golang で httpd を作成してみた話 その１

いかにして私は RFC7230 を読むハメになったのか

三行で説明すると

- httpd を作成中に、request を `conn.Read` せずに `conn.Write` した
- curl で `curl: (56) Recv failure: Connection reset by peer` と怒られてしまい、その理由を知りたくなった。
- RFC 7230(HTTP/1.1の仕様) を読むハメになった。

http サーバーを

- アプリケーションレイヤを扱うパッケージで作るのではなく、
- トランスポートレイヤのパッケージのみで作ってみたい

という動機があった。

まずは TCPソケット通信 について

ソケット通信の基本

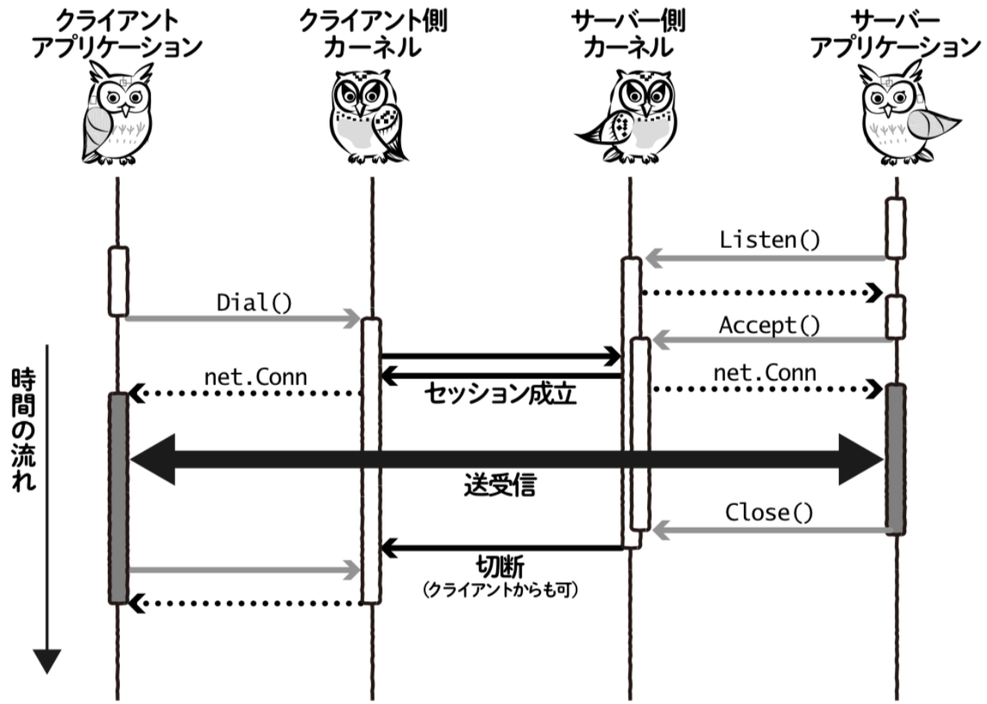
- サーバー：ソケットを開いて待ち受け、 読み込み、書き込みを行う
- クライアント：開いてるソケットに接続し、読み込み、書き込みを行う

TCP ソケット通信

システムコールの呼び出し

- Socket: ソケットのファイルディスクリプタ(`fd`)を取得する
- Bind: その取得した `fd` にポート番号、アドレスなどを紐付ける
- Listen: その `fd` に対して、外部からのアクセスを待つ
- Accept: アクセスがあったら、コネクションソケットを作成し、そのソケットの `fd` を返す。 Clientとの通信はこのときに作成した `fd` を通じて行う
- Read, Write: その `fd` に対して、読み込み、書き込みなどを行う。
- Close: 通信が終わったら、 `fd` を 閉じる。

TCP ソケット通信(golang)



▶ 図 6.6　TCP の通信手順

- `net.Dial` : 接続を待ちをしているソケットに対して、接続を行う
- `net.Listen` : Socket(2), Bind(2), Listen(2) を行う
- `net.Accept` : Accept(2) を行う
- `net.Conn` : 通信が確立できたら、クライアント、サーバー側の両方に通信を行うためのオブジェクト `net.Conn` が返ってくる
- `net.Close` : Close(2) ソケットを閉じる

golang では、
クライアントが呼ぶAPIの名前は Dial
サーバーが呼ぶAPIの名前は Listen という命名規則がある。

ではそれを実装していく

サーバー側についてのみ言及する

TCP サーバー

```
ln, err := net.Listen("tcp", ":8080")

conn, err := ln.Accept()

conn.Read(...)
conn.Write(...)
conn.Close()
```

一回でもアクセスされたら通信が終了する

TCP サーバー(ちょっとだけかしこい)

```
ln, err := net.Listen("tcp", ":8080")

// リクエストを複数受け付けるために for で何度も Accept する
for {
  conn, err := ln.Accept()

  // 1 リクエストを処理中に他のリクエストを受け付ける(Accept)ために非同期処理をする
  go func() {
    conn.Read(...)
    conn.Write(...)
    conn.Close()
  }
}
```

さっきの実装でいわゆる echo サーバーは実装できる。 デモ:

tchecho.go を telnet で確認

が、 HTTP 通信したいよね。

HTTP 通信

- 基本的にはさっきの Write のところに HTTP のメッセージ構文に沿った形で書き込めばおｋ

HTTP の基本

HTTP メッセージを BNF で書くとこんな感じ

```
HTTP-Message = start-line
               *( header-field CRLF)
               CRLF
               [ message-body ]

start-line = request-line / status-line
```

- * (hoge) 0回以上()内のhogeの繰り返しを表す
- [hoge] 0回か1回の []内のhogeを表す

cf. https://triple-underscore.github.io/RFC7230-ja.html#section-3

リクエスト

ex.

```
GET / HTTP/1.1 -- リクエスト行
Host: www.google.co.jp -- ヘッダ
User-Agent: curl/7.67.0 -- ヘッダ
Accept: */* -- ヘッダ
-- 空行(ヘッダとメッセージボディを区別するためにこれが必要)
-- メッセージボディ(POSTメソッドなどで使用する)
```

- -- hoge は、その行が何を表しているかを示している。

レスポンス

ex.

```
HTTP/1.1 200 OK -- レスポンス行
Connection: close -- ヘッダ
Content-Type: application/json; charset=utf-8 -- ヘッダ
Date: Sun, 05 Jan 2020 15:57:44 GMT -- ヘッダ
Content-Length: 177 -- ヘッダ
-- 空行(ヘッダとメッセージボディを区別するためにこれが必要)
{"msg": "ok"} -- メッセージボディ
```

ではそれを実装していく

サーバー側についてのみ言及する

- HTTP(アプリケーション層)のパッケージ(net/http)を使った実装
- TCP(トランスポート層)のパッケージ(net)を直接使った実装

  とを比べる

HTTPサーバー (golang) net/http

```
func handler(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "Hello World\n")
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

HTTPサーバー (golang) net

```
ln, err := net.Listen("tcp", ":8080")

for {
  conn, err := ln.Accept()

  go func() {
    conn.Read(...)
    // HTTP のレスポンスの構文に従ってメッセージを作成
    conn.Write([]byte("HTTP/1.0 200 OK\r\n\r\nHello World\n"))
    conn.Close()
  }
}
```

デモ： tcphttpserver.go を curl で確認

connection を read しなかったらどうなるんだろうか

TCP ソケットを使った実装で read(2) しなかったら curl に `connection reset by peer` と怒られてしまった。

以下それぞれのパケットキャプチャ結果を比べてみる

- client の request を読まなかった場合
- client の request を読みはしたが全て読まなかった場合
- client の request を全て読んだ場合

client の request を読まなかった場合

read を一切しない

```
X / _ /) < curl -w '%{http_code}\n' --http1.0 -4 localhost:8080
Hello World
curl: (56) Recv failure: Connection reset by peer
200
```

```
X / _ /) < tshark -i lo0 -Y "tcp.port==8080"
Capturing on 'Loopback: lo0'
    1   0.000000    127.0.0.1 → 127.0.0.1    TCP 68 64157 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1169567321 TSecr=0 SACK_PERM=1
    2   0.000076    127.0.0.1 → 127.0.0.1    TCP 68 8080 → 64157 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=1169567321 TSecr=1169567321 SACK_PERM=1
    3   0.000087    127.0.0.1 → 127.0.0.1    TCP 56 64157 → 8080 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1169567321 TSecr=1169567321
    4   0.000103    127.0.0.1 → 127.0.0.1    TCP 56 [TCP Window Update] 8080 → 64157 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1169567321 TSecr=1169567321
    5   0.000189    127.0.0.1 → 127.0.0.1    HTTP 134 GET / HTTP/1.0
    6   0.000201    127.0.0.1 → 127.0.0.1    TCP 56 8080 → 64157 [ACK] Seq=1 Ack=79 Win=408192 Len=0 TSval=1169567321 TSecr=1169567321
    7   0.000642    127.0.0.1 → 127.0.0.1    TCP 87 HTTP/1.0 200 OK  [TCP segment of a reassembled PDU]
    8   0.000659    127.0.0.1 → 127.0.0.1    TCP 56 64157 → 8080 [ACK] Seq=79 Ack=32 Win=408256 Len=0 TSval=1169567321 TSecr=1169567321
    9   0.000676    127.0.0.1 → 127.0.0.1    TCP 44 8080 → 64157 [RST, ACK] Seq=32 Ack=79 Win=408192 Len=0
```

client の request を読みはしたが全て読まなかった場合

read をちょっとする

```
request := make([]byte, 10)
conn.Read(request)
```

```
X / _ /) < curl -w '%{http_code}\n' --http1.0 -4 localhost:8080
Hello World
curl: (56) Recv failure: Connection reset by peer
200
```

```
X / _ /) < tshark -i lo0 -Y "tcp.port==8080"
Capturing on 'Loopback: lo0'
   48 157.056079    127.0.0.1 → 127.0.0.1    TCP 68 64200 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1169776055 TSecr=0 SACK_PERM=1
   49 157.056170    127.0.0.1 → 127.0.0.1    TCP 68 8080 → 64200 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=1169776055 TSecr=1169776055 SACK_PERM=1
   50 157.056180    127.0.0.1 → 127.0.0.1    TCP 56 64200 → 8080 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1169776055 TSecr=1169776055
   51 157.056195    127.0.0.1 → 127.0.0.1    TCP 56 [TCP Window Update] 8080 → 64200 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1169776055 TSecr=1169776055
   52 157.056339    127.0.0.1 → 127.0.0.1    HTTP 134 GET / HTTP/1.0
   53 157.056352    127.0.0.1 → 127.0.0.1    TCP 56 8080 → 64200 [ACK] Seq=1 Ack=79 Win=408192 Len=0 TSval=1169776055 TSecr=1169776055
   54 157.056503    127.0.0.1 → 127.0.0.1    TCP 87 HTTP/1.0 200 OK  [TCP segment of a reassembled PDU]
   55 157.056516    127.0.0.1 → 127.0.0.1    TCP 44 8080 → 64200 [RST, ACK] Seq=32 Ack=79 Win=408192 Len=0
   56 157.056516    127.0.0.1 → 127.0.0.1    TCP 56 64200 → 8080 [ACK] Seq=79 Ack=32 Win=408256 Len=0 TSval=1169776055 TSecr=1169776055
   57 157.056533    127.0.0.1 → 127.0.0.1    TCP 44 8080 → 64200 [RST] Seq=32 Win=0 Len=0
```

client の request を全て読んだ場合

```
request := make([]byte, 1024)
conn.Read(request)
```

```
X / _ /) < curl -w '%{http_code}' --http1.0 -4 localhost:8080
Hello World
200%
```

```
X / _ /) < tshark -i lo0 -Y "tcp.port==8080"
Capturing on 'Loopback: lo0'
    1   0.000000    127.0.0.1 → 127.0.0.1    TCP 68 64251 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1170062780 TSecr=0 SACK_PERM=1
    2   0.000079    127.0.0.1 → 127.0.0.1    TCP 68 8080 → 64251 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=1170062780 TSecr=1170062780 SACK_PERM=1
    3   0.000089    127.0.0.1 → 127.0.0.1    TCP 56 64251 → 8080 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1170062780 TSecr=1170062780
    4   0.000104    127.0.0.1 → 127.0.0.1    TCP 56 [TCP Window Update] 8080 → 64251 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1170062780 TSecr=1170062780
    5   0.000261    127.0.0.1 → 127.0.0.1    HTTP 134 GET / HTTP/1.0
    6   0.000274    127.0.0.1 → 127.0.0.1    TCP 56 8080 → 64251 [ACK] Seq=1 Ack=79 Win=408192 Len=0 TSval=1170062780 TSecr=1170062780
    7   0.000323    127.0.0.1 → 127.0.0.1    TCP 87 HTTP/1.0 200 OK  [TCP segment of a reassembled PDU]
    8   0.000335    127.0.0.1 → 127.0.0.1    TCP 56 64251 → 8080 [ACK] Seq=79 Ack=32 Win=408256 Len=0 TSval=1170062780 TSecr=1170062780
    9   0.000353    127.0.0.1 → 127.0.0.1    HTTP 56 HTTP/1.0 200 OK
   10   0.000364    127.0.0.1 → 127.0.0.1    TCP 56 64251 → 8080 [ACK] Seq=79 Ack=33 Win=408256 Len=0 TSval=1170062780 TSecr=1170062780
   11   0.000438    127.0.0.1 → 127.0.0.1    TCP 56 64251 → 8080 [FIN, ACK] Seq=79 Ack=33 Win=408256 Len=0 TSval=1170062780 TSecr=1170062780
   12   0.000458    127.0.0.1 → 127.0.0.1    TCP 56 8080 → 64251 [ACK] Seq=33 Ack=80 Win=408192 Len=0 TSval=1170062780 TSecr=1170062780
```

何が違うのかまとめる

- read した場合:
  - client: [FIN(接続の終了を通知する), ACK] を送る
  - server: [ACK] を送る
- read しなかった場合:
  - server: [RST(確立された接続を一方的に切断するパケット), ACK(close)] を送る

小ネタ: tshark はいいぞ

- tcpdump の小ネタを挟む。 ACKフラグが ． になってて混乱する
- tshark はパケットキャプチャ結果が見やすくていいぞ

キャプチャを見たがよくわからん。
ツイッターで質問してみた

RFC 7230 の 6.6節に書いてありますが、サーバー側はレスポンス後にcloseしてはならず、クライアントからのcloseをReadで待った後にcloseする必要があります。
TCP コネクションが閉じた後に RSTパケットが送られるとクライアント側のバッファがクリアされる可能性があります

https://twitter.com/kawasin73/status/1214122400003940352

RFC 7230(HTTP1.1の仕様) を読む

client: close を 送る

server: close を 読み取る

server: close を 送る

client: close を 読み取る

というルールに従う必要がある

で、server 側がデータを全て読み取ってない状態で close すると、 RSTパケットがClientに送られてしまい、connection refused エラーが発生する

The Connection header field (Section 6.1) provides a "close" connection option that a sender SHOULD send when it wishes to close the connection after the current request/response pair.

A client that sends a "close" connection option MUST NOT send further requests on that connection (after the one containing "close") and MUST close the connection after reading the final response message corresponding to this request.

A server that receives a "close" connection option MUST initiate a close of the connection (see below) after it sends the final response to the request that contained "close". The server SHOULD send a "close" connection option in its final response on that connection. The server MUST NOT process any further requests received on that connection.

A server that sends a "close" connection option MUST initiate a close of the connection (see below) after it sends the response containing "close". The server MUST NOT process any further requests received on that connection.

A client that receives a "close" connection option MUST cease sending requests on that connection and close the connection after reading the response message containing the "close"; if additional pipelined requests had been sent on the connection, the client SHOULD NOT assume that they will be processed by the server.

https://tools.ietf.org/html/rfc7230#section-6.6

[1]ListenしていないポートにSYNパケットが送信された場合、RSTパケットがSYNパケットの送信元に返される。

[2] Accept済みのソケットに対して、データを全て読み取っていない(EOFに達していない状態）でcloseを発行した場合にコネクションの相手側にRSTパケットが送られる。

[3] Linux限定だが、tcp*abort*on_overflowがonになっている状態で設定したバックログ以上の未Acceptのコネクションが張られた場合(http://linux.die.net/man/7/tcp)

[4] tcp*max*orphans以上のorhanコネクションが張られた場合、orhanなコネクションに対してRSTパケットが送られる。

＊orpahなコネクションってソケットがクローズされてるようなコネクションってことかな？

[5] Half-open connection
https://tools.ietf.org/html/rfc793#section-3.5

cf. https://ichiroku.hatenadiary.org/entry/20101027/1288199148

なるほど。納得

つまり、 client 側のリクエストをいったん全て読まない状態で、
close を サーバーが発行すると、 Client側に RST パケットが送られてしまい、
HTTP の仕様から外れてしまい、 curl がそれ、 HTTPの仕様から外れてるからだめだよ `Recv failure: Connection reset by peer` って教えてくれるということ。

おわり。

cf.

- GoでTCPサーバー上で独自プロトコルKey-Value Storeを実装する(知識編) - Qiita
- ASCII.jp：Goでたたく TCPソケット（前編）
- HTTP入門
- Goでechoサーバーを書いた時のメモ - bati11 の 日記
- RSTパケットとは｜「分かりそう」で「分からない」でも「分かった」気になれるIT用語辞典