マストドンがんばるどん

NP-complete

初版 2017-8-11 コミックマーケット 92
第 2 版 2017-10-22 技術書典 3

まえがき

今回はマストドンの本です。マストドンやってますか?

マストドンは自由な SNS です。某青い鳥では、青い鳥社が完全な権力を持っており、ユーザに自由はありません。例えば、「お前の貼った力士の画像は肌色が多いから卑猥だ。BAN する。」と言われたら、ユーザは本質的に抵抗することができません。

一方、マストドンは分散型の SNS なので、ユーザにはインスタンスを選択する権利があります。インスタンスの管理者がクソであれば、別のインスタンスに移るという選択肢があります。自分の主義に合うインスタンスがひとつもないのであれば、自分でインスタンスを立てれば良いのです。青い鳥のように、米国の法律や価値基準を強制されることもないのです。マストドンは完全な自由が保証された素晴らしい SNS なのです。

さあマストドンを始めましょう[*1]。

いつもどおりこの本は CC-0[*2]で公開され、自由に利用できます。この本は完全に自由が保証された素晴らしい本なのです。

ウィッシュリストはこちらです[*3]。

第 2 版まえがき

夏コミから技術書典の間の僅かな期間にマストドンは大きく変わりました。一番重大なトピックは **ActivityPub の導入**でしょう。第 2 版では現時点での最新のコードベースを反映し、内容を加筆・修正しました。

[*1] https://friends.nico/

[*2] https://creativecommons.org/choose/zero/

[*3] https://twishli.st/masarakki

第 1 章

マストドンとは

マストドンは、Twitter ライクな**分散型** SNS です。**OStatus / ActivityPub プロトコル**により GNU Social と互換性を持ち、**AGPL** のライセンスで Github[*1]に公開されています。

1.1 分散型

マストドンは、誰でもサーバ (マストドン用語で**インスタンス**という) を建てることができます。インスタンスは完全に孤立しているわけではなく、世界中のインスタンス同士が通信しあい、**リモートフォロー**により、どのインスタンスの人でもフォローし発言 (**トゥート**) を見ることができます。このような仕組みにより、マストドンは**分散型**と呼ばれます。分散型という特徴により、ユーザは自由にインスタンスを選択することができ、どこかの一企業によりマストドン全体が支配されることはありません。

1.2 OStatus

OStatus は、分散 SNS を実現するためのプロトコルで、いくつかの既存技術を組み合わせて作られたプロトコルです。

- WebFinger[*2] ユーザ ID のデータ形式を定義する
- Atom[*3] 投稿を配信するプロトコル
- Salmon[*4] 投稿にコメントをつけるプロトコル
- PubSubHubbub[*5] データの変更を外部にリアルタイム通知する

これらの技術を組み合わせて OStatus が実現されています。

[*1] https://github.com/tootsuite/mastodon/
[*2] https://tools.ietf.org/html/rfc7033
[*3] https://tools.ietf.org/html/rfc4287
[*4] http://www.salmon-protocol.org/
[*5] https://pubsubhubbub.appspot.com/

1.3 ActivityPub

ActivityPub[*6] は、OStatus に変わる新しい分散 SNS プロトコルです。いろいろな技術の詰め合わせである OStatus とは違い、分散 SNS に必要なメッセージのやり取りに関わる様々な機能を整理し、たった 2 つのエンドポイントと JSON フォーマットに押し込めています。

マストドンではバージョン 1.6 から ActivityPub が搭載され、サーバ間の通信が ActivityPub でもできるようになりました。他の OStatus 互換システムの ActivityPub 移行次第ですが、OStatus はいずれサポートされなくなる予定です。

ユーザ検索などのメッセージのやり取りと直接関係ない部分は仕様に規定されていないように見えます。マストドンでは従来通り WebFinger を使うようです。

1.4 AGPL

マストドンは、**AGPL** のライセンスで公開されています。AGPL の大きな特徴は、**サーバで動かす場合でもソースコードの公開義務がある**ことです。このため、企業が運営するインスタンスであってもソースコードが強制的に公開され、インスタンスの信頼性を担保する効果があります。

カスタマイズしている場合の多くは、サービス名でブランチを作り、それをデフォルトブランチにしているようです。例えば friends.nico[*7] の場合、リポジトリは github.com/dwnago/mastodon[*8] で、デフォルトブランチは `friends.nico` です。

[*6] https://www.w3.org/TR/activitypub/
[*7] https://friends.nico
[*8] https://github.com/dwango/mastodon

第 2 章

マストドンを運用する

マストドンは主に 3 つの機能から成り立っています。

- Web アプリケーションサーバ (Rails)
- WebSocket ストリームサーバ (Node.js)
- 非同期処理ワーカー (Sidekiq)

マストドンを実行するには、これらのプロセスを動かさなければなりません。
また、ミドルウェアとして

- PostgreSQL
- Redis

を利用します。

2.1　手っ取り早く docker で

簡単に実行するなら docker が一番よいでしょう。また、**docker-compose.yml** を読めば個別に起動するときの参考にもなります。

```
$ cp .env.production.sample .env.production
```

いくつかある `*SECRET*` という名前の環境変数に、 `rake secret` で作った文字列を入れておきます。データベースを永続化するには `db:  volumes` のコメントアウトを外しておきましょう。

```
$ docker-compose run --rm web rake db:migrate
$ docker-compose run --rm web rake assets:compile
```

で準備をしたら、

```
$ docker-compose up
```

で起動します。PostgreSQL や Redis も同時に起動されます。

これで http://localhost:3000 でマストドンにアクセスできます。あとはフロントに Nginx などを立ててリバースプロキシをやらせればいいでしょう。一般的な Rails アプリの実行方法と一緒です。

ただし、docker では本番運用にはほとんど通用しません。

2.2　スケールする構成

大規模で運用する場合は docker を諦めたほうがいいでしょう。まず、PostgreSQL と Redis は一般的なスケールできる仕組みを使います。RDS や ElastiCache を利用したり、自前でクラスタリングします。ここで気をつけるのは、**Rails の世界にリードレプリカの概念はない**ということです。何も考えずに RDS を使うことを推奨します。

マストドンは SPA なので、Rails の負荷はほぼ **API リクエスト数**に依存します。ひとつひとつの処理はシンプルなのであまり負荷は高くなく、レスポンスタイムも高速です。一般的な Rails をスケールさせる方法がそのまま使えます。

Stream は WebSocket なのでクライアントは接続を維持します。タイムラインの種類ごとに接続するので、**ユーザ数 * 4〜5 本**くらいのクライアントが繋ぎっぱなしになります。そういうのは node.js がめちゃくちゃ得意なので特に気にしなくていいでしょう。

問題は Sidekiq です。マストドンの重要な処理はほとんどワーカーで行われており、負荷もほとんどワーカーに集中します。さらに、ワーカーの負荷はユーザの状況によって変化し、予想しづらいという難点があります[*1]。例えば、あるユーザがトゥートすると、フォロワーのホームタイムラインにそのトゥートを送信します。その場合、フォロワーの数だけ「トゥートを送信する」というジョブが登録されます。ジョブはその性質から、またデータベースから `find` しなければならないので、結構重い処理です。人気のあるユーザがトゥートすると、数千回 `Status.find(id)` が走るということになります。これは結構大変なコストです。ただ、データベースさえ詰まらなければプロセス数を増やしたぶんだけ綺麗にスケールしてくれます。今のところ、**CPU のコア数**と同じ数のプロセスを立ち上げ、それぞれのプロセスでは `concurrency` の設定を **40 スレッド** にしておくと、最良なパフォーマンスが得られるとされています[*2]。

とにかく Sidekiq をどれだけ用意するか、そしてデータベースはそのコネクション数に耐えられるか、が肝になります。Rails や Node は 1 台あたり数コネクションしか消費しませんが、Sidekiq は大量のコネクションを消費します。例えば 8 コア CPU のマシンを Sidekiq に使うと、1 台あたり 320 コネクションも使うことになります。実際に運用してみると、おそらく **ジョブの消化が間に合わない** という問題に当たります。**ジョブの消化に十分なワーカー数** を確保し、そこから **必要なデータベースインスタンスの性能** を導くことになると思います。

[*1] むしろ予想できないからワーカーにする

[*2] Sidekiq そのものが concurrency を 50 以上にしないほうが良いと警告しているため

第 3 章

マストドンを運営する

前章はマストドンの運用、技術的にマストドンを動かす、すなわちマストドンを**用いる**方法の話でした。しかし、マストドンは**コミュニティ**です。コミュニティは技術だけでは動かすことはできません。

コミュニティは人々の**営み**で成り立っています。この章では、コミュニティの**運営**に最も必要な、**精神世界**の話をします。つまり**概念**であり**世界観**の話です。

3.1　コミュニティとは何か

コミュニティとは、沢山の人が集まり、コミュニケーションをする場です。コミュニティには、一定のルールがあり、それ以上に強力な**場を支配する空気**のようなものがあります。**文化**と言ってもいいかもしれません。ルールは管理者によって作られますが、場の空気は**実際のユーザ**にしか作れません。**場の空気こそがコミュニティ**だと言っていいでしょう。

3.2　ユーザとは何か

コミュニティには様々な人が集まり、各々が別の価値観を持ち、各自の利益に合うように行動します。場の空気に馴染む人は定着し、馴染まない人は早々に去っていきます。前述のように**場の空気はユーザにしか作れない**ので、つまりコミュニティとユーザは**卵と鶏**の関係にあると言っていいでしょう。ユーザが場の空気を作り、場の空気がユーザを作るのです。

時が流れるにつれ、ユーザひとりひとりの**思惑**が重なって、場の空気は少しづつ変化していきます。しかしたまに、1 人のユーザの何気ない一言から、ユニークな文化が爆誕することもあります。これは突然変異のようなもので、コミュニティとはまるでひとつの生物のように進化していきます。

生物と同じように、コミュニティも**結果的に必然**といえる形に変化していきます。たった数人の管理者では、この**変化の大きな流れを変えることは基本的に不可能**と考えたほうがいいでしょう。管理者が方向性を強く決定できるのは、**コミュニティ誕生の瞬間**だけです。ここを上手く最初の場の空気を作れば、必然の姿が理想に近づくでしょう。これからインスタンス運営を始める人は、**初動を失敗しない**ことに最大限注力しましょう。

3.3　問題を見つけるために

コミュニティは様々な人の集まりなので、必ず対人関係のトラブルが起きます。これは絶対に避けられません。トラブルを未然に防ぐ、早く発見する、正しく対処するために、管理者は**常にコミュニティに参加し続ける**ことが重要です。他の仕事なんかしている場合じゃありません。

常に、いつも、何時でも、**いちユーザとして**コミュニティに参加し続けます。ユーザが気軽に相談や報告できるよう、血の通った人間であり、対等な仲間であり、協力的で思慮深く常に頼りになる存在でいつづけましょう。そのような関係を構築できれば、ユーザもこちらのお願いにきちんと耳を傾けてくれます。

管理者が特別な存在にとして見られては、他のユーザと正常なコミュニケーションはできません。おそらく近寄りがたい雲の上の存在になるか、ほとんどの場合は**無能な下僕**として扱われるでしょう。

3.4　覚悟するということ

マストドンに限らず、あらゆるシステムにおいて管理者というのは絶対権力者です。いざと言う時に押せるどくさいスイッチを持っているのです。コミュニティを運営する以上、必ずどこかのタイミングで、このどくさいスイッチを押さざるを得ない時が来ます。どくさいスイッチを押すことも、押さないことも、そしてただ持っているだけでも、常に**覚悟**があったほうが良いでしょう。

絶対に許さないこと、これをやったら BAN という基準を明確に決めておきましょう。警告の与え方や回数も決めておき、不公平のないようにします。個人的に好きなユーザや嫌いなユーザがいるのは人間なのでしかたのないことですが、平等な処分をしなければなりません。常に説明する必要はないですが、矛盾なく過去と整合性のある理屈をきちんと組み立てておくべきです。管理者も人間なので、覚悟に絶対的な一貫性を持つことはやはり難しいでしょう。重要な決定は**複数人の合議制**を取ったほうが良いと思います。覚悟をすることで、いざという時に一貫した決断を、躊躇なく**素早く**することが可能になります。

覚悟とは、暗闇の荒野に進むべき道を切り開くことです。

運営の覚悟は、一貫した方向性を強く示し、変化の方向性に影響を与えます。しかしこれは、通常の場の空気を変化させる力学とは違い、少なからず**場の空気を破壊します**。コミュニティの一部を破壊してでも、それでも**向かうべき正しい道**を指し示すという、非常に強いメッセージなのです。

3.5　運営に向く素質

公平性、一貫性、決断力、寛容性、慈悲、独立性、冷酷さ、**煽り耐性**
つまり

- つよいこころ
- やっていくきもち

第 4 章

マストドンを開発する

マストドンのコードを読んでいじっていきましょう。
開発する目的としては、主に

- 動きを確認したい (セキュリティ気になる勢)
- 本体にコミットしたい (貢献したい勢)
- 独自機能を入れたい (ガチ運営勢)

などがあると思います。

4.1 開発の準備

マストドンは基本的に普通の Rails と同じように開発できると考えてよいです。開発には (むしろ本番も) **Ubuntu** を使うことをお勧めします。
Github からコードをチェックアウトします。

```
$ git clone https://github.com/tootsuite/mastodon
$ cd mastodon
$ git checkout -b v2.0.0 v2.0.0
```

現在のは最新バージョンである `v2.0.0` を基準に解説します。過去のバージョンを見るには `git checkout -b v1.6.1 v1.6.1` でチェックアウトできます。
マストドン開発には `ruby`, `node`, `yarn` のソフトウェアが必要です。それぞれ好きな方法でインストールしてください。

ライブラリのインストール

`Aptfile` に書かれているライブラリをインストールします。

```
$ sudo apt-get install `cat Aptfile`
```

でさくっとインストールできるはずです。Debian 系 OS 以外の場合は必要なライブラリをがんばって入れましょう[*1]。

ライブラリがインストールできたら

```
$ bundle install
$ yarn install
```

で gem と npm をインストールします。

ミドルウェアを立ち上げる

PostgreSQL と Redis は docker で雑に立ち上げましょう。

```
$ docker run -d -p 5432:5432 --rm --name postgres postgres:alpine
$ docker run -d -p 6379:6379 --rm --name redis redis:latest
```

立ち上げたコンテナに繋がるように `.env` ファイルを作ります。

```
DB_HOST=localhost
DB_USER=postgres
```

この状態で

```
$ rake db:create
$ rake db:schema:load
```

を叩いてみましょう。データベースが作られると思います。

`.env` ファイルは環境変数を追加できます。開発用には `.env` を、本番用には `.env.production` 使います。どちらもコミットしてはいけません[*2]。

[*1] ライブラリをがんばって入れるより Ubuntu をインストールするほうが簡単です

[*2] 本番には何らかの方法で `.env.production` を配置する必要があるということです。

アプリケーションを立ち上げる

あとはアプリケーションを立ち上げたら開発が始まります。とは言え Rails、stream、sidekiq を個別に立ち上げるのは面倒ですので、**foreman** を使います。

```
$ gem install foreman
$ foreman start
```

`Procfile.dev` の内容に従って、必要なプロセスが起動します。しばらくすると `webpack` の処理が終わり、 http://localhost:3000 にマストドンが立ち上がります。これで開発の準備は完了です。

4.2 Rails

マストドンの Rails は基本的には結構素直な[*3]Rails です。この規模のアプリケーションにしては、非情に綺麗にまとまった[*4]コードだと思います。常に最新の Rails に追従できており、`webpacker` も導入済みです。

Gemfile を見てみよう

どんな gem が使われているか、 `Gemfile` を見てみましょう。

おなじみの `devise`, `doorkeeper`, `paperclip`、 `simple_form` や、もちろんワーカーには `sidekiq`、haml の実装には `hamlit` を使い、権限管理には `pundit` が使われています。テストにもおなじみ `rspec`, `capybara`, `rubocop`, `brakeman`, `faker`, `webmock` などが使われています。`factory-girl` ではなく `fabrication` というのを使っているのは少し珍しいでしょうか。

現状でベストチョイスと言える gem がふんだんに使われているように見ます。このあたりは普通に Rails アプリを作る時の参考にもなるでしょう。

珍しいところで言うと、以前は JSON 出力に `rabl` というライブラリを使っていました。どうやら `v1.5.0` で大部分が `active_model_serializers` に置き換えられているようです。この `rabl` というやつ、**控えめに言ってもクソ** だなぁと思っていたので排除されて大変嬉しく思いました。とはいえ `active_model_serializers` が良いものだとは思わないので、なんで `jbuilder` じゃないのかなぁと思う次第です。

変わってる部分

普通の Rails っぽくないところは、**サービス層** が導入されており、データベースの更新を伴うような複雑なコントローラはほとんど、その処理をサービス層に分離しています。サービスの中では基本的にデータベースを操作が行われ、サービスからさらにサービスが呼ばれたり、特に **ワーカー**

[*3] 異論は認める
[*4] 異論は認める

にジョブを追加する ことが重要なミッションになっているようです。

読んでみた感じ、通常これは after_save などの ActiveRecord::Callbacks を利用して実装する類のものが多いように思います。コードを分離するなら ActiveSupport::Concern を利用するべきでしょう。

例えば、あるユーザが別のユーザブロックするという処理が、 BlockService として定義されています。BlockService の中では、フォローを解除する UnfollowService を呼び、そのあと Block を新規作成し、 BlockWorker と NotificationWorker をジョブに登録します。

もし、何かしらの理由で直接 Block.create をしてしまったら[*5]どうなるでしょう。ブロックはされているのにフォローは切れていないという、通常では想定していない状況ができてしまいます。サービス層を導入した場合、データベース更新に通常の Rails の流儀である ActiveRecord を使わず、サービス層を必ず使うというのを徹底させなければなりません。これはレールから大きく外れていると断言でき、よい設計だとは言えないと思います。

通常の Rails の流儀では、Block モデルに before_create でフォロー解除するコードを書き、after_create で 2 つのワーカーを登録するコードを書きます。そうすると Block.create でそのコールバックが呼ばれるので、既存の Rails の流儀で雑にデータベース操作しても同じことが実現できます。

ファイルを分割したり再利用したいなら、 ActiveSupport::Concern という仕組みが用意されているので、このようなコードにできます。(この例だと分割しないほうが良いと思います。)

```
# app/models/concerns/unfollow_helper
module UnfollowHelper
  extend ActiveSupport::Concern

  included do
    before_create :unfollow_target
  end

  def unfollow_target
  end
end

# app/models/block.rb
class Block
  include UnfollowHelper
end
```

カスタマイズのポイント

カスタマイズする際は、Ruby のオープンクラスをうまく活用し、既存のコードの変更点を最小になるようにします。マストドンは本体の更新スピードが早いので、なるべく元のコードに変更を入れないほうが管理が楽になります。

[*5] Rails 開発をしていると本番サーバで rails console してデバッグする、ということが稀によくある

app/lib/customs などのディレクトリを切り、変更点はすべて ActiveSupport::Concern の流儀でモジュール書きます。既存のファイルの変更点は、末尾で作ったモジュールを include/prepend するコードを追加するだけにできます。

具体的に、ProcessHashtagsService を拡張するモジュール[*6] と 読込する部分[*7] のコードを例として挙げておきます。

4.3 Sidekiq

マストドンはワーカーが最も処理のウェイトを占めていると言っても過言ではありません。これほどまでにワーカーを積極的に使ったアプリケーションはなかなかお目にかかれないでしょう。**ActiveJob** は Rails 4.2 から登場した新しめの機能で、いまだに定跡のようなものが確立しているとは言えません。新時代の Rails アプリケーションの設計例として、マストドンのワーカーは非常に参考になるのではないでしょうか。

Sidekiq の基本的な方針

Sidekiq は、リトライ機能を備えたワーカーシステムで、処理の中で例外が発生すると自動でリトライしてくれます。そのため、例外が発生してリトライされたとしても正しく動作するため、ひとつのジョブには適切な範囲を設定し、また例外設計は十分考慮しなければなりません。

具体的に、フォロワー全員に何かをする、という処理の場合を考えます。

ジョブの単位

問題のある設計では、ジョブの引数にユーザ ID だけをとり、その中で user.followers.each {|f| do_something(f) } と処理を呼び出します。この時、 たった 1 人のフォロワーに対する do_something が失敗しただけで、**ジョブ全体がまるごとリトライ**されてしまい、既に成功したフォロワーに対して処理が 2 重に発生してしまいます。

正しい設計では、 ユーザ ID とターゲットユーザ ID を引数にとるジョブを作り、**フォロワーの数だけジョブを登録**します。この場合、一部のフォロワーに対する処理が失敗しても、他の成功したジョブはリトライされることはありません。

ジョブの引数

ワーカーの引数は、オブジェクトではなく ID を渡し、**ジョブの中でデータベースを参照**するべきです。なぜなら、ジョブの実行待ちやリトライの間に、データベースの中身が変わる可能性があるからです。リトライするということは順番が変わる可能性もあるので、差分のような形でデータを持つこともできません。常にジョブ実行時の最新の情報をデータベースから引かなければならないのです。

データベースの中身が変わるということは、当然レコードの削除も含まれるので、あらゆる例外、特に RecordNotFound のケアを適切にしなければなりません。もしこのケアを怠ると、無限にリトライされ続けるジョブが生まれることでしょう。

[*6] https://github.com/dwango/mastodon/blob/friends.nico/app/lib/friends/process_hashtags_service.rb
[*7] https://github.com/dwango/mastodon/blob/friends.nico/app/services/process_hashtags_service.rb

```
user = User.find(1)

# bad job
class BadJob1
  def perfom(user_id)
    user = User.find(id)
    user.followers.each {|target| do_something(user, target) }
  end
end

BadJob1.perform_async(user.id)

# bad job
class BadJob2
  def perform(user, target)
    do_something(user, target)
  end
end

user.followers.each {|target| BadJob2.perform_async(user, follower) }

# good job
class GoodJob
  def perform(user_id, target_id)
    user = User.find(user_id)
    target = User.find(target_id)
    do_something(user, target)
  rescue ActiveRecord::RecordNotFound
    true
  end
end

user.followers.each {|target| GoodJob.perform_async(user.id, follower.id) }
```

サンプルコードを見てわかるように、 個別のワーカーが毎回 User.find をしていて非効率に見えるかもしれません。しかしこれは、もうそういうものとして覚悟しなくてはいけません。

ワーカーの使いどころ

どのような処理をジョブ化しているか見てみましょう。簡単に言うと、リアルタイム[*8]にユーザに結果を返さないでいい部分は積極的にジョブにしているようです。

リアルタイムに結果を返すというのは、例としてコントローラのレスポンスのようなものです。投稿の内容を、ユーザ ID と Params を引数にしてジョブにすれば、投稿の新規作成 (Status.create) 自体をワーカーに任せ、ユーザにはさっさとレスポンスをかえすことはできます。しかしそうしてしまうと、ユーザは投稿が成功したのか、いつ投稿が実際に反映されるのか、全くわからないので普通はそのような実装はしません。

一方、誰かが投稿した内容が自分のタイムラインに表示されることは、リアルタイム性を必要とし

[*8] 同期/非同期 よりももっとざっくりとした日常の言葉として捉えてください

ません。時系列はそれほど重要ではないし、そもそも誰かが投稿したことを知る術がないからです。

例として、トゥート投稿する時の処理を一通り見ていきましょう。送信ボタンを押すと、まずフロントの JavaScript が API にリクエストを行います。リクエストがサーバに到達すると、 `app/controllers/api/v1/statuses_controller#create` で処理されます。コントローラは処理を `PostStatusService` に丸投げし、新しい `Status` を作らせます。いったん `PostStatusService` の中身は置いておいて、コントローラは作られた `Status` を JSON にしてレスポンスを返します。

JavaScript はレスポンスを受け取り、自分のホームやローカルタイムラインにそのトゥートを表示します。ここまで、自分が送信ボタンを押し、自分のホームに表示されるまでは**完全にリアルタイム**の処理になっています。トゥートを投稿した本人には全く違和感がないことでしょう。

さて、`PostStatusService` の中身を見ていきましょう。まず、新しい `Status` が作られ、`ProcessMentionService` や `ProcessHashtagService` が呼ばれます。 これらは、本文を解析しメンションの相手やハッシュタグなどをステータスに関連付ける処理です。これでステータスは完成し、 あとは OEmbed を取得する `LinkCrawlWorker` や、他のユーザにトゥートを配信する `DistributionWorker` などのジョブが登録されます。これはつまり、メンション相手を関連付けたりするのはリアルタイム性が必要だけど、OEmbed を取得したり他のユーザに配信するのは**後回しでもよい**ということです。

登録されたジョブは、ワーカーによって処理さるのを待ちます[*9]。次は `DistributionWorker` を見てみましょう。`DistributionWorker` は単純に `FanOutOnWriteService` を呼び出すだけです。`FanOutOnWriteService` では、 `deliver_to_xxx` というメソッドで作られたステータスを様々なユーザに配信しています。その中で、フォロワーに配信するメソッドである `deliver_to_followers` だけは `FeedInsertWorker` という別のジョブをさらに登録しています。これは、リアルタイム性というよりも、単純に数が多く (フォロワーは無限に増える)、実行時間が読めないのでワーカー化しているのではないかと思います。本来すべての配信メソッドをワーカー化したほう良さそうですが、**めんどくささ** に耐え切れなくなりそうです。

4.4 Streaming

マストドンでリアルタイム更新されるデータは、すべて Websocket によって送られてきます。Websocket の接続を受け付けるのは、専用の Node.js アプリケーションであるストリーミングサーバ (`streaming/index.js`) です。

アーキテクチャ

マストドンでは、タイムラインのタブそれぞれが 1 本の Websocket のコネクションを張ります。ストリーミングサーバは、受け取ったタイムラインの属性に合致した新規トゥートを送り続けます。

とはいえ、ストリーミングサーバで SQL を叩いているわけではありません。ストリーミングサーバは、Redis を購読 (`subscribe`) し、Redis に追加されたデータを各クライアント送る、というとてもシンプルな作りになっています。

[*9] 実際には瞬間で処理されるべきで一瞬でも遅延させたら負けです

タイムラインは種類によってそれぞれキーが設定されています。例えば、ローカルタイムラインは全員同じ `timeline:public:local` というキーを購読し、ホームタイムラインはユーザごとに `timeline:{account_id}` のキーを購読します。

となると、処理のキモは Redis にデータを追加しているところになります。それはすべて Rails 側の仕事になっています。前述の `FanOutOnWriteService` の中には、様々な `deliver_to_xxx` がありましたが、この deliver メソッドの内部で `FeedManager` や `FeedInsertWorker` を使って Redis にデータを追加しています。

カスタマイズのポイント

表示条件を変えるなどの単純なカスタマイズをするなら、既存の Rails コードを弄るだけで変更できるでしょう。

新規タイムラインを追加するなどの重めのカスタマイズの場合、Redis のキーを考え、Rails 側の実装、Streming の実装、フロントエンドの実装すべて漏れ無くしなくてはなりません。既存のコードを参考にはできると思いますが、特に Rails 側が難しくなるのではないかと思います。

さらにタイムラインではない何か、例えば管理者からのお知らせを表示したいとか、メタ情報を動的に変更したいとか、強制リロードさせる制御命令を送りたいとか、そういうものを実装してみても面白いかもしません。

4.5　フロントエンド

フロントエンドは既に `webpacker` 化されていますが、Rails 5.1.x の流儀からも外れ、完全に JavaScript 世界の流儀で構成されています。ファイルは `app/javascript` 以下にすべて格納されています。

CSS

css は `scss` が使われいて、`app/javascript/styles` 以下に格納されています。完璧ではないものの命名規則に **BEM**[*10]を採用しているようです。

webpack の設定で、通常は `app/javascript/styles/application.scss` をエントリーファイルにしますが、`app/javascript/styles/custom.scss` がある場合はそちらをエントリーファイルにします。デザインをカスタマイズする場合、 `custom.scss` から `application.scss` と独自定義のファイルを読み込むようにします。

バージョン 2.0.0 から `custom.scss` の優先読み込みはなくなりました。代わりに、`config/themes.yml` でテーマとして設定可能なようなので、そこで `custom.scss` を使うように設定すれば良いようです。詳しく調べてはいませんが、テーマというからにはどうやらもっと深い使い方がありそうです。

[*10] http://getbem.com/

JavaScript

`app/javascript/mastodon` というディレクトリが存在し、おそらくこれがマストドンのフロントエンドの重要なファイル群だというのはひと目でわかります。が、とりあえず一旦 webpack がどんな動作をするのか確認しておきましょう。webpack の設定を見ると、どうやら `app/javascript/packs` の中にあるファイルをエントリーファイルにして、何種類かの JavaScript ファイルを生成しているようです。いくつかあるうち、`application.js` 以外は管理画面用などの小さなファイルです。webpack の設定では、 `custom.js` を除外するような記述が書かれていますが、どうやら css の時のようにカスタマイズはここに書けというわけでは無さそうで、**これが何を意味するのかわかりません**[*11]。

`application.js` は案の定 `app/javascript/mastodon/main` を読み込んでおり、やはり `app/javascript/mastodon` が本命であることが確認できました。

さてコードの中身を見ると、完全に **React** と **Redux** で作られたシングルページアプリケーションだということがわかります。TypeScript や Flow などの型支援をつかうような **トチ狂った選択** はしていません。

`actions/` があり、`reducers/` があり、おそらく教科書の通りの Redux だと言えるのではないでしょうか。実際のコンポーネントは、 `features/` の下にそれぞれの要素ごとに実装されています。

カスタマイズしたい場合は、**まぁ・・・頑張れや・・・** としか言えません。Ruby のような便利なクラス拡張などはできません。既存のファイルをガリガリいじっていくことになるでしょう。当然アップデートの追従は大変です。JavaScript いじるときは**覚悟を持って**やる必要があるでしょう。

ブランチ戦略

ガチ運営勢になろうと思っている人のために、実際のガチ運営勢がどのようなブランチ戦略をとっているか解説します。

前提として、企業に所属してマストドンインスタンスを運用すると想定します。また同時に、マストドン本体への貢献とも切り替えやすいことも前提とします。

方針

AGPL のため、コードの公開義務があります。**何も考えずに最初から Github を使いましょう**。会社の Organization を作り、そこで公開します。もちろん、Github を使うからには開発フローはプルリクエストベースです。

本家への追従は、 **リリースタグ** の単位で行います。**リリースタグですらぶっ壊れてる**ことが多いので、更にぶっ壊れてる可能性の高い master への追従は**基本的にお勧めしません**。

メインブランチは `master` ではなくサービス名などを付けます。仮に、Organization 名を **dwango**、 サービス名を **friends.nico** とします[*12]。

[*11] どうやら過去の遺物のようです。
[*12] この物語はフィクションです。 実在の人物、組織、サービスとは一切関係ありません。

チェックアウト

dwango に tootsuite/mastodon をフォークします。コードを取得し、最新のタグからブランチを作ります[*13]。

```
$ ghq get -p dwango/mastodon
$ cd dwango/mastodon
$ git checkout -b friends.nico v2.0.0
$ git push origin friends.nico
```

Github のリポジトリ設定でメインブランチを friends.nico にします。次に、**dwango/mastodon を自分のアカウントにフォーク** し、リモートリポジトリを登録します。

```
$ git remote add masarakki git@github.com:masarakki/mastodon
```

さらに、本家をリモートブランチとして登録します。

```
$ git remote add super https://github.com/tootsuite/mastodon
```

この本家のリモートブランチは、主に**バージョンアップのため**に使います。これで、super が tootsuite/mastodon、 origin が dwango/mastodon 、masarakki が masarakki/mastodon を向くようになりました。ここで hub pull-request のコマンド[*14]を叩くと dwango/mastodon に対するプルリクエストが作れます。

本体にも貢献する場合は、別のディレクトリに clone したほうが圧倒的に便利です。

```
$ ghq get tootsuite/mastodon
$ cd tootsuite/mastodon
$ git remote add masarakki git@github.com:masarakki/mastodon
```

こちらは origin が tootsuite/mastodon を向き、 masarakki が masarakki/mastodon を向く、普通のオープンソース開発と変わらない状態に出来ます。個人のリモートリポジトリは dwango/mastodon からのフォークですが、この状態で hub pull-request を叩くと、きちんと tootsuite/mastodon に対してのプルリクエストが生成され、通常と同じ感覚で開発できます。

[*13] ghq コマンドは便利だから使いましょう
[*14] hub コマンドも (ry

本家も同時に開発する場合、同じデータベースの同じテーブルを参照していると、スキーマの定義のマイグレーションがめんどくさいことになります。その場合、 それぞれのディレクトリの .env ファイルに DB_NAME=friends_nico_development のようにデータベース名を指定してあげれば、プロジェクトごとにデータベースを分離できて便利です。

開発ブランチ

開発ブランチは、当然 friends.nico から切ります。

```
$ git checkout -b new-feature friends.nico
$ # 開発
$ git push masarakki new-feature
$ hub pull-request
```

バージョンアップ

現在、 friends.nico ブランチは、本家の v2.0.0 から派生して、独自機能がいくつかコミットされている、という状態になっています。さて、本家の開発が進み無事に v2.0.1 がリリースされたとします。こちらのリポジトリも新しいバージョンに追従しましょう。

```
$ git tag                          # v2.0.0 までしかない
$ git fetch super
$ git tag                          # v2.0.1 がある
$ git checkout -b merge-v2.0.1 friends.nico
$ git merge v2.0.1
$ # コンフリクトいっぱいなおす
$ git push masarakki merge-v2.0.1
$ hub pull-request
```

デプロイ

デプロイには本家同様 capistrano を使いたいところですが、様々な秘匿情報を使う都合上、公開リポジトリには入れられないことが多いでしょう。そのため、社内の Github:Enterprise に専用のリポジトリを作り、その中から https://github.com/dwango/mastodon の friends.nico タグをデプロイするような設定をしています。また、プルリクエストを簡単にテストできるように、masarakki/merge-v2.0.1 のようにブランチを指定して開発環境にデプロイできる仕組みもあったほうが良いでしょう。

もっとヤバイ場合

某 P 社は複数のマストドンインスタンスを運用しており、某 p 社/mastodon に p なんとか と p なんとか-music という 2 つのブランチを抱えています。

聞くところによると某 P 社は、プルリクエストベースではなく、**社内のリポジトリで開発されたものを公開リポジトリに push するだけ** という方針をとっているらしいのですが、もしこれがプルリクエストベースの開発フローに移行した場合、間違いを起こさず運用できるのか・・・。自信がないので今後の課題かもしれません。

第 5 章

マストドンを活用する

マストドンは、**分散型**の SNS であり、**ローカルタイムライン**という非常に**文脈が強い**機能を備えています。現在、多くの Web サービスが、ユーザの行動をきっかけに **Twitter に自動投稿**をする機能[*1]を備えていますが、文脈の強すぎるマストドンに同じような自動投稿をしても基本的に上手くいきません。同様のことが企業公式アカウントなど、宣伝目的が強いアカウントにも言えます。マストドンの世界はビジネスに向かないのでしょうか? そうとも限らないと思います。

マストドン世界に対して情報発信するには、マストドンの特徴をきちんと理解しなければ上手くいきません。分散型 SNS という特徴を活かし、**情報はサービス側が発信し、それを外からリモートフォロー**できるようにするべきです。つまり、サービス側が **OStatus/ActivityPub を実装**し、ユーザの行動履歴や、デイリーランキングなどの情報を**自分自身が OStatus/ActivityPub で発信**するのです。ユーザは、自分が興味のある情報を、**普段自分が使っているインスタンス**からリモートフォローします。

具体例を出しましょう。**例のアレ毎時ランキングを投稿する Bot** が、自分が使っている平和なインスタンスの LTL に 1 時間に 1 回自動投稿したら**控えめに言って地獄**です。しかし、別のインスタンスでその Bot が動いていたら、**本当にその情報を必要としている人**、つまり**熱心なファン**は、リモートフォローしてホームタイムラインで読みたくなるでしょう。重要なのは、どれだけ熱心なファンでも**自分のインスタンスの LTL に出てくるのは歓迎しない**ということです。

この構造の重要なポイントは、**リモートフォローするほどの熱心なファン**が可視化できるという点です。熱心で、情報を求める本気度が高く、忠誠心が高く、**購買意欲が高い**ファンです。下手な鉄砲を数撃ちっぱなしが通用した時代は過ぎ去ろうとしています。マストドンは、新しい時代の宣伝手法を**強制される**プラットフォームと言えるかもしれません。

企業はぜひ **Closed な自前のインスタンス**を持つべきです。その中で、宣伝アカウントや、社員ひとりひとりのアカウントを運用します。自前ドメインで Closed なマストドンを運用すれば、そこにいるアカウントはすべて**本物のアカウント**であることが証明できます。ブランド力があれば、**そのインスタンスにアカウントを持つこと**が憧れの対象になり、採用に貢献したり更にブランド力を高めることに繋がるでしょう。限られたアカウント数なので、**サーバリソースはそれほど必要ない**というのも重要なポイントです。

たとえば、**電撃文庫の作家と編集者だけが登録できるインスタンス**とかあったらめっちゃ外から

[*1] 例えば 「動画を投稿しました http://nico.ms/ms9」 など

フォローしたいよね、と言えばなるほどと思うのではないでしょうか。

5.1 マストドンミドルウェア説

サービスに OStatus/ActivityPub を実装するのは、おそらく非常に大変です。なので、バックエンドに自前で立てた専用のマストドンを置き、サービスからそのマストドンに自動投稿し、OStatus を肩代わりさせる、という方法が最もシンプルで確実に実現できるでしょう。これを**マストドンミドルウェア説**と言います。

必要なのは ID 体系を適切に設計することだけです。ミドルウェアとして使うなら、ユーザがログインできる必要はありません。ユーザ作成を自動化してアクセストークンを取得するか、もしくはコードをいじってアカウント ID を指定して API が叩けるようにしても良いかもしれません。

基本的に API サーバと Sidekiq だけ動いていればいいでしょう。フロントエンドはおろか、Streaming サーバも必要ありません。リアルタイム性も問われないので、Sidekiq がそれなりに捌けていればサーバリソースは十分です。おそらくある程度は単体の Docker で動かすことも可能で、ミドルウェアとして十分可能性があります。今後はこちらの啓蒙活動もしていきたいなぁと思います。

あとがき

今回はマストドンの本でした。このマストドンというやつ、スマートフォンアプリなどという不自由で糞なものが跋扈する現代に、久しぶりに現れた**純度 100% のインターネッツ**という感じの存在で最高に楽しいです。すごい・・・おっさんたちが熱中するわけだ!!

ご存知だと思いますが、このマストドンで人生がちょこっと一変したような感じします。中の人として前に出ていく機会が増えました。久しぶりにスピード感のある仕事もできました。会社の底力みたいなところも実感できたと思います。ボーナスも普段よりたくさん貰えました。**基本給はまだ変わってません**。

スピード感が必要な仕事は楽しいですね。いろいろ面倒なことをすっ飛ばしてリリースできるのは楽でいいです。いろいろな人が味方になってくれるのも楽しいですね。本体に出したプルリクをレビューしたのが、**本業でガチ競合**の某 P 社だったりするのも面白いです。ただ、マストドンが面白すぎるせいで、あんなに好きだった twitter をほとんど見なくなりました。twitter を見ていないと、世間でなにが起こっているのか本気で全く分からなくなります。驚きです。

第 2 版あとがき

早いものでマストドンが世界に到来してから半年が経ちました。マストドン本体もリリースを重ね、順調に進化しています。

東京でばっかオフ会が開かれてずるいという不満を受け、この夏は町会議に合わせて全国を飛び回りオフ会をしました。福岡町会議のついでに寄った鹿児島では、てげどん[*2]の人たちにお世話になり、鹿児島の夜の街を連れ回してもらって美味しいものをこれでもかというほど堪能させてもらいました。マストドンをきっかけにいろいろなところに行き、たくさんの人に会い、美味しいものを食べて納得する、という非常に健康的で文化的な素晴らしいひと夏になったと思います。**旅費が自腹じゃなかったらもっと最高だったんですが**。

最近ではマストドンユーザの中で、自分でインスタンスを建てることが少し流行の兆しを見せています。その結果、全くテクノロジーに縁遠かった人が、プログラミングの勉強を初めたりしていて、とても興味深い状況になっています。我々プログラマが、今までどれだけ「プログラミング学ぶと便利だし楽しいよ」って言っても、興味を持ってくれる人はほとんどいなかったのに、です。これをきっかけにプログラミング学習者がもっと増えてくれるといいですねぇ。

なお、基本給はまだ変わっていません。

[*2] https://tegedon.net

次回予告

次回のイベント参加は冬コミです。

この夏に会社で募集したインターンのテーマがマストドンだったので、担当者として学生の対応をしていたときに気づいたのですが、最近の学生は本当に優秀でコードを読み解く力も書く力も非常に高く、サクサクと自分たちで企画を立て実装を進めていけるのですが、どうやら **Git の使い方**にみんなだいぶ苦労しているようです。

やはり仕事としてチームを組んで実際に試行錯誤をしないと、Git のよい使い方やハマるポイントなんかはわからないんだろうなぁと思います。というわけで冬コミは **かなり実践的な Git の使い方**をテーマに本を書こうと思います。

自分が実践している Git の流儀は完全にシステマティックで、作業の手順が完全に決まっていて叩くコマンドはほんの数種類しかありません。作業をしていて間違えることや迷うこともほとんどありません。実は Git は仕組みをきちんと構築すれば簡単に運用できるんだよというのを示したいと思います。

マストドンがんばるどん

2017 年 8 月 11 日　初版第 1 刷　発行
2017 年 10 月 22 日　第 2 版第 1 刷　発行
著　者　まさらっき
発行所　NP-complete
印刷所　キンコーズ