親と子の間でのデータの受け渡し

親と子の間でのデータの受け渡しはなぜ必要か？

- 1 ページの内容を機能ごとなどで区切り部品に分けることでコンポーネント化する
- 例えば検索画面の場合、DB にアクセスするのは親コンポーネント、検索機能、検索結果一覧を別々のコンポーネント(子コンポーネント)に分けた場合、データの受け渡しが必要となる。

Props

親コンポーネントから子コンポーネントへのデータの受け渡しのこと。

Props の使い方

- 子コンポーネント(データを使いたい側)で`Prop`を設定する
- 仕様としては、親コンポーネントのデータを受け渡し、子コンポーネントで表示する

```
import { Vue, Component, Prop } from "vue-property-decorator";

@Component
export default class DataDelivery extends Vue {
  @Prop() navItem!: string;
}
```

- 親の DOM テンプレートに`v-bind`した属性として設定する

```
<DataDelivery :nav-item="nav[0]"></DataDelivery>
```

```
export default class App extends Vue {
  nav: string[] = ["Works", "About", "Contact"];
}
```

> `navItem`としても動作する

ケバブケースとキャメルケースの補足

結論

- JavaScrip はキャメルケース(camelCase)
- HTML はケバブケース(kebab-case) 上記の `<DataDelivery :nav-item="nav[0]"></DataDelivery>`は `<DataDelivery :navItem="nav[0]"></DataDelivery>`でも動くが、HTML は慣習として属性をケバブケースで書くのでそちらに従うのがよい。

子コンポーネントで受け取った Props の値を加工する

他のリアクティブなデータと同じように参照して使うことができる

```
export default class DataDelivery extends Vue {
  @Prop() navItem!: string;

  //computed
  get navItemUpperCase() {
    return this.navItem.toUpperCase();
  }
}
```

```
<p>加工しました{{ navItemUpperCase }}</p>
```

親から 2 つ以上の値を受け渡す

- 親の DOM テンプレートに`v-bind`した属性を 2 つ作る

  ```
  <DataDelivery :nav-item="nav[0]" :nav-number="0"></DataDelivery>
  ```

- 子コンポーネント(データを使いたい側)で`Prop`を設定する

  ```
  @Prop() navItem!: string;
  @Prop() navNumber!: number;
  ```

- 子コンポーネントの DOM テンプレートで使う

  ```
  <p>
    このページは{{ navNumber }}:{{ navItem }}です
  </p>
  ```

  > 親コンポーネントにデータの送り口があり、子コンポーネントにデータの受け口がある

$emit

表面上の動きは子から親にデータを受け渡すように見える。 実際は、子コンポーネントの好きなタイミングで親コンポーネントのメソッドを発火できるイメージ。

- `$emit`はカスタムイベントを作ることができる
- 子コンポーネントが親コンポーネントのデータを変えているわけではなく、親自身が変えている。(子コンポーネントにデータ自体は依存していない)
- データは単一方向

$emit の使い方

- 子コンポーネントから親コンポーネントへデータを渡す
- 仕様としては、子コンポーネントのボタンがクリックされると親コンポーネントのデータが変わる

> 発火されるイベントを作る

```
<button @click="changeMsg">親コンポーネントのメッセージを変えるボタン</button>
```

```
//@Emit('渡したい名前')
@Emit("change-msg")
changeMsg(): string {
  return this.childMessage;
}
```

- 親コンポーネントで子コンポーネントで作ったカスタムイベントを受け取る

> $emit で受け取れる(イベントが発火される)

```
<DataDelivery :nav-item="nav[0]" :nav-number="0" v-on:change-
msg="handleClick($event)"></DataDelivery>
```

```
handleClick($event: string): void {
  this.msg = $event;
}
```

> `$emit`はカスタムイベントを作るもの。子コンポーネント`$emit`を作ることで、好きなタイミングで親コンポーネントのメソッドを発火できる。 データを送らなくてもよい。その場合、イベント発生時に親のメソッドを発火する。 結果的にデータを送ることもできるので子コンポーネントから親コンポーネントへデータを送っているともいえる。(**渡しているが変えてはいない**)

> 子コンポーネントが親のデータを変えているわけではなく、子コンポーネントのタイミングで親コンポーネントが変えている

なぜ$emit の場合複雑になっているか

子コンポーネントが親コンポーネントの値を直接変えることができたら…。 親が子に依存してしまう。データフロー複雑化し、理解しずらい構造となる。 親から子の単方向のデータの流れを意図的に作っている。