ニコ書を支えるGit運用マニュアル

NP-complete

2012/12/31

コミックマーケット83

1　まえがき

こんにちわ。 同人サークル np-complete です。

さて今回の本のテーマは 『**Git のプロジェクトをうまく管理する**』です。みなさんもちろん Git は使ってますよね?

エンジニアが Git を使っている可能性は 120%の確率と言われますが、(github にアカウントがある確率が 100%、会社が github:enterprise を導入している確率が 20%の意味) 独りで Git を使っていて多人数でプロジェクトを回した経験が無い人や、すでに多人数で Git を使っていて頻繁にトラブルに見舞われている人もいるのではないでしょうか?

Git は Subversion に比べると非常に単純明快ですが (当社比)、分散リポジトリという性格から、Subvesion とはまた違った罠がそこらじゅうにあります。Git をうまく使いこなし、リポジトリの状態を正しく保つことは、きれいなコードを保つことと同様に重要で、プロジェクトの成否に関わってきます。

この本では、実際に仕事で Git リポジトリを運用した経験を元に、リポジトリをカオスにしないことを最優先にした運用法を説明します。

1.1　注意

1.1.1　チーム

チーム規模は、ひとつのアプリケーションに対して 3〜5 人程度を想定しています。開発拠点は 1 箇所で、リモート作業する人もおらず、タイムゾーンはひとつ、見渡せば全員の声が届くくらいの距離感です。それぞれがパラレルに機能を開発していくようなスタイルを想定しています。

ひとつの機能に数十人が関わって、ひとつのファイルを数十人が編集するような状態では、なにをやっても失敗します。アサインされるメンバーが適切になるように、アプリケーションを API とフロントエンドなどに分割するなり、追加する機能を細かいタスクに分割するなりしましょう。

リモートで作業したい場合は、skype などを繋ぎっぱなしにして、声が届くようにしましょう。タイムゾーンの問題は実際に経験がないのでどう解決すればいいかわかりません。

1.1.2　絶対的な銀の弾丸はない

基本的に D 社の社内ルール下での、Web アプリケーション開発を例にして話を進めます。他の業種や、他の会社の社内ルールには合わない部分が多々あると思います。

この本に書かれていることは、 **最高の方法ではありません** し、**誰にとっても正しい方法ではない**です。

しかし、**「どのような問題があって、何を懸念して、どのように対処したか」**という、リポジトリ管理の考え方は、たぶん一般的に参考にできる内容だと思います。この本を参考に、チーム独自のリポジトリ運用方法をアレンジしていく、という形が最適だと思います。

2　Git の特徴

2.1　分散型

Git は分散リポジトリ型です。分散型には Subversion のような中央リポジトリはなく、リポジトリの完全な機能は個々のコンピュータにあります。以降、個々のコンピュータの中にあるリポジトリのことを**ローカルリポジトリ**と呼びます。

しかし、普通はプロジェクトの成果を共有する必要があるので、中央リポジトリと同じような立場の **共有リポジトリ**を用意するケースがほとんどだと思います。共有リポジトリは自分でサーバを立てたり、github に持ったりするでしょう。

その場合でも、共有リポジトリはただ共有するためでしかなく、コミットなどの作業はやはりローカルのリポジトリへ行うことになります。

2.2　分散型のメリット

分散型リポジトリでは、ローカルリポジトリへの操作で、共有リポジトリが変更されることはありません。そのため、

- ネットワークにつながっていなくてもリポジトリ操作ができる
- ネットワークを介さないので基本的な操作が速い
- ローカルで完結するのでコミットがしやすい
- ブランチが圧倒的に使いやすい
- ブランチの切り替えがしやすく、つまり作業の切り替えがしやすい
- 共有される前ならいくらでも操作のやり直しや歴史改変ができる

というメリットがあります。

2.3　分散型のデメリット

共有リポジトリが変更されないという事は、明示的に共有の操作をする必要があるという事です。多くの場合、共有リポジトリとローカルリポジトリには乖離が発生し、共有操作で少しずつ解消されていきます。Git では、この共有操作が一番難しく、これを完全にコントロールすることが重要です。

共有されているコードとされてないコードの区別を常に意識し、まだ共有されてはいけないコードが共有されないようにする必要があります。

3　失敗しない Git プロジェクト運用

Git がどのように使われ、どのようなミスが発生するのかを明らかにし、間違いを起こさない Git 運用法を作ります。

3.1　リリース戦略を考える

まず最初にリリース戦略を考えます。サーバに配置 (デプロイ) したり、リリースビルドを作ったりするのがリリース作業です。

プログラミングの最終目標はリリースです。リリース戦略から考えることで、作戦の方向性を間違えないようにします。

3.1.1　リリースを自動化する

手作業のリリース作業は絶対にしてはいけません。Git のコードを自動でリリースする仕組みを作りましょう。自動化することで作業の属人化を防ぎ、事故を減らすことができます。継続的なリリースにも自動化は欠かせません。

ruby には **capistrano** というデプロイツールがあり、広く使われています。仕組みとしては単純なので、Web アプリであれば ruby 以外の言語でも簡単に使うことができます。Chrome Extension のための crxmake など、めんどくさがりが多い rubyist は、いろいろビルドツールを作っていますので探してみてください。

よさそうなツールが見つからなくても、簡単な Makefile を書けば十分実用的なビルドツールが作れます。

- Git からリモートの master ブランチを毎回チェックアウトする
- コマンド一発で全部終わる
- 誰がやっても同じ結果になる 以上の条件を満たせば、十分に実用的です。

リポジトリに push したら自動でアプリケーションが更新される、**heroku** のようなかっこいいプラットフォームもあります。ごく普通の Web 開発なら heroku を使うのが一番楽なんじゃないかと思います。

3.1.2　リリースの種類を整理する

リリースにはいくつか種類があります。パッケージアプリにはデバッグビルドとリリースビルドがあり、Web アプリには本番環境と開発環境などがあります。

全てのリリース形態に対して、自動でリリースできるようにします。capistrano の場合、複数のデプロイ環境を持つことができ、それぞれ別のブランチや設定を指定できます。

Web のデプロイ環境例　Web アプリの場合、最低でも次の環境が必要です。

production (本番環境)　実際にユーザがアクセスする本番環境です。データは基本的にユーザが登録していくものだけです。「本番でテストデータ 500 件登録させてくださいー」とか言ってくるバカ (実在) は全員ぶっ殺しましょう。

staging (本番と同じ状態の開発検証環境)　本番環境と同じコードが動いている開発検証環境です。本番環境で出たバグを再現させて検証/修正したり、API を提供している場合は他チームの開発環境から繋いだり、「本番でテストデータ 500 件登録させてくださいー」とか言ってくるバカ (実在) に「staging でやれ」って言ったりします。

develop (開発用環境)　開発者が開発用に自由にいじる環境です。開発中の機能を確認したり、仕様バグを洗い出したりします。常に新しいコードがデプロイされるので不安定になったりもするので、他チームからはあまり繋がせるべきではないと思います。「dev 動かなくなってるんですけどー困るんですよねーこっちも開発止まっちゃうんでー」とか言ってくるバカ (実在) には「staging でやれって行ってんだろクズ!」って言っておきましょう。

3.1.3　リリースを踏まえたリモートブランチ運用

Web の場合、最低でも上記 3 つの環境が必要になります。これらの環境にどのブランチをデプロイするか決めます。

production と staging は、動いているコードなので **master ブランチ**をデプロイします。develop には開発中のコードが入るので、master ブランチではありません。**develop ブランチ** をデプロイすることにします。

他にも、大規模な変更がある場合、専用の開発環境とそれに対応するリモートブランチを作ります。ミドルウェアのバージョンアップや、なぜか頻繁にあるサイトリニューアルなどの場合です。扱いは develop 環境と同じです。

3.2　ブランチルールを考える

多人数で Git を使う時、一番ミスが起こりやすいのはブランチ操作です。ブランチのルールを決めることでミスを減らしましょう。

3.2.1　なにを避けたいか

本番環境にリリースされるのは master ブランチです。この master ブランチに糞を絶対に混ぜてはいけません。master ブランチは歴史改変を基本的に

するべきではないので、一度間違った状態になった master ブランチはずっと黒歴史として残ってしまいます。

ブランチ運用で起こしがちなミスを分析し、ミスが起こらないブランチ運用を目指しましょう。例えばこのようなミスが起こります。

入ってはいけないコードが入る　まだ動かないコードは絶対に master 入れてはいけません。当然気をつけると思いますが、ブランチを切る際のミスで簡単に起こってしまいます。

例えば、A ブランチから B ブランチを作り、B ブランチでひとつの機能を完成させたとします。B ブランチで作られた機能は十分にテストされているので、安心して master にマージするでしょう。この時、B ブランチだけをマージするつもりが、派生元の A ブランチの内容までマージされてしまいます。もし A ブランチが全然未完成だったらどうなるでしょう?

消してはいけないブランチを消してしまう　そんなミスしねえよと思うでしょうが、実際は気をつけないとよく起こります。例えばリニューアル案件のために renewal というブランチを共有していたとします。一旦入れた開発中のコードを取りやめるときに、development ブランチと同じ要領でぶっ壊して作りなおしたりしたくなるでしょう。この時、もし他の人が renewal は master と同じように開発が終わったコードのみを入れるブランチだと思っていたらどうなるでしょう? もしかしたら renewal にマージしたブランチはすでに消してしまっているかもしれません。チーム内で認識の相違があると、いつの間にか他の人のブランチを消してしまうことになってしまいます。

後戻りできない　何度でもやり直せることは重要です。問題を含んだリリースをしてしまった時、すぐに過去に戻れば問題は一旦解決します。余裕を持って修正して再リリースすれば良いのです。もしやり直しできない状態になってしまったら、問題を含んだまま急いで修正しなければなりません。これは非常にストレスがかかります。

3.2.2　git-flow と問題点

git-flow という、ブランチ運用ルールをサポートするツールがあります。
ブランチルールの特徴を簡単に説明すると、

- master の他に開発ブランチとして develop ブランチがある
- 新機能 (**feature**) は develop から作って develop にマージする
- 修正 (**hotfix**) は master から作って master と develop 両方にマージする

- リリースは develop から作られて master にマージされる

という流れになっています。
git-flow は、この操作を

```
$ git flow feature start topic_A
$ git flow feature finish topic_A
```

などのコマンドでできるようにするツールです。
git-flow はかなり人気のあるツールですが、実際に使うとたくさん問題点があります。まず、どうしても develop が不安定になります。開発ブランチというものは得てしてそうなります。エンジニアに負荷をかけないために、無責任なブランチは当然必要なのですが、それは master から隔離されるべきです。
一番の問題は、その不安定な develop から feature ブランチを切って開発しているところです。不安定な develop を開発の本流としてしまっていることに問題があります。
本流が不安定なので、リリースブランチなどという邪道な運用が必要になります。いったいこのリリースブランチはなにをすればいいのでしょうか? 不安定な開発ブランチの体裁を整えてリリースできるようにするのでしょうか? まるで Subversion やヘタしたら cvs の時代のリリースモデルのようです。
その他の問題として

- github の PullRequet ベースの開発と相性が悪い
- finish するとブランチがすぐ消されてしまうのでやり直しが効かない

などがあります。

3.2.3 ブランチルールの例

実際に運用して上手くいったブランチ運用ルールを例として示します。

1. ブランチは master から切る まず、絶対的に **正しいコードは** *master* **ブランチにある**ことを思い出しましょう。
基本的に、 すべてのブランチは **master ブランチから** 切ります。そうするとそのブランチは、現在動いているコードから、自分がコミットした以外には何も変更されていないことになります。もちろん、この状態は変更をテストしやすく、見通しが良い状態です。

2. 全ての変更はトピックブランチにコミットする　トピックブランチとは、ひとまとまりの機能を作るための専用のブランチのことです。機能ごとに適切な名前のトピックブランチを作り、作業は全てトピックブランチにコミットします。つまり、 *master* **ブランチに直接コミットしてはいけない**ということです。

一つのトピックブランチに目的の機能と関係ない変更を加えるのは基本的にしてはいけません。目的の機能のために、基礎となるクラスに変更を加えるなどの場合、もしそれがフレームワークの普遍的な部分だとしたら、やはり別のトピックブランチで作業をする方がいいでしょう。その場合、フレームワークへの変更を先に master にマージし、トピックブランチを master から rebase します。

3. リモートブランチにも直接コミットしない　リモートブランチ とは、他の人と共有しているブランチです。基本的に一つのトピックを複数人で開発するべきではありません。とは言っても、コードを書くプログラマと html を書くデザイナが協力してひとつの機能を作る、などのように役割が分担されていたりする場合は、一つのトピックに二人の作業がコミットされることになります。このような場合に、トピックブランチをリモートブランチとして共有します。

(自分が作る場合)

```
$ git checkout master
$ git checkout -b topic_master
$ git push origin topic_master # リモートブランチとして共有される
```

(他の人が作った場合)

```
$ git checkout -b topic_master origin/topic_master # リモートからローカルにコピー
```

このリモートブランチは、トピック開発という小さな世界の master ブランチのようなものなので、master と同様に直接コミットしてはいけません。**トピックマスター**とでも呼ぶことにします。自分の作業は、トピックマスターから更にトピックブランチを作り、そこにコミットします。作業が終わったトピックブランチはトピックマスターにマージし、共有します。

```
$ git checkout topic_master
$ git checkout -b topic_A      # 更にトピックブランチを作る
$ (commit)
$ git checkout topic_master
$ git merge --no-ff topic_A    # topic_master に自分の topic_A をマージ
```

```
$ git push origin topic_master # 更新を push
```

4. リモートブランチはいつでも破棄できる　master 以外のリモートブランチはいつでも **破棄**してリセットすることができます。なにかよくわからない状態や試行錯誤のあとが残ってグッチャグチャになった場合、対処に無駄な時間を使うより、何も考えずに破棄して作りなおしたほうがいいでしょう。本章のルールは、全体的にリセットとやり直しが容易にできることを目指して作られています。

リモートブランチを作りなおすには次のようにします。

```
$ git push --force origin master:topic_master  # master で topic_master
を書きかえる
```

その後、他の人に対して、リモートブランチを作りなおしたことを教えてください。単純に pull しても上手くいかないので、トピックマスターを消してチェックアウトしなおすか、次のようにリセットします。

```
$ git fetch --all                    # リモートの情報を更新する
$ git checkout topic_master
$ git reset --hard origin/topic_master  # リモートの topic_master
にリセット
```

リモートブランチはいつでも破棄される可能性があるので、開発したトピックブランチは、master にマージされるまで消してはいけません。リモートブランチで共有したから、トピックが完了したらきっとまるごと入れるんだろう、などと期待してはいけません。

master に入るその日まで、自分がした仕事は自分が責任を持って管理しておきましょう。消していいブランチは次のコマンドで抽出できます。

```
$ git checkout master
$ git branch --merged
```

5. master にマージする前に履歴を整理する　開発環境にデプロイすることで発覚した間違いなどは、トピックブランチに修正コミットを加えることで修正します。その後、develop ブランチにトピックブランチをマージし、修正コミットを適応した状態で再度デプロイします。

この修正コミットは、コミットとして残す意味はないので、トピックの開発が終わった段階で歴史を改変し、なかったことにします。

```
$ git rebase -i master
```

で、コミット順の変更や、コミットをまとめる (**squash**) などの歴史改変が可能です。

主観的でも全然構わないので、一つ一つのコミットに意味と意図が見えるように、コミットの単位を意識して開発しましょう。

6. master にマージしていいのはトピックブランチだけ　master にマージしていいのは基本的に個人の手元にあるトピックブランチだけです。トピックブランチは master から切られているので、そのまま master に戻せばゴミが混ざってないことになります。develop ブランチや、トピックマスターなどの共有されたブランチはマージしてはいけません。

例外として、マージ専用に **新しく作った**共有ブランチなら master にマージ可とします。これにより、複数人で開発したトピックマスターと同等のブランチを作ることや、リリースする内容を一旦まとめた **release ブランチ**を作って、それを master にマージすることができます。

7. トピックブランチをマージするときは –no-ff　トピックブランチが、最新の master からいくつかコミットが追加された状態のとき、**fast-forward** な状態と言います。fast-forward なブランチはマージしやすく、見通しはいいのですが、一つ重大な問題があります。

fast-forward なトピックブランチを master にマージすると、あたかも master がトピックブランチの場所に移動しただけの状態になり、その開発にトピックブランチがあったことが忘れ去られてしまいます。

それを防ぐために、トピックブランチをマージするときは –no-ff オプションを使います。 マージ自体がひとつのコミットとして扱われます。`$ git checkout master $ git merge --no-ff topic_A`

リリースブランチなどにトピックブランチをマージする場合も、 –no-ff をつけます。その後、リリースブランチ自体を master にマージするときは、つけないほうがいいでしょう。リリースブランチは、それが存在した意味は無いからです。

```
$ git checkout release
$ git merge --no-ff topic_A
$ git merge --no-ff topic_B
$ git checkout master
$ git merge release
```

github で PullRequest をマージすると –no-ff と同じ処理がされます。なので、github で PullRequest ベースで開発をするときは、リリースブランチを PullRequest するような方法はあまり推奨しません。トピックブランチ一つ一つを PullRequedt しましょう。

4　ワークフロー例

前章のルールに沿った開発風景の例です。

4.1　準備

まず master ブランチは最初から存在しているはずです。開発環境にデプロイするために、develop ブランチがない場合は作りましょう。

```
$ git branch -a
```

でリモートブランチを含んだブランチの一覧を表示します。develop がなかったら、

```
$ git checkout -b develop
$ git push origin develop
```

で develop ブランチを作って push します。
すでにリモートに develop ブランチがある場合は、

```
$ git checkout -b develop origin/develop
```

で develop ブランチを手元に持ってきます。

4.1.1　github を使っている場合

github を使っている場合は、チームのリモートリポジトリを直接触るのではなく、プロジェクトを自分のアカウントに fork して、それを clone します。

```
(github の Web で fork)
$ git clone git://github.com/team_account/project.git
$ cd ./project
$ git remote add my_account git@github.com:my_account/project.git
```

hub コマンドが使えるなら、

```
$ hub clone team_account/project
$ cd ./project
$ hub fork
```

でも同じことができます。
今後、origin のリモートリポジトリにしていい操作は、基本的に、develop を push することと、master を pull してくることだけです。他の操作は my_account のリモートリポジトリに対して行い、origin への変更は全てプルリクエスト経由で行います。

4.2 開発開始

まず新しい機能を作り始めるときは、ブランチを切ります。

```
$ git checkout master
$ git checkout -b topic_A
```

それなりにコミット単位を意識して、トピックブランチにコミットしていきます。あとから複数のコミットをくっつけるのは簡単ですが、1 つのコミットを分割するのは大変です。なるべく細かい単位で、複数意味の変更が一つのコミットに入らないようにしましょう。

4.3 開発環境にデプロイ

トピックブランチの開発が終わり、自動テストで十分テストできたら、開発環境にデプロイし、実際の動作や特にテストでは書きづらい見た目周りの確認をします。開発したトピックブランチを develop ブランチにマージし、デプロイします。

```
$ git checkout develop
$ git merge topic_A
$ git push origin develop
$ cap dev deploy          # 例えば cap を使う場合
```

develop ブランチは、開発者全員からトピックブランチを頻繁にマージされる、総受けブランチです。この段階でマージされるコミットは、どれだけ汚くても構いません。

develop ブランチにマージした際のコンフリクトは、今の段階では、develop ブランチ上で解決し、 適当に fix conflict とでもコミットしておけば大丈夫です。ただし master にマージする際に手間がかかるようになると覚えておきましょう。

試行錯誤のコミットや fix としか書かれていないコミットメッセージでも全然問題ありません。納得行くまでコミットとデプロイを繰り返し、トピックを改善していきます。(もちろん改善コミットはトピックブランチで行います。)

4.3.1 develop ブランチをぶっ壊す

開発を続けていくと、develop ブランチは非常に汚いブランチになります。こうなった場合は、綺麗サッパリ作り直しましょう。リモートの develop ブランチを消し、もう一度 master から作り直します。作りなおした develop ブランチに、トピックブランチを歴史改変で整理してからもう一度マージすれば、今度は以前より綺麗な develop ブランチになります。

```
$ git push origin :develop    # 空白のブランチで develop を上書き
$ git checkout master
$ git checkout -b develop
$ git push origin develop     # master から develop を切り直してリモートブランチに
$ git merge topic_A ; git merge topic_B ...
$ git push origin develop
```

作業の前にはチームメンバーにひと声かけましょう。

4.4 master にマージする

開発が終わったら master にマージして、コードを最新にします。

4.4.1 マージのタイミングはいつ?

master にマージするタイミングは、リリースのスケジュールと深く関わってきます。「master ブランチは今動いているコード」というルールを思い出してください。つまりこれは、master にマージしたらそのコードはすぐリリースされなければいけない、すぐにリリースできないのであれば master に入れてはいけない、という事を意味します。

リリースのタイミングは会社やプロジェクトによって様々だと思います。

頻繁に本番リリースが許されている会社の場合、開発が終わったトピックブランチ一つを master にマージして、すぐにデプロイするという方法が取れます。

1 イテレーション (例えば 1 週間) に 1 回、定期リリースをするような運用方法の場合、リリース直前に全てのトピックブランチをマージします。

4.4.2 master の最新を適応する

トピックブランチのマージ前に、最新の master の変更を適応します。

```
$ git checkout master
$ git pull --rebase origin master
```

トピックブランチのコミットが、最新の master から始まったことにするために、rebase をします。

```
$ git checkout topic_A
$ git rebase master
```

rebase でコンフリクトが起こった場合、コンフリクトを解決し、

```
$ git add .
$ git rebase --continue
```

でリベースを続けます。

4.4.3 履歴を整理

```
$ git rebase -i master
```

で squash して、ある程度履歴を綺麗にします。意味のあるコミット単位にしましょう。

github でオープンソースに PullRequest を送る場合、全ての変更を一つのコミットにまとめるように要求しているプロジェクトが多くあります。その場合は「squash してください」と言われるので、全てのコミットを squash して一つにまとめます。

もちろんこの場合も、一つのコミットでやりたいことを過不足なく表現する必要があります。トピックブランチではトピック以外の作業を絶対にしてはいけない、というルールに慣れると上手くできるようになると思います。

4.4.4 マージする

マージする瞬間は特に注意しないといけません。

定期リリース型の場合、1 回のリリースに複数の人がトピックブランチを持ち寄ることになります。この場合、全員のトピックブランチのマージが上手く行かない事や、単体ブランチのテストは上手く行っているが全部合わせた時のテストが失敗する場合があります。もし直接 master にマージしていたら、master に一瞬テストが落ちるという状態が発生してしまいます。

その状態を避けるために、いったんリリース予定のトピックブランチを全て集めた、**release ブランチ** を作るといいでしょう。もちろん release ブランチは master から切ります。

```
$ git checkout master
$ git checkout -b release
$ git push origin release
```

全員が以下を繰り返す

```
$ git pull origin release
$ git merge --no-ff topic_A
$ git push origin release
```

すべてのトピックブランチを release ブランチにマージし、コンフリクトが起こったら解決し、自動テストが全部通るのを確認します。テストが通ったら、新たに master から切り直した develop ブランチに release ブランチをマージし、開発環境にデプロイして動作確認します。動作確認が正常に終了したら、release ブランチを master ブランチにマージし (この時コンフリクトは絶対起きません)、本番環境にデプロイします。 デプロイが終わったらタグをつけておきます。

```
$ git checkout master
$ git merge release
$ git push origin master
$ git tag release_hogehoge
$ git push --tags origin master
```

4.4.5 github で PullRequest ベースの開発の場合

origin ではなく、自分のアカウントにトピックブランチを push し、その PullRequest を作ります。PullRequest は全員でチェックして、送ったのとは別の人がマージのボタンを押します。

```
$ git checkout topic_A
$ git push my_account topic_A
(PullRequest)
```

他の人の PullRequest をマージした後、自分の PullRequest がコンフリクトする場合、コンフリクトを解決した PullRequest で更新しなければいけません。master を最新に更新し、トピックブランチを master から rebase し (ここでコンフリクトを解決)、**いったん develop を作りなおして** マージし、検証から再開します。

```
$ git checkout master
$ git pull --rebase origin master
$ git checkout topic_A
$ git rebase master
(コンフリクト解決)
$ git push --force my_account topic_A
(PullRequest 更新)
```

4.5 後始末

リリースが無事に終了したらトピックブランチを削除しま ･･･ せん! 何かが起こった時のために、最低 2 回分くらいのトピックは手元に残しておきま

しょう。
もしリリースに問題があった場合は、まず真っ先に正常に動いていた頃のコードに戻します。デプロイツールがサポートしている場合は、その機能を使います。

```
$ cap production deploy:rollback
```

それができない場合は、前回のタグをチェックアウトして元に戻します。本来、master の歴史改変はやってはいけない作業ですが、こういう状態の時は仕方ないでしょう。master はちゃんと動いているコードであるべきです。

```
$ git checkout master
$ git reset --hard old_tag_hoge
$ git push --force origin master
```

リセットしたらとりえずデプロイします。前回と同じコードになったはずなので、問題なく動作するはずです。サービス停止時間を最小限に抑えました。
元に戻したことを全員がリポジトリに反映させます。

```
$ git checkout master
$ git fetch --all
$ git reset --hard origin/master
```

全てのリリースを先延ばしにするなり、問題のないコードだけを一旦リリースするなり決定し、今回のリリースは終了です。

5　おまけ: リリース先輩

実は弊社の公式中央リポジトリはSubversionです。最近ではgithub:enterpriseが導入されましたが、それ以前もチーム単位で Git サーバを運用していた例もありました。しかし現在でも最終的にコードは Subversion に置かなくてはいけません。

でももちろん、いまさら馬鹿正直に Subversion なんか使いたくありません。git-svn も使ってみたけど混乱が増すだけです。

5.1　スクリプト化

5.1.1　Subversion を自動化

とにかくコミットさえすればいいので、スクリプト化してみました。

- Subversion から trunk のコードを ./svn に取ってくる
- git archive で master を zip でとってきて ./svn に展開する
- ./svn が更新されているので add したりして commit

すごく乱暴な方法ですが社内ルールは十分に満たせています。 ポイントは git archive で上書きしているところでしょうか。Subversion の機能を完全に無視してる感じが最高にクールです。

5.1.2　リリース wiki を自動化

社内ルールで、リリース内容や作業手順をまとめた wiki ページを作らなければなりません。それを自動化するため、変更内容から wiki ページを自動で生成しています。

- git diff で前回と config の差分を出す
- git diff で前回の migration の差分を出す
- 変更のあるサブプロジェクトを列挙する

さすがに変更内容などは手作業で入力しますが、コミットメッセージになにかルールづけすれば可能かもしれません。

その他にも、diff を見る trac の url を生成し、リリース wiki の url と一緒にチームの IRC チャンネルに送信します。

5.2 先輩化

リリーススクリプトは誰でも実行できるはずですが、それでも属人性が完全に失われたわけではありませんでした。日常的に ruby を使っていないと、その実行環境を作ることも難しいのです。そこで、IRCbot としてリリーススクリプトを動かすことにしました。ついでに、通常はストレスがかかるリリース作業が少しでも楽しくなるように、スクリプトを擬人化しました。それがリリース先輩です。

IRC でリリース先輩に話しかけるとリリース作業してくれます。

```
(masarakki) release_senpai: 先輩お疲れ様です!
(release_senapi) masarakki: おう、おつかれ
(masarakki) release_senpai: 先輩! リリースおねがいします!
(release_senpai) おう、待っとれ


....
(release_senapi) リリースできたぞー
(release_senpai) wiki かけよー http://...
(release_senpai) diff はこっちなー http://...
```

こんな感じです。リリース先輩のプロセスが死に IRC から退出すると、「うわっリリース先輩バックレた!」みたいな悲鳴が上がる楽しい職場になりました。ちなみにチーム内のヒエラルキーは、jenkins 先生が頂点、リリース先輩がその次、人間はそれ以下となっています。

6　あとがき

第 3 弾です。 本当にギリギリに完成しました。 現在 12 月 27 日の 22 時です。

6.1　markdown でやってみた

前回、TeX とかもう死ねばいいのにと思ったので、今回は markdown で書いてみました。しかし本にするには markdown から印刷できる形式にしないといけません。さて markdown からどうやって印刷できる形式にすればいいのでしょう? pdf にできさえすれば一番簡単に行けそうです。 調べてみましょう …

いろいろ方法はあるようですが、うまくいったりいかなかったり、日本語対応してなかったり、確実にできそうな方法は …

結論:　markdown を tex に変換して更に pdf にする

orz …

6.2　近況とか

5 月くらいに完全に Windows を捨てました。ほとんどの生活は不自由なく過ごせるのですが、やはり SAI とコミスタがないと漫画は描けませんねぇ。あとコミケブラウザが使えませんでした。夏コミ 1 週間前に気づいて死を覚悟しましたが 3 日で Web アプリ版を作りました。すげーもんができたと思ったら今回から公式で Web アプリ対応されて orz … でも俺が作ったやつのほうが使いやすいもんね …

6.3　告知とか

ニコ書がリニューアルして角川以外のコンテンツも扱い始めました。いっぱいお金落としてください。

http://seiga.nicovideo.jp/book/

奥付

ニコ書を支えるGit運用マニュアル

発行
NP-complete
http://np-complete-doj.in

著者
まさらっき
http://twitter.com/masarakki
http://github.com/masarakki

発行日
2012/12/31