

# Web基礎講座

2021/7/31(土) 13:00～

# 目的と概要

- Web概観講座にスムーズに入れるように基礎事項を予習する
  - 基本となる概念はある程度しっかりやったほうがスムーズに入れそう
- 全ては（もちろん多すぎて）解説できないので、  
テーマをピックアップして解説
- 何らかのプログラミングはしたことがあるのを前提とします

# 目次

- Linux Basic
- Network Basic
- Object Oriented Programming
- Functional Programming

# Linux Basic

# なぜ Linux か

Web系の開発は、大きく分けてLinux系統で作る方法(これから解説)とWindows系統で作る方法(Windows Server/IIS 等)がある。

しかし、最近の流れとして Windows であっても Linux の知識が有効に使える場面が増えてきた (WSL等)。それに加え、

- オープンソースである
- 汎用的で広く使われている
- コマンドでの操作は自動化（= Webサービス）に向く

という特徴がWeb開発に向くため、Linuxを用いるのが吉

# 手元に Linux 環境がない人

- Windowsならある人
  - WSL で調べて仮想Linux環境をゲットしよう。
- Macの人
  - そのままでOK. Web開発は実際にはMacを使って行われることも多いです
- 自作OSの人
  - どうせなら Linux 互換のOSを作っちゃいましょう

# コマンド (1)

Linuxの操作は基本的にコマンドを用いて行う。単純なコマンドは以下の形式をしている：

```
(コマンド名) (引数1) (引数2) (引数3) ...
```

例えば

```
vim index.html
```

であれば、コマンド名は `vim`、引数1 は `index.html` である。

`-` から始まる引数は **オプション** と呼ばれ、細かい設定を指定できる。 7

# コマンド (2)

先程のコマンドは **ターミナル** (黒い画面のやつ) と呼ばれるプログラムに入力すると、その中(**シェル**) で解釈され、実行される。

例えば

```
vim index.html
```

は、「**vim** というプログラムを探してきて、そのプログラムへの引数として **index.html** という文字列を渡してプログラムを起動せよ」という意味になる。

起動したプログラムの実体を **プロセス** という。シェルもプロセス。



# コマンド (3)

コマンドは誰が実行するのか、という点が重要なことがある。特に

- 管理者権限を持った特別なユーザ (**root**)
- 一般ユーザ

のどちらが実行するのかはとても重要。

コマンドの表記の先頭に **\$** や **#** がついてる事が多い。

**\$** から始まるのは **一般ユーザでの実行**、**#** は **root** での実行を表す。  
ターミナルでもちゃんと表示されることが多いのでチェック。

# コマンド (4)

コマンドを現在のユーザではなく管理者として実行したければ、コマンドの先頭に `sudo` をつけば良い

```
sudo vim /etc/httpd.conf
```

`sudo` というプログラムに、`vim` と `/etc/httpd.conf` を引数として渡して実行している。

`sudo` は管理者権限を取得し、第一引数で与えられたプログラムを、第2引数以降を引数として実行する

権限が足りない、というようなエラーが出たときにはよく使われる

# コマンド (5)

複数のコマンド列を一つのテキストファイルに並べて書いておけば、bashなどのシェルにそのファイルを渡して、一括実行させることができる。

これを **バッチ実行** と呼ぶ。

```
$ vim test.sh # test.sh というテキストファイルにコマンドを改行区切りで並べて書く  
$ bash test.sh # 先程書いたコマンドが一括して実行される
```

自動化スクリプトを書いておくと、実行は `bash test.sh` など一瞬で済むので便利。

# manコマンド

コマンドの細かい使い方がわからなくなったら `man` コマンドを、  
分からないコマンド名を引数として実行することでヘルプを閲覧できる。

```
man echo
```

とすると、`echo` コマンドのヘルプが見れる。`j` キーで下に、`k` キーで上にスクロールし、`q` で閲覧終了できる。

もちろん `man man` とすれば、`man` コマンド自体のヘルプも見られる。

# パス (1)

passではなくpath。リソース（ファイルとか）の在り処を表した文字列。ディレクトリは `/` で区切って表す。

```
/var/www/html
```

rootの中のvarの中のwwwの中のhtmlという場所を表す。

```
~/work
```

~は現在のユーザのホームディレクトリ( `$HOME` ) を表す。

例えば `/usr/kakira/work` を指定したことになる

# パス (2)

パスは大きく分けて2種類ある。

- 絶対パス
  - `/` から始まる。ファイルシステムの根本(**root**)から末端まで指定
- 相対パス
  - `/` から始まらない。あるリソースから見た相対的な位置を指定

相対パスを記述する際、

`.` だけからなるディレクトリは**カレントディレクトリ**を、

`..` だけからなるディレクトリは一つ上（親）のディレクトリを表す

# パス (3)

```
/
|
+--foo # <- 現在ココ！
|   |
|   +-- a.txt
|
+--var
|
+-- b.txt
```

現在 `foo` ディレクトリにいる場合、`a.txt` は `a.txt` `./a.txt` `/foo/a.txt` と表せる。

`b.txt` は `../var/b.txt` や `/var/b.txt` と表せる

# カレントディレクトリ

今どこにいるかを表す。 `pwd` コマンドで取得可能。  
これを変更するには、 `cd` (change directory) コマンドを実行すれば良い

```
$ pwd
/var/www/html
$ cd .. # ひとつ上のディレクトリへ移動
$ pwd
/var/www
```

引数として相対パスを渡してコマンドを起動するとき、現在のカレントディレクトリが基準点となることが多い。



# 環境変数とシェル変数の違い

どちらも変数。コマンドを実行したとき、起動したプロセスについて、それらが引き継がれるかどうか異なる

- 環境変数: 引き継がれる
- シェル変数: 引き継がれない

色々なプログラムの設定値を保存しておくのは、環境変数の役目。計算の途中結果保存など普通の変数として使いたいのであればシェル変数を使う。

# 環境変数の使い方

## 値の設定

```
export x="hoge"  
export y=1
```

## 参照（展開）

`$x` または `${x}`

`=` の周りにスペースを付けてはならない（ゼツタイ！）。また、`{ }` で囲まないと期待通りに展開されない場合があるので注意

# シェル変数の使い方

## 値の設定

```
x="hoge"  
y=1
```

## 参照（展開）

```
$x または ${x}
```

export をつけない。あとは環境変数と同じ。 `echo` コマンドで表示してみよう(`echo $x`)

# 特殊な環境変数

いくつかの環境変数名は予約されていて、特殊な意味を持つ：

- HOME: 現在のユーザのホームディレクトリを表す
- PWD: カレントディレクトリのパスを表す
- LANG: 言語設定を表す
- **PATH**: 実行するコマンドを探すディレクトリのパスのリスト  
(`:`区切り)

もっとたくさんあるが、`printenv` コマンドで自分で設定した環境変数も合わせて確認可能

# パスを通す (1)

**パスを通す** = プログラムのデフォルトの置き場所を格納した環境変数 PATH に、新たなパスを追加すること。

コマンド名を単体で指定して実行するとき、シェルはPATHからプログラムを探す。 **:** 区切りのパスを、左から右に探していき見つかったらそれを使う（探索は打ち切る）。  
ここに登録されていないと command not found になる。

```
$ echo $PATH
```

でどんな値が設定されているか、確認してみよう。

# パスを通す (2)

`source` コマンドを実行すると、続く引数で指定されたテキストファイルの中身を、現在のターミナルで手入力するのと同じ挙動をする。

使っているシェルが **bash** の場合、`~/.bashrc` に

```
export PATH="$PATH:/usr/local/bin/hoge"
```

と追記して、`source ~/.bashrc` すると、もともとの環境変数 `$PATH` に `:/usr/local/bin/hoge` が連結された文字が再代入される。つまりそのパスが探索範囲として追加される。

# ~/.bash\_profile と ~/.bashrc

.から始まるファイル(ドットファイル)やディレクトリは、Linux では隠しファイルや隠しディレクトリを表す。

bash 用の設定ファイルとして `~/.bash_profile` と `~/.bashrc` がある。

- `~/.bash_profile` はユーザのログイン時に一回実行される
  - そのユーザで共通の設定(環境変数など)はこちら
- `~/.bashrc` は、bashを起動するたびに実行される
  - こちらにbash独自の設定等を書く(エイリアスなど)

# 所有者とパーミッション

ファイルやディレクトリには、所有者やパーミッションと呼ばれる情報があり、これによってアクセス制御を行っている

所有者はそのファイルを誰／どのグループが持っているか  
パーミッションはそのファイルに具体的に与えるアクセス権を表す。

これらの情報は、`ls -l` (`ls` コマンドに `-l` オプションをつけたもの) を実行すると表示される



```
kakira@kakira-7-PC:/mnt/c/Users/kakira-7/test/test$ ls -l
合計 0
-rwxrwxrwx 1 kakira kakira 0 7月 30 22:10 a.txt
-rwxrwxrwx 1 kakira kakira 0 7月 30 22:10 b.txt
drwxrwxrwx 1 kakira kakira 512 7月 30 22:12 src
```

パーミッション (後述)

所有者

ファイルサイズ (byte)

ファイル名

作成日時

所有グループ

ファイルタイプ

-: ファイル

d: ディレクトリ

# 所有者・所有グループ

パーミッションを定義する際に必要となる。

- 所有者: ファイル/ディレクトリを所有しているユーザを指定
- 所有グループ: // グループを指定

`chown` で所有設定を変えられる

```
$ chown hoge:wheel ./a.txt
```

`./a.txt` の所有者を `hoge` に、グループを `wheel` に変更

# パーミッション

r(Read), w(Write), x(eXecute) の3つの文字の組み合わせを、所有者、所有グループ、その他の人たちについて設定する。

- 例: `rwxr-xr-x`
  - 所有者には`rw`x、所有グループには`r-x`、その他の人には`r-x` を与えている
  - 所有者は読み書き実行, 所有グループとその他の人は読み・実行ができる
- これを**8進数**3文字で表す。(r: 4, w: 2, x: 1)
  - `rwxr-xr-x` は `755` になる。

# パーミッションの変更

`chmod` コマンドでできる。

例

```
$ chmod +x ./a.txt
```

→所有者、グループ、その他の人たち全てに対して実行権限を追加

```
$ chmod 755 ./a.txt
```

→パーミッションを `rwxr-xr-x` に設定

# 標準入力、標準出力

プログラムを、入力を標準入力と呼ばれるところから受け取り、出力を標準出力と呼ばれるところに行うよう作っておくと良いことがある。コマンドの組み合わせが容易になる。

標準入力を実際に何から入力を受け取るか、標準出力が何に出力を書き出すか、はその時の状況によって異なる。例えば標準出力なら、

- 直接ターミナルからコマンド実行 -> 標準出力はターミナルと接続
- **パイプ** を使って、標準出力を他のプログラムの入力に紐づけておく

などが考えられる。

# パイプ

2つ以上のコマンドを **|** で連結することによって、左のコマンドの標準出力を右のコマンドの標準入力につないでから、コマンドの実行が可能。

```
cat a.txt | grep hoge
```

**cat** コマンド(ファイルの中身を標準出力に出力) の標準出力(つまりファイルの中身)を、 **grep** コマンド(指定されたテキストが含まれる行のみを標準出力)の標準入力につないでいる。

# リダイレクト (1)

標準入力、標準出力以外にも、標準エラー出力や、ファイルなど、様々な入出力ソースが考えられる。

コマンドで何も指定せずに実行すると、デフォルトではプログラムの入力は標準入力、出力は標準エラー出力に結びついている。この紐付けをコマンド実行時に一時的に変えること（**リダイレクト**）が可能。

```
echo "hello" >test.txt
```

**>**によって、**echo** コマンドの標準出力を **test.txt** に紐づけている

# リダイレクト (2)

標準入力も変えられる

```
./a.out <test.txt
```

./a.out というプログラムの標準入力を `test.txt` に紐づけている。

標準エラー出力を標準出力に紐付け、標準出力を `out.txt` に紐付ける

```
./a.out >out.txt 2>&1
```

./a.out の標準エラー出力と標準出力の両方が `out.txt` に書き出される 32



# バックグラウンド実行

コマンドの最後に `&` をつけると、コマンドが裏で実行されつづける状態になる。

```
./a.out &
```

a.outの実行が終わる前に、ターミナルへ制御が戻ってくる（コマンドが打てる状態になる）

裏で実行されているプロセスは、`jobs`、`fg`、`bg`、`ps` コマンドなどで管理できる（調べてみよう）

# 逐次実行

2つのコマンドを `;` でつなげると、左のコマンドが終了してから、右のコマンドを実行することができる

```
sleep 5 ; echo hello
```

5秒後にhelloと標準出力に出力する。パイプだと2つのコマンド実行開始がほぼ同時になるので注意（実験してみよう）。

# 条件分岐

bashにも条件分岐がある。

```
x=1
if [ $x -eq 1 ]; then      # xが1なら
    echo "x is 1"
else                       # それ以外のときは
    echo "x is not 1"
fi
```

半角スペースなどを勝手に削除しないようにしよう。ハマります。

# 繰り返し

`for` と `while` が使える。

```
for i in `seq 1 10`; do  
    echo ${i}  
done
```

```
while read line ; do  
    echo ${line}  
  
done < test.txt
```

前者は1から10まで表示。後者はtest.txtの内容を1行ずつ表示。

# 色々なコマンド

とにかく調べよう。

- 文字列処理系

- `grep`, `sort`, `head`, `tail`, `cut`, `sed`, `wc`, `awk`

- プロセス管理系

- `ps`, `top`, `w`, `kill`, `fg`, `bg`, `jobs`

- その他よく使うやつ

- `find`, `seq`

# 便利なbashの機能 (1)

- 前のコマンドを実行
  - 矢印上キーを押す（履歴をたどる）
  - あるいは **!!** でもOK（直前のコマンドを再実行）
- **!a** とすると、今までのコマンド実行履歴の中で直近の **a** から始まるコマンドを拾ってきて実行してくれる。

# 便利なbashの機能 (2)

- コマンド置換

- `cat $(./a.out)` のように `$()` でコマンドを囲むと、そのコマンドの標準出力がそこに置き換わる。
  - `./a.out` で `test.txt` という文字列が出力される場合、  
`cat test.txt` が実行される。

- ブレース展開

- `echo {aaa, bbb}` みたいに実行すると、`echo aaa ; echo bbb` と同等の実行結果になる。
- コマンドの分配法則。

# ワンライナーを書いてみる

ワンライナー: ある目的を果たす、一行で書かれたシェルスクリプト

```
ls -l | grep .txt | sort -k 5r | head -n3
```

- カレントディレクトリにあるファイル・ディレクトリー一覧から、
- .txtという文字列を含む行のみを抜き出し
- サイズの大きい順に並び替え
- Top3だけ表示



# ワンライナー練習

**du** コマンドを調べ、あるディレクトリの中のファイルたちの中で大きいファイルTop5をあげよ。

# (コラム) プロセスとスレッド

**プロセス** はOSの中の実行インスタンス。他のプロセスのメモリ空間とは独立しているので、基本的に他のプロセスの内容は読み書きできない（OS（カーネル）にお願いすると間接的に可能）

**スレッド** はプロセスの中の実行インスタンス。同じプロセス内のスレッドたちは互いにメモリ空間を共有し合うことも可能。並列動作っぽくプログラミングできて嬉しい。

# (コラム) Linuxのディレクトリ構造

`ls /` としてみると、最初から色々なディレクトリがあることが分かる。これらは使用目的ごとに分けられていて、使い方の相場が決まっている。何個かピックアップすると次のようになる

```
/          # ルートディレクトリ。全ての根となる特殊なディレクトリ
+-- bin, sbin # binary。s付きはsuper user用。実行ファイルの本体が入る。
+-- etc       # 設定ファイル
+-- lib       # ライブラリ化されたバイナリの一部が格納されている
+-- opt       # option. あとから追加したプログラムはココに入れると良い。
+-- root      # root アカウントのホームディレクトリ（/ではない）
+-- usr       # 全てのユーザが使えるアプリをインストールする
+-- var       # variable. ログファイルなどが格納されるディレクトリ
+-- tmp       # 一時ファイル。後で消されても良いやつを入れる
```

# Network Basic

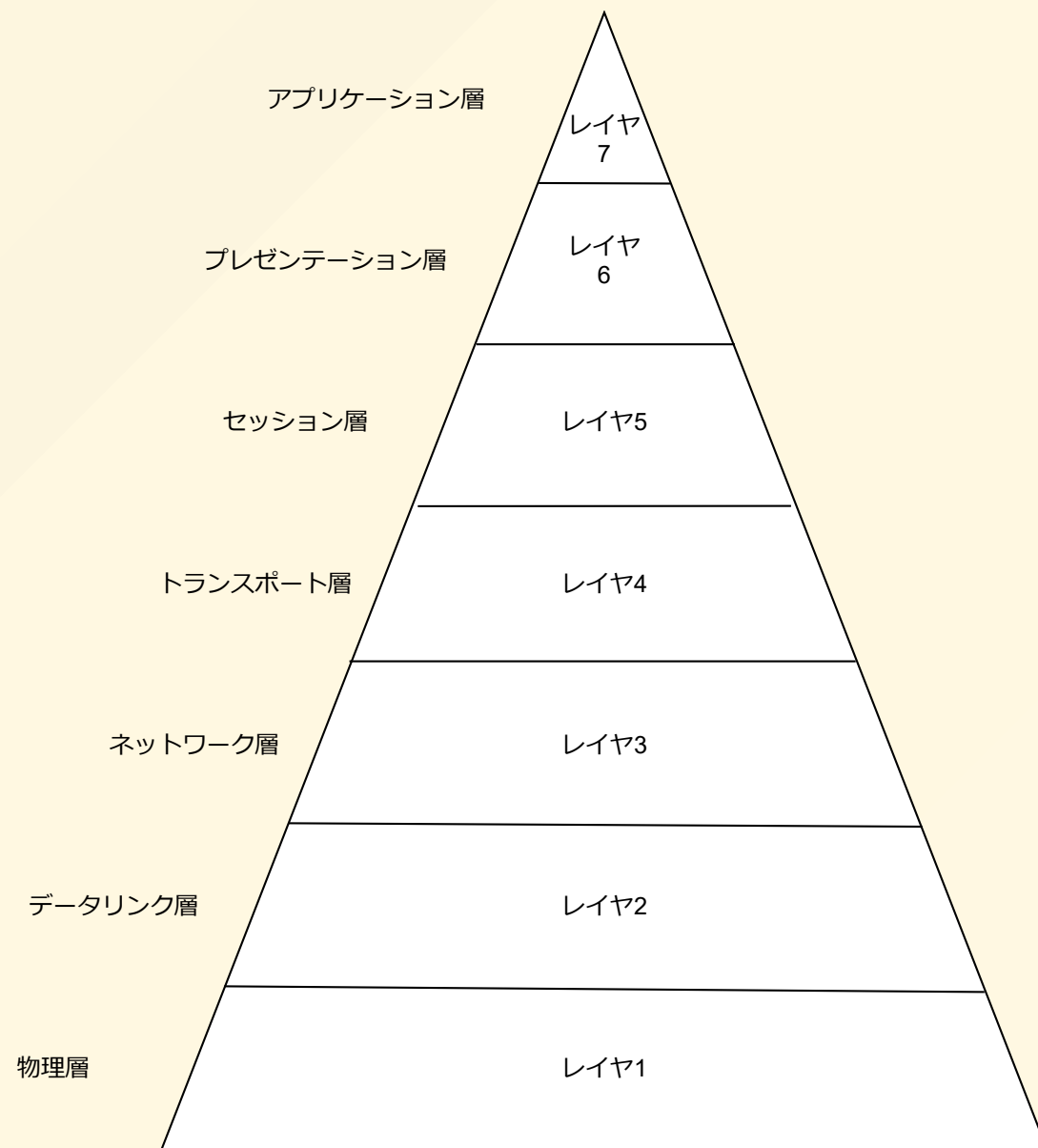
一部分だけ。詳しくは3分間ネットワーキングなどが良さそう

# やりたいこと

プログラムAから遠隔地にあるプログラムBへデータを届けたい。  
ここでは TCP/IP 通信を使った方法を学ぼう

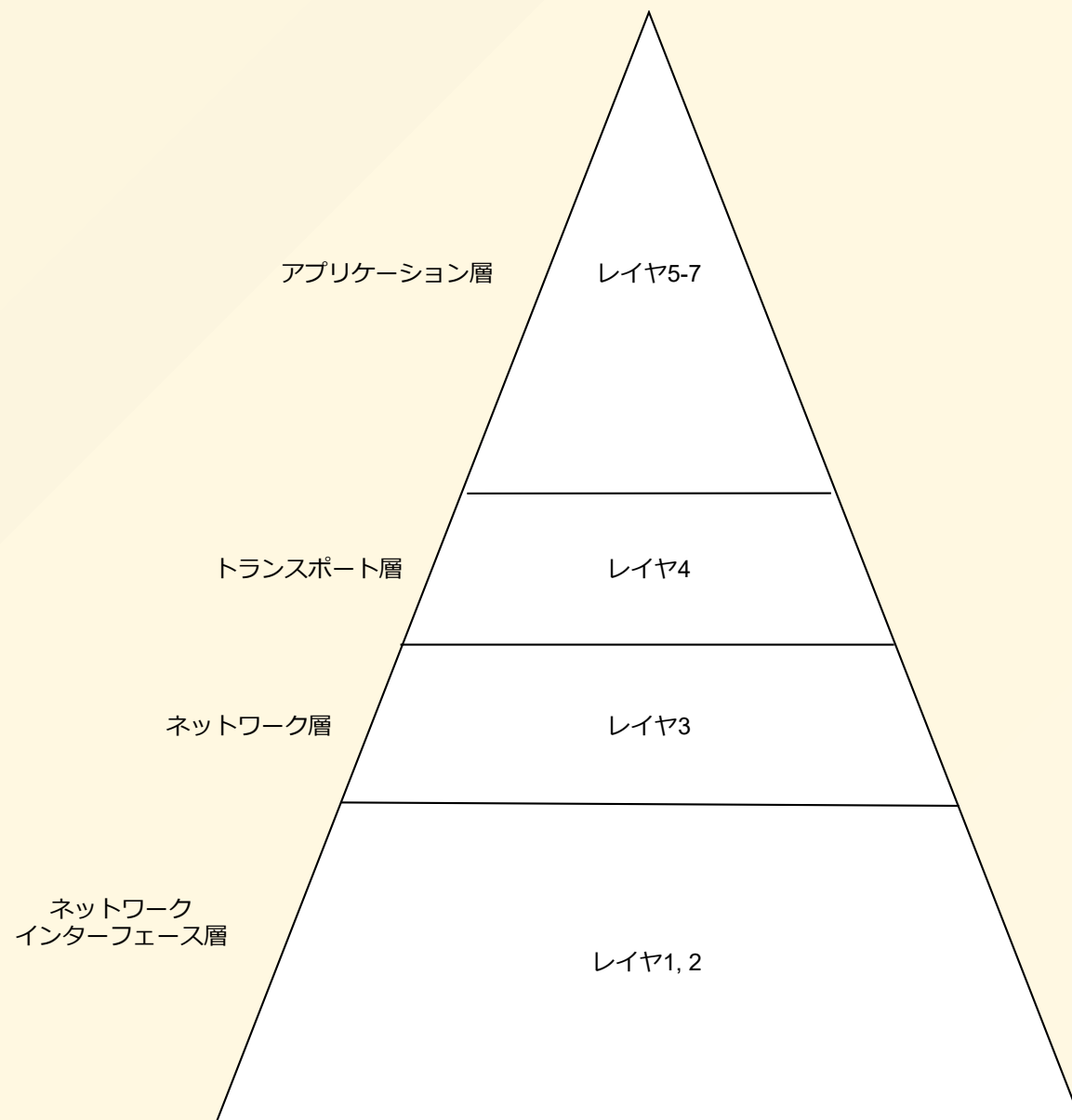
# OSI参照モデル

- 通信を、関心(目的)によって階層分けする。
  - 各階層の実装では自分のことだけ考えれば良い  
(実装が楽)
  - 同じ階層の違う実装、機器は「切替」可能
- 今回扱う範囲: レイヤー3, 4  
中心



# TCP/IP

- プロトコル(Protocol): 取り決め
  - 通信方法のルール（パケットのフォーマットや送り方など）を決めたもの
  - TCP/IP はプロトコルスイート（プロトコルの集合）
- OSI参照モデルと比較してみよう



# MACアドレス

- データリンク層での通信。隣接機器との通信に使う。
- 基本的にネットワークインターフェースカード（**NIC**）ごとに元から固定で割り当てられている16進数12桁のID。
- コレを指定して、隣接機器にデータフレームを届ける



# IPアドレス（1）

- ネットワーク層での通信。ルータで区切られたネットワーク同士を跨いだ通信に使う
- NICごとに1つ割り当てられるが、その値はその時のネットワーク構成によって異なる。
  - MACアドレスとは異なり、IPアドレスは可変
- ルータはNICを2つ以上もつ。つまりIPアドレスを2個以上もつ。

# IPアドレス (2)

- 一般にはルータの内側ではローカルIPが与えられる(DHCP/固定割当)。外側のIPは世界組織（ICANNなど）が管理している
  - 内側でのIPアドレスは、内側でのみ有効。
- ネットワークの内側から外側に対して通信するには NAT が必要
  - 行きのパケット（内側 -> 外側）において、ルータが適当なポート番号を選び、そのポート番号とIPアドレスをセットで

# サブネットマスク

- 与えられたIPアドレスが、同じネットワークなのかを判別するのに必要。機器側に設定しておく。
- 2進数で表して左から続く1の領域が「ネットワーク部」。そうでないところが「ホスト部」
- `255.255.255.0` は2進数で、 `11111111 11111111 11111111 00000000`
  - IPアドレスの左から24桁は、ネットワーク部で、それ以降はホスト部
  - ネットワーク部が違うIPアドレスは違うネットワークの中にあるはずなので、ルータにパケットを渡す。

# ポート番号

- IPアドレスだけでは、コンピュータ本体までしかデータが届かない
- コンピュータ内部ではいろいろなアプリが動いている
- どのアプリにデータを届けるのかを識別するための番号がポート番号

# ドメイン/DNS

- IPアドレスは人間にとっては覚えにくい
- ドメインという文字列とIPアドレスを紐付けておく
- ドメインからIPアドレス（逆も）を引ける仕組み = DNS
- ローカルだけでよければhostsファイル手段もある

# Object Oriented Programming

# クラスとオブジェクト

- データと手続き（メソッド）を一つにまとめたものがクラス
  - Cの構造体に関数も足したものとイメージすれば良い
- クラスを元にして、実際にメモリ空間を割り当て（実体化させ）たものがオブジェクト
  - クラスが設計図で、それを元にして作った製品がオブジェクトとよく言われる

# カプセル化

- データとメソッドをまとめたおかげで、それを外から使う人は、その中の詳細を知らなくても良い
- 外部に提供されているメソッドだけ使用すれば、いい感じに中のデータを書き換えてくれる
- 外からはデータが直接見えなくなる (**カプセル化**)



# ポリモーフィズム

- あるオブジェクトに対して、同じメソッドを呼んだとき、そのオブジェクトの実態に応じて異なるふるまいをすること
- いろいろな場面で使える性質 例：
  - DBを使ったサービスのテストをしたい！本番DBでテストするのは怖いので、とりあえず仮のDBに置き換えてテストだ！
    - DBを叩く側はコマンドを変更していない（テスト用に叩くコードを変えるとそこがバグるかも）
    - でもDB自体の振る舞いは、本番用とテスト用で異なる（**ポリモーフィズム**）

# 継承

- ある2つのクラスA, Bの関係性が「B is a kind of A」を満たすとき、BをAを元にして設計することができる
  - Aを元にして、差分だけをB側で実装するイメージ
- Aを派生元クラス（Base Class）、Bを派生先クラス(Super Class)という
- 実装が省けるのが魅力
- ポリモーフィズムの実装例によく使われる

# インターフェース (1)

- 「規格」だと思えると理解が早い
  - 規格に準拠しているものは、その中身の詳細を知らずに利用することが可能（例：単3電池, 標準入出力, OSI参照モデル）
- クラスが満たすべき条件を規格として定義 例:
  - この名前のこういう型の変数が定義されてること
  - この名前でこういう型の引数を受け取る関数が定義されてること
- 規格を満たすようなクラスは皆、インターフェースで条件として設定したメソッドやフィールドが存在することが保証できる

# インターフェース (2)

- 2つの違う概念があるとき、その間を直接つなぐと互いを知らないとプログラミングできなくなる
- 間に一個「規格」を挟んで、使用する側は規格の情報を元に使うようにする。使用される側は規格に準拠した実装にする。
  - こうすると、使用する側が使用される側の詳細を知らなくても、互いに「規格」だけ気にしておけば良い
  - 規格に準拠している実装であるならば、いつでも"替え"が効く
- 2つの異質なものの間に挟まるモノ、それはすなわちインターフェース

# インターフェース (3)

- 利点

- 変更に強い。プログラムの設計を疎結合にできる
  - 規格さえ変わらなければ、中身の実装が変わったところでどうということはない
- ポリモーフィズムできる
- テストが容易
- 一つのクラスが複数のインターフェースを実装することが可能

# 継承とインターフェースの違い

- どちらもポリモーフィズムを実現するための手段として利用可能
- 継承
  - (+) 実装を省略できる
  - (-) 派生元クラスが変更されるとそれが子孫まで伝わってバグる
  - (-) 2つ以上のクラスから派生できない。ひし形継承の問題
- インターフェース
  - (-) クラスごとに全部実装が必要
  - (+) 疎結合で変更につよいシステムが作れる
  - (+) 2つ以上のインターフェースを実装したクラスも実現可能

# Functional Programming

ここからはjavascriptです。ES6を前提にさせてください。

# 処理の一部を変える

```
function jikanKeisoku() {  
  const startTime = getTime();  
  
  // ここに任意の処理を挟みたい  
  
  const endTime = getTime();  
  return endTime - startTime;  
}
```

`jikanKeisoku()` を汎用的に（つまり `jikanKeisoku()` の中身を書き換えずに、どんな処理でも処理時間を計測できるように）作りたい場合、どうしたらよいか。



# 関数を渡す

処理を関数にラップして、その関数そのものを引数として渡せば良い。渡した関数を述語（predicator）というときもある。

```
function jikanKeisoku(func) { // func: predicator
  const startTime = getTime();
  func(); // 外から与えられた関数を実行
  const endTime = getTime();
  return endTime - startTime;
}
function f() { // 計測対象の関数の定義
  // なんか重い処理
}
jikanKeisoku(f); // 計測実行
```

# javascript で用意されている関数

- 配列に対して (任意の操作をして) 新たな配列を作ったり、値を求めたりすることが結構ある
- javascript では標準でそのような操作を行う関数が用意されている。

# Array.prototype.map

配列の各要素に対して、関数を適用して得られた要素からなる配列を作る

```
function dbl(x) { // 引数で与えられた値を2倍返す doubleのdbl
  return 2 * x;
}
const arr = [1, 2, 3];
console.log(arr.map(dbl)); // [2, 4, 6]
```

# Array.prototype.filter

配列の各要素に対して、関数を適用してtrueとなる要素だけからなる配列を作る

```
function judge(x) { // 奇数のみtrue
  return x % 2 == 1;
}
const arr = [1, 2, 3];
console.log(arr.filter(judge)); // [1, 3]
```

# Array.prototype.reduce

2引数関数fと初期値を受け取る。

- 初期値と配列の1つめの要素をfに入れ、結果1を得る。
- 結果1と配列の2つめの要素をfに入れ、結果2を得る。
- ... これを最後の要素まで行い、最終結果を返す

```
function acc(a, b) { //accumulatorのacc. 今回は積を返す
    return a * b;
}
const arr = [1, 2, 3];
console.log(arr.reduce(acc, 1)); // 1は初期値。1*1*2*3の6が返る。
```

# アロー関数

- javascriptでは、式の途中で関数の名前を定義せず、実装だけを記述して定義することができる（アロー関数）。

```
const arr = [1, 2, 3];  
console.log(arr.map((x) => {return 2 * x;}));
```

`(x) => {return 2 * x;}` の部分がアロー関数。

```
console.log(arr.map(x => 2 * x));
```

ともかける。

# 非同期プログラミング

- ネットワークを隔てた処理・時間のかかる処理を普通の関数にして呼び出すと、しばらく制御が返ってこない。
  - 同期プログラミング
- とりあえず処理の開始を予約して、終わったときにしてほしいことも渡しておく。
- 呼び出した本人はすぐに自分の作業に戻ることができる
  - サーバーの処理向き
  - GUIプログラミングでも頻出
- -> **非同期プログラミング**

# 関数を渡すことができる

- 「終わったときにしてほしいこと」をどうやって渡す？
- 関数で渡せば良い

```
setTimeout(() => {alert("hello");}, 1000);
```

1000ミリ秒後に hello とアラートが出る。

しかし、`setTimeout` の呼び出しにかかる時間は一瞬（処理の"登録"だけを行えば十分）

スグに次の行の実行に移れる。非同期プログラミングすごい。幸せ。



# 問題

- 後にやってほしいことの中に、非同期な処理が含まれていたら・・・？例外が起きたら？
  - それも関数を与えて非同期プログラミングすれば良さそう
  - 実際それで実装できる
    - しかしそれを繰り返していくうちにネストはどんどん深くなってメンテしにくいプログラムになってしまう
  - どうするか
    - Promise: 非同期処理を扱う専門のクラス。
    - async/await: Promiseと連携。文法として組み込まれていてシンプルに非同期プログラミングできる。

# Promise

- web 概観講座で！

# async/await

- web 概観講座で！