2018
春号

百萬石
Hyakumangoku Vol.50

Published by

部長挨拶

mizdra

mizdra@mma.club.uec.ac.jp

「百萬石」をお手に取っていただき、ありがとうございます。弊誌 (以下、部誌) は、電気通信大学の公認サークル MMA が新歓期 (春号) と弊学が開催する調布祭時期 (秋号) の年 2 回発行・頒布しているものです。MMA とは、"Microcomputer Making Association"の頭文字に由来しており、1975 年に創立されたサークルです。現在では計算機に関連することを中心に、ネットワークやセキュリティなどの分野でも活動しています。部誌には、MMA の部員が各々に行なっている活動を記事にしたものが掲載されています。内容は多岐に渡り、高度なプログラミングの話から旅行記など、MMA の活動の自由さが感じられるものとなっております。

さて, この「春号」は新歓期間中に配布されるので, きっと多くの新入生の方に御覧頂いていることかと思います. ですからここでは新入生向けにサークルに興味を持ってもらえるよう, ここ最近のサークルの活動を少しだけ紹介しようと思います.

ここ数年, 部員の何名かが全国規模・世界規模の大会に出場し, 素晴らしい成果を残しています. 情報セキュリティ技術を競う CTF という分野においては中国のセキュリティ企業が主催する WCTF 2017 にて 2 位入賞, Google が主催する Google CTF にて 5 位, 台湾で開催されている世界最大規模の HITCON CTF 2017 Final では 6 位でした. また, サーバネットワークのトラブルシューティングや運用技術を競うコンテスト[*1]の ICT トラブルシューティングコンテストや Tokyo Westerns CTF 3rd 2017, CODE BLUE CTF 2017 では大会の運営に部員が関わっています.

勉強会やイベントの開催も行っています. 2017 年度の勉強会は Scala Collection Library Code Reading [*2], CTF 勉強会の 2 つを開催しました. Dentoo.LT は年に 3, 4 回開催していて, 次回で 20 回目の開催を迎えます. LT の様子を生放送で配信しており, YouTube にてアーカイブが保存されています[*3]. 是非ご覧ください.

以上がサークルの活動の紹介となります. MMA に興味を持って頂けたでしょうか. サークルに興味を持って頂けた方, 部室を見学したい方, サークルについてもっと知りたい方は是非サークル棟の 2F 左奥の MMA 部室までお越し下さい[*4]. 勿論, 入部希望者も歓迎です. 新入生の皆さんの訪問をお待ちしています.

2018 年 4 月吉日

[*1] 人材という“ レガシー ”を未来へ紡ぐ ICT トラブルシューティングコンテスト（と、たくさんの“ 物語 ”） | さくらのナレッジ: https://knowledge.sakura.ad.jp/8048/ より引用

[*2] Scala のコレクションライブラリのコードを読む会

[*3] https://www.youtube.com/c/dentoolt

[*4] 新歓期間中は部室ではなく西 5 号館 209 教室に居ます.

百萬石 Vol.50

Hyakumangoku Vol.50

2018 Spring

MMA

目次

第 1 章

新入生に捧ぐ!!必修 Linux コマンド集 &シェル操作入門

akky

新入生の皆さんご入学おめでとうございます. この記事では新入生の皆さんがこの 1 年で授業‥や開発で必要になる・役に立つであろう Linux コマンドとシェル[*1]操作を紹介していきます.

1.1 必修 Linux コマンド集

1.1.1 cd コマンド

概要

今いるディレクトリを変更するコマンドです. change directory の略. ディレクトリというのは Windows や Mac でいうフォルダみたいなもので, ディレクトリの中にはファイルやさらにディレクトリを入れて階層構造を作ることが出来ます. ディレクトリを作成し, その中にソースコードを記述しながらプログラミングをするため cd コマンドを非常によく使います.

使用例

cd ./<移動先ディレクトリ名>

```
$ cd ./hello       # 直下のhello ディレクトリに移動
$ cd hello         # 上のはこれでも代替可能
$ cd ../           # 1階層上のディレクトリに移動
$ cd hello/world   # 2階層下のworld ディレクトリに移動
$ cd               # ホームディレクトリに移動
$ cd foo           # 存在しないディレクトリに移動しようとすると
-bash: cd: foo: そのようなファイルやディレクトリはありません
  # 怒られた
```

[*1] 本記事では bash を対象にしています.

補足

- . は今いるディレクトリ (カレントディレクトリ) を表し, いちいち入力するのが面倒なので省略することが出来ます.
- .. は 1 階層上のディレクトリを表します.
- 区切り文字/は日本語環境では￥のことがあります.
- cd 単体でホームディレクトリに移動できます. ホームディレクトリとはシェルにログインしたとき, 最初にいるディレクトリのことです. よくわからないディレクトリに行ってしまった時使うと便利です.

1.1.2 pwd コマンド

概要

カレントディレクトリへの絶対パスを表示するコマンドです. print working directory の略. パスというのはあるディレクトリへの道筋のことです. 例えばカレントディレクトリの直下に hello ディレクトリ, さらにその直下に world ディレクトリがある場合, カレントディレクトリから world ディレクトリのパスは./hello/world となります. 特にカレントディレクトリからのパスのことを相対パスといいます. 一方でルートディレクトリ/からのパスのことを絶対パスといいます.

使用例

`pwd`

```
$ pwd          # カレントディレクトリへの絶対パスを表示
/home/akky     # 絶対パスが表示された
```

補足

- 今いるディレクトリがどこかわからなくなった時によく使います.

1.1.3 ls コマンド

概要

ディレクトリに含まれるファイルやディレクトリを表示するコマンドです. list の略. ls コマンドには有用なオプションがたくさんあります. オプションはコマンドの後ろに付けることで, 動作を変えることが出来る追加情報です. 書式は`-`[**アルファベット１文字**] または`--`[**英単語** or **語句**] となっています. よく使うのは`-a`, `-l`, `-t` オプションです.`-a` オプションは全て表示, `-l` オプションは詳細情報表示, `-t` オプションは更新時間の新しい順に表示です.

使用例

ls［**オプション**］<**ディレクトリ名**>

```
$ ls          # カレントディレクトリに含まれるファイルやディレクトリを表示する
Maildir  dev  # 表示された
$ ls dev      # 直下のdev ディレクトリ内を表示する
booklet2018a  # 表示された
$ ls -a       # カレントディレクトリに含まれるファイルやディレクトリを全て表示する
.   .bash_history  .bashrc   .muttrc      .profile  .viminfo  dev
..  .bash_logout   .forward  .procmailrc  .ssh      Maildir
$ ls -l       # ファイル・ディレクトリの種類とパーミッション，所有者，更新日時など詳細情報も表示
rwx------ 5 akky akky 4096  4月  9  2017 Maildir
drwxr-xr-x 3 akky akky 4096  3月 25 17:30 dev
$ ls -t       # 更新日時の新しい順に表示する
dev  Maildir  # 最初のls と順番が変わった
$ ls -alt     # オプションを組み合わせることも可能
drwxr-xr-x   5 akky akky  4096  3 月 25 19:45 .
-rw-------   1 akky akky  2805  3 月 25 18:47 .bash_history
-rw-------   1 akky akky  3710  3 月 25 17:40 .viminfo
drwxr-xr-x   3 akky akky  4096  3 月 25 17:30 dev
drwx------   2 akky akky  4096  3 月 23 00:11 .ssh
drwxr-xr-x 415 root root 12288  3 月  8 01:07 ..
drwx------   5 akky akky  4096  4 月  9  2017 Maildir
-rw-r--r--   1 akky akky    42  4 月  9  2017 .forward
-rw-r--r--   1 akky akky   220  4 月  9  2017 .bash_logout
-rw-r--r--   1 akky akky  3392  4 月  9  2017 .bashrc
-rw-r--r--   1 akky akky    79  4 月  9  2017 .muttrc
-rw-r--r--   1 akky akky   130  4 月  9  2017 .procmailrc
-rw-r--r--   1 akky akky   675  4 月  9  2017 .profile
  # 全ての詳細情報が新しい順に表示された
```

補足

- -l オプションを使った際 1 列目のブロックに表示されるのがファイル・ディレクトリの種類とパーミッションです.
- ファイル・ディレクトリの種類は-がファイル, d がディレクトリを表します.
- パーミッション[*2]とはファイル・ディレクトリの中身を閲覧 (r), 書き込み (w), 実行 (x) する権限のことです.
- パーミッションは 9 個の英数字で表され, その後は 3 つ区切りで所有者, 所有グループ, その他の権限を表します.
- 例えば.ssh ディレクトリのパーミッションは `rwx------`と所有者のみに割り当てていますが, これは.ssh には他の人に見られたくない情報 (暗号鍵) を入れることがあるからです.

[*2] パーミッションは chmod コマンドで変えることが出来ます. 詳しくは 1 学期のコンピュータリテラシーの授業で出てきます.

1.1.4 less コマンド

概要

ファイルの中身を表示するコマンドです. 1 行ずつスクロールすることが出来るので, 行数が多い大きなファイルを閲覧するときに使います. j で下, k で上に移動できます. 終了したい時は q と打てば元のシェルに戻ります.

使用例

`less` **<ファイル名>**

```
$ less akky.tex  # 行数が多いファイルを閲覧する
```

補足

- 行数がそれほど多くなく, 1 画面に収まるぐらいの小さいファイルを閲覧する時は cat コマンドが便利です.
- cat コマンドの使い方は `cat` **<ファイル名>**です.

1.1.5 mkdir コマンド

概要

ディレクトリを新規作成するコマンドです. make directory の略. ファイルをまとめるのに便利なのでよく使います. -p オプションを使えば, 存在しないディレクトリ下にもディレクトリを作成することが出来ます.

使用例

`mkdir` ［**オプション**］ **<ディレクトリ名>**

```
$ mkdir hello       # 直下にhello ディレクトリを新規作成
$ cd hello          # hello ディレクトリに移動
$ mkdir world       # hello ディレクトリの下にworldディレクトリ を新規作成
$ ls
world               # 確かにworld ディレクトリが作成された
$ mkdir uec/mma     # 存在しないuec ディレクトリの下にmma ディレクトリを作成
    しようとすると…
mkdir: ディレクトリ `uec/mma' を作成できません: そのようなファイルやディレク
    トリはありません
  # 怒られた
$ mkdir -p uec/mma # こんな時はp オプションを使う
$ cd uec/mma
$ pwd
/home/akky/hello/uec/mma
  # 確かにhello ディレクトリ下にuec ディレクトリ, その下にmma ディレクトリ
      が作成された
```

補足

- 複数のディレクトリを作成するときは, ディレクトリ名の間に半角スペースを入れます.
- 例: `$ mkdir cat dog`

1.1.6 cp コマンド

概要

ファイルやディレクトリをコピーするコマンドです. copy の略. コピー元が消去されないため安全にファイルやディレクトリの移動を行えます. ただしコピー先に同名のファイルがある際は上書きされてしまうので注意が必要です. いきなり上書きされないようにするには`-i` オプションで上書きの確認を出すようにします. またディレクトリをコピーする際は`-r` オプションが必要です.

使用例

cp［**オプション**］**<コピー元名> <コピー先名>**

```
$ cp -r uec/mma  # uec/mma ディレクトリをカレントディレクトリにコピー
$ ls             # 確認
mma  uec  world  # uec ディレクトリがコピーされている
$ ls uec         # uec ディレクトリ下を確認
mma              # uec ディレクトリ下にmma ディレクトリもコピーされている
```

補足

- 作成されたかどうか ls コマンドで確認していますが, 階層構造を分かりやすく表示する tree コマンド[*3]を使って確認するのがおすすめです.

1.1.7 mv コマンド

概要

ファイルやディレクトリを移動するためのコマンドです. move の略. cp では移動元は削除されませんが, mv では削除されます. そのため移動先で変更を加える際は気をつけましょう. この性質を利用してファイルやディレクトリ名の変更する際にも使えます.

使用例

mv **<移動元名> <移動先名>**

```
$ ls
mma  uec world
$ mv world ../              # world ディレクトリを1 階層上に移動する
```

[*3] 今回は上手く出力結果を LaTeX に貼れなかったので, ls コマンドで確認しています.

```
$ ls                          # 確認
mma  uec                      # world ディレクトリはなくなった
$ ls ../                      # 1 階層上を見てみる
Maildir  dev  hello  world    # world ディレクトリが移動されている
$ mv uec uec18                # uec ディレクトリをuec18 ディレクトリに名前変更
$ ls
mma  uec18                    # 変更された
```

補足

- 複数のファイルやディレクトリを移動するにはスペースで区切ります.
- 例: `$ mv cat dog animals  # cat, dog` **ディレクトリを** `animals` **ディレクトリへ移動**

1.1.8 rm コマンド

概要

ファイルやディレクトリを削除するためのコマンドです. remove の略. 削除したものを復活させるのは大変なので, 慎重に使いましょう. ディレクトリを削除する際には`-r` オプションを付与します.

使用例

`rm` ［**オプション**］ **<削除対象名>**

```
$ ls
mma  uec18
$ rm -r mma          # mma ディレクトリを削除
$ ls
uec18                # 消えた
$ mkdir a ab abc     # ディレクトリを作成
$ ls
a  ab  abc  uec18    # 作成された
$ rm -r a*           # aから始まるものを全て削除
$ ls
uec18                # 消えた
```

補足

- *はワイルドカードといい, 任意の文字列を含むものを操作対象にできます.

1.1.9　必修コマンドまとめ

この記事で紹介したコマンドを以下にまとめます．[*4]

表 1.1　必修コマンドまとめ

コマンド	動作	オプション
cd	ディレクトリ移動	-
pwd	作業ディレクトリパス表示	-
ls	ディレクトリ表示	a, l, t
less	ファイル表示	-
mkdir	ディレクトリ作成	p
cp	ファイル・ディレクトリ複製	r
mv	ファイル・ディレクトリ移動	-
rm	ファイル・ディレクトリ削除	r

[*4] 今回の記事で紹介したコマンド・オプションはほんの一部分です．オプションの項目で-と書いていますが実際にはたくさんのオプションがあります．

1.2 シェル操作入門

1.2.1 カーソル移動

概要

シェルでのカーソル移動は横矢印キーまたは control[*5] と b(左移動), f(右移動) を押して行います. さらに先頭には control と a, 末尾には control と e キーで移動できます.

1.2.2 Tab 補完

概要

コマンドやパスを入力する際 Tab キーを押すと候補が 1 つの場合は補完入力され, 候補が複数個ある場合は Tab キーをもう 1 度押すと候補が表示されます. 入力の手間を省くことが出来て, スペルミスも無くせるので非常に便利です.

使用例

```
$ mkd           # Tab を押す
mkdep  mkdir    # コマンドの候補が出る
$ cd            # Tab を押す
ssh/    .vim/    Maildir/ dev/     hello/   world/
# パスの候補が出る
```

1.2.3 過去コマンドをたどる

概要

上矢印キー又は control と p を押すことで直前のコマンドをたどって実行することが出来ます. 新しい方向へは下矢印キー又は control と n を押すことでたどれます.

使用例

```
$   # 上矢印又はctrl-p を押す
cd  # 直前のコマンドが表示された
```

1.2.4 履歴探索モード

概要

control と r を押すことで履歴探索モードになり, 過去利用したコマンドの履歴を検索して実行することが出来ます. 前述の過去コマンドをたどる方法では, かなり前に実行したコマンドを実行するには何度

[*5] 普通のキーボードではデフォルトの control キーの位置は押しにくいところにあるので, caps lock キーを control キーに変更するのがおすすめです.

もキーを押さなくてはいけませんが, 履歴探索モードでは検索することが出来るので効率よく履歴をたどれます.

使用例

```
$                              # ctrl-r を押す
(reverse-i-search)`':          # 履歴探索モードになる
(reverse-i-search)`c': cd      # c と打つとcd コマンドの履歴が出てきた
```

1.3　まとめ

いかがだったでしょうか. Linux の世界はとても奥深いため, この記事で紹介できなかったトピックはたくさんあります. すべてのコマンドを覚えるのは不可能ですが, 利用頻度が高い基本的なコマンドから覚えていきましょう. ネットには分かりやすいコマンドの記事が沢山公開されているので, 興味がある人は調べてみましょう.

参考文献

[1] [改訂第 3 版] Linux コマンドポケットリファレンス
[2] 仕事を 10 倍速くするコマンドライン操作 Tips
https://heartbeats.jp/hbblog/2010/03/10tips.html

第 2 章

QWERTY 配列を捨てて Dvorak 配列へ

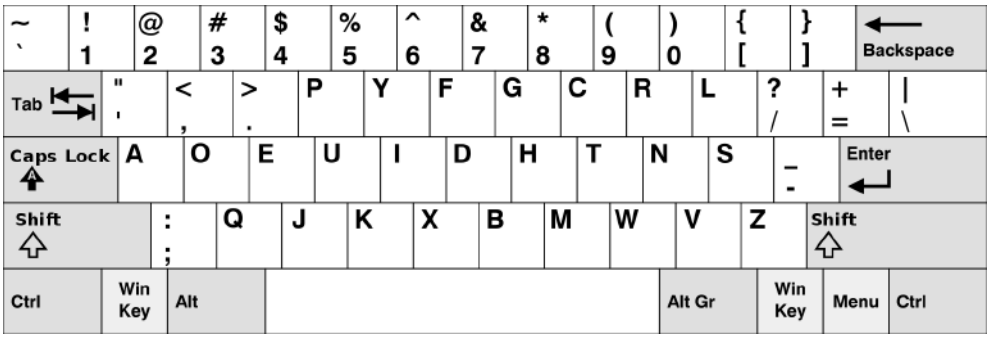
kyo1

大学に入ってからキーボードを使用することが多くなり，QWERTY 配列を使用するストレスが溜まり我慢できなくなったので Dvorak 配列へと変更したので，その感想を述べたり，どのように変更するのかをまとめてみようと思います．

2.1　キー配列の種類

まず自分は何のキー配列を使用しているか確認してみましょう．Dvorak 配列?　なにそれ? おいしいの? という人はおそらく QWERTY 配列を使用しているでしょう．上から 2 段目の英字が左から「QWERTY」となっていることから QWERTY 配列と呼ばれているそうです．しかし，QWERTY 配列は自分にとっては使いづらい配列です．使いづらいなら配列を変更してしまいましょう．ということで，メインは Dvorak ですが他のメジャーなキー配列を紹介するので好みの配列を見つけましょう．

2.1.1　Dvorak

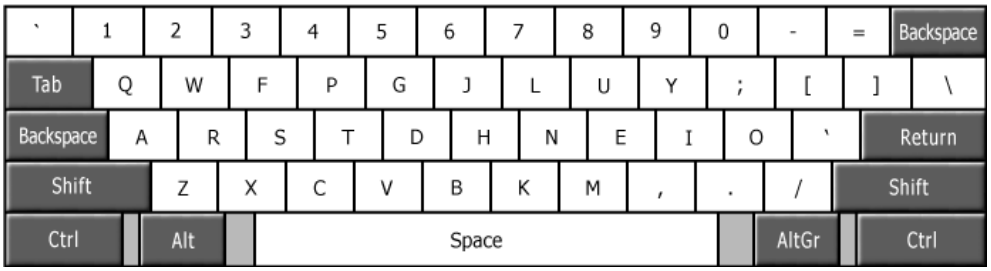


この配列は心理学者の August Dvorak によって開発されました．この配列の特徴的な部分としては，全ての母音を左手で入力できることです．これにより，右手で子音を，左手で母音を入力することができ，交互に入力することができます．

問題点としては，`ls` を入力するときに右小指を酷使すること，日本語を入力するとき (kya, kyu 等) に

左手を酷使することです.

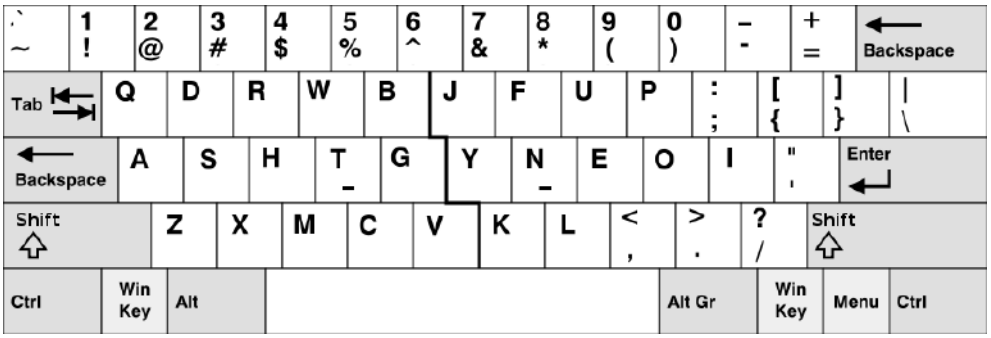
2.1.2　Colemak

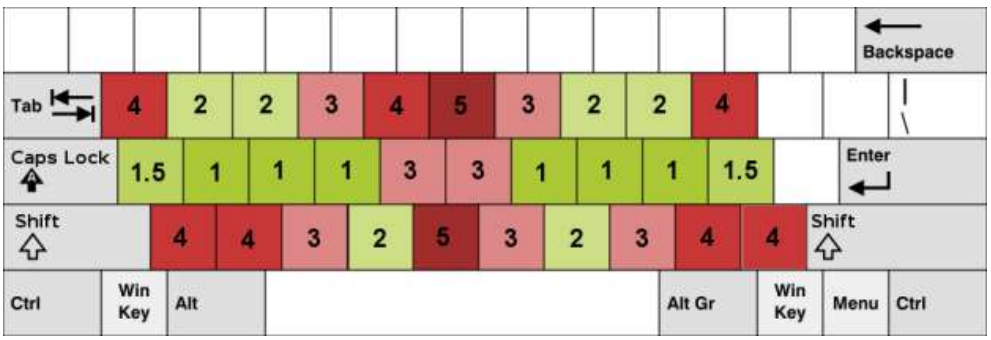


QWERTY 配列を改良したもので，QWERTY の代替かつ Dvorak の代替として開発されました．特徴的な部分としては，よく使うキーを中段に置くことにより，入力しやすくしています．また，QWERTY 配列を改良したということもあり，使いやすく移行しやすいレイアウトです．

問題点としては，「HE」という英語では 2 番目に多い文字列が打ちにくいというところです．

2.1.3　Workman





Workman は，Dvorak と Colemak の問題点を解決した配列です．各キーの位置をタイプする難易度に応じて点数付け (高いほうが悪い) することにより，打ちにくいキーを捨てるという考えで開発されました．難易度付けの基準の一部として，人差し指の横移動は負荷が高い (例として Colemak での「HE」) ということと，指関節は曲げ伸ばしできるので縦移動がしやすい，というものがあります．

問題点としては左手薬指の使用頻度が他配列と比べて高いことです．

2.1.4 Norman

紹介した中では最もモダンな配列です．利点としては，Norman は QWERTY よりも中段からの移動距離が 44 %短いということ，Workman で使用しているキーのスコアも高いということです．総合的には，紹介した配列の中で最も優れていると主張しています．

2.2 なぜ Dvorak か

著者は上で述べたように Dvorak 配列を使用しています．理由としては，交互に入力することができるからというのと，DvorakJP を利用することにより，か行や–yu 等の打ちにくいキーを打ちやすく改善することができるからです．

2.3 Dvorak を Mac に導入しよう

著者は Mac を使用しているので Mac での使用方法を紹介します．

Mac には標準で Dvorak が入っています．これによって楽に導入できます．

2.3.1 英語

1. システム環境設定を開く
2. システム環境設定のキーボードを選ぶ
3. 入力ソースの画面で「+」を押す

4. 英語のところから Dvorak を選択する

2.3.2 日本語 (DvorakJP を使用しない方法)

1. システム環境設定を開く
2. システム環境設定のキーボードを選ぶ
3. 入力ソースの画面で日本語を選択する
4. 設定を下の方にスクロールすると，「英字のレイアウト」があるので，デフォルトでは「U.S.」になっているがこれを「Dvorak」に変更する

2.3.3 日本語 (DvorakJP を使用)

1. Google 日本語入力をインストールする
2. あとは https://github.com/shinespark/dvorakjp-romantable に従って設定する

これで使えるようになっています.

2.4 Dvorak を使ってみよう

導入したら実際に使ってみましょう．いろいろなタイピングサイトがありますが，初心者には「The Typing Cat」 (http://thetypingcat.com/) というサイトがおすすめです．このサイトでは色々な配列のチュートリアルがあるので QWERTY 配列はもちろん，今回紹介した配列は全てあります.

2.5 Dvorak に移行した感想

Dvorak 配列を使用することにより，QWERTY 配列の頃と比べるとかなり楽になった気がします．しかし, それに伴って Vim のキーバインドがバラバラになってしまい, ノーマルモードの時だけ QWERTY 配列に戻しても，QWERTY 配列と Dvorak 配列がごちゃごちゃになってまともに使えなくなりました. その為泣く泣く Emacs に移動しました．ですが，総合的に見ると入力のときのストレスがかなり減ったので Dvorak 配列に変更して良かったです.

2.6 終わりに

皆さんはどの配列が気に入りましたでしょうか．配列を変更すると友達に入力してもらうときに QWERTY 配列に戻す必要があったりしますが，タイピングを快適にするためには仕方ないと思いましょう．また，今回紹介した配列では満足できず，Google 先生に聞いても好みの配列が見つからないときには自作するのも手です.

それでは皆さん，良いタイピングライフを!

参考文献

[1] Dvorak Wikipedia, https://ja.wikipedia.org/wiki/Dvorak 配列
[2] DvorakJP, http://www7.plala.or.jp/dvorakjp/
[3] Colemak Keyboard Layout, https://colemak.com/
[4] Workman Keyboard Layout, http://workmanlayout.org/
[5] Norman Keyboard Layout https://normanlayout.info/

第 3 章

How to Which?

hogas

Linux における `which` コマンドの中身を見てみようの回です．便利ですが，どう実現しているか，どういうときに使えたり使えなかったりするのか，知っておくとさらに便利になると思います．

3.1 What is Which?

`which` コマンドとは，「コマンドの場所をフルパスで表示する」コマンドです．実例を挙げると，

```
$ which ls
/bin/ls  # ここに存在
```

というように，「コマンドの実体であるファイルの存在する場所」を絶対パスで示してくれます．

でも，実は，これは厳密な説明ではありません．以下でだんだんわかっていきます．

3.2 Which is Which?

何より，`which` コマンド自体は，どこにあるのでしょうか．こういうときは，`which` コマンドが便利です ♡

```
$ which which
/usr/bin/which
```

場所がわかったので，これの中身を見ていきましょう．

3.2.1 どちらかが場所

コマンドは，外部コマンドとしてファイルで存在するものが多いですが，内部コマンドとしてシェルに組み込まれているものもあります[*1]．`which` は外部コマンド[*2]なので，シェルの種類に関係なく使えます．シェル組み込みコマンドには，実体のファイルが無いので，場所は表示されません．

[*1] 詳しい話は POSIX 標準とか様々なお話になるようです

[*2] zsh とか tcsh とかは `which` がシェル組み込みだったりするらしいので，以下は上手いこと見てください

```
$ which cd
  # 何も表示されない
$ type cd
cd is a shell builtin  #シェル組み込みなので……
```

内部コマンドの一覧は，bash では help コマンドでできます．コマンド自体については，type コマンド (シェル組み込み) で上のように確認できます．また，ここでは話題を Debian7 / bash に限ります (それぞれバージョン 3.16.7 / 4.2.37 で確認しました).

3.3 How to Which?

3.3.1 which の中身

$ cat /usr/bin/which すると，以下のようになっています．

ソースコード 3.1　which コマンドの中身

```
#! /bin/sh
set -ef

if test -n "$KSH_VERSION"; then
        puts() {
                print -r -- "$*"
        }
else
        puts() {
                printf '%s\n' "$*"
        }
fi

ALLMATCHES=0

while getopts a whichopts
do
        case "$whichopts" in
                a) ALLMATCHES=1 ;;
                ?) puts "Usage: $0 [-a] args"; exit 2 ;;
        esac
done
shift $(($OPTIND - 1))

if [ "$#" -eq 0 ]; then
 ALLRET=1
else
 ALLRET=0
fi
case $PATH in
        (*[!:]:) PATH="$PATH:" ;;
esac
for PROGRAM in "$@"; do
 RET=1
 IFS_SAVE="$IFS"
 IFS=:
 case $PROGRAM in
  */*)
```

```
39     if [ -f "$PROGRAM" ] && [ -x "$PROGRAM" ]; then
40      puts "$PROGRAM"
41      RET=0
42     fi
43     ;;
44   *)
45     for ELEMENT in $PATH; do
46      if [ -z "$ELEMENT" ]; then
47       ELEMENT=.
48      fi
49      if [ -f "$ELEMENT/$PROGRAM" ] && [ -x "$ELEMENT/$PROGRAM" ]; then
50       puts "$ELEMENT/$PROGRAM"
51       RET=0
52       [ "$ALLMATCHES" -eq 1 ] || break
53      fi
54     done
55     ;;
56  esac
57  IFS="$IFS_SAVE"
58  if [ "$RET" -ne 0 ]; then
59   ALLRET=1
60  fi
61 done
62
63 exit "$ALLRET"
```

which コマンドも，多数のコマンドで構成されていることがわかります．

3.3.2 which 縦断 1800km

各所の動きを順に見ていきます．

```
1 #! /bin/sh
2 set -ef
```

1 行目は shebang(シバン，シェバン)，このファイル自体を処理するインタプリタの指定です．明示的に$ sh /usr/bin/which とせずとも，$ /usr/bin/which だけで sh が実行します．

2 行目の set はシェルのオプションを設定する内部コマンドです．-e オプションは「コマンドが 0 以外のステータスで終了した場合シェルも即座に終了する」，-f オプションは「パス名のワイルドカード (* や?) 展開を無効にする」というものです．前者は，想定しない例外などを防ぐためと考えられます．後者は，ファイル名などにワイルドカードの文字が含まれていたりする場合にそれをワイルドカードとして認識せず，想定しないパスへアクセスしないようにしているものと思います．

```
4 if test -n "$KSH_VERSION"; then
5   puts() {
6     print -r -- "$*"
7   }
8 else
9   puts() {
10    printf '%s\n' "$*"
11  }
12 fi
```

ここではのちのち使う表示用の関数を定義しています．ksh だけ (printf が無いのか) 場合分けされています．ksh の print コマンドは，デフォルトで最後に改行し，-r オプションでエスケープ文字 (\n など) を無視します．printf コマンドはシェル組み込みで，C 言語の printf 関数と同じように使います．つまりこの puts 関数は，(この関数の) 引数$*をとにかく表示し，最後に改行する，をします．

```
14 ALLMATCHES=0
15
16 while getopts a whichopts
17 do
18         case "$whichopts" in
19                 a) ALLMATCHES=1 ;;
20                 ?) puts "Usage: $0 [-a] args"; exit 2 ;;
21         esac
22 done
23 shift $(($OPTIND - 1))
```

ここで，which コマンドに与えられたオプションの有無を見ています．getopts というシェル組み込みコマンドは，getopts **対象文字（複数可）変数名** と使います．オプションのうち最初の文字について，「対象文字の場合，その文字を」「対象文字以外の場合，?を」変数名で指定した変数に格納してくれます (つまり 1 文字オプションのみ対応)．これを while [*3]でループすることで，全てのオプションを検査します．ここでは，

- -a オプションがあるときは変数 ALLMATCHES を 1 に
- それ以外のオプションがあるときは Usage... と表示してからステータス 2 で終了[*4]
- オプションが無いときは何もしない

という挙動になります．つまり which コマンドには，-a オプションがあることがわかりました (機能は後述)．getopts の効果は，実際に which コマンドに-a 以外のオプションを与えるとよくわかります．Illegal... は getopts のエラーメッセージです．

```
$ which -b ls
Illegal option -b
Usage: /usr/bin/which [-a] args
```

また，引数からオプション部分を除く (この後が楽になる) ために，shift コマンド (シェル組み込み) を実行します．これは引数をずらすコマンドで，いくつずらすかを (shift の) 引数に取ります．変数 OPTIND は，getopts によってセットされる変数で，"プログラム名〜オプション"の個数です．対して，which コマンドに与えられた引数にはプログラム名が含まれていないので，そこからオプション引数を除くには，$OPTIND - 1 だけずらせば OK です．なお，$((**算術式**)) で算術式の計算結果を取り出せます[*5]．

[*3] while はシェル予約語

[*4] 終了ステータスは，0 が正常，それ以外は異常終了を示すという決まり (後述の終了ステータス 1 と数値を変えているのは，終了した原因/箇所を特定できるように)

[*5] ドル記号 ($) 無し (((**算術式**))) だと，計算結果が 0 のとき 0，それ以外のとき 1 が取り出せます．bash の Man page の「ARITHMETIC EVALUATION」に詳しいです

```
$ which -a ls
  :     :
  :     1  ...... which コマンドに与えられた引数のうちオプションの個数( == 1)
  1     2  ...... OPTIND( == 2)
```

これで，which コマンドに与えられた引数には，探したいコマンド名だけが残りました．ここからそのコマンドを検索していきますが，ちょっと準備が必要です．

```
25 if [ "$#" -eq 0 ]; then
26  ALLRET=1
27 else
28  ALLRET=0
29 fi
```

まずは，探したいコマンド名が指定されていないときの例外処理をします．$#は，「which コマンドに与えられた引数の個数」を表します．コマンド名が指定されていない ($#が 0) 場合は変数 ALLRET に 1 を，ちゃんと指定されている場合は 0 をセットしておきます (後述しますが，変数 ALLRET は which コマンドの終了ステータスとなります)．

```
30 case $PATH in
31         (*[!:]:) PATH="$PATH:" ;;
32 esac
```

次に，変数 PATH [*6]の前処理ですが，この部分の意図は，僕もよくわかっていません．挙動としては，case [*7]条件分岐によって「変数 PATH の末尾にコロン (:) が **1 つだけ**ついているとき，2 つに増やす (それ以外のとき，何もしない)」というものです．一応，bash の Man page ($ man bash で同じものを見られます) によれば，

> PATH The search path for commands. (略) A zero-length (null) directory name in the value of PATH indicates the current directory. A null directory name may appear as two adjacent colons, or as an initial or trailing colon. (略)

ということで，つまり，変数 PATH において「2 つのコロン (::)」「先頭のコロン」「末尾のコロン」は空白のディレクトリ名であり「現在のディレクトリ」として扱われるという仕組みになっているようです．が，あまり関連がないように思えました……詳しい方は教えてください．

ここからやっと本題の処理に入ります．スコープが切れてしまいますが，少しずつ引用します．

```
33 for PROGRAM in "$@"; do
34  RET=1
35  IFS_SAVE="$IFS"
36  IFS=:
```

[*6] 変数 PATH は環境変数で，シェルが外部コマンドを検索する先の絶対パスが，コロン区切りで示されています

[*7] case はシェル予約語

$@は，which コマンドに与えられた引数が**配列で**格納された変数です．引数を表す変数には，1 つの値として空白区切りで格納された$*もありますが，これで同じように for PROGRAM in "$*" などとしてしまうと想定した動作をしません[*8]．ダブルクオートは，空白や改行などでのさらなる単語分割が起こらないための安全策です．

ここで残っている引数は**探したいコマンド名**だけなので，それらが順に変数 PROGRAM に格納されつつ for [*9]ループが回ります．つまり which コマンドには複数のコマンド名を与えることができる，ということがわかりました (機能は後述)．

for ループの最初の 3 行について．まず，処理が成功したかを示すフラグとなる変数 RET に 1 をセットしておきます．

変数 IFS は，bash の環境変数で，bash の Man page によると

> IFS The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the read builtin command. The default value is "<space><tab><newline>".

つまり"Internal Field Separator"の略で，単語を区切る文字 (デフォルトでは，空白/タブ/改行) が指定されています．ここでは，変数 PATH を扱うためにコロン (:) に指定し直しています．また変数 IFS には既に値があるはずなので，処理後に元に戻せるよう，変数 IFS_SAVE に一時的にバックアップしています．

```
37 case $PROGRAM in
38  */*)
39   if [ -f "$PROGRAM" ] && [ -x "$PROGRAM" ]; then
40    puts "$PROGRAM"
41    RET=0
42   fi
43   ;;
```

ここで変数 PROGRAM つまり探したいコマンド名について，条件分岐があります．まずコマンド名の中にスラッシュ (/) が含まれる場合です．

[(大括弧開) は，見かけによらずシェル組み込みのコマンド[*10]で，] (大括弧閉) と組み合わせて条件式評価，ここでは if [*11]条件分岐に使います．-f オプション，-x オプションはそれぞれ，「ファイルが通常ファイルであるか」「ファイルが実行可ファイルである (実行権限がある) か」を評価します．よってここは変数 PROGRAM に示されたコマンド名が，**通常ファイル** かつ **実行可ファイル**として存在する場合に，そのコマンド名をそのまま表示し，変数 RET を 0 にセットします．スラッシュが含まれると (ファイル名でなく) パスなので，変数 PATH の検索 (後述) はせず，そのままファイルの有無のみを見る，という気持ちです．この辺りから，先に定義しておいた puts 関数を使っています．

[*8] もし$*を使うと， $ which foo bar baz と打ったとき，ループ 1 回目で"foo bar baz"という文字列がごっそり渡され，ループ 2 回目は来ません

[*9] for はシェル予約語

[*10] $ man [をするとわかるように，test コマンドと等価です

[*11] if はシェル予約語

```
44   *)
45    for ELEMENT in $PATH; do
46     if [ -z "$ELEMENT" ]; then
47      ELEMENT=.
48     fi
49     if [ -f "$ELEMENT/$PROGRAM" ] && [ -x "$ELEMENT/$PROGRAM" ]; then
50      puts "$ELEMENT/$PROGRAM"
51      RET=0
52      [ "$ALLMATCHES" -eq 1 ] || break
53     fi
54    done
55    ;;
56  esac
```

対してスラッシュが含まれない場合には変数 PATH を検索します．変数 PATH に含まれるパスたちを変数 ELEMENT に渡しつつ for ループをします[*12]．

まずは事前処理として，変数 ELEMENT が空，つまり「コロン 2 つ」「先頭コロン」「末尾コロン」のとき，カレントディレクトリ (.) にしておきます．それから，変数 ELEMENT で示されるパス直下に，コマンド名のファイルが**通常**かつ**実行可**で存在する場合，そのパスを表示し，変数 RET を 0 にセットして，**さらに**変数 ALLMATCHES が 1 のときすなわち-a オプションのとき break **しない** (逆に言えば，0(オプション無し) のとき break で for ループを抜ける) [*13]，とします．つまり，which コマンドは「コマンドの場所が変数 PATH の中で 1 つ見つかったら表示して終了」，-a オプションがあったら「(見つけたら表示しつつ) 変数 PATH が尽きるまで探し続ける」という挙動をすることがわかりました．

```
57  IFS="$IFS_SAVE"
58  if [ "$RET" -ne 0 ]; then
59   ALLRET=1
60  fi
61 done
62
63 exit "$ALLRET"
```

長かった旅路もラストスパート，事後処理です．まず変えてあった変数 IFS を元に戻します．ここで変数 RET が 0 でないとき，つまり見つからなかったとき，変数 ALLRET を 1 にセットします．for ループ内に変数 ALLRET を 0 に戻す処理は無いため，1 つでも見つからないコマンドがあったら 1 です．

which コマンドに与えられたコマンド名の数だけこれら (33〜61 行目) を行い，最後に変数 ALLRET をステータスとして終了すれば完了です．

3.4 What is Which? (reprise)

というわけで，which とは，「引数に与えられた全てのコマンド名についてそれぞれを変数 PATH にあるパスで検索し，(実行可ファイルで) 存在したらその絶対パスを表示する」「-a オプションがあった場合は，それぞれを変数 PATH にあるパス全てで検索し，(実行可ファイルで) 存在するすべての絶対パスを表

[*12] 先に IFS をコロンに変更しておいたので，ここでは for はコロンによって変数 PATH を分割します

[*13] || は論理和演算子で，「左辺が偽の場合のみ右辺を実行」のイディオム (cf. && は論理積演算子で，「左辺が真の場合のみ右辺を実行」イディオム)

示する」というコマンドでした．終了ステータスは，0(全てのコマンドが見つかった) または 1(見つからなかったコマンドがあった) または 2(オプションが正しくなかった) となります．内部コマンドなどのとき含め，変数 PATH のパスにコマンド名のファイルが存在しないときは，何も表示されません．

```
$ which type ip ls traceroute     # コマンドを沢山指定
/sbin/ip
/bin/ls
/usr/sbin/traceroute
$ which -a type ip ls traceroute  # PATHをとにかく全部探す
/sbin/ip
/bin/ip  # ........................ 増えた
/bin/ls
/usr/sbin/traceroute
/usr/bin/traceroute  # ............ 増えた
```

3.5 How was Which?

いかがでしたか．なんとなくでも which コマンドへの理解が深まっていれば幸いです．見ていくとわかる通り，which コマンドはその中の処理に他の外部コマンドを使っていません．もし複数の外部コマンド間で，互いに互いを使ってしまったら，循環参照により無限ループが発生するので，それはそうですね．

それにしても，which コマンドの中身のインデントがごちゃごちゃなのが気になります．なんとかならなかったんでしょうか…….

3.6 Why is Which?

ところで，何故 where でなく which なのでしょう？　tcsh には where コマンド (やってることはほぼ同じだけど) があるし「どこかで先に別のコマンドが where と命名されてしまったから」とか[*14]，あとは「変数 PATH のうち**どれ**が該当するのか，という意味だから」とか？　詳しい方は教えてください・パート 2 [*15]．

参考文献

[1] Nikkei Business Publications, Inc.,「【 set 】シェルのオプションを設定する」, http://tech.nikkeibp.co.jp/it/article/COLUMN/20060227/230881/，2018.

[2] it support sakura co., ltd.,「逆引きシェルスクリプト/getopts を利用して引数を取得する (bash ビルドイン)」，http://linux.just4fun.biz/?逆引きシェルスクリプト/getopts を利用して引数を取得する%28bash ビルドイン%29，2018.

[*14] でも tcsh では which コマンドはシェル組み込みのものが存在してしまっているし，あんまり関係ないような……

[*15] 昼下がりのサイコロ番組，懐かしいですね

第 4 章

WebAssembly 開発環境構築

mizdra

4.1 はじめに

最近 Web フロントエンド界隈で話題になっているキーワードに「WebAssembly(wasm)」があります. WebAssembly とはブラウザ上で動作することを目的とした低水準言語のことです. 事前にコンパイルされたバイナリ形式で高速に実行できるよう設計されているため, ブラウザ上で動く代表的な言語である JavaScript に比べて高速に実行できます[*1]. WebAssembly は以下を目標に定めて作成されています.

- 高速で、高効率であり、ポータブルである事 — WebAssembly のコードは共通ハードウェア対応環境[*2] を利用して異なるプラットフォーム間でネイティブ水準の速度で実行可能です。
- 可読性を持ちデバッグ可能である事 — WebAssembly は低水準のアセンブリ言語ですが、人間が読めるテキストフォーマットを持ちます（その仕様策定は終わっていないものの）。このフォーマットは人の手で読み書きできて、デバッグもできます。
- 安全である事 — WebAssembly は安全でサンドボックス化された実行環境上で動作するように設計されています。他のウェブ言語と同様に、ブラウザに対して same-origin および権限ポリシーの確認を強制します。
- ウェブを破壊しない事 — WebAssembly は他のウェブ技術と協調し、後方互換性を維持するように設計されます。

[*3]より引用

現状 C/C++ や Rust などの言語から WebAssembly へのコンパイルがサポートされています. 本文章では Rust を用いて WebAssembly の環境構築を行う方法を紹介します.

[*1] WebAssembly はなぜ速いのか| POSTD: https://postd.cc/what-makes-webassembly-fast
[*2] Portability - WebAssembly: http://webassembly.org/docs/portability/#assumptions-for-efficient-execution
[*3] WebAssembly のコンセプト - WebAssembly | MDN: https://developer.mozilla.org/ja/docs/WebAssembly/Concepts

4.1.1 本文章を読むにあたって

本文章では Rust/WebAssembly/JavaScript の書き方や仕様, 関連するツール・ライブラリなどが出てきます. WebAssembly の話に焦点を当てて解説していくため, 足りない知識は以下のサイトを参考に各自補完していって下さい.

- 言語を学ぶ
  - プログラミング言語 Rust: https://rust-lang-ja.github.io/the-rust-programming-language-ja/1.6/book/README.html
    - Rust の始め方, 書き方について丁寧に解説されています
    - Rust を初めて書くなら必読です
    - 日本語訳は原文よりも古いバージョンとなっているので英語が読めるなら原文を読むと良いです
  - 最速で知る！プログラミング言語 Rust の基本機能とメモリ管理【第二言語としての Rust】: https://employment.en-japan.com/engineerhub/entry/2017/07/10/110000
  - 実践的なアプリケーションを書いてみよう！ Rust の構造化プログラミング【第二言語としての Rust】: https://employment.en-japan.com/engineerhub/entry/2017/07/19/110000
  - Rust の標準ライブラリのドキュメント: https://doc.rust-lang.org/std
  - Rust Playground: https://play.rust-lang.org
  - Learn ES2015 · Babel: https://babeljs.io/learn-es2015
    - ES2015 で導入された構文や機能について解説されています
  - You Don't Know ES Modules: https://www.slideshare.net/teppeis/you-dont-know-es-modules
    - ES Modules について解説されています
  - Babel · The compiler for writing next generation JavaScript: https://babeljs.io/repl
- 言語の仕様書
  - ECMAScript® 2017 Language Specification: http://www.ecma-international.org/ecma-262/8.0/index.html
  - WebAssembly Specifications: http://webassembly.github.io/spec
- モジュールバンドラのドキュメント
  - Parcel Documentation: https://parceljs.org/getting_started.html
  - Webpack Documentation: https://webpack.js.org/concepts

4.1.2 開発環境について

以下は著者の開発環境です. 出来る限りこれらより高いバージョンのツール・ソフトウェアを使用するようにして下さい.

- Node.js v9.9.0
- npm v5.8.0
- rustc 1.24.1
- rustc 1.26.0-nightly
- cargo 0.25.0
- Mozilla Firefox v59.0.1
- Google Chrome v65.0.3325.162

4.2 WebAssembly 入門

それでは WebAssembly の入門から始めましょう! まずは WebAssembly の動作を理解しやすいよう, 引数として受け取った 2 つの数値の和を返す単純な関数 `add` を Rust で実装して, WebAssembly にコンパイルして JavaScript から呼び出してみます. まずは Rust のインストールを行います.

```
## Rust をインストール (cargo, rustc, rustup コマンドなどがインストールされる)
$ curl https://sh.rustup.rs -sSf | sh
$ cat ~/.cargo/env

## toolchain を更新
$ rustup update

## WebAssembly へのコンパイル機能を有効化
$ rustup target add wasm32-unknown-unknown
```

インストールが終わったら Cargo[*4]を用いて Rust のプロジェクトを作成しましょう.

```
$ cargo new --lib hello_world_wasm
$ tree hello_world_wasm
hello_world_wasm
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

Rust からコンパイルしたバイナリにはデフォルトで他の Rust プログラムからの利用する際に使われるメタデータなどが含まれています[*5]. これらのメタデータは WebAssembly では不要なので `crate-type` に`"cdylib"` を指定し, 削ぎ落とすようにしましょう. `/Cargo.toml` に以下を追加します.

```
// ...
[lib]
crate-type = ["cdylib"]
```

[*4] Rust のビルドシステム, 及びパッケージマネージャ.

[*5] rfcs/1510-cdylib.md at master · rust-lang/rfcs: https://github.com/rust-lang/rfcs/blob/master/text/1510-cdylib.md

準備が整ったのでコードを書いていきます．`/src/lib.rs` を次のように書き換えます．

```
#[no_mangle]
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

2 つの符号付き 32bit 整数を引数として受け取り，その和を返す関数です．`#[no_mangle]` はアトリビュートと言い，Java のデコレータのようにブロックやメソッドなどを修飾する構文です．`#[no_mangle]` では「Rust コンパイラに次の関数の名前をマングリングせずにコンパイルせよ」と指示します．これにより，JavaScript から `add` という名前で関数にアクセスできるようになります．また，関数を公開して外部から呼び出せるようにするため `pub` キーワードを付けています．

Rust のプロジェクトを WebAssembly にコンパイルするには次のコマンドを実行します．

```
$ cargo build --target=wasm32-unknown-unknown --release
```

`release` オプションにより最適化したバイナリを生成するよう指示しています．これは Rust の WebAssembly 向けコンパイルがまだ安定しておらず，最適化されていないバイナリにバグが含まれる可能性があるため付けています[*6]．時が経てば `release` オプションは不要になることでしょう．

コンパイルが成功すれば`/target/wasm32-unknown-unknown/release/hello_world_wasm.wasm` が生成されているはずです．早速これを JavaScript から実行してみましょう．`/index.html` を作成します．

[*6] 実のことを言うとつい最近この問題は解決されました（参考： https://github.com/rust-lang-nursery/rust-wasm/issues/1）．しかしながら念には念を入れて，ここでは `release` オプションを付けています．

```html
<!DOCTYPE html>
<html lang="ja">

<head>
  <meta charset="UTF-8">
  <title>Hello, World! on WebAssembly</title>
  <script>
    const wasm = './target/wasm32-unknown-unknown/release/hello_world_wasm.wasm'
    fetch(wasm)
      .then(response => response.arrayBuffer())
      .then(bytes => WebAssembly.instantiate(bytes, {}))
      .then(results => {
        console.log(results.instance.exports.add(1, 2))
      })
  </script>
</head>

</html>
```

ここで起こっていることを順に追っていきます.

1. Fetch API を用いて wasm ファイルを読み込む
2. `response.arrayBuffer` でファイルのデータをバイナリ配列に変換
3. `WebAssembly.instantiate` でバイナリ配列を WebAssembly コードとしてコンパイル・インスタンス化
4. WebAssembly インスタンスから `add` にアクセスし, 呼び出す

もしかしたらこの説明に疑問を持った方がいるかもしれません. 何故なら先程 Rust コンパイラを用いて Rust から WebAssembly にコンパイルしたにも関わらず, JavaScript 上で再度コンパイルをしているからです. これは WebAssembly があくまでブラウザ[*7]が理解できるフォーマットであり, そのままではそのブラウザが動いている OS やハードウェアなどのシステムが理解できるフォーマットではないためです. WebAssembly を実行するには最初にブラウザが WebAssembly をそのブラウザが動いている OS やハードウェアが理解できる機械語にコンパイルし, それから実行する必要があります. ブラウザと WebAssembly は, ちょうど Java でいうところの JVM とバイトコードの関係のようなものです.

さて, このコードを実際にブラウザで動かしてみます. 注意点として Fetch API は `file` URI Scheme[*8]をサポートしていないため, 任意の HTTP サーバで `index.html` と `hello_world_wasm.wasm`

[*7] 厳密にはブラウザだけでなく Node.js や組み込みシステムなど様々な環境で動作します.
[*8] `file:///path/to/file.ext` のようにローカルにあるファイルにアクセスするときに使う URI スキーマです.

を配信してファイルに http URI Scheme でアクセスできるようにする必要があります．ここでは npm パッケージの http-server を使用します．

```
## `npx` は npm にバンドルされているコマンドです
$ npx http-server .
Starting up http-server, serving .
Available on:
  http://127.0.0.1:8081
  http://192.168.0.14:8081
## HTTP サーバが立ち上がるのでブラウザから `http://127.0.0.1:8081` にアクセスする
```

ブラウザの開発者ツールのコンソールを開いて 3 が出力されていれば成功です!

4.2.1 WebAssembly から JavaScript の関数を呼び出す

WebAssembly から JavaScript の関数を呼び出す例も試してみましょう．今回は WebAssembly から JavaScript の Date.now 関数を呼び出してタイムスタンプを返す get_timestamp 関数を実装します．

/index.html の <script> タグの中を次のように編集します．

```
// 追加
const imports = {
  env: {
    date_now: Date.now,
  }
}
const wasm = './target/wasm32-unknown-unknown/release/hello_world_wasm.wasm'
fetch(wasm)
  .then(response => response.arrayBuffer())
  // `WebAssembly.instantiate` の引数に `imports` を追加
  .then(bytes => WebAssembly.instantiate(bytes, imports))
  .then(results => {
    const { add, get_timestamp } = results.instance.exports
    console.log(add(1, 2))
    // 追加
    console.log(get_timestamp())
  })
```

WebAssembly.instantiate の引数に WebAssembly 実行環境に渡したい関数が含まれるオブジェクトを指定します．env プロパティでネストしていることに注意して下さい．

次に /src/lib.rs に以下を追加します.

```
// ...
extern {
    fn date_now() -> f64;
}

#[no_mangle]
pub fn get_timestamp() -> f64 {
    unsafe {
        date_now()
    }
}
```

extern ブロックの中には Rust のコンパイラが他言語の関数を理解できるよう, 他言語の関数のシグネチャを書きます. Date.now 関数によって返される値は常に整数ですが, JavaScript の数値は全て IEE754 浮動小数点数なので date_now 関数の戻り値の型を f64 としています[*9]. また Rust ではデフォルトで他言語関数の呼び出しはアンセーフとみなされるので, 関数を呼び出す際は unsafe ブロックで囲って関数が安全であることをコンパイラに約束する必要があります.

コンパイルして実行してみましょう.

```
$ cargo build --target=wasm32-unknown-unknown --release
$ npx http-server .
```

ブラウザのコンソールにタイムスタンプが出力されましたでしょうか? システムのタイムスタンプが出力されるため, ページを更新する度に出力内容が変わるはずです.

4.2.2 Rust のサードパーティ製ライブラリの利用

最後に Rust のサードパーティ製ライブラリを利用してみます. /Cargo.toml に以下を追加し, Cargo にプロジェクトが tinymt クレートに依存していることを伝えます.

```
// ...
[dependencies]
tinymt = { git = "https://github.com/mizdra/rust-tinymt", tag = "0.1.0" }
```

/src/lib.rs に以下を追加します.

[*9] u64 とするとランタイムエラーが出ます.

```rust
// ...
extern crate tinymt;

use tinymt::tinymt32;

#[no_mangle]
pub fn rand() -> u32 {
    let param = tinymt32::Param {
        mat1: 0x8F7011EE,
        mat2: 0xFC78FF1F,
        tmat: 0x3793fdff,
    };
    let seed = 1;
    tinymt32::from_seed(param, seed).gen()
}
```

`extern crate` で `tinymt` クレートを利用することを Rust コンパイラに伝えています. `rand` 関数では `tinymt` クレートを利用して TinyMT という乱数生成方式で乱数を生成し, 得られた乱数を返しています[*10].

`/index.html` の `<script>` タグの中を編集し, JavaScript からこの関数を呼び出します.

[*10] 著者がポケモンの乱数調整に関するツール製作を趣味でやっているため, 乱数生成ライブラリを例に挙げました :P (参考: https://mizdra.hatenablog.com/entry/2016/12/01/235954)

```
const imports = {
  env: {
    date_now: Date.now,
  }
}
const wasm = './target/wasm32-unknown-unknown/release/hello_world_wasm.wasm'
const toUint32 = (num) => num >>> 0
fetch(wasm)
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.instantiate(bytes, imports))
  .then(results => {
    const { add, get_timestamp, rand } = results.instance.exports
    console.log(add(1, 2))
    console.log(get_timestamp())
    console.log(toUint32(rand()))
  })
```

toUint32 関数は JavaScript の数値を 32bit 符号無し整数として扱うためのトリックです．rand 関数は Rust のコードでは u32 型を返すことになっていますが，WebAssembly にコンパイルすると i32 型を返す関数へと変換されます[*11]．戻り値を u32 型で表した時に 2^31 未満であれば JavaScript 側で得られる値に変わりはありませんが，2^31 以上の場合は戻り値から 2^32 を引いた値が JavaScript 側で得られる値となります[*12]．今回は rand 関数の戻り値は u32 型で表した時に 2^31 以上となる可能性があるため，toUint32 関数を使って戻り値を 32bit 符号無し整数として扱っています[*13]．

それでは完成したプロジェクトをビルドし，実行してみましょう．

```
$ cargo build --target=wasm32-unknown-unknown --release
$ npx http-server .
```

ブラウザのコンソールの出力に 2545341989 が追加されていれば成功です!

4.2.3 本節のまとめ

これにて WebAssembly 入門は終了です．本節で学んだことを振り返ってみましょう．

[*11] この現象は WebAssembly を wast 形式と呼ばれる S 式ベースのテキスト表現へと変換すると確認できます．WebAssembly の仕様に u32 型が存在するのにも関わらず，このように敢えて i32 型へと変換する理由が書いてある文献を探してみましたが，見つけられませんでした．何か情報をお持ちの方がいれば教えてください...

[*12] この挙動は https://www.ecma-international.org/ecma-262/8.0/index.html#sec-toint32 に基づきます．

[*13] 関数の戻り値の型として u32 を期待しているのであれば，その戻り値の全てを toUint32 関数でラップしたほうが安全でしょう．手間ですが...

- コマンドを用いて地道に WebAssembly にコンパイルした
- WebAssembly をどのようにブラウザ上で実行するかを確認した
- WebAssembly から JavaScript の関数を呼び出した
- Rust のサードパーティ製ライブラリを使用した
- `u32` 型を返す Rust の関数を WebAssembly にコンパイルすると `i32` を返す関数に変換されることを確認し, その対処法を学んだ

本節で作成したプロジェクトは以下のリポジトリで公開しています.

- mizdra / hello_world_wasm · GitLab
  - https://gitlab.mma.club.uec.ac.jp/mizdra/hello_world_wasm

次節ではモジュールバンドラである Parcel を用いてより簡単に WebAssembly を実行できる開発環境を構築してみます.

4.3 Parcel による開発環境構築

Parcel は Web フロントエンドのための高速でゼロコンフィグレーション (設定ファイルが不要) なモジュールバンドラです. 少し前に以下の記事にて話題になったかと思います.

- webpack 時代の終わりと parcel 時代のはじまり - Qiita
  - https://qiita.com/bitrinjani/items/b08876e0a2618745f54a

2018 年 1 月末にリリースされた Parcel v1.5.0 では WebAssembly と Rust のサポートが入りました[*14]. これにより, 現時点で最も WebAssembly を試しやすい開発環境を Parcel を用いて構築することができるようになりました. 早速試してみましょう.

まずプロジェクトを作成, npm プロジェクトとして初期化して Parcel をインストールします.

```
$ mkdir parcel-wasm-skeleton && cd $_
$ npm init -y
$ npm install --save-dev parcel
```

`/src/lib.rs` を作成します.

```
#[no_mangle]
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

前節で書いたコードと全く同じですね. 次にこの Rust の関数を WebAssembly として呼び出す JavaScript `/src/index.js` を作成します.

```
import { add } from './lib.rs'

console.log(add(1, 2))
```

ES Modules の `import` 構文を用いて `lib.rs` を読み込もうとしています. Parcel はこの構文を見つけると自動で Rust を WebAssembly にコンパイルし, 前節で解説したような JavaScript コードへと変換します. このように Parcel によって WebAssembly の fetch やコンパイル, インスタンス化といったプロセスが隠蔽され, Rust の関数を WebAssembly として呼び出すという本質的な作業に集中できるようになります.

[*14] Parcel v1.5.0 released: Source Maps, WebAssembly, Rust, and more!: https://medium.com/@devongovett/parcel-v1-5-0-released-source-maps-webassembly-rust-and-more-3a6385e43b95

続いてプロジェクトのエントリポイントとなる /src/index.html を作成します.

```
<!DOCTYPE html>
<html lang="ja">

<head>
  <meta charset="UTF-8">
  <title>parcel-wasm-skeleton</title>
  <script src="./index.js"></script>
</head>

</html>
```

プロジェクトをビルドするために npm-scripts にビルドコマンドを追加しましょう. /package.json の scripts フィールドを次のように書き換えます.

```
{
  // ...
  "scripts": {
    "start": "npm run dev",
    "build": "parcel build --no-cache src/index.html",
    "dev": "parcel src/index.html",
    "predev": "npm run build"
  },
  // ...
}
```

準備が整ったので npm run dev で実行してみます. このコマンドでプロジェクトのビルドが行われ, 開発用の HTTP サーバが立ち上がります.

```
$ npm run dev

> parcel-wasm-skeleton@1.0.0 dev /Users/mizdra/src/gomi/parcel-wasm-skeleton
> parcel src/index.html

Server running at http://localhost:1234
Built in 2.95s.
```

ブラウザの開発者ツールのコンソールを開いて 3 が出力されていれば成功です! 開発用ビルドではなくプロダクションビルドを行いたい場合は `npm run build` コマンドを使いましょう.

4.3.1 Rust のサードパーティ製ライブラリの利用

ひとまず Parcel でどのように WebAssembly を動かすかを確認できたので, 次は前節で行っていたような外部ライブラリの呼び出しを Parcel を使って実現してみましょう. `/Cargo.toml` を作成し, 依存する Rust ライブラリを Parcel に伝わるようにします.

```
[package]
name = "parcel_wasm_skeleton"
version = "0.1.0"
authors = ["mizdra <pp.mizdra@gmail.com>"]

[dependencies]
tinymt = { git = "https://github.com/mizdra/rust-tinymt", tag = "0.1.0" }

[lib]
crate-type = ["cdylib"]
```

`/src/lib.rs` に以下を追加します.

```
// ...
extern crate tinymt;

use tinymt::tinymt32;

#[no_mangle]
pub fn rand() -> u32 {
  let param = tinymt32::Param {
    mat1: 0x8F7011EE,
    mat2: 0xFC78FF1F,
    tmat: 0x3793fdff,
  };
  let seed = 1;
  tinymt32::from_seed(param, seed).gen()
}
```

`/src/index.js` を `rand` 関数を呼び出すよう編集します.

```
import { add, rand } from './lib.rs'

const toUint32 = (num) => num >>> 0

console.log(add(1, 2))
console.log(toUint32(rand()))
```

Hot module replacement[*15]により編集内容を保存すればブラウザのページが更新されるはずです! コンソールに 3, 2545341989 が出力されていれば成功です!

4.3.2 Parcel を採用する際のデメリット

さて, このように Parcel を使えば WebAssembly の開発環境が簡単に構築できることが分かりました. しかしながら, Parcel による開発環境ではデメリットがあります. ここでは 2 つ例を挙げます.

1 つ目は前節で行った, WebAssembly からの JavaScript の関数の呼び出しができないことです. 前節では WebAssembly から呼び出したい JavaScript の関数を `WebAssembly.instantiate` に渡すことでこれを実現していました. しかし Parcel では Parcel 自身が自動で WebAssrmbly のコンパイルやインスタンス化を行うコードを生成してしまうため, 開発者が `WebAssembly.instantiate` に JavaScript の関数を渡す余地がありません. こちらの問題は現在 [*16] にて議論されています.

2 つ目は WebAssembly が基本的な数値型しか扱うことができないことです. 次の例を見て下さい.

```
#[no_mangle]
pub fn sum(slice: &[i32]) -> i32 {
  slice.iter().sum()
}
```

符号付き整数のスライスを受け取り, その和を返す Rust の関数です. これを JavaScript 側から呼び出してみます.

```
import { sum } from './lib.rs'

console.log(sum(new Int32Array([1, 2, 3, 4, 5]))) // 0
```

なんと `15` ではなく `0` と出力されてしまいました. これは WebAssembly が引数や戻り値として `i32`, `u32`, `f32`, `i64`, `u64`, `f64` などの基本的な数値型以外をサポートしていないことに起因しています. 現状では, 配列や文字列といった数値型以外を扱いたい場合は JavaScript, WebAssembly 双方からアクセス

[*15] モジュールが更新されたら変更されたモジュールのみをビルドし, 自動でブラウザのページを更新する機能のことです.

[*16] WASM loader does not expose importObject · Issue #647 · parcel-bundler/parcel: https://github.com/parcel-bundler/parcel/issues/647

可能なメモリ `WebAssembly.Memory` を利用する必要があります．JavaScript 側形メモリに配列を配置し, WebAssembly がメモリ上のバイト列をスライスとして読み込む... といったようにすれば先程の関数は動作しますが, 何だか面倒ですね... よくよく考えてみるとメモリに配置したデータはいつ解放するのか, どのデータをメモリ上のどの位置に配置するのか, などなど色々なことを意識しなければならないことが分かります．文字列や配列くらいメモリを意識せずにやり取りする方法は無いのでしょうか...

そこで wasm-bindgen というライブラリが登場します．このライブラリはメモリに関連する処理をラッパーで覆い隠し, JavaScript, WebAssembly 間でメモリを意識せず文字列や配列などをやりとりすることが出来るようにします．しかしながら現時点で Parcel からはこのライブラリを利用することができません[*17]．もし wasm-bindgen を利用するのであれば Parcel 以外のモジュールバンドラを使うか, 前節のようにモジュールバンドラを使わずに開発する必要があります.

4.3.3 本節のまとめ

さて, 本節で学んだことを振り返ります.

- Parcel を使って簡単に WebAssembly の開発環境を構築した
- Parcel を使って Rust のサードパーティ製ライブラリを利用した
- Parcel では解決できない問題があることを学んだ

本節で作成したプロジェクトは以下のリポジトリで公開しています.

- mizdra / parcel-wasm-skeleton · GitLab
  - https://gitlab.mma.club.uec.ac.jp/mizdra/parcel-wasm-skeleton

次節では Webpack という別のモジュールバンドラを用いて wasm-bindgen を利用した開発環境を構築してみます.

[*17] Use Rust WASM bindgen · Issue #775 · parcel-bundler/parcel: https://github.com/parcel-bundler/parcel/issues/775

4.4 Webpack による開発環境構築

Webpack は Web フロントエンドのための拡張性の高い, 高機能なモジュールバンドラです. 現在 Web フロントエンドで使われているモジュールバンドラの中で, 最もユーザの多いのがこの Webpack です.

2018 年 2 月末にリリースされた Webpack v4.0.0 にて, WebAssembly のサポートが入りました[*18]. これに合わせて wasm-bindgen も Webpack v4.x.x に対応し, Webpack を使って高機能な WebAssembly 開発環境を構築することができるようになりました. 試してみましょう!

wasm-bindgen は Nightly 版の Rust に依存しています[*19]. 次のコマンドで Nightly 版をインストールして下さい.

```
$ rustup install nightly
```

プロジェクトを作成・初期化し, プロジェクトのビルドに必要なツール群をインストールします. cargo-watch は Rust ファイルを監視ビルドする際に, wasm-bindgen-cli は JavaScript のラッパーを生成する際に必要になります.

```
$ cargo new --lib webpack-wasm-skeleton && cd $_
$ cargo install cargo-watch
$ cargo install wasm-bindgen-cli

$ npm init -y
$ npm install --save-dev webpack webpack-cli \
  webpack-dev-server html-webpack-plugin
```

`/src/lib.rs` を作成します.

[*18] webpack 4: released today!! – webpack – Medium: https://medium.com/webpack/webpack-4-released-today-6cdb994702d4

[*19] wasm-bindgen/README.md at 0.1.0 · alexcrichton/wasm-bindgen: https://github.com/alexcrichton/wasm-bindgen/blob/0.1.0/README.md#basic-usage

```rust
#![feature(proc_macro)]

extern crate wasm_bindgen;

use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

wasm-bindgen は Rust の実験的な機能である `proc_macro` を利用するので `feature` アトリビュートを付けています．`#` の後に `!` を付けることでアトリビュートをそれを囲むブロック全体に適応することを Rust コンパイラに指示します．ここでは `feature` アトリビュートはトップレベルに置かれているのでトップレベルを囲むブロック，つまり `/src/lib.rs` 全体で `proc_macro` が有効になります．

`add` 関数では `no_mangle` アトリビュートの代わりに `wasm_bindgen` アトリビュートを用いて関数を修飾しています．こうすることで WebAssembly-JavaScript 間で相互にやりとりしやすいように修飾された関数を変換します．また，本来であれば`#[wasm_bindgen::prelude::wasm_bindgen]` と書くところを `use` キーワードを用いることで `#[wasm_bindgen]` と短く書けるようにしています．

次に Webpack の設定ファイル `/webpack.config.js` を作成します．

```javascript
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  resolve: {
    extensions: ['.js', '.wasm'],
  },
  plugins: [new HtmlWebpackPlugin()],
}
```

wasm-bindgen-cli が生成する JavaScript のラッパーは WebAssembly を拡張子を付けずに import しているので `resolve.extensions` に `.wasm` を追加する必要があります．また，html-webpack-plugin を用いて Webpack でバンドルされた JavaScript を`<script>` タグで埋め込んだ HTML を出力するようにしています．この HTML をブラウザで開くことで Webpack でバンドルされた JavaScript が実行できるようになります[*20]．

[*20] 当たり前ですがブラウザで直接 JavaScript ファイルを開いても実行されません．JavaScript を実行するには `<script>` タグで JavaScript を埋め込んだ HTML を開く必要があります．

/Cargo.toml を編集してプロジェクトが wasm-bindgen に依存することを Cargo に伝わるようにします.

```
[package]
name = "webpack-wasm-skeleton"
version = "0.1.0"
authors = ["mizdra <pp.mizdra@gmail.com>"]

[dependencies]
wasm-bindgen = "0.1"

[lib]
crate-type = ["cdylib"]
```

プロジェクトをビルドするために npm-scripts にビルドコマンドを追加しましょう. /package.json の scripts フィールドを次のように書き換えます.

```
{
  // ...
  "scripts": {
    "prebuild:wasm": "cargo +nightly check",
    "build:wasm": "cargo +nightly build --target wasm32-unknown-unknown
--release",
    "postbuild:wasm": "wasm-bindgen
target/wasm32-unknown-unknown/release/webpack_wasm_skeleton.wasm--out-dir src",
    "build:js": "webpack --mode production",
    "build": "run-s build:wasm build:js",
    "dev:wasm": "cargo watch
-i 'src/{webpack_wasm_skeleton_bg.wasm,webpack_wasm_skeleton.js}'
-s 'npm run build:wasm'",
    "dev:js": "webpack-dev-server --mode development",
    "dev": "run-p dev:wasm dev:js"
  },
  // ...
}
```

npm run dev で開発用ビルド, npm run build でプロダクションビルドです. プロジェクトをビルドすると wasm-bindgen-cli により src ディレクトリ配下に WebAssembly ファイル

webpack_wasm_skeleton_bg.wasm とその JavaScript ラッパーの webpack_wasm_skeleton.js が生成されます．WebAssembly を利用する場合は WebAssembly を直接読み込むのではなく，この JavaScript ラッパーを読み込んでラッパー経由で WebAssembly を利用します．

それではラッパーを経由して WebAssembly の関数を呼び出す/src/index.js を作成しましょう．

```
import('./webpack_wasm_skeleton')
  .then(module => {
    const { add } = module
    console.log(add(1, 2))
  })
```

今のところ Webpack では WebAssembly の synchronously import[*21]がサポートされていない[*22]ので，ここでは dynamic import を使っています[*23]．

準備が整ったので実行してみましょう．npm run dev コマンドでプロジェクトのビルドが行われ，開発用の HTTP サーバが立ち上がります．ここで注意してほしいのですが，Cargo によるビルドが終わる前に Webpack によるビルドが実行されるのでビルドの途中でエラーが出ますが，無視して暫く放置して下さい．Cargo によるビルドが完了した時に Webpack がそれを検知して再度ビルドが掛かるので無事ビルドが成功するはずです．

```
$ npm run dev
...
i 「wdm」: Compiled successfully.
```

ブラウザのコンソールに 3 が出力されていれば成功です．ただ，もしかするとブラウザのコンソールに次のエラーが出ている人がいるかもしれません．

[*21] ES Modules による import のこと．

[*22] Synchronously importing wasm modules in the main chunk · Issue #6615 · webpack/webpack: https://github.com/webpack/webpack/issues/6615

[*23] dynamic import は ECMAScript の正式な仕様ではなく，現在 Stage 3 の Proposal です (参考: https://github.com/tc39/proposal-dynamic-import)．

```
Uncaught (in promise) RangeError: WebAssembly.Instance is disallowed on the
main thread, if the buffer size is larger than 4KB.
Use WebAssembly.instantiate.
    at eval (webpack_wasm_skeleton_bg.wasm:4)
    at Object../src/webpack_wasm_skeleton_bg.wasm (0.js:22)
    at __webpack_require__ (main.js:58)
    at eval (webpack_wasm_skeleton.js:25)
    at Object../src/webpack_wasm_skeleton.js (0.js:11)
    at __webpack_require__ (main.js:58)
```

これは WebAssembly を含むプロジェクトをビルドした時に Google Chrome で実行できないコードが出力されるという Webpack のバグに起因しています[*24]．もし上記のエラーが出た場合は Google Chrome の代わりに Mozilla Firefox を使って下さい．

4.4.1 WebAssembly から JavaScript の関数を呼び出す

Webpack でどのように WebAssembly を動かすかを確認できたので，次は WebAssembly から JavaScript の関数の呼び出しに挑戦してみましょう．

`/src/lib.rs` に以下のコードを追加します．

```rust
// ...
#[wasm_bindgen(module = "./index")]
extern {
    fn date_now() -> f64;
}

#[wasm_bindgen]
pub fn get_timestamp() -> f64 {
    date_now()
}
```

`/src/index.js` は次のように編集します．

[*24] Unable to import WebAssembly modules bigger than 4KB · Issue #6475 · webpack/webpack: https://github.com/webpack/webpack/issues/6475

```
export const date_now = Date.now

import('./webpack_wasm_skeleton')
  .then(module => {
    const { add, get_timestamp } = module
    // ...
    console.log(get_timestamp())
  })
```

ここでのポイントは extern ブロックを#[wasm_bindgen(module = "./index")] で修飾していることです．こうすると wasm-bindgen は /src/index.js で export されているアイテムを extern ブロックで定義されるアイテムへとバインディングします．やっていることは 2 節のものと同じですが，こちらの手法の方がより宣言的でモジュール指向です[*25]．JavaScript 側では ES Modules の export キーワードを用いて Rust 側からアイテムが参照できるようにしています．また，wasm-bindgen が JavaScript の関数を呼び出している箇所を自動で unsafe で囲ってくれるので unsafe ブロックを使用していないことにも注意して下さい．

Hot module replacement により編集内容を保存すればブラウザのページが更新されるはずです! コンソールにタイムスタンプが出力されましたか? リロードする度に出力される値が変わっていれば成功です!

4.4.2 Rust のサードパーティ製ライブラリの利用

続けて外部ライブラリの呼び出しを Webpack を使って実現してみましょう． /Cargo.toml の dependencies に tinymt クレートを追加しましょう．

```
// ...
[dependencies]
wasm-bindgen = "0.1"
tinymt = { git = "https://github.com/mizdra/rust-tinymt", tag = "0.1.0" }
// ...
```

/src/lib.rs に次のコードを追加します．

[*25] バインディングされるアイテムを静的に解析することが容易という理由で「宣言的」と表現しています．

```rust
// ...
extern crate tinymt;

use tinymt::tinymt32;

#[wasm_bindgen]
pub fn rand() -> u32 {
    let param = tinymt32::Param {
        mat1: 0x8F7011EE,
        mat2: 0xFC78FF1F,
        tmat: 0x3793fdff,
    };
    let seed = 1;
    tinymt32::from_seed(param, seed).gen()
}
```

/src/index.js を rand 関数を呼び出すよう編集します.

```js
// ...
const toUint32 = (num) => num >>> 0

import('./webpack_wasm_skeleton')
  .then(module => {
    const { add, get_timestamp, rand } = module
    // ...
    console.log(toUint32(rand()))
  })
```

特に前節でやったことと変わりはありませんね. 編集内容を保存してブラウザのコンソールを見てみましょう. 出力に 2545341989 が追加されていれば成功です!

4.4.3 コレクション, 文字列の受け渡し

さて, ここから wasm-bindgen の本領が発揮されます. まずは wasm-bindgen を使って前節で出てきた sum 関数を実装してみましょう. /src/lib.rs に以下を追加します.

```
// ...
#[wasm_bindgen]
pub fn sum(slice: &[i32]) -> i32 {
    slice.iter().sum()
}
```

そして /src/index.js から sum 関数を呼び出します.

```
// ...
import('./webpack_wasm_skeleton')
  .then(module => {
    const { add, get_timestamp, rand, sum } = module
    // ...
    console.log(sum(new Int32Array([1, 2, 3, 4, 5])))
  })
```

注意点としては Rust 側の関数の仮引数では配列型ではなくスライス型を使用し, JavaScript 側の関数呼び出しの実引数では通常の配列ではなく対応する TypedArray を使用することです.

Rust の関数からコレクションを返したい場合は std::vec::Vec を使うと, JavaScript 側で対応する TypedArray で受け取れます. 以下はコレクションの各要素を 2 倍した新しいコレクションを返す twice 関数の実装例です.

```
// ...
#[wasm_bindgen]
pub fn twice(slice: &[i32]) -> Vec<i32> {
    slice.iter().map(|x| x * 2).collect()
}
```

JavaScript から呼び出す場合はこうです.

```
// ...
import('./webpack_wasm_skeleton')
  .then(module => {
    const { add, get_timestamp, rand, sum, twice } = module
    // ...
    // console.log(sum(new Int32Array([1, 2, 3, 4, 5])))
    console.log(sum(twice(new Int32Array([1, 2, 3, 4, 5]))))
  })
```

文字列の受け渡しはどうでしょうか．文字列をブラウザのコンソールに出力する `hello` 関数を作成してみます．

```
// ...
#[wasm_bindgen(module = "./index")]
extern {
    // ...
    fn console_log(s: &str);
}

#[wasm_bindgen]
pub fn hello() {
    console_log("Hello, World!");
}
```

JavaScript 側ではバインディングするアイテムを export して `hello` 関数を呼び出すコードを追加するだけです．

```
// ...
export const console_log = console.log

import('./webpack_wasm_skeleton')
  .then(module => {
    const { add, get_timestamp, rand, sum, twice, hello } = module
    // ...
    hello()
  })
```

ブラウザのコンソールを開いて出力を確認してみましょう．正しくコードが書けていれば `30` と `Hello, World!` が出力に追加されているはずです．`"Hello, World!"` が出力できたのでこれで本当の WebAssembly 入門が終わったと言えそうですね! ハハハ...

4.4.4 本節のまとめ

本節では次のことを学びました．

- Webpack と wasm-bindgen を使って高機能な WebAssembly の開発環境を構築した
- Webpack と wasm-bindgen を使って Rust のサードパーティ製ライブラリを利用した
- Webpack と wasm-bindgen を使ってコレクションや文字列をやり取りする方法を学んだ
- `"Hello, World!"` を出力して本当の WebAssembly 入門を終えた

本節で作成したプロジェクトは以下のリポジトリで公開しています.

- mizdra / webpack-wasm-skeleton · GitLab
  - https://gitlab.mma.club.uec.ac.jp/mizdra/webpack-wasm-skeleton

4.5 おわりに

今回は Web の最先端の技術の 1 つである WebAssembly の開発環境の構築方法を紹介しました．今まで WebAssembly に興味を持っていた人に WebAssembly を触るきっかけになっていれば幸いです．

2017 年になって主要なブラウザ全てで WebAssembly がサポートされ[*26], 2018 年に入って主要なモジュールバンドラで WebAssembly がサポートされるようになりました．仕様策定もブラウザベンダが結束して勢い良く進められています．また本文章では触れませんでしたが, ブラウザだけでなく, Node.js や IoT デバイスなど様々な環境で WebAssembly を動作させようとする試みが現在行われています．もしかしたら WebAssembly はどこでも動くユニバーサルなバイナリとして発展・普及していくかもしれません．今後も WebAssembly に注目していきましょう!

4.6 参考文献

- インストール ・ プログラミング言語 Rust: https://www.rust-lang.org/ja-JP/install.html
- はじめる ・ プログラミング言語 Rust: https://rust-lang-ja.github.io/the-rust-programming-language-ja/1.6/book/getting-started.html
- Rust で WebAssembly を出力する: http://nmi.jp/2018-03-19-WebAssembly-with-Rust
- Setup - Wasm target - Hello, Rust!: https://www.hellorust.com/setup/wasm-target

[*26] WebAssembly 対応、主要ブラウザ Chrome/Firefox/Edge/Safari で整う。Web アプリの開発言語として JavaScript 以外の選択肢は広がるか？ー Publickey: http://www.publickey1.jp/blog/17/webassembly_browsers.html

第 5 章

新入生へのアドバイスというタイトルの偉そうな文章

kyontan

概要

kyontan です。この記事はこの部誌におけるいわば箸休めのような記事です。[*1]2015 年度入学の MMA の一部員である私の、現在から遡ること 3 年程度のタイムラインをつらつらと並べていきます。新入生の皆様の立ち振舞の参考になれば幸いです。

5.1　入学まで

全日制の高校 (工業課程) を卒業し、暇だったので運転免許を 3 つ取得しました。暇ですね。

5.1.1　アルバイトを始める

運転免許も取得して更に暇になったので、せっかくの大学生だしアルバイトもするかと思いたち、プログラミングができるアルバイトを始めました。知り合いに紹介して頂いた会社であったが、受託開発を主とする会社で、労働環境はホワイトで、特に悪いところもなかったのでその後 2 年ぐらい働くことになりました。面接に行った 1 時間後に、何故か炎上案件に放り込まれて死ぬ思いで Java を読み書きしたことを思い出しましたが、特にその後炎上していたことはなく、平和な会社であった。

5.2　学部 1 年 (2015 年度)

5.2.1　入学

電気通信大学の新歓期間というのは入学式の前日からとなっていますが、私は行かなった。実際どうなのでしょ……。行きたいサークルが決まっていないのが大半なので良い活動なのかもしれない。

[*1] 箸休めなのに最後に書いたので最後にあり、箸休めになっていない

入学式の書類で VB セミナーという団体 (?) のチラシを見て「今時 Visual Basic?」みたいなことを思ったのが記憶に残っています。この VB セミナーが後々私を苦しめる事になりました。

入学式の隣の席に座っていた人と知り合い、彼とはその後も交流があります。そんな彼は色々な文理問わず様々な分野に精通し、最近ではなぜか Web 系 IT ベンチャーで学生なのに社内の唯一のエンジニアみたいなことをしているらしい。人生どうなるかわからないですね。

5.2.2　MMA 入部

元々 MMA という団体を知っていて、入部する予定でいたので入学式の後そのまま入部届を書きに部室にいった気がします。先輩部員となる id: why は数年前からリアルで知り合いだったこともあり、部室でいきなり対面して驚かせました。

5.2.3　VB セミナー

部長の検閲により削除 [*2]

5.2.4　KnobCon

日頃参加していたコミケにサークル参加してみたいと思いたち、 KnobCon [*3] という PC の周辺機器を作り頒布しました。良い経験。

5.2.5　MMA 合宿

MMA の合宿では夏に伊豆に行って花火を打ち上げます。打ち上げました。

図 5.1　花火

[*2] 部長が悪いのではない、私が邪悪なのだ。 https://wiki.mma.club.uec.ac.jp/kyontan/WhatIsVbSeminer (部内専用)
[*3] https://tofu-ya.comike.moe/

5.2.6 アルバイト

2015 年 12 月から、VB セミナーであった講演で、とある Web 系サービス企業の CTO をしている電通大の OB に会い、トントン拍子でその企業でアルバイトすることになった。特に前職をやめるわけではなく、掛け持ち状態の始まりである。以降常に掛け持ち状態になります。時給も良く (1500 円)、最近っぽい Web 系ベンチャーで働けるのは良かったが、上司と反りがあわず苦しむことになりました。

成果がでないのは自分ももちろん悪いかもしれないですが、正しくマネージされないのが悪いということに気がつくまでの半年を苦しみ、そのまま夏のインターン参加を気に退職しました。[*4]

つらいときはすぐに辞めましょう。この文章における唯一のアドバイスです。

5.3 学部 2 年 (2016 年度)

5.3.1 ICT トラブルシューティングコンテスト

おそらく大学に在学した 4 年間で一番の活動はなんですか、と言われたらこの活動を挙げるでしょう。この大会はネットワークやサーバ管理といった、インフラ技術を主に取扱い、その技術を競うコンテストであり、その運営をしていた先輩に誘われて私も運営にジョインしました。私はこの大会の運営を通して、今の技術的な知識の基礎となるインフラに関連した技術を学んだり、チームマネジメントへチャレンジしたりしました、もちろん全てが上手く行ったわけではないですが、学んだことは多く、知り合いも多く増えました。良かったですね。結局、2016/3 に実施された第 5 回から、2017/8 に実施された第 8 回大会まで運営として関わっていました。ブログに、具体的にどんなことをしたが書いているので興味を持った方はご参照ください。[*5]

5.3.2 MMA に関連した活動

1 年次に合宿担当の手伝いという立場であったため、そのまま昇進 (?) して合宿担当になった。合宿担当というのはその名の通り合宿の企画が主な仕事であるが、MMA では前述の通り合宿で打ち上げ花火を行うため、これに類する調整の方が忙しくなりました。書類の提出先も数か所あり、提出時の経路を変更するなどして効率よく準備が行えるようになった。また、昨年度まではツアー会社に依頼して宿の予約を行っていたが、百害あって一利なしという状態であったため、MMA から直接宿へ交渉をするようにする、といったことも行った。これにより、合宿の参加費の削減や自由度の向上をすることができたので良かったのではないかと思っています。

そういえば合宿の 3 日目に成人し、酒を飲んだ。この時に初めて僕が酒を飲めることがわかり、ここから人生が堕落していくこととなった。また、年度終わりの総会における投票により、副部長の座につく

[*4] https://blog.monora.me/2016/10/retired-a-part-time-job/

[*5] https://blog.monora.me/2016/10/had-been-a-steering-committee-of-ictsc6/
https://blog.monora.me/2017/03/had-been-a-steering-committee-of-ictsc7/
https://blog.monora.me/2017/10/had-been-a-steering-committee-of-ictsc8/

ことになります。

5.3.3　シリコンバレーに行く

図 5.2　忘れな草

リクルートが U-30 程度の人々を対象に、シリコンバレーに連れて行ってエモい体験をさせてくれる Silicon Valley Workshop というものがあります。[*6] 秋に再会した高校の先輩が、これに参加しており、めっちゃ良かったよと教えられたこともありノリで応募。[*7]コーディングテストや面接が数回あり、流石に落ちるでしょと思っていたが奇跡的に通過する私。あとで聞いたところやはり変人枠だったらしい。[*8]そうして結果的にリクルートの金で 1 週間ほどシリコンバレーに滞在し、エモい体験をたくさんすることになります。Google のオフィス見学したり Airbnb 見たり、スタンフォード大学行ったり コンピュータ歴史博物館行ったり。エモいですね。自由日があったので BSD の B であるところのカリフォルニア大学バークレー校に行くなどしました。

図 5.3　Cisco 橋

往々にしてこういうイベントは怪しく、何らかの見返りを求められたりしがちですが、実体験上、このイベントに関しては大丈夫だと言えます。感化される人間はもちろんいて、そのままアメリカに留学する人間やベンチャーを起業する人間はいます。そういうところも含めてエモい。自分については、そういう意味合いでは感化されていない (と自分自身は思っています) が、やはり IT に関わる人間としてその産業の中心となる地を見に行くことができたのは最高の体験だったと思います。

大学に入ってからの最大の思考の転換点はここにあったのかなと今は思います。

ちなみにカリフォルニア州では飲酒は 21 歳からであり、当時 20 歳だった自分は酒を飲むことができなかった。悲しい。

[*6] http://techlabpaak.com/siliconvalleyworkshop2018
[*7] ちなみに実施は 3 月と 8 月の年 2 回。
[*8] 変人枠ってなんだよ……

5.3.4 アルバイトやインターン

夏休みにはインターンとして 21 世紀のインターネットを代表する会社[*9] に行ったが、これは良い体験であった。[*10] そのままその部署でアルバイトを初めて今に至ります。

5.4 学部 3 年 (2017 年度)

5.4.1 アルバイトやインターン

3 年になる直前に某インターネットの会社にアルバイトに来ないかという誘いを受けた。もともと使っていたサービスで興味があったこともあり、入学前から続けていた企業から転職、Rails エンジニアになりました。今も続いているが、環境としてはこれに勝るところはそうないでしょう。フルリモートが許可されるアルバイトは偉大。

3 年の夏休みには某社にインターンに行ったが、これはあまりよくない体験であった。このような企業に就職しなくてよかったとは思うが、貴重な時間を不意にしてしまったことは悔やむところだ。[*11]

また、4 年になる直前には、まだここにない何か、をモットーとする企業にインターンで参加し、良い体験をしました。[*12]

5.4.2 研究室配属

3 年次の目玉行事はなんといっても研究室配属でしょう。大学は研究をする場所であるというのはおそらく同意されるかと思いますが、どういった研究を、誰のもとで行うのか、といったことが決まるのがこの研究室配属という行事 (?) です。学科 (類) によっても違いがありますが、これに関連した活動は主に後期の調布祭の前に始まります。希望する研究室をいくつか選び、教授などと面談した上でマッチングを確認し、所定の規則にしたがって配属先が決定されます。

私はどういった研究をするか長期に渡って悩んだ結果、ネットワーク系の研究を行う研究室を志望することにしました。これが吉と出るか凶と出るかは執筆時点では分かっていない。[*13]アドバイスできることがあるとしたら、教授等のマッチングだけでなく、そこにいる学生の地頭の良さなどを見ておくと良いのではないかと思います。

[*9] 学生気分
[*10] https://blog.monora.me/2016/10/did-an-internship-at-cyberagent/
[*11] https://blog.monora.me/2017/10/did-an-internship-at-wantedly/
[*12] https://blog.monora.me/2018/03/did-an-internship-at-recruit-technologies/
[*13] 読者の課題にはしません

5.4.3　MMA に関連した活動

副部長を担っていたはずですが、あまりこれといったことはしなかった気がします。調布祭で配布する部誌の担当となり、記事を書く部員をかき集めたり、部室の極めて調子の悪いプリンタ[*14]と格闘したり、先輩の残した表紙ジェネレータのリファクタリングをしたりしていました。また、年度末の総会により役員 (副部長) を引退し、平の部員に戻りました。

次の部誌 (=この部誌) を印刷する次の部誌担当に既存の極めて調子の悪いプリンタを引き継ぐのは流石に申し訳ないと思い、新規のプリンタの導入を部会で提案し可決、購入します。この部誌がお手元で読まれていれば、その部誌の初版は新しいプリンタで刷られているはずでしょう。そして部誌を執筆している今、2018/4/3 に至ります。今日は初めてのゼミがあり、これからどういった生活になるか楽しみですね。

5.4.4　JobHunting

JobHunting、日本語で言えば就活ですが、MMA 内ではこちらで呼ばれることが多いです。MMA の部員の就活は大きく二分されており、就活で苦しまない人間と、好きで就活で苦しむ人間に分けられます。観測範囲ではあまり就活で苦しんでいる印象はない。Linux のデスクトップで開けない適性検査に苦しむ人はたまに見る。

僕は好きで就活をやっていた質であるが、周りには疲弊しているように見受けられたらしい。承認欲求を満たすためにすきでやっただけなのですが……あまりここに書く内容でもないと思うので、もし興味があれば、Wiki の `kyontan/JobHunting` [*15] を参照してください。

5.5　最後に

学内外問わず、面白いことを探して彷徨った結果、今に至ります。「人生の if ルート」というのは私が好きな言葉ですが、ハッピーエンドはどこにあるのでしょうか。[*16]これからも彷徨い続けることでしょう。

最後に、新入生の皆様、ありきたりな言葉ではありますがご入学おめでとうございます。皆様がこの電気通信大学で充実した生活を過ごされることを心より願っております。

[*14] :put_litter_in_its_place:

[*15] https://wiki.mma.club.uec.ac.jp/kyontan/JobHunting (部内専用)

[*16] MMA 結納部

百萬石 2018 -春- ©電気通信大学 MMA

2018 年 4 月 3 日　初版第一刷発行　【本書の無断転載を禁ず】

著　者　mizdra, hogas, kyo1, akky, kyontan
編集者　swkfn
発行者　電気通信大学 MMA
発行所　電気通信大学 MMA 部室
〒182−8585 東京都調布市調布ヶ丘 1−5−1 サークル会館 208 号室
http://www.mma.club.uec.ac.jp/
印刷所　電気通信大学 MMA 部室
製本所　電気通信大学 MMA 部室

電気通信大学