KLab Tech Book Vol.10

KLab Tech Book Vol.10

2022-09-10 KLab 技術書サークル発行

KLab Tech Book Vol. 10

2022-09-10 版　　**KLab** 技術書サークル 発行

はじめに

このたびは本書をお手に取っていただきありがとうございます。本書は KLab 株式会社の有志にて作成された KLab Tech Book の第 10 弾です。

KLab 株式会社では主にスマートフォン向けのゲームを開発していますが、本書では業務との関連によらず、社内のエンジニアが好きな内容で記事を執筆しています。表紙や扉絵は社内のデザイナーの方にも協力していただいています。

技術書典 3 からほぼ毎回参加して新刊を発行し続け、ついに 10 冊目となりました。節目の回ということもあってか 140 ページ越えの過去最高ボリュームになっています。

今回初めて執筆する人もいれば、これまでに何度も執筆してきた人もいます。発行に関わる人は変化しつつも、各々の興味分野について語る形式は変わらず続いています。これは、各々の得意なことや好きなことの深掘りを歓迎することが文化として根付いているからこそだと思います。

本書を通して、そんな KLab の雰囲気を読者のみなさまに感じてもらえるとさいわいです。

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

2000 円の SwitchBot センサーをハックしよう

Yoshio HANAWA / @hnw

1.1　はじめに

突然ですが、読者のみなさんは電気をこまめに切るタイプでしょうか。筆者は自分でもビックリするくらい何でもつけっぱなしにしてしまう人間です。別の部屋のテレビやエアコンをつけっぱなしで寝てしまい翌朝ショックを受けた経験は一度や二度ではありません。

その反省から、SwitchBot のセンサー製品を利用して照明やエアコンが勝手についたり消えたりする仕組みを作ってみたところ、つけっぱなしの事故がほぼ無くなりました。本稿ではその体験談を紹介します。また、SwitchBot のセンサー製品を簡単に扱える Python ライブラリを作成しましたので、その概要を説明します。

1.2　SwitchBot センサーの利用シーン

SwitchBot はスマートホームを実現する製品のブランドです。筆者は「SwitchBot 人感センサー」「SwitchBot 開閉センサー」を利用して、自宅に 2 つの仕掛けを作りました。それぞれについて説明していきます。

シーン 1：キッチンに人が来たら勝手に照明がつく

筆者の自宅はキッチンがリビングの奥まった部分にありまして、昼間でもキッチンが若干暗いんですね。キッチンが暗ければ照明をつければいいじゃない、と言われそうですが、我が家の場合はキッチンのダウンライト 3 個のスイッチが 2m 右と 2m 左に別々に設置されていて、照明をつけるのも消すのも大変だったりします。引っ越し直後にこの問題

点に気づいた時は部屋の設計者を呪いましたが、後の祭りでした。

そこで、「SwitchBot 人感センサー」をレンジフードに設置し、熱源が流し台に近づくと 3 つすべてのダウンライトが点灯、また 3 分連続で熱源を検出しなければ自動でライトを消すような仕組みを作りました。これにより我が家の食事の準備や洗い物の際の不便が激減しました。

▲図 1.1　SwitchBot 人感センサー：レンジフードに台座のマグネットで張り付いています

実装としては人感センサーの反応を自作プログラムで検出し、Google Assistant API で照明をつけるだけの単純なものです。3 つのダウンライトは他社製のスマート電球を利用しており、Google Assistant でコントロールできるようにしてあります。

シーン 2：寝室のドアを閉めると家中のテレビとエアコンと照明がオフになる

筆者は眠いと感じるギリギリまで元気で、眠くなると突然寝るタイプの人間です。リビングで映画を見ていた 10 秒後にフラフラと寝室に向かって即座に寝る、などということも珍しくありません。このような行動で問題になるのが、寝るまでにエアコンやテレビを消す暇がないという点です。

そこで、寝室のドアに「SwitchBot 開閉センサー」を設置し、ドアを閉めると家中のデバイスがオフになるような仕組みを作りました。いくら眠くてもドアを閉めるくらいはできます。この仕組みを作ってから、朝起きて「しまった、エアコンつけっぱなしだった」と後悔することがなくなりました。

▲図 1.2　　SwitchBot 開閉センサー：2 つのパーツに分かれており、ドアとドア枠に設置します

実装は前述のキッチン自動点灯の仕組みと同じです。エアコンやテレビは他社製のスマートリモコンで管理しており、Google Assistant 経由でコントロール可能になっています。

コラム：SwitchBot センサーを 2000 円で買う方法

実は、「SwitchBot 人感センサー」「SwitchBot 開閉センサー」の両商品を Amazon で調べると、どちらも定価 2580 円（税込）で売られています。記事タイトルに 2000 円って書いてあるのに高いな！ と思われるかもしれませんが、ちょっと待ってください。実は SwitchBot 製品は Amazon で頻繁にタイムセール対象になっており、10% オフや 20% オフが珍しくありません。20% オフなら 2064 円ですから、これなら 2000 円と言っても嘘にならないでしょう。筆者は 2021 年のブラックフライデーに 25% オフで購入しており、これが最大の値引き幅だと思われます。読者のみなさんもお得に SwitchBot センサーを購入してみてください。

1.3　SwitchBot の BLE 通信を監視するプログラムを作る

「1.2 SwitchBot センサーの利用シーン」でも少し触れましたが、SwitchBot センサーの状態はプログラムから検出できます。これらのセンサーは本来 SwitchBot 製品同士で

BLE 通信を行う商品ですが、BLE の通信仕様を開発元が公開している[*1]ため、プログラムを書けば PC や Raspberry Pi からも利用できるというわけです。

この仕様をもとにプログラムを書いている人はたくさんいるのですが、汎用的なライブラリが見つからなかったこと、また仕様が煩雑でベタ書きしたくなかったことから、Python ライブラリ switchbotble[*2]を自作してみました。以下では、このライブラリについて紹介していきます。

利用環境

- BLE 対応の PC
- Windows / macOS / Linux
- Python 3.7 以降

実運用には常時起動のマシンが必要です。私は Raspberry Pi 4 を利用しています。

Python ライブラリのインストール

switchbotble は PyPI に登録済みなので、`pip`コマンドでインストールできます。

```
pip install switchbotble
```

利用例

「1.2 SwitchBot センサーの利用シーン」で紹介した 2 つの内容を実現するコードを順に紹介していきます。リスト 1.1 がメイン処理になります。

▼リスト 1.1　switchbotble を使ったメイン処理

```
async def main():
    ble = SwitchBotBLE(motion_timeout = 180)
    while True:
        await ble.start()
        await asyncio.sleep(2.0)
        await ble.stop()

asyncio.run(main())
```

これは switchbotble で利用している BLE ライブラリ bleak を使ったときのメイン処理とほぼ同じです。BLE スキャンの間隔（秒）は`asyncio.sleep()`の引数で指定できます。間隔を短くしすぎると輻輳のような現象が見られたので筆者は 2 秒間隔で運用しています。

[*1] https://github.com/OpenWonderLabs/SwitchBotAPI-BLE
[*2] https://pypi.org/project/switchbotble/

switchbotble ではセンサーの状態変化をイベントと捉え、イベントハンドラの形で処理を書いていきます。SwitchBotBLE()の引数で人感センサーのタイムアウト時間（秒）を設定できます。この場合、180 秒間動体を検出しなければ不検出のイベントを発行するよう指定しています。

リスト 1.2 がイベントハンドラの定義部分です。この 3 つの関数で「キッチンに人が来たら照明をつける」「キッチンから人がいなくなったら照明を消す」「寝室のドアを閉じるとすべて消す」を実現しています。

▼リスト 1.2　switchbotble のイベントハンドラ

```
kitchen = '00:00:5E:00:53:C7'
bedroom = '00:00:5E:00:53:22'

@motion.connect_via(kitchen)
def kitchen_on(address, **kwargs):
    subprocess.Popen(['/home/pi/bin/g', ' キッチンの照明をつけて'])

@no_motion.connect_via(kitchen)
def kitchen_off(address, **kwargs):
    subprocess.Popen(['/home/pi/bin/g', ' キッチンの照明を消して'])

@closed.connect_via(bedroom)
def all_off(address, **kwargs):
    subprocess.Popen(['/home/pi/bin/g', ' 全部のデバイスを消して'])
```

switchbotble のイベントハンドラは Python の関数として定義します。また、各イベントハンドラがどのイベントに対応するかはデコレータで指定します。リスト 1.2 中の motionは動体検知イベント、closedはドアが閉まったイベントになります。それ以外の対応イベントについては表 1.1 にまとめました。

▼表 1.1　switchbotble で対応しているイベント

シンボル	対応するイベント
found	SwitchBot センサーを検出（初回のみ）
motion	動体検出
no_motion	動体不検出
light	明るくなった
dark	暗くなった
opened	ドアが開いた（開閉センサーのみ）
closed	ドアが閉まった（開閉センサーのみ）
entered	ドアが開いて人が中に入った（開閉センサーのみ）
exited	ドアが開いて人が外に出て行った（開閉センサーのみ）
pushed	ボタンが押された（開閉センサーのみ）

SwitchBot センサーが複数ある場合、デコレータの connect_via()の引数に BLE アドレスを渡すことでデバイスを区別できます。switchbotble ではイベントディスパッチ用ライブラリとして blinker[*3]を採用していますので、デコレータの流儀は blinker のマ

[*3] https://pypi.org/project/blinker/ 。Flask でも採用されていますので、人によっては見覚えのあるデ

ニュアルを参照してください。

イベントハンドラ内に書く処理には特に制限はありません。私が利用している gという外部コマンドは Google Assisstant API を呼び出すような OSS[*4]です。

また、イベントハンドラのキーワード引数 deviceでセンサー状態が取れますので、これを利用して複雑な処理を記述することができます。リスト 1.3 では照度センサー[*5]の状態で処理を分岐する例を示します。

▼リスト 1.3　イベントハンドラ内でセンサー状態を利用する例

```
@pushed.connect_via(bedside)
def floorlamp_on_off(address, device, **kwargs):
    if device.light:
        subprocess.Popen(['/home/pi/bin/g', '寝室のフロアランプと照明を消して'])
    else:
        subprocess.Popen(['/home/pi/bin/g', 'フロアランプをつけて'])
```

device.lightはセンサーの周囲が明るければ true、暗ければ false となる変数で、これを利用して「暗いときにボタンを押すとフロアランプがつく」「明るい時にボタンを押すと全部の照明が消える」を実現しています。

pushedは開閉センサーのボタンを押した時に発生するイベントです。ここでは開閉センサーを開閉センサーとして使わず、ベッドサイドに置いてプログラマブルな無線ボタンとして利用しています。ちょっと贅沢な使い方のような気もしますが、これはこれで便利です。

デーモン化

作成した BLE 監視プログラムを常用するため、systemd でデーモン化しましょう。

▼リスト 1.4　/etc/systemd/system/switchbotble-daemon.service

```
[Unit]
Description = switchbotble daemon
Requires = bluetooth.target

[Service]
User = pi
Group = pi
WorkingDirectory = /home/pi/src/github.com/hnw/my-switchbotble-daemon/
ExecStart = /home/pi/src/github.com/hnw/my-switchbotble-daemon/my-switchbotble-daemon.py
ExecStop = /bin/kill ${MAINPID}
Restart = always
RestartSec = 3s
Type = simple

[Install]
WantedBy = multi-user.target
```

コレータかもしれません

[*4] https://github.com/mishushakov/g

[*5] 本稿で扱っている SwitchBot センサー 2 製品はどちらも照度センサーを内蔵しています

リスト 1.4 で示したようなファイルを作成し、次のように systemd 管理下で動作させればマシン再起動後もデーモンとして自動起動するようになります。

```
sudo systemctl daemon-reload
sudo systemctl enable switchbotble-daemon.service
sudo systemctl start switchbotble-daemon.service
```

コラム：人感センサーくらい電子工作すればよくない？

本稿と同じことをするのに、秋葉原で人感センサー単品を買ってきて電子工作するような選択肢もあると思います。電子工作上級者ならその方が安くて良いものができるかもしれません。
ただ、筆者のような電子工作初心者だと長期運用に耐えるような「基板作成」「外装作成」がハードルになってきます。また、見栄えを考えて有線ではなく無線にしたい、なんてことになると「電池駆動」「無線通信」も必要になるので、かなり高コストになってしまうのではないでしょうか。
そう考えると、完成度も高く取り扱いも楽な SwitchBot センサーは十分魅力的に映るはずです。また、これらのデバイスは見た目が大人しく、ご家庭や職場に自然に設置できるのも利点といえるでしょう。

1.4 運用して得た知見

筆者の自宅ではこのシステムを 3 ヶ月ほど運用しており、少しずつ改善を続けています。本章ではその知見を紹介します。

照明がつくまでのタイムラグが数秒ある

「1.2 SwitchBot センサーの利用シーン」で紹介した人感センサーでキッチンの照明をつける仕組みですが、実際に使ってみると流し台に立ってから照明が点灯するまで平均 3 秒ほどのタイムラグがあります。これほどタイムラグがあると、玄関や洗面所など平均滞在時間が短い場所で使うには不便かもしれません。

筆者も最初のうちは反応が遅すぎて「あれ？ 壊れたかな？」と思うことがありましたが、今では慣れてしまって何とも思わなくなりました。

BLE 母艦はセンサーの近くに置くべき

運用してみて気付いたのですが、BLE で使っている 2.4GHz 通信は遮蔽や反射の影響が大きいようで、BLE 母艦（BLE セントラル）とセンサーの距離が離れていると BLE パケットを取りこぼすことがありました。

筆者の自宅の場合、BLE 母艦として使っている Raspberry Pi 4 をセンサーの近くに移動しただけで照明がつくまでの時間が体感で 1 秒ほど改善しました。パケットの取りこぼしが無くなったためだろうと思います。

食洗機が稼働しているとキッチンの照明が勝手に点灯する

筆者の自宅には食洗機があるのですが、食洗機が稼働していると無人のキッチンで電気が付いたり消えたりします。これは心霊現象などではなく、今回作った仕組みが原因です。食洗機は食器を洗っている最中に高温の排水を流し台に流すのですが、人感センサーは熱源の移動を検出するので、食洗機の排水に反応してしまうのです。

これは防ぎようがないため、「そういうもの」として私は納得しています。

bluetoothd を時々再起動する必要がある

原因はわかっていませんが、今回利用している BLE 監視プログラムを Linux 上で動かし続けているとメモリ消費量とロードアベレージが右肩上がりに上がっていきます[*6]。メモリリークは利用ライブラリのバグの可能性がありそうですが、BLE 監視プログラムを再起動すれば直ります。CPU については BLE 監視プログラム再起動だけでは改善せず、bluetoothd を再起動すると直ることがわかりました。

そこで、systemd で今回自作した BLE 監視プログラムと bluetoothd を毎朝再起動するようにしています。

▼リスト 1.5　/etc/systemd/system/switchbotble-daemon-restart.timer

```
[Unit]
Description=Restart switchbotble-daemon daily

[Timer]
OnCalendar=*-*-* 4:45

[Install]
WantedBy=timers.target
```

[*6] 1 日でロードアベレージが 0.10 増加、メモリ消費が 250MB 増加といったペースです

▼リスト 1.6　/etc/systemd/system/switchbotble-daemon-restart.service

```
[Unit]
Description=Restart switchbotble-daemon

[Service]
Type=oneshot
ExecStart=/bin/systemctl stop switchbotble-daemon.service
ExecStart=/bin/systemctl try-restart bluetooth.service
ExecStart=/bin/systemctl start switchbotble-daemon.service
```

リスト 1.5 およびリスト 1.6 で示したようなファイルを作成し、タイマーを有効化します。

```
sudo systemctl daemon-reload
sudo systemctl enable switchbotble-daemon-restart.timer
sudo systemctl start switchbotble-daemon-restart.timer
```

BLE 母艦の死活監視は必須

今回紹介した仕組みは非常に便利なので、逆に正常動作していないとちょっとしたストレスを感じるようになります。その意味で、母艦の死活監視は必須といえるでしょう。

筆者は母艦の Raspberry Pi 4 で Mackerel[*7]を利用しており、死活監視のアラートを個人の Slack に飛ばしています。実は一度だけアラートが飛んできたことがあるのですが、迅速に対応して事なきを得ました。

ちなみに、メモリリークやロードアベレージの増加に気づけたのも Mackerel のおかげです。はてなさんには足を向けて眠れません。

1.5　おわりに

本稿では、SwitchBot センサーを自作プログラムから利用する事例を紹介しました。今回紹介したセンサー 2 つは製品としての完成度が高いだけでなく、BLE 仕様が公開されていて自分でハックできるのが最高ですね。

実は、今回の内容であればスマートリモコンやスマート電球を SwitchBot 製品で統一すればプログラミングなしで同じことを実現できます。とはいえ、すでに他社製品を持っているのに買い換えるのはナンセンスですから、本稿で紹介したように他社製品と混在して使えるのは大きなメリットといえるでしょう。

筆者は SwitchBot センサーを自宅のハックに使いましたが、たとえば会社の会議室の入退室管理などにも活用できるのではないでしょうか。読者のみなさんもよい応用を思いついたら教えてもらえると嬉しいです。

[*7] はてな社のサーバー監視サービス。 https://ja.mackerel.io/

第 2 章

Raspberry Pi Pico と Rust で漢字を描画しよう

Yū KOBAYASHI

Rust、好きですか？ 僕は大好きです。業務時間外で書いたコードの 99% を Rust が占める程度には大好きです。この記事では、そんな Rust を組み込み用途として Raspberry Pi Pico のファームウェアを記述するのに使っていきます。

2.1 はじめに

Raspberry Pi Pico / RP2040 について

Raspberry Pi Pico（以下 Raspi Pico）は、シングルボードコンピューター Raspberry Pi（以下 Raspi）で有名な Raspberry Pi 財団が 2021 年にリリースしたマイコンボードです。搭載している SoC は同財団が設計・販売している **RP2040**[*1] という Arm Cortex-M0+ ベースのものです。

通常の Raspi と違い、 Raspi Pico は Linux を動作させることを目的としたものではありません。どちらかといえば Arduino などに近い性質のものです。主要なスペックを表 2.1 に示します。

特筆すべき点としてはファームウェア[*2]を格納する不揮発ストレージが搭載されていないということです。Raspberry Pi Pico は、 RP2040 に外付けで 2MB の QSPI Flash と周辺回路を搭載して販売しているものになります。つまり、「RP2040 を搭載するマイコンボード」というものは他にも存在しうるのです。実際に、 Arduino Nano RP2040 や Adafruit Feather RP2040 といったものがサードパーティーから販売されています。

[*1] ちなみに、型番の 2040 という部分はスペックの一部を数字に変換して表しているものです。

[*2] このようなマイコンで動作させるプログラムは広義的にファームウェアと呼ばれることもあり、この記事ではファームウェアで統一します。

▼表 2.1　RP2040 のスペック

項目	詳細
アークテクチャ	Arm Cortex-M0+
動作周波数	最大 133MHz
コア数	2 コア
SRAM	264KB (256KB + 8KB)
不揮発ストレージ	なし
ペリフェラル	UART*2, SPI*2, I2C*2, PWM*16, USB, etc...

RP2040 に搭載されているペリフェラル[*3]は、 表 2.1 で挙げた以外にも小さなステートマシンを実装可能なプログラマブル IO、各種数値演算のアクセラレーター、スピンロックなどがあります。 素の Cortex-M0+ の命令セットで不足しがちな部分[*4]を補っている印象です。

開発環境として、公式には C/C++ SDK と MicroPython のポーティングが提供されています。前者は、PC でコードを書いてビルドしたファームウェアのバイナリを USB 経由で直接転送して実行する方式です。一方後者は、 MicroPython のバイナリをあらかじめ転送しておき、 USB シリアルで REPL のようにプログラムを実行する方式になります。お手軽に試すなら後者ですね。

組み込み環境で Rust を書くということ

Rust は「効率的で信頼できるソフトウェアを誰もがつくれる言語」として、 CLI ツールや Web バックエンド 最近は Web フロントエンドにまで進出しています。

Rust が輝くのは、何も OS がある環境だけではありません。 OS そのものの開発や主要な OS が動作できない組み込み環境など、ベアメタル開発においても Rust の特長は失われることはありません。 malloc/free に代表されるアロケーターが存在せずヒープ領域を使えない環境でも、 enum や trait による抽象化はもちろんできます。やむを得ず生のポインタを操作するときには unsafe ブロックにする必要はありますが、 unsafe だからといって型検査やライフタイムチェッカーが無効になることはありません。

組み込み環境における抽象化や静的検査の強み

Rust を代表する概念のひとつに、**ゼロコスト抽象化**というものがあります。オーバーヘッドやメモリ消費の増加を伴わずに高度な抽象化を提供しようというものです。

組み込み環境で特にメリットが大きいのは、 **HAL**（Hardware Abstraction Layer / ハードウェア抽象化層） に対する静的検査です。一例として、「シリアルポートと GPIO

[*3] IO や PWM などの周辺機能のこと。

[*4] 代表的なものでは整数除算命令、浮動小数点数演算命令、アトミックなロード・ストア命令等がありません。

を初期化して、 1 文字出力するごとに GPIO をトグルする」という処理を考えてみましょう。

たとえば C 言語では、 リスト 2.1 のようなミスが起きてしまうかもしれません。

▼リスト 2.1　C 言語で書くと起こるかもしれないミス

```
#define PIN_UART_RX 1
#define PIN_UART_TX 2
// Oops! Pin number is conflicting!
#define PIN_LED 2

void do_action(void) {
    uart_initialize(PIN_UART_RX, PIN_UART_TX, 9600);
    gpio_initialize(PIN_LED, GPIO_OUTPUT);

    bool led_on = false;
    while (1) {
        uart_write_char('A');
        gpio_set_output(PIN_LED, led_on);
        led_on = !led_on;
    }
    return 0;
}
```

このコードはコンパイルは通りますが、実行時に UART TX ピンと LED ピンが被ってしまっているため、正常に文字を出力することはできないでしょう。

さて、Rust には `embedded_hal` をはじめとした組み込み環境に特化したクレート[*5]が多く存在します。これらのクレートは、ムーブセマンティクスや幽霊型といったテクニックを活用して安全な操作を提供しています。では、先ほどと同じようなミスを Rust でおかして リスト 2.2 のようなコードを書いた場合はどうなるでしょうか？

▼リスト 2.2　Rust で同じようなミスは発生するのか？

```
fn do_action() -> ! {
    let uart_pins = (
        pins.gpio1.into_mode::<FunctionUart>(),
        pins.gpio2.into_mode::<FunctionUart>(),
    );
    // Oops! GPIO number is conflicting!
    let led_pin = pins.gpio2.into_push_pull_output();

    let uart = Uart::new(peripherals.UART0, uart_pins).enable();

    let mut led_on = false;
    loop {
        uart.write(b"A");
        if led_on {
            led_pin.set_high();
        } else {
            led_pin.set_low();
        }
        led_on = !led_on;
    }
}
```

結果は……「コンパイルエラー」です。具体的には、「`pins.gpio2` はすでにムーブさ

[*5] クレート (crate) は、Rust におけるライブラリパッケージの単位。

れていて led_pin で再びムーブすることはできない」というような理由でエラーになります。

pins には各ピンを表すフィールドが列挙されています。それぞれのフィールドの値は、一度何かの機能に使うために取り出したらもう一度取り出す（ムーブする）ことはできないようになっています。これにより、同じピンを別々の機能で誤って使ってしまうことを防ぐことができます。

また、into_**を呼んだあとの各ピンは、静的な型情報として「どの機能としてわれているか」と「機能が有効かどうか」といった情報が付与されます。これらの型はもちろんコード中で参照することができますが、実際には一切のデータをもたない型であるため、実行時には取り除かれて一切メモリを消費しません。この型情報のおかげで、「UART0として使うのに有効な組み合わせのピン以外が初期化関数に渡されたらコンパイルエラーにする」というようなことが可能になるのです。

2.2 漢字描画への道

Raspi Pico のファームウェアを Rust で記述できるということで何をやるか。ずばり、「**外部のディスプレイモジュールに、漢字を含む日本語文字列を表示する**」です。ここからは、この目的を達成するために実装が必要ないくつかの要素について説明していきます。

過去に行った実験との比較

この記事で紹介する内容は、2019 年に個人的に僕が実験したものを別の環境で再実装したものになっています。当時のプロジェクトでは、Sipeed Maixduino（以下 Maixduino）というマイコンボードで動作させており、ファームウェアは C++/PlatformIO で記述していました。Maixduino に関する詳細な説明はここでは省きますが、FPU や CNN アクセラレーターといった AI 処理を想定した機能や構成が特長となっています。

今回の実験で比較的関係のありそうな相違点は表 2.2 のとおりです。

▼表 2.2　RP2040 のスペック

相違点	前回 (Maixduino)	今回 (Raspi Pico)
アークテクチャ	RISC-V RV64GC	Arm Cortex-M0+
動作周波数	400MHz	最大 133MHz
SRAM	8MB (6MB + 2MB)	264KB (256KB + 8KB)
不揮発ストレージ	16MB	2MB

アーキテクチャが違うためファームウェアの動作速度等について正確に議論することは難しいですが、 SRAM のサイズは確実に開発方針に響くと言ってよいでしょう。前回・今回ともに利用するのは k8x12[*6] というフリーのフォントですが、非圧縮 1bpp のビット

[*6] https://littlelimit.net/k8x12.htm

マップで約 103KB のサイズになります。 Maixduino であればすべて SRAM に載せてもかなり余裕がありますが、 Raspi Pico ですべて載せると 4 割近く消費することになります。これは他のファームウェアに実装する際に他の機能の足かせとなる可能性があるので、前回よりも総合的なメモリ消費量の削減に焦点をあてる必要があると判断しました。

2.3 UTF-8 文字列から文字列を描画するまで

このプロジェクトの達成のためには、**コードポイントから区点コードへの変換テーブルを構築し、実行時に文字を検索・グリフを取得して描画する**というタスクをこなさなければなりません。本節では、このタスクの要素についてステップごとに分けて説明していきます。

Step 0. なぜ変換テーブルが必要か

Rust における文字列のエンコーディングは UTF-8 です。ソースコードに記述した文字列と外部から入力された文字列のいずれも、基本的にはメモリ上の表現は UTF-8 のバイト列であることを前提としています。これらの文字列からは、`chars()` メソッドを使うことで Unicode コードポイント（以下コードポイント）単位で文字列中の文字を得ることができます。

一方、フォントデータとして今回ベースに使用するのは k8x12 フォントの PNG 版です。PNG 版は、 1 枚の PNG 画像の中にそれぞれの各文字のビットマップ、いわゆるグリフが JIS X 0208 の区点コード（以下区点コード）順で並べられています。

コードポイントと区点コードの間に機械的な対応づけは残念ながら存在しません。事前にコードポイントと区点コードの対応づけを計算して変換テーブルとして保存し、ファームウェアに埋め込んで実行時にテーブルから参照しなければなりません。

Step 1. コードポイントから対応する区点コードを求める

変換テーブルを生成するにも、まずは単一のコードポイントに対応する区点コードを求める必要があります。ただし、Unicode に収録されている文字数（基本多言語面 だけで 55000 字以上）より JIS X 0208 に収録されている文字数（7000 文字弱）のほうが大幅に少ないので、効率化のために実際の処理は「区点コードから対応するコードポイントを求める」というものになります。このプロジェクトの仕様として JIS X 0208 と ASCII に含まれない文字は描画しないこととしているので、対応づけが存在する文字についてはこのペアを逆にしてテーブルに登録すれば問題ありません。

それでは早速求めていきましょう。突然ですがここで Shift_JIS に登場してもらいます。 Shift_JIS のうち 2 バイトで表される文字[*7]は、そのバイト表現を区点コードから

[*7] かなや漢字、全角英数字などのことを「2 バイト文字」ということがありますが、これは現代においては

機械的に求めることが可能です。この変換処理を Rust で実装すると リスト 2.3 のようになります（ここで、 ku と ten はそれぞれ 1-based index であることに注意してください）。

▼リスト 2.3　JIS 区点コードから Shift_JIS のバイト列を得る

```
fn kuten_to_sjis(ku: u8, ten: u8) -> [u8; 2] {
    let first = if ku >= 63 {
        (ku + 1) / 2 + 192
    } else {
        (ku + 1) / 2 + 128
    };
    let second = if (ku + 1) % 2 == 0 && ten <= 63 {
        64 + (ten - 1) + ((ku - 1) % 2 * 94)
    } else {
        64 + ten + ((ku - 1) % 2 * 94)
    };
    [first, second]
}
```

Shift_JIS から Unicode への変換は、 WHATWG Encoding Standard[*8] の実装があれば容易です。ここでは、 Mozilla によって開発されている encoding_rs クレートを利用します。

▼リスト 2.4　Shift_JIS を経由してコードポイントと区点コードの対応を得る

```
use encoding_rs::SHIFT_JIS;

let mut unicode_to_jis0208 = BTreeMap::new();
for ku in 1..=94 {
    for ten in 1..=94 {
        let sjis_bytes = kuten_to_sjis(ku, ten);
        let (s, _, malformed) = SHIFT_JIS.decode(&sjis_bytes);
        if malformed {
            continue;
        }
        let mut chars = s.chars();
        let target_char = chars.next().expect("Should have at least 1 char");
        unicode_to_jis0208.insert(target_char, (ku, ten));
    }
}
```

なお、実際には ASCII 範囲内の文字も対応する全角文字に割り当てています。こうすることで、描画時の半角・全角の違いなどを気にする必要がなくなるほか、単純に視認性が向上します。

Step 2. 変換テーブルのバイナリデータを構成する

これで少なくとも「与えられたコードポイントに対応する区点コードを求める」というタスクはできるようになりました。次に、このテーブルデータを実際のファームウェアで

不正確かつ誤解を招きやすい表現です。実態としては ASCII に含まれているかどうかが重要なことが多いので、「ASCII 範囲外文字」といった表現を個人的にはお勧めします。

[*8] https://encoding.spec.whatwg.org/

利用可能なバイナリデータとして構成する方法を説明します。

先ほど、 JIS X 0208 の収録文字数は 7000 文字弱と紹介しました。存在する区点コードだけを単純に並べたとしてもそれぞれの区点コードの表現に最低 2byte 必要なので、合計サイズは最低 30KB 程度になります。

今回の環境ではアロケーターを用意していないため、ヒープ領域に動的にサイズが変わる `HashMap` などを配置することはできません。そこで、まずは静的にサイズが決定するデータコンテナを `heapless` クレートから選定して使用することを考えました。しかし、この変換テーブル全体が入るようなサイズのバッファを確保すると約 100KB 近いサイズになってしまうことが判明しました。これでは先のフォントビットマップを全部載せるのと大差なく、 SRAM を合計で 8 割近く消費してしまいます。というわけでこの案はボツになりました。

さて、効率よくコードポイントから区点コードを検索する構造を構築したいです。テーブルの要素をただ並べただけではすべての文字を探索することになってしまい、非常に効率が悪いです。そこで、より簡易的なハッシュテーブルをできるだけ小さい追加データで実装することを考えました。概念的には「ハッシュ関数がビットシフト、衝突時の探索がチェイン方式であるハッシュテーブル」のようなものになります[*9]。

まず最初にプログラム領域に埋め込む変換テーブルのバイナリデータを構築する必要があります。その方法は次のとおりです。

1. コードポイントと区点コードの対応付けのタプルの要素数を N としておきます。
2. テーブル内の各チェーンの最大長 L を設定します（ここでは 32）。
3. 対応付けタプルはソート可能であり要素数もすでに確定しているので、コードポイント順でソートして連続になるように配置します。
4. U+0000, U+0020, U+0040... というように U+0000 から L を足していった仮のコードポイント X について、「2. で生成した配列の中で X よりコードポイントが大きいもっとも左にある要素のインデックス I_X 」を探します。これは `lower_bound` として知られる処理で、ソート済みの配列に対して行うので計算量は $\mathrm{O}(\log N)$ になります。
5. 3. を U+FFFF までのすべての範囲に対して行い、 U+0000 から順に該当するインデックスのみを配列にして並べます。
6. L 、 N 、 3. の配列、 2. の配列を連結すれば完成です。

この方法で構築したハッシューテーブルは、検索のために増加するデータ量がほぼチェーン先頭を並べたインデックスの配列の大きさに依存します。言い換えれば、長いチェーンを使うほど小さく、短いチェーンを使うほど大きくなります。このチェーンの長さは後述する検索効率に直結します。ここで構築したバイナリデータは、視覚的に表現す

[*9] map データ構造の動作原理などについては、弊社 techblog の拙著「map データ構造の列挙順序と脆弱性について調べてみた」でも紹介しています。`https://www.klab.com/jp/blog/tech/2021/20211224-map.html`

ると 図 2.1 のようになります。

<table>
<tr><td colspan="3">Chain Length
32</td><td colspan="3">Registered Characters
7454</td></tr>
<tr><td colspan="2">Chain U+0000～
Index: 0</td><td>Chain U+0020～
Index: 32</td><td colspan="2">Chain U+0040～
Index: 64</td><td>Chain U+0060～
Index: 96</td></tr>
<tr><td colspan="2">Chain U+0080～
Index: 128</td><td>Chain U+00A0～
Index: 128</td><td colspan="2">Chain U+00C0～
Index: 134</td><td>Chain U+00E0～
Index: 135</td></tr>
<tr><td colspan="6">....</td></tr>
<tr><td colspan="2">Chain U+FFE0～
Index: 7448</td><td>Chain U+FFE8～
Index: 7448</td><td colspan="2">Chain U+FFF0～
Index: 7448</td><td>Chain U+FFFE～
Index: 7448</td></tr>
<tr><td>U+0000</td><td>1 区</td><td>1 点</td><td>U+0001</td><td>1 区</td><td>1 点</td></tr>
<tr><td colspan="6">....</td></tr>
<tr><td>U+0020</td><td>1 区</td><td>1 点</td><td>U+0021</td><td>1 区</td><td>10 点</td></tr>
<tr><td>U+0022</td><td>92 区</td><td>94 点</td><td>U+0023</td><td>1 区</td><td>84 点</td></tr>
<tr><td colspan="6">....</td></tr>
</table>

▲図 2.1　変換テーブルのバイナリレイアウト。1 行あたり 8byte で表現されている

チェーンを長くすると I_X の配列のサイズは小さくできますが、探索時に辿る必要のある要素数が増えてしまいます。短くすると辿る必要のある要素数は小さくなりますが、I_X の配列のサイズが倍々で大きくなってしまいます。前回の実験でいくつかの値を試した結果 $L = 32, |I| = 2048$ がバランスが良さそうという結論になったので、今回の実験でもこの値を採用しています。最終的に、この変換テーブルのサイズは約 34KB になりました。このデータは SRAM にはコピーせず、 Flash に格納したまま利用します。

Step 3. 変換テーブルを使ってコードポイントから区点コードを求める

このバイナリデータを埋め込んだ状態で、実際にコードポイントから区点コードを検索する際には次のように処理します。

1. 対象のコードポイントを、 L の分（ここでは 5 ビット）だけ右にビットシフトした値を H とします。

2. H はそのまま構築したデータのチェーン先頭の配列のインデックスになるので、該当箇所にある I_H を読み出します。これは定数時間 O(1) で実行可能です。
3. I_H の値が N 以上である場合は変換テーブルに存在しない文字なので該当なしとして探索を終了します。
4. そうでない場合、対応付けタプル配列の I_H 番目を参照し、そこに記録されているコードポイントと比較します。
5. コードポイントが一致した場合はそのタプルに記録されている区点コードを返して探索を終了し、一致しない場合は次の要素に進みます。
6. 比較先に記録されているコードポイントのほうが大きい値になるか、チェーンを進んだ個数が L を超えたら該当なしとして探索を終了します。この処理の計算量は O(L) です。

Step 4. グリフをキャッシュする

変換テーブルに続いて、フォントビットマップへのアクセスについても考える必要があります。といっても、ビットマップそのものの構造についてではありません。区点コードからリニアな要素位置を求めることができるので、ビットマップ自体は文字を区点の順で並べるだけで目的を達成できます。

ここで問題になるのはグリフへのアクセス効率です。Flash ストレージは SRAM ほど高速にアクセスできるわけではないので、1 文字ごとにランダムアクセスでグリフを取得していては全体の動作速度が下がってしまうおそれがあります。Raspi Pico 自体はそれなりに高速な QSPI Flash を搭載している[*10]ので気にする必要はないかもしれません。しかし、たとえば大きいフォントを使った際に Flash ストレージに載せきれず、より大容量だがアクセスが遅い SPI 接続の microSD カードなどに格納することを考えると、やはり可能な限り SRAM からグリフを読み出せるようにしたいところです。

そこでフォントキャッシュ機構も作ることにしました。キャッシュ戦略は当初 LRU（Least Recently Used）を実装することを考えていましたが、変換テーブルと同様にメタデータによるメモリ消費量がなかなか抑えられそうになかったため、代わりに「キャッシュが溢れたらもっとも古い登録のデータを追い出す」というものにしました。いうなれば LRR（Least Recently Registered）です。この場合に必要なのはキャッシュ本体の配列、およびコードポイントに対応するキャッシュ配列内の位置へのハッシュテーブルです。後者のハッシュテーブルは前節で言及した `heapless` クレートのものを使っても問題ない程度の消費量でした（512 文字分のキャッシュで約 13KB）。

[*10] Winbond W25Q16JV というものを搭載しており、データシートによるとシーケンシャルリードで最大 66MB/s のレートが出せるようです。

2.4 実装

Raspi Pico で漢字描画を実現するために必要なデータやアルゴリズムについて議論してきました。ここからは、実際の Rust のコードを紹介しつつ、主要な部分の実装について解説していきます。

文字列バッファ

no_std な環境において、 write!, writeln! といったマクロに代表される文字列へのフォーマッティングは core::fmt モジュールで提供されるため、依然として使用することができます。一方で std::io モジュールは存在しないため、 std::io::Error の存在を前提とするいくつかの構造体は使用することができなくなっています。 String 構造体が使用できません。

write! 系のマクロでフォーマッティングするためにはその書き込み先として core::fmt::Write トレイトを実装している構造体が必要になります。そこで、固定長バッファと書き込み長を保持するシンプルなバッファ構造体 FormatBuffer を リスト 2.5 のように実装しました。

▼リスト 2.5　文字列をメモリ上に保持するための固定長バッファ

```
pub struct FormatBuffer<const BUFFER_SIZE: usize> {
    buffer: [u8; BUFFER_SIZE],
    written: usize,
}

impl<const BUFFER_SIZE: usize> Write for FormatBuffer<BUFFER_SIZE> {
    fn write_str(&mut self, s: &str) -> FmtResult {
        let str_bytes = s.as_bytes();
        let str_length = s.len();
        let buffer_left = BUFFER_SIZE - self.written;
        if str_length > buffer_left {
            return Err(FmtError);
        }
        let target = &mut self.buffer[(self.written)..(self.written + str_length)];
        target.copy_from_slice(str_bytes);
        self.written += str_length;
        Ok(())
    }
}
```

使用するディスプレイモジュール

今回の実験では、Adafruit 社の 128x96 16bit カラー OLED モジュール[*11]を使用します。このモジュールは OLED の制御チップとして SPI 接続で制御できる SSD1351 を採用しており、数本の制御線をつなぐだけでさまざまなマイコンから比較的簡単に制御

[*11] https://www.adafruit.com/product/1673

することができます。 Rust での制御用にちょうど同名の ssd1351 クレートがあったのでこれを利用することにしました。このクレートはホスト側に表示領域と同じサイズの VRAM 領域を持たずに制御することができるので、ここでもメモリ消費量を節約できました。

余談ですが、この記事を書き始めた時点では 128x96 サイズのモジュールに対応していなかったため、対応用のコードを数行だけ追加して PR を投げ、無事マージ・リリースされました。小さな改善でも PR は投げてみるものですね。

変換テーブルの探索

コードポイントから区点コードへの変換テーブルにおいて検索する処理のコードを リスト 2.6 に示します。

▼リスト 2.6　変換テーブルから区点コードを検索する処理

```
/// 変換テーブルを表す構造体。
pub struct Unicode2JisTable<'a> {
    chain_indices: &'a [u8],
    table_elements: &'a [u8],
    chain_length_bit: u32,
    elements_count: usize,
}

/// 与えられたコードポイントから区点コードを検索するメソッド。
pub fn query(&self, c: char) -> Option<(u8, u8)> {
    let c = c as u16;
    let chain = (c as u32 >> self.chain_length_bit) as usize;
    let chain_start = u16::from_le_bytes([
        self.chain_indices[chain * 2],
        self.chain_indices[chain * 2 + 1],
    ]) as usize;
    let chain_end = (chain_start + (1 << self.chain_length_bit)).min(self.elements_count);
    for element_index in chain_start..chain_end {
        let element = &self.table_elements[(element_index * 4)..((element_index + 1) * 4)];
        let element_char = u16::from_le_bytes([element[0], element[1]]);
        match c.cmp(&element_char) {
            Ordering::Greater => continue,
            Ordering::Equal => return Some((element[2], element[3])),
            Ordering::Less => return None,
        }
    }
    None
}
```

ビットマップデータの出力

ssd1351 クレートだけでもビットマップデータを出力することは可能ですが、ピクセル単位の設定などプリミティブな操作に限定されます。そこで使えるのが embedded-graphics クレート。これは、矩形や円、多角形といった図形や任意のビットマップの描画といった高度な内容の描画を抽象化されたインターフェースで実行可能にしてくれるものです。また、このクレートはハードウェア層に対して共通のトレイトの実装を要求しており、異なるデバイスに対してもコードを変更せずに描画させることが可能です。もちろ

ん、ssd1351 クレートも要求されているトレイトの実装を提供しています。

最初期の実装では、グリフごとに 8x12 の領域領域を矩形のビットマップとして出力していました。しかし、この方法には「文字の背景に描画されていた内容が背景色で上書きされてしまう」という重大な問題があります。うまく文字に含まれないピクセルを避けて出力できないものでしょうか。

結論からいえば可能です。そして、最終的にそのように実装することができました。その方法を説明するためには、 embedded-graphics が提供する機能について少し掘り下げる必要があります。

embedded-graphics には DrawTarget というトレイトが存在します。これは描画先デバイスが共通して実装するべきもので、具体的には次の要素を要求します。

- デバイスにおけるピクセルの色のフォーマットを表す型 Color
- 描画エラーの型 Error
- 「座標と色のペア」のイテレーターを受け取って描画するメソッド draw_iter

つまり、どうにかして draw_iter メソッドにグリフに含まれるピクセルのみを抽出するイテレーターを構築すればいいことがわかります。これを実装したのが リスト 2.7 です。

▼リスト 2.7　グリフに含まれるピクセルだけを描画する関数

```
fn draw<C: PixelColor, D: DrawTarget<Color = C>>(
    target: &mut D,
    offset: Point,
    fore_color: C,
    back_color: Option<C>,
    glyph: &Self::Cached,
) -> Result<(), D::Error> {
    let pixels = glyph
        .into_iter()
        .enumerate()
        .map(|(byte, &x)| {
            let glyph_y = byte as i32;
            // Per-byte, part of column
            (0..8).map(move |glyph_x| {
                let point = Point::new(offset.x + glyph_x, offset.y + glyph_y);
                let shifted_bit = 1 << (7 - glyph_x);
                if x & shifted_bit != 0 {
                    Some(Pixel(point, fore_color))
                } else {
                    back_color.map(|c| Pixel(point, c))
                }
            })
        })
        .flatten()
        .flatten();
    target.draw_iter(pixels)?;
    Ok(())
}
```

glyph 引数はここでは &[u8; 12] 型、つまり 1byte に 1 行分のグリフのピクセルの情報が入った配列になっています。ポイントは .map() の後に .flatten() を 2 回呼び出しているところです。 .map() が終わった段階ではこのイテレーターは 1 要素に 8 ピ

クセル分の情報を含む多次元の構造になっているため、まず 1 回目の呼び出しで平坦化することで「描画するべきものだけ Some() になっているピクセルの情報」のイテレーターが得られます。そして Rust では `Option<T>` もイテレーターとして機能するため、もう一度呼び出すことで「描画するべきピクセルだけが列挙される」イテレーターに変形されます。この状態になると先ほど説明した `DrawTarget::draw_iter()` に渡せる状態になります。

より抽象的なコードを目指して

`embedded-graphics` は同時に描画される図形に対しても `Drawable` トレイトを提供しているので、これも実装しました。実装するべきメソッドは `draw()` のみですが、このメソッドはレシーバーとして `&self` をとる関係でキャッシュの操作に `RefCell` が必要になります。また、フォント情報と描画するテキストは分離可能なので、ここまでに説明した処理は `JisFont` 構造体に実装し、`Drawable` トレイトは描画したい文字列と `JisFont` が格納される `JisText` 構造体に実装しました。

最終的には、 リスト 2.8 のように比較的シンプルなコードで文字列を描画できるようになりました。

▼リスト 2.8　JisText 構造体を使って文字列を描画するコード

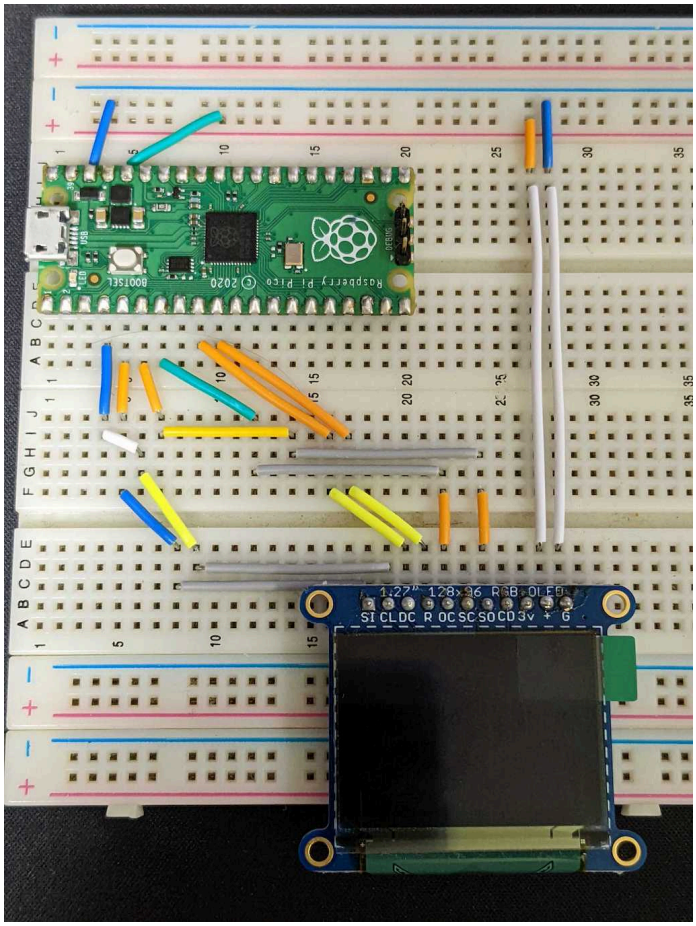
```
const FONT_K8X12_BITMAP: &[u8] = include_bytes!("../../assets/k8x12.bin");
const FONT_CACHE_SIZE: usize = 256;

let font_k8x12 = JisFont::<JisFont8x12, FONT_CACHE_SIZE>::new(&FONT_K8X12_BITMAP).unwrap();
let white_k8x12 = JisTextStyle::new(&font_k8x12, Rgb565::WHITE);

JisText::new("Hello, World!", Point::new(0, 0), &white_k8x12)
    .draw(&mut oled_display)
    .unwrap();
```

2.5　結果

動作している様子をいくつかの写真とともに紹介します。

実験回路

ブレッドボード上に 図 2.2 のような回路を組んで実験しました。Raspi Pico 側の SPI のピン配列は自由度が高くないのでディスプレイモジュール側のピン配列に合わせるために多少無理をしていますが、8MHz でデータ送信するぶんには問題ないようです。



▲図 2.2　実験用の回路

ポラーノの広場

最初に描画するのは宮沢賢治の『ポラーノの広場』の一節です。 macOS のフォント見本テキストとして採用されている文なので、この部分だけ知っているという人も多いのではないでしょうか。ディスプレイモジュールを拡大した様子を 図 2.3 に示します。

▲図 2.3　ポラーノの広場の一節とこのプロジェクトの説明文を描画している様子

テキスト背景が透過していることが分かりやすいように、いくつかの図形を背景に配置してあります。黒を含めた任意の背景色を指定することもできるように設計してあるので、用途に応じて使い分けられます。

デバッグ表示

Debug トレイトを実装した型はそのフィールドの値などの情報を文字列として表示できるようになります。このトレイトも no_std 環境で使用可能で、デフォルト実装ではアロケーターを要求しません (リスト 2.9)。

▼リスト 2.9　Debug 実装の文字列表現を描画するコード

```
let hoge = Hoge {
    foo: "Hello",
    bar: Some(42),
    baz: 3.1415926,
};
format_buffer.clear();
writeln!(format_buffer, "{hoge:?}").unwrap();
JisText::new(format_buffer.valid_str(), Point::new(0, 0), text_style)
    .with_wrapping(16)
    .draw(display)
    .unwrap();
```

このデフォルト実装による文字列表現を描画したものが 図 2.4 です。ASCII 範囲内の文字も全角文字に変換しているため、やや間延びな印象を受けるかもしれません。

▲図 2.4　Debug 実装の文字列が描画されている様子

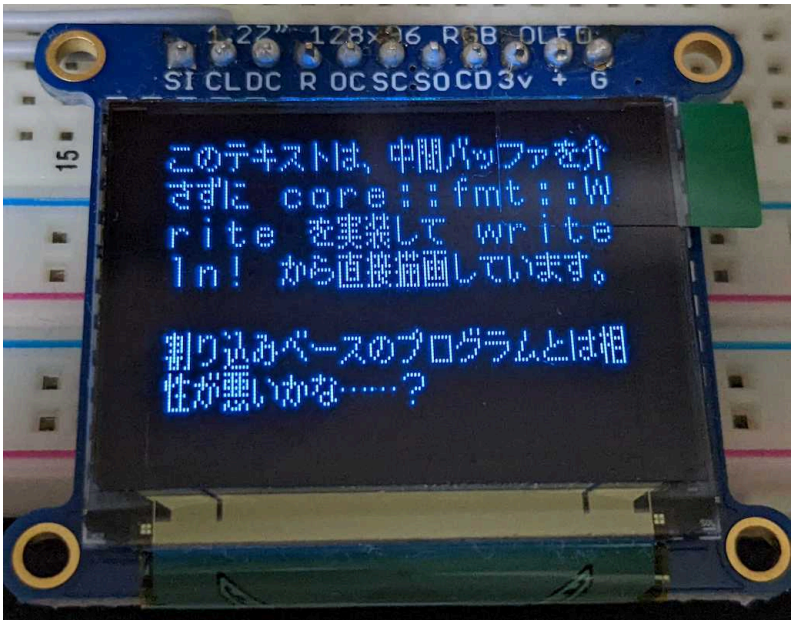
バッファレス描画

実験を進めていくうちに、Write トレイトから中間バッファを介さずに文字列を描画することができそうだということに気付きました。JisText 構造体をベースに、描画したい文字列の代わりに DrawTarget を実装する型の値と一部のステートを保持するように変更した JisTextDirect 構造体を実装します。描画処理にはほとんど変更はありません。これを使った描画コードは リスト 2.10 のようになります。

▼リスト 2.10　JisTextDirect を利用して文字列を描画するコード

```
let mut text = JisTextDirect::new(display, Point::new(0, 0), text_style).with_wrapping(16);
writeln!(
    text,
    "このテキストは、中間バッファを介さずに core::fmt::Write を実装して writeln! から直接描画してい
ます。"
)
.unwrap();
writeln!(text, "割り込みベースのプログラムとは相性が悪いかな……？ ").unwrap();
```

予想どおり、 Write トレイトから直接描画することに成功しました (図 2.5)。ただしこの場合「メインループで他のタスクと協調して一定量ずつ表示する」といった処理はできないので、バッファを使うパターンにメリットがないわけではありません。



▲図 2.5　JisTextDirect を利用したコードで描画した様子

余談

OLED は素子そのものが発光するので、 LCD にはあるバックライトが存在しません。逆にいうと、何も描画していない状態と通電していない状態に区別がつきません。前回の実験のときもそうだったのですが、想定した描画結果が得られないときの原因として SPI の制御を含む描画ルーチンが間違っているのか、配線が間違っているのか、はたまた描画領域の設定にミスがあるのかなどいくつかの分岐があって苦労しました。ただ Rust で書いたからかは定かではないですが、バグ修正の割合としては今回はよりロジックそのものの修正に時間を割くことができた気がします。

2.6　まとめ

以上、 Raspberry Pi Pico と Rust のおいしい組み合わせについて紹介・実装してきました。また、限られたリソースの中でメモリ消費量と実行速度を両立するためのニッチなテクニックについても解説しました。

コードポイントから区点コードに変換するテーブルは、より一般化して「コードポイントからフォントビットマップの位置を求めるテーブル」にすることも可能です。前回のプロジェクトで使用した Maixduino は SRAM が潤沢だったことから配布されている k8x12 の PNG 画像を直接ロードし、 PNG デコーダーを組み込んで各グリフのビット

マップを取得・描画していました。今回は PNG 画像に前処理を施して 1bpp 、すなわち 8pixel/byte になるように変換・圧縮していますから、このときに ASCII や JIS X 0208 に含まれない文字を追加したり、そもそもビットマップ自体を Unicode コードポイント順に並べることでより検索の効率を向上させられるでしょう。

この記事を執筆している 2022/08 現在 Raspberry Pi シリーズは世界的に入手困難な傾向にありますが、 Pico については比較的入手が容易なようです。これを機にマイコンプログラミングに入門してみてはいかがでしょうか？

第 3 章

USB Raw Gadget を触ってみた

Tomoaki Fude

3.1　本記事を書いた動機について

USB Raw Gadget は 2020 年上旬に linux カーネルの mainline に統合されました。自作の USB デバイス作成については configfs 経由で設定する USB gadget[*1]が有名ですが、USB raw-gadget については 2022 年 6 月にググっても公式以外の情報が少なく、今回の記事をきっかけにもっと触ってみたブログ記事等を書いてくださる方が増えるとよいなと考えています。

3.2　USB raw-gadget とは

カーネルのドキュメント[*2]にはリスト 3.1 のように書かれています。

▼リスト 3.1　公式ドキュメントから抜粋

```
USB Raw Gadget is a gadget driver that gives userspace low-level control over the gadget'
s communication process.
```

これは、ユーザー空間に低レベルなガジェットのコントロールを提供するものと見受けられます。名前からして、USB デバイスの機能を提供する既存の USB gadget に対して 'raw' を付与していることが分かります。raw には未処理といった意味合いがあります。raw-gadget では本来、デバイスドライバ・カーネルなどの低レイヤで処理されるべき USB に関する処理を行わないのです。USB に関する処理をカーネル空間で処理せず

[*1] The Linux Kernel documentation（USB support >> Linux USB gadget configured through configfs）: https://docs.kernel.org/usb/gadget_configfs.html

[*2] The Linux Kernel documentation（USB support >> USB Raw Gadget）: https://www.kernel.org/doc/html/latest/usb/raw-gadget.html

にユーザー空間で処理できるようにしたものが raw-gadget なのです。ただし、その代償としてユーザー空間での USB に関する処理をカーネル空間に伝える処理が必要となります。raw-gadget では ioctl システムコールを使って伝達します。USB Device Controller（UDC）で処理されるデータはどうしようもないですが、それ以外のデータをユーザー空間ですべて扱えるため、USB における Fuzzing に向いています。

3.3 USB の仕様（ディスクリプタについて）

多くの方が USB デバイスを作る側ではなく使う側として、普段意識せずマウスやキーボードなどの USB デバイスをお使いかと思います。USB デバイスをたとえば PC に挿したときに USB デバイスと PC 間でどういった通信が行われてデバイスが利用できるようになるのか、一部にはなりますが簡単にその流れを説明します。これから説明するディスクリプタの知識があれば Fuzzing 用のコードを書く際に各種ディスクリプタの意味を理解して書くことができるようになるかと思います。「仕組みはいいからはやく動かしたい!」という方は「サンプルコードを動かしてみる」までスキップいただいて構いません。

ディスクリプタ

USB におけるプラグアンドプレイの挙動の一部を説明することになります。マウスや USB メモリなどのデバイスを PC に挿した場合、PC は USB ホストとして振る舞い、マウスや USB メモリは USB デバイスとして振る舞います。USB デバイスを PC などの USB ホストに挿すと、次に述べる処理が内部で起こります。

Device Descriptor

USB ホストは USB デバイスに対して、「あなたはどんなデバイスですか？　教えてください。」という問い合わせである'Get Descriptor Request （DEVICE）' を投げます。そのリクエストを受け取った USB デバイスは返答として 'GET Descriptor Responce（DEVICE）' を返します。この返答には Device Descriptor という USB デバイスにおける自己紹介のようなものが含まれています。

まずは一例を見ていただきましょう。例として USB マウス（Logicool の Unifying 対応 USB マウスのレシーバ）を ubuntu マシンに挿し、lsusb コマンドで各種ディスクリプタを確認できます（リスト 3.2)。lsusb コマンドでは Device Descriptor だけでなく他の Descriptor も列挙されます。

▼リスト 3.2　lsusb コマンドで USB マウスのレシーバの各種ディスクリプタを確認

```
root@ubuntu:~# lsusb | grep Logitech
Bus 001 Device 004: ID 046d:c52b Logitech, Inc. Unifying Receiver
root@ubuntu:~# lsusb -d 046d:c52b -v

Bus 001 Device 004: ID 046d:c52b Logitech, Inc. Unifying Receiver
```

```
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        32
  idVendor           0x046d Logitech, Inc.
  idProduct          0xc52b Unifying Receiver
  bcdDevice           24.00
  iManufacturer           1 Logitech
  iProduct                2 USB Receiver
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           84
    bNumInterfaces          3
    bConfigurationValue     1
    iConfiguration          4 RQR24.00_B0018
    bmAttributes         0xa0
      (Bus Powered)
      Remote Wakeup
    MaxPower               98mA
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           1
      bInterfaceClass         3 Human Interface Device
      bInterfaceSubClass      1 Boot Interface Subclass
      bInterfaceProtocol      1 Keyboard
      iInterface              0
        HID Device Descriptor:
          bLength                 9
          bDescriptorType        33
          bcdHID               1.11
          bCountryCode            0 Not supported
          bNumDescriptors         1
          bDescriptorType        34 Report
          wDescriptorLength      59
         Report Descriptors:
           ** UNAVAILABLE **
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0008  1x 8 bytes
        bInterval               8
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        1
      bAlternateSetting       0
      bNumEndpoints           1
      bInterfaceClass         3 Human Interface Device
      bInterfaceSubClass      1 Boot Interface Subclass
      bInterfaceProtocol      2 Mouse
      iInterface              0
        HID Device Descriptor:
          bLength                 9
          bDescriptorType        33
          bcdHID               1.11
          bCountryCode            0 Not supported
          bNumDescriptors         1
```

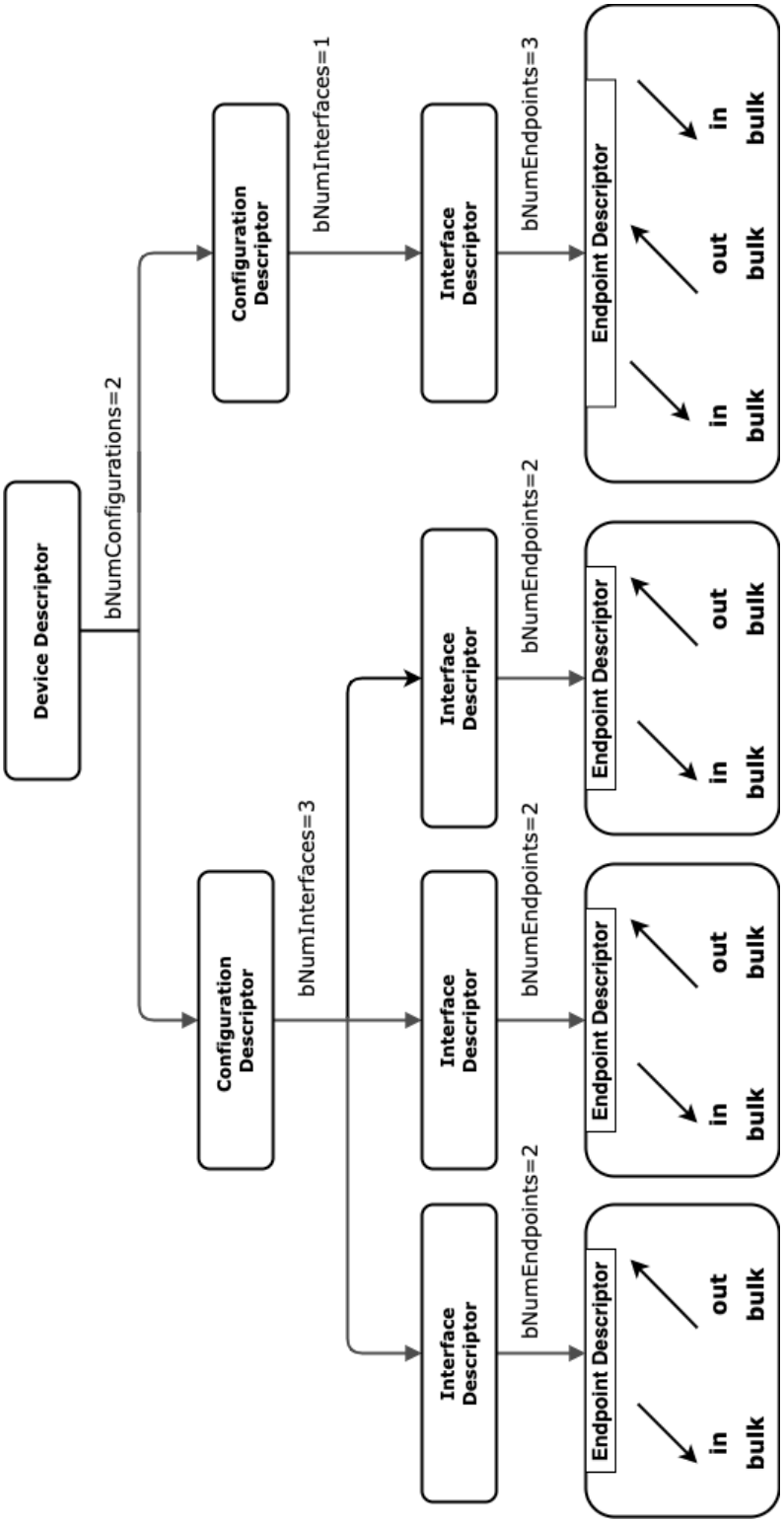
```
          bDescriptorType        34 Report
          wDescriptorLength     148
         Report Descriptors:
           ** UNAVAILABLE **
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x82  EP 2 IN
        bmAttributes            3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0008  1x 8 bytes
        bInterval               2
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        2
      bAlternateSetting       0
      bNumEndpoints           1
      bInterfaceClass         3 Human Interface Device
      bInterfaceSubClass      0 No Subclass
      bInterfaceProtocol      0 None
      iInterface              0
        HID Device Descriptor:
          bLength                 9
          bDescriptorType        33
          bcdHID                1.11
          bCountryCode            0 Not supported
          bNumDescriptors         1
          bDescriptorType        34 Report
          wDescriptorLength      98
         Report Descriptors:
           ** UNAVAILABLE **
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x83  EP 3 IN
        bmAttributes            3
          Transfer Type            Interrupt
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0020  1x 32 bytes
        bInterval               2
Device Status:     0x0000
  (Bus Powered)
root@ubuntu:~#
```

インデントで階層構造が表現されています。USB のバージョンやベンダー ID, プロダクト ID, Configuration の数などの値がこの Device Descriptor に書かれているため、これにより USB ホストはこの USB デバイスをどの USB バージョン（USB1.1 なのか USB2.0,USB3.0 なのか）で扱うか、どのドライバを使うかを判断するわけです。

USB デバイス 1 つにつき、Device Descriptor は 1 つしかもつことができません。Configuration Descriptor は複数もつことができますが、2 つ以上もつものはほとんどないようです（Device Descriptor の bNumConfigurations に何個もっているかの個数が書かれています）。

USB ホストは Device Descriptor を確認した後、Configuration ディスクリプタの問い合わせと応答を受け取り、Interface ディスクリプタを確認します。HID デバイスの場合は追加でリポートディスクリプタもしくは物理ディスクリプタを確認します。

図 3.1 にディスクリプタの階層構造イメージ図を用意しましたので参考にしてください。



▲図 3.1　USB の各種ディスクリプタの階層構造

Configuration Descriptor

このディスクリプタには、その構成が使用する電力量、セルフパワーなのかバスパワーなのか、リモートウェイクアップ機能をサポートしているか、インターフェイスの数などについて書かれています（Configuration Descriptor の bNumInterfaces にインターフェイスを何個持っているかの個数が書かれています）。少しややこしいのですが、bConfigurationValue に Configuration の番号がかかれていて、iConfiguration はストリングディスクリプタの index 番号となります。Configuration の番号は SET CONFIGURATION リクエスト実行時に指定する数字となります。USB ホストからの'GET DESCRIPTOR Request （CONFIGURATION）' のレスポンスとして USB デバイスは Configuration Descriptor だけでなく、Interface Descriptor と Endpoint Descriptor もまとめて返答します。次は Interface Descriptor についてみていきます。

Interface Descriptor

このディスクリプタは、機能ごとにグループ分けされたエンドポイント（USB ホストと USB デバイス間における通信用の pipe）の集まりと捉えればよいかと思います。エンドポイントの数もこのディスクリプタの bNumEndpoints に書かれています。

USB のクラスの種類もここで記述します（もしくは device ディスクリプタで記述します）。クラスのコードは 1byte で表されます。たとえば、0x03 だと HID、0x06 だとスキャナ、0x07 だとプリンター、0x08 だとマスストレージクラスです。多機能プリンターなどの複合機では、fax 用の Interface Descriptor, スキャナー用の Interface Descriptor, プリンター用の Interface Descriptor を記述していたりします。

実際の例として、先ほどリスト 3.2 の lsusb で確認した Logicool マウスのレシーバについて改めて見てみます。これは Unifying 対応のレシーバとなっており、ひとつのレシーバでマウスとキーボードを無線で繋げることができるものです。ディスクリプタを見るとインターフェイスディスクリプタが三つあり、1 つ目が Keyboard 用、2 つ目が Mouse 用です。最後の 3 つ目が bInterfaceProtocol が None と書かれており用途はよくわからないのですが、おそらくタッチパッドかと思われます（Unifying のサイトを見ているとタッチパッドも接続できることがわかります）。

複合機などの Interface が複数ある複合デバイス（composite device）の場合は bInterfaceNumber の番号が被らないように振られています。bAlternateSetting では代替設定の番号が書かれており Endpoint の設定を変えて利用することができるようになっています。USB ホストは USB デバイスに対して、SET INTERFACE リクエストで代替設定を切り替えます。SET INTERFACE を実施すると、それまで通信していたエンドポイントのデータトグルは DATA0 にリセットされシーケンス番号が 0 になります[*3]。

[*3] USB Complete Fifth Edition | Jan Axelson, Lakeview Research （2015/3/1 発売） | 5 Control Transfers: Structured Requests for Critical Data > Standard requests > Set Interface を参照

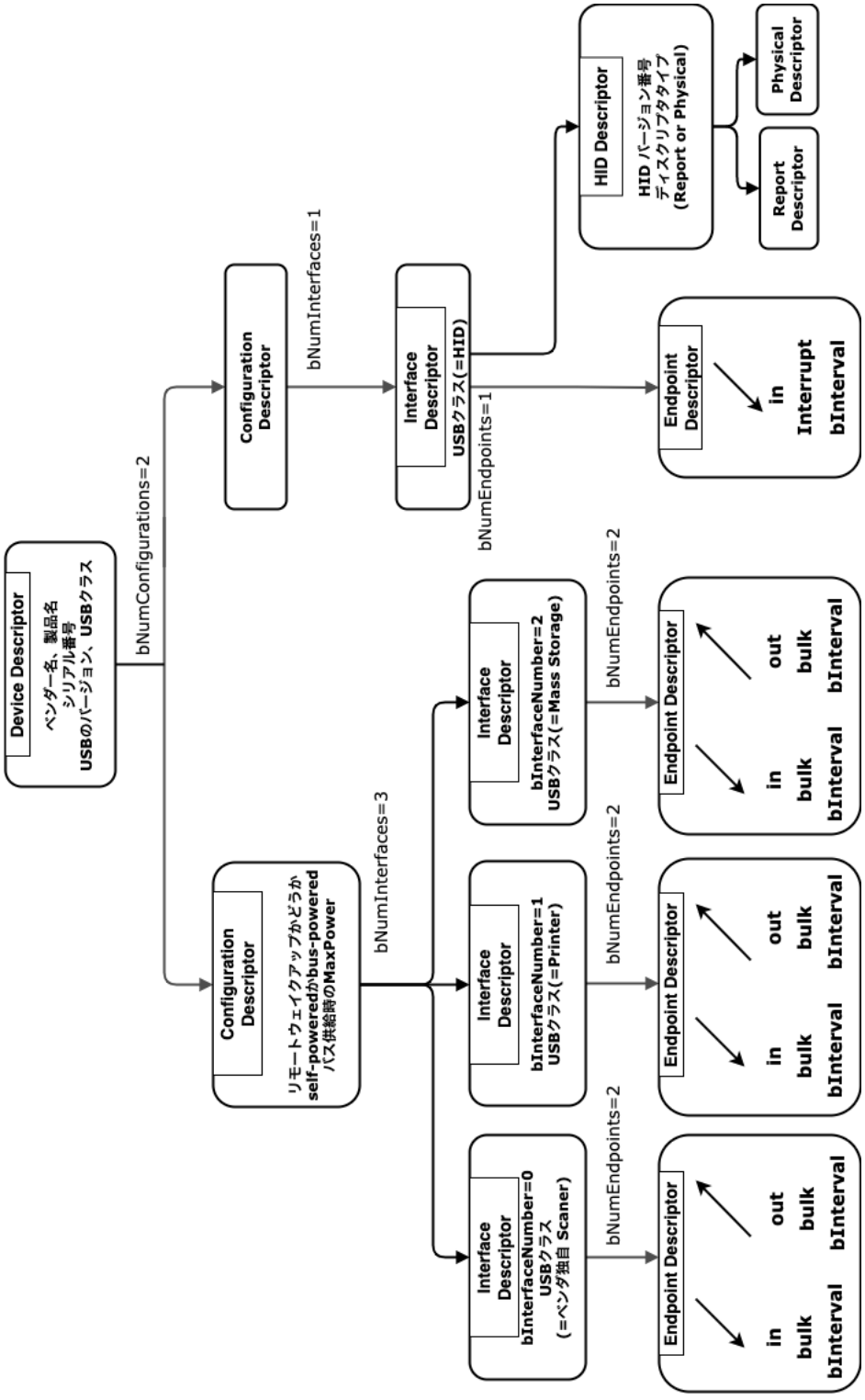
Endpoint Descriptor

USB ホストと USB デバイス間でデータのやり取りを行う通信のパイプ（単方向）の構成を記述します。バルク転送なのかアイソクロナス転送なのか、転送方向は in/out どちらなのか、データの送信間隔などが書かれています。

上記のディスクリプタをやり取りした後、USB ホストから USB デバイスに対して SET CONFIGURATION コマンドを送信して利用する Configuration 番号を選択し、USB デバイス側もその Configuration 番号で動作するように設定された状態になります。その後、USB ホストから USB デバイスに対して SET INTERFACE コマンドを送信して、利用するインターフェイスの代替設定を選択します。

ちなみに HID デバイスの場合は、インターフェイスディスクリプタに追加で HID ディスクリプタが含まれます。HID ディスクリプタの説明は省略します。

図 3.2 の左半分に複合プリンターの Configuration ディスクリプタの例を、右半分にマウスなどの HID デバイスの例を作成しました。（図のような Configuration ディスクリプタをもつデバイスは存在しないと思うので、あくまで説明のために作成したものということでご了承ください。）



▲図 3.2　USB の各種ディスクリプタの階層構造（詳細版）

3.4 コントロール転送、バルク転送、インタラプト転送、アイソクロナス転送

4 つのデータ転送方式があります。後ほどそれぞれについて説明していきますが、まずは概要の説明をします。

すべての USB デバイスには、USB ホストと USB デバイス間でコントロール用の通信のパイプである endpoint0 と呼ばれるものを持ちます。このパイプは双方向通信が可能です。デフォルトパイプとも呼ばれます。USB デバイスが USB ホストにコネクタで物理的に接続された直後の状態では、なにもディスクリプタの情報はやり取りされておらず、USB ホストと USB デバイス間で通信に使われるパイプは endpoint0 のみ存在します。この endpoint0 をつかって USB ホストと USB デバイスは各種ディスクリプタのやり取りを行い、その後で USB ホストと USB デバイス間でデータをやり取りする複数の通信用のパイプが新たに作成されます。この新たに作られたパイプではバルク転送、インタラプト転送、アイソクロナス転送といった転送方式でデータのやり取りが行われます。これらのパイプは単方向であり、USB ホストから USB デバイスへの方向かその逆かで通信の方向が固定で決まっています。方向については in 方向、out 方向と言ったりするのですが、どっちの方向への通信なのか混乱しやすいです。(ルータの ACL 設定で in/out がそれぞれどっち方向のフィルタ設定なのか混乱しやすいのと同じかと思います。ルータの場合は自分がルータの立場になって考えると in,out がどちら方向なのかが分かりやすいです)。USB は USB ホストを中心としたシステムであり、USB ホストの立場になって考えると分かりやすいです。in 方向の endpoint がどちらの方向なのか考えてみましょう。USB ホストから見て in 方向なので、USB デバイス（外）から USB ホスト（自分）への方向だと分かります。out 方向は自分（USB ホスト）から外へ出ていくので、USB ホストから USB デバイスへの通信の方向だと分かります。

コントロール転送

コントロール転送は、endpoint0 を使って行われる転送方式です。標準リクエストのひとつである GET DESCRIPTOR でのディスクリプタ情報のやり取りや、GET_STATUS、SET_ADDRESS、SET_INTERFACE コマンドを行う際に使用します。コントロール転送では 3 つのステージに分かれており、セットアップステージ、データステージ、ステータスステージがありますが本書では説明を割愛します。日本語で詳しく説明をしているドキュメントが microsoft のページ[*4]にあり、こちらを参照するの

[*4] USB コントロール転送の送信方法 | Windows ハードウェア開発者向けドキュメント（2022-08-10 参照）：https://docs.microsoft.com/ja-jp/windows-hardware/drivers/usbcon/usb-control-transfer

もよいかと思います。英語にはなりますが USB Complete Fifth Edition[*5]という本にも詳しく書かれています。

USB raw-gadget を使えば、当然ながら endpoint ディスクリプタも自分で記述できるため、USB ホストから Configuration Descriptor の要求時にレスポンスで、たとえば in のエンドポイントを 2 つ、out を 1 つにするだとか、どの転送方式を採用したいか、bInterbal を何 ms にしたいかを endpoint ディスクリプタとしてユーザー空間で自由に設定できます。

バルク転送

バルク転送は、まとまった大きなデータを転送するのに適しています。たとえば、動画ファイルなど大きなデータを USB メモリに転送する際に使用されます。

インタラプト転送

比較的少量のデータ転送を任意のタイミングで送りたい時に用いられる転送方式で、たとえばマウスを動かしたときの USB デバイスから USB ホストへのデータ転送などに使われます。

アイソクロナス転送

アイソクロナス転送は、一定の時間に一定量のデータを送る予約をしてデータ転送をする転送方式です。データ転送時の誤りなどは訂正されることはありません。そのため、データが欠けてもよいような利用用途に向いています。動画や音声の再生に向いています。

3.5 サンプルコードを動かしてみる

USB raw-gadget にはサンプルコード[*6]があります。keyboard.c と printer.c がありますが、keyboard.c はおよそ 1 秒間隔で x キーを送信するようです。動作確認に用いた環境は Raspberry Pi Zero2W で OS は Raspberry Pi OS Lite 32-bit です。まずはじめに dwc2 の有効化を行い、UDC の名前を確認します（リスト 3.2)。以降の手順では root ユーザーでセットアップ作業を行っています。

[*5] USB Complete Fifth Edition | Jan Axelson, Lakeview Research （2015/3/1 発売） | 5 Control Transfers: Structured Requests for Critical Data > Standard requests > Set Interface を参照

[*6] xairy/raw-gadget | Github：`https://github.com/xairy/raw-gadget/tree/master/examples`

▼リスト 3.3　dwc2 の有効化と UDC の名前を確認

```
echo 'dtoverlay=dwc2' >> /boot/config.txt
reboot

root@raspberrypi:~# ls /sys/class/udc
3f980000.usb
root@raspberrypi:~#
```

Zero2 の場合は UDC が 3f980000.usb なようです。ちなみに Zero は 20980000.usb になります。

次に、カーネルモジュールのビルドとロードを行います（リスト 3.4)。ラズパイにおけるカーネルヘッダの取得については Raspberry Pi Documentation の Kernel Headers セクションを参考にするとよいでしょう[*7]。

▼リスト 3.4　コンパイル

```
apt update
apt install raspberrypi-kernel-headers
apt install git
git clone https://github.com/xairy/raw-gadget
cd raw-gadget/raw_gadget/
make
./insmod.sh
```

raw-gadget の make 時にリスト 3.5 のようなエラーが出ることがあります。

▼リスト 3.5　make 時のエラー例

```
make[1]: *** /lib/modules/5.15.32-v7+/build: No such file or directory.  Stop.
make: *** [Makefile:7: default] Error 2
}
```

/lib/modules/ディレクトリ配下をみると分かるのですが、‘5.15.32-v7+‘ のカーネルリリースバージョンにマッチするフォルダ名がないため 'No such file or directory' で make に失敗していることが分かります。公式ドキュメントにも書かれているのですが最新の Raspberry Pi OS を利用していた場合、パッケージで取得したカーネルヘッダとラズパイで実際に稼働している OS のカーネルリリースバージョンが合わないことがあります。これは数週間遅れでパッケージの方に取り込まれるからです。make 時に '/lib/modules/$(uname -r)/build/' ディレクトリが無くてビルドできない場合は、Raspberry Pi OS のバージョンを 2 週間程度過去のもので microSD カードに焼き直してもう一度試すか、力技ですが公式ドキュメントの手順どおりにカーネルをビルドして差し替えて起動するとバージョンがマッチするようになるので make できるようになります。ただ、これにはかなり時間がかかり、執筆者はビルドに 3 時間はかかっていました。時間

[*7] Kernel Headers | Raspberry Pi Documentation：https://www.raspberrypi.com/documentation/computers/linux_kernel.html#kernel-headers

がかかるだけではなく Raspberry Pi Zero2W の発熱がやばく、手で持てないくらいの熱さになっていました。周囲に溶けやすいものがあると大変です。他によい方法がないか調べたところ、rpi-source[*8]というものがあることが分かりました。これを使えば、現在実行しているカーネルのソースコードをインストールし、リリースバージョンも合わせた '/lib/modules/$(uname -r)/build/' に正しい symlink が張られます。手順をリスト 3.6 にて記載していますが、make 時に当該ディレクトリがないエラーが発生していない方は実行しなくて大丈夫です。

▼リスト 3.6　rpi-souce によるカーネルヘッダ取得

```
apt install bc bison flex libssl-dev libncurses5-dev
wget https://raw.githubusercontent.com/notro/rpi-source/master/rpi-source \
  -O  /usr/bin/rpi-source
chmod +x /usr/bin/rpi-source
/usr/bin/rpi-source -q --tag-update
rpi-source
```

もし、rpi-source 実行時にエラーが発生した場合はおそらく、本来 python2 系で実行するところで python3 を呼び出してしまいエラーになっているかと思います。なお、この解決方法は 2022 年 8 月現在で遭遇したエラーです。今後は rpi-source の更新が進むにつれて別のエラーになるかもしれませんので、適宜エラー内容を確認してご対応ください。リスト 3.7 の手順を実行して、python2 系を呼ぶようにしてください

▼リスト 3.7　rpi-source で python2 を使用するように修正

```
apt install python2
vi /usr/bin/rpi-source   # 一行目にある python の文字列を python2 に書き換えてください
```

raw-gadget のカーネルモジュールのビルドとロードができた方は、サンプルプログラムのビルドを行いましょう（リスト 3.8)。

▼リスト 3.8　サンプルプログラムのビルド

```
cd ../examples/
make
```

ビルドができたらサンプルプログラムの keyboard を実行してみましょう（リスト 3.9)。

▼リスト 3.9　サンプルプログラムの実行例

```
./keyboard 3f980000.usb 3f980000.usb
```

これでキーボードとして振る舞う USB デバイスが稼働します。といってもこのサンプルプログラムの場合は本当に単純な動きしか組まれておらず、約 1 秒間隔で'x' キーを送

[*8] RPi-Distro/rpi-source | Github：https://github.com/RPi-Distro/rpi-source

信し続けるだけのプログラムです。試しに、テキストエディタを開いた状態にして、キー入力を受け付ける状態にし、ラズパイの OTG 用 USB ポートと PC をケーブルで繋げてみましょう。すると、図 3.3 のように'x' キーが入力されていきます。

▲図 3.3　x キーが 1 秒間隔で入力されていく様子

いかがでしょうか？ 送信するリポートディスクリプタに手を加えれば、x キー以外の入力もできるようになるでしょう。もしうまく動作しない場合は、PC とラズパイのケーブル接続をしたままプログラムを起動してみてください。

printer.c のサンプルプログラムについては動作確認を実施していないので、興味がある方は確認してみてください！

3.6　USB のセキュリティ面

前述したように、USB デバイスを USB ホストに挿してから USB デバイスが使えるようになるまで各種ディスクリプタのやりとりが行われます。しかし、USB デバイスと USB ホスト間に悪意のあるデバイスがあればどうでしょうか？ USB パケットを盗聴するデバイスがいればどうでしょうか？ たとえば、正規品とは別の、悪意のある偽物の USB デバイスが USB ホストに対して、正規の USB デバイスとそっくりあるいはまったく同じ応答を返すものが作られたとすればそれは USB ホストからは見抜くことができません。防ぎ方についてパッと思いつくのは、USB ホストと USB デバイスとで認証用の証明書を保持しておき、デバイスの利用時に証明書のチェックをしてあげる等の対応が有効なのではないでしょうか。証明書もコピーされてしまうと防げないかと思うのでコピーされないような仕組みも必要そうです。軽く調べてみると、そういった防ぎ方を採用されている製品も存在するようです。

USB ポートにおける Fuzzing ツールとして syzkaller があります。これを使って多く

のバグが発見され修正が行われました。[*9] メモリアクセス違反のエラーで OS がクラッシュすることもあったようです[*10]。つまり、USB ポートから（命令ポインタの書き換え等で）任意の命令実行が起こり得るはずです。USB ポートはシステムに侵入される可能性があるセキュリティホールとなり得るのです。

たとえばある製品の USB デバイスの挙動をそっくりそのまま再現するツールが作れたりしないか、あるいは微妙に挙動を変えた時に USB ホスト側が正常に処理をハンドルできるか考えてみましょう。ターゲットとなる製品の挙動を模倣するためにその製品の USB パケットキャプチャを実施して、それを USB raw-gadget で再現してあげれば良さそうです。

USB のパケットキャプチャ

USB デバイスのパケットキャプチャをする方法はいくつかあります。PC に wireshark と USBPcap をインストールし、その PC に USB デバイスを挿して、wireshark で USB パケットのキャプチャを行います。この方法は当然ながら、wireshark が稼働している PC 上でしかパケットキャプチャが実施できません。wireshark がインストールできない USB ホスト上でのパケットキャプチャが実施できないのです。

wireshark がインストールされていない環境ではどのようにパケットキャプチャすればよいでしょうか？ TotalPhase という会社が出している「Beagle USB 480 Protocol Analyzer」という製品がありますが、価格は$1,295 ドルなので個人では少々高すぎて手が出ません。他のメーカー・製品では Ellisys、LeCroy USB プロトコル アナライザーなどがありますが 20 万円以上したりと個人で買うにはかなり高価です。

それに対して、matlo 氏が作成した beableboard で稼働する USB sniffer ツールがあります[*11]。これを使えば wireshark がなくても USB ホストと USB デバイスの間に beagleboard-xM を挟むことでパケットキャプチャが行えます。beagleboard-xM は 2 万円程度で購入できるため比較的お手軽です。ただ、このツールにも問題があるようで、USB デバイスによってはパケットキャプチャが実施できないものがあります。

一方で、USB raw-gadget を使って作成された USB proxy ツールがあります[*12]。これを使えば、matlo 氏のツールでキャプチャできなかった USB デバイスも USB パケットの proxy を実施できることが確認できました。ということはこの proxy ツールを改修すれば、パケットキャプチャもできるのではないでしょうか。しかし、改修となると大変そうです。他にもっと良い方法があるとよいのですが。

改修作業を後回しにして日々を過ごしていると、なんと、もっと便利な方法があると

[*9] グーグルのファズツール、Linux カーネルの USB サブシステムに潜む複数の脆弱性発見（2017-11-09）| ZDNet Japan：https://japan.zdnet.com/article/35110109/

[*10] BSidesMunich2022 での Andrey Konovalov 氏によるセッションで OS がハングするデモが行われました)： https://twitter.com/andreyknvl/status/1532399313699147778

[*11] matlo/bb_usb_sniffer | Github：https://github.com/matlo/bb_usb_sniffer

[*12] AristoChen/usb-proxy | Github：https://github.com/AristoChen/usb-proxy

同僚に教えてもらいました。usbmon というモジュールを使えばこの proxy ツールでも tcpdump で USB パケットがキャプチャできるというのです。詳細につきましては「ラズパイと usb-proxy を使った USB プロトコルアナライザ （Qiita）」[*13]という記事に方法がまとめられていますので、興味のある方はご参照ください。なお、これらのパケットキャプチャツールでもキャプチャできなかった場合の最終手段としてロジックアナライザを使用するという手もあると思います。デジタル信号を USB パケットとして自動で解析して GUI で表示してくれる製品もあるようです。

3.7 おわりに

本記事で USB の仕様についてディスクリプタをメインに解説を行い、USB raw-gadget のサンプルプログラムを動かすところまで紹介しました。ぜひ読者の皆様に raw-gadget を使ってさまざまな自作の USB デバイスを作っていただき、その知見をインターネット上でシェアいただけると大変ありがたいです。自作の USB デバイスを作る際は、USB raw-gadget を使わずとも configfs 経由で設定する USB gadget でこと足りることがほとんどかと思われるため、Fuzzing やそれ以外の用途がメインとなってくると思います。

正直なところ、筆者自身が USB の仕様について学び始めたばかりでディスクリプタについて学ぶのがメインになってしまい、USB raw-gadget ならではの機能に触れられていないなと感じています。「USB にめちゃめちゃ詳しいよ！」という方がいらっしゃいましたらぜひ USB raw-gadget をつかったコードを公開していっていただけると泣いて喜びます！ USB raw-gadget を作られた方も、Android デバイスが USB 経由でログインできるとしたら？ という問いを投げかけています。それが実現できるポテンシャルは十分にあるのではないかと筆者は考えています。

今回、USB の仕様について執筆者がはじめに学ぶのに使用した書籍が「電子制御のための PIC 応用ガイドブック」[*14]という書籍でして、こちらに基本的なことが書かれており大変参考になりました。また、ディスクリプタの階層構造のイメージ図を作るにあたり BeyondLogic さまのサイト[*15]を参考に作図を行いました。こちらのサイトでは bNumConfigurations や bNumInterfaces、bNumEndpoints の値が分岐の個数と対応していることが図でとても分かりやすく描かれており、理解の助けとなりました。

意外にもこの描き方で説明されているところは日本語ではあまり無いようでして、ぜひこの描き方を流行らせたく、本稿でもこの描き方を採用しました。この場を借りて御礼申し上げます。

[*13] ラズパイと usb-proxy を使った USB プロトコルアナライザ | Qiita：https://qiita.com/msawahara/items/0fe982c1bf34125568ff

[*14] 電子制御のための PIC 応用ガイドブック | 後閑 哲也, 技術評論社（2002/05 発売）

[*15] USB Descriptors | Beyond Logic （2022-08-10 参照）：https://www.beyondlogic.org/usbnutshell/usb5.shtml

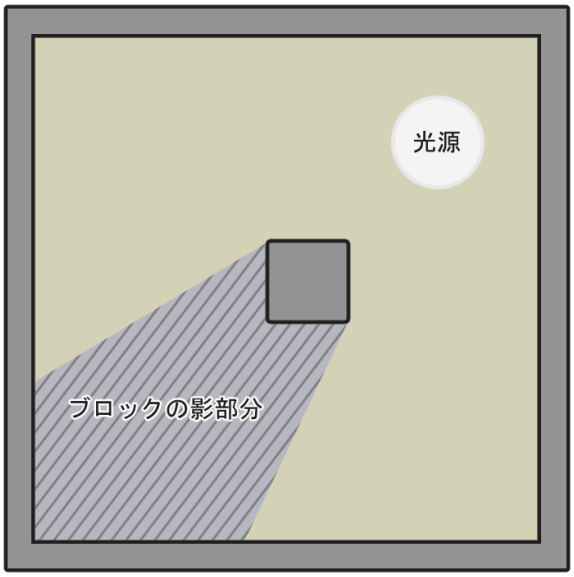
第 4 章

2D ボリュームライト用メッシュの作り方

Togo Kosaka

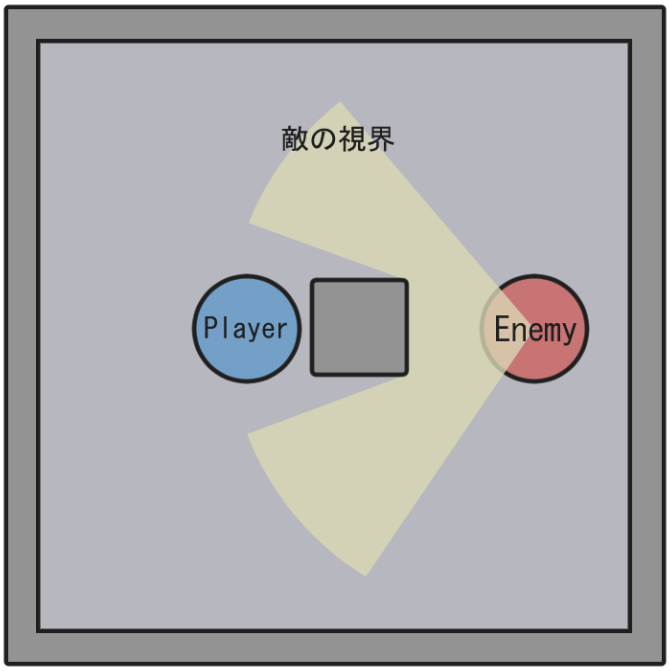
2D ゲームにおける、壁で阻まれて影[*1]ができるタイプの光源[*2]の作り方を紹介します。単に光源の表現としても使用できる他、ステルス系のゲーム等ではよく 図 4.2 のように敵キャラクターの視界を表現するためにも使用されます。技術自体はそこまで複雑ではないので、実装の概略を知る一助として頂ければ幸いです。

コード内容の詳細については「4.5 後書きとサンプル」にてサンプルコードのリポジトリの URL を記載しておりますのでそちらをご確認下さい。



▲図 4.1　成果物イメージ（光源）

[*1] 今回は影それ自体の描画は行わずに、光が当たらないため結果的に影になるものとします
[*2] 正確な意味での 2D ライティングでは無く、簡易的に加算合成による発光表現のみ扱います



▲図 4.2　成果物イメージ（ステルスゲーム）

4.1　2D ボリュームライトの作り方の概要

おおまかに表すと、次のような流れに従います。

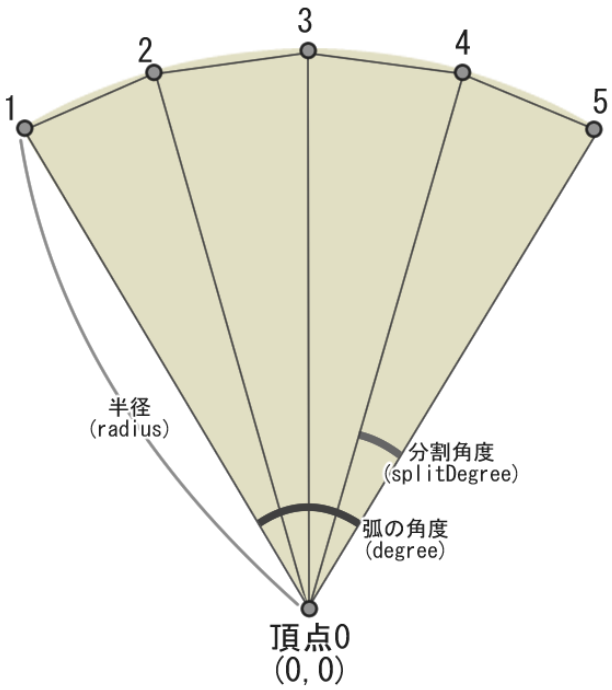
1. 光源として表示する円や弧の形のメッシュを用意する
2. 光源の周辺に存在するコライダー頂点をまとめて取得
3. 1 のメッシュ頂点 ＋ 2 のコライダー頂点（＋ 必要な場合はその他）に対してレイキャストし、遮蔽物や円弧との衝突位置を頂点として追加
4. 頂点を並べなおし、UV やインデックスを適切に入力してメッシュを生成

周囲のコライダー頂点が多いほど沢山のレイキャストを行うため、高負荷になります。

4.2　円形メッシュの動的生成

円（または弧）のメッシュを作成して表示するためにはまず、次の手順を踏みます。

1. 空のゲームオブジェクトに **MeshRendere** と **MeshFilter** をアタッチする
2. メッシュ生成を行うスクリプトを作成してアタッチ
3. スクリプトへ **弧の角度**, **半径**, **分割角度** の情報を与え、円形に頂点座標情報を生成
4. 座標と半径から UV を計算して生成
5. 三角ポリゴンを順に繋ぐインデックスを生成
6. 座標, UV, インデックス の各情報からメッシュを生成し、MeshFilter を通じて表示に反映



▲図 4.3　メッシュ作成イメージ

このうち、特に重要な 3,4,5 についてコードで表すと、大まかに次のようになります。

▼リスト 4.1　ベースとなるメッシュの頂点座標を計算 (3)

```
 1: var degree = [任意の角度の入力]; // 弧の開く角度
 2: var splitDegree = [任意の分割角度の入力] // 弧を分割する角度
 3: var radius = [任意の半径の入力] // メッシュの半径, 大きさ
 4: var startDirection = Quaternion.Euler(0f, 0f, degree / 2f) * transform.right;
 5: var currentDegree = 0;
 6: var baseVerticies = new List<Vector3>();
 7: while (true)
 8: {
 9:     // 刻む角度分ずつ向きを回転させ、メッシュ半径分先の頂点位置を計算
10:     var direction = Quaternion.Euler(0f, 0f, -currentDegree) * startDirection;
11:     baseVerticies.Add(direction * radius);
12:
13:     // 最終角度の計算が済んだら終了
14:     if (currentDegree >= degree)
15:         break;
16:
17:     currentDegree = Mathf.Min(currentDegree + splitDegree, degree);
18: }
```

▼リスト 4.2　UV を計算 (4)

```
1: var allUvs = new List<Vector2>();
2: foreach (var vertex in baseVerticies)
3: {
4:     allUvs.Add((vertex / radius) * 0.5f + new Vector2(0.5f, 0.5f));
5: }
```

▼リスト 4.3　インデックスを計算 (5)

```
1: var indexes = new List<int>();
2: for (int i = 0; i < baseVerticies.Count - 1; i++)
3: {
4:     // 0 番目に原点 (0, 0, 0) が配置される
5:     indexes.Add(0);
6:     indexes.Add(i + 1);
7:     indexes.Add(i + 2);
8: }
```

このようにすると描画に必要なメッシュの情報を生成することができます。ちなみに、transform.right を 2D 上の正面[*3]として扱っています。

これらの情報を Instanciate した Mesh.Mesh クラスへ渡すことになりますが、座標と UV のリストに原点のための情報を追加しておく[*4]必要がある点に注意して下さい。

後は生成したメッシュを MeshFilter.sharedMesh へ渡せばメッシュが反映されますが、前回分のメッシュ情報が残っている場合は Destroy しておかないとメモリリークが発生するのでここにも注意する必要があります。

[*3] 回転角度 0 の際の向き

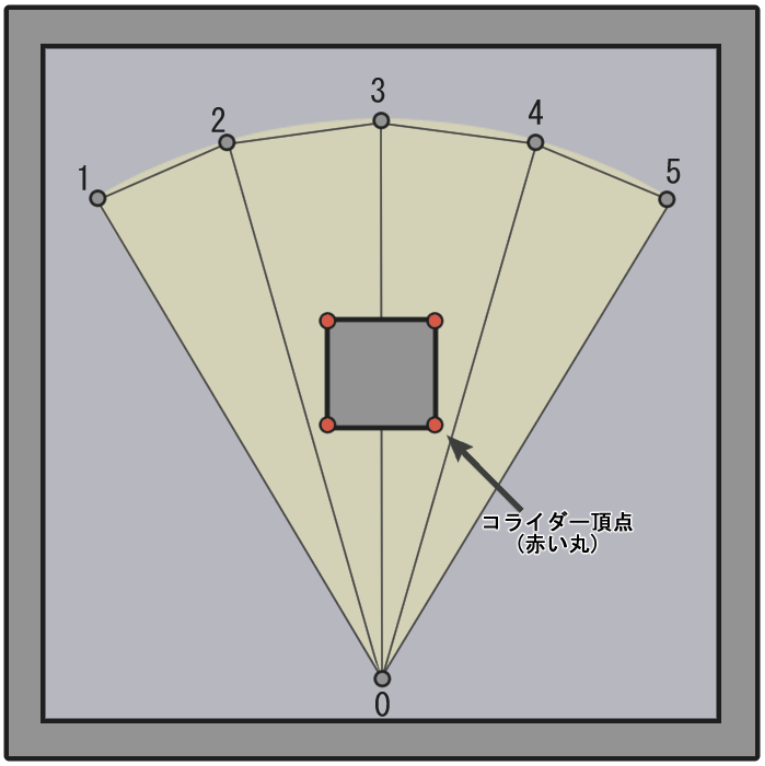
[*4] List を使用しているので処理効率を考えると原点情報は最後尾に追加する方がよいですが、本章では分かりやすさを重視して 0 番目に挿入しています。実装の際に効率化を図る場合は内容に応じてインデックス計算等を修正して下さい。

4.3　2D ボリュームライトメッシュを生成する

ここからは実際にボリュームライト用のメッシュを生成する手順を図やサンプルコードを交えて紹介します。

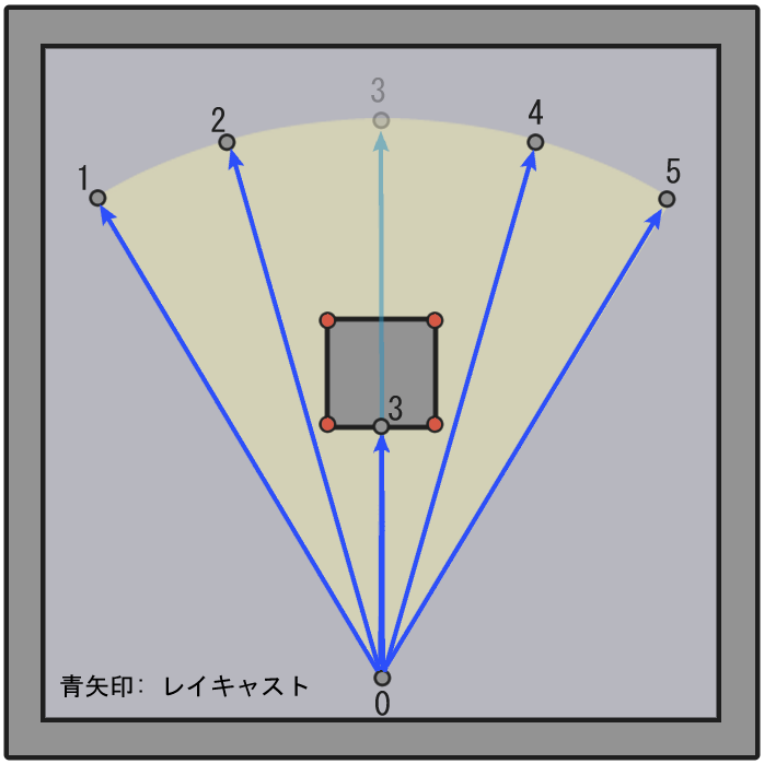
基本のメッシュ頂点へ向けてレイキャスト

ステージに先ほど生成したメッシュと障害物のブロックがあるとして、初めは次のような状態になっているかと思います。



▲図 4.4　ボリュームライト生成その 1

現状では 3 番頂点が明らかにブロックを突き抜けてしまっています。そこで原点位置から各頂点へ向けてレイキャストを行い、頂点位置の更新をします。



▲図 4.5　ボリュームライト生成その 2

▼リスト 4.4　ベースの頂点へ向けてレイキャスト

```
 1: var zeroPosition = new Vector3(transform.position.x, transform.position.y, 0f); // ワー
ルド座標上の原点座標
 2: var allVerticies = new List<Vector3>();
 3: var inverseAngle = Mathf.Atan2(transform.right.y, transform.right.x) * Mathf.Rad2Deg; // メ
ッシュを右向きに戻すための角度
 4: foreach (var target in baseVerticies)
 5: {
 6:     var ray = (target - zeroPosition).normalized;
 7:     var result = Physics2D.Raycast(zeroPosition, ray, radius, layerMask);
 8:     allVerticies.Add(Quaternion.Euler(0f, 0f, -inverseAngle) * (ray * (result.collider ==
null ? radius : result.distance)));
 9: }
```

上図では各頂点へのレイキャスト結果に応じて、壁に衝突していた場合はその位置まで頂点を移動させています。注意点としては、先ほどメッシュを生成した際には baseVerticies にはローカル座標を入力していましたが、今回はレイキャストを正常に動作させるためにワールド座標系で入力されている点です。つまり、すべての座標に zeroPosition が加算されている状態と考えて問題ありません。

なお、inverseAngle へのコメントに記載のあるとおり、頂点格納時に常にメッシュが

右向きになるように調整をしています。これを行わず、たとえば 90 度回転した上向きのメッシュが生成された場合、最終的にさらに Transform による回転が行われるため 2 倍回転するような状態となってしまい、意図どおりの結果ではなくなってしまいます。

周辺のコライダーのメッシュ頂点へ向けてレイキャスト

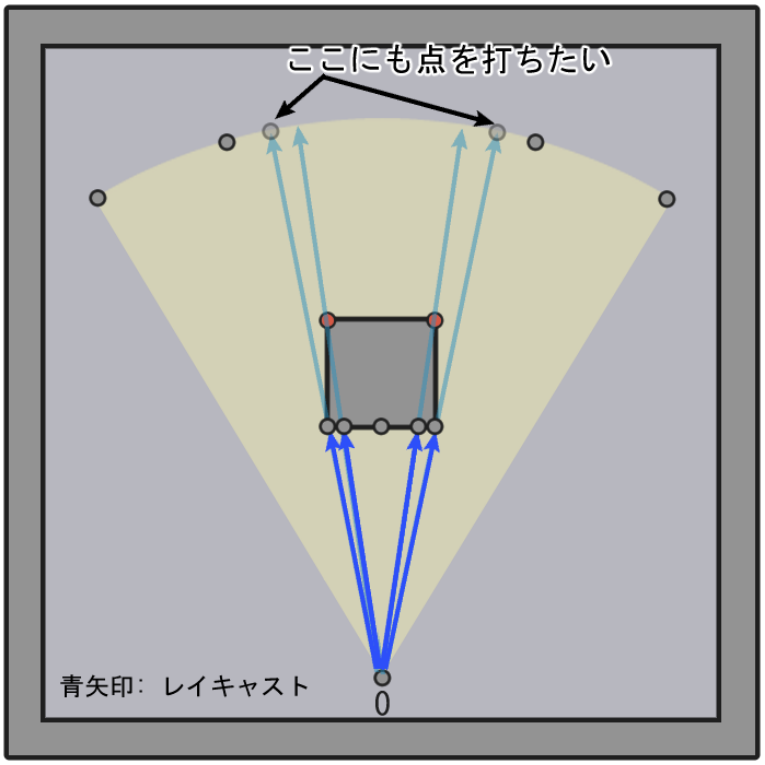
これで 3 番頂点は更新できましたが、ボリュームライトとしてはまだ不十分な状態なので、コライダー頂点に対してさらにレイキャストを行います。コライダー頂点は今回は簡易的にボックスコライダーのみの対応として、つぎのように取得します。

なお、余計な頂点を生成しないためにもボリュームライトの範囲外となる頂点は省いています。

▼リスト 4.5　ボックスコライダーのワールド頂点座標を取得

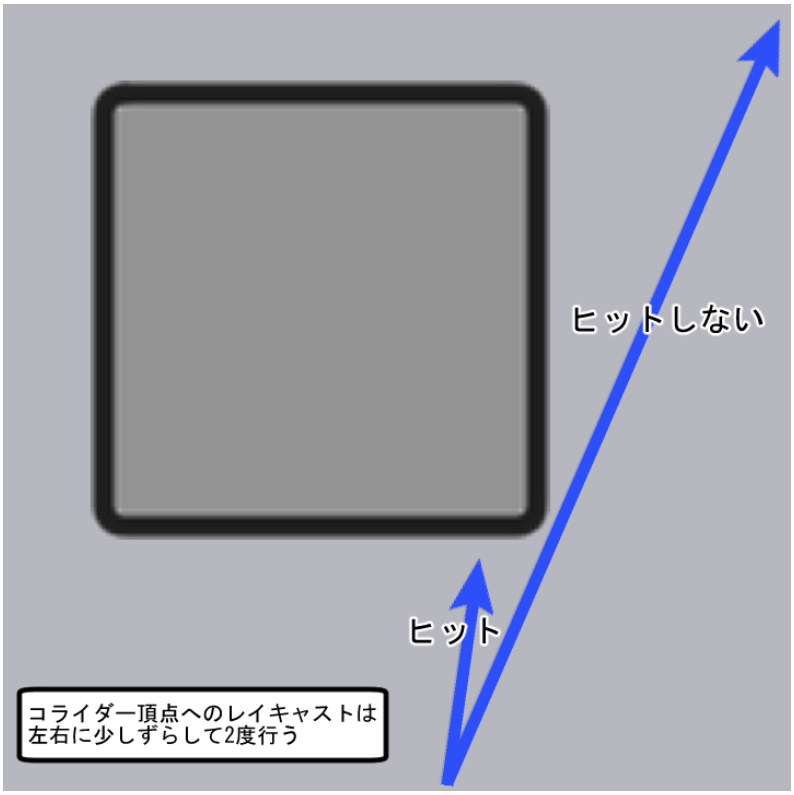
```
 1: List<Vector3> GetWorldVerticiesFromBoxCollider(BoxCollider2D boxCollider, float radius,
float degree)
 2: {
 3:     var verticies = new List<Vector3>(4);
 4:
 5:     verticies.Add(new Vector3(-boxCollider.size.x, boxCollider.size.y, 0f) * 0.5f);
 6:     verticies.Add(new Vector3(boxCollider.size.x, boxCollider.size.y, 0f) * 0.5f);
 7:     verticies.Add(new Vector3(-boxCollider.size.x, -boxCollider.size.y, 0f) * 0.5f);
 8:     verticies.Add(new Vector3(boxCollider.size.x, -boxCollider.size.y, 0f) * 0.5f);
 9:
10:     var zeroPosition = new Vector3(transform.position.x, transform.position.y, 0f);
11:
12:     for (int i = 0; i < verticies.Count; i++)
13:     {
14:         verticies[i] += new Vector3(boxCollider.offset.x, boxCollider.offset.y, 0f);
15:         verticies[i] = boxCollider.transform.TransformPoint(verticies[i]);
16:     }
17:
18:     // CommonUtility.IsInDotDegree: 内積の度数が範囲内であるかを判定する自作メソッド
19:     verticies = verticies.Where(v => ((v - zeroPosition).magnitude <= radius) &&
CommonUtility.IsInDotDegree(Forward, (v - zeroPosition).normalized, degree * 0.5f)).ToList();
20:
21:     return verticies;
22: }
```

取得した頂点に対して再びレイキャストを行います。



▲図 4.6　ボリュームライト生成その 3

上図のとおりレイキャストを行いますが、手前側の頂点に対してはヒットするパターン、ヒットしないパターンの 2 頂点が必要になることが分かります。そのまま素直にレイキャストすると結果がどちらか片方しか得られないため、次のように左右に僅かにずらして 2 度のレイキャスト処理を行います。



▲図 4.7　ボリュームライト生成その 4

▼リスト 4.6　コライダーの頂点座標へ 2 度レイキャスト処理

```
 1: const float ColliderCheckDifferenceAngle = 0.1f;
 2: foreach (var target in colliderVertices)
 3: {
 4:     var ray = (target - zeroPosition).normalized;
 5:
 6:     // 左側レイキャスト
 7:     // 生成したレイがライト角度の範囲外である場合はメッシュが狂ってしまう場合があるため除外
 8:     var lRay = Quaternion.Euler(0f, 0f, ColliderCheckDifferenceAngle) * ray;
 9:     if (CommonUtility.IsInDotDegree(transform.right, lRay, degree * 0.5f))
10:     {
11:         var lResult = Physics2D.Raycast(zeroPosition, lRay, radius, layerMask);
12:         allVerticies.Add(Quaternion.Euler(0f, 0f, -inverseAngle) * (lRay *
(lResult.collider == null ? radius : lResult.distance)));
13:     }
14:
15:     // 右レイキャスト
16:     var rRay = Quaternion.Euler(0f, 0f, -ColliderCheckDifferenceAngle) * ray;
17:     if (CommonUtility.IsInDotDegree(transform.right, rRay, degree * 0.5f))
18:     {
19:         var rResult = Physics2D.Raycast(zeroPosition, rRay, radius, layerMask);
20:         allVerticies.Add(Quaternion.Euler(0f, 0f, -inverseAngle) * (rRay *
(rResult.collider == null ? radius : rResult.distance)));
21:     }
22: }
```

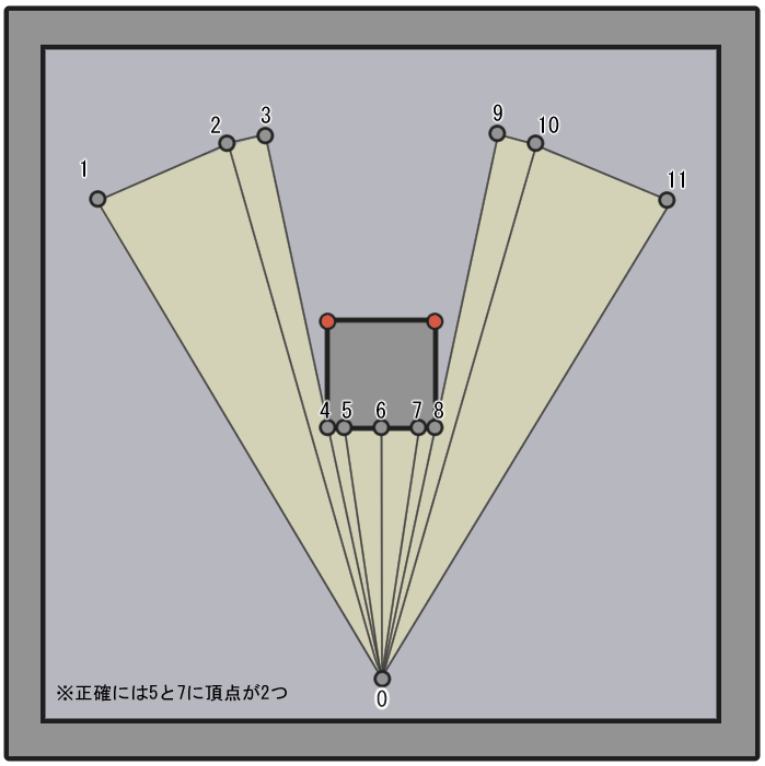
仕上げの頂点ソートとメッシュ情報作成

次は、雑多に追加した頂点情報を時計回りに並べ直します。

▼リスト 4.7　生成した頂点座標を並べなおし

```
 1: // 基準角度からの成す角によって並べ替え
 2: var orderBaseDirection = Quaternion.Euler(0f, 0f, degree / 2f) * Vector3.right;
 3: allVerticies = allVerticies.OrderBy(v =>
Mathf.Repeat(Vector3.SignedAngle(orderBaseDirection, v.normalized, -Vector3.forward),
360)).ToList();
```

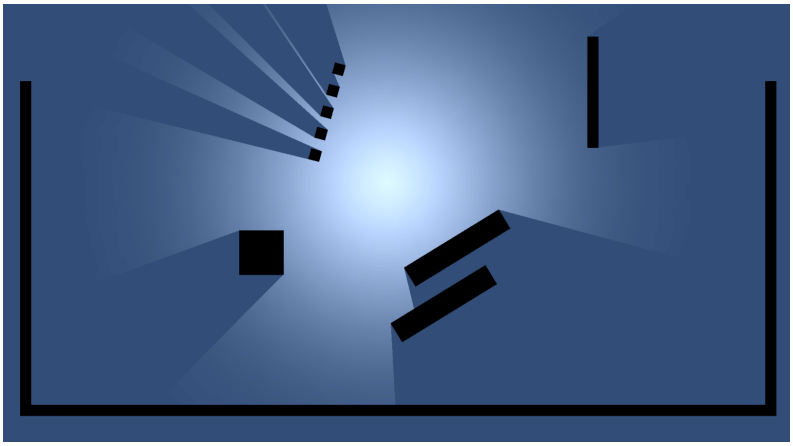
後はベースのメッシュ作成時と同様、UV とインデックスを作成すれば完了です。



▲図 4.8　ボリュームライト生成その 5

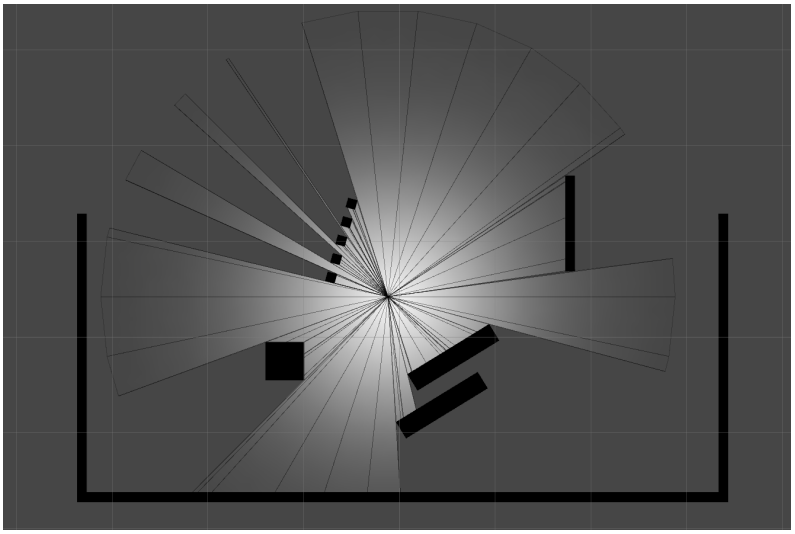
図上では頂点数は 12 個になっていますが、実際には奥側のコライダー頂点へも 2 度のレイキャストを行うため、14 個の頂点数となります。図が見辛くなるので、奥側のコライダーへ向けたレイキャスト頂点である 5 と 7 はひとつの頂点として簡略化して表示しています。

でき上がったメッシュに加算合成のマテリアルを設定し、実際に Unity 上で動作を確認すると次の図のようになりました。



▲図 4.9　ボリュームライト成果物

ワイヤーフレームも確認します。



▲図 4.10　ボリュームライト成果物（ワイヤーフレーム）

意図どおりに動作してそうです。勿論、複数配置しても大丈夫です。

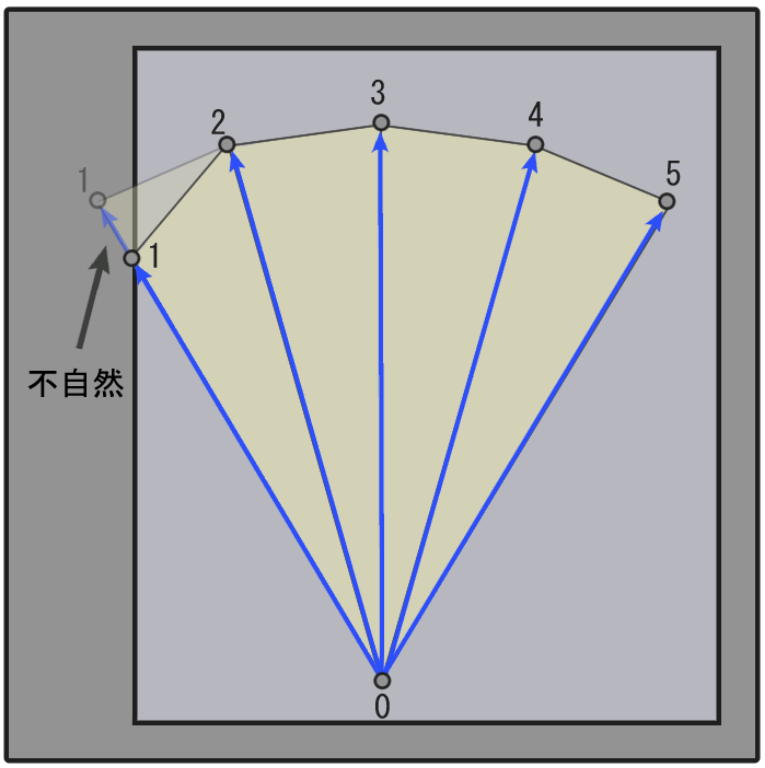
ちなみに図の状態のメッシュでは頂点数は 108 個となっていました。この数値は分割角度の細かさや光源内のコライダー頂点の数によって変動し、負荷が増減するのでご注意下さい。

4.4　補足

これまで紹介させて頂いた手順だけの場合、特定の状況で意図どおりに動作しない場合が存在します。補足として目立つエラーとその解消方法を 1 つ紹介します。

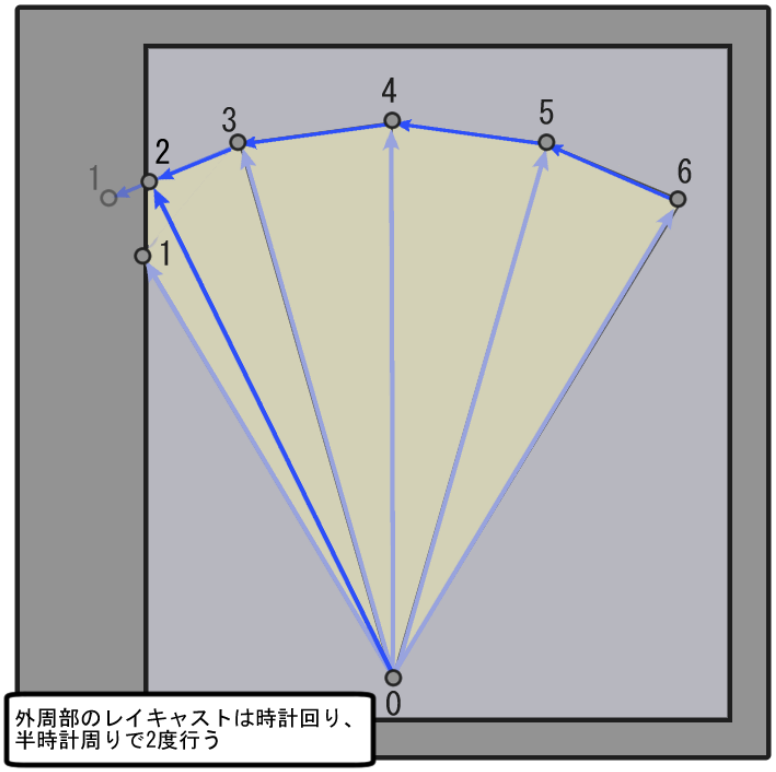
長い壁に衝突した場合

たとえば次の図のように、特定の頂点へのレイキャストが長いボックスコライダーの壁に衝突した場合を考えます。



▲図 4.11　補足 1

図のように不自然な欠けができてしまう場合があります。長い壁だと途中にコライダー頂点も存在しないため、補助の頂点も生成されません。この場合は次の図のように円周部を時計周り、半時計周りに 2 週分のレイキャストを行い、途中に衝突があった場合にそれを追加の頂点とします。



▲図 4.12　補足 2

このようにすると違和感を解消することができます。

4.5　後書きとサンプル

以上で本記事での紹介を終了とさせて頂きます。補足の項で紹介した以外にもエラーの発生するパターン（たとえば十字に交差する長い壁の隅に光が届かないパターンなど）が存在しますが、すべては網羅しきれないのでここでは許容とさせて頂きたく思います。本記事を皆様の技術開発の一助として頂けるなら幸いです。

なお、次の GitHub リポジトリにサンプルプロジェクトをアップロードしているので、参考用にご活用下さい。

■ Unity Version 2021.3.6f1

https://github.com/TogoKosaka/VolumeLight2D_Sample

第 5 章

Unity の Job/Burst を使ったマルチスレッド経路探索

Lingjian Wang

5.1 概要

経路探索はゲーム AI の基本といえます。Unity のデフォルトで使える NavMesh はマルチスレッドに対応できてない為、Agent の数が大きい場合、経路探索の負荷がボトルネックになりがちです。Unity2019 から DOTS のコンセプトが導入され、マルチスレッドコードを安全に扱える Job やバイナリを高速化できる Burst コンパイラが利用できるようになったので、これらを利用して経路探索の高速化を図りたいと思います。

今回紹介する手法は次の 2 つです：

- RaycastGrid/BoxcastGrid：グリッド状のマップの情報を Raycast/Boxcast で取得して、A*で経路探索する
- NavMeshQuery：NavMesh の Job 対応バージョンを利用して探索する（まだ Experimental 状態）

5.2 Unity の DOTS

Unity の DOTS とは「Data-Oriented Technology Stack」の略で、名前から分かるように、DOTS はデータを中心に機能を実装するためのツール集合です。データと処理を分けることでメモリを最適化し、マルチスレッドや LLVM などを利用して、より高速なバイナリコードを生成できるようになります。

DOTS は主に次の機能を提供してくれます：

- C# Job System:　C#で安全なマルチスレッド対応のコードを作成できる仕組み
- Burst Compiler:　C#の機能を一部限定的にしか使えない代わりに、より最適化されたバイナリコードを生成してくれるコンパイラ
- ECS(Entity Component System): GameObject/Component（MonoBehaviour）に変わる新しいゲーム関連データと実装の仕組み。これに関連して、「Physics」・「Network」・「Hybrid Renderer」などいろんな機能を提供してくれる個別のパッケージが存在します

今回の経路探索実装は、主にこの中の **C# Job System** と **Burst Compiler** を使います。

C# Job System

マルチスレッドコードを書くとき、スレッド間のデータアクセスの扱いが繊細で、デバッグもシングルスレッドコードより難しくなります。「C# Job System」は専用の NativeCollection や IJob インターフェースを提供することで、スレッド間のデータアクセス安全を保証してくれるため、マルチスレッドコードの作成コストを削減できます。

「C# Job System」を利用するには、まずやりたい処理を IJob インターフェースで実装する必要があります。

`IJob.Excute()`で実際の処理を書きます。入出力用の変数も一緒に定義します。変数の使い方に合わせて、修飾子を記入するとコンパイラー側で色々最適化してくれます。

- ReadOnly：読み込みのみ
- WriteOnly：書き込みのみ
- 修飾子を付けない：読み書き両方

たとえば、整数配列を入力として渡し、その和を求める Job はリスト 5.1 のように書けます：

▼リスト 5.1　整数配列の和を計算する Job

```
 1: using Unity.Collections;
 2: using Unity.Jobs;
 3:
 4: public struct MyJob : IJob {
 5:     //  足す数字
 6:     [ReadOnly] public NativeArray<int> input;
 7:     //  結果の NativeArray
 8:     public NativeArray<int> result;
 9:
10:     // マルチスレッドでの処理
11:     public void Execute() {
12:         for (int i = 0; i < input.Length; i++) {
13:             result[0] += input[i];
14:         }
15:     }
16: }
```

Job の実行は、

- 実装した Job に必要な値を渡して new
- `job.Schedule()`で Job をスケジュールし、`JobHandle`を取得
- `JobHandle.Complete()`で Job の完了を待ってから、結果を受け取る

コードはこんな感じ：

▼リスト 5.2　Job を実行

```
 1: using Unity.Collections;
 2: using Unity.Jobs;
 3: using UnityEngine;
 4:
 5: public class MyJobBehavior : MonoBehaviour {
 6:
 7:     // Start is called before the first frame update
 8:     void Start() {
 9:         //  足す数字
10:         var numbers = new NativeArray<int>(100, Allocator.TempJob);
11:         for (var i = 0; i < numbers.Length; i++){
12:             numbers[i] = i;
13:         }
14:         //  NativeArray の割り当て
15:         var result = new NativeArray<int>(1, Allocator.TempJob);
16:         //  Job の作成と初期化子を使ってジョブに変数を設定
17:         var myJob = new MyJob {
18:             input = numbers,
19:             result = result
20:         };
21:         //  ジョブのスケジューリング
22:         JobHandle myJobHandle = myJob.Schedule();
23:         //  ジョブの終了を待つ
24:         myJobHandle.Complete();
25:         //  ジョブの結果を変数に入れる
26:         float resultNum = result[0];
27:         //  コンソールに結果を表示
28:         Debug.Log(resultNum);
29:         //  NativeArray をメモリから解放する
30:         numbers.Dispose();
31:         result.Dispose();
32:     }
33: }
```

Burst Compiler

作成した Job を Burst Compiler でコンパイルすることで、生成されたバイナリコードは通常よりも数倍早くなります。

やり方は非常に簡単で、Job を定義したところに`[BurstCompile]`の修飾子（`using Unity.Burst;`も忘れずに）を追加するだけです。

注意点として、Burst パッケージはデフォルトでは Unity に入ってないため、あらかじめ Package Manager でインストールする必要があります。

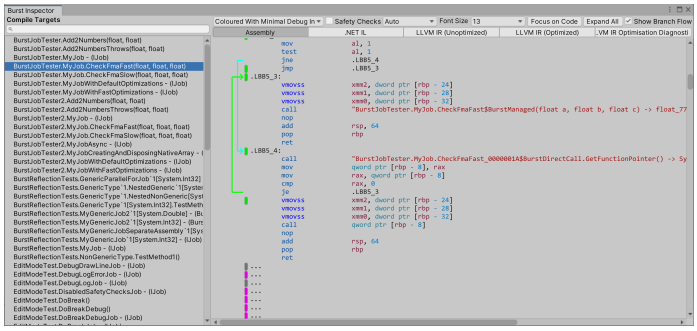
▼リスト 5.3　整数配列の和を計算する Job を Burst でコンパイル

```
 1: using System.Collections;
 2: using System.Collections.Generic;
 3: using Unity.Burst;
 4: using Unity.Collections;
 5: using Unity.Jobs;
 6: using UnityEngine;
 7:
 8: [BurstCompile]
 9: public struct MyJob : IJob {
10:     //  足す数字
11:     [ReadOnly] public NativeArray<int> input;
12:     //  結果の NativeArray
13:     public NativeArray<int> result;
14:
15:     public void Execute() {
16:         for (int i = 0; i < input.Length; i++) {
17:             result[0] += input[i];
18:         }
19:     }
20: }
```

当然ですが、タダでなんでも Busrt で早くできるわけではない。Burst の恩恵を受けるためには、いくつかの制限を受けます：

- C#の一部文法しかサポートしない
- C#の GC と関係あるものは使えない（クラスなど）
- `try...catch...`は使えない
- `static`類変数への書き込み（`Shared Static`という Burst 専用の仕組みを利用する必要がある）

幸、自分が作成した Job が Burst 対応できてるかどうかを確認するのは簡単です。Burst パッケージをインストールした後、「Jobs ＞ Burst ＞ Open Inspector...」から Burst Inspector を開くと図 5.1 のようになります：



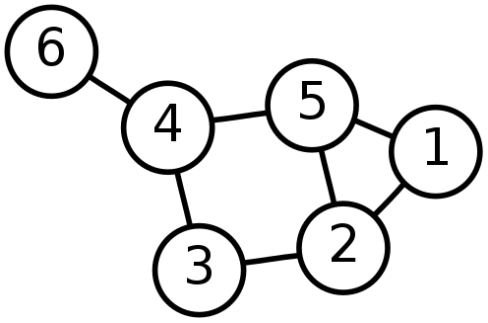
▲図 5.1　Burst Inspector

自分の Job がもし Busrt に対応できていれば、ここでコンパイル結果の Assembly を確認できます。コンパイルが失敗した時は、エラー内容が表示されるので、修正もしやすいです。

5.3　経路探索のおさらい

経路探索を理解するためには、まずグラフのコンセプトを知る必要があります：

- グラフ（英: Graph）: ノード（頂点）群とノード間の連結関係を表すエッジ（枝）群で構成される抽象データ型、and・or その実装である具象データ型である。



▲図 5.2　グラフの例

経路探索とは、グラフに存在する任意のふたつのノードを繋ぐ経路を探索するということです。ほとんどの場合、最短の経路を見つけたいです。

経路探索のアルゴリズムは色々あり、グラフの種類や用途によって最適のアルゴリズムは変わります。ゲームにおいては、汎用性が高く、比較的に速い A*アルゴリズムがもっとも使われています。

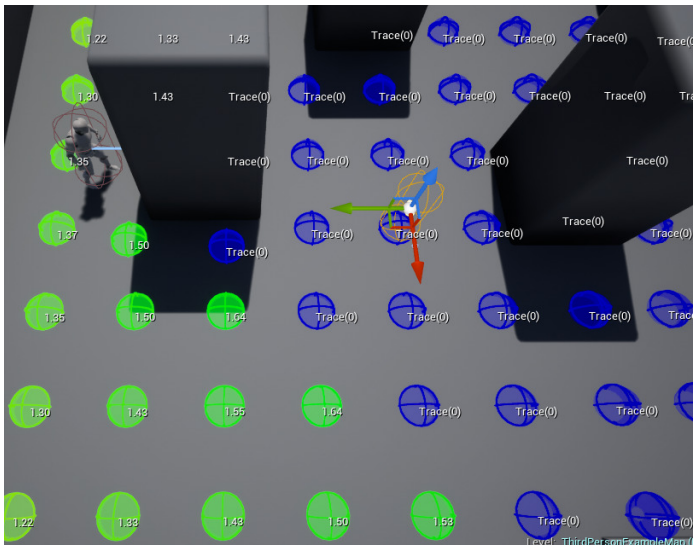
▼リスト 5.4　A*アルゴリズムの擬似コード

```
A_star(始点 s, 終点 t){
    for(すべての頂点 v){ D[v] ← +∞, visited[v] ← no, previous[v] ← nil; }
    集合 S ← {s}, D[s] ← 0; //初期化
    while(S が空でない){
        w ← S の中で D[w]+h_t(w)(w を通る最短路長の予測値) が最も小さい頂点 w;
        if(w = t){ D[w] を返し, 終了;} //探索終了
        S から w を削除;
        visited[w] ← yes; //w までの最短路が確定. w は訪問済みとする.
        for(すべての w の未訪問の隣接頂点 x){
            new_dist ← D[w] + d(w,x); //d(w,x) は (w,x) の辺長
            if(D[x] > new_dist){
                D[x] ← new_dist, previous[x] ← w;
                x が S に入っていなければ x を S に加える;
            }
        }
    }
    終了; //探索失敗
}
```

この記事では、A*を使った経路探索プログラムを実装します。

5.4 「RaycastGrid/BoxcastGrid」による経路探索の実装

Unreal Engine には EQS（Environment Query System） という Experimental 機能があります。ざっくりいうと図 5.3 のようにグリッド状の空間に関する情報をリアルタイムで取得するものです。



▲図 5.3　Environment Query System

ちょうど Unity では DOTS の導入によって、Job で並列計算を活用して大量の Raycast を効率よく実行できるようになった。Raycast でマップの情報（平地か障害物か・高度・コストなど）を取得して、その情報に対して A*のアルゴリズムを適用すれば、NavMesh のベイクなどの事前準備なしにリアルタイムで動作する経路探索ができるようになります。

グリッドデータの取得

Raycast でグリッドの情報を採っていきます。

▼リスト 5.5　必要なパラメータを用意する

```
1: /// <summary>
2: /// グリッド生成用クラス
3: /// </summary>
4: public class RaycastGrid : IDisposable
5: {
6:     public const float HeightMax = 100000f;  // Raycast の最大高度
7:     public float3 Center { get; private set; } // グリッドの中心
```

```
 8:     public int2 Size { get; private set; }  // グリッドのサイズ
 9:     public float2 Stride { get; private set; }  // グリッドのセルのサイズ
10:     public float2 Side { get; private set; }  // グリッドの長さ
11:     public float2 Offset { get; private set; }  // 中心からのオフセット（グリッド座標計算用）
12:     public LayerMask WalkableLayerMask { get; private set; }  // 通行可能のレイヤー
13:     public LayerMask ObstacleLayerMask { get; private set; }  // 障害物レイヤー
14:     public LayerMask RaycastLayerMask { get; private set; }  // 通行可能 + 障害物
15:     public NativeArray<RaycastHit> Hits { get; private set; }  // Raycast 結果配列
16:     public NativeArray<NodeInfo> Infos { get; private set; }  // ノード情報配列
17:
18:     private NativeArray<RaycastCommand> raycastCommands;  // RaycastCommand 配列
19:     private NativeHashMap<int, bool> colliderWalkableMap;  // コライダー情報
20:     public JobHandle Handle { get; private set; }  // Job ハンドル
21:     private readonly Collider[] colliders;  // Collider 用配列
22:
23:     private float radius; // Physics.OverlapSphere 用半径
24: }
```

▼リスト 5.6　コンストラクタでパラメータ初期化

```
 1: public RaycastGrid(
 2:         int2 size, float2 stride, LayerMask walkableLayerMask,
 3:         LayerMask obstacleLayerMask, int maxNumberOfColliders = 128)
 4: {
 5:     // 各変数の初期化
 6:     Size = size;
 7:     Stride = stride;
 8:     Side = Stride * (Size - 1);
 9:     Offset = Stride * Size * 0.5f;
10:     radius = math.sqrt(math.dot(Offset, Offset));
11:     WalkableLayerMask = walkableLayerMask;
12:     ObstacleLayerMask = obstacleLayerMask;
13:     RaycastLayerMask = walkableLayerMask | obstacleLayerMask;
14:
15:     var totalCount = Size.x * Size.y;
16:     // 各種 NativeCollection 配列の初期化
17:     Hits = new NativeArray<RaycastHit>( // Raycast 結果配列
18:             totalCount, Allocator.Persistent, NativeArrayOptions.UninitializedMemory);
19:     raycastCommands = new NativeArray<RaycastCommand>( // RaycastCommand 配列
20:             totalCount, Allocator.Persistent, NativeArrayOptions.UninitializedMemory);
21:     Infos = new NativeArray<NodeInfo>( // ノード情報配列
22:             totalCount, Allocator.Persistent, NativeArrayOptions.UninitializedMemory);
23:     colliders = new Collider[maxNumberOfColliders]; // Collider 用配列
24:     colliderWalkableMap = new NativeHashMap<int, bool>(
25:             maxNumberOfColliders, Allocator.Persistent); // コライダー情報
26: }
```

▼リスト 5.7　グリッドを更新するためのメソッド

```
 1: public void Update(Vector3 center, JobHandle depsHandle = default)
 2: {
 3:     Handle.Complete(); // 前回の Job 終了を待つ。待たないと変数のアクセスができない
 4:
 5:     // グリッド座標の計算をしやすくするため、グリッドの開始位置を計算する
 6:     Center = new float3(center.x - Offset.x, center.y, center.z - Offset.y);
 7:
 8:     // コライダー情報の更新
 9:     colliderWalkableMap.Clear();
10:     colliderWalkableMap.Add(0, false);
11:     var num = Physics.OverlapSphereNonAlloc(center, radius, colliders, RaycastLayerMask);
12:
13:     for (int i = 0; i < num; i++)
14:     {
15:         var collider = colliders[i];
```

```
16:         colliderWalkableMap.Add(
17:                 collider.GetInstanceID(),
18:                 (WalkableLayerMask & (1 << collider.gameObject.layer)) != 0);
19:     }
20:
21:     // ジョブのスケジュール
22:     Handle = RaycastJob(depsHandle);
23: }
```

▼リスト 5.8　RaycastCommand の実行

```
 1: private JobHandle RaycastJob(JobHandle depsHandle)
 2: {
 3:     // RaycastCommand 配列の値を更新する
 4:     for (int i = 0; i < Size.x; i++)
 5:     {
 6:         var start = Size.y * i;
 7:         var xoffset = i * Stride.x;
 8:         for (int j = 0; j < Size.y; j++)
 9:         {
10:             var index = start + j;
11:             var zoffset = j * Stride.y;
12:             raycastCommands[index] = new RaycastCommand(
13:                     new Vector3(Center.x + xoffset, HeightMax, Center.z + zoffset),
14:                     Vector3.down, maxHits: 1, layerMask: RaycastLayerMask);
15:         }
16:     }
17:
18:     // Raycast ジョブは並行計算でやるため、ScheduleBatch でバッチサイズ（ここでは Size.y）を
19:     // 決めてスケジュールする必要がある
20:     var rangedRaycastHandle = RaycastCommand.ScheduleBatch(
21:             raycastCommands, Hits, Size.y, depsHandle);
22:
23:     // Raycast データをプロセスするための Job（後述）を実行する
24:     var processJob = new ProcessRaycastResultJob()
25:     {
26:         hits = Hits,
27:         size = Size,
28:         colliderWalkableMap = colliderWalkableMap,
29:         nodeInfos = Infos
30:     };
31:     return processJob.Schedule(Hits.Length, Size.y, rangedRaycastHandle);
32: }
```

データプロセスは並列処理できるので、IJob の並列処理版の IJobParallelFor を使います：

▼リスト 5.9　RaycastCommand の結果を整理する Job

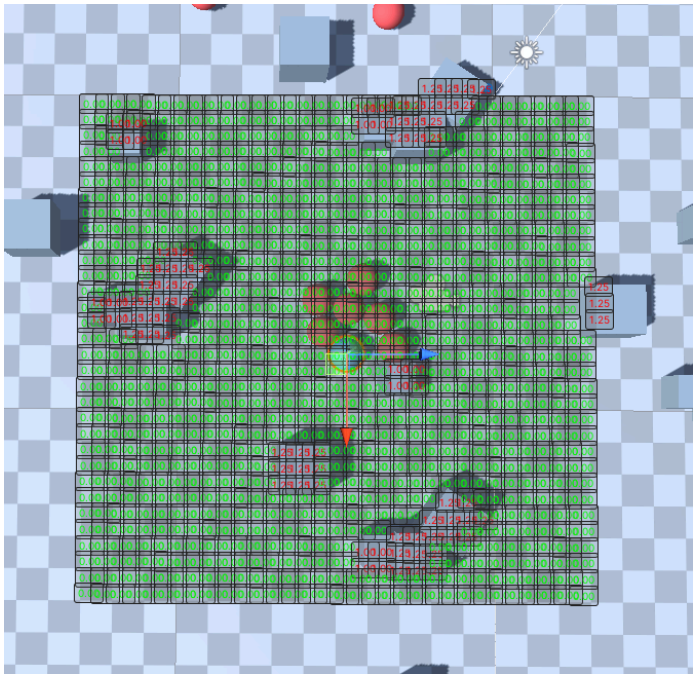
```
 1: using System.Runtime.CompilerServices;
 2: using Unity.Burst;
 3: using Unity.Collections;
 4: using Unity.Jobs;
 5: using Unity.Mathematics;
 6: using UnityEngine;
 7:
 8: // ノードデータ構造体
 9: public struct NodeInfo
10: {
11:     public int2 coord;
12:     public bool walkable;
13: }
14:
15: // データ処理ジョブ、Burst コンパイラーを適用する
```

```
16: [BurstCompile]
17: public struct ProcessRaycastResultJob : IJobParallelFor
18: {
19:     [ReadOnly] public NativeArray<RaycastHit> hits;
20:     [ReadOnly] public int2 size;
21:     [ReadOnly] public NativeHashMap<int, bool> colliderWalkableMap;
22:
23:     public NativeArray<NodeInfo> nodeInfos;
24:
25:     // グリッド座標をワールド座標に変換する関数
26:     // MethodImpl 修飾子でコンパイル時は Inline にして少し高速化できる
27:     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28:     public int2 Index2Coord(int index)
29:     {
30:         var x = index / size.y;
31:         var y = -math.mad(x, size.y, -index);
32:         return new int2(x, y);
33:     }
34:
35:     // 並列処理の実装、IJob との違いはスレッド番号がもらえる
36:     public void Execute(int i)
37:     {
38:         var hit = hits[i];
39:         if (colliderWalkableMap.ContainsKey(hit.colliderInstanceID))
40:         {
41:             // コライダーが小ライダー情報に入ってる
42:             nodeInfos[i] = new NodeInfo()
43:             {
44:                 coord = Index2Coord(i),
45:                 walkable = colliderWalkableMap[hit.colliderInstanceID]
46:             };
47:         }
48:         else
49:         {
50:             // 万が一コライダー情報に入ってない時の処理（何も Hit してない場合など）
51:             nodeInfos[i] = new NodeInfo()
52:             {
53:                 coord = Index2Coord(i),
54:                 walkable = true
55:             };
56:         }
57:     }
58: }
```

▼リスト 5.10　最後は NativeCollection 解放用メソッドの実装を忘れずに

```
1: public void Dispose()
2: {
3:     Handle.Complete(); // まずは前にスケジュールした Job の完了を待たなければならない
4:
5:     Hits.Dispose();
6:     raycastCommands.Dispose();
7:     Infos.Dispose();
8:     colliderWalkableMap.Dispose();
9: }
```

これで RaycastGrid.Updateでグリッド情報を更新できるようになります。



▲図 5.4　32x32 のグリッド：赤は障害物、青は通行可能なところ

A*

A*を実装するために、まず Unity の Mathematics パッケージを Package Manager 経由でインストールします。

A*を実装するためには、Min-Max Heap というデータ構造を使う必要がある（次の探索対象を確定する時の計算コスト削減用）。Unity では Job で使える NativeCollection 対応の Min-Max Heap がないため、Github で公開されている OSS を借ります。

▼リスト 5.11　A*ノード情報保持用構造体

```
 1: // 同じかどうか判定できるようにするため、IEquatable インターフェースを実装する
 2: public struct AStarNode : IEquatable<AStarNode>
 3: {
 4:     public int index;
 5:     public int2 coord;
 6:     public float cScore;
 7:     public float score;
 8:     public int prevNodeIndex;
 9:
10:     public readonly static AStarNode invalid = new() { index = -1 };
11:
12:     // 比較ロジックの実装
13:     public bool Equals(AStarNode other)
14:     {
15:         return index == other.index;
16:     }
```

```
17:
18:     // override object.GetHashCode
19:     public override int GetHashCode()
20:     {
21:         return index;
22:     }
23: }
```

▼リスト 5.12　A*ノード比較用構造体

```
 1: // 値の比較ができる様にするため、IComparer インターフェースを実装する
 2: public struct MinAStarNode : IComparer<AStarNode>
 3: {
 4:     // score の値で大きさを決める
 5:     public int Compare(AStarNode x, AStarNode y)
 6:     {
 7:         return x.score - y.score < 0 ? -1 : 1;
 8:     }
 9: }
```

▼リスト 5.13　A*の Job 実装

```
 1: [BurstCompile]
 2: public struct AStarJob : IJob
 3: {
 4:     [ReadOnly] public NativeArray<RaycastHit> hits;
 5:     [ReadOnly] public NativeArray<NodeInfo> nodeInfos;
 6:     [ReadOnly] public float3 gridCenter;
 7:     [ReadOnly] public int2 gridSize;
 8:     [ReadOnly] public float2 gridStride;
 9:     [ReadOnly] public float agentSize;
10:     [ReadOnly] public float3 start;
11:     [ReadOnly] public float3 destination;
12:     [ReadOnly] public float agentStepHeightMin;
13:     [ReadOnly] public float agentStepHeightMax;
14:
15:     /// <summary>
16:     /// パス保存用配列。
17:     /// </summary>
18:     public NativeList<float3> path;
19:
20:     /// <summary>
21:     /// 探索候補ノードの集まり
22:     /// MinHeap を使うことで、スコアが一番小さいノードを O(1) で取り出せる（挿入コストは O(logN)）
23:     /// </summary>
24:     public NativeHeap<AStarNode, MinAStarNode> heap;
25:
26:     /// <summary>
27:     /// 探索済みノードの集まり
28:     /// 入ってるかどうかの判断しか行わないから、HashMap を使用
29:     /// </summary>
30:     public NativeHashMap<int, AStarNode> explored;
31:
32:     int startIndex;  // 開始地点のノード Index
33:     int destIndex; // ゴール地点のノード Index
34:     int2 checkRadius; // サイズ補正半径サイズ保持用
35:
36:     public void Execute()
37:     {
38:         // 必要変数の計算
39:         startIndex = WorldPosition2GridIndex(start);
40:         destIndex = WorldPosition2GridIndex(destination);
41:         checkRadius = (int2)math.ceil(agentSize / gridStride) + 1;
42:
43:         // スタートノードを作って、探索候補と探索済みグループに入れる
```

```
44:         var startNode = new AStarNode()
45:         {
46:             index = startIndex,
47:             coord = nodeInfos[startIndex].coord,
48:             cScore = 0f,
49:             score = 0f,
50:             prevNodeIndex = -1
51:         };
52:         heap.Insert(startNode);
53:         explored.Add(startNode.index, startNode);
54:
55:         // 回帰的探索
56:         while (heap.Count > 0)
57:         {
58:             // 次の候補
59:             var next = heap.Pop();
60:
61:             // 周りの８方向隣接ノードを探索
62:             foreach (var tuple in [(1, 1), (1, 0), ..., (-1, -1)]) {
63:                 if (ExploreNeighbourNode(next, new int2(tuple.Item1, tuple.Item2)))
64:                 {
65:                     // 探索終了したら、パスを計算して終了
66:                     CalculatePath();
67:                     break;
68:                 }
69:             }
70:         }
71:     }
72: }
```

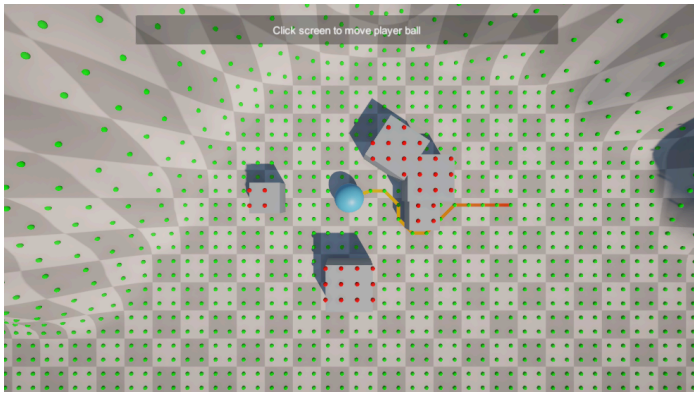
▼リスト 5.14　A*の Job 実装用 Helper 関数

```
 1: // ワールド座標をグリッド座標に変換
 2: private int2 WorldPos2GridCoord(float3 pos)
 3: {
 4:     var diff = pos - gridCenter;
 5:     var x = math.round(diff.x / gridStride.x);
 6:     var y = math.round(diff.z / gridStride.y);
 7:     return new int2((int)x, (int)y);
 8: }
 9:
10: // ワールド座標をグリッド Index に変換
11: private int WorldPosition2GridIndex(float3 pos)
12: {
13:     var coord = WorldPos2GridCoord(pos);
14:     return Coord2Index(coord);
15: }
16:
17:
18: // スコア計算用
19: private float CalculateHeuristicScore(int index)
20: {
21:     var diff = hits[index].point - hits[destIndex].point;
22:     return math.mad(diff.x, diff.x, diff.z * diff.z);
23: }
24:
25: // グリッド座標を Index に変換
26: private int Coord2Index(int2 coord)
27: {
28:     return math.mad(coord.x, gridSize.y, coord.y);
29: }
30:
31: // 新ノード構造体生成用
32: private AStarNode GetAStarNode(int index, int prevNodeIndex)
33: {
34:     var diff = hits[index].point - hits[prevNodeIndex].point;
35:     var cScore = explored[prevNodeIndex].cScore
36:             + math.mad(diff.x, diff.x, diff.z * diff.z);
37:     return new AStarNode()
```

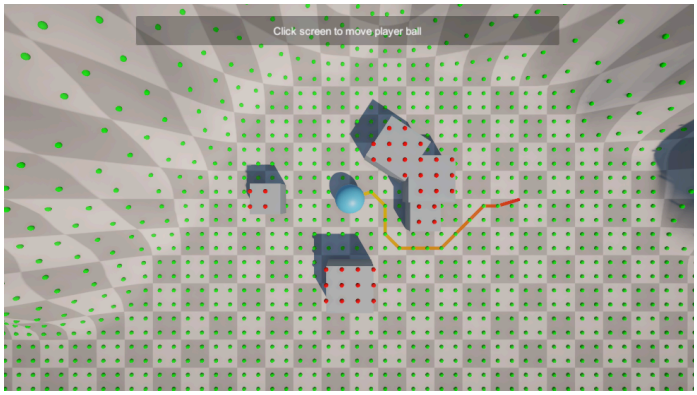
```
38:     {
39:         index = index,
40:         coord = nodeInfos[index].coord,
41:         cScore = cScore,
42:         score = CalculateHeuristicScore(index) + cScore,
43:         prevNodeIndex = prevNodeIndex
44:     };
45: }
46:
47: // 隣のノードを探索する。ゴールまで辿り着いたら true を返す
48: private bool ExploreNeighbourNode(AStarNode node, int2 direction)
49: {
50:     // グリッドの外になってるか
51:     var coord = node.coord + direction;
52:     if (coord.x < 0 || coord.x >= gridSize.x || coord.y < 0 || coord.y >= gridSize.y)
53:         return false;
54:
55:     // Index 取得
56:     var index = Coord2Index(coord);
57:
58:     // 既に探索済みか
59:     if (explored.ContainsKey(index)) return false;
60:
61:     // 探索済みに入れる
62:     var next = GetAStarNode(index, node.index);
63:     explored.Add(index, next);
64:
65:     // 通行可能か
66:     var nodeInfo = nodeInfos[index];
67:     if (!nodeInfo.walkable) return false;
68:
69:     // 垂直角度制限をチェック
70:     var diff = hits[index].point.y - hits[node.index].point.y;
71:     if (diff > agentStepHeightMax || diff < agentStepHeightMin) return false;
72:
73:     // ゴールに着いたか
74:     if (index == destIndex) return true;
75:
76:     // エージェントのサイズ補正
77:     if (math.max(checkRadius.x, checkRadius.y) > 1 && CheckIllegalNeighbourNodes(next))
78:         return false;
79:
80:     // 探索候補に入れる
81:     heap.Insert(next);
82:     return false;
83: }
84:
85: // 指定方向の隣接ノードも通行可能かをチェック（エージェントサイズ補正用）
86: private bool CheckIllegalNeighbourNode(AStarNode node, int2 direction)
87: {
88:     // check out of range
89:     var coord = node.coord + direction;
90:     if (coord.x < 0 || coord.x >= gridSize.x || coord.y < 0 || coord.y >= gridSize.y)
91:         return true;
92:
93:     // get index
94:     var index = Coord2Index(coord);
95:
96:     // check if walkable
97:     var nodeInfo = nodeInfos[index];
98:     if (!nodeInfo.walkable) return true;
99:
100:     // check if within angle limit
101:     var diff = hits[index].point.y - hits[node.index].point.y;
102:     if (diff > agentStepHeightMax || diff < agentStepHeightMin) return true;
103:
104:     return false;
105: }
106:
107: // 周りの一定範囲内のノードも通行可能かどうか（エージェントサイズ補正用）
108: private bool CheckIllegalNeighbourNodes(AStarNode node)
109: {
```

```
110:     // x > 0 & y == 0 case
111:     for (int i = 1; i < checkRadius.x; i++)
112:     {
113:         if (CheckIllegalNeighbourNode(node, new int2(i, 0))) return true;
114:         if (CheckIllegalNeighbourNode(node, new int2(-i, 0))) return true;
115:     }
116:
117:     // x == 0 & y> 0 case
118:     for (int j = 1; j < checkRadius.y; j++)
119:     {
120:         if (CheckIllegalNeighbourNode(node, new int2(0, j))) return true;
121:         if (CheckIllegalNeighbourNode(node, new int2(0, -j))) return true;
122:     }
123:
124:     // x > 0 & y > 0 case
125:     for (int i = 1; i < checkRadius.x; i++)
126:     {
127:         for (int j = 1; j < checkRadius.y; j++)
128:         {
129:             if (CheckIllegalNeighbourNode(node, new int2(-i, -j))) return true;
130:             if (CheckIllegalNeighbourNode(node, new int2(-i, j))) return true;
131:             if (CheckIllegalNeighbourNode(node, new int2(i, -j))) return true;
132:             if (CheckIllegalNeighbourNode(node, new int2(i, j))) return true;
133:         }
134:     }
135:
136:     // all checks passed
137:     return false;
138: }
139:
140: // パスを計算する
141: private void CalculatePath()
142: {
143:     // expected path length should be no more than the size of the grid
144:     var stack = new NativeStack<float3>(
145:             math.mad(gridSize.x, 5, gridSize.y), Allocator.Temp);
146:     explored.TryGetValue(destIndex, out var next);
147:
148:     while (next.prevNodeIndex > -1)
149:     {
150:         stack.Push(hits[next.index].point);
151:         explored.TryGetValue(next.prevNodeIndex, out next);
152:     }
153:
154:     // add start point
155:     path.Add(start);
156:
157:     // reverse stack to get path
158:     while (stack.TryPop(out var pt))
159:     {
160:         path.Add(pt);
161:     }
162:
163:     // change destination to actual value
164:     if (path.Length > 1) path[^1] = destination;
165:
166:     // clear
167:     stack.Dispose();
168: }
```

補足：CheckIllegalNeighbourNodesはエージェントのサイズ補正計算を行います。サイズ補正なしのパスは図 5.5 のように障害物と密接しているため、エージェントが大きすぎると通行困難です。サイズ補正を入れることで図 5.6 のようになり、エージェントのサイズに合わせたパスになります。



▲図 5.5　サイズ補正なし



▲図 5.6　サイズ補正あり

使いやすくするための Wrapper を用意する

実際の経路探索は複数の Job が絡み合って扱いづらいため、グリッド生成と A*計算の処理をまとめて、使いやすい Wrapper を作ります。

▼リスト 5.15　A*経路探索 Job 用 Wrapper

```
 1: public class AStarPath : IDisposable
 2: {
 3:     public RaycastGrid Grid { get; private set; }
 4:
 5:     public JobHandle Handle { get; private set; }
 6:     public bool IsScheduled { get; private set; } = false;
 7:
 8:     private NativeList<float3> path; // パス保存用配列
 9:     private NativeHeap<AStarNode, MinAStarNode> heap; // 探索候補ノード保存用
10:     private NativeHashMap<int, AStarNode> explored; // 探索済みノード保存用
11:
```

```
12:     /// <summary>
13:     /// コンストラクター
14:     /// </summary>
15:     /// <param name="grid">Raycast grid</param>
16:     public AStarPath(RaycastGrid grid)
17:     {
18:         Grid = grid;
19:         var capacity = grid.Size.x * grid.Size.y;
20:         path = new NativeList<float3>(capacity, Allocator.Persistent);
21:         heap = new NativeHeap<AStarNode, MinAStarNode>(Allocator.Persistent, capacity);
22:         explored = new NativeHashMap<int, AStarNode>(capacity, Allocator.Persistent);
23:     }
24:
25:     /// <summary>
26:     /// 公開される経路探索メソッド
27:     /// </summary>
28:     /// <param name="start">開始地点座標</param>
29:     /// <param name="destination">ゴール地点座標</param>
30:     /// <returns>開始・ゴール地点がグリッド範囲外になる場合は false, それ以外は true</returns>
31:     public bool SchedulePathFinding(
32:             Vector3 start, Vector3 destination, float agentSize,
33:             float agentStepHeightMin, float agentStepHeightMax)
34:     {
35:         Handle.Complete();
36:         if (Grid.IsPointOutsideGrid(start) || Grid.IsPointOutsideGrid(destination))
37:             return false;
38:         Handle = FindPathJob(
39:                 start, destination, agentSize,
40:                 agentStepHeightMin, agentStepHeightMax, Grid.Handle);
41:         IsScheduled = true;
42:         return true;
43:     }
44:
45:     /// <summary>
46:     /// Job 終了を待ち、経路を取得
47:     /// </summary>
48:     /// <returns>取得した経路</returns>
49:     public Vector3[] GetPath()
50:     {
51:         if (!IsScheduled) return new Vector3[0];
52:
53:         Handle.Complete();
54:         IsScheduled = false;
55:         var size = path.Length;
56:         var result = new Vector3[size];
57:         for (int i = 0; i < size; i++)
58:         {
59:             result[i] = path[i];
60:         }
61:         return result;
62:     }
63:
64:     /// <summary>
65:     /// Job 終了を待ち、経路を取得
66:     /// </summary>
67:     public void Cancel()
68:     {
69:         if (!IsScheduled) return;
70:
71:         Handle.Complete();
72:         IsScheduled = false;
73:         path.Clear();
74:     }
75:
76:     /// <summary>
77:     /// Job を用意してスケジュール
78:     /// </summary>
79:     private JobHandle FindPathJob(
80:             float3 start, float3 destination, float agentSize,
81:             float agentStepHeightMin, float agentStepHeightMax,
82:             JobHandle dependsOn = default)
83:     {
```

```
84:         // preparation
85:         path.Clear();
86:         heap.Clear();
87:         explored.Clear();
88:
89:         // create job
90:         var job = new AStarJob()
91:         {
92:             hits = Grid.Hits,
93:             nodeInfos = Grid.Infos,
94:             gridCenter = Grid.Center,
95:             gridSize = Grid.Size,
96:             gridStride = Grid.Stride,
97:             agentSize = agentSize,
98:             start = start,
99:             destination = destination,
100:             agentStepHeightMin = agentStepHeightMin,
101:             agentStepHeightMax = agentStepHeightMax,
102:             path = path,
103:             heap = heap,
104:             explored = explored
105:         };
106:
107:         return job.Schedule(dependsOn);
108:     }
109:
110:     // 後始末用
111:     public void Dispose()
112:     {
113:         Handle.Complete();
114:         path.Dispose();
115:         heap.Dispose();
116:         explored.Dispose();
117:     }
118: }
```

経路探索＆最適化

▼リスト 5.16　経路探索のやり方

```
 1: using System;
 2: using Unity.Mathematics;
 3: using UnityEngine;
 4:
 5: public class MyPathfinding : MonoBehaviour {
 6:
 7:     public RaycastGrid Grid { get; private set; } // グリッド
 8:     private readonly AStarPath Path; // A*計算用
 9:
10:     void Start() {
11:         // Grid 生成＆アップデート
12:         Grid = new RaycastGrid(
13:                 gridSize, gridStride,
14:                 walkableLayerMask, obstacleLayerMask, maxNumberOfColliders);
15:         Grid.Update(center, Path.Handle);
16:
17:         // A*で経路探索
18:         if (Path.SchedulePathFinding(
19:                 start, destination, agentSize, agentStepHeightMin, agentStepHeightMax))
20:         {
21:             var path = Path.GetPath();
22:             Debug.Log(path.Length); // 結果を出力
23:         }
24:     }
25: }
```

複数のエージェントで同じグリッドを使い回すことで、計算時間を減らせます。グリッドの更新は、必要な時だけすれば OK です：

- 中心が移動した時（グリッドがカバーする範囲が変わる）
- 障害物を追加・削除・形状変化した時

5.5 NavMeshQuery を試す

NavMeshQuery は Job で使える NavMesh の API は、現時点ではまだ Experimental 状態です。基本的に NavMesh 同様で、使うためにはあらかじめシーンの NavMesh をベイクする必要があります。

▼リスト 5.17　NavMeshQuery の Job を用意する

```
 1: using Unity.Burst;
 2: using Unity.Collections;
 3: using Unity.Jobs;
 4: using Unity.Mathematics;
 5: using UnityEngine.Experimental.AI;
 6:
 7: [BurstCompile]
 8: public struct NavMeshQueryJob : IJob
 9: {
10:     [ReadOnly] public NavMeshLocation StartPosition;
11:     [ReadOnly] public NavMeshLocation TargetPosition;
12:     [ReadOnly] public float3 Extents;
13:     [ReadOnly] public int agentTypeId;
14:     [ReadOnly] public int areaMask;
15:     [ReadOnly] public int iterations;
16:
17:     public NavMeshQuery Query;
18:     public NativeList<float3> path;
19:
20:     public void Execute()
21:     {
22:         // 経路探索が終了するまで実行
23:         var status = PathQueryStatus.InProgress;
24:         while (status == PathQueryStatus.InProgress)
25:         {
26:             status = Query.UpdateFindPath(iterations, out var iterationsPerformed);
27:         }
28:
29:         // 終了処理
30:         Query.EndFindPath(out var pathSize);
31:
32:         // パスを取得
33:         var pathPolygonId = new NativeArray<PolygonId>(pathSize, Allocator.Temp);
34:         pathSize = Query.GetPathResult(pathPolygonId);
35:         var pathStraight = new NativeArray<NavMeshLocation>(
36:                 path.Capacity, Allocator.Temp);
37:         var pathFlag = new NativeArray<StraightPathFlags>(
38:                 path.Capacity, Allocator.Temp);
39:         var vertexSize = new NativeArray<float>(path.Capacity, Allocator.Temp);
40:         var straightPathCount = 0;
41:         PathUtils.FindStraightPath(
42:                 Query, StartPosition.position, TargetPosition.position, pathPolygonId,
43:                 pathSize, ref pathStraight, ref pathFlag, ref vertexSize,
44:                 ref straightPathCount, path.Capacity);
45:         for (int i = 0; i < straightPathCount; i++)
```

```
46:         {
47:             path.Add(pathStraight[i].position);
48:         }
49:
50:         // 使い終わった配列の解放
51:         pathPolygonId.Dispose();
52:         pathStraight.Dispose();
53:         pathFlag.Dispose();
54:         vertexSize.Dispose();
55:     }
56: }
```

上記 Job の実装は次のユーティリティ関数をいくつか使用してます：

▼リスト 5.18　NavMeshQuery の Wrapper

```
 1: using System;
 2: using Unity.Collections;
 3: using Unity.Jobs;
 4: using Unity.Mathematics;
 5: using UnityEngine;
 6: using UnityEngine.Experimental.AI;
 7:
 8: public class NavMeshQueryPath : IDisposable
 9: {
10:     public JobHandle Handle { get; private set; }
11:     public bool IsScheduled { get; private set; } = false;
12:
13:     private NativeList<float3> path;
14:     private NavMeshQuery query;
15:
16:
17:     /// <summary>
18:     /// コンストラクター
19:     /// </summary>
20:     public NavMeshQueryPath(int maxPathSize)
21:     {
22:         path = new NativeList<float3>(maxPathSize, Allocator.Persistent);
23:         query = new NavMeshQuery(
24:                 NavMeshWorld.GetDefaultWorld(), Allocator.Persistent, path.Capacity);
25:     }
26:
27:     /// <summary>
28:     /// パスを計算する
29:     /// </summary>
30:     public bool SchedulePathFinding(
31:             Vector3 start, Vector3 destination, Vector3 extents,
32:             int agentTypeId, int areaMask, int iterations)
33:     {
34:         Handle.Complete();
35:
36:
37:         var startLoc = query.MapLocation(start, extents, agentTypeId, areaMask);
38:         var destinationLoc = query.MapLocation(
39:                 destination, extents, agentTypeId, areaMask);
40:
41:         try
42:         {
43:             query.BeginFindPath(startLoc, destinationLoc, areaMask);
44:         }
45:         catch (System.Exception)
46:         {
47:             return false;
48:         }
49:
50:         // Job を発行
51:         var job = new NavMeshQueryJob()
52:         {
53:             Query = query,
```

```
54:             StartPosition = startLoc,
55:             TargetPosition = destinationLoc,
56:             Extents = extents,
57:             agentTypeId = agentTypeId,
58:             areaMask = areaMask,
59:             iterations = iterations,
60:             path = path
61:         };
62:
63:         Handle = job.Schedule();
64:         IsScheduled = true;
65:         return true;
66:     }
67:
68:     /// <summary>
69:     /// 計算されたパスを取得する
70:     /// </summary>
71:     public Vector3[] GetPath()
72:     {
73:         if (!IsScheduled) return new Vector3[0];
74:
75:         Handle.Complete();
76:         IsScheduled = false;
77:         var size = path.Length;
78:         var result = new Vector3[size];
79:         for (int i = 0; i < size; i++)
80:         {
81:             result[i] = path[i];
82:         }
83:         return result;
84:     }
85:
86:     // 後始末用
87:     public void Dispose()
88:     {
89:         Handle.Complete();
90:         path.Dispose();
91:         query.Dispose();
92:     }
93: }
```

5.6 負荷比較

セットアップ：

- 同じフィールドで、複数の Agent を毎フレームで、ランダムのゴールを経路探索させる。
  - 使われるフィールド（Terrain）のサイズは 150x150、ランダムに障害物を配置してある
- NavMesh/NavMeshQuery の NavMesh データは事前にベイク済みのものを使用。
- Unity の Player 設定
  - IL2CPP + .NET Standard 2.1
  - Incremental GC は ON（特に NavMeshQuery の方は ON にすることで性能改善）
- 「同期」と「非同期」について
  -「同期」は経路探索の結果をその場で受け取る
  -「非同期」は経路探索の Job を発行した後、結果を次のフレームで受け取る

計測用端末：

- Android: Xiaomi M11 Lite 5G（Snapdragon 780G, 5nm, 2.4GHz x 1 + 2.2GHz x 3 + 1.9GHz x 4）
- iOS： iPad Air(4th gen)（Apple A14, 5nm, 3.1GHz x 2+ 1.8GHz x 4）

経路探索用の MonoBehaviour.Update() のコスト（単位は ms）

Android: Xiaomi M11 Lite 5G（Snapdragon 780G, 5nm, 1+3+4 cores）

▼表 5.1　NavMesh / NavMeshQuery

Agent 数	5 体	15 体	25 体	50 体	100 体
NavMesh	1.55	3.66	6.27	10.04	13.35
NavMeshQuery	0.96	2.37	3.58	6.65	10.23

▼表 5.2　RaycastGrid/BoxcastGrid (サイズ 16x16)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.32	0.58	0.83	1.18	2.06
RaycastGrid(同期)	1.11	0.85	0.10	1.22	2.00
BoxcastGrid	0.28	0.62	0.90	1.35	1.82
BoxcastGrid(同期)	1.12	0.63	0.90	1.39	1.70

▼表 5.3　RaycastGrid/BoxcastGrid (サイズ 32x32)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.33	0.78	1.06	1.41	2.20
RaycastGrid(同期)	1.14	1.50	1.64	1.35	2.13
BoxcastGrid	0.43	1.07	1.15	1.16	1.94
BoxcastGrid(同期)	1.15	1.68	1.53	1.14	2.00

▼表 5.4　RaycastGrid/BoxcastGrid (サイズ 64x64)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.45	1.04	1.23	1.89	3.45
RaycastGrid(同期)	1.25	1.74	2.00	1.85	3.54
BoxcastGrid	0.89	0.65	0.80	1.82	3.23
BoxcastGrid(同期)	1.52	1.62	1.77	1.80	3.52

▼表 5.5　RaycastGrid/BoxcastGrid (サイズ 128x128)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.72	1.56	2.18	4.49	7.15
RaycastGrid(同期)	1.53	2.28	2.96	5.20	7.43
BoxcastGrid	0.62	1.18	2.01	4.64	6.67
BoxcastGrid(同期)	1.76	2.04	3.13	5.23	7.45

iOS： iPad Air(4th gen)（Apple A14, 5nm, 2+4 cores）

▼表 5.6　NavMesh / NavMeshQuery

Agent 数	5 体	15 体	25 体	50 体	100 体
NavMesh	2.01	4.27	5.54	6.65	8.59
NavMeshQuery	2.91	5.81	7.55	8.26	10.04

▼表 5.7　RaycastGrid/BoxcastGrid (サイズ 16x16)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.25	0.61	0.97	1.56	2.27
RaycastGrid(同期)	0.35	0.59	0.94	1.55	2.22
BoxcastGrid	0.22	0.52	0.89	1.51	2.28
BoxcastGrid(同期)	0.33	0.51	0.86	1.48	2.28

▼表 5.8　RaycastGrid/BoxcastGrid (サイズ 32x32)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.29	0.79	1.24	1.84	2.68
RaycastGrid(同期)	0.40	0.89	1.30	1.90	2.71
BoxcastGrid	0.25	0.60	0.96	1.50	2.25
BoxcastGrid(同期)	0.37	0.72	1.05	1.47	2.33

▼表 5.9　RaycastGrid/BoxcastGrid (サイズ 64x64)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.38	1.04	1.56	2.26	3.41
RaycastGrid(同期)	0.48	1.11	1.68	2.26	3.36
BoxcastGrid	0.30	0.74	1.16	1.64	3.14
BoxcastGrid(同期)	0.40	0.90	1.27	1.62	2.94

▼表 5.10　RaycastGrid/BoxcastGrid (サイズ 128x128)

Agent 数	5 体	15 体	25 体	50 体	100 体
RaycastGrid	0.56	1.51	2.23	2.87	3.88
RaycastGrid(同期)	0.68	1.62	2.32	2.91	3.90
BoxcastGrid	0.41	0.75	1.88	1.79	3.33
BoxcastGrid(同期)	1.02	1.99	2.03	1.87	3.34

負荷分析

- NavMesh と比べて、RaycastGrid/BoxcastGrid は 2~6 倍程度軽い（Grid Size 依存)。
- NavMeshQuery は NavMesh の Job 版ですが、まだ Experimental であり、現時点では負荷においてあんまり明確な利点はない。
- RaycastGrid/BoxcastGrid の非同期処理は同期と比べて少し軽いが、大差はない。
- BoxcastGrid は RaycastGrid に比べて少し軽いが、大差はない。
- RaycastGrid/BoxcastGrid において、グリッドサイズが増えると Grid 更新のコストが増える・A ＊のノード数も増えるため、負荷が上がる。複数の Agent で Grid を共有する事で、コスト削減できるが、違うタイプの Agent（たとえば地面タイプと飛空タイプ）は Grid も違うので、別々の Grid を使う必要がある。

5.7　まとめ

Unity の Job/Burst で、マルチスレッドを利用できる経路探索を実装しました。Unity のデフォルト機能に比べて、このようなメリットとデメリットがあります：

「RaycastGrid/BoxcastGrid」のメリット：

- NavMesh はナビゲーション用メッシュをあらかじめベイクする必要があるが、RaycastGrid はその必要がない。毎フレームで Grid を更新すれば、ステージの構造が大きく変わっても対応できる。
- 素の経路探索も NavMesh よりは軽い。
- 元ネタとなる UE4 の EQS と同じように、歩ける・歩けない以外のマップ情報も原則上取得可能であるため、色々複雑な AI の実装にも使える。

「RaycastGrid/BoxcastGrid」のデメリット：

- RaycastGrid の作り方による制限で、2D/2.5D の環境しか対応できない（橋など重なってるステージは対応できない）が、NavMesh の方は 3 D 環境へ対応可能。
- RaycastGrid の経路探索範囲は Grid の範囲に制限されるため、大きいステージの

端から端への経路探索は一発ではできない。

「RaycastGrid/BoxcastGrid」は、2D/2.5D の経路探索で十分な場合、事前ベイクなし・より軽量な経路探索の選択肢として利用できます。

第 6 章

Python のマイナー文法の紹介

Shunsuke Ito / @fgshun

Python には便利な文法たちが存在しています。そのうち知名度が低いと私が勝手に判断したものやここ数年で追加されたばかりのものについて、紹介していきます。

6.1 for 文、 while 文の else 節

else 節は if 文の後に続くもの、という先入観があるのではないでしょうか。 Python ではループのための文の後にも else 節を書くことができます。

▼リスト 6.1 for 文の else 節

```
def find_7(L):
    for i in L:
        if i == 7:
            print('found 7')
            break
        else:
            print(f'{i} is not 7')
    else:
        print('not found 7')
```

この else 節はループを break することなく抜けた時にのみ実行されます。

▼リスト 6.2 else 節の動作

```
>>> find_7([2, 7, 1, 8, 2, 8])
2 is not 7
found 7
>>> find_7([3, 1, 4])
3 is not 7
1 is not 7
4 is not 7
not found 7
```

ループ中に条件を満たした時 break する処理を記述する際、条件をひとつも満たさなかった時の後処理をフラグ変数などを用意することなく記述することができます。

6.2 イテラブルのアンパック

次のような、複数の変数に複数の値を代入するコードを書いたことはあるでしょうか。

▼リスト 6.3 複数変数への複数の値の代入

```
a, b = 0, 1
```

一見、変数たちをカンマで区切り、イコール記号をまたいで値を同じ個数並べることで同時に代入する記法に見えます。しかし、実際には異なります。右辺は丸括弧が省略されたタプルです。

▼リスト 6.4 複数変数への複数の値の代入、と同等のコード

```
a, b = (0, 1)
```

右辺に適用できるものはタプルに限りません。イテラブルなオブジェクトであれば何であれ採用が可能です。個数が合わない時には ValueError となります。

▼リスト 6.5 複数変数への代入にイテラブルを用いる

```
a, b = [0, 1]  # リスト
a, b = open('spam.txt')  # ファイルオブジェクト
```

実行するまでは右辺の個数がわからない場合でも、左辺に星付きのターゲットを用いることで柔軟に対応することが可能です。星付きのターゲットには 0 個以上の値を格納したリストが代入されます。個数が足りない時には ValueError となります。

▼リスト 6.6 星付きのターゲット

```
# 先頭行を header へ。残りを body へ
header, *body = open('spam'.txt')
# 先頭行を header へ。最後の行を footer へ。途中を body へ
header, *body, footer = open('spam'.txt')
```

6.3 2 要素の assert - 説明をつける

assert 文はデバッグ用のチェック処理を記述できる文です。ところで、エラーを検知した時に送出される AssertionError がどのような理由で出力されたのかがわからなくなった経験はないでしょうか。

実は assert 文には 2 つめの要素があり、 AssertionError のコンストラクタに渡すためのオブジェクトを指定できます。ここにエラー発生の理由となるテキストを指定してお

くことで、スタックトレースにこれを表示することができます。

▼リスト 6.7　2 要素の assert 文

```
assert self.root, 'Missing root node.'
```

6.4　with 文で複数要素を使う

コンテキストマネージャが複数からむ処理を記述した際、 with 文がネストしてインデントが深くなってしまう問題があります。

▼リスト 6.8　ネストした with 文

```
with open('src.txt') as r:
    with open('dst.txt') as w:
        w.write(f.read())
```

実は with 文は複数要素をもつことができます。これによりインデントが深くなる問題を解消できます。

▼リスト 6.9　複数要素をもつ with 文

```
with open('src.txt') as r, open('dst.txt') as w:
    w.write(f.read())
```

この書き方をした際、 with 文そのものの文字数が増えてしまうという別の問題が起きてしまいます。これは Python 3.10 から[*1] 採用された複数行の with 文で解消できます。

▼リスト 6.10　複数行 with 文

```
with (
    open('src.txt') as r,
    open('dst.txt') as w,
):
    w.write(f.read())
```

6.5　代入式 :=

while 文や if 文の条件式で、 他の言語ならば代入しつつ評価できるのに、と感じたことはないでしょうか。

[*1] 正確には Python 3.9 の新パーサーから。 -X oldparser オプションをつけると旧パーサーが用いられます。旧パーサーでは複数行 with 文は動作しません。

▼リスト 6.11　代入は文であり式ではなかった

```
# while chunk = file.read(9000):  # シンタックスエラー
#     process(chunk)

chunk = file.read(9000)
while chunk:
    process(chunk)
    chunk = file.read(9000)
```

Python 3.8 からは代入式が追加されており、このようなわずらわしさは過去のものとなっています。

▼リスト 6.12　代入式

```
while chunk := file.read(9000):
    process(chunk)
```

代入式は乱用を防止する方向で実装されており、使える場所が制限されています[*2]。

▼リスト 6.13　代入式が禁止されている例

```
y := f(x)  # エラー
y = f(x)  # 通常の代入文でよい

y0 = y1 := f(x)  # エラー
y0 = y1 = f(x)  # やはり通常の代入文で良い

foo(x = y := f(x))  # エラー
y = f(x)  # キーワード引数に渡す値。分けて用意しておいてほしい
foo(x=y)

def foo(answer = p := 42)  # エラー
  ...
p = 42  # 仮引数の初期値。詰め込まずに分けて書いてほしい
def foo(answer=p)
  ...
```

6.6　match 文 - Python 流の switch 文

Python には switch, case 文に相当するものがなく、 if 文で代用してきていました。

▼リスト 6.14　if 文で他言語の switch 文相当の記述

```
if key == 'UP':
    ...
elif key == 'Down':
    ...
else:
    ...
```

[*2] 括弧で囲むことで強引に動かすことは可能です。たとえば y0=(y1:=f(x)) 。好ましい書き方ではありませんが。

Python 3.10 からは match 文が追加されています。

▼リスト 6.15　マッチ文

```
match key:
    case 'UP':
        ...
    case 'DOWN':
        ...
    case _:
        ...
```

case の後に記述できるパターンは一見すると式に見えますが、実は専用に定義された書式であり、既存の Python コードとはまったくの別物です。

▼リスト 6.16　case の後ろは通常の式ではない

```
a = 1
b = 2
match a:
    # case b.bit_count() - 1:
    #     シンタックスエラー。メソッド呼び出しや演算はできない

    case 1 | 2:
        # a が 1, 2 のいずれかであるか
        # ビット演算ではなく、複数の case を並べて or 条件でマッチする記法
        ...

    case ['spam', 'ham', 'eggs']:
        # a が長さ 3 のシーケンスかつ要素が一致するか
        # リストと比較するわけではなく、タプルでもマッチする
        ...

    case {'key': 'UP'}:
        # a['key'] が 'UP' であるか
        # 辞書と比較するわけではなく、一部の key, value がマッチすればよい
        ...

    case _:
        # a がなんであれマッチし、必ず実行されるパターン
        # _ 変数との比較でない
        ...
```

match 文でできることは値が定数と等しいことの確認に限りません。たとえばシーケンスパターンでは、等しいかを確認する代わりに値をキャプチャすることが可能で、要素数によって別のパターンにマッチさせることができます。

▼リスト 6.17　値のキャプチャ

```
commands = ['get', 'spam', 'ham', 'eggs']
match commands:
    case ['get']:  # 要素が 'get' 1 つのシーケンスにのみマッチ
        print('no target')
    case ['get', target]:  # 先頭が 'get' である 2 要素のシーケンスにマッチ
        print('single')
        # 2 つめの要素には target でアクセスできる
        print(target)
    case ['get', *targets]:  # 先頭が 'get' である 1 要素以上のシーケンスにマッチ
        print('multiple')
```

```
        # 2 つめ以降の要素には targets でアクセスできる
        for target in targets:
            print(target)
```

match 文は上から順に case を判定し、マッチした場合はそこで処理が終了します。このコード例では最後のケースは 1 要素のみの ['get'] や 2 要素ぴったりの commands にもマッチすることができるのですが、前段で拾われるため、そのような case が到達することはありません。つまり targets が長さ 0 や 1 になることはありません。

一癖ある match 文ではありますが、慣れるととても便利なものです。より詳しくは PEP 634[*3] を参照してください。

6.7 終わりに - Python ドキュメントに親しもう

この記事で紹介した文法たちは、公式ドキュメントの「Python 言語リファレンス[*4]」にまとめられています。 Python 言語そのものに興味を持った方、ぜひ公式ドキュメントに目を通してみてください。新たな気づきがあるかもしれません。

[*3] https://peps.python.org/pep-0634/
[*4] https://www.klab.com/jp/

第 7 章

React Concurrent Mode 完全に理解した（い）

Shinya Naganuma / @Pctg_x8

React 18 から正式に利用できるようになった **Concurrent Mode** の機能群について、最近触る機会が増えてきたもののいまひとつ理解が及んでないところがあると個人的に思うことがあります。そのため、本章では実際のコードを読むことでより一層理解を深めていくとともに、改めて Concurrent Mode について整理していこうと思います。

すでに Concurrent Mode についての詳細な解説文章は多数出ていますが、本章では **Suspense** と **Transition** に焦点を絞っていきます。

7.1 Concurrent Mode 理解のためのマインドモデル

Concurrent Mode の機能について書いていく前に、仕組みや要素について簡単に整理しておきます。

Concurrent Mode では、重くなりがちな Reconcilation/レンダリング処理を複数フレームに分散することでユーザー体験に悪影響を与えないような仕組みが働いています。また、各処理には優先度[*1]が設けられており、OS におけるマルチタスク管理のように処理をスケジューリングしています。

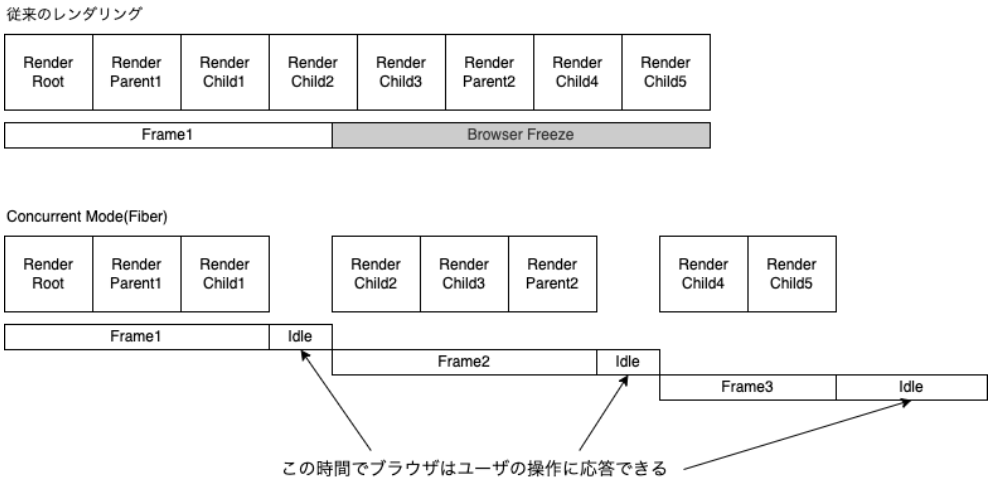
この優先度割り当ては React が内部で勝手に行ってくれるため、開発者が意識する必要はありません。

[*1] React では **Lane** と呼ばれる機構を使って低コストでスケジューリングの優先度管理をしています。Lane に関しては `https://jser.dev/react/2022/03/26/lanes-in-react.html` が詳しいです。

Fiber

React では **Fiber** という形で処理を管理しています。通常ひとつのコンポーネントには最大 2 つの Fiber が割り当てられており、ダブルバッファリングと同じような機構を構成しています。「レンダリングが完了した状態」と「途中の状態」をそれぞれ別でもつことで、レンダリング/ステートの巻き戻しといった操作が可能になっています[*2]。

また、処理を Fiber という形で細分化することで、一連の Reconcilation/レンダリングにタイムスライスの概念を取り込んだり中断の仕組みを取り込んだりして 1 フレームの時間を超えないように処理を分散することができるようになっています（図 7.1）。



▲図 7.1　Fiber による処理分散の例

7.2 Suspense

Concurrent Mode を象徴する機能のひとつが **Suspense** です。子コンポーネントのレンダリングで `Promise`が throw された場合に、その `Promise`が解決するまで `fallback` を表示する機能を持ちます。

リスト 7.1 に Suspense の使用例を示します。機能的にはリスト 7.2 と同じですが、Suspense を使った場合はより宣言的かつ簡潔な記述になります。

[*2] この仕組みを踏まえると、React のステートは不変でなければならないという理由がよりはっきりしますね。可変だと巻き戻し先のステートも変わってしまうためこの仕組みがうまく働かなくなってしまいます。

▼リスト 7.1　Suspense 使用例

```
// fetchItemDataResource: () => Resource<ItemData>; Resource 型については後述

function ItemInfoRes(props: {
  readonly item: Resource<ItemData>;
}): JSX.Element {
  // read: データ読み込み中なら Promise を throw する
  const data = props.item.read();

  return <article>...</article>;
}

function Component(): JSX.Element {
  const item = fetchItemDataResource();

  return <React.Suspense fallback={<p>Loading...</p>}>
    <ItemInfoRes item={item} />
  </React.Suspense>;
}
```

▼リスト 7.2　Suspense を使わない例

```
// fetchItemData: () => Promise<ItemData>;

function ItemInfo(props: { readonly item: ItemData }): JSX.Element {
  return <article>...</article>;
}

function Component(): JSX.Element {
  const [item, setItem] = React.useState<ItemData | undefined>(undefined);

  React.useEffect(() => {
    if (item === undefined) {
      fetchItemData().then(setItem);
    }
  }, [item]);

  return {
    item !== undefined ? <ItemInfo item={item} /> : <p>Loading...</p>
  };
}
```

Error Boundary

Promiseを throwするといういきなりとんでも概念が出てきましたが、JavaScript の throwは別に Errorクラスを継承したオブジェクトに限定していないので言語仕様上投げることは普通に可能です。

React 16 から **Error Boundary** と呼ばれるタイプのコンポーネントが追加されました。これは何かというと、「エラー境界」の直訳のとおり「子コンポーネントで発生したエラーを捕捉するコンポーネント」です。

React 16 では React.Componentに Error Boundary 用のコールバックを 2 つ追加しており、そのいずれかもしくはすべてを実装すると Error Boundary となります。Error Boundary の実装例をリスト 7.3 に示します。

Hooks では同等の機能をもつものは提供されていないので、Error Boundary は必ずクラスコンポーネントとなります。

▼リスト 7.3　Error Boundary の例

```
type ErrorBoundaryProps = {
  readonly fallback: JSX.Element;
  readonly children: JSX.Element;
};
type ErrorBoundaryState = {
  readonly hasErrored: boolean;
};
class ErrorBoundary extends React.Component<
  ErrorBoundaryProps,
  ErrorBoundaryState
> {
  constructor(props: ErrorBoundaryProps) {
    super(p);
    this.state = { hasErrored: false };
  }

  // Error Boundary: エラーを受け取って fallback UI をレンダリングするための State を返す
  static getDerivedStateFromError(_error: unknown): ErrorBoundaryState {
    return { hasErrored: true };
  }

  // Error Boundary: 子コンポーネントのレンダリング中にエラーが起きた際に呼ばれる
  componentDidCatch(_error: unknown, _errorInfo: unknown) {
    // do error logging etc...
  }

  render(): JSX.Element {
    if (this.state.hasErrored) {
      return this.props.fallback;
    }

    return this.props.children;
  }
}

function Component(): JSX.Element {
  return <ErrorBoundary fallback={<p>Errored!</p>}>
    <ItemInfo item={item} />
  </ErrorBoundary>;
}
```

これはちょうど React コンポーネントツリーにおける try-catch 構文に相当するものとなっています（実処理を Error Boundary のタグで囲うのと try-catch で囲うのとで見た目が似ていますね）。Error Boundary のおかげで、宣言的な形を保ちつつエラー処理ができるようになります。

ここで、Error Boundary により子コンポーネントの `throw`を拾うことができるようになったので、当然 `throw`された `Promise`も同じ仕組みで拾うことが可能です。`Promise`はのちに解決されるなど通常の `Error`との細かな違いから内部では若干特殊な実装になっていますが、基本的な仕組みは同じになっています。

Promise の特殊処理について

このコラムの内容は 2022/08/05 現在の `main`（`facebook/react@b4204ede66284e7153ffb11fd434cd9b9a64a56f`）に基づく内容となっています

`Promise`のようなオブジェクトは `react-reconciler`の内部では **Wakeable** として特別扱いされており、コンポーネントレンダリング中の `throw`を処理するひとつの関数内で分岐して処理されています。
具体的には、`react-reconciler/src/ReactFiberThrow.new.js`の 354 行目にある `throwException`がレンダリング中に投げられた例外やサスペンドを処理しています。この関数はレンダリング処理のエントリポイントである `renderRootConcurrent`/`renderRootSync`（in `react-reconciler/src/ReactFiberWorkLoop.new.js`）から、処理中の例外を `catch`したハンドラ内の `handleError`の呼び出しを経由して呼ばれています。
`throwException`の中で、371 行目の `if`文が、投げられた値が `Promise`かどうかを判定して分岐している箇所になります。判定文は `Promise`を厳密に判定しているというよりは「有効な `object`であり、`then`という関数型のメンバをもつ」というかなり緩い判定でとっています（満たす条件としては TypeScript の `PromiseLike`に近いです）。そのため、実は `Promise`でなくても `then`を定義したオブジェクトであればなんでもサスペンドを発生させることができます（有用な使い方があるかはさておき）。

Resource（Fetcher）

Concurrent Mode におけるレンダリングの中断機能を使うには「ロード中であれば `Promise`を `throw`し、ロードが完了していればは値を返す」といった処理が必要になります。たとえば状態管理ライブラリとして Recoil を使用している場合は `useRecoilValue`がこの処理を有していますが、React のみの場合は用意されていないのでリスト 7.4 のようなラッパークラスを自作することが多いです。

▼リスト 7.4　ラッパークラスの例

```
type ResourceStatePending = {
  readonly state: "Pending";
  readonly operation: Promise<void>;
};
type ResourceStateFulfilled<T> = {
  readonly state: "Fulfilled";
  readonly value: T;
};
type ResourceStateRejected = {
  readonly state: "Rejected";
  readonly error: unknown;
};
type ResourceState<T> =
  | ResourceStatePending
  | ResourceStateFulfilled<T>
  | ResourceStateRejected;

class Resource<T> {
  private state: ResourceState<T>;

  constructor(pendingOperation: () => Promise<T>);
  constructor(pendingOperation: Promise<T>);
  constructor(value: T);
  constructor(pendingOperation: T | Promise<T> | (() => Promise<T>)) {
    this.state =
      pendingOperation instanceof Function
        ? {
            state: "Pending",
            operation: pendingOperation().then(
              (x) => {
                this.state = { state: "Fulfilled", value: x };
              },
              (e) => {
                this.state = { state: "Rejected", error: e };
              }
            ),
          }
        : pendingOperation instanceof Promise
        ? {
            state: "Pending",
            operation: pendingOperation.then(
              (x) => {
                this.state = { state: "Fulfilled", value: x };
              },
              (e) => {
                this.state = { state: "Rejected", error: e };
              }
            ),
          }
        : { state: "Fulfilled", value: pendingOperation };
  }

  read(): T {
    switch (this.state.state) {
      case "Pending":
        throw this.state.operation;
      case "Fulfilled":
        return this.state.value;
      case "Rejected":
        throw this.state.error;
    }
  }
}
```

Promiseが解決されたら stateを「解決済み」のものに設定するようにしておき、read が呼ばれたタイミングでの stateの値によって throwするか returnするかを分岐しています。

レンダリングの再開時には Promiseの解決値は利用されず、Promiseを throwしたコンポーネントのレンダリング処理が再度 1 から走るようになります。そのため、readは次の 2 つのタイミングでそれぞれ最低 1 回は呼ばれることに注意してください。

- 初回のレンダリング
- Promise解決後のレンダリング

7.3 Transition

Transition は、ステートの変更によって引き起こされるレンダリングが完了するまでの間をうまく表示するための機能を提供します。Transition を使用して、ロード中はボタンを非活性化させる例をリスト 7.5 に示します。

▼リスト 7.5　Transition の使用例

```
// fetchItemDataResource: (id: number) => Resource<ItemData>;

function Component(): JSX.Element {
  const [itemData, setItemData] = React.useState<Resource<ItemData>>(
    fetchItemDataResource(0)
  );
  const [itemLoading, startLoadItem] = React.useTransition();
  const switchItem = React.useCallback(
    (id: number) => {
      startLoadItem(() => {
        setItemData(fetchItemDataResource(id));
      });
    },
    [startLoadItem]
  );

  return <section>
    <ItemData item={itemData} />
    <p>{itemLoading ? "Loading..." : ""}</p>
    <button onClick={() => switchItem(0)} disabled={itemLoading}>Show: ID 0</button>
    <button onClick={() => switchItem(1)} disabled={itemLoading}>Show: ID 1</button>
    <button onClick={() => switchItem(2)} disabled={itemLoading}>Show: ID 2</button>
    <button onClick={() => switchItem(3)} disabled={itemLoading}>Show: ID 3</button>
  </section>;
}
```

Suspense は比較的理解しやすい挙動をしますが、Transition のほうは「レンダリングステートが同時に複数ある」という概念をうまく理解できていないと理解するのが難しい挙動をします。

startTransitionに渡した関数でステートの変更が発生した場合は通常と同じように更新がかかります。ただし、この更新のレンダリング中にサスペンドが発生した場合は、いったんステートの更新をなかったことにして再度レンダリングを行うようになります。この再レンダリングでは startTransition前のステートをそのまま引き継ぎますが、一部だけ例外があり、useTransitionの返り値の 1 番目の要素が trueとなります。サスペンドが解消されると更新後のステートでレンダリングを再開します。

7.4 おわりに

ここまで見てきたように、Concurrent Mode の諸機能を使うことで UI 構築から非同期にまつわる処理を排除することができ、非同期的だろうと同期的だろうと同じく「どうやって表示するか」を宣言し、表示したいものをセットするだけという統一感のある記述が行えるようになります。

こうしてみると一見いいところだらけな Concurrent Mode ですが、`throw`という大域脱出を行う言語機能を応用しているため、処理の合成のしやすさといった点で若干難点があります。2 つ以上の非同期な値を 1 つのコンポーネント内で利用する場合は、あらかじめ内部の `Promise`を取り出して目的の形に別途合成する必要があります。

Concurrent Mode の「レンダリング処理を中断する」という挙動はなかなか掴みづらいものですが、慣れて使いこなすことができるようになれば簡潔かつバグの少ない UI が記述できるようになります。本章が Concurrent Mode の理解の一端にでもなれば幸いです。

7.5 Appendix

- `https://github.com/facebook/react`
- `https://github.com/acdlite/react-fiber-architecture`
- `https://jser.dev/react/2022/03/26/lanes-in-react.html`

第 8 章

ベジエ単体フィッティングで多目的最適化の解を近似する

Naoki Hamada

食欲の秋——みんなでご飯を食べるとき，お店をどうやって選びましょうか？ 予算の範囲内で，なるべく全員の好みにマッチしたお店を探しますよね．このように，与えられた制約のもとで，複数の相反する評価をできるだけ良くする答えを探す問題を**多目的最適化**といいます．本章では，ある種の多目的最適化問題の解をとても効率よく近似することができる**ベジエ単体フィッティング**という手法を紹介します．それを実装した Python パッケージの使い方を説明し，統計学のモデル選択の問題に応用して実データで検証します．食べたいご飯の意見が割れたときには，颯爽と PC を取り出してベジエ単体をフィッティングしてみましょう．みんなの注目を集めることは間違いありません！

本章の内容は文献 “Yusuke Mizota, Naoki Hamada, Shunsuke Ichiki (2021), All unconstrained strongly convex problems are weakly simplicial, arXiv:2106.12704”[*1]に基づいています．定理の証明や実験の詳細について興味があればご参照ください．

本章の図はすべてカラーで作成しています．本文中にモノクロ印刷ではわかりにくい説明が含まれておりますので，電子版でご覧いただくことをおすすめします．

8.1 多目的最適化問題

多目的最適化は世の中の様々な場面で使われています．一例を挙げると次のようなものがあります．

[*1] https://doi.org/10.48550/arXiv.2106.12704

- **自動車設計**：なるべく共通の部品を使いつつ，重量を軽くしたい.[*2]
- **月面探査**：なるべく日照時間が長く，通信しやすく，平らな場所に着陸したい.[*3]
- **風力発電**：なるべく発電量が多く，コストが低く，騒音が小さく，耐久性が高い風車を設計したい.[*4]
- **ゲーム開発**：なるべく偏りが少なく，理不尽な出目が少ない乱数を作りたい.[*5]
- **行政施策**：なるべく多くの困窮者を助けつつも公平でありたい.[*6]

これらの問題はみな，与えられた複数の関数の値を同時に小さくする問題として定式化されます.[*7]まずは次の簡単な例を考えてみましょう.

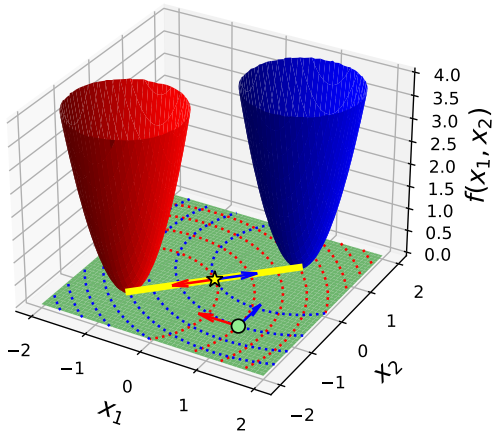
$$
\begin{aligned}
&\underset{(x_1,x_2)\in X}{\text{minimize}}\, f(x_1,x_2) = (f_1(x_1,x_2), f_2(x_1,x_2)),\\
&\quad \text{where } f_1(x_1,x_2) = (2x_1+2)^2 + (2x_2+2)^2,\\
&\qquad\quad f_2(x_1,x_2) = (2x_1-2)^2 + (2x_2-2)^2,\\
&\qquad\qquad X = \left\{ (x_1,x_2) \in \mathbb{R}^2 \;\middle|\; \forall n = 1,2 : -2 \le x_n \le 2 \right\}.
\end{aligned}
$$

この例題の定義域を図 8.1 に示します．最適化すべき関数 f_1 と f_2 はそれぞれ目的関数とよばれ，図では赤と青のグラフで表しています．解を探す範囲 X は実行可能領域とよばれ，図では緑の領域で表しています．もし f_1 だけを小さくする問題であったなら，赤いグラフの谷底の点 $(x_1,x_2) = (-1,-1)$ が最適でしょう．一方でもし f_2 だけを小さくしたいなら，青いグラフの谷底の点 $(x_1,x_2) = (1,1)$ が最適でしょう．2 つの点は異なるため，一方を最適化すると他方が最適化できないというトレードオフが生じています．そこで,「最適」の意味を考え直して，f_1 と f_2 の値を同時にこれ以上小さくできない点（一方を小さくすると他方が大きくなる点）を求めることにします．そのような点はパレート最適であるといいます．図の赤い矢印は f_1 を小さくする方向，青い矢印は f_2 を小さくする方向を表しています．図の○で示した点では赤い矢印と青い矢印を足し合わせた方向に進むことで f_1 と f_2 をともに小さくすることができるため，この点はパレート最適ではありません．図の☆で示した点では赤い矢印と青い矢印は正反対を向いており，f_1 と f_2 を同時に小さくする方向は存在しないため，この点はパレート最適です．パレート最適な点は 1 つだけではなく，黄色い線上の点ではすべて上記のトレードオフが生じています．そのような点全体をパレート集合とよび，$X^*(f)$ で表します.

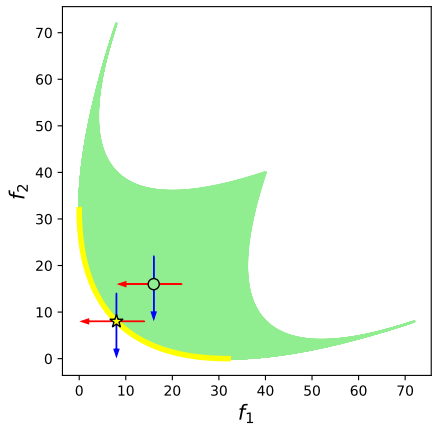
この例題の値域を図 8.2 に示します．緑の領域は実行可能領域の像 $f(X)$ です．黄色い線はパレート集合の像 $f(X^*(f))$ で，パレートフロントとよばれます．ある点がパレート最適かどうかは，こちらの空間でみたほうがわかりやすいかもしれません．図の○で示

[*2] http://is-csse-muroran.sakura.ne.jp/ec2017/EC2017compe.html
[*3] http://www.jpnsec.org/files/competition2018/EC-Symposium-2018-Competition.html
[*4] http://www.jpnsec.org/files/competition2019/EC-Symposium-2019-Competition.html
[*5] https://ec-comp.jpnsec.org/ja/competitions/eccomp2020
[*6] https://ec-comp.jpnsec.org/ja/competitions/eccomp2021
[*7] 最大化問題は関数に -1 をかければ最小化問題にできるため，本章では最小化問題に統一して考えます.

した点（図 8.1 の◯で示した点の像）では，点よりも左下に緑の領域が続いています．つまり，f_1 と f_2 の両方がより小さい値をとる点があるため，この点はパレート最適ではないことがわかります．一方，図の☆で示した点（図 8.1 の☆で示した点の像）では，点よりも左下に緑の領域がありません．つまり，f_1 と f_2 の両方がより小さい値をとる点はないため，この点はパレート最適であることがわかります．



▲図 8.1　例題の定義域



▲図 8.2　例題の値域

さらに，パレート集合とパレートフロントを合わせたものをパレートグラフとよび，$G^*(f)$ で表します．パレートグラフは図 8.1 と図 8.2 の直積をとった 4 次元空間に位置するので図示できませんが，とても重要な存在です．なぜなら，パレートグラフを定義域や値域へと射影することで，パレート集合やパレートフロントを取り出すことができるからです．言い換えれば，すべてのパレート最適解とその評価値についての情報をもった集合ですので，これを求めることができれば多目的最適化問題が解けたといえるでしょう．

以上を一般化して多目的最適化問題を定義します．M 以下の自然数の集合を $[M] = \{1, \ldots, M\}$ と表します．N 次元ユークリッド空間を $\mathbb{R}^N$ で表し，X をその部分集合とします．写像 $f = (f_1, \ldots, f_M) : X \to \mathbb{R}^M$ が与えられたとき，X の**パレート順序**を次のように定義します．

$$x \prec_f x' \Leftrightarrow (\forall m \in [M] : f_m(x) \leq f_m(x')) \wedge (\exists m \in [M] : f_m(x) < f_m(x')).$$

ここで x, x' は X の任意の 2 点です．この順序のもとでの最適解の集合

$$X^*(f) = \{\, x \in X \mid \nexists x' \in X : x' \prec_f x \,\}$$

を f の**パレート集合**といいます．その f による像

$$f(X^*(f)) = \left\{\, f(x) \in \mathbb{R}^M \,\middle|\, x \in X^*(f) \,\right\}$$

を f の**パレートフロント**といいます．さらに，f のパレート集合への制限写像 $f|_{X^*(f)} : X^*(f) \to \mathbb{R}^M$ のグラフ

$$G^*(f) = \left\{\, (x, f(x)) \in \mathbb{R}^N \times \mathbb{R}^M \,\middle|\, x \in X^*(f) \,\right\}$$

を f の**パレートグラフ**といいます．写像 f のパレートグラフを求める問題を N **変数** M **目的最適化問題**といい，

$$\underset{x \in X}{\text{minimize}}\, f(x) = (f_1(x), \ldots, f_M(x))$$

で表します．任意の空でない部分集合 $I = \{\, i_1, \ldots, i_k \,\} \subseteq [M]$ $(i_1 < \cdots < i_k)$ に対して，

$$f_I = (f_{i_1}, \ldots, f_{i_k}) : X \to \mathbb{R}^k$$

とおき，写像 f_I のパレートグラフ $G^*(f_I)$ を求める問題を**部分問題**といいます．

弱単体的な問題

以降の節ではベジエ単体を使ってパレートグラフを近似する方法を紹介しますが，その方法はどんな問題に対してもうまくいくわけではありません．ここでは，ベジエ単体に

よってパレートグラフを近似できることが理論的に保証された問題クラスとして，弱単体的な問題を説明します．

$\mathbb{R}^N$ の部分集合 X が M 次元**角付き** C^r **多様体**であるとは， X の各点 x に対してある近傍 U が存在して

$$\mathbb{R}^M_{\geq 0} = \left\{ (y_1, \ldots, y_M) \in \mathbb{R}^M \;\middle|\; \forall m \in [M] : y_m \geq 0 \right\}$$

のある点の近傍 V への C^r 微分同相写像 $\phi : U \to V$ が存在することをいいます．[*8]ここで， $\phi(x)$ の 1 つ以上の成分が 0 であるとき， x を X の**角**といいます．X のすべての点が角でないとき， X を**角のない多様体**といいます．

次の集合を $(M-1)$ 次元**標準単体**といいます．

$$\Delta^{M-1} = \left\{ (w_1, \ldots, w_M) \in \mathbb{R}^M \;\middle|\; \sum_{m=1}^{M} w_m = 1, \forall m \in [M] : w_m \geq 0 \right\}.$$

任意の空でない部分集合 $I \subseteq [M]$ について， Δ^{M-1} の**面**を次のように定義します．

$$\Delta_I = \left\{ (w_1, \ldots, w_M) \in \Delta^{M-1} \;\middle|\; \forall m \notin I : w_m = 0 \right\}.$$

単体とその面はそれぞれ $\mathbb{R}^M$ における $(|I|-1)$ 次元角付き C^∞ 多様体になっています．

本章を通して， r は非負正数または $r = \infty$ とします．（角付き，あるいは角のない）C^r 多様体 W と， $\mathbb{R}^L$ の部分集合 V について，写像 $g : W \to V$ が C^r **写像**（ C^r **微分同相写像**）であるとは， $g : W \to \mathbb{R}^L$ が C^r 写像である（ $g : W \to \mathbb{R}^L$ が C^r はめ込みでありかつ $g : W \to V$ が同相写像である）ことをいいます．本論文では上記の定義における C^0 写像と C^0 微分同相写像のことを**連続写像**と**同相写像**といいます．

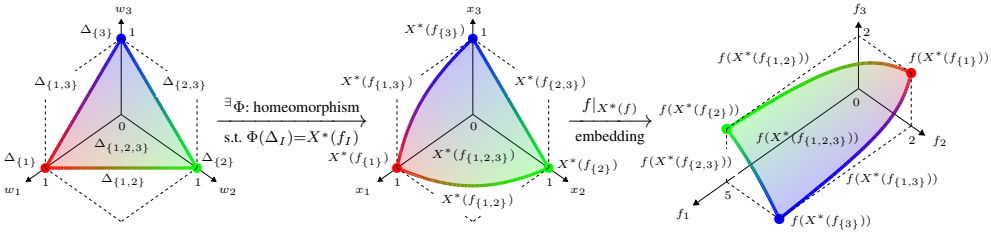
以上を用いて弱単体的な問題を定義します．

> **定義（弱単体的な問題）** X を $\mathbb{R}^N$ の部分集合とし， $f = (f_1, \ldots, f_M) : X \to \mathbb{R}^M$ を任意の写像とします． f を最小化する問題が C^r **単体的**であるとは， ある C^r 写像 $\Phi : \Delta^{M-1} \to X^*(f)$ が存在して， $[M]$ の任意の空でない部分集合 I について， 2 つの写像 $\Phi|_{\Delta_I} : \Delta_I \to X^*(f_I)$ と $f|_{X^*(f_I)} : X^*(f_I) \to f(X^*(f_I))$ が C^r 微分同相写像であることをいいます． ここで， $0 \leq r \leq \infty$ です．
> f を最小化する問題が C^r **弱単体的**であるとは，ある C^r 写像 $\phi : \Delta^{M-1} \to X^*(f)$ が存在して， $[M]$ の任意の空でない部分集合 I に対して $\phi(\Delta_I) = X^*(f_I)$ を満たすことをいいます． ここで， $0 \leq r \leq \infty$ です．

図 8.3 のように， C^r 単体的な問題のパレート集合とパレートフロントは単体を曲げた形状の角付き C^r 多様体をなします．図示することはできませんが，パレートグラフも単体を曲げた形状の角付き C^r 多様体をなします．パレート集合（パレートフロント，パ

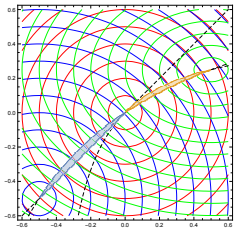
[*8] 定義を簡潔にするため，多様体はユークリッド空間に埋め込まれていると仮定しました．すべての多様体は（十分に次元の高い）ユークリッド空間に埋め込めるため，この定義は教科書的な定義と同値です．

レートグラフ）の角には単体の面と同じ入れ子構造が入っています．単体の面は，頂点のサブセットで張られる単体です．パレート集合（パレートフロント，パレートグラフ）の角は，関数のサブセットを最適化する部分問題のパレート集合（パレートフロント，パレートグラフ）です．ただし，単体的な問題では，パレート集合（パレートフロント，パレートグラフ）は退化する[*9]ことができません．一方で，弱単体的な問題は同様の構造をもちつつも図 8.4 のように退化することを許したものです．パレート集合が原点の近傍で双葉型に退化していることが見てとれます．



(a) Simplex Δ^2. (b) Pareto set $X^*(f)$. (c) Pareto front $f(X^*(f))$.

▲図 8.3　単体的な問題の例



▲図 8.4　弱単体的な問題の例

強凸問題

弱単体的な問題は（最適化の専門家にさえ）聞き慣れない問題クラスだと思います．最適化において良く知られた問題クラスとはどんな関係があるのでしょうか？ 結論からいうと，すべての無制約強凸最適化問題は弱単体的です．強凸最適化は実用上よく登場する問題クラスで，たとえば何らかの目標値との二乗誤差を最小化する問題（やそれに凸正則化を加えた問題）はすべて強凸になります．したがって，ベジエ単体によるパレートグラフの近似は，実用的な幅広い問題でうまくいくことがわかります．以下ではこのことを示します．

$\mathbb{R}^N$ の部分集合 X が**凸**であるとは，すべての $x, y \in X$ と $t \in [0,1]$ に対して $tx + (1-$

[*9] $(M-1)$ 次元角付き C^r 多様体にならない，すなわち，いずれかの点の近傍が $\mathbb{R}^{M-1}_{\geq 0}$ のいかなる点の近傍とも C^r 微分同相でないことをいいます．

$t)y \in X$ を満たすことをいいます．X を $\mathbb{R}^N$ の凸集合とします．関数 $f : X \to \mathbb{R}$ が**強凸**であるとは，すべての $x, y \in X$ と $t \in [0, 1]$ に対して

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y) - \frac{1}{2}\alpha t(1-t)\|x-y\|^2$$

を満たす $\alpha > 0$ が存在することをいいます．ここで，$\|z\|$ は $z \in \mathbb{R}^N$ のユークリッドノルムです．写像 $f = (f_1, \ldots, f_M) : X \to \mathbb{R}^M$ が**強凸**であるとは，すべての $m \in [M]$ について関数 f_m が強凸であることをいいます．C^r 強凸写像を最小化する問題を C^r **強凸問題**とよびます．

> **定理（強凸最適化問題は弱単体的）** 写像 $f : \mathbb{R}^N \to \mathbb{R}^M$ を任意の C^r 強凸写像とします $(0 \leq r \leq \infty)$．このとき，f を最小化する問題は，$r = \infty$ ならば C^∞ 弱単体的，$0 < r < \infty$ ならば C^{r-1} 弱単体的，$r = 0$ ならば C^0 弱単体的です．加えて，弱単体的な問題の条件にある C^r 写像 $\phi : \Delta^{M-1} \to \mathbb{R}^N$ は，$\phi(w) = \arg\min_{x \in \mathbb{R}^N} \sum_{m=1}^M w_m f_m(x)$ で与えられます．ここで，関数 $h : \mathbb{R}^N \to \mathbb{R}$ が唯一の最小点 $x^* \in X^*(h)$ をもつとき，$\arg\min_{x \in \mathbb{R}^N} h(x)$ は最小点 x^* を表します．

上記の定理において，いかなる強凸最適化問題に対しても写像 ϕ が一定の形で与えられていることに注目してください．以降の節で述べるように，写像 ϕ を用いて単体からパレートグラフへの写像を構成し，その写像をベジエ単体で近似することで，いかなる強凸最適化問題に対しても統一されたアプローチでパレートグラフを（ベジエ単体の像として）近似することができるわけです．

8.2 Elastic Net

本節では，先述の定理を統計学のモデル選択の問題に応用します．具体的には，Elastic net というスパースモデリングの手法のハイパーパラメータ選択を効率化します．Elastic net のための従来のハイパーパラメータ選択では，2 つのハイパーパラメータのうち 1 つを固定して，もう 1 つのハイパーパラメータを探索します．ハイパーパラメータの値を少しずつ変えながら繰り返しモデルを訓練して，訓練済みモデルの 1 パラメータ族を計算します．このモデル族は**解パス**とよばれ，計算した 1 つの解パスの中からユーザーにとってもっとも好ましいモデルを選びます．これを安直に 2 パラメータ族（本章では**解写像**とよびます）に拡張するならば，固定していたパラメータも少しずつ動かして多数の解パスを計算することになります．しかし，Elastic net の訓練を 2 パラメータに対して繰り返すと膨大な計算コストがかかります．

今から試みるのは，この解写像を少数の訓練済みモデルから近似することです．はじめに，Elastic net を紹介し，その 2 次元ハイパーパラメータ選択問題を 3 目的 C^0 強凸最適化問題へと定式化しなおします．前節の定理からこの C^0 強凸問題は C^0 弱単体的であり，したがって解写像は単体からユークリッド空間への連続写像であることがわかりま

す．3 目的問題のそれぞれの部分問題の構造が解写像にどう反映されるかを簡単な例を通して本節で示します．

なお，次節以降では，この解写像を（部分問題の構造まで含めて）ベジエ単体で近似できることを示します．ベジエ単体の普遍近似定理を示し，ベジエ単体のフィッティングアルゴリズムを紹介します．実験を通して，実データにおける解写像の近似精度を示し，ハイパーパラメータ選択がどれくらい高速化するかを議論します．

Elastic net の多目的最適化としての定式化

Elastic net は次の線形回帰モデルを扱います．

$$y = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_N x_N + \zeta.$$

ここで x_n と θ_n $(n = 1, \ldots, N)$ は説明変数とその係数で， y は予測したい応答変数， ζ はガウスノイズです．行列 X は M 行の観測と N 列の説明変数をもち，行ベクトル y は M 個の応答変数をもつとします．Elastic net 回帰は次の最適化問題の解になります．

$$\underset{\theta \in \mathbb{R}^N}{\text{minimize}}\, g_{\mu,\lambda}(\theta) = \frac{1}{2M}\|X\theta - y\|^2 + \mu|\theta| + \frac{\lambda}{2}\|\theta\|^2.$$

ここで $\|\cdot\|$ は L_2 ノルム， $|\cdot|$ は L_1 ノルム， μ と λ は正則化のための固定された非負実数です．もし $\mu = \lambda = 0$ と設定すると，上記の最適化問題は最小二乗法（Ordinary Least Squares; OLS）による回帰問題になります．また， $\mu > 0$ かつ $\lambda = 0$ と設定すると，上記の問題は Lasso 回帰問題となり，影響力の小さい説明変数を削除したスパースな解を求めることになります．さらに， $\mu = 0$ かつ $\lambda > 0$ と設定すると，上記の問題は Ridge 回帰となり，多重共線性のある説明変数に対して安定な解を求めることになります．したがって， $\mu > 0$ かつ $\lambda > 0$ と設定する Elastic net 回帰は，Lasso と Ridge の両方の性質を受け継ぎます．ハイパーパラメータ μ と λ の適切な値を選ぶことは，無制約の 2 次元ブラックボックス最適化となり，しばしば多大な計算コストを要します．

このような高コストなハイパーパラメータ探索を避けるために，問題を多目的強凸最適化に定式化しなおします．そして，その加重和スカラー化問題を考え，解写像の近似を作ります．こうすることで，もとのハイパーパラメータ探索よりも少数のモデルを訓練するだけですむようになります．こうして得られた近似曲面上でモデルを比較し，もっとも好ましいハイパーパラメータを選びます．

これを行うために，まず Elastic net 回帰問題の OLS 項， L_1 正則化項， L_2 正則化項をそれぞれ個別の関数とみなします．

$$f_1(\theta) = \frac{1}{2M}\|X\theta - y\|^2, \quad f_2(\theta) = |\theta|, \quad f_3(\theta) = \frac{1}{2}\|\theta\|^2.$$

関数 f_1 と f_2 は凸ですが，必ずしも強凸ではありません．一方で，任意の凸関数と任意の強凸関数の和は強凸関数になります．そこで，強凸関数 f_3 の微小倍を 2 つの関数に加

えることにより，それらを強凸関数にします．

$$\begin{aligned}\underset{\theta\in\mathbb{R}^N}{\text{minimize}}\ \tilde{f}(\theta) &= (\tilde{f}_1(\theta), \tilde{f}_2(\theta), \tilde{f}_3(\theta))\\ \text{where } \tilde{f}_i(\theta) &= f_i(\theta) + \varepsilon f_3(\theta) \quad (i = 1, 2, 3).\end{aligned}$$

ここで， ε は正の実数です．したがって，写像 $\tilde{f}$ は連続ですが微分不可能な強凸関数になります．

今から上記の問題のパレートグラフを求める方法を考えます．強凸問題に対する加重和最小化問題

$$\underset{\theta\in\mathbb{R}^N}{\text{minimize}}\, h_w(\theta) = w_1\tilde{f}_1(\theta) + w_2\tilde{f}_2(\theta) + w_3\tilde{f}_3(\theta)$$

は任意の重み $w = (w_1, w_2, w_3) \in \Delta^2$ に対して唯一の最適解をもちます．この解を $\arg\min_{\theta\in\mathbb{R}^N} h_w(\theta)$ で表します．先述の定理より，連続全射 $\theta^* : \Delta^2 \to X^*(\tilde{f})$ を次のように定義することができます．

$$\theta^*(w) = \arg\min_{\theta\in\mathbb{R}^N} h_w(\theta).$$

写像 $\tilde{f} : \mathbb{R}^N \to \mathbb{R}^3$ は連続なので，写像

$$\tilde{f} \circ \theta^* : \Delta^2 \to \tilde{f}(X^*(\tilde{f}))$$

もまた連続全射になります．ここで，空間 $\tilde{f}(X^*(\tilde{f}))$ の位相は $\mathbb{R}^3$ からの誘導位相としました．$\tilde{f}$ のパレートグラフは

$$G^*(\tilde{f}) = \{\, (\theta, \tilde{f}(\theta)) \in \mathbb{R}^{N+3} \mid \theta \in X^*(\tilde{f}) \,\}$$

であり，解写像

$$(\theta^*, \tilde{f} \circ \theta^*) : \Delta^2 \to G^*(\tilde{f}),$$

もまた連続全射です．ここで，空間 $G^*(\tilde{f})$ の位相は $\mathbb{R}^{N+3}$ からの誘導位相としました．$\emptyset \neq I \subseteq \{1, 2, 3\}$ を満たすすべての I に対して $\theta^*(\Delta_I^2) = X^*(\tilde{f}_I)$ が成り立つので，解写像はすべての部分問題のパレートグラフ（したがってパレート集合とパレートフロント）の情報を含んでいます．すなわち， $\emptyset \neq I \subseteq \{1, 2, 3\}$ を満たすすべての I に対して，写像

$$\begin{aligned}\theta^*|_{\Delta_I^2} &: \Delta_I^2 \to X^*(\tilde{f}_I),\\ \tilde{f}_I \circ \theta^*|_{\Delta_I^2} &: \Delta_I^2 \to \tilde{f}_I(X^*(\tilde{f}_I))\end{aligned}$$

は連続全射です．ここで，空間 $\Delta_I^2, X^*(\tilde{f}_I), \tilde{f}_I(X^*(\tilde{f}_I))$ の位相はそれぞれ $\Delta^2, \mathbb{R}^N, \mathbb{R}^{|I|}$ からの誘導位相としました．

任意の $w = (w_1, w_2, w_3) \in \Delta^2 \setminus \Delta^2_{\{2,3\}}$ に対して，点 $\theta^*(w)$ は関数 $g_{\mu(w),\lambda(w)}$ の最小点であることに注意してください．ここで，

$$\mu(w) = \frac{w_2}{w_1},$$
$$\lambda(w) = \frac{w_3 + \varepsilon}{w_1}$$

であり， ε は先述のものです．特に，任意の $I = \{1\}, \{1,2\}, \{1,3\}$ に対して写像 $(\theta^*, \tilde{f}_I \circ \theta^*)|_{\Delta^2_I} : \Delta^2_I \to G^*(\tilde{f}_I)$ の像はそれぞれ，OLS の解，Lasso の解パス，Ridge の解パスを近似します[*10]．なお，上記の $\mu(w), \lambda(w)$ の式は，単目的版と多目的版の Elastic net の定義を見比べることで簡単に得られます．

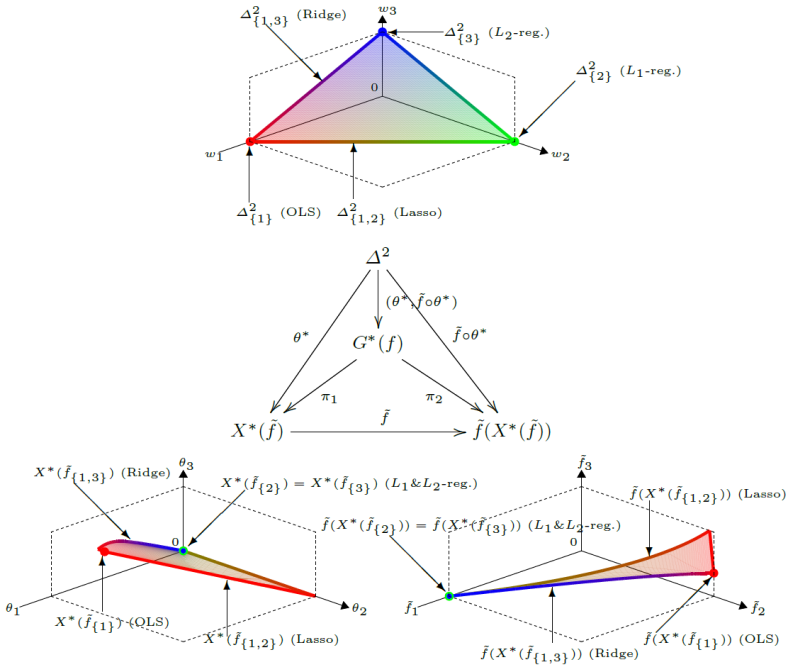
Elastic Net の解写像

Elastic net の解写像がどんな構造をもっているか簡単な例を使ってみてみましょう．次のデータに対する Elastic net の解写像を図 8.5 に示します．

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \\ 12 & 11 & 10 \end{pmatrix}, \quad y = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}.$$

重み w を単体の頂点 $\Delta^2_{\{1\}} = \{(1,0,0)\}$ から選べば，写像 $\tilde{f}_{\{1\}}$ を最適化する部分問題のパレートグラフである OLS 解（図の赤い点）が得られます．重み w を単体の辺 $\Delta^2_{\{1,2\}} = \{(t, 1-t, 0) \mid 0 \le t \le 1\}$ から選べば，写像 $\tilde{f}_{\{1,2\}}$ を最適化する部分問題のパレートグラフである Lasso の解パス（図の赤から緑へと変化する曲線）が得られます．重み w を単体の辺 $\Delta^2_{\{1,3\}} = \{(t, 0, 1-t) \mid 0 \le t \le 1\}$ から選べば，写像 $\tilde{f}_{\{1,3\}}$ を最適化する部分問題のパレートグラフである Ridge の解パス（図の赤から青へと変化する曲線）が得られます．

[*10] この像はオリジナルの Elastic net の解とは厳密には一致しません．なぜなら，多目的版 Elastic net を強凸にするために ε を導入したからです．



▲図 8.5　Elastic net の解写像

8.3　ベジエ単体

Elastic net の解写像は単体からユークリッド空間への連続写像でした．本節では，これをベジエ単体で近似する方法を紹介します．

ベジエ曲線を高次元に一般化したものが**ベジエ単体**です．非負整数の集合を $\mathbb{N}$ で表し，

$$\mathbb{N}_D^M = \left\{ (d_1, \dots, d_M) \in \mathbb{N}^M \;\middle|\; \sum_{m=1}^{M} d_m = D \right\}$$

とします．ユークリッド空間 $\mathbb{R}^N$ における次数 D の $(M-1)$ 次元ベジエ単体とは，**制御点** $p_d \in \mathbb{R}^N$ $(d \in \mathbb{N}_D^M)$ で指定される次の多項式写像 $b : \Delta^{M-1} \to \mathbb{R}^N$ です．

$$b(w) = \sum_{d \in \mathbb{N}_D^M} \binom{D}{d} w^d p_d.$$

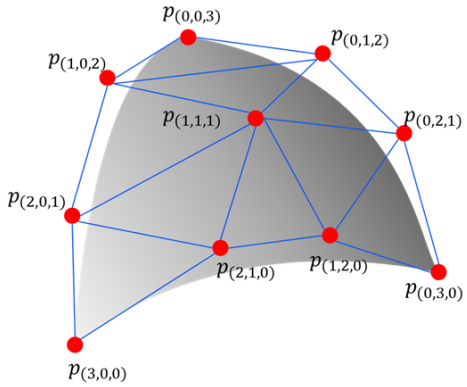
ここで，

$$\binom{D}{d} = \frac{D!}{d_1! d_2! \cdots d_M!}$$

は多項係数で，$w^d = w_1^{d_1} w_2^{d_2} \cdots w_M^{d_M}$ は多重指数です．

図 8.6 は $M = 3, D = 3$ のベジエ単体の例です．このベジエ単体は 10 個の制御点 $p_{(3,0,0)}, \dots, p_{(0,0,3)}$ をもちます．灰色の面はベジエ単体の像 $b(\Delta^2)$ で，その形は制御

点によって決まります．$b(\Delta^2)$ には単体の面の構造が入ります．すなわち， Δ^2 の頂点 $b(1,0,0), b(0,1,0), b(0,0,1)$ の値は制御点 $p_{(3,0,0)}, p_{(0,3,0)}, p_{(0,0,3)}$ の座標そのものです．さらに， Δ^2 の辺の値，たとえば $b(t, 1-t, 0)$ $(0 \leq t \leq 1)$ の値は辺上の制御点 $p_{(3,0,0)}, p_{(2,1,0)}, p_{(1,2,0)}, p_{(0,3,0)}$ の座標で決まります．この構造により，弱単体的な問題のパレート集合とパレートフロントの面構造を自然に反映することができます．



▲図 8.6　ベジエ単体の例

普遍近似定理

次の定理が示すように，単体からユークリッド空間へのあらゆる連続写像はベジエ単体によって近似することができます．

> **定理（普遍近似定理）** 任意の連続写像 $\phi : \Delta^{M-1} \to \mathbb{R}^N$ に対して，次を満たすようなベジエ単体の無限列 $b^i : \Delta^{M-1} \to \mathbb{R}^N$ が存在する．
> $\lim_{i\to\infty} \sup_{w\in\Delta^{M-1}} \left\| \phi(w) - b^i(w) \right\| = 0.$

この定理を直感的に説明すると，どれだけ複雑な連続写像をどれだけ小さな誤差で近似してほしいと要求されても，十分に次数の高いベジエ単体を使って制御点を適切に調節すれば要求を満たせると言っています．つまり，ベジエ単体は写像の近似に必要とされる表現力を十分に備えたモデルだといえます．一方でこの定理は，具体的な写像が与えられたときに，何次のベジエ単体を使えばいいかや，制御点をどう調節すればいいかまでは言及していません．次節では，具体的な写像を近似するためのアルゴリズムを紹介します．

フィッティングアルゴリズム

連続写像 $\phi : \Delta^{M-1} \to \mathbb{R}^N$ のサンプルが与えられたときに，次数を固定したベジエ単体をサンプルにフィッティングするアルゴリズムが開発されています．与えられた K 点のサンプル $S_K = \{ (w_k, x_k) \in \Delta^{M-1} \times \mathbb{R}^N \}_{k=1}^K$ にフィットするベジエ単体を求める問題は次の線形回帰問題になります．

$$\underset{p\in\mathbb{R}^{N\times\left|\mathbb{N}_D^M\right|}}{\text{minimize}}\ \frac{1}{K}\sum_{k=1}^{K}\|x_k - b_p(w_k)\|^2.$$

ここで p はすべての制御点を並べたベクトルです．これは 2 次関数なので逆行列法で厳密解を得ることもできますが， K, M, N が大きい場合には計算量が非常に大きくなります．L-BFGS などの勾配法を利用して近似解を求める方法もあり，よりスケーラブルです．

先述の定理とこのアルゴリズムに基づいて，Elastic net の解写像 $(\theta^*, \tilde{f}\circ\theta^*) : \Delta^2 \to \mathbb{R}^{N+3}$ のベジエ単体による近似を得ることができます．解写像のサンプルにおいて，入力はハイパーパラメータベクトル w であり，出力はモデルパラメータベクトル $\theta^*(w)$ と性能指標ベクトル $\tilde{f}(\theta^*(w))$ を並べたベクトルです．

8.4 PyTorch-BSF によるベジエ単体フィッティング

PyTorch-BSF はベジエ単体フィッティングの PyTorch 実装です．フィッティングには L-BFGS を利用しており，複数台の計算機での分散学習もサポートしているため，大規模データにもスケールします．

Miniconda のインストール

公式サイトから Miniconda[*11]をダウンロードして，インストールします．

コマンドとして使う

まずは手っ取り早く使ってみましょう．コマンドラインで呼び出すだけなら，インストールなしで使うこともできます．

まずは conda 環境に MLflow をインストールします．

```
conda install -c conda-forge mlflow
```

MLflow の `run`コマンドを使用して実行します．

```
mlflow run https://github.com/rafcc/pytorch-bsf \
  -P data=data.tsv \
  -P label=label.tsv \
  -P degree=3
```

実行時オプションの一覧を表 8.1 に示します．

[*11] `https://docs.conda.io/en/latest/miniconda.html`

▼表 8.1　実行時オプション一覧

名前	型	デフォルト	説明
data	path	required	データファイル．TSV 形式の数値データで，各行はベジエ単体への 1 つの入力ベクトルを表し，値はタブかスペースで区切ります．
label	path	required	ラベルファイル．TSV 形式の数値データで，各行はベジエ単体からの 1 つの出力ベクトルを表し，値はタブかスペースで区切ります．
degree	int ($x \geq 1$)	required	ベジエ単体の次数．高いほど複雑な超曲面を表現できますが過適合しやすくなります．
header	int ($x \geq 0$)	`0`	データ/ラベルファイルの読み飛ばす行数．
delimiter	str	`" "`	データ/ラベルファイルの列の区切り文字．
normalize	`max`, `std`, `quantile`, `none`	`none`	データの正規化手法．`max`は最小 0最大 1になるようにスケールします（一様分布するデータに適しています）．`std`は平均 0分散 1にスケールします（一様でないデータに適しています）．`quantile`は 5 パーセンタイルが 0で 95 パーセンタイルが 1になるようにスケールします（外れ値を含むデータに適しています）．`none`はスケールを行いません（正規化済みのデータに適しています）．
split_ratio	float ($0 < x < 1$)	`0.5`	訓練データと検証データの分割比．1に近いほど訓練データが多くなります．
batch_size	int ($x \geq 0$)	`0`	ミニバッチの大きさ．0はすべてのデータを 1 つのバッチとして訓練します．
max_epochs	int ($x \geq 1$)	`1000`	訓練を止めるエポック数．
accelerator	`auto`, `cpu`, `gpu`, ...	`auto`	訓練に使うアクセラレーター．詳しくは PyTorch Lightning のドキュメントを参照[*12]．
devices	int ($x \geq -1$)	`-1`	訓練に使うデバイスの数．-1はすべてのデバイスを使います．詳しくは PyTorch Lightning のドキュメントを参照[*13]．
num_nodes	int ($x \geq 1$)	`1`	訓練に使う計算ノード数．詳しくは PyTorch Lightning のドキュメントを参照[*14]．
strategy	`auto`, `dp`, `ddp`, ...	`auto`	分散処理の戦略．詳しくは PyTorch Lightning のドキュメントを参照[*15]．
loglevel	int ($0 \leq x \leq 2$)	`2`	何をログ出力するか．0: 何も出力しません．1: 指標を出力します．2: 指標とモデルを出力します．

[*12] https://pytorch-lightning.readthedocs.io/en/1.7.3/extensions/accelerator.html
[*13] https://pytorch-lightning.readthedocs.io/en/1.7.3/accelerators/gpu_basic.html
[*14] https://pytorch-lightning.readthedocs.io/en/1.7.3/guides/speed.html
[*15] https://pytorch-lightning.readthedocs.io/en/1.7.3/extensions/strategy.html

Python モジュールとして使う

Python モジュールとしてインストールすることもできます．次のコマンドでインストールします．

```
pip install pytorch-bsf
```

パッケージ名は pytorch-bsfですが，モジュール名は torch_bsfであることに注意してください．Python インタプリタでモジュールを実行することで，MLflow を使った場合と同様にコマンドラインからフィッティングを行うことができます．

```
python -m torch_bsf \
  --data=data.tsv \
  --label=label.tsv \
  --degree=3
```

Python スクリプトの中でモジュールをインポートすることで，他のプログラムと組み合わせて使うこともできます．

▼リスト 8.1　main.py

```
import torch
import torch_bsf

# Prepare training data
ts = torch.tensor(  # parameters on a simplex
    [
        [3/3, 0/3, 0/3],
        [2/3, 1/3, 0/3],
        [2/3, 0/3, 1/3],
        [1/3, 2/3, 0/3],
        [1/3, 1/3, 1/3],
        [1/3, 0/3, 2/3],
        [0/3, 3/3, 0/3],
        [0/3, 2/3, 1/3],
        [0/3, 1/3, 2/3],
        [0/3, 0/3, 3/3],
    ]
)
xs = 1 - ts * ts  # values corresponding to the parameters

# Train a model
bs = torch_bsf.fit(params=ts, values=xs, degree=3)

# Predict by the trained model
t = [[0.2, 0.3, 0.5]]
x = bs(t)
print(f"{t} -> {x}")
```

8.5　実験

多目的版 Elastic net の解写像が本当にベジエ単体で近似できるのか実験で確かめてみます．表 8.2 の 8 つのデータセットを使いました．これらはすべて UCI Machine

Learning Repository で公開されています．Elastic net で回帰を行うために，それぞれのデータセットの Web ページに書かれている方法で変数を説明変数と応答変数に分けました．2 つの応答変数をもつデータセットでは，Elastic net は説明変数をコピーし，それぞれの応答変数を予測するために使いました．つまり，Elastic net は Residential Building データセットでは合計 206 個の説明変数をもち，Slice Localization では 770 個の説明変数をもちます．説明変数と応答変数は，変数ごとに最小 0 最大 1 に正規化しました．

それぞれのデータセットについて，次のようにして解写像のサンプルを作りました．Δ^2 において格子状に 5,151 個のハイパーパラメータを生成しました．

$$w = \frac{1}{100}(n_1, n_2, n_3) \text{ such that } n_1, n_2, n_3 \in \{0, 1, \ldots, 100\},\ n_1 + n_2 + n_3 = 100.$$

それぞれの格子点 w について， $\theta^*(w)$ と $\tilde{f} \circ \theta^*(w)$ の値を計算しました．そうするために，重み $w = (w_1, w_2, w_3)$ を正則化係数 (μ, λ) に変換しました．ここで摂動の大きさ ε は 1E-16に設定しました．そして，単目的版の Elastic net 問題を座標降下法で解きました．

▼表 8.2　データセット

データセット	説明変数	応答変数	データ数
Blog Feedback[*16]	280	1	60,021
Fertility[*17]	9	1	100
Forest Fires[*18]	12	1	517
QSAR Fish Toxicity[*19]	6	1	908
Residential Building[*20]	103	2	372
Slice Localization[*21]	385	2	53,500
Wine[*22]	11	1	178
Yacht Hydrodynamics[*23]	6	1	308

上記の w を入力とし，Elastic net の学習結果 $(\theta^*(w), \tilde{f} \circ \theta^*(w))$ を出力として，ベジエ単体を先述の方法で訓練しました．

次の設定のすべての組み合わせに対して，異なる乱数種を用いて 10 試行ずつ実験を行いました．

[*16] https://archive.ics.uci.edu/ml/datasets/BlogFeedback
[*17] https://archive.ics.uci.edu/ml/datasets/Fertility
[*18] https://archive.ics.uci.edu/ml/datasets/Forest+Fires
[*19] https://archive.ics.uci.edu/ml/datasets/QSAR+fish+toxicity
[*20] https://archive.ics.uci.edu/ml/datasets/Residential+Building+Data+Set
[*21] https://archive.ics.uci.edu/ml/datasets/Relative+location+of+CT+slices+on+axial+axis
[*22] https://archive.ics.uci.edu/ml/datasets/wine
[*23] https://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics

Train-test 分割比

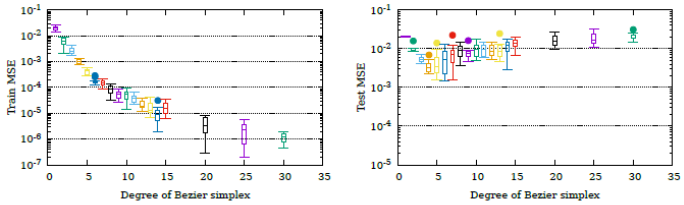
$$\text{Train} : \text{Test} = 51 : 5100, 257 : 4894, 515 : 4636, 5100 : 51.$$

ベジエ単体の次数

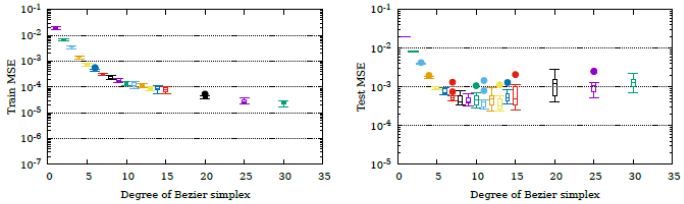
$$D = 1, 2, \ldots, 14, 15, 20, 25, 30.$$

実験は ITO[*24]で行いました．Elastic net とベジエ単体フィッティングは `Python 3.9` で `scikit-learn 0.24.2`と `pytorch-bsf 0.0.1`を用いて実装しました．

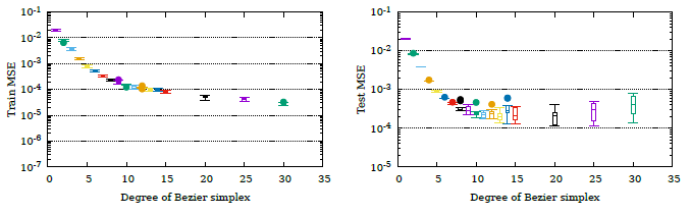
結果



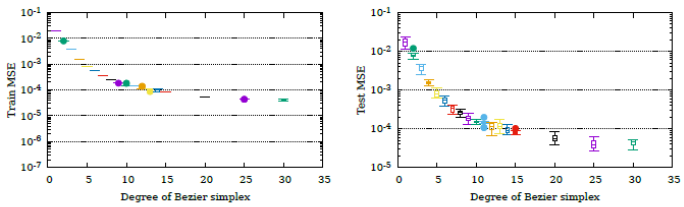
(a) Train : Test = 51 : 5100



(b) Train : Test = 257 : 4894



(c) Train : Test = 515 : 4636

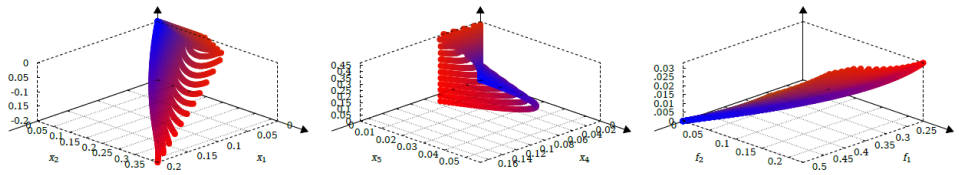


(d) Train : Test = 5100 : 51

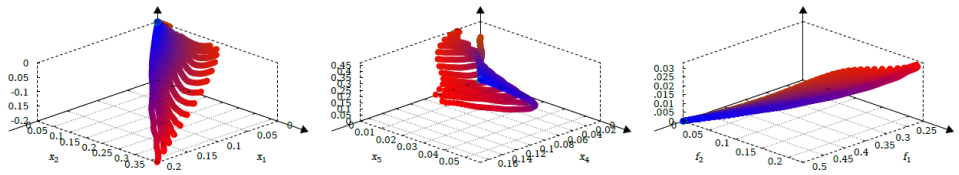
▲図 8.7　QSAR Fish Toxicity データセットにおける次数と MSE

[*24] `https://www.cc.kyushu-u.ac.jp/scp/eng/system/ITO/01_intro.html`

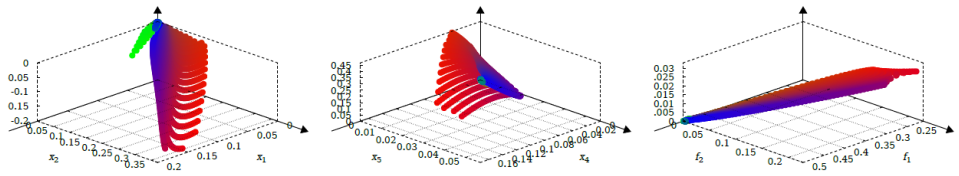
図 8.7 はベジエ単体の次数を変化させたときの訓練誤差とテスト誤差の変化を示しています．QSAR Fish Toxicity データセットをさまざまな train-test 分割比で学習したときのものです．分割比 51 : 5100 のとき (a) には，次数 $D = 4$ が平均テスト誤差最小となっています．分割比 257 : 4894 のとき (b) には，最適な次数は $D = 10$ になります．その平均テスト誤差は分割比 51 : 5100 で $D = 4$ としたときより低くなっています．分割比がより高くなるほど (c, d)，最適次数は高まっていき，その平均テスト誤差は低くなっていきます．この結果はベジエ単体の普遍近似定理と一致します．



(a) Ground truth (5151 elastic net models trained with varying hyper-parameters).



(b) Large sample approximation (A Bézier simplex of $d = 25$ trained with 5100 data points).



(c) Small sample approximation (A Bézier simplex of $d = 4$ trained with 51 data points).

▲図 8.8　QSAR Fish Toxicity データセットにおける解写像とそのベジエ単体近似（左列：$(\theta_1^*, \theta_2^*, \theta_3^*)(W)$，中列：$(\theta_4^*, \theta_5^*, \theta_6^*)(W)$，右列：$\tilde{f} \circ \theta^*(W)$）

次に，近似結果を観察してみます．図 8.8 は真の解写像のサンプルと 2 つの近似結果を比べています．1 つは大標本（train-test 分割は 5100 : 51 で，次数は $D = 25$ ）でもう 1 つは小標本（train-test 分割は 51 : 5100 で，次数は $D = 4$ ）です．図の各点は重み w の (w_1, w_2, w_3) 座標値を RGB 値に変換して色付けしてあります．真の解写像 (a) において色が連続的に変化していることから，θ^* が連続写像であること（さらに $\tilde{f} \circ \theta^*$ も連続写像であること）が確認できます．大標本近似 (b) は真の解写像に近く，異なるハイパーパラメータで訓練したすべての Elastic net モデルとその性能指標値が 1 つのベジエ

単体で表されています．小標本近似 (c) でさえ，パレートフロントは依然としてよく近似できていることは驚くべきことです．このことは，近似的なパレートフロントに基づいてトレードオフ分析を行い，ハイパーパラメータを選んでも，実際の性能と大きく乖離しないであろうことを意味しています．

▼表 8.3　最適次数 D^* とその近似誤差（平均 ± 標準偏差）

データセット	D^*	Test MSE（大標本）	D^*	Test MSE（小標本）
Blog Feedback	30	5.21E-04 ± 4.28E-04	1	5.62E-03 ± 1.26E-04
Fertility	30	4.71E-05 ± 1.34E-05	3	7.56E-03 ± 1.82E-03
Forest Fires	30	5.52E-05 ± 3.08E-05	3	7.17E-03 ± 1.11E-03
QSAR Fish Toxicity	25	4.16E-05 ± 1.09E-05	4	3.66E-03 ± 1.41E-03
Residential Building	25	3.55E-04 ± 2.55E-04	3	6.94E-03 ± 7.20E-04
Slice Localization	30	5.95E-04 ± 4.38E-04	3	8.83E-03 ± 1.60E-03
Wine	30	6.71E-05 ± 1.42E-05	3	7.00E-03 ± 5.63E-04
Yacht Hydrodynamics	30	6.75E-05 ± 4.32E-05	3	3.51E-03 ± 3.62E-04

すべてのデータセットに対する近似誤差を表 8.3 に示します．大標本（train-test 分割比 5100 : 51 ）では，次数を $D = 30$ に設定することは QSAR Fish Toxicity と Residential Building を除くすべてのデータセットでテスト誤差を最小にしています．このことは，普遍近似定理が示唆するように，（十分に次数の高い）ベジエ単体はどんなデータセットを学習した Elastic net の解写像をも近似できることを示しています．一方で，小標本（train-test 分割比 51 : 5100 ）では，次数を $D = 3$ に設定することで Blog Feedback と QSAR Fish Toxicity を除くすべてのテスト誤差が最小化されています．この結果の意味することは，最適な次数の値は，どんなデータセットを Elastic net が学習しているかにはあまり左右されず，主にベジエ単体に与える訓練データの数（訓練した Elastic net モデルの数）によって決まるということです．

考察

図 8.8 でみたように，パレート集合を精度よく近似するには大標本に次数の高いベジエ単体をフィットさせる必要がありますが，パレートフロントを精度よく近似するには小標本に次数の低いベジエ単体をフィットさせれば十分な傾向があるようです．これは解写像の滑らかさの違いが原因と思われます．Elastic net の L_1 正則化がもたらす変数選択によって，写像 $\theta^* : \Delta^2 \to X^*(\tilde{f})$ の像は角をもちます（図 8.8(a) 中央）．それにもかかわらず，その角は写像 $\tilde{f} \circ \theta^* : \Delta^2 \to \tilde{f}(X^*(\tilde{f}))$ の像には現れません（図 8.8(a) 右）．同様の性質は実験に使ったすべてのデータセットで観察されました（紙面の都合でそれらの図は省略します）．この角でベジエ単体を分割することができれば，パレート集合を精度よく近似するために必要な標本サイズとベジエ単体の次数を低減できると思われます．現在ではそのような再分割アルゴリズムも研究が進んでいます．

もっとも好ましいハイパーパラメータを見つけるために，もっとも普及した Elastic

net 実装である Rパッケージ glmnetのデフォルト設定では，1 つのハイパーパラメータの 100 通りの値それぞれについて 10 組交差検証を計算します．もし 2 つのハイパーパラメータの組み合わせについて同じことを行うならば，Elastic net の訓練が $10 \times 100 \times 100 = 100{,}000$ 回も必要となります．それに対して，ベジエ単体フィッティングを用いると，わずか 51 回の Elastic net の訓練でパレートフロント全体を明らかにすることができました．訓練回数をおよそ 1/2000 にまで削減できた理由は，主に 2 つあります．

- ベジエ単体によって解写像のサンプルが滑らかに補間されることを期待して，訓練するハイパーパラメータの間隔を大きくとることができます．そのため，グリッドサンプリングを避けてランダムサンプリングで済ませることができます．
- 個々の Elastic net の訓練では train-test ランダム分割によって性能指標値にゆらぎが生じますが，フィッティングによって周辺の訓練結果と平均化されてゆらぎが消えます．そのため，交差検証を避けてハイパーパラメータあたり 1 回ずつの訓練で済ませることができます．

ベジエ単体近似を使ってもっとも好ましいハイパーパラメータを見つけるために，重みとハイパーパラメータの対応関係を把握しておくことは重要です．もし単目的版の Elastic net 問題のハイパーパラメータ (μ, λ) が与えられたら，それを多目的版 Elastic net 問題の重み (w_1, w_2, w_3) に変換することができます．任意の μ, λ が

$$0 \leq \mu \leq \frac{\lambda - \varepsilon}{\varepsilon}$$

を満たすとき，多目的版 Elastic net 問題の関数 $g_{\mu,\lambda}$ の最小点は点 $\theta^*(w(\mu, \lambda))$ です．ここで， $w(\mu, \lambda) = (w_1(\mu, \lambda), w_2(\mu, \lambda), w_3(\mu, \lambda))$ は次の式で定義されます．

$$\begin{aligned} w_1(\mu, \lambda) &= \frac{1 + \varepsilon}{\lambda + \mu + 1}, \\ w_2(\mu, \lambda) &= \frac{(1 + \varepsilon)\mu}{\lambda + \mu + 1}, \\ w_3(\mu, \lambda) &= \frac{\lambda - \varepsilon(\mu + 1)}{\lambda + \mu + 1}. \end{aligned}$$

また， $\lambda + \mu + 1 \neq 0$ かつ $w(\mu, \lambda) \in \Delta^2 \setminus \Delta^2_{\{2,3\}}$ であることに注意してください．

8.6 おわりに

本章では，多目的最適化とベジエ単体フィッティングについて紹介しました．はじめに，すべての無制約多目的強凸最適化問題は弱単体的であることを示しました．また，任意の弱単体的な問題の解写像はベジエ単体で近似できることを示しました．そして，ベジエ単体を与えられたデータにフィットさせるアルゴリズムを紹介し，それを実装したパッ

ケージ PyTorch-BSF の使い方を説明しました．以上の応用として，スパースモデリングの一手法である Elastic net に対して，解写像（この事例では，あらゆるハイパーパラメータで訓練されたすべてのモデル）をベジエ単体で近似する方法を紹介しました．

ベジエ単体フィッティングの研究は今も進歩しており，その適用範囲はさらに広がり続けています．今回は，最適化問題の強凸性から弱単体性を導くことで，ベジエ単体フィッティングで解写像が近似できることを保証しました．一方で，世の中には強凸ではない最適化問題もたくさんあります．そのような場合であっても，パレート集合を近似するサンプルを求めることができるケースは多いです．そのようなサンプルをベジエ単体でフィッティングしてもよいかどうかを判断するために，サンプルから問題が弱単体的であるかどうかを検定する方法が開発されています[*25]．

今回紹介した応用例ではパレート集合の高精度なサンプルが得られたもとでフィッティングを行いましたが，より難しい最適化問題ではそもそも解を精度良く求めることができるとは限りません．パレート集合との誤差が大きいサンプルに対しては，今回紹介した決定的なフィッティングアルゴリズムは過適合を起こしやすいことが知られています．その場合でも過適合を起こしにくいように拡張された，近似ベイズ計算によるベジエ単体フィッティングがあります[*26]．

今回は何らかの方法でパレート集合のサンプルを求めてから，サンプルに対してベジエ単体をフィッティングしました．一方で，ベジエ単体反復更新して最適化問題を解くことで，パレート集合にフィットしたベジエ単体を求めるアプローチもあります[*27]．特に，解の評価回数が限られている場合にはこのアプローチが有効であるとの実験結果が得られています．

[*25] https://doi.org/10.48550/arXiv.1804.07179
[*26] https://doi.org/10.48550/arXiv.2104.04679
[*27] https://doi.org/10.48550/arXiv.2205.11099

第 9 章

Jupyter カーネル自作入門

Makiuchi Daisuke

Jupyter Lab（あるいは Jupyter Notebook）[*1]は、ブラウザ上でプログラムを記述し実行できるウェブアプリケーションです。プログラムと一緒に説明や実行時の入出力を**ノートブック**という形式でまとめて保存でき、実験の記録などに便利なツールです。機械学習やデータ分析でよく使われているので、ご存知の方も多いと思います。

Project Jupyter は元々は Python のインタラクティブインタプリタ IPython の派生プロジェクトですが、プログラムの実行環境が**カーネル**として分離されているため、現在では Python 以外にもコミュニティによるものも含め数十の言語がサポートされています。

この章では、この Jupyter に新たな言語のカーネルを自作して追加する方法を解説します。題材として、**Whitespace**[*2]を実行するカーネルを Go 言語で実装した「whitenote」を用意しました。紙面の都合上抜粋しての解説となりますので、実際に動かしたりコードの全体を見たい場合はリポジトリをご覧ください。

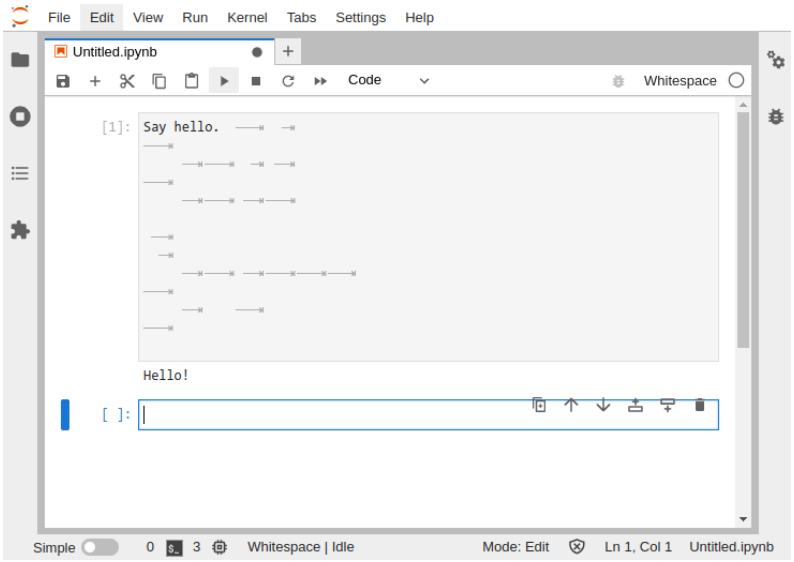
- `https://github.com/makiuchi-d/whitenote`

また、筆者の開発環境は次のとおりです。

- Ubuntu 20.04
- Jupyter Lab 3.4.4
- Go 1.19

[*1] `https://jupyter.org/`

[*2] `https://web.archive.org/web/20150618184706/http://compsoc.dur.ac.uk/whitespace/`



▲図 9.1　whitenote

9.1　Jupyter カーネルの基本

Jupyter のカーネルは Jupyter から起動される独立したプログラムで、基本的には 1 ノートブックに対し 1 プロセスが起動されます。Jupyter との通信には **ZeroMQ**[*3]というライブラリを利用します。このため、ZeroMQ が利用できるものであれば、どんな言語でもカーネルを開発できます。

公式のドキュメントにもカーネルの作り方の解説があります[*4]。Python で実装する場合は `ipykernel.kernelbase.Kernel`を拡張することで簡単に実装できますが、ここでは他言語でも真似できるよう、ZeroMQ を直接操作する方法を紹介します。

ZeroMQ とは

Jupyter が利用する ZeroMQ は、軽量な非同期メッセージングライブラリです。ZeroMQ 自体は C++ で開発されていますが、多くの言語で利用できるようにライブラリとバインディングが用意されていて[*5]、相互に通信できるようになっています。

ZeroMQ ではインターフェイスとして、TCP などのソケットをラップしたような使い勝手の**ソケット**が提供されます。このソケットにはさまざまなタイプ、たとえばメッセー

[*3] `https://zeromq.org/`
[*4] `https://jupyter-client.readthedocs.io/en/latest/kernels.html`
[*5] `https://zeromq.org/get-started/#pick-your-language`

ジを分配したり、ルーティングを自動で行ってくれるものなどが用意されています。これらを組み合わせることで、Pub/Sub や分散タスク処理のような N 対 N の通信を柔軟に組み立てることができます。

Go 言語で ZeroMQ を利用するにはいくつかの選択肢があります。公式サイトで紹介されている、goczmq[*6]、pubbe/zmq4[*7]のほか、Go 言語のみで再実装された go-zeromq/zmq4[*8]などがあります。

ここでは、他言語でも利用できる **libzmq** をシンプルにラップしている pubbe/zmq4 を使うことにしました。Ubuntu（focal, jammy）や Debian（bullseye）では次のコマンドで libzmq をインストールできます。

```
apt install libzmq3-dev libzmq5
```

通信に使うソケット

Jupyter のカーネルは、表 9.1 の 5 つのソケットを使用します。

▼表 9.1　ソケット一覧

名前	タイプ	役割
Shell	ROUTER	コードの実行や各種情報のリクエストを受け付ける
IOPub	PUB	標準出力や状態を Jupyter に通知する
Stdin	ROUTER	標準入力への入力を Jupyter にリクエストし受け取る
Control	ROUTER	Shell と並行しての情報の取得や、終了リクエストを受け付ける
HB	REP	疎通確認（HeartBeat）の送受信を行う

コードの実行のような Jupyter からのリクエストは、**Shell ソケット**に届きます。つまりカーネルの基本動作は Shell ソケットに届いたリクエストを順次処理していくことです。その過程で入出力があれば、IOPub や Stdin のソケットを使って通信します。

Jupyter とカーネルの通信は基本的に 1 対 1 ですが、複数のリクエストを並行して送受信できるように ROUTER タイプのソケットが使われています。

メッセージの基本構造

メッセージの構造は公式ドキュメントでも解説されているのですが[*9]、libzmq を直接使って実装するには説明が不十分なので注意が必要です[*10]。

[*6] https://github.com/zeromq/goczmq
[*7] https://github.com/pebbe/zmq4
[*8] https://github.com/go-zeromq/zmq4
[*9] https://jupyter-client.readthedocs.io/en/latest/messaging.html
[*10] Python で実装する場合はライブラリが隠蔽しているのでしっかりとは書いていないのでしょう。

さっそくドキュメントには書かれていないのですが、Jupyter との通信は ZeroMQ のマルチパートメッセージで行います。これは、複数のブロックをまとめてひとつのメッセージとして扱うものです。

pubbe/zmq4 では、`RecvMessageBytes()`と `SendMessage()`を利用します。libzmq の API としては、送受信時に `ZMQ_SNDMORE`、`ZMQ_RCVMORE`を使うことになります[*11]。

▼リスト 9.1　マルチパートメッセージの送受信関数

```
func (*zmq4.Socket) RecvMessageBytes(flags zmq4.Flag) (msg [][]byte, err error)
func (*zmq4.Socket) SendMessage(parts ...interface{}) (total int, err error)
```

メッセージの内容は表 9.2 に示すブロックの列になっています。このうち`{header}`、`{parent_header}`、`{metadata}`、`{content}`はそれぞれ JSON エンコードされた辞書データです。

▼表 9.2　メッセージの内容

ブロック	内容
`"<IDS|MSG>"`	メッセージの先頭を表すデリミタ文字列
HMAC	検証のためのシグネチャ（16 進数文字列）
`{header}`	メッセージの種別を表すヘッダ
`{parent_header}`	親メッセージのヘッダ（ない場合は`"{}"`）
`{metadata}`	メタデータ
`{content}`	メッセージのコンテンツ
..	追加データがある場合はブロックが続く

ROUTER タイプのソケットで通信する場合、メッセージ本体の前に ZeroMQ が利用する ID（ZmqID）が付加されます。ZeroMQ でよくあるソケットの組み合わせ、たとえば ROUTER-DEALER パターンなどでは、この ZmqID はソケットが自動的に付け外ししてくれるので意識する必要はありません。しかし ROUTER ソケットを直接扱う場合、つまり Shell、Stdin、Control のソケットの処理では、この ZmqID を適切に操作しなくてはなりません。

ROUTER の詳細は ZeroMQ のガイドブック[*12]に書かれているので、興味のある方は参照ください。

[*11] `http://api.zeromq.org/master:zmq-send`、`http://api.zeromq.org/master:zmq-recv`
[*12] `https://zguide.zeromq.org/` 日本語訳:`https://www.cuspy.org/diary/2015-05-07-zmq/`

9.2 最小のカーネル

カーネルとして最低限必要なのは次の 4 つです。

- 起動してもらえるようカーネルを登録する
- 通信に使うソケットを準備する
- kernel_info_requestに応答する
- HeartBeat に応答する

カーネルの登録

カーネルは Jupyter とは独立したプログラムなので、まずは Jupyter に起動してもらえるよう登録します。具体的には、特定のディレクトリに kernel.jsonファイルを配置することで登録します。このファイルには、カーネルのコマンドやパラメータを記載します（リスト 9.2）。詳細は公式ドキュメントをご覧ください[*13]。

▼リスト 9.2　kernel.json

```
{
    "argv": [
        "whitenote",
        "{connection_file}"
    ],
    "display_name": "Whitespace",
    "language": "whitespace",
}
```

"argv"がカーネルのコマンドとパラメータです。"{connection_file}"は、後述する通信のための情報が書かれたファイルのパスに置き換えられます。他に必要なパラメータがある場合ここに追加します。"display_name"が Jupyter 上に表示される名前です。Jupyter でカーネルが選択されると、この指定にしたがってコマンドが起動されます。

ロゴ画像を設定するには logo-64x64.pngという PNG ファイルを同じディレクトリに配置します[*14]。画像がなくても名前の頭文字がロゴ画像として使われるので、必須ではありません。

ファイルを用意したら次のコマンドで配置します。OS によって異なりますが、Linux では~/.local/share/jupyter/kernelsに--nameで指定した名前のディレクトリが作られ、コピーされます。

```
jupyter kernelspec install --name=whitenote --user {kernel.json のディレクトリ}
```

[*13] https://jupyter-client.readthedocs.io/en/stable/kernels.html#kernelspecs
[*14] logo-32x32.pngは使われていません。https://github.com/ipython/ipython/pull/6537

正しく登録されているかは、Jupyter の画面や次のコマンドで確認できます。

```
jupyter kernelspec list
```

ソケットの準備

通信に使うソケットの接続情報は、起動パラメータで指定される"{connection_file}"という名の JSON ファイルで渡されます。これにはソケットの接続プロトコル、ポート番号、IP アドレス、そしてメッセージの署名に使うアルゴリズムとキーが含まれます。

▼リスト 9.3　connection_file の内容

```
{
  "shell_port": 49835,
  "iopub_port": 53257,
  "stdin_port": 34911,
  "control_port": 42447,
  "hb_port": 55339,
  "ip": "127.0.0.1",
  "key": "ef710209-2e9d78e0f61f5ec628d0c840",
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "kernel_name": "whitenote"
}
```

単純な JSON ファイルなので、Go 言語では標準ライブラリで読み取ることができます。whitenote ではリスト 9.4 の構造体にマッピングしています。

▼リスト 9.4　ConnectionInfo 構造体

```
type ConnectionInfo struct {
    SignatureScheme string `json:"signature_scheme"`
    Transport       string `json:"transport"`
    StdinPort       int    `json:"stdin_port"`
    ControlPort     int    `json:"control_port"`
    IOPubPort       int    `json:"iopub_port"`
    HBPort          int    `json:"hb_port"`
    ShellPort       int    `json:"shell_port"`
    Key             string `json:"key"`
    IP              string `json:"ip"`
}
```

この情報を元に Jupyter との通信に使うソケットを作り、ポートに紐づけるコードをリスト 9.5 に示します。5 つのソケットは Sockets構造体にまとめました。

▼リスト 9.5　ソケットの準備

```
type Sockets struct {
    conf    *ConnectionInfo
    shell   *zmq4.Socket
    control *zmq4.Socket
    stdin   *zmq4.Socket
    iopub   *zmq4.Socket
```

```
    hb      *zmq4.Socket
}

func bindSocket(typ zmq4.Type, transport, ip string, port int) *zmq4.Socket {
    sock, err := zmq4.NewSocket(typ)
    if err != nil {
        panic(err)
    }
    sock.Bind(fmt.Sprintf("%s://%s:%d", transport, ip, port))
    return sock
}

func newSockets(conf *ConnectionInfo) *Sockets {
    return &Sockets{
        conf:    conf,
        shell:   bindSocket(zmq4.ROUTER, conf.Transport, conf.IP, conf.ShellPort),
        control: bindSocket(zmq4.ROUTER, conf.Transport, conf.IP, conf.ControlPort),
        stdin:   bindSocket(zmq4.ROUTER, conf.Transport, conf.IP, conf.StdinPort),
        iopub:   bindSocket(zmq4.PUB, conf.Transport, conf.IP, conf.IOPubPort),
        hb:      bindSocket(zmq4.REP, conf.Transport, conf.IP, conf.HBPort),
    }
}

func main() {
    ... (略)

    socks := new Sockets(conf)

    go socks.shellHandler()
    go socks.controlHanlder()
    go socks.hbHandler()

    ... (略)
```

これらのソケットはすべて、カーネル側で Bind して Jupyter からの接続を待ち受ける形をとります。実際の接続処理は ZeroMQ がバックグラウンドで行ってくれます。あとは待っていれば Jupyter 側からメッセージを送ってくるので、それをハンドラ関数で処理していくことになります。

Shell ハンドラの実装

カーネルに接続した Jupyter は、最初に Shell ソケットに `kernel_info_request`を送ってきます。最小のカーネルでも、このリクエストにだけは応答しなければなりません。

このリクエストに対してカーネルは `kernel_info_reply`を返し、IOPub 経由で状態を`"idle"`として通知します。また、Control ソケットにも `kernel_info_request`が送られてきますが、Shell ソケットで応答するので、そちらは読み捨てます。

ここではまず、リスト 9.6 に Shell のハンドラメソッドを `shellHandler`に示し、その内容について詳しく説明していきます。

▼リスト 9.6　Shell のハンドラメソッド

```
func (s *Sockets) shellHandler() {
    for {
        // メッセージの受信
        msg, err := s.recvRouterMessage(s.shell)
        if err != nil {
            log.Printf("shell: recv: %v", err)
```

```
            continue
        }

        // header のデコード
        var hdr map[string]any
        if err := json.Unmarshal(msg.Header, &hdr); err != nil {
            log.Printf("shell: header: %v", err)
            continue
        }

        // メッセージ種別ごとの処理
        switch hdr["msg_type"] {

        case "kernel_info_request":
            // kernel_info_reply の送信
            s.sendRouter(s.shell, msg, "kernel_info_reply", kernelInfo)

            // 状態を"idle"に
            s.sendState(msg, stateIdle)
        }
    }
}
```

メッセージの受信

メッセージを受信する関数をリスト 9.7 に示します。Shell は ROUTER なので、先頭に ZmqID が付加されます。ROUTER が多段になっている場合、ZmqID が複数ブロックになっていることもあります。ZmqID とメッセージの区切りは、デリミタ文字列"<IDS|MSG>"のブロックによって識別します。

▼リスト 9.7　ROUTER ソケットからの Message 読み込み

```
const delimiter = "<IDS|MSG>"

type Message struct {
    ZmqID    [][]byte
    Header   []byte
    Parent   []byte
    Metadata []byte
    Content  []byte
    Buffers  [][]byte
}

func (s *Sockets) recvRouterMessage(sock *zmq4.Socket) (*Message, error) {
    mb, err := sock.RecvMessageBytes(0)
    if err != nil {
        return nil, err
    }

    // デリミタを探す
    var d int
    for d = 0; d < len(mb); d++ {
        if bytes.Equal(mb[d], []byte(delimiter)) {
            break
        }
    }
    if d > len(mb)-5 {
        return nil, fmt.Errorf("invalid message: %v,%v, %v", d, len(mb), mb)
    }

    msg := &Message{
        ZmqID:    mb[:d],
        Header:   mb[d+2],
        Parent:   mb[d+3],
```

```
        Metadata: mb[d+4],
        Content:  mb[d+5],
        Buffers:  mb[d+6:],
    }

    // シグネチャの検証
    sig := string(mb[d+1])
    mac := calcHMAC(s.conf.Key, msg.Header, msg.Parent, msg.Metadata, msg.Content)
    if sig != mac {
        return msg, fmt.Errorf("invalid hmac: %v %v", sig, mb)
    }

    return msg, nil
}
```

シグネチャの検証

デリミタの次のブロックは、メッセージの検証のためのシグネチャです。受信時の検証をスキップしたり、送信時もシグネチャを空文字列とすることで検証を無効にもできますが、簡単なので実装してしまいます。

アルゴリズムは ConnectionInfoの"signature_scheme"で指定されますが、いまのところ SHA256 の HAMC 固定です。また、HMAC のキーも ConnectionInfoの"key"として渡されます。このキーを使い、受信したメッセージの{header}、{parent_header}、{metadata}、{content}をこの順に連結したもののハッシュを計算し検証します。追加データ（Buffers）はここに含みません。

▼リスト 9.8　HMAC の計算

```
func calcHMAC(key string, header, parent, metadata, content []byte) string {
    h := hmac.New(sha256.New, []byte(key))
    h.Write(header)
    h.Write(parent)
    h.Write(metadata)
    h.Write(content)
    return hex.EncodeToString(h.Sum(nil))
}
```

メッセージの種別

メッセージの{header}はリスト 9.9 のような JSON オブジェクトです。shellHandlerでは辞書 map[string]anyとしてデコードしています。

▼リスト 9.9　メッセージの{header}

```
{
  "date": "2022-08-13T06:32:13.893Z",
  "msg_id": "c1735592-e938-4d8a-b7a2-769d795f65d0",
  "msg_type": "kernel_info_request",
  "session": "aa3af91f-a747-42c7-b0b8-a02179aee1e1",
  "username": "",
  "version": "5.2"
}
```

ここで必要なのは、メッセージ種別を表す"msg_type"だけです。カーネルが処理しないメッセージは単に読み捨てるだけでよいので、ここでは"kernel_info_request"のメッセージのみ処理します。

kernel_info_replyの送信

"kernel_info_request"に対しては、"kernel_info_reply"という msg_typeのメッセージを返します。このときメッセージの{content}はリスト 9.10 のようにカーネルの情報をまとめた JSON オブジェクトです。これに{header}などを合わせてメッセージを組み立て送信します。このカーネル情報は基本的に固定値なので、init()で初期化して保持しています。

また、後で必要となる sessionIdと、基本的に空のままの metadataも起動中変更されることはないので、同じようにグローバルに保持することにします。

▼リスト 9.10　固定値の初期化

```
var (
    sessionId  string // プロセスごとにユニークな ID
    kernelInfo []byte // カーネル情報

    metadata  = []byte("{}")
)

func init() {
    sid, _ := uuid.NewRandom()
    sessionId = sid.String()

    kernelInfo, _ = json.Marshal(map[string]any{
        "status":                 "ok",
        "protocol_version":       "5.3",
        "implementation":         "whitenote",
        "implementation_version": "0.1",
        "language_info": map[string]any{
            "name":               "whitespace",
            "version":            "0.1",
            "mimetype":           "text/x-whitespace",
            "file_extension":     ".ws",
            "pygments_lexer":     "",
            "codemirror_mode":    "",
            "nbconvert_exporter": "",
        },
        "banner": "",
    })
}
```

次に、ヘッダを構築する関数はリスト 9.11 のようにしました。msg_typeだけ指定すれば構築できるようにしてあります。

▼リスト 9.11　ヘッダ構築関数

```
func newHeader(msgtype string) []byte {
    mid, _ := uuid.NewRandom()
    h := map[string]any{
        "date":     time.Now().Format(time.RFC3339), // 現在時刻
```

```
        "msg_id":   mid.String(), // メッセージ毎にユニークな UUID
        "username": "kernel",
        "session":  sessionId,    // プロセスごとにユニークな UUID
        "msg_type": msgtype,
        "version":  "5.3",
    }
    hdr, _ := json.Marshal(h)
    return hdr
}
```

Shell ソケットは ROUTER なので、送信するときには ZmqID がメッセージの先頭に必要です。ここでは親メッセージ、つまり"kernel_info_request"の ZmqIDをそのまま使います。また、{parent_header}も親メッセージの{header}です。

これで返信に必要な情報が揃いました。

- ZmqID：親メッセージの ZmqID
- HMAC：calcHMAC()で計算
- {header}：msg_typeを"kernel_info_reply"として構築
- {parent_header}：親メッセージの{header}
- {metadata}：{}
- {content}：カーネル情報 kernelInfo

これらを順番どおりに結合してソケットの SendMessage()で送信します。この処理を sendRouter()メソッドとしてまとめました（リスト 9.12）。

▼リスト 9.12　sendRouter メソッド

```
func (s *Sockets) sendRouter(
    sock *zmq4.Socket, parent *Message, msgtype string, content []byte) {

    hdr := newHeader(msgtype)
    phdr := parent.Header
    mac := calcHMAC(s.conf.Key, hdr, phdr, metadata, content)
    data := make([]any, 0, len(parent.ZmqID)+6)
    for _, p := range parent.ZmqID {
        data = append(data, p)
    }
    data = append(data, delimiter)   // "<IDS|IMG>"
    data = append(data, mac)         // HMAC
    data = append(data, hdr)         // {header}
    data = append(data, phdr)        // {parent_header}
    data = append(data, metadata)    // {metadata}
    data = append(data, content)     // {content}
    _, _ = sock.SendMessage(data...)
}
```

状態の通知

"kernel_info_reply"を返した後、カーネルはコードの実行準備が整ったことを Jupyter に伝えます。これは IOPub ソケットに対して"idle"状態を通知することで行います。この状態通知メソッドを sendState()としてリスト 9.13 のように定義しました。

▼リスト 9.13　state の送信

```
var (
    stateIdle = []byte(‘{"execution_state":"idle"}‘)
    stateBusy = []byte(‘{"execution_state":"busy"}‘)
)

func (s *Sockets) send(sock *zmq4.Socket, parent *Message, msgtype string, content []byte) {
    hdr := newHeader(msgtype)
    phdr := parent.Header
    mac := calcHMAC(s.conf.Key, hdr, phdr, metadata, content)
    _, _ = sock.SendMessage(delimiter, mac, hdr, phdr, metadata, content)
}

func (s *Sockets) sendState(parent *Message, state []byte) {
    s.send(s.iopub, parent, "status", state)
}
```

{content}は{"execution_state":"idle"}とします。このバイト列は不変でかつ何度も使うことになるので、"busy"のものと合わせてグローバルに保持しました。{header}は msg_typeを"status"とし、{parent_header}は"kernel_info_request"のものにします。

IOPub は PUBソケットなので、ZmqID は必要ありません。デリミタ（"<IDS|MSG>"）から順にマルチパートメッセージを送ります。

ここまで実装したら、Jupyter はカーネルをきちんと起動できるようになります。"kernel_info_reply"を正しく返せなかったり、"idle"状態にできなかったりすると、Jupyter はしつこく"kernel_info_request"を何度も送ってきます。もしそのような挙動になったら、今一度実装を見直してみてください。

Control と HB（HeartBeat）のハンドラ

Control ソケットには"kernel_info_request"のほか、いくつかのリクエストが届きます。Jupyter のカーネルでは、処理しないリクエストは単に読み捨てることになっています。また、シャットダウン要求"shutdown_request"も届きますが、これを無視しても Jupyter からは SIGINT が送られてくるので、シグナルハンドラを変更していないなら自動的に終了してくれます。ということで、Control のハンドラはリスト 9.14 のように、すべて読み捨てるだけの実装としました。

▼リスト 9.14　Control ハンドラの実装

```
func (s *Sockets) controlHandler() {
    for {
        _, _ = s.recvRouterMessage(s.control)
    }
}
```

HB ソケットには疎通確認のメッセージが届きます。このメッセージはそのまま HB ソケットで送り返すことで疎通していることを伝えます。

メッセージをひとつひとつ Recv()、Send()するループを書いてもよいのですが、ZeroMQ の組み込み Proxy を使うこともできます（リスト 9.15）。

▼リスト 9.15　組み込み Proxy による HBHandler

```
func (s *Sockets) hbHandler() {
    zmq4.Proxy(s.hb, s.hb, nil)
}
```

これで最小の何もしないカーネルが実装できました。コードの実行要求"execute_request"に対して何もしていないので、Jupyter 上で実行ボタンを押してもなにも起こりませんが、通信はできています。

9.3　Whitespace とは

ここからは、Jupyter に新たな言語として Whitespace のカーネルを実際に組み込んでみます。Whitespace を選択したのは、実装が簡単なことに加え、調べた限り誰も作っていなさそう[*15]だったからです。

Whitespace は難解プログラミング言語のひとつで、2003 年 4 月 1 日に Edwin Brady と Chris Morris によって開発、発表されました。公式サイトはすでに消滅していますが、Internet Archive で見ることができます。

この言語の特徴はなんといっても、スペース、タブ、改行という空白文字 3 種のみで記述することです。それ以外の文字は全て無視されます。リスト 9.16 に「Hello!」と表示するプログラムを示します[*16]。

▼リスト 9.16　Hello!と表示するプログラム

```

```

Whitespace はヒープメモリを備えたスタックベースの言語で、表 9.3 の命令セットで構成されます。便宜上、スペースを S、タブを T、改行を N として表記します。詳細は公

[*15] ググラビリティが低いため、見つけられていないだけかもしれません。

[*16] 可視化するとこうなります: SSSTSSTSSSNTNSSSSSTTSSTSTNTNSSSSSTTSTTSSN
SNSTNSSTNSSSSSTTSTTTTNTNSSSSSTSSSSTNTNSSNNN

式サイトのチュートリアル[*17]をご覧ください。

▼表 9.3　Whitespace の命令セット

命令	引数	意味
SS	数値	数値をスタック先頭に Push
SNS		スタック先頭のアイテムを複製
STS	数値	スタックの N 番目のアイテムを先頭にコピー
SNT		スタック先頭の 2 つを入れ替え
SNN		スタック先頭のアイテムを破棄
STN	数値	先頭のアイテムを保持したまま N 個のアイテムを破棄
TSSS		加算
TSST		減算
TSSN		乗算
TSTS		除算
TSTT		剰余
TTS		先頭アイテムを 2 番目の示すアドレスのヒープに保存
TTT		先頭の示すアドレスのヒープから値をスタックに取り出す
NSS	ラベル	ラベルを設置
NST	ラベル	サブルーチン呼び出し
NSN	ラベル	ラベルへジャンプ
NTS	ラベル	スタック先頭が 0 ならラベルへジャンプ
NTT	ラベル	スタック先頭が負ならラベルへジャンプ
NTN		サブルーチン呼び出し元へ戻る
NNN		プログラム終了
TNSS		スタック先頭を文字として出力
TNST		スタック先頭を数値として出力
TNTS		入力から 1 文字読み、スタック先頭の示すヒープに保存
TNTT		入力から数値を読み、スタック先頭の示すヒープに保存

9.4　インタプリタの実装

whitenote の wspace パッケージ[*18]に Whitespace インタプリタを実装しました。実装の詳細はリポジトリを見ていただくとして、ここではインタプリタの本体である `wspace.VM`の使い方を簡単に紹介します。

[*17] https://web.archive.org/web/20150618184706/http://compsoc.dur.ac.uk/whitespace/tutorial.php

[*18] https://github.com/makiuchi-d/whitenote/tree/main/wspace

▼リスト 9.17　wspace.VMの使い方

```
vm := wspace.New()

err := vm.Load("   \t\t \t\n\t\n \t\n\n\n")
if err != nil {
    panic(err)
}

err = vm.Run(context.Background(), os.Stdin, os.Stdout)
if err != nil {
    panic(err)
}
```

wspace.VMでは、コードの読み込み vm.Load()と実行 vm.Run()が分かれています。Whitespace の文法上、ラベルの定義より前にそのラベルへのジャンプ命令が出現しうるため、実行する前にコード全体を読み込んでおかないと適切にジャンプできません。

また、vm.Load()を複数回実行することで、VM 内部の命令列（vm.Program）にプログラムを追記できるようにしました。これにより、Jupyter 上で最初のコードセルにサブルーチンを記述し、それを呼び出すコードを次のセルに分けて書くような使い方ができます[*19]。

▼リスト 9.18　VM.Load()メソッド

```
// Load loads code segment to VM
// return: segment number, read size, error
func (*wspace.VM) Load(code []byte) (int, int, error)
```

コードの実行は vm.Run()で、入出力に io.Readerと io.Writerを渡します。標準入出力以外を渡したいときも、これらのインターフェイスを実装することで対応できる、Go 言語ではよくある形です。

▼リスト 9.19　VM.Run()メソッド

```
// Run the program.
func (*wspace.VM) Run(ctx context.Context, in io.Reader, out io.Writer) error
```

9.5　カーネルへの組み込み

Jupyter からのコード実行リクエストは"execute_request"として Shell ソケットに届きます。{content}はリスト 9.20 のような JSON で、"code"に実行すべきコードが入っています。

[*19] セルごとに実行されてしまうので、サブルーチンを記述するセルの先頭に終了命令を置くなど工夫が必要です

▼リスト 9.20　execute_request の content

```
{
  "silent": false,
  "store_history": true,
  "user_expressions": {},
  "allow_stdin": true,
  "stop_on_error": true,
  "code": "   \t    \t\n\t\n  !"
}
```

カーネルに VM を組み込んでコードを実行するには、起動時に VM を初期化しておき、この"execute_request"ごとに vm.Load()と vm.Run()を実行することになります。これを組み込んだ Shell ハンドラはリスト 9.21 のようになります。

▼リスト 9.21　execute_request を処理する Shell ハンドラ

```
func (s *Sockets) shellHandler(vm *wspace.VM) {
    execCount := 0
    for {
        ... (略)

        // メッセージ種別による分岐
        switch hdr["msg_type"] {

        case "kernel_info_request":
            ... (略)

        case "execute_request":
            // "busy"状態に変更（処理を終えたら"idle"に戻す）
            s.sendState(msg, stateBusy)

            execCount++

            // 入力した場所から実行できるようにする
            vm.PC = len(vm.Program)
            vm.Terminated = false

            // コードの読み込み
            var content map[string]any
            _ = json.Unmarshal(msg.Content, &content)
            code := []byte(content["code"].(string))
            _, pos, err := vm.Load(code)
            if err != nil {
                s.sendStderr(msg, fmt.Sprintf("%v: %v", lineNum(code, pos), err.Error()))
                s.sendExecuteErrorReply(s.shell, msg, execCount, "LoadingError", err.Error())
                s.sendState(msg, stateIdle)
                continue
            }

            // 実行
            out := new(bytes.Buffer)
            in := &stdinReader{socks: s, parent: msg, stdout: out}
            err = vm.Run(context.Background(), in, out)
            if len(out.Bytes()) > 0 {
                s.sendStdout(msg, string(out.Bytes()))
            }
            if err != nil {
                op := vm.CurrentOpCode()
                s.sendStderr(msg,
                    fmt.Sprintf("%v: %v: %v", lineNum(code, op.Pos), op.Cmd, err.Error()))
                s.sendExecuteErrorReply(s.shell, msg, execCount, "RuntimeError", err.Error())
                s.sendState(msg, stateIdle)
                continue
            }
```

```
            s.sendExecuteOKReply(s.shell, msg, execCount)
            s.sendState(msg, stateIdle)
        }
    }
}
```

"execute_request"に限らず、Shell に届いたリクエストを処理するときは最初に状態を"busy"にし、処理を終えたら"idle"に戻します。これを忘れるとリプライが正しく反映されないことがあります。[*20]

コードの読み込み

wspace.VMは読み込んだコードを命令列 VM.Programとともに、その実行位置を指し示すプログラムカウンタ VM.PCを持っています。また、プログラム終了命令を実行したりエラーになって停止したことを示す vm.Terminatedフラグもあります。

直前の実行で停止した場合、vm.PCは最後に実行した命令を指したままですし、vm.Terminatedが trueになっていると続けて実行できません。ここでは新たに読み込んだ場所から実行してほしいので、vm.PCを読み込み済みの vm.Programの末尾を指すようにし、vm.Terminatedも falseにしておきます。

その後、送られてきたリクエストの"code"をそのまま vm.Load()で読み込みます。読み込みエラー時は stderrにメッセージを表示してから"execute_reply"をエラーとして返すのですが、この詳細は後述します。

出力

VMからの出力は標準ライブラリの bytes.Bufferで受け取るようにしました。実行中の出力をバッファリングしておき、終了後にまとめて Jupyter の標準出力に送信します。

送信に使うメソッドはリスト 9.22 の sendStdout()です。IOPub ソケットに、メッセージタイプを"stream"、{content}に出力内容を入れて送信します。

▼リスト 9.22　標準出力の送信メソッド

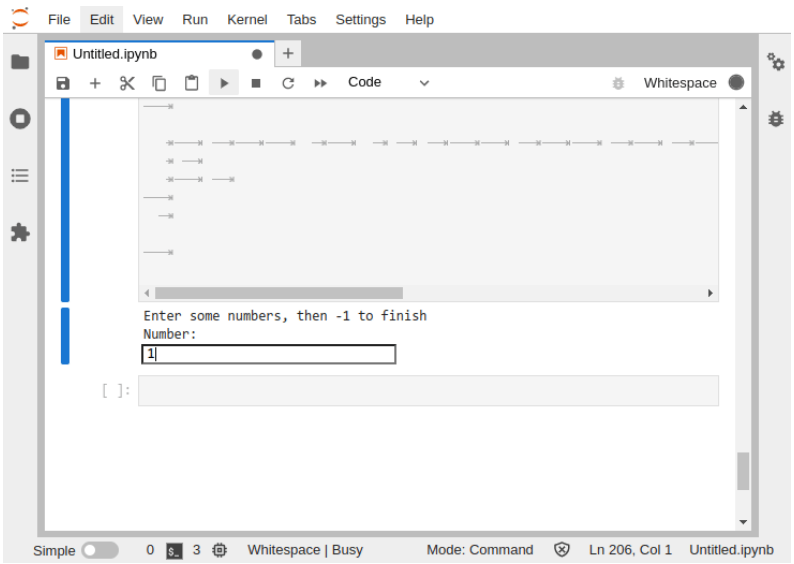
```
func (s *Sockets) sendStdout(parent *Message, output string) {
    content, _ := json.Marshal(map[string]string{
        "name": "stdout",
        "text": output,
    })
    s.send(s.iopub, parent, "stream", content)
}
```

[*20] ZeroMQ のメッセージ送信は非同期で行われるため、Shell への reply 送信と IOPub への busy/idle 通知の順序が入れ替わることがありえます。その際の挙動は未定義とされていて、ちょっと危ういシステムです。

標準エラー出力にしたいときは、{content}の"name"を"stderr"にします。これも sendStderr()として定義しました。

入力

Stdin ソケットの使い方は Shell ソケットとは逆で、カーネルから Jupyter に対してリクエストを投げます。プログラムの実行中に標準入力を受け取る必要ができた時にリクエストを投げ、それを受取った Jupyter は画面上に入力ボックスを表示します。そしてユーザの入力をリプライとして返してくるので、カーネルはそれを受け取りプログラムに伝えます。



▲図 9.2　入力ボックス

VMへの入力は io.Reader、つまり Read()メソッドをもつインターフェイスです。VM が入力を要求する命令を処理する時、この Read()メソッドを呼び出します[*21]。したがって、Read()の中で Stdin ソケットにリクエスト投げてリプライを受け取り、それを返すような型を実装することになります。

そのような型として、stdinReaderを実装しました（リスト 9.23)。

[*21] 実際の実装では効率化のため、ReadByte()も実装しています。

▼リスト 9.23　stdinReader

```
type stdinReader struct {
    socks  *Sockets
    parent *Message
    stdout *bytes.Buffer
    buf    []byte
}

func (i *stdinReader) Read(p []byte) (int, error) {
    // stdout のフラッシュ
    if out := i.stdout.Bytes(); len(out) > 0 {
        i.socks.sendStdout(i.parent, string(out))
        i.stdout.Reset()
    }

    buf := i.buf
    if len(buf) == 0 {
        // stdin を Jupyter に要求し受け取る
        b, err := i.socks.getStdin(i.parent)
        if err != nil {
            return 0, err
        }
        buf = b
    }
    n := copy(p, buf)
    i.buf = buf[n:]
    return n, nil
}
```

Read()メソッドで最初に行っているのは、出力のフラッシュ処理です。入力を求めるプログラムでは大抵、何を入力するのか示す文字列を出力してから入力を受け付けます。たとえば Whitespace のサンプルの Calculator[*22]では、次のように表示しています。このような表示を先に出力するために、バッファリングされている出力をフラッシュするようにしました。

```
$ wspace calc.ws
Enter some numbers, then -1 to finish
Number:
```

入力の要求と取得をしているのは、ちょうど中央あたりの getStdin()メソッドです。この中で Stdin ソケットと通信しています。

Whitespace で文字の入力を受け取るには、1 文字ずつ読む命令を使います。しかし、Jupyter での入力テキストボックスは 1 行単位で入力するようになっているので、毎回入力を要求して 1 文字しか使わないのは直感に反しますし、非効率です。なので、ここでは受取った入力をバッファリングし、バッファに入力が残っているときは Jupyter への要求はせずにバッファの内容を切り出して返すようにしています。

一方、このバッファリングされた入力は、次の"execute_request"には引き継ぎません。"execute_request"はノートブックのセル単位で行われ、入力のテキストボックスもそのセルのすぐ下の入出力エリアに表示されます。このため、前のセルの実行時の入力が混ざってしまうのは望ましくないと考え、stdinReaderは"execute_request"ごとに

[*22] https://web.archive.org/web/20150717115008/http://compsoc.dur.ac.uk/whitespace/calc.ws

初期化するようにしました。

続いて、Stdin ソケットで入力を要求し受け取る getStdin()をリスト 9.24 に示します。

▼リスト 9.24　Stdin ソケットで通信するメソッド

```
func (s *Sockets) getStdin(parent *Message) ([]byte, error) {
    s.sendRouter(s.stdin, parent, "input_request", []byte(`{"prompt":"","password":false}`))
    msg, err := s.recvRouterMessage(s.stdin)
    if err != nil {
        return nil, err
    }
    var d map[string]string
    _ = json.Unmarshal(msg.Content, &d)
    return append([]byte(d["value"]), '\n'), nil
}
```

Stdin ソケットは ROUTER なので、メッセージを書き込むには ZmqID が必要です。ここでは Shell ソケットで受信した"execute_request"の ZmqID と同じものを設定すれば大丈夫です。というのも、Jupyter 側で Stdin には Shell と同じ ZmqID を設定しているためです。

リクエストメッセージの msg_typeは"input_request"で、{content}は"prompt"文字列と"password"フラグを指定します。このメッセージを、Shell と同じように sendRouterMessage()で送信します。すると Jupyter 上で入力のテキストボックスが表示されます。

テキストボックスに入力してエンターキーを押すと、Stdin ソケットに"input_reply"メッセージが届きます。{content}はリスト 9.25 のようになっています。

▼リスト 9.25　"input_reply"の{content}

```
{
  "status": "ok",
  "value": "hello"
}
```

入力値の"value"には末尾に改行は付いていません。Whitespace では、数値の入力では末尾に改行（または EOF）を要求します。また、一般的な標準入力では、大抵のターミナルで行単位で末尾の改行を含めて入力されます。この挙動に合わせたほうが都合がよいので、改行文字'\n'を末尾に追加して入力値としました。

"execute_reply"

コードの実行が終わったら"execute_reply"を送信します。{content}はリスト 9.26 のような JSON です。"execution_count"は Jupyter 上で実行したコードの左に表示される番号です。"execute_request"を処理する毎に execCountをインクリメントしてこの値としています。

▼リスト 9.26　execute_reply の content

```
{
    "status: "ok",
    "execution_count": 1
}
```

エラー時は"status"を"error"にするほか、エラーの名前と内容を示す "ename" "evalue" などのフィールドを加えますが、Jupyter 上には表示されないようです。ユーザーに見せるメッセージは stderrへ出力するようにしましょう。

▼リスト 9.27　"execute_reply"を送信するメソッド

```
func (s *Sockets) sendExecuteOKReply(sock *zmq4.Socket, parent *Message, count int) {
    content := fmt.Sprintf(`{"status":"ok","execution_count":%d}`, count)
    s.sendRouter(sock, parent, "execute_reply", []byte(content))
}

func (s *Sockets) sendExecuteErrorReply(
    sock *zmq4.Socket, parent *Message, count int, ename, evalue string) {

    content, _ := json.Marshal(map[string]any{
        "status":          "error",
        "execution_count": count,
        "ename":           ename,
        "evalue":          evalue,
        "traceback":       []any{},
    })
    s.sendRouter(sock, parent, "execute_reply", content)
}
```

これで Whitespace を Jupyter 上で実行できるようになりました。余談ですが、Jupyter は空文字列しかないセルは実行してくれません（"execute_request"を送信してくれません）。Whitespace のコードを実行するときは最低 1 文字は見える文字を混ぜておく必要があります。

9.6　おわりに

この章では、Jupyter のカーネルの実装方法を、Whitespace のカーネル「whitenote」のコードを使って解説しました。細かいお約束が多いため長くなってしまいましたが、必要な実装はそれほど多くなく、シンプルな仕組みになっていることが分っていただけたと思います。

ぜひ皆さんも、お気に入りの言語の Jupyter カーネルを自作してみてください。

タブ文字を入力するには

Jupyter のコードセルで Whitespace のコードを入力しようとすると、タブキーを押してもタブ文字が入力されないことに気づくと思います。これは、タブキーがコード補完に割り当てられているためです。
Jupyter がコードを補完するとき、カーネルには`"complete_request"`が送られます。ここで補完候補を複数返すと Jupyter 上で選択する UI が表示されますが、候補が 1 つしかないときは直接それが入力されます。
つまり、`"complete_request"`に対してタブ文字だけを補完候補として返すことで、タブ文字を入力できるようになります。実装の詳細は whitenote のリポジトリをご覧ください。
ただし、行頭から空白文字しかない場合は補完ではなく、自動インデントになってしまいます。（しかもタブ文字ではなくスペースで！）この挙動は Javascript の CodeMirror[*23]によるもので、カーネルからは挙動を変えられそうにはありません。
Whitespace を記述するときは行頭になにか見える文字を入力しておくと快適に入力できます。

[*23] https://codemirror.net/

執筆者・スタッフコメント

第 1 章 Yoshio HANAWA / @hnw
次のブラックフライデーにストックを増やしたい

第 2 章 KOBAYASHI Yū
Rust はいいぞ

第 3 章 Tomoaki Fude
USB のセキュリティは今が黎明期なのかも

第 4 章 Togo Kosaka
全部レイキャストすれば解決

第 5 章 Lingjian Wang
マルチスレッドで経路探索

第 6 章 Shunsuke Ito
Python 3.11 の Exception Groups もおもしろそう

第 7 章 Shinya Naganuma / @Pctg_x8
Web フロントなんもわからん

第 8 章 Naoki Hamada / @hmkz_
ベジエ単体ってぷにぷにしてて可愛いです

第 9 章 Daisuke Makiuchi / @makki_d
眼鏡っ娘が好きです

企画進行・イラスト・デザイン

Toshifumi Umezawa

代表者として企画進行を担当しました。〆切が早まる分厚さになるかヒヤヒヤしました

Sumire Amano

表紙イラストを描かせて頂きました！ 眼鏡っ娘、楽しく描けました！ 今夜は鮎の塩焼きです

Asahi Komatsu

扉絵など担当しました！ 夜ごはんはしらす丼です

既刊・電子版ダウンロード

https://www.klab.com/jp/blog/tech/2022/tbf13.html

KLab Tech Book Vol. 10

2022 年 9 月 10 日　技術書典 13 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


KLabTechBook