Vuex

```
// Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。
npm i vuex -S
src新建store目录
  store
    index.js
      import Vue from 'vue'
      import Vuex from 'vuex'

      Vue.use(Vuex)
      const store = new Vuex.Store({
        state: {
          num: 10
        },
        mutations: {
          addNum (state) {
            state.num++
          }
        }
      })
      export default store
  main.js
  import store from './store'
  new Vue({
    store
  })

  // 组件中使用
  Vue.use(Vuex)后
  自动在所有的组件中 通过this.$store获取创建的仓库

  this.$store.state.xxx 拿到数据
  // 组件中如何修改数据
  this.$store.commit('mutation名字)
  // 传参  只能传一个参数 可以在mutation第二个参数中接收到
  this.$store.commit('mutation'[,参数])
```

```
// vuex提供了 mapState mapMutations 等 助手函数 方便 获取 和 修改数据
import { mapState } from 'vuex'

{
  computed: {
    ...mapState(['num'])
  }
}
//会自动 新增计算属性 num 依赖 store中state中的num
```

什么是状态管理模式

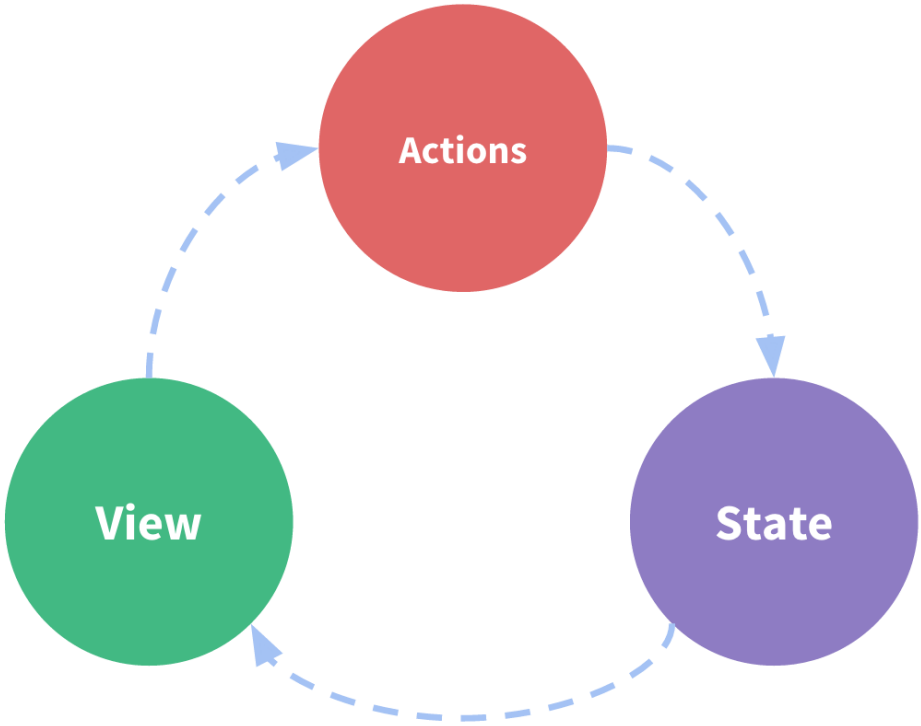
让我们从一个简单的Vue 计数应用开始：

```
new Vue({
  // state
  data () {
    return {
      count: 0
    }
  },
  // view
  template: `
    <div>{{ count }}</div>
  `,
  // actions
  methods: {
    increment () {
      this.count++
    }
  }
})
```

这个状态自管理应用包含以下几个部分：

- state，驱动应用的数据源；
- view，以声明方式将 state 映射到视图；
- actions，响应在 view 上的用户输入导致的状态变化。

以下是一个表示"单向数据流"理念的极简示意：



简单的抽象成一个简单数据流概念，视图View是由State数据驱动的，视图上可以响应数据的应用交互，比如派发一些Actions，Actions可以对数据进行一些修改，单向数据流的概念图

但是，当我们的应用遇到多个组件共享状态时，单项数据流的简洁性很容易被破坏

- 多个视图依赖同一状态
- 来自不同视图的行为需要变更同一状态

对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。
对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。

因此，我们为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？在这种模式下，我们的组件树构成了一个巨大的“视图”，不管在树的哪个位置，任何组件都能获取状态或者触发行为。

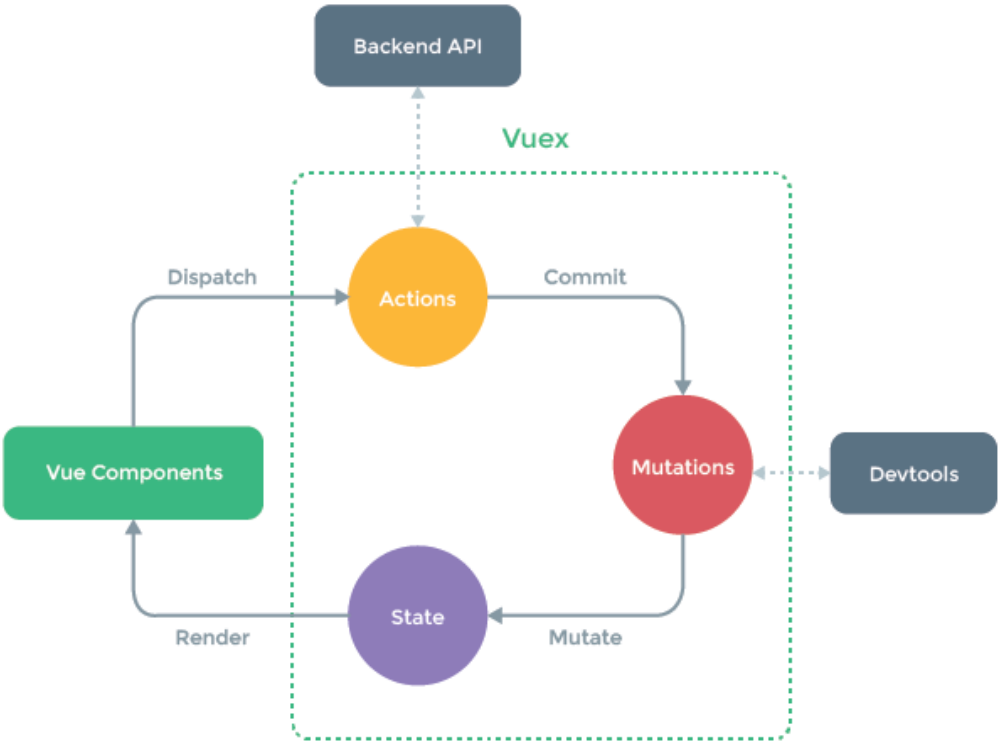
简单来说就是
比如兄弟组件之间需要共享同一块数据，或者是操作这块数据的话，是很难实现这样的应用的。我们通常都是通过派发事件的方式来解决这样的问题，但是如果我们的组件非常多非常复杂的话，那么如果到处派发事件、监听事件会让代码变得非常难以维护。
Vue如何解决这样的问题呢？可以把State从组件中抽离出来，放在一个专门的地方维护这个State。本来这个是组件级别的数据，抽离出去就变成了应用级别的数据，那我们就可以在多个组件中共享这个数据了。

Vuex核心思想
Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。有些同学可能会问，那我定义一个全局对象，再去上层封装了一些数据存取的接口不也可以么？

Vuex 和单纯的全局对象有以下两点不同：

- Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新，这个过程是自动化的，如果说继续写数仓，写全局对象的话，需要去调用方法，Vuex是自动化的，非常方便。
- 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。

另外，通过定义和隔离状态管理中的各种概念并强制遵守一定的规则，我们的代码将会变得更结构化且易维护。



Vuex存储的状态可以很好的映射到各个组件(Vue Components)中，组件都可以去Dispatch一个Action，去修改我们的数据，Actions也可以提交Commit一个Mutations去修改我们的State，然后Action也可以跟我们的后台API做一些交互，并且有个Devtools开发者工具可以帮助我们监听所有的变化。
但是也并非是所有的数据都是放在Vuex中，如果这个数据是和这个组件相关的话，那么就可以定义一些prop去自管理这些数据，全局共享的数据放在Vuex中

Vuex 初始化

我们主要来分析 Vuex 的初始化过程，它包括安装、Store 实例化过程 2 个方面。

使用Vuex解析

```
import Vue from 'vue'
// 1. 第一步需要import
import Vuex from 'vuex'
// 2. 然后使用use对Vuex进行一个注册
Vue.use(Vuex)

// 3. 然后我们定义一些配置，比如一个module，一个module包含state、mutations、actions、
getters
const moduleA = {
    namespaced:true,
    state:{
        count:1
    },
    mutations:{
        increment(state){
            state.count++
        }
    }
```

```
    actions:{
        increment(context){
            context.commit('increment')
        }
    },
    getters:{
        computedCount(state){
            return state.count + 1
        }
    }
}
const moduleB = {...}

// 4. Vuex暴露Store一个对象，然后去实例化它，实例化的时候我们可以传入modules，也可以传入state，或者是getter等等
const store = new Vuex.Store({
    modules:{
        a:moduleA,
        b:moduleB
    },
    state:{
        count:1
    }
})

// 5. 然后我们将实例化的store作为new Vue的一个参数
const app = new Vue({
    el:'#app',
    store
})
```

Vuex actions

action是vue中的方法，不能直接修改state,提交mutation,让mutation修改问题？为什么要用action。

为什么不直接提交mutation

原因：

mutation中不能包含异步操作

action可以包含异步操作

什么时候用action:

公共的数据 需要请求（ajax）接口,得到 mutation 不能有异步操作，

所以应该在action 中 发送ajax请求，请求成功后，提交 mutation,把请求的值传过去

```
{
  state,
  mutations,
  actions: {
    addNumAsync (context[,params]) {
      // context
      setTimeout(()=>{ // 模拟ajax请求
        const value = 100;
        context.commit('addNum', value)
        // 提交一个mutation 同时传值
      },2000)
    }
  }
}
// 如何在组件中 触发 action
```

```
this.$store.dispatch('action名字', params)
// 利用助手函数
import { mapActions } from 'vuex'

{
  methods: {
    ...mapActions(['action名字'])
    // 调用时 直接 this.action名字([参数])
  }
}
```

Vuex getters
相当于 Vuex中的计算属性

```
store
{
  getters: {
    num2 (state) {
      return state.num*2
    }
  }
}
// 组件中获取
this.$store.getters.num2
// 助手函数 mapGetters
import { mapGetters } from 'vuex'
{
  computed:{
    ...mapGetters(['num2'])
  }
}
// 使用时 this.num2
```

安装

当我们在代码中通过 `import Vuex from 'vuex'` 的时候，实际上引用的是一个对象，它的定义在 `src/index.js` 中：

```
export default {
// Vuex对象上还有一个Store对象
  Store,
// 当我们使用Vue.use(Vuex)去注册Vuex的时候，实际上就会执行install
  install,
  version: '__VERSION__',
  mapState,
  mapMutations,
  mapGetters,
  mapActions,
  createNamespacedHelpers
}
```

分析Vue.use

Vue.use(plugin);

参数

`{ Object | Function } plugin`

用法

安装Vue.js插件。如果插件是一个对象，必须提供install方法。如果插件是一个函数，它会被作为install方法。调用install方法时，会将Vue作为参数传入。install方法被同一个插件多次调用时，插件也只会被安装一次。

作用

注册插件，此时只需要调用install方法并将Vue作为参数传入即可。但在细节上有两部分逻辑要处理：

- 插件的类型，可以是install方法，也可以是一个包含install方法的对象。
- 插件只能被安装一次，保证插件列表中不能有重复的插件。

实现

```
Vue.use = function(plugin){
 const installedPlugins = (this._installedPlugins || (this._installedPlugins =
[]));
 if(installedPlugins.indexOf(plugin)>-1){
  return this;
 }
 // 其他参数
 const args = toArray(arguments,1);
 args.unshift(this);
 if(typeof plugin.install === 'function'){
  plugin.install.apply(plugin,args);
 }else if(typeof plugin === 'function'){
  plugin.apply(null,plugin,args);
 }
 installedPlugins.push(plugin);
 return this;
}
```

1. 在Vue.js上新增了use方法，并接收一个参数plugin。

2、首先判断插件是不是已经别注册过，如果被注册过，则直接终止方法执行，此时只需要使用indexOf方法即可。

3、toArray方法我们在就是将类数组转成真正的数组。使用toArray方法得到arguments。除了第一个参数之外，剩余的所有参数将得到的列表赋值给args，然后将Vue添加到args列表的最前面。这样做的目的是保证install方法被执行时第一个参数是Vue，其余参数是注册插件时传入的参数。

4、由于plugin参数支持对象和函数类型，所以通过判断plugin.install和plugin哪个是函数，即可知用户使用哪种方式祖册的插件，然后执行用户编写的插件并将args作为参数传入。

5、最后，将插件添加到installedPlugins中，保证相同的插件不会反复被注册。(面试官问为什么插件不会被重新加载)

当我们执行 `Vue.use(Vuex)` 的时候就执行下面的逻辑

和 `Vue-Router` 一样，Vuex 也同样存在一个静态的 `install` 方法，它的定义在 `src/store.js` 中：

```
export function install (_Vue) {
// 下面这部分的代码是保证了我们反复调用只会执行一次
  if (Vue && _Vue === Vue) {
    if (process.env.NODE_ENV !== 'production') {
      console.error(
        '[vuex] already installed. Vue.use(Vuex) should be called only once.'
```

```
      )
    }
    return
  }
  Vue = _Vue
// 然后就会执行 applyMixin的方法
  applyMixin(Vue)
}
```

`install` 的逻辑很简单，把传入的 `_Vue` 赋值给 `Vue` 并执行了 `applyMixin(Vue)` 方法，它的定义在 `src/mixin.js` 中：

```
// 参数Vue，分析Vue.use的时候，在执行install的时候，将Vue作为参数传进去。
export default function (Vue) {
  const version = Number(Vue.version.split('.')[0])

// 这里会对Vue的版本进行一个判断，因为Vuex2.x和1.x都是用了这一片的代码，只是2.x的时候不太一样
  if (version >= 2) {
// Vuex2.x就是通过mixin方法，在beforeCreate钩子函数混入vuexInit函数。mixin的作用是将mixin的内容混合到Vue的初始参数options中。
// 为什么是beforeCreate而不是created呢？因为如果是在created操作的话，$options已经初始化好了。
    Vue.mixin({ beforeCreate: vuexInit })
  } else {
    // override init and inject vuex init procedure
    // for 1.x backwards compatibility.
    const _init = Vue.prototype._init
    Vue.prototype._init = function (options = {}) {
      options.init = options.init
        ? [vuexInit].concat(options.init)
        : vuexInit
      _init.call(this, options)
    }
  }

  /**
   * Vuex init hook, injected into each instances init hooks list.
   */
  function vuexInit () {
// 然后vuexInit的时候就会取这个this.$options
    const options = this.$options
    // store injection
    if (options.store) {
// 如果发现options有store这个方法的时候，就会先进行类型判断，如果options.store是函数就执行上面的options.store()，如果不是就直接options.store，最后将结果赋值给this.$store
// 如果判断当前组件是根组件的话，就将我们传入的store挂在到根组件实例上，属性名为$store
      this.$store = typeof options.store === 'function'
        ? options.store()
        : options.store
// 但是如果options没有store那么就会去找它的parent上面的store
// 如果判断当前组件是子组件的话，就将我们根组件的$store也复制给子组件。注意是引用的复制，因此每个组件都拥有了同一个$store挂载在它身上。
    } else if (options.parent && options.parent.$store) {
      this.$store = options.parent.$store
    }
  }
}
```

```
// 这里有个问题，为什么判断当前组件是子组件，就可以直接从父组件拿到$store呢？面试官：父组件和子组件的执行顺序？
A：父beforeCreate-> 父created -> 父beforeMounte -> 子beforeCreate ->子create ->子beforeMount ->子 mounted -> 父mounted
// 可以得到，在执行子组件的beforeCreate的时候，父组件已经执行完beforeCreate了，那理所当然父组件已经有$store了。

// 这个过程就和Vue-Router的Init过程非常相似，这样的过程就保证了每一个Vue实例都会拥有一个$store这样一个对象，那么我们就可以在任意一个组件中都可以通过this.$store访问到这个store实例

// 那我们这个store什么时候传入呢？其实就是我们写代码的时候在执行new Vue的时候将store传入：代码
const app = new Vue({
    el:"#app",
    store
})
// 所以在根Vue下面就将store传入，非根组件就会去他们的parent上找，也能访问到这个store实例，所以上面在每一个组件beforeCreate之后都快可以拿到一个store实例
```

`applyMixin` 就是这个 `export default function`，它还兼容了 `Vue 1.0` 的版本，这里我们只关注 Vue 2.0 以上版本的逻辑，它其实就全局混入了一个 `beforeCreate` 钩子函数，它的实现非常简单，就是把 `options.store` 保存在所有组件的 `this.$store` 中，这个 `options.store` 就是我们在实例化 `Store` 对象的实例，稍后我们会介绍，这也是为什么我们在组件中可以通过 `this.$store` 访问到这个实例。

Store 实例化

我们在 `import Vuex` 之后，会实例化其中的 `Store` 对象，返回 `store` 实例并传入 `new Vue` 的 `options` 中，也就是我们刚才提到的 `options.store`.
举个简单的例子，如下：

```
export default new Vuex.Store({
  actions,
  getters,
  state,
  mutations,
  modules
  // ...
})
```

`Store` 对象的构造函数接收一个对象参数，它包含 `actions、getters、state、mutations、modules` 等 `Vuex` 的核心概念，它的定义在 `src/store.js` 中：

当代码中执行new Vuex.Store是做哪些事情呢?

```
let Vue //bind on install
// 1.首先定义这么一个class
export class Store {
// 2. 当执行new Vuex.store的时候就会去执行这个构造函数
  constructor (options = {}) {
    // Auto install if it is not done yet and `window` has `Vue`.自动安装，如果尚未完成，' window '有' Vue '。
    // To allow users to avoid auto-installation in some cases,为了让用户在某些情况下避免自动安装，
```

```
    // this code should be placed here. See #731
// 如果不通过 npm 方式去开发我们的代码，我们通过其他方式加载Vue或者Vuex的话，这个时候就会在window上注册一个Vue变量，这个时候需要手动去实现install方法，去注册Vuex
//在浏览器环境下，如果插件还未安装（!Vue即判断是否未安装），则它会自动安装。它允许用户在某些情况下避免自动安装。
    if (!Vue && typeof window !== 'undefined' && window.Vue) {
      install(window.Vue)
    }
// 判断如果是非生产环境下，这里会有几个断言
    if (process.env.NODE_ENV !== 'production') {
// 首先会去断言Vue，这个Vue其实就是上面定义的变量 let Vue，这个Vue什么时候会赋值呢？在install里面的时候会将_Vue作为参数赋值给Vue，那么就可以在文件任意地方去使用_Vue了
// 断言的意义在于我们执行store实例化之前，使用Vue.use这个方法去install对Vuex进行一个注册，注册完之后才能实例化
      assert(Vue, `must call Vue.use(Vuex) before creating a store instance.`)
// 这里也需要对Promise进行一个断言，因为这个Vuex库是依赖Promise的，如果浏览器没有Promise的话，需要对Promise打一个补丁
      assert(typeof Promise !== 'undefined', `vuex requires a Promise polyfill in this browser.`)
// 判断 this是Store的一个实例，意思是我们去执行Store构造函数的时候，必须通过new的方式，如果是直接调用Store构造函数的话，就会报一个警告
      assert(this instanceof Store, `Store must be called with the new operator.`)
    }

// 通过options拿一些属性，比如plugins(Vuex支持的一些插件)、strict(严格模式)
    const {
// 一个数组，包含应用在 store 上的插件方法。这些插件直接接收 store 作为唯一参数，可以监听 mutation（用于外部地数据持久化、记录或调试）或者提交 mutation （用于内部数据，例如 websocket 或 某些观察者）
      plugins = [],
// 使 Vuex store 进入严格模式，在严格模式下，任何 mutation 处理函数以外修改 Vuex state 都会抛出错误。
      strict = false
    } = options
//从option中取出state，如果state是function则执行，最终得到一个对象
    let {
      state = {}
    } = options
    if (typeof state === 'function') {
      state = state()
    }

// 给Store上面定义一些私有属性，定义一些初始值
        //用来判断严格模式下是否是用mutation修改state的
    // store 实例对象 内部的 state
        this._committing = false
        // 用来存放处理后的用户自定义的actions
        this._actions = Object.create(null)
        // 用来存放 actions 订阅
        this._actionSubscribers = []
        // 用来存放处理后的用户自定义的mutations
        this._mutations = Object.create(null)
        // 用来存放处理后的用户自定义的 getters
        this._wrappedGetters = Object.create(null)
        // 模块收集器，构造模块树形结构(重要1)
        this._modules = new ModuleCollection(options)
        // 用于存储模块命名空间的关系
```

```
        this._modulesNamespaceMap = Object.create(null)
        // 订阅
        this._subscribers = []
        // 用于使用 $watch 观测 getters
        this._watcherVM = new Vue()

    // bind commit and dispatch to self
// 通过 store缓存this。将dispatch与commit调用的this绑定为store对象本身，否则在组件内部this.dispatch时的this会指向组件的vm
    const store = this
// 通过 this 拿到 dispatch 和 commit 方法
    const { dispatch, commit } = this
// 然后对这两个方法进行重新赋值
// 为dispatch与commit绑定this（Store实例本身）。dispatch和commit是通过改变数据的两个方法，派发一个Action和提交一个Mutation
    this.dispatch = function boundDispatch (type, payload) {
  // 重写的意义就是当我们执行dispatch的时候上下文就是store实例
            return dispatch.call(store, type, payload)
    }
    this.commit = function boundCommit (type, payload, options) {
      return commit.call(store, type, payload, options)
    }

    // strict mode
        // 严格模式(使 Vuex store 进入严格模式，在严格模式下，任何 mutation 处理函数以外修改 Vuex state 都会抛出错误)
    this.strict = strict
// this._modules的初始化是在上面 this._modules = new ModuleCollection(options)
    const state = this._modules.root.state

    // init root module.
    // this also recursively registers all sub-modules
    // and collects all module getters inside this._wrappedGetters
// (重要2)这个的作用就是给actions、mutations、wrappedGetters这些值进行一些赋值
// 初始化根module，这也同时递归注册了所有子module，收集所有module的getter到_wrappedGetters中去，this._modules.root代表根module才独有保存的Module对象
    installModule(this, state, [], this._modules.root)

    // initialize the store vm, which is responsible for the reactivity
    // (also registers _wrappedGetters as computed properties)
// (重要3)这里是将 getters和state建立一些关系，变成响应式。通过vm重设store，新建Vue对象使用Vue内部的响应式实现注册state以及computed
    resetStoreVM(this, state)

    // apply plugins
    plugins.forEach(plugin => plugin(this))
// 安装devtools，可以很好的帮助我们观察store变化的整个过程
    if (Vue.config.devtools) {
      devtoolPlugin(this)
    }
  }
}
```

Store的构造类除了初始化一些内部变量以外，主要执行了installModule（初始化module）以及resetStoreVM（通过VM使store“响应式”）。

installModule

installModule的作用主要是用为module加上namespace名字空间（如果有）后，注册mutation、action以及getter，同时递归安装所有子module。

```
/*初始化module*/
function installModule (store, rootState, path, module, hot) {
  /* 是否是根module */
  const isRoot = !path.length
  /* 获取module的namespace */
  const namespace = store._modules.getNamespace(path)

  // register in namespace map
  /* 如果有namespace则在_modulesNamespaceMap中注册 */
  if (module.namespaced) {
    store._modulesNamespaceMap[namespace] = module
  }

  // set state
  if (!isRoot && !hot) {
    /* 获取父级的state */
    const parentState = getNestedState(rootState, path.slice(0, -1))
    /* module的name */
    const moduleName = path[path.length - 1]
    store.`_withCommit`(() => {
      /* 将子module设置称响应式的 */
      Vue.set(parentState, moduleName, module.state)
    })
  }

  const local = module.context = makeLocalContext(store, namespace, path)

  /* 遍历注册mutation */
  module.forEachMutation((mutation, key) => {
    const namespacedType = namespace + key
    registerMutation(store, namespacedType, mutation, local)
  })

  /* 遍历注册action */
  module.forEachAction((action, key) => {
    const namespacedType = namespace + key
    registerAction(store, namespacedType, action, local)
  })

  /* 遍历注册getter */
  module.forEachGetter((getter, key) => {
    const namespacedType = namespace + key
    registerGetter(store, namespacedType, getter, local)
  })

  /* 递归安装mudule */
  module.forEachChild((child, key) => {
    installModule(store, rootState, path.concat(key), child, hot)
  })
}
```

resetStoreVM

在说resetStoreVM之前，先来看一个小demo。

```
let globalData = {
    d: 'hello world'
};
new Vue({
    data () {
        return {
            ?state: {
                globalData
            }
        }
    }
});

/* modify */
setTimeout(() => {
    globalData.d = 'hi~';
}, 1000);

Vue.prototype.globalData = globalData;

/* 任意模板中 */
<div>{{globalData.d}}</div>
```

上述代码在全局有一个globalData，它被传入一个Vue对象的data中，之后在任意Vue模板中对该变量进行展示，因为此时globalData已经在Vue的prototype上了所以直接通过this.prototype访问，也就是在模板中的{{prototype.d}}。此时，setTimeout在1s之后将globalData.d进行修改，我们发现模板中的globalData.d发生了变化。其实上述部分就是Vuex依赖Vue核心实现数据的"响应式化"。

响应式原理（知识扩充部分）

Vue.js是一款MVVM框架，上手快速简单易用，通过响应式在修改数据的时候更新视图。Vue.js的响应式原理依赖于Object.defineProperty，尤大大在Vue.js文档中就已经提到过，这也是Vue.js不支持IE8 以及更低版本浏览器的原因。Vue通过设定对象属性的 setter/getter 方法来监听数据的变化，通过getter进行依赖收集，而每个setter方法就是一个观察者，在数据变更的时候通知订阅者更新视图。

1. 将数据data变成可观察（observable）的

那么Vue是如何将所有data下面的所有属性变成可观察的（observable）呢？

```
function observe(value, cb) {
    Object.keys(value).forEach((key) => defineReactive(value, key, value[key] , cb))
}

function defineReactive (obj, key, val, cb) {
    Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get: ()=>{
            /*....依赖收集等....*/
            /*Github:https://github.com/answershuto*/
            return val
        },
        set:newVal=> {
            val = newVal;
            cb();/*订阅者收到消息的回调*/
        }
    })
}
```

```
class Vue {
    constructor(options) {
        this._data = options.data;
        observe(this._data, options.render)
    }
}

let app = new Vue({
    el: '#app',
    data: {
        text: 'text',
        text2: 'text2'
    },
    render(){
        console.log("render");
    }
})
// 为了便于理解，首先考虑一种最简单的情况，不考虑数组等情况，代码如上所示。在initData中会调用observe这个函数将Vue的数据设置成observable的。当_data数据发生改变的时候就会触发set，对订阅者进行回调（在这里是render）。

// 那么问题来了，需要对app._data.text操作才会触发set。为了偷懒，我们需要一种方便的方法通过app.text直接设置就能触发set对视图进行重绘。那么就需要用到代理。

// 2. 代理
我们可以在Vue的构造函数constructor中为data执行一个代理proxy。这样我们就把data上面的属性代理到了vm实例上。
_proxy.call(this, options.data);/*构造函数中*/

/*代理*/
function _proxy (data) {
    const that = this;
    Object.keys(data).forEach(key => {
        Object.defineProperty(that, key, {
            configurable: true,
            enumerable: true,
            get: function proxyGetter () {
                return that._data[key];
            },
            set: function proxySetter (val) {
                that._data[key] = val;
            }
        })
    });
}
就可以用app.text代替app._data.text了
```

了解完响应式原理，接着来看代码。

```
/* 通过vm重设store，新建Vue对象使用Vue内部的响应式实现注册state以及computed */
function resetStoreVM (store, state, hot) {
  /* 存放之前的vm对象 */
  const oldVm = store._vm

  // bind store public getters
  store.getters = {}
```

```
  const wrappedGetters = store._wrappedGetters
  const computed = {}

  /* 通过Object.defineProperty为每一个getter方法设置get方法，比如获取
this.$store.getters.test的时候获取的是store._vm.test，也就是Vue对象的computed属性 */
  forEachValue(wrappedGetters, (fn, key) => {
    // use computed to leverage its lazy-caching mechanism
    computed[key] = () => fn(store)
    Object.defineProperty(store.getters, key, {
      get: () => store._vm[key],
      enumerable: true // for local getters
    })
  })

  // use a Vue instance to store the state tree
  // suppress warnings just in case the user has added
  // some funky global mixins
  const silent = Vue.config.silent
  /* Vue.config.silent暂时设置为true的目的是在new一个Vue实例的过程中不会报出一切警告 */
  Vue.config.silent = true
  /*  这里new了一个Vue对象，运用Vue内部的响应式实现注册state以及computed*/
  store._vm = new Vue({
    data: {
      ?state: state
    },
    computed
  })
  Vue.config.silent = silent

  // enable strict mode for new vm
  /* 使能严格模式，保证修改store只能通过mutation */
  if (store.strict) {
    enableStrictMode(store)
  }

  if (oldVm) {
    /* 解除旧vm的state的引用，以及销毁旧的Vue对象 */
    if (hot) {
      // dispatch changes in all subscribed watchers
      // to force getter re-evaluation for hot reloading.
      store._withCommit(() => {
        oldVm._data.?state = null
      })
    }
    Vue.nextTick(() => oldVm.$destroy())
  }
}
```

resetStoreVM首先会遍历wrappedGetters，使用Object.defineProperty方法为每一个getter绑定上get方法，这样我们就可以在组件里访问this.$store.getter.test就等同于访问store._vm.test。

```
forEachValue(wrappedGetters, (fn, key) => {
  // use computed to leverage its lazy-caching mechanism
  computed[key] = () => fn(store)
  Object.defineProperty(store.getters, key, {
    get: () => store._vm[key],
    enumerable: true // for local getters
  })
})
```

之后Vuex采用了new一个Vue对象来实现数据的“响应式化”，运用Vue.js内部提供的数据双向绑定功能来实现store的数据与视图的同步更新。

```
store._vm = new Vue({
  data: {
    ?state: state
  },
  computed
})
```

这时候我们访问store._vm.test也就访问了Vue实例中的属性。
这两步执行完以后，我们就可以通过this.$store.getter.test访问vm中的test属性了。