

# 1. Bravo项目

## 1.1 说明

Bravo项目是基于Alpha项目进行升级改造的项目，要彻底告别alpha项目先天性测试体系缺陷，文档混乱，CICD阻碍，代码臃肿等情况。

### 1.1.1 最佳实践

通过询问kimi和deepseek把各个功能步骤的最佳实践更新到这里，然后再让cursor按照这里的思路去进行项目开发。确保细节把握

## 1.2 项目初步框架：

### 1.2.2 Bravo项目级基础框架

Bravo/	# → 项目根目录
├─ .cursor/	# → AI开发助手配置
└─ rules/	# → AI规则定义
└─ django_split.mdc.keep	# → Django应用拆分规则模板
└─ vue_component.mdc.keep	# → Vue组件生成规则模板
└─ test_coverage.mdc.keep	# → 测试覆盖率规则模板
└─ code_quality.mdc.keep	# → 代码质量检查规则模板
└─ security_scan.mdc.keep	# → 安全扫描规则模板
└─ mcp.json.keep	# → MCP服务器配置模板
├─ .github/	# → GitHub配置
└─ workflows/	# → CI/CD工作流
└─ ci.yml.keep	# → 持续集成流水线模板
└─ e2e.yml.keep	# → 端到端测试流水线模板
└─ deploy-cloudrun.yml.keep	# → 云部署流水线模板
└─ nightly.yml.keep	# → 夜间构建流水线模板
└─ PULL_REQUEST_TEMPLATE.md.keep	# → Pull Request模板
└─ dependabot.yml.keep	# → 依赖自动更新配置模板
├─ .envs/	# → 环境变量配置
└─ .env.local.example	# → 本地环境示例
└─ .env.staging.example	# → 预发布环境示例
└─ .env.prod.example	# → 生产环境示例
├─ backend/	# → Django后端服务
└─ Dockerfile.keep	# → 容器构建配置模板
└─ requirements.txt.keep	# → 依赖入口文件模板
└─ .python-version.keep	# → Python版本锁定模板
└─ requirements/	# → 依赖管理
└─ base.txt.keep	# → 基础依赖模板
└─ local.txt.keep	# → 本地开发依赖模板
└─ test.txt.keep	# → 测试环境依赖模板
└─ prod.txt.keep	# → 生产环境依赖模板
└─ manage.py.keep	# → Django管理脚本模板
└─ logs/	# → 应用日志目录
└─ directory.keep	# → 目录占位文件
└─ bravo/	# → Django项目配置

			└─ __init__.py.keep	# → Python包标识模板
			├─ asgi.py.keep	# → ASGI服务器配置模板
			├─ wsgi.py.keep	# → WSGI服务器配置模板
			├─ settings/	# → 项目设置
			── __init__.py.keep	# → Python包标识模板
			── base.py.keep	# → 基础配置模板
			── local.py.keep	# → 本地开发配置模板
			── test.py.keep	# → 测试环境配置模板
			── prod.py.keep	# → 生产环境配置模板
			└─ urls.py.keep	# → 主URL路由配置模板
			├─ apps/	# → 业务应用模块
			── users/	# → 用户管理应用
			── __init__.py.keep	# → Python包标识模板
			── models.py.keep	# → 数据模型定义模板
			── api.py.keep	# → API路由定义模板
			── services/	# → 业务逻辑层
			── __init__.py.keep	# → Python包标识模板
			── selectors.py.keep	# → 数据查询封装模板
			── filters.py.keep	# → 数据过滤配置模板
			── tests/	# → 测试用例
			── __init__.py.keep	# → Python包标识模板
			── conftest.py.keep	# → 测试配置模板
			── test_models.py.keep	# → 模型测试模板
			── test_api.py.keep	# → API测试模板
			── test_integration.py.keep	# → 集成测试模板
			── test_performance.py.keep	# → 性能测试模板
			── factories.py.keep	# → 测试数据工厂模板
			── fixtures/	# → 测试夹具数据
			── initial_data.json.keep	# → 初始数据模板
			── blog/	# → 博客应用
			── (类似users结构，所有文件都带.keep后缀)	# → 类似用户应用结构模板
			── english/	# → 英语学习应用
			── (类似users结构，所有文件都带.keep后缀)	# → 类似用户应用结构模板
			── jobs/	# → 求职应用
			── (类似users结构，所有文件都带.keep后缀)	# → 类似用户应用结构模板
			── common/	# → 通用功能应用
			── __init__.py.keep	# → Python包标识模板
			── permissions.py.keep	# → 权限控制模板
			── pagination.py.keep	# → 分页配置模板
			── testing/	# → 测试工具
			── __init__.py.keep	# → Python包标识模板
			── factories.py.keep	# → 基础工厂类模板
			── helpers.py.keep	# → 测试辅助函数模板
			── ai_pipeline/	# → AI能力管道
			── __init__.py.keep	# → Python包标识模板
			── clients/	# → AI客户端
			── __init__.py.keep	# → Python包标识模板
			── openai_client.py.keep	# → OpenAI客户端模板
			── mock_client.py.keep	# → 模拟客户端模板(测试用)
			── services/	# → AI服务封装
			── __init__.py.keep	# → Python包标识模板
			── tasks.py.keep	# → 异步任务定义模板
			── tests/	# → AI管道测试

[illegible]

	— user.factory.ts	# → 用户数据工厂
	— blog.factory.ts	# → 博客数据工厂
	— coverage/	# → 覆盖率配置
	— __init__.py	# → Python包标识
	— index.html	# → pytest-cov / vitest 生成
	— vitest.config.ts	# → 覆盖率配置
	— src/	# → 源代码目录
	— main.ts.keep	# → 应用入口模板
	— App.vue.keep	# → 根组件模板
	— router/	# → 路由配置
	— __init__.py.keep	# → Python包标识模板
	— index.ts.keep	# → 路由定义模板
	— stores/	# → 状态管理
	— __init__.py.keep	# → Python包标识模板
	— index.ts.keep	# → Store定义模板
	— api/	# → API客户端
	— __init__.py.keep	# → Python包标识模板
	— users.ts.keep	# → 用户API模板
	— blog.ts.keep	# → 博客API模板
	— english.ts.keep	# → 英语API模板
	— components/	# → 组件目录
	— BaseButton.vue.keep	# → 基础按钮组件模板
	— BaseButton.spec.ts.keep	# → 按钮组件测试模板
	— BaseButton.stories.ts.keep	# → 按钮组件文档模板
	— views/	# → 页面组件
	— Login.vue.keep	# → 登录页面模板
	— Login.spec.ts.keep	# → 登录页面测试模板
	— Login.e2e.ts.keep	# → 登录页面E2E测试模板
	— composables/	# → 组合式函数
	— __init__.py.keep	# → Python包标识模板
	— useApi.ts.keep	# → API组合函数模板
	— i18n/	# → 国际化配置
	— __init__.py.keep	# → Python包标识模板
	— index.ts.keep	# → 国际化设置模板
	— utils/	# → 工具函数
	— __init__.py.keep	# → Python包标识模板
	— format.ts.keep	# → 格式化工具模板
	— styles/	# → 样式文件
	— __init__.py.keep	# → Python包标识模板
	— main.scss.keep	# → 主样式文件模板
	— types/	# → TypeScript类型定义
	— __init__.py.keep	# → Python包标识模板
	— user.ts.keep	# → 用户类型定义模板
	— api.ts.keep	# → API类型定义模板
	— .eslintrc.js.keep	# → ESLint代码检查配置模板
	— .stylelintrc.keep	# → StyleLint样式检查配置模板
	— playwright.config.ts.keep	# → Playwright测试配置模板
	— lighthouse.config.js.keep	# → Lighthouse性能检测配置模板
	— docs/	# → 全部非源码文档
	— 00_product/	# → 产品需求侧
	— PRD_v1.0.md.keep	# → 产品需求文档 (Markdown)
	— PRD_v2.0.md.keep	
	— feature_flags.md.keep	# → 功能开关地图

```
| | — 01_guideline/ # → 指导文档
| | | — onboarding.md.keep # → 新人 onboarding
| | | — api_guideline.md.keep # → API 设计规范
| | | — commit_rules.md.keep # → 提交消息约定
| | | — release_process.md.keep # → 发版流程
| | — 02_test_report/ # → 测试成果
| | | — 2024-09-04_sprint1.md.keep # → 迭代测试报告（Markdown）
| | | — 2024-09-04_coverage.html.keep -> ../tests/reports/coverage/index.html # → 软链
到 CI 生成页
| | | — allure/ # → 软链到 tests/allure-results
| | — 03_operate/ # → 运维侧
| | | — runbook.md.keep # → 故障 playbook
| | | — grafana_dashboard.md.keep # → 监控面板说明
| | | — oncall.md.keep # → 值班轮值表

| — e2e/ # → 端到端测试
| | — tests/ # → 测试用例
| | | — auth.login.spec.ts.keep # → 认证登录测试模板
| | | — english.learn.spec.ts.keep # → 英语学习测试模板
| | | — user-management.spec.ts.keep # → 用户管理测试模板
| | | — blog.spec.ts.keep # → 博客功能测试模板
| | | — fixtures/ # → 测试数据
| | | | — users.json.keep # → 用户测试数据模板
| | | | — posts.json.keep # → 文章测试数据模板
| | | — support/ # → 测试支持文件
| | | | — commands.ts.keep # → 自定义命令模板
| | | | — helpers.ts.keep # → 测试辅助函数模板
| | | — playwright.config.ts.keep # → Playwright配置模板
| | | — selenium.config.js.keep # → Selenium配置模板
| | | — package.json.keep # → E2E测试依赖模板
| | | — lighthouse/ # → Lighthouse性能测试
| | | | — config.js.keep # → 配置模板
| | | | — scripts/ # → 脚本
| | | | — run-audit.js.keep # → 执行审计模板
| — tests/ # → 项目级测试相关目录
| | — integration/ # → 集成测试
| | | — __init__.py.keep # → Python包标识模板
| | | — test_user_end_to_end.py.keep # → 用户端到端测试模板
| | | — test_api_integration.py.keep # → API集成测试模板
| | | — test_frontend_backend.py.keep # → 前后端集成测试模板
| | — system/ # → 系统测试
| | | — __init__.py.keep # → Python包标识模板
| | | — test_smoke.py.keep # → 冒烟测试模板
| | — load/ # → 负载测试
| | | — __init__.py.keep # → Python包标识模板
| | | — locustfile.py.keep # → Locust负载测试模板
| | | — k6/ # → k6性能测试
| | | | — load-test.js.keep # → 负载测试脚本模板
| | — security/ # → 安全测试
| | | — __init__.py.keep # → Python包标识模板
| | | — test_auth_security.py.keep # → 认证安全测试模板
| | | — test_api_security.py.keep # → API安全测试模板
| — reports/ # → 测试报告
```

.gitignore	# → Git忽略配置
README.md	# → 报告说明
allure-results/	# → Allure测试结果
.gitkeep	# → Git空目录占位
junit/	# → JUnit测试报告
.gitkeep	# → Git空目录占位
coverage/	# → 覆盖率报告
__init__.py.keep	# → Python包标识模板
index.html.keep	# → pytest-cov / vitest 生成
backend/	# → 后端覆盖率
.gitkeep	# → Git空目录占位
frontend/	# → 前端覆盖率
.gitkeep	# → Git空目录占位
merged/	# → 合并覆盖率
.gitkeep	# → Git空目录占位
allure-results/	
index.html.keep	# → Allure 报告
data/	# → 测试数据
__init__.py.keep	# → Python包标识模板
fixtures/	# → 测试夹具
sample_data.json.keep	# → 示例数据模板
factories/	# → 数据工厂
__init__.py.keep	# → Python包标识模板
base_factory.py.keep	# → 基础工厂模板
k8s/	# → Kubernetes配置
base/	# → 基础配置
__init__.py.keep	# → Python包标识模板
deployment.yaml.keep	# → 部署配置模板
service.yaml.keep	# → 服务配置模板
configmap.yaml.keep	# → 配置映射模板
secret.yaml.keep	# → 密钥配置模板
overlays/	# → 环境覆盖配置
development/	# → 开发环境
__init__.py.keep	# → Python包标识模板
kustomization.yaml.keep	# → Kustomize配置模板
patches/	# → 补丁文件
replica-count.yaml.keep	# → 副本数配置模板
staging/	# → 预发布环境
__init__.py.keep	# → Python包标识模板
kustomization.yaml.keep	# → Kustomize配置模板
patches/	# → 补丁文件
resource-limits.yaml.keep	# → 资源限制配置模板
production/	# → 生产环境
__init__.py.keep	# → Python包标识模板
kustomization.yaml.keep	# → Kustomize配置模板
patches/	# → 补丁文件
hpa.yaml.keep	# → 水平扩展配置模板
ingress.yaml.keep	# → 入口配置模板
monitoring/	# → 监控配置
prometheus.yaml.keep	# → Prometheus配置模板
grafana.yaml.keep	# → Grafana配置模板
helm-bravo/	# → Helm Chart
Chart.yaml.keep	# → Chart定义模板

		values.yaml.keep	# → 默认值模板		
		templates/	# → 模板文件		
				__init__.py.keep	# → Python包标识模板
				deployment.yaml.keep	# → 部署模板
				service.yaml.keep	# → 服务模板
				charts/	# → 子Chart
				.gitkeep	# → Git空目录占位
		scripts/	# → 项目脚本		
				dev.sh.keep	# → 开发环境启动模板
				build.sh.keep	# → 项目构建模板
				deploy.sh.keep	# → 项目部署模板
				test.sh.keep	# → 测试执行模板
				coverage.sh.keep	# → 覆盖率报告模板
				quality.sh.keep	# → 代码质量检查模板
				lighthouse.sh.keep	# → Lighthouse审计模板
		docker-compose.yaml.keep	# → 开发环境Docker编排模板		
		docker-compose.prod.yaml.keep	# → 生产环境Docker编排模板		
		docker-compose.test.yaml.keep	# → 测试环境Docker编排模板		
		Makefile.keep	# → 项目命令管理模板		
		pyproject.toml.keep	# → Python工具配置模板		
		.pre-commit-config.yaml.keep	# → 预提交钩子配置模板		
		.gitignore	# → Git忽略规则		
		cloudbuild.yaml.keep	# → 云构建配置模板		
		LICENSE	# → 项目许可证		
		SECURITY.md	# → 安全策略		
		CODE_OF_CONDUCT.md	# → 行为准则		
		CONTRIBUTING.md	# → 贡献指南		
		CHANGELOG.md	# → 变更日志		
		README.md	# → 项目说明		

模块级「放什么文件」速查 (AI 生成命令一并给)

模块	目录	关键文件	AI 一键生成 (Cursor)
用户域	backend/apps/users/	models.py api.py services/ selectors.py tests/	> MCP: Django-Ninja CRUD users + test
博客域	backend/apps/blog/	models.py api.py signals.py tests/	> MCP: Django blog app with tag & category
英语域	backend/apps/english/	models.py api.py ai_client.py tests/	> MCP: English learning domain with AI tutor
AI 管道	backend/ai_pipeline/	clients/ services/ tasks.py tests/	> MCP: AI pipeline with failover + cost monitor
前端组件	frontend/src/components/	xx.vue xx.spec.ts xx.stories.ts	> MCP: Vue3 SFC + test + story
前端视图	frontend/src/views/	xx.vue xx.spec.ts xx.e2e.ts	> MCP: Vue3 view with router + pinia + test

模块	目录	关键文件	AI 一键生成 (Cursor)
E2E	e2e/tests/	*.spec.ts	> MCP: Playwright E2E flow for english learning

一键跑命令 (Makefile)

```
dev:          ## 本地热重载
    docker compose up -d
install-be:   ## 后端安装 + 预提交钩子
    cd backend && python -m venv .venv && .venv/bin/pip install -r requirements/local.txt
install-fe:   ## 前端安装
    cd frontend && pnpm install
test:        ## 全量测试
    docker compose exec backend pytest && cd frontend && pnpm run test:ci && cd ../e2e &&
pnpm run test
coverage:    ## 合并覆盖率
    python -m coverage combine backend/tests/reports frontend/tests/reports && coverage html
deploy:      ## 推送到 CloudRun
    gcloud builds submit --config cloudbuild.yaml
```

1.2.3 原Alpha项目目录:

```
# Alpha 技术共享平台 - 完整项目目录框架

## 项目概述
Alpha 是一个现代化的技术共享平台，集成了英语学习、求职管理、待办笔记、AI助手和搜索功能。采用前后端分离架构，支持容器化部署。

## 整体目录结构

...

alpha/
├── .cursor/                                # Cursor IDE 配置
│   ├── mcp.json                          # MCP 服务器配置
│   └── rules/                            # 代码规则和工作流
│       ├── cursor_rules.mdc
│       ├── self_improve.mdc
│       └── taskmaster/
│           ├── dev_workflow.mdc
│           └── taskmaster.mdc
├── .github/                              # GitHub 工作流和模板
│   ├── ISSUE_TEMPLATE/
│   │   ├── example_requirement_template.md
│   │   └── idiomatic_expressions_requirement_template.md
│   └── workflows/
│       ├── ci.yml                        # 持续集成
│       ├── deploy-staged.yml            # 预发布部署
│       └── deploy.yml                    # 生产部署
```



```

|       |─ e2e.yml                # E2E 测试
|       |─ test.yml              # 单元测试
|─ .taskmaster/                  # TaskMaster AI 项目管理
|   |─ config.json               # 配置文件
|   |─ state.json                # 状态文件
|   |─ docs/                     # 项目需求文档
|   |   |─ ai-service-config-ui-prd.txt
|   |   |─ frontend-js-error-fix-analysis-prd.txt
|   |   |─ frontend-testing-and-fix-prd.txt
|   |   |─ idiomatic_expressions_prd.txt
|   |   |─ super_reliable_test_system_rebuild.txt
|   |   |─ TEST_SYSTEM_REBUILD_PLAN.md
|   |   |─ TEST_SYSTEM_REBUILD_PRD.md
|   |─ reports/                  # 任务复杂度报告
|   |   |─ task-complexity-report_ai-config-ui.json
|   |   |─ task-complexity-report_feature-test_task_master.json
|   |─ tasks/                    # 任务管理文件
|   |   |─ tasks.json
|   |   |─ js-error-fix-tasks.json
|   |   |─ task_*.txt
|─ ai_pipeline/                  # AI 管道服务
|   |─ config/
|   |─ models/
|   |─ processors/
|   |─ utils/
|─ backend/                      # Django 后端服务
|   |─ alpha/                    # Django 项目配置
|   |   |─ __init__.py
|   |   |─ asgi.py                # ASGI 入口
|   |   |─ celery.py              # Celery 配置
|   |   |─ settings.py            # Django 设置
|   |   |─ urls.py                # URL 路由
|   |   |─ wsgi.py                # WSGI 入口
|   |─ apps/                     # Django 应用模块
|   |   |─ ai/                    # AI 服务模块
|   |   |   |─ migrations/        # 数据库迁移
|   |   |   |─ services/          # AI 服务层
|   |   |   |   |─ cost_monitor.py
|   |   |   |   |─ degradation.py
|   |   |   |   |─ failover_service.py
|   |   |   |   |─ health_check_service.py
|   |   |   |   |─ health_monitor.py
|   |   |   |   |─ key_manager.py
|   |   |   |   |─ load_balancer.py
|   |   |   |   |─ manager.py
|   |   |   |   |─ model_discovery.py
|   |   |   |   |─ monitoring_service.py
|   |   |   |   |─ quality_assessor.py
|   |   |   |   |─ rate_limiter.py
|   |   |   |   |─ token_statistics.py
|   |   |   |   |─ tutor.py
|   |   |   |─ tests/              # AI 模块测试
|   |   |─ admin.py

```

```

| | | └─ apps.py
| | | └─ models.py
| | | └─ monitoring_models.py
| | | └─ routing.py
| | | └─ serializers.py
| | | └─ tasks.py
| | | └─ urls.py
| | | └─ views.py
| | | └─ websocket_service.py
| | └─ api/ # API 核心模块
| | | └─ cache_middleware.py
| | | └─ cache_strategy.py
| | | └─ cache_views.py
| | | └─ ...
| | └─ articles/ # 文章管理
| | └─ categories/ # 分类管理
| | └─ common/ # 公共组件
| | └─ english/ # 英语学习模块
| | | └─ migrations/
| | | └─ models/
| | | └─ views/
| | | └─ serializers/
| | | └─ tests/
| | └─ jobs/ # 求职管理
| | └─ links/ # 链接管理
| | └─ rbac/ # 权限控制
| | └─ search/ # 搜索功能
| | └─ todos/ # 待办事项
| | └─ users/ # 用户管理
| └─ feature_flags/ # 功能开关
| └─ tests/ # 后端测试
| └─ Dockerfile # Docker 构建文件
| └─ manage.py # Django 管理脚本
| └─ pytest.ini # pytest 配置
| └─ requirements.txt # Python 依赖
└─ docs/ # 项目文档
| | └─ api/ # API 文档
| | └─ deployment/ # 部署文档
| | └─ development/ # 开发文档
└─ e2e/ # E2E 测试配置
└─ frontend/ # Vue.js 前端应用
| | └─ src/ # 源代码目录
| | | └─ api/ # API 接口层
| | | | └─ ai.ts
| | | | └─ auth.ts
| | | | └─ english.ts
| | | | └─ jobs.ts
| | | | └─ ...
| | | └─ components/ # Vue 组件
| | | | └─ ai/ # AI 相关组件
| | | | | └─ AIAssistantChat.vue
| | | | | └─ AIConfigPanel.vue
| | | | └─ ...

```

```
| | | | └─ common/ # 通用组件
| | | | | └─ BaseButton.vue
| | | | | └─ BaseCard.vue
| | | | | └─ BaseDialog.vue
| | | | | └─ ...
| | | | └─ english/ # 英语学习组件
| | | | | └─ ExpressionCard.vue
| | | | | └─ LearningDashboard.vue
| | | | | └─ LearningChart.vue
| | | | | └─ ...
| | | | └─ jobs/ # 求职管理组件
| | | | └─ layout/ # 布局组件
| | | | | └─ AppHeader.vue
| | | | | └─ AppSidebar.vue
| | | | | └─ AppFooter.vue
| | | | └─ todos/ # 待办事项组件
| | └─ composables/ # Vue 组合式函数
| | | └─ useAuth.ts
| | | └─ useApi.ts
| | | └─ useNotification.ts
| | | └─ ...
| | └─ directives/ # Vue 指令
| | └─ hooks/ # React 风格的 hooks
| | └─ i18n/ # 国际化
| | | └─ locales/
| | | | └─ en.json
| | | | └─ zh.json
| | | └─ index.ts
| | └─ mocks/ # Mock 数据
| | └─ router/ # Vue Router 配置
| | | └─ index.ts
| | | └─ routes.ts
| | └─ services/ # 业务服务层
| | | └─ aiService.ts
| | | └─ authService.ts
| | | └─ englishService.ts
| | | └─ ...
| | └─ stores/ # Pinia 状态管理
| | | └─ aiStore.ts
| | | └─ authStore.ts
| | | └─ englishStore.ts
| | | └─ learningStore.ts
| | | └─ ...
| | └─ styles/ # 样式文件
| | | └─ global.css
| | | └─ variables.css
| | | └─ components/
| | └─ utils/ # 工具函数
| | | └─ api.ts
| | | └─ auth.ts
| | | └─ constants.ts
| | | └─ format.ts
| | | └─ validation.ts
```

```

| | | | └─ views/                                # 页面组件
| | | | |   └─ ai/
| | | | |   |   └─ AIAssistant.vue
| | | | |   |   └─ AIConfig.vue
| | | | |   └─ auth/
| | | | |   |   └─ Login.vue
| | | | |   |   └─ Register.vue
| | | | |   └─ dashboard/
| | | | |   |   └─ Dashboard.vue
| | | | |   └─ english/
| | | | |   |   └─ ExpressionLearning.vue
| | | | |   |   └─ LearningProgress.vue
| | | | |   |   └─ vocabulary.vue
| | | | |   └─ jobs/
| | | | |   |   └─ JobList.vue
| | | | |   |   └─ JobDetail.vue
| | | | |   └─ todos/
| | | | |   |   └─ TodoList.vue
| | | | |   |   └─ TodoDetail.vue
| | | | └─ App.vue                                # 根组件
| | | └─ main.ts                                # 应用入口
| | └─ tests/                                    # 前端测试
| | | └─ e2e/                                    # E2E 测试
| | | | └─ idiomatic-expressions.spec.ts
| | | | └─ ...
| | | └─ integration/                            # 集成测试
| | | └─ selenium/                              # Selenium 测试
| | | └─ unit/                                  # 单元测试
| | | | └─ components/
| | | | └─ stores/
| | | | └─ utils/
| | └─ ci-reports/                              # CI 报告
| | └─ playwright-report/                       # Playwright 测试报告
| | └─ package.json                             # 前端依赖配置
| | └─ vite.config.js                           # Vite 构建配置
| | └─ vitest.config.ts                         # Vitest 测试配置
| | └─ playwright.config.ts                     # Playwright 配置
| | └─ tsconfig.json                            # TypeScript 配置
| | └─ tailwind.config.js                       # Tailwind CSS 配置
| └─ htmlcov/                                  # 测试覆盖率报告
| └─ issues/                                   # 问题跟踪
| └─ k8s/                                       # Kubernetes 部署配置
| | └─ deployments/
| | └─ services/
| | └─ configmaps/
| | └─ ingress/
| └─ mysql/                                    # MySQL 配置和初始化
| | └─ init/
| | └─ conf/
| └─ nginx/                                   # Nginx 配置
| | └─ nginx.conf
| | └─ ssl/
| └─ scripts/                                # 部署和维护脚本

```

```
|   |─ deploy.sh
|   |─ backup.sh
|   └─ migrate.sh
└─ tests/                                # 项目级测试
    |─ integration/
    └─ performance/
└─ .cursorrules                         # Cursor 规则配置
└─ .cz.json                             # Commitizen 配置
└─ .env                                 # 环境变量
└─ .env.example                         # 环境变量示例
└─ .flake8                              # Flake8 配置
└─ .gitignore                           # Git 忽略文件
└─ .pre-commit-config.yaml              # Pre-commit 钩子配置
└─ .releaserc.json                      # 语义化版本发布配置
└─ docker-compose.yml                  # Docker Compose 配置
└─ package.json                        # 项目根依赖
└─ pyproject.toml                      # Python 项目配置
└─ README.md                           # 项目说明文档
...
```

## ## 核心模块详解

### ### 1. 前端架构 (frontend/)

- \*\*技术栈\*\*: Vue 3 + TypeScript + Vite + Element Plus + Pinia
- \*\*组件化\*\*: 按功能模块组织组件 (ai/, english/, jobs/, todos/)
- \*\*状态管理\*\*: Pinia stores 管理全局状态
- \*\*路由管理\*\*: Vue Router 实现 SPA 路由
- \*\*测试体系\*\*: Vitest (单元) + Playwright (E2E) + Selenium (集成)
- \*\*构建工具\*\*: Vite 提供快速开发和构建

### ### 2. 后端架构 (backend/)

- \*\*技术栈\*\*: Django + Django REST Framework + Celery + Redis
- \*\*应用模块\*\*: 按业务功能划分 (ai/, english/, jobs/, users/ 等)
- \*\*AI 服务\*\*: 完整的 AI 服务管理系统, 包含负载均衡、健康监控、故障转移
- \*\*数据库\*\*: MySQL 主数据库 + Redis 缓存
- \*\*异步任务\*\*: Celery 处理后台任务
- \*\*API 设计\*\*: RESTful API + WebSocket 实时通信

### ### 3. AI 管道 (ai\_pipeline/)

- \*\*模型管理\*\*: AI 模型的加载、配置和版本管理
- \*\*数据处理\*\*: 输入预处理和输出后处理
- \*\*服务集成\*\*: 与后端 AI 服务模块集成

### ### 4. 部署与运维

- \*\*容器化\*\*: Docker + Docker Compose 本地开发
- \*\*编排\*\*: Kubernetes 生产环境部署
- \*\*CI/CD\*\*: GitHub Actions 自动化流水线
- \*\*监控\*\*: 集成监控和日志系统
- \*\*负载均衡\*\*: Nginx 反向代理和负载均衡

### ### 5. 开发工具链

- \*\*代码质量\*\*: ESLint, Prettier, Black, isort, flake8
- \*\*测试覆盖\*\*: 前后端完整测试覆盖

- **\*\*版本控制\*\***: **Git** + 语义化版本管理
- **\*\*项目管理\*\***: **TaskMaster AI** 智能项目管理
- **\*\*IDE 集成\*\***: **Cursor IDE** 深度集成

## ## 功能模块说明

### ### 英语学习模块 (english/)

- 习语表达式学习和练习
- 学习进度跟踪和统计
- 个性化学习推荐
- 发音练习和评估

### ### AI 助手模块 (ai/)

- 多模型支持和智能切换
- 实时对话和上下文管理
- 成本监控和使用统计
- 服务健康监控和故障恢复

### ### 求职管理模块 (jobs/)

- 职位信息管理
- 简历和求职进度跟踪
- 面试安排和记录

### ### 待办事项模块 (todos/)

- 任务创建和管理
- 优先级和截止日期
- 进度跟踪和提醒

### ### 用户与权限模块 (users/, rbac/)

- 用户认证和授权
- 角色和权限管理
- 用户配置和偏好设置

## ## 技术特色

1. **\*\*现代化技术栈\*\***: 采用最新的前后端技术
2. **\*\*微服务架构\*\***: 模块化设计, 易于扩展和维护
3. **\*\*AI 深度集成\*\***: 完整的 **AI** 服务管理和监控体系
4. **\*\*完整测试覆盖\*\***: 单元、集成、**E2E** 全方位测试
5. **\*\*自动化部署\*\***: **CI/CD** 流水线和容器化部署
6. **\*\*智能项目管理\*\***: **TaskMaster AI** 辅助开发流程
7. **\*\*国际化支持\*\***: 多语言界面和内容
8. **\*\*性能优化\*\***: 缓存策略、懒加载、代码分割
9. **\*\*安全保障\*\***: 认证授权、数据加密、安全审计
10. **\*\*监控运维\*\***: 完整的监控、日志和告警体系

## ## 开发环境搭建

1. **\*\*前端开发\*\***:

```
```bash
cd frontend
npm install
npm run dev
```

## 2. \*\*后端开发\*\*:

```
```bash
cd backend
pip install -r requirements.txt
python manage.py runserver
```
```

## 3. \*\*完整环境\*\*:

```
```bash
docker-compose up -d
```
```

## ## 测试运行

### 1. \*\*前端测试\*\*:

```
```bash
npm run test:unit    # 单元测试
npm run test:e2e     # E2E 测试
```
```

### 2. \*\*后端测试\*\*:

```
```bash
pytest                # 单元测试
python manage.py test # Django 测试
```
```

这个项目展现了现代 **web** 应用开发的最佳实践，集成了 **AI** 技术、完整的测试体系、自动化部署和智能项目管理，是一个功能完整、技术先进的企业级应用平台。

## 1.3 项目设计方向

### 1.3.4 项目规则文件设计

```
.cursor/rules/
├─ code_quality.mdc.keep    # 代码质量标准
├─ django_split.mdc.keep    # Django架构规范
├─ project_startup.mdc.keep # 项目启动规范  新增
├─ security_scan.mdc.keep   # 安全扫描规则
├─ test_coverage.mdc.keep   # 测试覆盖率要求
└─ vue_component.mdc.keep   # Vue组件规范
```

### 1.3.5 首先使用虚拟环境

使用 requirements.txt 文件，在另一台设备上的正确操作流程

```
# 1. 克隆项目
git clone <repository-url>
cd Bravo
```

```
# 2. 创建新的虚拟环境
python -m venv venv

# 3. 激活虚拟环境
# Windows (Git Bash)
source venv/Scripts/activate
# Windows (PowerShell)
venv\Scripts\Activate.ps1
# Linux/macOS
source venv/bin/activate

# 4. 升级 pip
python -m pip install --upgrade pip

# 5. 安装依赖
cd backend
pip install -r requirements/local.txt

# 6. 配置环境变量
cp .envs/env.local.example .env
# 编辑 .env 文件，设置正确的配置

# 7. 初始化数据库
python manage.py migrate
python manage.py createsuperuser
```

## 1.3.6 使用docker开发，不走主机开发，保证项目环境一致性

### 1.3.7 服务概览

| 服务    | 镜像               | 端口        | 容器名              |
|-------|------------------|-----------|------------------|
| MySQL | mysql:8.0        | 3307:3306 | bravo-mysql-1    |
| Redis | redis:7-alpine   | 6379:6379 | bravo-redis-1    |
| 后端    | python:3.11-slim | 8000:8000 | bravo-backend-1  |
| 前端    | node:18-alpine   | 3000:3000 | bravo-frontend-1 |

#### 1.3.7.1 环境变量配置

| 变量名          | 值                                                         |
|--------------|-----------------------------------------------------------|
| DATABASE_URL | mysql://bravo:bravo@mysql:3306/bravo                      |
| REDIS_URL    | redis://redis:6379/0                                      |
| VITE_API_URL | <a href="http://localhost:8000">http://localhost:8000</a> |



1.3.7.2 数据卷映射

| 服务    | 卷名称          | 容器内路径             | 宿主机路径      | 用途         |
|-------|--------------|-------------------|------------|------------|
| MySQL | mysql_data   | /var/lib/mysql    | 自动管理       | MySQL数据持久化 |
| Redis | redis_data   | /data             | 自动管理       | Redis数据持久化 |
| 后端    | backend_logs | /app/logs         | 自动管理       | 后端日志存储     |
| 后端    | 绑定挂载         | /app              | ./backend  | 代码实时同步     |
| 前端    | 绑定挂载         | /app              | ./frontend | 代码实时同步     |
| 前端    | 匿名卷          | /app/node_modules | 自动管理       | 依赖隔离       |

1.3.7.3 存储位置

- Windows: \\wsl\$\docker-desktop-data\data\docker\volumes\
- 查看命令: docker volume ls 和 docker volume inspect 卷名

1.3.7.4 关键文件

docker-compose.yml 开发环境编排

backend/Dockerfile.dev 后端开发镜像

frontend/Dockerfile.dev 前端开发镜像

.env.local.example 环境变量模板

1.3.7.4.1 后端依赖

- 📄 文件结构

```
backend/requirements/
├─ base.txt      # 基础依赖（所有环境共用）
├─ local.txt     # 本地开发环境
├─ prod.txt      # 生产环境
└─ test.txt      # 测试环境
```

- 🛠 各环境说明

| 环境文件      | 使用命令                     | 核心功能 | 关键依赖                                    |
|-----------|--------------------------|------|-----------------------------------------|
| base.txt  | pip install -r base.txt  | 基础依赖 | Django 4.2.7, MySQL, Redis, Celery, JWT |
| local.txt | pip install -r local.txt | 本地开发 | debug-toolbar, 代码质量工具, 测试框架             |
| prod.txt  | pip install -r prod.txt  | 生产部署 | gunicorn, 监控, 安全优化                      |
| test.txt  | pip install -r test.txt  | 测试环境 | pytest, 测试数据, 性能测试                      |



|  |                         |                              |
|--|-------------------------|------------------------------|
|  | └─ 03_operate/          | # → 运维侧                      |
|  | ├─ runbook.md           | # → 故障 playbook              |
|  | ├─ grafana_dashboard.md | # → 监控面板说明                   |
|  | └─ oncall.md            | # → 值班轮值表                    |
|  | └─ tests/               | # → CI 生成的 html/soft-link    |
|  | ├─ coverage/            |                              |
|  | ├─├─ index.html         | # → pytest-cov / vitest 生成   |
|  | └─ allure-results/      |                              |
|  | ├─├─ index.html         | # → Allure 报告                |
|  | └─ .github/             |                              |
|  | ├─ workflows/           |                              |
|  | └─├─ ci.yml             | # → CI 步骤: 生成报告 → 推 gh-pages |

1.3.8.8 文档粒度与命名规则

| 类型    | 粒度    | 命名模板                      | 跟踪格式               |
|-------|-------|---------------------------|--------------------|
| PRD   | 迭代    | PRD_v{major}.{minor}.md   | Markdown           |
| 测试报告  | 迭代/周  | {YYYY-MM-DD}_{sprintX}.md | Markdown + 软链 HTML |
| 操作手册  | 功能级   | {module}_runbook.md       | Markdown           |
| 可视化报告 | 每次 CI | index.html (自动覆盖)         | HTML (软链)          |

1.3.8.9 CI 自动化 (GitHub Actions 片段)

```
- name: Generate docs
  run: |
    pytest --cov --cov-report=html:docs/02_test_report/coverage
    allure generate tests/allure-results -o docs/02_test_report/allure --clean

- name: Deploy Docs to GitHub Pages
  uses: peaceiris/actions-gh-pages@v3
  with:
    github_token: ${ secrets.GITHUB_TOKEN }
    publish_dir: ./docs
```

→ 每次 push 自动更新**测试覆盖率 + Allure 报告**在线浏览地址。

1.3.8.10 二进制大文件 (>5MB) 处理

- 1. 本地: docs/02\_test\_report/2024-09-04\_load-test.pdf
- 2. CI 上传云存储 (S3/OSS) → 返回公开 URL
- 3. Markdown 只留链接:

```
## 负载测试报告

[下载 PDF](https://bravo-reports.oss-cn-shanghai.aliyuncs.com/2024-09-04_load-test.pdf)
```

- 4. Git 只追踪 Markdown, 仓库体积不膨胀。

### 1.3.8.11 快速开始（一次性命令）

```
mkdir -p docs/{00_product,01_guideline,02_test_report,03_operate}
touch docs/00_product/PRD_v1.0.md \
      docs/01_guideline/onboarding.md \
      docs/02_test_report/2024-09-04_sprint1.md \
      docs/03_operate/runbook.md
git add docs/
git commit -m "docs: init doc skeleton per best practice"
```

→ 后续 CI 自动软链/发布，零人工维护 HTML 报告。

### 1.3.9 完善项目的代码函数、变量、代码文件，数据库表、字段等等的命名规则，小驼峰、大驼峰命名。

### 1.3.10 使用task master、figma、pytest-mcp-server、playwright等等MCP工具

### 1.3.11 设计项目框架，标定重要文件GUIDE、README、FAQ等等

### 1.3.12 使用Figma MCP工具设计前端界面

### 1.3.13 所使用库、依赖自动更新到requirements.txt、package.json，开发环境一键安装

- `pip install` 只会把包装进当前环境；  
想让 `requirements.txt` 同步，必须再执行 `pip freeze > requirements.txt`（或手工编辑、用 `pip-tools`）。
- 团队规范：一律用 `npm install 包名` / `npm uninstall 包名`，禁止手动改 `node_modules`。
- 提交代码前扫一眼 `git diff package.json package-lock.json`，确认依赖变更符合预期。

#### 1.3.13.12 依赖管理最佳实践

##### 1.3.13.12.2 添加新包的流程：

1. 开发环境添加包：

```
# 激活虚拟环境
source venv/Scripts/activate

# 安装新包
pip install new-package==1.0.0

# 更新 requirements/local.txt
pip freeze > requirements/current.txt
# 手动编辑 local.txt 添加新包

# 提交更改
git add backend/requirements/local.txt
git commit -m "feat: 添加 new-package 用于开发环境"
```

## 2. 测试环境添加包:

```
# 在测试环境中安装
pip install -r requirements/test.txt
pip install new-test-package==1.0.0

# 更新 requirements/test.txt
# 手动添加新包到 test.txt

# 提交更改
git add backend/requirements/test.txt
git commit -m "test: 添加 new-test-package 用于测试"
```

## 3. 生产环境添加包:

```
# 在生产环境中安装
pip install -r requirements/prod.txt
pip install new-prod-package==1.0.0

# 更新 requirements/prod.txt
# 手动添加新包到 prod.txt

# 提交更改
git add backend/requirements/prod.txt
git commit -m "prod: 添加 new-prod-package 用于生产环境"
```

### 1.3.13.12.3 ⚠️ 注意事项:

1. 版本锁定: 始终使用具体版本号, 避免 `>=` 或 `~=`
2. 环境隔离: 不同环境使用不同的 requirements 文件
3. 依赖冲突: 定期检查依赖冲突, 使用 `pip check`
4. 安全更新: 定期更新安全相关的包
5. 文档记录: 在提交信息中说明添加包的原因

### 1.3.14 Git commit 备注最佳实践

"一行摘要 + 空行 + 详述（可选） + 元数据（可选）"，用约定式提交（Conventional Commits）+ 50/72 换行规则，CI 能自动发版、Changelog 自动生成。

#### 1.3.14.13 模板结构

|                            |                               |
|----------------------------|-------------------------------|
| <type>(<scope>): <subject> | ← 50 字符内，中英文均可                |
| <body>                     | ← 72 字符自动换行，解释"为什么"而不是"怎么做"   |
| <footer>                   | ← 关联 Issue、PR、Breaking Change |

#### 1.3.14.14 类型说明（Angular 标准）

| 类型       | 说明                    |
|----------|-----------------------|
| feat     | 新功能（feature）          |
| fix      | 修复 Bug                |
| docs     | 仅文档变更                 |
| style    | 代码格式、缺少分号等无功能变化       |
| refactor | 代码重构，既不是 fix 也不是 feat |
| perf     | 性能优化                  |
| test     | 测试相关（单元、E2E、重构测试）     |
| build    | 构建系统或外部依赖变动           |
| ci       | CI/CD 脚本、配置变动         |
| chore    | 杂项（升级依赖、改 Makefile 等） |
| revert   | 回滚到上一个版本              |

#### 1.3.14.15 示例 1（后端）

|                                                                                                              |
|--------------------------------------------------------------------------------------------------------------|
| feat(users): 添加手机号验证码登录                                                                                      |
| <div><div>- 新增 SMSClient 封装阿里云短信</div><div>- 验证码 5 分钟内有效，3 次错误锁定 1 小时</div><div>- 支持测试环境 mock 开关</div></div> |
| Closes #123                                                                                                  |

### 1.3.14.16 示例 2 (前端)

`fix(english)`: 修复词汇卡片翻转动画卡顿

使用 `transform-gpu` 代替 `left/top`, 减少重排

**BREAKING CHANGE:** 移除 `--card-flip-left` 变量, 主题包需同步更新

See #456

### 1.3.14.17 示例 3 (杂项)

`chore(deps)`: 升级 Django 4.2 → 5.0

- 官方 LTS 支持至 2028
- 移除已弃用 `url()` 写法

Refs: <https://docs.djangoproject.com/en/5.0/releases/5.0/>

---

### 1.3.14.18 7 条黄金规则

1. 首字母小写、句末无句号
2. 摘要 ≤ 50 字符; Body ≤ 72 字符自动换行
3. 用现在时态: "添加"而非"添加了"
4. 首行写"做了什么", Body 写"为什么做"
5. 关联 Issue: `Closes #123` 或 `Fixes #456`
6. Breaking Change 必须以 **BREAKING CHANGE:** 开头
7. 提交前用 `git commit -v` 查看 diff, 确保无调试代码

---

### 1.3.14.19 自动化收益

- CI 识别 `feat/fix` 自动打 Tag 发版 (semantic-release)
- 自动生成 CHANGELOG.md
- IDE / Git GUI 首行截断显示, 50 字符内最美观

---

### 1.3.14.20 懒人工具

```
# 全局安装 commitizen → 交互式填写
npm i -g commitizen cz-conventional-changelog
echo '{ "path": "cz-conventional-changelog" }' > ~/.czrc
# 使用
git cz
```

### 1.3.15 使用task master生成任务时，产品文档存放以及任务文件存放分层，分目录，命名规则统一

## 1.4 测试体系建设

### 1.4.16 测试目录架构设计

- 1. 前端「就近原则」+后端「就近原则」
- 2. 统一收口到 tests/ 根目录做分层汇总 (CI 报告、性能、跨端 E2E)
- 3. 文件级清单：每个目录该放什么测试文件→直接照抄即可

```
Bravo/                                     # 项目根
├─ backend/
│  └─ apps/
│     └─ users/
│        ├── models.py
│        ├── views.py
│        └─ tests/                          # 后端: Django 就近单元测试
│           ├── __init__.py
│           ├── test_models.py
│           ├── test_views.py
│           └─ test_integration.py
│  └─ tests/                               # 后端: 顶层汇总 (可选)
│     ├── integration/                     # 跨 app 集成
│     │   └─ test_user_flow.py
│     ├── performance/                    # Locust/k6
│     │   └─ locustfile.py
│     └─ factories/                       # Factory Boy 共享
│        └─ user_factory.py
└─ pytest.ini

├─ frontend/
│  └─ src/
│     ├── components/
│     │   ├── BaseButton.vue
│     │   └─ BaseButton.spec.ts          # 前端: 就近单元
│     ├── views/
│     │   ├── Login.vue
│     │   ├── Login.spec.ts              # 前端: 就近集成
│     │   └─ Login.e2e.ts                # 前端: 就近 E2E
│     └─ utils/
│        ├── format.ts
│        └─ format.spec.ts
└─ tests/                                # 前端: 顶层汇总
   ├── unit/
   │   └─ setup.ts
   ├── e2e/
   │   ├── playwright.config.ts
   │   └─ flows/
   │      └─ auth.flow.e2e.ts
   └─ coverage/
      └─ vitest.config.ts
```



```
└─ tests/                                # 项目级汇总（前后端一起跑）
  └─ integration/                        # 跨端 API ↔ UI 集成
    └─ test_user_end_to_end.py
  └─ system/                            # 全链路冒烟
    └─ test_smoke.py
  └─ reports/                           # 统一覆盖率/Allure
    └─ .gitignore
    └─ README.md
```

## 后端 (Django 就近)

```
apps/<module>/tests/
└─ test_models.py      # Model 方法/信号
└─ test_views.py       # 视图/序列化器/权限
└─ test_integration.py # 同 app 多模型联调
```

## 后端 (顶层汇总)

```
backend/tests/
└─ integration/    # 跨 app 调用 (Celery、外部 API)
└─ performance/   # Locust/K6 脚本
└─ factories/     # Factory Boy 共享构造器
```

## 前端 (就近)

```
src/xxx/xxx.vue
src/xxx/xxx.spec.ts    # Vitest 组件/视图
src/xxx/xxx.e2e.ts     # Playwright 单页 E2E
```

## 前端 (顶层汇总)

```
frontend/tests/
└─ unit/setup.ts      # 全局 mock、插件
└─ e2e/playwright.config.ts
└─ coverage/vitest.config.ts
```

## 项目级汇总 (前后端一起跑 CI)

```
tests/
└─ integration/      # 调后端 API + 前端页面联合断言
└─ system/           # 冒烟 (健康检查、登录→下单→支付)
└─ reports/          # 统一 Allure/覆盖率入口
```

## CI 一键全跑 (GitHub Actions 片段)

yaml

```
- name: Backend tests
  run: |
    docker compose exec -T backend pytest backend/tests apps/*/tests --cov --cov-
report=xml:tests/reports/backend-cov.xml

- name: Frontend tests
  run: |
    cd frontend && npm run test:ci -- --coverage.reporter=xml --
coverage.reporter.dst=./tests/reports/frontend-cov.xml

- name: Merge coverage
  run: |
    python -m pip install coverage
    coverage combine tests/reports/*-cov.xml
    coverage xml -o tests/reports/full-coverage.xml
```

一句话总结

「前后端各自就近写测试，顶层 tests/ 只做汇总和跨端场景」，既保留前端/后端目录的「颜值」，又满足 CI 统一报告需求——直接抄！

**1.4.17 CI是否可以强制检验，文档是否有乱放，乱生成指定文档之外的文档，以及是否可以自由定制其他的检验规则。**

**1.4.18 先搭建基础框架，比如CICD、测试框架、日志打印设计。**

**1.4.19 make统一测试入口，测试文件自动识别并纳入**

**1.4.20 pre-commit钩子，push前自动跑全量测试**

```
# .pre-commit-config.yaml
repos:
  - repo: local
    hooks:
      - id: ai-test-gate
        name: AI Test Gate
        entry: cursor mcp pytest-gate --fail-under=80
        language: system
        pass_filenames: false
```

项目配置了多层自动校验机制：

#### 1.4.20.21 📄 校验原理和工作流程

##### 1. 本地预提交钩子 (Pre-commit Hooks)

- 触发时机：每次执行 git commit 时
- 配置文件：.pre-commit-config.yaml
- 工作原理：Git钩子机制，在提交前自动运行代码检查

包含的检查项目：

- Python代码：Black格式化、isort导入排序、flake8代码检查、mypy类型检查、bandit安全扫描
- 前端代码：Prettier格式化、ESLint代码检查
- 通用检查：去除尾随空格、文件结尾修复、YAML/JSON语法检查、大文件检测等
- Docker文件：hadolint检查
- 提交信息：commitizen规范检查

## 2. 远程CI/CD流水线 (GitHub Actions)

- 触发时机：推送到main/develop分支或创建Pull Request时
- 配置文件：.github/workflows/ci.yml.keep
- 工作原理：GitHub Actions自动运行完整的测试和质量检查

包含的检查阶段：

- 后端测试：单元测试、覆盖率检查、代码质量扫描
- 前端测试：单元测试、类型检查、代码规范检查
- E2E测试：端到端功能测试
- 安全扫描：Trivy漏洞扫描

## 1.4.20.22 🛠️ 如何设置和自定义门禁

### 1. 修改Pre-commit配置

编辑 .pre-commit-config.yaml 文件：

```
# 添加新的钩子
- repo: https://github.com/example/tool
  rev: v1.0.0
  hooks:
    - id: custom-check
      args: ["--custom-option"]
```

### 2. 自定义CI/CD流水线

修改 .github/workflows/ci.yml.keep 文件：

- 添加新的检查步骤
- 修改触发条件
- 调整测试覆盖率要求
- 配置部署条件

### 3. 本地质量检查脚本

- Makefile：提供统一的命令接口 ( make quality )
- scripts/quality.sh：详细的质量检查脚本

### 1.4.20.23 🛠️ 门禁自定义选项

严格程度调整：

- 代码覆盖率阈值：在pytest配置中设置最小覆盖率
- 代码复杂度限制：在flake8配置中调整复杂度阈值
- 安全扫描级别：在bandit配置中设置严重程度过滤 跳过特定检查：

```
# 跳过pre-commit检查（不推荐）
git commit --no-verify -m "紧急修复"

# 跳过特定钩子
SKIP=flake8 git commit -m "跳过flake8检查"
```

分支保护原则

在GitHub仓库设置中配置：

- 要求通过状态检查
- 要求代码审查
- 限制强制推送

### 1.4.20.24 🏠 质量门禁的价值

1. 代码质量保证：确保代码符合团队规范
2. 安全防护：及早发现安全漏洞
3. 自动化测试：防止功能回归
4. 团队协作：统一的代码风格和质量标准

---

**1.4.21 完善测试门禁设计，代码覆盖率、复杂度，等等，利用现有库或工具比如radon、coverage、pytest、pylint、playwright、selenium、Lighthouse、Codecov、GitHub Checks 等等，探索需要使用的测试库以及工具**

### 1.4.22 用 GitHub Actions + Vercel + Slack 建立“自动部署+回滚”

- 部署前必须所有测试通过
- 部署失败自动回滚
- 你只在 Slack 收到“成功/失败”通知

### 1.4.23 测试工厂配置，用数据代替代码、用生成代替手写、用配置代替复制粘贴

### 1.4.24 防止Cursor 为了“让测试通过”而偷偷改测试（或断言），而不是修业务逻辑

- 测试用例双仓库（Test-Monorepo）

```
repo/
├─ src/           # 业务代码（Cursor 可写）
├─ tests/         # 生成层：Cursor 可写
└─ tests-golden/  # 手写层：只读，Cursor 禁止写
```

- tests-golden/ 用 Git LFS 或单独私有仓库托管，强制只读  
在 CI 里加钩子：

```
# .github/workflows/guard-golden-tests.yml
- name: Fail if golden tests are touched
  run: |
    if git diff --name-only origin/main...HEAD | grep -q '^tests-golden/'; then
      echo "❌ 禁止修改黄金测试！"
      exit 1
    fi
```

- “黄金测试”由你亲自 Review 才能合并

所有改动 tests-golden/ 的 PR 必须人工 Approve

GitHub 设置规则：

Settings → Branches → Require pull request reviews before merging

并指定 Code Owner：

```
/tests-golden/    @your-username
```

- Cursor 的提示词里加“测试锁”

在每次生成代码前，把下面系统提示词喂给 Cursor：

#### 【测试安全锁】

1. 你绝对禁止修改 tests-golden/ 目录下的任何文件。
2. 如果测试失败，只能修改 src/ 里的业务逻辑或 tests/ 里的辅助代码。
3. 如果你认为测试用例本身有误，请输出“TEST\_CASE\_ISSUE: <描述>”并停止生成，等待人类确认。

- 测试快照（Snapshot）+ Diff 报警

对关键断言使用 Jest snapshot / Playwright screenshot snapshot

在 CI 里跑：

```
npx jest --ci --updateSnapshot
git diff --exit-code tests-golden/ # 有差异就失败
```

1.4.25 防止cursor可能只是“挑好过的测试跑一遍”，甚至“假装跑”，然后告诉你“全通过”。

- 强制全量测试 + 公开日志

```
# .github/workflows/gate.yml
- name: Run ALL tests with trace
  run: |
    npx jest --ci --coverage --reporters=default --reporters=jest-junit
    npx playwright test --reporter=html
    npx nyc report --reporter=text-lcov > coverage.lcov
```

把报告上传到 **GitHub Actions Artifacts** 与 **Codecov**，你点一下就能看到：

- ✔ 跑了多少用例
- ✔ 哪些文件没覆盖
- ✔ 失败截图 / trace

- 覆盖率阈值 + 强制锁

在 `jest.config.js` 里加硬性阈值：

```
module.exports = {
  coverageThreshold: {
    global: {
      branches: 90,
      functions: 90,
      lines: 90,
      statements: 90,
    },
  },
};
```

CI 里如果低于阈值，直接失败 → Cursor 无法“蒙混过关”。

- 测试执行“双钥匙”制度

**钥匙 A：**Cursor 只能把代码推到 `dev` 分支

**钥匙 B：**GitHub Actions 在 `dev` → `main` 的 PR 上强制跑一次完整测试

你在 PR 页面一眼就能看到 ✔ / ✖

没有你的 Approve，无法合并到 `main`（部署也触发不了）

- 可视化监控面板（你一眼就能看懂）

| 指标    | 工具            | 阈值    | 不达标会怎样  |
|-------|---------------|-------|---------|
| 单测通过率 | GitHub Checks | 100 % | PR 被标 ✖ |

| 指标     | 工具                     | 阈值     | 不达标会怎样                                   |
|--------|------------------------|--------|------------------------------------------|
| 端到端通过率 | Playwright HTML Report | 100 %  | PR 被标 <span style="color: red;">✖</span> |
| 代码覆盖率  | Codecov                | ≥ 90 % | PR 被标 <span style="color: red;">✖</span> |
| 性能评分   | Lighthouse CI          | ≥ 90 分 | PR 被标 <span style="color: red;">✖</span> |

• 1.4.25.25 功能清单 → 结构化 JSON（你写一次即可）

`features.json`（放在仓库根目录，只读）

```
[
  { "id": "ENG-001", "desc": "英语新闻列表页显示标题和摘要" },
  { "id": "ENG-002", "desc": "英语新闻详情页点击翻译按钮出现中文翻译" },
  { "id": "ENG-003", "desc": "打字练习页面统计正确率" },
  { "id": "BLOG-001", "desc": "博客列表页展示 3 篇最新博客" }
]
```

你只需维护这个文件，用自然语言描述功能点即可。

- 让 Cursor 自动把“测试 ↔ 功能”链接起来  
在 `jest.config.js` 里加：

```
module.exports = {
  testMatch: ["**/__tests__/**/*.test.js"],
  setupFilesAfterEnv: ["<rootDir>/matchFeatures.js"],
};
```

`matchFeatures.js`（自动生成，Cursor 每次写测试时都要按模板加一行）

```
// 在每个测试文件顶部
const { linkTestToFeature } = require('./testMap');
linkTestToFeature('ENG-001'); // 告诉系统这条测试覆盖 ENG-001
```

Cursor 的**系统提示词**里追加：

- 【功能-测试映射锁】
1. 写完测试后，必须在文件顶部调用 `linkTestToFeature('功能ID')`。
  2. 不允许把 `linkTestToFeature` 指向不存在的功能ID。
  3. CI 会检查每个功能ID是否至少被一条测试映射；缺了就失败。

• 1.4.25.26 CI 自动生成“功能-测试覆盖地图”

`.github/workflows/map.yml`

```
- name: Build Feature-Test Map
  run: |
    node scripts/buildFeatureMap.js          # 读取 features.json + 所有测试文件
    cat feature-test-map.md >> $GITHUB_STEP_SUMMARY # 在 PR 页面直接显示
```

生成的 `feature-test-map.md` (直接出现在 PR 底部, 你一眼可读)

| 功能ID     | 描述              | 测试文件                          | 状态    |
|----------|-----------------|-------------------------------|-------|
| ENG-001  | 英语新闻列表页显示标题和摘要  | <code>newsList.test.js</code> | ✅     |
| ENG-002  | 详情页点击翻译按钮出现中文翻译 | —                             | ❌ 无测试 |
| ENG-003  | 打字练习统计正确率       | <code>typing.test.js</code>   | ✅     |
| BLOG-001 | 博客列表展示 3 篇最新博客  | —                             | ❌ 无测试 |

**1.4.26 需求一改，对应代码没同步删，就会变成“僵尸代码”——功能入口没了，文件还在；测试还绿，实际没人调用。项目越来越肿，后续维护、构建、部署、AI 生成时间都会拖慢。**

| 步骤          | 机制                            | 工具/脚本                                          | 触发时机                       | 输出                                                |
|-------------|-------------------------------|------------------------------------------------|----------------------------|---------------------------------------------------|
| 1 静态引用扫描    | 找出“从未被 import / require 的文件”  | <code>npx unimported</code>                    | CI 每 push 跑一次              | 未引用文件列表                                           |
| 2 运行时覆盖率    | 跑 E2E + 单元测试, 收集“零命中”函数/行     | <code>nyc -- reporter=html + Playwright</code> | CI 跑 <code>gate.yml</code> | <code>coverage/index.html</code> 红色区域             |
| 3 自动化 PR 提醒 | 僵尸文件 > 3 天就自动提 PR 建议删除        | GitHub Action <code>dead-code-pr.yml</code>    | nightly cron               | PR 标题: <code>chore: remove dead code (bot)</code> |
| 4 人工一键合并    | 你只看 diff → Approve → Bot 自动合并 | GitHub <code>auto-merge</code> 规则              | 你点按钮                       | 仓库瘦身, 构建时间 ↓                                      |

- 示例 CI 片段 (直接复制)  
`.github/workflows/dead-code.yml`

```
name: Dead-Code-Scanner
on:
  schedule: [{ cron: '0 4 * * *' }] # 每天凌晨 4 点
jobs:
  scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
      - run: npm ci
      - run: npx unimported --json > unimported.json
      - run: node scripts/create-dead-code-pr.js # 读取 json 并提 PR
```



一句话总结

把“僵尸代码检测”做成夜间保安：每天自动巡逻，发现就贴条，等你一键清理。

✅ 今天就能做的 2 分钟动作

- 1. npm i -D unimported
- 2. 把上面的 dead-code.yml 扔进 .github/workflows
- 3. 明早看 GitHub → Pull requests 里是否出现“chore: remove dead code (bot)”

1.4.27 让前端、后端、E2E 三套测试各自独立，又能在 CI 里一键串起来。

```
repo-root/
├─ backend/
│  ├─ apps/                # Django 应用
│  └─ tests/               # 后端单元/集成测试
│     ├─ __init__.py
│     ├─ test_models.py
│     └─ test_views.py
├─ frontend/
│  ├─ src/
│  └─ tests/               # 前端单元/组件测试
│     ├─ unit/
│     └─ __init__.py
├─ e2e/                    # 端到端 (Playwright)
│  ├─ tests/
│  │  └─ blog.spec.js
│  └─ playwright.config.js
├─ .github/
│  └─ workflows/
│     ├─ backend.yml
│     ├─ frontend.yml
│     └─ e2e.yml
```

| 测试维度 | 代码路径                 | 命令                    | CI 文件        |
|------|----------------------|-----------------------|--------------|
| 后端单元 | backend/tests/       | python manage.py test | backend.yml  |
| 前端单元 | frontend/tests/unit/ | npm run test:unit     | frontend.yml |
| 端到端  | e2e/tests/           | playwright test       | e2e.yml      |

1.4.28 如何确保cursor开发的新功能不会影响到已实现功能

一句话：用「回归测试门禁」把已实现功能锁死——任何新改动跑不过老测试就自动拒收，Cursor 想破坏都难。

- 给已有功能建黄金测试（锁死）

```
# 示例: Django
echo "class BlogTests(TestCase):
    def test_create_post(self):
        r = self.client.post('/api/posts/', {'title':'hi'})
        self.assertEqual(r.status_code, 201)
" > backend/tests/test_regression.py
```

- CI 设置「回归门禁」  
    .github/workflows/regression.yml

```
name: Regression Gate
on: [pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: docker compose up -d
      - run: pytest backend/tests/test_regression.py # 只跑旧功能
```


若回归测试失败，PR 自动 ✖，禁止合并。

- Cursor 系统提示词（防手抖）

【回归锁】

1. 任何新功能必须追加到 tests/ 新文件，禁止修改 test\_regression.py
2. 先跑 `pytest test_regression.py` 100 % 通过，再提交代码
3. 如测试失败，仅修业务逻辑，不改测试

1.4.29 cursor可能有很多地方没有设置测试用例，项目已进行一半，如何查漏补缺

| 步骤               | 工具/命令                                                                                                                                     | 输出                              | 下一步                             |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|---------------------------------|
| ① 一键生成「覆盖率缺口地图」  | <code>pytest --cov=backend --cov-report=html</code>                                                                                       | <code>htmlcov/index.html</code> | 浏览器打开，红色区域就是 <b>0% 覆盖的文件/函数</b> |
| ② 自动生成「功能-测试对照表」 | <code>python scripts/build_feature_map.py</code>  脚本已给 | <code>missing_tests.md</code>   | 每行对应一个 <b>无测试的功能点</b>           |
| ③ 增量补测           | 让 Cursor 按 missing_tests.md 逐条补单测                                                                                                         | <code>test_*.py</code> 新增文件     | 再跑一次 ①，直到红色全灭                   |

一键脚本（直接复制到 `scripts/build_feature_map.py`）

```
#!/usr/bin/env python3
import ast, os, re, json
```

```

FEATURES_FILE = 'features.json'
COVERAGE_FILE = '.coverage'

def list_functions(path):
    funcs = set()
    for root, _, files in os.walk(path):
        for f in files:
            if f.endswith('.py') and 'test_' not in f:
                try:
                    tree = ast.parse(open(os.path.join(root, f)).read())
                    for node in ast.walk(tree):
                        if isinstance(node, ast.FunctionDef):
                            funcs.add(f"{f}:{node.name}")
                except: pass
    return funcs

def list_tests():
    tests = set()
    for root, _, files in os.walk('backend/tests'):
        for f in files:
            if f.startswith('test_'):
                txt = open(os.path.join(root, f)).read()
                tests.update(re.findall(r'def\s+(test_\w+)', txt))
    return tests

if __name__ == '__main__':
    funcs = list_functions('backend')
    tests = list_tests()
    missing = sorted(funcs - {t.replace('test_', '') for t in tests})
    with open('missing_tests.md', 'w') as f:
        f.write('| 文件:函数 | 原因 |\n|---|---|\n')
        for m in missing:
            f.write(f'| `{m}` | 无测试 |\n')
    print('🔗 missing_tests.md 已生成, 共', len(missing), '处缺口')

```

- 2 小时实操流程

### 1. 生成缺口

```

pytest --cov=backend --cov-report=html
python scripts/build_feature_map.py

```

### 2. 浏览器打开 `htmlcov/index.html` + `missing_tests.md`, 红色即目标。

### 3. Cursor 补测

根据 `missing_tests.md` 第 1 行, 为 `backend/views.py:create_post` 写 `pytest` 单测

逐行补完, 再跑覆盖率, 直到红色区域 < 5 %。

### 1.4.30 我又担心cursor并没有建立完善的测试体系，如何验证

用「一键体检脚本」+「随机破坏测试」双重验证，10 分钟就能确认 Cursor 到底有没有给你搭好完整的测试体系。

验证脚本（复制即用）

在项目根目录新建 `health_check.sh`：

```
#!/bin/bash
set -e
echo "===== 1  测试文件存在性 ====="
find backend -name "test_*.py" | wc -l          # >0 表示有测试文件
echo "===== 2  单测可运行 ====="
pytest backend/tests --collect-only -q | tail -1
echo "===== 3  覆盖率阈值 ====="
pytest --cov=backend --cov-report=term-missing --cov-fail-under=80
echo "===== 4  回归测试 ====="
pytest backend/tests/test_regression.py
echo "===== 5  E2E  可跑 ====="
npx playwright test --reporter=line
echo "===== 6  随机破坏（Mutation） ====="
pip install mutmut
mutmut run --paths-to-mutate=backend/apps || echo "Mutation 测试已跑完"
```

一键执行：

```
chmod +x health_check.sh
./health_check.sh
```

结果判定

| 检查项        | 通过标准            | 失败即说明           |
|------------|-----------------|-----------------|
| 1 测试文件数    | ≥1              | Cursor 根本没建测试文件 |
| 2 单测可运行    | 0 error         | 测试文件语法/依赖错误     |
| 3 覆盖率      | ≥80 %           | 大量代码无测试         |
| 4 回归测试     | 100 % pass      | 已有功能被破坏         |
| 5 E2E      | 0 error         | 前端/接口链路未覆盖      |
| 6 Mutation | kill-rate ≥60 % | 测试质量低（断言弱）      |

跑 `health_check.sh` 6 项全绿，就说明 Cursor 真的给你搭好了完整测试体系；任何一项红，就按对应缺口打回补测。

1.4.31 我如何能够将所有的测试文件入口到一个脚本，包括单元测试、回归测试、前端测试、性能测试、边缘测试、后端测试，playwright、selenium 等各种测试体系和框架，并且新增用例都要链接到这个脚本，每次强制他去执行这个脚本

业内标准做法：分层任务编排（Task Runner）

| 层级   | 工具                                    | 说明       | 示例                               |
|------|---------------------------------------|----------|----------------------------------|
| 统一入口 | justfile / Makefile / npm scripts     | 一条命令跑所有  | make test                        |
| 分层任务 | pytest + playwright + lighthouse + k6 | 各自独立，可并行 | make test-unit, make test-e2e    |
| 自动收录 | 通配符 + 目录约定                            | 新增文件零配置  | tests/**/test_*.py 自动被 pytest 收集 |

推荐方案：用 Makefile（跨平台，一行命令）

1 创建 Makefile

```
.PHONY: test test-unit test-e2e test-perf test-regression

# 一键跑所有
test: test-unit test-e2e test-perf test-regression

# 单元 (Django)
test-unit:
    pytest backend/tests -q --cov=backend --cov-fail-under=80

# E2E (Playwright)
test-e2e:
    npx playwright test --reporter=line

# 性能 (Lighthouse)
test-perf:
    npx lhci autorun

# 回归
test-regression:
    pytest backend/tests/test_regression.py

# 前端单元
test-frontend:
    npm run test:unit --prefix frontend
```

2 统一入口脚本（可选，CI/本地通用）

```
#!/bin/bash
# test_all.sh
set -e
echo " 🚀 开始全量测试..."
make test
echo " ✅ 全部通过"
```

### 3 新增用例自动收录

- **后端**: `backend/tests/**/test_*.py` (pytest 自动收集)
- **前端**: `frontend/tests/**/*.test.js` (Vitest 自动收集)
- **E2E**: `e2e/**/*.spec.js` (Playwright 自动收集)
- **性能**: `lighthousec.js` 通配符 `**/lhci/**/*.json`

✅ 一键验证 (新增用例自动进脚本)

bash

复制

```
# 本地
./test_all.sh
# CI
- uses: actions/checkout@v4
- run: ./test_all.sh
```

✅ 一句话总结

业内普遍用「Makefile / Taskfile」做统一入口，所有测试框架各自目录 + 通配符自动收录；新增文件零配置即可被跑。