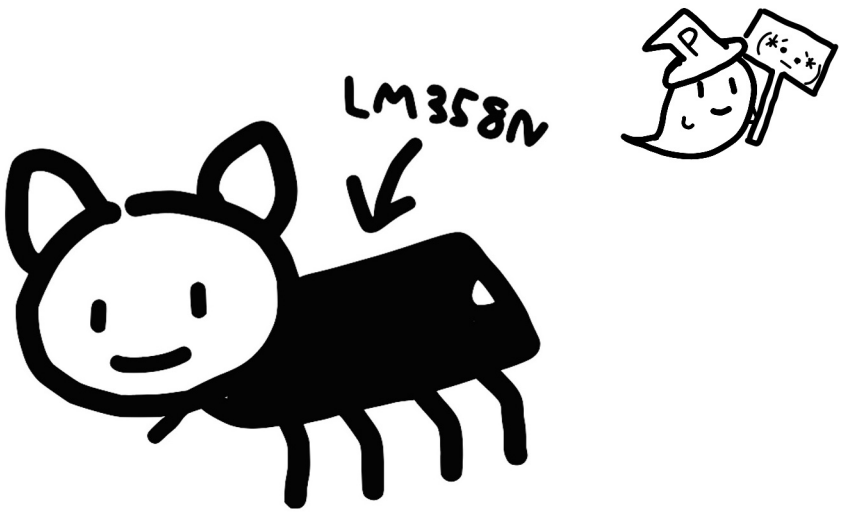
# PMOB

の読むやつ #1



# はじめに

月日が経つのは早いですね．おはようございます．cohalz です．去年の 10 月 16 日にサークルを立ち上げると決心してからもう一年が過ぎました．初期メンバーは自分を含め 3 人でしたが，現在 Slack のメンバーは 21 人にもなり非常に賑やかになりました．

今回はついにそんなメンバーで協力して，技術部誌を出す運びになりました．計 8 人で 9 つの記事があり，ページ数も 60 ページを超える立派なものになりました．

鳩祭で部誌を出すにあたって多くの人に感謝します．部誌作成作業ををほとんど一人で進めてくれた shamison，サーバやインフラ周りでお世話になった sanex_now，賑やかなアイコンを提供し，部誌の内容を 2 本も提供してくれための，pmob.github.io や表紙のデザインを考え，普段の活動では技術顧問のように動いてくれた matsub.rk，数学などあまり知識のある人がいなかった分野の話題を提供してもらうなど活動の幅を広げてくれた zodiac，ここでは紹介しきれないその他のメンバー，及び協力してくれたサークル外部の方にも感謝します．

そんな PMOB ですが，普段は週に 2 から 3 回の頻度で図書館三階のグループスタディおよび HATO CAFE にて活動を行っていました．グループスタディではホワイトボードが使えるため，議論しやすい環境でしたし，HATO CAFE ではコーヒーやアイスが楽しめました．また HATO CAFE では不特定に人が来るため，オフ会会場になって交友が広がるということも多々ありました．助教の先生も来ることがあり，面白い話を聞ける機会も多くありました．

PMOB は鳩祭が終わったあとも，ハンズオンやハッカソン，その他実験などを予定しています．興味を持たれた方は是非 Twitter: @PMOB_や pmob.slack.com を覗いてみてもらえると嬉しいです．人生何が起こるかわかりませんね．これからもよろしくお願いします．

2015 年 11 月 1 日 cohalz

# 目次

# 第 1 章

# main = putStrLn "Haskell で hshs!w"

Zodiac Caulfield(@zodi_G12)

## 1.1 序文

世の中何があるか分からないものである。何が起こるか分からないものの一つとして「出逢い」がある。出逢いは必然か、はたまた偶然かという議論はここではしない。ともあれ、ある出会いによって人生が大きく変わることがある。私の経験上でもそういったことが何度かあった。特に、人生の大きな転換期とも呼べる期間があった。

思えばそれは去年の春のことである。秋葉原で女神に出会ったことで私は「真理」に触れることができたのがきっかけである。真理、それは「絶対的な美」である。このことに気が付いた。女神の名前は「矢澤にこ」。自明に宇宙ナンバーワンアイドルである。たとえ、量子力学上の解釈の 1 つである多世界解釈を導入したとしても宇宙ナンバーワンアイドルである。

私は、矢澤にこに出会って以来「真理」を追い求めることに夢中になってしまった。矢澤にこは「ラブライブ！」というアニメに出てくる女神なので読者諸君は何度もこのアニメを観なければならない。ラブライブ！を観ない。こんなに嘆かわしい事があろうか。技術者として絶対に観るべきアニメであるというのに。こんな事件を防ぐ為にも「ラブライブ！を見ることを国民の義務として課す法律」を定めることが急務であろう。教育カリキュラムの中に組み込むべきである。

次にテーマについてだが、テーマは「**Haskell** に関して」である。Haskell の凄みを魅力を読者諸君に分かって戴きたい一心で書くつもりだ。だがしかし、私は Haskell 初心者であり、プログラミング自体は今年から学び始めたので、そういった意味ではまだまだ青いというのが現状である。この点に関しては読者諸君に十分留意して戴きたい。温かい目で見守って戴ければ有難い。また、ある程度 Haskell に関して知識がある方は、本著は読む意味がないので控えてもらいたい。いや、読まないで。やめて。虐めないで。虐めかっこ悪い。実に良くない。

本著の形式としては「にこちゃん」と「まきちゃん」の二人の美少女による対話形式で記述していこうと考えている。にこちゃんは高校三年生の黒髪ツインテの少女で、まきちゃんは高校一年生

の赤髪癖毛の少女であり、両者優劣つけ難い絶世の美女であり、その美貌はこの世には存在しない美しさである。それもそのはず、両者は女神であるのだから。また、女神の二人は私の嫁であるので読者諸君が気安く呼び捨てで呼ぶことは許されない。

## 1.2 関数

まきちゃん「にこちゃん。関数って何？」

にこちゃん「関数は関数じゃない！当たり前でしょ！」

まきちゃん「ナニソレ！イミワカンナイ！」

関数とは函数のことである。函数とは関数のことである (循環論法 (違う))。Haskell における関数とは 1 つ以上の引数を取り、1 つの結果を返す変換器である。

## 1.3 関数型言語 Haskell とは

まきちゃん「にこちゃん。Haskell って何？」

にこちゃん「Haskell は関数型プログラミング言語のことよ。すごくかっこいい言語なんだから！」

まきちゃん「かっこいいってどういう風にかっこいいのよ・・・」

にこちゃん「Haskell は私の次にかっこいいわよ！」

まきちゃん「ナニソレ。キモチワルイ。」

読者諸君は**「Haskell」**という関数型プログラミング言語を一度は聞いたことがある筈だ。Haskell は関数型プログラミング言語の一種である。関数型プログラミングとは「計算とは関数の引数に適用すること」だというプログラミング手法である。の関数型言語とは関数型の手法を提供し奨励している言語である。

個人的には、Haskell は知的な感じがしてイケメンで、小難しい一面もツンデレな感じがして好感を持てる言語である。何よりも記法が素敵と言わざるを得ない。まきちゃんとにこちゃんに気持ち悪いと言われたい人生だった。

## 1.4 Haskell の特徴

にこちゃん「Haskell の主な特徴として以下の事が挙げられるわ。」

1. 簡潔なプログラム
2. 強力な型システム 多相型と多重定義型を許す
3. リスト内包表記
4. 再帰関数 パターンマッチとガードは便利
5. 高階関数 ドメイン特化言語の定義

6. モナド 纏まった数学の概念
7. 遅延評価
8. プログラムの論証

まきちゃん「へぇ〜」
にこちゃん「詳細はググって」
まきちゃん「ヴェェ！？」←言わせたかっただけ

## 1.5 Haskell 関連歴史

まきちゃん「Haskell が出来た経緯を知りたいわ。にこちゃん。お願い。」
にこちゃん「しょうがないわねぇ〜」
Haskell を語る上ではやはり Haskell に関連の歴史を振り返ることが必須であろう。Haskell に関連の歴史を知ることが、Haskell とは何かという理解に繋がると考えているためである。Haskell 関連の歴史を以下にまとめる。

- 1930 年代、Alonzo Church は、λ計算を考え出した。
- 1950 年代、John McCarthy が、最初の関数型言語である Lisp (LISt Processor) を開発した。Lisp はλ計算に影響を受けているが、変数への代入を採用していた。
- 1960 年代、Peter Landin は、ISWIM (If you See What I Mean) を開発した。ISWIM は、λ計算に基づいた最初の純粋関数型言語であり、変数への代入はない。
- 1970 年代、John Backus は、FP (Functional Programming) を開発した。FP は、高階関数とプログラミングの論証という特徴を持つ。
- 同じくして 1970 年代、Robin Milner たちは、ML (Meta-Language) を開発した。ML は、最初の近代的な関数型言語であり、多相型と型推論を導入した。
- 1970 年代と 1980 年代に、David Turner はいくつかの遅延関数型言語を開発した。その頂点は Miranda とされている。
- 1987 年、研究者で構成された国際委員会が、(論理学者の Haskell Curry に因んだ) Haskell を遅延関数型言語の標準とすべく開発を進めた。
- 2003 年、この委員会は、設計者の 15 年に及ぶ作業の成果として、Haskell Report を公開した。この Haskell Report は、長年待ち望まれた Haskell の安定板を定義している。

にこちゃん「上記研究者のうち、McCarthy、Backus、そして、Milner が、計算機科学のノーベル賞とも言える ACM チューリング章を授与されたことは特に凄い事ね。」
にこちゃんはやっぱり何でも知ってるんだね。流石私の嫁だね。prpr

## 1.6 Haskell の妙味

まきちゃん「はぁ・・・」

にこちゃん「どうしたのよ、まき。」

まきちゃん「うん。トマトが食べたくって・・・」

にこちゃん「」

Haskell の妙味は杏仁豆腐の妙味と似ている。非常に美味という事である。私は、にこちゃんが食べたいと白亜紀から言い続けている。さあ、Haskell の妙味を説明していこう。

## 1.7 Haskell のクイックソート

まきちゃん「[252, 25, 722, ・・・]、はぁめんどくさい・・・」

にこちゃん「どうしたの？まき。」

まきちゃん「あのね、リストを昇順に並び替えたいのよ。C 言語で書こうと思ったんだけどめんどくさくって・・・」

にこちゃん「Haskell ならすごい簡単に書けるわよ？ (カキカキ・・・)」

まきちゃん「ヴェェェ！？凄いわにこちゃん！やっぱりにこちゃんは天才ね！！！」

にこちゃん「そ、そう？照れるわね///」

私「ぶ、ぶひ〜〜〜(‘ω’)」

にこちゃん・まきちゃん「」

Haskell の強大な力を披露する方法として「**Quick Sort (クイックソート) 」**というものがある。この関数は、任意の数値型のリストに対して、要素を並び替えたリストを生成するというものである (今回の場合は昇順に並び替える)。Haskell でクイックソートは以下のように書ける。

```
qsort []       = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
                   where
                     smaller = [a | a <- xs, a <= x]
                     larger = [b | b <- xs, b > x]
```

++ は 2 つのリストを連結する二項演算子である。[] はガウス記号ではなく、リストを表す。[] は空のリストで、qsort [] = [] というのは、空のリストを与えられたときに空のリストを返すというただそれだけである。where は局所定義のキーワードである。また where で smaller, larger を定義している。smaller は xs から x 以下である全ての要素 a を取り出して作ったリストである。larger は xs から x より大きい全ての要素 b を取り出して作ったリストである。上記のクイックソートの実装は、明瞭で簡潔であるという Haskell の力を示す素晴らしい例である。はぁ、にこ

ちゃんとまきちゃんをクイックソートしたい・・・

## 1.8 参考文献

主に参考にした著書は「オーム社のプログラミング Haskell (Graham Hutton 著、山本和彦訳)」と「圏論の基礎 (S. マックレーン著)」である。

## 1.9 謝辞

このような機会に皆さんに読んでいただいたのと、書かせて戴けたのは至上の喜びです。PMOB の先輩方、それと私に親しくしてくださっている皆様のおかげです。私の TwitterID は **zodi_G12** です。Haskell に詳しいにこちゃん bot は **zodi_has** です。気持ち悪く書いてしまいすみませんでした。みなさんににこにーの加護のあらんことを。

# 第 2 章

# はじめよう自宅サーバ

@sanex_now

あると便利な自宅サーバみんなもやってみよう

## 2.1　はじめに

こんにちは　いっちーと申します。現在東京電機大学理工学部情報システムデザイン学系所属の 3 年生になります。趣味は自作 PC と Linux サーバいじりです。

## 2.2　自宅サーバとは

### 2.2.1　そもそもサーバとは

サーバコンピュータ (以下サーバ) というものをご存知でしょうか。皆さんもサーバの恩恵を知らず知らずのうちに利用しています。Web ページの閲覧は Web サーバに対してブラウザが要求を送り、それにサーバが応答する形でページの配信を行います。スマートフォンのアプリのほとんどもサーバとの通信を行いサービスを提供しております。

### 2.2.2　自宅にサーバを置くメリット

最近は PaaS,IaaS などのクラウドコンピューティングサービスを低価格で利用できるようになってきており、自宅にサーバを置くメリットは少なくなってきております。しかし、自宅サーバはクラウドを上回る魅力があります。実際に動いているサーバを手元で管理する魅力はとても大きいです。

## 2.3　自宅サーバを構築してみよう

このセクションでは自宅サーバ環境を構築するために必要なことについて解説します。具体的な機器の操作方法などは説明いたしませんのでサーバ構築のフローを理解するのにご活用ください。

### 2.3.1 サーバ機器を用意しよう

自宅に置くサーバを選定しましょう。用途にもよりますがサーバは基本的に 24 時間稼働させる機器になります。それなりの耐久性、消費電力などを考慮する必要があるでしょう。中古のノート PC を自宅サーバとして使う方が多いようです。しかし、ここで注意が必要です。ノート PC は消費電力も少なく、モニタ、キーボードも揃っているので良い選択ではあるのですが、長時間の稼働を前提とした設計になっていないことがほとんどです。熱設計においてもデスクトップ PC より筐体内が高い温度になるためサーバとしてのノート PC の使用は控えた方が良いでしょう。

ではどのような PC が理想的か、

理想を言ってしまえばタワーサーバと呼ばれるメーカがサーバ用途を前提として販売している PC になりますが、高耐久の部品を使っているため価格が高く手が出しにくいものです。はじめは通常のデスクトップ PC などが望ましいでしょう。

### 2.3.2 サーバ機器を設定しよう

#### OS の選定

企業のサーバなどでも多く利用されている Linux を利用するのが望ましいでしょう。Linux には多数のディストリビューションがあり、それぞれ特徴もあります。以下に一般的にサーバとして利用される Linux ディストリビューションは以下のようなものがあります。

| ディストリビューション |
|---|
| CentOS |
| Ubuntu Server |
| Debian |
| Fedora |

企業においては RedHatEnterprizeLinux(以下 RHEL) が絶大な人気を誇っているらしく、RHEL ベースの CentOS が日本語ドキュメントも多くお勧めです。

#### OS のセットアップ

OS セットアップの具体的な方法は各 OS ごとに異なりますのでここでは割愛いたします。このセクションでは OS セットアップの基本的な流れを説明します。OS 付属のインストーラでは順番が前後している可能性もあります。

#### LAMP

Linux(OS), Apache(Web サーバ), Mysql(データベース), PHP の頭文字からとった語基本的な Web サーバに必要とされるミドルウェアのことである。

#### セキュリティ設定

サーバには、基本的に SSH を用いて操作を行う。ここでは SSH 接続を安全に利用するための注意事項を解説する。

1. SSH ログイン UNIX 互換環境の場合は ssh コマンド、Windows の場合は TeraTerm などのアプリケーションを用いて接続を行う。
2. SSH 鍵生成 ssh-keygen コマンドにより SSH 接続に用いる公開鍵と秘密鍵のペアを生成する。
3. SSH 鍵登録 接続先サーバに公開鍵を設定する。
4. 公開鍵方式ログイン ログイン時に秘密鍵を指定してログイン
5. SSH 使用 port 変更設定 SSH はデフォルトで port22/tcp を用いて通信を行う。経路は暗号化されているが、この 22 番 port に対しては乗っ取りを行おうとする悪意のある攻撃が頻繁に行なわれている。そのため、攻撃を未然に防ぐためにも使用 port の変更を行うことをお勧めする。

### 2.3.3　ネットワーク機器設定

#### NAT 静的 IP マスカレード

サーバをルータに接続しただけでは外部からの通信に応答することができません。DHCP によりローカル IP アドレスが割り振られているためです。外部からの接続に応答するためには NAT の設定を行う必要があります。

- NAT とは Network Address Translation IPv4 アドレスが枯渇状態にあるため、LAN 内のホストにはプライベート IP アドレスを割り振る。
- 動的 NAT 外部と通信する際はルータの持つグローバル IP を一時的に借りて通信を行う。これが動的 NAT である。外部からは LAN にどのようなホストが存在しているかわからないため、外部からの接続に LAN 内のホストが応答することはできない。
- 静的 NAT 外部と通信する際、一つのグローバル IP に静的に変換を行い通信を行う。これが静的 NAT である。グローバル IP に対してローカルの IP が一対一で変換されるので、外部から LAN 内のホストに対して通信を行うことができる。
- NAPT(PAT, IP マスカレード) 現在最も一般てきなのはこれです。IP アドレスの変換の際

port 番号をも変換することができる。

これらの技術を抑えた上で作業すればスムーズに設定ができるでしょう。

## 2.4 終わりに

この記事では自宅でサーバを立てるには何をすればいいか、基本の基本をさらっと紹介しました。少しでもサーバ構築に興味を持ってくだされば嬉しいです。

# 第 3 章

# Brainf*ck を嗜む

うにの抜け殻

難解プログラミング言語として有名な Brainf*ck でちょっと遊ぶだけ

## 3.1　Brainf*ck ってそもそも何よ

> “開発者 Urban Muller がコンパイラがなるべく小さくなる言語として考案した。実際、Muller が開発したコンパイラのサイズはわずか 123 バイト、インタプリタは 98 バイトであった。” (wikipedia から引用)

この言語での命令は 8 つのみである。

1. “>” ポインタ を 1 加算する。
2. “<” ポインタ を 1 減算する。
3. “+” ポインタが指す 値 を 1 加算する。
4. “-” ポインタが指す 値 を 1 減算する。
5. “.” ポインタが指す値を出力する。
6. “,” 入力から 1 バイト読み込み、ポインタが指す先に代入する。
7. “[” ポインタが指す値が 0 の時、対応する “]” にジャンプする。
8. “]” ポインタが指す値が 0 でない時、対応する “[” にジャンプする。

これらを組み合わせるだけでプログラムが書ける！すごい (怖い)！

## 3.2　インタプリタを用意するぞい

とりあえず自分の使用しているインタプリタを書いておく・・・と言いたかったが、どこでダウンロードしたものかを失念してしまった。(似たような名前のものが多いのでもう見つかりそうにない)
とりあえずは「Brainf*ck interpreter」などで検索していただければもっと使いやすいエディタが

見つかるのでそちらを使っていただきたい。

## 3.3 とりあえず HelloWorld を出力したい

準備ができたところで、とりあえず手始めに HelloWorld を出力するためのプログラムが以下の通りである。

```
++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++.
+++++++++++++++++++++++++++++.
+++++++
..
+++.
------------------------.
++++++++++++++++++++++++.
+++.
------.
--------.
```

いかがだろうか。HelloWorld の声 (?) が聞こえただろうか。
解説すると、+ による値のインクリメントでポインタが指している 1 番目の場所を 65、つまり ASCII Code の'A' にし、出力。その後また値のインクリメントで 1 番目の場所を 101、つまり'e' にして出力。これを HelloWorld の文字列になるように何度も行っているのがこのコードである。
もちろんこんな長ったらしいコードを書く必要など無く、もっと短く書くことができる。

```
+++++++++[>++++++++>++++++++++>++++++++++++<<<-]>.H
>++.e
>.l
.l
+++.o
<<<+++++[>+++<-]>.W
>>.o
+++.r
------.l
<-.d
```

このように短いコードになる。なお、ソース中のアルファベットは他言語のコメントアウトと同じように読み飛ばされる。

### 3.3.1　解説

ポインタが最初に指している場所を 1 番目とする。1 番目を 9 までインクリメントし、ループに入り、2,3,4 番目をそれぞれ 8,11,12 回インクリメントする。これによって 9 * 8, 9 * 11, 9* 12, つまり 2, 3, 4 番目にはそれぞれ 72(H), 99(c), 108(l) が入る。後は HelloWorld にそれぞれ近い数値を加減して出力していく。
設計する時は以下のように表示する文字の ASCII code の数値と、使うポインタの指す場所を書き出しておくとコードが書きやすい。

| 文字 | H | e | l | l | o | W | o | r | l | d |
|---|---|---|---|---|---|---|---|---|---|---|
| 数値 | 72 | 101 | 108 | 108 | 111 | 87 | 11 | 114 | 108 | 100 |

場所と数値
[1] 9 * 8 -> 72 + 15 -> 87
[2] 9 * 11 -> 99 + 2 -> 101 - 1 -> 100
[3] 9 * 12 -> 108 + 3 -> 111 + 3 -> 114 - 6 -> 108

## 3.4　とりあえず一つプログラムを組みたい

これだけでは文字を表示することしかできないように見えるので、とりあえず何かプログラムを書きたいと思う。
今回はこのような仕様のプログラムを書く。
1. '0' から'9' の入力を 1 文字受け取る。 1. 入力された数字からの連番を出力する。 1. 出力する数字が 10 以上の時は 0 から出力する。

- sample input
  8
- sample output
  890

### 3.4.1 ソースコード

```
ぶ れ い ん ☆ ふ ぁ っ く
,.>+++++++<[>>+>+<<<-]1 copy to 2 3
>[>---------<-]if input == 8 then 890
>[-if input == 9 then 901
[>+.+.>]other output
>[----------.+.>]]901 output
>[+.----------.>]890 output
脳 汁 こ こ ま で
```

### 3.4.2 解説

最初に入力をとり、そのまま出力する。
2 番目をカウンタとして利用して、入力を 3,4 番目にコピーする。
2 番目をまたカウンタとして利用して、3 番目を 56 回 (‘8’) デクリメントする。
入力が 8 であった場合は 890 の出力まで進み、出力して終了する。
そうでない場合は、さらに 1 回デクリメントして、入力が 57(‘9’) であるかを判別する。
入力が 9 であった場合は 901 の出力まで進み、出力して終了する。
それ以外は other の部分で出力して終了している。

これについては設計を書くとややこしくなりがちであるが、上述したとおり、使う場所と出力する文字の数値をしっかりと書いておくことは重要である。
制御文、つまり if 文のように分岐するには、入力が分岐条件と同値であるかを判別するのが手っ取り早い。ただし、判別する時に入力した数値が消失するので、あらかじめ 2 箇所以上にコピーしておくことが必要である。
また、条件として「n 以上」などの条件を用いたい場合は、n までを判別しながら n 以下で一致した時にループを抜ける設計にし、n 以上で一致するまで無限ループさせつづける必要がある。(非常に手間がかかる)
2 桁以上の数値の出力についてはどこかしらの場所を、出力する文字ではなく、数値の保管用の変数 として使うと楽に実装できる (簡単だとは言ってない)
この際 10 進法での出力をさせる処理を作る必要があるだろう。(出力する桁数は限らないと無理かもしれない)
2 桁以上の和算などは除算のアルゴリズムを利用すれば書くことができるので、ぜひ書いてみてほしい (執筆時間の不足を押し付けるスタイル)

## 3.5　終わりに

Brainf*ck は非常に手間のかかる言語であるが、理解し、簡単なプログラムでも書けた時はすごく気持ちがいい。
また、その仕様上、ポインタの学習にも非常に最適な言語である。
皆さんもこのすばらしい Brainf*ck で遊んでみてはいかがだろうか。
以上、Brain ファッ！？ k の記事でした。(いんむようそはありません)

# 第 4 章

# Git のなんか

@matsubrk

明日使える Git のなんやら。驚くべきことにコンフリクト解消については一切触れていません。あと Git の具体的な操作については触れませんでした。ごめんちょ。

## 4.1 Git の基本

Git の細かなコマンドとその動きを知ることも Git を上手く扱うのに必要なことですが、その概要を把握しておくことも重要な要素です。基礎知識は Git を思うように動かせないときに心強い味方になってくれます。

慣れないうちは、「んん、今これはどういう状態で、私はこれをどうしたいんだ？」という風に考え込んでしまうことも多いと思いますが、この章ではそんなときに役立つ基礎知識を掲載します。

### 4.1.1 バージョン管理システム

「バージョン管理システム (Version Control System: VCS) って何だ」と思って調べてみると、だいたい「ファイルの変更履歴を管理する為のシステム」とか出てきます。VCS を知っている人ならこう言われて「まあそうね」となると思いますが、知らない人は「で？」となると思います。直感的な理解を得るために、ちょっと触ってみましょう。

```
$ git init
$ echo 'Hello, World!' > hello.txt
$ git add hello.txt
$ git commit -m 'first commit'
[master (root-commit) 332874f] first commit
```

標準出力は一部省略しています。VCS に関するコマンドの意味は今は置いておいて、ここでは`hello.txt`というファイルを作っていつもの`"Hello, World!"`の文を書き込んでいます。

```
$ echo 'Hello, Japan!' > hello.txt
$ cat hello.txt
Hello, Japan!
$ git add hello.txt
$ git commit -m 'Modify location'
[master 1ebf9ac] Modify location
```

またまたとりあえず VCS のコマンドは置いておいて、hello.txtの中身を"Hello, Japan!"に書き換えました。中身は確かに変更されています。ここで VCS を使ってちょいちょいってやります。

```
$ git checkout 332874f -- hello.txt
$ cat hello.txt
Hello, World!
```

書き換える前に戻りました。 では元に戻します。

```
$ git checkout 1ebf9ac -- hello.txt
$ cat hello.txt
Hello, Japan!
```

書き換えたあとに戻りました。 直感的にはこれがバージョン管理です。ファイルの状態を好きな時点のものにできる。そして、この「時点」のことを「バージョン」と呼びます（正確には「リビジョン」で、「バージョン」というと「バージョン番号が振られたもの」というイメージがありますが、最近はごっちゃになってるらしいです)。

VCS はあるバージョンと別のバージョンの差分を取ったりスナップショットを撮ったりすることでバージョン管理を実現しています。ちなみに今回紹介する Git はスナップショットでバージョン管理を行っています。

### 4.1.2　VCS のキーワード

まず、VCS 一般のキーワード「リポジトリ」「コミット」「ブランチ」「マージ」「チェックアウト」について簡単に説明しておきます。VCS によって呼び方が違うものがありますが、Git 準拠とします。最初に聞いておいて欲しいけど今理解しなくてもいい系の話です。気楽に読み流して行きましょう。

- リポジトリ

VCS がバージョンを管理するファイルの状態を保持しておくための場所です。ユーザは、ファイルを変更してリポジトリに変更を書き込むことでバージョン管理を行います。

- コミット

リポジトリへファイルの変更を書き込むことをコミットといいます。コミットはリポジトリに記録されたファイルの変更を表す最小の単位です。

- ブランチ

プロジェクトの分岐（ブランチ）です。あるバージョンから分岐させて別のバージョンを作成することを「ブランチを切る」などといいます。例えば、安定版に変更を加えるとき、「安定版」を「安定版と開発版」にブランチ（分岐）することで安定版には手を加えずに作成したブランチで開発をすすめることができます。これがブランチの目的です。

- マージ

あるブランチから別のブランチの変更内容を取り込むことをマージといいます。ブランチで紹介した例でいくと、ブランチでの開発が完了し安定版に変更を適用したいとき、作成したブランチを安定版にマージすることで安定版を変更が加えられた最新版にすることができます。

- チェックアウト

リポジトリからデータを取り出すことをチェックアウトといいます。チェックアウトの対象はブランチであったりコミットであったりします。この操作によってファイルを指定したバージョンのものにすることができます。

### 4.1.3 Git のもつ概念

Git を理解するのに重要な概念がいくつかあります。まず、Git を使った作業には 3 つの領域があります。一つはユーザーが実際に見てファイルを弄る「作業ディレクトリ」、もう一つはリポジトリの情報を管理する「Git リポジトリ」、そして最後に特徴的な「ステージングエリア」です。「Git リポジトリ」については特別なことはないので「変更が記録されたリポジトリの領域」として覚えていただければ大丈夫です。「作業ディレクトリ」ももちろん簡単な話です。問題は「ステージングエリア」でして、この「ステージング」の概念を理解していただきたいと思います。

まず、多くの VCS におけるコミットについて理解していただきたいと思います。VCS においてコミットは作業ディレクトリに施された「変更」であり、記録されるコミットは diff で生成する差分のようなイメージです。この差分っぽいものをすごい量保存するのが「リポジトリ」です。そ

してリポジトリが監視するのは作業ディレクトリの「変更」です。ファイルの内容の変更だけでなく、ファイルの追加・削除も作業ディレクトリの「変更」と考えることが出来ます。そして、これらの変更をリポジトリに登録するのが「コミットする」ということなのですが、Git では作業ディレクトリの変更全てをコミットする必要はありません。記録したい変更だけを選択して記録することが出来ます。これを可能にするのが「ステージング」です。

Git でコミットを行うにはファイルを書き込んでコミットするだけでなく、`git-add`コマンドによって「ステージング」を行う必要があります。ステージングを行うことで、その時点の変更を「ステージングエリア」に書き込むことが出来ます。ステージングエリアは次のコミットの変更を記録するための一時的な領域で、Git でのコミットは「ステージングエリアに書かれた変更を Git リポジトリに書き込むこと」となります。

一気に変更をコミットせずにステージングエリアを用意するメリットは十分にあります。この機能のおかげで、細かいコミット分けのことを考えずにファイルを編集することができます。適当に編集を終わらせたあとでゆっくりステージングとコミットを行い、適切な単位でコミットすることが出来ます。

「は？だるい」という方には `git-commit`に`-a`のオプションをつけることをお勧めします。このオプションをつけることによってコミット時、リポジトリの追跡対象になっているファイルを自動的にすべてステージングしてからコミットすることができます。

```
$ git commit -a -m 'commit all changes of tracked files'
```

また、Git リポジトリは作業ディレクトリに存在するそれぞれのディレクトリやファイルを、「追跡されていない」「変更されていない」「変更がある」「ステージ済み」の 4 つの状態で監視しています。「追跡されていない」状態は、直前のスナップショットに存在していない状態です。「変更されていない」状態は、直前のスナップショットとファイルの内容が同一である状態です。変更があって且つステージングされていないものは、「変更がある」とされます。ステージングされているものは「ステージ済み」となります。

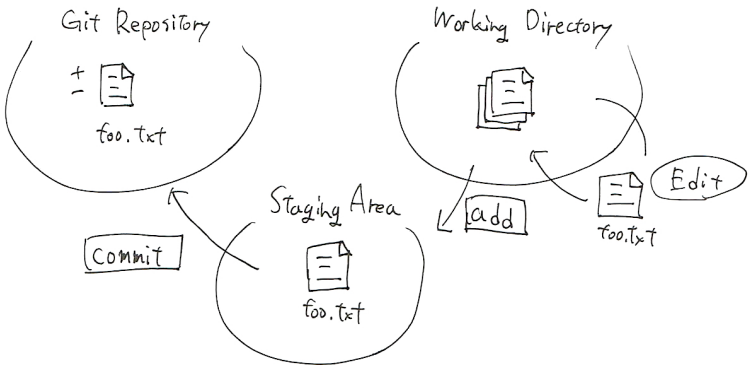



図 4.1 編集からコミットまでの流れ

### 4.1.4 分散型 VCS

Git は分散型 VCS と呼ばれるタイプの VCS です。分散型は集中型と対比されるもので、集中型はリポジトリを一つしか持ちませんが、Git などの分散型 VCS はリポジトリを複数持つことができます。分散型では開発者一人一人が 1 つのメインリポジトリを複製するということができるのです。

VCS を使った開発手法の中で、リモートリポジトリというものを設ける場合があります。これはプロジェクトの本体で、サーバーに置かれます。分散型では、開発者はこのリポジトリの複製（クローン）をローカルの環境に作成し、そこで開発を行います。

分散型のメリットは、リポジトリへのアクセスにサーバーとの接続が必要ないことです。集中型 VCS ではリポジトリにアクセスするコミットなどの操作をするために、サーバーへアクセスできる環境が要求されます。

分散型 VCS にはリモートリポジトリとローカルリポジトリを同期するための操作があります。pushと pullです。ローカルリポジトリの変更をリモートリポジトリへ適用するのが push、リモートリポジトリの変更をローカルリポジトリへ適用するのが pullです。分かりやすいですね。Git を使った開発ではチームで決めたタイミングで pushを行い、リモートリポジトリに変更があった際は pullすると良いでしょう。

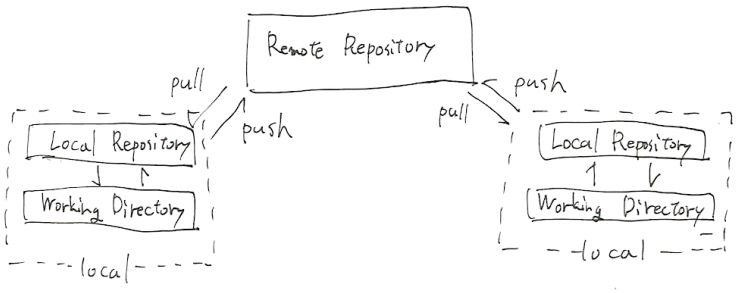



図 4.2 分散型 VCS

### 4.1.5 Git を使った開発の流れ

Git を使った開発の流れについて説明します。Git を使った開発のパターンはいくつも考案されていますが、その大半は反復型開発に親和性の高いブランチの利用です。主軸となるブランチに対してある目的を持ったブランチが切られ、その目的が達成されるとブランチが主軸のブランチにマージされるという形式で、反復の単位を明確にしています。

今回は Git を使った開発の流れの中でもメジャーと思われる “A successful Git branching model”(gitflow) を例として紹介したいと思います。これは Git を活かしたブランチモデルの一つとして有名でして、これを例に Git の開発の流れを掴んでいただきたいと思います。

gitflow ではブランチに 5 種類の役割を与えます。安定版で常にデプロイできる状態である「master」ブランチ、安定版に機能を追加するための副線となる開発版の「develop」ブランチ、develop からブランチして機能を追加する為の「feature」ブランチ、開発版をリリースするために最終調整を行う「release」ブランチ、そして安定版で見つかったバグを緊急に修正を行うための「hotfix」ブランチです。

- メインブランチ
  メインブランチは直接のコミットをせず、ブランチとマージのみを行います。これはメインブランチの安定性を保持するための工夫です。
  - master

  基本的にリリースバージョンで、常にデプロイ可能であるべきブランチです。ブランチの名前はそのまま master とするのが慣習です。このブランチの役割はそれだけですが、これが gitflow の根幹です。このブランチを常にデプロイ可能にする為に以下のブランチを作成します。
  - develop

  master から分岐する開発用ブランチです。ブランチの名前はそのまま develop とするのが慣習です。このブランチの役割は feature ブランチのクッションです。master ブランチへ干渉せずに、機能を実装する為の feature ブランチを束ねます。master ブランチに変更が加えられた場合は直ちにマージして変更を適用します。
- サポートブランチ
  サポートブランチは直接コードの変更のコミットを受けるブランチです。サポートブランチは具体的な機能の実装を目的として切られ、その機能の実装が完了したら破棄される寿命のあるブランチです。ブランチを破棄することで機能の実装完了を明確にします。
  - feature

  develop からブランチし、develop にマージされる実装用ブランチです。future と勘違いする方が時々いらっしゃいますが、フィーチャーです。このブランチを切る目的は、機能の実装やバグフィックスです。ブランチの名前は実装する機能を意味する分かりやすい単語であるべきで、他の命名規則に引っかからない名前ならなんでもいいです。
  例えば WEB サイトを作るプロジェクトでログイン画面を実装しようとするなら以下のようになります。

```
$ git checkout -b login develop
```

実装が完了したら develop へチェックアウトし、マージしてブランチを `git branch -d`で

破棄します。
ところで、feature ブランチは同時に複数存在することが出来ます。自分のログイン画面を実装中に、チームメイトがデータベースへのアクセスを実装していることもあると思います。

```
$ git checkout -b access_db develop
```

feature 同士は干渉せず、開発版である（とりあえず動くであろう）develop からのみ影響を受けるため、feature ブランチでの開発はチームメイトの都合を多少考えずに済みます。Git はファイルをロックしないので、干渉しない範囲であれば同一ファイルを複数のブランチで弄っても大丈夫です。干渉した場合はマージの際に「コンフリクト」が発生しますので、適宜解決してください。コンフリクトの解消は話が長くなるため、今回はコンフリクト解消については「diftool」というヒントだけ残して省略させていただきます。許してにゃん。
ともあれ、ログイン画面の実装が完了したとします。チームの GO サインが出たら早速マージしてサヨナラしましょう。

```
$ git branch
  master
  develop
* login
  access_db
$ git checkout -m develop
$ git branch -d login
```

さて、ログイン画面が実装されて develop ブランチが更新されました。ログイン画面の実装が完了したらしいことを聞いたデータベースの担当者は develop の変更を取り込まなければなりません。

```
$ git branch
  master
  develop
* access_db
$ git merge develop
```

これで最新の開発版の上でデータベースを開発できるようになりました。
これが大体の feature の流れです。

– release

develop からブランチし、master にマージされるリリース用ブランチです。ブランチの名前は `release-foo`の形にするのが慣習です。foo の部分はバージョン番号であったりビル

ドの日時であったり、バージョンを示すものになることが多いと思います。
このブランチではバージョン情報の更新など、リリースのための最終調整を行います。もちろんここでバグフィックスを行う可能性もあります。ここでは開発版を次の安定版としてリリースできる状態にします。
開発が完了したら master にマージしてブランチを破棄します。release を破棄したあと、バージョンを記録する為に `git-tag`でタグをつけることがあります。

- hotfix

master からブランチし、master にマージされる緊急のバグフィックス用ブランチです。ブランチの名前は `hotfix-foo`の形にするのが慣習です。foo の部分は修正内容を示す短い言葉が好ましいです。この役割のブランチも複数同時に存在することがあるかもしれません。
ここまで来たらもう分かると思いますが、このブランチはバグフィックスを目的とし、それが終わるとマージされて破棄されます。
ただ一点あるとすれば、通常 hotfix での修正もバージョンを持つべきだということです。例えばバージョン `1.2`の hotfix による修正版は `1.2.1`なんかになると思います。これは、ユーザにバグの修正を伝えるために重要な要素です。
release のときもそうですが、このブランチは master によってマージされます。master がマージする（変更が加えられる）ので、連鎖的に develop が master をマージすることになります。ですので、gitflow の説明では develop が直接 release や hotfix をマージするとあります。

これらの役割を持ったブランチを切って切って切りまくって開発を進めて行きます。

「master から develop を切って develop から feature を切ってマージして feature を切ってマージして... develop から release を切って master にマージして develop が master をマージして feature...」

これが gitflow の大体の流れになります。
注意として、gitflow でマージを行う際はマージした履歴を残す為にファストフォワードを無効にする必要があります。全てのマージに`--no-ff`オプションをつけるのも面倒ですから、

```
$ git config merge.ff no
```

などとして gitconfig で常にファストフォワードを切っておくことを推奨します。ファストフォワードって何だよと思う方もいらっしゃると思いますがここでは「マージの履歴を残す為に切った方がいいやつ」という説明に省略させていただきます。めっちゃ誤解させる説明ですがここではそんな感じです。

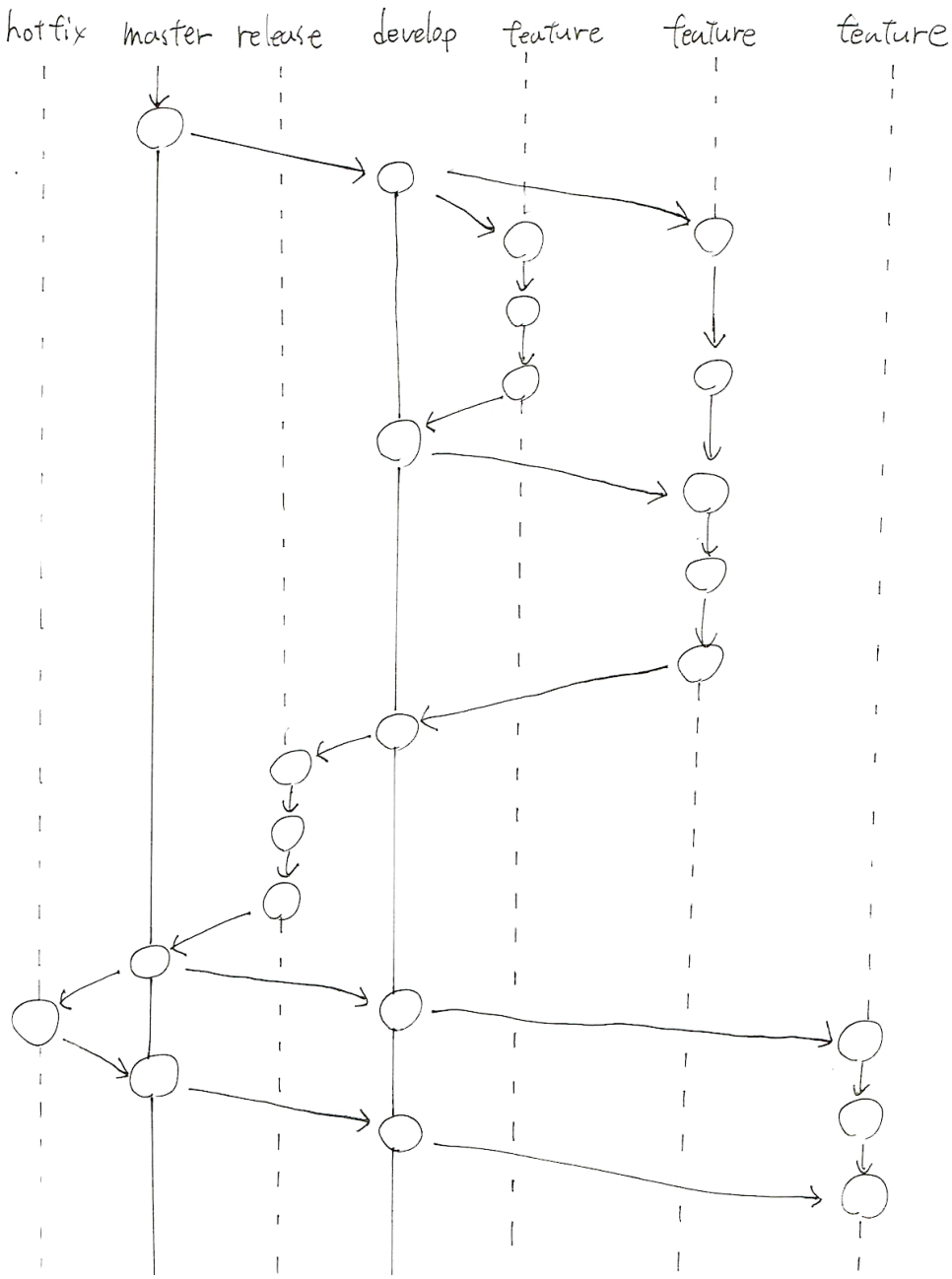



図 4.3 手作り感溢れる Gitflow

### 4.1.6 Git やってみる

それではちょっと Git に触ってみましょう。Git を始めるには、まずバージョン管理を行いたいプロジェクトのディレクトリで git-initコマンドによって Git リポジトリを作成する必要があります。このコマンドによりカレントディレクトリに.git/ディレクトリが作成されます。Git はこの Git リポジトリに「Git オブジェクト」と呼ばれる blob (バイナリラージオブジェクト) を保存することでバージョン管理を行います。リモートリポジトリをクローンしたい場合は git-cloneを使います。

```
$ cd /path/to/project
$ cd my_project
$ git init
$ # or
$ cd /path/to/project
$ git clone git@git.example.com:my_project.git
```

さて、めでたく Git リポジトリが作成されました。あとは git-branchでブランチを切り、ファイルを編集して変更を git addでステージングし、変更を git-commitして変更してコミットして...ブランチを git-mergeしてブランチを切って... 基本的にこれを繰り返すことで開発を進めることが出来ます。リモートリポジトリがある場合は git-pushや git-pullも必要です。これらの操作に必要な Git コマンドの簡単な使い方を以下に紹介します。

```
$ # create ‘foo‘ branch from ‘develop‘ branch
$ # and check out to new branch
$ git checkout -b foo develop
$ touch hogehoge
$
$ # staging ‘hogehoge‘
$ git add hogehoge
$
$ # commit
$ git commit -m ’add hogehoge’
$
$ # push local reposiotry to ‘origin‘ repository
$ git push origin foo
$
$ # pull ‘origin‘ repository into local reposiotry
```

```
$ # and apply the changes of ‘develop‘ to ‘foo‘
$ git pull origin
$ git merge develop
$
$ # checkout to ‘develop‘ and merge commits
$ # into ‘develop‘ from ‘foo‘
$ git checkout -m develop
```

## 4.2 Git サーバー

Git のリモートリポジトリを管理するサーバーを Git サーバーと呼んだりします。Git ではデータ転送のプロトコルに Local、HTTP、SSH、Git の 4 種類のプロトコルを使用でき、これらを用いてリモートリポジトリを建てることができるので軽く紹介します。

### 4.2.1 サーバーの構築

Git サーバーを建てるためには、リポジトリをエクスポートして作業ディレクトリを持たないサーバー用のリポジトリ「ベアリポジトリ（the Bare Git Repository)」を作らなければなりません。既に存在するリポジトリのベアリポジトリを作るときは、`git-clone`に`--bare`オプションを指定することで簡単に作成することが出来ます。

```
$ git clone --bare my_project my_project.git
```

このコマンドによって.git ディレクトリを持つ作業ディレクトリ `my_project`のベアリポジトリ`my_project.git`が作成されます。慣例として、ベアリポジトリの末尾は`.git`とすることになっています。新しいリポジトリをベアリポジトリとして作成する場合はもっと簡単です。

```
$ git init --bare my_project.git
```

これでおおよその操作は完了です。 なんて簡単なんでしょう！ここでサーバーにリモートリポジトリを作成するには、`git.example.com`サーバーに `git`ユーザとして SSH で`/opt/git`にアクセスするとして、`scp`コマンドを用いて

```
$ scp -r my_project.git git@git.example.com:/opt/git
```

とすれば`/opt/git/my_project.git`というリモートリポジトリが作成できます。この時点でこのサーバーに SSH でアクセスできて`/opt/git`ディレクトリへの読み込み権限があるユーザなら、こんな感じでリポジトリをクローンできます。

```
$ git clone git@git.example.com:/opt/git/my_project.git
```

よく見る画面ですね！また、リモートリポジトリに対する書き込み権限を git-initコマンドの--sharedオプションで与えることが出来ます。

```
$ git init --bare --shared my_project.git
```

--shared=0xxxの形で umask の値を指定することも出来ます。

### 4.2.2 Local プロトコル

これは、リモートリポジトリを同一ディスク上の別のディレクトリに置くものです。主にファイル共有システムがある環境で使われると思います。Git へのアクセスは以下のようになります。

```
$ git clone /path/to/project/my_project.git
```

あるいはこのように表現することも出来ます。

```
$ git clone file:///path/to/project/my_project.git
```

file://プレフィックスがあるときと無い時では少し挙動が違います。これがないとき、Git は単純にハードリンクを張ったりそのままコピーしようとします。あるときはトランスファープロトコルを通してデータを転送します。後者の方が確実に Git のデータが転送されますが、非常に非効率的です。他の VCS から Git にインポートしたでも無い限り Git リポジトリは無駄な参照やオブジェクトを持たないので、普通プレフィックスなしで問題ありません。

Local プロトコルを使う利点はシンプルであることと、ファイルのアクセス権を流用できることです。欠点は LAN 外からリポジトリにアクセスする際にリモートディスクをマウントする必要があること、全ユーザがシェルでリポジトリにアクセスできるため、Git 内部ファイルの変更・削除を防止できないことです。

### 4.2.3 HTTP プロトコル

HTTP プロトコルは HTTP を使う Dumb HTTP と HTTP/S を使う Smart HTTP の 2 つのモードがあります。

Dumb HTTP プロトコルは、ベアリポジトリを HTTP 上にそのまま上げておくスタイルです。Smart HTTP プロトコルにサーバーが応答しない場合、Git クライアントは簡易な Dumb HTTP プロトコルへフォールバックします。 Dumb HTTP プロトコルをセットアップするのはとても簡単で、ベアリポジトリを HTTP ドキュメントに配置し、ベアリポジトリの post-updateフックで git update-server-infoを実行させるだけです。例として Apache の標準設定でこれを行うためには以下のようにします。

```
$ cd /var/www/htdocs
$ git clone --bare /path/to/project/my_project my_project.git
$ cd my_project.git
$ echo '#!/bin/sh\nexec git update-server-info' > hooks/post-
    update
$ chmod a+x hooks/post-update
```

Dumb HTTP プロトコルではリポジトリへの書き込みに認証・暗号化を行わないので、通常読み込み専用のリモートリポジトリとして運用することになると思います。

Smart HTTP は Git プロトコルが提供する匿名の読み込みと SSH プロトコルが提供する認証・暗号化機能の両方が同時に実現できます。このプロトコルを使う為には Git に同梱されている`git-http-backend`という CGI スクリプトをサーバー上で走らせる必要があります。この CGI はクライアントが git でサーバーへ送る`git-fetch`や`git-push`のデータのパスやヘッダ情報を読み込み、クライアントが HTTP/S を使ってやりとりできるかどうかを判断します。できなければここで Dumb HTTP プロトコルにフォールバックします。

dumb のときと同様に Apache での設定とします。

```
SetEnv GIT_PROJECT_ROOT /path/to/project
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend
```

`/git`パスを CGI のハンドラに設定しています。`GIT_PROJECT_ROOT`は CGI が参照するパスで、ここ以下のリポジトリにアクセスできるようにします。`GIT_HTTP_EXPORT_ALL`はすべてのリポジトリを読み込み可能にする設定で、これをしない場合は認証のないクライアントは空ファイル`git-daemon-export-ok`が存在するリポジトリにしかアクセスできなくなります。

次に、例として Apache のセクションコンテナの設定でさきほどのパスへのアクセスを許可します。

```
<Directory "/usr/libexec/git-core*">
    Options ExecCGI Indexes
    Order allow,deny
    Allow from all
    Require all granted
</Directory>
```

最後に書き込みのために Auth ブロックでユーザ認証を求めます。

```
<LocationMatch "^/git/.*/git-receive-pack$">
    AuthType Basic
    AuthName "Git␣Access"
    AuthUserFile /path/to/project/.htpasswd
    Require valid-user
<LocationMatch>
```

許可されたユーザーのリストを作成します。

```
$ htdigest -c /path/to/project/.htpasswd "Git␣Access" user
```

ちょっと面倒ですかね。さっきからなんか Apache 使っていますが、Apache で話せば大体伝わるだろうというのが理由で、特に Apache が推奨されていたりはしません。要求するのは git-http-backendが動く為に CGI が走る環境だけです。

HTTP プロトコルの利点は、ブラウザでリポジトリを見る時に使う URL をそのままリモートリポジトリとして扱えることです。また Smart HTTP を使う場合は先ほど挙げたように、SSH プロトコルの欠点を補えることです。

### 4.2.4 SSH プロトコル

SSH プロトコル越しにリポジトリをクローンする方法は 2 種類あります。1 つは ssh://スキームを利用する方法と、もう 1 つは scp コマンドのように指定する方法です。

```
$ git clone ssh://user@path/to/project/my_project.git
$ git clone user@path/to/project:my_project.git
```

ユーザ名を省略することもできます。その場合ユーザー名は現在ログインしているユーザーのものになります。

SSH を使う利点は、ネットワーク管理者にとって SSH はありふれたツールなので準備が楽であること、そして何より SSH 越しのアクセスには安全性があることです。欠点は、匿名でリポジトリにアクセスすることが出来ないことです。読み込み専用であっても SSH の接続を要求するためリポジトリの配布には向いておらず、これをするためには読み込み専用の別のプロトコルを用意しなければなりません。

### 4.2.5 Git プロトコル

Git プロトコルは、Git デーモンを用いた認証無しの読み込み専用リモートリポジトリを提供します。このプロトコルはデフォルトでポート 9418 番を食います。Git デーモンは以下のようにして使います。

```
$ git daemon --reuseaddr --base-path=/path/to/project/ /path/to/
    project/
```

--reuseaddrはサーバーの再起動のために前の接続がタイムアウトするのを待たないようにするオプションです。--base-pathは読んで字の如くアクセスの際の基本ディレクトリを指定するオプションです。他にも--portで使用ポートを変更したり、--timeoutでタイムアウト秒を設定できたりします。

Git プロトコルでアクセス可能なリポジトリは git-daemon-export-okという空ファイルを持っている必要があります。

```
$ cd /path/to/project/my_project.git
$ touch git-daemon-export-ok
```

これでさっきのデーモンを走らせれば git://スキームでアクセスすることができます。セキュリティを考慮して、デーモンはリポジトリに対して読み込み権限しかないユーザで実行しましょう。

```
$ git clone git://git.example.com/my_project.git
```

### 4.2.6 Git サーバーホスティング

自前で建てられるといっても実際自前で Git サーバーのシステムを用意している人は多分そんなにいません。Git サーバーの需要に応えるシステムが世の中にはたくさんございますので有名どころを紹介したいと思います。

- Bitbucket

HipChat や Stash で有名な Atlassian が運営する VCS のホスティングサイトです。無料アカウントで 1 つのリポジトリに 5 ユーザーまで使用可能で、プライベートリポジトリを無制限に持てるという圧倒的なメリットがあります。5 人を越えるチームでの開発はチームの人数によって決められた月額を支払う必要があります。Bitbucket は Git だけでなく Mercurial のリモートリポジトリを建てることが出来ます。ちなみに学生向けの制限解除もあるっぽいのですが、私は登録しておらず、.edu ドメインについてのみ書いてあるので.ac.jp でいけるのか分かりません。

- Bitbucket Server（旧 Stash）

Bitbucket が Atlassian の提供するクラウドサービスなのに対して、Bitbucket Server は自前のサーバーで建てることの出来る Bitbucket です。つい最近、分かりやすくするために Stash がこの名前になったようです。これは企業などクローズドチーム向けのサービスです。

- GitHub

Github, Inc. の運営する、恐らく最も技術者に浸透している Git サーバーのホスティングサイトです。グラフィカルにバージョン管理をできるだけでなく、Pull request や Issues などの共同開発を助けるサービスで多くの支持を得ています。また、Git ホスティングだけでなく、スニペット共有サービス Gist、プロジェクトのホームページを作成できる GitHub Pages などの開発者のためのサービスが提供されています。プライベートリポジトリの作成には有料プランの登録が必要ですが、学生は申請を認証されることで Micro プラン相当のライセンスを得ることが出来ます。月 7 ドル儲けです。 やったね。

また、企業向けに自前のサーバーで GitHub を走らせる GitHub Enterprise というサービスもあります。

- GitLab

キモいアライグマのロゴが目印の Git サーバーです。GitHub ライクな Git サーバーを手軽に自前で建てられます。いつからあったのか知りませんがクラウドホスティングもやってます。UI は好みが別れると思いますが、私は GitHub より GitLab の方が好きです。企業向けの GitLab EE(Enterprise Edition) もございます。なんか NASA とか OReilly も使ってるらしいですよ。クローズドな環境でソースコードを管理したい時などいかがでしょう。ちなみにロゴのアライグマが世界中からキモがられているため最近ロゴが変更されました。

## 4.3　終わりに

尺も時間もヤバいという事情により今回書ききれなかった内容がかなりございます。ですので、まず本記事に興味を持たれた方は本記事の参考文献である Pro Git を読まれることをお勧めします。また、機会があればサークルの何かしらの際にハンズオン形式で Git の説明を行いたいと企てておりますので、ぜひとも当サークルに一目置かれたいと思います。

# 第 5 章

# マイコンを使ってカーライフを快適なものに…

めの

## 5.1　経緯

誰もが作ったことのある (?) ライントレースカーや、簡易ラジコン。これらはきっとマイコン制御で動かしているはず。ここではマイコンを使用し、夏の間に行ったカーライフを快適にしようと奮闘した話を書いていこうかと思います。他の先輩方が執筆された文章の方が遥かに素晴らしいためこちらは軽い箸休め、暇潰しレベルで読んで頂けると幸いです。

### 5.1.1　軽い自己紹介

情報システムデザイン学系　ネットワークシステムコースな人。CTF とかに興味もあれば旋盤や CAD、3D プリンターとか使って物作るし、たまに溶接とかもします。ソフトウェアなことはまちまち。全く専門外である商業系資格取得するわで変に広いジャンルを彷徨ってる人間です。興味が湧いたらすぐ実行！→飽きる！→次へ...がデフォ Github はリアルネームでやっているのでググればすぐ出ちゃう（こわい）。本記事で掲載されているプログラム等も Github に全てあがっています。(今は上げていませんがきっと鳩祭後の自分が頑張ってます)

### 5.1.2　使用機材とか

- NISSAN MARCH AK12
- Arduino uno
- Mac Book Pro
- 各種センサ

### 5.1.3　本記事での犠牲者

- サーボモーター

## 5.2　本編

制御装置の構成面では企業や個人しかり、この技術が溢れかえり、考えが出尽くされたこの世の中で、それぞれ独自の工夫がでにくい状況になってるといえると考えられます。その中で組み込み機器の特徴をいかに創出するかが今後の課題と言えると考えられます。

そこで自分が考えたのは Arduino uno 程度のマイコンでも実現可能であり、まだ実車では実装されていない機能（もしかしたら企業側で不必要とされ、切り捨てられたかもしれませんが）を本記事ではちょこちょこと実装していこうと思います。

まず実装しようと思ったものである「雪が降った時にワイパーを自動で上部にあげワイパーを雪から守るっぽい機械」を実装しようと考えました。

これは昨年の冬、自分自身の実体験からこの考えが思いつきました。

昨年の冬に降った雪絶対に許さねえ

なんやかんやで製作時間はハード 1 時間にソフト 1 日とそんなにかからずに製作することができました。

プログラムは使用するピン番号を Setup で指定して loop 関数で宣言し、サーボモーターを制御するために使う analogWrite 関数とダクトスイッチを使い雪の重みで動くするようにするために、ダクトスイッチの制御が必要となってきます。そのため、Setup 関数で指定した button という変数の番号にダクトスイッチを接続するので loop 関数には digitalRead(button) という形でプログラムを書きました〜。

検証はは季節上どうしても雪が降る機がしないので友人に頼んで上からかき氷器で人工的に雪を降らす。といったあまりにも頭の悪い方法で検証しました。あまり暑い日でもなく日陰での検証でしたので友人がそこまで疲れずに済んで助かりました。

## 5.3　反省

結果にこだわりすぎて写真撮り忘れた。防水化してなかったのでサーボモーター 1 個壊した電池ではなくバッテリーから供給してるが消費電力が小さいとはいえ常時起動はコスパ悪すぎ最悪バッテリー上がる

## 5.4 後書き

本来、音拾ったりするセンサーや超音波センサーでも用いて雨が降ったことを検知したらワイパー動かすものでも作ろうかなって思ってました（ワイパーに固着し過ぎとかそういうツッコミは無しで）

ですが、内蔵されているものをいじるため、施工にちょっと危険であることや、それ以前に多くのメーカーで実際に採用されていることを尻、実行には移しませんでした。また、この機会に自分の知識の無さと各自動車メーカーが行っているあらゆることの自動化がどんどん進んでいるなぁって思いました。(こなみ)

# 第 6 章

# マイコンを使った IC カード摘出的なアレ？

めの

## 6.1　経緯

みなさんご存知 IC カード、我が大学の学籍証も IC カードです。（これが結構便利で e-pass の代わりに使えます。さあみなさんも学籍証で音ゲーを始めて単位を落とそう！）一般的には IC カードとは、IC(集積回路) を搭載したカードであり、その IC には情報の記録や演算をするための役割があります。ときたま、スマートカードとも呼ばれるそうです。実際に広く知られている例ですと、Suica やワオンカードがありますね。前置きが長くなりましたが、私はこの IC カードの技術を使用しもっと日常生活を便利に出来ないのではないかと考えます。

### 6.1.1　使用機材とか

- Windows 7 Ultimate
- Mac Book Pro　Yosemite
- Arduino uno 1 台
- ブレッドボード
- 各種 IC カード
- KONAMI e-AMUSEMENT PASS
- 学生証

## 6.2　本記事での犠牲

- 学生証
- e-AMUSEMENT PASS

### 6.2.1 本編に入る前の注意書き

部誌に書いておいてアレですが、本編の記事を真似して下手に IC カードをイジイジするのはやめましょう！最悪 IC カード中のデータを吹き飛ばしたりその IC カードを再起不能にさせることになってしまいます！

~~ぼくは学生証 1 枚と e-PASS2 枚を犠牲にしました！~~

どーしても IC カードの情報が見たい！改変したい！って人はネット上に先駆者が多く居ますのでその方々を参考にしましょう。

## 6.3 ほんぺ

早速ですが Arduino uno とブレッドボード、IC カードを読み取るためのセンサである RC522 を接続します。接続図はこのセンサ側に記述してあったままです。

配線図も Github にのせる予定ですが、ここで簡単に説明します。

RFID センサの端子側を左側に置いて上から SDA、SCK、MOSI、MISO、GND、RST、Vcc と言った形。そんな IC カード情報が読み取れる超便利な機械を使って何をするか。それは「IC カードを使って快適な BMS ライフを送ろう」って感じのことをします。(これをタイトルにしようと思いましたが読まれ無さそうなのでやめました)

みんな大好き音楽ゲーム、特にゲーセンで遊ぶ方が多いかと思われます。そして某音ゲーを嗜んでいる方はきっとオウチマニアしている方も多いはず。

自分は暫く BMS 勢でいたのですが最近 AC にハマってしまいまして。ほんと某ぷりん許さねえ BMS と AC 版の違いといえばやはり e-pass をかざすかかざさないかですよね。筐体に e-pass かざしてクロノスとカイロスがお迎えしてくれるだけできっと地力が上昇するはず。そう考えた僕はさっさと実装することにしました。

処理の流れとしては

e-pass をかざす→ WelcomeVoice 流す→ BMS 起動→幸せ

といった流れです。

まず e-pass をかざして WelcomeVoice と LR2 を起動する流れとしては Arduino と RFID 読み取りセンサを使用して適当な磁気を発生するカードをかざしたらそれらが動く、と言った形にすれば良いと考えました。C なら以下の様なプログラムなのですが、Arduino の関数をあまり把握してないためそこのとこを調べるところから始まりました。

```
int main(void)
{
    /* LR2の場所の絶対パス */
    char buf[] = "\"C:\\Program Files\\Internet Explorer\\
       IEXPLORE.EXE\"";
    system(buf);
    return 0;
}
```

調べてみると外部 exe 拡張子を開く関数は C と同じ system 関数で良いようです。そして IC カードを読み込んだと確認する記述はこれ。

```
if ( ! mfrc522.PICC_IsNewCardPresent()) return;
```

んで以下のように書いてあげれば読み込んだ際に Loop 内の処理を行ってくれます。

```
void loop() {
    if ( ! mfrc522.PICC_IsNewCardPresent()) return;
}
```

そしてお次は条件を満たした処理の流れの中に音楽を流すプログラムと肝心の LR2（BMS）を起動する関数をぶち込むだけで完成です。これで快適な BMS ライフが送れますね。GitHub にも全く同じのが上がっていますので実装したい！改変してもっといいの作りたい！って変人さんはどうぞ。

思い立ってから実行〜実装〜結果まとめ（本記事）まで 1 時間ちょっとでした。1 日 1 プログラミングってのもいいかもしれませんね。

追記：epass かざしてパスワード入力（LR2 起動にあるアレ）を別 GUI を制作して促しても面白いと思ったので近いうちにやります。

# 第 7 章

# update_name と生きる

しゃみそん (@_sham258)

ネタ記事です. まともに読もうとしたら損をします. あと Twitter してないと理解に苦しんだりするかもです.

## 7.1 update_name とは

update_name とは「Twitter 上でリプライ等のツイートで, Twitter のスクリーンネームを変更する」というものである.

おそらく暇人しかやらないであろう (要出典) 機能であるが, 個人的に最高に楽しいのでおすすめだ.

プログラミングしたいけどネタがない方や, Twitter が好きな人や, ファボ魔の人や, 初音ミクが好きな人などはやってみると, 楽しかったりすると思われる.

また, 近年において, 派生のものとして, 「update_icon」であったり, 「update_location」などの, 「ツイートによってプロフィールを変更するもの」 も観測されており, 今後の発展が期待される (?) 分野の一つである.

## 7.2 update_name の歴史

update_name の歴史は長く, 2015 年 10 月現在のツイート検索の結果からすると, 最古の update_name は, 「2013 年 8 月 27 日」に post されているようだ.

当時の update_name は 「@XXX update_name 新しい名前」のように post することで名前が変わるもので, 現在の update_name の基礎が, ツイートの一文字一文字から読み取れるものとなっている.

また, 2013 年が update_name の誕生の年ということもあってか, 「update_name Advent Calendar 2013」などもあり, 当時の勢いというか, 熱気というか, そういったものが感じられる. (Advent Calendar には当時の記事が見れるので必見である.)

## 7.3　update_name の良さ

さてこれまで update_name についてのみを記述してきたが, 「update_name を利用したことも聞いたことも無い」という方や, 「Twitter？ なにそれ？ 美味しいの ?」という方 (今すぐ Twitter に登録しましょう) からすれば, 「update_name なんてわざわざ実装しなくても, 手動で名前変えればいいじゃん. 何が楽しいの???」 となることが必至だ. ここらで良さを列挙していこうと思う.

update_name をすることで...

- フォロワーさんに名前を変えてもらえる!!
- Twitter に新たな楽しみ方が生まれる!!
- サーバに詳しくなる!!
- 身長が伸びる!!
- 滑舌が良くなる!!
- 知らない人と繋がることができる!!
- プログラミングができるようになる!!
- 大好きなあの娘に思いが届く!!
- ロリババアの彼女ができる!!
- ミクに会える!!
- 島村卯月さん結婚してくれ!!
- Twitter の仕組みへの理解が進む!!
- 新たな人格が芽生える!!
- etc, etc...

となっている.(半分冗談で半分本当である.)

## 7.4　update_name を始める

読者の皆様に update_name の魅力が伝わったところで, 実際に update_name を実装してみようと思う.

まずは実装に必要なものとして,

- 特定のプログラミング言語への知識
- 動かすサーバ (Heroku 等もある)

が挙げられる. また, 個人的にあるよいと思われるものとして,

- ライブラリを導入する知識

- java でいう maven や ruby でいう gem など
- ネットワークや認証の知識
- ライブラリを使わない場合はかなり必要

が挙げられる. といってもそんなに高いハードルではないのでぜひ手を出していただきたい.

### 7.4.1 おすすめライブラリとその使い方

ここで言語別のオススメライブラリを列挙していこうと思う.

- Java(等の JVM 言語)

圧倒的に Twitter4j をすすめる. 開発者の方が超イケメンの方らしいので, 使うと自分もイケメンになれるとか.(個人差があります) (あと普通に使いやすいです)
使用する際には「maven」「gradle」「sbt」「leiningen」などを利用するし導入すると良いと思われる.

- Ruby

```
gem install twitter
```

でおしまい!

- Python

```
pip install twitter
```

を推す. python 界隈では「tweepy」や「twitter-python」など, 多くのものがあるので色々試してみると良いかもしれない.

- C#

書いたことは無いが, 友人からの知見で「CoreTweet」を使うと良いらしい.
ライブラリは VS 様のちからをつかって「Nuget」ってのを使うと良いらしい...

### 7.4.2 実装する上での方針と注意点

実装した際, プログラムがどのような動きをして名前を変更していくのかについて考えて見よう.

1. Twitter への認証
2. Twitter のタイムラインを監視. ツイートを一つずつ見ていく.
3. 特定の文字列に対して反応.

4. ツイートから情報の抽出と, 加工.
5. 名前の変更.
6. 反応したツイートをした人に対してリプライを送る.
7. タイムライン監視に戻る.

のような形である. ここから一つ一つの要素について詳しく見ていく.

- Twitter のタイムラインの監視.

通常の REST API を使うと, 15 分に 15 回のみしか Tweet を取得することができない. ので, REST API ではなく, STREAM API を使う. 中でも「userstream」と言われ, これを使うことで, ユーザのホームタイムラインをストリーミングすることができる. つまり, ツイートを見まくりである.

なおサードパーティ製の Twitter クライアントでは, この機能が実装されていることが多く, ツイッター廃人の方々からすれば常識であることだろう.

また, API 利用の際は Twitter へアプリの登録が必要となり, 予め「comsumer key」「comsumer secret」が必要となり, その後, その 2 つのキーを用いて「access token」「access token secret」を取得することで, 初めて各 API を自由に使うことができるようになるので, そこら辺の登録とかもしていかないと行けない.

- 特定の文字列に対して反応.

流れてくる文字列に対し, いわゆる, 「フィルター」をかけていく.

単純に「@XXX update_name 名前」に反応するのみである場合は, 殆ど必要無いが,

「名前 (@XXX)」という形であったり, 特定の語尾に対して反応等の, 独自性を求めていくと,

「RT には反応しない」とか「@が名前に含まれていたら反応しない」等の処理が, 必要になってくるので, 注意が必要である.

- ツイートから情報の抽出と, 加工.

まず反応したツイートから情報を抽出する. ひとつはリプライを返すために「つぶやいた人の ID」. もうひとつに, 「ツイートの文字列」が挙げれれる. 文字列を取得した際, 「文字列から名前を抽出する」という行為が必要となる. 正規表現を使うと良いだろう. また, 名前の文字数制限として, 「20 文字以内である」ということを, 注意していただきたい.

- 名前の変更.

抽出した名前を実際に適用する. 変更する際は「update_profile」という api を用いる.

- 反応したツイートをした人に対してリプライを送る.

先ほど覚えておいた ID と組み合わせて, リプライを返してあげる.
と, 以上である.

### 7.4.3 例) Python で実装する

実装の容易さから考えて Python が良いと思ったので簡単に紹介してみる. このプログラムは最適とは言えるものではないので, 雰囲気を感じ取ってもらえるとありがたい.
事前準備として,

```
pip install twitter
```

して上で以下のコードを実行した. また, 各種キーは各位のキーに読み替えて読んでいただきたい.

```
# -*- coding: utf-8 -*-
__author__ = 'shamison'

from twitter import *
import os

# userstream用に作成.
oauth = OAuth(
  consumer_key="コンシューマーキー",
  consumer_secret="コンシューマーシークレット",
  token="アクセストークン",
  token_secret="アクセストークンシークレット"
)

# ツイートやらプロフィールを取ってくるため作成
tw = Twitter(
  auth=OAuth(
    "コンシューマーキー",
    "コンシューマーシークレット",
    "アクセストークン",
    "アクセストークンシークレット"
  )
)
```

```
# screen_nameを記憶
my_name = '@' + tw.account.settings()['screen_name']

# user_streamをする.
tw_us = TwitterStream(auth=oauth, domain='userstream.twitter.com
    ')

# update_nameの実装
def update_name(msg):
  # tweetの文字列の加工
  new_name = msg['text']
  new_name = new_name.replace(my_name + ' update_name', '')

  # 名前の制限
  if len(new_name) > 19:
    tw.statuses.update(status='@' + msg['user']['screen_name'] +
      '文字数制限です> <')
  else:
    tw.account.update_profile(name=new_name)
    tw.statuses.update(status='@' + msg['user']['screen_name'] +
      '名前を' + new_name + 'に変更しました!')

# tweetを取得する.
for msg in tw_us.user():
  # 必要が無いものが流れて来たらスルー
  if "friends" in msg:
    continue
  elif "delete" in msg:
    continue

  # update_nameするかを判定.
  # 特定の文字列が流れてきたらupdate_nameする.
  if "text" in msg and (msg['text'].startswith(my_name + '
     update_name')):
    update_name(msg)
```

みたいな感じだ.

## 7.5 update_name を応用する

ここでは筆者が観測した, update_name の応用だと思われるもの, について言及しておこうと思う.

### 7.5.1 name(@XXX) 形式

もはやスタンダードの形式と言ってもよいだろうが, 「名前 (@XXX)」という形式で名前を変更するものである. これにより長い文字列を打つ必要がなくなり, 最も update_name の発展を担った形式であると言えるだろう.

### 7.5.2 update_profile

ここでは主に「名前」に限らず, プロフィールにおける, 「説明文」「現在地」「URL」に関する部分を変更するというものを, 「update_profile」としてまとめて表記する. これらはそれぞれ,

- update_description
- update_location
- update_url

として利用されているようだ.
利用法としては, 実装してある人に対して, まるでメモ帳のように使用するという方法がある.

- 例: 休校情報を description に update し, 名前を休校情報に変え, location に学校名, url を画像にする, など

### 7.5.3 語尾による update_name

これは近年筆者の周りの人間において見られるものであるが, ツイートの語尾に「っちー」, 「そん」, 「犬」, 「菜」などの特定の語尾になった時に名前を「そのツイートの文字列」に変更する, というものである. この大きな特徴として,

- リプをとばすことなく名前を変更できる
- 特定の語尾がつくことでアイデンティティの崩壊を防げる
- オリジナリティを出せる

という点がある.

また, これに関して, 該当ツイートに対してリプをするかしないか, という点が良く挙げられる.

予期せぬ update_name 事故を防ぐか, もしくはそれを楽しむか, それは実装者の裁量で決定されるのだ.

### 7.5.4　update_icon

ツイートによりアイコンを変更するというものである. 一見よさ気なような感じがするが, これを実装したものは update_name とのコンボで, ほぼ確実にプロフィールにおけるアイデンティティを失うだろう.

筆者が観測した事例を上げると, アイコンをおもむろに某大盛り系ラーメンの画像に変えられた後, 名前を「ラーメン二郎三田店」 に変更される, 等のもので, 自分の知らないところで, 勝手に非公式二郎アカウント化しているということがあった. 自分があのようなおぞましいラーメンのアカウントになると思うと, 恐怖で夜も眠れません. update_name 界の異端児と言えるだろう.

### 7.5.5　update_default

こちらは先程と変わって, 混沌とかしたプロフィール情報をリセットするものである. アイデンティティを一時的に取り戻すことができるが, それは一時的に過ぎないものであるだろう.

また, プロの update_namer たちからすればそれはまさしく「逃げ」. update_default を実装したということは, 所詮エンジョイ勢でしかなく, プロ update_namer には程遠いのだ.

(筆者は update_default をよく使います.)

### 7.5.6　その他

その他として挙げられるものとして, update_name を利用して, リプをすることで, 様々なアクション (天気, くじ引き, ログ出力, etc...) をおこすというものがあるが, 幾つ上げても網羅できるものではないため, 割愛する.

## 7.6　まとめ

昨今における update_name は汎用性が高く多くのものに活用でき, また日々の Twitter の生活に潤いをもたらすものである. 本記事によって update_name を実装しようと思う人が, 一人でも増えれば幸いである.

# 第 8 章

# 誰も得しない，Linux 導入奮闘談

zacky

始めに，この記事はバッドエンドです．(私は約 40000 円も溶かしました)　また，Windows8 以降の OS プリインストールモデル notePC ユーザー向けの記事です．
私が Linux に憧れて notePC 分解やブートローダー周りの変更，投げやりになってすべてを削除してしまった末に「おとなしく Windows でなんとかしよう」となった話．

## 8.1　やったこと

今回使用した PC は KIRA V63/PS(TOSHIBA)．Windows8 プリインストールモデルでメモリは 8GB，SSD128GB 内蔵．
いたって普通の，いわゆる “最近のノート PC” といった感じ．

- 失敗から学んだバックアップ手順
- Linux 導入（失敗談）
- もし失敗したら
- Windows でがんばる

## 8.2　失敗から学んだバックアップ手順

一般的にはリカバリーディスクまたはリカバリー USB を作成することが推奨されるが，**OS 関連の作業では HDD/SSD のクローンを作ることをお勧めする．**
なぜなら HDD/SSD が完全にフォーマットされてしまった場合，Windows プリインストール版 notePC 内蔵の HDD/SSD に存在する “隠しパーティション” なるものが消えてしまい，リカバリーメディアが機能しなくなるからだ．

1. 必要なもの
   - もちろん自分の notePC（今回は KIRA V63/PS）

- PC に積んである SDD 容量と同等かそれ以上の SSD（mSATA）
- mSATA SSD( SATA3/6Gbps 対応) → USB3.0　小型外付けドライブケース (EnlargeCorp)
- 自分の notePC のネジ穴に合ったドライバー

2. クローンをつくる（バックアップ）

クローンとは，データだけでなく OS の入ったパーティションごと SSD/HDD をコピーすることである．クローンを作ること（クローニング）により，どんな事態が起きても大体のことは元に戻すことができる．

まず，『EaseUS Todo Backup free』というソフトをダウンロード&インストールする．小型外付けドライブケースに SSD をセットし，notePC に繋げる．

『EaseUS Todo Backup free』を起動しクローンを選択．まずはコピー元のディスク（自身の PC の SSD）を型番や容量で判断し選択．その後コピー先のディスクを選択する．このとき，”Optimize for SSD” にチェックを入れる．(HDD から SSD にクローニングする際に必須らしいのだが，今回は SSD から SSD へのクローニング．和訳すると “SSD 最適化” とのことなので私はチェックを入れた)

オプションでパーティションサイズを変更してクローニングすることもできるが，今回は割愛．

後は流れに任せて進めていけばクローニングが始まる．終わったら安全な取り外しをお忘れなく．

## 8.3　Linux 導入（失敗談）

始めに，**Windows8 以降の OS プリインストールモデルの notePC** はマザーボードの関係上，他 **OS** のインストールができない可能性がとてつもなく高い，ということを言っておく．　今回は Ubuntu15.04(日本語 remix) を入れることにした．

### 8.3.1　デュアルブート

まずはセオリーである (?) デュアルブートでの失敗報告．

とにかくインストールには成功したが，何をしても Windows が起動する事態に．このバグより，boot 関連が問題 (?) と判断．

1. パーティション

efi パーティションを Ubuntu 専用に作成．

[ブートローダをインストールするデバイス] をシステムパーティションや efi パーティションに設定したがどれも失敗 .

Ubuntu に割り当てるパーティションの確保を Ubuntu USB live mode で行ったが変化なし．

1. **BIOS 設定**

   もちろん secure boot は無効に．UEFI boot と Legacy boot どちらに設定しても Windows が起動した．

   boot manager 上には Ubuntu が表示されるが，選択しても Windows が起動した．

2. **Windows で boot Manager を弄る (最大の失敗)**

   『EasyBCD』を使い，明示的に Ubuntu のブートローダを追加した．PC を起動すると黒背景の boot 画面が現れ，Ubuntu の項目が出現するものの ブートエラーが起こった．

   もういっそのこと『EasyBCD』で Windows を boot 候補から Windows を除外してみたが，失敗．起動時に “Insert boot device...” のような警告が表示された．さらに (当たり前だが)Windows すらもブートできなくなり，この警告画面から抜け出せなくなる．

### 8.3.2 詰む前にやってみたかったこと

- [ブートローダをインストールするデバイス] を Windows の efi パーティションに選択し，ブートローダを上書きする．

- Ubuntu USB live mode で boot を最適化する．

  Live モードの Ubuntu 上の tarminal から以下のコマンドを打ち込み，notePC の boot を最適化するソフトを使用する．

```
sudo add-apt-repository ppa:yannubuntu/boot-repair
sudo apt-get update
sudo apt-get install boot-repair
```

  Boot Repair を起動し，“おすすめの修復” をする．

参考: http://kowaimononantenai.blogspot.jp/2014/02/windows81ubuntuuefi.html

### 8.3.3 シングルブート

デュアルブートができないなら Ubuntu 専用 notePC にしてしまえ，という考えから SSD を完全にフォーマットして Ubuntu をインストール．Ubuntu が起動できなかったため，ここで初めてマザーボードが悪いことに気が付いた．

ここでリカバリー USB を使用し工場出荷状態に戻そうとしたが，SSD フォーマット後なので Windows 復元は失敗した．

## 8.4　もし失敗したら

バックアップとして作成したクローン SSD を，失敗した SSD と入れ替えれば以前のように notePC は動くようになる．　入れ替えるには notePC の裏蓋を開けなければならない．事前に自分の持っている型の notePC が個人で SSD 換装できる構造であるかネットで調べることが大切である．　例えば，Panasonic 製レッツノートシリーズは側面にスライド式の電源ボタンがあるため，特別な開き方が要求される．

1. notePC(今回は KIRA) を裏返しにし，ネジを外す．このとき，各部位を締めているネジの種類を覚えておく．その後，裏蓋を外す．(すんなり外れる)
2. SSD を留めているネジを外すと，SSD が斜めに突き出るので SSD を抜く．
3. クローン SSD を先ほど SSD が刺さっていたところに斜めに挿す．その後，ネジで留める．
4. 裏蓋を閉じる．この際，notePC を裏向きの状態からひっくり返してしまうと主に電源ボタンが反応しなくなってしまう可能性があるので，うまくはまらなくても向きは変えないようにすること．私はここでしくじってマザーボードが壊れた．(￥35000)

## 8.5　Windows でがんばる

Windows プリインストールモデルに Windows しかインストールできない問題については，「寡占しているのでは?」という声も多く，メーカーによってはそういった制約を BIOS 画面で外すことができる　ら　し　い　．

なんだかんだ Windows でも Linux のようなことができるので，冒険したくない人はおとなしく Windows で環境構築するのも一つの選択肢．我が子を育てるようで意外と楽しい．

- **tarminal エミュレータの導入**

  有名どころなら『Cygwin』．豊富なパッケージが吉と出るか凶とでるかは使う人次第．ネットでの情報も多い．

  Windows の文字コードと cygwin 上の文字コードが噛み合わないことが多く，設定がめんどくさいので Cygwin に vim か emacs を入れると良い．(Cygwin 上で完結させると良い)

  ほかにも，若干動作が重い Cygwin の代用品として『MobaXterm』が注目されている．使用感は Cygwin とほぼ同じなので，誰にでもとっつきやすいのが特徴．

- **git の導入**

  最近になって『git for Windows』がリリースされ，git が Windows でも使えるように


図 8.1 PC 内部

なった．GUI と CUI，どちらにも対応しているので，初めての人にもわかりやすい．CUI は MINGW ベースのターミナルエミュレータ．

- 仮想 **OS**

  Windows の操作感がそもそも嫌な人，Windows で環境構築が面倒な人は『VMware』で仮想環境を作るほうが早い．

「仮想だからいつでも消せる」という安心感は，全 Windows ユーザの心を和ませることだろう．

私が使っているのは『openSUSE』．アップデート方式は “ZYpper” と特殊だが，いつで

も『openSUSE Tumbleweed』というローリングリリース版にすることができるため，Linux ディストリビューションならではの楽しさを味わえる，“オイシイ OS” だと思う．

# 第 9 章

# あなたの知らない超絶 Scala プログラミングの世界

cohalz

細かい言語仕様だとかめちゃ便利構文だとかめちゃ便利じゃない構文とか集めました．コップ本などに載ってない 2.10 以降の機能紹介も．

## 9.1 初めに

Scala で仕事をしている cohalz です．

今回はぼくが仕事で使っているような最近の Scala 構文や，変わった言語仕様などを少し集めました．

早速本編に入りましょう．

## 9.2 オートタプリング

引数が Any である関数に対して，コンマで複数の値を渡すと自動で括弧が補完されタプルとなります．標準では標準出力がちょうど Any を受け取るようになっているため試すことができます．

```
scala> println(1,2)
(1,2)

scala> println((1,2))
(1,2)
```

複数の値を同時に出力して試したい時に便利です．

この機能は-Ywarn-adapted-args オプションを付けることで無効にすることができます．

これだけだと大した事ないし，オプションなんか付けずにそのままでもよいのでは？

となるかもしれませんが実は厄介です．

例として，List を Set に変換したいとき，つい以下のコードを書いてしまうかもしれません．

```
List(1, 2, 3).toSet()
```

この実行結果はなんと false: Boolean です．

```
List(1, 2, 3).toSet()
warning: there was one deprecation warning; re-run with -
   deprecation
 for details
res0: Boolean = false
```

確認のために-deprecation を付けて実行してみます．

```
scala> List(1, 2, 3).toSet()
<console>:8: warning: Adaptation of argument list by inserting
   ()
 has been deprecated: this is unlikely to be what you want.
        signature: GenSetLike.apply(elem: A): Boolean
  given arguments: <none>
 after adaptation: GenSetLike((): Unit)
              List(1, 2, 3).toSet()
                                  ^
res0: Boolean = false
```

何故このような結果になるのかというと，まず，toSet は括弧なしで宣言されておりこれは

```
List(1, 2, 3).toSet.apply()
```

と扱われます．

更にオートタプリングにより，

```
List(1, 2, 3).toSet.apply(())
```

となります．

Set の apply メソッドは List でいう contains として定義されているため，要素数 0 のタプル，つまり Unit と比較して false を返します．

しかし toSet したものを一旦変数にして，apply を呼ぶと正しい動作をします．

```
scala> val s = List(1, 2, 3).toSet
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

```
scala> s()
<console>:9: error: not enough arguments for method apply:
(elem: Int)Boolean in trait GenSetLike.
Unspecified value parameter elem.
              s()
                ^
scala> s(())
<console>:9: error: type mismatch;
 found   : Unit
 required: Int
              s(())
```

謎.

## 9.3 構造的部分型

継承関係にない複数のクラスに関して，共通したフィールド名やメソッド名を持つクラスを引数に持つメソッドを作成したい場合，構造的部分型を用いて実現することができます.

Ruby などではダックタイピングと呼ばれる手法ですが，Scala は静的型付き言語のためそのまま書けるわけではありません.

```
def test(testClass: {def display: Unit}): Unit = {
  testClass.display
}
```

また,「特定のメソッドを持つ型」を宣言して使用することもできます.

```
type HasDisplay = { def display: Unit }

def test(testClass: HasDisplay): Unit = {
  testClass.display
}
```

List や Option など Generic な型に対して用いたい場合は，型制約の部分に書きます.

```
def test[T <: {def startDate: Date}](list: List[T]): Unit = {
  list.foreach(l => println(l.startDate))
}
```

DB を扱う際には，継承関係にない複数のテーブルで同じ名前のカラムを作成することはよくあるため，非常に便利です．

ちなみに，Scala では def と val の名前空間が同じ関係か，フィールドに対して構造的部分型を用いる際も def を使います．val ではコンパイルエラーになります．

## 9.4 implicit class

Scala2.10 で追加された implicit class を使用することで，既存クラスの機能拡張を行うことができます．

```
implicit class StringHogeHugaPiyo(self: String) {
  def display(): Unit = println(self)
}
```

このように self の部分に拡張したいクラスのインスタンスを引数に取るような形で取り，メソッドを書くだけで既存クラスの機能拡張をすることができます．

String のメソッドを呼ぶかのように書くだけで使うことができます．

```
"foobar".display
```

StringHogeHugaPiyo の名前は参照することはないので自由に名前をつけて構いません．

ただし，Ruby のオープンクラスとは違い，既存メソッドの改変をすることはできません．以下の例では indexOf: Char => Int を右から探索するよう書き換えています．

```
implicit class StringHogeHugaPiyo(self: String) {
  def indexOf(chr: Char): Int = {
    for(i <- (self.length -1) to 0 by -1) {
      if(self(i) == chr) return 0 - (self.length - i)
    }
    1
  }
}

//書き換わっていれば'c'は右から4番目なので-3が表示されるはず
println("abcdef".indexOf('c'))
```

しかし，この実行結果は変わらず 2 が出力されます．

implicit(暗黙の) という意味の通り，メソッドが見つからないという不都合があった場合に，都合のいいクラスを探してくれるという感じです．

上記の例では，indexOf: Char => Int は既に String クラスにあるため，StringHogeHugaPiyo クラスを見に行ってくれないという具合です．
イメージとしては Ruby の method_missing に近いと思います．

以上の理由から，既存クラスを機能拡張したい際には，既存クラスにない名前でメソッドを作成するか，素直に継承して override をするかにしましょう．ちなみに，Scala の String は final 宣言されており override はできません．final 宣言されたクラスに対して機能拡張する際は implicit class を用いましょう．

最後に，implicit class は実行時のオーバーヘッドを伴います．この実行時のオーバーヘッドをなくすため，AnyVal クラスから継承するというテクニックを組み合わせることがあります．

```
implicit class StringHogeHugaPiyo(self: String) extends AnyVal
```

時間の都合上解説はできませんが，AnyVal を継承した場合のデメリットももちろんあります．
状況をよく考えて選択するようにしましょう．

## 9.5 文字列フォーマット

文字列内の変数埋め込みや文字列整形が Scala 2.10 で追加されました．
ダブルクオートの prefix に s を追加し，変数名の前に$を付けることで展開することができます．

```
scala> val str = "world"
str: String = world

scala> s"hello $str"
res0: String = hello world
```

$の後に {} で囲むことで，任意の式を展開することができます．

```
scala> println(s"1 + 1 = ${1 + 1}")
1 + 1 = 2
```

また，s ではなく，f を指定すれば printf のようなフォーマット指定が出来ます．${}%format という形で書きます．

```
scala> f"${0.1}%.3f"
res7: String = 0.100
```

別にこんなフォーマット指定を使わず printf でいいのでは？と思うかもしれません．
しかし，残念ながら Scala の printf は型安全ではありません．
以下の様なコードでコンパイルが通ってしまいます．

```
val n = "hello"
printf("%d\n",n)
```

このコードは実行時に java.util.IllegalFormatConversionException を吐き出します.

```
scala> val n = "hello"
n: String = hello

scala> printf("%d\n",n)
java.util.IllegalFormatConversionException: d != java.lang.
    String
  at java.util.Formatter$FormatSpecifier.failConversion(
      Formatter.java:4045)
  at java.util.Formatter$FormatSpecifier.printInteger(Formatter.
      java:2748)
  at java.util.Formatter$FormatSpecifier.print(Formatter.java
      :2702)
  at java.util.Formatter.format(Formatter.java:2488)
  at java.util.Formatter.format(Formatter.java:2423)
  at java.lang.String.format(String.java:2792)
  at scala.collection.immutable.StringLike$class.format(
      StringLike.scala:31
 8)
  at scala.collection.immutable.StringOps.format(StringOps.scala
      :30)
  at scala.Predef$.printf(Predef.scala:311)
  ... 33 elided
```

それに対し，f を使ったフォーマットはコンパイルエラーになります.

```
scala> val n = "hello"
n: String = hello

scala> println(f"$n%3d")
error: type mismatch;
 found   : Double
 required: Int
```

```
    println(f"$n%3d")
                ^
one error found
```

Scala の型システムを享受するため，printf は使用せず format 指定子を用いるようにしましょう．

さらに，s や f のようなこの prefix は独自に定義することができます．

試しに，どんな文字列を渡しても Hello, World!という文字列が返ってくるようにしましょう．

```
implicit class HelloWorldHoge(val sc: StringContext) extends
  AnyVal {
    def h(args: Any): String = "Hello, World!"
  }

scala> h"yo"
res0: String = Hello, World!
```

StringContext クラスに対して，implicit class でメソッドを定義することで使用することができます．

また，前述のとおりオーバーヘッドの関係上，AnyVal から継承させるようにしましょう．

上記の s や f のように，変数展開するときの解釈を変えることもできます．

StringContext と args により「文字列の部分」と「変数の部分」という部分ごとに分けることができます．

sc.parts の部分が文字列 (String*)，args に渡された変数 (Any*) が入ります．

例として，文字列のリストと変数のリストのタプルを返すようにしてみましょう．

```
implicit class InterpolationHoge(val sc: StringContext) extends
  AnyVal {
    def t(args: Any*) = (sc.parts.toList, args.toList)
  }

scala> t"abc${1}def${1 + 2}ghi${1 + 2}"
res0: (List[String], List[Any]) =
  (List(abc, def, ghi, ""),List(1, 2, 3))
```

文字列の部分と変数の部分に分離することができました．手順を追っていきましょう．

まず，prefix の t というものは StringContext には存在しないため，InterpolationHoge に渡されます．

以下のように展開されます．(AnyVal を継承しているため実際は少し異なります)

```
new InterpolationHoge( new StringContext("abc", "def", "ghi",
    "") ).t(1, 2, 1 + 2)
```

InterpolationHoge の sc に StringContext の中身が入ります．

```
val sc = new StringContext("abc", "def", "ghi", "")
//sc.partsはString*型， つまりArray[String]になっている
```

t の部分には変数部分が入ります．

```
t(1, 2, 1 + 2) //この引数はAny*型， つまりArray[Any]になる
```

そして両方とも最後に toList をして見やすい出力にしていると言う具合です．(Scala における Array の toString はそのまま出力するには向かないという都合上)

## 9.6　名前渡し

名前渡しは Google で検索すると Scala ばかりヒットします．

“渡し” という名前の通り，関数の引数のことですがこれは何を意味するのでしょう？

まず先に，名前渡しの文法は以下のようになります．

```
def test(s: => String): Unit
```

これは何なんでしょう？

Scala では単に引数を書く場合，以下のように書きました．

```
def test(s: String): Unit
```

対して，関数を引数に取る場合以下のように書きました．

```
def test(s: String => String): Unit
```

3 つを見比べてみると，名前渡しは関数と値の間のように見えます．

試してみましょう．

第一引数が普通の Boolean，第二引数が名前渡しの Int です．

```
def test(flag: Boolean, n: => Int): Unit = {
  if(flag) println(n)
  else println("falseだよ")
}
```

普通に値を渡してみます.

```
scala> test(true, 1)
1

scala> test(false, 1)
falseだよ
```

次は単なる値ではなく関数を噛んだ値を渡してみます.

=> Int という見た目の通りに Int を返す関数なら何でも良いです.

今回は以下の「a を出力して 1 を返す関数」で試してみます.

```
def testMethod(): Int ={
  println("a")
  1
}
```

試してみましょう.

```
scala> test(true, testMethod)
a
1
```

今度は false で試します.

```
scala> test(false, testMethod)
falseだよ
```

今度は a が出力されませんでした.

これは名前渡しの遅延性によるもので，名前渡しで渡された引数は必要になるまで評価されません.

Scala の関数は正格評価のため，本来であれば関数に入る前に評価されてしまいます.

このような正格評価を避けるために，名前渡しを用いて遅延評価をし，無駄な評価を避けたいという場合に便利です.

しかし，名前渡しを採用する理由は遅延評価性だけではありません.

遅延評価性だけであれば，関数の形で渡して実行すれば良いはずです.

```
def test(flag: Boolean, f: () => Int): Unit = {
  if(flag) println(f())
  else println("falseだよ")  //ここではfは評価されない
}
```

```
scala> test(true, testMethod)
a
1

scala> test(false, testMethod)
falseだよ
```

やりました！

しかし，これでは単に値を渡したい時に困ります．

```
scala> test(false, 1)
<console>:9: error: type mismatch;
 found   : Int(1)
 required: () => Int
              test(false, 1)
                          ^
```

解決方法としては関数の形で書いてあげることです．

```
scala> test(true, () => 1)
1
```

これで解決しました...がダサいですね．

名前渡しが test(false, 1) と書けたことを考えると非常にダサいです．

このように遅延評価ももちろんですが，それよりもダサさを排除するために，名前渡しの構文が用意されました．

実際に名前渡しが使われている例としては，Option の getOrElse が有名です．

getOrElse ではオブジェクトが Some(nanika) であった場合には，デフォルト値は必要が無いため名前渡しで無駄な評価を避けています．

おそらく以下の様なコードが書かれています．

```
getOrElse[B >: A](default: ⇒ B): B = {
  this match {
    case Some(value) => value
    case _ => default //ここに入らなければdefaultは評価されない
  }
}
```

また，Boolean における&&と || の短絡評価 (ショートサーキット) も名前渡しと同じ動きをします．(名前渡しで実装されているかはわかりません)

## 9.7 最後に

Scala は非常に複雑な言語です．

大学入学当初から Scala を読み書きしていますが，未だに implicit や型クラスなどは理解が追いついていない状況です．

ですが，Scala は Java っぽさもあれば Ruby っぽさもあり，また Haskell のような感覚もあり非常に面白い言語でもあります．

この記事を読んで Scala および Scala の言語仕様について興味を持ってもらえたら嬉しいです．

# あとがき

PMOB 所属の方々に一言お願いしてみました.

- cohalz

来月沖縄に行きます.

- iammazo4545

『キレイナプログラムガカンセイスルトキモチイイヨォ www』

- Komachi_Z

21 歳、学生です。(自己紹介) ちまたでは Komachi おじさんとして親しまれてます。が、マジで 15Rx 生が別世代に見えてくるのでおじさんです。(諦観) 鳩祭おわったら “(21)” もまもなく終了なんだよなぁ...。就活？なんのことか n うわなにをするやめ r (ry 履修計画はしっかり組んで、どうぞ。間違っても人間形成科目ばっかりとるのは NG でも、大学にいるうちに人生迷っておくのがいいんじゃない？ (留年のススメ) 清覧ありがとうございました。Twitter: @Komachi_Z

- matsub.rk

生きものです

- sanex_now

食事と排泄行為に生きがいを感じる

- zodiac

TDU15RD のゾディアックです。TDU の人ならツイッター (@zodi_G12) で知っているかもしれませんね。よろしくです。

- shamison

締め切りは絶対守ろうな (ニッコリ

- meno

NOR ちゃん描いたら部誌データ全部飛びました

- うにの抜け殻

書きたいことが書ききれませんでした。本当は環境が用意できれば pspsdk についてを書きたかったのですが・・・

著者： PMOB
表紙絵： めの, matsub
発行： 2015 年 11 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