Discord

クラブテックブック

Club Tech Book

Vol. 7

Unity

Go

Ice

Mesh

Raspberry

1 Raspberry Piで作る自作Linuxデバイスドライバー

2 Raspberry PiとGoogle Meetでお手軽ペットカメラ

3 電子工作で重さを量る

4 Unityエディタ拡張のプログラム設計
～業務で使えるツールの作り方～

5 itertools repeatを使ってみて読んでみる

6 DiscordとSlackの架け橋

7 会社の有志で技術同人誌を書き続けている話

KLab Tech Book Vol.7

2020-12-26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 7 弾です。

KLab 株式会社では主にスマートフォン向けのゲームを開発していますが、本書ではこれまでどおり社内で執筆者を募り、著者の興味や得意分野をベースに各自好きな技術について好きなように書き執筆者同士（＋α）レビューを行なった記事を収録しています。業務に少しだけ関係のあることもあり、完全に趣味の内容もあります。

第 7 弾ともなると継続して執筆に参加してきた著者もいます。特定の技術領域について継続して執筆している著者がいたり、回ごとに執筆する内容の技術領域を変えている著者もいたり。また今回も、はじめて技術同人誌を書いてみたという著者がいます。気になった記事の著者が過去執筆していたか？ 過去執筆していた記事でどんな内容を書いていたのか？ 巻末の QR コードから追いかけてみるのも楽しいのではないかと思います。

また既にご覧頂いている表紙、中扉ですが、これも今までの KLab Tech Book 同様、社内のイラストレーター、デザイナーが世界観やキャラクターをイチから制作してくれたものです。今回は、読者のみなさまに少しでも楽しんでいただくために、表紙では「エンタメの力でコロナと戦う」、中扉では「リモートワーク」というテーマで制作いただき、その他のページにも表紙と合わせた明るくにぎやかなデザインを入れて頂きました。本書の最後にはそれらの表紙や中扉などがどのように作られたか紹介する記事もあります。

表紙も本文も、そして作られた過程も。一冊まるごと楽しんで頂ければさいわいです。

水澤 絹子

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。

contents

KLab Tech Book
Vol.7 2020-2021 WINTER

第 1 章

Raspberry Pi で作る自作 Linux デバイスドライバー

Ryota Togai / @garicchi

みなさんはデバイスドライバーと聞いて何を想像するでしょうか。

パソコンが映像をディスプレイに出力できたり、キーボードから文字を入力できるのはデバイスドライバーのおかげです。また、特殊な周辺機器を購入した時には、それらの周辺機器をパソコンに接続するときに周辺機器メーカーが配布しているデバイスドライバーをインストールされたりもしています。

デバイスドライバーの一般的な説明としては、「OS から見えるデバイスの操作を抽象化するもの」となるでしょう。デバイスドライバーというものは我々の生活を確かに支えるものではあるのですが、その実体のプログラムを見る機会はなかなかなく、その中身で何が行われているのかイメージを正確に持たれている方はそう多くないのではないかと思います。

そこで、本稿の執筆では、「簡単な Linux デバイスドライバーを作ってみることで Linux がどのように周辺機器を操作しているのかを学び、OS のカーネルへの理解を深めたい」「そのようにして学び理解した経過を記事にまとめることで、読者のみなさまにもデバイスドライバーのイメージをつける手助けがしたい」そのように考えました。

デバイスドライバーを作ってみる Linux 環境としては、手軽に扱える Raspberry Pi を選択しています。本稿をお読み頂くことで、きっとデバイスドライバーのイメージをつけて頂くことが出来ると思います。

それでは、始めていきましょう。

1.1 準備

はじめに

本稿は下記記事および書籍を大変参考にさせていただきました。ぜひ本稿にあわせて、下記資料をご参照ください。

- 組み込み Linux デバイスドライバーの作り方, @iwatake2222 著, Qiita[*1]
- Raspberry Pi で学ぶ ARM デバイスドライバープログラミング - 米田聡著, ソシム[*2]

ハードウエアを準備する

今回、Raspberry Pi の GPIO を操作するデバイスドライバーを作成し、GPIO への入出力を確認します。使用した部品を表 1.1 に示します。実際に動作を試される方は参照してください。どれも一般的な電子工作ショップで購入することができます。

▼表 1.1　使用した部品一覧

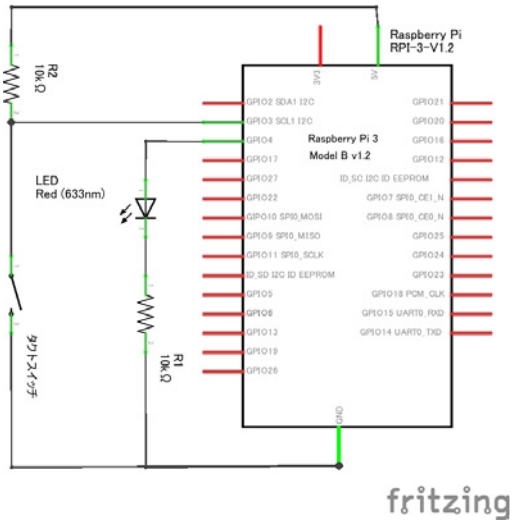
部品名	個数	説明
Raspberry Pi 3 Model B+[*3]	1	デバイスドライバーを動かすコンピュータです。[*4]
MicroSD カード	1	Raspberry Pi の OS を書き込む用
LED	1	出力確認用。普通の LED で大丈夫です。
抵抗	2	LED に十分な電圧がかかればなんでも OK です。筆者は 10k Ω± 5% を使用しました。
タクトスイッチ	1	入力確認用
ブレッドボード	1	回路作成用
ジャンパーワイヤー	必要な分	回路作成用

部品を図 1.1 のように接続し、回路を完成させます。LED は GPIO4 へ接続し、GPIO4 を HIGH にすることで点灯します。タクトスイッチは GPIO3 へプルアップ抵抗とともに接続し、スイッチが押されていない状態で GPIO3 が HIGH、スイッチが押されている状態で GPIO3 が LOW になります。

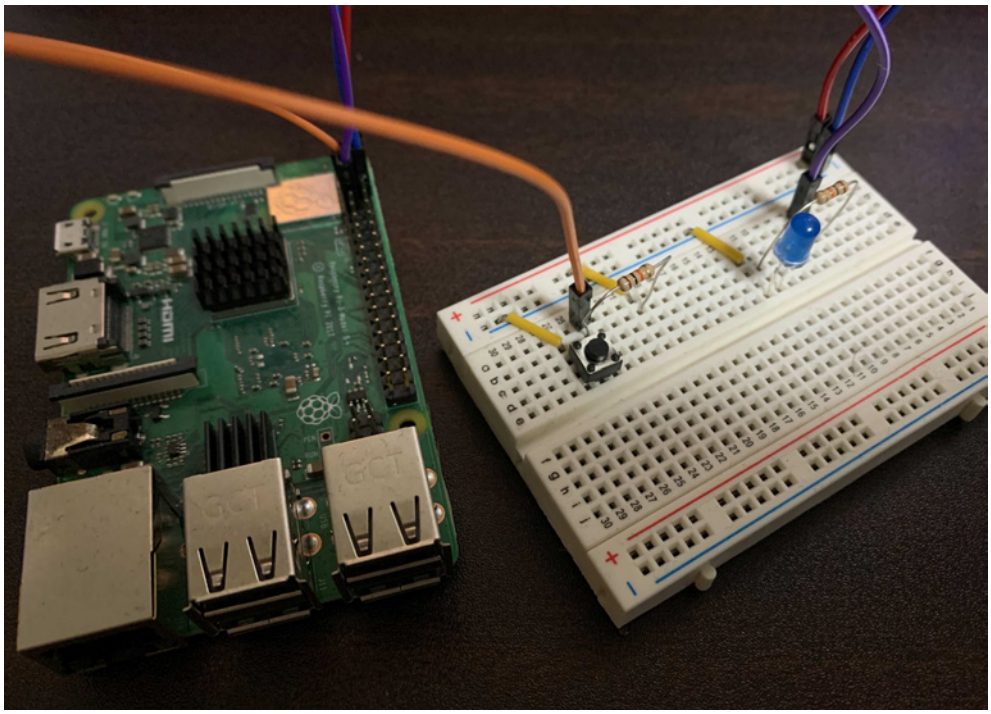
[*1] https://qiita.com/iwatake2222/items/1fdd2e0faaaa868a2db2

[*2] https://www.socym.co.jp/book/940

[*3] 他バージョンの Raspberry Pi でもよいかもしれませんが、デバイスドライバーはハードウエアに強く依存するため、動作しない可能性もあります。



▲図 1.1　回路図



▲図 1.2　回路の写真

Raspberry Pi OS は公式 Imager[*5]などを使用し、32bit のイメージを micro SD カードに書き込み、起動させます。その後、ネットワークに接続し、作業用の PC から ssh できる状態にしておきます。

今回は、Raspberry Pi 上でソースコードのコンパイルを行っています。したがって、Raspberry Pi 上でソースコードを書いても問題はなかったのですが、利便性を考えると慣れた作業用 PC でコードを書き、rsync などで Raspberry Pi に転送する方が良いと考え、Raspberry Pi ではない使い慣れた作業 PC で実装を進めました。

サンプルソースコード

本稿では動作するサンプルコードの解説に必要な一部だけ掲載します。完全なサンプルコードは `https://github.com/garicchi/pi-gpio-driver` に置いてありますので適宜参照してください。

回路が正しく動くことを確認する

回路が完成したのでデバイスドライバーを早速作っていきましょう！……という前に、やるべきことがあります。部品が壊れていたり、回路が間違っている可能性があるので、作成した回路が本当に正しく動くのかを確認する必要があります。デバイスドライバーはデバッグが比較的難しいプログラムです。いきなりデバイスドライバーを動かしてしまうと、想定通りに動かなかったときにデバイスドライバーの問題なのか回路の問題なのかを判別するのが難しくなります。

幸いにも Raspberry Pi の GPIO はユーザー空間のプロセスから制御でき、公式に Python で書かれた非常にシンプルなサンプルがあります[*6]。まずはこのプログラムを利用して GPIO を操作し、回路が本当に正しいかをチェックしましょう。

筆者が用意したサンプルコードは gpio_easy フォルダ[*7]以下にあります。

まずは output.py を実行しましょう。なお、筆者のサンプルは公式サンプルコードから少し変えているので注意してください。

▼リスト 1.1　ホスト PC から Raspberry Pi へコードの転送（ホスト PC 上）

```
cd path/to/raspi-gpio-driver
RASPI_HOST=<自分の raspi の IP アドレス or ホスト名に書き換える>
rsync -rv --delete . ${RASPI_HOST}:~/pi-gpio-driver
```

▼リスト 1.2　出力テストプログラム実行（Raspberry Pi 上）

[*5] `https://www.raspberrypi.org/software`
[*6] `https://www.raspberrypi.org/documentation/usage/gpio/python/README.md`
[*7] `https://github.com/garicchi/pi-gpio-driver/tree/main/gpio_easy`

```
python3 ./pi-gpio-driver/gpio_easy/output.py
```

LED が点滅すれば成功です。
続いて input.py を実行しましょう。

▼リスト 1.3　入力テストプログラム実行（Raspberry Pi 上）

```
python3 ./pi-gpio-driver/gpio_easy/input.py
```

タクトスイッチを押すまでは Released と表示され続け、タクトスイッチを押している間は Pressed と表示されているなら成功です。
ここまでで想定通りの挙動が確認できなかった場合は、回路や部品を今一度確認してみましょう。

1.2　何もしないデバイスドライバーを作る

回路が正しいことを確認できたので、デバイスドライバーを作っていきましょう。まずは GPIO もなにも制御せず、ログに情報を出力するだけのデバイスドライバーを作成してみましょう。

カーネルヘッダのインストール

本稿で作るデバイスドライバーは Raspberry Pi 上でコンパイルします。本来であれば、もっと CPU の速い PC でコンパイルを行いたいところですが、多くの方の作業用 PC の CPU は x64 でしょう。一方、Raspberry Pi は ARM なので、クロスコンパイルを行わなければなりません。環境構築も人によって方法が変わってしまうので、今回は Raspberry Pi 上でコンパイルを行うことにします。
Raspberry Pi 上でデバイスドライバーのコンパイルを行うためにはリスト 1.4 のように apt で kernel header をインストールする必要があります。

▼リスト 1.4　カーネルヘッダのインストール（Raspberry Pi 上）

```
sudo apt install raspberrypi-kernel-headers
```

デバイスドライバーとカーネルモジュール

デバイスドライバーは OS と周辺機器との中間に入り、周辺機器を制御する方法を OS に提供するプログラムになります。そして、ユーザー空間で動くのではなく OS のカーネ

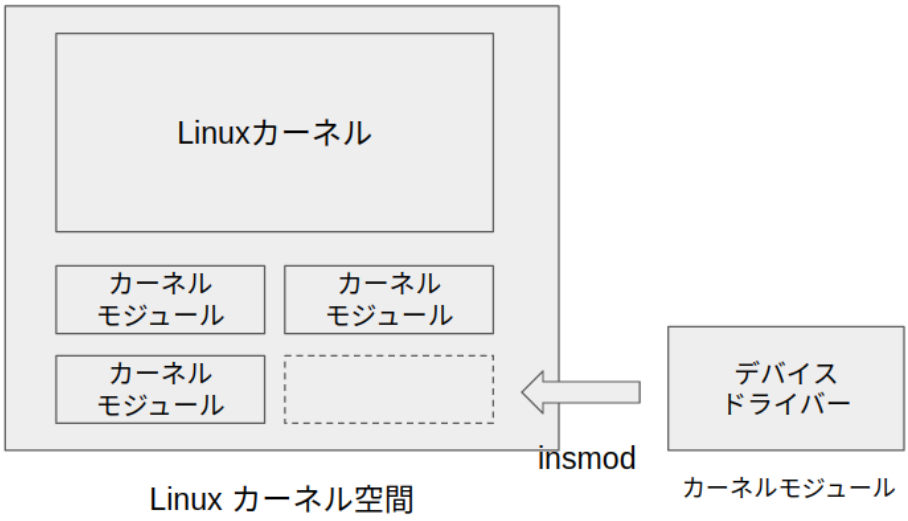
ル空間で動く、カーネルモジュールとして作成します。なぜユーザー空間で動くようにデバイスドライバーを作らないのでしょうか。

まず 1 つめの観点として、仮想メモリがあります。デバイスによってはメモリマップド I/O、すなわち特定の物理メモリアドレスの値を読み書きすることによるデバイスの制御ができる場合があります。しかし、近代的な OS はユーザー空間のプログラムに対し、メモリ使用の効率化や安全性の観点から、物理メモリではなく仮想メモリを提供しています。仮想メモリではユーザー空間のプロセスが許可されていないアドレスへアクセスできないようにすることで安全性を保ちます。メモリマップド I/O は物理メモリ上の特定アドレスにアクセスしなければいけないため、ユーザー空間で実行するとメモリアクセス違反になる恐れがあります。したがって、デバイスドライバーは OS のカーネル空間で動作することで物理メモリアドレスを直接参照し、物理メモリにマップされた周辺機器を制御します。

2 つめの観点として、プロセススケジューリングと割り込みがあります。ユーザー空間上で様々なプロセスが同時に動いていた時、どのプロセスがいつ CPU を使用できるかは OS が管理しています。これをプロセススケジューリングと呼び、Linux の場合、なるべく多くのプロセスがなるべく同じ量の時間 CPU を使えるようにスケジューリングしています。したがって、ユーザー空間に大量のプロセスがいた場合は、その分 CPU を使用できる時間は短くなり、順番待ちが長くなってしまいます。しかし例えばキーボードなどの周辺機器は、ユーザーの入力に対しすばやく応答してほしいです。デバイスドライバーは、ユーザー空間にどんなプロセスがあったとしてもすばやく応答する必要があります。そのために、OS は割り込みという仕組みを用意していますが、この割り込みを使うためにはカーネル空間のプロセスでないといけません。したがって、デバイスドライバーはカーネル空間で動作し、OS の割り込みを利用してリアルタイムに周辺機器を制御します。

上記の観点から、周辺機器を制御するデバイスドライバーは OS のカーネル空間で動作しないといけません。しかし、カーネル空間で独自のコードを走らせるためだけに OS のカーネルコードを変更し、全体をコンパイルして入れ替えるのは、あまりにもハードルが高いです。そこで Linux では**カーネルモジュール**という仕組みを用意しています。カーネルモジュールとは、既に動いているカーネルに対し、独自のプログラムを動的に組み込んだり外したりすることができる仕組みです。

Linux のデバイスドライバーはカーネルモジュールとして作成し、既に動いているカーネルに組み込みます。こうすることで、OS のカーネルを 1 からコンパイルすることなく、カーネル空間で独自コードを動かすことができます。

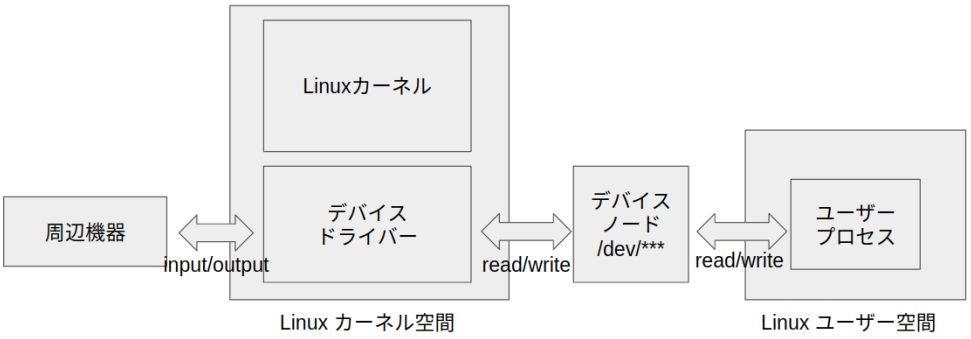


▲図 1.3　カーネル空間とカーネルモジュール

デバイスノードと udev

Linux はあらゆる操作や状態をファイルとして表現します。周辺機器に関しても、/dev/以下の特殊ファイルを読み書き（read/write）することで操作します。この/dev/以下のデバイスを操作するための特殊ファイルのことを**デバイスノード**と呼びます。

デバイスノードは read/write などの操作を認識すると、それをデバイスドライバー（カーネルモジュール）へ伝え、デバイスドライバーの特定の関数がコールされます。そして、デバイスドライバーが周辺機器を制御することで、ユーザによる周辺機器の操作が実現されます。



▲図 1.4　デバイスノード

デバイスノードは何もしなくても生成されるわけではありません。そのデバイスドライバーが扱うデバイスの数だけ、手動で生成する必要があります。とはいえ、毎回デバイスノードを作るのは大変なので、**udev** というデバイスノードを自動で作ってくれる仕組みがあります。

今回は udev を使用して、デバイスドライバーが初期化されたタイミングで 2 個のデバイスノードを自動生成してみましょう。

何もしないデバイスドライバーコードの全体像

リスト 1.5 に何もしないドライバーのコードの概観を示します。これはデバイスドライバーというよりは Linux のカーネルモジュールの雛形になります。`hoge`の部分は好きな名前に置き換えてもらって問題ありません。基本的にはデバイスノードに対するそれぞれの操作（open/read/write/release）に対し、対応する関数を作成します。それぞれの関数では、コールされたことを確認するために文字列を print しています。`printk`はカーネル空間内で実行できる print 出力関数です。出力結果は`/var/log/syslog`などで確認できます。

`hoge_init`と `hoge_exit`関数に関しては、`module_init()`と `module_exit()`に与えることで、デバイスドライバーのインストール、アンインストール時に呼ばれます。

▼リスト 1.5　何もしないドライバーのコードの概観

```
#include <linux/init.h>

// 省略

// デバイス open 時の処理
static int hoge_open(struct inode *inode, struct file *file)
{
    printk("[%s] open", DRIVER_NAME);
    return 0;
```

```
}

// デバイス release 時の処理
static int hoge_release(struct inode *inode, struct file *file)
{
    printk("[%s] release", DRIVER_NAME);
    return 0;
}

// デバイス read 時の処理
static ssize_t hoge_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk("[%s] read", DRIVER_NAME);
    buf[0] = 'A';
    return 1;
}

// デバイス write 時の処理
static ssize_t hoge_write(struct file *filp, const char __user *buf,
    size_t count, loff_t *f_pos)
{
    printk("[%s] write", DRIVER_NAME);
    return 1;
}

// 各システムコールのハンドラを構造体にまとめる
struct file_operations hoge_fops = {
    .open = hoge_open,
    .release = hoge_release,
    .read = hoge_read,
    .write = hoge_write,
};

// insmod 時に呼ばれる初期化関数
static int hoge_init(void)
{
    // 省略
}

// rmmod 時に呼ばれる終了関数
static void hoge_exit(void)
{
    // 省略
}

module_init(hoge_init);
module_exit(hoge_exit);
```

デバイスドライバーの初期化関数である`hoge_init`関数では、デバイスドライバの初期化と udev によるデバイスノードの作成をします。

すべてをここに掲載すると冗長になってしまうので、サンプルコードの empty_driver/driver.c[*8]を参照してください。ここでは重要な関数を抜き出して説明します。

まず、デバイスドライバーのメジャー番号を決定します。デバイスノードにはメジャー番号とマイナー番号が存在します。メジャー番号はデバイスの種類を表し、どのデバイスドライバーと対応するかを識別します。マイナー番号は同じ種類のデバイスの中の何番目のノードかを表します。今回作成するデバイスドライバーは独自の種類にしたいため、他と衝突しない番号を選ぶ必要があります。このため、`alloc_chrdev_region`関数で OS

[*8] `https://github.com/garicchi/pi-gpio-driver/blob/main/empty_driver/driver.c`

に空いているメジャー番号を問い合わせています。

▼リスト 1.6　メジャー番号確保関数

```
dev_t dev; // major 番号と minor 番号をまとめた構造体
// メジャー番号を動的に確保する
int r_alloc = alloc_chrdev_region(&dev, minor_base, minor_num, DRIVER_NAME);
major = MAJOR(dev); // major 番号を取り出して static に保持しておく
```

次に各ハンドラの登録と、デバイスの初期化を行います。各ハンドラとして、open/write/read/release に対応する関数を定義し hoge_fopsに格納しました。これを cdev_init関数にアドレスを渡し、ハンドラの登録を行います。

▼リスト 1.7　キャラクタデバイス初期化関数

```
// 各ハンドラともにキャラクタデバイスを初期化する
cdev_init(&hoge_cdev, &hoge_fops);
```

続いて、cdev_add関数で作成したキャラクタデバイスをカーネルへ登録します。

▼リスト 1.8　キャラクタデバイスのカーネルへの登録

```
dev = MKDEV(major, minor_base);
int r_cdev = cdev_add(&hoge_cdev, dev, minor_num);
```

最後に、udev が認識できるようデバイスクラスを登録し、デバイスノードを作成します。これで、/dev/hoge0と/dev/hoge1が作成されます。

▼リスト 1.9　デバイスノードの作成

```
// /sys/class/hoge/ を作る
hoge_class = class_create(THIS_MODULE, DRIVER_NAME);
// デバイスノードを作る
// /sys/class/hoge/hoge* を作る
for (int i = minor_base; i < (minor_base + minor_num); i++) {
    device_create(hoge_class, NULL, MKDEV(major, i), NULL, "hoge%d", i);
}
```

hoge_exit関数での処理は、hoge_init関数とは逆に各種登録を解除するだけなので説明は割愛します。こうしてデバイスドライバーの初期化関数にて、デバイスドライバーの初期化、ハンドラ関数の登録、デバイスノードの作成ができました。

何もしないデバイスドライバーを動かしてみる

では先ほど解説した何もしないデバイスドライバーを実際にコンパイルして動かしてみましょう。リスト 1.10 にコンパイルコマンドを示します。

▼リスト 1.10　何もしないドライバーのコンパイル（Raspberry Pi 上）

```
cd ~/pi-gpio-driver/empty_driver
make
```

Makefile はリスト 1.11 のようになっており、カーネルが用意している Makefile を利用することでカーネルモジュールをコンパイルしています。

▼リスト 1.11　何もしないドライバーの Makefile

```
ccflags-y += -std=gnu99 -Wno-declaration-after-statement

CFILES = driver.c

obj-m := mymodule.o
mymodule-objs := $(CFILES:.c=.o)

build:
        make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
```

コンパイルが完了したらリスト 1.12 に示すコマンドでデバイスドライバーのインストールを行いましょう。/etc/udev/rules.d/にルールファイルを作っているのは、デフォルトでは作成されたデバイスノードに root 権限でしか書き込むことができないのでユーザー権限で読み書きできるようにファイル権限のルールを追加しています。

▼リスト 1.12　何もしないドライバーのインストール（Raspberry Pi 上）

```
echo 'KERNEL=="hoge[0-9]*", GROUP="root", MODE="0666"' | \
      sudo tee -a /etc/udev/rules.d/mymodule.rules
sudo insmod mymodule.ko
```

正常に成功すればデバイスドライバーのインストールは完了です。/dev/以下に hoge0 と hoge1というデバイスノードができているはずです。

デバイスノードが完成したら、さっそく write を試してみましょう。その後、/var/log/syslogを確認し、リスト 1.5 で printkした文字列が表示されるか確認しましょう。

▼リスト 1.13　デバイスドライバーのログ表示（Raspberry Pi 上）

```
# write の動作確認をしてみる
echo 1 > /dev/hoge0

# ログの確認
tail -n 20 /var/log/syslog

# 次のように表示される
kernel: [27088.021918] [hoge] write
```

何もしないデバイスドライバーを作成し、デバイスノードを作成することができました。また、デバイスノードに対し write を行うと、作成したデバイスドライバーの `hoge_write`関数がコールされていることが確認できました。read についても `cat`コマンドなどで試してみてください。うまくいけば無限に `A`が表示されるはずです。

最後に、デバイスドライバーのアンインストールを行いましょう（リスト 1.14）。

▼リスト 1.14　何もしないドライバーのアンインストール（Raspberry Pi 上）

```
sudo rmmod mymodule.ko
sudo rm -v /etc/udev/rules.d/mymodule.rules
```

これで、何もしないデバイスドライバーを作ることができました。

1.3　GPIO を使用するデバイスドライバーを作る

先程までで、デバイスドライバーの雛形を作ることができました。あとは Raspberry Pi のハードウエアの仕様に則り、正しく入出力をすることができれば目的のデバイスドライバーの完成です。

メモリマップド I/O

Raspberry Pi は GPIO への入出力が特定の**メモリアドレス**（レジスタとも呼ぶ）に割り当ててあり、そのアドレスの読み書きをすることで GPIO を制御することができます。

このように入出力が特定のメモリアドレスに割り当てられている機構を、**メモリマップド I/O** と呼びます。本稿でも、デバイスドライバーから特定のアドレスの値を書き換え、GPIO を制御することとします。

Broadcom BCM2837 のメモリマップ

ではどのアドレスがどの GPIO に割り当てられているのでしょうか。その答えは SoC のデータシートを見ることでわかります。今回使用する Raspberry Pi 3 Model B+ の SoC は Broadcom BCM2837 という製品名のものです。

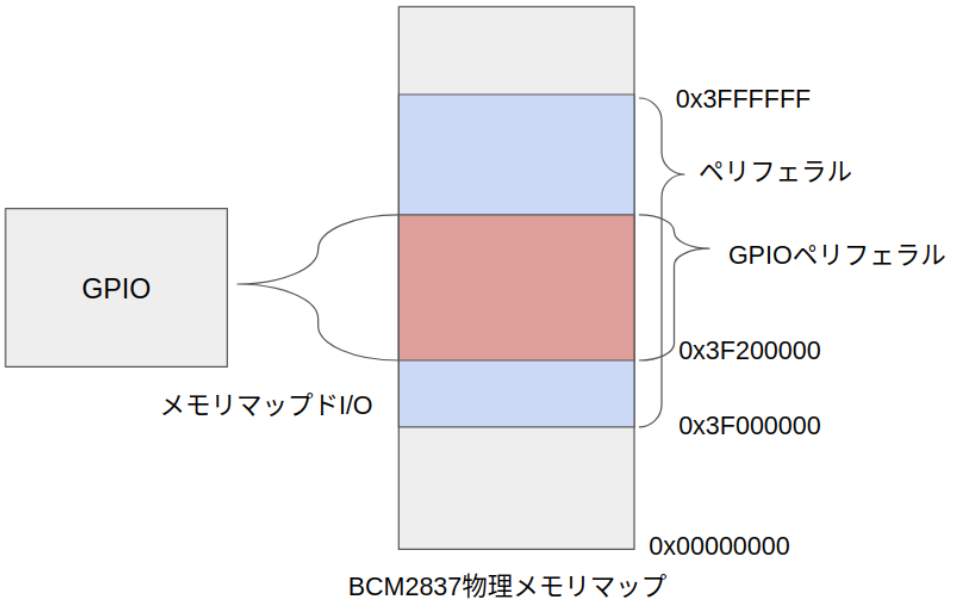
GPIO ペリフェラルアドレス

まず、BCM2837 の**ペリフェラルアドレス**がどこかを知る必要があります。ペリフェラルアドレスとは、周辺機器がマップされるメモリアドレスの範囲です。BCM2837 のデータシート[*9]を確認すると、IO ペリフェラルアドレスが `0x3F000000`からと書いてあります。

> Physical addresses range from 0x3F000000 to 0x3FFFFFFF for peripherals.

したがって、Raspberry Pi 3 Model B+ は `0x3F000000`のアドレスから[*10]周辺機器がマップされていることになります。

さらに、周辺機器の 1 つである GPIO がマップされている位置も把握する必要があります。GPIO が割り当てられているアドレスは、ペリフェラルの開始アドレスから `0x00200000`進めた位置にあります。



▲図 1.5　BCM2837 物理メモリマップ

ペリフェラルの開始アドレスを `REG_ADDR_BASE`、GPIO の開始アドレスを `REG_ADDR_GPIO_BASE`とすると、リスト 1.15 のようになります。

[*9] `https://cs140e.sergio.bz/docs/BCM2837-ARM-Peripherals.pdf`
[*10] Raspberry Pi のバージョンによって異なる可能性があります。

▼リスト 1.15　ペリフェラルアドレス

```
REG_ADDR_BASE = 0x3F000000
REG_ADDR_GPIO_BASE = REG_ADDR_BASE + 0x00200000
```

GPIO Function Select レジスタ

GPIO ピンは様々な機能を持っており、事前にどのピンを何に使うかを設定しておく必要があります。その設定に使うレジスタは複数存在します。表 1.2 にその中から 4 つ抜き出したものを示します。

▼表 1.2　GPIO Function Select レジスタ

レジスタ名	アドレス
GPFSEL_0	0x3F200000
GPFSEL_1	0x3F200004
GPFSEL_2	0x3F200008
GPFSEL_3	0x3F20000C

4 つの GPFSEL レジスタの中に 3 ビットずつ、各 GPIO ピンをどう使うかのデータを入れます。すべてを示すのは大変なので、今回使用したい GPIO ピンのみに絞ります。今回は GPIO4 を出力として使用します。GPIO4 は GPFSEL_0の 12 ビット〜14 ビットに値を入れて設定します。値は、0x0を入れると入力として使用し、0x1を入れると出力として使用することになります。したがって、GPIO4 を出力として使用したい場合、0x3F200000（GPFSEL_0）の 12 ビット目に 1を入れれば良いということになります。

GPIO Pin Set レジスタと GPIO Pin Clear レジスタ

GPIO Function Select レジスタはあくまで、どの GPIO ピンを何の機能に使うかを設定するものでした。**GPIO Pin Set レジスタ**はどの GPIO ピンを電圧 High に設定するかを設定し、**GPIO Pin Clear レジスタ**はどの GPIO ピンを電圧 Low にするかを設定するレジスタになります。レジスタのアドレスは表 1.3 になります。

▼表 1.3　GPIO Pin Set/Clear レジスタ

レジスタ名	アドレス
GPIO Pin Set レジスタ	0x3F20001C
GPIO Pin Clear レジスタ	0x3F200028

GPIO Pin Set レジスタと GPIO Pin Clear レジスタはそれぞれ 1 つしかなく、レジスタの各ビットが GPIO の各ピンに相当します。したがって、GPIO4 を High にしたい場合は 0x3F20001C（GPIO Pin Set レジスタ）の 4 ビット目を立てれば良いことになります。

GPIO Pin Level レジスタ

GPIO Pin Level レジスタは GPIO Pin Set レジスタとは逆に、入力を扱うレジスタになります。アドレスは表 1.4 になります。

▼表 1.4　GPIO Pin Level レジスタ

レジスタ名	アドレス
GPIO Pin Level レジスタ	0x3F200034

GPIO Pin Level の各ビットが GPIO の各ピンの入力状態に相当します。

したがって、GPIO3 の電圧が High か Low かは、0x3F200034（GPIO Pin Level レジスタ）の 3 ビット目が 1 か 0 かを見れば良いことになります。

レジスタのアドレスを知る方法

上記のように、Raspberry Pi は GPIO がメモリの指定のアドレスに割り当てられており、アドレスさえわかればデータを読み書きするだけで入出力を行うことができます。

しかしながら、SoC のデータシートからこれらの情報を読み取るのは非常に困難です。冒頭にも紹介しました、Raspberry Pi で学ぶ ARM デバイスドライバープログラミング - 米田聡著, ソシム[*11]という書籍にこの辺の知識が載っていますので別の GPIO ピンを使用したいなどありましたら参照してください。

また、将来新しいバージョンの Raspberry Pi が誕生した場合、このアドレスが変化する可能性が大いにあります。その場合は各レジスタの役割を理解したうえで、データシートから適宜読み取っていただけるとよいかと思います。

1.4　GPIO ドライバーを動かす

GPIO 入出力に必要なアドレスは揃いました。あとはこれらのレジスタをデバイスドライバーから読み書きするだけで GPIO 入出力ができるはずです。

ソースコードはサンプルコード内の gpio_driver/driver.c[*12]です。

何もしないデバイスドライバーの雛形から変わった点としては、ペリフェラルアドレスを define したのと、open、write、read 関数の中身を実装したことです。

ソースコードの冒頭部分でリスト 1.16 のように、先ほど調べた各種レジスタのアドレスを define しています。各種レジスタのアドレスが先ほど調べた値と一致していることがわかります。

▼リスト 1.16　レジスタアドレスの define

[*11] https://www.socym.co.jp/book/940

[*12] https://github.com/garicchi/pi-gpio-driver/blob/main/gpio_driver/driver.c

```
// Raspberry Pi のメモリマップド I/O アドレス
#define REG_ADDR_BASE 0x3F000000  // IO ペリフェラルの開始アドレス
#define REG_ADDR_GPIO_BASE (REG_ADDR_BASE + 0x00200000) // GPIO の開始アドレス
#define REG_ADDR_GPIO_GPFSEL_0 0x0000  // GPIO Function Select レジスタのアドレス
#define REG_ADDR_GPIO_OUTPUT_SET_0 0x001C  // GPIO Pin Set レジスタのアドレス
#define REG_ADDR_GPIO_OUTPUT_CLR_0 0x0028  // GPIO Pin Clear レジスタのアドレス
#define REG_ADDR_GPIO_LEVEL_0 0x0034  // GPIO Pin Level レジスタのアドレス
```

open 関数を実装する

open 関数では GPIO Function Select レジスタに値をセットして GPIO4 を出力に設定します。`ioremap_nocache(REG_ADDR_GPIO_BASE + REG_ADDR_GPIO_GPFSEL_0, 15);`で物理アドレスを仮想アドレスにマッピングします。カーネルモジュールはカーネル空間で動きますが、カーネルも仮想メモリを使用しているので、物理アドレスにアクセスするときは仮想アドレスにマッピングしてからアクセスします。その後、仮想アドレスの GPIO Function Select レジスタに値をセットします。

▼リスト 1.17　open 関数の中身

```
// GPIO Function Select レジスタを設定
// カーネルも仮想空間上で動くので物理アドレスを仮想アドレスに変換する
int addr = (int)ioremap_nocache(REG_ADDR_GPIO_BASE + REG_ADDR_GPIO_GPFSEL_0, 15);
// GPIO 4 を出力に設定 (12 ビット目)
set_register(addr, 1 << 12);
// 仮想アドレスのマッピングを解除
iounmap((void*)addr);
```

`set_register`関数の中身はリスト 1.18 です。特定のアドレスをポインタとみなして値を代入することでレジスタへの値のセットを実現しています。

このとき、`volatile`キーワードによってコンパイラによる最適化を抑止しています。この関数は、コンパイラからはあるポインタに値がセットされ、そのポインタは以降使われていないように見えます。したがってコンパイラは無意味な処理として最適化処理で消してしまう可能性があります。しかしデバイスドライバー上はこの命令には意味があるので `volatile`で最適化を抑止します。

▼リスト 1.18　set_register 関数

```
static void set_register(unsigned int addr, unsigned int val)
{
    *((volatile unsigned int*)(addr)) = val;
}
```

write 関数を実装する

write 関数の目的はユーザーの入力を受け取って Pin Set レジスタか Pin Clear レジスタに値を出力することです。今回は文字の 1 を受け取ったら GPIO4 を High にし、文字の 0 を受け取ったら GPIO4 を Low にするようにしましょう。

▼リスト 1.19　write 関数の中身

```
// GPIO Pin Set レジスタと GPIO Pin Clear レジスタを使って出力を行う
// ユーザーからの入力を受け取る
char userVal;
get_user(userVal, &buf[0]);
printk("%d", userVal);
int addr = -1;
// ユーザーの入力値によって High にするか Low にするかを分岐
if(userVal == '1') {
    // 1 なら Pin Set レジスタ
    addr = (int)ioremap_nocache(REG_ADDR_GPIO_BASE + REG_ADDR_GPIO_OUTPUT_SET_0, 5);
} else if (userVal == '0') {
    // 0 なら Pin Clear レジスタ
    addr = (int)ioremap_nocache(REG_ADDR_GPIO_BASE + REG_ADDR_GPIO_OUTPUT_CLR_0, 5);
}

// 仮想アドレスを取得できたなら、レジスタに書き込む
if (addr != -1) {
    // GPIO4 なので 4bit 目
    set_register(addr, 1 << 4);
    iounmap((void*)addr);
    print_register(addr);
}
```

read 関数を実装する

read 関数の目的は Pin Level レジスタからデバイスの出力を読み取って、ユーザーに伝えることです。今回は GPIO3 が High の場合は 1 を返し、Low の場合は 0 を返すこととします。

▼リスト 1.20　read 関数の中身

```
// GPIO Pin Level レジスタを参照して GPIO の入力を得る
// 物理 -> 仮想アドレスマッピング
int addr = (int)ioremap_nocache(REG_ADDR_GPIO_BASE + REG_ADDR_GPIO_LEVEL_0, 3);
// Pin Level レジスタの値を得る
int reg_val = get_register(addr);
// 3 ビット目 (GPIO3) の値を取り出し、1 か 0 にする
char val = (reg_val & (1 << 3)) > 0 ? '1' : '0';
// 値をユーザーへ返す
put_user(val + '\0', &buf[0]);
// 仮想アドレスのマップを解除
iounmap((void*)addr);
```

コンパイルと動作チェック

それでは作成した GPIO デバイスドライバーをコンパイルし、動作させてみましょう。コンパイルは何もしないデバイスドライバーと同じように make でできます。

▼リスト 1.21　GPIO ドライバーのコンパイル（Raspberry Pi 上）

```
cd ~/pi-gpio-driver/gpio_driver
make
```

その後、同じようにインストールすることで、デバイスノードができます。

▼リスト 1.22　GPIO ドライバーのインストール（Raspberry Pi 上）

```
sudo insmod mymodule.ko
```

デバイスノードができたら、そのデバイスノードに向かって"1"か"0"を書き込んでみましょう。LED がついたり消えたりするはずです。

▼リスト 1.23　GPIO ドライバーへの出力（Raspberry Pi 上）

```
# LED が点灯する
echo "1" > /dev/hoge0

# LED が消灯する
echo "0" > /dev/hoge0
```

続いて、入力を試しましょう。デバイスノードを cat で read します。すると、1 という文字が大量にでてきます。これは回路がプルアップなのでスイッチから手を離している間は High となるためです。スイッチを押すと押している間だけ 0 が表示されると思います。

▼リスト 1.24　GPIO ドライバーからの入力（Raspberry Pi 上）

```
cat /dev/hoge0
# スイッチから手を話している間は 1 が表示され、スイッチを押すと 0 が表示される
```

最後にデバイスドライバをアンインストールしましょう。

▼リスト 1.25　GPIO ドライバーのアンインストール（Raspberry Pi 上）

```
sudo rmmod mymodule.ko
rm -v /etc/udev/rules.d/mymodule.rules
```

1.5 おわりに

今回、Raspberry Pi を用いて簡単なデバイスドライバーを作ることにより、Linux カーネルへの理解を深めることができました。デバイスドライバーという名前は聞いたことがあるけれど、実際にどのようなものなのかはイマイチ想像がつかないという状態から、本稿を読むことでどういうものなのかの具体的なイメージがついたのではないかなと思います。みなさんもぜひご自宅にある Raspberry Pi で試してみてください！

第 2 章

Raspberry Pi と Google Meet でお手軽ペットカメラ

Daisuke Makiuchi / @makki_d

皆さんは犬派ですか？ 猫派ですか？

今年は COVID-19 の影響でリモートワークを採用する企業が増えました。家にいる時間が長くなったことで、ペットを飼い始めた方も多いのではないのでしょうか。KLab でもほとんどの社員がリモートワークとなったことで[*1]、ちょっとしたペットブームになっているようです。筆者も保護猫兄弟を家族に迎えました。

リモートワークとはいえ、まったく外出しないわけにはいきません。しかし、ペットだけの留守番は何かと不安です。そんなとき、ペットカメラで出先からペットの様子を確認できたら安心ですね。

2.1 市販のペットカメラについて

ペットカメラとここでは呼びますが、要するにネットワークカメラです。ペット用はもちろんのこと介護や防犯目的のものも含め、多種多様な製品が販売されています。これらの製品の多くはペットカメラ自体が動画配信サーバーとなり、出先からはインターネットを介して直接そのサーバーに接続することで映像を見れるようになっています。

インターネットから直接アクセスするためにはグローバル IP アドレスが必要です。しかし筆者の場合、建物全体が LAN になっている集合住宅に住んでいるため、グローバル IP アドレスがありません。そのため、ペットカメラがサーバーとなる製品はそもそも選択肢になりませんでした。

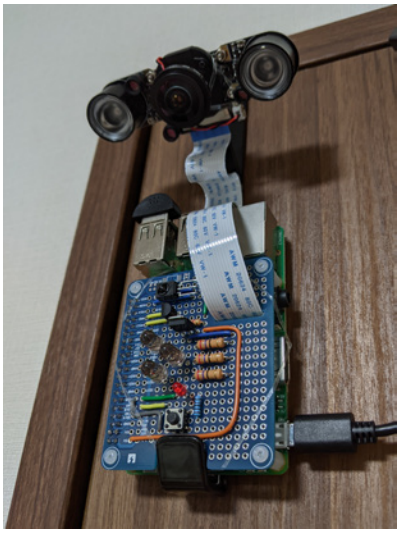
加えて、直接アクセスされる製品の場合、セキュリティに気を配らないとペットカメラの制御が奪われて映像を盗み見られたり、そこを起点に自宅 LAN 内の他の機器へ侵入さ

[*1] 在宅勤務率は約 99 % 新型コロナウイルス感染症対策のこれまでの取り組みについて
https://www.klab.com/jp/press/release/2020/0427/99.html

れたりする可能性もあります。

そんなわけで、筆者は家に転がっていた Raspberry Pi をよりセキュアなペットカメラに仕立て上げることにしました。

2.2 ハードウェア構成



▲図 2.1　全体像

Raspberry Pi

たまたま家で余っていた Raspberry Pi 3B を使いました。3B でもメモリ容量は足りているので、新しく購入するなら 4B の 2GB モデル[*2]が良さそうです。

カメラモジュール

魚眼レンズのものを選ぶと広い範囲を映せて便利です。さらに夜間も映したい場合は、ナイトビジョン対応で赤外線 LED の付く製品[*3]がお勧めです。

そのような機能が不要であれば、Raspberry Pi 専用ではない一般的な USB ウェブカメラでも十分です。

[*2] https://amzn.to/36uK2Ye

[*3] たとえば https://amzn.to/2Kd41m4 と https://amzn.to/3atUx0s

マイク

無くてもよいですが、シンプルな USB マイク[*4]を付けると音声も聞き取れるようになります。

その他

他にも図 2.1 に写っているとおり、スマートリモコンを実装した基板を搭載しています。単純なペットカメラには不要ですが、これについては後述します。

2.3 映像配信方法

ペットカメラ自体を配信サーバーにするのではなく、外部のビデオ通話サービスを利用する形にしました。そうすることで、グローバル IP アドレスが無くても出先から映像を見ることができますし、直接自宅 LAN 内にアクセスさせなくて済むため、セキュリティ上の不安もありません。

ビデオ通話サービスはいくつもありますが、ここでは Google Meet[*5]を利用します。Google Meet は Web ブラウザで完結するので、Puppeteer[*6]や Selenium[*7]による自動化が可能です。また、無料版の Google Meet には通話あたりの時間制限[*8]はあるものの通話数は無制限のため、新しい通話に切り替えていくことで[*9]実質 24 時間 365 日の稼働が可能となります[*10]。さらに専用アカウントで運用すれば、万が一アカウントが奪取されても被害は最小限で済みます。

また、出先からアクセスするのも特別な設定は一切不要で、普通に Google Meet の通話に参加するだけなのでお手軽です。

2.4 Google Meet の自動化

ここでは、ヘッドレス Chrome を Node.js で操作するためのライブラリ Puppeteer を使って、Google Meet を自動化していきます。

[*4] たとえば `https://amzn.to/33AyEbu`
[*5] `https://meet.google.com/`
[*6] `https://pptr.dev/`
[*7] `https://www.selenium.dev/`
[*8] 最長 60 分ですが、2021 年 3 月末までは最長 24 時間となっています。
[*9] 部屋（会議）の URL は使い回せるので、通話終了後リロードするだけで新しい通話になります。
[*10] このような使い方について規約では言及されていませんが、利用したいときだけの通話に留めましょう。

Google へのログイン

Puppeteer で自動操作しているブラウザで Google にログインしようとすると、図 2.2 のようなメッセージが出てログインできない場合があります。

▲図 2.2　ログインできない場合がある

実は、自動化が制限されるのはログインだけなので、ログイン済みセッションの Cookie を Puppeteer にインポートすることで Google の各サービスを利用できるようになります。事前に「Edit This Cookie[*11]」や拙作の「Copy Cookies[*12]」といった Chrome 拡張で Cookie を json 形式でエクスポートし、リスト 2.1 のように Puppeteer にインポートします。

▼リスト 2.1　Puppeteer の起動と Cookie のインポート

```
// puppeteer でブラウザを起動
const browser = await puppeteer.launch({
    args: ['--use-fake-ui-for-media-stream'],    // カメラ起動の許可をスキップ
    executablePath: '/usr/bin/chromium-browser', // Raspberry Pi OS の chromium を使用
});

// 操作対象のページを作成
const page = await browser.newPage();

// json ファイルから Cookie をインポート
const cookies = JSON.parse(fs.readFileSync('~/cookies.json', 'utf-8'));
await page.setCookie(...cookies);
```

頑張れば Cookie をエクスポートする操作も自動化もできますが、詳細は筆者の Web サイト[*13]をご覧ください。

[*11] https://www.editthiscookie.com/
[*12] https://chrome.google.com/webstore/detail/copy-cookies/jcbpglbplpblnagieibnemmkiamekcdg
[*13] http://makiuchi-d.github.io/2020/12/03/autologin-to-google.ja.html

通話の開始

Google Meet では一度作成した部屋（会議）の URL は何度でも使えます。なので、事前に手動で作成した部屋の URL にアクセスして「今すぐ参加」ボタンを押すことで通話を開始できます。

Puppeteer でボタンを押すには、ボタンの DOM 要素を `Page`内から探し `click()`することで簡単にできます。要素の検索にはセレクタや XPath が利用できますが、Google Meet の HTML はリスト 2.2 のようになっています。

▼リスト 2.2　「今すぐ参加」ボタンの HTML

```
<div role="button" class="uArJ5e UQuaGc Y5sE8d uyXBBb xKiqt RDPZE" jscontroller="VXdfxd"
    jsaction="click:cOuCgd; mousedown:UX7yZ; mouseup:lbsD7e; mouseenter:tfO1Yc;
    mouseleave:JywGue;touchstart:p6p2H; touchmove:FwuNnf;
    touchend:yfqBxc(preventMouseEvents=true|preventDefault=true);
    touchcancel:JMtRjd;focus:AHmuwe; blur:O22p3e; contextmenu:mg9Pef;" jsshadow
    jsname="Qx7uuf" aria-disabled="true" tabindex="-1" >
  <div class="Fvio9d MbhUzd" jsname="ksKsZd"></div>
  <div class="e19J0b CeoRYc"></div>
  <span jsslot class="l4V7wb Fxmcue">
    <span class="NPEfkd RveJvd snByac">今すぐ参加</span>
  </span>
</div>
```

このようにクラス名などが明らかに自動生成されたものなので、意味的に要素を特定することが難しく、また頻繁に変更される可能性もあります。セレクタによる要素の検索はやめたほうがよいでしょう。

一方で、ユーザに見せる文字列が変更されることはあまりないと考えられるので[*14]、XPath を使って検索するのがよさそうです。

▼リスト 2.3　通話の開始

```
// 部屋の URL に移動
await page.goto('https://meet.google.com/your-meet-code');

// 「今すぐ参加」ボタンが現れるのを待つ
const button = await page.waitForXPath(
    '//span[text()="今すぐ参加"]', {'visible': true});

// ボタンをクリック（button.click() が動かない場合があるので evaluate する）
await button.evaluate(node => node.click());
```

通話を開始できたら URL を添えて LINE や Slack などに通知しておくと、スマートフォンなどから通話に参加するときに便利です。

[*14] 変更されることもあります。実際「ミーティング」が「通話」に変更されて動かなくなりました。

参加の承諾（拒否）

他のユーザが通話に参加しようとしたとき、部屋のオーナーの画面には参加リクエストダイアログが表示されます。ここで「承諾」ボタンを押してはじめて、そのユーザは通話に参加できるようになり、映像が見れるようになります。

この操作を自動化するには、`waitForXPath()`でダイアログが表示されるのを監視し、表示されたダイアログの「承諾」または「拒否」ボタンを`click()`するという流れになります。具体的には、リスト 2.4 のようなループを非同期に動かしておくことになります。

▼リスト 2.4　自動で参加承諾する

```
const sleep = msec => new Promise(resolve => setTimeout(resolve, msec));

while (true) {
    // 参加リクエストダイアログが現れるのを無限に待つ
    const dialog = await page.waitForXPath(
        '//div[@aria-label="この通話への参加をリクエストしているユーザーがいます"]',
        {'visible': true, 'timeout': 0});

    // ユーザ名とアイコン画像 URL を取得
    const user = await (page.$x('img[@title]'))[0];
    const name = await (await user.getProperty('title')).jsonValue();
    const image = await (await user.getProperty('src')).jsonValue();

    // ここでユーザを確認

    // 許諾ボタンをクリック（"拒否"する場合も同様）
    const button = (await dialog.$x('//span[text()="承諾"]'))[0];
    await button.evaluate(node => node.click();

    // click が処理されるのを待ってからダイアログを消す
    sleep(1000);
    await dialog.evaluate(node => node.parentNode.removeChild(node));
}
```

リスト 2.4 では省略しましたが、知らない人が入ってこないようにユーザを確認する必要があります。ユーザ名だけでは同姓同名の人を識別できないので、アイコン画像の URL も使ってチェックするとよいでしょう。ユニークな画像をアイコンにしていれば一意に識別できるはずです。

あらかじめユーザ名とアイコン画像 URL のホワイトリストを作ってチェックすることで、たとえ部屋の URL が漏れたとしても他人が参加することを防げます。また、ここでも LINE や Slack に承諾・拒否したことを通知しておくとさらに安心です。

2.5　ペットカメラの拡張

ここまでで最低限のペットカメラとして使えるようになりました。せっかくなのでもうちょっと Raspberry Pi らしい拡張をしたいと思います。

スマートリモコンの実装

たとえば外泊するときなど、ペットの部屋の照明やエアコンを外から操作したくなります。市販されているスマートリモコンを使っても解決できますが、どうせならこの Raspberry Pi ペットカメラにスマートリモコン機能を加えてみたいと思います。

スマートリモコンの基板は、Qiita の「格安スマートリモコンの作り方[*15]」を、GPIO13 で制御することと赤外線 LED を 3 並列にすること以外はそのまま実装しました。また、リモコンの信号も学習させてコマンドラインから点灯・消灯できるようにしました。

余談ですが、筆者の環境の Raspberry Pi 3B と Raspberry Pi OS 10 の組み合わせでは、なぜか `irrp.py`の playback が 1/2 倍速になってしまいました。この現象は pigpio の Issue #331[*16]に報告されており、`pigpiod`起動時に`-t0`オプションを加えることで解消できました。

チャットでコマンド実行

Google Meet にはチャットもついており、外からメッセージを投げるにはちょうど良さそうです。チャットメッセージが届いたときに表示されるポップアップの要素を取得できればよいのですが、例によってクラス名などで識別することができません。

そこでリスト 2.5 のように、あらかじめコマンドにしたい文字列を含めた XPath でポップアップを待ち受けることにしました。チャットメッセージからコマンド文字列を取得できたら、定義しておいた外部コマンドを子プロセスとして実行します。

基本的にどんなコマンドでも登録できるので、リモコンだけでなく汎用的に使うことができます。

▼リスト 2.5　チャットでコマンドを実行

```
// 実行したいコマンドを定義しておく
const commands = {
    '点灯': 'python3 irrp.py -p -g13 -f codes light:on',
    '消灯': 'python3 irrp.py -p -g13 -f codes light:off'
};

// コマンド文字列を含む要素を待ち受ける XPath を構築
const cmdxpath = '//div[@role="button"]/span[@jsslot]/span/div/div[{}]'.replace(
    '{}', Object.keys(commands).map(s => `text()="${s}"`).join(" or "));

while (true) {
    // コマンドの書かれたチャットダイアログを待ち受ける
    const msg = await page.waitForXPath(cmdxpath, {'visible': true, 'timeout': 0});

    // コマンド文字列を取り出す
    const cmd = await (await msg.getProperty('innerText')).jsonValue();
```

[*15] https://qiita.com/takjg/items/e6b8af53421be54b62c9

[*16] https://github.com/joan2937/pigpio/issues/331

```
    // 外部コマンドの実行
    child_process.spawnSync('bash', ['-c', commands[cmd]]);

    // 何度も実行されないように要素を削除
    await msg.evaluate(node => node.parentNode.removeChild(node));
}
```

2.6 起動の自動化

最後に、Raspberry Pi 起動時に今回紹介した Puppeteer のスクリプトを自動起動するようにしたいと思います。やることは単純で、自動 GUI ログインを有効にして Autostart にコマンドを登録するだけです。

Raspberry Pi OS 10 の場合、`raspi-config`コマンドで「`1 System Options`」→「`S5 Boot / Auto Login`」と進み、「`B4 Desktop Autologin`」を選択すると、起動時に `pi`ユーザとしてログインした状態で GUI が立ち上がります。

Autostart の登録は`/home/pi/.config/lxsession/LXDE-pi/autostart`ファイルにコマンドを書くだけです。初期状態ではこのファイルが無いので、`/etc/xdg/lxsession/LXDE-pi/autostart`をコピーしてきます。たとえば今回のスクリプトを`/home/pi/automeet.js`とした場合、リスト 2.6 のようにします。

▼リスト 2.6　autostart ファイル

```
@lxpanel --profile LXDE-pi
@pcmanfm --desktop --profile LXDE-pi
@xscreensaver -no-splash
node /home/pi/automeet.js
```

2.7 まとめ

Raspberry Pi を使い、Google Meet を Puppeteer で自動操作することでペットカメラにする方法を紹介しました。

スマートリモコン機能だけでなく、アイディア次第で色々な拡張ができるのも Raspberry Pi のいいところですね。みなさんも余っている Raspberry Pi の活用方法として試してみてはいかがでしょう。

最後に、犬派猫派は排他の関係ではないことを強く主張しておきます。

第 3 章

電子工作で重さを量る

Toshifumi Umezawa

3.1　はじめに

ボタンではなく押圧によって操作をするものが作りたかったため、圧力がかかったことを感知できるセンサーを調べていたところ、関連商品として数十 kg の重さを量ることができるモジュールが目に入りました。

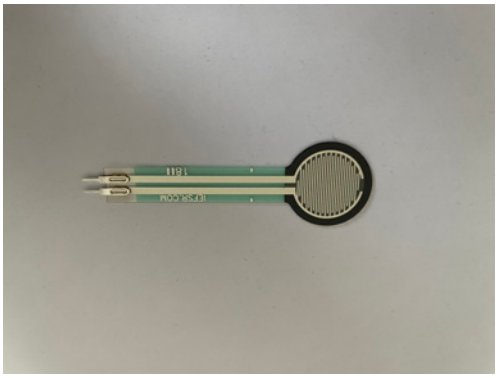
わが部屋では最近、在宅環境の改善のために予想以上に重い机を購入してしまったり、スペースを空けるためにスチールラックに限界まで荷物を詰め込んだりしているので、いつか床が抜けるんじゃないかと心配になっているところでした。重さを量るモジュールをうまく使えば、大きな家具でもそれぞれの脚の下にモジュールを仕込むことで、全体の重さを測定できるかもしれません。

このような目的のために、モジュールの使用方法について調べてみました。

3.2　圧力センサー

FSR402 のような圧力センサーは、数 kg 程度までの小さめの圧力を測定することができます。使い方も簡単で、押し込みの強さによって抵抗値が変わるだけなので、別の抵抗と直列に接続して中間点の電圧を読み取ることで圧力を逆算できます。

今回は大人 1 人分くらいの重さを量りたいので、このタイプのモジュールは不向きです。



▲図 3.1　FSR402

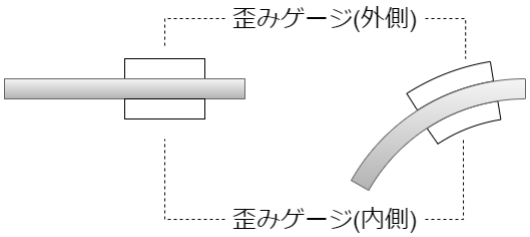
長時間使用で出る特性

圧力センサーのようなモジュールは、長い間圧力をかけ続けると抵抗値が変化してしまう特性があるので[*1][*2]、用途によっては使えないこともあるようです。

3.3　ロードセル

ロードセルは、歪みゲージを使って重さを測るモジュールです[*3]。歪みゲージとは伸び縮みで抵抗値が変化するセンサーで、たとえば図 3.2 のように、しなるように変形する板の両面にゲージを貼り付けた場合、曲げた際に内側のものは縮み、外側のものは伸びることになり、抵抗値に差が出ます。

[*1] https://qiita.com/6LxAi9GCOmRigUI/items/ed290a5fbebc56b7bd0e
[*2] https://shirouzu.jp/tech/coffeepot-iot
[*3] 参考 URL: https://www.unipulse.tokyo/techinfo/loadcetthowto/



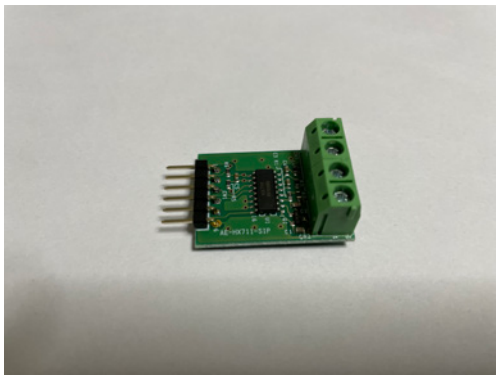
▲図 3.2　歪みゲージの伸縮

板に重さを加えることでゲージが歪み、それによる抵抗値の変化を電圧の変化という形で取り出します。荷重と出力電圧は比例関係として扱うことができ[*4]、出力電圧から荷重を計算することができます。このときの電圧の変化量は微小なものなので、オペアンプで増幅した値をマイコン側で測定します。

3.4　ドライバ

HX711 はロードセルの値を読み取れる AD コンバータです。ロードセルによる微小な電圧変化を増幅する機能はもちろんのこと、供給電圧によらず一定の電圧を作る機能[*5]などいくつかの機能も備えています。

今回は秋月電子通商にて販売されている「ＨＸ７１１使用　ロードセル用ＡＤコンバータ　モジュール基板[*6]」を使用します。



▲図 3.3　HX711 使用基板

[*4] 実際には比例ではないことによる誤差は非直線性と呼ばれます。秋月のロードセルの商品ページだとリニアリティ誤差と呼んでいるようです。

[*5] Band Gap Reference と呼ばれ、大抵の場合 1.25V の電圧になっています。

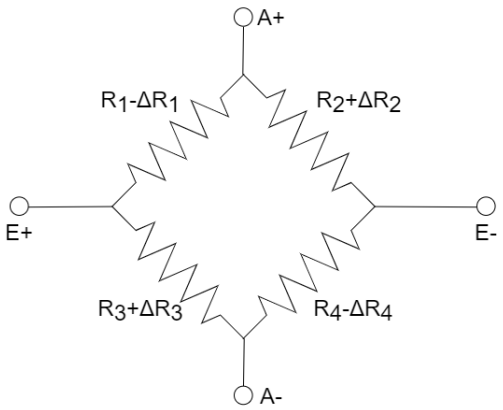
[*6] https://akizukidenshi.com/catalog/g/gK-12370/

HX711 の出力間隔

本来 HX711 のデジタル出力の間隔は 10Hz/80Hz のどちらかを選択できますが、ブレイクアウトボードによっては 10Hz 側に固定になっていることも多いようです。今回使用する秋月の基盤も 10Hz 動作に固定されています。

3.5 フルブリッジ型ロードセル

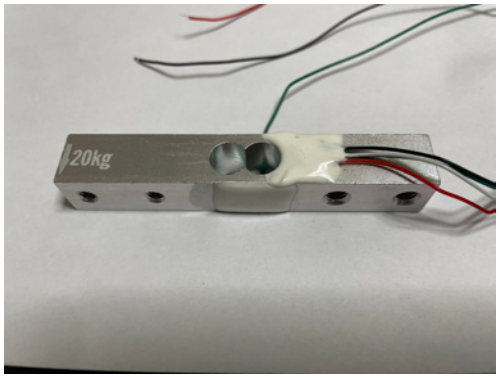
SC133[*7]のようなフルブリッジ型のロードセルは、ホイートストンブリッジの形で歪みゲージが配置されています。ホイートストンブリッジ（図 3.4）では、電圧の高い方から低い方（E+ → E-）に到達する 2 つのルートがあります。ロードセルの歪みによって抵抗値が微小に変化することで、それぞれのルートの中間点どうし（A+ と A-）に電圧差が生まれます。これによって歪みの測定が可能になります。



▲図 3.4　ロードセルのホイートストンブリッジ

SC133 からは赤/白/緑/黒の 4 本の線が出ており、それぞれ [E+,A+,A-,E-] となっているので、すべて HX711 の所定の場所に繋ぐだけで測定できるようになります。

[*7] https://akizukidenshi.com/catalog/g/gP-12034/



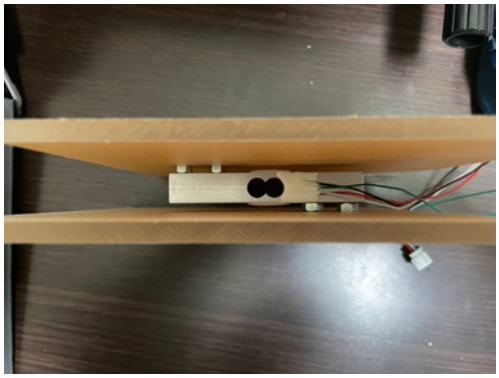
▲図 3.5　ロードセル SC133

実験

まず、SC133 をうまく動作させるためにはネジで固定する必要があります[*8]。M5 ネジ 2 つを指定の間隔で固定できる板と、同様に M4 ネジ 2 つを固定できる板が必要なので、今回は東急ハンズの工房で 150mm*300mm のアクリル板を半分に切って 4 つ穴を開けてもらうことで準備しました。何も考えず店頭で思いついた形で作ってもらったため変なずれ方をしたものが組み上がりましたが、これでも測定できるので問題なかったことにします。

▲図 3.6　SC133 を使う装置

[*8] ちょうどいい部品を用意するのが面倒だったので、ロードセル用の板やネジのセットが売られているといいなと思いました。



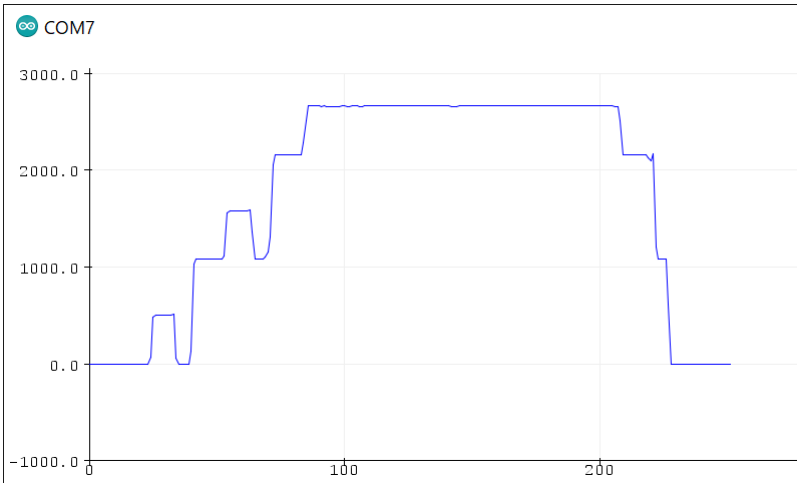
▲図 3.7　SC133 を使う装置を横から見た図

▲表 3.1　SC133 を固定する板の材料と加工費

内容	値段（税込）
アクリル板150*300*5	966 円
直線カット	55 円
円切り抜き*4	220 円
合計	1,241 円

HX711 ドライバの商品ページから DL できるサンプルコードを使い、Arduino と HX711 の基盤と SC133 を配線し、シリアルで出力される情報をグラムの数字だけにしてシリアルプロッタにてグラフを出力しました。

梅干しを漬けるのに使った 1kg/2.5kg/2.5kg のつけもの石を使って、1kg/2.5kg/3.5kg/5kg/6kg の荷重になるように乗せかえたときのグラフは図 3.8 のようになりました。



▲図 3.8　フルブリッジ型の測定

▲図 3.9　6kg の荷重を加えている様子

グラフを見ると、1kg のつけもの石を乗せたときに 500g として表示されているように見受けられますが、これはプログラムとモジュールの情報の不一致によるもののようで

す[*9]。SC133 の定格出力は 1.0 ± 0.1 mV/V なのですが、サンプルプログラムでは 2.0 mV/V として定義されているので、この値を修正することで本来の重さが表示されるようになります。コードを見た限り、SC601 の定格出力も間違っていたので、おそらく 2 つのモジュールで定数が入れ替わってしまっているようです。

▼リスト 3.1　定数の修正

```
@@ -21,13 +21,13 @@
 //-----------------------------------------------------//
 // ロードセル  シングルポイント（ ビーム型）  ＳＣ６０１  １２０ｋＧ [P-12035]
 //-----------------------------------------------------//
-//#define OUT_VOL   0.001f      //定格出力 [V]
+//#define OUT_VOL   0.002f      //定格出力 [V]
 //#define LOAD      120000.0f   //定格容量 [g]

 //-----------------------------------------------------//
 // ロードセル  シングルポイント（ ビーム型）  ＳＣ１３３  ２０ｋＧ [P-12034]
 //-----------------------------------------------------//
-#define OUT_VOL   0.002f      //定格出力 [V]
+#define OUT_VOL   0.001f      //定格出力 [V]
 #define LOAD      20000.0f    //定格容量 [g]

 //-----------------------------------------------------//
```

図 3.8 では、6kg 分すべての重しを乗せた状態で 2.7kg 程度の表示になりました。定数の修正前なので実際には 5.4kg ですが、それでも明らかに重さが足りていません。この点はあとで掘り下げることにします。

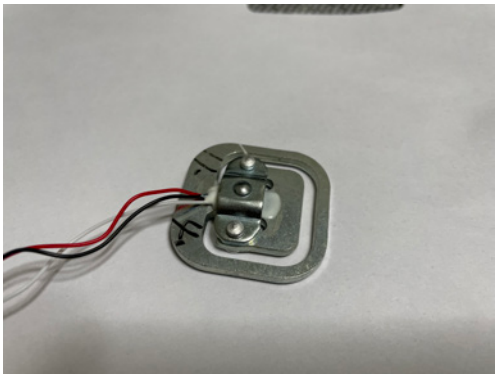
3.6　ハーフブリッジ型ロードセル

次に使用するのは SC902 というハーフブリッジ型のロードセルです。これは、ホイートストンブリッジの片側のルートだけ実装したような構造になっています。

このロードセルからは赤/白/黒の 3 本の線しか出ていないため、そのまま HX711 に繋ごうとすると一本足りません。

一見不親切に思えるかもしれませんが、実は柔軟にいろいろな実装ができる気の利いたモジュールなのです。

[*9] 問い合わせを送ったので今後修正される可能性があります。



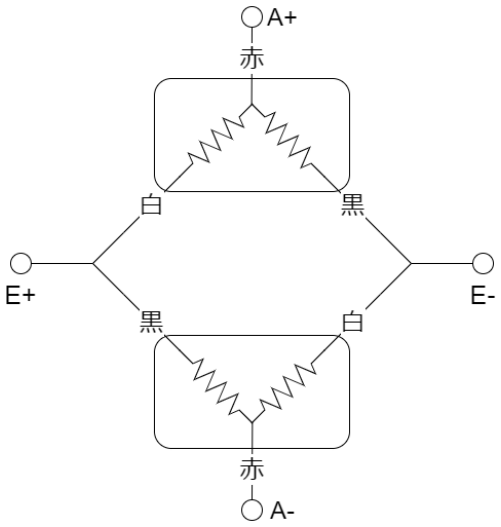
▲図 3.10　ロードセル SC902

2 つ組で使う

「ハーフ」ブリッジなので、2 つ使うことでフルブリッジ同等の配線をすることができます。おそらく一番思いつきやすい方法ではないでしょうか。

図 3.11 のように配線することで、2 つの赤い線をそれぞれ電圧の微増/微減の出力として扱うことができます。

注意点として、モジュールから出ている白/黒の線について、2 つのモジュールそれぞれで逆の色どうしを繋ぐようにします。同じ色どうしを繋いでしまうと、2 つのモジュールに均等に荷重がかかった際の電圧の変化が同じになってしまい、電圧の差を測定できません。違う色どうしを繋ぐことで電圧の変化の方向が逆になり、正しく電圧の差を測定できるようになります。

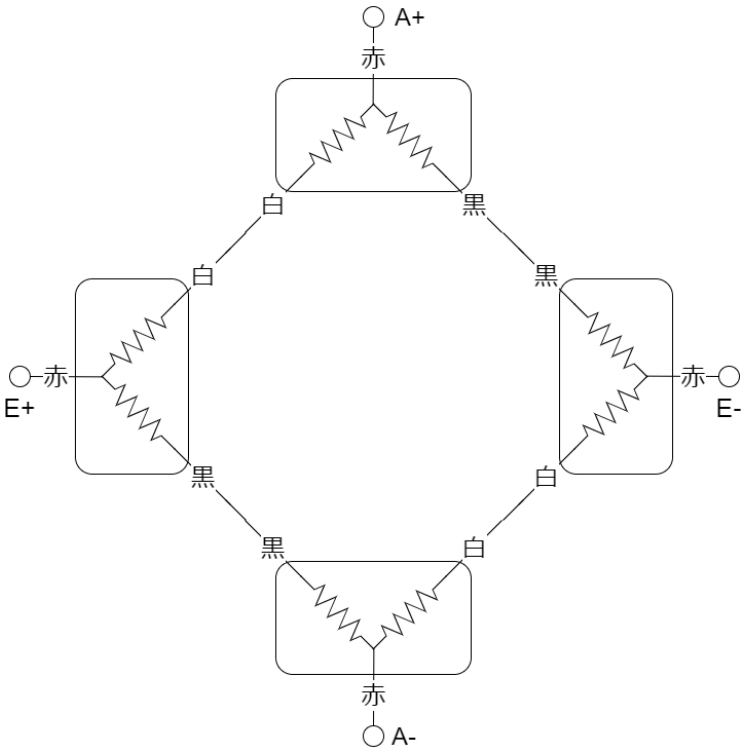


▲図 3.11　2 つ組で使うときの繋ぎ方

4 つ組で使う

この方法はデータシートにも画像と配線が書かれています。物理的なバランス的にも 4 箇所で支える方が安定しますし、おそらく一番よく使われている方法ではないでしょうか。

図 3.12 のように配線することで、4 つの赤い線を電圧の入力と出力の端子として使えるようになります。



▲図 3.12　4 つ組で使うときの繋ぎ方

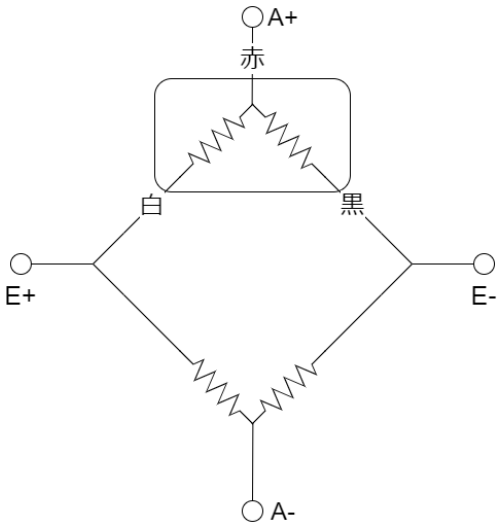
注意点としては、2 つ組で使うときとは逆で、同じ色どうしを繋ぐようにします。理由は 2 つ組の場合と同様で、異なる色どうしを繋いでしまうと均等に荷重がかかった際に電圧差がなくなってしまうからです。

2 つ組と 4 つ組で使う場合はどちらも、複数のモジュールに分散してかかった荷重を HX711 ひとつだけで量れる点はメリットとして大きいと思います。

1 つだけで使う

ホイートストンブリッジとして欠けてるもう片方のルートを、図 3.13 のように同じ抵抗 2 つを使ったルートとして実装することで、モジュール 1 つだけでも測定することが可能なようです。

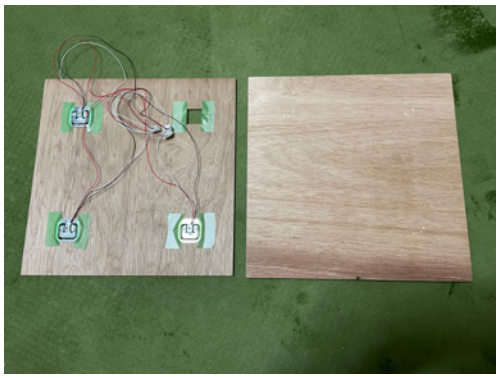
注意点として、2 つ組の場合と比較して出力される差分電圧が半分になる点と、無負荷状態での電圧差を可能な限り 0 にしておく必要があります。この実装は少し難しく、適当に地面に転がっている 1k Ω抵抗 2 つを繋いでみたところ、めちゃくちゃな重さが出力されました。抵抗ひとつひとつにも多少の個体差があり、抵抗値が近いものを使わないと大きすぎる電圧差が出力されてしまうようです。



▲図 3.13　1 つだけで使うときの繋ぎ方

実装

図 3.14 のように、板（5.5mm）に SC902 をはめ込んで歪ませられる穴を開け、乗せたあと養生テープで固定し、その上に人が乗っても大丈夫そうな厚み（12mm）の板を載せるようにします。この穴あけ加工も東急ハンズの工房に依頼しました。



▲図 3.14　SC902 を使う装置

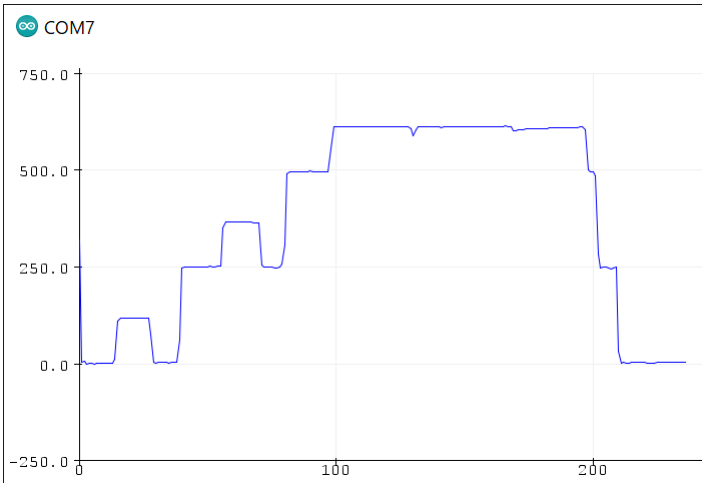
あとはフルブリッジ型の実装と同様にサンプルプログラムをそのまま動かし、手持ちのつけもの石で荷重をかけます。

結果は図 3.15 に示すとおりです。フルブリッジ型と同様の形になっているのがわかり

▲表 3.2　SC902 を固定する板の材料と加工費

内容	値段（税込）
ラワン合板300*300*12	429 円
ラワン合板300*300*5.5	209 円
四角切り抜き*4	1,320 円
合計	1,958 円

ます。



▲図 3.15　4 つ組での測定

4 つ組のパターンでは 1kg の重しが 125g と表示されています。定数修正前の SC133 用のプログラムをそのまま流用したため、定格出力の違いで 1/2、さらにこのモジュールの定格容量が 50kg なので 2/5 の値、すなわち 1kg の 1/5 の 200g と表示されるはずです。

ずれの原因は不明ですが、1kg と 2.5kg の重しで値が変化する比率は問題なさそうなので、係数さえうまく合わせれば正しい重さを表示させることはできそうです。

3.7　重さの計測

ここまでに作成したフルブリッジ型、ハーフブリッジ型 4 つ組の装置では、どちらもすべての重しを乗せて 6kg の荷重をかけたときの数値が明らかに足りていないように見受けられました。しかし、自宅には 2kg までしかはかれない料理用の電子秤しかないため、真偽を確かめることができません。

悩んだ末、2 つのことを実行してみました。

1. 既製品の料理用電子秤を正として、2kg 以下のものの測定結果を使って表示値と重さの対応を作り、それを元に 2kg 以上のものが電子秤基準で何 g と表示されるはずかを求める。
2. 体重計を買ってきて、上で求めた 2kg 以上のものの重さの整合性を確認する。

1. については、モジュールにかかる荷重の大きさと出力される値には比例関係があるはずなので、比較的軽い重しなどを使って電子秤と自作装置の値との比例関係の係数を求め、それを使うことで電子秤で本来計測できない重さも自作装置の値から推定することができるはずです。2. については、1 の方法で求めた重さが本当に正しいのかの検証のために行います。

既製品の料理用電子秤との対応

自作装置に載せているときの出力されている値から連続した 5 つをとってきて、その平均をとりました。電子秤の方はかなり優秀で何度載せ直しても同じ値になってくれたので、1 回載せて表示された値を使用しました。

フルブリッジ型の装置での実験

まず、2kg 以下でぱっと目に入ったものを載せた結果は表 3.3 のようになりました。電子秤は表示値そのまま、自作装置は精度 10g ですがサンプルプログラムで出力していた小数点以下 4 桁の表記をそのまま記載しています。[*10]

▲表 3.3　フルブリッジ型での 2kg 以下のものの重さの対応

載せたもの	自作装置（フルブリッジ）	電子秤（g）
1kg重り	502.4382	1066
iphone6	72.9740	156
pixel3	68.8726	148
iphone12	100.8444	216
LGリモコン	36.6750	83
ドライバーセット	162.3284	351
1kg重り＋ドライバーセット	664.1664	1418

電子秤の表示値 $=$ 自作装置の表示値 $\times\, a + b$ としたとき、線形回帰により $a = 2.1261, b = 2.6547$ となりました。これをもとに電子秤では計測できない重さを推定した

[*10] SC133 の精度はフルスケール 20kg の 0.05% のため 10g です。

ところ表 3.4 のようになりました。これをもとに、自作装置で 2kg 以上の重しを量った数値から実際の重さを推定すると表 3.4 のようになりました。

▲表 3.4　フルブリッジ型での 2kg 以上のものの測定値と重さの推定値

載せたもの	自作装置（フルブリッジ）	推定値（g）
2.5kg	1073.5868	2285.2162
3.5kg	1575.6322	3352.6189
5.0kg	2148.5556	4570.7160
6.0kg	2648.8282	5634.3496

4 つ組の装置での実験

ハーフブリッジ 4 つ組の装置でも同様に実験をしました。

まず、2kg 以下の物を載せた結果は表 3.5 のようになりました。自作装置は精度 200g ですが同様に小数点以下 4 桁の表記を記載しています。[*11] なお、何も載せていない状態の表示値のブレが多少あったので、物を降ろしたあとの何も載せていない状態の表示値を 5 つ平均したものを風袋として引いています。

▲表 3.5　4 つ組での 2kg 以下のものの重さの対応

載せたもの	自作装置（4つ組）	電子秤（g）
1kg重り	114.2532	1066
iphone6	15.7911	156
pixel3	14.5644	148
iphone12	22.5416	216
LGリモコン	8.7390	83
ドライバーセット	37.5682	351
1kg重り＋ドライバーセット	152.3486	1418

ここでも線形回帰により、 $a = 9.2610, b = 7.1824$ となりました。これを元に 2kg 以上の重しを測定し、実際の重さを推定すると表 3.4 のようになりました。4 つ組の装置では 50kg まで計測可能なモジュール 4 つに負荷分散しているため、200kg まで量れるはずです。なので、容赦なく自分も乗ってみます。体重は 50kg ± 1kg くらいのはずです。

[*11] SC902 の精度はフルスケール 50kg の 0.2% のため 100g ですが、4 つ組で使用した際の精度は $\frac{100 \times 4}{\sqrt{4}} = 200$ g となります。参考: http://www.minebea-mcd.com/product/system_a/total_accuracy.html

▲表 3.6　4 つ組での 2kg 以上のものの測定値と重さの推定値

載せたもの	自作装置（4つ組）	推定値（g）
2.5kg	246.0148	2285.5205
3.5kg	362.1154	3360.7257
5.0kg	494.3574	4585.4162
6.0kg	609.4216	5651.0234
自分	5506.7812	51005.3708
1.0kg+自分	5623.9336	52090.3168
2.5kg+自分	5755.7672	53311.2251
3.5kg+自分	5877.1934	54435.7507
5.0kg+自分	6001.2836	55584.9475
6.0kg+自分	6125.6812	56736.9911

2 種類の装置での結果考察

電子秤を基準に補正することで、2.5kg〜6.0kg の重さについて両方の装置で同じような値が出せていることがわかります。

1kg の重しについては本来あるべき重さよりちょっと重そうです。問題は 2.5kg の重しがやはり両方とも 2.285kg 程度しかなさそうなことです。通販で買っていたので改めて商品ページを見に行きましたが、間違いなく「2.5kg」と書かれていました。

自分の体重についてはほぼほぼ想定どおりの値が出ていそうです。ちょっと太った？

体重計との比較

思い立ったその日に体重計を買ってきました。残念ながらある程度の重さがないと計測開始してくれないようだったので、2.5kg の重しについては自分が抱きかかえたまま乗ることで測定しました。

体重を量ってみると 50.2kg、2.5kg のはずの重しを 2 つ抱えて乗ると 54.9kg と表示されました。もう一度やってみるとそれぞれ 50.5kg, 55.0kg となりました。つまり、2.5kg の重しは実際には 1 個あたり約 2.3kg だったことになります。

小さめの体重計を買ってしまったので、どうしても乗り方によって表示がずれてしまうのですが、どちらにせよ 2.5kg のはずだった重しは表記より軽かったらしいことがわかりました。

また、自分の体重については自作装置での測定値は 51kg でしたが、買ってきた体重計だと 50.2kg, 50.5kg でした。差は 1kg 以内とまずまずだったので、厳密な値を必要としないのであればこの自作装置も十分使えるのではないでしょうか。

今回の実験では、本来なら加えて電子秤と体重計の重さ表示に差分がないか調べるべきですが、体重計の方がそもそも 0.1kg 単位でしか表示されないものだったため、厳密にどちらが正しいか判定するのは難しそうです。

3.8 体重計について考える

体重計による表示値のブレについて検索してみると、「体重計によって表示される値が変わる」といった声が見受けられます。

今回実際に計測してみて分かるとおり、どうしても使うモジュールやそのまわりの環境によって表示値に差が出てしまいます。20kg まで計測できる SC133 を使った装置の実験の際には、電子秤と装置の表示値の間の比例係数に小数点以下のずれがありました。実際に重さを量る装置を作ってみてわかったのは、微小な電位差を扱うため抵抗の実際の抵抗値に気をつけたり、何か別の重さ計測機器の表示値に合わせるように微調整してあげる必要があることです。

また、もし自宅の体重計の外箱や取扱説明書を保存していたら仕様を確認していただきたいのですが、家庭用の体重計の精度はそこまで高くないようです。自分の買ったものだと、50kg での仕様では「目盛りは 100g 単位、精度は ± 200g」とあり、重くなるほど目盛りも精度も荒くなるようでした。

高機能な体重計だと地域の設定ができるものもあります。これは地球上のどこにいるかによって若干変わる重力加速度を考慮するためのもののようです。

3.9 まとめ

重さをはかれるモジュールを使ってみることで、歪みと比例した電圧変化によって重さの計測が可能なことがわかりました。そして、重さを量るには何か別のものを基準として補正してあげる必要があることもわかりました。

体重計によって重さの表示が違うことがあるようですが、これは正しく調整していないとは限らず、どうしても微小値を扱うため環境や経年劣化などでずれてしまうものだと思います。正確に正しい重さを表示するようにするためには、それ相応に誤差の出にくい信頼性のあるパーツを使ったり、同じ重さを共有しているはかりを使って値を調整する必要があると思われます。

3.10 最後に

「家具の脚にピンポイントでセンサーを配置することで重さを計測できるかも」という動機から調べに入ったわけなのですが、重量物を持ち上げて自作装置を仕込んでバランスを取るという操作は 1 人でやるには非常に危険なため実現していません。

第 4 章

Unity エディタ拡張のプログラム設計 〜業務で使えるツールの作り方〜

Yuuki Hirai / @96yuuki331

4.1 はじめに

初めまして、KLab で Unity エンジニア（プログラマ）をしております Hirai です。

スマートフォンゲームの開発で Unity を採用している事例は多々あり、KLab でも多くのタイトルで Unity を利用しています。Unity エディタはプログラムを書かずとも GUI だけで多くのことができるため、エンジニアだけでなくプランナーやデザイナーも Unity エディタを使用することがあるかと思います。Unity にはエディタ自体をカスタマイズし独自の機能を追加することができる、Unity エディタ拡張という仕組みが用意されています。以下は公式サイトの引用です。[*1]

> Unity では独自のインスペクターと エディターウィンドウ、さらにインスペクターにプロパティーを表示する独自の Property Drawer を作成することができます。

社内にも、一部プランナーやデザイナーが自分たちの作業効率化に合うように Unity エディタをカスタマイズしている事例があります。しかし、自作のエディタ拡張の不具合対応や機能追加など、いわゆる保守や拡張の面で悪戦苦闘している場合もありました。

Unity エディタ拡張の不具合対応や機能追加を簡単にするためには、一般的なプログラム開発同様、設計をしっかり行うことが大切です。プログラミング経験があまりなく、Unity エディタ拡張を自身できちんと作れるようになりたいプランナーやデザイナーの方

[*1] Unity エディタ拡張とは `https://docs.unity3d.com/ja/2018.4/Manual/ExtendingTheEditor.html`

に向けてプログラマから提供できる情報があると筆者は思います。

そこで、デザイナー向けのエディタ拡張を開発し、実際に業務で使って頂いた私の経験を踏まえて、プログラマが考える **Unity エディタ拡張の設計**について、実際に簡単なツールの作成を体験してもらいながら解説したいと思います。

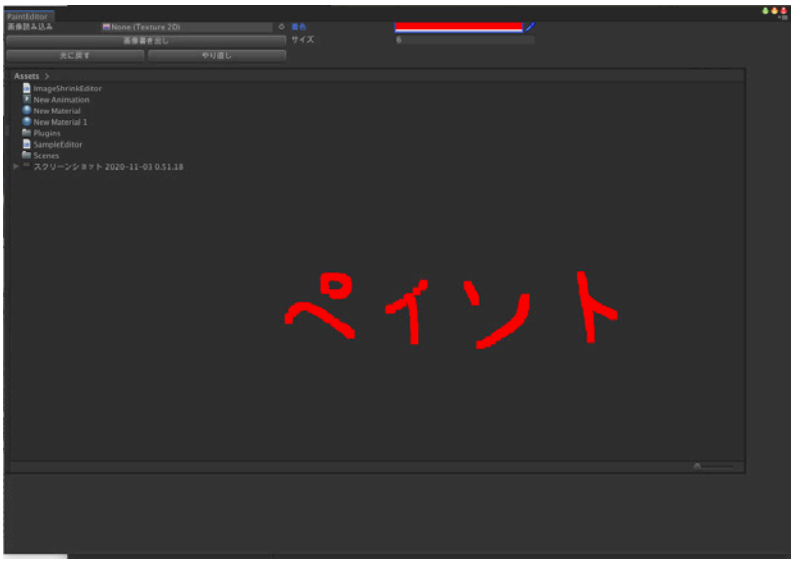
ある程度の Unity エディタ拡張開発の経験がある方向けの凝った内容も記載していますが、よろしくお願いいたします。

なお、本章は Unity 2017 以降を対象としています。

4.2 作成するツール

今回、例題として作成するツールは **画像ペイントツール** です（図 4.1）。このツールは、PNG 画像を読み込んでエディタ上に表示し、そこにマウスでペイントできるものです。そして、ペイントした内容は PNG 画像として保存できます。

また、他のツールを作成するときにも流用できるファイルの自動保存、アンドゥ・リドゥといった機能の実装も紹介するので、ぜひ参考にしてみてください。



▲図 4.1　画像ペイントツール

4.3 実践

画像ペイントツールを作成していく中で次の順番で紹介していきます。

1. 基本処理の構成

2. UI の作成
3. 画像の読み込み
4. マウスでペイント
5. 編集した画像を書き出し
6. ScriptableObject で自動保存
7. 保存データをもとにアンドゥ・リドゥ実装

基本処理の構成

Unity エディタ拡張では `void OnGUI()` という関数の中に UI の表示や処理を実装していきます。このとき、`OnGUI()`にいろいろな処理を書いてしまうと見通しがわるくなり、具体的に何をやっているのかわかりにくくなります。そして、あとからコードを修正したり追記するときに、変更すべき箇所を特定することが大変になります。そこで次の 3 つの処理フローに分けて実装していきます。

1. 初期化関数（`Initialize()`）
2. 更新関数（`DataUpdate()`）
3. 描画関数（`Rendering()`）

それぞれの関数にどのような処理を記述していくかを次にまとめます。

初期化関数

`Initialize()` には一番最初にしないといけない処理を記述していきます。たとえば、データの読み込みといった一番最初に処理して後続の処理に関わる処理を記述します。

更新関数

`DataUpdate()`では変数やクラスの更新、つまり内部状態の変更処理を行います。この関数の中も見通しが悪くならないように、処理のまとまりごとに関数を分けて記述しましょう。

描画関数

`Rendering()`では、UI を表示する処理を主に記述します。GUILayout.~や GUI.~といった関数の呼び出しがメインになります。Unity では、GUI 周りの処理で変数の更新処理を記述することになります。最小限に止めておくと見通しが良くなります。今回作成するツールでは、UI を表示することに加え、ボタンなどの入力に伴う処理もここに実装していきます。

最初のソースコードはリスト 4.1 のようになります。`OnGUI()`にて先ほどの 3 つの関数を順に呼び出しています。これをベースに拡張していきましょう。

▼リスト 4.1　最初のソースコード

```
using UnityEditor;
/// <summary>
/// ペイントエディタエディタ
/// </summary>
public class PaintEditor : EditorWindow
{

    [MenuItem("Tools/PaintEditor")]
    static void Create()
    {
        GetWindow<PaintEditor>("PaintEditor");
    }

    void OnGUI()
    {
        // 初期化
        Initialize();
        // 更新
        DataUpdate();
        // 描画
        Rendering();
    }

    /// <summary>
    /// 初期化
    /// 注意：毎フレーム最初に更新される初期化処理
    /// </summary>
    void Initialize()
    {

    }

    /// <summary>
    /// 更新
    /// クラスや変数を更新する
    /// </summary>
    void DataUpdate()
    {

    }

    /// <summary>
    /// 描画
    /// 画像やボタン等の UI を描画する
    /// </summary>
    void Rendering()
    {

    }
}
```

UI の作成

UI を作成していきましょう。
先に UI の見た目（ボタンや文字や数値を入力するためのフォーム）を作成しておき、そこに UI を操作した時の処理（ボタンを押した時やフォームに入力した時の処理）を追加していけば実装確認がしやすいです。

Unity エディタ拡張ではさまざまな UI が存在します。その中でも今回使用する各 UI を解説します。

▼リスト 4.2　ボタン描画

```
bool isClick = GUILayout.Button("ボタン内のテキスト");
if (isClick)
{
    // ボタンが押された時の処理を記述
}
```

GUILayout.Buttonでボタンを描画します。isClickはボタンが押されるとtrueになります。そのため、if文と組み合わせてボタンが押された時の処理を記述します。

▲図 4.2　Unity エディタ拡張　Button 紹介

▼リスト 4.3　ラベル描画

```
GUILayout.Label("ラベルのテキスト");
```

GUILayout.Labelでラベル（テキスト）を描画します。今回は文字や数字などをマウスやキーボードなどで入力できるEditorGUILayout.ObjectFieldと組み合わせて使います。

▲図 4.3　Unity エディタ拡張　Label 紹介

▼リスト 4.4　オブジェクトフィールド

```
Object obj = null;
obj = EditorGUILayout.ObjectField(obj, typeof(Object), false);
```

GUILayout.ObjectFieldは、Unity エディタでよく見かける Asset フォルダ内のアセットを入れる UI を描画するコードです。objに入力したアセットが入ります。

▼リスト 4.5　int 型フィールド

```
int a;
a = EditorGUILayout.IntField("テキスト", a);
```

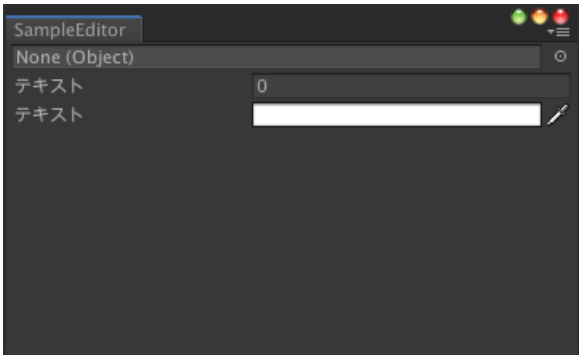
▼リスト 4.6　Color 型フィールド

```
Color color;
color = EditorGUILayout.ColorField("テキスト", color);
```

オブジェクトフィールドの派生として、次のものが存在します。

- 整数値だけ入力できる IntField（リスト 4.5）
- カラー情報だけ入力できる ColorField（リスト 4.6）

他にも存在しますが、今回使用するのはこの 2 点だけです。



▲図 4.4　Unity エディタ拡張　~Field 紹介

▼リスト 4.7　横並び

```
using (new GUILayout.HorizontalScope())
{
    ～
}
```

▼リスト 4.8　縦並び

```
using (new GUILayout.VerticalScope())
{
    ～
}
```

GUILayout.HorizontalScopeと GUILayout.VerticalScopeは、括弧で括られて記述されている UI をグループとして整列させることができる処理です。紹介した UI 等をきれいに整列させる際に使用します。

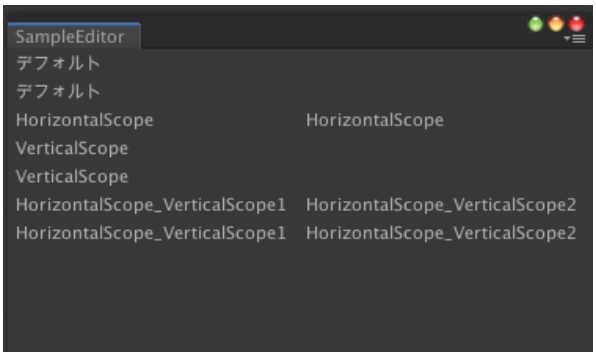
▼リスト 4.9　～Scope のサンプル

```
GUILayout.Label("デフォルト");
GUILayout.Label("デフォルト");

using (new GUILayout.HorizontalScope())
{
    GUILayout.Label("HorizontalScope");
    GUILayout.Label("HorizontalScope");
}

using (new GUILayout.VerticalScope())
{
    GUILayout.Label("VerticalScope");
    GUILayout.Label("VerticalScope");
}
```

```
using (new GUILayout.HorizontalScope())
{
    using (new GUILayout.VerticalScope())
    {
        GUILayout.Label("HorizontalScope_VerticalScope1");
        GUILayout.Label("HorizontalScope_VerticalScope1");
    }
    using (new GUILayout.VerticalScope())
    {
        GUILayout.Label("HorizontalScope_VerticalScope2");
        GUILayout.Label("HorizontalScope_VerticalScope2");
    }
}
```



▲図 4.5　Unity エディタ拡張　~Scope 紹介

void Rendering()に UI を記述していきましょう。
今回、作成する画像ペイントツールでは「画像読み込み」「画像書き出し」「アンドゥ（元に戻す)」「リドゥ（やり直し)」「ペイントする色」「ペイントするサイズ」を実装していくので、それらのインターフェイス（入力処理）を作成します（リスト 4.10)。

▼リスト 4.10　UI 描画

```
public class PaintEditor : EditorWindow
{
    // 定数リスト
    const int GuiSpace = 2;             // UI のスペース値
    const int MenuSpaceWidth = 400;     // 左上のメニュー UI 横幅
    const int BrushSpaceWidth = 350;    // 右上のペイント編集 UI 横幅
    const int TextureHeightPos = 65;    // 着色するテクスチャの高さ位置
    const int RedoUndoStockCount = 20;  // リドゥ・アンドゥで使用する最大ストック数

    // 着色する色
    Color paintColor = Color.black;

    // 着色するブラシサイズ
    int paintSize = 1;

    [MenuItem("Tools/PaintEditor")]
    static void Create()
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/// <summary>
/// 描画
/// 画像やボタン等の UI を描画する
/// </summary>
void Rendering()
{
    using (new GUILayout.HorizontalScope())
    {
        using (new GUILayout.VerticalScope
          (GUILayout.Width(MenuSpaceWidth + GuiSpace)))
        {
            using (new GUILayout.HorizontalScope())
            {
                GUILayout.Label("画像読み込み");
                Texture2D tex = null;
                tex = (Texture2D) EditorGUILayout.ObjectField(tex,
                   typeof(Texture2D), false);
            }
            using (new GUILayout.HorizontalScope())
            {
                if (GUILayout.Button("画像書き出し"))
                {

                }
            }

            using (new GUILayout.HorizontalScope())
            {
                if (GUILayout.Button("元に戻す"))
                {

                }

                if (GUILayout.Button("やり直し"))
                {

                }
            }
        }

        using (new GUILayout.VerticalScope
          (GUILayout.Width(BrushSpaceWidth + GuiSpace)))
        {
            paintColor = EditorGUILayout.ColorField("着色", paintColor);
            paintSize = EditorGUILayout.IntField("サイズ", paintSize);
        }
    }
}
```

～Unity エディタ拡張で記述する数字は定数化しよう～

マジックナンバーという用語を聞いたことはあるでしょうか。プログラマの中ではよくやりとりされるキーワードでプログラム処理に直接埋め込まれている数字です。
Unity エディタ拡張に限らずプログラムにはソースに埋め込む数字が発生します。それらは定数化して管理しなければ調整が大変なことになります。
たとえば、Unity エディタ拡張で複数のボタンが上下に並んでおり、それらを 20px ほど下にズラしたい時を考えてみます。ボタンの最大数や位置を定数として一箇所で定義して管理する形を取っていれば、ボタンの位置を変更する際にもその定数を変更するだけで済みます。しかしながら、定数化して管理するような形にしていなければ、ボタンひとつひとつを 20px 下にズラす用に書き換えないといけません。
これは非常に無駄なことで、位置を調整し忘れた際には不具合の原因にもなり得ます。筆者としてはこういったことがないよう定数化する癖をつけることを強くお勧めします。

画像の読み込みと表示

次に画像読み込み機能を実装します。

`void Rendering()`にて、`Texture2D`のオブジェクトフィールドにテクスチャが読み込まれた時に `paintTexture`へコピーします（リスト 4.11）。C#で用意されている `FileStream`を用いて画像ファイルを開き、`BinaryReader`でファイルの中身を読み取りそれを画像データとして `Texture2D`の生成を行なっています。

処理負荷が高いのであまり連続して使うことはできないですが、画像読み込み処理を連続することは滅多にないと思うので大丈夫かと思います。

▼リスト 4.11　paintTexture にコピー

```
GUILayout.Label("画像読み込み");
Texture2D tex = null;

tex = (Texture2D) EditorGUILayout.ObjectField(tex, typeof(Texture2D), false);
if (tex != null)
{
    FileStream fileStream = new FileStream(AssetDatabase.GetAssetPath(tex),
```

```
        FileMode.Open, FileAccess.Read);
    BinaryReader bin = new BinaryReader(fileStream);
    byte[] values = bin.ReadBytes((int) bin.BaseStream.Length);
    bin.Close();

    if (values != null)
    {
        paintTexture = new Texture2D(0, 0, TextureFormat.ARGB32, false);
        paintTexture.LoadImage(values);
    }
}
```

void Rendering()にて GUI.DrawTextureを使い、paintTextureを描画します（リスト 4.12）。paintTexture は設定されていないこと（Null になっている）があるため、paintTexture が Null かどうか判定し、Null でなければ描画します。

▼リスト 4.12　paintTexture を描画

```
    if (GUILayout.Button("やり直し"))
    {

    }
}

if (paintTexture != null)
{
    // 画像を描画
    GUI.DrawTexture(new Rect(GuiSpace, TextureHeightPos, paintTexture.width,
      paintTexture.height), paintTexture, ScaleMode.StretchToFill, true);
}
```

マウスでペイント

大まかなロジックとしては次のような構成になっております。

1. マウス管理クラスを更新
2. マウスの座標を取得
3. マウス座標が画像領域に入っているか確認
4. マウス座標に位置している画像のピクセル情報を上書き

最初にマウスを管理する static クラスを作成します（リスト 4.13）。
PaintEditorクラスの中に作成しましょう。主にこのクラスで管理するのはマウスの状態（クリック、ドラッグ）と座標の２点になります。汎用性を持たせるためにクラスを作成して運用しています。今回、そこまで汎用性を持たせる必要性はありませんが、読者にコピーして使ってもらえればと思いクラス化させています。

▼リスト 4.13　マウス管理クラス作成

```
/// <summary>
/// マウス管理
/// </summary>
static class Mouse
{
    static Event currentEvent;
    static Vector2 position;

    // マウスに関するイベントの情報を更新する。
    public static void Update()
    {
        currentEvent = Event.current;
        position = currentEvent.mousePosition;
    }

    // マウスのボタンが離れた時
    public static bool Up()
    {
        return currentEvent.type == EventType.MouseUp;
    }

    // マウスのボタンが押し続けている時
    public static bool Drag()
    {
        return currentEvent.type == EventType.MouseDrag;
    }

    // マウスの座標
    public static Vector2 GetPosition()
    {
        return position;
    }
}
```

次に void DataUpdate()にマウスでペイントする処理を記述していきます（リスト 4.15）。

ペイントするために必要な編集する画像の変数を用意します。

▼リスト 4.14　ペイントするテクスチャ変数を作成

```
  int paintSize = 1;

  // 編集するテクスチャ
  Texture2D paintTexture;

  [MenuItem("Tools/PaintEditor")]
```

また、void Update()を追加して Repaint()を入れています。実は OnGUIは UI を更新した時などの再描画が必要になった時にしか実行されません。画像にペイントするため、マウスの座標が移動しながら着色出来るようにします。そのため、毎フレームかけてエディタの描画処理を実行しています。

▼リスト 4.15　ペイント処理

```
// 編集するテクスチャ
Texture2D paintTexture;

bool isPaint = false;    // ペイント中かどうか
```

```
[MenuItem("Tools/PaintEditor")]
static void Create()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void Update()
{
    // エディタの再描画
    Repaint();
}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// 更新
/// クラスや変数を更新する
/// </summary>
void DataUpdate()
{
    // マウス情報更新
    Mouse.Update();

    // 着色更新
    PaintUpdate();
}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/// <summary>
/// マウスで着色していくペイント処理
/// </summary>
void PaintUpdate()
{
    // マウスが離された時、ペイント中なら
    if (Mouse.Up() && isPaint)
    {
        isPaint = false;
    }

    // マウスがドラッグ中ではなく paintTexture が設定されていない場合は処理をスキップ
    if (!Mouse.Drag() || paintTexture == null)
    {
        return;
    }

    // マウスの座標を取得して画像内にあるか確認、なければ処理をスキップ
    var pos = Mouse.GetPosition();
    if (pos.x < GuiSpace || pos.x > GuiSpace + paintTexture.width ||
        pos.y < TextureHeightPos ||
        pos.y > TextureHeightPos + paintTexture.height)
    {
        return;
    }

    // 着色するピクセル位置を取得　Y 座標は下から数えるため、最大値引き算で計算
    int paintPosX = (int) pos.x - GuiSpace;
    int paintPosY = paintTexture.height - ((int) pos.y - TextureHeightPos);

    // ペイントサイズによって着色する大きさを調整
    if (paintSize == 1)
    {
        paintTexture.SetPixel(paintPosX, paintPosY, paintColor);
    }
    else
    {
      // ピクセル位置を中心に着色
      // paintSize が 2 の場合、3x3 ドットを着色　3 の場合、5x5 ドットを着色
      int paintLength = paintSize * 2 - 1;
      Color[] paintColors = new Color[paintLength * paintLength];
      for (int i = 0; i < paintColors.Length; i++)
      {
          paintColors[i] = paintColor;
      }

      // paintColors をもとに着色　中心点が真ん中に来るように SetPixel する
      paintTexture.SetPixels(paintPosX - (paintSize - 1),
        paintPosY - (paintSize - 1), paintLength, paintLength, paintColors);
    }
```

```
    // paintTexture を更新
    paintTexture.Apply();

    // ペイント中を設定
    isPaint = true;
}
```

～Unity エディタ拡張の実装における関数化のポイント～

Unity エディタ拡張には、OnGUI という UI を描画するメインループが 1 本構成されており、基本的にそのメインループに処理を記述していきます。そのため、機能を追加するごとに if文が多くなる傾向になります。

if文が多いとソースコードが見にくくなるため、PaintUpdateのように処理を関数化して ifの際に return で返すように処理するとその後の処理を気にすることが無くなりプログラムは見やすくなります！

編集した画像を書き出し

void Rendering()の「画像書き出し」ボタンで paintTextureを書き出します（リスト 4.16）。
EditorUtility.SaveFilePanelを使うことで、ファイル書き出しする際によく見かけるエクスプローラの機能を再現できます。

▼リスト 4.16　画像書き出し

```
using (new GUILayout.HorizontalScope())
{
    if (GUILayout.Button("画像書き出し") && paintTexture != null)
    {
        string filePath = EditorUtility.SaveFilePanel(
          "画像書き出し", "Assets", "", "png");

        if (!string.IsNullOrEmpty(filePath))
        {
            File.WriteAllBytes(filePath, paintTexture.EncodeToPNG());
            AssetDatabase.Refresh();
        }
    }
}
```

ScriptableObject で自動保存

次に自動保存機能を実装していきます。
編集したデータを PaintEditDataに示すような ScriptableObject で保存、運用を行なっていきます。

ScriptableObject は、公式サイトに次のように記載されています。[*2]

> ScriptableObject は、クラスインスタンスとは無関係に、大量のデータを保存するために使用できるデータコンテナです。

要約すると Unity の変数を保存できるセーブデータのような働きをします。

シンプルに作成しやすく、Unity バージョン違いによる影響をあまり受けない点でとてもお勧めです。C#ではクラスのコピーを行う際、参照渡しされます。ここで読み込んだテクスチャを参照渡しすると、既存の読み込んだテクスチャを編集してしまいます。そのため、Texture2Dをコピーする際は newで Texture2Dクラスを新規作成します。SetPixelsを使いピクセル情報をコピーすることにより同じ Texture2Dを新規作成します。これにより、参照渡しを回避して運用できるようにしました。

▼リスト 4.17　ScriptableObject 作成

```
/// <summary>
/// 保存するエディタ編集データクラス
/// </summary>
[CreateAssetMenu]
public class PaintEditData : ScriptableObject
{
    // ペイントテクスチャ
    [SerializeField] Texture2D paintTexture;
    public Texture2D PaintTexture { get { return this.paintTexture; }
      set { this.paintTexture = value; } }

    // ペイントカラー
    [SerializeField] Color paintColor;
    public Color PaintColor { get { return this.paintColor; }
      set { this.paintColor = value; } }

    // ペイントカラー
    [SerializeField] int paintSize;
    public int PaintSize { get { return this.paintSize; }
      set { this.paintSize = value; } }

    /// <summary>
    /// コピーコンストラクタ
    /// </summary>
    public void Copy(PaintEditData copyData)
    {
        if (copyData.PaintTexture != null)
        {
            paintTexture = new Texture2D(copyData.PaintTexture.width,
              copyData.PaintTexture.height, TextureFormat.ARGB32, false);
            paintTexture.SetPixels(copyData.PaintTexture.GetPixels());
            paintTexture.Apply();
```

[*2] https://docs.unity3d.com/ja/2018.4/Manual/class-ScriptableObject.html

```
        }

        paintColor = copyData.PaintColor;
        paintSize = copyData.PaintSize;
    }
}
```

参照渡しについて簡単に解説します。

▼リスト 4.18　参照渡し解説処理

```
class Data
{
  public int value = 0;
}

void Start()
{
  Data a = new Data();
  Data b = a;
  a.value = 10;
  Debug.LogError("b.value = " + b.value);
  // 「b.value = 0」ではなく「b.value = 10」とログに出ます。
}
```

リスト 4.18 のプログラムを見てください、a という Data クラスを用意して同じく Data クラスの b を a に代入して a の変数 value に 10 を入れています。この際の b.value の値は初期値の 0 が入っているかと初見の方は思われますが、実際には b.value には 10 が入っています。これは個人的要約すると b は a の代用状態になっているからです。これを防ぐにはリスト 4.19 のように Data クラスを New して b に入れなければいけません。

▼リスト 4.19　参照渡し改善解説処理

```
  Data a = new Data();
  Data b = new Data();
  a.value = 10;
  Debug.LogError("b.value = " + b.value);
  // 「b.value = 0」と表示されます。
```

まだ解説しないといけない点はありますが、リスト 4.18 のような挙動を防ぐ手段を ScriptableObject を運用する際に利用します。

PaintEditorのメンバ変数として使用していた「paintColor」「paintSize」「paintTexture」を ScriptableObject のものに置き換えます（リスト 4.20)。プロパティの値が編集されると ScriptableObject が編集されるので自動保存されるようになります。

▼リスト 4.20　ScriptableObject を使った自動保存

```
const int RedoUndoStockCount = 20;  // アンドゥ・リドゥで使用する最大ストック数

// 自動保存データ
[SerializeField] PaintEditData autoSave = null;

// 着色する色
Color paintColor
{
    get { return autoSave.PaintColor; }
    set { autoSave.PaintColor = value; }
}

// 着色するブラシサイズ
int paintSize
{
    get { return autoSave.PaintSize; }
    set { autoSave.PaintSize = value; }
}

// 編集するテクスチャ
Texture2D paintTexture
{
    get { return autoSave.PaintTexture; }
    set { autoSave.PaintTexture = value; }
}
```

void Awake()はエディタを開いた時に最初に 1 回だけ実行される関数です。今回はそこに自動保存のデータ読み込み処理を組み込みます（リスト 4.21）。もし自動保存のデータが存在しなかった時は ScriptableObject.CreateInstanceで ScriptableObject を作成して、AssetDatabase.CreateAssetで新規アセットとして保存します。注意しないといけないのが AssetDatabase.CreateAssetを使用した後は AssetDatabase.Refreshで更新をかける必要がある点です。

▼リスト 4.21　エディタの初期化処理

```
[MenuItem("Tools/PaintEditor")]
static void Create()
{
    GetWindow<PaintEditor>("PaintEditor");
}

// エディタ初期化
void Awake()
{
    // 自動保存の PaintEditData を格納するパス
    var path = "Assets/AutoSave.asset";
    // PaintEditData を格納するパスよりデータを取得
    autoSave = (PaintEditData) AssetDatabase.LoadAssetAtPath(path,
      typeof(PaintEditData));

    // パスにデータがない場合、新規作成
    if (autoSave == null)
    {
        autoSave = ScriptableObject.CreateInstance<PaintEditData>();
        AssetDatabase.CreateAsset(autoSave, path);
        AssetDatabase.Refresh();

        // 自動保存データの初期値セット
        autoSave.PaintSize = 1;
        autoSave.PaintColor = Color.black;
    }
}
```

保存データをもとにアンドゥ・リドゥ実装

自動保存の ScriptableObject を編集するごとに複製してアンドゥ・リドゥした際に ScriptableObject をロードします（リスト 4.22)。
これで手軽にアンドゥ・リドゥを実装できます。

この処理は `void Initialize()`内に記述します。`DataUpdate`で利用する変数を、その呼び出し前にセットアップするためです。

▼リスト 4.22　ScriptableObjectRedoUndo

```
// 自動保存データ
[SerializeField] PaintEditData autoSave = null;

// アンドゥ・リドゥのモード Enum
enum RedoUndoMode
{
    None = 0,
    Undo,
    Redo,
}

// リドゥ・アンドゥ
int RedoUndoBaseIndex;                       // 現在のリドゥカウント
List<PaintEditData> StockEditDataList;    // 編集データをストックしておくリスト
RedoUndoMode isMode = RedoUndoMode.None;  // リドゥかアンドゥを行なったか保持
〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜

        // 自動保存データの初期値セット
        autoSave.PaintSize = 1;
        autoSave.PaintColor = Color.black;
    }

    // アンドゥ・リドゥ用のローカル変数を初期化
    RedoUndoBaseIndex = 0;
    StockEditDataList = new List<PaintEditData>();

    // 初期状態をストックとして保存する
    StockEditData();
}
〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜
// マウスが離された時、ペイント中なら
if (Mouse.Up() && isPaint)
{
    isPaint = false;

    // ペイント編集アクション保存
    StockEditData();
}
〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜
void Initialize()
{
    // 編集アクションが登録されていない場合またはリドゥ・アンドゥ更新添字が無効状態の時
    if (StockEditDataList == null || StockEditDataList.Count <= 0 ||
      RedoUndoBaseIndex < 0)
    {
        isMode = RedoUndoMode.None;
        return;
    }

    // リドゥ・アンドゥ更新実行処理
    switch (isMode)
```

```
    {
        case RedoUndoMode.None:
            return;
            break;
        case RedoUndoMode.Undo:
            if (RedoUndoBaseIndex - 1 < 0)
            {
                return;
            }
            RedoUndoBaseIndex--;
            break;
        case RedoUndoMode.Redo:
            if (RedoUndoBaseIndex + 1 > StockEditDataList.Count - 1)
            {
                return;
            }
            RedoUndoBaseIndex++;
            break;
    }

    autoSave.Copy(StockEditDataList[RedoUndoBaseIndex]);

    // モードリセット
    isMode = RedoUndoMode.None;
}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (GUILayout.Button("元に戻す"))
{
    isMode = RedoUndoMode.Undo;
}

if (GUILayout.Button("やり直し"))
{
    isMode = RedoUndoMode.Redo;
}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

using (new GUILayout.VerticalScope(GUILayout.Width(
  BrushSpaceWidth + GuiSpace)))
{
    EditorGUI.BeginChangeCheck();

    paintColor = EditorGUILayout.ColorField("着色", paintColor);
    paintSize = EditorGUILayout.IntField("サイズ", paintSize);

    if (EditorGUI.EndChangeCheck())
    {
        StockEditData();
    }
}
```

4.4 まとめ

いかがでしたでしょうか。本章で紹介した Unity エディタ拡張は、デザイナー向けに開発した作業効率化ツールを参考にしています。そして、初心者にもわかりやすく各機能をピックアップしてまとめてみたものです。実際に内定者アルバイトの業務の一環として作業効率化ツールの不具合の対応や機能の追加をしてもらっていました。そのため、保守性や拡張性においては実績ある設計だと個人的に思います。

作業効率化ツールを業務で利用してもらっている中で「巻き戻し機能が欲しい」「作業を中断したところから再開したい」「作業をプロジェクトデータ化して他人と共有したい」

といった要望がありました。このとき、なるべくシンプルな実装で関数毎に処理を分けて記述していたため、簡単に対応することができました。このように、新たな要望に応えるためにも、設計をしっかり行うことが望ましいと筆者は思います。

今回、紹介した設計は最適解ではないと思いますが、本章の内容が読者の皆様のこれからの Unity エディタ拡張等の参考になれれば幸いです。

4.5 作成したツールのソースコード（URL）

今回紹介した画像ペイントツールは Unity パッケージとして、GitHub 上に置いています。

`https://github.com/YuukiHirai0331/TechBook_UnityEditorPaint`

第 5 章

itertools repeat を使ってみて読んでみる

Shunsuke Ito / @fgshun

5.1 はじめに

Python の標準ライブラリのひとつ itertools モジュール。便利なライブラリです。この章では、そのモジュールの中から `repeat`[*1]について紹介してみます。単純ながら侮れない、そんなイテレータです。

5.2 動作の紹介

`repeat(element[, times])`は与えられたオブジェクトを `times`回くりかえすイテレータをつくります。`times`引数を省略すると無限長のイテレータとなります。

▼リスト 5.1　動作の紹介

```
>>> import itertools
>>> it = itertools.repeat('spam')
>>> next(it)
spam
>>> for s in itertools.repeat('spam'):
    print(s)
spam
spam
   ……無限に出力されて停止しない。
```

停止しないのは困るので、普通は何らかの方法で停止するように工夫して使う必要があります。一見するとなんの役に立つのかわからないイテレータです。事実、単体ではあま

[*1] https://docs.python.org/ja/3/library/itertools.html#itertools.repeat

り意味をなしません。そんな repeatについて、まずは有用に使ってみる例をひとつあげてみます。

5.3 使用例

repeatは、実行時にしか長さのわからない有限の長さのイテレータとともに用いると便利です。簡単な例として、整数値の列を受け取ってすべての値を二倍にする処理を考えます。これを実現する方法としてまず思いつくものとしては for ループがあります。

▼リスト 5.2　for ループによる実装

```
input_ = [0, 1, 2, 3]
output = []
for i in data:
    output.append(i * 2)
# [0, 2, 4, 6] の list が得られる
```

これと同等の処理を mapを使って書き換えるとリスト 5.3 のようになります。

▼リスト 5.3　map と lambda による実装

```
output = map(lambda v: v * 2, input_)
# 0, 2, 4, 6 と値を返すイテレータが得られる
```

ところで、値をふたつ受け取り乗算する関数として operator.mulが存在しています。これを流用できないか[*2]を考えます。

▼リスト 5.4　map と mul と list による実装

```
from operator import mul
temp = [2] * len(input_)
output = map(mul, input_, temp)
```

これで既存の処理の流用ができました。しかし、この方法では入力と同じ長さのリスト tempを用意しています。これはいままでの実装と比べるとリストの用意が無駄な処理に見えます。入力がネットワーク越しにすこしずつ届く場合、長さを確認するための待ち時間が無駄になりえますし、入力が巨大になるとリスト作成が不可能となることもありえます。

困ったことになりました。mulを用いた実装には無理があったのでしょうか？ いいえ、このような時こそ repeatの出番です。

[*2] 高品質の実装が既存であるのであれば流用し再実装を避けるのは自然なことでしょう。ただの乗算といった単純なものであれば lambda で都度実装しても問題ないのですが。

▼リスト 5.5　map と mul と repeat による実装

```
from operator import mul
output = map(mul, input_, repeat(2))
```

このように、repeatはリスト 5.3 の lambda内に現れる定数の役割をになうことができるのです。repeatのくり返し回数に制限がないという性質はこのようなケースで有用となります。

5.4　repeat を自作する

repeatと同等の処理を記述するにはどのような方法があるでしょうか？ もっとも平易に書けてお勧めなのは、while無限ループを用いたジェネレータ関数でしょう。yieldにより処理が中断・再開するため、ジェネレータ関数には停止条件のない whileループを書くことができます。

▼リスト 5.6　while ループとジェネレータ関数で repeat を模倣する

```
def repeat(element, times=None):
    if times is None:
        while True:
            yield element
    else:
        for i in range(times):
            yield element
```

他にも、イテレータプロトコルを実装したクラスとして実装することも可能です。イテレータプロトコルに沿ったクラスに必要なのは、自身を返す__iter__関数と値を返す__next__関数です。また、インスタンス属性としてくりかえし返す値そのものと、あと何回返すのかの回数をもつ必要があります。イテレータプロトコルでは返す要素が尽きたときは StopIteration例外によって通知することが要求されています。このため回数が残り 0 となったときに raise StopIterationが実行されるようにします。また、例外を出すことなく値を返し続けることで無限長のイテレータとすることができます。

▼リスト 5.7　クラス文で repeat を模倣する

```
class repeat:
    def __new__(cls, element, times=None):
        obj = super().__new__(cls)
        obj.element = element
        obj.cnt = -1 if times is None else times
        return obj

    def __iter__(self):
        return self

    def __next__(self):
```

```
        if self.cnt == 0:
            raise StopIteration
        if self.cnt > 0:
            self.cnt -= 1
        return self.element
```

5.5 repeat の実装を読む

CPython における repeatの実装である C 言語のコード[*3]を読んでみます。動作が単純であるため、その実装もリスト 5.7 と同等の簡素なものとなっています。Python/C API を用いるコードを読み書きする際の入門用としてお勧めできます。

まずは repeatobject構造体。くりかえし返す値 elementとあと何回返すのかの回数 cntがあることがわかります。

▼リスト 5.8　repeatobject 構造体

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

つぎは__new__。PyArg_ParseTupleAndKeywords[*4]で引数のパースが、tp_allocで repeatオブジェクト用のメモリの確保がおこなわれます。末尾の 3 行は Python コードに近い見た目をしており、おこなわれていることも実際そっくりです。

C API 使用のコードならではの特徴は、参照カウントのインクリメント操作（Py_INCREF）です。ここでは引数として渡された elementの参照を 1 増やしています。これを怠った場合、まだ使用しているはずの elementオブジェクトが意図せぬタイミングでガベージコレクションに回収されてバグの原因になります。Pure Python でコードを書いていればいらぬ注意であり、CPython 用の C コードを書くにあたってめんどうで神経を使う要素のひとつです。

▼リスト 5.9　__new__関数

```
static PyObject *
repeat_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    repeatobject *ro;
    PyObject *element;
    Py_ssize_t cnt = -1, n_args;
    static char *kwargs[] = {"object", "times", NULL};
    // 中略
```

[*3] https://github.com/python/cpython/blob/v3.9.1/Modules/itertoolsmodule.c#L4220-L4374

[*4] https://docs.python.org/ja/3/c-api/arg.html

```
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "O|n:repeat", kwargs,
                                     &element, &cnt))
        return NULL;
    // 中略
    ro = (repeatobject *)type->tp_alloc(type, 0);
    if (ro == NULL)
        return NULL;
    Py_INCREF(element);
    ro->element = element;
    ro->cnt = cnt;
    return (PyObject *)ro;
}
```

__iter__は既存の PyObject_SelfIterが用いられています。その実装は object.c[*5]にありリスト 5.10 となっています。ここでも自身の参照カウントを Py_INCREFで 1 増やして返します。

▼リスト 5.10　__iter__関数

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

__next__はリスト 5.11 のようになっています。これにも返す値の参照カウントのインクリメント操作が含まれています。cntの操作はリスト 5.7 と同じです。異なるように見える点は StopIteration例外をセットすることなくただ NULLを返していることです。通常、このようにただ NULLを返すと Python インタプリタはこれを実装ミスとみなし SystemError例外に変換します。しかし、イテレータプロトコルでは StopIteration例外を設定してもしなくてもよい[*6]とされているため、これは同じように動作します[*7]。

▼リスト 5.11　__next__関数

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

[*5] https://github.com/python/cpython/blob/v3.9.0/Objects/object.c#L1052-L1057

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

[*7] https://github.com/python/cpython/blob/v3.9.1/Objects/abstract.c#L2683-L2700 。同じように動作するのは repeat_next を呼び出す PyIter_Next が StopIteration 例外をクリアするためです。

5.6 おわりに

itertoolsは駆使することでコードを平易に記述することが可能となる便利なものです。シンプルながら掘り下げてみると Python の言語仕様、イテレータプロトコルについて見えてくるものがあったりと面白いものでもあります。ぜひ使って、そして読んでみてください。そして他のイテレータが難しく感じたときには、repeatに立ち戻って読んでみるのがお勧めです。それでは、よき Python ライフを！

第 6 章

Discord と Slack の架け橋

Shinya Naganuma / @Pctg_x8

この章では、Slack 向けに作っていた bot を Discord に移行するお話をします。

個人でほそぼそと開発しているゲームエンジン「Peridot[*1]」では、GitHub と Slack をベースとした簡易的な ChatOps 環境を構築しています。この中で利用している Slack を Discord に置き換えようと思い、この記事を書いている時点ではまだ途中なのですが、これまでに得られた Slack と Discord の違いや、移行するにあたっての注意点などのお話をします。

ちなみに Discord に移るのは、単にデザインが好み的なレベルの話だったりします。

6.1 Incoming Webhook とテキストフォーマットの話

Slack にも Discord にも、共通するメッセージ投稿の仕組みとして Incoming Webhook[*2][*3]が存在します。Incoming Webhook は簡単にいうと「サービス側が発行した特定の URL に対してリクエストを行うことで、特定のチャンネルにメッセージを投稿できる」機能です。

Slack にも Discord にも別途メッセージを投稿する API は存在しますが、Slack の場合は任意の API トークンさえあれば投稿できるのに対して、Discord の API は **Gateway** と呼ばれる機構で認証を通す必要があり、メッセージを投稿するためだけに使用するには少し手間がかかります。そのため、Discord に CI の結果通知や GitHub 上のアクティビティを通知するといったことをする場合には、基本的には Webhook を使用する方が簡単です。

さて、Discord も Slack と同じくメッセージをフォーマットすることができます。このフォーマットは、Slack では Markdown に似ているものの独自色の強い書式なのに対し、

[*1] https://github.com/Pctg-x8/peridot
[*2] https://api.slack.com/messaging/webhooks
[*3] https://discord.com/developers/docs/resources/webhook

Discord ではほとんど純粋な Markdown のサブセットとなっています。

リンクフォーマット

Slack と Discord のフォーマットでもっとも差分が大きいのがリンクのフォーマットです。Slack では、リスト 6.1 のように角かっこと縦線を使用することで任意の文字列にリンクを張ることができます。

▼リスト 6.1　Slack のリンク

```
<URL|テキスト>
```

一方で、Discord では Markdown そのままの記法となっています。

▼リスト 6.2　Discord(Markdown) のリンク

```
[テキスト](URL)
```

Slack と Discord を同時に利用する場合、この違いがもっとも大きな壁になります。見てのとおり URL とテキストの位置が逆になるので、単純にフォーマット用に差し込む文字列を切り替える方式では対応できません。

この場合の対処方法は色々あるかと思われますが、今回はリンク書式の抽象化として専用のフォーマッタを用意することにしました（リスト 6.3）。Peridot でチャットへの投稿を管理している Lambda 関数では、開発言語として Rust を使用しているため、この手の抽象化をゼロコストで行うことができます。

▼リスト 6.3　リンクフォーマッタ

```
// 共通コンストラクタ
pub trait LinkFormatter<Text: Display, Url: Display> : Sized + Display {
  fn new(text: Text, url: Url) -> Self;
}

// 各種フォーマッタ定義
pub struct SlackLinkFormatter<Text: Display, Url: Display>(Text, Url);
impl<Text, Url> LinkFormatter<Text, Url> for SlackLinkFormatter<Text, Url>
    where Text: Display, Url: Display {
  fn new(text: Text, url: Url) -> Self { Self(text, url) }
}
pub struct MarkdownLinkFormatter<Text: Display, Url: Display>(Text, Url);
impl<Text, Url> LinkFormatter<Text, Url> for MarkdownLinkFormatter<Text, Url>
    where Text: Display, Url: Display {
  fn new(text: Text, url: Url) -> Self { Self(text, url) }
}

// 具体的なフォーマット定義
impl<Text, Url> Display for SlackLinkFormatter<Text, Url>
    where Text: Display, Url: Display {
  fn fmt(&self, fmt: &mut Formatter) -> std::fmt::Result {
```

```
    write!(fmt, "<{}|{}>", self.1, self.0)
  }
}
impl<Text, Url> Display for MarkdownLinkFormatter<Text, Url>
    where Text: Display, Url: Display {
  fn fmt(&self, fmt: &mut Formatter) -> std::fmt::Result {
    write!(fmt, "[{}]({})", self.0, self.1)
  }
}

// 使用例
fn usage<LF: LinkFormatter<&'static str, &'static str>>() -> String {
  format!(
    "{}の{}",
    LF::new("このブランチ", "https://github.com/Pctg-x8/peridot/tree/dev"),
    LF::new(
      "このファイル", "https://github.com/Pctg-x8/peridot/blob/dev/src/lib.rs"
    )
  )
}
let text_slack = usage::<SlackLinkFormatter>();
let text_discord = usage::<MarkdownLinkFormatter>();
```

Emoji（Custom Emoji）

Discord でも Slack と同じように Emoji を使用することができます。ただし、bot の投稿内容に含める場合は、Discord の場合だけ少し複雑です。

Slack の場合は、リスト 6.4 のように Emoji の名前を : で囲むことで Emoji が挿入されます。これは一般ユーザーの入力と同じ挙動です。

▼リスト 6.4　Slack の Emoji

```
:emoji:
```

一方で、Discord ではユーザー入力の場合は Slack と同じように : で囲めば挿入されますが、bot から送信する場合にはリスト 6.5 のように Emoji の ID が必要になります。

▼リスト 6.5　Discord の Emoji

```
<:emoji:214520500443>
```

さらに Discord では通常の Emoji とアニメーションする Emoji が区別されており、これを bot から送信する際にはリスト 6.6 のような形をとる必要があります。

▼リスト 6.6　Discord の Emoji（Animated）

```
<a:emoji:45443320459>
```

ID が必要な関係上、先のリンクフォーマットのようにフォーマッタを分けるだけでは足りません。Discord の場合にはフォーマットのタイミングで Emoji 一覧取得 API を使用し、該当 Emoji の ID とアニメーション Emoji かどうかの判定を行う必要があります。

アタッチメント

Slack には**アタッチメント**という、メッセージに補助的なデータを付加する機能がありますが、Discord にも **Embed** という名称で似たような機能が存在します。フィールドや画像の埋め込みも行える点では Slack のアタッチメントとほとんど同じように扱うことができます。

▼表 6.1　Attachment と Embed

箇所	Slack	Discord	備考

Author 情報[*4]	`author_name`
	`author_link`
	`author_icon`
		`author`	Discord ではオブジェクトを指定する

タイトル	`title`	`title`	
本文	`text`	`description`	
左端の色指定	`color`	`color`	Slack ではプリセットカラー（danger など）が使える
フォールバック[*5]	`fallback`	なし	
フィールド	`fields`	`fields`	表 6.2 に示すオブジェクトを指定する

▼表 6.2　Field オブジェクトの対応表

箇所	Slack	Discord
タイトル	`title`	`name`
値	`value`	`value`
短縮表示をするか？	`short`	`inline`

unfurl

Slack では、URL を含む投稿を行なった場合に自動でアタッチメントに URL のプレビューを展開する **unfurl** という挙動が存在します。Discord でも URL をアタッチメント（Embed）に展開する挙動は同じように動作します。

Slack と Discord で違う点としては、Twitter など SNS のようなコンテンツの unfurl が Discord では標準で提供されているところです。Slack で Twitter リンクを unfurl す

[*4] タイトルより上に表示され、アタッチメントの内容の作者情報を表します。たとえば、Twitter App はリンクを unfurl するときにユーザー ID やアイコンをこの欄に設定します。

[*5] 通知などで、アタッチメントの内容の代わりとして展開されるテキストを指定します。

るためには、ワークスペースに Twitter App をインストールして適切なアカウントで認証を行う必要がありますが、Discord では Twitter 連携はアカウントに紐づいているため、特別な Bot などがなくてもプレビューを展開してくれます。

アカウントオーバーライドの挙動

Slack と Discord のどちらも、アカウント名やアイコンをメッセージ単位でオーバーライド（上書き）する機能が存在します。それぞれのサービスでの該当パラメータ一覧を表 6.3 に示します。

ただし、Slack でアカウントオーバーライドを行う場合は `chat:write.customize`権限がトークンに付与されている必要があります。

▼表 6.3　アカウントオーバーライドに使うパラメータ

サービス	アカウント名指定	アイコン指定（emoji）	アイコン指定（url）
Slack	`username`	`icon_emoji`	`icon_url`
Discord（Webhook）	`username`	なし	`avatar_url`

Discord でも Slack と同じようにカスタム Emoji をアイコンに使用することができますが、Discord の API には Emoji をアイコンに指定するパラメータは存在しません。このため、まず Emoji 一覧取得 API を用いて Emoji の実体の URL を取得し、それをアイコンに設定するという手順を踏む必要があります。

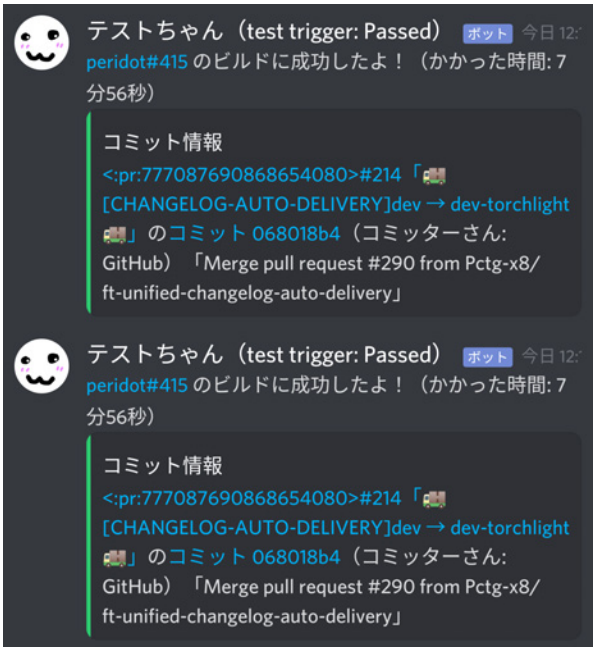
ところで、Slack も Discord も共通の挙動として「アイコンだけ変更してもタイムライン上では再表示されない」といった問題があります（図 6.1）。CI の結果によってアイコンを変えるようなことをしたい場合、この仕様は不便なものとなります。

ただし、ユーザー名を変更すればアイコンも含め再表示されるため、ユーザー名に細工を施すことでアイコンだけの変更を強制的に再表示させたような表現を行うことができます。文字列に細工を施すときによく使われるのは**ゼロ幅スペース**かと思いますが、残念ながらこれは Slack でのみ認識されて Discord では無視されてしまいます（図 6.2）。

一方で、モバイル版の Discord クライアントでは毎回再表示される仕様のため（図 6.3）、特に対策をしなくてもアイコン変更を表現できます。

▲図 6.1　連続して投稿するとユーザー名とアイコン表示が省略される（左：Slack、右：Discord）

▲図 6.2　ゼロ幅スペースを仕込んだ場合、Slack のみ別名として再表示される（左：Slack、右：Discord）



▲図 6.3　iOS/Android 版 Discord では毎回表示される

Discord Webhook API の Slack/GitHub 互換フォーマット

Discord の Webhook API のドキュメントを見ると、Slack や GitHub と互換性がありそうな API が存在します[*6]。これらの API を使用した場合、上記のフォーマット変換もある程度行われるようです。ただし Emoji だけは ID が必要なためか変換されませんでした。
とはいえ内容のフォーマット変換まで行ってくれるのはかなり強力なため、`icon_emoji`を使っていないのであれば、今回のように Slack からの移行の際に試してみてもよいでしょう。

[*6] `https://discord.com/developers/docs/resources/webhook#execute-slackcompatible-webhook`

6.2 Gateway と RTM の話

Slack と Discord のどちらにも、WebSocket を使用してリアルタイムにメッセージングを行う API があります。Slack では **RTM**（RealTime Messaging API）、Discord では **Gateway** と命名されています。

Slack の RTM は、この文章を執筆している時点（2020 年 11 月）では classic bot か `client`権限が付与されたトークンでのみ利用できる API となっています。新しく Socket Mode と呼ばれる API もあるみたいですが、執筆時点では該当する権限をトークンに付与できず、まだ使えないようです[*7]。

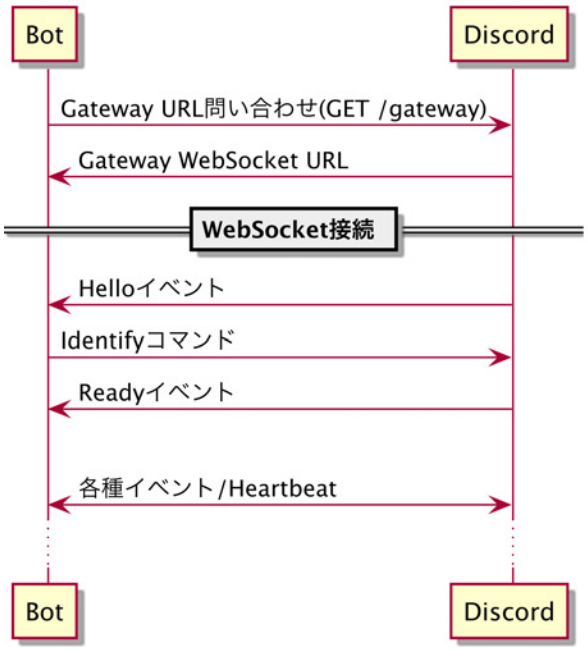
両者とも、テキストフレームに JSON 形式でデータをのせて、イベントが発生するごとに送ってくる点は同じです。

認証ルート

Slack では、RTM を開始する際に `rtm.start`または `rtm.connect`を発行して WebSocket の URL を得ます。この API を発行するタイミングでトークンによる認証が必要になるため、WebSocket 通信中には特に認証作業などは必要ありません。

一方 Discord では、WebSocket を接続するまでの間に認証作業が一切入りません。Discord の Gateway では WebSocket 通信フローとして図 6.4 のような形をとり、**Identify** コマンドではじめて bot を認証します。

[*7] ドキュメントも存在しませんが、なぜか npm には参考実装があります: `https://www.npmjs.com/package/@slack/socket-mode`



▲図 6.4　Gateway の通信フロー

Heartbeat（Ping/Pong）

定期的に Heartbeat（Ping）を送る点も、両者ともに同じです。WebSocket 経由で指定のメッセージを送る形をとります。Heartbeat の間隔は、Slack では特に指定がなく**数秒おきに**（every few seconds）となっています。参考値としては、Slack API の Go ライブラリ[*8]ではデフォルトで 30 秒間隔で Ping メッセージを発行するようになっています。Discord では Heartbeat の間隔が Hello イベントのデータにミリ秒単位で入っているため、特に何もなければその時間に従って Heartbeat コマンドを発行します。

Intent（Discord のみ）

Discord の Gateway には、要求するメッセージのタイプをある程度選択できる **intent** と呼ばれるパラメータがあります。これは初回の Identify コマンドに含めるもので、要求するメッセージをビットフラグで指定します。

[*8] `https://github.com/slack-go/slack/blob/master/rtm.go`

▼表 6.4　intent フラグ一覧と簡単な説明

フラグ値	説明
0x0001	ギルド（サーバ）の生成、更新やチャンネルの作成など チャンネルの pin 更新イベントもここ
0x0002	ギルドメンバーの追加、削除など
0x0004	ギルドメンバーの BAN 情報の変更
0x0008	ギルドへの Emoji 追加など
0x0010	ギルドへの **Integration**（bot など）の追加など
0x0020	ギルドの Incoming Webhook の更新
0x0040	ギルドへの招待の作成、削除
0x0080	ギルドのボイスステートの更新
0x0100	ギルドの Presence の更新
0x0200	ギルドへのメッセージの作成、更新など
0x0400	ギルドのメッセージへのリアクションの付与、削除など
0x0800	ギルドメンバーのタイプ中情報
0x1000	DM のメッセージの作成、更新など
0x2000	DM のメッセージへのリアクションの付与、削除など
0x4000	DM でのタイプ中情報

6.3　おわりに

もともと Discord の bot 作成についての記事を書く予定だったのですが、それだけでは普通に情報も出回っていて面白くないかなと思い、Slack との対比の形で書いてみました。

両方使ってみた感想ですが、どちらも良い点とよくない点が同じくらいあるので一概にどっちがいいかは決められない感じになっています。Discord は通知にアイコンが出てくれるので結構見た目が楽しいのですが、リスト表記が使えなかったり Emoji の数に制限があったりしてそこが微妙な点です。それから、Discord のアタッチメントは上下に妙な空白が入るので、そのデザインに関しては Slack の方が個人的には好みです。

第 7 章

会社の有志で技術同人誌を書き続けている話

Kinuko MIZUSAWA

早いもので今回で Vol. 7 になる KLab Tech Book。筆者も初回から本文記事や取りまとめなど、何らかの形で関わってきています。

この章では、これまで制作してきた KLab Tech Book の歴史を振り返り、その制作の流れをご紹介します。

その中でも、筆者が特にお伝えしたいのは会社の有志で技術同人誌を作る意義、楽しさです。

この章を通じて、わたし達がどのように「楽しんで」技術同人誌を作っているのかが伝わればさいわいです。

7.1 KLab Tech Book の歴史

KLab ではこれまでに本誌を含めて 7 冊の技術同人誌を発行しています。ここでは、その歴史を振り返っていきます。なお、ここで紹介するすべての既刊の電子版を次のページで公開していますので、ご興味のある記事がありましたらぜひご覧ください。

`https://www.klab.com/jp/blog/tech/2020/tbf10.html`

Vol. 1（2017/10 発行）

▲図 7.1　KLab Tech Book Vol. 1

1. 物理ベースレンダラーを Rust 実装して、表紙絵をレンダリングした話
   （Sho HOSODA）
2. Sprache を CPS 変換 （岡本和樹）
3. Emscripten で動画再生する （やまだ）
4. テキストマクロプロセッサ「M4」のチューリング完全性について
   （Daisuke MAKIUCHI）
5. FPGA 初心者が試行錯誤しながら擬似乱数生成回路を作る話 （Suguru OHO）
6. 家庭内ネットストーカーシステムを作った話 （Yoshio HANAWA）
7. とある KLab のスマホアプリのビルド事情 （Kinuko MIZUSAWA）

Vol. 2（2018/04 発行）

▲図 7.2　KLab Tech Book Vol. 2

1. Cygwin 環境で Docker を快適に使うために（Daisuke MAKIUCHI）
2. Unity で開発しているスマホゲームで Xcode のバージョンアップをする（Kinuko MIZUSAWA）
3. Re:VIEW で書いた原稿の CI を AWS CodeBuild で回す（Kinuko MIZUSAWA）
4. WebAssembly で行列の演算をする（やまだ）
5. Python インスタンスの属性は（dict）で管理されているのか調べてみた（Shunsuke Ito）
6. リアルの街を Unity に取り込む　〜GIS へのいざない〜（Suguru OHO）

Vol. 3（2018/10 発行）

▲図 7.3　KLab Tech Book Vol. 3

1. プロシージャルモデリングを支える Houdini の機能紹介（Kinuko MIZUSAWA）
2. 2.5 万円で買える 3D プリンタのススメ（黒井春人）
3. Airtest を用いた Unity アプリの自動実機テスト（Daisuke TAKAI）
4. Rider+Unity で Roslyn Analyzers を使う（Yoshihiro KASHIMA）
5. バーコードリーダーになろう（Daisuke MAKIUCHI）
6. Unity Timeline Tips 集（Junichi KIKUCHI）
7. 物理ベースレンダラーを Rust 実装して、ちょっと高速化した話
   （Sho HOSODA）
8. ヘッドレス Chrome でリボ払いを回避している話（Yoshio HANAWA）

Vol. 4（2019/04 発行）

▲図 7.4　KLab Tech Book Vol. 4

1. Argument Clinic を使ってみよう（Shunsuke Ito）
2. Rust で世界を統一する（Shinya Naganuma）
3. QR コードマニアックス　ー数字・英数字・漢字モード（Daisuke Makiuchi）
4. Unity ×レイマーチングによる映像制作の実践手法（Sho HOSODA）
5. 自作キーボード入門（Suguru OHO）
6. 機械学習 API の力で CAPCHA を破る（Yoshio HANAWA）
7. Unity での条件付きコンパイルシンボル定義にエディタ拡張を活用する（Kinuko MIZUSAWA）
8. デジカメで撮影した写真の正確なタイムスタンプを推測する（@muo_jp）

Vol. 5（2019/09 発行）

▲図 7.5　KLab Tech Book Vol. 5

1. Starlette - きらめく ASGI フレームワーク（Shunsuke Ito）
2. Vulkan メモリマネジメント / RenderPass 虎の巻（Shinya Naganuma）
3. 2 次元コード DataMatrix（Daisuke Makiuchi）
4. オフィス内の C02 濃度を測るデバイスを作成する（黒井春人）
5.「OK グーグル！　銀行振込 1000 円」（Yoshio HANAWA）
6. GoogleCalendar を扱う Slack Bot の制作記
   （Toshifumi Umezawa & Ayato Takeichi）

Vol. 6（2020/2 発行）

▲図 7.6　KLab Tech Book Vol. 6

1. NVIDIA OptpiX「レイマーチング×パストレーシング」による物理ベースレンダラーを自作する（Sho HOSODA）
2. Unity の描画ラインをハイジャックして遊ぶ（Shinya Naganuma）
3. Unity 2D Animation でキャラクターを動かす（Kinuko Mizusawa）
4. Arduino UNO を USB 接続の HID デバイスにする（Toshifumi Umezawa）
5. バーコードリーダーになろうー Code128 編（Daisuke Makiuchi）
6. Python 組み込み関数マニアックス（Shunsuke Ito）
7. 退屈なことは Emacs にやらせよう（Ryota Togai）
8. CloudFront-WAF 制御下で Lambda@Edge を利用して ReactSPA を動かす（Daisuke Yamaguchi）
9. 天下一 GameBattleContest（β）の裏側（Takuya Hashimoto）

KLab Tech Book は、各章の著者自身が書きたい技術ネタを書くという方針で作成してきました。このため、会社での業務とは関係のない内容の記事が多くなっています。

また、Vol. 1 の表紙はエンジニアがプログラムにより描画した画像で、その描画手法を記事としても掲載しています。一方、Vol. 2 以降は社内のデザイナー・イラストレーターと一緒に制作してきました。どのように進めてきたか簡単に紹介したいと思います。

7.2 制作の流れ

スケジュール

紙の本を印刷する場合は印刷所の早期割引の入稿期限を元に、およそ 2 ヶ月（約 60 日）前から活動を始めています。本文の執筆だけでなく、表紙や扉絵などのイラストも準備する必要があります。ここでは、大まかなスケジュール感をお見せしたいと思います。

約 60 日前

本文：執筆者募集、章立てフェーズ
イラスト：イラスト、デザイン担当者募集

約 35 日前

本文：章立て締切、執筆フェーズ
イラスト：アイディア出し、担当決め

約 28 日前

イラスト：ラフ締切、線画着手

約 21 日前

本文：初稿締切、レビューフェーズ
イラスト：着彩着手、デザイン案出し

約 14 日前

本文：校正フェーズ
イラスト：着彩締切、デザイン決定

約 7 日前

本文：タイトル、見出し FIX
イラスト：文字組み、デザイン調整

約 5 日前

本文：最終稿締切、最終調整
イラスト：表紙入稿[*1]

約 1 日前

本文：入稿 PDF 作成、入稿

[*1] 表紙の入稿期限は通常本文より早いため、少し早く入稿することになります。

本文執筆

執筆者集めとして、社内の技術者向けのメーリングリストや Slack などで立候補者を募ります。また、面白そうな技術ネタを持っている人を直接勧誘することもあります。やはり「この前話してたあのネタで書いてみない？」と誘われると、書くモチベーションにもつながるのではないでしょうか。

執筆作業はいくつかのフェーズに分けて進めています。まず執筆者が集まってきたら、章立てフェーズとして各自が何を書くかをとりまとめます。

続いて、執筆フェーズとして原稿を書きはじめます。原稿は Re:VIEW[*2]を使って書き、GitHub Enterprise（GHE）でバージョン管理しています。Markdown に似たテキスト形式はエンジニアにとって馴染みやすく、Git との親和性も高くて重宝しています。また、GHE へプッシュすると自動で PDF を作る CI も用意しています。これについては、Vol. 2 の第 3 章で紹介していますので、合わせてご覧ください。

レビューフェイズでは、執筆者がお互いの記事を読み合い、GHE のプルリクエストの形で修正の提案をしたり議論し、原稿を完成させます。

▲図 7.7　プルリクエストによる修正提案

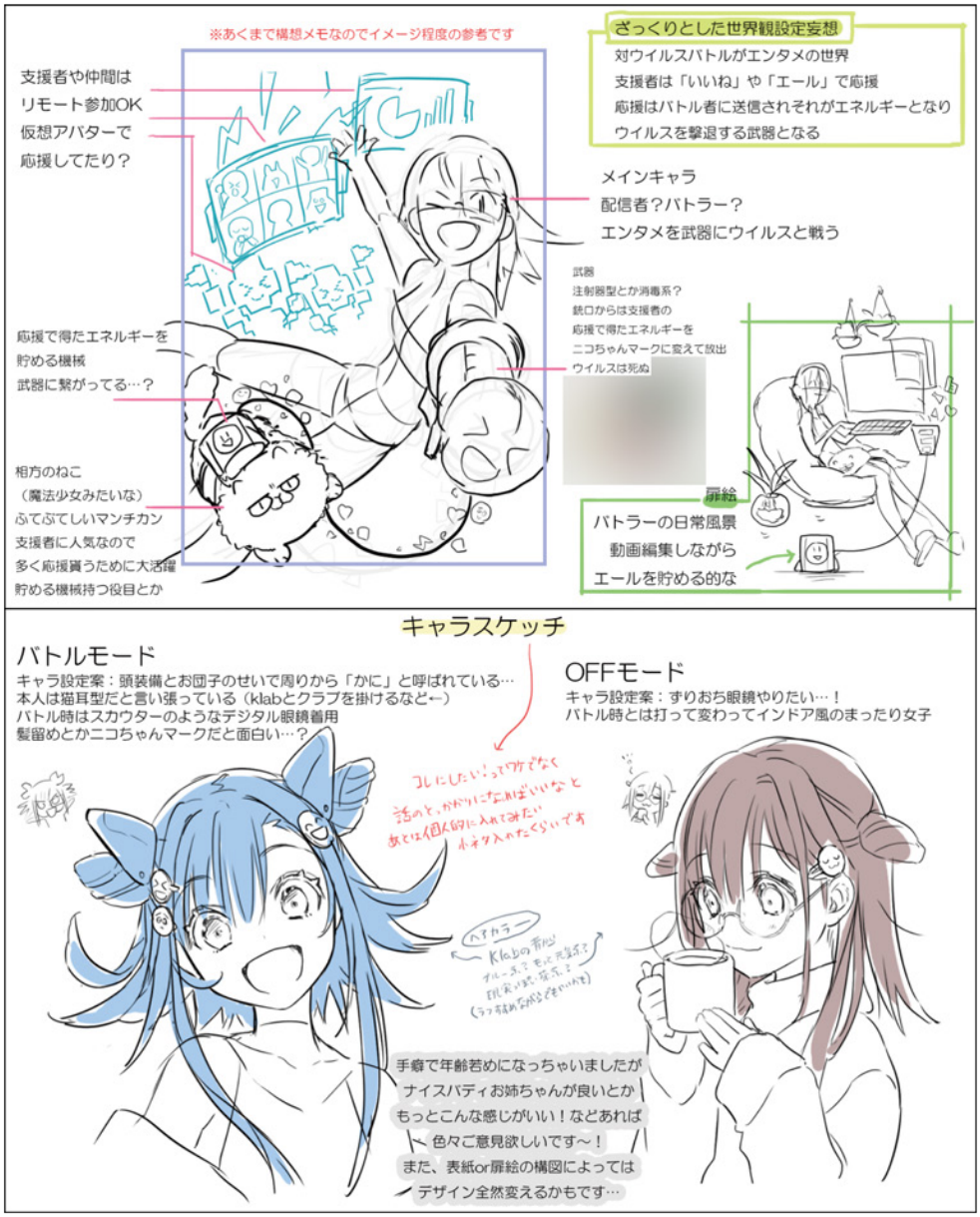
最後の校正フェーズでは最終的な誤字脱字のチェックや掲載順序の調整などを行います。そして改ページ位置や画像サイズなどを微修正し、入稿用の PDF を作成して完成です。

イラスト・デザイン

KLab Tech Book は Vol.2 以降、表紙や扉絵、ダウンロードカードなどを社内のデザイナー・イラストレーターに作成してもらっています。担当者は執筆者と同じく立候補を募る形で募集しています。

[*2] https://reviewml.org/ja/

イラストのテーマやモチーフ、キャラクターの設定やデザインの方向性は、執筆者と同じ Slack チャンネルで皆でアイディアを出し合いながら決めています。ときにエンジニアの妄想を具現化してもらえることもあります。また、製作途中のラフ画像が共有されるたびに Slack がとても盛り上がります。



▲図 7.8　表紙・扉ラフ、世界観、キャラ設定

▲図 7.9　表紙デザインアイディア

本来の業務とは異なる同人誌作成ですが、プロとしての本気の仕事に触れられることで、筆者も良い記事を書こうと日々刺激を受けています。

そんなデザイナー・イラストレーターの取り組みについて、KLab のクリエイティブブログにて紹介していますのでぜひご覧ください。

KLab Tech Book 表紙イラスト制作についての取り組み

`https://www.klab.com/jp/blog/creative/2019/22202758.html`

7.3　まとめ

この章では、これまでに頒布した KLab Tech Book と、おおまかな制作の流れをご紹介しました。

好きなことを書いて発表したいというのがスタートの KLab Tech Book の記事執筆ですが、KLab ではどぶろく制度や情報発信促進制度といったサポート、KLab Tech Book の取り組みが人事評価でのプラスアルファの成果として評価されるといった会社としてのフォローがあります。

しかしながら、そのようなメリットがある中でもひとりのエンジニアとしては、メンバー同士、お互いの記事のレビューをしあうことで技術の知識を深めあえるというところに魅力を感じます。また、KLab Tech Book の制作を通してデザイナー、イラストレーターといったエンジニア以外の職種の方とも交流でき、いろいろな刺激を与えあえます。

本章を参考にして自分たちでもやってみたいと思っていただけたら、また読んで下さったみなさまの技術研鑽、技術発信、交流に役立てていただけたら嬉しいです。

comment

執筆者

第1章　Raspberry Piで作る自作Linuxデバイスドライバー

Ryota Togai @garicchi

英語トレーニング中

第2章　Raspberry PiとGoogle Meetでお手軽ペットカメラ

Daisuke Makiuchi @makki_d

眼鏡っ娘が好きです

第3章　電子工作で重さを量る

Toshifumi Umezawa

今年も梅干しをたくさん漬けました

第4章　Unityエディタ拡張のプログラム設計〜業務で使えるツールの作り方〜

Yuuki Hirai @96yuuki331

最近、YouTubeの猫チャンネルばかり見ています。
アイコン描いてもらいました！ ありがとうございます！

第5章　itertools repeatを使ってみて読んでみる

Shunsuke Ito @fgshun

飽きることなくPythonで遊んでいます

第6章　DiscordとSlackの架け橋

Shinya Naganuma @Pctg_x8

長らくネコ派だったんですが、最近になってイヌ（キツネ / オオカミ）もいいな.....
ってなってきました。

第7章　会社の有志で技術同人誌を書き続けている話

Kinuko MIZUSAWA

みんなでわいわい本作り、さいこうです！

スタッフ

企画進行

きたさき

企画進行にて参加させていただきました。ありがとうございました！

イラスト

もりやす

表紙イラスト担当しました。やってみたい事も盛り込めたので楽しかったです！

イラスト

いわもと

扉絵イラスト担当しました。楽しく描けました！

デザイン

おが

表紙デザイン担当しました。毎回色々な方のイラストでデザインできるので楽しいです！

デザイン

たのりん

デザイン周り担当しました。初参加でしたが次やってみたいこともできたかもです〜。

Special Thanks

木 / ヤマウラ

電子版ダウンロード

https://www.klab.com/jp/blog/tech/2020/tbf10.html

KLab Tech Book Vol. 7

2020年12月26日　技術書典10版(1.0)
著　者 KLab 技術書サークル
編　集 水澤 絹子、牧内 大輔
発行所 KLab 技術書サークル

クラブテックブック KLab Tech Book Vol.7