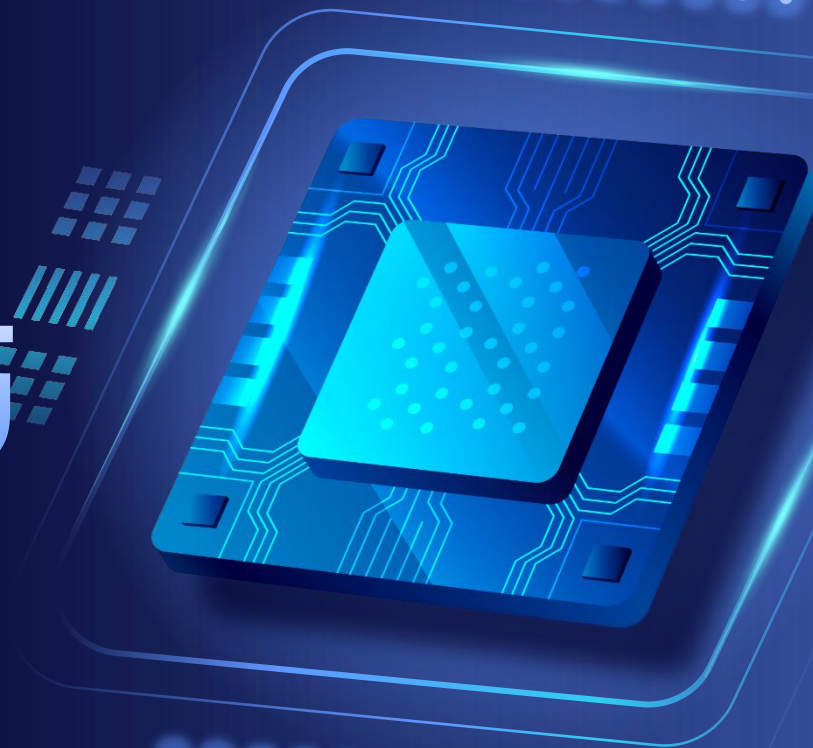


2025/10/22

Copilot的不同模式 及其使用方法和技巧

汇报人: cloud.ye (老叶)



CONTENTS

目录

01 | Copilot模式总览

02 | Agent模式

03 | Edit模式

04 | Ask模式

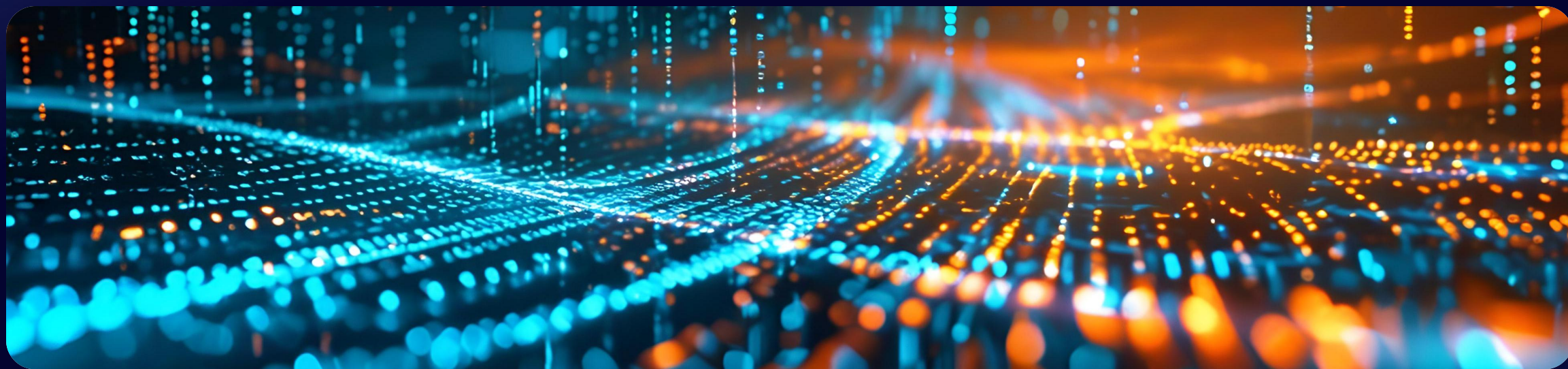
05 | 对比与选择建议



01

Copilot模式总览

模式典型用途和关键特点



Inline 模式（代码内联建议）

用途：快速补全、模板、参数建议

特点：最轻量、实时、上下文敏感

Chat（侧边栏聊天）

用途：项目级问答、设计讨论、跨文件重构

特点：对话式、可带指令、上下文宽

Ask（Ask Copilot）

用途：针对选区的解释、审查、重构、测试

特点：精准针对选区，适合关键代码

模式种类及触发方式



Agent (自动代理 / 工作流代理)

用途：自动化多步骤任务（如 CI 修改、批量重构、自动 PR 管理）

特点：能串联多个动作、可保存脚本式任务、长流程自动化



Edit (代码编辑建议 / Live Edit)

用途：在上下文中直接给出完整替换、补丁（Patch）或代码动作

特点：输出以“编辑”为单位（而非只给片段），便于一次性应用多处改动



Copilot Labs / 实验功能

用途：快速原型、提示工程实验

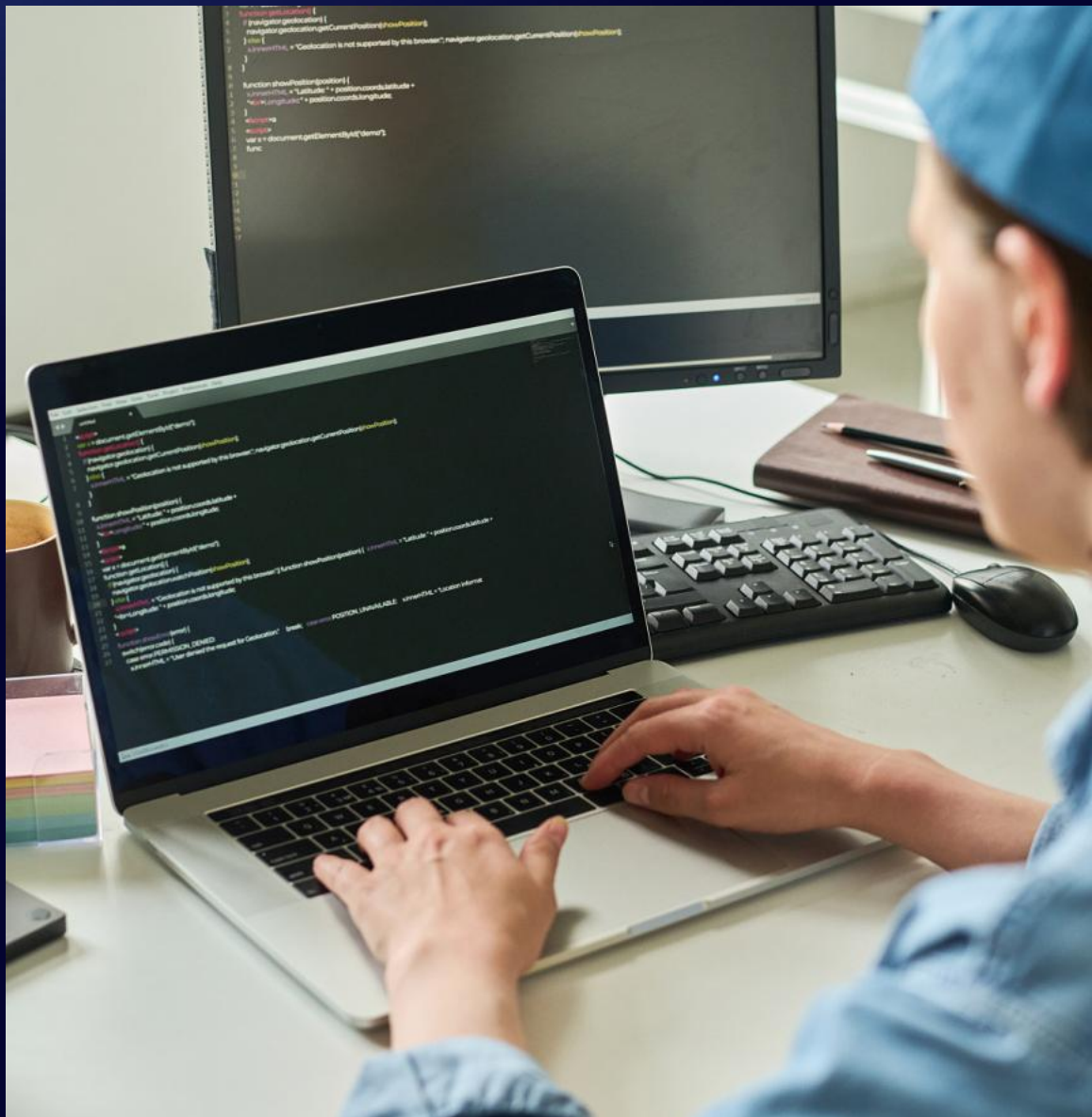
特点：实验性功能与自定义 prompt 模板



02

Agent模式

Agent模式作用和常见用例



自动化任务执行

Agent模式可自动执行重复性任务，如代码格式化、提交代码到版本控制系统。

智能代码建议

在编写代码时，Agent模式提供实时的代码建议，帮助开发者提高编码效率。

环境适应性

Agent模式能够根据不同的开发环境和项目需求，自动调整其行为和建议。

如何使用Agent模式

设置目标和约束

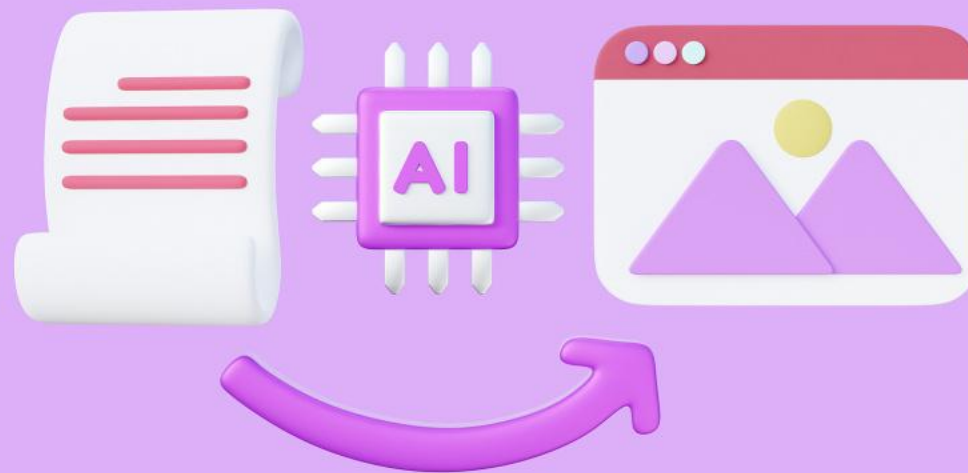
在Agent模式中，用户可以设定具体目标和约束条件，引导AI生成符合预期的代码或文本。

交互式反馈调整

通过与AI的实时交互，用户可以提供反馈，调整Agent的行为，以优化输出结果。

监控和评估性能

用户需要定期监控Agent模式的性能，评估其输出的准确性和效率，确保使用效果。



注意事项和风险提示



限制性指令的使用

在Agent模式下，避免使用可能引发不适当或危险行为的限制性指令。

监控与干预

用户应持续监控Copilot的输出，及时干预以纠正任何错误或不当内容。

隐私保护

确保在使用Agent模式时，不输入敏感或私人信息，以防止数据泄露风险。



Agent示例：依赖升级与兼容性检查 Agent



场景：自动扫描项目依赖（package.json / requirements.txt / pom.xml），升级安全/次要版本并检查兼容性。

任务流程：

- 扫描依赖文件
- 比对版本 → 查找安全更新（minor/patch）
- 更新依赖文件
- 运行测试（如 npm test / pytest）
- 生成变更报告
- 创建 PR 并附报告

提示词

Agent name: Dependency Upgrader

Goal: Keep dependencies secure and up to date

Steps:

- Scan dependency files for outdated packages
- For each, check latest compatible version
- Update version numbers (avoid breaking changes)
- Run project tests
- Summarize results and open PR with changelog

Constraints:

- Do not update to major versions
- Skip packages marked as "locked"

dependency-upgrader.yaml 配置文件

name: Dependency Upgrader

description: Keep dependencies secure and up to date across the repository

trigger: manual

permissions:

read: ["package.json", "requirements.txt", "pom.xml"]

write: ["package.json", "requirements.txt", "pom.xml"]

workflow:

- name: Scan outdated dependencies

run: |

copilot scan dependencies --outdated

- name: Suggest updates

run: |

copilot suggest "Update all minor and patch versions without breaking changes"

- name: Apply updates

run: |

copilot edit "Update versions to latest compatible versions"

- name: Run tests

run: |

npm test || pytest || mvn test

- name: Create PR

run: |

git checkout -b chore/dependency-upgrade

git add .

git commit -m "chore: upgrade dependencies (auto)"

gh pr create --fill

constraints:

- Do not update to major versions
- Skip packages marked as locked

Agent示例：文档同步与注释生成 Agent



场景：代码与文档不同步（函数无注释、README 过时），Agent 自动生成或更新注释。

任务流程：

- 扫描项目源代码
- 检测无注释或注释过时的函数
- 生成 docstring 或 README 段落
- 应用并提交变更

提示词

Agent name: DocSync Agent

Goal: Keep documentation and code synchronized

Steps:

- Identify functions/classes missing docstrings
- Generate structured docstrings (param, return, example)
- Update README.md with latest function references
- Commit all documentation updates

Constraints:

- Use markdown for documentation
- Do not change business logic

docsync.yaml

name: DocSync Agent

description: Keep documentation and code comments synchronized

trigger: manual

permissions:

read: ["src/**"]

write: ["src/**", "README.md", "docs/**"]

workflow:

- name: Identify undocumented functions

run: |

copilot scan "missing docstrings"

- name: Generate docstrings

run: |

copilot suggest "Add structured docstrings with param, return, and example sections"

- name: Apply documentation edits

run: |

copilot edit "Insert or update docstrings and README sections"

- name: Commit updates

run: |

git checkout -b docs/sync

git add .

git commit -m "docs: sync code comments and README (auto)"

gh pr create --fill

constraints:

- Do not alter business logic
- Use Markdown and standardized docstring formats

Agent示例：安全审查与修复 Agent



场景：扫描源代码查找潜在安全风险（输入验证、SQL注入、XSS、硬编码密钥等），自动提出修复方案。

任务流程：

- 扫描指定目录下源代码
- 检测安全漏洞模式
- 生成修复建议或直接 patch
- 生成安全报告

提示词

Agent name: Security Scan Agent

Goal: Identify and patch potential security vulnerabilities

Steps:

- Scan codebase for insecure patterns (eval, raw SQL, hardcoded secrets)
- Suggest safer alternatives (parameterized queries, env vars)
- Apply patch for low-risk cases
- Output detailed report with file locations and risk scores

Constraints:

- Never expose or log secrets
- Only modify files in 'src/' directory

```
security-scan.yml
name: Security Scan Agent
description: Identify and patch potential security vulnerabilities in source code
trigger: manual
permissions:
  read: ["src/**"]
  write: ["src/**"]
workflow:
  - name: Static scan
    run: |
      copilot scan security --patterns "eval, raw SQL, hardcoded secret"
  - name: Suggest fixes
    run: |
      copilot suggest "Replace insecure patterns with safe alternatives"
  - name: Apply low-risk patches
    run: |
      copilot edit "Auto-fix simple vulnerabilities"
  - name: Generate report
    run: |
      copilot report security --format markdown --output SECURITY_REPORT.md
  - name: Optional PR
    run: |
      git checkout -b fix/security-scan
      git add .
      git commit -m "fix: security patches (auto)"
      gh pr create --fill
constraints:
  - Never log or expose secrets
  - Modify only files under src/
```

Agent示例：自动测试修复 Agent



场景：测试用例挂了但错误较小（命名、路径、mock 数据），让 Agent 自动修复。

任务流程：

- 运行测试收集失败报告
- 识别错误类型（断言、路径、mock、import）
- 提出修复建议
- 应用可安全修复
- 再运行测试验证

提示词

Agent name: Test Fixer Agent

Goal: Automatically repair broken tests caused by minor code changes

Steps:

- Parse latest test results
- Identify non-breaking failures (e.g., changed function names)
- Propose patch and apply if low-risk
- Rerun tests to confirm fix
- Summarize changes

Constraints:

- Do not modify production code
- Only fix test-related logic

test-fixer.yml

```
name: Test Fixer Agent
description: Automatically repair broken tests caused by minor code changes
trigger: on_failure
permissions:
  read: ["tests/**", "src/**"]
  write: ["tests/**"]
workflow:
  - name: Analyze failed tests
    run: |
      copilot scan test-results --failed
  - name: Suggest safe fixes
    run: |
      copilot suggest "Fix minor test issues (assert mismatch, path, import)"
  - name: Apply edits
    run: |
      copilot edit "Repair tests for renamed functions or changed data"
  - name: Re-run tests
    run: |
      npm test || pytest
  - name: Summarize results
    run: |
      copilot log summary --tests
constraints:
  - Do not modify production code
  - Only fix test-related logic
```



使用方法

- Code WorkSpace目录下建 .github/copilot-agents目录
- 把yaml放到目录下面
- 打开chat面板，选中agent模式
- 在agent中输入提示词：Import from Repo



03

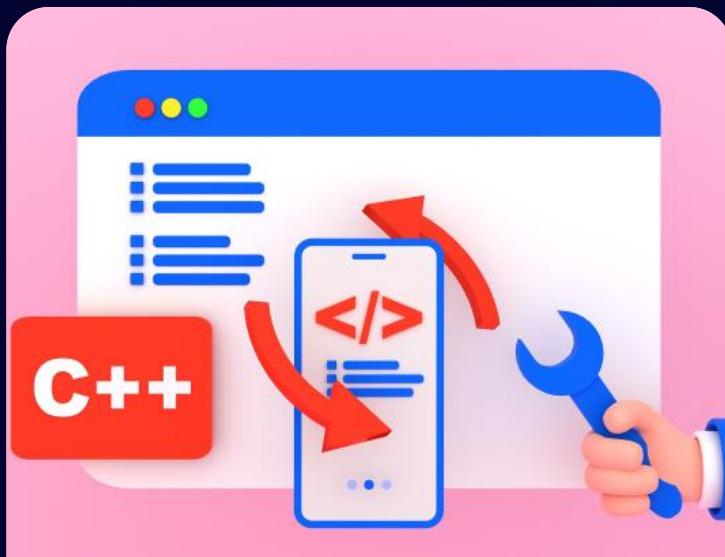
Edit模式

Edit模式作用和常见用例



代码审查与修正

在Edit模式下，开发者可以利用Copilot对现有代码进行审查和修正，提高代码质量。



功能扩展与完善

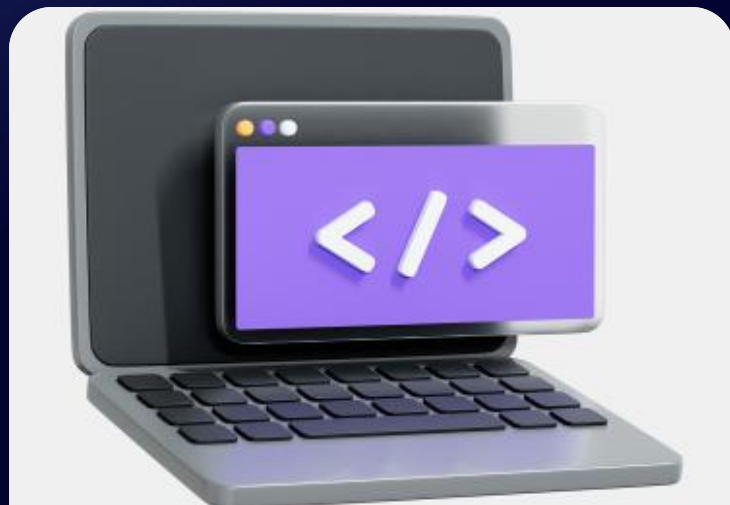
Copilot的Edit模式允许用户对现有功能进行扩展，添加缺失的代码段以完善程序。



自动化测试脚本编写

使用Edit模式，可以快速生成自动化测试脚本，加速软件测试流程，确保代码的稳定性。

使用方法和优势



直接代码编辑

在Edit模式下，用户可以直接对代码进行修改，提高开发效率。

即时反馈与修正

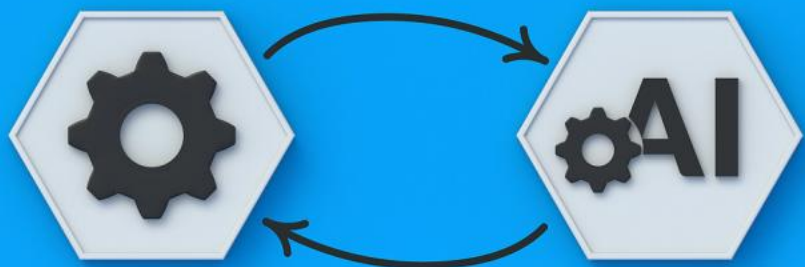
Edit模式允许开发者即时看到代码更改的效果，便于快速修正错误。



增强代码控制

该模式提供了对代码更细致的控制，帮助开发者精确调整功能实现。

Ask模式应用例子



理解复杂逻辑

选中关键函数 → Ask Copilot → “Explain this code step by step”

安全审查

选中一段与输入输出相关的代码 → Ask Copilot → “Check for potential security vulnerabilities”

性能优化建议

“Refactor this code to improve readability and maintainability”



04

Ask模式

Ask模式举例

```
// ApiKeyLogin API密钥登录
// godoc
// @Summary API密钥登录
// @Description API密钥登录
// @Tags 用户操作
// @Accept json
// @Produce json
// @Param request body requests.ApiKeyLoginReqReq true "API密钥登录请求"
// @Success 200 {object} responses.DefaultResponse "成功"
// @Failure 400 {object} responses.DefaultResponse "参数错误"
// @Failure 500 {object} responses.DefaultResponse "服务器错误"
// @Router /system/apiKeyLogin [post]

func ApiKeyLogin(c echo.Context, req requests.ApiKeyLoginReqReq, resp responses.DefaultResponse) error { 2
    userSrv := users.GetUsersHttpServiceInstance()
    result, err := userSrv.UserApiKeyLogin(req.UserName, req.ApiKey)
    if err != nil {
        return protocol.Response(c, constants.ErrInternalServerError.AppendErrors(err), data: "API密钥登录失败")
    }

    c.Response().Header().Set(key: "Access-Control-Expose-Headers", value: "id, jwt")
    c.Response().Header().Set(key: "id", fmt.Sprintf(format: "%d", result.UserID))
    c.Response().Header().Set(key: "jwt", result.Token)
    return protocol.Response(c, err: nil, result)
}

// UserSaveSelectLLMInfo 用户保存LLM信息
// godoc
// @Summary 用户保存LLM信息
```

This response was cancelled and may be incomplete

🔍 📄 🗑

Explain this code step by step

📄 users_function.go current file

> 3 steps completed successfully

简短说明：下面按执行顺序逐行解释该处理函数的行为与目的，并在代码块中用注释列出对应步骤（不重复原代码）。

逐步解释：

1. 从 users 包获取一个单例的用户服务实例（用于处理与用户相关的逻辑）。
2. 调用该服务的 UserApiKeyLogin 方法，传入请求中的 UserName 和 ApiKey，接收返回的登录结果与可能的错误。
3. 如果 UserApiKeyLogin 返回错误，使用统一的错误响应格式把错误返回给客户端（当前实现将内部错误附加到响应中）。
4. 在响应头中设置 Access-Control-Expose-Headers，以便前端能读取自定义头 id 与 jwt（跨域时允许访问这些头）。
5. 将登录结果中的 UserID 写入响应头 id，将生成的 token 写入响应头 jwt。
6. 最后把完整的 result 作为响应体返回给客户端（通常包含成功状态、token、用户信息等）。

安全/健壮性注意（简短提示）：

- 不要在响应头或响应体中泄露过多内部错误细节或敏感信息：应对外返回泛化错误消息，内部记录详细日志。
- 对输入做 trim/校验并使用常量时间比较 API Key，防止旁路匹配与时间侧信道。
- 暴露 JWT 在响应头需谨慎（CORS/浏览器存取与泄露风险）：考虑使用 HttpOnly cookie 或在响应体内且通过 HTTPS 传输并短时有效。

示意性注释（对应上述步骤；为避免直接重复源码，只用注释描述行为）：

```
// go
// 1. 获取用户服务实例（单例）
// 2. 调用 UserApiKeyLogin(userName, apiKey) -> result, err
// 3. if err != nil -> 返回统一的错误响应给客户端
// 4. 设置 Access-Control-Expose-Headers 以允许前端读取自定义头
// 5. 在响应头中写入 id（用户ID）和 jwt（token）
// 6. 返回 result 作为响应体
```

操作：选中代码，在chat侧边栏中使用：Explain this code step by step



05

对比与选择建议

研发工具场景适配指南

即时补全加速编码效率

Inline 功能专为快速补全设计，让代码输入更流畅，显著提升开发效率。



深入解析复杂代码逻辑

Ask 提供对关键代码的分步解读，帮助开发者全面理解复杂逻辑与潜在问题。



代码里做具体编辑

用于“在代码里做具体编辑”的场景——不是只给一段建议，Edit 模式更接近 IDE 的 code action（例如快速修复、重命名并同步引用、插入错误处理模板等）



自动化重构优化 workflow

Agent 能够执行多步骤自动化任务，如大规模重构或自动修复，确保一致性与可靠性。



自动化工具的核心价值与安全优先策略

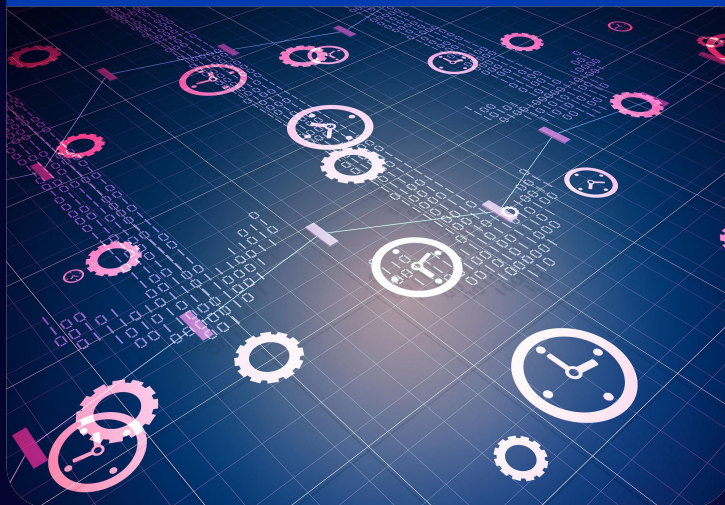


Agent：多步流程的高效执行者

Agent以自动化代理形式运作，擅长处理复杂、多步骤任务，支持保存与重放功能。尤其在批量改动场景中，其可重复性与一致性表现卓越。然而，正因其强大的执行能力，必须强化分支管理与预览机制，确保每一步变更均可追溯且安全可控。

Edit：精准修改的最佳实践

Edit通过“可应用编辑/patch”形式输出变更，适合一次性或批量的准确修改需求。相比传统单条建议，Edit更注重结果的精确性。但为避免潜在风险，应在应用前提供全面的安全检查与效果预览，从而降低错误操作的可能性。



安全与预览：执行前的双重保障

无论是Agent还是Edit，其核心优势在于“执行/应用”导向。因此，构建完善的安全机制、分支管理和实时预览功能至关重要。只有将这些防护措施前置，才能在提升效率的同时，最大限度地保障系统稳定性与数据完整性。

Agent 与 Edit 的安全实践优化



沙箱隔离与分支管理

默认将 Agent 的写操作限制在临时分支，确保主分支稳定性，避免直接修改引发的潜在风险。



变更规模的智能控制

设置文件数与变更数上限，防止大规模失控操作，保障系统安全性与可控性。



强制预览与透明操作

Edit 模式下强制显示 diff，杜绝“盲目应用”，提升变更的可见性与可验证性。



团队模板

把常用 Agent 流程（依赖更新、license 检查、测试修复）做成团队共享模板。



审计日志与团队协作优化

启用 Agent 操作日志并构建团队共享模板，便于回溯审批及标准化常用流程如依赖更新与测试修复。

06

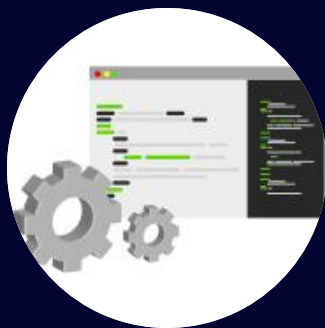
大语言模型给开发者带来的一些变化

LLM 提升开发效率



需求到编码提速

从需求澄清到初版编码，LLM 将文档阅读与框架设计效率提升数倍。Chat 模式快速生成概要和流程图，Inline 功能直接生成函数骨架，显著缩短开发前期耗时。



调试与重构优化

在错误修复与代码重构阶段，LLM 自动定位问题并解释原因，平均减少 40-60% 的调试时间。同时，多文件重构保证全局一致性，避免手动调整的潜在风险。



测试与文档同步

自动生成单元测试覆盖率达更高标准，降低测试成本；Agent 实时更新注释与文档，显著提升文档时效性，为团队协作提供可靠支持。

LLM助力代码质量提升——从“写正确”到“写得好”...



正确性、可读性保障

- 通过即时补全功能，LLM有效减少语法和逻辑错误
- Inline + Edit模式提供实时验证支持，显著提升代码开发的准确性与效率。



一致性和安全性优化

- Agent / Chat 模式整库修改，保证跨文件风格统一性
- 主动扫描潜在漏洞和风险，Ask “check for security issues” 帮助开发者快速定位问题，构建更安全的代码基础。



可维护性增强

自动生成测试用例和文档，DocSync Agent与Test Fixer Agent协同工作，大幅降低后续维护成本并提升代码可扩展性。

提升Prompt策略，优化LLM表现



明确目标导向

在与大型语言模型交互时，清晰的目标能够显著提升输出质量。例如，“Refactor for readability without changing behavior”不仅明确了任务方向，还限定了修改范围。始终以目标为核心，避免模糊表达。

上下文引用

在提升与优化LLM的Prompt策略时，可添加上下文引用以增强分析的精准性，例如：Based on the code logic within utils.js & index.js，从而明确跨文件间的依赖关系与交互逻辑。

设置边界条件

边界设定是防止模型“过度发挥”的关键。“Do not modify imports or global variables”这样的约束性指令能有效降低误改风险。通过合理划定边界，保护代码结构完整性，增强人机协作的信任感。

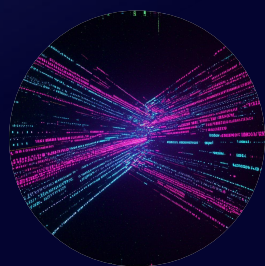
多阶段思维拆解

将复杂任务分步执行，如“First explain logic, then suggest optimization”，能让模型更稳定地完成工作。多阶段思维不仅能提高逻辑连贯性，还能帮助用户更好地追踪每一步结果。在处理跨文件或系统性任务时，这种方法非常适用。

积累项目模板和知识库

将一些常用的prompt精心保存为模板文件，以便随时调用，或是在开发过程中逐步积累并形成知识库，确保其内容得到持续更新与优化。

隐性收益：让开发者更专注“核心创造”



将低价值任务交给LLM

LLM能够高效处理生成与粘合层的重复性工作，从而让开发者专注于架构设计与业务规则的核心创新，显著提升工作效率。



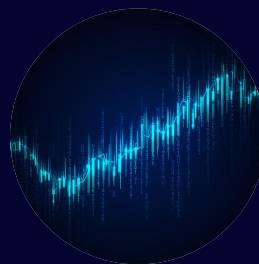
减少文档查询时间

通过直接向LLM提问，开发者可以快速获取所需信息，大幅缩短查阅文档的时间，将更多精力投入到高价值任务中。



加速知识迁移与自我进化

借助LLM，开发者从编写prompt到总结pattern的过程中实现知识内化，推动自身技能的持续进化与提升。



快速验证多种技术方案

利用LLM的高效能力，开发者可以在一分钟内完成多版本代码实现的比较，快速评估不同方案的优劣，优化决策效率。

个人化实践策略



固定节奏提效

先通过 Chat/Ask 规划任务，再 Inline 编写代码，经 Ask 优化后完成 Edit 应用，形成高效闭环，最大化提升开发效率与质量。

模板库化赋能

将常用 Prompt 存入 .copilot-labs/prompts/ 文件夹，支持团队共享复用，减少重复劳动，增强协作一致性与产出速度。
Tips: 启用文件过滤，避免 LLM 从编译文件或 node_modules 学无用知识

数据驱动改进

使用 Copilot Insights 或 VS Code Metrics 插件量化改进效果，关注代码编写时间、提交通过率及 PR 审查次数，持续优化 workflow。

安全与隐私相关：在分支上执行 Agent、敏感数据不输入 Prompt

模式组合



01

快速开发 Inline 和 Ask 的高效配合，实现响应快、精度高

02

维护遗留代码时 Ask 和 Chat 的强解释力优势，快速过流程，熟悉原有代码

03

CI/CD自动化流程中 Agent 和 Test Fixer 的无缝协作

04

代码审查时，使用Chat+Edit，得到高质量重构和审查报告



07

不同任务的模型选择策略

不同模型的性能与应用场景对比



模型名称	优势特征	最佳用途	速度/成本
GPT-4 Turbo / GPT-4o (OpenAI)	推理强、跨文件理解好	复杂逻辑、重构、架构分析	中速 / 高质
Claude 3.5 Sonnet (Anthropic)	长上下文、注释与文档生成强	大文件理解、文档同步	稍慢 / 高准确
Claude 4.5 Sonnet	Agent能力和跨文件生成能力成	Agent模式下，编写复杂程序	稍慢/高准确
Gemini 1.5 Pro (Google)	高上下文、代码补全敏捷	语言间迁移 (Java→Go)	快 / 中质
Mistral / Codestral (Mistral)	轻量快速、擅长短函数	快速原型、模板生成	极速 / 中低质
GitHub Copilot (OpenAI tuned)	与 IDE 深度整合	实时补全、Inline 生成	极快 / IDE 最优体验

使用经验

任务类型	推荐模式	推荐模型	说明
快速补全/模板生成	Inline	Stand GPT-4	最快，无需额外交互
优化特定逻辑	Ask	GPT-4o / Claude	选区精准分析，响应短
跨文件重构	Edit / Agent	GPT-4 Turbo/Claude	可读上下文 + 生成 diff
架构方案/性能分析	Chat	Claude / GPT-4o	适合长上下文讨论
语言迁移 / Prompt 试验	Labs	Gemini / GPT-4o	支持多语言与模板复用
自动生成测试 / CI	Agent + Test Fixer	GPT-4 Turbo	多文件协作与脚本化

模型选择：写用小模型，想用大模型；Inline 快、Ask 精、Edit 稳、Agent 智；

提示词技巧：

- 1、分步提示 + 上下文锚定 + 限制条件
- 2、对输出条件对比，保留效果最好的模板化



08

Copilot的一些巧配置技巧

.vscode .github .copilot 目录作用说明

目录	位置	典型文件	说明
.vscode	项目根	settings.json launch.json tasks.json extensions.json snippets/*	VS Code 本地配置 (IDE、调试、任务、扩展、片段)
.github	项目根	CODEOWNERS pre-commit-config.yam IISSUE_TEMPLATE/*.md PULL_REQUEST_TEMPLATE.md dependabot.yml workflows/*.yml	与 GitHub 相关的自动化/协作配置：工作流、代码归属、Issue/PR 模板、依赖更新等
.copilot	项目根	config.yaml/json .copilotignore suppress.txt enable.txt	GitHub Copilot 的本地配置与规则文件（支持忽略文件、强制禁用/启用、限制语言等）

文件	说明
.github/workflows/*.yml	任何 CI/CD (GitHub Actions) 脚本。可在push、pull_request、schedule等事件触发。例如ci.yml、deploy.yml。
.github/ISSUE_TEMPLATE/*.md	Issue 模板，支持多文件 (Bug、Feature、Help 等)。GitHub 会在新建 Issue 时提示选择对应模板。
.github/PULL_REQUEST_TEMPLATE.md	PR 模板，GitHub 在创建 PR 时会自动填充。
.github/CODEOWNERS	指定代码拥有者/审查者。可按路径/文件模式指定。
.github/dependabot.yml	Dependabot 配置 (自动依赖升级)。
.github/pre-commit-config.yaml	Pre-commit 钩子脚本配置 (可选，用pre-commit运行)。
.github/docs/...	任何额外的文档文件，GitHub 对其特殊渲染 (如docs/README.md)。
.github/SECURITY.md	安全政策。
.github/workflows/labeler.yml	labeler 自动给 PR 或 Issue 打标签。

文件名	作用	说明
.copilot/config.json	结构化配置偏好	定义风格、命名、测试框架等
.copilot/rules.md	Copilot 内置的检测器据此做智能过滤	Copilot 的 Review/Auto- Review 功能会在每次生成代码、提交 PR 或执行 “Copilot Review” 时读取这个文件。
.copilot/prompts	个人偏好 prompt 模板	每次 Chat 贴上即可模拟记忆
.copilot/instructions	指令模板目录	用于存储instructions 的md文件
.copilotignore	告诉 Copilot 在哪些文件/目录 不 做内容索引，也不生成建议。	单行一个 glob，支持 **、*、?，与 .gitignore 相同

提示词构造技巧：



明确任务目标

精准定义任务是成功的第一步，例如“优化代码性能”需具体化为“通过算法重构将时间复杂度从 $O(n^2)$ 降至 $O(n \log n)$ ”，确保指令清晰且可执行。

规范输出结构

输出格式的约束能显著提升结果质量，例如要求“仅展示最终代码改动”而非“解释每一步”，这使返回内容更契合实际需求。

设定限制条件

添加上下文限制如“遵循团队代码规范文件 `.copilot/rules.md`”或“限定在当前文件内操作”，有助于生成内容更加贴合实际开发环境与规则。

指令优先于对话

避免模糊表达，例如用“根据 StarDust 规则修复函数语法和风格问题”替代“你能帮我修复这个问题吗？”，明确指令更能激发 Copilot 的高效响应。

核心观点：避免模糊的自然语言陈述。例如：请帮我修复这个BUG，应该更改为：
请按照StarDust的Rules修复选定函数的语法和风格问题

提示词对话控场策略



优先级引导提升效率

将任务划分为明确阶段，如：@copilot
Stage 1: 按Stardust语法风格进行重构
Stage 2: 添加单元测试(仅在Stage1完成以后执行)。



单轮对话上下文优化

1) 显式缩小上下文范围
@copilot focus only on this class/function
或者@copilot ignore all other files; work only in current file.
原因：Copilot 默认会扫描 workspace 中的数十个文件，在多语言仓库下极易造成上下文爆炸。



分文件提示词举例

先在文件 A 执行：
@copilot analyze dependencies of this function
然后在文件 B 执行：
@copilot implement the interface expected by A
再执行：
@copilot run integration plan across A and B



上下文来源的精确配置

配置默认上下文为当前文件或指定相关文件，利用
`.copilot/instructions/base.yml` 优化输入来源，减少无关干扰并提升响应准确性。

防止死循环与崩溃的最佳实践

01 限制文件范围，避免指令含糊
Copilot重复生成相同修改常因workspace过大或指令不明确，通过限制文件范围并指定输出格式可显著提升效率。

02 改用Agent模式优化资源占用
CPU占用过高或冻结多源于多文件上下文循环调用，切换至Agent模式（支持任务切分）可有效缓解性能压力。

03 每轮对话前重载个人风格配置
输出风格错乱通常由上下文被旧对话覆盖引起，在每轮对话前执行`@copilot load personal-style`确保风格一致性。

04 明确终止条件防止无限循环
“Refactor again” 无限循环问题可通过在prompt中设置明确的停止条件解决，例如“Stop after producing final version”。

控制Prompt短语集

场景	指令示例
限制作用范围	Work only on this file / function / class.
禁止过度重构	Do not refactor unrelated code.
强制安全提交	Prepare a minimal diff, review before apply.
限定格式	Show output as unified diff, no prose.
多步骤任务	Perform step 1 only, wait for my next instruction.
性能警告	Avoid analyzing entire project.



09

Copilot的实战演示