「PHP、バージョンアップしナイト！！」

注：15時は昼間です

PHPバージョンアップけもの道

at PHPバージョンアップ kickoff - 2021/07/15

@uzulla

自己紹介

- uzulla（うずら）
- PHPが好き
- 新規も改修もやるPHPerです
- varや、&=にも対応PHPerです

最近PR TIMESさんのお手伝いしてます

- CTO「レガシー対応PHPerで、パッと来れる人が他に思いつかなかった」
- 私「わかる、俺も知らない」

最初にまとめ

- PHPバージョンアップは「作業」である
- 複雑に見える問題は分解し単純に、やりきれる手法で実行する
- テストが充実していなければ、新旧比較するE2Eテストを推奨
- 実はコード修正は難関ではない、段取りが8割。魔法はない
- 全員で解決すべき事、辛い作業なので周囲はサポートしよう
- 事故はおこる、なにをやってもおこる、見極めや覚悟が必要
- 最後マインドが変われば成功、Ver UPを日常にする、礎を築く

はい

どうしてPHP(ランタイム)をバージョンアップ？

- (私感ですが…)
- EOLなので
- PHPの進化が使えないので
  - 最近は言語も機能も速度もツールも段違いに良くなったのにどれもこれも使えない。つらいし、気分が盛り上がらない

「みんなバージョン上げてるのか？」

- 私感ですが…
- PHPメインの所はv7.2、v7.4位には上げている所が多い
  (v8はまだこれからのイメージ)
- 一方v5.4の頃にPHPの勢いを信じられず、停滞した現場も多い
- v8.1も出るし、今もPHPは加速度的に進化している、
  追いつきづらさはさらに加速する

PHP version, Age Night ...

- だから、いますぐあなたもVersion UP!!
- **本トークは、上げてない人が対象です**
- 本トークは、PR TIMES内のエンジニアさん向けです
- (実状とフィットしてないかも、ご了承ください)
- 今日飛ばした範囲は聞いてくれたら個別にいくらでもやります
- (例的なものは一般論で、PR TIMESさんの現状ではありません)

ここでいう「バージョンアップ」とは？

- 色々なところで語られる「PHPバージョンアップ」は「リファクタリング」と混同されている
- **バージョンアップはリファクタリングではない**
- (今回、リファクタリングの話はしない)
- リファクタリングに近い事を「必要なので」行う場面はある

PHPバージョンアップは
レガシーシステム改善の一環

(ここを間違えないように気をつけて)

(なお、リファクタリングも
レガシーシステム改善の一環)

実践編の前に

- アイスブレイク的に、携わる人や関わる人がもつべき心構え等...

必要なもの

やる気 (Motivation)

時間 (Schedule)

度胸 (Brave heart)

やる気

やる気こととモチベーション

- これはヒットポイントです、なお数値は見えない
- 「戦うと減ります、無理するとゼロになります」
- 大きければどうにかなるものではないが、
  少ないとどうにもならない。自己・周囲での管理が必要
- 「回復したり、増やせます」
- 回復の必要・不必要は論じる価値なし、必要である
  まずは許容、次に応援、そして理解と支援をしていく

「レガシーシステム改善やりたくない」問題

- 「一度しかやらない意味のない作業」
  - NO、別実装でも、他言語でもまたやります
- 「勉強にならない」
  - NO、システムや設計に詳しくなります
- 「新しい技術(?)で実装したほうが潰しがきく」
  - NO、新しい技術(?) も、こうなります。前向きに捉えましょ

時間

説明不要だが、時間は必要

- 時間＝人月≒機会損失▶「最低限にしよう」▶リスケ▶断念
- 見積困難作業なのに、なぜか短時間で見積もりがち
  挙げ句、短期のスケジュールが組まれることが多い…
- 「新規で作ったほうが工数が読める」
  「それは『新規で作った経験』が多いだけでは？」
  (まあ、本当に新規で作ったほうが良い場合もある)

一番融通がきくのは大抵時間

- だが、無駄に時間を伸ばしてもダレて、結果がゼロになる
  効率が悪くとも、手戻りがあっても、他をブロックしても、
  小さい歩みでも、重ねて、継続し、やりきる必要がある
- **やりきる必要がある** (大事なので二度言いました)
- できるのは「一番短い時間を見積もる」事ではなく、
  「短くなるように試行錯誤する」ことである

度胸

度胸と覚悟

- バージョンアップをしたいプロダクトは絶賛稼働中
- 止まる、壊れるサービス。保証は不可能
- 手間暇かけてカナリアデプロイとかするのも大変
  （まあ、ちゃんとする場合はするんだけど）
- 気づかないデータ破損…ウッ頭が…
- だが恐れていては進めない、「全員」に度胸と覚悟が必要

「失敗を許せって話か？(笑)」

- そうじゃない！もう失敗してる！目をさまして！！
- 「触らぬシステムにたたりなし」で、今がたたりですよ！
- つまりさらに失敗するだけです！
- (気が済むなら)太陽のせいにしてもいいからやるぞ
- 度胸もって行きましょう

皆さんの顔色が変わった所で

- (喜んでる人も、具合が悪くなった人もいるでしょうが)
- 実践的な話に入っていくまえに勢い良く言いました
- ポエムは終わり、ここから実践に入っていきます

実践といっても

- 「高度なテクニック」なことは言いません
  - (ハイテクを使うなという意味ではない)
- 今回は「できる」に全振りします
  - (その上で、つかえたら使いましょう)

目次

- 課題設定
- E2Eテスト
- サンドボックス
- マイグレーション作業
- 南無三デプロイ

PHPのバージョンアップとは？

※「リファクタリング」ではない

課題：PHPのバージョンアップ

施策：バージョンをアップする

？ ？ ？ ？ ？ ？ ？

(ご存知の通り、単にPHPランタイムのバージョンを上げると99%壊れます)

「バージョンアップできますか？」

- レガシーシステムの現場で、運良くざっくばらんに現場の方と会話した時に見聞きした、私の知る一般的な認識
- 「できるよ」 ▶ ごくわずか
- 「できないよ」「やらないよ」 ▶ 9割以上

なぜ「できない」かを私見で総論すると

- 「大変すぎるから」「意味がないから」

  ▶ 作業する人材、コスト、モチベーションが確保できない

- 「『やり方』を知らないから」

  ▶ 『正しい』手法やツールの存在を信じている

- 「新規でつくるほうが良いから」

  ▶ 経験者に多い。新規の方が見積もりをしやすいのはそうですね

ここでは「やり方を知らない」にフォーカスします

- なにせ、前提として、それをやらないといけないんでね！

問題を解くには適切な課題設定が必要

- 「これをバージョンアップして」 ▶ 見積もり困難に見える
- 「このテストを通るようにして」 ▶ 可能に見える
- なので、テストに落とし込む
- **新環境と旧環境のズレ(不具合)を問題にする**
- そのズレのリストを「作る」のは**可能といえるのではないか？**
  (問題の分割)

大変そうすぎる！

- （気持ちはわかる）
- しかし、「わかる所まで、ブレイクダウンする」のは王道
  - 解法は１つではないし、沢山ある
- 唯一の正解はないが、事例や情報は探せば沢山ある
  - どんな事でも、前例は「正解」ではない
  - あくまで参考にして、やっていく

「実例！改善の様子！！」

```
foreach($課題リスト as $課題){
```

「この機能・フローが動かない」 ▶ （原因調査の開始）

「このURLが動かない」 ▶

「このコードのこの行が動かない」 ▶

「関数がなくなっている」 ▶ （原因調査の達成）

「よし！しらべて書き換えよう！」 ▶（施策の開始）

「テストしてうごいたぞ！」▶ （達成）

```
} # これの繰り返し、これが入れ子になる
```

一般的な現場において

- 個々の問題(修正作業)は、皆さん解決できる
  - 規模や、難しい問題もあるけど、出来ない話ではない
    (PHP得意（自称）なので手伝います)
- しかし、そこにたどり着くのが色々難しいんですよね...
  - (自信がない？わからない？手伝います、そのために来ました)

ではどうやってやっていくか？

(一般論です、PR TIMESにおいて「こうする・すべき」という話ではない)

E2Eテスト（End to End テスト）

E2Eテスト（End to End テスト）

- 入力(リクエスト)に対し、期待通りの出力(レスポンス)か？
- 新環境に実行して成功・失敗がわかる
  (そして、現行にもかけられる)
- 「バージョンアップ後、E2Eテストが通れば成功」と言える
- (…理論上の理想で言えば。例外もあります)
- ブラウザで手動でも可能、日々皆さんしてると思います。

しかし結合テストは辛い

- 修正したら毎回フルテストしたいが、人力は大変
  - ▶ テストは大変、できるだけ自動化しよう
- 「自動化といえば○○をつかおう！」の罠
  - 一つでも多くのテストを作るべきなのに
    「テスト書くのが ~~面倒~~ 大変」問題
  - ▶ ツールで解決はしない(経験上、悪化しやすい)

重要なので二回いいます

- **テストがたくさんあることが重要**
- 「かっこいいツール」は重要ではない
  - テストのモダンさ、綺麗さは短期には意義が薄い(主観)
  - （かっこいいツールが便利なこともある）
- （多くの場合）「Req/Res だけ検証できれば良い」
  - 慣れたPHPとタスクランナーで書ける

じゃあ素朴なツールとは？の例

- symfony/http-client + symfony/dom-crawler + phpunit位でいい
- (HTMLの完全一致は無茶なので)
  レスポンスコード、ヘッダー、DOM等の確認
- 「作業」なので、「かっこよい」必要はない
  極論いえば、curlとシェルスクリプトだっていい
- (JSがないと動かない？

phpunitだが、ユニットテストではない

- phpunitをタスクランナーとしてつかう(PhpStorm連携便利)
- **CI等での利用性・将来性は一番の問題ではない**
  リファクタリングではない、バージョンアップだ
- ロジックの健全性テストをかきたいわけではない
- なので、この「テスト」は、アプリと別(のrepo等)で良い
  (というか、分けないと古いPHPを使う事になる…)

なぜE2Eテスト用ツールではないのか

- ヘッドレスブラウザより、**実行速度が速い**
  (強いPCで並列度を上げていくと、負ける事はある)
- PHPエンジニアなら、単純に**早く書ける**
  (不慣れなDSL等、PHP以外の言語を触らずに済む)
- DBを裏側からさわって答え合わせ等が可能！
  （本番コードから流用することができる）

サンプル

```php
<?php
/** @noinspection NonAsciiCharacters */
/** @noinspection PhpUnhandledExceptionInspection */
declare(strict_types=1);

namespace MyTest\Scenario;

use PHPUnit\Framework\TestCase;
use Symfony\Component\DomCrawler\Crawler;
use Symfony\Component\HttpClient\HttpClient;

class TopPageTest extends TestCase
{
    public function testトップページを表示(): void
    {
        // https://symfony.com/doc/current/http_client.html
        $c = HttpClient::create();
        $res = $c->request('GET', 'https://cfe.jp/');

        // check response header
        $this->assertEquals(200, $res->getStatusCode());
        $header = $res->getHeaders();
        $this->assertArrayHasKey('content-type', $header);
        $this->assertStringStartsWith('text/html', $header['content-type'][0]);

        $this->assertStringContainsString('html', $res->getContent());

        // DomCrawler
        $d = new Crawler($res->getContent());

        // check title
        $this->assertEquals("cfe.jp - uzulla", $d->filter('title')->text());

        // find with css selector
        $sns_link_list = $d->filter('body > ul:nth-child(4)');

        // find in found result
        $a_that_have_li = $sns_link_list->filter("li a");

        // check in loop
        foreach($a_that_have_li as $a){
            $this->assertStringStartsWith('https://', $a->attributes->getNamedItem('href')->nodeValue);
        }
    }
}
```

```php
<?php
/** @noinspection NonAsciiCharacters */
/** @noinspection PhpUnhandledExceptionInspection */
declare(strict_types=1); # せっかくなので新しいPHPに合わせていく

# 「機能」ではなく、「URL」単位のテストである
namespace MyTest\Scenario;

use PHPUnit\Framework\TestCase;
use Symfony\Component\DomCrawler\Crawler;
use Symfony\Component\HttpClient\HttpClient;

class TopPageTest extends TestCase
{
    public function testトップページを表示(): void
    { # 日本語のテスト名を許す（エンジニアの母国語にもよる）
        // => 次ページへ
    }
}
```

```php
// https://symfony.com/doc/current/http_client.html
$c = HttpClient::create();
# リクエストを飛ばす
$res = $c->request('GET', 'https://cfe.jp/');

# レスポンスヘッダを確認する（重要）
$this->assertEquals(200, $res->getStatusCode());

# 以下はHTMLの例なので、application/jsonなどの場合はまた変える
$header = $res->getHeaders();
$this->assertArrayHasKey('content-type', $header);
$this->assertStringStartsWith('text/html', $header['content-type'][0]);
```

```
# とりあえず文字列で色々判定する
$this->assertStringContainsString('html', $res->getContent());

# 最後まで出てそう？
$this->assertStringContainsStringIgnoringCase('</html>', $res->getContent());

# Display Errorsを拾う（調整が必要な事も多いので各自適当に）
$this->assertDoesNotMatchRegularExpression(
    '/(Notice|Error|Warning):/u',
    $res->getContent()
);
```

```
# HTMLをパースして色々確認する
$d = new Crawler($res->getContent());

# タイトルの確認
$this->assertEquals("cfe.jp - uzulla", $d->filter('title')->text());

# XPathでなく、慣れたCSSセレクタがつかえるよ
$sns_link_list = $d->filter('body > ul:nth-child(4)');

# 選択したDomNodeListからさらに絞り込めるよ
$a_that_have_li = $sns_link_list->filter("li a");

# ちょっとややこしいんだけど、今どきのPHPなら補完できるよ
foreach($a_that_have_li as $a){
    $this->assertStringStartsWith('https://', $a->attributes->getNamedItem('href')->nodeValue);
}
```

```
$ composer test
> vendor/bin/phpunit
PHPUnit 9.5.6 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.202, Memory: 6.00 MB

OK (1 test, 16 assertions)
```

サンプルコード： https://github.com/uzulla/mch

基本的な事はこれでどうにかなります

- HTMLが最後まででていて、エラーがなければ大体大丈夫(?)
- シナリオ名は日本語でも良い（英語でも良い）
  - ただ、日本語だと短くて済む…絵文字も良い
  - 今は多国籍な現場も多いので、調整しましょう
- エラーログも確認しましょうね、自動化しづらいけど
  (これは今回スキップ)

どんどんカスタムしても良い

- TestCaseクラスをラップする、Traitに分離する
  例: clientをWrapして、Status Codeチェックとかを共通化
- 現場にマッチする省略記法を増やすと効率が上がります、
  「複雑でない」ならよし
- URLリストをCSVやGoogle Sheetsから引くなんて便利もアリ
- モダンな静的解析が効くコードにすればリファクタも簡単！
- 「あたらしいPHP」を使うことで気分をアゲていく

(再掲載)面倒で書かない、それが一番やばい

- とにかく簡単に書ける、読める、直せるように
  - 新人でも書けるくらいがちょうどよいしそうすべき
  - ステータスコード見るだけでもいいから書く
- HTML側に、テストのためにセレクタフレンドリなcss classをいれたり、`data-test="hoge"`とかいれて工夫する気になる
  - (「N番目のボタン」問題はモダンツールでもある)

さて、ツールはよいとして、どうテストケース書くか？

- 「完璧なケース」を作ろうとして、
  一つのケースにむやみに時間を掛けるのはつらい
- 平たく言えば試すURLが多いほどよい、たかが知れている
  (…事が多い、運が悪くなければ)

- 「条件網羅」(入力値の多様さ)はさほど重要ではない
- 境界条件でこわれることは(想像よりは)少ないか、
  見つける事自体が困難
- 全体のエンドポイント(URL)の網羅、機能の網羅が重要、
  非互換はある程度固まって存在している事が多い

- アクセスが少ないURLは難しいので、人が補完していく
  - QAの人と取り決めましょう
- アクセスがゼロなら、**直す前に消す検討**

なので、テストのURLパターン収集が重要

- どうやって？ ▶ 本物のログを見る
- アクセスログを切った貼ったしまくる
- アクセスログがないなら、今すぐ保存をはじめる

大量の生ログを切った貼ったとは？

- アクセスログの集計ツールは世の中には結構ある
- ただ、集計・グループ化が「自在に楽に」できないとつらい
- おすすめは、なんとISUCONでもおなじみの **alp**
- かんたんにグループ化ができるので大変便利
- (まあ、自分でツール書いてもいいと思います)

- ちょっとしたビッグデータなのでBigQueryや、
  S3にあげてAthena等を構築してもよいでしょう
- ただ、なんだかんだこねくり回して眺めて回るので
  ある程度は生ログがある方が便利（体験談）
- (SQLで正規表現を駆使できる人は、それでも良いと思います)

LTSV

- alpを使うなら半自動的にそうですが、生ログは取り扱いが面倒なので、解析にはLTSVに変換することを強く勧めます
- LTSVはJSONみたいなもので（雑）パースもしやすい
- 不要な情報は最初から捨てておけます(重要)
  (例: User-agent、IP、等)
- (私は自作ツールでストリーム変換しました)

LTSV例です

```
time:2021-06-23T03:59:30+00:00  method:GET  uri:/   status:200  size:12306  apptime:0.226   type:text/html
time:2021-06-23T03:59:30+00:00  method:GET  uri:/hoge   status:404  size:306    apptime:0.226   type:text/html
time:2021-06-23T03:59:30+00:00  method:POST uri:/api.php    status:200  size:306    apptime:0.226   type:application/json
time:2021-06-23T03:59:30+00:00  method:GET  uri:/redirect?hoge  status:302  size:0  apptime:1.226   type:text/html
```

label:Value{\t}label:Value{\t}label:Value

time アクセス日時

method HTTP メソッド

uri アクセスURL（ホストはない）

status ステータスコード

type Content-Type

素材(生ログ)は鮮度も重要

- 古いログより、最近のログをみましょう
- 「何ヶ月」みるかは現場によります
- (無いなら無いし、捨ててる所も多いし)
- しかし最低でも２ヶ月は必要、できれば１年
- (月またぎで発生するパターンやバッチなどあるので)

その上で泥臭い人力が重要

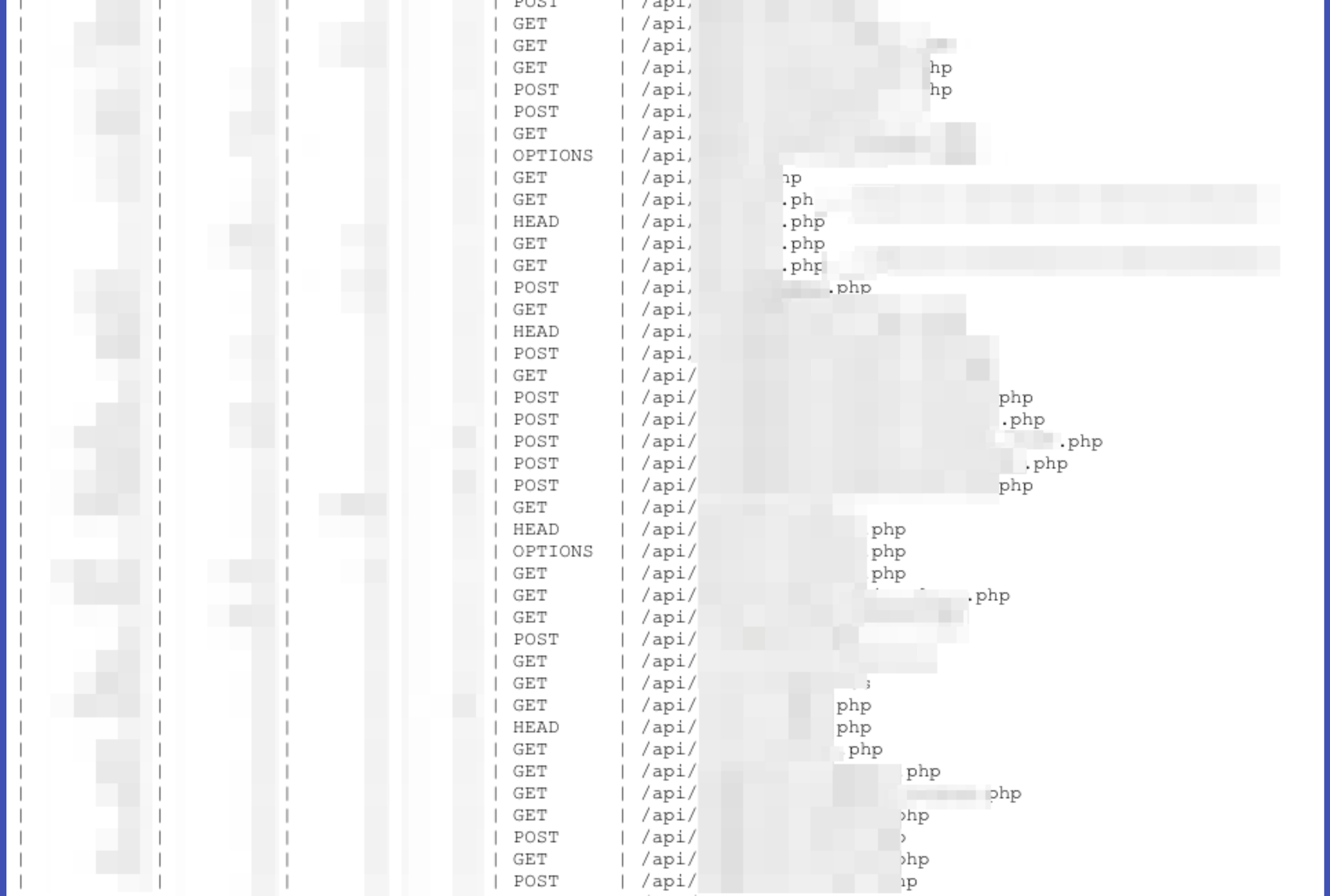
- 機械的にやっても重要なURLを取りこぼします
- ログは機械的攻撃などで汚れてるのでクレンジングも重要です
- 手動テスト(QA)してる人と膝つきで確認する
- 本物をよく見る、よく触る
- 「もともと壊れてた」「そんな機能は無い」こともよくある
- (このあたりで、積極的なチーム体制・応援がないとキツい)

集計しよう！

- access.log.gz を確保する（ここでは確保したことにします）
- alp (https://github.com/tkuchiki/alp) にパターン付きで流す
- これでパターンゲットだぜ！
- (alpのDLやLTSV出力設定は alpのサイト参照)

実行例 (長い)

```
$ gzip -dc access.gz.* | head -10000 |grep -v "/ignore/pattern"| \
 alp ltsv -m "/i/.+,/static/.+,/assets/.+,/item/search/.+,/wordpress/.+,/item/tag/.+,
 /item/detail/.+,/cto/diary/.+,/img/.+,/news/page/.+,/rss/some_one/.+,
 /some_great_page/.+,/campaign_2021/.+,/html/.+,/api/hoge/.+,/ultra-nice-page/.+,
 /you-and-me-relation/.+,/apple-touch-icon.+,/wp-content/.+,/some-creative/.+,
/company-map/.+,/mail/form/.+,/too-many/.+,/already/im-tired/.+,/but-on/going/.+,
/nice_feature/(css|font|js)/.+,/too/old/pages/.+,/product/thumbnail/.+,/&unknown_something=.+,
/wow/what-is-this-page/.+,/detail/get/.+,/we-must-go-php-8/plan/.+,/we-must-go-php-8/do/.+,
/we-must-go-php-8/c-is-what/.+,/we-must-go-php-8/i-dont-know-i/.+,
/already/dead/page/(css|font|js)/.+,/2001-future-travel/(css|font|js),/php4/(css|font|js),
/php/5/.+,/php/(7|8)/.+,/php/6/.+,/controll/panel/(css|js),/admin/.+,/api/get-json.+,
/api/post-json.+,/api/another-json.+,/articles/admin/.+,/articles/page/.+,/articles/tags/.+,
/wp2/.*,/special-april-fool-page/.+,/_nuxt/.+,/wp-includes/.+,/x/.+,/index.php.+,/css/.+,
/js/.+,/img/.+,"
 -o 2xx,3xx,4xx,5xx,method,uri --sort=uri |grep -v -e "/some/" -e "/attack" -e "/%20../../../pattern"
> output.txt
```



- 社内では共有しておりますので、ご覧いただけます
- (まだクレンジング中なので、すぐに参考にはなりませんが)

ISUCONのときの例です

COUNT	1XX	2XX	3XX	4XX	5XX	METHOD	URI	MIN	MAX	SUM
1	0	1	0	0	0	POST	/initialize	8.332	8.332	8.332
11	0	11	0	0	0	POST	/api/estate	0.288	1.708	14.712
11	0	11	0	0	0	POST	/api/chair	0.240	1.200	9.308
457	0	373	0	84	0	POST	/api/estate/nazotte	0.004	2.005	267.969
631	0	631	0	0	0	POST	^/api/chair/buy/[0-9]+$	0.004	0.116	9.036
642	0	642	0	0	0	GET	/api/chair/search/condition	0.000	0.000	0.000
1267	0	1265	0	2	0	GET	^/api/recommended_estate/[0-9]+$	0.004	0.264	28.703
1291	0	1279	0	12	0	GET	^/api/chair/[0-9]+$	0.004	0.160	15.121
1730	0	1730	0	0	0	GET	/api/estate/search/condition	0.000	0.000	0.000
2571	0	2571	0	0	0	POST	^/api/estate/req_doc/[0-9]+$	0.000	0.024	2.912
2823	0	2823	0	0	0	GET	/api/estate/low_priced	0.004	0.276	32.158
2823	0	2823	0	0	0	GET	/api/chair/low_priced	0.004	0.184	45.235
4538	0	3445	0	9	1084	GET	/api/chair/search	0.032	0.432	381.550
4952	0	4939	0	13	0	GET	^/api/estate/[0-9]+$	0.004	0.256	49.671
11443	0	10353	0	6	1084	GET	/api/estate/search	0.001	0.700	308.877

gzipのまま扱う

- ログファイルは巨大、gzip（.gz）状態であつかうと楽
- ログファイルは１つにする必要ない、分割したままでよし

  ```
  gzip -dc * |alp
  ```

  工夫してxargsつかえば並列にもできないこともない
- (再掲)モダンにBigQueryやAthena…と軽くいいたいですが、全員が無心で「できる」には素朴な方が良いので背伸びはダメ

前処理のフィルタ

- headをつかって、だんだんと処理範囲広げると試行錯誤が楽
- grep -vをつかって、「ゴミをある程度捨てる」と楽

alpのコツ

- headで処理行数を制限しつつパターンをグループ化していく
  `-m "/controll/panel/(css|js),/assets/.+"` パターン指定
  `--sort=uri`するとまとまるので、パターン指定時楽になる
- パターンはかなり増えてもalpは速いが、デフォルトでは5000超えると止まるし、積極的に絞ろう
- 今回応答速度は不要なので、出力を設定
  `-o 2xx,3xx,4xx,5xx,method,uri`

- `status:404`のパターンを無視するのは注意が必要、そういうAPIかもしれないから
- 絶対に存在しない(存在しないPhpMyAdminへのアクセスとか…)はいらないので捨てるが、最後に捨てたほうが良い
- 処理には、すごく（数時間〜）時間がかかるので強い計算資源が必要です。並列化もしたい所です、ただ、話が長くなるので省略
  (やはりBigQueryやAthe(略))

で、実際のファイルと突き合わせ

- 突き合わせて、「無い」URIは消し込んでいく
- ファイルはあるがアクセスがないなら「デッドコード」かも？
- 消すことが検討できますね！
  （ただし、本当に消すかは慎重に）
- ただ、ほかのコードからのincludeは元より、rewriteなどでURLが変換されていることもあるので注意(.htaccessやhttpdの設定をチェックしましょう)

話戻) リストができたら、テストケースを作る

- この時点ではGET系だけでよいから書ききる
- なんの魔法もない、テストコードを書く(サンプルは別途)
- 工夫はできるかもしれないが、とにかく無心で作る
- (テストのテスト、現行に流してFailしないことを確認する)

できたらどんどん流してどんどん直す

- テストができたら、環境をたたきまくって検証する
- 修正したらまた叩く
- GET系なので、遠慮なく無限に叩く
- ガンガン直す
- なんかルールがたりてなければ補充する
- また壊れるのでなおす、延々と繰り返す…

POSTは？

- POST系を直すのはGET系の後でよい
- なんなら、POST系は半分手作業でも良い（規模によります）
- 特に管理画面の細かーい場所は網羅がつらいので、
  多少差っ引いても良い（良いのか？）
- とはいえリクエストを適切に送れば、
  POSTとGETは手間の差なので書く、書きまくる。
- (CSRF tokenみたいなのはもうズルして固定しましょう)

その上で！

- 「ログインして所定のJSダイアログをクリックして
  この操作をしてCookieを育ててログアウトして戻るボタン」
- みたいな特殊ケースのためだけに
  ヘッドレスブラウザなどのツールを「やっと」導入する
- おしゃれなE2Eテストツールは文献も多いし、
  ここまでの経験があれば、どうにかなるはずです

こだわりで手を止めてはダメ

(主観です)

道具を選ぶより、完了させる。

「終わらない」という悪夢がくるので

叩かれる道具

叩く道具の次は、叩かれる道具です

(実際は叩く道具とある程度同時に作ると良いでしょう)

サンドボックスを作る

- PHPだけでなく、できるだけミドルウェアも想定に揃える
- 依存するライブラリ（curlとかlibjepgとか）も頑張る
- (同一ArchのLinuxなら、バイナリコピーする等もありうる)
- データは必要な範囲で、「あんまり大きくないように」する
  (無論本番同様が好ましいが、テスト速度に影響が出やすいし、切り分けもしんどい)

(環境を)何でつくるか？

- まあ、Docker(-compose)が便利なのだが…
  - あまりにも古い環境だと、普通のVMが便利な場合もある
    (古いディストリビューションを拾ってくるなど)
  - `php.ini`や`httpd.conf` 等も修正する事が多いので、
    消えると困るので、それらもマウントしgit管理がNice
- このあたりは現場によりますし、やりながら考える

- (mysql/)data 等もVolumeでマウントしたほうが早い
  あるいは、内部で tar.gz を解凍し、毎回上書きする。
- データリセットのために再度SQLを流するより、
  dataをディレクトリごと上書きして再起動が早い
- これはその他のデータも同様、初期化が高速にできる方が良い
- (そのためには、データを外から見えるようにしたほうが便利、
  ただ、Linux以外は遅いんだよな…)
- (このデータはgitにいれなくても良い、入れてもいいけど)

まずは現在の確認のために『現行』を作る（真剣）

- エラーを全有効、かつ表示にもだしてしまう、PHPは便利
- すると元々ある「無視しているエラー」が出る場合がある
- というか、十中八九無視していたWarningやNoticeやStrictやDeprecatedがでる
- **まず、これを直す**
- @で安易にエラーを握りつぶしてる所も、当然！全部！直す！

「はい、ここでは治ったことにしますね」

やっと「新環境」サンドボックスをつくる

- ここらへんは現場現場があるので、どうこうはない
- ポート番号かIPを変えて、前後を同時に上げられるとなお良い
- 最初から「最終地点」のPHPバージョンにするのはむずかしい
- 一気にあげて、巨大に壊れると、直すのに消耗してつらい
- 順番に上げる。元々一気に行けるコードならトントンとすすめる

(再掲)ビッグジャンプは難しい

- v4やv5.1.6から直接v8.0するなんて前代未聞(作り直しでは？)
- 全く動かないままNヶ月あるいはN年経過する等は辛い
- 例: v5.1.6 ▶ v5.3.3 ▶ v5.4 ▶ v5.6 ▶ v7.1 ▶ v7.4 ▶ v8...
- ミドルウェアも上げていく(例: MySQL v5.1 ▶ v5.6 ▶ v8とか)
- (DeprecatedコードからDeprecatedコードにするのは辛いので、静的解析だけは7.4設定で都度実施し様子を見るのはGood)

改修作業

さあ、マイグレーション作業だ！

(多少順序は前後しましたが)

準備の話ばっかりだったし、ついに作業だ！

最初に残念なお知らせ

- 「すごいツールを教えて下さい！」「本音をいうと、無い」
- 「勉強します！本をください！」「正直にいうと、無い」
- 「やり方を教えて下さい！」「ぶっちゃけていうと、無い」
- あるなら、誰も困ってないよね、ハハハ！！！！
- (世にあるのは、大抵アプリ設計のリファクタリングの本です)

それでも静的解析ツールはあります

- 例(これ以外にも色々ある):
  - PhpStorm
  - https://github.com/wapmorgan/PhpDeprecationDetector
  - https://github.com/phan/phan
  - https://github.com/phpstan/phpstan
- 「どれが正解ですか？」
  「現場(対象コード)による、正解はないので全部試す」

- 「なんか、PHP>=7.2ってありますが？」
  「そう、ツールも古いPHP切ってるんだよね、ハハハ」
- 古いバージョンを持ってくることはできる
  （GitHub等に感謝）
- なお、解析ツール自体は最新PHPで動かしても良い
  （ことが多いし、最新PHPのほうが断然高速に処理できる）
- **ただし、私はどれも信じておらず、参考にとどめている**

静的解析には限界がある

現実の問題は検知が困難

```php
<?php

list($a[], $a[], $a[]) = [1, 2, 3];
var_dump($a);
```

eval(); all supported versions

Output Performance VLD References Branches

Output for 7.0.0 - 7.0.33, 7.1.0 - 7.1.33, 7.2.0 - 7.2.34, 7.3.0 - 7.3.29, 7.4.0 - 7.4.21, 8.0.0 - 8.0.8

```
array(3) {
  [0]=>
  int(1)
  [1]=>
  int(2)
```

Output for 5.4.0 - 5.4.45, 5.5.0 - 5.5.38, 5.6.0 - 5.6.40

```
array(3) {
  [0]=>
  int(3)
  [1]=>
  int(2)
```

あくまで公式から引っ張ってきた例です

- 配列順序の「期待」を裏切る挙動、たとえば順序が変わる
- 前述は「違法ではない」のでかからない(と思う、多分)
- PHPは丁寧なので(?)マイグレーション指導にはある
- https://www.php.net/manual/ja/appendices.php
- (読んで「見つけられる様になる」わけではないが
  見つけた後に「解る」ためにはとても手助けにはなる)

余談：挙動の確認には xdebug（デバッガ）

- これは静的解析より**必須**、PhpStormと連携させましょう
- デバッガが嫌いな人は好きになったほうが良い
- どうしてもコードが意図する挙動が理解できなかったり
  挙動の差がでるなら、ステップ実行しかない
- (print debugは「極稀に」挙動が変わるのでお勧めしません)
- デバッガのために三項演算子を展開する、等の工夫はあります
  (壊さないように、慎重に…)

話を戻す: 静的解析は銀の弾丸たり得ない

- Deprecatedを探すツールはあるが、完璧ではない
- 基本的に「良いコード」をサジェストするものが多く、
  レガシーコードにかけると、無限にサジェストされて溺れる
- 昔の動的にinclude多用等は推論出来ずにエラーになる
  「見つかった！」と駆け寄っても、それはエラーではない
  (静的解析のエラーといえる)
- (駆け寄ってリファクタしてしまう罠にかかる人が多い)

矛盾するようですがツールは使える

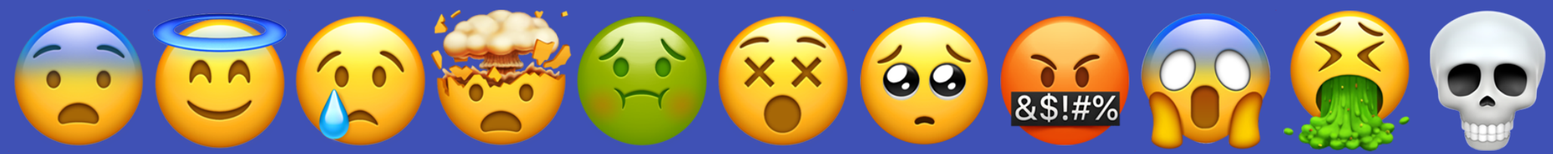
- 「ここおかしくない？」と言われれば当然捗る
- しかし、修正すべきかは**静的解析ツールが決める所ではない**
- **静的解析ツールの警告を潰す事が目的ではない**
- ここを間違えたり、信頼しすぎて罠にハマる人が多い
- 安易なAuto fixなんてもってのほか
- (どちらかといえば「どこまで無視して良いか？」の方が重要)

「ツールがないなんて！どうすれば…！！」

- 既存の挙動が(PHP等の)バグ前提のコードも稀にある
  - (特に文字列検証や配列操作等でエラーを直すと壊れる例)
- 皆さん、もうおわかりでしょうが…
- そう、E2Eテストするしかありませんね！
- 頑張りましょう
- （これは冗談でなく、単なる事実です）

🤠 < 頑張ってテストしようね！

(「PHPのスペシャリストと呼ばれて来た人間がこう発言した時」に私がよく見る現場の皆さんの顔)



（経験した上で、魔法はないんだと俺は言っています、あるいは魔法使いの方を雇ってくれ）

余談：PhpStormでの静的解析のかけ方

- PhpStormのCode inspectionは使えるっちゃ使える
  - 1. PHPの希望するバージョンを設定する
  - 2. ルールをカスタムする
    (HTMLやCSSのエラーをオフにする)
    (色々あるPHPの項目も頑張って考えてオンオフする…)
    (勿論、ここにも「正解」はない)
  - 3. すごい時間がかかって、大量にでてくる

かさねていいますが

- 『Error数千、Warn数万、誤検知あり』であっても静的解析は便利です、必携です
- ただ、手を引かれて言った先はエラーじゃなかったりバージョンアップに関係ないリファクタリングだったりします
- ちなみに「モダンなコード」になると静的解析がバッッキバキに効果を発揮し、無限のメリットが成就でき、エラーもAuto fixもほとんど間違いなくなります。
  しかしレガシーコードでは無理なんですよ、びっくりします

ハッハッハ、これが本当だから困る

（レガシーシステム改善マンへのエール）

「大丈夫、デプロイしちゃえば
全員を巻き込めるぞ」

（つまりさっさとやっつけてしまえ）

実際どうやってコードを直していくか（やっと！）

- **テストを回す ▶ 直す ▶ テストを回す** これの繰り返し（再掲載）
- 手をとめない、どんどんアドホックに書き直す
- 気になったら、とりあえず 「`// TODO ここがヤバい`」とか書いて、すぐに忘れましょう。
- デグレを恐れてはならない、そのためにテストがある
- ただし「テストが壊れないな？」とおもったらテストも疑う事

コミット粒度について

- テストを回す度にコミットするくらいでよい
- すると「確認」とかいうコミットがたくさんできる
- しかし悩んで遠慮して、もどせないのが一番つらい
- どうせデグレも発生するし作業ブランチのログは諦めるべき
- **こんな所でストレスを増やすほうがつらい**
- どうしてもcommit logが気になるなら `git squash`

プルリクについて

- 特に影響範囲が広い「共通コード」を直す場合
  こっちを直すと、あっちが壊れるということがあります
- 「共通コード」を避けてとりかかった端っこのコードが
  共通コードに根ざしたバグに関連することも多々あります
- 結局直す必要があり「プルリク粒度」が滅茶苦茶になります
- するとレビュー、コミュニケーションコストが増大します
- （これは現場によります、そうならないコードもある）

- なので「できるだけ」共通コードを先に直したほうが良い
  そのためだけに、簡単なユニットテストを書く場合もあります
- しかしそれは理想論なので、うまく行かない事の方が多い
  トップページの小修正が、DB周りのコードまでいくことがある
- やりなおすべきか？いや、私は割り切ったほうが良いと思う
- **The done is better than a perfect.**
- (これは手抜きしろ、ということではなく、
  「美しさ」とかはリファクタフェーズにしろということです)

余談：「なんか荒々しい意見多くない？」

- 全てが順調スマートにできるなら、もう8にまで上げているはず
- バージョンアップってのは、性質が炎上案件に近い
- あくまでこれは「私の意見」なので、PR TIMESさんにおいては色々まあ調整したりなんだりするとおもいます

話を戻して: ログのエラーを潰していく

- (再掲ですが)
- ログをE_ALLにして、エラーを全部直していく
  Fatal, Error, Warn, Notice, Deprecated, Strict.
- **PHPのエラーは全て違法**、VerUPにもほぼ有害、全部対応する
- 割と皆さんエラー無視しがちだが、無視してはならない、
  Deprecatedなんて見つけたらラッキーチャンスと言える
- (なお、アプリの『デバッグログ』は邪魔なのでオフしましょ

勿論ライブラリも壊れるので対応する

- バージョンを上げる、代替を探す、無ければ作る
- あるいは「forkして動くようにする」、OSS最高！
  - (有名FW修正より、オレオレ/自社FWの方がまだ小型な分楽)
- モデル、データ層周りだと、かなり気合がいる
- これは現場によります

差し替えに必要なリファクタリングはゆるされるか？

- 「リファクタリングするな」に違反してよいか？
- （仮に、どれもPHP8まで「動く」として…）
  - A「結構リファクタリングして、新しい物を使う」
  - B「修正範囲が少ないが、少々古めかしい物を使う」
  - C「ライブラリをWrapして、今のレガシーコードに合わせる」

壊れていないがすごく古いライブラリは？

- 古くて、静的解析がエラーをだしまくっていて、セキュリティ的にもよくわからなくて、配布元が潰れていて、でも動く
- もう正解はわかりますね？
- いまは触らない
- それはバージョンアップと関係がない
- (別で差し替えて、rebaseするとかはあるでしょうが)

開発ブランチの新規機能の追従について

- これは現場によるのだが...
- 振り出し感がつらいよね、わかる
- レガシーコード部分のリファクタだけはやめてもらう
- 「随時マージができるといいですね」（願望）
- テスト書いて欲しいが、協力度、理解度、修羅場度による
- まあ、諦めが重要。終わりよければ全て良し

大体できたら

- 人力でもがんばってチェックする
- 全員でやる、全員に関わらせる
- なにせ、ダメだったら全員が辛いぞ？
- テスト抜けがみつかったら、**必ずそれをテストに追加する**
- 「もうデプロイしようよ…」レベルまでうんざりできたら…

「ジェロニモ！！」

(デプロイする時の掛け声)

Jump in!!!!

- 度胸一発、本番のバージョンを上げてデプロイしてみる
- (ちゃんとするなら、カナリアデプロイする)
- もし「多少の不具合」で動くみたいなら、
  (判断が難しいが…)勢いでそのままいったほうが良い(真顔)
- お知らせに掲示しつつ必死に直すターンです

本番は最高のテスト環境なので

- 無限のテストが行われ、無限のバグレポがきて嬉しい(?)
- 張り付きでPHPのエラーログを見る
- 必死に直す
- 本番だから、全員で対応しよう！（笑顔）
- (ここで散々な目にあう事がありますが、頑張ろう)

「精神論じゃねーか！」

- **日常に改善フローが出来るまでは** こうなるでしょう
- ジャンプのたびにHPが削れるので、沢山するとつらい
- しかしジャンプしないと先に進めない
- ジャンプしちゃえば、案外どうにかなる(楽観論)
- 怖がってデプロイしないのが一番やばいって
  それみんな言ってるから

続く…

- 「やったっ！第一部完！！」
- しかし次の強敵が即座に登場する…
- 「俺はバージョンアップの道を上り始めたばかりだぜ…！」
- 完…

完ではない！

そう…バージョンアップはレガシーシステム改善の「準備」！

動いた後に、あらたな戦いが始まる！

- "Include Oriented PHP"から脱却し、OOP、Autoloader化
- コーディング規約の適用
- テンプレートエンジンやルーターの刷新
- 「本物のテスト」や「本物の静的解析」、CI、CD化
- 強敵にみえます？いやいや「着実にやる」と大差ないですよ
- やっていきましょう

「ふむ、100000ファイルか、有限時間内に終わるな」

- **Keep Calm and Carry On.**
  "平静を保ち、普段の生活を続けよ"
- 実際、やれば終わる（経験談）
  投げ出せば終わらない（経験談）
- 日々のルーチンワークへの落とし込み(思想,道具,体制)が重要
- それが自信と、作業ハードルを下げることに繋がります
- まあ、それは別の話なので、またいつか

改善は終わらない

未完

ありがとうございました

その他教訓

聖域のない雑談は便利（？）

- 「ここで困ってる」「ここ本当につらい」「今ならこうすべき」「代替ライブラリがない」「うんざりしてきた」
- 辛い仕事に不平不満は当然ある。単なる不満はのみこむ・溜め込むくらいなら、少々ガス抜きをして気晴らし（？）をしてでも進めたほうが良い、困りを率直にシェアできる空気は重要
- ただ、既存コードには間違いなく価値があり「当時そうだった」だけで「悪」ではない。今やっている作業の効率が重要で、活発な情報共有や連帯感拡大等の目的を忘れぬ事

コードフォーマット、先にかけるか後にかけるか？

- おすすめは先、なぜなら壊れてもこれから気付くし
  モダンIDEで自動整形をオフにして書くのは結構苦痛
- 装飾性は気にしたら負け、時代はオートフォーマット
  - 2スペースやハードタブインデントは4スペースに
    古式ゆかしい装飾性の高いコメントはシンプルに
    /**\n*などの行ズレは気になったら直す
    空白による =(代入) 揃え等は無視する

その他、Typoや古い記法、表記ゆれについて

- 「動く間は直さない」(一部途中で直す事にはなる)
  - TRUE▶true、Var_Dump("")、array()▶[]、"$a"▶"{$a}"
  - <?、末尾の?>、&=、var、return (true)?true:false;
  - 行の幅制限（これは無限にしておくことを勧める）
- 静的解析が完全に効くレベルになったら直す
- (そのころにはAuto fixを使っても壊れなくなるでしょう)

レビューもテスト寄りの視点で

- バージョンアップは作業なので「良い書き方」で粘らない
  たとえば「このifは整理して早期リターン」とかは未来にやる
- レビューは「さっさと終わらせる」の視点で望むべき
  - 勿論、潜在バグやエラー処理不足等は指摘する
  - その上で、誘導学習より「終わる」が大事
    考えさせるより実例書いたほうが良い

最初に掃除する

- 相手にするコードを減らす、これは本当に馬鹿にならない
- ログ調査・聞き込み調査をしたら、総コード量7割を占めるバカでかいエンドポイントをまるごと潰して良いケースもあった
- **とってあるだけ** を相手にするバカバカしさは本当にひどい
- （ただ「これはゴミでは？」といって消して全部がとまったサービスも知っています）
- 「これ、まだいります？消しません？」も重要です

ファイルを消してよいか安心するテク

- 「このファイルは消しても良い筈」とおもったら、そのコードにアクセスがあったらログを出すようにすると良い。マーク代わりにもなる。
- 例:`error_log("Unexcepted use :".__FILE__, 3, "/tmp/log");`（ログ基盤にながすのはOKだが、エラーログと簡単に分離できること。Slackに流すのもGood）
- 納得したら、定型文で探して消す。

たまにある、シリアライザ非互換

- 何らかのデータ・オブジェクトを文字列などに（たとえばJSON）変換して保存することがよくあるが、これのデシリアライズの互換性が消える事がある
- (クラスを修正したらdeserializeできなくなる事はよくある）
- serialize、encodeまわりは重点的にチェックすること
- セッション等は元より、ジョブキュー、DB等に潜んでいる事も

「壊れた文字列」などの非互換

- 稀に、いままで「壊れたデータでも素通ししていた」ものが突然許されなくなる事がある。ライブラリのバグが「なおっちゃう」等が原因
- デバッガでないと原因がよくわからない事が多い
- さらに稀に「壊れているとデータを捨てる」挙動もある
- これまた非常に辛い、がんばろう

ビッグにいくか、刻むか

- どれくらいの頻度でメインブランチとマージやリベースをするか
- **バージョンアップに関しては**、
  細かくしすぎると永遠に終わらないことがある
- ある程度はビッグ＆フリーズをおすすめする
  - 機能追加をするなということでなく、メリハリをつける
- リファクタリングフェーズでは刻んだほうが良い

レガシーシステム改善はメンタルに注意

- 大体評価されないし、進化のブロッカーになる事も多い
- デプロイで壊せば落ち込むし、悪ければ公式謝罪になる
- 「気にするな！」とかいわれても **気にしない人間はまずいない**
- **それは普通**なので、辛くなったら宣言する、どんどん息抜きする
- 不満は負債同様、たまる前に主張して **全員** で処理する

本当の資産ことシナリオテスト

- レガシーシステム改善で実感する
  - あらゆる機能は再実装できる
  - あらゆる機能は置換できる
- しかし、シナリオテストがないと「わからない」
- バージョンや、フレームワークの新旧はマジで関係ない
- まずは正常系シナリオテストがバージョンアップには重要