KLab Tech Book

クラブテックブック

vol.13

1 TCP_NODELAYの効果を確かめる

2 2024年から始めるPython asyncio入門

3 欲しい釣具の入荷状況を生成AIで監視してみた

4 Cloudflare WARP経由で自宅の透過Proxyを使う話

KLab Tech Book Vol. 13

KLab 技術書サークル　著

2024-05-25 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 13 弾です。

KLab では主にモバイルオンラインゲームを開発していますが、KLab Tech Book では社内の有志のエンジニアが業務との関連によらず好きな内容を執筆しています。表紙のデザインも社内のデザイナーの方にご協力いただき、KLab 感溢れる一冊に仕上がっています。

今回は 4 記事を収録しました。それぞれの著者によって、仕様についての丁寧な説明や、実用的な手法の紹介などが行われています。章ごとに内容が独立しているので、気になるものから順に読み進めていただいて問題ありません。

記事で触れられている技術領域は様々ですが、あまり触れたことがない領域であっても、一読していただくことで新しい興味分野の発掘や思いもよらぬ発想の種になるかもしれません。

本書を通して、そんな知的な営みの楽しさを感じていただけたらさいわいです。

梅澤　寿史

お問い合わせ先

本書に関するお問い合わせは tech-book@support.klab.com まで。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第 1 章

TCP_NODELAY の効果を確かめる

Daisuke Makiuchi / @makki_d

KLab では独自のリアルタイム通信基盤「WSNet2」を開発運用し、OSS としても公開しています[*1]。ある日、WSNet2 のサーバーを海外に建て、手元のクライアントからのメッセージの応答時間を調べていたところ、想定より妙に長い時間がかかっていることに気づきました。

WSNet2 はメッセージの送受信に WebSocket を利用しています[*2]。WebSocket は HTTP をベースとしていて、基本的には TCP でパケットを送り合うことになります。そしてこのとき見つけた遅延の原因は TCP の機能にありました。

この章では、遅延の原因となった TCP の機能である **Nagle のアルゴリズム**と `TCP_NODELAY` オプションについて解説し、実際にどのような動きをするのか確認します。加えて、Unity 特有の事情についても解説します。

1.1 TCP の小さなパケット問題

TCP ではパケットが到達することをプロトコル自体で保証しています。送信側はパケットのヘッダにシーケンス番号などの情報を付け加えておき、また受信側は受信したことを ACK というメッセージで送信側に通知します。これらの情報を使って、未到達のパケットを自動で再送するようになっています。

このため、TCP では 1 バイトのデータを送るだけでも TCP と IP のヘッダを合わせて 40 バイト以上送信することになります。このような小さなパケットを多数送るような状況は telnet セッションなどでよく発生し、通信帯域の限られていた 1980 年代には輻輳崩壊の原因になりうるような無視できないオーバーヘッドだったようです。この問題を回避する方法とし

[*1] `https://github.com/KLab/wsnet2`
[*2] WSNet2 の WS は WebSocket の略です

て、RFC 896[*3]が提案されました。

1.2 Nagle のアルゴリズムと TCP_NODELAY

RFC 896 で提案された手法は、送信するデータをある程度バッファリングしてひとつの TCP パケットにまとめることで効率化するというものです。データをまとめるかどうかの決定方法は、RFC の著者の名前をとって Nagle のアルゴリズムと呼ばれています。

Nagle のアルゴリズムでは、次の条件を満たすまでデータをバッファリングして、ひとつのパケットにまとめます。

1. 未送信のデータが最大セグメントサイズ[*4]を超える
2. 未受信の ACK がなくなる

2024 年 05 月 12 日時点の日本語版 Wikipedia[*5]では条件に「タイムアウトになる」が加えられていますが間違いです。元の RFC にはタイマーは不要と明記されていますし[*6]、少なくとも Linux カーネルの実装にはタイムアウトはありません。

さて、WSNet2 で問題になっていたのはサーバーを海外に建てているときでした。物理的な距離によりレイテンシーが高く、ACK が届くのにも時間がかかっていました。

他にも ACK が遅延する要因として、RFC 1122[*7]に記載されている TCP 遅延 ACK という機能もあります。これは ACK の返答を一時的に遅らせ、複数の ACK 応答をまとめて返すことでプロトコルのオーバーヘッドを減らすものです。

これらの要因で ACK が遅延したために、Nagle のアルゴリズムにより、ACK が届くまでの間に送信したメッセージは TCP のレイヤーでバッファリングされてしまいました。このバッファリングされている時間の分、アプリケーションからは応答時間が長くなっているように見えたわけです。

このようなケースに対応するために、Nagle のアルゴリズムを無効にするオプション `TCP_NODELAY` が用意されており、`setsockopt` 関数で設定できます。

[*3] "Congestion Control in IP/TCP Internetworks", J. Nagle, RFC 896, Jan 1984. `https://datatracker.ietf.org/doc/html/rfc896`

[*4] TCP の 1 つのパケットに載せられる最大サイズ

[*5] `https://ja.wikipedia.org/wiki/Nagle%E3%82%A2%E3%83%AB%E3%82%B4%E3%83%AA%E3%82%BA%E3%83%A0`

[*6] 原文: *This inhibition is to be unconditional; no timers, tests for size of data received, or other conditions are required.*

[*7] "Requirements for Internet Hosts -- Communication Layers", Internet Engineering Task Force, R. Braden, Ed., RFC 1122, Oct 1989. `https://datatracker.ietf.org/doc/html/rfc1122`

1.3 実験

それでは実際に Nagle のアルゴリズムの働きと TCP_NODELAY の効果を確認してみましょう。

TCP サーバーとネットワーク遅延設定

まずは Docker を使って単純な TCP サーバーとネットワーク遅延環境を用意します。Docker は Linux なので、tc コマンド（traffic control）で通信遅延のエミュレートが簡単にできます。

最初に TCP サーバー用のコンテナを立ち上げます。ネットワーク遅延を設定するためには、`--privileged` オプションの指定が必要です[*8]。また、名前を `tcpsv` としておきます。

```
docker run -it --name=tcpsv --privileged alpine
```

コンテナが起動したら `tc` コマンドをインストールし、送信パケットを 1 秒遅延させるように設定します。`tc` コマンドで指定するデバイス `eth0` は外部との通信に使われるネットワークインターフェイスです。このコンテナから外部へ送信するパケットは遅延しますが、受信するものは遅延しないことに注意してください。

```
apk add iproute2-tc
tc qdisc add dev eth0 root netem delay 1s
```

TCP サーバーは `nc` コマンド（netcat）を使うことで簡単に用意できます。次のように 5000 番ポートで待ち受けます。

```
nc -l -p 5000
```

[*8] userns-remap を設定している場合は`--userns=host` の指定も必要です。userns-remap については KLabTechBook Vol.11「Docker を使うなら当然 userns-remap してるよね！」をご覧ください。

クライアントの実装

指定サーバーに TCP で 1 文字ずつ送るプログラムを用意しました。このソースコードは GitHub Gist にも置いてあるのでダウンロードしてお使いください[*9]。

リスト 1.1: main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/tcp.h>

int main(int argc, char **argv)
{
        if (argc <= 2) {
                printf("usage: %s <host> <port> [nodelay]\n", argv[0]);
                return 0;
        }

        struct addrinfo hint, *addr;
        memset(&hint, 0, sizeof(hint));
        hint.ai_family = AF_INET;
        hint.ai_socktype = SOCK_STREAM;
        if (getaddrinfo(argv[1], argv[2], &hint, &addr) != 0) {
                perror("getaddrinfo");
                return 1;
        }

        int sock = socket(
                addr->ai_family, addr->ai_socktype, addr->ai_protocol);
        if (sock == -1) {
                perror("socket");
                return 1;
        }

        /* TCP_NODELAYの設定 */
        int n = (argc > 3) ? atoi(argv[3]) : 0;
        if (setsockopt(sock, SOL_TCP, TCP_NODELAY, &n, sizeof(n)) == -1) {
                perror("setsockopt");
                return 1;
        }
```

[*9] https://gist.github.com/makiuchi-d/f748ca25bd4089756faa45fe3af4ced0/raw/main.c

```
        if (connect(sock, addr->ai_addr, addr->ai_addrlen) == -1) {
                perror("connect");
                return 1;
        }

        /* 1文字ずつ100ms間隔で送信 */
        for (int i=0; i<100; i++) {
                char c = '0' + (i % 10);
                write(sock, &c, 1);
                usleep(100000);
        }

        close(sock);
        freeaddrinfo(addr);

        return 0;
}
```

クライアント用のコンテナを立ち上げ、gcc でコンパイルします。サーバーコンテナ tcpsv にアクセスしやすいよう--link も指定しておきます。

```
docker run -it --link=tcpsv alpine

apk add gcc libc-dev
wget https://gist.github.com/makiuchi-d/f748ca25bd4089756faa45fe3af4ced0/raw/mai→
n.c
gcc -o tcpcl main.c
```

ビルドした tcpcl コマンドは引数でサーバーとポートを指定して実行します。サーバーコンテナには tcpsv という名前でアクセスできます。

```
./tcpcl tcpsv 5000
```

サーバーコンテナの画面に 0~9 の数字が順に表示されることが確認できるはずです。クライアントが終了すると nc も停止するので、サーバーコンテナ側で毎回 nc コマンドを実行しなおしてください[*10]。

[*10] nc を終了するためにサーバーコンテナ側で何度かエンターキーを入力する必要があることがあります

通信内容の確認

Linux マシンで Docker を使っている場合は簡単です。ホストの `docker0` インターフェイスを Wireshark[*11]などでキャプチャすることでコンテナ間の通信内容を見ることができます。

`tcpcl` 実行時の通信内容は図 1.1 のようになっています。

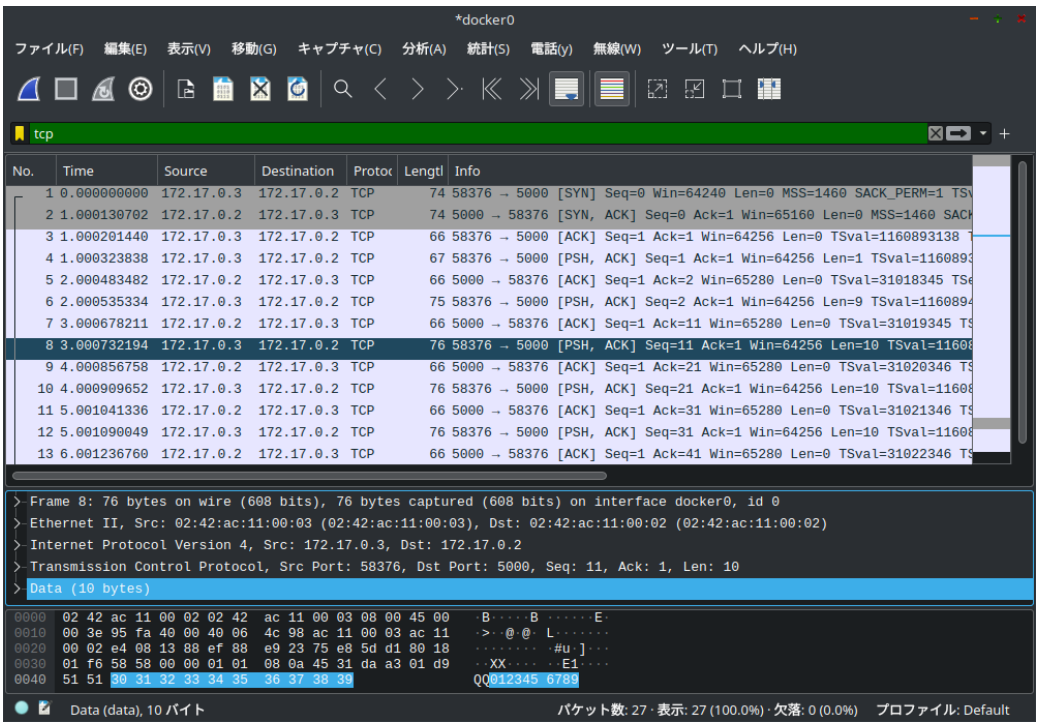

図 1.1: Nagle のアルゴリズムの効果

サーバーからの ACK を受け取ってからデータを送信していて、Data 部が"0123456789" の 10 文字になっているのがわかります。また実行中にサーバーの画面をみていると、約 10 文字ごとに表示が進んでいくのが見て取れると思います。

続いて `TCP_NODELAY` を有効にしてみます。`tcpcl` の 3 番目の引数に 1 を指定すると有効になります。

[*11] `https://www.wireshark.org/`

```
./tcpcl tcpsv 5000 1
```

図 1.2 のように、サーバーからの ACK を待たずに 1 文字ずつ送信しているのが見て取れます。サーバーの画面も 1 文字ずつ順に表示が進んでいくことが分かるはずです。

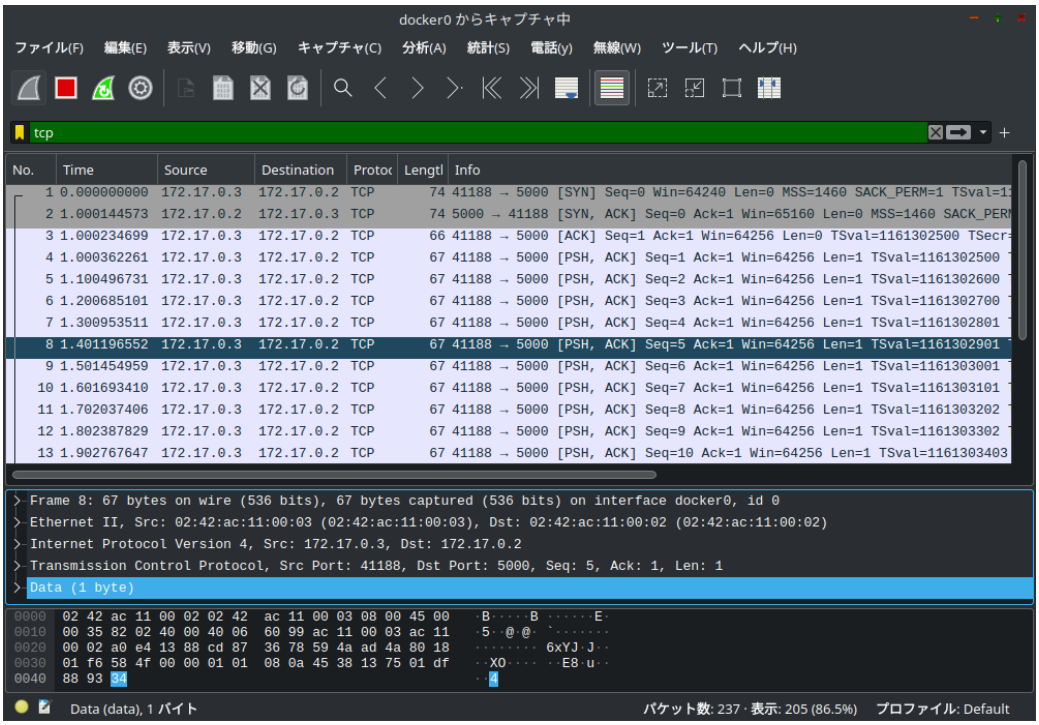

図 1.2: TCP_NODELAY の効果

このように、Nagle のアルゴリズムによって 1 パケットにデータがまとめられている様子や、TCP_NODELAY によってそれが無効になっている様子が確認できました。

1.4 Unity 特有の事情

C#では Socket クラスの NoDelay プロパティによって TCP_NODELAY の有効無効を設定できます。このプロパティをセットすると、内部では最終的に setsockopt 関数がよばれます。

WSNet2 の C#クライアント実装では、標準ライブラリの System.Net.WebSockets.ClientWebSocket を使用しています。現在公式サポートされている Unity の C#ランタイム

は.NET Framework 4.8 相当[*12]で、とても残念なことに、WebSocket 接続時の NoDelay は false になっています。このために、冒頭で言及した応答時間調査のときには Nagle のアルゴリズムが有効になっていて余計な遅延が発生していました。

さらに酷いことに、ClientWebSocket クラスには内部の Socket にアクセスする手段がありません。仕方がないので、WSNet2 では Unity の場合にはリフレクションを使って Socket を取り出し、NoDelay プロパティを設定するようにしました[*13]。

.NET 5 以降では、依然として ClientWebSocket から NoDelay を設定するインターフェイスは存在しないものの、WebSocket 接続時に NoDelay は true に設定されるので、Nagle のアルゴリズムによる遅延の心配はありません。

1.5 まとめ

ここまで、Nagle のアルゴリズムと TCP_NODELAY オプションについて解説し、実際の動作を確認しました。ゲームの協力プレイやオンライン対戦では、パケットをまとめることによる帯域の節約よりもリアルタイム性のほうが大切です。このようなケースでは TCP_NODELAY を設定したほうがよいでしょう。また、想定以上の遅延が見られたときは TCP_NODELAY が設定されているか確認してみてください。

[*12] https://docs.unity3d.com/ja/2023.2/Manual/dotnetProfileSupport.html

[*13] https://github.com/KLab/wsnet2/blob/v0.6.1/wsnet2-unity/Assets/WSNet2/Scripts/Core/Connection.cs#L476-L521

第 2 章

2024 年から始める Python asyncio 入門

Shunsuke Ito / @fgshun

Python の **asyncio** ライブラリ。並行処理のコードを簡潔に記述できる便利なものです。**async def 文**と **await 式**が追加されたのも今は昔、使用されているケースも増え、コード例を探すにも困らなくなってきました。そんな 2024 年現在ならではの、もう不要になった知識を省いたシンプルな紹介ができれば、と思い立ち、 紹介記事を書いてみます。

2.1 Hello asyncio

さっそく、 **asyncio** を使ったコードをみてみましょう。

リスト 2.1: Hello asyncio

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('asyncio!')

if __name__ == '__main__':
    asyncio.run(main())
```

同様の挙動をする asyncio を用いないコードは次のようになるでしょう。

リスト 2.2: asyncio 未使用のコード

```
import time

def main():
    print('Hello ...')
    time.sleep(1)
    print('asyncio!')

if __name__ == '__main__':
    main()
```

異なる点が 3 つあります。

- def の代わりに async def が用いられている
- main が asyncio.run でラップされている
- sleep が非同期のコルーチン関数版になっていて、 await 式とともに使われている

ひとつずつみていきましょう。**async def 文**とは**コルーチン関数**を記述するために用いる文です。この中に書くことができる内容は def 文内に書くことができるもの全般を含みます。たとえば、次の print だけを含む hello コルーチン関数は正当なものです。

リスト 2.3: シンプルなコルーチン関数

```
async def hello():
    print('Hello World')
```

しかし、単に hello() のように実行しても、 print 文が実行されることはありません。次のような警告が表示されるだけとなります。

リスト 2.4: 動作しない例、動作結果

```
/Users/ito-sh/src/gijutsushoten-16/a.py:4: RuntimeWarning: coroutine 'hello' was→
 never awaited
  hello()
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
```

async def で記述したコルーチン関数は通常の関数とは異なり、呼び出すだけでは処理が始まらないことがわかります。コルーチン関数を実行すると**コルーチン**が得られます。コルー

チンとは中断そして再開が可能である処理の集まりとなります。コルーチンを動かすためにはそのための仕組み、イベントループが別途必要となります。これを用意し、コルーチンを実行してくれるのが **asyncio.run** です。

そして、関数内には記述できず、コルーチン関数内にならば記述できるもののひとつが **await 式**となります。await 式にコルーチンを渡した場合、現在実行中のコルーチンは中断されて、渡されたコルーチンが動作開始します。そして動作完了時に中断したコルーチンが再開します。await 式の値は渡されたコルーチンにある return 式の値となります。関数から関数を呼び出して戻り値を受け取るのと同様の記法、同様の流れとなります。

2.2 並行処理ことはじめ

先ほど、 await 式にコルーチンを渡しました。await 式に渡せるものはコルーチンに限りません。正確には、渡せるものは **Awaitable オブジェクト**であり、コルーチンは Awaitable オブジェクトの一種なのです。Awaitable オブジェクトにはコルーチンのほかに、実行中の処理である **Task** が存在します。Task は create_task 関数にコルーチンを渡すことで作成します。渡したコルーチンは即座に実行開始されます。その結果は await することで受け取ります。つまり、結果がまだいらない時点では await をしないことで別のことを行う余地ができます。

たとえば 10 秒かかる処理があったとして、それを 2 つ同時に実行するコードは次のようなものになります。これがもし並行処理が可能な独立した処理、たとえばサーバーに問い合わせて結果の受信を待つようなものであれば、 10 秒かかる処理が 2 つあったとしても 10 秒で終えることができます。

リスト 2.5: タスクを使い並行処理をする例

```
async def heavy_task():
    await asyncio.sleep(10)
    return 'spam'

async def main():
    task0 = asyncio.create_task(heavy_task())  # 処理開始
    task1 = asyncio.create_task(heavy_task())  # もうひとつの処理開始
    await task0  # 終了待ち。結果が不要で待つだけであれば捨ててよし
    ret1 = await task1  # 終了待ち。結果がいる時は代入してとっておく
```

タスクを作る際の注意点として、その参照を保持しなければならないというものがあります。イベントループ側はタスクの弱参照しか持たないため、ガベージコレクションに実行中の

タスクが回収されてしまうということが起こります。タスクを作った側で変数に直接代入して持っておく、コンテナに入れて持っておくなどの対応が必要です。

リスト 2.6: タスクの参照を保持する

```
    # 変数に代入して持っておく例
    task0 = asyncio.create_task(heavy_task())
    # list に入れて持っておく例
    tasks = []
    for _ in range(10):
        tasks.append(asyncio.create_task(heavy_task()))
```

2.3 TaskGroup

Task を複数作成した場合、それらすべての完了を待ちたくなることがあります。先ほどのコードのように await を複数回実行してもよいのですが、ひとかたまりの処理として扱えると便利です。**TaskGroup** がそれを可能にしてくれます。TaskGroup は、脱出時に登録された Task の終了を待つ非同期コンテキストマネージャです。タスクの作成と登録は TaskGroup.create_task で行います。結果は Task の result メソッドから得ることができます。

リスト 2.7: タスクを束ねる例

```
async with asyncio.TaskGroup() as tg:
    task0 = tg.create_task(heavy_task())
    task1 = tg.create_task(heavy_task())
# ここに到達した時点で task0, task1 は終了している
ret0 = task0.result()
ret1 = task1.result()
```

TaskGroup オブジェクトを引き回してコルーチン関数から Task を作成して TaskGroup に加えるような使い方もできます。生成側のコルーチンは生成されたタスクの終了を待ち合わせることなく処理を続行でき、生成されたタスクは async with ブロックを抜ける際に終了待ちが行われることとなります。

リスト 2.8: タスクグループオブジェクトを引き回す

```
async def read_data(tg: asyncio.TaskGroup):
    # データの読み出しのつもり
    await asyncio.sleep(1)
```

```
    value = 42

    # 読み出したデータを書き始める
    tg.create_task(write_data(value))
    # 書き終わるのを待たずに終了

async def write_data(value):
    # データの書き出しのつもり
    await asyncio.sleep(7)
    print(value)

async def main():
    async with asyncio.TaskGroup() as tg:
        tg.create_task(read_data(tg))
        print('waiting')
    # この時点で読み込みも書き出しも完了している
    print('done')
```

■コラム: 非同期コンテキストマネージャ

非同期コンテキストマネージャは前処理を表現する __aenter__ コルーチン関数と後処理を表現する __aexit__ コルーチン関数をあわせ持つオブジェクトです。async with 文とともに用います。このブロックへの突入時に前処理が、脱出時に後処理が行われることとなります。async with と同等のコードを簡単のために例外処理を省いて記述すると次のようになります。

リスト 2.9: async with と同等のコード

```
async with MANAGER as TARGET:
    SUITE
# 上記コードと同等のコード
TARGET = await MANAGER.__aenter__()
SUITE
await MANAGER.__aexit__(None, None, None)
```

実際には SUITE にて例外が発生した時の対応があるため、より複雑なことが行われます。公式ドキュメントの async with 文[*1] を参照してください。

[*1] https://docs.python.org/ja/3/reference/compound_stmts.html#the-async-with-statement

2.4 Queue を用いたデータのやりとり

依存関係をもつタスクを記述するにあたり、間に **asyncio.Queue** を用意するという手法があります。たとえば リスト 2.8 の `read_data` 内には `write_data` の開始が 1 対 1 関係として記述されてしまっています。`Queue` を間に挟むことにより前段の処理に後段の処理の開始を書く必要がなくなる、同時実行数を制御し得るという 2 つの利点が生まれます。

リスト 2.10 は読み込み処理を 5 並列で行い、読み込んだデータの書き込み処理を 2 並列で行う例です。読み込み側は `Queue` にデータを登録します。書き込み側は `while` 無限ループにより `Queue` にデータが届くのを待ち続けます。`main` 側では、 読み込み処理すべてが終了するのを待ち、その後キューが空になるのを待ち、最後に書き込みのループを停止するようになっています。

リスト 2.10: queue 越しにデータをやり取りする

```
async def read_data(que):
    # データの読み出しのつもり
    await asyncio.sleep(1)
    value = 42

    # 読み出したデータの書き込みを予約する
    await que.put(value)

async def write_data(que):
    while True:
        # キューにデータが届くのを待つ
        value = await que.get()
        # データの書き出しのつもり
        await asyncio.sleep(7)
        print(value)
        que.task_done()

async def main():
    que = asyncio.Queue()
    async with asyncio.TaskGroup() as writer_g:
        writers = []
        for i in range(2):
            # 書き込みループを起動
            writer = writer_g.create_task(write_data(que))
            writers.append(writer)
        async with asyncio.TaskGroup() as reader_g:
            for i in range(5):
                reader_g.create_task(read_data(que))
            # 読み込み処理の終了待ち
```

```
        await que.join()  # キューが空になるのを待つ
        for writer in writers:
            writer.cancel()  # 書き込みループを停止
```

2.5 例外処理

実行中のタスクで例外が発生したときは、タスクは即座に終了します。発生した例外はタスクの結果を得ようと await したり result メソッドを実行した際に再送されます。

では、TaskGroup で並行処理中のタスクたちから例外が発生した時、なにが起こり、どのように対処したらよいのでしょうか？ TaskGroup は管理下のタスクの終了を監視しており[*2]、例外で終了したタスクを見つけ次第、管理下の動作中のタスクをキャンセルし始めます。そして、 TaskGroup 下のすべてのタスクが終了すると、それまでに発生していた例外たちが ExceptionGroup 例外にまとめられます。つまり、ひとつの ExceptionGroup 例外が発生することとなります。

そして ExceptionGroup 例外を、そこに含まれる例外たちを拾いあげて解決する機会を得るために **except* 節**があります。リスト 2.11 のように記述します。

リスト 2.11: except* で例外を拾う

```
try:
    async with asyncio.TaskGroup() as writer_g:
        ...
except* ValueError as eg:
    for value_error in eg.exceptions:
        ...
except* Exception as eg:
    for other_error in eg.exceptions:
        ...
```

従来の except 節との違いは、複数の except* 節が処理対象となる場合がありうることです。ただし、同種の例外はひとつに束ねられるので、各節の実行は最大 1 回までです。束ねられた元のエラーたちを得るには ExceptionGroup の exceptions 読み出し専用属性を使います。

[*2] タスクの終了の監視はイベントループにコールバック関数を登録することで行われています。

2.6 タスクのキャンセル

タスクはその動作をキャンセルすることができます。これは、 task の cancel メソッドを呼び出すことで行います。タスクはキャンセルされると、実行中のコルーチン関数内で asyncio.CancelledError が発生することで動作を停止します。この例外は TaskGroup が例外たちを束ねる処理において特別扱いされており、 ExceptionGroup には CancelledError は含まれません。そのため、 except* 節ではなく、タスクたちの結果を取得・処理する際に別の try 文を用いて捕捉することとなります。特に対応することがなければログを残すか単に無視することとなるでしょう。

リスト 2.12 はタスクのキャンセルにも対応したコードの例となります。100 秒かかるタスクと 1 秒かかるタスクを実行し、 100 秒かかるタスクの方をキャンセルしています。大外の try 文、その else 節に到達できた場合にはタスクグループに属するタスクたちは、すべて正常終了したかキャンセルされたこととなります。そのため、 else 節では各タスクたちを個別に、正常終了したのかキャンセルされたのかを判別していくこととなります。もし、別の例外を起こすタスクを加えた場合、その例外が伝搬した時点で他の実行中のタスクはキャンセルされ、else 節には到達せずにこのコードは実行を終了することとなります。

リスト 2.12: タスクのキャンセル

```
async def raise_value_error():
    raise ValueError(42)

async def main():
    try:
        async with asyncio.TaskGroup() as tg:
            tasks = [
                tg.create_task(asyncio.sleep(100)),
                tg.create_task(asyncio.sleep(1)),
                # tg.create_task(raise_value_error()),
                ]
            tasks[0].cancel()
    except* ValueError as eg:
        pass
    else:
        for task in tasks:
            print(task)
            try:
                ret = task.result()
            except asyncio.CancelledError:
                # キャンセルされたタスクに対する処理
                print('Cancelled')
```

```
            else:
                # 完了したタスクに対する処理
                print(f'Done: {ret}')
```

2.7 排他制御

並行処理の際、同時には実行不可能である処理を記述しなければならない場合があります。このための仕組みとして **asyncio.Lock** があります。これはロックを獲得する・解放するという操作ができるオブジェクトです。複数のコルーチンから獲得が行われる際には 2 番手以降のものは解放されるまで待たされることとなります。リスト 2.13 は `create_text`, `clean up` コルーチンの同時実行は認めつつ、同一ファイルへの書き込みは認めない、というコードの例となります。

リスト 2.13: Lock による同時実行の禁止

```
async def write_file(lock):
    text = await create_text()
    async with lock:
        with open('spam.txt', 'a') as f:
            f.write(text)
    await cleanup()

async def main():
    lock = asyncio.Lock()
    async with asybcio.TaskGroup as tg:
        tg.create_task(write_file(lock))
        tg.create_task(write_file(lock))
```

動作を禁止するのではなく同時実行数を制限するという、有限なリソースへの負荷への対策が必要となることもあるでしょう。たとえば、なんらかの外部サービスの API 呼び出しなど、こちらが並行実行数をたやすく増やせる状況でも負荷の問題などでそこまでは同時に実行したくないケースが該当します。

同時実行数を制限する仕組みとして、上限の数だけのタスクを実行しておく方法を 「2.4 Queue を用いたデータのやりとり」 で紹介しましたが、もう一つの方法として **asyncio.Semaphore** を使うという方法もあります。使用方法は `Lock` に似ています。追加点は認める並行数を引数にとるところです。リスト 2.14 は処理を 4 並列までに制限する例となり

ます。

リスト 2.14: semaphore による同時実行数の制限

```
async def call_external_searvice(value, semaphore):
    async with semaphore:
        ...

async def do_something(values):
    semaphore = asyncio.Semaphore(4)
    async with asybcio.TaskGroup as tg:
        for value in values:
            tg.create_task(call_external_service(value, semaphore))
```

2.8 外部プロセス呼び出し

Python を外部プロセス呼び出しのためのツールとして、シェルスクリプト代わりに用いるという機会はありえるでしょう。**asyncio.create_subprocess_exec** はこれを可能としてくれます。subprocess ライブラリとは異なり外部プロセスの完了を待つことなく、 Python 側での処理を行ったり別の外部プロセスを立ち上げることができます。

リスト 2.15: create_subprocess を用いる

```
proc = await asyncio.create_subprocess_exec('echo', 'spam')
# ここで別の処理を行っておくことができる
returncode = await proc.wait()  # 完了待ち
```

外部プロセスと標準入出力を用いてやり取りを行うときは PIPE を用いることを指示した上で communicate を用います。標準入力へデータを渡すには、 communicate の引数を用います。プロセスの終了を待って標準出力と標準エラー出力を受け取るには、 communicate から得られたコルーチンを await します。

リスト 2.16: communicate でやり取りをする

```
data = gzip.compress(b'spamhameggs')
# gzip 圧縮データを同名コマンドの呼び出しで展開
proc = await asyncio.create_subprocess_exec(
    'gzip', '-cd',
```

```
    stdin=asyncio.subprocess.PIPE,
    stdout=asyncio.subprocess.PIPE,
    stderr=asyncio.subprocess.PIPE)
coro = proc.communicate(data)
# ここで別の処理を行っておくことができる
out, err = await coro  # 完了待ち
print(repr(out)) # b'spamhameggs' と出力される
```

2.9 スレッドを新たに作る

並行処理を意識せずに記述された Python コードをなんとか並行に処理したい場合はあるでしょう。**asyncio.to_thread** は処理を別スレッドに逃すことでこれを可能とします。リスト 2.17 は `async` ではない関数を並行に実行して実行時間の短縮をはかる例となります。

リスト 2.17: to_thread を用いた別スレッドでの処理

```
def do_someting():
    time.sleep(1)
    print('spam')
async def main():
    async with asyncio.TaskGroup() as tg:
        tg.create_task(asyncio.to_thread(do_someting))
        tg.create_task(asyncio.to_thread(do_someting))
```

2.10 タスク終了の監視

タスクを複数生成し、そのすべての完了を待ち、どれかひとつでも失敗したら実行中の残りはキャンセルする、という用途には `TaskGroup` をすでに紹介しました。これとは別にタスクたちの完了や例外処理を事細かに制御したいケースはありうるでしょう。この目的には **asyncio.wait** を用います。`wait` は実行完了したタスクと、まだ実行中であるタスクを分けて返してきます。結果もしくは起きた例外を得るには完了したタスクの `result` メソッドを使います。例外は再発生させられるため、これを `try` 文で捕まえることで制御ができます。

リスト 2.18 は 1, 3, 5 秒をかけて成功するタスク 3 つと、 2 秒をかけて `ValueError` を起こすタスクと、 4 秒をかけて `IndexError` を起こすタスクを実行する例となります。タスクのいずれかが完了するたびに確認をとり、正常終了していれば戻り値の出力を、 `ValueError`

が起きていれば無視を、その他のエラーが起きていれば実行中タスクをキャンセルします。 5 つのタスクの結果は終了順にそれぞれ、成功、 ValueError、成功、 IndexError 、キャンセルされた、となります。全体の処理は 4 秒で終わります。

リスト 2.18: wait を用いたタスクの個別制御

```
async def do_something(sec):
    await asyncio.sleep(sec)
    return sec
async def value_error(sec):
    await asyncio.sleep(sec)
    raise ValueError('spam')
async def index_error(sec):
    await asyncio.sleep(sec)
    raise IndexError('spam')

async def main():
    tasks = set()
    for sec in (1, 3, 5):
        tasks.add(asyncio.create_task(do_something(sec)))
    tasks.add(asyncio.create_task(value_error(2)))
    tasks.add(asyncio.create_task(index_error(4)))
    while tasks:
        done_tasks, tasks = await asyncio.wait(
            tasks,
            return_when=asyncio.FIRST_COMPLETED)
        for done_task in done_tasks:
            print(done_task)
            try:
                ret = done_task.result()
            except ValueError:
                pass
            except IndexError:
                for task in tasks:
                    task.cancel()
            else:
                print(ret)
```

2.11 タイムアウト

処理に実行時間の制限を課し、超過した時に処理をキャンセルしたい場合はあるでしょう。**asyncio.timeout** がこれを可能とします。リスト 2.19 のように用います。制限時間を秒数で指定します。時間超過によるキャンセルが行われた時には TimeoutError 例外が発生する

ため、正常終了した場合との区別がつくようになっています。

リスト 2.19: タイムアウト処理

```
try:
    async with asyncio.timeout(7.0):
        ... # 長時間かかる処理
except TimeoutError:
    ... # タイムアウト時の処理
```

2.12 終わりに

asyncio の中から、今日から使える機能を集めて紹介してみました。Python で非同期処理を初めて書く際に、これらの機能を組み合わせるところから始めてみるのがよいというものに厳選してあります。非同期処理自体は複雑なものではあり続けていますが、その難しさを軽減してくれる asyncio をぜひ使ってみてください。

第 3 章

欲しい釣具の入荷状況を生成 AI で監視してみた

Kusunoki Teruhiko

3.1 動機

私は魚釣りが趣味で、以前から欲しい釣り道具があるのですが、販売サイトではいつ見ても在庫切れで困っていました。入荷状況を毎日自分で確認するのもめんどうなので自動で確認[*1]しようと思い立ち、せっかくなので従来のスクレイピングを用いた手法ではなく、最近流行りの生成 AI を利用して「GCP[*2] + ヘッドレス Web ブラウザ[*3] + 生成 AI」でサクッと入荷状況を確認してみようというのが今回のネタです。

3.2 実現したいこと

以下の処理を、GCP の Cloud Run[*4] で定期実行したいです。

1. ヘッドレス Web ブラウザで販売サイトを訪問して、商品ページのスクリーンショットを撮影する
2. 商品ページのスクリーンショットを生成 AI で分析し、在庫の有無を判定する

[*1] サイトによってはボットによるアクセスが禁止されている場合もありますので、試される場合はサイトの利用規約をよくご確認ください。

[*2] Google Cloud Platform

[*3] ヘッドレス Web ブラウザとは、ユーザーが見て操作するような GUI を持たないが、プログラム等からの操作やレンダリングはできるブラウザです。

[*4] GCP の Cloud Run はコンテナをスケーラブルに実行するためのマネージドサービスです。

3. 在庫の有無を何らかの方法で私に通知[*5]する

3.3 やってみよう

今回使用したコードは私の GitHub にて公開していますので参考になれば幸いです。

- `https://github.com/terukusu/sukusho_summary`

今回の環境

- Python: 3.11
- Google Chrome: 125.0.6422.76
- CrhomeDriver[*6]: 125.0.6422.76
- Selenium[*7]: 4.20.0
- Flask: 3.0.3
- OpenAI Python API library: 1.26.0

Docker[*8] イメージの準備

ディレクトリ構成

- `sukusho_summary`
  - `Dockerfile`
  - `main.py`
  - `requirements.txt`
  - `sukusho_summary.py`

[*5] 私は LINE Notify で通知してもらっています。もっと簡単なメールなどの方法でも良いと思います。
[*6] Chrome をプログラムから操作するためのツールです。
[*7] Web ブラウザをプログラムから操作するためのツールです。
[*6] Docker は、アプリケーションを軽量なコンテナとして仮想化し、どこでも一貫して実行できるプラットフォームです。

各ファイルの内容 (ポイントとなる部分のみ掲載)

環境構築のために リスト 3.1 に示すような Dockerfile[*9]を用意します。ここでインストールしている ChromeDriver は Chrome をプログラムから操作するためのツールです。ChromeDriver のバージョンは、使用する Chrome のバージョンに合わせる必要があるので、現在の Stable のバージョンを Chrome for Testing のサイト[*10] で確認し、図 3.1 にあるような linux64 版の ChromeDriver バイナリの URL を Dockerfile に記載します。

リスト 3.1: Dockerfile の例

```
FROM python:3.11-slim

# Chromeのインストール
RUN wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.d→
eb
RUN apt-get update && apt-get install -y -f \
    ./google-chrome-stable_current_amd64.deb \
    && rm -rf /var/lib/apt/lists/*

# ChromeDriverの適切なバージョンを取得してインストール
# https://googlechromelabs.github.io/chrome-for-testing/#stable から Chrome Stab→
le 版に合った「chromedriver」のバイナリのURLを↓ここにコピペする。
RUN wget -q "https://storage.googleapis.com/chrome-for-testing-public/125.0.6422→
.76/linux64/chromedriver-linux64.zip" \
    && unzip chromedriver-linux64.zip \
    && mv chromedriver-linux64/chromedriver /usr/bin/chromedriver \
    && chmod +x /usr/bin/chromedriver
```

[*9] Dockerfile は、Docker コンテナのイメージを構築するための設定ファイルで、ベースイメージ、依存関係、コマンドの実行方法などを指定します。

[*10] https://googlechromelabs.github.io/chrome-for-testing/#stable

Binary	Platform	URL	HTTP status
chrome	linux64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/linux64/chrome-linux64.zip	200
chrome	mac-arm64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/mac-arm64/chrome-mac-arm64.zip	200
chrome	mac-x64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/mac-x64/chrome-mac-x64.zip	200
chrome	win32	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/win32/chrome-win32.zip	200
chrome	win64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/win64/chrome-win64.zip	200
chromedriver	linux64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/linux64/chromedriver-linux64.zip	200
chromedriver	mac-arm64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/mac-arm64/chromedriver-mac-arm64.zip	200
chromedriver	mac-x64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/mac-x64/chromedriver-mac-x64.zip	200
chromedriver	win32	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/win32/chromedriver-win32.zip	200
chromedriver	win64	https://storage.googleapis.com/chrome-for-testing-public/125.0.6422.76/win64/chromedriver-win64.zip	200

図 3.1: ChromeDriver バイナリの URL

リスト 3.2 に示すメインの処理は、生成 AI へのプロンプトや監視したい商品ページの URL、ページ内のどの領域を分析するか、などを `sukusho_summary` モジュール (後述) へ渡し、在庫の有無に応じて通知を送信する (ここではログに出しているだけ) コードとなっています。ここでは記載が省略されていますが、この `main.py` は Web アプリのエントリポイントにもなっています。

リスト 3.2: main.py

```
# 監視したいECサイトのURL
_TARGET_URL = 'https://github.com/terukusu/sukusho_summary/wiki/Sample-EC-Page'

# 監視したい商品の名前など、ページ内の監視したい領域を特定するための文字列。
# サンプルでは「購入」という項目に購入ボタンが配置されているのでその近辺をの領域を監視する→
。
_TARGET_ELEMENT = '購入'

# 商品名。通知メッセージに表示される。
_TARGET_PRODUCT = 'スーパーツレルンダー'

def main():
    url = _TARGET_URL
    element = _TARGET_ELEMENT

    # サイトの内容に合わせてよしなにプロンプトを書く
    prompt = "在庫は有りますか？「かごへ入れる」は在庫ありと言う意味です。「在庫なし」は→
在庫なしという意味です。ただ一言、yes か no で答えてください。"
```

```
    # サイトの内容に合わせて、スクショを取りたいエリアをよしなに設定する

    # 目的の要素が表示されていさえすれば判定できる場合は、マージンの指定は不要
    f = sukusho_summary.StringFinder(element)

    # 撮影エリアを細かく指定したい場合は、margin_top, margin_bottom, margin_left, ma→
rgin_rightを指定する
    # f = sukusho_summary.StringFinder(element, margin_top=1, margin_bottom=120,→
 margin_left=20, margin_right=20)

    s = sukusho_summary.SukushoSummary(url, prompt=prompt, finder=f)
    summary = s.browse_site()

    logging.debug(f'summary: {summary}')

    message = f'在庫があります！！ ヽ(=´▽`=)ﾉ: {_TARGET_PRODUCT}'
    if 'YES' != summary.upper().strip():
        message = f'在庫がありません(T_T): {_TARGET_PRODUCT}'

    # 結果を通知
    send_notification(message)


def send_notification(message):
    # ここでは簡単のためログに出すだけ。LINEなど好きな通知方法によしなに変えればOK
    logging.info(f'message: {message}')
```

メインの処理から呼び出される、sukusho_summary モジュールの処理の流れは、リスト 3.3 に示す通り、次のようになります:

1. Selenium[*11]を用いてヘッドレス Chrome をプログラムから操作して Web ページをブラウズ
2. ページの構造を解析
3. ページ内の撮影領域を算出
4. スクリーンショットを撮影
5. スクリーンショットを生成 AI で分析
6. 分析結果 (AI からの応答) を返す

ページの構造解析と撮影領域の算出の処理は、指定されたページ内の要素が表示されるまでブラウザを適切にスクロールして、指定されたマージンに応じて要素まわりの画像を切り取る

[*11] Web ブラウザをプログラムから操作するためのツールです。

という処理をしています。

リスト 3.3: sukusho_summary.py

```
    def browse_site(self) -> str:
        screenshot_path = None

        try:
            self.trigger_progress('サイトをブラウズします...')
            self.driver.get(self.url)
            self.driver.execute_script(f"document.body.style.zoom = '{self.zoom}→
'")

            self.trigger_progress('サイトを解析します...')
            element = self._scroll_to_element()

            self.trigger_progress('撮影領域を検出します...')
            crop_area = self._determine_crop_area(element)

            self.trigger_progress('スクリーンショットを撮影します...')
            screenshot_path = self._take_screenshot(crop_area)

            logging.info(f'screenshot_path: {screenshot_path}')

            self.trigger_progress('スクリーンショットをAIで処理します...')
            result = self._process_screenshot(screenshot_path)
            return result
        finally:
            self.driver.quit()

    def _process_screenshot(self, screenshot_path: str) -> str:
        with open(screenshot_path, 'rb') as f:
            screenshot_data = f.read()

        result = openai_chat(self.prompt, images=[('image/png', screenshot_data)→
])

        return result
```

注目してほしいのは撮影したスクリーンショットを AI で分析する部分で、リスト 3.4 に示す通り、スクリーンショットの PNG 画像をデータ URI スキームの形にしてプロンプトと一緒に AI へ渡しています。

リスト 3.4: sukusho_summary.py の中の AI でスクリーンショットを分析する部分

```
def get_openai_client() -> OpenAI:
    client = OpenAI(
        api_key=_OPENAI_API_KEY,
    )

    return client


def openai_chat(prompt: str, *args, images: list[(str, bytes)] = None):
    """
    OpenAI APIを使用してテキスト生成を行う関数。

    Args:
        prompt (str): ユーザーからのプロンプト。
        *args: その他の引数（現在使用されていない）。
        images (list[(str, bytes)], optional): MIMEタイプと画像のバイナリデータのタ→
プルからなるリスト。

    Returns:
        str: OpenAI APIからの生成されたテキスト。
    """

    client = get_openai_client()

    messages = [
        {
            'role': 'user',
            'content': [
                {
                    'type': 'text',
                    'text': prompt
                }
            ]
        }
    ]

    if images is not None:
        for image in images:
            mime_type, image_bin = image
            image_b64 = base64.b64encode(image_bin).decode('ascii')
            messages[0]['content'].append({
                'type': 'image_url',
                'image_url': {
                    'url': f'data:{mime_type};base64,{image_b64}'
                }
            })
```

```
    kwargs = {
        'model': _OPENAI_MODEL_NAME,
        'messages': messages
    }

    # OpenAI APIで文書生成
    result = client.chat.completions.create(**kwargs)
    return_result = result.choices[0].message.content

    return return_result
```

ローカルの Docker 環境でビルド & 実行

あとで実際に動く様子は見ていきますが、先に手順だけさっと説明します。

以下のように docker build コマンドで Docker イメージをビルドしてから、docker run コマンドで Docker コンテナを起動します。「＜YOUR OPENAI API KEY＞」の部分は OpenAI の API にアクセスするための API キーに差し替えます。キーを持っていない場合は、OpenAI API のサイト[*12]で登録すれば発行できます[*13]。このアプリは Web アプリとして動くように実装してあるので、ローカルのポート 8080 でアクセスできるようにしておきます。

```
$ docker build -t sukusho_summary .
$ docker run -e OPENAI_API_KEY="＜YOUR OPENAI API KEY＞" -e PORT=8080 -p 8080:80→
80 sukusho_summary
```

ローカルの Docker 環境で入荷状況の確認の処理を走らせるには以下のように Dcoker コンテナ上のウェブアプリへアクセスします。

```
$ curl -X POST "http://localhost:8080/"
```

すると、リスト 3.5 に示すように、在庫の有無が docker run の出力として報告されます。

[*12] https://platform.openai.com/overview

[*13] OpenAI API の利用は有料です。料金については OpenAI API のサイトをご確認ください。

リスト 3.5: docker run の出力例

```
root INFO screenshot_path: /tmp/tmpf4zyr71b.png
root INFO message: 在庫がありません(T_T): ＜商品名＞
```

どんなスクリーンショットが撮影されたのかを確かめたい場合は、以下のように docker cp コマンドで、screenshot_path の画像ファイルをコンテナから取得して表示します。

```
$ docker cp ＜コンテナID＞:/tmp/tmpf4zyr71b.png foo.png
```

それでは具体的な動作の様子を見ていきましょう。今回、私の GitHub Wiki に図 3.2 のような EC サイトに見立てたページ[*14]を用意しました。ここに掲載するコードでは、そのページを分析して入荷状況を判定するようにしています。

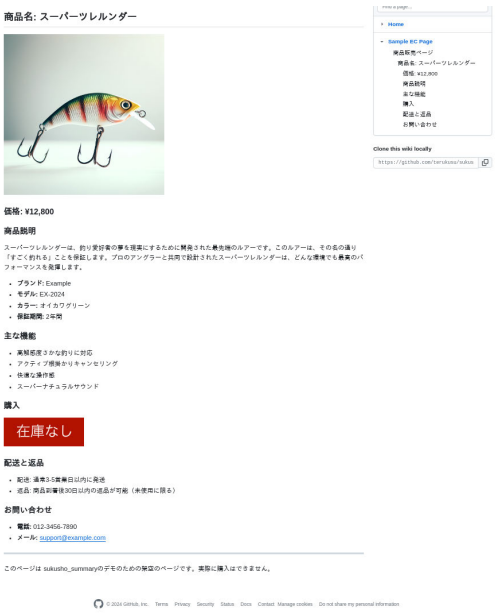

図 3.2: サンプル EC サイト

[*14] https://github.com/terukusu/sukusho_summary/wiki/Sample-EC-Page

まずは、サンプル EC サイトのページに「在庫なし」のボタンを表示させて、分析した場合の例です。撮影されたスクリーンショットは 図 3.3 の通りでした。ページ上部の商品写真が見切れていることからもわかるように、リスト 3.2 の冒頭で指定していた「購入」要素が画面に収まるまで適切にスクロールしてからスクリーンショットが撮影されています。

図 3.3: 在庫がないときのスクリーンショット

このスクリーンショットを AI で分析した結果、図 3.4 のように「在庫がありません (T_T): スーパーツレルンダー」と言われました。在庫が無いのは残念ですが、在庫の有無の判定は成功しましたね。

```
2024-05-23 12:36:42,999 root INFO dpr: 1
2024-05-23 12:36:43,077 root INFO scroll_x: 352.0, scroll_y: 1351.0
2024-05-23 12:36:43,249 root INFO screenshot_path: /tmp/tmpu8_u_4st.png
2024-05-23 12:36:47,352 httpx INFO HTTP Request: POST https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"
2024-05-23 12:36:47,421 root INFO message: 在庫がありません(T_T): スーパーツレルンダー
2024-05-23 12:36:47,423 werkzeug INFO 172.17.0.1 - - [23/May/2024 12:36:47] "POST / HTTP/1.1" 200 -
```

図 3.4: 在庫がないときの docker run 出力

次に、サンプル EC サイトのページに在庫があるときに表示される「かごへ入れる」ボタンを表示させて分析した場合の例です。撮影されたスクリーンショットは 図 3.5 の通りでした。

図 3.5: 在庫が有るときのスクリーンショット

このスクリーンショットを AI で分析した結果、図 3.6 のように「在庫があります！！ ヽ(=´▽`=)ﾉ: スーパーツレルンダー」との判定が！ ボタンの意味を解釈して、入荷の検知に成功しました！

```
2024-05-23 12:38:52,994 root INFO scroll_x: 352.0, scroll_y: 1351.0
2024-05-23 12:38:53,166 root INFO screenshot_path: /tmp/tmpveig_noo.png
2024-05-23 12:38:56,655 httpx INFO HTTP Request: POST https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"
2024-05-23 12:38:56,729 root INFO message: 在庫があります！！ヽ(=´▽`=)ﾉ: スーパーツレルンダー
2024-05-23 12:38:56,731 werkzeug INFO 172.17.0.1 - - [23/May/2024 12:38:56] "POST / HTTP/1.1" 200 -
```

図 3.6: 在庫が有るときの docker run 出力

では最後に、ページの構造が複雑であったり、複数商品が 1 ページに列挙されていて、ページ全体を AI で分析すると在庫の有無の判定に失敗してしまう場合に、スクリーンショットの撮影領域を詳細に指定する例を見てみましょう。

リスト 3.2 の中ではコメントアウトされていた、リスト 3.6 に示す部分をアンコメントして、ページ内の「購入」要素の近傍のみを撮影するように、要素に対する上下左右のマージンを詳細に指定します。

リスト 3.6: main.py で詳細な撮影領域を指定する例

```
    # 目的の要素が表示さえされていれば判定できる場合は、マージン等の細かい撮影領域の指定は→
不要
    # f = sukusho_summary.StringFinder(element) # ← この行をコメントアウト

    # 撮影領域を細かく指定したい場合はマージンを詳細にを指定する
    f = sukusho_summary.StringFinder(element, margin_top=1, margin_bottom=120, m→
argin_left=20, margin_right=20) # ← この行をアンコメント
```

撮影されたスクリーンショットは 図 3.7 の通りでした。「購入」要素の近傍のみ撮影されていることがわかります。

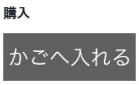

図 3.7: 「購入」要素の近傍のみ撮影

このスクリーンショットを AI で分析した結果、図 3.8 のように「在庫があります！！ ヽ(=´▽`=)ノ: スーパーツレルンダー」と在庫の有無の判定に成功しています。複雑なページでは、手間はかかりますがこのように注目すべき領域を詳細に指定してスクリーンショットを撮影したほうが、在庫の有無を判定する精度が高くなります。

```
2024-05-23 14:49:44,603 root INFO scroll_x: 332.0, scroll_y: 1350.0
2024-05-23 14:49:44,986 root INFO screenshot_path: /tmp/tmpw142puhh.png
2024-05-23 14:49:48,933 httpx INFO HTTP Request: POST https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"
2024-05-23 14:49:49,002 root INFO message: 在庫があります！！ヽ(=´▽`=)ノ: スーパーツレルンダー
2024-05-23 14:49:49,004 werkzeug INFO 172.17.0.1 - - [23/May/2024 14:49:49] "POST / HTTP/1.1" 200 -
```

図 3.8: 「購入」要素の近傍のみの場合の docker run 出力

Cloud Run で入荷状況の監視を動かす場合

ここまでできれば、詳細は割愛しますが、以下の手順で Cloud Run で定期的に入荷状況の確認をすることが可能です。

1. `sukusho_summary` の `Dockerfile` を使用して Cloud Run のサービスを作成する
2. Cloud Scheduler で、1 日 1 回「1」で設定した Cloud Run へアクセスするように設定する

上記の手順に相当する Google Cloud 公式マニュアル は以下の 3 つです。

- コンテナをビルドする
  - `https://cloud.google.com/run/docs/building/containers`
- Cloud Run へのデプロイ
  - `https://cloud.google.com/run/docs/deploying`
- スケジュールに従ってサービスを実行する
  - `https://cloud.google.com/run/docs/triggering/using-scheduler`

3.4 従来のスクレイピングを用いた手法との比較

メリット

- シンプルでわかりやすいページだと、判定のしかたを生成 AI に日本語で指示するだけで判定ができるので楽ちん。従来のスクレイピングによる手法のようにめんどうな Web ページの解析を行う必要はありません
- Web ページの内部的な構成 (HTML 構造など) が変わっても見た目がそれほど変わっていなければ、見た目で判断する AI は影響を受けません
- HTML の文字列そのものを AI で分析する場合と比べると、見た目からしか意味がわからないページでも分析できます。HTML 中の文字列や画像ファイル名からでは意味がわからない場合でも、スクリーンショットを分析する場合はレンダリング後の見た目を扱うので分析が可能となります
- ページ全体のスクリーンショットを撮影して、商品についてどんな情報が書かれているかを AI に説明してもらうといった、スクレイピングでは難しかったようなページ全体の分析もできます

デメリット

- 複雑でわかりにくいページだと、Web ページ中のどの領域に注目するか (どの部分のスクリーンショットを撮って生成 AI に渡すか) を、文字列検索、HTML 要素の `id` 属性での検索など、従来のスクレイピングによる手法と同程度の Web ページ解析の手間がかかります
- 完璧な精度は期待できません。「在庫があります！」と言われて見に行ったら、本当は在庫がないということが稀に起きます。とはいえ、シンプルなページでも複雑なページでも調整次第で 95% 以上の精度にはなります

3.5 まとめと展望

シンプルなページだと HTML 等の Web ページの知識が無くても、人に指示をするかのように生成 AI に判定ロジックを指示するだけで在庫の有無を判定できて便利でした。しかしながら、完璧な精度があるわけではないので 100% の精度が求められるような判定には向かないと思います。

展望としては、Web サイトの自動訪問＋スクリーンショット撮影＋生成 AI による分析というのは、Web 開発のテストや Web デザインの研究など、実は応用範囲が結構あるのではないかと感じました。

今後もこの技術のさらなる応用を模索しながら、私が欲しい釣り道具の「在庫があります！」という通知が来る日を待ちたいと思います。

第 4 章

Cloudflare WARP 経由で自宅の透過 Proxy を使う話

Yoshio HANAWA / @hnw

4.1 はじめに

筆者はスマホから自宅に VPN 接続する環境を作り、スマホアプリから開発者向けプロキシサーバを使えるようにしてみました。そんなことをして何が嬉しいの？ と思われるかもしれませんが、筆者はここ 1 ヶ月頻繁に利用しています。本稿ではその詳細を紹介します。

4.2 開発者向けプロキシサーバーの活用

一般的にプロキシサーバーと言えば、ネットワーク上の通信を中継し、ユーザーのリクエストを代理でインターネットに送信するようなサーバーです。通常のプロキシサーバは個人が日常的に使うようなものではありませんから、筆者がプロキシにこだわっているのが不思議に思えるかもしれません。

筆者が使いたいのは通常のプロキシではなく、Web アプリのデバッグやセキュリティ監査に特化した開発者向けプロキシサーバーです。こうしたプロキシの有名どころとしては Burp Suite[*1]や Fiddler[*2]が挙げられますが、筆者は Charles[*3]を愛用しています。Charles は Java 製の有償プロダクトで、自宅のサブ機の MacBook 上で常時起動しています。

Charles では下記のようなことが可能です。

[*1] https://portswigger.net/burp
[*2] https://www.telerik.com/fiddler
[*3] https://www.charlesproxy.com/

- HTTPS 通信を復号して内容を確認する。
- 特定のリクエストを編集して送信する。
- サーバーからのレスポンスを編集してクライアントに返す。

リクエスト編集機能は SQL インジェクションなどセキュリティホールを突くのにも利用できます。電子計算機損壊等業務妨害罪などに問われる可能性もありますから、自分が管理していない外部サーバに対して利用すべきではありませんし、本稿でも触れません。

今回はレスポンスデータを書き換える話題を紹介します。

レスポンスデータ書き換えで何ができるか

Web アプリやスマートフォンアプリの開発者がセキュリティ対策を考えるとき、ユーザーから渡されるデータは信用せず、サーバ側でのバリデーションをしっかりやるのが基本戦略になります。逆に考えると、理想状態であればユーザー側のデータであるレスポンスデータを改ざんされても何も困らないはずです。

しかし、サーバ側のセキュリティ対策が堅牢であったとしても、レスポンスデータの書き換えでユーザーが得する状況がありえます。

たとえば EC アプリで安売り商品が 5 月 6 日の 17 時から公開される場合を考えてみましょう。商品一覧を JSON で受け渡ししている場合に、リスト 4.1 のような JSON プロパティが見つかったとします。

リスト 4.1: 日時を含む JSON プロパティの例

```
"DISPLAY_START_AT":"202405061700",
```

この JSON プロパティを元に 17 時からアプリ上に安売り商品を表示している場合、これを書き換えればアプリ上で安売り商品が他人より早く表示されることになります。このような場合に、Charles を利用していれば図 4.1 のような設定でレスポンスを書き換えることが可能です。

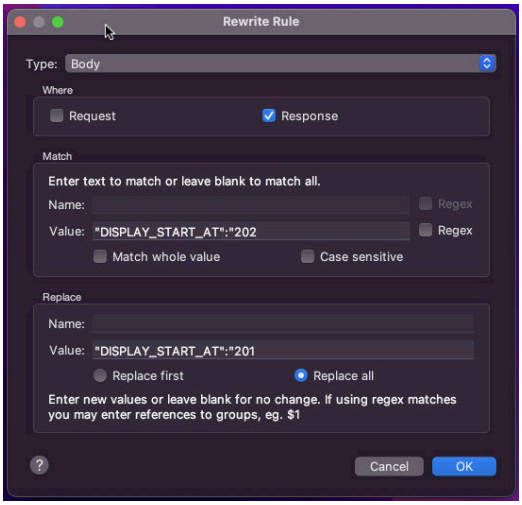

図 4.1: Charles によるレスポンス書き換えの例

もちろん、サーバ側で安売り開始時間のチェックを行っていれば 17 時より前に購入することはできませんが、アプリ上の UI 操作を 17 時以前に済ませておくことで他のライバルより数秒先行できてしまいます。

このように、開発者向けプロキシを使うことで有利不利が生まれる可能性は否定できません。セキュリティ対策を考える側の人間としても、このような環境を知っておくことは価値があるはずです。

4.3 iOS アプリでのプロキシサーバー設定手順

今回プロキシサーバを利用したい環境は iOS アプリですので、iOS 端末のプロキシ設定手順を紹介します。

Wi-Fi のプロキシ設定

iOS 端末でプロキシサーバーを設定するには、以下の手順を実行します。

1. 設定アプリを開く。
2. Wi-Fi を選択し、接続しているネットワークをタップする。
3. ネットワークの詳細画面の「プロキシを構成」をタップする。
4. HTTP プロキシを手動に設定し、プロキシサーバーの IP アドレスとポート番号を入力する。

これにより、Wi-Fi 経由の全通信がプロキシサーバーを経由するようになります。

CA 証明書のインストール

HTTPS 通信を Charles でプロキシする場合、以下の手順で CA 証明書をインストールする必要があります。

1. iOS 端末から Wi-Fi 経由で Charles を利用している状態にする。
2. http://chls.pro/ssl にアクセスし、Charles の CA 証明書を iOS 端末にダウンロードする。
3. 一般 > VPN とデバイス管理 を選択し、ダウンロードした証明書をインストールする。
4. 一般 > 情報 > 証明書信頼設定 を選択し、「ルート証明書を全面的に信頼」で Charles の証明書を有効にする。

これにより、iOS 端末はプロキシサーバーが提供する証明書を信頼し、HTTPS 通信の監視・編集が可能となります。

■コラム: Android だと同じ環境は作れない

本稿を読んで Android アプリでもプロキシを使いたい！ と思われる方がいるかもしれませんが、あまりお勧めできません。

というのも、Android の証明書まわりはセキュリティが厳しくなっており、アプリで独自の CA 証明書を利用するにはアプリ同梱の XML ファイルで設定する必要があるのです[*4]。ビルド権限がないアプリで使いたい場合は apk の展開・再作成が必要になってしまうので、常用には向かないように思います。

一般的に Android の方が開発者にとって利便性が高い傾向にあると思うのですが、プロキシの利用に限っては iOS の方が便利だと言えそうです。こんなこともありますから、iOS 端末と Android 端末の両方を持っておいた方が良いかもしれませんね。

[*4] Android 7 以降では `manifest.xml` で `trust-anchors` の設定が必要です。

4.4 Cloudflare Zero Trust 経由で自宅プロキシサーバーを使う

ようやく本稿の本題です。これまでの設定により自宅でスマホアプリから Charles を使えるようになったわけですが、外出時にも使いたい！ と考えるようになりました。

そこで Cloudflare Zero Trust で自宅の VPN アクセス環境を整え、iPhone に Cloudflare WARP アプリをインストールして外出時でも自宅のプロキシを使えるようにしました。その詳細を紹介します。

Cloudflare Zero Trust とは

Cloudflare Zero Trust は Cloudflare 社が提供するゼロトラストソリューションで、単なる VPN 以上の機能を提供しています。今回の用途にはオーバースペックとも言えるのですが、接続用スマホアプリが提供されていること、セットアップが簡単であること、個人であれば無料[*5]で使えることから、今回は自宅への VPN 環境を構築するのに利用してみました。

Cloudflare Zero Trust のセットアップ

まずは、Cloudflare Zero Trust の基本設定をします。

1. （もし持っていなければ）Cloudflare のアカウントを作成する。
2. Cloudflare のマネジメントコンソールから「Zero Trust」を選び、チーム名を決める。Cloudflare WARP でアクセスするときに必要になりますので、自分のハンドル名などわかりやすい文字列にするのが良いでしょう。
3. Cloudflare のマネジメントコンソールから「Zero Trust」「Settings」「WARP Client」を選び、Device enrollment permissions の「Manage」ボタンを押す。
4. 自分や家族のアクセスのみ許すような設定を行う。筆者は自己所有ドメインのメールアドレスのみ許可しました。
5. Authentication タブから自分が使いたい ID プロバイダを選ぶ。筆者は自己所有ドメインに紐づいた Google Workspace の認証のみ許可しました。

この状態では Cloudflare WARP で接続しても全ての通信が Cloudflare 経由になるだけで、あまり嬉しくありません。これをベースに他の設定を足していきます。

[*5] 50 ユーザーまで無料なので、スタートアップ企業でも利用可能です。

Cloudflare Zero Trust の VPN としての利用

Cloudflare Zero Trust を VPN として使うためには、VPN に参加させたいネットワークのマシンからトンネルを作る必要があります。作業としては下記の手順のようになります。

1. Cloudflare のマネジメントコンソールから「Zero Trust」「Networks」「Tunnels」を選び、「Create a tunnel」ボタンを押す。
2. トンネル名の入力を求められるので、cloudflared をインストールする自宅のマシン名（例：pi4）をトンネル名として指定し、「Save tunnel」ボタンを押す。
3. cloudflared のインストールを求められるので、適切な OS とアーキテクチャを選び、ダウンロードしたバイナリを自宅のマシンにインストールして「Next」ボタンを押す。筆者は常時起動している Raspberry Pi 上で動かすことにしました。
4.「Route Traffic」画面で Private Networks を選び、自宅のネットワークの CIDR（例：192.0.2.0/24）を入力して「Save tunnel」ボタンを押す。
5. Zero Trust の「Settings」「WARP Client」「Device settings」から「configure」を選び、Split Tunnels の「Manage」を押して、先ほど設定した CIDR が含まれるネットワーク（例：192.168.0.0/16）を除外する。

この設定により、自宅外から Cloudflare WARP で接続すれば自宅ネットワーク内のマシンに IP アドレスでアクセスできるようになります。これはまさに VPN と言えるでしょう。

特定ホストへのアクセスを透過プロキシ経由にする

これまでの設定により家の外からプロキシサーバに IP アドレスで接続できるようになりましたが、この状況でプロキシを設定する機能は iOS に存在しません[*6]。そこで、Cloudflare の設定でプロキシサーバを使えるようにします。

Cloudflare Zero Trust では DNS 応答を差し替えられるので、特定のホストへの DNS 問い合わせについて透過プロキシの IP アドレスを返すようにして、透過プロキシ経由で実際のホストにアクセスするようにします。手順は下記の通りです。

1. Cloudflare のマネジメントコンソールから「Zero Trust」「Gateway」「Firewall Policies」を選び、「DNS Policies」「Add a policy」ボタンを押す。
2. プロキシ経由にするホスト名の AAAA レコードを返さないようにする。（図 4.2）
3. 同様に A レコードを自宅の透過プロキシあてにする。（図 4.3）

[*6] 自宅外でも Wi-Fi 環境なら設定できるかもしれませんが、あくまでモバイル回線で使いたいので他の方法を選びました。

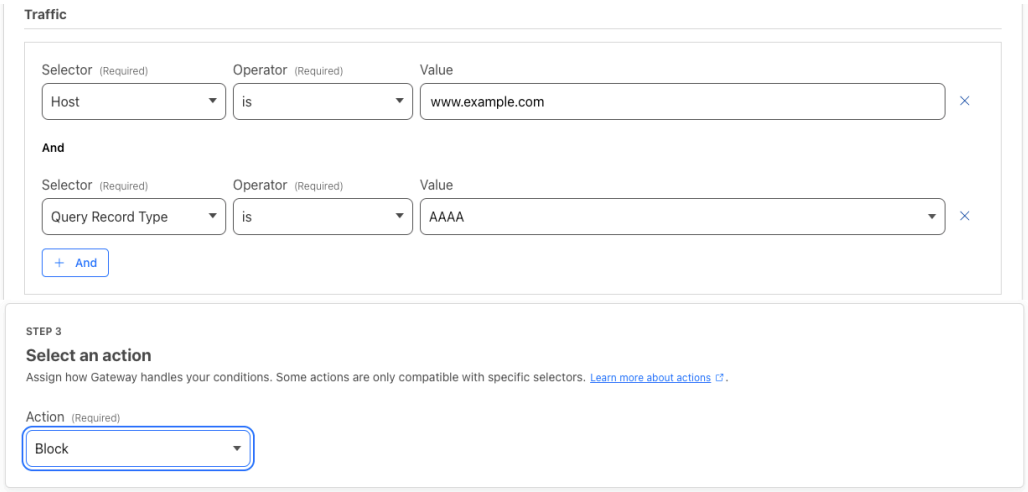

図 4.2: 特定ホスト名の AAAA レコードを返さないようにする

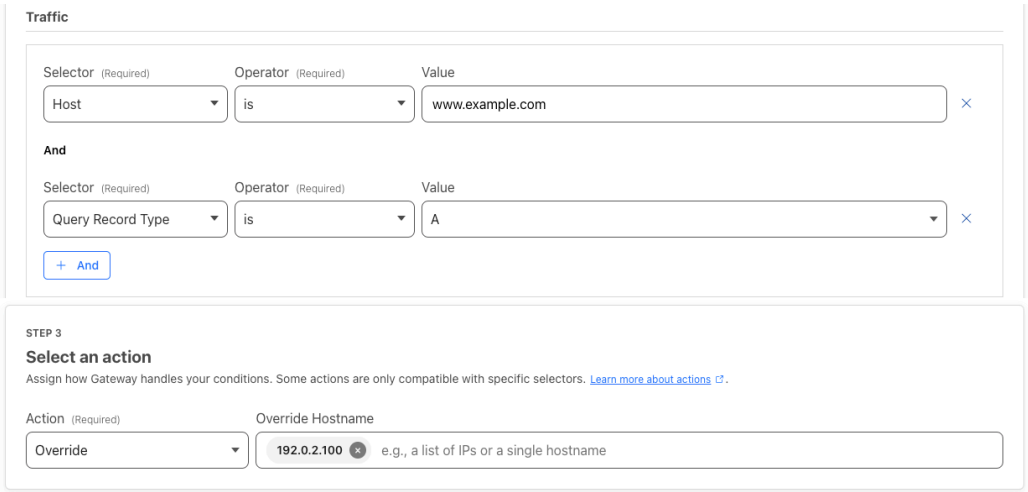

図 4.3: 特定ホスト名の A レコードを差し替える

この設定により特定ホストあての通信のみプロキシ経由で HTTPS 通信を監視・編集することができ、それ以外の通信はプロキシ経由せず直接インターネットに出ていきます。

「4.3 iOS アプリでのプロキシサーバー設定手順」で紹介した Wi-Fi 環境ではすべての通信がプロキシ経由になるため、セキュリティレベルが高いアプリでエラーが出てマトモに使えな

いことがあります[*7]。Cloudflare Zero Trust 環境であればエラーが出るような通信は直接アクセスにすることができますから、Wi-Fi 環境よりも便利だと言えるでしょう。

Charles を透過プロキシ化する

これまで透過プロキシが存在する前提で説明してきましたが、実は Charles は透過プロキシに対応していません。正確にいうと、透過プロキシは HTTP のみ対応で、HTTPS の透過プロキシを提供していないのです。この問題を解決するため、通常のフォワードプロキシを HTTPS 対応の透過プロキシに変換するようなサーバが必要です。

そんな都合のいい実装あるわけないでしょ…と思ったのですが、SNI[*8]対応の透過プロキシサーバー sniproxy[*9]を見つけました。DNS までプロキシしてくれるなど若干オーバースペックではありますが、今回の用途にピッタリのツールです。これは以下のように利用できます。

```
sniproxy --dns-redirect-ipv4-to=8.8.8.8 --forward-proxy=http://192.0.2.10:8888
```

引数のうち DNS については使わないので何を書いても構いません。proxy については MacBook の Charles を指すように指定します。これを Raspberry Pi 上で動かせば透過プロキシの完成というわけです。

4.5 まとめ

かなりニッチな要求を Cloudflare Zero Trust で実現した話を紹介しました。Cloudflare Zero Trust はホントにタダで使っていいんですか？ というくらい機能が充実しているので、皆さんも自宅に導入してみてはいかがでしょうか。

[*7] App Store アプリや一部の銀行アプリでは SSL pinning を採用しているため、想定と異なる CA 証明書で通信しているとエラーが出てしまうのです。

[*8] TLS 通信でバーチャルホスティングを実現するための TLS の仕様。TLS を復号せずにホスト名がわかる。

[*9] `https://github.com/ameshkov/sniproxy`。機能が異なる同名の実装がいくつかあることに注意。

執筆者・スタッフコメント

第 1 章 Daisuke Makiuchi / @makki_d

眼鏡っ娘が好きです

第 2 章 Shunsuke Ito / @fgshun

Python をグルー言語として用いる際の強い味方、 asyncio について紹介したいと思い立ち。外部プロセスの非同期呼び出しが簡単にできて便利です。

第 3 章 Teruhiko KUSUNOKI

早く AI から在庫発見の報告が来ないかな。

第 4 章 Yoshio HANAWA / @hnw

Charles のライセンスは 3 年前のブラックフライデーセールで買いました

企画進行・イラスト・デザイン

UME

代表者として企画進行を担当しました。電子工作の記事も書こうとしましたが動作検証が終わらず...

たのりん

表紙周りのデザインをしました。世界観たっぷりなイラストの邪魔になりすぎず適度に目を引くことを目指しました！

うすい

イラストを担当いたしました。技術書展の知のイメージから発想してかいています！

既刊・電子版ダウンロード

https://www.klab.com/jp/blog/tech/2024/tbf16.html

KLab Tech Book Vol. 13

2024 年 5 月 25 日　技術書典 16 版（1.0）
著　者　KLab 技術書サークル
編　集　梅澤 寿史、牧内 大輔
発行所　KLab 技術書サークル
印刷所　日光企画