KLab Tech Book

VOL.12

クラブ テックブック

1 GitHub Actionsをローカル環境で実行する「act」

2 Google Cloud を用いた Serverless な Slack Botの作り方

3 Python の match 文に詳しくなってみましょう

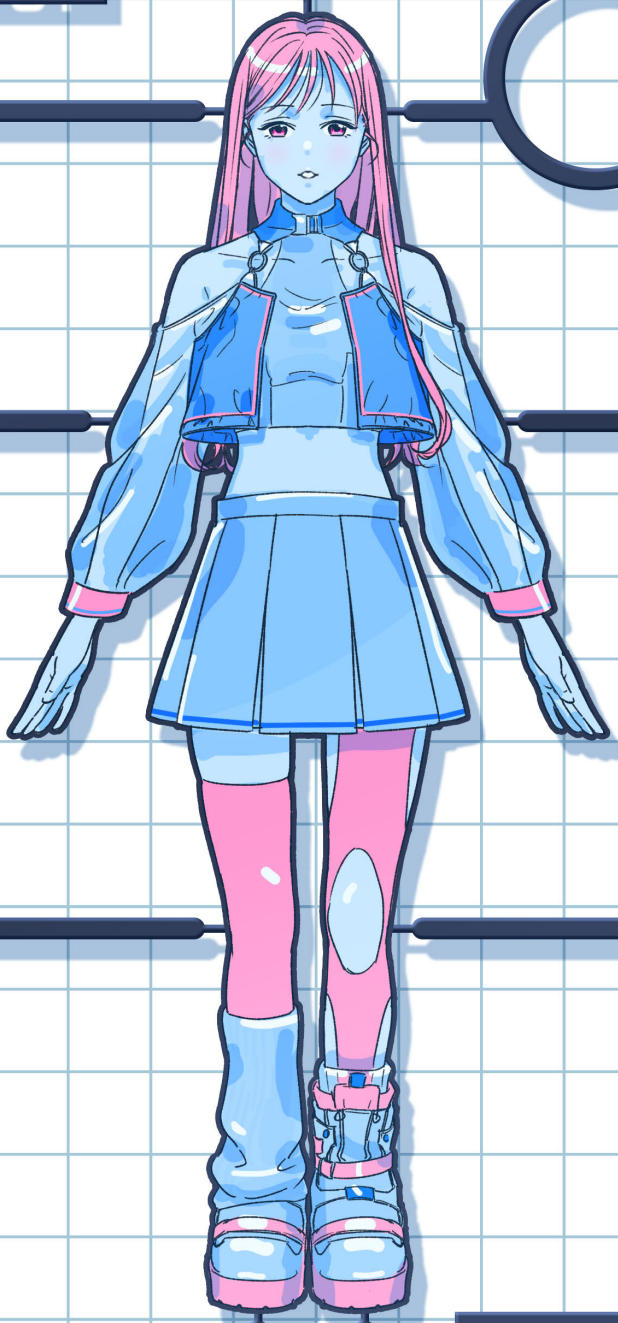
4 アニメーションGIFとファイルサイズの話

5 GPUメモリアロケーター自作入門

02

03

01



04

05

06

KLabTechBook

KLab Tech Book Vol. 12

KLab 技術書サークル　著

2023-11-11 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 12 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、本書は社内の有志のエンジニアが業務との関連によらず各自好きな内容を執筆しています。表紙や扉絵などにおいては社内のデザイナーの方にもご協力いただき、KLab の雰囲気を詰め込んだ一冊に仕上げています。

今回は 5 人のエンジニアが記事を執筆しました。章ごとに内容が独立しているので、気になるものから順に読み進めてもらって問題ありません。どの記事も 10 ページ程度となっており、伝わりやすいように心がけながら執筆しているので、知らない分野であっても軽い気持ちで読んでいただくだけで世界が広がるかもしれません。

本書を通して、技術や知識に触れる楽しさを感じていただけたらさいわいです。

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

第 1 章

GitHub Actions をローカル環境で実行する「act」

Daisuke Makiuchi / @makki_d

1.1 GitHub Actions と act

GitHub Actions は GitHub に統合された CI/CD プラットフォームです。リポジトリへの Push や Pull Request などのイベントと関連付けて、自動テストやビルド、デプロイといったさまざまなワークフローを自動実行できます。

しかし、GitHub Actions のワークフローを起動するには、コードをコミットしてリポジトリに Push しなければなりません。このため、新しいワークフローを作成するときなど、動作確認のために毎回 Push する必要があり煩雑です。また、単純なエラー、例えばタイポやコーディングスタイルのミスなどは、リポジトリに Push する前に検出したいところです。

そこで登場するのが「act」というツールです[*1]。act を使えば GitHub Actions のワークフローの各ジョブをローカル環境で実行でき、リポジトリへコミットや Push をすることなく、事前にエラー検知できるようになります。これにより、大人数での開発プロジェクトで起こりがちなジョブのランナーの枯渇も軽減できるでしょう。

この章では、GitHub Actions をローカル環境で実行する「act」について、基本的な使い方と制限事項、便利に使うための Tips を紹介します。

[*1] `https://github.com/nektos/act`

1.2 act のインストール

Docker の用意

act は GitHub Actions のジョブのランナーを Docker コンテナによって再現するため、事前に Docker をインストールしておく必要があります。Docker Desktop、またはその他の Docker 環境をセットアップしてください。

act のインストール

act のインストールはパッケージマネージャを使うと簡単です。macOS や Linux でよく使われている Homebrew、Windows の Chocolatey や Winget の他、多くのパッケージマネージャに登録されています[*2]。ご自身の環境にあわせて選ぶとよいでしょう。リスト 1.1 とリスト 1.2 に Homebrew と Chocolatey の例を示します。

リスト 1.1: Homebrew によるインストール（Linux、macOS）

```
brew install act
```

リスト 1.2: Chocolatey によるインストール（Windows）

```
choco install act-cli
```

また、act は Go 言語で実装されているため、Go の開発環境が整っている場合はリスト 1.3 のようにインストールすることもできます。

リスト 1.3: Go によるインストール

```
go install github.com/nektos/act@latest
```

[*2] Homebrew、MacPorts、Chocolatey、Scoop、Winget、AUR、COPR、Nix など

コンテナイメージの準備

act がインストールできたのでさっそく動かしたいところですが、その前に使用するコンテナイメージを準備しましょう。act によって自動生成される設定では若干不都合があるので、リスト 1.4 のように設定ファイル `.actrc` を用意します[*3]。

リスト 1.4: `.actrc`

```
-P ubuntu-latest=ghcr.io/catthehacker/ubuntu:act-latest
-P ubuntu-22.04=ghcr.io/catthehacker/ubuntu:act-22.04
-P ubuntu-20.04=ghcr.io/catthehacker/ubuntu:act-20.04
```

このファイルは act 実行時に毎回指定するコマンドラインオプションを記載するもので、`-P`（`--platform`）オプションでジョブを実行するコンテナのイメージを指定しています。

また、act の実行によってコンテナイメージをダウンロードした場合、その進捗が表示されません。あらかじめ `docker` コマンドでダウンロードしておく方が安心できます。

リスト 1.5: docker コマンドでのイメージのダウンロード

```
docker pull ghcr.io/catthehacker/ubuntu:act-latest
docker pull ghcr.io/catthehacker/ubuntu:act-22.04
docker pull ghcr.io/catthehacker/ubuntu:act-20.04
```

[*3] Linux や macOS では`$HOME`（つまり`/home/ユーザ名/`）、Windows では `%USERPROFILE%`（つまり `C:\USER\ユーザ名\`）に配置します

■コラム: デフォルトのコンテナイメージ

コンテナイメージを何も指定せずにジョブを実行しようとすると、図 1.1 のようにデフォルトのコンテナイメージを選択する画面が表示されます。

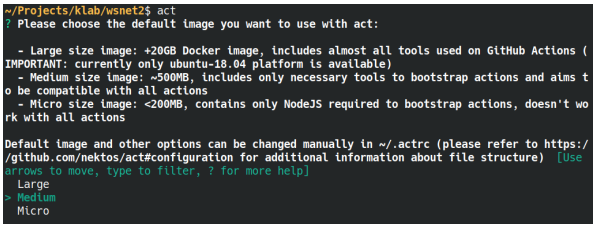


図 1.1: デフォルトイメージの選択画面

act version 0.2.52 において、この表示内容は古く、実際のイメージと乖離しています。

- **Large**: 利用されうるほぼすべてのツールがインストールされていますが、サイズは 20GB どころか 50GB 以上あります。また現在は Ubuntu 20.04 と 22.04 がサポートされています。
- **Medium**: アクションの起動に必要なツールのみインストールしてサイズは 1.5GB 以下に抑えられています。ほとんどのアクションが動作します。
- **Micro**: 表示のとおり、アクションを起動するための Node.js だけのイメージです。サイズも 200MB 未満です。

これらのイメージは Docker Hub にホストされています。

Large	`catthehacker/ubuntu:full-latest` など
Medium	`catthehacker/ubuntu:act-latest` など
Micro	`node:16-buster-slim` など

act はジョブ実行時に最新のイメージがあるかを確認して自動的に pull するのですが、このとき Docker Hub のダウンロード率制限[*4]に引っかかることがあります。`catthehacker/ubuntu` のイメージと同じものが GitHub コンテナレジストリにも登録されているので、`ghcr.io/catthehacker/ubuntu` のイメージを指定することで、この制限を回避できます。

[*4] https://matsuand.github.io/docs.docker.jp.onthefly/docker-hub/download-rate-limit/

1.3 act の実行

ジョブの一覧

それでは、act を使ってみましょう。最初にジョブ一覧を表示する`-l`（`--list`）オプションを紹介します。リポジトリのルートディレクトリで `act -l` としてみます。

```
~/Projects/klab/wsnet2$ act -l
Stage  Job ID         Job name       Workflow name        Workflow file         Events
0      gopherjs       gopherjs       WSNet2 dashboard ci  wsnet2-dashboard.yml  pull_request
0      dotnet         dotnet         WSNet2 dotnet ci     wsnet2-dotnet.yml     pull_request
0      unit-test      unit-test      WSNet2 server ci     wsnet2-server.yml     pull_request
0      combined-test  combined-test  WSNet2 server ci     wsnet2-server.yml     pull_request
```

図 1.2: ジョブ一覧

これは KLab の OSS、WSNet2[*5]のジョブ一覧です。リポジトリの`.github/workflow`以下の yaml ファイルを読み取り、一覧化してくれます。

act でジョブを実行するときは、ここに表示されている Job ID や Workflow file、Events の値を指定することになるので、このコマンドで確認するとよいでしょう。

ジョブの実行

`act` をパラメータなしで実行すると、すべてのジョブが実行されます。しかし、これではログが見にくいことに加え、ローカルでの確認では GitHub のイベントや Job ID を指定して必要なジョブだけ実行したいケースが多いでしょう。

たとえば、WSNet2 で.NET アプリのビルドとテストを行う `dotnet` ジョブを実行するにはリスト 1.6 のようにします。

リスト 1.6: act で dotnet ジョブを実行

```
act pull_request -j dotnet -W .github/workflow/wsnet2-dotnet.yml
```

パラメータの `pull_request` でイベントを指定し、`-j`（`--job`）オプションでジョブを指定しています。この場合 `dotnet` ジョブを起動するイベントは `pull_request` しか定義されていないため、イベントの指定は省略できます。また、`-W`（`--workflows`）オプションは複

[*5] `https://github.com/KLab/wsnet2`

数の yaml ファイルでワークフローを定義しているときに、どの yaml ファイルのジョブかを識別するために必要になります。WSNet2 では wsnet2-dotnet.yml の他に wsnet2-dashboard.yml、wsnet2-server.yml が存在するため指定が必要です。

ここで、dotnet ジョブの中身をリスト 1.7 に示します。

リスト 1.7: wsnet2-dotnet.yml

```
 1: name: WSNet2 dotnet ci
 2:
 3: on:
 4:   pull_request:
 5:     branches: [ main ]
 6:     paths:
 7:       - '.github/workflows/wsnet2-dotnet.yml'
 8:       - 'wsnet2-dotnet/**'
 9:       - 'wsnet2-unity/**'
10:
11: jobs:
12:   dotnet:
13:     runs-on: "ubuntu-latest"
14:     defaults:
15:       run:
16:         working-directory: wsnet2-dotnet
17:     steps:
18:       - uses: actions/checkout@v3
19:       - name: Setup .NET
20:         uses: actions/setup-dotnet@v3
21:         with:
22:           dotnet-version: "6.x"
23:       - name: dotnet build
24:         run:  dotnet build WSNet2.sln
25:       - name: dotnet test
26:         run:  dotnet test WSNet2.sln
```

まず注目していただきたいのが 13 行目の runs-on: "ubuntu-latest"です。この行でジョブのランナーを指定しています。

ここで、.actrc に記載した-P ubuntu-latest=ghcr.io/catthehacker/ubuntu:act-latest を思い出してください。この-P オプションが、runs-on に指定された ubuntu-latest でのジョブの実行に使うコンテナイメージの指定になっています。

こうして指定されたイメージのコンテナを起動したら、act は yaml の 17 行目以降に指定された各ステップを順に実行していきます。

```
~/Projects/klab/wsnet2$ act pull_request -j dotnet -W .github/workflows/wsnet2-dotnet.yml
[WSNet2 dotnet ci/dotnet] 🚀  Start image=ghcr.io/catthehacker/ubuntu:act-latest
[WSNet2 dotnet ci/dotnet]   🐳  docker pull image=ghcr.io/catthehacker/ubuntu:act-latest platf
orm= username= forcePull=true
[WSNet2 dotnet ci/dotnet]   🐳  docker create image=ghcr.io/catthehacker/ubuntu:act-latest pla
tform= entrypoint=["tail" "-f" "/dev/null"] cmd=[]
[WSNet2 dotnet ci/dotnet]   🐳  docker run image=ghcr.io/catthehacker/ubuntu:act-latest platfo
rm= entrypoint=["tail" "-f" "/dev/null"] cmd=[]
[WSNet2 dotnet ci/dotnet]   ☁  git clone 'https://github.com/actions/setup-dotnet' # ref=v3
[WSNet2 dotnet ci/dotnet] ⭐ Run Main actions/checkout@v3
[WSNet2 dotnet ci/dotnet]   🐳  docker cp src=/home/makki/Projects/klab/wsnet2/. dst=/home/mak
ki/Projects/klab/wsnet2
[WSNet2 dotnet ci/dotnet]   ✅  Success - Main actions/checkout@v3
[WSNet2 dotnet ci/dotnet] ⭐ Run Main Setup .NET
[WSNet2 dotnet ci/dotnet]   🐳  docker cp src=/home/makki/.cache/act/actions-setup-dotnet@v3/
dst=/var/run/act/actions/actions-setup-dotnet@v3/
[WSNet2 dotnet ci/dotnet]   🐳  docker exec cmd=[node /var/run/act/actions/actions-setup-dotne
t@v3/dist/setup/index.js] user= workdir=
|  [command]/run/act/actions/actions-setup-dotnet@v3/externals/install-dotnet.sh --channel 6.0
|  dotnet-install: Attempting to download using aka.ms link https://dotnetcli.azureedge.net/dot
```

図 1.3: ジョブの実行

act は基本的には steps に定義されたアクションやコマンドをそのまま実行しますが、actions/checkout だけは例外です。デフォルトではローカルのファイルを docker cp でコンテナ内に転送します[*6]。これによりローカルの変更をリポジトリにコミットや Push しなくても、そのまま実行されるジョブの対象とできるわけです。

```
|
|  Passed!  - Failed:     0, Passed:   104, Skipped:     0, Total:   104, Duration: 1 s - /home
/makki/Projects/klab/wsnet2/wsnet2-dotnet/WSNet2.Core.Test/bin/Debug/net6.0/WSNet2.Core.Test.d
ll (net6.0)
[WSNet2 dotnet ci/dotnet]   ✅  Success - Main dotnet test
[WSNet2 dotnet ci/dotnet] ⭐ Run Post Setup .NET
[WSNet2 dotnet ci/dotnet]   🐳  docker exec cmd=[node /var/run/act/actions/actions-setup-dotne
t@v3/dist/cache-save/index.js] user= workdir=
[WSNet2 dotnet ci/dotnet]   ✅  Success - Post Setup .NET
[WSNet2 dotnet ci/dotnet] 🏁  Job succeeded
~/Projects/klab/wsnet2$
```

図 1.4: ジョブの完了

ジョブが完了すると、Job succeeded のログが表示されてコンテナも終了します。エラーが発生したときはコンテナが消えずに残るため、アタッチして原因を究明できます。もしエラー時にもコンテナを削除したい場合は--rm オプションを指定してください。

このほか詳しい使い方はユーザーガイド[*7]をご覧ください。

1.4 act の制限

ここまで紹介したように、act は Docker コンテナによってジョブの実行環境を再現します。GitHub がホストするランナーは Ubuntu だけでなく Windows や macOS も提供されていま

[*6] -b (--bind) オプションを指定すると、バインドマウントによってファイルを同期するようにできます。

[*7] https://nektosact.com/

すが、Docker コンテナは基本的には Linux であるため、act では Ubuntu のみをサポートします[*8]。加えて、GitHub が提供するランナーと完全に同じ環境ではないため、動かないアクションもあるかもしれません。

また、サービスコンテナやジョブのタイムアウトなどいくつか未実装の機能があります[*9]。

サービスコンテナについては、手動でコンテナを起動することで一応対処できます。ジョブ実行コンテナのネットワークモードは host になっているため、サービスコンテナ起動時に`-p`（`--publish`）オプションでポートをホストに公開しておくことで、ジョブ実行コンテナからも `localhost` の該当ポートでサービスコンテナにアクセスできます。

1.5 その他の Tips

ここで、act で使える便利な Tips をいくつか紹介します。

シークレットの受け渡し

GitHub 上で保存したシークレットの値や `GITHUB_TOKEN` は、当然ながら act からは参照できません。これらの値を必要とするジョブを実行するには、`-s`（`--secret`）または`--secret-file` オプションによって値を受け渡します。

このとき、コマンドラインで `act -s MY_SECRET=value` のように値を指定してしまうと、コマンド履歴ファイルなどにシークレットの値が保存されてしまう可能性があり好ましくありません。代わりに `act -s MY_SECRET` のようにして、コマンドラインでは値を指定せずにセキュアな対話インターフェイスで値を入力する方法が推奨されています[*10]。

act での実行ではスキップする

act で実行しているときは特定のジョブやステップをスキップしたいこともあるでしょう。act はジョブを実行するときに環境変数 `ACT=true` を設定します。これを利用して、リスト 1.8 のようにジョブやステップに `if` 条件文[*11]を書くことでスキップできます。

[*8] 筆者は試していませんが、実行環境が整っていれば Windows コンテナを使用して Windows ランナーを再現できるかもしれません。

[*9] `https://nektosact.com/not_supported.html`

[*10] このとき環境変数 `MY_SECRET` が定義されているとそちらが優先されることに注意が必要です。

[*11] `https://docs.github.com/ja/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idif`

リスト 1.8: act での実行時はスキップする

```
jobs:
  my-job:
    runs-on: ubuntu-latest
    steps:
      - run: echo "スキップされない"

      - if: ${{ !env.ACT }}
        run: echo "スキップされる"

      - run: echo "スキップされない"
```

セルフホステッドランナーへの対応

GitHub Actions では、自身で用意したマシンでジョブを実行するセルフホステッドランナーもサポートされています。セルフホステッドランナーでジョブを実行するには、`runs-on` に `self-hosted` を指定します。

act でセルフホステッドランナーに対応するには、まずランナーとなるコンテナイメージを用意します。そのうえで、act 実行時に`-P self-hosted=イメージ名` のようにコンテナイメージを指定します。このとき、act は指定のイメージの最新版がないかオンラインのコンテナリポジトリを確認するのですが、リポジトリ上にイメージが存在しない場合はエラーになってしまいます。ローカルにしかないイメージを使うときは`--pull=false` オプションを指定して、最新版の確認をスキップする必要があります。

`docker` コマンド使用時の注意点

act で実行されるコンテナ内でも `docker` コマンドを使用できます。ただし、接続する Docker デーモンはコンテナ内ではなく act を実行しているホストで動いているものです[*12]。このため、ポートなどのリソースの競合やバインドマウントするときのパスなどに注意する必要があります。

[*12] ホストの`/var/run/docker.sock` がコンテナにバインドマウントされています。

Docker で userns-remap を有効にしている場合

前巻 KLabTechBook Vol.11[*13] 第 1 章「Docker を使うなら当然 userns-remap してるよね！」で紹介したように、Linux で Docker を利用するときは `userns-remap` を有効にして一般ユーザーの名前空間でコンテナを動かすことをお勧めしています。

一方、act はコンテナをホストの名前空間で実行することを要求するので、コンテナ起動時に`--userns=host` オプションを指定する必要があります。act でのジョブ実行時のオプションに`--container-options --userns=host` を追加することでこれを実現できます。この設定を`.actrc` ファイルに記載しておくとよいでしょう。

1.6 おわりに

この章では、GitHub Actions をローカル環境で実行する「act」というツールについて、基本的な使い方を紹介しました。

GitHub Actions はそれ自体でも十分すぎるほど強力なツールですが、act を活用することでより効率的に使用できるでしょう。ぜひ活用してみてください。

[*13] 巻末に既刊ダウンロードページの URL があります。ぜひご覧ください

第 2 章

Google Cloud を用いた Serverless な Slack Bot の作り方

Daisuke Yamaguchi / @_gutio_

2.1 はじめに

この記事では、Google Cloud を用いた Serverless な Slack Bot の作り方を紹介します。

私は今まで Slack のメッセージをトリガーにする Bot を作る際には、Real Time Messaging API (RTM API) [*1] を利用してメッセージ受信用のストリームを使っていました。しかし、RTM API は Slack としてはレガシーなものとされており、今後は Events API の Event Subscriptions [*2] を利用することが推奨されています。

Event Subscriptions は、Slack の特定のイベントが発生した際に Slack から指定した URL へ向けて POST する仕組みです。この POST を受け取った際に処理できれば十分であり、RTM API のように Bot のプロセスを起動させ続ける必要はありません。このため、Serverless な環境でも Bot を作ることができます。そして Cloud Functions を使えば、Bot だけならほぼ無料で動かすことができます。

しかし、Event Subscription は 3 秒でタイムアウトしてしまうため、レスポンスに時間のかかるような Bot の場合は少し工夫が必要です。そのような工夫として、POST を受け取ったら一旦レスポンスを返し、そのあとで時間のかかるメイン処理を行うという方法も紹介します。

[*1] https://api.slack.com/rtm
[*2] https://api.slack.com/events-api

2.2 構成

最初に、Serverless に Bot を作る際に使用した構成を紹介します。

使っているもの

Slack：Event Subscriptions , OAuth API
Google Cloud：Cloud Functions (C#) , Cloud Tasks

Serverless な Slack Bot の基本的なコンセプト

1. 特定の条件の Slack メッセージを Slack Bot に転送する
2. 受信用の Cloud Functions で Slack からのメッセージを受け取り、メイン処理用の Cloud Functions を呼び出すために Cloud Tasks に登録する
3. Cloud Tasks がメイン処理用の Cloud Functions を呼び出す
4. メイン処理用の Cloud Functions で時間のかかる処理を実行し、Slack にレスポンスする

図にするとこのような形です

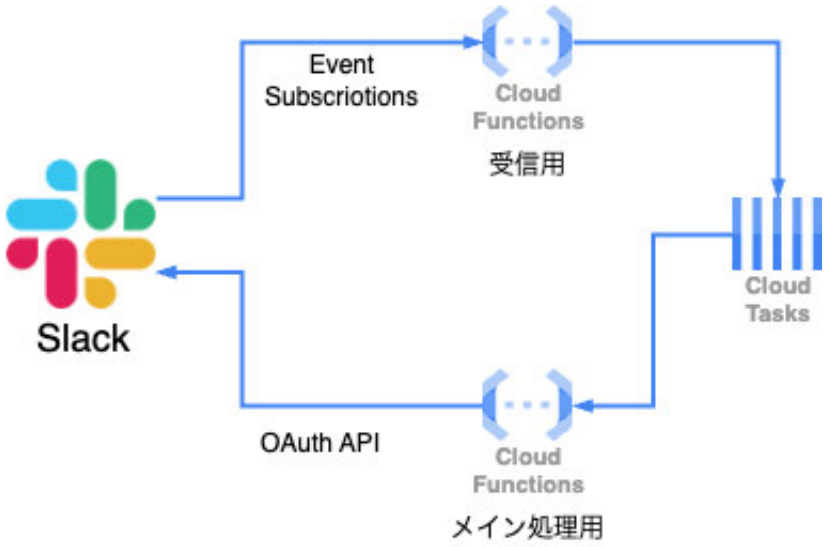


図 2.1: 構成概要

2.3 Slack App の作成

ここからは、実際に作成する手順を紹介します。まずは、Slack Apps の設定ページ[*3]で新しい App を作成します。任意のアプリ名と Workspace を指定して作成してください。

作成したら、Basic Information の App Credentials 項目にある Signing Secret を控えておいてください。また、OAuth & Permissions のページで、Bot Token Scopes にレスポンス投稿用の chat:write の権限を付与して、Bot User OAuth Token を控えておいてください。

2.4 メイン処理用の Cloud Functions の作成

アプリの実装の話に入りますが、メイン処理用の Cloud Functions から作成します。Slack から届くメッセージの正当性検証などは受信用の Cloud Functions で行うため、メイン処理用の Cloud Functions では実際に行いたい処理に集中して作成します。

Cloud Functions にデプロイしたら、Cloud Functions の URL を控えておいてください。

リスト 2.1: メイン処理用の Cloud Functions

```
// 略 imports
public class MainOperationFunction : IHttpFunction
{
    // Cloud Functions に登録するエントリポイント
    public async Task HandleAsync(HttpContext context)
    {
        // Bodyを読み出す
        string body = await Utilities.ReadStreamAsync(context.Request.Body);
        // NewtonsoftでJSONのパース
        var jsonObj = JObject.Parse(body);
        var channel = jsonObj["event"]["channel"].ToString();

        // 長い処理のダミー 10sec.
        await Task.Delay(10000);

        // Slackのチャンネルにレスポンスを返す
        await PostSlack("長い処理が終わったよ", channel);
    }

    private async Task PostSlack(string text, string channel)
    {
        const string apiUrl = "https://slack.com/api/chat.postMessage";
        // Slackに返答
```

[*3] https://api.slack.com/apps

```
        var httpClient = new HttpClient();
        // 控えておいた Bot User OAuth Tokenを入れること
        httpClient.DefaultRequestHeaders.Add(
            "Authorization", "Bearer " + "BotUserOAuthToken");
        var response = new
        {
            channel,
            text
        };
        await httpClient.PostAsJsonAsync(apiUrl, response);
    }
}
```

2.5 Cloud Tasks の作成

Cloud Tasks は、今回の構成だとメイン処理用の Cloud Functions の呼び出しを予約するために利用するサービスです。Cloud Tasks を有効にして 任意のキュー名とリージョンで Push Queue を作成します。

作成したキュー名、リージョン、プロジェクト ID を控えておいてください。

2.6 受信用の Cloud Functions の作成

次に、Slack からのメッセージを受け取る Cloud Functions を作成します。Event Subscriptions での Slack からのメッセージの受信処理は、いくつかの要件を満たす必要があります。

- 改ざんされていないことを検証するために、Slack からのヘッダに含まれる署名を検証する必要がある
- Event Subscriptions の設定ページで、URL を登録する際の専用メッセージに適切にレスポンスする必要がある
- Slack からの POST に 3 秒以内にレスポンスを返す必要がある

これらの検証と早期レスポンスを受信用の Cloud Functions で行うことで、メイン処理用の Cloud Functions では検証や実行時間の制約を気にする必要がなくなります。

署名の検証

Slack からの POST に含まれる X-Slack-Signature ヘッダには、検証用署名が含まれています。控えておいた Signing Secret と、Slack からの POST のボディ、X-Slack-Request-Timestamp ヘッダの 3 つの値を組み合わせてシグネチャを計算し、この値と検証用の署名を比較することで Slack からの POST が改ざんされていないことを検証します。

リスト 2.2: 署名の検証関数

```
// 署名の検証
private bool VerifySignature(
    string body, string timestamp, string headerSignature)
{
    // 署名の検証
    // SEE: https://api.slack.com/docs/verifying-requests-from-slack
    var sigBaseString =
        $"v0:{timestamp}:{body}";
    var generateSignature =
        $"v0={CreateSignature(sigBaseString, "SigningSecret")}";
    return generateSignature == headerSignature;
}

// 署名の生成
private string CreateSignature(string inputStr, string secretKey)
{
    HMACSHA256 hmacsha256 =
        new HMACSHA256(Encoding.UTF8.GetBytes(secretKey));
    var signature =
        hmacsha256.ComputeHash(Encoding.UTF8.GetBytes(inputStr));
    var base16Signature = Convert.ToHexString(signature).ToLower();
    return base16Signature;
}
```

検証には Verification Token を使う方法もありましたが廃止予定[*4]になっており、より安全な方法として署名を使う方法が推奨されています。

[*4] https://api.slack.com/authentication/verifying-requests-from-slack#deprecation

url_verification への応答

Event Subscriptions の設定ページで URL を登録する際には、POST URL に Slack から検証用の特定のリクエスト[*5]が行われるのでそれに応答する必要があります。

Slack からは JSON が POST され、type が url_verification の場合に含まれる challenge パラメータを、改めてレスポンスに含めて返す必要があります。

https://api.slack.com/events/url_verification

リスト 2.3: URL 検証への応答処理

```
public async Task HandleAsync(HttpContext context)
{
    // Bodyを読み出す
    string body = await Utilities.ReadStreamAsync(context.Request.Body);
    // 略 ヘッダを読み出してVerifySignature()で署名の検証をする

    // NewtonsoftでJSONのパース
    JObject jsonObj = JObject.Parse(body);
    // リクエストタイプの取得
    string requestType = jsonObj["type"].ToString();
    // WebHook URL の検証
    if (requestType == "url_verification")
    {
        var challenge = jsonObj["challenge"].ToString();
        await context.Response.WriteAsync(challenge);
        return;
    }

    // 略 後述のCloud Tasksに登録する処理
    // 略 200 OKを返す
}
```

メイン処理用の Cloud Functions の呼び出し

Slack からの POST には 3 秒以内にレスポンスを返す必要があります。Cloud Functions は、レスポンスを返したあとは更に外部へ HTTP リクエストなどはできません。そのため時間のかかる処理をする必要がある場合には、サイドキックするような形で別途処理を続ける必要があります。

[*5] https://api.slack.com/events/url_verification

今回の構成では、Cloud Tasks を経由して、メイン処理用の Cloud Functions を呼び出すようにしています。Cloud Tasks には、1 時間以内に同一 ID の Task が登録された場合排除されるという仕様[*6]があります。そのため、Slack からの POST に含まれるメッセージ ID をタスク名として登録することで、受信用の Cloud Functions への Slack からの POST がリトライされていても重複排除ができます。

リスト 2.4: メイン処理用の Cloud Functions を呼び出す Task の登録

```
// CloudTasksに登録する
CloudTasksClient client = await CloudTasksClient.CreateAsync();
// 控えておいたプロジェクトID、リージョン、キュー名を入れる
QueueName parent = new QueueName("ProjectId", "Region", "QueueName");
// タスク名にメッセージごとのユニークなIDを使う
var msgId = jsonObj["event"]["client_msg_id"].ToString();
TaskName taskName =
    new TaskName("ProjectId", "Region", "QueueName", msgId);

await client.CreateTaskAsync(new CreateTaskRequest
{
    Parent = parent.ToString(),
    Task = new Task
    {
        Name = taskName.ToString(),
        // 型名がNamespace間でバッティングするので明示する
        HttpRequest = new Google.Cloud.Tasks.V2.HttpRequest
        {
            HttpMethod = HttpMethod.Post,
            Url = "MainFuncUrl", // メイン処理用のCloud FunctionsのURL
            Body = ByteString.CopyFromUtf8(body) // そのまま渡す
        },
        ScheduleTime = Timestamp.FromDateTime(
            DateTime.UtcNow.AddSeconds(0)) // 遅延なしに今すぐキック
    }
});
```

ここまでのコードをまとめて、受信用の Cloud Functions のコードをデプロイします。Cloud Functions にデプロイしたら、こちらも Cloud Functions の URL を控えておいてください。

[*6] https://cloud.google.com/tasks/docs/reference/rest/v2/projects.locations.queues.tasks/create

2.7 Slack の Event Subscriptions の設定

Slack App の Event Subscriptions の設定ページに戻って、まずは Enable Events を On にして有効化します

Request URL の設定

イベントが発生した際に Slack から POST される URL を設定します。先ほど作成した受信用の Cloud Functions の URL を入れます。即座に検証用のリクエストが受信用の Cloud Functions に送られて、WebHook URL の検証用に指定した処理が適切に動けば Verified と表示されます。

転送 Event の設定

Subscribe to bot events で受信したいイベントを追加します。サブスクライブ対象を app_mention のみにすることで、Bot へのメンションのみ送るように指定できます。このサブスクライブの指定を保存する際に、必要な OAuth の権限追加を促されるので、承認して追加しておきます。

2.8 動作確認

ここまでで Bot の設定は完了しました。任意のチャンネルにボットを追加して、メンションを送って動作確認してください。

2.9 まとめ

今回は、Google Cloud を用いた Serverless な Slack Bot の作り方を紹介しました。実際に私は、時間のかかる処理として OpenAI の API を使って画像生成を行う Bot を作成しました。他にも、例えば BigQuery へのクエリ発行や、オペレーションの ChatOps 化なども簡単にできそうだと考えています。今回の記事が、Serverless な Slack Bot を作る際の参考になれば幸いです。

この記事で紹介したプログラムを整理したものは、以下のリポジトリにあります。Cloud Logging の機能を活かす Log 出力の工夫なども入っていますので、ぜひご覧ください。

https://github.com/gutio/ServerlessSlackBot

第 3 章

Python の match 文に詳しくなってみましょう

Shunsuke Ito / @fgshun

Python の **match 文**。Python 3.10 にて追加された、 Python オブジェクトが特定のパターンにマッチするか、した時にどのような処理を行うかを記述できる記法です。そんな match 文について紹介します。

3.1 if 文の代わりに使う

ある関数の戻り値が 0 か 1 か それ以外かで分岐する処理を行ってみましょう。 if 文で記述すると次のようになるでしょう。

リスト 3.1: if 文による条件分岐

```
ret = something()
if ret == 0:
    ...
elif ret == 1:
    ...
else:
    ...
```

このコードの気になるところは関数を複数回呼び出さないために戻り値を保持する変数 ret を用意し、何度も記述しなければならないところです。この煩わしさは match 文を用いることで解消することができます。

リスト 3.2: if 文の代わりに match 文を用いる

```
match something():
    case 0:
        ...
    case 1:
        ...
    case _:
        ...
```

match 文は、 match 、調査したい値、それにつづくひとつ以上の case という構成の複合文です。その動作は次のようなものです。

- 先頭の case に記されたパターンが調査したい値に合致するかを判定する
- 合致したならば続く処理をして match 文を抜ける
- しなければその直後の case を探し、あるのであれば同様の処理をくり返す

同じ値との比較をくり返す類の if の代わりとして用いるだけでも、とりあえずは便利なものです。しかし、 match 文でできることは等しいかどうか判定するだけではないのです。あるパターンを示し、それに合致するかどうかを判定することができるのです。名は体を表す、です。

3.2 パターンマッチ記法

Python 、 match 文の多彩で便利なパターンマッチ記法たちを紹介していきます。まずは、単独で意味を持つパターンたちからです。

ワイルドカードパターン

ワイルドカードパターンはアンダースコア _ で書き記されます。あらゆる値にマッチするパターンです。単独で用いた場合は if 文における else 句の役割を担うことができます。

リスト 3.3: ワイルドカードパターンの例

```
match 0:
    case _:
        ...  # 何であれ必ずマッチして実行される
```

リテラルパターン

リテラルパターンは数字、文字列、バイト列、 True, False, None を単独で記述したのとそっくりな見た目のパターンです。値が等しいかどうかを判定するパターンです。より正確には、数字・文字列・バイト列は同値かを、 True, False, None は同一かを判定します。

リスト 3.4: リテラルパターンの例

```
match 1:
    case 0:  # 1 == 0 が False なので合致しない
        ...
    case 'spam':  # 1 == 'spam' が False なので合致しない
        ...
    case True: # 1 is True が False なので合致しない
        ...
    case 1.0:  # 型は違えど、 1 == 1.0 は True なので合致する
        ...
    case 2+3:  # シンタックスエラー
               # 式が記述できる場所ではない
               # 前述の式・単独のリテラルに見えるものも式ではない
               # あくまでパターン・リテラルパターン
        ...
```

バリューパターン

バリューパターンは識別子のように記述され、それと値が等しいかを判定するパターンです。より正確には、ドット . を含む識別子のように記述されたものです。後述のキャプチャパターンとの区別のため、キャプチャパターンを手軽に書けるようにするために課された制約です。想定される用法としては Enum を調査対象にとって分岐する if 文の代替です。

リスト 3.5: バリューパターンの例

```
from enum import Enum, auto
class Environment(Enum):
    BLUE = auto()
    GREEN = auto()
def do_something(env: Environment) -> None:
    match env:
        case Environment.BLUE:
            ...
        case Environment.GREEN:
            ...
```

キャプチャパターン

キャプチャパターンはやはり識別子のように記述され、必ずマッチして値をキャプチャするパターンです。より正確には、ドット . を含まない識別子のように記述されたものです。

単独で用いた場合は必ずマッチして、キャプチャした値は match に渡した値そのものとなるのでワイルドカードパターン単独と特に変わるものではないです。

リスト 3.6: キャプチャパターンの例

```
match 0:
    case A:
        # 何であれ必ずマッチして変数 A に match に渡した値がキャプチャされる
        print(f'{A=}')
```

OR パターン

OR パターンは、2 つのパターンを縦線（|）で結合する形で記述されるパターンです。左のパターンがマッチしている時は問答無用でマッチし、していない時にも右のパターンがマッチしている時にはマッチします。これにより、どちらかがマッチしたならば、という判定が可能となります。OR パターンもパターンのひとつなので 2 つ以上連ねることも可能です。

リスト 3.7: OR パターンの例

```
ret = do_something()
match ret:
    case 0 | 1:
        ...  # 0 か 1 の時にマッチして実行される
    case 2 | 3 | 4:
        ...  # 2 か 3 か 4 の時にマッチして実行される
             # (2 | 3) | 4 や 2 | (3 | 4) と同等
```

AS パターン

OR パターン、ワイルドカードパターンにおいて、パターンがマッチした際に確認対象となった値を後から利用したい場合があります。先程の リスト 3.7 では match 文の前に関数を実行して戻り値を `ret` に保持していましたが、リスト 3.8 のように `AS` パターンを使うことで、必要になってから名前を付けてバインドすることができます。

リスト 3.8: as パターンの例

```
match do_something():
    case 0 | 1 as ret:
        # 0 か 1 の時にマッチして実行される
        # do_something() の結果は ret として参照できる
        print(f'{ret=}')
```

より正確には、パターン、 `as` 、変数名という組で表現され、左辺のパターンがマッチしたときに問答無用でマッチして、右辺の変数へ判定していた値を代入するというパターンの一種としてこの機能は実現されています。

そして、 `AS` パターンの使い道は match 文に判定対象として渡された単独の値を、後からバインドすることができるだけではありません。

3.3 パターンを組み合わせる

今まで「単独で用いた場合」「単独の値」のように匂わせてきていましたが、パターンたちは組み合わせてより複雑な表現を行うことが想定されています。

クラスパターン

クラスパターンは、そうした複数のパターンを組み合わせて表現されるパターンのひとつです。クラスパターンは値がとあるクラスもしくは子クラスのインスタンスかどうか、そして値の属性はクラスパターン内のすべてのサブパターンとマッチするかを確認するパターンです。その見た目はコンストラクタ `__init__` 呼び出しに似たものとなりますが、コンストラクタ呼び出しではなくあくまでパターンであり、インスタンス生成がなされることはありません。

リスト 3.9: クラスパターンの例

```
import datetime
d = datetime.date(2023, 11, 26)
match d:
    case datetime.date(year=2023, month=_, day=D):
        # 2023, _, D の 3 つのサブパターンをもつクラスパターン。
        # まず d が date のインスタンスかが確認される。
        # 続いて属性を持っているか、持っていればサブパターンと合致するかを確認。
        # d.year 属性にリテラルパターン 2023 が、
        # d.month 属性にワイルドカードパターン _ が、
        # d.day 属性 にキャプチャパターン D が対応し
        # それぞれマッチするかが確認される。
        # キャプチャパターンにより変数D に d.day すなわち 26 が代入される
        # サブパターンすべてがマッチしたときに
        # このクラスパターンはマッチしたと判断される
        ...
```

クラスパターンのサブパターンは、キーワード引数のような書式でどの属性を対象に取るかを指定するようになっています。クラス側に仕込みを入れておくことで位置引数のような書式での指定も可能となります。これには `__match_args__` クラス属性を用います。

リスト 3.10: 位置引数風の指定を可能にする

```
class Product:
    __match_args__ = ('name', 'date')
    def __init__(self, name, date):
        self.name = name
        self.date = date
match Product('KLab Teck Book', datetime.date(2023, 11, 11)):
    case Product(name='KLab Tech Book'):
        ...  # name 属性の確認が行われる
    case Product('KLab Tech Book'):
        ...  # Product.__match_args__[0] 属性、
             # すなわち name 属性の確認が行われる。同じ意味。
```

クラスパターンでは、サブパターンを指定しないということが可能です。これは属性の確認は行わずに型の確認だけを行うことを意味します。`isinstance` をくり返す `if` 文の代用とすることが可能となっています。

リスト 3.11: match 文で型チェックを行う

```
match v:
    case str():  # v が str 型のインスタンスかが確認される
        ...
```

クラスパターンの属性確認にクラスパターンを使う、という入れ子の指定が可能です。そして AS パターンと組み合わせることで、対象となった属性を抜き出すということも可能です。

リスト 3.12: クラスパターンで属性の型チェックを行う

```
match Product('KLab Teck Book', datetime.date(2023, 11, 11)):
    case Product(name=str() as product_name):
        ...  # name 属性がクラスパターン str() と合致するかが、
             # つまり str 型かどうかの確認がおこなわれる
             # str 型であった場合、 AS パターンも合致し
             # name 属性が product_name 変数に代入される
    case Product(name=str(product_name)):
        ...  # 同義。組み込み型たちには位置引数0番目で自身を示す仕込みが
             # なされている。ここにキャプチャパターンを置くことが想定されている
```

シーケンスパターン

シーケンスパターンは値がシーケンス型 (例: list, tuple) で、長さが一致し、内容がサブパターンすべてとマッチする場合にマッチするパターンです。

リスト 3.13: シーケンスパターンの例

```
L = [0, 1, 2]
match L:
    case [0, 1]:
        # 長さが 2 か
        # その値たちはリテラルパターン 0, 1 にマッチするか
        ...
    case [0, 10, 20]:
        # 長さは 3 か
        # その値たちはリテラルパターン 0, 10, 20 にマッチするか
        ...
    case [0, A, B]:
        # 長さは 3 か
        # その値たちはリテラルパターン 0 、
        # キャプチャパターン A, B にマッチするか
        ...
```

シーケンスパターンでは 0 個以上を意味するチャプチャパターンをひとつ含めることが可能となっています。この場合、個数の確認は他のサブパターンの数以上の要素数を持つかどうか、となります。

リスト 3.14: 0 個以上の要素をまとめて扱う

```
L = [0, 1, 2, 3, 4]
match L:
    case [0, *B]:
        # 1 つ以上の要素を含み、先頭が リテラルパターン 0 に一致するか
        # 先頭以降を B に代入
        ...
    case [A, *B]:
        # 1 つ以上の要素を含むか
        # 先頭要素を A に、以降を B に代入
        ...
    case [A, *B, C]:
        # 2 つ以上の要素を含むか
        # 先頭要素を A に、末尾要素を C に、 残りを B に代入
        ...
```

マッピングパターン

マッピングパターンは値がマッピング型 (例: dict) で、内容がサブパターンすべてとマッチする場合にマッチするパターンです。シーケンスパターンと異なる点は、要素数の確認は行われないところです。

リスト 3.15: マッピングパターンの例

```
m = {0: 1, 2: 3}
match m:
    case {0: 1, 2: 3}:
        # m[0], m[2] がリテラルパターン 1, 3 にマッチするか
        ...
    case {0: 10, 2: A}:
        # m[0] がリテラルパターン 10 に、
        # m[2] がキャプチャパターン A にマッチするか
        ...
    case {0: 100}:
        # m[0] がリテラルパターン 100 にマッチするか
        ...
```

3.4 パターンで表現しきれないとき

match 文ではパターンで表現しきれないケースに対し、後判定を入れることができるようになっています。この処理は、 case の後ろに if を入れることで書き記され、guard と呼ばれています。

リスト 3.17: guard の例

```
m = {0: 1, 2: 3}
match m:
    case {0: int() as A} if A <= 10:
        # マッピングパターン
        # クラスパターン int() に m[0] がマッチするか
        # int であれば変数 A に代入
        # その後、 A <= 10 という通常の Python 式の真偽の評価が行われ
        # 真であった時にはじめて case ブロックが実行される
        ...
```

guard には通常の Python 式が記述できます。できてしまうがゆえに、副作用もコード次第です。例外も発生しえます。乱用・悪用すれば挙動が読みにくいコードを書くことができてしまいますので、注意が必要となります。guard で行うことは、真偽の判定を副作用のない範囲に留めておくことをおすすめします。

リスト 3.17: guard の乱用例 - テキストファイル書き込み

```
match s:
    case str() if open('spam', 'w').write(s):
        # よくない例
        # s が str 型かを確認した後に追加の確認を入れたいのだな、
        # と思わせておいてからの唐突なファイル書き出し
        # guard でやるべきことではない
        ...
```

3.5 おわりに

今まで if 文で表現してきた場合分け。これを、わかりやすく書き記すことができる可能性を秘めた match 文が追加されました。ぜひ、使ってみてください。

第 4 章

アニメーション GIF とファイルサイズの話

Toshifumi Umezawa

4.1　はじめに

最近 Misskey[*1]を使いはじめました。絵文字を登録できる SNS なので、バラエティ豊かな絵文字が投稿内容やリアクションで使われていてとても賑やかなのですが、ふと容量が大きそうなアニメーション画像のサイズを確認すると約 500KB でした。アニメーション画像の場合、1 フレームあたりのサイズが小さくてもフレーム数の分だけ容量がかさむため思いのほか容量が大きいことがあり、気付かないうちにまあまあな「ギガを食う」かもしれません。

SNS で容量を食うのは基本的に添付された画像だとは思うんですが、絵文字についても既存で登録されているものについて容量を削減できないかと思い、画像の仕様について色々と調べてみました。この記事ではアニメーション GIF のファイルサイズ削減に繋がる話をします[*2]。

4.2　アニメーションを行える画像形式

まずは、アニメーション画像が扱える画像形式についてざっくりと説明します。

[*1] 分散型 SNS のひとつで、1 つのサーバで作成したアカウントから、共通仕様を持つ別のサーバと互いに交流することができる。

[*2] 本当はより多機能かつ容量でもメリットのある WEBP について説明したかったが、内容が多く執筆しきれなかったため軽く触れる程度にした。

GIF

GIF では、最大 256 色のカラーテーブルを使い、1 ピクセルごとの色情報はカラーテーブルの色番号 (インデックス) で表現します。また、アニメーションも可能で、複数のフレームと制御用の情報 (1 フレームあたりの表示時間など) を格納できるようになっています。

データは圧縮されて格納されますが、可逆形式のため (256 色以下の画像については) 画素を変化させずに圧縮/解凍できます

古くからあるため、ブラウザや画像ビューワ等ではほぼ間違いなく表示に対応しています。ただし、そもそも 256 色までしか扱えないなど現代では機能不足な側面もあり、使う場面は限られています。

PNG

PNG では、GIF と同様のインデックス形式に加えて、フルカラー形式での保存やピクセル単位でのアルファ値 (透明度) の設定も可能になっています。

データの圧縮方法についてもより効率が良いものが採用されており、基本的に GIF と同じ内容であれば PNG で保存したほうがデータサイズが減ります。また、圧縮レベルの指定によって処理時間とトレードオフで圧縮率を上げることもできるようになっています。

アニメーションが可能な形式として、PNG の拡張の APNG という形式があります。GIF と同様に複数のフレームを格納した形式なのですが、PNG の基本仕様には含まれておらず拡張という形になっています。そのため、ツールによっては正しくアニメーションが再生されなかったり編集ができなかったりします[*3]。

WEBP

2010 年に公開された比較的新しいフォーマットです。

GIF/PNG/JPEG の主な機能が合わさったような形で、可逆/非可逆の両方の形式を扱うことができ、アニメーションもできます。今までのフォーマットではできなかった非可逆形式で透過する画像や、非可逆形式のアニメーションする画像なども扱えるようになっています。[*4]

これによって、容量に関して有利な JPEG[*5]を使いたくても透過やアニメーションのために PNG/GIF にしたり、実写映像をアニメーション画像にしたいという理由で相性が悪い GIF にしていた場面などで、効率の良い形でデータを持てるようになりました。

[*3] 例として、画像編集ツールの GIMP は標準で非対応となっていたり、Windows10 標準のフォトだと静止画として扱われるなど

[*4] 完全な上位互換ではなく、既存の画像形式と全く同じ細かい設定が使えるわけではない

[*5] JPEG は非可逆形式で圧縮するため元データから劣化するが、可逆形式と比較してデータサイズが大きく下がることが一般に知られている

データの圧縮方法についても改良が加えられており、基本的に既存のフォーマットより容量が削減できます。表現力でも容量でもメリットがあるので、今後は対応状況を見ながら WEBP を使っていくのが良いと思います。実際、冒頭で触れた Misskey ではアップロードした画像が自動で WEPB に変換されていたりします[*6]。

欠点は、新しめのフォーマットのため表示や編集に対応していないツールがある点です。メジャーなブラウザでは対応されていますがまだバグ等がある可能性があったり[*7]、ビューワやエディタによっては対応していないこともあります。

また。拡張子では可逆/非可逆の判別ができないため、編集用の元データを非可逆形式で上書きしてしまわないように気をつける必要があります。

4.3 GIF の構造と容量削減に使える仕様

GIF の容量を削減するための方法について、実際のデータ構造について触れながら説明していきます。

データ構造

バイナリデータは基本的に情報のまとまりごとに構造が定義されています。アニメーション GIF の場合、コメントの領域などを除いたおおまかな構造は表 4.1 のようになっています。

Application Extension のブロックが入ることでアニメーションとして扱われ、その後にフレーム数の分だけ Graphics Control Extension と Image Data を追加します。

表 4.1: アニメーション GIF のブロック構造

ブロック	概要
Header	画像の縦横サイズなどの基本情報が格納されている
Application Extension	"NETSCAPE 2.0" という文字列が固定で入っている
Graphic Control Extension	アニメーション時の表示間隔などを設定する拡張ブロック
Image Data	このブロックに適用されるローカルの設定 + 画像データ
Graphic Control Extension	
Image Data	
..	
Trailer	ファイルの終わりを表す (0x3B の 1 バイトのみ)

[*6] 2023 年 10 月現在、ブラウザ上では場合によって元データの gif が表示されていることもあるので、gif 自体の容量を減らすことのメリットはある。

[*7] iOS のブラウザでアニメーションの残像が表示されることがある気がする...

■コラム: APNG のブロック構造

APNG のブロック構造は表 4.2 のようになっています。

表 4.2: APNG のブロック構造

チャンク	概要
Signature	PNG ファイルであることを表す先頭 8 バイト (チャンクではない)
IHDR	ヘッダ
acTL	アニメーション画像として全体で持つ情報
fcTL	アニメーションのフレームに設定する情報が入る
IDAT	画像情報
fcTL	
fdAT	アニメーション用に IDAT と同じ画像情報が入る
fcTL	
fdAT	
..	
IEND	ファイルの終わり

方向性は GIF と同様で、静止画として必須の情報にアニメーション画像用の情報を追加で入れた形になっています。

PNG におけるチャンクは名前が表 4.2 のようにアルファベット 4 文字で格納されているのですが、先頭 1 文字が大文字のものが必須チャンクと決まっています。そのため、PNG には対応しているが APNG には対応していないアプリで APNG を表示しようとすると、アニメーション画像としての情報は無視された結果として静止画として正常に表示されます。

こういった仕様のため、WEB 上にアップロードしたものがそのまま表示されるような WEB アプリの場合、PNG としてアップロードさえできれば表示されるかどうかはブラウザ側の対応次第になったりします。

グローバルカラーテーブル、ローカルカラーテーブル

Header ブロックにはグローバルカラーテーブルと使用有無のフラグ、ImageData ブロックにはローカルカラーテーブルとの使用有無のフラグが格納されています。最大で 256 色まで 3 バイトの RGB 値を指定でき、ローカルカラーテーブルを使用する場合はフレームごとに固有のものを使うことができます。

仮に 256 色全て使う場合、カラーテーブル 1 つあたり 768 バイト使うことになります。ローカルカラーテーブルはフレーム単位で設定することが可能なので、場合によってはフレーム数 × 0.75KB の容量を食うことになります。(画質のことを考えるとしょうがないとはいえ、実写画像を繋げて GIF にするような WEB アプリを利用するとこの形式の GIF が出来上がることが多いです)

色数

カラーテーブルのサイズについては 2,4,8,16,32,64,128,256 色の 8 通りから選ぶことができます。256 色の色番号を表現するのに 8 ビット必要になるのですが、仮に 16 色しか使わない場合は 4 ビットで十分になります。

使う色数を削減するとカラーテーブル自体が小さくなることに加え、そもそも 1 ピクセルを表現するために必要なビット数が少なくて済むことと、圧縮処理の中で既に出現した近いパターンに置き換える際に類似パターンが多くなるため、より小さなサイズになります。

■コラム: 圧縮アルゴリズム

GIF のデータ圧縮に使われている **LZW** というアルゴリズムは、辞書と呼ばれるものを動的に追加しながら圧縮/展開を行います。例として、A, B, C のみが登場する文字列で圧縮時の動作例を示すと図 4.1 のようになります。[*8]

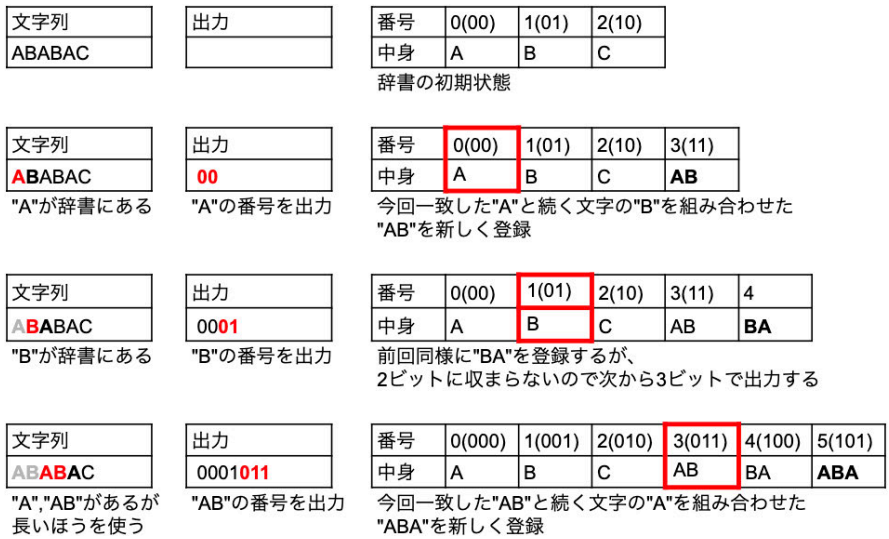


図 4.1: GIF で使われる圧縮アルゴリズムの動作イメージ

処理開始前に、存在する全ての 1 文字 (この例では A, B, C) を辞書に登録しておきます。以降は、既に辞書にある一番長い一致文字列の番号を出力し、その文字列 + 続く不一致文字を辞書に追加する、という動作を繰り返します。展開する際には、A~C がそれぞれ 0~2 に割り当てられているという前提のもとで逆の操作を行います。このような処理によって、辞書が充実するほど長い文字列を 1 つの短い番号で代替できるようになってきます。

色数を削減することで、辞書の番号を表現するのに必要なビット数が減ることと、辞書に出現した並びが出現しやすくなることがイメージできると思います。

また、新しい画像フォーマットのほうが圧縮率が高いと記事内で記載したのですが、PNG では圧縮データを **zlib** 形式で保持しており、これは **Deflate** という圧縮アルゴリズムを利用しています。Deflate は **LZ77** と**ハフマン符号化**を組み合わせたもので、zip や gzip などの非常に身近な圧縮ファイルにおいても使われています。

lz77 による圧縮時の動作例を示すと図 4.2 のようになります。

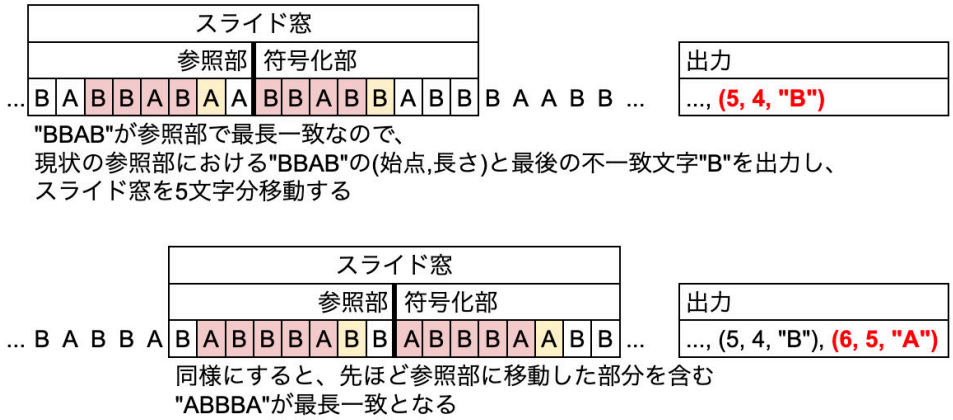


図 4.2: PNG で使われる圧縮アルゴリズムの動作イメージ

このアルゴリズムでは、参照部 (直近で読み込んだ部分) と符号化部 (まだ読み込んでない部分) の 2 つを合わせた部分をスライド窓と呼びます。符号化部の先頭から参照部内で最長一致する部分を探し、(始点, 長さ, 次の不一致文字) を記録し[*9]、出力が終わった分だけスライド窓全体を移動していきます。また、LZW では `A=00` のように登録順にビット表現を割り当てていましたが、Deflate では LZ77 の適用後さらにハフマン符号化を適用し、(始点, 長さ, 次の不一致文字) それぞれについて出現頻度が高いものに短い表現が割

り当てられるように置き換えられます。

GIF のアルゴリズムでは登録済みの辞書が 1 文字ずつ長くなっていく形でしたが、PNG のアルゴリズムでは参照部自体が辞書としての役割を果たすため、長い文字列の一致をより探しやすいということが感覚的に伝わるでしょうか。欠点としては GIF より圧縮/解凍に時間がかかることですが、現代のデバイスの処理速度では気になる場面はほとんどないはずです。

差分のみの更新

ImageData ブロックには、画像の左上始点と縦横サイズの情報が格納されています。Header にも画像サイズが設定されていましたが、ImageData 個別のサイズはフレーム更新の際に再描画が必要な矩形領域の設定です。例として、100x100 の画像のうち特定の 5x5 の領域だけしか変化がない場合、5x5 の画像データを持っていればよいということになります。

また、GraphicControlExtension にはフレーム更新の際の再描画方法についての設定があります。上記で設定した矩形領域内に再描画不要な領域が含まれる場合、透過色で塗りつぶして透過部分は描画しないという設定が可能です。

4.4 容量削減を適用できる例

ここまでで説明した内容が実際に容量削減に繋がる例を紹介します。ここでは画像の処理は GUI ツールの GIMP を使います。また、gif の構造を実際に見たかったため、お手製の構造解読スクリプト[*10]で説明に必要な情報をピックアップして出力しています。[*11]。

[*8] 実際には制御用の文字の登録やビット桁数に関する細かい話がありますが、ここでは紙面の都合上アルファベット 3 文字のみを 2 ビットサイズの辞書で表せるものとしています。

[*9] 実際には改良版の LZSS が使われる場合が多く、一致文字列なし扱いの場合の始点/長さの出力 (0,0) を省略するために一致有無を表す先頭 1 ビットが追加されている。一致する場合に余分な 1 ビットが追加された分を差し引いて圧縮効率が高いらしい。

[*10] https://gist.github.com/random25umezawa/26babb20ff751e4279fa89b9bfa51943

[*11] imagemagick でもある程度詳細な情報は見れるが、カラーテーブルの利用可否や ImageBlock が食っている容量などブロックとしての生のデータは見れなかった。使用できるパラメータは https://imagemagick.org/script/escape.php を参照

GIF の減色

まず例として、文字の絵文字を作成する際によく使われる MEGAMOJI[*12]で生成した図 4.3 のような単色のアニメーション画像を見てみます。

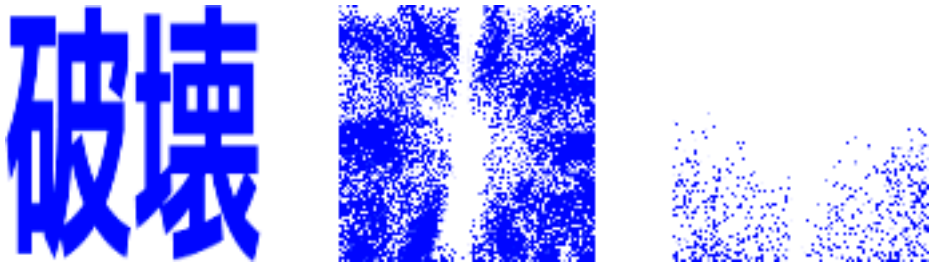


図 4.3: 「破壊」の文字が破壊されて塵になる GIF(96x96)

構造を出力するとリスト 4.1 のようになっていました。

リスト 4.1: 「破壊」の GIF の中身

```
Header, 96x96, gtable=(True,64)
ApplicationExtension
GraphicControlExtension, disposal_method=0, delay_time=60ms
Image, rect=(0,0,96,96), ltable=(False,2), data_size=2142B
GraphicControlExtension, disposal_method=0, delay_time=60ms
Image, rect=(0,0,96,96), ltable=(True,64), data_size=2852B
GraphicControlExtension, disposal_method=0, delay_time=60ms
Image, rect=(0,0,96,96), ltable=(True,64), data_size=2956B
...
```

gtable,ltable がカラーテーブルの使用有無のフラグとテーブルサイズなんですが、それぞれのフレームで個別に作られていることがわかります。また、この画像はほぼ青一色なので、少ない色数に減色してもさほど問題にならなさそうです。

GIMP で画像を開くと図 4.4 のメニュー項目にて画像のモードとして RGB が選択された状態になっているため、これをインデックスに変更します。すると図 4.5 のような項目が表示されるので、最大色数として必要な色数を選択します。

[*12] https://zk-phi.github.io/MEGAMOJI/

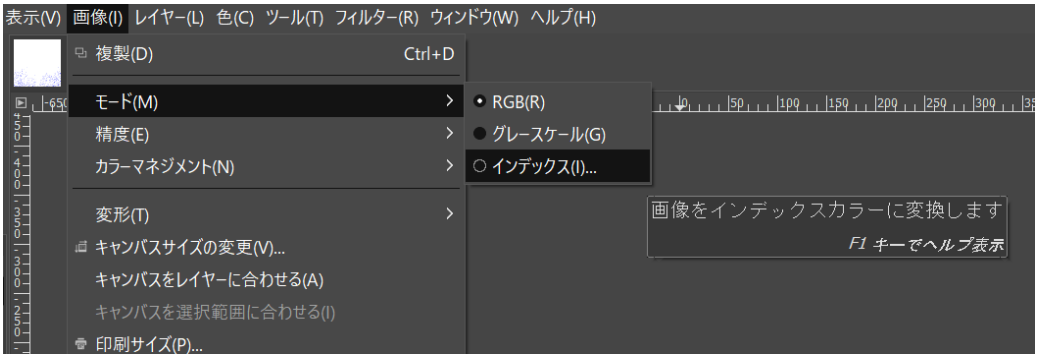


図 4.4: 画像のモード

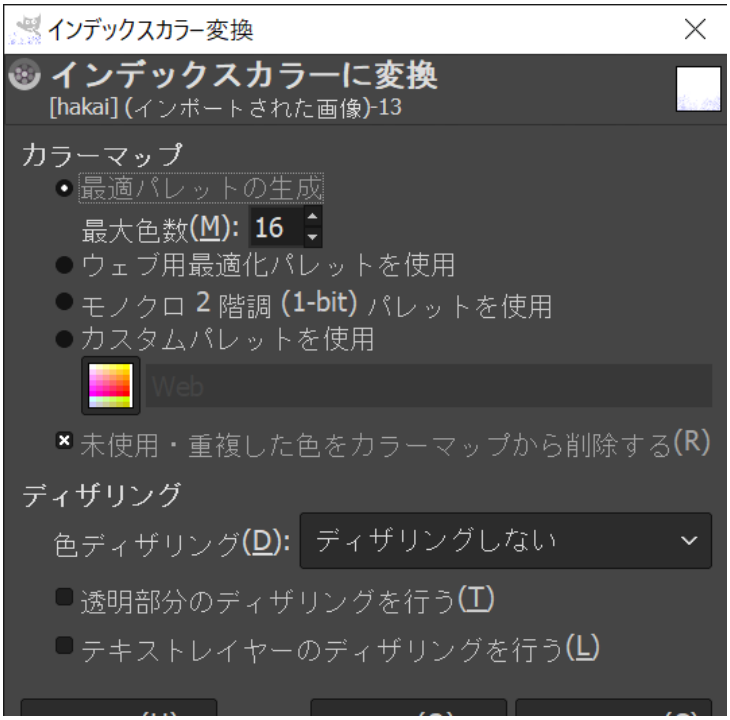


図 4.5: インデックス画像のパレットを設定する画面

16 色になるように指定して変換後、アニメーション GIF としてエクスポートした画像の構造を出力するとリスト 4.2 のようになっていました。

リスト 4.2: 16 色に減色後の GIF の中身

```
Header, 96x96, gtable=(True,32)
ApplicationExtension
GraphicControlExtension, disposal_method=0, delay_time=60ms
Image, rect=(0,0,96,96), ltable=(False,2), data_size=1729B
GraphicControlExtension, disposal_method=0, delay_time=60ms
Image, rect=(0,0,96,96), ltable=(False,2), data_size=2392B
GraphicControlExtension, disposal_method=0, delay_time=60ms
Image, rect=(0,0,96,96), ltable=(False,2), data_size=2509B
...
```

(グローバルカラーテーブルのサイズが指定した 16 色ではなく 32 色になっていますが、) 個別のカラーテーブルを使用しないようになっています。また、ImageBlock 内に含まれる画像データ自体の合計バイト数を出力しているのですが、ここのサイズがリスト 4.1 と比較してフレームごとに数百バイト減っていることが確認できます。

いくつかの色数への減色を試したところ、ファイルサイズは表 4.3 となりました。

表 4.3: 減色適用後のファイルサイズの変化

状態	サイズ (KB)
変換前	28.3
256 色	26.7
16 色	22.8
2 色	19.2

このように、減色してもさほど気にならない画像の場合は、色数を落とすアプローチが有効です。大量の色が使われている画像を減色したい場合は、状況に応じて既存のカラーテーブル[*13]を利用したり、図 4.6 のように中間色を擬似的に表現するディザリングを活用するなどして不自然さが少なくなるように気を付ける必要があります[*14]。

[*13] 人間の目の特性を考慮して (R,G,B)=(3,3,2) ビットで割り振った 256 色のカラーテーブルなど (https://imagemagick.org/Usage/quantize/#332_colormap)

[*14] 図 4.6 に示すとおり、ディザリングすることで単色ベタ塗りの箇所が減る影響などで圧縮効率が多少下がるので、必ずしもディザリングが万能なわけではない。実際に「破壊」の画像はドットが散らばりはじめた 2 フレーム目以降で ImageBlock のサイズが増えている。

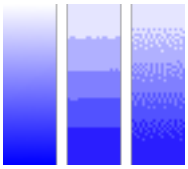


図 4.6: グラデーション (左)、5 色に減色 (中)、ディザリングを利用 (右)

フレーム差分の最適化

次に、フレーム差分の最適化について試してみます。今度は図 4.7 のような、変化のないノイズ画像の端の方に色の変化が少しあるアニメーション画像を用意しました。ノイズの部分は圧縮が効きづらいです。

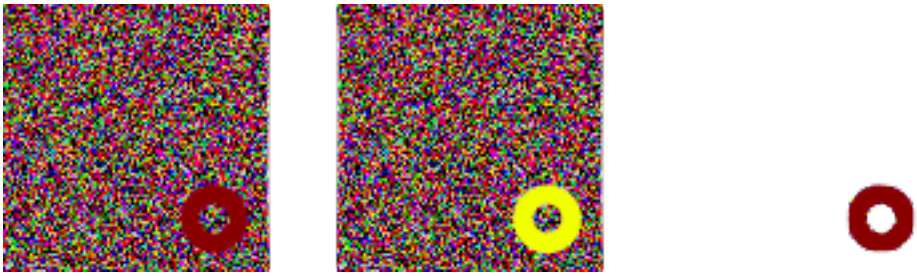


図 4.7: ノイズの上に色の変わるドーナツが配置された 2 フレームのアニメーション画像 (100x100)、一番右は差分のみ

この画像を GIMP で開き、図 4.8 のように「GIF 用最適化」を適用してエクスポートします。

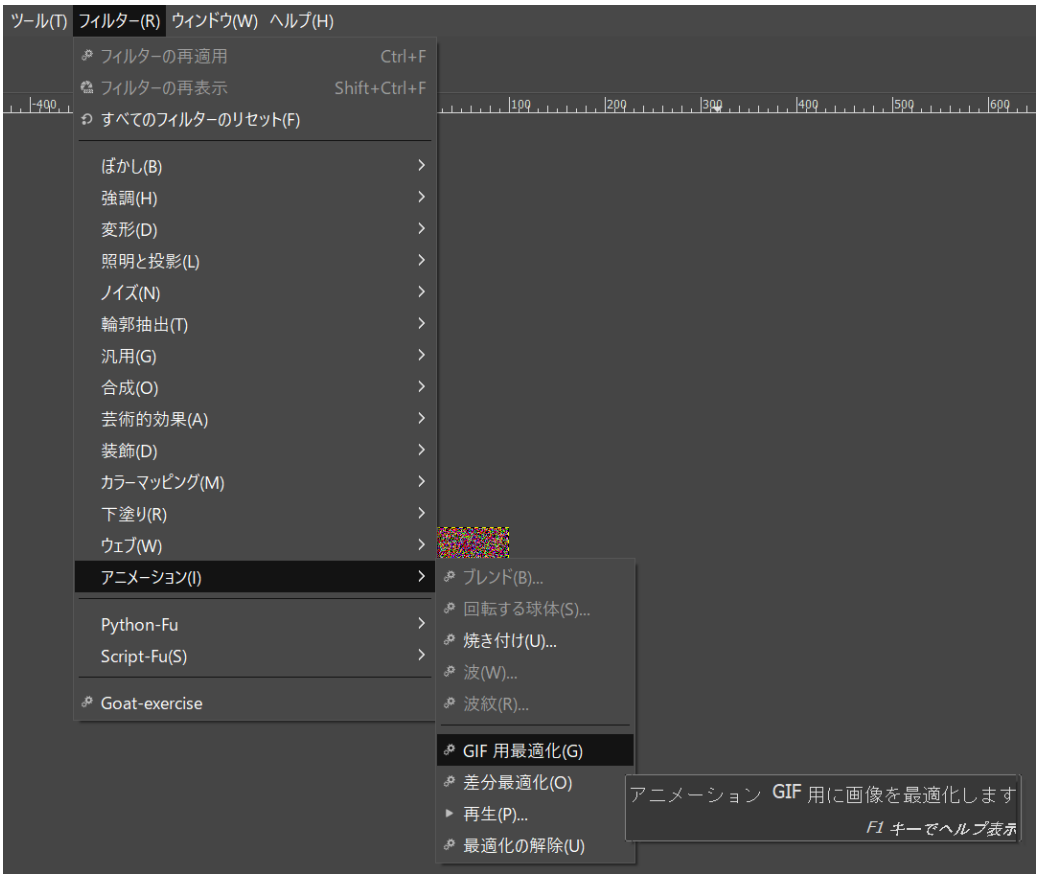


図 4.8: GIF 用最適化の適用

構造を出力するとリスト 4.3 のようになっていました。

リスト 4.3: GIF 用最適化をかけた画像の中身

```
Header, 100x100, gtable=(True,256)
ApplicationExtension
GraphicControlExtension, disposal_method=1, delay_time=100ms
Image, rect=(0,0,100,100), ltable=(False,2), data_size=12136B
GraphicControlExtension, disposal_method=0, delay_time=100ms
Image, rect=(66,67,26,26), ltable=(False,2), data_size=224B
```

1 フレーム目は ImageBlock の data_size が 12136 バイトとなっており、ほぼ同じピクセル数の「破壊」の ImageBlock の data_size(2000 バイト台) と比べてとても大きく、圧縮が効き

にくいことがわかります。そして、2 フレーム目は 1 フレーム目より data_size が 2 桁も減っていることがわかります。

GIF 用最適化を適用することで 2 フレーム目は図 4.7 の一番右のようになり、両フレーム間で全く同じものを表示している圧縮されづらいノイズの部分を描画対象外にしています。不要な部分が透過されているだけでなく、リスト 4.3 の ImageBlock 上で再描画する矩形領域 (rect で示した部分) が右下の色の変わる部分だけに限定されているのがわかります。

このように、一部分しか動かないうえにそれ以外の部分の圧縮率が悪い画像の場合、差分の最適化を行うことで容量が減らせる場合があります。使える場面は限られており、「破壊」の画像の場合は画像全体で動くピクセルが多く、背景も白一色で圧縮されやすいため、この最適化の恩恵はほとんどありません。

■コラム: ImageMagick での差分最適化

この複雑そうな最適化は、CLI の ImageMagick でも同様に適用できます。

```
# 最適化を適用
convert hoge.gif -layers Optimize hoge_optimize.gif

# 最適化を解除
convert gamingdonut_optimize.gif -layers coalesce hoge.gif
```

`-layers Optimize` で差分の最適化を適用した状態にすることができます。逆に解除する際は`-layers coalesce` をつけることで、各フレームで差分ではなく画像全体を保持するようにできます。

差分の最適化をしたままの画像を取り扱う場合、フレームごとに画像を分割する際に描画する矩形領域のみが出力されたり、WEBP に変換する際にフレームごとのサイズが違っているためエラーになったりします。

4.5 さいごに

GIF の容量にまつわる構造の話や、GIF の場合にできる特殊な手法について説明しました。実例の紹介では効果が分かりやすいように容量が減りやすいものをピックアップして説明しましたが、他の画像で容量削減を検討する場合にも役に立つ場面はあると思います。

冒頭で説明した通り、新しめの環境では基本的に WEBP を使用するほうが高機能かつ容量も小さくなることが多いため、極力こちらを使うことをオススメします。特に絵文字としての利用を想定した場合には小さなサイズで表示される[*15]ため、非可逆形式で保存した際のデータの劣化が気にならないことが多いはずです。

また、記事内で触れた色数削減に加えて、画像の縮小やフレームの間引きといった基本的な操作でも容量を下げることが可能です。再編集する想定がない画像であれば、伝えたい情報が伝わる必要最低限の解像度、フレーム数にすることのほうが容量削減を達成できることが多そうです。この方法に関しては WEBP を利用する場合にも効果がある内容のため、細かい最適化を考える前にまずはここから検討すべきです。

現代の増え続けるネットワークトラフィックの中では微粒子レベルの影響しかないであろう細かい内容でしたが、個人レベルでは「ギガを食う」のを避けるために多少効果がある話でもあるので、容量について気にする人が少しでも増えてくれたら嬉しいです。

[*15] Misskey の場合は文字の装飾として拡大することも可能なため、その際に粗が目立つ可能性はある。

第 5 章

GPU メモリアロケーター自作入門

Shinya Naganuma / @Pctg_x8

ゲームエンジンに欠かせない機能のひとつとして**メモリ管理機能**があります。ゲームエンジンが管理すべきメモリは主に 2 種類です。

1. 通常のシステムメモリ
2. GPU デバイス上にある **VRAM**（Video RAM）と呼ばれるメモリ

今回は、趣味で実験的に作っているゲームエンジン（**Peridot**）[*1]に、後者の VRAM を管理する**メモリマネージャー**（Peridot Memory Manager）[*2]を実装したので、それについて解説します。

5.1 動機

ゲームにはさまざまな**リソース**（画像、3D モデル、音声など）があり、それらをうまくメモリ上に配置することはゲームエンジンの大きな役割のひとつです。

Peridot は Vulkan をバックエンド API として利用しています。Vulkan では Buffer や Image といった**リソースオブジェクト**を用いて、それらを **DeviceMemory**（GPU デバイス上のメモリブロックを表すオブジェクト）に **bind** することでリソースのメモリ上への配置を行うことができます。

このとき、単純な実装であれば DeviceMemory とリソースオブジェクトを 1:1 で作成して配置するといったことが考えられます。しかし、DeviceMemory の確保やリソースオブジェ

[*1] `https://github.com/Pctg-x8/peridot`

[*2] 今回解説するものはオープンソース（MIT ライセンス）として公開しています（本書に掲載のコードは執筆用に一部手を加えてあります）：`https://github.com/Pctg-x8/peridot/tree/ft-memory-manager/modules/memory-manager`

クトの bind といった処理は CPU に対する負荷が高いため、**サブアロケーション**（ひとつのメモリブロックを細分化して複数のリソースで共有する）などの手法を駆使して可能な限り処理回数を減らす必要があります[*3]。

サブアロケーションを含むメモリ管理の機構をゲームごとに持つのは大変かつ非生産的なため、大半のエンジンでは何らかの形で効率よく柔軟にメモリ管理をするための仕組みをもっています。Peridot にはまだシーン基盤などのリソースを使用するエンジン機能が存在しないため、スクリプトから自由に確保/解放できる形のインターフェイスが必要になります。

スクリプトから使えるインターフェイスにする場合、`malloc` と `free` のように個別のリソース用のメモリを好きなタイミングで確保できるような仕組みにできると使いやすいものにできるため、理想的です。呼び出し側であらかじめ必要なリソースをリストアップしてもらうほうがより効率のよい仕組みにできそうですが、そうすると呼び出し側のプログラムの設計にいくつかの縛りができてしまい使いづらいものとなってしまいます。

このような考えから、「背景の複雑なメモリ管理機構を意識することなく、ゲーム側で個別のリソースを確保/解放できる機能」をゲームエンジン側から提供することを考えました。

■コラム: Vulkan Memory Allocator を採用しなかった理由

Vulkan をよくご存知の方は「Vulkan におけるメモリアロケーションといえば **Vulkan Memory Allocator** があるじゃないか」と思うかもしれません。

Vulkan Memory Allocator[*4]は GPUOpen が公開している OSS のメモリアロケーションライブラリです。GPUOpen（AMD, GPU ベンダー）によるライブラリなので非常に信頼性が高く、大きな理由がなければ通常はこれを使うべきではあります。

ただし、実装が C++ なので Rust などから使う場合は FFI を書かなければならない[*5]のと、別途 C++ のビルド環境も整えなければならないという手間がかかります。また、ゲームエンジンのいち機能として提供する場合、動的リンクで提供するか静的リンクで提供するかの選択肢があるなど、メモリ管理機能をつけるだけなのに別の新たな関心ごとが出てきます。ソフトウェアはシンプルな方が望ましいので、そういった本質でない関心ごとを作らないようにできるだけ複数の言語で構築することは避けたいです。

他にも、自作しておけばバグや考慮漏れ、特殊なケースが発生した場合にもすぐに対処できる利点がある[*6]ため、そういった観点でも今回は Vulkan Memory Allocator の採用を見送ることにしました。

——まあ、ここまで書いておいて単に自作はロマンだったりするわけですが。

[*3] Vulkan のデバッグレイヤーを有効化していると、小さな DeviceMemory の確保を検出して Performance Warning として報告されます

5.2 Buddy System と Slab Allocator

Peridot Memory Manager では、空きメモリの管理アルゴリズムとして **Buddy System** と **Slab Allocator** を実装しています。Linux カーネルに精通している方であればこのふたつについてはお馴染みかと思いますが、簡単にアルゴリズムの概要を説明しておきます。

Buddy System

Buddy System は一定サイズの連続した**メモリブロック**単位で空き領域を管理するアルゴリズムです。

仕組みとしてはシンプルで、2^n 個の連続した空きブロックごとにリストを構成して管理しています。たとえば、図 5.1 に示すような状況では 0, 1, 4, 5, 7 番のメモリブロックが未使用となっており、2 つ空きが連続している 0 番と 4 番を「2 連続」の空きリストに、前後に空きブロックのない 7 番を「1 連続」の空きリストに登録しています。

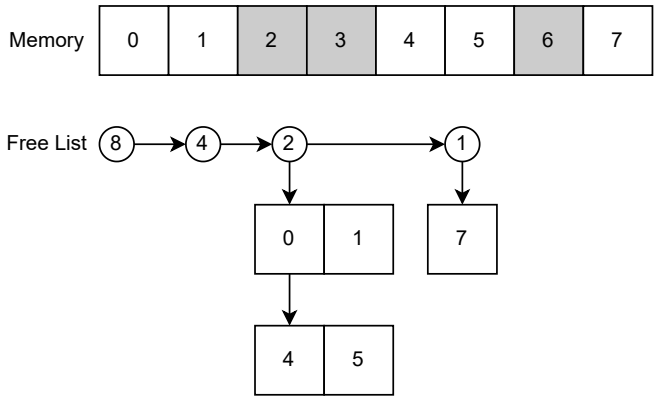


図 5.1: Buddy System

確保の際は次の手順で空きブロックを検索します。

1. 必要なブロック数を 2^n に切り上げる
2. 該当の空きリストを見て、空きがあればその空きをリストから削除して先頭アドレスを

*5 `https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator`

*6 単純な FFI ならただひたすら書くだけですが、Vulkan 側の FFI ライブラリ（bedrock）との組み合わせを考えると難易度が跳ね上がります。場合によってはどうにもできず詰むかもしれません

*7 ほんとうは OSS なんだからバグ報告して既存のものをよくしていくことに貢献するべきではあります

返す

3. なければ、さらに大きな領域（2^{n+1}）の空きリストを見て、空きがあればその先頭アドレスを返す
   - このとき、獲得ブロックの後半は空きのままなので 2^n の塊になるように分割してそれぞれの空きリストに追加する
4. 3. を繰り返し、最大サイズの空きリストにもブロックがなければ空きなしとする

解放の際は次の手順で空きブロックを追加します。

1. 2^n の塊になるようにブロックを分割する
2. それぞれの空きリストにブロックを追加する
   - このとき、追加しようとしているブロックに連なるようなブロックが他にあれば、追加せずそれと結合して大きなサイズの空きブロックとする
3. 分割したすべてのブロックがリストに登録されるまで 2. を繰り返す

Slab Allocator

Slab Allocator もまた固定サイズのメモリブロックの空き情報を管理するアルゴリズムですが、小さなサイズのものに使われることが多いです。Slab Allocator の構成を図 5.2 に示します。

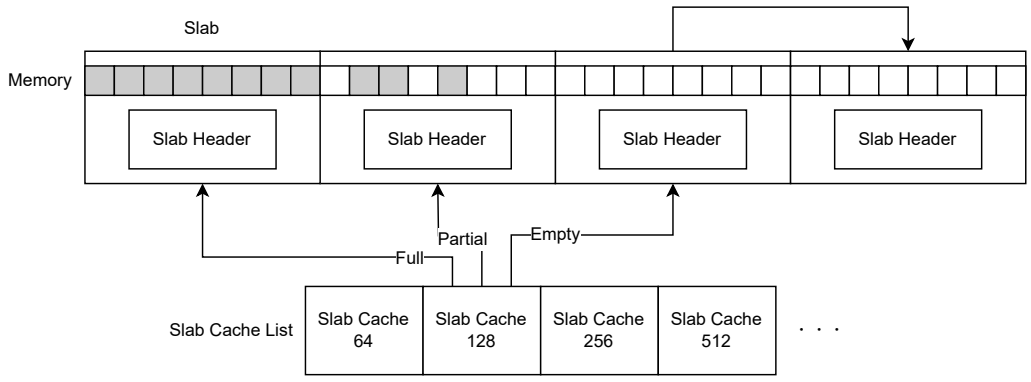


図 5.2: Slab Allocator

このアルゴリズムでは固定サイズの**オブジェクト**を複数個のまとまりごとに管理しています。このオブジェクトのまとまりに**スラブヘッダ**（管理情報）をくっつけた構造のことを**スラブ**（Slab）と呼びます。スラブはまたオブジェクトのサイズごとにグループ化され（**スラ**

ブキャッシュ)、またその中でもオブジェクトの使用状況によって次の 3 つにグループ化されます。

- **Empty**（すべてのオブジェクトが未使用）
- **Partial**（一部のオブジェクトが使用中）
- **Full**（すべてのオブジェクトが使用中）

Slab Allocator でオブジェクトを確保する際は次の手順で適切なオブジェクトを検索します。

1. 要求サイズを、用意しているスラブキャッシュのうちもっとも近いサイズのものに切り上げる
2. 該当サイズのスラブキャッシュの、Partial, Empty にある最初のスラブから空いているオブジェクトを検索する
3. オブジェクトが見つかったら使用中とマークする
4. 必要に応じて、該当のスラブの所属グループを変更する
   - Partial にいたスラブから確保し、そのスラブの空きがなくなった場合は Full に移動させる
   - Empty にいたスラブから確保した場合は Partial に移動させる
5. Partial, Empty どちらにもスラブがなければ確保不可とする
   - 実装によっては、追加のスラブを確保してそれを Empty に追加したのち、2. に戻る

オブジェクトを解放する際は次の手順で適切なオブジェクトを空き状態にします。

1. 要求サイズを、用意しているスラブキャッシュのうちもっとも近いサイズのものに切り上げる
2. 該当サイズのスラブキャッシュの、Full, Partial にあるスラブから対象のオブジェクトが含まれるスラブを検索する
3. スラブが見つかったらオブジェクトのインデックスを計算し、該当オブジェクトを未使用とマークする
4. 必要に応じて、該当のスラブの所属グループを変更する
   - Full にいたスラブで解放処理を行った場合は Partial に移動させる
   - Partial にいたスラブで解放処理を行い、そのスラブが空になった場合は Empty に移動させる
5. 必要に応じて、Empty なスラブが一定数溜まり、解放可能になったときはスラブを削除し、管理していたメモリを解放する

スラブキャッシュとして用意されるサイズには、適当な最小値（ほとんどの場合は 32）の 2^n 倍のサイズが選択されることがほとんどです。用途によってはよく使用される構造体（たとえば Linux のタスク構造体など）のサイズでスラブキャッシュを作ることもあります。

Peridot Memory Manager では、最小サイズを 64 としてその 2^n 倍のサイズのものを DeviceMemory のサイズ以下まで用意しています。64 は `float4x4` ひとつぶんのサイズと同じであり（`sizeof(float) * 4 * 4 = 64`）、ゲームで登場するメモリブロックはほとんどの場合でこれ以上になるため下限としては十分です[*7]。

5.3 Peridot Memory Manager 実装詳細

ここから具体的な実装の話に入っていきます。

まずは小さなサイズ向けの実装について説明します。Peridot Memory Manager では、一定サイズ（1MB[*8]）以下の小さなリソースに対してはひとつの大きな DeviceMemory を確保し、それを細分化して先述のアルゴリズムでサブアロケートしています。基本的には先に確保した DeviceMemory を使い回し、空きがなくなった時に新しいものを確保する形になっています。このとき確保する DeviceMemory のサイズは 1MB でも良いのですが、現在の実装では余裕を持って 4MB 確保するようにしています。

管理アルゴリズムには先述した Slab Allocator と Buddy System を組み合わせて使用しています。VRAM の管理の場合は別途システムメモリが使えるため、管理系のデータをシステムメモリに持つことで確保した VRAM をフルに活用することが可能です。

わざわざアルゴリズムを組み合わせているのはメモリ使用率をあげるためです。Slab Allocator だけでもメモリ管理は行えますが、ひとつの DeviceMemory 全体に小さなサイズを管理するスラブを割り当てるのはムダが多すぎます。ひとつの DeviceMemory は 4MB の大きさで確保されるのに対し、Slab Allocator で管理できる最小サイズは 64 バイトなので 65536 個のオブジェクトを詰められることになります。しかし、1 つのゲームで 64 バイト以下のオブジェクトだけを 65536 個も作るかというとまずそんなことはないので、仮に 64 バイトのスラブが全体に割り当てられてしまうと大半のメモリが使われないおそれがあります。

そこで、Slab Allocator では DeviceMemory のサイズにかかわらず一定数までのオブジェクトのみを管理するようにして、Buddy System で DeviceMemory を最小スラブのサイズに切り分けて管理することを考えます（図 5.3）。こうすれば、ひとつの DeviceMemory 内に複数サイズのスラブを同居させることができ効率的です。

[*7] `float4x4` が 1 個というのはある程度の規模になるとレアケースな気がするので、128 や 256 に上げてもいいかもしれません

[*8] 1MB という数値は、単に Vulkan のデバッグレイヤーでの警告の閾値がこれだからというだけでとくに深い意図はないです

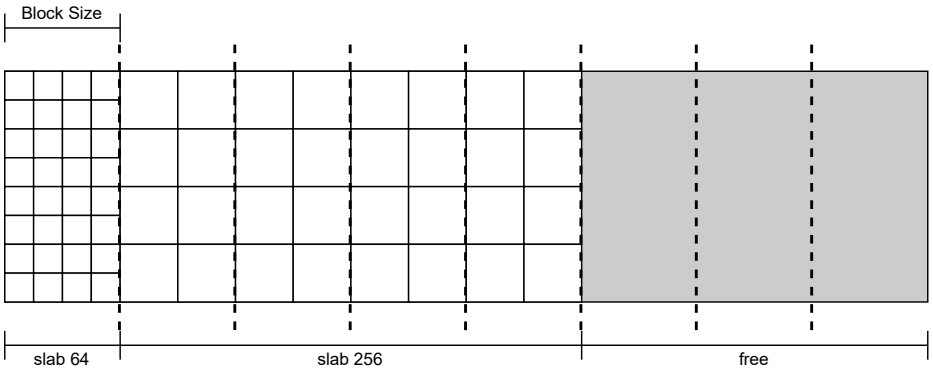


図 5.3: 複数サイズのスラブが入っている様子

Peridot Memory Manager ではひとつのスラブが管理するオブジェクトの数を最大 32 としています。オブジェクトを確保する際、該当のスラブキャッシュに空きがない場合はひとまず 32 個[*9]のオブジェクトが入るスラブを作成し、それをスラブキャッシュに追加してから確保を行うようにしています。

オブジェクト数を最大 32 個にすることで使用中のフラグを 32bit 整数（u32）ひとつで管理できる上、管理に必要ないくつかの操作をビット操作で高速に行えるようになります。

Slab Allocator の空き情報をビット演算で管理する

未使用を 0、使用中を 1 として管理すると、スラブが Full か Empty かは立っているビットの数を数えることで行えます（リスト 5.1）。立っているビットの数を数えるのは近年の CPU では組み込み命令として存在する[*10]ため、非常に高速に行うことができます。

リスト 5.1: 空き状況をチェックするコード（抜粋）

```
pub struct MemoryBlockSlab {
    used_bits: u32,
    /// このスラブが管理しているオブジェクトの数（後述する問題の解決に必要）
    max: u32,
    // ...
}
impl MemoryBlockSlab {
    const fn is_filled(&self) -> bool {
        self.used_bits.count_ones() == self.max
```

[*9] どのサイズでも固定で 32 個だと溢れる場合があることは明らかですが、それについては後述します
[*10] Rust で popcnt 命令を発行させるには別途コンパイラのフラグを指定する必要があります

```
    }

    const fn is_empty(&self) -> bool {
        self.used_bits.count_ones() == 0
    }
}
```

オブジェクトを使用中にする、未使用にする操作は一般的なビットフラグ操作で行えます（リスト 5.2）。

リスト 5.2: 使用中フラグを操作するコード（抜粋）

```
impl MemoryBlockSlab {
    fn mark_used(&mut self, index: usize) {
        self.used_bits |= 1 << index;
    }

    fn mark_unused(&mut self, index: usize) {
        self.used_bits &= !(1 << index);
    }
}
```

未使用のオブジェクトの検索にはループを使う必要はなく、端に立っているビットの数を数えることで実装できます（リスト 5.3）。この操作も組み込み命令を利用して実装されることが多いため非常に高速に行うことができます。

リスト 5.3: 未使用を探すコード（抜粋）

```
impl MemoryBlockSlab {
    const fn get_first_free(&self) -> Option<usize> {
        let x = self.used_bits.trailing_ones();

        // 全部使用中ならtrailing_onesはmaxと同じになるので、
        // それ未満なら空きがあることになる
        if x < self.max {
            Some(x as _)
        } else {
            None
        }
    }
}
```

DeviceMemory 向けの小規模 Buddy System 実装

Slab Allocator 用のメモリを管理するために簡易的な Buddy System を用意します。スラブキャッシュが満杯になった場合、これを使って DeviceMemory 内でまだ空いている箇所のメモリブロックを探して新たなスラブを作成します。

Buddy System はメモリを一定サイズのブロックに分割し、それを 2^n 個の連続したブロックのグループとして管理するものでした。ここでは、2^n の n を**レベル**と表現することにします。たとえば、レベル 0 のグループといえば 1 個連続したブロックのグループを、レベル 1 のグループといえば 2 個連続したブロックのグループを表します。

Buddy System の管理するメモリブロックのサイズは最小のスラブキャッシュのサイズ（64バイト*32個=2048バイト）単位で良いでしょう。

Peridot Memory Manager の Buddy System の実装は空きブロックの先頭インデックスを二重配列で管理する形になっています（リスト 5.4）。外側の配列が先述したレベルをインデックスとしてグループを管理するもので、内側の配列はレベルごとに空きブロックの先頭インデックスを保持しています。

リスト 5.4: Buddy System マネージャー定義

```
pub struct MemoryBlockSlabCacheFreeAreaManager {
    free_block_index_by_chains: Vec<Vec<u32>>
}
```

レベルとブロック数の相互変換は頻出するため関数化しておくと便利です。ブロック数は 2^n に限定できるためビット演算を用いることで高速に相互変換ができます（リスト 5.5）。

リスト 5.5: ブロック数-レベル相互変換（抜粋）

```
const fn level_to_block_count(level: usize) -> u32 {
    // Note: 1 << nは2^nと等価（オーバーフローしない かつ unsignedの場合のみ）
    1u32 << level
}

const fn block_count_to_level(aligned_block_count: u32) -> usize {
    // Note: 末尾の0を数えることは1 << nの逆演算になるので、
    // これでlevel_to_block_countの逆になる
    aligned_block_count.trailing_zeros() as _
}
```

2 の累乗数への切り上げ演算

メモリ確保先のスラブキャッシュを特定する際など、もっとも近い 2^n に切り上げる場面がいくつかありますが、これはビット演算といくつかの加減算だけで実装が可能です[*11]。

まずは最上位のビットだけを立てる関数を定義します（リスト 5.6）。この操作が 2^n に切り上げるために必要になります。

リスト 5.6: 最上位ビットだけ立てる関数

```
/// 最上位ビットのみが立っている状態にする
/// # Example
/// ```
/// assert_eq!(only_top_bit(0b0100_0000), 0b0100_0000);
/// assert_eq!(only_top_bit(0b0110_1101), 0b0100_0000);
/// ```
pub const fn only_top_bit(x: u64) -> u64 {
    1u64 << (64 - (x.leading_zeros() + 1))
}
```

この関数では、ビット幅（64）から「MSB から連続する 0 の数 +1」を引いて、その値分 1 を左にシフトしています。前者の演算でもっとも MSB に近い 1 のビットの位置を計算しており、そのぶん 1 をシフトさせることで最上位の 1 のみが立った状態にできます。

実を言うとこの演算は「2^n への切り捨て」であり、この演算を使うと通常の切り上げの要領で 2^n への切り上げができます（リスト 5.7）。

リスト 5.7: 切り上げ関数

```
/// 2の累乗数に切り上げ
pub const fn round_up_to_next_power_of_two(value: u64) -> u64 {
    // Note: 最上位ビット以下のみ立っている状態
    //  (= 最上位ビットのみ立っている状態 - 1) を足すと
    // 純粋な2の累乗数以外は繰り上がるので、
    // その状態で再び最上位ビットのみの状態にすると切り上げになる
    only_top_bit(value + only_top_bit(value) - 1) as _
}
```

[*11] Rust には`{integer}::next_power_of_two` があるので、これを使えば安全かつもっと楽できます（全部実装しきった後で気づいた）

通常の切り上げ処理と同じく、端数がある場合のみ繰り上がるように加算をしてから端数を切り捨てることで、切り上がった値を計算できます。

5.4 大きなリソースの場合

ここまでの実装でほとんどのリソースに関してはメモリを管理できるようになりましたが、ひとつだけ抜けているケースがあります。

それは「1MB 未満だけどそこそこでかい（400KB とかの）リソース」のケースで、これについてはうまく捌くことができません。DeviceMemory の確保サイズである 4MB の中で 400KB（切り上げると 512KB）ものリソースを 32 個も入れることは当然不可能なわけですが、先ほどまでの実装では `512KB*32`、つまり `16MB` ものメモリブロックを確保しようとして失敗します。

ではどうするかというと、「確保できる分だけのメモリブロックを確保して[*12]、そこに入る個数だけのオブジェクトを管理するスラブを作る」ことにします。実装詳細で「ひとつのスラブが管理するオブジェクトの数を**最大** 32 とする」と書いたのはこのためで、今回のような場合には 32 個未満で管理するようになります。

Slab Allocator では連続したメモリブロックが必要なので「もっとも連続している空きブロック」が「どこからあるか」を Buddy System に問い合わせます。Buddy System では 2^n 個ごとに階層分けして連続した空きメモリブロックを管理しているので、この階層を大きい方から辿っていくことで効率よく最大の個数を求めることができます。最大の空きブロック数とその開始インデックスを探索するコードをリスト 5.8 に示します[*13]。

リスト 5.8: 最大連続ブロック数を探す

```
impl MemoryBlockSlabCacheFreeAreaManager {
    /// returns: (start block number, block count)
    pub fn max_unallocated_memory_block_length(&self) -> (u32, u32) {
        // レベルリストを逆順に探して、
        // 空きがあったらブロック数や最初のインデックスを計算する
        let largest = self
            .free_block_index_by_chains
            .iter()
            .enumerate()
            .rev()
```

[*12] 厳密には全部使うともったいないので、確保できる分の半分だけにしています

[*13] 執筆しながら気づきましたが、このコードだと必ずしも最大のブロック数が出るとは限らないですね。大抵の場合は問題にはならないとは思いますが……

```
            .find(|(_, v)| !v.is_empty())
            .map(|(level, v)| {
                (
                    unsafe { v.first().copied().unwrap_unchecked() },
                    Self::level_to_block_count(level),
                    level,
                )
            });
        let Some((mut starting_number, mut block_count, largest_level)) =
            largest else {
                // 空きがない
                return (0, 0);
            };

        // 下位レベルに連続するメモリブロックがないか？
        for (level, v) in self.free_block_index_by_chains[..largest_level]
            .iter()
            .enumerate()
            .rev()
        {
            let level_block_count = Self::level_to_block_count(level);

            let chainable_before = v
                .iter()
                .find(|&&n| n + level_block_count == starting_number);
            if let Some(&chainable_start) = chainable_before {
                // 前に接続できるものがある
                block_count += level_block_count;
                starting_number = chainable_start;
            }

            let chainable_after = v
                .iter()
                .find(|&&n| starting_number + block_count == n);
            if chainable_after.is_some() {
                // 後ろに接続できるものがある
                block_count += level_block_count;
            }
        }

        (starting_number, block_count)
    }
}
```

これによって確保できる最大ブロック数が出るので、そこから入れられるオブジェクト数を計算してスラブを作成すれば、あとは通常通りです。

5.5 もっと大きなリソースの場合

ここまでは 1MB 未満のリソースに対しての話でした。ここからは 1MB 以上のリソースに対してのメモリ管理戦略になります。

このサイズとなると、同じようにサブアロケーション方式を適用すると未使用領域が大きめに出てしまってムダが多いので直接 DeviceMemory を確保しています。つまりリソースと DeviceMemory が 1:1 の関係になります。

ここで、DeviceMemory に紐づいたリソースがひとつだけであればメモリエイリアシングなどを考慮する必要がなく、そのための検証や制御は必要なくなるはずです。となると、この情報をデバイス側に提供できれば更なる最適化が期待できそうです。

Vulkan には「DeviceMemory を使用しているリソースがひとつだけであり、占有している」ことをデバイス側に伝えるための拡張が存在します。それが `VK_KHR_dedicated_allocation` です。この拡張は Vulkan 1.1 で標準機能に Promote されており、1.1 以上をサポートしている環境であれば拡張の指定なしで使えるようになっています。

すべてのリソースで Dedicated Allocation が最適とは限らないため、利用する際はまずデバイス側に最適かどうかを判定してもらう必要があります。`VK_KHR_dedicated_allocation` 拡張が有効な場合、`VkMemoryRequirements2` の拡張構造体として `VkMemoryDedicatedRequirements` を指定でき、これで判定結果を受け取ることができます（リスト 5.9）。

リスト 5.9: Dedicated Allocation 判定

```
let mut dedicated_req = VkMemoryDedicatedRequirements {
    sType: VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS,
    pNext: std::ptr::null_mut(),
    prefersDedicatedAllocation: 0,
    requiresDedicatedAllocation: 0
};
let mut memory_req = VkMemoryRequirements2 {
    pNext: &mut dedicated_req as *mut _ as _,
    // 残りは省略
};
vkGetImageMemoryRequirements2(device, &info, &mut memory_req);
// dedicated_req.prefersDedicatedAllocation,
// dedicated_req.requiresDedicatedAllocationに結果が入っている
```

新たに確保する DeviceMemory を占有するリソースの指定は `VkMemoryDedicatedAllocateInfo` を使用して行います（リスト 5.10）。この構造体は `VkMemoryAllocateInfo` の拡張構造体なので、`pNext` のチェーンにつなげます。

リスト 5.10: Dedicated Allocation

```
let dedicated_alloc = VkMemoryDedicatedAllocateInfo {
    sType: VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO,
    pNext: std::ptr::null(),
    // bufferかimageにリソースハンドルを指定する
    // 片方はnullにする必要がある
};
let alloc = VkMemoryAllocateInfo {
    pNext: &dedicated_alloc as *const _ as _,
    // 残りは省略
};
```

あとは通常通り bind すれば通常のリソースと変わらず取り扱えます。

5.6 おわり

ここまで VRAM のメモリマネージャーについて解説してきました。

Vulkan におけるメモリ周りはこのように結構考慮することが多く大変なので、普通に Vulkan Memory Allocator に代表されるようなメモリマネージャーを使うことをオススメします。冒頭のコラムでも最後に書きましたが、正直な話自作した一番の理由はロマンだったりします。

ただまあこういうのを作ることで GPU 完全理解に一歩近づけた気もするので、完全理解したい方はぜひとも挑戦してみるとなにかしら得られるものがあるかもしれません。この章がなんらかの参考になれば幸いです。

■コラム: VK_KHR_bind_memory2

冒頭で「リソースオブジェクトの bind といった処理は CPU 負荷が高い」と書きましたが、本編中でそれへの対応策を触れるところがなかったのでおまけとして置いておきます。

タイトルにある拡張を使うことで、複数リソースの bind 処理を一括で行うことができ CPU 負荷を削減できます。この拡張も Vulkan 1.1 で標準機能に Promote されており、Dedicated Allocation と同じく対応した環境であれば使えます。

この拡張ではリスト 5.11 に示すふたつの関数および構造体が追加されます。

リスト 5.11: 追加される関数/構造体

```
pub extern "system" fn vkBindBufferMemory2(
    device: VkDevice,
    bindInfoCount: u32,
    pBindInfos: *const VkBindBufferMemoryInfo
) -> VkResult;

pub extern "system" fn vkBindImageMemory2(
    device: VkDevice,
    bindInfoCount: u32,
    pBindInfos: *const VkBindImageMemoryInfo
) -> VkResult;

pub struct VkBindBufferMemoryInfo {
    // vkBindBufferMemoryの引数にsTypeとpNextつけただけなので省略
}

pub struct VkBindImageMemoryInfo {
    // vkBindImageMemoryの引数にsTypeとpNextつけただけなので省略
}
```

見ての通り、それぞれ従来の bind 関数の引数を構造体にして複数個指定できるようになっています。そのため、従来の bind 関数からの移行は容易ですし、構造体定義を使い回すことで「対応している環境でのみまとめて bind し、非対応環境では個別 bind にフォールバックする」などの柔軟な実装も簡単です。

5.7 参考資料

- https://ja.tech.jar.jp/ac/2018/day22.html
- http://www.coins.tsukuba.ac.jp/~yas/coins/os2-2010/2011-01-11/

執筆者・スタッフコメント

第 1 章 Daisuke Makiuchi / @makki_d

眼鏡っ娘が好きです

第 2 章 Daisuke Yamaguchi / @_gutio_

生成 AI と GitHub Actions がマイブーム。日課はベランダ菜園

第 3 章 Shunsuke Ito / @fgshun

ゲームで遊んでゲームの作りのことを考えている。 Python コードを書いて Python の仕組みのことを考えてもいる

第 4 章 Toshifumi Umezawa

本当にそうなっているのか？ を実際に見に行く作業が好き

第 5 章 Shinya Naganuma / @Pctg_x8

Web とネイティブを行ったり来たりしてる

企画進行・イラスト・デザイン

Satsuki Motokawa

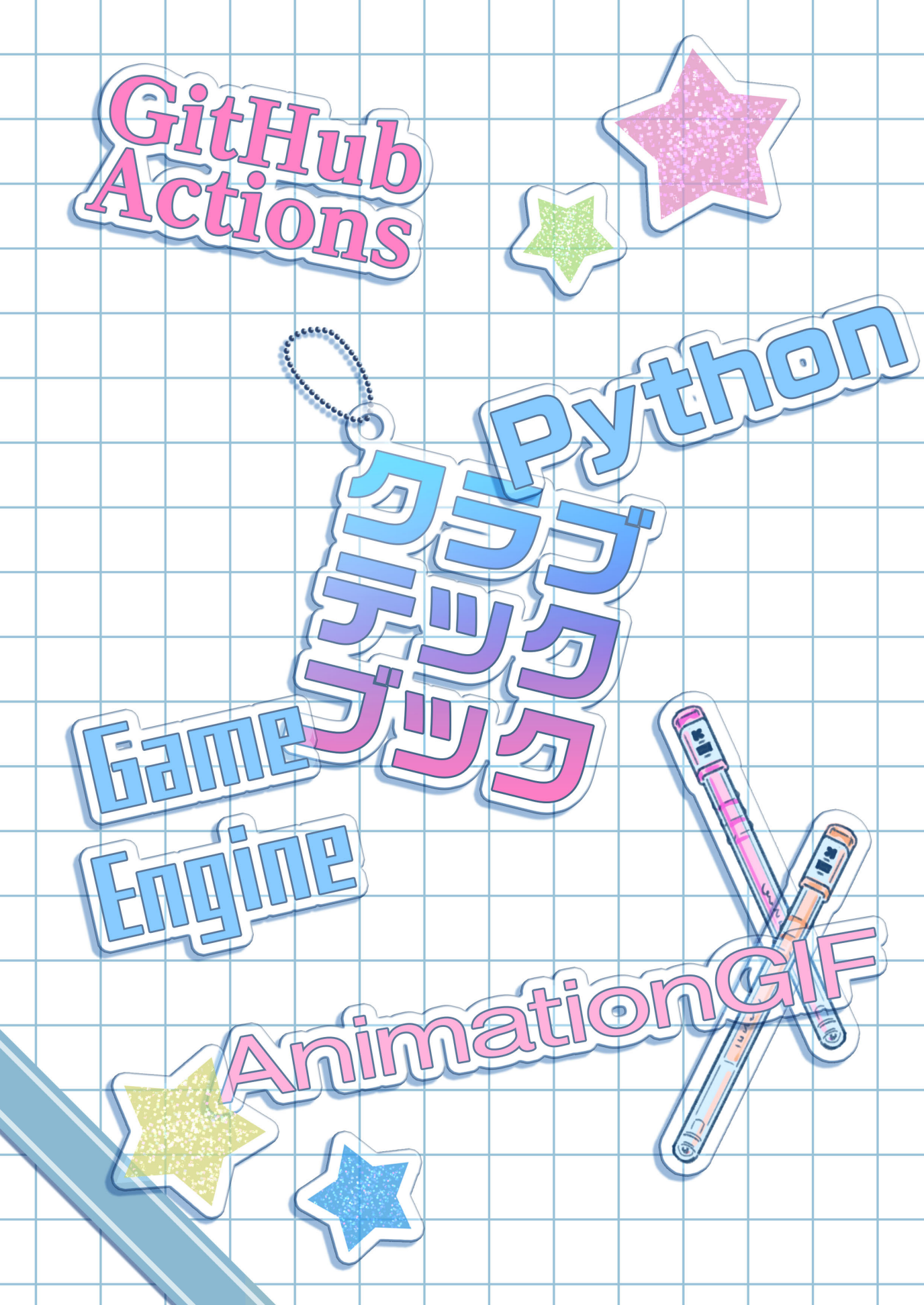
カバーイラスト制作を担当しました！ 見た目変えまくってすみませんでした。平成モチーフが好きなのでたくさん描けて楽しかったです！

既刊・電子版ダウンロード

https://www.klab.com/jp/blog/tech/2023/tbf15.html

KLab Tech Book Vol. 12

2023 年 11 月 11 日　技術書典 15 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


GitHub Actions
Python
クラブ
テック
ブック
Game Engine
AnimationGIF