Vue.js Options API の課題と Composition APIのメリット

はじめに

背景

- 前職でVue2のOptionsAPIを使用
- Vue3のCompositionAPI導入の理由に疑問

目的

- OptionsAPI→CompositionAPIになってよくなったことを話す
- 個人的に発表になれる

Options APIとは

- Vue 2から使われているAPI
- `data`, `methods`, `computed` などのオプションでコンポーネントを構成

```
export default {
  data() { return { /* データ */ } },
  methods: { /* メソッド */ },
  computed: { /* 算出プロパティ */ }
  etc...
}
```

Composition APIとは

- Vue 3で導入された新しいコンポーネント定義方法
- `ref` や `reactive` を利用する

```
import { ref, computed } from 'vue'

export default {
  setup() {
    const count = ref(0)
    const doubled = computed(() => count.value * 2)

    function increment() {
      count.value++
    }

    return { count, doubled, increment }
  }
}
```

課題1: コードが散在する

- 関連する機能のコードが異なるオプションブロックに分散
- コードの把握・追跡が困難になる

課題1: コードが散在する

- 関連するロジックを一箇所にまとめられる
- コードの把握・追跡が困難になる

Options API

Composition API

課題2: 機能の再利用性

同じ機能を複数のコンポーネントで実装したい場合OptionsAPIだと下記になる

```
// 対策1: ユーティリティ関数の分離(validation.js)
export function validateEmail(email) {
  return !email.includes('@') ? 'メールアドレスが無効です' : '';
}

// 対策2: ミックスインの利用(validation-mixin.js)
export const validationMixin = {
  methods: {
    validateEmail(email) {
      return !email.includes('@') ? 'メールアドレスが無効です' : '';
    }
  }
}
```

課題2: 機能の再利用性

ただこれだと下記のような課題がある。。

ユーティリティ関数:

- 状態（data）の共有ができない
- リアクティブな連携が複雑

ミックスイン:

- 名前空間の衝突リスク
- コードの出所が不明確になりやすい
- 複数ミックスインの相互作用が複雑

課題2: 機能の再利用性

- 特にmixinは複雑。。

```
export default {
  // どのミックスインが何を提供しているか把握しづらい
  mixins: [validationMixin, formMixin, userMixin],
  methods: {
    // コンポーネント独自の実装があるとミックスインとの関係が複雑に
    validateEmail() { /* オーバーライドのロジック */ }
  }
}
```

余談：ミックスインの優先ルール

- data: オブジェクト同士がマージされる
- methods/computed: コンポーネント側が優先され、ミックスインを上書き
  - コンポーネント側に同名のものが存在する場合コンポーネント側を利用する
  - `mixins: [mixinA, mixinB]` では mixinB が優先される
- ライフサイクルフック: すべて実行される
  - `mixins: [mixinA, mixinB]` の場合の実行順序
    mixinA のフック → mixinB のフック → コンポーネントのフック

課題2: 機能の再利用性

コンポーザブルを利用することでユーティリティ関数とmixinの課題が解決できる！

```
import { ref } from 'vue';

export function useValidation(initialValue = '') {
  const value = ref(initialValue);
  const error = ref('');

  const validateEmail = () => {
    error.value = !value.value.includes('@')
      ? 'メールアドレスが無効です' : '';
  };

  return { value, error, validateEmail };
}
```

課題3: TypeScriptとの相性の悪さ

TypeScriptの利点（自動補完、型推論）を最大限活用できない

```
export default {
  data() {
    return {
      count: 0,
      user: { name: 'タロウ' }
    }
  },

  methods: {
    increment() {
      this.count++; // OK
      this.user.age++; // エラーにならないが実行時エラー（ageは存在しない）
      this.nonExistentMethod(); // エラーにならないが実行時エラー
    }
  }
}
```

課題3: TypeScriptとの相性の悪さ

`Vue.extend`を利用すれば型定義は可能ではある(らしい)が面倒

```
interface ComponentData {
  count: number;
  user: { name: string };
}

export default Vue.extend({
  data(): ComponentData {
    return {
      count: 0,
      user: { name: 'タロウ' }
    }
  },
  methods: {
    increment() {
      this.count++;
      // this.user.age++; // エラーになる
      // this.nonExistentMethod(); // エラーになる
    }
```

課題3: TypeScriptとの相性の悪さ

標準的なTypeScript機能がそのまま使え、型推論も機能するようになる

```
import { ref, Ref } from 'vue';

interface ValidationResult {
  value: Ref<string>;
  error: Ref<string>;
  validate: () => void;
}

export function useValidation(initialValue = ''): ValidationResult {
  const value = ref(initialValue);
  const error = ref('');

  const validate = () => {
    error.value = !value.value.includes('@')
      ? 'メールアドレスが無効です' : '';
  };

  return { value, error, validate };
}
```

6. まとめ

Options API→Composition APIで下記を解決！

- 関連ロジックをまとめられるため保守性向上
- 状態とロジックが再利用しやすくなった
- TypeScriptが使いやすい

できるだけCompositionAPIを使いましょう！

ご清聴ありがとうございました！