# 规范

**在做eslint和commitlint的时候，可以使用--no-verify 跳过，这种情况下该如何强制卡点【热度:233】**

跳过 eslint 和 commitlint 的钩⼦，使⽤ --no-verify （对于 git commit 来说是 -n ），的确是⼀个容许开发者在紧急情况下超越钩⼦检查的⼿段。然⽽，这也削弱了代码质量保证的制度。以下是⼀些⽅法，可以⽤来加强这些卡点的靠谱办法：

• CI/CD流⽔线中增加检查：在你的CI/CD流程中增加 eslint 和 commitlint 的检查步骤。如果检查失败，则阻⽌代码合并或部署。

• 强制挂钩：虽然开发者可能在本地禁⽤钩⼦，但你不能控制别⼈的本地环境。相反，你可以编写服务器端的钩⼦，⽐如在Git仓库的服务器上使⽤ pre-receive 钩⼦，来拒绝不符合规范的提交。

• 定期⾃动化检查：定期运⾏⼀个⾃动化的脚本或GitHubAction，检查代码库的eslint与commitlint违规情况，并⾃动创建⼀个修复问题的issue或拉取请求。

你可以最⼤限度地减少绕过 eslint 和 commitlint 检查的情况。然⽽，值得记住的是，在极少数情况下，可能存在合法的理由需要紧急提交代码。因此，为了灵活性和效率，完全禁⽌ --noverify 可能不是⼀个最佳的选择。好的实践中应该找到安全和灵活性之间的平衡，核⼼在于建⽴⼀个质量意识，制定明智的操作规范。

**如何做commit lint 【热度:425】**

Commit lint 是一种实践，用于在代码库中规范化提交信息的格式。这种做法通常有助于团队成员理解代码库的历史记录，以及自动化生成变更日志。下面是实施 Commit lint 的一些基本步骤：

1.选择 Commit 信息规范： 首先，你需要选择一个提交信息的规范，最常见的是Conventional Commits，它具有明确的结构和规则。

2.配置 Linter 工具： commitlint 是一个流行的工具，用于检查提交信息是否符合规定的格式。安装 commitlint，通常是作为项目的开发依赖。

3.设置 commitlint 配置： 在你的项目根目录下创建一个名为 commitlint.config.js 的文件，并且导入你选择的规范：

4.安装钩子（Hook）管理工具： Husky 是一个钩子管理工具，它可以助你轻松的在 Git 挂钩中添加脚本（例如，在 commit 之前检查提交信息格式）。

5.配置 Husky 来使用 commitlint:

6.提交代码： 当你提交代码时，Husky 会触发 commit-msg 钩子调用 commitlint 检查提交信息。如果信息不符合规范，提交将被拒绝，并显示错误信息。

7.配置 CI/CD 流水线： 为了确保规范被强制执行，可以在 CI/CD 流水线中添加一步来执行 commitlint。这样，如果提交的信息不符合规范，构建将会失败。

// 安装commitlint

@commitlint/cli

@commitlint/config-conventional

// commitlint.config.js

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

// 安装 husky

npm install husky --save-dev

// 配置 husky

npx husky install

npx husky add .husky/commit-msg 'npx --no-install commitlint --edit "$1"'

// package.json添加

"husky": {

"hooks": {

"pre-commit": "yarn lint-staged",

"commit-msg": "yarn commitlint -E HUSKY\_GIT\_PARAMS"

}

}

在做eslint和commitlint的时候，可以使用--no-verify 跳过，这种情况下该如何强制卡点

跳过 eslint 和 commitlint 的钩⼦，使⽤ --no-verify （对于 git commit 来说是 -n ），的确是⼀个容许开发者在紧急情况下超越钩⼦检查的⼿段。然⽽，这也削弱了代码质量保证的制度。以下是⼀些⽅法，可以⽤来加强这些卡点的靠谱办法：

• CI/CD流⽔线中增加检查：在你的CI/CD流程中增加 eslint 和 commitlint 的检查步骤。如果检查失败，则阻⽌代码合并或部署。

• 强制挂钩：虽然开发者可能在本地禁⽤钩⼦，但你不能控制别⼈的本地环境。相反，你可以编写服务器端的钩⼦，⽐如在Git仓库的服务器上使⽤ pre-receive 钩⼦，来拒绝不符合规范的提交。

• 定期⾃动化检查：定期运⾏⼀个⾃动化的脚本或GitHubAction，检查代码库的eslint与commitlint违规情况，并⾃动创建⼀个修复问题的issue或拉取请求。

你可以最⼤限度地减少绕过 eslint 和 commitlint 检查的情况。然⽽，值得记住的是，在极少数情况下，可能存在合法的理由需要紧急提交代码。因此，为了灵活性和效率，完全禁⽌ --noverify 可能不是⼀个最佳的选择。好的实践中应该找到安全和灵活性之间的平衡，核⼼在于建⽴⼀个质量意识，制定明智的操作规范。

**v8里面的JIT是什么?【热度: 694】**

**关键词：V8 JIT**

在计算机科学中，**JIT 是“Just-In-Time”（即时编译）**的缩写，它是一种提高代码执行性能的技术。具体来说，在 V8 引擎（Google Chrome 浏览器和 Node.js 的 JavaScript 引擎）中，JIT 编译器在 JavaScript 代码运行时，将其编译成机器语言，以提高执行速度。

这里简要解释下 **JIT 编译器的工作原理**：

* 解释执行：V8 首先通过一个解释器（如 Ignition）来执行 JavaScript 代码。这个过程中，代码不会编译成机器语言，而是逐行解释执行。这样做的优点是启动快，但执行速度较慢。
* 即时编译：当代码被多次执行时，V8 会认为这部分代码是“热点代码”（Hot Spot），此时 JIT 编译器（如 TurboFan）会介入，将这部分热点代码编译成机器语言。机器语言运行在 CPU 上比解释执行要快得多。
* 优化与去优化：JIT 编译器会对热点代码进行优化，但有时候它会基于错误的假设做出优化（例如认为某个变量总是某种类型）。如果后来的执行发现这些假设不成立，编译器需要去掉优化（Deoptimize），重新编译。

JIT 编译器的一个关键优点是它能够在不牺牲启动速度的情况下，提供接近于或同等于编译语言的运行速度。这使得像 JavaScript 这样原本被认为执行效率较低的语言能够用于复杂的计算任务和高性能的应用场景。

随着 V8 和其他现代 JavaScript 引擎的不断进步，JIT 编译技术也在持续优化，以提供更快的执行速度和更高的性能。

**单元测试中，TDD、BDD、DDD分别指?【热度: 166】**

**关键词：TDD、BDD、DDD**

TDD、BDD 和 DDD 这三个缩写在软件开发中分别代表以下概念：

* TDD（Test-Driven Development） - 测试驱动开发：

TDD 是一种软件开发过程，其中开发人员首先编写一个小测试用例，然后编写足够的代码来使这个测试通过，最后重构新代码以满足所需的设计标准。这个过程就是一个循环，被成为“红-绿-重构”循环，其中测试先失败（红色），编写代码使其通过（绿色），然后优化代码（重构）。TDD 的焦点在于编写干净的代码和降低未来的缺陷。

* BDD（Behavior-Driven Development） - 行为驱动开发：

BDD 将 TDD 的基本思想和原则扩展到软件的整个开发生命周期，但其着重点在于软件的行为——即软件应如何表现，而不仅仅是它应该完成什么功能。BDD 强调的是与利益相关者的交流与协作，通过使用通俗易懂的语言来写测试，让非技术人员也能理解测试内容。BDD 鼓励团队成员之间更好地沟通，确保所有人都对软件应有的行为有共同的理解。

* DDD（Domain-Driven Design） - 领域驱动设计：

DDD 与 TDD 和 BDD 并不是同一类型的概念。DDD 是一种软件设计哲学，强调了在软件项目的设计与开发中应以业务领域（Domain）为中心。它主张将业务领域的专业知识嵌入到软件的设计中，从而使软件能更好地解决业务问题。DDD 通常涉及到丰富的领域模型以及分层的架构设计，以确保业务逻辑清晰和维护性高。

这三个概念在软件开发中都扮演着重要的角色。TDD 和 BDD 都与确保软件质量和满足用户需求有关，而 DDD 则是一种更宏观层面上对软件设计的方法论。

**husky 作用是啥，有哪些重要配置【热度: 192】**

**关键词：husky 作用、husky 配置**

Husky 是一个基于 Node 的 Git 钩子管理工具，用于在你的工作流程中强制执行 Git 钩子。Husky 允许你定义脚本，这些脚本会在不同的 Git 生命周期事件触发时自运行，比如在提交、推送或合并前。

使用 Husky 可以：

* 保证提交质量：Husky 可以在你提交代码之前运行代码校验，确保代码符合项目规范，提高代码质量。
* 维护代码风格：可以在提交时检查代码风格，确保代码风格一致性。
* 自动化流程：支持在推送前执行代码部署、测试脚本，让整个开发流程自动化。
* 预防错误：例如在允许推送到远程仓库之前检查代码中是否有遗留的更改。

Husky 的一些重要配置如下：

* npm install husky@latest --save-dev: 安装 husky。
* npx husky install: 在新建的项目管理下生成 husky 的配置文件。
* npx husky add .husky/\*.sh: 添加 Git 钩子脚本，这里的 \*.sh 是你想触发的钩子点，例如：pre-commit、commit-msg 等。

Husky 支持的钩子包括：

* apply-patch-msg: 应用一个补丁到暂存区并生成提交信息时。
* pre-applypatch: 打补丁前。
* post-applypatch: 打补丁后。
* pre-commit: 提交前，常用于检查代码、分析代码风格等。
* prepare-commit-msg: 提交准备工作完成后，修改提交信息之前运行。
* commit-msg: 检查提交信息有效性。
* post-commit: 提交后。
* pre-rebase: 回滚操作开始前。
* post-checkout: 检出操作后（如切换分支）。
* post-merge: 合并和变基操作后。

记得在 .husky 文件夹里配置这些钩子脚本，你可以根据项目需求来写自己的 hook 脚本。比如，设置一个 .husky/pre-commit 脚本（可能是一个 shell 脚本和 Node.js 脚本的组合），当你尝试提交代码时，Husky 将会运行这个脚本作为 pre-commit 钩子。

在一些场景下的 .husky/pre-commit 脚本，你可以指定运行如下：

#!/bin/sh

. "$(dirname -- "$0")/\_/husky.sh"

npm run lint # 运行 ESLint 检查代码

./node\_modules/.bin/pretty-quick # 格式化代码

./node\_modules/.bin/tsc # 检查 TypeScript

以上脚本将确保代码在提交前通过了 linter 检查，并通过 prettier 快速格式化以及 TypeScript 编译。

使用的时候，请确认你的项目已经有了 Node.js 环境，并且已经安装了 Husky 和相应的代码检查、格式化工具。

**Husky和lint-staged有什么区别?【热度: 387】**

Husky和lint-staged都是与Git钩⼦(hooks)配合使⽤的Node.js库，但它们的⽤途和⼯作⽅式有所不同：

**1. Husky：**

◦ Husky是⼀个Git钩⼦管理器，它允许你触发⾃定义脚本在git事件发⽣时运⾏，如 precommit , pre-push , post-merge 等。

◦ 它的主要⽬的是**⾃动化你的版本控制⼯作流程**，例如在提交(commit)前运⾏代码检查、格式化代码或执⾏测试，以确保代码库的质量和⼀致性。

**2. lint-staged：**

◦ lint-staged是⼀个运⾏在Husky钩⼦之上的⼯具，它专门**⽤于对暂存区(staged)⽂件的检查**。

◦ 当你运⾏ git commit 并且Husky触发 pre-commit 钩⼦时，lint-staged会检查你即将提交的代码（即 git add 后的⽂件列表），并运⾏你配置好的检查脚本，如代码格式化程序、linter或其他⼯具。

◦ 它的⽬的是确保在提交之前，只有没有检查错误的代码会被提交。

简⽽⾔之，**Husky是⼀个可以触发多种钩⼦事件的⼯具，⽽lint-staged是⼀种专⻔⽤于检查Git暂存区⽂件的⼯具。它们通常是配合使⽤的**，因为lint-staged需要通过Husky来触发钩⼦。在你初始化项⽬并配置CI/CD流程时，通常会同时⽤到它们。

**git 中回滚代码有哪些操作?【热度: 237】**

**关键词：git 回滚代码、git reset 操作、git revert 操作**

在 Git 中，可以使用以下几种操作来回滚代码：

* git revert：使用 git revert 命令可以创建一个新的提交来撤销之前的提交。它会创建一个新的提交，恢复之前的更改，从而回滚代码。这种方式比较安全，不会改变历史提交记录。
* git reset --soft：使用 git reset --soft 命令可以将当前分支的 HEAD 指向指定的提交，同时保留修改。这将撤销指定提交之后的所有提交，但是保留更改的文件在暂存区中，可以用于回滚代码并重新提交。
* git reset --mixed（默认操作）：使用 git reset --mixed 命令可以将当前分支的 HEAD 指向指定的提交，同时取消之后的提交，但保留修改。这将撤销指定提交之后的所有提交，并重置暂存区的内容，但是不改变工作区的内容，可以用于回滚代码并重新修改后重新提交。
* git reset --hard：使用 git reset --hard 命令可以将当前分支的 HEAD 指向指定的提交，并丢弃之后的修改。这将撤销指定提交之后的所有提交，并重置工作区和暂存区的内容，慎用，会永久丢失未提交的修改。

需要注意的是，**git revert 会保留之前提交的历史记录，而 git reset 则会直接修改历史提交记录**。因此，在多人协作或者公共仓库中，**建议使用 git revert 来回滚代码**，以避免对他人代码的影响。在个人项目或者正在开发中的分支中，可以使用 git reset 更灵活地回滚代码。

**git reset 作用是啥，有哪些操作?【热度:275】**

**关键词：git reset 作用、git reset 操作**

git reset 是一个用于撤销提交的命令，可以将当前分支指向某个特定的提交，同时可以选择是否修改工作区和暂存区。

常见的 git reset 操作有以下三种：

* git reset --soft ：将当前分支的 HEAD 指向指定的 commit，并保留之后的修改。这种方式不改变工作区和暂存区的内容，可以用于撤销之前的提交，重新修改后重新提交。
* git reset --mixed （默认操作）：将当前分支的 HEAD 指向指定的 commit，并取消之后的提交，但保留修改。这种方式会重置暂存区的内容，但不改变工作区的内容，可以用于撤销之前的提交，重新修改后重新提交。
* git reset --hard ：将当前分支的 HEAD 指向指定的 commit，并丢弃之后的修改。这种方式会重置工作区和暂存区的内容，慎用，会永久丢失未提交的修改。

除了上述操作之外，还可以搭配使用 git reset 的其他选项和参数。其中， 可以是一个提交的哈希值、分支名或者标签名，用于指定要回退到的提交。

**如何定制前端项目代码规范【热度: 1,155】**

当按照上述步骤定制前端代码规范时，可以按照以下详细步骤执⾏：

1. 选择代码规范⼯具：

◦ 研究可⽤的代码规范⼯具，如ESLint、Prettier等。

◦ ⽐较各⼯具的功能、灵活性和社区⽀持，并选择最适合你团队和项⽬的⼯具。

2. 定义规范：

◦ 针对项⽬的需求和团队的编码⻛格，制定代码规范的具体规则和约定。

◦ 参考⾏业内的代码规范，如AirbnbJavaScriptStyleGuide、GoogleJavaScriptStyleGuide等，以获取最佳实践和通⽤规则的参考。

◦ 考虑以下⽅⾯进⾏规范定义：

▪ 缩进和空格：确定使⽤的缩进⼤⼩、空格还是制表符等。

▪ 命名约定：定义变量、函数、类、⽂件等的命名约定。

▪ 代码⻛格：确定代码的⻛格规则，如花括号的位置、换⾏符的使⽤等。

▪ 语法约定：定义应该使⽤的语⾔特性和语法约定，如使⽤严格模式、避免使⽤特定的语⾔功能等。

3. 配置规范⼯具：

◦ 创建代码规范⼯具的配置⽂件，如 .eslintrc.js 或 .prettierrc 。

◦ 在配置⽂件中指定所选规范⼯具的规则和选项，根据定义的规范进⾏配置。

◦ 根据需要，可以启⽤或禁⽤不同的规则，并进⾏其他⾃定义配置。

4. 集成到开发环境：

◦ 集成代码规范⼯具到开发环境，以实现⾃动检测和修复代码规范问题。

◦ 针对使⽤的编辑器或集成开发环境（IDE），安装相应的插件或扩展来⽀持代码规范检查和格式化。

◦ 配置构建⼯具（如webpack）或版本控制系统（如Git）的钩⼦，以在代码提交前运⾏代码规范检查。

5. 告知团队：

◦ 与团队成员分享定制的代码规范，并解释规范的⽬的和重要性。

◦ 提供规范的⽂档或指南，以便团队成员参考和遵循。

◦ 组织⼀个会议或培训，介绍代码规范并解答团队成员的疑问。

6. 定期审查和更新：

◦ 定期审查代码规范的有效性，并根据实际需求进⾏更新和调整。

◦ 接收团队成员的反馈和建议，以改进和优化代码规范。

◦ 在项⽬演进和技术发展的过程中，适时地更新代码规范以适应变化的需求。

以上步骤是⼀个通⽤的指南，你可以根据⾃⼰的团队和项⽬的特定要求进⾏调整和执⾏。此外，团队中的讨论和协商也是⾮常重要的，确保所有成员都参与到代码规范的制定和执⾏中。

## 如何定制化开发一个 eslint 插件， 功能是实现提示检验某一个项目里面的字符串error【热度: 10】

以下是一个可以检测特定字符串并支持参数传递的 ESLint 插件开发步骤：

### 一、创建插件项目

创建一个新的目录来存放插件项目，例如eslint-plugin-custom-string-check。

在该目录下，运行npm init初始化一个 npm 项目。

### 二、插件结构

在项目目录下创建一个index.js文件作为插件的入口文件。

定义插件对象：

module.exports = {

rules: {},

};

### 三、实现规则

定义规则函数，接收一个参数options，这个参数可以包含你要检测的字符串。

module.exports = {

rules: {

"check-custom-string": (context, options) => {

return {

Program(node) {

const sourceCode = context.getSourceCode();

const text = sourceCode.getText();

const targetString = options && options.stringToCheck ? options.stringToCheck : null;

if (targetString && text.includes(targetString)) {

context.report({

node,

message: `Found custom string "${targetString}".`,

});

}

},

};

},

},

};

### 四、测试插件

在项目目录下创建一个tests目录。

在tests目录下创建一个测试文件，例如test.js。

const ruleTester = require("eslint").RuleTester;

const rule = require("../index").rules["check-custom-string"];

const ruleTester = new RuleTester();

ruleTester.run("check-custom-string", rule, {

valid: [

{

code: 'const message = "This is a test.";',

options: { stringToCheck: "errorMessage" },

},

],

invalid: [

{

code: 'const errorMessage = "This is an error.";',

options: { stringToCheck: "errorMessage" },

errors: [

{

message: 'Found custom string "errorMessage".',

},

],

},

],

});

### 五、使用插件

在你的项目中安装这个插件：

npm install /path/to/your/plugin/eslint-plugin-custom-string-check --save-dev

在项目的.eslintrc文件中配置插件：

{

"plugins": ["custom-string-check"],

"rules": {

"custom-string-check/check-custom-string": ["error", { "stringToCheck": "yourTargetString" }]

}

}

### eslint 如何集成到 webpack【热度: 488】

### 一、安装必要的包

1.确保已经安装了 Webpack 和 ESLint。如果还没有安装，可以使用以下命令进行安装：

使用 npm：

npm install webpack webpack-cli eslint --save-dev

使用 yarn：

yarn add webpack webpack-cli eslint --dev

2.安装eslint-webpack-plugin插件，这个插件可以将 ESLint 集成到 Webpack 构建过程中。

使用 npm：

npm install eslint-webpack-plugin --save-dev

使用 yarn：

yarn add eslint-webpack-plugin --dev

### 二、配置 ESLint

1.在项目根目录下，运行eslint --init命令来初始化 ESLint 配置。按照提示选择适合项目的选项，生成.eslintrc.\*配置文件。

2.根据项目需求，调整 ESLint 配置文件中的规则、解析器、插件等选项。

### 三、配置 Webpack

1.在 Webpack 配置文件（通常是webpack.config.js）中，引入eslint-webpack-plugin插件：

const ESLintPlugin = require("eslint-webpack-plugin");

2.在 Webpack 配置对象的plugins数组中添加ESLintPlugin实例：

module.exports = {

//...其他配置项

plugins: [

new ESLintPlugin({

// 可以在这里配置 ESLintPlugin 的选项，例如指定要检查的文件路径

files: ["./src/\*\*/\*.js"],

}),

],

};

### 四、运行 Webpack 构建

当运行 Webpack 构建时，eslint-webpack-plugin会在构建过程中自动运行 ESLint 检查。如果有不符合 ESLint 规则的代码，会在控制台输出错误信息。

## eslint 有哪些实用的插件， 该如何配置【热度: 488】

### 一、常见插件介绍

**eslint-plugin-import：**

作用：用于检查和规范导入语句。可以确保导入路径的正确性、防止重复导入、检查导入顺序等。

例如，它可以检测未使用的导入、循环导入等问题，并给出相应的错误提示。

**eslint-plugin-vue：**

作用：专门为 Vue.js 项目设计的插件。可以检查 Vue 单文件组件（.vue文件）中的模板、脚本和样式部分的代码规范。

例如，它可以检测模板中的错误使用的指令、脚本中的未定义变量等问题。

**eslint-plugin-prettier：**

作用：将 Prettier 的代码格式化规则集成到 ESLint 中，确保代码在风格上的一致性。

例如，它可以自动修复代码的缩进、空格、换行等格式问题，使代码更加美观易读。

**eslint-plugin-jsx-a11y：**

作用：用于检查 React 和 Vue 等框架中的 JSX 代码的可访问性问题。

例如，它可以检测图像是否缺少替代文本、表单元素是否有正确的标签等问题，以提高应用的可访问性。

### 二、配置方法

#### 1.安装插件：

使用 npm 或 yarn 安装所需的插件。例如，安装eslint-plugin-import插件：

npm install eslint-plugin-import --save-dev

或者使用 yarn：

yarn add eslint-plugin-import --dev

#### 2.创建.eslintrc文件：

在项目根目录下创建一个.eslintrc文件，用于配置 ESLint。

#### 3.配置插件：

在.eslintrc文件中，添加插件的名称到plugins数组中，并在rules对象中配置相应的规则。

例如，配置eslint-plugin-import插件：

{

"plugins": ["import"],

"rules": {

"import/no-unresolved": "error",

"import/order": [

"error",

{

"groups": ["builtin", "external", "internal"],

"pathGroups": [

{

"pattern": "vue",

"group": "external",

"position": "before"

}

],

"pathGroupsExcludedImportTypes": [],

"alphabetize": {

"order": "asc",

"caseInsensitive": true

}

}

]

}

}

上述配置中，plugins数组中添加了import插件，rules对象中配置了两个规则：import/no-unresolved用于检查导入的模块是否存在，import/order用于规范导入的顺序。

#### 4.集成到开发流程中：

根据你的开发环境和工具，将 ESLint 集成到你的开发流程中。例如，在 VS Code 中，可以安装 ESLint 扩展，并在设置中配置自动修复和实时检查。