代码规范: 如何利用 Lint 工具链来保证代码风格和质量?

发布于 2022-05-09

在上一节的内容中，我们详细讲解了如何在 Vite 中实现 CSS 工程化方案，迈出了搭建脚手架项目的第一步。本节，让我们把目光投向项目中另外一个非常重要的部分——**代码规范**。

> 代码不仅是让机器看的，它也是给人看的。

在真实的工程项目中，尤其是多人协作的场景下，代码规范就变得非常重要了，它可以用来统一团队代码风格，避免不同风格的代码混杂到一起难以阅读，有效提高**代码质量**，甚至可以将一些**语法错误**在开发阶段提前规避掉。但仅有规范本身不够，我们需要**自动化的工具**(即 `Lint 工具`)来保证规范的落地，把代码规范检查(包括 `自动修复`)这件事情交给机器完成，开发者只需要专注应用逻辑本身。

本节，我们将一起来完成 Lint 工具链在项目中的落地，实现自动化代码规范检查及修复的能力。学完本节内容后，你不仅能熟悉诸如 `ESLint` 、 `Prettier` 、 `Stylelint` 和 `Commitlint` 等诸多主流 Lint 工具的概念和使用，还能配合 `husky` 、 `lint-staged` 、 `VSCode 插件` 和 `Vite 生态` 在项目中集成完整的 Lint 工具链，搭建起完整的前端开发和代码提交工作流，这部分内容虽然和 Vite 没有直接的联系，但也是 Vite 项目搭建中非常重要的一环，是前端工程化的必备知识。

> 小节示例项目仓库: 点击直达

JS/TS 规范工具: ESLint

简介

> ESLint 是在 ECMAScript/JavaScript 代码中识别和报告模式匹配的工具，它的目标是保证代码的一致性和避免错误。

Eslint 是国外的前端大牛 `Nicholas C. Zakas` 在 2013 年发起的一个开源项目，有一本书被誉为前端界的"圣经"，叫《JavaScript 高级程序设计》(即红宝书)，他正是这本书的作者。

`Nicholas` 当初做这个开源项目，就是为了打造一款插件化的 JavaScript 代码静态检查工具，通过解析代码的 AST 来分析代码格式，检查代码的风格和质量问题。现在，Eslint 已经成为一个非常成功的开源项目了，基本上属于前端项目中 Lint 工具的标配。

ESLint 的使用并不复杂，主要通过配置文件对各种代码格式的规则(`rules`)进行配置，以指定具体的代码规范。目前开源社区也有一些成熟的规范集可供使用，著名的包括 Airbnb JavaScript 代码规范、Standard JavaScript 规范、Google JavaScript 规范等等，你可以在项目中直接使用这些成熟的规范，也可以自己定制一套团队独有的代码规范，这在一些大型团队当中还是很常见的。

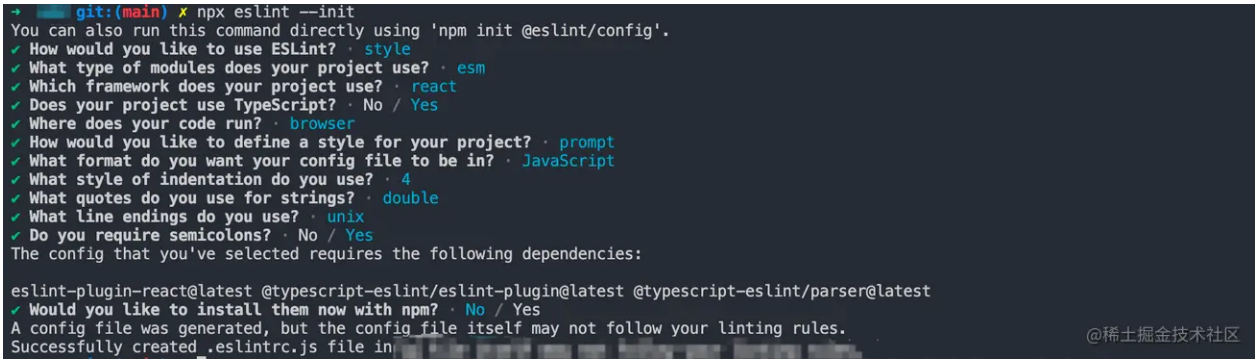
初始化

接下来我们来利用 ESLint 官方的 cli 在现有的脚手架项目中进行初始化操作，首先我们需要安装 ESLint:

```
pnpm i eslint -D
```

接着执行 ESLint 的初始化命令，并进行如下的命令行交互:

```
npx eslint --init
```



接着 ESLint 会帮我们自动生成 `.eslintrc.js` 配置文件。需要注意的是，在上述初始化流程中我们并没有用 npm 安装依赖，需要进行手动安装:

```
pnpm i eslint-plugin-react@latest @typescript-eslint/eslint-plugin@latest @typescript-eslint/
```

核心配置解读

大家初次接触配置文件可能会有点不太理解，接下来我来为你介绍一下几个核心的配置项，你可以对照目前生成的 `.eslintrc.js` 一起学习。

1. parser - 解析器

ESLint 底层默认使用 Espree来进行 AST 解析，这个解析器目前已经基于 `Acron` 来实现，虽然说 `Acron` 目前能够解析绝大多数的 ECMAScript 规范的语法，但还是不支持 TypeScript ，因此需要引入其他的解析器完成 TS 的解析。

社区提供了 `@typescript-eslint/parser` 这个解决方案，专门为了 TypeScript 的解析而诞生，将 `TS` 代码转换为 `Espree` 能够识别的格式(即 **Estree 格式**)，然后在 Eslint 下通过 `Espree` 进行格式检查， 以此兼容了 TypeScript 语法。

2. parserOptions - 解析器选项

这个配置可以对上述的解析器进行能力定制，默认情况下 ESLint 支持 ES5 语法，你可以配置这个选项，具体内容如下:

- ecmaVersion: 这个配置和 `Acron` 的 ecmaVersion 是兼容的，可以配置 `ES + 数字` (如 ES6)或者 `ES + 年份` (如 ES2015)，也可以直接配置为 `latest` ，启用最新的 ES 语法。

- sourceType: 默认为 `script` ，如果使用 ES Module 则应设置为 `module`
- ecmaFeatures: 为一个对象，表示想使用的额外语言特性，如开启 `jsx` 。

3. rules - 具体代码规则

`rules` 配置即代表在 ESLint 中手动调整哪些代码规则，比如 `禁止在 if 语句中使用赋值语句` 这条规则可以像如下的方式配置:

```
// .eslintrc.js
module.exports = {
  // 其它配置省略
  rules: {
    // key 为规则名，value 配置内容
    "no-cond-assign": ["error", "always"]
  }
}
```

在 rules 对象中， `key` 一般为 `规则名` ， `value` 为具体的配置内容，在上述的例子中我们设置为一个数组，数组第一项为规则的 `ID` ，第二项为 `规则的配置` 。

这里重点说一说规则的 ID，它的语法对所有规则都适用，你可以设置以下的值:

- `off` 或 `0` : 表示关闭规则。
- `warn` 或 `1` : 表示开启规则，不过违背规则后只抛出 warning，而不会导致程序退出。
- `error` 或 `2` : 表示开启规则，不过违背规则后抛出 error，程序会退出。

具体的规则配置可能会不一样，有的是一个字符串，有的可以配置一个对象，你可以参考 ESLint 官方文档。

当然，你也能直接将 `rules` 对象的 `value` 配置成 ID，如: `"no-cond-assign": "error"` 。

4. plugins

上面提到过 ESLint 的 parser 基于 `Acorn` 实现，不能直接解析 TypeScript，需要我们指定 parser 选项为 `@typescript-eslint/parser` 才能兼容 TS 的解析。同理，ESLint 本身也没有内置 TypeScript 的代码规则，这个时候 ESLint 的插件系统就派上用场了。我们需

要通过添加 ESLint 插件来增加一些特定的规则，比如添加 `@typescript-eslint/eslint-plugin` 来拓展一些关于 TS 代码的规则，如下代码所示:

```
// .eslintrc.js
module.exports = {
  // 添加 TS 规则，可省略`eslint-plugin`
  plugins: ['@typescript-eslint']
}
```

值得注意的是，添加插件后只是拓展了 ESLint 本身的规则集，但 ESLint 默认并**没有开启**这些规则的校验！如果要开启或者调整这些规则，你需要在 rules 中进行配置，如:

```
// .eslintrc.js
module.exports = {
  // 开启一些 TS 规则
  rules: {
    '@typescript-eslint/ban-ts-comment': 'error',
    '@typescript-eslint/no-explicit-any': 'warn',
  }
}
```

5. extends - 继承配置

extends 相当于 继承 另外一份 ESLint 配置，可以配置为一个字符串，也可以配置成一个字符串数组。主要分如下 3 种情况:

从 ESLint 本身继承;
从类似 `eslint-config-xxx` 的 npm 包继承;
从 ESLint 插件继承。

```
// .eslintrc.js
module.exports = {
   "extends": [
     // 第1种情况
     "eslint:recommended",
     // 第2种情况，一般配置的时候可以省略 `eslint-config`
     "standard"
     // 第3种情况，可以省略包名中的 `eslint-plugin`
     // 格式一般为: `plugin:${pluginName}/${configName}`
     "plugin:react/recommended"
     "plugin:@typescript-eslint/recommended",
   ]
}
```

有了 extends 的配置，对于之前所说的 ESLint 插件中的繁多配置，我们就**不需要手动一一开启**了，通过 extends 字段即可自动开启插件中的推荐规则:

```
extends: ["plugin:@typescript-eslint/recommended"]
```

6. env 和 globals

这两个配置分别表示 运行环境 和 全局变量 ，在指定的运行环境中会预设一些全局变量，比如:

```
// .eslint.js
module.export = {
  "env": {
    "browser": "true",
    "node": "true"
  }
}
```

指定上述的 `env` 配置后便会启用浏览器和 Node.js 环境，这两个环境中的一些全局变量(如 `window` 、 `global` 等)会同时启用。

有些全局变量是业务代码引入的第三方库所声明，这里就需要在 `globals` 配置中声明全局变量了。每个全局变量的配置值有 3 种情况:

- `"writable"` 或者 `true` ，表示变量可重写;
- `"readonly"` 或者 `false` ，表示变量不可重写;
- `"off"` ，表示禁用该全局变量。

那 `jquery` 举例，我们可以在配置文件中声明如下:

```
// .eslintrc.js
module.exports = {
  "globals": {
    // 不可重写
    "$": false,
    "jQuery": false
  }
}
```

相信有了上述核心配置部分的讲解，你再回头看看初始化生成的 ESLint 配置文件，你也能很好地理解各个配置项的含义了。

与 Prettier 强强联合

虽然 ESLint 本身具备自动格式化代码的功能(`eslint --fix`)，但术业有专攻，ESLint 的主要优势在于 `代码的风格检查并给出提示` ，而在代码格式化这一块 Prettier 做的更加专业，因此我们经常将 ESLint 结合 Prettier 一起使用。

首先我们来安装一下 Prettier:

```
pnpm i prettier -D
```

在项目根目录新建 `.prettierrc.js` 配置文件，填写如下的配置内容:

```
// .prettierrc.js
module.exports = {
  printWidth: 80, //一行的字符数，如果超过会进行换行，默认为80
  tabWidth: 2, // 一个 tab 代表几个空格数，默认为 2 个
  useTabs: false, //是否使用 tab 进行缩进，默认为false，表示用空格进行缩减
  singleQuote: true, // 字符串是否使用单引号，默认为 false，使用双引号
  semi: true, // 行尾是否使用分号，默认为true
  trailingComma: "none", // 是否使用尾逗号
  bracketSpacing: true // 对象大括号直接是否有空格，默认为 true，效果：{ a: 1 }
};
```

接下来我们将 `Prettier` 集成到现有的 `ESLint` 工具中，首先安装两个工具包:

```
pnpm i eslint-config-prettier eslint-plugin-prettier -D
```

其中 `eslint-config-prettier` 用来覆盖 ESLint 本身的规则配置，而 `eslint-plugin-prettier` 则是用于让 Prettier 来接管 `eslint --fix` 即修复代码的能力。

在 `.eslintrc.js` 配置文件中接入 prettier 的相关工具链，最终的配置代码如下所示，你可以直接粘贴过去:

```
// .eslintrc.js
module.exports = {
  env: {
    browser: true,
    es2021: true
  },
  extends: [
    "eslint:recommended",
    "plugin:react/recommended",
    "plugin:@typescript-eslint/recommended",
    // 1. 接入 prettier 的规则
```

```
    "prettier",
    "plugin:prettier/recommended"
  ],
  parser: "@typescript-eslint/parser",
  parserOptions: {
    ecmaFeatures: {
      jsx: true
    },
    ecmaVersion: "latest",
    sourceType: "module"
  },
  // 2. 加入 prettier 的 eslint 插件
  plugins: ["react", "@typescript-eslint", "prettier"],
  rules: {
    // 3. 注意要加上这一句，开启 prettier 自动修复的功能
    "prettier/prettier": "error",
    quotes: ["error", "single"],
    semi: ["error", "always"],
    "react/react-in-jsx-scope": "off"
  }
};
```

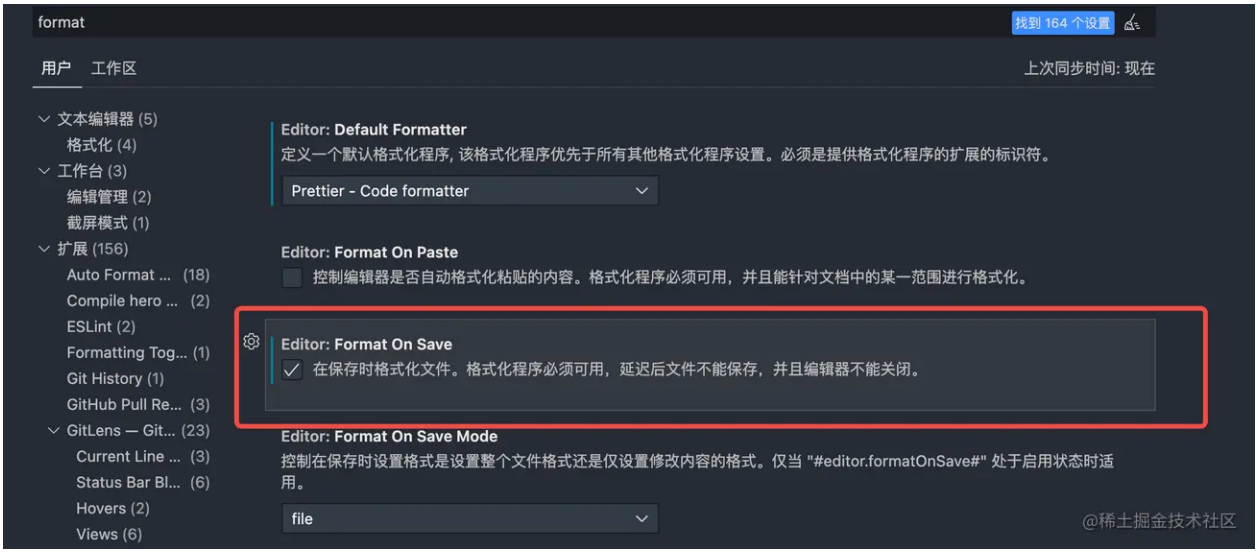
OK，现在我们回到项目中来见证一下 ESLint + Prettier 强强联合的威力，在 package.json 中定义一个脚本:

```
{
  "scripts": {
    // 省略已有 script
    "lint:script": "eslint --ext .js,.jsx,.ts,.tsx --fix --quiet ./",
  }
}
```

接下来在命令行终端执行:

```
pnpm run lint:script
```

这样我们就完成了 ESLint 的规则检查 以及 Prettier 的自动修复 。不过每次执行这个命令未免会有些繁琐，我们可以在 VSCode 中安装 ESLint 和 Prettier 这两个插件，并且在设置区中开启 Format On Save：



接下来在你按 `Ctrl + S` 保存代码的时候，Prettier 便会自动帮忙修复代码格式。

在 Vite 中接入 ESLint

除了安装编辑器插件，我们也可以通过 Vite 插件的方式在开发阶段进行 ESLint 扫描，以命令行的方式展示出代码中的规范问题，并能够直接定位到原文件。

首先我们安装 Vite 中的 ESLint 插件:

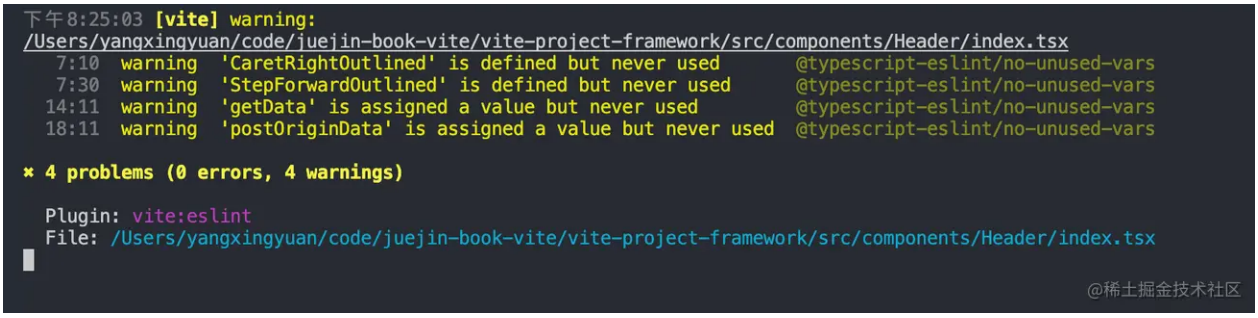
```
pnpm i vite-plugin-eslint -D
```

然后在 `vite.config.ts` 中接入:

```
// vite.config.ts
import viteEslint from 'vite-plugin-eslint';

// 具体配置
{
  plugins: [
    // 省略其它插件
    viteEslint(),
  ]
}
```

现在你可以试着重新启动项目， ESLint 的错误已经能够及时显示到命令行窗口中了。



> 由于这个插件采用另一个进程来运行 ESLint 的扫描工作，因此不会影响 Vite 项目的启动速度，这个大家不用担心。

样式规范工具: Stylelint

接下来我们进入 `Stylelint` 的部分，先来看看官方的定义:

> Stylelint，一个强大的现代化样式 Lint 工具，用来帮助你避免语法错误和统一代码风格。

Stylelint 主要专注于样式代码的规范检查，内置了 **170 多个 CSS 书写规则**，支持 **CSS 预处理器**(如 Sass、Less)，提供**插件化机制**以供开发者扩展规则，已经被 Google、Github 等**大型团队**投入使用。与 ESLint 类似，在规范检查方面，Stylelint 已经做的足够专业，而在代码格式化方面，我们仍然需要结合 Prettier 一起来使用。

首先让我们来安装 Stylelint 以及相应的工具套件:

```
pnpm i stylelint stylelint-prettier stylelint-config-prettier stylelint-config-recess-order s
```

然后，我们在 Stylelint 的配置文件 `.stylelintrc.js` 中一一使用这些工具套件:

```
// .stylelintrc.js
module.exports = {
  // 注册 stylelint 的 prettier 插件
  plugins: ['stylelint-prettier'],
  // 继承一系列规则集合
  extends: [
    // standard 规则集合
    'stylelint-config-standard',
    // standard 规则集合的 scss 版本
    'stylelint-config-standard-scss',
```

```
    // 样式属性顺序规则
    'stylelint-config-recess-order',
    // 接入 Prettier 规则
    'stylelint-config-prettier',
    'stylelint-prettier/recommended'
  ],
  // 配置 rules
  rules: {
    // 开启 Prettier 自动格式化功能
    'prettier/prettier': true
  }
};
```

可以发现 Stylelint 的配置文件和 ESLint 还是非常相似的，常用的 `plugins` 、 `extends` 和 `rules` 属性在 ESLint 同样存在，并且与 ESLint 中这三个属性的功能也基本相同。不过需要强调的是在 Stylelint 中 rules 的配置会和 ESLint 有些区别，对于每个具体的 rule 会有三种配置方式:

- `null` ，表示关闭规则。
- 一个简单值(如 true，字符串，根据不同规则有所不同)，表示开启规则，但并不做过多的定制。
- 一个数组，包含两个元素，即 `[简单值，自定义配置]` ，第一个元素通常为一个简单值，第二个元素用来进行更精细化的规则配置。

接下来我们将 Stylelint 集成到项目中，回到 `package.json` 中，增加如下的 `scripts` 配置:

```
{
  "scripts": {
    // 整合 lint 命令
    "lint": "npm run lint:script && npm run lint:style",
    // stylelint 命令
    "lint:style": "stylelint --fix \"src/**/*.{css,scss}\""
  }
}
```

执行 `pnpm run lint:style` 即可完成样式代码的规范检查和自动格式化。当然，你也可以在 VSCode 中安装 `Stylelint` 插件，这样能够在开发阶段即时感知到代码格式问题，提前进行修复。

当然，我们也可以直接在 Vite 中集成 Stylelint。社区中提供了 Stylelint 的 Vite 插件，实现在项目开发阶段提前暴露出样式代码的规范问题。我们来安装一下这个插件:

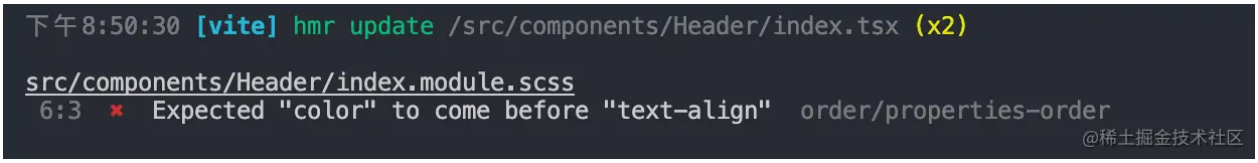
```
pnpm i @amatlash/vite-plugin-stylelint -D
```

然后在 Vite 配置文件中添加如下的内容:

```
import viteStylelint from '@amatlash/vite-plugin-stylelint';

// 具体配置
{
  plugins: [
    // 省略其它插件
    viteStylelint({
      // 对某些文件排除检查
      exclude: /windicss|node_modules/
    }),
  ]
}
```

接下来，你就可以在命令行界面看到对应的 Stylelint 提示了:



Husky + lint-staged 的 Git 提交工作流集成

提交前的代码 Lint 检查

在上文中我们提到了安装 `ESLint` 、 `Prettier` 和 `Stylelint` 的 VSCode 插件或者 Vite 插件，在开发阶段提前规避掉代码格式的问题，但实际上这也只是将问题提前暴露，并不能保证规范问题能完全被解决，还是可能导致线上的代码出现不符合规范的情况。那么如何来避免这类问题呢?

我们可以在代码提交的时候进行卡点检查，也就是拦截 `git commit` 命令，进行代码格式检查，只有确保通过格式检查才允许正常提交代码。社区中已经有了对应的工具——`Husky` 来完成这件事情，让我们来安装一下这个工具:

```
pnpm i husky -D
```

值得提醒的是，有很多人推荐在 `package.json` 中配置 husky 的钩子:

```
// package.json
{
  "husky": {
    "pre-commit": "npm run lint"
  }
}
```

这种做法在 Husky `4.x` 及以下版本没问题，而在最新版本(7.x 版本)中是无效的！在新版 Husky 版本中，我们需要做如下的事情:

初始化 Husky: `npx husky install` ，并将 `husky install` 作为项目启动前脚本，如:

```
{
  "scripts": {
    // 会在安装 npm 依赖后自动执行
    "postinstall": "husky install"
  }
}
```

添加 Husky 钩子，在终端执行如下命令:

```
npx husky add .husky/pre-commit "npm run lint"
```

接着你将会在项目根目录的 `.husky` 目录中看到名为 `pre-commit` 的文件，里面包含了 `git commit` 前要执行的脚本。现在，当你执行 `git commit` 的时候，会首先执行 `npm run lint` 脚本，通过 Lint 检查后才会正式提交代码记录。

不过，刚才我们直接在 Husky 的钩子中执行 `npm run lint` ，这会产生一个额外的问题: Husky 中每次执行 `npm run lint` 都对仓库中的代码进行全量检查，也就是说，即使某些文件并没有改动，也会走一次 Lint 检查，当项目代码越来越多的时候，提交的过程会越来越慢，影响开发体验。

而 `lint-staged` 就是用来解决上述全量扫描问题的，可以实现只对存入 暂存区 的文件进行 Lint 检查，大大提高了提交代码的效率。首先，让我们安装一下对应的 npm 包:

```
pnpm i -D lint-staged
```

然后在 `package.json` 中添加如下的配置:

```
{
  "lint-staged": {
    "**/*.{js,jsx,tsx,ts}": [
```

```
      "npm run lint:script",
      "git add ."
    ],
    "**/*.{scss}": [
      "npm run lint:style",
      "git add ."
    ]
  }
}
```

接下来我们需要在 Husky 中应用 `lint-stage`，回到 `.husky/pre-commit` 脚本中，将原来的 `npm run lint` 换成如下脚本:

```
npx --no -- lint-staged
```

如此一来，我们便实现了提交代码时的 `增量 Lint 检查` 。

提交时的 commit 信息规范

除了代码规范检查之后，Git 提交信息的规范也是不容忽视的一个环节，规范的 commit 信息能够方便团队协作和问题定位。首先我们来安装一下需要的工具库，执行如下的命令:

```
pnpm i commitlint @commitlint/cli @commitlint/config-conventional -D
```

接下来新建 `.commitlintrc.js`：

```
// .commitlintrc.js
module.exports = {
  extends: ["@commitlint/config-conventional"]
};
```

一般我们直接使用 `@commitlint/config-conventional` 规范集就可以了，它所规定的 commit 信息一般由两个部分: `type` 和 `subject` 组成，结构如下:

```
// type 指提交的类型
// subject 指提交的摘要信息
<type>: <subject>
```

常用的 `type` 值包括如下:

- `feat`：添加新功能。
- `fix`：修复 Bug。
- `chore`：一些不影响功能的更改。
- `docs`：专指文档的修改。
- `perf`：性能方面的优化。
- `refactor`：代码重构。
- `test`：添加一些测试代码等等。

接下来我们将 `commitlint` 的功能集成到 Husky 的钩子当中，在终端执行如下命令即可:

```
npx husky add .husky/commit-msg "npx --no-install commitlint -e $HUSKY_GIT_PARAMS"
```

你可以发现在 `.husky` 目录下多出了 `commit-msg` 脚本文件，表示 `commitlint` 命令已经成功接入到 husky 的钩子当中。现在我们可以尝试对代码进行提交，假如输入一个错误的 commit 信息，commitlint 会自动抛出错误并退出:

至此，我们便完成了 Git 提交信息的卡点扫描和规范检查。

小结

恭喜你，学完了本节的内容。本小节你应该了解前端的**自动化代码规范工具的使用**以及**在 Vite 中的接入方法**。

我主要给你介绍了 3 个方面的自动化代码规范工具:

> JavaScript/TypeScript 规范。主流的 Lint 工具包括 `Eslint` 、 `Prettier` ；
> 样式开发规范。主流的 Lint 工具包括 `Stylelint` 、 `Prettier` ；
> Git 提交规范。主流的 Lint 工具包括 `Commitlint` 。

我们可以通过编辑器的插件或者 Vite 插件在开发阶段暴露出规范问题，但也无法保证这类问题在开发时完全被解决掉，因此我们尝试在代码提交阶段来解决这个问题，通过

`Husky` + `lint-staged` 成功地拦截 `git commit` 过程，只有在各项 Lint 检查通过后才能正常提交代码，这样就有效提高了线上代码和 Git 提交信息的质量。

上一篇： 样式方案：在 Vite 中接入现代化的 CSS 工程化方案

下一篇： 静态资源: 如何在 Vite 中处理各种静态资源?