Vue Composition APIの背景と意義

はじめに

本日は「Vue Composition APIが導入された背景と意義」について詳しくお話しします。
この内容は主に公式ドキュメント、Evan You氏のブログ記事、カンファレンス発表、およびコミュニティの知見を元にしています。

Options APIの基本

Vue 2までのコンポーネント定義方法

```
export default {
  data() {
    return {
      count: 0,
      user: { name: '', email: '' },
      searchQuery: '',
      isLoading: false
    }
  },
  computed: {
    doubledCount() { return this.count * 2 },
    filteredItems() { /* 長い処理 */ }
  },
  methods: {
    increment() { this.count++ },
    fetchUserData() { /* API呼び出し */ }
  },
  mounted() {
    this.fetchUserData()
  },
  watch: {
    searchQuery() {
      // 検索ロジック
    }
```

Options APIの根本的な課題①

コードの論理的な整理が困難に

関連する機能が複数のオプションブロックに分散:

![コードの断片化を表す図]

例: ユーザー関連機能は、data、methods、computedなど複数の場所に散らばる

Options APIの根本的な課題②

コードの可読性低下

コンポーネントが大きくなるにつれ:

- どのプロパティがどの機能に関連するか把握しづらくなる
- コードを追うために常に上下にスクロールが必要
- 関連コードの距離が離れることで論理的な繋がりが見えにくい

Options APIの根本的な課題③

コード再利用の問題

主な再利用パターン:

1. **ミックスイン**: 名前空間の衝突、出自不明の変数、暗黙的な依存関係
2. **Higher-Order Components(HOC)**: コンポーネントツリーが深くなり複雑化
3. **Renderless Components**: スコープスロットを使用した複雑な実装

Evan You氏の思考プロセス

Vue 3の設計において:

> "コンポーネントのコードベースが大きくなるにつれ、関連するロジックの断片が分離され、コードの理解と保守が困難になる。機能ごとにコードを整理する代替アプローチが必要だった"

出典: Vue 3発表時のカンファレンス, 2019

Reactのアプローチからのヒント

React Hooksの登場(2018年)がComposition API構想の触媒に:

```
// React Hooks
function Counter() {
  const [count, setCount] = useState(0);
  const doubledCount = useMemo(() => count * 2, [count]);

  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]);

  return (/* JSX */);
}
```

Composition APIの基本設計思想

1. 関心事ごとにコードをグループ化
2. コンポーネントロジックを関数として抽出可能に
3. フルTypeScriptサポート
4. シンプルでミニマルなAPI設計

Composition APIの実装例

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

export default {
  setup() {
    // カウンター機能
    const count = ref(0)
    const doubledCount = computed(() => count.value * 2)
    function increment() { count.value++ }

    // ユーザーデータ機能
    const user = ref({ name: '', email: '' })
    const isAdmin = computed(() => user.value.email.includes('admin'))

    async function fetchUserData() {
      // API呼び出し
    }

    // 検索機能
    const searchQuery = ref('')
    const isLoading = ref(false)

    watch(searchQuery, async (newQuery) => {
      if (newQuery.length) {
        isLoading.value = true
        // 検索ロジック
        isLoading.value = false
      }
    })

    onMounted(() => {
      fetchUserData()
    })

    return {
      // 露出するプロパティとメソッド
      count, doubledCount, increment,
      user, isAdmin, fetchUserData,
      searchQuery, isLoading
    }
  }
}
```

"Composition"の本質: コンポーザブル関数

ロジックを再利用可能な関数として抽出:

```
// useCounter.js
export function useCounter() {
  const count = ref(0)
  const doubledCount = computed(() => count.value * 2)

  function increment() { count.value++ }
  function decrement() { count.value-- }

  return {
    count,
    doubledCount,
    increment,
    decrement
  }
}

// コンポーネント内で使用
import { useCounter } from './useCounter'

export default {
  setup() {
    const { count, doubledCount, increment } = useCounter()

    return { count, doubledCount, increment }
  }
```

コンポーザブルとミックスインの比較

ミックスイン:

- コンポーネントにマージされる
- 出自が不明確
- 名前の衝突リスク
- 暗黙的な依存関係

コンポーザブル:

- 明示的にインポート
- 関数の戻り値として明示的に使用
- 変数名を変更可能で衝突回避
- 明示的な依存関係

TypeScriptとの統合強化

Options APIの型付けの問題:

```
export default {
  data() {
    return { count: 0 }
  },
  computed: {
    // thisの型推論が複雑
    doubled(): number {
      return this.count * 2
    }
  }
}
```

Composition APIの型付け:

```
export default {
```

パフォーマンス最適化の視点

ツリーシェイキングによる最適化

Options APIでは全機能がバンドルに含まれます。

Composition APIでは使用する機能だけをインポート:

```
// 使うAPIだけをインポート
import { ref, computed, watch } from 'vue'
```

エコシステムへの影響

VueUseライブラリの誕生:

- 400以上のコンポーザブル関数
- ブラウザAPI、センサー、アニメーション、状態管理など

```
import { useLocalStorage, useMouse, useEventListener } from '@vueuse/core'

export default {
  setup() {
    // ローカルストレージと同期
    const name = useLocalStorage('user-name', '')

    // マウス位置の追跡
    const { x, y } = useMouse()

    return { name, x, y }
  }
}
```

まとめ: Composition APIの意義

1. **コードの論理的整理**: 関心事ごとにコードをグループ化
2. **再利用性の向上**: コンポーザブル関数による明示的な再利用
3. **TypeScript統合**: 型安全なコンポーネント開発
4. **パフォーマンス**: ツリーシェイキングによる最適化
5. **スケーラビリティ**: 大規模アプリケーション開発における保守性

参考資料

- Vue.js公式ドキュメント: https://vuejs.org/guide/extras/composition-api-faq.html
- Composition API RFC: https://github.com/vuejs/rfcs/blob/master/active-rfcs/0013-composition-api.md
- Evan You氏のブログ記事: https://blog.vuejs.org/posts/vue-3-as-the-new-default.html
- Vue Mastery講座: https://www.vuemastery.com/courses/vue-3-essentials/why-the-composition-api/
- VueUseライブラリ: https://vueuse.org/

ご清聴ありがとうございました