クラブテックブック

Kab Tech Book






Vol.5

- Starlette-きらめくASGIフレームワーク
- Vulkan メモリマネジメント/ RenderPass 虎の巻
- 2次元コード DataMatrix
- オフィス内のCO_2濃度を測るデバイスを作成する
- 「OK グーグル! 銀行振込1000円」
- GoogleCalendarを扱うSlack Botの製作記


KLab Tech Book
クラブテックブック
Vol.5

KLab Tech Book Vol. 5

2019-09-22 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 5 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 5 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。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、表紙の可愛い眼鏡っ子達も実は第 4 弾のキャラクターの SD 化です。電子版では本文中にも挿絵として掲載されています。表紙も本文も一冊まるごと、読者のみなさまに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。

目次

第 1 章

Starlette - きらめく ASGI フレームワーク

Shunsuke Ito / @fgshun

1.1 はじめに

Starlette は WSGI の後継である ASGI プロトコルにのっとった Python Web アプリケーションのためのフレームワークおよびツールキットです。自身には多くの機能を持たず、機能の不足を補うときには他のライブラリや Starlette 用プラグインに頼るというつくりは Python 製マイクロフレームワーク Flask を連想させます。しかし、プロトコルが WSGI でなく ASGI となっています。またフレームワークを作るためのツールキットでもある点も違いといえるでしょう。Flask は Werkzeug という WSGI ライブラリを用いているのですが、Starlette は ASGI 版の Werkzeug のようなもの、ともいえます。このため他の ASGI フレームワークをインストールした際 Starlette が一緒についてくる、なんてことが見受けられます[*1]。直接つかわなくとも裏で仕事をしているライブラリなのです。そして、Starlette のツールキットとしての機能をもちいて作られたのが Starlette フレームワークです。

ASGI をあつかうライブラリたちの間で小さくともきらめいている、そんな Starlette を紹介します。

[*1] Starlette を用いて作られたフレームワークには Responder, FastAPI, Bocadillo などがあります

1.2 インストール

Starlette の動作には Python 3.6 以上[*2]が必要です。以降の記事の動作確認は Python 3.7.4 を用いています。インストールは リスト 1.1 のように行います。

動作させるために ASGI サーバーも必要なため、今回は Uvicorn サーバーをあわせて用意します。またコマンドラインから Web アプリの動作確認を行うのに HTTP クライアントである httpie を使用しています。もちろん curl などなじみのツールが他にあるのであれば、それを用いても問題ありません。

▼リスト 1.1　pip による Starlette と Uvicorn と httpie のインストール

```
$ pip3 install starlette[full] uvicorn
$ pip3 install httpie
```

starlette で full を指定することで、動作に必須ではないものの一緒に使用することが想定されているライブラリ requests, aiofiles, jinja2, python-multipart, itsdangerous, pyyaml, graphene, ujson がまとめてインストールされます。はじめて Starlette を使って本記事や公式のドキュメントをなぞってみる際には、ライブラリの不足にわずらわされることのないように full を指定して環境構築を行うとよいでしょう。

1.3 Hello Starlette

さいしょのアプリケーション

Starlette を使用した Web アプリケーションがどのようなコードになるかをみていきましょう。

▼リスト 1.2　hello.py

```
from starlette.applications import Starlette
from starlette.responses import PlainTextResponse


app = Starlette()


@app.route('/')
def homepage(request):
    return PlainTextResponse('Hello, Starlette!')
```

[*2] Python 3.8.0b3 でも Starlette 0.12.7 本体が動作することを確認できています。しかし Uvicorn 0.8.6 が依存する uvloop 0.12.2 がまだ Python 3.8 対応を終えていません。 uvloop 0.13 を待ってください。どうしても Python 3.8 で動作させてみたいということであれば `--loop=asyncio` オプションを使用してください

まず、アプリケーションクラス Starlette のインスタンスを作ります。つづいて、リクエストを受けとりレスポンスを返すメソッドを作り、そのメソッドがどの URL に対応するものか route デコレーターで示します。これを Uvicorn サーバーにのせて動かすには リスト 1.3 のようにします。

▼リスト 1.3　Uvicorn サーバーの起動

```
$ uvicorn hello:app  # モジュール名:インスタンス名
```

http://localhost:8000 にブラウザで、もしくはなんらかの HTTP クライアントでアクセスすると Hello Starlette! の文字が確認できます。

▼リスト 1.4　httpie を用いた http://localhost:8000 へのアクセス

```
$ http :8000
HTTP/1.1 200 OK
content-length: 17
content-type: text/plain; charset=utf-8
date: Fri, 09 Aug 2019 07:58:21 GMT
server: uvicorn

Hello, Starlette!
```

並行処理への積極的な対応

Starlette では通常の関数のかわりにコルーチン関数を用いることができます。さっそく リスト 1.2 をコルーチン関数を用いて記述してみます。

▼リスト 1.5　コルーチン関数を用いる

```
@app.route('/')
async def homepage(request):
    return PlainTextResponse('Hello, Starlette!')
```

def が async def になっています。コード上は 5 文字加わっただけですが、 homepage を実行して得られるものが「PlainTextResponse オブジェクト」から「PlainTextResponse オブジェクトを返すコルーチン」に変わっています。この違いを Starlette は正しく認識・吸収するため、アプリを動作させてみると先ほどと変わりなく Hello Starlette の文字を返すことがわかります。

asyncio もしくはその互換の uvloop のイベントループ上で動く Uvicorn のような ASGI サーバーは、 await 文で示された中断可能な箇所を認識します。たとえば aiofiles ライブラリを用いることで、プログラマは組み込み関数 open の使用とたいして変わらないコードでファイル読み出しを記述することができ、サーバーはファイル読み出しにおけ

る I/O 待ちの時間に別の処理を行うことができます。

▼リスト 1.6　aiofiles ライブラリによるファイルの読み出し

```
@app.route('/')
async def homepage(request):
    async with aiofiles.open('hello.py') as f:
        data = await f.read()
    return PlainTextResponse(data)
```

1.4　よくある書き方

ルーティングとリクエストの処理

ここからは Web アプリケーションを書くにあたってよくある処理を Starlette を使って書くにはどのようにするのかを紹介します。まずはルーティングとリクエストの処理です。実装したコルーチン関数がどの URI に対するものであるかを示すのには `app.route` デコレータを用います。これには中括弧で示した箇所から値を取り出す機能があります。`route` デコレータは複数指定することができます。

リクエストを処理するコルーチン関数は `requests.Request` 型の引数をひとつもちます。このオブジェクトがもつ情報を リスト 1.7 に示します。

▼リスト 1.7　Request オブジェクトが保持する情報

```
from starlette.responses import PlainTextResponse

@app.route('/spam', methods=['GET', 'POST'])  # methods でどのメソッドに対応するかを選べる
@app.route('/spam/{ham}')  # 中括弧で URI から値を拾える. request.path_params 辞書に収まる
@app.route('/spam/{ham}/{eggs:int}')  # :int, :float, :path で値のバリデートおよび変換ができる
async def spam(request):
    s = f"""
URL を表す文字列: {request.url}
HTTP リクエストメソッドを表す文字列: {request.method}
パスパラメータを収めた変更不能辞書: {request.path_params.get('ham', 'spamspam')}
リクエストヘッダーを収めた変更不能辞書: {request.headers}
クエリーパラメータを収めた文字列: {request.query_params}
クライアントの hostname と port を収めた NamedTuple: {request.client}
クッキー情報を収めた辞書: {request.cookies}
"""
    return PlainTextResponse(s)
```

また、リクエストボディを得るには `await request.body()` を使います。バイト型のデータが得られます。これが JSON もしくはフォームデータであることを想定するのであれば `json()` や `form()` を使うこともできます。

▼リスト 1.8　POST された JSON データを読む

```
from starlette.responses import JSONResponse

@app.route('/ham', methods=['POST'])
async def ham(request):
```

```
    data = await request.json()
    return JSONResponse(data)
```

レスポンスの生成

レスポンスを生成するためにはリクエストを処理するコルーチン関数から responses.Response クラスのインスタンスを返します。このクラスには media type の初期設定と値の変換処理が追加されている子クラスがいくつか用意されています。いままでにでてきた PlainTextResponse や JSONResponse はその一部でした。まずは素の Response から紹介します。

▼リスト 1.9　Response の使用例

```
from starlette.responses import Response

@app.route('/res')
async def res(request):
    return Response(content='Hello Starlette',
                    status_code=200,
                    headers={'Content-Disposition': 'attachment; filename="hello.txt"'},
                    media_type='text/plain')
```

content には文字列型のほかバイト型を用いることもできます。status_code には任意の整数（初期値 200）を、 headers には追加のレスポンスヘッダー示す辞書を、 media_type には任意の文字列を指定できます。

さらに Response.set_cookie メソッドで Set-Cookie ヘッダを付与することができます。

▼リスト 1.10　Cookie の設定例

```
@app.route('/cookie')
async def cookie(request):
    c = request.cookies.get('starl', '')
    response = Response(content=c, media_type='text/plain')
    # 10 秒の間 starl key に starlette cokkie という値をもたせる
    response.set_cookie('starl', 'starlette cookie', max_age=10)
    return response
```

次に Response の子クラスたちの紹介です。まずは media_type になんらかの初期値を設定しただけのクラスです。

PlainTextResponse は media_type が text/plain になっている以外 Response と違いがありません。その実装は リスト 1.11 のようにシンプルなものです。

▼リスト 1.11　starlette.responses.py より PlainTextResponse

```
class PlainTextResponse(Response):
    media_type = "text/plain"
```

このようなクラスは他にもあり、たとえば HTMLResponse では text/html となっています。

次に紹介するのは JSONResponse です。これは media_type が application/json となっており、さらに json.dump で content を JSON に変換する処理が加わっています。また、 ujson ライブラリをもちいてより高速に動作する UJSONResponse もあります。

次は RedirectResonse です。これは status_code の初期値が 307 に、コンストラクタの第 1 引数がリダイレクト先を示す url に変更されているものです。この url 引数をもとに Location ヘッダが生成されるようになっています。status_code 引数は Response クラス同様に有効なので 301, 302, 303, 308 を指定することも可能です。

▼リスト 1.12　リダイレクトを行う例

```
from starlette.responses import RedirectResponse

@app.route('/redirect')
async def redirect(request):
    return RedirectResponse(url='/')  # 307 Temporary Redirect
    # 302 FOUND を指定する場合 return RedirectResponse(url='/', status_code=302)
```

最後に紹介するのは transfer-encoding: chunked をもちいてデータを小出しにする 2 つのクラスです。StreamingResponse は文字列を返すイテレータもしくは非同期イテレータを対象にとるものです。もうひとつは FileResponse です。こちらはファイル名をとり FileResponse.chunk_size = 4096 バイトずつ送信するものです。Content-Length, Last-Modified, そして ETag の生成を行ってくれる機能も追加されています。なお FileResponse の動作には aiofiles ライブラリが必要です。

静的ファイルの配信

Starlette には、あるディレクトリ下にあるファイルをそのまま返す ASGI アプリケーションを生成する StaticFiles クラスが用意されています。そして、 Starlette クラスには他の ASGI アプリケーションを載せるための mount メソッドがあります。これを組み合わせることで、アプリケーションから静的ファイルを配信できます。

▼リスト 1.13　静的ファイルを配信する

```
from starlette.staticfiles import StaticFiles

app.mount('/static', StaticFiles(directory='static'))
```

テンプレートエンジン

Starlette 自身にはテンプレートエンジンとしての機能はありませんが、 Jinja2 と連携するための `Jinja2Templates` クラスが用意されています。このクラスにより Jinja2 側に URL の逆引きを行うための `url_for` 関数が提供されます。静的ファイルの処理と合わせてその配下にあるファイルへのリンクを得るには リスト 1.14 のようにします。

▼リスト 1.14　Jinja2 テンプレートエンジンとの連携

```
from starlette.staticfiles import StaticFiles
from starlette.templating import Jinja2Templates

app.mount('/static', StaticFiles(directory='static'), name='static')

templates = Jinja2Templates(directory='templates')

@app.route('/jinja2')
async def use_jinja2(request):
    # template context に request インスタンスをわたす必要がある
    return templates.TemplateResponse('spam.html', {'request': request})
```

▼リスト 1.15　Jinja2 テンプレートの例 templates/spam.html

```
<!doctype html>
<html>
<head><meta charset='UTF-8'></head>
<body><p><a href="{{ url_for('static', path='/spam.txt') }}">spam.txt</a></p></body>
</html>
```

アプリケーション初期化時・終了時の処理

Starlette アプリケーションには初期化時および終了時に何らかの処理をおこなわせるための `on_event` デコレータがあります。初期化時には startup を、終了時には shutdown を引数に指定します。

例として RDBMS をあつかうライブラリ `databases` の操作をあげてみます。Starlette 自身には RDBMS の操作のための機能は含まれていないので、 なんらかのライブラリを利用することになります。そのようなライブラリの初期化処理に `on_event` は適しています。

▼リスト 1.16　on_event の使用例 - databases ライブラリのコネクションプールの準備

```
import databases
from starlette.applications import Starlette

DATABASE_URL = 'sqlite:///spam.sqlite3'

database = databases.Database(DATABASE_URL)
app = Starlette()
```

```
@app.on_event('startup')
async def startup():
    await database.connect()

@app.on_event('shutdown')
async def shutdown():
    await database.disconnect()
```

設定ファイルの分離

リスト 1.16 では SQLite3 データベースファイルのパスが直接コードに書き込まれてしまっています。Starlette ではこのような設定を別ファイルに分離するために `Config` クラスが用意されています。設定ファイルから読み出した文字列を別の型にするための `cast` 引数と、項目がないときに返す値を設定する `default` 引数をもちます。

▼リスト 1.17　Config クラスの使用

```
from starlette.config import Config

config = Config('.env')
DEBUG = config('DEBUG', cast=bool, default=False)
DATABASE_URL = config('DATABASE_URL', default='sqlite3:///default.sqlite3')
```

▼リスト 1.18　コンフィグファイルの例 .env

```
DEBUG = True
DATABASE_URL = sqlite:///spam.sqlite3
```

ユニットテスト

Starlette にはユニットテストにつかうことができる requests ライブラリのラッパー `TestClient` クラスが用意されています。使用方法は requests と同じです。たとえば リスト 1.2 に対するテストの例は リスト 1.19 のようになります。

▼リスト 1.19　hello.py に対するテストの例

```
from starlette.testclient import TestClient

from hello import app

def test_homepage():
    client = TestClient(app)
    response = client.get('/')
    assert response.status_code == 200
    assert response.text == 'Hello, Starlette!'
```

バックグラウンドタスク

Starlette には BackgroundTasks クラスが用意されています。このクラスでは、レスポンスを返したあとで実行するタスクを指定できます。タスクはコルーチン関数として実装し BackgroundTasks.add_task でひもづけ Response コンストラクタの background 引数にわたすことで登録できます。登録されたタスクはレスポンスを正常に返し終えた後に順次実行されます。たとえば確認メールの送信など、レスポンスをクライアントに返した後に時間をかけて行ってもよい処理に適しています。

▼リスト 1.20　BackgroundTasks

```
from starlette.applications import Starlette
from starlette.background import BackgroundTasks
from starlette.responses import JSONResponse
app = Starlette()

@app.route('/signup', methods=['POST'])
async def signup(request):
    data = await request.json()
    res = await create_user(data)
    tasks = BackgroundTasks()
    tasks.add_task(logging,  data)
    tasks.add_task(send_mail, data)
    return JSONResponse(res, background=tasks)

async def logging(data):
    ...
```

1.5　WebSockets

Starlette が採用する ASGI にできて WSGI には難しいことに HTTP 以外のプロトコルへの対応があげられます。WSGI はステートレスの HTTP をあつかうことに特化しすぎていましたが、ASGI はそういった問題を解決するためにつくられました。そのため HTTP 以外をあつかうサーバーアプリケーションにも用いることができます。たとえば WebSocket です。

Starlette には WebSocket を利用して双方向通信を行うアプリケーションのための WebSocket クラスが用意されています。これを使用するには route デコレータの代わりに websocket_route デコレータを使います。するとエンドポイントの引数が Response ではなく WebSocket にかわります。

WebSocket を利用する例を リスト 1.21 に示します。await accept() でハンドシェイクをおこない接続を確立し、 await send_text() で送信を、 await receive_text() で受信を行います。接続を終了するには await close() を呼びます。send_text/receive_text には 2 つの変種があります。バイト型のままの送受信をする send_bytes/receive_bytes と、JSON へのエンコード・デコードをはさむ send_json/receive_json です。

▼リスト 1.21　WebSocket を利用して双方向通信を行うアプリの例

```
from starlette.applications import Starlette

app = Starlette()

@app.websocket_route('/')
async def homepage(websocket):
    await websocket.accept()
    name = await websocket.receive_text()
    await websocket.send_text(f'hello {name}')
    await websocket.close()
```

このアプリの動作確認を Python でおこなうには websockets ライブラリを使うのがよいでしょう。

▼リスト 1.22　websockets ライブラリの簡易クライアントをつかった動作確認

```
$ pip3 install websockets
Installing collected packages: websockets
Successfully installed websockets-8.0.2
$ python3 -m websockets 'ws://localhost:8000'
Connected to ws://localhost:8000.
> fgshun
< hello fgshun
Connection closed: code = 1000 (OK), no reason.
```

1.6　GraphQL

Starlette には GraphQL のためのライブラリ graphene へのサポートもあります。`GraphQLApp` クラスを使えば graphene で作成したスキーマをもとに ASGI アプリケーションを生成することができるのです。これで生成したアプリケーションを `Starlette` クラスの `mount` メソッドと組み合わせて使います。

▼リスト 1.23　GraphQL をつかう

```
from starlette.applications import Starlette
from starlette.graphql import GraphQLApp
import graphene

class Query(graphene.ObjectType):
    progress = graphene.String(q=graphene.String())

    def resolve_progress(self, info, q):
        return '進捗ダメです' if '進捗' in q else ''

app = Starlette()
app.mount('/', GraphQLApp(schema=graphene.Schema(query=Query)))
```

▼リスト 1.24　GraphQL 動作確認

```
$ http post :8000 query='{ q0: progress(q: "進捗はどうですか")
q1: progress(q: "進捗どうですか")
q2: progress(q: "進捗……")}'
HTTP/1.1 200 OK
content-length: 102
content-type: application/json
date: Sun, 18 Aug 2019 13:20:17 GMT
server: uvicorn

{
    "data": {
        "q0": "進捗ダメです",
        "q1": "進捗ダメです",
        "q2": "進捗ダメです"
    },
    "errors": null
}
```

ブラウザでアクセスすると GraphiQL が立ち上がるので開発中の動作確認に便利です。運用中など GraphiQL が不要であるのであれば `GraphQLApp(graphiql=False)` を指定することで止めることができます。

1.7 おわりに

Web フレームワークとしての Starlette について紹介しました。小さくまとまっていてくせがないフレームワークではないでしょうか。あらたに Python で Web アプリケーションを作ってみる際の選択肢のひとつとしていかがでしょうか。

第 2 章

Vulkan メモリマネジメント / RenderPass 虎の巻

Shinya Naganuma / @Pctg_x8

Vulkan は 2016 年にはじめて発表されて以降、Android が正式な対応を押し進めたり iOS などでも変換レイヤーが用意されるなど、幅広いプラットフォームで動く API へと着々と成長を続けています。

そんな Vulkan ですが、DirectX12 をはるかにしのぐほどの低レイヤー API であり、重複して提供する必要のあるパラメータが多かったりメモリコンテンツの配置をある程度自前で制御する必要があるなど、API の扱いはとても高度な知識を要するものとなっています。

本章では Vulkan が提供する API の中でも、必須であるがとりわけ難しいとされる「メモリ/バッファ」および「RenderPass」の管理戦略について解説していきます。

2.1 Vulkan におけるメモリマネジメント戦略

Vulkan では、Uniform Buffer などが使用するメモリ領域を確保するためには大きく分けて 2 つの手順を踏みます。

1. 具体的なオブジェクト (`Buffer`/`Image`) を作成する
2. デバイス上のメモリ (`DeviceMemory`) を確保し、オフセットを指定して上記オブジェクトをバインドする

つまり、`Buffer`や `Image`を作成しただけでは実際のメモリ領域が存在しないため描画などで使用することができません。

このように具体的なオブジェクトとメモリ確保を別にすることで、オブジェクトをコマンドバッファにバインドしたままメモリのスワップアウトや再利用、必要時にのみ割り当

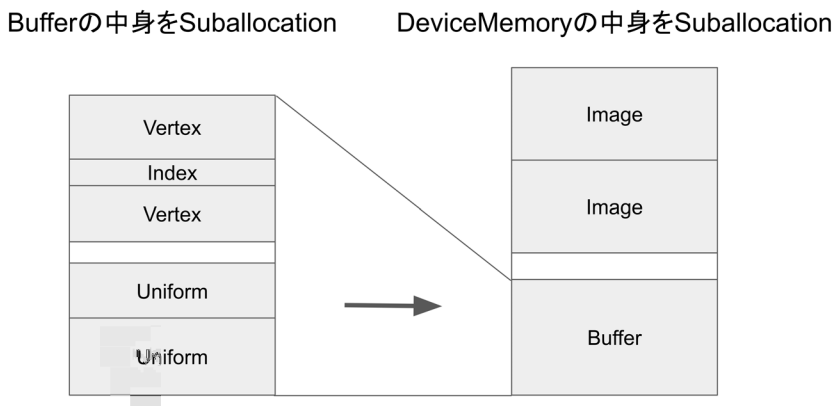
てをするなど柔軟なメモリ利用管理を行うことが可能になっています[*1]。

ミニマムな実装としては、ひとつの DeviceMemoryにつきひとつの Buffer/Imageをバインドする形で既存の Uniform Buffer などを作成することが可能です。しかしその場合、大量の DeviceMemoryが確保、作成されることとなり、開放処理が多くなることによるオーバーヘッドの増加、管理変数が増えることによるコードの煩雑化、実装方法によってはフラグメンテーションが発生するようになるなど、一部[*2]では推奨されない実装とされています。

ではどういった形式が推奨されているか、それは「ひとつの DeviceMemory上で、必要最低限の Buffer/Imageを作成して、その中でサブアロケーションを行う」です。

固定長の場合

ゲームアプリケーション中、あるいは単一のシーン中において固定長、かつ永続的に存在するメモリリソースであれば話は簡単です。この場合、前述の推奨される管理方法を愚直に実装すれば OK となります。「DeviceMemoryに対して Buffer/Imageをバインドする」とは、広く捉えれば DeviceMemoryの中のサブアロケーション処理と言い換えることができるため、サブアロケーションを 2 段階に渡って行うことで最大効率でメモリリソースを作成することができます。



▲図 2.1　サブアロケーションを 2 回行う

ひとつ注意していただきたいのが、Uniform Buffer などの配置オフセットにはかなり大きめなアラインメントが要求されていることが多く、サブアロケーションの順序によっては大きく損をする作りになります。たとえば筆者が使用している NVIDIA GeForce GTX 960 では、それぞれ次のアラインメントが要求されています。

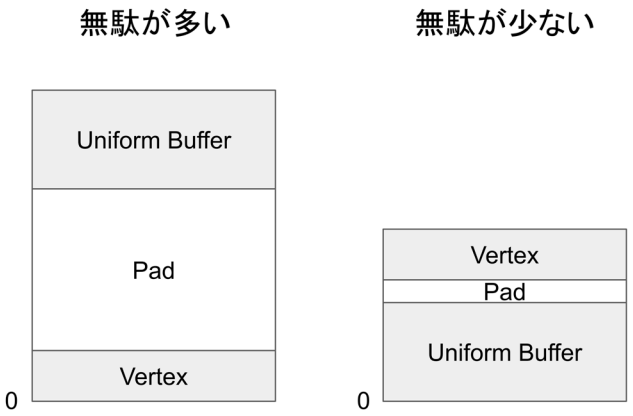
[*1] この機能を使うためには、Image の生成時に特定のフラグを指定する必要があります
[*2] https://developer.nvidia.com/vulkan-memory-management

- Uniform Buffer は 256byte
- Storage Buffer は 32byte

上記に加えて、使用するデータ型によるアラインメントも加わります。たとえば、多くの頂点では vec4型を使用しますが、これは 16byte のアラインメントを要求します。アラインメント制約に従わないと、最悪の場合詳細不明の Device Lost を引き起こします。アラインメント制約違反は実行時になってはじめて検知できる項目なので、vkEndCommandBufferなどの API では検知できず、遅れてやってくる厄介なエラーになります[*3]。

Imageにかかるアラインメント制約は上記のものに比べてもう少し厄介です。Vulkan のイメージデータは、さまざまな要因により必ずしもピクセルごとのデータが左上から直列に並んだものであるとは限りません。このようなレイアウトを NonLinearと呼びます。一方でバッファのデータは一般的に CPU 上でのメモリイメージと完全に一致します。つまり 0 バイトから増える方向に直列にデータが並んでいます。こちらのレイアウトは Linearと呼びます。Vulkan をサポートするデバイスでは、メモリの用途以外にもうひとつ Linearなデータ領域と NonLinearなデータ領域との間にのみ存在するアラインメント制約が存在します。こちらのアラインメント制約は、デバッグレイヤーを導入していればバインド時に検査されるため、万が一忘れても比較的容易に検出することが可能です。

では、どのようにサブアロケーションするのがよいでしょうか。一般的には、アラインメント要求サイズが大きいものを先頭 (オフセット 0 側) に配置するのがよいでしょう。また、Bufferの後ろに Imageを配置するルールにしておけば、上記の Linearと NonLinearの間にかかるアラインメント制約についても、最初の Imageにのみ適用すればよいので、実装が簡単になると思います。



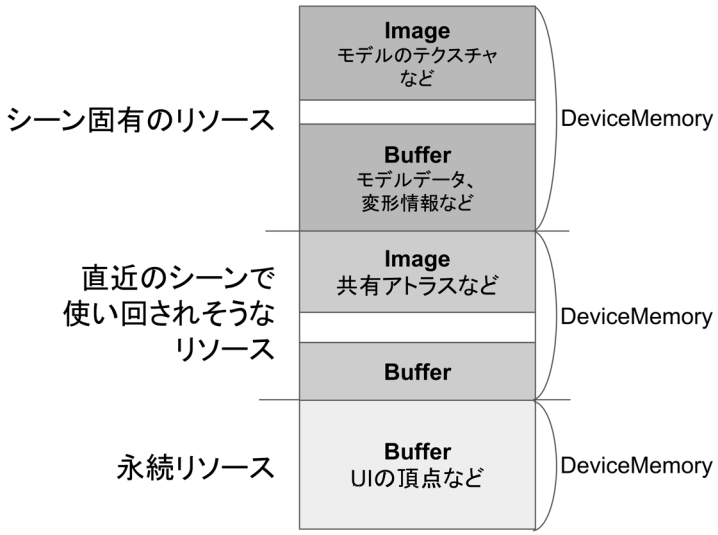
▲図 2.2　省パディング

[*3] 筆者は一度頂点バッファのアラインメントを無視して Device Lost を引き起こした結果 1〜2 日ほど時間を潰したことがあります

可変長にしたい場合

パフォーマンス状の観点からは、固定長のメモリを確保して、その上でスタックやリングバッファなど可変長な構造を再現するのが望ましいです。ただ、シーンチェンジなど大規模にメモリの利用方法が変わる場面では、旧メモリを解放したのち新たにサブアロケーションを行うことで必要最低限のメモリのみを使用した運用が行えると思います。

その場合に限っていえば、ゲーム全体を通して使い回されるようなデータ (UI の頂点など) をいちいち生成し直すのは不要なコストがかかるでしょうから `DeviceMemory`を分割してしまうほうがよいでしょう。



▲図 2.3　DeviceMemory レベル分け 概念図

2.2　RenderPass 概略

Vulkan によるレンダリングを行う上で避けて通れないオブジェクトが「RenderPass」です。重要なわりに非常に謎に包まれた難解な存在とされるオブジェクトですが、今まで実用してきた結論をいうと、どの場合でもほとんど同じ実装になりますので、パイプラインなどと違って実はあまり悩む必要がないオブジェクトです。

RenderPass とは何か

RenderPass とは、詳細は省きますが[*4]レンダリングパイプラインの効率化を補助するための機能です。モバイル GPU ではもはや定番に近い実装手法になった「Tile-Based Deferred Rendering」の機構を効率よく動かすことができます。デスクトップ GPU でも、NVIDIA GeForce シリーズでは Maxwell アーキテクチャからは同様の機構を採用しており、RenderPass をうまく使用することでモバイル GPU と同様の効果を見込むことができます。

レンダリングプロセスとプロセス間依存を合わせて有向グラフとして構築することで、「Tile-Based Deferred Rendering を最大効率で回す」「VRAM の帯域消費を最小限に抑える」など処理効率向上ために、処理順序などをドライバが適切にならべかえる際のヒントとなる情報を提供します。

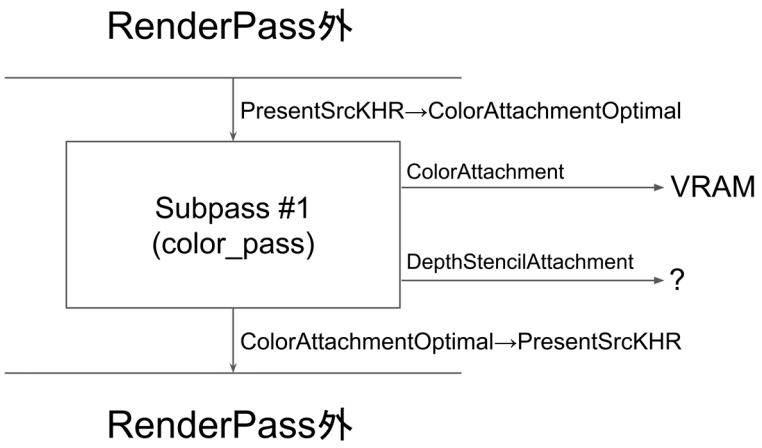
この RenderPass にはひとつ厳しい制限が存在しています。「Attachment として使用した ImageView は同一 RenderPass 内で Attachment 以外で使用できない」というものです。具体的には、直前のパスで描いたイメージを後続のパスでぼかすといったことができません。たとえば、あるイメージオブジェクトを、一方のパスで ColorAttachment として出力、他方のパスで InputAttachment として入力するようにした場合、この 2 つのパスを連続して実行すれば最小 1px ぶんのメモリ領域で全体の作業を完了できます[*5]。このような厳しい制約が存在することで、ドライバが上記のような最適化を行える余地が生まれ、VRAM の帯域消費量が改善されたり処理速度が向上したりする可能性が高まります。また、RenderPass オブジェクトが Attachment を完全に管理するように強制することで、暗黙的なパイプラインバリア (イメージレイアウト変更) を効率よく発行できるといった利点も生まれます。

2.3 RenderPass をどう管理するか？

前述の制限があるため、RenderPass はほとんどの場合で「色データと深度データを出力するのみ」となります。図にすると図 2.4 のように、コードにするとリスト 2.1 のようになります。

[*4] ref: `https://qiita.com/Pctg-x8/items/2b3d5c8a861f42aa533f`

[*5] InputAttachment には、フラグメントシェーダがキックされた時に対応するピクセル以外を参照することができない制限があります



▲図 2.4　シンプルな RenderPass

▼リスト 2.1　シンプルな RenderPass の設定

```
VkAttachmentDescription attachments[2];
VkAttachmentReference aref_color, aref_depth;
VkSubpassDescription color_pass;
VkSubpassDependency into_color_deps;

/* 共通設定 */
attachments[0].flags = attachments[1].flags = 0;
attachments[0].samples = attachments[1].samples = VK_SAMPLE_COUNT_1_BIT;
// ステンシルは使わないので下四つは無視される
attachments[0].stencilLoadOp = VK_LOAD_OP_DONT_CARE;
attachments[1].stencilLoadOp = VK_LOAD_OP_DONT_CARE;
attachments[0].stencilStoreOp = VK_STORE_OP_DONT_CARE;
attachments[1].stencilStoreOp = VK_STORE_OP_DONT_CARE;
/* アタッチメントごとの設定 */
attachments[0].format = VK_FORMAT_R8G8B8A8_UNORM;
attachments[0].initialLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
attachments[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
attachments[0].loadOp = VK_LOAD_OP_CLEAR;
attachments[0].storeOp = VK_STORE_OP_STORE;
attachments[1].format = VK_FORMAT_D32_SFLOAT;
attachments[1].initialLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
attachments[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
attachments[1].loadOp = VK_LOAD_OP_CLEAR;
attachments[1].storeOp = VK_STORE_OP_DONT_CARE;

aref_color.attachment = 0;
aref_color.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
aref_depth.attachment = 1;
aref_depth.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

color_pass.flags = 0;
color_pass.inputAttachmentCount = 0;
color_pass.pInputAttachments = nullptr;
color_pass.colorAttachmentCount = 1;
color_pass.pColorAttachments = &aref_color;
color_pass.pResolveAttachments = nullptr;
color_pass.pDepthStencilAttachment = &aref_depth;
color_pass.preserveAttachmentCount = 0;
color_pass.pPreserveAttachments = nullptr;
```

```
into_color_deps.srcSubpass = VK_SUBPASS_EXTERNAL;
into_color_deps.dstSubpass = 0;
into_color_deps.srcStageMask = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
into_color_deps.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
into_color_deps.srcAccessMask = VK_ACCESS_MEMORY_READ_BIT;
into_color_deps.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
into_color_deps.dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

VkRenderPassCreateInfo create_info;
create_info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
create_info.pNext = nullptr;
create_info.flags = 0;
create_info.attachmentCount = 2;
create_info.pAttachments = attachments;
create_info.subpassCount = 1;
create_info.pSubpasses = &color_pass;
create_info.dependencyCount = 1;
create_info.pDependencies = &into_color_deps;
```

color_passと into_color_depsについてはこれでほぼ固定で、アプリケーション固有で気にすべき箇所は VkAttachmentDescriptionの各種設定になります。

たとえば、リスト 2.1 はスワップチェーンのバックバッファに描画するための設定になっています。これをオフスクリーンレンダリング用にし、後続でポストエフェクトを掛ける予定があるのであれば attachments[0].initialLayoutと attachments[0].finalLayoutを VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL(シェーダ読み取り専用)にするとよいでしょう[*6]。パスの終了時に自動で finalLayoutで指定したレイアウトに変更されますので、あらかじめ指定しておけば追加でパイプラインバリアを張る必要がなくなります。

Deferred Lighting などで後続のパスで深度バッファの情報を読み取りたい場合は、attachments[1].storeOpを VK_STORE_OP_STOREにする必要があります。VK_STORE_OP_DONT_CAREでは VRAM へのライトバックを強制しないため、ドライバが最適な実装として「パス終了時にデータを捨てる」選択をする場合があります。

2.4 終わりに

本章では、Vulkan におけるメモリマネジメント戦略の一例と RenderPass についてテンプレート設定を紹介しました。ここで紹介した以外にもさまざまなテクニックがあり、うまく活用すれば本当に OpenGL の数倍といったパフォーマンスを引き出せるだけのポテンシャルを秘めた API になっているので、ぜひ Vulkan を完全に理解して高効率なアプリケーションを作成していきましょう。

[*6] この場合、厳密には into__color__deps.srcStageMask と srcAccessMask も正しいものに変える必要があります

第 3 章

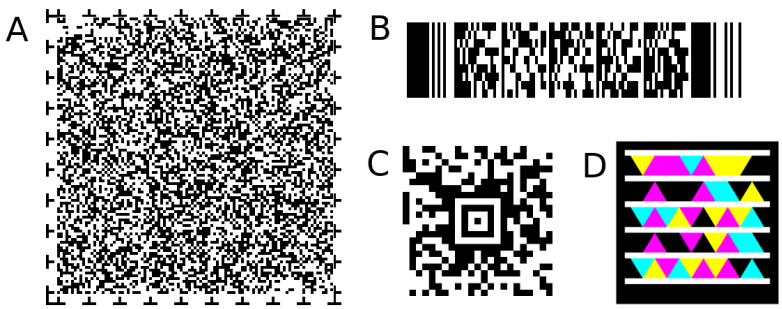
2 次元コード DataMatrix

Daisuke Makiuchi / @makki_d

3.1　2 次元コードの世界

QR コードの登場から 25 年、2 次元コードはもはや私達の生活に不可欠なほどよく使われるようになりました。ところで、バーコードにさまざまな規格があるように、2 次元コードにも数々の規格があることをご存知でしょうか。

身近なところでは地方自治体からの通知に **SP コード**や **Uni-Voice** という視覚障害者向けの音声コードが印字されているのを見たことがあると思います。また、航空券に **PDF417** が印字されているのをたまに見かけますし、他にも **Aztec** や **MaxiCode**、変わったところでは **HCCB** や**カメレオンコード**といったカラフルなものもあります。図 3.1 にいくつか挙げてみました[*1]。



▲図 3.1　さまざまな 2 次元コード. A: Uni-Voice; B: PDF-417; C: Aztec; D: HCCB

[*1] 画像出典 A: https://www.javis.jp/ B: https://en.wikipedia.org/wiki/PDF417 C: https://en.wikipedia.org/wiki/Aztec_Code D: https://en.wikipedia.org/wiki/High_Capacity_Color_Barcode

この章では、電子機器や医療機器産業の分野でよく使われている **DataMatrix** について、QR コードと比較しながら紹介したいと思います。

▲図 3.2　ノート PC に貼られている DataMatrix

3.2　DataMatrix とは

DataMatrix は、QR コードと同じく白黒の四角（セル）を並べてテキストデータや数値データを表現する 2 次元コードで、QR コードの登場より約 7 年早い 1987 年に ID Matrix 社により開発されました。

その後、歪み補正やエラー訂正機能を強化した ECC 200 というバージョンが策定され、このバージョンが広く使われています。QR コードが ISO 規格（ISO/IEC 18004）となった 2000 年に、DataMatrix も ISO 規格（ISO/IEC 16022）となりました。この ISO 規格をもとに JIS 規格（JIS X 0512）にもなっています。

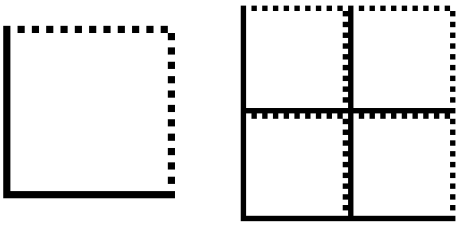
DataMatrix は最小 10 × 10 セルから最大 144 × 144 セルで、一般的な英文（アルファベットの大文字小文字と数字、いくつかの記号）であれば最大 2,335 文字を格納できます。また、正方形だけでなく長方形のパターンも用意されており、細長い領域への印字もしやすくなっています。

▲図 3.3　長方形の DataMatrix

それでは、DataMatrix の特徴について QR コードと比較しながら見ていきましょう。

3.3 検出パターン

QR コードには特徴的な 3 つの検出パターンがあり、これを目印として QR コードを検出します。一方、DataMatrix の検出パターンは左下の L 字の直線部分のみです。上と右にはタイミングパターンとして白と黒のセルが交互に配置されています。大きなサイズの DataMatrix では、データ領域が分割されて直線と白黒交互のセルからなる位置合わせパターンが挿入されます。



▲図 3.4　DataMatrix の固定パターン. 右は位置合わせパターンのあるもの

DataMatrix の固定パターンはこれだけで、内側はすべてデータ領域となっています。QR コードに比べて固定パターンの占める割合が少ないため、より小さいサイズに多くの情報を格納できるようになっています。一方で、L 字のパターンは QR コードのように判別しやすいパターンではないため、検出精度は劣ります。

3.4 エラー訂正符号

DataMatrix においても QR コードと同じく、エラー訂正符号として**リード・ソロモン符号**が使われています。

リード・ソロモン符号は一定数ビットを 1 ワードとして、ワード単位でのエラー訂正を行う符号です。DataMatrix と QR コードはどちらも 8 ビットを 1 ワードとしています。この符号では 1 ワードのうち何ビットのエラーがあったとしても 1 ワードのエラーとして扱われるため、連続して起こるバーストエラーに強いという特徴があります。これは、印刷したときに一部がかすれたり剥がれたりというエラーに対して強いため、2 次元コードにはとても向いている性質です。

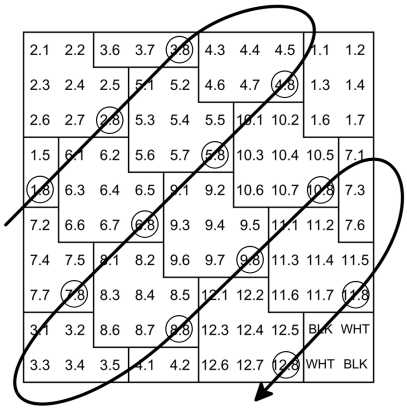
QR コードの場合はエラー訂正レベルを L、M、Q、H の 4 段階から選択することができますが、DataMatrix ではサイズによって自動的に決まり、10 × 10 でもっとも高く約 66%、もっとも低くなる 132 × 132 でも約 19% のエラーを訂正することができます。QR コードにおいては L での約 7% から H の約 30% なので、同等以上の十分なエラー訂正能力があるといえます。

3.5 データの配置方法

リード・ソロモン符号のバーストエラーに強いという性質を最大限に活かすためには、1 ワードができるだけまとまった位置にあるのが望ましいです。

DataMatrix では 3 × 3 の正方形から右上を切り欠いた、いわゆるユタ州型として 1 ワードをまとめます。一方 QR コードでは基本的に 2 × 4 の縦長に 1 ワードをまとめるため、DataMatrix の方がエラー耐性がわずかに高いと考えられます。

しかし、ユタ州型を隙間なく並べるには工夫が必要で、DataMatrix では右上の切り欠きに隣の文字の左下をはめ込み、図 3.5 のように斜めに往復するように並べていきます。そしてデータ領域からはみ出る部分は、反対側の辺のちょうどはまり込む空き領域に配置します。この配置の決定方法は ISO や JIS に図と C 言語のコードで掲載されています。



▲図 3.5　ワードの配置順. 右下のセルが斜めの線上に並ぶように配置する

3.6 エンコード方式

QR コードにはデータを 1 バイトずつそのまま格納する 8 ビットバイトモードの他に、英数字モードや漢字モードといった文字種を制限して効率よく格納するモードがあります[*2]。DataMatrix でも同様に、1 バイトをそのまま格納する **Base256** モードの他、**ASCII**、**C40**、**テキスト**、**X12**、**EDIFACT** というモードがあり、最適なモードに切り替えながらエンコードしていきます。

ここで、"http://klabgames.tech.blog.jp.klab.com/"をエンコードする例をリスト 3.1 に示します。

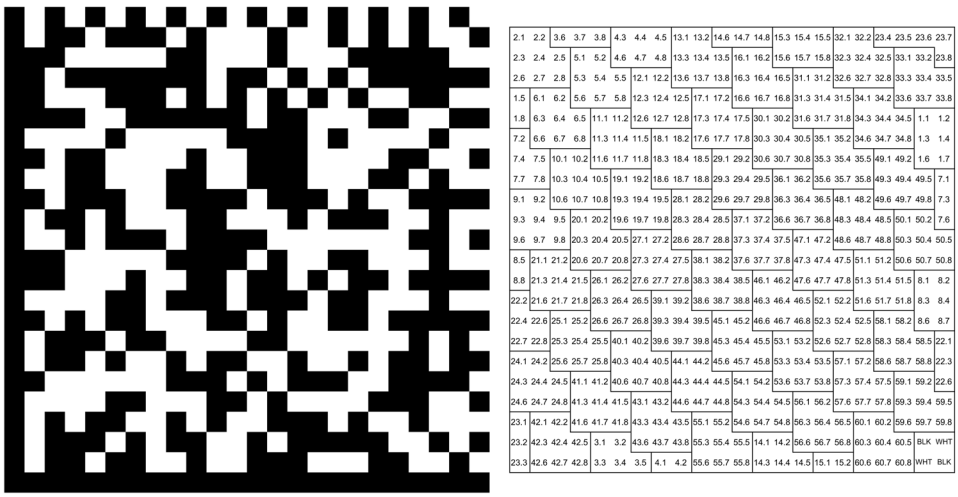
[*2] KLabTechBook Vol.4「第 3 章 QR コードマニアックス—数字・英数字・漢字モード」参照

▼リスト 3.1　URL をエンコード

```
(ASCII モードで開始)
01101001            h=105
01110101            t=117
01110101            t=117
01110001            p=113
00111011            :=59
00110000            /=48
00110000            /=48
11101111            (テキストモードへ移行)
10011001 11110111   k=24 l=25 a=14;     (24*1600)+(25*40)+14+1 = 153*256 + 247
01100000 11101111   b=15 g=20 a=14;     (15*1600)+(20*40)+14+1 =  96*256 + 239
10100101 01110001   m=26 e=18 s=32;     (26*1600)+(18*40)+32+1 = 165*256 + 113
00001000 01101010   .=(shift2),13 t=33; ( 1*1600)+(13*40)+33+1 =   8*256 + 106
01110011 00010110   e=18 c=16 h=21;     (18*1600)+(16*40)+21+1 = 115*256 +  22
00001000 01011000   .=(shift2),13 b=15; ( 1*1600)+(13*40)+15+1 =   8*256 +  88
10100000 10110101   l=25 o=28 g=20;     (25*1600)+(28*40)+20+1 = 160*256 + 181
00001000 01100000   .=(shift2),13 j=23; ( 1*1600)+(13*40)+23+1 =   8*256 +  96
10110101 01110110   p=29 .=(shift2),13; (29*1600)+( 1*40)+13+1 = 181*256 + 118
10011001 11110111   k=24 l=25 a=14;     (24*1600)+(25*40)+14+1 = 153*256 + 247
01011101 11110110   b=15 .=(shift2),13; (15*1600)+( 1*40)+13+1 =  93*256 + 246
01101000 01111011   c=16 o=28 m=26;     (16*1600)+(28*40)+26+1 = 104*256 + 123
11111110            (ASCII モードへ戻る)
00110000            /=48
```

最初の 7 文字が ASCII モードのままなのは、テキストモードでは記号 1 文字あたり 10.67 ビット必要なので、ASCII モードのままの方が短くなるためです。また最後の"/"はテキストモードの 3 文字区切りに合わないため、ASCII モードに戻って符号化されます。

QR コードではマスクを掛けてデータを配置しますが、DataMatrix ではこのビット列をそのままデータ領域に配置します。そのため、図 3.6 のように対応する配置図が脳内にあれば[*3]、人力でも簡単に読み取ることができます。



▲図 3.6　URL を格納した DataMatrix と各ビットの配置

[*3] 「3.5 データの配置方法」と、左下端・右上端の例外的な配置だけ覚えていれば簡単に構築できます。

3.7　なぜ QR コードのほうが普及しているのか

QR コード自体が日本生まれということもあり、日本国内においては QR コードのほうが普及しやすかったのは当然でしょう。海外でも特に中国において、決済サービスなどで QR コードが広く使われていることはみなさんも聞いたことがあると思います。一方、アメリカでは DataMatrix が使われることが多いという話もありますが、最近では Twitter や Facebook などがアカウント共有に QR コードを採用しているように、どうやら QR コードが主流となっているように思われます。

本章で紹介してきたように、データの表現力やエラー訂正レベルの点では DataMatrix も QR コードも決定的な差はなく、DataMatrix が優っているのはデータの集積度、つまり小さいサイズでより多くの情報を表示することができるという点です。しかし、現在二次元コードがもっともよく使われているのは、URL のような短いテキストを共有するという用途で、データ集積度を気にするほどのデータの大きさではないのではないでしょうか。

一方 QR コードが大きく勝っているのは、判別しやすい検出パターンを採用したことによる検出精度の高さでしょう。モバイル端末で手軽に素早く読み取るためにも、検出精度の高さが重要視されているのではないでしょうか。

3.8　まとめ

本章では DataMatrix という二次元コードを QR コードと比較しながら紹介しました。普段あまり見かけることのない DataMatrix ですが、基本的には QR コードと似たようなものです。見かけたときはぜひ DataMatrix 対応のリーダーで読み取ってみてください。

第 4 章

オフィス内の CO_2 濃度を測るデバイスを作成する

黒井春人 / @halt

4.1 はじめに

会社の成長などでオフィス内の人口が増えてくると、空気中の二酸化炭素（CO_2）濃度が恒常的に増える事になります。CO_2 濃度が高い環境では頭がボーッとするなど業務効率への影響があるため、定期的な換気などが求められます。皆さんが所属している会社のオフィスでは CO_2 濃度を測定しているでしょうか？ この章では CO_2 濃度を測定するデバイスの作り方を紹介します。

4.2 CO_2 濃度が人体に与える影響について

CO_2 濃度は外気では 400ppm を少し上回る程度ですが、1500ppm を超えると判断力や集中力が低下しはじめ、2500ppm 以上で吐き気や頭痛、めまいなどの影響が出て来るそうです。厚生労働省の建築物環境衛生管理基準[*1]によると、1000ppm 以下になるように空調・換気設備の維持管理をする必要があると書かれています。

4.3 CO_2 濃度を測定する最も簡単な方法

単に CO_2 濃度を取得したいだけであれば、CO_2 濃度を測ってくれるデバイスを購入するのが一番簡単です。私は自宅の CO_2 濃度確認のために Netatmo というデバイスを数年前に購入し運用しています。Netatmo は計測結果を自動的にクラウドにアップロードするデバイスで、PC やスマホからいつでも計測結果を見ることができます。専用アプリ

[*1] 建築物環境衛生管理基準について https://www.mhlw.go.jp/bunya/kenkou/seikatsu-eisei10/

のインストールで、CO_2 濃度が高くなったときに通知がくるようにする事もできます。

▲図 4.1　この円柱を家において Web にアクセスすると室内環境に関する情報を得る事ができます

Netatmo を購入して CO_2 濃度をチェックしましょう。原稿の締め切りはとうに過ぎていますしこの章は終わりです。ありがとうございました！

ということにしたいのですが、Amazon にある Netatmo の商品ページを見てみると国内正規代理店品は在庫切れになっており、並行輸入品が 26,380 円（2019 年 8 月 27 日現在）となっています。[*2] この値段で健康が買えるなら安い！ と思える方はそれで良いのですが、2 万円あれば 3D プリンタが購入できる時代に 26,380 円はちょっと気軽に買うにはハードルが高いですね。

さらに Netatmo の場合、センサー自体を Wi-Fi 環境につないでクラウド上にデータをアップロードするため、オフィスで利用するにはネットワーク管理者にセキュリティ上のリスクを説明した上で許可をもらう必要があり、気軽に使うにはハードルが高そうです。

少し調べてみると、CO_2 濃度を測定できるセンサーは 2700 円ほどで購入できます。このセンサーと、どのご家庭にも転がっている Arduino を組み合わせて動かす事ができれば、10 分の 1 の出費で CO_2 濃度測定デバイスを作れそうです。Arduino とセンサーの組み合わせならネットワーク接続する必要もないので会社に持ち込んでも問題なさそうです。自分で作るなんて面倒ですしやりたくないですが、10 分の 1 と聞くとモチベがわいてきますね。

[*2] https://www.amazon.co.jp/dp/B00FFS73GG

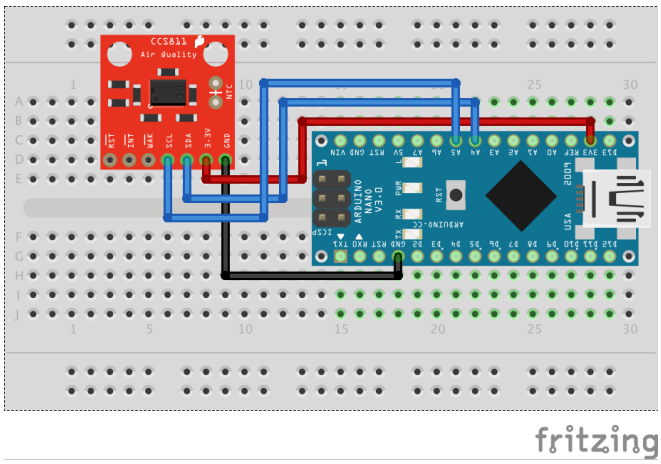
4.4 CCS811 と Arduino で CO_2 濃度を取得してみる

今回はスイッチサイエンスで購入できる CCS811[*3] というセンサーを利用して CO_2 濃度を取得してみます。[*4]

スイッチサイエンスの商品説明欄にあるように、CCS811 は空気品質センサモジュールで、純粋な CO_2 のみを検出するのではなく揮発性有機化合物を検出し、その値をもとにして CO_2 濃度を出力します。そのため、ガスコンロやストーブの燃焼などによる空気品質の低下が測定結果に大きな影響を与えます。

まずセンサーから CO_2 濃度を正しく取得できる最低限の環境を作ります。センサーの駆動と値の取得には Arduino Nano を利用します。

CCS811 は I^2C で接続できるので、Arduino との接続は簡単です。 電子回路設計ツールの Frizting[*5] を利用してブレッドボード上での配線図をかいてみました。みなさんはこれを参考に配線してみてください。



▲図 4.2　CCS811 と Arduino Nano を接続

図だけだと読み取りづらいと思うので、CCS811 のどのピンを Arduino のどのピンに接続すればよいかを記した対応表 (表 4.1) を用意しました。列がデバイス、行が接続するピンをあらわしています。CCS811 の VSS を Arduino の 3V3 に接続という具合です。SCL と SDA に関しては、Arduino の種類によってピンが違う場合があるので自分が使う Arduino の種類にあわせて読みかえてください。

[*3] https://www.switch-science.com/catalog/3298/

[*4] CO_2 濃度を測定できるセンサーは CCS811 の他にも S-300L-3V や MH-Z19 などがあります。いくつか試してみた感じだと CCS811 が一番てっとり早く使えてエージングだけで比較的それっぽい値がでてくるようになりました。

[*5] https://fritzing.org/home/

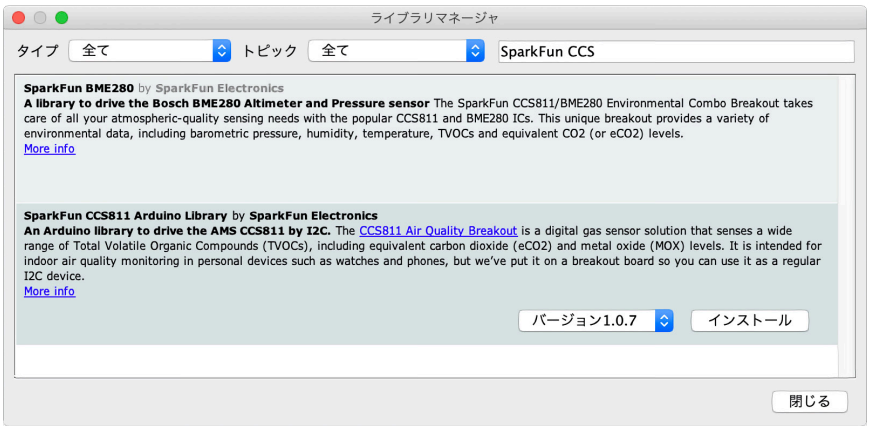
▼表 4.1　CCS811 配線表

CCS811	Arduino
VSS	3V3
GND	GND
SCL	A5(SCL)
SDA	A4(SDA)

配線が完了したら、次はソフトウェアです。Arduino に制御プログラムを書き込む必要があるので、パソコンに Arduino IDE[*6]をインストールします。

今回利用する CCS811 センサーの製造元である SparkFun が、センサーデータを簡単に取得するためのライブラリとサンプルコードを公開しているので、そのまま使わせてもらいましょう。

ライブラリをインストールするには、Arduino IDE の「スケッチ」から、「ライブラリをインクルード」を選び、その中にある「ライブラリの管理」から、「ライブラリマネージャ」を開いて、検索窓から「SparkFun CCS811」で検索すると「SparkFun CCS811 Arduino Library」が表示されるので、インストールボタンを押すとインストールできます。



▲図 4.3　「SpackFun CCS811 で検索すると見つけられます

ライブラリのインストールができたらリスト 4.1 のコードを書き込んでください。

▼リスト 4.1　CCS811 を動かす最小限のスクリプト

[*6] https://www.arduino.cc/en/Main/software

```
 1: #include <Wire.h>
 2: #include <SparkFunCCS811.h>
 3:
 4: #define CCS811_ADDR 0x5B
 5:
 6: CCS811 mySensor(CCS811_ADDR);
 7:
 8: void setup() {
 9:   Serial.begin(115200);
10:   Serial.println("CCS811 Basic Example");
11:
12:   Wire.begin();
13:
14:   CCS811Core::status returnCode = mySensor.begin();
15:   if (returnCode != CCS811Core::SENSOR_SUCCESS)
16:   {
17:     Serial.println(".begin() returned with an error.");
18:     while (1);
19:   }
20: }
21:
22: void loop() {
23:   if (mySensor.dataAvailable())
24:   {
25:     mySensor.readAlgorithmResults();
26:
27:     Serial.print("CO2[");
28:     Serial.print(mySensor.getCO2());
29:     Serial.print("] tVOC[");
30:     Serial.print(mySensor.getTVOC());
31:     Serial.print("] millis[");
32:     Serial.print(millis());
33:     Serial.print("]");
34:     Serial.println();
35:   }
36:
37:   delay(10);
38: }
```

書き込みに成功したら、シリアルモニタを開くと図 4.4 のように CO_2 濃度と tVOC、実行時間がミリ秒で表示されます。やりましたね。

▲図 4.4　シリアル通信で CO_2 の値を取得

ただし、CCS811 は最低 48 時間のエージングと 20 分のコンディショニングが必要なため、起動直後は正しい値が出力されません。外気の CO_2 濃度が 400ppm 前後なので、最初のタイミングでは 400-1500ppm 程度の値が取得できていればひとまず大丈夫です。息をふきかけて数値が大幅に上がることを確認してもよいでしょう。

私は Netatmo を持っているのでその値と比較してみたところ、最初のうちは随分高い値が記録されるのと、48 時間経過後も瞬間的に高い値がでる事が多いようです。

温度計の場合は、1 度、2 度の差が体感で分かったり、部屋のどこに置いてもわりと近い値を示しますが、CO_2 センサーの場合、100-200ppm 程度の変化は体感ではわからないですし設置場所によってもかなり変わるので、細かい値の変化を見るというよりは 1500ppm を越えているか、通常時 600ppm 以下かどうかなどのざっくりとした状況の把握を目的にしたほうがよさそうです。

4.5 パソコンなしでも CO_2 濃度を確認したい

CO_2 濃度を確認する事ができるようになったわけですが、値はシリアル通信経由で PC から見ているので、パソコンを起動してシリアル通信している間しか値を確認する事ができません。業務時間中だけ動作させるのであればそれでもよさそうですが、会議室にノートパソコンを持って移動するたびにケーブルを外したりつないだりするのは避けたいです。

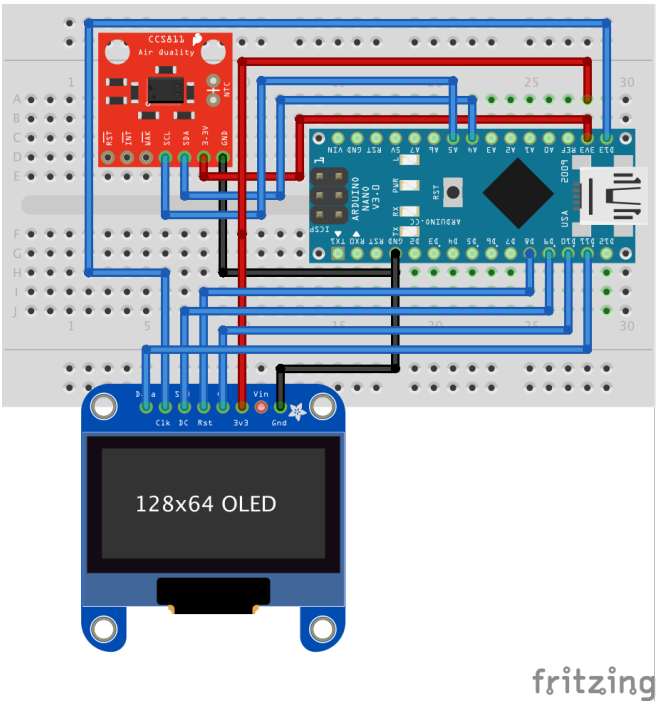
そこで、Arduino で小さい液晶を制御して、そこに取得した CO_2 濃度を表示させてみます。

4.6 OLED 液晶 SSD1306 を接続して CO_2 濃度を表示する

Arduino で制御できる液晶はたくさんあるのですが、今回はたまたま家にあった SSD1306 をつかいます。SSD1306 は非常に小さい上に OLED なので視認性が高く、通常のキャラクタ液晶ディスプレイのようにバックライトについて考慮する必要もないので便利です。私は ebay で購入しましたが、これも CCS811 と同様にスイッチサイエンスで購入することができます。[*7]

SSD1306 を手に入れたら 図 4.5 を参考に Arduino に接続してください。

[*7] https://www.switch-science.com/catalog/2729/



▲図 4.5　CCS811 から取得したデータを SSD1306 に出力

SSD1306 の接続方法には I^2C と SPI があるのですが、手元にあるデバイスが SPI しかなかったので SPI を利用します。

SPI は I^2C よりも高速に通信できるので、ディスプレイのような描画速度が求められるデバイスには良いのですが、I^2C よりも利用するピン数が多いので配線は面倒になります。

さきほど接続した CCS811 と Arduino の配線図に、さらに SSD1306 を追加しています。追加分の配線は 表 4.2 を参考にしてください。

▼表 4.2　SSD1306 配線表

SSD1306	Arduino
VSS	3V3
GND	GND
CLK	D13
DATA	D11
CS	D10
DC	D9
RST	D8

配線が完了したら次はソフトウェアです。SSD1306 への表示は U8g2 というディスプ

レイ用のライブラリを利用します。このライブラリはたくさんの種類のディスプレイに対応しており、初期値の設定だけで様々なディスプレイを同じように扱う事ができて便利です。

U8g2 のインストールは CCS811 の時と同じよう Arduino IDE のライブラリマネージャから「U8g2」で検索してインストールします。

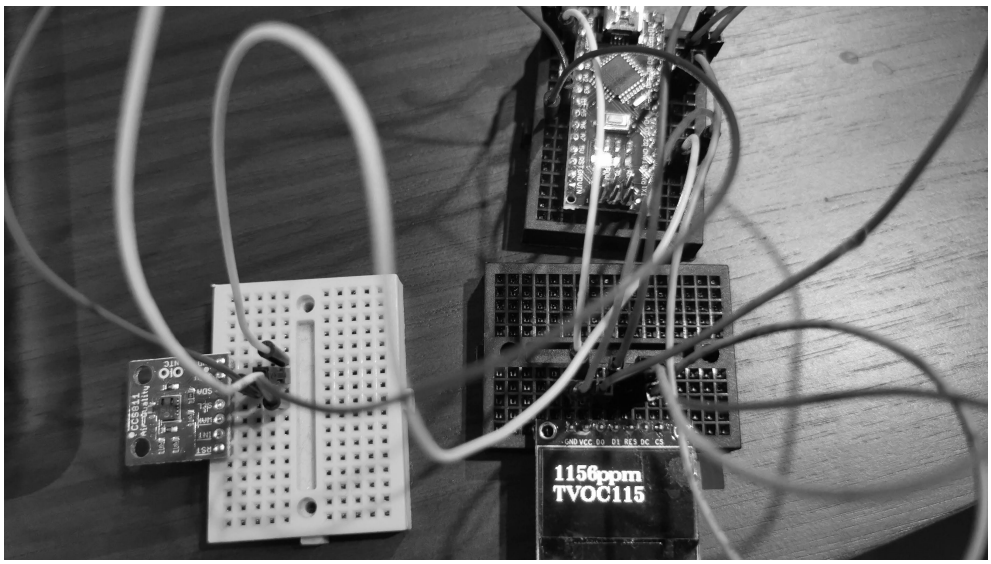
▲図 4.6　「U8g2 で検索すると見つけられます

▼リスト 4.2　CCS811 の出力結果を SSD1306 に表示する最小限のスクリプト

```
 1: #include <Wire.h>
 2: #include <SPI.h>
 3: #include <SparkFunCCS811.h>
 4: #include <U8g2lib.h>
 5:
 6: #define CCS811_ADDR 0x5B
 7:
 8: CCS811 mySensor(CCS811_ADDR);
 9:
10: U8G2_SSD1306_128X64_NONAME_F_4W_SW_SPI u8g2(
11:   U8G2_R0,
12:   /* clock=*/ 13,
13:   /* data=*/ 11,
14:   /* cs=*/ 10,
15:   /* dc=*/ 9,
16:   /* reset=*/ 8
17: );
18:
19: void setup() {
20:   delay(3000);
21:
22:   u8g2.begin();
```

```
23:   u8g2.clearBuffer();
24:   u8g2.setFont(u8g2_font_ncenB08_tr);
25:   u8g2.drawStr(0,10,"Loading...");
26:   u8g2.sendBuffer();
27:   delay(1000);
28:
29:   Wire.begin();
30:
31:   CCS811Core::status returnCode = mySensor.begin();
32:   if (returnCode != CCS811Core::SENSOR_SUCCESS)
33:   {
34:     u8g2.clearBuffer();
35:     u8g2.drawStr(0,10,".begin() returned with an error.");
36:     u8g2.sendBuffer();
37:     while (1);
38:   }
39: }
40:
41: void loop() {
42:   if (mySensor.dataAvailable())
43:   {
44:     mySensor.readAlgorithmResults();
45:     u8g2.clearBuffer();
46:     u8g2.setFont(u8g2_font_ncenB14_tr);
47:     u8g2.drawStr(0,20, (String(mySensor.getCO2()) + "ppm").c_str());
48:     u8g2.drawStr(0,40, ("tVOC" + String(mySensor.getTVOC())).c_str());
49:     u8g2.sendBuffer();
50:   }
51:
52:   delay(1350);
53: }
```

表示方法についてはグラフにするなど改良の余地がありそうですが、これでひとまず表示することができました。



▲図 4.7　ブレッドボード上で配線した場合

4.7　おわりに

急成長している会社において、オフィスの拡充が社員の増加に間に合わず人口密度が高くなっている場合や、シード期の小さなオフィスは、CO_2 濃度が高くなりがちで業務効率に悪影響を及ぼしている可能性があります。CO_2 濃度測定デバイスはそれなりに高いですが、自分でセンサーを購入すれば比較的安価に入手できるので、皆さんも CO_2 濃度測定デバイスを自作してオフィスでのパフォーマンス向上に役立てましょう。

第 5 章

「OK グーグル！ 銀行振込 1000 円」

Yoshio HANAWA / @hnw

5.1 はじめに

昨今、Siri や Alexa、Goole アシスタントなどの音声アシスタント機能が身近になってきました。筆者の自宅にも Google Home があり便利に使っています。特に布団の中から「OK グーグル！ 電気を消して」というと家のすべての電気が消えるのは最高ですね[*1]。

一方で「音声アシスタントの何が便利かわからない」「すっかり使わなくなってしまった」という声もよく聞きます。筆者も買った当初は「できることが限られている」「カスタマイズ性が低い」という印象で、どう使っていいかまったくわかりませんでした。

だからというわけではないのですが、筆者は音声入力で銀行振込を行うという、おそらく世間的にも珍しいシステムを作成してみました。

本稿ではこのシステムで利用した技術・ノウハウ・ハマりポイントなどについて説明します。また、本システムを実際に運用するならどの銀行が適しているか、何に気をつけるといいか、などシステム外の情報も合わせて紹介します。

5.2 動機

そもそも、筆者はなぜこんなシステムを作ろうと思ったのでしょうか？ 第一の理由は、前述の通り音声入力で誰もやったことがないことをしたら面白そうだ、と思ったためです。

また、筆者は自分の口座間で資金移動することが月に数回程度あるため[*2]、これを自動化したいというのも理由でした。

[*1] Google Home では別売の赤外線リモコンモジュールが必要です。

[*2] クレカの引き落とし口座が分散しているなどやむを得ない理由があるのです (?)

5.3 システムの要件

システムの要件は次の通りです。

1. 自動で銀行振込ができる
2. システムの金銭的な維持コストが 0 円
3. 銀行振込の処理は自宅の Raspberry Pi で動かす

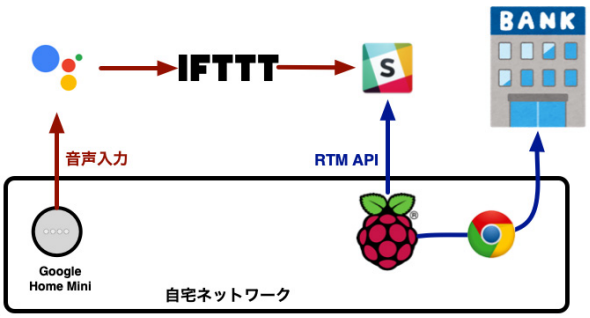
1 番目の要件は解説不要として、2 番目と 3 番目について説明します。

2 番目については、お金がもったいないとかではなくサービスの継続性を高めるための要件です。筆者は過去にクレカの更新を忘れてサーバを止めた経験があるため、個人用のシステムは可能な限り無料のサービスで構成してうっかりミスによるサービス停止を防ぎたいと考えています。

3 番目の要件ですが、銀行振込処理では銀行の ID・パスワードを保持する必要がありますから、できるだけセキュアにしたいですよね。家庭内ネットワークがそこまで安全とも思いませんが、心情的にクラウドに預けるのはイヤだと思ったので必須要件としました。

5.4 システムの概要

今回作ったシステムの全体図を紹介します（図 5.1）。



▲図 5.1　音声入力による銀行振込システムの概要図

振込をしたい人はまず Google Home に話しかけます。すると、振込金額などの情報が Slack のプライベートチャンネルに書き込まれます。このプライベートチャンネルを Raspberry Pi 上の bot が監視しており、書き込まれた情報を元に銀行振込コマンドを起動します。銀行振込コマンドは銀行のネットバンキングシステムに対してヘッドレスブラウザでアクセスし、人間と同じように複数ページを遷移して銀行口座間の振込を実施し

ます。

ブラウザの自動操作に利用したライブラリは Puppeteer[*3]です。Puppeteer については後で改めて説明します。

なぜ Slack か

前述の通り、銀行のネットバンキングサービスにアクセスする Raspberry Pi は筆者の自宅ネットワーク内に配置しています。そのため、Google アシスタントから Raspberry Pi に直接通信することはできません。そこで、両者をつなぐ道具として Slack を使うことにしました。

Slack では複数種類の API が提供されているのですが、その中の RTM API が Web Socket を利用しており、ファイアウォールを越えて自宅ネットワーク内にメッセージを送れる唯一の API です。今回はこれを使うことにしました。

ネットワークを越える通信を実現してくれるサービスなら何でもよいので、本来なら Amazon SQS や GCP の Cloud Pub/Sub などを使うべきところでしょう。今回は 0 円にこだわって Slack ということにしましたが、他に LINE Messaging API なども選択肢になるはずです。

なぜ Puppeteer か

銀行振込を自動処理したい場合、Beautiful Soup や Nokogiri など有名どころのスクレイピングライブラリは使えません。というのも、銀行のネットバンキングシステムは大半が JavaScript 必須のつくりになっているため、DOM 解釈しか行わないライブラリで `href`のリンクを辿るとエラーで怒られてしまうのです。

Puppeteer は Node.js 製のライブラリで、Chrome/Chromium の操作 API を提供するものです。プログラムからブラウザのフル機能が使えるので、銀行サイトの自動操作を問題なく実現することができます。

また、セットアップが簡単なのも優れている点です。「`npm i puppeteer`」とすればライブラリだけでなくブラウザもインストールされるので、即座に利用することができます。

他の選択肢としては Selenium が挙げられます。技術的には同等だと思うので、どちらを使うかは慣れと趣味の問題だと思います。

5.5 システムの詳細

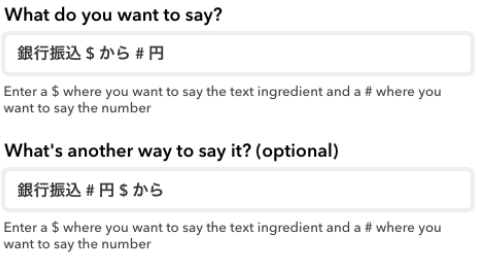
本システムは大きく 3 つのサブシステムに分かれます。これらについて順に説明していきます。

[*3] `https://github.com/GoogleChrome/puppeteer`

Google Home から Slack へ

Google アシスタントから Slack へのメッセージ送信には IFTTT を使っています。

IFTTT で Google アシスタントを扱う場合、固定文字列のコマンドの他に文字と数字とで自由入力部分を 1 つずつ作ることができます（図 5.2）。また、音声入力内容を組み替えたり追加したりして Slack に送るメッセージを指定することができます（図 5.3）。



▲図 5.2　Google アシスタントのコマンド文字列の指定

▲図 5.3　Slack への送信内容の指定

今回は振込額だけを指定するパターンと、振込額と支払元口座の 2 つを音声で指定するパターンと 2 種類を実現してみました[*4]。

Slack のテキストを元にコマンド実行

本システムでは Slack のメッセージを読み取る処理を Go で書きました。Raspberry Pi 上で動かす前提だったため、Go ならフットプリントが比較的小さく有利だろうという

[*4] 音声入力で多くの情報を指定すると認識率が下がりそうだったので、振込先をメイン口座に固定しました。

判断もありましたし、複数メッセージを並行処理する場合にも有利そうだと考えたためです。

処理としては、Slack の RTM API を利用して全メッセージを監視し、キーワードにマッチしたら外部コマンドを起動しています。起動したコマンドの標準出力やエラー出力は Slack のメッセージとして返されます。

また、TOML で起動コマンドの定義ができるように作っています（リスト 5.1）。

▼リスト 5.1　TOML による bot の定義例

```
slack_token = 'XXXX'

[[commands]]
keyword = '振込 *'
command = 'node /path/to/banking.js 振込 * -v'
```

このようなつくりにすることで Slack bot の作り方がわからない人でも bot が作れる、また好きな言語で bot が作れる、などのメリットがあると考えています。bot の詳細は GitHub リポジトリ[*5]をご確認ください。

Puppeteer による銀行振込

銀行サイトは JavaScript 必須のページが多いのですが、Puppeteer を使うと比較的簡単にページを辿ることができます。リスト 5.2 にコード例を示します。

▼リスト 5.2　Puppeteer スクリプトの例

```
const puppeteer = require('puppeteer');

(async () => {
  // 以下は開発時に便利な設定
  const options = {headless: false}; // ブラウザのウインドウを表示させる
  const browser = await puppeteer.launch(options)
  const page = await browser.newPage();

  // 指定 URL を開いた後、通信しなくなってから 500ms 待つ
  await page.goto('https://google.com', {waitUntil: 'networkidle0'});

  // 検索文字列をテキストフォームに入力する
  await page.type('input[name="q"]', 'puppeteer');

  // submit ボタンが隠れて再度現れるので、それを待つ
  await page.waitForSelector('input[type="submit"]', {visible: true});

  // submit ボタンをクリックして、画面遷移後に通信しなくなってから 500ms 待つ
  await Promise.all([
    page.waitForNavigation({waitUntil: 'networkidle0'}),
    page.click('input[type="submit"]')
  ]);
  // await browser.close(); // 開発中はブラウザを閉じない方が便利（HTML を DevTools で確認できる）
})();
```

Puppeteer スクリプトの基本スタイルは「遷移後に DOM 構築を待つ」「必要なら入力

[*5] https://github.com/hnw/slack-commander

フォームを埋める」「クリックする」の繰り返しです。処理が終わってもブラウザを close せず、DevTools で要素を特定していけば 1 ページずつ進んでいけるはずです。

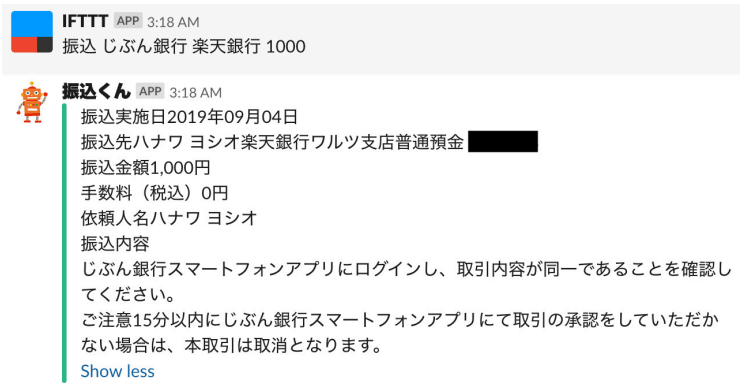
遷移後に DOM 構築を待つ方法としては `page.waitForNavigation()` のオプション `{waitUntil: 'networkidle0'}` が非常に便利で、これを使うだけで DOM 構築待ちのトラブルをかなり減らせます。また、JavaScript などにより時間差でレンダリングされるページでは `page.waitForSelector()` で要素の出現を待つと良いでしょう。

振込処理を実装するときのノウハウですが、今回作成したスクリプトでは振込自動化の対象を登録済口座のみにしました。これは銀行振込の実装を楽にするのとリスク回避と両方の意味で有効です。

また、事故で 1000 回振込が走ってしまった場合でも破産しないように、手数料が 0 円のときだけ振込を実行するようなバリデーション処理も入れました。

さらに、振込時にスマホで認証できる銀行だと万一の事故があった場合でもリスクが限定的になるのでオススメです。

実際の動作例が図 5.4 です。この表示後にスマホで認証して振込を確定させています。



▲図 5.4　音声入力から振込を実施する例、スマホ認証を促すメッセージが表示されている

5.6　利用銀行の選定

このシステムを実用的に使うには、少なくとも月 3 回程度は他行宛振込手数料が無料でないと厳しいように思います。また、前述の通りスマホ認証を提供している銀行の方が安心でしょう。

これらを無条件で満たす銀行はないのですが、預金額など一定の条件を達成できれば次の 3 行が選択肢として挙げられます。

- 住信 SBI ネット銀行 （スマプロランク 2 以上）
- じぶん銀行 （じぶんプラス +4 以上）

- 新生銀行 （新生ゴールド以上）

詳細は各行の優遇プログラムをご確認ください[*6][*7][*8]。
また、次の 2 行についてはスマホ認証こそありませんが、振込手数料の無料回数が多いのでオススメです。

- スルガ銀行 （スルガ STAR プログラム 4 ツ星）
- 大和ネクスト銀行

筆者の知る限り、それ以外の銀行は振込手数料の優遇条件が厳しかったり Puppeteer での実装が難しかったりして自動化に適さないように思います。

5.7 所感

このシステムを実際に使ってみた感想としては「意外と実用的だ」ということです。
まず、Google アシスタントの認識率が予想より高く、誤認識はほとんどありませんでした。これは入力コマンドを 2 単語または 3 単語にしたのも良かったかもしれません。コマンドを文章にしたり 4 単語以上にしたりすると認識率は下がりそうです。
筆者自身が音声入力での振込を利用し続けるかは疑問ですが、年配の方などで今でもテレホンバンキングを使っている人がいるとしたら、その大半を本システムで置き換えられそうな感触でした。
筆者にとっては音声入力よりも Slack から銀行振込ができることの方が断然便利だと感じました。普段から使い慣れているツールであることもありますが、成功にせよ失敗にせよ実行結果が Slack 上にログとして残るのも良い点です。また、Slack のリマインダー機能を使って `cron`や `at`の代わりにすることも可能です。コマンド実行環境として Slack が便利だというのは面白い発見かもしれません。

5.8 ハマったこと

本システム制作中のトラブルについても紹介します。

Google アシスタントの表現ブレ

Google アシスタントで数字を認識する場合、ちょっとした罠があります。数字が一定以上になると音声認識の結果が数字ではなく数字漢字混じりに変換されてしまい、IFTTT アプレットの条件にマッチしなくなることがあるのです。具体的には「銀行振込 10000

[*6] 住信 SBI ネット銀行 スマートプログラム `https://www.netbk.co.jp/contents/lineup/smartprogram/`
[*7] じぶんプラス `https://www.jibunbank.co.jp/jibunplus/`
[*8] 新生ステップアッププログラム `https://www.shinseibank.com/powerflex/relationship/`

円」となってほしいところが実際には「銀行振込 1 万円」となり、文字列マッチに失敗します。

いまのところ「# 円」「# 万円」の 2 パターンしかバリエーションがないようなので両者を登録して対処していますが、新たなパターンが見つかった場合に都度登録する必要があってイマイチですね。

サイトごとの実装難易度に差がある

Puppeteer での実装は大半の銀行サイトでは順調に進みます。しかし、一部の銀行サイトではレンダリングを JavaScript で行っていて、DOM 構築が不完全なうちにボタンをクリックしようとして何も起こらない、という事故が起きたりします。こうしたサイトは汎用的な対処が難しく、いまのところ遷移のたびに数秒待つくらいしか対策を思いついていません。

新生銀行でデバッグしすぎて怒られた事件

筆者は新生銀行のデバッグのため振込ページ付近の遷移を繰り返していたのですが、結果として何かの上限に達してしまったようで、振込の際に次の警告文が出るようになってしまいました。

> お客さまの振込先口座確認サービスのご利用を制限させていただいています。振込先登録口座の情報に誤りが無いかご確認のうえ、振込手続を行ってください。なお、振込先口座確認サービスの利用再開をご希望される場合、新生パワーコール（0120-456-858 ：＊ 1 → 2 ＜スマホ認証サービス・操作方法などに関するお問い合わせ＞）までご連絡ください。

どうやら犯罪防止用 (?) の機構を発動させてしまったようです。遊びで銀行さんに負担をかけてしまうのはよろしくないですね。同様の取り組みをされる皆さまは Puppeteer スクリプトを一発で正常動作させるよう頑張ってみてください。

ちなみに電話してみたところ「今回は復旧させますが 2 回目の復活はできないので注意してください」と言われました。

5.9 まとめ

- 音声入力から銀行振込する仕組みを作成した、事前の予想より実用的だった
- コマンド実行環境として Slack は優秀
- 銀行サイトでデバッグしすぎると怒られる

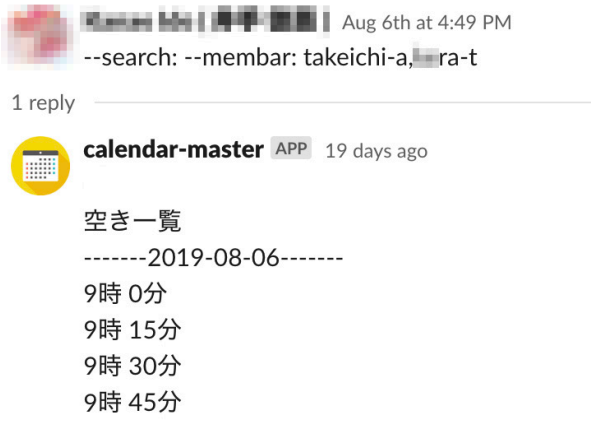
第 6 章

GoogleCalendar を扱う Slack Bot の製作記

Ayato Takeichi / @takeichi-a
Toshifumi Umezawa / @umezawa-to

6.1　この章について

会議などの予定の調整に苦労した新卒 2 人が協力して、Slack に話しかけることでカレンダーの予定の取得や確保をしたり、複数人の空きが重なっている部分を確認できる Bot を作成したことについての記事です。



▲図 6.1　Slack 上で takeichi-a ともう一人の共通の空き時間を探す例

前半の「6.2 クライアント側の紹介」から「6.6 操作方法について」までをクライアントサイドの得意な takeichi-a が Bot の開発について紹介し、後半の「6.8 サーバー側の紹

介」から「6.10 各コンテナの実装」はサーバーサイドの得意な umezawa-t が Bot を動かす環境の構築について紹介します。

6.2 クライアント側の紹介

takeichi-a です。

予定の調整を行う場合苦労することがありませんか？ 私はあります。会議室の空きを確認する手間がかかったり、メンバーと都合のつく時間がなかなか見つからないなどストレスを感じることが多くありました。

また、同期などにこの話をするとみんな同じ苦労をしていることがわかりました。このストレスを無くすために自動化しようと考え制作を開始しました。

製作するにあたって、自分はクライアント側の製作に集中したかったので、同期でサーバーの能力の高い umezawa-t さんを誘い 2 人で作業を開始しました。

KLab では GoogleCalendar と Slack を利用しているので、SlackBot として実装すれば新しいツールを配布、導入する必要がなくスムーズに活用できるからです。

Bot の開発は Go 言語で行い、GoogleCalendarAPI を使ってカレンダーの操作をしました。また、Go 言語はあまり使ったことがなかったのですが、構成を工夫したことについても紹介します。

6.3 準備するもの

今回の環境では Go 言語と GoogleCalendarAPI・SlackAPI を利用します。Go 言語を利用可能な環境を用意したあとにリスト 6.1 を行うことでそれぞれの API をプログラムから利用することができます。

▼リスト 6.1　API を使用可能にするための go get

```
go get google.golang.org/api/calendar/v3
go get github.com/nlopes/slack
```

また Slack の Bot 利用やカレンダーのアクセスのために次の作業も必要です。

- Slack 内にある機能のひとつの Bot user の作成
- Google 内にある機能のひとつ Google プロジェクトの作成と CalendarAPI を利用可能にする

6.4 予定を取得

GoogleCalendarApi のページには複数言語のサンプルが用意されていて、そこに Go のサンプルもあったためそれをそのまま利用してみました。

実際にイベントを取得してくる命令がこちらです。

▼リスト 6.2　カレンダーからイベントを取得するプログラム

```
srv.Events.List("primary").ShowDeleted(false).SingleEvents(true)
    .TimeMin(t).MaxResults(10).OrderBy("startTime").Do()
```

このコードは「イベント＋自身の予定＋削除された予定は表示せず＋繰り返しイベントはひとつの予定として扱い＋指定時刻から＋最大１０件＋開始時間順にソートし＋取得する」となっています。

.List({ID})の ID を取得したいカレンダーの ID にすると、対象のカレンダーの予定を取得することができます。

今回参照したいカレンダーは ID が規則的な文字列では無かったので、対応した map を作成し呼び出しやすくしました。

▼リスト 6.3　map 化した ID

```
var IDList = map[string]string{}
IDList["カレンダー A"] = "fghweojiosmlkdnfjbhifspjoaklmksdjbfhhjoi"

// 利用例
srv.Events.List(IDList["カレンダー A"])
```

▲図 6.2　カレンダーの ID

6.5 共通の空き時間を算出する

複数のカレンダーから共通の空き時間を算出するために、それぞれのカレンダーの予定の入っている時間をビットフラグに落とし込みました。

9 時から 21 時の 12 時間を 15 分刻みを検索の条件にし、そのデータの保存方法を日付をキーにした map[string]uint64{}にしました。実際使用するデータのサイズは 12(時間)*4(60分/15分)=48bitなので、64bit 整数に収まります。

▼リスト 6.4　保存されるデータの例

```
{
    "2019-08-20" : 0000000000000000000000000000000011110000110000000000110011110000
    "2019-08-21" : 0000000000000000000000000011111111000000001111000000001111100000000
}
```

9 時から 4bit ずつ右から保存されているので、この例の場合 2019 年 8 月 20 日は

- 10 時から 1 時間
- 11 時半から 30 分
- 14 時半から 30 分
- 14 時から 1 時間

にすでに予定が入っていて、それ以外の時間が調整可能ということになります。

まずはデータを入れるための準備を行います。

▼リスト 6.5　指定期間の日をキーにした map

```
m_schedules := map[string]uint64{}
layout := "2006-01-02"
for ; startTime.Day() <= eneTime.Day(); startTime = startTime.AddDate(0, 0, 1) {
    date := startTime.Format(layout)
    m_schedules[date] = 0
}
```

処理する期間の日付を元に辞書型で入れ物を作成しています。

次に予定が 9 時から 21 時の間に収まっているかをチェックし、そうでない場合は正しく処理されるように補正をかけます。

▼リスト 6.6　チェック、補正をかける処理

```
startTime, _ := time.Parse(time.RFC3339, item.Start.DateTime)
endTime, _ := time.Parse(time.RFC3339, item.End.DateTime)
if startTime.Hour() < 9 {
    if endTime.Hour() < 9 {
        //予定が 9 時以前の場合処理を行わない。
        continue
    }
    //早朝から開始し 9 時以降までかかる予定は開始時間を 8 時に合わせます。
    startTime =
```

```
        time.Date(startTime.Year(), startTime.Month(), startTime.Day(), 9, 0, 0, 0, timeLoc)
}
if endTime.Hour() > 21 {
    //予定の終了時間が 21 時を超えていた場合は終了時間を 21 時に合わせます。
    endTime =
        time.Date(endTime.Year(), endTime.Month(), endTime.Day(), 21, 0, 0, 0, timeLoc)
} else if endTime.Hour() < 9 {
    //予定の終了時間が 0 時をまたいだ場合の処理も同じで 21 時に合わせます。
    endTime =
        time.Date(endTime.Year(), endTime.Month(), startTime.Day(), 21, 0, 0, 0, timeLoc)
} else if endTime.Sub(startTime).Hours() > 24 {
    //予定が丸一日以上続く場合は分割します。
    //開始時間を次の日にした予定を追加し、今回分の予定は 21 時で締める
    newEvent := item
    nextStartTime :=
        time.Date(endTime.Year(), endTime.Month(), startTime.Day()+1, 9, 0, 0, 0, timeLoc)
    newEvent.Start.DateTime = nextStartTime.Format(time.RFC3339)
    events.Items = append(events.Items[:idx+1], events.Items[idx:]...)
    events.Items[idx] = newEvent
    endTime =
        time.Date(endTime.Year(), endTime.Month(), startTime.Day(), 21, 0, 0, 0, timeLoc)
}
```

そして、チェックされだデータを変換、格納します。

リスト 6.7 でビット変換計算を行う時に 36 を引いている理由は、開始時間の 9 時を 0bit に合わせるためです。(9 時間× 4bit=36)

▼リスト 6.7　対応するビットに変換する処理

```
// 予定の開始時刻のビット位置を計算
startTimeBit := uint64(startTime.Hour()*4 - 36)
switch {
case startTime.Minute() >= 45:
    startTimeBit += 3
    break
case startTime.Minute() >= 30:
    startTimeBit += 2
    break
case startTime.Minute() >= 15:
    startTimeBit++
    break
}
// 予定の終了時刻のビット位置を計算
endTimeBit := uint64(endTime.Hour()*4 - 36)
switch {
case endTime.Minute() > 45:
    endTimeBit += 3
    break
case endTime.Minute() > 30:
    endTimeBit += 2
    break
case endTime.Minute() > 15:
    endTimeBit++
    break
case endTime.Minute() == 0:
    endTimeBit--
    break
}
// 予定の時刻にビットを立てる
date := startTime.Format(layout)
for i := startTimeBit; i <= endTimeBit; i++ {
    m_schedules[date] |= 1 << i
}
```

これを繰り返し、すべてのカレンダーの m_schedulesで日別に論理和を取れば bit が 0

の部分が空き時間となります。

6.6 操作方法について

Slack での発言で操作をするため、呼び出しコマンドの書式を決めました。

▼リスト 6.8　コマンドの書式

```
--{コマンド名}:コマンドの内容

例: --searchFree:カレンダー A, カレンダー B, カレンダー C
```

▼リスト 6.9　コマンドを判別する

```
//メッセージの text からコマンド名を取り出す
snum := strings.Index(text, "--")
enum := strings.Index(text, ":")
// 検索文字が見つからない場合の処理を省略
command := text[snum+2:enum]

//コマンドの内容テキストを取り出す（前後の空白を除去）
retText := strings.Trim(text[enum+1:], " ")

//コマンドが登録されている辞書に一致するものがあるかを判定する
_, ok := commandList[command]
if ok {
    return retText, command
}
```

Bot を呼び出すユーザーはこの書式に従い欲しいカレンダーを指定するだけで空き時間が返ってくるようにしてあります。さらに時間指定などにも対応するためにリスト 6.10 のようなオプションコマンドにも対応しました。

▼リスト 6.10　オプションコマンドの追加

```
「--startTime:」「--endTime:」「--startDay」「--endDay」
```

リスト 6.11 のように使用します。ここでは日付指定が省略されているため今日の日付が設定され、13 時から 17 時の間でカレンダー A,B 共に空いている時間が返ってきます。

▼リスト 6.11　オプションコマンドの使用例

```
--searchFree:カレンダー A, カレンダー B --startTime:13:00 --endTime:17:00
```

これを実装している時、Go 言語のプログラム面で不便に感じた部分がありました。リスト 6.12 ような形でコマンド別の処理の行き先を判定していたため、コマンドが増えるたびにケースを増やす必要があり追加を忘れることも時々発生していました。

▼リスト 6.12　不便な switch 文

```
switch msgCommand {
case "get":
    getCalendarEvent()
    break
case "insert":
    insertCalendarEvent()
    break
case "searchFree":
    searchFreeCalendarEvent()
    break
}
```

他の言語のようにクラス化を行うことができれば処理を纏めることができるとは考えていたのですが、Go 言語にクラスの機能はありませんでした。

しかし、調べていると Go 言語でクラスを再現してみた記事が数個見つかりました。これを参考にしながら組んだ形がこちら。

▼リスト 6.13　基本クラスとなるもの

```
var (
        commandList = map[string]commandSet{}
)

type commandSet struct {
        do func(commandSet)
    // 他フィールドは説明しないため省略
}

func (u commandSet) commandDo() {
        u.do()
}
```

構造体に関数を持たせ、レシーバ関数で構造体内部の関数を呼び出しています。

実際に追加する処理がこちら。

▼リスト 6.14　派生クラスとなるもの

```
func init() {
        commandList["get"] = commandSet{
                do: func(u commandSet) {
            //カレンダーを取得する処理
        },
    }
}
```

init() 関数で構造体を定義し処理を上書きすることにより、呼び出し側はコマンドの先を意識する必要がなくなりました。

▼リスト 6.15　呼び出しコマンドをキーに対応する処理を呼び出す

```
commandList[msgCommand].commandSetDo()
```

この改良により、コマンドを追加や編集するたびに関連する部分をすべて修正する作業が無くなり製作速度の向上に繋がりました。

6.7 bot の今後の展開

現在実装されている機能は「予定の取得」、「空き時間の探索」、「予定の確保」のみです。追加したい機能として次のようなものを考えています。

- 空き時間を探索したあとで新たに予定の確保のコマンドを実行する必要があり不便なので「空き時間を探索した結果を使いそのまま予定を抑える」
- 近いうちに会議室を使いたいが空いている場所が無い。空きができるのを見張っているのは時間の無駄、なので「会議室を使う予定がキャンセルされ空きが発生したことを教えてくれる機能」

また、現状のものを何人かに操作してもらったところエンジニアの方はすぐに理解して操作できていましたが、非エンジニアの方は使いづらそうでした。なので、GUI タイプの操作ができるようにする必要があり現在も製作中です。

6.8 サーバー側の紹介

ここからの執筆は umezawa-to が担当します。

私は機能が実装できた時点で満足してしまうタイプで、人に使ってもらえるようなアプリに仕上げる部分はとても苦手です。今回、フロントの処理は takeichi-a がサクサクと作り上げてくれたので、私はサーバー上でないとできない外部からのリクエストを受け付ける環境を作ったり、そのような環境を管理がしやすい形で用意する部分について協力しました。

まともに機能するような環境をゼロから作る経験はあまりなかったですし、今回はじめて Docker を使ったりリバースプロキシを扱ったり go の動く環境を触ったのですが、最終的には自分なりに満足できる構成に仕上げることができました。

ここでは、その構築内容を簡潔に紹介したいと思います。何かサーバー側の環境を作る際の参考になれば幸いです。

6.9 概要

要件

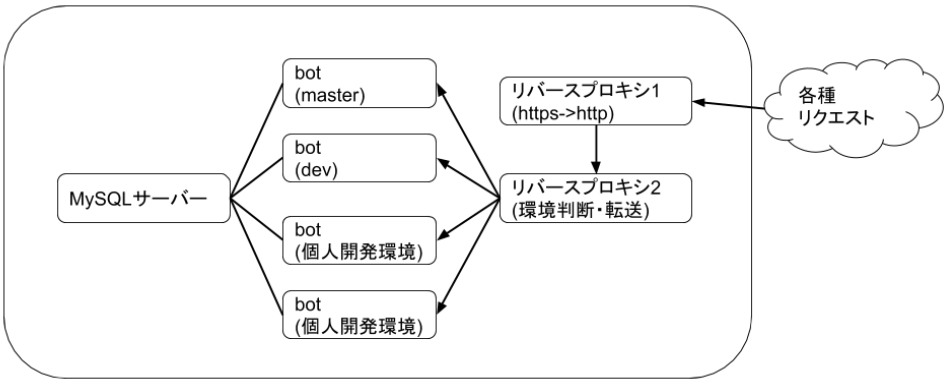
今回用意するサーバーの要件は次のとおりです。

- bot の本体は go で書かれている
- 本番・開発・個人などの環境も用意
- github の指定ブランチを自動追従する設定にできる
- webhook などのリクエストを個別の環境で受け取る

本番や開発用の bot は自動でデプロイされるようにし、個人環境などはファイルをアップロードすると反映される柔軟な環境が必要です。また、Google の OAuth 認証時のリダイレクトや、その他 GitHub からの webhook をそれぞれの環境に飛ばせる必要があります。

構成

ここまでに挙げた要件を考慮しつつ、最終的に図 6.3 のような形で動くようにしました。特徴的なのは、すべてのアプリが一つのマシン内で個別の Docker コンテナとして動作している点と、リバースプロキシを 2 つ用意している点です。これらの Docker コンテナが AWS の EC2 上で立ち上がっています。



▲図 6.3　用意したサーバーの構成

以下、図 6.3 の各アプリの概要です。

- リバースプロキシ 1：https-portal[*1]コンテナ、HTTPS を終端する
- リバースプロキシ 2：go で用意したサーバー、url を見て対応する環境にリクエストを転送する
- bot：bot の実行環境、git リポジトリの変更をうけて自動でデプロイする

リバースプロキシ 1 を用意しているのは、自分で用意する go のサーバーで https を受け取らなくてよくするためです。特に Google の OAuth 認証では https が必須なのですが、https-portal というコンテナイメージを利用することで、証明書を自分で扱う必要がなくなります。

リバースプロキシ 2 では、"/{環境名}/*"のリクエストを受け取ると"/*"に変換し、指定の環境名にあたるポートに転送する処理を行なっています。slack の slashcommand の

[*1] https://hub.docker.com/r/steveltn/https-portal/

リクエスト先や、Google の OAuth 認証時のリダイレクト先として bot を指定する際に、環境名を入れた url を指定することで、その環境に対するリクエストを送ることができるようになります。ただし、例外的に github からの webhook は全環境向けにリクエストを複製するようになっています。これによって github 側の webhook 設定が一回で済むようにしています。

各 Docker コンテナ間の通信は、すべてホストマシンを経由して行います。あるコンテナから別のコンテナにリクエストを送る際には、コンテナが所属する Docker ネットワークのデフォルトゲートウェイのアドレスに対して送ります。デフォルトゲートウェイのマシンはコンテナ起動元のホストマシンなので、ホストマシンの指定したポートに対してリクエストを送ることができます。

本来なら、コンテナを同じ Docker ネットワークに所属させておけばコンテナ名を使って直接通信できるのですが、そのような設定をする前に今の状態で稼働させてしまったので、現状はこのままにしています。ホストマシンのポートをいくつも解放した状態になっていますが、今回は AWS 側で外部からのアクセスを拒否するようにしているため問題ありません。

動作環境

Debian 上に Docker をインストールし、sudo なしで実行できる状態にしています。得体の知れない人やスクリプトに docker コマンドが実行されないように気をつけましょう。今回の環境では docker-compose の version3 が使えるように、最新のバージョンを github より直接入手しています。

また、github 上の bot のリポジトリにアクセスできるように、事前にサーバー上に鍵ファイルを用意し、Deploy keys に追加しています。

今回の構成では、bot の動くコンテナ内でも git が使えるようになっていますが、git 操作を行うために毎回コンテナに入るのは面倒なので、一応ホストマシンにも git を入れています。

6.10　各コンテナの実装

MySQL コンテナ

MySQL の Docker イメージは公式のものがあり、それを使うと root のパスワードなどを設定して起動するだけで使えます。MySQL のようなメジャーなアプリは公式の Docker イメージが存在するので便利ですね。

Docker コンテナ起動は docker-compose コマンドを利用します。リスト 6.16 は docker-compose の設定ファイルの例です。

▼リスト 6.16　docker-compose-mysql.yml

```
version: "3"

services:
    mysql:
        image: mysql:5.7
        environment:
            MYSQL_DATABASE: dbname
            MYSQL_USER: user
            MYSQL_PASSWORD: password
            MYSQL_ROOT_PASSWORD: root_password
        ports:
            - 3306:3306
        volumes:
            - ./db/mysql_data:/var/lib/mysql
```

yml ファイルが書けたら起動します。

```
$ docker-compose --file docker-compose-mysql.yml up -d
```

これで Docker コンテナ内で MySQL サーバーが起動し、ホストマシンの 3306 番からコンテナの 3306 番へのポートフォーワーディングによってアクセスできるようになりました。次のどちらかの方法で接続します。

```
mysql -uuser -p -h 127.0.0.1
mysql -uuser -p -protocol tcp
```

起動したサーバーに go のプログラムから接続するためにはリスト 6.17 のように指定します。

▼リスト 6.17　go から Docker で起動した MySQL サーバーに接続

```
import (
        "database/sql"
        _ "github.com/go-sql-driver/mysql"
)

db, err := sql.Open("mysql", "user:password@tcp(127.0.0.1:3306)/dbname")
```

Docker コンテナの MySQL に `localhost`で繋げない

MySQL のコンテナへのアクセスには IP アドレスを指定するか、プロトコルの指定をしています。
これらの指定をせず localhost に接続しようとした場合、Unix ドメインソケットに接続しようとしてしまうのですが、コンテナで起動している場合ホストマシンには Unix ドメインソケットのファイルがありません。
これを回避するために、明示的に TCP での接続として扱う必要があります。

https-portal コンテナ（リバースプロキシ 1）

リクエストの一時受けのコンテナですが、ここは https-portal という既存の Docker イメージを利用することで簡単に実現できます。このイメージの利用方法の説明も交えつつ、設定した内容の紹介をしたいと思います。

https-portal というイメージは、https でやってきたリクエストを http に変換して投げ直してくれ、さらに Let's Encrypt の証明書ファイルの取得と自動更新までやってくれるものです。official なイメージではないですが、DockerHub で検索すると steveltn/https-portal というイメージが見つかります。スターの量や github での開発状況などを見て、今回はこれを信頼して利用することにします。

今回は次のような docker-compose ファイルを用意しました。

▼リスト 6.21　docker-compose-https-portal.yml

```
version: "3"

services:
    https-portal:
        image: steveltn/https-portal:1
        ports:
            - 80:80
            - 443:443
        restart: always
        environment:
            DOMAINS: 'www.example.com -> http://dockerhost:8001'
            STAGE: staging
        volumes:
            - ./ssl-certs:/var/lib/https-portal
```

https-portal 固有の設定項目として DOMAINS と STAGE があります。

DOMAINS にはリクエストの転送先を指定します。例のように'->' を使って記述することで、左辺のアドレスでのアクセスを右辺のアドレスで投げ直してくれます。今回は使っていませんが、矢印を'=>' にするとリダイレクトになったり、ドメイン名だけを記述するとコンテナ内に静的ファイル置き場ができたり、これらをコンマ区切りで複数個指定できたりもするので、機会があればそれらも使ってみたいです。

STAGE では証明書の取得方法を表す 3 種類の中から指定します。'local' だと自己証明書、'staging' だと名前のとおりテスト用の証明書を、'production' だと本番用の証明書を取得します。本番の証明書を何度も発行していると発行上限に達してしまうので、テスト中は staging を指定しましょう。

ports の設定で 80 番ポートを開いていますが、これは https-portal の証明書取得の処理にてドメインの検証に使われているようで、設定しておかないと証明書の取得に失敗します。

今回は投げ直す先として'dockerhost' を指定していますが、これは https-portal のイメージ内でコンテナ起動元のホストマシンを指します。なので、ホストマシン上の 8001 番ポートに対してのアクセスになります。

bot が動くコンテナ

bot が動くコンテナについては、既存のイメージをそのまま使うのではなくプログラム側の対応や Docker イメージの用意が必要なので、自動デプロイに関する部分とそれが動くコンテナの準備についての２つに分けて説明します。

自動デプロイ

スクリプト言語であればソースファイルを置き換えるだけで新しい処理が走るようになりますが、go はコンパイル言語なので新しいソースをビルドして再起動しなければいけません。

この対応には go の fresh というライブラリを利用します。ファイルの変更を監視し、変更があった時にビルドし直して実行してくれます。fresh を `go get`した後、プロジェクトのルートのディレクトリにて fresh コマンドを実行することで、監視とリビルドが行われるようになります。

```
$ go get github.com/pilu/fresh
$ fresh
```

自動で再起動するようにはなりましたが、プログラムの変更によって必要なモジュールが増えた場合には対応出来ないかもしれません。

ここで、go1.11 以降は環境設定として GO111MODULE という値が存在します。これを on に設定しておくことで、事前に必要モジュールを get しておかなくても build 時に自動で取得してくれるようになります（1.12 以降ではデフォルトで on になっています）。

ここまでの設定を済ませることで、ファイルの変更によってデプロイが完了する環境ができました。あとは個々の環境ごとにプロジェクトのディレクトリを用意して fresh で実行させておけば完了です。

github の webhook によるデプロイは、各環境ごとに受け取りポートを指定しておき、リクエストがあれば pull を実行するという形で、リスト 6.19 に示すように実装しました。

▼リスト 6.19　http.ListenAndServe

```
// webhook のハンドラ
func githubWebhookHandler(w http.ResponseWriter, r *http.Request) {

    // webhook が github から送られてきたか検証

    // pull 実行
    err = exec.Command("git", "pull").Run()
    if err != nil {
        fmt.Println("err in exec", err)
    } else {
        fmt.Println("Pull Success!", err)
    }
}
```

```
// /github に上記ハンドラを登録
http.HandleFunc("/github", githubWebhookHandler)

// http でアクセスできるポートを開ける
http.ListenAndServe(":80", nil)
```

bot が動いているディレクトリのブランチを、上流ブランチの設定が存在するものに事前に切り替えておくことで、git pullコマンドで反映されるようになります。

リポジトリ配下のファイルを手動で変更してしまうと pull が行えなくなる欠点がありますが、サーバーを触っている人が 2 人しかいないので、現状は適切に.gitignore を設定しておくことで回避しています。

設定ファイルによって github の webhook を受け取らないようにもできるようにしたので、この設定を適用することで、github と連動させず任意のタイミングでファイルを更新できる個人環境としても使うことができます。

コンテナ

ここまで説明した内容をコンテナ上で動かせるようにします。また、個別の環境についてコンテナを起動する際に手間が極力少なくなるようにします。

各環境は go のソースのみが違う状態なので、事前に go と fresh と git が使える Docker イメージをビルドしておき、動かしたい環境のプログラムが置かれているディレクトリをマウントさせることで動作させます。

まずリスト 6.20 に示すようなビルド用の Dockerfile を用意します。ベースとなるイメージは go の公式のものを使用します。

▼リスト 6.20　Dockerfile

```
FROM golang:1.12

WORKDIR /go/src/bot_app
ENV GO111MODULE=on

EXPOSE 80

RUN apt-get -y update
RUN apt-get -y install git

RUN go get github.com/pilu/fresh

CMD ["fresh"]
```

これをビルドします。

```
$ docker build -t bot_app .
```

これで fresh と git が使える環境のイメージが作成できました。このイメージで立ち上げたコンテナは/go/src/bot_app にて fresh を実行します。このイメージを使って各環境を立ち上げて行きます。

bot 本体のあるディレクトリにリスト 6.21 のような docker-compose のファイルを用意します。

▼リスト 6.21　docker-compose-https-portal.yml

```
version: "3"

services:
        app:
                image: bot_app
                ports:
                        - ${APP_PORT}:80
                volumes:
                        - ${HOME}/.ssh:/root/.ssh
                        - ${HOME}/bot_app_${APP_ENV}:/go/src/bot_app
```

サーバーのホーム配下に bot のソースを clone してきて、bot_app_{環境の名前}にしておきます。イメージの方は/go/src/bot_app 配下にあるソースを実行するように準備されているので、docker-compose 側でアプリのソースのある各環境のディレクトリをマウントしてあげるだけで動作します。

また、git pull が行えるように、ホストマシン側の鍵ファイルが参照できるようにします。

docker-compose には環境変数が使えるので、ホストマシン側のパスを$HOME の値を使って指定します。また、その他に使いたい変数がある場合は docker-compose のコマンドにて指定するか、.env ファイルを置いておくことで指定できます。今回は環境のフォルダ名の一部と、ホストマシン側に割り当てるポート番号を個別設定として指定することにしました。

▼リスト 6.22　.env

```
APP_PORT=8002
APP_ENV=master
```

コンテナ内で git pull の弊害

コンテナ内は root ユーザーで動いているため、git 操作によって新規作成されたファイルの所有者が root になってしまい、コンテナ外の一般ユーザーが git 操作しようとすると失敗するようになってしまいます。
ユーザー名前空間の設定（userns-remap）によって、コンテナ内の root ユーザーをホストマシンの一般ユーザーにマッピングするようにできますが、今回はブランチを追従している環境を一般ユーザーが書き換える必要がないのでこのままにしています。

アクセス管理用コンテナ（リバースプロキシ 2）

リクエストの url を見て各環境に転送するアプリを go を使って実装します。これを動かす Docker イメージは bot 動作用のものを転用し、コンテナ内の bot_app にマウントさせることで動作させます。

アクセスされる url は現状では次の 3 とおりです。

- /github：github からの webhook
- /{環境名}/slashcommand：slack の slashcommand の送信先 url
- /{環境名}/google_oauth：OAuth 認証のリダイレクト url

これらのリクエストについて、環境名とポート番号の対応を書いた json を事前に用意して読み込んでおき、"{環境名}"が設定のあるものだった場合に url を書き換えて該当する bot が動く環境にアクセスできるポートに転送するようにします。

例外的に"/github"の場合のみ、リクエストの header や body をコピーしたリクエストを作成してすべての環境に送信するようにします。bot 側には webhook の内容が正しいか検証する処理があるので、直接"/github"に空のリクエストが送られても pull の処理が走らないようになっています。

リスト 6.23 にリバースプロキシ部分の処理の大枠を示します。なお、このコンテナ内ではホストマシンを示すアドレスを指定していなかったので、デフォルトの 172.17.0.1 が該当のアドレスになります。そのため、別のコンテナへのリクエスト先は"172.17.0.1:{環境のポート番号}"となります。

リクエストを複製する makeProxy 関数や json のデータをもつ変数の実装は省略しています。

▼リスト 6.23　go_proxy.go

```
director := func(r *http.Request) {
    splittedPathArr := strings.Split(r.URL.Path[1:], "/")

    // "/github"だった場合は http.NewRequest ですべての環境に同じリクエストを複製して投げる
    if len(splittedPathArr) > 0 && splittedPathArr[0] == "github" {
        r.URL.Scheme = "http"
        r.URL.Host = fmt.Sprintf("172.17.0.1:%d", rootingConfig["master"])
        for env, port := range rootingConfig {
            if env != "master" {
                httpClient.Do(
                    makeProxy(r, fmt.Sprintf("http://172.17.0.1:%d", port), "/github"))
            }
        }
        return
    }

    // "/{環境名}/*"だった場合は url を"/*"にして投げ直す
    if len(splittedPathArr) > 1 {
        urlEnv := splittedPathArr[0]
        urlAct := splittedPathArr[1]
        if port, ok := rootingConfig[urlEnv]; ok {
            if _, ok2 := actionMap[urlAct]; ok2 {
```

```
                newPath := "/" + strings.Join(splittedPathArr[1:], "/")
                r.URL.Scheme = "http"
                r.URL.Host = fmt.Sprintf("172.17.0.1:%d", port)
                r.URL.Path = newPath
                return
            }
        }
    }

    // どれにも当てはまらないときは 404 にさせる
    r.URL.Scheme = "http"
    r.URL.Host = "localhost:8080"
}

rp := &httputil.ReverseProxy{
    Director:     director,
}
server := http.Server{
    Addr:    ":80",
    Handler: rp,
}
if err := server.ListenAndServe(); err != nil {
    log.Fatal(err.Error())
}
```

go では標準ライブラリでリバースプロキシが使えるようになっており、受け取ったリクエストを一部書き換えて投げ直し、結果をよしなに返してくれます。ただ、handlerFunc を設定する方法では登録していないパスへのリクエストには 404 を返してくれたのですが、リバースプロキシを設定した場合に許可しないパスについて即座に 404 を返すよい方法が見つからなかったので、今回はコンテナ内の 8080 番ポートにアクセスすると必ず 404 を返すようにしておき、条件に合致しない url の場合はそこに転送することにしました。これは、http.HandleFunc によるパスの指定を一切せずにポートを解放することで実現しています。

6.11　サーバーの今後の展開

今回用意した環境で、諸々のリクエストを受け付ける仕組みについては完成だと思います。これを書いている間に、bot の slack からの情報受け取りの仕組みを Real Time Messaging から Slash Command に乗り換えたのですが、事前にここまでの対応をしていたのもあってスムーズに移行できているので、今後新たに受け取り先が増えても対応できると思います。

惜しい点として、各コンテナ間の通信をホストマシンを経由する形になっている点と、そのために環境名とポート番号の対応表が必要な状態になっている点が挙げられます。コンテナ名で接続できるようにするだけでなく、Docker ネットワークの変更を検知することで環境の一覧を自動で認識するようにできると思うので、頑張って実装しようと思います。

また、Google の calendar には変更を通知する webhook も存在するようなので、この情報の受け取って takeichi-a が処理しやすい形に渡す部分まで実装したいと思っています。

6.12 まとめ

takeichi-a:

製作開始する段階で仲間を増やす判断をし二人で作業を始めたのは正解だったと思います。それぞれの得意分野に集中することができ、苦手な作業中によくあるモチベーションの低下もほとんど発生しませんでした。

今回はエンジニア職以外の人に便利さが伝わりづらいできあがりとなりましたが、それでも色々な反応があり、作っていくこととその反応があることという物作りの楽しさを感じました。今後もバージョンアップを重ね、すべての人が予定の調整によるストレスを感じないための bot を完成させたいです。

umezawa-to:

はじめて触る領域について、実際に動かすものを作ることを通してかなり勉強になりました。特に Docker については、そもそもの概念や操作のための様々なコマンドを触る必要があり、bot の環境構築をする以前の段階でかなり苦労しました。ここまで作り上げられるようになれば、別途 Docker を使った環境に出会っても難なく挙動の理解ができると思います。

また、期限のあるタスクとして存在していたこともあり（かなり遅延しましたが...)、完全に個人の趣味としてやるよりも早いペースで理解が進んだと思います。

このようにして用意できた環境を実際に使う人がいるというのは、責任も感じますがそれ以上に「ありがたい」という気持ちの方が強いです。

著者

第 1 章 Shunsuke Ito / @fgshun
Python, Starlette で輝きたい

第 2 章 Shinya Naganuma / @Pctg_x8
トラさんよりはユキヒョウさんのほうがすき

第 3 章 Daisuke Makiuchi / @makki_d
眼鏡っ娘が好きです

第 4 章 黒井春人/ @halt
3D プリンタが好きです

第 5 章 Yoshio HANAWA / @hnw
Puppeteer のノウハウ交換会をしたいです

第 6 章 Toshifumi Umezawa
梅を漬けました

第 6 章 Ayato Takeichi / @take
GO 言語勉強中です。

表紙・ポストカード・ポスター

木
素敵なデザインとイラストに感謝です！！

あべ
vol.2 ぶりに描かせていただきました！ ありがとうございました！

おが

印刷物全般のデザインを担当しました。普段作らないテイストのデザインなので楽しめました！

ヤマウラ

いいものができてうれしいです

あつち

ポストカード描かせていただきました、楽しかったです！

二本足おたまじゃくし

ポストカード描かせていただきました。普段描かないようなイラストを描くことができ楽しかったです。

電子版ダウンロード

http://klabgames.tech.blog.jp.klab.com/archives/tbf07.html

KLab Tech Book Vol. 5

2019 年 9 月 22 日　技術書典 7 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
印刷所　日光企画

KLab Tech Book Vol.5