【新人エンジニア向け】Vue.js ライフサイクル完全ガイド - mount・破棄まで徹底解説

1. ライフサイクルとは？

Vue.jsの**ライフサイクル**とは、Vueコンポーネントが「生まれてから死ぬまで」の一連の過程のことです。人間で例えると「誕生→成長→老化→死亡」のような流れがあります。

コンポーネントの一生

```
作成準備 → 作成 → マウント → 更新 → 破棄
   ↓        ↓      ↓       ↓      ↓
beforeCreate created beforeMount updated beforeUnmount
              ↓      mounted   ↓      unmounted
            DOM生成   画面表示  DOM更新   メモリ解放
```

2. 基本的なライフサイクルフック

ステップ1: 最もシンプルな例

vue

```
<template>
  <!-- HTMLテンプレート部分 -->
  <div>
    <!-- メッセージを表示するエリア -->
    <h1>{{ message }}</h1>
    <!-- ボタンクリックでメッセージを変更 -->
    <button @click="changeMessage">メッセージ変更</button>
  </div>
</template>

<script>
export default {
  // コンポーネント名を定義
  name: 'LifecycleExample',

  // データプロパティを定義
  data() {
    return {
      // 画面に表示するメッセージ
      message: 'こんにちは、Vue.js!'
    }
  },

  // ===== ライフサイクルフック =====

  // 1. コンポーネントインスタンス作成前
  beforeCreate() {
    // この時点では data や methods にアクセスできない
    console.log('1. beforeCreate: コンポーネント作成準備中...')
    console.log('   this.message:', this.message) // undefined が表示される
  },

  // 2. コンポーネントインスタンス作成後
  created() {
    // data や methods にアクセス可能になる
    console.log('2. created: コンポーネント作成完了！')
    console.log('   this.message:', this.message) // 正常に表示される

    // APIからデータを取得する処理などをここに書く
    this.fetchDataFromAPI()
  },

  // 3. DOM にマウントする前
  beforeMount() {
    // まだ実際のDOMは作成されていない
    console.log('3. beforeMount: DOM作成前...')
```

```
    console.log('   DOMは未作成')
  },

  // 4. DOM にマウント完了後
  mounted() {
    // DOMが作成され、画面に表示される
    console.log('4. mounted: DOM作成完了！画面表示OK')
    console.log('   実際のDOM要素:', this.$el)

    // DOM操作が必要な処理をここに書く
    this.initializeDOM()
  },

  // 5. データ更新前
  beforeUpdate() {
    // データは変更されたが、まだDOMは更新されていない
    console.log('5. beforeUpdate: データ更新前...')
    console.log('   新しいメッセージ:', this.message)
  },

  // 6. データ更新後
  updated() {
    // データ変更がDOMに反映された
    console.log('6. updated: DOM更新完了！')
    console.log('   画面が更新されました')
  },

  // 7. コンポーネント破棄前
  beforeUnmount() {
    // まだコンポーネントは動作している
    console.log('7. beforeUnmount: 破棄準備中...')

    // クリーンアップ処理をここに書く
    this.cleanup()
  },

  // 8. コンポーネント破棄後
  unmounted() {
    // コンポーネントが完全に破棄された
    console.log('8. unmounted: 破棄完了')
    console.log('   さようなら...')
  },

  // ===== メソッド =====
  methods: {
    // メッセージを変更するメソッド
    changeMessage() {
```

```
      // この変更により beforeUpdate → updated が実行される
      this.message = '更新されました！'
    },

    // APIからデータを取得する想定のメソッド
    fetchDataFromAPI() {
      // 実際のAPI呼び出しの代わりにコンソール出力
      console.log('   → API からデータ取得中...')

      // 非同期処理をシミュレート
      setTimeout(() => {
        console.log('   → API データ取得完了')
      }, 1000)
    },

    // DOM初期化処理の想定メソッド
    initializeDOM() {
      // 例：スクロール位置の設定、フォーカスの設定など
      console.log('   → DOM初期化処理実行')
    },

    // クリーンアップ処理
    cleanup() {
      // 例：タイマーの削除、イベントリスナーの削除など
      console.log('   → クリーンアップ処理実行')
    }
  }
}
</script>
```

3. 実践的な使用例

ステップ2: API データ取得の例

vue

```
<template>
  <div>
    <!-- ローディング表示 -->
    <div v-if="loading">
      <p>データ読み込み中...</p>
    </div>

    <!-- エラー表示 -->
    <div v-else-if="error">
      <p style="color: red;">エラー: {{ error }}</p>
      <button @click="retryFetch">再試行</button>
    </div>

    <!-- データ表示 -->
    <div v-else>
      <h2>ユーザー一覧</h2>
      <ul>
        <li v-for="user in users" :key="user.id">
          {{ user.name }} ({{ user.email }})
        </li>
      </ul>
    </div>
  </div>
</template>

<script>
export default {
  name: 'UserList',

  data() {
    return {
      // ユーザーデータを格納する配列
      users: [],
      // ローディング状態を管理
      loading: true,
      // エラー情報を格納
      error: null
    }
  },

  // コンポーネント作成時にAPIからデータ取得
  async created() {
    console.log('created: ユーザーデータ取得開始')

    try {
      // ローディング開始
```

```
    this.loading = true
    this.error = null

    // APIからユーザーデータを取得
    await this.fetchUsers()

    console.log('created: ユーザーデータ取得完了')

  } catch (error) {
    // エラーハンドリング
    console.error('created: データ取得エラー', error)
    this.error = 'データの取得に失敗しました'

  } finally {
    // 最終的にローディングを終了
    this.loading = false
  }
},

methods: {
  // APIからユーザーデータを取得するメソッド
  async fetchUsers() {
    // 実際のAPI呼び出し（例：JSONPlaceholder）
    const response = await fetch('https://jsonplaceholder.typicode.com/users')

    // レスポンスのエラーチェック
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`)
    }

    // JSONデータを取得してusersに格納
    this.users = await response.json()
  },

  // 再試行メソッド
  async retryFetch() {
    // エラー状態をリセット
    this.error = null
    this.loading = true

    try {
      // データ取得を再実行
      await this.fetchUsers()
    } catch (error) {
      this.error = 'データの取得に再度失敗しました'
    } finally {
      this.loading = false
```

```
      }
    }
  }
}
</script>
```

ステップ3: DOM操作とクリーンアップの例

vue

```
<template>
  <div>
    <h2>リアルタイム時計</h2>
    <!-- 現在時刻を表示 -->
    <div class="clock">{{ currentTime }}</div>

    <!-- カウンター表示 -->
    <div class="counter">
      <p>カウンター: {{ counter }}</p>
      <button @click="startCounter">カウント開始</button>
      <button @click="stopCounter">カウント停止</button>
    </div>

    <!-- スクロール位置表示 -->
    <div class="scroll-info">
      <p>スクロール位置: {{ scrollY }}px</p>
    </div>

    <!-- 高さを作るためのダミーコンテンツ -->
    <div style="height: 2000px; background: linear-gradient(to bottom, #f0f0f0, #e0e0e0);">
      <p>スクロールしてみてください</p>
    </div>
  </div>
</template>

<script>
export default {
  name: 'ClockCounter',

  data() {
    return {
      // 現在時刻を格納
      currentTime: '',
      // カウンターの値
      counter: 0,
      // スクロール位置
      scrollY: 0,
      // タイマーIDを格納（クリーンアップ用）
      clockTimer: null,
      counterTimer: null
    }
  },

  // DOM作成完了後に初期化処理を実行
  mounted() {
    console.log('mounted: DOM作成完了、初期化開始')
```

```
    // 時計の初期化
    this.initializeClock()

    // スクロールイベントリスナーを追加
    this.addScrollListener()

    console.log('mounted: 初期化完了')
  },

  // コンポーネント破棄前にクリーンアップ
  beforeUnmount() {
    console.log('beforeUnmount: クリーンアップ開始')

    // タイマーをクリア
    this.clearTimers()

    // イベントリスナーを削除
    this.removeScrollListener()

    console.log('beforeUnmount: クリーンアップ完了')
  },

  methods: {
    // 時計の初期化
    initializeClock() {
      // 最初の時刻を設定
      this.updateTime()

      // 1秒ごとに時刻を更新するタイマーを設定
      this.clockTimer = setInterval(() => {
        this.updateTime()
      }, 1000) // 1000ミリ秒 = 1秒

      console.log('時計タイマー開始')
    },

    // 現在時刻を更新
    updateTime() {
      // 現在の日時を取得
      const now = new Date()

      // 時刻を「HH:MM:SS」形式でフォーマット
      this.currentTime = now.toLocaleTimeString('ja-JP')
    },

    // カウンター開始
```

```
startCounter() {
  // 既にタイマーが動いている場合は停止
  if (this.counterTimer) {
    this.stopCounter()
  }

  // 1秒ごとにカウンターを増加
  this.counterTimer = setInterval(() => {
    this.counter++
  }, 1000)

  console.log('カウンタータイマー開始')
},

// カウンター停止
stopCounter() {
  // タイマーをクリア
  if (this.counterTimer) {
    clearInterval(this.counterTimer)
    this.counterTimer = null
    console.log('カウンタータイマー停止')
  }
},

// スクロールイベントリスナーを追加
addScrollListener() {
  // スクロールイベントのハンドラー関数を定義
  this.handleScroll = () => {
    // 現在のスクロール位置を取得
    this.scrollY = window.scrollY
  }

  // windowオブジェクトにスクロールイベントリスナーを追加
  window.addEventListener('scroll', this.handleScroll)

  console.log('スクロールイベントリスナー追加')
},

// スクロールイベントリスナーを削除
removeScrollListener() {
  // イベントリスナーを削除（メモリリーク防止）
  if (this.handleScroll) {
    window.removeEventListener('scroll', this.handleScroll)
    console.log('スクロールイベントリスナー削除')
  }
},
```

```
    // 全タイマーをクリア
    clearTimers() {
      // 時計タイマーをクリア
      if (this.clockTimer) {
        clearInterval(this.clockTimer)
        this.clockTimer = null
        console.log('時計タイマー削除')
      }

      // カウンタータイマーをクリア
      if (this.counterTimer) {
        clearInterval(this.counterTimer)
        this.counterTimer = null
        console.log('カウンタータイマー削除')
      }
    }
  }
}
</script>

<style scoped>
.clock {
  font-size: 2rem;
  font-weight: bold;
  color: #2c3e50;
  margin: 20px 0;
}

.counter {
  margin: 20px 0;
}

.counter button {
  margin: 0 5px;
  padding: 8px 16px;
  background-color: #3498db;
  color: white;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

.counter button:hover {
  background-color: #2980b9;
}

.scroll-info {
```

```
    position: fixed;
    top: 10px;
    right: 10px;
    background-color: rgba(0, 0, 0, 0.8);
    color: white;
    padding: 10px;
    border-radius: 4px;
  }
  </style>
```

4. よくある使用パターン

ステップ4: フォームバリデーションの例

vue

```
<template>
  <div>
    <h2>ユーザー登録フォーム</h2>

    <form @submit.prevent="submitForm">
      <!-- 名前入力 -->
      <div class="form-group">
        <label for="name">名前:</label>
        <input
          id="name"
          v-model="form.name"
          type="text"
          :class="{ error: errors.name }"
          @blur="validateName"
        />
        <span v-if="errors.name" class="error-message">{{ errors.name }}</span>
      </div>

      <!-- メール入力 -->
      <div class="form-group">
        <label for="email">メール:</label>
        <input
          id="email"
          v-model="form.email"
          type="email"
          :class="{ error: errors.email }"
          @blur="validateEmail"
        />
        <span v-if="errors.email" class="error-message">{{ errors.email }}</span>
      </div>

      <!-- パスワード入力 -->
      <div class="form-group">
        <label for="password">パスワード:</label>
        <input
          id="password"
          v-model="form.password"
          type="password"
          :class="{ error: errors.password }"
          @blur="validatePassword"
        />
        <span v-if="errors.password" class="error-message">{{ errors.password }}</span>
      </div>

      <!-- 送信ボタン -->
      <button type="submit" :disabled="!isFormValid || isSubmitting">
```

```
        {{ isSubmitting ? '送信中...' : '登録' }}
      </button>
    </form>
  </div>
</template>

<script>
export default {
  name: 'UserRegistrationForm',

  data() {
    return {
      // フォームデータ
      form: {
        name: '',
        email: '',
        password: ''
      },

      // バリデーションエラー
      errors: {
        name: null,
        email: null,
        password: null
      },

      // 送信中フラグ
      isSubmitting: false
    }
  },

  // フォームの初期設定
  mounted() {
    console.log('mounted: フォーム初期化')

    // 名前フィールドにフォーカスを設定
    this.focusFirstField()

    // フォームの自動保存機能を初期化
    this.initializeAutoSave()
  },

  // データ更新時にバリデーション実行
  updated() {
    console.log('updated: フォームデータが更新されました')

    // リアルタイムバリデーション
```

```
  this.validateForm()
},

// コンポーネント破棄前に未保存データの警告
beforeUnmount() {
  console.log('beforeUnmount: フォーム破棄前チェック')

  // 未保存データがある場合の警告
  this.checkUnsavedData()

  // 自動保存タイマーをクリア
  this.clearAutoSaveTimer()
},

computed: {
  // フォーム全体のバリデーション状態
  isFormValid() {
    // 全フィールドにエラーがなく、かつ全て入力済み
    return !this.errors.name &&
           !this.errors.email &&
           !this.errors.password &&
           this.form.name.trim() !== '' &&
           this.form.email.trim() !== '' &&
           this.form.password.trim() !== ''
  }
},

methods: {
  // 最初のフィールドにフォーカス
  focusFirstField() {
    // DOM操作のため、nextTick で確実にDOM更新後に実行
    this.$nextTick(() => {
      const nameInput = document.getElementById('name')
      if (nameInput) {
        nameInput.focus()
        console.log('名前フィールドにフォーカス設定')
      }
    })
  },

  // 名前のバリデーション
  validateName() {
    const name = this.form.name.trim()

    // 空文字チェック
    if (!name) {
      this.errors.name = '名前は必須です'
```

```
    return false
  }

  // 文字数チェック
  if (name.length < 2) {
    this.errors.name = '名前は2文字以上で入力してください'
    return false
  }

  if (name.length > 50) {
    this.errors.name = '名前は50文字以内で入力してください'
    return false
  }

  // バリデーション成功
  this.errors.name = null
  return true
},

// メールのバリデーション
validateEmail() {
  const email = this.form.email.trim()

  // 空文字チェック
  if (!email) {
    this.errors.email = 'メールアドレスは必須です'
    return false
  }

  // メール形式チェック
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/
  if (!emailRegex.test(email)) {
    this.errors.email = '正しいメールアドレス形式で入力してください'
    return false
  }

  // バリデーション成功
  this.errors.email = null
  return true
},

// パスワードのバリデーション
validatePassword() {
  const password = this.form.password

  // 空文字チェック
  if (!password) {
```

```
    this.errors.password = 'パスワードは必須です'
    return false
  }

  // 文字数チェック
  if (password.length < 8) {
    this.errors.password = 'パスワードは8文字以上で入力してください'
    return false
  }

  // 複雑さチェック（英数字を含む）
  const hasLetter = /[a-zA-Z]/.test(password)
  const hasNumber = /[0-9]/.test(password)

  if (!hasLetter || !hasNumber) {
    this.errors.password = 'パスワードは英字と数字を含む必要があります'
    return false
  }

  // バリデーション成功
  this.errors.password = null
  return true
},

// フォーム全体のバリデーション
validateForm() {
  // 各フィールドのバリデーションを実行
  const nameValid = this.validateName()
  const emailValid = this.validateEmail()
  const passwordValid = this.validatePassword()

  // 全て有効な場合のみtrueを返す
  return nameValid && emailValid && passwordValid
},

// フォーム送信処理
async submitForm() {
  console.log('submitForm: フォーム送信開始')

  // 送信前の最終バリデーション
  if (!this.validateForm()) {
    console.log('バリデーションエラーのため送信中止')
    return
  }

  try {
    // 送信中状態に設定
```

```
    this.isSubmitting = true

    // API送信（模擬）
    console.log('API送信中...')
    await this.sendToAPI(this.form)

    // 送信成功
    console.log('送信成功！')
    alert('ユーザー登録が完了しました！')

    // フォームをリセット
    this.resetForm()

  } catch (error) {
    // エラーハンドリング
    console.error('送信エラー:', error)
    alert('送信に失敗しました。もう一度お試しください。')

  } finally {
    // 送信中状態を解除
    this.isSubmitting = false
  }
},

// API送信（模擬）
async sendToAPI(userData) {
  // 実際のAPI呼び出しを模擬
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // 90%の確率で成功
      if (Math.random() > 0.1) {
        resolve({ success: true, message: '登録完了' })
      } else {
        reject(new Error('サーバーエラー'))
      }
    }, 2000) // 2秒の遅延を模擬
  })
},

// フォームリセット
resetForm() {
  // フォームデータをクリア
  this.form.name = ''
  this.form.email = ''
  this.form.password = ''

  // エラーメッセージをクリア
```

```
    this.errors.name = null
    this.errors.email = null
    this.errors.password = null

    // 最初のフィールドにフォーカス
    this.focusFirstField()

    console.log('フォームをリセットしました')
  },

  // 自動保存機能の初期化
  initializeAutoSave() {
    // 30秒ごとに自動保存
    this.autoSaveTimer = setInterval(() => {
      this.autoSaveFormData()
    }, 30000) // 30秒

    console.log('自動保存機能を開始')
  },

  // フォームデータの自動保存
  autoSaveFormData() {
    // ローカルストレージに保存（実際のアプリでは適切な保存先を選択）
    const formData = JSON.stringify(this.form)
    localStorage.setItem('userRegistrationForm', formData)

    console.log('フォームデータを自動保存しました')
  },

  // 未保存データのチェック
  checkUnsavedData() {
    // フォームに入力データがある場合
    const hasData = this.form.name || this.form.email || this.form.password

    if (hasData && !this.isSubmitting) {
      // 本来はconfirmダイアログで確認すべき
      console.warn('未保存のデータがあります')
    }
  },

  // 自動保存タイマーのクリア
  clearAutoSaveTimer() {
    if (this.autoSaveTimer) {
      clearInterval(this.autoSaveTimer)
      this.autoSaveTimer = null
      console.log('自動保存タイマーを停止')
    }
```

```
    }
  }
}
</script>

<style scoped>
.form-group {
  margin-bottom: 20px;
}

label {
  display: block;
  margin-bottom: 5px;
  font-weight: bold;
}

input {
  width: 100%;
  padding: 8px;
  border: 1px solid #ddd;
  border-radius: 4px;
  font-size: 16px;
}

input.error {
  border-color: #e74c3c;
  background-color: #fdf2f2;
}

.error-message {
  color: #e74c3c;
  font-size: 14px;
  margin-top: 5px;
  display: block;
}

button {
  background-color: #3498db;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 16px;
}

button:disabled {
```

```
  background-color: #bdc3c7;
  cursor: not-allowed;
}

button:hover:not(:disabled) {
  background-color: #2980b9;
}
</style>
```

5. ライフサイクルの注意点とベストプラクティス

よくある間違いと対策

javascript

```
// ❌ 悪い例：メモリリークの原因
export default {
  mounted() {
    // タイマーを設定するが、クリーンアップしない
    setInterval(() => {
      console.log('動作中...')
    }, 1000)

    // イベントリスナーを追加するが、削除しない
    window.addEventListener('resize', this.handleResize)
  }
  // beforeUnmount でクリーンアップしていない！
}

// ✅ 良い例：適切なクリーンアップ
export default {
  data() {
    return {
      timer: null
    }
  },

  mounted() {
    // タイマーIDを保存
    this.timer = setInterval(() => {
      console.log('動作中...')
    }, 1000)

    // イベントリスナーを追加
    window.addEventListener('resize', this.handleResize)
  },

  beforeUnmount() {
    // タイマーをクリア
    if (this.timer) {
      clearInterval(this.timer)
    }

    // イベントリスナーを削除
    window.removeEventListener('resize', this.handleResize)
  },

  methods: {
    handleResize() {
      // リサイズ処理
    }
```

```
  }
}
```

パフォーマンス最適化

```javascript
export default {
  // 頻繁に呼ばれるupdatedの代わりにwatchを使用
  data() {
    return {
      searchQuery: '',
      results: []
    }
  },

  // ❌ 悪い例：updatedで毎回処理
  updated() {
    // データが変わるたびに呼ばれてしまう
    if (this.searchQuery) {
      this.performSearch()
    }
  },

  // ✅ 良い例：watchで特定のデータのみ監視
  watch: {
    searchQuery(newQuery, oldQuery) {
      // searchQueryが変更された時のみ実行
      if (newQuery !== oldQuery) {
        this.performSearch()
      }
    }
  },

  methods: {
    performSearch() {
      // 検索処理
    }
  }
}
```

6. まとめ

ライフサイクルフック使い分けガイド

フック	主な用途	注意点
created	API呼び出し、データ初期化	DOMはまだ存在しない
mounted	DOM操作、外部ライブラリ初期化	DOM操作が可能
updated	DOM更新後の処理	頻繁に呼ばれる可能性
beforeUnmount	クリーンアップ処理	必ずメモリリークを防ぐ

開発のベストプラクティス

1. **API呼び出しは created で**
   - DOM操作が不要なデータ取得
   - 早期のデータロード
2. **DOM操作は mounted で**
   - 要素への参照が必要な処理
   - 外部ライブラリの初期化
3. **必ずクリーンアップを**
   - タイマーのクリア
   - イベントリスナーの削除
   - メモリリークの防止
4. **パフォーマンスを意識**
   - updated の多用を避ける
   - watch の活用
   - 必要な時のみ処理実行

Vue.jsのライフサイクルを理解することで、適切なタイミングで処理を実行し、パフォーマンスの良いWebアプリケーションを開発できるようになります！