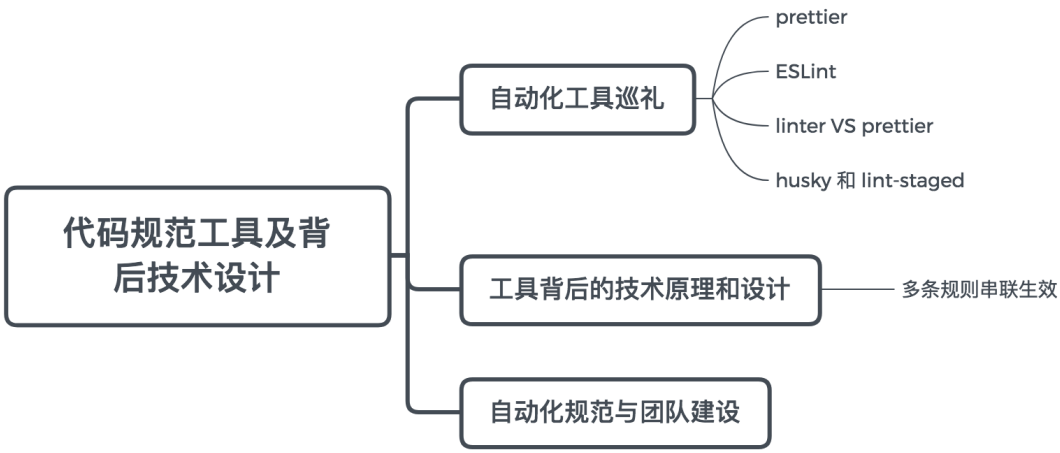
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

代码规范工具及背后技术设计（下）

上一节课，我们主要介绍了代码规范工具，了解了它们的配置、使用方式。这一节，我们将深入原理，并根据其实现和扩展能力，开发更加灵活的工具集。

在此之前，我们先回顾一下「代码规范」主题的知识点：



工具背后的技术原理和设计

这一小节，我们挑选实现更为复杂精妙的 ESLint 来分析。大家都清楚 ESLint 是基于静态语法分析（AST）进行工作的，AST 已经不是一个新鲜话题，我们在 webpack 章节就有介绍。ESLint 使用 Espree 来解析 JavaScript 语句，生成 AST。有了完整的解析树，我们就可以基于解析树对代码进行检测和修改。

ESLint 的灵魂是每一条 rule，每条规则都是独立且插件化的，我们挑一个比较简单的「禁止块级注释规则」源码来分析：

```
module.exports = {
 meta: {
   docs: {
     description: '禁止块级注释',
     category: 'Stylistic Issues',
     recommended: true
   }
 },
 create (context) {
   const sourceCode = context.getSourceCode()
   return {
     Program () {
       const comments = sourceCode.getAllComments()
       const blockComments = comments.filter(({ type }) =>
type === 'Block')
       blockComments.length && context.report({
         message: 'No block comments'
       })
     }
   }
 }
}
```

从中我们看出，一条规则就是一个 node 模块，它由 meta 和 create 组成。meta 包含了该条规则的文档描述，相对简单。而 create 接受一个 context 参数，返回一个对象：

```
{
   meta: {
       docs: {
           description: '禁止块级注释',
           category: 'Stylistic Issues',
           recommended: true
       }
   },
   create (context) {
       // ...
```

```
        return {

        }
    }
}
```

从 context 对象上我们可以取得当前执行扫描到的代码，并通过选择器获取当前需要的内容。如上代码，我们获取代码的所有 comments（sourceCode.getAllComments()），如果 blockComments 长度大于 0，则 report No block comments 信息。

我们再来看一个 no-console rule 的实现：

```
"use strict";

module.exports = {
    meta: {
        type: "suggestion",

        docs: {
            description: "disallow the use of `console`",
            category: "Possible Errors",
            recommended: false,
            url: "https://eslint.org/docs/rules/no-console"
        },

        schema: [
            {
                type: "object",
                properties: {
                    allow: {
                        type: "array",
                        items: {
                            type: "string"
                        },
                        minItems: 1,
                        uniqueItems: true
```

```
                    }
                },
                additionalProperties: false
            }
        ],

        messages: {
            unexpected: "Unexpected console statement."
        }
    },

    create(context) {
        const options = context.options[0] || {};
        const allowed = options.allow || [];

        /**
         * Checks whether the given reference is 'console'
or not.
         *
         * @param {eslint-scope.Reference} reference - The
reference to check.
         * @returns {boolean} `true` if the reference is
'console'.
         */
        function isConsole(reference) {
            const id = reference.identifier;

            return id && id.name === "console";
        }

        /**
         * Checks whether the property name of the given
MemberExpression node
         * is allowed by options or not.
         *
         * @param {ASTNode} node - The MemberExpression
node to check.
```

```
         * @returns {boolean} `true` if the property name
of the node is allowed.
         */
        function isAllowed(node) {
            const propertyName =
astUtils.getStaticPropertyName(node);

            return propertyName &&
allowed.indexOf(propertyName) !== -1;
        }

        /**
         * Checks whether the given reference is a member
access which is not
         * allowed by options or not.
         *
         * @param {eslint-scope.Reference} reference - The
reference to check.
         * @returns {boolean} `true` if the reference is a
member access which
         *      is not allowed by options.
         */
        function isMemberAccessExceptAllowed(reference) {
            const node = reference.identifier;
            const parent = node.parent;

            return (
                parent.type === "MemberExpression" &&
                parent.object === node &&
                !isAllowed(parent)
            );
        }

        /**
         * Reports the given reference as a violation.
         *
         * @param {eslint-scope.Reference} reference - The
```

```
reference to report.
         * @returns {void}
         */
        function report(reference) {
            const node = reference.identifier.parent;

            context.report({
                node,
                loc: node.loc,
                messageId: "unexpected"
            });
        }

        return {
            "Program:exit"() {
                const scope = context.getScope();
                const consoleVar =
astUtils.getVariableByName(scope, "console");
                const shadowed = consoleVar &&
consoleVar.defs.length > 0;

                /*
                 * 'scope.through' includes all references
to undefined
                 * variables. If the variable 'console' is
not defined, it uses
                 * 'scope.through'.
                 */
                const references = consoleVar
                    ? consoleVar.references
                    : scope.through.filter(isConsole);

                if (!shadowed) {
                    references
                        .filter(isMemberAccessExceptAllowed
)
                        .forEach(report);
```

```
                }
            }
        };
    }
};
```

代码中通过 astUtils.getVariableByName(scope, "console") 以及 isConsole 函数来判别 console 语句的出现，通过 allowed.indexOf(propertyName) !== -1 来过滤白名单。

实现非常简单，了解了这些，相信你也能写出 no-alert，no-debugger 的规则内容。

我们再来看一下 no-duplicate-case 规则，它监测 switch...case 中是否存在相同的 case 分支：

```
module.exports = {
    meta: {
        type: "problem",

        docs: {
            description: "disallow duplicate case labels",
            category: "Possible Errors",
            recommended: true,
            url: "https://eslint.org/docs/rules/no-
duplicate-case"
        },

        schema: [],

        messages: {
            unexpected: "Duplicate case label."
        }
    },

    create(context) {
```

```
        const sourceCode = context.getSourceCode();

        return {
            SwitchStatement(node) {
                const mapping = {};

                node.cases.forEach(switchCase => {
                    const key =
sourceCode.getText(switchCase.test);

                    if (mapping[key]) {
                        context.report({ node: switchCase,
messageId: "unexpected" });
                    } else {
                        mapping[key] = switchCase;
                    }
                });
            }
        };
    }
};
```

代码非常简单，只是初始化时使用一个空的 mapping，每次添加 case 是进行对 mapping 的扩充，如果存在相同的 case 则 report。

虽然 ESLint 背后的技术内容比较复杂，但是基于 AST 技术，它已经给开发者提供了较为成熟的 APIs。写一条自己的规则并不是很难，只需要开发者找到相关的 AST 选择器，比如上面代码中的 getAllComments()，更多的选择器可以参考：Selectors - ESLint - Pluggable JavaScript linter。熟练掌握选择器，将是我们开发插件扩展的关键。

当然，更复杂的场景远不止这么简单，比如，多条规则是如何串联起来生效的?

多条规则串联生效

事实上，**规则可以从多个源来定义，比如代码的注释当中，或者配置文件当中。**

ESLint 首先收集到所有规则配置源，将所有规则归并之后，进行多重遍历：遍历由源码生成的 AST，将语法节点传入队列当中；之后遍历所有应用规则，采用事件发布订阅模式（类似 webpack tapable），为所有规则的选择器添加监听事件；在触发事件时执行，如果发现有问题，会将 report message 记录下来。最终记录下来的问题信息将会被输出。

具体 ESLint 的源码如下：

```
function runRules(sourceCode, configuredRules, ruleMapper,
parserOptions, parserName, settings, filename) {
   const emitter = createEmitter();
   const nodeQueue = [];
   let currentNode = sourceCode.ast;

   Traverser.traverse(sourceCode.ast, {
       enter(node, parent) {
           node.parent = parent;
           nodeQueue.push({ isEntering: true, node });
       },
       leave(node) {
           nodeQueue.push({ isEntering: false, node });
       },
       visitorKeys: sourceCode.visitorKeys
   });


   const lintingProblems = [];

   Object.keys(configuredRules).forEach(ruleId => {
       const severity =
ConfigOps.getRuleSeverity(configuredRules[ruleId]);

       if (severity === 0) {
           return;
       }

       const rule = ruleMapper(ruleId);
```

```
        const messageIds = rule.meta && rule.meta.messages;
        let reportTranslator = null;
        const ruleContext = Object.freeze(
            Object.assign(
                Object.create(sharedTraversalContext),
                {
                    id: ruleId,
                    options:
getRuleOptions(configuredRules[ruleId]),
                    report(...args) {

                        if (reportTranslator === null)
{...}
                        const problem =
reportTranslator(...args);
                        if (problem.fix && rule.meta &&
!rule.meta.fixable) {
                            throw new Error("Fixable rules
should export a `meta.fixable` property.");
                        }
                        lintingProblems.push(problem);
                    }
                }
            )
        );

        const ruleListeners = createRuleListeners(rule,
ruleContext);

        // add all the selectors from the rule as listeners
        Object.keys(ruleListeners).forEach(selector => {
            emitter.on();
        });
    });

    const eventGenerator = new CodePathAnalyzer(new
NodeEventGenerator(emitter));
```

```
    nodeQueue.forEach(traversalInfo => {
        currentNode = traversalInfo.node;
        if (traversalInfo.isEntering) {
            eventGenerator.enterNode(currentNode);
        } else {
            eventGenerator.leaveNode(currentNode);
        }
    });

    return lintingProblems;
}
```

请再思考，我们的程序中免不了有各种条件语句、循环语句，因此 **代码的执行是非顺序的**。相关规则比如：「检测定义但未使用变量」，「switch-case 中避免执行多条 case 语句」，这些规则的实现，就涉及 ESLint 更高级的 code path analysis 概念等。ESLint 将 code path 抽象为 5 个事件。

onCodePathStart

onCodePathEnd

onCodePathSegmentStart

onCodePathSegmentEnd

onCodePathSegmentLoop

利用这 5 个事件，我们可以更加精确地控制检测范围和粒度。更多的 ESLint rule 实现，可以翻看源码进行学习，总之根据这 5 种事件，我们可以监测非顺序性代码，其核心原理还是事件机制。

我们通过 no-unreachable 规则来进行了解，该规则可以通过监测 return，throws，break，continue 的使用，识别出不会被执行的代码，并 report：

```
/**
* Checks whether or not a given variable declarator has
the initializer.
* @param {ASTNode} node - A VariableDeclarator node to
check.
* @returns {boolean} `true` if the node has the
initializer.
*/
function isInitialized(node) {
   return Boolean(node.init);
}

/**
* Checks whether or not a given code path segment is
unreachable.
* @param {CodePathSegment} segment - A CodePathSegment to
check.
* @returns {boolean} `true` if the segment is unreachable.
*/
function isUnreachable(segment) {
   return !segment.reachable;
}

/**
* The class to distinguish consecutive unreachable
statements.
*/
class ConsecutiveRange {
   constructor(sourceCode) {
       this.sourceCode = sourceCode;
       this.startNode = null;
       this.endNode = null;
   }

   /**
    * The location object of this range.
    * @type {Object}
```

```
     */
    get location() {
        return {
            start: this.startNode.loc.start,
            end: this.endNode.loc.end
        };
    }

    /**
     * `true` if this range is empty.
     * @type {boolean}
     */
    get isEmpty() {
        return !(this.startNode && this.endNode);
    }

    /**
     * Checks whether the given node is inside of this
range.
     * @param {ASTNode|Token} node - The node to check.
     * @returns {boolean} `true` if the node is inside of
this range.
     */
    contains(node) {
        return (
            node.range[0] >= this.startNode.range[0] &&
            node.range[1] <= this.endNode.range[1]
        );
    }

    /**
     * Checks whether the given node is consecutive to this
range.
     * @param {ASTNode} node - The node to check.
     * @returns {boolean} `true` if the node is consecutive
to this range.
     */
```

```
    isConsecutive(node) {
        return
this.contains(this.sourceCode.getTokenBefore(node));
    }

    /**
     * Merges the given node to this range.
     * @param {ASTNode} node - The node to merge.
     * @returns {void}
     */
    merge(node) {
        this.endNode = node;
    }

    /**
     * Resets this range by the given node or null.
     * @param {ASTNode|null} node - The node to reset, or
null.
     * @returns {void}
     */
    reset(node) {
        this.startNode = this.endNode = node;
    }
}

//------------------------------------------------------------
----------------------
// Rule Definition
//------------------------------------------------------------
----------------------

module.exports = {
    meta: {
        type: "problem",

        docs: {
            description: "disallow unreachable code after
```

```
`return`, `throw`, `continue`, and `break` statements",
            category: "Possible Errors",
            recommended: true,
            url: "https://eslint.org/docs/rules/no-
unreachable"
        },

        schema: []
    },

    create(context) {
        let currentCodePath = null;

        const range = new
ConsecutiveRange(context.getSourceCode());

        /**
         * Reports a given node if it's unreachable.
         * @param {ASTNode} node - A statement node to
report.
         * @returns {void}
         */
        function reportIfUnreachable(node) {
            let nextNode = null;

            if (node &&
currentCodePath.currentSegments.every(isUnreachable)) {

                // Store this statement to distinguish
consecutive statements.
                if (range.isEmpty) {
                    range.reset(node);
                    return;
                }

                // Skip if this statement is inside of the
current range.
```

```
                if (range.contains(node)) {
                    return;
                }

                // Merge if this statement is consecutive
to the current range.
                if (range.isConsecutive(node)) {
                    range.merge(node);
                    return;
                }

                nextNode = node;
            }

            /*
             * Report the current range since this
statement is reachable or is
             * not consecutive to the current range.
             */
            if (!range.isEmpty) {
                context.report({
                    message: "Unreachable code.",
                    loc: range.location,
                    node: range.startNode
                });
            }

            // Update the current range.
            range.reset(nextNode);
        }

        return {

            // Manages the current code path.
            onCodePathStart(codePath) {
                currentCodePath = codePath;
            },
```

```
            onCodePathEnd() {
                currentCodePath = currentCodePath.upper;
            },

            // Registers for all statement nodes (excludes
    FunctionDeclaration).
            BlockStatement: reportIfUnreachable,
            BreakStatement: reportIfUnreachable,
            ClassDeclaration: reportIfUnreachable,
            ContinueStatement: reportIfUnreachable,
            DebuggerStatement: reportIfUnreachable,
            DoWhileStatement: reportIfUnreachable,
            ExpressionStatement: reportIfUnreachable,
            ForInStatement: reportIfUnreachable,
            ForOfStatement: reportIfUnreachable,
            ForStatement: reportIfUnreachable,
            IfStatement: reportIfUnreachable,
            ImportDeclaration: reportIfUnreachable,
            LabeledStatement: reportIfUnreachable,
            ReturnStatement: reportIfUnreachable,
            SwitchStatement: reportIfUnreachable,
            ThrowStatement: reportIfUnreachable,
            TryStatement: reportIfUnreachable,

            VariableDeclaration(node) {
                if (node.kind !== "var" ||
    node.declarations.some(isInitialized)) {
                    reportIfUnreachable(node);
                }
            },

            WhileStatement: reportIfUnreachable,
            WithStatement: reportIfUnreachable,
            ExportNamedDeclaration: reportIfUnreachable,
            ExportDefaultDeclaration: reportIfUnreachable,
            ExportAllDeclaration: reportIfUnreachable,
```

```
            "Program:exit"() {
                reportIfUnreachable();
            }
        };
    }
};
```

实现中，通过 isUnreachable 函数来判别一个 code path 是否无法触及，我提供一些返例帮助大家理解：

```
function foo() {
   return true;
   console.log("done");
}

function bar() {
   throw new Error("Oops!");
   console.log("done");
}

while(value) {
   break;
   console.log("done");
}

throw new Error("Oops!");
console.log("done");

function baz() {
   if (Math.random() < 0.5) {
       return;
   } else {
       throw new Error();
   }
   console.log("done");
}
```

因为 unreachable 的代码需要放在一个区块当中去理解，单条语句无法去进行判别，因此使用 ConsecutiveRange 类来保留连续代码信息。

最后，这种优秀的插件扩展机制对于设计一个库，尤其是设计一个规范工具来说，是非常值得借鉴的模式。事实上，prettier 也会在新的版本中引入插件机制，目前已经在 beta 版，感兴趣的读者可以尝鲜。

自动化规范与团队建设

自动化规范还有其他一些细节，比如使用 EditorConfig 来保证编辑器的设置统一，确定在制表符空格或换行方面的一致性，又如使用 commitlint 并配合 husky，来保证 commit message 的规范：

```
# 安装 commitlint cli 和 conventional config
npm install --save-dev @commitlint/{config-
conventional,cli}

# 配置 commitlint
echo "module.exports = {extends: ['@commitlint/config-
conventional']}" > commitlint.config.js
```

并在 commit-msg 的 git hook 阶段进行检查，在 package.json 中添加：

```
{
    "husky": {
        "hooks": {
            "commit-msg": "commitlint -E HUSKY_GIT_PARAMS"
        }
    }
}
```

我们也可以根据团队需求做更多定制化的尝试，比如自动规范化或生产 commit message，有了规范的 commit message 之后，就可以提取关键内容，规范化生产 changelog 等。

其他方向上，还可以从团队文档的生产来考虑。举个例子，如果使用 React 开发项目，那么 React 组件文档如何规范化生成？如何提高组件使用的效率，减少学习成本？我在掘金 AMA 上做客时，有人便提出了这样的问题。

我们组内面临着最古老的 React 管理平台重构任务，这次我们想生成关于管理平台的阅读文档（包括常用的样式命名、工具方法、全局组件、复杂 API 交互流程等）。

所以我想提出的问题是：面向 React 代码的可维护性和可持续发展（不要单个功能每个团队成员都实现一遍，当新成员加入的时候知道有哪些功能能从现在代码中复用， 也知道有哪些功能还没有，他可以添加实现进去），业内有哪些工具或 npm 库或开发模式是可以确切能够帮助解决痛点或者改善现状的呢?

确实，随着项目复杂度的提升，各种组件也「爆炸式」增长。如何让这些组件方便易用，能快速上手，同时不成为负担，又避免重复造轮子现象，良好的组件管理在团队中非常重要。

关于「React 组件管理文档」，简单梳理一下：总得来说，社区在这方面的探索很多，相关方案也各有特色。

> 最知名的一定是 storybook，它会生成一个静态页面，专门用来展示组件的实际效果以及用法；缺点是业务侵入性较强，且 story 编写成本较高。
>
> 我个人很喜欢的是 react-docgen，比较极客风格，它能够分析并提取 React 组件信息。原理是使用了 recast 和 @babel/parser AST 分析，最终产出一个 JSON 文档。 https://github.com/reactjs/react-docgen 是它的网页链接，缺点是它较为轻量，缺乏有效的可视化能力。
>
> 那么在 react-docgen 之上，我们可以考虑 React Styleguidist，这款 React 组件文档生成器，支持丰富的 demo，可能会更符合需求。
>
> 一些小而美的解决方案：比如 react-doc 、react-doc-generator 、cherrypdoc，都可以考虑尝试。

「自己动手、丰衣足食」，其实开发一个类似的工具并不会太复杂。如果有时间和精力，你可以根据自己的需求，实现一个完全匹配自己团队的 React 组件管理文档，或者其他框架相关、业务相关的文档，这非常有意义。

总结

在规范化的道路上，只有你想不到，没有你做不到。

简单的规范化工具用起来非常清爽，但是背后的实现却蕴含了很深的设计与技术细节，值得我们深入学习。

作为前端工程师，我们应该从平时开发的痛点和效率瓶颈入手，敢于尝试，不断探索。保证团队开发的自动化程度，就能减少不必要的麻烦。

除了「偏硬」的强制规范手段，一些「软方向」，比如团队氛围、code review/analyse 等，也直接决定着团队的代码质量。进阶的工程师不仅需要在技术上成长，在团队建设上更需要主动交流。

课程代码仓库： https://github.com/HOUCe/lucas-gitchat-courses

点击查看下一节

性能监控和错误收集与上报（上）