Vue.ts

引用文章

markdown文件生成目录的方式

实用帖 | 如何为 Markdown 文件自动生成目录?

Vue-CLI支持

Vue-CLI内建了TypeScript工具支持，在新建项目时可以选择使用TypeScript扩展，包括了针对Vue Core的官方类型声明，还包括了Vue Router和Vuex提供了相应的声明文件

使用Vue-CLI会自动创建 `tsconfig.json` 文件，基本上使用默认的配置文件就可以满足要求。

改造组件

使用TypeScript编写Vue单文件组件有两种方式，一种是通过 `Vue.extend()` 方法，另一种是基于类的Vue组件（在使用Vue-CLI创建项目的时候可以选择），我选择使用了后者，可以提供更优雅、更类似于JSX的书写体验。

需要安装vue-class-component用来将Vue组件改写为基于Class的形式，也可以选择使用vue-property-decorator，后者依赖于前者，而且提供了额外的装饰符，让编写更简单。

使用的时候，将原来导出的类型由对象改为了Class，并且使用 `@Component` 装饰符，如果有要引入的其他子组件，也放到 `@Component` 中。

```
@Component({
  components: {
    Child
  }
})
export default class HelloVue extends Vue {
  // 组件内容
}
```

要注意，虽然使用了 `export default` ，但是Class的名字还是最好准确定义，这样便于IDE和Lint工具进行追踪、提示。

组件属性顺序

没有发现Lint和Prettier规则来强制规定组件内的属性顺序，所以约定好一个书写顺序，作为最佳实践

要注意，组件引用、Mixin和Filters都放到了组件外部。总体的顺序分为了三部分：

数据（ `Inject` → `Prop` → `Data` → `Computed` → `Model` → `Vuex-State` → `Vuex-Getter` → `Proivde` ）
方法（Vuex-Mutation → Vuex-Action → Methods → Watch）
钩子函数（生命周期钩子 → 路由钩子）
完整的组件如下，具体写法后面单独列出来（不包含Mixin）：

```
@Component({ components: { Child } })
export default class App extends Vue {
  // 数据 (Inject → Prop→ Computed → Model → Vuex-State → Vuex-Getter → Proivde)
  // 使用祖先组件注入的数据
  @Inject() readonly value1!: string;

  // 组件的 Data
  value = 'hello';

  // 父组件传入 Prop
  @Prop(Number) readonly value2!: number;

  // 计算属性
  get value3(): string {
    return this.value1;
  }

   // 定义 组件的 Model 属性
  @Model('change', { type: Boolean, default: false }) checked!: boolean;

  // Vuex Store 中定义的 state，作为计算属性定义在组件内
  @State value4!: string;

  // Vuex Store 中定义的 getter，作为计算属性定义在组件内
  @Getter value5!: string;

  // 为子孙组件提供数据
  @Provide() root = 'Root';

  /* ------------------------------------------------ */
  // 方法 (Vuex-Mutation → Vuex-Action → Methods → Watch)
  // Vuex Store 中定义的 Mutation，作为方法定义在组件内
  @Mutation(UPDATE_TITLE_MUTATION) updateTitle!: (payload: { title: string }) => void;

  // Vuex Store 中定义的 Action，作为方法定义在组件内
  @Action(UPDATE_TITLE_ACTION) updateTitleSync!: () => void;

  // 组件内的 Method
  get foo(): string {
    return this.isCollapse ? 'collapsed-menu' : 'expanded-menu';
  }

  // 组件内的 Watch
  @Watch('value1', { immediate: true, deep: true })
  onDataChanged(newVal: string, oldVal: string): void {
    this.foo();
  }

  /* ------------------------------------------------ */
  // 钩子函数 (生命周期钩子 → 路由钩子)
  beforeCreated()

  created()

  beforeMount()

  mounted() {}

  beforeUpdate() {}

  updated(){}

  activated(){}

  deactivated(){}

  beforeDetory(){}

  destoryed(){}

  errorCaptured(){}

  beforeRouteEnter(){}

  beforeRouteUpdate(){}
```

```
  beforeRouteLeave(){}
}
```

相关API

Data

直接在Class定义即可（实际上就是Class的新语法，与在Class的 `constructor` 中定义相同）

```
import { Vue, Component, Prop } from 'vue-property-decorator'

@Component
export default class YourComponent extends Vue {
  msg: number = 123;
}
```

计算属性

计算属性采取使用 `getter` 的形式定义，在Class内部可以使用 `get` 和 `set` 关键字，设置某个属性的存值函数和取值函数：

```
import { Vue, Component, Prop } from 'vue-property-decorator'

@Component
export default class YourComponent extends Vue {
  num: number = 1;

  get: value: string() {
    return this.num + 1;
  }
}
```

同时定义 `set` 实现了对计算属性的赋值。

@Prop

@Prop接受的参数就是原来在Vue中 `props` 中传入的参数

```
import { Vue, Component, Prop } from 'vue-property-decorator'

@Component
export default class YourComponent extends Vue {
  @Prop(Number) readonly propA: number | undefined
  @Prop({ default: 'default value' }) readonly propB!: string
  @Prop([String, Boolean]) readonly propC: string | boolean | undefined
}
```

@PropSync

`@PropSync` 与 `Prop` 类似，不同之处在于 `@PropSync` 会自动生成一个计算属性，计算属性的 `getter` 返回传入的Prop，计算属性的 `setter` 中会执行Vue中提倡的更新Prop的 `emit:updatePropName`

```
import { Vue, Component, PropSync } from 'vue-property-decorator'

@Component
export default class YourComponent extends Vue {
  @PropSync('name', { type: String }) syncedName!: string
}
```

相当于：

```
export default {
  props: {
    name: {
      type: String
    }
  },
  computed: {
    syncedName: {
      get() {
        return this.name
      },
      set(value) {
        this.$emit('update:name', value)
      }
    }
  }
```

使用时需要配合 `.sync` 修饰符使用（即在组件上定义对应的更新方法）：

```
<hello-sync :my-prop.sync="syncValue" />
<!-- 相当于 -->

<hello-sync :my-prop="syncValue" @update:name="(name) => syncValue = name" />
```

定义方法

定义方法与Data类型，直接在Class中定义方法即可：

```
@Component
export default class HelloChild extends Vue {
  sayHi(): string {
    return 'hello'
  }
}
```

@Watch

使用 `@Watch` 定义侦听器，被装饰的函数就是侦听器执行方法：

```
@Component
export default class HelloChild extends Vue {
  @Watch('msg', { immediate: true, deep: true })
  onMsgChanged(newVal: string, oldVal: string): void {
    this.oldMsg = oldVal;
  }
}
```

@Emit

想要触发父组件中定义在组件实例上的方法，需要使用 `@Emit` 装饰符。 `@Emit` 接受一个参数，是要触发的事件名，如果要出发的事件名和被装饰的方法同名，那么这个参数可以省略。 `@Emit` 返回值就是传递给事件的参数。

```
@Component
export default class HelloChild extends Vue {
  @Emit()
  sayHi(): string {
    return 'hello'
  }

  @Emit('go')
  goHere(): string {
    return 'gogogo'
  }
}
```

相当于：

```
export default {
  sayHi() {
    this.$emit('sayHi', 'hello')
  },

  goHere() {
    this.$emit('go', 'gogogo')
  }
}
```

@Model

一般用来在自定义的组件上使用 `v-model` ，自定义组件中包含可交互元素（例如 `<input>` 或者 `<checkbox>` ），当组可交互元素绑定的值发生变化（ `oninput` 、 `onchange` ）时，会传递到父组件绑定的 `v-model` 属性上。

关于自定义组件 `v-model` 的介绍可以参考官方文档。

```
<template>
  <el-checkbox :checked="checked" @change="changeHandler" />
</template>

<script lang="ts">
import { Component, Vue, Model, Emit } from 'vue-property-decorator';

@Component
export default class HelloVModel extends Vue {
  @Model('change', { type: Boolean, default: false }) checked!: boolean;

  @Emit('change')
  changeHandler(checked: boolean) {
    return checked;
  }
}
</script>
```

使用的时候：

```
<hello-v-model v-model="componentVModel" />
```

自定义组件利用了@Model，定义了checked属性，并且利用了@change事件，当checkbox发生了change事件后，父组件中的componentVModel就会随之发生变化。

实际上Modal和.sync修饰符都是Vue为了方便子组件同步数据到父组件而实现的语法糖

@Ref

当使用 `ref` 属性标记一个子组件或者HTML元素的时候，需要使用 `@Ref` 修饰符来找到标记的组件或元素。例如：

```
<div ref="someRef"></div>
<hello-ref ref="hello" />
```

如果我们需要获取 `ref` 引用时：

```
import { Component, Vue, Watch, Ref } from 'vue-property-decorator';
import HelloRef from '@/views/baseKnowledge/hello-vue/components/HelloRef.vue';

@Component({
  components: {
    HelloChild,
    HelloSync,
    HelloVModel,
    HelloRef
  }
})
export default class HelloVue extends Vue {
  @Ref() readonly hello!: HelloRef;
  @Ref() readonly someRef!: HTMLDivElement;
}
```

`@Ref` 后面跟的参数就是对应的 `ref` 的值，需要为其指定类型，如果是原生的元素，可以使用对应的与内置原生元素类型，如果是自定义组件，那么可以将引入的组件作为类型

如果在 `HelloRef` 中定义了一个 `notify` 方法，我们就可以按照如下调用：

```
this.hello.notify()
```

但是现在应该是Vue-Cli内置的Vue类型系统有一个Bug，始终会报如下的错误：

```
Error:(141, 16) TS2551: Property 'notify' does not exist on type 'Vue'. Did you mean '$notify'?
```

我的处理方法时，在为 `hello` 定义类型时，手写类型，传入我们需要的方法类型就OK了

```
@Ref() readonly hello!: { notify: (from?: string) => {} };
```

Mixins

`vue-property-decorator` 的 `Mixins` 方法完全来源于 `vue-class-component` ，使用方法如下。首先创建一个Mixin：

```
// visible-control-mixin.ts
import Vue from 'vue';
import Component from 'vue-class-component';

@Component
export default class MyMixin extends Vue {
  visible = false;

  get buttonText(): string {
    return this.visible ? 'Close' : 'Open';
  }

  toggleVisible() {
    this.visible = !this.visible;
  }
}
```

然后在组件中引入，这时候我们就不再需要组件继承自 `Vue` 了，而是继承自Mixin后的组件， `Mixins` 方法可以接受个参数，作为混入的Mixin：

```
import { Component, Mixins } from 'vue-property-decorator';
import VisibleControlMixin from '@/mixins/visible-control-mixin';

@Component
export default class MixinExample extends Mixins(VisibleControlMixin) {}
```

@Inject/@Provide

`provide` 和 `inject` 主要的目的就是透传属性，从一个根节点 `provide` 一个属性，无论多远的一个子节点都可以通过 `inject` 获得这个属性，与React的Context特性非常类似

虽然可以通过使用这两个属性，实现全局的数据共享，但是Vue的文档提示，这两个属性主要为高阶插件和组件库提供用例，并不直接推荐用于应用程序代码中，所以简单了解即可。

在根组件中使用 `@Provide` 提供数据：

```
import { Component, Vue, Provide } from 'vue-property-decorator';
import Child from '@/views/baseKnowledge/inject-provide/components/Child.vue';

@Component()
export default class InjectProvide extends Vue {
  @Provide() root = 'Root';
  @Provide('parent') readonly parentValue = 'Grandpa';

  // 相当于
  //  provide() {
  //   return {
  //     root: 'Root Initial Value',
  //     parent: this.parentValue
  //   }
  // }
}
```

在子组件中使用 `@Inject` 获取数据：

```
import { Component, Vue, Inject } from 'vue-property-decorator';

@Component()
export default class InjectProvideChild extends Vue {
  @Inject() readonly root!: string;
  @Inject() readonly parent!: string;
}
```

要注意， `provide` 和 `inject` 绑定并不是可响应的。这是刻意为之的。然而，如果传入了一个可监听的对象，那么其对象的属性还是可响应的。

`vue-property-decorator` 也提供了响应式插入数据的装饰器 `@ProvideReactive` 和 `@InjectReactive` ，但是有两个问题：

无法与 `@Inject` / `@Provide` 在同一个组件中同时工作
当从一个其他组件跳转到使用了 `@ProvideReactive` 和 `@InjectReactive` 后，会很有大概率报错 `Error in nextTick: "TypeError: Cannot redefine property: parent"` 导致渲染出错
所以暂时不推荐使用这两个装饰器。

改造Vue Router

使用Vue CLI创建的TypeScript项目，Vue Router与TypeScript配合基本不再需要进行额外的处理，除了对组件内的路由钩子方法需要提前进行注册。

使用 `vue-class-component` 提供的 `Component.registerHooks` 方法来提前注册，要注意，注册需要在引入路由之前完成

```
// ./src/components/class-component-hooks.ts

// 在此注册其他插件提供的钩子函数，用来在 Vue Class 组件中使用
// 例如 Vue Router 提供的钩子函数
// 必须在 router 之前引入
import Component from 'vue-class-component';

// Register the router hooks with their names
Component.registerHooks(['beforeRouteEnter', 'beforeRouteLeave', 'beforeRouteUpdate
```

在 `main.ts` 中引入：

```
import '@/components/class-component-hooks';
import router from './router';
```

改造VueX

Vuex与TypeScript配合会复杂一些，并且体验也不算太好，需要安全额外的包实现与TypeScript的配合使用，有三种方案来帮助我们使用TypeScript版本的Vuex

使用vue-class-component

第一种方案是使用 `vue-class-component` 配合以前常常使用 `mapState` 等帮助方法：

```
import { Component, Vue } from 'vue-property-decorator';
import { mapState, mapMutations } from 'vuex'

@Component(
  {
    // Vuex's component binding helper can use here
    computed: mapState(['count']),
    methods: mapMutations(['increment'])
  }
)
export default class App extends Vue {
  // additional declaration is needed
  // when you declare some properties in `Component` decorator
  count!: number
  increment!: () => void
}
```

这种方式的好处是可以通过 `mapState` 等方法将Store中定义的数据、方法一次性引入组件，确定就是这种『一次性』其实也还需要在组件内部再次定义，并且如果采用这种形式配合 `vue-property-decorator` 使用时，会将计算属性、方法等逻辑打乱。另外，通过这种方式调用Mutation和Action，也不是类型安全的（即没有办法校验我们传入的参数是否与Store中定义的 `payload` 类型相匹配

使用vuex-class

第二种方案是vuex-class，它与上一种方案相同，并没有对Vuex的Store中的代码进行改造，而是在组件消费Store中的数据、方法时，提供了一些遍历的API，简化使用方法

```
import { Component, Vue } from 'vue-property-decorator';
import {
  State,
  Getter,
  Action,
  Mutation,
  namespace
} from 'vuex-class'

const someModule = namespace('path/to/module')

@Component
export class MyComp extends Vue {
  @State('foo') stateFoo
  @State(state => state.bar) stateBar
  @Getter('foo') getterFoo
  @Action('foo') actionFoo
  @Mutation('foo') mutationFoo
  @someModule.Getter('foo') moduleGetterFoo

  // If the argument is omitted, use the property name
  // for each state/getter/action/mutation type
  @State foo
  @Getter bar
  @Action baz
  @Mutation qux

  created () {
    this.stateFoo // -> store.state.foo
    this.stateBar // -> store.state.bar
    this.getterFoo // -> store.getters.foo
    this.actionFoo({ value: true }) // -> store.dispatch('foo', { value: true })
    this.mutationFoo({ value: true }) // -> store.commit('foo', { value: true })
    this.moduleGetterFoo // -> store.getters['path/to/module/foo']
  }
}
```

注意，给 `namespace` 传入的参数是Vuex中 `module` 的命名空间，并非模块的目录路径

这种方法虽然不能使用 `mapState` 等辅助函数，但是好在使用 `@State` 等装饰符集中导入，也还算清晰明了。但是缺点仍然是没有办法完全进行类型安全的Mutation和Action调用

使用vuex-module-decorators

如果想要实现获得完全类型安全的Vuex，那么就需要使用 `vuex-module-decorators` ，它对Vuex的Store也进行了Class化的改造，引入了 `VuexModule` 和 @Mutation等修饰符，让我们能够使用Class形式来编写Store

使用的时候，按照下面的形式来改写Store：

```
import { Module, Mutation, Action, VuexModule } from 'vuex-module-decorators';
import store from '@/store';
import { setTimeoutThen } from '@/utils';

@Module({ dynamic: true, namespaced: true, store, name: 'testStore' })
export default class TestStore extends VuexModule {
  // state
  message: string = '';

  get UpperMessage() {
    return this.message;
  }

  @Mutation
  UPDATE_MESSAGE_MUTATION(title: string): void {
    this.message = title;
  }

  @Action
  async UPDATE_MESSAGE_ACTION(): Promise<string> {
    const result: string = await setTimeoutThen(1000, 'ok');
    this.context.commit('UPDATE_MESSAGE_MUTATION', result);
    return result;
  }
}
```

要注意，改写的Module在 `@Module` 中传入了几个属性，传入 `namesapced` 和name来使用Module成为命名空间下的模块，此外还需要传入 `dynamic` ，让这个模块成为动态注册的模块，同时还需要将完全空白的 `store` 传入给这个模块

完成改造之后，在使用的时候就可以使用他提供的 `getModule` 方法获得类型安全了，使用方法：

```
import { getModule } from 'vuex-module-decorators';
import TestStore from '@/store/modules/testStore';

const testStore = getModule(TestStore);
testStore.message;
testStore.UPDATE_MESSAGE_MUTATION('Hello');
testStore.UPDATE_MESSAGE_ACTION();
```

最终选择vuex-class

我选择了使用第二种方案，相比于第一种方案能够将组件内的逻辑几种，并且通过相关的修饰符能够显示的提醒代码的含义。相比于第三种方案编写复杂度也有了一定降低

对于类型安全我的做法是，当在组件内引入Mutation时再次编写对应的函数接口，在Vuex中编写的时候，通过引入Vuex提供的类型配合自定义类型，保证类型安全。

具体的实践在我们『相关实践』部分会有更具体写的介绍。

相关实践

TypeScript类型校验

Vue-CLI使用的TypeScript插件是 `@vue/cli-plugin-typescript` ，它将 `ts-loader` 和 `fork-ts-checker-webpack-plugin` 配合使用，实现线程外的快速类型检查。

在默认配置下，如果发现了TypeScript类型错误，仅仅会在终端进行提示，而不会中断编译过程。我认为TypeScript发现的类型错误是比较严重的错误类型，应当中断编译过程，让开发者给予足够的重试，所以需要进行配置，让TypeScript发现的错误中断编译过程并且在浏览器界面上进行提示。

常规的TypeScript项目只需要在 `tsconfig.json` 中的 `compilerOptions` 选项中配置 `noEmitOnError` 即可，这就会阻止TypeScript编译器在发现错误的时候继续将 `.ts` 文件编译成为.js文件。

但是由于Vue CLI使用了 `fork-ts-checker-webpack-plugin` 这个插件，需要进行额外的配置（在 `@vue/cli-plugin-typescript` 的文档中并没有明确的介绍，需要到 `fork-ts-checker-webpack-plugin` 的文档中自行查找）

在 vue.config.js 中，使用 chainWebpack 属性，对其进行配置，将 async 设置为 false：

```
module.exports = {
  chainWebpack: config => {
    // 配置 TypeScript 检查配置
    // https://github.com/TypeStrong/fork-ts-checker-webpack-plugin#options
    config.plugin('fork-ts-checker').tap(option => {
      option[0].async = false;
      return option;
    })
  },
};
```

另外，在 tsconfig.json 中的 compilerOptions 选项中将 noImplicitAny 设定为 true ，这样如果编译器推导出的结果默认为 any 的话，编译器会报错。不推荐轻易使用 any ，除非有明确的理由。即使需要 any 也要现实的标注为 any ，这样才能享受到TypeScript的强类型提示的好处（更何况这不是一个就项目改造）

Lint工具

配置比较高的Lint级别，可能会导致开发时的效率稍微降低，但是有助于项目的长期发展，以及良好的代码习惯的养成，也避免了保存代码时不提示，但是在Commit时一堆错误不好修改的问题。

配置的Lint工具包括了：

ESLint

使用了 plugin:vue/recommended/@vue/prettier/@vue/typescript/plugin:prettier/recommended 四个规则，使用 @typescript-eslint/parser 解析器对 .vue 文件和 .ts 文件都会进行校验（这些都是Vue CLI自动配置的）。

同时在 vue.config.js 中配置了 lintOnSave: process.env.NODE_ENV === 'development' ? 'error' : 'false', 让ESLint 检测到错误时不仅在终端中提示，还会在浏览器界面上展示，同时中断编译过程

Prettier

配置了Prettier，根据它提供的不多的选项进行了配置，有可能会与公司代码提交平台的规范有冲突，如果发现冲突后面再进行调整。

由于ESLint中配置了 @vue/prettier 和 plugin:prettier/recommended ， Prettier发现的错误也会中断编译过程。

不过Prettier的问题相对比较好修复，IDE中配置好Prettier的插件后，可以一键进行修复。

StyleLint

对于样式文件使用StyleLint进行了检查，在 vue.config.js 中通过 configureWebpack 方法引入了StyleLint插件，对所有样式文件以及 .vue 单文件组件、HTML组件中的样式代码进行校验。

同样如果出错会中断编译过程（这个应该是Bug，即便想关闭配置了相关选项后也无法关闭）

在 .stylelintrc.js 中定义了一些规则，也可能与公司的代码规范有冲突，后续进行调整。

目录组织

这部分也是我个人的尝试，带有一定个人喜好，希望在这个项目中验证是否可行.

types.ts文件

一般来说，如果 .vue 文件或者其他的 .ts 文件，如果涉及到的类型不多不复杂，可以直接在文件中进行定义，但是如果对应的类型接口需要复用，或者比较多，最好将原来的文件变为目录，文件名改为 index ，里面配套添加 types.ts 文件，用来声明类型，例如有一个 example.ts 文件，如果需要定义的类型比较复杂，那么将这个文件替换为：

```
- example
  |
  - index.ts
  - types.ts
```

目录组织结构

基本按照Vue CLI生成目录结构，各个插件的配置文件（例如 .eslintrc.js 、 .eslintignore 文件）、环境配置文件（例如 .env ）都放置在根目录，ElementUI定制样式文件放置在根目录下的 theme 目录，其余文件都放在 src 目录下。（ test 目录是单元测试文件的目录，暂时没有引入单测的计划，不过在业务空闲期可能会考虑使用Jest进行单元测试）

Vue CLI为我们配置了Webpack的Alias， @ 会指向 src 目录，所以导入文件时，除非是只被一个组件引用的内部组件可使用 ./ 相对路径，其余路径都建议使用 @ 进行引入，保证文件移动时无需关注组件和资源的路径问题。

src 目录下，除了根目录下的几个文件 App.vue 等之外，还有以下几个目录：

（1） src/assets 目录，将所有前端的资源（图片、音频、视频、字体等）都建立单独的目录放到该目录下，例如：

```
- src
  |
  - assets
    |
    - images
    - videos
    - icons
```

如果图片资源比较多，还可以在 images 目录下按模块进行分割

（2） src/components 目录，这里放置的组件应该是一些自己封装的、作为模块的功能化基础组件，例如在ElementUI基础上根据封装的组件，会被多个页面引用，例如自己封装的筛选组件等。

这里的结构应该只有单层，即一个目录是一个 component ，不再划分新的目录（组件内部可以根据需要划分子目录）：

```
- src
  |
  - components
    |
    - query
      |
      - index.vue
        |
        - components
          |
          - handler.vue
          - result.vue
    - user-list
      |
      - index.vue
```

（3） src/mixins ，放置Mixin文件，不再划分目录，通过文件名和注释描述该文件实现的Mixin的功能

（4） src/plugins ，引入的或者自己实现的插件，目录要求与 components 相同

（5） src/router ，路由相关文件，具体在路由部分的实践介绍

（6） src/store ，Vuex相关文件，具体在Vuex部分的实践介绍

（7） src/styles ，样式相关文件，只放置全局的样式变量和样式Mixin，通过Style-resource-loader配置自动进行引入

（8） src/utils ，帮助函数的目录，总的出口文件是 index.ts ，用于组织帮助函数的分类，其余文件按照帮助函数的类别进行划分，无法分类比较零散的帮助函数放置在 common-helper ，目前其他的帮助函数分为了下面几个：

date-time-helper.ts ，用来处理日期时间相关的函数都在这里
router-helper.ts ，与路由相关的函数在这里，例如路由守卫中应用的方法等
network-helper ，与网络请求有关的方法都在这里，最主要的就是对Axios的封装，在后面的网络请求部分单独介绍
（9） src/views ，是最常用的目录，里面按照模块划分目录（可以大致理解为按照导航菜单的一级目录划分），里面有一个 common 目录，用来放置被多个页面同时引用的组件。

例如有一个 user-create.vue ，在 user-info 路由下有使用，在 user-admin 中也有使用，那么就可以把它放到 common 中：

```
- src
  |
  - views
    |
    - common
      |
      - user
        |
        - user-create.vue
```

现在的 common 目录下有两个组件，分别是 not-found （对应404路由）和 menu 目录，不过这个目录的划分可能会比较主观，也会随着业务不同进行调整。

命名

这里的命名风格主要参考了Vue的风格指南和Element的实践

目录的命名

目录使用 kebab-case 格式进行命名，如果里面的主文件使用 index.ts 或者 index.vue 命名，如果有子组件则放到与 index 平级的 components 目录下，例如：

```
- src
  |
  - views
    |
    - layout.vue
      |
      - index.vue
      - components
        |
        - header.vue
        - main.vue
        - footer.vue
```

另外，如果一个组件的代码长度超过了300行，就需要考虑拆分组件了，如果超过了500行，则必须拆分组件。

这样做的好处是，当进入一个目录后，一眼就能看出主文件是哪一个，对应的组件在哪里。缺点就是当在IDE中打开多个文件时，每个代码文件的文件名为防止重名会变得很长，不太便于切换。

组件的命名

如果组件不以目录的形式存在，而是一个单独的组件，则使用 kebab-case 格式进行命名，例如 user-store.ts 、 user-list.vue 。

在编写组件时，导出的Class的名字需要是 PascalCase 格式的，且语义正确：

```
@Component()
export default class UserList extends Vue {
}
```

导入组件时，也按照 PascalCase 格式进行导入和注册：

```
import UserList from '@/views/user-list/index.vue';

@Component({ components: { UserList } })
export default class UserList extends Vue {
}
```

在模板中使用的时候，需要使用 kebab-case 格式，例如：

```
<template>
  <user-lit />
</template>
```

在单文件组件、字符串模板和JSX中没有内容的组件应该是自闭合的（在DOM模板中不可以）

这样做的好处是符合HTML和JS的语言规范，但是就是当我在组件里想要找到 UserList 时会习惯的直接搜索 Userlist 却搜不到。

变量的命名

变量的命名由ESLint控制，要求使用驼峰拼写法，这也和公司准入平台的要求是一致的。

组件的Prop的命名应该遵循的规则：在声明Prop的时候，命名始终使用 `camelCase` ，在模板中使用 `kebab-case` ：

```
<template>
  <User user-name="Jay" />
</template>

<script lang="ts">
import User from '@/views/user/index.vue';

@Component({ components: { User } )
export default class UserList extends Vue {
  @Prop(String) username!: string;
}
</script>
```

接口的命名

此处指的接口是TypeScript中的 `interface` ，命名应遵循 `PascalCase` 规则：

```
export interface RootState {
  title: string;
}
```

类型

为了充分享受TypeScript带来的强类型的好处，提高项目的可维护性和代码质量，我在项目中开启了 `noImplicitAny` ，即所有的隐式的 `any` 类型都不不被允许的。

所以对于比较复杂的类型、变量建议显示的定义类型，简单的可以令编译器自动推导来确定。

谨慎使用 `any` 类型，对于 `as` 断言的使用也要仔细考虑是否合理。

路由

路由的目录是 `./src/router` ， `index.ts` 是实际进行路由组装的地方，在 `modules` 中按照目录对路由进行了分割代理，分割的维度也可以认为是按照导航的一级目录。在 `router-guards.ts` 中定义路由守卫相关的功能。

按模块组织路由

在 `modules` 中，每个模块是一个.ts文件，具体的业务模块都在这里定义，例如我定义了一个 `base-knowledge.ts` 模块，里面的路由都是与『基础知识』相关的路由定义，路由对象的类型直接使用了 `vue-router` 定义的 `RouteConfig`

要善于利用第三方包已经定义好的类型，具体可以在IDE中按住 `alt` 点击进入其类型声明文件，选择使用。

Route对象的定义与Vue Route需要的路由对象一直，在 `meta` 中定义我们需要自定义的数据，具体的路由守卫都在 `router-guards.ts` 中定义，在此引用。

引入组件的时候，使用了 `lazyLoaderHelper` 辅助函数，在生产环境中会实现路由懒加载，所以引入组件的方式与原来不同：

```
const baseKnowledgeRoutes: RouteConfig[] = [
  {
    path: '/base/hello-vue',
    name: 'base',
    component: lazyLoadHelper('base-knowledge/hello-vue/index'),
    meta: {
      title: 'Hello Vue'
    }
  },
];

export default baseKnowledgeRoutes;
```

传给 `lazyLoadHelper` 参数是组件位于 `./src/views/` 路径下的目录和文件名，不必添加.vue后缀（其实看一下 `lazyLoadHelper` 的实现就明白了），上面导入的组件实际路径是 `./src/view/base-knowledge/hello-vue/index.vue`

这样带来的弊端就是IDE搜索组件引用的时候没有办法直接按照建立的索引搜索到组件的使用，因为我们传入的是字符串形式的路径，所以在搜索组件引用时候需要到这个目录下使用字符串搜索的形式进行搜索\

如果业务复杂，或者需要再嵌套子路由，后续可以在 `modules` 中再换按照目录进行划分，组织形式类似

路由汇总

在 `./src/router/index.ts` 中对各个模块进行汇总，汇总的方式就是采取结构赋值的形式，同时将一些公用的、有特定顺序的路由（例如 `404` ）插入到最终的 `routes` 对象中：

```
const routes: RouteConfig[] = [
  {
    path: '/',
    name: 'home',
    component: lazyLoadHelper('home/index'),
    meta: {
      title: 'Vue Learning Demos'
    }
  },
  ...baseKnowledge,
  ...application,
  {
    path: '*',
    name: 'NotFound',
    component: lazyLoadHelper('common/not-found'),
    meta: {
      title: 'Not Found'
    }
  }
];

const router = new VueRouter({ routes });
```

同时路由守卫也是在这里进行组装的。

导航

设置完路由后，大部分情况下需要在导航菜单中进行处理。项目中关于菜单的组件我放在了 `./src/views/common/menu` 目录下

导航组件是基于 `<el-menu>` 封装的视图组件，基本上不会被其他组件引用，也没有做太多通用性的抽象，所在放到了 `views/common` 中而没有放到 `components` 中。

`menu` 目录下有三个文件：

```
- src
  |
  - views
    |
    - common
      |
      - menu
        |
        - index.vue
        - config.ts
        - types.ts
```

`index.vue` 是导航的主文件，用来将数据映射为视图， `types.ts` 是前面提到的类型文件，而剩下的 `config.ts` 就是视图的数据来源。格式如下：

```
const menuConfigs: MenuConfig[] = [
  {
    path: '/base',
    icon: 'el-icon-location',
    title: '基础知识',
    children: [
      { path: '/base/hello-vue', title: 'Hello Vue', icon: 'el-icon-location' },
      { path: '/base/life-circles', title: 'Life Circles', icon: 'el-icon-location' },
      { path: '/base/inject-and-provide', title: 'Inject/Provide', icon: 'el-icon-location' },
      { path: '/base/mixin-example', title: 'Mixin Example', icon: 'el-icon-location' }
    ]
  },
  {
    path: '/application',
    icon: 'el-icon-basketball',
    title: '综合应用',
    children: [{ path: '/application/todo-list', title: 'Todo List', icon: 'el-icon-basketball' }]
  }
];
```

`menuConfigs` 数组的成员都是一级菜单的属性，成员的 `children` 属性是二级菜单的属性。可以添加哪些属性看 `types.ts` 中的定义即可。如果添加了错误的或者不存在的属性，TypeScript编译器就会报错，不必等到编译成功后在浏览器中看不到预期的结果才发现属性传错，这就是TypeScript静态检查带来的好处之一。

我们的导航组件目录只支持二级导航

Vuex

关于Vuex的代码都在 `.src/store` 目录中，目录分为了三个部分， `index.ts` 组装Store，不负责具体的实现， `root-sotre` 来定义根Store的具体实现， `modules` 中按目录实现各个模块的Store

Store目录划分

在 `root-store` 中定义根Store的具体实现，在 `modules` 里面按目录定义各模块的Store的实现，每个模块目录建议以 `模块名-store` 命名，导出时根节点的Store将各个属性分别导出，插入到 `index.ts` 中 `new Vuex.Store` 的各个属性中，子模块的Stroe直接将模块整体插入到 `new Vuex.Store` 的 `modules` 属性中，属性名（即命名空间）以模块名命名：

```
import Vue from 'vue';
import Vuex from 'vuex';
import RootStore from '@/store/root-store/index';
import TodoStore from '@/store/modules/todo-store/index';

Vue.use(Vuex);

export default new Vuex.Store({
  strict: process.env.NODE_ENV === 'development',
  state: RootStore.state,
  getters: RootStore.getters,
  mutations: RootStore.mutations,
  actions: RootStore.actions,
  modules: {
    todo: TodoStore
  }
});
```

Store的实现

根Store（以及其他每个Module的Store）都应该包含三个文件：

（1） `index.ts` ，用来定义Store的具体实现，例如 `state` 、 `getters` 、 `Mutation` 和 `Action` 等，

（2） `interface-types.ts` 是用来定义Store对应的类型的文件（因为还存在另外一个 `store-types.ts` 文件，所以命名为 `interface-types.ts` ），这里面也大量借用了 `vuex` 提供了类型帮助构建我们自己的类型，具体实现可以参考代码。

（3） `store-types.ts` 用来定义Mutation的Type常量，实际上由于Mutation、Action的Type在Store中、组件中以及上面定义类型的 `interface-types.ts` 中，所以需要在这里定义常量然后导出

另外，由于在之前的项目中，API是单独划分目录存放的，但是实际上目录结构与Store中的结构相同，并且在新项目中我仍然准备所有API请求都都通过Action完成，相当于API与Store也是强耦合的，所以我定义Mutaion和Action的Type常量时，我将对应的URL也作为一个常量保存在了 `store-types.ts` 中。例

如我们有要完成的Action是更新标题，那么在 store-types.ts 中我会做如下的定义：

```
export const UPDATE_TITLE_URL = '/title';
export const UPDATE_TITLE_MUTATION = 'UPDATE_TITLE_MUTATION';
export const UPDATE_TITLE_ACTION = 'UPDATE_TITLE_ACTION';
```

我将上面三个常量成为为一组Type，注意，常量命名应该全大写，以_分割，结尾单词是固定的，以 MUTATION / ACTION / URL 结尾，用来表明类型。多组常量之间应该间隔一个空行。

这一组Type有可能会包含一个URL，但是对应多个Mutaion和Action，例如严格遵守REST风格的API接口是：

```
export const TITLE_URL = '/title';
// 对应 PUT 方法，更新资源
export const UPDATE_TITLE_MUTATION = 'UPDATE_TITLE_MUTATION';
export const UPDATE_TITLE_ACTION = 'UPDATE_TITLE_ACTION';
// 对应 POST 方法，创建资源
export const CREATE_TITLE_MUTATION = 'UPDATE_TITLE_MUTATION';
export const CREATE_TITLE_ACTION = 'UPDATE_TITLE_ACTION';
// 对应 GET 方法，获取资源
export const GET_TITLE_MUTATION = 'UPDATE_TITLE_MUTATION';
export const GET_TITLE_ACTION = 'UPDATE_TITLE_ACTION';
```

想到过采用一个对象里面多个属性的方式来维护，例如：

```
export const UPDATE_TITLE = {
  mutation: 'UPDATE_TITLE_MUTATION',
  action: 'UPDATE_TITLE_ACTION',
  url: 'title'
};
```

但是在 interface-types.ts 中用这个变量来定义对应的Mutaion或者Action的类型时会报错：

```
export interface RootMutations extends MutationTree<RootState> {
  [UPDATE_TITLE.action]: RootUpdateTitleMutation;
}
// Error:(16, 3) TS1169: A computed property name in an interface must refer
// to an expression whose type is a literal type or a 'unique symbol' type.
```

就是说如果对象的属性名是一个用 [] 包裹的计算属性，那么计算属性只能是字面量或者是 unique symbol 类型，字面量就是我现在才去的方案，而如果要使用 unique symbol 又遇到了Vuex不支持Symbol类型作为Mutation Type的限制，所以只能采取了上面的方案。

使用常量的时候直接将常量导入，属性名使用 [] 的形式来插入常量：

```
import { UPDATE_TITLE_MUTATION, UPDATE_TITLE_ACTION } from '@/store/root-store/store-types';

const mutations: RootMutations = {
  [UPDATE_TITLE_MUTATION](state, { title }) {
    state.title = title;
  }
};

const actions: RootActions = {
  async [UPDATE_TITLE_ACTION]({ commit }) {
    const result: string = await setTimeoutThen(1000, 'ok');
    commit(UPDATE_TITLE_MUTATION, { title: result });
  }
};

export default { state, getters, mutations, actions };
```

什么数据需要存到Store中

需要共享的全局数据自不必说，需要存到Store中，而且一般要放到 root-store 中

对于组件中的数据，如果组件内有多个子组件共享数据建议也放到 Store 中的state中进行维护，如果父组件获取数据不全仍需要子组件通过网络请求获取数据的情况，不建议放到Store中。

例如，有一个组件 UserList ，保存了用户数据列表， User 是子组件，点击 UserList 其中一列会打开 User 组件，展示用户详情，如果：

`User` 展示的详情数据在 `UserList` 中已经完全具备， `User` 不需要再通过网络请求获取数据，那么 `UserList` 的数据建议放到Store中，在Action中将数据 `commit` 到 `state` 中
如果情况相反，则不建议将 `UserList` 的数据建议放到Store中，不需要 `commit` ，直接 `return` 返回的数据

缓存

对于 `root-store` 中的数据应该有全局缓存的功能，如果请求的数据短期内不变，就不必再次发送网络请求，简单做的话就是对 `root-store` 中的每个请求进行判断，如果State中数据已经存在则直接返回该数据

这样做需要每个请求单独处理，而且实现的功能比较简陋，比较完善的全局缓存我能想到的是：

不需要每个请求（即 `dispatch` 每个Action）时都需要处理，那么也许将缓存这一部分与Axios的封装结合在一起比较好
为缓存添加可控制的属性，比如最大缓存时间、强制忽略缓存等
缓存控制的粒度，是以请求为粒度，还是URL作为更细的控制粒度?
当多个请求请求同一份资源的时候，后续请求如何处理？是直接忽略缓存发送多个请求（简单粗暴）？还是使用资源池配合监听订阅模式实现效率最大化（精致复杂）？
这一块我还没有进行处理，后面有了具体的实践经验再来补充。

网络请求处理

Axios的封装

项目的网络工具选了最主流的Axios，在 `utiles/network-helper` 中进行了处理，这个目录下除了 `types.ts` 是类型声明文件， `index.ts` 是具体实现的主文件， `loading-counter.ts` 是用来对全局Loading进行处理的方法。

对Axios的封装其实还是老一套，只是没有直接导出的 `get` 等方法，而是作为 `request` 的一个属性，按照 `request.get` 的形式使用。

在 `index.ts` 也添加了一些Interceptors，目前主要添加的功能时错误提示、Loading处理，后面会加上对请求参数自动拼装（例如JWT参数）。后续如果Interceptors比较多，可以单独拆分文件维护。

要注意的是，对错误的拦截，如果是网络异常导致的错误，拦截器会进行提示后，将错误对象 `reject` ，在 `main.ts` 的 `Vue.config.errorHandler` 统一处理，如果是业务返回的错误码，会在拦截器中进行提示，错误对象会 `resolve` ，响应结果在业务代码中处理（单不需要处理Loading和提示错误了）

全局Loading

`loading-counter.ts` 中定义了 `LoadingCounter` 这个类，并导出了一个实例，在Axios的成功的请求拦截器中Loading数 `+1` ，失败的请求拦截器不作处理，在成功或失败的响应拦截器中Loading数 `-1` ，并提供了 `clearLoading` 方法强制清除Loading，

这个方案的优点就是不需要再网络请求时反复的添加Loading的处理逻辑，缺点就是只能默认添加全屏的Loading，而且实现比较简单，没有经过复杂项目的验证，可能由于欠考虑导致什么潜在的问题。

Mock数据

前端Mock数据选择了使用Yapi来完成（内网地址），让前端编写Mock数据能够更加轻松，它采用了Mock.js的语法，也可以自己编写Mock脚本来实现复杂数据的输出。

其实它完全可以充当API的文档，不过需要后端的配合，一般实现起来是比较难的，所以可以前端自己来维护一份Mock的API。

理想状态时，在本地开发时，所有请求都应该是通过Mock完成的，这样有一些数据在测试环境也没有办法获取到真实反映（例如车辆接管数据，需要同步到Monitor，所以后端只返回一种结果），但是在Mock数据里我们就可以自由控制。

样式

UI组件

项目的UI组件仍然选择了ElementUI，对于ElementUI的引入应该采取按需引入的方式，尽可能减少构建后的体积。可以直接使用Element为Vue Cli@3提供的插件，它会帮助我们完成ElementUI的按需引入。

如果是手动安装的话需要安装 `babel-plugin-component` 使用，安装之后在根目录下新建 `.babel.config.js` 文件（如果Vue CLI已经创建的话可以直接使用），然后在 `plugins` 中添加 `element-ui` 的使用）

然后在 `/src/plugins/elemment.ts` 中引入需要的Element组件，并将需要的方法（比如 `$loading` 挂载到Vue的原型上（不推荐挂载大量的属性到Vue原型上，会导致性能的下降）。

同时，ElementUI提供了主题定制的功能，详细使用方法参考文档。完成主题定制后，下载定制后的文件放到根目录下新建的 `theme` 目录内，然后在 `.babel.config.js` 添加 `styleLibraryName` 属性，指向 `theme` 目录， `.babel.config.js` 完整的配置如下：

```
module.exports = {
  "presets": [
    "@vue/cli-plugin-babel/preset"
  ],
  "plugins": [
    [
      "component",
      {
        "libraryName": "element-ui",
        "styleLibraryName": "~theme"
      }
    ]
  ]
};
```

在开发类似CRM的后台项目时，如果对样式有要求，尽量提前与UE/UI同学进行沟通，基于ElementUI的组件规范进行样式定制，尽量避免通过覆盖样式的方式来修改ElementUI的内置组

样式规范

前面提到了，项目会使用Stylelint对样式的编写进行规范，同时还有一些规范要遵守：

在Vue单文件组件内，除非极特殊的情况， 都需要使用scoped属性 ，避免组件样式成为全局样式，污染全局样式，并且导致在开发或编译时组件间的样式相互干扰。
如果没有添加 `scoped` 属性，那么意味着你明确知晓并且意图将这个样式继承给子组件，但是前提是为这些样式添加了一个足够独一无二的类名。
另外，除非UI组件的要求，不允许使用内联样式
不允许使用ID选择器编写样式
除非极特殊情况，不允许使用 `!important`
`class` 命名使用 `kebab-case` 格式，例如 `user-list-item`

关于变量名：无论是CSS的类名还是JavaScript的变量名，在遵守格式要求的基础上（ `PascalCase` 或者 `kebab-case`)尽可能传达出有效的信息，想 `value1` 、 `value2` 这样没有任何意义的命名要避免（除非是在具体的回调函数中有具体的上下文环境），使用带有足够信息量的变量名提高代码的可读性。使用英文命名，不要使用拼音。

另外，避免（！）语意不明或者是错误的缩写。

自动导入全局样式变量

样式预处理器选择使用了Less，在 `/src/styles` 下面声明了一些全局的样式文件，目前有三个：

`reset.css` ，用来重置浏览器默认样式

`mixins.less` ，用来实现一些可以传入参数的样式模块，比如文字剪切等，直接在组件样式中调用

`variables.less` ，用来定义一些全局的样式变量，比如主题颜色、边框颜色等等

对于后两者，使用了 `style-resources-loader` ，让我们不需要手动导入这些全局的样式文件。需要在 `vue.config.js` 中进行配置：

```
const path = require('path');
const StyleLintPlugin = require('stylelint-webpack-plugin');

module.exports = {
  chainWebpack: config => {
    // 自动导入样式文件
    const types = ['vue-modules', 'vue', 'normal-modules', 'normal'];
    types.forEach(type => addStyleResource(config.module.rule('less').oneOf(type)));
  },
};

function addStyleResource (rule) {
  rule.use('style-resource')
    .loader('style-resources-loader')
    .options({
      patterns: [
        path.resolve(__dirname, './src/styles/variables.less'),
        path.resolve(__dirname, './src/styles/mixins.less'),
      ],
    })
}
```

如果后续需要导入更多的样式变量，那么只需要在 `addStyleResource` 的我 `patterns` 数组中添加对应的路径即可。

5.10 环境变量和构建脚本

根据不同的环境新建了几个 `.env` 文件，例如我们有线上（ `production` ）/开发（ `development` ）/测试（ `staging` ）三个环境，在各自的环境文件中（例如 `.env.production` ）配置 `VUE_BASE_URL` 等变量。

依赖管理工具选择了使用Yarn代替NPM， 在`package.json` 中的脚本除了常规的 `serve` 、 `build` 之外，还未编译的时候也创建对应的脚本，

```
"scripts": {
  "serve": "vue-cli-service serve --open",
  "build": "vue-cli-service build --modern",
  "build-dev": "vue-cli-service build --modern --mode development",
  "build-staging": "vue-cli-service build --modern --mode staging",
  "test:unit": "vue-cli-service test:unit",
  "lint": "vue-cli-service lint",
  "analyze": "vue-cli-service build --modern --report",
  "preview": "cd dist && npx http-server -a 127.0.0.1 -p 7000"
},
```

另外，配置了 `pre-commit` 的钩子，在每次提交前都会对改动的文件进行代码格式和规范的检查，不通过本地也无法提交，这样就缩短了代码质量的反馈链，不必等到Icode检查半天才告诉你不符合规范

Code Reivew

Code Reivew是一个可以提升团队代码质量的手段，更重要的是，它也可以提高自己的代码水平，无论是你Review他人的代码，还是被别人Review代码。

关于Code Reivew，提交代码时有如下几点建议：

分节点提交代码（开始写之前做好规划），每次提交Reivew的代码不要过多

分批次Commit

清晰的Commit Message

Review他人代码时：

尽量抽出时间Review别人的代码

了解需求详情

首先关注高层次问题（例如接口设计、函数分解等）

多打-2

Axios + Vuex的全局缓存

前面提到了，需要考虑的问题：

不需要每个请求（即 `dispatch` 每个Action）时都需要处理，那么也许将缓存这一部分与Axios的封装结合在一起比较好

为缓存添加可控制的属性，比如最大缓存时间、强制忽略缓存等

缓存控制的粒度，是以请求为粒度，还是URL作为更细的控制粒度？

当多个请求请求同一份资源的时候，后续请求如何处理？是直接忽略缓存发送多个请求（简单粗暴）？还是使用资源池配合监听订阅模式实现效率最大化（精致复杂）？

接口超时自动重试

除了前面提到的，与Vuex的全局缓存相结合的问题，有时间再研究下接口超时自动重试的方案。

其实在JS版本的Vue项目中已经实现了接口自动重试的方案，但是TS版本中没有办法向Axios的配置对象中传入自定义的参数，所以不能自由的控制是否开启接口的自动重试，所以不太完美，而且这个需求也是低优的，再仔细考虑一下

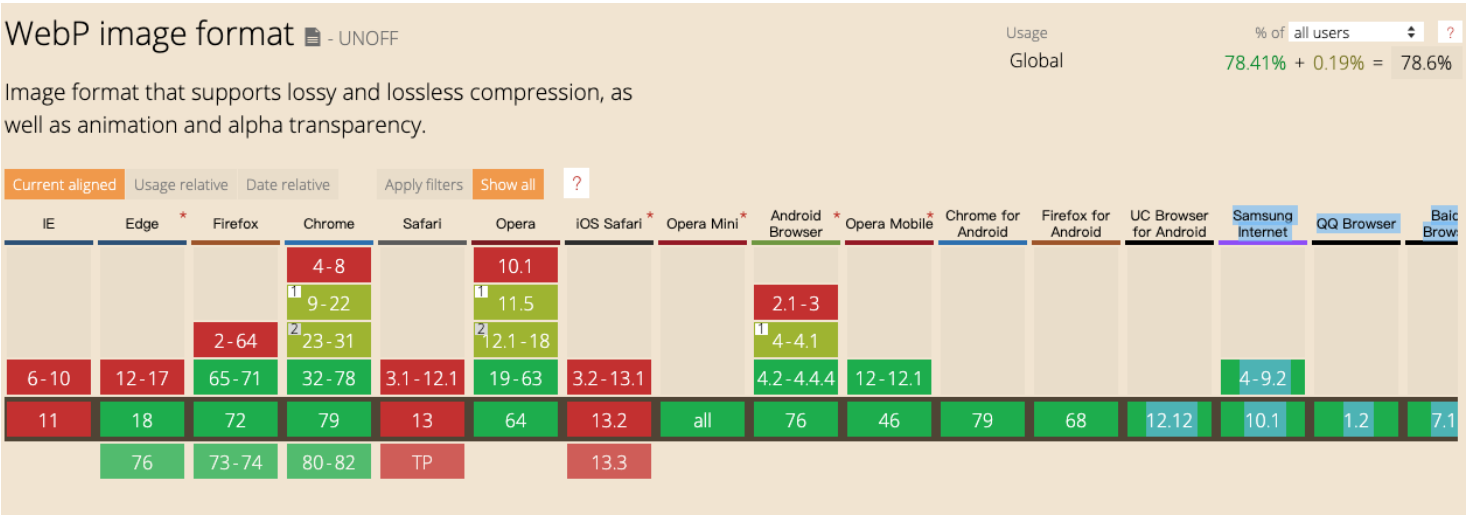
找到了解决方案，就是利用TypeScript的声明合并，为Axios的AxiosRequestConfig添加自定义的属性。具体解决方法时在根目录下创建一个shims-cunstom.d.ts文件，里面用来定制第三方模块的类型

```
import * as axios from 'axios';


declare module 'axios' {
  export interface AxiosRequestConfig {
    customProperty?: boolean;
  }
}

declare module '*.png';
declare module '*.jpg';
```

如果没有第一行， `axios.d.ts` 中定义的接口就不会被合并进来，细节参考TypeScript的文档吧

WebP图片引入

WebP是Google推出的图片格式，相比于PNG、JPG能够在保证图片质量的同时，大幅减小图片的体积，兼容性如下：



IE系列和Safari系列的兼容性需要处理，所以需要在引入WebP的同时实现优雅降级。

异常监控和上报

图片压缩

CLI工具

将上面基本的配置和实践抽离成为了一个模板vue-ts-template，预置的功能可以看项目的介绍。使用的时候可以直接将这个仓库 `clone` 下来使用。

为了更方便的使用，作者以前的博客，做了一个简单的CLI工具vue-ts-cli，并且发布到了NPM上，可以更方便的拉取模板。

使用方法（推荐使用 `npx` 工具，需要 `npm` 版本高于5.2）：

```
npx vue-ts-cli-zh init [project_name]
```

`init` 跟着的 `project_name` 就是新创建的项目所在的目录名。

其他