mikutter

の薄い本 vol.8

テオクレズガーデン

ーー"TL"の向こうは、別世界。

本当に、そのとおりだ。

ておくれアイコンで統一された TL。

ておくれ情緒あふれる、ﾌｫﾛｲｰたちの顔。

「うん。あたしね、"ておくれ"なんだよ」

目次

表紙イラスト

@soramame_bscl	霧雨魔理沙 トトゥーリア・ヘルモルト
@shijin_cmpb	ラピス・メルクリウス・フレイア

うぎゅぎゅ

前書きにかえて　@brsywe

1.　待ち人

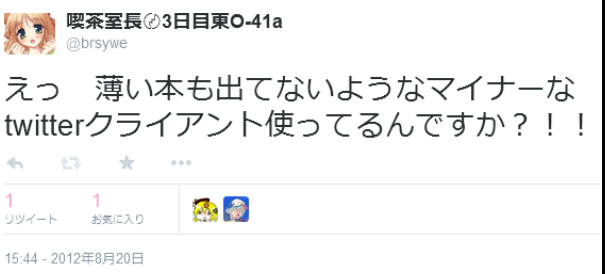
　まず、過去を振り返っていきたい。決して逃避しているのではない。

2012 年末　mikutter の薄い本 vol.3
『人類は手遅れました。』

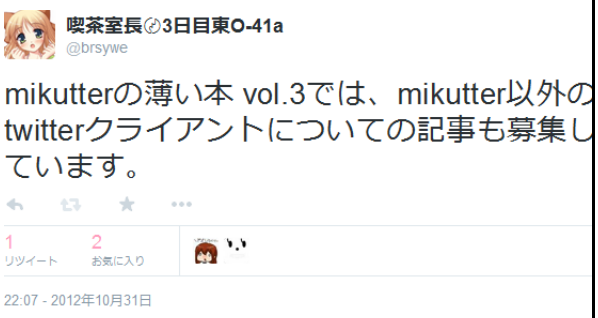
「*おれ*

今一瞬でも ShootingStar の機能が思い浮かんだやつは　@haru067 にふぁぼ爆撃して中指飛ばした上で ShootingStar の薄い本でも書いてろ。」

そして私。
vol.2 までを出した 2012 年 8 月 20 日。



次に 2012 年 10 月 31 日。



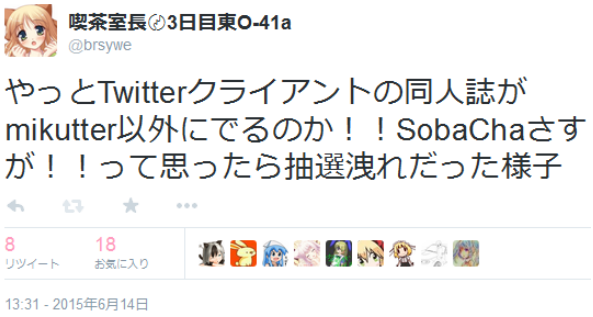
当時から mikutter 以外の Twitter クライアントの同人誌を切望していたことがわかる。2012 年末にはサークル「スマサー」発行の『季刊スマサー vol.5』に於いてメガネケエス・ShootingStar　Twitter 作者対談記事がある。

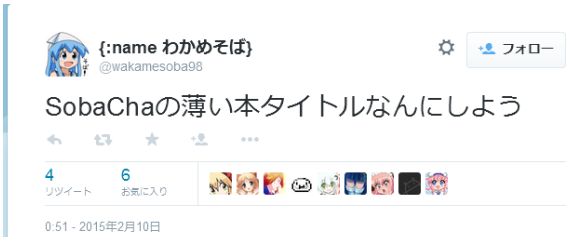
　この後もメガネケエスのグッズ類は出ていたが、他には観測されていない。mikutter 以外の Twitter クライアントの同人誌はいつになるのか。全ﾂｲｯﾀﾗｰは待ちに待っていた。

2.　時機

　ある夏の日のことである。業務をやっつけて定時で退社し、Circle.ms で mikutter の薄い本制作委員会のコミックマーケット 88 当選を確認した。

　その 2 日後。





https://twitter.com/wakamesoba98/status/564813799555727361

　結果は残念だが、おそらく初めての mikutter 以外の Twitter クライアント同人誌である。今夏は落選したため薄い本を出さないとのことであるが、今後に期待したい。出すなら買いに行きます。よろしくお願いします。

3. 今回の mikutter の薄い本

1：ワードアートの巨匠 @misodengaku によるワードアート作品を表紙に用いたほか、平和的な暴力によって勝ち取った記事である。いまや(mikutter ユーザーで)知らない人はいない画像サービス totori.dip.jp の詳説である。
2：いつも d250g2 している @ahiru3net。もっと d250g2 するためには。
3：@moguno 。いつもやばいプラグインつくってるひとのやばい新作。
4：@toshi_a 。病床に伏して執筆したといわれる遺稿である。mikutter の薄い本原稿の取立て中に取立て班が京都御所地中に埋まっていた原稿を発掘し掲載することとなった。今回は Minecraft にイワシが追加されたにもかかわらず馬刺しが追加されなかったことと、イワシのおいしい焼き方についての考察である。

イワシがいっぱいだあ…

ちょっとだけもらっていこうかな

4. 魔女庭

明乃ちゃんうぎゅぎゅぎゅ

ワードアート:@misodengaku
イラスト:@shijin_cmpb

totori.dip.jp 80 年の歩み

@misodengaku

mikutter 3.2 で突如として画像サービスとして対応した totoridipjp、謎の多い過去を振り返ります

totori.dip.jp のはじまり

始めは自分が Linux サーバーの学習用に建てたサーバーで、学校の課題研究に使用するため適当な DDNS サービスでドメイン名を割り振っただけのものでした。ちょうどその頃友人から PS3 とトトリのアトリエを借りて沼に腰辺りまで浸かり始めたところだったためこのドメイン名となりました。

「コンテンツはよくわからないがとりあえずトップの画像だけはかわいい」とまで言われるゲームのスクリーンショット画像ですが、設立当時はまだトトリのアトリエ Plus が存在しなかったため、PC 用のディスプレイに表示した PS3 の画面をカメラで撮影するという暴力的な手段で得られた画像を使用していました。

totori.dip.jp の今

課題研究が終了すると同時に totori.dip.jp をホスティングしていたサーバーの HDD がぶっ壊れ、どうせ電源付けっぱなしになっているメインの PC とサーバー機を 2 台動かすのも電気的に無駄だろうということで仮想環境でのホスティングに移行しました。

Windows Server 2012 (後に R2) 上の Hyper-V で動く Gentoo Linux という普通に考えたらまともじゃないっぽい構成ではありますが、学習用ということで敢えてわけのわからん環境で運用してきました。

一般的に知られている歴史はおそらくこの辺りまでですが、田端 bot や superfuckjp bot などの比較的重要度の高いサービスが走っている上に mikutter が対応してくるなど、当初の目的と異なりだんだんと稼働率の維持が重要となってきたため、楽園追放 BD を見るために BD ドライブのホットスワップに挑戦したところ誤って蹴りを入れてサーバーが停止するなどの従来のような雑な運用が許されないような雰囲気に (少なくとも自分の中では) なってきました。

そう考えていた 2015 年 3 月、某社でアルバイトをすることが決まり VPS が借りられるようになったので、ついに VPS への移行を決めて環境を新しく構築しなおすことにしました。さらにトップページも Django による自作 CMS で生成するようになったなど、見えないところでではありますが割と大きな変化がありました。

totoridipjp API と mikutter

執筆時点での mikutter では対応しておらず、唯一 totoridipjp4j が対応しているのみという具合ですが、今後トップ画像の URL に簡単かつ確実に追従できる（ようにするつもり）ので個人的には利用を推奨しています。

totori.dip.jp 各コンテンツの解説

「意味不明」「書いた奴の気が知れない」などの意見をよく目にする各種コンテンツですが、ここで簡単に成り立ちを紹介しておこうと思います。

- 平和

  県教育委員会にケンカを売ったことが原因で夏休み明けたらいなくなっていた母校の教師を称えるページです。

- ピザカウンター

  以前大量のピザをカウントする必要性が出たとき、WindowsPhone 用にアプリを開発して使用したのですが、友人に「同様のアプリを Android 向けに作って欲しい」と頼まれ、Java なんか書く気がしないので JavaScript で書いたものです。

- 松屋計算機

  「昼飯にx円出すならその金で松屋の牛めし並を$(x \div 280)$回食った方がマシ」（当時）という計算を簡素化するために作りましたが、プレミアム牛めしへの対応がめんどくさくなって放置されています。

- トトリちゃん色

  トトリちゃんの髪の色です。

- 知るかバカ

  必要だった

- Gentoo の ISO

  知り合いに Gentoo のインストールを勧める際、公開されている最新のイメージに含まれているカーネルがバグ持ちで、ちょうど手元にあった古いバージョンの ISO を受け渡すために置いたもの。

- ネットワーク環境

  メインコンテンツです。2015 年 5 月頃に RTX1000 から RTX1210 への置き換えが行われました。

  toshi_a 氏にいただいたトトリちゃんフィギュアの股の下をパケットが通っていることが売りだったのですが、VPS へ移行してからは事実上股の下を通過することは不可能になってしまっており、対応を模索中です。

totori.dip.jp のこれから

これからもクソもないようなサイトなのでとりあえず現状維持と気分で追加していく感じになると思いますがよろしくお願いします。

あひる焼くなプラグイン

あひる(@ahiru3net)

1. はじめに

mikutter に出会ってから 3 年、気付いたらておくれてた。

初めまして、あひる(@ahiru3net)です。

最初は Ubuntu で apt-get して入れて使ってみていましたが、今では成行き的に OSX と Windows でしか使っていません。作者に消し炭にされるのも時間の問題です。

今回薄い本で書かないかとお誘いを某ておくれご本尊からいただいたので書かせていただきました。

この記事では「あひる焼き」とは何なのか、あひる焼くなプラグインはどのようにやばいのか書いていこうと思います。ちなみにこのプラグインは私のメインマシンである自作 PC の Windows 上の mikutter で稼働しています。

2. あひる焼きとは

あひる焼きとは何ぞやという方も多いと思います。知らなかった方は正常です。すでにあひる焼きとつぶやいたことがある人はておくれです。

あひる焼きがつぶやきだされたのは 2014 年のカーネル/VM 探検隊@北陸という勉強会の後からになります。私はその時 Eject コマンドユーザー会の一員として LT をするためにカーネル/VM に参加していました。その際に、発表の中で GPU を焼いて直すというタイトルだけ見るといかにもやばそうなやばい発表をしたすしさんがいました。

その流れからか、

GPU を焼く

↓

あひるがなんかやらかしたぞ！

↓

一緒に焼いてしまえ？ ！

↓

あひる焼き爆誕

という流れができ、以後「あひる焼き」がされるようになりました。

3. あひる焼くなプラグイン

最初はやらかすと焼かれていたのでが、次第に何もしていないのに焼かれるようになりました。

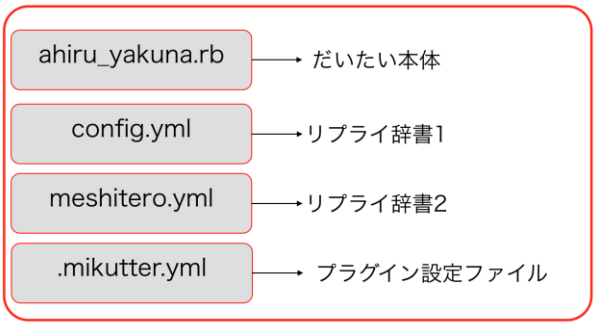
そこで反撃すべく作ったのがあひる焼くなプラグインです。

このプラグインは「あひる焼き」もしくは「ahiruyaki」に反応してリプライを送るプラグインです。自身の TL にどちらかの単語が現れた時点で反応します。

@toshi_a 氏のイベントうぉっちプラグインを併用して反応する単語をリアルタイムにエゴサしているのでたいていの場合、ツイートした直後に発火します。

プラグインの構成は以下の通りです。

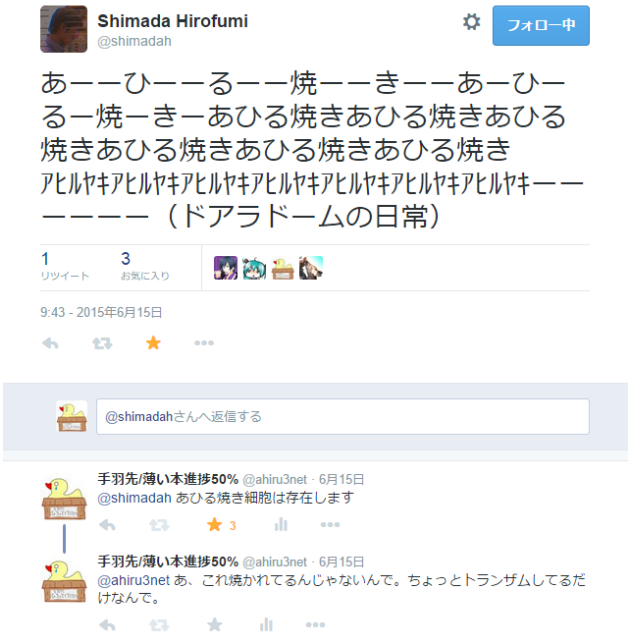
プラグインの構成



mikutter 起動時に yml ファイルから辞書をロードします。

時間によって辞書にあらかじめ登録した反撃文もしくは飯テロ画像の URL を登録します。ただし飯テロモードに関してはいまいち安定しておらず、設定した時間に切り替わったり切り替わらなかったりします。要検証です。

また、自身のツイートにも反応し、リプライ辞書にもあひる焼きがいくつか含まれているので連鎖という現象が起こります。(下図参照)



@shimadah さんのあひる焼きツイートに対して bot があひる焼きを含んだリプライをし、さらにそれに対し bot が反応するといった感じです。

詳しくはソースコードを GitHub で管理しているので詳しくはそちらを参照してください。
(https://github.com/Na0ki/ahiru_yakuna)

この記事を書いている途中で気づいたのですが、このプラグインを使用できるのは機能的に私だけだということです。もちろん反応する単語などを変えれば別のプラグインとなりますが、基本的にあひる焼きに反応するのは私ぐらいではないでしょうか。他のプラグインの多くはたいていユーザーを選ばず、インストールしたユーザーの mikutter ライフをておくれさせることが可能ですが、このプラグインはそれができないのが大きな欠点の一つです。ただ使っているパーツは応用が効くと思います。(後述しますがこのプラグインは他のプラグインを~~パクって~~参考にしました)

これから初めて作ろうとしている人の助けに少しでもなればうれしい限りです。

4. プラグインを稼働した結果

反撃のために作ったプラグインでしたが作った結果…

```
__人人人人人人人人人__
＞　焼く人が増えた　＜
￣Y^Y^Y^Y^Y^Y^Y^Y￣
```

ついでにあひる焼きガチ勢やら定期的に焼いてくる人、さらには別の bot に学習されて焼かれるなどいろいろと面白い状態です。

日々こんな感じです。(次頁参照)

あひる焼きをして鬱憤を晴らしたり、連鎖引いて楽しんだりしているようなので、これはこれでありなのでしょうか。

5. プラグインを作るにあたって

実は mikutter で使われている Ruby という言語はこのプラグインを作るまで使ったことがなく、さらには mikutter のプラグインを書くというのも初めてで最初は全くわかりませんでした。完全に初心者です。

とにかくわからないので@penguin2716 さんのプラグイン(https://gist.github.com/penguin2716/4518448)をコピペして TL の任意のワードを拾う機能を実現しました。

さらには、@cosmo__さんや@firstspring1845 さんからの超いい感じなプルリクにより、RT には反応しない、リプライを辞書に格納などを実現し、@toshi_a さんのイベントうぉっちプラグインによりエゴサタブへのリアルタイム反応を成し遂げました。

いろいろな知識をありがとうございます。

また、実験ツイートをしていただいたあひる焼き勢のみなさんもありがとうございます。完全に廃れるまでは細々とアップデートとメンテを続けていこうと思います。

6. 最後に

現在持ち合わせている飯テロ画像が質・量ともによろしくないので、この画像使っていいよという方お待ちしております。

焼くな焼くなも焼けのうち。

今日も一日焼かれるぞい。

サンプルコードにコンテンツ力を求めるのは間違っているだろうか？

もぐの(@moguno)

=begin
　ちす！もぐの＠絶賛デスマ中です。マジ死にそうです。何もかも忘れて有馬温泉とかに行きたいです。
でもなんとか今回も mikutter の薄い本に寄稿したい・・・何かいい方法は・・・そうだ！

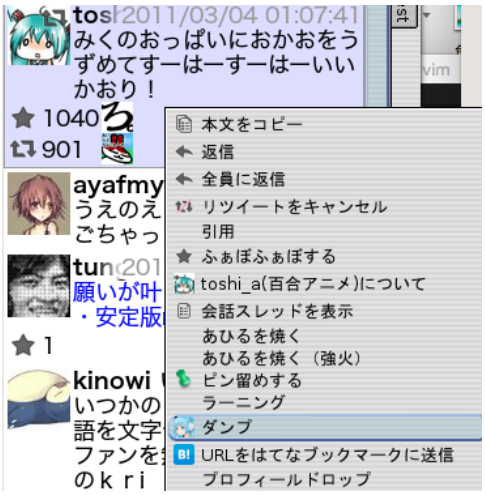
「ソースコードはプログラマーの最高のコミュニケーションツール！」

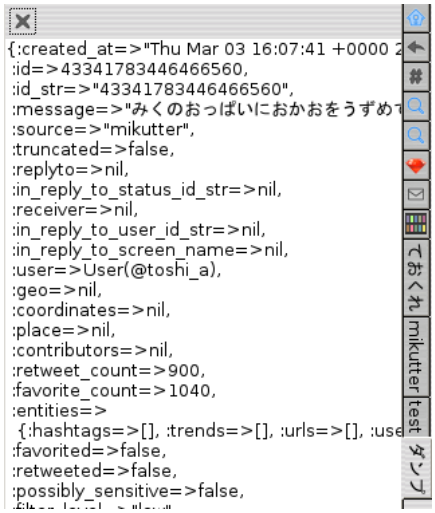
例えば拙作プラグインのソースコードにコメントをしこたま付ければ、プラグイン作成講座として成立するんじゃないか？そのまま動く生きたサンプルコードの詳説とか。お、格好つくんじゃない？

よし、これ以上は考えちゃダメだ明日が来ちゃう。それではスタートです！
=end

1. mikutter コマンドを作ってみよう

　mikutter コマンドを実装すると、右クリックメニューにアイテムを追加することができます。
ここでは、選択したメッセージの内部情報をダンプする mikutter-command-dump.rb を使って、そのへん解説してみようと思います。





・mikutter-command-dump.rb

```
# coding: UTF-8

# mikutter プラグインは必ず Plugin.create()から始まります。
# いわゆる「おまじない」ですね。
# 引数に渡すシンボルは、ファイル名及び.mikutter.yml で指定したものに合わせてください。
Plugin.create(:"mikutter-command-dump") {

  # require や関数の定義は Plugin.create()のブロック内で行いましょう。
  # このブロックの中は他のプラグインやコアの処理に影響を与えない安全地帯なのです。
  require "pp"

  # まずは command()を使ってコマンドを定義していきます。
  # これで右クリックメニューに項目が追加されて、ショートカットキーが割当可能になります。
  command(
          # スラッグ（なめくじ？）
          # 他のコマンドと被らなさそうな、いい感じの名前をシンボルで渡します。
          :dump_message,

          # コマンドの名称
          # 文字列を_()で囲むと国際化対応の対象になるので、
          # 誰かが言語ファイルを書いてくれるかもしれません。
```

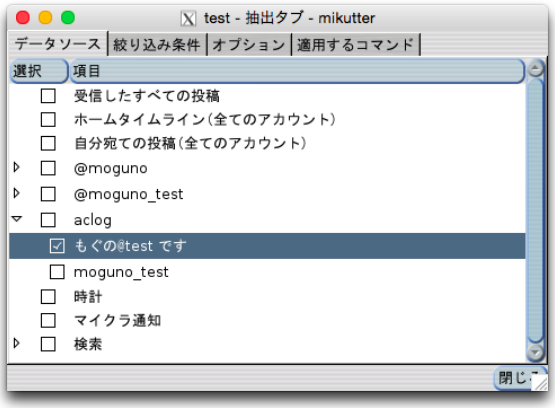
```
        :name => _("ダンプ"),

        # コマンドが有効になる条件
        # ここでは「メッセージが選択されていること」を条件にしています。
        # メッセージが選択されていない場合は右クリックメニューにも表示されず、
        # ショートカットキーも発動しません。
        :condition => lambda { |opt| Plugin::Command[:HasMessage] },

        # メニューに表示する？
        # false にするとメニューに表示されなくなるので、ショートカットキー呼び出し
        # 専用のコマンドになります。
        # また、拙作 mikutter-extract-fire-command で使う「自動処理では便利だけど
        # 単体では役に立たないコマンド」は、false にしておくとエレガントです。
        :visible => true,

        # アイコン
        # メニューに表示するアイコンのファイル名を指定します。
        # Skin.get()で画像ファイル名を取ってくるようにしておけば、サードパーティの
        # アイコンセットの恩恵に与れるかもしれません。
        # （薄い本 Vol.7 参照）
        # 実は URL も指定可能なので、どこかの Web サイトの favicon.ico などを指定する
        # のもお手軽です。
        :icon => Skin.get("icon.png"),

        # ロール
        # コマンドを画面上のどの部品にアサインするかを決めます。
        # :timeline :タイムライン
        # :tab          :タブ
        # :pane         :ペイン
        # :postbox      :投稿ボックス
        # :profile    :プロフィール
        # :profiletab   :プロフィール内のタブ
        # :window       :ウインドウ
        # なお、ロールをウインドウにするとステータスバー部分にボタンが出現します。
        :role => :timeline) { |opt|

    # ここからいよいよコマンドの実装に入っていきます。

    # 「ダンプ」タブがすでに表示されているか調べる。
    # mikutter が管理してる GUI 部品のリストは、そのクラスの cuscaded()で取得可能です。
    if !Plugin::GUI::Tab.cuscaded.has_key?(:dump)

      # メッセージのダンプを表示する GTK ウィジェット（TextView）を作ります。
      @debug_textview = Gtk::TextView.new.show_all
      debug_textview = @debug_textview

      # 「ダンプ」タブを作っていきます
      tab(:dump, _("ダンプ")) {
        # タブにしいた・・・「タブを閉じる」ボタンを表示させます。
        set_deletable(true)

        # このタブは再起動時には消滅して欲しいので temporary_tab()を呼んでおきます。
        temporary_tab

        # 今回は独自に GTK ウィジェットを置くので nativewidget()を使います。
        # なお、普通のタイムラインを作りたいときは timeline()を呼びます。
        nativewidget(debug_textview)
      }
    end

    # タイムラインロールの場合、command()のブロック引数 opt に選択中のメッセージが
    # 入るのでそれを読みやすく整形してテキストボックスに表示します。
    @debug_textview.buffer.text = opt.messages[0].to_hash.pretty_inspect

    # 最後に「ダンプ」タブをアクティブにして処理終了です。
    Plugin::GUI::Tab.instance(:dump).active!
  }
}
```

いかがだったでしょう？案外いい感じ・・・いい感じ？まぁいいや。続きます。

2. 抽出タブのデータソースを作ってみよう

mikutter の抽出タブは地味なれど超ステキな機能で、複数のデータソース(情報源)を束ねたり、必要な情報のみをフィルタリングしたりして、自分だけの TL を構築することができます。さらに拙作 mikutter-extract-fire-command を使えば、抽出タブを使って Bot 的な処理を行うことなんかもできちゃいます。
今後、データソースが充実していけば mikutter がどんどん便利になっていくこと請け合いです。

ここでは日本初のふぁぼられツイート管理サービス(？)aclog から、自分の最新のふぁぼられツイートを流すデータソースについて解説してみます。



・mikutter-datasource-aclog.rb

```
#coding: utf-8

# いつものようにおまじないでスタートです。
Plugin.create(:"mikutter-datasource-aclog") {
  require "yaml"

  counters = {}

  # プラグインは mikutter で発生した出来事(イベント)に反応して処理を行うことができます。
  # mikutter のイベント処理は「on_イベント名」と言う名前のメソッドのブロックとして記述
  # します。
  # on_boot はプラグインが有効になるときに発生するイベントです。
  on_boot { |service|

    # アカウントごとにデータソース更新カウンタを生成します。
    Service.each { |s|
      counters[s] = gen_counter
    }
  }

  # on_period は 1 分周期で発生するイベントです。
  on_period { |service|

    # gen_counter は call()するたびに+1 した値が帰ってくる面白機構です。
    count = counters[service].call

    # 更新タイミングが来たら、データソースを更新します。
    # UserConfig は mikutter の設定が格納されているハッシュです。
    if count >= UserConfig[:retrieve_interval_search]
      counters[service] = gen_counter
      refresh(service.user_obj)
    end
  }


  # 今回作るデータソースを mikutter に登録します。
  # データソースの登録には extract_datasources フィルタを用います。
  # フィルタとは mikutter のもう一つのイベント機構です。
  # フィルタは複数のプラグインから情報を得たいときや、他のプラグインが返した情報を
  # 書き換えて mikutter の挙動を変更したいときに便利です。
  # フィルタ処理は「filter_イベント名」メソッドのブロックとして記述します。
  filter_extract_datasources { |datasources|
```

```
    # アカウントごとに、"aclog/ユーザ名"と言うデータソースを追加登録します。
    # ブロック変数 datasources には、他のプラグインが登録したデータソースの情報が
    #入っています。誤って内容を壊さないようにしつつ、自分のデータソースを追加しましょう。
    Service.each { |service|
      datasources[:"aclog_#{service.user_obj[:idname]}"] = "aclog/#{service.user_obj[:name]}"
    }

    # 次のプラグインに情報を伝えます。
    [datasources]
  }


  # データソースに aclog から取得したメッセージを流します。
  def refresh(user)
    notice "datasource_aclog refresh:#{user.to_s}"

    # aclog から最近ふぁぼられたツイートのメッセージ ID を取得します。
    url = "http://aclog.koba789.com/api/tweets/user_timeline.yaml?screen_name=#{user[:idname]}"

    data = open(url) { |fp|
      YAML.load(fp.read)
    }

    data.each { |omoshiro_tweet|
      # mikutter 独自の Twitter ライブラリである mikutwitter を使って、aclog から得た
      # メッセージ ID からメッセージ本文を取ってきます。
      # mikutwitter は非同期に Twitter に API を発行するため、結果を待つことなく次の
      # 処理に進んで行くことに注意です。
      # 問い合わせ結果は next()に渡したブロックで得ることができます。
      ((Service.primary.twitter/"statuses/show/#{omoshiro_tweet["id"]}").message).next { |res|

        # 得られたメッセージをデータソースに流します。
        # イベントは mikutter コアだけでなく、プラグインからも発生させることができます。
        # :extract_receive_message イベントを発生させて、mikutter コアにデータソースへの
        # メッセージ追加を通知します。
        Plugin.call(:extract_receive_message, :"aclog_#{user[:idname]}", Messages.new([res]))
      }
    }
  end

  # これでデータソースプラグインの完成です。お疲れ様でした。
}
```

よし、数時間で 4 ページの原稿が書けてしまいました。作戦成功です。

mikutter はプラグインでその機能を飛躍的に向上させることができます。さらに、自分でプラグインを書けば無限の可能性がそこに広がります。今回掲載した以外にも mikutter には面白い API やテクニックが沢山あります。プラグインのサンプルは mikutter 本体のソースコードにゴロゴロしてるので、ぜひ自分だけの mikutter を創って見てください。

3. おまけ:としぁさんのプラグインを読んでみよう。

mikutter 作者のとしぁ(@toshi_a)さんも、いくつかのプラグインを公開されています。
そのプラグインはどれも美しく、そして面白く。なにより mikutter フレームワークの特性を最大限に活かしたものになっています。

ここでは、ておくれ必携の GUI をゴニョるプラグイン display_requirements.rb を題材に、僭越ながらその超絶テクニックを解説してみたいと思います。

・display_requirements.rb

もはや芸術ですね。ジョン・ケージが 1952 年に作曲した「4 分 33 秒」に通ずるものがあります。

このファイル。「UNICODE 不可視文字」や「ておくれにしか見えないコード」なんてちゃちなもんじゃ断じてなく、本当に 0 バイトの空ファイルがプラグインとして成立してしまうのです！

ほら、mikutter プラグインって面白そうでしょ？

リファクタリング: mikutter エディション

@toshi_a

1. はじめに

mikutter プラグインは mikutter の全てを制御することができ、今や簡単な mikutter プラグインを自分用に書く人も珍しくありません。 多くのプラグインは小さいものですが、中には複雑にならざるを得ないような要件もあり、コードが長く読みづらいものになってしまうこともあります。 mikutter 標準プラグインには、実際に長編プラグインがいくつも含まれています。それらのプラグインを参考にしながら、プラグインが肥大化しても見通しの良さを保つ方法を考えてみます。

まだプラグインの書き方を知らない人は、 mikutter の薄い本 vol.1 の toshi_a の記事 *Writing Mikutter Plugin* を読んでみましょう。

http://toshia.github.io/writing-mikutter-plugin/

この記事は、 *Writing Mikutter Plugin* の内容を理解していることを前提としています。

1.1. 本稿の見方

event	mikutter コアやプラグインで定義される変数、クラス、イベント等です
Array	Ruby や外部ライブラリ、例示したコードで提供されるクラス等です
コマンド	mikutter 用語です
`code`	サンプルコードなどの一部です

2. 複数のプラグインに分ける

そもそも一般に、コードは短いほうが理解しやすいものです。 あるプラグインを綺麗にしようと思うとき、どうしてもそのプラグインをそのまま綺麗にしようと思ってしまうものです。 しかし実際には、長く読みづらいプラグインというのは複数の問題を解決しようとしているからそうなっていることが多いのです。 解決する問題ごとにプラグインを分けてしまえば、一つ一つのプラグインは短くなり、ひとつの目的しか持たなくなるので理解しやすくなります。

分け方は簡単で、ふたつプラグインを作ればいいのです。しかし、もちろん同じプラグインだったものを分けると、それらが連携する必要が出てくると思います。

2.1. 実例: sound

mikutter 0.2.2 から、alsa プラグインは **sound** と **alsa** に分かれました。従来の **alsa** プラグインは「効果音の再生指示を受け取る」という仕事と「効果音を再生する」という仕事をこなしていたため、コードが煩雑で応用が効かなかったからです。

実際の **sound** プラグインを見てみましょう。

```
Plugin.create :sound do
  Sound = Struct.new(:slug, :name, :play)

  # サウンドDSL
  defdsl :defsound do |slug, name, &play|
      filter_sound_servers do |servers|
        [servers + [Sound.new(slug, name, play)]] end end

  on_play_sound do |filename|
      use_sound_server = UserConfig[:sound_server]
      Plugin.filtering(:sound_servers, []).first.each{ |value|
        if not(use_sound_server) or use_sound_server == value.slug
            value.play.call(filename)
            break end } end

  settings _("サウンド") do
      select _("サウンドの再生方法"), :sound_server do
        Plugin.filtering(:sound_servers, []).first.each{ |value|
            option value.slug, value.name } end end

end
```

こちらは **alsa** プラグインです。

```
Plugin.create :alsa do

  aplay_exist = command_exist?('aplay')

  defsound :alsa, "ALSA" do |filename|
      bg_system("aplay","-q", filename) if FileTest.exist?(filename) and
aplay_exist end

end
```

2.2. defdsl

defdsl [1] DSL メソッドで **defsound** という DSL メソッドが定義されています。**defsound** を一度呼び出せば、 **sound_servers** フィルタの結果に自身をサウンドサーバのひとつとして含めるようになります。 **sound** プラグインは **sound_servers** フィルタの結果を受け取って、例えば設定画面に登録されたサウンドサーバを表示しているのです。 **defsound** を呼び出しているのは **alsa** プラグインですね。ブロック内はこの内容を見て分かる通り、サウンドを再生すべき時に、ファイル名を引数に呼ばれます。

1 **defdsl** について詳しく知りたい人は、mikutter の薄い本 vol.4 の記事「Plugin の新機能」を読みましょう

2.3. フィルタを使った連携

defsound を呼ぶと **sound_servers** フィルタを定義するようになっており、 **sound** プラグインは必要に応じて **sound_servers** フィルタを呼んでいます。なぜ **sound** プラグインが配列でこれらを保持せず、フィルタにするのでしょうか。

こうする理由は、 **defdsl** のブロックのスコープは、定義した DSL を呼び出した **Plugin** のインスタンスになるので、 **sound** プラグイン内部を触れないからです。また、プラグインがアンロードされたらフィルタも取り外されるので、毎回プラグインの存在確認をしなくていいというメリットもあります。

もともと一つだったプラグインが通信するのにフィルタはとても便利です。更にフィルタを使えば、比較的自然に複数のプラグインと通信できるようになります。例えば今回の例では、ALSA 以外のサウンドの再生方法を追加できるようになっています。実例を見てみましょう。

2.3.1. サードパーティプラグインとの連携の例: mikutter-windows

moguno さんが開発している mikutter-windows[2] は、mikutter を Windows で使用するときに遭遇するさまざまな問題を解決するものです。問題の中に、Windows には ALSA がないのでサウンドが再生できないというものがあるようです。mikutter-windows では、まさに **sound** プラグインと連携するサードパーティプラグインを提供するという方法でこの問題を解決しています。実際に見てみましょう[3]。

```
Plugin.create :mikutter_windows do
  defsound :win32, "Windows" do |filename|
      SND_ASYNC = 0x0001
      playsound = Win32API.new('winmm', 'PlaySound', 'ppl', 'i')
      playsound.call(filename.encode(Encoding.default_external), nil, SND_ASYNC)
  end
end
```

Windows API を呼んでサウンドを再生するようになっていますね。このプラグインを入れたら **sound** プラグインが提供する設定画面に Windows が追加されます。これを選べば、以後サウンドの再生は全て **mikutter-windows** が行います[4]。

2.4. イベントを利用した作業の依頼

プラグインを複数に分けると、メソッド呼び出しができなくて困ることが多いと思います。そもそも呼び出す相手が複数になる場合があるのでフィルタを使うのが得策ですが、中には戻り値が必要ない処理もあります。 **sound** プラグインの例では、サウンドの再生を指示する処理は、戻り値は要りませんね。こういった場合は、単にイベントを使えばいいの

2 https://github.com/moguno/mikutter-windows

3 本稿で取り扱いやすいように一部編集しています。元のソースコードは https://github.com/moguno/mikutter-windows/blob/master/mikutter-windows.rb

4 実際にはこのプラグイン、初めて入れた時に win32 を使用するように設定を書き換えるようなので、選ぶ必要はないです

です。

```
Plugin.call :play_sound, "path/to/sound.wav"
```

このようなコードで発生した **play_sound** イベントは、 **sound** プラグインが受け取り、適切な方法でサウンドを再生します。もちろんフィルタで引数を書き換えられるので、全部の効果音を変えてしまうこともできます（ん、これはエイプリルフールネタにいいかも…）。

2.5. 依存関係

プラグインを分けた場合、それらは密接に関係するのは当然です。すると、 **sound** がないと **alsa** はロードできないなど、明確な親子関係が出来てしまう場合があります。 こういう関係ができてしまった場合は依存関係を書いておくことで、ロード順を制御できるうえ、依存しているプラグインがない場合にユーザがそれを知ることが出来ます。

あるプラグインが別のプラグインに依存しているとはどういうことか、定義を明確にしておきます。 専ら本稿で「依存している」というのは、それがないとクラッシュする（プラグインをロードできない）ということです。 そのプラグインがないと機能が提供できないという場合は、依存していると言いたいところですが、依存関係とは言わないことにします。

また **sound** プラグインを例に考えてみましょう。サウンド再生をさせる目的で **notify** プラグインが **play_sound** イベントを発生させても、イベントを受け取る **sound** プラグインがいなければ、サウンドは鳴りません。これを依存関係と言ってしまうと、例えば **sound** が何らかの通知を発生させるために **notify** プラグインに受け取ってもらうつもりでイベントを発生させるような仕様になった場合、循環依存になってしまいます。mikutter プラグインの循環依存は、ロード順が確定できないため認められません。

一方、 **alsa** プラグインは **defsound** DSL を使用していますが、これは **sound** プラグインが提供する DSL なので、 **sound** がロードされていないと NoMethodError 例外が発生してクラッシュします。 現在の mikutter の仕様では、プラグインは特に依存関係がなければアルファベット順にロードされます。つまり、 **alsa** は **sound** より先にロードされるので、 **alsa** をロードした時に **defsound** がないというエラーが発生するでしょう。 **defdsl** が絡んでくると依存しているということにせざるを得ません。これは **defdsl** の重要なデメリットの一つです。諦めて、 **alsa** プラグインの .mikutter.yml には以下のように依存関係を明記しましょう。

```
slug: :alsa
depends:
  mikutter: '0.2'
  plugin: [sound]
version: '1.0'
author: toshi_a
name: ALSA
description: aplay コマンドを用いてサウンドを再生する
```

depends: キーの plugin: の中ですね。配列で、依存するプラグインの slug を羅列します。

「依存関係は、ロード順がシャッフルされてしまったら問題がある、という場合に設定する」

と考えるとわかりやすいかもしれません。イベントを受け取ってもらいたい程度の話しなら、送信側、受信側どちらが先にロードされようが関係ありませんが、 **defdsl** はそうはいきません。もちろん依存関係が設定されていなくても適切な順番でロードされることもあるでしょうが、ここらへんの理屈を踏まえて適切に設定するべきです。

3. 処理を分ける

一般に、長いコードは関数などを使って処理を分けて可読性を上げるというテクニックはよく使われます。mikutter プラグインで処理を分ける方法について考えてみます。

3.1. メソッドではなくイベントを

「メソッドを定義する方法は？」と聞かれることがありますが、やりたい内容を聞くとイベントでもいい場合が結構あります。前節で、戻り値がない場合はイベントと書きましたが、イベントを発生させたプラグイン自身が受け取ることもできます。標準プラグインでもよくやっていることです。

ただし、イベントはメソッド呼び出しと違ってあとで実行されることと、別のプラグインが定義するイベント名と衝突してしまう危険があることには注意する必要があります。mikutter プラグインではそういうイベントの名前が被らないように、 **(プラグイン slug)_(イベント名)** という名前をつける習慣があります。

3.2. それでもメソッドを使う

イベントのデメリットが許容できない場合や、フィルタの柔軟性がいらないなど、どうしてもメソッドを定義しなければならない場合もあります。

```
Plugin.create :method do
  on_boot do
      load_file
  end

  def load_file
      ...
  end
end
```

単に def を使って定義します。注意しなければいけないのは、 def のスコープはレキシカルではないということです。つまりプラグイン DSL 内部で宣言したローカル変数を参照することができないということです。そういう場合は、わざわざインスタンス変数を使わなければなりません。しかし **Plugin** 自体が当然インスタンス変数をいくつか定義して使っているので、これと被らないように注意が必要です。

4. プラグインから処理を追い出す

前節の「処理を分ける」と似ていますが、こちらはプラグインの外に処理を出してしまうというアプローチです。プラグインとして分けることは適切ではないが、ファイルを分けたいという場合にも使います。

4.1. プラグインに与えられた暗黙のモジュール

mikutter プラグインは、 **Plugin::(slug をキャメルケースにしたもの)** という名前のモジュールを作って、その中に任意の定数やメソッドを定義して良いことになっています。

```
module Plugin::ModuleExample
  SETTING_FILE = "path/to/setting.yml"
end

Plugin.create :module_example do
  setting = nil

  on_boot do
      setting = YAML.load(Plugin::ModuleExample::SETTING_FILE)
  end
end
```

この例では設定ファイルのパスを定数 `Plugin::ModuleExample::SETTING_FILE` に書いています。どの言語でもよく見るテクニックです。

以下のように書いてはいけないのでしょうか。

```
Plugin.create :module_example do
  SETTING_FILE = "path/to/setting.yml"
  setting = nil

  on_boot do
      setting = YAML.load(SETTING_FILE)
  end
end
```

短く簡潔になりました。しかしこれは典型的なアンチパターンです。上のプラグインと同時に、以下のプラグインを入れるとどうなるでしょうか。

```
require 'rexml/document'

Plugin.create :module_example_2 do
  SETTING_FILE = "path/to/setting.xml"
  setting = nil

  on_boot do
      setting = REXML::Document.new(open(SETTING_FILE))
  end
end
```

こちらのプラグインは、設定ファイルが XML 形式になっていますね。実はこの 2 つのプラグインは衝突しています。 **Plugin.create** の中で定数を宣言すると、トップレベルに定義されます。ということは、この 2 つのプラグインは同じスコープに `SETTING_FILE` を

定義するので、定数の多重定義になります。 Ruby 2.2 では定数の再定義を行うと警告が表示されるが上書きは成功するようです。ということは、XML ファイルに YAML が書かれたり、逆のことが起こりうるということです。プラグイン同士がお互いの設定を破壊しあう、非常に危険な状態です。

しかしこういった不変の値は定数として定義しておきたいもので、そういった要求に安全に答えるために「Plugin::ModuleExample」のようなモジュールを好きに使って良いということになっているのです。

4.2. モジュール空間からプラグインにアクセスする

プラグインのモジュール内に処理を書く必要性がある場面というのはどうしても出てきます。例えば mikutter では、 Gtk::Widget のサブクラスを継承した独自のウィジェットをいくつも使っていますが、クラスを定義するならこの「プラグインに割り当てられたモジュール」を使うのが適切でしょう。 ここで問題になってくるのは、外部クラスからプラグインへのコールバックはどうするのか、ということです。多くの場合は戻り値でよいでしょうが、例えば「ボタンがクリックされたらプラグインが何かしたい」という場合は、外部クラスからイベントなどを発生させる必要があります。

```
module Plugin::FavFav
  class SoundButton < Gtk::Button
      def initialize(*a)
        super
        signal_connect :clicked do
            Plugin.call :play_sound, "/mikutter/core/skin/data/sounds/favo.wav"
            true
        end
      end
  end
end

Plugin.create :favfav do
  settings "ふぁぼふぁぼ" do
      add Plugin::FavFav::SoundButton.new("再生")
  end
end
```

設定画面にボタンを出現させ、押すと favo.wav を再生するボタン
`Plugin::FavFav::SoundButton` を定義し使用しています。

mikutter であればどこでも **Plugin.call** を呼ぶことでイベントを発生させることができるので、あとはプラグイン側がそれを受け取って任意の処理をすればよいのです。例では **play_sound** イベントを発生させているので、 **sound** プラグインがこれを受け取って効果音を再生するでしょう。イベントなのですから、受け取り先がどのプラグインでも構わないわけです。

同様に **Plugin.filtering** を用いてフィルタを利用することもできます。

4.3. モジュールを別のファイルに書く

前の例のようなコードだと、改修を重ねるごとにモジュールが肥大化していくのが目に見えています。

mikutter プラグインのコードは、 *slug* と同じ名前のファイル名のファイルに書かないといけないのでした。他の名前のファイルがあっても無視されるだけなので、プラグインディレクトリにはどんな ruby ファイルを置いても問題ありません。それを読み込めばいいのです。

```
# sound_button.rb

module Plugin::FavFav
  class SoundButton < Gtk::Button
      def initialize(*a)
        super
        signal_connect :clicked do
            Plugin.call :play_sound, "/mikutter/core/skin/data/sounds/favo.wav"
            true
        end
      end
  end
end
```

```
# favfav.rb

require_relative "sound_button.rb"

Plugin.create :favfav do
  settings "ふぁぼふぁぼ" do
      add Plugin::FavFav::SoundButton.new("再生")
  end
end
```

新たに sound_button.rb というファイルを作成して、
`Plugin::FavFav::SoundButton` の定義を移動しました。 当然このままでは無視されるので、 require_relative で作成したファイルを読み込みます。

モジュールは再オープンできるので、クラス毎にファイルを分けることもできますし、定数定義だけ別のファイルにするといった使い方もできます。

5. まとめ

プラグインのコードの見通しを良くする方法と、注意点について説明しました。

プラグインを分けてしまうとインストールの手間が増えてしまうなどデメリットもありますが、 **sound** プラグインのように、メタプラグインにするという発想を得られるといったメリットもあります。サウンドサーバをハードコーディングして選べるようにするのではなく、別のプラグインと連携して増やすことができるというのは、いかにも mikutter らしいプラグインだと思います。

複雑なプラグインを書くと、他のプラグインと衝突し、処理や値を意図しないものに上書きされる危険性があることも確認しました。 特に定数のスコープについては、知っていないと小規模なプラグインでも衝突が起こりかねないので注意が必要です。

今までは、間口を広げるために小さなプラグインを書くためのテクニックを書くことが多かったと思いますが、今回はそれが肥大化してきた時に具体的にどういう課題があり、どのようにクリアしていくのかを具体的に説明しました。皆さんの mikutter ハックの一助となれば幸いです。

絵 @soramame_bscl

卓上のぼりに

＜＜あとがき＞＞

そんなものはなかった

――次号予告と寄稿者募集――

『ふぁぼが捗りすぎてヤバい。』

原稿提出期限：12 月 2 日

頒布予定：コミックマーケット 89,こみトレ 27

問合せ先：@brsywe , @ch_print

奥付 発行日：2015 年９月６日　　PDF 版初版 発行：mikutter の薄い本制作委員会 発行者：　@brsywe 西端の放送局内喫茶室長 連絡先：　brsywe　　@　　hotmail.co.jp
ご意見・ご感想は Amazon ギフト券のメッセージ欄にどうぞ。 mikutter の薄い本制作委員会ウェブページ http://kohinata.sumomo.ne.jp/mikutter.html

mikutter の薄い本制作委員会では、Amazonギフト券による金銭面の支援を受け付けております。

もし、あなたがこの薄い本を読んで、何かしら満足感を得られたなら、ギフト券を貰えるとその満足感を誰かと共有できるかも。