# 介绍

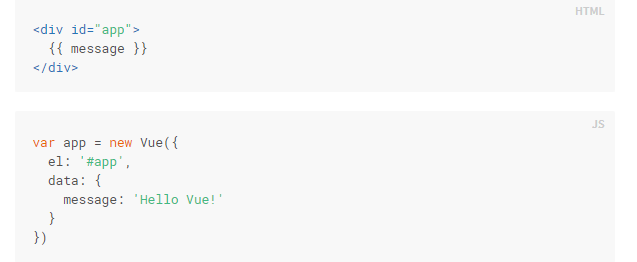
## Vue.js 是什么

Vue (读音 /vjuː/，类似于 view) 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。

Vue 的核心库只关注视图层

1. **声明式渲染**

Vue.js 的核心是一个允许采用简洁的**模板语法**来声明式地**将数据渲染进 DOM 的系统**：

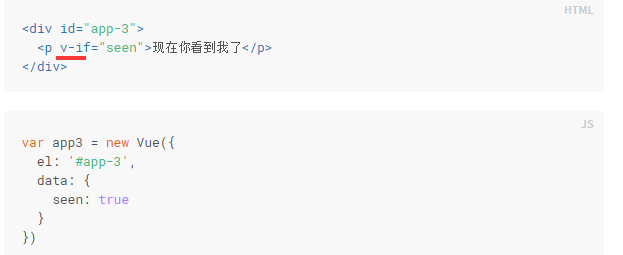


现在数据和 DOM 已经被建立了关联，所有东西都是**响应式的**。

**绑定元素特性**

v-bind 特性被称为指令。指令带有前缀 v-，以表示它们是 Vue 提供的特殊特性。

1. **条件与循环**



Vue 也提供一个强大的过渡效果系统，可以在 Vue 插入/更新/移除元素时自动应用过渡效果。

1. **处理用户输入**

用 v-on 指令添加一个事件监听器，通过它调用在 Vue 实例中定义的方法：



1. **组件化应用构建**

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。

## Vue 实例

每个 Vue 应用都是通过用 Vue 函数创建一个新的 Vue 实例开始的：

1. **数据与方法**

当一个 Vue 实例被创建时，它向 Vue 的响应式系统中加入了其 data 对象中能找到的所有的属性。当这些属性的值发生改变时，视图将会产生“响应”，即匹配更新为新的值。

当这些数据改变时，视图会进行重渲染。值得注意的是只有当实例被创建时 data 中存在的属性才是响应式的。也就是说如果你添加一个新的属性，那么对 b 的改动将不会触发任何视图的更新。

如果你知道你会在晚些时候需要一个属性，但是一开始它为空或不存在，那么你仅需要设置一些初始值。

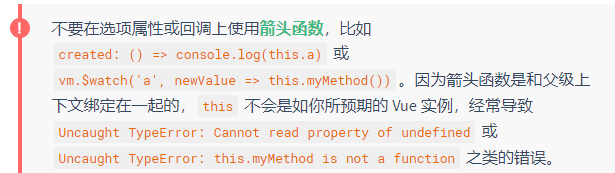
这里唯一的例外是使用 Object.freeze()，这会阻止修改现有的属性，也意味着响应系统无法再追踪变化。

除了数据属性，Vue 实例还暴露了一些有用的实例属性与方法。它们都有前缀 $，以便与用户定义的属性区分开来。例如：



1. **实例生命周期钩子**

每个 Vue 实例在被创建时都要经过一系列的初始化过程, 同时在这个过程中也会运行一些叫做生命周期钩子的函数，这给了用户在不同阶段添加自己的代码的机会。



1. **生命周期图示**



# 探索

A Vue object is created using the **new** keyword and the constructor accepts a **configuration object** whose properties provide settings that control how the application **behaves** and define the **content** that it presents to the user.

* **el**：identify the **element** in **index.html** that will be **replaced** to display the application content
* template：provide Vue.js with the **HTML content** that will **replace** the element matched by the **el** property.

1. **Adding Data to the Vue Object**

**Data bindings** are an important Vue.js feature that provides the connection between the Vue object’s **template** content and its **data** object.

The data is **live** or **reactive**, meaning that a change to the value of the counter property will be automatically reflected in the contents

When the **Vue object** is created, it processes the **data object** and **replaces** the properties so that it can detect when there are changes.

1. **Adding an Event Handler**



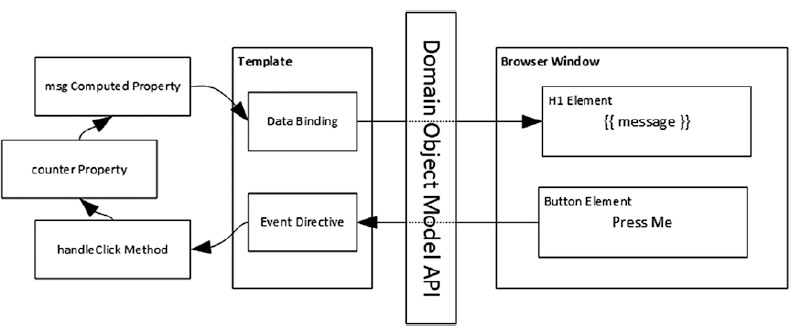
The attribute I added to the button element is a **directive**

1. **Correcting the Message**

The **computed** configuration defines data values that require some computation to produce a result.

## Understanding the Vue Object Structure

The Vue object provides the **same** functionality as the code that used the DOM API。but the result is an application that has **better structure**, which is easier to **develop** and **maintain** than working **directly** with the DOM.



### Introducing a Component

The **Vue object** helps to separate the different **aspects** of an application, but the result is still awkward

In most projects, the **Vue object** is used to **configure** the application, but the application’s **functionality** is defined using ***components***, which **extend** the functionality of the Vue object and which can be defined in files that contain a more helpful mix of content types.

Components allow an application to be **composed** of **multiple components** that are combined to provide the user with complex functionality.

There are **two important changes** to the JavaScript code when moving from a regular Vue object to a component.

1. The first is that the configuration object must be **exported**

This allows the object to be imported from its **module**

1. The other change is to the **data property**, which must be expressed as a **function**

The **data property** is assigned a **function** that **returns an object** whose properties provide the **data values** for the component.

#### Registering and Applying the Component

Components must be **registered** with Vue.js **before they are used**, and this is one of the main jobs usually performed in the **main.js** file.

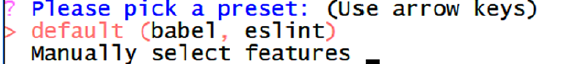
#### Using Separate JavaScript and HTML Files



# Understanding Vue.js Projects and Tools

## Creating a Vue.js Development Project

The **vue create** command uses an interactive process for selecting the options to create a project



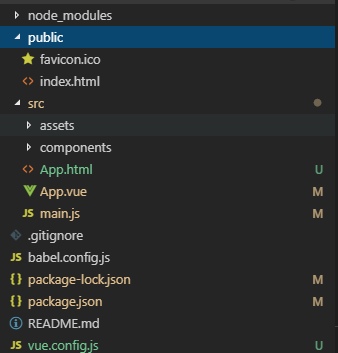
* default
* The other option is to manually select the features that you require

There are two options.

You can choose to use **separate configuration files** for each of the features that require configuration,

or you can choose to include all of the configuration settings in the **package.json** file, which is also used to **keep track** of the packages required by the application.

## Understanding the Project Structure



* public

**static** project assets, such as images that are **not incorporated into the bundle files** sent to the browser

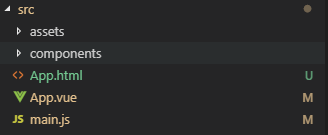
* src

the Vue.js application and its resources, and it is the main focus during development

* .babel.config.js

contains the configuration settings for the Babel compiler

### Understanding the Source Code Folder



* assets

used for the static resources required by the application and that are **included** in the **bundling process**,

* components

used for the application’s components。

* views

contain components that are **displayed** using the URL routing feature

* App.vue

the root component

* main.js

creates the Vue object

* router.js

used to configure the URL routing system，which is used to select the components that are displayed to the user.

* store.js

configure the data store，which is used to share data throughout the application.

## Using the Development Tools

The key package used by the development tools is called webpack, which is the backbone for many JavaScript development tools and frameworks.

**Webpack** is a module bundler, which means that it packages JavaScript modules for use in a browser, although that’s a bland description for an important function, and it is one of the key tools that you will rely on while developing a Vue.js application

Webpack **starts** with the main.js file and **loads** all the modules for which there are **import** or **require** statements to create a set of dependencies.

### Understanding the Compilation and Transformation Process

Although webpack focuses on pure JavaScript, its functionality can be extended to deal with other types of content through extensions called **loaders**.

The Vue.js development tools include several **loaders**, but there are **two** of particular note.

1. **vue-loader**

responsible for **transforming** the **mixed content** in .vue files so that it can be compiled and packaged to be consumed by the browser, which is the **foundation** for being able to define components in a single file that **mixes HTML, JavaScript, and CSS content**.

1. **Babel compiler**

responsible for **compiling** JavaScript code that uses responsible for compiling JavaScript code that uses

### Understanding the Development HTTP Server

the webpack-dev-server package: HTTP server that is integrated with webpack

When an HTTP request is received, the development HTTP server returns the contents of the **public**/index.html file.

As it processes the index.html file, the development server makes an important addition, which you can see by right-clicking in the browser window and selecting View Page Source from the pop-up menu

The development server adds **a script element** that tells the browser to load a file called **app.js**, which is the name of the bundle created by webpack during the startup sequence for the development tools.

The bundle doesn’t contain just the application code, however, and additional features are sent to the browser that are integral to the Vue.js development experience, as described in the sections that follow.

### Understanding Hot Model Replacement

The bundle that webpack creates includes support for a feature called **Hot Module Replacement (HMR).**

When you make a change to the application’s source or content files, the altered file is processed by webpack and its loaders, bundled, and sent to the browser.

# Understanding Data Bindings

## Understanding the Elements of a Component

Components are defined in .vue files that contain HTML content that is displayed to the user, data and JavaScript code to support the content, and CSS styles

The convention in Vue.js projects is to define a root component that orchestrates the rest of the application; this is the App component

1. **Template Element**

The **template** element contains the component’s **HTML content**.

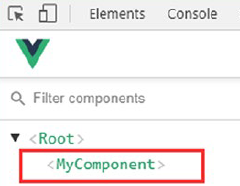
Templates contain **a single top-level** element, which replaces the element that applies the component

A component’s template contains a mix of regular **HTML** elements and Vue.js enhancements that apply application features, such as **data bindings**, **directives**, and **custom elements** that apply other components.

1. **Script Element**

component’s JavaScript module, whose properties **configure** the component, define its **data model**, and provide the **logic** that supports its features.

The Vue Devtools browser extension uses the optional **name** property to show the structure of the application, and picking a distinctive and meaningful name can help make sense of what’s happening when you inspect the application’s state.



1. **Style Element**

defines CSS styles

apply **only** to the template elements or have a wider effect in the application.

## Displaying a Data Value

Displaying a data value requires two steps:

1. defining a **data property** in the **script** element
2. and adding a **data binding** to the **template** element to present the value to the user

the **data** property returns a **function**, the function returns an **object**, and the properties defined by the object can be used to display data values to the user.

Data values in the **script** element are linked to **HTML** elements in the template element using **data bindings**.

The double braces are said to resemble a mustache, which is why this is also known as the *mustache binding*.

### Using More Complex Expressions in a Data Binding

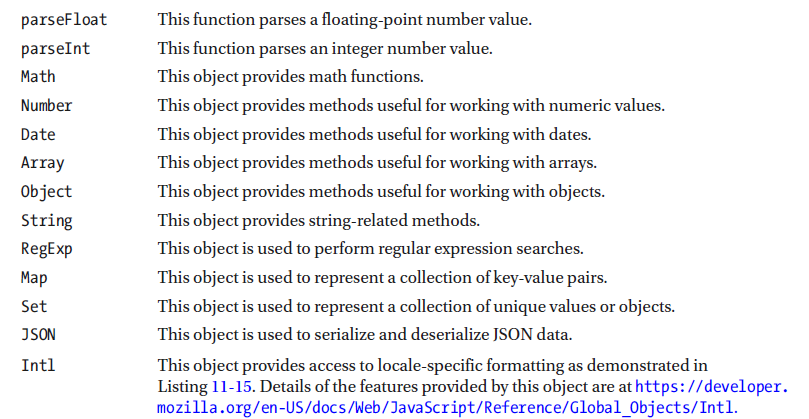
Treating data bindings as **expressions** means you can use more complex JavaScript statements in data bindings.

Expressions can contain **only a single statement**, and it must produce a result，which means you cannot invoke functions or perform complex tasks in a data binding.

The **context** of the expression is the **component**, which means you can only access the data defined by the component whose template contains the data binding

it also prevents me from accessing any of the global objects that browsers provide for JavaScript code (such as window and document) or data defined by other components.

Expressions can also access **commonly used JavaScript global objects** and functions, such as the **Math** and **JSON** objects, which are used to access math functions and to deal with JSON data



**excludes** access to objects that could be used to perform dangerous operations, such as manipulating the DOM.

### Calculating Values with Computed Properties

generate values **based on data properties** so that you don’t have to include complex expressions in data bindings.

A **computed** property is added to the component’s JavaScript module and assigned **a literal object** whose property names are the names of the computed properties.

Computed properties are used just like regular data properties in data bindings

## new Vue的工作

new Vue

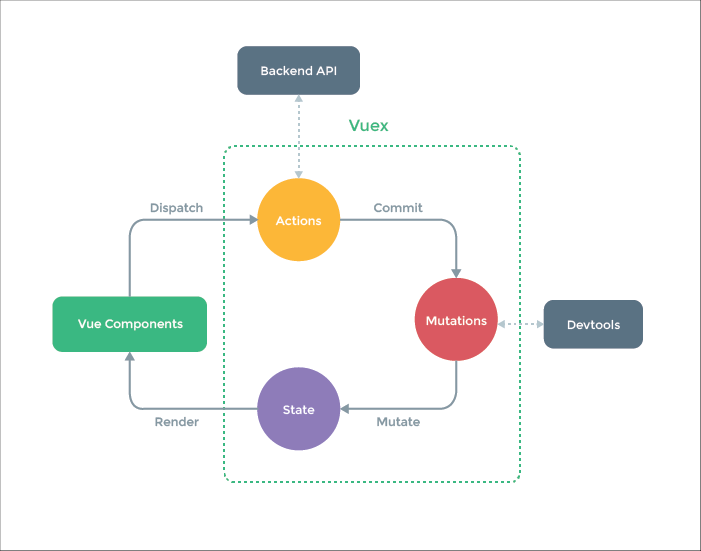
# vuex

Vuex is a **state management pattern + library** for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion.

 It also integrates with Vue's official [devtools extension](https://github.com/vuejs/vue-devtools" \t "_blank) to provide advanced features such as zero-config time-travel debugging and state snapshot export / import.

So why don't we extract the shared state out of the components, and manage it in a global singleton? With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

Vuex is also a library implementation tailored specifically for Vue.js to take advantage of its granular reactivity system for efficient updates.



At the center of every Vuex application is the **store**. A "store" is basically a container that holds your application **state**.

There are two things that make a Vuex store different from a plain global object:

* Vuex stores are **reactive**. When Vue components retrieve state from it, they will reactively and efficiently update if the store's state changes.
* You cannot directly mutate the store's state. The only way to change a store's state is by explicitly **committing mutations**. This ensures every state change leaves a track-able record, and enables tooling that helps us better understand our applications.

## State

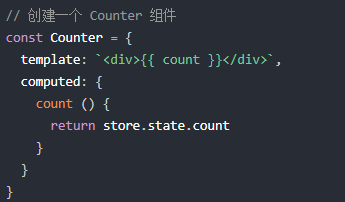
### 单一状态树

Vuex 使用单一状态树: 用一个对象就包含了全部的应用层级状态。

每个应用将仅仅包含一个 store 实例。单一状态树让我们能够直接地定位任一特定的状态片段，在调试的过程中也能轻易地取得整个当前应用状态的快照。

### 在 Vue 组件中获得 Vuex 状态

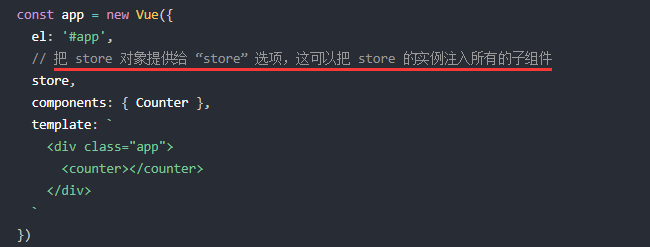
由于 Vuex 的状态存储是**响应式**的，从 store 实例中读取状态最简单的方法就是**在计算属性中返回某个状态**：



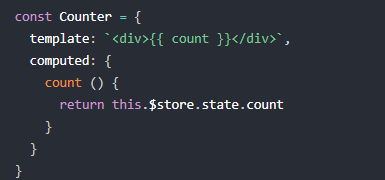
每当 store.state.count 变化的时候, 都会**重新求取计算属性**，并且**触发更新**相关联的 DOM。

然而，这种模式**导致组件依赖全局状态单例**。在模块化的构建系统中，在每个需要使用 state 的组件中需要频繁地导入，并且在测试组件时需要模拟状态。

Vuex 通过 store 选项，提供了一种机制**将状态从根组件“注入”到每一个子组件中（需调用 Vue.use(Vuex)）**：



通过在根实例中注册 store 选项，该 store 实例会注入到根组件下的所有子组件中，且子组件能通过 this.$store 访问到。让我们更新下 Counter 的实现：

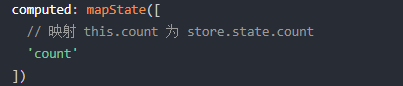


### mapState 辅助函数

当一个组件需要获取多个状态时候，将这些状态都声明为计算属性会有些**重复和冗余**。为了解决这个问题，我们可以使用 **mapState** 辅助函数帮助我们**生成计算属性**，让你少按几次键：

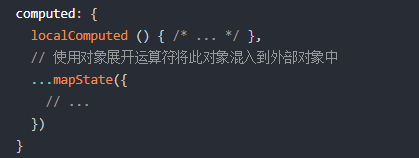


当映射的计算属性的名称与 state 的子节点名称相同时，我们也可以给 mapState 传一个字符串数组。



### 对象展开运算符

mapState 函数返回的是一个**对象**。我们如何将它与局部计算属性混合使用呢？通常，我们需要使用一个工具函数将多个对象合并为一个，以使我们可以将最终对象传给 computed 属性。但是自从有了[对象展开运算符](https://github.com/sebmarkbage/ecmascript-rest-spread" \t "_blank)（现处于 ECMASCript 提案 stage-4 阶段），我们可以极大地简化写法：

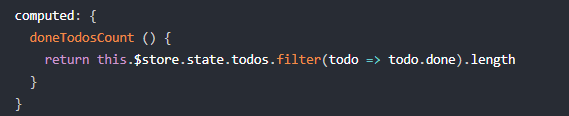


### 组件仍然保有局部状态

使用 Vuex 并不意味着你需要将**所有的**状态放入 Vuex。虽然将所有的状态放到 Vuex 会使状态变化更显式和易调试，但也会使代码变得冗长和不直观。如果有些状态严格属于单个组件，最好还是作为组件的局部状态。你应该根据你的应用开发需要进行权衡和确定。

## Getter

有时候我们需要从 store 中的 state 中**派生**出一些状态，例如对列表进行过滤并计数：



如果有多个组件需要用到此属性，我们要么复制这个函数，或者抽取到一个共享函数然后在多处导入它——无论哪种方式都不是很理想。

Vuex 允许我们在 store 中定义“getter”（可以认为是 store 的计算属性）。就像计算属性一样，getter 的返回值会根据它的依赖被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。

Getter 接受 state 作为其第一个参数：

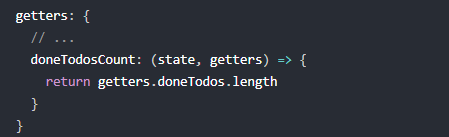


### 通过属性访问

Getter 会暴露为 store.getters 对象，你可以以属性的形式访问这些值：

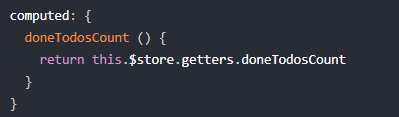


Getter 也可以接受其他 getter 作为第二个参数：



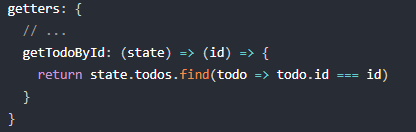


我们可以很容易地在任何组件中使用它：



### 通过方法访问

也可以通过让 getter 返回一个**函数**，来实现给 getter 传参。在你对 store 里的数组进行查询时非常有用

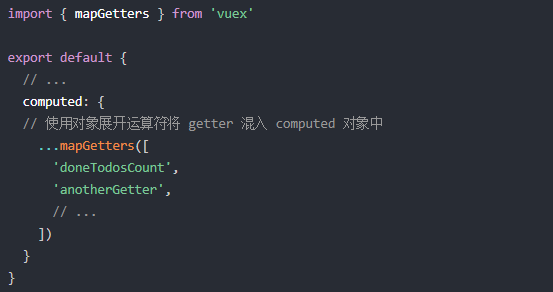




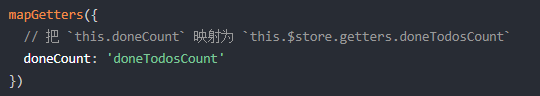
注意，getter 在通过方法访问时，每次都会去进行调用，而**不会缓存结果**。

### mapGetters 辅助函数

mapGetters 辅助函数仅仅是将 store 中的 **getter** 映射到**局部计算属性**：

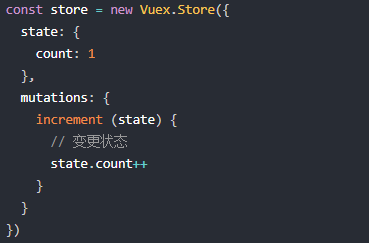


如果你想将一个 getter 属性另取一个名字，使用对象形式：



## Mutation

更改 Vuex 的 store 中的状态的**唯一方法**是**提交 mutation**。Vuex 中的 mutation 非常**类似于事件**：每个 mutation 都有一个字符串的 **事件类型 (type)** 和 一个 **回调函数 (handler)**。这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数：

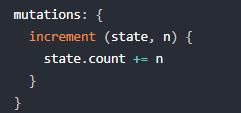


不能直接调用一个 mutation handler。这个选项更像是事件注册：“当触发一个类型为 increment 的 mutation 时，调用此函数。”要唤醒一个 mutation handler，你需要以相应的 type 调用 **store.commit** 方法：



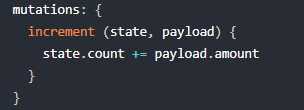
### 提交载荷（Payload）

可以向 store.commit 传入额外的参数，即 mutation 的 载荷（payload）：





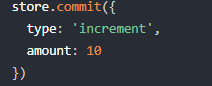
在大多数情况下，载荷应该是一个对象，这样可以包含多个字段并且记录的 mutation 会更易读：



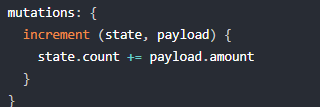


### 对象风格的提交方式

提交 mutation 的另一种方式是直接使用包含 type 属性的对象：



当使用对象风格的提交方式，整个对象都作为载荷传给 mutation 函数，因此 handler 保持不变：



### Mutation 需遵守 Vue 的响应规则

既然 Vuex 的 store 中的状态是响应式的，那么当我们变更状态时，监视状态的 Vue 组件也会自动更新。这也意味着 Vuex 中的 mutation 也需要与使用 Vue 一样遵守一些注意事项：

1. 最好提前在你的 store 中**初始化**好所有所需属性。
2. 当需要在对象上添加新属性时，你应该

* 使用 Vue.set(obj, ‘newProp’, 123)，或
* 以新对象替换老对象。



### 使用常量替代 Mutation 事件类型

使用常量替代 mutation 事件类型在各种 Flux 实现中是很常见的模式。这样可以使 linter 之类的工具发挥作用，同时把**这些常量放在单独的文件**中可以让你的代码合作者对整个 app 包含的 mutation 一目了然：

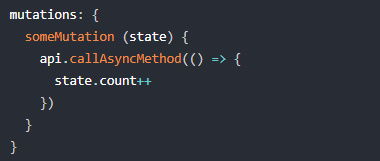




用不用常量取决于你——在需要多人协作的大型项目中，这会很有帮助。

### Mutation 必须是同步函数

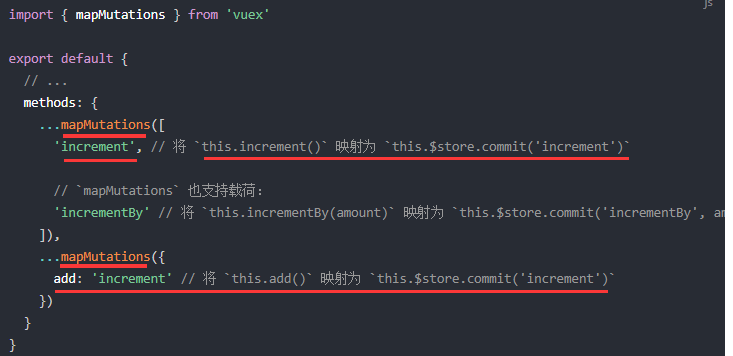
一条重要的原则就是要记住 **mutation 必须是同步函数**。为什么？请参考下面的例子：



现在想象，我们正在 debug 一个 app 并且观察 devtool 中的 mutation 日志。每一条 mutation 被记录，devtools 都需要捕捉到前一状态和后一状态的快照。然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的。

### 在组件中提交 Mutation

可以在组件中使用 this.$store.commit('xxx') 提交 mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为 store.commit 调用（需要在根节点注入 store）。



### 下一步：Action

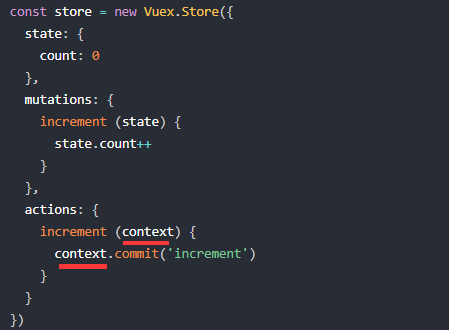
在 mutation 中混合异步调用会导致你的程序很难调试。例如，当你调用了两个包含异步回调的 mutation 来改变状态，你怎么知道什么时候回调和哪个先回调呢？这就是为什么我们要区分这两个概念。在 Vuex 中，**mutation 都是同步事务**：



## Action

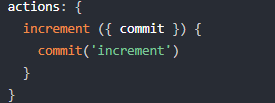
Action 类似于 mutation，不同在于：

* Action 提交的是 mutation，而不是直接变更状态。
* Action 可以包含任意异步操作



Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，因此你可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters。当我们在之后介绍到 [Modules](https://vuex.vuejs.org/zh/guide/modules.html) 时，你就知道 context 对象为什么不是 store 实例本身了。

实践中，我们会经常用到 ES2015 的 [参数解构](https://github.com/lukehoban/es6features#destructuring) 来简化代码（特别是我们需要调用 commit 很多次的时候）：

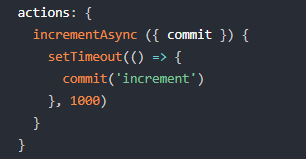


### 分发 Action

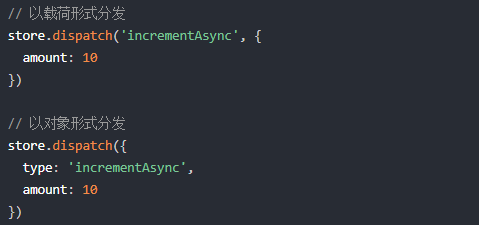
Action 通过 store.dispatch 方法触发：



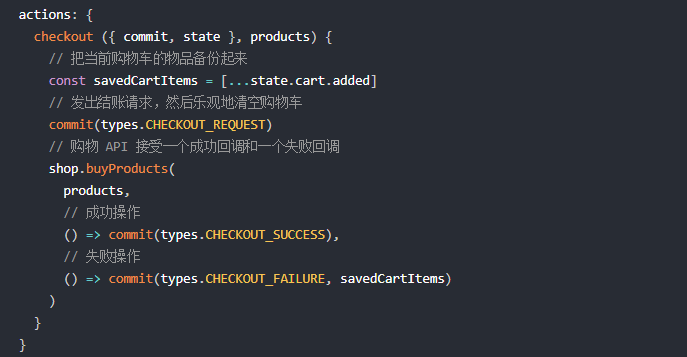
乍一眼看上去感觉多此一举，我们直接分发 mutation 岂不更方便？实际上并非如此，还记得 **mutation 必须同步执行**这个限制么？Action 就不受约束！我们可以在 action 内部执行**异步**操作：



Actions 支持同样的载荷方式和对象方式进行分发：



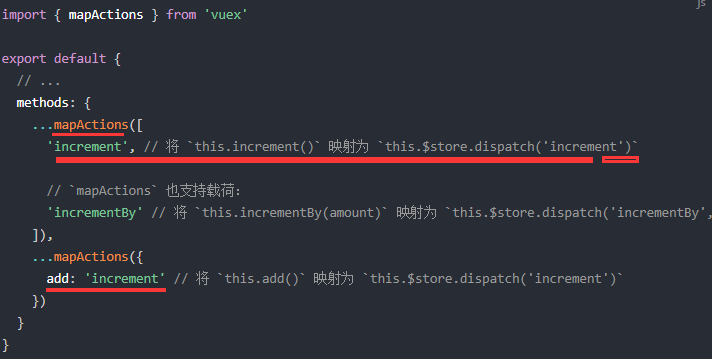
来看一个更加实际的购物车示例，涉及到**调用异步 API** 和**分发多重 mutation**：



注意我们正在进行一系列的异步操作，并且通过提交 mutation 来记录 action 产生的副作用（即状态变更）。

### 在组件中分发 Action

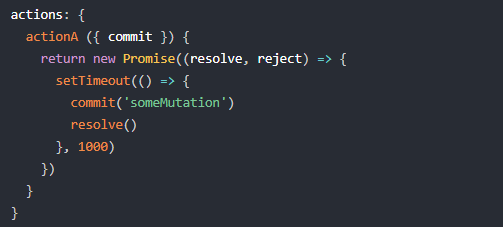
你在组件中使用 this.$store.dispatch('xxx') 分发 action，或者使用 mapActions 辅助函数将组件的 methods 映射为 store.dispatch 调用（需要先在根节点注入 store）：



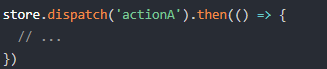
### 组合 Action

Action 通常是异步的，那么如何知道 action 什么时候结束呢？更重要的是，我们如何才能组合多个 action，以处理更加复杂的异步流程？

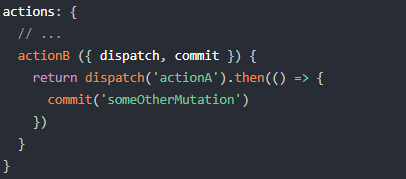
首先，你需要明白 store.dispatch 可以处理被触发的 action 的处理函数返回的 Promise，并且 store.dispatch 仍旧返回 Promise：



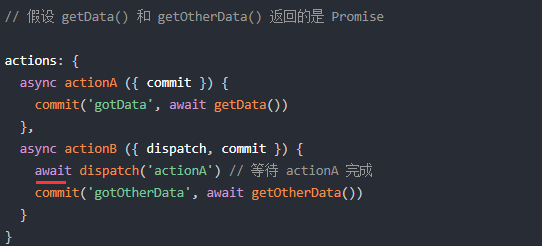
现在你可以：



在另外一个 action 中也可以：



最后，如果我们利用 [async / await](https://tc39.github.io/ecmascript-asyncawait/)，我们可以如下组合 action：

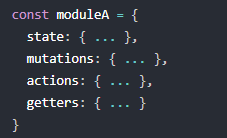
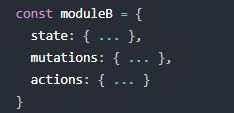
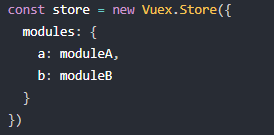


一个 store.dispatch 在不同模块中可以触发多个 action 函数。在这种情况下，只有当所有触发函数完成后，返回的 Promise 才会执行。

## Module

由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。

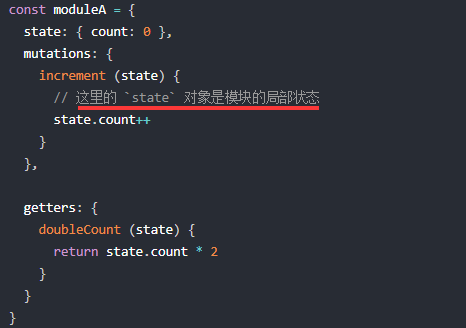
为了解决以上问题，Vuex 允许我们将 store 分割成**模块（module）**。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块——从上至下进行同样方式的分割：

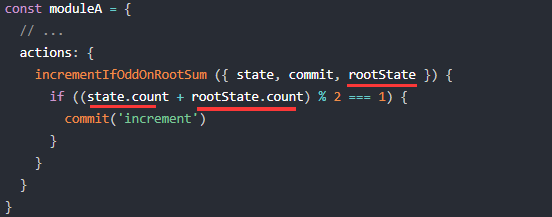


### 模块的局部状态

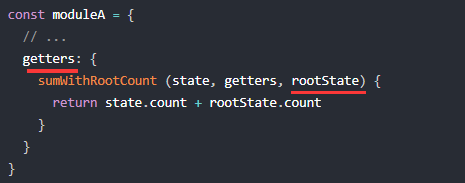
对于模块内部的 mutation 和 getter，接收的第一个参数是**模块的局部状态对象**。



同样，对于模块内部的 action，局部状态通过 context.state 暴露出来，根节点状态则为 context.rootState：



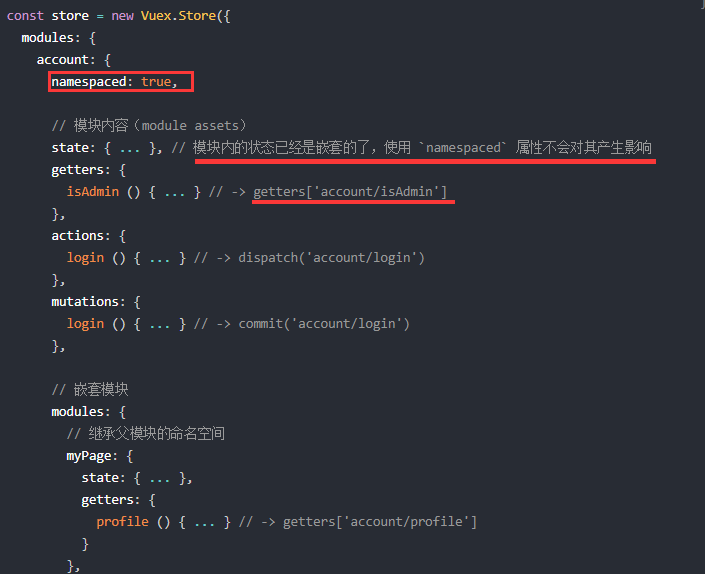
对于模块内部的 getter，**根节点状态**会作为**第三个参数**暴露出来：



### 命名空间

默认情况下，模块内部的 action、mutation 和 getter 是注册在**全局命名空间**的——这样使得多个模块能够对同一 mutation 或 action 作出响应。

如果希望你的模块具有更高的封装度和复用性，你可以通过添加 namespaced: true 的方式使其成为带命名空间的模块。当模块被注册后，它的所有 getter、action 及 mutation 都会自动根据模块注册的路径调整命名。例如：



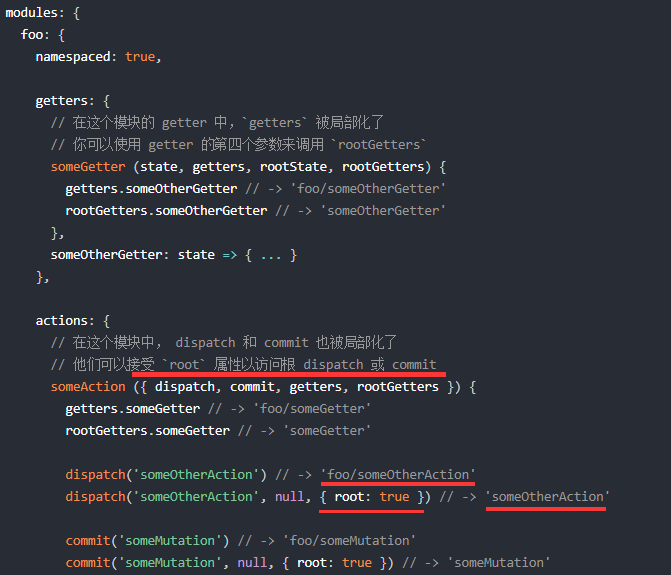
启用了命名空间的 getter 和 action 会收到局部化的 getter，dispatch 和 commit。

换言之，你在使用模块内容（module assets）时不需要在同一模块内额外添加空间名前缀。更改 namespaced 属性后不需要修改模块内的代码。

### 在带命名空间的模块内访问全局内容（Global Assets）

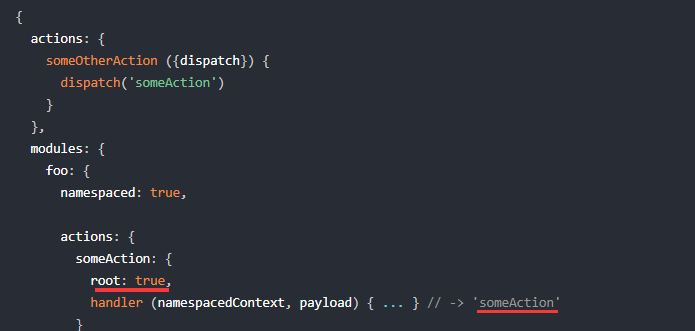
如果你希望使用全局 state 和 getter，**rootState** 和 **rootGetter** 会作为第三和第四参数传入 getter，也会通过 context 对象的属性传入 action。

若需要在全局命名空间内分发 action 或提交 mutation，将 { root: true } 作为第三参数传给 dispatch 或 commit 即可。



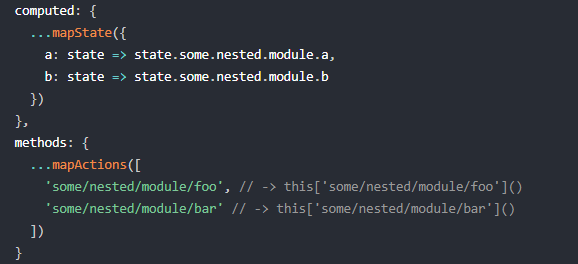
### 在带命名空间的模块注册全局 action

若需要在带命名空间的模块注册全局 action，你可添加 root: true，并将这个 action 的定义放在函数 handler 中。例如：

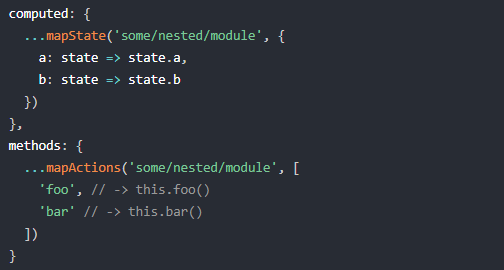


### 带命名空间的绑定函数

当使用 mapState, mapGetters, mapActions 和 mapMutations 这些函数来绑定带命名空间的模块时，写起来可能比较繁琐：



对于这种情况，你可以**将模块的空间名称字符串作为第一个参数**传递给上述函数，这样所有绑定都会自动将该模块作为上下文。于是上面的例子可以简化为：

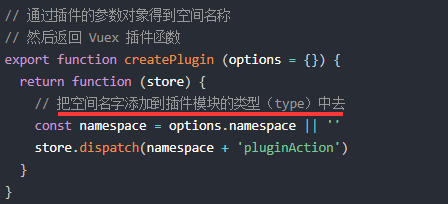


而且，你可以通过使用 createNamespacedHelpers 创建**基于某个命名空间辅助函数**。它返回一个对象，对象里有新的绑定在给定命名空间值上的组件绑定辅助函数：



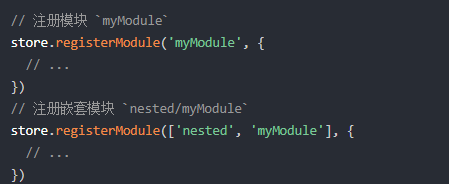
### 给插件开发者的注意事项

如果你开发的[插件（Plugin）](https://vuex.vuejs.org/zh/guide/plugins.html)提供了模块并允许用户将其添加到 Vuex store，可能需要考虑模块的空间名称问题。对于这种情况，你可以通过插件的参数对象来允许用户指定空间名称：



### 模块动态注册

在 store 创建之后，你可以使用 store.registerModule 方法注册模块：



之后就可以通过 store.state.myModule 和 store.state.nested.myModule 访问模块的状态。

模块动态注册功能使得其他 Vue 插件可以通过在 store 中附加新模块的方式来使用 Vuex 管理状态。例如，[vuex-router-sync](https://github.com/vuejs/vuex-router-sync" \t "_blank) 插件就是通过动态注册模块将 vue-router 和 vuex 结合在一起，实现应用的路由状态管理。

也可以使用 store.unregisterModule(moduleName) 来动态卸载模块。注意，你不能使用此方法卸载静态模块（即创建 store 时声明的模块）

在注册一个新 module 时，你很有可能想保留过去的 state，例如从一个服务端渲染的应用保留 state。你可以通过 preserveState 选项将其归档：store.registerModule('a', module, { preserveState: true })。

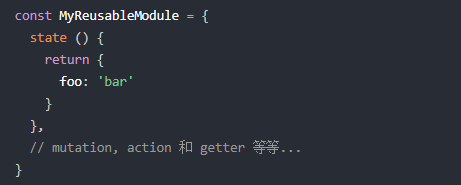
### 模块重用

有时我们可能需要创建一个模块的多个实例，例如：

* 创建多个 store，他们公用同一个模块 (例如当 runInNewContext 选项是 false 或 'once' 时，为了[在服务端渲染中避免有状态的单例](https://ssr.vuejs.org/en/structure.html" \l "avoid-stateful-singletons" \t "_blank))
* 在一个 store 中多次注册同一个模块

如果我们使用一个纯对象来声明模块的状态，那么这个状态对象会通过引用被共享，导致状态对象被修改时 store 或模块间数据互相污染的问题。

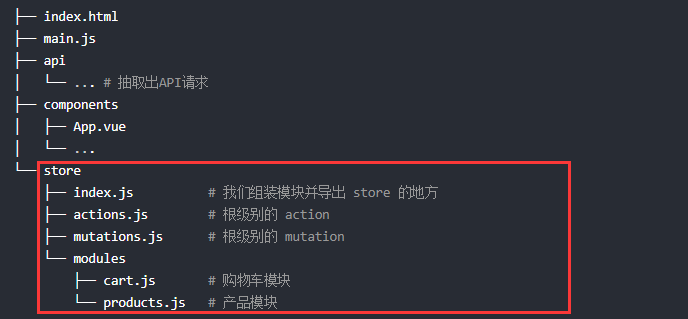
实际上这和 Vue 组件内的 data 是同样的问题。因此解决办法也是相同的——**使用一个函数来声明模块状态**（仅 2.3.0+ 支持）：



## 项目结构

* 应用层级的状态应该集中到单个 store 对象中。
* 提交 mutation 是更改状态的唯一方法，并且这个过程是同步的。
* 异步逻辑都应该封装到 action 里面

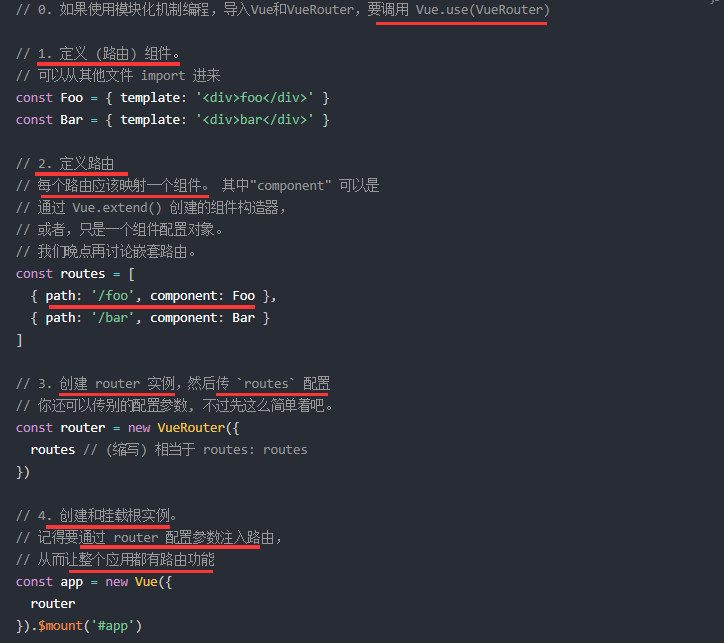
对于大型应用，我们会希望把 Vuex 相关代码分割到模块中。下面是项目结构示例：



# vue-router

当你要把 Vue Router 添加进来，我们需要做的是，将组件 (components) 映射到路由 (routes)，然后告诉 Vue Router 在哪里渲染它们。





通过注入路由器，我们可以在任何组件内通过 **this.$router** 访问路由器，也可以通过**this.$route** 访问当前路由：



留意一下 this.$router 和 router 使用起来完全一样。我们使用 this.$router 的原因是我们并不想在每个独立需要封装路由的组件中都导入路由。

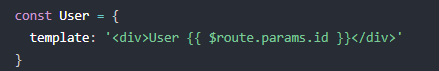
要注意，当 <router-link> 对应的路由匹配成功，将自动设置 class 属性值 .router-link-active。

## 动态路由匹配

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件。例如，我们有一个 User 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 vue-router 的路由路径中使用“动态路径参数”(dynamic segment) 来达到这个效果：



一个“路径参数”使用冒号 : 标记。当匹配到一个路由时，参数值会被设置到 this.$route.params，可以在每个组件内使用。于是，我们可以更新 User 的模板，输出当前用户的 ID：



可以在一个路由中设置多段“路径参数”，对应的值都会设置到 $route.params 中。例如：

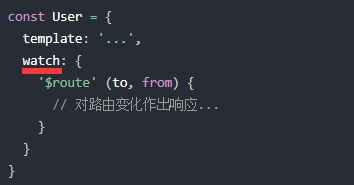


除了 $route.params 外，$route 对象还提供了其它有用的信息，例如，$route.query (如果 URL 中有查询参数)、$route.hash 等等。你可以查看 [API 文档](https://router.vuejs.org/zh/api/#%E8%B7%AF%E7%94%B1%E5%AF%B9%E8%B1%A1) 的详细说明。

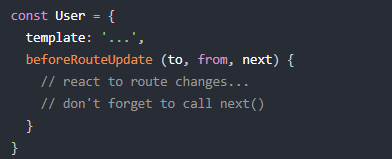
### 响应路由参数的变化

当使用路由参数时，例如从 /user/foo 导航到 /user/bar，**原来的组件实例会被复用**。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。**不过，这也意味着组件的生命周期钩子不会再被调用**。

复用组件时，想对路由参数的变化作出响应的话，你可以简单地 watch (监测变化) $route 对象：



或者使用 2.2 中引入的 beforeRouteUpdate [导航守卫](https://router.vuejs.org/zh/guide/advanced/navigation-guards.html)：



### 高级匹配模式

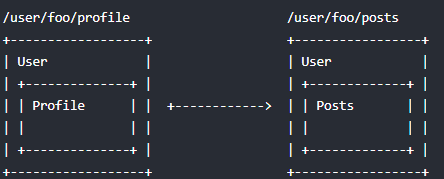
vue-router 使用 [path-to-regexp](https://github.com/pillarjs/path-to-regexp) 作为路径匹配引擎，所以支持很多高级的匹配模式，例如：可选的动态路径参数、匹配零个或多个、一个或多个，甚至是自定义正则匹配。查看它的 [文档](https://github.com/pillarjs/path-to-regexp#parameters) 学习高阶的路径匹配，还有 [这个例子](https://github.com/vuejs/vue-router/blob/next/examples/route-matching/app.js) 展示 vue-router 怎么使用这类匹配。

### 匹配优先级

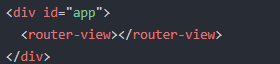
有时候，同一个路径可以匹配多个路由，此时，匹配的优先级就按照路由的定义顺序：谁先定义的，谁的优先级就最高。

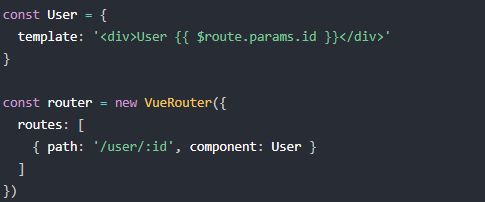
## 嵌套路由

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各层组件，例如：

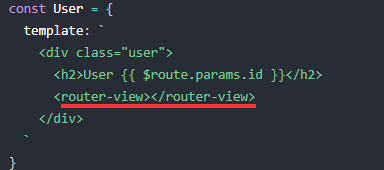


借助 vue-router，使用嵌套路由配置，就可以很简单地表达这种关系。

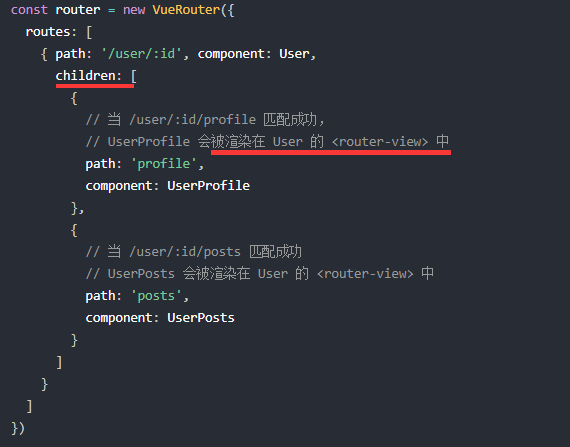




这里的 <router-view> 是最顶层的出口，**渲染最高级路由匹配到的组件**。同样地，**一个被渲染组件同样可以包含自己的嵌套 <router-view>**。例如，在 User 组件的模板添加一个 <router-view>：



要在嵌套的出口中渲染组件，需要在 VueRouter 的参数中使用 children 配置：



**要注意，以 / 开头的嵌套路径会被当作根路径。 这让你充分的使用嵌套组件而无须设置嵌套的路径。**

你会发现，children 配置就是像 routes 配置一样的路由配置数组，所以呢，你可以嵌套多层路由。

此时，基于上面的配置，当你访问 /user/foo 时，User 的出口是不会渲染任何东西，这是因为没有匹配到合适的子路由。如果你想要渲染点什么，可以提供一个 空的 子路由：



## 编程式的导航

除了使用 <router-link> 创建 a 标签来定义导航链接，我们还可以借助 router 的实例方法，通过**编写代码**来实现。

### router.push



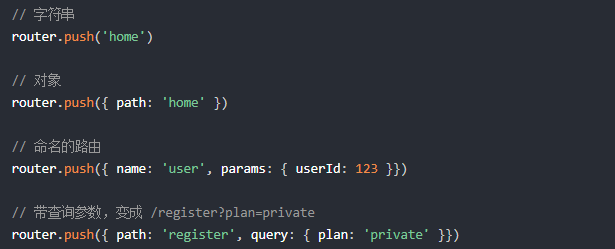
**注意：在 Vue 实例内部，你可以通过 $router 访问路由实例。因此你可以调用 this.$router.push。**

想要导航到不同的 URL，则使用 router.push 方法。这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，则回到之前的 URL。

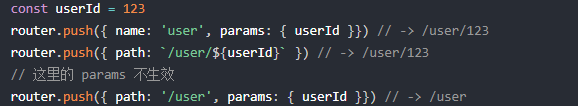
当你点击 <router-link> 时，这个方法会在内部调用，所以说，点击 <router-link :to="..."> 等同于调用 router.push(...)。



该方法的参数可以是一个字符串路径，或者一个描述地址的对象。例如：



**注意：如果提供了 path，params 会被忽略，上述例子中的 query 并不属于这种情况。取而代之的是下面例子的做法，你需要提供路由的 name 或手写完整的带有参数的 path：**



同样的规则也适用于 router-link 组件的 to 属性。

\*\*注意：\*\*如果**目的地和当前路由相同**，只有参数发生了改变 (比如从一个用户资料到另一个 /users/1 -> /users/2)，你需要使用 [beforeRouteUpdate](https://router.vuejs.org/zh/guide/essentials/dynamic-matching.html" \l "%E5%93%8D%E5%BA%94%E8%B7%AF%E7%94%B1%E5%8F%82%E6%95%B0%E7%9A%84%E5%8F%98%E5%8C%96) 来响应这个变化 (比如抓取用户信息)。

### router.replace



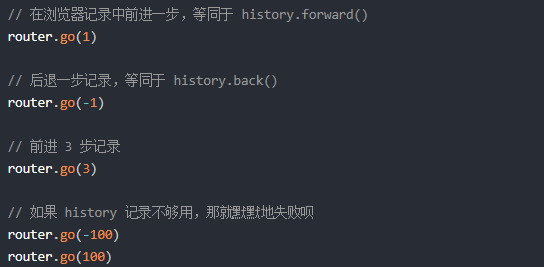
跟 router.push 很像，唯一的不同就是，它不会向 history 添加新记录，而是跟它的方法名一样 —— 替换掉当前的 history 记录。



### router.go(n)



这个方法的参数是一个整数，意思是在 history 记录中向前或者后退多少步，类似 window.history.go(n)。



### 操作 History

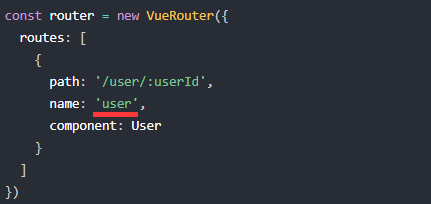
 router.push、 router.replace 和 router.go 跟 [window.history.pushState、 window.history.replaceState 和 window.history.go](https://developer.mozilla.org/en-US/docs/Web/API/History" \t "_blank)好像， 实际上它们确实是效仿 window.historyAPI 的。

因此，如果你已经熟悉 [Browser History APIs](https://developer.mozilla.org/en-US/docs/Web/API/History_API)，那么在 Vue Router 中操作 history 就是超级简单的。

还有值得提及的，Vue Router 的导航方法 (push、 replace、 go) 在各类路由模式 (history、 hash 和 abstract) 下表现一致。

## 命名路由

有时候，通过一个名称来标识一个路由显得更方便一些，特别是在链接一个路由，或者是执行一些跳转的时候。你可以在创建 Router 实例的时候，在 routes 配置中给某个路由设置名称。



要链接到一个命名路由，可以给 router-link 的 to 属性传一个对象：



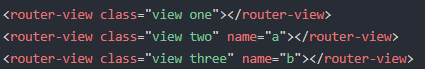
这跟代码调用 router.push() 是一回事：



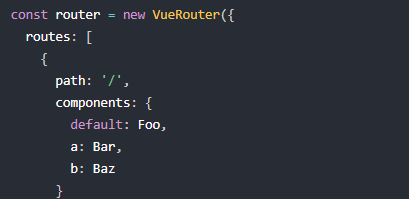
## 命名视图

有时候想同时 (同级) 展示多个视图，而不是嵌套展示，例如创建一个布局，有 sidebar (侧导航) 和 main (主内容) 两个视图，这个时候命名视图就派上用场了。

你可以在界面中拥有**多个单独命名的视图**，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。

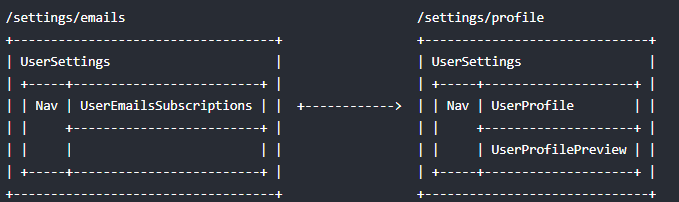


一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components配置 (带上 s)：



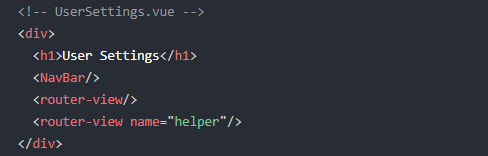
### 嵌套命名视图

我们也有可能使用命名视图创建嵌套视图的复杂布局。这时你也需要命名用到的嵌套 router-view 组件。我们以一个设置面板为例：





UserSettings 组件的 <template> 部分应该是类似下面的这段代码：



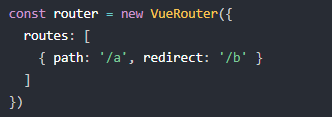
然后你可以用这个路由配置完成该布局：



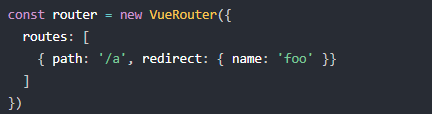
## 重定向和别名

### 重定向

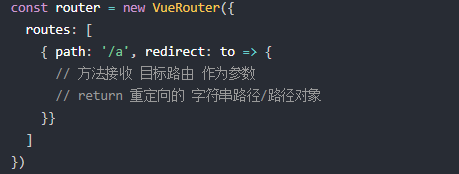
重定向也是通过 routes 配置来完成，下面例子是从 /a 重定向到 /b：



重定向的目标也可以是一个命名的路由：



甚至是一个方法，动态返回重定向目标：

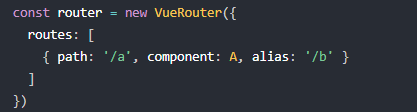


注意[导航守卫](https://router.vuejs.org/zh/guide/advanced/navigation-guards.html)并没有应用在跳转路由上，而仅仅应用在其目标上。在下面这个例子中，为 /a 路由添加一个 beforeEach 或 beforeLeave 守卫并不会有任何效果。

### 别名

“重定向”的意思是，当用户访问 /a时，URL 将会被替换成 /b，然后匹配路由为 /b，那么“别名”又是什么呢？

**/a 的别名是 /b，意味着，当用户访问 /b 时，URL 会保持为 /b，但是路由匹配则为 /a，就像用户访问 /a 一样。**



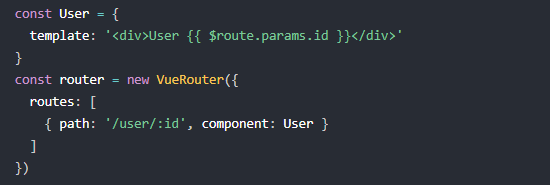
“别名”的功能让你可以自由地将 UI 结构映射到任意的 URL，而不是受限于配置的嵌套路由结构。

## 路由组件传参

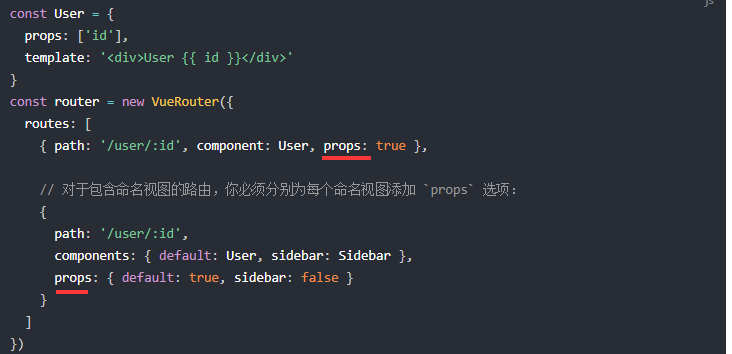
在组件中使用 $route 会使之与其对应路由形成高度耦合，从而使组件只能在某些特定的 URL 上使用，限制了其灵活性。

使用 props 将组件和路由解耦：

**取代与 $route 的耦合**



**通过 props 解耦**



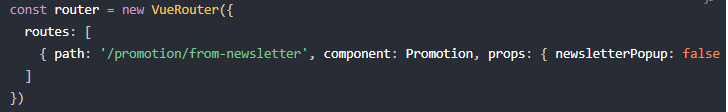
这样你便可以在任何地方使用该组件，使得该组件更易于重用和测试。

### 布尔模式

如果 props 被设置为 true，route.params 将会被设置为组件属性

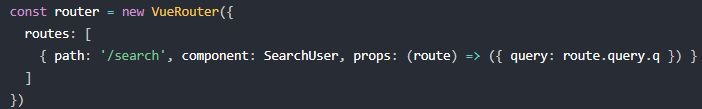
### 对象模式

如果 props 是一个对象，它会被按原样设置为组件属性。当 props 是静态的时候有用。



### 函数模式

可以创建一个函数返回 props。这样你便可以将参数转换成另一种类型，将静态值与基于路由的值结合等等。

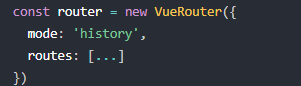


尽可能保持 props 函数为无状态的，因为它只会在路由发生变化时起作用。如果你需要状态来定义 props，请使用包装组件，这样 Vue 才可以对状态变化做出反应。

## HTML5 History 模式

vue-router 默认 hash 模式 —— 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。

如果不想要很丑的 hash，我们可以用路由的 **history 模式**，这种模式充分利用 history.pushState API 来完成 URL 跳转而无须重新加载页面。

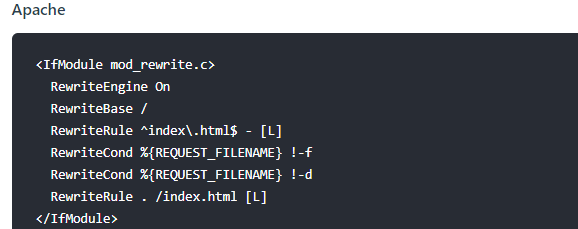


当你使用 history 模式时，URL 就像正常的 url，例如 http://yoursite.com/user/id，也好看！

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 http://oursite.com/user/id 就会返回 404，这就不好看了。

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

### 后端配置例子



除了 mod\_rewrite，你也可以使用 [FallbackResource](https://httpd.apache.org/docs/2.2/mod/mod_dir.html" \l "fallbackresource" \t "_blank)

## 导航守卫

“导航”表示路由正在发生改变。

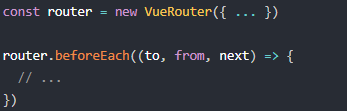
vue-router 提供的导航守卫主要用来通过跳转或取消的方式守卫导航。

有多种机会植入路由导航过程中：全局的, 单个路由独享的, 或者组件级的。

记住**参数或查询的改变并不会触发进入/离开的导航守卫**。你可以通过[观察 $route 对象](https://router.vuejs.org/zh/guide/essentials/dynamic-matching.html#%E5%93%8D%E5%BA%94%E8%B7%AF%E7%94%B1%E5%8F%82%E6%95%B0%E7%9A%84%E5%8F%98%E5%8C%96)来应对这些变化，或使用 beforeRouteUpdate 的组件内守卫。

### 全局守卫

可以使用 router.beforeEach 注册一个全局前置守卫：



当一个导航触发时，全局前置守卫按照创建顺序调用。守卫是异步解析执行，此时导航在所有守卫 resolve 完之前一直处于 **等待中**。

每个守卫方法接收三个参数：

**to: Route**: 即将要进入的目标 [路由对象](https://router.vuejs.org/zh/api/#%E8%B7%AF%E7%94%B1%E5%AF%B9%E8%B1%A1)

**from: Route**: 当前导航正要离开的路由

**next: Function**: 一定要调用该方法来 **resolve** 这个钩子。执行效果依赖 next 方法的调用参数。

* next(): 进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 confirmed (确认的)。
* next(false): 中断当前的导航。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 from 路由对应的地址。
* next('/') 或者 next({ path: '/' }): 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。你可以向 next 传递任意位置对象，且允许设置诸如 replace: true、name: 'home' 之类的选项以及任何用在 router-link 的 to prop 或 router.push 中的选项。
* next(error): (2.4.0+) 如果传入 next 的参数是一个 Error 实例，则导航会被终止且该错误会被传递给 router.onError() 注册过的回调。

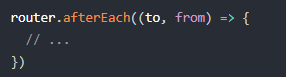
**确保要调用 next 方法，否则钩子就不会被 resolved。**

### 全局解析守卫

可以用 router.beforeResolve 注册一个全局守卫。这和 router.beforeEach 类似，区别是在导航被确认之前，**同时在所有组件内守卫和异步路由组件被解析之后**，解析守卫就被调用。

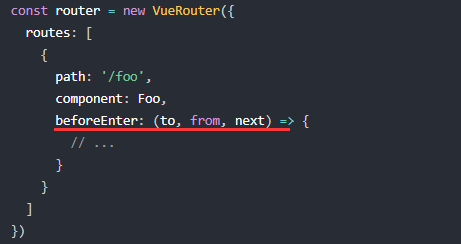
### 全局后置钩子

也可以注册全局后置钩子，然而和守卫不同的是，这些钩子不会接受 next 函数也不会改变导航本身



### 路由独享的守卫

可以在路由配置上直接定义 beforeEnter 守卫：



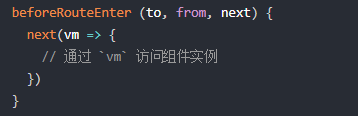
### 组件内的守卫

最后，你可以在路由组件内直接定义以下路由导航守卫：

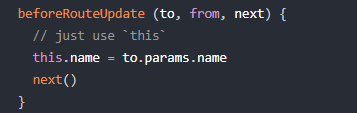


beforeRouteEnter 守卫 **不能** 访问 this，因为守卫在导航确认前被调用,因此即将登场的新组件还没被创建。

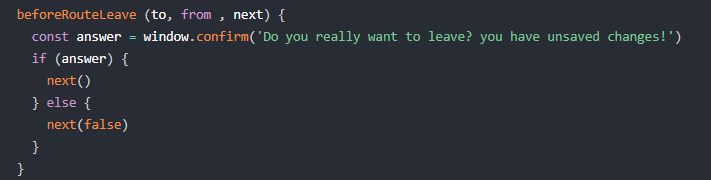
不过，你可以通过传一个回调给 next来访问组件实例。在导航被确认的时候执行回调，并且把组件实例作为回调方法的参数。



注意 beforeRouteEnter 是支持给 next 传递回调的唯一守卫。对于 beforeRouteUpdate 和 beforeRouteLeave 来说，this 已经可用了，所以**不支持**传递回调，因为没有必要了。



这个离开守卫通常用来禁止用户在还未保存修改前突然离开。该导航可以通过 next(false) 来取消。

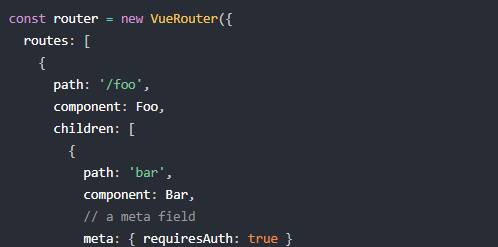


### 完整的导航解析流程



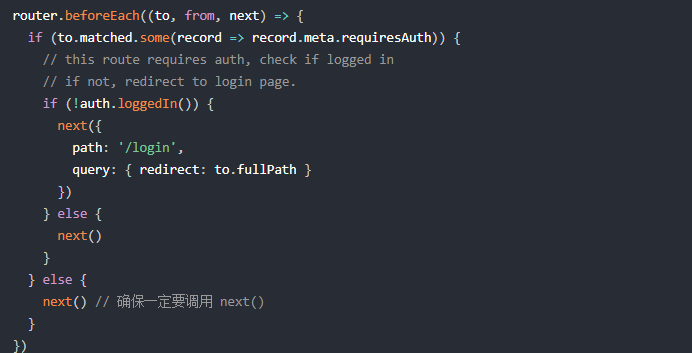
## 路由元信息

定义路由的时候可以配置 meta 字段：



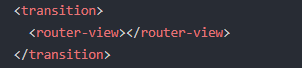
首先，我们称呼 routes 配置中的每个路由对象为 **路由记录**。路由记录可以是嵌套的，因此，当一个路由匹配成功后，他可能匹配多个路由记录

一个路由匹配到的所有路由记录会暴露为 $route 对象 (还有在导航守卫中的路由对象) 的 $route.matched 数组。因此，我们需要遍历 $route.matched 来检查路由记录中的 meta 字段。



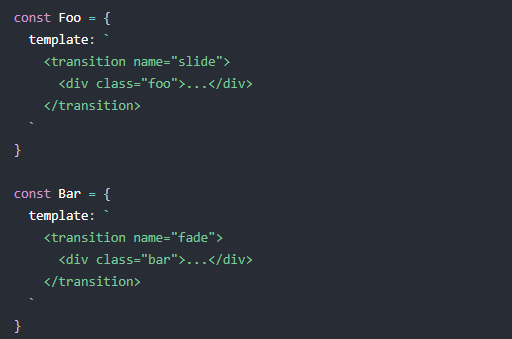
## 过渡动效

<router-view> 是基本的动态组件，所以我们可以用 <transition> 组件给它添加一些过渡效果：



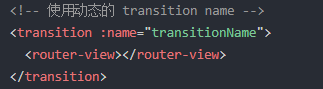
### 单个路由的过渡

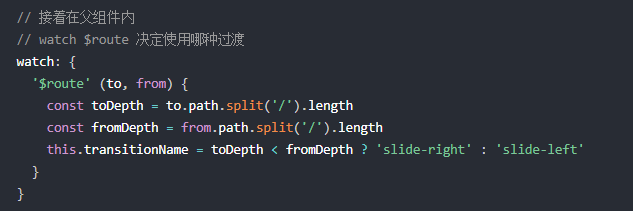
上面的用法会给所有路由设置一样的过渡效果，如果你想让每个路由组件有各自的过渡效果，可以在各路由组件内使用 <transition> 并设置不同的 name。



### 基于路由的动态过渡

还可以基于当前路由与目标路由的变化关系，动态设置过渡效果





## 数据获取

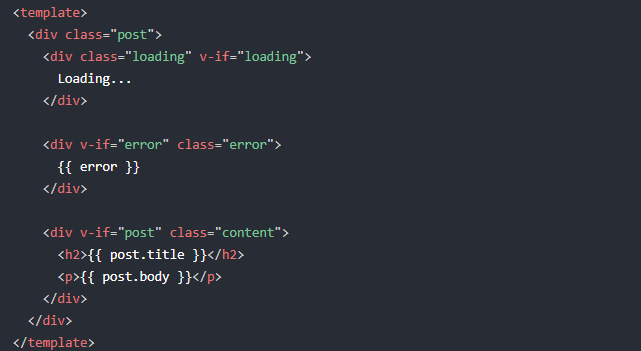
有时候，进入某个路由后，需要从服务器获取数据。例如，在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现：

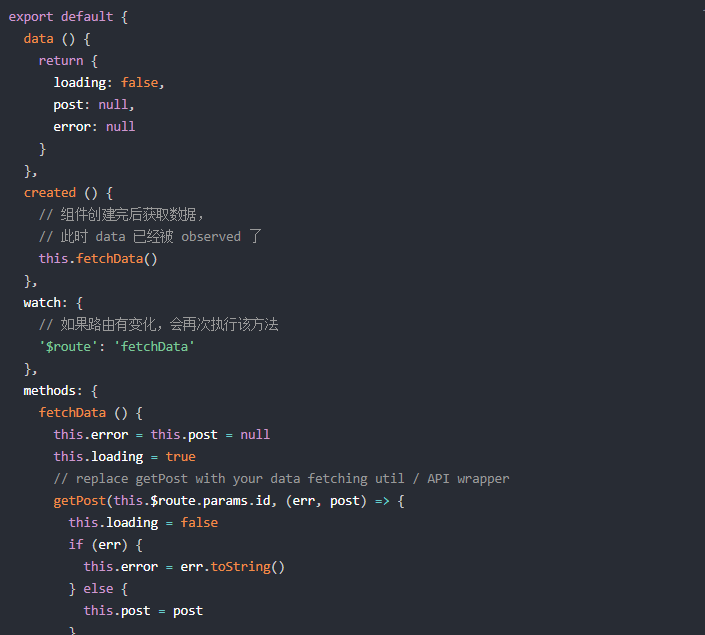
* 导航完成之后获取：先完成导航，然后在接下来的组件生命周期钩子中获取数据。在数据获取期间显示“加载中”之类的指示。
* 导航完成之前获取：导航完成前，在路由进入的守卫中获取数据，在数据获取成功后执行导航。

### 导航完成后获取数据

当你使用这种方式时，我们会马上导航和渲染组件，然后在组件的 created 钩子中获取数据。这让我们有机会在数据获取期间展示一个 loading 状态，还可以在不同视图间展示不同的 loading 状态。

假设我们有一个 Post 组件，需要基于 $route.params.id 获取文章数据：





### 在导航完成前获取数据

通过这种方式，我们在导航转入新的路由前获取数据。我们可以在接下来的组件的 beforeRouteEnter 守卫中获取数据，当数据获取成功后只调用 next 方法。



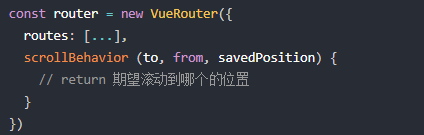
在为后面的视图获取数据时，用户会停留在当前的界面，因此建议在数据获取期间，显示一些进度条或者别的指示。如果数据获取失败，同样有必要展示一些全局的错误提醒。

## 滚动行为

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。 vue-router 能做到，而且更好，它让你可以自定义路由切换时页面如何滚动。

**注意: 这个功能只在支持 history.pushState 的浏览器中可用。**

当创建一个 Router 实例，你可以提供一个 scrollBehavior 方法：

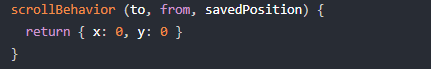


scrollBehavior 方法接收 to 和 from 路由对象。第三个参数 savedPosition 当且仅当 popstate导航 (通过浏览器的 前进/后退 按钮触发) 时才可用。

这个方法返回滚动位置的对象信息，长这样：

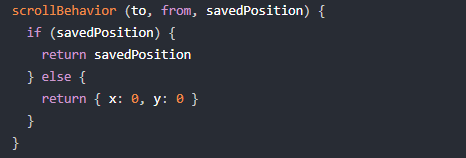


如果返回一个 falsy (译者注：falsy 不是 false，[参考这里](https://developer.mozilla.org/zh-CN/docs/Glossary/Falsy" \t "_blank))的值，或者是一个空对象，那么不会发生滚动。

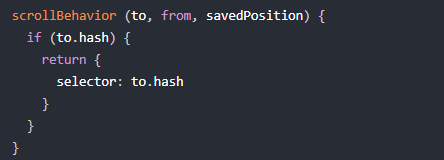


对于所有路由导航，简单地让页面滚动到顶部。

返回 savedPosition，在按下 后退/前进 按钮时，就会像浏览器的原生表现那样：



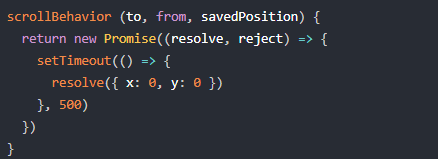
如果你要模拟“滚动到锚点”的行为：



我们还可以利用[路由元信息](https://router.vuejs.org/zh/guide/advanced/meta.html)更细颗粒度地控制滚动。查看完整例子请[移步这里](https://github.com/vuejs/vue-router/blob/next/examples/scroll-behavior/app.js)。

### 异步滚动

你也可以返回一个 Promise 来得出预期的位置描述



将其挂载到从页面级别的过渡组件的事件上，令其滚动行为和页面过渡一起良好运行是可能的。但是考虑到用例的多样性和复杂性，我们仅提供这个原始的接口，以支持不同用户场景的具体实现。

## 路由懒加载

当打包构建应用时，Javascript 包会变得非常大，影响页面加载。如果我们能**把不同路由对应的组件分割成不同的代码块**，**然后当路由被访问的时候才加载对应组件**，这样就更加高效了。

结合 Vue 的[异步组件](https://cn.vuejs.org/v2/guide/components-dynamic-async.html#%E5%BC%82%E6%AD%A5%E7%BB%84%E4%BB%B6)和 Webpack 的[代码分割功能](https://doc.webpack-china.org/guides/code-splitting-async/#require-ensure-/)，轻松实现路由组件的懒加载。

首先，可以将异步组件定义为返回一个 Promise 的工厂函数 (该函数返回的 Promise 应该 resolve 组件本身)：



第二，在 Webpack 2 中，我们可以使用[动态 import](https://github.com/tc39/proposal-dynamic-import)语法来定义代码分块点 (split point)：

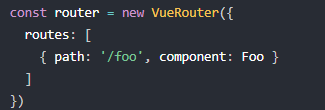


如果您使用的是 Babel，你将需要添加 [syntax-dynamic-import](https://babeljs.io/docs/plugins/syntax-dynamic-import/) 插件，才能使 Babel 可以正确地解析语法。

结合这两者，这就是如何定义一个能够被 Webpack 自动代码分割的异步组件。



在路由配置中什么都不需要改变，只需要像往常一样使用 Foo：



### 把组件按组分块

有时候我们想把某个路由下的所有组件都打包在同个异步块 (chunk) 中。只需要使用 [命名 chunk](https://webpack.js.org/guides/code-splitting-require/#chunkname)，一个特殊的注释语法来提供 chunk name (需要 Webpack > 2.4)。

