KLabTechBook

volume.11

1. Dockerを使うなら当然userns-remapしてるよね！
2. Pythonのwith文に詳しくなってみましょう
3. RustでUnityのネイティブプラグインを開発しよう
4. UIToolkitでUnityのブックマークツールを作る
5. PHPのARM向け最適化の中身を見てみた


ZAWA
DIVE INTO
TAP!
THE
GAME
ZAWA…
Remind
XLab Group
TECH
BOOK

KLab Tech Book Vol. 11

KLab 技術書サークル　著

2023-05-20 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 11 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、KLab Tech Book では社内の有志のエンジニアが業務との関連によらず好きな内容を執筆しています。今回もバラエティ豊かな 5 記事を収録しました。表紙や扉絵のデザインも社内のデザイナーの方にご協力いただき、KLab 感溢れる一冊に仕上がっています。

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。実用性が高いものや、一見難しそうでも前提知識からやさしく説明しているものもあり、知らない分野の記事も一読すること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

Docker を使うなら当然 userns-remap してるよね！

Makiuchi Daisuke

Docker[*1]は開発作業において今や無くてはならないツールとなっています。ところで、コンテナを利用してファイルを生成したとき、所有者がホストマシン上で `root` になってしまい困ったことはありませんか？ この章では **userns-remap** [*2]という機能を使って、ファイルの所有権問題を解決する方法を紹介します。

1.1 Docker とは

Docker は、アプリケーションを「コンテナ」と呼ばれる隔離環境で実行するプラットフォームです。コンテナの中にアプリケーションの実行に必要なファイルや設定を詰め込むことで、ホストマシンの環境を変更すること無くアプリケーションを実行できます。またこれらのファイルや設定は「イメージ」という形で保存でき、イメージを共有すれば異なるマシン上でも同一の環境を再現できます。

Docker の利用場面は常駐するサービスの実行環境だけでなく、単発のアプリケーションを実行するものとしても便利です。特にファイル生成ツールのようなアプリケーションは、バージョンによって出力が微妙に異なることも少なくありません。このようなとき、生成ツールをまるごとイメージとして共有しておくと、複数人での作業でもトラブルを避けられます。

ちなみに、この KLabTechBook も pdf の生成に Docker イメージの vvakame/review[*3]を

[*1] `https://www.docker.com/`

[*2] `https://matsuand.github.io/docs.docker.jp.onthefly/engine/security/userns-remap/`

[*3] `https://hub.docker.com/r/vvakame/review/`

利用しています。

1.2 ファイルの所有権問題

Docker でファイル生成する時、ボリュームマウントでホストのディレクトリをコンテナにマウントするのがよくある手法です。リスト 1.1 では、カレントディレクトリを/work にマウントし、/work/hello.txt を生成しています。

リスト 1.1: 単純なファイルの生成

```
$ docker run --rm -v $(pwd):/work busybox sh -c 'echo Hello! > /work/hello.txt'
$ cat hello.txt
Hello!
$ ls -l
total 4
-rw-r--r-- 1 root root 7 Apr 14 18:40 hello.txt
```

ファイルを生成したあと ls コマンドで調べてみると、所有者が root になっていました。これではホストの一般ユーザーでは生成したファイルの編集ができず不便です。これがファイルの所有権問題です。

Docker は Linux カーネルの機能を組み合わせることでコンテナを実現しているため、Windows や macOS で Docker を使うには、仮想マシンで Linux を動かし、その上で Docker を動かすことになります。これによりパフォーマンス上の問題がある一方で、ファイルの所有権問題は仮想マシンのファイル共有機能によって解決されている場合があります。このため Web サーバーなどの Linux マシンで直接 Docker を使おうとしてはじめて所有権問題に悩まされることになった人も多いのではないでしょうか。

Linux では、ユーザーは UID（User ID）という番号で識別されます。ファイルやディレクトリにも UID が割り当てられ、所有者を表します[*4]。一般的に root の UID には 0 が割り当てられていて、特権ユーザーを表します。

Docker のコンテナ内では、基本的には root ユーザーがコマンドを実行するため、リスト 1.1 の例では、作成された hello.txt の所有者として UID=0 が割り当てられました。UID=0 はホスト上でも root なので、hello.txt の所有者は root になってしまいました。

この問題への対処方法はいくつかやり方がありますが、ここでは **userns-remap** を使ってスマートに解決してみます。

[*4] GID（Group ID）の説明は省略します

1.3 userns-remap とは

デフォルトでは Docker コンテナ内の UID は、ホストの UID とそのまま対応しています。このためコンテナ内の root（UID=0）が作ったファイルはホスト上でも root（UID=0）の所有物でした。また、コンテナ内の root はホスト上でも root と同じ権限を持ってしまうので、万が一コンテナによる隔離が破られてしまった場合に重大なセキュリティリスクとなってしまいます。

userns-remap は、コンテナの UID をホスト上の別の範囲にマッピングする機能です。つまり、コンテナ内の root はコンテナ内では UID=0 の特権ユーザーですが、ホスト上では一般ユーザーの UID が割り振られていることになります。これにより、コンテナによる隔離が破られてしまっても、コンテナ内の root は一般ユーザーの権限しか持たないため、ホストマシン全体への悪影響を防げます。また、ホストの root やマッピング範囲外のユーザーの所有物はコンテナ内では nobody の所有物となり、コンテナ内の root であっても許可されていなければ書き換えられなくなります。

この機能を使ってファイルの所有権問題を解決するためには、コンテナ内の root をホストの自分自身の UID にマッピングすればよいです。つまり、コンテナ内の root の所有物は、自動的にホスト上では自分自身の所有物になるという寸法です。それでは、具体的な設定方法を見ていきましょう。

1.4 userns-remap の設定方法

例として、自分自身をログイン名 makki、UID=1000、GID=1000 として、userns-remap の設定方法を説明します。ホストマシン上に、/etc/subuid（リスト 1.2）、/etc/subgid（リスト 1.3）、/etc/docker/daemon.json（リスト 1.4）の 3 つのテキストファイルを用意します[*5]。存在しない場合は作成してください。

リスト 1.2: /etc/subuid

```
makki:1000:65536
```

[*5] dockerd コマンドの引数でも指定できますが、設定ファイルの方がわかりやすいと思います

リスト 1.3: /etc/subgid

```
makki:1000:65536
```

リスト 1.4: /etc/docker/daemon.json

```
{
  "userns-remap": "makki"
}
```

/etc/subuid では、ログインユーザー makki が UID1000 番から 65536 個をマッピング先として利用できるようにする設定です。すなわち、コンテナ内の UID の 0〜65535 を、ホストの 1000〜66535 に対応させるようにできます。/etc/subgid も GID についての同様の設定です。

/etc/docker/daemon.json で、userns-remap に利用するユーザー名を指定しています。もしすでに他の設定項目がある場合は、"userns-remap"の項目をそこに加えます。

これらのファイルを用意したら、Docker デーモンを再起動してください。これで userns-remap が有効になりました。戻すときは daemon.json から"userns-remap"の項目を消して再起動すれば元通りです。

ひとつ注意点として、Docker のイメージやボリュームなどの保存場所が、デフォルトの/var/lib/docker から/var/lib/docker/1000.1000 に変更されます[*6]。このため、これまで使っていたイメージなどをそのまま使いたい場合は自身でエクスポート/インポートする必要があります。

それではさっそく、ファイル生成を試してみましょう。

リスト 1.5: もう一度ファイルを生成

```
$ docker run --rm -v $(pwd):/work busybox sh -c 'echo Hello! > /work/hello2.txt'
$ ls -l
total 8
-rw-r--r-- 1 makki makki 7 Apr 14 20:36 hello2.txt
-rw-r--r-- 1 root  root  7 Apr 14 18:40 hello.txt
```

[*6] 1000.1000 の部分はマッピングされる先頭の UID.GID です

コンテナ内で生成された `hello2.txt` の所有者が自分になっているので、コンテナ外での編集も問題なくできます。

1.5 他の方法との比較

Rootless モード

Rootless モードは、Docker のデーモン自体を一般ユーザーで動かす Docker の機能です。このとき、コンテナ内の `root` はデーモンを動かしている一般ユーザーにマッピングされます。つまり、普段作業しているユーザーで Rootless モードの Docker デーモンを動かせば、ファイルの所有権問題は解決できます。

ただし、Rootless モードでは一部の機能が制限されていて、たとえば TCP/UDP の 1024 未満のポートを Listen できないなど、普通の開発作業を行う上での利便性が損なわれてしまいます。

一方で userns-remap では、Docker デーモン自体は `root` ユーザーで起動しているので、機能的な制限はほとんどありません。

コンテナ内にユーザーを作る

ホストの作業ユーザーと同じ `UID`、`GID` のユーザーをコンテナ内に作り、そのユーザーでファイルを生成すれば、ホスト上でも作業ユーザーの所有物となるはずです。このやり方はウェブ上で多くの記事を見かけますが、ファイルの所有権問題の解決法としては完全に悪手です。

まず、ホストの作業ユーザーの `UID` は環境によって異なる可能性があります。作業者の `UID` が異なっていた場合、生成されたファイルの所有者は別のユーザーになってしまい、問題が解決できていないばかりか、セキュリティリスクにもなりえます。

これを解決するために、作業ユーザーの `UID` に合わせてコンテナのイメージを作り直すこともできますが、これでは同じ環境を再現するという Docker の利点が失われてしまいます。

以前はコンテナ内に専用ユーザーを作ることをセキュリティの面で推奨する向きもありましたが、これは userns-remap や Rootless モードによってほとんど解消されています。常駐型のサービスを動かすコンテナであれば専用ユーザーをつくるほうがよいケースもありますが、ファイル生成のような単発のアプリケーションであれば、Docker の標準的な方法のとおり `root` で動作させるのがよいでしょう。

コンテナ内でファイルを生成するユーザーを `root` に固定しておけば、ホストごとに作業

ユーザーの UID が違っていても userns-remap でそれぞれのホストに適切なマッピングを設定すればよく、問題なく同じイメージを共有することができます。

1.6 userns-remap の注意点

userns-remap では、コンテナ内の UID を単純な連番としてホストの UID に割り当てます。つまり、コンテナ内の root をホストの UID=1000 に割り当てた場合、コンテナ内の UID=1 はホストの UID=1001 に割り当てられてしまいます。

ユーザーが自分のみであれば大きな問題にはなりませんが、ホストマシンを複数のユーザーで共用している場合、他ユーザーのファイルにコンテナ内からアクセスできてしまいます。このような環境では別の方法を考えないといけません。

また、userns-remap を有効にすると、一部のリソースへのアクセスが制限されます。たとえば、GitHub Actions をローカルで実行する「act[*7]」というツールでは、ホストの名前空間のネットワークを必要としますが、userns-remap が有効な場合、ネットワークの名前空間がホストとは別のものになってしまいアクセスできません。

これを回避するには、docker run コマンドなどに--userns=host オプションを指定して実行することで、一時的に userns-remap を無効にできます。特に act では、~/.actrc ファイルに--container-options --userns=host と記載しておけば、内部での docker 呼び出し時にこのオプションが自動的に付加されるため便利です。

1.7 おわりに

Linux のホストマシンで Docker を使ったときに直面しがちなファイルの所有権問題について、userns-remap を使った解決方法を紹介しました。これはセキュリティ面からも有効にしたほうがよいお勧めの機能です。

DockerDesktop が有料化して以降、Windows や macOS では Docker を動かす方法が乱立していて、迷ったり困ったりすることも多いのではないでしょうか。ホストが Linux であれば、公式のパッケージを無料で使えますし[*8]、パフォーマンス面でも圧倒的に有利です。また、ホストを Linux にしてまず直面するであろうファイルの所有権問題は、ここまで解説したとおり userns-remap によって解決できます。

ぜひみなさんも、Docker を使う時はホストを Linux にしましょう。そして Docker コンテナの中ではできるだけ root で動作させてください。くれぐれもよろしくお願いします。

[*7] https://github.com/nektos/act

[*8] とはいえ Docker は大変すばらしいツールなので、気前よく支払いたいですね

第 2 章

Python の with 文に詳しくなってみましょう

Shunsuke Ito / @fgshun

Python の **with 文**。コードブロックを作る、そして入る時と出る時の処理を定義した**コンテキストマネージャ**でラップすることができるという記法です。よくある用途はファイルの閉じ忘れ防止というところでしょうか。しかし、それだけの理解で終わるには勿体無い便利なものであるのです。そんな `with` 文とコンテキストマネージャについて紹介します。

2.1 コンテキストマネージャを使う

`with` 文の使用例としてファイルの閉じ忘れ防止を行ってみましょう。 `with` 文を用いない場合のコードは `try finally` を用いた次のようなものとなります。

リスト 2.1: try finally によるファイル閉じ

```
fobj = open('spam.txt')
try:
  ...
finally:
  fobj.close()
```

このコードの気になるところは、ファイルオブジェクトを使う人が次の 2 点について意識しないといけないということです。

- ファイルオブジェクトを使い終わったら後処理をしなければならない

- 後処理の方法とは close メソッドを呼ぶことである

後処理が必要であるということを意識するのは仕方がないとしても、方法については意識せずに済ませられないものでしょうか？ with 文はこれを可能としてくれます。

ファイルオブジェクトは自身がコンテキストマネージャです。ファイルオブジェクトには自身を閉じるという後処理が実装されています。このため、 with 文を用いて次のように書くことで try finally によるファイル閉じと同じ動作をさせることができます。

リスト 2.2: with によるファイル閉じ

```
with open('spam.txt') as fobj:
  ...
```

2.2 コンテキストマネージャを自作する

ファイルを扱うコンテキストマネージャをあえて自作してみることで、その作り方をみてみます。コンテキストマネージャとは、前処理 __enter__ と後処理 __exit__ という 2 つのメソッドを持ったクラスです。この 2 つのメソッドを次のように実装します [*1]。

- __enter__ 前処理
  - ファイルを開く
  - 開いたファイルを覚えておく
  - 開いたファイルを返す
- __exit__ 後処理
  - 覚えておいたファイルを閉じる

リスト 2.3: ファイルを開き、閉じるコンテキストマネージャの自作例

```
class FileOpener:
  def __init__(self, *args, **kwargs):
    self.fobj = None
    self.args = args
    self.kwargs = kwargs
  def __enter__(self):
    self.fobj = open(*self.args, **self.kwargs)
    return self.fobj
```

[*1] このコンテキストマネージャには再入した際に問題があります。詳しくは「2.7 再入可能性」にて解説します。

```
  def __exit__(self, exc_type, exc_val, exc_tb):
    self.fobj.close()
```

こうして作った FileOpener は組み込み関数 open と同じような使い方ができます。

リスト 2.4: ファイルを開き、閉じる自作コンテキストマネージャの使用例

```
with FileOpener('spam.txt') as fobj:
  data = fobj.read()
print(data)
```

2.3 with 文の動作を追う

with 文が実行された時にどのような処理が行われるのかについて、先ほどの自作コンテキストマネージャを動かした時の動作を例に追ってみましょう。次のような順で処理が走ります。

1. FileOpener コンテキストマネージャオブジェクトが生成される。
2. コンテキストマネージャの __enter__ メソッドの実行。ファイルが開かれる。コンテキストマネージャはこれを保持。
3. __enter__ メソッドが終了。メソッドからの戻り値であるファイルが as 句の fobj 変数へ代入される。
4. with 文ブロックの実行。ファイルを使っての読み込み処理を行ってブロックの処理を完遂。もしくは読み込み失敗などの例外が発生してその場で終了。
5. with 文ブロックからの脱出。コンテキストマネージャの __exit__ メソッドの実行。ファイルが閉じられる。

__exit__ の処理は __enter__ の処理と比べると複雑です。ブロックからの脱出が処理終了による場合と例外によるジャンプであった場合とがありうるからです。with 文にはブロック内で発生した例外・トレースバックをコンテキストマネージャの __exit__ に渡すという処理と、__exit__ からの戻り値が偽値の時、ブロックから送出された例外を再送するという処理があります。これをコードで示すと次のようになります。

リスト 2.5: with 文がもつ例外再送機構

```
hit_except = False
try:
  ...
except:
  hit_except = True
  if not manager.__exit__(*sys.exc_info()):
    raise
finally:
  if not hit_except:
    manager.__exit__(None, None, None)
```

return 文がない素朴な __exit__ らの戻り値は None であり、これは偽値です。このため、先ほどの自作コンテキストマネージャはファイルを閉じた後、 with ブロック内で例外が起きていた場合は再送するという動きをします。

もし、特定の例外を捕捉し処理したいのであれば、 ext_type によって処理を分岐させ、然るべき処理をした後に True を返すとよいです。

2.4 contextlib.contextmanager を使って楽をする

前処理と対になる後処理を定義するだけのシンプルなコンテキストマネージャを作るのであれば、 contextlib.contextmanager が便利です。これは「yield を一度だけ行うジェネレータ」をコンテキストマネージャに変換してくれます。yield 文までの処理が __enter__ に、 yield する値が __enter__ の戻り値に、 yield 文から後の処理が __exit__ に対応します。

ファイルを閉じ忘れないための自作コンテキストマネージャを contextmanager を用いて書き直してみます。

リスト 2.6: contextlib.contextmanager を使う

```
@contextlib.contextmanager
def file_opener(*args, **kwargs):
  fobj = open(*args, **kwargs)
  try:
    yield fobj
  finally:
    fobj.close()
```

このようにシンプルな見た目となってくれます。また、例外の処理は try 文でおこなえばよいです。with 文、 __exit__ メソッドにおける例外の扱いを覚えていなくとも素直に書き下すことが可能です。class 文による表現では処理が 2 つのメソッドに別れているがゆえに、try 文を使えなかったのとは対照的です。

contextmanager はジェネレータの yield で処理が中断する、処理が yield を境目に前半と後半に分かれるという性質をうまく利用している、よくできたデコレータです。これでは表現できない類の処理、たとえば「何度も取得・放棄をくり返すことができるロック」のようなものがあるのはたしかですが、事足りるケースであれば使うのをおすすめします。

2.5 便利な既存コンテキストマネージャたち

Python には便利なコンテキストマネージャたちが用意されています。これらを紹介していきます。

まずは、使用し終えたら後処理をしてくれるものです。gzip.GzipFile, tempfile.TemporaryFile などファイルオブジェクトを扱うもの、 memoryview, mmap.mmap などメモリの確保を行うものなどは、自身を閉じる・解放するといった後処理をしてくれます。

リスト 2.7: gzip.Gzipfile

```
with gzip.Gzipfile('spam.gz', 'rb') as fobj:
  ...
# fobj.close() される
```

データベースなど各種サーバーとの接続関連のオブジェクト、たとえば sqlite3.connect はトランザクションの開始とコミットもしくはロールバックを行ってくれます。ブロック脱出の経路によって動作が変わる、賢いコンテキストマネージャです。

リスト 2.8: sqlite3.connect

```
conn = sqlite3.connect(':memory:')
with conn:
  ...
# conn.commit() もしくは conn.rollback() される
```

10 進浮動小数点数モジュール decimal の設定は動作しているスレッド全体に影響する作り

となっていますが、decimal.localcontext は設定変更の影響を with 文ブロック内に留めてくれるものです。元の設定を保持しておいて、別の設定もしくはコピーされた設定の元でブロック内を処理し、ブロックを抜けたときに元の設定を復元する、という動作をします。

リスト 2.9: decimal.local_context

```
with decimal.local_context() as ctx:
  ctx.prec = 42  # 有効桁数 42 へ
  ...
# 設定が元に戻る
```

排他制御関連のもの、 threading, multiprocessing, asyncio らの Lock などは with 文進入時にロックの取得を試みます。取得できた時点でブロックの処理が開始され、ブロックから抜けるときにロックを解放するという動作をします。

リスト 2.10: threading.Lock

```
lock = threading.Lock()
def do_something():
  with lock:  # ここでロック取得待ちが行われる
    ...
  #  ロック解放が行われる
```

contextlib は コンテキストマネージャ関連のユーティリティがおさめられたモジュールです。これにはコンテキストマネージャ実装の際に役立つという性格のものだけでなく、そのまますぐに使えるものが含まれています。たとえば closing は任意のオブジェクトの close 忘れを防止してくれるものです。

リスト 2.11: contextlib.closing

```
with contextlib.closing(spam()) as ham:  # spam() の結果が ham に代入される
  ...
# ham.close() が呼び出される
```

2.6 コンテキストマネージャを複数使う

with 文のネスト

コンテキストマネージャは複数をまとめてひとつの文脈として扱うことが可能です。そのための最も簡単な方法は with 文をコンテキストマネージャの数だけネストすることです。たとえば、ファイルを読み込み、その内容を別のファイルに書き出すには次のようにします。

リスト 2.12: with 文のネスト

```
with open('spam.txt') as spam:
  with open('ham.txt', 'w') as ham:
    ham.write(spam.read())
```

with 文のネストは 1 つの with 文で書くことが可能です。

リスト 2.13: with 文のネストを束ねる

```
with open('spam.txt') as spam, open('ham.txt', 'w') as ham:
  ham.write(spam.read())
```

扱うコンテキストマネージャが増えて横長になりすぎると感じた時には、括弧でくくることで自由にインデント・改行しつつ書くことが可能です。

リスト 2.14: with 文ヘッダーを括弧でくくる

```
with (
  open('spam.txt') as spam,
  open('ham.txt', 'w') as ham,
):
  ham.write(spam.read())
```

より低レイヤーのツールとしては contextlib.ExitStack が存在します。

リスト 2.15: ExitStack

```
with contextlib.ExitStack() as stack:
  spam = stack.enter_context(open('spam.txt'))
  ham = stack.enter_context(open('ham.txt', 'w'))
  ham.write(spam.read())
```

これは、より複雑な書き方となってしまう代わりに扱うコンテキストマネージャが実行するまでいくつ用意されるかがわからないケースにも対応が可能です。たとえば、可変個のファイルパスをリストで受け取り、すべてを開いて閉じ忘れないようにするには次のようにします。

リスト 2.16: 可変個のコンテキストマネージャを扱う

```
with contextlib.ExitStack() as stack:
  files = [stack.enter_context(open(path)) for path in paths]
  ...
```

2.7 再入可能性

with 文はネストすることが可能なのでした。ここで、同一のコンテキストマネージャを扱うとどのように動作するのでしょうか？ 答えは __enter__ が 2 度呼ばれたのちに __exit__ が 2 度呼ばれる、です。その結果、どのような動きになるかはコンテキスとマネージャ次第となります。

contextlib.contextmanager を用いて作ったコンテキストマネージャはエラーとなります。

リスト 2.17: contextmanager を再入させてみる

```
@contextlib.contextmanager
def a():
    print('spam')
    yield
    print('ham')

mng = a()
with mng:
    with mng:
        ...
```

threading.Lock はデッドロックします。

リスト 2.18: Lock を再入させてみる

```
lock = threading.Lock()
with lock:
    with lock:
        ...
```

一方、threading.RLock は動作します。

リスト 2.19: RLock を再入させてみる

```
lock = threading.RLock()
with lock:
    with lock:
        ...
```

コンテキストマネージャーを再入可能にするためには、「開始回数を覚えておいて初回実行時以外は何もしない」、「前処理時に状態をスタックに積んでおいて後処理時に復旧する」など、再入を念頭においた設計をしなければなりません。標準ライブラリのコンテキストマネージャたちは再入可能である場合、「このコンテキストマネージャは再入可能 (リエントラント) です」と書かれています。逆に、こういった記述がないコンテキストマネージャは再入できないため注意が必要です。

ところで。 リスト 2.3 のコードは再入可能ではありません。__init__ や __enter__ が呼び出される度に fobj が上書きされてしまいます。実用に足るようにするには、再入不可能であり異常動作しうることをドキュメントに記す、不許可の処理として例外とする、前のものをスタックに積んでおいて最後にすべて閉じることで再入可能にする、など、なんらかの備えをしておくとよいでしょう。

2.8 おわりに

対となる処理を文脈を失うことなく扱えるコンテキストマネージャ。それを簡易な書式で扱える with 文。そんな便利な道具を Python でのコーディングのお供にどうぞ。きっと、役に立つことと思います。

第 3 章

Rust で Unity のネイティブプラグインを開発しよう

Yu Kobayashi

はじめまして、あるいはお久しぶりです。前回の vol.13 では、 Raspberry Pi Pico で動作するプログラムを Rust で記述し、組み込み環境において Rust で記述することのメリットについて解説しました。今回は、Rust をネイティブプラグインの開発に活用してそれを Unity から使用する際の注意点や勘所について解説します。

3.1 Rust における FFI (Foreign Function Interface)

Rust では、既存の C/C++ コードを一緒にビルドして Rust コードから使うことができるほか、 C ABI（Application Binary Interface）で関数をエクスポートしたライブラリとしてバイナリビルドすることもできます。単に動的ライブラリ (dll, so, dylib) をビルドするようにするだけならば、通常のライブラリ crate の Cargo.toml に リスト 3.1 のように設定を追加するだけです。

リスト 3.1: Cargo.toml の設定例

```
# 動的ライブラリとしてビルドする場合は "cdylib" を追加する。
crate-type = ["lib", "cdylib"]
```

これだけを設定しても関数はエクスポートされないので、エクスポートしたい関数は リスト 3.2 のように記述します。

リスト 3.2: エクスポートする関数の記述例

```
// no_mangle 属性を付与することにより、エクスポートする関数の mangling（関数名の変換）→
を抑止する。
// extern "C" で呼び出し規約を設定する(後述)。
#[no_mangle]
pub extern "C" fn export_function() {
    // ...
}
```

3.2 Rust で書くメリット・デメリット

一般的に利用される C/C++ で直接ネイティブプラグインを記述するのに比べて Rust で記述するメリットはいくつかあります。

所有権システムやライフタイムに基づく静的で強力なメモリ管理はやはり強力です。C++11 以降は STL が強化されましたが、`std::move()` でムーブした変数の抜け殻にアクセスできてしまうといった落とし穴は未だにあります。Rust ではムーブした変数を再利用しようとするとコンパイルエラーになるなど、メモリ関連で踏みがちな罠をコンパイラが指摘して防いでくれる場面が多くあります。

既存の C/C++ ライブラリを活用する場合、Visual Studio や Makefile を経由して先に別途ビルドしてから (静的動的問わず) リンクすることが多いかと思います。Cargo (Rust 標準のツールチェーン) には build.rs という仕組みがあり、 Rust のソースコードと同時に C/C++ のソースコードをコンパイル・リンクさせることが可能です。この際プラットフォームごとのコマンドなどの違いは Cargo や専用の crate によって吸収されているので、従来の方法よりも簡単にポータビリティーが得られます。

なお、既存の C/C++ ライブラリを Rust 経由でエクスポートする活用法については Cysharp 開発の csbindgen [*1] を使う方法が簡単で良いのでそちらもおすすめです。

Unity 開発においてそもそもネイティブプラグインを使うべきなのか、という問題はあります。実際この記事を執筆するきっかけになった個人プロジェクトでも、当初は全て C# で実装することを検討していました。しかし、当時対象にしていた Unity のバージョンは 2019.x であり、したがって利用可能な言語バージョンは C# 7.3 と微妙に古い仕様のものでした。また、ランタイム側も .NET Framework 4.7.2 相当を想定しなければならず、 .NET 6 環境で C# 10.0 を記述したことがある身としてはあまり歓迎できなかったのです。

[*1] `https://github.com/Cysharp/csbindgen`

ネイティブプラグインとして書くと Unity の .NET ランタイムバージョンをほぼ気にする必要がなくなるという大きなメリットがあります。これを果たしてメリットととる人がどれだけいるかは定かではないですが、少なくとも筆者はメリットだと受け取っています。このような理由から最終的に Rust で記述するという選択をしました。

3.3 unsafe

Rust 外の世界から Rust の世界へ何らかのデータを持ち込む上で避けては通れないのが unsafe の存在です。

リスト 3.3: unsafe ブロックの例

```
// let ptr: *const i32;
let ref = unsafe { &*ptr };
```

この記事において unsafe ブロックを使う主な場面はポインターの参照外し (dereference) です。Rust 外から呼ばれた際に Rust の参照がそのまま渡されるわけではなく何かしらのポインターが渡されるので、それを Rust で扱える参照にキャストするために必要な作業になります。unsafe ブロック内に記述する処理はできるだけ少なくするのが原則です。これは、未定義動作が発生する余地をできるだけ減らすという点において重要なポイントになります。

3.4 Rust 側の実装

ここからは Rust から関数をエクスポートする際に重要なポイントについて解説します。

所有権を意識する

Rust で記述する場合に最も重要なポイントがここになります。

ある構造体 `Foo` を通してなんらかの機能を提供する場合、一般的な Rust コードでは リスト 3.4 のようになるでしょう。

リスト 3.4: 普通に Rust で記述する場合

```
struct Foo {
    value: i32,
}
```

```
impl Foo {
    fn new() -> Foo {
        Foo { value: 0 }
    }

    fn read(&self) -> i32 {
        self.value
    }

    fn write(&mut self, value: i32) {
        self.value = value;
    }

    fn consume(self) -> i32 {
        self.value
    }
}
```

所有権と借用について、このコードに対応させつつ C ABI でエクスポートするためには リスト 3.5 のようにします。

リスト 3.5: C ABI でエクスポートするために記述する場合

```
struct Foo {
    value: i32,
}

#[no_mangle]
pub extern "C" fn foo_new() -> *mut Foo {
    let foo = Box::new(Foo { value: 0 });
    Box::into_raw(foo)
}

// Foo が Drop を実装しない場合、 Rust では自動的に drop が挿入されるが外部にポインタで→
渡した場合はこれが発生しない。
// よって、手動で drop するための関数も実装してエクスポートする必要がある。
pub extern "C" fn foo_destroy(foo: *mut Foo) {
    let foo = unsafe { Box::from_raw(foo) };
    // 必須ではないが明示的に drop する
    drop(foo);
}

// &self に対応する引数として *const T を受け取る。
pub extern "C" fn foo_read(foo: *const Foo) -> i32 {
    let foo = unsafe { &*foo as &Foo };
    foo.value
```

```
}

// &mut self に対応する引数として *mut T を受け取る。
pub extern "C" fn foo_write(foo: *mut Foo, value: i32) {
    let foo = unsafe { &mut *foo as &mut Foo };
    foo.value = value;
}

// self に対応する引数として *mut T を受け取り、 Box::from_raw で所有権を奪う。
pub extern "C" fn foo_consume(foo: *mut Foo, value: i32) {
    let foo = unsafe { Box::from_raw(foo) };
    foo.value = value;
}
```

opaque [*2] なポインタとして扱う構造体のポインタは `Box::new` で生成してから `Box::into_raw` で取得し、 `Box::from_raw` で回収するのが基本になります。また、各操作を行う関数において借用はポインタの参照外しによって参照を得て、`Box::from_raw` を使わないようにします。借用であるべきところでこれを使ってしまうとその関数の終了時に drop によって解放されてしまい、呼び出した側が保持しているポインタが無効になってしまうからです。

メモリレイアウトを意識する

メモリレイアウト、特にアラインメントについても少し注意が必要です。

メモリレイアウトとは、端的には「データが実際にアドレス空間上にどのように配置されるか」を示しているものです。一般にコンパイラは、構造体やクラスのフィールドを定義された順番ではなく並べ替えてメモリ上に配置することがあります。これが行われる理由は CPU がより高速にデータにアクセスできるようにするためであったり、後述するアラインメントを合わせるためであったり様々です[*3]。C# か Rust どちらかの構造体をもう片方に公開する場合、それらのレイアウトが一致するように調整する必要があります。Rust 側では `#[repr(C)]` 属性を構造体に付与することで C 言語と同じルールでレイアウトが決定されるようになります。一方 C# 側では `StructLayout` 属性や `FieldOffset` 属性を利用して、構造体内のフィールドの位置を比較的自由に決定することができます。

アラインメントは「アドレス空間でそのデータの先頭番地を揃える (align) 際の単位 (バイ

[*2] データの詳細な構造について開示せずにポインタのみをやりとりするとき、このポインタは opaque (不透明) であるといいます。

[*3] メモリレイアウトと直接の関係はありませんが、Rust 特有の最適化として「決して null にならないポインタをどこかに含む型を Option でラップした場合 Some/None の表現に別途フィールドを用意せずに null で None を表現する」というものがあります。

ト数)」で、これはアーキテクチャや言語、データ型などによって異なります。Rust では、例えば参照とポインタはそのプラットフォームにおけるポインターの幅、単純な数値型などではその型のサイズにアラインメントされることになっています。.NET 側から Rust に渡せるデータの範囲内ではアライメントは全て同じなので特別な作業は不要ですが、Rust はアライメントが不正なポインタから参照外しをすると未定義動作となるので注意しましょう。

panic を正しく扱う

これはコールバック関数を Rust 側で実装する場合に重要になりますが、そうではない (Rust のコードがそれ以外のコードを跨がない) 場合はかならずしも考慮する必要はないかもしれません。

Rust では panic がいわゆる大域脱出を伴う例外に相当する機能になりますが、これは .NET の例外とも C++ の例外とも異なる機構です。したがって、Rust 以外のスタックフレームをまたいで panic の巻き戻しが発生する可能性がある場合は適切に対応するコードを挿入する必要があります。具体的には panic が発生しうるコードを `catch_unwind` で囲って panic を捕捉[*4]し、一旦コールバック関数などを抜けてスタックフレームを跨いだあとに改めて Rust 側で `resume_unwind` する、という形になります (リスト 3.6)。

リスト 3.6: C ABI でエクスポートするために記述する場合

```
// some_ext_function() というコールバック関数を受け取る外部コードの関数があると仮定。
// Rust 外のコードに向かって panic の巻き戻しが発生すると未定義動作となる。

// 巻き戻し中の panic を一時的に確保しておくための変数。
// このサンプルコードはシングルスレッドを前提としている。
static mut PANIC: Option<Box<dyn Any + Send>> = None;

fn callback_across_ext_code() {
    some_ext_function(rust_callback);
    unsafe {
        if let Some(p) = PANIC {
            resume_unwind(p);
        }
    }
}

// 外部コードから呼ばれる呼ばれる関数。
// この関数が呼ばれている時点ではコールスタックに外部コードが挟まっている状態。
```

[*4] 原則として Rust において panic は捕捉するべきではなくそのままプログラムを終了させるべきですが、子スレッドだけを終了させる場合や今回のように FFI でスタックフレームをまたぐ場合に捕捉することがあります。

```
extern "C" fn rust_callback(external_value: i32) {
    let result = catch_unwind(|| {
        // panic が発生しうる処理をこの中に記述する。

        println!("callback function called");
        if external_value != 0  {
            panic!("not zero");
        }
    }).unwrap_or_else(|p| {
        // catch_unwind のクロージャの中で panic が発生した場合 Err の値として panic →
の情報が返る。
        // ここでは、 unwrap_or_else メソッドを利用して PANIC 変数にそのデータを格納す→
る。

        unsafe {
            PANIC = Some(p);
        }
    });
}
```

3.5 C#/Unity 側の実装

ここからは Rust で実装した機能を呼び出す C#/Unity 側について注意するポイントなどを解説します。

SafeHandle について

C# では Rust 側から渡されたポインタは基本的に `IntPtr` として扱うことになります。一方、.NET には `SafeHandle` クラスがあり、生のポインタを `SafeHandle` の継承クラスでラップして扱うこともできます[*5]。

これについては筆者は「場合によるがどちらかといえば不要」という立場です。特に、opaque なポインタを受け取って引き回す必要がない用途の場合は完全に不要だと思います。また、そのような引き回しが発生する場合でもオブジェクトの確保から解放までのスパンが短く、十分にコード上で仮想的な所有権を追える場合も不要でしょう。一方で、オブジェクトを確保したあと .NET の GC システムに任せて所有権を強く追跡しない場合は `SafeHandle` でラップすることが有効と考えられます。

[*5] `SafeHandle` で実際にラッパークラスを作成する方法については筆者自身がベストプラクティスを完全に把握できていないため割愛させていただきます。

呼び出し規約

Rust 側から関数をエクスポートする際にも注意するべきポイントですが、関数の呼び出し規約はエクスポート・インポートで統一しなければなりません。

呼び出し規約はレジスタやメモリにどのように関数の引数や返り値を配置するかなどについての取り決めで、OS やアーキテクチャによっていくつかの種類が存在します。最もよく使われているのは cdecl と呼ばれているもので、そのプラットフォームで C 言語のコンパイラがするのと同じ方法で配置することを指します。厳密には cdecl は常に同じ規約を指すわけではないのですが、このように指定しておくと一般的にポータビリティーが確保しやすくなります。

ところが、 cdecl ではない呼び出し規約を採用していた比較的有名な API が存在していました。それが x86 版 Windows における Win32 API です。x86 における Win32 API は stdcall というものを採用しており、DLL として関数をエクスポートする際はこれがデファクトスタンダードでした。

x64 版 Windows では Win32 API でも cdecl が使われるようになったため一律で cdecl を指定しても問題は発生しづらくなりましたが、最大限のポータビリティーを確保するために Rust にはより良い方法が存在します。それは、エクスポートする関数の呼び出し規約として `extern "system"` を指定することです。これにより、Rust は「システム標準の呼び出し規約」でエクスポートするようになります。x64 版 Windows や Linux、 macOS では cdecl が、 x86 版 Windows では stdcall が選択されるようになります。

C# 側では呼び出し規約を指定するのに `CallingConvention` 引数を使用し、デフォルト値である `Winapi` は Rust の `extern "system"` 同様にプラットフォームによって変化するものになっています。ただしこちらは x64 版 Windows でも stdcall が選択されるようにとれる記述が MSDN にある[*6]ので、常に cdecl を使用して `CallingConvention` を指定したほうがよいかもしれません。

3.6 おわりに

「Unity のネイティブプラグインを Rust で記述して呼び出す」という行為は必ずしも得策とはいえないものです (大半の場合においては悪手でしょう)。Rust であるだけいくらかましとはいえどもやはりクロスコンパイルする苦労はついてまわるものですし、C# コードを保守できる人が Rust コードも保守できるかといえば必ずしもそうではありません。

[*6] `https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.dllimportattribute.callingconvention?view=net-7.0`

その上で、この行為は「技術的に可能」という以上に「現実的に可能」ということはもっと知られるべきではないでしょうか。ネイティブプラグインを使わざるを得ないとなった場合の選択肢として、 C/C++ で普通に記述する以外の選択肢として Rust で記述することはいくつものメリットがあります。メモリ関連のバグの多くをコンパイル時に防止できるので実際に組み込んだあとのデバッグ作業はより楽になると思います。

第 4 章

UIToolkit で Unity のブックマークツールを作る

Lingjian Wang / @superkerokero

Unity でシーンセットアップやデバッグするときに、頻繁に特定のアセットやシーンオブジェクトを切り替えた経験はないでしょうか？

Project ビューや Hierarchy ビューにはキーワード検索機能がありますが、オブジェクトを変える度に検索キーワードを入れ直さないといけないため、非常に面倒です。Project ビューに「Favorites」という機能があって、検索結果を名前つけて保存できますが、自由にアセットグループを作ることはできず、たくさんの検索キーワードを用意しなければなりません。

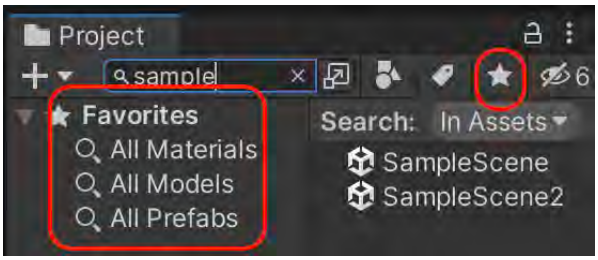

図 4.1: Project ビューや Hierarchy ビューのキーワード検索機能

本当に欲しい機能は、「好きなアセットやシーンオブジェクトだけをコレクションに入れて、一覧表示から素早くアクセスできる」のようなもの。

Google や UnityAssetStore で検索してみると、似たような機能を持つツールは既にいくつかあります：

Bookmark Everything [*1]

必要最低限のアセットブックマーク機能はありますが、シーンオブジェクトをブックマークできない模様。カテゴリ分けはありますが、決められた数種類しかない。ブックマークのコレクションの保存・取り込み機能もない。

Kris's Favourite Assets [*2]**/Kris's Favourite Scene** [*3]

Bookmark Everything をアセットとシーンで分けた感じのツール、機能の違いは特にない模様。

Shortcuts Suite [*4]

こちらは Hierarchy にあるシーンオブジェクトもブックマークでき、おまけに選択された事のあるオブジェクトの履歴機能も。有料だけあって、上記の無料ツールより使えそう。

Scene Bookmarks [*5]

こちらはアセットやシーンオブジェクトではなく、シーンビューのカメラ位置をブックマークするツールとなっている。シーンビューカメラブックマークも割と役立ちます。

色々ありますが、結局どれも機能が不完全で、UI も簡素で必要最低限のものしかありません。次のような最高のブックマークを自分で作ってみたいと思いました：

- Project ビューにあるアセットをブックマークできる
- Hierarchy ビューにあるシーンオブジェクトをブックマークできる
- Scene ビューのカメラ位置をブックマークできる
- 使いやすさを改善するその他オマケ機能（コレクション作成やドラッグ・ドロップ機能など）

ここからは、UIToolkit や新しい EditorTool の API を使った、オリジナルブックマークツールの作り方を解説します。

Bookmark4Unity [*6] は、この記事で解説するブックマークツールの完成品です。ソースコードを参考にして、自分の好みに合わせて改造してみてください。

[*6] https://github.com/superkerokero/Bookmark4Unity

[*1] https://assetstore.unity.com/packages/tools/utilities/bookmark-everything-134467

[*2] https://assetstore.unity.com/packages/tools/utilities/kris-favorite-assets-143105?description

[*3] https://assetstore.unity.com/packages/tools/utilities/kris-favorite-scenes-138708

[*4] https://assetstore.unity.com/packages/tools/utilities/shortcuts-suite-141511

[*5] https://assetstore.unity.com/packages/tools/utilities/scene-bookmarks-82232

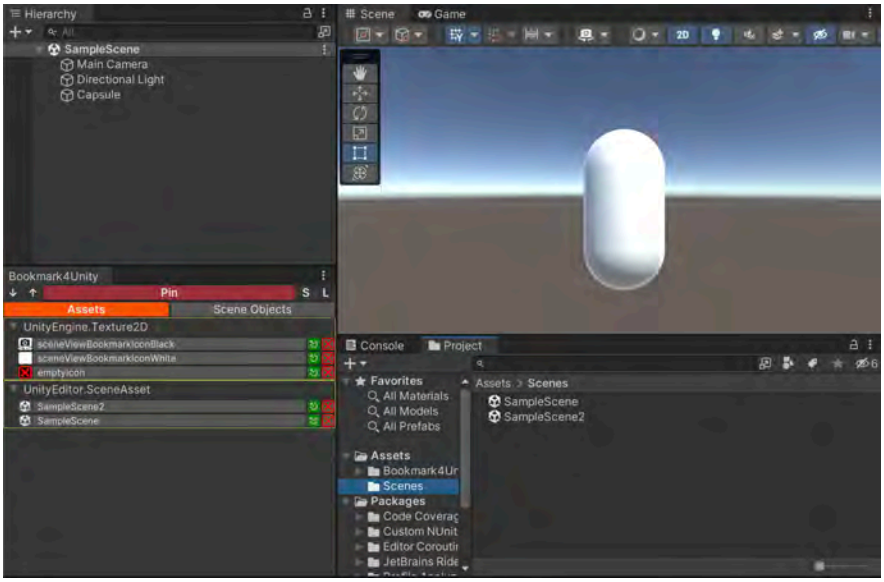

図 4.2: Bookmark4Unity の アセットブックマーク

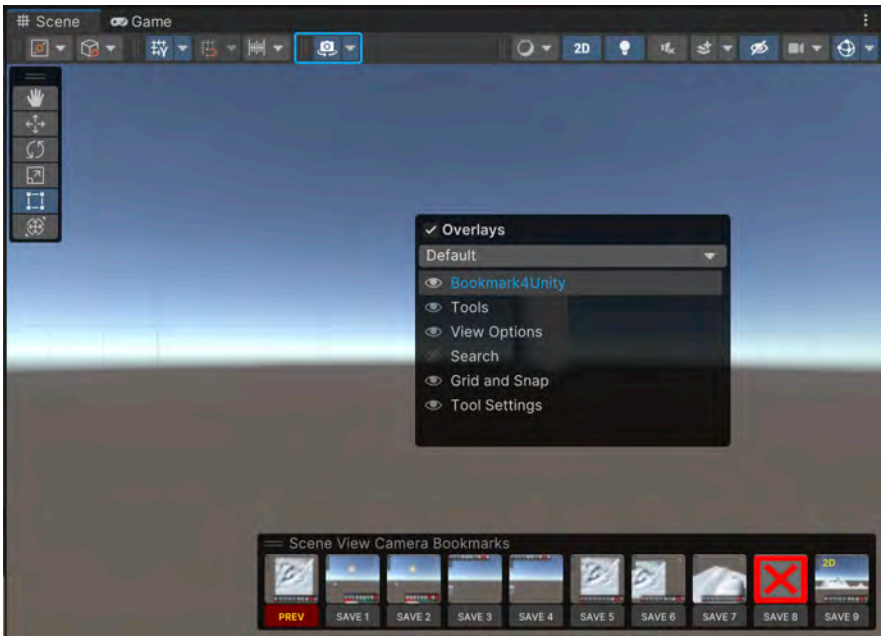

図 4.3: Bookmark4Unity のシーンカメラブックマーク

4.1 基本機能の実装

アセットブックマークのデータ構造

アセットブックマークに必要な情報は、アセットの GUID、パス、名前、タイプです。これらの情報を持つクラスを作ります：

リスト 4.1: アセットブックマークのデータ構造

```
[System.Serializable] // シリアライズ可能にする
public class AssetData
{
    public string guid; // アセットの GUID
    public string path; // アセットのパス
    public string name; // アセットの名前
    public string type; // アセットのタイプ、分類用
}
```

アセットが重複してブックマークされないように、AssetData クラスに IEquatable インターフェースを実装します：

リスト 4.2: アセットブックマークのデータ構造 (IEquatable インターフェース)

```
[System.Serializable]
public class AssetData : IEquatable<AssetData>
{
    // ...

    public override bool Equals(object obj) => this.Equals(obj as AssetData);

    // IEquatable<AssetData> インターフェースの実装
    public bool Equals(AssetData other)
    {
        if (other is null) return false;
        if (System.Object.ReferenceEquals(this, other)) return true;
        return this.guid == other.guid;
    }

    // GetHashCode() も実装する
    public override int GetHashCode() => guid.GetHashCode();

    // == 演算子のオーバーロード
    public static bool operator ==(AssetData lhs, AssetData rhs)
```

```
    {
        if (lhs is null)
        {
            if (rhs is null) return true;
            return false;
        }

        return lhs.Equals(rhs);
    }

    // != 演算子のオーバーロード
    public static bool operator !=(AssetData lhs, AssetData rhs)
        => !(lhs == rhs);
}
```

シーンオブジェクトブックマークのデータ構造

シーンオブジェクトブックマークに必要な情報は、オブジェクトの GUID、名前、シーン名です。これらの情報を持つクラスを作ります：

リスト 4.3: シーンオブジェクトブックマークのデータ構造

```
[System.Serializable] // シリアライズ可能にする
public class GuidData : IEquatable<GuidData>
{
    public string guid; // base64 string
    public string cachedName; // オブジェクトの名前
    public string cachedScene; // オブジェクトが存在するシーン名、
                               // シーン毎にブックマークを分けるために必要
}
```

アセットブックマーク同様、シーンオブジェクトブックマークも重複しないように、GuidData クラスに IEquatable インターフェースを実装します：

リスト 4.4: シーンオブジェクトブックマークのデータ構造 (IEquatable インターフェース)

```
[System.Serializable]
public class GuidData : IEquatable<GuidData>
{
    // ...
```

```
    public override bool Equals(object obj) => this.Equals(obj as GuidData);

    // IEquatable<GuidData> インターフェースの実装
    public bool Equals(GuidData other)
    {
        if (other is null) return false;
        if (System.Object.ReferenceEquals(this, other)) return true;
        return this.guid == other.guid;
    }

    // GetHashCode() も実装する
    public override int GetHashCode() => guid.GetHashCode();

    // == 演算子のオーバーロード
    public static bool operator ==(GuidData lhs, GuidData rhs)
    {
        if (lhs is null)
        {
            if (rhs is null) return true;
            return false;
        }

        return lhs.Equals(rhs);
    }

    // != 演算子のオーバーロード
    public static bool operator !=(GuidData lhs, GuidData rhs)
        => !(lhs == rhs);
}
```

シーンを跨いだオブジェクト参照

Unity ではシーンを跨いだオブジェクト参照は直接できないため、これを実現するための仕組みは自分で用意する必要があります。ここでは、Unity Technologies が提供している Guid Based Reference [*7] をベースに、シーンを跨いだオブジェクト参照を実現するための仕組みを作ります。

Guid Based Reference には、Editor 用のスクリプトと Runtime 用のスクリプトがそれぞれ用意されています。ここでは、Runtime 用のスクリプトのうちの `GuidReference.cs` に少し手を加えて、上記の `GuidData` を使えるようにします。

まずは重複を検知できるように、`GuidReference` クラスに `IEquatable` インターフェースを実装します：

[*7] https://github.com/Unity-Technologies/guid-based-reference

リスト 4.5: GuidReference 変更 1

```
[System.Serializable] // シリアライズ可能にする
public class GuidReference :
    ISerializationCallbackReceiver, IEquatable<GuidReference>
{
    // ...

    public override bool Equals(object obj)
        => this.Equals(obj as GuidReference);

    public bool Equals(GuidReference other)
    {
        if (other is null) return false;
        if (System.Object.ReferenceEquals(this, other)) return true;
        return this.guid == other.guid;
    }

    public override int GetHashCode() => guid.GetHashCode();

    public static bool operator ==(GuidReference lhs, GuidReference rhs)
    {
        if (lhs is null)
        {
            if (rhs is null) return true;
            return false;
        }

        return lhs.Equals(rhs);
    }

    public static bool operator !=(GuidReference lhs, GuidReference rhs)
        => !(lhs == rhs);
}
```

次は GuidData から GuidReference を作るためのコンストラクタを作ります：

リスト 4.6: GuidReference 変更 2

```
[System.Serializable]
public class GuidReference :
    ISerializationCallbackReceiver, IEquatable<GuidReference>
{
    // ...

#if UNITY_EDITOR
```

```
    public string GuidString => Convert.ToBase64String(guid.ToByteArray());
    public string CachedName => _cachedName;
    public string CachedSceneName => _cachedSceneName;

    public GuidData ToData()
    {
        return new GuidData()
        {
            guid = GuidString,
            cachedName = CachedName,
            cachedScene = CachedSceneName
        };
    }
#endif
}
```

GuidReference から GuidData を作るためのコンストラクタも作ります：

リスト 4.7: GuidReference 変更 3

```
[System.Serializable]
public class GuidReference :
    ISerializationCallbackReceiver, IEquatable<GuidReference>
{
    // ...

#if UNITY_EDITOR
    public GuidData ToData()
    {
        return new GuidData()
        {
            guid = GuidString,
            cachedName = CachedName,
            cachedScene = CachedSceneName
        };
    }
#endif
}
```

ここまでで、アセットブックマークとシーンオブジェクトブックマークのデータ構造を用意できたので、次はブックマークの GUI を作ります。

UIToolkit でブックマークの GUI を作る

UIToolkit のおさらい

「UI Toolkit（旧称:UI Elements）」とは、Unity の新 UI システムです。Editor UI、Runtime UI の両方で使えるように設計されています。

UIToolkit の特徴は、HTML と CSS のような構文（UXML と USS）で UI を記述できることです。また、Unity の古い UI システムとは異なり、UI のレイアウトを「UIBuilder」という GUI ツールを使って、直感的に設計することができます。

今回作るブックマークツールは Editor 専用ツールなので、Editor UI である IMGUI で作ることもできますが、UIToolkit を使うことで、より簡単に、より美しい UI を作ることができます。

UIBuilder で UXML と USS を生成する

UIBuilder を使って、UXML と USS を生成します。UXML は HTML のような構文でレイアウトを記述するためのファイルで、USS は CSS のような構文でスタイルを記述するためのファイルです。

UIBuilder は、Window > UI Toolkit > UI Builder から開くことができます。

今回作るブックマークツールは Editor 専用ツールなので、Hierarchy 画面でベースとなる UXML ファイルを選択して、Inspector 画面で 「Editor Extension Authoring」をチェックします。これで Editor 専用のパーツを使えるようになります。

図 4.4: Editor 専用のパーツを使えるように

まず作るのは、GUI のベースとなる Window です。

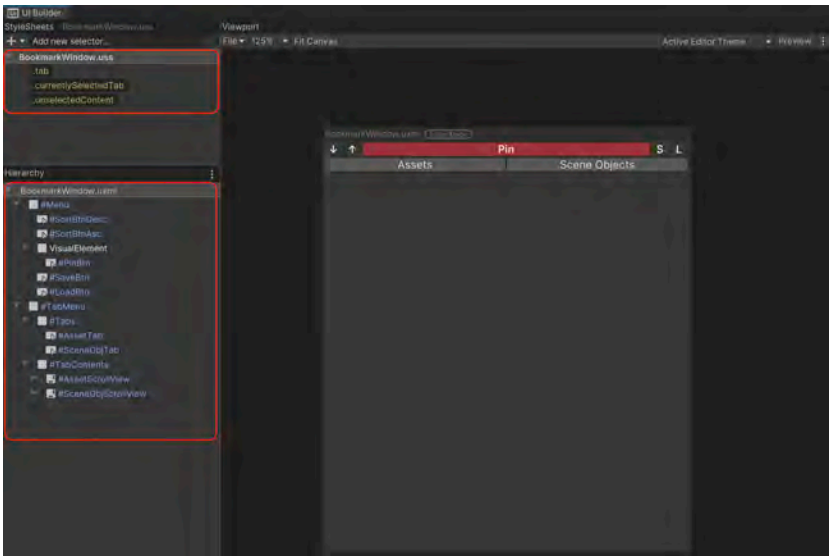

図 4.5: GUI のベースとなる Window

UXML は `BookmarkWindow.uxml` という名前で保存します。
`BookmarkWindow.uxml` に必要な要素は、以下の通りです：

- 「Pin」ボタンなどのボタンを格納する一番上のメニューバー
  - 「Pin」・「ソート」などのボタンは、`Button` 要素で作ります
- 「Assets」・「Scene Objects」表示を切り替えて、ブックマークを表示するためのタブメニュー
  - 「Assets」・「Scene Objects」のタブを切り替えるための `Tabs` 要素（中身は普通の `Button`）
  - 「Assets」・「Scene Objects」のタブの中身を格納する `TabContents` 要素（中身は `ScrollView`）
    - 「Assets」・「Scene Objects」のタブの中身は、それぞれのブックマークを表示するためのボタン。再利用可能にするため、単独の UXML で作ります

USS は `BookmarkWindow.uss` という名前で保存します。
`BookmarkWindow.uss` に必要な要素は、以下の通りです：

- `.currentlySelectedTab`: 選択されているタブの表示設定
- `.unselectedContent`: 選択されてないタブの表示設定

リスト 4.8: .currentlySelectedTab

```
.currentlySelectedTab {
    background-color: rgb(250, 85, 0); /* 背景色 */
    -unity-font-style: bold;  /* フォントスタイル */
}
```

リスト 4.9: .unselectedContent

```
.unselectedContent {
    display: none; /* 非表示 */
}
```

Assets タブの中身を表示するための UXML は `AssetBookmarkBtn.uxml` という名前で保存します。

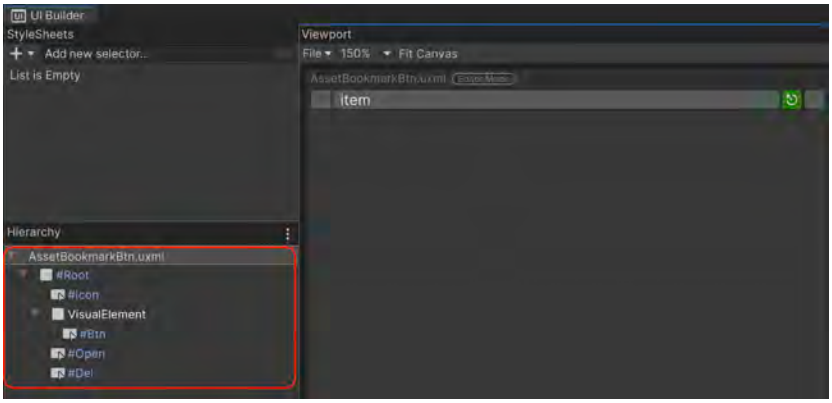

図 4.6: Assets タブの中身を表示するための UXML

`AssetBookmarkBtn.uxml` に必要な要素は、以下の通りです（全部 `Button`）：

- アイコンを表示するための `Icon` 要素
- ブックマークの名前を表示するための `Btn` 要素
- ブックマークを開くための `Open` 要素
- ブックマークを削除するための `Del` 要素

Scene Objects タブの中身を表示するための UXML は `SceneObjectBookmarkBtn.uxml`

という名前で保存します。

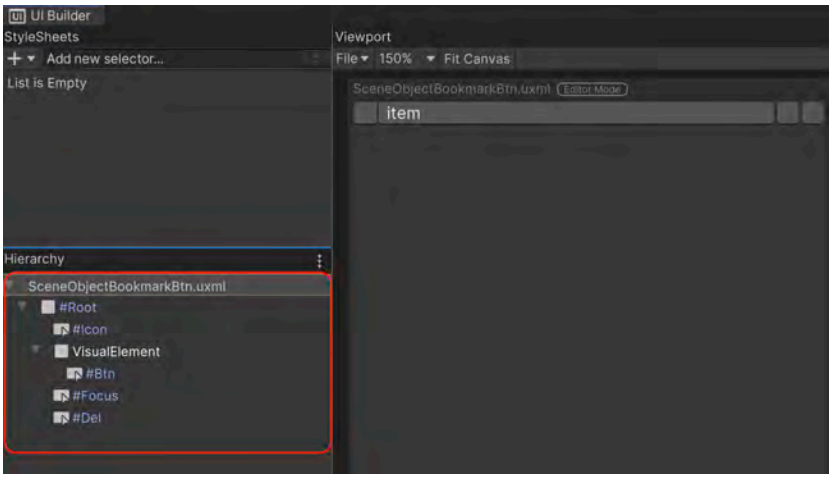

図 4.7: Scene Objects タブの中身を表示するための UXML

`SceneObjectBookmarkBtn.uxml` に必要な要素は、以下の通りです（全部 `Button`）：

- アイコンを表示するための `Icon` 要素
- ブックマークの名前を表示するための `Btn` 要素
- ブックマークのオブジェクトにシーンカメラを合わせるための `Focus` 要素
- ブックマークを削除するための `Del` 要素

UXML と USS を使って EditorWindow を作る

UXML と USS を用意できたら、それを使って EditorWindow を作ります。

EditorWindow に必要そうな using はこんな感じです：

リスト 4.10: EditorWindow に必要そうな using

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using Bookmark4Unity.Guid;
using UnityEditor;
using UnityEditor.SceneManagement;
using UnityEngine;
```

```
using UnityEngine.UIElements;
```

まず、UXML と USS ファイルの読み込みについて解説します。

UXML と USS は、UnityEditor.UIElements 名前空間の AssetDatabase.LoadAsset AtPath メソッドで読み込むことができます。API の名前の通り、ファイルのパスを指定する必要があります。ただ、今回作るツールは、ユーザーがプロジェクトのどこに置いても動くようにしたいので、パスを直接指定するのではなく、各ファイルの GUID を使って読み込みます。

ファイルの GUID は、ファイルを作成する時に自動で割り振られる一意の ID で、ファイルと同じ名前の.meta ファイルに保存されています。それぞれのファイルの GUID を.meta ファイルで調べて、定数として定義しておきます：

リスト 4.11: ファイルの GUID

```
public class Bookmark4UnityWindow : EditorWindow
{
    // UXML と USS の GUID

    private const string UXML_GUID_BookmarkWindow
                            = "7789041336e00410f91f040d6e09f772";

    private const string USS_GUID_BookmarkWindow
                            = "c2575018492804a408595d8f9445083b";

    private const string UXML_GUID_BookmarkGroup
                            = "8898fc16a31fe4cd7887b75843e59563";

    private const string UXML_GUID_AssetBookmarkBtn
                            = "71961e69d456347bd93e543583938f3c";

    private const string UXML_GUID_SceneObjectBookmarkBtn
                            = "95c8e9370e2d54333a3bb77afd33e6a7";
}
```

GUID からファイルを読み込むには、AssetDatabase.GUIDToAssetPath メソッドを使います：

リスト 4.12: GUID からファイルを読み込む

```
var uxml = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>(
               AssetDatabase.GUIDToAssetPath(UXML_GUID_BookmarkWindow));
var uss = AssetDatabase.LoadAssetAtPath<StyleSheet>(
              AssetDatabase.GUIDToAssetPath(USS_GUID_BookmarkWindow));
```

GUI の生成処理は、CreateGUI メソッドに書きます。主な内容は UXML と USS を読み込んで rootVisualElement へ追加することと、必要な UI パーツへの参照の取得です：

リスト 4.13: CreateGUI　メソッド

```
public class Bookmark4UnityWindow : EditorWindow
{
    // VisualTreeAsset への参照
    private VisualTreeAsset bookmarkGroupUxml;
    private VisualTreeAsset assetBtnUxml;
    private VisualTreeAsset sceneObjBtnUxml;

    // タブへの参照
    private Button assetTab;
    private Button sceneObjTab;
    private ScrollView assetScrollView;
    private ScrollView sceneObjScrollView;


    // GUI の生成処理
    private void CreateGUI()
    {
        // タイトルを設定
        titleContent = new GUIContent(Name);

        // UXML と USS を読み込む
        var uxml = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>(
              AssetDatabase.GUIDToAssetPath(UXML_GUID_BookmarkWindow));
        var uss = AssetDatabase.LoadAssetAtPath<StyleSheet>(
              AssetDatabase.GUIDToAssetPath(USS_GUID_BookmarkWindow));
        bookmarkGroupUxml = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>(
              AssetDatabase.GUIDToAssetPath(UXML_GUID_BookmarkGroup));
        assetBtnUxml = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>(
              AssetDatabase.GUIDToAssetPath(UXML_GUID_AssetBookmarkBtn));
        sceneObjBtnUxml = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>(
              AssetDatabase.GUIDToAssetPath(UXML_GUID_SceneObjectBookmarkBtn));
        rootVisualElement.Add(uxml.Instantiate());
        rootVisualElement.styleSheets.Add(uss);
```

```
        // 各種要素への参照を取得
        var pinBtn = rootVisualElement.Q<Button>("PinBtn");
        assetTab = rootVisualElement.Q<Button>("AssetTab");
        sceneObjTab = rootVisualElement.Q<Button>("SceneObjTab");
        assetScrollView = rootVisualElement.Q<ScrollView>("AssetScrollView");
        sceneObjScrollView = rootVisualElement.Q<ScrollView>(
                                                    "SceneObjScrollView");

        // ...
    }
}
```

同じく CreateGUI メソッド内で、各種ボタンのクリックイベントを登録します：

- pinBtn: ピン留めボタン
  - ブックマークの追加操作は、Asset と Scene Object で分ける必要があります。
  - 具体的な管理方法はそれぞれ AssetBookmarkGroup と SceneObjectBookmarkGroup で実装しますが、具体的な内容は割愛します。気になる方は Bookmark4Unity のソースコードを参照してください。
- assetTab: Asset Tab をアクティブにするボタン
- sceneObjTab: Scene Object Tab をアクティブにするボタン

リスト 4.14: 各種ボタンのクリックイベントを登録

```
public class Bookmark4UnityWindow : EditorWindow
{
    // assetブックマークのグループ（アセット種類毎に分ける）
    private readonly Dictionary<string, AssetBookmarkGroup>
                        assetBookmarkGroups = new();

    // scene objectブックマークのグループ（Scene毎に分ける）
    private readonly Dictionary<string, SceneObjectBookmarkGroup>
                        sceneObjectBookmarkGroups = new();

    // タブの状態
    public bool IsAssetTabActive
        => assetTab is not null && assetTab.ClassListContains(
                                    currentlySelectedTabClassName);

    // タブのUSSクラス名
    private const string currentlySelectedTabClassName
                            = "currentlySelectedTab";
```

```
public const string HiddenContentClassName = "unselectedContent";

// GUI の生成処理
private void CreateGUI()
{
    // ...

    // ボタンのクリックイベントを登録
    pinBtn.RegisterCallback<ClickEvent>(PinSelected);
    assetTab.RegisterCallback<ClickEvent>(ActivateAssetTab);
    sceneObjTab.RegisterCallback<ClickEvent>(ActivateSceneObjTab);

    // ...
}

// Asset Tab をアクティブにする
 private void ActivateAssetTab(ClickEvent evt = null)
{
    assetTab.AddToClassList(currentlySelectedTabClassName);
    assetScrollView.RemoveFromClassList(HiddenContentClassName);
    sceneObjTab.RemoveFromClassList(currentlySelectedTabClassName);
    sceneObjScrollView.AddToClassList(HiddenContentClassName);
}

// Scene Object Tab をアクティブにする
private void ActivateSceneObjTab(ClickEvent evt = null)
{
    assetTab.RemoveFromClassList(currentlySelectedTabClassName);
    assetScrollView.AddToClassList(HiddenContentClassName);
    sceneObjTab.AddToClassList(currentlySelectedTabClassName);
    sceneObjScrollView.RemoveFromClassList(HiddenContentClassName);
}

// ピン留めボタンが押されたときの処理
public void PinSelected(ClickEvent evt = null)
{
    if (Selection.activeTransform is null)
    {
        // add assets
        foreach (string assetGUID in Selection.assetGUIDs)
        {
            PinAsset(assetGUID);
        }

        if (Selection.assetGUIDs.Length > 0)
        {
            ActivateAssetTab();
            SaveData();
        }
```

```
        }
        else
        {
            // add scene objects
            if (Selection.transforms.Length < 1) return;

            // pin selected transform
            PinTransform(Selection.transforms[0]);
            ActivateSceneObjTab();
            SaveData();
        }
    }

    // Scene Object をピン留めする
    private void PinTransform(Transform trans)
    {
        var gameObj = trans.gameObject;
        var guidComponent = gameObj.GetComponent<GuidComponent>();
        if (guidComponent == null)
            guidComponent = gameObj.AddComponent<GuidComponent>();
        var reference = new GuidReference(guidComponent);

        // Scene で分ける
        if (sceneObjectBookmarkGroups.ContainsKey(reference.CachedSceneName))
        {
            if (sceneObjectBookmarkGroups[reference.CachedSceneName]
                    .AddSceneObject(reference))
            {
                Debug.Log(
              $"<color=green><b>{Bookmark4UnityWindow.Name}</b></color>: "
            + $"[<color=yellow><b>{reference.CachedSceneName}</b></color>] "
            + $"Scene object <color=red><b>{reference.CachedName}</b></color> "
            + $"bookmarked.");
            }
        }
        else
        {
            var group = new SceneObjectBookmarkGroup(
                reference.CachedSceneName,
                Random.ColorHSV(0f, 1f, 0.65f, 0.65f, 1f, 1f),
                new() { reference },
                new(),
                bookmarkGroupUxml,
                sceneObjBtnUxml);
            sceneObjectBookmarkGroups[reference.CachedSceneName] = group;
            sceneObjScrollView.Add(group.Element);
            Debug.Log(
              $"<color=green><b>{Bookmark4UnityWindow.Name}</b></color>: "
            + $"[<color=yellow><b>{reference.CachedSceneName}</b></color>] "
```

```
            + $"Scene object <color=red><b>{reference.CachedName}</b></color> "
            + $"bookmarked.");
        }
    }

    // Asset をピン留めする
    private void PinAsset(string assetGUID)
    {
        var assetData = new AssetData
        {
            guid = assetGUID,
            path = AssetDatabase.GUIDToAssetPath(assetGUID)
        };
        var asset = AssetDatabase.LoadAssetAtPath<Object>(assetData.path);
        assetData.name = asset.name;
        assetData.type = asset.GetType().ToString();

        // Asset の種類で分ける
        if (assetBookmarkGroups.ContainsKey(assetData.type))
        {
            if (assetBookmarkGroups[assetData.type].Add(assetData))
            {
                assetBookmarkGroups[assetData.type].Root.value = true;
                Debug.Log(
                   $"<color=green><b>{Bookmark4UnityWindow.Name}</b></color>: "
                 + $"<color=yellow><b>{assetData.type}</b></color> "
                 + $"asset <color=red><b>{assetData.path}</b></color> "
                 + $"bookmarked.");
            }
        }
        else
        {
            var group = new AssetBookmarkGroup(
                assetData.type,
                Random.ColorHSV(0f, 1f, 0.65f, 0.65f, 1f, 1f),
                new() { assetData },
                bookmarkGroupUxml,
                assetBtnUxml);
            assetBookmarkGroups[assetData.type] = group;
            assetScrollView.Add(group.Element);
            assetBookmarkGroups[assetData.type].Root.value = true;
            Debug.Log(
              $"<color=green><b>{Bookmark4UnityWindow.Name}</b></color>: "
            + $"<color=yellow><b>{assetData.type}</b></color> "
            + $"asset <color=red><b>{assetData.path}</b></color> bookmarked.");
        }
    }
}
```

データの読み込みも、CreateGUI 内で行うようにします：

リスト 4.15: データの読み込み

```
public class Bookmark4UnityWindow : EditorWindow
{
    // データのコンテナ
    [System.Serializable]
    public class DataWrapper
    {
    public List<GuidData> references = new();
    public List<AssetData> assets = new();
    public List<string> closedAssetTypes = new();
    public bool isAssetTabActive;
    }

    // GUI の生成処理
    private void CreateGUI()
    {
        // ...

        // データの読み込み
        LoadData();

        // ...
    }

    // データの読み込み
    public void LoadData(DataWrapper data)
    {
        // Asset Bookmark Groups の更新
        foreach (var asset in data.assets)
        {
            if (assetBookmarkGroups.ContainsKey(asset.type))
            {
                assetBookmarkGroups[asset.type].Add(asset);
            }
            else
            {
                var group = new AssetBookmarkGroup(
                    asset.type,
                    Random.ColorHSV(0f, 1f, 0.65f, 0.65f, 1f, 1f),
                    new() { asset },
                    bookmarkGroupUxml,
                    assetBtnUxml);
                assetBookmarkGroups[asset.type] = group;
                assetScrollView.Add(group.Element);
            }
```

```
        }

        // Scene Object Bookmark Groups の更新
        foreach (var reference in data.references)
        {
            if (sceneObjectBookmarkGroups.ContainsKey(reference.cachedScene))
            {
                sceneObjectBookmarkGroups[reference.cachedScene]
                    .AddSceneObject(new GuidReference(reference));
            }
            else
            {
                var group = new SceneObjectBookmarkGroup(
                    reference.cachedScene,
                    Random.ColorHSV(0f, 1f, 0.65f, 0.65f, 1f, 1f),
                    new() { new GuidReference(reference) },
                    new(),
                    bookmarkGroupUxml,
                    sceneObjBtnUxml);
                sceneObjectBookmarkGroups[reference.cachedScene] = group;
                sceneObjScrollView.Add(group.Element);
            }
        }

        // 閉じられた Asset Bookmark Groups の更新
        foreach (var assetType in data.closedAssetTypes)
        {
            if (assetBookmarkGroups.ContainsKey(assetType))
                assetBookmarkGroups[assetType].Root.value = false;
        }

        // Asset Tab と Scene Object Tab の表示切り替え
        if (data.isAssetTabActive)
        {
            ActivateAssetTab();
        }
        else
        {
            ActivateSceneObjTab();
        }
    }

    // データの読み込み（EditorPrefs から読み込む）
    public void LoadData()
    {
        if (EditorPrefs.HasKey(PinnedKey))
        {
            var data = JsonUtility.FromJson<DataWrapper>(
                            EditorPrefs.GetString(PinnedKey));
```

```
            LoadData(data);
        }
    }

}
```

開かれてない Scene のオブジェクトのブックマークは使えないので、表示をスッキリさせるために、閉じられた Scene のオブジェクトのブックマークは非表示にしておきます。

Scene が開かれた時や閉じられた時に、Scene Object Tab の表示内容を併せて更新するための処理は Editor イベントに登録します。

データの保存と Editor イベント解除は、`OnDestroy` 内で行うようにします：

リスト 4.16: Editor イベントに登録

```
public class Bookmark4UnityWindow : EditorWindow
{
    // GUI の生成処理
    private void CreateGUI()
    {
        // ...

        // Editorイベントを登録
        EditorSceneManager.sceneOpened += OnSceneLoaded;
        EditorSceneManager.sceneClosed += OnSceneClosed;
        EditorApplication.quitting += OnDestroy;
    }

    // データの保存とEditorイベントの解除
    private void OnDestroy()
    {
        EditorSceneManager.sceneOpened -= OnSceneLoaded;
        EditorSceneManager.sceneClosed -= OnSceneClosed;
        EditorApplication.quitting -= OnDestroy;
        SaveData();
    }

    // Scene が開かれたときに呼び出される
    private void OnSceneLoaded(Scene scene, OpenSceneMode mode)
    {
        UpdateSceneObjectFoldoutStatus();
    }

    // Scene が閉じられたときに呼び出される
    private void OnSceneClosed(Scene scene)
```

```
    {
        UpdateSceneObjectFoldoutStatus();
    }

    // Scene Object Tab の Foldout の状態を更新
    // 開いているシーンのオブジェクトのみを表示し、それ以外は非表示にする
    // シーンが開かれたり閉じられたりしたときに呼び出す
    private void UpdateSceneObjectFoldoutStatus()
    {
        var opendScenes = new HashSet<string>();
        for (int i = 0; i < EditorSceneManager.sceneCount; i++)
        {
            opendScenes.Add(EditorSceneManager.GetSceneAt(i).name);
        }
        foreach (var group in sceneObjectBookmarkGroups.Values)
        {
            group.Refresh();
            group.Root.value = opendScenes.Contains(group.Root.text);
        }
    }
}
```

データの保存関連処理はこんな感じに EditorPrefs を利用して実装します：

リスト 4.17: データの保存関連処理

```
public class Bookmark4UnityWindow : EditorWindow
{
    // 定数を定義
    public const string Name = "Bookmark4Unity";
    public static string Prefix
                          => Application.productName + "_BOOKMARK4UNITY_";
    public static string PinnedKey => Prefix + "pinned";

    // 現在のデータを取得
    private DataWrapper GetCurrentData()
    {
        var data = new DataWrapper();
        foreach (var group in sceneObjectBookmarkGroups.Values)
        {
            data.references.AddRange(group.SceneObjListView.ToData());
        }

        foreach (var group in assetBookmarkGroups.Values)
        {
            data.assets.AddRange(group.Data);
```

```
            if (!group.Root.value) data.closedAssetTypes.Add(group.Root.text);
        }

        data.isAssetTabActive = IsAssetTabActive;
        return data;
    }

    // データの保存（EditorPrefs に保存）
    public static void SaveData(DataWrapper data)
    {
        EditorPrefs.SetString(PinnedKey, JsonUtility.ToJson(data));
    }

    // データの保存（EditorPrefs に保存）
    public void SaveData()
    {
        SaveData(GetCurrentData());
    }
}
```

ここまでで、Asset と Scene Object のブックマークの主な機能は実装できました。

シーンカメラのブックマークのデータ構造

シーンカメラのブックマークに必要なデータは、以下のようなものです：

- カメラの位置
- カメラの向き
- シーンビューのサイズ
- シーンビューの 2D モードの有無

リスト 4.18: シーンカメラのブックマークのデータ構造

```
[System.Serializable]  // シリアライズ可能なクラスにする
struct SceneViewCameraBookmark
{
    public Vector3 pivot; // カメラの位置
    public Quaternion rotation; // カメラの向き
    public float size; // シーンビューのサイズ
    public bool in2DMode; // シーンビューの 2D モードの有無

    public SceneViewCameraBookmark(SceneView sceneView)
```

```
    {
        pivot = sceneView.pivot;
        rotation = sceneView.rotation;
        size = sceneView.size;
        in2DMode = sceneView.in2DMode;
    }
}
```

続いて、シーンカメラのブックマークのデータを扱うための Manager クラスを作ります。

どこからでもアクセスできるように、static クラスにしておきます。

まずは、シーンカメラのブックマークを保存するための EditorPrefs のキーを生成するメソッドを作ります：

リスト 4.19: EditorPrefs のキーを生成するメソッド

```
static class SceneViewBookmarkManager
{
    // シーンカメラのブックマークを保存するための EditorPrefs のキーを生成する
    static string GetEditorPrefsKey(int slot)
    {
        return Bookmark4UnityWindow.Prefix + "SCENE*VIEW_BOOKMARK*" + slot;
    }
}
```

次に、シーンカメラのブックマークを保存するためのメソッドを作ります：

リスト 4.20: ブックマークを保存するためのメソッド

```
static class SceneViewBookmarkManager
{
    // ...

    // シーンカメラのブックマークを指定された Slot に保存する
    public static void SetBookmark(SceneViewCameraBookmark bookmark, int slot)
    {
        WriteToEditorPrefs(slot, bookmark);

        if (slot != previousViewSlot)
        {
            Debug.Log(
        $"<color=green><b>{Bookmark4UnityWindow.Name}</b></color>: "
      + $"[<color=yellow><b>{SceneManager.GetActiveScene().name}</b></color>] "
```

```
      + $"Scene view camera bookmarked at "
      + $"<color=red>slot <b>{slot}</b></color>.");
        }
    }

    // シーンカメラのブックマークを指定された Slot に保存する
    public static void SetBookmark(int slot)
    {
        var bookmark = new SceneViewCameraBookmark(
                               SceneView.lastActiveSceneView);
        SetBookmark(bookmark, slot);
    }

    // シーンカメラのブックマークを EditorPrefs に保存する
    static void WriteToEditorPrefs(
        int slot, SceneViewCameraBookmark bookmark)
    {
        var key = GetEditorPrefsKey(slot);
        var json = JsonUtility.ToJson(bookmark);
        EditorPrefs.SetString(key, json);
    }
}
```

保存されたブックマークを適用するためのメソッドを作ります：

リスト 4.21: 保存されたブックマークを適用するためのメソッド

```
static class SceneViewBookmarkManager
{
    // ...

    // 指定されたブックマークを適用する
    public static void MoveToBookmark(int slot)
    {
        // load bookmark
        var bookmark = ReadFromEditorPrefs(slot);
        var sceneView = SceneView.lastActiveSceneView;
        if (sceneView == null) return;

        // save current scene view camera
        var prevBookmark = new SceneViewCameraBookmark(
                                    SceneView.lastActiveSceneView);

        sceneView.in2DMode = bookmark.in2DMode;
        sceneView.pivot = bookmark.pivot;
        if (!bookmark.in2DMode) sceneView.rotation = bookmark.rotation;
```

```
        sceneView.size = bookmark.size;

        // update previous view slot
        SetBookmark(prevBookmark, previousViewSlot);
        MoveToBookMarkEvent?.Invoke(slot);
    }

    // EditorPrefs に保存されたシーンカメラのブックマークを読み込む
    public static SceneViewCameraBookmark ReadFromEditorPrefs(int slot)
    {
        var key = GetEditorPrefsKey(slot);
        var json = EditorPrefs.GetString(key);
        return JsonUtility.FromJson<SceneViewCameraBookmark>(json);
    }

}
```

最後に、指定された Slot にブックマークがあるかどうかのメソッドや一個前のブックマークに戻るためのメソッドも定義しておきます：

リスト 4.22: 指定された Slot にブックマークがあるかどうかのメソッドなど

```
static class SceneViewBookmarkManager
{
    // ...

    // 最大のブックマーク数
    public const int maxBookmarkCount = 9;

    // 一個前のブックマーク専用の Slot
    const int previousViewSlot = 0;

    // 一個前のブックマークが有効かどうか
    public static bool HasPreviousView => HasBookmark(previousViewSlot);

    // 指定された Slot にブックマークがあるかどうか
    public static bool HasBookmark(int slot)
    {
        var key = GetEditorPrefsKey(slot);
        return EditorPrefs.HasKey(key);
    }

    // 一個前のブックマークに戻る
    public static void ReturnToPreviousView()
    {
```

```
        MoveToBookmark(previousViewSlot);
    }
}
```

ここまでで、シーンカメラのブックマークデータを扱う仕組みを用意できました。あとはこれを使って、GUI を作っていきます。

EditorTool と Overlay を利用する

シーンビューの上に GUI を表示するためには、EditorTool[*8] と Overlay[*9] を利用します[*10]。

Unity の EditorTool は、「SceneView 上で GameObject や Component を操作しやすくする機能を実装するための API」とのことです。これと Overlay を併せて使うことで、シーンビューの上に 自由に GUI を作ることができます。

EditorTool と Overlay については、Unity 公式の学習資料「EditorTools という機能について[*11]」をまず確認することをお勧めします。

大まかな流れとしては、EditorTool の API でシーンビューにカスタムボタンやメニューを追加して、Overlay でカスタムの GUI を作ります。

EditorTool でシーンビューにブックマーク用のボタンを追加する

まずは EditorTool でシーンビューにブックマーク用のボタンを追加します。

`ToobarOverlay` を継承したクラスと `EditorToolbarDropdownToggle` を継承したクラスを作ります：

リスト 4.23: EditorTool でシーンビューにブックマーク用のボタンを追加

```
#if UNITY_2021_2_OR_NEWER
using UnityEditor;
using UnityEditor.Overlays;
using UnityEditor.Toolbars;
using UnityEngine;
using UnityEngine.UIElements;
```

[*8] https://docs.unity3d.com/ja/2021.2/ScriptReference/EditorTools.EditorTool.html

[*9] https://docs.unity3d.com/ja/2021.2/ScriptReference/Overlays.Overlay.html

[*10] 今回使う Overlay の API は、Unity 2021.2 から追加されたものです。Unity 2021.2 以前のバージョンでは使えません。

[*11] https://learning.unity3d.jp/5002/

```
// ToolbarOverlay を継承したクラス
[Overlay(typeof(SceneView), "Bookmark4Unity")]
class SceneViewBookmarkToolbarOverlay : ToolbarOverlay
{
    SceneViewBookmarkToolbarOverlay() : base(SceneViewBookmarkToggle.id) { }
}

// EditorToolbarDropdownToggle を継承したクラス
// IAccessContainerWindow を実装することで、
// Toolbar の EditorWindow にアクセスできるようになる
[EditorToolbarElement(id, typeof(SceneView))]
class SceneViewBookmarkToggle :
    EditorToolbarDropdownToggle, IAccessContainerWindow
{
    // EditorWindow にアクセスするための id、カスタムツール毎に一意になるようにする
    public const string id = "SceneViewBookmarkToolbarToggle";

    public SceneViewBookmarkToggle()
    {
        dropdownClicked += ShowMenu;
        this.RegisterValueChangedCallback(OnValueChanged);
        text = "Toggle";
        // ボタンのアイコン、Texture2D 型
        icon = SceneViewBookmarkManager.SceneViewBookmarkIcon;
        tooltip = "Bookmark the scene view camera.";
    }

    private void OnValueChanged(ChangeEvent<bool> evt)
    {
        SceneViewBookmarkManager.IsOverlayVisible = evt.newValue;
    }

    // EditorWindow にアクセスするためのプロパティ
    public EditorWindow containerWindow { get; set; }

}
#endif
```

ドロップダウンメニューの実装は ShowMenu で行います。ブックマークの操作は前に作った SceneViewBookmarkManager を使います：

リスト 4.24: ドロップダウンメニューの実装

```
// EditorToolbarDropdownToggle を継承したクラス
// IAccessContainerWindow を実装することで、
// Toolbar の EditorWindow にアクセスできるようになる
```

```
[EditorToolbarElement(id, typeof(SceneView))]
class SceneViewBookmarkToggle :
    EditorToolbarDropdownToggle, IAccessContainerWindow
{
    // ...

    // ドロップダウンメニューを表示する
    private void ShowMenu()
    {
        var menu = new GenericMenu();

        for (var slot = 1;
             slot <= SceneViewBookmarkManager.maxBookmarkCount;
             slot++)
        {
            var content = new GUIContent($"Move to Bookmark {slot} &{slot}");

            if (SceneViewBookmarkManager.HasBookmark(slot))
            {
                menu.AddItem(content, false, HandleMoveToBookmark, slot);
            }
            else
            {
                menu.AddDisabledItem(content);
            }
        }

        menu.AddSeparator(string.Empty);

        var returnToPreviousViewContent
            = new GUIContent("Return to Previous View &0");

        if (SceneViewBookmarkManager.HasPreviousView)
        {
            menu.AddItem(
                returnToPreviousViewContent, false,
                SceneViewBookmarkManager.ReturnToPreviousView);
        }
        else
        {
            menu.AddDisabledItem(returnToPreviousViewContent);
        }

        menu.AddSeparator(string.Empty);

        for (var slot = 1;
             slot <= SceneViewBookmarkManager.maxBookmarkCount;
             slot++)
        {
```

```
            menu.AddItem(
                new GUIContent($"Set Bookmark {slot} &={slot}"),
                false, HandleSetBookmark, slot);
        }

        menu.ShowAsContext();
    }

    // ブックマーク適用の処理
    static void HandleMoveToBookmark(object userData)
    {
        var slot = (int)userData;
        SceneViewBookmarkManager.MoveToBookmark(slot);
    }

    // ブックマーク設定の処理
    static void HandleSetBookmark(object userData)
    {
        var slot = (int)userData;
        SceneViewBookmarkManager.SetBookmark(slot);
    }
}
```

UIBuilder で Overlay の UI を作る

Overlay の UI も、Unity の UIBuilder を使って作ります。

Overlay のメニュー自体はただの ScrollView です。

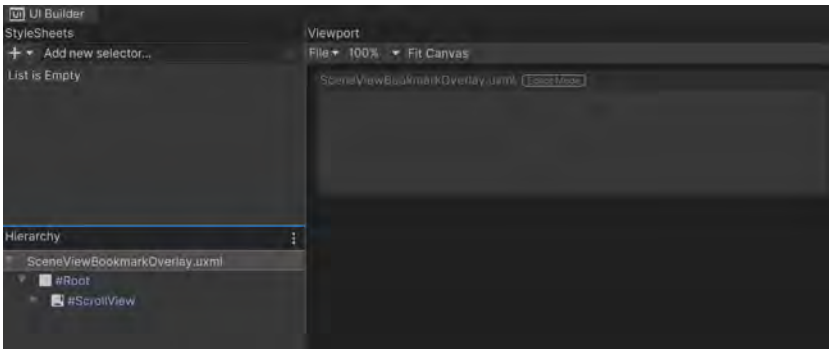

図 4.8: Overlay のメニュー自体

Overlay メニューのボタンは読み込み用のボタンと新規保存用のボタンで構成されます。読

み込み用のボタンは大きく四角形にして、Slot 番号やプレビュー画像を表示します。新規保存用のボタンは小さく、Slot 番号だけを表示します。

「2D」の文字はこのブックマークが 2D モードで保存されているかどうかを表示するだけの `Label` です。2D モード時のみ表示され、3D モード時には空の文字列にします。

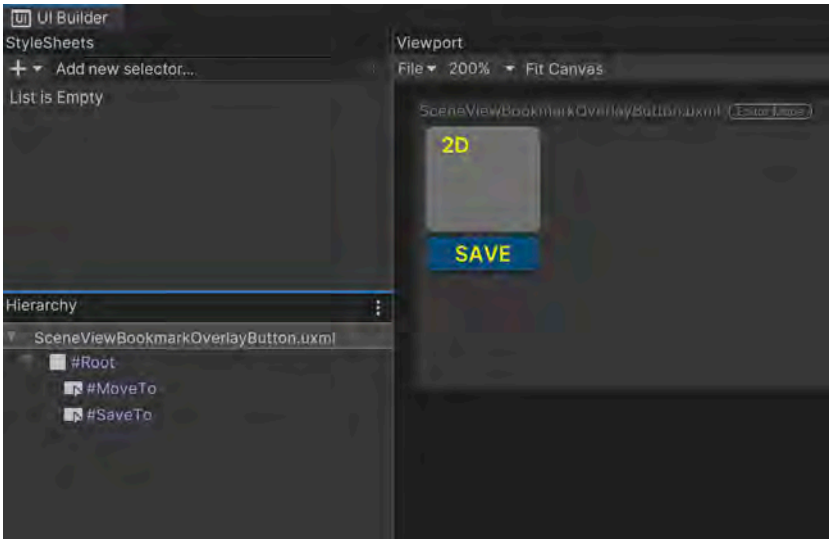

図 4.9: Overlay メニューのボタン

Overlay の 組み込み

UXML の読み込みは `OnCreate` で行います。`OnCreate` は Overlay の初期化時に呼ばれるメソッドです。

表示・非表示の切り替えは `ITransientOverlay` インターフェースの `visible` で制御します。Static クラス `SceneViewBookmarkManager` に `IsOverlayVisible` プロパティを追加して、 `visible` を実装します：

リスト 4.25: Overlay の 組み込み

```
#if UNITY_2021_2_OR_NEWER
using UnityEditor;
using UnityEditor.Overlays;
using UnityEditor.SceneManagement;
using UnityEngine;
using UnityEngine.UIElements;
using Scene = UnityEngine.SceneManagement.Scene;
```

```
[Overlay(typeof(SceneView), "Scene View Camera Bookmarks")]
public class SceneViewBookmarkOverlay : Overlay, ITransientOverlay
{
public bool visible => SceneViewBookmarkManager.IsOverlayVisible;

    private const string UXML_GUID = "0a5cb4dbb8d6b4b8b9aabfe0f499af04";
    private const string BTN_UXML_GUID = "4598592c7ca0248ea97b3bdb91dd66c1";
    private Color SAVE_BTN_COLOR = new(0.25f, 0.25f, 0.25f, 1f);
    private Color PREV_BTN_COLOR = new(0.45f, 0f, 0f, 1f);
    private VisualElement rootVisualElement;
    private readonly Button[] moveToBtns
        = new Button[SceneViewBookmarkManager.maxBookmarkCount + 1];
    private readonly Button[] saveToBtns
        = new Button[SceneViewBookmarkManager.maxBookmarkCount + 1];
    private int prevIndex;

    // Overlay の初期化時に呼ばれる
    public override void OnCreated()
    {
        base.OnCreated();
        rootVisualElement = new VisualElement();
        rootVisualElement.style.height = 60;
        var uxml = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>(
                        AssetDatabase.GUIDToAssetPath(UXML_GUID));
        var btnUxml = AssetDatabase.LoadAssetAtPath<VisualTreeAsset>(
                            AssetDatabase.GUIDToAssetPath(BTN_UXML_GUID));
        rootVisualElement.Add(uxml.Instantiate());
        var scroll = rootVisualElement.Q<ScrollView>("ScrollView");
        for (int i = 0; i <= SceneViewBookmarkManager.maxBookmarkCount; i++)
        {
            var btn = btnUxml.Instantiate();
            moveToBtns[i] = btn.Q<Button>("MoveTo");
            saveToBtns[i] = btn.Q<Button>("SaveTo");
            moveToBtns[i].text = "";

            if (SceneViewBookmarkManager.HasBookmark(i))
            {
                var bookmark = SceneViewBookmarkManager.ReadFromEditorPrefs(i);
                moveToBtns[i].text = bookmark.in2DMode ? "2D" : "";
            }
            else
            {
                moveToBtns[i].text = "";
            }

            var index = i; // saved for lambda functions
            moveToBtns[i].clicked += () =>
            {
                if (!SceneViewBookmarkManager.HasBookmark(index)) return;
```

```
                SceneViewBookmarkManager.MoveToBookmark(index);
            };

            if (i > 0)
            {
                // previous slot
                saveToBtns[i].text = $"SAVE {i}";
                saveToBtns[i].style.backgroundColor = SAVE_BTN_COLOR;
                saveToBtns[i].style.color = Color.white;
                saveToBtns[i].clicked += () =>
                {
                    SceneViewBookmarkManager.SetBookmark(index);
                };
            }
            else
            {
                // normal slots
                saveToBtns[i].text = $"<b>PREV</b>";
                saveToBtns[i].style.backgroundColor = PREV_BTN_COLOR;
                saveToBtns[i].style.color = Color.yellow;
                saveToBtns[i].clicked += () =>
                {
                    if (!SceneViewBookmarkManager.HasBookmark(index)) return;
                    SceneViewBookmarkManager.MoveToBookmark(index);
                };
            }

            scroll.Add(btn);
        }
    }

    // Overlayの表示に必須
    public override VisualElement CreatePanelContent()
    {
        return rootVisualElement;
    }

}
endif
```

4.2 ツールのブラッシュアップ

ここまでで、冒頭で紹介したツールの機能を実装できました。

最後に、ツールの使い勝手を向上させるための工夫を幾つかピックアップして紹介します。

コレクションの保存/読み取り・メニューの追加

メニューを追加して、コレクションをファイルに保存したり、ファイルから読み込んだりできるようにします。ファイルはバイナリ形式で保存します。これで同じプロジェクト内であれば、用途に応じて複数のブックマークコレクションを作ることができます。

IHasCustomMenu インターフェースを実装することで、ツールウィンドウにメニューアイテムを追加できます：

リスト 4.26: コレクションの保存/読み取り・メニューの追加

```
public class Bookmark4UnityWindow : EditorWindow, IHasCustomMenu
{
    // IHasCustomMenu インターフェースの実装
    void IHasCustomMenu.AddItemsToMenu(GenericMenu menu)
    {
        menu.AddItem(
            new GUIContent("Save Collections"), false, SaveDataToFile);
        menu.AddItem(
            new GUIContent("Load Collections"), false, LoadDataFromFile);
    }

    // コレクションをファイルに保存する
    private void SaveDataToFile()
    {
        var path = EditorUtility.SaveFilePanel(
                       "Bookmark4Unity", ".", "", "dat");
        if (path == "") return;

        var data = GetCurrentData();
        using var dataStream = new FileStream(path, FileMode.Create);
        var converter = new BinaryFormatter();
        converter.Serialize(dataStream, data);
    }

    // ファイルからコレクションを読み込む
    private void LoadDataFromFile()
    {
        var path = EditorUtility.OpenFilePanel("Bookmark4Unity", ".", "dat");
        if (path == "") return;

        using var dataStream = new FileStream(path, FileMode.Open);
        var converter = new BinaryFormatter();
        var data = converter.Deserialize(dataStream) as DataWrapper;
        LoadData(data);
    }
}
```

メニュー・ショートカットキーの追加

ショートカットキーの追加は MenuItem 属性を使います。MenuItem を追加することで、Unity の上部メニューにも項目を追加できます：

リスト 4.27: メニュー・ショートカットキーの追加

```
public class Bookmark4UnityWindow : EditorWindow, IHasCustomMenu
{
// ...

    // Unity の MenuItem を追加
    [MenuItem("Tools/Bookmark4Unity/Open Bookmark Window")]
    public static void ShowMyEditor()
    {
        GetWindow<Bookmark4UnityWindow>(Name);
    }

    // ショートカットキーの追加
    [MenuItem("Tools/Bookmark4Unity/Pin Selected %&a")]
    public static void PinSelectedToCollection()
    {
        // ツールウィンドウが開いていれば、そこのメソッドを呼び出す
        if (EditorWindow.HasOpenInstances<Bookmark4UnityWindow>())
        {
            var window = GetWindow<Bookmark4UnityWindow>(Name);
            window.PinSelected();
        }
        // ツールウィンドウが開いていなければ、新しく開いてからメソッドを呼び出す
        else
        {
            ShowMyEditor();
            var window = GetWindow<Bookmark4UnityWindow>(Name);
            window.PinSelected();
        }
    }

}
```

Editor 内のドラッグ・ドロップ対応

Project ウィンドウや Hierarchy ウィンドウからアセットをドラッグ・ドロップでブックマークに追加できるようにするためには、DragUpdatedEvent と DragPerformEvent に関連する操作を実装する必要があります：

リスト 4.28: Editor 内のドラッグ・ドロップ対応

```
public class Bookmark4UnityWindow : EditorWindow, IHasCustomMenu
{
    private void CreateGUI()
    {
        // ...

        // ドラッグ・ドロップ対応
        rootVisualElement.RegisterCallback<DragUpdatedEvent>(
            evt => DragAndDrop.visualMode = DragAndDropVisualMode.Copy);
        rootVisualElement.RegisterCallback<DragPerformEvent>(evt =>
        {
            foreach (var obj in DragAndDrop.objectReferences)
            {
                // assets
                if (AssetDatabase.Contains(obj))
                {
                    var guid = AssetDatabase.AssetPathToGUID(
                                    AssetDatabase.GetAssetPath(obj));
                    PinAsset(guid);
                    ActivateAssetTab();
                    continue;
                }

                // game objects
                if (obj is GameObject go && go.transform is not null)
                {
                    PinTransform(go.transform);
                    ActivateSceneObjTab();
                }
            }

            // アイテムを追加したら保存
            SaveData();
        });
    }
}
```

4.3 まとめ

今回は、Unity 向けのブックマークツールの基本的な実装方法を紹介しました。思ったよりも長くなってしまいましたが、実際に実装してみると、そこまで難しくないことがわかると思います。まだ説明しきれていない部分（シーンビューカメラのプレビュー画像の扱いなど）もありますが、Github で公開している Bookmark4Unity リポジトリを参考にしていただければと思います。

- `https://github.com/superkerokero/Bookmark4Unity`

第 5 章

PHP の ARM 向け最適化の中身を見てみた

Yoshio HANAWA / @hnw

5.1 はじめに

意外と知られていないのですが、ARM アーキテクチャの CPU は我々の身近に多く潜んでいます。スマートフォンの SoC（=CPU）の大半は ARM アーキテクチャですし、各家庭の無線 AP の中身も高確率で ARM だったりします[*1]。

また、昨今ではサーバ用途でも ARM の採用が目立つようになってきました。Raspberry Pi の CPU が ARM 系なのは有名ですし、Apple も M1/M2 という ARM アーキテクチャの独自 CPU を開発しています。また、Amazon Web Services (AWS) も ARM ベースの CPU である Graviton を独自開発しています。

ARM が注目される理由は電力あたりの計算性能が高いこと、言い換えるとコストパフォーマンスが高い点でしょう。しかし、仮に CPU 単体の性能が良いとしても CPU 上で動作するアプリケーションの最適化が不十分だとその実力は発揮できません。アプリケーションの最適化まで含めて考えないと ARM 環境を採用すべきか判断できないと言えるでしょう。

ところで、PHP については ARM 向けの最適化が実装されています。本稿では、この ARM 向け最適化の中身とその背景知識を紹介し、PHP の現在の実装が ARM 環境を活かせるかどうかについて議論します。

[*1] 昔は MIPS も多かったのですが、最近はほぼ ARM なんじゃないでしょうか

5.2 特定アーキテクチャ向けの最適化とは

特定アーキテクチャ向けの最適化と言われてピンと来る人は OSS のソースコードを読んだことがあるのかもしれません。多くの人は何のことかわからないのではないでしょうか。

多くの OSS は C や C++ といった高級言語で実装されており、どんな CPU アーキテクチャでも同じソースコードで動作するように作るのが普通です。一方で、高速化が必要な一部処理だけはアーキテクチャに特化した最適化処理を実装することがあります。

リスト 5.1: 特定アーキテクチャ向けソースコードの一例

```
#if defined(__AVX__)
# if defined(__GNUC__) && defined(__x86_64__)
static zend_always_inline void fast_memcpy(void *dest, const void *src, size_t s→
ize)
{
        size_t delta = (char*)dest - (char*)src;

        __asm__ volatile (
                ".align 16\n\t"
                ".LL0%=:\n\t"
                "prefetchnta 0x40(%1)\n\t"
                "vmovaps (%1), %%ymm0\n\t"
                "vmovaps 0x20(%1), %%ymm1\n\t"
                "vmovaps %%ymm0, (%1,%2)\n\t"
                "vmovaps %%ymm1, 0x20(%1,%2)\n\t"
                "addq $0x40, %1\n\t"
                "subq $0x40, %0\n\t"
                "ja .LL0%="
                : "+r"(size),
                "+r"(src)
                : "r"(delta)
                : "cc", "memory", "%ymm0", "%ymm1");
}
(以下略)
```

たとえば、リスト 5.1 では x86_64 アーキテクチャ[*2]かつ AVX 命令が有効な環境でのみ動作する処理を定義しており、インラインアセンブリで AVX 命令を利用しています。

このように、一部の環境のみ SIMD 命令による高速な処理を利用し、それ以外の環境では従来通りの C の処理を使うのが特定アーキテクチャ向け最適化の典型例です。

[*2] x86 の 64bit アーキテクチャ。amd64、x64 などとも表記されます。

5.3 SIMD 命令とは

SIMD（Single Instruction / Multiple Data）命令は複数のデータに同時に同じ計算を行うアセンブリ命令で、特にグラフィックスや音声処理など多くのデータを一括処理する場合に有用です。

通常、SIMD を実装した CPU では SIMD 専用の演算器および専用の比較的長めのレジスタ[*3]を持っており、1 命令で多くの演算を処理できるため速度面で有利です。たとえば 4 つの 32 ビット整数をそれぞれ加算する場合に、通常の処理では 4 回の加算が必要ですが、SIMD 命令を使用すると 1 命令で 4 つの加算をすることができます。

代表的な SIMD 命令の実装としては、Intel 系 CPU の SSE 命令や AVX 命令、ARM 系 CPU の NEON 命令などがあります。

昨今のコンパイラは非常に賢いので、通常であればアセンブリ命令を人が書くよりも C で書いて C コンパイラに最適化を任せたほうが高性能になります。しかし、SIMD 命令についてはまだコンパイラよりも人間の方がうまく扱えることが多いようで、最適化の文脈で SIMD 命令の話題になることは珍しくありません。

SIMD 命令の利用例

SIMD 命令の典型的な利用例の一つがグラフィックス処理です。

通常、連続メモリ領域にあるピクセルデータ（RGB24bit）は赤緑青がかわるがわる配置されています。このような場合に「赤色だけを強めたい」と言われた場合、3 バイトおきにループを回して取り出すような方法だと効率がよくありません。

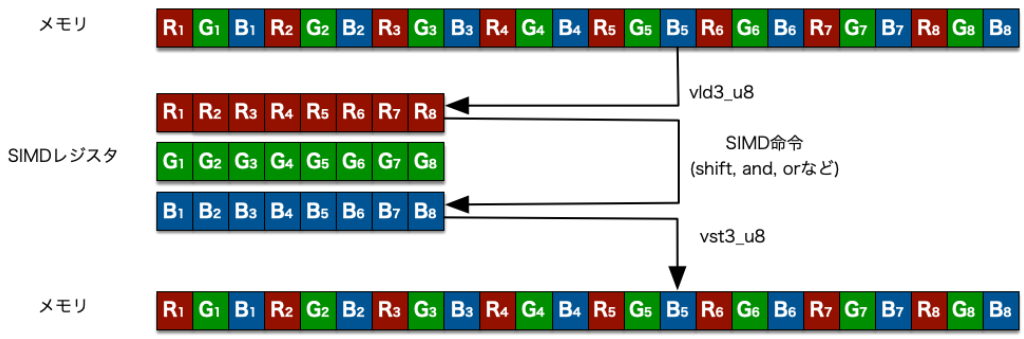

図 5.1: 同じ色に対応するバイトを 1 つのレジスタで一括処理する例

[*3] SSE や NEON なら 128bit 長、AVX なら 256bit 長のレジスタを持っています。

このような場合に SIMD 命令が活躍してくれます。ARM の SIMD 命令である NEON 命令では 3 バイトおきにインターリーブしながら 24 バイトを一括ロードする命令（`vld3_u8`）があるので、簡単に同じ色のバイトを同じレジスタに集めることができます。

このレジスタに対して NEON 命令を使えば「赤色だけを強める」も 1 命令で実現できますので、効率的な処理が実現できます。

5.4 PHP での ARM 向け性能改善の内容

他の OSS と同様、これまで PHP では Intel 系 CPU（x86/x86_64）向けの最適化が多く行われてきましたが、他のアーキテクチャの最適化はあまり進んでいませんでした。しかし、2019 年に 64bit ARM 向け最適化の Pull Request[*4]が出され、同年リリースの PHP 7.4 から利用できる状況になりました。

この ARM 向け対応の面白い点は、この修正を contribute したのが AWS 社員である、という点です。AWS 社では ARM ベースの CPU である Graviton を自社開発しており、Graviton を使ったインスタンスも提供しています。自社の CPU を使ってもらうため[*5]に業務として特定の OSS に contribute していたのなら、かなり珍しい部類の OSS 活動のように思います。

筆者はこの PHP7.4 での 64bit ARM 向け最適化の中身を確認し、表 5.1 にまとめました。やはり NEON 命令（ARM の SIMD 命令）を使ったものが多いことがわかります。

表 5.1: PHP 7.4 の ARM 向け最適化の内容と性能改善

Function	性能改善	最適化の中身
`inc/dec`	1.5x	整数オーバーフロー検出高速化
`add/sub`	1.82x	整数オーバーフロー検出高速化
`hash_init`	1.61x	NEON 命令利用
`hash_func`	1.72x	命令レベル並列性改善
`crc32`	29x	ARM64 命令利用
`strrev`	7.8x	NEON 命令利用
`base64_encode`	3.5x	NEON 命令利用
`base64_decode`	2.15x	NEON 命令利用
`addslashes`	2.8x	NEON 命令利用
`stripslashes`	4.9x	NEON 命令利用

以下、筆者が面白いと感じた最適化処理の詳細を紹介します。

[*4] https://github.com/php/php-src/pull/4094 など

[*5] 企業としての利益追求が目的なら随分遠回りなやり方に思えますが、それも含め面白いと感じました

最適化事例 1：base64_encode

これは PHP 関数 base64_encode の処理本体を NEON 命令で最適化するものですが、かなりトリッキーな処理になっています。

以下、図 5.2 に沿って説明します。

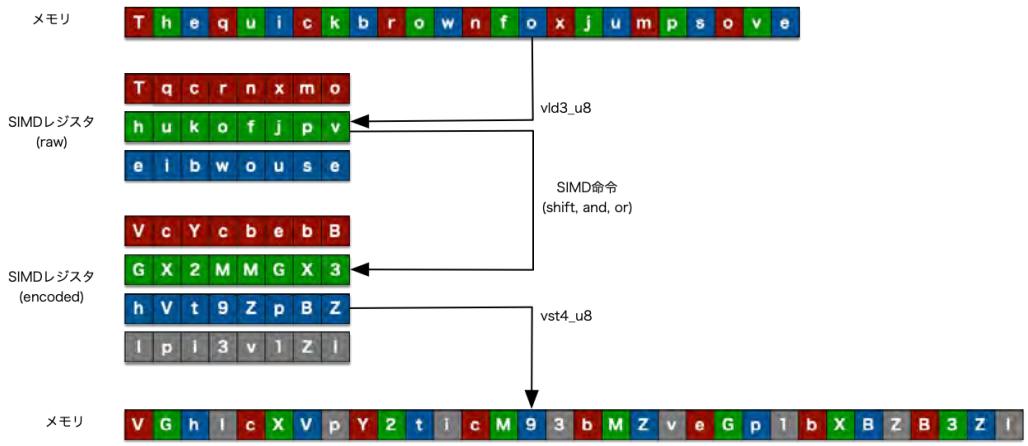

図 5.2: base64_encode の NEON 命令を利用した実装

まず、vld3u8 命令で、Base64 エンコード対象の 24 バイトを 3 バイトおきにインターリーブしながら 3 つの 64bit レジスタにロードします。こうすることで、同じレジスタに元の文字列の 1,4,7,10,…バイト目が入っていることになります。

このようにレジスタにデータをロードすると、Base64 エンコードも NEON 命令で一括処理できます。というのも、Base64 エンコード処理では 6bit ごとに処理を進めていくため、1,4,7,10,…バイト目の上位 6bit がエンコード後の 1,5,9,13,…バイト目に対応する、といった具合になるからです。

同じ調子で SIMD 命令を利用していくと Base64 エンコード結果がインターリーブされた状態で 4 本のレジスタに入ります。ここで vst4_u8 命令を使うことで、4 バイトおきにインターリーブしながらメモリにストアされ、期待した結果が得られます。

ロード時とストア時でインターリーブの間隔が変わるのは非常にトリッキーに思えますが、実は元ネタの論文[*6]があるそうで、最適化界隈では有名なテクニックなのかもしれません。

[*6] https://arxiv.org/abs/1704.00605

最適化事例 2：hash_init

PHP の連想配列を初期化する内部処理 hash_init において、連続メモリ領域にある 32bit 整数 16 個を全て-1 にする処理を高速化するものです。これも NEON 命令を使った性能改善です。

NEON 命令の 128bit レジスタを利用すれば連続領域の 32bit 整数 4 個を vst1q_s32 の 1 命令で初期化できるので、それを利用しています。

こんな地味な処理が高速化できるとは意外な気がしますね。

最適化事例 3：インクリメント/デクリメント/加減算

PHP の整数演算において、整数の範囲を超えたら浮動小数点数に自動キャストする処理があります。この自動キャストが必要かどうかの判定処理をアセンブリで書くことで 1 命令減らせるというものです。

具体的には、C で書くと比較演算で 1 命令必要になってしまうところが、アセンブリで書けば整数オーバーフローフラグを利用して 1 命令減らせる、というものです。確かに、リスト 5.2 とリスト 5.3 を見比べると 1 命令減っているのが分かります。

リスト 5.2: C で書いた場合のアセンブリ出力

```
fast_long_add_function:
        ldr     x0, [x1]
        add     x2, x0, 1
        cmp     x2, x0
        blt     .L11
        str     x2, [x1]
        ret
```

リスト 5.3: 最適化したアセンブリ処理

```
fast_long_add_function:
        ldr x5, [x1]
        adds x5,x5,1
        bvs .L2
        str x5, [x1]
        ret
```

アセンブリ命令を 1 命令減らすだけのメチャクチャ地味な改善ですけど、高頻度で使われる処理なので全体性能への寄与が大きいということなんでしょうね。

5.5 おわりに

AWS の社員さんによる PHP の ARM 向け最適化の中身を探ってみました。Base64 の最適化手法はトリッキーで面白いですけど、実際に効果があるのは「最適化事例 3：インクリメント/デクリメント/加減算」のような地味な最適化なんでしょうね。

AWS が PHP に対して OSS コントリビュートしていること自体あまり知られていないので、これはもっと宣伝してもいいと感じます。AWS さんもブログ記事[*7]でアピールしてはいるんですけど、内容の割に知名度が低いですね…。

[*7] https://aws.amazon.com/jp/blogs/compute/improving-performance-of-php-for-arm64-and-impact-on-amazon-ec2-m6g-instances/

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 Shunsuke Ito / @fgshun

コーヒー淹れて。のんびり、まったりと。

第 3 章 Yu Kobayashi

最近はミニ四駆にハマっています

第 4 章 Lingjian Wang / @superkerokero

最強のブックマークツールを作ろう

第 5 章 Yoshio Hanawa

手元に M1/M2 がないので要らないけど欲しい

企画進行・イラスト・デザイン

梅澤寿史

代表者として企画進行を担当しました。余裕あれば記事も書きたいね…

鈴野元梨

表紙イラストを携わらせていただきました。

クールセクシーな眼鏡女子を描くの楽しかったです、有難うございます！

たねさと

表紙デザイン・ロゴ・扉絵を担当しました。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/tbf14.html

KLab Tech Book Vol. 11

2023 年 5 月 20 日　技術書典 14 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

Klab★TECH★BOOK
Mail
You've got a message!
Remind
Klab Group
GAME
2023
Event Scenario 01