WINGS
More Progress, More Future

速習 Vue.js 3

Composition API 編

―人気のJavaScriptフレームワークVue.jsを
素早く学んでWeb開発をスマートに ―

山田祥寛 著

- バージョン3の新標準 Composition APIでサンプルを刷新
- Viteによる骨組みの自動生成＆ビルド環境整備
- Vue Routerを使ったSPA＆ページ分割
- Piniaを使ったアプリデータの一元管理
- Vitestによる単体テストの自動化 etc ...

WINGSプロジェクト

目次

はじめに

対象読者

本書では、JavaScriptフレームワークであるVue.jsに初めて触れる人のための書籍です。導入から、SPA（Single Page Application）開発のための基礎知識までを手早く習得していただくことを目的としています。

その性質上、開発のための言語であるJavaScriptについては、最低限理解していることを前提に解説を進めます。本書でもできるだけ細かな解説を心掛けていますが、JavaScriptについてきちんと押さえておきたいという人は、「改訂新版JavaScript本格入門」（技術評論社）、「JavaScript逆引きレシピ 第2版」（翔泳社）などの専門書も合わせて参照してください。

なお、本文でも後述しますが、フレームワークにはさまざまな特性を持ったものがあります。本書の解説は、これまでjQueryなどを利用してフロントエンド開発をしてきたが、開発生産性／保守性の面で悩んでいる――「なんとか簡単化できないか！」というレベルの層を想定しています。

動作確認環境について

本書内の記述／サンプルプログラムは、次の動作環境で確認しています。異なる環境、バージョンでは、結果の表示も異なる可能性があるので、注意してください。

- Windows 10 Pro（64bit）
- Vue.js 3.2.31
- Vite 2.8.4
- Vue Router 4.0.12
- Pinia 2.0.11
- Google Chrome 100

なお、Vue.js 3では、アプリ実装の手段として

- Options API
- Composition API

と2種類のAPIを提供しています。本書では、より新しいComposition APIをベースに解説を進めるものとします。クラシカルなOptions APIの記法については、別巻「速習 Vue.js 3」（Amazon Kindle）を参照してください。

Part 1：イントロダクション

1.1 JavaScriptフレームワークとは？

Webアプリのフロントエンド開発に、今やJavaScriptは欠かすことができない存在です。もっとも、それはJavaScriptが優れた言語だから、というわけでは必ずしもありません。

JavaScriptの開発生産性は、決して高くはありません。それは、「型の認識が緩い」「JavaScript固有の癖が強い」など、言語そのものの問題でもありますし、「ブラウザーによって動作に違いがある」（クロスブラウザー問題）のような環境の問題でもあります。それでもJavaScriptを利用しているのは、単にメジャーなブラウザーで共通して動作するスクリプト言語がJavaScriptだけだからです。消去法的にJavaScriptを使わされている、という開発者の方々は、意外と多いのではないでしょうか。

そのようなJavaScriptの生産性を補う手段は、これまでにも提供されてきました。JavaScriptライブラリの導入です。ライブラリによってJavaScriptに不足している機能を補い、ブラウザー間での微細な差を埋めようというのです。

あまた存在するJavaScriptライブラリの中でも、何といっても、有名どころはjQueryでしょう。基本的なページの操作からアニメーション、Ajax通信、標準

JavaScriptの拡張など、JavaScriptにおけるUI開発を広くサポートする、優れたライブラリです。

　jQueryの便利さは、既にさまざまな資料で語られていますが、近年、フロントエンド開発がより高度化するに伴い、不足な面も目立ってきました。

1.2 jQueryの問題点とJavaScriptフレームワーク

たとえば、なにかしらの入力をトリガーにデータを取得し、その結果をページに反映させる、といった処理も、jQueryでは、「入力値を文書ツリーから取得し」「Ajax通信に引き渡し」「取得した結果を（たとえば）`<ul> <li>`要素に加工したものをページに埋め込む」という操作が必要になります。JavaScript側では、常に入出力にあたって、文書ツリーを意識しなければならないのです。

この交換は大概面倒なもので、レイアウトとコードの混在は、アプリ全体の見通しを悪くします。日常的に文書ツリーの操作を繰り返すSPA（Single Page Application）ともなれば、jQueryで実装するのは現実的ではないでしょう。

そこで求められたのが、文書ツリーとオブジェクト（JavaScript）との間を取り持つJavaScriptフレームワークの存在です。アプリ全体を俯瞰し、文書ツリーに変化があればオブジェクトに反映させ、逆にオブジェクトが変化すればテンプレートに反映させる――そのためのしくみを提供する存在です。これによって、アプリ開発者はテンプレート（HTML）、ロジック（JavaScript）それぞれの開発に集中できるので、コードの見通しも改善し、アプリの開発生産性／保守性が向上します。

［Note］銀の弾丸ではない

ただし、フレームワークも銀の弾丸ではありません。シンプルなページ開発には、依然としてjQueryのようなライブラリの手軽さは有効です。現時点

で「そんなに複雑なことはしていないから！」と感じるならば、右に倣えでフレームワークを導入する必要はありません（むしろ、その手間は害悪です）。

1.3 主なJavaScriptフレームワーク

JavaScriptフレームワークとして、執筆時点で有名なものには、以下のようなものがあります。

表：主なJavaScriptフレームワーク

名称	概要
Angular	Googleを中心に開発されているフルスタックフレームワーク
React	Facebook開発のフレームワークで、ビュー相当の機能を提供
Vue.js	ビューに特化したシンプルなフレームワーク

本書では、個々のフレームワークを詳らかに比較するのは避けますが、大雑把にまとめるならば、中規模以上の本格的なフロントエンド開発にはAngular、より小規模な開発にはVue.js、ライブラリの組み合わせによって規模に応じた柔軟な構成にできるのがReactといったところでしょうか。

Angularはビューからサービスまで幅広くサポートした高機能なフレームワークですが、反面、導入のハードルが高いというデメリットもあります。初期段階で学習すべき点も多く、これまでjQueryなどでライトにJavaScriptに接してきた人にとっては難しく感じるかもしれません。また、既にあるアプリに対して、後付けでAngularを導入するのは厄介です。最初からAngularを採用することを前提に、きちんと設計された環境でこそ、強みを発揮するフレームワークとも言えるでしょう。

一方、Vue.jsはビュー（見た目）の部分に特化したフレームワークなので、導入はカンタンです。学ぶこともごく限られています。Angularに比べると、原始的に感じるところもありますが、「既存のアプリ（たとえばjQueryで管理していたアプリ）が複雑になってきたので、フレームワークを導入したい」という場合には、気軽に後乗せできるという手軽さが強みです。

周辺ライブラリも充実しているので、あとからフロントエンド開発の範囲が広がってきた場合に、徐々にスパイラルアップしていくことも可能です [1]。

これらの特性のいずれが、より優れているというわけではありません。いずれも状況に応じた強み（とその裏側に弱み）があるというだけです。本書では、よりライトに利用できるVue.jsを解説していますが、これが絶対というフレームワークはありません。開発している（開発予定の）アプリの特性を見据えながら、適材適所の道具を選択してください。

［Note］参考書籍

AngularやReactについて、これ以上は本書では踏み込みません。詳しくは、以下の専門書を参照することをお勧めします。

- 「Angularアプリケーションプログラミング」（技術評論社）
- 「速習 React」（Amazon Kindle）

1. このような性質から**Progressive Framework**（段階的な成長に対応できるフレームワーク）とも呼ばれます。

Part 2：はじめてのVue.js

2.1 Vue.jsのインストール方法

Vue.js（以降はVueと表記します）を利用するには、以下の方法があります。

1. CDNからの取得
2. create-vueによる雛形の生成

2. のcreate-vueは、Vueアプリの実行に必要なライブラリからビルドツール、開発サーバーまでをまとめて用意してくれるツールです。あらかじめNode.jsなどをインストールする手間はありますが、本格的に開発するならば、まずはこの方法がお勧めです（本書でも、この方法を採用します）。

より手軽な方法は1. です。事前の準備もなく、そのまま開発を開始できるのがメリットですが、実行時にテンプレートの解析／変換などのオーバーヘッドが生じることから低速です。主に学習用途、小規模なアプリで利用することになるでしょう。具体的な方法は、別巻「速習 Vue.js 3」（Amazon Kindle）を参照してください。

［Note］Vue CLI

従来、VueではVue CLIと呼ばれるコマンドラインツールが提供されていましたが、現在ではメンテナンスモードの扱いとなっています。つまり、今後は新たな機能は追加されず、不具合の修正だけが行われます。新しい開発では、原

則としてcreate-vueを優先して利用するようにしてください。create-vueは、内部的にはViteというツールをベースにしています。

2.1.1 create-vueの準備とアプリの作成

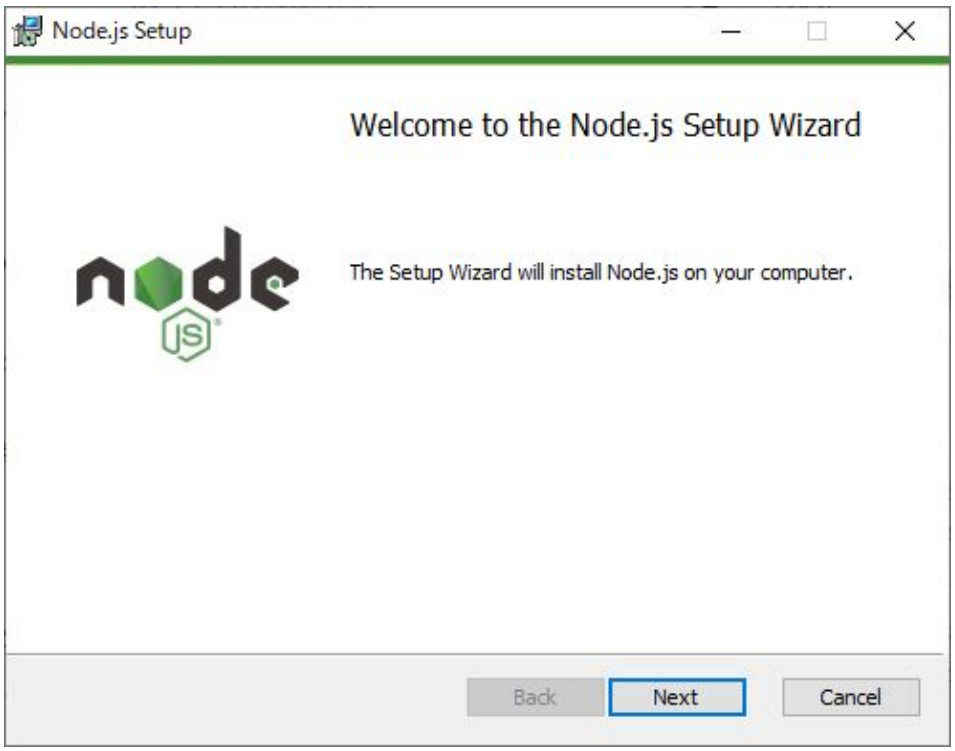
では、create-vueをインストールし、実際にアプリを作成／実行してみます。

［1］Node.jsをインストールする

create-vueは、Node.js上で動作するコマンドラインツールです。あらかじめ本家サイトからNode.jsをインストールしておきましょう。

ダウンロードしたnode-vx.xx.x-x64.msi（x.xx.xはバージョン番号）をダブルクリックすると、インストーラーが起動します。あとは、その指示に従って進めるだけなので、特に迷うところはないでしょう。本書では、執筆時点での最新安定版である16.14.2を利用しています。

図：Node.jsのインストーラー



本家サイトからインストーラー（node-v16.14.2-x64.msi）をダウンロードできたら、これを実行するだけです。ウィザードが起動するので、あとは既定の設定のままに進めていくだけなので、迷うところはありません。

create-vueそのものは、あとからプロジェクトを作成する際に自動的にインストールされるので、これで開発のための準備は完了です。

［2］プロジェクトを作成する

create-vueでは、アプリをひとつのフォルダー配下で管理します。このフォルダー配下で管理されるファイル群のことを**プロジェクト**と呼びます。プロジェクトを作成するには、プロジェクトを作成したフォルダーに移動した上で、npm initコマンドを実行します（ここでは「c:¥data」配下にプロジェクトを作成するものとします）。

```
> cd c:¥data　（カレントフォルダーを移動）
> npm init vue@latest　（プロジェクトを作成）
Need to install the following packages:
  create-vue@latest
Ok to proceed? (y)
```

最初の実行ではcreate-vueをインストールするかを訊かれるので、Yes（既定）のままで先に進みます。以降も、以下のような設問が表示されるので、順に答えていきましょう。プロジェクト名を「quick-vue」とするほかは、まずは、No（既定）のままで進めて構いません。

- Project name：プロジェクト名
- Add TypeScript?：TypeScript[1]を利用するか
- Add JSX Support?：JSX[2]を利用するか
- Add Vue Router for Single Page Application development?：ルーター（Part 11）を利用するか
- Add Pinia for state management?：状態管理ツール（Part 12）を利用するか
- Add Vitest for Unit Testing?：ユニットテストツール（Part 13）を利用するか
- Add Cypress for both Unit and End-to-End testing?：E2Eテスト[3]ツールを利用するか
- Add ESLint for code quality：Lint[4]を利用するか

すべての設問に答え終わると、プロジェクトの生成が開始されます。

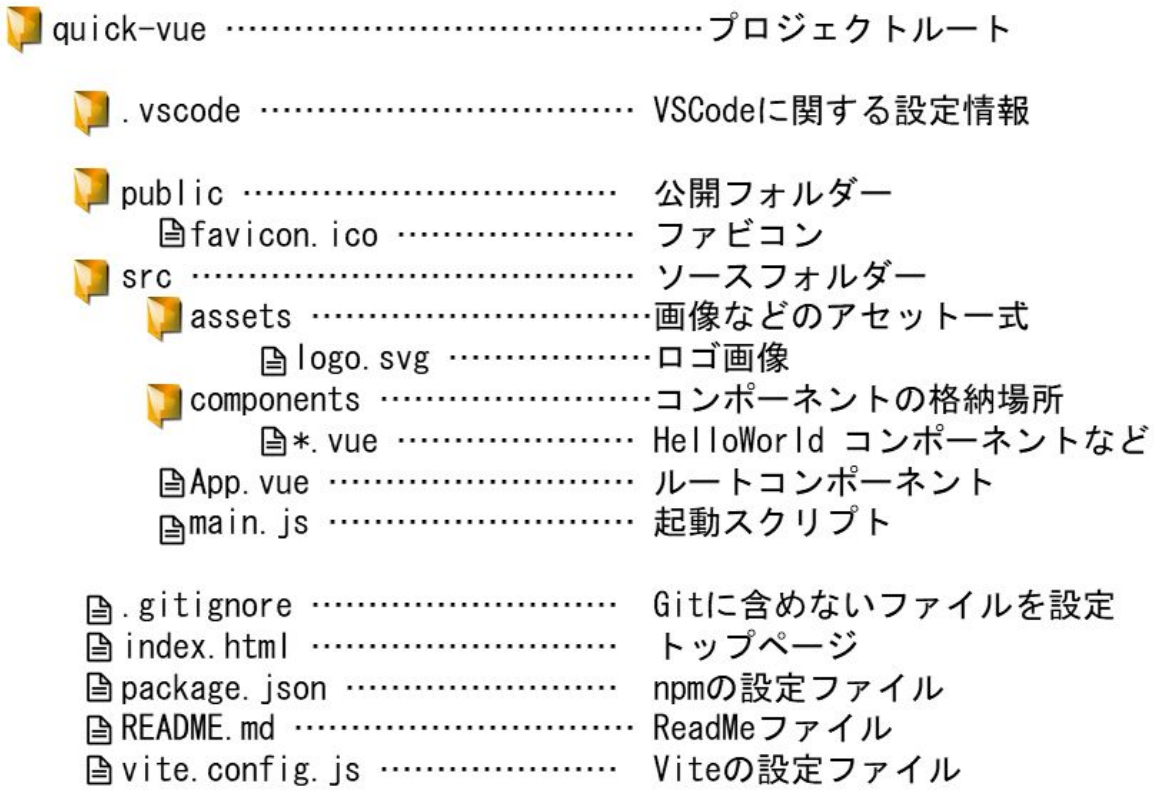
```
Scaffolding project in C:¥data¥quick-vue...
Done. Now run:
  cd quick-vue
  npm install
  npm run dev

>
```

上のような結果が表示されたら、プロジェクトは正しく設定できています。

［3］プロジェクトフォルダーの内容を確認する

/quick-vueフォルダー配下に生成されるフォルダー／ファイル構造も確認しておきましょう（主なものを抜粋しています）。

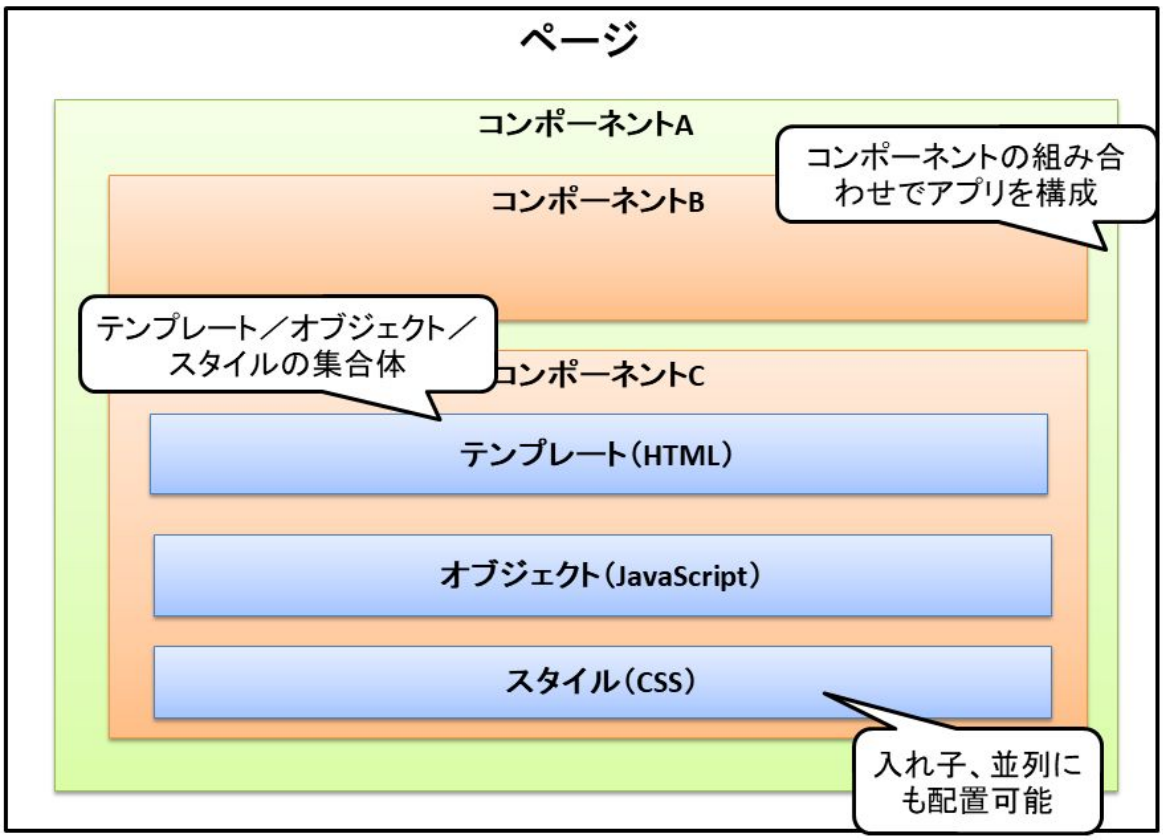
図：create-vueで作成されたアプリ（主なフォルダー／ファイル）



/srcフォルダー（特に、その配下の/componentsフォルダー）が、Vueアプリの本体です。この後も、主に/srcフォルダー配下のファイルを編集していきます。

［Note］コンポーネント指向

Vueアプリを構成するさまざまな要素の中で、中核となる存在がコンポーネント（component）です。コンポーネントとは、アプリ（ページ）を構成するUI部品のこと。見た目（テンプレート）、ロジック（JavaScriptのコード）、スタイルなどから構成されます。



Vueでは、これら部品化されたコンポーネントを組み合わせることで、ページを構成していくのが基本です。これを**コンポーネント指向**と言います。

［4］ライブラリをインストールする

［2］の段階では、プロジェクトの骨組みを生成しただけでライブラリは組み込まれていません。実行に先立って、以下のコマンドを実行し、ライブラリをプロジェクトに組み込んでおきましょう（当然、最初の一回だけ行えばよい手順です）。

```
> cd c:¥data¥quick-vue　（プロジェクトルートに移動）
> npm install　（ライブラリをインストール）
added 105 packages, and audited 106 packages in 17s
```

```
  run `npm fund` for details
found 0 vulnerabilities
```

上のような結果が表示されれば、ライブラリは正しく組み込まれています。

> **［Note］コマンドの実行場所**
>
> この後もさまざまな箇所でnpmコマンドを実行していきますが、実行場所はプロジェクトルートでなければなりません（本書であれば「c:¥data¥quick-vue」です）。以降、カレントフォルダーを移動する手順は省きますが、実行に際しては、カレントフォルダーが正しい場所にあるかを再確認してください。

［5］アプリを起動する

プロジェクトには、既定で最低限のアプリが用意されています。実行して、動作を確認してみましょう。アプリを起動するには、npm run devコマンドを利用します。

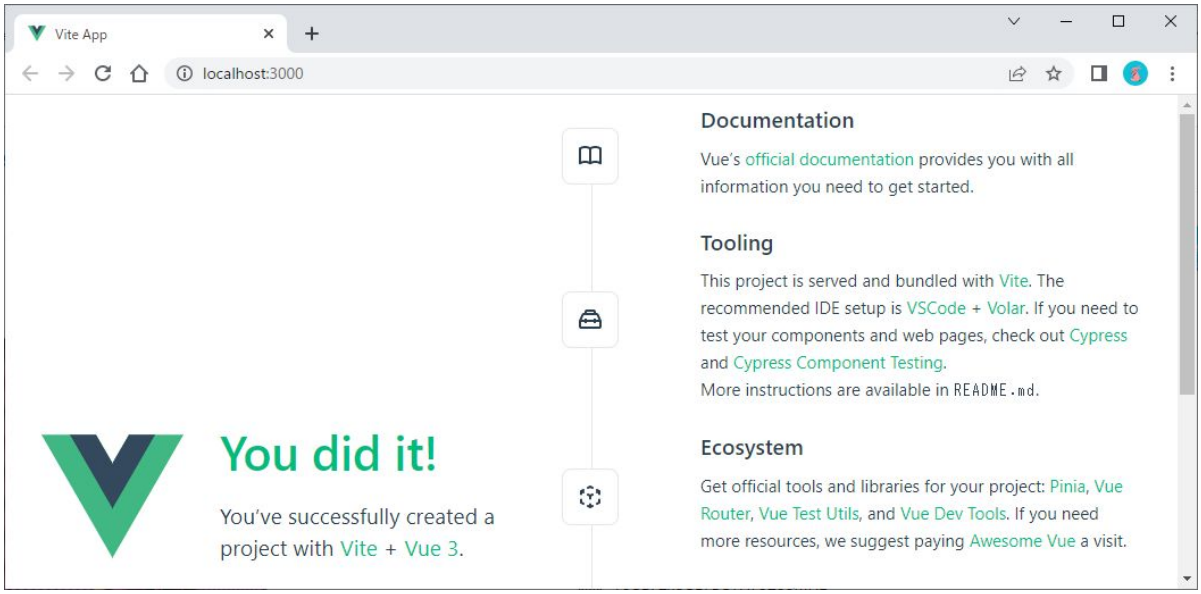
```
> npm run dev     （アプリを起動）
…中略…
  vite v2.9.1 dev server running at:
  > Local: http://localhost:3000/
  > Network: use `--host` to expose
  ready in 402ms.
```

上のような結果が表示されれば、アプリを実行するための開発サーバーが起動できています。ブラウザーを起動し、「http://localhost:3000」でアクセスしてみましょ

う。

以下のようなページが表示されれば、アプリは正しく動作しています。開発サーバーは［Ctrl］＋［c］で終了できます。

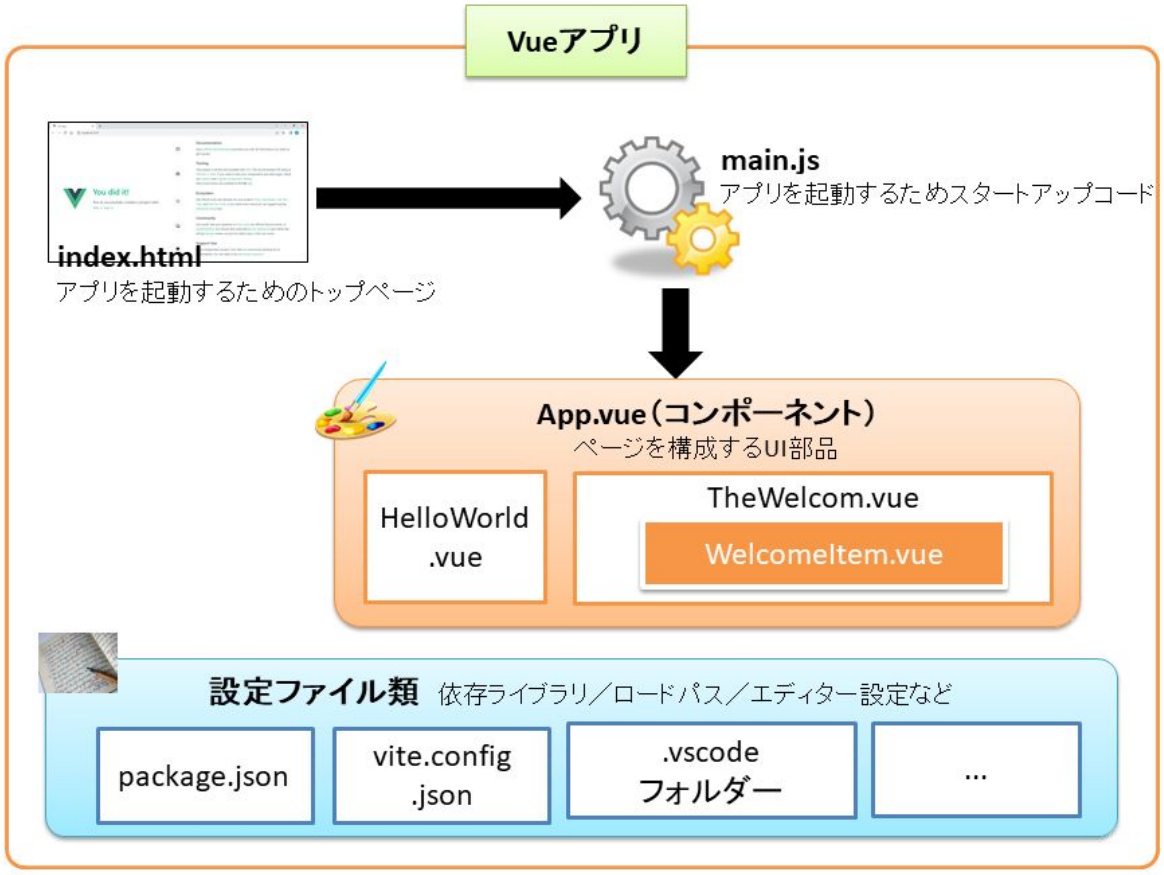
図：create-vue既定のトップページ



2.2 サンプルアプリの内容を確認する

はじめてのVueアプリを実行できたところで、プロジェクト既定で用意されているファイルを読み解いていきましょう。

図：サンプルアプリの大まかな構造



複数のファイルが連携していますが、ここで注目すべきはindex.html→main.js→App.vueのラインです。順番に見ていきましょう。

2.2.1 トップページの準備 - index.html

アプリが動作するメインページです。

［リスト］index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <link rel="icon" href="/favicon.ico" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Vite App</title>
</head>
<body>
  <!--a. コンポーネントを反映する領域 -->
  <div id="app"></div>
  <!--b. アプリをインポート -->
  <script type="module" src="/src/main.js"></script>
</body>
</html>
```

a.が、あとでVueアプリ（コンポーネント）の実行結果を反映するための領域です。「id="app"」となっていることだけを覚えておいてください。

b.は、アプリのエントリーポイント（＝実行基点）です。main.jsを基点に、アプリを構成するコンポーネントやVue本体が芋づる式にインポートされます。

2.2.2 アプリを起動するためのエントリーポイント - main.js

Vueアプリを実行する際に、最初に呼び出されるファイルです（index.htmlでも最初にインポートされたファイルです）。

［リスト］main.js

```
// a. Vueライブラリのインポート
import { createApp } from 'vue'
// b. アプリ本体のインポート
import App from './App.vue'
// c. Vueアプリの起動
createApp(App).mount('#app')
```

a.～**b.** は、アプリを動作するためのライブラリをインポートしています。**a.** がVueを起動するために最低限必要な関数、**b.** はアプリ固有のファイルです。

createAppは、**アプリインスタンス**を生成するための関数です（**c.**）。アプリの基点となるコンポーネント（ここではApp）を渡します。

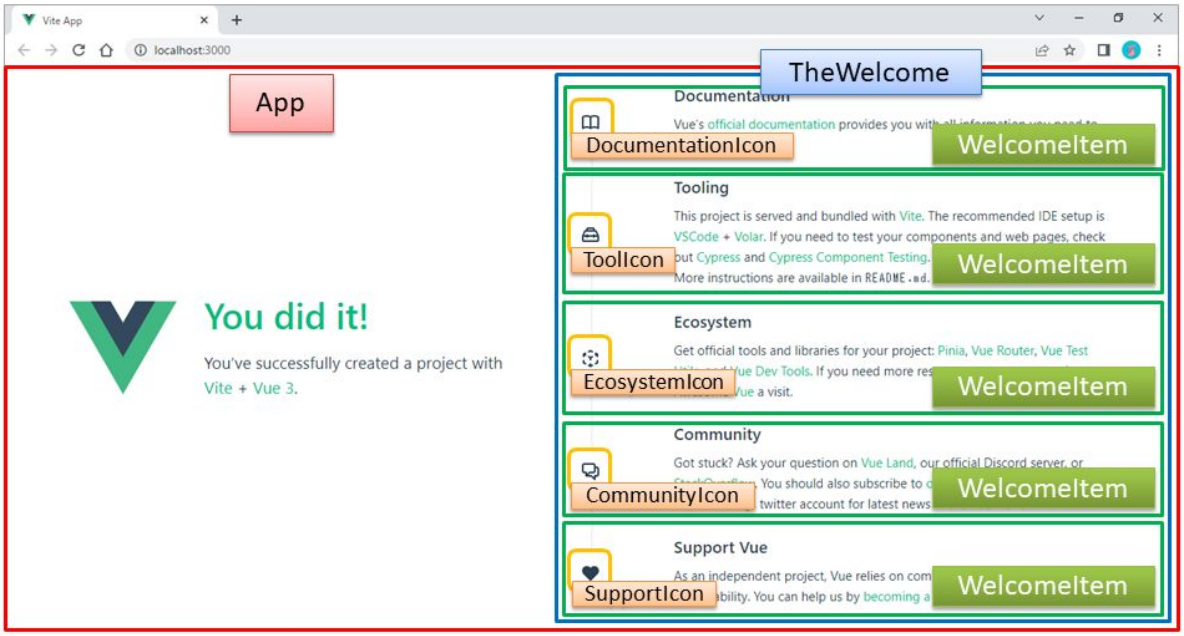
アプリインスタンスとは、Vueアプリを管理するためのオブジェクトのこと。アプリの起動／終了、アプリの構成管理などの役割を担います。この例であれば、mountメソッドでVueアプリを「id="app"である要素」上で起動しています。これをアプリを**マウントする**、とも言います。

> ［構文］アプリの起動
>
> createApp(*comp*).mount(*el*)
>
> - comp：基点となるコンポーネント
> - el：Vueアプリを適用する要素（セレクター式）

2.2.3 アプリを構成するコンポーネント - App.vue

2.1.1項でも触れたように、Vueはコンポーネント指向なフレームワークです。Vueアプリとは、ひとつ以上のコンポーネントの集合体、と言い換えても良いでしょう。初期画面も、実は以下のような複数のコンポーネントが互いに関連しながら構成されています。

図：トップページの構造



コンポーネント間の連携についてはPart 8で改めるとして、Part 3～7では、まずはページ全体が単一のコンポーネントで構成されるシンプルなアプリを扱っていきます。ここでも、まずは最上位のコンポーネントであるAppコンポーネントのコードだけを引用しておきます。

［リスト］App.vue

```
<!--コンポーネント定義のコード-->
<script setup>
```

```
import HelloWorld from './components/HelloWorld.vue'
import TheWelcome from './components/TheWelcome.vue'
</script>

<!--テンプレート定義-->
<template>
  <header>
    <img alt="Vue logo" class="logo"
      src="./assets/logo.svg" width="125" height="125" />
    <div class="wrapper">
      <!--入れ子のコンポーネントを呼び出し（Part 8を参照）-->
      <HelloWorld msg="You did it!" />
    </div>
  </header>

<main>
    <TheWelcome />
  </main>
</template>

<!-- スタイル定義 -->
<style>
@import './assets/base.css';

#app {
  max-width: 1280px;
  margin: 0 auto;
  padding: 2rem;
  font-weight: normal;
```

```
}
...中略...
</style>
```

Vue（create-vue）の世界では、コンポーネントに関わる情報をひとつのファイル（.vueファイル）としてまとめるのが基本です。具体的には、以下のような要素から構成されます。

- `<script>`要素：コンポーネント定義（JavaScript）
- `<template>`要素：コンポーネントの見た目（HTML）
- `<style>`要素：コンポーネントに適用するスタイル（CSS）

`.vue`ファイルを用いることで、ロジック（JavaScript）、見た目（HTML）、デザイン（CSS）をまとめて管理できるわけです。その記法から、.vueファイルのことを**単一ファイルコンポーネント**（SFC：Single File Component）と呼ぶこともあります。

2.2.4 `.vue`ファイルの構文

`<script>`／`<template>`要素の中身については、まさにVueのキモなので、以降のPartで徐々に解説していきます。本項では、.vueファイルを利用するにあたって、個々の要素の大枠（基本ルール）と`<style>`要素についてのみまとめておきます。

`<script>`要素にはsetup属性を付与する

Vueでは、大まかに以下の形式でコードを記述できます。

1. Options API
2. Composition API
3. Composition API（省略記法）

create-vueでは、標準で3. の形式を採用しており、本書でもこれを採用しています。setup属性は、3. の形式を有効にするためのスイッチと考えておけば良いでしょう。その他の記法を理解したい場合には、別巻「速習 Vue.js 3」（Amazon Kindle）を合わせて参照してください。

<script>／<template>要素はひとつだけ

.vueファイルでは、<script>／<template>要素はそれぞれひとつ、<style>要素だけが複数列記できます。ただし、例外があって、setup付きの<script>要素とsetupなしの<script>要素だけは併記できます。具体的な例については8.2.4項で改めます。

<style>要素のscoped属性

<style>要素にscoped属性を付与することで、現在のコンポーネントでのみ有効なスタイルを設定できます。一般的には、コンポーネント固有のスタイルが他のコンポーネントに影響するのは望ましくないので、scoped付きの<style>要素でスタイル定義するのが吉です。このようなスタイルを**ローカルスタイル**と言います（本書サンプルもすべてローカルスタイルとしています）。

```
<style scoped>
  …ローカルスタイル…
</style>
```

scopedなしの<style>要素（グローバルスコープ）は、一般的には、最上位のコンポーネントにまとめるべきです。

2.3 学習を進める前に

さて、次のPartからはVueの基本要素を個別に取り上げながら、より詳細に踏み込んでいきます。以降に進む前に、本書サンプルの入手／実行方法について解説しておきます。

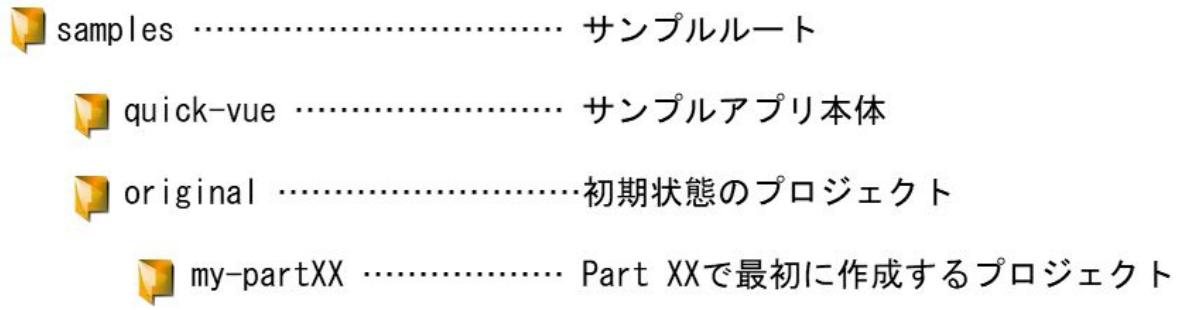
2.3.1 サンプルファイルの入手方法

本書のサンプルプログラムは、以下のページからダウンロードできます。サンプルの動作を手元で確認したい場合などにご利用ください。

https://wings.msn.to/index.php/-/A-03/WGS-JSF-007/

ダウンロードサンプルは、以下のような構造となっています。

図：ダウンロードサンプルの構造



/originalフォルダー配下の/my-partXXフォルダーは、Part 2、11～13で作成する初期プロジェクトを収録しています。自動生成されるコードは将来的に変化する可

能性があるので、執筆時点での内容を収録しています。自分で一から生成しても構いませんが、もしも内容に食い違いがある場合には参考にしてください。

/quick-vueフォルダーが、本書サンプルをまとめたプロジェクトです。任意のフォルダー（たとえば「c:¥data」フォルダー）にコピーして、以下のコマンドを実行してください。これでライブラリがインストールされます。

```
> cd c:¥data¥quick-vue　（プロジェクトルートに移動）
> npm install　（ライブラリをインストール）
```

2.3.2 サンプルファイルの利用方法

コンポーネントファイル（.vueファイル）は、Partごとに/src/components/pXXフォルダーに配置されています（XXはPart番号）。実行に際しては、main.jsを編集して、動作したいコンポーネントをマウントするようにしてください（コメントアウトしたコードが用意されているので、これを切り替えるだけです）。

図：main.jsを編集

あとは、以下のコマンドで開発サーバーを起動＆ブラウザーにアクセスすることで、サンプルを実行できます。

```
> npm run dev　（開発サーバーを起動）
```

> ［Note］Bootstrap
>
> 本書のサンプルでは、ページの見た目を整えるため、index.html（トップページ）にBootstrapを適用しています。Bootstrapを利用することで、class属性を付与するだけで見栄えのするデザインを手軽に実装できます。

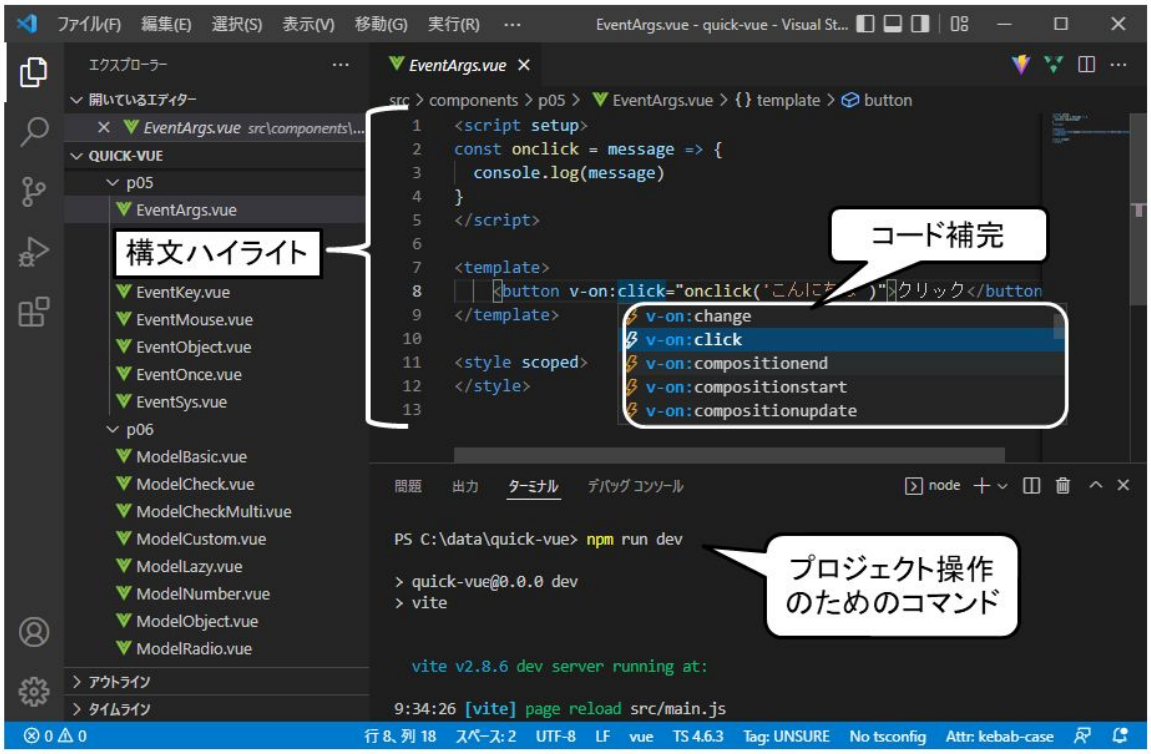
2.3.3 補足：学習／開発に便利なツール

Vueアプリは、メモ帳などのエディターに、PowerShellなどのCUIシェルなど、プラットフォーム標準のツールだけでも開発できます。しかし、開発の効率性を考えれば、

Visual Studio Code（以降、VSCode）のようなコードエディターを導入することをお勧めします。

VSCodeはWindows、macOS、Linuxとマルチな環境で動作し、さまざまな言語／フレームワークにも対応していることから（Vueに限らず）、近年よく利用されているエディターです。VSCodeに慣れることは、きっと他の言語／環境での開発にも役立つはずです。

図：VSCodeのメイン画面



そして、VSCodeを利用するならば、その拡張機能であるVolarも導入しておくことをお勧めします。VolarはVue公式で開発されているツールで、.vueファイルに対して構文ハイライト、コード補完、コード検査などの基本機能を提供してくれます。

1. altJS（JavaScriptの代替言語）の一種。静的に型を指定できる特徴があります。↵

2. JavaScriptのコード内でタグ構文を利用するためのしくみ。↵

3. End to Endテストは、アプリを本番環境に近い形でテストすること。**インテグレーションテスト**とも言います。↵

4. 「べからず」なコードを検出するためのツール。↵

Part 3：Vueアプリの基本ルール

Vueアプリの構造を理解できたところで、ここからは実際にコードを記述しながら、アプリ（主にコンポーネント）開発の理解を深めていきましょう。

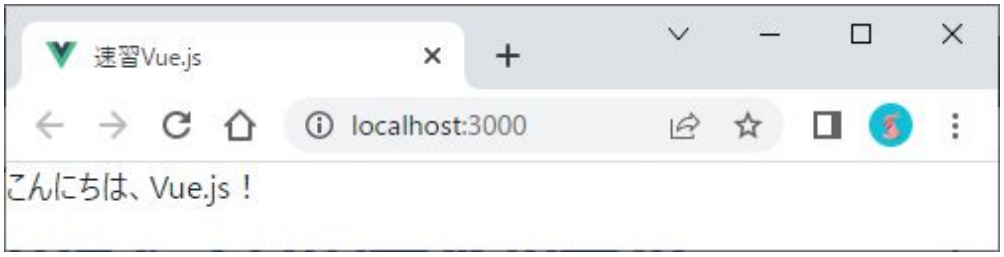
3.1 Vueアプリで「Hello, World」

まず紹介するのは、おなじみのHello, Worldサンプル――Vue経由で「こんにちは、Vue.js！」というメッセージを表示する例です（2.3.2項でも触れたように、コードは/src/components/p03フォルダーに保存されています）。

［リスト］FirstApp.vue

```
<script setup>
import { ref } from 'vue'
// a. ビュー変数を準備
const message =  ref('こんにちは、Vue.js！')
</script>

<template>
  <!--b. 準備済みの値を出力-->
  <p>{{ message }}</p>
</template>
```

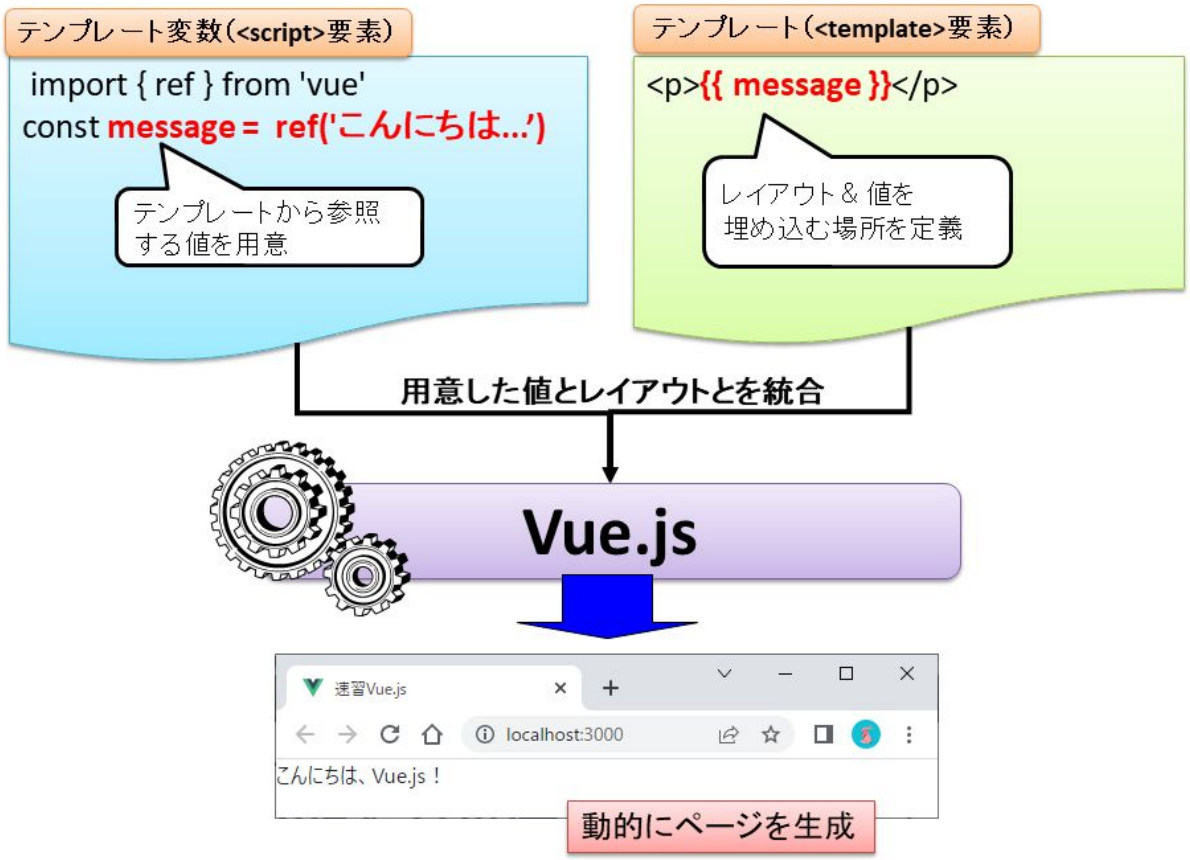
図：あらかじめ用意されたメッセージを表示



a. テンプレート変数を準備する

<script>要素配下で最低限行うべきは、テンプレートで利用する変数（便宜的に**テンプレート変数**と呼びます）を準備することです。Vueの世界では、アプリで利用する値をテンプレート変数として用意しておいて、テンプレート側でこれを参照する、という役割分担が基本です。このようなデータ割り当てのしくみのことを**データバインディング**と言います。

図：データバインディング



テンプレート変数を定義するには、ref関数に初期値を渡すだけです。

> **［構文］ref関数**

> ref(*init*)
>
> - init：初期値

この例であれば、初期値「こんにちは、Vue.js ！」であるテンプレート変数messageを定義したことになります。ここで定義しているテンプレート変数はひとつだけですが、もちろん複数の変数を定義するならば、ref関数を列記するだけです。

b. テンプレート変数にアクセスする

テンプレート側でテンプレート変数にアクセスするには、{{...}}という構文を利用します。これを**Mustache（マスタッシュ）構文**と言います。

［Note］口髭構文

Mustacheは英語で「口髭」という意味です。デリミターである「{」を横に倒してみると、口髭に似ていることから、そのように呼ばれます。

ここでは、{{ message }}で、テンプレート変数messageの値をそのまま引用しているだけですが、{{...}}には任意のJavaScript式を表すことも可能です。たとえば以下は、いずれも妥当なMustache式です。

```
{{ 2 + 3 }}
{{ value + 4 }}
```

```
{{ message.substring(1) }}
{{ Math.abs(-10) }}
```

{{…}}では、Math、Dateなどの組み込みオブジェクトにもアクセスできる点に注目です。Vueでは、これらの組み込みオブジェクトをMustache式でも利用できるよう、あらかじめ登録しているからです（よって、Vueが登録していない自前のオブジェクトに、{{…}}からアクセスすることはできません）。

3.2 リアクティブシステム

テンプレート変数は、テンプレートに反映されてそれで終わり、というわけではありません。Vueによって監視され、元の値に変化があった場合にはテンプレート側にも反映されます。このようなVueのしくみを**リアクティブシステム**と言います。リアクティブ――反応できる、という意味です。

3.2.1 リアクティブシステムの例

まずは、テンプレート変数がリアクティブであることを確認してみましょう。

［リスト］ReactiveVar.vue

```
<script setup>
import { ref } from 'vue'
// a. 現在時刻を表すテンプレート変数current
const current =  ref(new Date().toLocaleTimeString())
// b. currentの値を1000ミリ秒おきに更新
setInterval(() => {
  current.value = new Date().toLocaleTimeString()
}, 1000)
</script>

<template>
  <!--c. current値を取得 -->
  <p>{{ current }}</p>
</template>
```

図：現在時刻を1000ミリ秒ごとに取得

setInterval関数を経由して1000ミリ秒間隔で、currentの値を更新しています。サンプルを実行すると、確かにcurrentの変化に反応して、ページ（テンプレート）の側も変化していることが確認できます。

先ほどは、「ref関数でテンプレート変数を定義する」とだけ説明しましたが、リアクティブシステムの観点からは、「自分の変化をVueに通知するための機能を持った」オブジェクトを生成していたわけです。このように、Refで生の値を修飾することを**ラップ**すると言い、生成されたオブジェクトを**Ref変数**とも言います。

Ref変数は、更新する際にも注意が必要です。というのも、直接値を代入することはできません。たとえば**b.**の太字は、以下のように書いてはいけません [1]。

```
current = new Date().toLocaleTimeString()
```

Ref変数そのものを置き換えてしまうので、リアクティブの性質が解除されてしまうのです。元の例のように、valueプロパティ経由で更新することで、Ref変数配下の値だけを置き換えできます。

ちなみに、テンプレートからRef変数を参照する際には、Refの値が暗黙的に取り出されるので（これを**アンラップ**と言います）、「current.value」のようにする必要はありません（**c.**）。

3.2.2 複数値のリアクティブ化

ref関数が単一の値をリアクティブ化するのに対して、複数の値をまとめてリアクティブ化するreactive関数もあります。試しに先ほどのサンプルをreacitive関数で書き換えてみましょう。

［リスト］ReactiveMulti.vue

```
<script setup>
import { reactive } from 'vue'
// a. Reactive変数を準備
const data =  reactive({
  current: new Date().toLocaleTimeString()
})
// b. Reactive変数を更新
setInterval(() => {
  data.current = new Date().toLocaleTimeString()
}, 1000)
</script>

<template>
  <!--c. Reactive変数を参照-->
  <p>{{ data.current }}</p>
</template>
```

reactive関数（a.）には、オブジェクトリテラルを渡すことで、リアクティブ化されたオブジェクトが返されます（そのようなオブジェクトをRef変数と区別して、**Reactive変数**とも言います）。

> **［構文］reactive関数**
> reactive(*obj*)
> - obj：リアクティブ化するオブジェクト

Reactive変数では、（Ref変数と異なり）「data.current」のようにプロパティを直接変更できる点に注目です（「data.current.value」ではありません！）。ただし、オブジェクトそのもの（data）の参照があるので、コードそのものは短くなるわけではありません。テンプレートからも「data.current」のように記述しなければならないので（c.）、コードを短くする目的でreactive関数を利用する意味はほとんどありません。関連する値をひとつのオブジェクトで管理したい場合に、reactive関数を利用することになるでしょう。

3.3 算出プロパティとメソッド

先ほども触れたように、テンプレートには、{{...}}を用いることで、任意のJavaScript式を埋め込むことができます。たとえば以下は、与えられたemailプロパティ（メールアドレス）の値から「@」の前だけを取り出し、小文字に変換する例です。

［リスト］ComputeBad.vue

```
<script setup>
import { ref } from 'vue'
const email = ref('Yamada@example.com')
</script>

<template>
  <p>{{ email.split('@')[0].toLowerCase() }}</p>
</template>
```

▼結果

```
yamada
```

これは文法的には正しいコードですが、望ましくはありません。複雑な式はテンプレートの可読性を損ないますし、修正も困難になります（複数の箇所で同じ式を記述する場合にはなおさらです！）。

テンプレートでは単純なプロパティの参照に留め、演算やメソッドの呼び出しはできるだけコード側に委ねるべきです。このような用途で有用なのが、**算出プロパティ**です。

3.3.1 算出プロパティ

まずは、具体的な例を見てみましょう。以下は、先ほどの例を算出プロパティを使って書き換えたものです。

［リスト］ComputeGood.vue

```
<script setup>
import { ref, computed } from 'vue'
const email = ref('Yamada@example.com')
// a. 演算した結果を取得する算出プロパティ
const localEmail = computed(() => {
  return email.value.split('@')[0].toLowerCase()
})
</script>

<template>
  <!--b. 算出プロパティを参照 -->
  <p>{{ localEmail }}</p>
</template>
```

算出プロパティとは、言うなれば、既存のプロパティを演算（算出）した結果を取得するためのゲッター（getter）です。computed関数に演算のためのアロ

–関数として指定することで定義できます（もちろん、複数の算出プロパティを定義したければ、computed関数を列挙するだけです。a.）。

> **［構文］computed関数**
> computed(*func*)
> - func：値算出のための関数

算出プロパティからRef変数を参照するには、先ほどと同じく「変数名.value」とするだけです（太字部分）。

算出プロパティは「プロパティ」なので、参照する際にも関数呼び出しの()などは不要です（b.）。

3.3.2 メソッド

算出プロパティは、単なるメソッドとして表してもほぼ同じです。たとえば、以下は先ほどの例をメソッドを使って書き換えています。

［リスト］MethodBasic.vue

```
<script setup>
import { ref } from 'vue'
const email = ref('Yamada@example.com')
// a. email変数の値を加工するlocalEmailメソッドを定義
const localEmail = () => {
  return email.value.split('@')[0].toLowerCase()
```

```
}
</script>

<template>
  <!--b. メソッドを実行-->
  <p>{{ localEmail() }}</p>
</template>
```

アロー関数を修飾していたcomputed関数を解除しただけです（a.）。メソッドなので、テンプレート側でも関数の呼び出しの()が加わっている点に注目です（b.）。

3.3.3 算出プロパティとメソッドの違い

アプリを実行すると、算出プロパティ（ComputeGood.vue）とメソッド（MethodBasic.vue）とで、同じ結果が得られるはずです。

双方が異なるのは、更新のタイミングです。違いを理解するために、もうひとつ、サンプルを見てみましょう。

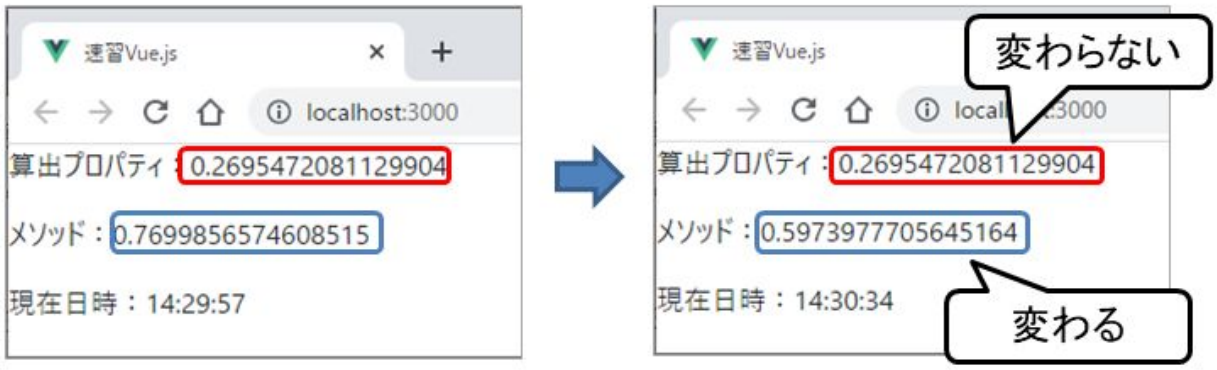
以下は、それぞれ算出プロパティ／メソッドで乱数を表示するためのコードです。また、1000ミリ秒間隔で現在時刻を更新します。

［リスト］MethodCompute.vue

```
<script setup>
import { ref, computed } from 'vue'
// 算出プロパティ経由で乱数を取得
const randomc = computed(() => Math.random())
```

```
// メソッド経由で乱数を取得
const randomm = () => Math.random()
const current =  ref(new Date().toLocaleTimeString())
// 1000ミリ秒置きに現在時刻を更新
setInterval(() => {
  current.value = new Date().toLocaleTimeString()
}, 1000)
</script>

<template>
  <p>算出プロパティ：{{ randomc }}</p>
  <p>メソッド：{{ randomm() }}</p>
  <p>現在日時：{{ current }}</p>
</template>
```

図：メソッドの結果だけが変更（算出プロパティは変わらない）



setIntervalによる更新で、メソッドに紐づいた<p>要素だけが変化しています。

これは、メソッドが再描画（＝この場合であれば現在日時の書替）に際して常に評価されるのに対して、算出プロパティはそれが依存するRef／Reactive変

数が変更された場合にのみ評価されるためです。

この場合、算出プロパティrandomcは、他のRef／Reactive変数に依存しないので、初回に呼び出された後は、二度と呼び出されることはありません。再描画のたびにすべての式が再評価されるのは無駄なので、まずは算出プロパティを基本とし、値を常に更新したいという意図がある場合にだけメソッドを使うと良いでしょう。

---

1. この例ではconst定数なので、そもそも定数違反になるのですが、let変数にしても不可であるのは同じです。↵

Part 4：ディレクティブとデータバインディング

ディレクティブとは、v-～で始まる特別な属性のこと。directive（指令）という名前の通り、Vueになんらかの指示を渡すためのしくみです。{{...}}と並んで、テンプレート操作の中核となるしくみなので、Part 4～7にかけて主なディレクティブについて解説していきます。

ディレクティブにはさまざまな種類がありますが、まずはデータバインディングに関わるディレクティブからです。

4.1 ディレクティブによるデータアクセス - v-text

JavaScript式を埋め込むのに、（{{...}}の代わりに）v-textディレクティブを利用しても構いません。v-textは、現在の要素配下を指定された式の値で置き換えます。よって、以下のコードは、いずれも意味的に等価です。

［リスト］TextBasic.vue

```
<template>
  <p>{{ message }}</p>
  <p v-text="message"></p>
</template>
```

{{...}}、v-textディレクティブは、いずれを利用しても構いませんが、コードの可読性という意味では、アプリの中では統一すべきでしょう。また、v-textディレクティブは配下のテキストを丸ごと置き換えるので、テキストの一部を置き換えるような用途では{{...}}を利用しなければなりません（そうした意味でも、記述の簡便さを考慮しても、本書では{{...}}を優先して採用します）。

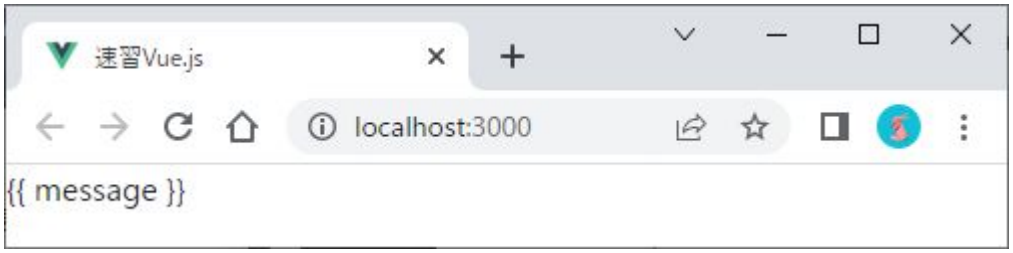
4.2 {{...}}式を無効化する - v-pre

{{...}}をJavaScriptの式としてではなく、そのまま文字列として表示したい（＝式を無効化したい）場合には、v-preディレクティブを利用します。たとえば先ほどの例で、以下のようにテンプレートを書き換えてみましょう。

［リスト］PreBasic.vue

```
<template>
  <p v-pre>{{ message }}</p>
</template>
```

図：{{...}}式がそのまま表示される



v-pre配下の{{...}}式は、そのまま出力されることが確認できます。

4.3 文字列をHTMLとして埋め込む - v-html

Vueでは、式によるテキストの埋め込みにも、セキュリティ的な考慮がなされています。たとえば、以下のような例を見てみましょう。

［リスト］HtmlBasic.vue

```
<script setup>
import { ref } from 'vue'
const message = ref(`<h1>こんにちは</h1>
  <span><a href="https://jp.vuejs.org/">Vue.js！</a></span>`)
</script>

<template>
  <p>{{ message }}</p>
</template>
```

図：HTML文字列がそのまま表示される

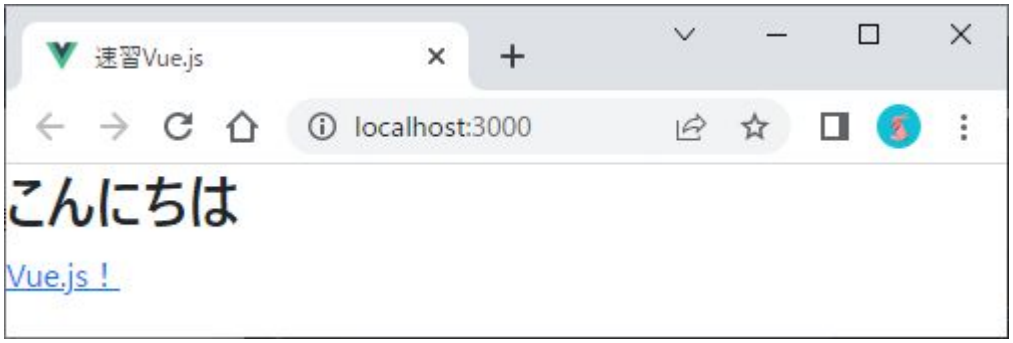
文字列が埋め込まれる際に、内部的にエスケープ処理されて、（タグではなく）単なる文字列としてページに埋め込まれています。意図しないHTMLをページに混入させることはクロスサイトスクリプティング脆弱性の原因ともなるので、こ

れはあるべき挙動です。しかし、時として、動的にHTMLを生成し、ページに反映させたいというケースもあります。

これには、太字を以下のように書き換えてください。

```
<p v-html="message"></p>
```

図：HTML文字列をHTMLとしてページに反映



果たして、v-htmlディレクティブで指定された式の値は、エスケープされずにそのまま要素配下に埋め込まれることが確認できます。

{{...}}式やv-textディレクティブが要素オブジェクトのtextContentプロパティを設定するのに対して、v-htmlディレクティブはinnerHtmlプロパティを設定する、と考えると判りやすいかもしれません。

［Note］信頼できるコンテンツにだけ利用する

任意の――特に、ユーザー／外部サービスからの入力をv-htmlディレクティブで埋め込むのは、深刻な脆弱性の原因となります。v-htmlディレク

ティブは、信頼できる（＝適切なエスケープ処理が為されていることが判っている）コンテンツに対してのみ利用してください。

4.4 属性値にJavaScript式を埋め込む - v-bind

属性に対して式の値を埋め込むのに{{...}}は利用できません。たとえば、以下のコードは正しく動作しません。

```
<a href="{{ url }}">サポートサイト</a>
```

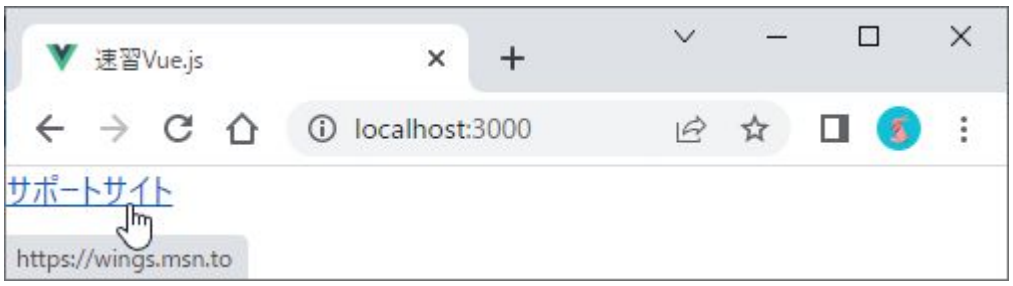
4.4.1 属性操作の基本

属性値の操作には、v-bindディレクティブを利用してください。以下は、Ref変数として用意されたアドレスをアンカータグに反映させる例です。

［リスト］BindBasic.vue

```
<script setup>
import { ref } from 'vue'
const url = ref('https://wings.msn.to/')
</script>

<template>
  <a v-bind:href="url">サポートサイト</a>
</template>
```

図：リンクが動的に生成された



「v-bind:属性名="値"」のように、ディレクティブ名と属性名はコロン（:）区切りで表記する点に注目です。

> **［Note］ディレクティブの引数**
>
> Vueの文法的にはコロンの後方は、ディレクティブの引数です。細かい点ですが、ディレクティブによっては引数を受け取るものがあること、その場合はコロン区切りで表記することを覚えておきましょう。

なお、v-bindはよく利用する、という理由から、省略構文も用意されています。リストの太字部分は、以下のように表しても同じ意味です。ただし本書では、初心者にも意図を汲み取りやすいという理由から、省略構文は避けて、v-bind:～構文を用いていきます。

```
<a :href="url">サポートサイト</a>
```

4.4.2 複数の属性をまとめて指定する

v-bindディレクティブには、「属性名: 値」形式のハッシュ（オブジェクト）を渡すことで、複数の属性をまとめてバインドすることもできます。たとえば以下は、<input>要素にsize、maxlength、required属性をまとめて設定する例です。

［リスト］BindObject.vue

```
<script setup>
import { reactive } from 'vue'
// 属性情報をオブジェクト形式で準備
const attrs = reactive({
  size: 12,
  maxlength: 15,
  required: true
})
</script>

<template>
  <form>
    <label for="message">メッセージ：</label>
    <input type="text" id="message" v-bind="attrs" />
  </form>
</template>
```

▼結果

```
<form>
  <label for="message">メッセージ：</label>
  <input type="text" id="message" size="12" maxlength="15" required>
</form>
```

属性情報をハッシュとして渡す場合には、v-bindディレクティブの引数（「v-bind:xxxxx」の「xxxxx」）は不要です。また、required属性のように値を持たない（＝固定である）属性を設定する場合には、値はtrueとしておきます（falseの場合は属性は付与されません）。

4.4.3 JavaScriptの式から属性名を決定する

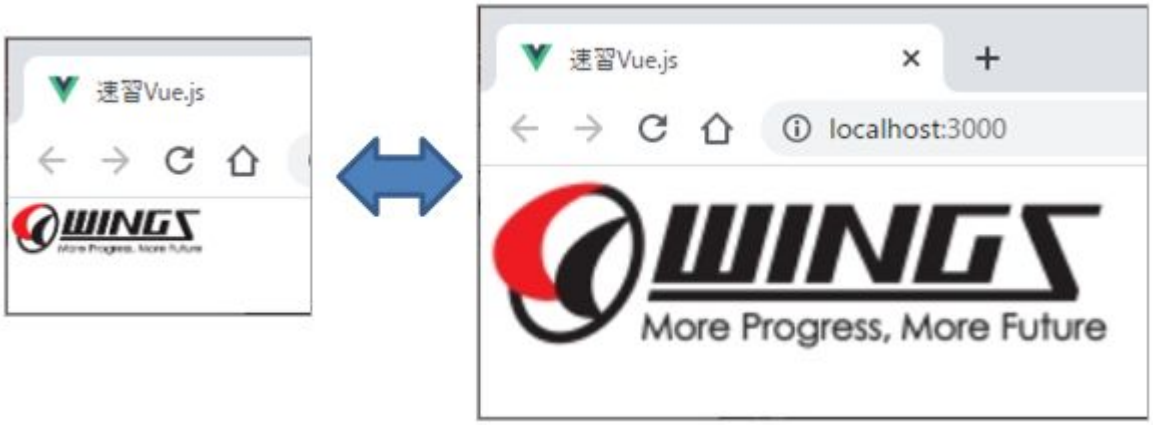
[...]で括ることで、属性名を式の値から動的に生成することも可能です（**動的引数**）。たとえば以下は`<img>`要素の属性を動的引数として設定する例です。

［リスト］BindDynamic.vue

```
<script setup>
import { ref } from 'vue'
// 属性名を設定
const attr = ref('width')
const value = ref(100)
</script>

<template>
  <img src="https://wings.msn.to/image/wings.jpg"
    v-bind:[attr]="value" />
</template>
```

この例であれば、属性名（attr）にwidthが、値（value）に100が設定されているので、「width="100"」と同じ意味です（＝画像幅が設定されます）。以下はその結果と、「ref('height')」と書き換えた場合の結果です。

図：（左）画像の幅を設定、（右）画像の高さを設定



　変数attrの値に応じて、異なる属性が設定されていることを確認してください。

4.5 値を一度だけバインドする - v-once

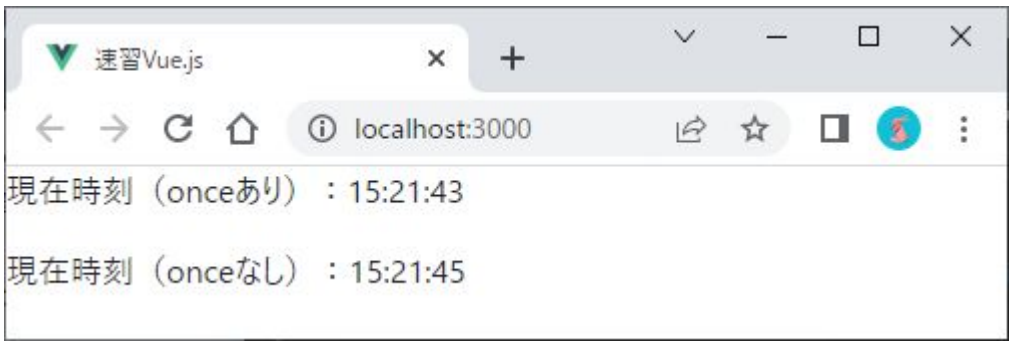
v-onceディレクティブを利用することで、配下のコンテンツを一度だけ描画します。コンテンツが初期値から変更されないことが判っているならば、v-onceを利用することで、ページの更新性能を最適化できます。

たとえば以下は、現在時刻currentをv-onceあり／なしの`<div>`要素にそれぞれ反映させる例です。currentは1000ミリ秒おきに更新されるものとします。

［リスト］OnceBasic.vue

```
<script setup>
import { ref } from 'vue'
const current =  ref(new Date().toLocaleTimeString())
// 1000ミリ秒おきにRef変数を更新
setInterval(() => {
  current.value = new Date().toLocaleTimeString()
}, 1000)
</script>

<template>
  <p v-once>現在時刻（onceあり）：{{ current }}</p>
  <p>現在時刻（onceなし）：{{ current }}</p>
</template>
```

図：current値を変更しても片方しか反映されない



サンプルを実行すると、確かに、v-once付きの 要素は初期値から変化しない（＝setIntervalによる更新を反映しない）ことが確認できます。

4.6 要素にスタイルプロパティを設定する - v-bind:style

v-bind:styleディレクティブを利用することで、インラインスタイルを設定できます。Vueでスタイルを設定する、最もシンプルな手段です。

［Note］v-bind:styleの意味

説明の便宜上、独立したディレクティブのように呼んでいますが、v-bind:styleとはv-bindディレクティブでstyle属性を設定しなさい、という意味です。ただし、その設定値は、他の属性を設定する場合と異なるので、本書でも別ものとして解説しています。

style属性へのバインドを、Vueでは**スタイルバインディング**と呼んでいます。

4.6.1 スタイルバインディングの基本

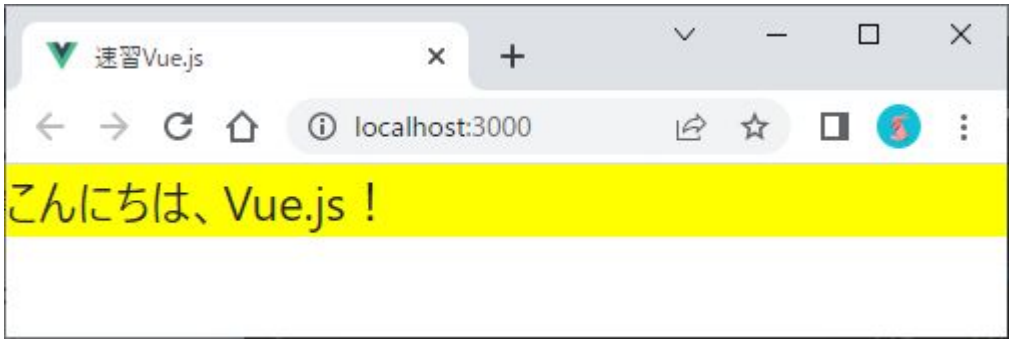
以下は、v-bind:styleディレクティブの具体的な例です。

［リスト］StyleBasic.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('こんにちは、Vue.js！')
</script>
```

```
<template>
  <div v-bind:style="{ backgroundColor: 'Yellow', fontSize: '1.5em' }">
    {{ message }}
  </div>
</template>
```

図：要素に指定のスタイルが適用された



スタイル情報は「スタイルプロパティ名: 値」のオブジェクト形式で指定します。ハイフン区切りの名前は、サンプルのようにキャメルケースで表記してください（ここではbackground-colorをbackgroundColorとします）。

［Note］ハイフン区切りの名前

以下のように名前の前後をクォートで括ることで、ハイフン区切りのままで表すことも可能です。

```
<div v-bind:style="{ 'background-color': 'Yellow', 'font-size': '1.5em' }">
```

4.6.2 複数のスタイル情報を適用する

v-bind:styleには、配列形式で複数のオブジェクトを渡すこともできます。この場合、オブジェクト（スタイル情報）は内部的に結合された状態で、要素に適用されます。オブジェクト間で重複したスタイルプロパティは、後者が優先されます。

［リスト］StyleMulti.vue

```
<script setup>
import { ref, reactive } from 'vue'
const message = ref('こんにちは、Vue.js！')
// スタイル情報を準備
const color = reactive({
  backgroundColor: 'Yellow',
  color: 'Red'
})
const big = reactive({
  fontSize: '1.5em'
})
</script>

<template>
  <div v-bind:style="[ color, big ]">
    {{ message }}
  </div>
</template>
```

▼結果

```
<div style="background-color: yellow; color: red; font-size: 1.5em;">
 こんにちは、Vue.js！
</div>
```

4.6.3 ベンダープレフィックスを自動補完する

v-bind:styleディレクティブでは、ブラウザーの対応状況に応じてベンダープレフィックスを補完する機能を備えています。ベンダープレフィックスは、往々にしてスタイル指定を冗長にしますが、この機能を利用することでスタイル指定がシンプルになります。

［リスト］StylePrefix.vue

```
<script setup>
import { ref } from 'vue'
const url = ref('https://wings.msn.to/')
</script>

<template>
  <a v-bind:style="{ 'tap-highlight-color': 'Yellow' }"
    v-bind:href="url">
    {{ url }}
  </a>
</template>
```

▼結果

```
<a href="https://wings.msn.to/"
  style="-webkit-tap-highlight-color: yellow;">
```

```
  https://wings.msn.to/
</a>
```

上はChromeでアクセスした場合の結果です。確かにChrome向けのベンダープレフィックス-webkitが付与されて、-webkit-tap-highlight-colorという名前が生成されていることが確認できます。

4.7 要素にスタイルクラスを設定する - v-bind:class

v-bind:styleによるスタイルの操作は手軽で便利ですが、問題もあります。というのも、JavaScriptコード（もしくはテンプレート）の中にスタイル情報が混在してしまうのです。スタイルを修正するために、スタイルとコードの双方を見なければならないのは、望ましい状態ではありません。

そこで、基本的にはv-bind:styleはあくまで手軽なスタイル操作の手段と割り切り、本格的なアプリではv-bind:classディレクティブを利用すべきです。v-bind:classは、あらかじめ用意されたスタイルクラスを要素に割り当てるためのディレクティブです。

4.7.1 クラスバインディングの基本

たとえば以下は、`<div>`要素に対してbig、color、frameクラスを割り当てる例です [1]。

［リスト］ClassBasic.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('こんにちは、Vue.js！')
const isChange = ref(true)
</script>

<template>
  <div class="big"
```

```
    v-bind:class="{ color: true, frame: isChange }">
    {{ message }}
  </div>
</template>
```

▼結果

```
<div class="big color frame">
  こんにちは、Vue.js！
</div>
```

v-bind:classディレクティブは「クラス名: true／false」形式のオブジェクトを受けとります。これで、値がtrueであるクラスだけを有効にする、というわけです。

また、v-bind:classは、静的なclass属性と併存できる点に注目してください。この場合、class属性とv-bind:classの結果がマージされた結果が描画されます。

4.7.2 v-bind:classのさまざまな設定方法

v-bind:classディレクティブには、オブジェクト形式で指定する他、以下のような設定方法があります。

（1） 文字列配列として渡す

スタイルクラス名を文字列の配列として渡すこともできます。

［リスト］ClassString.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('こんにちは、Vue.js！')
const colorClass = ref('color')
const frameClass = ref('frame')
</script>

<template>
  <div class="big"
    v-bind:class="[ colorClass, frameClass ]">
    {{ message }}
  </div>
</template>
```

▼結果

```
<div class="big color frame">
こんにちは、Vue.js！
</div>
```

（2）文字列／オブジェクトの配列として渡す

配列に、文字列（クラス名）と「クラス名: true／false」形式のオブジェクトを混在させることもできます。スタイルリストの一部だけが条件によってオンオフ変動する場合に利用できる構文です。

［リスト］ClassMulti.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('こんにちは、Vue.js！')
const colorClass = ref('color')
const isChange = ref(true)
</script>

<template>
  <div class="big"
    v-bind:class="[colorClass, { frame: isChange }]">
    {{ message }}
  </div>
</template>
```

▼結果

```
<div class="big color frame">
  こんにちは、Vue.js！
</div>
```

4.8 {{...}}式による画面のチラツキを防ぐ - v-cloak

{{ ... }}式では、要素配下にそのまま記述するという性質上、ページを起動した最初のタイミングで、一瞬だけ生の式表現が表示されてしまうという問題があります。これは、Vueが初期化処理を終え、{{ ... }}式を処理するまでの僅かなタイムラグによって生じる問題です。大概はごく一瞬の現象ですが、内部的なコードがエンドユーザーの目に触れるのは望ましい状態ではありません。

そこで利用するのが、v-cloakディレクティブです。

［リスト］CloakBasic.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('こんにちは、Vue.js！')
</script>

<template>
  <p v-cloak>{{ message }}</p>
</template>

<style scoped>
[v-cloak] {
  display: none;
}
</style>
```

v-cloakを利用する場合、まず、太字のようなスタイルシートでv-cloak付きの要素を非表示にします。

Vueは、初期化のタイミングでv-cloak属性（ディレクティブ）を見つけると、これを破棄します。これによって、初期化前には非表示だった要素が、初期化タイミングで初めて、表示状態になるというわけです。

---

1. 本来であれば、対応するスタイルクラスも定義しておくべきですが、本項ではそれが目的ではないので、割愛します。↵

Part 5：イベント処理

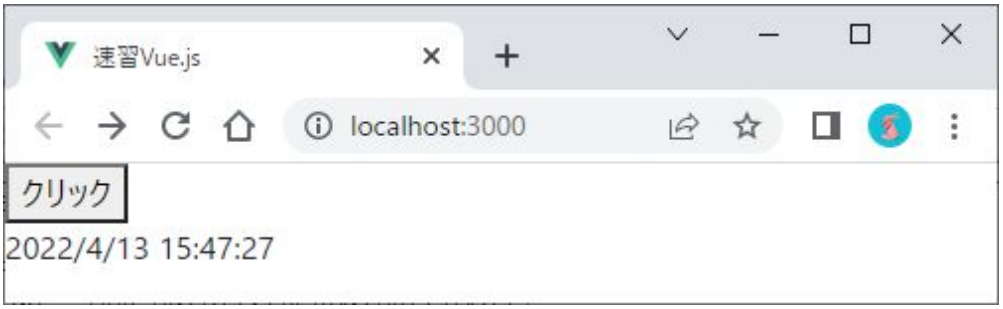
5.1 イベントの基本

Vueでは、イベントハンドラーもまたディレクティブを使って設定します。以下は、ボタンクリック時に現在時刻を表示するサンプルです。

［リスト］EventBasic.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('')
// クリック時に実行されるイベントハンドラー
const onclick = () => {
  message.value = new Date().toLocaleString()
}
</script>

<template>
  <!--a. イベントハンドラーを登録 -->
  <button v-on:click="onclick">クリック</button>
  <p>{{ message }}</p>
</template>
```

図：ボタンクリック時に現在時刻を表示



イベントハンドラーを設定するのは、v-onディレクティブの役割です（a.）。

> **［構文］v-onディレクティブ**
> v-on:イベント名="..."

ここでは「...」の部分にメソッドの名前を指定しており、まずはこれが基本と考えてください。ただし、別解として、以下のように表すこともできます。

```
<!--1. JavaScript式を直書き-->
<button v-on:click="message=new Date().toLocaleString()">クリック</button>
<!--2. メソッド呼び出し-->
<button v-on:click="onclick()">クリック</button>
```

1. は、v-onにJavaScriptの式を直書きするパターンです。手軽ですが、テンプレートにコードが混在するため、見通しが悪化します。テストコードや、ごくシンプルなコードでの利用に留め、原則的には独立したメソッドとして切り出すべきです。

2. は、元のEventBasic.vueと似ていますが、末尾に「()」が付いている点に注目です。つまり、元のEventBasic.vueがメソッドの名前を指定しているのに対して、2. はメソッド呼び出しの式であるということです。

2. の構文を利用することで、たとえば

```
v-on:click="onclick('Hoge')"
```

のように、イベントハンドラーになんらかの値を渡すことも可能になります（具体的な例は、あとで改めます）。

［Note］v-onの省略構文

v-bindと並んで、v-onはよく利用することから、省略構文が用意されています。先ほどのEventBasic.vueを省略構文で書き換えると、以下のようになります。

```
<button @click="onclick">クリック</button>
```

いずれを利用しても構いませんが、先のv-bindの時と同じく、アプリの中では記法を揃えることを強くお勧めします。本書では、初学者にも意図を汲み取りやすいよう、非省略構文を利用していきます。

5.2 イベントオブジェクトを参照する

イベントオブジェクトとは、その名の通り、イベントに関わる情報を管理するためのオブジェクトで、JavaScriptによって自動生成されます。

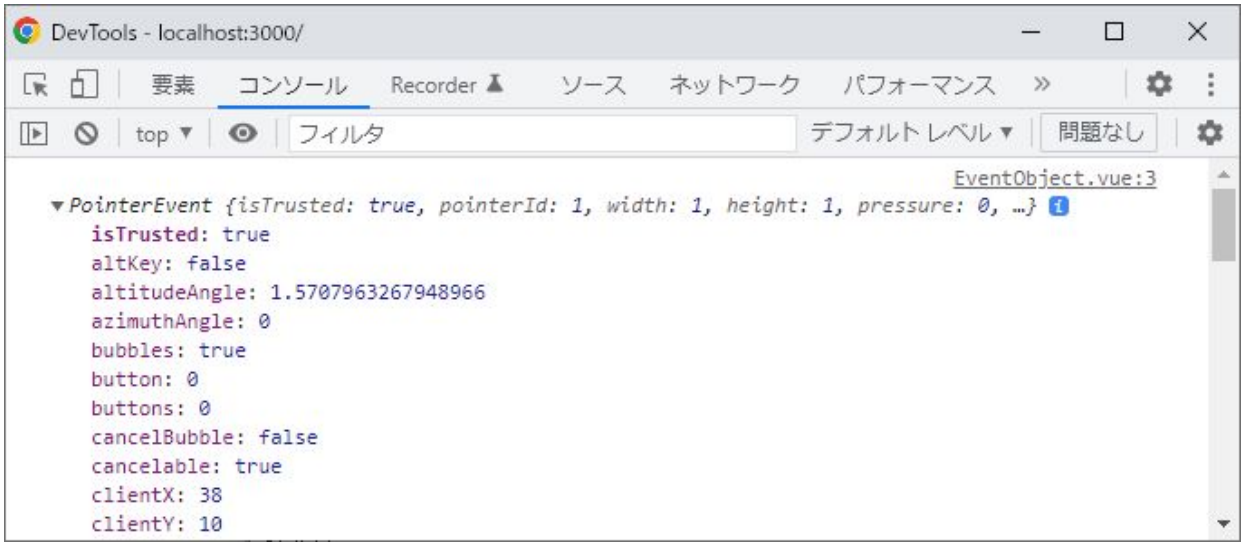
5.2.1 イベントオブジェクトの基本

イベントハンドラーでイベントオブジェクトを参照するには、イベントハンドラーの第1引数に「e」「ev」（名前はなんでも構いません）を設置しておくだけです。たとえば以下は、ボタンクリック時にイベントオブジェクトをログ出力する例です。

［リスト］EventObject.vue

```
<script setup>
const onclick = e => {
  console.log(e)
}
</script>

<template>
  <button v-on:click="onclick">クリック</button>
</template>
```

図：ログからイベントオブジェクトを確認



イベントオブジェクトの主なメンバーは、以下の通りです。

- target：イベント発生元の要素
- type：イベントの種類（click、focusなど）
- timeStamp：イベントの作成日時を取得
- clientX：イベントの発生時のブラウザー上でのX座標
- clientY：イベントの発発生時のブラウザー上でのY座標
- screenX：イベントの発生時のスクリーン上でのX座標
- screenY：イベントの発発生時のスクリーン上でのY座標
- pageX：イベントの発生時のページ上でのX座標
- pageY：イベントの発生時のページ上でのY座標
- offsetX：イベントの発生時の要素上でのX座標
- offsetY：イベントの発生時の要素上でのY座標

5.2.2 イベントハンドラーに引数を渡す場合

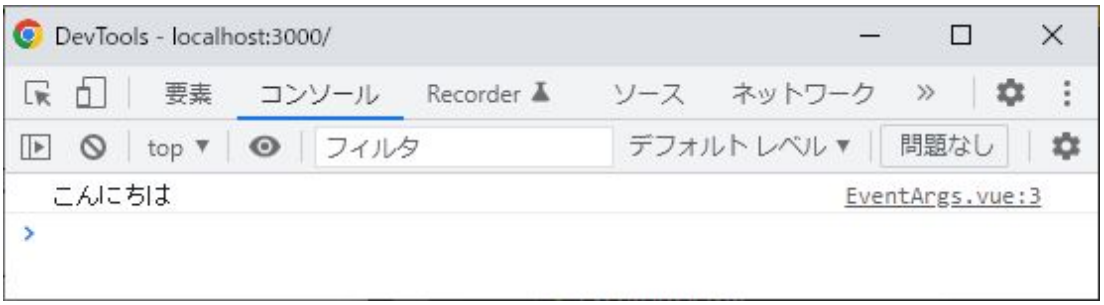
ここまでは標準的なJavaScriptのルールなので、迷うところはないはずです。しかし、イベントハンドラーになんらかの値を引き渡す場合には、どうでしょう。

たとえば以下は、クリック時に引数経由で渡した文字列をログ表示する例です。

［リスト］EventArgs.vue

```
<script setup>
const onclick = message => {
  console.log(message)
}
</script>

<template>
  <button v-on:click="onclick('こんにちは')">クリック</button>
</template>
```

図：引数経由で渡した値をログ表示



ただし、この状態では、第1引数が他の値で埋められてしまうので、イベントオブジェクトを参照することができません。

　このような場合には、呼び出し側で明示的に$event（イベントオブジェクト）を渡してください（$eventはVueで決められた名前です）。

［リスト］EventArgs2.vue

```
<script setup>
const onclick = (message, e) => {
  console.log(message)
  console.log(e)
}
</script>

<template>
  <button v-on:click="onclick('こんにちは', $event)">クリック</button>
</template>
```

　これで、第2引数以降でイベントオブジェクトを受け取り、イベントハンドラーの配下でも受け取れるようになります。

5.3 定型的なイベント処理を宣言的に指定する - イベント修飾子

イベントハンドラーを記述していると、たとえばイベント既定の動作をキャンセルしたい、イベントのバブリング（＝上位ノードへの伝播）を防ぎたいなど、ごく決まりきったコードが発生します。

もちろん、これらのコードはごくシンプルですが、定型的なものであれば、コードから追い出した方が、本来のロジックに集中でき、コードの見通しも改善します。そのような用途のためにVueで用意しているのが、**イベント修飾子**です。イベントに関わる定型的な処理を、属性の形式で表すための仕掛けです。

> **［構文］イベント修飾子**
> v-on:イベント名.修飾子="..."

利用できる主なイベント修飾子は、以下の通りです。

- .stop：イベントの親要素への伝播を中止（stopPropagationに相当）
- .prevent：イベント既定の動作をキャンセル（preventDefaultに相当）
- .capture：イベントハンドラーをキャプチャモードで動作
- .self：イベント発生元がその要素自身の場合にだけ実行
- .once：イベントハンドラーを一度だけ実行
- .passive：passiveモードを有効化（後述）

［Note］修飾子

修飾子は、ディレクティブの機能を制御するための指示の一種です。「.」（ドット）区切りで指定します。v-onだけでなく、他のディレクティブでも登場します。

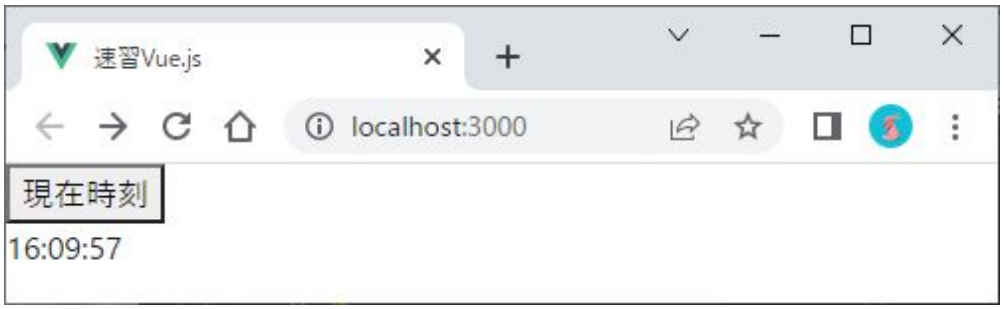
5.3.1 イベント修飾子の基本

イベント修飾子の具体的な例として、以下ではイベントハンドラーを一度だけ実行するコードを示します。［現在時刻］ボタンを押した初回は現在時刻が表示されるが、2度目以降のクリックでは時刻は更新**されない**ことを確認してください。

［リスト］EventOnce.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('')
// クリック時に実行されるイベントハンドラー
const onclick = () => {
  message.value = new Date().toLocaleTimeString()
}
</script>

<template>
  <button v-on:click.once="onclick">現在時刻</button>
  <p>{{ message }}</p>
</template>
```

図：初回に表示された時刻はボタンをクリックしても更新されない



5.3.2 イベント修飾子を利用する場合の注意点

その他、イベント修飾子を利用する場合の注意点をまとめておきます。

（1）複数の修飾子を連結できる

イベント修飾子は、たとえば「v-on:click.**once.prevent**」のように複数を連結することも可能です。ただし、修飾子は連結順に解釈される点に注意してください。

よって、（たとえば）「v-on:click.prevent.self」は要素配下のclickイベントすべてで既定の動作を無効化しますが、「v-on:click.self.prevent」は現在の要素のclickイベントについてのみ既定の動作を無効化する、という意味になります。

（2）ハンドラー本体を省略できる

イベント修飾子を指定した場合、イベントハンドラーの本体を省略することもできます。たとえば、フォームをサブミットした時のデータ送信をキャンセルするためのコードは、以下のように表せます（追加のコードが不要なのであれば、空のハンドラーを用意する必要はありません）。

```
<form v-on:submit.prevent>
```

（3）passive修飾子とprevent修飾子は同時に指定しない

passive修飾子は、イベントハンドラーがpreventDefaultメソッドを呼び出さ**ない**ことを宣言します。scrollイベントでpassive修飾子を有効にすることで、ブラウザー（特にモバイル環境）ではイベントハンドラーの完了を待たずにスクロールを開始できるので、パフォーマンスを改善できます。

その性質上、passive修飾子とprevent修飾子を同時に利用した場合、preventは無視され、ブラウザーからも警告されます。

5.4 キーボードからの入力を識別する - キー修飾子

keyup、keydown、keypressなどのキーイベントを利用する場合、押されたキーを識別してから処理を行うのが一般的です。そこでVueでは、いちいちハンドラー側でキー判別のコードを記述しなくても良いように、**キー修飾子**を用意しています。

> **［構文］キー修飾子**
> v-on:keyup.キー修飾子="..."

キー修飾子として利用できる値は、イベントオブジェクト（$event）のkeyプロパティが返す値をケバブケース記法に変換したものです。以下に、主な値をまとめます [1]。

- escape：［Esc］キー
- backspace：［Backspace］キー
- delete：［Delete］キー
- enter：［Enter］キー
- home：［Home］キー
- end：［End］キー
- page-up：［PageUp］キー
- page-down：［PageDown］キー
- arrow-up：［↑］キー
- arrow-down：［↓］キー

- arrow-left：［←］キー
- arrow-right：［→］キー

5.4.1 キー修飾子の基本

たとえば以下は、テキストボックスで［Esc］キーが押された場合に、テキストボックスの値をクリアする例です（v-modelについては、後述します）。

［リスト］EventKey.vue

```
<script setup>
import { ref } from 'vue'
const name = ref('ゲスト')
// テキストボックスの内容をクリア
const clear = () => {
  name.value = ''
}
</script>

<template>
  <form>
    <label for="name">氏名：</label>
          <input  type="text"  id="name"  v-on:keyup.escape="clear"  v-
model="name" />
  </form>
</template>
```

図：［Esc］キーでテキストボックスの内容をクリア

5.4.2 システムキーとの組み合わせを検知する

［Alt］［Ctrl］［Shift］などは、大概、他のキーとセットで利用するキーです（たとえば［Shift］＋［Space］のように）。これらの組み合わせを表現するには、

- .ctrl
- .alt
- .shift
- .meta

などの修飾子を利用してください。

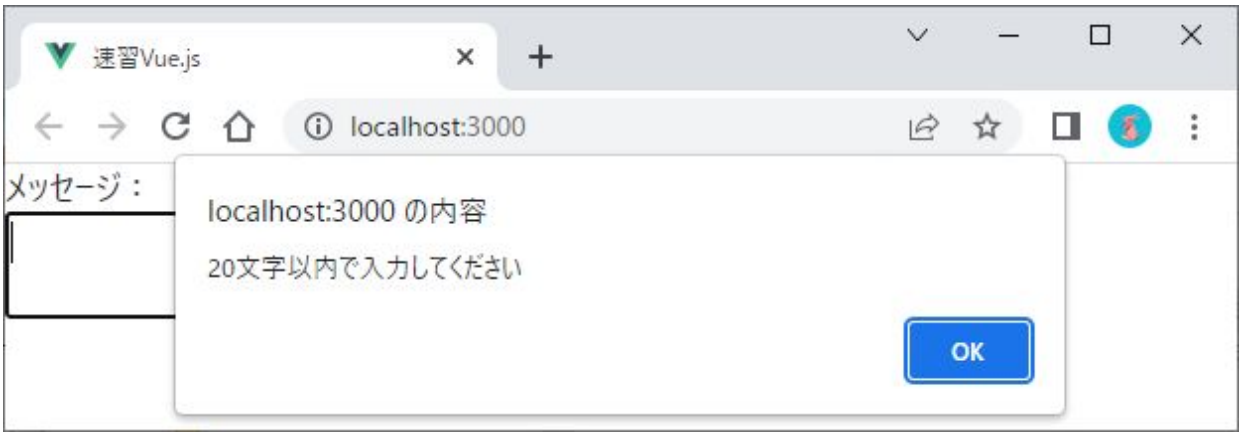
たとえば以下は、テキストエリア上で［Alt］＋［Enter］を押下することで、簡易ヘルプを表示する例です。

［リスト］EventSys.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('')
// 決められたキーの押下でヘルプを表示
```

```
const help = () => {
  window.alert('20文字以内で入力してください')
}
</script>

<template>
  <form>
    <label for="message">メッセージ：</label><br />
                <textarea  id="message"  v-on:keyup.alt.enter="help"  v-
model="message">
    </textarea>
  </form>
</template>
```

図：キー押下でヘルプダイアログを表示



ちなみに、keyup.alt.enterは最低限［Alt］＋［Enter］が押されていることを検知するもので、（たとえば）［Ctrl］＋［Alt］＋［Enter］のように余計なキーが押されていても、同じくイベントを検知します。もしも厳密に［Alt］＋［Enter］の組み合わせを検知したいならば、exact修飾子を利用してください。

```
<textarea id="message" v-on:keyup.alt.enter.exact="help" v-model="message"></textarea>
```

5.5 マウスの特定のボタンを検知する - マウス修飾子

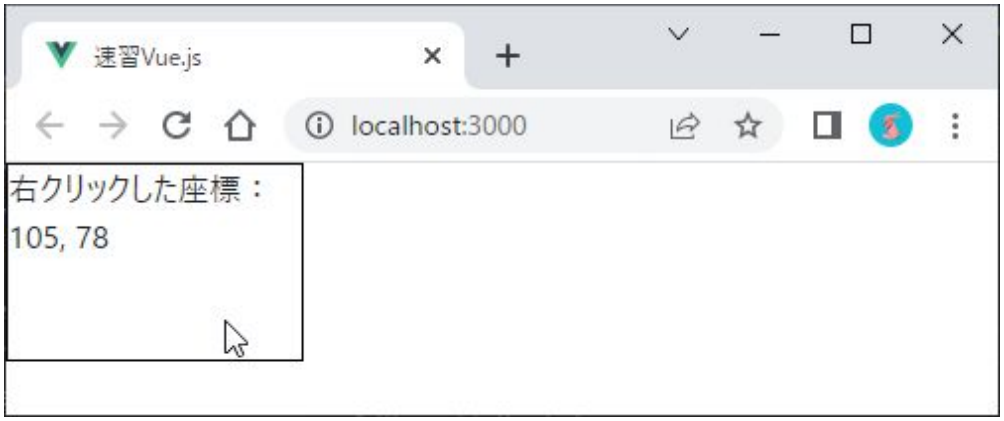
マウス修飾子を利用することで、マウスの特定のボタン（left、right、middle）に応じたハンドラーを設置することもできます。たとえば以下は、右ボタンをクリックした時に現在のマウス位置をダイアログ表示する例です。

［リスト］EventMouse.vue

```
<script setup>
import { ref } from 'vue'
const msg = ref('')
const onclick = e => {
  msg.value = `右クリックした座標：${e.clientX}, ${e.clientY}`
}
</script>

<template>
  <div id="main" v-on:click.right.prevent="onclick">
    {{ msg }}
  </div>
</template>

<style scoped>
 #main {
  border: 1px solid black;
  width: 150px;
  height: 100px;
```

```
}
</style>
```

図： 要素内で右クリックすると座標を表示



.prevent修飾子を付けているのは、右クリックでブラウザー標準のコンテキストメニューが表示されてしまうのを防ぐためです。

［Note］システムキーとの組み合わせも可能

マウスイベントでも、システムキー修飾子（.ctrl、.altなど）、exact修飾子は利用できます。たとえば以下は［Ctrl］キーを押しながらマウスボタンを右クリックした時に、イベントハンドラーが実行されます。

```
<div id="main" v-on:click.ctrl.exact.right.prevent="onclick">
```

---

1. 完全なリストは「Key Values」も参照してください。⏎

Part 6：フォーム開発

Vueでは、v-modelディレクティブを利用することで、いわゆる**双方向データバインディング**を実装できます。

これまでに説明したデータバインディングは、いずれもRef変数／Reactive変数⇒テンプレートの片方向データバインディングでした。一方、双方向データバインディングでは、Ref変数／Reactive変数⇒テンプレートのデータ反映はもちろん、テンプレート（一般的にはテキストボックスなどの入力）⇒Ref変数／Reactive変数のデータ反映を可能にします。

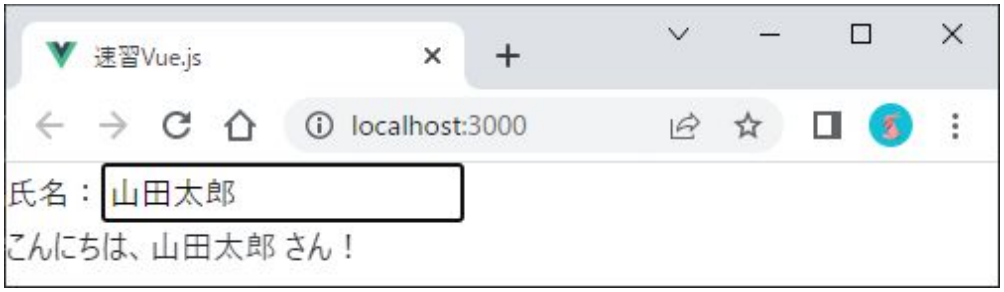
6.1 フォーム開発の基本

たとえば以下は、テキストボックスに入力された名前に応じて、「こんにちは、●○さん！」という挨拶メッセージを生成します。

［リスト］ModelBasic.vue

```
<script setup>
import { ref } from 'vue'
const myName = ref('ゲスト')
</script>

<template>
  <form>
    <label for="name">氏名：</label>
    <input type="text" id="name" v-model="myName" />
  </form>
  <div>こんにちは、{{ myName }} さん！</div>
</template>
```

図：入力された名前に応じて、メッセージが変化



「v-model="myName"」で、テキストボックスとRef変数myNameを紐づけているわけです。果たして、Ref変数の値がテキストボックスに反映されること、テキストボックスへの入力がRef変数に反映され、結果、「こんにちは、{{ myName }}さん！」にも合わせて反映されることを確認しておきましょう。

［Note］value属性は使わない

v-modelを利用した場合、テキストボックスの初期値は紐づいたRef変数の値となります。value属性を指定した場合には、「Unnecessary value binding used alongside v-model. It will interfere with v-model's behavior」（value属性がv-modelの挙動を邪魔している）のようなエラーとなるので注意してください。

これは、ラジオボタン／チェックボックスのchecked属性、選択ボックス／リストボックスのselected属性も同様です。

6.2 さまざまなフォーム要素の例

v-modelを利用することで、さまざまなフォーム要素の値を受けとれます。

6.2.1 ラジオボタン

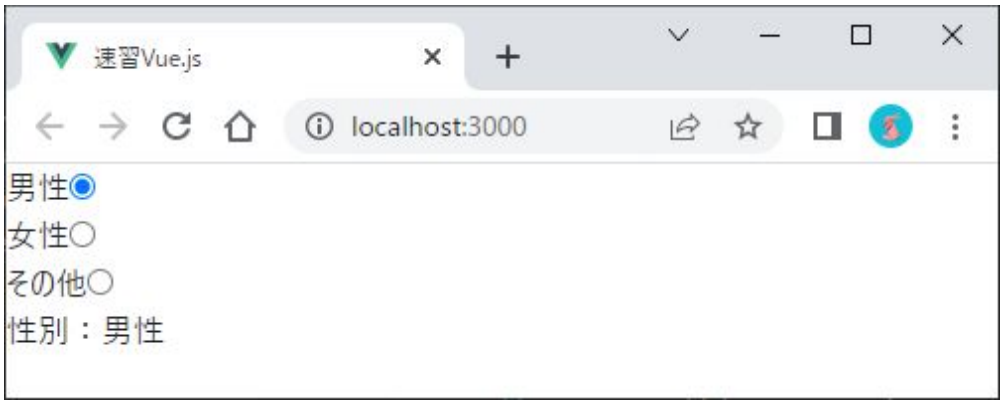
ラジオボタンでは、すべての選択オプションに対して、同一のv-modelを渡すのがポイントです。

［リスト］ModelRadio.vue

```
<script setup>
import { ref } from 'vue'
const gender = ref('男性')
</script>

<template>
  <form>
    <label for="man">男性</label>
    <input type="radio" id="man" value="男性" v-model="gender" />
    <br />
    <label for="woman">女性</label>
    <input type="radio" id="woman" value="女性" v-model="gender" />
    <br />
    <label for="other">その他</label>
    <input type="radio" id="other" value="その他" v-model="gender" />
  </form>
```

```
  <p> 性別：{{ gender }}</p>
</template>
```

図：ラジオボタンの選択値を表示



6.2.2 チェックボックス（単一）

チェックボックスは、単一でオンオフを表す場合と、リストで複数選択オプションを表す場合とがあります。

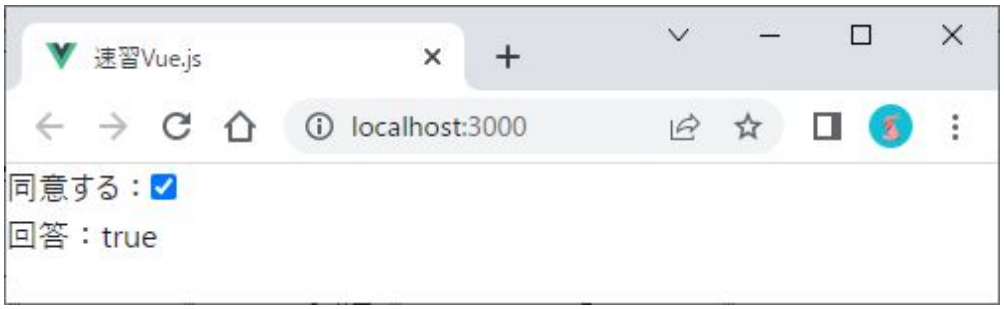
まずは、オンオフボタンを表す場合です。

［リスト］ModelCheck.vue

```
<script setup>
import { ref } from 'vue'
const agree = ref(true)
</script>

<template>
  <form>
    <label for="agree">同意する：</label>
    <input type="checkbox" id="agree" v-model="agree" />
```

```
  </form>
  <div>回答：{{ agree }} </div>
</template>
```

図：チェックボックスのオンオフ状態を表示



チェックボックスの値は、既定でbool値（true／false）で表されます。もしも特定の値で置き換えたい場合には、true-value／false-value属性を利用してください。

```
<input type="checkbox" id="agree" v-model="agree"
 true-value="○" false-value="×" />
```

6.2.3 チェックボックス（複数）

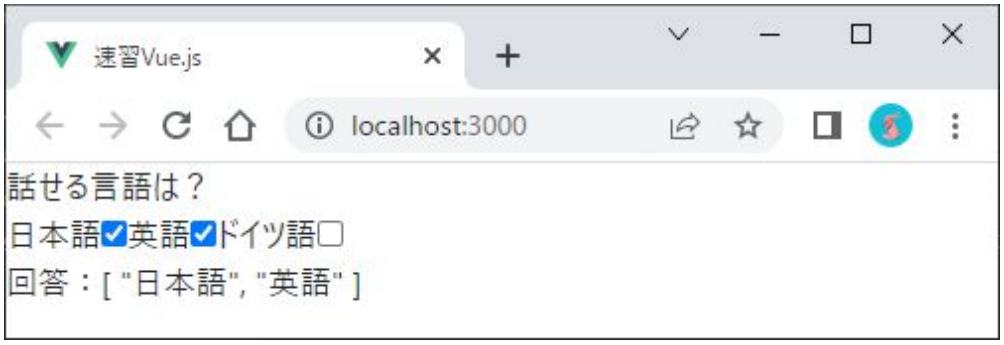
複数のチェックボックスを並べる場合には、（ラジオボタンの時と同じく）これらすべてに対して同一のv-modelを渡します。

［リスト］ModelCheckMulti.vue

```
<script setup>
import { ref } from 'vue'
```

```
const language = ref(['日本語', '英語'])
</script>

<template>
  <form>
    <div>話せる言語は？ </div>
    <label for="japanese">日本語</label>
          <input type="checkbox" id="japanese" value="日本語" v-model="language" />
    <label for="english">英語</label>
     <input type="checkbox" id="english" value="英語" v-model="language" />
    <label for="german">ドイツ語</label>
          <input type="checkbox" id="german" value="ドイツ語" v-model="language" />
  </form>
  <p> 回答：{{ language }}</p>
</template>
```

図：複数のチェックボックスの状態を表示



チェックボックスが複数選択された場合には、プロパティにも複数の値が配列として格納されます。

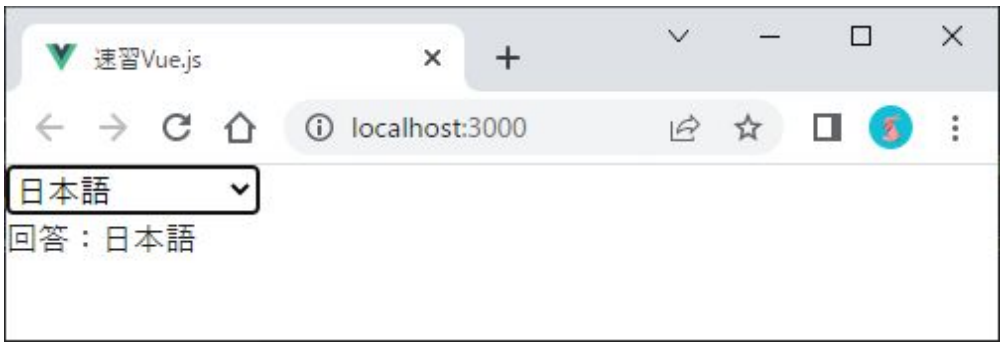
6.2.4 選択ボックス

選択ボックスはほぼ特筆すべき点はありません。`<select>`要素にv-modelを指定するだけです。

［リスト］ModelSelect.vue

```
<script setup>
import { ref } from 'vue'
const language = ref('')
</script>

<template>
  <form>
    <select v-model="language">
      <option value="">話せる言語は？</option>
      <option>日本語</option>
      <option>英語</option>
      <option>ドイツ語</option>
    </select>
  </form>
  <p> 回答：{{ language }}</p>
</template>
```

図：選択ボックスの選択値を表示

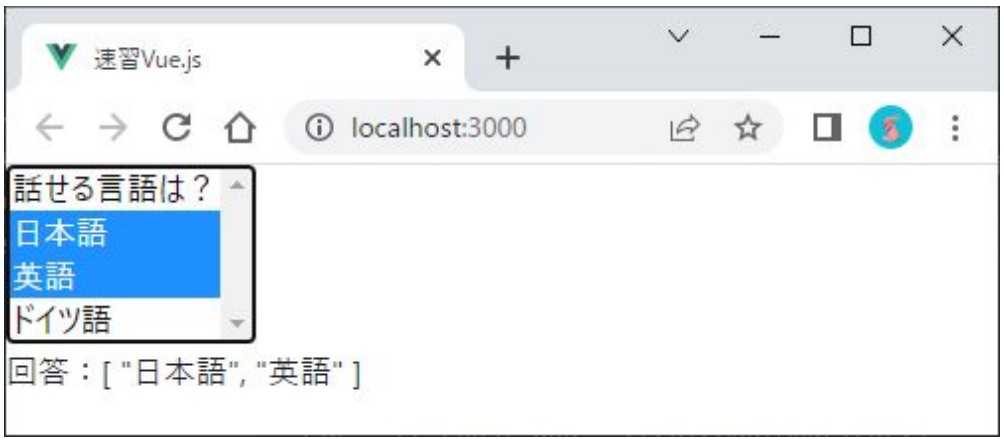


　複数選択を可能にした場合に、プロパティには配列が格納される点はチェックボックスと同じです。

［リスト］ModelSelectMulti.vue

```
<script setup>
import { ref } from 'vue'
const language = ref(['日本語' ,'英語'])
</script>

<template>
  <form>
    <select v-model="language" multiple size="4">
      <option value="">話せる言語は？ </option>
      <option>日本語</option>
      <option>英語</option>
      <option>ドイツ語</option>
    </select>
  </form>
  <p> 回答：{{ language }}</p>
</template>
```

図：複数の選択値を反映



6.2.5 補足：オブジェクトをバインドする

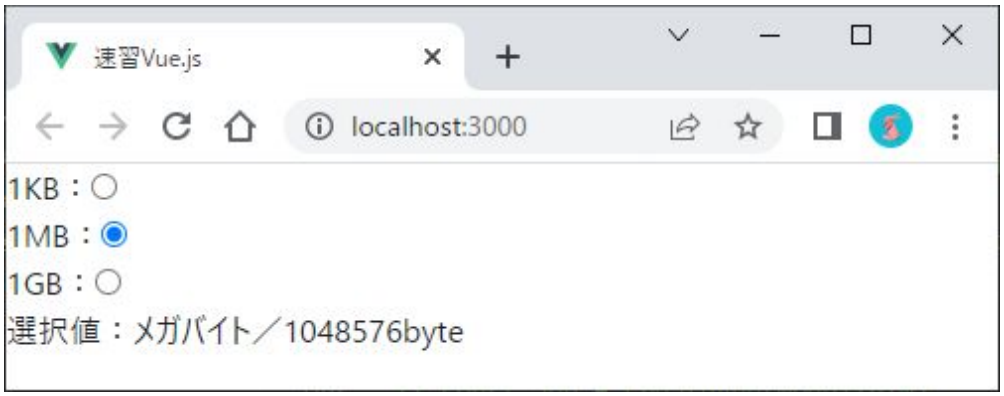
ラジオボタン／チェックボックス、選択ボックスなどには（文字列だけでなく）オブジェクトを引き渡すこともできます。

［リスト］ModelObject.vue

```
<script setup>
import { ref } from 'vue'
const unit = ref({})
</script>

<template>
  <form>
    <label for="kb">1KB：</label>
    <input type="radio" id="kb" v-model="unit"
      v-bind:value="{ name: 'キロバイト', size: 1024 }" /><br />
    <label for="mb">1MB：</label>
    <input type="radio" id="mb" v-model="unit"
      v-bind:value="{ name: 'メガバイト', size: 1048576 }" /><br />
    <label for="gb">1GB：</label>
```

```
    <input type="radio" id="gb" v-model="unit"
      v-bind:value="{ name: 'ギガバイト', size: 1073742000 }" />
  </form>
  <!--a. バインドされたオブジェクトの中身を表示 -->
  <div>選択値：{{ unit.name }}／{{ unit.size }}byte</div>
</template>
```

図：選択されたラジオボタンの値を表示



確かに、value属性にバインドしたオブジェクトがunitプロパティにも反映されて、そのname／sizeプロパティを読み出せていることが確認できます（a.）。選択ボックスであれば、オブジェクトは`<option>`要素のvalue属性にバインドします。

6.2.6 ファイル入力ボックス

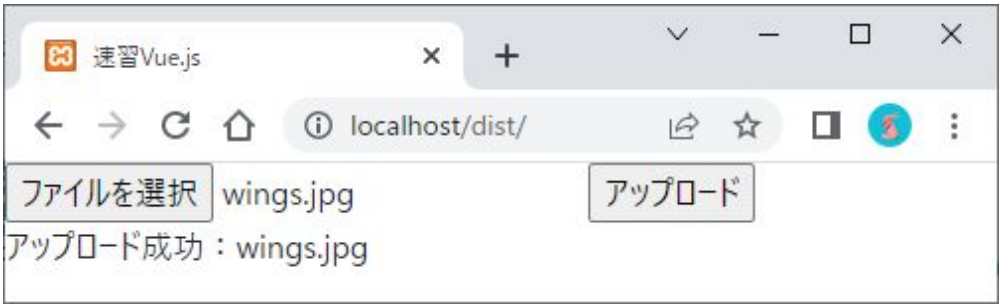
他のフォーム要素と異なり、ファイル入力ボックスにJavaScript側から値を設定することはできません。つまり、`<input type="file">`要素では双方向データバインディングという概念もなく、click／changeなどのイベントを受けて処理を実行します。

［リスト］ModelUpload.vue

```
<script setup>
import { ref } from 'vue'
const result = ref('')
// b. ref属性を取得
const upfile = ref(null)
const onclick = () => {
  // c. アップロードファイルを取得
  const file = upfile.value.files[0]
  // d.フォーム送信のための形式に変換
  const form = new FormData()
  form.append('upfile', file, file.name)
  // e. ファイルを送信
  fetch('upload.php', {
    method: 'POST',
    body: form
  })
  // f. 成功時にレスポンスデータを取得
  .then(function(response) {
    return response.text()
  })
  // g. ページに結果を表示
  .then(function(text) {
    result.value = text
  })
}
</script>
```

```
<template>
  <form>
    <!--a. ファイル入力ボックスを準備 -->
    <input type="file" ref="upfile" />
    <input type="button" value="アップロード" v-on:click="onclick" />
  </form>
  <div>{{ result }}</div>
</template>
```

図：指定のファイルをアップロード



`<input type="file">`要素のref属性に注目です（a.）。これはVueで用意された特殊な属性で、あとからイベントハンドラー側で参照できるよう、要素に名前付けしておくためのものです。

ref属性で指定した名前は、`<script>`要素の配下でも同名のRef変数として宣言しておきます（b.）。ref属性と紐づくので、値そのものはnullで構いません。

ref属性によって、イベントハンドラー（onclickメソッド）では、おなじみの「名前.value」の形式で要素を取得できるようになります（c.）。`<input>`要素を取

得できたら、そのfilesプロパティを参照することで、指定されたファイル群（FileListオブジェクト）を取得できます。[0]としているのは、FileListから先頭のファイル（Fileオブジェクト）を決め打ちで取り出すという意味です。

ただし、Fileオブジェクトのままではアップロードできないので、通信のための形式（FormDataオブジェクト）に中身を詰め替えておきます。FormDataは、multipart/form-data形式のフォームデータをキー／値の形式で表すためのオブジェクトです。

FormDataオブジェクトにデータを登録するのは、appendメソッドの役割です（**d.**）。

> **［構文］appendメソッド**
> append(*name*, *value*, *file*)
> - name：キー名
> - value：ファイル本体
> - file：ファイル名

あとは、fetchメソッドで指定のファイルを送信するだけです（**e.**）。

> **［構文］fetchメソッド**
> fetch(*url*, *opts*)
> - url：送信先のパス
> - opts：リクエストオプション

引数optsには「オプション名: 値, ...」の形式で、リクエスト時のオプションを指定できます。利用できる主なオプションは、以下の通りです。

- method：リクエストメソッド（GET、POSTなど。既定はGET）
- body：リクエスト本体
- headers：リクエストヘッダー（「ヘッダー名: 値, ...」形式）

ファイルをアップロードする場合はmethodオプションはPOST、先ほど作成したFormDataオブジェクトはbodyオプションに渡します。

fetchメソッドは、通信の結果をPromise<Response>（Responseオブジェクトを含んだPromiseオブジェクト）として返します。Promiseはネットワーク通信のような非同期処理を管理するためのオブジェクトです。Promise#thenメソッド（f.）を利用することで、処理に成功した場合の処理を記述できます。

この例であれば、Response#textメソッドで応答データを平のテキストとして取得しています。取得したテキストは、更にg.でページにも反映させています。

［Note］アップロード先の処理について

アップロード先（upload.php）についての詳細は、本書の守備範囲を外れるため、解説は割愛します。詳細は、拙著「独習PHP 第4版」（翔泳社）などの専門書を参照してください。

ここでは、送信されたアップロードファイルを/docフォルダーに保存し、「アップロード成功: *ファイル名*」のような結果文字列を返す、とだけ理解してお

けば十分です。/docフォルダー、upload.phpは、いずれもプロジェクトルート配下の/publicフォルダーに用意しています。

6.3 バインドの動作オプションを設定する - 修飾子

v-model属性にはさまざまな修飾子が用意されており、データバインディング時の挙動を制御できるようになっています。修飾子は「v-model.**number.lazy**」のように複数連結しても構いません（イベント修飾子の場合と同じです）。

6.3.1 入力値を数値としてバインドする - number修飾子

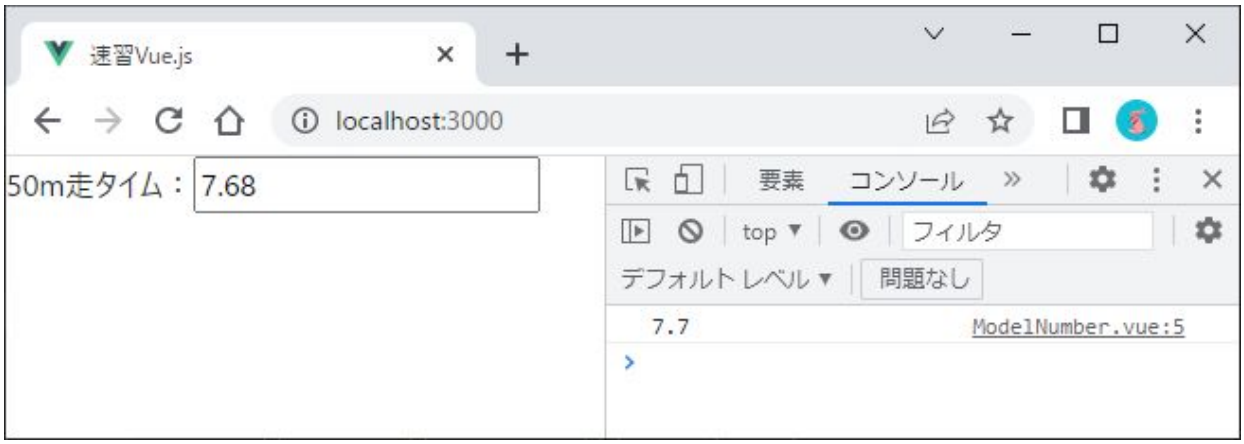
number修飾子を利用することで、テキストボックスへの入力値を（文字列ではなく）数値としてプロパティにバインドします。ユーザーの入力値は、既定で文字列と見なされますが（それはtype="number"からの入力でも同じです）、number修飾子を利用することで、コード側での数値変換が不要になります。

［リスト］ModelNumber.vue

```
<script setup>
import { ref } from 'vue'
const time = ref(0)
// 入力値を小数点第1位に丸めたものをログ表示
const onchange = () => {
  console.log(time.value.toFixed(1))
}
</script>

<template>
  <form>
    <label for="time">50m走タイム：</label>
```

```
    <input type="text" id="time" v-model.number="time"
      v-on:change="onchange" />
  </form>
</template>
```

図：入力された値を小数点以下1位に丸めたものをログ表示



確かに入力された値が数値として扱えている（＝Numberオブジェクトの ToFixedメソッドを呼び出せている）ことを確認してください。number修飾子がない場合、ToFixedメソッドの呼び出しはエラーとなります。

6.3.2 入力値の前後の空白を除去する - trim修飾子

trim修飾子を利用することで、入力値をプロパティにバインドする前に、前後の空白を除去できます。

以下は、入力された文字列から空白を除去した上で、ログに出力する例です。

［リスト］ModelTrim.vue

```
<script setup>
import { ref } from 'vue'
const message = ref('')
// 入力値の前後から空白を除いた上でログ出力
const onchange = () => {
  console.log(`入力値は「${message.value}」です。`)
}
</script>

<template>
  <form>
    <label for="message">メッセージ：</label>
    <input type="text" id="message" v-model.trim="message"
      v-on:change="onchange" />
  </form>
</template>
```

図：入力値から空白が除去されている

6.3.3 バインドのタイミングを遅延させる - lazy修飾子

v-modelによるバインドのタイミングはinputイベントの発生時です。つまり、キー入力のタイミングで即座にバインドします [1]。これをchangeイベント――変更後、フォーム要素からフォーカスが移動したタイミングでバインドさせるのが、lazy修飾子の役割です。

［リスト］ModelLazy.vue

```
<script setup>
import { ref } from 'vue'
const myName = ref('ゲスト')
</script>

<template>
  <form>
    <label for="name">氏名：</label>
    <input type="text" id="name" v-model.lazy="myName" />
  </form>
  <div>こんにちは、{{ myName }} さん！ </div>
</template>
```

図：フォーカスが外れたタイミングで入力値が変化

6.4 双方向データバインディングのカスタマイズ

v-modelによる双方向データバインディングは、内部的には、v-bind／v-on:changeのシンタックスシュガーです。よって、以下のコードは、いずれも意味的に等価です。

```
<input v-model="message" />
<input v-bind:value="message" v-on:input="message=$event.target.value" />
```

inputイベントで、入力値（$event.target.value）をmessageプロパティにバインドすると共に、value属性にmessageプロパティをバインドしているわけです。もちろん、一般的にはv-modelでまとめて表すのがシンプルですが、v-modelでは事足りない場合があります。

入力された値をプロパティにバインドする際になんらかの処理を挟みたい場合です。そのような場合には、v-bind:value／v-on:inputの組み合わせを利用します。

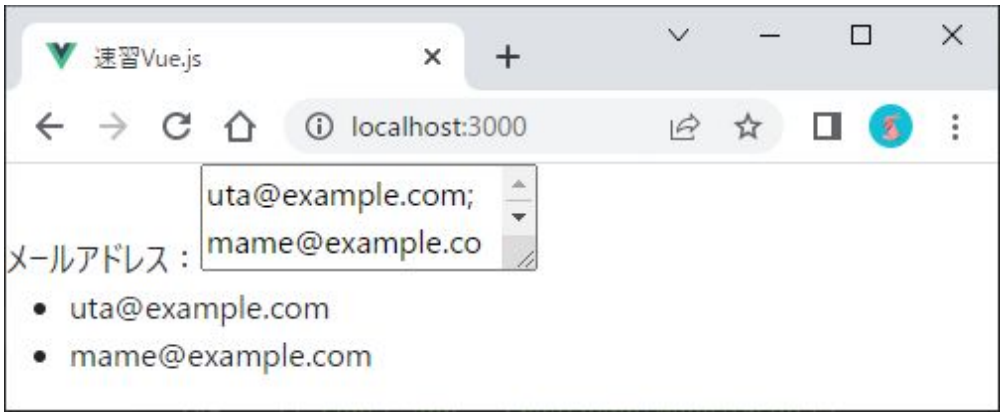
たとえば以下は、入力されたメールアドレス（セミコロン区切り）を分割し、配列としてRef変数emailsに反映させる例です（本来であれば、算出プロパティなどを利用すべきですが、簡単化のためにテンプレート内でコードを記述しています）。

v-forによる配列の展開については、この後改めて解説します。まずは配列をもとに<ul>／<li>リストを作成している、とだけ理解しておきましょう。

［リスト］ModelCustom.vue

```
<script setup>
import { ref } from 'vue'
const emails = ref([])
</script>

<template>
  <form>
    <label for="name">メールアドレス： </label>
    <!--Ref変数emailsの値をセミコロン区切りで反映
        & 入力値はセミコロンで分割した結果をRef変数に反映-->
    <textarea v-bind:value="emails.join(';')"
      v-on:input="emails=$event.target.value.split(';')"></textarea>
  </form>
  <!--配列emailsの内容を列挙-->
  <ul>
    <li v-for="email in emails" v-bind:key="email">
      {{ email }}
    </li>
  </ul>
</template>
```

図：入力されたメールアドレスをリスト表示



---

1. ただし、trim修飾子を利用した場合にはフォーカスを移動したタイミングでバインドが実行されます。↵

Part 7：条件分岐とループ

本節では、条件分岐／ループ（繰り返し）に関するディレクティブについて解説します。

7.1 式の真偽に応じて表示／非表示を切り替える - v-if

v-ifは、JavaScriptのif命令に相当するディレクティブです。指定された条件式がtrueの場合にだけ、現在の要素を出力します。

7.1.1 v-ifの基本

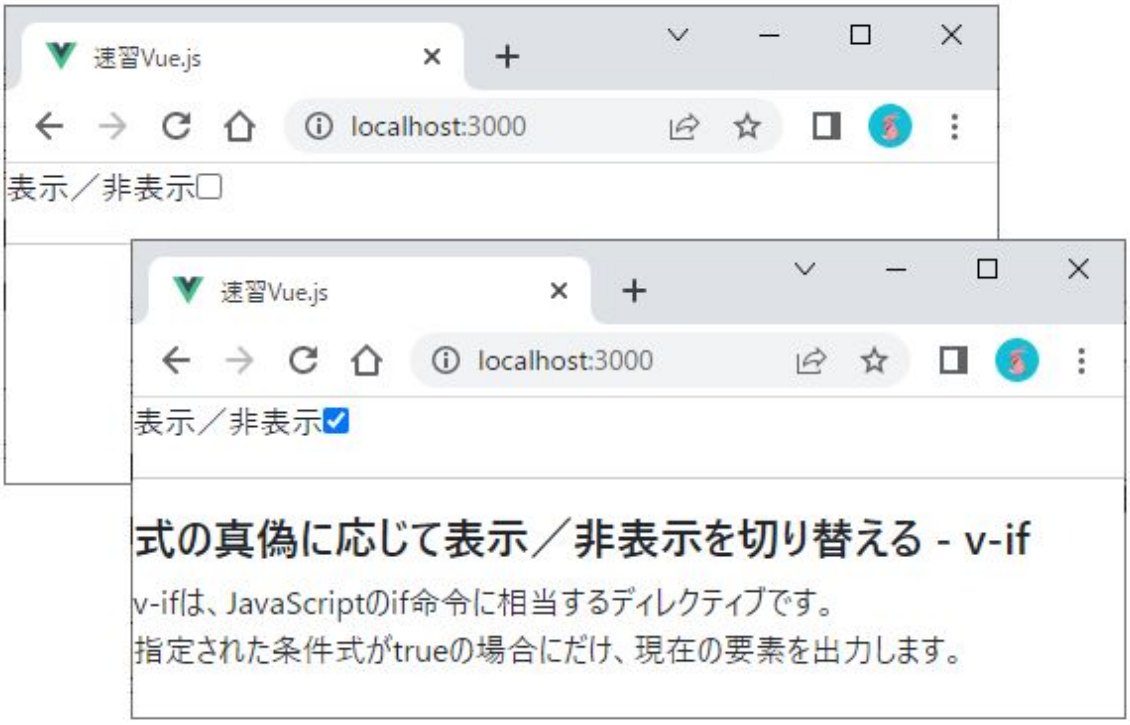
たとえば以下は、チェックボックスのオンオフに対して、`<div id="panel">`要素の表示／非表示を切り替える例です。

［リスト］IfBasic.vue

```
<script setup>
import { ref } from 'vue'
const toggle = ref(true)
</script>

<template>
  <form>
    <label for="show">表示／非表示</label>
    <input type="checkbox" id="show" v-model="toggle" />
  </form>
  <hr />
  <div id="panel" v-if="toggle">
    <h4>式の真偽に応じて表示／非表示を切り替える - v-if</h4>
    <p>v-ifは、JavaScriptのif命令に相当するディレクティブです。<br />
    指定された条件式がtrueの場合にだけ、現在の要素を出力します。</p>
```

```
  </div>
</template>
```

図：チェックボックスのオンオフでパネルの表示を切り替え



v-ifにはtrue／false値として評価できる式を指定します。この例では、Ref変数toggleを渡しています。Ref変数toggleはチェックボックスに紐づいているので、結果として、チェックボックスのオンオフに応じてパネルの表示／非表示も切り替わるというわけです。

7.1.2 式がfalseの場合の表示を定義する - v-else

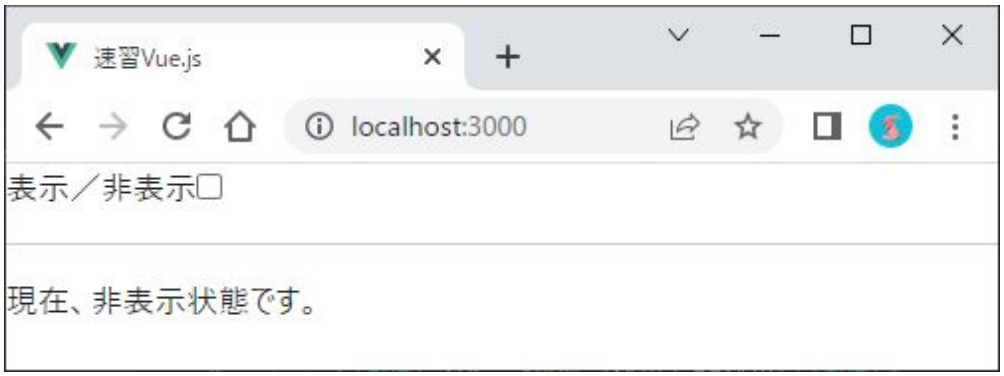
条件式がtrueの時だけでなく、falseの場合にもなんらかのコンテンツを表示したい場合には、v-elseディレクティブを利用します。

たとえば以下は、先のIfBasic.vueを修正して、チェックボックスをオフにした場合は「現在、非表示状態です。」というメッセージを表示する例です。

［リスト］IfBasic.vue

```
<template>
  …中略…
  <div id="panel" v-if="toggle">
    …中略…
  </div>
  <div v-else>現在、非表示状態です。</div>
</template>
```

図：チェックボックスをオフにすると、非表示メッセージを表示



v-elseは、v-if（または後述するv-else-if）の直後に記述しなければなりません。

7.1.3 複数の分岐を表現する - v-else-if

if…else ifに相当するディレクティブもあります。v-else-ifディレクティブです。v-else-ifはv-ifの直後に複数列記でき、いわゆる多岐分岐を表すために利用でき

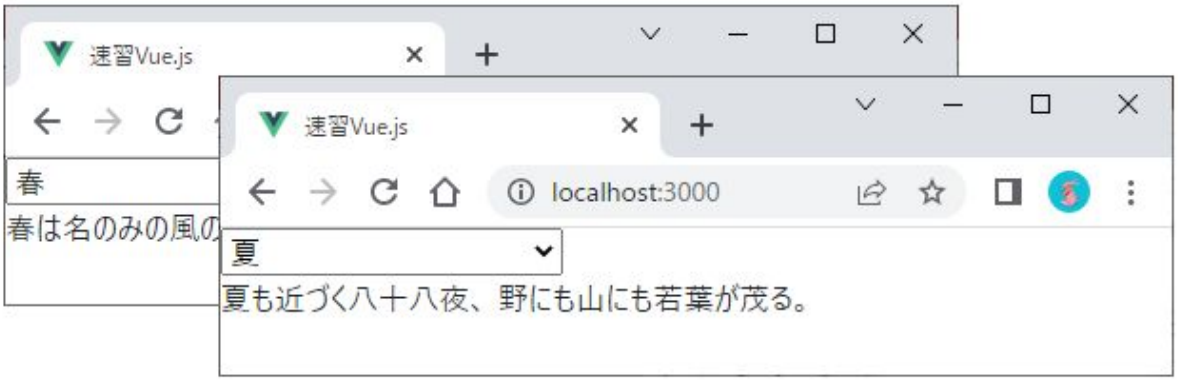
ます。

　たとえば以下は、選択ボックスでの選択に応じて、パネルの表示を切り替える例です。

［リスト］IfElse.vue

```
<script setup>
import { ref } from 'vue'
const season = ref('')
</script>

<template>
  <form>
    <select v-model="season">
      <option value="">季節を選択してください。</option>
      <option value="spring">春</option>
      <option value="summer">夏</option>
      <option value="autumn">秋</option>
      <option value="winter">冬</option>
    </select>
  </form>
  <div v-if="season==='spring'">春は名のみの風の寒さや…</div>
  <div v-else-if="season==='summer'">夏も近づく八十八夜…</div>
  <div v-else-if="season==='autumn'">秋の夕日に照る山もみじ…</div>
  <div v-else-if="season==='winter'">さ霧消ゆる湊江の舟に…</div>
  <div v-else>なにも選択されていません。</div>
</template>
```

図：選択ボックスに応じてメッセージを切り替え



7.2 式の真偽に応じて表示／非表示を切り替える - v-show

与えられた条件式がtrueの場合にだけ、現在の要素を出力するためのディレクティブとして、v-showディレクティブがあります。一見して、v-ifと同じに見えますが、（もちろん）双方は異なるものです。

まずは、v-ifのサンプル（IfBasic.vue）を実行し、Chrome付属のデベロッパーツール（［要素］タブ）から文書ツリーの変化を確認してみましょう。

図：パネルが表示状態の時（上）／非表示の時（下）

v-ifの世界では、パネルが非表示になった時、要素そのものが文書ツリーから破棄されていることが確認できます。逆に言えば、v-ifは条件式がtrueになるまで要素を出力しない、ということです（文書ツリーに非表示コンテンツを残しておくことは、リソース効率という意味でも望ましい状態ではありません）。これを**遅延描画**と言います。

その性質上、v-ifは頻繁に表示／非表示を切り替える場合に、描画コストが高まるおそれがあります。そのような状況では、v-showを利用してください。以下は、先ほどのIfBasic.vueをv-showで書き換えた例です。

［リスト］IfBasic.vue

```
<div id="panel" v-show="toggle">
…中略…
</div>
```

この状態でサンプルを実行し、同じくデベロッパーツールを確認してみましょう。

図：パネルが表示状態の時（上）／非表示の時（下）

今度は要素そのものは常に文書ツリーに組み込まれた状態で、スタイルシート（displayプロパティ）によってのみ表示／非表示が切り替わっていることが確

認できます。

以上から、一般的には、以下の基準でv-show／v-ifを使い分けてください。

- 表示／非表示を頻繁に切り替えるコンテンツにはv-show
- 最初に表示（非表示）にしたら滅多に変更しないものはv-if

7.3 配列／オブジェクトを繰り返し処理する - v-for

v-forディレクティブは、指定された配列／オブジェクトから順に要素を取り出し、その内容をループ処理します。

さまざまな構文があるので、具体的な例と共に用法を見ていきましょう。

7.3.1 配列から要素を順に取得する

たとえば以下は、あらかじめ用意された書籍情報（オブジェクト配列）から書籍リストを生成するサンプルです。

［リスト］ForBasic.vue

```
<script setup>
import { ref } from 'vue'
// オブジェクト配列を準備
const books = ref([
  {
    isbn: '978-4-7981-5382-7',
    title: '独習C# 新版',
    price: 3600
  },
  …中略…
])
</script>
```

```
<template>
 <table class="table">
  <th>ISBN</th><th>書名</th><th>価格</th>
  <tr v-for="b in books" v-bind:key="b.isbn">
   <td>{{ b.isbn }}</td>
   <td>{{ b.title }}</td>
   <td>{{ b.price }}円</td>
  </tr>
 </table>
</template>
```

図：オブジェクト配列booksをもとにリストを生成

ISBN	書名	価格
978-4-7981-5382-7	独習C# 新版	3600円
978-4-7741-9618-3	Ruby on Rails 超入門	2060円
978-4-7981-5202-8	Androidアプリ開発の教科書	2750円
978-4-7741-9572-8	3ステップでしっかり学ぶ MySQL入門	2480円
978-4-8222-5355-4	アプリを作ろう！Visual C#入門	2000円

v-forディレクティブの構文は、以下の通りです。

［構文］v-forディレクティブ

<element v-for="*item* in *list*" v-bind:key="*key*">...</*element*>

- element：任意の要素
- item：仮変数
- list：任意の配列
- key：要素を一意に特定するキー

この例では、配列books（引数list）から順に書籍オブジェクトを取り出し、仮変数bに格納します。配下では、「b.isbn」のような形式で、オブジェクトのプロパティ値にアクセスできます。v-forでは、これを配列の中身がなくなるまで、繰り返すわけです。

［Note］key属性

key属性は、ループ内で要素を一意に識別するためのキー値を表します。key属性の指定は任意ですが、明示しておくことで、要素の増減などに際してVueが対象の要素を識別できるため、処理が効率的になります。増減の処理があるなしに関わらず、明示しておくことをお勧めします。

なお、ここでは、ISBNコードを渡していますが、一意性を保証できるならば、任意の値を指定して構いません（ただし、配列のインデックス番号などは要素の増減によって変動する可能性があるため、不可です）。

7.3.2 インデックス番号を取得する

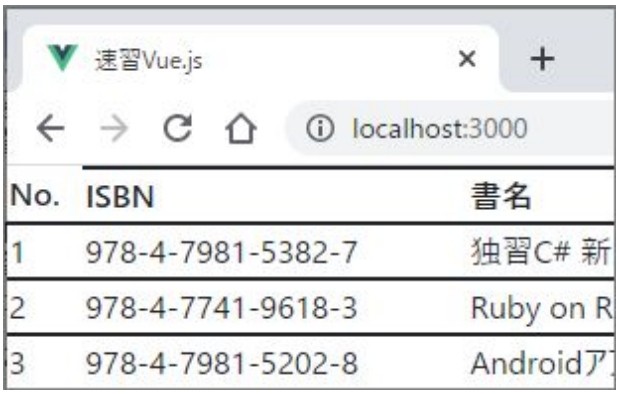
v-forの仮変数には、既定で配列要素がセットされます。しかし、仮変数を2個用意することで「配列要素, インデックス番号」の順にセットすることも可能で

す。先ほどの例に、インデックス番号でNo.を振ってみましょう。インデックス番号は0スタートなので、+1している点にも注目です。

［リスト］ForIndex.vue

```
<table class="table">
  <th>No.</th><th>ISBN</th><th>書名</th><th>価格</th>
  <tr v-for="(b, i) in books" v-bind:key="b.isbn">
    <td>{{ i + 1 }}</td>
    …中略…
  </tr>
</table>
```

図：インデックス番号をもとにNo.を振る



7.3.3 オブジェクトのプロパティを順に処理する

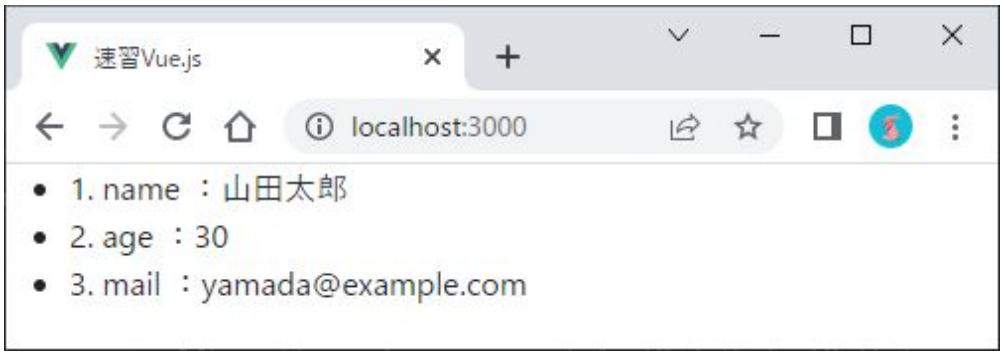
v-forでは、配列だけでなく、オブジェクトを繰り返し処理することもできます。以下は、オブジェクトmemberの内容を順にリスト表示する例です。

［リスト］ForObject.vue

```
<script setup>
import { reactive } from 'vue'
```

```
const member = reactive({
  name: '山田太郎',
  age: 30,
  mail: 'yamada@example.com'
})
</script>

<template>
  <ul>
    <li v-for="(value, key, i) in member" v-bind:key="key">
      {{ i + 1 }}. {{ key }} ：{{ value }}
    </li>
  </ul>
</template>
```

図：オブジェクトmemberの内容を順に列挙



　オブジェクトを扱う場合、仮変数は最大3個（先頭から順に「値、キー名、インデックス番号」）受け取れます。この例では、先頭から順にvalue、key、iで表しています。ただし、キー名、インデックス番号は不要であれば、省略しても構いません。

［Note］プロパティの列挙順

プロパティの列挙順は、ブラウザーの実装によって変動します。必ずしも定義順に沿うわけではないので、注意してください。

7.3.4 数値を列挙したい場合

v-forでは、配列／オブジェクトの代わりに、整数値を渡すこともできます。この場合、v-forは1～指定値の間で値を変化させながら、ループを繰り返します（いわゆるJavaScriptの一般的なforループです）。

［リスト］ForNumber.vue

```
<template>
  <span v-for="i in 5" v-bind:key="i">
    {{ i }}    </span>
</template>
```

▼結果

```
1 2 3 4 5
```

7.4 v-forによるループ処理の注意点

以上、v-forディレクティブの基本的な用法はごく明快ですが、よく利用するディレクティブだけに利用にあたっては注意点もあります。以下は、その主なものをまとめます。

7.4.1 配列の絞り込みには算出プロパティを利用する

たとえば価格が2500円未満の書籍情報だけを列挙したい場合には、以下のようにします。

［リスト］ForFilter.vue

```
<script setup>
import { ref, computed } from 'vue'
const books = ref([
  …中略…
])
// 2500円未満の書籍情報を取得する算出プロパティ
const cheapBooks = computed(() => {
  return books.value.filter(b => b.price < 2500)
})
</script>

<template>
  <table class="table">
    <th>ISBN</th><th>書名</th><th>価格</th>
    <tr v-for="b in cheapBooks" v-bind:key="b.isbn">
```

```
    ...中略...
   </tr>
  </table>
</template>
```

図：2500円未満の書籍だけを取得

filterはJavaScript標準のメソッドで、コールバック関数の条件に合致する（＝戻り値がtrueである）要素だけを返します。算出プロパティcheapBooksは、filterメソッドの戻り値を返すことで、フィルター済みの配列をv-forに渡しています。

［Note］ソートも同じように

配列をソートした上で列挙したい、という場合にも、同じように算出プロパティを利用できます。算出プロパティ経由で、ソート済みの配列を返すようにするわけです。

また、算出プロパティを利用できない状況では、メソッドを利用しても構いません。

7.4.2 異なる要素のセットを繰り返し出力する - <template>要素

v-forは、それが指定された開始タグから終了タグまでをひとつの塊として、要素を繰り返し出力します。その性質上、複数の要素セットをそのままv-forで出力することはできません。たとえば以下のコードであれば、<header>要素だけが繰り返しの対象となり、 ／<footer>要素はループの外です。

```
<header v-for="a in articles" ...>...</header>
<div>...</div>
<footer>...</footer>
```

もしも<header>〜<footer>要素をループの対象としたい場合、まずは以下のような方法があります。ループしたい要素一式を 要素で括っています。

```
<div v-for="a in articles" ...>
  <header>...</header>
  <div>...</div>
  <footer>...</footer>
</div>
```

ただし、v-forの都合で、本来のマークアップとしては無駄な 要素を一段介するのは望ましい状態ではありません。このような状況では、<template>要素を利用します。<template>は、その名の通り、テンプレートを定義するための要素で、それそのものは出力されません。複数の要素を束ねるためだけの役割を担います。

［リスト］ForMulti.vue

```
<script setup>
import { ref } from 'vue'
const articles = ref([
  {
    title: 'Angular TIPS',
    description: '人気のJavaScriptフレームワーク「Angular」...',
    author: '山田祥寛'
  },
  ...中略...
])
</script>

<template>
  <template v-for="a in articles" v-bind:key="a.title">
    <header>{{ a.title }}</header>
    <div>{{ a.description }}</div>
    <footer>{{ a.author }} 著</footer>
  </template>
</template>
```

▼結果

```
<header>Angular TIPS</header>
<div>人気のJavaScriptフレームワーク「Angular」の目的別リファレンス</div>
<footer>山田祥寛 著</footer>
<header>Tessel 2ではじめるセンサー電子工作入門</header>
<div>Tessel 2を使った面白い電子工作を紹介</div>
```

```
<footer>高江賢 著</footer>
<header>Web業界で働くためのPHP入門</header>
<div>「PHP」の文法を一から学ぶための入門連載</div>
<footer>齊藤新三 著</footer>
<header>jQuery逆引きリファレンス</header>
<div>jQueryの基本機能や実用Tipsを目的別で探せるリファレンス</div>
<footer>山田祥寛 著</footer>
```

［Note］<template>要素はv-ifでも利用できる

<template>要素は、v-ifで複数の要素を束ねる場合も同様に利用できます。

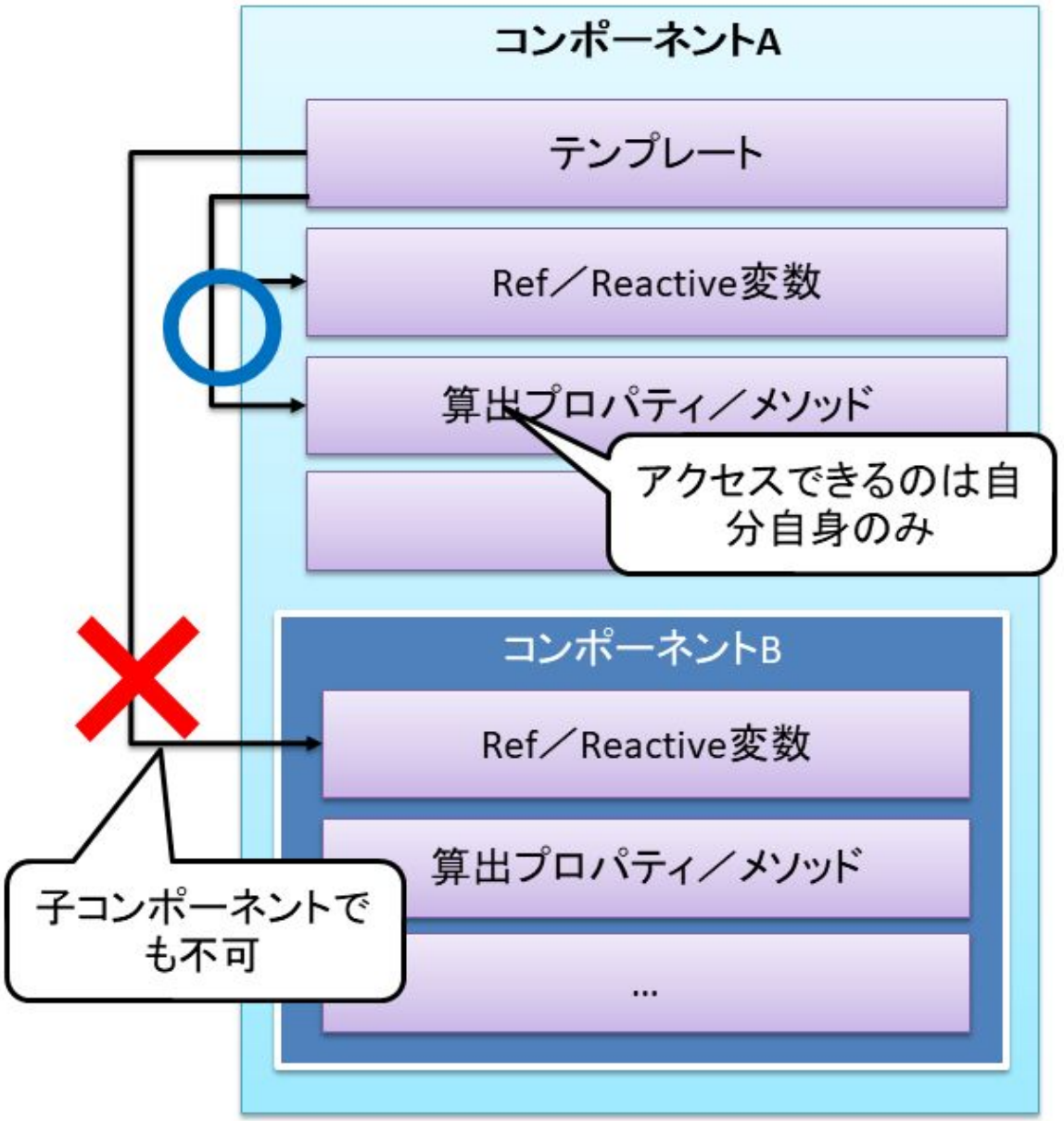
Part 8：コンポーネント連携

ここまでは、アプリの中で単一のコンポーネントが動作する例を紹介してきました。ただし、一般的なアプリでは、ひとつのコンポーネントにすべての機能を詰め込むことは稀です。複数のコンポーネントを並列に、あるいは入れ子に配置、組み合わせていくのが、より一般的なVueアプリの構成です。それによって、コードの見通しが改善するのはもちろん、部品としての再利用性も向上するからです。

8.1 コンポーネント連携の種類

コンポーネントを複数に分離するようになると、コンポーネント同士の連携（通信）にも無関心ではいられなくなります。というのも、コンポーネントで定義された情報――Ref／Reactive変数、算出プロパティ、メソッドなどの有効範囲（**スコープ**）は、あくまでコンポーネント内部に閉じています（コンポーネント外部からは見れらません）。

図：コンポーネントのスコープ



コンポーネント間で値を受け渡しするためには、以下のようなしくみを用いる必要があります。

- プロパティ：親コンポーネントから子コンポーネントへの伝達（属性）
- カスタムイベント：子コンポーネントから親コンポーネントへの伝達

- スロット：親コンポーネントから子コンポーネントへの伝達（タグ本体）
- Provide／Inject：親コンポーネントから子／子孫コンポーネントへの伝達
- Pinia：アプリ全体での情報共有

本章では、コンポーネント間連携の基本となるプロパティ／カスタムイベントを手始めに解説した後、スロット、Provide/Injectまでを解説していきます。PiniaについてはVue本体とは別建てのしくみのため、Part 12で改めます。

8.2 コンポーネントへのパラメーターの引き渡し - プロパティ

まずは、最もよく見かけるプロパティ（Props）からです。親コンポーネントから直下の子コンポーネントに対して、なんらかのパラメーターを渡すために利用できます。

8.2.1 プロパティの基本

具体的な例も見ていきましょう。たとえば以下は「こんにちは、●○さん！」というメッセージを表示するだけの、MyHelloコンポーネントの例です。MyHelloコンポーネントでは、●○に割り当てるべき名前をnameプロパティ経由で受け取れるものとします。

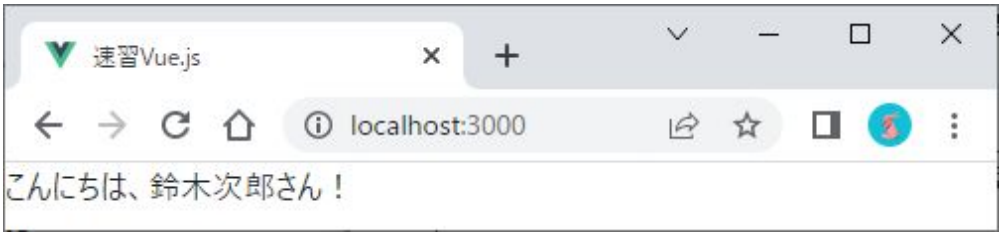
［リスト］PropBasic.vue

```
<script setup>
import MyHello from './MyHello.vue'
</script>

<template>
  <!--b. MyHelloコンポーネントの呼び出し -->
  <MyHello name="鈴木次郎" />
</template>
```

［リスト］MyHello.vue

```
<script setup>
// a. プロパティを定義
defineProps(['name'])
</script>

<template>
  <div>こんにちは、{{ name }}さん！ </div>
</template>
```

図：プロパティ値を結果にも反映



a. プロパティはdefineProps関数で定義する

プロパティはdefineProps関数に[プロパティ名, …]形式の文字列配列を渡すことで定義できます（a.）。ここではnameプロパティをひとつ定義しているだけですが、もちろん、複数のプロパティを列記しても構いません。

なお、definePropsを利用するにあたって、事前のインポートが**いらない**点にも注目です。というのも、definePropsは、`<script setup>`要素を処理するに先立って、あらかじめ処理される**コンパイラーマクロ**だからです。実行時に処理されるref／reactiveなどの関数とは異なるしくみなので、import命令による宣言も不要です。

b. 子コンポーネントを呼び出す

`<template>`要素の配下では、

- コンポーネントはタグ
- プロパティはその属性

として表せます。

属性（プロパティ）のデータ型についても注目です。サンプルのように「name="鈴木次郎"」と表した属性値は、すべて文字列です。この例では明快ですが、たとえば「name="108"」のような数値でもJavaScript側での扱いは文字列なので注意してください。

属性値を数値として渡したい場合には、v-bindを利用して「v-bind:name="108"」とします。もちろん、数値以外の任意のオブジェクトも渡せますし、Ref変数を引き渡す場合も同様です。

```
v-bind:name="name"
```

ただし、v-bind経由で文字列リテラルを渡す場合には要注意です。

```
v-bind:name="'鈴木次郎'"
```

v-bindで渡された値は、あくまでJavaScriptの式なので、文字列リテラルもクォートで括らなければならないのです。

［Note］グローバル登録とローカル登録

個々の.vueファイルでコンポーネントをインポート＆登録する方法を**ローカル登録**と言います。ローカル登録されたコンポーネントは、現在の.vueファイルでのみ有効となります（＝個々の.vueファイルで都度インポートする必要があります）。

アプリ全体で有効にしたい場合には、main.jsで以下のように登録してください（**グローバル登録**）。

```
import MyHello from './components/p08/MyHello.vue'
…中略…
app.component('MyHello', MyHello)
```

ただし、.vueファイル間の依存関係を明確にするという意味では、コンポーネントは極力ローカル登録することをお勧めします。本書でも、以降はローカル登録を優先して利用するものとします。

8.2.2 プロパティ値の型を制限する

definePropsマクロには、単にプロパティ名を列挙するだけでなく、「プロパティ名: 検証ルール」形式のオブジェクトを指定することもできます。これによって、プロパティ値が（たとえば）数値型であるか、そもそも指定されているかをVueが検証してくれます。たとえば、以下は、先ほどのMyHello.vueを検証ルールを使って書き換えたものです。

［リスト］MyHello.vue

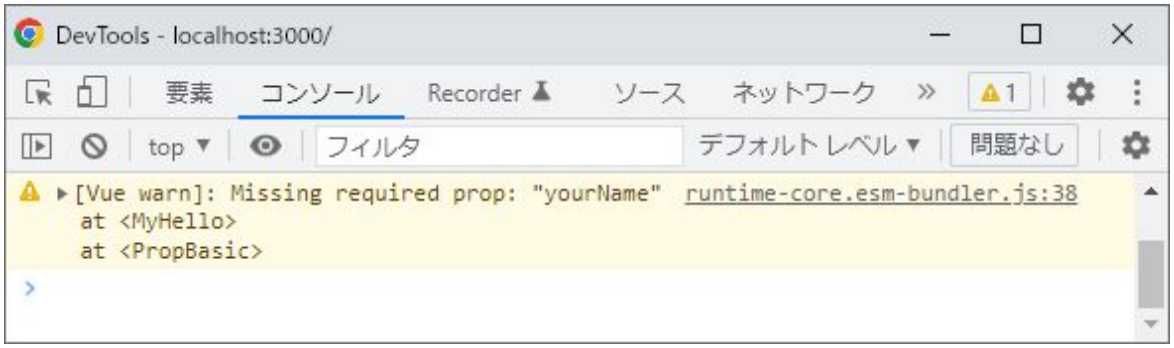
```
<script setup>
defineProps({
  name: {
    type: String,
    required: true
  }
})
</script>
```

検証ルールには「ルール名: 値」形式のオブジェクトとして指定できます。ルール名として指定できるのは、以下のものです。

- type ： データ型 （String、Number、Boolean、Function、Object、Array、Symbol、または任意のclass型[1]）
- required：プロパティが必須か
- default：値が指定されなかった場合の既定値
- validator：カスタムの検証関数

この例であればnameプロパティがString型で必須であることを示しています。果たして、この状態で呼び出し側のname属性を省略してみると、コンソール（デベロッパーツール）に「Missing required prop: "name"」のようなエラーメッセージが表示されます。

図：不正なプロパティ値を警告



8.2.3 検証ルールのさまざまな表現方法

以下では、主な検証ルールの表現方法について示しておきます。

（1）データ型だけを指定する

データ型（typeパラメーター）だけを指定したい場合には、「name: String」のように、型名をそのまま表記できます。最もシンプルな検証ルールの表記です。

プロパティが複数の型を取りうる場合には（たとえばNumberかStringのように）型名を配列として渡してください。「name: [String, Number]」のように、です。

（2）配列／オブジェクトの既定値は注意

defaultパラメーター（既定値）に配列／オブジェクトを指定する場合には、既定値そのものではなく、既定値を返す関数を渡します。

```
propName: {
  type: Object,
  // 既定値を返す関数を指定
```

```
  default: () => ({ value: '権兵衛' })
},
```

戻り値（オブジェクトリテラル）全体をカッコで括っているのは、アロー関数では{...}がブロックと認識されてしまうからです。以下のようにカッコなしで記述した場合には「value:」はラベル、「'権兵衛'」がただの文字列と見なされ、既定値はundefinedとなります。

```
default: () => { value: '権兵衛' }
```

戻り値全体を丸カッコで括ることで、正しくオブジェクトリテラルと見なされます。

（3）自作の検証ルールも指定できる

validatorパラメーターを利用することで、自作の検証ルールを指定することもできます。たとえば以下はnameプロパティが文字列で、文字数が5文字以内であることをチェックします。検証関数は、引数としてプロパティ値を受け取り、戻り値として検証の成否をtrue／falseで返すようにします。

```
defineProps({
  name: {
    type: String,
    required: true,
    validator: v => v.length <= 5
```

```
  }
})
```

8.2.4 プロパティ利用の注意点

以下では、プロパティを利用するにあたって注意すべき点を幾つかまとめておきます。

プロパティは読み取り専用

プロパティ値は親コンポーネントから任意のタイミングで変更される可能性があります。よって、コンポーネント内部でプロパティ値を変更してはいけません（実際、コンポーネント内部でプロパティ値を変更した場合には「Set operation on key "init" failed: target is readonly」のような警告が発生し、値も反映されません）。

もしもなんらかの理由でプロパティ値を操作したい場合には、値を一旦Ref変数に退避させるようにしてください。

たとえば以下は、［増加］ボタンのクリック回数をカウントできるMyCounterコンポーネントの例です。カウンターの初期値をinit属性で引き渡せるものとします。

［リスト］PropInner.vue

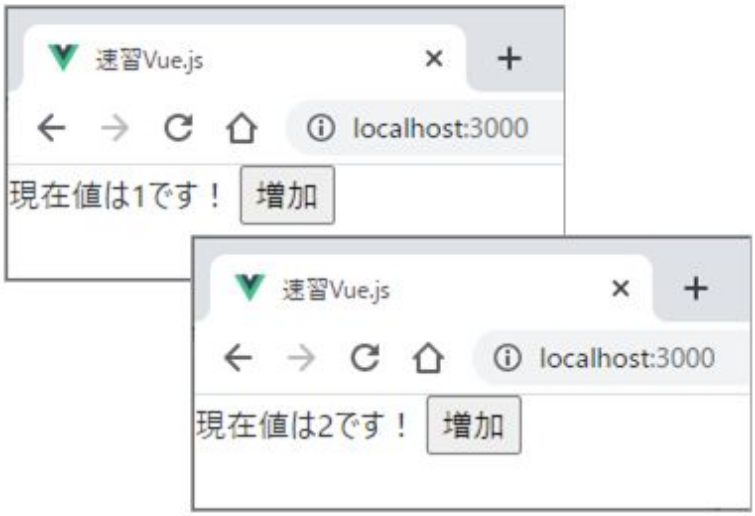
```
<script setup>
import MyCounter from './MyCounter.vue'
</script>
```

```
<template>
  <MyCounter v-bind:init="1" />
</template>
```

［リスト］MyCounter.vue

```
<script setup>
import { ref, toRef } from 'vue'
// a. プロパティを定義
const props = defineProps({
  init: Number
})
// Ref変数を定義
const current = ref(props.init)
// b. ボタンクリック時にcurrentプロパティをインクリメント
const onclick = () => current.value++
</script>
```

図：［増加］ボタンクリックでカウンターをインクリメント



この例であれば、init属性（プロパティ）をそのまま更新することはできません。そこでコンポーネント内部のRef変数currentに退避させ、これを更新するようにします。

<script>要素でプロパティ値にアクセスするには、definePropsの戻り値を一旦、変数で受け取っておくようにしましょう（a.）。これで「props.名前」でプロパティ値にアクセスできるようになります。

あとは、clickイベントハンドラーで（initプロパティではなく）Ref変数currentを操作します（b.）。

プロパティ宣言を省略した場合

コンポーネントには、definePropsで定義した以外のプロパティを渡すこともできます。たとえば以下の例であれば、太字部分―id／class属性が未定義のプロパティです。

［リスト］PropBasic.vue

```
<MyHello name="鈴木次郎" id="hello" class="hoge" />
```

この場合、未定義のプロパティはコンポーネントのルート要素（この例であれば 要素）に反映されます。

```
<div id="hello" class="hoge">こんにちは、鈴木次郎さん！ </div>
```

ただし、ルート要素が複数ある場合には「Extraneous non-props attributes (id, class) were passed to component but could not be automatically inherited ～」のような警告が発生します。

また、そもそも未定義の属性をルート要素に無条件に反映させたくない場合には、inheritAttrsというオプションをfalseに設定します。

［リスト］PropAttr.vue

```
<script setup>
import MyHelloAttr from './MyHelloAttr.vue'
</script>

<template>
  <MyHelloAttr name="鈴木次郎" id="hello" class="hoge" />
</template>
```

［リスト］MyHelloAttr.vue

```
<script>
// a. 属性の自動的な反映をオフに
```

```
export default {
  inheritAttrs: false
}
</script>

<script setup>
defineProps({
  name: String
})
</script>

<template>
  <!--b. 属性が反映されないことを確認 -->
  <div>こんにちは、{{ name }}さん！ </div>
</template>
```

▼結果

```
<div>こんにちは、鈴木次郎さん！ </div>
```

a.の記述は、Options APIと呼ばれる記法です。本書では扱っていないので、詳しくは「速習 Vue.js 3」（Amazon Kindle）を参照いただくとして、ここでは太字をイディオムとして理解してください。意識すべきは、inheritAttrsオプションは、**`<script setup>`配下には書けない**（＝ただの`<script>`配下に記述しなければならない）という点です。

結果を確認すると、確かに未定義の属性がルート要素に反映**されなく**なっていることが判ります。もちろん、自動で反映されなくなっただけで、内部的には$attrsという変数に情報が反映されています。試しに**b.**を書き換えてみましょう。

```
<div v-bind="$attrs">こんにちは、{{ name }}さん！ </div>
```

▼結果

```
<div id="hello" class="hoge">こんにちは、鈴木次郎さん！ </div>
```

太字の記述は、属性の一括反映です（詳しくは4.4.2項を参照してください）。今度は、確かに未定義の属性が`<div>`要素に反映されることが確認できます。

もちろん、この例であればinheritAttrsオプションをtrue（既定）に戻せばよいだけですが、$attrsを利用することで、任意の要素に対して属性値をばらすことが可能になります。

［Note］特定の属性にアクセスするには？

特定の属性を取り出すならば、以下のようにも書けます。

```
<div v-bind:id="$attrs.id">こんにちは、{{ name }}さん！ </div>
```

ルートコンポーネントへの属性の渡し方

ルートコンポーネントだけは、createApp経由で呼び出す都合上、プロパティの指定方法も異なります。具体的には、createAppメソッドの第2引数に「プロパティ名: 値, ...」のオブジェクトリテラル形式で指定してください。

［**リスト**］**main.js**

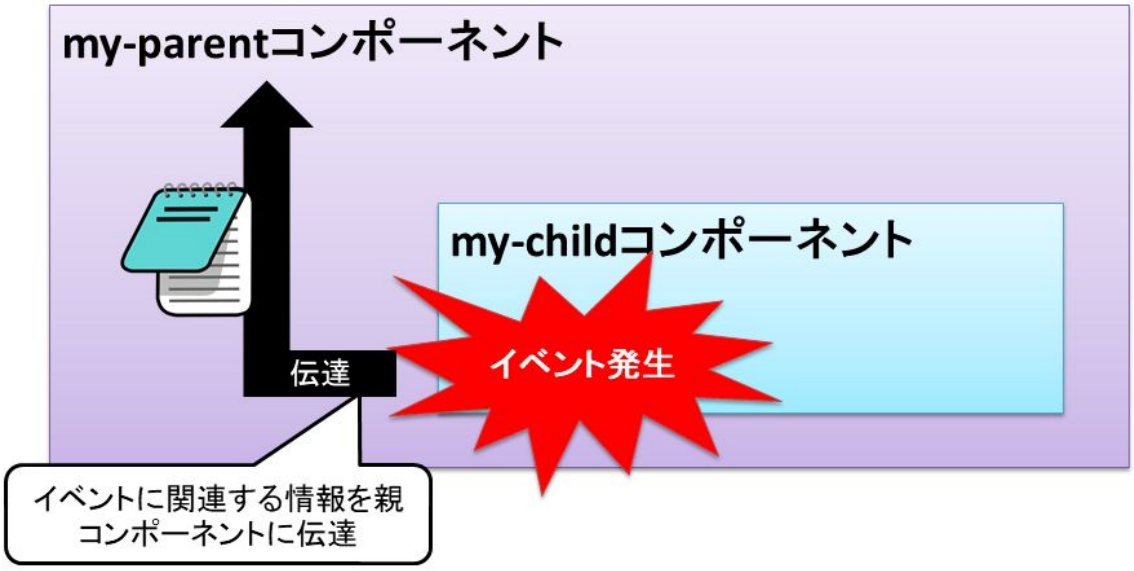
```
import MyHello from './components/p08/MyHello.vue'
...中略...
const app = createApp(MyHello, { name: '鈴木次郎' })
```

あまり利用する機会はありませんが、属性付きのコンポーネントを単体で試すような場合には、ルートコンポーネントを介さなくてよい分、手軽になります。

8.3 子コンポーネントから親コンポーネントへの伝達 - カスタムイベント

親コンポーネントから子コンポーネントに対して情報を伝達するプロパティに対して、子コンポーネントから親コンポーネントに対して情報を渡すには、**カスタムイベント**というしくみを利用します。

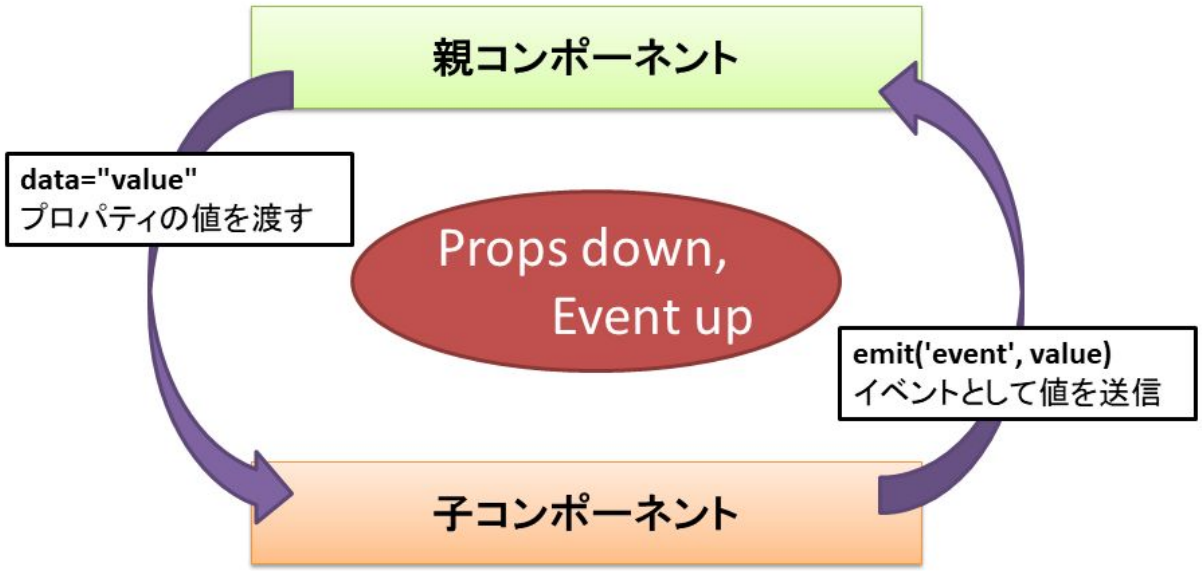
図：カスタムイベント



子コンポーネントでなにかしらの処理を行った時に、親コンポーネントに対して「処理の結果、なんらかの変化が生じたこと」（イベント）を通知するわけです。その際に、イベントに関連する情報（オブジェクト）を添付できます。このオブジェクトが、子⇒親方向に流れる情報です。

Vueの世界では、親⇒子方向の情報伝達をPropsで、子⇒親方向の情報伝達をEventで行うのが基本です。このようなしくみを**Props down, Event up**と呼びます。

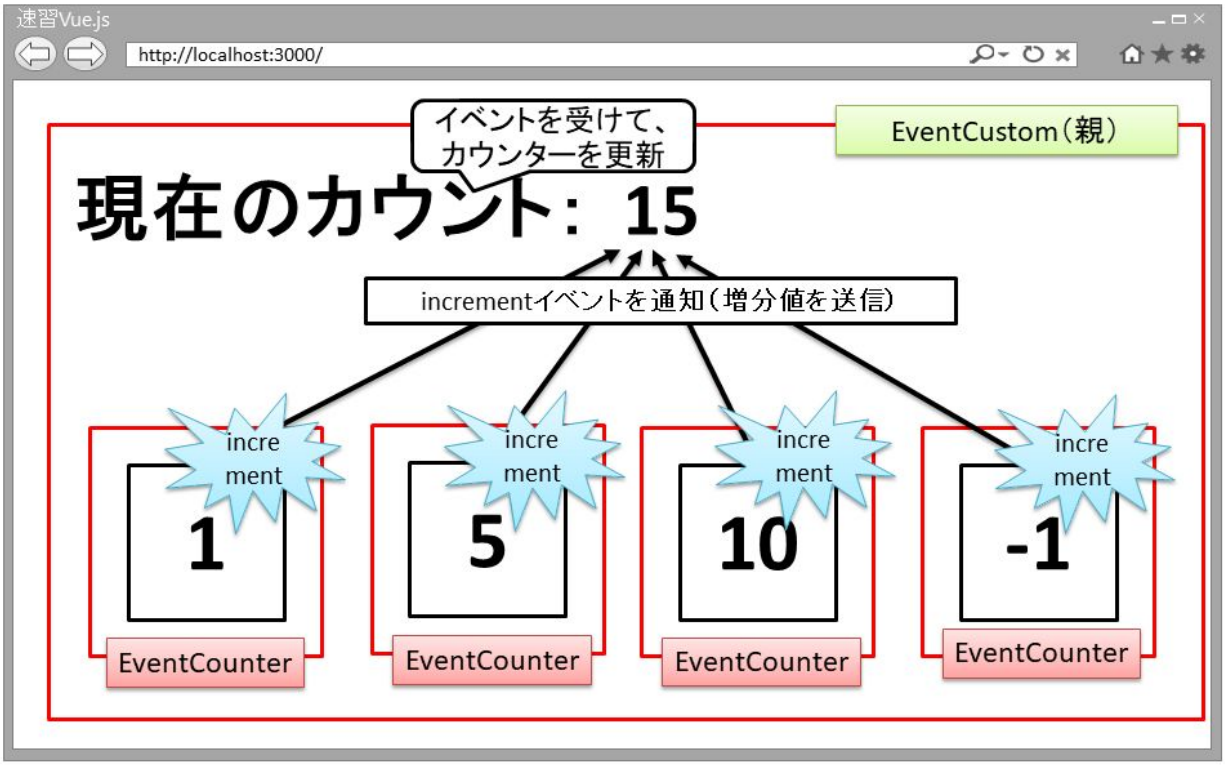
図：親／子コンポーネントでの情報伝達



8.3.1 カウンターアプリの例

早速、具体的な例も見ていきましょう。ページには［1］［5］［10］［-1］ボタンが配置されており、これらのボタンをクリックすると、上部のカウンターがボタンの値に応じて変化していく――そんなサンプルです。

図：カウンターアプリの例



親コンポーネントEventCustomの配下に、子コンポーネントEventCounterが4個配置されている構造です。この関係において、EventCounterではカウント値そのものを管理**しない**点に注目です。EventCounterは増減ボタンを提供するだけで、カウント値そのものは親コンポーネントで集中管理しているのです。

この時、EventCounter（子）が増減すべき値を伝えるために用いているのが、カスタムイベントです。具体的には、ボタンがクリックされたタイミングで、独自のincrementイベントを発生させ、増減値を通知します。

以上の関係を理解した上で、ここからは具体的なコードを見ていきます。

8.3.2 incrementイベントの実装

まずは、カスタムイベントの発生側――子コンポーネントであるEventCounterからです。

［リスト］EventCounter.vue

```
<script setup>
import { ref } from 'vue'
// プロパティの定義（増減値）
const props = defineProps([ 'step' ])
// a. incrementイベントの宣言
const emit = defineEmits([ 'increment' ])
// b. クリック時に増減値を通知（イベント発生）
const onclick = () => emit('increment', Number(props.step))
</script>

<template>
  <input type="button" v-on:click="onclick" v-bind:value="step" />
</template>
```

まずは、defineEmits関数で利用可能なイベントを列挙しておきます（**a.**）。引数は[イベント名, …]形式の配列です。definePropsと同じく、コンパイラーマクロなので、事前のインポートも不要です。

ただし、カスタムイベントは宣言しただけでは意味がありません。なんらかのタイミング（一般的にはイベントハンドラー）でイベントを発生させる必要があります。これを行うのがemit関数です（**b.**）。emit関数は、正しくはそういう名前の関数が存在するわけではなく、defineEmitsメソッドの戻り値を便宜上そのように呼んでいます。

［構文］ **emit関数**

emit(*event* [, *args*])

- event：イベント名
- args：親コンポーネントに伝達するデータ

引数argsは、親コンポーネントに引き渡す値を表します。この例では、プロパティ経由で受け取った増減値（step）をNumberに変換したものを渡しているだけですが、複数の値を渡すならば「プロパティ名: 値,...」の形式で束ねます。

8.3.3 カスタムイベントの監視

カスタムイベントを監視し、イベント発生時の処理を表すのは親コンポーネントの役割です。

［リスト］ **EventCustom.vue**

```
<script setup>
import { ref } from 'vue'
import EventCounter from './EventCounter.vue'
// カウンター値
const count = ref(0)
// b. incrementイベントでカウンター値を増減
const onincrement = e => count.value += e
</script>

<template>
  <div>現在のカウント：{{ count }}</div>
  <div>
```

```
    <!--a. 子コンポーネントのイベントを監視-->
    <EventCounter v-bind:step="1" v-on:increment="onincrement" />
    <EventCounter v-bind:step="5" v-on:increment="onincrement" />
    <EventCounter v-bind:step="10" v-on:increment="onincrement" />
    <EventCounter v-bind:step="-1" v-on:increment="onincrement" />
  </div>
</template>
```

カスタムイベントも、標準のイベントと同じく、v-onディレクティブで監視できます。この例であれば「incrementイベントを受けて、onincrementハンドラーを実行」します（**a.**）。

onincrementメソッド（**b.**）でイベントオブジェクトeを受け取るのも、5.2.1項で学んだ通りです。イベントオブジェクトeに引き渡されるのは、先ほどemit関数の引数argsで指定された値（増減値）です。ここでは、その値をcountプロパティに加算しています。

8.3.4 カスタムイベントの検証

プロパティと同じく、カスタムイベントでも渡された値が妥当かどうかを検証できます。これには、イベント宣言（defineEmits）に対して「イベント名: 検証関数, ...」形式のオブジェクトを渡します。

［リスト］EventCounter.vue

```
const emit = defineEmits({
  // incrementイベントの検証
  increment: step => {
```

```
    // step値が整数値であれば正しい
    if (step && Number.isInteger(step)) {
      return true
    } else {
      console.log('Invalid increment event!!')
      return false
    }
  }
})
```

検証関数は、emit関数の第2引数で渡されたイベントデータを受け取ります（この例であればstep）。ここでは、引数stepが整数であればイベントは妥当、さもなければ不正であるとみなします。検証関数は妥当性の是非をtrue／falseで返すようにします。

以上を理解したら、コンポーネントの呼び出し側を以下のように修正して、あえて検証エラーを発生させてみましょう。

［リスト］EventCustom.vue

```
<div>
  <EventCounter step="Hoge" v-on:increment="onincrement" />
  …中略…
</div>
```

step属性に文字列を渡しているので、incrementイベントはエラーです。コンソール（デベロッパーツール）に「Invalid event arguments: event validation

failed for event "increment"」のような警告が返されることを確認してください。

図：不正なイベント呼び出しに警告

8.4 コンポーネント配下のコンテンツをテンプレートに反映させる - スロット

属性の形式で値を受け渡しするプロパティに対して、タグ配下のコンテンツとして値を表せるのが**スロット**です。属性（プロパティ）がその性質上、短い値、もしくは単一のRef変数を引き渡すのに向いているのに対して、スロットがより長い値――テンプレートそのものを引き渡すのに向いたしくみです。

8.4.1 スロットの基本

たとえば以下は、8.2.1項のMyHelloコンポーネントを修正して、挨拶メッセージに埋め込むべき名前を、配下のコンテンツとして指定するようにしてみます（伴い、名前もMyHelloSlotとしています）。

［リスト］SlotBasic.vue

```
<script setup>
import MyHelloSlot from './MyHelloSlot.vue'
</script>

<template>
  <!-- b. MyHelloSlotコンポーネントの呼び出し -->
  <MyHelloSlot>鈴木次郎</MyHelloSlot>
</template>
```

［リスト］MyHelloSlot.vue

```
<template>
  <!-- a. テンプレートに<slot>要素を埋め込む-->
  <div>こんにちは、<slot>権兵衛</slot>さん！ </div>
</template>
```

▼結果

```
<div>こんにちは、鈴木次郎さん！ </div>
```

スロットを利用するには、テンプレートの配下に`<slot>`要素を埋め込むだけです（**a.**）。これによって呼び出し側で指定されたコンテンツ（**b.**）が、`<slot>`要素のあった場所に埋め込まれます。

`<slot>`要素配下のコンテンツ（ここでは「権兵衛」）は、呼び出し元からコンテンツが渡されなかった場合に出力される既定のコンテンツです。

8.4.2 複数のスロットを利用する

テンプレートに複数のスロットを準備し、呼び出し側から複数のコンテンツを埋め込むことも可能です。まずは、具体的な例を見てみましょう。

［リスト］SlotMulti.vue

```
<script setup>
import MyHelloMulti from './MyHelloMulti.vue'
</script>

<template>
  <MyHelloMulti>
```

```
    <!--b. 名前付きスロットに値を引き渡す-->
    <template v-slot:header>
      <h3>ようこそ速習Vue.jsへ</h3>
    </template>
    <p>一緒に勉強しましょう。</p>
    <template v-slot:footer>
      <a href="#">Q＆A掲示板</a>
    </template>
    <p>質問は掲示板へどうぞ。</p>
  </MyHelloMulti>
</template>
```

［リスト］MyHelloMulti.vue

```
<template>
  <!-- a. 名前付きのスロットを準備-->
  <div>
    <header>
      <slot name="header"></slot>
    </header>
    <div>
      <slot></slot>
    </div>
    <footer>
      <slot name="footer"></slot>
    </footer>
  </div>
</template>
```

▼結果

```
<div>
  <header>
    <h3>ようこそ速習Vue.jsへ</h3>
  </header>
  <div>
    <p>一緒に勉強しましょう。</p>
    <p>質問は掲示板へどうぞ。</p>
  </div>
  <footer>
    <a href="#">Q＆A掲示板</a>
  </footer>
</div>
```

複数のスロットを埋め込む場合には、互いを区別できるよう、`<slot>`要素にname属性を付与します（**a.**）。「`<slot> </slot>`」のような名前のないスロットは、既定でdefaultと命名されます（つまりこの例では、上からheader、default、footerスロットが用意されたことになります）。

あとは、これに対応するよう、呼び出し元でもスロット単位にテンプレートを準備します。

> **［構文］<template>要素**
> <template v-slot:*name*>*contents*</template>

- name：埋め込み先スロットの名前
- contents：スロットに埋め込むコンテンツ

埋め込み先はv-slot:xxxxx属性で指定します。また、<template>要素で括られなかった要素（この例では2個の<p>要素）は、既定のスロット（＝name属性のない<slot>要素）に埋め込まれます。

［Note］v-slotの省略構文

v-bind／v-onと並んで、v-slotはよく利用することから、省略構文が用意されています。先ほどのSlotMulti.vue（headerテンプレートのみ）を省略構文で書き換えると、以下のようになります。

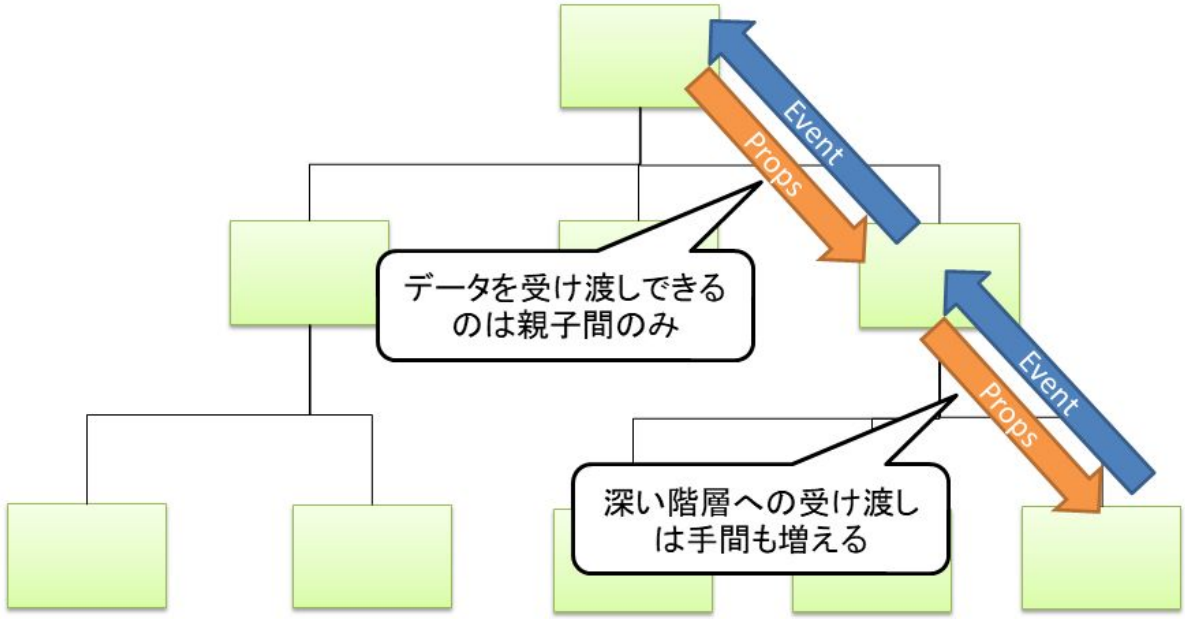
```
<template #header>
  <h3>ようこそ速習Vue.jsへ</h3>
</template>
```

いずれを利用しても構いませんが、先のv-bind／v-onの時と同じく、アプリの中では記法を揃えることを強くお勧めします。本書では、初学者にも意図を汲み取りやすいよう、非省略構文を利用していきます。

8.5 子孫コンポーネントへの値の引き渡し - Provide／Inject

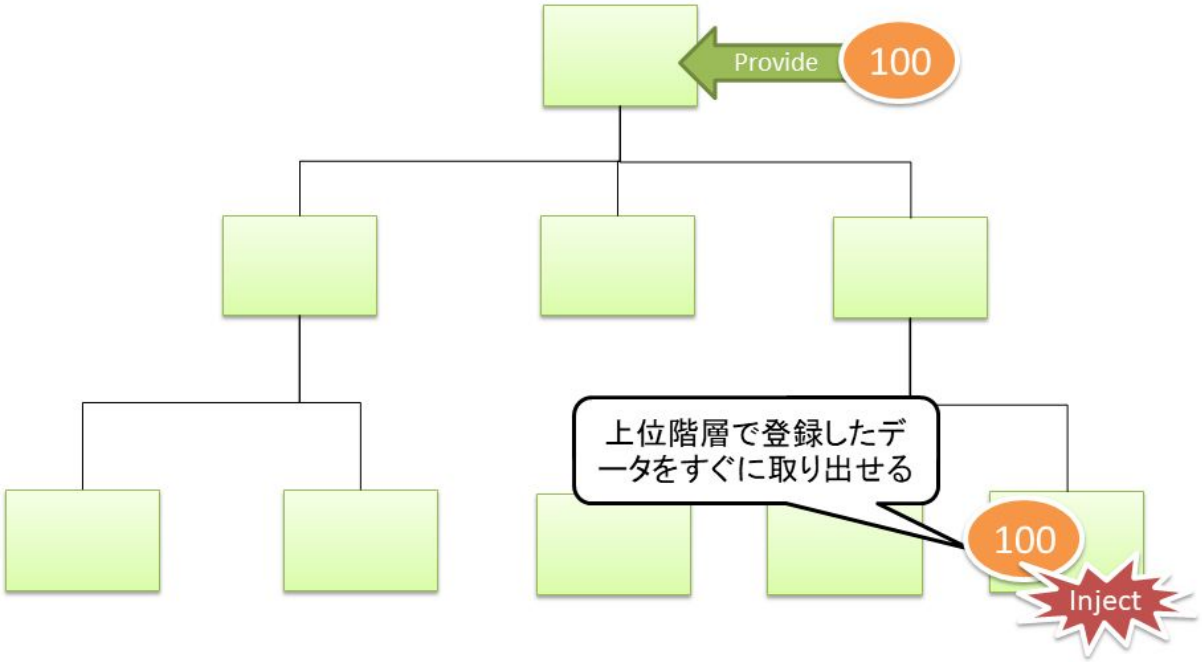
「Props down, Event up」のアプローチはVueコンポーネントの基本ですが、問題もあります。というのも、コンポーネント階層が深くなった場合に、データのバケツリレーが発生します。これは冗長であるのみならず、潜在的なバグの原因ともなります。

図：Props down, Event upの問題



Provide／Injectは、このようなバケツリレーを解消するためのしくみです。上位のコンポーネントで提供（Provide）された値を、配下の任意のコンポーネントで取得（Inject）できるようになります。

図：Provide／Injectによる解決



8.5.1 値をProvideする

早速具体的な例を見てみましょう。以下は、親コンポーネント（ProvideBasic）で用意された書籍情報を子コンポーネント（InjectList）で取得し、一覧表示する例です。

図：親がProvideした書籍情報を取得＆表示

まずは親コンポーネント――Provide側から見ていきます。

［リスト］ProvideBasic.vue

```
<script setup>
import InjectList from './InjectList.vue'
import { ref, provide } from 'vue'
const books = ref([
  {
    isbn: '978-4-7981-5382-7',
    title: '独習C# 新版',
    price: 3600
  },
  …中略…
])
// a. 用意済みのRef変数をProvide値として登録
provide('list', books)
</script>
```

```
<template>
  <InjectList />
</template>
```

値をProvide（提供）するには、上位のコンポーネントでprovide関数を呼び出すだけです（a.）。

> **［構文］provide関数**
> provide(*key*, *value*)
> - key：キー
> - value：値

引数keyは、あとで値を取り出すときに利用するキー値です。互いに識別できるよう、コンポーネント階層内で一意となる名前を付与しておきましょう。ここではlistというキーでRef変数booksをひとつだけ登録していますが、もちろん、provide関数を列挙することで複数のキーを登録することも可能です。

> **［Note］アプリレベルでのProvide**
> アプリレベルでProvideするならば、アプリインスタンス経由でprovideメソッドを呼び出すことも可能です。
>
> ```
> app.provide('list', books)
> ```

8.5.2 値をInjectする

上位コンポーネントで提供（Provide）された値を、下位のコンポーネントに注入（Inject）するには、inject関数を利用します。

［リスト］InjectList.vue

```
<script setup>
import { inject } from 'vue'
// a. Provideされた値を注入
const books = inject('list')
</script>

<template>
  <table class="table">
    <th>ISBN</th><th>書名</th><th>価格</th><th></th>
    <tr v-for="b in books" v-bind:key="b.isbn">
      <td>{{ b.isbn }}</td>
      <td>{{ b.title }}</td>
      <td>{{ b.price }}円</td>
    </tr>
  </table>
</template>
```

inject関数（a.）では、Provide時に指定したキーで値を取得できます。以下のように、Provide値が見つからなかった時に利用する既定値を指定することも可能です。

```
const books = inject('list', [])
```

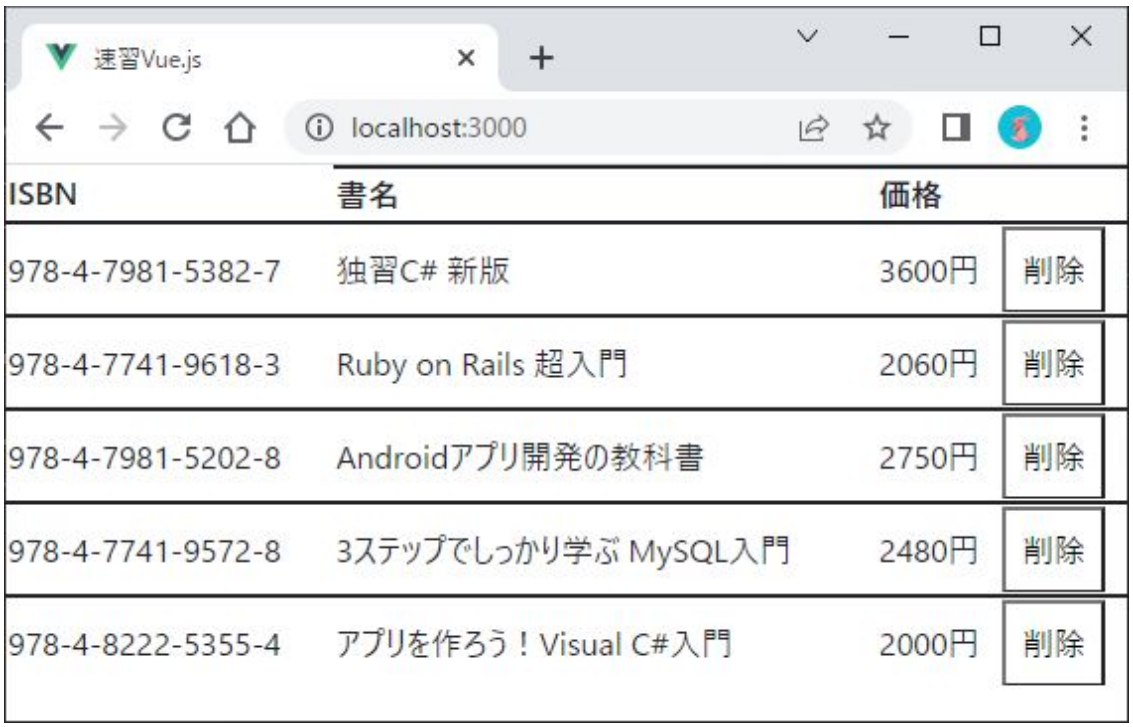
先ほどのProvideBasic.vueを見ても、プロパティなどを指定することなく、上位コンポーネントの値を下位コンポーネントに引き渡せていることが確認できます。この例では、ProvideBasic／InjectListは直接の親子関係にありますが、間に複数の階層が挟まっていても同じように値を受け渡しできます。

8.5.3 Provide値の操作

Provide値をInject側で更新することもできますが、その場合、更新コードもProvide側でまとめるべきです。これによって、更新元が限定されるので、値の変化も追跡しやすくなります。

具体的な例として、先ほどのサンプルに削除機能を追加してみましょう。一覧から［削除］ボタンをクリックすることで、既存の書籍情報を削除できるようにします。

図：書籍一覧に削除機能を付与



以下に差分のコードを示します（太字が追加／変更部分）。

［リスト］ProvideBasic.vue

```
<script setup>
…中略…
// 指定されたisbnで書籍情報を削除（フィルター）
const onclick = isbn => {
  books.value = books.value.filter(b => b.isbn !== isbn)
}
// a. データと操作メソッドをまとめてProvide
provide('list', { books, onclick })
</script>
```

［リスト］InjectList.vue

```
<script setup>
import { inject } from 'vue'
// b. listキーの内容をInject（個々の変数に分解）
const { books, onclick } = inject('list')
</script>

<template>
  <table class="table">
    <th>ISBN</th><th>書名</th><th>価格</th><th></th>
    <tr v-for="b in books" v-bind:key="b.isbn">
      …中略…
      // c. isbnをキーに削除するボタン
      <td><input type="button" value="削除"
        v-on:click="onclick(b.isbn)" /></td>
    </tr>
  </table>
</template>
```

まず、Provide側で操作コードをまとめて登録します（**a.**）。{ books, onclick }はオブジェクトの省略構文で、{ books: books, onclick: onclick }と書いても同じ意味です。複数のキーにばらしてProvideしても構いませんが、関連する情報であれば、このように一個のオブジェクトに束ねた方が扱いやすいはずです。

伴い、Inject側も**b.**のように修正します。これでlistキーの内容がbooks／onclickに分解されます。あとは**c.**のようにonclick関数を呼び出すためのボタンを設置するだけです。

8.5.4 補足：共通コードの分離

値の生成／操作コードがアプリを跨って再利用できるものであるならば、これを別ファイルとして切り出すことも可能です（また、積極的に切り出していくべきです）。

たとえば先ほどの書籍情報の生成、操作コードを切り出してみましょう（ここでは書籍情報はハードコーディングしていますが、一般的にはネットワークなどから取得することになるはずです）。保存先は、通常のコンポーネントとは切り離して、/src/components/compositeフォルダーとしています。

［リスト］useBook.js

```
import { ref } from 'vue'

export default function() {
  const books = ref([
    {
      isbn: '978-4-7981-5382-7',
      title: '独習C# 新版',
      price: 3600
    },

])
  const onclick = isbn => {
    books.value = books.value.filter(b => b.isbn !== isbn)
  }
  // コンポーネントに提供すべき情報を束ねる
  return {
```

```
    books,
    onclick
  }
}
```

このように再利用を目的として切り出した関数を**コンポジション関数**と呼びます。コンポジション関数は、一般的に、以下のようなルールで関数を定義します。

- 戻り値はコンポーネントで利用するRef／Reactive変数、メソッドの集合
- 名前（ファイル名）はuseXxxxxとする

この例であれば、書籍情報を管理するためのコンポジション関数なので、useBookとしています。

作成したコンポジション関数は、以下のように利用できます（先ほどのProvideBasic.vueを修正したものです）。

［リスト］CompositeBasic.vue

```
<script setup>
import { provide } from 'vue'
import InjectList from './InjectList.vue'
import useBook from '../composite/useBook.js'
// コンポジション関数から必要な情報を取得
const { books, onclick } = useBook()
provide('list', { books, onclick })
</script>
```

```
<template>
  <InjectList />
</template>
```

useBook関数を呼び出し、その結果を変数に反映させるだけなので、特に迷うことはありませんね。書籍情報の取得／操作コードを外部化したことで、コンポーネント本体のコードはぐんと見通し良くなったことが確認できます。

この例であれば、useBook関数の内容をそのままProvideしているだけなので、太字は以下のように書いても同じ意味です。

```
provide('list', useBook())
```

---

1. class型を指定した場合、内部的には属性値がinstanceof演算子で指定の型と比較されます。⏎

Part 9：組み込みコンポーネント

Vueでは、事前の定義なしに利用できる組み込みコンポーネントも用意されています [1]。

- `<component>`：指定されたコンポーネントを実行
- `<KeepAlive>`：コンポーネントの状態を維持
- `<slot>`：名前付きスロットを定義（8.4節）
- `<Transition>`／`<TransitionGroup>`：配下の要素に対してアニメーションを適用
- `<Teleport>`：配下の要素をページの異なる場所に移動
- `<Suspense>`：非同期処理に際して、待ち受けメッセージを表示

本Partでは、既出の`<slot>`要素を除く、その他のコンポーネントについて解説していきます。

9.1 コンポーネントを動的に切り替える - <component> 要素

<component>要素を利用することで、コンポーネントの動的な入れ替えが可能になります。コンポーネントを表示する――コンポーネントの入れ物とも言うべきコンポーネントで、**メタコンポーネント**、**動的コンポーネント**とも呼びます。

9.1.1 動的コンポーネントの基本

具体的な例も見てみましょう。たとえば以下は、選択ボックスでの指定に応じてコンテンツ（コンポーネント）を切り替える例です。切り替え対象には、BannerNew.vue、BannerWings.vue、BannerMail.vueと3種類のコンポーネントを用意しておきます（テンプレートだけの簡単な内容なので、BannerMail.vueのみ掲載します。完全なコードはダウンロードサンプルから参照してください）。

［リスト］MetaComp.vue

```
<script setup>
import BannerNew from './BannerNew.vue'
import BannerWings from './BannerWings.vue'
import BannerMail from './BannerMail.vue'
import { shallowRef } from 'vue'
// b. 浅いRef変数を宣言
const selected = shallowRef(BannerNew)
</script>
```

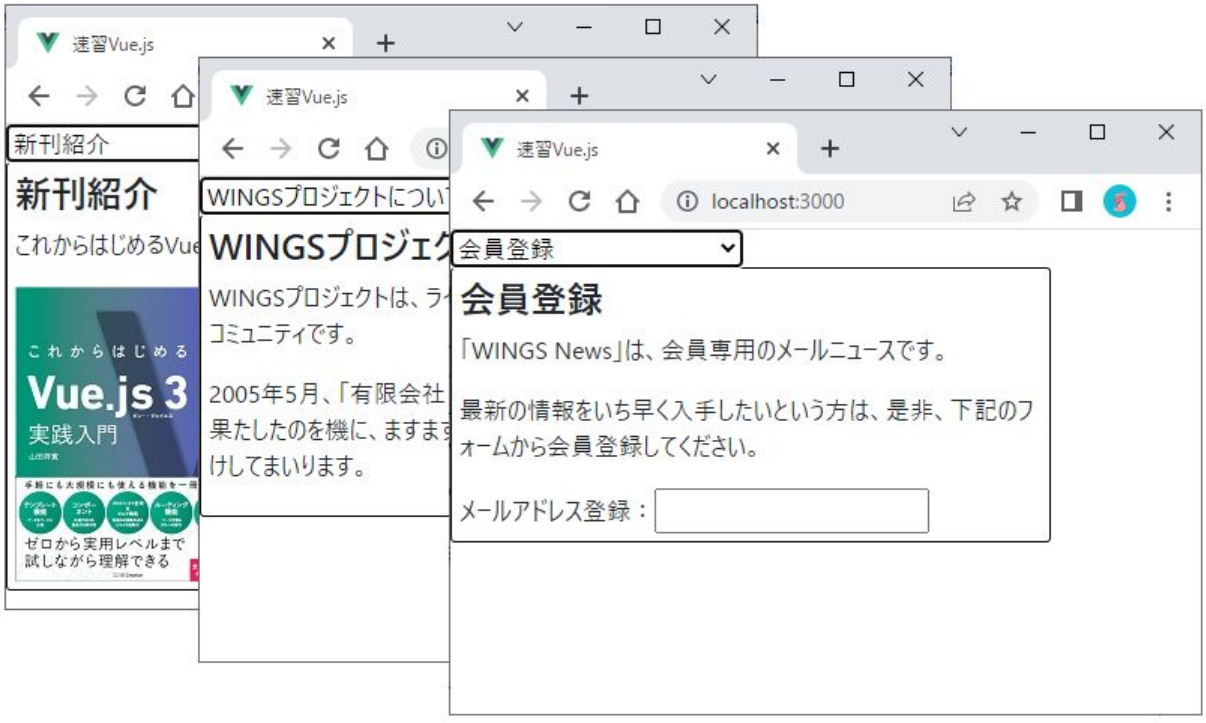
```
<template>
  <form>
    <select v-model="selected">
      <option v-bind:value="BannerNew">新刊紹介</option>
          <option  v-bind:value="BannerWings">WINGS プロジェクトについて
</option>
      <option v-bind:value="BannerMail">会員登録</option>
    </select>
    <!--a. コンポーネントを反映する領域 -->
    <component v-bind:is="selected" />
  </form>
</template>

<!--c. 深いスタイルを定義-->
<style scoped>
:deep(.banner) {
  border: 1px solid #000;
  border-radius: 3px;
  padding: 5px;
  width: 80%;
}
</style>
```

［リスト］BannerMail.vue

```
<template>
  <div class="banner">
    <h3>会員登録</h3>
```

```
    <p>「WINGS News」は、会員専用のメールニュースです。</p>
    <p>最新の情報をいち早く入手したいという方は、是非、下記のフォームから会員登録してください。</p>
    <label for="name">メールアドレス登録：</label>
    <input type="text" v-model="name" />
  </div>
</template>
```

図：指定されたコンポーネントを表示



a. コンポーネントを表示する

`<component>`要素の用法はカンタン、is属性で表示すべきコンポーネントを指定するだけです。この例であれば、Ref変数selectedの値――つまり、選択ボックスの選択値によって、表示するコンポーネントを決定しています。

細かい点ですが、is属性（<component>要素）、value属性（<option>要素）に渡しているのは、文字列ではなく、あらかじめインポートされたコンポーネントそのものです。よって、単なるis、valueではなく、v-bind:is、v-bind:valueである点に注意してください。

b. 浅いRef変数を作成する

コンポーネントのような比較的大きなオブジェクトを、そのままRef変数にしてしまうのは望ましくありません。というのも、Vue側が、Ref変数配下のプロパティ値まで監視しなければならないため、動作のオーバーヘッドが大きくなってしまうのです。そこで、Ref変数に登録されたオブジェクトそのものの置き換えだけを監視するのが「浅い」Ref変数です。

コンポーネントに限らず、大きなオブジェクトをRef化する場合、それで問題なければ、浅いRef変数を利用することをお勧めします。浅いRef変数を生成するには、ref関数の代わりにshallowRef関数を利用します。

c. 深いスタイルを定義する

2.2.4項でも触れたように、<style scoped>要素は現在のコンポーネントでのみ利用可能なスタイルを表します。現在の、とは、配下のコンポーネントには適用**されない**ということです。

もしも配下にもスタイルを適用したい場合には、:deep擬似セレクターを利用してください。:deepは<style scoped>要素配下でのみ利用できるセレクターで、（現在のコンポーネントだけでなく）配下のコンポーネントにまで波及するスタイル

を設定できます。この例であれば、<component>要素によって呼び出されたコンポーネントのスタイルを、MetaComp.vueで一括して宣言しています。

> **［Note］特殊なセレクター**
>
> その他にも、<style scoped>要素の配下では、以下のような特殊セレクターを利用できます。
>
> - :slotted：スロットに対して適用されるスタイル
> - :global：グローバルに適用されるスタイル

9.1.2 コンポーネントの状態を維持する - <KeepAlive>要素

前節の例で、［会員登録］メニューからメールアドレスを入力して、他のメニューに移動した後、再び［会員登録］メニューに戻ってみましょう。最初に入力した内容は消えてしまっているはずです。動的コンポーネントでは、コンポーネント切り替え時に、以前のコンポーネントを破棄するからです。

不要なリソースを残しておかないのは一般的には正しいことですが、このような例では不便です。そこで非表示になったコンポーネントを維持するために利用できるのが、<KeepAlive>要素です。先ほどの例を、以下のように書き換えてみましょう。

［リスト］MetaComp.vue

```
<KeepAlive>
  <component v-bind:is="selected" />
</KeepAlive>
```

この状態で、再度サンプルを実行して、先ほどと同じ手順を取ってみます。メニュー行き来の前後で、入力内容が消えない（＝状態が維持される）ことが確認できます。

補足：<KeepAlive>要素の属性

<KeepAlive>要素を使うからと言って、すべてのコンポーネントを無条件に維持するのはリソースの観点からも得策ではありません。そこで、以下のような属性を利用できます。

- max：維持するコンポーネントの最大数
- include：維持するコンポーネント（名前の配列）
- exclude：維持しないコンポーネント（名前の配列）

ただし、include／exclude属性の場合には、コンポーネント本体の名前（nameオプション）を明示しておく必要があります。

［リスト］BannerMail.vue

```
<script>
export default {
  name: 'BannerMail'
}
</script>
```

［リスト］MetaComp.vue

```
<KeepAlive v-bind:include="[ 'BannerMail' ]">
  <component v-bind:is="selected" />
</KeepAlive>
```

　この例では配列として指定していますが、対象がひとつの場合は文字列でも構いません（文字列なので、v-bindも不要です）。

```
<KeepAlive include="BannerMail">
```

9.2 アニメーション機能を実装する - <Transition>要素

<Transition>要素を利用することで、要素の追加／削除に際して、たとえば要素をスライドイン／アウト、フェードイン／アウトさせるようなアニメーションを追加できます。具体的には、以下のような機能との連携が可能です。

- 条件付きの描画（v-if、v-show）
- リストへの追加／削除（v-for）
- キー値の変化（key属性）
- コンポーネントの切り替え（<component>要素）

9.2.1 アニメーションの基本

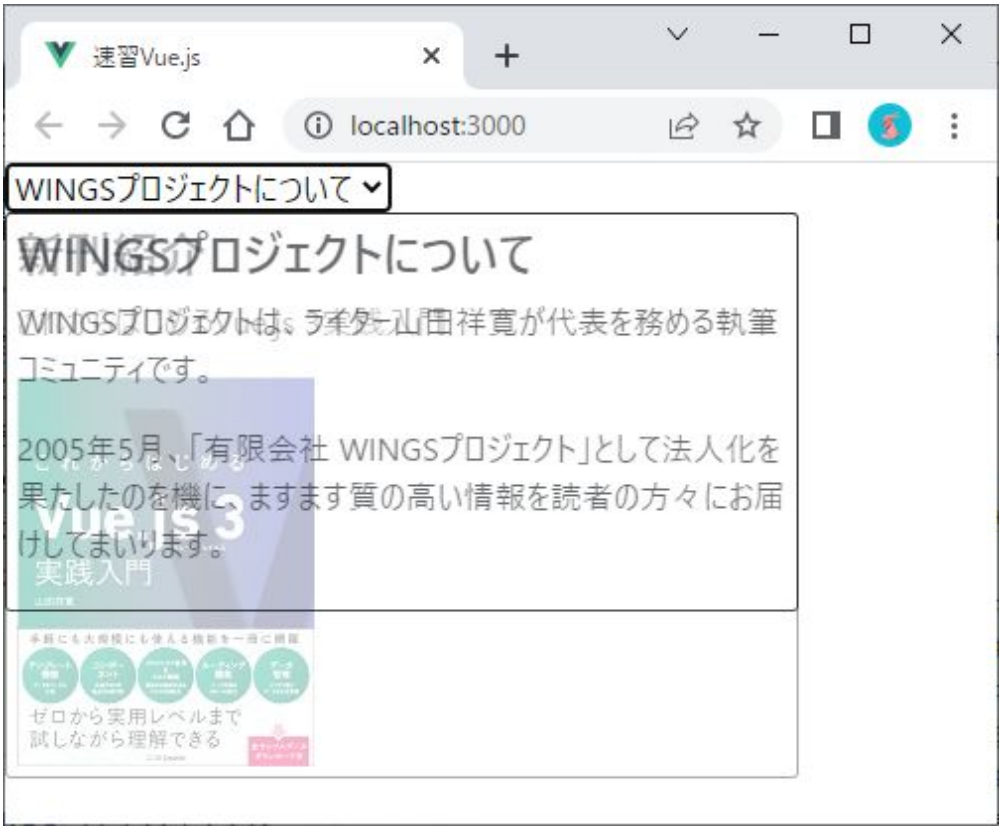
早速、具体的な例を見てみましょう。以下は、9.1.1項のサンプルを修正して、コンポーネントを切り替える際に、フェード効果を適用しています。

［リスト］MetaComp.vue

```
<template>
 <form>
  …中略…
  <!--a. アニメーションを有効化 -->
  <Transition>
   <KeepAlive include="BannerMail">
    <component v-bind:is="selected" />
   </KeepAlive>
```

```
    </Transition>
  </form>
</template>

<style scoped>
…中略…
/* c. アニメーション全体の設定 */
 .v-enter-active, .v-leave-active {
  transition: opacity 3s;
}

.v-leave-active {
  position: absolute;
}

/* b. アニメーション前後のスタイルを設定 */
 .v-enter-from, .v-leave-to {
  opacity: 0.0;
}

.v-enter-to, .v-leave-from {
  opacity: 1.0;
}
</style>
```

図：徐々にコンテンツが切り替わる



a. アニメーションを有効化する

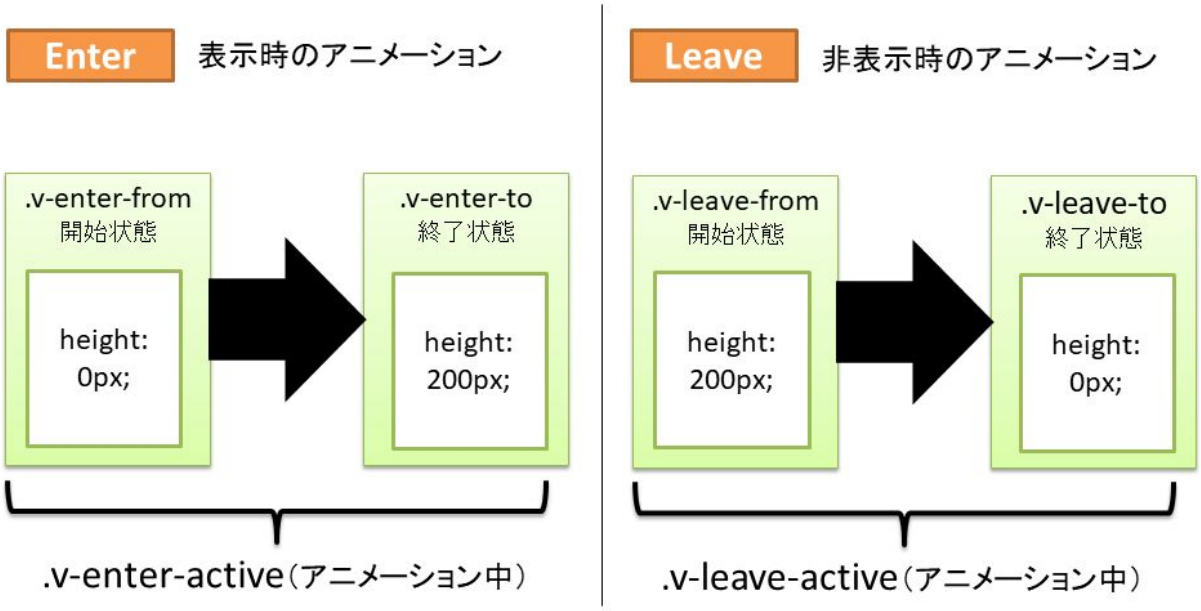
アニメーションを有効にするには、対象の要素を`<Transition>`要素で括るだけです。この例であれば、`<component>`要素――正しくは、その状態を維持するための`<KeepAlive>`要素が、その対象です。

`<Transition>`要素の直下は、単一の要素でなければならない点にも注意です。複数の要素を束ねる場合には、代わりに`<TransitionGroup>`要素を利用します（詳しくは次項で紹介します）。

b.～c. アニメーションの方法を定義する

<transition>直下の要素には、以下のタイミングでスタイルクラスが付与されます。

図：アニメーションのためのスタイルクラス



このうち、アニメーション前後の状態を表すのがv-enter-from／v-enter-to、v-leave-from／v-leave-toです。b.であれば、v-enter-from（表示前）にopacity（透明度）を1.0（不透明）に、v-enter-leave（表示後）に0.0（透明）にすることで、フェードイン効果を表現しています。v-leave-from／v-leave-toはその逆です。

ただし、これだけではアニメーションの開始／終了の状態を宣言しただけで、どのように変化するのかは決まりません。これを宣言するのがtransitionプロパティです（c.）。

> **［構文］transitionプロパティ**
>
> transition [*prop*] [*dur*] [*func*] [*delay*]
>
> - prop：アニメーションの対象となるプロパティ（複数はカンマ区切り）
> - dur：変化にかかる時間
> - func：変化の方法（ease、linear、ease-out、ease-inなど）
> - delay：開始タイミング

この例であれば、opacityプロパティを3秒かけて変化させるという意味になります。

> **［Note］省略可能なスタイル**
>
> opecityの既定値は1.0なので、「.v-enter-to, .v-leave-from { … }」は省略しても挙動には影響しません。

9.2.2 `<transition>`要素の主な属性

`<transition>`要素には、アニメーションの方法を制御するために、以下のような属性が用意されています。

- name：アニメーションの名前
- appear：初期表示時にアニメーションを再生するか[2]
- mode：遷移モード（out-in値で前の要素が非表示になった後、新しい要素を表示）

name属性は、ひとつのページに複数のアニメーションを同居させたい場合に、互いを識別するために用います。そして、name属性を指定した場合には、スタイル側のクラス名も変化するので注意してください。たとえば`<Transition name="hoge">`とした場合には、スタイルも以下のように変化します。

```
.hoge-enter-active, .hoge-leave-active { ... }

.hoge-leave-active { ... }
```

これまで利用していた「v-」は、名前のないアニメーションの既定の接頭辞であったわけです。

9.2.3 複数要素を対象とするアニメーション

単一要素を対象とする`<transition>`要素に対して、v-forリストのように複数要素を対象としたアニメーションを実装するならば、`<TransitionGroup>`要素を利用します。たとえば以下は、v-forリストに対して項目を追加する際に、フェードイン効果を適用する例です。スタイル設定は9.2.1項のものと同じなので、本項では割愛します。

［リスト］EffectList.vue

```
<script setup>
import { ref } from 'vue'
const newitem = ref('')
// Todoリストの初期値
const list = ref([
  'メールニュースの配信',
```

```
  '経理の月末処理',
  '粗大ごみの回収',
  '読者質問への対応',
  'レビューデータのバックアップ'
])
// ［追加］ボタンで項目を追加
const onclick = () => {
  list.value.push(newitem.value)
  newitem.value = ''
}
</script>

<template>
  <form>
    <input type="text" v-model="newitem" />
    <input type="button" value="追加" v-on:click="onclick" />
  </form>
  <!--リストにアニメーションを適用-->
  <TransitionGroup tag="ul">
    <li v-for="item in list" v-bind:key="item">{{ item }}</li>
  </TransitionGroup>
</template>
```

図：ボタンクリックで追加項目をフェードイン

`<TransitionGroup>`要素も、`<Transition>`要素と同じく、アニメーション対象の要素を括るだけです [3]。ただし、複数要素のコンテナーとなりうるので、tag属性を指定することで、実際にタグを出力できる点が異なります。この例であれば、`<li>`要素を括るために`<ul>`要素を出力しています。

9.3 テンプレート配下のコンテンツを任意の場所に反映させる - <Teleport>要素

<Teleport>要素を利用することで、テンプレート（＝コンポーネントによる出力）の一部を、ページ内の任意の場所に移動（teleport）できます。たとえばアラート、モーダルダイアログのような――コンポーネントの外で記述している要素に対して、コンポーネントの結果を反映させるような状況で活用できるでしょう。

以下は、その具体的な例です。TeleportBasicコンポーネントでは、テンプレートの一部を「id="result"である要素」に反映させています [4]。

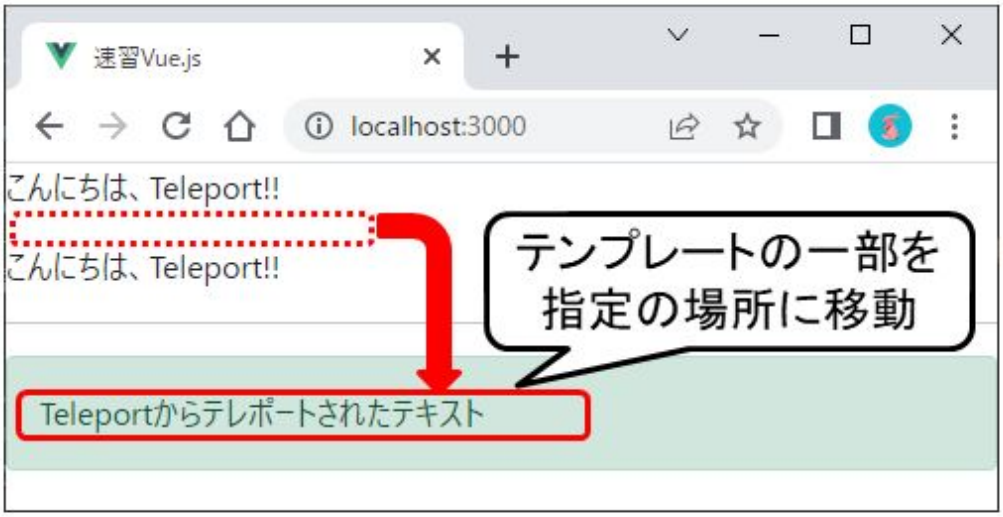
［リスト］TeleportBasic.vue

```
<template>
  <div class="jumbotron">
    <p>こんにちは、Teleport!!</p>
    <!--a. 移動させるテキストを定義 -->
    <Teleport to="#result">
      Teleportからテレポートされたテキスト
    </Teleport>
    <div>こんにちは、Teleport!!</div>
  </div>
  <hr />
</template>
```

［リスト］index.html

```
<body>
  <div id="app"></div>
  <!--b. コンポーネントからのテレポート先-->
  <div id="result" class="alert alert-success"></div>
  <script type="module" src="/src/main.js"></script>
</body>
```

図：<Teleport>要素の内容を指定の要素に移動



テレポート機能を利用するには、a.のように、移動させるコンテンツを<Teleport>要素で括り、to属性で移動先を指定するだけです。移動先（ターゲット）はセレクター式として表します。この例であれば、#resultとしているので、id="result"である要素に<Teleport>要素の内容が反映されます（b.）。

［Note］理想的なテレポート先

テレポート先は、一般的には、Vueで管理された範囲（この例であれば「id="app"である要素」）**の外**であることが理想です。というのも、`<Teleport>`呼び出しのタイミングでターゲットが特定できなければならないからです。たとえば`<div id="result">`要素をTeleportBasicコンポーネントの配下に移動した場合、「Failed to locate Teleport target with selector "#result".〜」のような警告が発生します。

9.4 非同期処理の待ちメッセージを表示する - <Suspense>要素

コンポーネントによっては、コンテンツを準備するために外部サービスへの非同期通信が発生することはよくあります。このような場合、なんらかのローディングメッセージを表示させた方がユーザーにも親切です。これをタグだけで制御するのが、<Suspense>要素の役割です。

> **［Note］現時点では実験的API**
>
> <suspense>要素は、執筆時点（Vue 3.2）では実験的APIという扱いです。将来的には、用法などが変化する可能性があります。

具体的な例も見てみましょう。MyHeavyコンポーネントはコンテンツのロードまでに時間がかかることを模した非同期コンポーネントです。よって、MyHeavyがロード完了するまで、ローディングメッセージを表示します。

［リスト］SuspenseBasic.vue

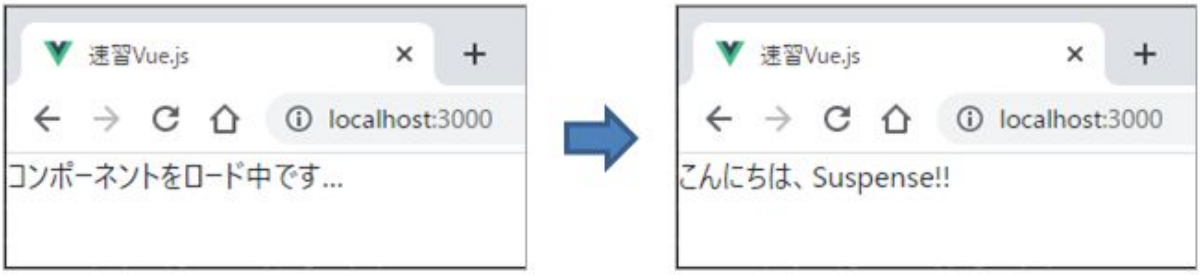
```
<script setup>
import MyHeavy from './MyHeavy.vue'
</script>

<template>
  <!--a. 非同期コンポーネントの表示-->
  <Suspense>
```

```
    <template v-slot:default>
      <MyHeavy />
    </template>
    <template v-slot:fallback>
      コンポーネントをロード中です…
    </template>
  </Suspense>
</template>
```

［リスト］MyHeavy.vue

```
<script setup>
import { ref } from 'vue'
// b. 非同期処理（時間の掛かる処理）
const msg = await new Promise((resolve) => {
  setTimeout(() => { resolve() }, 5000)
})
 .then(() => ref('こんにちは、Suspense!!'))
</script>

<template>
  <div>{{ msg }}</div>
</template>
```

図：ロード完了で本来のコンテンツを表示



`<Suspense>`要素（a.）では、配下に2個のテンプレートを配置します（v-slotで命名します [5]）。

- dafault：本来のコンテンツ（非同期ロードするもの）
- fallback：default待ちの時に表示するコンテンツ

これでdefaultテンプレートの描画待ちの間はfallbackテンプレートが表示され、defaultテンプレートの準備ができたところで表示が切り替わります。ここでは、defaultテンプレート配下に非同期コンポーネントが1つあるだけですが、複数配置することも可能です。その場合には、すべての準備が完了したところで、表示が切り替わります。

補足：非同期コンポーネント

非同期に実行される処理には、await演算子を付与するだけです（b.）。これで、コンポーネント全体としても非同期リソース（Ref変数）が揃うのを待って描画されます。

b.の処理は、setupオプションを利用して、以下のように表しても同じ意味です[6]。

```
<script>
import { ref } from 'vue'
export default {
  async setup() {
    const msg = await new Promise((resolve) => {
      setTimeout(() => { resolve() }, 5000)
    })
    .then(() => ref('こんにちは、Suspense!!'))
    // 用意できたRef変数を束ねる
    return {
      msg
    }
  }
}
</script>
```

この例では、Promiseオブジェクトの中でsetTimeout関数で5000ミリ秒の遅延を設けていますが、一般的にはfetchメソッドなどで非同期にリソースにアクセスするなどの処理を挟むことになるでしょう。Promise／fetchについては、拙著「速習ECMAScript 2020」（Amazon Kindle）を参照してください。

---

1. 正しくは、<component>、<slot>要素はコンポーネントライクなテンプレート構文で、真のコンポーネントではありません。しかし、コンポーネント的な構文で利用できることから、ここでまとめて紹介しておきます。↩

2. 論理属性なので、<Transition appear>のように属性名のみで表記します。↩

3. 利用できる属性もmodeが利用できないほかは、<Transition>要素とほぼ同じです。↵

4. 他のサンプルへの影響を防ぐため、ダウンロードサンプル上、index.htmlの 要素はコメントアウトしています。実行に際してはコメントを解除してください。↵

5. 省略形で、<template #default>のように表しても構いません。↵

6. setupオプションについては、別冊「速習 Vue.js 3」（Amazon Kindle）を参照してください。↵

Part 10：ディレクティブ／プラグイン

本Partでは、以下のような部品化技術の基本的な用法について学びます。

- ディレクティブ：属性の形式で文書ツリーを操作するしくみ
- プラグイン：Vueインスタンスを拡張するためのしくみ

［Note］フィルター

Vue 2.xまではフィルターと呼ばれるしくみがありました。フィルターとは、テンプレート上に埋め込まれたデータを加工するためのしくみで、{{ str | uppercase }}のような形式で用います（この例であれば文字列strを大文字に変換します）。

ただし、フィルターのしくみはVue 3で廃止になっています。今後は、算出プロパティ、メソッドで代用してください。

10.1 ディレクティブの自作

Part 4～7でも触れたように、Vueでは標準でさまざまなディレクティブが用意されており、文書ツリーを限りなくコーディングレスで操作できるようになっています。もっとも、本格的なアプリを開発するようになると、標準ディレクティブだけでは事足りない局面も出てきます。そのような場合には、ディレクティブそのものを自作することもできます。

10.1.1 ディレクティブの基本

まずは、ごくシンプルなディレクティブとして、現在の要素に対して指定された背景色を付与するv-highlightを定義してみます。

図：v-highlightディレクティブで指定された色で着色

［1］ディレクティブを呼び出す

順番は前後しますが、まずは用法をイメージする方が判りやすいので、呼び出し側のコードから見ていきます（もちろん、これだけでは動作しません）。

［リスト］DirectiveBasic.vue

```
<script setup>
import { ref } from 'vue'
const color = ref('Yellow')
</script>

<template>
  <!--v-highlightディレクティブを呼び出し-->
    <p>JavaScriptフレームワークの代表として、Angular、React、<span v-highlight="color">Vue.js</span>などが挙げられます。</p>
</template>
```

自作ディレクティブだからと言って、なんらこれまでと変わるところはありません。属性の形式での呼び出しが可能です（太字）。

［2］ディレクティブを準備する

では、v-highlightディレクティブ本体の実装を見ていきましょう。ディレクティブは、/srcフォルダー配下に新たに/directivesフォルダーを作成し、その配下に保存しておきましょう。慣例的に「v＋名前.js」のように命名します。

［リスト］vHighlight.js

```
export default {
  mounted(el, binding, vnode, oldVnode) {
    el.style.backgroundColor = binding.value
  }
}
```

ディレクティブは「export default { ... }」ブロック（＝オブジェクトリテラル）として定義します。配下で定義するのは、以下のような**フック関数**です。フック関数とは、決められたタイミングで呼び出される関数のこと。これらフック関数の組み合わせで、文書ツリーを操作していくわけです。

- created：ディレクティブが生成された時
- beforeMount：親コンポーネントがマウントされる前
- mounted：親コンポーネントがマウントされた後
- beforeUpdate：親コンポーネントが更新される前
- updated：親コンポーネント（と、配下のコンポーネント）が更新された後
- beforeUnmount：親コンポーネントがマウントされる前
- unmounted：親コンポーネントがマウントされた後

たとえばディレクティブの挙動を初期化するならば、まずはmountedを利用すれば良いでしょう。フック関数が受け取る引数の意味は、以下の通りです。

- el：ディレクティブが適用された要素
- binding：バインド情報オブジェクト（具体的なプロパティは以下）
- vnode：現在の仮想ノード（Vueが内部的に生成するノードオブジェクト）
- oldVnode：変更前の仮想ノード（beforeUpdate／updatedでのみ利用可）

いずれの引数も省略可能なので、この例であれば「mounted(el, binding)」と書いても同じ意味です。また、引数elを除くすべてのプロパティは読み取り専用として扱わなければなりません（フック関数の中で変更してはいけません）。

引数bindingで利用できるプロパティには、以下のようなものがあります。

- instance：ディレクティブが配置されたコンポーネントのインスタンス
- value：ディレクティブに渡された値（「v-mydir="2 + 3"」ならば「5」）
- oldValue：変更前の値（beforeUpdate／updatedでのみ利用可）
- arg：引数（「v-mydir:hoge」ならば「hoge」）
- modifiers：修飾子（「v-mydir.hoge.piyo」ならば{ hoge: true, piyo:true }）
- dir：ディレクティブの定義オブジェクト（directiveメソッドの第2引数に渡されたもの）

この例であれば、引数el経由で、ディレクティブが紐づいた要素のスタイル（style.backgroundColor）にアクセスし、引数binding経由で取得した設定値（binding.value）を設定しています。

［3］ディレクティブをアプリに登録する

定義されたディレクティブをアプリ全体で有効にするならば、directiveメソッドを利用します。

［リスト］main.js

```
import vHighlight from './directives/vHighlight.js'
...中略...
const app = createApp(DirectiveBasic)
// ディレクティブを有効化
app.directive('highlight', vHighlight)
```

```
...中略...
app.mount('#app')
```

directiveメソッドの構文は、以下の通りです。名前は接頭辞（v-）を除いた部分を指定します。

［構文］directiveメソッド

directive(*name*, *def*)

- name：ディレクティブの名前
- def：動作の定義

［Note］特定のコンポーネントだけで有効にする

アプリ全体ではなく、特定のコンポーネントだけでディレクティブを有効にするならば、directiveメソッドをコメントアウトした上で、該当のコンポーネント（ここでは、DirectiveBasic.vue）に、以下のコードを追加してください[1]。

```
<script setup>
import vHighlight from '@/directives/vHighlight.js'
...中略...
</script>
```

インポート名は、ファイル名と同じく「v＋名前」とします。これで「v-名前="..."」の形式でディレクティブを呼び出せるようになります。

10.1.2 親コンポーネントの監視

以下は、前項のサンプルに色選択のための選択ボックスを追加した例です。選択ボックスで指定した色に応じて、ハイライトカラーを変化させてみます。

［リスト］DirectiveChange.vue

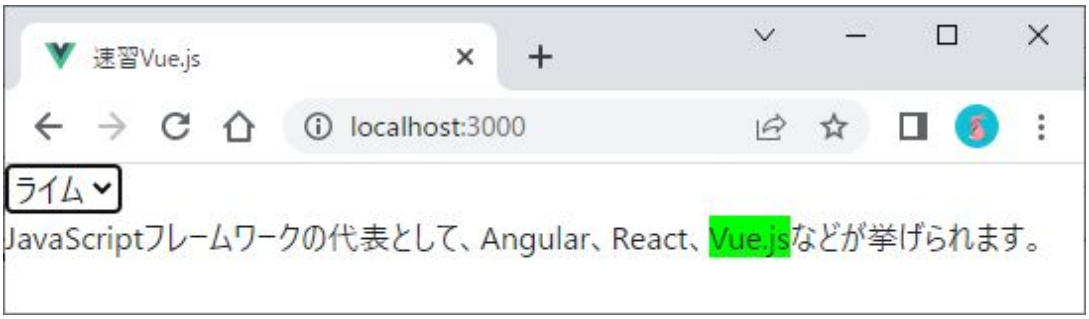
```
<script setup>
import { ref } from 'vue'
const color = ref('Yellow')
</script>

<template>
  <select v-model="color">
    <option value="Yellow">黄</option>
    <option value="Black">黒</option>
    <option value="Lime">ライム</option>
  </select>
    <p>JavaScriptフレームワークの代表として、Angular、React、<span v-highlight="color">Vue.js</span>などが挙げられます。</p>
</template>
```

［リスト］vHighlight.js

```
export default {
  // a. マウント時に実行（初回のみ）
  mounted(el, binding, vnode, oldVnode) {
    el.style.backgroundColor = binding.value
  },
  // 親コンポーネントが変化した時
```

```
  updated(el, binding, vnode, oldVnode) {
    el.style.backgroundColor = binding.value
  },
}
```

図：選択ボックスの値に応じてハイライトカラーを変更



mountedフックに加えて、updatedフックが指定されている点に注目です。mounted（a.）は、あくまでディレクティブが要素に紐づいた最初の1回だけ実行されるフック関数です。よって、これだけでは選択ボックス（color）の変更は反映されません。

親コンポーネント（Ref変数／Reactive変数）の変化を検知するには、updatedフック（b.）を利用します。ちなみに、updatedフックだけ（＝mountedフックなし）でも不可である点に注意です。mountedフックがない場合、初期化が行われないため、選択ボックスを変化させるまでハイライトが反映されません。

補足：mounted／updatedをまとめて定義する

初期化（mounted）、更新（updated）の組み合わせはよく利用することから、省略構文も利用できます。vHighlight.js全体を以下のように書き換えてみましょう。

［リスト］vHighlight.js

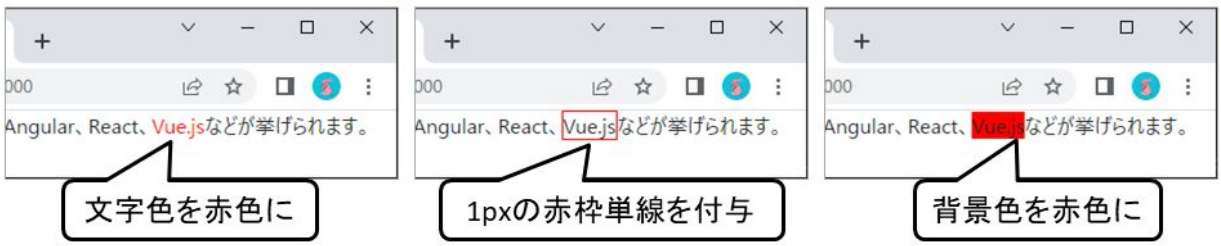
```
export default function(el, binding, vnode, oldVnode) {
  el.style.backgroundColor = binding.value
}
```

mounted／updatedフックを備えたオブジェクトリテラルの代わりに、関数リテラルを定義するわけです。mounted／updatedフックで同じ内容を表している場合には、このような形でコードを簡単化できます。

10.1.3 引数付きのディレクティブ

v-on、v-bindなどと同じく、自作のディレクティブでも引数を指定することが可能です。たとえば以下は、v-highlightを修正し、ハイライトすべき対象を引数で指定できるようにした例です。指定可能な値はbackground（背景）、text（文字色）です。それ以外の値が指定された、または引数が省略された場合には、枠線でハイライトします。

図：引数値によってスタイルを変更



以下に手順を見ていきましょう。

［1］ディレクティブを登録する

新たにvArgedHighlight.jsを作成し、/src/directivesフォルダーに保存します（簡単化のため、updatedフックの記述は省略します）。

［リスト］vArgedHighlight.js

```
export default {
  mounted(el, binding, vnode, oldVnode) {
    // 引数の値に応じてスタイルを操作
    switch(binding.arg) {
      case 'background' :
        el.style.backgroundColor = binding.value
        break
      case 'text' :
        el.style.color = binding.value
        break
      default :
        el.style.border = `1px solid ${binding.value}`
        break
    }
  }
}
```

［リスト］main.js

```
import vArgedHighlight from './directives/vArgedHighlight.js'
…中略…
```

```
const app = createApp(DirectiveArgs)
app.directive('argedHighlight', vArgedHighlight)
```

引数値は、フック関数の引数bindingからargプロパティ経由で取得できます。引数値は文字列なので、あとはその値に従って、処理を分岐するだけです。

［Note］修飾子を取得する

同様に、binding.modifiersプロパティ経由で修飾子を取得することもできます。「v-mydir.hoge.piyo」とした場合、modifiersプロパティの戻り値は{ hoge: true, piyo:true }となります。

その性質上、引数はディレクティブの挙動を左右するキーとなる情報に適しているのに対して、修飾子はオプショナルなオン／オフを表すような情報を渡すのに適しています。

［2］ディレクティブを呼び出す

.vueファイルから、実際にディレクティブを呼び出してみましょう。

［リスト］DirectiveArgs.vue

```
<script setup>
import { ref } from 'vue'
const color = ref('Red')
</script>

<template>
  <p>JavaScriptフレームワークの代表として、Angular、React、<span v-arged-
```

```
highlight:text="color">Vue.js</span>などが挙げられます。</p>
</template>
```

太字の部分を「v-arged-highlight:background」「v-arged-highlight（引数なし）」のように変化させることで、結果も変化することを確認してください。

10.1.4 イベント処理を伴うディレクティブ

ディレクティブでは、イベントハンドラーを設定することも可能です。これにはフック関数の中で、引数elを介してaddEventListenerメソッドを呼び出すだけです。

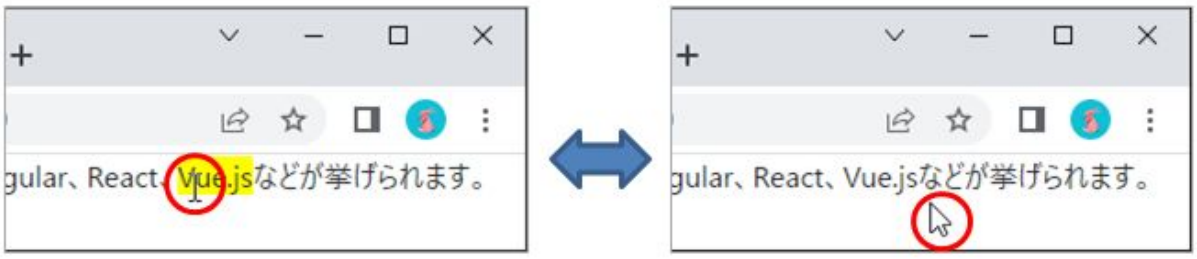
たとえば以下は、先ほど作成したv-highlightディレクティブを修正して、マウスポインターを要素に当てると背景色を付与し、外れると背景色を戻すようにした例です（.vueファイルはほぼこれまでと同じなので、紙面上は割愛します。呼び出しのコードはDirectiveEvent.vueを参照してください）。

［リスト］vEventHighlight.vue

```
export default {
  mounted(el, binding, vnode, oldVnode) {
    // mouseenter時のイベント処理を定義
    el.addEventListener('mouseenter', function() {
      el.style.backgroundColor = binding.value
    }, false)
    // mouseleave時のイベント処理を定義
    el.addEventListener('mouseleave', function() {
      el.style.backgroundColor = ''
```

```
    }, false)
  },
}
```

図：マウスポインターの出入りに応じて背景色をオンオフ



10.2 プラグイン

プラグインとは、Vueアプリに対してディレクティブ、メソッドなどを追加（拡張）するための、標準的なしくみです。作成したディレクティブなどを不特定多数に配布したいという場合には、プラグインのルールに則って記述することをお勧めします（組み込みの手順が標準化されるので、導入がスムーズになります）。

> **［Note］プラグイン**
>
> たとえば本書後半で解説するVue Router、Piniaもプラグインの一種です。その他にも、Vueに対応したプラグインは豊富に用意されているので、以下のようなページから探してみると良いでしょう。
>
> - Awesome Vue.js
>
> 現時点ではVue 3に対応したものは限られますが、今後、Vue 3の普及に伴って、周辺ライブラリも徐々に対応してくるはずです。

10.2.1 プラグインの基本

では、ここでは前節で作成したv-highlightプラグインをプラグイン化してみましょう（名前が衝突できないので、ここではv-plugin-highlightとリネームします）。

［1］MyUtilプラグインを定義する

まずは、プラグインの定義から見ていきます。プラグインは、/srcフォルダー配下に/pluginsフォルダーを作成し、その配下に保存するものとします。

［リスト］MyUtil.js

```
// a. プラグインを定義
export default {
  install(app, options) {
    // b. v-plugin-highlightディレクティブを定義
    app.directive('pluginHighlight', {
      mounted(el, binding, vnode, oldVnode) {
        …中略…
      },
    })
  }
}
```

プラグインを定義するには、installメソッドを持つオブジェクトを用意するだけです（a.）。installメソッドは引数として

- app：アプリインスタンス（createAppメソッドの戻り値）
- options：動作オプション（後述）

を受け取るので、この例では、app.directiveメソッドを呼び出してディレクティブを宣言します。

ディレクティブの定義そのものは、前節でも触れたとおりなので、特筆すべき点はありません（b.）。

［2］定義済みのプラグインを利用する

定義したMyUtilプラグインを、アプリに登録してみましょう。

［リスト］main.js

```
import MyUtil from './plugins/MyUtil.js'
…中略…
app.use(MyUtil)
app.mount('#app')
```

プラグインを登録するには、useメソッドでプラグインオブジェクトを呼び出すだけです。このタイミングでinstallメソッドが呼び出され、ディレクティブが登録されます。

登録されたv-plugin-highlightディレクティブを呼び出すコードは、これまでとほぼ同じなので、紙面上は割愛します。呼び出しのコードはPluginBasic.vueを参照してください。

10.2.2 動作オプションの追加

プラグインに対して、動作オプションを渡すこともできます。たとえば以下は、先ほどのMyUtilプラグインにborderオプションを追加する例です。borderオプションが有効（true）である場合、ハイライトを（背景ではなく）枠線として付与します。

図：枠線でハイライトを表現

［1］プラグインを修正する

borderオプションを受け入れるように、MyUtilプラグインを修正します。

［リスト］MyUtil.js

```
export default {
  // a. optionsの既定値を設定
  install(app, options = {}) {
    app.directive('pluginHighlight', {
      mounted(el, binding, vnode, oldVnode) {
        // b. borderオプションの有無で処理を切り替え
        if (options.border) {
          el.style.border = `1px solid ${binding.value}`
        } else {
          el.style.backgroundColor = binding.value
        }
      },
    })
  }
}
```

プラグインの動作オプションはinstallメソッドの引数optionsで受け取るのでした（a.）。ただし、呼び出し側で動作オプションが指定されなかった場合に備え

て、既定値（ここでは空オブジェクト）を用意しておくのが望ましいでしょう。

あとは、directiveメソッドの配下でoptions.borderプロパティの有無を確認して、border／backgroundColorプロパティのいずれかを設定するよう、処理を分岐するだけです（**b.**）。

［2］プラグイン呼び出しのコードを修正する

プラグインに動作オプションを渡すには、useメソッドを以下のように書き換えます。

［リスト］main.js

```
app.use(MyUtil, { border: true })
app.mount('#app')
```

useメソッドの第2引数には「オプション名: 値, ...」の形式で動作オプションを指定できます。指定できるオプションはプラグインによりますが、useメソッドの構文そのものはプラグインに関わらず共通です。同じ要領でプラグインを有効化できる共通性が、本節冒頭でも述べたように、プラグイン化するメリットです。

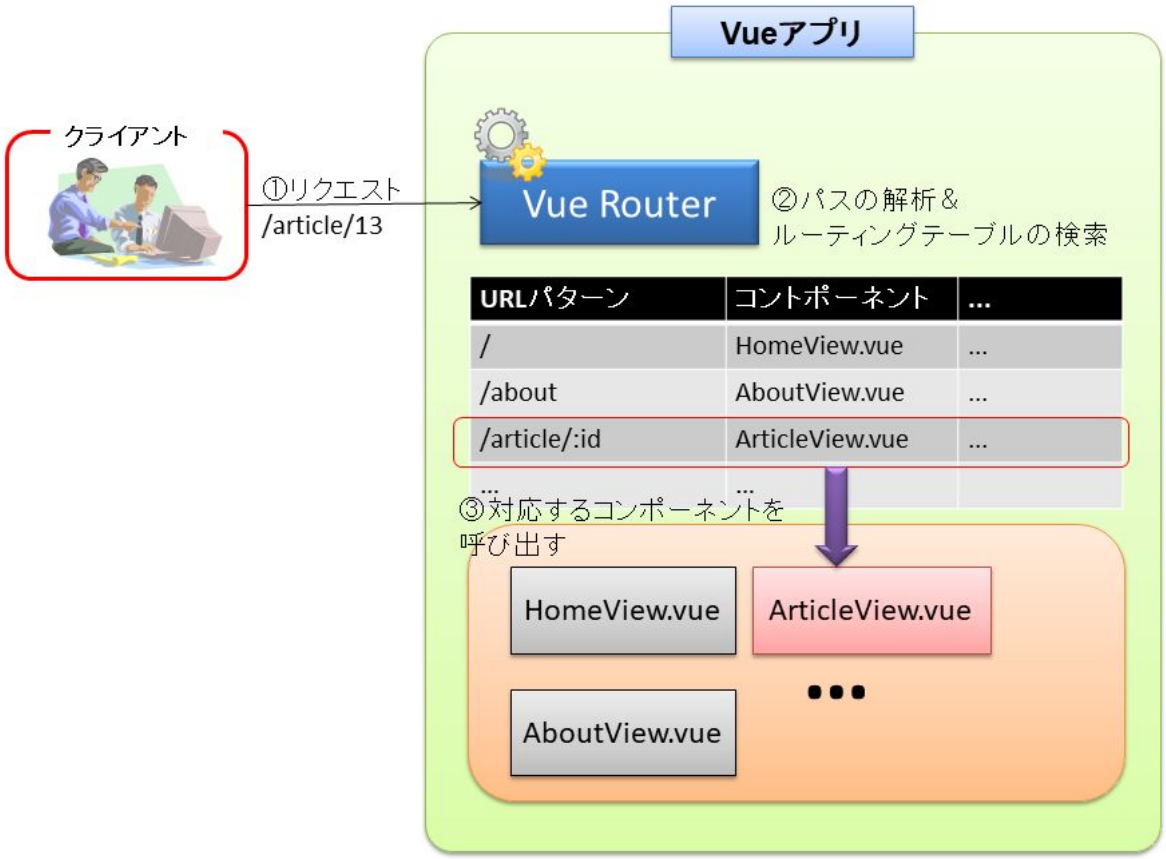
---

1. パス文字列「@/directives/vHighlight.js」に含まれる「@」は、/srcフォルダーのエイリアスです。↩

Part 11：ルーティング

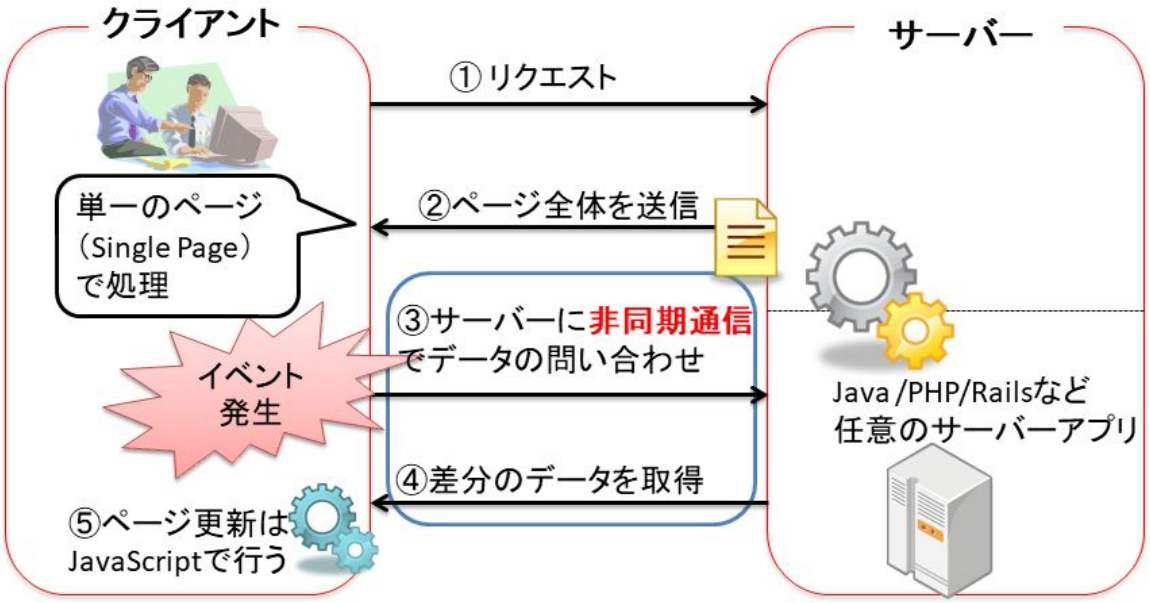
ルーティングとは、リクエストURLに応じて処理の受け渡し先（コンポーネント）を決定すること、あるいは、そのしくみのことを言います。ルーティングは、いわゆるSPA（Single Page Application）を実装する上で欠かせない仕組みです。ルーティング機能を提供するモジュールのことを**ルーター**と言います。Vueでは、標準的なルーターとして**Vue Router**というライブラリを提供しています。

図：ルーティング



［Note］SPA

SPAとは、名前の通り、単一のページで構成されるアプリのこと。初回アクセスでページ全体を取得し、以降のページ切り替えは基本的にJavaScriptで行うのが一般的です。



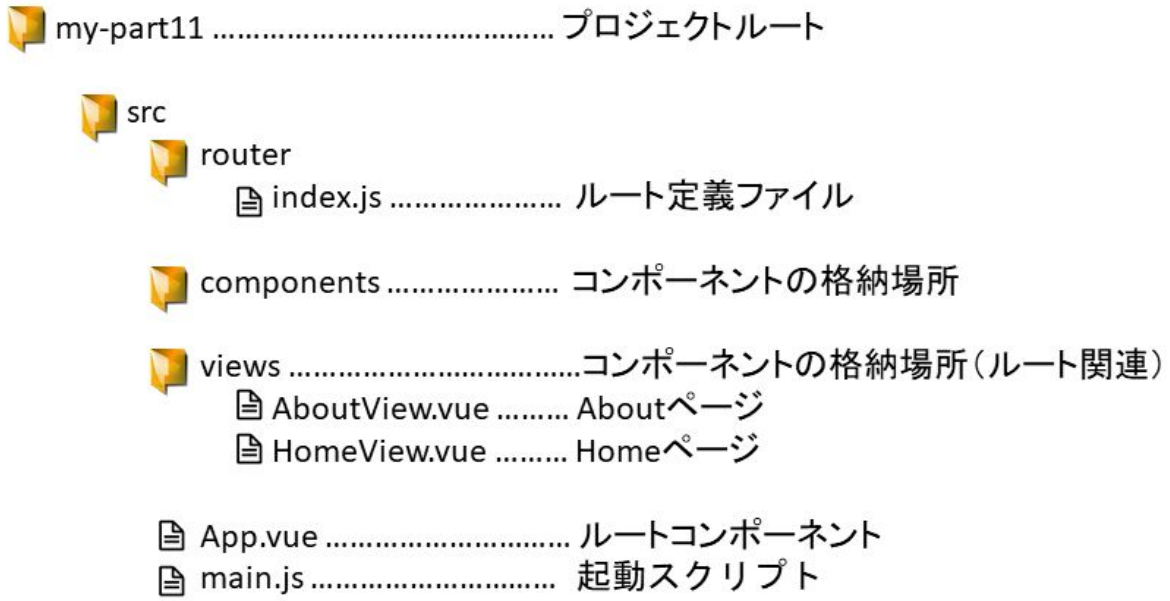
非SPAアプリでは、ページ遷移はブラウザー任せでしたが、SPAの世界ではページ遷移もアプリの責務です。ルーターは、URLを解析し、適切なコンポーネントを選択し、その処理結果をページの決められた領域に反映させるまでを担うモジュールです。

11.1 ルーターの基本

create-vueからVue Routerを利用するには、プロジェクト作成ウィザードでモジュール選択の際に「Add Vue Router for Single Page Application development?」という設問に対してYesを選択してください。

Vue Routerを有効にした場合、プロジェクトには標準的なフォルダー／ファイルに加えて、以下のようなフォルダー／ファイルが追加されます（main.js、App.vueは元からあったものですが、Vue Routerを加えることで中身も変化するので、図でも挙げています）。

図：Vue Router環境のフォルダー構造



/viewsフォルダーはルーティングに関わるコンポーネントを格納します。似た用途のフォルダーとして/componentsフォルダーもありますが、こちらはより細かなUI部品を

格納するのに利用するのが慣例です。

ただし、本書では他の章と並びを揃えるために、これまで通り、/componentsフォルダー配下に.vueファイルを配置するものとします。

11.1.1 ルーティング情報の定義

Vue Routerの準備ができたところで、ここからは自動生成されたコードをもとにルーターの基本的な構文を理解していきましょう。

まず、ルーターの実行に必要なのは、ルーティング情報――「どのアドレスに対して、どのコンポーネントを紐づけるか」という情報です。ルーティング情報を定義しているのは、/src/router/index.jsです（自動生成されたコメントは割愛しています）。

［リスト］router/index.js

```
import { createRouter, createWebHistory } from 'vue-router'
// c. コンポーネントをインポート
import HomeView from '../views/HomeView.vue'

// a. ルーターを生成
const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  // b. ルーティング情報を準備
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView
    },
```

```
    {
      path: '/about',
      name: 'about',
      // d. AboutViewコンポーネントを非同期インポート
      component: () => import('../views/AboutView.vue')
    }
  ]
})
// ルーターをエクスポート
export default router
```

ルーターを利用するには、まずcreateRouterメソッドでRouterオブジェクトを生成します（a.）。createRouterメソッドには「オプション名: 値, ...」の形式でルーターオプションを指定します。

- history：Historyモードの基本情報
- routes：ルーティング情報
- linkActiveClass：現在ページを表すリンクに適用するスタイルクラス
- linkExactActiveClass：現在ページを表すリンクに適用するスタイルクラス（完全一致）

Historyモードとは、JavaScriptのHistory APIを利用したページ遷移の手段です。Historyモードの有効／無効によって、生成されるアドレスが変化します。

- 有効：http://localhost:8080/about
- 無効：http://localhost:8080/#/about

有効時の方が自然な表記ですし、イマドキのブラウザーであればHistory APIのサポートは問題ないはずなので、まずはHistoryモードは有効化する、と覚えておきましょう。

historyオプションには、Historyモードでルーターを動作する際の基本情報を渡します。「createWebHistory(process.env.BASE_URL)」で、現在の環境の基底パス（BASE_URL）を渡すコードは、ほぼ定型と思っておいて構いません。

もうひとつ、注目すべきはroutesオプション（ルーティング情報）です。ルーター定義のコアとなる部分です（**b.**）。

ルーティング情報は「オプション名: 値, ...」形式のオブジェクト配列として表します。ひとつのオブジェクト（ルートオブジェクト）がひとつのルートを表す、というわけです。ルートオブジェクトで利用できるプロパティには、以下のようなものがあります（pathだけが必須です）。

- path：リクエストパス
- name：ルートの名前
- component：ルーティングによって呼び出されるコンポーネント
- components：ルーティングによって呼び出されるコンポーネント（複数）
- redirect：リダイレクト先のパス
- children：配下のルート定義

この例であれば、以下のようなルートを定義したことになります。

- 「/」でHomeViewコンポーネントを呼び出す（ルート名はhome）

- 「/about」でAboutViewコンポーネントを呼び出す（ルート名はabout）

コンポーネントの実体（.vueファイル）は、既定で/src/viewsフォルダーに置かれているので、あらかじめ「import コンポーネント名 from '../views/ファイル名'」のようにインポートしておきましょう（**c.**）。

> **［Note］非同期インポート**
>
> create-vueの既定の設定では、ビルドの結果、すべてのコードはindex.xxxxx.js（xxxxxは任意のハッシュ値）にバンドルされます。しかし、アプリの規模が大きくなれば、index.xxxxx.jsも肥大化し、起動時間も増加します。そこで利用頻度の少ないコンポーネントは、非同期インポートさせることで、index.xxxxx.jsからも切り離し、必要になったところで読み込めるようになります。
>
> これを行っているのが、**d.** のimport関数です。ただし、componentオプションに渡すのはimport関数そのものではなく、import関数を呼び出すための関数（() => ...）です。「component: import(...)」のようにしないよう、注意してください。

11.1.2 ルーターの有効化

定義されたルートは、main.jsでVueインスタンスに紐づけられています。

［リスト］main.js

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
```

```
const app = createApp(App)
app.use(router)
app.mount('#app')
```

useは、Vueアプリに拡張ライブラリ（プラグイン）を組み込むためのメソッドなのでした。この例であれば、router/index.jsで定義されたルーターを引き渡すだけで、ルーターは有効になります。

11.1.3 トップページのテンプレート

続いて、Vueインスタンスに紐づいたトップコンポーネント――Appコンポーネント（/src/App.vue）を確認しておきます。

［リスト］App.vue

```
<template>
  <header>
    ...中略...
    <div class="wrapper">
      <HelloWorld msg="You did it!" />
      <!-- a. 個別ページへのリンク -->
      <nav>
        <RouterLink to="/">Home</RouterLink>
        <RouterLink to="/about">About</RouterLink>
      </nav>
    </div>
  </header>
  <!-- b. リンク先の表示場所-->
  <RouterView />
</template>
```

ルーター経由のリンクを表すには、標準的なアンカータグの代わりに<RouterLink>要素を利用します（a.）。to属性でリンク先を表します [1]。

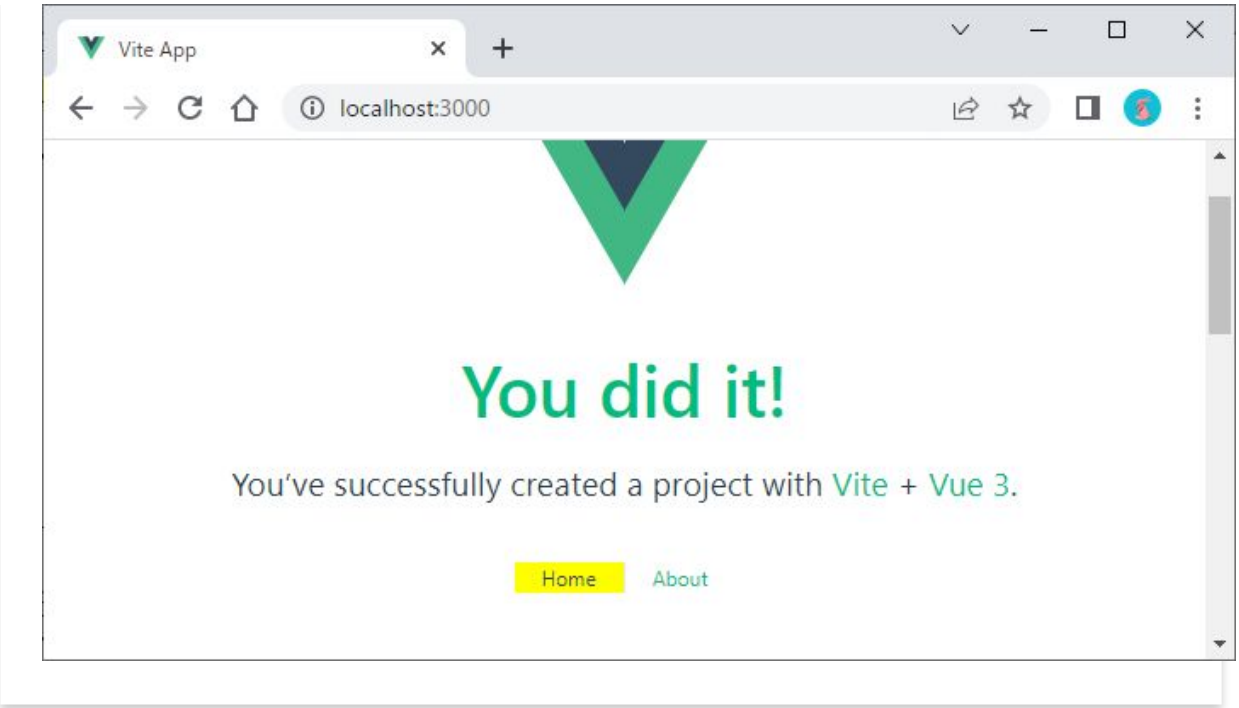
ルーター経由で呼び出されたコンポーネントは、<RouterView>要素の領域に反映されます（b.）。ルーターを利用する場合、<RouterView>要素による表示領域の確保は必須です。

［Note］アクティブリンクにスタイル指定する

<RouterLink>要素では、リンク先が現在のパスと同じである場合に、既定でrouter-link-activeというスタイルクラスをリンクに付与します。たとえば本文の例で、router-link-activeスタイルクラスを定義しておくと、以下のような結果が得られるはずです。

```
.router-link-active {
  background-color: yellow;
}
```

▼



11.1.4 補足：プログラムからページ遷移

ページを移動するには、標準的なJavaScriptであればlocation.hrefプロパティなどを利用します。しかし、これはページ全体を差し替える命令なので、ルーター環境では利用できません。ルーター経由でのページ移動には、pushメソッドを利用します。

たとえば以下は、ボタンクリック時に「/」（トップ画面）に移動する例です（サンプルは、既定で用意されている/src/views/AboutView.vueを書き換えています）。

［リスト］AboutView.vue

```
<script setup>
import { useRouter } from 'vue-router'
// a. Routerオブジェクトを取得
const router = useRouter()
// b. ボタンクリック時の処理
```

```
const onclick = () => {
  router.push('/')
}
</script>
```

useRouterは、Routerオブジェクトを取得するためのメソッドです（**a.**）。コンポーネント側からルーターを操作する際には、まずuseRouterメソッドを呼び出しておきましょう。Routerオブジェクトを取得できたら、あとは、イベントハンドラーなどからpushメソッドを呼び出すだけです（**b.**）。

11.2 パスの一部をパラメーターとして引き渡す - ルートパラメーター

たとえば「～/articles/108」「～/book/978-4-7981-5382-7」のようなパスで、コンポーネントに対して108、978-4-7981-5382-7のような値を引き渡すことができます。パラメーター値をパスの一部として表現できるため、視認性にも優れ、ルーター経由での値の引き渡しとしては、よく利用されるアプローチです。このようなパラメーターのことを**ルートパラメーター**と言います。

11.2.1 ルートパラメーターの基本

具体的な例も確認してみましょう。

［1］ルーティング情報を追加する

ルートパラメーターを受け取るには、以下のようなルートを定義します。

［リスト］router/index.js

```
import ArticleView from '../components/p11/ArticleView.vue'
const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    ...中略...
    // :idパラメーターを受け取るarticleルート
    {
      path: '/article/:id',
      name: 'article',
      component: ArticleView,
```

```
    props: true,
  }
 ]
})
```

ポイントとなるのは、pathオプションに含まれた「:名前」の表記です（この例では「:id」）。これはパラメーターの置き場所（プレイスホルダー）で、「:名前」の部分に「～/article/108」「～/article/13」のように、任意の値を埋め込めることを意味します。

また、propsオプションにtrueをセットしておきましょう。これはルートパラメーターを対応するプロパティに割り当てなさい、という意味です（この例では:idパラメーターはidプロパティに代入されます）。

ここでは、:idパラメーターをひとつだけ配置していますが、「/diary/:year/:month/:day」のように、複数のパラメーターを埋め込むことも可能です。

［2］ルートパラメーターを受け取る

ルートパラメーターを受け取るコンポーネントの例が、以下の通りです。

［リスト］ArticleView.vue

```
<script setup>
// idプロパティを宣言
defineProps({
 id: {
  type: String,
```

```
    required: true
  }
})
</script>

<template>
  <div class="article">
    <span>記事コード：{{ id }}</span>
  </div>
</template>
```

先ほども触れたように、ルートパラメーターは同名のプロパティに引き渡されるのでした。idプロパティを宣言し、テンプレート側でもプロパティを参照するようにしています[2]。

［Note］propsオプションを指定しなかった場合

ルート定義でpropsオプションを指定しなかった場合、ルートパラメーターはRouteオブジェクトのparams.idプロパティからアクセスできます。たとえば本文の例であれば、`<script>`要素配下を以下のように書き換えます。

```
import { useRoute } from 'vue-router'
const route = useRoute()
const id = route.params.id
```

useRouteは、現在のルート情報を表すRouteオブジェクトを取得するためのメソッド。useRouter（r付き）ではないので、混同しないように注意してください。

ただし、Route経由での受け渡しは、コンポーネントの再利用性という意味でも好ましくありません（ルーター以外の環境では再利用できなくなります）。

［3］リンク文字列を生成する

アプリの基点となるルートコンポーネントを用意しておきましょう。いわゆるmain.jsでcreateApp関数に渡すコンポーネントです。これまでは起動の基点となるコンポーネントが明確でしたが、ルーティングを扱うようになったことで、扱うコンポーネントが増えています。main.jsの編集にも迷わないよう、意識してください（本Partでは、以降もこのコンポーネントを再利用していきます）。

［リスト］RouteBasic.vue

```
<template>
  <nav>
    ［<RouterLink to="/">Home</RouterLink>］
    ［<RouterLink to="/about">About</RouterLink>］
    ［<RouterLink to="/article/108">Article</RouterLink>］
  </nav>
  <hr />
  <RouterView />
</template>
```

［Note］別解：to属性

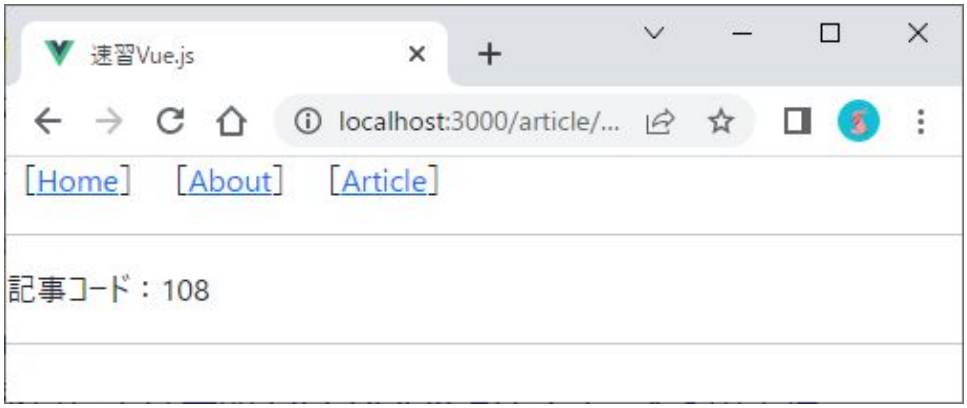
to属性（リンク先）は文字列で指定する他、オブジェクト形式による指定も可能です。

```
<RouterLink v-bind:to="{ name: 'article', params: { id: 108 }}">Article</RouterLink>
```

nameプロパティはrouter/index.js（nameオプション）で定義されたルート名、paramsプロパティはルートに引き渡すパラメーターを「名前: 値, ...」の形式で、それぞれ意味します。表記そのものは冗長になりますが、name／paramsプロパティを用いることで、呼び出しのパスが変化してもリンク側には影響が出にくくなります。

以上を理解したら、サンプルを実行してみましょう。［Article］リンクをクリックすると、記事番号が表示され、確かにルートパラメーターの情報が取得できたことを確認できます。

図：ルートパラメーター経由で渡された記事コードを表示



11.2.2 ルートの優先順位

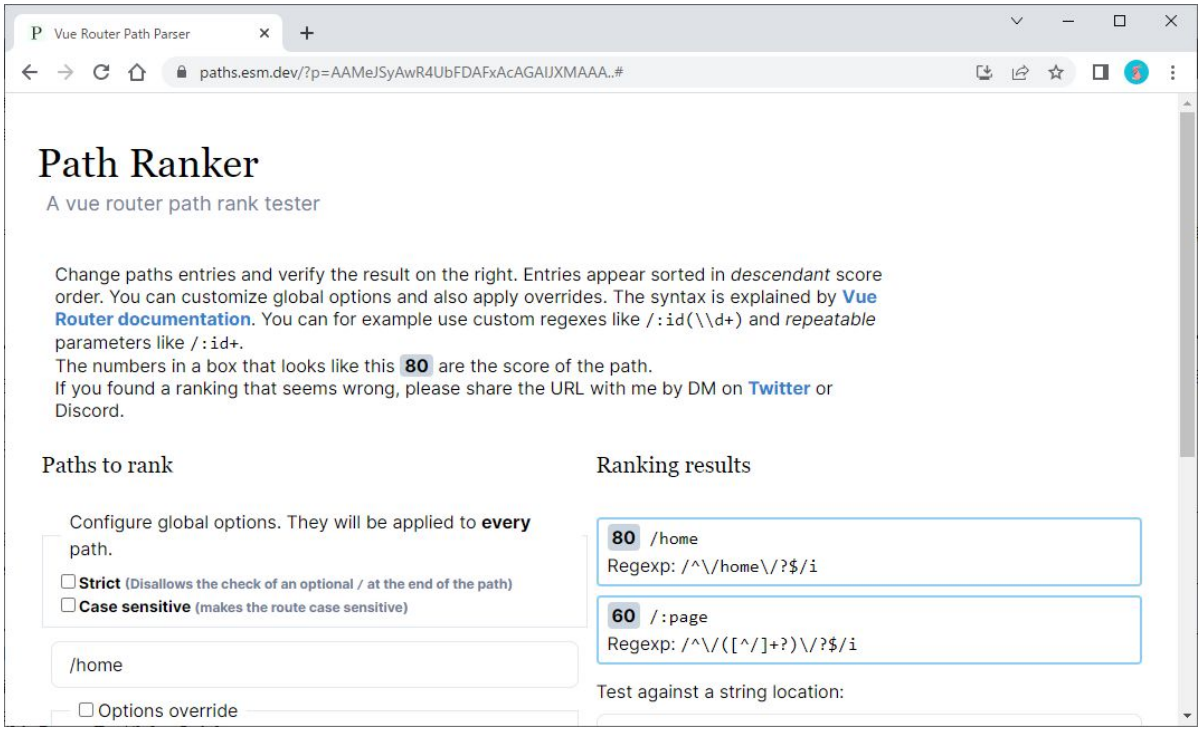
ルートパラメーターを利用するようになると、ルートそのものの優先順位も意識しておく必要があります。具体的には、ルート情報はより明示的に示されたものから優先

して適用されるからです。たとえば以下のようなルートが列記されている場合を考えてみましょう。

1. /:category/:keyword
2. /articles/:id

この場合、リテラルの入っている2. の方がより具体的にパスを特定しているので、1. よりも優先順位が高くなります。ただし、このような優先順位は、より複雑なルート定義では判別が難しくなります。そこで優先順位が曖昧になってきた場合には、以下のようなツールを利用することをお勧めします。

図：Path Ranker



Path Rankerはオンラインのツールで、左ペインからルート情報を入力すると、右側に優先順位の高い順に並び替えられた結果が表示されます。

11.2.3 ルートパラメーターの記法

ルートパラメーターの末尾に修飾子を付与することで、さまざまなパラメーターを表現できます。

（1）任意のパラメーター

パラメーターの末尾に「?」を付与することで、パラメーターを省略可能にできます（既定では必須です）。

```
/article/:id?
```

この例では「/article」（:idなし）のようなパスにもマッチします。ただし、:idパラメーターはundefinedとなるので、コンポーネント側でなんらかの既定値を用意するのが一般的です。

（2）可変長パラメーター

パラメーターの末尾に「*」を付与した場合、「/」を跨いだパスにマッチします。

```
/article/:id*
```

この例では、「/article/10」のようなパスはもちろん、「/article/javascript/vue/3」のようなパスにもマッチします。この場合、:idパラメーターの値は['javascript', 'vue', '3']のような配列となります。

パラメーター値のチェック

(pattern)の形式で、正規表現を付与することもできます。

```
/article/:id(¥¥d{1,3})
```

この場合、正規表現に合致した場合にだけルートも採用されます[3]。「¥d{2,3}」であれば、1〜3桁の数値を意味するので、たとえば「/article/108」にはマッチしますが、「/article/1024」にはマッチしません。

11.3 複数のビュー領域を設置する

Vue Routerでは、テンプレートに複数の<RouterView>要素を配置することで、複数のビューを設置することもできます。早速、具体的な手順を見ていきましょう。

［1］ルートコンポーネントを編集する

まずは、ルートコンポーネントを複数ビューに対応しておきます。

［リスト］RouteBasic.vue

```
<template>
  <nav>
    …中略…
    <!-- b. マルチビューに対応したページを追加 -->
    ［<RouterLink to="/multi/108">MultiView</RouterLink>］
  </nav>
  <hr />
  <!-- a. 複数のビューを準備 -->
  <RouterView />
  <hr />
  <RouterView name="sub" />
</template>
```

<RouteView>要素を複数配置した場合には、個々のビューを区別するために、それぞれの領域に任意の名前を付けてください（a.）。名前のないビューもひとつだけ配置できますが、その場合、名前は暗黙的にdefaultと見なされます。よって、この例であれば、default／subビューが準備されたことになります。

また、複数ビューに対応したルート（ページ）へのリンクも追加しておきます（**b.**）。

［2］ルート情報を編集する

複数のビューを設置した場合、ルート定義の側でも、複数の領域に対応してコンポーネントを割り当てられるよう、componentsパラメーター（複数形）で定義しなければなりません。

［リスト］router/index.js

```
import ArticleView from '../components/p11/ArticleView.vue'
import SubView from '../components/p11/SubView.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    ...中略...
    {
      path: '/multi/:id',
      name: 'multi',
      // a. 複数ビュー対応のコンポーネント指定
      components: {
        default: ArticleView,
        sub: SubView
      },
      // b. 複数ビュー対応のプロパティ設定
      props: {
        default: true,
        sub: false
```

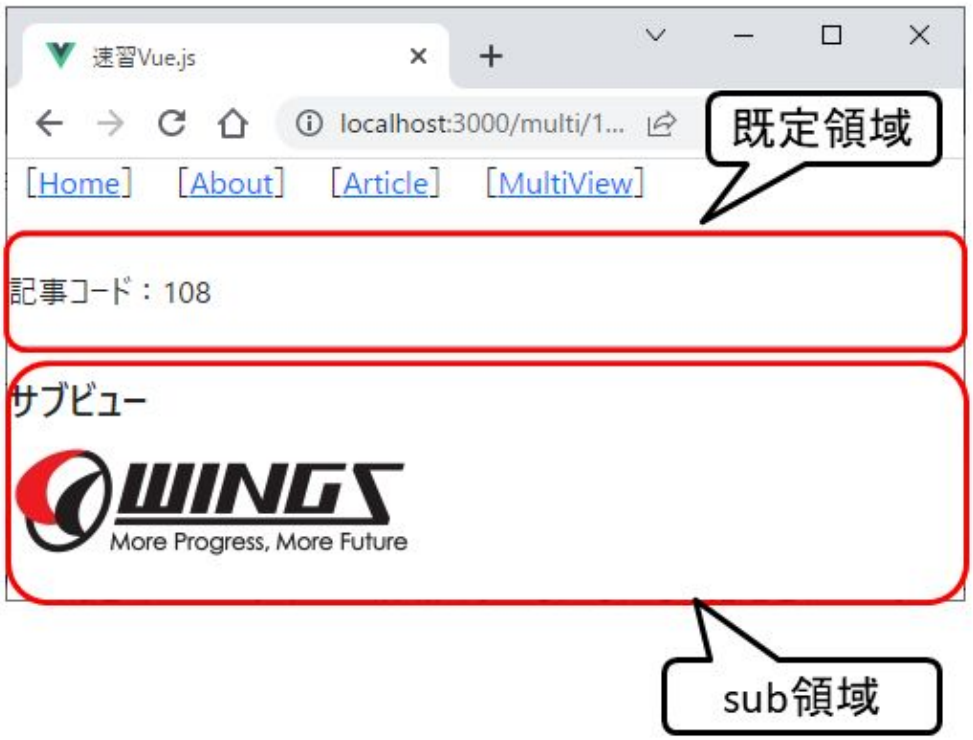
```
    }
   },
  ]
})
```

componentsパラメーターは「領域名: コンポーネント」の形式で指定します（a.）。defaultは、先ほども触れたように、<RouterView>要素のname属性を指定しなかった場合の既定の領域名です。

複数ビューにした場合、propsパラメーターについても領域ごとに設定します（b.）。この例であれば、defaultでのみルートパラメーターをプロパティとして受け取ります。

以上を理解できたら、サンプルを実行し、［MultiView］リンクにアクセスしてみましょう。確かに既定（default）、sub領域に対して、それぞれ指定されたコンポーネントが反映されていることが確認できます。

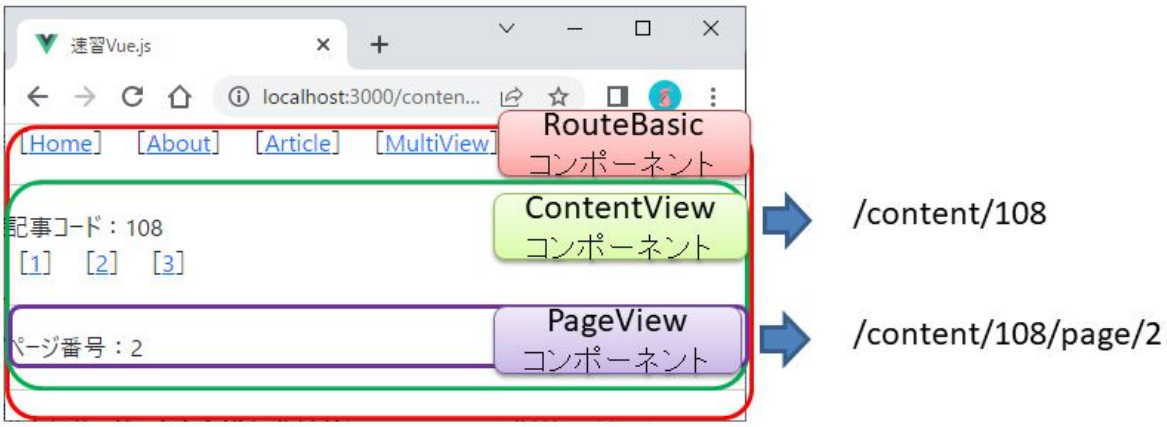
図：既定／sub領域に対してコンポーネントが反映


速習Vue.js
localhost:3000/multi/1...
既定領域
[Home] [About] [Article] [MultiView]
記事コード：108
サブビュー
WINGS
More Progress, More Future
sub領域

11.4 入れ子のビューを設置する

Vue Routerでは、ビュー同士を入れ子に配置することもできます。たとえば「/content/13」で記事のリード文を表示し、「/content/13/page/1」「/content/13/page/2」のようにすることで、それぞれ各ページの内容を表示する、といったケースです。

図：入れ子のビュー



［1］ルート情報を編集する

入れ子のルート情報はchildrenパラメーターで定義します。

［リスト］router/index.js

```
import ContentView from '../components/p11/ContentView.vue'
import PageView from '../components/p11/PageView.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
```

```
    ...中略...
    {
      path: '/content/:id',
      name: 'content',
      component: ContentView,
      props: true,
      children: [
        {
          path: 'page/:page_num',
          name: 'page',
          component: PageView,
          props: true,
        }
      ]
    },
  ]
})
```

この例であれば、「/content/:id」ルート配下に子ルート「/page/:page_num」が連なり、「/content/:id/page/:page_num」のようなパスが生成されます。

children（複数形）となっていることからも判るように、子ルートは複数列記することも可能です。

［2］入れ子のコンポーネントを準備する

ルートの準備ができたら、配下のContentView／PageViewコンポーネントも準備しておきます。

［リスト］ContentView.vue

```
<script setup>
defineProps({
  id: {
    type: String,
    required: true
  },
  page_num: {
    type: String,
    required: false
  },
})
</script>
<template>
  <div class="article">
    <span>記事コード：{{ id }}</span>
    <div>
      ［<RouterLink v-bind:to="`/content/${id}/page/1`">1</RouterLink>］
      ［<RouterLink v-bind:to="`/content/${id}/page/2`">2</RouterLink>］
      ［<RouterLink v-bind:to="`/content/${id}/page/3`">3</RouterLink>］
    </div>
    <hr />
    <RouterView />
  </div>
</template>
```

［リスト］PageView.vue

```
<script setup>
defineProps({
  page_num: {
    type: String,
    required: true
  },
})
</script>
<template>
  <div class="page">
    <span>ページ番号：{{ page_num }}</span>
  </div>
</template>
```

ルートを入れ子にする場合、親テンプレートの側でも子コンポーネントを埋め込むための領域を、<RouterView>要素（太字）で確保しておかなければならない点に注目です。

なお、「/content/108」のようなパスでアクセスした場合には、PageViewコンポーネントは表示されません（「page/:page_num」にはマッチしていないからです）。

［3］ルートコンポーネントを編集する

最後に、入れ子のビューを表示するためのルートにリンクを追加しておきます。

［リスト］RouteBasic.vue

```
<nav>
  …中略…
```

```
  ［<RouterLink to="/content/108">NestView</RouterLink>］
</nav>
```

以上を理解できたら、サンプルを実行し、［NestView］リンクにアクセスしてみましょう。更にページ番号を表す［1］～［3］リンクにアクセスすることで、ContentView→PageViewコンポーネントが入れ子に表示されることが確認できます。

---

1. リンク先のコンポーネントについては、既出の知識で理解できる内容なので、紙面上は割愛します。↵

2. ここでは、ルートパラメーターをそのまま表示しているだけですが、一般的にはデータベースなどへの問い合わせキーとして利用することになるでしょう。↵

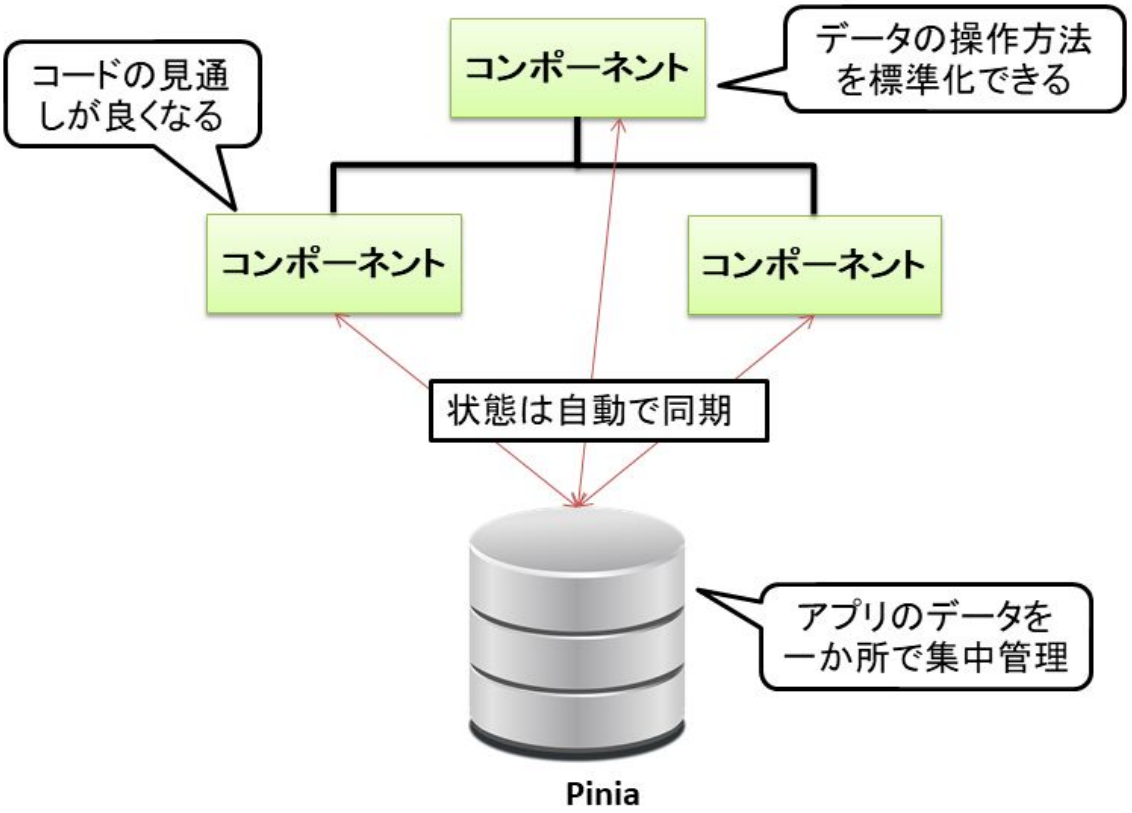
3. 「¥¥」は「¥」の意味です。文字列リテラルでは「¥」がエスケープシーケンスを意味するので、エスケープしています。↵

Part 12：Pinia

Part 8では、コンポーネント間でのデータの受け渡しについて学びました。「Props down, Event up」の手法はデータ共有の基本的な手段ですが、アプリが複雑になればコードも煩雑になります（コンポーネント階層が深くなれば、受け渡しのコードも増えるからです）。Provide／Injectの手法を用いることで、ある程度の煩雑さは解消できますが、データ処理が散在する点は変わりませんし、そもそもコンポーネント階層を跨いだ情報共有にProvide／Injectは利用できません。

そこで一定以上の規模のアプリでは、Piniaを利用することをお勧めします。Piniaは、アプリで扱うデータを一元的に管理するためのライブラリです。

図：Piniaの利点



Piniaを利用することで、アプリのデータを一か所で集中管理できるようになりますし、データの操作方法を標準化できるので、コードの見通しも改善します。一度散在してしまったデータを、あとでPiniaに集めるのは大変なので、ある程度の規模のアプリになることが判っている場合には、開発に取り組む段階で最初からPiniaを導入しておくことをお勧めします。

［Note］すべてのデータを集めなくても良い

ただし、Piniaを導入したからといって、すべてのデータをPiniaで管理しなければならないわけではありません。コンポーネントローカルな状態を管理す

るような情報は、これまでと同じく、コンポーネント内部（Ref変数／Reactive変数）で管理すれば十分です。

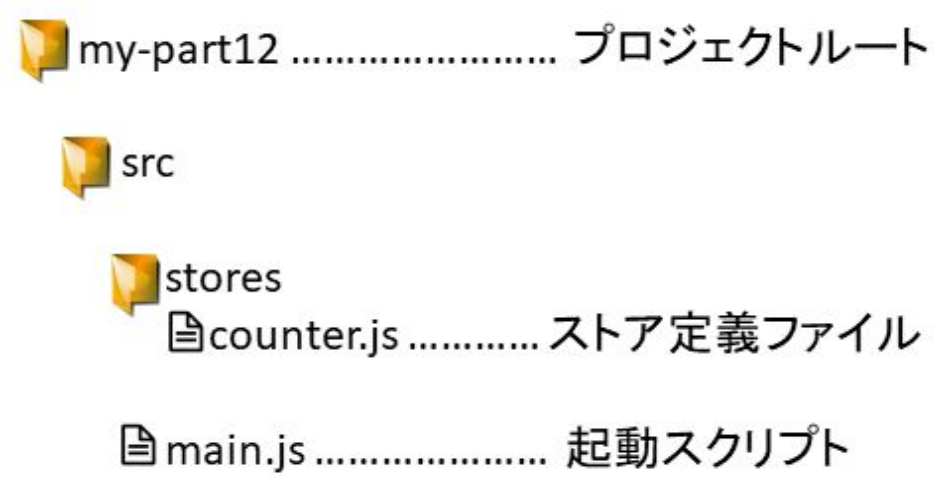
［Note］Vuex

従来、データ管理ライブラリとしてはVuexが提供されていましたが、現在ではVue CLIと同様、メンテナンスモードの扱いとなっています。新しい開発では、原則としてPiniaを優先して利用するようにしてください。

12.1 Piniaの組み込み

create-vueからPiniaを利用するには、プロジェクト作成ウィザードでモジュール選択の際に「Add Pinia for state management?」という設問に対してYesを選択してください。Piniaを有効にした場合、プロジェクトには標準的なフォルダー／ファイルに加えて、以下のようなフォルダー／ファイルが追加されます（main.jsは元からあったものですが、Piniaを加えることで中身も変化するので、図でも挙げています）。

図：Pinia環境のフォルダー構造



main.jsはPiniaをアプリに組み込んでいるだけなので、create-vueを利用している限りはほとんど意識する必要はありません。

［リスト］main.js

```
import { createApp } from 'vue'
import { createPinia } from 'pinia'
```

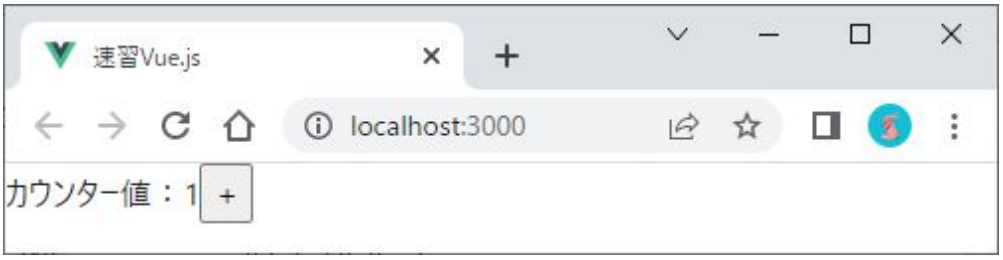
```
import App from './App.vue'

const app = createApp(App)
// Piniaを有効化
app.use(createPinia())
app.mount('#app')
```

Piniaを利用するには、まずは/storesフォルダー配下のストア定義ファイルを編集して、管理すべきデータ／操作（メソッド）を定義する、と覚えておきましょう。具体的な内容は次節以降で、順に解説していきます。

12.2 Piniaによるカウンターアプリ

Piniaを利用するための準備ができたところで、簡単なアプリを作成してみましょう。扱うのは、［+］［-］ボタンをクリックすると、カウンターがインクリメントされていく――ごく典型的なカウンターアプリです。

図：［+］［-］ボタンでカウンターが増減



［1］ストアを定義する

まずは、Piniaストアの準備からです。アプリで扱うデータと、これを更新するための手段（メソッド）を準備します。プロジェクトを作成した時点で、サンプルストア（/stores/counter.js）が準備されているので、こちらを確認するに留めます。

［リスト］stores/counter.js

```
import { defineStore } from 'pinia'

// a. ストアを生成
export const useCounterStore = defineStore({
  id: 'counter',
  // b. ステートの定義
  state: () => ({
```

```
    counter: 0
  }),
  // ゲッターの定義
  getters: {
    doubleCount: (state) => state.counter * 2
  },
  // c. アクションの定義
  actions: {
    increment() {
      this.counter++
    }
  }
})
```

a. ストアを定義する

Piniaストアを作成するのは、defineStore関数の役割です。

> **［構文］defineStoreメソッド**
> defineStore(*defs*)
> - defs：ストアの構成情報

引数defには、ストアを構成する要素を「要素名: 定義, ...」の形式で列挙していきます。指定できる要素には、以下のようなものがあります。

- id：ストアのid値

- state：ステート（データ本体）
- getters：ゲッター（ステートからの値を取得するためのしくみ）
- actions：アクション（ステートを更新するためのしくみ）

idは、ストアを一意に特定するためのキーです。アプリで重複しない名前を付けると共に、管理しやすくするためにファイルと同じ名前としておきます（この例であれば、counter.js）。もちろん、名前は用途に応じて適宜変更しても構いません[1]。

defineStoreメソッドの戻り値は、あとでStoreオブジェクトを生成するためのコンポジション関数です。一般的には「use + ストア名 + Store」形式（キャメルケース記法）で命名します。

b. ステートを定義する

ここからは具体的なストア配下の構成要素について見ていきます。既に複数の例が用意されていますが、ここでは、最低限、ステートとアクションについて確認しておきましょう。

まず、ストアの中核となるのはステートです。**ステート**とは、Piniaストアで管理されるデータ本体。ステートで管理すべき情報を「名前: 初期値, ...」のオブジェクト形式で定義しておきましょう。

ただし、書き方に少しだけ癖があります。というのも、オブジェクトそのものではなく、「オブジェクトを返す関数」を渡さなければなりません。最初は違和感を感じるかもしれませんが、まずは以下の記法をイディオムとして覚えてしまいましょう。

```
state: () => ({
  プロパティ名: 値,
  …
}),
```

この例では、カウンター値を表すcounter（初期値は0）だけを定義していますが、もちろん、必要に応じて複数の項目を列挙しても構いません。

c. アクションを定義する

Piniaの世界では、ステートは専用のメソッド経由で更新するのがお作法です。それによって、ステートの更新元が限定できるので、コードの見通しが良くなります。

このようなステート更新のためのメソッドのことを**アクション**と呼びます。今回の例では、ステートcounterを増減するためのincrementメソッドを用意しています。アクション配下では「this.名前」でステートにアクセスできます [2]。

［3］ストアにアクセスする

以上でストアそのものの準備は完了です。あとは、コンポーネントから実際にストアを呼び出してみましょう。

［リスト］PiniaBasic.vue

```
<script setup>
import { storeToRefs } from 'pinia'
import { useCounterStore } from '@/stores/counter'
// a. ストアを準備
```

```
const store = useCounterStore()
// b. 現在のカウンターを取得
const { counter } = storeToRefs(store)
// c. ［+］ボタンでカウンター値を加算
const onclick = () => {
  store.increment()
}
</script>
<template>
  <!-- カウンター値を取得-->
  カウンター値：{{ counter }}
  <input type="button" v-on:click="onclick" value="+" />
</template>
```

ストアを利用するには、先ほどストア定義ファイルで定義したコンポジション関数useCounterStoreでStoreオブジェクトを生成しておきます（a.）。一般的には、ストア同士を区別するために変数名は、コンポジション関数からuseを取り除いたもの――ここではcounterStoreとしておくべきですが、単一のストアしか利用していないならばstoreで十分でしょう。

Storeオブジェクトを生成できてしまえば、あとは「store.名前」でステートにアクセスできます（ストアはそれ自体がReactive変数なので、valueプロパティを介する必要もありません）。

ただし、テンプレートでいちいち「store.～」と表すのは冗長なので、分割構文で必要なステートを取り出しておくのが一般的です（b.）。その際、以下のように

書いてはいけない点に注意です。

```
const { counter } = store
```

分解した際にReactiveが解除されてしまうからです。そこで分解に際しては、storeToRefs関数を介して、明示的にRef化するのを忘れないようにしてください。

c. は［+］ボタンをクリックしたときに呼び出されるイベントハンドラーです。incrementアクションを呼び出して、counterステートを更新しています。アクションもStoreオブジェクトのメソッドとして呼び出せる点に注目です。

［Note］onclickメソッドの別解

onclickメソッドの中身は、「counter.value++」と書いても同じ意味です。ただし、先ほども触れたように、ステートの操作コードがアプリのそちこちに散在するのは、更新元が不明瞭になるという意味でも望ましくありません。まずは、ストア側でアクションを定義するようにしてください。

12.3 Piniaストアの活用

以上が、Piniaを利用した最低限のコードです。ここからは、Piniaをより活用するためにPiniaストアが提供する、その他の要素について解説しておきます。

12.3.1 ステート値を加工／演算する - ゲッター

ゲッター（getters）は、ステートの値を加工／演算するためのメソッドです。コンポーネントの算出プロパティにも似ていますが、ステートに関わる加工／演算は、ストアにまとめておいた方が、似たような算出プロパティが複数のコンポーネントに散在するのを防げます。

たとえば以下は、カウンター値（counter）を一の位で切り上げたものを返すceilingゲッターの例です。

［1］ゲッターを定義する

まずはPiniaストアの側を修正します。

［リスト］stores/counter.js

```
export const useCounterStore = defineStore({
  id: 'counter',
  …中略…
  // ゲッターを定義
  getters: {
    …中略…
    ceiling: (state) => {
```

```
    // 丸めの桁数（10の位で丸め）
    const round = 10
    return Math.ceil(state.counter / round) * round
   },
  },
  …中略…
})
```

ゲッターは、gettersキーの配下に「名前: 関数,...」の形式で定義します。既定ではdoubleCountゲッターが既に用意されていますが、ここでは自分でceilingゲッターを追加してみましょう。

ゲッターは、引数としてステート（state）を受け取るので、ステート値にはstate.counterのようにアクセスできます [3]。この例であれば、取得したカウンター値をround桁で切り上げたものを返しています。

［2］ゲッターにアクセスする

準備したceilingゲッターに、コンポーネントからアクセスしてみましょう（修正部分は太字）。

［リスト］PiniaBasic.vue

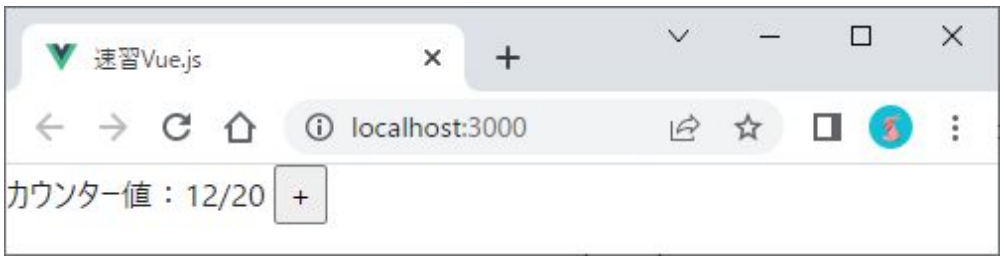
```
<script setup>
const store = useCounterStore()
const { counter, ceiling } = storeToRefs(store)
…中略…
</script>
<template>
```

```
  カウンター値：{{ counter }}/{{ ceiling }}
  <input type="button" v-on:click="onclick" value="+" />
</template>
```

ゲッターも、ステート／アクションと同じくStoreオブジェクトのプロパティとしてアクセスできます。分解に際して、storeToRefs関数で括る点もステートの場合と同様です。

以上を理解したら、サンプルを再度実行してみましょう。以下のように、10の位で丸められたカウンター値が表示されます。

図：ゲッター経由で取得した値を表示



補足：引数付きのゲッターを定義する

ゲッターには、引数を渡すこともできます。たとえば先ほどのceilingゲッターで、丸めの桁数を引数として渡せるようにするには、以下のようにします。

［リスト］stores/counter.js

```
getters: {
  ceiling: (state) => {
    return round => Math.ceil(state.counter / round) * round
```

```
  }
},
```

引数を受け取るゲッターでは、（ゲッターが返すべき値ではなく）「値を返すための関数」を返す必要があります（太字）。そして、ゲッターが受け取るべき引数（ここではround）も、入れ子になった関数で受け取るようにするわけです。

このように定義したゲッターには、コンポーネント側ではメソッドのようにアクセスできます。

［リスト］PiniaBasic.vue

```
import { computed } from 'vue'
…中略…
const { counter } = storeToRefs(store)
const ceiling = computed(() => store.ceiling(10))
```

引数付きの場合は、分解構文で単純に分解することはできないので、computed関数で算出プロパティ化しています。

12.3.2 アクションによる操作を監視する - アクションサブスクリプション

これまでに何度も触れたように、ストアに関わる操作はアクションに限定すべきです。それによってコードの見通しが良くなるというメリットもありますが、Pinia標準の機能によるアクションの監視が可能になります。

たとえば以下は、アクション成功時にcounterステートの内容をローカルストレージ [4]に保存する例です。

［リスト］PiniaBasic.vue

```
const store = useCounterStore()
const unsubscribe = store.$onAction(
  ({ name, store, args, after, onError }) => {
  // アクション成功時の処理
  after(result => {
    localStorage['quick_vue'] = JSON.stringify(store.counter)
  })
  onError(error => {
    console.log(error)
  })
})
```

アクション実行時の処理（**アクションサブスクリプション**）を登録するのは$onActionメソッドの役割です。

> ［構文］$onActionメソッド
>
> $onAction(*callback*, *keep*)
>
> - callback：アクション実行時に呼び出されるコールバック関数
> - keep：コンポーネント破棄時にサブスクリプションを維持するか（既定はfalse）

コールバック関数（callback）に渡されるオブジェクトは、以下のようなプロパティを持っています（引数全体を{ … }で括っているのは分割構文で、プロパティを対応する変数に振り分けています）。

- name：アクションの名前
- store：ストアオブジェクト
- args：アクションに渡された引数（配列）
- after：アクションが成功した時に実行される関数
- onError：アクションが失敗した時に実行される関数

この例であれば、after／onErrorを利用して、アクション成功／失敗時の処理を登録しています。after／onError関数には、それぞれアクションの戻り値（result）、エラー情報（error）が渡されるので、それらの情報に基づいて

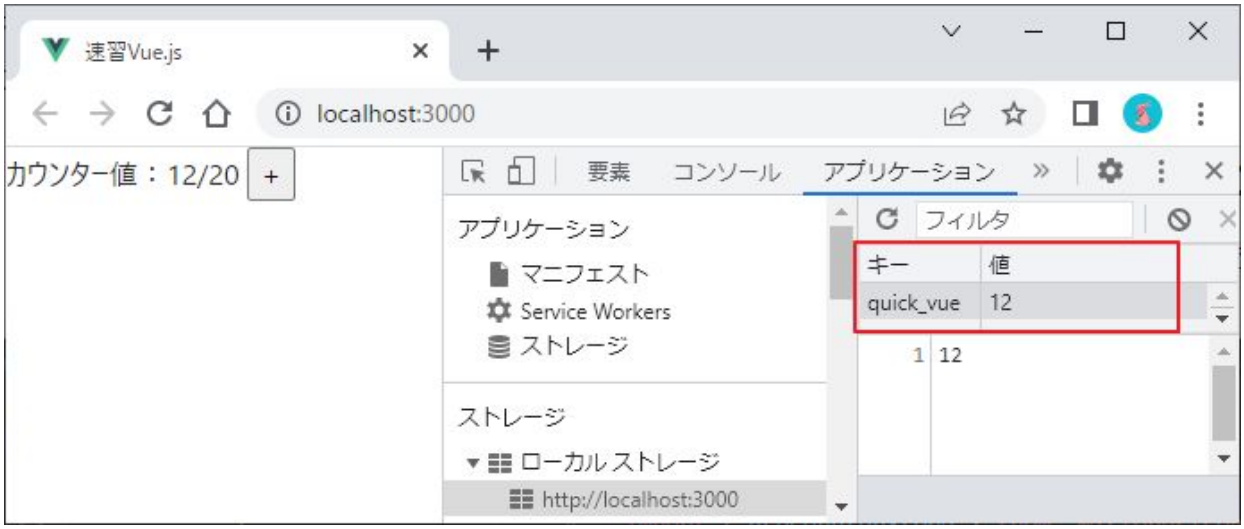
- after：ローカルストレージへの登録
- onError：エラーログの出力

を行っているわけです。

なお、$onActionの戻り値は、サブスクリプションを解除するための関数です。既定でコンポーネントが破棄される際に、サブスクリプションも破棄されますが、任意のタイミングでサブスクリプションを停止するならば、「unsubscribe()」のように明示的に解除関数を呼び出します。

以上を理解したら、サンプルを再度実行してみましょう。カウンターの値に応じて、ローカルストレージの内容も更新されることが確認できます。

図：ローカルストレージに反映される



ローカルストレージの内容は、開発者ツールの［アプリケーション］タブから［ストレージ］－［ローカルストレージ］－［http://localhost:3000］を辿ることで確認できます。

12.3.3 Piniaストアを拡張する - プラグイン

プラグインとは、Piniaストアそのものの機能を拡張するためのしくみです。大雑把には、以下のような実装に際して利用します。

- ストアにステート／アクションを追加する
- 既存のアクションのラップ
- 独自オプションの追加

たとえば以下は、ストアをロードした時に、先ほど記録したカウンター値をステートに反映させるStoragePluginプラグインの例です。

［リスト］StoragePlugin.js

```
export default function ({ pinia, app, store, options }) {
  store.counter = localStorage['quick_vue'] ?
    JSON.parse(localStorage['quick_vue']) : 0
}
```

プラグインの実体は関数です（便宜的に**プラグイン関数**と呼びます）。プラグイン関数の引数はコンテキストオブジェクトです。具体的には、Piniaに関連する以下のようなプロパティを提供しています。

- pinia：Piniaオブジェクト（createPinia関数の戻り値）
- app：アプリインスタンス（createApp関数の戻り値）
- store：ストアオブジェクト
- options：ストアオプション（defineStore関数に渡されたオブジェクト）

この例であれば、「store.counter =～」でストレージの内容をcounterステートに反映させています（指定のキーが存在しない場合にはゼロで初期化します）。

定義済みのプラグインを適用するには、main.jsを以下のように編集してください。

［リスト］main.js

```
import StoragePlugin from './plugins/StoragePlugin'
…中略…
app.use(router)
const pinia = createPinia()
```

```
pinia.use(StoragePlugin)
app.use(pinia)
```

プラグインを登録するのは、Piniaオブジェクトのuseメソッドの役割です。Piniaオブジェクトはcreate Pinia関数の戻り値として取得できます。プラグインを登録した後、Piniaオブジェクトそのものをapp.useメソッドでアプリに登録するのを忘れないようにしてください。

12.3.4 補足：アクションサブスクリプションのプラグイン化

12.3.2節では、説明の都合上、コンポーネント上でサブスクリプションを登録しましたが、プラグインでまとめて登録することもできます（この例であれば、ストレージへの出し入れという対となる操作なので、プラグインとしてまとめる方が適切でしょう）。

［リスト］StoragePlugin.js

```
export default function ({ pinia, app, store, options }) {
  store.counter = localStorage['quick_vue'] ?
    JSON.parse(localStorage['quick_vue']) : 0
  const unsubscribe = store.$onAction(({ name, store, args, after, onError })
=> { ... })
}
```

---

1. ストアを複数設置したい場合には、同じ要領でストア定義ファイルを準備します。繰り返しですが、idは重複してはいけません。

2. この例では引数なしのアクションを定義していますが、普通のメソッドと同じく、引数を加えても構いません。↵

3. ストア内の他のゲッターにアクセスすることも可能です。それには、「this.ゲッター名」のようにthis経由でアクセスしてください。↵

4. ブラウザー標準のストレージです。キー／値の組み合わせで値を管理できます。↵

Part 13：ユニットテスト

アプリの品質を保証するために、テストという過程は欠かせません。テストと一口に言っても、人間が実際にアプリを操作して動作を確認するようなテストもありますが、本Partで扱うのは自動化されたテストです。テストをあらかじめコード化し、ツールから自動で実行することで、何度も繰り返し確認が可能です。つまり、アプリを更新してもコマンド一つで敏速にアプリ全体を再確認できます。

Vue（create-vue）でも自動テストを重要視しており、以下のようなテストを標準でサポートしています。

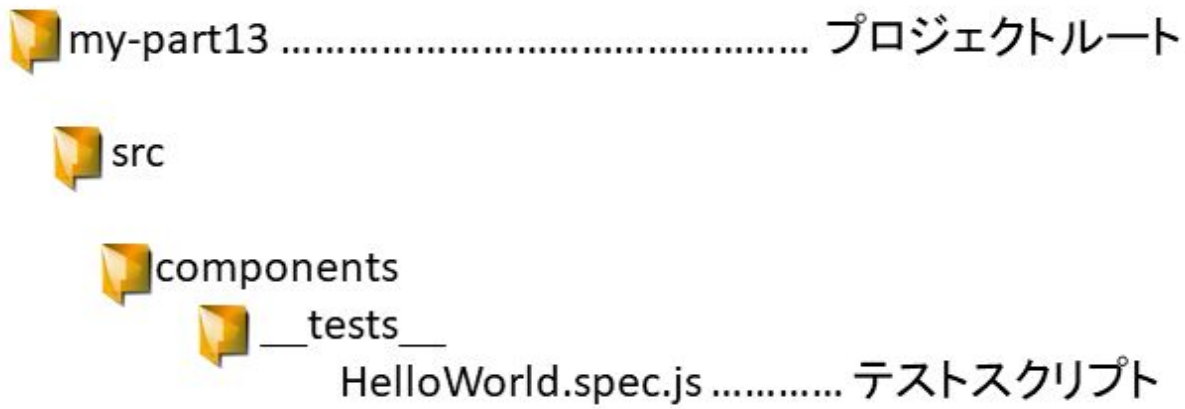
- ユニットテスト：コンポーネント、JavaScriptオブジェクトなど、要素単体の動作を確認するためのテスト。単体テストとも。
- E2E（End to End）テスト：複数のコンポーネントに跨って、アプリ全体の動作を確認するためのテスト。インテグレーションテストとも。

本Partでは、より基本的なユニットテストを例に、Vueによる標準的なテストの手法を理解します。Vue（create-vue）では、ユニットテストのためのライブラリとしてVitest＋Vue Test Utilsを推奨しています。Vitestは汎用的なテスティングフレームワーク、Vue Test UtilsはVueコンポーネントをテストするためのユーティリティです。

13.1 ユニットテストの基本

create-vueからVitest＋Vue Test Utilsを導入するには、プロジェクト作成ウィザードでモジュール選択の際に「Add Vitest for Unit Testing?」という設問に対してYesを選択してください。Vitestを有効にした場合、プロジェクトには標準的なフォルダー／ファイルに加えて、以下のようなフォルダー／ファイルが追加されます。

図：Vitest環境のフォルダー構造



`/components/__tests__`フォルダーは、コンポーネントに関わるテストコードを格納するためのフォルダーです。既定でHelloWorldコンポーネント（2.2節）をテストするためのHelloComponent.spec.jsが用意されていますが、自分でコードを記述した場合にも、このフォルダーに対してファイルを追加していきます。

13.1.1 テストコードを確認する

ユニットテストの基本を理解するために、まずは既定で用意されたテストコードを確認してみましょう。テスト対象との対応関係が明確になるよう、まずはテストコードとテスト対象の名前は同名にしておくべきです。この例であれば、テスト対象がHelloWorld.vueであるのに対して、テストコードはHelloWorld.spec.jsです。

［リスト］HelloWorld.spec.js

```
import { describe, it, expect } from 'vitest'
import { mount } from '@vue/test-utils'
import HelloWorld from '../HelloWorld.vue'
// a. テストスイートを定義
describe('HelloWorld', () => {
  // b. テストケースを定義
  it('renders properly', () => {
    // c. コンポーネントを起動
    const wrapper = mount(HelloWorld, { props: { msg: 'Hello Vitest' } })
    // d. 結果の検証
    expect(wrapper.text()).toContain('Hello Vitest')
  })
})
```

ポイントを順番に見ていきます。

a. テストスイートを定義する

テストスイートとは、関連するテストを束ねる入れ物のようなものです。describe関数で定義します。

> **［構文］describe関数**
> describe(*name*, *specs*)
> - name：テストスイートの名前
> - specs：テストケース（群）

引数nameはテスト実行時に表示される名前なので、なるべく対象が明確になるように命名してください。

b. テストケースを定義する

テストスイートの配下で表された具体的なテストを**テストケース**と言います。テストケースを定義するのはit関数の役割です。

> **［構文］it関数**
> it(*name*, *fn* [, *timeout*])
> - name：テストケースの名前
> - fn：テスト本体
> - timeout；タイムアウト時間（既定は5000ミリ秒）

ここではrenders properlyテストをひとつだけ定義していますが、複数のテストを定義するならばit関数を列記するだけです。

c. コンポーネントを起動する

テストコード（＝it関数の引数fn）は、大雑把に以下の手順で表します。

- 本来のコードを実行する
- 実際の結果と期待する値とを比較する

コンポーネントをテストするならば、まずはコンポーネントを起動しておきましょう。これを行うのが、Vue Test Utilsのmount関数です。

> ［**構文**］**mount関数**
> mount(*comp*, *opts*)
> - comp：起動するコンポーネント
> - opts：起動のための情報（「オプション名: 値, ...」形式）

引数optsで指定できる主なオプションには、以下のようなものがあります。

- attrs：属性（「属性名: 値, ...」形式）
- props：プロパティ（「プロパティ名: 値, ...」形式）
- data：Ref／Reactive変数（「名前: 値, ...」形式）
- slots：スロット情報（「スロット名: 値, ...」形式）
- shallow：浅いマウントをするか

この例であれば、以下のようにコンポーネントを呼び出しなさい、という意味になります。

```
<HelloWorld msg="Hello Vitest" />
```

mount関数の戻り値はコンポーネントのラッパー（VueWrapper）で、コンポーネントの情報を取得＆テストするための種々のプロパティ／メソッドを提供します。

- vm：アプリインスタンス
- attributes(*name*)：属性nameの値を取得
- classes(*name*)：スタイルクラスclazzの有無を判定
- find(*exp*)：セレクター式expに合致する単一の要素を取得
- findAll(*exp*)：セレクター式expに合致する要素群を取得
- text()：配下のテキスト
- html()：配下の要素（HTML文字列）
- setProps(*obj*)：「名前: 値,...」形式でプロパティを設定
- setValue(*v*)：フォーム要素に値vを設定
- trigger(*type* [,*opts*])：イベントtypeを発生（引数optsはイベントオブジェクト）
- emitted()：発生したイベント情報

d. 実行結果が正しいかを検証する

コンポーネントを実行できたところで、その結果が意図したものであるかを確認するのが、以下の構文です。

> **［構文］結果の検証**
> expect(*resultValue*).*matcher*(*expectValue*)
> - resultValue：テスト対象のコード（式）

> - matcher：検証のためのメソッド
> - expectValue：期待する値

この例であれば、textメソッドでコンポーネント配下のテキストを取得し、その中に「Hello Vitest」という文字列が含まれているか（toContain）を確認しています。toContainは**Matcher**とも呼ばれ、expect関数で得られた結果値が正しいかを判定するためのメソッドです。Vitestでは、toContainの他にも、以下のようなMatcherを用意しています。

- toBe(*value*)：値がvalueと等しいか
- toBeTruthy()：値がtrueに変換可能な値であるか
- toBeFalsy()：値がfalseに変換可能な値であるか
- toBeNull()：値がnullであるか
- toBeGreaterThan(*value*)：値がvalueより大きいか
- toBeGreaterThanOrEqual(*value*)：値がvalue以上であるか
- toBeLessThan(*value*)：値がvalue未満であるか
- toBeLessThanOrEqual(*value*)：値がvalue以下であるか
- toMatch(*re*)：値が正規表現reにマッチするか
- toBeInstanceOf(*clazz*)：値が指定された型clazzであるか

ちなみに、否定を表すならば、以下のようにnotメソッドを付与します。以下は、指定のテキストが含まれ**ない**ことを検証します。

```
expect(wrapper.text()).not.toContain('Hello Vitest')
```

d. の例では、コンポーネント配下のテキストを全て取得して、そこに目的の文字列が含まれるかを確認していますが、より細かく、`<h1>`要素配下のテキストが、意図した文字列に等しいことを確認することもできます。

```
expect(wrapper.find('h1').text()).toBe('Hello Vitest')
```

13.1.2 テストを実行する

準備済みのテストを実行するには、以下のコマンドを実行します。

```
> npm run test:unit     （テストを実行）
...中略...
✓ src/components/__tests__/HelloWorld.spec.js (1)

Test Files  1 passed (1)
     Tests  1 passed (1)
      Time  8.36s (in thread 16ms, 51771.85%)

PASS  Waiting for file changes...
       press h to show help, press q to quit
```

この例であれば、テストスイート（ファイル）、テストケース共にひとつのうちひとつが成功しています。また、「Waiting for file changes...」とメッセージが表示されて、テストコードの変更が待機待ちになっていることを確認してください。

単発でテストが終了するわけではなく、以降、テストコードが監視されて、変更があった場合に自動的に再実行してくれるのです。試してみましょう。

［リスト］HelloWorld.spec.js

```
expect(wrapper.text()).toContain('Hello JavaScript')
```

ファイルを保存すると同時にテストが再実行され、今度は以下のような結果が表示されます（比較する文字列だけを変化させたので、当然失敗します）。

```
Re-running tests... [ src/components/__tests__/HelloWorld.spec.js ]

> src/components/__tests__/HelloWorld.spec.js (1)
   > HelloWorld (1)
     × renders properly
…中略…
AssertionError: expected 'Hello Vitest You've successfully crea…' to include
'Hello JavaScript'
…中略…
Test Files  1 failed (1)
     Tests  1 failed (1)
      Time  1.89s (in thread 18ms, 10645.03%)

FAIL  Tests failed. Watching for file changes...
       press h to show help, press q to quit
```

13.1.3 テスト共通のコードを切り出す

テストケースが増えてくると、複数のテストケースで重複した記述が出てきます。たとえばコンポーネントの初期化（mount）などがそれです。そのような処理は、beforeEachメソッドとして切り出すべきです。

先ほどのHelloWorld.spec.jsを書き換えてみましょう。

［リスト］HelloWorld.spec.js

```
describe('HelloWorld', () => {
  // テストスイート共通で利用できる変数
  let wrapper
  // a. テストケースの単位に実行される処理
  beforeEach(() => {
    wrapper = mount(HelloWorld, { props: { msg: 'Hello Vitest' } })
  })
  // 本来のテストケース
  it('renders properly', () => {
    expect(wrapper.text()).toContain('Hello Vitest')
  })
})
```

beforeEach関数は、テストケースそれぞれが実行される前に実行すべき処理を表します。この例では却って冗長になっていますが、これによって、it関数が増えた場合にもmount呼び出しの重複を防げます [1]。

ちなみに、共通的な処理を表す関数としては、他にも以下のようなものがあります。

- afterEach：テストケース個々が実行された直後に行うべき処理
- beforeAll：すべてのテストを実行する前に行うべき処理
- afterAll：すべてのテストを実行した後に行うべき処理

13.2 ユニットテストのさまざまな手法

ユニットテストの基本を理解できたところで、プロパティ／イベントなど、コンポーネントの基本的な機能をテストするための方法を見ていきます。

13.2.1 プロパティのテスト

コンポーネントのプロパティは、マウント時に初期化して終わり、というものではありません。親コンポーネントから随時更新される可能性のあるものです。

そこでユニットテストでも、プロパティの変更が出力に正しく反映されるかを確認してみましょう。たとえば以下は、HelloWorldコンポーネントのmsgプロパティを変更した結果を確認するためのテストです。

［リスト］HelloWorld.spec.js

```
it('Props Change', async () => {
  const msg = 'こんにちは、Vue！'
  // a. msgプロパティの値を設定
  await wrapper.setProps({ msg })
  expect(wrapper.find('h1').text()).toBe(msg)
})
```

プロパティ値を設定するのは、VueWrapper#setPropsメソッドの役割です（a.）。「プロパティ名: 値, ...」のオブジェクト形式で指定します [2]。

構文そのものはシンプルですが、呼び出しに際してawait演算子を付与しなければならない点に注意です。というのも、Vueの世界ではプロパティ値の反映は非同期に実施されます（＝即座に反映されません）。そこでawait演算子で非同期処理の終了を待ってから、値の正否を検証しているのです（awaitを外した場合には、正しくプロパティの変化を確認できません）。

await演算子を付与した場合には、テスト関数そのものにもasync修飾子（太字）を付与して、非同期関数であることを明示しなければなりません。

13.2.2 入れ子になったコンポーネントのテスト

mount関数は、既定で階層化されたすべてのコンポーネントをマウントします。8.2.1項のPropBasic.vueで確認してみましょう。

［リスト］MyComponent.spec.js[3]

```
import { describe, it, expect, beforeEach } from 'vitest'
import { mount } from '@vue/test-utils'
import PropBasic from '@/components/p08/PropBasic.vue'

describe('Quick Vue', () => {
  it('Nest Component', () => {
    const wrapper = mount(PropBasic, { })
    // コンポーネントの実行結果を出力
    console.log(wrapper.html())
  })
})
```

Nest Componentテストは、PropBasicコンポーネントをマウントし、その結果を出力しているだけのコードで、Matcherは含まれていません。

このテストを実行した結果は、以下の通りです。配下のMyHelloコンポーネントまでが実行されていることが確認できます。

```
stdout | src/components/__tests__/MyComponent.spec.js > Quick Vue > Nest Component
<div>こんにちは、鈴木次郎さん！ </div>
```

では、太字を以下のように書き換えるとどうでしょう。

```
const wrapper = mount(PropBasic, { shallow: true })
```

結果が以下のように変化します。

```
stdout | src/components/__tests__/MyComponent.spec.js > Quick Vue > Nest Component
<my-hello-stub name="鈴木次郎"></my-hello-stub>
```

配下のコンポーネントが`<my-hello-stub>`で置き換わって、そのまま描画されます。これがshallowオプションの意味で、コンポーネントを「浅く」マウントします。配下のコンポーネントをダミーで置き換え、実行そのものは無視するという意味です。

テストの目的にも依りますが、親コンポーネントをテストする上で、常に子コンポーネントの解釈が必要なケースばかりではありません。むしろ子コンポーネントと連携するために、「テストが複雑になる」「テストの処理時間が長くなる」などの問題が発生することがあります。テストが子コンポーネントとの連携を目的としていないならば、積極的にshallowオプションでマウント範囲を限定することをお勧めします。

13.2.3 イベントのテスト

Vue Test Utilsでは、イベント（ユーザー操作）を伴うテストを可能です。たとえば以下は、5.5節でも触れたEventMouseコンポーネントをテストするコードです。

［リスト］MyComponent.spec.js

```
import EventMouse from '@/components/p05/EventMouse.vue'
…中略…
it('Event Basic', async () => {
  const wrapper = mount(EventMouse)
  // a. マウスイベントを発生
  await wrapper.trigger('click.right.prevent', {
    clientX: 100,
    clientY: 50
  })
  // 結果の確認
  expect(wrapper.find('#main').text()).toContain('100, 50')
})
```

マウスイベントを発生させるのはtriggerメソッドの役割です（**a.**）。

> **［構文］triggerメソッド**
> trigger(*eventType* [, *options*])
> - eventType：イベントの種類
> - options：イベントオプション（「名前: 値, ...」形式）

引数eventTypeには「click.right.prevent」のような修飾子を含んだ形式で、イベント名を指定できます。引数optionsは、要は、イベントオブジェクトです。この例であれば、マウスポインターの座標を擬似的に生成しています。

なお、プロパティの場合と同じく、イベントによる処理の反映は非同期です。triggerメソッドはawait演算子でマークすると共に、テスト関数そのものをasyncで修飾しておきます。

13.2.4 カスタムイベントのテスト

VueWrapperオブジェクトでは、カスタムイベントの発生を監視し、イベントの発生回数、授受されたイベントオブジェクトの値をテストすることもできます。たとえば以下は、8.3.2項でも触れたEventCounterコンポーネントをテストする例です。

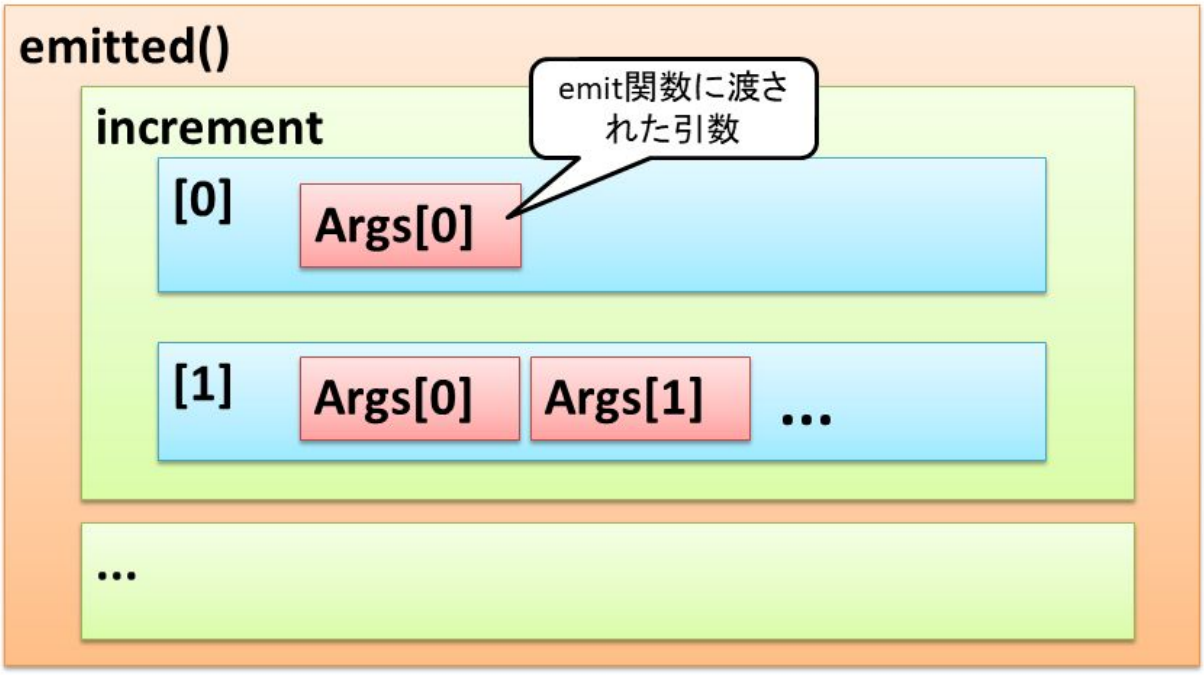
［リスト］MyComponent.spec.js

```
import EventCounter from '@/components/p08/EventCounter.vue'
...中略...
it('Custom Event', async () => {
```

```
  const wrapper = mount(EventCounter, { props: { step: 5 } })
  await wrapper.find('input').trigger('click')
  // イベント情報を取得
  const emit = wrapper.emitted()
  console.log(emit)
    // 結果：{ increment: [ [ 5 ] ], click: [ [ [MouseEvent] ] ] }
  // b. incrementイベントが何回発生したか
  expect(emit.increment.length).toBe(1)
  // c. incrementイベントで発生したデータの確認
  expect(emit.increment[0][0]).toBe(5)
})
```

コンポーネント配下で発生したイベント情報はVueWrapperオブジェクトに蓄積されます。これを取得するのがemittedメソッドの役割です（**a.**）。取得できるイベント情報は、以下のような構造になっています。

図：イベント情報の構造



イベントの単位で、発生した順にイベント情報が格納されます。個々のイベント情報には、イベント発生時に渡されたイベントオブジェクトの内容が保存されます。**b.**では、この情報に基づいてincrementイベントの発生回数を、**c.**ではincrementイベントで渡されたイベントオブジェクトの内容を、それぞれ確認しています。

13.2.5 Provide／Injectのテスト

最後に、Provide／Injectを伴うコンポーネントをテストしておきます。テスト対象となるのは8.5.2項のInjectListコンポーネントです。

［リスト］MyComponent.spec.js

```
import InjectList from '@/components/p08/InjectList.vue'
...中略...
it('Provide/Inject', () => {
```

```
    // ダミーのProvide値を登録
  const wrapper = mount(InjectList, { global:
   {
     provide: {
       list: {
         books: [
           { isbn: '978-4-8156-1336-0', title: 'Vue入門', price: 1000 },
           { isbn: '978-4-8156-1336-1', title: 'React入門', price: 2000 },
           { isbn: '978-4-8156-1336-2', title: 'Angular入門', price: 3000 }
         ],
         onclick: isbn => console.log(isbn)
       }
     }
   }
  })
  // 描画された<tr>要素の個数を確認
  expect(wrapper.findAll('tr').length).toBe(3)
})
```

Provide値を伴うコンポーネントをテストする場合、global－provideオプションであらかじめテスト用の値を登録しておきます。これで上位コンポーネントで値がProvideされたことを前提に、Inject側のコンポーネントをテストできます。

［Note］globalオプション

globalオプションは、主にアプリレベルで登録された（＝テスト対象のコンポーネントで明示されていない）情報を補うために利用します。provide

サブオプションの他にも、以下のようなオプションを指定できます（いずれも「名前: 値, ...」形式で登録します）。

- components：コンポーネント
- config：設定情報
- directives：ディレクティブ
- plugins：プラグイン

---

1. ただし、以降のコードではシンプル化のため、it配下でmount関数を呼び出しています。↵
2. { msg }とあるのはオブジェクトリテラルの省略構文で、{ msg: msg }と同じ意味です。↵
3. 本来であれば、テスト対象に即した名前にすべきですが、本書では複数のテストをまとめている関係上、そのルールには沿っていません。↵

書籍情報

著者プロフィール

山田 祥寛（やまだ よしひろ）

Microsoft MVP for Visual Studio and Development Technologies。執筆コミュニティ「WINGSプロジェクト」の代表でもある。主な著書に「速習Spring Boot」「速習Django3」「速習TypeScript 第2版」「速習React」（以上、Amazon Kindle）、「改訂新版JavaScript本格入門」「Angularアプリケーションプログラミング」（以上、技術評論社）、「独習Python」「独習Java 新版」「独習C# 新版」「独習PHP 第4版」（以上、翔泳社）、「はじめてのAndroidアプリ開発 Kotlin編」（秀和システム）、「これからはじめるVue.js 3実践入門」（SBクリエイティブ）など。

基本情報

2022年4月発行

著者：山田 祥寛（やまだ よしひろ）

発行者：WINGSプロジェクト

サポートサイト

- https://wings.msn.to/
- https://wings.msn.to/index.php/-/B-14/ （お問い合わせ）

表紙の写真について

家族で、房総半島のマザー牧場へグランピングに行った時に撮った写真です。いつもは、たくさんの機械に囲まれて生活していますが、たまには、ゆっくりのんびり自然の中で過ごすのも良いですね。風の強い日で、小心者の作者はテントが剥がされないか、一晩中冷や冷やしていました（2022.04.25）。