

使用ライブラリのバージョン管理

現代のソフトウェア開発では多くのライブラリに依存して開発を行います。大抵、システムの開発開始時には、その時点での新しいライブラリを使うと思いますが、長期的な運用を考えると、使用するライブラリやパッケージの更新というのは避けて通れない話です。本章ではそのあたりについて紹介します。

前半では技術的な説明ですが、後半はソフトウェア開発に詳しくない人向けに説明する時に参考用の啓蒙的な内容になっています。コストをかけてバージョン更新をしなければならない理由がわかっている方は前半だけ読んでおけば良いです。

バージョンとは

現在提供されているシステムの多くは3つの数字を並べたバージョンを使っています。

- x.y.z

xをメジャーバージョン、yをマイナーバージョン、zをパッチバージョンと呼んだりします。例えば、12.0.4とか、3.7.3とかそういうやつです。

Windowsは商品名としては95とか2000とか10とかつけたりもしますが、内部的には2つの数字の列になっています。18362.175とかそういうやつです。

数字付けのルールは各システムが勝手につけることが多いので、全部のシステムで統一的なルールというのは、大きい数字ほど新しい、ぐらいのものです。昔はxが偶数が安定板、yが奇数が開発版みたいなのがよく使われたりもしていましたが、マーケティングの都合でいきなりxが大幅にジャンプしたりとかあります。xが上がると後方互換性がないバージョンアップだが、yの更新は後方互換性があるとかもよく見かけます（セマンティックバージョニング）が、気分でxをあげるシステムもあります（Linuxとか）。

フロントエンド開発でよく出てくるルールがセマンティックバージョニングです。この3桁でバージョン間の大小を一意に定めます。これにより、「古い」「新しい」が判断できるようになります。

細かいルールはもっといろいろあり、例えば、1.0.0よりも1.0.0-alphaの方が古い、というルールもあります。詳しくは次のページを参照してください。

- [セマンティックバージョニング 2.0.0](#)

バージョンのサポートの考え方

サポートの考え方は大きく3種類ぐらいですね。

- 最新メジャーバージョンのみサポート
- 最新のいくつかのメジャーバージョンのみサポート
- 最新のメジャーバージョンと、特定の不連続なメジャーバージョン（LTS）のみサポート

たいてい、メジャーバージョンごとにサポート期間を設定することがほとんどです。開発リソースの多いプロジェクトでは、複数メジャーバージョンを同時サポートします。小さいプロジェクトや個人プロジェクトでは最新バージョンのみサポートというケースがほとんどです。また、変化の早いブラウザも最新バージョンのみです。

最新のいくつかのメジャーバージョンというのは、例えばOracle社製のデータベースは最新2バージョンのみサポートとかそういうやつです。ただ、ウェブのフロントエンド開発ではあまりみないかもしれません。

よく見るのがLTS（ロングタームサポート）という長期サポートバージョンを定めているライブラリとかツールです。

Node.jsは、現在は半年ごとにメジャーバージョンアップします。最新のメジャーバージョンのものはcurrent扱いです。奇数バージョンはcurrentでなくなるとすぐにサポートが終わりますが、1年に一回出る偶数バージョンは、currentでなくなると（次のバージョンが出ると）LTSになり、2年半サポートされます。現在の最新は14で14はメジャーバージョンアップ対象ですが、現在も活発に機能追加が行われていますので、LTSにはなっていません。15が出ると14がLTSになります。

例えば、12系は、2019年10月21日に12.13と13.0がリリースされ、12.13がLTSになりました。執筆時点では12.18がリリースされています。しかし、12.12以前はLTSではなく、新機能もどんどん追加されます。このあたりは他のソフトウェアと異なるルールになっていますが、アルファ版やベータ版では利用する人がおらず、数値が大きく上がったリリースのタイミングで使い始める人が多く、そこで初めて不具合が出て報告されたりするので、多くの人に使ってもらってバグ修正をするというOSSのエコシステムを考慮すると合理的な考え方であると思います。

ライブラリではAngularがすでにLTSを含む運用をしており、現在最新の8は、次の9が出るとLTSになって、その後1年サポートされます。Vue.jsも、[3.xが出たら2.xの最終版がLTSとして18ヶ月サポートされる](#)と宣言されています。

バージョン選びの作戦

Node.jsやアプリケーションで使うパッケージのバージョン選びの戦略は主に3つあります。バージョンアップ作業には時間がかかります。バージョン更新そのものに加えて、確認の工数もかかります。その分、新機能開発の工数は削減されます。時間がかかるということはそれに対して費用も発生します。どこの費用を使ってやるか、どこに請求するか、稟議をどう投げるかの考慮が必要です。なので、それをどこで消化するかを決めるのがバージョン選びの大切なところです。

「そんなの決めなくてもなんとかなるよ」というのは、チーム内の誰かの善意（やる気）に甘えているだけなので要注意です。

いくつか考えられる作戦を列挙してみます。どれか一つを選ぶというよりかは、状況に応じて複数のパターンを利用すると良いでしょう。

1. 最新バージョンを積極的に選ぶ

常に最新バージョンを取り込んでいくスタイルです。バージョンアップタスクの分散化です。日々最新のものを取り込むスタイルです。例外としてはiOSのモバイル開発で、最新iOSのGMが出てから、ユーザーの手に最新のOSが渡り始める1-2週間で動くものを作って提出しなければなりません。あと、Reactとか、安定版などなく、最新版のみが更新されていくライブラリがコアとなると必然的にこうなります。

新しいバージョンを試してその知見を公開するだけでもありがたいと言われるというメリットもあります。バグ報告とかでそのソフトウェアに貢献もできるかもしれません。類似パターンとしては、ベータ版も試すパターン、最新のmasterのバージョンも試すパターンもあります。

ただ、現在は複数のライブラリを組み合わせる行うため、新しすぎるバージョンでは他の連動して動くライブラリの準備ができていないケースもあり、予想外に時間が取られることがあります。特に、Babelなどの影響の大きなライブラリのバージョンには要注意です。以前、Next.jsが、Babel 7 beta42だかの中途半端なバージョンに依存して、いろいろ食い合わせが悪くて苦労したことがあります。

2. LTSを中心に組み立てる

メインの部分（Node.jsとかAngularとか）をLTSで固め、その周りをそれに準拠する形で固めていきます。メリットとしては、スケジュールが見えているので、あらかじめバージョンアップのタイミングを計画に折り込みやすい（年間予算の計画が立てやすい、稟議にかけやすい）というものがあります。LTSのリリースの前には安定化のための期間が置かれており、比較的問題が起きにくいでしょう。

ただ、LTSが提供されているものならこれでいけますが、現状、LTSを提供しているコアのライブラリはあんまりないので、現状はAngularを使っている場合のみしか適用できません。Vueはそのうち始まりますね。Reactの場合は、LTSはないが、きちんと検証された組み合わせであることを期待して、Next.jsのメジャーバージョンアップに淡々とついていく、という方法があります。

LTSを使う場合も、LTSの範囲内で最新のものを積極的に使うか、固定化するかみたいな細かい作戦の差はあります。

AngularでLTS固定運用をしてみた感想でいうと、TypeScriptのバージョンが古く、lit-elementが利用できない、LTSの範囲内だと次に説明するセキュリティ脆弱性の更新が得られないことがある、といったデメリットがあるのは感じました。

3. セキュリティの脆弱性が検知されたものを更新する

ライブラリのセキュリティの診断はここ数年でいくつかの手法が利用可能になっています。数年ぐらい前は `snyk` にユーザー登録をして `snyk` コマンドを実行して検知していました。現在は `npm` コマンドを使って `npm install` するだけでも脆弱性のあるパッケージが検知できます。また、GitHub にソースコードをアップロードすると、GitHubが検知してくれます。

脆弱性のあるパッケージは自分が直接インストールしたもので発生するだけではなく、そのパッケージが利用している別のパッケージのさらに依存しているパッケージが・・・みたいな依存の深いところで起きがちです。特にウェブフロントエンド開発をしていると、依存パッケージ数が簡単に4桁とかいってしまうので、すべてを目視で確認するのは難しいです。自動検知を活用しましょう。

なお、検知されたすべてを修正しないといけないかというと、そんなことはありません。例えば `node-sass` は `request` という外部ネットアクセスのライブラリに依存しており、これが更新されなくて脆弱性が検知されたことがありました。 `node-sass` はCSSを書きやすくしてくれるユーティリティで、実行時には動作しません。ビルド前のCSSの中で外部リソースに依存していないのであればこのライブラリは使われないはずで、「これは検知されたが影響はありません」というように、説明がつけばOKです。

バージョンアップの方法

`npm` コマンドにはバージョンアップを支援するサブコマンドがいくつかあります。

まずは、現在のバージョンを知るための、 `npm outdated` コマンドです。これを実行すると、現在インストールされているバージョン、現在のバージョン指定でインストールされる最新バージョン、リリースされている最新バージョンが表示されます。

<u>Package</u>	<u>Current</u>	<u>Wanted</u>	<u>Latest</u>	<u>Location</u>
ava	0.25.0	0.25.0	2.4.0	shadow-fetch
browserify	16.2.0	16.5.0	16.5.0	shadow-fetch
codecov	3.0.0	3.6.1	3.6.1	shadow-fetch
eslint	4.19.1	4.19.1	6.6.0	shadow-fetch
express	4.16.3	4.17.1	4.17.1	shadow-fetch
flow-bin	0.70.0	0.70.0	0.112.0	shadow-fetch
node-fetch	2.1.2	2.6.0	2.6.0	shadow-fetch
nyc	11.7.0	11.9.0	14.1.1	shadow-fetch
streamtest	1.2.3	1.2.4	1.2.4	shadow-fetch
unfetch	3.0.0	3.1.2	4.1.0	shadow-fetch

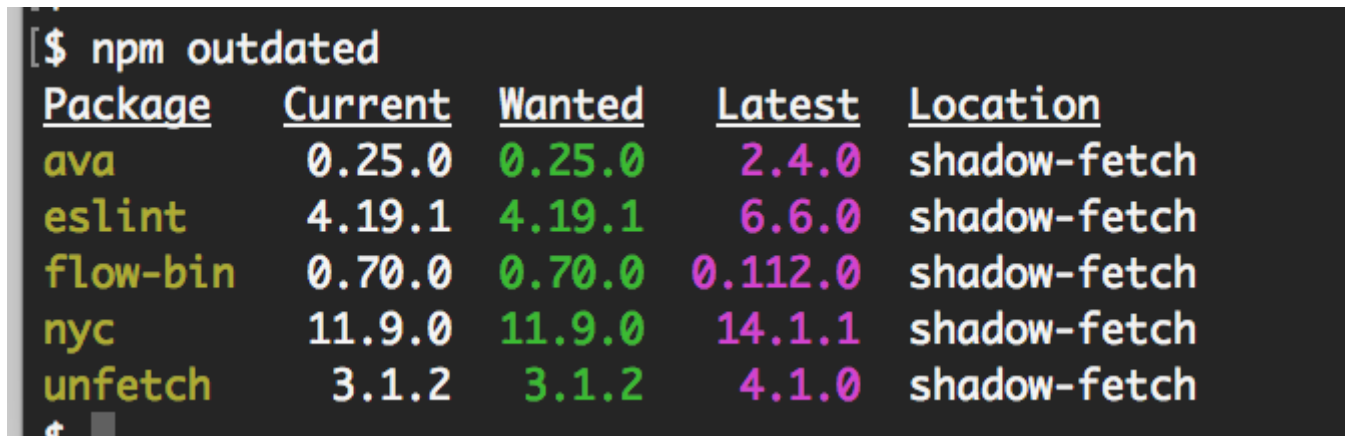
`npm outdated` の実行結果

npmでインストールするときは、最新のバージョンがインストールされますが、package.json上はその時のバージョンが固定されているわけではありません。 `"browserify": "^16.2.0"` のように、`^` や `~` が先頭に付与されています。`^` であれば `16.3.0` があればそれも利用する、`~` であれば `16.2.1` であれば利用するなど、マイナーバージョンやパッチバージョンの変更は吸収する意思がありますよ、という表示になっています。「現在のバージョン指定でインストールされる最新バージョン」というのは、変更可能な範囲での最新という意味です。

更新する場合は、 `npm update` コマンドを使用します。

```
# まとめて更新
$ npm update

# 一部だけ更新
$ npm update express
```



<u>Package</u>	<u>Current</u>	<u>Wanted</u>	<u>Latest</u>	<u>Location</u>
ava	0.25.0	0.25.0	2.4.0	shadow-fetch
eslint	4.19.1	4.19.1	6.6.0	shadow-fetch
flow-bin	0.70.0	0.70.0	0.112.0	shadow-fetch
nyc	11.9.0	11.9.0	14.1.1	shadow-fetch
unfetch	3.1.2	3.1.2	4.1.0	shadow-fetch

`npm update`を実行し、`Current`が`Wanted`になった

この場合、メジャーバージョンアップしたライブラリは更新されません。その場合は手動でインストールします。

```
$ npm install ava@2.4.0
```

セキュリティ目的の自動バージョンアップ

GitHub上、あるいは開発環境での `snyk` コマンドや、`npm install`時に脆弱性診断が行われます。また、インストールを行わなくても、 `npm audit` コマンドを実行しても脆弱性が報告されたパッケージがあると検知されます。何かしらを検知したら次のコマンドで可能なものを修正します。

```
$ npm audit fix
```


これだけでちょっとした修正は完了できるはずですが。メンテナンスがあまりされていないパッケージの場合は、脆弱性ありで修正がないまま放置されていたりします。あるいは、脆弱性ありのバージョンに依存したまま、ということもあります。こうなると `npm audit fix` をしても脆弱性があるモジュールでも修正できなくなってきました。それをトリガーにして一部のパッケージのメジャーバージョンアップが必要となります。もちろん、他の戦略をとっていても重大なセキュリティの場合には応急処置せざるを得ない可能性があります。まあ、セキュリティの緊急度が高いほど、いろんなメジャーバージョンに対してパッチが発行されることもあり、逆に簡単かもしれません。

あまりにも脆弱性が放置されているライブラリがある場合は、バージョンアップではなくて、類似の別のライブラリに置き換える、というのも選択に入ります。

バージョンアップ時のトラブルを減らす

バージョンアップでのトラブルを完全にゼロにはできません。ただ、日頃からの心がけで少し楽にすることはできます。

CIをしておく

普段からCIをしておくことで、いざバージョンアップ時の確認の補助に使えますし、最近では、利用されているモジュールの中に脆弱性のある古いバージョンが紛れていないかの自動検知が行えるようになってきています。

JavaScriptと比べたTypeScriptの場合、一番有利なのがここですね。ライブラリのAPIが変わってビルドができない、というのが検知できるのがメリットです。もちろん、ロジックなどが正しく動くかどうかというテストもあるに越したことはありません。

こまめにバージョンアップ

セキュリティのバージョンアップ、パッチバージョンアップなど、小さい修正はこまめにやっておけば、いざというときにあげるバージョンの差が小さくなります。例えば、1.6.5から1.6.6で、0.0.1だけあげたら問題が起きた、と分かれば、エラーの原因の追求、問題の報告が極めて簡単になります。

人気のある安定しているライブラリを利用する

身もふたもないのですが、APIのbreaking changeが頻繁に行われないライブラリを選べば楽になります。後方互換性やサポートポリシーについて言及があるライブラリが良いです。あと、人気があるライブラリであれば、バージョンアップで困ったときに情報が入手しやすくなります。

ライブラリやフレームワークを浅く使う

ライブラリやフレームワークを使う場合、メジャーな一般的な使い方からなるべく外さないようにします。ライブラリをラップして完全なオレオレフレームワークを作るとかすると、バージョンアップ時の作業が多くなります。また、メジャーな使い方に近づけておけば、ネットで情報を調べるときにも問題が発見しやすくなりますし、チームメンバーが途中から入ったとしても、実は最初から使い方を知っている、ということも期待できるかもしれません。

（参考）式年遷宮

近年のウェブフロントエンドは、たくさんの小さなツールやライブラリを組み合わせる使うことが多いです。ライブラリの数が多ければ多いほど組み合わせの数は爆発していきます。世界であまり多くの人が試していないバージョンの組み合わせでやらざるを得なくて・・・ということも起きるかもしれません。

複雑化するにつれて、プロジェクトの新規作成を手助けするツールが提供されることが増えてきました。特にVue.jsのCLIはきちんと作り込まれています。いっそのこと、バージョンアップ作業をするのではなく、CLIツールを最新化して、それでプロジェクトを新規に作り、それに既存のコードを持ってくるという方法もありかもしれません。

なぜバージョンを管理する必要があるのか

なぜライブラリのバージョンの管理が必要なのでしょう？バージョン固定じゃダメなのでしょう？

プログラムを開発するときは、他のツールやライブラリを当たり前に使います。これは今に始まったことではなく、はるか昔からそうですね。OS組み込みの機能だけで開発するとしても、OSベンダーの提供する開発ツール、OSの機能を使うライブラリ（API）は最低限使います。

例えば、Javaで開発する場合、Javaの言語、言語組み込みのライブラリは使いますし、Gradleみたいな別なビルドツールやらも使います。SpringBootみたいなライブラリも使います。それぞれ、どのバージョンを使うかというのをスタート時に決めますし、メンテ期間等で見直しをする必要がでてきます。

なぜ固定ではダメかということ、主に2つの理由があります。

機能的な問題

それぞれのツールやライブラリは、それぞれの開発元が考えるライフサイクルで更新されていきます。そのタイミングで、機能が追加されることもあれば、過去のバージョンで提供されていた機能が削られたり、挙動が変わったり、というのがありえます。その過去のバージョンがもう手に入らない、ということもありえます。

ウェブの場合だと、アプリケーション側でコントロールできないものにブラウザバージョンがあります。ある程度は使用バージョンを固定するなども業務システムではありますが、古いブラウザでしか動かないとかはダサいですよね。

例えばFlashを使っていると、もう動かすことはできません。実装していた機能を取り除いて、その互換実装に置き換える、という作業が発生します。

もっと小さい例でいえば非推奨になっているReactの特定のライフサイクルメソッドの関数（`componentWillMount`）を使っていたら、React 17が出るとそのアプリケーションは動かなくなってしまう。これはReactのバージョンを固定してしまえばなんとかなるのかもしれませんが、追加の機能を入れようとして別のライブラリを入れようとしたときに、それがReact 16では動かなくて、React 17しかサポートしていないと、そのライブラリが使えないということになります。4Kブルーレイを見たいけど、うちの古いブラウン管テレビにはHDMI端子がなくてプレイヤー繋がられないわー、みたいな感じのことが起きます。

時間が経てば経つほどそのようなものが増えてきます。

セキュリティ的な問題

インターネットがなかった時代・接続しない時代は良かったんですが、今ではネットワーク前提のシステムが大幅に増えています。それにより、今までよりもセキュリティのリスクにさらされる機会は増えています。また、ネットワークに直接アクセスしないシステムであっても、USBメモリ経由でやってきたワームの攻撃を受けるなどがあります。

セキュリティに関しては存在（&攻撃方法）が報告されているセキュリティホールを放置して、システムを危険にさらされると、システムの提供元や開発元が責任を追求されることになります。「無能で説明できることに悪意を見出してはいけない」という格言があります。ただ、これらは「悪意」を持っていると誤解する人が多いからこそこういう言葉が生まれたのだと思います。あと、僕個人としては「時間不足で説明できることに無能を見出してはいけない」という持論があります。組み合わせると、忙しくて直せなかったとしても、「悪意があってユーザーを危険にさらしたのだ」と批判される恐れがあるということです。加害者になってしまうのです。

現代のシステムは数多くの部品で組み上げられています。ゼロからすべてのコードを自分で書くことはありません（ほとんど）。脆弱性に対する防御は社会的な仕組みが構築されています。特定のライブラリやツールに脆弱性があると、その攻撃手法などを報告する窓口があります。また、そこから開発元にこっそり連絡がいき（対策されていない時点での存在発表はそれ自体が加害行為になる）、脆弱性が修正されたバージョンのリリースと同時に公表、という流れです。

同時といっても、大きな問題は発表されたら即座に対策を取らないと、加害者になりかねません。そのためには、最新の修正済みのバージョンを入れる必要が出てきます。

問題はすごく古いバージョンのサポートまでは行われない点です。だいたい、大きめのOSSや商用のミドルウェアやライブラリを出しているベンダーであれば、きちんとサポートポリシーを定義して、バージョンごとのサポート期限を定めています。ただし、そこから外れてしまうと、よっぽど大きな問題でない限りは更新が提供されないことがあります。そのため、普段から更新を心がけていないと、必要な修正の入ったバージョンへの更新が遠すぎて、なかなか適用できないということもありえます。

バージョン管理をしなかった場合のデメリットまとめ

- 既存機能が動かなくなる
- 世間一般では普通の新規機能の追加が困難になっていく
- セキュリティの修正が提供されずに、システムに穴が開いたままになりかねない
- いざ、おおきなインシデントが発生したときに、その変更を取り込むのが困難になる

まとめ

なぜバージョンアップが必要なのか、そのための方法などについて紹介してきました。

常に更新しつづけるシステムであっても、バージョンの更新がおろそかになってしまうことがよくあります。バージョンアップは自社サービスであってもそうでなくても、保守として工数を確保して行う必要があります。影響を考慮してタイミングを見極めて行ったり、セキュリティ上必要であれば他の作業を止めてでも更新してデプロイなどスケジュールにも影響がありえる話になってきます。

適切にコントロールすれば痛みを減らせる分野でもありますので、本章の内容を頭の片隅に置いてもらえると幸いです。