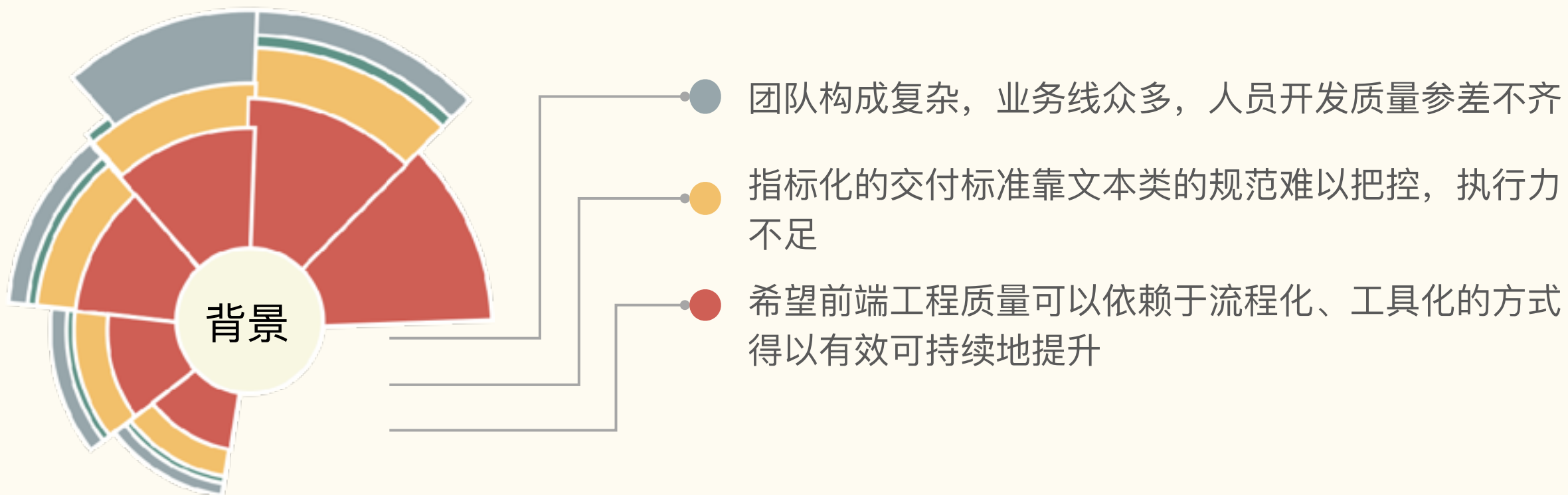


前端工程质量保障体系建设

演讲人：张蕾

网易严选主站C端前端团队

Why





CONTENTS

01

当我们在谈论前端工程质量时，我们讨论的是什么？

前端工程质量包含哪几种维度

02

规范的制定

针对前端工程质量的不同维度分别输出验收标准

03

具体约束策略

如何基于DevOps进行前端工程质量约束

04

总结与规划

回顾关键内容，展望未来发展愿景



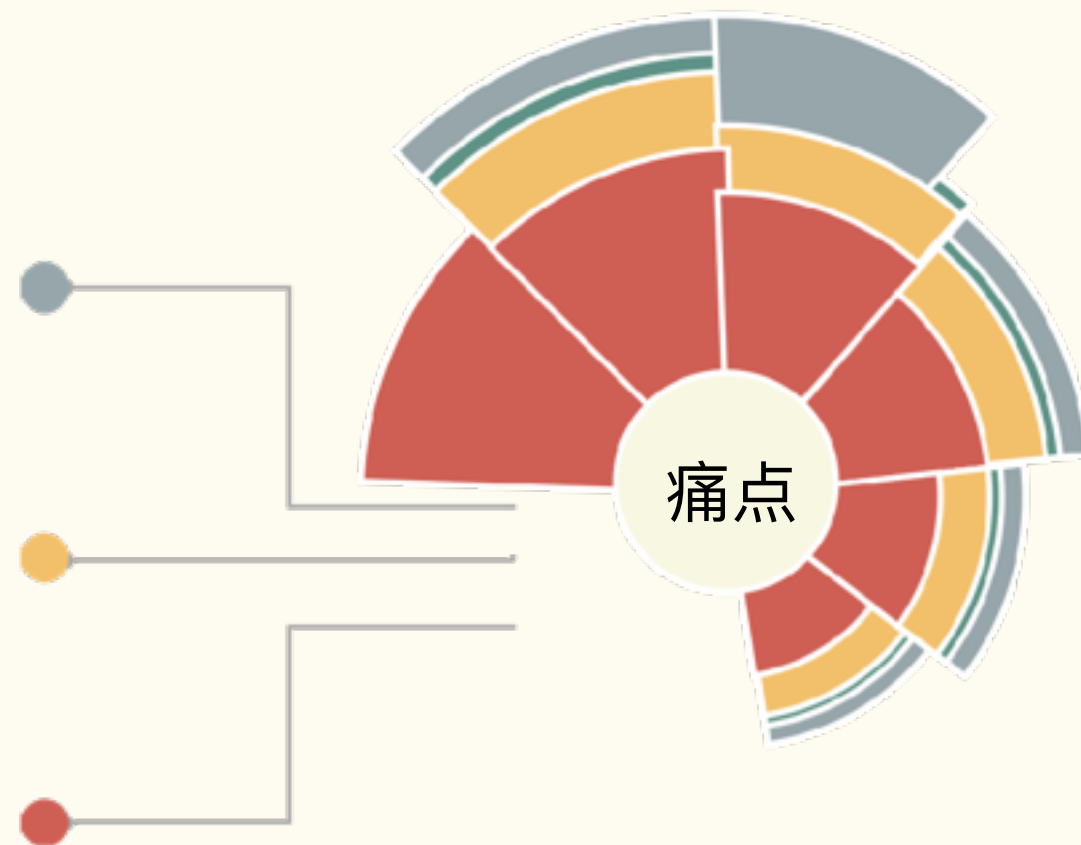
当我们在谈论前端工程质量时，我们讨论的是什么？

你是否遇到这样的场景？

【场景1】 A同学刚刚接手新的业务，但该业务没有任何文档沉淀，又缺少注释和详尽的readme，导致A开发过程中只能看代码梳理业务逻辑，频繁踩坑

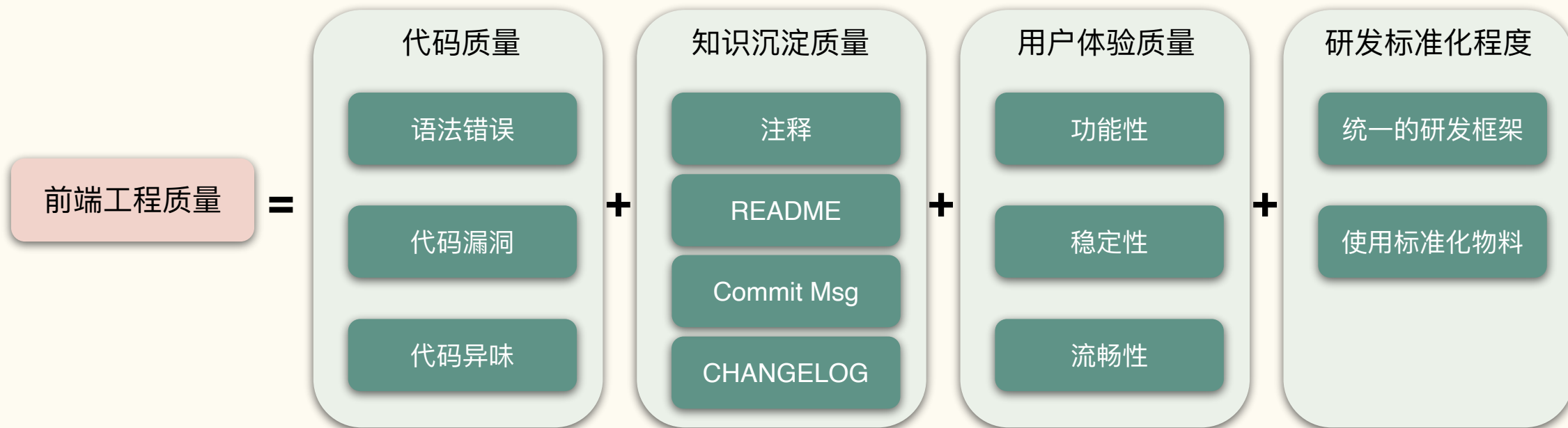
【场景2】 B团队中各个同学的commit msg风格不一致且规范性较差，changelog缺失。等其他团队接手该业务时难以获取工程项目的变更关键信息

【场景3】 C团队缺失项目交付前的质量约束环节，eslint仅停留在高亮提醒。进入高速发展阶段时，团队人员快速扩张，且素质参差不齐，从而形成代码质量隐患。当线上出现问题时才反推代码质量的提升



1. 当我们在谈论前端工程质量时，我们讨论的是什么？

➤ 前端工程质量包含哪几种维度？



1. 当我们在谈论前端工程质量时，我们讨论的是什么？

➤ 代码质量

语法错误

不合规的语法等书写错误，直接导致bug的产生，

如：

- 变量未声明
- `await`未被用在`async`修饰的函数内

代码具有隐患，特定情况下会导致bug的产生，如：

- 使用 `==` 进行判等，而不是 `===`

代码漏洞

代码异味

低质量代码/无用代码等，会加速项目的腐化，如：

- 声明的变量/方法未被使用
- `import`引入的模块未被使用

1. 当我们在谈论前端工程质量时，我们讨论的是什么？

➤ 知识沉淀质量

知识沉淀对于一个团队的良性发展来说尤为重要，它不仅关乎日常开发的工作效率、沟通成本，还决定着团队整体的研发质量。

知识沉淀 = 写文档？

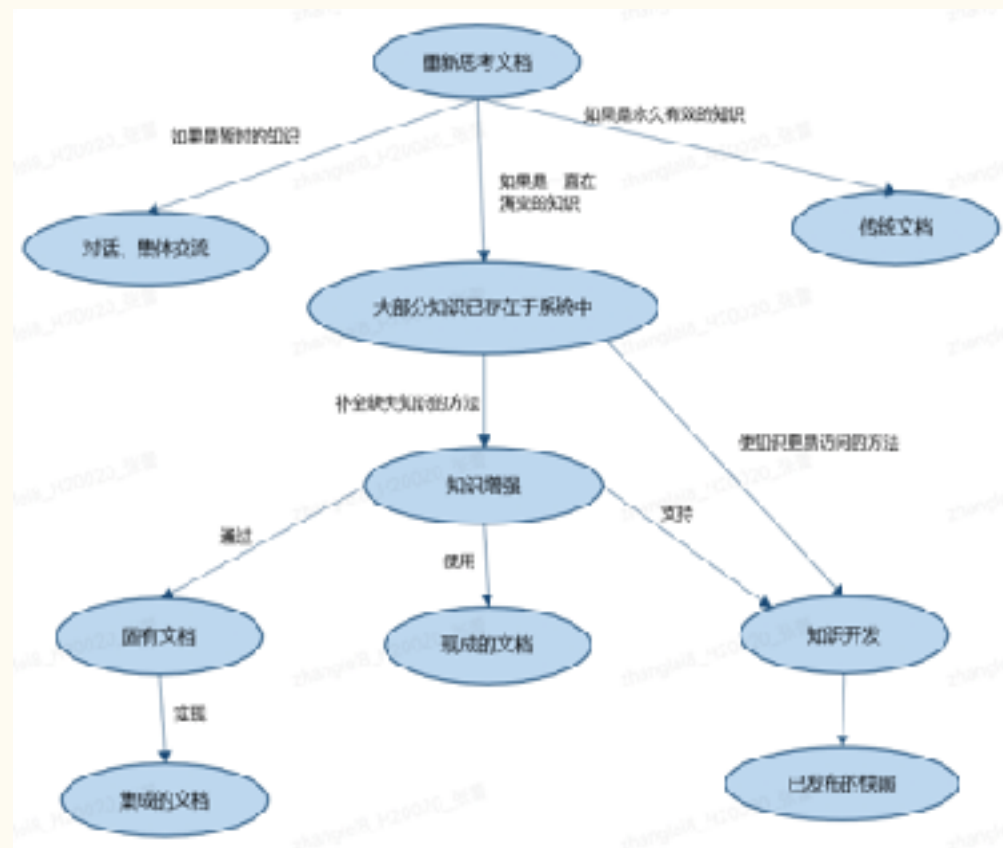
需要明确的是，知识沉淀不限于文档的沉淀，需求交付过程中的每个环节涉及到的有效信息沉淀都属于知识管理的范畴。

1. 当我们在谈论前端工程质量时，我们讨论的是什么？

➤ 知识沉淀质量

传统文档存在的问题

- 基本依托于wiki、在线文档等这种第三方系统来统一管理。维护成本大，易腐化，进而影响可阅读性
- 文档与代码太过脱离，而脱离于项目代码的知识并不能真正满足开发者的需求



推动知识本地化建设，打造与代码共同演进的“活文档”

1. 当我们在谈论前端工程质量时，我们讨论的是什么？

➤ 知识沉淀质量

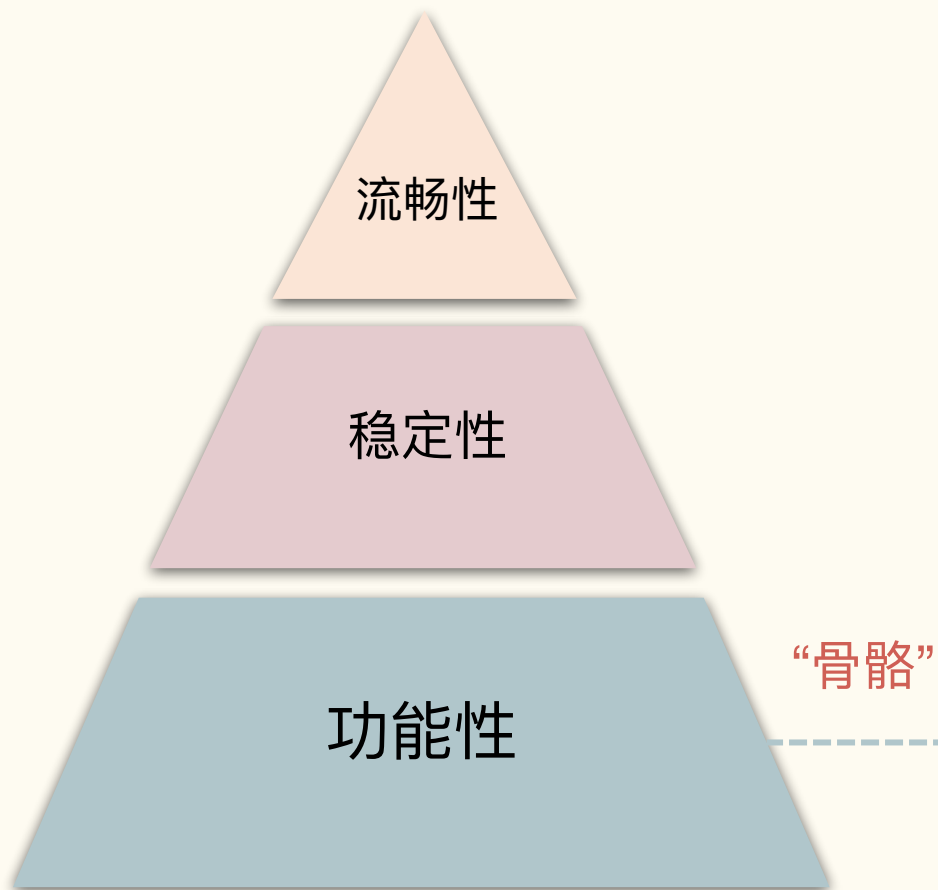
1. 如果知识已经记录在项目中，就无须再被编写成文档

2. 存储知识的最佳位置是被记录的事物本身

知识类型	沉淀形式
代码规范	Lint规则
项目描述信息、开发教程、DevOps等	README，本地文档系统
项目迭代信息	Commit Msg，CHANGELOG
业务逻辑描述信息	注释

1. 当我们在谈论前端工程质量时，我们讨论的是什么？

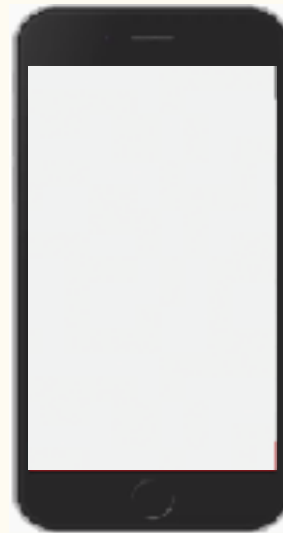
➤ 用户体验质量



页面正常打开，不影响功能的正常使用。

常见的异常表现主要有两种：

- 页面展示异常——白屏

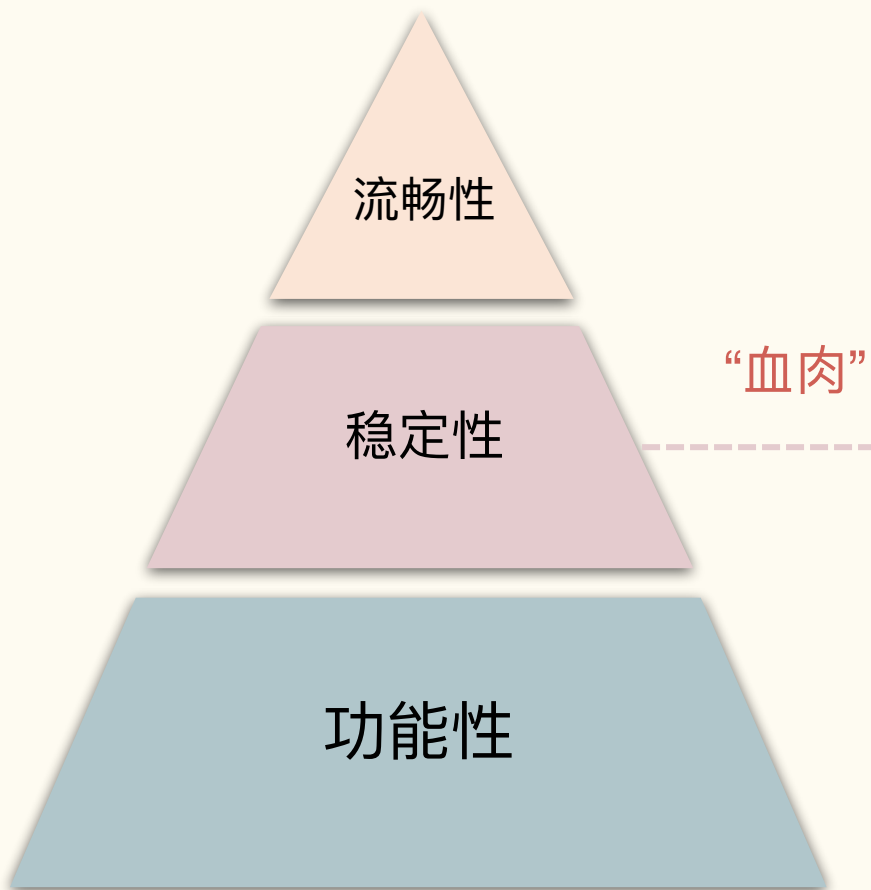


- 页面交互异常——js报错

```
Uncaught TypeError: Cannot read properties of undefined (reading 'cash')  
at CouponAlert.render (coupon-alert.js:10:40) | 10/11/2024  
at render (react-dom.js:21:17) | 10/11/2024
```

1. 当我们在谈论前端工程质量时，我们讨论的是什么？

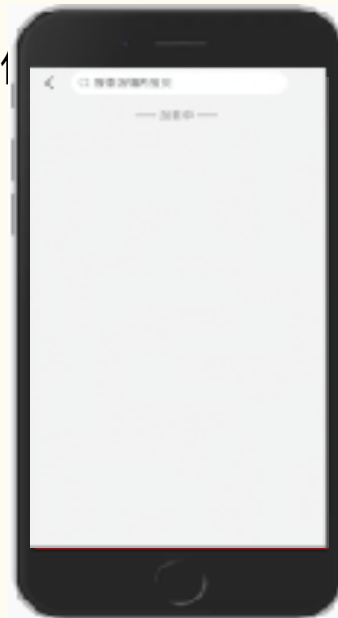
➤ 用户体验质量



核心流程接口异常情况下页面的容灾能力。

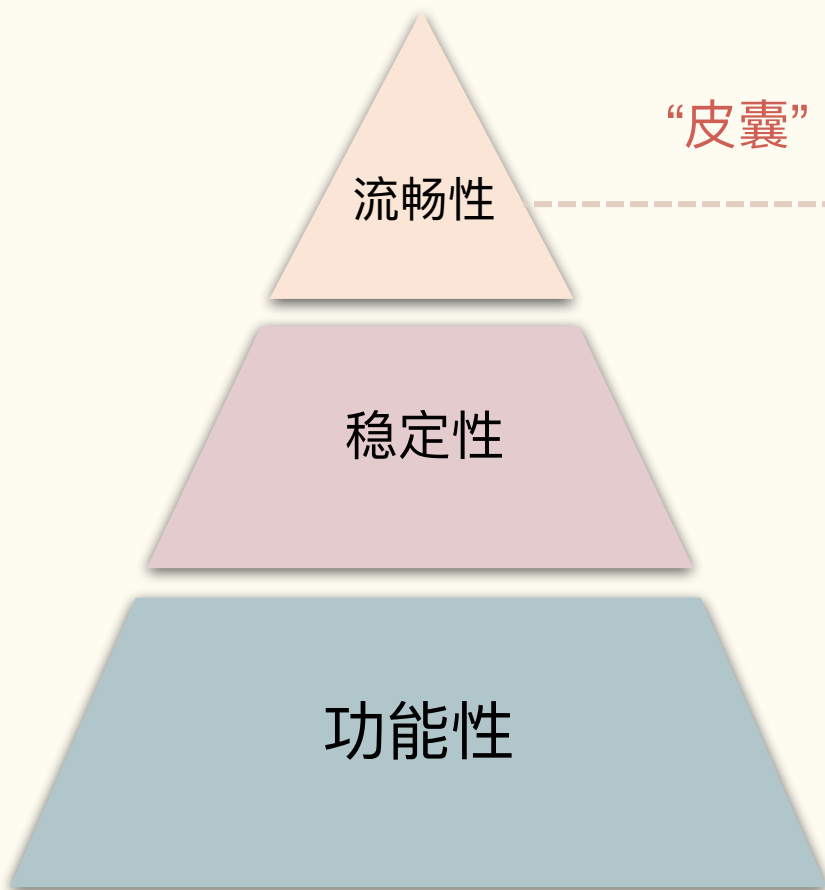
容灾能力的缺失主要有两种：

- 缺少兜底策略
- 缺少补位



1. 当我们在谈论前端工程质量时，我们讨论的是什么？

➤ 用户体验质量

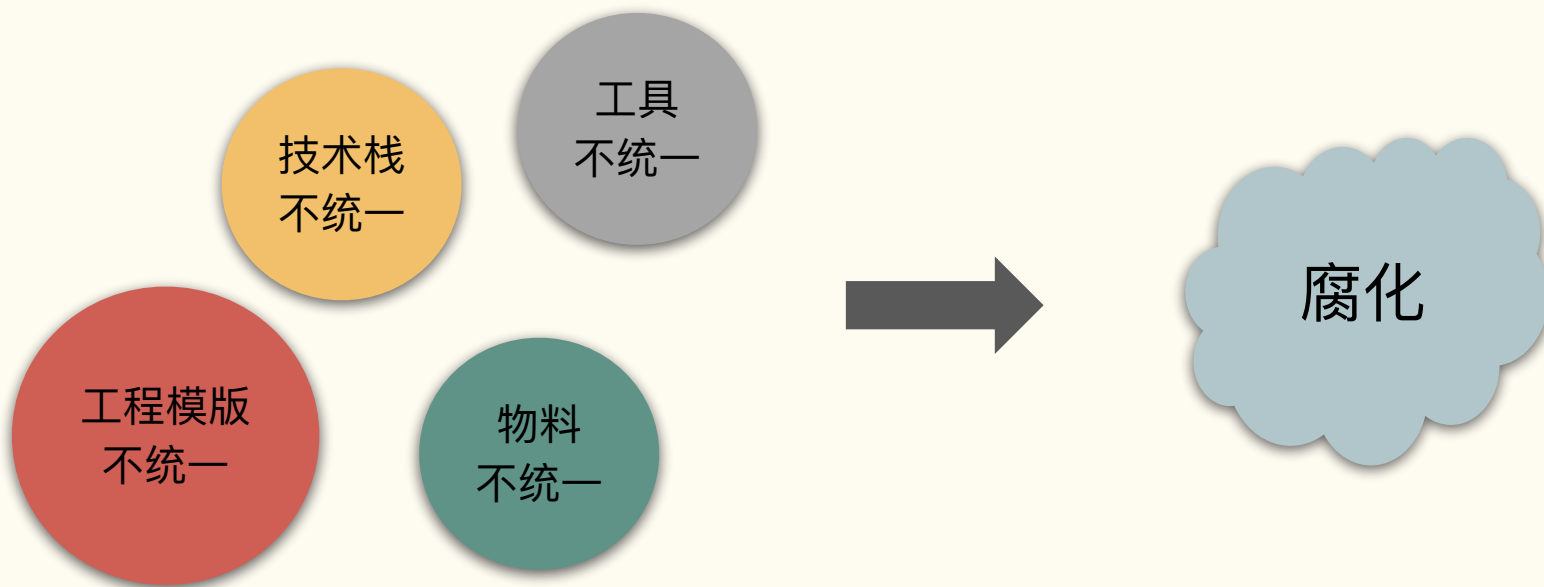


页面加载的快慢、交互时的卡顿情况
等性能问题直接影响用户的使用感：



1. 当我们在谈论前端工程质量时，我们讨论的是什么？

➤ 研发标准化程度



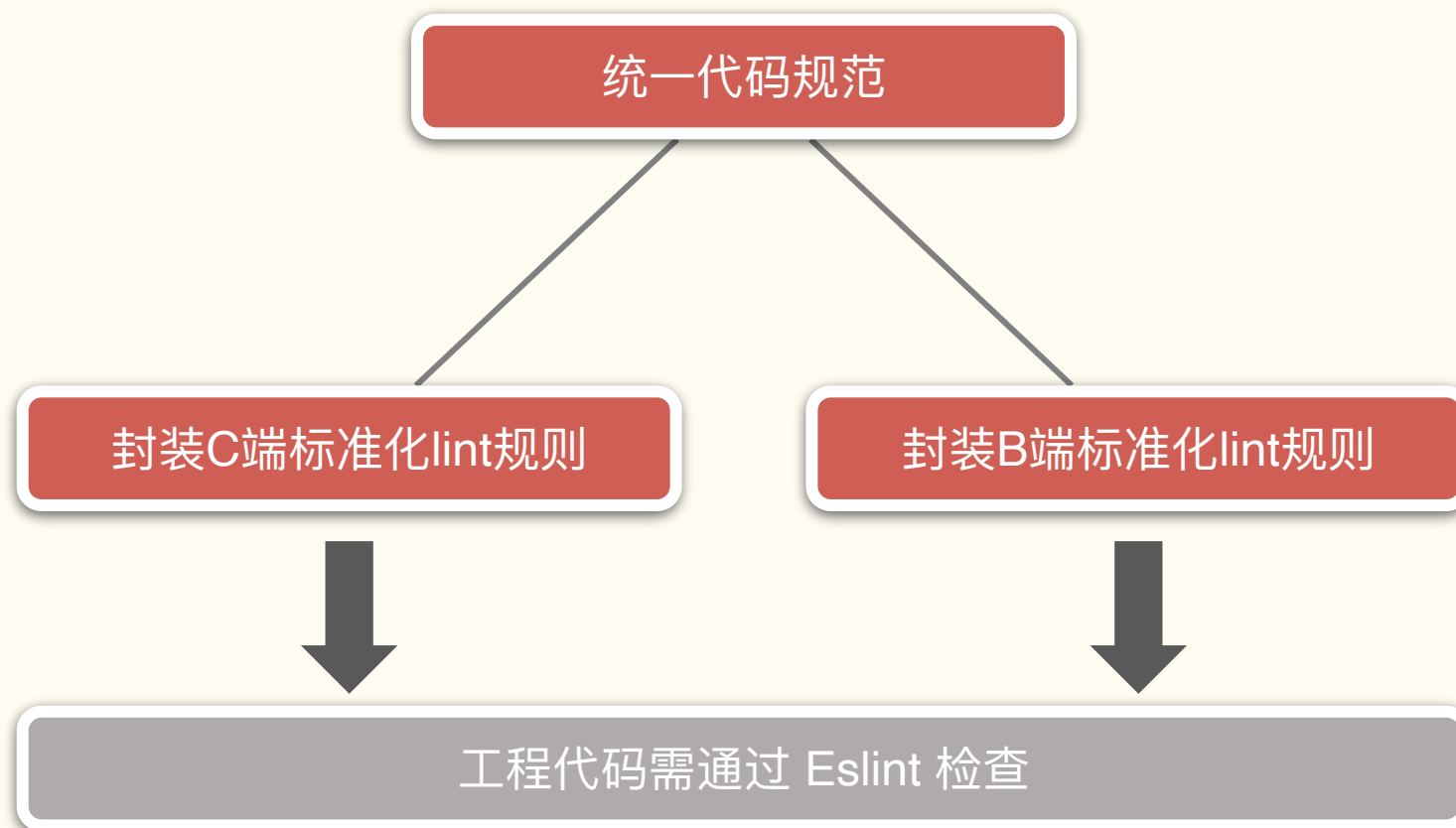


制定相关规范及验收标准

2. 制定相关规范及验收标准

➤ 代码质量

统一
代码
规范



2. 制定相关规范及验收标准

➤ 代码质量

统一代码规范的好处



- 提高代码整体的可读性、可维护性、可复用性、可移植性和可靠性
- 保证代码的一致性
- 提升团队整体研发、沟通效率

2. 制定相关规范及验收标准

➤ 知识沉淀质量

(1) 需要给出注释的场景

注释

场景	文件头部	组件	方法	组件/方法内部	接口&枚举
描述	.js,.jsx,.ts,.tsx	类组件,函数组件	— —	— —	— —
必要程度 (强制  , 建议 )					

2. 制定相关规范及验收标准

➤ 知识沉淀质量

(2) 标准化注释格式

注释

```
// 声明一个变量  
const count = 10;
```

无效注释，大可不必

```
/**  
 * @description: 点击切换领取消息订阅  
 * @param {number} status 订阅状态  
 * @returns {void}  
 */  
handleTogglePush(status) {  
  if (!status) {  
    Notify.notify('取消订阅容易错过福利哦');  
    return;  
  }  
  // ....  
}
```

参考 jsdoc<Javascript官方的注释文档标准>制定注释规范

2. 制定相关规范及验收标准

➤ 知识沉淀质量

工程项目中必须包含 README.md，且其中需要包含如下七部分内容：

- (1) 概述——必填
- (2) 功能简述
- (3) 运行指南——必填
- (4) 开发指南——必填
- (5) 业务介绍
- (6) CHANGELOG
- (7) FAQ

README

2. 制定相关规范及验收标准

➤ 知识沉淀质量

- 要求每个commit message必须包含三个部分：header、body（选填）、footer（选填）。而header由 type、scope（选填）、subject 组成。

```
1 <header><type>(<scope>): <subject>
2 <BLANK LINE>
3 <body>
4 <BLANK LINE>
5 <footer>
```

- 严格限制 type 的书写规范：feat、fix、change、doc、chore、merge、revert

2. 制定相关规范及验收标准

➤ 用户体验质量

- 度量指标：错误数
- 验收标准：各个类型错误数 = 0

2. 制定相关规范及验收标准

➤ 用户体验质量

- 度量指标：白屏率——首屏接口异常的情况下，页面出现白屏的比例

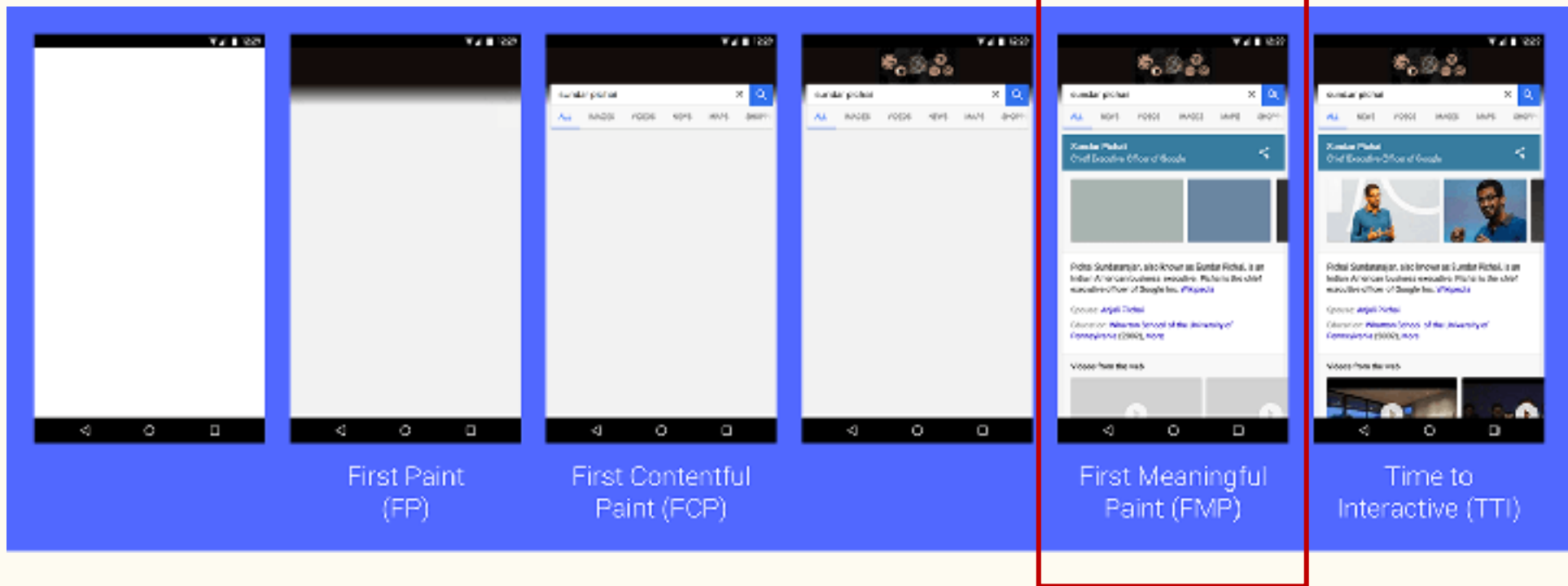
比例 = 请求异常导致白屏的接口数 / 接口总数量

- 验收标准：0%，原则上需要杜绝白屏的出现

2. 制定相关规范及验收标准

➤ 用户体验质量

- 度量指标：FMP (First Meaningful Paint)



- 验收标准：多页面FMP均值 $\leq 1200\text{ms}$

2. 制定相关规范及验收标准

➤ 研发标准化程度

1. 建立技术栈规范

进行React技术栈统一建设，并输出相关前端开发规范

2. 统一研发框架

产出能支撑各业务线开发的的统一研发框架，涵盖页面/组件维度

3. 统一开发物料

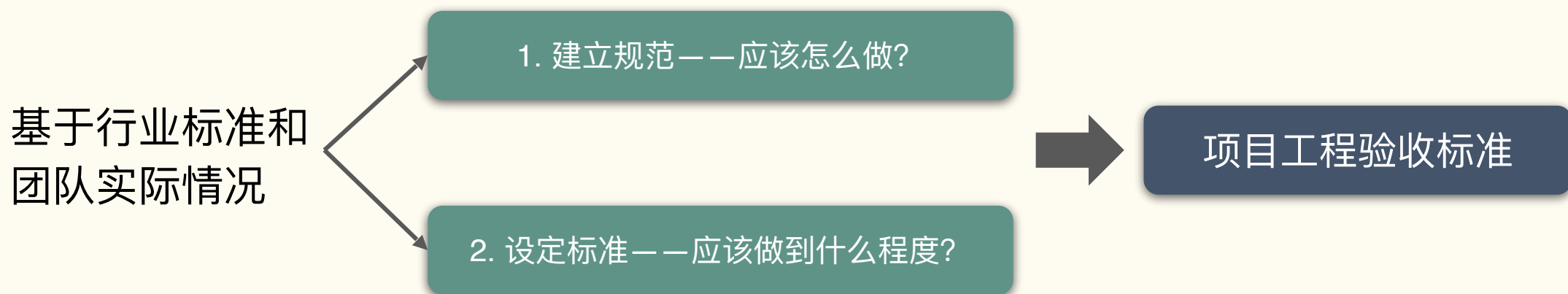
建设统一的组件库和 jssdk 基础库，新项目覆盖率 $\geq 70\%$

4. 统一开发工具

研发配套的工程化 CLI 工具及可视化智能编码辅助工具

2. 制定相关规范及验收标准

➤ 小结

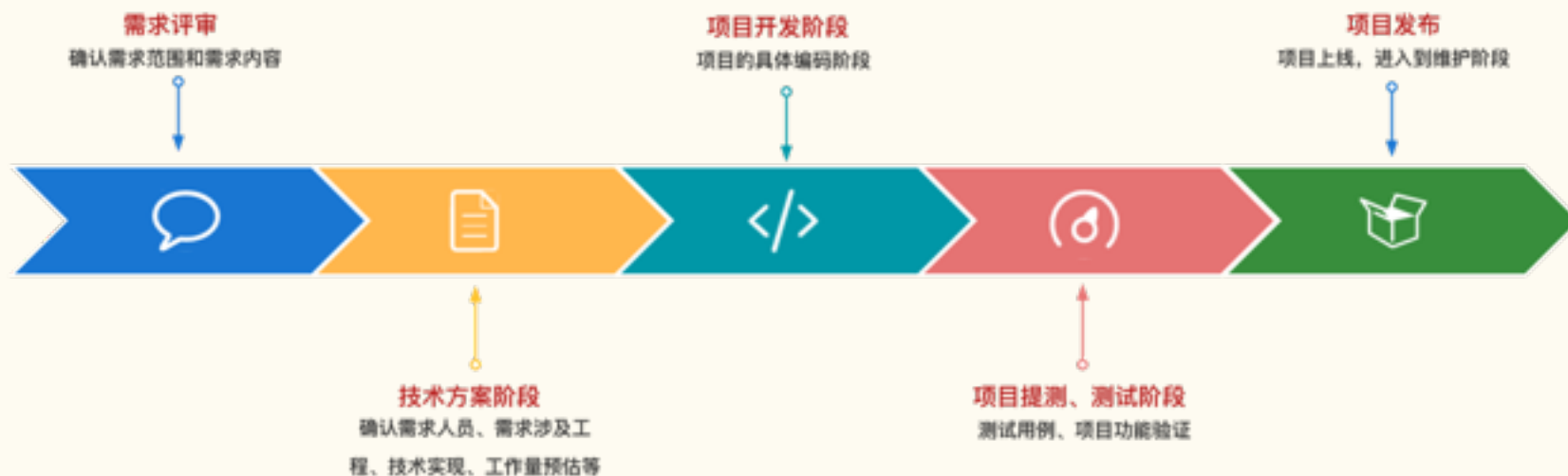




前端工程质量约束策略

3. 前端工程质量约束策略

➤ 整体思路



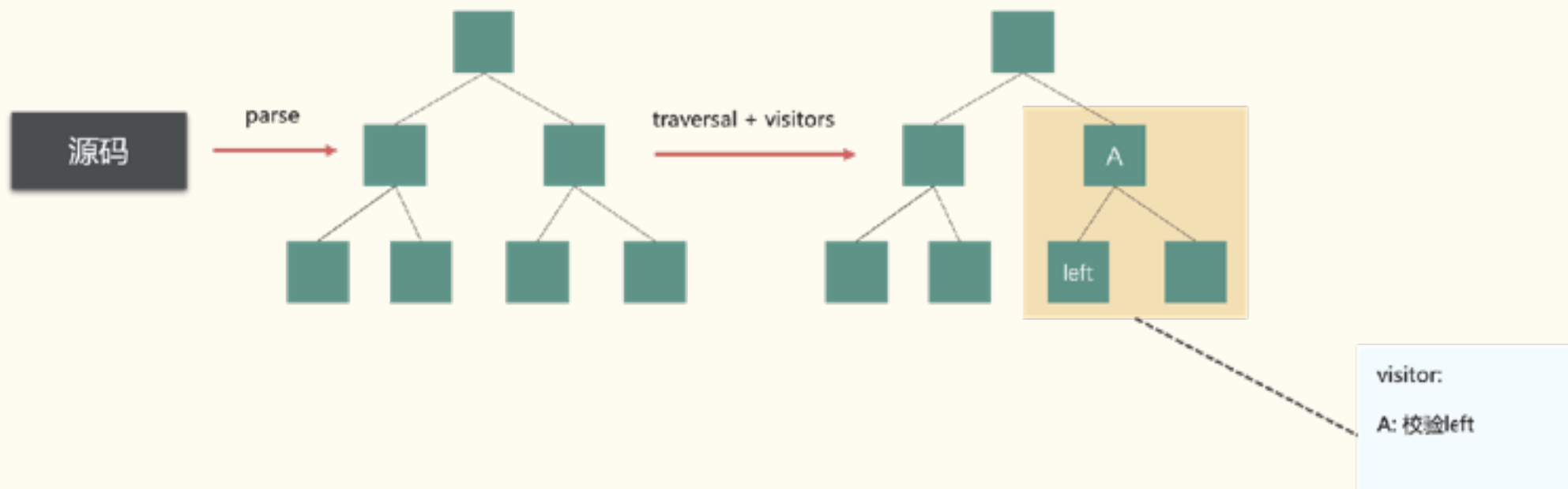
审视整个研发生命周期，针对不同研发阶段通过 **工具化 + 流程化** 的方式保障前端工程质量：

- 工具化：产出静态代码检测工具、性能检测工具、错误检测工具、容灾检测工具、标准化物料使用覆盖率检测工具等等
- 流程化：基于DevOps分别在项目开发阶段、CI 阶段、提测&发布阶段进行约束，实现 **通过**、**告警**、**阻断** 三种策略，进行项目的自动化验收

3. 前端工程质量约束策略

➤ 静态代码检测方案

- 基于抽象语法树（Abstract Syntax Tree, AST）进行静态代码分析

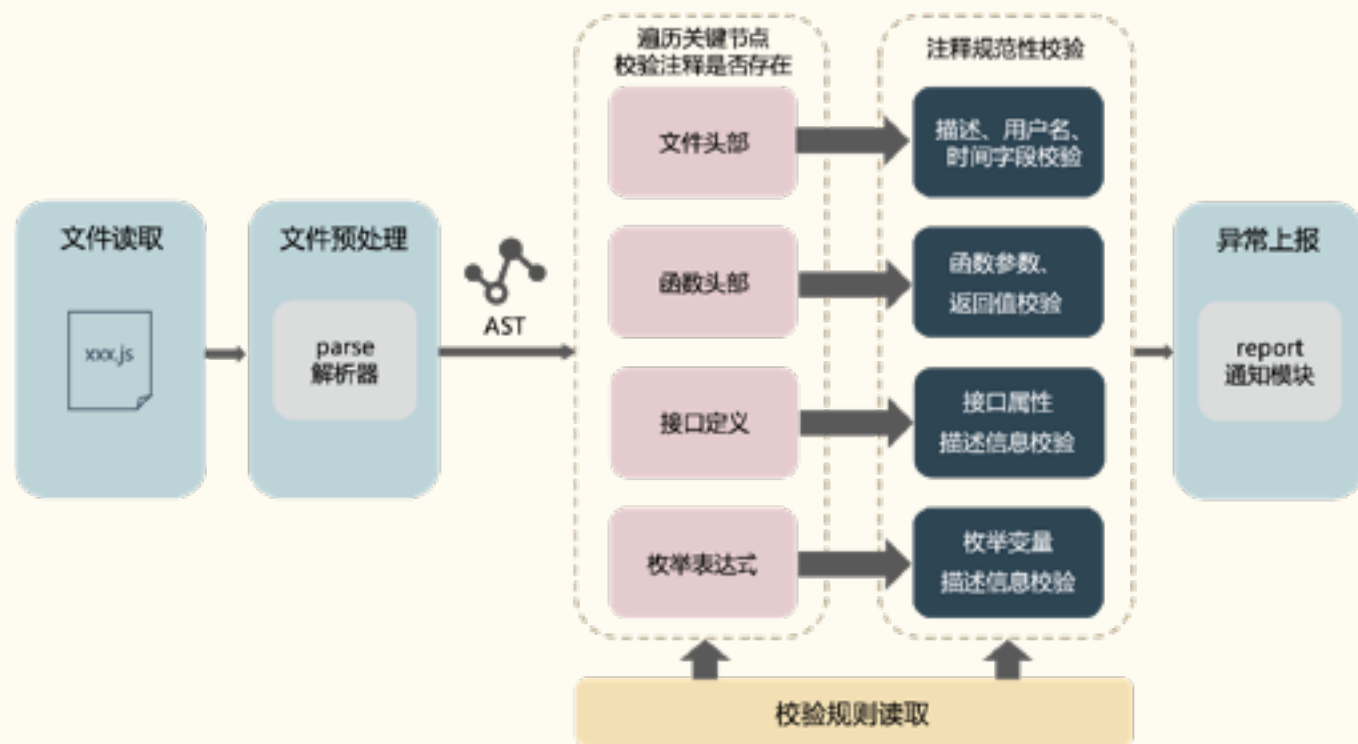


3. 前端工程质量约束策略

➤ 静态代码检测方案

1. 代码注释检测：扩展 Eslint plugin——开发 eslint-plugin-comments。好处是不用再次遍历代码，还可以直接使用 Eslint 完善的告警机制和规则配置。

工具化



3. 前端工程质量约束策略

➤ 静态代码检测方案

Eslint plugin有其固定的开发模版，主要由 meta 和 create 两部分组成，其中：

- meta：代表了这条规则的元数据，如其类别，文档，可接收的参数的 schema 等等
- create：如果说 meta 表达了我们想做什么，那么 create 则表达了这条 rule 具体会怎么分析代码

工具化

```
export default {
  // 规则元数据，用于描述规则
  meta: {
    type: 'suggestion',
    // 用于配置参数的 schema
    schema: [],
    // 文档地址
    docs: {
      description: '',
      url: 'https://tech.qq.com/opensource/center/develop/guide/develop-from-the-plugin.html',
    },
  },
  create(context) {
    // 规则实现逻辑
    return {
      Program(node) {
        fileCount++;
        const sourceCode = context.getSourceCode();
        const comments = sourceCode.getAllComments();
        const firstComment = comments[0] || {};
        if (firstComment.type === 'Block' && firstComment.range[0] < firstComment.range[2] === 2) {
          return;
        } else {
          context.report({
            loc: {
              start: {
                column: 1,
                line: 1,
              },
            },
            message: 'Missing Heading Comment. currentCheckCount: >{fileCount}',
          });
        }
      },
    };
  },
};
```



```
export default {
  rules: {
    'require-file-comment': requireFileComment,
    'require-function-comment': requireFunctionComment,
    'require-interface-comment': requireInterfaceComment,
    'require-enum-comment': requireEnumComment,
    'require-file-description': requireFileDescription,
    'require-file-author': requireFileAuthor,
    'require-file-date': requireFileDate,
    'require-function-description': requireFunctionDescription,
    'require-function-params': requireFunctionParam,
    'require-function-params-description': requireFunctionParamDescription,
    'require-function-param-name': requireFunctionParamName,
    'require-function-param-type': requireFunctionParamType,
    'require-function-returns': requireFunctionReturns,
    'require-function-returns-description': requireFunctionReturnsDescription,
    'require-function-returns-type': requireFunctionReturnsType,
    'no-undefined-types': noUndefinedTypes
  },
};
```

3. 前端工程质量约束策略

➤ 静态代码检测方案

我们产出的插件 `eslint-plugin-comments` 的使用效果如下：

```
import React, { lazy } from 'react';
```

```
Missing Heading Comment. currentCheckCount: 5 eslint(@fe-  
sdk/comments/require-file-comment)
```

查看问题 (⇧F8) 快速修复... (⌘。)

```
  constructor(props) {  
    super(props);  
    this.state = {  
      template: 'I am yxnode template'  
    };  
  }  
}
```


3. 前端工程质量约束策略

➤ 静态代码检测方案

2. README.md检测：由于文件类型的不同，js和markdown对应的AST节点类型是不同的。所以需要特定的编译器去解析markdown文件。这里我们用到的是结构化文本处理利器 —— **unified**。

工具化

unified

remark

rehype

retext

redot

unist

vfile

mdx

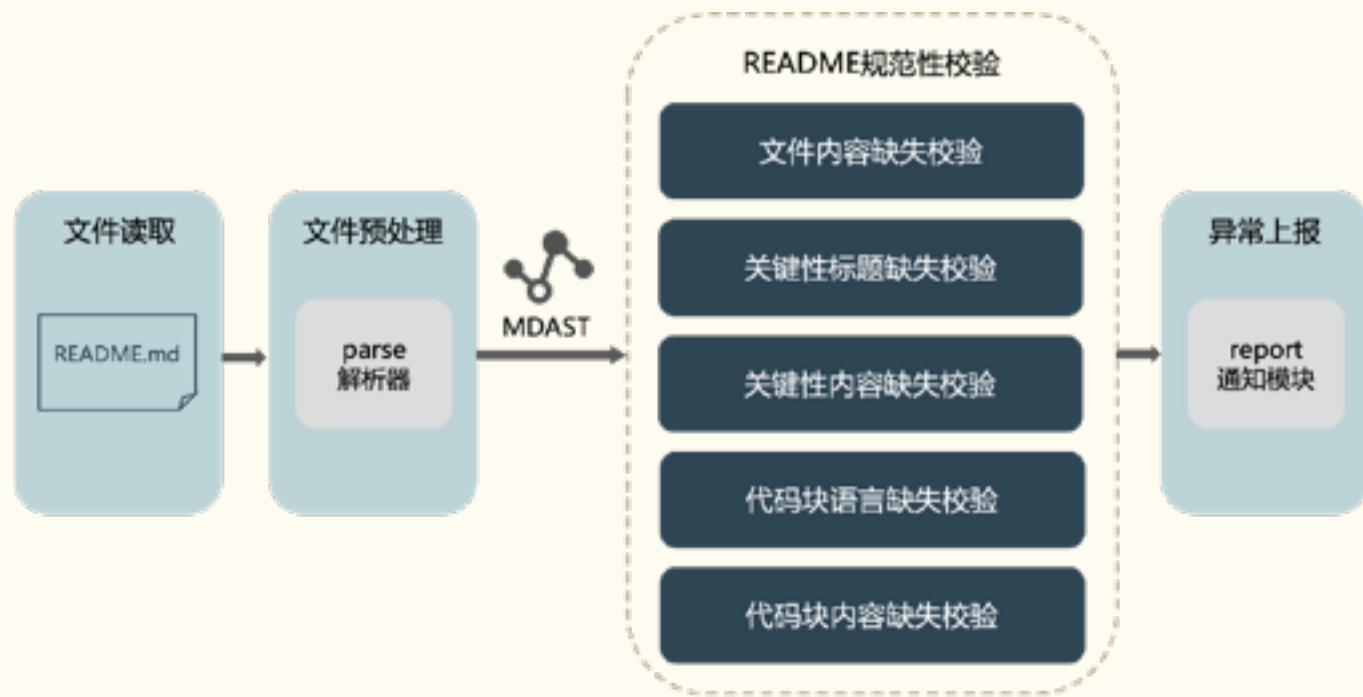
unified 生态相关插件

3. 前端工程质量约束策略

➤ 静态代码检测方案

我们使用 unified 生态下的 **remark-parse** 插件，它可以提供解析 Markdown 的能力，将markdown 文本转换为对应的 AST —— **MDAST**，进而遍历 AST 节点进行一系列的规则校验。

工具化



3. 前端工程质量约束策略

➤ 静态代码检测方案

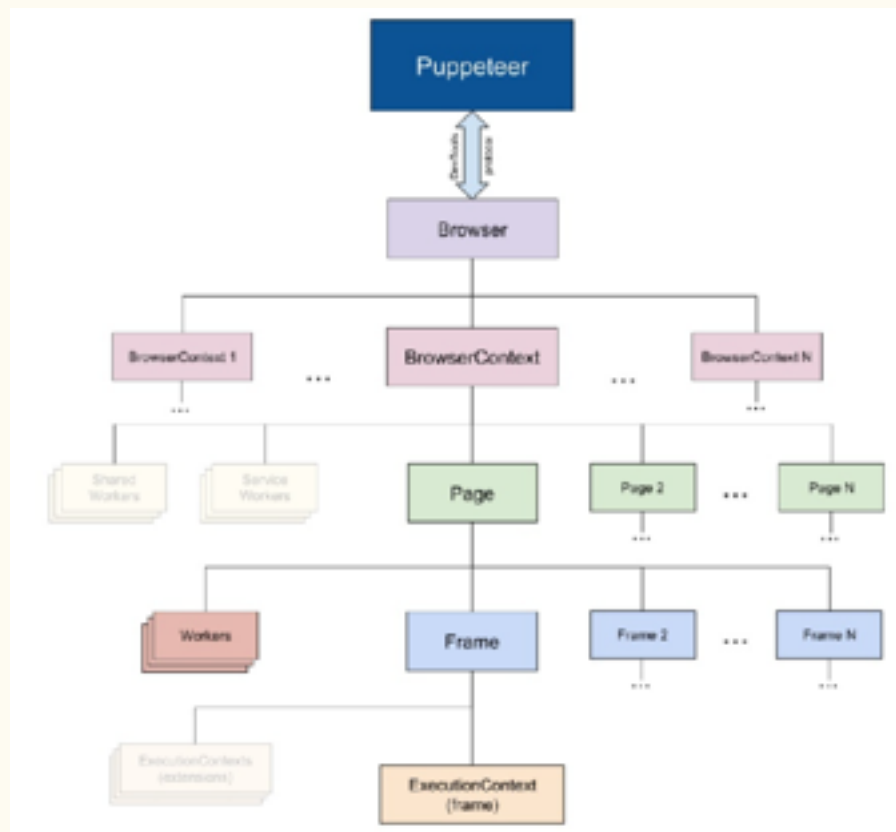
最后，基于上述静态代码检测工具输出检测结果日志 lint-analyze.log，通过分析脚本对其进行分
析，可以得出工程的代码质量评分和知识沉淀质量评分，然后将数据通过埋点上报到数据平台上进
行可视化分析。

3. 前端工程质量约束策略

➤ 错误检测方案

Puppeteer 是Google 提供的无头浏览器。本质上是一个 Node 库，它基于 DevTools 协议提供了不少高度封装的接口方便控制Chrome浏览器。

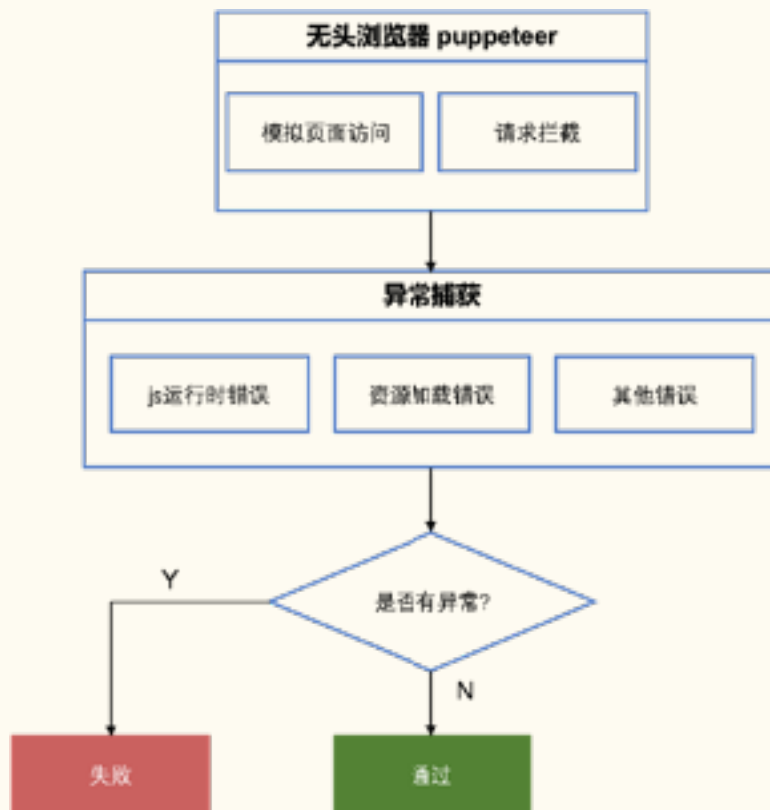
Puppeteer的结构也反映了浏览器的结构



3. 前端工程质量约束策略

➤ 错误检测方案

通过无头浏览器 puppeteer 模拟正常网络下的页面访问，对目标页面进行错误检测：



3. 前端工程质量约束策略

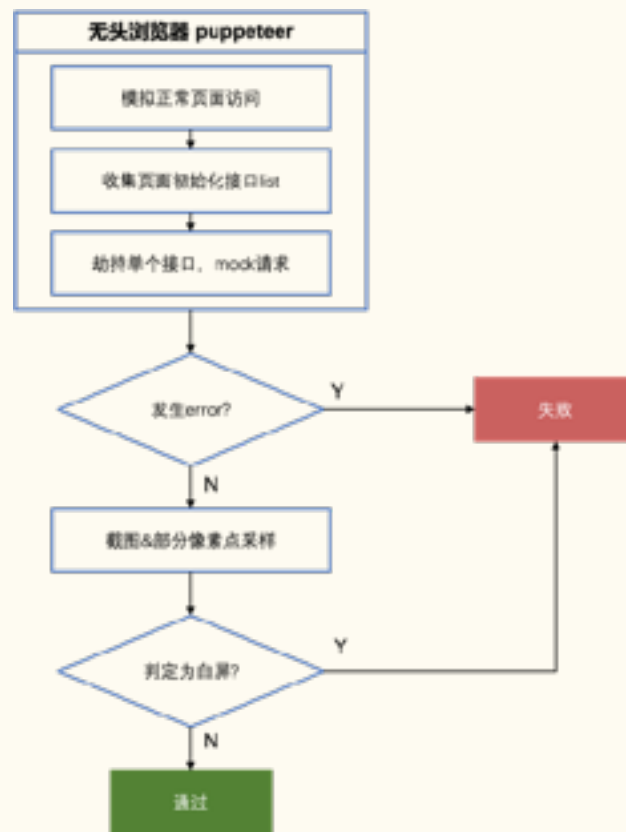
➤ 容灾能力检测方案

评估页面在渲染或执行核心流程时，各个接口异常导致的结果，从而暴露出页面容灾的不足。

主要的评估指标是白屏率。

- 出现白屏或异常的请求数 -> X
- 页面初始化接口总数 -> Y

$$\text{页面白屏率} = X / Y$$

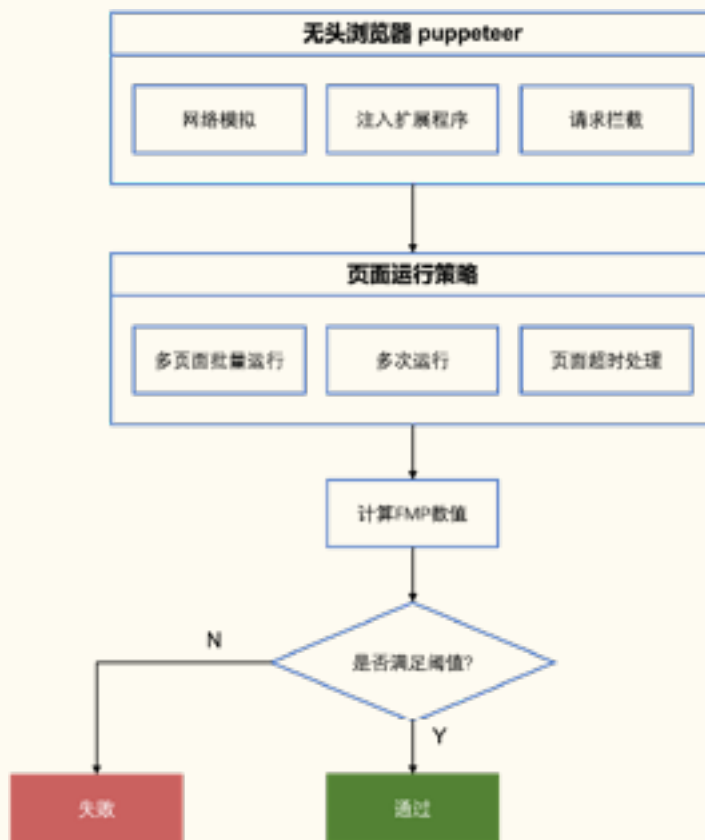


3. 前端工程质量约束策略

➤ 性能检测方案

通过 puppeteer 工具模拟网络，模拟页面访问，对目标页面进行性能检测：

- 页面注入性能测试插件，计算FMP
- 配合一系列页面运行策略



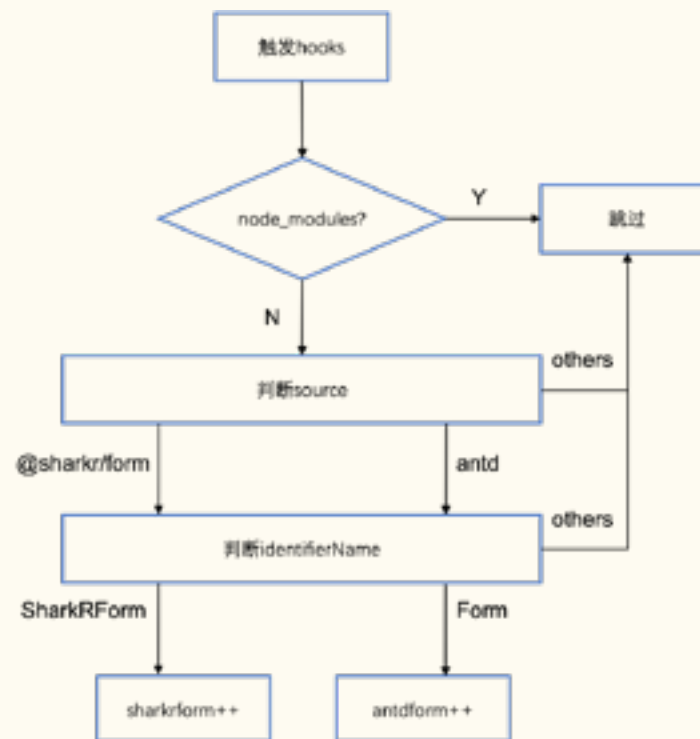
3. 前端工程质量约束策略

➤ 标准化物料覆盖率统计方案

通过 webpack 暴露的钩子 importSpecifier 对标准化物料和非标准化物料的引用次数进行统计，代码构建完成后计算出比例。数据将通过埋点上报到数据平台进行可视化分析。

- 标准化组件库的引用次数 -> X
- 其他组件库的引用次数 -> Y

$$\text{工程标准化组件覆盖率} = X / (X + Y)$$



3. 前端工程质量约束策略

➤ 项目开发阶段——结合git hooks设置本地卡点

流程化

1. 基于 ESLint 在编辑器中进行
代码与注释规范的 soft lint

2. 配置 husky 来操作 git hooks
实现 hard lint

pre-commit 阶段进行 eslint 与 readmelint

commit-msg 阶段进行 commitlint

post-commit 阶段生成 changelog

3. 前端工程质量约束策略

➤ 项目开发阶段——结合git hooks设置本地卡点

流程
化

增加约束的同时保证体验

使用 lint-staged 仅对本次提交的内容进行 lint

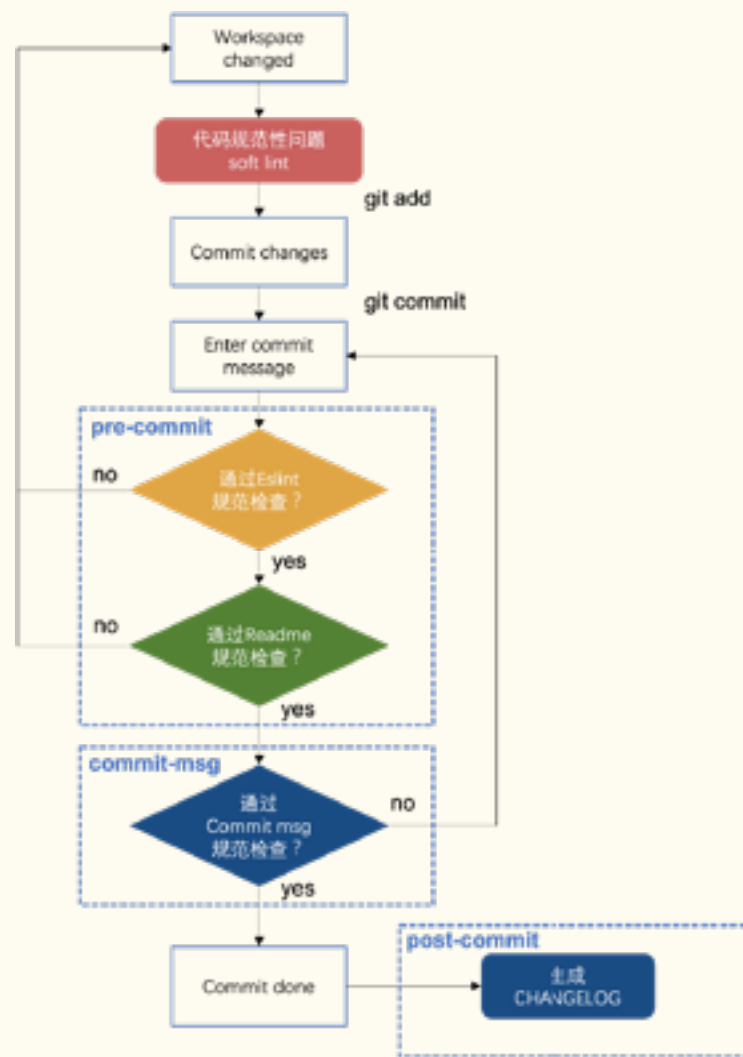
使用 commitizen、cz-git 辅助填写规范化的commit msg

prepare-commit-msg 阶段将 git commit 改写为 git cz

CHANGELOG.md 发生变更时自动 add、commit

3. 前端工程质量约束策略

➤ 项目开发阶段整体约束流程



流程化

3. 前端工程质量约束策略

➤ 项目CI阶段——.gitlab-ci.yml 增加 lint 任务形成卡点

流程化

增加约束的同时保证效率

任务的并行代替串行

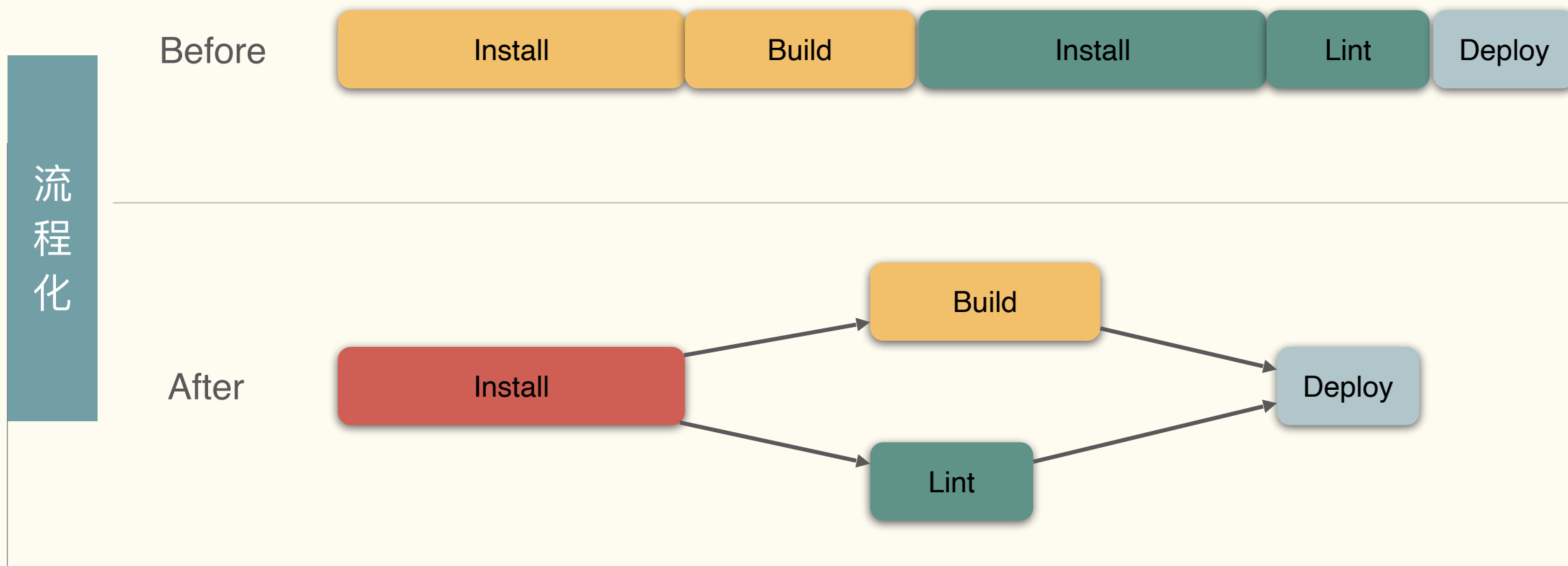
将 Install 过程前置

适时利用 Cache 机制

CI lint 阶段包含：eslint、readmelint、标准化物料覆盖率检测，以及上述检测结果的上报

3. 前端工程质量约束策略

➤ 项目CI阶段——.gitlab-ci.yml 增加 lint 任务形成卡点



3. 前端工程质量约束策略

- 项目CI阶段——.gitlab-ci.yml 增加 lint 任务形成卡点

流程化

Before

dev 2e045910



Merge branch 'feature-test' int...



00:06:08



4 weeks ago

After

dev 40f0b47c



Merge branch 'feature-test' int...



00:03:03



4 weeks ago

3. 前端工程质量约束策略

➤ 与 git hooks 的不同

- git hooks 是客户端检查
- CI 是服务端检查

客户端检查是天生不可信任的！

流程化

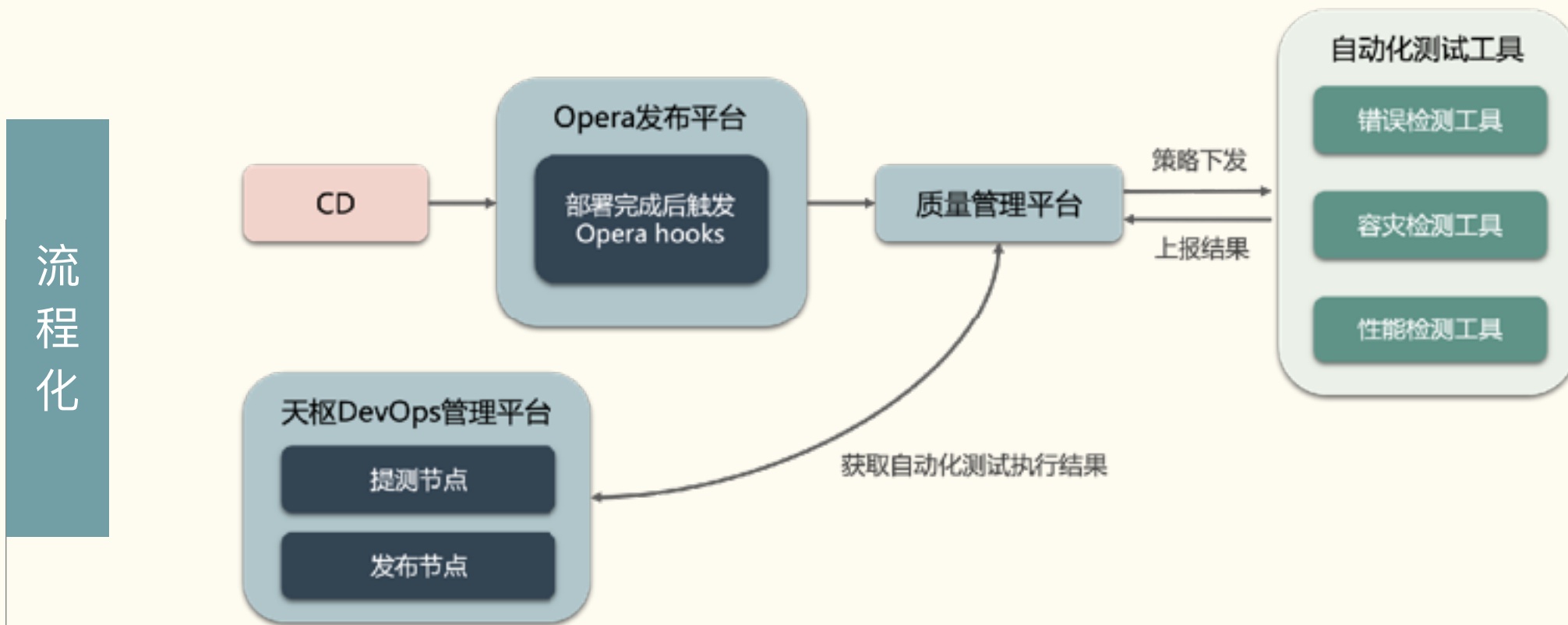
Client side validation



7 1.2K 2.5K

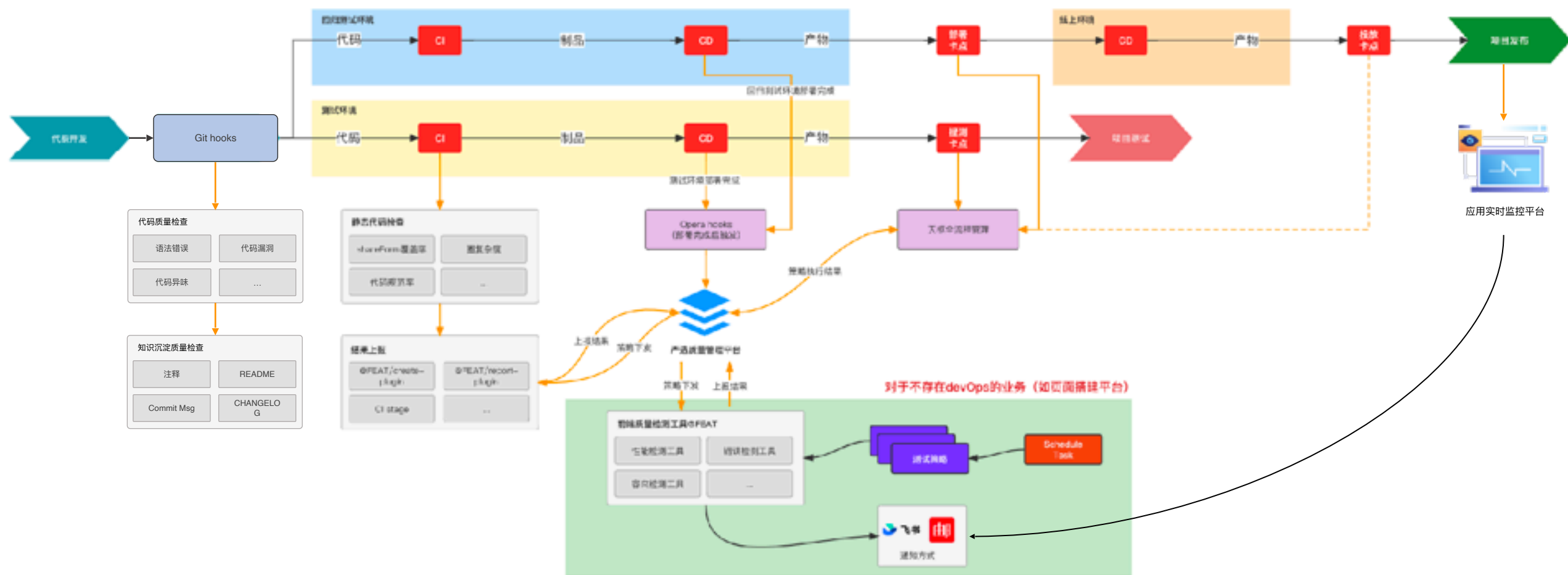
3. 前端工程质量约束策略

➤ 项目提测、发布阶段——基于DevOps的流程卡点



3. 前端工程质量约束策略

➤ 全景图

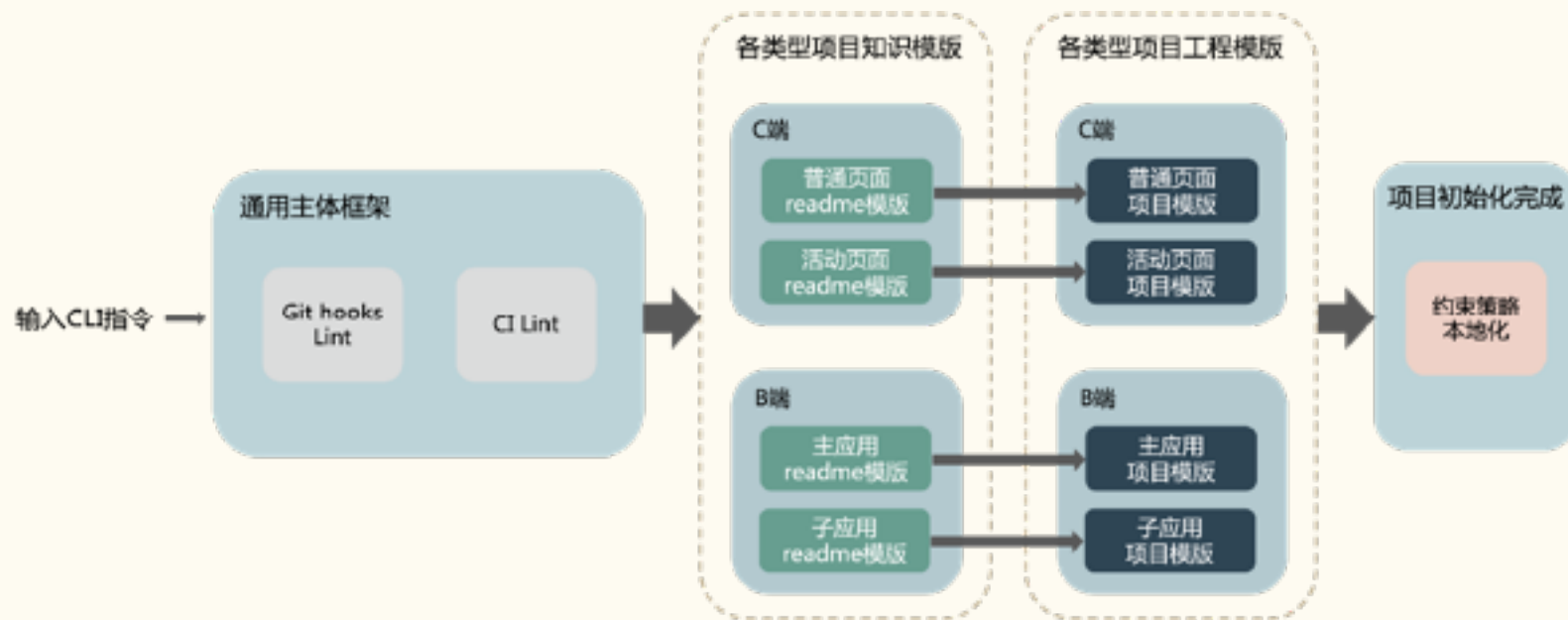


3. 前端工程质量约束策略

➤ 渐进式落地

1. 与项目工程强相关的约束策略内置于工程模版中：

- 工程模版内置标准化的知识模版及知识自动化生成能力
- 工程模版内置项目开发阶段和CI阶段的约束策略



3. 前端工程质量约束策略

➤ 渐进式落地

2. 与项目工程非强相关的约束策略以 API 的形式嵌入质量平台：

- 质量平台可定制化地针对特定服务，在特定节点（提测/发布）上配置指定的质量约束策略



总结

总结

- 通过项目案例，分析研发生命周期中常见的工程质量问题
- 讨论前端工程质量的本质，可以分为代码质量、知识沉淀质量、用户体验质量、研发标准化程度
- 基于行业标准和团队实际情况制定规范，并建立度量体系，形成完整的项目验收标准
- 分别针对四类工程质量问题制定相应的约束策略，通过工具化、流程化的方式嵌入到前端项目全流程中，使得质量保障体系形成闭环

THANK YOU

