
} teeobject;
```

LINKCELLS 個、CPython 3.9.5 では 57 個の `PyObject*` をひとまとめにした連結リストの亜種である `teedataobject` がデータのキャッシュに用いられていました。この `teedataobject` を複数の `teeobject` で共有します。ここで `it` は大元となったイテレータ、`numread` は先読み済みのデータの件数で、`nextlink` はつながっている次の `teedataobject`

^{*5} <https://docs.python.org/ja/3/library/itertools.html#itertools.tee>

で、index はイテレート済みのデータの件数となっています。この teedataobject の操作は teedataobject_getitem 関数で行われます。

▼リスト 4.11 teedataobject_getitem 関数

```
static PyObject *
teedataobject_getitem(teedataobject *tdo, int i)
{
    PyObject *value;

    assert(i < LINKCELLS);
    if (i < tdo->numread)
        value = tdo->values[i];
    else {
        /* this is the lead iterator, so fetch more data */
        assert(i == tdo->numread);
        if (tdo->running) {
            PyErr_SetString(PyExc_RuntimeError,
                "cannot re-enter the tee iterator");
            return NULL;
        }
        tdo->running = 1;
        value = PyIter_Next(tdo->it);
        tdo->running = 0;
        if (value == NULL)
            return NULL;
        tdo->numread++;
        tdo->values[i] = value;
    }
    Py_INCREF(value);
    return value;
}
```

teedataobject_getitem 関数は tee イテレータから次の値を取り出す際に用いられる tee_next 関数より引数 i に index を渡す形で呼び出されます。

▼リスト 4.12 tee_next 関数

```
static PyObject *
tee_next(teeobject *to)
{
    PyObject *value, *link;

    if (to->index >= LINKCELLS) {
        link = teedataobject_jumplink(to->dataobj);
        if (link == NULL)
            return NULL;
        Py_SETREF(to->dataobj, (teedataobject *)link);
        to->index = 0;
    }
    value = teedataobject_getitem(to->dataobj, to->index);
    if (value == NULL)
        return NULL;
    to->index++;
    return value;
}
```

index が numread と等しい場合は大元のイテレータ it より値を取り出して values 配列に格納しつつ返す処理となっています。index が numread 未満の場合は分割された他の teeobject により it から values 配列への値の格納済みです。この状態からは tdo->values[i] を返すだけの軽量の処理となっています。

tee_next 関数では teedataobject_getitem 関数を呼ぶ前に LINKCELLS 個の読み出しが終わっていて values 配列にあきがない場合に、次の teedataobject があれば返し、なければあらたな teedataobject を作って古い teedataobject とをつないでから返す teedataobject_jumplink が呼び出されるようになっています。

▼リスト 4.13 teedataobject_jumplink 関数

```
static PyObject *
teedataobject_jumplink(teedataobject *tdo)
{
    if (tdo->nextlink == NULL)
        tdo->nextlink = teedataobject_newinternal(tdo->it);
    Py_XINCRREF(tdo->nextlink);
    return tdo->nextlink;
}
```

4.6 終わりに

強力なライブラリゆえに理解の及ばない難解なコードだらけであろうという先入観がありました。itertools でしたが、平易な、参照カウンタの操作やイテレータの扱い方をゼロから学べるレベルのコードも多くみられることがわかりました。読む前からあきらめずに普段世話になっているライブラリのコードを読んでみると、面白い発見があるかもしれません。

第5章

AR マーカーを使った画像処理でアナログ値の読み取り

Toshifumi Umezawa

5.1 はじめに

この発端は土壌の pH を読み取ろうとした時のことでした。

ph を読み取るモジュールというものは存在しているのですが、配置場所として屋外を想定しているので防水や給電方法を考える必要があることや、複数地点で計測しようとするとその数だけモジュールが必要になって資金面でも辛くなる^{*1}ことが気がかりでした。

そのため、監視カメラのように一箇所にカメラを配置して、撮影した画像から値を読み取るという方法を考えました。実際には角度があると読み取り結果がずれそうなことと、光の反射などでうまく読めないことが容易に想像できたので、カメラをバーコードリーダーのようにかざして読み取る方式を試してみました。

既製品や似たようなことをやっている記事と比較すると、カメラを計器の目の前に固定しないという点が大きな違いになります。そのため、読み取り対象の計器の判別と読み取り領域の固定のために AR マーカーを活用しました。すでに固定カメラを使って似たような値読み取りをしている場合でも、AR マーカーを使う部分を導入すれば位置がずれた場合の補正にも使えます。

5.2 状況設定と戦略

今回はこの温湿度計の読み取りを試してみます。

^{*1} ph 測定モジュールの例: <https://akizukidenshi.com/catalog/g/gM-12547/> 3500 円



▲ 図 5.1 温湿度計

AR マーカーを基準に固定領域を切り出し、針を検出し、角度から温湿度をそれぞれ検出してみます。環境は Python3.9.4 です。

5.3 AR マーカーを使った処理

AR マーカーを使うと、カメラで撮影した画像に映るマーカーを判別した上で、3次元空間上の位置や姿勢を推定できます。

今回は **ArUco** という、OpenCV に含まれる AR マーカーを扱うライブラリを使用します。pip で `opencv-contrib-python` をインストールするのですが、すでに `opencv` が入っていると ArUco が含まれないように追加でインストールできないようなので、一度消して入れ直す必要があります。

画像生成

AR マーカーは事前に{id: 画像}となる辞書式のデータが必要なのですが、ArUco では AR マーカーのセットが用意されており、特にこだわりがなければこれを利用します。

マーカー内のタイルの数が $n \times n$ で種類数を m となる `DICT_nXn_m` がいくつか用意されており、こだわりがなければタイル数が少なく種類数も少ない `DICT_nXn_m` を使うとよいです。例として、 5×5 のサイズの 50 種類が登録された中から `id=25` となるマーカーはリスト 5.1 のようなコードで取り出せます。

ArUco でマーカーを生成したら、画像として出力して印刷して任意の場所に貼り付けて使用します。

▼リスト 5.1 マーカーの生成

```
from cv2 import aruco
aruco_dict = aruco.Dictionary_get(aruco.DICT_5X5_50)
img = aruco.drawMarker(aruco_dict, 0, 25)
```

読み取り

事前に用意した辞書式のデータを用いることでマーカーを検出できます。

リスト 5.2 のように、カメラで取得した画像と照らし合わせることで、登録されたマーカーの画像上での位置とそのマーカーの id を複数同時に取得できます。

▼リスト 5.2 マーカーの検出

```
corners, ids, rejectedImgPoints = aruco.detectMarkers(frame, aruco.DICT_5X5_50)
```

固定領域の切り取り

検出したマーカーは四隅の位置が検出されているので、今回の針検出のためにマーカーを基準とした固定の領域を切り出します。

この記事では、マーカーの中心から上下左右の 4 辺への距離を基準として、それぞれの方向に何倍かした距離という形で領域を切り出しています。今回は `rect` という変数内に全方向 4.5 倍として設定しておき、切り取った領域を (500, 500) の画像サイズに固定しています。厳密には撮影する角度によって切り取り範囲は少しずつ異なりますが、正面から撮影している場合はほぼ同じ領域を切り出せます。

▼リスト 5.3 領域の切り出し

```
def cutImage(frame: ndarray, corner: ndarray, rect: Rect) -> ndarray:
    # 中心点と上下左右辺の中点の 5 点
    c = np.mean(corner, axis=0)
    t = np.mean(np.concatenate([[corner[0]], [corner[1]]]), axis=0)
    r = np.mean(np.concatenate([[corner[1]], [corner[2]]]), axis=0)
    b = np.mean(np.concatenate([[corner[2]], [corner[3]]]), axis=0)
    l = np.mean(np.concatenate([[corner[3]], [corner[0]]]), axis=0)
    parsed_center_rect = np.float32([
        c + (t - c) * rect.t + (l - c) * rect.l,
        c + (t - c) * rect.t + (r - c) * rect.r,
        c + (b - c) * rect.b + (l - c) * rect.l,
        c + (b - c) * rect.b + (r - c) * rect.r,
    ])
    return getParsedFrame(frame, parsed_center_rect, (500, 500))

def getParsedFrame(frame, rect, output_size):
    width = output_size[0]
    height = output_size[1]
    parsed_frame = cv2.warpPerspective(
        frame,
        cv2.getPerspectiveTransform(
            np.float32([rect[0], rect[1], rect[2], rect[3]]),
            np.float32([[0, 0], [width, 0], [0, height], [width, height]])
        ),
        output_size,
        cv2.INTER_CUBIC,
    )
    return parsed_frame
```

この処理によって、斜めから撮影したり回転していても毎回同じ位置・向きに補正できます。

姿勢推定

撮影角度が急だと針の位置がずれたり針が枠に隠れて見えなくなったりするので、マーカー正面との角度が小さい時の画像のみ読み取るようにします。そのために、カメラから見た AR マーカーの位置と回転方向を考慮して、AR マーカーの正面からどれだけ傾いた位置から撮影しているかを推定します。

カメラパラメータの推定

空間上の姿勢推定のために**カメラパラメータ**（使っているカメラの内部パラメータ）が必要となります。今回は <https://qiita.com/ReoNagai/items/5da95dea149c66ddbddd> のプログラムをそのまま使わせてもらうことで取得しましたが、どういう処理をしているのかざっくり解説を入れていきます。

▼ カメラパラメータ推定

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import cv2
import matplotlib.pyplot as plt
import numpy as np

square_size = 2.2      # 正方形の1辺のサイズ [cm]
pattern_size = (7, 7) # 交差ポイントの数

reference_img = 40 # 参照画像の枚数

# チェスボード (X,Y,Z) 座標の指定 (Z=0)
pattern_points = np.zeros( (np.prod(pattern_size), 3), np.float32 )
pattern_points[:, :2] = np.indices(pattern_size).T.reshape(-1, 2)
pattern_points *= square_size
objpoints = []
imgpoints = []

capture = cv2.VideoCapture(0)

while len(objpoints) < reference_img:
# 画像の取得
    ret, img = capture.read()
    height = img.shape[0]
    width = img.shape[1]

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# チェスボードのコーナーを検出
    ret, corner = cv2.findChessboardCorners(gray, pattern_size)
# コーナーがあれば
    if ret == True:
        print("detected coner!")
        print(str(len(objpoints)+1) + "/" + str(reference_img))
        term = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT, 30, 0.1)
        cv2.cornerSubPix(gray, corner, (5,5), (-1,-1), term)
```

```
# append メソッド：リストの最後に因数のオブジェクトを追加
imgpoints.append(corner.reshape(-1, 2))
objpoints.append(pattern_points)

cv2.imshow('image', img)
# 毎回判定するから 200 ms 待つ。遅延するのはココ
if cv2.waitKey(200) & 0xFF == ord('q'):
    break

print("calculating camera parameter...")
# 内部パラメータを計算
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
                                                  gray.shape[::-1], None, None)

# 計算結果を保存
np.save("mtx", mtx) # カメラ行列
np.save("dist", dist.ravel()) # 歪みパラメータ
# 計算結果を表示
print("RMS = ", ret)
print("mtx = \n", mtx)
print("dist = ", dist.ravel())
```

このプログラムでは、 8×8 のチェスボード（白黒タイル）の絵を複数回検出して、3次元空間上の点が画像上のどの位置に写っているかの対応からカメラパラメータを推定しています。

`cv2.findChessboardCorners` で二値化画像からチェスボードの交差点を検出します。 8×8 マスの場合は 7×7 個の交差点が得られます。

`cv2.cornerSubPix` で検出した点の位置の精度を向上させています。これは、検出した点位置を中心とした領域内からより適切なコーナー位置になるように繰り返し処理によって精度を上げてくれます。パラメータ設定は (`cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT, 30, 0.1`) の場合、30 回の試行か差分が 0.1 以下になると終了します。

`cv2.calibrateCamera` で実際の三次元空間上の点の位置と画像上の交差点の位置の組み合わせを渡すことで、カメラパラメータが得られます。三次元空間上の交差点の位置はカメラの位置や角度によらず、 $z = 0$ として xy 平面上の格子点状に並んでいる固定の点として計算できます*2。

cornerSubPix は必要？

OpenCV の `findChessboardCorners` のソースコード*3には次のコードが含まれており、この精度で充分であれば `SubPix` をあえて呼ぶ必要はないようです。

```
cv::cornerSubPix(img, out_corners, Size(2, 2), Size(-1,-1), cv::TermCriteria(TermCriteria::EPS + TermCriteria::MAX_ITER, 15, 0.1));
```

*2 チェスボードを原点とした空間上でカメラが動いていると捉えると固定座標となるため

スマホ利用時のカメラパラメータ

解像度が変わるとカメラパラメータも変わります。カメラパラメータの調整は動画、それ以外の処理は撮影しておいたテスト用の画像を使って検証していたところ、両者の解像度の違いによってうまく角度の計算がされず一日が潰れました。

マーカーの姿勢推定

読み取った AR マーカーの画像上での四隅の位置と、先ほど計算したカメラパラメータを組み合わせることで、カメラから見たマーカーの位置・回転を推定したものが得られます。これを使ってリスト 5.4 のように行列・ベクトルの計算を行うと、マーカーに向けられたカメラの角度を計算できます。

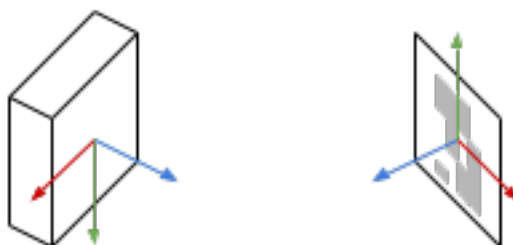
▼リスト 5.4 AR マーカーを撮影する角度の計算

```
rvec, tvec, _ = aruco.estimatePoseSingleMarkers(
    corner, marker_size, camera_matrix, distortion_coeff)

R = cv2.Rodrigues(rvec)[0] # 回転行列に変換し、マーカー1つ分の結果を取得
v = R @ np.array([0,0,-1]) # マーカー座標系での方向をカメラ座標系での方向に回転
t = np.squeeze(tvec)      # 余分な次元を削除
dot = v @ (t / np.linalg.norm(t)) # v と t の単位ベクトルの内積 (なす角の cos)
rad = math.acos(dot)      # radian に変換
deg = np.rad2deg(rad)     # degree に変換
return deg
```

`aruco.estimatePoseSingleMarkers`で事前に用意したカメラパラメータを渡すことで、**マーカー座標系**から**カメラ座標系**へ変換する**回転ベクトル**と**並進ベクトル**が得られます。2つの座標系が出てきましたが、これは同じ空間上の点でも図 5.2 のようにマーカー・カメラを基準としたそれぞれの座標系について別の値として表現されるためです。

*3 <https://github.com/opencv/opencv/blob/4.5.2/modules/calib3d/src/calibinit.cpp#L683-L684>



▲図 5.2 2つの座標系

カメラ座標系における座標 (x_c, y_c, z_c) は、マーカー座標系における座標 (x_m, y_m, z_m) と、マーカー座標系からカメラ座標系に変換する回転行列 R と並進ベクトル t を使って式のように表せます。

$$\begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = R \begin{pmatrix} x_m \\ y_m \\ z_m \end{pmatrix} + t$$

マーカーの座標はマーカー座標系では $(0, 0, 0)$ のため、並進ベクトルはそのままカメラ座標系での AR マーカーの座標になります。

カメラ座標系における AR マーカーの垂直下方向のベクトル v (マーカー座標系における z 軸方向のベクトルを回転行列にかけたベクトル) と、カメラから AR マーカー中央へのベクトル t (並進ベクトル) の 2 つのベクトルの角度を求めます。

演算子としての '@'

numpy を使ったコードではよく `np.dot(a, b)` を用いて行列式の計算が行われていますが、Python3.5 から '@' 演算子で行列式が計算できるようになっています。

AR マーカーの処理結果

入力画像に対する切り取り後の画像とマーカーとの角度は図 5.3 のようになりました。

元画像	切り取り後	マーカーとの角度
		19.11556518
		5.298622298
		9.077650041
		8.04543845

▲図 5.3 切り取りの比較

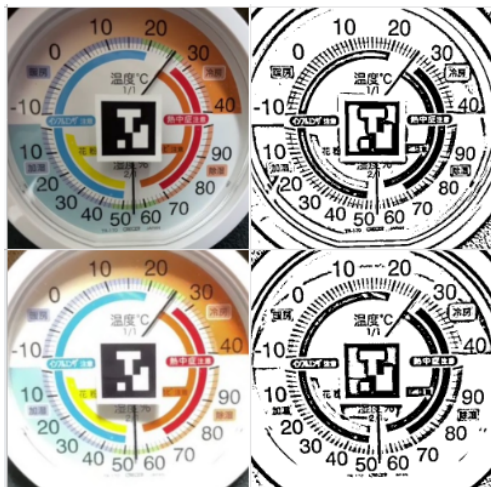
5.4 画像処理

固定の向き、大きさに補正されたため、ここから針の検出をしていきます。

二値化

以降の処理のために二値化します。二値化の方法にはいくつか選択肢がありますが、今回は白背景に黒い針がある状態のため、単純に輝度による分類をします。

`cv2.adaptiveThreshold`を使うと閾値を自動で調整してくれるため、図 5.4 のように明るさの違う画像であってもほぼ同じように二値化を行ってくれます。

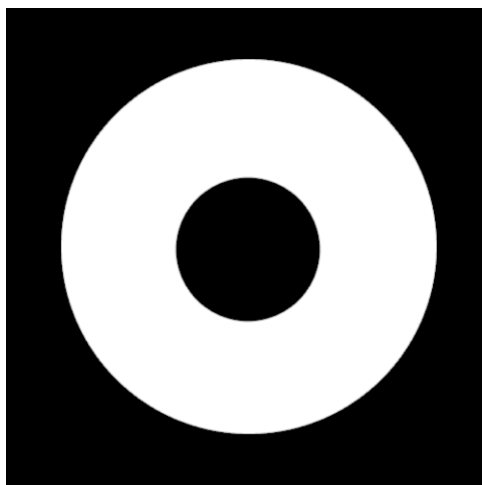


▲ 図 5.4 二値化の適用例

エッジ検出

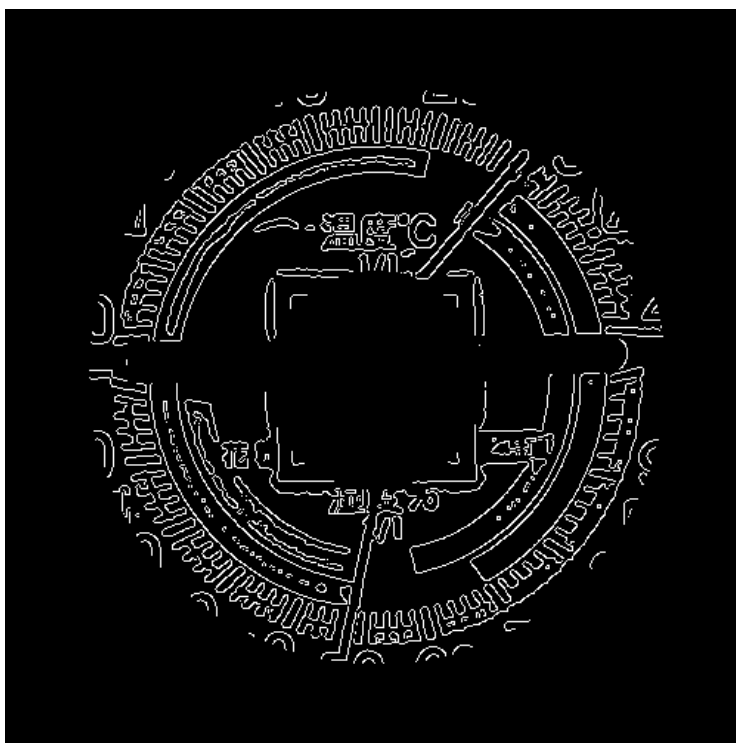
この後に線分を検出する処理をするので、その前処理としてエッジの検出をしておきます。これを行うことで、色がベタ塗りの箇所などについても内部の点を除外して枠だけを残すことで、ベタ塗りの領域内に内包される線分が多数検出されなくなります。

また、これだけだと AR マーカーのセルの境界の線やその他の誤検出となりやすい箇所もあるため、これらの領域を除外するための図 5.5 のようなマスク画像も用意しておきます。



▲ 図 5.5 マスク画像

cv2.Canny によってエッジ検出をしたうえで、マスクを適用して欲しい部分の情報だけ抽出した結果は図 5.6 のようになります。



▲ 図 5.6 エッジ検出

ハフ変換

ハフ変換は直線などを検出するアルゴリズムです。仕組みとしては単純で、点ひとつひとつについて所属している直線に投票していき、票数の多い直線は多くの点を通っていると判定するような処理になっています。

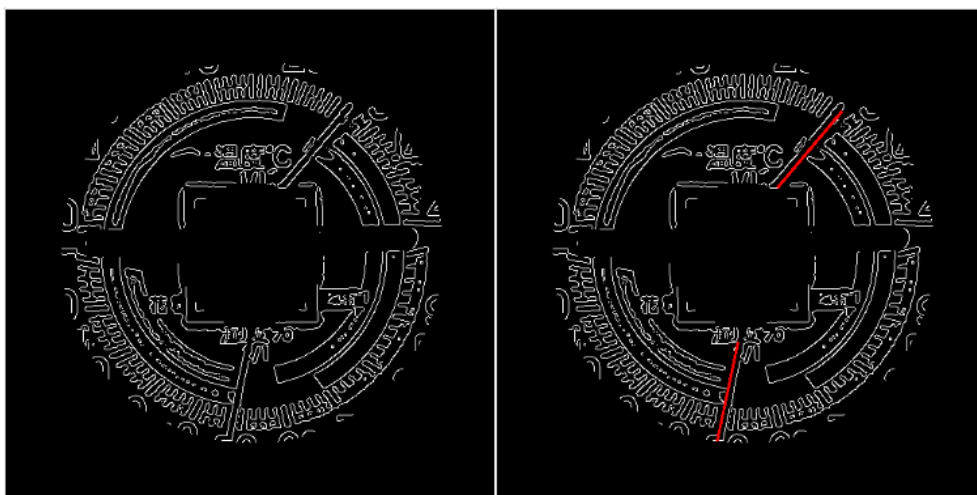
通常のハフ変換だと直線が検出されますが、今回は確率的ハフ変換を用いて直線ではなく線分を検出します。

いくつかパラメータを試して調整した結果、リスト 5.5 のように設定しました。引数の 4~6 番目で検出する線分の最小の長さなどを指定する必要がありますが、今回は画像サイズを 500×500 で固定しているためこのパラメータは固定で使いまわせます。

▼リスト 5.5 ハフ変換のパラメータ例

```
hough_lines = cv2.HoughLinesP(  
    frame,  
    rho=1,  
    theta=np.pi/180,  
    threshold=100,  
    minLineLength=75,  
    maxLineGap=25  
)
```

検出した線分を描画すると図 5.7 のようになります。



▲ 図 5.7 線分の検出

5.5 値の検証

線分の検出はできましたが、見つかった線分が本当に針を検出したのか判別しないといけません。この温湿度計の構造を見る限り、以下が成り立つと思われるのでこれを検証していきます。

- 2つのグループに分類したとき、同じグループの線分の位置にばらつきがない
- 片方のグループは上半分に、もう片方は下半分にある
- どちらのグループの線分も中心付近を通る

前処理

図 5.10 のように明らかに外れた位置に検出された線分は事前に除外しておきます。リスト 5.6 のようにして画像の中心から垂線の足までの距離を計算し、一定の値以上になるものを取り除きます。

▼リスト 5.6 線分が画像中心からどれだけ離れているか計算

```

cx = define.FRAME_WIDTH/2
cy = define.FRAME_HEIGHT/2
c = np.array([define.FRAME_WIDTH/2, define.FRAME_HEIGHT/2]) # 画像の中心点 C
p1 = line[:2] # 端点 1 (x, y, *, *)
p2 = line[2:] # 端点 2 (*, *, x, y)
v = (p2 - p1) / np.linalg.norm(p2 - p1) # P1->P2 の単位ベクトル v
h = p1 + v * ((c - p1) @ v) # C から線分を下ろした垂線 H の位置を ∠CP1H から計算
d = np.linalg.norm(c - h) # 線分 CH の長さ

```

また、このタイミングで線分の向きを画像中心から近いほう→遠いほうとなるように揃え、`np.squeeze` で余分な次元をなくしておきます。

線の分類

検出された線分が針であれば、温度と湿度の 2 本が検出されているはずです。線分は始点と終点の x, y 座標で 4 次元の情報になっていますが、これをそのまま `k-means` 法で 2 種類に分類します。

2 本の針の線分が正しく検出されているなら、ほとんどバラツキのない 2 つのグループに分けられるはずです。

▼リスト 5.7 `k-means` 法による分類

```

# 2つに分類できるか
if hough_lines is None:
    return {ReadValueType.Error: 'no line detected'}

```

```
if len(hough_lines) <= 1:
    return {ReadValueType.Error: 'line count not enough 2'}
compactness, labels, centers = cv2.kmeans(
    data=hough_lines.astype(np.float32),
    K=2,
    bestLabels=None,
    criteria=(cv2.TERM_CRITERIA_MAX_ITER, 5, 1.0),
    attempts=5,
    flags=cv2.KMEANS_RANDOM_CENTERS,
)
label_and_lines = zip(labels.flatten(), hough_lines)
label_lines = [
    [x for label, x in label_and_lines if label == 0],
    [x for label, x in label_and_lines if label == 1],
]

# 分類した 2 グループがほぼ同じ線のみ検出できているか
std_border = 10
label_stds = np.array(
    [np.std(lines, axis=0) for lines in label_lines if len(lines) >= 2], dtype=object)
if not np.all(label_stds.flatten() < std_border):
    return {ReadValueType.Error: 'maybe not 2 groups'}
```

上下 2 つの針

2 本の線分のグループに分けられることはわかりましたが、もしかしたら片方の針を表す線分だけで 2 グループ作っているかもしれませんし、まったく別の箇所を線分として認識しているかもしれません。

今回の計器の構造上、上半分と下半分にそれぞれ針があるはずなのでこれを検証します。

▼リスト 5.8 温度と湿度に分けられるか

```
# 上半分と下半分に分けられるか
label_types = {0: None, 1: None}
for i, line in enumerate(centers):
    half_h = define.FRAME_HEIGHT/2
    line_ys = line[1::2]
    if np.all(line_ys > half_h):
        label_types[i] = ReadValueType.Humid
    if np.all(line_ys < half_h):
        label_types[i] = ReadValueType.Temperature
if label_types[0] is None or label_types[1] is None or label_types[0] == label_types[1]:
    return {ReadValueType.Error: 'cant detect each temperature or humid'}

type_line = {
    ReadValueType.Humid:
        centers[0] if label_types[0] == ReadValueType.Humid else centers[1],
    ReadValueType.Temperature:
        centers[0] if label_types[0] == ReadValueType.Temperature else centers[1],
}
```

本当に針か

上下の領域にそれぞれ 1 本ずつ線分があることはわかりましたが、たまたま針とまったく関係のない場所の線分が上下の領域にそれぞれ 1 本ずつあっただけかもしれません。針は軸を中心に回っているの、2 つの線分がそれぞれ軸周辺を通るかどうかを検証します。

すでに前処理で中心付近を通るものだけに絞っていますが、今回は温度と湿度でそれぞれ回転軸の位置が上下にずれていることを利用してさらに厳密に検証しています。処理内容はリスト 5.6 と同様です。

値を求める

これらを満たせば、よほど偶然に検証をすり抜ける変な場所を検出しない限り針を検出できていると思います。もし光の当たり方などで間違った値を読み取ってしまった場合、何度か読み取って外れ値をとり除けば問題ないと思われます。上側が温度、下側が湿度を指している線分なので、それらの角度から値を逆算します。

リスト 5.9 のように、角度の基準となる方向ベクトル、正の角度となる方向ベクトル、値の計算に使う角度と読み取り値のセット 2 つを使って計算します。

▼リスト 5.9 値の取得

```

type_params = {
    ReadValueType.Humid: {
        'base_dir': [0, 1],
        'positive_dir': [1, 0],
        'v1': [0, 53],
        'v2': [90, 94]
    },
    ReadValueType.Temperature: {
        'base_dir': [0, -1],
        'positive_dir': [1, 0],
        'v1': [0, 15],
        'v2': [90, 39]
    },
}
ret = {}
for rvtype, line in type_line.items():
    params = type_params[rvtype]
    base_dir = np.array(params['base_dir']) # 針の可動域の中心方向
    positive_dir = np.array(params['positive_dir']) # 針の角度の正の方向
    v1 = np.array(params['v1']) # 角度と値のセット v1
    v2 = np.array(params['v2']) # 角度と値のセット v2
    v = (v2 - v1) / np.linalg.norm(v2 - v1) # 値計算用の単位ベクトル
    p1 = line[:2]
    p2 = line[2:]
    p = (p2 - p1) / np.linalg.norm((p2 - p1)) # 針の方向ベクトル
    sign = 1 if positive_dir @ p >= 0 else -1 # 正の角度かどうか
    deg = np.rad2deg(math.acos(p @ base_dir)) * sign # 内積から角度へ
    value = (v1 + v * (deg - v1[0]))[1] # 角度から値を計算
    ret[rvtype] = value

```

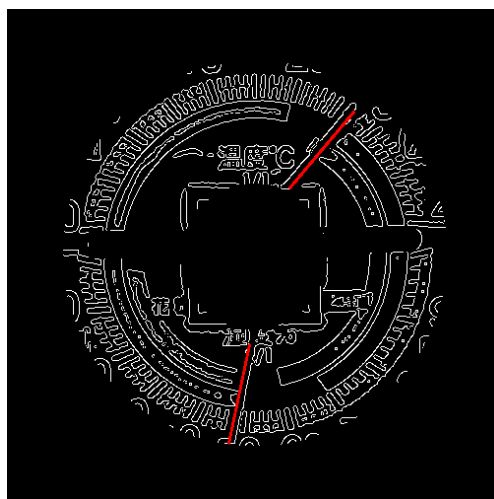
読み取り結果

温度 25.3、湿度 48.1 として値の読み取りに成功したときの元画像は図 5.8、検出した線分は図 5.9 のようになりました。

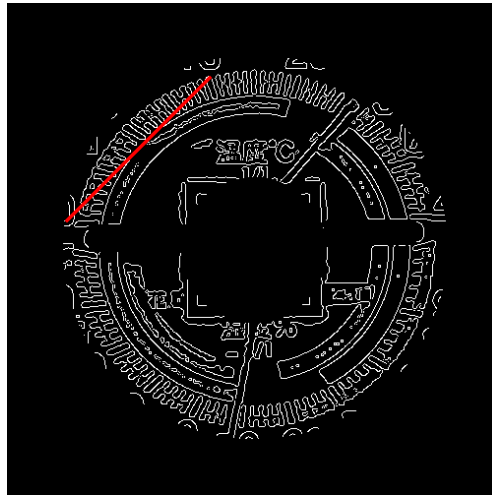
また、針の検証に失敗した場合の例は図 5.10 や図 5.11 のようになりました。



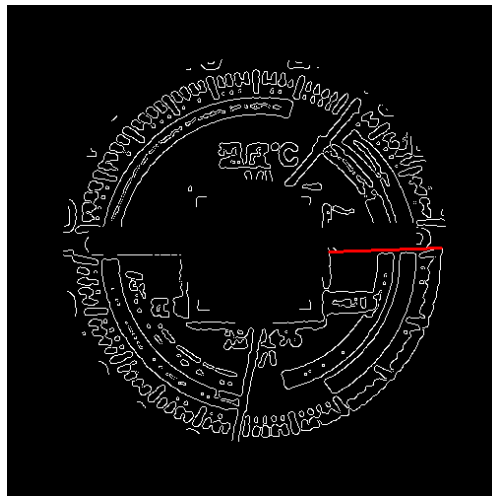
▲ 図 5.8 読み取りに成功したときの元画像



▲ 図 5.9 読み取りに成功したときの線分



▲ 図 5.10 画像中心から外れた線分が除外され、線分の本数が 0 本となったケース



▲ 図 5.11 検出した線分の本数が足りないため読み取りに失敗したケース

5.6 考察

読み取り試行ごとに毎回値が取れるわけではないですが、向きなどを変えながら撮影しておかしたパターンを除外すれば、ある程度想定どおりの値を取り出すことができます。

今回は回転する針なので角度を見ればほぼ OK なのですが、水平な針が上下に平行移動するものなど位置を読み取る必要があるもの場合には、AR マーカーの姿勢推定の結果

を使って位置を補正してあげる必要もありそうです。

ボツ案について

元々は針の動くのと同じ平面上に AR マーカーを設置していました。これによって、ある程度マーカーを斜めから撮影しても補正後の針の位置があまりブレないようになっていたのですが、マーカーを計器の外の空中に固定する必要があったためボツになりました。

5.7 おわりに

思っていた以上に調整に手間がかかりましたが、AR マーカーを貼り付けるだけでお手軽に検証でき、自分の考えた判定ロジックを直接適用できます。

OpenCV と numpy の機能を色々使うことになるので、Python で画像処理を行う教材としてもちょうどいい内容なのではないかと感じました。OpenCV や numpy は機能がたくさんあって引数や返り値の扱い方を調べるのが大変ですが、うまく扱えると少ない行数で色々実現できて面白いです。

第 6 章

TypeScript で低レベルメモリ操作 入門

Shinya Naganuma / @Pctg_x8

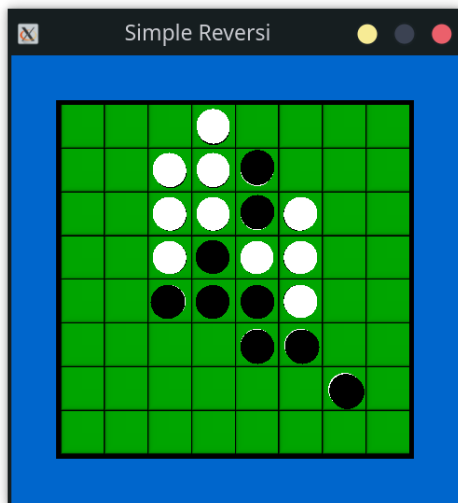
ゲームを作る際、GPU に送るためのバッファデータを構築することがあると思います。このとき GPU 側と同一のメモリレイアウトでデータの読み書きができれば、GPU 側との同期をとる際に変換処理を挟む必要がなくなるため効率のよいプログラムを書くことができます。しかし、ゲームロジックの記述言語として JavaScript (TypeScript) などのメモリレイアウトを指定できない言語を採用している場合、先述のようなデータの構築は簡単にはできないといった問題があります。

本章では、その問題に対するひとつの解決策について、TypeScript 上でリバーシゲームを作成する例を用いて紹介していきます。

6.1 前提

図 6.1 のような単純なリバーシゲームの盤面を単一の定数バッファで表現することを考えます。単一の定数バッファで盤面の状態を管理することで、インスタンス描画を使用して Draw Call ひとつで盤面を描画できるようになります。

シェーダの記述に使用する言語としては GLSL (バージョン 1.5) をここでは前提とします。



▲ 図 6.1 今回のサンプル

また、今回、TypeScript はゲームロジックの処理に集中させ、GPU との通信やウィンドウ管理などは Rust 側のプログラムに任せる方式をとることにします。つまり、Rust 側のコードがエンジンで TypeScript がユーザー定義コードといったシンプルなゲームエンジンのような構成をここでは前提とします。

Rust 側のプログラムがどういった形になっているかは、詳しくはサンプルプログラム*1を参照してほしいのですが次の 2 つからなっています。

- GLFW/OpenGL を使って描画する部分
- V8 を使って JavaScript を駆動する部分

6.2 単純な表現: 白、黒、または石なしの 3 ステート

リバーシゲームにおける 1 つのマスが取りうる状態は次の 3 つです。そのため、1 マスを最低限表現するためには 2bit あれば足りることになります。

- 石が置かれていて、面が白の状態
- 石が置かれていて、面が黒の状態
- どの石も置かれていない状態

ただし、GLSL では 2bit の数値を表現する型が存在しないので `uint` (32bit) で表現することになります (リスト 6.1)。

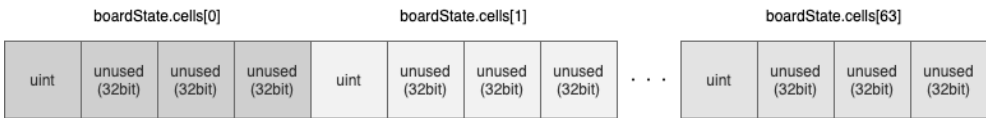
*1 https://github.com/Pctg-x8/simple_reversi

▼リスト 6.1 GLSL でのマス/盤面の表現

```
layout(std140) uniform BoardState {
    uint cells[8 * 8];
} boardState;
```

GLSL 側の定義はこれで OK です。次は TypeScript 側でこれをどう表現するかを考えます。

GLSL 側の定数バッファのメモリレイアウトは、`layout(std140)`では配列の各要素のサイズは 16 バイトに切り上げられるため図 6.2 のようになっています*2。これと同等のメモリ領域を JavaScript 上で確保し、同じようにアクセスを行う必要があります。



▲図 6.2 uniform BoardState のメモリレイアウト

JavaScript では、生のバイナリデータを扱う方法として `ArrayBuffer`が標準で提供されています。ただし `ArrayBuffer`のみで値を直接操作できず、`DataView`を経由する必要があります。これら 2つを組み合わせると、シェーダ側と同一の盤面表現をリスト 6.2 のように記述できます。

▼リスト 6.2 TypeScript での盤面表現

```
class BoardState {
    private readonly cells = new ArrayBuffer(8 * 8 * 16);

    getCell(x: number, y: number): number {
        return new DataView(this.cells, (x + y * 8) * 16).getUint32(0, true);
    }
    setCell(x: number, y: number, state: number) {
        new DataView(this.cells, (x + y * 8) * 16).setUint32(0, state, true);
    }

    // 省略
}
```

ついでですので、セルの状態ビットの操作もモデリングして隠蔽してしまいます (リスト 6.3)。今回は最下位 1 ビットを石の色、`0x80`を石が置いてあるかどうかで表現します。また、余計な変換コストを抑えるために単純な `number`の別名で表現します。

*2 [https://www.khronos.org/opengl/wiki/Interface_Block_\(GLSL\)#Memory_layout](https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Memory_layout)

▼リスト 6.3 セル状態のモデリング

```
/** uint32 value */
export type CellState = number;
export function newCellState(): CellState {
  return 0;
}
export function isStonePlacedCell(state: CellState): boolean {
  return (state & 0x80) != 0;
}
export function isWhiteStoneCell(state: CellState): boolean {
  return (state & 0x01) != 0;
}
export function place(color: "white" | "black"): CellState {
  return 0x80 | (color === "white" ? 0x01 : 0x00);
}
export function flip(state: CellState): CellState {
  if (!isStonePlacedCell(state)) return state;
  return state ^ 0x01;
}
```

ここまでで、あとはリスト 6.3 で定義した関数を使ってリバーシのロジックを組みればひととおり動くものになります。

6.3 Rust との連携

次は TypeScript 側の状態バッファを GPU に転送するため、TypeScript 側のバッファを Rust 側に共有できるようにしていきます。これにはいくつか方法がありますが、今回は JavaScript から Rust 側の関数を呼ぶことで `ArrayBuffer` を Rust 側に渡せるようにする形をとります。

V8 (`rusty_v8`) を使って JavaScript から Rust の関数を呼べるようにするには、まずは Rust 側でリスト 6.4 のような定義の関数を用意します。

▼リスト 6.4 V8 から呼ばれる関数のシグネチャ

```
fn rust_js_bridged_function(
  scope: &mut rusty_v8::HandleScope,
  args: rusty_v8::FunctionCallbackArguments,
  return_value: rusty_v8::ReturnValue
) { ... }
```

`scope`はこの関数が呼ばれたスコープを表し、主にローカルスコープでのみ生存する一時的な値の生成に使用します。`args`はその名のとおり引数を表し、`return_value`には関数の戻り値を設定します。

V8 の値は型が動的に決まるため、`args`から得られる値も任意の型をとるものとなっています。`rusty_v8::Local<T>`は `std::convert::TryFrom` を実装していますので、これを使うことで型検査と値の取り出しを行うことができます。`TryFrom` を使って値を取り出し、型が違った場合に `TypeError` の例外を送出するコードをリスト 6.5 に示します。

▼リスト 6.5 引数の型チェック

```

let v = match rusty_v8::Local::<rusty_v8::ArrayBuffer>::try_from(args.get(0)) {
    Ok(v) => v,
    Err(e) => {
        // Construct TypeError and throws it
        let msg = rusty_v8::String::new(scope, &e.to_string()).unwrap();
        let err = rusty_v8::Exception::type_error(scope, msg);
        scope.throw_exception(err);
        // Requires escaping(throw_exception does not interrupt the execution!)
        return;
    }
};

```

関数が用意できたら、次はその関数を JavaScript のグローバルから呼べるように V8 に登録します。V8 に関数を登録するコードをリスト 6.6 に示します。

▼リスト 6.6 関数の登録

```

let mut scope = v8::HandleScope::with_context(&mut isolate, &context);
let global = context.get(&mut scope).global(&mut scope);
let name = v8::String::new(&mut scope, "functionName").unwrap();
let func = v8::FunctionTemplate::new(&mut scope, rust_js_bridged_function)
    .get_function(&mut scope)
    .unwrap();
global.set(&mut scope, name.into(), func.into());

```

Rust 側で定義した関数の実体を V8 で取り扱うには、まず該当の関数をもとに `FunctionTemplate` を作成してから、そのテンプレートの値を取得するといった順序をとります。関数の値が取れれば、あとはグローバルを表すオブジェクトのメンバとして登録すれば JavaScript 側から見えるようになります。グローバルを表すオブジェクトは V8 のコンテキストオブジェクトから入手できます。

こうして登録した関数は、TypeScript からはリスト 6.7 のように宣言することで利用できます。

▼リスト 6.7 外部関数の宣言

```

declare function functionName(arg1: number, arg2: ArrayBuffer): number;

```

ここまでで JavaScript からオブジェクトを Rust 側で受け取ることができるようになりましたが、受け取ったオブジェクトを次の定数バッファの更新のタイミングまでどこかに保持しておく必要があります。グローバルな `static` 変数に置いてもよいですが、`rusty_v8::Isolate::get_slot_mut` を使用して `rusty_v8::Isolate` に保持させることでグローバル空間を汚さず、かつ `unsafe` なしで値を持ち運ぶことができます。`get_slot_mut` を使用して値を `Isolate` に保持させる例をリスト 6.8 に示します。

▼リスト 6.8 rusty_v8::Isolate::get_slot_mut の使用例

```

struct IsoState {
    pub next_updated_buffer: Option<v8::Global<v8::ArrayBuffer>>
}
// 値を変更
scope.get_slot_mut::<IsoState>().unwrap().next_updated_buffer = Some(buffer);
// 値を参照
scope.get_slot::<IsoState>().unwrap().next_updated_buffer.is_some();
    
```

slot や static 変数で JavaScript のオブジェクトを持ち回す場合は rusty_v8::Local を rusty_v8::Global に変換する必要があります。

6.4 少し複雑な表現: フリップを GPU で行う

現在のままでは石を反転する際に動きがなくて少し寂しいので、シェーダで簡易的なアニメーションを作成しようと思います。

シェーダでアニメーションさせるには次の 2 つが必要です。

- シェーダにゲーム開始からの時間を渡す
- インスタンスごとのシェーダ定数にアニメーション開始時間を追加する

前者は単に uniform 変数を追加してあげればよいので、ここでは後者について主に考えていきます。

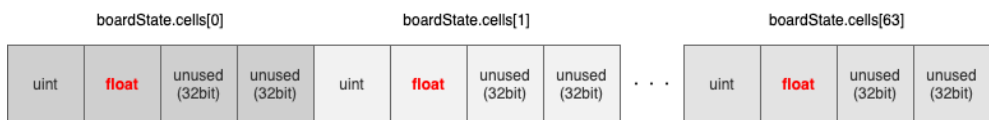
まずはシェーダ側の変更後の定義をリスト 6.9 に示します。配列の要素を構造体にして、マスの状態フラグの他にアニメーション開始時間のフィールドを記述すれば OK です。

▼リスト 6.9 GLSL での盤面状態

```

struct CellState {
    uint stateFlags;
    float flipStartTime;
};
layout(std140) uniform BoardState {
    CellState cells[8 * 8];
} boardState;
    
```

リスト 6.9 の場合のメモリレイアウトは図 6.3 のようになります。



▲図 6.3 uniform BoardState のメモリレイアウト

この定義に沿って TypeScript 側の定義を変更したものがリスト 6.10 になります。

▼リスト 6.10 TypeScript での盤面表現

```

class BoardState {
  private readonly cells = new ArrayBuffer(8 * 8 * 16);

  getCellStateFlags(x: number, y: number): number {
    return new DataView(this.cells, (x + y * 8) * 16).getUint32(0, true);
  }
  setCellStateFlags(x: number, y: number, state: number) {
    new DataView(this.cells, (x + y * 8) * 16).setUint32(0, state, true);
  }

  setCellFlipStartTime(x: number, y: number, time: number) {
    new DataView(this.cells, (x + y * 8) * 16).setUint32(4, time, true);
  }
}

// 省略
}

```

さて、これでもよいのですが今後フィールドを追加していくことを考えるとこれはすこし厳しいコードです。特に配列の要素サイズの 16 や各フィールドのオフセット値は `BoardState` というより `CellState` が管理した方が自然ですし、フィールド追加時の変更量も抑えられるはずなのでここをなんとかして整理します。

ここでは、Haskell の `class Foreign.Storable.Storable` を参考にした仕組みを構築していきます。Haskell はネイティブコードにコンパイルされる言語でありながら、JavaScript と同じように言語仕様として型のメモリレイアウトを明示的に指定する機能を持ちません。そのかわり、標準ライブラリにメモリ取り扱いの共通プロトコルを定めた型クラスが存在し、それを用いることで実質的にメモリレイアウトを詳細に指定できるようになっています。似たような境遇の言語ですからそこで使われている仕組みもきょうまく流用できるはずで

`Foreign.Storable.Storable` の最小の定義をリスト 6.11 に示します。

▼リスト 6.11 `class Foreign.Storable.Storable`

```

class Storable a where
  -- 型 a のメモリ上のサイズを返す
  sizeOf :: a -> Int
  -- 型 a のアラインメントを返す
  alignment :: a -> Int
  -- ポインタから型 a を読む (要するにデリファレンス)
  peek :: Ptr a -> IO a
  -- ポインタに a を書き込む
  poke :: Ptr a -> a -> IO ()

```

`sizeOf/alignment` でメモリ上の配置情報を記述し、`peek/poke` で詳細なフィールドの読み書きを行う形になっています。これを参考にして、TypeScript のコードも変更しやすい形に修正していきます。

まずは取り扱うデータ構造 (`CellState`) の `Storable` 定義をリスト 6.12 に示します。データの `getter/setter` もあわせて定義するため、コンストラクタで `DataView` を受け取るようにしています。

▼リスト 6.12 CellState の Storable 定義

```
class CellStateStorable {
    static readonly SIZE: number = 4 * 2;
    private static readonly STATE_FLAGS_OFFSET: number = 0;
    private static readonly FLIP_START_TIME_OFFSET: number = 4;
    constructor(private readonly view: DataView) {}

    get stateFlags(): number {
        return this.view.getUint32(CellStateStorable.STATE_FLAGS_OFFSET, true);
    }
    set stateFlags(value: number) {
        this.view.setUint32(CellStateStorable.STATE_FLAGS_OFFSET, value, true);
    }
    set flipStartTime(value: number) {
        this.view.setUint32(
            CellStateStorable.FLIP_START_TIME_OFFSET,
            value,
            true
        );
    }
}
```

この定義を使うと、CellState本体はリスト 6.13 のように記述できます。

▼リスト 6.13 CellState

```
class CellState extends CellStateStorable {
    init() {
        this.stateFlags = 0;
        this.flipStartTime = 0;
    }

    get placed(): boolean {
        return (this.stateFlags & 0x80) != 0;
    }
    // 中略
    flip() {
        if (!this.placed) return;
        this.stateFlags ^= 0x01;
    }
    // 省略
}
```

だいぶ通常のフィールドと近い形で自然に読み書きができています。

次に Storable 定義を受け取って適切な ArrayBuffer を作成してくれるラップクラスも作成していきます。layout(std140) の配列は 16 バイトにアラインメントが切り上げられるので、それもこのクラスでハンドリングするようにします。

まずは任意の Storable 定義を受け取るためのインターフェイス型をリスト 6.14 に示します。

▼リスト 6.14 Storable 定義の共通インターフェイス

```
type StorableClassProperties<T> = {
    readonly SIZE: number;
    new (view: DataView): T;
};
```

TypeScript では、new を使うことにより型定義でコンストラクタを表現できます。この

定義を使った配列のラップクラスの定義をリスト 6.15 に示します。

▼リスト 6.15 layout(std140) で使える配列のラップクラス

```
class StorableStd140Array<T> {
  readonly buffer: ArrayBuffer;
  private readonly stride: number;

  constructor(
    private readonly cls: StorableClassProperties<T>,
    elements: number
  ) {
    // glsl std140 layout restriction
    this.stride = Math.trunc((cls.SIZE + 15) / 16) * 16;
    this.buffer = new ArrayBuffer(this.stride * elements);
  }

  get(index: number): T {
    return new this.cls(
      new DataView(this.buffer, this.stride * index, this.cls.SIZE)
    );
  }
}
```

これらの定義を使うと、BoardStateはリスト 6.16 のようになります。

▼リスト 6.16 BoardState

```
class BoardState {
  // cells は自動的に StorableStd140Array<CellState>に推論される
  private readonly cells = new StorableStd140Array(CellState, 8 * 8);

  constructor() {
    for (let y = 0; y < 8; y++) {
      for (let x = 0; x < 8; x++) {
        this.cells.get(x + y * 8).init();
      }
    }
    this.cell(3, 3)!.place("black");
    this.cell(4, 4)!.place("black");
    this.cell(4, 3)!.place("white");
    this.cell(3, 4)!.place("white");
  }

  cell(x: number, y: number): CellState | undefined {
    if (0 <= x && x < 8 && 0 <= y && y < 8) {
      return this.cells.get(x + y * 8);
    }
  }
  // 省略
}
```

メモリレイアウトにまつわる処理をすべて BoardStateからはがすことができました。

6.5 おわり

本章では、TypeScript 上でメモリレイアウトを定義するために Haskell の `Foreign.Storable` を参考にした仕組みの例を紹介しました。

依然 `CellStateStorable` が手書きですので、ここをうまい具合に必要なデータだけを与えて自動生成するようなものが作れるとよりいっそう洗練された仕組みにできるかなと

思います。JavaScript はかなりメタプログラミング的なこともできるため、そういった自動化をするのも今後の課題として取り組んでいきたいなと思います。

サンプルプログラムについて

https://github.com/Pctg-x8/simple_reversi に今回の仕組みを実装したサンプルが置いてあります。

次のコマンドで実行できるようになっています。npm 7.14 以降が必要です。

```
$ cd scripts
$ npm i && npm run build
$ cd ..
$ cargo run
```

第7章

ミニマル Git を自作しよう

Hiroki Gomi / @530506

今日、世界でもっとも広く利用されている分散バージョン管理システム (DVCS)*1は Git です。このオープンソースプロジェクトは、Linux カーネルの有名な作成者である Linus Torvalds 氏によって 2005 年に開発され、以降積極的に保守されています。いまや Git は当たり前のように使われているかもしれませんが、本章ではその一部コマンドを自作することで、内部で実際にどのような処理が行われているのか、理解を深めてみましょう。

本章で扱うミニマル Git の実装例には Haskell を使用し、Git バージョン 2.17.1*2を対象とすることとします。ここでいう「ミニマル」とは、初期化からステージング、コミットが行えることと定義します。本章を通じて、各種 Git コマンドのうち、表 7.1 に記載のコマンドに相当する操作を自作のミニマル Git で行います。

▼表 7.1 コマンドの一覧

名称	機能
add	ファイルの内容をインデックスに追加する
cat-file	リポジトリオブジェクトのコンテンツまたはタイプとサイズの情報を提供する
diff	コミット間、またコミットとワーキングディレクトリ間の変更差分を表示する
hash-object	オブジェクト ID を計算し、オプションによってファイルから blob を作成する
init	空のリポジトリを作成する
ls-files	インデックスとワーキングディレクトリ内のファイルに関する情報を表示する
status	ワーキングディレクトリの状態を表示する
commit	リポジトリへの変更を記録する

*1 DVCS は、しばしば集中バージョン管理システム (CVCS) と比較されることがあります。CVCS はソフトウェアの完全なバージョン履歴を 1 箇所で作成するシステムであるのに対し、Git による分散管理は、すべての開発者がリポジトリ全体をミラーリングすることによって、全データを分散管理します。これらのより詳しい説明や経緯については、Pro Git を確認することをお勧めします。

*2 git/git at v2.17.1

本章で扱われるサンプル実装の完全なソースコードは、下記からご覧いただけます。

- <https://github.com/falgon/hmgit/tree/v1.0.0-gijutsushoten11>

サンプル実装に関するドキュメントは、下記からご覧いただけます。

- <https://falgon.github.io/hmgit/>

なぜ Haskell ?

Git オブジェクトやインデックス等のバイナリを読むためのパーサや、pathspec のパーサを書くのが楽であるため、またコンパイル時にバージョン情報を埋め込んだり、テスト時に必要な情報を埋め込んだりするのためのメタプログラミングができる言語であることが好ましかったため、特に深い意味はありません:)

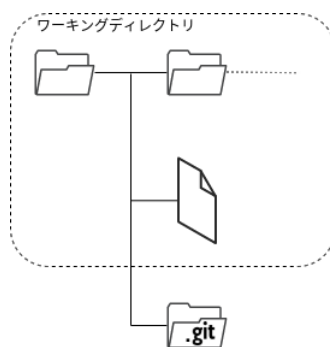
7.1 Git のしくみと構造

ミニマル Git の自作に必要な事前知識について確認します。

Git によって管理されるプロジェクト

Git によって管理されるプロジェクトは、ワーキングディレクトリと `.git` ディレクトリの2つから成ります。ワーキングディレクトリとは、Git によって管理されるファイルがあるディレクトリ、すなわち Git によって管理されるプロジェクトのディレクトリ全体を言います。`.git` ディレクトリとは基本的に^{*3}、ワーキングディレクトリのもっとも高い階層にある、ローカルリポジトリ^{*4}の本体です。

Git の構造を確認するため、後の節ではこの `.git` ディレクトリ内に着目していきます。



▲図 7.1 Git によって管理される基本的なプロジェクト

^{*3} `--git-dir` オプションによって明示的に指定する場合や、`GIT_DIR` 環境変数の設定により、必ずしもそうでないケースが有ります。

^{*4} bare リポジトリとの対称として non-bare リポジトリともいわれます。

「ワーキングディレクトリ」について

ここで定義した「ワーキングディレクトリ」は、「ワーキングツリー」と同義です。「ワーキングディレクトリ」は、かつて「現在のディレクトリ」を意味する言葉と混同されていた経緯があり、たとえば Git v2.9.1 のリリースでは関連する `status` コマンドのメッセージが修正されています (<https://github.com/git/git/commit/2a0e6cddedab306eccbd297c051035c13d0266343>)。しかし、たとえば `git status some-sub-dir` を実行したときに得られる結果は、ワーキングツリー全体に関するものではありません。したがって、修正されたメッセージも、依然として最適であるとはいえないでしょう——

本章では広く一般的に用いられてきた(?)「ワーキングディレクトリ」という言葉を使用します。

Git オブジェクトモデル

`.git/objects/` ディレクトリ配下に格納されるファイルは (Git) オブジェクトといわれ、その種類として、**Blob**、**Commit**、**Tree**、**Tag**^{*5}があります。**Blob** はファイル、**Commit** はコミット、**Tree** はディレクトリ、**Tag** はタグに対応します。それぞれがどのような場面でどのように使われるのかの詳細については後述するとして、まずはこのオブジェクトらがどのような命名規則で保存されるのか、またその中身はどのようなものなのかについて着目します。

オブジェクト ID と contents

オブジェクトの名前をオブジェクト ID と言います。オブジェクト ID は、次の形式 7.1^{*6}に従うバイト列を入力とした、SHA1 ハッシュ値によって定まります。

$$\begin{array}{c}
 \text{header} \\
 \overbrace{\text{object type} \diamond 0x20 \diamond \text{contents size} \diamond 0x00 \diamond \text{contents}} \longrightarrow_{\text{SHA1}} \text{object ID} \\
 \\
 \text{object type} := \begin{cases} \text{"blob"} \\ \text{"commit"} \\ \text{"tag"} \\ \text{"tree"} \end{cases} \quad (7.1)
 \end{array}$$

*5 本章では、**Tag** については扱いません。

*6 \diamond は、接続の二項演算記号とします。字母集合上の有限文字列全体と接続の二項演算、単位元として空文字列を定義すると、これらの組はモノイドになります。

contents は、**Blob** の場合はファイルの中身、**Commit**^{*7}と **Tree** の場合はそれぞれ次の形式となります^{*8}。

```
Commit := "tree" ◊ 0x20 ◊ SHA1 tree hash ◊ 0xa
        ◊ {
            "parent" ◊ 0x20 ◊ SHA1 parent tree hash ◊ 0xa
            ""
        }
        ◊ "author" ◊ 0x20 ◊ author email ◊ 0x20 ◊ author time ◊ 0xa
        ◊ "committer" ◊ 0x20 ◊ committer email ◊ 0x20 ◊ committer time ◊ 0xa ◊ 0xa
        ◊ commit message

Tree := {
    octal ◊ 0x20 ◊ file path ◊ 0x00 ◊ SHA1 object hash ◊ Tree
    ""
}
```

(繰り返しになりますが) **Tree** オブジェクトはディレクトリに対応しています。**Tree** オブジェクトの contents の定義は再帰的な形になっていますが、これは自分自身の配下にあるファイル (**Blob**) またはサブディレクトリ (サブ **Tree**) によって、contents 内に次々と羅列されることが表されています。

先頭の octal の箇所には、たとえば 0100644 や 0040000 等、ファイルの状態を示す 8 進数値が置かれます。stat(2) に記載があるように、ファイル種別を示すビット領域を表すビットマスク 0170000 (8 進数値) を使用し、照合することで、ファイル種別の識別ができます。

Git には、コンテンツからオブジェクト ID を計算する hash-object コマンドがあります。実際に sha1sum コマンド等で計算させ、それらの結果を照らし合わせることで、オブジェクト ID が 69 ページに記載の形式 7.1 にしたまっていることが確認できます。以下は、**Blob** オブジェクトでの例です。

```
$ SHA1_SUM=$(echo -n "blob 4\0hoge" | sha1sum | cut -f1 -d' ' | tee /dev/stderr)
c2684e0321eedff1890b7690c89726387d2af3ca
$ GIT_HASH_OBJECT=$(echo -n hoge | git hash-object -t blob --stdin | tee /dev/stderr)
c2684e0321eedff1890b7690c89726387d2af3ca
$ test $GIT_HASH_OBJECT = $SHA1_SUM; echo $?
0
```

オブジェクトの関係性

Tree オブジェクトは、自分自身の名前や場所について一切知りません。**Tree** オブジェクトが知っているのは、自分自身の配下にあるサブディレクトリ (サブ **Tree**) またはファイル (**Blob**) のみです。それでも、ワーキングディレクトリをすべてたどることができるのであれば、何ら問題はありません。そして、そのためには、ワーキングディレク

^{*7} **Commit** オブジェクトの contents は、コミットへの署名がオンになっている場合、"committer" の行の後に、"gpgsign" が続き、その後にキーが配置されます。

^{*8} 単一の線形的なブランチの場合

トリのルート **Tree** が分かればよいということが分かります。**Commit** オブジェクトの SHA1 tree hash は、まさにこのワーキングディレクトリのルート **Tree** のハッシュ値が入っています。この関係性によって、任意のコミットからワーキングディレクトリのすべてをたどることができるようになっています。実際にコマンドを実行して、それぞれのオブジェクトがどのような関係を持っているのか確認してみます。

```
$ git init playground \
  && cd playground \
  && mkdir -p foo/bar \
  && echo hoge > foo/bar/hoge1.txt \
  && echo hoge > foo/hoge2.txt \
  && git add -A \
  && git -c commit.gpgsign=false \
  -c user.name=roki \
  -c user.email=roki@example.com \
  commit \
  -m 'initial commit' \
  --author 'roki <roki@example.com>'
Initialized empty Git repository in /home/roki/Documents/src/shell/playground/.git/
[master (root-commit) 2b631ca] initial commit
2 files changed, 2 insertions(+)
create mode 100644 foo/bar/hoge1.txt
create mode 100644 foo/hoge2.txt
```

該当の **Commit** オブジェクト ID は 2b63... のようです。冒頭でも示しましたように、**git**にはオブジェクトのコンテンツやタイプ、またサイズ等を確認できる **cat-file** というコマンドがあります。これを使って、2b63... の中身を確認してみましょう。

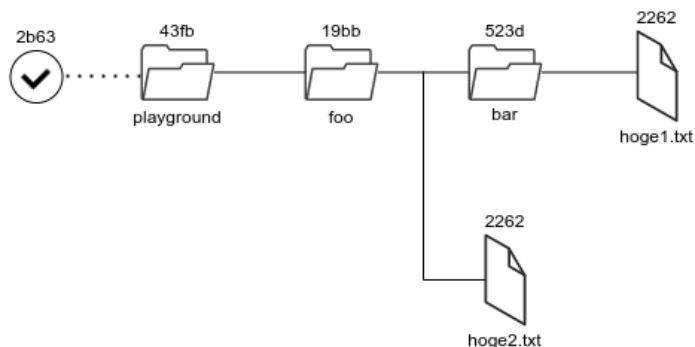
```
$ git cat-file -p 2b63
tree 43fb1248d0a63903af1df18c1734c408b37f69e7
author roki <roki@example.com> 1621913303 +0900
committer roki <roki@example.com> 1621913303 +0900

initial commit
```

該当のルート **Tree** オブジェクト ID は 43fb... のようです。あとはこれを次々に繰り返すことで、全ワーキングディレクトリをたどっていくことができるということが分かります。

```
$ git cat-file -p 43fb # ワーキングディレクトリのルート Tree オブジェクト
040000 tree 19bb2a16cebedacbc124ebffcc12f3b2d5af3378   foo # サブ Tree foo がある
$ git cat-file -p 19bb
040000 tree 523d7205b2f4849f90b28fe651878fbc8cd1dd06   bar # foo にはサブ Tree bar がある
100644 blob 2262de0c121f22df8e78f5a37d6e114fd322c0b0   hoge2.txt # foo には Blob hoge2.txt
          がある
$ git cat-file -p 523d
100644 blob 2262de0c121f22df8e78f5a37d6e114fd322c0b0   hoge1.txt # bar には Blob hoge1.txt
          がある
$ git cat-file -p 2262
hoge
```


ここで、`git cat-file -p 19bb`によって表示される `hoge2.txt` の **Blob** オブジェクト ID と、`git cat-file -p 523d`によって表示される `hoge1.txt` の **Blob** オブジェクト ID が一致していることが確認できます。ページ 69 の 7.1 を確認すると分かるように、`contents` の中身が同じファイルは、そのファイルパスの階層が異なっていたとしても同一の **Blob** オブジェクトとしてリポジトリに登録されます。



▲ 図 7.2 Commit オブジェクトとの関係図

オブジェクトの中身

生成された Git オブジェクト、たとえば `2262...` を確認してみると、オブジェクト ID の 2 文字をサブディレクトリ、残りの 38 文字をファイル名として配置していることが分かります。

```
$ file .git/objects/22/62de0c121f22df8e78f5a37d6e114fd322c0b0
.git/objects/22/62de0c121f22df8e78f5a37d6e114fd322c0b0: zlib compressed data
```

この実行結果を見ると、オブジェクトは `zlib` 形式で圧縮されているようです。実際、次のようにしてオブジェクトの中身を確認できます。

```
$ cat .git/objects/22/62de0c121f22df8e78f5a37d6e114fd322c0b0 | \
  zlib-flate -uncompress
blob 5hoge
$ echo -e "-t\n-s\n-p" | xargs -n1 -I{} git cat-file {} 2262
blob
5
hoge
```

いまは **Blob** オブジェクトの例を見ましたが、他のオブジェクトについても同様に、69ページの形式 7.1 の関数を `zlib` に変えたものがオブジェクトの本体となります。

$$\overbrace{\text{object type} \diamond 0x20 \diamond \text{contents size} \diamond 0x00 \diamond \text{contents}}^{\text{header}} \rightarrow_{\text{zlib}} \text{object body}$$

$$\text{object type} := \begin{cases} \text{"blob"} \\ \text{"commit"} \\ \text{"tag"} \\ \text{"tree"} \end{cases} \quad (7.2)$$

ステージングとインデックス

たとえば、あるプロジェクト内にローカルリポジトリを作成し、次に、そのプロジェクトに対して新しいファイル `hello` を新規作成したとき、ファイル `hello` は（まだ）ローカルリポジトリの一部ではありません。ローカルリポジトリの一部とするためには、「ステージング領域」へ反映させる必要があります。ステージング領域への反映は、ファイル `hello` を **Blob** オブジェクトとして配置し、ファイル `hello` の（更新日時等の）情報を `.git/index` ファイルへ追記することによって行われます。

```
$ cd ../ && git init playground2 && cd playground2
$ find .git/objects -type f && find .git -type f -name index # まだ何もない
$ echo hello > hello # ワーキングディレクトリに変更を加える
$ git add hello # ステージング領域へ反映する
$ find .git/objects -type f
.git/objects/ce/013625030ba8dba906f756967f9e9ca394464a # オブジェクトが追加されている
$ git cat-file -t ce01 # オブジェクトタイプを確認
blob
$ find .git -type f -name index
.git/index # index ファイルが追加されている
$ git cat-file -p ce01 # オブジェクトの中身を確認
hello
```

ここで `.git/index` はインデックスといわれます。インデックスには、主に変更日時等の `stat(2)` によって取得できるデータが書き込まれます。インデックスのデータフォーマットに関する詳細は、次のドキュメントから確認できます。

- <https://github.com/git/git/blob/v2.17.1/Documentation/technical/index-format.txt>

コミット

「ステージング領域」への反映が完了しても、（まだ）リポジトリへの変更の記録はされていません。リポジトリへの変更の記録は、**Tree** オブジェクトと **Commit** オブジェクトを配置し、また、その変更がブランチ内の先頭コミットであることをファイル `.git/refs/heads/ブランチ名` に記録することで行われます。

```

$ find .git/objects -type f | \
  cut -d/ -f3,4 | \
  sed 's/$/$$g' | \
  xargs -n1 git cat-file -t
blob # (まだ) Blob オブジェクトしかない
$ git commit -m 'Add a hello text' hello
[master (root-commit) 215f506] Add a hello text
1 file changed, 1 insertion(+)
create mode 100644 hello
$ find .git/objects -type f | \
  cut -d/ -f3,4 | \
  sed 's/$/$$g' | \
  xargs -n1 -I{} bash -c "echo -n \"{}: \" && git cat-file -t {}"
b4d01e9b0c4a9356736dfddf8830ba9a54f5271c: tree
ce013625030ba8dba906f756967f9e9ca394464a: blob
215f506b8c6b30ec01fe32c0d8dadd8933940102: commit

```

70 ページで示したように、Commit オブジェクトには"parent"に続く SHA1 ハッシュ値が（ルート Commit 以外で）含まれています。これが親のコミットです。再度 playground プロジェクトに移り、リポジトリに変更を与え、Commit オブジェクトのそれぞれの関係について確認してみましょう。

```

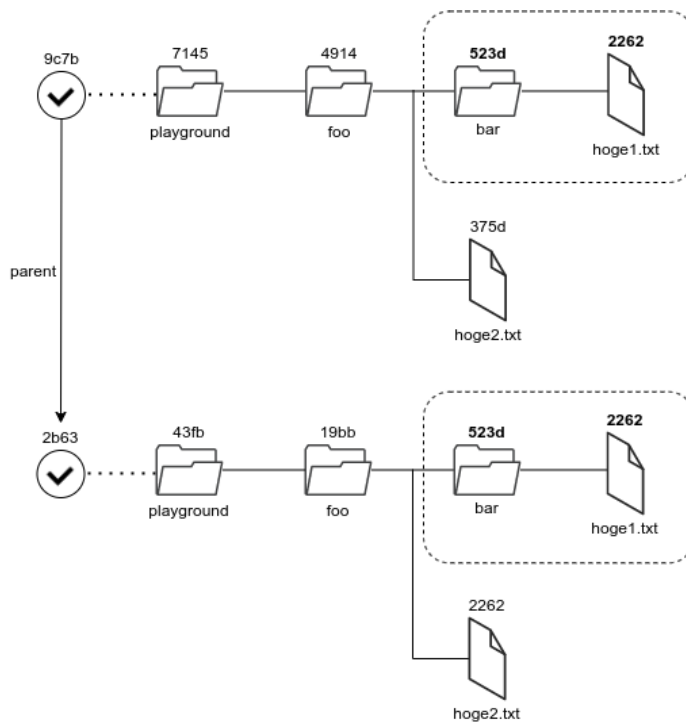
$ cd ../playground \
  && echo meow > foo/hoge2.txt \ # foo/hoge2.txt の内容を変更
  && git add foo/hoge2.txt \
  && git -c commit.gpgsign=false \
    -c user.name=roki \
    -c user.email=roki@example.com \
    commit \
    -m 'make hoge2.txt more cute \
    --author 'roki <roki@example.com>'
[master 9c7b053] make hoge2.txt more cute
1 file changed, 1 insertion(+), 1 deletion(-)
$ git cat-file -p 9c7b # 作成された Commit オブジェクトを確認
tree 7145f606253051e7187ed1034e0ba14b6cacf261
parent 2b631ca7066232e852c0c6ccfdc78469d53a5294
author roki <roki@example.com> 1621921028 +0900
committer roki <roki@example.com> 1621921028 +0900

make hoge2.txt more cute
$ git cat-file -p 2b63 # 親コミットを確認
tree 43fb1248d0a63903af1df18c1734c408b37f69e7
author roki <roki@example.com> 1621913303 +0900
committer roki <roki@example.com> 1621913303 +0900

initial commit
$ git cat-file -p 7145 # いまのコミットのワーキングディレクトリのルート Tree オブジェクト
040000 tree 491483dca385eb8616cbc2f4263df318384eccc0   foo # サブ Tree foo がある
$ git cat-file -p 4914
040000 tree 523d7205b2f4849f90b28fe651878fbc8cd1dd06   bar # foo にはサブ Tree bar がある
100644 blob 375d5c3ce54c13190b2be179e7f3717ebdf45adf   hoge2.txt # foo に
は Blob hoge2.txt がある
$ git cat-file -p 523d
100644 blob 2262de0c121f22df8e78f5a37d6e114fd322c0b0   hoge1.txt # bar に
は Blob hoge1.txt がある
$ git cat-file -p 2262 # foo/bar/hoge1.txt の中身
hoge
$ git cat-file -p 375d # foo/hoge2.txt の中身
meow

```

いま行った Commit である 9c7b... が親の Commit オブジェクトの情報をもっていることがわかります。これらの関係によって、Commit オブジェクトもまた、木構造を形成します。



▲図 7.3 Commit オブジェクトの親子関係図

破線の枠で覆われたオブジェクト ID が各 **Commit** 間で変化していないことに注意してください。変更を与えられた `foo/hoge2.txt` を起点として、その親 **Tree** へと変化が伝搬していることがわかります。

リファレンス

コミット（またはリファレンス）を指し示す概念がリファレンスで、`.git/refs/`配下に置かれます。中でも、たとえば`.git/refs/heads/`ブランチ名は、そのブランチの先頭 **Commit** オブジェクトの ID が書かれています。

```
$ cat .git/refs/heads/master
9c7b053fe0929135b3ac61c4595ef12589baa7bb
$ git cat-file -p 9c7b
tree 7145f606253051e7187ed1034e0ba14b6cacf261
parent 2b631ca7066232e852c0c6ccfdc78469d53a5294
author roki <roki@example.com> 1621921028 +0900
committer roki <roki@example.com> 1621921028 +0900

make hoge2.txt more cute
```

たとえば、`reset` コマンドによって、指し示す **Commit** を変更できます。

```
$ git log
commit 9c7b053fe0929135b3ac61c4595ef12589baa7bb (HEAD -> master)
Author: roki <roki@example.com>
Date: Tue May 25 14:37:08 2021 +0900

    make hoge2.txt more cute

commit 2b631ca7066232e852c0c6ccfdc78469d53a5294
Author: roki <roki@example.com>
Date: Tue May 25 12:28:23 2021 +0900

    initial commit
$ git reset --soft 2b63 # インデックスやワーキングディレクトリに影響を与えず、指し示す先を変更
$ cat .git/refs/heads/master
2b631ca7066232e852c0c6ccfdc78469d53a5294
$ git cat-file -p 2b63
tree 43fb1248d0a63903af1df18c1734c408b37f69e7
author roki <roki@example.com> 1621913303 +0900
committer roki <roki@example.com> 1621913303 +0900

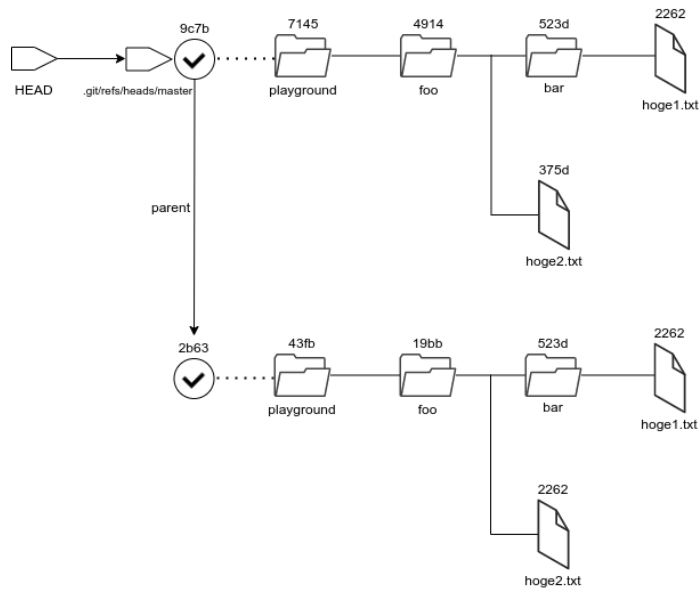
initial commit
$ git reset --soft HEAD@{1} # 一応戻す
```

また、`.git/HEAD`には、いまチェックアウトしているブランチを指す特別なリファレンスが記録されており、`checkout` コマンド^{*9}でこの内容は書き換わります。

```
$ cat .git/HEAD
ref: refs/heads/master
```

`.git/HEAD`は必ずしもブランチを指しているとは限りません。たとえば、**Commit** オブジェクトの ID を指定して `clone` を行くと、`.git/HEAD`にはその **Commit** オブジェクトの ID が記録された状態となり、これを `detached HEAD` 状態と言います。`detached HEAD` 状態でも **Commit** はできますが、その **Commit** を指し示すリファレンス、すなわちブランチがないため、この状態のまま `.git/HEAD` を変更すると、その **Commit** はどこからも参照されなくなり、やがて `git gc` によって削除されます。

^{*9} Git バージョン 2.23 から追加された、ブランチの切り替えや作成等専用の `switch` コマンドにおいても同様



▲図 7.4 ブランチと.git/HEAD リファレンスの関係図

7.2 実装

以上の説明でミニマル Git を実装するための理解は進みました。後は実装するだけです。お疲れさまでした——ではあまりにもですので、本節では、インクリメンタルに開発を進めていくことに焦点を置いて、サンプル実装を追っていきます。

冒頭でも示しましたとおり、今回は Haskell で実装します。まずは、最低限、リポジトリのデータベースへそのワーキングディレクトリ内のどこからでもアクセスできるように、その絶対パスを持ち回れるようにしたいです。このようなときは、ReaderT モナド変換子との組み合わせが定番です*10。

▼リスト 7.1 大域的な変数の設定

```
import qualified Path as P

data HMGitConfig = HMGitConfig {
  hmGitDir      :: P.Path P.Abs P.Dir
  , hmGitTreeLimit :: Int
} | HMGitConfigInit

type HMGitT = ReaderT HMGitConfig
```

今後は、この HMGitT 文脈下での作業が主となります。

*10 RIO モナドといったものもあります:<https://hackage.haskell.org/package/rio-0.1.20.0/docs/RIO.html#:RIO>

リポジトリの初期化

まずは、リポジトリを初期化するコマンド、`init`を実装します。最低限の機能として、リポジトリ名を受け取り、必要なファイル郡をそのディレクトリ配下に作成すればよいでしょう。後に扱いやすいように、`Init`という型に包んでいます。

▼リスト 7.2 `init` コマンドの核部分

```

type RepositoryName = String
newtype Init m = Init (RepositoryName -> HMGitT m ())

initBase :: MonadIO m
=> String
-> RepositoryName
-> HMGitT m ()
initBase gitName repoName = let hmGitRoot = repoName </> gitName in liftIO
$ mapM_ (createDirectoryIfMissing True) (dirs hmGitRoot)
*> BL.writeFile (hmGitRoot </> "HEAD") "ref: refs/heads/master"
where
  dirs hmGitRoot = map (hmGitRoot </>) [
    "objects"
    , "refs" </> "heads"
  ]

initDefault :: (MonadThrow m, MonadIO m) => String -> Init m
initDefault gitName = Init $ \repoName -> initBase gitName repoName
*> liftIO (putStrLn ("Initialized empty HMGit repository in: " <> repoName))

```

後はコマンドライン引数をパースして、よしなに呼び出してあげれば完成です。サンプル実装では、コマンドライン引数のパースに `Options.Applicative` モジュールを用いています*11。

次は、オブジェクトの生成と読み込みを行えるようにします。まずは、`hash-object` コマンドを作ります。ページ 69 の 7.1 にしたがって、任意のオブジェクトタイプとコンテンツからオブジェクトとその ID、またそのパスを生成する関数を用意します*12。

▼リスト 7.3 オブジェクトの生成 1

```

formatObject :: ObjectType
-> BL.ByteString
-> BL.ByteString
formatObject objType contents = BP.runPut $
  BP.putByteString (fromString $ show objType)
  *> BP.putWord8 32 -- Space
  *> BP.putByteString (fromString $ show $ BL.length contents)
  *> BP.putWord8 0
  *> BP.putLazyByteString contents

hashToObjectPath :: MonadCatch m
=> String
-> HMGitT m (Either (P.Path P.Abs P.Dir) (P.Path P.Abs P.File))
-- 略

```

*11 cf. <https://falgon.github.io/hmgit/HMGit-Commands-Porcelain-Init-Cmd.html#v-initCmd>

*12 cf. <https://falgon.github.io/hmgit/HMGit-Internal-Parser-Index.html>

```

-- | Create object ID, path, zlib compressed data from object type and content
fromContents :: MonadCatch m
=> ObjectType
-> BL.ByteString
-> HMGitT m ObjectInfo
fromContents objType contents = hashToObjectPath (hexStr objId)
>>= either
  (const $ throw $ BugException "fromContents: hashToObjectPath must give the Abs file")
  (pure . ObjectInfo objId (compress objFormat))
  where
    objFormat = formatObject objType contents
    objId = hashlazy objFormat

```

hashToObjectPathで見つかるのはディレクトリであるかファイルであるか分からないため^{*13}、Eitherで返しています。これを使って、リスト 7.4 のようにオブジェクトを配置する関数を用意します。

▼リスト 7.4 オブジェクトの生成 2

```

-- | Create an object to HMGit database
storeObject :: (MonadIO m, MonadCatch m)
=> ObjectType
-> BL.ByteString
-> HMGitT m B.ByteString
storeObject objType contents = do
  objInfo <- fromContents objType contents
  P.createDirIfMissing True (P.parent $ objectPath objInfo)
  *> liftIO (BL.writeFile (P.toFilePath $ objectPath objInfo) $ objectData objInfo)
  $> objectId objInfo

```

あとは、リスト 7.5 のようによしなに呼び出せば、hash-object コマンドの完成です。

▼リスト 7.5 hash-object コマンドの核部分

```

newtype HashObject m = HashObject (ObjectType -> BL.ByteString -> HMGitT m ())

hashObjectShow :: (MonadCatch m, MonadIO m)
=> HashObject m
hashObjectShow = HashObject $ \objType ->
  fromContents objType
  >=> liftIO . putStrLn . hexStr . objectId

hashObjectWrite :: (MonadCatch m, MonadIO m)
=> HashObject m
hashObjectWrite = HashObject $ \objType ->
  storeObject objType
  >=> liftIO . putStrLn . hexStr

hashObject :: (MonadCatch m, MonadIO m)
=> HashObject m
-> ObjectType
-> FilePath
-> HMGitT m ()
hashObject (HashObject f) objType fpath =
  liftIO (readFile fpath)
  >>= f objType . BLU.fromString

```

*13 これはオブジェクト ID の先頭 2 文字でオブジェクトを取り出せるようにするためです。

次は、`cat-file`コマンドを作ります。オブジェクトを読み込む関数と、`Tree` オブジェクトをパースする関数を用意します。

▼リスト 7.6 オブジェクトの読み込み

```
-- | Read an object from HMGit database by Object ID
loadObject :: (MonadIO m, MonadCatch m, MonadPlus m)
=> String
-> HMGitT m (ObjectType, BL.ByteString)
loadObject sha1 = do
  fname <- hashToObjectPath sha1
  >>= uncurry findTarget
    . either (,empty) (first P.parent . second (P.toFilePath . P.filename) . dupe)
  liftIO (BL.readFile $ P.toFilePath fname)
  >>= runByteStringParser objectParser fname . decompress
  where
    findTarget dir fname = catchAny (findTargetObject dir fname) $ const
      $ lift
      $ throw
      $ noSuchThing
      (printf "objects %s not found or multiple object (%d) with prefix %s"
        sha1 (length sha1) sha1)
      (P.toFilePath dir <> "/" <> fname)

    findTargetObject dir fname = P.listdirRel dir
      >>= strictOne . filter (isPrefixOf fname . P.toFilePath) . snd
      <&> (dir P.</>)

-- | Read a tree from raw data
loadTree :: MonadThrow m
=> BL.ByteString
-> HMGitT m [(CMode, P.Path P.Rel P.File, String)]
loadTree body = hmGitTreeLim
  >>= flip ('runByteStringParser' $(P.mkRelFile "index")) body
  . treeParser
```

`objectParser`は `Text.Megaparsec` を使ったパーサです。 `hashToObjectPath` で見つかったものがディレクトリであるとき、その内部に格納されているファイル (オブジェクト) は1つでなければなりません。これらを使って、オブジェクトが `Tree` であったとき、 `loadTree` でパースするようにすれば、 `cat-file` コマンドの完成です。

▼リスト 7.7 `cat-file` コマンド (`-p` オプション) の核部分

```
data CatFile m = CatFileObjectType ObjectType (ObjectType -> BL.ByteString -> HMGitT m ())
| CatFileMode (ObjectType -> BL.ByteString -> HMGitT m ())

instance MonadIO m => IsString (CatFile m) where
  fromString = catOptObject . read

catOptObjectPP :: (MonadIO m, MonadThrow m) => CatFile m
catOptObjectPP = CatFileMode $ \objType body ->
  if objType `elem` [ Commit, Blob ] then liftIO (BLC.putStrLn body)
  else loadTree body
  >>= mapM_ (\(mode, fpath, sha1) ->
    liftIO $ putStrLn $ unwords [
      printf "%06o" (fromIntegral mode :: Integer)
      , if sIsDir mode then "tree" else "blob"
      , sha1
    ] <> printf "\t%s" (P.toFilePath fpath))
```

インデックスの読み込みと書き込み

次はインデックスの読み込みと書き込みを行えるようにして、ワーキングディレクトリ内のファイルをステー징できるようにします。インデックスのフォーマットは先に記載したとおり Git のドキュメントから参照できますので、まずはその内容を参考に型とバイナリ生成器を用意します。

▼リスト 7.8 インデックスのデータ型とインデックスのバイナリ生成器

```
-- | Index format,
-- https://github.com/git/git/blob/v2.17.1/Documentation/technical/index-format.txt#L9-L17
data IndexHeader = IndexHeader {
  ihSignature :: BL.ByteString -- ^ The signature is { 'D', 'I', 'R', 'C' }
  , ihVersion  :: Word32        -- ^ The current supported versions are 2, 3 and 4.
  , ihNumEntries :: Word32      -- ^ Number of index entries.
}
deriving Show

putIndexHeader :: Word32 -> BP.Put
putIndexHeader len = BP.putByteString "DIRC"
  *> BP.putWord32be 2
  *> BP.putWord32be len

-- | Index entry
-- https://github.com/git/git/blob/v2.17.1/Documentation/technical/index-format.txt#L38
data IndexEntry = IndexEntry {
  ieCtimeS :: Word32 -- ^ the last time a file's metadata changed, this is stat(2)
  , ieCtimeN :: Word32 -- ^ nanosecond fractions, this is stat(2)
  , -- 略. いくつかフィールドが続く...
}
deriving Show

putIndexEntry :: IndexEntry -> BP.Put
-- 略

putIndex :: Foldable t => t IndexEntry -> BP.Put
putIndex ies = putIndexHeader (fromIntegral $ length ies)
  *> foldMapM putIndexEntry ies
```

サンプル実装では `ByteStringParser` のチャンクを `Get` への入力としていますが、このような場合は `Get` モナドから `ByteStringParser` モナドへの変換関数を用意しておく
と便利です*14。

▼リスト 7.9 `Get` モナドから `ByteStringParser` モナドへの変換

```
fromBinaryGetter :: (String -> ParseException) -> BG.Get -> ByteStringParser
fromBinaryGetter pException binGetter = M.getInput
  >>= either (M.customFailure . pException . thd3) (first3M M.setInput >>= pure . thd3)
  . BG.runGetOrFail binGetter

fromBinaryGetter' :: BG.Get -> ByteStringParser
fromBinaryGetter' = fromBinaryGetter IndexParser
```

*14 与太話: `Get` と `ByteStringParser` はモナドですので、どちらも関手の一種です。関手間の対応付けは自然変換になりますが、型の宣言->でそれが表されています。Haskell プログラミングの楽しみの 1 つは、こういった数学的構造を簡単に落とし込めることにあるのかもしれないですね。

後はこれらを用いてインデックスのパーサを作成するのですが、本書内に掲載すると非常に長くなってしまいますので、割愛します*15。

▼リスト 7.10 インデックスのパーサ

```
lookSignature :: IndexHeader -> ByteStringParser IndexHeader
lookSignature ih
  | ihSignature ih == "DIRC" = pure ih
  | otherwise = M.customFailure $ IndexParser "invalid index signature"

lookVersion :: IndexHeader -> ByteStringParser IndexHeader
lookVersion ih
  | ihVersion ih == 2 = pure ih
  | otherwise = M.customFailure $ IndexParser "unknown index version"

indexBody :: Word32 -> ByteStringParser [IndexEntry]
-- 略

indexParser :: ByteStringParser [IndexEntry]
indexParser = indexHeader
  >>= lookSignature
  >>= lookVersion
  >>= indexBody . ihNumEntries
```

このパーサを使って、インデックスを読み込む関数を用意します。

▼リスト 7.11 インデックスを読む関数

```
-- | Read index entries
loadIndex :: (MonadIO m, MonadThrow m) => HMGitT m [IndexEntry]
loadIndex = do
  fname <- hmGitIndexPath
  ifM (not <$> P.doesFileExist fname) (pure []) $
    liftIO (BL.readFile $ P.filePath fname)
    >>= runByteStringParser indexParser fname
```

これがそろえば、`ls-files`コマンドを実装できます。ですが、Git の `ls-files` コマンドには `pathspec` と呼ばれる引数を指定できるようになっています。pathspec についての詳細は `gitglossary(7)` に記載がありますので割愛させていただきますが、簡単にいうと `fnmatch(3)+α` の構文でマッチできる構文のことです。

```
# 単 1 文字のワイルドカード。mp3 や mp4 等...
$ git ls-files '*.mp?'
# 何かに続いて、の後、ブラケット内の 1 文字にマッチして's' にマッチ。ts や js ファイル等...
$ git ls-files '*. [tj]s'
# 一致したものについて大文字と小文字を区別しない。jpg と JPG 等...
$ git ls-files ':(icase)*.jpg'
```

*15 cf. <https://falgon.github.io/hmgit/HMGit-Internal-Parser-Index.html#v:indexParser>

最後のものは magic signature と呼ばれる機能です。これについては今回のサンプルでは実装していませんが、glob くらいの検索能力は欲しいので、ミニマル Git としては少し脇道になりますが作ってみましょう。まずは、glob の構文を BNF に起こしました。

```

<wildcard> ::= <expr>
              | <expr> <wildcard>
<expr> ::= <br>
           | '*'
           | '?'
           | <word>
br ::= <pos_bracket_expr>
      | <neg_bracket_expr>
<pos_bracket_expr> ::= '[' <word> ']'
<neg_bracket_expr> ::= '[' '!' <word> ']'

```

今回は簡単のため、glob を評価のための中間言語として正規表現に変換し、すでにある正規表現の評価器によってパターンマッチングを行うようにします。パーサコンビネータを使うと、実装はリスト 7.12 のように BNF の形をほぼそのまま書き出すことができます（後に識別が容易くなるよう、ラベリングを行っています）。

▼リスト 7.12 glob から正規表現への変換器

```

type GlobIRParser = M.Parsec Void String

data GlobIRToken = GlobIRSymbol String
                 | GlobIRLiteral String

instance Show GlobIRToken where
  show (GlobIRSymbol s) = s
  show (GlobIRLiteral s) = s

type GlobIR = [GlobIRToken]

ast, que, lbr, rbr, exc, word, br, expr, wildcard :: GlobIRParser GlobIR

wildcard = M.choice [
  mempty <$ M.eof
  , (<>) <$> expr <*> wildcard
]

ast = [GlobIRSymbol ".*"] <$ M.single '*'
que = [GlobIRSymbol "."] <$ M.single '?'
lbr = (:[]) . GlobIRSymbol . (:[]) <$> M.single '['
rbr = (:[]) . GlobIRSymbol . (:[]) <$> M.single ']'
exc = [GlobIRSymbol "^"] <$ M.single '^'

word = (:[]) . GlobIRLiteral . escapeRegular
      <$> M.some (M.noneOf [ '*', '?', '[', ']', '!', ' ' ])

```

```
br = (\x y z -> x <> y <> z)
  <$> lbr
  <*> M.choice [ mappend <$> exc <*> word, word ]
  <*> rbr

expr = M.choice [
  br
  , ast
  , que
  , word
]
```

後は、リスト 7.13 のように glob から正規表現へ変換するトランスパイラとマッチのための関数を用意すれば完成です。

▼リスト 7.13 glob から正規表現へのトランスパイラとマッチ関数

```
stringify :: (Foldable t, Show a) => t a -> String
stringify = concatMap show

transpile :: MonadThrow m => String -> m GlobIR
transpile = fromMonad (Nothing :: Maybe Void)
  . M.runParser wildcard mempty

match :: String -> GlobIR -> Bool
match s = (s ==-)
  . stringify
  . ([GlobIRSymbol "~"] <>) . (<> [GlobIRSymbol "$"])

match' :: String -> String -> Bool
match' s = maybe False (match s)
  . transpile
```

リスト 7.13 の関数を使って `ls-files` コマンドが作成できます*16。

▼リスト 7.14 `ls-files` コマンドの核部分

```
newtype LsFiles m = LsFiles { lsFiles :: LsFilesCfg -> HMGitT m () }
newtype LsFilesCfg = LsFilesCfg { lsFilesPathsSpecs :: [String] }

lsFilesBase :: (MonadCatch m, MonadIO m, Alternative m)
=> (IndexEntry -> FilePath -> HMGitT m ())
-> LsFiles m
lsFilesBase printer = LsFiles $ \lsFilesCfg ->
  let pat' = if null (lsFilesPathsSpecs lsFilesCfg)
    then ["."]
    else lsFilesPathsSpecs lsFilesCfg in do
    cDir <- P.getCurrentDir
    loadIndex
    >>= mapM_ (\e ->
      (pathsSpecs cDir (P.Rel (iePath e)) pat'
        >>= printer e) 'catchAny' const (pure ()))
```

*16 リスト 7.14 では、リスト 7.13 の関数を使って作られた、`pathsSpecs`、`lsMatches`関数を用いています (cf. <https://falgon.github.io/hmgit/HMGit-Internal-Parser-PathsSpecs.html>)。この関数のテストは <https://github.com/falgon/hmgit/blob/v1.0.0-gijutsushoten11/test/Spec.hs> のとおりで、`stack test`により実行可能です。

ファイルモード 100664

Git の生成する index ファイルを眺めていたところ、元ファイルのファイルモードとどうしても合致しない箇所がありました。

```
$ echo hello > hello
$ printf "%o\n" 0x$(stat -c %f hello)
100664
$ git add hello && hmgit add hello
$ diff -y -W 64 \
  <(xxd -l 2 -s 38 -c 2 .git/index) \
  <(xxd -l 2 -s 38 -c 2 .hmgit/index)
00000026: 81a4 .. | 00000026: 81b4 ..
```

Git の出力する index ファイルの 39、40 バイト目のファイルモードが 0x81a4 (100644) となっており、元のファイルモードとは異なるのです。これについて調べてみると、100664 ファイルモードは歴史的経緯上のミス (cf. <https://github.com/libgit2/libgit2sharp/blob/df3b22a754ef56da8d7e3c330ce2d783c2b7982e/LibGit2Sharp/Mode.cs#L25-L32>) であり、現在は非推奨とされているようで、Git の内部で読み替えているようです。インデックスのデータフォーマットに関する文書には、次のように示されています。

9-bit unix permission. Only 0755 and 0644 are valid for regular files.

差分の取得

次に、差分を取得できるようにします。単に、各ファイルについて、ワーキングディレクトリ内の最新のオブジェクト ID とすでにインデックスに追加済みのオブジェクト ID を比較すればよいでしょう。

▼リスト 7.15 差分を取得する関数

```
latestBlobHashes :: (MonadIO m, MonadCatch m)
=> HMGitT m (ML.Map (P.Path P.Rel P.File) String)
-- 略

indexedBlobHashes :: (MonadIO m, MonadCatch m)
=> HMGitT m (ML.Map (P.Path P.Rel P.File) String)
-- 略

getStatus :: (MonadIO m, MonadCatch m) => HMGitT m HMGitStatus
getStatus = do
  latest <- latestBlobHashes
  indexed <- indexedBlobHashes
  pure $ HMGitStatus {
```

```
statusChanged = ML.keysSet
  $ ML.filter (not . null)
  $ ML.intersectionWith (\l r -> if l /= r then r else mempty) latest indexed
, statusNew = ML.keysSet
  $ latest 'ML.difference' indexed
, statusDeleted = ML.keysSet
  $ indexed 'ML.difference' latest
}
```

あとはこれらをよしなに呼び出せば、diffコマンドと statusコマンドの完成です。リスト 7.13 の関数を使うことで pathspec にも対応できます*17。

コミット

いよいよコミットできるようにします。Tree オブジェクトを生成する関数を用意して、ページ 70 に記載の形式で Commit オブジェクトを生成する関数を用意すれば完成です*18。

▼リスト 7.16 Tree オブジェクトと Commit オブジェクトの生成

```
storeTree :: (MonadIO m, MonadCatch m) => HMGitT m B.ByteString
storeTree = loadIndex
  >>= storeObject Tree . foldMap (BP.runPut . putter)
  where
    putter e = BP.putLazyByteString (fromString (printf "%o %s\0" (ieMode e)
      (P.toFilePath $ iePath e)))
      *> BP.putLazyByteString (ieShal e)

commitDefault :: (MonadIO m, MonadCatch m) => Commit m
commitDefault = Commit $ \ccfg -> do
  -- 略
```

7.3 終わりに

本章では、Git のしくみと構造の理解から始まり、冒頭で示した 8 つのコマンドを実装してみました。Git はまだまだ奥が深く、多種多様な機能があります。それらのより実用的な機能を自作 Git に加えていくのも、おもしろいかもしれません。

本章が、Git と Haskell により興味を持ってもらうためのきっかけとなれば幸いです。

サンプル実装：HMGit

冒頭で示したサンプル実装は、次のようにビルドします。

*17 cf. <https://falgon.github.io/hmgit/HMGit-Commands-Porcelain-Diff-Core.html>

*18 cf. <https://falgon.github.io/hmgit/HMGit-Commands-Porcelain-Commit-Core.html>

```

$ git clone git@github.com:falgon/hmgit.git -b v1.0.0-gijutsushoten11 && cd hmgit
$ stack install
$ hmgit --version

  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
--  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /

Version: 1.0.0
Commit hash: 16409c9eb0ba8b163fac434a462de8de0f7ca517

```

--helpオプションを指定すると、コマンドの一覧とその機能の説明を表示します。

```

$ hmgit --help

  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
--  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /

Usage: hmgit [--version] [--db-name <database name>] COMMAND
The subset of awesome content tracker Git
Version: 1.0.0
Commit hash: 16409c9eb0ba8b163fac434a462de8de0f7ca517

Available options:
-h,--help              Show this help text
--version              Prints the HMGit suite version that the hmgit program
                      came from.
--db-name <database name>
                      hmgit database name

Available commands:
add                    Add file contents to the index
cat-file               Provide content or type and size information for
                      repository objects
diff                   Show changes between commits, commit and working
                      tree, etc
hash-object             Compute object ID and optionally creates a blob from
                      a file
init                   Create an empty Git repository or reinitialize an
                      existing one
ls-files               Show information about files in the index and the
                      working tree
status                 Show the working tree status
commit                 Record changes to the repository

```

各コマンドについても同様です。

```

# e.g. diff
$ hmgit diff --help
Usage: hmgit diff [--quiet] [--no-prefix] [--src-prefix <prefix>]
                [--dst-prefix <prefix>] [<path>...]
Show changes between commits, commit and working tree, etc

Available options:
-h,--help              Show this help text
--quiet                Disable all output of the program.
--no-prefix             Do not show any source or destination prefix.
--src-prefix <prefix> Show the given source prefix instead of "a/".
--dst-prefix <prefix> Show the given destination prefix instead of "b/".
<path>...              The <paths> parameters, when given, are used to limit
                      the diff to the named paths (you can give directory
                      names and get diff for all files under them).
-h,--help              Show this help text

```


第7章 ミニマル Git を自作しよう

initコマンドで、リポジトリの初期化を行います。--db-nameオプションにより、リポジトリデータベースの名前を決めることができます。

```
$ hmgit --db-name .git init test-hmgit && cd test-hmgit
Initialized empty HMGit repository in: test-hmgit
```

何か適当に追加して、リポジトリの状態を確認します。

```
$ echo hello git > hello
$ hmgit --db-name .git status
New files:
  hello
```

追加したファイルのステージングを行います。

```
$ hmgit --db-name .git add hello
$ hmgit --db-name .git status
```

コミット（とプッシュ）と行います。

```
$ hmgit --db-name .git commit -m 'hello git commit' --author "roki <roki@example.com>"
[main (commit) ec891bf] hello git commit

# git を使えばそのまま push できます
$ git remote add origin git@github.com:falgon/hmgit-playground.git
$ git push origin main
Counting objects: 3, done.
Writing objects: 100% (3/3), 211 bytes | 211.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:falgon/hmgit-playground.git
 + 41dab4a...ec891bf main -> main
```

何か適当に変更を加えて、差分を確認します。

```
$ echo meow >> hello
$ hmgit --db-name .git status
Changes files:
  hello
$ hmgit --db-name .git diff
--- a/hello
+++ b/hello
@@
  hello git
+meow
```

オブジェクトを確認します。

```
$ find .git/objects -type f | \
  cut -d/ -f3,4 | \
  sed 's/$/$$g' | \
  xargs -n1 hmgit --db-name .git cat-file -t
commit
tree
blob
$ find .git/objects -type f | \
  cut -d/ -f3,4 | \
  sed 's/$/$$g' | \
  xargs -n1 hmgit --db-name .git cat-file -s
163
33
10
$ find .git/objects -type f | \
  cut -d/ -f3,4 | \
  sed 's/$/$$g' | \
  xargs -n1 hmgit --db-name .git cat-file -p
tree 66eea8c80abea0e9836aab458e48ab9a379186e5
author roki <roki@example.com> 1621539748 +0900
committer roki <roki@example.com> 1621539748 +0900

hello git commit
100644 blob 8d0e41234f24b6da002d962a26c2495ea16a425f    hello
hello git
```

オブジェクト ID とオブジェクトを生成します。

```
$ echo bow | hmgit --db-name .git hash-object --stdin
de03f25c7324281fc5b6f146fe273fa85b689690
```

pathspec によるパターンマッチでリポジトリの状態を確認します。

```
$ mkdir -p foo/hoge/piyo foo/hoge/bar
$ echo "main = pure ()" > foo/hoge/piyo/a.hs
$ echo "main = pure ()" > foo/hoge/bar/b.hs
$ echo "main = pure ()" > c1.hs
$ hmgit --db-name .git status '*.hs'
New files:
  foo/hoge/piyo/a.hs
  foo/hoge/bar/b.hs
  c1.hs
$ hmgit --db-name .git status '?1.hs'
New files:
  c1.hs
```

ステージング済みファイルを確認します。

```
$ hmgit --db-name .git add .
$ hmgit --db-name .git ls-files '*.hs'
c1.hs
foo/hoge/bar/b.hs
foo/hoge/piyo/a.hs
$ hmgit --db-name .git ls-files -s '*.hs'
100644 76a9bdb5d48831da2e59c811ecbccb5a379bdfb5 0    c1.hs
100644 76a9bdb5d48831da2e59c811ecbccb5a379bdfb5 0    foo/hoge/bar/b.hs
100644 76a9bdb5d48831da2e59c811ecbccb5a379bdfb5 0    foo/hoge/piyo/a.hs
```

第 8 章

Makefile に秘められた真の力を開放する

Daisuke Makiuchi / @makki_d

皆さんは Makefile や make コマンドをどれだけ活用しているでしょうか。OSS をビルドするとき `configure` や `CMake` で Makefile を生成し、`make` コマンドを叩いたことのある人は多いでしょう。そのほか、とある界限では簡単な Makefile を記述して `make` をコマンドランチャーとして使うのが流行ったこともありました。しかし、それだけでは `make` と Makefile が本来もつ力をちっとも活かしていません。

`make` の本来の機能は、Makefile へ記述された依存関係にしたがって、再生成が必要なファイルだけを効率よく生成してくれるものです。この生成というのはソースコードのコンパイルに限りません。Makefile を工夫して書くことで、多数のファイルを順に変換するような作業を格段に効率化できます。

本章では、複雑な依存関係を記述できる、Makefile の強力な機能を紹介します。なお、ここで扱う `make` は GNU Make とし、バージョン 4.2.1 で動作確認しています。

8.1 Makefile の基本

Makefile にはファイル生成のルールをリスト 8.1 のように記述していきます。

▼リスト 8.1 Makefile のファイル生成ルール

```
ターゲット: ソース1 ソース2
  生成コマンド1
  生成コマンド2
```

ターゲット (生成されるファイル) を行頭に書き、「:」を挟んだうしろに**ソース** (元となるファイル) を列挙します。そして次の行からタブ字下げして**生成コマンド**を書いています。この字下げは必ずハードタブでなければなりません。

ターゲットを生成するには「make ターゲット」のように実行します。ターゲットの指定を省略すると、Makefile の中で最初に定義されたターゲットを指定したことになります。このとき、ターゲットのタイムスタンプよりもソースのいずれかが新しいときに、生成コマンドが実行されます。

さらにこのルールは多段にもできます。

▼リスト 8.2 多段ルールの Makefile

```
ターゲット: 中間物 1 中間物 2
            ターゲット生成コマンド

中間物 1: ソース 1
          中間 1 生成コマンド

中間物 2: ソース 2
          中間 2 生成コマンド
```

リスト 8.2 のような Makefile で `make` を実行すると、`make` は「ターゲット」の元となる「中間物 1」「中間物 2」のファイルとルールを探します。このとき「ソース 2」だけが更新されていた場合「中間 2 生成コマンド」を実行して「中間物 2」を更新しますが、`make` は賢いので「中間物 1」が「ソース 1」より新しいのであれば中間 1 生成コマンドは実行しません。そして「中間物 2」が「ターゲット」より新しくなるので、「ターゲット生成コマンド」が実行されます。

このように、ルールを丁寧に記述しておけば、最小限のコマンド実行で最新の「ターゲット」を生成できるようになります。

8.2 ファイル名のパターンを使ったルール

すべてのファイルについてルールを記述するのは大変ですが、ターゲットのファイル名から単純にソースのファイル名が決まるような、たとえば拡張子が変わるだけの場合などでは、「%」をワイルドカードとしたパターンルールが使えます。

▼リスト 8.3 パターンルールの例

```
%.pb.go: %.proto
        protoc --go_out=. $<
```

リスト 8.3 は Protocol Buffers の *.proto ファイルから Go 言語のコード *.pb.go を生成するパターンルールです。「make user.pb.go」のようにターゲットを指定すると、「user.proto」をソースとしてターゲットを生成するルールとして働きます。コマンドの中でターゲットやソースのファイル名を使うには「\$@」や「\$<」のような自動変数を利用します。

▼表 8.1 主な自動変数

<code>\$\$</code>	ターゲット名
<code>\$\$<</code>	ソースの先頭のもの
<code>\$\$^</code>	すべてのソース
<code>\$\$?</code>	ソースのうちターゲットより新しいもの
<code>\$\$*</code>	%に一致した部分文字列

8.3 ファイルの内容からルールを生成する

これまで紹介したように、Makefile ではソースのファイル名と生成されるファイル名によってルールを記述します。ここではさらに発展した例として、ファイルの内容からルールを生成する方法を紹介します。まずはリスト 8.4 をご覧ください。

▼リスト 8.4 ファイルの内容からルールを生成

```

STRINGERS := $(shell grep -r '^//go:generate stringer ' . | \
sed -E 's/^([:]*):.*-type=([^\ ]*)(.*)?$$/\1>\2/g')

define stringer_rule
$(eval params := $(subst '<', , $1))
$(eval source := $(word 1,$(params)))
$(eval name := $(shell echo $(word 2,$(params)) | tr A-Z a-z)_string.go)
$(eval target := $(dir $(source)))$(name)
$(target): $(source)
    go generate $(source)
endif

$(foreach s,$(STRINGERS),$(eval $(call stringer_rule,$(s))))

```

これは Go 言語の `stringer` というコード生成ユーティリティのためのルールを動的に生成する Makefile です。`stringer` を利用するには Go のソースコードにリスト 8.5 のようなコメントを書いておき、`go generate` コマンドを呼び出すことでコードが生成されます。

▼リスト 8.5 `go generate` の `stringer` の書式の例

```
//go:generate stringer -type=MyEnum
```

ここでは `stringer` の詳細は省きますが、このコメントの書かれたファイルがソースになります。そして出力されるファイル名は、`-type` で指定された型名を小文字にして `_string.go` を付けたもの、この例の場合は `myenum_string.go` になります。つまり、リスト 8.5 の書かれたファイルを検索し、書かれている型名から生成されるファイル名を構築すればルールを生成できます。そしてそのルールを `make` に認識させれば、必要なときだけコマンドを実行する効率のよい Makefile となります。それでは順番に見ていきましょう。

ソースの検索と型名の抽出

▼リスト 8.6 ソースの検索と型名の抽出

```
STRINGERS := $(shell grep -r '^//go:generate stringer ' . | \
sed -E 's/^([:]*):.*-type=([ ]*)(.*)?$$/\1>\2/g')
```

Makefile にはさまざまな関数が用意されていて、`$(function param,param,...)`の形で呼び出せます。ここで使っているのは `shell` 関数です。その名から分かるとおりシェルコマンドを呼び出し、標準出力の文字列に展開されます。

ここではまず `grep` で「`//go:generate stringer`」を含むファイル名とその行を抽出しています。つづいてパイプで `sed` に流し込み、`ファイル名>型名`の形に編集しています。たとえばリスト 8.5 の書かれたファイル `myprogram.go` がある場合、「`./myprogram.go>MyEnum`」のようになります。ファイルが複数ある場合コマンドの出力は複数行になりますが、`shell` 関数はそれを空白文字区切りのリストとして展開します。こうして得られたリストを `STRINGERS` 変数に格納しています。この変数はあとで `$(STRINGERS)` と書くことで展開できます。

ルールのテンプレート

続いて、`define~endif` の部分です。Makefile では変数への値の格納は `:=` などによる代入が一般的ですが、GNU Make では `define` を使うことで、複数行にまたがるような文字列も変数へ格納できます。

▼リスト 8.7 テンプレートの定義

```
define stringer_rule
$(eval params := $(subst '<', , $1))
$(eval source := $(word 1, $(params)))
$(eval name := $(shell echo $(word 2, $(params)) | tr A-Z a-z)_string.go)
$(eval target := $(dir $(source))$(name))
$(target): $(source)
    go generate $(source)
endif
```

リスト 8.7 では「`stringer_rule`」という名前でもテンプレートとして使用する変数を定義しています。この変数はあとで `call` 関数で呼び出します。

▼リスト 8.8 call 関数

```
$(call variable,param,param,...)
```

`call` 関数は、変数 `variable` にパラメータを与えて展開します。各パラメータには `$1`、`$2`……の形でアクセスできます。

`stringer_rule`のパラメータには、さきほど `STRINGERS`変数に格納した「`./myprogram.go>MyEnum`」を渡します。まずは `subst`関数でパラメータ\$1の`>`を空白に置換し、リストの形にして `params`に保存します。ここで、`call`関数によるテンプレートの展開の時点では、`:=`を含む文字列を生成するだけで代入は行われません。このため、`eval`関数で評価することで変数への代入を実行します。

`params`をリストにしたことで、`word`関数でファイル名と型名を取り出せます。1 番目がファイル名「`./myprogram.go`」、2 番目が型名「`MyEnum`」となっています。ソースとなるファイル名はこれで取り出せます。

続いて、リスト 8.9 ではターゲットとなるファイル名を型名から生成します。

▼リスト 8.9 ターゲットファイル名の生成

```
$(eval name := $(shell echo $(word 2,$(params)) | tr A-Z a-z)_string.go)
```

Make 自体に小文字へ変換する関数がみあたらないので、`shell`関数で `tr`コマンドを呼び出すことにし、後ろに「`_string.go`」を結合して `name`変数に格納しています。実は、Linux など GNU の `sed`であれば型名を抽出する段階で `\L`で小文字変換できるのですが、macOS の `sed`にはそのような機能がないので、ここでは `tr`コマンドを利用しています。

`stringer`ではソースと同じディレクトリにファイルを生成するので、`dir`関数を使ってソースのディレクトリ名を取り出しファイル名と結合してターゲット名とします。

ここまでの処理をまとめると、パラメータとして「`./myprogram.go>MyEnum`」が渡された時、テンプレート `stringer_rule`はおおよそリスト 8.10 ように展開されます。

▼リスト 8.10 展開されたテンプレート

```
params := ./myprogram.go MyEnum
source := ./myprogram.go
name := myenum_string.go
target := ./myenum_string.go
./myenum_string.go: ./myprogram.go
go generate ./myprogram.go
```

すべての対象でテンプレート展開

▼リスト 8.11 すべての対象にテンプレートを適用

```
$(foreach s,$(STRINGERS),$(eval $(call stringer_rule,$(s))))
```

最後にリスト 8.11 の行では、最初に検索した `STRINGERS`の各要素に対して、`call`関数と `eval`関数を呼び出しています。

▼リスト 8.12 foreach 関数

```
$(foreach var,list,text)
```

foreach関数は、listに与えられた空白文字区切りのリストのそれぞれの要素ついて、textを適用していきます。textの中では処理中の要素はvarの名前でアクセスできます。

最初に検索した STRINGERSは「./dir1/program1.go>Enum1 ./dir2/program2.go>Enum2」のような空白文字区切りのリストになっています。これがforeachによって、リスト 8.13 のように適用されることになります。

▼リスト 8.13 foreach の適用イメージ

```
$(eval $(call stringer_rule,./dir1/program1.go>Enum1))
$(eval $(call stringer_rule,./dir2/program2.go>Enum2))
```

そして call関数によってテンプレートが展開されるとリスト 8.14 になります。

▼リスト 8.14 call による展開イメージ

```
$(eval
./dir1/enum1_string.go: ./dir1/program1.go
    go generate ./dir1/program1.go
)
$(eval
./dir2/enum2_string.go: ./dir2/program2.go
    go generate ./dir2/program2.go
)
```

call関数で展開されたものはまだただの文字列ですので、これを eval関数で評価することで、makeのルールとして認識されます。

このようにして、ファイル内の stringerの書式からルールを生成できました。これで make実行時に効率よく必要なファイルだけ go generateを呼び出すようになりました。

8.4 まとめ

Makefile にはこの他にも ifeqのような条件文やさまざまな関数、make自体を再帰的に使う方法などたくさんの構文があり、一記事ではとても紹介しきれないほど高機能です。使いこなせばあらゆる状況で効率よくファイルを生成するルールを自在に記述できるでしょう。

しかし、なんでも Makefile でやってしまうのは可読性の面からもお勧めできません。この記事で紹介した stringerのルール生成を初見で理解できる人はそういないと思います。

とはいえ、シンプルなルールを書くだけでも役に立つ場面はきっと多いことでしょう。Makefile をうまく使って、日々の作業を効率化していきましょう。

執筆者・スタッフコメント

第 1 章 Mitsuteru Furukawa / @ttt_333

個人的に興味がある UD-701N の発売はいつまで延期されるのだろうか・・・

第 2 章 Atsushi Takada

原稿のかなりの部分を GPT-2 に書かせたおかげで執筆が楽でした

第 3 章 Keita Yamamoto

/* おばあちゃんの遺言でコメントを書きません。 */

第 4 章 Shunsuke Ito

CPython の実装を読むと C 言語とも仲良くなれる……かもしれない

第 5 章 Toshifumi Umezawa

家の中のあれこれを自前で IoT 化するのが目標

第 6 章 Shinya Naganuma / @Pctg_x8

TypeScript より PureScript が好きです

第 7 章 Hiroki Gomi / @530506

「実装したからといって、理解したとは限らない」

第 8 章 Daisuke Makiuchi / @makki_d

眼鏡っ娘が好きです

企画進行・イラスト・デザイン

おが

表紙デザイン担当しました。楽しく制作できました！

あつ

vol.2 に出てきた眼鏡っ子をドット化しました！ 可愛い表紙でお気に入りです。

既刊・電子版ダウンロード

<https://www.klab.com/jp/blog/tech/2021/tbf11.html>



KLab Tech Book Vol. 8

2021年7月10日 技術書典11版(1.0)

著者 KLab 技術書サークル

編集 梅澤 寿史、牧内 大輔

発行所 KLab 技術書サークル

印刷所 日光企画

(C) 2021 KLab 技術書サークル



KLab Tech Book

クラブテックブック

Vol. 8

2021 SUMMER