PMOB

のよむやつ #2

はじめに

春です．こはるです．

第39回鳩山祭で『PMOB の読むやつ #1』を頒布したところ，予想以上に多くの方の手に渡りました．ありがとうございます．初日に3日分の冊子が捌けてしまい，期間中に増刷を行うこともありました．タイミングが悪く手に入らなかった方には非常に申し訳ありませんでした．

そんな PMOB の読むやつですが，好評につき新入生向けに第二弾を制作することになりました．これからは春と秋で年に二回のペースで制作できたらいいなぁという感じです．

また，活動の方も1月末に行った LT 会である PMOB LT 会 hanasuyatsu#2 では千住キャンパス，千葉ニュータウンキャンパスの方も登壇してくれました．登壇者はその後 Slack に参加してもらい，互いにイベントの告知などをスムーズに行うことができるようになりました．

3月末の千住キャンパスで行われた勉強会にも鳩山から数名参加してお互いにイベントを盛り上げていくことができたと思います．

2016年度は学内外ともに活動を盛り上げていければと思います．これからもよろしくお願いします．

2016年4月12日 cohalz

目次

第 1 章 春のエディタ特集 1
1.1 エディタってなに 1
1.2 どんなエディタがあるの 1
1.3 atom 1
1.4 vim 2
1.5 emacs 3
1.6 nano 6
1.7 おわりに 6

第 2 章 いまさら人に聞けない dotfiles 7
2.1 dotfiles なに 7
2.2 リポジトリで管理する 7
2.3 git-submodule で快適拡張 9
2.4 デプロイスクリプトの作成 11
2.5 インストールスクリプトの作成 12

第 3 章 厨二病をもう一度こじらせて PSP 向けプログラムを作る 14
3.1 まず始めに 14
3.2 準備するもの 15
3.3 Hello World を表示させる 16
3.4 感想とか 22

第 4 章 猫でもわかるグラフィックスパイプライン 23
4.1 描画のために何が必要か 23
4.2 グラフィックスパイプライン 24
4.3 補足とか 27

第 5 章 珠玉の Ruby プログラミング 30

5.1 方針 . . . 30
5.2 動機 . . . 30
5.3 Ruby とは . . . 31
5.4 題材の詳細 . . . 32
5.5 解法 . . . 33
5.6 プログラムコード . . . 34
5.7 参考文献 . . . 35
5.8 良書と思われる書籍 . . . 35
5.9 あとがき . . . 36

第 6 章 Linux のすゝめ 37
6.1 始めに . . . 37
6.2 Linux カーネルとはなんぞや . . . 38
6.3 で，何がいいの？ . . . 40
6.4 インストール？ . . . 42
6.5 あなたの Linux に . . . 44
6.6 For Linux Users . . . 48
6.7 おわりに . . . 50

第 7 章 大著謎ジャンル本のススメ 51
7.1 大著謎ジャンル本とは . . . 51
7.2 大著謎ジャンル本の面白さとは . . . 52
7.3 本の探し方 . . . 53
7.4 最後に . . . 53

第 1 章

春のエディタ特集

いっちー (@sanex_now)

大学の講義では学ばないエディタの種類と使い方。自分にぴったりのエディタを見つけよう！

1.1　エディタってなに

この記事ではテキストエディタのことをエディタと呼んでいます。エディタは主にテキストファイルを編集する目的のアプリケーションをさしています。
今回この記事ではプログラムを作成する際に利用されるエディタを紹介していきます。

1.2　どんなエディタがあるの

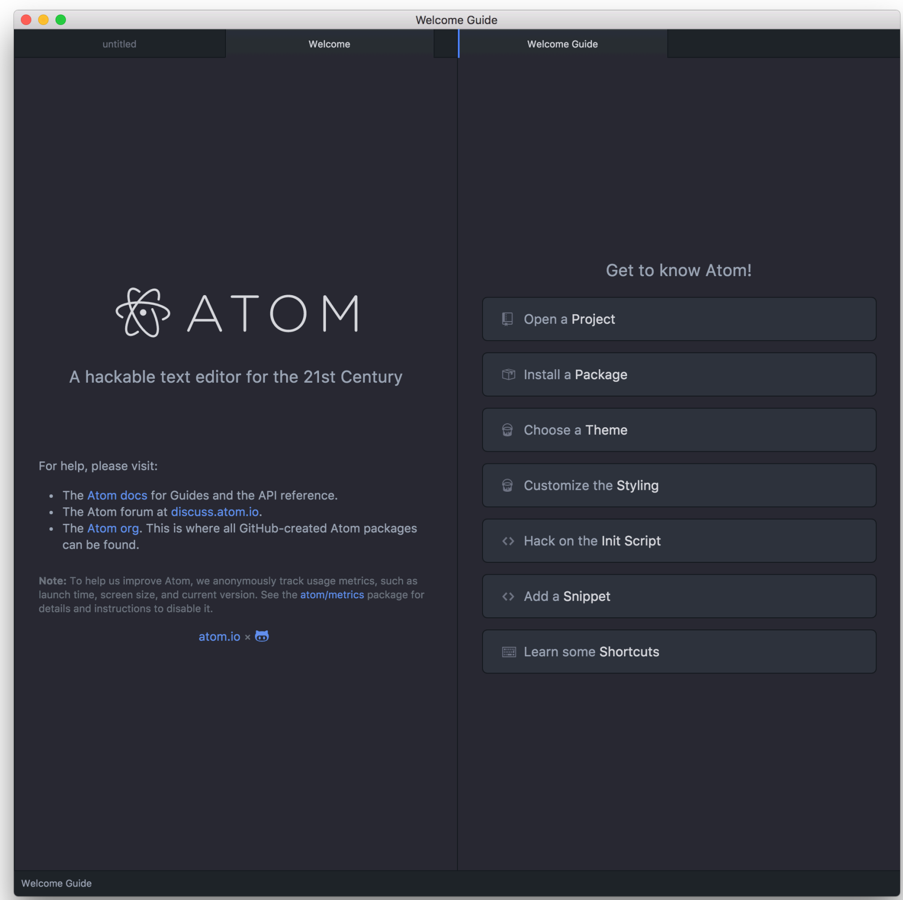
PMOB のメンバーがどのエディタを使っているのか聞いてみました。

- emacs
- vim
- atom

1.3　atom

atom は github 謹製のエディタです。独自のパッケージマネージャを搭載しており、様々なパッケージを追加することができます。

atom.io



MacOSX,Windows，多くの Linux ディストリビューションで動作する GUI エディタなので、キャラクタベースのエディタに慣れていない方などにはオススメです。

1.4　vim

vim とは UNIX で人気のある vi エディタの派生です。vi の流れを汲んでおりかなり歴史のあるエディタです。PMOB 内でもかなり人気があるみたいです。

www.vim.org

1.4.1　vim の操作

vim は一般的なエディタと違い、操作が独特です。
モードという概念があり、ノーマルモード、インサートモード、ビジュアルモードが存在します。

表 1.1 vim のモード切り替え

コマンド	説明
i	挿入モード
v	ビジュアルモード
<ESC>	ノーマルモード
C-[	ノーマルモード

vim でのファイル編集はこのモードを切り替えつつ行います。

1.2 に基本的な操作方法を示します。

1.5 emacs

vim と同じく人気なのが emacs です。emacs もかなりの歴史を持っており、vi 派 PMOB 内でも vim と同程度の人気があるようです。

www.gnu.org

私が主に使っているエディタでもあります。特徴は何と言っても Emacs Lisp による拡張です。バージョン 24.5 から package.el がデフォルトで入っており、パッケージの追加を簡単に行うことができます。

1.5.1 emacs の操作

emacs は vim とは大きく操作方法が異なります。モード切り替えによって単一キーに多数の機能を割りあてる vim とは異なり、emacs は編集中に頻繁にモードを切り替えることはありません。

emacs は Ctrl キー、Meta キーなどのキーとの同時押しにより単一モードで多数の機能が割り当ててあります。

1.5.2 spacemacs

emacs ディストリビューションの一つに spacemacs というものがあります。

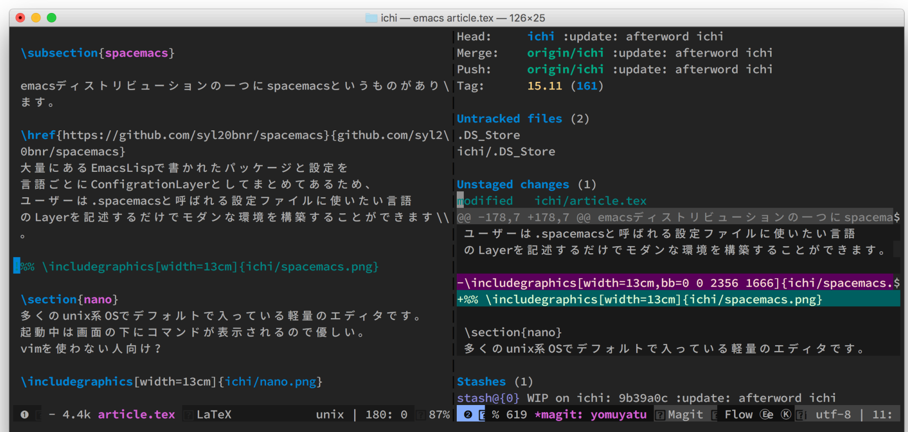
github.com/syl20bnr/spacemacs 大量にある EmacsLisp で書かれたパッケージと設定を言語ごとに ConfigrationLayer としてまとめてあるため、ユーザーは.spacemacs と呼ばれる設定ファイルに使いたい言語の Layer を記述するだけでモダンな環境を構築することができます。

表 1.2　ノーマルモードの基本操作

カテゴリ	コマンド	説明
移動	h	一文字左へ移動
	j	一行下へ移動
	k	一行上へ移動
	l	一文字左へ移動
	l	一文字左へ移動
	^	行頭へ移動
	$	行末へ移動
コピー＆ペースト	[数値]dd	カーソル位置の行から [数値] 行削除
	[数値]yy	カーソル位置の行から [数値] 行をコピー
	P	カーソル位置後方にペースト
	p	カーソル位置前方にペースト
アンドゥ＆リドゥ	u	アンドゥ
	U	行への変更全てを取り消す
	C-r	リドゥ
検索	/[文字列]	前方検索（カーソル位置より後を検索）
	?[文字列]	後方検索（カーソル位置より前を検索）
	#	カーソル位置の単語を前方検索
	*	カーソル位置の単語を後方検索
選択	v	選択開始
	V	行選択
ファイル操作	:e	ファイルオープン＆新規作成
	:w	保存（:w!で強制保存）
	:q	終了（:q!で強制終了）
	:wq	保存して終了
	ZZ	保存して終了
	ZQ	保存せず終了

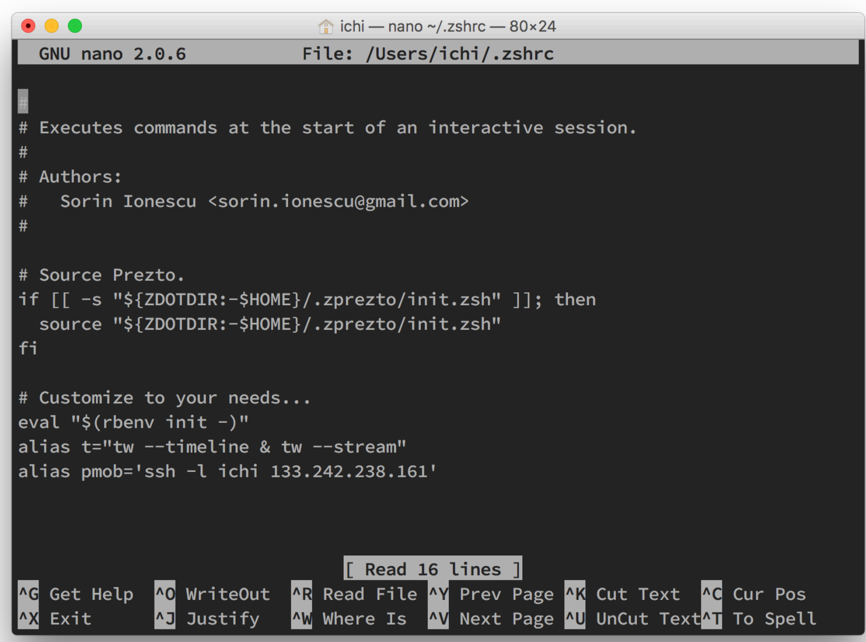
表 1.3 emacs の基本操作

カテゴリ	コマンド	説明
移動	C-f	一文字右へ移動
	C-b	一文字左へ移動
	C-n	一行下へ移動
	C-p	一行上へ移動
	C-a	行頭へ移動
	C-e	行末へ移動
編集	C-k	現在行のカーソル位置以降をカット
	C-w	選択範囲をカット（無選択時はカーソル前単語カット）
	M-w	選択範囲をコピー（無選択時はカーソル後単語コピー）
	C-y	ペースト
	C-x u（C-_）	アンドゥ
	Backspace	カーソル左を一文字削除
	C-d	カーソル位置を一文字削除
ファイル操作	C-x C-f	ファイルオープン
	C-x C-s	保存
	C-x C-c	終了
	C-x i	ファイルを指定して挿入
困った時	C-g	キャンセル コマンドの実行中でもキャンセルしてくれるので困ったらこれ叩けばどうにかなる
	C-h t	チュートリアルが読めるゾ



1.6　nano

多くの unix 系 OS でデフォルトで入っている軽量のエディタです。起動中は画面の下にコマンドが表示されるので優しい。vim を使わない人向け?



1.7　おわりに

今回は PMOB メンバーたちが使っているエディタを中心に紹介しました。特に vim と emacs は操作が独特で慣れるまでに時間を要しますが、指が覚えてくると作業効率が格段に上がります。はじめはとっつきにくいですが是非試してみてください。

第 2 章

いまさら人に聞けない dotfiles

matsub(@matsubrk)

今回は春号ということで新入生向けの記事をこしらえ候。dotfiles を git で管理してワンライナーでデプロイしちゃうぞ。UNIX 系 OS 前提です。

2.1 dotfiles なに

dotfiles とは、UNIX 系 OS でよく使われる設定ファイルの総称です。シェルのリソースファイル（ `.bashrc` や `.zshrc` ）だとか、エディタの設定ファイル（ `.emacs.d/` や `.vim/` ）なんかがあると思います。これら設定ファイルは配置すると便利だけど `ls` したときに毎回出てくるとうっとうしいので、隠しファイルとなるように先頭が `.` で命名されています。このことから、これらの環境設定ファイルを dotfiles と呼ぶことがあります。

2.2 リポジトリで管理する

サーバへアクセス、別の PC、職場のユーザなど、普段と違うユーザでの操作は多いものと思います。そのとき面倒になるのがユーザーの設定である dotfiles の配置です。以前は普段使っているユーザーからコピーしてきたりとかしてましたが、これをバージョン管理しちゃうと楽ですよ。というのが本記事の内容。

2.2.1 VCS を使って管理する

ここでの問題は複数ユーザーで共通の dotfiles を扱うことですから、以下のような要件を設定できます。

- dotfiles を一括で管理できる
- 各ユーザーので行った変更を同期できる

この課題を解決するために 1. 一箇所のサーバで設定ファイルを管理して、2. ファイルの変更を管理できる、ということで VCS を使いたいと思います。環境が違う場合でも、パッチを使えばアーキテクチャによる小さな違いにも対応できるので便利。

サーバはナウい VCS サーバホスティングサービスを使いましょう。GitHub や Bitbucket、GitLab、assembla などが代表的だと思います。VCS は Subversion、Mercurial、Git などがあります。今回は VCS には上記のホスティングサービスで共通して使える **Git**、ホスティングサービスは弊サークルもお世話になっている **GitHub** を前提に話を進めていこうと思います。

2.2.2 リポジトリの準備

git の導入、GitHub アカウントのサインアップ他設定は済ませているものとします。

まずリポジトリの作成が必要になります。リポジトリ名は "dotfiles" あるいは ".dotfiles" とする習慣になっています。ローカルリポジトリを格納するディレクトリを隠しファイルとしたいかどうかで判断してください。今回は "dotfiles"とします。また、以下の"<name>"とあるところはあなたの github のユーザ名となります。

リモートリポジトリが作成できたらクローンします。dotfiles の実体の場所はどこでもいいです。

```
git clone https://github.com/<name>/dotfiles.git
```

できたら dotfiles をローカルリポジトリに移動しましょう。ただし、`.gitconfig` などの git の操作に必要な dotfiles は移動させると git が即死しますので、ステージング作業の前にホームディレクトリへシンボリックリンクを張ります。

注意として、dotfiles として管理するファイルは基本的に設定ファイルですので**ヒストリファイルや `.ssh` などは絶対に含めないように**してください。秘密鍵は聞いたことありませんが、API キーを dotfiles に突っ込んでしまってお金が吹き飛んだエピソードは時々聞きます。

リスト 2.1 dotfiles の初期設定

```
1  #!/bin/bash
2  DOTFILES=(
3  ".zshenv"
4  ".zsh"
5  ".gitconfig"
6  # その他もろもろ
7  )
8
9  for file in ${DOTFILES[@]}; do
10     mv $HOME/$file dotfiles/
11     ln -s dotfiles/$file $HOME/$file
12 done
```

最初は情けないけど手動で動かさないといけない。作業が済んだら dotfiles をステージングしてコミットしてプッシュしちゃってください。これでとりあえずは完了です。

2.3 git-submodule で快適拡張

dotfiles 内、例えば `.zsh/` や `.vim/` 内でプラグインを利用する際、プラグインのリポジトリをクローンして利用いることは多いと思います。dotfiles をリポジトリに入れるのに際して、リポジトリの中に別のリポジトリが入ってるとかキモすぎるので、これらを dotfiles のサブモジュールにします。サブモジュールにすることで、別のリポジトリを親リポジトリの一部として取り込むことができます。これをすると、親リポジトリ上で扱うサブモジュールのリビジョンを指定できるので便利です。何より実体ファイルを自分のリポジトリ自体が持つわけではないので、**安全に依存プロジェクトのライセンスを守る**ことができます。

2.3.1 やってみるホイ

例えば私は zsh に "auto-fu.zsh" というプラグインを使用しています。これをサブモジュール化する行程を紹介します。

まず、auto-fu.zsh のインストール先は `.zsh/plugins/` 以下です。auto-fu.zsh の Git リポジトリの URL は `https://github.com/hchbaw/auto-fu.zsh.git` ですので、これをサブモジュールとして登録します。dotfiles のディレクトリへ移動し、

```
git submodule add \
https://github.com/hchbaw/auto-fu.zsh.git.zsh/plugins/auto-fu.zsh \
.zsh/plugins/auto-fu.zsh
```

これでサブモジュール登録完了です。普通はこれで動きますが、私の環境ではなぜか master ブランチでは auto-fu.zsh が正常動作しませんでした。こういった理由でリビジョンを指定したいということは常々あると思うので、この対策も書いておきます。サブモジュールのリビジョンは親リポジトリで操作できる範囲では、ブランチ単位での指定ができます。今回、auto-fu.zsh を pu ブランチに切り替えれば動作することがわかりましたので、私の環境では pu ブランチを利用するように設定したいと思います。

サブモジュールの設定は `.gitmodules` で行われており、これはファイル指定の `git-config` で制御できます。

```
# git config -f .gitmodules submodule.<path/to/submodule>.branch \
# <branch>
git config -f .gitmodules submodule..zsh/plugins/auto-fu.zsh.branch pu
```

このコマンドですが、ときどきエスケープかなんかの問題でうまくいかなかった記憶があります。うまくいかなかったらブチ切れながら `.gitmodules` を直接いじりましょう。中身は以下のよ

うになっているので、最後の一行を加えればオーケーです。

```
[submodule ".zsh/plugins/auto-fu.zsh"]
    path = .zsh/plugins/auto-fu.zsh
    url = https://github.com/hchbaw/auto-fu.zsh.git
    branch = pu
```

ブランチの設定が完了したらサブモジュールの設定を適用しましょう。

```
git submodule init
git submodule update
```

コミットへチェックアウトするにはサブモジュールのリポジトリで適宜チェックアウトする必要がありますが、これは親リポジトリの設定として記録することはできません。

2.3.2 .gitignore の設定

ちょっと本題とはズレますが、`.gitignore` の設定の話です。vim の Vundle や neobundle、emacs の el-get や Cask など、アプリケーションごとのパッケージマネージャでプラグインのリポジトリを管理することがあると思います。これらを git-submodule で管理するのは複数のプログラムが一つのリポジトリに干渉することになってﾁｮﾄﾖｸﾅｲので、そういったプラグインのインストール先ディレクトリは `.gitignore` に登録して無視したいですよね。例えば neobundle を使う際、`.vim/bundle` にプラグインをインストールしているなら、`.vim/.gitignore` に以下のような記述をします。

```
bundle
```

こうすると neobundle が管理するプラグインをリポジトリの管理外におけるのですが、neobundle 自体もまとめてこのディレクトリ入れている場合、dotfiles インストールの際に neobundle をインストールする手間が加わってしまいます。`.vim/bundle/neobundle` だけはリポジトリの管理下に置きたいので、`.vim/.gitignore` を以下のように修正します*[1]。

```
bundle
!bundle/neobundle.vim
```

このように `!` をつけることで ignore されたファイルの中から選択して git の管理下に置くことができます。覚えておくと便利。

*[1] でも実は neobundle についてはインストールスクリプトがあるのでそれ使うのがいいと思います。

2.4 デプロイスクリプトの作成

dotfiles のデプロイは、初期設定で行ったようにシンボリックリンクで行います。これはデプロイスクリプトを用意すると楽です。お好きな言語で書くといいと思いますが、せっかくですのでクロスプラットフォームで書きたいですね。私は Python が好きなので Python で書いています。シェルスクリプト支持層が多いですがシェルスクリプトってなんだかんだ方言あるよね。

リスト 2.2 deploy.py

```
#!/usr/bin/env python
# coding: utf-8

import os
import re
import sys

ignore = ['.git', '.gitignore', '.gitmodules']
dotfiles = re.compile('\.\w+')
pwd = os.path.dirname( os.path.abspath(__file__) )

if len(sys.argv) < 2:
    cmd = lambda a, l: None if os.path.islink(l) else os.symlink(a, l)
elif sys.argv[1] == "unlink":
    cmd = lambda a, l: os.unlink(l) if os.path.islink(l) else None

for f_name in os.listdir(pwd):
    match = dotfiles.match(f_name)

    if match is None or f_name in ignore:
        continue

    actual = os.path.join(pwd, f_name)
    link = os.path.join(os.environ["HOME"], f_name)
    cmd(actual, link)
```

ファイル名の先頭が `.` で `ignore` に含まれていないファイルについて、ホームディレクトリにシンボリックリンクを張ります。`ignore` には dotfiles リポジトリ事態に必要な設定ファイルを登録します。そんなスクリプトができればオーケー。アンインストール用の工夫もあると便利です。

2.5 インストールスクリプトの作成

つい最近まで私はデプロイヤを書くだけで終わらせていたのですが、恐ろしいことに世の中には「git clone してデプロイヤを走らせるのすらダルいわ」という人がいるのです。エンジニアの真髄は怠惰といいますがここまでとは。私も修行が足りませんでした。せかせかと二行もコマンドを打つのは遅れた愚か者の仕事なのです。一行で dotfiles のインストールを済ませましょう。

2.5.1 準備

インストールのためのスクリプトを書きます。ここで OS のチェックとそれに対応した設定などを行うので、コマンドを多用するというプログラムの性質上シェルスクリプトで書いた方が楽です。デプロイヤはさっきの Python スクリプトとして、インストーラのサンプルを書いておきます[*2]。

リスト 2.3 install.sh

```
#!/bin/sh

if [ -d dotfiles ]; then
    pushd dotfiles
    git pull origin master:master
else
    git clone https://github.com/<name>/dotfiles.git
    pushd dotfiles
fi

python deploy.py

git submodule init
git submodule update

popd
```

dotfiles を git clone してデプロイスクリプトを走らせて、サブモジュールがあれば展開するというスクリプトです。

キュートなバナーを表示したり大儀なインストールスクリプト組んでる方もいらっしゃるので色々調べて作ってみると面白いと思いますぞ。

[*2] [] は bash や ksh などの built-in なのでその辺のモダンなシェルじゃないと動かないでがんす。

2.5.2 installing dotfiles

やることは Web クライアントを使ってインストールスクリプトだけ引っ張ってきて、パイプでシェルに流し込む感じ。例として `curl` と `wget` を書いておきます。本誌の幅の関係で変なところで改行入れてますがワンライナーでインストール完了です。

```
# curl
curl \
https://raw.githubusercontent.com/<name>/dotfiles/master/install.sh \
| sh
# wget
wget -qO - \
https://raw.githubusercontent.com/<name>/dotfiles/master/install.sh \
| sh
```

超クール。

はい。ワンライナーでのデプロイができたところで dotfiles の記事を結びとさせていただきます。目指せ dotfiles マスター。

第 3 章

厨二病をもう一度こじらせて PSP 向けプログラムを作る

ﾅｯﾂﾂうにﾅｯﾂの抜けﾂｯﾂ殻ﾅﾂﾂ（@LuCielNoblesse）

中学生の頃に PSP 上で自作プログラムを動かしたあの頃の思い出が蘇る。蘇れ。
PSP で東方 SS サイトを巡回していたあの頃の思い出よ、蘇れ。

3.1 まず始めに

おはこんばんにちは。ﾅｯﾂﾂうにﾅｯﾂの抜けﾂｯﾂ殻ﾅﾂﾂです。

さて、皆様は PSP を覚えているでしょうか?

PSP といえば、Sony より発売されていたプレステ携帯機の初代です。今では PSVita も出て、美少女ゲー専用機に限りなく近くなっている印象がありますね。PSP-1000 から PSP-3000、同じ OS で PSP GO なんてのまで出ていました。今では 3DS で発売されているモンハンも、初プレイは PSP だったという方は多いのではないでしょうか。懐かしいですね。

今回書く記事では、PSPSDK を用いて PSP 向けの実行ファイルを作って動かしてみよう、という内容になっています。

ところで、「PSP　自作ソフト　作り方」なんかでググると卒倒しそうなぐらいに臭いブログと知恵袋が大量に出てくることかと思います。あの頃の俺らはこんなに厨二病をこじらせていたんだなぁ、と懐かしくなって悶えることでしょう。裏を返せばあの当時の時点で、中学生程度の知識でも PSP プログラミングができるほどに開発環境の開発が進んだ分野であった、という意味でもあると思います。

市販のゲーム機上で自分の書いたプログラムが動いた時のあの感覚を、今一度、楽しんではみませんか？

3.2 準備するもの

今回は windows 上で C 言語を用いて開発することにします。

3.2.1 開発環境の準備

- Cygwin
  C 言語のコンパイラー（gcc）と make.exe が欲しいので Cygwin を入れます。make と gcc のパッケージをインストールしておきましょう。
  コンピューター右クリックからのプロパティ→システムの詳細設定→環境変数にＣｙｇｗｉｎの bin フォルダをしっかり追加しておきましょう。C ドライブ直下にインストールしてある場合は path とかでも環境変数を新規作成して値に「C:/Cygwin/bin」を記述する感じです。これで make をコマンドプロンプトから実行できるようになりました。
- PSPSDK
  PSP 向けのプログラムを書く上では欠かせないものです。導入方法はさまざまですが、一番手のかからない方法を取りましょう。
  Minimalist PSPSDK(https://sourceforge.net/projects/minpspw/) を用います。
  以上のサイトから pspsdk-setup をダウンロードして実行し、ウィザードに従ってインストールを進めます。
  インストールが終わったら環境変数に「(PSPSDK をインストールしたディレクトリ)/pspsdk/bin」を追加して終了です。

3.2.2 実行環境の準備

書いたプログラムを実行する環境を整えます。エミュレータを用いて実行する方法と、実機で実行できる環境にするのどちらかでしょう。

エミュレータは恐らく、ppsspp(http://www.ppsspp.org/) を使うのがいいかと思われます。

デバッグ機能に関して逆アセンブルやメモリビューーが搭載されてるので強いかなと（深く弄ってないので適当に言ってます。ｽﾏﾇ。）

実機での環境構築については長くなるので詳しくは書きませんが、バージョン 1.5 へのダウングレードやカスタムファームウェアの導入さえできれば実行できます。HalfByteLoader なんかでも実行できるかと思います。

導入手順は「PSP CFW」等でググって出てくる、枕に顔を埋めて転げまわりたくなるような厨二病全開なブログ等に載っているかと思います。

3.3 Hello World を表示させる

というわけで環境が整ったところで、定番の Hello World を表示させてみましょう。
まず、Makefile に様々な情報を書き込む必要があります。

3.3.1 Makefile

Makefile の内容

リスト 3.1 Makefile

```
1  TARGET = Brainfuck
2  OBJS = bf.o
3
4  CFLAGS = -O2 -G0 -Wall -g
5
6  LIBDIR =
7  LDFLAGS =
8  LIBS =
9
10 EXTRA_TARGETS = EBOOT.PBP
11 PSP_EBOOT_TITLE = Brainf*ck
12 PSP_EBOOT_ICON = NULL
13 PSP_EBOOT_PIC1 = NULL
14
15 PSPSDK=$(shell psp-config --pspsdk-path)
16 include $(PSPSDK)/lib/build.mak
```

Makefile の解説

Makefile については授業で習う、もしくは習ったかと思いますので説明を省きます。

表 3.1 Makefile

マクロ名	説明
TARGET	コンパイラが生成するファイル名
OBJS	オブジェクトファイル名のリスト
CFLAGS	C 言語のファイルのコンパイル時のオプション
LIBDIR	ライブラリを読み込むディレクトリ
LDFLAGS	リンカのオプション
LIBS	読み込むライブラリ
`EXTRA_TARGETS`	出力するファイル名。PSP の実行ファイルの EBOOT.PBP を指定。
`PSP_EBOOT_TITLE`	EBOOT.PBP の PSP 上での表示名の指定
`PSP_EBOOT_ICON`	実行ファイルのアイコン。144*80 の png 形式
`PSP_EBOOT_PIC1`	実行ファイルの背景。480*272 の png 形式
`PSPSDK=$(shell psp-config --pspsdk-path)`	shell 関数で psp-config を pspsdk-path を PSPSDK に代入?
`include $(PSPSDK)/lib/build.mak`	build.mak の処理を行う

こんな感じです。

アイコンを動画にしたりだとか、選択中に音声を再生したりだとかもここで指定することができます。

リンクエラーが起きる場合は LIBS に読み込むライブラリが足りていない場合があるので、ここをチェックしましょう。

次のページに、Hello World を PSP に表示させるプログラムを書きます。

ただし Brainf*ck でなぁ！！！！

というわけで PSP 上で動くように Brainf*ck インタプリタを書きました。

3.3.2 ソースコード

Brainf*ck インタプリタのソースコード

リスト 3.2 bf.c

```
#include <pspdebug.h>
#include <pspkernel.h>
#include <pspdisplay.h>
#include <stdio.h>
#include <stdlib.h>

#define SRCLOC "bf.txt"
#define INLOC "in.txt"
#define BUFF_SIZE 3000

PSP_MODULE_INFO("Brainf*ck interpreter", PSP_MODULE_USER, 0, 0);
PSP_MAIN_THREAD_ATTR(PSP_THREAD_ATTR_USER);

int main(int argc, char **argv)
{
    pspDebugScreenInit();
    pspDebugScreenSetXY(0, 0);

    FILE *f, *f2;
    int *p;
    int buf[BUFF_SIZE] = { 0 };//バッファ
    char c;
    int i;//ループ回数の保持

    p = buf;

    if ((f = fopen(SRCLOC, "r")) == NULL) {
        return -1;
    }

    if ((f2 = fopen(INLOC, "r")) == NULL) {
        return -1;
    }


```

```
36     while ((c = (char)fgetc(f)) != EOF) {//ファイル終端まで回し続ける
37         switch (c) {
38         case '>':
39             p++;
40             break;
41         case '<':
42             p--;
43             break;
44         case '+':
45             (*p)++;
46             break;
47         case '-':
48             (*p)--;
49             break;
50         case '.':
51             pspDebugScreenPrintf("%c",*p);//出力
52             break;
53         case ',':
54             if ((*p = (int)fgetc(f2)) == EOF)return -1;//入力はファイルから
                   読む
55             break;
56         case '[':
57             if (*p == 0) {//指す値が0
58                 for (i = 0; c != ']' || i;) {
59                     if ((c = (char)fgetc(f)) == EOF) {
60                         return -1;
61                     }
62                     if (c == '[') {
63                         i++;
64                     }
65                     else if (c == ']' && i) {
66                         i--;
67                     }
68                 }
69             }
70             break;
71         case ']':
72             if (*p != 0) {
73                 for (i = 0; c != '[' || i;) {
74                     if (fseek(f, -2, SEEK_CUR)) {
75                         return -1;
76                     }
77                     if ((c = (char)fgetc(f)) == EOF) {
78                         return -1;
```

```
79                          }
80                          if (c == ']') {
81                              i++;
82                          }
83                          else if (c == '[' && i) {
84                              i--;
85                          }
86                      }
87                  }
88                  break;
89              }
90      }
91      fclose(f);
92      fclose(f2);
93
94      return 0;
95  }
```

最低な最低限の解説

- `PSP_MODULE_INFO`
  モジュールの情報の指定するマクロ。左から順番に、モジュールの内部名、カーネルモードかユーザーモードかの指定、メジャーバージョン番号、マイナーバージョン番号。
  モジュールの内部名は外からは見えない（PSP 上で表示される名前は Makefile に記載した方になる）
  カーネルモードかユーザーモードかの違いは公式ファームウェアでしか実行できないのがカーネルモード、どちらでも実行できるのがユーザーモードといったところ。らぼこ（PSP のセーブデータ改変によく使われていたプログラム）が CFW だと動かないのはらぼこがカーネルモードのプログラムであるため。
- `PSP_MAIN_THREAD_ATTR`
  メインスレッドの種類の指定。`PSP_THREAD_ATTR_USER` でユーザーモード。
- pspDebugScreenInit
  デバッグスクリーン機能の初期化
- pspDebugScreenSetXY
  デバッグスクリーンの座標の指定。座標の指定は左から順に x,y 座標だが、単位は文字であることに注意。
- pspDebugScreenPrintf
  出力には pspDebugScreenPrintf 関数を用いる。printf 関数と同じように、フォーマット文字列、可変個の引数が引数である。

なお、標準入力をどうするか考えた結果、ファイルから入力をとることにした。PSP の入力するインターフェースの呼び出す関数があるはずなので、後で調べたいとおもいますぅ↑

実行ファイルを生成するときはコマンドプロンプトでソースと Makefile のあるディレクトリ内で make を実行すればよい。エラー無く通れば EBOOT.PBP が生成されるはずである。

確認する場合は、これをエミュレーターで実行する、もしくは実機であれば ms0:/PSP/GAME/(何らかのディレクトリ)/EBOOT.PBP という構成にして実行すればよい。

ただし、ppsspp において、相対パスでのファイルの参照が上手くいかない。ppsspp のフォルダのあるディレクトリは認識する様子？

絶対パスで指定する場合は、umd0:もしくは ms0:がルートになる。

3.3.3 実機で実行した結果

生成されたファイルを PSP にコピーし、実行した。

実行した brainf*ck のコードは wikipedia の Brainf*ck のページからもってきたものです。

リスト 3.3 Hello,World!の Brainf*ck コード峻

```
+++++++++[>++++++++>+++++++++++>+++++<<<->.>++.+++++++..+++.>-.
------------.<++++++++.--------.+++.------.--------.>+.
```

Hello World!の出力をするコードになっています。

実際に実行した写真 (見えにくい)

図 3.1　Hello,World!

終了処理は書いていないので、電源を長押し（長上げ？）して切ります。

3.4　感想とか

今回記事を書くにあたって、久々に PSP を引っ張り出してきて、懐かしのサイトをググり、ちょっといじったわけですが、やっぱり書いたプログラムがゲーム機で実際に動くのは見ていて楽しいものですね。

厨房当時では知識もかなり浅く、ほっぽり出すのも早かったものですが、今の程度で弄ってみるとそれなりに遊べることに気づけたのでしばらくは PSP を弄っていろいろと動かしてみようかな、と思います。

第 4 章

猫でもわかるグラフィックスパイプライン

ころも

おそらく猫では理解できないです。この記事では全く知識の無い人を対象としたグラフィックスパイプラインの簡単な解説をします。グラフィックスパイプラインとは 3D モデルや画像などを GPU を利用して高速に描画するための手法（流れ）のことをいいます。

基本的にゲームを作る際には絶対に必要になる知識ですが、最近はグラフィックスパイプラインという言葉ですら知らない人でも単純なゲームなら作れちゃう時代です。しかし「ライブラリ等を使わないで 1 からゲームを作りたい」とか、「商業レベルのものを作りたい」、「ゲーム業界に就職したい」と考えているなら覚えておいて損はないというか絶対覚えて下さい。

ちなみにこの記事では DirectX11 を前提に解説をします。OpenGL とかだと使っている用語とか全然違ってくるのですが、意味的にはほとんど同じなので OpenGL でも十分に使える知識です。また、基本的な理解を目的としているのでかなり端折ってます。細かく厳密な話をしても混乱してしまうだけなので・・・

4.1　描画のために何が必要か

まず最初に 3D モデルを描画するために何が必要となるのかを考えてみます。後々重要になる単語は "こうやって" 囲ってあるので脳内メモでもしておいて下さい。

当然のことですが 3D モデルのデータは必要ですね。さて、一概に 3D モデルデータといってもこれにはいくつかの情報が含まれています。1 つは形の情報です。よくメッシュと言われており、グラフィックスパイプライン（以下パイプライン）においては "頂点データ" として処理されます。2 つ目の情報は色です。マテリアルとよく言われていますね。これがなかなか曲者な奴で、ただ赤いとか青いとか最初から決まっている訳ではなく、様々なパラメータから最終的な色が計算される

ものなんです。この "色パラメータ" がデータとしてあるわけです。

3D アプリケーションというのは別の言い方をすれば数学的な 3D シミュレーションです。シュミレーションじゃありませんよ！シミュですよ！シミュ！映画の撮影を想像してください。俳優がいて、カメラがあって、照明があって・・・、3D アプリケーションでも同じように仮想ワールドを定義して 3D モデルやカメラをワールド上に配置します。ということはそれぞれの 3D モデルとカメラにワールド内での "姿勢情報" があると言うことです。姿勢は座標、向き、大きさの 3 つで構成されていて、行列で表現しますがそこまでは解説しません。

今までに頂点データ、色パラメータ、姿勢情報の 3 つがでてきました。最終的にはこれらをもとに 3D モデルをスクリーン上の正しい位置と向き、正しい色で出力しなければなりません。そのためには "出力ピクセル座標の計算" と "出力ピクセル色の計算" をしてやる必要があります。この 2 つは非常に重要で、ぶっちゃけパイプラインそのものと言ってもいいでしょう。

4.2 グラフィックスパイプライン

さて、ここからはパイプラインの解説に入りましょう。ところがいきなり解説するのはちょっと荷が重いので、まずは超簡略化してついさっき挙げた 5 つの要素だけを使って解説します (図 4.1)。

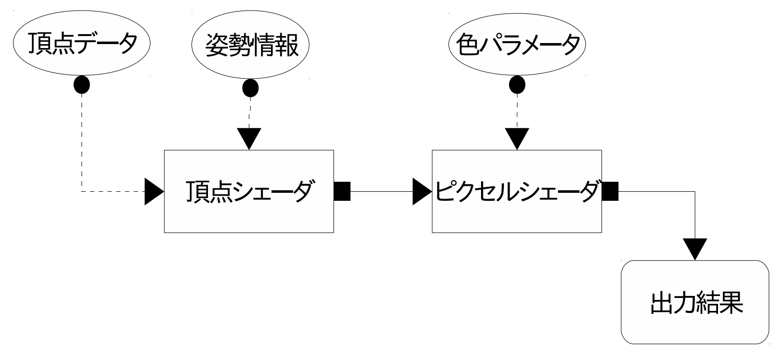


図 4.1 簡略化したグラフィックスパイプライン

パイプラインは様々な計算を総称したものでそれぞれの計算はステージと呼ばれます。四角で囲ってあるのが各ステージ、丸は入力パラメータを表しています。順を追って解説をしますと、まずは頂点シェーダステージを実行します。ここでは姿勢情報を使い、行列の掛け算などを行って入力された頂点データをスクリーン上のどのピクセルに投影するかを計算します。ここで 3D だったモデルが 2D になるわけです。ピクセルの座標が決まったら、そのピクセルに対してピクセルシェーダステージを実行します。ここで色パラメータを使ってピクセルの色を決めます。終わり！

基本的な流れはたったこれだけなんです！でも大変なのはここからです。では、本当のパイプラインをご覧いただこう (図 4.2)。

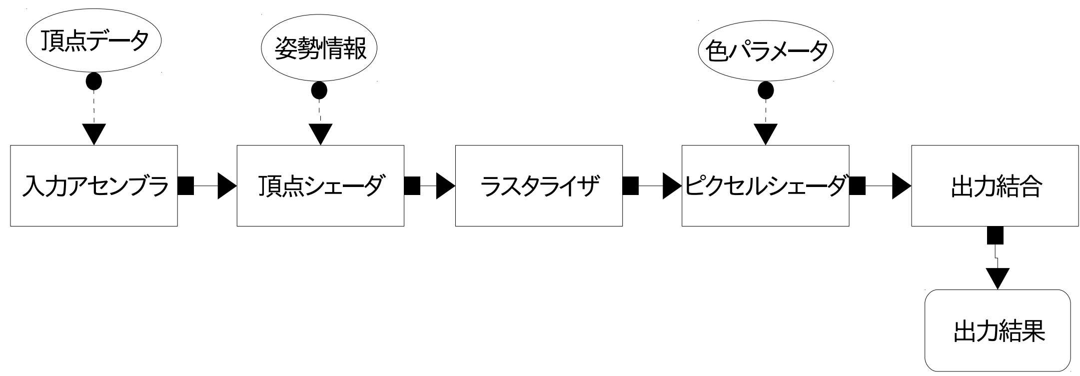


図 4.2　本当のグラフィックスパイプライン

見た目的には 2 倍くらいに増えましたが、理解しなければならないことは 2 倍以上に増えています。

4.2.1　入力アセンブラステージ

このステージの目的は入力された頂点データをプリミティブ化し、次の頂点シェーダステージへ頂点を１つずつ流し込むことです。プリミティブ化ってのは、まず頂点データといっても一度に全部処理するわけではなくて、線や 3 角形といった形に分解してそれらを１つ１つパイプラインで処理していきます。この分解する作業をプリミティブ化といいます

4.2.2　頂点シェーダステージ

このステージの目的は入力された頂点をスクリーン上のどの位置に投影するかの「比」を計算することです。頂点シェーダには必ず 1 つの頂点が入力され、1 つの頂点を出力します。先ほどは座標を計算すると言いましたが、本当は比を計算します。というのもスクリーンの大きさ（モニターの解像度）は環境によって違いますよね。例えば僕の場合ノート PC は 1366 × 768 でデスクトップ PC は 1920 × 1080 のモニターを使ってます。座標を直接指定してしまうと、ある環境では画面の真ん中に、一方他の環境では左端に投影されてしまうなんてことが起こってしまうのです。さて、この比ですが (x, y, z) の 3 次元ベクトルで指定します。x は横軸の比で、y は縦軸の比で双方 $-1 \sim +1$ の範囲で指定します。z は奥行きを表しますが範囲は $0 \sim +1$ です。なぜ z なんていう

奥行きが必要かといいますと・・・後で話します。このステージの目的をまとめると、3D 空間上の頂点座標 (x, y, z) をスクリーン上の座標比 (x, y, z) へ変換するわけです (図 4.3)。

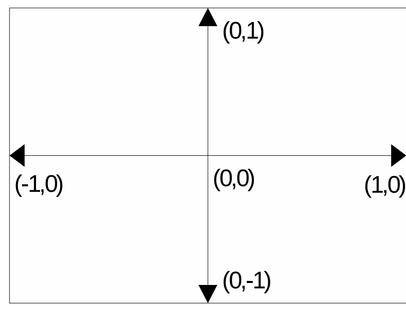


図 4.3 スクリーン上の座標比

4.2.3 ラスタライザステージ

ラスタライズとは rasterize ですが、これはベクトルで表された情報を点の集合で表現しなおすという意味です。このステージに到達すると入力アセンブラステージで分割された頂点プリミティブがスクリーン上の座標比で表現されています。このステージの目的は、座標比から実際のピクセル座標を計算し、色をつけるピクセルを決定します。つまり 3 角形とかのプリミティブをピクセルの集まりで表現しなおします。これにはビューポートというスクリーンの解像度を表す情報を使います。あくまでパイプライン全体の流れを理解したいだけなので詳しくは解説しませんが、ビューポートを使ってピクセル座標を計算します。そういうものなんです。実はこのステージでは描画の必要がない頂点の間引きも行っていて、裏面を向いているプリミティブはこの時点で計算を打ち切り、描画されなくなります。皆さんはゲームをやっていて、普段入れない建物の中にバグとかで入り込んでしまったときに、中から外の様子が透けて見えるとかって経験ありますよね？あれはまさにこの裏面が関係していて、メッシュには裏表があるんです。気になる人はこの世で最も博識な google 先生に聞いて下さい。

このステージをまとめると、スクリーン上の座標比 (x, y, z) を実際のピクセル座標 (x, y, z) に変換し、そのピクセルに色をつけるかどうかの決定を行います。まだ z 値という謎が残っていますが、次の次くらいで解説するのでちょっと待ってて下さい。

4.2.4 ピクセルシェーダステージ

このステージは簡略化したバージョンと同じなでの特に解説はありません。ピクセルの色を決定するだけです。

4.2.5 出力結合ステージ

ピクセルシェーダステージが終わったのでピクセルの色まで決まりました。が、まだやることは残っているんです。このステージでは本当の意味でのピクセル色を決定します。実は今までの計算で得た色をそのまま画面に出力するわけにはいかないのです。というのも今までの計算では 3D モデルの前後関係を考慮していません。当たり前ですが 2 つの 3D モデルが画面上で重なっていると後ろの 3D モデルは描画されません。これを実現するために登場するのが z 値なんです。ラスタライザステージで計算したピクセル座標 (x, y, z) の z 値はそのピクセルに描画される 3D モデルがどれくらい後ろにあるかを表しています。もしかすると他の 3D モデルが既に同じピクセルに描画されており、その z 値が今描画している 3D モデルよりも小さい場合は他の 3D モデルのほうが前面にあるということですので、今計算したばかりのピクセル色は無かったことにしてしまいます。こうすることで 2D 空間上での前後関係を維持しているんです。

本当は色の合成処理もあるんですが、今のところはスルーでいいでしょう。ということでまとめると、このステージではピクセルシェーダで計算した色で前後関係を維持できるようにピクセルを塗りつぶします。色の合成処理も考慮して一般的に言うと、ピクセルシェーダで計算した色と既に計算済みの色から最終的なピクセルの色を決定します。

4.3 補足とか

一通りパイプラインの解説をしましたが、各ステージの中でも頂点シェーダとピクセルシェーダはちょっと特別でプラグラマブルシェーダと呼ばれています。プログラマブルというのはプラグラムによる制御が可能と言う意味で、High Level Shader Language（HLSL）というシェーダー言語を書くことになります。なので 3D 座標を 2D に変換したり、ピクセルの色を計算するために必要なパラメータを入力することが出来ます。様々なシェーダプログラムとパラメータを組み合わせることで、より色鮮やかでリアルなグラフィックを作りだせるのです。

この記事では DirectX11 の解説でしたが、最初に言ったように OpenGL でもほとんど変わりません。最近は DirectX12 もでていますが、12 と 11 のパイプラインは全く同じです。ちなみにちょこっと出てきましたが 3D グラフィックスをやりたかったら数学の知識は絶対必要です。特に行列とかベクトルなどの線形代数学と幾何学ができないと話になりません。とはいっても高校数学の知識である程度は事足りるので、高校で授業中寝てばっかりだった人は頑張って復習してください。がんばれ ♡ がんばれ ♡。あれ？今の高校って行列やらないんだっけ？えらいこっちゃ。

役に立ちそうな本とか置いておきますね

ゲームプログラマになる前に覚えておきたい技術

（平山尚．　秀和システム）
通称セガ本、セガの新人教育カリキュラムを元に、ゲームプログラマに必要な基礎知識が沢山載っている。難易度は初から中

ゲームコーディング・コンプリート 一流になるためのゲームプログラミング

（Mike Mcshaffry．　ソフトバンククリエイティブ）
ゲームプログラミングに慣れてきたら読むと良い。1 ステップ上のテクニックが載っている。難易度は中

ゲームエンジン・アーキテクチャ

（ジェイソン・グレゴリー．　SB クリエイティブ）
著者は「アンチャーテッド」とか「The Last of Us」とかのリードプログラマ。世界中で評価されるようなゲームを作るための技術が載っている。とてもおすすめ。難易度は中から高

ゲームプログラマのための C++

（マイケル・ディックハイザー．　ソフトバンククリエイティブ）
良い意味でも悪い意味でも基本的なことが網羅されている。高度なゲームを作ろうとしている人にとっては物足りない内容だが、初心者にとってはとても分かりやすい。難易度は初から中

実例で学ぶゲーム 3D **数学**

（Fletcher Dunn，Ian Parberry．　オライリージャパン）
3D グラフィックスに必要な数学の知識のほとんどが載っている。数学力が足りないと思ったらこれを読むべし。少なくとも僕が高校生のときにやった内容がほとんどだったので難しくは無いはず。

Game Programing Gems **シリーズ**

（ボーンデジタル）
最先端のゲーム開発技術が練り込まれた本。論文集のようなもので、それはそれは世界中からの色んな記事が集まっている。難易度は超高。僕もまともに理解できない。

GPU Gems **シリーズ**

（ボーンデジタル）
こちらは上の本の GPU を使ったゲーム開発技術に焦点を当てているバージョン。これも難

易度は超高。

あ、そうだ (唐突)。ゲーム開発には直接関係ないけど、プログラミング全般に役立つ本もあるゾ。プログラムをバリバリ書けるようになってきたら読むゾ。

UNIX という考え方―その設計思想と哲学

(Mike Gancarz.　オーム社)

UNIX 開発チームがその経験から得たありがたいお言葉、思想が載っている。

ストラウストラップのプログラミング入門

(ビャーネ・ストラウストラップ.　翔泳社)

C++ 作者による入門書。とはいってもちょっとレベルが高くて、頭の良い初心者向けって感じだが、とても分かりやすいので初心者もそうでない人も 1 度は読んだ方が良い。

オブジェクト指向入門 原則・コンセプト

(バートランド・メイヤー.　翔泳社)

オブジェクト指向って何？って思ったらまずはこれ。僕はこれを読んで世界が変わりました。オブジェクト指向を語る前に、実践する前に、是非是非是非是非読んで欲しい。

オブジェクト指向のこころ

(アラン・シャロウェイ.　丸善出版)

上の本はあくまで理論的な話。こちらは実践的な話になる。この 2 冊を読めばオブジェクト指向マスターになれる。

アルゴリズムイントロダクション

(T. コルメン，R. リベスト，C. シュタイン，C. ライザーソン.　近代科学社)

アルゴリズムはめちゃくちゃ大切です。とりあえずこれ読んどけば問題ないです。

第 5 章

珠玉の Ruby プログラミング

Zodiac(ID:zodi_G12)

おはようございます. どうも. ゾディアックです. いつも主にちょっとした数学における なんかと, ネットワークやセキュリティ, アルゴリズムの学習に励んでおります. プログラミング言語は C と Ruby と Haskell が書ける（ようになりたい...). 僕は主に Twitter(ID:zodi_G12) にいます. フォロー非推奨です. ところで "PMOB" って一体, 何の略なんですかね？

5.1 方針

今回は Ruby プログラミングについて書かせて戴く. 新入生向きかというと微妙だが, そこまで難しい内容でも無いと思われる.

5.2 動機

Ruby プログラミングに関する記事を書こうと思った経緯は, 極些細なことである. 一つ目に, 前回の部誌で Haskell プログラミング (?) に関して書いたので, 今回は別の言語で書こうと考えていたこと. そこで, 消去法的に僕が書ける言語は,C 言語と Ruby しかないため,Ruby の記事を書くことになった. 何故 C ではなくて Ruby なのか. これは Ruby は, 書いていて楽しいからというのと,C だとちょっと面倒くさい題材であったためである. 二つ目に, 休暇中にとある "題材" に出会った（正確には存在を思い出した）ことだ. そして, その問題を Ruby で解けたことが, まあまあ嬉しかったので, この問題の解法について書くことを決めた.PMOB は数学分野における学術的な話の出来るような集団であるので, 数学の話題でも良かったのだが, 僕の理解が及ばなかったため, 数学に関して書くことは, 今回は控えさせて戴く. また今度にでも...

5.3 Ruby とは

まず,「Ruby とは何であるか?」という哲学的な問いに，僕が知っていて理解している範囲で答えていこうと思う.

5.3.1 誰が作ったのでしょう?

Ruby は日本人の まつもとゆきひろ氏が開発したプログラミング言語である． まつもとゆきひろ氏は「Ruby の言語仕様策定において最も重視しているのは，ストレスなくプログラミングを楽しむことである (enjoy programming)」と述べている. 成る程書いていて楽しいわけだ.

5.3.2 Ruby はスクリプト言語

次に,Ruby はスクリプト言語で,C 言語や Java のように "コンパイル" という作業を要しない. これは利点であるかと問わたならば, 四方八方から様々な回答が寄せられるのではなかろうか. 微妙なところである.1 つの機能として捉えることにする.

5.3.3 Ruby はオブジェクト指向プログラミング言語

重ねて,Ruby はオブジェクト指向プログラミング言語である. オブジェクトとは，プログラム上の手続きの対象を抽象化する概念である． また, メソッドとは，オブジェクト指向プログラミングにおいて，各オブジェクトが持っている自身に対する操作のことである． オブジェクトは "データ" と "手続き" から成っているが，メソッドは, その "手続き" の部分に当たる． オブジェクト指向プログラミング言語の構成要件としては，以下のものがある.

- カプセル化
- ポリモルフィズム

クラスベース方式や Mixin 方式等には今回は触れない.(豆知識:クラスとインスタンスは抽象度が異なるものである)

カプセル化

オブジェクトが管理するデータを, オブジェクトの外部から直に操作できないようにして, 変更したり, 参照したりする際には, メソッドを呼び出させるような方式にすること.

ポリモルフィズム

1 つのメソッド名が複数のオブジェクトに属すること，そして，そのオブジェクトによって異なる結果が得られることを，ポリモルフィズム（多相性または多態性）という． ".to_i" メソッドや，".to_s" メソッドは，対象となるオブジェクトによって結果が異なる.(なお，これらのメソッドは今回用いる)

5.3.4 Ruby はマルチプラットフォームな言語

また,Ruby はマルチプラットフォームな言語であり，様々な環境，Unix 系 OS や Windows で動く．移植性が高い．

5.3.5 Ruby はオープンソースソフトウェア

そして,Ruby はオープンソースソフトウェアである．多くの様々な人々が関わり，開発がなされている．これは非常に高い可能性を秘めていることを示している．

5.3.6 初めて Ruby に触れた感想

中でも，情報科の青二才であるところの僕が，今まで書いてきた中で不思議に感じた機能が "動的型付け" という型システムの性質である．これも奥が深いのだが，簡略化すると，実行時に実際の値から型を決定するというものだ．よく知りたい読者は，型システム等の本等を読んで戴くと良い．

5.3.7 提案

これまで，ざっくりと説明させて戴いたが，この時点において読者諸君は，分からないことが少なからずあると思われる．したがって，この時点で知識を曖昧なものにしてしまわないためにも，この機会に，より理解を深めてみてはどうだろうか．

5.4 題材の詳細

今回の題材は,Google の広告で自然対数の底，いわゆるネイピア数の，初めの 10 桁の素数を見つけて,「初めの十桁の素数.com」にアクセスしてみよう，というものである． "first 10-digit prime found in consecutive digits of e（自然対数の底 e の中で最初に出てくる連続した 10 桁の素数)" これが本文である．

5.5 解法

大体の方針としては，予め 6 桁の素数を用意しておいて，ネイピア数を生成し，先頭から 10 桁ずつ抽出して，素数判定をしていき，一つでも 10 桁の素数が発見されれば，探索を終了するプログラムを Ruby で作成するというものである．解法の主な難解な手順は，以下のようなものであった．

1．10 桁の整数を完全に素因数分解できる程度の素数生成
2．巨大桁のネイピア数の生成
3．ネイピア数から 10 桁の整数を抽出する手段

5.5.1 素数生成

素数生成の際の素数判定においては，エラトステネスの篩（ふるい）という方法を用いた． エラトステネスの篩とは，初めに，必要個数の配列を確保し,1 から昇順に代入していく．例えば，初めの "2" を先頭値として，配列を順々に見ていくと，この "2" を因数に持つ値が数多くある．そこで，"2" を因数に持つこれらの値を，後々，素数判定の候補から外すことで，計算量を減らすことができるといった画期的なアルゴリズムである．さらに，例として，"100" までの素数を生成したいときには，先頭値の最大値は，"100" の平方根，つまり "10" までで良いのである．正当性を示すような証明は割愛させて戴くので，御了承戴きたい．分からなかったら調べて戴きたい.(自明ですよね)

5.5.2 ネイピア数の生成

ネイピア数の生成には "module BigMath" を用いた．文字通り，結構巨大な桁の値を生成できる．

5.5.3 ネイピア数から 10 桁の整数を抽出

ネイピア数から 10 桁抽出する方法としては,[i] から [i+9] までを ".to_s"（配列を文字列として変換するメソッド）を用いて抽出．それを ".to_i"（文字列を 10 進数の表現と見なして整数に変換するメソッド）を用いて素数判定する．

5.6 プログラムコード

このプログラムを用いることで，先程の問題で，正しいものと思われる素数を抽出できた． なお，このプログラムは Ruby で書かれている.#それはそう

リスト 5.1 zodiac-G12/e_10keta_sosuu.rb

```
1  require 'bigdecimal/math'
2
3  N = 100000 # マクロ
4  e = BigMath::E(N).to_s # ネイピア数の配列
5  sosuu = [2] # 最初は探索リストで最終的には素数リスト初めは探索リストの先頭値が代入
6  base = Array.new(N){|idx| 1 ++ idx} # "1..N" までの昇順の値を要素とする配列の生成
7  base.slice!(0..1) # "1,2" は邪魔なので除く
8  counts = Math::sqrt(N) # 探索リストの先頭値の上限値は "N" の平方根
9
10 # "base" で素数でない値は "0" に置き換える
11 i = 0 # 探索リストのカウンタ用に用意
12 while i <= counts
13   j = 0 # 配列の最大値まで読み込むためのカウンタ用に用意
14   header = 0 # 探索リストの先頭値
15   while j <= N
16     if base[j].to_i % sosuu[i].to_i != 0 # 余りが "0" の素数
17       if header == 0 # 先頭だけ
18         sosuu << base[j] # 素数なら "sosuu" に代入
19         header = 1 # これ以降は "sosuu" に代入しない
20       end
21     else
22       base[j] = 0 # 素数でないとき値を  "0" に置き換える
23     end
24     j += 1
25   end
26   i += 1
27 end
28
29 # 素数を "sosuu" に代入していく
30 # 先頭値の素数は既に入っているのでそれ以外,
31 i = 0
32 header = 0 # 先頭の素数は代入しない
33 while i <= N
34   if base[i] != 0
35     if header == 0 # 初めて発見する素数はもう代入済み
36       header = 1 # 次の素数から代入していく
```

```
37      else
38        sosuu << base[i]
39      end
40    end
41    i += 1
42  end
43
44  # ネイピア数の素数判定
45  i = 3
46  counts = 0 # 先頭からの素数の個数のカウンタ用に用意
47  while i <= N - 10 # ネイピア数の桁の分の回数だけ素数判定
48    j = 0
49    while j <= sosuu.length - 1 # 素数リストの素数で割れるか判定
50      if e[i..i + 9].to_i % sosuu[j] == 0 # "10" 桁抽出した値が素数でない
51        break
52      end
53      if j == sosuu.length - 1 # 全ての素数で試し "10" 桁抽出した値が素数
54        p e[i..i + 9] # "10" 桁の素数を出力
55        counts += 1
56        break
57      end
58      j += 1
59    end
60    if counts == 10 # 先頭から "10" 個の素数を列挙したら終わる
61      break
62    end
63    i += 1
64  end
```

5.7 参考文献

- たのしい Ruby "第 4 版"
- 題材提供: http://blog.livedoor.jp/blogartmode/archives/51763852.html
- 我が脳味噌
- 色んなページを跳ねて回ったピョン

5.8 良書と思われる書籍

風の便りで良い噂を聞いた本等も含めて，かなり良さ気で勉強になりそうで力になってくれそうな本達をご紹介いたします．

- たのしい Ruby "第 4 版" (※初心者向け)
- メタプログラミング Ruby ("推せる")
- Effective Ruby ("推せる")
- パーフェクト Ruby
- Ruby のしくみ "うんちゃら Scope"
- Ruby によるクローラー開発技法

5.9 あとがき

最後まで熟読戴き感謝いたします．こういった内容のものを書くのは不慣れな点が多くて，誰が読んでも「わかり哲也」と感じるような，完璧な文章にまとめ上げることが出来ませんでした．お詫び申し上げます．それに，内容も表現も，かなり冗長なものになってしまったと感じております．「わからない哲也」といった感想をお持ちの方は，僕の Twitter にリプライを飛ばして戴くか,PMOB の活動中に来て戴けたらお教えいたします． 僕は,PMOB の活動にほぼ毎回参加しておりますので，十中八九お会いすることができると思われます． なので，何か質問がございましたら，上記のような手段でお聞きください． 最後に 1 つよろしいでしょうか？ 圏論はいいぞ．

第 6 章

Linux のすゝめ

しゃみそん（@_sham258）

鯖じゃなくてデスクトップ用途で Linux を使おうのやつ．パソコンって何？ OS ってなんですか？という方には辛い記事ですのでご注意を．

6.1 始めに

Linux （リナックス）は，「Unix ライクな OS カーネルである Linux カーネル，およびそれをカーネルとして周辺を整備したシステムのこと」である．

本記事では「Linux について」と，「デスクトップ用途での，Linux を使うことで円滑な開発ライフを送ること」についてまとめていく．

読者の中に Linux ディストリユーザの人間がどれだけいるのかわからないが，Unix ないし Unix ライクなシステムにはハッカーが住んでいるので，ハッカーの住居にｺﾝｺﾝとノックしてもしもーしと入っていきたい．

最初に Linux は「Unix ライクな OS カーネル」と述べたが，まずはカーネルついてざっくり見ていこう．

6.2 Linux カーネルとはなんぞや

私は「カーネル」と聞いたときはﾎﾟﾝｯ頭に浮かんだのはロックマンエグゼであったが，ここでいうカーネルはそれとは違う．

「カーネル」を端的に表すと OS の「心臓」であり，「コア」であり，「核」である．カーネルのがやることの中には，以下のものがある．

- 「デバイスの管理」
  - デバイスドライバでハードウェアの入出力を制御する．カーネルがこれを行うことで，ハードウェアが抽象化され，ソフトウェア側から触りやすくなっている．
- 「プロセスの管理」
  - プロセスが一度に利用できる CPU は一つであるため，特定のプロセスが CPU を専有しないように，また，効率よくプロセスが働くように管理している．各プロセスには，「PID」という ID が割り振られており，Unix 系の OS では `ps` コマンドでプロセスの様子を観察できる．アリの巣を観察する感覚で `ps aux` とやってみると良い．
- 「メモリの管理」
  - プログラムを実行する際には，実行する場所が必要だが，その場所となるのがメモリだ．実行中のプログラムはシステムからは「プロセス」と見なされ，様々なお仕事をこなしていく．カーネルは，様々なプロセスがお仕事をしていく過程で，各プロセスの仕事場（メモリ）の配分を管理しているのだ．
- 「システムコールの提供」
  - システムコールというのは，OS のカーネルが可能なものを，OS を使っているユーザのプログラムから使えるようにする仕組みのことだ．例えば，C 言語で `printf` 関数を使うときは，システムコールの `write` 関数を用いて実現している，という具合だ．個人的に API 的なものだと解釈している．

以上からざっくりまとめると，カーネルというのは「パソコン上でのプログラムの快適な動作のために，ハードウェアの抽象化と，その管理をしているもの」という感じだろうか．

6.2.1 Unix ライクな OS カーネル Linux

少々脱線するがよく間違えられるのでここに記述する．

再帰的頭字語として「Linux - Linux is not Unix」が挙げられる Linux は Unix ではない．そもそも，Unix とは 1969 年にベル研究所の Ken Thompson，Dennis Ritchie らが作った OS のことであり，Linux とは別物である．

では Linux とは何なのか，という疑問が湧くが，Linux とは，まだ当時学生であった「Linus

Torvalds（リーナス・トーバルズ）」という１大学生が，勉強のために自分で実装した，Unix「っぽい」OS のことである．

言ってしまえば，普通に触る分には Unix も Linux も殆ど変わらないが，違うものだ，ということは分かっておくと良いかもしれない．

6.2.2 Linux ディストリビューションとは

Linux は「カーネル」と「Linux をカーネルとして周辺を整備したシステム」とあるが，前者を「Linux カーネル」，後者はしばしば「Linux ディストリビューション」と呼ばれる．

ディストリビューション（distribution）の意味は「流通・分布」を意味しており，Linux カーネルを含み，設定済みで実行可能の様々なソフトウェア群（コンパイラ，ブラウザ，エディタなど）によって，構成された OS のことを Linux ディストリビューションと呼ばれ，具体的には「Ubuntu」や「CentOS」などがある．

また，Linux ディストリの人気なものには，大きく分けて，「Debian 系」と「RedHat 系」と「Slackware 系」があり，独立したディストリを含め，各系列には以下のようなものがある．

- Debian 系
  - Debian GNU/Linux
  - Ubuntu
  - Steam OS
- RedHat 系
  - Fedora
  - Red Hat Enterprise Linux
  - CentOS
- Slockware 系
  - openSUSE
  - Slackware
- その他独立系
  - Arch Linux
  - Gentoo Linux

もちろんこれは全てではないため，読者各位には，ぜひ自分の好きなディストリを見つけていただきたい．始めるのにオススメなものは「Ubuntu」であり，ある程度 Ubuntu 等を触った各位には「Arch Linux」を勧めたい．

6.3 で，何がいいの？

Linux を使うと何が良いのか，と言う点を上げていく

- カスタマイズ性が高い
- 基本的に無料
- プログラミングの環境設定が簡単になる
- シェルが強力
- etc...

一つずつ見ていく.

6.3.1 異常なまでのカスタマイズ性

狂気を感じるほどにカスタマイズ性が高い．読者はどんな Linux PC を運用する際に，無限の構成ができる.

Windows のデスクトップ画面のような, スタートボタンのあるデスクトップ（Window Manager）を採用することも可能であり，Mac の Dock 的なものを追加することもできれば，マウスを殆ど使わず，キーボードからの操作で完結するものにカスタマイズもできる．最初のログイン画面をリッチにすることもできるし，UI がしょっぱくなるが，動作がとても軽いものを採用，などなど.

各々の PC に一番合うものを試行錯誤して，各々の趣味趣向宗教にあう自分だけの PC を作ることができるのだ.

6.3.2 人生と一緒で無料

全てが無料という訳ではないが，無料のものが多い．Ubuntu も CentOS も無料だ．じゃあどうやって開発しているの？という疑問がある.

これは基本的に Linux 好きな開発者たちの愛によって開発が勧められてる，と言っていいだろう．つまりはボランティアだ．開発者は基本的に普段の仕事のあいまで，そして趣味で Linux やそのディストリの開発を行っている.

また無料であることも相まって，容易に OSS（オープンソースソフトウェア）として開発できることから，誰もが開発者側に周りやすい仕組みができているという点でも良いだろう.

かといって有料の Linux もある．有料で有名なのは Red Hat Enterprise Linux だろうが，商用サポートという強いサポートが受けられるのが有料の最大の利点だろうか.

6.3.3　プログラミングが簡単？

基本的に Linux ディストリには「パッケージマネージャ」がついている．このパッケージマネージャがディストリに付属していることで，環境設定が非常に容易になっている要因として挙げられる．

例えば，「Ubuntu 入ったの PC で，エディタを Emacs で，Python3 を使ってなんか作りたいなぁ．開発するときは Git でバージョン管理したいしうーん」とよくある状況が発生した際には，環境は以下のコマンドを叩けばおしまいである．

```
$ sudo apt-get install emacs python3 git
```

あら簡単．

Debian 系では「apt-get」や「aptitude」，RedHat 系では「yum」，Arch Linux では「pacman」「yaourt」などなど，様々であるが，コマンド叩き方の違いこそあれど，機能としては違いはほぼ無い．様々なパッケージをサクッとインストールして，さっさと環境構築を終わらせたい怠惰なハッカーにはピッタリだ．

6.3.4　シェル

Linux にはシェルというものがあり，これは OS を取り囲む外殻のようなものである．ユーザはターミナルエミュレータを介して，シェルにコマンドを渡し，様々なプログラムを走らせることができる．

例えば，`ls` コマンドを使えば，今いるディレクトリの中身を見ることが出来たり，`cd` コマンドでディレクトリを移動，`less` コマンドでファイルの中身をチラ見したできる．

「別にターミナルエミュレータ使わなくて良くない？」と思う読者もいると思うが，これで終わりではない．シェルで実行できるコマンドは「パイプ」でつなげて組み合わせることができるのだ．

例えば「.md の拡張子のファイルがどれだけあるか見たいと言うとき」は，`ls` コマンドと `grep` を組み合わせて，

```
$ ls -la | grep *.md
```

でおしまいである．すごい．

ユーザが触る際は，基本的にターミナルエミュレータ（Ubuntu では Gnome Terminal を使うのが良いだろう）を通してコマンドを叩いていくことになる．ターミナルエミュレータにも様々であり，こだわってみると面白い．

ディストリのデフォルトの Unix シェル（Unix 系システムで使われる，シェルコマンドを解釈できる，コマンドラインインタプリタの意味）として有名なものが「Bash」と呼ばれるもので，Linux 界のデファクトスタンダードとなっている．が，もちろん他の Unix シェルもあり，「zsh」

や，「fish」などもあるのでグーグル先生に聞いてみると良いだろう．

6.4 インストール？

実際に Ubuntu インストールの手順を記述しようかと考えたが，冗長な記事になりそう（本音はめんどい）と思ったので，インストールする際の Tips をまとめておこうと思う．

ぶっちゃけ，この記事をここまで読んで初めて Linux を入れるって人は少ないだろうが，つらつらと記しておこう．

6.4.1 インストールの準備

Unix 系ユーザ各位は `dd` コマンドを使えばいいが，Windows ユーザ各位には LiveCD（or LiveUSB）を作るだけでも少々面倒な部分があると思う．

Windows ユーザにはぜひ「Rufus*[1]」を勧めたい．

詳しくはググって補完していただきたいが，他のインストール用ツールに比べ，冗長な部分が少なく必要最低限でまとまっており，インストールしたいディストリの ISO さえ取ってきてしまえば，容易に LiveCD 等を作れるのでオススメである．

また，インストールの際は出来るだけ UEFI ブートでインストールするほうが，PC の起動が早くなるので，できる限りは UEFI ブートのほうが良いと思われる．

6.4.2 パーティションの切り方と切った後

基本的にはデュアルブートする人の悩みとなるだろうが，切る際には `cgdisk` が役に立つだろう．また，GUI 環境があるのなら `gparted` もオススメだ．Windows ユーザ各位は，Windows で予め切っておく等もできる．

また，切る際に気をつけておきたいのが，「MBR（Master Boot Record）形式」で切るのか，「GPT（GUID Partition Table）」で切るのかという点である．

基本的に 2016 年現在では GPT 形式で切るのが主流であるが，これはブートローダに左右されるので，UEFI ブートなら GPT，BIOS ブートなら MBR，のようにするのが安定だろうか．

また，パーティションを切ったあとは，どのファイルシステムにフォーマットすれば良いかの知見が必要となるだろう．個人的な主観が大いに入っているが，さらっとファイルシステムについて眺めてみよう．

*[1] https://rufus.akeo.ie/

表 6.1 ファイルシステムのあれこれ

ファイルシステム	説明
ext4	大体はこのファイルシステムが使われているイメージがある．ext2 から派生してきた GNU/Linux のファイルシステムだ.
swap	スワップ領域において使われる．自分の PC のメモリ量に不安があれば作成しても良いだろう.
vfat	UEFI ブートの際よく Boot パーティションにおいて使われる．このファイルシステムは事実上の全ての OS でサポートされているようで，Boot パーティションに使われるのもそれが要因だろうか.
NTFS	Windows における基本的なファイルシステム．Linux で扱えないという訳ではないが，Linux のファイルシステムであれば選択する必要は無いだろう.
btrfs	「Better FS」という名をもち，現在不安定ながらも，将来 GNU/Linux 標準のファイルシステムになるのではないか，と期待されているファイルシステムである.

なお，私は「UEFI ブートで GPT でパーティションを分けて，"/" に ext4，"/boot" に vfat」という構成をしている．今度暇なときにルートを btrfs にしてみようかと思う．なお，グーグル先生に聞いたところ，"/home" とかを他のパーティションに分ける人とかもいるようである．利点等はぶっちゃけよくわからないが，やる人がいるということは何かしらの利点があるんだろうと思う．（各位で調べて頂きたい）

6.5 あなたの Linux に

Linux をインストールした人も，してない人も，Linux で使われるソフト群について記述するので，目を通してみて頂きたい．

6.5.1 ブラウザ

主に使われるブラウザについて紹介する．

Firefox

言わずと知れた Mozilla 社の燃え盛る狐さん．執筆時現在私は Firefox 派である．現在私は比較的低スペックの PC を使っているが，低スペック PC でも比較的快適に動くため，良いと思われる．

また，マウス操作に飽きたら Vimperator という Firefox の操作を vim 化して，マウス操作いらずにするということもできるのでやってみると面白い．

なお，Emacs 化については KeySnail というものがあるが，動作が安定しないイメージがあるので，インストールの際は調査が必要だろう．

Google Chrome（Chromium）

いつもお世話になっているあの Google 先生が作ったブラウザ．プロセス欄が大量に Chrome まみれになることでも有名．低スペ PC でなければ動作等は気になる範囲ではないし，ちょっとしたプラグイン等では優秀なものが多いので，その点ではこちらのほうが好きである．

だが，Linux 機の低スペック PC では少々重いかなーと感じているが，どうだろうか．Vim 化もできるようなので，UI を除いて考えると，FireFox とそこまで大きな違いは無いだろう．

luakit

Lua 言語で作られた軽量ブラウザ．起動こそ少々時間がかかるものの，操作はデフォルトで Vim バインドになっており，UI がシンプルで余計なものがあまりないのが良い．

が，メンテナンス頻繁にされていないという致命的な欠点を抱えているので，使っていくには少々悲しみと妥協が必要か．

UI が個人的に一番シンプルで感動したのでここで紹介しておく．

6.5.2 ツイッタークライアント

日々の生活に欠かせない（？）Twitter クライアントの紹介．

TweetDeck

言わずとしれた公式製のあいつ．ブラウザからも使えるし，Chromium エンジンのやつもあった記憶がある．私は，ブラウザから利用しているが，これよりいいものがあまり思いつかないので，王道としておいておく．

Mikutter

OSS で超有名！可愛い Ruby 製の Twitter クライアント Mikutter ちゃん．名前からオタク感漂っているかも知れないが，Twitter にいる人は大体オタクなので，むしろこれが良いのだ．（ミクちゃんかわいいすき）

特徴的なことは，OSS のクライアントなので，サクッとソースコードを見ることができるという点と，超簡単に Mikutter プラグインが作れるという点だろうか．（要 Ruby スキル）また，開発者の方々の Twitter は眺めているだけで面白いので要チェックである．

もちろんであるが，クライアントとしても優秀で，ストリーミング対応はもちろん，動作も軽いので，TweetDeck に不満があったり，軽いクライアントを使いたい各位は使うと良いのでは無いだろうか．

tw

CLI（コマンドライン）アプリケーションとなっていて，Ruby gem 経由で入る超々軽量 Twitter クライアントとなるものである．といっても，Twitter に張り付くためのものではなく，stream でずらーっと眺めたり，ちょっとしたことをターミナルからつぶやきたいというときに使う，という用途が最適なクライアントである．インストールも gem からなので簡単なので良い．

ターミナルの画面分割が可能となる「tmux」と組み合わせて使うのが個人的にオススメである．

6.5.3 エディタ

パソコンで文字を書くにはみなさんつかわれるエディタ．戦争ダメゼッタイ．

Emacs

かのリチャード・ストールマンが作り出した最高峰のエディタ Emacs．Ctrl と Meta（Alt）を華麗に使いこなしていく様子はまさに「小指の魔術師」．Ctrl-H を削除コマンドに割り当てるのがオススメだ．elisp を用いたエディタの拡張はまさに S 式の真髄．甘美なるエディタ拡張に精を出しすぎて目的を忘れる人間が後を立たない．

Vim

古来から信じられてきたエディタ VI の拡張 VIM．あらゆるモードを瞬時に切り替えて文書を編集していく様はまさに「モードの貴公子」．ノーマルモードでのオペレータとモーションの組み合わせは無限大．vimscript をマスターし暗黒美無王に謁見を願おう．

gedit

みんな大好き Linux のメモ帳的存在 gedit．コードハイライトもタブサイズ指定等もあるから，Windows のメモ帳とは比べるまでもない．汎用的で便利であるのでおすすめではある．

Atom

汎用エディタの Atom．Electron 製ということで少々の動作の重みは抱えているものの，中身はデフォルトで大体全部入っているので初心者に優しくなっている．デフォルトで自動補完や，スニペットがあるので，何も設定しなくても大体問題ないのが本当に良い点.

6.5.4 X ウィンドウマネージャ

サーバ用途ではなく，デスクトップ用途で使われる Linux PC には，「X Window System」と呼ばれるシステムを用いて，様々なデスクトップ環境を提供している．そのシステムの中で動作する X におけるデスクトップの部分である，ウィンドウマネージャを紹介していく

GNOME

あの Linus Torvalds が使っている，とされるウィンドウマネージャ（以下 WM）としても有名な GNOME．Windows の操作と比べれば多少の違いはあれど，Linux PC を使う上で，なに不自由の無い環境がそこにはある．

GNOME では WM のみのサポートではなく，GNOME Project として，WM の他にも，ファイラー，ターミナルエミュレータなどの様々なアプリの開発を行っており，全ての操作を GNOME のアプリケーションで行える．

GNOME は全体的に「無難な WM」と言える．困ったら GNOME を入れれば良いだろう．

Cinnamon

先ほどの GNOME を若干リッチにして，Windows っぽい UI にしたのが Cinnamon．Linux Mint と呼ばれる，Ubuntu を参考に作られたディストリの標準 WM の一つだ．

左下にはスタートメニューがあり，そこからアプリケーションを起動する感じは Windows を思わせる．UI がリッチになった分少々重くはなるが，使いやすさは間違いないだろう．

Awesome

タイル型と呼ばれる玄人向けの WM の中でも，一番設定が簡単（？）な WM である Awesome.

特に設定をしなくとも簡単に動き，タイル型 WM の強みである「マウス触ること無くキーボードで全ての操作ができる」という利点を得ることができる.

また，Awesome の設定は Lua で書く形となるので，時刻表示，バッテリ表示等々の拡張をする際は，自力でも，グーグル先生と一緒にでも，Lua を書く必要があるので注意が必要である.

6.5.5　ターミナルエミュレータ

みんな大好きターミナルのあれこれ．ぶっちゃけ明確な違いはわからないので，主観ではあるがざっくりレビューするという形で紹介する.

GNOME-terminal

ウィンドウマネージャである GNOME に，デフォルトで付属しているターミナルエミュレータ. ユーザビリティが高く，比較的簡単に扱うことができる.

特にエディタ等で設定項目を弄らずとも，マウスでクリックしてポチポチやっていくだけで，ざっくり設定できるという点もあるので初心者にもオススメであり，もちろんいじろうと思えばいくらでもいじり倒せるので，万人にオススメである.

xterm

X Window System 用の標準のターミナル．設定無しのデフォルトのままでは使いにくいので，ある程度の設定は必須かと思われる．しかし，標準のターミナルとあってか，先の GNOME Terminal よりも動作が軽いのが利点と言えるだろうか.

また，xterm は設定は，通常 `.Xresources` というファイルに書き込むことで行うので，お気に入りのエディタでカキカキしていくと良いだろう.

URxvt

`rxvt` というターミナルの Unicode 対応版ということで，URxvt という名を冠している．軽量ターミナルとして有名であり，設定は先に xterm と同様に `.Xresources` に書いていく感じになる.

6.5.6　ファイラー

Windows ではエクスプローラ，Mac では Finder が担っている機能が，ファイラーの成すべき機能である．Linux PC では自分好みのファイラーを選択して頂きたい.

Nautilus

GNOME 標準のファイラー.

GNOME のアプリ群と操作が似ており，また直感的に操作できる点でやはり無難なファイラーであると言える.

Nemo

Nautilus のフォークとして使われ，Linux Mint 標準のファイラーとされるのが Nemo.

Nautilus とは基本的に変わりはないが，ファイラー内で右クリックで「端末で開く」という選択肢が出るところがポイントが高い.

6.5.7 IME，変換メソッド

日本人なら日本語入力が必須なので紹介する.

IME は「ibus」と「fcitx」などがあり，また，変換メソッドは「Anthy」と「Mozc（Google 日本語入力の派生)」などがある.

ぶっちゃけどれが良いのか，というのはわからないが，個人的な主観（つまりは経験）で言うと，「fcitx」と「mozc」を組み合わせたものが一番かなぁと行った感じだ.

他にも「uim」や「scim」等の他の IME もあったりするので，こればっかりは読者に試して頂きたい.

6.6 For Linux Users

ここからは Linux を日々利用している読者らに，中・上級者向けのディストリの紹介をしていく.

6.6.1 Arch Linux

今回紹介するのは Arch Linux というディストリだ．特徴的であるのは，他のディストリとはインストール方式が異なり，「GUI でのインストールではなく CUI でのインストールとなる点」と，それにより「ミニマルなパッケージ構成が可能になるという点」という点だ.

Arch には「The Arch Way」という哲学があり，そこには以下 5 つの言葉がある.

1. シンプルであること.
2. 便利であることよりも正確なコードであること.
3. ユーザ中心であること.
4. オープンであること.

5．自由であること．

この 5 つの言葉は「KISS（Keep It Simple，Stupid；シンプルにしておけよ，バカチンが）」という言葉に集約されており，Arch のユーザは無駄の無い最小の構成で Linux を扱うことができるのだ．

ここにインストールを手順を書いても良いが，公式のドキュメント*[2]があまりにも優秀なため割愛する．

また, Arch Linux の Wiki である ArchWiki では, Linux をハックするための大量の知見があるので，チェックしてみると良いだろう．（Linux のソフトウェアで詰まったときは，大抵 ArchWiki にたどり着くほどである）

6.6.2 Gentoo Linux

Linux カーネルに興味を持った人は，このディストリで存分にカーネルコンパイルをすると良いだろう．Gentoo の基本的なイメージとしては挙げられるのは，良くも悪くも「自由」という点だ．

それは（このディストリの最大の特徴として挙げられるが）「使いたいソフトウェアがあった際は，そのソフトウェアのソースコードを取ってきて，基本的にコンパイルしてから，インストールをする」と言う点である．

他の OS であればパッケージマネージャが,「実行形式のプログラムをインストールする」という形だが，Gentoo では，Portage というパッケージ管理システムを採用しており，「ソフトのソースを取ってきてコンパイルしてインストールする」という形なのだ．

これにより Gentoo は多様な環境で動作することを保証している．プログラムが動かないなら，コンパイルすればイイじゃない！という感じだろうか．

また，ソースからプログラムをインストールすることで，そのハードに最適なバイナリコードを生成できるので，確実に動作早いだろう．

しかし，これはハードにもよるが，コンパイルというのはものによっては大量の時間がかかる．CLI アプリケーションだけ入れるのなら良いが，X や Office 等の GUI アプリケーションを利用をしようとしたら，場合によっては，１日がコンパイルだけで終わってしまう，ということも無い話ではない．良くも悪くも,「コンパイル」が Gentoo には必要なのだ．

（もちろんデスクトップ用途の Gentoo を構築するのも良いものだ．個人の感想であるが，Gentoo の動作は全体的に早い．X の起動も早いし，動作も軽快だ）

どのような Gentoo を作るのも，ユーザに委ねれられているが，サーバ用途や，勉強用途など，明確な目的をもって Gentoo を使うと良いのではないのだろうか．

*[2] Beginners’ guide - ArchWiki https://wiki.archlinux.org/index.php/Beginners’_guide

6.7 おわりに

これまで Linux とデスクトップ用途での Linux PC について記述したが，これらに関しては Linux についてのほんの一部でしかない．興味をもった読者にはぜひその魅力を身をもって味わって，その深みにはまって頂きたいと思うところである．

拙い文書，また無作法な部分があったかもしれないが，これまで読んで頂き感謝感激雨霰である．

第 7 章

大著謎ジャンル本のススメ

こはる（@cohalz）

今回は技術的な話ではなくただの本の紹介です．特に若い人向けへの記事です．

7.1 大著謎ジャンル本とは

簡単にいえば，様々なジャンルを渡り歩いているジャンル分類が難しい本を指します．様々なジャンルの解説及び関連付けの説明のために普通の本よりも数倍分厚いこともあります．

7.1.1 例

ロジャー・ペンローズの『皇帝の新しい心』やダグラス・R．ホフスタッターの『ゲーデル，エッシャー，バッハ』と『メタマジック・ゲーム』，吉田武の『虚数の情緒』他などが例として挙げられます．これらはどれも 800 ページから 1000 ページほどもある大著です．

7.1.2 内容

『ゲーデル，エッシャー，バッハ』の商品紹介は以下のようになっています．

「GEB の内容を一言で説明するのはむずかしい。中心となっているテーマは「自己言及」だが、これが数学におけるゲーデルの不完全性定理、計算機科学におけるチューリングの定理、そして人工知能の研究と結びつけられ、渾然一体となっている。エッシャーのだまし絵やバッハのフーガはこれらをつなぐメタファーとして機能している。ホフスタッター自身、本書の中で「これは自分にとっての信仰告白である」といっているように、おそらくこの本は特定の概念を読者に説明するといった目的のものではない。むしろ人間は自分自身に興味をもつことを永久にやめられないであろうという、ホフスタッターの信念をひたすら熱狂的に記述したものとなっている。GEB では自己言及を人間の知性のもっとも高度な形態として位置づけており、それゆえに人工知能の研究を礼讃している。また随所に自己言及のパラドックスや言葉遊び、数学パズル、そして禅などがちりばめられており、この本自体も自己言及をおこなっている。このようなスタイルは当時の計算機にかかわる研究者やプログラマーから生まれたハッカー文化に類似している。」

なるほど．虚数の情緒の商品紹介の一部は以下のようになっています．

題名からすると、中学生以上に向けた数学解説書のように思われるが、実際はそんな生半可な本ではない。本書は「虚数」の概念を軸として人類文化全体を鳥瞰（ちょうかん）した、実に 1000 ページを超える大著である。いままで西洋人によって書かれた類書はいくつかあり、それらに触れるたびに西洋文化の重厚さに圧倒される思いをしてきた。本書はその西洋文化の華々しい成果を扱っているわけだが、根底に流れる思想からは強く日本文化の香りがする。その理由は著者が対象について深く理解し血肉とし、それを改めて自らの言葉で述べているからである。
著者は文中で「新しい文化を取り入れるという事は、決して自らの文化への" 接ぎ木" をすることではなく、それを深く理解し自らの血肉とすることである」と繰り返し強調しているが、本書はその実践の結果である。また、副題からもわかるとおり中高生の読者を意識しており、冒頭からかなりのページを割いて「学ぶとは、理解するとはどういう事か？」について説いている。

説いている．

7.2 大著謎ジャンル本の面白さとは

7.2.1 面白さ

『ゲーデル，エッシャー，バッハ』では特に異なった分野の関連付けが多く，こんな繋がり・こんな見方があったのかと興味深くなるような内容でいっぱいです．一生を丸々使って，大掛かりな言

葉遊びや仕掛けが施されていることもありしっかりと考察しながら読みたい人にもオススメです．
『虚数の情緒』は最初に現代社会の問題について書かれ，次に歴史をおさらいして，その次に自然数から数学を解説し始めます．寄り道を経ながら虚数の解説まで終わった後は，古典物理学に入り最後には量子力学の解説にまで到達します．
どの本も分厚く図も多いので，適当にページをパラパラとめくる用途にも向いています．

7.2.2　難しくないの？

そもそも分厚くなっているのは多くの解説によるものなので，薄めの本よりはだいぶわかりやすく丁寧に書かれていることが多いです．どれくらい丁寧かというと，万能チューリングマシンの説明のためにに 2 ページほど二進数で埋め尽くされた項があるくらい（万能チューリングマシンそのものの二進数が書かれている），紙面の都合を気にしない丁寧さがあります．一部の本は中学生から読めると謳っているものもあるくらいなので，それを考えると意欲的な大学生なら十分読めると思います．

7.3　本の探し方

大著謎ジャンルに限らず，本の探し方の一つとして便利なのが，読んだ本の参考文献を見るというのがあります．大著謎ジャンル本では特に，多彩なジャンルと多くの解説で様々な本で紹介されていることも多いので結構見つかると思います．個人的には結城浩さんの本の参考文献，読書案内から見つけたことが多いです．
それに加え，最新の本や埋もれた本の見つけ方として，Twitter で流れてきた本のタイトルをすぐ記録しておくという方法も取っています．記録方法として詳細情報や値段がすく確認でき，また購入しようと思った時にも楽な Amazon ほしい物リストを利用しています．

7.4　最後に

技術書だけではなく，こういった変わった本も読んでみてはという話でした．特に余裕のある長期休暇でこの話を思い出して読んで貰えたら嬉しいです．

あとがき

PMOB 所属の方々に一言お願いしてみました．

Zodiac

数学が理解できないと死に至る病だって，キルケゴールさんに診断されたよ．

うにの抜け殻

Unity ゎ、ぅにてぃし゛ ゃぁなぃんた゛ ょ｡｡｡

falco

19 歳になりました

matsub

T-ウィルスに抗体があります。

ころも

ｶﾁｭｰｼｬ･･･

shamison

D アニメストアはいいぞぉ。

ichi

小指相撲挑戦者募集

まのめっち

新入生向けの記事書こうと思ったけど win10 の bash がもしかしたら最良かもしれないので今回は記事見送りました (大嘘)。Twitter もやってますが名前はコロコロ変わってますのでアイコンと ID と雰囲気で判断してね (@Mano_tetsu)

cohalz

最近食べに行っても休みなこと多いけど不景気ってやつですか？

著者： PMOB

表紙絵： めの, shamison

発行： 2016 年 4 月 1 日

印刷： 東京電機大学デザイン工房

Twitter： @PMOB_

Home Page： http://pmob.github.io

P