vol.10

テオクレズコンプレックス

2016 summer

OSC KYOTO ／ C90

mikutter の薄い本

なんでそんなにておくれなの？

目次

なまえ	ミクちゃんの好きなところ	題	頁
@brsywe	は？？？ミクはああ見えて結構あるんやぞ！！！！	前書「まいてつで忙しい」	3
@Akkiesoft	ツインテのつけ根の香り	ﾚｨﾃﾞｨｽｴﾝﾄﾞｼﾞｪﾝﾄｩﾒﾝ。ｳｪｩｶﾑﾄｩｻﾞｼｨﾝｶｧﾝｾｪﾝ。	4
@shibafu528	フラットデザイン	mruby is miku-ruby	5
@osa_k	おっぱい	魔法の一筆で DM を描いて	10
@toshi_a	いいかおり	餓死地獄が来るぞ	20
@brsywe	-	後書	28

イラスト

@soramame_bscl	ミクさんマジ天使	表紙イラスト
@shijin_cmpb	かわいい	前書・後書イラスト

まいてつで忙しい

@brsywe

1. 言い訳

今回は記念すべき 10 巻目となることから、いろいろと考えていたけれども、地震の影響でそれどころではなかった。

2. 今回の mikutter の薄い本

@Akkiesoft 。薄い本史上最も読みにくい記事。

@shibafu528。ｼｨﾝｶｱﾝｾｪﾝ プラグインや、mikutter プラグイン史上最も政治的である「野々村プラグイン」などを Android 用クライアントで動かすすごいやつ。

@osa_k。野々村プラグインを DM に悪用した悪質な事例。

@toshi_a。政治的な㋮屋である播磨屋本店を痛く気に入ってしまった。

3. 右の絵

@shijin_cmpb

ﾚｲﾃﾞｨｽｴﾝﾄﾞｼﾞｪﾝﾄｩﾒﾝ。ｳｪｳｶﾑﾄｩｻﾞｼｨﾝｶｧﾝｾｪﾝ。

ｱｯｷｨ（@Akkiesoft）

ﾃﾞｨｽｲｽﾞｻﾞ ﾐｸｯﾀｧ ｷﾁｶﾞｲﾌﾟﾗｸﾞｲﾝ ｱｰﾃｨｺｰ。ﾌﾟﾘｰｽﾞ ｹｱﾌﾙ ﾕｱ ﾃｵｸﾚ。

ﾃﾞｨｽｲｽﾞﾄｫｶｲﾄﾞｳｼｨﾝｶｧﾝｱﾝﾄﾞｻｧﾝﾖｫｼｨﾝｶｧﾝｾｪﾝｵｫﾝﾘｨ。

ﾄｩ ﾗｲﾄﾞｻﾞ ｼｨﾝｶｧﾝｾｪﾝ、ﾕｰ ｼｭﾄﾞｩ ｾﾚｸﾄ ｻﾞ ﾀｲﾌﾟ ｵﾌﾞ ﾄﾚｲﾝ ｱﾝﾄﾞ ﾃﾞｨｽﾃｨﾈｲｼｮﾝ。ﾋｱｰｲｽﾞﾘｽﾄ。

```
train = ["ﾉｫｿﾞｫﾐｨ","ﾋｨｶｧﾘｨ","ｺﾀﾞｧﾏｧ"]
station = ["ﾄｰｷｮｰ","ﾐｨｼｨﾏ","ｼｨｽﾞｳｵｫｶ","ﾊｧﾏｧﾏｧﾂ","ﾅｧｺﾞｫﾔ","ｼｨﾝｵｫｻｧｶ","ﾋｨﾒｪｼﾞ","ｵｫｶｧﾔｧﾏ","ﾐｨﾊｧﾗ","ﾋｨﾛｫｼｨﾏ","ﾊｧｶｧﾀ"]
```

ｾﾚｸﾄ ﾀﾞｲｱﾛｸﾞ

ﾃﾞｨｽﾌﾟﾗｸﾞｲﾝ ﾕｳｽﾞｨﾝｸﾞ ｻﾞ ｼﾞｨﾃｨｹｪ ﾀﾞｲｱﾛｸﾞ。ﾕｰｷｬﾝ ｾﾚｸﾄ ｻﾞ ﾄﾚｲﾝ ｲｰｽﾞﾘｨ（図 1・図 2）。

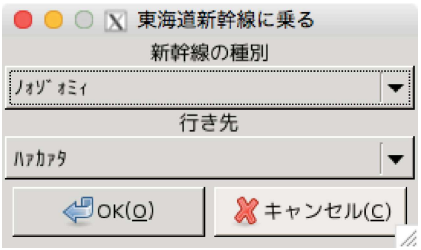


図 1　ｴｸﾞｻﾞﾝﾎﾟｩ ﾉｫｿﾞｫﾐｨｽｳﾊﾟ ｱｴｸｽﾌﾟﾚｽ ﾊﾞｳﾝﾄﾞ ﾌｫｰ ﾊｧｶｧﾀ。

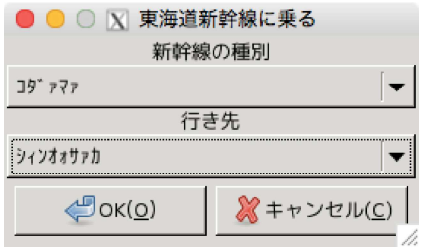


図 2　ｴｸﾞｻﾞﾝﾎﾟｩ ｺｫﾀﾞｧﾏｧｽｳﾊﾟ ｱｴｸｽﾌﾟﾚｽ ﾊﾞｳﾝﾄﾞ ﾌｫｰ ｼｨﾝｵｫｻｧｶ。

ﾘｻﾞﾙﾄｩ

ｱﾌﾀｧ ｾﾚｸﾄﾄﾚｲﾝ、ﾕｰｳｨﾙ ﾂｲｰﾄ ｻﾞｯﾄ ﾕｰ ﾗｲﾃﾞｨﾝｸﾞ ﾄｩｻﾞ ｼｨﾝｶｧﾝｾｪﾝ（図 3）。ｺﾝｸﾞﾗｯﾁｭﾚｲｼｮﾝ。ﾕｰｱｰﾃｵｸﾚ。

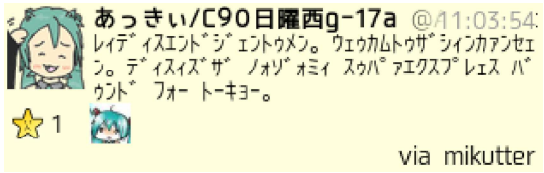


図 3　ｱｲｱﾑﾃｵｸﾚ。

ﾊｳﾄｩ ｹﾞｯﾄ ﾃﾞｨｽ ｷﾁｶﾞｲﾌﾟﾗｸﾞｲﾝ

ﾃﾞｨｽﾌﾟﾗｸﾞｲﾝ ﾊﾟﾌﾞﾘｯｼｭﾄﾞｩ ﾄｩ ｷﾞｯﾊﾌﾞ。ﾌﾟﾘｰｽﾞ ｸﾛｰﾝ ｲｯﾄ。

https://github.com/Akkiesoft/mikutter_shinkansen

ｾﾝｷｭｳﾌｫｵﾘｰﾃﾞｨﾝｸﾞ。ｻﾝｸｽ。

mruby is miku-ruby

芝生(@shibafu528)

1. はじめに

こんにちは、芝生です。何かとギリギリな日々を過ごしつつ mruby と Pluggaloid を使った自由研究を鍵垢でツイートしていたところ、発表のお誘いを受けてしまいました。せっかくの機会ですので、自分が忘れないようにするのも兼ねて文書化したいと思います。

2. mruby とは

軽量で、各種アプリケーションに組み込みやすい形で開発されている Ruby 実装です。

組み込み向けということで、主に静的ライブラリの形をとり、また CRuby における gem のような各種外部ライブラリはビルド時にすべて組み入れてしまうなど、普段私達が使っている Ruby とはだいぶ毛色の異なるものとなっています。

今回はこの子をおもちゃにします。

3. Pluggaloid とは

~~しらない？~~ 2015 年の夏くらいに公開された、mikutter の心臓です。みくったーちゃんそのものかもしれません。

要するに、mikutter のプラグイン機能を本体から分離しライブラリの形にしたものだそうです。これを使うと、お使いの Ruby を使った何かに mikutter みたいなプラグイン機能を組み込めるというわけです。

普通に使う例は、薄い本 vol.9 には作者による解説がありますので、そちらを参照いただくとイメージが掴めるかと思います。

今回重要なことは、これは mikutter の一部ですから、プラグインには互換性があるということです。

4. mruby-pluggaloid

この 2 つを組み合わせれば、mruby が使えるならどこでもプラグインが……と、そう上手くはいかないのが mruby なのです。

mruby-pluggaloid は、Pluggaloid を mruby で使えるように文法の互換性が無い部分を修正し、スレッド回りのライブラリを使用しなくても済むようにした mrbgem[1]です。といっても、主に require 行の削除や無引数 Proc が使えないことへの対処が大半です。スレッド回りについては、@toshi_a さんが pull-req する形で協力してくださいました。

ここまで作ってしまえば、mruby の build_config.rb に依存設定を書き入れて(図 1)ビルドを行えばもうそれは mikutter みたいなものです。

[1] mruby における gem のようなもの。

図 1　build_config の依存設定(抜粋)

```
conf.gem :github =>
    'shibafu528/mruby-pluggaloid'
```

ただ、この状態で mikutter のようにパワフルなプラグイン動作環境を提供することは難しいと考えられます。何故なら、mruby の標準ビルドには正規表現エンジンやファイル回りの API が含まれていないからです。

これは、mruby が組み込み等制限の強い環境でも使えることを想定して、あえて最低限の機能しかコアには持ち合わせていない故の事情です。しかし、大抵のことは Github に上がっている mrbgem で解決できます。Pluggaloid を組み込むことでプラグインエンジンを獲得できたように、探しては組み込むことで相当リッチな実行環境にすることが出来ます。

図 2 に私が実際に使用している依存設定を示します。例えば「mruby-sleep」は Sleep モジュールを提供し、sleep メソッドが使えるようになります。

「mruby-io」「mruby-require」があればファイル回りの API と require を獲得することが出来るので、ユーザプラグインを書かせるにはほぼ必須でしょう。

「mruby-reqexp-pcre」は PCRE を正規表現エンジンとして組み込む mrbgem です。本来Ruby(2.0+)では鬼雲が使用されていますが、動かしたい環境のツールチェインで

図 2　build_config.rb 実例

```
conf.gem :github =>
    'shibafu528/mruby-pluggaloid'
conf.gem :github =>
    'matsumoto-r/mruby-sleep'
conf.gem :github => 'mattn/mruby-json'
conf.gem :github => 'mattn/mruby-thread'
conf.gem :github =>
    'iij/mruby-regexp-pcre'
conf.gem :github => 'iij/mruby-io'
conf.gem :github => 'iij/mruby-require'
```

はうまくビルドが通らなかったので代わりに使用しています。詳しくは把握していませんが当然差異があったはずですので、互換性を求めるのであれば「mruby-onig-regexp」を使えることが望ましいでしょう。

「mruby-thread」は mruby でマルチスレッドを実現します。ですが、mruby の VM には GVL が無いために VM をコピーすることで何とかしているようです。そのため、欲しいものとは違うかな……？ということで今後外すかもしれません。

このように、必要性や環境の事情に応じて柔軟な構成を行えるのは mruby の魅力と言えます。もし必要なものが無かったとしても C 言語か mruby、またはその両方を使って自分で mrbgem を作ることも簡単に出来ます。

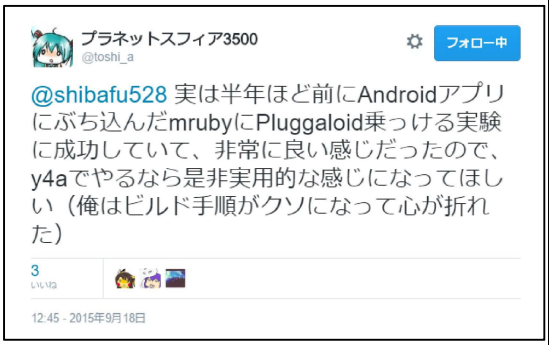
もしも私が高校生だった頃にこういった

ことができたら、Pluggaloid で拡張可能な Window Manager を作っていたかもしれません。名前は Teokure Window Manager とかどうでしょうか。私は最近 Linux をデスクトップで触っていないので今は作れませんが、そんな WM があって拡張が出来たらきっと楽しいと思います。というか、過去に似たようなことを打診されたことがあります。案外、今こそかもしれませんね。

5. ゆかミクに向けて

さて、ここからはこんなことするに至った経緯や、実践的に運用してみたお話です。

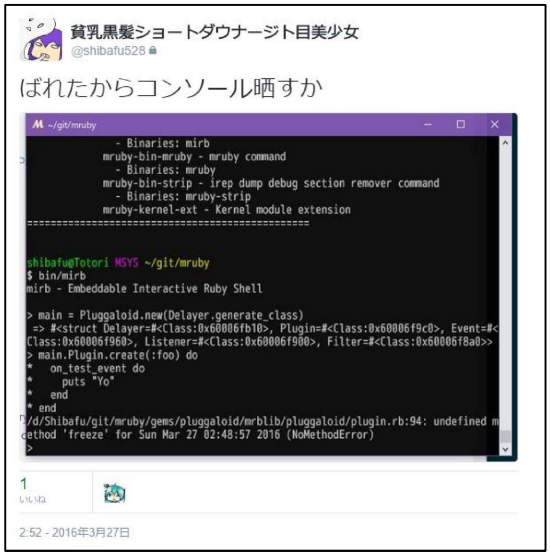
まず、~~mikutter の薄い本だというのに~~私は Android でとある Twitter クライアントを開発し公開しています。ある夏の TL で mikutter を使って遊んでいる人たちを眺めながらツイートしていたら、こんなリプライが刺さってきました。

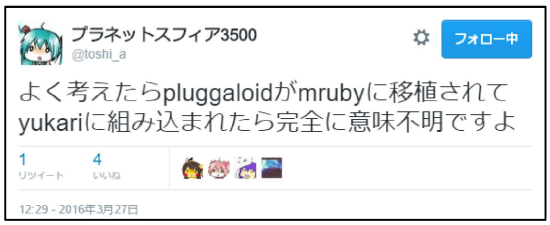


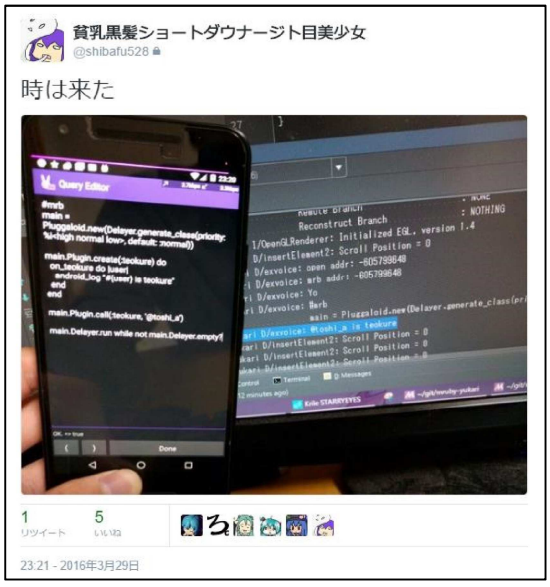
きです。Android で mruby が使えることすら知りませんでした。しかも、当時公開されたばかりの Pluggaloid が動いているとは。これはもしかして、自作クライアントに互換性のあるプラグイン実行環境を載せられるのでは……？と予感しました。しかし、モチベーションが出てこなかったため、実際に取り掛かるのは春となりました。

深夜テンションで一気に Pluggaloid を移植、その 2 日後には Android 上での動作を実現しました。

図 3　高速な展開







Android アプリ上で mruby を動かすにあたって行ったことは、mruby を Android NDK のツールチェインでビルドすること、JNI を使ってバインディングクラスを作ることです。そして、プラグイン動作環境として使うためにブートストラップとして mruby コードを少々書いてアプリの assets としてバンドル、実行時にロードしています。JNI はとてもつらく、できることならあまり書きたくありません。そのため、この辺りで何をしているかは割愛します。

ブートストラップコードには、Delayer と Pluggaloid の初期化、そしてコアプラグインのロードが書かれています。コアプラグインには、Twicca プラグインのような拡張を mruby で書けるようにする DSL と隠しコマンドを拡張する DSL のプラグイン、そして「39-yukamiku.rb」というプラグインを用意してみました。今回やったことの最大のキモとなるプラグインです。

6. ゆかミク.rb

実はこのプラグインを用意しなくても概念上はゆかミクです。Pluggaloid は mikutter のコア機能を持っていることから事実上のみくったーちゃんなので、それをゆかりさんがライブラリとして抱え込むということは、ゆかミクとしてくっついていると言えます。

ですが、この状態では mikutter の DSL はほぼありません。みくったーちゃんは知らない土地で地元なら分かりきっていることもよく分からない状態になっていると言えます。そこで、ゆかりさんが手取り足取り教えてあげる必要が出てきました。技術的に言えば、互換 DSL やクラスを定義することです。これが「39-yukamiku.rb」の役割です。

このプラグインの本体では、Message や Service を始めとした各種クラスの互換プログラムを読み込み、command や settings、activity などの DSL を定義しています。とはいえ、現時点ではまだほとんど着手できておらず、timeline と postbox ロールのコマンドが辛うじて定義できるくらいです。settings や activity が呼び出された場合、まだサポートしていない旨をコンソールに出力して無視します。

互換プログラムは現時点ではごく最低限動かすために必要なものしか入っていません。例えば、Message クラスや User クラスのとりあえず呼ばれそうなメソッドや、Service.primary.update、Environment モジュールや UserConfig クラスの定義です。UserConfig に至っては self.[]や self.[]=が一応書かれているだけで、内容は永続化されていないというやっつけぶりです。

ですが、これだけの準備で簡単なプラグインであれば動作してしまいます。例えば、

mikutter_yodobashi[2]という mikutter プラグインがあります。これは TL コマンドで、選択したツイートに対してリプライで「Welcome to ヨォドォバァシィカァメェラ」するだけのシンプルなプラグインです。これをプラグインディレクトリに配置すると、なんとあっさりと認　されます。そして、mikutter では直接ツイートされる代わりに投稿画面が開きます。これは Android というプラットフォームの手軽さと危険性を考慮し、敢えて制約をかけています。

7. うれしいこと

こんな百合錬金をして何が嬉しいか。百合だからというだけではありません。訓練された mikutter ユーザによって作られたプラグインが Web 上に数多くある一方、mikutter をモバイルでも気軽に使える時代というのはまだ完全にやってきているわけではないように思います[3][4]。TL でおもしろプラグインが盛り上がっていても、モバイルでは気軽にプラグインで遊べないのです。そこで、今 Android で動いているクライアントに実行環境が乗っかっていれば、たまには既存の資産が活かせて、少しは遊べるのです。

図 4　Yukari で動く mikutter プラグイン

本稿で取り上げたソフトウェアは、下記バージョンを基準としています。

mruby (ea03352)

mruby-pluggaloid 1.0.2 (5ed4fff)

Yukari 1.3.0.306α (lilium 160615)

[2] https://github.com/Akkiesoft/mikutter_yodobashi
[3] Debian noroot を使ったモバイル環境は操作性に難があるので完全とは言い難い。
[4] Ubuntu Phone というものがあるらしいですが時代が追いついていない。

魔法の一筆で DM を描いて

@osa_k

こんにちは。みなさんは DM 使ってますか？僕はあまり使わないです。使うときもだいたいグループ DM なので公式クライアント以外に見る術がないという残念な状況です。そんなちょっと残念な DM という機能ですが、mikutter は先日（5 月末くらい）にとても大きな変更がありました。その変更は、このテキストを書いている 6 月末時点では develop ブランチに入っており、3.5 でリリースされる予定となっています。

みなさんは、mikutter 上で DM がどういう風に見えるか知っているでしょうか。

こうですよね。左カラムにアイコンが表示され、右カラムには本文が入っています。

それが 3.5 以降は以下のようになります（プラグインの影響で少しデフォルトとは表示が異なります。具体的には `display_requirements`[1]と `mikutter-subparts-image`[2]によって、表示内容が少しリッチになっています）。

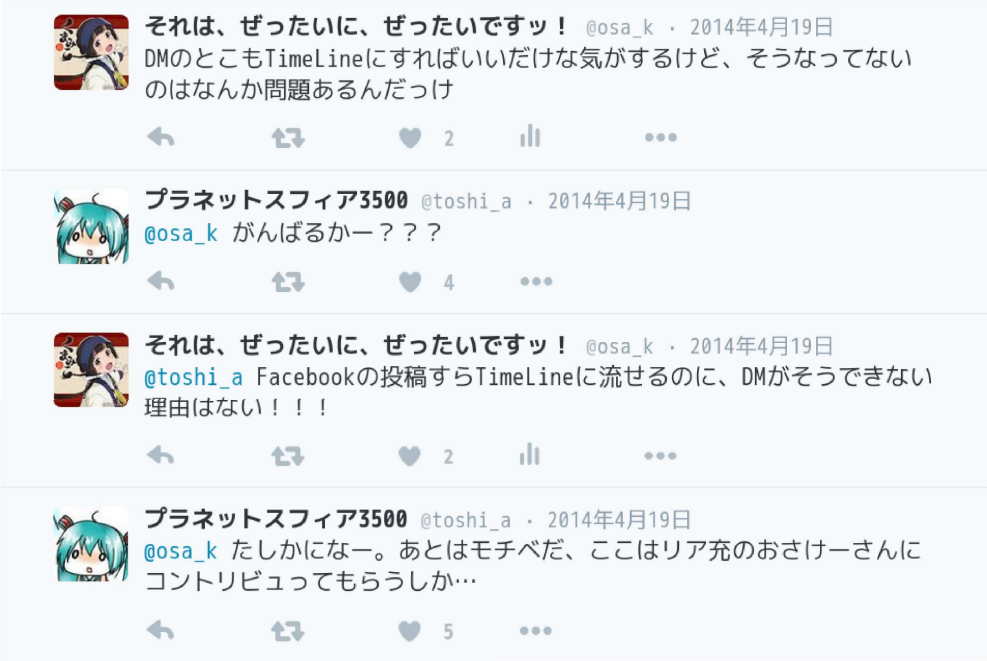
1 なんか知らないけど表示がいい感じになるプラグイン https://github.com/toshia/display_requirements

2 画像のサムネイルをインライン展開してくれるプラグイン https://github.com/moguno/mikutter-subparts-image

普通の Tweet は MiraclePainter というクラスを使って描画されているのですが、今回の変更により、DM も同じエンジンを使って描画されるようになります。結果、普通の Tweet でできるようなリッチな描画処理やコマンド実行が DM でもできるようになります。本稿では、この変更が mikutter に取り込まれるまでのあれやこれやを書いていきます。

0. 見つめ合ったら始まるよ

Twilog によれば、始まりは 2014 年の 4 月 19 日のようです。当時は就活をしており、なぜか人事が Twitter の DM でコンタクトを取ってくるというよく分からない状況だったので DM の重要度が高かったのですが、mikutter は先述したようなしょぼい見た目の DM しかなかったり、そもそも DM を送ると落ちるという問題[3]があったりと、かなり限定的なサポートしかありませんでした。そんな折、以下のような会話が交わされた結果、mikutter にコントリビュらされる（コミット権を得る）運びとなりました。こういう流れだったんですねぇ、この記事のために調べるまで完全に忘れていました。



そして、なんか雑な概要とともにチケットを作成しました[4]。Facebook の投稿というのは当時 @moguno さんが作っていた、Facebook の投稿を流すプラグイン[5]のことを指しています。このプラグインは Message オブジェクトに Facebook の投稿を無理やり詰め込むというもので、この発想はさすがもぐのさんという感じだったのですが、一方で普通の Tweet と明らかに性質の違う Facebook メッセージを共通のクラスで表現していたり、User に固定の ID を振っていたりする点で少し不満がありました。mikutter がもともと汎用のメッセージ表示ツールとして開発されていたこともあり、メッセージの取得とレンダリングの仕組みがかなり抽象化されていたため、少し頑張れば DM 専用のクラスを作って管理することも可能なんじゃないか？と思わせたことも、この発言に至った理由の 1 つです。

……そして、2 年が過ぎた……

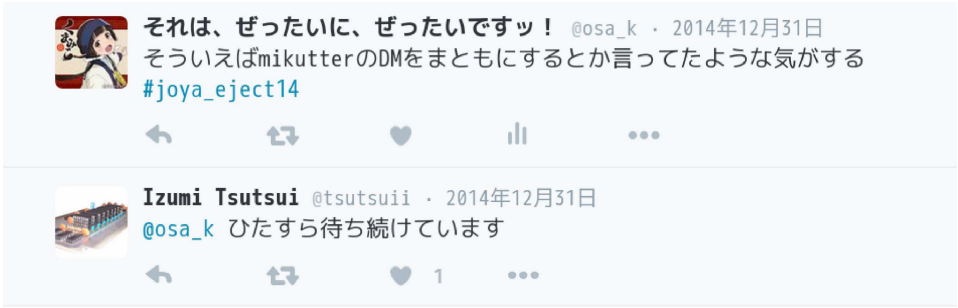
3 致命的 #643: DM を送信すると落ちる - mikutter - やること http://dev.mikutter.hachune.net/issues/643

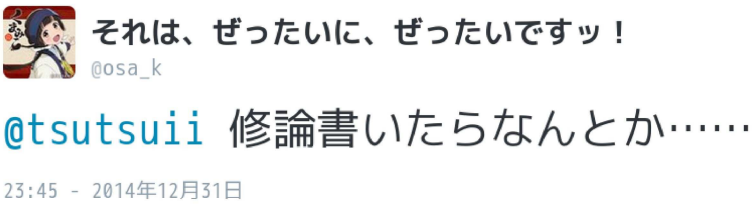
4 機能 #652: DM を MiraclePainter で表示する - mikutter - やること http://dev.mikutter.hachune.net/issues/652

5 「とうとう Facebook までておくれてしまいましたとさ。」 https://github.com/moguno/mikutter-fb

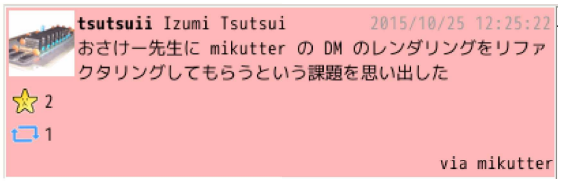
1. あふれる思いは数えきれない

方針はなんとなく決まったものの、いかにも面倒くさそうなので身辺が一段落したら取りかかろう……と思っているうちに就活も終わり、また DM をあまり使わない生活に戻ってきました。mikutter 自体はずっと使っていたので、折に触れてコードを読みつつ、なんとなくいけそうだなーとは思っていたのですが、一度後回しにし始めるとダラダラと引き伸ばしてしまう悪い癖で、全然着手する気が湧いてきません。





年末には除夜の eject を観ながら煩悩を eject してもらっていて、修論を書いたら進めると言っています。しかしそれは罠で、なんだかんだで修論終わったら遊んでいたり、他にやることが降ってきたりして、結局特に進展はありませんでした。



これは OSC2015 Tokyo / Fall で展示していたときに流れてきた Tweet です……というのは前回の記事[6]でも触れました。これとは別に何回かつついさんから Amazon テロが送られてきたりしていて、流石にやる気出そうという気持ちになっていきました。

2. だからまだ終わらないで

転機は 2016 年 4 月末、ゴールデンウイークに合わせて開催された、温泉旅館での開発合宿でした。これは会社の同期数名と一緒に伊東温泉の山喜旅館[7]で開発合宿しようぜ!という話があり、せっかくなので伸ばしに伸ばしていた mikutter の DM レンダリングをやろうと思い立ちました。いかに方針が立っているとはいえ、それなりに時間がかかる上に、作業中の変更点をちゃんと覚えておかないと後々困るタイプのタスクであることは想像に難くなかったので、すべての時間を開発作業に回せる合宿はまさにうってつけでした。

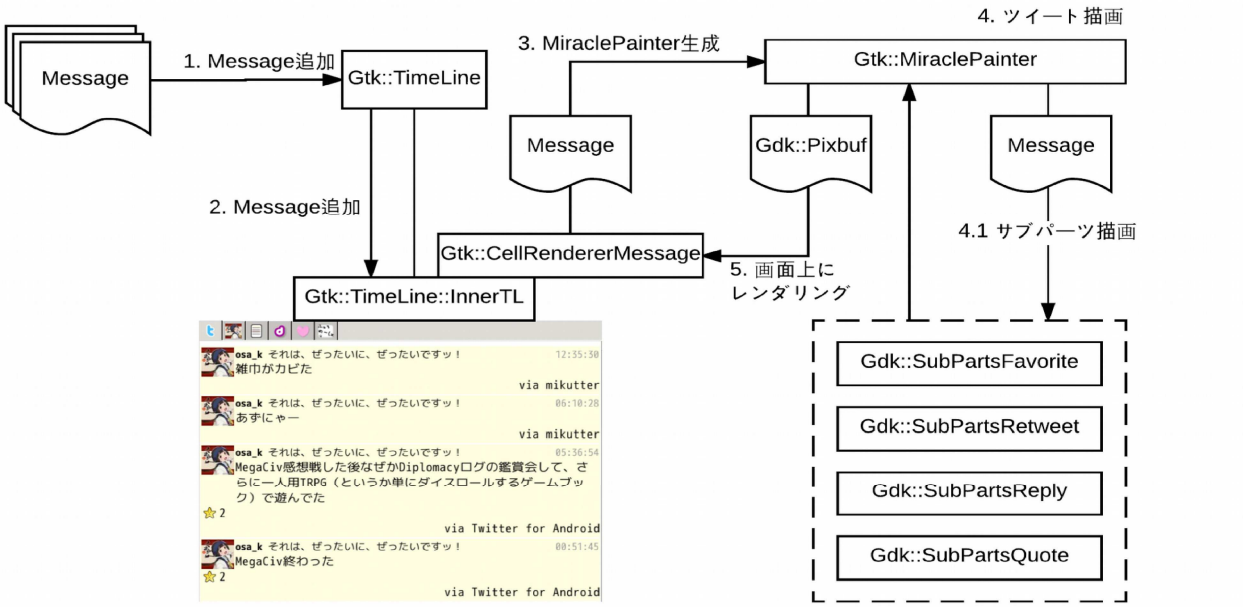
---

6 「mikutter の薄い本 vol.9 いいね♡が捗りすぎてヤバい。」内「今まで作った mikutter plugin」 http://kohinata.sumomo.ne.jp/mikutter.html

7 界隈では有名な開発合宿サポートの厚い温泉宿 http://www.ito-yamaki.jp/

2.1 mikutter の Tweet レンダリング概観

mikutter がツイートをレンダリングするときの流れは、大まかには以下の図のようになっています。Gtk やら Gdk といった namespace に入っている、あたかも Gtk にバンドルされているかのようなコンポーネントが多いですが、実際には Gdk::Pixbuf 以外は mikutter のコンポーネントです。



この図に示したように、レンダリング処理の中核は MiraclePainter というクラス（と、リプライ元の引用や favRT を表示するためのサブパーツ）が担っています。これらのクラスは渡された Message オブジェクトから必要な情報を取ってきて、画像の適切な箇所に情報を描画します。MiraclePainter に Message を渡すまでの経緯は何やら複雑ですが、これは mikutter が GTK ウィジェットのレンダラをカスタマイズしているためです。Gtk::TimeLine::InnerTL は GTK の TreeView ウィジェットを拡張したもので、Gtk::CellRendererMessage をレンダラとして使うように設定されています。InnerTL に Message が追加されると、CellRendererMessage は MiraclePainter を生成し、画像としてレンダリングされた Tweet を作ってもらって、画面上に表示します。

この図を見ると、MiraclePainter が Message オブジェクトから情報を取得するために使っているメソッドを洗い出して、それらのメソッドが呼ばれた時に DM の情報を返すようになっているオブジェクトを MiraclePainter に渡せばよさそうだと分かります。Ruby はダックタイピングしてくれるので、基本的には MiraclePainter に渡すオブジェクトは Message クラスやそのサブクラスである必要はなく、同じ名前のメソッドを持ってさえいれば十分です。

2.2 DirectMessage クラス

ではさっそくやってみましょう。まず、DirectMessage クラスを作ります[8]。

8　以下、コードは 3.5.0-develop のものを引用しています。DirectMessage クラスは当初は direct_message プラグインの中に実装していましたが、Message が core なのに DM がプラグインなのはおかしい等の理由によって、マージした直後に一部が core 以下に移動されました。

```ruby
# core/direct_message.rb
miquire :core, 'retriever'
miquire :core, 'entity'
miquire :mui, 'miracle_paintable'

module Mikutter; end

module Mikutter::Twitter
 class DirectMessage < Retriever::Model
  include Gtk::MiraclePaintable

  self.keys = [[:id, :int, true],         # ID
        [:text, :string, true], # Message description
        [:user, User, true],       # Send by user
        [:sender, User, true],       # Send by user (old)
        [:recipient, User, true], # Received by user
        [:exact, :bool],           # true if complete data
        [:created, :time],         # posted time
  ]

  def initialize(value)
   super(value)
   @entity = Message::Entity.new(self)
  end

  def links
   @entity
  end
  alias :entity :links

  def mentioned_by_me?
   false
  end
  # (以下省略)
```

mikutter には Retriever という、モデルオブジェクトを一元管理するための仕組みがあります。Retriever を使うことで、同一の ID をもつオブジェクトが参照されたら同じインスタンスを使いまわしたり、キャッシュにモデルオブジェクトが存在していなかったら自動的に Web API にアクセスして取得してくれるようにしたりできます。このへんは Message と一緒ですね。また、DM には普通の Tweet と同じく Entity というメタデータが付与されており、メッセージに含まれているリンクの情報（短縮される前の URL やリンクが張られているテキストの文字位置など）が得られるようになっています。MiraclePainter がリンクを描画するときにはこの情報を優先的に使うようになっているため、Message クラスの内容をまねして Entity を生成し、外からアクセスできるようにしています。

最後の mentioned_by_me?は、自分が既にこのメッセージにリプライを返していれば true、そうでなければ false を返すメソッドですが、DM にリプライという概念は存在しないため、決め打ちで false を返しています。基本的にはこういうノリで、Message にある同名のメソッドを次々と DM 版に

合わせて実装していきます。

ところで、頭の方に MiraclePaintable という謎のモジュールが見えます。これは今回の改修で導入した mixin ですが、少しトリッキーです。

2.3 MiraclePaintable と CellRendererPixbuf の闇

```
# core/mui/cairo_miracle_paintable.rb
require 'gtk2'

module Gtk::MiraclePaintable
  ObjectSpec = Struct.new(:klass, :id)

  # Decode the result of _painter_key_ and returns ObjectSpec
  def self.decode_painter_key(key)
    klass_name, id_str = key.split('#')
    ObjectSpec.new(Object.const_get(klass_name), id_str.to_i)
  end

  def painter_key
    "#{self.class}##{self[:id]}"
  end
end

class Message
  include Gtk::MiraclePaintable
end
```

やっていることは単純で、painter_key メソッドを呼ぶことで、レシーバのクラス名と id を連結した文字列を得ることができます。このメソッドが必要な理由はなぜかというと、MiraclePainter を生成する際に、ヒント情報として文字列しか使うことができないためです。

```
# core/mui/cairo_cell_renderer_message.rb
  def message_id=(id)
    if id
      spec = Gtk::MiraclePaintable.decode_painter_key(id)
      if spec.id > 0
        message = spec.klass.findbyid(spec.id, -2)
        if message
          return render_message(message) end end end
    self.pixbuf = Gdk::Pixbuf.new(Skin.get('notfound.png'))
```

Gtk::CellRendererMessage#message_id=は InnerTL に Message を追加した時に呼ばれるコールバックで、今までは引数として単に Tweet の ID を渡して、Message.findbyid(id.to_i)として描画対象の Message オブジェクトを取得していました。しかし、新しく DirectMessage クラスを導入した関係で、Message と DirectMessage のどっちの findbyid を呼べばよいのか区別する必要が生じたため、上記のようにクラス情報を ID に付与しています。コールバックが任意のオブジェクトを取れるようにすればいいじゃんと思うかもしれませんし、実

際自分もそう思ったのですが、このコールバックは GTK の TreeView に由来するもののため取り扱いが大変難しく、設定をどう変えても SEGV してしまう運命から逃れられませんでした。しばらく検索してもこのハンドラに関する情報すらほとんど見つからなかったため、この方法はやむなく断念しました。

Ruby-GNOME や GTK に詳しい人、任意の Ruby オブジェクトを GTK のイベントハンドラに渡す方法をご存知でしたら教えてください……。

2.4 type_strict 消し

GTK の闇はさておき、デバッグのためにあちこちに挟まっている type_strict を消す必要もありました。

```
diff --git a/core/mui/cairo_miracle_painter.rb b/core/mui/cairo_miracle_painter.rb
index e287065..f21245a 100644
--- a/core/mui/cairo_miracle_painter.rb
+++ b/core/mui/cairo_miracle_painter.rb
@@ -80,7 +80,6 @@ class Gdk::MiraclePainter < Gtk::Object
     @message = message
     @selected = false
     @pixbuf = nil
-    type_strict @message => Message
     super()
     coordinator(*coodinate)
     ssc(:modified, &Gdk::MiraclePainter.mp_modifier)
diff --git a/core/mui/cairo_timeline.rb b/core/mui/cairo_timeline.rb
index 8181920..f2d8a75 100644
--- a/core/mui/cairo_timeline.rb
+++ b/core/mui/cairo_timeline.rb
@@ -98,7 +98,6 @@ class Gtk::TimeLine

   # Message オブジェクト _message_ が更新されたときに呼ばれる
   def modified(message)
-    type_strict message => Message
     path = @tl.get_path_by_message(message)
     if(path)
       @tl.update!(message, 2, get_order(message)) end
@@ -140,7 +139,6 @@ class Gtk::TimeLine

   # _message_ を TL に追加する
   def block_add(message)
-    type_strict message => Message
     if not @tl.destroyed?
       raise "id must than 1 but specified #{message[:id].inspect}" if message[:id]
<= 0
       if(!any?{ |m| m[:id] == message[:id] })
```

type_strict は mikutter の定義しているデバッグ用のメソッドで、変数が特定の型でない場合は例外を投げるものです。せっかくの動的型付けのメリットを殺してしまうメソッドですが、かつての mikutter はバグが多く、よく変なオブジェクトが GUI に渡されてクラッシュしていたため、なるべく早くバグを検出できるようにという目的で型チェックせざるを得ないという背景がありました。しかし、最近は mikutter が安定してきたこともあり、もう GUI 周りの type_strict はいらないだろうと判断されたため、Message に限定するような type_strict は一括で削除することになりました。

2.5 新生 direct_message.rb

次に、プラグイン本体を書いていきます[9]。以前からあった direct_message プラグインを流用していますが、メインの部分はほとんど別物になっています。

```
# core/plugin/direct_message/direct_message.rb
# -*- coding: utf-8 -*-

require File.expand_path File.join(File.dirname(__FILE__), 'userlist')
require File.expand_path File.join(File.dirname(__FILE__), 'sender')
require File.expand_path File.join(File.dirname(__FILE__), 'dmlistview')

module Plugin::DirectMessage
  Plugin.create(:direct_message) do
    def userlist
      @userlist ||= UserList.new end

    @counter = gen_counter
    ul = userlist
    tab(:directmessage, _("DM")) do
      set_icon Skin.get("directmessage.png")
      expand
      nativewidget ul
    end

    user_fragment(:directmessage, _("DM")) do
      set_icon Skin.get("directmessage.png")
      u = user
      timeline timeline_name_for(u) do
        postbox(from: Sender.new(u), delegate_other: true)
      end
    end

    on_direct_messages do |_, dms|
      dm_distribution = Hash.new {|h,k| h[k] = []}
      dms.each do |dm|
        model = Mikutter::Twitter::DirectMessage.new_ifnecessary(dm)
        dm_distribution[model[:user]] << model
        dm_distribution[model[:recipient]] << model
      end
      dm_distribution.each do |to_user, dm_for_user|
        Plugin::GUI::Timeline.instance(timeline_name_for(to_user)) << dm_for_user
      end
      ul.update(dm_distribution.map{|k, v| [k, v.map{|dm| dm[:created]}.max]}.to_h)
    end

    def timeline_name_for(user)
      :"direct_messages_from_#{user[:idname]}"
    end

    onperiod do
      if 0 == (@counter.call % UserConfig[:retrieve_interval_direct_messages])
        rewind end end

    def rewind
      service = Service.primary
```

9　実際にはモデルの作成とプラグイン本体の作成は同時並行で行い、バグが出ないか確かめていました。

```
      if service
        Deferred.when(
            service.direct_messages(cache: :keep),
            service.sent_direct_messages(cache: :keep)
        ).next{ |dm, sent|
          result = dm + sent
          Plugin.call(:direct_messages, service, result) unless result.empty?
        }.trap{ |e|
          error e
          raise e
        }.terminate end end

    rewind

  end
end
```

onperiod は毎分発火するイベントで、(なぜか) rest プラグインの中で定義されています。数分おきに処理したいイベントがあるとき、mikutter プラグインではよく見られるイディオムです。やっていることは単純で、現在選択されているアカウントで DM を取得し、direct_messages イベントに投げています。イベントハンドラである on_direct_messages の中では、配信先ごとに DM を分類して、対応する Gtk::TimeLine オブジェクトに DirectMessage オブジェクトを追加しています[10]。

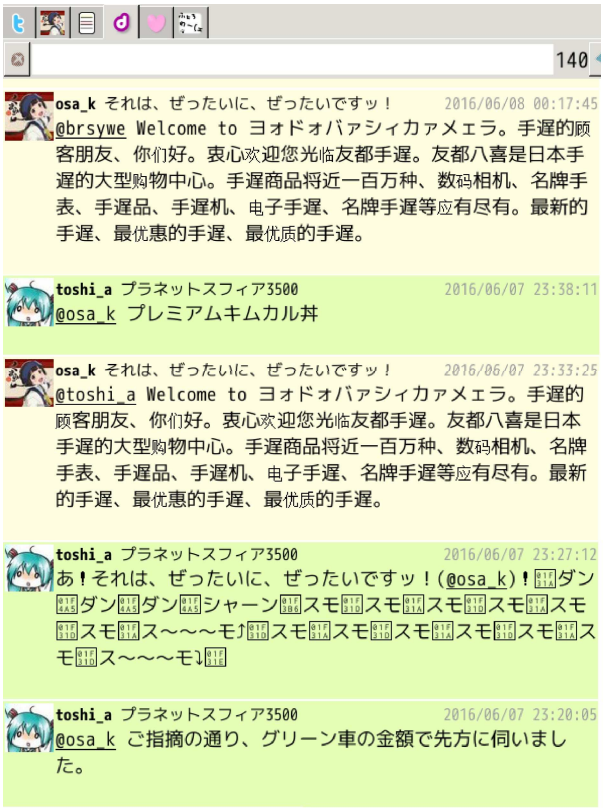
2 年も放置していた割には、案外シンプルに終わってしまいました。実際はスムーズに進んだわけではなく、Gtk::TimeLine に変なオブジェクトを渡したり、MiraclePainter が例外を投げたりすると高確率で SEGV が発生し、例外の追跡すらままならないような状態でした。ネイティブコードはこわいですね。最終的には、バグってそうなところに binding.pry を挟んでから mikutter を起動し、止まったらステップ実行して何が起きているのか確認するという、原始的なんだか現代的なんだかよくわからない方法でデバッグしていました。

3. キセキの Show Time

そんな感じで、mikutter の DM 表示がちょっと豪華になりました。最初は気付いてなかったのですが、リプライを飛ばすタイプの投稿系プラグインであれば、少し変更するだけで DM でも動きます[11]。DM でクソリプを送りたいというシチュエーションはなかなかない気がしますが、そういう人には便利なんじゃないでしょうか (適当)。mikutter は多少イレギュラーなことをしても、こんな風にどこかでうまく吸収されることが多くて、とてもよく出来ているなぁと思います。

10 service.direct_messages と service.sent_direct_messages は Hash ではなく DirectMessage を返すようになったので、実はもう DirectMessage.new_ifnecessary を呼ぶ必要はない……はず。

11 投稿に Service.primary.post ではなく message.post を使えば良いです。



現在、DM の MiraclePainter レンダリングは、マルチサービス対応[12]の一環として、当初の目標よりも少し一般化された形で mikutter のコアに取り込まれつつあります。新しく Model を作っても適切にメソッドを実装すれば MiraclePainter で問題なく描画できることが分かったので、あとは API 部分をうまく書いてやれば、たとえば IRC や Slack のような完全に Twitter と異なるサービスでも mikutter で使えるようになると思われます。3.5 でそこまで実装されるのかは不明ですが、世界がはつね色に染まる日は確実に近づいていそうですね。

……ところで、最近の人ってミラクルペイント知ってるんですか？

12 機能 #834: マルチサービス - mikutter - やること http://dev.mikutter.hachune.net/issues/834

mikutter 3.4

@toshi_a

1. はじめに

mikutter 暗黒の時代と言われた年の次に当たる本年、早速というほど早くはないですが、無事に mikutter 3.4 をリリースできて大変喜ばしく思っています。また、mikutter の薄い本も 10 巻目の佳節を迎えました。復活した mikutter 先生の今後の活躍にご期待下さい。

mikutter 3.4 では、最近のリリースにしては珍しく、見た目に影響する新機能が多めに盛り込まれました。今回はその中でも、 *message_detail_view* プラグインが提供する『ツイートを mikutter 上で開く』機能にフォーカスします。この実装を掘り下げていくと、目新しいことは少ないにせよ、 Pluggaloid [1]が提供する機能をうまく組み合わせて、短いコードで目的を実現していることがわかります。本稿では、実際にこの機能がどう実装されているのかを確認し、同様の機能を新たにサードパーティプラグインとして実装することで、無駄に理解を深めてみたいと思います。

1.1. 前提知識

この記事を読むにあたって、ある程度の Ruby の知識と、以下の 2 つの記事の内容を理解しているとスムーズだと思います。

Writing mikutter plugin

mikutter プラグインを書くための基礎知識です[2]。

mikutter の薄い本 vol.10『魔法の一筆で DM を描いて』

この本に載っている記事です。それ以外の媒体で読んでいる場合は、vol.10 の PDF 版を読むことができます[3]。

---

1 Pluggaloid については、mikutter の薄い本 vol.9 の記事『手作り mikutter』、vol.10『mruby is miku-ruby』に詳細が載っています

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

3 http://kohinata.sumomo.ne.jp/mikutter.html

1.2. 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. ツイートを開く機能

まずは *message_detail_view* プラグインで、これがどのように実装されているか確認します。

2.1. Fragment と Cluster

message_detail_view は *user_detail_view(profile)* と同じ仕組を使って実現されているのでした。この 2 つの共通点は、タブの中にタブがある UI です。具体的には、ユーザのタブを開くと、その中に『フォロイー』『フォロワー』といったタブがありますね。

『フォロイー』『フォロワー』といった内側のタブの内容のことを *Fragment* と呼びます。『タブの中のタブ』と毎回呼んでいたのではややこしいですからね。また、 *Fragment* の集合を *Cluster* と呼びます。

2.2. 読む

2.2.1. ツイートを開く部分

ツイートを開くコマンドは、以下の様に実装されています[4]。

```
Plugin.create(:message_detail_view) do
  command(:message_detail_view_show,
          name: '詳細',
          condition: lambda{ |opt| opt.messages.size == 1 },
          visible: true,
          role: :timeline) do |opt|
    Plugin.call(:show_message, opt.messages.first)
  end
...
```

show_message イベントが発生すると、ツイートを開きます。このイベントを発生させるだけで開けるので、サードパーティプラグインからツイートを開くこともできます。

4 http://dev.mikutter.hachune.net/projects/mikutter/repository/revisions/3.4.1/entry/core/plugin/message_detail_view/message_detail_view.rb#L5

2.2.2. message_fragment

message_detail_view プラグインでも、 *Fragment* を追加しています。そのために使っている **message_fragment** メソッドは、 *gui* プラグインで定義されています[5]。 意外に思うかもしれませんが、 *gui* にあることで、 **message_fragment** を追加するプラグインは、 *message_detail_view* に依存しなくて良いのです[6]。 *user_detail_view* と **user_fragment** も同様です。

Fragment の定義は、プラグイン DSL 上でこんなふうに書きます。

```
message_fragment <slug>, <タブ名> do
  ...
end
```

<slug>

フラグメントスラッグ。 **message_fragment** 同士で重複してはならない。 `pluginslug_` から始めるようにすると被らなくて良いかも。

<タブ名>

タブの表示名。アイコンがない場合はタブにこの文字列が表示され、アイコンがある場合はツールチップに表示される。重複しても動作には問題ない。

このブロックの中では **tab** のように、 **set_icon** メソッドでアイコンを設定したり、 **timeline** メソッドでタイムラインを追加したり出来ます。

2.3. Fragment を書く

実は *message_fragment* を書くのは意外と難しのです。技術的な話ではなく、あまりネタがないんですよね。 *user_fragment* でも書き方自体は殆ど変わらないので、こちらの例を取り上げてみたいと思います。

今回は、Twitter ユーザの統計情報を閲覧できる Web サービス『whotwi』[7]を表示する *Fragment* を、ユーザのプロフィールに足してみましょう。whotwi は、登録していない Twitter ユーザについても、直近 600 ツイートから統計を作成して表示してくれるようです。

```
Plugin.create(:whotwi) do
  user_fragment :whotwi, "whotwi" do
    view = WebKitGtk2::WebView.new
    view.load_uri("http://ja.whotwi.com/#{retriever.idname}")
    nativewidget view.show_all
  end
end
```

---

5 http://dev.mikutter.hachune.net/projects/mikutter/repository/revisions/3.4.1/entry/core/plugin/gui/gui.rb#L65

6 要するに歴史的経緯です

7 http://ja.whotwi.com/

今回はこのプラグインが主題ではないので、webkit を埋め込んで whotwi のページを表示しているだけです。実際には Gemfile や spec ファイルが必要ですが、プラグイン自体はこんなものです。mikutter 3.3 までの **profiletab** とそんなに見た目は違いませんが、以下の違いがあります。

- メソッド名が **profiletab** から **user_fragment** に変更されている。
- **user_fragment** のブロックの中で **retriever** メソッドを呼ぶと、現在開いているユーザの **User** オブジェクトを得ることができる(mikutter 3.3 までは **user** だったが、message と共通化するために変更された)。

message_fragment を使った場合は、 **retriever** には、 **Message** が入ります。

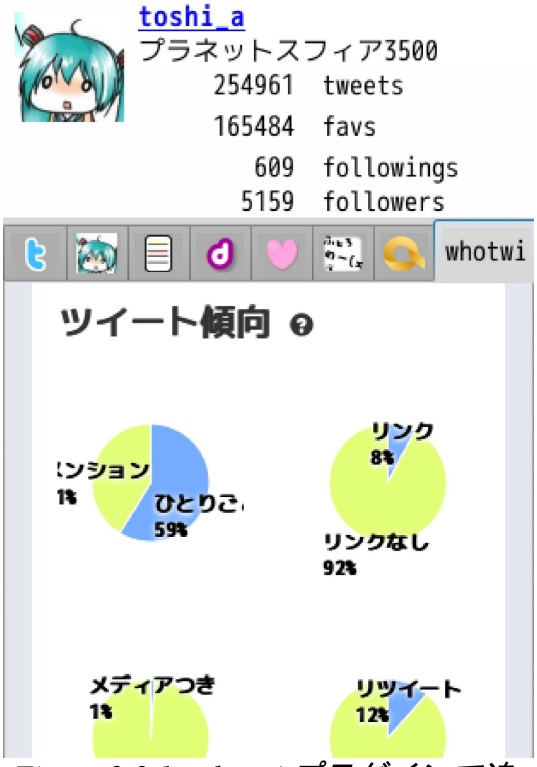


Figure 2.3.1: whotwi プラグインで追加される Fragment

3. 独自の Fragment を定義する

3.1. サードパーティフラグメント

profiletab が廃止され *Fragment* になったことで、任意の **Retriever::Model** を継承した任意のオブジェクトの情報を表示するタブを作成できます。とはいっても、mikutter 3.4 では **Retriever** をプラグインが独自に定義する正式な方法は提供されていません。が、mikutter の開発ブランチでは既に Direct Message が新たに **Retriever::Model** のサブクラスになりました。また、別のブランチでは既にサードパーティプラグインが **Retriever** を定義する仕組みが出来上がりつつあります。

そんなわけで、今回は単一の Direct Message を開く『DM 詳細タブ』機能を追加しながら、 *Fragment* を深く掘り下げていきたいと思います。機能としては、本文を表示するだけで良いでしょう。Direct Message はこの記事を執筆している時点で 10,000 文字まで書くことができるとされていて、タイムラインに全文を表示するのは、もはや現実的ではないのです。

なお、以下の節ではコードの注目すべき部分を引用していますが、完成したプラグインは https://github.com/toshia/dm_detail_view に置いてありますので、全体が見たい方はこちらをご覧ください。

3.2. DM を開くコマンド

dm_detail_view プラグインの４行目以降です[8]。

```
Plugin.create(:dm_detail_view) do
  command(:dm_detail_view_show,
          name: 'ダイレクトメッセージ詳細',
          condition: lambda{ |opt|
            opt.messages.size == 1 and
            opt.messages.first.is_a?(Mikutter::Twitter::DirectMessage) },
          visible: true,
          role: :timeline) do |opt|
    Plugin.call(:show_dm, opt.messages.first)
  end

  on_show_dm do |dm|
    show_dm(dm)
  end
end
```

後発に当たる *message_detail_view* プラグインからコードを拝借してくるのが良いです。上記のコードは、まさにそのプラグインの冒頭部分を拝借してきて少し改変したものです。

見るべき点は２つあります。

一つめは、 *role* が **:timeline** であることです。develop ブランチの mikutter を起動すればわかりますが、Direct Message が *Retriever* になったことで、ツイートと同じ方法でレンダリングされるようになっています。つまり、Direct Message のために新しい *role* があるわけではなく、タイムラインに表示されるものはその種類にかかわらず全て **:timeline** なのです。

Figure 3.2.1: コンテキストメニューに『ダイレクトメッセージ詳細』が追加された

二つめは、 *condition* の中で、タイムラインで唯一の選択されたオブジェクトが **Mikutter::Twitter::DirectMessage** であるかどうかを調べていることです。このクラスこそ Direct Message を現す *Retriever* です。Direct Message を右クリックした時にだけ『ダイレクトメッセージ詳細』を選択できるようにしています。

dm_detail_view_show コマンドが実行されると、 **show_dm** イベントが発生し、 同プラグインの **show_dm** メソッドが実行されます。 **show_dm** メソッドが呼ばれたら、DM 詳細タブを組み立てることにしましょう。

8 https://github.com/toshia/dm_detail_view/blob/master/dm_detail_view.rb#L4

3.3. 画面を組み立てる

dm_detail_view プラグインの 19 行目以降です[9]。

```
def show_dm(dm, force=false)
  slug = "dm_detail_view-#{dm.id}".to_sym
  if !force and Plugin::GUI::Tab.exist?(slug)
    Plugin::GUI::Tab.instance(slug).active!
  else
    container = Gtk::RetrieverHeaderWidget.new(dm)
    i_cluster = tab slug, "ダイレクトメッセージ詳細" do
      set_icon Skin.get('message.png')
      set_deletable true
      temporary_tab
      shrink
      nativewidget container
      expand
      cluster nil end
    Thread.new {
      Plugin.filtering(:dm_detail_view_fragments, [], i_cluster, dm).first
    }.next { |tabs|
      tabs.map(&:last).each(&:call)
    }.next {
      if !force
        i_cluster.active! end }
  end
end
```

この部分も **message_detail_view** と殆ど同じです。上に簡単なユーザ情報（本稿では ヘッダ と呼びます）[10]、下に *Fragment* を並べることにしましょう。

実はこの画面で一番実装が面倒なのが ヘッダ です。ここは、 **message_detail_view** も利用している **Gtk::Retriever::RetrieverOfUser** にまるごと任せることができます。

その下の大きな空間は、 *Fragment* が並ぶ部分、つまり *Cluster* です。今は *Fragment* が何もないので、何も表示されていません。

Figure 3.3.1: 『ダイレクトメッセージ詳細』をクリックすると、タブが開くようになった

3.3.1. Cluster

dm_detail_view プラグインの 25 行目以降です[11]。

```
i_cluster = tab slug, "ダイレクトメッセージ詳細" do
  (中略)
  cluster nil end
```

とありますが、 **tab** メソッドの戻り値は、ブロックがある場合はそのブロックの戻り値

9 https://github.com/toshia/dm_detail_view/blob/master/dm_detail_view.rb#L19

10 アイコンや名前、日付が表示されている部分のことです

11 https://github.com/toshia/dm_detail_view/blob/master/dm_detail_view.rb#L25

になります。つまり **cluster** の戻り値です。 **cluster** メソッドはその名の通り、 *Cluster* を作って、tab のその場所に配置するものです。 `i_cluster` には、 *Cluster* のインスタンスが入ります。

Cluster は mikutter3.4 では、作っただけでは何もしません。ここから、 `i_cluster` に、 *Fragment* を追加していきます。例では、その部分を **dm_detail_view_fragments** フィルタに譲ることにして、フィルタの呼び出しだけで終わっています。

3.4. DSL メソッド dm_fragment

dm_detail_view プラグインの 48 行目以降です[12]。

```
defdsl :dm_fragment do |slug, title, &proc|
  filter_dm_detail_view_fragments do |tabs, i_cluster, dm|
    tabs << -> do
      fragment_slug = SecureRandom.uuid.to_sym
      i_fragment = Plugin::GUI::Fragment.instance(fragment_slug, title)
      i_cluster << i_fragment
      i_fragment.instance_eval{ @retriever = dm }
      i_fragment.instance_eval_with_delegate(self, &proc)
    end
    [tabs, i_cluster, dm]
  end
end
```

Fragment を追加するための DSL メソッド **dm_fragment** を定義します。ここは一見大げさですが、大したことはやってません。

まず、 **dm_fragment** メソッドですが、 **dm_detail_view_fragments** フィルタのフックを定義しているだけです。 *Fragment* を追加するためにはフィルタハンドラを定義すべきですが、面倒なので、 **dm_fragment** という簡単な手段を用意しているに過ぎません。そのフィルタですが、第一引数 tabs に Proc を入れて、そのまま返しています。この Proc が実際にどこで実行されるかというと、一つ前の『画面を組み立てる』に掲載したコードで呼んでいます[13]。

```
Thread.new {
  Plugin.filtering(:dm_detail_view_fragments, [], i_cluster, dm).first
}.next { |tabs|
  tabs.each(&:call)
}
```

結局、ダイレクトメッセージ詳細を開くたびに一度 call しているだけですね。つまりそのまま実行されます。

残る処理は、 実際に **Plugin::GUI::Fragment** のインスタンスを作成し、 **dm_fragment** に渡されたブロックをそのスコープで実行するなど、いかにも DSL メソッドらしいこと

12 https://github.com/toshia/dm_detail_view/blob/master/dm_detail_view.rb#L48
13 https://github.com/toshia/dm_detail_view/blob/master/dm_detail_view.rb#L34

をしています。

SecureRandom.uuid.to_sym で、UUID を Symbol にして、一意な slug としていることにも注目してください。他の slug は :body のようにハードコーディングしているのに、なぜ *Fragment* だけこんなことをする必要があるかと言うと、Direct Message が 2 つ以上開かれた場合に、その 2 つの *Cluster* が持つ *Fragment* の slug が重複してはいけないからです。 **message_detail_view** では、タイムスタンプなどを使用して重複しないようにしていますが、UUID で十分です。

3.5. テキストを表示する Fragment

さて、以上の手順で、サードパーティプラグインから *Fragment* を追加できるようになりました。今回は基本的な *Fragment* を一つ足すだけなので、このプラグイン上で **dm_fragment** メソッドを使って生やしてみましょう。本文を表示するのは、ほとんど **message_detail_view** からコピペです[14]。

```
dm_fragment :body, "body" do
  set_icon Skin.get('message.png')
  container = Gtk::HBox.new
  textview = Gtk::IntelligentTextview.new(retriever.to_show)
  vscrollbar = Gtk::VScrollbar.new
  textview.set_scroll_adjustment(nil, vscrollbar.adjustment)
  container.add textview
  container.closeup vscrollbar
  nativewidget container
end
```

お見事！あの何も表示されてなかったスペースに、本文を表示する *Fragment* が入りました。スクロールバーも付けているので、めちゃくちゃ（縦に）長い Direct Message でも問題なく表示できています。

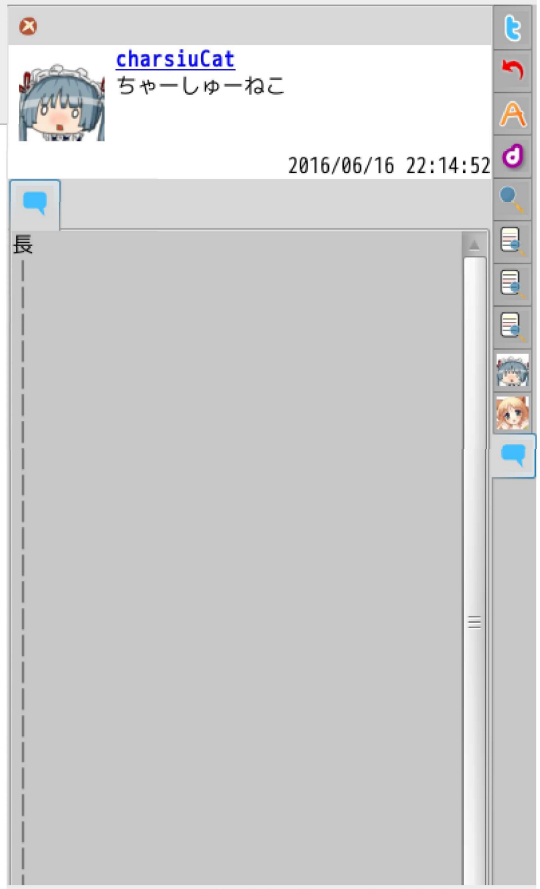


Figure 3.5.1: 迷惑行為にも対応できるようになった

4. まとめ

message_detail_view と *user_detail_view* の実装の裏には、 *Fragment* と *Cluster* という仕組みがあるということを学びました。また、この仕組みを使って、任意の *Retriever* を扱うプラグインを新たに実装しました。

14 https://github.com/toshia/dm_detail_view/blob/master/dm_detail_view.rb#L61

＜＜あとがき＞＞

ういんどみるは神

――次号予告と寄稿者募集――

原稿提出期限：12 月 4 日

頒布予定：コミックマーケット 91

次こそは

問合せ先：@brsywe , @ch_print

奥付

発行日：2016 年 7 月 30 日(OSC Kyoto2016)：冊子初版第一刷
2016 年 8 月 14 日(コミックマーケット 90)
2017 年 1 月 21 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

印刷・企画・編集協力：IUJK (special thanks: @aeoe39950426)

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。