保证一个类仅有一个实例，并提供一个访问它的全局访问点，这样的模式就叫做单例模式。

单例模式是设计模式中相对较为容易理解、容易上手的一种模式，同时因为其具有广泛的应用场景，也是**面试题里的常客**。因此单例模式这块我们除了讲解单例模式的原理及其在 Vuex 中的应用实践(本节)，还会附上两道面试真题供大家练手(下节)。

单例模式的实现思路

现在我们先不考虑单例模式的应用场景，单看它的实现，思考这样一个问题：如何才能保证一个类仅有一个实例？ 一般情况下，当我们创建了一个类（本质是构造函数）后，可以通过new关键字调用构造函数进而生成任意多的实例对象。像这样：

```
class SingleDog {
    show() {
        console.log('我是一个单例对象')
    }
}

const s1 = new SingleDog()
const s2 = new SingleDog()

// false
s1 === s2
```

楼上我们先 new 了一个 s1，又 new 了一个 s2，很明显 s1 和 s2 之间没有任何瓜葛，两者是相互独立的对象，各占一块内存空间。而单例模式想要做到的是，**不管我们尝试去创建多少次，它都只给你返回第一次所创建的那唯一的一个实例。**

要做到这一点，就需要构造函数**具备判断自己是否已经创建过一个实例**的能力。我们现在把这段判断逻辑写成一个静态方法(其实也可以直接写入构造函数的函数体里）：

```
class SingleDog {
    show() {
        console.log('我是一个单例对象')
    }
    static getInstance() {
        // 判断是否已经new过1个实例
        if (!SingleDog.instance) {
            // 若这个唯一的实例不存在，那么先创建它
            SingleDog.instance = new SingleDog()
        }
        // 如果这个唯一的实例已经存在，则直接返回
        return SingleDog.instance
    }
}

const s1 = SingleDog.getInstance()
const s2 = SingleDog.getInstance()

// true
s1 === s2
```

除了楼上这种实现方式之外，getInstance的逻辑还可以用**闭包**来实现：

```
SingleDog.getInstance = (function() {
    // 定义自由变量instance，模拟私有变量
    let instance = null
    return function() {
        // 判断自由变量是否为null
        if(!instance) {
            // 如果为null则new出唯一实例
            instance = new SingleDog()
        }
        return instance
    }
})()
```

可以看出，在getInstance方法的判断和拦截下，我们不管调用多少次，SingleDog都只会给我们返回一个实例，s1和s2现在都指向这个唯一的实例。

生产实践：Vuex中的单例模式

近年来，基于 Flux 架构的状态管理工具层出不穷，其中应用最广泛的要数 Redux 和 Vuex。无论是 Redux 和 Vuex，它们都实现了一个全局的 Store 用于存储应用的所有状态。这个 Store 的实现，正是单例模式的典型应用。这里我们以 Vuex 为例，研究一下单例模式是怎么发光发热的：

理解 Vuex 中的 Store

> 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。至此它便作为一个“唯一数据源(SSOT)”而存在。这也意味着，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。 —— Vuex官方文档

在Vue中，组件之间是独立的，组件间通信最常用的办法是 props（限于父组件和子组件之间的通信），稍微复杂一点的（比如兄弟组件间的通信）我们通过自己实现简单的事件监听函数也能解决掉。

但当组件非常多、组件间关系复杂、且嵌套层级很深的时候，这种原始的通信方式会使我们的逻辑变得复杂难以维护。这时最好的做法是将共享的数据抽出来、放在全局，供组件们按照一定的的规则去存取数据，保证状态以一种可预测的方式发生变化。于是便有了 Vuex，这个用来存放共享数据的唯一数据源，就是 Store。

关于 Vuex 的细节，大家可以参考**Vuex的官方文档**，此处提及 Vuex，除了为了拓宽大家的知识面，更重要的是为了说明单例模式在生产实践中广泛的应用和不可或缺的地位。如果对 Vuex 没有兴趣，那么大家只需关注“一个 Vue 实例只能对应一个 Store”这一点即可。

Vuex如何确保Store的唯一性

我们先来看看如何在项目中引入 Vuex：

```
// 安装vuex插件
Vue.use(Vuex)

// 将store注入到Vue实例中
new Vue({
    el: '#app',
    store
})
```

通过调用 `Vue.use()` 方法，我们安装了 Vuex 插件。Vuex 插件是一个对象，它在内部实现了一个 install 方法，这个方法会在插件安装时被调用，从而把 Store 注入到Vue实例里去。也就是说每 install 一次，都会尝试给 Vue 实例注入一个 Store。

在 install 方法里，有一段逻辑和我们楼上的 `getInstance` 非常相似的逻辑：

```
let Vue // 这个Vue的作用和楼上的instance作用一样
...

export function install (_Vue) {
  // 判断传入的Vue实例对象是否已经被install过Vuex插件（是否有了唯一的state）
  if (Vue && _Vue === Vue) {
    if (process.env.NODE_ENV !== 'production') {
      console.error(
        '[vuex] already installed. Vue.use(Vuex) should be called only once.'
      )
    }
    return
  }
  // 若没有，则为这个Vue实例对象install一个唯一的Vuex
  Vue = _Vue
  // 将Vuex的初始化逻辑写进Vue的钩子函数里
  applyMixin(Vue)
}
```

楼上便是 Vuex 源码中单例模式的实现办法了，套路可以说和我们的 `getInstance` 如出一辙。通过这种方式，可以保证一个 Vue 实例（即一个 Vue 应用）只会被 install 一次 Vuex 插件，所以每个 Vue 实例只会拥有一个全局的 Store。

小结

这里大家不妨开个脑洞，思考一下：如果我在 install 里没有实现单例模式，会带来什么样的麻烦？

我们通过上面的源码解析可以看出，每次 install 都会为Vue实例初始化一个 Store。假如 install 里没有单例模式的逻辑，那我们如果在一个应用里不小心多次安装了插件：

```
// 在主文件里安装Vuex
Vue.use(Vuex)

...(中间添加/修改了一些store的数据)

// 在后续的逻辑里不小心又安装了一次
Vue.use(Vuex)
```

失去了单例判断能力的 install 方法，会为当前的Vue实例重新注入一个新的 Store，也就是说你中间的那些数据操作全都没了，一切归 0。因此，单例模式在此处是非常必要的。

除了说在 Vuex 中大展身手，我们在 Redux、jQuery 等许多优秀的前端库里也都能看到单例模式的身影。重要的单例模式自然在面试中有了重要的地位，下一节，我们就来看两道面试真题~

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）