# [Svelte 官方入门教程（1）—— 简介](https://zhuanlan.zhihu.com/p/367431533)

在介绍 Svelte 之前，首先解决的要务问题是：

Svelte 不是 Servlet，Svelte 是一个前端 Web 框架，而 Java Servlet 与 Svelte 不是一个类型的程序，务请明鉴。

如有误入，请出门右转。

## 一、初窥门径

花径不曾缘客扫，教程今始为君开。

本教程循序渐进，为你倾囊相授如何使用 Svelte 轻松创建快速小巧的应用。

如果你想深入精微，探求透彻，不妨随时翻阅 [API 文档](https://zhuanlan.zhihu.com/p/372950225) 及 [示例](https://link.zhihu.com/?target=https%3A//svelte.dev/examples)。

倘若你现在就迫不及待想在本机上开发，[60秒极速入门](https://zhuanlan.zhihu.com/p/351561304) 正是为你而准备的。

本教程所有的示例都可以在线试验，你在学习过程中无需创建任何项目或者安装软件包，点击下方的链接进入 REPL，即可边看教程，边实践：

[https://svelte.dev/repl](https://link.zhihu.com/?target=https%3A//svelte.dev/repl)

1. **什么是 Svelte？**

Svelte 用于构建快速的 Web 应用。

Svelte 类似 React 和 Vue，都致力让你轻而易举地构建灵活的可交互的用户界面。

与它们截然不同的是：Svelte 在*构建时* 将你的代码转为更优的 JavaScript，而不是在 *运行时* 才解释执行你的代码。

这预示着你无需付出框架本身的性能成本，且首次加载也无额外性能损耗。

你可以使用 Svelte 编写整个应用，也可以用来逐步重构现有代码，整半皆可；还可以只输出单个独立组件（无需强制附带框架自身）并将其用于任意框架中，可谓百面玲珑。

**2. 如何使用本教程？**

首先，你至少已对 HTML、CSS 和 JavaScript 有过初步了解，否则学习过程将会如堕烟海，茫然失措。

请跟随本教程精心规划的步伐按部就班，每一小节在介绍新知识时，都会配套一个小小练习。所有章节的学习次第承前启后，切忌肆意跳跃阅读。

**3. 理解组件**

一个 Svelte 应用由一个或多个*组件*（Components）组合而成。

每个组件对应一个可重用的 .svelte 文件，包含了组件的 HTML、CSS 和 JavaScript 全部代码。

以下是一个最简单的组件：

<h1>Hello world!</h1>

## 二、添加数据

只呈现一些标签的组件让人感觉索然无趣，我们尝试加些数据。

首先，在组件中添加一个 script 标签，并声明一个 name 变量：

<script>

let name = 'world';

</script>

<h1>Hello world!</h1>

然后，你可以在标记中使用这个 name 变量：

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

在花括号中，我们可以编写任意 JavaScript。

你接着可以尝试以下把 name 改为 name.toUpperCase()，这样打招呼显得更强烈。

注意：你看似创建的是全局变量，实际上，name 仍然只是局限于组件的局部变量，也就是说，你无法在全局对象（Window）中访问到组件中声明的 name 变量。

## 三、动态属性

花括号除了可以控制文本以外，同时适用于元素的属性。以下方代码为例：

<script>

let src = 'tutorial/image.gif';

</script>

<img>

下方的图片标记缺少了 src 特性属性，现在加上试试：

<img src={src}>

小试牛刀，初露锋芒。

但是 Svelte 在 build 时仍然给出一条警告：

A11y: <img> element should have an alt attribute

据警告所述，img 元素应该加上 alt 属性。

此举的目的是期望你的应用尽可能扩大受众，顾及全民，包括视力欠佳、运动障碍、低端硬件或网速缓慢的人群。

可访问性（Accessibility，缩写 a11y）往往费时费力，如果你编写了有违可访问性的标记，Svelte 会通过警告来提醒你。

在这种情况下，我们缺少了为使用屏幕阅读器的用户描述图像的 alt 属性，或者为网络连接缓慢或不稳定、无法下载图像的用户描述图像。让我们添加一个：

<img src={src} alt="A man dances.">

我们可以在 *属性内* 使用花括号。

尝试将它修改为 "{name} dances."，记得在 <script> 标签中声明 name 变量。

**属性缩写**

名称和值相同的属性并不少见，例如 src={src}。

Svelte 为我们提供了这种情况的便捷速记缩写方式：

<img {src} alt="A man dances.">

## 四、样式

就像在HTML中一样，您可以在组件中添加<style>标记。

<style>

/\* 样式写在此处 \*/

</style>

<p>这是一段话。</p>

让我们为<p>元素添加一些样式：

App.svlete

<style>

p {

color: purple;

font-family: 'Comic Sans MS', cursive;

font-size: 2em;

}

</style>

<p>这是一段话。</p>

重要的是，这些规则是“作用域内的组件”。 您不会在应用程序的其他地方意外更改<p>元素的样式，正如我们将在下一步中看到的那样。

## 五、嵌套组件

将整个应用程序放在一个组件中是不切实际的。 取而代之的是，我们可以从其他文件中导入组件，并像包含元素一样包含它们。

以下是我们的 App.svelte 组件

<style>

p {

color: purple;

font-family: 'Comic Sans MS', cursive;

font-size: 2em;

}

</style>

<p>This is a paragraph.</p>

我们先写一个 Nested 组件：

Nested.svelte

<p>This is another paragraph.</p>

接着，我们在 App.svelte 中添加一个<script>标记并导入Nested.svelte...

<script>

import Nested from './Nested.svelte';

</script>

...然后可以在代码中使用它：

<p>这是一个段落。</p>

<Nested/>

请注意，即使 Nested.svelte 具有 <p> 元素，App.svelte 的样式也不受影响。

另请注意，组件名称 Nested 首字母是大写的。 采用此约定是为了使我们能够区分用户定义的组件和常规的HTML标签。

## 六、HTML 标签

通常，字符串以纯文本形式插入，这意味着像 < 和 > 这样的字符没有特殊含义。

但是有时您需要直接将HTML呈现到组件中。 例如，您正在阅读的单词现在存在于markdown文件中，该文件以HTML的形式包含在此页面中：

<script>

let string = `this string contains some <strong>HTML!!!</strong>`;

</script>

<p>{string}</p>

Svelte 中你可以使用特殊的标签 {@html ...} 来解决这个问题：

<p>{@html string}</p>

Svelte 不会在插入到DOM中之前对 {@html ...} 中的表达式进行任何清理。换句话说，如果您使用此功能，则至关重要的是，您必须手动转义来自不信任来源的HTML，否则就有使用户遭受XSS攻击的风险。

## 七、创建一个 App

本教程旨在使您熟悉编写组件的过程。 但是有时，您将希望在自己的文本编辑器中开始编写组件。

首先，您需要将Svelte与构建工具集成在一起。 有针对 [Rollup](https://link.zhihu.com/?target=https%3A//rollupjs.org/) 和 [webpack](https://link.zhihu.com/?target=https%3A//webpack.js.org/) 的官方维护的插件...

* [rollup-plugin-svelte](https://link.zhihu.com/?target=https%3A//github.com/sveltejs/rollup-plugin-svelte)
* [svelte-loader](https://link.zhihu.com/?target=https%3A//github.com/sveltejs/svelte-loader)

...以及各种由 [社区维护](https://link.zhihu.com/?target=https%3A//github.com/sveltejs/integrations%23bundler-plugins) 的插件。

如果您是 Web 开发的新手，并且以前没有使用过这些工具，请不要担心。 我们已经为新开发人员准备了一个[简单的 Svelte 指南](https://zhuanlan.zhihu.com/p/353316935)，它将引导您完成整个过程。

您还需要配置文本编辑器。 如果您使用的是VS Code，请安装 [Svelte扩展名](https://link.zhihu.com/?target=https%3A//marketplace.visualstudio.com/items%3FitemName%3Dsvelte.svelte-vscode)，否则请按照 [本指南](https://link.zhihu.com/?target=https%3A//svelte.dev/blog/setting-up-your-editor) 的说明将文本编辑器配置为将 .svelte 文件与 .html 相同，以便突出显示语法。

然后，一旦建立了项目，就可以轻松使用Svelte组件。 编译器将每个组件变成常规的JavaScript类-只需导入并使用 new 实例化即可：

import App from './App.svelte';

const app = new App({

target: document.body,

props: {

// 我们稍候就会学到 props

answer: 42

}

});

然后，您可以按需使用 [组件API](https://link.zhihu.com/?target=https%3A//svelte.dev/docs%23Client-side_component_API) 与 app 进行交互。

## 八、调试

有时候，检查数据流经应用的数据是很有用的。

一种方案是在标记内使用 console.log(...)。但是，如果你想暂停执行，则可以使用 {@debug ...} 标签与要检查的值以逗号分隔的列表在一起使用：

App.svelte

<script>

let user = {

firstname: 'Ada',

lastname: 'Lovelace'

};

</script>

{@debug user}

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

如果你现在打开 devtools 并准备与<input>元素进行交互，则将在user修改其值时触发调试器。

## 总结

本章已简要介绍了 Svelte 的基本用法，Svelte 虽然与 React、Vue 一样，致力构建前端 Web 应用程序，Svelte 为了最优的性能及最简的代码输出，使用了编译器直接生成最终代码的方式。

Svelte 与 Vue 的 SFC 较为类似，即一个文件代表一个组件。

除非特殊的情况下（后面《模块》一章将详细说明），否则顶层 HTML 仅允许一个 script 标签（其他 script 可以放在子级元素之中，但是处理结果则不同），一个 style 标签（其他 style 标签可以放置在子元素之中，但是处理结果则不同）。

我们要添加一个类似双向绑定的变量，仅需如平常一样，声明一个普通变量即可。

而给元素复制相关的属性，则与 React、Vue 非常接近。

样式默认是局部作用域的，在组件内的样式，不会影响到组件以外。

最后，Svelte 的组件可以直接输出为原生的 JavaScript 对象，可以轻松将这个对象在 React、Vue 或其他地方导入，然后通过 new 实例化，就可以使用。

# Svelte 官方入门教程（2）—— 反应性

[上一章](https://zhuanlan.zhihu.com/p/367431533)已为诸君引出 Svelte 反应性的基础，它的声明方式与声明普通变量极为近似，对于初学者来说非常友好，本章将对反应性进行详尽讲述。

所谓反应性，对应React的状态（Status）或Vue的 data（实际上也是状态）。

Svelte 的组件状态，在用户看来，跟普通的 变量 无异：

<script>

**let** count **=** 0;

**let** username **=** 'anders'

**let** things **=** 'pens'

</script>

<div>{username} has {count} {thins} </div>

而组件状态对应于 Vue，则叫 data：

<script>

**new** Vue({

el**:** '#app',

data**:** {

count**:** 0,

username**:** 'anders',

things**:** 'pens',

},

})

</script>

<template>

<div>{username} has {count} {thins} </div>

</template>

React 则是 state：

**function** Counter () {

**const** [count, setCount] **=** useState(0)

**const** [username, setUsername] **=** useState('anders')

**const** [things, setThings] **=** useState('pens')

**return** (

**<**div**>**{username} has {count} {thins} **<**/div>

)

}

可见，对于组件状态而言，异曲同工，没有实质差别。

尤其 Vue3 最新的 <script setup> 语法，更是愈发接近。

对比三者而言，Svelte 没有学习新概念的负担。

框架/库总是热衷于创造这类术语隐喻，明显这样更显高端专业，可以确保的是，至少你能与朋友闲聊时，对 Hooks 和不可变性需要点时间来侃侃而谈，以便消遣时日。

遵照 React 的指引，应用 useState、useEffect 等 Hooks 之类的样板代码，使用者便有迹可循，只需按章办事就可完成代码的编写。

Svelte 的状态应用则来得直观简单、干脆利落，它就和我们声明一个普通变量没有差别，这些变量自动被认为是“状态”，当你改变它的值，所有引用的地方都会自动更新，你无需再重新学习一些术语，当然因此你也少了很多专业的谈资。

孰优孰劣，不置可否，纯属个人喜好的抉择。

**1、触发反应性的核心——赋值**

Svelte 的核心是功能强大的*响应式*系统，用于使 DOM 与你的应用程序状态保持同步，例如响应事件。

<script>

**let** count **=** 0;

**function** handleClick() {

*// 事件处理代码写在这儿*

}

</script>

<button>

Clicked {count} {count === 1 ? 'time' : 'times'}

</button>

为了演示它，我们首先需要连接一个事件处理程序。用以下内容替换 button 的开始标记：

<button on:click**=**{handleClick}>

在handleClick函数里，我们要编写代码更改count的值：

**function** handleClick() {

count **+=** 1;

}

Svelte 会针对这个**赋值**生成一些代码，该代码告诉 DOM 什么地方需要更新。

**2、声明**

当组件的状态更改时，Svelte 会自动更新 DOM。

通常，组件状态的某些部分需要从 *其他* 部分中计算得出（例如，fullname 系由 firstname 和 lastname 组合而成），并在每次更改时重新计算。

对于这些，我们有 *响应式声明*。他们看起来像这样：

**let** count **=** 0;

$**:** doubled **=** count **\*** 2;

完整代码：

<script>

**let** count **=** 0;

$**:** doubled **=** count **\*** 2;

**function** handleClick() {

count **+=** 1;

}

</script>

<button on:click**=**{handleClick}>

Clicked {count} {count === 1 ? 'time' : 'times'}

</button>

如果这看起来有点陌生，请不要担心。它是[有效的](https://link.zhihu.com/?target=https%3A//developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label)（只不过没那么常见的）JavaScript，Svelte 能理解代码的意思：“只要任何引用的值发生更改，都要重新运行此代码”。一旦习惯了，你就再也回不了头了。

现在让我们在标记中再加一个doubled：

<script>

**let** count **=** 0;

$**:** doubled **=** count **\*** 2;

**function** handleClick() {

count **+=** 1;

}

</script>

<button on:click**=**{handleClick}>

Clicked {count} {count === 1 ? 'time' : 'times'}

</button>

<p>{count} doubled is {doubled}</p>

当然，您可以只在标记中编写{count \* 2}，而不必使用响应式值。当您需要多次引用它们，或者您的值依赖于*其他*响应式的值时，响应式的值变得特别有用。

**3、语句**

我们不仅限于声明响应式的*值*，我们还可以响应式地运行任意*语句*。 例如，只要更改count的值，我们就可以在控制台输出它的值：

<script>

**let** count **=** 0;

$**:** doubled **=** count **\*** 2;

*// 只要更改count 的值，我们就可以在控制台输出它的值*

$**:** console.log(`the count is ${count}`);

**function** handleClick() {

count **+=** 1;

}

</script>

<button on:click**=**{handleClick}>

Clicked {count} {count === 1 ? 'time' : 'times'}

</button>

<p>{count} doubled is {doubled}</p>

你可以轻松将一组语句放在一个块里：

$**:** {

console.log(`the count is ${count}`);

alert(`I SAID THE COUNT IS ${count}`);

}

你甚至可以将$:放在一个if语句之前：

$: if (count >= 10) {

alert(`count is dangerously high!`);

count = 9;

}

**4、更新数组和对象**

Svelte 的响应性是由赋值触发的，因此使用诸如 push、splice 等数组方法不会引起自动更新。

正如下方示例，点击按钮后界面不会有任何变化：

<script>

**let** numbers **=** [1, 2, 3, 4];

**function** addNumber() {

numbers.push(numbers.length **+** 1);

}

$**:** sum **=** numbers.reduce((t, n) => t **+** n, 0);

</script>

<p>{numbers.join(' + ')} = {sum}</p>

<button on:click**=**{addNumber}>Add a number</button>

解决这个问题的其中一个办法是写一行多余的赋值给自身的表达式：

**function** addNumber() {

numbers.push(numbers.length **+** 1);

numbers **=** numbers;

}

但还有一个更常用的方案：

**function** addNumber() {

numbers **=** [...numbers, numbers.length **+** 1];

}

你可以使用类似的办法来替代 pop、shift、unshift 和 splice。

此外，赋值给数组和对象的 *属性*，例如 obj.foo += 1 或 array[i] = x，其实与直接赋值给自身的方式是一样的。

**function** addNumber() {

numbers[numbers.length] **=** numbers.length **+** 1;

}

这里有一个简单的规则：更新变量的名称，必须出现在赋值的左侧。例如这个...

**const** foo **=** obj.foo; foo.bar **=** 'baz';

...不会更新对 obj.foo.bar 的引用，除非后面加上 obj = obj。

例如下方的代码可以触发更新：

<script>

**let** obj **=** { a**:** { b**:** { c**:** 10 } } }

</script>

<button on:click**=**{ () => obj.a.b.c = 20 }>update</button>

<p>c = {obj.a.b.c}</p>

但是下方代码不会更新：

<script>

**let** obj **=** { a**:** { b**:** { c**:** 10 } } }

**function** handleClick() {

**let** b **=** obj.a.b

b.c **=** 20 *// 不会触发更新*

*// 除非有一个赋值操作，尝试取消下面这行注释：*

*// obj = obj;*

}

</script>

<button on:click**=**{ handleClick }>update</button>

<p>c = {obj.a.b.c}</p>

**总结**

赋值——是触发 Svelte 反应性的核心，Svelte 编译器根据赋值来确定是否需要更新相应的值。

除了常规的赋值语句外，$ 符号是另一种十分强大的反应性声明，它类似 Vue 的 computed 属性，不仅支持普通表达式，还可以是大括号括起的语句，甚至是 if 语句。

需要特别注意的是，数组的 push、pop、shift、unshift、splice 方式，不会触发反应性，可以通过两种方式来让 Svelte“认为”此处需要进行更新，第一种是使用数组的方法，然后使用看起来十分“呆板”的 **numbers = numbers**；第二种实际上也是赋值，不过它通过 ... 解构后重组了整个数组，然后赋值回了给 numbers 引用。

同样地，对象的子属性单独拿出来使用，也不会触发反应性。

# Svelte 官方入门教程（3）—— Props

在[上一章](https://zhuanlan.zhihu.com/p/368248774)，详尽探讨了 Svelte 的反应性(Reactivity)，它是当下各个前端框架必备的特性之一，今时今日大家都司空见惯，不足为奇。

而本章探讨的 Props，与React、Vue之间并无过多差异，甚至某些场景下显得更弱一些，譬如使用对象展开的属性 {...props}，对于 HTML 元素尤甚，主要表现为无法非常简单地添加一些动态的事件和动画相关的属性，要实现这一点，需要使用客户端组件API，以及动作（Action，在后续章节中将会详述）协力相辅。

**1、声明组件的 Props**

到目前为止，我们只处理组件内部的状态，也就是说，这些值只能在给定的组件中访问。

在任何实际的应用程序中，您都需要将数据从一个组件传递到其子组件。为此，我们需要为组件声明 *属性*，通常缩写为 'props'。

参看下方的子组件 Nested.svelte 及父组件 App.svelte

Nested.svelte

<script>

**let** answer;

</script>

<p>The answer is {answer}</p>

App.svlete

<script>

**import** Nested from './Nested.svelte';

</script>

<Nested answer**=**{42}/>

在 Svelte 中，我们使用 export 关键字来实现接受父组件传递过来的属性值。

编辑 Nested.svelte 组件：

<script>

**export** **let** answer;

</script>

正如 $: 一样，刚开始可能感觉有点怪异，export 在 JavaScript 模块中正常的使用方式也不是这样的！暂且尝试先接受这种书写方式，很快它会成为你的第二天性。

别忘记我们在名称和值相同时，可以使用速记的方式书写，节省一些键盘的敲击量，例如src={src}的速记为 **{src}**。

**2、属性默认值**

在 Nested.svelte 中我们可以很方便就可以为 props 指定默认值：

<script>

**export** **let** answer **=** 'a mystery';

</script>

如果我们现在编写第二个 Nested 组件且没有 answer 属性的话，它会用回默认值：

<Nested answer**=**{42}/>

<Nested/>

**3、展开 Props**

如果你的对象拥有很多属性，你可以 ‘展开’ 它们到组件中，无需逐个指定：

<Info {...pkg}/>

相反，如果需要传递所有属性给组件，包括没有用 export 声明的属性，可以直接使用 $$props 来赋值，不过一般不推荐直接使用它，因为 Svelte 较难优化这种操作，极小数情况下它很有用。

# Svelte 官方入门教程（4）—— 模板逻辑

前两章（[第2章](https://zhuanlan.zhihu.com/p/368248774)，[第3章](https://zhuanlan.zhihu.com/p/368436516)）所探讨的关于组件的状态（反应性）以及组件向子组件传递参数，这些内容的核心仍然聚焦在 JavaScript 之上。

除去脚本，组件的另一个核心点是 UI，用于描述 UI 的 HTML 是一种“静态”的语言，HTML 是无法表达 *逻辑* 的，比如条件和循环，但 Svelte 可以。Svelte 为其增加了一些逻辑支持，譬如判断、遍历等，以增强你对 UI 的表达能力。

这些额外附加的逻辑能力似曾相识，它十分接近一些以往非常流行的 [Handlebars](https://link.zhihu.com/?target=https%3A//handlebarsjs.com/) 或者 [Mustache](https://link.zhihu.com/?target=https%3A//github.com/janl/mustache.js) 模板语言，例如 **{{#if ...}}**，稍有区别的是，Svelte 使用单个大括号括起：**{#if ...}**。

Svelte 的编译器将会编译这些逻辑，通过将 HTML 编译成使用原生 JS 创建的形式，也即这些 HTML 最终将使用 createElement 等原生的 DOM API 来创建。

解释 HTML 模板时，Svelte 就已经明确哪些状态需要更新。

**1、If 块**

先来参看这些代码：

App.svelte

<script>

**let** user **=** { loggedIn**:** **false** };

**function** toggle() {

user.loggedIn **=** **!**user.loggedIn;

}

</script>

<button on:click**=**{toggle}>Log out</button>

<button on:click**=**{toggle}>Log in</button>

为了可以按条件渲染某些标记，我们将其放置在 if 块中：

App.svelte

<script>

**let** user **=** { loggedIn**:** **false** };

**function** toggle() {

user.loggedIn **=** **!**user.loggedIn;

}

</script>

{#if user.loggedIn}

<button on:click**=**{toggle}>Log out</button>

{/if}

{#if !user.loggedIn}

<button on:click**=**{toggle}>Log in</button>

{/if}

当然，这仅仅是为了演示，实际上我们知道两个 button 唯一的区别是“out”和“in”两个单词之差，完全可以使用一个 ? 号表达式来替代上述繁琐的 if。

**2、Else 块**

由于if user.loggedIn和if !user.loggedIn这两个条件是互斥的，因此可以使用else块来简化它：

App.svelte

<script>

**let** user **=** { loggedIn**:** **false** };

**function** toggle() {

user.loggedIn **=** **!**user.loggedIn;

}

</script>

{#if user.loggedIn}

<button on:click**=**{toggle}>Log out</button>

{:else}

<button on:click**=**{toggle}>Log in</button>

{/if}

# 字符代表 *块开始* 标记。/ 字符代表 *块结束* 标记。而 {:else} 中的 : 字符，则表示 *块连续* 标记。不用担心，至此你几乎已经学完 Svelte 为 HTML 添加的所有语法了。

**3、Else-If 块**

假设我们已有如下代码：

App.svelte

<script>

**let** x **=** 7;

</script>

{#if x > 10}

<p>{x} 大于 10</p>

{:else}

{#if 5 > x}

<p>{x} 小于 5</p>

{:else}

<p>{x} 在 5 和 10 之间</p>

{/if}

{/if}

可以使用else if将多个条件 “连接” 在一起：

App.svelte

<script>

**let** x **=** 7;

</script>

{#if x > 10}

<p>{x} 大于 10</p>

{:else if 5 > x}

<p>{x} 小于 5</p>

{:else}

<p>{x} 在 5 和 10 之间</p>

{/if}

**4、each 块**

如果你需要遍历数据列表，请使用each块，如下方代码，我们有一个数组需要遍历：

App.svelte

<script>

**let** cats **=** [

{ id**:** 'J---aiyznGQ', name**:** 'Keyboard Cat' },

{ id**:** 'z\_AbfPXTKms', name**:** 'Maru' },

{ id**:** 'OUtn3pvWmpg', name**:** 'Henri The Existential Cat' }

];

</script>

<h1>The Famous Cats of YouTube</h1>

<ul>

{#each cats as cat}

<li>

<a target**=**"\_blank" href**=**"https://www.youtube.com/watch?v={cat.id}">

{cat.name}

</a>

</li>

{/each}

</ul>

这个例子中的表达式 cats 的地方，事实上可以是任意数组或类数组的对象（只要它具有 length 属性）。你可以使用 each [...iterable] 来遍历迭代器。

你可以将当前的 *index* 索引作为第二个参数，如下所示：

App.svelte

<script>

**let** cats **=** [

{ id**:** 'J---aiyznGQ', name**:** 'Keyboard Cat' },

{ id**:** 'z\_AbfPXTKms', name**:** 'Maru' },

{ id**:** 'OUtn3pvWmpg', name**:** 'Henri The Existential Cat' }

];

</script>

<h1>The Famous Cats of YouTube</h1>

<ul>

{#each cats as cat, i}

<li>

<a target**=**"\_blank" href**=**"https://www.youtube.com/watch?v={cat.id}">

{i + 1}:{cat.name}

</a>

</li>

{/each}

</ul>

只要你愿意，还可以使用解构语法each cats as { id, name }, i，也即是用id、name来代替cat.id、cat.name。

**5、提供 key 的 each 块**

默认情况下，当你修改 each 块的值时，它将该块的 *结束* 位置添加和删除项，并更新所有已更改的值。不过那可能不是你想要的。

示例胜千言，不过这个示例还挺复杂的，分为两个组件：

Thing.svelte

<script>

*// 每当 prop 的值更改时，`current` 都会更新...*

**export** **let** current;

*// ...但 `initial` 初始化后就固定不变。*

**const** initial **=** current;

</script>

<p>

<span style**=**"background-color: {initial}">initial</span>

<span style**=**"background-color: {current}">current</span>

</p>

<style>

span {

**display**: **inline-block**;

**padding**: 0.2**em** 0.5**em**;

**margin**: 0 0.2**em** 0.2**em** 0;

**width**: 4**em**;

**text-align**: **center**;

**border-radius**: 0.2**em**;

**color**: **white**;

}

</style>

App.svlete

<script>

**import** Thing from './Thing.svelte';

**let** things **=** [

{ id**:** 1, color**:** 'darkblue' },

{ id**:** 2, color**:** 'indigo' },

{ id**:** 3, color**:** 'deeppink' },

{ id**:** 4, color**:** 'salmon' },

{ id**:** 5, color**:** 'gold' }

];

**function** handleClick() {

things **=** things.slice(1);

}

</script>

<button on:click**=**{handleClick}>Remove first thing</button>

{#each things as thing}

<Thing current**=**{thing.color}/>

{/each}

点击几次 “Remove first thing” 按钮，需要注意的是，它从最后位置删除 <Thing>，并为剩余的组件更新了 color。

假设我们期望删除第一个 <Thing> 组件，其余的不受影响。

为此，我们为 each 块指定一个唯一的 ID：

{#each things as thing (thing.id)}

<Thing current**=**{thing.color}/>

{/each}

这个 (thing.id) 告知 Svelte 如何计算变化的地方。

你可以使用任何对象作为 key，因为 Svelte 内部使用的是 Map 来实现的，换而言之，你甚至可以使用 (thing) 代替 (thing.id)。但是字符串或数字通常会更安全，因此这代表 ID 保持不变，且没有引用相等性问题，例如使用来自 API 服务传回来的新数据进行更新的时候。

**6、await 块**

大多数Web应用程序必须在某个时候处理异步数据，比如下方的代码：

App.svelte

<script>

async **function** getRandomNumber() {

**const** res **=** await fetch(`tutorial/random-number`);

**const** text **=** await res.text();

**if** (res.ok) {

**return** text;

} **else** {

**throw** **new** Error(text);

}

}

**let** promise **=** getRandomNumber();

**function** handleClick() {

promise **=** getRandomNumber();

}

</script>

<button on:click**=**{handleClick}>生成随机数字</button>

*<!-- 我们稍候将在此处使用 await 块 -->*

<p>{promise}</p>

Svelte使您可以轻松地在标记中直接 *await*[Promise](https://link.zhihu.com/?target=https%3A//developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises)的值：

App.svelte

<script>

async **function** getRandomNumber() {

**const** res **=** await fetch(`https://svelte.dev/tutorial/random-number`);

**const** text **=** await res.text();

**if** (res.ok) {

**return** text;

} **else** {

**throw** **new** Error(text);

}

}

**let** promise **=** getRandomNumber();

**function** handleClick() {

promise **=** getRandomNumber();

}

</script>

<button on:click**=**{handleClick}>生成随机数字</button>

{#await promise}

<p>{promise}</p>

{:then number}

*<!-- 返回的结果使用 :then 块连续标记，*

*后面提供接收Promise resolve 的变量名称*

*-->*

<p>The number is {number}</p>

{:catch error}

*<!-- 如果发生错误，在 :catch 子块中接收错误对象 -->*

<p style**=**"color: red">{error.message}</p>

{/await}

仅考虑最新的 promise，这说明你无需顾虑 race 情况。

如果你事前就明确了 promise 不会 reject，则可省略 catch 块。如果在 promise 被 resolve 之前不想显示任何内容，甚至也可以省略第一个块：

App.svelte

<script>

async **function** getRandomNumber() {

**const** res **=** await fetch(`https://svelte.dev/tutorial/random-number`);

**const** text **=** await res.text();

**if** (res.ok) {

**return** text;

} **else** {

**throw** **new** Error(text);

}

}

**let** promise **=** getRandomNumber();

**function** handleClick() {

promise **=** getRandomNumber();

}

</script>

<button on:click**=**{handleClick}>生成随机数字</button>

{#await promise then number}

<p>The number is {number}</p>

{/await}

**7、key 块**

现在我们有一个需求是展示一个数字，点击按钮每次加 2：

App.svelte

<script>

**let** num **=** 0

</script>

<p>{num}</p>

<button on:click**=**{ () => num += 2 }> +2 </button>

如果我们希望数字变化时使用“淡入”的效果，在不使用 Svelte 内置的 transition 的情况下（动画相关的内容在后续章节有详尽的讲述），可能需要一个定时器配合 opacity 样式规则来做：

App.svelte

<script>

**let** num **=** 0

**let** opacity **=** 1

**let** duration **=** 300

**function** plus2() {

opacity **=** 0

setTimeout(() => {

opacity **=** 1

num **+=** 2

}, duration)

}

</script>

<p style**=**"transition: {duration}ms; opacity: {opacity};">{num}</p>

<button on:click**=**{ plus2 }> +2 </button>

这显得繁琐了些，我们使用内置的 fade：

App.svelte

<script>

**import** { fade } from 'svelte/transition'

**let** num **=** 0

</script>

<p in:fade>{num}</p>

<button on:click**=**{ () => num += 2 }> +2 </button>

因为自带的 fade 淡入效果需要元素创建时才会展现，如你所见上述代码点击 +2 按钮并没有任何“动静”。

如何能够让 Svelte 知道：**num 这个变量的值被修改（每次 +2，状态变化） 后，p 元素需要强制销毁，并重新渲染？**

这就是 key 块的用途，事实上它的作用与第5节《提供 key 的 each 块》的作用是一样的，所不同的是一个应用在可迭代的对象，而 key 可以应用在任意状态。

现在我们通过加入 key 块轻松解决这个问题，现在它满足需求了：

App.svelte

<script>

**import** { fade } from 'svelte/transition'

**let** num **=** 0

</script>

{#key num}

<p in:fade>{num}</p>

{/key}

<button on:click**=**{ () => num += 2 }> +2 </button>

只要 num 的值改变了，key 块内的元素都会被**销毁**且**重新渲染**。

**总结**

本章已涵盖了模板逻辑的全部内容，长篇累牍。

用得较多的通常会是 #each 块，包括使用了 key 的 each 块，使用 key 目的是计算出哪些是没有变化的元素，这样可以不需要重新创建它，以便减少 DOM 渲染的工作量。

与 React、Vue 的 key 作用没有什么不同，不过第7节介绍的 key 块，则显得特别，它可以应用到非 each 块的元素中。

Svelte 的模板明显有较浓厚的 Handlebars 身影，初看颇为熟悉。

Svelte 将 Handlerbars 模板的双重大括号括起表达式，改为单个大括号括起值得赞赏，这更显简洁。

不过你完全有可能一开始不太适应这种模板语言，甚至是 —— 别扭。

例如那个容易使人不爽的 {:else}、{:then} 和 {:catch} 之类的丑东西（连续指示符），这种不愉快可能就是来自 else 等前面那个莫名其妙的冒号，似乎是一种多余的存在；此外，结尾的 {/if} 也是值得吐槽的地方，它也属于多余。

虽然近似于 HTML 的关闭标记，然而是否有必要这么做？

说到简洁，可能 Razor 的模板语法会比类 Handlebars 模板来得更简单直接，如下所示：

@**if**(user.loggedIn) {

**<**button on**:**click**=**{toggle}**>**Log out**<**/button>

} **else** {

**<**button on**:**click**=**{toggle}**>**Log **in<**/button>

}

是否更为简洁易懂？

起码我不用写了 {#if} 记得 {/if}，或者 {#each} 和 {/each}，关闭统统是这个简洁的 }。

# Svelte 官方入门教程（5）—— 事件

在[上一章](https://zhuanlan.zhihu.com/p/368665162)的内容中，我们在按钮上添加了一个 on:click 事件：

<button on:click**=**{toggle}>Log out</button>

本章我们专门探讨 Svelte 事件处理。

## 1、DOM 事件

我们来写一个简单的程序，界面上展示鼠标当前所在的坐标：

<script>

**let** m **=** { x**:** 0, y**:** 0 };

**function** handleMousemove(event) {

m.x **=** event.clientX;

m.y **=** event.clientY;

}

</script>

<style>

div { **width**: 100**%**; **height**: 100**%**; }

</style>

<div>

The mouse position is {m.x} x {m.y}

</div>

我们此前已经初步了解到，要在元素上监听任意事件，我们可以使用 on: 指令：

<div on:mousemove**=**{handleMousemove}>

The mouse position is {m.x} x {m.y}

</div>

## 2、内联事件

你也可以用内联的方式声明事件处理：

<div on:mousemove**=**"{e => m = { x: e.clientX, y: e.clientY }}">

The mouse position is {m.x} x {m.y}

</div>

属性值上的双引号其实是可选的，只是某些环境下有助于语法突出显示。

在一些框架中，出于性能原因，尤其是在循环内部，可能会建议你避免使用内联事件处理。不过这建议不适用 Svelte，无论你选择哪种形式，编译器都将始终会做正确的事情。

## 3、事件修饰符

DOM 事件处理程序允许使用*修饰符*来变更行为。例如，对某个处理程序使用once修饰符后，这个处理程序只会运行一次：

<script>

**function** handleClick() {

alert('no more alerts')

}

</script>

<button on:click|once**=**{handleClick}>Click me</button>

事件修饰符完整的列表：

* preventDefault：在运行事件处理程序之前先调用 event.preventDefault() ，这对客户端在表单处理时就很有用。
* stopPropagation：调用 event.stopPropagation() 停止事件冒泡，防止事件传播到下一个元素。
* passive：改进触摸/滚轮事件的滚动性能（Svelte 默认会自动在安全的地方添加它）。
* nonpassive：显式设置 passive: false。
* capture：在 *捕获* 阶段就触发处理程序，而非 *冒泡* 阶段（[参阅 MDN 文档](https://link.zhihu.com/?target=https%3A//developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events%23Event_bubbling_and_capture)）
* once：在运行一次事件处理后，立即将其移除。
* self：仅当 event.target 是元素本身时才触发事件处理程序。

修饰符支持链式使用，例如：on:click|once|capture={...}。

## 4、组件事件

组件也可以发送（dispath）事件。为此，需要创建一个事件分发器（dispatcher）。

为了展示组件事件，需要先写一个组件，假设是 Inner.svelte：

Inner.svelte

<script>

*// setup code goes here*

**function** sayHello() {

}

</script>

<button on:click**=**{sayHello}>Click to say hello</button>

我们在 App.svelte 引入这个 Inner.svelte 组件：

App.svelte

<script>

**import** Inner from './Inner.svelte';

**function** handleMessage(event) {

alert(event.detail.text);

}

</script>

<Inner on:message**=**{handleMessage}/>

你可以看到，在 App.svelte 中绑定了一个 on:message 自定义的事件名给 Inner，这说明 Inner 组件需要能够发送这个事件，handleMessage 才有可能被调用。

如何在 Inner 中对外发送一个自定义的事件？—— 这就需要用到事件分发器 EventDispatcher。

尝试修改一下 Inner.svelte:

Inner.svelte

<script>

**import** { createEventDispatcher } from 'svelte';

**const** dispatch **=** createEventDispatcher();

**function** sayHello() {

dispatch('message', {

text**:** 'Hello!'

});

}

</script>

<button on:click**=**{sayHello}>Click to say hello</button>

createEventDispatcher 必须在组件实例化时就调用它来创建分发器，你无法在组件一些特殊位置中创建（例如利用 setTimeout 延迟几秒之后创建）。dispatch 将会关联到组件实例中。

注意：由于 on:message 指令，使得 App 组件得以监听由 Inner 组件发出的消息。

这个指令是一个以 on: 作为前缀的属性，后面跟着我们正在分发的事件名称（在当前例子叫 message）。

不过就算没有这个属性，消息仍然会被分发，只是应用不会有任何响应而已。你可以尝试删除 on:message，然后再点击按钮。

你也可以尝试修改事件的名称为其他任意名称。例如，在 Inner.svelte 中将 dispatch('message') 改为 dispatch('myevent')，然后在 App.svelte 中将属性 on:message 改为 on:myevent。

## 5、事件转发

与DOM事件不同，组件事件不会 *冒泡*。 如果要在某个深层嵌套的组件上监听事件，则中间组件必须 *转发* 该事件。

在这种情况下，我们具有与上一节相同的 App.svelte 和 Inner.svelte，不过现在还需创建一个 Outer.svelte，并且在 Outer.svelte 组件中添加 <Inner/>。

Outer.svelte

<script>

**import** Inner from './Inner.svelte';

</script>

<Inner/>

解决这个问题的其中一种方案，是在 Outer.svelte 添加 createEventDispatcher，然后监听 message 事件，并为其创建处理程序：

Outer.svelte

<script>

**import** Inner from './Inner.svelte';

**import** { createEventDispatcher } from 'svelte';

**const** dispatch **=** createEventDispatcher();

**function** forward(event) {

dispatch('message', event.detail);

}

</script>

<Inner on:message**=**{forward}/>

不过如果层次过深，这么一层一层地转发枯燥乏味，Svelte 有一种等效的速写方式：on:message事件指令不写其值，就代表“转发所有message事件”，Svelte 自动生成与上方相同的代码替你包圆了：

<script>

**import** Inner from './Inner.svelte';

</script>

<Inner on:message/>

## 6、DOM 事件转发

上一节学到的组件自定义事件转发，同样适用于 DOM。

下方包装了一个 CustomButton.svelte 闪亮按钮组件：

CustomButton.svelte

<style>

button {

**height**: 4**rem**;

**width**: 8**rem**;

**background-color**: #aaa;

**border-color**: #f1c40f;

**color**: #f1c40f;

**font-size**: 1.25**rem**;

**background-image**: linear-gradient(45**deg**, #f1c40f 50**%**, **transparent** 50**%**);

**background-position**: 100**%**;

**background-size**: 400**%**;

**transition**: **background** 300**ms** **ease-**in-out;

}

button:hover {

**background-position**: 0;

**color**: #aaa;

}

</style>

<button>Click me</button>

App.svelte

<script>

**import** CustomButton from './CustomButton.svelte';

**function** handleClick() {

alert('clicked');

}

</script>

<CustomButton on:click**=**{handleClick}/>

作为一个纯粹的通用按钮的封装，CustomButton.svelte 不处理具体点击事件，而应该将事件发送给使用端去处理。

例如，当 <CustomButton> 的点击的事件，我们期望可以收到通知，为此，只需转发 CustomButton.svelte 中的 <button> 元素的 click 事件即可。

CustomButton.svelte

......

<button on:click>Click me</button>

如何响应这个点击事件，交由 App.svelte 接收到 click 事件后自行处理。

## 总结

本章探讨了 Svelte 事件的全部内容，最简单也是最常用的 DOM 事件绑定。

通过使用 on: 指令，例如 on:click，你可以直接内联一个函数的形式 on:click={ () => ... }，又或者将函数提到外部的形式 on:click={ handler }。两者的结果没有什么区别。

它同时也支持绑定组件的自定义事件，组件需要借助 createEventDispatcher() 方法将事件转发到使用端。

如果你正在编写一个 Svelte 组件库，那么事件转发是组件库创作者的必备工具箱，组件库里的组件一般不处理事件响应，而是应该将这个响应移交使用端处理。

不得不说，对比起 Vue 的写法，Vue 支持 @ 符号声明事件更为简洁一些，例如：

<button @click>Click me</button>

# Svelte 官方入门教程（6）—— 绑定

前面的章节（第[1](https://zhuanlan.zhihu.com/p/367431533)，[2](https://zhuanlan.zhihu.com/p/368248774)，[3](https://zhuanlan.zhihu.com/p/368436516)，[4](https://zhuanlan.zhihu.com/p/368665162)，[5](https://zhuanlan.zhihu.com/p/369228329)章）还没有涉及到双向绑定，本章所探讨的，主要是对表单元素（input、textarea等）如何与变量双向绑定。

## 1、文本输入框 input[type=text]

通常，Svelte 中的数据流是 *自上而下*，父组件可以在子组件上设置 props，而组件可以在元素上设置属性，但反之则不行。

App.svelte

<script>

**let** name **=** 'world';

</script>

<input value**=**{name}>

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

某些情况亟需打破这种规则。以组件中的 <input> 元素为例，我们 *可以* 为其添加 on:input 事件处理程序，在该程序中将 name 的值设置为 event.target.value，不过这样写法有点拖沓累赘，以后在表单元素的情况愈加苦不堪言。

好在，我们可以使用 bind:value 指令：

<input bind:value**=**{name}>

这意味着更改name的值时会更新 input 的值，反之亦然（双向绑定）。

## 2、数字或范围输入框 input[type=number|range]

App.svelte

<script>

**let** a **=** 1;

**let** b **=** 2;

</script>

<label>

<input type**=**number value**=**{a} min**=**0 max**=**10>

<input type**=**range value**=**{a} min**=**0 max**=**10>

</label>

<label>

<input type**=**number value**=**{b} min**=**0 max**=**10>

<input type**=**range value**=**{b} min**=**0 max**=**10>

</label>

<p>{a} + {b} = {a + b}</p>

在 DOM 中的值都是字符串。不能很好地帮助你处理数字输入（type="number" 和 type="range"），这需要你记得在使用 input.value 之前先将其强制转换一下。

使用 bind:value，Svelte 会为你排忧解难：

<input type**=**number bind:value**=**{a} min**=**0 max**=**10>

<input type**=**range bind:value**=**{a} min**=**0 max**=**10>

## 3、复选框 input[type=checkbox]

App.svelte

<script>

**let** yes **=** **false**;

</script>

<label>

<input type**=**checkbox checked**=**{yes}>

Yes! Send me regular email spam

</label>

{#if yes}

<p>Thank you. We will bombard your inbox and sell your personal details.</p>

{:else}

<p>You must opt in to continue. If you're not paying, you're the product.</p>

{/if}

<button disabled**=**{!yes}>Subscribe</button>

复选框可用于状态之间的切换，但不是绑定到input.value，而是input.checked：

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

## 4、输入框组 radio & checkbox

当你有多个输入框需要关联到同一个值的时候，则可以将bind:group与value属性一起使用。同一组的单选框（Radio）之间是互斥的；而同一组复选框中被选中的值会汇集到一个数组中。

App.svelte

<script>

**let** scoops **=** 1;

**let** flavours **=** ['Mint choc chip'];

**function** join(flavours) {

**if** (flavours.length **===** 1) **return** flavours[0];

**return** `${flavours.slice(0, **-**1).join(', ')} and ${flavours[flavours.length **-** 1]}`;

}

</script>

<h2>Size</h2>

<label><input type**=**radio group**=**{scoops} value**=**{1}>One scoop</label>

<label><input type**=**radio group**=**{scoops} value**=**{2}>Two scoops</label>

<label><input type**=**radio group**=**{scoops} value**=**{3}>Three scoops</label>

<h2>Flavours</h2>

<label>

<input type**=**checkbox group**=**{flavours} value**=**"Cookies and cream">Cookies and cream

</label>

<label>

<input type**=**checkbox group**=**{flavours} value**=**"Mint choc chip">Mint choc chip

</label>

<label>

<input type**=**checkbox group**=**{flavours} value**=**"Raspberry ripple">Raspberry ripple

</label>

{#if flavours.length === 0}

<p>Please select at least one flavour</p>

{:else if flavours.length > scoops}

<p>Can't order more flavours than scoops!</p>

{:else}

<p>

You ordered {scoops} {scoops === 1 ? 'scoop' : 'scoops'} of {join(flavours)}

</p>

{/if}

给每个输入框都加上bind:group：

<input type**=**radio bind:group**=**{scoops} value**=**{1}>

在这个例子中，我们可以通过将复选框放在each块中以简化代码，首先，在<script>中新增一个menu变量...

**let** menu **=** [

'Cookies and cream',

'Mint choc chip',

'Raspberry ripple'

];

...然后，替换掉第二部分 Flavours 的代码：

<h2>Flavours</h2>

{#each menu as flavour}

<label>

<input type**=**checkbox bind:group**=**{flavours} value**=**{flavour}>

{flavour}

</label>

{/each}

现在，这个冰淇淋菜单可以轻松地向新的令人兴奋的方向扩展了。

## 5、多行纯文本 textarea

我们计划先做一个简易的“MarkDown 在线编辑器”，通过 textarea 输入 markdown 代码，然后直接渲染到页面进行预览：

App.svelte

<script>

**import** marked from 'marked';

**let** value **=** `Some words are \*italic\*, some are \*\*bold\*\*`;

</script>

<style>

textarea { **width**: 100**%**; **height**: 200**px**; }

</style>

<textarea value**=**{value}></textarea>

{@html marked(value)}

在 Svelte 中，<textarea>元素的做法类似于文本输入框，也是使用bind:value：

<textarea bind:value**=**{value}></textarea>

像下面这个例子属性名都一样，我们可以使用速写形式：

<textarea bind:value></textarea>

这种速写形式适用于所有绑定，而不仅仅是 textarea 标签。

## 6、选择框 select（单选）

常见的找回密码功能经常使用密码提示问题的方式，下方以“安全提示问题”为例，下拉选择框中选择安全提示问题，并给出正确答案。

App.svelte

<script>

**let** questions **=** [

{ id**:** 1, text**:** `Where did you go to school?` },

{ id**:** 2, text**:** `What is your mother's name?` },

{ id**:** 3, text**:** `What is another personal fact that an attacker could easily find with Google?` }

];

**let** selected;

**let** answer **=** '';

**function** handleSubmit() {

alert(`answered question ${selected.id} (${selected.text}) with "${answer}"`);

}

</script>

<style>

input { **display**: **block**; **width**: 500**px**; **max-width**: 100**%**; }

</style>

<h2>Insecurity questions</h2>

<form on:submit|preventDefault**=**{handleSubmit}>

<select value**=**{selected} on:change**=**"{() => answer = ''}">

{#each questions as question}

<option value**=**{question}>{question.text}</option>

{/each}

</select>

<input bind:value**=**{answer}>

<button disabled**=**{!answer} type**=**submit>Submit</button>

</form>

<p>selected question {selected ? selected.id : '[waiting...]'}</p>

bind:value 可以与 <select>元素一起使用。修改 select 元素：

<select bind:value**=**{selected} on:change**=**"{() => answer = ''}">

值得注意的是，<option> 的值是对象，而非字符串，Svelte 是支持这种用法的。

由于尚未初始化 selected 的值，因此绑定会将其自动设置为默认值（列表中的第一个），不过不可大意，因为在绑定初始化之前，selected 实际上仍然是未定义的，我们还是要加以判断，不能盲目地随意引用，比如 39 行中的 selected.id。

## 7、选择框 select（多选）

当某个选择框使用了multiple属性的时候，那么这种情况下，它的值需要绑定到一个数组而非一个简单的值。

我们再次使用第4节《输入框组 radio & checkbox》的冰淇淋的例子：

App.svelte

<script>

**let** scoops **=** 1;

**let** flavours **=** ['Mint choc chip'];

**let** menu **=** [

'Cookies and cream',

'Mint choc chip',

'Raspberry ripple'

];

**function** join(flavours) {

**if** (flavours.length **===** 1) **return** flavours[0];

**return** `${flavours.slice(0, **-**1).join(', ')} and ${flavours[flavours.length **-** 1]}`;

}

</script>

<h2>Size</h2>

<label>

<input type**=**radio bind:group**=**{scoops} value**=**{1}>One scoop

</label>

<label>

<input type**=**radio bind:group**=**{scoops} value**=**{2}>Two scoops

</label>

<label>

<input type**=**radio bind:group**=**{scoops} value**=**{3}>Three scoops

</label>

<h2>Flavours</h2>

{#each menu as flavour}

<label>

<input type**=**checkbox bind:group**=**{flavours} value**=**{flavour}>{flavour}

</label>

{/each}

{#if flavours.length === 0}

<p>Please select at least one flavour</p>

{:else if flavours.length > scoops}

<p>Can't order more flavours than scoops!</p>

{:else}

<p>

You ordered {scoops} {scoops === 1 ? 'scoop' : 'scoops'} of {join(flavours)}

</p>

{/if}

我们可以使用，我们用<select multiple>来代替复选框:

<h2>Flavours</h2>

<select multiple bind:value**=**{flavours}>

{#each menu as flavour}

<option value**=**{flavour}>{flavour}</option>

{/each}

</select>

## 8、可编辑属性 contenteditable

在本章第5节《多行纯文本 textarea》我们实现一个简易的 Markdown 编辑器。

下方我们尝试另一种简易的编辑器实现 —— HTML编辑器，通过给元素加上 contenteditalbe 属性，元素即可获得可编辑能力：

App.svelte

<script>

**let** html **=** '<p>Write some text!</p>';

</script>

<div contenteditable**=**"true"></div>

<pre>{html}</pre>

<style>

**[**contenteditable**]** {

**padding**: 0.5**em**;

**border**: 1**px** **solid** #eee;

**border-radius**: 4**px**;

}

</style>

具有contenteditable="true"属性的元素都支持绑定textContent和innerHTML：

<div

contenteditable**=**"true"

bind:innerHTML**=**{html}

></div>

通过 bind:innerHTML 的绑定，现在编辑 div 的内容，立即会更新 pre 元素中的 HTML。

## 9、在 each 块中绑定

我们先来实现一个 todo 列表的功能：

App.svelte

<script>

**let** todos **=** [

{ done**:** **false**, text**:** 'finish Svelte tutorial' },

{ done**:** **false**, text**:** 'build an app' },

{ done**:** **false**, text**:** 'world domination' }

];

**function** add() {

todos **=** todos.concat({ done**:** **false**, text**:** '' });

}

**function** clear() {

todos **=** todos.filter(t => **!**t.done);

}

$**:** remaining **=** todos.filter(t => **!**t.done).length;

</script>

<style>

.**done** { **opacity**: 0.4; }

</style>

<h1>Todos</h1>

{#each todos as todo}

<div class:done**=**{todo.done}>

<input type**=**checkbox checked**=**{todo.done}>

<input placeholder**=**"What needs to be done?" value**=**{todo.text}>

</div>

{/each}

<p>{remaining} remaining</p>

<button on:click**=**{add}>Add new</button>

<button on:click**=**{clear}>Clear completed</button>

在 Svelte 中，你甚至可以在each块中绑定属性。

<input type**=**checkbox bind:checked**=**{todo.done}>

<input placeholder**=**"What needs to be done?" bind:value**=**{todo.text}>

请注意，与 <input> 元素进行绑定将使数组发生变化。如果你偏好使用不可变数据，则应避免使用这类绑定，而应使用事件处理程序。

## 10、媒体相关元素 audio & video

<audio>和<video>元素具有一些可以绑定的属性。其中部分可以参看下方示例（代码有点长）：

App.svelte

<script>

*// 下方这些值都绑定到 video 的属性*

**let** time **=** 0;

**let** duration;

**let** paused **=** **true**;

**let** showControls **=** **true**;

**let** showControlsTimeout;

**function** handleMousemove(e) {

*// 使控件可见，2.5秒内无任何动作则淡出*

clearTimeout(showControlsTimeout);

showControlsTimeout **=** setTimeout(() => showControls **=** **false**, 2500);

showControls **=** **true**;

**if** (**!**(e.buttons **&** 1)) **return**; *// mouse not down*

**if** (**!**duration) **return**; *// video not loaded yet*

**const** { left, right } **=** **this**.getBoundingClientRect();

time **=** duration **\*** (e.clientX **-** left) **/** (right **-** left);

}

**function** handleMousedown(e) {

*// 我们不使用内置的click事件而是自行监听，因为它会在拖动后才触发*

**function** handleMouseup() {

**if** (paused)

e.target.play();

**else**

e.target.pause();

cancel();

}

**function** cancel() {

e.target.removeEventListener('mouseup', handleMouseup);

}

e.target.addEventListener('mouseup', handleMouseup);

setTimeout(cancel, 200);

}

**function** format(seconds) {

**if** (isNaN(seconds)) **return** '...';

**const** minutes **=** Math.floor(seconds **/** 60);

seconds **=** Math.floor(seconds **%** 60);

**if** (seconds **<** 10) seconds **=** '0' **+** seconds;

**return** `${minutes}:${seconds}`;

}

</script>

<style>

div { **position**: **relative**; }

.**controls** {

**position**: **absolute**;

**top**: 0;

**width**: 100**%**;

**transition**: **opacity** 1**s**;

}

.**info** {

**display**: **flex**;

**width**: 100**%**;

**justify-content**: **space-**between;

}

span {

**padding**: 0.2**em** 0.5**em**;

**color**: **white**;

**text-shadow**: 0 0 8**px** **black**;

**font-size**: 1.4**em**;

**opacity**: 0.7;

}

.**time** { **width**: 3**em**; }

.**time**:last-child { **text-align**: **right** }

progress {

**display**: **block**;

**width**: 100**%**;

**height**: 10**px**;

**-webkit-appearance**: **none**;

**appearance**: **none**;

}

progress::-webkit-progress-bar { **background-color**: rgba(0,0,0,0.2); }

progress::-webkit-progress-value { **background-color**: rgba(255,255,255,0.6); }

video { **width**: 100**%**; }

</style>

<h1>Caminandes: Llamigos</h1>

<p>From <a href**=**"https://cloud.blender.org/open-projects">Blender Open Projects</a>. CC-BY</p>

<div>

<video

poster**=**"https://sveltejs.github.io/assets/caminandes-llamigos.jpg"

src**=**"https://sveltejs.github.io/assets/caminandes-llamigos.mp4"

on:mousemove**=**{handleMousemove}

on:mousedown**=**{handleMousedown}>

<track kind**=**"captions">

</video>

<div class**=**"controls" style**=**"opacity: {duration && showControls ? 1 : 0}">

<progress value**=**"{(time / duration) || 0}"/>

<div class**=**"info">

<span class**=**"time">{format(time)}</span>

<span>click anywhere to {paused ? 'play' : 'pause'} / drag to seek</span>

<span class**=**"time">{format(duration)}</span>

</div>

</div>

</div>

找到 video 元素，添加currentTime={time}、duration和paused绑定：

<video

poster**=**"https://sveltejs.github.io/assets/caminandes-llamigos.jpg"

src**=**"https://sveltejs.github.io/assets/caminandes-llamigos.mp4"

on:mousemove**=**{handleMousemove}

on:mousedown**=**{handleMousedown}

bind:currentTime**=**{time}

bind:duration

bind:paused

></video>

bind:duration 等效于 bind:duration={duration}，bind:paused 同理。

现在，当你点击视频时，将会自动更新 time、duration 和 paused。这说明我们可以基于它们构建自定义的控件。

通常你应该监听 timeupdate 事件来跟踪 currentTime。但是这些事件较少触发，导致 UI 不稳定，Svelte 可以做得更好，它通过使用 requestAnimationFrame 来检查 currentTime。

<audio> 和 <video> 的完整可以 *只读* 绑定的属性如下6个...

* duration (只读) — 视频的总时长（以秒为单位）
* buffered (只读) — 一个 {start, end} 对象数组
* seekable (只读) — 同上
* played (只读) — 同上
* seeking (只读) — boolean
* ended (只读) — boolean

...还有5个支持 *双向* 绑定的属性：

* currentTime — 视频中的当前位置，以秒为单位
* playbackRate — 视频播放倍速，'正常速度' 为 1
* paused — 暂停
* volume — 音量，其值介乎 0 ~ 1 之间
* muted — 一个布尔值，为 true 则说明静音状态

Video 还支持绑定只读的 videoWidth 和 videoHeight 属性。

## 11、尺寸绑定

元素的宽高也支持绑定，我们下方的例子是通过 range 输入框的值来指定 span 字体的尺寸，随着字体尺寸的变化，其外部包裹的元素 div 的尺寸也会变化：

App.svelte

<script>

**let** w;

**let** h;

**let** size **=** 42;

**let** text **=** 'edit me';

</script>

<style>

input { **display**: **block**; }

div { **display**: **inline-block**; }

span { **word-break**: break-all; }

</style>

<input type**=**range bind:value**=**{size}>

<input bind:value**=**{text}>

<p>size: {w}px x {h}px</p>

<div>

<span style**=**"font-size: {size}px">{text}</span>

</div>

每个块级元素都可以绑定到clientWidth、clientHeight、offsetWidth以及offsetHeight：

<div bind:clientWidth**=**{w} bind:clientHeight**=**{h}>

<span style**=**"font-size: {size}px">{text}</span>

</div>

这些绑定都是只读的，修改 w 或 h 的值不会有任何效果。

使用[这类](https://link.zhihu.com/?target=http%3A//www.backalleycoder.com/2013/03/18/cross-browser-event-based-element-resize-detection/)测量技术，会涉及到一些开销，因此不建议在大量元素上使用。  
display: inline 的内联元素是不能测量的，没有子级的元素也不能（例如 <canvas>），这种情况，你需要测量的是包裹它的外层元素。

## 12、绑定 this

绑定 this 与 Vue 的 ref 功能相似，都是为了取得当前组件的实际 DOM 对象，例如 canvas：

App.svelte

<script>

**import** { onMount } from 'svelte';

**let** canvas;

onMount(() => {

**const** ctx **=** canvas.getContext('2d');

**let** frame **=** requestAnimationFrame(loop);

**function** loop(t) {

frame **=** requestAnimationFrame(loop);

**const** imageData **=** ctx.getImageData(0, 0, canvas.width, canvas.height);

**for** (**let** p **=** 0; p **<** imageData.data.length; p **+=** 4) {

**const** i **=** p **/** 4;

**const** x **=** i **%** canvas.width;

**const** y **=** i **/** canvas.height **>>>** 0;

**const** r **=** 64 **+** (128 **\*** x **/** canvas.width) **+** (64 **\*** Math.sin(t **/** 1000));

**const** g **=** 64 **+** (128 **\*** y **/** canvas.height) **+** (64 **\*** Math.cos(t **/** 1000));

**const** b **=** 128;

imageData.data[p **+** 0] **=** r;

imageData.data[p **+** 1] **=** g;

imageData.data[p **+** 2] **=** b;

imageData.data[p **+** 3] **=** 255;

}

ctx.putImageData(imageData, 0, 0);

}

**return** () => { cancelAnimationFrame(frame); };

});

</script>

<style>

canvas {

**width**: 100**%**;

**height**: 100**%**;

**background-color**: #666;

**-webkit-mask**: url(svelte-logo-mask.svg) 50**%** 50**%** **no-repeat**;

**mask**: url(svelte-logo-mask.svg) 50**%** 50**%** **no-repeat**;

}

</style>

<canvas width**=**{32} height**=**{32}></canvas>

this只读绑定适用于任意元素或组件，允许你获取渲染的元素的引用。例如我们可以获取对<canvas>元素的引用：

<canvas

bind:this**=**{canvas}

width**=**{32}

height**=**{32}

></canvas>

请注意，在组件 mounted 之前，变量canvas的值还是undefined。因此我们需要将逻辑放在onMount这个生命周期函数（下一章将深入介绍）中。

## 13、组件绑定

我们尝试实现一个 Pin 码（类似密码）输入键盘组件，每输入一个字符，前面的字符全部使用星号或者实心圆代替，先是键盘组件部分：

Keypad.svelte

<script>

**import** { createEventDispatcher } from 'svelte';

**export** **let** value **=** '';

**const** dispatch **=** createEventDispatcher();

**const** select **=** num => () => value **+=** num;

**const** clear **=** () => value **=** '';

**const** submit **=** () => dispatch('submit');

</script>

<style>

.**keypad** {

**display**: **grid**;

**grid-template-columns**: **repeat**(3, 5**em**);

**grid-template-rows**: **repeat**(4, 3**em**);

**grid-gap**: 0.5**em**

}

button { **margin**: 0 }

</style>

<div class**=**"keypad">

<button on:click**=**{select(1)}>1</button>

<button on:click**=**{select(2)}>2</button>

<button on:click**=**{select(3)}>3</button>

<button on:click**=**{select(4)}>4</button>

<button on:click**=**{select(5)}>5</button>

<button on:click**=**{select(6)}>6</button>

<button on:click**=**{select(7)}>7</button>

<button on:click**=**{select(8)}>8</button>

<button on:click**=**{select(9)}>9</button>

<button disabled**=**{!value} on:click**=**{clear}>clear</button>

<button on:click**=**{select(0)}>0</button>

<button disabled**=**{!value} on:click**=**{submit}>submit</button>

</div>

App.svelte

<script>

**import** Keypad from './Keypad.svelte';

**let** pin;

$**:** view **=** pin **?** pin.replace(/\d(?!$)/g, '•') **:** 'enter your pin';

**function** handleSubmit() {

alert(`submitted ${pin}`);

}

</script>

<h1 style**=**"color: {pin ? '#333' : '#ccc'}">{view}</h1>

<Keypad on:submit**=**{handleSubmit}/>

就像可以绑定 DOM 元素的属性一样，你也可以绑定组件的 props。例如，我们可以绑定<Keypad>组件的value属性，就好像它是一个表单元素一样对待：

<Keypad bind:value**=**{pin} on:submit**=**{handleSubmit}/>

绑定之后，当用户与 Keypad 进行交互时，父组件的 pin 值会自动更新。

使用组件绑定需要谨慎小心。如果你的 App 中数据过多，尤其是在没有 “单一数据源” 的情况下，可能难以跟踪 App 相关的数据流。

## 总结

本章的内容十分长，一匹布那么长……

详尽介绍了与 DOM 元素和自定义组件之间的绑定操作，其中最常用的莫过于表单元素的绑定，如 input 家族就有 text、number、range、radio、checkbox，此外还有 选择框 select、多行文本 textarea，甚至是 contenteditable 元素都支持绑定其 textContent 和 innerHTML。

媒体相关的绑定虽然也有强大的绑定支持，毕竟还是小众。

Svelte 当然也支持在 each 块中对上述元素进行绑定，这倒是常见的需求。

难得的是跟尺寸相关的部分属性也纳入了绑定的支持，例如宽高。

对比尺寸的绑定，this 绑定更为实用一些，它类似 Vue 的 ref，绑定实际渲染的 DOM 对象。对于类似 canvas 这类组件尤其是不可或缺。

Svelte 也支持自定义组件的属性的绑定，这是相当不错的特性，在组件中修改了值，直接影响外部提供的状态，对比 React 的属性不可变性，这种更显实用省事。

一如既往需要吐槽一下这个绑定语法，如果能参考一下 Vue，直接用冒号代表属性绑定，那就最好不过了。

# Svelte 官方入门教程（7）—— 生命周期

如果你看完了[上一章](https://zhuanlan.zhihu.com/p/369240017)《绑定》的内容，那么看来你是 Svelte 真爱粉。

组件的生命周期函数不是新鲜玩意，各个主流框架都有，它是一种 Hook 钩子函数，所谓钩子，实际上是一种回调，它提前声明，并将在适当的时机被自动调用。

**1、onMount**

本节，我们以一个照片列表作为示例：

App.svelte

<script>

**let** photos **=** [];

</script>

<style>

.**photos** {

**width**: 100**%**;

**display**: **grid**;

**grid-template-columns**: **repeat**(5, 1fr);

**grid-gap**: 8**px**;

}

figure**,** img {

**width**: 100**%**;

**margin**: 0;

}

</style>

<h1>Photo album</h1>

<div class**=**"photos">

{#each photos as photo}

<figure>

<img src**=**{photo.thumbnailUrl} alt**=**{photo.title}>

<figcaption>{photo.title}</figcaption>

</figure>

{:else}

*<!-- 当 photos.length === 0 时展示下方内容 -->*

<p>loading...</p>

{/each}

</div>

每个组件从创建开始，到销毁结束这个过程，都有一个 *生命周期*。有一些功能可让您在生命周期的关键时刻运行代码。

onMount 再常见不过，它在组件首次渲染到 DOM 后立即触发。[前面的章节](https://link.zhihu.com/?target=http%3A//localhost%3A3000/tutorial/bind-this)中，我们已曾粗略了解到它了，还记得我们需要在渲染后与 <canvas> 元素进行交互。

我们将添加一个 onMount 事件处理程序，该程序通过网络加载一些数据：

<script>

**import** { onMount } from 'svelte';

**let** photos **=** [];

onMount(async () => {

**const** res **=** await fetch(`https://jsonplaceholder.typicode.com/photos?\_limit=20`);

photos **=** await res.json();

});

</script>

如果你使用服务端渲染（SSR），建议将 fetch 放入 onMount 中，而不是在 <script> 顶层位置。除了 onDestroy，其余生命周期函数不会在 SSR 运行期间执行，这说明我们可以将某些应该延迟加载的数据，放在组件加载到 DOM 之后获取。

生命周期函数只能在组件初始化时编写，以便将回调绑定到组件的实例，可以写在其他函数中调用，然而不可以将周期函数放在例如异步的 setTimeout 中。

如果 onMount 回调返回一个函数，该函数将在组件被销毁时被调用。

**2、onDestroy**

要在销毁组件时运行代码，请使用onDestroy。

App.svelte

<script>

**import** { onDestroy } from 'svelte';

**let** seconds **=** 0;

</script>

<p>

The page has been open for

{seconds} {seconds === 1 ? 'second' : 'seconds'}

</p>

例如，我们可以在组件初始化时添加setInterval函数，不再使用时对其进行清理。这样做可以防止内存泄漏。

<script>

**import** { onDestroy } from 'svelte';

**let** seconds **=** 0;

**const** interval **=** setInterval(() => seconds **+=** 1, 1000);

onDestroy(() => clearInterval(interval));

</script>

<p>

The page has been open for

{seconds} {seconds === 1 ? 'second' : 'seconds'}

</p>

虽然在组件初始化期间调用生命周期函数很重要，但是从何处调用它们都无关紧要。 因此，如果需要的话，我们可以将间隔逻辑抽象为utils.js中的一个辅助函数...

utils.js

**import** { onDestroy } from 'svelte';

**export** **function** onInterval(callback, milliseconds) {

**const** interval **=** setInterval(callback, milliseconds);

onDestroy(() => clearInterval(interval));

}

...并将其导入到我们的组件中：

<script>

**import** { onInterval } from './utils.js';

**let** seconds **=** 0;

onInterval(() => seconds **+=** 1, 1000);

</script>

<p>

The page has been open for

{seconds} {seconds === 1 ? 'second' : 'seconds'}

</p>

**3、beforeUpdate 和 afterUpdate**

beforeUpdate 函数会在 DOM 更新之前执行，而相应的 afterUpdate 则在数据同步到 DOM 之后执行。

它们一前一后，对于强制执行那些仅以状态驱动的方式却难以实现的事情非常有用，例如更新元素的滚动位置。

下方是一个[Eliza](https://link.zhihu.com/?target=https%3A//en.wikipedia.org/wiki/ELIZA)聊天机器人：

App.svelte

<script>

**import** Eliza from 'elizabot';

**import** { beforeUpdate, afterUpdate } from 'svelte';

**let** div;

beforeUpdate(() => {

*// 确认在 DOM 更新后，我们是否需要自动滚动*

});

afterUpdate(() => {

*// ...DOM 现在与数据同步*

});

**const** eliza **=** **new** Eliza();

**let** comments **=** [

{ author**:** 'eliza', text**:** eliza.getInitial() }

];

**function** handleKeydown(event) {

**if** (event.key **===** 'Enter') {

**const** text **=** event.target.value;

**if** (**!**text) **return**;

comments **=** comments.concat({author**:** 'user', text});

event.target.value **=** '';

**const** reply **=** eliza.transform(text);

setTimeout(() => {

comments **=** comments.concat({author**:** 'eliza', text**:** '...', placeholder**:** **true**});

setTimeout(() => {

comments **=** comments.filter(comment => **!**comment.placeholder).concat({

author**:** 'eliza',

text**:** reply

});

}, 500 **+** Math.random() **\*** 500);

}, 200 **+** Math.random() **\*** 200);

}

}

</script>

<style>

.**chat** {**display**: **flex**; **flex-direction**: **column**; **height**: 100**%**; **max-width**: 320**px**;}

.**scrollable** {

**flex**: 1 1 **auto**;

**border-top**: 1**px** **solid** #eee;

**margin**: 0 0 0.5**em** 0;

**overflow-y**: **auto**;

}

article {**margin**: 0.5**em** 0;}

.**user** {**text-align**: **right**;}

span {**padding**: 0.5**em** 1**em**; **display**: **inline-block**;}

.**eliza** span {**background-color**: #eee; **border-radius**: 1**em** 1**em** 1**em** 0;}

.**user** span {

**background-color**: #0074D9;

**color**: **white**;

**border-radius**: 1**em** 1**em** 0 1**em**;

**word-break**: break-all;

}

</style>

<div class**=**"chat">

<h1>Eliza</h1>

<div class**=**"scrollable" bind:this**=**{div}>

{#each comments as comment}

<article class**=**{comment.author}><span>{comment.text}</span></article>

{/each}

</div>

<input on:keydown**=**{handleKeydown}>

</div>

令人讨厌的地方，是你必须不断滚动聊天窗口，我们来解决这个问题。

**let** div;

**let** autoscroll;

beforeUpdate(() => {

autoscroll **=** div **&&** (div.offsetHeight **+** div.scrollTop) **>** (div.scrollHeight **-** 20);

});

afterUpdate(() => {

**if** (autoscroll) div.scrollTo(0, div.scrollHeight);

});

请注意，beforeUpdate首次运行是在组件 mounted 之前，因此我们需要在读取div的属性之前，检查div是否存在。

**4、tick**

tick 函数与其他生命周期函数有所不同，不一定是在首次初始化时调用，你可以随时调用它。它返回一个 Promise，并 resolve 任何挂起的状态（如果没有就立即 resolve）。

在 Svelte 中，当你更新组件状态时，它不会立即更新 DOM。相反，它会等到下一个 *微任务* 时更新，期间查看是否需要应用其他的更改，包括其他组件的更改。这样做可以减少一些无用功，让浏览器更有效地批量处理这些事情。

你可以在下方例子中观察到这种行为：

App.svelte

<script>

**let** text **=** `Select some text and hit the tab key to toggle uppercase`;

async **function** handleKeydown(event) {

**if** (event.key **!==** 'Tab') **return**;

event.preventDefault();

**const** { selectionStart, selectionEnd, value } **=** **this**;

**const** selection **=** value.slice(selectionStart, selectionEnd);

**const** replacement **=** /[a-z]/.test(selection)

**?** selection.toUpperCase()

**:** selection.toLowerCase();

text **=** (

value.slice(0, selectionStart) **+**

replacement **+**

value.slice(selectionEnd)

);

*// 这里没有任何作用，因为 DOM 尚未更新*

**this**.selectionStart **=** selectionStart;

**this**.selectionEnd **=** selectionEnd;

}

</script>

<style>

textarea {**width**: 100**%**; **height**: 200**px**;}

</style>

<textarea value**=**{text} on:keydown**=**{handleKeydown}></textarea>

在多行文本框中选中一些文本，并按下 Tab 键，这会导致<textarea>的值改变了，因此当前的选择会被清除，然后光标会让人讨厌地自动跳至文本末尾，我们可以通过 importtick来解决这个问题（将重新选中文本的代码，移到下一个更新周期才正式更新）...

**import** { tick } from 'svelte';

...然后在handleKeydown函数的末尾，设置this.selectionStart和this.selectionEnd之前运行它：

await tick();

**this**.selectionStart **=** selectionStart;

**this**.selectionEnd **=** selectionEnd;

**总结**

onMount/onDestroy 分别对应组件创建及销毁的两个过程。

onMount 是已经装载到 DOM 后触发，这一般是在调用完 appendChild 函数插入组件的 DOM 后，就会调用你的 onMount 钩子函数。

同样地，remove 操作执行后，通过 onDestroy 我们仍然有机会在这个时间点上执行一些代码，通常是做一些收尾或清除工作。

update 相关的两个钩子，一个在更新DOM之前 beforeUpdate，一个在更新DOM之后 afterUpdate，这种较为精细的控制用的不会太多。

关于 tick，与 Vue 的 tick 功能是一样的，强烈建议运行一下 tick 一节的代码，以体会 tick 的真正作用。

我们可以再看一个 tick 的简单的例子，将小写的书籍名称更新为大写：

<script>

**let** bookName **=** 'js starter'

**let** bookNode *// book 的 DOM 节点*

**function** upperBookName() {

*// 书名更新为大写*

bookName **=** bookName.toUpperCase()

*// 输出 bookNode 的 textContent，请注意是 bookNode*

console.log(bookNode.textContent)

}

</script>

<span bind:this**=**{bookNode}>书名：{bookName}</span>

<button on:click**=**{upperBookName}>大写</button>

bookNode 是 span 元素的 DOM 引用，在 upperBookName 函数中，我们故意向控制台输出 bookNode 元素的文本内容，目的是要验证：执行完 bookName = bookName.toUpperCase() 这一行后，DOM 元素有没有立即被更改。

从结果得知，bookName 虽然是大写了，但是 bookNode.textContent 去获取的 DOM 文本实际上还是小写。

这就是第4节《tick》中所述“**当你更新组件状态时，它不会立即更新 DOM。相反，它会等到下一个 *微任务* 时更新**”的缘故。

如果不用 tick 应该如何解决？

首先可以使用 setTimeout 创建一个宏观任务，它会在微观任务都执行后才开始执行，能达到延迟的效果：

**function** upperBookName() {

*// 书名更新为大写*

bookName **=** bookName.toUpperCase()

**let** timer **=** setTimeout(() => console.log(bookNode.textContent), 0)

clearTimeout(timer)

}

第二种办法是本章第3节所介绍的 afterUpdate：

<script>

**import** { afterUpdate } from 'svelte'

**let** bookName **=** 'js starter'

**let** bookNode *// book 的 DOM 节点*

*// 使用生命周期 afterUpdate 钩子*

afterUpdate(() => console.log(bookNode.textContent))

**function** upperBookName() {

bookName **=** bookName.toUpperCase()

}

</script>

<span bind:this**=**{bookNode}>书名：{bookName}</span>

<button on:click**=**{upperBookName}>大写</button>

显然也不失为是一个办法，不过无论 setTimeout 创建宏观任务的方式也好，还是用生命周期函数 afterUpdate 也好，总感觉不尽满意，tick 就是第3种最好的办法：

<script>

**import** { tick } from 'svelte'

**let** bookName **=** 'js starter'

**let** bookNode *// book 的 DOM 节点*

**function** upperBookName() {

bookName **=** bookName.toUpperCase()

await tick()

console.log(bookNode.textContent)

}

</script>

<span bind:this**=**{bookNode}>书名：{bookName}</span>

<button on:click**=**{upperBookName}>大写</button>

不妨斟酌一下：它既没有必要将微观任务改为宏观任务，也没有因 afterUpdate 而需要改变 console.log 这一句的位置。

# Svelte 官方入门教程（8）—— 状态管理

行文至[上一章](https://zhuanlan.zhihu.com/p/369399405)《生命周期》为止，Svelte 基础部分就已介绍完毕。

接下来是高级话题……

我们熟谙 React 的 Redux，或者 Vue 的 Vuex，它们都是以程序包的形式提供，对于复杂的状态管控，还可能用 mobx 或者 rxjs 之类的专业库。

前端开发者对于使用这类工具其实会感到迷茫无措，莫衷一是者大有人在。尤其在面试时无法对此陈举这方面的内容以自证能力之时，不免怨之怯怯。

这些状态管理库看起来举足轻重，不过使用起来又显得繁琐不堪（除了库作者感觉良好以外），我们耗费了时间去学习他们的理念，却又艰难地去应用它们，为的是什么？就没有简单易懂不绕弯子的状态管理工具了吗？

Svelte 是没有对应的状态库的，因为它内置了状态管理，它被称为 store。

当期望脱离组件的层级（父-子）关系且能够在任意位置都能访问某个状态（变量）时，状态管理仍然是非常有用的一个特性。

总的来说，Svelte 的状态管理更为简单直接 —— 我对这种简单的热爱毫不掩饰，它起码不会使用起来要绕晕脑袋（可能我的理解能力比较低），使得我的代码在别人眼里看来很“高级”。

**1、可写状态（Writable stores）**

并非所有的状态都属于在组件层次的结构内。某些时候，有些状态需要被多个毫不相干的组件或普通的 JavaScript 模块访问。

在 Svelte 中，我们通过*store*来实现。

本节的示例会比较复杂，它由一个主（父）组件加上三个子组件组成，之所以分成3个子组件，其目的是为了展示出 store 的特性。

Svelte 将状态划分为两种，一种可读可写，一种只读，都用可读可写（可以读取，也可以修改）虽然省事，不过允许或者说强制状态是只读的，可以防止状态被意外修改。

只读状态会在第3节详尽介绍，本节仅关注可写状态。

要创建一个可读且可写的状态十分简单，例如我们要创建一个数字型的可写状态 count，Svelte 提供 writable 函数来创建：

stores.js

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

**export** **const** count **=** writable(0);

我们可以将创建状态的代码，放到单独的文件 store.js 中，以便其他需要用到 count 状态的组件可以引入使用。

接下来我们计划要实现的功能是这样：

可写状态示例

非常简单的例子，点击 + 按钮，count 加 1，- 按钮，count 减 1，reset 按钮重置 count 为 0。

首先我们刻意将 +、-、reset 三个按钮分别写三个组件，这三个按钮在 App.svelte 中是并列出现的，不是父子关系：

Incrementer.svelte (递增)

<script>

**import** { count } from './stores.js';

**function** increment() {

*// TODO 递增 count 的值，需要用到 count 当前值，应该使用 update*

}

</script>

<button on:click**=**{increment}>+</button>

Decrementer.svelte (递减)

<script>

**import** { count } from './stores.js';

**function** decrement() {

*// TODO 递减 count 的值，需要用到 count 当前值，应该使用 update*

}

</script>

<button on:click**=**{decrement}>-</button>

Reset.svelte (重置)

<script>

**import** { count } from './stores.js';

**function** reset() {

*// TODO 重置 count 的值为 0（无需知道当前 count 的值，可以使用 set。*

}

</script>

<button on:click**=**{reset}>reset</button>

最后，我们将三个组件汇聚 App.svelte：

App.svelte

<script>

**import** { count } from './stores.js';

**import** Incrementer from './Incrementer.svelte';

**import** Decrementer from './Decrementer.svelte';

**import** Reset from './Reset.svelte';

**let** count\_value;

**const** unsubscribe **=** count.subscribe(value => {

count\_value **=** value;

});

</script>

<h1>count 当前的值是：{count\_value}</h1>

<Incrementer/>

<Decrementer/>

<Reset/>

所谓 store（也即状态），只不过是具有 subscribe 方法的对象，它允许当 store 的值改变时自动通知对此感兴趣的相关组件或程序。在 App.svelte 中，count 便是一个 store，我们在 count.subscribe 的回调中设置 count\_value 的值。

点击 stores.js 选项卡看看 count 的定义，可见它是一个 *writable* store（可写状态），这表示除了 subscribe 方法外，它还具有 set 和 update 方法。

现在转到 Incrementer.svelte 组件，我们可以关联 + 按钮：

**function** increment() {

count.update(n => n **+** 1);

}

现在再点击 + 按钮应该会更新 count 了。同理，在 Decrementer.svelte 中实现递减。

最后，在 Reset.svelte 里实现 reset：

**function** reset() {

count.set(0);

}

当我们需要知道 count 当前值的时候，应该使用 update，它会将当前值传递到回调函数供你使用；如果无需知道，则使用 set。

**2、自动订阅（Auto-subscriptions）**

上一个例子中，程序虽然可以这么写，不过存在一个不易察觉的错误：unsubscribe 函数没有机会被调用。如果该组件会被多次实例化和销毁，这将导致 *内存泄露*。

解决之道，应该使用 onDestroy 这个[生命周期](https://zhuanlan.zhihu.com/p/369399405) Hook。

<script>

**import** { onDestroy } from 'svelte';

**import** { count } from './stores.js';

**import** Incrementer from './Incrementer.svelte';

**import** Decrementer from './Decrementer.svelte';

**import** Reset from './Reset.svelte';

**let** count\_value;

**const** unsubscribe **=** count.subscribe(value => {

count\_value **=** value;

});

onDestroy(unsubscribe);

</script>

<h1>count 当前的值是：{count\_value}</h1>

不过事情又开始变得有点呆板重复了。

特别是当你的组件 subscribe 了很多的 store 的时候。Svelte 给出一个绝佳的替代方案，你可以在 store 名称前面加上$前缀来引用这个 store 的值：

<script>

**import** { count } from './stores.js';

**import** Incrementer from './Incrementer.svelte';

**import** Decrementer from './Decrementer.svelte';

**import** Reset from './Reset.svelte';

</script>

<h1>count 当前的值是：{$count}</h1>

自动订阅仅适用于在组件的顶层范围声明（或者导入的JS文件中）的 stroe 变量。

在标记中使用 $count 不会有任何限制，你也可以在 <script> 的任何位置使用它，例如在事件处理程序或者响应式声明中。

Svelte 假定所有以 $ 开头的任何标识符都表示引用某个 store 值，而 $ 实际上是一个保留字符，Svelte 会禁止你使用 $ 作为你声明的变量的前缀。

**3、只读状态（Readable stores）**

并非所有 store 允许所有人*可写*的。例如，你可能有一个 store 表示鼠标位置或者用户地理位置，允许 ‘外部’ 来修改这个值是没有意义的。对于这种情况，我们可以用*只读*store。

本节我们要制作一个数字钟，显示当前的时间，先看看主程序的代码：

App.svelte

<script>

**import** { time } from './stores.js';

**const** formatter **=** **new** Intl.DateTimeFormat('zh-CN', {

hour12**:** **false**,

hour**:** '2-digit',

minute**:** '2-digit',

second**:** '2-digit'

});

</script>

<h1>The time is {formatter.format($time)}</h1>

App.svelte 引用了状态文件 stores.js，因此需要编写这份文件：

stores.js

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

**export** **const** time **=** readable(**new** Date(), **function** start(set) {

*// 在此处实现*

**return** **function** stop() { };

});

readable 的第一个参数代表初始值，如果还没有具体的初始值，可以先设为 null 或 undefined。

第二个参数 start 是一个函数，该函数接受一个 set 回调用于设置值，并且返回一个 stop 函数。

当 store 被第一个订阅者读取时，就会调用 start 函数（然后在 start 函数中使用 set 提供一个最终的值；

最后一个订阅者退订时，调用 stop 函数。

stores.js

**export** **const** time **=** readable(**new** Date(), **function** start(set) {

**const** interval **=** setInterval(() => set(**new** Date()), 1000);

**return** **function** stop() { clearInterval(interval); };

});

**4、状态继承（Derived stores）**

你可以调用derived来创建一个新的 store，它将继承自某一个或多个*其他*的 store。

我们沿用上一节的数字钟例子，稍作添加：

App.svelte

<script>

**import** { time } from './stores.js';

**const** formatter **=** **new** Intl.DateTimeFormat('zh-CN', {

hour12**:** **false**,

hour**:** '2-digit',

minute**:** '2-digit',

second**:** '2-digit'

});

</script>

<h1>北京时间：{formatter.format($time)}</h1>

<p>页面已打开 {$elapsed} 秒</p>

stores.js

**import** { readable, derived } from 'svelte/store';

**export** **const** time **=** readable(**new** Date(), **function** start(set) {

**const** interval **=** setInterval(() => {set(**new** Date()), 1000);

**return** **function** stop() { clearInterval(interval); };

});

**const** start **=** **new** Date();

**export** **const** elapsed **=** derived(

time,

$time => {} *// 稍候在此处填写供订阅的值*

);

我们可以创建一个继承自time的 store 来记录页面打开的时间：

**export** **const** elapsed **=** derived(

time,

$time => Math.round(($time **-** start) **/** 1000)

);

可从多个 store 派生为一个 store，并显式地 set 一个值而不是返回它（这对于异步派生的值很有用处）。有关更多信息，请查阅[API 参考](https://link.zhihu.com/?target=https%3A//svelte.dev/docs%23derived)。

**5、自定义状态（Custom stores）**

我们第一节《可写状态》中的例子，创建了3个子组件来实现 +1、-1 及 reset 功能。

当然这只是为了演示而写，实际的开发中我们不用真的创建3个子组件，我们会用一个 store 来封装相关的业务：

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

**function** createCount() {

**const** { subscribe, set, update } **=** writable(0);

**return** {

subscribe,

increment**:** () => {},

decrement**:** () => {},

reset**:** () => {}

};

}

**export** **const** count **=** createCount();

App.svelte

<script>

**import** { count } from './stores.js';

</script>

<h1>count 的当前值是：{$count}</h1>

<button on:click**=**{count.increment}>+</button>

<button on:click**=**{count.decrement}>-</button>

<button on:click**=**{count.reset}>reset</button>

只要一个对象正确地实现了 subscribe 方法，它即是一个 store。除了之外，怎样都行。因此，使用特定领域的逻辑来创建 store 非常容易。

例如，我们前面例子中的 count store 可以将 increment、decrement 和 reset 方法包含进来，并避免暴露 set 和 update 两个方法：

**function** createCount() {

**const** { subscribe, set, update } **=** writable(0);

**return** {

subscribe,

increment**:** () => update(n => n **+** 1),

decrement**:** () => update(n => n **-** 1),

reset**:** () => set(0)

};

}

**6、状态绑定（Store bindings）**

我们假设有一个 store 叫 name，此外再创建一个 greeting 的 store 来继承 name，然后两者都导出：

stores.js

**import** { writable, derived } from 'svelte/store';

**export** **const** name **=** writable('world');

**export** **const** greeting **=** derived(

name,

$name => `Hello ${$name}!`

);

App.svelte

<script>

**import** { name, greeting } from './stores.js';

</script>

<h1>{$greeting}</h1>

<input value**=**{$name}>

如果 store 是可写的，即它具有 set 方法，则可以绑定到其值，就像绑定到本地组件的状态是一样的。

在这个例子中，我们有一个可写的 store name 和一个派生 store greeting，尝试修改 <input> 元素：

<input bind:value**=**{$name}>

现在修改 input 的 value 会自动更新 name 及其所有相关的依赖项。

我们还可以直接为组件内部的 store 值进行赋值。添加一个 <button> 元素：

<button on:click**=**"{() => $name += '!'}">

Add exclamation mark!

</button>

$name += '!' 赋值是等效于 name.set($name + '!') 的。

**7、与 RxJS 结合**

如前面所述，要获得 store 的值，是通过 store.subscribe 方法去订阅，store 产生的值会自动推送给订阅者，这点与 RxJS 如出一辙，实际上，你可以将 Svelte 的 store 看作是 RxJS 的 Observable。

这样做不是没有原因的，其一是为了更容易与类似 rxjs 这样的库相结合，共同发挥作用；其二是认同 RxJS 这套观察者-订阅的理念且值得借鉴。

下面我们尝试演示 RxJS 相结合的例子：

<script>

**import** { fromEvent } from 'rxjs'

**import** { map } from 'rxjs/operators'

**let** position **=** fromEvent(document, 'mousemove')

.pipe(map(e => `鼠标位置: ${e.clientX},${e.clientY}`))

</script>

{$position}

上述例子可见，由 RxJS 产生的 Observable，竟然可以无缝当成 store 来用。

**总结**

我们通常将状态写到单独的 JS 文件中，例如每个状态一份 JS 文件。

可写状态 writable 应该是我们最常用的状态，只读状态 readable 是在防止被意外修改的情况下使用。

要读取状态的值，使用状态的 subscribe 方法，之所以取这个名字，显然有意为之，例如它兼容 rxjs 的写法，可以轻松与 rxjs 结合使用。

当然，你绝不会放弃使用被称为自动订阅（第2节）的速写形式：使用 $ 符号，有了这个速写形式，可以精简很多状态管理相关的代码，简直爽翻。

自定义状态让你有机会封装一个状态，使其更贴近业务，以及使用起来更为便捷。

Svelte 的状态支持继承，目的是在于可以快速从一个已存在的状态中将其复用。

而状态同时也支持[绑定](https://zhuanlan.zhihu.com/p/369240017)。

# Svelte 官方入门教程（9）—— 动画

[上一章](https://zhuanlan.zhihu.com/p/369408950)的状态管理说完，接着我们讲动画效果……

动画是交互设计中重要的视觉元素，页面如果缺少了动画，会显得俨乎其然而变得生硬无趣，少了许多生机。

但是与游戏类程序不同，页面上的动画要谨慎克制，并非越是花俏夺目就越好，正如女人浓妆艳抹一般，切忌用力过猛，否则浮华不实，过犹不及。

动画讲求从容雅正，恰如其分，目的是用于为页面加添交互气氛，也就是稍微“增鲜提味”，不可喧宾夺主。

因此，页面上的交互动画主要都是一些偏线性的**过渡效果**，在操作与操作之间缓和过渡，以避免生硬冒失，唐突而出。

下文将效果划分为三个小节进行逐一介绍：运动效果（motion）、过渡效果（transition）和动画（animate）。

## 1、运动效果

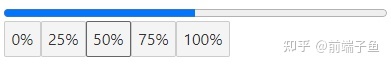
**1.1、补间动画（Tweened）**

设置值并监视 DOM 自动更新非常炫酷。那么还有更酷的吗？

更酷的就是为那些值加入 *缓动* 效果。Svelte 内置了这些工具可使用动画来凸显更改，以帮助你构建漂亮的用户界面。

首先介绍的是补间动画（tweened），tweened最常用于频繁变化的值。

我们的示例是一个进度条，进度条下方有数个按钮，通过点击按钮设置进度条的当前进度：

进度条示例

App.svelte

<script>

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

**const** progress **=** writable(0);

</script>

<style>

progress { **display**: **block**; **width**: 100**%**; }

</style>

<progress value**=**{$progress}></progress>

{#each [0, 0.25, 0.5, 0.75, 1] as p}

<button on:click**=**{() => $progress = p}>

{p\*100}%

</button>

{/each}

以上例子还没有加入动画效果，因此看起来较为生硬，让我们修改progressstore 为tweened：

<script>

**import** { tweened } from 'svelte/motion';

**const** progress **=** tweened(0);

</script>

单击按钮会使进度条在更新值的时候有动画效果，但仍然还是有点 Low，不太让人满意。为此我们需要添加一个 easing 缓动函数：

<script>

**import** { tweened } from 'svelte/motion';

**import** { cubicOut } from 'svelte/easing';

**const** progress **=** tweened(0, {

duration**:** 400,

easing**:** cubicOut

});

</script>

duration 我们使用 400 毫秒完成，动画时间在 300 ~ 400 毫秒最佳，多则拖沓冗长，少则浅淡无感。

svelte/easing 模块包含了 [Penner easing 缓动方程](https://link.zhihu.com/?target=https%3A//web.archive.org/web/20190805215728/http%3A//robertpenner.com/easing/)，你也可以使用自己的 p => t 函数，其中 p 和 t 都是 0 ~ 1 之间的值。

tweened 效果可用的全套选项如下：

* delay：补间延迟多少毫秒之后开始
* duration：补间效果的持续实践（以毫秒为单位），或者用一个 (from, to) => milliseconds 函数，可让你指定长实践补间缓动，以进行较大的值更改。
* easing：一个 p => t 函数
* interpolate：一个自定义的 (from, to) => t => value 函数，用于在任意值之间进行插值。默认情况下，Svelte 将在数字、日期以及形式近似的数组和对象之间进行插值（只要它们仅包含数字和日期，或其他有效的数组和对象）。如果要插值，比方说，颜色字符串或者变换矩阵，应该提供一个自定义插值器。

你还可以将这些选项作为第二个参数传递给 progress.set 和 progress.update，在这种情况下，它们将覆盖默认值。set 和 update 方法都返回一个Promise，当补间效果完成时，Promise 将被 resolve。

**1.2、弹簧动画（Spring）**

spring函数是tweened之外的另一个选择。

现在假设有一个圆，它跟随鼠标移动，移动过程中加入一些物理弹簧效果 spring，UI 实现的最终效果如下：

spring 演示UI

下面先来准备 UI，稍候再加入 spring：

App.svelte

<script>

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

**let** coords **=** writable({ x**:** 50, y**:** 50 }); *// 圆的位置*

**let** size **=** writable(10); *// 圆的大小*

</script>

<style>

svg { **width**: 100**%**; **height**: 100**%**; **margin**: -8**px**; }

circle { fill: #ff3e00 }

</style>

<div style**=**"position: absolute; right: 1em;">

<label>

<h3>刚度 ({coords.stiffness})</h3>

<input bind:value**=**{coords.stiffness} type**=**"range" min**=**"0" max**=**"1" step**=**"0.01">

</label>

<label>

<h3>阻尼 ({coords.damping})</h3>

<input bind:value**=**{coords.damping} type**=**"range" min**=**"0" max**=**"1" step**=**"0.01">

</label>

</div>

*<!-- 使用 SVG 绘制的画布 -->*

<svg

on:mousemove**=**"{e => $coords={ x: e.clientX, y: e.clientY }}"

on:mousedown**=**"{() => $size=30}"

on:mouseup**=**"{() => $size=10}"

>

<circle cx**=**{$coords.x} cy**=**{$coords.y} r**=**{$size}/>

</svg>

在这个例子中，我们有两个 store：一个代表圆的坐标，另一个代表圆的大小。让我们改用spring试试：

<script>

**import** { spring } from 'svelte/motion';

**let** coords **=** spring({ x**:** 50, y**:** 50 });

**let** size **=** spring(10);

</script>

两个 spring 的参数都有两个默认值，分别是表示刚度的stiffness和表示阻尼的damping，我们用这些值来控制 spring 的弹性，除了用默认值，我们也可以指定自己的初始值：

**let** coords **=** spring({ x**:** 50, y**:** 50 }, {

stiffness**:** 0.1,

damping**:** 0.25

});

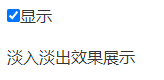
你尝试左右摇晃鼠标看看，然后尝试在滑块控件更改那些值，了解这些值是如何影响 spring 的弹性的。请注意，在 spring 运动期间你也可以调整这些值。

## 2、过渡效果

**2.1、transition 指令**

将进入元素或离开元素的过程如能优雅地使用过渡效果，可以大大增加用户界面的吸引力。Svelte 使用transition指令让这种操作轻而易举。

我们的示例也十分简单，让一行文本淡出淡入，通过一个 checkbox 来控制文本是否可视，UI 如下：

transition 指令演示

我们代码准备如下，fade 效果稍候再添加：

<script>

**let** visible **=** **true**;

</script>

<label><input type**=**"checkbox" bind:checked**=**{visible}>显示</label>

{#if visible}

<p>淡入淡出效果展示</p>

{/if}

首先，从svelte/transition导入fade函数...

<script>

**import** { fade } from 'svelte/transition';

**let** visible **=** **true**;

</script>

...然后将它添加到<p>元素：

<p transition:fade>淡入淡出效果展示</p>

**2.2、附加参数**

过渡效果的函数可以接受参数。上方例子中的效果用fly来代替fade过渡看看...

<script>

**import** { fly } from 'svelte/transition';

**let** visible **=** **true**;

</script>

...添加至<p>后，为其指定一些选项参数：

<p transition:fly**=**{{ y: 200, duration: 2000 }}>

飞入效果展示

</p>

请注意，过渡效果是*可逆*的，在例子中，如果在过渡效果进行期间，切换了复选框，它将从当前位置开始过渡，而不是起点或终点位置过渡。

**2.3、in 和 out**

除了用transition指令来过渡外，元素也可以使用一个in或一个out指令来过渡，或者两者齐用。

在fly旁边再导入fade...

**import** { fade, fly } from 'svelte/transition';

...让后将transition指令替换为各自单独的in和out指令：

<p in:fly**=**{{ y: 200, duration: 2000 }} out:fade>

飞入、淡出效果展示

</p>

在这种情况下，过渡效果则是 *不可逆*的了。

**2.4、自定义 CSS 过渡**

svelte/transition模块中已内置了一些过渡效果，其实创建自定义的过渡效果也很简单：

《自定义 CSS 过渡》演示

App.svelte

<script>

**import** { fade } from 'svelte/transition';

**let** visible **=** **true**;

**function** spin(node, { duration }) {

**return** { duration, css**:** t => `` };

}

</script>

<style>

.**centered** { **position**: **absolute**; **left**: 50**%**; **top**: 50**%**; **transform**: translate(-50**%**,-50**%**); }

span { **position**: **absolute**; **transform**: translate(-50**%**,-50**%**); **font-size**: 4**em**; }

</style>

<label><input type**=**"checkbox" bind:checked**=**{visible}>显示</label>

{#if visible}

<div class**=**"centered" in:spin**=**"{{duration: 8000}}" out:fade>

<span>transitions!</span>

</div>

{/if}

下方是fade过渡效果的源码：

**function** fade(node, { delay **=** 0, duration **=** 400 }) {

**const** o **=** **+**getComputedStyle(node).opacity;

**return** { delay, duration, css**:** t => `opacity: ${t **\*** o}` };

}

该函数接受两个参数，分别是应用过渡效果的节点，以及相关的参数，并返回一个过渡效果的对象，这个对象具有以下属性：

* delay：延迟指定毫秒后开始过渡
* duration：完成过渡的总时间（单位：毫秒）
* easing：一个 p => t easing 缓动函数（参看运动效果/补间动画这一节）
* css：一个 (t, u) => css 函数，其中 u === 1 - t
* tick：一个 (t, u) => {...} 函数，该函数会对元素节点有些影响

在刚进入过渡效果或效果已终止时，t 的值为 0；在已经进入过渡效果之后或从终止位置开始时，t 的值为 1。

大多数情况下，你应该返回 css 属性而 *不是* tick 属性，因为 CSS 动画会在主线程上运行，以尽可能避免做无用功。Svelte '模拟' 这个过渡效果并构建一个 CSS 动画，接着使其运行起来。

比方说，fade 过渡效果会生成类似以下 CSS 动画：

0**%** { **opacity**: 0 }

10**%** { **opacity**: 0.1 }

20**%** { **opacity**: 0.2 }

*/\* ... \*/*

100**%** { **opacity**: 1 }

不过，我们可以更具创意，让我们做出一些有真正酷炫的东西：

<script>

**import** { fade } from 'svelte/transition';

**import** { elasticOut } from 'svelte/easing';

**let** visible **=** **true**;

**function** spin(node, { duration }) {

**return** {

duration,

css**:** t => {

**const** eased **=** elasticOut(t);

**return** `

transform: scale(${eased}) rotate(${eased **\*** 1080}deg);

color: hsl(

${**~~**(t **\*** 360)},

${Math.min(100, 1000 **-** 1000 **\*** t)}%,

${Math.min(50, 500 **-** 500 **\*** t)}%

);`

}

};

}

</script>

不过请记住：强大的能力意味着要付出巨大的维护责任。

**2.5、自定义 JS 过渡**

通常，你应该尽可能使用 CSS 来做过渡效果，但是有些效果不用 JavaScript 则无法实现，例如打字机效果：

App.svelte

<script>

**let** visible **=** **false**;

**function** typewriter(node, { speed **=** 50 }) {

*// 在此处编写代码*

**return** {

};

}

</script>

<label><input type**=**"checkbox" bind:checked**=**{visible}>显示</label>

{#if visible}

<p in:typewriter>

人面不知何处去，桃花依旧笑春风。

</p>

{/if}

我们接着补全打字机的效果试试：

**function** typewriter(node, { speed **=** 50 }) {

**const** valid **=** (

node.childNodes.length **===** 1 **&&**

node.childNodes[0].nodeType **===** Node.TEXT\_NODE

);

**if** (**!**valid) {

**throw** **new** Error(`这个过渡效果仅适用于只有文本的元素。`);

}

**const** text **=** node.textContent;

**const** duration **=** text.length **\*** speed;

**return** {

duration,

tick**:** t => {

**const** i **=** **~~**(text.length **\*** t);

node.textContent **=** text.slice(0, i);

}

};

}

**2.6、过渡效果事件**

知道过渡效果是何时开始和结束在某些时候十分有用。你可以像监听其他 DOM 事件一样去监听到 Svelte 发送的过渡效果事件。

我们沿用本章 2.1节的飞入效果为基础：

App.svelte

<script>

**import** { fly } from 'svelte/transition';

**let** visible **=** **true**;

**let** status **=** '等待中...';

</script>

<p>状态：{status}</p>

<label><input type**=**"checkbox" bind:checked**=**{visible}>显示</label>

{#if visible}

<p transition:fly**=**{{ y: 200, duration: 2000 }}>

飞入飞出效果展示

</p>

{/if}

我们可以监听以下过渡效果的事件：

* introstart：进入效果开始
* introend：进入效果结束
* outrostart：退出效果开始
* outroend：退出效果结束

我们给 <p> 元素添加这些事件以观察其状态：

<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'}"

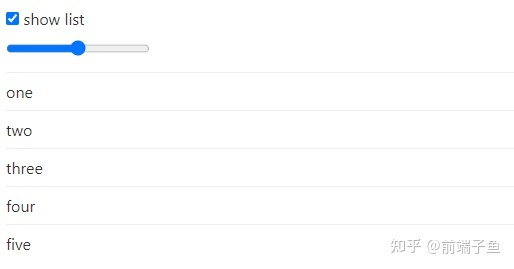
>

飞入飞出效果展示

</p>

**2.7、局部过渡（Local transitions）**

一般来说，添加或删除任何块级元素时，便会在该元素上执行过渡效果。请观察下方例子：

局部过渡效果演示

App.svelte

<script>

**import** { slide } from 'svelte/transition';

**let** showItems **=** **true**;

**let** i **=** 5;

**let** items **=** [

'one', 'two', 'three', 'four', 'five',

'six', 'seven', 'eight', 'nine', 'ten'

];

</script>

<style>

div { **padding**: 0.5**em** 0; **border-top**: 1**px** **solid** #eee; }

</style>

<label>

<input type**=**"checkbox" bind:checked**=**{showItems}>

show list

</label>

<label>

<input type**=**"range" bind:value**=**{i} max**=**10>

</label>

{#if showItems}

{#each items.slice(0, i) as item}

<div transition:slide>{item}</div>

{/each}

{/if}

在这个例子中，切换整个列表的可见性，会将过渡效果应用到各个列表中的元素。

有时候我们仅希望添加或删除元素中的某些项时（也就是说，当用户拖动例子中的滑块时），才开始执行过渡效果。

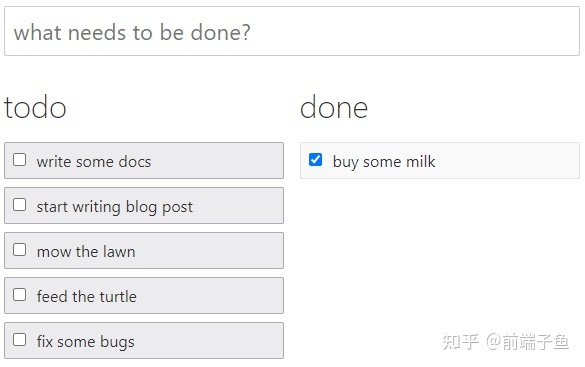
我们可以通过 *local* 过渡来实现，这个 local 指示过渡效果仅在添加或删除父元素时才会执行：

<div transition:slide|local>{item}</div>

**2.8、延迟过渡（Deferred transitions）**

Svelte 的 transition 过渡动画引擎有一个特别强大的功能是*延迟*过渡，以便可以在多个元素之间进行协调。

我们以一个待办事项为例，这个例子稍微有点大：

延迟过渡示例

App.svelte

<script>

**import** { quintOut } from 'svelte/easing';

**import** { crossfade } from 'svelte/transition';

**const** [send, receive] **=** crossfade({

duration**:** d => Math.sqrt(d **\*** 200),

fallback(node, params) {

**const** style **=** getComputedStyle(node);

**const** transform **=** style.transform **===** 'none' **?** '' **:** style.transform;

**return** {

duration**:** 600,

easing**:** quintOut,

css**:** t => `

transform: ${transform} scale(${t});

opacity: ${t}

`

};

}

});

**let** uid **=** 1;

**let** todos **=** [

{ id**:** uid**++**, done**:** **false**, description**:** 'write some docs' },

{ id**:** uid**++**, done**:** **false**, description**:** 'start writing blog post' },

{ id**:** uid**++**, done**:** **true**, description**:** 'buy some milk' },

{ id**:** uid**++**, done**:** **false**, description**:** 'mow the lawn' },

{ id**:** uid**++**, done**:** **false**, description**:** 'feed the turtle' },

{ id**:** uid**++**, done**:** **false**, description**:** 'fix some bugs' },

];

**function** add(input) {

**const** todo **=** { id**:** uid**++**, done**:** **false**, description**:** input.value };

todos **=** [todo, ...todos];

input.value **=** '';

}

**function** remove(todo) { todos **=** todos.filter(t => t **!==** todo); }

**function** mark(todo, done) {

todo.done **=** done;

remove(todo);

todos **=** todos.concat(todo);

}

</script>

<div class**=**'board'>

<input

placeholder**=**"what needs to be done?"

on:keydown**=**{e => e.key === 'Enter' && add(e.target)}

>

<div class**=**'left'>

<h2>todo</h2>

{#each todos.filter(t => !t.done) as todo (todo.id)}

<label>

<input type**=**checkbox on:change**=**{() => mark(todo, true)}>

{todo.description}

<button on:click**=**"{() => remove(todo)}">remove</button>

</label>

{/each}

</div>

<div class**=**'right'>

<h2>done</h2>

{#each todos.filter(t => t.done) as todo (todo.id)}

<label class**=**"done">

<input type**=**checkbox checked on:change**=**{() => mark(todo, false)}>

{todo.description}

<button on:click**=**"{() => remove(todo)}">remove</button>

</label>

{/each}

</div>

</div>

<style>

.**board** {

**display**: **grid**;

**grid-template-columns**: 1fr 1fr;

**grid-gap**: 1**em**;

**max-width**: 36**em**;

**margin**: 0 **auto**;

}

.**board** **>** input { **font-size**: 1.4**em**; **grid-column**: 1**/**3; }

h2 { **font-size**: 2**em**; **font-weight**: 200; **user-select**: **none**; **margin**: 0 0 0.5**em** 0; }

label {

**position**: **relative**;

**line-height**: 1.2;

**padding**: 0.5**em** 2.5**em** 0.5**em** 2**em**;

**margin**: 0 0 0.5**em** 0;

**border-radius**: 2**px**;

**user-select**: **none**;

**border**: 1**px** **solid** hsl(240, 8**%**, 70**%**);

**background-color**:hsl(240, 8**%**, 93**%**);

**color**: #333;

}

input**[**type**=**"checkbox"**]** { **position**: **absolute**; **left**: 0.5**em**; **top**: 0.6**em**; **margin**: 0; }

.**done** { **border**: 1**px** **solid** hsl(240, 8**%**, 90**%**); **background-color**:hsl(240, 8**%**, 98**%**); }

button {

**position**: **absolute**;

**top**: 0;

**right**: 0.2**em**;

**width**: 2**em**;

**height**: 100**%**;

**background**: **no-repeat** 50**%** 50**%** url("data:image/svg+xml,%3Csvg xmlns='http://www.w3.org/2000/svg' viewBox='0 0 24 24'%3E%3Cpath fill='%23676778' d='M12,2C17.53,2 22,6.47 22,12C22,17.53 17.53,22 12,22C6.47,22 2,17.53 2,12C2,6.47 6.47,2 12,2M17,7H14.5L13.5,6H10.5L9.5,7H7V9H17V7M9,18H15A1,1 0 0,0 16,17V10H8V17A1,1 0 0,0 9,18Z'%3E%3C/path%3E%3C/svg%3E");

**background-size**: 1.4**em** 1.4**em**;

**border**: **none**;

**opacity**: 0;

**transition**: **opacity** 0.2**s**;

**text-indent**: -9999**px**;

**cursor**: **pointer**;

}

label:hover button { **opacity**: 1; }

</style>

当点击待办事项时，会发送到相反的清单中。在真实的应用场景中，对象的行为不应该是这边点了消失，然后从另一边突兀地出现。它们应该使用运动（motion）动态地在靠中间位置移动，可以更好地帮助用户了解程序处理这个过程到底发生了什么事情。

我们可以使用 crossfade 函数来实现这种效果，该函数创建一对名为 send （发送）和 receive （接收）的过渡效果，当元素在 'sent' 的时候，它会找到相应的 'received' 接收元素，并将两者之间生成一个转换过程，将元素转换到对应位置后淡出。当一个元素是 'received' 的操作时，情况恰好相反，如果没有对应的项时，则使用 fallback 过渡来填补缺省效果。

找到 div.left 中的 <label>元素，然后加上send和receive过渡效果：

<label in:receive**=**{{key: todo.id}} out:send**=**{{key: todo.id}}>

div.right 中的<label>元素如法炮制：

<label

class**=**"done"

in:receive**=**"{{key: todo.id}}"

out:send**=**"{{key: todo.id}}"

>

至此，当你切换代办列表项时，它们会平滑移动到新的位置了。

没有过渡效果的列表项，依然会尴尬地“跳来跳去”，欲知应对手段如何，且看下回分解。

## 3、动画效果

回顾上一节，当元素从一个代办事项列表中移动到另一个代办事项列表时，我们使用了延迟过渡来创建运动效果。

为了完成这种效果，我们还需要对*没有*过渡效果的元素使用运行效果。为此，我们可以使用animate指令。

App.svelte

<script>

**import** { quintOut } from 'svelte/easing';

**import** { crossfade } from 'svelte/transition';

**const** [send, receive] **=** crossfade({

duration**:** d => Math.sqrt(d **\*** 200),

fallback(node, params) {

**const** style **=** getComputedStyle(node);

**const** transform **=** style.transform **===** 'none' **?** '' **:** style.transform;

**return** {

duration**:** 600,

easing**:** quintOut,

css**:** t => `

transform: ${transform} scale(${t});

opacity: ${t}

`

};

}

});

**let** uid **=** 1;

**let** todos **=** [

{ id**:** uid**++**, done**:** **false**, description**:** 'write some docs' },

{ id**:** uid**++**, done**:** **false**, description**:** 'start writing blog post' },

{ id**:** uid**++**, done**:** **true**, description**:** 'buy some milk' },

{ id**:** uid**++**, done**:** **false**, description**:** 'mow the lawn' },

{ id**:** uid**++**, done**:** **false**, description**:** 'feed the turtle' },

{ id**:** uid**++**, done**:** **false**, description**:** 'fix some bugs' },

];

**function** add(input) {

**const** todo **=** {

id**:** uid**++**,

done**:** **false**,

description**:** input.value

};

todos **=** [todo, ...todos];

input.value **=** '';

}

**function** remove(todo) { todos **=** todos.filter(t => t **!==** todo); }

**function** mark(todo, done) {

todo.done **=** done;

remove(todo);

todos **=** todos.concat(todo);

}

</script>

<div class**=**'board'>

<input

placeholder**=**"what needs to be done?"

on:keydown**=**{e => e.key === 'Enter' && add(e.target)}

>

<div class**=**'left'>

<h2>todo</h2>

{#each todos.filter(t => !t.done) as todo (todo.id)}

<label in:receive**=**"{{key: todo.id}}" out:send**=**"{{key: todo.id}}">

<input type**=**checkbox on:change**=**{() => mark(todo, true)}>

{todo.description}

<button on:click**=**"{() => remove(todo)}">remove</button>

</label>

{/each}

</div>

<div class**=**'right'>

<h2>done</h2>

{#each todos.filter(t => t.done) as todo (todo.id)}

<label class**=**"done" in:receive**=**"{{key: todo.id}}" out:send**=**"{{key: todo.id}}">

<input type**=**checkbox checked on:change**=**{() => mark(todo, false)}>

{todo.description}

<button on:click**=**"{() => remove(todo)}">remove</button>

</label>

{/each}

</div>

</div>

<style>

.**board** {

**display**: **grid**;

**grid-template-columns**: 1fr 1fr;

**grid-gap**: 1**em**;

**max-width**: 36**em**;

**margin**: 0 **auto**;

}

.**board** **>** input { **font-size**: 1.4**em**; **grid-column**: 1**/**3; }

h2 { **font-size**: 2**em**; **font-weight**: 200; **user-select**: **none**; **margin**: 0 0 0.5**em** 0; }

label {

**position**: **relative**;

**line-height**: 1.2;

**padding**: 0.5**em** 2.5**em** 0.5**em** 2**em**;

**margin**: 0 0 0.5**em** 0;

**border-radius**: 2**px**;

**user-select**: **none**;

**border**: 1**px** **solid** hsl(240, 8**%**, 70**%**);

**background-color**:hsl(240, 8**%**, 93**%**);

**color**: #333;

}

input**[**type**=**"checkbox"**]** {

**position**: **absolute**;

**left**: 0.5**em**;

**top**: 0.6**em**;

**margin**: 0;

}

.**done** { **border**: 1**px** **solid** hsl(240, 8**%**, 90**%**); **background-color**:hsl(240, 8**%**, 98**%**); }

button {

**position**: **absolute**;

**top**: 0;

**right**: 0.2**em**;

**width**: 2**em**;

**height**: 100**%**;

**background**: **no-repeat** 50**%** 50**%** url("data:image/svg+xml,%3Csvg xmlns='http://www.w3.org/2000/svg' viewBox='0 0 24 24'%3E%3Cpath fill='%23676778' d='M12,2C17.53,2 22,6.47 22,12C22,17.53 17.53,22 12,22C6.47,22 2,17.53 2,12C2,6.47 6.47,2 12,2M17,7H14.5L13.5,6H10.5L9.5,7H7V9H17V7M9,18H15A1,1 0 0,0 16,17V10H8V17A1,1 0 0,0 9,18Z'%3E%3C/path%3E%3C/svg%3E");

**background-size**: 1.4**em** 1.4**em**;

**border**: **none**;

**opacity**: 0;

**transition**: **opacity** 0.2**s**;

**text-indent**: -9999**px**;

**cursor**: **pointer**;

}

label:hover button { **opacity**: 1; }

</style>

首先，导入flip翻转函数，flip 表示svelte/animate中的['First, Last, Invert, Play'](https://link.zhihu.com/?target=https%3A//aerotwist.com/blog/flip-your-animations/)：

**import** { flip } from 'svelte/animate';

将其加到<label>元素中：

<label

in:receive**=**{{key: todo.id}}

out:send**=**{{key: todo.id}}

animate:flip

>

在这种情况下，移动稍显缓慢，因此我们添加一个duration参数控制动画的完成时长：

<label

in:receive**=**"{{key: todo.id}}"

out:send**=**"{{key: todo.id}}"

animate:flip**=**"{{duration: 200}}"

>

duration 也可以是一个 d => milliseconds 缓动函数，其中 d 是元素要移动的像素值

请注意，所有 transition 过渡动画和 animate 动画效果都是通过 CSS 实现的，而非 JavaScript，也就是说，它们不会阻塞主线程，或被主线程阻塞。

## 总结

页面动画主要是些过渡效果为主，与交互相关的还有运动（motion）。

动画效果就像西餐中一碟牛扒上的小花摆设，只为了“肉上添花”，不应恣意纵用，应宜谨慎克制，否则容易弄巧成拙。

Svelte 支持两种运动效果，一种是补间动画效果 tweened，另一种是弹簧效果 spring。这两种效果在页面中都十分常见，因此 Svelte 内置支持它们。

过渡效果更是页面动画的重头戏，由于十分常用，Svelte 为过渡动画效果专门创建了指令级别的支持，分别是 transition、in、out，它们都支持设置附加参数，例如 duration 用于设置动画运动的总时长。

transition 是将进入过渡和退出过渡两者齐集一身，我对 transition 功能没有意见，就是名字太长，简化为 trans 之类的就好了。

如果要更精细地分别控制进入和退出所使用的效果，此时应该使用 in 和 out 两个指令。

Svelte 无法为各种各样的动画提供预设支持，因此如果某些效果特别引人入胜，你可以通过自定义 CSS 过渡效果来包装它们，有些还需要配合 JS 创建的效果，Svelte 也是支持的。

局部过渡其实是将动画效果限制只应用在元素添加或删除时才触发，而在首次创建时元素不会被执行动画效果。

延迟过渡经常用于例如穿梭框之类的组件中，它可以实现提高用户体验的交互。这种效果通常会有一个发送端（send）和接收端（receive），例如 crossfade 效果。

# Svelte 官方入门教程（10）—— 动作

[上一章](https://zhuanlan.zhihu.com/p/369563637)，我们深入探讨了动画，能全部看完搞完练习完，是要花点时间的，起码说服自己愿意为了学习花这个时间，就需要自我挣扎不少时间。

人之常情是也。

每天略有小成即可，所谓日拱一卒，功不唐捐。

无论战国的苏秦自虐式恐怖学习方法——悬梁刺股，还是西汉的匡衡违反物业管理条例在邻居墙上凿壁偷窥…偷光，都无法持续，反而应该先治好精神病再说。似乎做人不够勤快地球就不转了，鼓吹怠惰可耻，孔子都没这么高的要求。

不必焦虑那些玄虚的宇宙熵增定律，什么时候有了情绪，来了兴致就看看学学，可坐可卧，看到妙处窃窃自喜，不亦乐乎，即便学过就忘，增进不了技术，增进了情操撒。

闲话休提，接下来我们探讨的是 Svelte 动作（Action）。

## 1、use 指令

动作（Action），其本质是元素上的生命周期函数。它们可用于譬如以下几个方面：

* 与第三方库对接
* 延迟加载图片
* 工具提示（tooltips）
* 添加自定义事件处理程序

我们模拟先做一个第三方库，要使第三方库适配 Svelte 的 Action 十分简单，实际上，Action 只是一个普通的函数，它接收一个参数，就是当前元素的 DOM 节点对象。

我们自制的“第三方”库的功能假设是一个让任意元素支持移动的 JS 库，它的框架大概是这样：

movable.js

**export** **function** movable(node) {

*// ... 第三方库的代码*

**return** {

destroy() {

*// ... destroy 函数会在元素被清除时调用，*

*// 我们可以在此做一些清理工作*

}

}

}

这个函数返回一个包含了 destroy 方法的对象，destroy 函数在元素被移除时自动被调用。

让任意元素支持移动，我们需要监听一些事件：

movable.js

**export** **default** **function** movable(node) {

**let** moving **=** **false**

**let** x, y, left, top

**function** handleMove(e) {

**if** (moving) {

node.style.left **=** (e.clientX **-** x **+** left) **+** 'px'

node.style.top **=** (e.clientY **-** y **+** top) **+** 'px'

}

}

**function** startMove(e) {

moving **=** **true**

x **=** e.clientX

y **=** e.clientY

left **=** parseInt(node.style.left) **||** 0

top **=** parseInt(node.style.top) **||** 0

}

**function** endMove() { moving **=** **false** }

window.addEventListener('mousemove', handleMove)

window.addEventListener('mouseup', endMove)

node.addEventListener('mousedown', startMove)

**return** {

destroy() {

window.removeEventListener('mousemove', handleMove)

window.removeEventListener('mouseup', endMove)

node.removeEventListener('mousedown', startMove)

}

}

}

完成了“第三方”组件后，我们现在来实现主程序，并通过 use 指令，将 movable 应用到指定的元素中：

App.svelte

<script>

**import** movable from './movable'

</script>

<style>

div {

**position**: **absolute**;

**width**: 100**px**;

**height**: 100**px**;

**background-color**: **blue**;

**text-align**: **center**;

**line-height**: 100**px**;

**color**: #fff;

**user-select**: **none**;

**cursor**: **move**;

}

</style>

*<!-- 通过 use 指令将 movable 应用到元素 div -->*

<div use:movable>Box</div>

此时，尝试拖动 div，可以看到 div 已经支持拖动。

下一步，我们要思考的问题是：div 能否监听得到它是什么时候开始移动？什么时候结束移动的？在这些时刻使用端可能有自己的事情要做。

这就需要 movable 支持发送这些事件供外部使用，我们计划设计3个事件，分别是：movestart、moving 和 moveend，顾名思义，它们分别代表移动开始、移动中和移动结束。

先在 UI 上监听这些事件：

<script>

**import** movable from './movable'

</script>

<style>

div {

**position**: **absolute**;

**width**: 100**px**;

**height**: 100**px**;

**background-color**: **blue**;

**text-align**: **center**;

**line-height**: 100**px**;

**color**: #fff;

**user-select**: **none**;

**cursor**: **move**;

}

</style>

<div

use:movable

on:movestart**=**{e => console.log(`start => x: ${e.detail.x}, y: ${e.detail.y}`)}

on:moving={e => console.log(`moving => x: ${e.detail.x}, y: ${e.detail.y}`)}

on:moveend={() => console.log('move end...')}

>Box</div>

然后转到 movable 去实现这些事件的发送：

**export** **default** **function** movable(node) {

**let** moving **=** **false**

**let** x, y, left, top

*// Note: 发送自定义事件*

**function** fire(type, option) {

node.dispatchEvent(**new** CustomEvent(type, option))

}

**function** handleMove(e) {

**if** (moving) {

node.style.left **=** (e.clientX **-** x **+** left) **+** 'px'

node.style.top **=** (e.clientY **-** y **+** top) **+** 'px'

*// Note: 发送 moving*

fire('moving', {

detail**:** { x**:** e.clientX **-** x **+** left, y**:** e.clientY **-** y **+** top }

})

}

}

**function** startMove(e) {

moving **=** **true**

x **=** e.clientX

y **=** e.clientY

left **=** parseInt(node.style.left) **||** 0

top **=** parseInt(node.style.top) **||** 0

*// Note: 发送 movestart*

fire('movestart', { detail**:** {x**:**left, y**:**top}})

}

**function** endMove() {

moving **=** **false**

*// Note: 发送 moveend*

fire('moveend')

}

window.addEventListener('mousemove', handleMove)

window.addEventListener('mouseup', endMove)

node.addEventListener('mousedown', startMove)

**return** {

destroy() {

window.removeEventListener('mousemove', handleMove)

window.removeEventListener('mouseup', endMove)

node.removeEventListener('mousedown', startMove)

}

}

}

这个实现方案仅用于演示，真实场景还需考虑触摸事件。

## 2、附加参数

正如 transition 过渡效果和 animate 动画一样，动作（Action）也支持附带参数，动作函数会与它所在的元素一起被调用。

我们现在设计了一个无聊透顶的游戏，叫 长按按钮 游戏，界面上有一个按钮，长按指定时间显示“恭喜你按够了 x 秒”之类的傻瓜提示，先来写 UI：

App.svelte

<script>

**import** { longpress } from './longpress.js';

**let** pressed **=** **false**;

**let** duration **=** 8;

</script>

<label>

<input type**=**range bind:value**=**{duration} max**=**{8} step**=**{1}>

{duration}s

</label>

<button

use:longpress

on:longpress**=**{() => pressed = true}

on:mouseenter={() => pressed = false}

>按住我别放</button>

{#if pressed}

<p>恭喜你，长按了 {duration} 秒</p>

{/if}

在这里我们使用longpress这个动作，每当用户按下按钮并持续一段给定的时间（通过 rang 组件提供）的话，该动作就会触发一个具有相同名称的时间。

现在有一个问题，我们如何将 pressed 告知 longpress 这个 action 呢？

答案自然就是通过动作参数。

事不宜迟，我们开始编写longpress.js文件：

longpress.js

**export** **function** longpress(node, duration) {

**let** timer;

**const** handleMousedown **=** () => {

timer **=** setTimeout(() => {

node.dispatchEvent(

**new** CustomEvent('longpress')

);

}, duration **\*** 1000);

};

**const** handleMouseup **=** () => clearTimeout(timer);

node.addEventListener('mousedown', handleMousedown);

node.addEventListener('mouseup', handleMouseup);

**return** {

destroy() {

node.removeEventListener('mousedown', handleMousedown);

node.removeEventListener('mouseup', handleMouseup);

}

};

}

longpress 借助计时器来控制时间，如果按够了 duration 秒，就展示恭喜信息。

回到App.svelte，我们可以传递一个具体的duration值给动作了：

<button

use:longpress**=**{duration}

...

>按住我别放</button>

如果你调整了持续时间的滑块，例如从5秒调整到2秒，你会发现还是需要5秒。

要修复这个问题，我们可以在longpress.js中添加一个update方法，每当参数有变，立即调用此方法：

longpress.js

**return** {

destroy() {

node.removeEventListener('mousedown', handleMousedown);

node.removeEventListener('mouseup', handleMouseup);

},

update(newDuration) {

duration **=** newDuration;

}

};

如果你需要将多个参数同时传递给一个动作，则需要将它们组合成一个对象，比如： use:longpress={{duration, spiciness}}

## 总结

此前在[《绑定》](https://zhuanlan.zhihu.com/p/369240017)一章，我们有讨论过 bind:this，通过 this 绑定，我们可以轻易获取到 DOM 节点，action 所不同的是，它有更强的封装性和复用性，将逻辑代码拆分到 JS 文件中供反复应用。

action 同样是一种为组件增强能力的特性，至此为止，已介绍 Svelte 三种封装这些增强能力的特性，第1种是封装 CSS 效果，第2种是自定义的 JS 效果，它们都将使用 transition、in 或 out 指令中。本章介绍的 action 则使用 use 指令。

各式各样功能强大的第三方库浩如烟海，通过 action，我们可以将其集成到 Svelte，利我之物，为我所用。物唯适用而称之为器。

# Svelte 官方入门教程（11）—— 样式

在 React 中操作样式类，我们最常用的是 [classnames](https://link.zhihu.com/?target=https%3A//www.npmjs.com/package/classnames) 这个库；Svelte 在这方面与 Vue 较为接近，是直接内置了对样式类操作的支持，不过没有 Vue 支持得那么强大。

**1、样式类**

我们的例子是一个选择性别的按钮组，最终效果如下：

样式示例

用户点击可以选择性别。例子的目的是展示如何进行样式切换。代码非常简单：

App.svelte

<script>

**let** current **=** '男';

</script>

<style>

.**selected** { **background-color**: #ff3e00; **color**: **white**; }

</style>

性别：

{#each ['男', '女', '保密'] as sex}

<button

class**=**{current === sex ? 'selected' : ''}

on:click**=**{() => current = sex}

>

{sex}

</button>

{/each}

如例子所示，设置样式类和其他属性一样，你可以使用 JavaScript 属性来指定样式类：

**class=**{current **===** 'sex' **?** 'selected' **:** ''}

这是 UI 开发中的一种常见模式，Svelte 有一个特殊的指令可以简化它：

<button

class:selected**=**{current === sex}

on:click**=**{() => current = sex}

>

{sex}

</button>

每当表达式的值为真时，都会将selected类添加到元素中，反之则将该类移除。

下方是另外一个例子，假设某个输入框必须输入数字：

https://pic1.zhimg.com/80/v2-08bb4fe16695ced555f0cfe3265d4564_1440w.jpg样式示例

如果输入了非数字，边框变红：

样式示例

通过 class 指令并配合使用 bind:value，可以较为轻松实现这种功能：

App.svelte

<script>

**let** value **=** 0

</script>

今天收入：<input bind:value class:error**=**{isNaN(Number(value))} />

<style>

.**error** { **border**: 1**px** **solid** **red**; }

input { **outline**: 0 **none**; }

</style>

**2、class 指令简写形式**

类的名称与它绑定的变量通常是相同的：

<div class:big**=**{big}>

*<!-- ... -->*

</div>

在这种情况下，我们可以使用简写形式：

<div class:big>

*<!-- ... -->*

</div>

**3、CSS自定义属性**

这是 v3.38.0 新增的功能。

之所以增加这个功能特性，是因为 Svelte 不会动态编译 CSS，也即不会动态编译 <style> 部分，且看下码：

<script>

**let** color **=** 'red';

</script>

<style>

.**btn** {

**color**: {**color**}**;**

border**:** 1px solid #000**;**

padding**:** 3px**;**

}

</style>

<span class**=**"btn">按钮</span>

在 <style> 标签中的 .btn 类，我们其中有一条规则 color: {color}，企图使用 color 变量的值来代表它的颜色值，当然，如果可以顺便的话，在 color 变量变化时，拜托也反应式一下自动更新字体颜色的值。

—— 这是奇思妙想，也是胡思乱想。

我其实十分喜欢上面这种方式，直观简单，我甚至以为本来就支持，不过，Svelte 是一个（引以为豪的）[编译器](https://zhuanlan.zhihu.com/p/351445824)，要实现编译出上面这种反应性的代码，实操起来会是一场噩梦。

故此，这段代码不会如你臆测中那样神奇地奏效。

但是，我们确实又需要这么一个特性，比方说主题，在线一换全换。即使 Svelte 一定程度上已经支持 [CSS in JS](https://zhuanlan.zhihu.com/p/351946894)。

既然在 style 中使用变量已是梦幻泡影，不过，你仍然可以用 CSS 变量的：

<script>

**let** color **=** 'red';

</script>

<style>

.**btn** {

**color**: **var**(**--color**);

**border**: 1**px** **solid** **var**(**--color**);

**padding**: 3**px**;

}

</style>

<span class**=**"btn" style**=**"--color: {color}">按钮</span>

这是结果展示（让边框也跟着变颜色）：

https://pic3.zhimg.com/80/v2-dec93da23c1d3e6ec4bd70e763ad9a1a_1440w.png

不过如上代码这么写，<span> 其中的 style 内联的对 --color 的赋值，又显得多此一举，让人觉得不尴不尬，左支右绌（似乎有些过分地吹毛求疵，得寸进尺）。

还要特别注意的是，CSS 变量会破坏 Svelte 组件局部作用域的特性，除非你控制缜密。

其他的主题的实现方式更复杂一些，如使用 :global 或者 上下文（Context，在后续章节介绍）。

事实上，这是在新特性之前，内联 style 是较为现实一些的方案，当然拼凑样式的这种方式看起来就很差。

于是乎，新特性是将它封装在一个组件中，假设是 Btn：

Btn.svelte

<style>

.**btn** {

**color**: **var**(**--color**);

**border**: 1**px** **solid** **var**(**--color**);

**padding**: 3**px**;

}

</style>

<span class**=**"btn">按钮</span>

然后使用端通过类似属性的方式，赋予这个 --color 的实际值：

App.svelte

<script>

**import** Btn from './Btn.svelte'

</script>

<Btn --color**=**"red" />

它同时也支持变量：

<script>

**import** Btn from './Btn.svelte'

**let** color **=** 'blue'

</script>

<Btn --color**=**{color} />

https://pic4.zhimg.com/80/v2-fd4d5739aca298dd28a8c423f7b695f7_1440w.png样式演示

目前这种方式来实现主题变换，至少比拼凑内联 style 属性的实现方式来得舒服些许，style 让样式与变量参糅混合更显脏乱。

增加自定义样式属性对当前的 Props 影响不大，因为特征鲜明，CSS 变量都是两个横杠作为前缀的，因此以 -- 开头的属性会被识别为样式属性，传递给组件中的任意样式，无论是类或者内联样式，不论元素层级。

当然，使用了这项特性的程序，浏览器的兼容性最低要求是得支持 CSS 变量的，至少浏览器得新一点较为稳妥（Chrome 89+）。

**总结**

我们在 UI 上操作类十分频繁，过从甚密。Svelte 内置了类似 classnames 的功能，自然最好不过，虽然不是很强大。

Svelte 有很多地方都支持简写或者叫速写形式，例如组件的属性，如果还记得的话，例如 href={href} 可以直接简写成 {href}。

再比如绑定指令，bind:value={value} 可简写为 bind:value，要我说，为何不干脆再“简益求简”简写为 :value，岂不更好，从语法分析上编译器并不难做到这一点。

除了简写外，还有内置的状态管理的 $ 符号，也能明显感觉到 Svelte 对开发者友好地释减重负，以实用为主。

不过 Svelte 的 class 指令目前不支持对象形式，或者数组形式，不免让人诟病。

Svelte 对样式类操作的内置支持其实是较为轻浅薄弱的。当然，事出有因，作者可能认为这样的设计未必是最优的，还在思考有什么更闪亮的方案。

某个组件/元素如果有多个样式要根据情况来应用，比如文本设置红色，边框设置红色，你就需要有多少个，写多少个：

<script>

**let** red **=** **false**;

**let** border **=** **false**;

</script>

<style>

.**red** { **color**: **red**; }

.**border** { **border**: 1**px** **solid** **red**; }

</style>

<h3 class:red class:border>this is text</h3>

<button on:click**=**{() => red = !red}>color</button>

<button on:click**=**{() => border = !border}>border</button>

这种属性如果有4、5个的话，我就会乖乖把 classnames 引入进来，因为它目前不支持下方这样的对象或者数组：

<h3 class**=**{{red, border}}>this is text</h3>

此外，目前已支持以自定义属性的方式，向组件传递 CSS 变量，对样式动态性要求的程序有一定程度上缓解。

对于样式相关的方案很多，那些年不管错过没错过，至少听过，如Sass、Less（个人用的较多）、PostCSS、CSS-in-JS 等等。

至少我是希望偏向使用框架直接内置的方案，这次的自定义样式属性，姑且妄论优劣，先得有，就是一种进步，先有后优。

# Svelte 官方入门教程（12）—— 上下文

Context，中文译为上下文，即语境，但从字面入手，恐怕较难理解。

打个比方，假设你30好几还是单身，今天准备去相亲，跟一个男孩/女孩约在南京路中餐厅吃饭，你与相亲对象之间，必然有一个联系人，我们通常称之为“媒婆”，通常嘴角有颗痣，还有媒婆扇子，媒婆就是你们之间（组件之间）能产生联系的上下文。

你对相亲对象有任何想要事前了解的信息，都能通过媒婆获悉；对方亦然。

女方一般要深入了解两类重要信息，分别是经济层面及哲学层面。

经济层面包括：房、车和钱，这无可厚非，不一定就说明此人物质，对此愤慨的一般是些刚毕业到30左右的小年青，如果你刚好全部都没有，但刚好女方看重这些，你就暂时不要强求了。

哲学层面包括：掉水里救谁之类的意识形态问题，这才是真正的麻烦，通常手尾很长。

男方那边通常就是肤浅的看漂不漂亮，较为理性的男生不一定只看外表，同样建议还有一个需要了解的哲学问题就是：生气时会不会摔东西？一般是房门、锅碗之类的无辜器物。

当然男人生气也可能摔门。即使外表中等，但决不摔门，就娶了吧，这辈子错不了。

**一、组件上下文**

**1、setContext 及 getContext**

组件 Context（上下文） API 为组件提供一种彼此 '对话' 机制，而无需将数据和函数作为属性来传递，或者发送大量的事件。

这是一项高级功能，但是很有用。

Context 相关的 API 函数有两个，分别是setContext和getContext。

如果组件调用了setContext(key, context)，那么其任意的***子组件***都可以通过const context = getContext(key)来获取这个 context。

先来写好媒婆：

MeiPo.svelte

<script>

**import** { setContext } from 'svelte'

**import** Boy from './Boy.svelte'

**import** Girl from './Girl.svelte'

setContext('boy', { car**:** 1, house**:** 0, money**:** 500000 })

setContext('girl', { slam**:** **true**, pretty**:** **true** })

</script>

<Boy />

<hr />

<Girl />

接着是男生和女生（不搞哲学属性了）：

Boy.svelte

<script>

**import** { getContext } from 'svelte'

**let** girl **=** getContext('girl')

</script>

<pre>

Boy 看女生：

-> 女生是否摔门？({girl.slam})

-> 女生是否漂亮？({girl.pretty})

</pre>

Girl.svelte

<script>

**import** { getContext } from 'svelte'

**let** boy **=** getContext('boy')

</script>

<pre>

Girl 看男生：

-> 房 ({boy.house}套)

-> 车 ({boy.car}辆)

-> 钱 (￥{boy.money})

</pre>

最后把媒婆 show 出来：

App.svelte

<script>

**import** MeiPo from './MeiPo.svelte'

</script>

<MeiPo />

Context 的值可以是任意对象，无所拘束。setContext及getContext类似[生命周期函数](https://zhuanlan.zhihu.com/p/369399405)，只能在组件初始化期间调用。

上述例子可以看出，如果你将 Boy 或者 Girl 组件单独拿出来用的话，会引发错误，因此在 getContext 时应该判断是否有值，如果它不在 MeiPo 的上下文中，它其实是没有值的。

**2、Context 的 Key**

我们可以使用任何类型的值作为键。

例如可以这样用setContext({}, ...)。

使用字符串作为键的不利之处就在于，不同的组件库有可能会意外地使用了同一个组件库。

而使用对象字面量，则可在任何情况下都保证键不会发生冲突（因为对象仅在与自身比较才相等，即{} !== {}而"x" === "x"），即使你有跨多组件层次的多个不同 context。

**3、Contexts vs Props vs Stores**

Contexts 与 Props 都是将在父组件声明的数据传达给子组件，它们唯一不同的是 Props 只能父传子，Contexts 可以父传孙或所有后代，即便中间隔了几代，或者是通过 slot （将在[下一章](https://zhuanlan.zhihu.com/p/371923722)详述）传过来的组件也能 get 得到。

当然，Contexts 同时也是享受不到 Props 那种属性绑定的好处了。

Contexts 和 [Stores](https://zhuanlan.zhihu.com/p/369399405) 如出一辙。它们之间的区别在于：stores 可用于应用 ***任意*** 位置；

而 Contexts 仅能用于 ***组件及其后代***，如果要使用一个组件的多个实例，每个实例的状态不希望干扰其他实例的状态，这可能会有所帮助。

实际上你可能会同时使用两者。Context 不是响应式的，因此那些会随时间变化的值，你应该使用 Stores，也即 Context 里某些属性可能是 Stores：

**const** { these, are, stores } **=** getContext('key');

Context 默认不具备反应性，如果需要上下文中的值支持反应性，应将值先存入 store，然后将 store 放到上下文中。

**二、模块上下文**

**1、共享代码**

到目前为止，在所有示例中，<script> 块均包含在初始化每个组件实例时运行的代码。对于绝大多数组件来说，这就是你所需要的。

有时，你需要在单个组件中的实例之外运行一些代码。例如，你可以同时播放所有这五个音频播放器。假设现在的交互是这么设计的：其中一个播放完毕，则同时也停止其他播放器。

先来实现音频播放器组件：

AudioPlayer.svelte

<script>

**export** **let** src;

**export** **let** title;

**let** audio;

**let** paused **=** **true**;

**function** stopOthers() {

*// TODO: 实现停止播放其他*

}

</script>

<article class:playing**=**{!paused}>

<h2>{title}</h2>

<audio bind:this**=**{audio} bind:paused

on:play**=**{stopOthers} controls {src}>

<track kind**=**"captions">

</audio>

</article>

我们拿5首曲子来试试：

App.svelte

<script>

**import** AudioPlayer from './AudioPlayer.svelte';

**let** url **=** 'https://sveltejs.github.io/assets/music/'

**let** songs **=** [

{ src**:** 'strauss.mp3', title**:** 'The Blue Danube Waltz' },

{ src**:** 'holst.mp3', title**:** 'Mars, the Bringer of War' },

{ src**:** 'satie.mp3', title**:** 'Gymnopédie no. 1' },

{ src**:** 'beethoven.mp3', title**:** 'Symphony no. 5 in Cm...' },

{ src**:** 'mozart.mp3', title**:** 'Requiem in D minor...' },

]

</script>

{#each songs as song}

<AudioPlayer src**=**{url + song.src} title**=**{song.title} />

{/each}

要使“其中一个播放完毕，则同时也停止其他播放器”，那么我们可以在 AudioPlayer.svelte 中，通过声明一个<script context="module">的代码块来实现这一点。

该模块中包含的代码，将在模块第一次运行时只执行一次，而不是每次实例化组件时都重复执行。我们将它放置在AudioPlayer.svelte的顶部：

<script context**=**"module">

**let** current;

</script>

现在，无需任何状态管理，组件之间就可以彼此 “交流” 了，看下方最后完善的 stopOthers 函数：

**function** stopOthers() {

**if** (current **&&** current **!==** audio) {

current.pause();

}

current **=** audio;

}

**2、导出**

从context="module"代码块中导出的任意内容，都将成为模块本身的导出。如果我们从AudioPlayer.svelte中导出stopAll函数...

AudioPlayer.svelte

<script context**=**"module">

**const** elements **=** **new** Set();

**export** **function** stopAll() {

elements.forEach(element => element.pause() });

}

</script>

...我们可以在App.svelte中导入...

App.svelte

<script>

**import** AudioPlayer, { stopAll } from './AudioPlayer.svelte';

</script>

...并在某个事件处理程序中使用它：

<button on:click**=**{stopAll}>

stop all audio

</button>

我们无法使用默认的导出，因为组件 *是* 默认就导出的。

**总结**

Svelte 中的 Context 概念和 React 和 Vue 没有区别，API 稍微不同，做的是一样的事情。

Svelte 对比 React 稍微要简单直观一些，抛却了 Provider 和 Consumer 的概念，事实上简单些就很好，就想组件间共享点数据罢了，没必要有板有眼创造一些高级隐喻，让其漂染浓厚的设计气息。

模块上下文的主要用途，是提供一种方法，让某个组件的所有实例都共享同一个上下文（这其实用 store 也完全能实现同等功能，只不过模块上下文更显得关联性高一些，以及封装性强一些）。

比方说最常见的是音乐播放器，在播放列表中，点击任意一曲，正在播放的音乐就要立即停止，然后播放选中的那一首，也就是说同一时刻只允许一首歌在播放，这种场景很适合模块上下文。

模块上下文写在 <script context="module"> 这个顶层 script 标记中，这是 Svelte 唯一允许你使用的第二个 顶层 script。

# Svelte 官方入门教程（13）—— 插槽

HTML 元素允许相互组合，由此构建复杂的页面。

如果不是普通的 HTML 元素，而是 Svelte 组件，同样支持让组件作为容器，为其添加子组件，以此组合出更大更强的组件，这种容器被称为 插槽（Slots）。

**1、插槽**

元素可以有子级节点...

<div>

<p> p 元素是 div 的子元素 </p>

</div>

..组件也可以有。但是，组件接收子级节点之前，它需要知道应该将子级节点放置何处。我们使用<slot>元素来应对此操作。

假设我们有一个名为 Box.svelte 的组件：

Box.svelte

<style>

.**box** {

**width**: 300**px**;

**border**: 1**px** **solid** #aaa;

**padding**: 20**px**;

}

</style>

<div class**=**"box">

*<!-- 内容应该注入到此处 -->*

</div>

我们在使用端引入 Box 组件：

App.svelte

<script>

**import** Box from './Box.svelte';

</script>

<Box>

*<!-- 将内容放置于此 -->*

</Box>

然后我们在Box.svelte内部加入插槽：

Box.svelte

<div class**=**"box">

<slot></slot>

</div>

你现在可以放置一些东西在这个 Box 中了：

App.svelte

<Box>

<h2>Hello!</h2>

<p>This is a box. It can contain anything.</p>

</Box>

**2、插槽缺省内容**

组件可以指定<slot>元素当内容为空的*缺省*情况应该默认显示什么内容：

Box.svelte

<div class**=**"box">

<slot>

<em>使用端没有传递任何内容时，将会展示此处的内容</em>

</slot>

</div>

现在我们再创建一个没有任何子级节点的<Box>看看：

App.svelte

<Box>

<h2>Hello!</h2>

<p>This is a box. It can contain anything.</p>

</Box>

<Box /> *<!-- 没有提供内容，则展示默认缺省情况的内容 -->*

**3、命名插槽**

上一个例子中，我们包含了一个*默认插槽*，用于直接代替组件的子节点。不过有些时候，你需要对放置的内容有更多的控制能力，例如使用下方示例中的<ContactCard>组件，我们可以使用 *命名插槽*：

ContactCard.svelte

<style>

.**contact-card** { **width**: 300**px**; **border**: 1**px** **solid** #aaa; **padding**: 1**em**; }

</style>

<article class**=**"contact-card">

<h3>

<slot>Unknown name</slot>

</h3>

<div>

<slot>Unknown address</slot>

</div>

</article>

默认的展示效果：

插槽演示

在ContactCard.svelte中，向每个插槽添加一个name属性：

<style>

.**contact-card** { **width**: 300**px**; **border**: 1**px** **solid** #aaa; **padding**: 1**em**; }

</style>

<article class**=**"contact-card">

<h3>

<slot name**=**"name">Unknown name</slot>

</h3>

<div>

<slot name**=**"address">Unknown address</slot>

</div>

</article>

然后在 App.svelte 中的<ContactCard>组件内添加相应的slot="..."属性：

App.svelte

<script>

**import** ContactCard from './ContactCard.svelte';

</script>

<ContactCard>

<span slot**=**"name">P. Sherman</span>

<span slot**=**"address">42 Wallaby Way Sydney</span>

</ContactCard>

某些情况下，我们可能会在 ContactCard.svelte 中，使用两个相同名称的 <slot>，当你需要在两个地方都展示一模一样的内容时，它就十分有用，例如你会遇到一些跟邮箱里的收件箱一样的需求：

插槽演示

可以看到邮件列表的顶部和底部，都有相同的操作按钮和分页导航：

https://pic2.zhimg.com/80/v2-542aaf22e0da16635618eae217562001_1440w.png

这种重复渲染的情况，我们通过使用相同名称的 <slot> 即可：

ContactCard.svelte

<style>

.**contact-card** { **width**: 300**px**; **border**: 1**px** **solid** #aaa; **padding**: 1**em**; }

</style>

<article class**=**"contact-card">

<h3>

<slot name**=**"name">Unknown name</slot>

</h3>

<div>

<slot name**=**"address">Unknown address</slot>

</div>

*<!-- 下方又添加了一个名称为 'name' 的 slot -->*

<h3>

<slot name**=**"name">Unknown name</slot>

</h3>

</article>

展示的结果是：



但是，在引用了 ContactCard 组件的使用端，你不允许提供两个相同名称相同的内容：

App.svelte

<script>

**import** ContactCard from './ContactCard.svelte';

</script>

<ContactCard>

<span slot**=**"name">P. Sherman</span>

*<!-- 不能提供两个 slot 为 'name' 的内容 -->*

<span slot**=**"name">P. Sherman</span>

<span slot**=**"address">42 Wallaby Way Sydney</span>

</ContactCard>

**4、检测插槽内容**

在某些情况下，你可能希望根据父级组件是否已提供某个插槽的内容来控制组件内的各个部分如何展示。

也许这个插槽被包裹在其他元素中，并且如果父级没有提供该插槽的内容，你可能这种情况下，不想渲染组件的某一部分。

或者你只想在插槽不为空的情况下，才应用某个样式类。

这种情况下，你可以使用专门用于检查插槽的变量 $$slots。

$$slots 是一个对象，对象的属性是父组件传入的所有插槽的名称。如果父级组件将插槽置空，$$slots 中不会有该插槽的名称。

例如下方的文章组件：

Article.svelte

<script>

**export** **let** title **=** ''

</script>

<article>

<h2>{title}</h2>

<hr />

<div><slot name**=**"content" /></div>

<div style**=**"margin-top: 30px;">

评论:

<slot name**=**"comments" />

</div>

</article>

App.svelte

<script>

**import** Article from './Article.svelte'

</script>

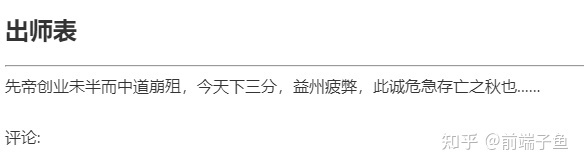
<Article title**=**"出师表">

<div slot**=**"content">

先帝创业未半而中道崩殂，今天下三分，益州疲弊，此诚危急存亡之秋也……

</div>

</Article>

插槽演示

可以看到，上方使用 Article 组件时，没有给出任何评论部分的内容，但是 “评论：” 还是展示了，我们要使用$$slots来确认仅在父组件<App>已传入comments插槽在有内容时才渲染这几个字。

我们编写一个if块，块的范围包括了comments评论插槽以及插槽所在的整个<div>，在if块中需要检测$$slots中的comments插槽是否存在内容：

Article.svelte

<script>

**export** **let** title **=** ''

</script>

<article>

<h2>{title}</h2>

<hr />

<div><slot name**=**"content" /></div>

{#if $$slots.comments}

<div style**=**"margin-top: 30px;">

评论:

<slot name**=**"comments" />

</div>

{/if}

</article>

现在，当<App>没有提供comments插槽任何内容时，整个评论部分的 UI 都不会渲染了。

**5、插槽属性**

假设现在要写一个简单的组件，该组件跟踪鼠标当前是否在其之上，假设命名为<Hoverable>：

Hoverable.svelte

<script>

**let** hovering;

**function** enter() { hovering **=** **true**; }

**function** leave() { hovering **=** **false**; }

</script>

<div on:mouseenter**=**{enter} on:mouseleave**=**{leave}>

<slot></slot>

</div>

App.svelte

<script>

**import** Hoverable from './Hoverable.svelte';

</script>

<style>

div { **padding**: 1**em**; **margin**: 0 0 1**em** 0; **background-color**: #eee; }

.**active** { **background-color**: #ff3e00; **color**: **white**; }

</style>

<Hoverable>

<div class:active>

{#if active}

<p>I am being hovered upon.</p>

{:else}

<p>Hover over me!</p>

{/if}

</div>

</Hoverable>

在 Hoverable 组件中，进入了 div 后会更新 hovering 的值。

Hoverable 需要将结果 *回传* 给父组件去，以便我们可以更新插槽中的内容。

于是我们用上了 *插槽属性*，在 Hoverable.svelte 中，将 hovering 的值传给插槽：

<div on:mouseenter**=**{enter} on:mouseleave**=**{leave}>

<slot hovering**=**{hovering}></slot>

</div>

再次安利你一下可以用 {hovering} 简写形式。

然后，我们要接受从<Hoverable>组件暴露的 hovering 的值，则可使用let指令：

<Hoverable let:hovering**=**{hovering}>

<div class:active**=**{hovering}>

{#if hovering}

<p>I am being hovered upon.</p>

{:else}

<p>Hover over me!</p>

{/if}

</div>

</Hoverable>

你可根据需要重命名这个变量，我们姑且在父组件中称之为active吧：

<Hoverable let:hovering**=**{active}>

<div class:active>

{#if active}

<p>I am being hovered upon.</p>

{:else}

<p>Hover over me!</p>

{/if}

</div>

</Hoverable>

你可以任意使用这类组件，并且插槽属性将仍然是在它们声明所在的组件中。

命名插槽也可以有属性；在具有 slot="..." 属性的元素上使用 let 指令，而不是在组件自身上使用。

**总结**

Svelte 的插槽，对比 Vue 的插槽而言，近似度很高，不过，目前 Svelte 不支持在组件中直接访问其插槽中的子组件，但功能已经在更新列表中，相信指日可待。

React 没有插槽的概念，它“默认”就允许组件嵌入子组件，子组件通过 props.children 来读取这些子组件，是否需要使用，完全视乎你的决定。

# Svelte 官方入门教程（14）完结篇 —— 特殊元素

Svelte 提供了各种内置的特殊元素。

## 1、<svelte:self>

首个介绍的是<svelte:self>，它表示当前组件，允许在某些情况递归自身，这对于展示文件夹树之类的视图很有用，其中文件夹可以包含 *其他*文件夹。

下方以展示文件夹树 Folder 为例：

Folder.svelte

<script>

**export** **let** name **=** 'noname';

**export** **let** folders **=** []

</script>

<style>

ul { **padding**: 0.2**em** 0 0 1.5**em**; **margin**: 0 0 0 0.5**em**; **list-style**: **none**; }

li { **padding**: 0.2**em** 0; }

</style>

<span>- {name}</span>

<ul>

{#each folders as f}

<li>

*<!-- 此处展示下一级的文件夹 -->*

</li>

{/each}

</ul>

App.svelte

<script>

**import** Folder from './Folder.svelte';

**let** folders **=** [

{

name**:** '音乐',

folders**:** [

{ name**:** '流行音乐' },

{ name**:** '纯音乐' },

{ name**:** '吉他' }

]

},

{

name**:** '电影',

folders**:** [

{

name**:** '喜剧片',

folders**:** [

{ name**:** '周星驰' },

{ name**:** '黄渤' },

{ name**:** '沈腾' }

]

},

{

name**:** '动作片',

folders**:** [ { name**:** '李连杰' } ]

}

]

}

];

</script>

<Folder name**=**"D盘" {folders} />

当前的结果只显示了根目录：

https://pic4.zhimg.com/80/v2-1fefeb10a1e7f9e9c38d6cd43be94177_1440w.png

要显示所有子目录，这样是不可以的：

Folder.svelte

{#each folders as f}

<li>

<Folder {...f} />

</li>

{/each}

也就是说，你不可以在 Folder.svelte 中直接使用 <Folder> 自身，因为模块不能导入自身。但是我们可以使用<svelte:self>代表自身：

Folder.svelte

{#each folders as f}

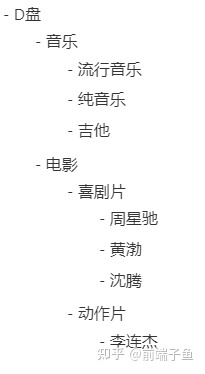
<li>

<svelte:self {...f} />

</li>

{/each}

最终的结果：



基于 <svelte:self> 的作用是引用自身组件的构造函数，因此它十分容易导致堆栈溢出。

试想一下，Folder 组件中又使用了 Folder 组件，如果不加以“限制”，默认将会无穷无尽地渲染下去，直到爆栈报错。

所以，<svelte:self> 必须包裹在 #if 块或者 #each 块中，或者由插槽传给组件，其余情况均是禁止的。

## 2、<svelte:component>

组件可以通过<svelte:component>进行 “变身”，这样可以省掉一系列的if块...

例如我们有3个十分简单的组件（显示不同颜色的文本）：

Red.svelte

<strong style**=**"color: red">red text</strong>

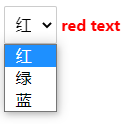
Green.svelte

<strong style**=**"color: green">green text</strong>

Blue.svelte

<strong style**=**"color: blue">blue text</strong>

如果我们的主程序中需要根据某个下拉框的选择的不同，展示不同的颜色文本：

https://pic4.zhimg.com/80/v2-337b7ad9bb43e340b629fcebe3380e67_1440w.png

那么程序可能需要很多的 if 来做判断：

App.svelte

<script>

**import** Red from './Red.svelte';

**import** Green from './Green.svelte';

**import** Blue from './Blue.svelte';

**let** selected;

</script>

<select bind:value**=**{selected}>

<option value**=**"red">红</option>

<option value**=**"green">绿</option>

<option value**=**"blue">蓝</option>

</select>

{#if selected == 'red'}

<Red />

{:else if selected == 'green'}

<Green />

{:else }

<Blue />

{/if}

通过使用 <svelte:component>，可以消除掉诸多 if，更为动态地使用这3个组件：

App.svelte

<script>

**import** Red from './Red.svelte';

**import** Green from './Green.svelte';

**import** Blue from './Blue.svelte';

**let** selected **=** '0';

$**:** component **=** [Red, Green, Blue][selected]

</script>

<select bind:value**=**{selected}>

<option value**=**"0">红</option>

<option value**=**"1">绿</option>

<option value**=**"2">蓝</option>

</select>

<svelte:component this**=**{component} />

this的值可以是任何组件的构造函数，如果提供的是一个假值，则不会渲染这个组件。

## 3、<svelte:window>

正如任何 DOM 元素都可以将监听事件一样，你可以通过<svelte:window>来监听window对象的事件，例如鼠标移动的事件：

App.svelte

<script>

**let** x, y

</script>

<svelte:window on:mousemove**=**{ e **=**> ({x, y} = e) } />

<p>鼠标位置：{x}, {y}</p>

你也可以添加像 preventDefault 这样的 [事件修饰符](https://zhuanlan.zhihu.com/p/369228329)，这与 DOM 元素的方法并无二致。

**绑定**

我们还可以绑定一些window的属性，某些业务我们可能需要监测浏览器窗口的大小，原生的写法是通过监听 window.onresize 来实现的。

在 Svelte 中，它更为简单，并且支持绑定：

App.svelte

<script>

**let** width, height

</script>

<svelte:window bind:innerWidth**=**{width} bind:innerHeight**=**{height} />

<p>Window Size: width={width}, height={height}</p>

支持绑定的属性列表如下：

* innerWidth - 获得浏览器窗口的内容区域的宽度，包含垂直滚动条(如果有的话)
* innerHeight - 获得浏览器窗口的内容区域的高度，包含水平滚动条(如果有的话)
* outerWidth - 获得浏览器窗口的宽度
* outerHeight - 获得浏览器窗口的高度
* scrollX - document 在水平方向已滚动的像素值
* scrollY - document 在垂直方向已滚动的像素值
* online - window.navigator.onLine 的别名，代表当浏览器能够访问网络。

除了 scrollX 和 scrollY 两个属性之外，其余均是只读的。

## 4、<svelte:body>

与<svelte:window>类似，<svelte:body>元素允许你监听document.body上的事件。这对于mouseenter和mouseleave事件来说十分有用，它们无法在window上触发。

App.svelte

<script>

**let** msg **=** ''

</script>

<svelte:body

on:mouseenter**=**{() => msg = 'Enter'}

on:mouseleave={() => msg = 'Leave'}

/>

<div>{msg}</div>

## 5、<svelte:head>

<svelte:head>元素允许你将其他元素插入到文档中的<head>中：

App.svelte

<svelte:head>

*<!-- 插入到 <head> 的内容写在这里 -->*

</svelte:head>

<h1>Hello world!</h1>

<head> 中可以插入例如 <script>、<link>、<title> 和 <meta> 等等元素：

App.svelte

<svelte:head>

<link rel**=**"stylesheet" href**=**"./global.css">

</svelte:head>

在服务器端渲染 (SSR) 模式下，<svelte:head> 的内容与 HTML 的其余部分是分别返回的。

## 6、<svelte:fragment>

回顾[第13章](https://zhuanlan.zhihu.com/p/371923722)《插槽》中的第3小节 —— 命名插槽，我们向命名插槽提供内容时，无可避免需要借助一个元素或者组件，例如：

<span slot**=**"address">42 Wallaby Way Sydney</span>

如果没有 span 元素，那么 slot="address" 无处安放。

假设在某些情况下，我们借助别的元素还可能会影响到组件原有的布局，甚至要被迫放弃原有的 UI 描述方案，例如下方用于罗列菠萝包制作所需的材料列表的例子：

Material.svelte

<h4>菠萝包的食材用料：</h4>

<ul>

<li>盐</li>

<li>砂糖</li>

<li>奶粉</li>

<slot name**=**"bottom" />

</ul>

<style>

h4 { **background**: **green**; **padding**: 1**em**; **color**: #fff; }

</style>

例子中预留了一个位置供使用端补充更多的用料，比如在 bottom 位置再添加一个“高筋面粉”：

App.svelte

<script>

**import** Material from './Material.svelte'

</script>

<Material>

<li slot**=**"bottom">高筋面粉</li>

</Material>

从展示的结果图来看，目前十分完美：

&amp;amp;amp;amp;lt;svelte:fragment&amp;amp;amp;amp;gt; 示例

菠萝包的制作用料中还有“低筋面粉”，如果你也想放到 bottom 位置，那么尴尬的事情就发生了：

App.svelte

<script>

**import** Material from './Material.svelte'

</script>

<Material>

<li slot**=**"bottom">高筋面粉</li>

<li slot**=**"bottom">低筋面粉</li>

</Material>

Svelte 不允许提供两个相同 slot 名称的内容（参考[第13章](https://zhuanlan.zhihu.com/p/371923722)《插槽》中的第3小节），你将收到一个错误信息：“Duplicate slot name "bottom" in <Material>”。

下一个立即被想到的方案是这样：

App.svelte

<script>

**import** Material from './Material.svelte'

</script>

<Material>

<div slot**=**"bottom">

<li>高筋面粉</li>

<li>低筋面粉</li>

</div>

</Material>

为赋新词强说愁。强加了一个 div 作为容器，看看结果似乎又是对的：



但不要被表面蒙蔽，以下是实际的 HTML：



如果将所有用料使用横向布局，这个问题原形毕露：

Material.svelte

...

<style>

h4 { **background**: **green**; **padding**: 1**em**; **color**: #fff; }

*/\* 使列表呈横向布局 \*/*

ul { **display**: **flex**; }

:global**(**.**list** li**)** { **margin**: **auto** 20**px**; }

</style>

你可能留意到，最后一个样式规则，使用的是 :global( )，这是因为通过 <slot> 添加的 <li> 用料项目，**不会**自动应用与 Material 组件中的 <li> 相同的样式。  
因为样式在组件中默认就是局部作用域的。  
  
这种情况下，你必须将这个样式使用全局样式（通过 :global 伪类），<slot> 内容里的 <li> 才可以应用得 Material 中 <li> 的样式。

结果如下所示：



这就是前面所述的**“可能会影响到组件原有的布局”**的问题，<svelte:fragment> 正是为了解决这个问题而生，只需要将 <div> 改为 <svelte:fragment>，问题迎刃而解：

App.svelte

<script>

**import** Material from './Material.svelte'

</script>

<Material>

<svelte:fragment slot**=**"bottom">

<li>高筋面粉</li>

<li>低筋面粉</li>

</svelte:fragment>

</Material>

## 7、<svelte:options>

<svelte:options>允许你配置编译器的选项。

例如你希望为组件生成属性访问器（默认不生成）：

App.svelte

<script>

**export** **let** x **=** 'hello'

</script>

<svelte:options accessors />

<p>{x}</p>

通过给 svelte:options 加入 accessors 后，最终生成的 App 类如下：

**class** App **extends** SvelteComponent {

constructor(options) {

**super**();

init(**this**, options, instance, create\_fragment, safe\_not\_equal, { x**:** 0 });

}

get x() {

**return** **this**.$$.ctx[0];

}

set x(x) {

**this**.$set({ x });

flush();

}

}

**export** **default** App;

你可以留意到，变量 x 已经生成为 App 类的一个属性，如果没有指定这个编译选项，生成的 App 类则是这样：

**class** App **extends** SvelteComponent {

constructor(options) {

**super**();

init(**this**, options, instance, create\_fragment, safe\_not\_equal, { x**:** 0 });

}

}

**export** **default** App;

生成属性访问器的目的，是提供给使用端更容易去访问的，这通常是你编写了一个 Svelte 组件，想同时在 React 或者 Vue 等地方上用时才需要这个选项。

下列是允许在此处设置可选用的值：

* immutable={true}：你确信不会使用可变数据，因此编译器可以执行简单的引用相等性检查来确定值是否已更改。
* immutable={false}：默认值，Svelte 将按默认的方式处理可变对象的更改
* accessors={true}：为组件的 props 添加 getter 和 setter
* accessors={false}：默认值
* namespace="..."：将使用此组件的命名空间，最常见的就是 "svg"
* tag="..."：将组件编译为自定义元素时，指定的标签名称

有关这些选项的详细信息，请参考API文档。

## 总结

对于 <svelte:options> 来说，99.9% 的情况下，你无需手工干预编译器的默认选项，除非你有十分明确的理由。

<svelte:fragment> 跟 Vue 的 <template> 功能近似，都是期望作为元素的容器，但同时自身无需渲染。

<svelte:window> 提供了一些十分有用的绑定，例如与尺寸相关的 innerWidth和 innerHeight，以及与滚动条相关的 scrollX 和 scrollY，这两个还支持赋值，它可以使编写这方面的程序变得更简单。