

SVELTE Docs - 日本語



Table of contents

• Introduction	3
• Getting started	4
• Component format	5
• Template syntax	12
• Run time	39
• Compile time	59
• Accessibility warnings	65

始める前に (Before we begin)

このページには、API リファレンスの詳細なドキュメントが含まれています。これは既に Svelte にある程度親しんでいる人のためのリソースとなることを意図しています。

もしあなたが（まだ）そうでないのであれば、このリファレンスを参照する前に [対話型のチュートリアル](#) や [example](#) を参照することを推奨します。

恥ずかしがらずに [Discord chatroom](#) で助けを求めるましょう。

古いバージョンの Svelte をお使いですか？ [v2 docs](#) をご覧ください。

日本語翻訳版 追記：上記のDiscordはSvelte本体のもので、英語でコミュニケーションが行われています。もし日本語で質問したり交流したいのであれば、[Svelte日本のDiscord](#)にどうぞ！

Getting started

インタラクティブなオンライン環境で Svelte を試す場合は、[こちらの REPL](#) か [StackBlitz](#) をお試しください。

ローカルにプロジェクトを作成する場合は、以下を実行してください：

```
npm create vite@latest myapp -- --template svelte
cd myapp
npm install
npm run dev
```

もしくは、[SvelteKit](#) をお使いください。Svelte チームによる公式アプリケーションフレームワークです（現時点はリリース候補版(release candidate status)です）。

Svelte チームでは [VS Code extension](#) をメンテナンスしております。そして Svelte コミュニティではその他様々な種類の [ツールやエディタ](#) のインテグレーションのリストも作成しています。

お困りの場合は、[Discord](#) や [StackOverflow](#) でご質問ください。

Component format

コンポーネントは、Svelteアプリケーションを構成するブロックです。これらは `.svelte` ファイルにHTMLのスーパーセットを使って記述されます。

ここで説明される `script`、`style`、マークアップのいずれもコンポーネントに必須のものではありません。

```
<script>
  // ロジックを記述
</script>

<!-- 0個以上のマークアップを記述 -->

<style>
  /* styleを記述 */
</style>
```

<script>

`<script>` ブロックは、コンポーネントのインスタンスが生成されるときに実行される JavaScript を含みます。トップレベルで宣言（またはインポート）された変数は、そのコンポーネントのマークアップから '見る' ことができます。`<script>` には、4つのルールがあります。

1. `export` creates a component prop

Svelte では、変数の宣言を プロパティ (*prop*) としてマークするために `export` キーワードを使います。これによってそのコンポーネントを使用する際にその変数にアクセスできるようになります（より詳しい情報は [属性とプロパティ](#) を見てください）。

```
<script>
  export let foo;

  // プロパティとして渡された変数は、
  // 即座に使用可能になります
  console.log({ foo });
</script>
```

プロパティはデフォルトの初期値を指定することができます。これはコンポーネントの初期化時にプロパティが指定されていない場合（または初期値が `undefined` の場合）に使用されます。プロパティを削除すると、その値は初期値ではなく `undefined` になることに注意してください。

development モード（[コンパイラオプション](#)を参照）では、デフォルトの初期値が指定されておらず、使用時に値を指定していない場合警告が表示されます。この警告を解消するためには、たとえ `undefined` であってもデフォルトの初期値を指定してください。

```
<script>
  export let bar = 'optional default initial value';
  export let baz = undefined;
</script>
```

`const` や `class`、`function` をエクスポートすると、コンポーネントの外からは読み取り専用になります。ただし、以下で示すように、関数は有効なプロパティ値(valid prop values)です。

```
<script>
  // これらは読み取り専用です
  export const thisIs = 'readonly';

  export function greet(name) {
    alert(`hello ${name}`);
  }

  // これはプロパティです
  export let format = n => n.toFixed(2);
</script>
```

読み取り専用のプロパティは要素のプロパティとしてアクセスでき、`bind:this` 構文を使用してコンポーネントに結び付けられます。

予約語もプロパティの名前として使用することができます。

```
<script>
  let className;

  // `class` は予約語ですが、
  // `class` プロパティを作ることができます
  export { className as class };
</script>
```

2. Assignments are 'reactive'(代入は'リアクティブ')

コンポーネントの状態を変更して再レンダリングをトリガーするために必要なのは、ローカルで宣言された変数に代入することだけです。

更新式 (`count += 1`) とプロパティの代入 (`obj.x = y`) には同じ効果があります。

```
<script>
  let count = 0;

  function handleClick () {
    // マークアップが `count`、を参照している場合、
    // この関数を呼び出すと更新がトリガーされます
    count = count + 1;
  }
</script>
```

Svelteのリアクティビティは代入に基づいているため、`.push()` や `.splice()` のような配列のメソッドを使用しても自動的に更新をトリガーしません。これを回避する方法は[チュートリアル](#)に記載しています。

```
<script>
  let arr = [0, 1];

  function handleClick () {
    // this method call does not trigger an update
    arr.push(2);
    // this assignment will trigger an update
    // if the markup references `arr`
```

```
    arr = arr
}
</script>
```

Svelteの `<script>` ブロックはコンポーネントが作成されたときのみ実行されるため、`<script>` ブロック内の代入は、プロパティの更新時に自動で再実行されません。プロパティの変更を追跡したい場合は、次のセクションの例をご覧ください。

```
<script>
  export let person;
  // this will only set `name` on component creation
  // it will not update when `person` does
  let { name } = person;
</script>
```

3. `$:` marks a statement as reactive

トップレベルの（つまりブロック内や関数内でない）ステートメントは `$:` という JS ラベル構文 の接頭辞をつけることでリアクティブにできます。リアクティブステートメントは、他のスクリプトコードの後、かつコンポーネントのマークアップがレンダリングされる前に実行されます。また、依存する値が変更されるたびにも実行されます。

```
<script>
  export let title;
  export let person

  // これは `title` プロパティが変わるたびに
  // `document.title` を更新します
  $: document.title = title;

  $: {
    console.log(`複数のステートメントをまとめることができます`);
    console.log(`現在のタイトルは ${title}`);
  }

  // this will update `name` when 'person' changes
  $: ({ name } = person);

  // don't do this. it will run before the previous line
  let name2 = name;
</script>
```

`$:` のブロック内に直接現れる値だけが、リアクティブステートメントが依存しているものになります。例えば次のコードで `total` は `x` が変更された時にのみ更新され、`y` では更新されません。

```
<script>
  let x = 0;
  let y = 0;

  function yPlusAValue(value) {
    return value + y;
  }

  $: total = yPlusAValue(x);
</script>
```

```
Total: {total}
<button on:click={() => x++}>
```

```
    Increment X
</button>

<button on:click={() => y++}>
  Increment Y
</button>
```

It is important to note that the reactive blocks are ordered via simple static analysis at compile time, and all the compiler looks at are the variables that are assigned to and used within the block itself, not in any functions called by them. This means that `yDependent` will not be updated when `x` is updated in the following example:

```
<script>
  let x = 0;
  let y = 0;

  const setY = (value) => {
    y = value;
  }

  $: yDependent = y;
  $: setY(x);
</script>
```

Moving the line `$: yDependent = y` below `$: setY(x)` will cause `yDependent` to be updated when `x` is updated.

宣言されていない変数への代入だけでステートメントが構成されている場合、Svelte はあなたの代わりに `let` 宣言を挿入します。

```
<script>
  export let num;

  // `squared` や `cubed` を宣言する必要はありません
  // - Svelte がやってくれます
  $: squared = num * num;
  $: cubed = squared * num;
</script>
```

4. Prefix stores with `$` to access their values

ストアは、シンプルなストアコントラクト(store contract)を介して値へのリアクティブなアクセスを可能にするオブジェクトです。`svelte/store` モジュールにはこのコントラクト(contract)を満たす最小限のストア実装が含まれています。

ストアへの参照を持っているときはいつでも、`$`を接頭辞として付けることで、コンポーネント内からその値にアクセスできます。これによってSvelteは接頭辞付きの変数を宣言し、ストアのサブスクリプションを設定します。このサブスクリプションは適切なタイミングで解除されます。

`$`接頭辞が付いた変数に代入するには、その変数が書き込み可能なストアである必要があります。また、代入時にはストアの `.set` メソッドが呼び出されます。

ストアはコンポーネントのトップレベルで宣言しなければいけないことに注意してください。例えば、`if` ブロックや関数の中では宣言できません。

(ストア値を表すものではない)ローカル変数には、`$`接頭辞を付けてはいけません。

```
<script>
  import { writable } from 'svelte/store';

  const count = writable(0);
  console.log($count); // logs 0

  count.set(1);
  console.log($count); // logs 1

  $count = 2;
  console.log($count); // logs 2
</script>
```

Store contract

```
store = { subscribe: (subscription: (value: any) => void) => (() => void), set?: (value: any) => void }
```

ストアコントラクト(store contract)を実装することで、`svelte/store` に依存せずに独自のストアを作ることができます。

- ストアは `.subscribe` メソッドを含まなければならず、その引数としてサブスクリプション関数を受けとる必要があります。このサブスクリプション関数は `.subscribe` が呼ばれたらストアの現在の値と同期して即座に呼び出されなければいけません。ストアのアクティブなサブスクリプション関数は全て、ストアの値が変更されるたびに同期して呼び出されなければいけません。
- `.subscribe` メソッドはサブスクリプションを解除する関数を返さなければいけません。サブスクリプションを解除する関数が呼ばれたら、そのサブスクリプションを停止してそれに対応するサブスクリプション関数がそのストアから再び呼び出されないようにしなければいけません。
- ストアはオプションで `.set` メソッドを含むことができます。`.set` メソッドは、引数としてストアの新しい値を受ける必要があり、全てのアクティブなサブスクリプション関数を同期的に呼び出します。このようなストアは `書き込み可能なストア (writable store)` と呼ばれます。

RxJSのObservablesとの相互運用性のため、`.subscribe` メソッドはサブスクリプションを解除する関数を直接返すのではなく、`.unsubscribe` メソッドを持つオブジェクトを返すこともできます。ただし、`.subscribe` が同期的にサブスクリプションを呼び出さない限り(これはObservableの仕様で要求されていませんが)、サブスクリプションを呼び出すまでは、Svelte がストアの値を `undefined` とみなすことに注意してください。

<script context="module">

`context="module"` 属性をもつ `<script>` タグは、コンポーネントインスタンスごとではなく、モジュールが最初に評価するときに1回実行されます。このブロックで宣言された値は、通常の `<script>` (およびコンポーネントのマークアップ) からアクセスできますが、その逆はできません。

このブロックからバインディングを `export` でき、それらはコンパイルされたモジュールのエクスポートになります。

デフォルトのエクスポートはコンポーネント自体であるため、`export default` はできません。

`module` スクリプトで定義された変数はリアクティブではありません。つまり、変数の再代入は、変数自体の更新はしますが、再レンダリングのトリガーにはなりません。複数のコンポーネント間で共有される値については、[スコープ](#)の使用を検討してください。

```
<script context="module">
  let totalComponents = 0;

  // this allows an importer to do e.g.
  // `import Example, { alertTotal } from './Example.svelte'`
  export function alertTotal() {
    alert(totalComponents);
  }
</script>

<script>
  totalComponents += 1;
  console.log(`total number of times this component has been created:
${totalComponents}`);
</script>
```

<style>

`<style>` ブロック内の CSS は、そのコンポーネントにスコープされます。

これは、影響を受ける要素にクラスを追加することで動作し、そのクラスはコンポーネントのスタイルのハッシュに基づいています (例えば `svelte-123xyz`)。

```
<style>
  p {
    /* これはこのコンポーネントの <p> 要素にのみ影響します */
    color: burlywood;
  }
</style>
```

スタイルをグローバルなセレクタに適用するには、`:global(...)` 修飾子を使用します。

```
<style>
  :global(body) {
    /* これは <body> に適用されます */
    margin: 0;
  }

  div :global(strong) {
    /* これは、このコンポーネント内の <div> 要素の中にある
       任意のコンポーネント内の <strong> 要素に
       適用されます */
    color: goldenrod;
  }

  p:global(.red) {
    /* これは、このコンポーネントに属し、red クラスを持つ
       すべての <p> 要素に適用されます (class="red" が
       最初のマークアップに現れず、実行時に追加された場合
       でも)。これは要素の classList プロパティを直接
       更新するなど、要素のクラスが動的に適用されるときに
```

```
    便利です。 */
}
```

グローバルにアクセスできる @keyframes を作りたい場合は、キーフレーム名の前に `-global-` を付ける必要があります。

コンパイル時に `-global-` の部分は削除され、キーフレームはコード内の他の箇所では `my-animation-name` だけを使って参照されます。

```
<style>
  @keyframes -global-my-animation-name {...}
</style>
```

トップレベルの `<style>` タグは、1 つのコンポーネントにつき 1 つだけでなければなりません。

ただし、他の要素や論理ブロックの中に `<style>` タグを入れ子にすることは可能です。

その場合、`<style>` タグはそのまま DOM に挿入され、`<style>` タグのスコープや処理は行われません。

```
<div>
  <style>
    /* この style タグはそのまま挿入されます */
    div {
      /* これは DOM 内のすべての `<div>` 要素に適用されます */
      color: red;
    }
  </style>
</div>
```

Template syntax

Tags

`<div>` のような小文字のタグは、通常の HTML 要素を表します。大文字のタグ、例えば `<Widget>` や `<Namespace.Widget>` は コンポーネントを表します。

```
<script>
  import Widget from './Widget.svelte';
</script>

<div>
  <Widget/>
</div>
```

Attributes and props

デフォルトでは、属性はHTMLと全く同じように動作します。

```
<div class="foo">
  <button disabled>can't touch this</button>
</div>
```

HTMLのように、値は引用符で囲まれていない場合があります。

```
<input type=checkbox>
```

属性値には JavaScript の式を含めることができます。

```
<a href="page/{p}">page {p}</a>
```

あるいは、JavaScript の式にすることもできます。

```
<button disabled={!clickable}>...</button>
```

Boolean の属性は、その値が `truthy` であれば要素に含まれ、`falsy` であれば除外されます。

それ以外の属性は、その値が `nullish` (`null` または `undefined`) でない限り含まれます。

```
<input required={false} placeholder="This input field is not required">
<div title={null}>This div has no title attribute</div>
```

式には、通常の HTML ではシンタックスハイライトに失敗するような文字が含まれている可能性があるので、値を引用符で囲むことが許可されています。引用符は値の解析方法には影響しません。

```
<button disabled="{number !== 42}">...</button>
```

属性名と値が一致する場合(`name={name}`)は、`{name}` で置き換えることができます。

```
<!-- These are equivalent -->
<button disabled={disabled}>...</button>
<button {disabled}>...</button>
```

慣習として、コンポーネントに渡される値は DOM の機能である *属性* ではなく、*プロパティ* または *props* と呼ばれます。

要素の場合と同様に、`name={name}` は `{name}` の短縮形に置き換えることができます。

```
<Widget foo={bar} answer={42} text="hello"/>
```

スプレッド属性は、多くの属性やプロパティを一度に要素やコンポーネントに渡すことを可能にします。

要素またはコンポーネントは、通常のものと混在させて、複数のスプレッド属性を持つことができます。

```
<Widget {...things}/>
```

`$$props` は、`export` で宣言されていないものも含めて、コンポーネントに渡されるすべてのプロパティ(props)を参照します。これは Svelte の最適化が難しいので、一般的には推奨されません。しかし、コンパイル時にどのようなプロパティがコンポーネントに渡されるかわからない場合など、稀なケースでは便利です。

```
<Widget {...$$props}/>
```

`$$restProps` には、`export` で宣言されていないプロパティ(props)のみが含まれます。これは他の未知の属性をコンポーネントの要素に渡すために使用できます。`$$props` と同じ最適化の問題を共有しており、同様に推奨されません。

```
<input {...$$restProps}>
```

`input` 要素やその子要素である `option` 要素の `value` 属性は、`bind:group` や `bind:checked` を使用している場合、スプレッド属性で設定してはいけません。このような場合、バインドされる変数にリンクできるように、Svelteがその要素の `value` をマークアップの中で直接見ることができます。

Text expressions

```
{expression}
```

テキストにもJavaScriptの式を含めることができます。

正規表現 (RegExp) の [リテラル記法](#)、括弧で囲う必要があります。

```
<h1>Hello {name}!</h1>
<p>{a} + {b} = {a + b}.</p>

<div>{(/^[A-Za-z ]+$/).test(value) ? x : y}</div>
```

Comments

コンポーネント内でHTMLコメントを使用することができます。

```
<!-- this is a comment! -->
<h1>Hello world</h1>
```

`svelte-ignore` で始まるコメントは、マークアップの次のブロックに対する警告を無効にします。通常、これらはアクセシビリティの警告です。正当な理由で警告を無効にしていることを確認してください。

```
<!-- svelte-ignore a11y-autofocus -->
<input bind:value={name} autofocus>
```

{#if ...}

```
{#if 式}...{/if}
```

```
{#if 式}...{:else if 式}...{/if}
```

```
{#if 式}...{:else}...{/if}
```

条件付きでレンダリングされるコンテンツは、if ブロックで囲むことができます。

```
{#if answer === 42}
  <p>what was the question?</p>
{/if}
```

追加の条件は `{:else if 式}` で付け足すことができ、`{:else}` 句で終わらせることもできます。

```
{#if porridge.temperature > 100}
  <p>too hot!</p>
{:else if 80 > porridge.temperature}
  <p>too cold!</p>
{:else}
  <p>just right!</p>
{/if}
```

{#each ...}

```
{#each 式 as name}...{/each}
```

```
{#each 式 as name, index}...{/each}
```

```
{#each 式 as name (key)}...{/each}
```

```
{#each 式 as name, index (key)}...{/each}
```

```
{#each 式 as name}...{:else}...{/each}
```

each ブロックで値のリストの反復処理ができます。

```
<h1>Shopping list</h1>
<ul>
  {#each items as item}
    <li>{item.name} x {item.qty}</li>
  {/each}
</ul>
```

配列や配列のような値（つまり `length` プロパティを持つオブジェクト）を反復処理するのに `each` ブロックを使用できます。

`each` ブロックは `array.map(...)` のコールバックの第2数に相当するインデックスを指定することもできます。

```
{#each items as item, i}
  <li>i + 1: {item.name} x {item.qty}</li>
{/each}
```

`key` の式（各リストアイテムを一意に識別できる必要があります）が与えられた場合、Svelte は、データが変化したときに（末尾にアイテムを追加したり削除するのではなく）キーを使用してリストの差分を取ります。`key` はどんなオブジェクトでもよいですが、そのオブジェクト自体が変更されたときに同一性を維持できるため、文字列か数値をお勧めします。

```
{#each items as item (item.id)}
  <li>{item.name} x {item.qty}</li>
{/each}


{#each items as item, i (item.id)}
  <li>i + 1: {item.name} x {item.qty}</li>
{/each}
```

`each` ブロックでは分割代入や残余構文のパターンを自由に使えます。

```
{#each items as { id, name, qty }, i (id)}
  <li>i + 1: {name} x {qty}</li>
{/each}

{#each objects as { id, ...rest }}
  <li><span>{id}</span><MyComponent {...rest}></li>
{/each}

{#each items as [id, ...rest]}
  <li><span>{id}</span><MyComponent values={rest}></li>
{/each}
```

`each` ブロックには `:else` 句を入れることもできます。これはリストが空の場合にレンダリングされます。

```
{#each todos as todo}
  <p>{todo.text}</p>
{:else}
  <p>No tasks today!</p>
{/each}
```

{#await ...}

```
{#await 式}...{:then name}...{:catch name}...{/await}
```

```
{#await 式}...{:then name}...{/await}
```

```
{#await 式 then name}...{/await}
```

```
{#await 式 catch name}...{/await}
```

`await` ブロックを使用すると、Promise が取りうる 3 つの状態 (pending(保留中)、fulfilled(成功)、rejected(失敗)) に分岐できます。

```
{#await promise}
  <!-- promise is pending -->
  <p>waiting for the promise to resolve...</p>
{:then value}
  <!-- promise was fulfilled -->
  <p>The value is {value}</p>
{:catch error}
  <!-- promise was rejected -->
  <p>Something went wrong: {error.message}</p>
{/await}
```

promise が失敗した時に何もレンダリングする必要がない場合（もしくはエラーが発生しない場合）は `catch` ブロックを省略できます。

```
{#await promise}
  <!-- promise is pending -->
  <p>waiting for the promise to resolve...</p>
{:then value}
  <!-- promise was fulfilled -->
  <p>The value is {value}</p>
{/await}
```

保留中の状態を気にしない場合は、最初のブロックを省略することもできます。

```
{#await promise then value}
  <p>The value is {value}</p>
{/await}
```

同様に、エラー状態のみを表示したい場合は `then` ブロックを省略できます。

```
{#await promise catch error}
  <p>The error is {error}</p>
{/await}
```

{#key ...}

```
{#key 式}...{/key}
```

`key` ブロックは式の値が変更されたときに、その中身を破棄して再作成します。

これは、ある値が変更されるたびに要素のトランジションを再生したい場合に便利です。

```
{#key value}
  <div transition:fade>{value}</div>
{/key}
```

コンポーネントを囲んで使用した場合、コンポーネントの再インスタンス化と再初期化をもたらします。

```
{#key value}
  <Component />
{/key}
```

{@html ...}

{@html 式}

テキスト式（{式} の構文）では、< や > のような文字はエスケープされますが、HTML 式ではエスケープされません。

式は単独で正しい HTML になっている必要があります。{@html "<div>"}content{@html "</div>"} は </div> の部分が正しい HTML ではないため、動作しません。また、Svelteコードをコンパイルすることもできません。

Svelte は HTML を挿入する前に式をサニタイズしません。データが信頼できないソースからのものである場合は自分でサニタイズする必要があります。そうしないと、ユーザーを XSS の脆弱性にさらしてしまいます。

```
<div class="blog-post">
  <h1>{post.title}</h1>
  {@html post.content}
</div>
```

{@debug ...}

{@debug}

{@debug 変数1, 変数2, ..., 変数N}

{@debug ...} タグは console.log(...) の代わりになります。指定した変数の値が変更されるたびログに出力し、devtools が開いているとコードの実行を一時停止します。

```
<script>
  let user = {
    firstname: 'Ada',
    lastname: 'Lovelace'
  };
</script>

{@debug user}

<h1>Hello {user.firstname}!</h1>
```

{@debug ...} はカンマ区切りの（任意の式ではなく）変数名のリストを受け取ります。

```
<!-- コンパイルされる -->
{@debug user}
{@debug user1, user2, user3}

<!-- コンパイルできない -->
{@debug user.firstname}
```

```
{@debug myArray[0]}
{@debug !isReady}
{@debug typeof user === 'object'}
```

引数なしの `{@debug}` タグは、(変数を指定した場合とは逆に) 状態のどれかが変化した時にトリガされる `debugger` 文を挿入します。

`{@const ...}`

```
{@const assignment}
```

`{@const ...}` タグはローカル定数を定義します。

```
<script>
  export let boxes;
</script>

{-#each boxes as box}
  {@const area = box.width * box.height}
  {box.width} * {box.height} = {area}
{/each}
```

`{@const}` は、`{#if}`、`{:else if}`、`{:else}`、`{#each}`、`{:then}`、`{:catch}`、`<Component />`、`<svelte:fragment />` の直下にのみ配置することができます。

Element directives

要素には、属性と同じようにディレクティブを持たせることができます。これは何らかの方法で要素の動作を制御します。

`on:eventname`

```
on:eventname={handler}
```

```
on:eventnamemodifiers={handler}
```

DOM イベントをリッスンするには `on:` ディレクティブを使用します。

```
<script>
  let count = 0;

  function handleClick(event) {
    count += 1;
  }
</script>

<button on:click={handleClick}>
  count: {count}
</button>
```

ハンドラはパフォーマンスを低下させることなくインラインで宣言できます。 属性と同様、ディレクティブの値はシンタックスハイライトのために引用符で囲むことができます。

```
<button on:click="{() => count += 1}">
  count: {count}
</button>
```

| の文字を使って DOM イベントに修飾子を追加します。

```
<form on:submit|preventDefault={handleSubmit}>
  <!-- `submit` イベントの規定の動作が妨げられ
      ページはリロードされません -->
</form>
```

次の修飾子を使用できます:

- `preventDefault` — ハンドラを実行する前に `event.preventDefault()` を呼び出します
- `stopPropagation` — `event.stopPropagation()` を呼び出し、イベントが次の要素に到達するのを防ぎます
- `passive` — タッチ/ホイールイベントのスクロールパフォーマンスを向上させます (Svelte は追加することが安全な箇所には自動的に追加します)
- `nonpassive` — 明示的に `passive: false` を設定します
- `capture` — バブリングフェーズではなくキャプチャフェーズ中にハンドラを実行します
- `once` — ハンドラが最初に実行された後、削除します
- `self` — `event.target` がその要素自体だった場合のみハンドラをトリガします
- `trusted` — `event.isTrusted` が `true` の場合にのみハンドラをトリガします。つまり、ユーザーのアクションによってイベントがトリガされた場合です。

修飾子は連鎖させることができます。例 `on:click|once|capture={...}`

`on:` ディレクティブが値なしで使用された場合、コンポーネントはイベントを転送します。つまりコンポーネントの使用者がイベントをリッスンできます。

```
<button on:click>
  コンポーネント自体がクリックイベントを発火します
</button>
```

同じイベントに対して複数のイベントリスナを持つことができます。

```
<script>
  let counter = 0;
  function increment() {
    counter = counter + 1;
  }

  function track(event) {
    trackEvent(event)
  }
</script>

<button on:click={increment} on:click={track}>Click me!</button>
```

bind:property

```
bind:property={variable}
```

データは通常、親から子へと流れていきます。bind: ディレクティブにより、データを子から親へと逆方向に流すことができます。ほとんどのバインディングは個々の要素に特有です。

もっともシンプルなバインディングは、`input.value` のようなプロパティの値を示します。

```
<input bind:value={name}>
<textarea bind:value={text}></textarea>

<input type="checkbox" bind:checked={yes}>
```

名前が値と一致する場合は、省略形を使用できます。

```
<!-- These are equivalent -->
<input bind:value={value}>
<input bind:value>
```

数値の入力値は強制されます。つまり、DOM に関する限り `input.value` は文字列ですが、Svelte はそれを数値として扱います。入力が `empty` や 無効な値の場合 (`type="number"` であれば) 値は `undefined` になります。

```
<input type="number" bind:value={num}>
<input type="range" bind:value={num}>
```

`type="file"` である `<input>` 要素では、選択ファイルの `FileList` を取得するために `bind:files` を使用できます。これは読み取り専用です。

```
<label for="avatar">Upload a picture:</label>
<input
  accept="image/png, image/jpeg"
  bind:files
  id="avatar"
  name="avatar"
  type="file"
/>
```

bind: ディレクティブと on: ディレクティブと一緒に使用する場合、イベントハンドラが呼ばれた際には、定義された順番がバインドされた変数の値に影響します。

```
<script>
  let value = 'Hello World';
</script>

<input
  on:input="{() => console.log('Old value:', value)}"
  bind:value
  on:input="{() => console.log('New value:', value)}"
/>
```

ここではテキストの `input` の値を、`input` イベントを使用してバインドしています。他の要素のバインディングでは、例えば `change` などの異なるイベントが使用されることになります。

Binding <select> value

`<select>` 値のバインディングは、選択された `<option>` の `value` プロパティに対応しており、(通常の DOM のように文字列だけでなく) どんな値でも設定できます。

```
<select bind:value={selected}>
  <option value={a}>a</option>
  <option value={b}>b</option>
  <option value={c}>c</option>
</select>
```

<select multiple> 要素はチェックボックスのグループと同様の動作になります。

```
<select multiple bind:value={fillings}>
  <option value="Rice">Rice</option>
  <option value="Beans">Beans</option>
  <option value="Cheese">Cheese</option>
  <option value="Guac (extra)">Guac (extra)</option>
</select>
```

<option> の値がテキスト内容と一致する場合、value 属性は省略できます。

```
<select multiple bind:value={fillings}>
  <option>Rice</option>
  <option>Beans</option>
  <option>Cheese</option>
  <option>Guac (extra)</option>
</select>
```

contenteditable 属性を持つ要素は innerHTML と textContent のバインディングをサポートします。

```
<div contenteditable="true" bind:innerHTML={html}></div>
```

<details> elements support binding to the open property.

```
<details bind:open={isOpen}>
  <summary>Details</summary>
  <p>
    Something small enough to escape casual notice.
  </p>
</details>
```

Media element bindings

メディア要素 (`<audio>` と `<video>`) には、独自のバインディングのセットがあります -- 6つの `readonly` と...

- duration (readonly) — 動画の総再生時間(秒単位)です。
- buffered (readonly) — {start, end} オブジェクトの配列です。
- played (readonly) — 同上
- seekable (readonly) — 同上
- seeking (readonly) — Boolean
- ended (readonly) — Boolean

...そして5つの 双方向 バインディング。

- currentTime — 動画の現在の再生時間、秒単位です。
- playbackRate — どれぐらい早く、または遅く動画を再生するか、1 が '通常値' です。
- paused — これは自明のはずです。

- `volume` — 0 から 1 の間の値です。
- `muted` — Booleanの値で、`true` はミュートになります。

動画にはさらに、`videoWidth` と `videoHeight` という読み取り専用のバインディングがあります。

```
<video
  src={clip}
  bind:duration
  bind:buffered
  bind:played
  bind:seekable
  bind:seeking
  bind:ended
  bind:currentTime
  bind:playbackRate
  bind:paused
  bind:volume
  bind:muted
  bind:videoWidth
  bind:videoHeight
></video>
```

Block-level element bindings

ブロックレベル要素は、[これ](#)と同様の手法で測定された4つの読み取り専用バインディングを持っています。

- `clientWidth`
- `clientHeight`
- `offsetWidth`
- `offsetHeight`

```
<div
  bind:offsetWidth={width}
  bind:offsetHeight={height}
>
  <Chart {width} {height}/>
</div>
```

bind:group

```
bind:group={variable}
```

グループ化させたい入力には `bind:group` を使用できます。

```
<script>
  let tortilla = 'Plain';
  let fillings = [];
</script>

<!-- こちらのグループ化されたラジオinputは相互に排他的です --&gt;
&lt;input type="radio" bind:group={tortilla} value="Plain"&gt;
&lt;input type="radio" bind:group={tortilla} value="Whole wheat"&gt;
&lt;input type="radio" bind:group={tortilla} value="Spinach"&gt;

<!-- こちらのグループ化されたチェックボックスinputは配列にデータを挿入します --&gt;
&lt;input type="checkbox" bind:group={fillings} value="Rice"&gt;</pre>

```

```
<input type="checkbox" bind:group={fillings} value="Beans">
<input type="checkbox" bind:group={fillings} value="Cheese">
<input type="checkbox" bind:group={fillings} value="Guac (extra)">
```

bind:this

```
bind:this={dom_node}
```

DOM ノードを参照するには `bind:this` を使用します。

```
<script>
  import { onMount } from 'svelte';

  let canvasElement;

  onMount(() => {
    const ctx = canvasElement.getContext('2d');
    drawStuff(ctx);
  });
</script>

<canvas bind:this={canvasElement}></canvas>
```

class:name

```
class:name={value}
```

```
class:name
```

`class:` というディレクティブは要素のクラスを切り替えるための簡単な方法を提供してくれます。

```
<!-- この二つは同等です -->
<div class="{active ? 'active' : ''}">...</div>
<div class:active={active}>...</div>

<!-- 名前と値が一致する場合の省略形がこちらです -->
<div class:active>...</div>

<!-- 複数のクラスの切り替えを含めることができます -->
<div class:active class:inactive={!active} class:isAdmin>...</div>
```

style:property

```
style:property={value}
```

```
style:property="value"
```

```
style:property
```

`style:` ディレクティブは、要素に対して複数のスタイルをセットするためのショートハンドを提供します。

```
<!-- These are equivalent -->
<div style:color="red">...</div>
<div style="color: red;">...</div>

<!-- Variables can be used -->
```

```
<div style:color={myColor}>...</div>
<!-- Shorthand, for when property and variable name match -->
<div style:color>...</div>
<!-- Multiple styles can be included -->
<div style:color style:width="12rem" style:background-color={darkMode ? "black" : "white"}>...</div>
<!-- Styles can be marked as important -->
<div style:color!important="red">...</div>
```

`style:` ディレクティブが `style` 属性と一緒に使用されている場合、`style:` ディレクティブのほうが優先されます。

```
<div style="color: blue;" style:color="red">This will be red</div>
```

use:action

`use:action`

`use:action={parameters}`

```
action = (node: HTMLElement, parameters: any) => {
    update?: (parameters: any) => void,
    destroy?: () => void
}
```

アクションは、要素が作成されたときに呼び出される関数です。要素がアンマウントされたときに呼び出される `destroy` メソッドをもつオブジェクトを返すことができます。

```
<script>
    function foo(node) {
        // the node has been mounted in the DOM

        return {
            destroy() {
                // the node has been removed from the DOM
            }
        };
    }
</script>

<div use:foo></div>
```

アクションはパラメータを取ることができます。戻り値に `update` メソッドがあると、そのパラメータが変化するときはいつも、Svelte がマークアップに更新を適用した直後にそのメソッドが呼び出されます。

すべてのコンポーネントインスタンスに対して `foo` 関数を再宣言しているという事実について心配する必要はありません。Svelte は、ローカル状態に依存しない関数をコンポーネント定義から巻き上げます。

```
<script>
    export let bar;

    function foo(node, bar) {
```

```
// the node has been mounted in the DOM

return {
  update(bar) {
    // the value of `bar` has changed
  },
  destroy() {
    // the node has been removed from the DOM
  }
};

</script>

<div use:foo={bar}></div>
```

transition:fn

`transition:fn`

`transition:fn={params}`

`transition:fn||local`

`transition:fn||local={params}`

```
transition = (node: HTMLElement, params: any) => {
  delay?: number,
  duration?: number,
  easing?: (t: number) => number,
  css?: (t: number, u: number) => string,
  tick?: (t: number, u: number) => void
}
```

トランジションは、状態変化の結果として DOM に出入りする要素によってトリガーされます。

ブロックが外へのトランジションをしているとき、ブロック内のすべての要素（独自のトランジションを持たない要素を含む）は、ブロック内のすべてのトランジションが完了するまで DOM に保持されます。

`transition:` ディレクティブは 双方向 トランジションを示しており、トランジションが進行している間、円滑に反転させることができます。

```
{#if visible}
  <div transition:fade>
    fades in and out
  </div>
{/if}
```

デフォルトでは、イントロトランジションは最初のレンダリングでは再生されません。この動作は、[コンポーネントを作成する](#)ときに `intro: true` を設定することで変更できます。

Transition parameters

アクションと同様に、トランジションはパラメータを持つことができます。

(ダブル {{中括弧}} は特殊な構文ではありません。これは式タグ内のオブジェクトリテラルです)

```
{#if visible}
  <div transition:fade="{{ duration: 2000 }}">
    fades in and out over two seconds
  </div>
{/if}
```

Custom transition functions

トランジションはカスタム関数を使うことができます。返されたオブジェクトに `css` 関数があれば、Svelte は要素上で再生される CSS アニメーションを作成します。

`css` に渡される `t` 引数は `easing` 関数を適用した後の `0` から `1` の間の値です。 `In` トランジションは `0` から `1` まで、`out` トランジションは `1` から `0` までの間で実行されます。-- 言い換えれば、`1` はトランジションが適用されていないかのような要素の自然な状態です。引数 `u` は `1 - t` と等しくなります。

この関数はトランジションが始まる 前に繰り返し呼び出され、異なる `t` と `u` の引数を持ちます。

```
<script>
  import { elasticOut } from 'svelte/easing';

  export let visible;

  function whoosh(node, params) {
    const existingTransform = getComputedStyle(node).transform.replace('none', '');
    return {
      delay: params.delay || 0,
      duration: params.duration || 400,
      easing: params.easing || elasticOut,
      css: (t, u) => `transform: ${existingTransform} scale(${t})`;
    };
  }
</script>

{#if visible}
  <div in:whoosh>
    whooshes in
  </div>
{/if}
```

カスタムのトランジション関数は `tick` 関数を返すこともでき、これは同じ `t` と `u` の引数を持つトランジションの 最中に呼び出されます。

`tick` の代わりに `css` を使うことが可能ならば、そうしてください。-- CSS アニメーションはメインスレッドの外で実行することができるため、遅いデバイスでのジャンクを防ぐことができます。

```
<script>
  export let visible = false;

  function typewriter(node, { speed = 1 }) {
```

```

const valid = (
  node.childNodes.length === 1 &&
  node.childNodes[0].nodeType === Node.TEXT_NODE
);

if (!valid) {
  throw new Error(`This transition only works on elements with a single
text node child`);
}

const text = node.textContent;
const duration = text.length / (speed * 0.01);

return {
  duration,
  tick: t => {
    const i = ~~(text.length * t);
    node.textContent = text.slice(0, i);
  }
};
</script>

#{if visible}
<p in:typewriter="{{ speed: 1 }}">
  The quick brown fox jumps over the lazy dog
</p>
{/if}

```

トランジションがトランジションオブジェクトではなく関数を返す場合、その関数は次のマイクロタスクで呼び出されます。これにより、複数のトランジションを調整することができ、[クロスフェード効果](#)が可能になります。

Transition events

トランジションを持つ要素は、標準的なDOMイベントに加えて以下のイベントをディスパッチします。

- introstart
- introend
- outrostart
- outroend

```

#{if visible}
<p
  transition:fly="{{ y: 200, duration: 2000 }}"
  on:introstart="{}() => status = 'intro started'"
  on:outrostart="{}() => status = 'outro started'"
  on:introend="{}() => status = 'intro ended'"
  on:outroend="{}() => status = 'outro ended'"
>
  Flies in and out
</p>
{/if}

```

ローカルトランジションは、親ブロックが作成または破棄されたときには再生されず、所属するブロックが作成または破棄されたときにのみ再生されます。

```
{#if x}
  {#if y}
    <p transition:fade>
      fades in and out when x or y change
    </p>

    <p transition:fade|local>
      fades in and out only when y changes
    </p>
  {/if}
{/if}
```

in:fn/out:fn

in:fn

in:fn={params}

in:fn|local

in:fn|local={params}

out:fn

out:fn={params}

out:fn|local

out:fn|local={params}

`transition:` に似ていますが、`in:` は DOM に入る要素だけに、`out:` は出る要素だけに適用されます。

`transition:` とは違って、`in:` と `out:` を適用したトランジションは双方向ではありません。つまり、もしトランジションの最中にブロックがアウトロされた場合、逆転するのではなく、イントロトランジションはアウトロトランジションと一緒に「再生」し続けます。アウトロトランジションが中止された場合、トランジションは最初から再開されます。

```
{#if visible}
  <div in:fly out:fade>
    flies in, fades out
  </div>
{/if}
```

animate:fn

animate:name

animate:name={params}

```
animation = (node: HTMLElement, { from: DOMRect, to: DOMRect } , params: any) => {
  delay?: number,
  duration?: number,
  easing?: (t: number) => number,
  css?: (t: number, u: number) => string,
  tick?: (t: number, u: number) => void
}
```

```
DOMRect {
  bottom: number,
  height: number,
  left: number,
  right: number,
  top: number,
  width: number,
  x: number,
  y: number
}
```

アニメーションは、[keyed each block](#) の内容が並び替えられたときに発生します。アニメーションは、要素が削除されたときには実行されず、each ブロックのデータが並べ替えられたときにのみ実行されます。animate ディレクティブは、キー付き each ブロックの直接の子要素になければいけません。

アニメーションは Svelte の組み込みアニメーション関数 または [カスタムアニメーション関数](#) を使用することができます。

```
<!-- When `list` is reordered the animation will run-->
{#each list as item, index (item)}
  <li animate:flip>{item}</li>
{/each}
```

Animation Parameters

アクションやトランジションと同様に、アニメーションはパラメータを持つことができます。

(ダブル {{中括弧}} は特殊な構文ではありません。これは式タグ内のオブジェクトリテラルです)

```
{#each list as item, index (item)}
  <li animate:flip="{{ delay: 500 }}>{item}</li>
{/each}
```

Custom animation functions

アニメーションは、`node`、`animation` オブジェクト、および任意の `parameters` を引数として指定するカスタム関数を使用することができます。`animation` パラメータは、`from` と `to` プロパティを含むオブジェクトで、それぞれ要素の `start` と `end` の位置におけるジオメトリを記述した `DOMRect` を含みます。`from` プロパティは要素の開始位置の `DOMRect` であり、`to` プロパティはリストが並び替えられ DOM が更新された後の最終位置の `DOMRect` です。

返されたオブジェクトが `css` メソッドを持つ場合、Svelte は要素上で再生される CSS アニメーションを作成します。

`css` に渡される `t` 引数は `easing` 関数が適用された後の `0` と `1` の値です。引数 `u` は `1 - t` に等しい値です。

この関数はアニメーションが始まる 前に 繰り返し呼ばれ、異なる `t` と `u` の引数を持ちます。

```
<script>
  import { cubicOut } from 'svelte/easing';

  function whizz(node, { from, to }, params) {
    const dx = from.left - to.left;
    const dy = from.top - to.top;

    const d = Math.sqrt(dx * dx + dy * dy);
```

```

        return {
          delay: 0,
          duration: Math.sqrt(d) * 120,
          easing: cubicOut,
          css: (t, u) =>
            `transform: translate(${u * dx}px, ${u * dy}px)
rotate(${t*360}deg);`
        };
      }
    
```

```

{#each list as item, index (item)}
  <div animate:whizz>{item}</div>
{/each}

```

カスタムアニメーション関数は `tick` 関数を返すこともできます。`tick` 関数はアニメーションの 最中に呼び出され、同じ `t` と `u` を引数に取ります。

`tick` の代わりに `css` を使うことが可能ならば、そうしてください。-- CSS アニメーションはメインスレッドの外で実行することができるため、遅いデバイスでのジャンクを防ぐことができます。

```

<script>
  import { cubicOut } from 'svelte/easing';

  function whizz(node, { from, to }, params) {
    const dx = from.left - to.left;
    const dy = from.top - to.top;

    const d = Math.sqrt(dx * dx + dy * dy);

    return {
      delay: 0,
      duration: Math.sqrt(d) * 120,
      easing: cubicOut,
      tick: (t, u) =>
        Object.assign(node.style, {
          color: t > 0.5 ? 'Pink' : 'Blue'
        });
    };
  }
</script>

{#each list as item, index (item)}
  <div animate:whizz>{item}</div>
{/each}

```

Component directives

on:eventname

```
on:eventname={handler}
```

コンポーネントは `createEventDispatcher` を用いるか、または DOM イベントをフォワードすることでイベントを発火することができます。コンポーネントのイベントをリッスンするための書き方は、DOM イベントをリッスンする書き方と同じです：

```
<SomeComponent on:whatever={handler}/>
```

DOM イベントと同様に、`on:` ディレクティブが値なしに使われる場合、コンポーネントはイベントをフォワードしますが、これはコンポーネントのユーザーがイベントをリッスンできることを意味します。

```
<SomeComponent on:whatever/>
```

--style-props

```
--style-props="anycssvalue"
```

[Svelte 3.38 \(RFC\)](#) から、テーマ設定のためにスタイルをプロパティとしてコンポーネントに渡すことができます。これには CSS カスタムプロパティを使用します。

Svelte の実装は、基本的にラッパー要素を追加するためのシンタックスシュガー(糖衣構文)です。この例では：

```
<Slider
  bind:value
  min={0}
  --rail-color="black"
  --track-color="rgb(0, 0, 255)"
/>
```

デシュガー(脱糖)すると：

```
<div style="display: contents; --rail-color: black; --track-color: rgb(0, 0, 255)">
  <Slider
    bind:value
    min={0}
    max={100}
  />
</div>
```

注意: 余分なdivが追加されるため、あなたのCSS構造が誤ってこれをターゲットにしてしまう可能性があるので注意してください。この機能を使用する際は、この追加されるラッパー要素に気をつけてください。また、全てのブラウザが `display: contents` をサポートしているわけではないことに注意してください: <https://caniuse.com/css-display-contents>

SVG namespace の場合、上記の例はデシュガー(脱糖)すると `<g>` が代わりに使用されます：

```
<g style="--rail-color: black; --track-color: rgb(0, 0, 255)">
  <Slider
    bind:value
    min={0}
    max={100}
  />
</g>
```

Note: 余分な `<g>` が追加されるため、あなたのCSS構造が誤ってこれをターゲットにしてしまう可能性があるので注意してください。この機能を利用する際は、この追加されるラッパー要素に気をつけてください。

Svelte の CSS Variables サポートによって、テーマに沿ったコンポーネントを作るのは容易です：

```
<!-- Slider.svelte -->
<style>
  .potato-slider-rail {
    background-color: var(--rail-color, var(--theme-color, 'purple'));
  }
</style>
```

ハイレベルなテーマカラーを設定できますし、

```
/* global.css */
html {
  --theme-color: black;
}
```

コンシューマーレベルでそれをオーバーライドできます。

```
<Slider --rail-color="goldenrod"/>
```

bind:property

```
bind:property={variable}
```

要素と同じ構文を用いてコンポーネントのプロパティにバインドすることができます。

```
<Keypad bind:value={pin}/>
```

bind:this

```
bind:this={component_instance}
```

コンポーネントは `bind:this` もサポートしており、これを用いることでコンポーネントのインスタンスをプログラムで操作できるようになります。

ボタンが最初にレンダリングされるときには `cart` が `undefined` であり、エラーを投げる所以 `{cart.empty}` を実行できないことに注意してください。

```
<ShoppingCart bind:this={cart}>
<button on:click={() => cart.empty()}>
  Empty shopping cart
</button>
```

<slot>

```
<slot><!-- optional fallback --></slot>
```

```
<slot name="x"><!-- optional fallback --></slot>
```

```
<slot prop={value}></slot>
```

コンポーネントは要素と同じ様に、子コンテンツを持つことができます。

コンテンツは `<slot>` 要素を用いて子コンポーネントに公開され、子が何も提供されない場合、レンダリングされるフォールバックのコンテンツを含みます。

```
<!-- Widget.svelte -->
<div>
  <slot>
    this fallback content will be rendered when no content is provided, like
    in the first example
  </slot>
</div>

<!-- App.svelte -->
<Widget></Widget> <!-- this component will render the default content -->

<Widget>
  <p>this is some child content that will overwrite the default slot content</p>
</Widget>
```

`<slot name="name">`

名前付きスロットは、特定の場所をターゲットにすることを可能にします。また、フォールバックのコンテンツを含むこともできます。

```
<!-- Widget.svelte -->
<div>
  <slot name="header">No header was provided</slot>
  <p>Some content between header and footer</p>
  <slot name="footer"></slot>
</div>

<!-- App.svelte -->
<Widget>
  <h1 slot="header">Hello</h1>
  <p slot="footer">Copyright (c) 2019 Svelte Industries</p>
</Widget>
```

コンポーネントは、`<Component slot="name" />` という構文を使って、名前付きスロットに入れることができます。ラッパー要素を使うことなくスロットに内容を入れるために、特殊な要素 `<svelte:fragment>` を使うことができます。

```
<!-- Widget.svelte -->
<div>
  <slot name="header">No header was provided</slot>
  <p>Some content between header and footer</p>
  <slot name="footer"></slot>
</div>

<!-- App.svelte -->
<Widget>
  <HeaderComponent slot="header" />
  <svelte:fragment slot="footer">
    <p>All rights reserved.</p>
```

```
<p>Copyright (c) 2019 Svelte Industries</p>
</svelte:fragment>
</Widget>
```

\$\$slots

`$$slots` は、親からコンポーネントに渡されたスロットの名前がキーとなるオブジェクトです。親が特定の名前のスロットを渡さなかった場合、その名前は `$$slots` には存在しません。これにより、親がスロットを指定した場合にのみコンポーネントがスロット (と他の要素、例えばスタイリング用のラッパーなど) をレンダリングすることができます。

注意してください、明示的に空の名前付きスロットを渡すと、そのスロットの名前が `$$slots` に追加されます。例えば、親が `<div slot="title" />` を子コンポーネントに渡した場合、`$$slots.title` は子コンポーネント内で有効になります。

```
<!-- Card.svelte -->
<div>
  <slot name="title"></slot>
  {#if $$slots.description}
    <!-- この<hr>とスロットは、"description"という名前のスロットが提供されている場合にのみ
    レンダリングされます。 -->
    <hr>
    <slot name="description"></slot>
  {/if}
</div>

<!-- App.svelte -->
<Card>
  <h1 slot="title">Blog Post Title</h1>
  <!-- "description"という名前のスロットは提供されていないので、該当しないスロットはレンダリン
  グされません。 -->
</Card>
```

<slot key={ value }>

スロットは0回以上レンダリングすることができ、プロパティ(props)を使って親に値を 戻す ことができます。親は `let:` ディレクティブを使ってスロットテンプレートに値を公開します。

通常の短縮ルールが適用されます -- `let:item` は `let:item={item}` と同等であり、`<slot {item}>` は `<slot item={item}>` と同等です。

```
<!-- FancyList.svelte -->
<ul>
  {#each items as item}
    <li class="fancy">
      <slot prop={item}></slot>
    </li>
  {/each}
</ul>

<!-- App.svelte -->
<FancyList {items} let:prop={thing}>
  <div>{thing.text}</div>
</FancyList>
```

名前付きスロットは値を公開することもできます。`let:` ディレクティブは `slot` 属性を持つ要素に適用されます。

```
<!-- FancyList.svelte -->
<ul>
  {#each items as item}
    <li class="fancy">
      <slot name="item" {item}></slot>
    </li>
  {/each}
</ul>

<slot name="footer"></slot>

<!-- App.svelte -->
<FancyList {items}>
  <div slot="item" let:item>{item.text}</div>
  <p slot="footer">Copyright (c) 2019 Svelte Industries</p>
</FancyList>
```

<svelte:self>

<svelte:self> 要素を使用すると、コンポーネントにそれ自体を再帰的に含めることができます。

マークアップのトップレベルに表示することはできません。また、無限ループを防ぐために、`if` や `each` ブロックの内側に入れるか、コンポーネントのスロットに渡す必要があります。

```
<script>
  export let count;
</script>

{#if count > 0}
  <p>counting down... {count}</p>
  <svelte:self count="{count - 1}" />
{:else}
  <p>lift-off!</p>
{/if}
```

<svelte:component>

```
<svelte:component this={expression}>/>
```

<svelte:component> 要素は、`this` プロパティで指定されたコンポーネントのコンストラクタを用いて、コンポーネントを動的にレンダリングします。プロパティが変更されると、コンポーネントは破棄されて再生成されます。

`this` の値が `falsy` である場合、コンポーネントはレンダリングされません。

```
<svelte:component this={currentSelection.component} foo={bar}>/>
```

<svelte:element>

```
<svelte:element this={expression}>/>
```

<svelte:element> 要素は、動的に指定されたタイプの要素をレンダリングさせることができます。これは例えば、CMS のリッチなテキストコンテンツを表示する場合などに便利です。プロパティやリスナーが存在する場合は、その要素に適用されます。

Svelte がビルド時に処理する要素タイプ固有のバインディング（例：input 要素の bind:value）は動的なタグタイプでは動作しないため、サポートされているバインディングは bind:this のみです。

this が nullish な値である場合、この要素と子要素はレンダリングされません。

this が void 要素のタグ名（例えば br）で、<svelte:element> が子要素を持っている場合、開発モードの場合はランタイムエラーがスローされます。

```
<script>
  let tag = 'div';
  export let handler;
</script>

<svelte:element this={tag} on:click={handler}>Foo</svelte:element>
```

<svelte:window>

```
<svelte:window on:event={handler}/>
```

```
<svelte:window bind:prop={value}/>
```

<svelte:window> 要素を使うと、コンポーネントが破棄されたときにイベントリスナーを削除したり、サーバサイドでレンダリングするときに window が存在するかどうかをチェックしたりすることなく、window オブジェクトにイベントリスナーを追加することができます。

<svelte:self> とは逆に、この要素はコンポーネントのトップレベルにのみ置くことができ、ブロックや要素の中に置くことはできません。

```
<script>
  function handleKeydown(event) {
    alert(`pressed the ${event.key} key`);
  }
</script>

<svelte:window on:keydown={handleKeydown}/>
```

また、以下のプロパティにバインドすることもできます。

- innerWidth
- innerHeight
- outerWidth
- outerHeight
- scrollX
- scrollY
- online — window.navigator.onLine の別名です

scrollX と scrollY 以外はすべて読み込み専用です。

```
<svelte:window bind:scrollY={y}/>
```

アクセシビリティの問題を避けるため、ページは初期値にスクロールされないことにご注意ください。scrollX と scrollY にバインドされている変数が変更された後にのみ、スクロールが発生します。ただし、スクロールの挙動が必要であれば、onMount() 内で scrollTo() を呼び出してください。

<svelte:body>

```
<svelte:body on:event={handler}>
```

<svelte:window> と同様に、この要素を使うことで document.body のイベント、例えば window では発生しない mouseenter や mouseleave などのリスナーを追加することができます。また、<body> 要素に action を使用することもできます。

<svelte:window> と同様に、この要素はコンポーネントのトップレベルにのみ置くことができ、ブロックや要素の中に置くことはできません。

```
<svelte:body
  on:mouseenter={handleMouseenter}
  on:mouseleave={handleMouseleave}
  use:someAction
/>
```

<svelte:head>

```
<svelte:head>...</svelte:head>
```

この要素を使うと、document.head に要素を挿入することができます。サーバサイドのレンダリングでは、head の内容はメインの html の内容とは別に公開されます。

<svelte:window> と <svelte:body> と同様に、この要素はコンポーネントのトップレベルにのみ置くことができ、ブロックや要素の中に置くことはできません。

```
<svelte:head>
  <link rel="stylesheet" href="/tutorial/dark-theme.css">
</svelte:head>
```

<svelte:options>

```
<svelte:options option={value}>
```

<svelte:options> 要素は、コンポーネントごとのコンパイラオプションを指定する場所を提供します。これらは [コンパイラセクション](#) で詳しく説明されています。使用できるオプションは以下の通りです。

- immutable={true} — 変異可能なデータは絶対に使いません。そのため、コンパイラは値が変更されたかどうかを判断するために単純な参照等価性チェックを行うことができます。
- immutable={false} — デフォルトです。Svelte は、変更可能なオブジェクトが変更されたかどうかについて、より保守的になります。
- accessors={true} — コンポーネントのプロパティ(props)のゲッターとセッターを追加します。

- `accessors={false}` — 初期値です
- `namespace="..."` — このコンポーネントが使用される名前空間、最も一般的には "svg" です。大文字小文字を区別しない属性名とHTML固有の警告を除外するために "外部の" 名前空間を使用します。
- `tag="..."` — このコンポーネントをカスタム要素としてコンパイルする際に使用する名前。

```
<svelte:options tag="my-custom-element"/>
```

<svelte:fragment>

<svelte:fragment> 要素によって、コンテナとなるDOM要素でラップすることなく名前付きスロットに内容を入れることができます。

```
<!-- Widget.svelte -->
<div>
  <slot name="header">No header was provided</slot>
  <p>Some content between header and footer</p>
  <slot name="footer"></slot>
</div>

<!-- App.svelte -->
<Widget>
  <h1 slot="header">Hello</h1>
  <svelte:fragment slot="footer">
    <p>All rights reserved.</p>
    <p>Copyright (c) 2019 Svelte Industries</p>
  </svelte:fragment>
</Widget>
```

[Go to TOC](#)

Run time

svelte

`svelte` パッケージは、[ライフサイクル関数](#) と `context API` を公開します。

onMount

```
onMount(callback: () => void)
```

```
onMount(callback: () => () => void)
```

`onMount` 関数は、コンポーネントが DOM にマウントされるとすぐに実行されるコールバックをスケジュールします。これはコンポーネントの初期化中に呼び出されなければなりません（ただし、コンポーネントの `内部` に存在する必要はありません。外部モジュールから呼び出すことができます）。

`onMount` は [サーバーサイドコンポーネント](#) の内部では実行されません。

```
<script>
  import { onMount } from 'svelte';

  onMount(() => {
    console.log('the component has mounted');
  });
</script>
```

`onMount` から関数が返された場合、コンポーネントがアンマウントされたときに呼び出されます。

```
<script>
  import { onMount } from 'svelte';

  onMount(() => {
    const interval = setInterval(() => {
      console.log('beep');
    }, 1000);

    return () => clearInterval(interval);
  });
</script>
```

この挙動は、`onMount` に渡された関数が [同期的に](#) 値を返す場合にのみ動作します。`async` の関数は常に `Promise` を返すため、[同期的に](#) 値を返すことはできません。

beforeUpdate

```
beforeUpdate(callback: () => void)
```

`state` が変化した後、コンポーネントが更新される直前に実行されるコールバックをスケジュールします。

コールバックが最初に実行されるのは、初回の `onMount` の前になります。

```
<script>
  import { beforeUpdate } from 'svelte';

  beforeUpdate(() => {
    console.log('the component is about to update');
  });
</script>
```

afterUpdate

```
afterUpdate(callback: () => void)
```

コンポーネントが更新された直後に実行するコールバックをスケジュールします。

コールバック実行が最初に実行されるのは、最初の `onMount` の後です

```
<script>
  import { afterUpdate } from 'svelte';

  afterUpdate(() => {
    console.log('the component just updated');
  });
</script>
```

onDestroy

```
onDestroy(callback: () => void)
```

コンポーネントがアンマウントされる直前に実行するコールバックをスケジュールします。

`onMount` , `beforeUpdate` , `afterUpdate` , `onDestroy` のうち、サーバサイドコンポーネントの中で動作するのはこれだけです。

```
<script>
  import { onDestroy } from 'svelte';

  onDestroy(() => {
    console.log('the component is being destroyed');
  });
</script>
```

tick

```
promise: Promise = tick()
```

保留中の state の変更が適用されると `resolve` する promise を返します。もしくは何も保留していない場合に、その次のマイクロタスクで実行されます。

```
<script>
  import { beforeUpdate, tick } from 'svelte';

  beforeUpdate(async () => {
    console.log('the component is about to update');
    await tick();
    console.log('the component just updated');
  });
</script>
```

setContext

```
setContext(key: any, context: any)
```

任意の `context` オブジェクトを現在のコンポーネントと指定された `key` に関連付け、オブジェクトを返します。 `context` は、コンポーネントの子（スロットコンテンツを含む）が `getContext` で利用できるようになります。

ライフサイクル関数と同様に、これはコンポーネントの初期化時に呼ばれなければなりません。

```
<script>
  import { setContext } from 'svelte';

  setContext('answer', 42);
</script>
```

`context` は本質的にリアクティブではありません。リアクティブな値が必要な場合は、`context` にストアを渡すことができます。これはリアクティブになります。

getContext

```
context: any = getContext(key: any)
```

指定された `key` を持つ、最も近い親コンポーネントに属する `context` を取得します。コンポーネントの初期化中に呼び出されなければなりません。

```
<script>
  import { getContext } from 'svelte';

  const answer = getContext('answer');
</script>
```

hasContext

```
hasContext: boolean = hasContext(key: any)
```

与えられた `key` が親コンポーネントの `context` に設定されているかどうかをチェックします。コンポーネントの初期化時に呼び出されなければなりません。

```
<script>
  import { hasContext } from 'svelte';

  if (hasContext('answer')) {
```

```
// do something
}
</script>
```

getAllContexts

```
contexts: Map<any, any> = getAllContexts()
```

最も近い親コンポーネントにある全ての context マップを取得します。これはコンポーネントの初期化中に呼び出す必要があります。例えば、プログラムでコンポーネントを作成し、既存の context を渡したい場合などに便利です。

```
<script>
  import { getAllContexts } from 'svelte';

  const contexts = getAllContexts();
</script>
```

createEventDispatcher

```
dispatch: ((name: string, detail?: any, options?: DispatchOptions) => boolean) = createEventDispatcher();
```

コンポーネントイベント のディスパッчに使用できるイベントディスパッチャーを作成します。イベントディスパッチャーは、`name` と `detail` の2つの引数を取る関数です。

`createEventDispatcher` で作成されたコンポーネントイベントは カスタムイベント を作成します。これらのイベントは バーリング しません。引数 `detail` は `CustomEvent.detail` プロパティに対応し、任意のタイプのデータを含むことができます。

```
<script>
  import { createEventDispatcher } from 'svelte';

  const dispatch = createEventDispatcher();
</script>

<button on:click="{() => dispatch('notify', 'detail value')}">Fire Event</button>
```

子コンポーネントからディスパッチされたイベントは、親でリッスンできます。イベントがディスパッチされたときに提供されたデータはすべて、イベントオブジェクトの `detail` プロパティで利用できます。

```
<script>
  function callbackFunction(event) {
    console.log(`Notify fired! Detail: ${event.detail}`)
  }
</script>

<Child on:notify="{callbackFunction}">
```

`dispatch` 関数に3番目のパラメータを渡すことで、イベントがキャンセルできるようになります。もしイベントが `event.preventDefault()` によってキャンセルされると、その関数は `false` を返します。それ以外の場合は `true` を返します。

```
<script>
    import { createEventDispatcher } from 'svelte';
    const dispatch = createEventDispatcher();
    function notify() {
        const shouldContinue = dispatch('notify', 'detail value', { cancelable: true });
        if (shouldContinue) {
            // no one called preventDefault
        } else {
            // a listener called preventDefault
        }
    }
</script>
```

svelte/store

`svelte/store` モジュールは、`readable`、`writable`、`derived` ストアを作成するための関数をエクスポートします。

コンポーネントで [リアクティブな \\$store 構文](#) を便利に使うために、これらの関数を使用する必要がないことを覚えておいてください。`.subscribe` とそのサブスクライブの解除、(オプションで) `.set` を正しく実装したオブジェクトは有効なストアであり、その特殊な構文と Svelte に組み込まれた `derived` ストアの両方で機能します。

これにより、Svelte で使用するための、ほぼすべての他のリアクティブ状態を扱うライブラリをラップすることが可能になります。続いて説明する正しい実装がどのようなものか理解するために、[store contract](#) も読んでみてください。

writable

```
store = writable(value?: any)
store = writable(value?: any, start?: (set: (value: any) => void) => () => void)
```

「外部」コンポーネントから設定できる値を持つストアを作成する関数。これは、`set` と `update` 関数を併せ持つオブジェクトを作成します。

`set` は、設定する値である1つの引数を受け取る関数です。ストア値が引数の値とまだ等しくない場合、ストア値は引数の値に設定されます。

`update` は、コールバックである1つの引数を受け取る関数です。コールバックは、既存のストア値を引数として受け取り、ストアに設定される新しい値を返します。

```
import { writable } from 'svelte/store';
const count = writable(0);
count.subscribe(value => {
    console.log(value);
}); // logs '0'
count.set(1); // logs '1'
count.update(n => n + 1); // logs '2'
```

関数が第2引数として渡された場合、サブスクリーバーの数が0から1になると呼び出されます（ただし、1から2になった場合には呼び出されません）。その関数には、ストアの値を変更する `set` 関数が渡されます。その関数は、サブスクリーバーの数が1から0になったときに呼び出される `stop` 関数を返す必要があります。

```
import { writable } from 'svelte/store';

const count = writable(0, () => {
    console.log('got a subscriber');
    return () => console.log('no more subscribers');
});

count.set(1); // does nothing

const unsubscribe = count.subscribe(value => {
    console.log(value);
}); // logs 'got a subscriber', then '1'

unsubscribe(); // logs 'no more subscribers'
```

`writable` の値は、ページが更新されたときなど破棄されると失われるので注意してください。ただし、`localStorage` などに値を同期する独自ロジックを作ることはできます。

readable

```
store = readable(value?: any, start?: (set: (value: any) => void) => () => void)
```

「外側」から値を設定できないストアを作成します。第1引数はストアの初期値です。`readable` の第2引数は `writable` の第2引数と同じです。

```
import { readable } from 'svelte/store';

const time = readable(null, set => {
    set(new Date());
    const interval = setInterval(() => {
        set(new Date());
    }, 1000);
    return () => clearInterval(interval);
});
```

derived

```
store = derived(a, callback: (a: any) => any)
```

```
store = derived(a, callback: (a: any, set: (value: any) => void) => void | () => void, initial_value: any)
```

```
store = derived([a, ...b], callback: ([a: any, ...b: any[]]) => any)
```

```
store = derived([a, ...b], callback: ([a: any, ...b: any[]]), set: (value: any) => void) => void | () => void, initial_value: any)
```

1つ以上の他のストアからストアを派生させます。これらの依存しているものが変化するたびに、コールバックが実行されます。

最もシンプルな例だと、`derived` は単一のストアを受け取り、コールバックは派生値を返します。

```
import { derived } from 'svelte/store';

const doubled = derived(a, $a => $a * 2);
```

コールバックは、第2引数に `set` を受け取り、しかるべき時にそれを呼び出すことで非同期に値を設定できます。

この場合、`derived` に第3引数として、`set` が初めて呼び出される前の派生ストアの初期値を渡すこともできます。

```
import { derived } from 'svelte/store';

const delayed = derived(a, ($a, set) => {
    setTimeout(() => set($a), 1000);
}, 'one moment...');
```

コールバックから関数を返すと、a) コールバックが再度実行される時や b) 最後のサブスクリーパーがサブスクリーブを解除する時に呼び出されます。

```
import { derived } from 'svelte/store';

const tick = derived(frequency, ($frequency, set) => {
    const interval = setInterval(() => {
        set(Date.now());
    }, 1000 / $frequency);

    return () => {
        clearInterval(interval);
    };
}, 'one moment...');
```

どちらの場合も、第1引数として、ストア1つではなく、引数の配列を渡すことができます。

```
import { derived } from 'svelte/store';

const summed = derived([a, b],([$a, $b]) => $a + $b);

const delayed = derived([a, b],([$a, $b], set) => {
    setTimeout(() => set($a + $b), 1000);
});
```

get

```
value: any = get(store)
```

通常は、ストアをサブスクリーブして、ストア値が切り替わるたびに使用する方法で読み取るほうがよいでしょう。しかし、場合によっては、サブスクリーブしていないストア値を取得する必要があります。`get` はそれを可能にします。

これは、サブスクリーピションを作成し、値を読み取ってから、サブスクリーピションを解除することで機能します。したがって、ホットコードパスではお勧めしません。

```
import { get } from 'svelte/store';
const value = get(store);
```

svelte/motion

`svelte/motion` モジュールは、`tweened` と `spring` という2つの関数をエクスポートします。これは書き込み可能なストアを作成するためのもので、値がすぐにではなく、`set` と `update` の後に時間の経過とともに変化するものです。

tweened

```
store = tweened(value: any, options)
```

トゥイーンされたストアは、一定の期間にわたって値を更新します。以下のオプションが利用可能です。

- `delay` (`number`, `default 0`) — 開始前の待ち時間のミリ秒
- `duration` (`number` | `function`, `default 400`) — トゥイーンの持続時間のミリ秒
- `easing` (`function`, `default t => t`) — イージング関数
- `interpolate` (`function`) — 下記を参照してください。

`store.set` と `store.update` は、インスタンス化時に渡されたオプションを上書きする第2引数 `options` を受け取ることができます。

どちらの関数も、トゥイーンが完了すると `resolve` する `promise` を返します。トゥイーンが中断されると、`promise` は `reject` されません。

Svelte は2つの数値、2つの配列、または2つのオブジェクトの間を補間します(配列とオブジェクトが同じ '形状' であり、それらの '子孫' プロパティも数値である限り)。

```
<script>
  import { tweened } from 'svelte/motion';
  import { cubicOut } from 'svelte/easing';

  const size = tweened(1, {
    duration: 300,
    easing: cubicOut
  });

  function handleClick() {
    // これは size.update(n => n + 1) と等価です
    $size += 1;
  }
</script>

<button
  on:click={handleClick}
  style="transform: scale({$size}); transform-origin: 0 0"
>embiggen</button>
```

初期値が `undefined` または `null` の場合、最初の値の変更はすぐに有効になります。これは、プロパティをベースにしたトゥイーン値を設定していて、コンポーネントの最初のレンダリング時にモーションをかけたくない場合に便利です。

```
const size = tweened(undefined, {
  duration: 300,
  easing: cubicOut
});
$: $size = big ? 100 : 10;
```

`interpolate` オプションを指定すると、任意の値の間でトゥイーンを行うことができます。関数 `(a, b) => t => value` で、`a` は開始値、`b` は目標値、`t` は 0 から 1 の間の数値、`value` は結果です。例えば、[d3-interpolate](#) パッケージを使えば、2つの色の間をスムーズに補間することができます。

```
<script>
  import { interpolateLab } from 'd3-interpolate';
  import { tweened } from 'svelte/motion';

  const colors = [
    'rgb(255, 62, 0)',
    'rgb(64, 179, 255)',
    'rgb(103, 103, 120)'
  ];

  const color = tweened(colors[0], {
    duration: 800,
    interpolate: interpolateLab
  });
</script>

#{each colors as c}
  <button
    style="background-color: {c}; color: white; border: none;"
    on:click="{e => color.set(c)}"
  >{c}</button>
{/each}

<h1 style="color: {$color}">{$color}</h1>
```

spring

```
store = spring(value: any, options)
```

`spring` ストアは、`stiffness` と `damping` パラメータに基づいて目標値まで徐々に変化します。`tweened` ストアが一定のタイミングで値を変化させるのに対し、`spring` ストアは既存の速度によって決定されるタイミングで変化するため、多くの状況でより自然に見える動きを可能にします。以下のオプションが利用可能です。

- `stiffness` (`number`, default `0.15`) — 0 から 1 の間の値で、高い方が「よりタイトな」スプリングを意味します。
- `damping` (`number`, default `0.8`) — 0 から 1 の間の値で、少ない方が「より弾力のある」スプリングを意味します。
- `precision` (`number`, default `0.001`) — は、スプリングが「止まった」とみなされる閾値を決定します。少ない方がより精密であることを意味します。

上記のオプションは全て `spring` のモーション中に変更することができ、すぐにその効果に反映されます。

```
const size = spring(100);
size.stiffness = 0.3;
size.damping = 0.4;
size.precision = 0.005;
```

`tweened` ストアと同様に、`set` と `update` はスプリングが止まれば `resolve` する `promise` を返します。

`set` と `update` はどちらも第2引数として `hard` または `soft` プロパティを持つオブジェクトを取ることができます。`{ hard: true }` は対象の値を即座に設定します。`{ soft: n }` は既存の運動量を `n` 秒間保持してから止まります。`{ soft: true }` は `{ soft: 0.5 }` と同等です。

```
const coords = spring({ x: 50, y: 50 });
// updates the value immediately
coords.set({ x: 100, y: 200 }, { hard: true });
// preserves existing momentum for 1s
coords.update(
  (target_coords, coords) => {
    return { x: target_coords.x, y: coords.y };
  },
  { soft: 1 }
);
```

スプリングチュートリアルの例を参照してください。

```
<script>
  import { spring } from 'svelte/motion';

  const coords = spring({ x: 50, y: 50 }, {
    stiffness: 0.1,
    damping: 0.25
  });
</script>
```

初期値が `undefined` または `null` の場合、最初の値の変更は `tweened` の場合と同様に即座に有効になります（上記を参照）。

```
const size = spring();
$: $size = big ? 100 : 10;
```

svelte/transition

`svelte/transition` モジュールは7つの関数をエクスポートします。`fade`、`blur`、`fly`、`slide`、`scale`、`draw`、`crossfade` の7つの関数をエクスポートします。これらは Svelte `transitions` で使用します。

fade

```
transition:fade={params}
```

```
in:fade={params}
```

```
out:fade={params}
```

要素の `opacity` を、`in` トランジションでは 0 から現在の `opacity` まで、`out` トランジションでは現在の `opacity` から 0 までアニメーションします。

`fade` は以下のパラメータを受け付けます。

- `delay` (`number`, default 0) — 開始前の待ち時間のミリ秒

- duration (number , default 400) — トランジションの持続時間のミリ秒
- easing (function , default linear) — イージング関数

トランジション チュートリアル で fade トランジションの動作を見ることができます。

```
<script>
    import { fade } from 'svelte/transition';
</script>

{#if condition}
    <div transition:fade="{{delay: 250, duration: 300}}">
        fades in and out
    </div>
{/if}
```

blur

transition:blur={params}

in:blur={params}

out:blur={params}

要素の opacity で blur フィルタをアニメーション化します。

blur は以下のパラメータを受け付けます。

- delay (number , default 0) — 開始前の待ち時間のミリ秒
- duration (number , default 400) — アニメーションの持続時間のミリ秒
- easing (function , default cubicInOut) — イージング関数
- opacity (number , default 0) - アニメーション化する opacity の値
- amount (number , default 5) - ぼかしのサイズをピクセル単位で表します

```
<script>
    import { blur } from 'svelte/transition';
</script>

{#if condition}
    <div transition:blur="{{amount: 10}}">
        fades in and out
    </div>
{/if}
```

fly

transition:fly={params}

in:fly={params}

out:fly={params}

要素の `x` と `y` の位置と `opacity` をアニメーション化します。`in` トランジションは、要素の現在の(デフォルトの)値からパラメータとして渡された値にアニメーションします。`out` トランジションは、指定された値から要素のデフォルト値にアニメーションします。

`fly` は以下のパラメータを受け付けます。

- `delay (number , default 0)` — 開始前の待ち時間のミリ秒
- `duration (number , default 400)` — トランジションの持続時間のミリ秒
- `easing (function , default cubicOut)` — イージング関数
- `x (number , default 0)` - アニメーションで移動する `x` 位置のオフセット
- `y (number , default 0)` - アニメーションで移動する `y` 位置のオフセット
- `opacity (number , default 0)` - アニメーションで変化する `opacity` のオフセット

`fly` トランジションの動作は [トランジション チュートリアル](#) で見ることができます。

```
<script>
    import { fly } from 'svelte/transition';
    import { quintOut } from 'svelte/easing';
</script>

{#if condition}
    <div transition:fly="{{delay: 250, duration: 300, x: 100, y: 500, opacity: 0.5, easing: quintOut}}">
        flies in and out
    </div>
{#/if}
```

slide

```
transition:slide={params}
in:slide={params}
out:slide={params}
```

要素をスライドさせて出し入れします。

`slide` は下記のパラメータを受け付けます。

- `delay (number , default 0)` — 開始前の待ち時間のミリ秒
- `duration (number , default 400)` — トランジションの持続時間のミリ秒
- `easing (function , default cubicOut)` — イージング関数

```
<script>
    import { slide } from 'svelte/transition';
    import { quintOut } from 'svelte/easing';
</script>

{#if condition}
    <div transition:slide="{{delay: 250, duration: 300, easing: quintOut }}">
        slides in and out
    </div>
{#/if}
```

scale

`transition:scale={params}`

`in:scale={params}`

`out:scale={params}`

要素の `opacity` と `scale` をアニメーション化します。`in` トランジションは、要素の現在の(デフォルトの)値からパラメータとして渡された値にアニメーションします。`out` トランジションは、指定された値から要素のデフォルト値にアニメーションします。

`scale` は以下のパラメータを受け付けます。

- `delay` (`number`, default 0) — 開始前の待ち時間のミリ秒
- `duration` (`number`, default 400) — トランジションの持続時間のミリ秒
- `easing` (`function`, default `cubicOut`) — イージング関数
- `start` (`number`, default 0) - アニメーションで変化する `scale` の値
- `opacity` (`number`, default 0) - アニメーションで変化する `opacity` の値

```
<script>
    import { scale } from 'svelte/transition';
    import { quintOut } from 'svelte/easing';
</script>

{#if condition}
    <div transition:scale="{{duration: 500, delay: 500, opacity: 0.5, start: 0.5,
easing: quintOut}}">
        scales in and out
    </div>
{/if}
```

draw

`transition:draw={params}`

`in:draw={params}`

`out:draw={params}`

SVG 要素のストロークを蛇が管の中を進むようにアニメーション化します。`in` トランジションはパスが見えない状態から始まり、時間の経過とともにパスが画面に描画されます。`out` トランジションはパスが見える状態から始まり、徐々にパスを消していきます。`draw` は `<path>` や `<polyline>` のように `getTotalLength` メソッドを持つ要素でのみ動作します。

`draw` は以下のパラメータを受け付けます。

- `delay` (`number`, default 0) — 開始前の待ち時間のミリ秒
- `speed` (`number`, default undefined) - アニメーションの速度、下記を参照してください
- `duration` (`number` | `function`, default 800) — トランジションの持続時間のミリ秒
- `easing` (`function`, default `cubicInOut`) — イージング関数

速度パラメータ `speed` はパスの長さに対する遷移の持続時間を設定する手段です。これはパスの長さに適用される修飾子で `duration = length / speed` となります。1000ピクセルで速度が1のパスの持続時間は `1000ms` であり、速度を `0.5` に設定すると持続時間は2倍になります、`2` に設定すると半分になります。

```
<script>
  import { draw } from 'svelte/transition';
  import { quintOut } from 'svelte/easing';
</script>

<svg viewBox="0 0 5 5" xmlns="http://www.w3.org/2000/svg">
  {#if condition}
    <path transition:draw="{{duration: 5000, delay: 500, easing: quintOut}}"
      d="M2 1 h1 v1 h1 v1 h-1 v1 h-1 v-1 h-1 v-1 h1 z"
      fill="none"
      stroke="cornflowerblue"
      stroke-width="0.1px"
      stroke-linejoin="round"
    />
  {/if}
</svg>
```

crossfade

`crossfade` 関数は `send` と `receive` という **トランジション** のペアを作成します。ある要素が「送信」されると、それに対応する「受信」される要素を探し、その要素を相手の位置に変換してフェードアウトさせるトランジションを生成します。要素が「受信」されると、その逆が起こります。対応する要素がない場合は、`fallback` トランジションが使用されます。

`crossfade` は以下のパラメータを受け付けます:

- `delay (number, デフォルト 0)` — 開始するまでのミリ秒
- `duration (number | function, デフォルト 800)` — トランジションが継続するミリ秒
- `easing (function, デフォルト cubicOut)` — イージング関数
- `fallback (function)` — 受信している要素に一致するものがいない場合の送信時や、送信している要素がない場合の受信時に使用するフォールバックトランジションです。

```
<script>
  import { crossfade } from 'svelte/transition';
  import { quintOut } from 'svelte/easing';

  const [send, receive] = crossfade({
    duration: 1500,
    easing: quintOut
  });
</script>

{#if condition}
  <h1 in:send={{key}} out:receive={{key}}>BIG ELEM</h1>
{:else}
  <small in:send={{key}} out:receive={{key}}>small elem</small>
{/if}
```

svelte/animate

`svelte/animate` モジュールは、Svelte [animations](#) で使用するための関数を1つエクスポートします。

flip

`animate:flip={params}`

`flip` 関数は要素の開始位置と終了位置を計算し、その間で `x` と `y` の値を変換してアニメーションを行います。`flip` は [First](#), [Last](#), [Invert](#), [Play](#) の略です。

`flip` は以下のパラメータを受け付けます。

- `delay` (`number`, `default 0`) — 開始前の待ち時間のミリ秒
- `duration` (`number` | `function`, `default d => Math.sqrt(d) * 120`) — 下記を参照してください
- `easing` (`function`, `default cubicOut`) — イージング関数

`duration` は、以下のいずれかを指定することができます。

- `number` はミリ秒単位です。
- 関数 `distance: number => duration: number` は、要素の移動距離をピクセル単位で受け取り、ミリ秒単位で返します。これにより、各要素の持続時間に対する移動距離を割り当てることができます。

[アニメーションのチュートリアル](#) で全ての例を見るることができます。

```
<script>
    import { flip } from 'svelte/animate';
    import { quintOut } from 'svelte/easing';

    let list = [1, 2, 3];
</script>

{#each list as n (n)}
    <div animate:flip="{{delay: 250, duration: 250, easing: quintOut}}">
        {n}
    </div>
{/each}
```

svelte/easing

イージング関数は、時間の経過とともに変化する速度を指定するもので、Svelte に組み込まれたトランジションやアニメーション、`tweened` や `spring` ユーティリティの作業をする際に便利です。`svelte/easing` には、31の名前付きエクスポートが含まれています。`linear` イージング、10種類のイージング関数の3つのバリエーション `in`, `out`, `inOut` です。

[examples section](#) の `ease visualiser` で様々なイージングを試してみることができます。

```
| ease | in | out | inOut | | | --- | --- | --- | | back | backIn | backOut | backInOut | | bounce | bounceIn |
bounceOut | bounceInOut | | circ | circIn | circOut | circInOut | | cubic | cubicIn | cubicOut |
cubicInOut | | elastic | elasticIn | elasticOut | elasticInOut | | expo | expoIn | expoOut | expoInOut | |
quad | quadIn | quadOut | quadInOut | | quart | quartIn | quartOut | quartInOut | | quint | quintIn |
quintOut | quintInOut | | sine | sineIn | sineOut | sineInOut |
```

svelte/register

Svelteコンポーネントをビルドせずに Node.js でレンダリングするには、`require('svelte/register')` を使います。その後 `require` を使って `.svelte` ファイルをインクルードすることができます。

```
require('svelte/register');

const App = require('./App.svelte').default;
...
const { html, css, head } = App.render({ answer: 42 });
```

`.default` は、ネイティブの JavaScript モジュールから Node が認識する CommonJS モジュールに変換するために必要です。コンポーネントが JavaScript モジュールをインポートすると、Node での読み込みに失敗するので、代わりにバンドラを使う必要があることに注意してください。

コンパイルオプションを設定したり、カスタムファイルの拡張子を使用したりするには、`register` フックを関数として呼び出します。

```
require('svelte/register')({
  extensions: ['.customextension'], // defaults to ['.html', '.svelte']
  preserveComments: true
});
```

Client-side component API

Creating a component

```
const component = new Component(options)
```

クライアントサイドのコンポーネント、つまり `generate: 'dom'`（もしくは `generate` オプションを指定しないまま）でコンパイルされたコンポーネントは JavaScript のクラスです。

```
import App from './App.svelte';

const app = new App({
  target: document.body,
  props: {
    // App.svelte に `export let answer` のようなものが
    // 含まれていると仮定:
    answer: 42
  }
});
```

以下の初期化オプションを与えることができます。

| オプション | デフォルト | 説明 | | | --- | --- | | `target` | `none` | レンダリング先の `HTMLElement` または `ShadowRoot`。このオプションは必須です | `anchor` | `null` | `target` の子要素。これのすぐ前にコンポーネントがレンダリングされます | `props` | `{}` | コンポーネントに渡すプロパティのオブジェクト | `context` | `new Map()` | コンポーネントに提供するルートレベルの `context` のキーと値のペアの `Map` | `hydrate` | `false` | 下記参照 | `intro` | `false` | `true` なら、その後の状態変化を待つのではなく、初回レンダリング時にトランジションを再生します。

`target` の既存の子要素はそのまま残されます。

`hydrate` オプションは、新しい要素を作成するのではなく、既存の DOM を（大抵はサーバーサイドレンダリングから）アップグレードするよう Svelte に指示します。これはコンポーネントが `hydratable: true` のオプションでコンパイルされた場合にのみ機能します。`<head>` 要素のハイドレーションは、サーバーサイドレンダリングのコードも `hydratable: true` を使ってコンパイルされた場合にのみ適切に動作します。これは `head` 内の各要素にマーカーを追加して、コンポーネントがハイドレーション中にどの要素を除去すべきかを認識できるようにします。

通常、`target` の子要素はそのまま残されますが、`hydrate: true` ではすべての子要素が削除されます。そのため `anchor` オプションは `hydrate: true` と一緒に使用できません。

既存の DOM はコンポーネントと一致している必要はありません。Svelte は DOM をそのまま「修復」します。

```
import App from './App.svelte';

const app = new App({
  target: document.querySelector('#server-rendered-html'),
  hydrate: true
});
```

\$set

```
component.$set(props)
```

プログラムでインスタンスにプロパティ(props)をセットします。`component.$set({ x: 1 })` はコンポーネントの `<script>` ブロック内の `x = 1` と同じです。

このメソッドを呼ぶと次のマイクロタスクに更新がスケジュールされます。DOM は同期的に更新されません。

```
component.$set({ answer: 42 });
```

\$on

```
component.$on(event, callback)
```

コンポーネントが `event` をディスパッチするたびに、`callback` 関数が呼び出されるようにします。

呼び出されたときにイベントリスナーを削除する関数が返されます。

```
const off = app.$on('selected', event => {
  console.log(event.detail.selection);
});

off();
```

\$destroy

```
component.$destroy()
```

DOM からコンポーネントを削除し、すべての `onDestroy` ハンドラをトリガします。

Component props

```
component.prop
```

```
component.prop = value
```

コンポーネントが `accessors: true` でコンパイルされている場合、各インスタンスはコンポーネントの各プロパティ (props)に対するゲッターとセッターを持ちます。値をセットすると (`component.$set(...)`) によって起こるデフォルトの非同期更新ではなく)、*同期的な更新*が起こります。

カスタム要素としてコンパイルする場合を除き、デフォルトでは `accessors` は `false` です。

```
console.log(app.count);
app.count += 1;
```

Custom element API

Svelte コンポーネントは、`customElement: true` コンパイラオプションを使ってカスタム要素 (別名Webコンポーネント) にコンパイルすることもできます。コンポーネントのタグ名は `<svelte:options> element` で指定する必要があります。

```
<svelte:options tag="my-element" />

<script>
  export let name = 'world';
</script>

<h1>Hello {name}!</h1>
<slot></slot>
```

あるいは、`tag=null` を使って、カスタム要素の利用者がそれに名前を付けるべきであることを示します。

```
import MyElement from './MyElement.svelte';
customElements.define('my-element', MyElement);
```

一度カスタム要素が定義されると、それを通常の DOM 要素として使用することができます。

```
document.body.innerHTML =
<my-element>
  <p>This is some slotted content</p>
</my-element>
`;
```

デフォルトでは、カスタム要素は `accessors: true` でコンパイルされます。これは、任意の *プロパティ* が DOM 要素のプロパティとして公開されることを意味します (また、可能であれば属性として読み書き可能です)。

これを防ぐには、`<svelte:options>` に `accessors={false}` を追加します。

```
const el = document.querySelector('my-element');

// get the current value of the 'name' prop
console.log(el.name);

// set a new value, updating the shadow DOM
el.name = 'everybody';
```

カスタム要素は、非Svelteアプリで利用するためのコンポーネントをパッケージ化するのに便利な方法です。それらは純粋な HTML と JavaScript の同様に、[ほとんどのフレームワーク](#) でも動作します。しかし、注意すべき重要な違いがいくつかあります。

- スタイルは単なる `scoped` ではなく `encapsulated`（カプセル化）です。これは、`:global(...)` 修飾子を持つスタイルを含む、コンポーネントにはないスタイル（`global.css` ファイルにあるような）はカスタム要素には適用されないことを意味します。
- スタイルは、別の `.css` ファイルとして抽出されるのではなく、JavaScript の文字列としてコンポーネントにインライン化されます。
- JavaScript が読み込まれるまでシャドウ DOM は見えないので、カスタム要素は一般的にサーバーサイドのレンダリングには適していません。
- Svelte では、スロットコンテンツは 遅延してレンダリングされます。DOMでは 先行してレンダリングします。言い換えれば、コンポーネントの `<slot>` 要素が `{#if ...}` ブロックの中にあっても、常に作成されます。同様に、`{#each ...}` ブロックの中に `<slot>` 要素を含めても、スロットの内容が何度もレンダリングされることはありません。
- `let:` ディレクティブは何の効果もありません。
- 古いブラウザをサポートするにはポリフィルが必要です。

Server-side component API

```
const result = Component.render(...)
```

クライアント側のコンポーネントとは異なり、サーバー側のコンポーネントはレンダリングしてもライフサイクルがありません。-- それらは HTML と CSS を作成するのが仕事です。そのため API が多少異なります。

サーバーサイドコンポーネントは任意のプロパティ(`props`)と一緒に呼びだせる `render` メソッドを公開しています。これは `head`、`html`、`css` プロパティを持つオブジェクトを返します。この `head` は見つけた `<svelte:head>` 要素の内容を含みます。

Svelte コンポーネントを直接 Node にインポートするには、`svelte/register` を使ってください。

```
require('svelte/register');

const App = require('./App.svelte').default;

const { head, html, css } = App.render({
    answer: 42
});
```

`.render()` メソッドは以下のパラメータを受け付けます：

| パラメータ | デフォルト | 説明 | | --- | --- | | `props` | `{}` | コンポーネントに渡すプロパティのオブジェクト |
`options` | `{}` | オプションのオブジェクト

`options` オブジェクトは、以下のオプションを取ります:

| オプション | デフォルト | 説明 | | --- | --- | | `context` | `new Map()` | コンポーネントに提供するルートレベルの
`context` のキーと値のペアの `Map`

```
const { head, html, css } = App.render(  
  // props  
  { answer: 42 },  
  // options  
  {  
    context: new Map([['context-key', 'context-value']])  
  }  
);
```

Compile time

通常、Svelte コンパイラと直接やり取りすることはありません。その代わり、バンドラープラグイン(bundler plugin)を使ってビルドシステムにインテグレートします。Svelte チームが最も推奨している、また注力もしているバンドラープラグインは [vite-plugin-svelte](#) です。SvelteKit フレームワークは `vite-plugin-svelte` を活用し、アプリケーションをビルドするためのセットアップと、[Svelte コンポーネントライブラリをパッケージングするツール](#)を提供しています。Svelte Society には、Rollup や Webpack などのツール向けの [その他のバンドラープラグイン](#) のリストがあります。

とはいっても、バンドラープラグインは一般的にコンパイラのオプションを公開しているので、コンパイラの使い方を理解しておくと便利です。

svelte.compile

```
result: {
  js,
  css,
  ast,
  warnings,
  vars,
  stats
} = svelte.compile(source: string, options?: {...})
```

ここでマジックが起こります。`svelte.compile` はコンポーネントのソースコードを受け取ります。そしてそれを使って、クラスをエクスポートするJavaScriptモジュールに変えます。

```
const svelte = require('svelte/compiler');

const result = svelte.compile(source, {
  // options
});
```

以下のオプションをコンパイラに渡すことができます。どれも必須ではありません。

`| option | default | description || | --- | --- || | filename | null |` デバッグのヒントやソースマップに使われる `string` です。バンドラープラグインが自動的に設定します。`| name | "Component"` | 結果として得られるJavaScriptクラスの名前を設定する `string` です(ただし、スコープ内の他の変数と競合する場合はコンパイラが名前を変更します)、通常は `filename` から推測されます。`| format | "esm" | "esm"` の場合、JavaScriptモジュールを作成します(`import` と `export` を指定します)、`"cjs"` の場合、CommonJS モジュールを作成します(`require` と `module.exports` を指定します)、これは、いくつかのサーバーサイドのレンダリング状況やテストに便利です。`| generate | "dom" | "dom"` の場合、SvelteはDOMにマウントするためのJavaScriptクラスを生成します。`"ssr"` の場合、サーバサイドのレンダリングに適した `render` メソッドを持つオブジェクトを出力します。`false` の場合、JavaScriptやCSSは返されず、メタデータだけが返されます。`| errorMode | "throw" | "throw"` の場合、Svelteはコンパイルエラーが発生したときにエラーをスローします。`"warn"` の場合、Svelteはエラーを警告として扱い、その警告を `warning report` に追加します。`| varsReport | "strict" | "strict"` の場合、Svelteはグローバルまたはインターナルな変数以外のみの変数レポートを返します。`"full"` の場合は検出された全ての変数を返します、`false` の場合は変数レポートは返しません。`| dev | false | true` の場合、コンポーネントに特別なコードを追加します。これは、ランタイムチェックを実行し、開発中にデバッグ情報を提供するためのものです。`| immutable | false | true` の場合、オプ

ジェクトを変更させないことをコンパイラに伝えます。これにより、値が変更されたかどうかのチェックをより控えめにすることができます。| `hydratable` | `false` | `true` を指定すると、DOMコードを生成する際に `hydrate: true` ランタイムオプションが有効になり、新しいDOMをゼロから生成するのではなく、既存のDOMをアップグレードすることができます。これにより、SSRコードを生成する際に `<head>` 要素にマーカーが追加され、ハイドレーションがどれを置き換えるべきかを知ることができます。| `legacy` | `false` | `true` ならば、`element.dataset` のようなものをサポートしていないIE9とIE10で動作するコードを生成します。| `accessors` | `false` | `true` の場合、ゲッターとセッターはコンポーネントのプロパティ(props)に対して作成されます。`false` の場合、それらは読み書きされた値に対してのみ作成されます。(つまり `const`, `class`, `function` で宣言されたもの)`customElement: true` でコンパイルした場合、このオプションのデフォルトは `true` です。| `customElement` | `false` | `true` ならば、コンパイラに通常の Svelteコンポーネントの代わりにカスタム要素のコンストラクタを生成するように指示します。| `tag` | `null` | Svelte にカスタム要素を登録するタグ名を指定する `string`。文字列は小文字の英数字で、少なくとも1つのハイフンを含んだ文字列でなければなりません。例えば `"my-element"` . | `css` | `'injected'` | `'injected'` (以前は `true`) の場合、スタイルはJavaScriptクラスに含まれ、実行時にコンポーネントが実際にレンダリングされるときに注入されます。`'external'` (以前は `false`) の場合、CSS はコンパイル結果の `css` フィールドで返されます。多くの Svelte バンドラープラグインはこれを `external` に設定して静的に生成されたCSSを使用するので、JavaScriptのバンドルが小さくなり、キャッシュ可能な `.css` ファイルをサーブすることができるようになります。パフォーマンスが向上します。`'none'` の場合、スタイルは完全に取り除かれ、CSS 出力は生成されません。| `cssHash` | 右記 | `{ hash, css, name, filename }` を引数に取り、スコープ付きCSSのクラス名として使われる文字列を返す関数。デフォルトでは、`svelte-$\{hash(css)\}` を返します。| `loopGuardTimeout` | `0` | `loopGuardTimeout` msを超えてスレッドがブロックされた場合にループを解除するように Svelteに指示する `数値` です。これは無限ループを防ぐのに便利です。利用可能なのは `dev: true` の場合のみです | `preserveComments` | `false` | `true` の場合、サーバサイドでのレンダリング中に HTML コメントが保存されます。デフォルトではコメントは削除されます。| `preserveWhitespace` | `false` | `true` の場合、要素内や要素間の空白は、可能であれば削除されたり单一の空白になったりするのではなく、入力したとおりに保持されます。| `sourcemap` | `object` \| `string` | 最終的な出力ソースマップにマージされる初期のソースマップです。これは通常、プリプロセッサのソースマップです。| `enableSourcemap` | `boolean` \| `{ js: boolean; css: boolean; }` | `true` の場合、Svelteはコンポーネントのソースマップを生成します。ソースマップ生成をより詳細に制御するには、`js` または `css` を持つオブジェクトを使用します。デフォルトでは `true` です。| `outputFilename` | `null` | JavaScriptのソースマップに使われる `文字列` です。| `sveltePath` | `"svelte"` | `svelte` パッケージの場所。`svelte` または `svelte/[module]` からのインポートは、それに応じて変更されます。| `namespace` | `"html"` | 要素の名前空間。例えば、`"mathml"`, `"svg"`, `"foreign"`

返ってきた `result` オブジェクトには、有用なメタデータとともにコンポーネントのコードが含まれます。

```
const {
  js,
  css,
  ast,
  warnings,
  vars,
  stats
} = svelte.compile(source);
```

- `js` と `css` は以下のプロパティを持つオブジェクトです。
 - `code` は JavaScript の文字列です。
 - `map` はソースマップに `toString()` と `toUrl()` の便利なメソッドを追加したものです。
- `ast` はコンポーネントの構造を表す抽象構文ツリーです。

- `warnings` はコンパイル時に生成された警告オブジェクトの配列です。各警告にはいくつかのプロパティがあります。
 - `code` は警告のカテゴリを識別する文字列です。
 - `message` は人間が読みやすい言葉で問題を説明したものです。
 - `start` and `end` はもしも警告が特定の場所に関連している場合には `line`, `column`, `character` プロパティを持つオブジェクトです。
 - `frame` は該当する場合に問題のあるコードを行番号で強調表示する文字列です。
- `vars` はコンポーネントの宣言の配列です、例えば [eslint-plugin-svelte3](#) で使用されているように、各変数はいくつかのプロパティを持っています。
 - `name` はそのままの意味です
 - `export_name` は、エクスポートされた場合に使用される名前です。(`export...as` でない限り、`name` と一致します)
 - `injected` が `true` ののは、宣言はあなたが書いたコードではなく、Svelteによって注入されている場合です。
 - `module` が `true` ののは、モジュールの値が `context="module"` スクリプトで宣言されている場合です。
 - `mutated` が `true` ののは、値のプロパティがコンポーネント内部に割り当てられている場合です。
 - `reassigned` が `true` ののは、コンポーネント内で値が再割り当てされている場合です。
 - `referenced` が `true` ののは、テンプレート内で値が使われている場合です。
 - `referenced_from_script` が `true` ののは、宣言外の `<script>` の中で値が使われている場合です。
 - `writable` が `true` ののは、値が `let` または `var` で宣言されている場合です。(ただし `const`, `class` と `function` は除きます)
- `stats` はSvelte開発チームがコンパイラを診断するために使用するオブジェクトです。これに依存してはいけません！

svelte.parse

```
ast: object = svelte.parse(
  source: string,
  options?: {
    filename?: string,
    customElement?: boolean
  }
)
```

`parse` 関数はコンポーネントを解析し、その抽象構文木のみを返します。`generate: false` オプションを指定してコンパイルするのとは異なり、これはコンポーネントを解析する以外の検証やその他の解析を行いません。戻り値の AST はパブリックな API とは見なされないため、将来どこかのタイミングで破壊的な変更が発生する可能性があることにご注意ください。

```
const svelte = require('svelte/compiler');
const ast = svelte.parse(source, { filename: 'App.svelte' });
```

svelte.preprocess

Svelte を TypeScript, PostCSS, SCSS, Less などのツールで利用できるようにするための [コミュニティでメンテナスされているプリプロセッサプラグイン](#) が多数用意されています。

`svelte.preprocess` APIを使って独自のプリプロセッサを書くことができます。

```

result: {
  code: string,
  dependencies: Array<string>
} = await svelte.preprocess(
  source: string,
  preprocessors: Array<{
    markup?: (input: { content: string, filename: string }) => Promise<{
      code: string,
      dependencies?: Array<string>
    }>,
    script?: (input: { content: string, markup: string, attributes: Record<string, string>, filename: string }) => Promise<{
      code: string,
      dependencies?: Array<string>
    }>,
    style?: (input: { content: string, markup: string, attributes: Record<string, string>, filename: string }) => Promise<{
      code: string,
      dependencies?: Array<string>
    }>,
    options?: {
      filename?: string
    }
)

```

`preprocess` 関数は、コンポーネントのソースコードを任意に変換するための便利なフックを提供します。例えば、`<style lang="sass">` ブロックを純粋なCSSに変換するために使うことができます。

最初の引数はコンポーネントのソースコードです。2番目の引数は、プリプロセッサの配列です（1つしかない場合は単独のプリプロセッサになります）。このプリプロセッサは `markup`, `script`, `style` 関数を持つオブジェクトであり、これらは全てオプションです。

各 `markup`, `script`, `style` 関数は、変換されたソースコードを表す `code` プロパティと、任意で `dependencies` の配列を含んだオブジェクト（またはオブジェクトを `resolve` する `promise`）を返さなければなりません。

`markup` 関数は、コンポーネントのソーステキスト全体と、第3引数にコンポーネントの `filename` が指定されている場合はそのコンポーネントの `filename` を受け取ります。

プリプロセッサ関数は、`code` や `dependencies` に加えて `map` オブジェクトを返すことがあります。この `map` は変換を表すソースマップです。

```

const svelte = require('svelte/compiler');
const MagicString = require('magic-string');

const { code } = await svelte.preprocess(source, {
  markup: ({ content, filename }) => {
    const pos = content.indexOf('foo');
    if(pos < 0) {
      return { code: content }
    }
    const s = new MagicString(content, { filename })
    s.overwrite(pos, pos + 3, 'bar', { storeName: true })
    return {
      code: s.toString(),
    }
}

```

```

        map: s.generateMap()
    }
},
{
    filename: 'App.svelte'
});

```

`script` 関数と `style` 関数はそれぞれ `<script>` と `<style>` 要素の内容(`content`)とコンポーネントのソーステキスト全体(`markup`)を受け取ります。これらの関数は `filename` に加えて要素の属性のオブジェクトを取得します。

依存関係 の配列が返された場合、それが結果オブジェクトに含まれます。これは（例えば）[rollup-plugin-svelte](#) のようなパッケージで、`<style>` タグに `@import` がある場合などに、追加ファイルの変更を監視するために使われます。

```

const svelte = require('svelte/compiler');
const sass = require('node-sass');
const { dirname } = require('path');

const { code, dependencies } = await svelte.preprocess(source, {
    style: async ({ content, attributes, filename }) => {
        // only process <style lang="sass">
        if (attributes.lang !== 'sass') return;

        const { css, stats } = await new Promise((resolve, reject) =>
            sass.render({
                file: filename,
                data: content,
                includePaths: [
                    dirname(filename),
                ],
            }, (err, result) => {
                if (err) reject(err);
                else resolve(result);
            }));
        return {
            code: css.toString(),
            dependencies: stats.includedFiles
        };
    }
},
{
    filename: 'App.svelte'
});

```

複数のプリプロセッサを併用することができます。最初のプリプロセッサの出力は、2番目のプリプロセッサへの入力になります。最初に `markup` 関数が実行され、次に `script` と `style` が実行されます。

```

const svelte = require('svelte/compiler');

const { code } = await svelte.preprocess(source, [
    {
        markup: () => {
            console.log('this runs first');
        },
        script: () => {
            console.log('this runs third');
        },
        style: () => {
            console.log('this runs fifth');
        }
    }
]

```

```

    },
    {
      markup: () => {
        console.log('this runs second');
      },
      script: () => {
        console.log('this runs fourth');
      },
      style: () => {
        console.log('this runs sixth');
      }
    }
  ],
  {
    filename: 'App.svelte'
  });

```

svelte.walk

```

walk(ast: Node, {
  enter(node: Node, parent: Node, prop: string, index: number)?: void,
  leave(node: Node, parent: Node, prop: string, index: number)?: void
})

```

`walk` 関数はパーサーによって生成された抽象構文木をウォークする方法を提供します。コンパイラの組み込みインスタンスである`estree-walker`を使用します。

ウォーカーは歩くための抽象構文木と、オプションの2つのメソッド `enter` と `leave` を持つオブジェクトを受け取ります。各ノードに対して、(存在すれば) `enter` が呼び出されます。そして `enter` を実行している間に `this.skip()` が呼ばれない限り、各子プロセスを巡回した後、ノード上で `leave` が呼ばれます。

```

const svelte = require('svelte/compiler');
svelte.walk(ast, {
  enter(node, parent, prop, index) {
    do_something(node);
    if (should_skip_children(node)) {
      this.skip();
    }
  },
  leave(node, parent, prop, index) {
    do_something_else(node);
  }
});

```

svelte.VERSION

`package.json` で設定されている現在のバージョンです。

```

const svelte = require('svelte/compiler');
console.log(`running svelte version ${svelte.VERSION}`);

```

[Go to TOC](#)

Accessibility warnings

アクセシビリティ (a11y と略されます) を正しく理解することは容易ではありませんが、Svelte は、アクセシブルではないマークアップを書くとコンパイル時に警告してくれます。しかし、多くのアクセシビリティの問題は、他の自動化されたツールを使用したり、手動でアプリケーションをテストするなど、実行時に特定できることを忘れないでください。

Svelte が行うアクセシビリティチェックのリストは以下の通りです。

a11y-accesskey

要素に `accesskey` を設定しないように強制します。アクセスキーとは、Web 開発者が要素にキーボードのショートカットを割り当てることができる HTML 属性です。キーボードショートカットと、スクリーンリーダーやキーボードのみのユーザが使用するキーボードコマンドの間に不整合があるとアクセシビリティ対応が複雑になるので、複雑さを避けるためにもアクセスキーを使用してはいけません。

```
<!-- A11y: accesskeyの使用を避けましょう -->
<div accessKey='z'></div>
```

a11y-aria-attributes

DOM 要素の中には、ARIA role やステート、プロパティをサポートしていないものがあります。これは `meta`、`html`、`script`、`style` などのように、表示されないものがあるからです。このルールは、これらの DOM 要素が `aria-*` プロパティを含まないことを強制します。

```
<!-- A11y: <meta>はaria-*属性を持つべきではありません -->
<meta aria-hidden="false">
```

a11y-autofocus

要素で `autofocus` が使われないよう強制します。オートフォーカス要素は、目の見える人にも見えない人にもユーザビリティの問題を引き起こす可能性があります。

```
<!-- A11y: autofocusの使用を避けましょう -->
<input autofocus>
```

a11y-click-events-have-key-events

`on:click` が `onKeyUp`、`onKeyDown`、`onKeyPress` のうち少なくともいずれか1つを伴って使用されることを矯正します。キーボード操作用のコーディングは、マウスを使用することができないユーザーや、支援技術(AT)の互換性、スクリーンリーダーを使用するユーザーのためにとても重要です。

これはインタラクティブな要素や隠し要素(hidden elements)には適用されません。

```
<!-- A11y: visible, non-interactive elements with an on:click event must be
accompanyed by an on:keydown, on:keyup, or on:keypress event. -->
<div on:click={() => {}}> />
```

a11y-distracting-elements

気が散るような要素が使われていないかを確認します。視覚的に邪魔になる要素は、視覚障害のあるユーザにアクセシビリティ上の問題を引き起こす可能性があります。このような要素は、ほとんどの場合非推奨であり、避けるべきです。

以下の要素は視覚的に気を散らす要素の `<marquee>` と `<blink>` です。

```
<!-- A11y: <marquee>要素の使用を避けましょう -->
<marquee />
```

a11y-hidden

一部の DOM 要素は、スクリーンリーダーのナビゲーションに有用であるため、非表示にすべきではありません。

```
<!-- A11y: <h2>要素を非表示にしてはなりません -->
<h2 aria-hidden="true">invisible header</h2>
```

a11y-img-redundant-alt

`img` の `alt` 属性には、`image`、`picture`、または `photo` という単語は含んではいけません。スクリーンリーダーは、すでに `img` 要素を画像として認識しています。`image`、`photo`、または `picture` のような単語を使う必要はありません。

```


<!-- aria-hiddenによりスクリーンリーダーでは読み上げられません -->


<!-- A11y: スクリーンリーダーでは、すでに<img>要素を「画像」として読み上げています -->


<!-- A11y: スクリーンリーダーでは、すでに<img>要素を「画像」として読み上げています -->


<!-- A11y: スクリーンリーダーでは、すでに<img>要素を「画像」として読み上げています -->

```

a11y-incorrect-aria-attribute-type

ARIA 属性に正しいタイプの値のみが使用されることを強制します。例えば、`aria-hidden` は `boolean` のみを受け取ります。

```
<!-- A11y: The value of 'aria-hidden' must be exactly one of true or false -->
<div aria-hidden="yes"/>
```

a11y-invalid-attribute

アクセシビリティ属性が有効な値であることを強制します。例えば `href` を空にすべきではないし、`'#'` や `javascript:` にすべきではありません。

```
<!-- A11y: ''は有効なhref属性ではありません -->
<a href=''>invalid</a>
```

a11y-label-has-associated-control

ラベルタグは、テキストラベルと関連するコントロールを持つことを強制します。

ラベルとコントロールの関連付けには、次の2つの方法があります。

- コントロールをラベルタグで囲む。
- ラベルに `for` を追加し、ページ上の入力を示す ID 文字列を割り当てます。

```
<label for="id">B</label>
<label>C <input type="text" /></label>
<!-- A11y: フォームラベルは、コントロールに関連付ける必要があります -->
<label>A</label>
```

a11y-media-has-caption

メディアにキャプションを提供することは、耳の不自由なユーザが情報を得るために不可欠です。キャプションは、ダイアログ、サウンドエフェクト、関連する音楽のキー、およびその他の関連するオーディオ情報の文字起こしまたは翻訳がなければなりません。これはアクセシビリティ上重要であるだけでなく、メディアが利用できない場合にすべてのユーザにとって有用です（画像が読み込めない場合に画像上に `alt` テキストを表示するのと同様です）。

キャプションには、対応するメディアを理解するための重要な関連情報をすべて含める必要があります。これは、キャプションがメディアコンテンツのダイアログと1対1で対応していないことを意味します。ただし、キャプションは `muted` 属性を持つビデオコンポーネントには必要ありません。

```
<video><track kind="captions"/></video>
<audio muted></audio>
<!-- A11y: メディア要素には、<track kind="captions"/>が必要です -->
<video></video>
<!-- A11y: メディア要素には、<track kind="captions"/>が必要です -->
<video><track /></video>
```

a11y-misplaced-role

DOM 要素の中には、ARIA role やステート、プロパティをサポートしていないものがあります。これは `meta`、`html`、`script`、`style` などのように、表示されないものがあるからです。このルールは、これらの DOM 要素が `role` プロパティを含んでいないことを強制します。

```
<!-- A11y: <meta>はrole属性を持つべきではありません -->
<meta role="tooltip">
```

a11y-misplaced-scope

scope 属性は、`<th>` 要素でのみ使用してください。

```
<!-- A11y: scope属性は、<th>要素でのみ使用されます -->
<div scope="row" />
```

a11y-missing-attribute

アクセシビリティに必要な属性が要素上に存在することを強制します。以下のチェックが含まれます。

- `<a>` には href が必要です ([fragment-defining tag](#) ではない限り)
- `<area>` には alt、aria-label または aria-labelledby が必要です
- `<html>` には lang が必要です
- `<iframe>` には title が必要です
- `` には alt が必要です
- `<object>` には title、aria-label または aria-labelledby が必要です
- `<input type="image">` には alt、aria-label または aria-labelledby が必要です。

```
<!-- A11y: <input type=\"image\">要素にはalt、aria-label、aria-labelledby属性が必要です -->
<input type="image">

<!-- A11y: <html>要素は、lang属性を持つ必要があります -->
<html></html>

<!-- A11y: <a>要素にはhref属性が必要です -->
<a>text</a>
```

a11y-missing-content

見出し要素（`h1`、`h2` など）とアンカーに対し、コンテンツを持つこと、そのコンテンツがスクリーンリーダーからアクセス可能であることを強制します。

```
<!-- A11y: <a>要素は子コンテンツを持つべきです -->
<a href='/foo'></a>

<!-- A11y: <h1>要素は子コンテンツを持つべきです -->
<h1></h1>
```

a11y-mouse-events-have-key-events

`on:mouseover` と `on:mouseout` に対し、それぞれ `on:focus` と `on:blur` を付けることを強制します。これにより、これらのマウスイベントによってトリガされる機能が、キーボードユーザーもアクセス可能であることが保証されます。

```
<!-- A11y: on:mouseover must be accompanied by on:focus -->
<div on:mouseover={handleMouseover} />

<!-- A11y: on:mouseout must be accompanied by on:blur -->
<div on:mouseout={handlemouseout} />
```

a11y-no-redundant-roles

HTML の要素には、デフォルトで ARIA role を持つものがあります。対象の要素に、すでにブラウザで設定されている ARIA role を与えても効果はなく、冗長になるだけです。

```
<!-- A11y: 冗長な role 'button' -->
<button role="button" />

<!-- A11y: 冗長な role 'img' -->

```

a11y-no-interactive-element-to-noninteractive-role

WAI-ARIA role を、インタラクティブな要素を非インタラクティブな要素に変換するために使用してはいけません。非インタラクティブな ARIA role には、`article`、`banner`、`complementary`、`img`、`listitem`、`main`、`region`、`tooltip` が含まれます。

```
<!-- A11y: <textarea> cannot have role 'listitem' -->
<textarea role="listitem" />
```

a11y-no-noninteractive-tabindex

タブキーでのナビゲーションは、ページ上のインタラクティブに操作できる要素に限定する必要があります。

```
<!-- A11y: noninteractive element cannot have positive tabIndex value -->
<div tabIndex='0' />
```

a11y-positive-tabindex

`tabIndex` プロパティを正の値にすることは避けてください。要素が期待されるタブの順序から外れてしまい、キーボードユーザーに混乱を招くことになります。

```
<!-- A11y: tabIndexの値が0を超えないようにする -->
<div tabIndex='1' />
```

a11y-role-has-required-aria-props

ARIA role を持つ要素は、その role に必要な属性をすべて持つ必要があります。

```
<!-- A11y: A11y: Elements with the ARIA role "checkbox" must have the following
attributes defined: "aria-checked" -->
<span role="checkbox" aria-labelledby="foo" tabIndex="0"></span>
```

a11y-structure

特定の DOM 要素が正しい構造を持つことを強制します。

```
<!-- A11y: <figcaption>は、<figure>の直接の子でなければなりません -->
<div>
  <figcaption>Image caption</figcaption>
</div>
```

a11y-unknown-aria-attribute

WAI-ARIA States and Properties specに基づいて、既知の ARIA 属性のみを使用することを強制します。

```
<!-- A11y: 不明なaria属性 'aria-labeledby' ('labelledby'ではないでしょうか) -->
<input type="image" aria-labeledby="foo">
```

a11y-unknown-role

ARIA role を持つ要素は有効で、抽象的でない ARIA role を使用しなければなりません。role の定義については、[WAI-ARIA](#)サイトを参照してください。

```
<!-- A11y: 不明な'toooltip' role ('tooltip'ではないでしょうか) -->
<div role="toooltip"></div>
```

[Go to TOC](#)

Colophon

This book is created by using the following sources:

- Svelte - 日本語
- GitHub source: [svelte-jp/svelte-site-jp/site/content/docs](https://github.com/svelte-jp/svelte-site-jp/site/content/docs)
- Created: 2022-11-28
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>