ネットワーク及び演習

第 5 回 HTTP プロトコルと httpServer.py

佐野 彰

本日の到達目標

HTTP プロトコル

- URL のどの部分が通信の何を決めているかを説明できる
- HTTP リクエスト／レスポンスの形式（1 行目・ヘッダ・空行・本文）を書ける
- 主なメソッドとステータスコードの意味を説明できる

httpServer.py

- `echoServer.py` に何を加えれば Web サーバになるかを説明できる
- `with` によるファイル処理と、`try` 〜 `except` による例外処理を書ける

> 今日は、ブラウザからアクセスできる**本物の Web サーバ**を動かす。プログラムは 40 行ほどの Python。

1. HTTP プロトコル

Web のやり取りで実際に送受信されているテキスト

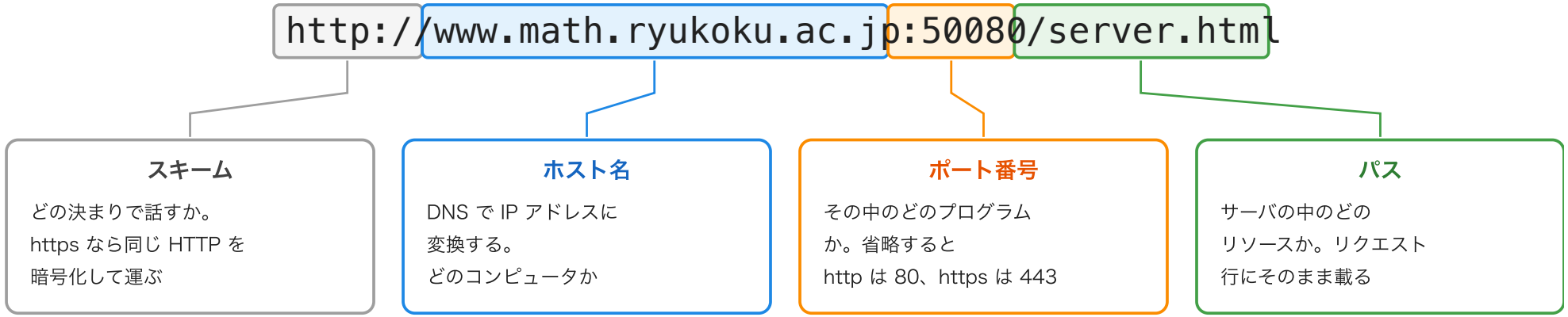
HTTP — Web ページを転送するためのプロトコル

- HyperText Transfer Protocol。アプリケーション層のプロトコル
- 既定のポートは **80**（HTTP）と **443**（HTTPS）

版	年	現在の RFC	特徴
HTTP/0.9	1991	—	`GET` だけ。ヘッダも無い
HTTP/1.0	1996	RFC 1945	ヘッダとステータスコードが入る
HTTP/1.1	1997	RFC 9112	接続を使い回せる。この科目で書くのはこの版
HTTP/2	2015	RFC 9113	同じ意味のまま、バイナリ形式で多重化
HTTP/3	2022	RFC 9114	TCP の代わりに QUIC（UDP ベース）を使う

- 1.1 以降も、やり取りの意味は変わっていない。変わったのは転送の方式だけ

URL — 通信の宛先を 1 行で表す



ブラウザはこの 1 行を分解して、DNS を引き、そのポートへ TCP でつなぎ、GET /server.html HTTP/1.1 を送る。

実際に送受信されるデータ

ブラウザ → httpServer.py（リクエスト）

```
GET /server.html HTTP/1.1
Host: localhost:50080
User-Agent: Mozilla/5.0 (...)
（空行）
```

① リクエスト行 — メソッド・パス・版

② ヘッダ — 何行でも続けられる

③ 空行 —「ヘッダはここまで」の合図

httpServer.py → ブラウザ（レスポンス）

```
HTTP/1.1 200 OK
Content-Type: text/html
（空行）
<!DOCTYPE html>
<html> ... </html>
```

① ステータス行 — 200 OK / 404 Not Found

② ヘッダ

③ 空行

④ 本文 — ファイルの中身がそのまま入る

流れているのはただのテキスト。サーバの仕事は、この文字列を読んで、この文字列を作って返すこと。

メソッドとステータスコード

主なメソッド

メソッド	意味
GET	指定したリソースを取得する
POST	送ったデータをサーバに処理させる
PUT	送ったデータでリソースを作成・置き換える
DELETE	指定したリソースを削除する
HEAD	ヘッダだけを取得する

この科目では **GET だけ**を実装する。

主なステータスコード

コード	意味
200 OK	成功
301 / 302	リソースが別の場所へ移った
400 Bad Request	リクエストの形式が正しくない
403 Forbidden	アクセスが禁止されている
404 Not Found	リソースが見つからない
500 Internal Server Error	サーバ側で失敗した

先頭の桁で分類する: 2xx 成功、3xx 転送、4xx クライアント側の誤り、5xx サーバ側の誤り。

2. httpServer.py

echoServer.py に何を加えるか

echoServer.py から httpServer.py へ

	echoServer.py	httpServer.py
受け取り方	`recv()` で受け取った分をそのまま扱う	**空行が届くまで**読み続ける
中身の解釈	しない	1 行目から `GET` とパスを取り出す
返すもの	受け取ったバイト列	ステータス行 + ヘッダ + 空行 + ファイルの中身
失敗したとき	扱わない	ファイルが無ければ 404 を返す

- 基本の流れ（`bind` / `listen` / `accept` / `recv` / `sendall`）はまったく同じ
- 加えるのは、受け取った文字列を解釈して、返す文字列を組み立てる部分だけ

ブラウザ → httpServer.py（リクエスト）

```
GET /server.html HTTP/1.1
Host: localhost:50080
User-Agent: Mozilla/5.0 (...)
(空行)
```

httpServer.py → ブラウザ（レスポンス）

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
(空行)
<!DOCTYPE html> ... </html>
```

並べて比べる（1）準備と接続の受け付け

echoServer.py

```
import socket
HOST = ''
PORT = 50007
                                                    
                                                    
                                                    
                                                    
                                                    
                                                    
                                                    
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                                                    
    s.bind((HOST, PORT))
                                                    
    while(True):
        s.listen(1)
                                                    
        conn, addr = s.accept()
                                                    
                                                    
        with conn:
```

httpServer.py

```
import socket
                                                    
port = 50080
file = "server.html"
cType = b"Content-Type: text/html; charset=utf-8\r\n"
htHead = b"<html><body>"
htTail = b"</body></html>"
HOST = ""
print(f"port={port}, file={file}")
                                                    
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind((HOST, port))
    s.settimeout(1.0)
    while True:
        s.listen(1)
        try:
            conn, addr = s.accept()
        except socket.timeout:
            continue
        with conn:
```

変更　追加　相手側に無い行（コメントは省略）

- `settimeout(1.0)` と `try` 〜 `except` は、Windows で Ctrl-C を効かせるため（後のスライド）

並べて比べる（2）空行まで受信する

echoServer.py

```
        with conn:
            print('Connected by', addr)
                                                
            while True:
                data = conn.recv(1024)
                if not data: break
                                                
                                                
                                                
                print('Received', repr(data))
```

httpServer.py

```
        with conn:
            print("Connected by,", addr)
            data = b""
            while True:
                chunk = conn.recv(1024)
                if not chunk:
                    break
                data += chunk
                if b"\r\n\r\n" in data or b"\n\n" in data:
                    break
            print("Received:", data.decode(), end="")
```

変更　追加　相手側に無い行（コメントは省略）

- echoServer.py は `recv()` のたびに送り返す（`while` の中）。httpServer.py は空行まで読み終えてから、ループの外で 1 回だけ送る

並べて比べる（3）応答を組み立てて送る

echoServer.py

```
                print('Received', repr(data))
                conn.sendall(data)
                print('Send', repr(data))
```

httpServer.py

```
            print("Received:", data.decode(), end="")

            rData = b"HTTP/1.1 "
            if data.startswith(b"GET "):
                try:
                    with open(file, "rb") as f:
                        rData += b"200 OK\r\n" + cType + b"\r\n"
                        rData += f.read()
                except OSError:
                    rData += b"404 Not Found\r\n" + cType + b"\r\n"
                    rData += htHead + file.encode() + b" is not found" + htTail
            else:
                rData += b"405 Method Not Allowed\r\n" + b"Allow: GET\r\n" + cType + b"\r\n"
                rData += htHead + b"Method Not Allowed" + htTail

            conn.sendall(rData)
            print("Sent:", rData.decode())
            conn.close()
```

変更　追加　相手側に無い行（コメントは省略）

- 応答に `Content-Length` が無いので、ブラウザは接続が閉じられたところを本文の終わりとみなす

httpServer.py — 受信部（空行まで読む）

```
with conn:
    print('Connected by', addr)
    data = b''
    while True:
        chunk = conn.recv(1024)
        if not chunk:                                   # 相手が空行を送らずに閉じた
            break
        data += chunk
        if b'\r\n\r\n' in data or b'\n\n' in data:    # 空行が来た = ヘッダの終わり
            break
```

- `recv()` 1 回で全部が届く保証はないので、受け取った分を連結しながら（`data += chunk`）繰り返す
- ループを抜ける条件は、空行が現れたかどうか。HTTP では空行が**ヘッダの終わり**を示す
- `if not chunk: break` が無いと、相手が途中で接続を閉じたときに無限ループになる（第 4 回と同じ）

`if b'\r\n\r\n' or b'\n\n' in data:` と書くと、条件が**常に真**になる（空でないバイト列は真）。`in data` はそれぞれの条件に書く。よくある間違いなので注意する。

改行は CRLF — 送信は厳密に、受信は寛容に

- HTTP の行末は **CRLF**（`\r\n`）。ヘッダの終わりは空行、つまり `\r\n\r\n`（RFC 9112 §2.2）
  - 送信側: 必ず CRLF で書く。`\n` だけで送ってはいけない
  - 受信側: CRLF に加えて、単独の `\n` も行末として認めてよい
- そのため、受信のループは 2 つの条件を調べる。送信と受信で厳しさが違う

> これは OS の違いではない。ソケットは改行を変換しないので、Windows でも macOS でも送られたバイト列がそのまま届く。`\n\n` が届くのは、HTTP を厳密に守らないクライアント（手書きのテストクライアント、`nc`、`telnet`）を使ったとき。

- OS による差が出るのは、ファイルを読むときだけ。`open(file)` のテキストモードは CRLF を LF に変換する
  - ファイルの中身をそのまま送るなら `open(file, 'rb')`。画像なども同じ方法で扱える

httpServer.py — 送信部（200・404・405）

```
file = 'server.html'                              # 配布サンプルでは固定
rData = b'HTTP/1.1 '
if data.startswith(b'GET '):
    try:
        with open(file, 'rb') as f:               # rb なので改行が変換されない
            rData += b'200 OK\r\n' + cType + b'\r\n'   # ステータス行・ヘッダ・空行
            rData += f.read()                             # 本文
    except OSError:                               # 開けなかった
        rData += b'404 Not Found\r\n' + cType + b'\r\n'
        rData += htHead + file.encode() + b' is not found' + htTail
else:                                             # GET 以外のメソッド
    rData += b'405 Method Not Allowed\r\n' + b'Allow: GET\r\n' + cType + b'\r\n'
    rData += htHead + b'Method Not Allowed' + htTail

conn.sendall(rData)
```

- 組み立てているのは、ただのバイト列。`str` を混ぜるときは `encode()` で `bytes` に変換する
- ステータス行・ヘッダ・**空行**・本文の順番は、前の図のとおり。`\r\n` を書き落とさない

配布サンプルは `file` を `server.html` に固定していて、リクエストされたパスを使っていない。受け取った 1 行目からリソース名を取り出して返すようにするのが、今日の課題 **ex5**。

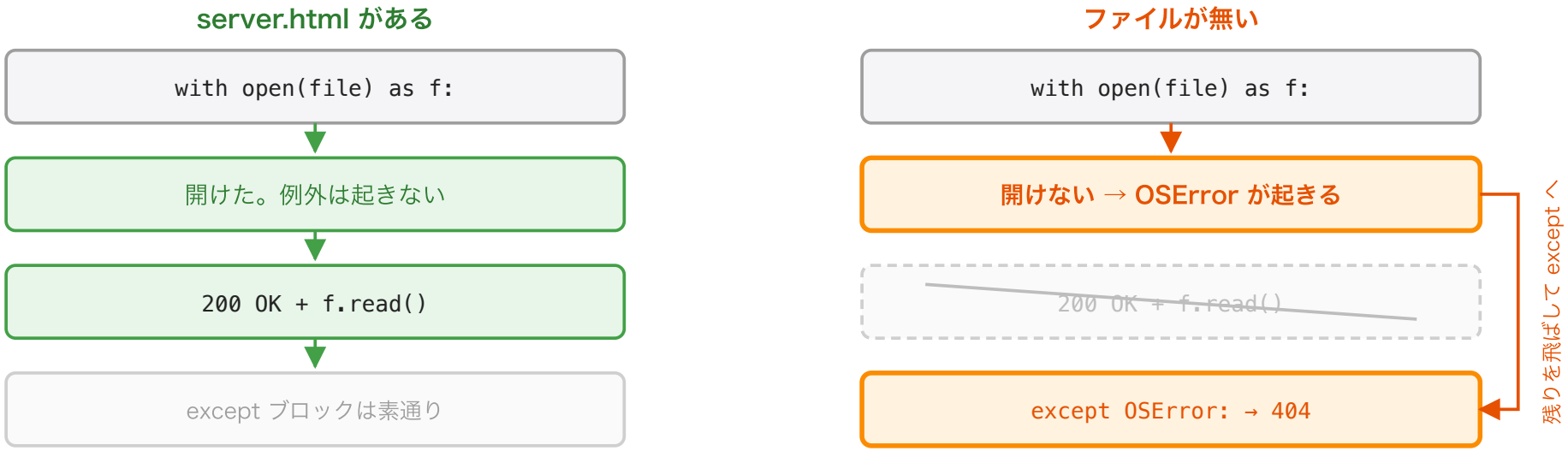
ファイル処理 — ここでも with を使う

```
with open('server.html') as f:
    text = f.read()
# ここを抜けたら自動的に close()
```

- `open()` → `read()` / `write()` / `readline()` → `close()`
- `with` の役割はソケットのときと同じ。ブロックを抜けるときに必ず `close()` する

- テキストモードで開いたファイルの `read()` は `str` を返す
  - そのため、送る前に `.encode()` が要る
- 画像など、そのままバイト列で読みたいときは `open(file, 'rb')`
- ファイルが無ければ、`open()` の時点で**例外**が発生する（次のスライド）

例外 — 失敗を戻り値ではなく、別の処理の流れで伝える



例外は「起きた行で止まって、except まで一気に飛ぶ」。だから失敗しうる処理を try の中に置く。

try 〜 except の書き方

```
try:
    with open(file) as f:
        body = f.read()
except OSError:
    body = 'not found'
```

```
try:
    n = int(input())
except ValueError:
    print('数値ではありません')
else:
    print('成功したときだけ実行')
finally:
    print('成功でも失敗でも実行')
```

- `try` の中で例外が発生すると、その行で処理を中断して `except` へ移る
- `except 例外クラス名:` で、例外の種類ごとに処理を書き分けられる
  - `OSError` — ファイルが開けない
  - `ValueError` — `int('abc')` のような変換の失敗
  - `ZeroDivisionError` — 0 で割った
- 例外を何もせずに無視しない。`except:` だけで書くと、想定外のバグまで隠れる

Ctrl-C で止める — accept() に時間制限を付ける

```
s.settimeout(1.0)               # accept() の待ちを 1 秒で打ち切る
while True:
    s.listen(1)
    try:
        conn, addr = s.accept()
    except socket.timeout:      # 1 秒間だれも接続してこなかった
        continue                # もう一度待つ
    with conn:
        ...
```

- Windows では、`accept()` で接続を待っている間は Ctrl-C が効かない。Python は Ctrl-C を受け付けても、`accept()` から戻るまで処理できない
- `settimeout(1.0)` を設定すると、`accept()` は 1 秒たったところで例外 `socket.timeout` を発生させて戻る。`except` で受け止めて待ち直すので、1 秒に 1 回は Ctrl-C を処理できる
- 時間制限が付くのは待ち受け用のソケット `s` だけ。`accept()` が返す `conn` には付かない

今日やること・次回

5 講時（1-608 で実習）

1. `httpServer.py` を起動し、**ブラウザ**から `http://localhost:50080/server.html` を開く
2. 存在しないファイル名でアクセスして、404 が返ることを確かめる
3. aiJudge の **ex5** に取り組む（提出後に自動採点。ブラウザや `httpTestClient.py` でも自分で確認する）
4. paiza ラーニング「新・Python入門編」23〜24（2 レッスン）の認定証を、まとめて 1 回で aiJudge へ提出

> `OSError: [Errno 48] Address already in use` が表示されたら、前に起動したサーバがまだ動いていないか確かめ、`Ctrl-C` で止める。それでも表示されるときは、別のポート番号を使う（`httpServer.py` には `SO_REUSEADDR` を設定済み）。

次回（10/23）：スレッドとネットワーク通信の暗号化

質問は **Teams「★ 質問用チャネル」** か、教室の TA へ。