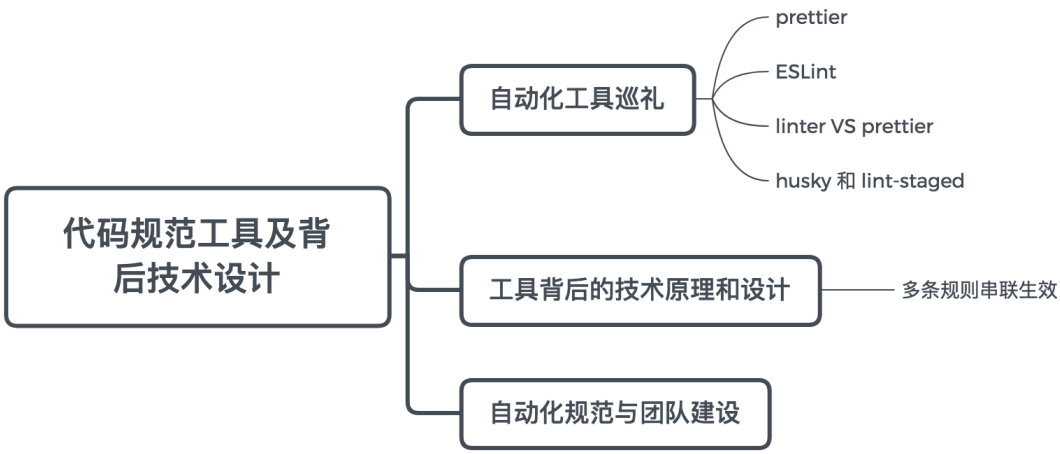
代码规范工具及背后技术设计（上）

不管是团队的扩张还是业务的发展，都会导致项目代码量出现爆炸式增长。为了防止「野蛮生长」现象，我们需要有一个良好的技术选型和成熟的架构做支撑，也需要团队中每一个开发者都能用心维护项目。在此方向上除了人工 code review 以外，相信大家对于一些规范工具并不陌生。

作为一名前端工程师，**在使用现代化工具的基础上，如何尽可能发挥其能量？在必要的情况下，如何开发适合自己团队需求的工具？** 本节将围绕这些问题展开，我们重点分析：



接下来，我们通过 2 节内容来学习这个主题。

自动化工具巡礼

现代前端开发，「武器」都已经非常自动化了。不同工具分工不同，我们的目标是合理结合各种工具，打造一条完善的自动化流水线，以高效率、低投入的方式，为我们的代码质量提供有效保障。

首先从 prettier 说起，英文单词 prettier 是 pretty 的比较级，pretty 译为「漂亮、美化」。顾名思义，prettier 这个工具能够美化我们的代码，或者说格式化、规范化代码，使其更加工整。它一般不会检查我们代码具体的写法，而是在「可读性」上做文章。目前支持包括 JavaScript、JSX、Angular、Vue、Flow、TypeScript、CSS（Less、SCSS）、JSON 等多种语言、数据交换格式、语法规范扩展。总结一下，它能够将原始代码风格移除，并替换为团队统一配置的代码风格。虽然几乎所有团队都在使用这款工具，不过我们还是简单分析一下使用它的原因吧：

构建并统一代码风格

帮助团队新成员快速融入团队

开发者可以完全聚焦业务开发，不必在代码整理上花费过多心思

方便低成本灵活接入，并快速发挥作用

清理并规范已有代码

减少潜在 Bug

丰富强大的社区支持

我们来看一个从零开始的简单 demo，首先创建一个项目（该 demo 引用自系列文章 Prettier-Eslinst-Editor-Config-Article）：

```
mkdir prettier-demo && cd prettier-demo
```

进行项目初始化：

```
yarn init -y
```

安装依赖：

在 package.json 中加入 script：

```
{
   "name": "prettier-demo",
   "version": "1.0.0",
   "scripts": {
       "prettier": "prettier --write src/index.js"
   },
}
```

prettier --write src/index.js 意思是运行 prettier，并对 src/index.js 文件进行处理：--write 标识告诉 prettier 要把格式化好的内容保存到当前文件中。

我们在 ./src 目录中新建 index.js 文件，键入一些格式缺失的代码：

```
 let person = {
          name: "Yoda",
   designation: 'Jedi Master '
   }

     function trainJedi (jediWarrion) {
     if (jediWarrion.name === 'Yoda') {
      console.log('No need! already trained')
     }
   console.log(`Training ${jediWarrion.name} complete`)
   }

            trainJedi(person)
   trainJedi({ name: 'Adeel',    designation: 'padawan'})
```

同时在根文件中创建 prettier.config.js 文件，添加 prettier 规则：

```
module.exports = {
   printWidth: 100,
```

```
    bracketSpacing: true,
    jsxBracketSameLine: false,
    tabWidth: 2,
    semi: true,
}
```

prettier 读取这些规则，并按照以上规则配置美化代码。对于这些规则，我们看其命名便能理解大概，更多内容留给大家去官网寻找。

现在运行：

```
yarn prettier
```

代码就会自动被格式化了。

当然，prettier 也可以与编辑器结合，在开发者保存后立即进行美化，也可以集成到 CI 环境中，或者 git pre-commit 的 hook 阶段。比如使用 pretty-quick：

```
yarn add prettier pretty-quick husky --dev
```

并在 package.json 中配置：

```
{
    "husky": {
        "hooks": {
            "pre-commit": "pretty-quick --staged"
        }
    }
}
```

husky 中，定义 pre-commit 阶段，对变化的文件运行 prettier，--staged 参数表示 pre-commit 模式：只对 staged 的文件进行格式化。

husky 当中介绍。
通过 demo 我们能看出，prettier 确实很灵活，且自动化程度很高，接入项目也十分方便。

ESLint

下面来看一下以 ESLint 为代表的 linter。code linting 表示基于静态分析代码原理，找出代码反模式的这过程。多数编程语言都有 linter，它们往往被集成在编译阶段，完成 coding linting 的任务。

对于 JavaScript 这种动态、松类型的语言来说，开发者更容易犯错。由于 JavaScript 不具备先天编译流程，往往在运行时暴露错误，而 linter，尤其最具代表性的 ESLint 的出现，允许开发者在执行前发现代码错误或不合理的写法。

ESLint 最重要的几点哲学思想：

- 所有规则都插件化
- 所有规则都可插拔（随时开关）
- 所有设计都透明化
- 使用 espree 进行 JavaScript 解析
- 使用 AST 分析语法

最后两点我们将在「工具背后的技术原理和设计」一小节进行分析。下面我们简单配置一个 ESLint 规则：

初始化项目：

```
yarn init -y
```

安装依赖：

并执行：

```
npx eslint --init
```

之后，我们就可以对任意文件进行 lint：

```
eslint XXX.js
```

当然，想要顺利执行 eslint，还需要安装应用规则插件。

那么如何声明并应用规则呢？在根目录中打开 .eslintrc 配置文件，我们在该文件中加入：

```
{
    "rules": {
        "semi": ["error", "always"],
        "quote": ["error", "double"]
    }
}
```

semi、quote 就是 ESLint 规则的名称，其值对应的数组第一项可以为：off/0、warn/1、error/2，分别表示关闭规则、以 warning 形式打开规则、以 error 形式打开规则。

off/0：关闭规则

warn/1：以 warning 形式打开规则

error/2：以 error 形式打开规则

同样我们还会在 .eslintrc 文件中发现：

```
"extends": "eslint:recommended"
```

Google JavaScript Style Guide

Airbnb JavaScript Style Guide

我们继续拆分 .eslintrc 文件，其实它主要由六个字段组成：

```
module.exports = {
  env: {},
  extends: {},
  plugins: {},
  parser: {},
  parserOptions: {},
  rules: {},
}
```

env：表示指定想启用的环境

extends：指定额外配置的选项，如 ['airbnb'] 表示使用 Airbnb 的 linting 规则

plugins：设置规则插件

parser：默认情况下 ESLint 使用 espree 进行解析

parserOptions：如果将默认解析器更改，需要制定 parserOptions

rules：定义拓展并通过插件添加的所有规则

注意，上文中 .eslintrc 文件我们采用了 .eslintrc.js 的 JavaScript 文件格式，此外还可以采用 .yaml、.json、yml 等格式。如果项目中含有多种配置文件格式，优先级顺序为：

```
.eslintrc.yml
.eslintrc.json
.eslintrc
package.json
```

最终，我们在 package.json 中可以添加 script：

```
"scripts": {
   "lint": "eslint --debug src/",
   "lint:write": "eslint --debug src/ --fix"
},
```

lint 这个命令将遍历所有文件，并在每个找到错误的文件中提供详细日志，但需要开发者手动打开这些文件并更正错误。

lint:write 与上类似，但这个命令可以自动纠正错误。

linter VS prettier

我们应该如何对比以 ESLint 为代表的 linter 和 prettier 呢，它们到底是什么关系？就像开篇所提到的那样，它们解决不同的问题，定位不同，但是又可以相辅相成。

所有的 linter 类似 ESLint，其规则都可以划分为两类。

格式化规则（formatting rules）

这类「格式化规则「典型的有 max-len、no-mixed-spaces-and-tabs、keyword-spacing、comma-style，它们「限制一行的最大长度」、「禁止使用空格和 tab 混合缩进」等代码格式方面的规范。事实上，即便开发者写出的代码

prettier 重叠的地方。

代码质量规则（code quality rules）

这类「代码质量规则」类似 no-unused-vars、no-extra-bind、no-implicit-globals、prefer-promise-reject-errors，它们限制「声明未使用变量」，「不必要的函数绑定」 等代码写法规范。这个时候，prettier 对这些规则无能为力。而这些规则对于代码质量和强健性至关重要，还是需要 linter 来保障的。

如同 prettier，ESLint 也可以集成到编辑器或者 git pre-commit 阶段。前文已经演示过了 prettier 搭配 husky，下面我们来介绍一下 husky 到底是什么。

husky 和 lint-staged

其实，husky 就是 git 的一个钩子，在 git 进行到某一时段时，可以交给开发者完成某些特定的操作。安装 husky：

```
yarn add --dev husky
```

然后在 package.json 文件中添加：

```
"husky": {
   "hooks": {
        "pre-commit": "YOUR_SCRIPT",
        "pre-push": "YOUR_SCRIPT"
   }
},
```

这样每次提交（commit 阶段）或者推送（push 阶段）代码时，就可以执行相关 npm 脚本。需要注意的是，在整个项目上运行 lint 会很慢，我们一般只想对更改的文件进行检查，这时候就需要使用到 lint-staged：

```
yarn add --dev lint-staged
```

```
"lint-staged": {
   "*.(js|jsx)": ["npm run lint:write", "git add"]
},
```

最终代码为：

```
"scripts": {
   "lint": "eslint --debug src/",
   "lint:write": "eslint --debug src/ --fix",
   "prettier": "prettier --write src/**/*.js"
},
"husky": {
   "hooks": {
       "pre-commit": "lint-staged"
   }
},
"lint-staged": {
   "*.(js|jsx)": ["npm run lint:write", "npm run
prettier", "git add"]
},
```

它表示在 pre-commit 阶段对于 js 或者 jsx 后缀且修改的文件执行 ESLint 和 prettier 操作，通过之后再进行 git add 添加到暂存区。

俗话说「工欲善其事，必先利其器」，本节课我们对常用工具进行了「巡礼」，请读者们亲自动手实践，了解其中奥秘。

点击查看下一节

代码规范工具及背后技术设计（下）