

Claude Code

完整技术文档

By Anthropic

制作日期：2026-03-17

一个在终端中运行的 AI 编码工具，
理解您的代码库，帮助您更快地编写程序。

内容编译自 code.claude.com/docs/zh-CN
如需最新版本，请访问网站。

Generated by: github.com/bepsvpt/cc-docs

Table of Contents

Part 1: Getting Started	5
Claude Code 概述	5
快速开始	
开始使用桌面应用	19
设置 Claude Code	24
Claude Code 如何工作	33
Part 2: Core Usage	41
扩展 Claude Code	41
交互模式	51
Claude Code 最佳实践	65
常见工作流程	82
Part 3: Commands & Reference	
CLI 参考	104
自定义快捷键	113
Part 4: Configuration	124
配置权限	124
Claude 如何记住您的项目	132
Claude Code 设置	143
模型配置	184
输出样式	192
使用快速模式加快响应速度	195
有效管理成本	200
Part 5: Extensibility	207
Hooks 参考	207
使用 hooks 自动化工作流	268
通过 MCP 将 Claude Code 连接到工具	
使用 skills 扩展 Claude	325
创建插件	347
Plugins 参考	359
创建和分发 plugin marketplace	
通过市场发现和安装预构建插件	406
Part 6: Advanced & Automation	417
创建自定义 subagents	417

协调 Claude Code 会话团队	442
以编程方式运行 Claude Code	454
使用 Remote Control 从任何设备继续本地会话	
按计划运行提示词	462
代码审查	466
Part 7: CI/CD & Integrations	
Claude Code GitHub Actions	471
Claude Code GitLab CI/CD	490
Part 8: IDE & Platform Integration	504
在 VS Code 中使用 Claude Code	
JetBrains IDEs	519
使用 Claude Code Desktop	523
在 Chrome 中使用 Claude Code (测试版)	
Slack 中的 Claude Code	550
Claude Code on the web	556
Part 9: Security & Privacy	
安全性	574
Sandboxing	579
checkpointing	588
数据使用	590
法律和合规	595
零数据保留	596
Part 10: Enterprise & Monitoring	599
使用分析跟踪团队使用情况	599
监控	605
企业部署概述	621
配置服务器管理的设置 (公开测试版)	627
Part 11: Cloud Providers	632
Amazon Bedrock 上的 Claude Code	
Google Vertex AI 上的 Claude Code	
Claude Code on Microsoft Foundry	645
LLM gateway 配置	648
企业网络配置	652
Part 12: Environment Setup	655
开发容器	655
优化您的终端设置	658

自定义你的状态行

660

Part 13: Troubleshooting & Changelog

690

故障排除

690

Part 1: Getting Started

Claude Code 概述

Claude Code 是一个代理编码工具，可以读取你的代码库、编辑文件、运行命令，并与你的开发工具集成。可在终端、IDE、桌面应用和浏览器中使用。

Claude Code 是一个由 AI 驱动的编码助手，可以帮助你构建功能、修复错误和自动化开发任务。它理解你的整个代码库，可以跨多个文件和工具工作以完成任务。

开始使用

选择你的环境来开始。大多数界面需要 [Claude 订阅](#) 或 [Anthropic 控制台](#) 账户。终端 CLI 和 VS Code 也支持 [第三方提供商](#)。

Terminal

功能完整的 CLI，用于直接在终端中使用 Claude Code。编辑文件、运行命令，并从命令行管理整个项目。

To install Claude Code, use one of the following methods:

Native Install (Recommended)

macOS, Linux, WSL:

```
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell:

```
irm https://claude.ai/install.ps1 | iex
```

Windows CMD:

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del  
install.cmd
```

Windows requires [Git for Windows](#). Install it first if you don't have it.

Info:

Native installations automatically update in the background to keep you on the latest version.

Homebrew

```
brew install --cask claude-code
```

Info:

Homebrew installations do not auto-update. Run `brew upgrade claude-code` periodically to get the latest features and security fixes.

WinGet

```
winget install Anthropic.ClaudeCode
```

Info:

WinGet installations do not auto-update. Run `winget upgrade Anthropic.ClaudeCode` periodically to get the latest features and security fixes.

然后在任何项目中启动 Claude Code：

```
cd your-project  
claude
```

首次使用时，系统会提示你登录。就这么简单！[继续使用快速入门 →](#)

Tip:

查看[高级设置](#)了解安装选项、手动更新或卸载说明。如果遇到问题，请访问[故障排除](#)。

VS Code

VS Code 扩展在编辑器中直接提供内联差异、@-提及、计划审查和对话历史。

- [为 VS Code 安装](#)
- [为 Cursor 安装](#)

或在扩展视图中搜索” Claude Code”（Mac 上为 `Cmd+Shift+X`，Windows/Linux 上为 `Ctrl+Shift+X`）。安装后，打开命令面板（`Cmd+Shift+P` / `Ctrl+Shift+P`），输入” Claude Code”，然后选择在新标签页中打开。

[开始使用 VS Code →](#)

Desktop app

一个独立应用，用于在 IDE 或终端之外运行 Claude Code。直观地查看差异、并行运行多个会话、安排定期任务，并启动云会话。

下载并安装：

- [macOS](#)（Intel 和 Apple Silicon）
- [Windows](#)（x64）
- [Windows ARM64](#)（仅限远程会话）

安装后，启动 Claude，登录，然后点击**代码**标签开始编码。需要[付费订阅](#)。

[了解更多关于桌面应用的信息 →](#)

Web

在浏览器中运行 Claude Code，无需本地设置。启动长时间运行的任务并在完成时检查，处理你本地没有的仓库，或并行运行多个任务。可在桌面浏览器和 Claude iOS 应用中使用。

在 claude.ai/code 开始编码。

[开始使用网页版 →](#)

JetBrains

一个用于 IntelliJ IDEA、PyCharm、WebStorm 和其他 JetBrains IDE 的插件，具有交互式差异查看和选择上下文共享功能。

从 JetBrains Marketplace 安装 [Claude Code 插件](#)并重启你的 IDE。

[开始使用 JetBrains →](#)

你可以做什么

以下是你可以使用 Claude Code 的一些方式：

Claude Code 处理那些占用你一整天的繁琐任务：为未测试的代码编写测试、修复项目中的 lint 错误、解决合并冲突、更新依赖项和编写发布说明。

```
claude "write tests for the auth module, run them, and fix any failures"
```

用简单的语言描述你想要的内容。Claude Code 规划方法、跨多个文件编写代码，并验证其工作。

对于错误，粘贴错误消息或描述症状。Claude Code 通过你的代码库追踪问题、识别根本原因并实施修复。查看[常见工作流](#)了解更多示例。

Claude Code 直接与 git 配合作。它暂存更改、编写提交消息、创建分支并打开拉取请求。

```
claude "commit my changes with a descriptive message"
```

在 CI 中，你可以使用 [GitHub Actions](#) 或 [GitLab CI/CD](#) 自动化代码审查和问题分类。

[Model Context Protocol \(MCP\)](#) 是一个用于将 AI 工具连接到外部数据源的开放标准。使用 MCP，Claude Code 可以读取 Google Drive 中的设计文档、更新 Jira 中的工单、从 Slack 拉取数据，或使用你自己的自定义工具。

`CLAUDE.md` 是一个你添加到项目根目录的 markdown 文件，Claude Code 在每个会话开始时都会读取它。使用它来设置编码标准、架构决策、首选库和审查清单。Claude 还会在工作时构建[自动内存](#)，保存学习内容，如构建命令和调试见解，跨会话保存，无需你编写任何内容。

创建[自定义命令](#)来打包你的团队可以共享的可重复工作流，如 `/review-pr` 或 `/deploy-staging`。

[Hooks](#) 让你在 Claude Code 操作之前或之后运行 shell 命令，如在每次文件编辑后自动格式化或在提交前运行 lint。

生成多个 [Claude Code 代理](#)，同时处理任务的不同部分。一个主导代理协调工作、分配子任务并合并结果。

对于完全自定义的工作流，[Agent SDK](#) 让你构建由 Claude Code 的工具和功能驱动的自己的代理，完全控制编排、工具访问和权限。

Claude Code 是可组合的，遵循 Unix 哲学。将日志管道传入其中、在 CI 中运行它，或将其与其他工具链接：


```
## Monitor logs and get alerted
tail -f app.log | claude -p "Slack me if you see any anomalies"

## Automate translations in CI
claude -p "translate new strings into French and raise a PR for review"

## Bulk operations across files
git diff main --name-only | claude -p "review these changed files for security issues"
```

查看 [CLI 参考](#) 了解完整的命令和标志集。

会话不受限于单一界面。当你的上下文改变时，在环境之间移动工作：

- 离开你的办公桌，使用[远程控制](#)从你的手机或任何浏览器继续工作
- 在[网页](#)或 [iOS 应用](#)上启动长时间运行的任务，然后使用 `/teleport` 将其拉入你的终端
- 使用 `/desktop` 将终端会话交给[桌面应用](#)进行视觉差异审查
- 从团队聊天路由任务：在 [Slack](#) 中提及 `@Claude` 并附上错误报告，获得拉取请求

在任何地方使用 Claude Code

每个界面都连接到相同的底层 Claude Code 引擎，因此你的 CLAUDE.md 文件、设置和 MCP 服务器可在所有界面中工作。

除了上面的[终端](#)、[VS Code](#)、[JetBrains](#)、[桌面](#)和[网页](#)环境外，Claude Code 还与 CI/CD、聊天和浏览器工作流集成：

我想要…	最佳选项
从我的手机或另一台设备继续本地会话	远程控制
在本地启动任务，在移动设备上继续	网页 或 Claude iOS 应用
自动化 PR 审查和问题分类	GitHub Actions 或 GitLab CI/CD

我想要…	最佳选项
在每个 PR 上获得自动代码审查	GitHub Code Review
将 Slack 中的错误报告路由到拉取请求	Slack
调试实时网络应用	Chrome
为你自己的工作流构建自定义代理	Agent SDK

后续步骤

安装 Claude Code 后，这些指南可以帮助你深入了解。

- [快速入门](#)：通过你的第一个真实任务，从探索代码库到提交修复
- [存储说明和内存](#)：使用 CLAUDE.md 文件和自动内存为 Claude 提供持久说明
- [常见工作流](#)和[最佳实践](#)：充分利用 Claude Code 的模式
- [设置](#)：为你的工作流自定义 Claude Code
- [故障排除](#)：常见问题的解决方案
- [code.claude.com](#)：演示、定价和产品详情

快速开始

欢迎使用 Claude Code !

本快速开始指南将在几分钟内让您使用 AI 驱动的编码辅助。完成本指南后，您将了解如何使用 Claude Code 完成常见的开发任务。

开始前

确保您拥有：

- 打开的终端或命令提示符
 - 如果您之前从未使用过终端，请查看[终端指南](#)
- 一个要处理的代码项目
- 一个 [Claude 订阅](#)（Pro、Max、Teams 或 Enterprise）、[Claude Console](#) 账户，或通过[支持的云提供商](#)的访问权限

Note:

本指南涵盖终端 CLI。Claude Code 也可在[网页](#)、[桌面应用](#)、[VS Code](#) 和 [JetBrains IDE](#)、[Slack](#) 中使用，以及通过 [GitHub Actions](#) 和 [GitLab](#) 进行 CI/CD。请参阅[所有界面](#)。

步骤 1：安装 Claude Code

To install Claude Code, use one of the following methods:

Native Install (Recommended)

macOS, Linux, WSL:

```
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell:

```
irm https://claude.ai/install.ps1 | iex
```

Windows CMD:

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del  
install.cmd
```

Windows requires [Git for Windows](#). Install it first if you don't have it.

Info:

Native installations automatically update in the background to keep you on the latest version.

Homebrew

```
brew install --cask claude-code
```

Info:

Homebrew installations do not auto-update. Run `brew upgrade claude-code` periodically to get the latest features and security fixes.

WinGet

```
winget install Anthropic.ClaudeCode
```

Info:

WinGet installations do not auto-update. Run `winget upgrade Anthropic.ClaudeCode` periodically to get the latest features and security fixes.

步骤 2：登录您的账户

Claude Code 需要账户才能使用。当您使用 `claude` 命令启动交互式会话时，您需要登录：

```
claude  
## 首次使用时系统会提示您登录
```

```
/login
```

```
## 按照提示使用您的账户登录
```

您可以使用以下任何账户类型登录：

- [Claude Pro](#)、[Max](#)、[Teams](#) 或 [Enterprise](#)（推荐）
- [Claude Console](#)（具有预付费额度的 API 访问）。首次登录时，会在 Console 中自动创建一个“Claude Code”工作区，用于集中成本跟踪。
- [Amazon Bedrock](#)、[Google Vertex AI](#) 或 [Microsoft Foundry](#)（企业云提供商）

登录后，您的凭证将被存储，您无需再次登录。要稍后切换账户，请使用 `/login` 命令。

步骤 3：启动您的第一个会话

在任何项目目录中打开您的终端并启动 Claude Code：

```
cd /path/to/your/project
claude
```

您将看到 Claude Code 欢迎屏幕，其中包含您的会话信息、最近的对话和最新更新。输入 `/help` 查看可用命令，或输入 `/resume` 继续之前的对话。

Tip:

登录后（步骤 2），您的凭证将存储在您的系统上。在[凭证管理](#)中了解更多信息。

步骤 4：提出您的第一个问题

让我们从了解您的代码库开始。尝试以下命令之一：

```
这个项目做什么？
```

Claude 将分析您的文件并提供摘要。您也可以提出更具体的问题：

```
这个项目使用什么技术？
```

主入口点在哪里？

解释文件夹结构

您也可以询问 Claude 关于其自身功能的问题：

Claude Code 能做什么？

我如何在 Claude Code 中创建自定义 skills？

Claude Code 能与 Docker 一起工作吗？

Note:

Claude Code 根据需要读取您的项目文件。您无需手动添加上下文。

步骤 5：进行您的第一次代码更改

现在让我们让 Claude Code 进行一些实际的编码。尝试一个简单的任务：

在主文件中添加一个 hello world 函数

Claude Code 将：

1. 找到适当的文件
2. 向您显示建议的更改
3. 请求您的批准
4. 进行编辑

Note:

Claude Code 在修改文件前始终请求许可。您可以批准单个更改或为会话启用“全部接受”模式。

步骤 6：在 Claude Code 中使用 Git

Claude Code 使 Git 操作变得对话式：

我更改了哪些文件？

用描述性消息提交我的更改

您也可以提示更复杂的 Git 操作：

创建一个名为 feature/quickstart 的新分支

显示我最后的 5 次提交

帮我解决合并冲突

步骤 7：修复错误或添加功能

Claude 擅长调试和功能实现。

用自然语言描述您想要的内容：

向用户注册表单添加输入验证

或修复现有问题：

有一个错误，用户可以提交空表单 - 修复它

Claude Code 将：

- 定位相关代码
- 理解上下文
- 实现解决方案
- 如果可用，运行测试

步骤 8：尝试其他常见工作流

有多种方式可以与 Claude 一起工作：

重构代码

重构身份验证模块以使用 `async/await` 而不是回调

编写测试

为计算器函数编写单元测试

更新文档

使用安装说明更新 README

代码审查

审查我的更改并建议改进

Tip:

像与有帮助的同事交谈一样与 Claude 交谈。描述您想要实现的目标，它将帮助您实现。

基本命令

以下是日常使用中最重要的命令：

命令	功能	示例
<code>claude</code>	启动交互模式	<code>claude</code>
<code>claude "task"</code>	运行一次性任务	<code>claude "fix the build error"</code>
<code>claude -p "query"</code>	运行一次性查询，然后退出	<code>claude -p "explain this function"</code>
<code>claude -c</code>	在当前目录中继续最近的对话	<code>claude -c</code>

命令	功能	示例
<code>claude -r</code>	恢复之前的对话	<code>claude -r</code>
<code>claude commit</code>	创建 Git 提交	<code>claude commit</code>
<code>/clear</code>	清除对话历史	<code>/clear</code>
<code>/help</code>	显示可用命令	<code>/help</code>
<code>exit</code> 或 Ctrl+C	退出 Claude Code	<code>exit</code>

有关完整的命令列表，请参阅 [CLI 参考](#)。

初学者专业提示

有关更多信息，请参阅[最佳实践](#)和[常见工作流](#)。

不要说：‘修复错误’

尝试：‘修复登录错误，用户输入错误凭证后看到空白屏幕’

将复杂任务分解为步骤：

1. 为用户配置文件创建新的数据库表
2. 创建 API 端点以获取和更新用户配置文件
3. 构建允许用户查看和编辑其信息的网页

在进行更改之前，让 Claude 了解您的代码：

分析数据库架构

构建一个仪表板，显示英国客户最常退回的产品

- 按 `?` 查看所有可用的快捷键
- 使用 Tab 进行命令补全
- 按 `↑` 查看命令历史
- 输入 `/` 查看所有命令和 skills

接下来呢？

现在您已经学习了基础知识，探索更多高级功能：

- [Claude Code 如何工作](#) 了解代理循环、内置工具以及 Claude Code 如何与您的项目交互
- [最佳实践](#) 通过有效的提示和项目设置获得更好的结果
- [常见工作流](#) 常见任务的分步指南
- [扩展 Claude Code](#) 使用 CLAUDE.md、skills、hooks、MCP 等进行自定义

获取帮助

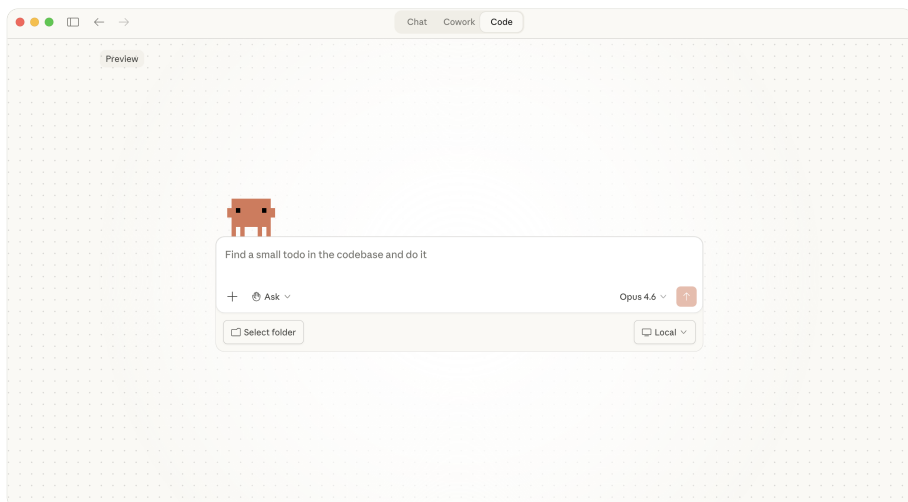
- **在 Claude Code 中：** 输入 `/help` 或询问「我如何…」
- **文档：** 您在这里！浏览其他指南
- **社区：** 加入我们的 [Discord](#) 获取提示和支持

开始使用桌面应用

在桌面上安装 Claude Code 并开始您的第一个编码会话

桌面应用为您提供具有图形界面的 Claude Code：可视化 diff 审查、实时应用预览、GitHub PR 监控和自动合并、使用 Git worktree 隔离的并行会话、计划任务以及远程运行任务的能力。无需终端。

本页面将指导您安装应用并开始您的第一个会话。如果您已经设置完成，请参阅[使用 Claude Code Desktop](#)了解完整参考。



Claude Code Desktop 界面显示已选择 Code 选项卡，包含提示框、权限模式选择器设置为询问权限、模型选择器、文件夹选择器和本地环境选项

桌面应用有三个选项卡：

- **Chat**：无文件访问权限的常规对话，类似于 `claude.ai`。
- **Cowork**：一个自主后台代理，在云虚拟机中处理任务，拥有自己的环境。它可以独立运行，而您可以进行其他工作。
- **Code**：一个交互式编码助手，可直接访问您的本地文件。您可以实时审查和批准每项更改。

Chat 和 Cowork 在 [Claude Desktop 支持文章](#)中有介绍。本页面重点关注 **Code** 选项卡。

Note:

Claude Code 需要 [Pro](#)、[Max](#)、[Teams](#) 或 [Enterprise](#) 订阅。

安装

Step 1: 下载应用

为您的平台下载 Claude。

- [macOS](#) 适用于 Intel 和 Apple Silicon 的通用版本
- [Windows](#) 适用于 x64 处理器

对于 Windows ARM64，[在此下载](#)。

目前不支持 Linux。

Step 2: 登录

从您的应用程序文件夹 (macOS) 或开始菜单 (Windows) 启动 Claude。使用您的 Anthropic 账户登录。

Step 3: 打开 Code 选项卡

点击顶部中心的 **Code** 选项卡。如果点击 Code 提示您升级，您需要先[订阅付费计划](#)。如果提示您在线登录，请完成登录并重启应用。如果您看到 403 错误，请参阅[身份验证故障排除](#)。

桌面应用包含 Claude Code。您无需单独安装 Node.js 或 CLI。要从终端使用 `claude`，请单独安装 CLI。请参阅[开始使用 CLI](#)。

开始您的第一个会话

打开 Code 选项卡后，选择一个项目并告诉 Claude 要做什么。

Step 1: 选择环境和文件夹

选择 **Local** 以在您的机器上运行 Claude，直接使用您的文件。点击 **Select folder** 并选择您的项目目录。

Tip:

从一个您熟悉的小项目开始。这是查看 Claude Code 能做什么的最快方式。在 Windows 上，必须安装 [Git](#) 才能使本地会话正常工作。大多数 Mac 默认包含 Git。

您也可以选择：

- **Remote：**在 Anthropic 的云基础设施上运行会话，即使关闭应用也会继续。远程会话使用与 [Claude Code on the web](#) 相同的基础设施。
- **SSH：**通过 SSH 连接到远程机器（您自己的服务器、云虚拟机或开发容器）。必须在远程机器上安装 Claude Code。

Step 2: 选择模型

从发送按钮旁的下拉菜单中选择模型。请参阅[模型](#)了解 Opus、Sonnet 和 Haiku 的比较。会话开始后无法更改模型。

Step 3: 告诉 Claude 要做什么

输入您想让 Claude 做的事情：

- Find a TODO comment and fix it
- Add tests for the main function
- Create a CLAUDE.md with instructions for this codebase

一个[会话](#)是与 Claude 关于您的代码的对话。每个会话跟踪自己的上下文和更改，因此您可以处理多个任务而不会相互干扰。

Step 4: 审查并接受更改

默认情况下，Code 选项卡以[询问权限模式](#)启动，其中 Claude 提议更改并等待您的批准后再应用。您将看到：

1. 一个[差异视图](#)，显示每个文件中将发生的确切更改
2. 接受/拒绝按钮以批准或拒绝每项更改
3. Claude 处理您的请求时的实时更新

如果您拒绝更改，Claude 将询问您希望如何以不同的方式进行。在您接受之前，您的文件不会被修改。

接下来呢？

您已经进行了第一次编辑。有关 Desktop 可以做的所有事情的完整参考，请参阅[使用 Claude Code Desktop](#)。以下是一些接下来可以尝试的事情。

中断并引导。您可以随时中断 Claude。如果它走错了方向，点击停止按钮或输入您的更正并按 **Enter**。Claude 停止正在做的事情并根据您的输入进行调整。您无需等待它完成或重新开始。

为 Claude 提供更多上下文。 在提示框中输入 `@filename` 以将特定文件拉入对话，使用附件按钮附加图像和 PDF，或直接将文件拖放到提示中。Claude 拥有的上下文越多，结果越好。请参阅[添加文件和上下文](#)。

使用 skills 处理可重复的任务。 输入 `/` 或点击 `+ → Slash commands` 以浏览[内置命令](#)、[自定义 skills](#) 和插件 skills。Skills 是可重用的提示，您可以在需要时调用，例如代码审查清单或部署步骤。

在提交前审查更改。 Claude 编辑文件后，会出现 `+12 -1` 指示器。点击它以打开[差异视图](#)，逐个文件审查修改，并对特定行进行评论。Claude 会读取您的评论并进行修订。点击 **Review code** 让 Claude 自己评估差异并留下内联建议。

调整您拥有的控制量。 您的[权限模式](#)控制平衡。询问权限（默认）在每次编辑前需要批准。自动接受编辑会自动接受文件编辑以加快迭代。Plan mode 让 Claude 在不触及任何文件的情况下规划方法，这在大型重构前很有用。

添加插件以获得更多功能。 点击提示框旁的 `+` 按钮并选择 **Plugins** 以浏览和安装[插件](#)，这些插件添加 skills、代理、MCP servers 等。

预览您的应用。 点击 **Preview** 下拉菜单以直接在桌面中运行您的开发服务器。Claude 可以查看正在运行的应用、测试端点、检查日志并迭代它看到的内容。请参阅[预览您的应用](#)。

跟踪您的拉取请求。 打开 PR 后，Claude Code 监控 CI 检查结果，可以自动修复失败或在所有检查通过后合并 PR。请参阅[监控拉取请求状态](#)。

将 Claude 放在日程上。 设置[计划任务](#)以定期自动运行 Claude：每天早上进行代码审查、每周进行依赖项审计，或从您连接的工具中提取信息的简报。

准备好时扩展。 从侧边栏打开[并行会话](#)以同时处理多个任务，每个任务都在自己的 Git worktree 中。将[长期运行的工作发送到云](#)，以便即使关闭应用也会继续，或者如果任务花费的时间比预期长，[在网络或 IDE 中继续会话](#)。[连接外部工具](#)，如 GitHub、Slack 和 Linear，以整合您的工作流。

来自 CLI？

Desktop 运行与 CLI 相同的引擎，但具有图形界面。您可以在同一项目上同时运行两者，它们共享配置（CLAUDE.md 文件、MCP servers、hooks、skills 和设置）。有关功能、标志等效项和 Desktop 中不可用内容的完整比较，请参阅[CLI 比较](#)。

接下来是什么

- [使用 Claude Code Desktop](#)：权限模式、并行会话、差异视图、连接器和企业配置
- [故障排除](#)：常见错误和设置问题的解决方案

- [最佳实践](#)：编写有效提示和充分利用 Claude Code 的提示
- [常见工作流](#)：调试、重构、测试等教程

设置 Claude Code

在您的开发机器上安装、认证和开始使用 Claude Code。

系统要求

- **操作系统：**
 - macOS 13.0+
 - Windows 10 1809+ 或 Windows Server 2019+ ([查看设置说明](#))
 - Ubuntu 20.04+
 - Debian 10+
 - Alpine Linux 3.19+ ([需要额外依赖](#))
- **硬件：**4 GB+ RAM
- **网络：**需要互联网连接 (查看[网络配置](#))
- **Shell：**在 Bash 或 Zsh 中效果最佳
- **位置：**[Anthropic 支持的国家](#)

额外依赖

- **ripgrep：**通常包含在 Claude Code 中。如果搜索失败，请查看[搜索故障排除](#)。
- **Node.js 18+：**仅在[已弃用的 npm 安装](#)中需要

安装

To install Claude Code, use one of the following methods:

Native Install (Recommended)

macOS, Linux, WSL:

```
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell:

```
irm https://claude.ai/install.ps1 | iex
```


Windows CMD:

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del  
install.cmd
```

Windows requires [Git for Windows](#). Install it first if you don't have it.

Info:

Native installations automatically update in the background to keep you on the latest version.

Homebrew

```
brew install --cask claude-code
```

Info:

Homebrew installations do not auto-update. Run `brew upgrade claude-code` periodically to get the latest features and security fixes.

WinGet

```
winget install Anthropic.ClaudeCode
```

Info:

WinGet installations do not auto-update. Run `winget upgrade Anthropic.ClaudeCode` periodically to get the latest features and security fixes.

安装过程完成后，导航到您的项目并启动 Claude Code：

```
cd your-awesome-project  
claude
```

如果您在安装过程中遇到任何问题，请查阅[故障排除指南](#)。

Tip:

安装后运行 `cclaude doctor` 以检查您的安装类型和版本。

平台特定设置

Windows: 本地运行 Claude Code（需要 [Git Bash](#)）或在 WSL 中运行。支持 WSL 1 和 WSL 2，但 WSL 1 支持有限，不支持 Bash 工具沙箱等功能。

Alpine Linux 和其他基于 musl/uClibc 的发行版：

Alpine 和其他基于 musl/uClibc 的发行版上的本地安装程序需要 `libgcc`、`libstdc++` 和 `ripgrep`。使用您的发行版的包管理器安装这些，然后设置 `USE_BUILTIN_RIPGREP=0`。

在 Alpine 上：

```
apk add libgcc libstdc++ ripgrep
```

认证**对于个人用户**

1. **Claude Pro 或 Max 计划**（推荐）：订阅 Claude 的 [Pro 或 Max 计划](#)，获得统一订阅，包括 Claude Code 和网页版 Claude。在一个地方管理您的账户，并使用您的 Claude.ai 账户登录。
2. **Claude Console**：通过 [Claude Console](#) 连接并完成 OAuth 流程。需要在 Anthropic Console 中有活跃的账单。“Claude Code”工作区会自动创建用于使用情况跟踪和成本管理。您无法为 Claude Code 工作区创建 API 密钥；它专门用于 Claude Code 使用。

对于团队和组织

1. **Claude for Teams 或 Enterprise**（推荐）：订阅 [Claude for Teams](#) 或 [Claude for Enterprise](#)，获得集中账单、团队管理以及对 Claude Code 和网页版 Claude 的访问权限。团队成员使用其 Claude.ai 账户登录。
2. **Claude Console 团队账单**：设置共享的 [Claude Console](#) 组织，使用团队账单。邀请团队成员并分配角色以进行使用情况跟踪。
3. **云提供商**：配置 Claude Code 以使用 [Amazon Bedrock](#)、[Google Vertex AI](#) 或 [Microsoft Foundry](#)，以便与您现有的云基础设施进行部署。

安装特定版本

本地安装程序接受特定版本号或发布渠道（`latest` 或 `stable`）。您在安装时选择的渠道将成为自动更新的默认渠道。有关更多信息，请查看[配置发布渠道](#)。

要安装最新版本（默认）：

macOS、Linux、WSL

```
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell

```
irm https://claude.ai/install.ps1 | iex
```

Windows CMD

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del  
install.cmd
```

要安装稳定版本：

macOS、Linux、WSL

```
curl -fsSL https://claude.ai/install.sh | bash -s stable
```

Windows PowerShell

```
& ([scriptblock]::Create((irm https://claude.ai/install.ps1))) stable
```

Windows CMD

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd stable &&  
del install.cmd
```

要安装特定版本号：

macOS、Linux、WSL

```
curl -fsSL https://claude.ai/install.sh | bash -s 1.0.58
```

Windows PowerShell

```
& ([scriptblock]::Create((irm https://claude.ai/install.ps1))) 1.0.58
```

Windows CMD

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd 1.0.58 &&  
del install.cmd
```

二进制完整性和代码签名

- 所有平台的 SHA256 校验和发布在发布清单中，目前位于 <https://storage.googleapis.com/claude-code-dist-86c565f3-f756-42ad-8dfa-d59b1c096819/claude-code-releases/{VERSION}/manifest.json>（示例：将 {VERSION} 替换为 2.0.30）
- 签名的二进制文件分发给以下平台：
 - macOS：由 “Anthropic PBC” 签名并由 Apple 公证
 - Windows：由 “Anthropic, PBC” 签名

NPM 安装（已弃用）

NPM 安装已弃用。尽可能使用[本地安装](#)方法。要将现有 npm 安装迁移到本地，请运行 `claude install`。

全局 npm 安装

```
npm install -g @anthropic-ai/claude-code
```

Warning:

不要使用 `sudo npm install -g`，因为这可能导致权限问题和安全风险。如果您遇到权限错误，请查看[故障排除权限错误](#)以获取推荐的解决方案。

Windows 设置

选项 1：WSL 中的 Claude Code

- 支持 WSL 1 和 WSL 2
- WSL 2 支持[沙箱](#)以增强安全性。WSL 1 不支持沙箱。

选项 2：使用 Git Bash 在本地 Windows 上运行 Claude Code

- 需要 [Git for Windows](#)
- 对于便携式 Git 安装，指定您的 `bash.exe` 的路径：

```
$env:CLAUDE_CODE_GIT_BASH_PATH="C:\Program Files\Git\bin\bash.exe"
```

更新 Claude Code

自动更新

Claude Code 会自动保持最新状态，以确保您拥有最新的功能和安全修复。

- **更新检查**：在启动时和运行时定期执行
- **更新过程**：在后台自动下载和安装
- **通知**：安装更新时您会看到通知
- **应用更新**：更新在您下次启动 Claude Code 时生效

Note:

Homebrew 和 WinGet 安装不会自动更新。使用 `brew upgrade claude-code` 或 `winget upgrade Anthropic.ClaudeCode` 手动更新。

已知问题：Claude Code 可能会在新版本在这些包管理器中可用之前通知您有更新。如果升级失败，请稍候后重试。

配置发布渠道

使用 `autoUpdatesChannel` 设置配置 Claude Code 为自动更新和 `claude update` 遵循的发布渠道：

- `"latest"`（默认）：在新功能发布后立即接收
- `"stable"`：使用通常约一周前的版本，跳过有重大回归的发布

通过 `/config` → **自动更新渠道** 配置此项，或将其添加到您的 [settings.json 文件](#)：

```
{  
  "autoUpdatesChannel": "stable"  
}
```

对于企业部署，您可以使用[托管设置](#)在整个组织中强制执行一致的发布渠道。

禁用自动更新

在您的 shell 或 [settings.json 文件](#) 中设置 `DISABLE_AUTOUPDATER` 环境变量：

```
export DISABLE_AUTOUPDATER=1
```

手动更新

```
claude update
```

卸载 Claude Code

如果您需要卸载 Claude Code，请按照您的安装方法的说明进行操作。

本地安装

删除 Claude Code 二进制文件和版本文件：

macOS、Linux、WSL：

```
rm -f ~/.local/bin/claude  
rm -rf ~/.local/share/claude
```

Windows PowerShell：

```
Remove-Item -Path "$env:USERPROFILE\.local\bin\claude.exe" -Force  
Remove-Item -Path "$env:USERPROFILE\.local\share\claude" -Recurse -Force
```

Windows CMD：

```
del "%USERPROFILE%\local\bin\claude.exe"  
rmdir /s /q "%USERPROFILE%\local\share\claude"
```

Homebrew 安装

```
brew uninstall --cask claude-code
```

WinGet 安装

```
winget uninstall Anthropic.ClaudeCode
```

NPM 安装

```
npm uninstall -g @anthropic-ai/claude-code
```

清理配置文件（可选）

Warning:

删除配置文件将删除您的所有设置、允许的工具、MCP 服务器配置和会话历史记录。

要删除 Claude Code 设置和缓存数据：

macOS、Linux、WSL：

```
## 删除用户设置和状态  
rm -rf ~/.claude  
rm ~/.claude.json  
  
## 删除项目特定设置（从您的项目目录运行）  
rm -rf .claude  
rm -f .mcp.json
```

Windows PowerShell：

删除用户设置和状态

```
Remove-Item -Path "$env:USERPROFILE\.claude" -Recurse -Force
```

```
Remove-Item -Path "$env:USERPROFILE\.claude.json" -Force
```

删除项目特定设置（从您的项目目录运行）

```
Remove-Item -Path ".claude" -Recurse -Force
```

```
Remove-Item -Path ".mcp.json" -Force
```

Windows CMD :

REM 删除用户设置和状态

```
rmdir /s /q "%USERPROFILE%.claude"
```

```
del "%USERPROFILE%.claude.json"
```

REM 删除项目特定设置（从您的项目目录运行）

```
rmdir /s /q ".claude"
```

```
del ".mcp.json"
```


Claude Code 如何工作

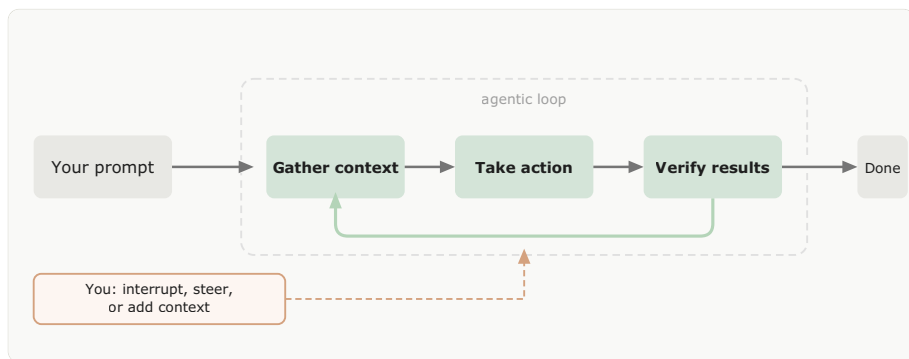
了解代理循环、内置工具以及 Claude Code 如何与您的项目交互。

Claude Code 是一个在您的终端中运行的代理助手。虽然它在编码方面表现出色，但它可以帮助您完成从命令行可以做的任何事情：编写文档、运行构建、搜索文件、研究主题等。

本指南涵盖核心架构、内置功能和[有效使用 Claude Code 的提示](#)。有关分步演练，请参阅[常见工作流](#)。有关 skills、MCP 和 hooks 等可扩展性功能，请参阅[扩展 Claude Code](#)。

代理循环

当您给 Claude 一个任务时，它会经历三个阶段：**收集上下文**、**采取行动**和**验证结果**。这些阶段相互融合。Claude 始终使用工具，无论是搜索文件以了解您的代码、编辑以进行更改，还是运行测试以检查其工作。



代理循环：您的提示导致 Claude 收集上下文、采取行动、验证结果，并重复直到任务完成。您可以在任何时刻中断。

循环会根据您的要求进行调整。关于您代码库的问题可能只需要收集上下文。错误修复会循环通过所有三个阶段多次。重构可能涉及广泛的验证。Claude 根据从前一步学到的内容决定每一步需要什么，将数十个操作链接在一起并沿途进行纠正。

您也是这个循环的一部分。您可以在任何时刻中断以引导 Claude 朝不同方向发展、提供额外上下文或要求它尝试不同的方法。Claude 自主工作但对您的输入保持响应。

代理循环由两个组件驱动：[推理的模型](#)和[采取行动的工具有效使用 Claude Code 的提示](#)。Claude Code 充当围绕 Claude 的[代理工具](#)：它提供工具、上下文管理和执行环境，将语言模型转变为能够进行编码的代理。

模型

Claude Code 使用 Claude 模型来理解您的代码并推理任务。Claude 可以读取任何语言的代码、理解组件如何连接，以及找出需要改变什么来实现您的目标。对于复杂任务，它将工作分解为步骤、执行它们，并根据学到的内容进行调整。

[多个模型](#)可用，具有不同的权衡。Sonnet 可以很好地处理大多数编码任务。Opus 为复杂的架构决策提供 stronger 的推理能力。在会话期间使用 `/model` 切换或使用 `claude --model <name>` 启动。

当本指南说” Claude 选择” 或” Claude 决定” 时，是模型在进行推理。

工具

工具是使 Claude Code 成为代理的原因。没有工具，Claude 只能用文本回应。有了工具，Claude 可以采取行动：读取您的代码、编辑文件、运行命令、搜索网络并与外部服务交互。每个工具使用都会返回信息，反馈到循环中，告知 Claude 的下一个决定。

内置工具通常分为五类，每一类代表不同类型的代理能力。

类别	Claude 可以做什么
文件操作	读取文件、编辑代码、创建新文件、重命名和重新组织
搜索	按模式查找文件、使用正则表达式搜索内容、探索代码库
执行	运行 shell 命令、启动服务器、运行测试、使用 git
网络	搜索网络、获取文档、查找错误消息
代码智能	编辑后查看类型错误和警告、跳转到定义、查找引用（需要 代码智能插件 ）

这些是主要功能。Claude 还有用于生成 subagents、询问您问题和其他编排任务的工具。有关完整列表，请参阅[Claude 可用的工具](#)。

Claude 根据您的提示和沿途学到的内容选择使用哪些工具。当您说” 修复失败的测试” 时，Claude 可能会：

- 1. 运行测试套件以查看失败的内容
- 2. 读取错误输出
- 3. 搜索相关源文件
- 4. 读取这些文件以理解代码
- 5. 编辑文件以修复问题

6. 再次运行测试以验证

每个工具使用都给 Claude 新信息，告知下一步。这就是代理循环的实际应用。

扩展基本功能： 内置工具是基础。您可以使用 [skills](#) 扩展 Claude 知道的内容、使用 [MCP](#) 连接到外部服务、使用 [hooks](#) 自动化工作流，以及使用 [subagents](#) 卸载任务。这些扩展形成了核心代理循环之上的一层。有关为您的需求选择正确扩展的指导，请参阅[扩展 Claude Code](#)。

Claude 可以访问什么

本指南重点关注终端。Claude Code 也在 [VS Code](#)、[JetBrains IDE](#) 和其他环境中运行。

当您在目录中运行 `cclaude` 时，Claude Code 可以访问：

- **您的项目。** 您目录和子目录中的文件，以及您允许的其他地方的文件。
- **您的终端。** 您可以运行的任何命令：构建工具、git、包管理器、系统实用程序、脚本。如果您可以从命令行做到，Claude 也可以。
- **您的 git 状态。** 当前分支、未提交的更改和最近的提交历史。
- **您的 `CLAUDE.md`。** 一个 markdown 文件，您可以在其中存储项目特定的说明、约定和 Claude 应该在每个会话中了解的上下文。
- **自动内存。** Claude 在您工作时自动保存的学习内容，如项目模式和您的偏好。`MEMORY.md` 的前 200 行在每个会话开始时加载。
- **您配置的扩展。** 用于外部服务的 [MCP servers](#)、用于工作流的 [skills](#)、用于委派工作的 [subagents](#) 和用于浏览器交互的 [Claude in Chrome](#)。

因为 Claude 看到您的整个项目，它可以跨越它工作。当您要求 Claude “修复身份验证错误”时，它搜索相关文件、读取多个文件以理解上下文、跨它们进行协调编辑、运行测试以验证修复，并在您要求时提交更改。这与只看到当前文件的内联代码助手不同。

环境和界面

上面描述的代理循环、工具和功能在您使用 Claude Code 的任何地方都是相同的。改变的是代码执行的位置以及您与它交互的方式。

执行环境

Claude Code 在三个环境中运行，每个环境对代码执行位置有不同的权衡。

环境	代码运行位置	用例
本地	您的机器	默认。完全访问您的文件、工具和环境

环境	代码运行位置	用例
云	Anthropic 管理的虚拟机	卸载任务、处理您本地没有的仓库
远程控制	您的机器，从浏览器控制	使用网络 UI 同时保持一切本地

界面

您可以通过终端、[桌面应用](#)、[IDE 扩展](#)、[claude.ai/code](#)、[远程控制](#)、[Slack](#) 和 [CI/CD 管道](#) 访问 Claude Code。界面决定了您如何看到和与 Claude 交互，但底层代理循环是相同的。有关完整列表，请参阅[在任何地方使用 Claude Code](#)。

使用会话

Claude Code 在您工作时将您的对话保存在本地。每条消息、工具使用和结果都被存储，这使得[回退](#)、[恢复和分叉](#)会话成为可能。在 Claude 进行代码更改之前，它还会对受影响的文件进行快照，以便您在需要时可以恢复。

会话是独立的。 每个新会话都以新的上下文窗口开始，没有来自以前会话的对话历史。Claude 可以使用[自动内存](#)跨会话保持学习，您可以在 [CLAUDE.md](#) 中添加您自己的持久说明。

跨分支工作

每个 Claude Code 对话都是与您当前目录绑定的会话。当您恢复时，您只会看到来自该目录的会话。

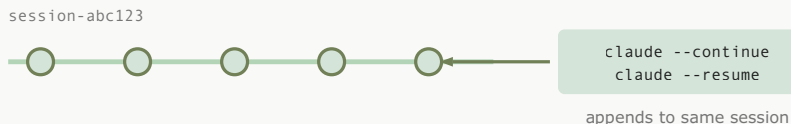
Claude 看到您当前分支的文件。当您切换分支时，Claude 看到新分支的文件，但您的对话历史保持不变。Claude 记得您讨论过的内容，即使在切换后也是如此。

由于会话与目录绑定，您可以通过使用 [git worktrees](#) 运行并行 Claude 会话，这为各个分支创建单独的目录。

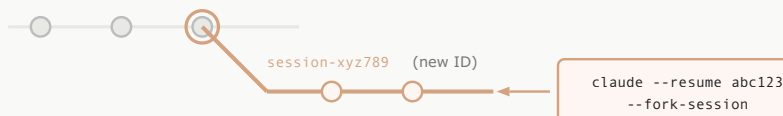
恢复或分叉会话

当您使用 `claude --continue` 或 `claude --resume` 恢复会话时，您使用相同的会话 ID 从中断处继续。新消息附加到现有对话。您的完整对话历史被恢复，但会话范围的权限不会。您需要重新批准这些。

Session continuity



Fork creates a new session from any point:



会话连续性：恢复继续相同的会话，分叉创建具有新 ID 的新分支。

要分支并尝试不同的方法而不影响原始会话，请使用 `--fork-session` 标志：

```
claude --continue --fork-session
```

这会创建一个新的会话 ID，同时保留到该点的对话历史。原始会话保持不变。与恢复一样，分叉的会话不继承会话范围的权限。

在多个终端中的相同会话：如果您在多个终端中恢复相同的会话，两个终端都会写入相同的会话文件。来自两者的消息会交错，就像两个人在同一个笔记本中写字一样。没有任何内容损坏，但对话变得混乱。每个终端在会话期间只看到自己的消息，但如果您稍后恢复该会话，您会看到所有内容交错。对于从相同起点的并行工作，使用 `--fork-session` 为每个终端提供自己的干净会话。

上下文窗口

Claude 的上下文窗口保存您的对话历史、文件内容、命令输出、[CLAUDE.md](#)、加载的 skills 和系统说明。当您工作时，上下文填满。Claude 自动压缩，但对话早期的说明可能会丢失。将持久规则放在 [CLAUDE.md](#) 中，并运行 `/context` 以查看什么在占用空间。

当上下文填满时

Claude Code 在您接近限制时自动管理上下文。它首先清除较旧的工具输出，然后在需要时总结对话。您的请求和关键代码片段被保留；对话早期的详细说明可能会丢失。将持久规则放在 [CLAUDE.md](#) 中，而不是依赖对话历史。

要控制在压缩期间保留的内容，请在 CLAUDE.md 中添加” Compact Instructions” 部分或使用焦点运行 `/compact`（如 `/compact focus on the API changes`）。

运行 `/context` 以查看什么在占用空间。MCP servers 将工具定义添加到每个请求，因此几个服务器可以在您开始工作之前消耗大量上下文。运行 `/mcp` 以检查每个服务器的成本。

使用 skills 和 subagents 管理上下文

除了压缩，您可以使用其他功能来控制什么加载到上下文中。

Skills 按需加载。Claude 在会话开始时看到 skill 描述，但完整内容仅在使用 skill 时加载。对于您手动调用的 skills，设置 `disable-model-invocation: true` 以将描述保留在上下文之外，直到您需要它们。

Subagents 获得自己的新上下文，完全独立于您的主对话。他们的工作不会使您的上下文膨胀。完成后，他们返回摘要。这种隔离是为什么 subagents 有助于长会话。

有关每个功能的成本，请参阅[上下文成本](#)，有关管理上下文的提示，请参阅[减少令牌使用](#)。

使用检查点和权限保持安全

Claude 有两个安全机制：检查点让您撤销文件更改，权限控制 Claude 可以在不询问的情况下做什么。

使用检查点撤销更改

每个文件编辑都是可逆的。 在 Claude 编辑任何文件之前，它会对当前内容进行快照。如果出现问题，按两次 `Esc` 以回退到之前的状态，或要求 Claude 撤销。

检查点是会话本地的，独立于 git。它们仅涵盖文件更改。影响远程系统（数据库、API、部署）的操作无法检查点，这就是为什么 Claude 在运行具有外部副作用的命令之前询问。

控制 Claude 可以做什么

按 `Shift+Tab` 循环通过权限模式：

- **默认**：Claude 在文件编辑和 shell 命令之前询问
- **自动接受编辑**：Claude 编辑文件而不询问，仍然询问命令
- **Plan Mode**：Claude 仅使用只读工具，创建您可以在执行前批准的计划

您也可以在 `.claude/settings.json` 中允许特定命令，以便 Claude 不会每次都询问。这对于受信任的命令（如 `npm test` 或 `git status`）很有用。设置可以从组织范围的策略范围到个人偏好。有关详细信息，请参阅[权限](#)。

有效使用 Claude Code

这些提示可以帮助您从 Claude Code 获得更好的结果。

向 Claude Code 寻求帮助

Claude Code 可以教您如何使用它。提出问题，如“我如何设置 hooks？”或“构建我的 CLAUDE.md 的最佳方式是什么？”Claude 会解释。

内置命令也会指导您完成设置：

- `/init` 引导您为项目创建 CLAUDE.md
- `/agents` 帮助您配置自定义 subagents
- `/doctor` 诊断您的安装的常见问题

这是一个对话

Claude Code 是对话式的。您不需要完美的提示。从您想要的开始，然后细化：

修复登录错误

[Claude 调查，尝试一些东西]

这不太对。问题在于会话处理。

[Claude 调整方法]

当第一次尝试不对时，您不会重新开始。您迭代。

中断和引导

您可以在任何时刻中断 Claude。如果它走错了路，只需输入您的更正并按 Enter。Claude 将停止正在做的事情并根据您的输入调整其方法。您不必等待它完成或重新开始。

预先具体

您的初始提示越精确，您需要的更正就越少。参考特定文件、提及约束并指出示例模式。

结账流程对于持有过期卡的用户来说已损坏。

检查 `src/payments/` 中的问题，特别是令牌刷新。

首先编写一个失败的测试，然后修复它。

模糊的提示有效，但您会花更多时间引导。像上面这样的具体提示通常在第一次尝试时就成功。

给 Claude 一些东西来验证

Claude 在能够检查自己的工作时表现更好。包括测试用例、粘贴预期 UI 的屏幕截图，或定义您想要的输出。

```
实现 validateEmail。测试用例: 'user@example.com' → true,
'invalid' → false, 'user@.com' → false。之后运行测试。
```

对于视觉工作，粘贴设计的屏幕截图并要求 Claude 将其实现与其进行比较。

在实现之前探索

对于复杂问题，将研究与编码分开。使用 plan mode（按 **Shift+Tab** 两次）首先分析代码库：

```
读取 src/auth/ 并理解我们如何处理会话。
然后为添加 OAuth 支持创建一个计划。
```

审查计划，通过对话细化它，然后让 Claude 实现。这种两阶段方法比直接跳到代码产生更好的结果。

委派，不要指示

想象委派给一个有能力的同事。提供上下文和方向，然后相信 Claude 会弄清楚细节：

```
结账流程对于持有过期卡的用户来说已损坏。
相关代码在 src/payments/ 中。您可以调查并修复它吗？
```

您不需要指定要读取哪些文件或运行什么命令。Claude 会弄清楚。

接下来是什么

- [使用功能扩展](#) 添加 Skills、MCP 连接和自定义命令
- [常见 workflows](#) 典型任务的分步指南

Part 2: Core Usage

扩展 Claude Code

了解何时使用 [CLAUDE.md](#)、[Skills](#)、[subagents](#)、[hooks](#)、[MCP](#) 和 [plugins](#)。

Claude Code 结合了一个能够推理代码的模型和[内置工具](#)，用于文件操作、搜索、执行和网络访问。内置工具涵盖了大多数编码任务。本指南涵盖扩展层：您添加的功能，用于自定义 Claude 的知识、将其连接到外部服务以及自动化工作流。

Note:

有关核心代理循环如何工作的信息，请参阅 [Claude Code 如何工作](#)。

初次使用 Claude Code？从 [CLAUDE.md](#) 开始了解项目约定。根据需要添加其他扩展。

概述

扩展插入代理循环的不同部分：

- [CLAUDE.md](#) 添加 Claude 每个会话都能看到的持久上下文
- [Skills](#) 添加可重用的知识和可调用的工作流
- [MCP](#) 将 Claude 连接到外部服务和工具
- [Subagents](#) 在隔离的上下文中运行自己的循环，返回摘要
- [Agent teams](#) 协调多个独立会话，具有共享任务和点对点消息传递
- [Hooks](#) 完全在循环外作为确定性脚本运行
- [Plugins](#) 和 [marketplaces](#) 打包和分发这些功能

[Skills](#) 是最灵活的扩展。Skill 是一个包含知识、工作流或说明的 markdown 文件。您可以使用 `/deploy` 之类的命令调用 skills，或者 Claude 可以在相关时自动加载它们。Skills 可以在您当前的对话中运行，也可以通过 subagents 在隔离的上下文中运行。

将功能与您的目标相匹配

功能范围从 Claude 每个会话都能看到的始终开启的上下文，到您或 Claude 可以调用的按需功能，再到在特定事件上运行的后台自动化。下表显示了可用的功能以及何时使用每个功能。

功能	作用	何时使用	示例
CLAUDE.md	每次对话加载的持久上下文	项目约定、“始终执行 X” 规则	“使用 pnpm，而不是 npm。提交前运行测试。”
Skill	Claude 可以使用的说明、知识和工作流	可重用内容、参考文档、可重复的任务	<code>/deploy</code> 运行您的部署清单；包含端点模式的 API 文档 skill
Subagent	返回摘要结果的隔离执行上下文	上下文隔离、并行任务、专门的工作者	读取许多文件但仅返回关键发现的研究任务
Agent teams	协调多个独立的 Claude Code 会话	并行研究、新功能开发、使用竞争假设进行调试	生成审查者同时检查安全性、性能和测试
MCP	连接到外部服务	外部数据或操作	查询您的数据库、发布到 Slack、控制浏览器
Hook	在事件上运行的确定性脚本	可预测的自动化，不涉及 LLM	每次文件编辑后运行 ESLint

[Plugins](#) 是打包层。Plugin 将 skills、hooks、subagents 和 MCP servers 捆绑到单个可安装单元中。Plugin skills 是命名空间的（如 `/my-plugin:review`），因此多个 plugins 可以共存。当您想在多个存储库中重用相同的设置或通过 [marketplace](#) 分发给他人时，使用 plugins。

比较相似的功能

某些功能可能看起来相似。以下是如何区分它们。

Skill vs Subagent

Skills 和 subagents 解决不同的问题：

- **Skills** 是可重用的内容，您可以将其加载到任何上下文中
- **Subagents** 是与您的主对话分开运行的隔离工作者

方面	Skill	Subagent
它是什么	可重用的说明、知识或工作流	具有自己上下文的隔离工作者

方面	Skill	Subagent
关键优势	在上下文之间共享内容	上下文隔离。工作单独进行，仅返回摘要
最适合	参考材料、可调用的工作流	读取许多文件的任务、并行工作、专门的工作者

Skills 可以是参考或操作。 参考 skills 提供 Claude 在整个会话中使用的知识（如您的 API 风格指南）。操作 skills 告诉 Claude 执行特定操作（如运行您的部署工作流的 `/deploy`）。

当您需要上下文隔离或上下文窗口变满时，使用 subagent。 Subagent 可能读取数十个文件或运行广泛的搜索，但您的主对话仅接收摘要。由于 subagent 工作不消耗您的主上下文，当您不需要中间工作保持可见时，这也很有用。自定义 subagents 可以有自己的说明并可以预加载 skills。

它们可以结合。 Subagent 可以预加载特定的 skills（`skills:` 字段）。Skill 可以使用 `context: fork` 在隔离的上下文中运行。有关详细信息，请参阅 [Skills](#)。

CLAUDE.md vs Skill

两者都存储说明，但它们的加载方式和用途不同。

方面	CLAUDE.md	Skill
加载	每个会话，自动	按需
可以包含文件	是，使用 <code>@path</code> 导入	是，使用 <code>@path</code> 导入
可以触发工作流	否	是，使用 <code>/<name></code>
最适合	“始终执行 X” 规则	参考材料、可调用的工作流

如果 Claude 应该始终知道它，请将其放在 CLAUDE.md 中： 编码约定、构建命令、项目结构、“永远不要执行 X” 规则。

如果它是 Claude 有时需要的参考材料（API 文档、风格指南）或您使用 `/<name>` 触发的工作流（部署、审查、发布），请将其放在 skill 中。

经验法则： 保持 CLAUDE.md 在 200 行以下。如果它在增长，将参考内容移到 skills 或拆分为 `.claude/rules/` 文件。

CLAUDE.md vs Rules vs Skills

所有三者都存储说明，但它们的加载方式不同：

方面	CLAUDE.md	.claude/rules/	Skill
加载	每个会话	每个会话，或当打开匹配的文件时	按需，当调用或相关时
范围	整个项目	可以限定到文件路径	特定于任务
最适合	核心约定和构建命令	特定于语言或目录的指南	参考材料、可重复的工作流

对于每个会话需要的说明，使用 **CLAUDE.md**：构建命令、测试约定、项目架构。

使用 **rules** 来保持 **CLAUDE.md** 专注。带有 `paths` `frontmatter` 的 **rules** 仅在 Claude 处理匹配文件时加载，节省上下文。

对于 Claude 有时只需要的内容，使用 **skills**，如 API 文档或您使用 `/<name>` 触发的部署清单。

Subagent vs Agent team

两者都并行化工作，但它们在架构上不同：

- **Subagents** 在您的会话内运行并将结果报告回您的主上下文
- **Agent teams** 是相互通信的独立 Claude Code 会话

方面	Subagent	Agent team
上下文	自己的上下文窗口；结果返回给调用者	自己的上下文窗口；完全独立
通信	仅向主代理报告结果	队友直接相互发送消息
协调	主代理管理所有工作	具有自我协调的共享任务列表
最适合	仅结果重要的专注任务	需要讨论和协作的复杂工作
令牌成本	较低：结果摘要返回到主上下文	较高：每个队友是一个单独的 Claude 实例

当您需要一个快速、专注的工作者时，使用 **subagent**：研究一个问题、验证一个声明、审查一个文件。Subagent 完成工作并返回摘要。您的主对话保持清洁。

当队友需要共享发现、相互质疑和独立协调时，使用 **agent team**。Agent teams 最适合具有竞争假设的研究、并行代码审查以及每个队友拥有单独部分的新功能开发。

过渡点： 如果您运行并行 subagents 但遇到上下文限制，或者您的 subagents 需要相互通信，agent teams 是自然的下一步。

Note:

Agent teams 是实验性的，默认禁用。有关设置和当前限制，请参阅 [agent teams](#)。

MCP vs Skill

MCP 将 Claude 连接到外部服务。Skills 扩展 Claude 的知识，包括如何有效地使用这些服务。

方面	MCP	Skill
它是什么	连接到外部服务的协议	知识、工作流和参考材料
提供	工具和数据访问	知识、工作流、参考材料
示例	Slack 集成、数据库查询、浏览器控制	代码审查清单、部署工作流、API 风格指南

这些解决不同的问题，可以很好地协同工作：

MCP 给予 Claude 与外部系统交互的能力。没有 MCP，Claude 无法查询您的数据库或发布到 Slack。

Skills 给予 Claude 关于如何有效使用这些工具的知识，以及您可以使用 `/<name>` 触发的工作流。Skill 可能包括您团队的数据库架构和查询模式，或带有您团队消息格式规则的 `/post-to-slack` 工作流。

示例：MCP 服务器将 Claude 连接到您的数据库。Skill 教导 Claude 您的数据模型、常见查询模式以及用于不同任务的表。

了解功能如何分层

功能可以在多个级别定义：用户范围、每个项目、通过 plugins 或通过托管策略。您还可以在子目录中嵌套 CLAUDE.md 文件或在 monorepo 的特定包中放置 skills。当相同的功能存在于多个级别时，以下是它们的分层方式：

- **CLAUDE.md 文件** 是累加的：所有级别同时向 Claude 的上下文贡献内容。来自您的工作目录及以上的文件在启动时加载；子目录在您在其中工作时加载。当说明冲突时，Claude 使用判断来协调它们，更具体的说明通常优先。有关详细信息，请参阅 [CLAUDE.md 文件如何加载](#)。

- **Skills 和 subagents** 按名称覆盖：当相同的名称存在于多个级别时，一个定义根据优先级获胜（对于 skills 为托管 > 用户 > 项目；对于 subagents 为托管 > CLI 标志 > 项目 > 用户 > plugin）。Plugin skills 是 [命名空间的](#) 以避免冲突。有关详细信息，请参阅 [skill 发现](#) 和 [subagent 范围](#)。
- **MCP 服务器** 按名称覆盖：本地 > 项目 > 用户。有关详细信息，请参阅 [MCP 范围](#)。
- **Hooks** 合并：所有注册的 hooks 为其匹配的事件触发，无论来源如何。有关详细信息，请参阅 [hooks](#)。

组合功能

每个扩展解决不同的问题：CLAUDE.md 处理始终开启的上下文，skills 处理按需知识和工作流，MCP 处理外部连接，subagents 处理隔离，hooks 处理自动化。真实的设置根据您的工作流组合它们。

例如，您可能使用 CLAUDE.md 处理项目约定、使用 skill 处理部署工作流、使用 MCP 连接到数据库、使用 hook 在每次编辑后运行 linting。每个功能处理它最擅长的事情。

模式	工作原理	示例
Skill + MCP	MCP 提供连接；skill 教导 Claude 如何很好地使用它	MCP 连接到您的数据库，skill 记录您的架构和查询模式
Skill + Subagent	Skill 为并行工作生成 subagents	<code>/audit</code> skill 启动在隔离上下文中工作的安全性、性能和风格 subagents
CLAUDE.md + Skills	CLAUDE.md 保存始终开启的规则；skills 保存按需加载的参考材料	CLAUDE.md 说”遵循我们的 API 约定”，skill 包含完整的 API 风格指南
Hook + MCP	Hook 通过 MCP 触发外部操作	编辑后 hook 在 Claude 修改关键文件时发送 Slack 通知

了解上下文成本

您添加的每个功能都会消耗 Claude 的一些上下文。太多可能会填满您的上下文窗口，但它也可能增加噪音，使 Claude 效率降低；skills 可能无法正确触发，或 Claude 可能会失去对您的约定的跟踪。了解这些权衡有助于您构建有效的设置。

按功能的上下文成本

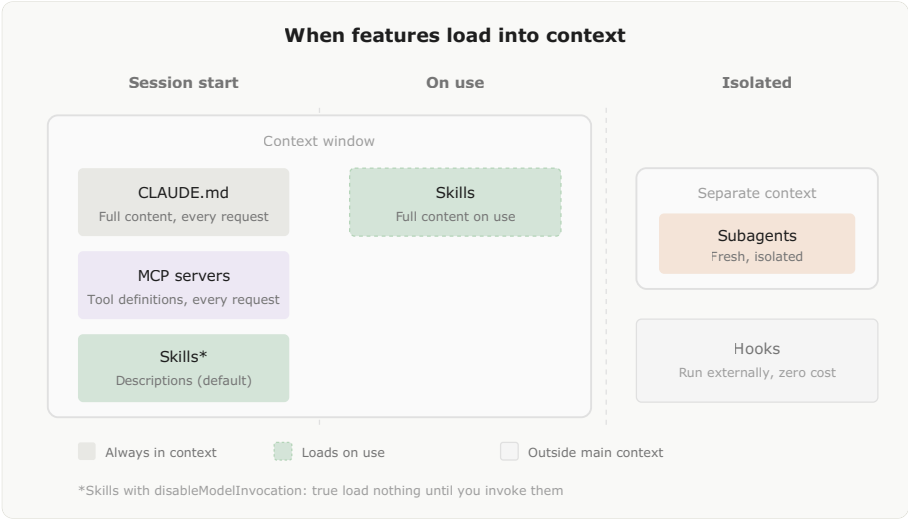
每个功能都有不同的加载策略和上下文成本：

功能	何时加载	加载内容	上下文成本
CLAUDE.md	会话开始	完整内容	每个请求
Skills	会话开始 + 使用时	启动时的描述，使用时的完整内容	低（每个请求的描述）*
MCP 服务器	会话开始	所有工具定义和 JSON 架构	每个请求
Subagents	生成时	具有指定 skills 的新鲜上下文	与主会话隔离
Hooks	触发时	无（外部运行）	零，除非 hook 返回额外上下文

*默认情况下，skill 描述在会话开始时加载，以便 Claude 可以决定何时使用它们。在 skill 的 frontmatter 中设置 `disable-model-invocation: true` 以将其完全隐藏在 Claude 中，直到您手动调用它。这将 skills 的上下文成本降低到零，您只需自己触发这些 skills。

了解功能如何加载

每个功能在会话的不同点加载。下面的选项卡解释了每个功能何时加载以及什么进入上下文。



上下文加载：CLAUDE.md 和 MCP 在会话开始时加载并保留在每个请求中。Skills 在启动时加载描述，在调用时加载完整内容。Subagents 获得隔离的上下文。Hooks 外部运行。

CLAUDE.md

何时： 会话开始

加载内容： 所有 CLAUDE.md 文件的完整内容（托管、用户和项目级别）。

继承： Claude 从您的工作目录读取 CLAUDE.md 文件直到根目录，并在访问这些文件时发现子目录中的嵌套文件。有关详细信息，请参阅 [CLAUDE.md 文件如何加载](#)。

Tip:

保持 CLAUDE.md 在约 500 行以下。将参考材料移到 skills，这些 skills 按需加载。

Skills

Skills 是 Claude 工具包中的额外功能。它们可以是参考材料（如 API 风格指南）或可调用的工作流，您可以使用 `/<name>` 触发（如 `/deploy`）。Claude Code 附带 [捆绑的 skills](#)，如 `/simplify`、`/batch` 和 `/debug`，可以开箱即用。您也可以创建自己的。Claude 在适当时使用 skills，或者您可以直接调用一个。

何时： 取决于 skill 的配置。默认情况下，描述在会话开始时加载，完整内容在使用时加载。对于仅用户 skills（`disable-model-invocation: true`），在您调用它们之前不加载任何内容。

加载内容： 对于模型可调用的 skills，Claude 在每个请求中看到名称和描述。当您使用 `/<name>` 调用 skill 或 Claude 自动加载它时，完整内容加载到您的对话中。

Claude 如何选择 skills： Claude 将您的任务与 skill 描述相匹配，以决定哪些相关。如果描述模糊或重叠，Claude 可能加载错误的 skill 或错过会有帮助的 skill。要告诉 Claude 使用特定的 skill，请使用 `/<name>` 调用它。带有 `disable-model-invocation: true` 的 Skills 对 Claude 不可见，直到您调用它们。

上下文成本： 低，直到使用。仅用户 skills 在调用前成本为零。

在 subagents 中： Skills 在 subagents 中的工作方式不同。不是按需加载，而是传递给 subagent 的 skills 在启动时完全预加载到其上下文中。Subagents 不从主会话继承 skills；您必须明确指定它们。

Tip:

对于有副作用的 skills，使用 `disable-model-invocation: true`。这节省上下文并确保只有您触发它们。

MCP 服务器

何时： 会话开始。

加载内容： 来自连接的服务器的所有工具定义和 JSON 架构。

上下文成本： [工具搜索](#)（默认启用）将 MCP 工具加载到上下文的 10%，并延迟其余部分直到需要。

可靠性说明： MCP 连接可能在会话中途无声地失败。如果服务器断开连接，其工具会无警告地消失。Claude 可能尝试使用不再存在的工具。如果您注意到 Claude 无法使用它之前可以访问的 MCP 工具，请使用 `/mcp` 检查连接。

Tip:

运行 `/mcp` 查看每个服务器的令牌成本。断开您未主动使用的服务器。

Subagents

何时： 按需，当您或 Claude 为任务生成一个时。

加载内容： 新鲜、隔离的上下文，包含：

- 系统提示（与父级共享以提高缓存效率）
- agent 的 `skills:` 字段中列出的 skills 的完整内容
- CLAUDE.md 和 git 状态（从父级继承）
- 主 agent 在提示中传递的任何上下文

上下文成本： 与主会话隔离。Subagents 不继承您的对话历史或调用的 skills。

Tip:

对于不需要您完整对话上下文的工作，使用 subagents。它们的隔离防止膨胀您的主会话。

Hooks

何时： 触发时。Hooks 在特定的生命周期事件上触发，如工具执行、会话边界、提示提交、权限请求和压缩。有关完整列表，请参阅 [Hooks](#)。

加载内容： 默认情况下无。Hooks 作为外部脚本运行。

上下文成本： 零，除非 hook 返回作为消息添加到您的对话中的输出。

Tip:

Hooks 非常适合不需要影响 Claude 上下文的副作用（linting、logging）。

了解更多

每个功能都有自己的指南，包含设置说明、示例和配置选项。

- [CLAUDE.md](#) 存储项目上下文、约定和说明
- [Skills](#) 给予 Claude 领域专业知识和可重用的 workflow
- [Subagents](#) 将工作卸载到隔离的上下文
- [Agent teams](#) 协调多个并行工作的会话
- [MCP](#) 将 Claude 连接到外部服务
- [Hooks](#) 使用 hooks 自动化 workflow
- [Plugins](#) 捆绑和共享功能集
- [Marketplaces](#) 托管和分发 plugin 集合

交互模式

Claude Code 会话中键盘快捷键、输入模式和交互功能的完整参考。

快捷键

Note:

快捷键可能因平台和终端而异。按 [?](#) 查看您的环境中可用的快捷键。

macOS 用户： Option/Alt 键快捷键（[Alt+B](#)、[Alt+F](#)、[Alt+Y](#)、[Alt+M](#)、[Alt+P](#)）需要在终端中将 Option 配置为 Meta：

- **iTerm2：** 设置 → 配置文件 → 键 → 将左/右 Option 键设置为” Esc+”
- **Terminal.app：** 设置 → 配置文件 → 键盘 → 勾选” 使用 Option 作为 Meta 键”
- **VS Code：** 设置 → 配置文件 → 键 → 将左/右 Option 键设置为” Esc+”

有关详细信息，请参阅[终端配置](#)。

常规控制

快捷键	描述	上下文
Ctrl+C	取消当前输入或生成	标准中断
Ctrl+F	终止所有后台代理。在 3 秒内按两次以确认	后台代理控制
Ctrl+D	退出 Claude Code 会话	EOF 信号
Ctrl+G	在默认文本编辑器中打开	在默认文本编辑器中编辑您的提示或自定义响应
Ctrl+L	清除终端屏幕	保留对话历史
Ctrl+O	切换详细输出	显示详细的工具使用和执行情况
Ctrl+R	反向搜索命令历史	交互式搜索以前的命令

快捷键	描述	上下文
<code>Ctrl+V</code> 或 <code>Cmd+V</code> (iTerm2) 或 <code>Alt+V</code> (Windows)	从剪贴板粘贴图像	粘贴图像或图像文件的路径
<code>Ctrl+B</code>	后台运行任务	后台运行 <code>bash</code> 命令和代理。Tmux 用户按两次
<code>Ctrl+T</code>	切换任务列表	在终端状态区域中显示或隐藏 任务列表
<code>Left/Right arrows</code>	在对话框选项卡之间循环	在权限对话框和菜单中的选项卡之间导航
<code>Up/Down arrows</code>	导航命令历史	回忆以前的输入
<code>Esc + Esc</code>	回退或总结	将代码和/或对话恢复到上一个点，或从选定的消息进行总结
<code>Shift+Tab</code> 或 <code>Alt+M</code> (某些配置)	切换权限模式	在自动接受模式、Plan Mode 和正常模式之间切换。
<code>Option+P</code> (macOS) 或 <code>Alt+P</code> (Windows/ Linux)	切换模型	在不清除提示的情况下切换模型
<code>Option+T</code> (macOS) 或 <code>Alt+T</code> (Windows/ Linux)	切换扩展思考	启用或禁用扩展思考模式。 首先运行 <code>/terminal-setup</code> 以启用此快捷键

文本编辑

快捷键	描述	上下文
<code>Ctrl+K</code>	删除到行尾	存储已删除的文本以供粘贴
<code>Ctrl+U</code>	删除整行	存储已删除的文本以供粘贴
<code>Ctrl+Y</code>	粘贴已删除的文本	粘贴用 <code>Ctrl+K</code> 或 <code>Ctrl+U</code> 删除的文本

快捷键	描述	上下文
<code>Alt+Y</code> （在 <code>Ctrl+Y</code> 之后）	循环粘贴历史	粘贴后，循环浏览以前删除的文本。在 macOS 上需要将 <code>Option</code> 作为 <code>Meta</code>
<code>Alt+B</code>	将光标向后移动一个单词	单词导航。在 macOS 上需要将 <code>Option</code> 作为 <code>Meta</code>
<code>Alt+F</code>	将光标向前移动一个单词	单词导航。在 macOS 上需要将 <code>Option</code> 作为 <code>Meta</code>

主题和显示

快捷键	描述	上下文
<code>Ctrl+T</code>	切换代码块的语法突出显示	仅在 <code>/theme</code> 选择器菜单内有效。控制 Claude 响应中的代码是否使用语法着色

Note:

语法突出显示仅在 Claude Code 的原生构建中可用。

多行输入

方法	快捷键	上下文
快速转义	<code>\ + Enter</code>	在所有终端中有效
macOS 默认	<code>Option+Enter</code>	macOS 上的默认值
Shift+Enter	<code>Shift+Enter</code>	在 iTerm2、WezTerm、Ghostty、Kitty 中开箱即用
控制序列	<code>Ctrl+J</code>	多行的换行符
粘贴模式	直接粘贴	对于代码块、日志

Tip:

Shift+Enter 在 iTerm2、WezTerm、Ghostty 和 Kitty 中无需配置即可工作。对于其他终端（VS Code、Alacritty、Zed、Warp），运行 `/terminal-setup` 以安装绑定。

快速命令

快捷键	描述	注释
<code>/</code> 在开始	命令或 skill	请参阅 内置命令 和 skills
<code>!</code> 在开始	Bash 模式	直接运行命令并将执行输出添加到会话
<code>@</code>	文件路径提及	触发文件路径自动完成

内置命令

在 Claude Code 中输入 `/` 以查看所有可用命令，或输入 `/` 后跟任何字母以进行筛选。并非所有命令对每个用户都可见。有些取决于您的平台、计划或环境。例如，`/desktop` 仅在 macOS 和 Windows 上显示，`/upgrade` 和 `/privacy-settings` 仅在 Pro 和 Max 计划上可用，`/terminal-setup` 在您的终端本身支持其快捷键时隐藏。

Claude Code 还附带[捆绑的 skills](#)，如 `/simplify`、`/batch` 和 `/debug`，当您输入 `/` 时会与内置命令一起显示。要创建您自己的命令，请参阅 [skills](#)。

在下表中，`<arg>` 表示必需的参数，`[arg]` 表示可选参数。

命令	目的
<code>/add-dir</code> <code><path></code>	将新的工作目录添加到当前会话
<code>/agents</code>	管理 agent 配置
<code>/btw</code> <code><question></code>	提出快速 侧问题 ，无需添加到对话
<code>/chrome</code>	配置 Chrome 中的 Claude 设置
<code>/clear</code>	清除对话历史并释放上下文。别名： <code>/reset</code> 、 <code>/new</code>
<code>/compact</code> <code>[instructions]</code>	压缩对话，可选的焦点说明
<code>/config</code>	打开 设置 界面以调整主题、模型、 输出样式 和其他首选项。别名： <code>/settings</code>
<code>/context</code>	将当前上下文使用情况可视化彩色网格

命令	目的
<code>/copy</code>	将最后一个助手响应复制到剪贴板。当存在代码块时，显示交互式选择器以选择单个块或完整响应
<code>/cost</code>	显示令牌使用统计信息。有关特定于订阅的详细信息，请参阅 成本跟踪指南
<code>/desktop</code>	在 Claude Code 桌面应用中继续当前会话。仅限 macOS 和 Windows。别名： <code>/app</code>
<code>/diff</code>	打开交互式差异查看器，显示未提交的更改和每个回合的差异。使用左/右箭头在当前 git 差异和单个 Claude 回合之间切换，使用上/下浏览文件
<code>/doctor</code>	诊断并验证您的 Claude Code 安装和设置
<code>/exit</code>	退出 CLI。别名： <code>/quit</code>
<code>/export</code> <code>[filename]</code>	将当前对话导出为纯文本。使用文件名时，直接写入该文件。不使用时，打开对话框以复制到剪贴板或保存到文件
<code>/extra-usage</code>	配置额外使用以在达到速率限制时继续工作
<code>/fast [on off]</code>	切换 快速模式 开启或关闭
<code>/feedback</code> <code>[report]</code>	提交关于 Claude Code 的反馈。别名： <code>/bug</code>
<code>/fork [name]</code>	在此点创建当前对话的分支
<code>/help</code>	显示帮助和可用命令
<code>/hooks</code>	管理工具事件的 hook 配置
<code>/ide</code>	管理 IDE 集成并显示状态
<code>/init</code>	使用 CLAUDE.md 指南初始化项目
<code>/insights</code>	生成报告，分析您的 Claude Code 会话，包括项目区域、交互模式和摩擦点
<code>/install-github-app</code>	为存储库设置 Claude GitHub Actions 应用。引导您选择存储库并配置集成
<code>/install-slack-app</code>	安装 Claude Slack 应用。打开浏览器以完成 OAuth 流程
<code>/keybindings</code>	打开或创建您的快捷键配置文件

命令	目的
<code>/login</code>	登录到您的 Anthropic 账户
<code>/logout</code>	从您的 Anthropic 账户登出
<code>/mcp</code>	管理 MCP server 连接和 OAuth 身份验证
<code>/memory</code>	编辑 <code>CLAUDE.md</code> 内存文件，启用或禁用 自动内存 ，并查看自动内存条目
<code>/mobile</code>	显示二维码以下载 Claude 移动应用。别名： <code>/ios</code> 、 <code>/android</code>
<code>/model [model]</code>	选择或更改 AI 模型。对于支持的模型，使用左/右箭头 调整努力级别 。更改立即生效，无需等待当前响应完成
<code>/passes</code>	与朋友分享一周免费的 Claude Code。仅在您的账户符合条件时可见
<code>/permissions</code>	查看或更新 权限 。别名： <code>/allowed-tools</code>
<code>/plan</code>	直接从提示进入 plan mode
<code>/plugin</code>	管理 Claude Code plugins
<code>/pr-comments [PR]</code>	获取并显示来自 GitHub pull request 的评论。自动检测当前分支的 PR，或传递 PR URL 或编号。需要 <code>gh</code> CLI
<code>/privacy-settings</code>	查看和更新您的隐私设置。仅对 Pro 和 Max 计划订阅者可用
<code>/release-notes</code>	查看完整的更改日志，最新版本最接近您的提示
<code>/reload-plugins</code>	重新加载所有活跃 plugins 以应用待处理的更改，无需重启。报告已加载的内容并注明需要重启的更改
<code>/remote-control</code>	使此会话可从 claude.ai 进行 远程控制 。别名： <code>/rc</code>
<code>/remote-env</code>	为 teleport 会话 配置默认远程环境
<code>/rename [name]</code>	重命名当前会话。不使用名称时，从对话历史自动生成
<code>/resume [session]</code>	按 ID 或名称恢复对话，或打开会话选择器。别名： <code>/continue</code>
<code>/review</code>	已弃用。改为安装 <code>code-review plugin</code> ： <code>claude plugin install code-review@claude-code-marketplace</code>

命令	目的
<code>/rewind</code>	将对话和/或代码回退到上一个点，或从选定的消息进行总结。请参阅 checkpointing 。别名： <code>/checkpoint</code>
<code>/sandbox</code>	切换 sandbox 模式 。仅在支持的平台上可用
<code>/security-review</code>	分析当前分支上的待处理更改以查找安全漏洞。审查 git 差异并识别风险，如注入、身份验证问题和数据泄露
<code>/skills</code>	列出可用的 skills
<code>/stats</code>	可视化每日使用情况、会话历史、连胜和模型偏好
<code>/status</code>	打开设置界面（状态选项卡），显示版本、模型、账户和连接性
<code>/statusline</code>	配置 Claude Code 的 状态行 。描述您想要的内容，或不带参数运行以从您的 shell 提示自动配置
<code>/stickers</code>	订购 Claude Code 贴纸
<code>/tasks</code>	列出和管理后台任务
<code>/terminal-setup</code>	为 Shift+Enter 和其他快捷键配置终端快捷键。仅在需要它的终端中可见，如 VS Code 、Alacritty 或 Warp
<code>/theme</code>	更改颜色主题。包括浅色和深色变体、色盲无障碍（daltonized）主题和使用您终端的调色板的 ANSI 主题
<code>/upgrade</code>	打开升级页面以切换到更高的计划层级
<code>/usage</code>	显示计划使用限制和速率限制状态
<code>/vim</code>	在 Vim 和正常编辑模式之间切换

MCP prompts

MCP servers 可以公开显示为命令的 prompts 。这些使用格式 `/mcp__<server>__<prompt>` ，并从连接的 servers 动态发现。有关详细信息，请参阅 [MCP prompts](#) 。

Vim 编辑器模式

使用 `/vim` 命令启用 vim 风格编辑，或通过 `/config` 永久配置。

模式切换

命令	操作	来自模式
<code>Esc</code>	进入 NORMAL 模式	INSERT
<code>i</code>	在光标前插入	NORMAL
<code>I</code>	在行首插入	NORMAL
<code>a</code>	在光标后插入	NORMAL
<code>A</code>	在行尾插入	NORMAL
<code>o</code>	在下方打开行	NORMAL
<code>O</code>	在上方打开行	NORMAL

导航（NORMAL 模式）

命令	操作
<code>h / j / k / l</code>	向左/下/上/右移动
<code>w</code>	下一个单词
<code>e</code>	单词末尾
<code>b</code>	上一个单词
<code>0</code>	行首
<code>\$</code>	行尾
<code>^</code>	第一个非空白字符
<code>gg</code>	输入开始
<code>G</code>	输入结束
<code>f{char}</code>	跳转到下一个字符出现处
<code>F{char}</code>	跳转到上一个字符出现处
<code>t{char}</code>	跳转到下一个字符出现处之前
<code>T{char}</code>	跳转到上一个字符出现处之后

命令	操作
<code>;</code>	重复最后一个 f/F/t/T 动作
<code>,</code>	反向重复最后一个 f/F/t/T 动作

Note:

在 vim 正常模式中，如果光标在输入的开始或结束处且无法进一步移动，箭头键将导航命令历史。

编辑（NORMAL 模式）

命令	操作
<code>x</code>	删除字符
<code>dd</code>	删除行
<code>D</code>	删除到行尾
<code>dw / de / db</code>	删除单词/到末尾/向后
<code>cc</code>	更改行
<code>C</code>	更改到行尾
<code>cw / ce / cb</code>	更改单词/到末尾/向后
<code>yy / Y</code>	复制行
<code>yw / ye / yb</code>	复制单词/到末尾/向后
<code>p</code>	在光标后粘贴
<code>P</code>	在光标前粘贴
<code>>></code>	缩进行
<code><<</code>	取消缩进行
<code>J</code>	连接行
<code>.</code>	重复最后一个更改

文本对象（NORMAL 模式）

文本对象与 `d`、`c` 和 `y` 等运算符一起工作：

命令	操作
<code>iw / aw</code>	内部/周围单词
<code>iW / aW</code>	内部/周围 WORD（空白分隔）
<code>i" / a"</code>	内部/周围双引号
<code>i' / a'</code>	内部/周围单引号
<code>i(/ a(</code>	内部/周围括号
<code>i[/ a[</code>	内部/周围方括号
<code>i{ / a{</code>	内部/周围大括号

命令历史

Claude Code 为当前会话维护命令历史：

- 输入历史按工作目录存储
- 当您运行 `/clear` 以启动新会话时，输入历史会重置。上一个会话的对话会被保留，可以恢复。
- 使用上/下箭头导航（请参阅上面的快捷键）
- 注意：历史扩展（`!`）默认禁用

使用 Ctrl+R 反向搜索

按 `Ctrl+R` 以交互方式搜索您的命令历史：

1. **开始搜索：**按 `Ctrl+R` 激活反向历史搜索
2. **输入查询：**输入文本以在以前的命令中搜索。搜索词在匹配结果中突出显示
3. **导航匹配：**再次按 `Ctrl+R` 以循环浏览较旧的匹配
4. **接受匹配：**
 - 按 `Tab` 或 `Esc` 接受当前匹配并继续编辑
 - 按 `Enter` 接受并立即执行命令
5. **取消搜索：**
 - 按 `Ctrl+C` 取消并恢复您的原始输入
 - 在空搜索上按 `Backspace` 以取消

搜索显示匹配的命令，搜索词突出显示，因此您可以找到并重用以前的输入。

后台 bash 命令

Claude Code 支持在后台运行 bash 命令，允许您在长时间运行的进程执行时继续工作。

后台运行的工作原理

当 Claude Code 在后台运行命令时，它异步运行命令并立即返回后台任务 ID。Claude Code 可以在命令继续在后台执行时响应新提示。

要在后台运行命令，您可以：

- 提示 Claude Code 在后台运行命令
- 按 Ctrl+B 将常规 Bash 工具调用移到后台。（Tmux 用户必须按两次 Ctrl+B，因为 tmux 的前缀键。）

主要功能：

- 输出被缓冲，Claude 可以使用 TaskOutput 工具检索它
- 后台任务有唯一的 ID 用于跟踪和输出检索
- 当 Claude Code 退出时，后台任务会自动清理

要禁用所有后台任务功能，请将 `CLAUDE_CODE_DISABLE_BACKGROUND_TASKS` 环境变量设置为 `1`。有关详细信息，请参阅[环境变量](#)。

常见的后台命令：

- 构建工具（webpack、vite、make）
- 包管理器（npm、yarn、pnpm）
- 测试运行器（jest、pytest）
- 开发服务器
- 长时间运行的进程（docker、terraform）

使用 `!` 前缀的 Bash 模式

通过在输入前加上 `!` 来直接运行 bash 命令，无需通过 Claude：

```
! npm test
! git status
! ls -la
```

Bash 模式：

- 将命令及其输出添加到对话上下文
- 显示实时进度和输出
- 支持相同的 **Ctrl+B** 后台运行以处理长时间运行的命令
- 不需要 Claude 解释或批准命令
- 支持基于历史的自动完成：输入部分命令并按 **Tab** 从当前项目中的上一个 **!** 命令完成
- 使用 **Escape** 、 **Backspace** 或在空提示上按 **Ctrl+U** 退出

这对于快速 shell 操作同时保持对话上下文很有用。

提示建议

当您首次打开会话时，灰显的示例命令会出现在提示输入中以帮助您入门。Claude Code 从您项目的 git 历史中选择此命令，因此它反映了您最近一直在处理的文件。

Claude 响应后，建议会根据您的对话历史继续出现，例如多部分请求的后续步骤或工作流的自然延续。

- 按 **Tab** 接受建议，或按 **Enter** 接受并提交
- 开始输入以关闭它

建议作为后台请求运行，重用父对话的 prompt cache，因此额外成本最小。当缓存冷时，Claude Code 会跳过建议生成以避免不必要的成本。

在对话的第一个回合、非交互模式和 plan mode 中，建议会自动跳过。

要完全禁用提示建议，请设置环境变量或在 `/config` 中切换设置：

```
export CLAUDE_CODE_ENABLE_PROMPT_SUGGESTION=false
```

使用 `/btw` 的侧问题

使用 `/btw` 提出关于您当前工作的快速问题，无需添加到对话历史。当您想要快速答案但不想使主上下文混乱或使 Claude 偏离长时间运行的任务时，这很有用。

```
/btw what was the name of that config file again?
```

侧问题可以完全看到当前对话，因此您可以询问 Claude 已经读过的代码、它之前做出的决定或会话中的任何其他内容。问题和答案是短暂的：它们出现在可关闭的覆盖层中，永远不会进入对话历史。

- **在 Claude 工作时可用：**即使 Claude 正在处理响应，您也可以运行 `/btw`。侧问题独立运行，不会中断主回合。
- **无工具访问：**侧问题仅从已在上下文中的内容回答。Claude 在回答侧问题时无法读取文件、运行命令或搜索。
- **单一响应：**没有后续回合。如果您需要来回往复，请改用正常提示。
- **低成本：**侧问题重用父对话的 prompt cache，因此额外成本最小。

按 **Space**、**Enter** 或 **Escape** 关闭答案并返回提示。

`/btw` 是 `subagent` 的反面：它看到您的完整对话但没有工具，而 `subagent` 有完整的工具但从空上下文开始。使用 `/btw` 询问 Claude 从此会话已知的内容；使用 `subagent` 去发现新的东西。

任务列表

在处理复杂的多步骤工作时，Claude 会创建任务列表来跟踪进度。任务出现在终端的状态区域中，指示器显示待处理、进行中或完成的内容。

- 按 `Ctrl+T` 切换任务列表视图。显示一次最多 10 个任务
- 要查看所有任务或清除它们，直接询问 Claude：“show me all tasks” 或 “clear all tasks”
- 任务在上下文压缩中持续存在，帮助 Claude 在较大的项目上保持组织
- 要在会话之间共享任务列表，请设置 `CLAUDE_CODE_TASK_LIST_ID` 以使用 `~/.claude/tasks/` 中的命名目录：`CLAUDE_CODE_TASK_LIST_ID=my-project claude`
- 要恢复到上一个 TODO 列表，请设置 `CLAUDE_CODE_ENABLE_TASKS=false`。

PR 审查状态

在处理具有开放 pull request 的分支时，Claude Code 在页脚中显示可点击的 PR 链接（例如“PR #446”）。该链接有一个彩色下划线，指示审查状态：

- 绿色：已批准
- 黄色：待审查
- 红色：请求更改
- 灰色：草稿
- 紫色：已合并

`Cmd+click` (Mac) 或 `Ctrl+click` (Windows/Linux) 链接以在浏览器中打开 pull request。状态每 60 秒自动更新一次。

Note:

PR 状态需要安装并验证 `gh` CLI (`gh auth login`) 。

另请参阅

- [Skills](#) - 自定义提示和工作流
- [Checkpointing](#) - 回退 Claude 的编辑并恢复以前的状态
- [CLI reference](#) - 命令行标志和选项
- [Settings](#) - 配置选项
- [Memory management](#) - 管理 CLAUDE.md 文件

Claude Code 最佳实践

从配置环境到跨并行会话扩展，充分利用 Claude Code 的提示和模式。

Claude Code 是一个代理式编码环境。与等待回答问题的聊天机器人不同，Claude Code 可以读取你的文件、运行命令、进行更改，并在你观看、重定向或完全离开的情况下自主解决问题。

这改变了你的工作方式。与其自己编写代码并要求 Claude 审查，不如描述你想要什么，让 Claude 弄清楚如何构建它。Claude 会探索、规划和实现。

但这种自主性仍然伴随着学习曲线。Claude 在某些约束条件下工作，你需要理解这些约束。

本指南涵盖了在 Anthropic 内部团队和在各种代码库、语言和环境中使用 Claude Code 的工程师中已被证明有效的模式。有关代理循环如何在幕后工作的信息，请参阅 [Claude Code 如何工作](#)。

大多数最佳实践都基于一个约束：Claude 的 context window 填充速度很快，随着填充，性能会下降。

Claude 的 context window 保存你的整个对话，包括每条消息、Claude 读取的每个文件和每个命令输出。但这可能会很快填满。单个调试会话或代码库探索可能会生成并消耗数百万个 token。

这很重要，因为当 context 填充时，LLM 性能会下降。当 context window 即将满时，Claude 可能会开始“遗忘”早期的指令或犯更多错误。context window 是最重要的资源。使用 [自定义状态行](#) 持续跟踪 context 使用情况，并查看 [减少 token 使用](#) 了解减少 token 使用的策略。

给 Claude 一种验证其工作的方式

Tip:

包括测试、屏幕截图或预期输出，以便 Claude 可以检查自己。这是你能做的最高杠杆的事情。

当 Claude 能够验证自己的工作时，例如运行测试、比较屏幕截图和验证输出，它的表现会显著提高。

没有明确的成功标准，它可能会产生看起来正确但实际上不起作用的东西。你成为唯一的反馈循环，每个错误都需要你的关注。

策略	之前	之后
提供验证标准	“实现一个验证电子邮件地址的函数”	“编写一个 validateEmail 函数。示例测试用例： use r@example.com 为真，invalid 为假， user@.com 为假。实现后运行测试”
以视觉方式验证 UI 更改	“让仪表板看起来更好”	“[粘贴屏幕截图] 实现此设计。对结果进行屏幕截图并与原始设计进行比较。列出差异并修复它们”
解决根本原因，而不是症状	“构建失败”	“构建失败，出现此错误：[粘贴错误]。修复它并验证构建成功。解决根本原因，不要抑制错误”

UI 更改可以使用 [Chrome 中的 Claude 扩展](#) 进行验证。它在浏览器中打开新标签页，测试 UI，并迭代直到代码工作。

你的验证也可以是测试套件、linter 或检查输出的 Bash 命令。投资使你的验证非常可靠。

先探索，再规划，最后编码

Tip:
将研究和规划与实现分开，以避免解决错误的问题。

让 Claude 直接跳到编码可能会产生解决错误问题的代码。使用 [Plan Mode](#) 将探索与执行分开。

推荐的工作流有四个阶段：

Step 1: 探索

进入 Plan Mode。Claude 读取文件并回答问题，不进行任何更改。

```
read /src/auth and understand how we handle sessions and login.  
also look at how we manage environment variables for secrets.
```

Step 2: 规划

要求 Claude 创建详细的实现计划。

```
I want to add Google OAuth. What files need to change?  
What's the session flow? Create a plan.
```

按 **Ctrl+G** 在文本编辑器中打开计划进行直接编辑，然后 Claude 继续。

Step 3: 实现

切换回 Normal Mode 并让 Claude 编码，根据其计划进行验证。

```
implement the OAuth flow from your plan. write tests for the  
callback handler, run the test suite and fix any failures.
```

Step 4: 提交

要求 Claude 使用描述性消息进行提交并创建 PR。

```
commit with a descriptive message and open a PR
```

Plan Mode 很有用，但也增加了开销。

对于范围明确且修复很小的任务（如修复拼写错误、添加日志行或重命名变量），要求 Claude 直接执行。

当你方法不确定、更改修改多个文件或你不熟悉被修改的代码时，规划最有用。如果你能用一句话描述 diff，跳过计划。

在提示中提供具体的上下文

Tip:

你的指令越精确，你需要的更正就越少。

Claude 可以推断意图，但它不能读心术。引用特定文件、提及约束，并指出示例模式。

策略	之前	之后
限定任务范围。 指定哪个文件、什么场景和测试偏好。	“为 foo.py 添加测试”	“为 foo.py 编写测试，涵盖用户已注销的边界情况。避免 mock。”
指向来源。 指导 Claude 到可以回答问题的来源。	“为什么 ExecutionFactory 有这样奇怪的 api？”	“查看 ExecutionFactory 的 git 历史并总结其 api 是如何形成的”
参考现有模式。 指向代码库中的模式。	“添加日历小部件”	“查看主页上现有小部件的实现方式以了解模式。HotDogWidget.php 是一个很好的例子。按照模式实现一个新的日历小部件，让用户选择月份并向前/向后分页以选择年份。从头开始构建，除了代码库中已使用的库外，不使用其他库。”
描述症状。 提供症状、可能的位置以及”修复“的样子。	“修复登录错误”	“用户报告会超时后登录失败。检查 src/auth/ 中的身份验证流程，特别是 token 刷新。编写一个失败的测试来重现问题，然后修复它”

当你在探索并能够改正方向时，模糊的提示可能很有用。像 “你会改进这个文件的什么？” 这样的提示可以表面你不会想到要问的东西。

提供丰富的内容

Tip:

使用 @ 引用文件、粘贴屏幕截图/图像或直接管道数据。

你可以通过多种方式向 Claude 提供丰富的数据：

- 使用 @ 引用文件，而不是描述代码的位置。Claude 在响应前读取文件。
- 直接粘贴图像。复制/粘贴或拖放图像到提示中。
- 提供 URL 用于文档和 API 参考。使用 /permissions 来允许列表经常使用的域。

- **管道数据** 通过运行 `cat error.log | claude` 直接发送文件内容。
- **让 Claude 获取它需要的东西**。告诉 Claude 使用 Bash 命令、MCP 工具或通过读取文件来自己拉取上下文。

配置你的环境

一些设置步骤使 Claude Code 在所有会话中显著更有效。有关扩展功能的完整概述和何时使用每个功能，请参阅 [扩展 Claude Code](#)。

编写有效的 CLAUDE.md

Tip:

运行 `/init` 根据你的当前项目结构生成启动 CLAUDE.md 文件，然后随时间精化。

CLAUDE.md 是一个特殊文件，Claude 在每次对话开始时读取。包括 Bash 命令、代码风格和工作流规则。这给 Claude 提供了它无法从代码中推断的持久上下文。

`/init` 命令分析你的代码库以检测构建系统、测试框架和代码模式，为你提供坚实的基础来精化。

CLAUDE.md 文件没有必需的格式，但保持简短和易读。例如：

```
## Code style
- Use ES modules (import/export) syntax, not CommonJS (require)
- Destructure imports when possible (eg. import { foo } from 'bar')

## Workflow
- Be sure to typecheck when you're done making a series of code changes
- Prefer running single tests, and not the whole test suite, for performance
```

CLAUDE.md 在每个会话中加载，所以只包括广泛适用的东西。对于仅有时相关的域知识或工作流，改用 [skills](#)。Claude 按需加载它们，不会使每次对话都膨胀。

保持简洁。对于每一行，问自己：“删除这个会导致 Claude 犯错吗？” 如果不会，删除它。膨胀的 CLAUDE.md 文件会导致 Claude 忽略你的实际指令！

✅ 包括	❌ 排除
Claude 无法猜测的 Bash 命令	Claude 可以通过读取代码弄清楚的任何东西

✔ 包括	✗ 排除
与默认值不同的代码风格规则	Claude 已经知道的标准语言约定
测试指令和首选测试运行器	详细的 API 文档（改为链接到文档）
存储库礼仪（分支命名、PR 约定）	经常变化的信息
特定于你的项目的架构决策	长解释或教程
开发者环境怪癖（必需的环境变量）	自明的实践，如”编写干净的代码”
常见陷阱或非显而易见的行为	文件逐个描述代码库

如果 Claude 继续做你不想要的事情，尽管有反对的规则，该文件可能太长，规则被遗漏了。如果 Claude 问你在 CLAUDE.md 中回答的问题，措辞可能不明确。像对待代码一样对待 CLAUDE.md：当事情出错时审查它，定期修剪它，并通过观察 Claude 的行为是否实际改变来测试更改。

你可以通过添加强调（例如”IMPORTANT”或”YOU MUST”）来调整指令以改进遵守。将文件检入 git，以便你的团队可以贡献。该文件随时间增加价值。

CLAUDE.md 文件可以使用 `@path/to/import` 语法导入其他文件：

```
See @README.md for project overview and @package.json for available npm commands.

## Additional Instructions
- Git workflow: @docs/git-instructions.md
- Personal overrides: @~/ .claude/my-project-instructions.md
```

你可以在多个位置放置 CLAUDE.md 文件：

- **主文件夹**（`~/ .claude/CLAUDE.md`）：适用于所有 Claude 会话
- **项目根目录**（`./CLAUDE.md`）：检入 git 以与你的团队共享
- **父目录**：对于 monorepos 有用，其中 `root/CLAUDE.md` 和 `root/foo/CLAUDE.md` 都会自动拉入
- **子目录**：当处理这些目录中的文件时，Claude 按需拉入子 CLAUDE.md 文件

配置权限

Tip:

使用 `/permissions` 来允许列表安全命令或 `/sandbox` 用于操作系统级隔离。这减少了中断，同时让你保持控制。

默认情况下，Claude Code 请求可能修改你的系统的操作的权限：文件写入、Bash 命令、MCP 工具等。这是安全的但繁琐。在第十次批准后，你不是真的在审查，你只是点击通过。有两种方式来减少这些中断：

- **权限允许列表**：允许你知道是安全的特定工具（如 `npm run lint` 或 `git commit`）
- **沙箱**：启用操作系统级隔离，限制文件系统和网络访问，允许 Claude 在定义的边界内更自由地工作

或者，使用 `--dangerously-skip-permissions` 来绕过所有权限检查，用于包含的工作流，如修复 lint 错误或生成样板。

Warning:

让 Claude 运行任意命令可能导致数据丢失、系统损坏或通过提示注入进行数据渗透。仅在在没有互联网访问的沙箱中使用 `--dangerously-skip-permissions`。

阅读更多关于 [配置权限](#) 和 [启用沙箱](#)。

使用 CLI 工具

Tip:

告诉 Claude Code 在与外部服务交互时使用 CLI 工具，如 `gh`、`aws`、`gcloud` 和 `sentry-cli`。

CLI 工具是与外部服务交互的最 context 高效的方式。如果你使用 GitHub，安装 `gh` CLI。Claude 知道如何使用它来创建问题、打开拉取请求和读取评论。没有 `gh`，Claude 仍然可以使用 GitHub API，但未认证的请求经常会触发速率限制。

Claude 也有效地学习它不知道的 CLI 工具。尝试像 `Use 'foo-cli-tool --help' to learn about foo tool, then use it to solve A, B, C.` 这样的提示。

连接 MCP 服务器

Tip:

运行 `claude mcp add` 来连接外部工具，如 Notion、Figma 或你的数据库。

使用 [MCP servers](#)，你可以要求 Claude 从问题跟踪器实现功能、查询数据库、分析监控数据、集成来自 Figma 的设计并自动化工作流。

设置 hooks

Tip:

使用 hooks 来处理必须每次发生且没有例外的操作。

[Hooks](#) 在 Claude 工作流中的特定点自动运行脚本。与 `CLAUDE.md` 指令不同，hooks 是确定性的，保证操作发生。

Claude 可以为你编写 hooks。尝试像“编写一个在每次文件编辑后运行 eslint 的 hook”或“编写一个阻止写入迁移文件夹的 hook”这样的提示。运行 `/hooks` 进行交互式配置，或直接编辑 `.claude/settings.json`。

创建 skills

Tip:

在 `.claude/skills/` 中创建 `SKILL.md` 文件，为 Claude 提供域知识和可重用工作流。

[Skills](#) 使用特定于你的项目、团队或域的信息扩展 Claude 的知识。Claude 在相关时自动应用它们，或者你可以使用 `/skill-name` 直接调用它们。

通过向 `.claude/skills/` 添加带有 `SKILL.md` 的目录来创建 skill：

```
---
name: api-conventions
description: REST API design conventions for our services
---

## API Conventions

- Use kebab-case for URL paths
- Use camelCase for JSON properties
- Always include pagination for list endpoints
- Version APIs in the URL path (/v1/, /v2/)
```

Skills 也可以定义你直接调用的可重复工作流：


```

---
name: fix-issue
description: Fix a GitHub issue
disable-model-invocation: true
---

Analyze and fix the GitHub issue: $ARGUMENTS.

1. Use `gh issue view` to get the issue details
2. Understand the problem described in the issue
3. Search the codebase for relevant files
4. Implement the necessary changes to fix the issue
5. Write and run tests to verify the fix
6. Ensure code passes linting and type checking
7. Create a descriptive commit message
8. Push and create a PR

```

运行 `/fix-issue 1234` 来调用它。对于具有你想手动触发的副作用的工作流，使用 `disable-model-invocation: true`。

创建自定义 subagents

Tip:

在 `.claude/agents/` 中定义专门的助手，Claude 可以委托给它们来处理隔离的任务。

[Subagents](#) 在自己的 context 中运行，拥有自己的一组允许的工具。它们对于读取许多文件或需要专门关注而不会使你的主对话混乱的任务很有用。

```

---
name: security-reviewer
description: Reviews code for security vulnerabilities
tools: Read, Grep, Glob, Bash
model: opus
---

You are a senior security engineer. Review code for:
- Injection vulnerabilities (SQL, XSS, command injection)
- Authentication and authorization flaws
- Secrets or credentials in code
- Insecure data handling

Provide specific line references and suggested fixes.

```

明确告诉 Claude 使用 subagents：“使用 subagent 来审查此代码的安全问题。”

安装 plugins

Tip:

运行 `/plugin` 来浏览市场。Plugins 添加 skills、工具和集成，无需配置。

[Plugins](#) 将 skills、hooks、subagents 和 MCP 服务器捆绑到来自社区和 Anthropic 的单个可安装单元中。如果你使用类型化语言，安装 [代码智能 plugin](#) 来为 Claude 提供精确的符号导航和编辑后的自动错误检测。

有关在 skills、subagents、hooks 和 MCP 之间选择的指导，请参阅 [扩展 Claude Code](#)。

有效沟通

你与 Claude Code 沟通的方式显著影响结果的质量。

提出代码库问题

Tip:

问 Claude 你会问资深工程师的问题。

当加入新代码库时，使用 Claude Code 进行学习和探索。你可以问 Claude 你会问另一个工程师的相同类型的问题：

- 日志如何工作？
- 我如何创建新的 API 端点？
- `foo.rs` 第 134 行的 `async move { ... }` 做什么？
- `CustomerOnboardingFlowImpl` 处理哪些边界情况？
- 为什么这段代码在第 333 行调用 `foo()` 而不是 `bar()` ？

以这种方式使用 Claude Code 是一个有效的入职工作流，改进了加入时间并减少了对其他工程师的负担。无需特殊提示：直接提问。

让 Claude 采访你

Tip:

对于更大的功能，让 Claude 先采访你。从最小的提示开始，要求 Claude 使用 `AskUserQuestion` 工具采访你。

Claude 会问你可能还没有考虑过的东西，包括技术实现、UI/UX、边界情况和权衡。

```
I want to build [brief description]. Interview me in detail using the
AskUserQuestion tool.
```

```
Ask about technical implementation, UI/UX, edge cases, concerns, and tradeoffs.
Don't ask obvious questions, dig into the hard parts I might not have considered.
```

```
Keep interviewing until we've covered everything, then write a complete spec to
SPEC.md.
```

一旦规范完成，启动新会话来执行它。新会话有干净的 context，完全专注于实现，你有一个书面规范可以参考。

管理你的会话

对话是持久的和可逆的。利用这一点！

尽早且经常改正方向

Tip:

一旦你注意到 Claude 偏离轨道，立即改正它。

最好的结果来自紧密的反馈循环。虽然 Claude 有时会在第一次尝试时完美地解决问题，但快速改正它通常会更快地产生更好的解决方案。

- **Esc**：使用 **Esc** 键在中途停止 Claude。Context 被保留，所以你可以重定向。
- **Esc + Esc** 或 **/rewind**：按 **Esc** 两次或运行 **/rewind** 来打开 rewind 菜单并恢复之前的对话和代码状态，或从选定的消息进行总结。
- **"撤销那个"**：让 Claude 恢复其更改。
- **/clear**：在不相关的任务之间重置 context。长会话与无关的 context 可能会降低性能。

如果你在一个会话中对同一问题改正了 Claude 两次以上，context 就充满了失败的方法。运行 **/clear** 并使用更具体的提示重新开始，该提示包含你学到的东西。干净的会话与更好的提示几乎总是优于长会话与累积的改正。

积极管理 context

Tip:

在不相关的任务之间频繁运行 **/clear** 来重置 context。

当你接近 context 限制时，Claude Code 会自动压缩对话历史，这保留了重要的代码和决策，同时释放空间。

在长会话中，Claude 的 context window 可能会充满无关的对话、文件内容和命令。这可能会降低性能，有时会分散 Claude 的注意力。

- 在任务之间频繁使用 **/clear** 来完全重置 context window
- 当自动压缩触发时，Claude 总结最重要的东西，包括代码模式、文件状态和关键决策
- 为了更多控制，运行 **/compact <instructions>**，如 **/compact Focus on the API changes**
- 要仅压缩对话的一部分，使用 **Esc + Esc** 或 **/rewind**，选择消息检查点，并选择 **从这里总结**。这会压缩从该点开始的消息，同时保持早期 context 完整。

- 在 CLAUDE.md 中使用像 "When compacting, always preserve the full list of modified files and any test commands" 这样的指令来自定义压缩行为，以确保关键 context 在总结中存活
- 对于不需要留在 context 中的快速问题，使用 `/btw`。答案出现在可关闭的覆盖层中，永远不会进入对话历史，所以你可以检查细节而不增加 context。

使用 subagents 进行调查

Tip:

使用 "use subagents to investigate X" 委托研究。它们在单独的 context 中探索，为实现保持你的主对话干净。

由于 context 是你的基本约束，subagents 是可用的最强大的工具之一。当 Claude 研究代码库时，它读取许多文件，所有这些都消耗你的 context。Subagents 在单独的 context windows 中运行并报告摘要：

```
Use subagents to investigate how our authentication system handles token refresh, and whether we have any existing OAuth utilities I should reuse.
```

subagent 探索代码库、读取相关文件并报告发现，所有这些都不会使你的主对话混乱。

你也可以在 Claude 实现某些东西后使用 subagents 进行验证：

```
use a subagent to review this code for edge cases
```

使用检查点进行 Rewind

Tip:

Claude 进行的每个操作都会创建一个检查点。你可以将对话、代码或两者恢复到任何之前的检查点。

Claude 在更改前自动检查点。双击 `Escape` 或运行 `/rewind` 来打开 rewind 菜单。你可以仅恢复对话、仅恢复代码、恢复两者或从选定的消息进行总结。有关详细信息，请参阅 [Checkpointing](#)。

与其仔细规划每一步，你可以告诉 Claude 尝试一些冒险的事情。如果不起作用，rewind 并尝试不同的方法。检查点在会话中持续，所以你可以关闭你的终端并稍后仍然 rewind。

Warning:

检查点仅跟踪 Claude 进行的更改，不跟踪外部进程。这不是 git 的替代品。

恢复对话

Tip:

运行 `claude --continue` 来继续你离开的地方，或 `--resume` 来从最近的会话中选择。

Claude Code 在本地保存对话。当任务跨越多个会话时，你不必重新解释 context：

```
claude --continue    # Resume the most recent conversation
claude --resume      # Select from recent conversations
```

使用 `/rename` 给会话起描述性名称，如 `"oauth-migration"` 或 `"debugging-memory-leak"`，以便你稍后可以找到它们。像对待分支一样对待会话：不同的工作流可以有单独的、持久的 context。

自动化和扩展

一旦你对一个 Claude 有效，通过并行会话、非交互模式和扇出模式来增加你的输出。

到目前为止，一切都假设一个人、一个 Claude 和一个对话。但 Claude Code 水平扩展。本部分中的技术展示了你如何能做更多。

运行非交互模式

Tip:

在 CI、pre-commit hooks 或脚本中使用 `claude -p "prompt"`。添加 `--output-format stream-json` 用于流式 JSON 输出。

使用 `claude -p "your prompt"`，你可以非交互地运行 Claude，不需要会话。非交互模式是你将 Claude 集成到 CI 管道、pre-commit hooks 或任何自动化工作流中的方式。输出格式让你以编程方式解析结果：纯文本、JSON 或流式 JSON。

```
## One-off queries
claude -p "Explain what this project does"

## Structured output for scripts
claude -p "List all API endpoints" --output-format json

## Streaming for real-time processing
claude -p "Analyze this log file" --output-format stream-json
```

运行多个 Claude 会话

Tip:
并行运行多个 Claude 会话以加快开发、运行隔离的实验或启动复杂的工作流。

有三种主要方式来运行并行会话：

- [Claude Code 桌面应用](#)：以视觉方式管理多个本地会话。每个会话获得自己的隔离 worktree。
- [Claude Code 在网络上](#)：在 Anthropic 的安全云基础设施中的隔离 VM 上运行。
- [Agent teams](#)：具有共享任务、消息和团队主管的多个会话的自动协调。

除了并行化工作，多个会话启用了质量关注的工作流。新鲜的 context 改进了代码审查，因为 Claude 不会偏向于它刚刚编写的代码。

例如，使用 Writer/Reviewer 模式：

会话 A (Writer)	会话 B (Reviewer)
为我们的 API 端点实现速率限制器	
	审查 @src/middleware/rateLimiter.ts 中的速率限制器实现。查找边界情况、竞态条件和与我们现有中间件模式的一致性。
这是审查反馈：[会话 B 输出]。解决这些问题。	

你可以用测试做类似的事情：让一个 Claude 编写测试，然后另一个编写代码来通过它们。

跨文件扇出

Tip:

循环遍历任务，为每个调用 `claude -p`。使用 `--allowedTools` 来限定批量操作的权限。

对于大型迁移或分析，你可以跨许多并行 Claude 调用分配工作：

Step 1: 生成任务列表

让 Claude 列出所有需要迁移的文件（例如，`list all 2,000 Python files that need migrating`）

Step 2: 编写脚本来循环遍历列表

```
for file in $(cat files.txt); do
  claude -p "Migrate $file from React to Vue. Return OK or FAIL." \
    --allowedTools "Edit,Bash(git commit *)"
done
```

Step 3: 在几个文件上测试，然后大规模运行

根据前 2-3 个文件出错的情况精化你的提示，然后在完整集合上运行。`--allowedTools` 标志限制 Claude 能做什么，这在你无人值守运行时很重要。

你也可以将 Claude 集成到现有的数据/处理管道中：

```
claude -p "<your prompt>" --output-format json | your_command
```

在开发期间使用 `--verbose` 进行调试，在生产中关闭它。

避免常见失败模式

这些是常见的错误。尽早识别它们可以节省时间：

- **厨房水槽会话。** 你从一个任务开始，然后问 Claude 一些不相关的东西，然后回到第一个任务。Context 充满了无关的信息。> **修复：**在不相关的任务之间 `/clear`。
- **一次又一次地改正。** Claude 做错了什么，你改正它，它仍然是错的，你再改正。Context 被失败的方法污染。> **修复：**在两次失败的改正后，`/clear` 并编写一个更好的初始提示，包含你学到的东西。

- **过度指定的 CLAUDE.md**。如果你的 CLAUDE.md 太长，Claude 会忽略一半，因为重要的规则在噪音中丢失。> **修复**：无情地修剪。如果 Claude 已经在没有指令的情况下正确地做某事，删除它或将其转换为 hook。
- **信任然后验证的差距**。Claude 产生一个看起来合理的实现，但不处理边界情况。> **修复**：始终提供验证（测试、脚本、屏幕截图）。如果你不能验证它，不要发布它。
- **无限探索**。你要求 Claude “调查”某些东西而不限定范围。Claude 读取数百个文件，填充 context。> **修复**：狭隘地限定调查或使用 subagents，以便探索不会消耗你的主 context。

培养你的直觉

本指南中的模式不是一成不变的。它们是通常效果很好的起点，但可能不是每种情况的最优选择。

有时你_应该_让 context 累积，因为你深入一个复杂的问题，历史很有价值。有时你应该跳过规划，让 Claude 弄清楚，因为任务是探索性的。有时模糊的提示正是你想要的，因为你想看看 Claude 如何解释问题，然后再限制它。

注意什么有效。当 Claude 产生很好的输出时，注意你做了什么：提示结构、你提供的 context、你所在的模式。当 Claude 遇到困难时，问为什么。Context 太嘈杂了吗？提示太模糊了吗？任务对于一次通过来说太大了吗？

随着时间的推移，你会培养没有指南能捕捉的直觉。你会知道何时具体，何时开放，何时规划，何时探索，何时清除 context，何时让它累积。

相关资源

- [Claude Code 如何工作](#)：代理循环、工具和 context 管理
- [扩展 Claude Code](#)：skills、hooks、MCP、subagents 和 plugins
- [常见工作流](#)：调试、测试、PR 等的分步配方
- [CLAUDE.md](#)：存储项目约定和持久 context

常见工作流程

使用 Claude Code 探索代码库、修复错误、重构、测试和其他日常任务的分步指南。

本页涵盖日常开发的实用工作流程：探索陌生代码、调试、重构、编写测试、创建 PR 和管理会话。每个部分都包含示例提示，您可以根据自己的项目进行调整。有关更高级的模式和提示，请参阅[最佳实践](#)。

理解新的代码库

获取代码库的快速概览

假设您刚加入一个新项目，需要快速了解其结构。

Step 1: 导航到项目根目录

```
cd /path/to/project
```

Step 2: 启动 Claude Code

```
claude
```

Step 3: 请求高级概览

```
give me an overview of this codebase
```

Step 4: 深入了解特定组件

```
explain the main architecture patterns used here
```

```
what are the key data models?
```

```
how is authentication handled?
```

Tip:

提示:

- 从广泛的问题开始，然后缩小到特定领域
- 询问项目中使用的编码约定和模式
- 请求项目特定术语的词汇表

查找相关代码

假设您需要定位与特定功能相关的代码。

Step 1: 要求 Claude 查找相关文件

```
find the files that handle user authentication
```

Step 2: 获取有关组件如何交互的上下文

```
how do these authentication files work together?
```

Step 3: 理解执行流程

```
trace the login process from front-end to database
```

Tip:

提示:

- 明确说明您要查找的内容
- 使用项目中的领域语言
- 为您的语言安装[代码智能插件](#)，以便 Claude 能够精确地进行”转到定义”和”查找引用”导航

高效修复错误

假设您遇到了错误消息，需要找到并修复其来源。

Step 1: 与 Claude 分享错误

```
I'm seeing an error when I run npm test
```

Step 2: 请求修复建议

```
suggest a few ways to fix the @ts-ignore in user.ts
```

Step 3: 应用修复

```
update user.ts to add the null check you suggested
```

Tip:

提示:

- 告诉 Claude 重现问题的命令并获取堆栈跟踪
- 提及重现错误的任何步骤
- 让 Claude 知道错误是间歇性的还是持续的

重构代码

假设您需要更新旧代码以使用现代模式和实践。

Step 1: 识别用于重构的遗留代码

```
find deprecated API usage in our codebase
```

Step 2: 获取重构建议

```
suggest how to refactor utils.js to use modern JavaScript features
```

Step 3: 安全地应用更改

```
refactor utils.js to use ES2024 features while maintaining the same behavior
```

Step 4: 验证重构

```
run tests for the refactored code
```

Tip:

提示：

- 要求 Claude 解释现代方法的优势
- 请求在需要时保持向后兼容性的更改
- 以小的、可测试的增量进行重构

使用专门的 subagents

假设您想使用专门的 AI subagents 来更有效地处理特定任务。

Step 1: 查看可用的 subagents

```
/agents
```

这显示所有可用的 subagents 并让您创建新的。

Step 2: 自动使用 subagents

Claude Code 自动将适当的任务委派给专门的 subagents：

```
review my recent code changes for security issues
```

```
run all tests and fix any failures
```

Step 3: 明确请求特定的 subagents

```
use the code-reviewer subagent to check the auth module
```

```
have the debugger subagent investigate why users can't log in
```

Step 4: 为您的工作流创建自定义 subagents

```
/agents
```

然后选择” Create New subagent” 并按照提示定义：

- 描述 subagent 目的的唯一标识符（例如 `code-reviewer`、`api-designer`）。
- Claude 何时应该使用此代理
- 它可以访问哪些工具
- 描述代理角色和行为的系统提示

Tip:

提示：

- 在 `.claude/agents/` 中创建项目特定的 subagents 以供团队共享
- 使用描述性的 `description` 字段来启用自动委派
- 限制工具访问权限为每个 subagent 实际需要的内容
- 查看[subagents 文档](#)了解详细示例

使用 Plan Mode 进行安全的代码分析

Plan Mode 指示 Claude 通过使用只读操作分析代码库来创建计划，非常适合探索代码库、规划复杂更改或安全地审查代码。在 Plan Mode 中，Claude 使用 `AskUserQuestion` 来收集需求并在提出计划之前澄清您的目标。

何时使用 Plan Mode

- **多步骤实现**：当您的功能需要对许多文件进行编辑时
- **代码探索**：当您想在更改任何内容之前彻底研究代码库时
- **交互式开发**：当您想与 Claude 迭代方向时

如何使用 Plan Mode

在会话期间打开 Plan Mode

您可以在会话期间使用 **Shift+Tab** 循环切换权限模式来切换到 Plan Mode。

如果您处于 Normal Mode，**Shift+Tab** 首先切换到 Auto-Accept Mode，在终端底部显示 `>> accept edits on`。随后的 **Shift+Tab** 将切换到 Plan Mode，显示 `plan mode on`。

在 Plan Mode 中启动新会话

要在 Plan Mode 中启动新会话，请使用 `--permission-mode plan` 标志：

```
claude --permission-mode plan
```

在 Plan Mode 中运行“无头”查询

您也可以使用 `-p` 在 Plan Mode 中直接运行查询（即在“[无头模式](#)”中）：

```
claude --permission-mode plan -p "Analyze the authentication system and suggest improvements"
```

示例：规划复杂的重构

```
claude --permission-mode plan
```

```
I need to refactor our authentication system to use OAuth2. Create a detailed migration plan.
```

Claude 分析当前实现并创建全面的计划。通过后续问题进行细化：

```
What about backward compatibility?
```

```
How should we handle database migration?
```

Tip:

按 `Ctrl+G` 在默认文本编辑器中打开计划，您可以在 Claude 继续之前直接编辑它。

将 Plan Mode 配置为默认值

```
// .claude/settings.json
{
  "permissions": {
    "defaultMode": "plan"
  }
}
```

有关更多配置选项，请参阅[设置文档](#)。

使用测试

假设您需要为未覆盖的代码添加测试。

Step 1: 识别未测试的代码

```
find functions in NotificationsService.swift that are not covered by tests
```

Step 2: 生成测试框架

```
add tests for the notification service
```

Step 3: 添加有意义的测试用例

```
add test cases for edge conditions in the notification service
```

Step 4: 运行并验证测试

```
run the new tests and fix any failures
```

Claude 可以生成遵循您项目现有模式和约定的测试。请求测试时，请明确说明您想验证的行为。Claude 检查您现有的测试文件以匹配已在使用的样式、框架和断言模式。

为了获得全面的覆盖，要求 Claude 识别您可能遗漏的边界情况。Claude 可以分析您的代码路径并建议测试错误条件、边界值和容易被忽视的意外输入。

创建拉取请求

您可以通过直接要求 Claude 创建拉取请求（“create a pr for my changes”），或逐步指导 Claude：

Step 1: 总结您的更改

```
summarize the changes I've made to the authentication module
```

Step 2: 生成拉取请求

```
create a pr
```

Step 3: 审查和细化

```
enhance the PR description with more context about the security improvements
```

当您使用 `gh pr create` 创建 PR 时，会话会自动链接到该 PR。您可以稍后使用 `claude --from-pr <number>` 恢复它。

Tip:

在提交前审查 Claude 生成的 PR，并要求 Claude 突出显示潜在的风险或注意事项。

处理文档

假设您需要为代码添加或更新文档。

Step 1: 识别未记录的代码

```
find functions without proper JSDoc comments in the auth module
```

Step 2: 生成文档

```
add JSDoc comments to the undocumented functions in auth.js
```

Step 3: 审查和增强

improve the generated documentation with more context and examples

Step 4: 验证文档

check if the documentation follows our project standards

Tip:

提示:

- 指定您想要的文档样式（JSDoc、docstrings 等）
- 请求文档中的示例
- 请求公共 API、接口和复杂逻辑的文档

使用图像

假设您需要在代码库中使用图像，并希望 Claude 帮助分析图像内容。

Step 1: 将图像添加到对话中

您可以使用以下任何方法：

1. 将图像拖放到 Claude Code 窗口中
2. 复制图像并使用 ctrl+v 将其粘贴到 CLI 中（不要使用 cmd+v）
3. 向 Claude 提供图像路径。例如，“Analyze this image: /path/to/your/image.png”

Step 2: 要求 Claude 分析图像

What does this image show?

Describe the UI elements in this screenshot

Are there any problematic elements in this diagram?

Step 3: 使用图像获取上下文

Here's a screenshot of the error. What's causing it?

This is our current database schema. How should we modify it for the new feature?

Step 4: 从视觉内容获取代码建议

Generate CSS to match this design mockup

What HTML structure would recreate this component?

Tip:

提示:

- 当文本描述不清楚或繁琐时使用图像
- 包含错误、UI 设计或图表的屏幕截图以获得更好的上下文
- 您可以在对话中使用多个图像
- 图像分析适用于图表、屏幕截图、模型等
- 当 Claude 引用图像时（例如 `[Image #1]`），`Cmd+Click` (Mac) 或 `Ctrl+Click` (Windows/Linux) 链接以在默认查看器中打开图像

引用文件和目录

使用 @ 快速包含文件或目录，无需等待 Claude 读取它们。

Step 1: 引用单个文件

Explain the logic in @src/utils/auth.js

这在对话中包含文件的完整内容。

Step 2: 引用目录

What's the structure of @src/components?

这提供了带有文件信息的目录列表。

Step 3: 引用 MCP 资源

```
Show me the data from @github:repos/owner/repo/issues
```

这使用 `@server:resource` 格式从连接的 MCP 服务器获取数据。有关详细信息，请参阅 [MCP 资源](#)。

Tip:

提示：

- 文件路径可以是相对的或绝对的
- `@` 文件引用在文件的目录和父目录中添加 `CLAUDE.md` 到上下文
- 目录引用显示文件列表，而不是内容
- 您可以在单个消息中引用多个文件（例如，“`@file1.js and @file2.js`”）

使用扩展思考（Thinking Mode）

[扩展思考](#) 默认启用，为 Claude 提供空间在响应前逐步推理复杂问题。此推理在详细模式中可见，您可以使用 `Ctrl+0` 切换。

此外，Opus 4.6 引入了自适应推理：不是固定的思考令牌预算，而是模型根据您的[努力级别](#)设置动态分配思考。扩展思考和自适应推理一起工作，让您能够控制 Claude 在响应前的推理深度。

扩展思考对于复杂的架构决策、具有挑战性的错误、多步骤实现规划和评估不同方法之间的权衡特别有价值。

Note:

“think”、“think hard”和“think more”等短语被解释为常规提示指令，不分配思考令牌。

配置 Thinking Mode

思考默认启用，但您可以调整或禁用它。

范围	如何配置	详情
努力级别	在 <code>/model</code> 中调整或设置 <code>CLAUDE_CODE_EFFORT_LEVEL</code>	控制 Opus 4.6 和 Sonnet 4.6 的思考深度：低、中、高。请参阅 调整努力级别
<code>ultrathink</code> 关键字	在提示中的任何地方包含 “ultrathink”	在 Opus 4.6 和 Sonnet 4.6 上为该轮设置努力为高。对于需要深度推理的一次性任务很有用，无需永久更改您的努力设置
切换快捷键	按 <code>Option+T</code> (macOS) 或 <code>Alt+T</code> (Windows/Linux)	为当前会话切换思考开/关（所有模型）。可能需要 终端配置 来启用 Option 键快捷键
全局默认值	使用 <code>/config</code> 切换 Thinking Mode	在所有项目中设置默认值（所有模型）。保存为 <code>~/ .claude/ settings.json</code> 中的 <code>alwaysThinkingEnabled</code>
限制令牌预算	设置 <code>MAX_THINKING_TOKENS</code> 环境变量	将思考预算限制为特定数量的令牌（在 Opus 4.6 上被忽略，除非设置为 0）。示例： <code>export MAX_THINKING_TOKENS=10000</code>

要查看 Claude 的思考过程，按 `Ctrl+0` 切换详细模式，并查看显示为灰色斜体文本的内部推理。

扩展思考如何工作

扩展思考控制 Claude 在响应前执行多少内部推理。更多思考提供更多空间来探索解决方案、分析边界情况和自我纠正错误。

使用 Opus 4.6，思考使用自适应推理：模型根据您选择的[努力级别](#)（低、中、高）动态分配思考令牌。这是调整速度和推理深度之间权衡的推荐方式。

使用其他模型，思考使用固定预算，最多 31,999 个令牌来自您的输出预算。您可以使用 `MAX_THINKING_TOKENS` 环境变量限制此，或通过 `/config` 或 `Option+T` / `Alt+T` 切换完全禁用思考。

`MAX_THINKING_TOKENS` 在 Opus 4.6 和 Sonnet 4.6 上被忽略，因为自适应推理控制思考深度。一个例外：设置 `MAX_THINKING_TOKENS=0` 仍然在任何模型上完全禁用思考。要禁用自适应思考并恢复到固定思考预算，请设置 `CLAUDE_CODE_DISABLE_ADAPTIVE_THINKING=1`。请参阅[环境变量](#)。

Warning:

您需要为所有使用的思考令牌付费，即使 Claude 4 模型显示总结的思考

恢复以前的对话

启动 Claude Code 时，您可以恢复以前的会话：

- `claude --continue` 继续当前目录中最近的对话
- `claude --resume` 打开对话选择器或按名称恢复
- `claude --from-pr 123` 恢复链接到特定拉取请求的会话

从活跃会话内，使用 `/resume` 切换到不同的对话。

会话按项目目录存储。`/resume` 选择器显示来自同一 git 存储库的会话，包括 worktrees。

命名您的会话

给会话起描述性名称以便稍后找到它们。这是在处理多个任务或功能时的最佳实践。

Step 1: 命名当前会话

在会话期间使用 `/rename` 给它一个易记的名称：

```
/rename auth-refactor
```

您也可以从选择器重命名任何会话：运行 `/resume`，导航到会话，然后按 `R`。

Step 2: 稍后按名称恢复

从命令行：

```
claude --resume auth-refactor
```

或从活跃会话内：

```
/resume auth-refactor
```

使用会话选择器

`/resume` 命令（或 `claude --resume` 不带参数）打开具有以下功能的交互式会话选择器：

选择器中的键盘快捷键：

快捷键	操作
↑ / ↓	在会话之间导航
→ / ←	展开或折叠分组的会话
Enter	选择并恢复突出显示的会话
P	预览会话内容
R	重命名突出显示的会话
/	搜索以过滤会话
A	在当前目录和所有项目之间切换
B	过滤到来自当前 git 分支的会话
Esc	退出选择器或搜索模式

会话组织：

选择器显示带有有用元数据的会话：

- 会话名称或初始提示
- 自上次活动以来经过的时间
- 消息计数
- Git 分支（如果适用）

分叉的会话（使用 `/rewind` 或 `--fork-session` 创建）在其根会话下分组，使查找相关对话更容易。

Tip:

提示：

- **尽早命名会话：** 在开始处理不同任务时使用 `/rename` —— 稍后找到 “payment-integration” 比 “explain this function” 容易得多

- 使用 `--continue` 快速访问当前目录中最近的对话
- 当您知道需要哪个会话时使用 `--resume session-name`
- 当您需要浏览和选择时使用 `--resume`（不带名称）
- 对于脚本，使用 `claude --continue --print "prompt"` 以非交互模式恢复
- 在选择器中按 **P** 在恢复前预览会话
- 恢复的对话以与原始对话相同的模型和配置开始

工作原理：

1. **对话存储**：所有对话都自动保存在本地，包含完整的历史记录
2. **消息反序列化**：恢复时，整个消息历史记录被恢复以保持上下文
3. **工具状态**：来自以前对话的工具使用和结果被保留
4. **上下文恢复**：对话以所有以前的上下文完整恢复

使用 Git worktrees 运行并行 Claude Code 会话

同时处理多个任务时，您需要每个 Claude 会话都有自己的代码库副本，以便更改不会冲突。Git worktrees 通过创建单独的工作目录来解决这个问题，每个目录都有自己的文件和分支，同时共享相同的存储库历史记录和远程连接。这意味着您可以让 Claude 在一个 worktree 中处理功能，同时在另一个 worktree 中修复错误，而不会相互干扰。

使用 `--worktree (-w)` 标志创建隔离的 worktree 并在其中启动 Claude。您传递的值成为 worktree 目录名称和分支名称：

```
## 在名为 "feature-auth" 的 worktree 中启动 Claude
## 创建 .claude/worktrees/feature-auth/ 和新分支
claude --worktree feature-auth

## 在单独的 worktree 中启动另一个会话
claude --worktree bugfix-123
```

如果您省略名称，Claude 会自动生成一个随机名称：

```
## 自动生成名称如 "bright-running-fox"
claude --worktree
```

Worktrees 在 `<repo>/claude/worktrees/<name>` 创建，并从默认远程分支分支。worktree 分支命名为 `worktree-<name>`。

您也可以在会话期间要求 Claude “work in a worktree” 或 “start a worktree”，它会自动创建一个。

Subagent worktrees

Subagents 也可以使用 worktree 隔离来并行工作而不会冲突。要求 Claude “use worktrees for your agents” 或在 [自定义 subagent](#) 中通过在代理的 frontmatter 中添加 `isolation: worktree` 来配置它。每个 subagent 获得自己的 worktree，当 subagent 完成而没有更改时自动清理。

Worktree 清理

当您退出 worktree 会话时，Claude 根据您是否进行了更改来处理清理：

- **无更改**：worktree 及其分支自动删除
- **存在更改或提交**：Claude 提示您保留或删除 worktree。保留会保留目录和分支，以便您稍后可以返回。删除会删除 worktree 目录及其分支，丢弃所有未提交的更改和提交

要在 Claude 会话外清理 worktrees，请使用[手动 worktree 管理](#)。

Tip:

将 `.claude/worktrees/` 添加到您的 `.gitignore` 以防止 worktree 内容在主存储库中显示为未跟踪的文件。

手动管理 worktrees

为了更好地控制 worktree 位置和分支配置，直接使用 Git 创建 worktrees。当您需要检出特定的现有分支或将 worktree 放在存储库外时，这很有用。

```
## 使用新分支创建 worktree
git worktree add ../project-feature-a -b feature-a

## 使用现有分支创建 worktree
git worktree add ../project-bugfix bugfix-123

## 在 worktree 中启动 Claude
cd ../project-feature-a && claude

## 完成时清理
git worktree list
git worktree remove ../project-feature-a
```

在[官方 Git worktree 文档](#)中了解更多。

Tip:

记住根据您的项目设置在每个新 worktree 中初始化您的开发环境。根据您的堆栈，这可能包括运行依赖项安装（`npm install`、`yarn`）、设置虚拟环境或遵循您的项目标准设置过程。

非 git 版本控制

Worktree 隔离默认使用 git。对于其他版本控制系统如 SVN、Perforce 或 Mercurial，配置 [WorktreeCreate](#) 和 [WorktreeRemove hooks](#) 以提供自定义 worktree 创建和清理逻辑。配置后，这些 hooks 在您使用 `--worktree` 时替换默认的 git 行为。

对于具有共享任务和消息的并行会话的自动协调，请参阅[代理团队](#)。

在 Claude 需要您的注意时获得通知

当您启动长时间运行的任务并切换到另一个窗口时，您可以设置桌面通知，以便在 Claude 完成或需要您的输入时了解。这使用 `Notification hook 事件`，每当 Claude 等待权限、空闲并准备好新提示或完成身份验证时触发。

Step 1: 打开 hooks 菜单

输入 `/hooks` 并从事件列表中选择 `Notification`。

Step 2: 配置匹配器

选择 **+ Match all (no filter)** 以在所有通知类型上触发。要仅针对特定事件通知，选择 **+ Add new matcher...** 并输入以下值之一：

匹配器	触发时机
<code>permission_prompt</code>	Claude 需要您批准工具使用
<code>idle_prompt</code>	Claude 完成并等待您的下一个提示
<code>auth_success</code>	身份验证完成
<code>elicitation_dialog</code>	Claude 在问您一个问题

Step 3: 添加您的通知命令

选择 **+ Add new hook...** 并输入您的操作系统的命令：

macOS

使用 `osascript` 通过 AppleScript 触发本机 macOS 通知：

```
osascript -e 'display notification "Claude Code needs your attention" with title "Claude Code"'
```

Linux

使用 `notify-send`，它在大多数带有通知守护程序的 Linux 桌面上预装：

```
notify-send 'Claude Code' 'Claude Code needs your attention'
```

Windows (PowerShell)

使用 PowerShell 通过 .NET 的 Windows Forms 显示本机消息框：

```
powershell.exe -Command  
"[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms');  
[System.Windows.Forms.MessageBox]::Show('Claude Code needs your attention',  
'Claude Code')"
```

Step 4: 保存到用户设置

选择 **User settings** 以在所有项目中应用通知。

有关完整的演练和 JSON 配置示例，请参阅[使用 hooks 自动化 workflow](#)。有关完整的事件架构和通知类型，请参阅[通知参考](#)。

将 Claude 用作 unix 风格的实用程序

将 Claude 添加到您的验证过程

假设您想使用 Claude Code 作为 linter 或代码审查者。

将 Claude 添加到您的构建脚本：

```
// package.json
{
  ...
  "scripts": {
    ...
    "lint:claude": "claude -p 'you are a linter. please look at the changes vs. main and report any issues related to typos. report the filename and line number on one line, and a description of the issue on the second line. do not return any other text.'"
  }
}
```

Tip:

提示：

- 在您的 CI/CD 管道中使用 Claude 进行自动代码审查
- 自定义提示以检查与您的项目相关的特定问题
- 考虑为不同类型的验证创建多个脚本

管道进出

假设您想将数据管道传入 Claude，并获取结构化格式的数据。

通过 Claude 管道数据：

```
cat build-error.txt | claude -p 'concisely explain the root cause of this build error' > output.txt
```

Tip:

提示:

- 使用管道将 Claude 集成到现有的 shell 脚本中
- 与其他 Unix 工具结合以实现强大的工作流
- 考虑使用 `--output-format` 获取结构化输出

控制输出格式

假设您需要 Claude 的输出采用特定格式，特别是在将 Claude Code 集成到脚本或其他工具时。

Step 1: 使用文本格式（默认）

```
cat data.txt | claude -p 'summarize this data' --output-format text > summary.txt
```

这输出仅 Claude 的纯文本响应（默认行为）。

Step 2: 使用 JSON 格式

```
cat code.py | claude -p 'analyze this code for bugs' --output-format json > analysis.json
```

这输出包含元数据（包括成本和持续时间）的消息的 JSON 数组。

Step 3: 使用流式 JSON 格式

```
cat log.txt | claude -p 'parse this log file for errors' --output-format stream-json
```

这在 Claude 处理请求时实时输出一系列 JSON 对象。每条消息都是有效的 JSON 对象，但如果连接，整个输出不是有效的 JSON。

Tip:

提示:

- 对于简单集成（您只需要 Claude 的响应），使用 `--output-format text`
- 当您需要完整的对话日志时使用 `--output-format json`

- 对于每个对话轮次的实时输出，使用 `--output-format stream-json`

询问 Claude 关于其功能

Claude 内置访问其文档，可以回答关于其自身功能和限制的问题。

示例问题

can Claude Code create pull requests?

how does Claude Code handle permissions?

what skills are available?

how do I use MCP with Claude Code?

how do I configure Claude Code for Amazon Bedrock?

what are the limitations of Claude Code?

Note:

Claude 根据文档提供对这些问题的答案。有关可执行示例和实际演示，请参阅上面的特定工作流程部分。

Tip:

提示：

- Claude 始终可以访问最新的 Claude Code 文档，无论您使用的版本如何
- 提出具体问题以获得详细答案
- Claude 可以解释复杂的功能，如 MCP 集成、企业配置和高级工作流

后续步骤

- [最佳实践](#) 从 Claude Code 获得最大收益的模式
- [Claude Code 如何工作](#) 理解代理循环和上下文管理
- [扩展 Claude Code](#) 添加 skills、hooks、MCP、subagents 和 plugins
- [参考实现](#) 克隆我们的开发容器参考实现

Part 3: Commands & Reference

CLI 参考

Claude Code 命令行界面的完整参考，包括命令和标志。

CLI 命令

您可以使用这些命令启动会话、管道内容、恢复对话和管理更新：

命令	描述	示例
<code>claude</code>	启动交互式会话	<code>claude</code>
<code>claude "query"</code>	使用初始提示启动交互式会话	<code>claude "explain this project"</code>
<code>claude -p "query"</code>	通过 SDK 查询，然后退出	<code>claude -p "explain this function"</code>
<code>cat file claude -p "query"</code>	处理管道内容	<code>cat logs.txt claude -p "explain"</code>
<code>claude -c</code>	继续当前目录中最近的对话	<code>claude -c</code>
<code>claude -c -p "query"</code>	通过 SDK 继续	<code>claude -c -p "Check for type errors"</code>
<code>claude -r "<session>" "query"</code>	按 ID 或名称恢复会话	<code>claude -r "auth-refactor" "Finish this PR"</code>
<code>claude update</code>	更新到最新版本	<code>claude update</code>
<code>claude auth login</code>	登录您的 Anthropic 账户。使用 <code>--email</code> 预填充您的电子邮件地址，使用 <code>--sso</code> 强制 SSO 身份验证	<code>claude auth login --email user@example.com --sso</code>
<code>claude auth logout</code>	从您的 Anthropic 账户登出	<code>claude auth logout</code>

命令	描述	示例
<code>claude auth status</code>	以 JSON 格式显示身份验证状态。使用 <code>--text</code> 获得人类可读的输出。如果已登录则以代码 0 退出，如果未登录则以代码 1 退出	<code>claude auth status</code>
<code>claude agents</code>	列出所有已配置的 subagents ，按来源分组	<code>claude agents</code>
<code>claude mcp</code>	配置 Model Context Protocol (MCP) 服务器	请参阅 Claude Code MCP 文档 。
<code>claude remote-control</code>	启动 Remote Control 会话 以从 Claude.ai 或 Claude 应用控制 Claude Code，同时在本地上运行。请参阅 Remote Control 了解标志	<code>claude remote-control</code>

CLI 标志

使用这些命令行标志自定义 Claude Code 的行为：

标志	描述	示例
<code>--add-dir</code>	添加额外的工作目录供 Claude 访问（验证每个路径是否存在为目录）	<code>claude --add-dir ../apps ../lib</code>
<code>--agent</code>	为当前会话指定代理（覆盖 <code>agent</code> 设置）	<code>claude --agent my-custom-agent</code>
<code>--agents</code>	通过 JSON 动态定义自定义 subagents （请参阅下面的格式）	<code>claude --agents '{ "reviewer": { "description": "Review", "prompt": "You are a code reviewer" } }'</code>
<code>--allow-dangerously-skip-permissions</code>	启用权限绕过作为选项，而不立即激活它。允许与 <code>--permission-mode</code> 组合（谨慎使用）	<code>claude --permission-mode plan --allow-dangerously-skip-permissions</code>

标志	描述	示例
<code>--allowedTools</code>	无需提示权限即可执行的工具。请参阅 权限规则语法 了解模式匹配。要限制哪些工具可用，请改用 <code>--tools</code>	<code>"Bash(git log *)"</code> <code>"Bash(git diff *)"</code> <code>"Read"</code>
<code>--append-system-prompt</code>	将自定义文本附加到默认系统提示的末尾	<code>claude --append-system-prompt "Always use TypeScript"</code>
<code>--append-system-prompt-file</code>	从文件加载额外的系统提示文本并附加到默认提示	<code>claude --append-system-prompt-file ./extra-rules.txt</code>
<code>--betas</code>	要包含在 API 请求中的 Beta 标头（仅限 API 密钥用户）	<code>claude --betas interleaved-thinking</code>
<code>--chrome</code>	启用 Chrome 浏览器集成 以进行网络自动化和测试	<code>claude --chrome</code>
<code>--continue, -c</code>	加载当前目录中最近的对话	<code>claude --continue</code>
<code>--dangerously-skip-permissions</code>	跳过所有权限提示（谨慎使用）	<code>claude --dangerously-skip-permissions</code>
<code>--debug</code>	启用调试模式，可选类别过滤（例如， <code>"api,hooks"</code> 或 <code>"!statsig,!file"</code> ）	<code>claude --debug "api,mcp"</code>
<code>--disable-slash-commands</code>	为此会话禁用所有 skills 和命令	<code>claude --disable-slash-commands</code>
<code>--disallowedTools</code>	从模型的上下文中删除的工具，无法使用	<code>"Bash(git log *)"</code> <code>"Bash(git diff *)"</code> <code>"Edit"</code>
<code>--fallback-model</code>	当默认模型过载时启用自动回退到指定模型（仅打印模式）	<code>claude -p --fallback-model sonnet "query"</code>
<code>--fork-session</code>	恢复时，创建新的会话 ID 而不是重用原始 ID（与 <code>--resume</code> 或 <code>--continue</code> 一起使用）	<code>claude --resume abc123 --fork-session</code>

标志	描述	示例
<code>--from-pr</code>	恢复链接到特定 GitHub PR 的会话。接受 PR 号或 URL。通过 <code>gh pr create</code> 创建时会自动链接会话	<code>claude --from-pr 123</code>
<code>--ide</code>	如果恰好有一个有效的 IDE 可用，则在启动时自动连接到 IDE	<code>claude --ide</code>
<code>--init</code>	运行初始化 hooks 并启动交互模式	<code>claude --init</code>
<code>--init-only</code>	运行初始化 hooks 并退出（无交互式会话）	<code>claude --init-only</code>
<code>--include-partial-messages</code>	在输出中包含部分流事件（需要 <code>--print</code> 和 <code>--output-format=stream-json</code> ）	<code>claude -p --output-format stream-json --include-partial-messages "query"</code>
<code>--input-format</code>	为打印模式指定输入格式（选项： <code>text</code> 、 <code>stream-json</code> ）	<code>claude -p --output-format json --input-format stream-json</code>
<code>--json-schema</code>	在代理完成其工作流后获得与 JSON Schema 匹配的验证 JSON 输出（仅打印模式，请参阅 结构化输出 ）	<code>claude -p --json-schema</code> <code>'{"type":"object","properties":{"...}}'</code> <code>"query"</code>
<code>--maintenance</code>	运行维护 hooks 并退出	<code>claude --maintenance</code>
<code>--max-budget-usd</code>	API 调用前停止的最大美元金额（仅打印模式）	<code>claude -p --max-budget-usd 5.00</code> <code>"query"</code>
<code>--max-turns</code>	限制代理转数（仅打印模式）。达到限制时以错误退出。默认无限制	<code>claude -p --max-turns 3</code> <code>"query"</code>
<code>--mcp-config</code>	从 JSON 文件或字符串加载 MCP 服务器（以空格分隔）	<code>claude --mcp-config ./mcp.json</code>
<code>--model</code>	为当前会话设置模型，使用最新模型的别名（ <code>sonnet</code> 或 <code>opus</code> ）或模型的完整名称	<code>claude --model</code> <code>claude-sonnet-4-6</code>

标志	描述	示例
<code>--no-chrome</code>	为此会话禁用 Chrome 浏览器集成	<code>claude --no-chrome</code>
<code>--no-session-persistence</code>	禁用会话持久化，以便会话不保存到磁盘且无法恢复（仅打印模式）	<code>claude -p --no-session-persistence "query"</code>
<code>--output-format</code>	为打印模式指定输出格式（选项： <code>text</code> 、 <code>json</code> 、 <code>stream-json</code> ）	<code>claude -p "query" --output-format json</code>
<code>--permission-mode</code>	以指定的 权限模式 开始	<code>claude --permission-mode plan</code>
<code>--permission-prompt-tool</code>	指定 MCP 工具以在非交互模式下处理权限提示	<code>claude -p --permission-prompt-tool mcp_auth_tool "query"</code>
<code>--plugin-dir</code>	仅为该会话从目录加载插件（可重复）	<code>claude --plugin-dir ./my-plugins</code>
<code>--print</code> , <code>-p</code>	打印响应而不进入交互模式（请参阅 Agent SDK 文档 了解程序化使用详情）	<code>claude -p "query"</code>
<code>--remote</code>	在 claude.ai 上创建新的 网络会话 ，提供任务描述	<code>claude --remote "Fix the login bug"</code>
<code>--resume</code> , <code>-r</code>	按 ID 或名称恢复特定会话，或显示交互式选择器以选择会话	<code>claude --resume auth-refactor</code>
<code>--session-id</code>	为对话使用特定的会话 ID（必须是有效的 UUID）	<code>claude --session-id "550e8400-e29b-41d4-a716-446655440000"</code>
<code>--setting-sources</code>	要加载的设置源的逗号分隔列表（ <code>user</code> 、 <code>project</code> 、 <code>local</code> ）	<code>claude --setting-sources user,project</code>
<code>--settings</code>	设置 JSON 文件的路径或要加载的 JSON 字符串	<code>claude --settings ./settings.json</code>
<code>--strict-mcp-config</code>	仅使用 <code>--mcp-config</code> 中的 MCP 服务器，忽略所有其他 MCP 配置	<code>claude --strict-mcp-config --mcp-config ./mcp.json</code>

标志	描述	示例
<code>--system-prompt</code>	用自定义文本替换整个系统提示	<code>claude --system-prompt "You are a Python expert"</code>
<code>--system-prompt-file</code>	从文件加载系统提示，替换默认提示	<code>claude --system-prompt-file ./custom-prompt.txt</code>
<code>--teleport</code>	在本地终端中恢复 网络会话	<code>claude --teleport</code>
<code>--teammate-mode</code>	设置 agent team 队友的显示方式： <code>auto</code> （默认）、 <code>in-process</code> 或 <code>tmux</code> 。请参阅 设置 agent teams	<code>claude --teammate-mode in-process</code>
<code>--tools</code>	限制 Claude 可以使用的内置工具。 使用 <code>" "</code> 禁用所有， <code>"default"</code> 表示全部，或工具名称如 <code>"Bash,Edit,Read"</code>	<code>claude --tools "Bash,Edit,Read"</code>
<code>--verbose</code>	启用详细日志记录，显示完整的逐轮输出	<code>claude --verbose</code>
<code>--version</code> , <code>-v</code>	输出版本号	<code>claude -v</code>
<code>--worktree</code> , <code>-w</code>	在隔离的 git worktree 中启动 Claude，位于 <code><repo>/ .claude/ worktrees/<name></code> 。如果未给出名称，则自动生成	<code>claude -w feature-auth</code>

Tip:

`--output-format json` 标志对于脚本和自动化特别有用，允许您以编程方式解析 Claude 的响应。

Agents 标志格式

`--agents` 标志接受一个 JSON 对象，该对象定义一个或多个自定义 subagents。每个 subagent 需要一个唯一的名称（作为键）和一个具有以下字段的定义对象：

字段	必需	描述
<code>description</code>	是	何时应调用 subagent 的自然语言描述
<code>prompt</code>	是	指导 subagent 行为的系统提示
<code>tools</code>	否	subagent 可以使用的特定工具数组，例如 <code>["Read", "Edit", "Bash"]</code> 。如果省略，则继承所有工具。支持 <code>Agent(agent_type)</code> 语法
<code>disallowedTools</code>	否	为此 subagent 明确拒绝的工具名称数组
<code>model</code>	否	要使用的模型别名： <code>sonnet</code> 、 <code>opus</code> 、 <code>haiku</code> 或 <code>inherit</code> 。如果省略，默认为 <code>inherit</code>
<code>skills</code>	否	要预加载到 subagent 上下文中的 <code>skill</code> 名称数组
<code>mcpServers</code>	否	此 subagent 的 <code>MCP servers</code> 数组。每个条目是服务器名称字符串或 <code>{name: config}</code> 对象
<code>maxTurns</code>	否	subagent 停止前的最大代理转数

示例：

```
claude --agents '{
  "code-reviewer": {
    "description": "Expert code reviewer. Use proactively after code changes.",
    "prompt": "You are a senior code reviewer. Focus on code quality, security,
and best practices.",
    "tools": ["Read", "Grep", "Glob", "Bash"],
    "model": "sonnet"
  },
  "debugger": {
    "description": "Debugging specialist for errors and test failures.",
    "prompt": "You are an expert debugger. Analyze errors, identify root causes,
and provide fixes."
  }
}'
```

有关创建和使用 subagents 的更多详情，请参阅 [subagents 文档](#)。

系统提示标志

Claude Code 提供四个标志用于自定义系统提示。所有四个都在交互和非交互模式中工作。

标志	行为	用例
<code>--system-prompt</code>	替换整个默认提示	完全控制 Claude 的行为和指令
<code>--system-prompt-file</code>	替换为文件内容	从文件加载提示以实现可重复性和版本控制
<code>--append-system-prompt</code>	附加到默认提示	添加特定指令，同时保持默认 Claude Code 行为
<code>--append-system-prompt-file</code>	附加文件内容到默认提示	从文件加载额外指令，同时保持默认值

何时使用每个：

- `--system-prompt`：当您完全需要控制 Claude 的系统提示时使用。这会删除所有默认 Claude Code 指令，为您提供一个空白板。

```
claude --system-prompt "You are a Python expert who only writes type-annotated code"
```

- **--system-prompt-file**：当您想从文件加载自定义提示时使用，对于团队一致性或版本控制的提示模板很有用。

```
claude --system-prompt-file ./prompts/code-review.txt
```

- **--append-system-prompt**：当您想添加特定指令同时保持 Claude Code 的默认功能完整时使用。这是大多数用例的最安全选项。

```
claude --append-system-prompt "Always use TypeScript and include JSDoc comments"
```

- **--append-system-prompt-file**：当您想从文件附加指令同时保持 Claude Code 的默认值时使用。对于版本控制的添加很有用。

```
claude --append-system-prompt-file ./prompts/style-rules.txt
```

--system-prompt 和 **--system-prompt-file** 互斥。附加标志可以与任一替换标志一起使用。

对于大多数用例，建议使用 **--append-system-prompt** 或 **--append-system-prompt-file**，因为它们保留 Claude Code 的内置功能，同时添加您的自定义要求。仅当您需要完全控制系统提示时才使用 **--system-prompt** 或 **--system-prompt-file**。

另请参阅

- [Chrome 扩展](#) - 浏览器自动化和网络测试
- [交互模式](#) - 快捷键、输入模式和交互功能
- [快速入门指南](#) - Claude Code 入门
- [常见工作流](#) - 高级工作流和模式
- [设置](#) - 配置选项
- [Agent SDK 文档](#) - 程序化使用和集成

自定义快捷键

使用快捷键配置文件在 Claude Code 中自定义快捷键。

Note:

可自定义的快捷键需要 Claude Code v2.1.18 或更高版本。使用 `claude --version` 检查您的版本。

Claude Code 支持可自定义的快捷键。运行 `/keybindings` 来创建或打开位于 `~/.claude/keybindings.json` 的配置文件。

配置文件

快捷键配置文件是一个包含 `bindings` 数组的对象。每个块指定一个上下文和一个按键映射到操作的映射。

Note:

快捷键文件的更改会自动检测并应用，无需重启 Claude Code。

字段	描述
<code>\$schema</code>	可选的 JSON Schema URL，用于编辑器自动完成
<code>\$docs</code>	可选的文档 URL
<code>bindings</code>	按上下文分组的绑定块数组

此示例将 `Ctrl+E` 绑定到在聊天上下文中打开外部编辑器，并取消绑定 `Ctrl+U`：

```
{
  "$schema": "https://www.schemastore.org/claude-code-keybindings.json",
  "$docs": "https://code.claude.com/docs/en/keybindings",
  "bindings": [
    {
      "context": "Chat",
      "bindings": {
        "ctrl+e": "chat:externalEditor",
        "ctrl+u": null
      }
    }
  ]
}
```

上下文

每个绑定块指定一个上下文，其中绑定适用：

上下文	描述
Global	在应用程序的任何地方应用
Chat	主聊天输入区域
Autocomplete	自动完成菜单已打开
Settings	设置菜单（仅限 Escape 关闭）
Confirmation	权限和确认对话框
Tabs	选项卡导航组件
Help	帮助菜单可见
Transcript	记录查看器
HistorySearch	历史搜索模式（Ctrl+R）
Task	后台任务正在运行
ThemePicker	主题选择器对话框
Attachments	图像/附件栏导航

上下文	描述
Footer	页脚指示器导航（任务、团队、差异）
MessageSelector	回溯和总结对话框消息选择
DiffDialog	差异查看器导航
ModelPicker	模型选择器工作量级别
Select	通用选择/列表组件
Plugin	插件对话框（浏览、发现、管理）

可用操作

操作遵循 `namespace:action` 格式，例如 `chat:submit` 发送消息或 `app:toggleTodos` 显示任务列表。每个上下文都有特定的可用操作。

应用程序操作

在 `Global` 上下文中可用的操作：

操作	默认	描述
<code>app:interrupt</code>	Ctrl+C	取消当前操作
<code>app:exit</code>	Ctrl+D	退出 Claude Code
<code>app:toggleTodos</code>	Ctrl+T	切换任务列表可见性
<code>app:toggleTranscript</code>	Ctrl+O	切换详细记录

历史操作

用于导航命令历史的操作：

操作	默认	描述
<code>history:search</code>	Ctrl+R	打开历史搜索
<code>history:previous</code>	Up	上一个历史项
<code>history:next</code>	Down	下一个历史项

聊天操作

在 `Chat` 上下文中可用的操作：

操作	默认	描述
<code>chat:cancel</code>	Escape	取消当前输入
<code>chat:cycleMode</code>	Shift+Tab*	循环权限模式
<code>chat:modelPicker</code>	Cmd+P / Meta+P	打开模型选择器
<code>chat:thinkingToggle</code>	Cmd+T / Meta+T	切换扩展思考
<code>chat:submit</code>	Enter	提交消息
<code>chat:undo</code>	Ctrl+_	撤销上一个操作
<code>chat:externalEditor</code>	Ctrl+G	在外部编辑器中打开
<code>chat:stash</code>	Ctrl+S	隐藏当前提示
<code>chat:imagePaste</code>	Ctrl+V (Windows 上为 Alt+V)	粘贴图像

*在沒有 VT 模式的 Windows 上 (Node <24.2.0/<22.17.0 , Bun <1.2.23) , 默认为 Meta+M 。

自动完成操作

在 `Autocomplete` 上下文中可用的操作：

操作	默认	描述
<code>autocomplete:accept</code>	Tab	接受建议
<code>autocomplete:dismiss</code>	Escape	关闭菜单
<code>autocomplete:previous</code>	Up	上一个建议
<code>autocomplete:next</code>	Down	下一个建议

确认操作

在 `Confirmation` 上下文中可用的操作：

操作	默认	描述
<code>confirm:yes</code>	Y, Enter	确认操作

操作	默认	描述
<code>confirm:no</code>	N, Escape	拒绝操作
<code>confirm:previous</code>	Up	上一个选项
<code>confirm:next</code>	Down	下一个选项
<code>confirm:nextField</code>	Tab	下一个字段
<code>confirm:previousField</code>	(未绑定)	上一个字段
<code>confirm:cycleMode</code>	Shift+Tab	循环权限模式
<code>confirm:toggleExplanation</code>	Ctrl+E	切换权限说明

权限操作

在 `Confirmation` 上下文中可用的权限对话框操作：

操作	默认	描述
<code>permission:toggleDebug</code>	Ctrl+D	切换权限调试信息

记录操作

在 `Transcript` 上下文中可用的操作：

操作	默认	描述
<code>transcript:toggleShowAll</code>	Ctrl+E	切换显示所有内容
<code>transcript:exit</code>	Ctrl+C, Escape	退出记录查看

历史搜索操作

在 `HistorySearch` 上下文中可用的操作：

操作	默认	描述
<code>historySearch:next</code>	Ctrl+R	下一个匹配项
<code>historySearch:accept</code>	Escape, Tab	接受选择
<code>historySearch:cancel</code>	Ctrl+C	取消搜索
<code>historySearch:execute</code>	Enter	执行选定的命令

任务操作

在 `Task` 上下文中可用的操作：

操作	默认	描述
<code>task:background</code>	Ctrl+B	后台当前任务

主题操作

在 `ThemePicker` 上下文中可用的操作：

操作	默认	描述
<code>theme:toggleSyntaxHighlighting</code>	Ctrl+T	切换语法高亮

帮助操作

在 `Help` 上下文中可用的操作：

操作	默认	描述
<code>help:dismiss</code>	Escape	关闭帮助菜单

选项卡操作

在 `Tabs` 上下文中可用的操作：

操作	默认	描述
<code>tabs:next</code>	Tab, Right	下一个选项卡
<code>tabs:previous</code>	Shift+Tab, Left	上一个选项卡

附件操作

在 `Attachments` 上下文中可用的操作：

操作	默认	描述
<code>attachments:next</code>	Right	下一个附件
<code>attachments:previous</code>	Left	上一个附件
<code>attachments:remove</code>	Backspace, Delete	删除选定的附件
<code>attachments:exit</code>	Down, Escape	退出附件栏

页脚操作

在 `Footer` 上下文中可用的操作：

操作	默认	描述
<code>footer:next</code>	Right	下一个页脚项
<code>footer:previous</code>	Left	上一个页脚项
<code>footer:openSelected</code>	Enter	打开选定的页脚项
<code>footer:clearSelection</code>	Escape	清除页脚选择

消息选择器操作

在 `MessageSelector` 上下文中可用的操作：

操作	默认	描述
<code>messageSelector:up</code>	Up, K, Ctrl+P	在列表中向上移动
<code>messageSelector:down</code>	Down, J, Ctrl+N	在列表中向下移动
<code>messageSelector:top</code>	Ctrl+Up, Shift+Up, Meta+Up, Shift+K	跳到顶部
<code>messageSelector:bottom</code>	Ctrl+Down, Shift+Down, Meta+Down, Shift+J	跳到底部
<code>messageSelector:select</code>	Enter	选择消息

差异操作

在 `DiffDialog` 上下文中可用的操作：

操作	默认	描述
<code>diff:dismiss</code>	Escape	关闭差异查看器
<code>diff:previousSource</code>	Left	上一个差异源
<code>diff:nextSource</code>	Right	下一个差异源
<code>diff:previousFile</code>	Up	差异中的上一个文件

操作	默认	描述
<code>diff:nextFile</code>	Down	差异中的下一个文件
<code>diff:viewDetails</code>	Enter	查看差异详情
<code>diff:back</code>	(特定于上下文)	在差异查看器中返回

模型选择器操作

在 `ModelPicker` 上下文中可用的操作：

操作	默认	描述
<code>modelPicker:decreaseEffort</code>	Left	降低工作量级别
<code>modelPicker:increaseEffort</code>	Right	提高工作量级别

选择操作

在 `Select` 上下文中可用的操作：

操作	默认	描述
<code>select:next</code>	Down, J, Ctrl+N	下一个选项
<code>select:previous</code>	Up, K, Ctrl+P	上一个选项
<code>select:accept</code>	Enter	接受选择
<code>select:cancel</code>	Escape	取消选择

插件操作

在 `Plugin` 上下文中可用的操作：

操作	默认	描述
<code>plugin:toggle</code>	Space	切换插件选择
<code>plugin:install</code>	I	安装选定的插件

设置操作

在 `Settings` 上下文中可用的操作：

操作	默认	描述
<code>settings:search</code>	/	进入搜索模式
<code>settings.retry</code>	R	重试加载使用数据（出错时）

按键语法

修饰符

使用修饰符键和 `+` 分隔符：

- `ctrl` 或 `control` - Control 键
- `alt`、`opt` 或 `option` - Alt/Option 键
- `shift` - Shift 键
- `meta`、`cmd` 或 `command` - Meta/Command 键

例如：

<code>ctrl+k</code>	单个键加修饰符
<code>shift+tab</code>	Shift + Tab
<code>meta+p</code>	Command/Meta + P
<code>ctrl+shift+c</code>	多个修饰符

大写字母

独立的大写字母意味着 Shift。例如，`K` 等同于 `shift+k`。这对于 vim 风格的绑定很有用，其中大写和小写键有不同的含义。

带有修饰符的大写字母（例如 `ctrl+K`）被视为风格上的，不意味着 Shift — `ctrl+K` 与 `ctrl+k` 相同。

和弦

和弦是由空格分隔的按键序列：

`ctrl+k ctrl+s` 按 Ctrl+K，释放，然后按 Ctrl+S

特殊键

- `escape` 或 `esc` - Escape 键
- `enter` 或 `return` - Enter 键

- `tab` - Tab 键
- `space` - 空格键
- `up` 、 `down` 、 `left` 、 `right` - 箭头键
- `backspace` 、 `delete` - 删除键

取消绑定默认快捷键

将操作设置为 `null` 以取消绑定默认快捷键：

```
{
  "bindings": [
    {
      "context": "Chat",
      "bindings": {
        "ctrl+s": null
      }
    }
  ]
}
```

保留的快捷键

这些快捷键无法重新绑定：

快捷键	原因
Ctrl+C	硬编码的中断/取消
Ctrl+D	硬编码的退出

终端冲突

某些快捷键可能与终端多路复用器冲突：

快捷键	冲突
Ctrl+B	tmux 前缀（按两次发送）
Ctrl+A	GNU screen 前缀
Ctrl+Z	Unix 进程暂停（SIGTSTP）

Vim 模式交互

启用 vim 模式（`/vim`）时，快捷键和 vim 模式独立运行：

- **Vim 模式**在文本输入级别处理输入（光标移动、模式、动作）
- **快捷键**在组件级别处理操作（切换待办事项、提交等）
- vim 模式中的 Escape 键从 INSERT 切换到 NORMAL 模式；它不触发 `chat:cancel`
- 大多数 Ctrl+key 快捷键通过 vim 模式传递到快捷键系统
- 在 vim NORMAL 模式中，`?` 显示帮助菜单（vim 行为）

验证

Claude Code 验证您的快捷键并显示以下警告：

- 解析错误（无效的 JSON 或结构）
- 无效的上下文名称
- 保留快捷键冲突
- 终端多路复用器冲突
- 同一上下文中的重复绑定

运行 `/doctor` 查看任何快捷键警告。

Part 4: Configuration

配置权限

通过细粒度权限规则、模式和托管策略来控制 Claude Code 可以访问和执行的操
作。

Claude Code 支持细粒度权限，因此您可以精确指定代理允许执行的操作和不允许执行的操作。权限设置可以检入版本控制并分发给组织中的所有开发人员，也可以由个别开发人员自定义。

权限系统

Claude Code 使用分层权限系统来平衡功能和安全性：

工具类型	示例	需要批准	“是，不再询问” 行为
只读	文件读取、Grep	否	不适用
Bash 命令	Shell 执行	是	每个项目目录和命令永久有效
文件修改	Edit/Write 文件	是	直到会话结束

管理权限

您可以使用 `/permissions` 查看和管理 Claude Code 的工具权限。此 UI 列出所有权限规则
和它们来自的 `settings.json` 文件。

- **Allow** 规则让 Claude Code 使用指定的工具而无需手动批准。
- **Ask** 规则在 Claude Code 尝试使用指定工具时提示确认。
- **Deny** 规则防止 Claude Code 使用指定的工具。

规则按顺序评估：**deny -> ask -> allow**。第一个匹配的规则获胜，因此 deny 规则始终优先。

权限模式

Claude Code 支持多种权限模式来控制工具的批准方式。在您的[设置文件](#)中设置 `defaultMode`：

模式	描述
<code>default</code>	标准行为：在首次使用每个工具时提示权限
<code>acceptEdits</code>	自动接受会话的文件编辑权限
<code>plan</code>	Plan Mode：Claude 可以分析但不能修改文件或执行命令
<code>dontAsk</code>	自动拒绝工具，除非通过 <code>/permissions</code> 或 <code>permissions.allow</code> 规则预先批准
<code>bypassPermissions</code>	跳过所有权限提示（需要安全环境，请参见下面的警告）

Warning:

`bypassPermissions` 模式禁用所有权限检查。仅在隔离环境（如容器或虚拟机）中使用此模式，其中 Claude Code 无法造成损害。管理员可以通过在[托管设置](#)中将 `disableBypassPermissionsMode` 设置为 `"disable"` 来防止此模式。

权限规则语法

权限规则遵循格式 `Tool` 或 `Tool(specifier)`。

匹配工具的所有使用

要匹配工具的所有使用，只需使用工具名称而不带括号：

规则	效果
<code>Bash</code>	匹配所有 Bash 命令
<code>WebFetch</code>	匹配所有网络获取请求
<code>Read</code>	匹配所有文件读取

`Bash(*)` 等同于 `Bash` 并匹配所有 Bash 命令。

使用说明符进行细粒度控制

在括号中添加说明符以匹配特定的工具使用：

规则	效果
<code>Bash(npm run build)</code>	匹配确切的命令 <code>npm run build</code>

规则	效果
<code>Read(./env)</code>	匹配读取当前目录中的 <code>env</code> 文件
<code>WebFetch(domain:example.com)</code>	匹配对 <code>example.com</code> 的获取请求

通配符模式

Bash 规则支持带有 `*` 的 glob 模式。通配符可以出现在命令中的任何位置。此配置允许 `npm` 和 `git commit` 命令，同时阻止 `git push`：

```
{
  "permissions": {
    "allow": [
      "Bash(npm run *)",
      "Bash(git commit *)",
      "Bash(git * main)",
      "Bash(* --version)",
      "Bash(* --help *)"
    ],
    "deny": [
      "Bash(git push *)"
    ]
  }
}
```

`*` 前的空格很重要：`Bash(ls *)` 匹配 `ls -la` 但不匹配 `ls of`，而 `Bash(ls*)` 匹配两者。旧版 `:*` 后缀语法等同于 `*` 但已弃用。

工具特定的权限规则

Bash

Bash 权限规则支持带有 `*` 的通配符匹配。通配符可以出现在命令中的任何位置，包括开头、中间或结尾：

- `Bash(npm run build)` 匹配确切的 Bash 命令 `npm run build`
- `Bash(npm run test *)` 匹配以 `npm run test` 开头的 Bash 命令
- `Bash(npm *)` 匹配任何以 `npm` 开头的命令
- `Bash(* install)` 匹配任何以 `install` 结尾的命令

- `Bash(git * main)` 匹配 `git checkout main`、`git merge main` 等命令

当 `*` 出现在末尾且前面有空格时（如 `Bash(ls *)`），它强制执行单词边界，要求前缀后跟空格或字符串结尾。例如，`Bash(ls *)` 匹配 `ls -la` 但不匹配 `ls of`。相比之下，`Bash(ls*)` 没有空格匹配 `ls -la` 和 `ls of` 两者，因为没有单词边界约束。

Tip:

Claude Code 知道 shell 运算符（如 `&&`），因此前缀匹配规则如 `Bash(safe-cmd *)` 不会给它权限运行命令 `safe-cmd && other-cmd`。

Warning:

尝试约束命令参数的 Bash 权限模式很脆弱。例如，`Bash(curl http://github.com/ *)` 旨在将 curl 限制为 GitHub URL，但不会匹配以下变体：

- URL 前的选项：`curl -X GET http://github.com/...`
- 不同的协议：`curl https://github.com/...`
- 重定向：`curl -L http://bit.ly/xyz`（重定向到 github）
- 变量：`URL=http://github.com && curl $URL`
- 额外空格：`curl http://github.com`

为了更可靠的 URL 过滤，请考虑：

- **限制 Bash 网络工具：**使用 deny 规则阻止 `curl`、`wget` 和类似命令，然后对允许的域使用带有 `WebFetch(domain:github.com)` 权限的 WebFetch 工具
- **使用 PreToolUse hooks：**实现一个 hook 来验证 Bash 命令中的 URL 并阻止不允许的域
- 通过 CLAUDE.md 指示 Claude Code 关于您允许的 curl 模式

请注意，仅使用 WebFetch 不会阻止网络访问。如果允许 Bash，Claude 仍然可以使用 `curl`、`wget` 或其他工具来访问任何 URL。

Read 和 Edit

`Edit` 规则适用于所有编辑文件的内置工具。Claude 尽力将 `Read` 规则应用于所有读取文件的内置工具，如 `Grep` 和 `Glob`。

`Read` 和 `Edit` 规则都遵循 [gitleaks](#) 规范，具有四种不同的模式类型：

模式	含义	示例	匹配
<code>//path</code>	来自文件系统根目录的绝对路径	<code>Read(//Users/alice/secrets/**)</code>	<code>/Users/alice/secrets/**</code>

模式	含义	示例	匹配
<code>~/path</code>	来自主目录的路径	<code>Read(~/Documents/ *.pdf)</code>	<code>/Users/alice/ Documents/*.pdf</code>
<code>/path</code>	相对于项目根目录的路径	<code>Edit(/src/**/ *.ts)</code>	<code><project root>/ src/**/*.ts</code>
<code>path</code> 或 <code>./path</code>	相对于当前目录的路径	<code>Read(*.env)</code>	<code><cwd>/*.env</code>

Warning:

像 `/Users/alice/file` 这样的模式不是绝对路径。它相对于项目根目录。对于绝对路径，使用 `//Users/alice/file`。

示例:

- `Edit(/docs/**)`: 编辑 `<project>/docs/` 中的文件 (不是 `/docs/` 也不是 `<project>/claude/docs/`)
- `Read(~/.zshrc)`: 读取您主目录的 `.zshrc`
- `Edit(/tmp/scratch.txt)`: 编辑绝对路径 `/tmp/scratch.txt`
- `Read(src/**)`: 从 `<current-directory>/src/` 读取

Note:

在 gitignore 模式中, `*` 匹配单个目录中的文件, 而 `**` 递归匹配目录。要允许所有文件访问, 只需使用工具名称而不带括号: `Read`、`Edit` 或 `Write`。

WebFetch

- `WebFetch(domain:example.com)` 匹配对 `example.com` 的获取请求

MCP

- `mcp_puppeteer` 匹配由 `puppeteer` 服务器提供的任何工具 (在 Claude Code 中配置的名称)
- `mcp_puppeteer_*` 通配符语法, 也匹配来自 `puppeteer` 服务器的所有工具
- `mcp_puppeteer_puppeteer_navigate` 匹配由 `puppeteer` 服务器提供的 `puppeteer_navigate` 工具

Agent (subagents)

使用 `Agent(AgentName)` 规则来控制 Claude 可以使用哪些 [subagents](#)：

- `Agent(Explore)` 匹配 Explore subagent
- `Agent(PPlan)` 匹配 Plan subagent
- `Agent(my-custom-agent)` 匹配名为 `my-custom-agent` 的自定义 subagent

将这些规则添加到您的设置中的 `deny` 数组，或使用 `--disallowedTools` CLI 标志来禁用特定代理。要禁用 Explore 代理：

```
{
  "permissions": {
    "deny": ["Agent(Explore)"]
  }
}
```

使用 hooks 扩展权限

[Claude Code hooks](#) 提供了一种方法来注册自定义 shell 命令以在运行时执行权限评估。当 Claude Code 进行工具调用时，PreToolUse hooks 在权限系统之前运行，hook 输出可以确定是否批准或拒绝工具调用来代替权限系统。

工作目录

默认情况下，Claude 可以访问启动它的目录中的文件。您可以扩展此访问：

- **启动期间**：使用 `--add-dir <path>` CLI 参数
- **会话期间**：使用 `/add-dir` 命令
- **持久配置**：添加到[设置文件](#)中的 `additionalDirectories`

其他目录中的文件遵循与原始工作目录相同的权限规则：它们变为可读的而无需提示，文件编辑权限遵循当前权限模式。

权限如何与沙箱交互

权限和[沙箱](#)是互补的安全层：

- **权限控制** Claude Code 可以使用哪些工具以及它可以访问哪些文件或域。它们适用于所有工具（Bash、Read、Edit、WebFetch、MCP 和其他）。
- **沙箱**提供 OS 级别的强制执行，限制 Bash 工具的文件系统和网络访问。它仅适用于 Bash 命令及其子进程。

使用两者进行深度防御：

- 权限 `deny` 规则阻止 Claude 甚至尝试访问受限资源
- 沙箱限制防止 Bash 命令到达定义边界之外的资源，即使提示注入绕过 Claude 的决策制定
- 沙箱中的文件系统限制使用 Read 和 Edit `deny` 规则，而不是单独的沙箱配置
- 网络限制结合 WebFetch 权限规则与沙箱的 `allowedDomains` 列表

托管设置

对于需要对 Claude Code 配置进行集中控制的组织，管理员可以部署无法被用户或项目设置覆盖的托管设置。这些策略设置遵循与常规设置文件相同的格式，可以通过 MDM/OS 级别策略、托管设置文件或[服务器托管设置](#)传递。有关传递机制和文件位置，请参见[设置文件](#)。

仅托管设置

某些设置仅在托管设置中有效：

设置	描述
<code>disableBypassPermissionsMode</code>	设置为 <code>"disable"</code> 以防止 <code>bypassPermissions</code> 模式和 <code>--dangerously-skip-permissions</code> 标志
<code>allowManagedPermissionRulesOnly</code>	当为 <code>true</code> 时，防止用户和项目设置定义 <code>allow</code> 、 <code>ask</code> 或 <code>deny</code> 权限规则。仅应用托管设置中的规则
<code>allowManagedHooksOnly</code>	当为 <code>true</code> 时，防止加载用户、项目和插件 hooks。仅允许托管 hooks 和 SDK hooks
<code>allowManagedMcpServersOnly</code>	当为 <code>true</code> 时，仅尊重来自托管设置的 <code>allowedMcpServers</code> 。 <code>deniedMcpServers</code> 仍然从所有来源合并。请参见 托管 MCP 配置
<code>blockedMarketplaces</code>	市场来源的黑名单。在下载前检查被阻止的来源，因此它们永远不会接触文件系统。请参见 托管市场限制

设置	描述
<code>sandbox.network.allowManagedDomainsOnly</code>	当为 <code>true</code> 时，仅尊重来自托管设置的 <code>allowedDomains</code> 和 <code>WebFetch(domain: ...)</code> allow 规则。非允许的域被自动阻止，不提示用户。被拒绝的域仍然从所有来源合并
<code>strictKnownMarketplaces</code>	控制用户可以添加哪些插件市场。请参见 托管市场限制
<code>allow_remote_sessions</code>	当为 <code>true</code> 时，允许用户启动 远程控制 和 网络会话 。默认为 <code>true</code> 。设置为 <code>false</code> 以防止远程会话访问

设置优先级

权限规则遵循与所有其他 Claude Code 设置相同的[设置优先级](#)：

1. **托管设置**：无法被任何其他级别覆盖，包括命令行参数
2. **命令行参数**：临时会话覆盖
3. **本地项目设置**（`.claude/settings.local.json`）
4. **共享项目设置**（`.claude/settings.json`）
5. **用户设置**（`~/claude/settings.json`）

如果工具在任何级别被拒绝，没有其他级别可以允许它。例如，托管设置 `deny` 无法被 `--allowedTools` 覆盖，`--disallowedTools` 可以添加超出托管设置定义的限制。

如果权限在用户设置中被允许但在项目设置中被拒绝，项目设置优先，权限被阻止。

示例配置

此[存储库](#)包括常见部署场景的启动设置配置。将这些用作起点并根据您的需要调整它们。

另请参见

- [Settings](#)：完整的配置参考，包括权限设置表
- [Sandboxing](#)：Bash 命令的 OS 级文件系统和网络隔离
- [Authentication](#)：设置用户对 Claude Code 的访问
- [Security](#)：安全保障和最佳实践
- [Hooks](#)：自动化工作流并扩展权限评估

Claude 如何记住您的项目

使用 CLAUDE.md 文件为 Claude 提供持久指令，并让 Claude 通过自动记忆功能自动积累学习内容。

每个 Claude Code 会话都从一个全新的 context window 开始。两种机制可以跨会话传递知识：

- **CLAUDE.md 文件**：您编写的指令，为 Claude 提供持久上下文
- **自动记忆**：Claude 根据您的更正和偏好自己编写的笔记

本页面涵盖以下内容：

- [编写和组织 CLAUDE.md 文件](#)
- [使用 `.claude/rules/` 将规则范围限定为特定文件类型](#)
- [配置自动记忆](#)，使 Claude 自动记笔记
- [故障排除](#)当指令未被遵循时

CLAUDE.md 与自动记忆

Claude Code 有两个互补的记忆系统。两者都在每次对话开始时加载。Claude 将它们视为上下文，而不是强制配置。您的指令越具体和简洁，Claude 遵循它们的一致性就越高。

	CLAUDE.md 文件	自动记忆
谁编写	您	Claude
包含内容	指令和规则	学习和模式
范围	项目、用户或组织	每个工作树
加载到	每个会话	每个会话（前 200 行）
用于	编码标准、工作流、项目架构	构建命令、调试见解、Claude 发现的偏好

当您想指导 Claude 的行为时，使用 CLAUDE.md 文件。自动记忆让 Claude 从您的更正中学习，无需手动操作。

subagents 也可以维护自己的自动记忆。有关详细信息，请参阅 [subagent 配置](#)。

CLAUDE.md 文件

CLAUDE.md 文件是 markdown 文件，为项目、您的个人工作流或整个组织为 Claude 提供持久指令。您以纯文本形式编写这些文件；Claude 在每个会话开始时读取它们。

选择 CLAUDE.md 文件的位置

CLAUDE.md 文件可以位于多个位置，每个位置都有不同的范围。更具体的位置优先于更广泛的位置。

范围	位置	目的	用例示例	共享对象
托管策略	• macOS : <code>/Library/Application Support/ClaudeCode/CLAUDE.md</code> • Linux 和 WSL : <code>/etc/claude-code/CLAUDE.md</code> • Windows : <code>C:\Program Files\ClaudeCode\CLAUDE.md</code>	由 IT/DevOps 管理的组织范围指令	公司编码标准、安全策略、合规要求	组织中的所有用户
项目指令	<code>./CLAUDE.md</code> 或 <code>./.claude/CLAUDE.md</code>	项目的团队共享指令	项目架构、编码标准、常见工作流	通过源代码控制的团队成员
用户指令	<code>~/.claude/CLAUDE.md</code>	所有项目的个人偏好	代码样式偏好、个人工具快捷方式	仅您（所有项目）

工作目录上方目录层次结构中的 CLAUDE.md 文件在启动时完整加载。子目录中的 CLAUDE.md 文件在 Claude 读取这些目录中的文件时按需加载。有关完整的解析顺序，请参阅 [CLAUDE.md 文件如何加载](#)。

对于大型项目，您可以使用 [项目规则](#) 将指令分解为特定主题的文件。规则允许您将指令范围限定为特定文件类型或子目录。

设置项目 CLAUDE.md

项目 CLAUDE.md 可以存储在 `./CLAUDE.md` 或 `./.claude/CLAUDE.md` 中。创建此文件并添加适用于在项目上工作的任何人的指令：构建和测试命令、编码标准、架构决策、命名约定和常见工作流。这些指令通过版本控制与您的团队共享，因此请关注项目级标准而不是个人偏好。

Tip:

运行 `/init` 自动生成起始 CLAUDE.md。Claude 分析您的代码库并创建一个包含构建命令、测试指令和它发现的项目约定的文件。如果 CLAUDE.md 已存在，`/init` 会建议改进而不是覆盖它。从那里进行细化，添加 Claude 不会自己发现的指令。

编写有效的指令

CLAUDE.md 文件在每个会话开始时加载到 context window 中，与您的对话一起消耗令牌。因为它们是上下文而不是强制配置，您编写指令的方式会影响 Claude 遵循它们的可靠性。具体、简洁、结构良好的指令效果最好。

大小：每个 CLAUDE.md 文件目标在 200 行以下。较长的文件消耗更多上下文并降低遵守度。如果您的指令变得很大，请使用 [导入](#) 或 `.claude/rules/` 文件进行拆分。

结构：使用 markdown 标题和项目符号来分组相关指令。Claude 扫描结构的方式与读者相同：有组织的部分比密集段落更容易遵循。

具体性：编写具体到足以验证的指令。例如：

- “使用 2 空格缩进”而不是“正确格式化代码”
- “在提交前运行 `npm test`”而不是“测试您的更改”
- “API 处理程序位于 `src/api/handlers/`”而不是“保持文件有组织”

一致性：如果两条规则相互矛盾，Claude 可能会任意选择一条。定期审查您的 CLAUDE.md 文件、子目录中的嵌套 CLAUDE.md 文件和 `.claude/rules/` 以删除过时或冲突的指令。在 monorepos 中，使用 `claudeMdExcludes` 跳过来自与您的工作无关的其他团队的 CLAUDE.md 文件。

导入其他文件

CLAUDE.md 文件可以使用 `@path/to/import` 语法导入其他文件。导入的文件在启动时展开并加载到上下文中，与引用它们的 CLAUDE.md 一起。

允许相对路径和绝对路径。相对路径相对于包含导入的文件解析，而不是工作目录。导入的文件可以递归导入其他文件，最大深度为五跳。

要引入 README、package.json 和工作流指南，请在 CLAUDE.md 中的任何位置使用 `@` 语法引用它们：

有关项目概述，请参阅 @README，有关此项目的可用 npm 命令，请参阅 @package.json。

```
## 其他指令
- git 工作流 @docs/git-instructions.md
```

对于您不想签入的个人偏好，从您的主目录导入文件。导入进入共享 CLAUDE.md，但它指向的文件保留在您的机器上：

```
## 个人偏好
- @~/ .claude/my-project-instructions.md
```

Warning:

Claude Code 第一次在项目中遇到外部导入时，它会显示一个批准对话框，列出文件。如果您拒绝，导入将保持禁用状态，对话框不会再次出现。

有关组织指令的更结构化方法，请参阅 `.claude/rules/`。

CLAUDE.md 文件如何加载

Claude Code 通过从当前工作目录向上遍历目录树来读取 CLAUDE.md 文件，检查沿途的每个目录。这意味着如果您在 `foo/bar/` 中运行 Claude Code，它会从 `foo/bar/CLAUDE.md` 和 `foo/CLAUDE.md` 加载指令。

Claude 还在当前工作目录下的子目录中发现 CLAUDE.md 文件。它们不是在启动时加载，而是在 Claude 读取这些子目录中的文件时包含。

如果您在大型 monorepo 中工作，其中其他团队的 CLAUDE.md 文件被拾取，请使用 `claudeMdExcludes` 跳过它们。

从其他目录加载

`--add-dir` 标志使 Claude 可以访问主工作目录外的其他目录。默认情况下，不加载这些目录中的 CLAUDE.md 文件。

要也从其他目录加载 CLAUDE.md 文件，包括 `CLAUDE.md`、`.claude/CLAUDE.md` 和 `.claude/rules/*.md`，请设置 `CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD` 环境变量：

```
CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1 claude --add-dir ../shared-config
```

使用 `.claude/rules/` 组织规则

对于较大的项目，您可以使用 `.claude/rules/` 目录将指令组织到多个文件中。这使指令保持模块化并更容易让团队维护。规则也可以 [范围限定为特定文件路径](#)，因此它们仅在 Claude 处理匹配文件时加载到上下文中，减少噪音并节省上下文空间。

Note:

规则在每个会话或打开匹配文件时加载到上下文中。对于不需要始终在上下文中的特定任务指令，请改用 [skills](#)，它仅在您调用它们或 Claude 确定它们与您的提示相关时加载。

设置规则

在项目的 `.claude/rules/` 目录中放置 markdown 文件。每个文件应涵盖一个主题，具有描述性文件名，如 `testing.md` 或 `api-design.md`。所有 `.md` 文件都被递归发现，因此您可以将规则组织到子目录中，如 `frontend/` 或 `backend/`：

```
your-project/
├── .claude/
│   ├── CLAUDE.md          # 主项目指令
│   └── rules/
│       ├── code-style.md  # 代码样式指南
│       ├── testing.md     # 测试约定
│       └── security.md    # 安全要求
```

没有 `paths` [frontmatter](#) 的规则在启动时加载，优先级与 `.claude/CLAUDE.md` 相同。

特定路径的规则

规则可以使用带有 `paths` 字段的 YAML frontmatter 范围限定为特定文件。这些条件规则仅在 Claude 处理与指定模式匹配的文件时适用。


```
---
paths:
  - "src/api/**/*.ts"
---
```

```
## API 开发规则

- 所有 API 端点必须包括输入验证
- 使用标准错误响应格式
- 包括 OpenAPI 文档注释
```

没有 `paths` 字段的规则无条件加载并适用于所有文件。路径范围规则在 Claude 读取与模式匹配的文件时触发，而不是在每次工具使用时。

在 `paths` 字段中使用 glob 模式按扩展名、目录或任何组合匹配文件：

模式	匹配
<code>**/*.ts</code>	任何目录中的所有 TypeScript 文件
<code>src/**/*.*</code>	<code>src/</code> 目录下的所有文件
<code>*.md</code>	项目根目录中的 Markdown 文件
<code>src/components/*.tsx</code>	特定目录中的 React 组件

您可以指定多个模式并使用大括号扩展在一个模式中匹配多个扩展名：

```
---
paths:
  - "src/**/*.{ts,tsx}"
  - "lib/**/*.ts"
  - "tests/**/*.test.ts"
---
```

使用符号链接跨项目共享规则

`.claude/rules/` 目录支持符号链接，因此您可以维护一组共享规则并将它们链接到多个项目中。符号链接被解析并正常加载，循环符号链接被检测并优雅处理。

此示例链接共享目录和单个文件：

```
ln -s ~/shared-claude-rules .claude/rules/shared
ln -s ~/company-standards/security.md .claude/rules/security.md
```

用户级规则

`~/.claude/rules/` 中的个人规则适用于您机器上的每个项目。使用它们来处理不是项目特定的偏好：

```
~/.claude/rules/
├─ preferences.md    # 您的个人编码偏好
└─ workflows.md     # 您的首选工作流
```

用户级规则在项目规则之前加载，给予项目规则更高的优先级。

为大型团队管理 CLAUDE.md

对于在团队中部署 Claude Code 的组织，您可以集中指令并控制加载哪些 CLAUDE.md 文件。

部署组织范围的 CLAUDE.md

组织可以部署一个集中管理的 CLAUDE.md，适用于机器上的所有用户。此文件无法通过个人设置排除。

Step 1: 在托管策略位置创建文件

- macOS：`/Library/Application Support/ClaudeCode/CLAUDE.md`
- Linux 和 WSL：`/etc/claude-code/CLAUDE.md`
- Windows：`C:\Program Files\ClaudeCode\CLAUDE.md`

Step 2: 使用您的配置管理系统部署

使用 MDM、Group Policy、Ansible 或类似工具在开发人员机器上分发文件。有关其他组织范围配置选项，请参阅 [托管设置](#)。

排除特定的 CLAUDE.md 文件

在大型 monorepos 中，祖先 CLAUDE.md 文件可能包含与您的工作无关的指令。

`claudeMdExcludes` 设置允许您按路径或 glob 模式跳过特定文件。

此示例排除顶级 CLAUDE.md 和来自父文件夹的规则目录。将其添加到 `.claude/settings.local.json` 以使排除保持本地到您的机器：

```
{
  "claudeMdExcludes": [
    "**/monorepo/CLAUDE.md",
    "/home/user/monorepo/other-team/.claude/rules/**"
  ]
}
```

模式使用 glob 语法与绝对文件路径匹配。您可以在任何 [设置层](#)：用户、项目、本地或托管策略中配置 `claudeMdExcludes`。数组跨层合并。

托管策略 CLAUDE.md 文件无法排除。这确保组织范围指令始终适用，无论个人设置如何。

自动记忆

自动记忆让 Claude 在您不编写任何内容的情况下跨会话积累知识。Claude 在工作时为自己保存笔记：构建命令、调试见解、架构笔记、代码样式偏好和工作流习惯。Claude 不会每个会话都保存内容。它根据信息在未来对话中是否有用来决定值得记住的内容。

Note:

自动记忆需要 Claude Code v2.1.59 或更高版本。使用 `claude --version` 检查您的版本。

启用或禁用自动记忆

自动记忆默认开启。要切换它，在会话中打开 `/memory` 并使用自动记忆切换，或在项目设置中设置 `autoMemoryEnabled`：

```
{
  "autoMemoryEnabled": false
}
```

要通过环境变量禁用自动记忆，请设置 `CLAUDE_CODE_DISABLE_AUTO_MEMORY=1`。

存储位置

每个项目在 `~/.claude/projects/<project>/memory/` 获得自己的记忆目录。`<project>` 路径来自 git 存储库，因此同一存储库中的所有工作树和子目录共享一个自动记忆目录。在 git 存储库外，使用项目根目录。

要将自动记忆存储在不同位置，请在用户或本地设置中设置 `autoMemoryDirectory`：

```
{
  "autoMemoryDirectory": "~/my-custom-memory-dir"
}
```

此设置从策略、本地和用户设置接受。它不从项目设置（`.claude/settings.json`）接受，以防止共享项目将自动记忆写入重定向到敏感位置。

目录包含 `MEMORY.md` 入口点和可选的主题文件：

```
~/ .claude/projects/<project>/memory/
├─ MEMORY.md           # 简洁索引，加载到每个会话
├─ debugging.md        # 调试模式的详细笔记
├─ api-conventions.md  # API 设计决策
└─ ...                 # Claude 创建的任何其他主题文件
```

`MEMORY.md` 充当记忆目录的索引。Claude 在整个会话中读取和写入此目录中的文件，使用 `MEMORY.md` 跟踪存储的内容。

自动记忆是机器本地的。同一 git 存储库中的所有工作树和子目录共享一个自动记忆目录。文件不在机器或云环境之间共享。

它如何工作

`MEMORY.md` 的前 200 行在每次对话开始时加载。第 200 行之外的内容在会话开始时不加载。Claude 通过将详细笔记移到单独的主题文件中来保持 `MEMORY.md` 简洁。

此 200 行限制仅适用于 `MEMORY.md`。CLAUDE.md 文件无论长度如何都完整加载，尽管较短的文件会产生更好的遵守度。

主题文件如 `debugging.md` 或 `patterns.md` 在启动时不加载。Claude 在需要信息时使用其标准文件工具按需读取它们。

Claude 在您的会话期间读取和写入记忆文件。当您在 Claude Code 界面中看到” Writing memory” 或” Recalled memory” 时，Claude 正在主动更新或读取 `~/ .claude/projects/<project>/memory/`。

审计和编辑您的记忆

自动记忆文件是纯 markdown，您可以随时编辑或删除。运行 `/memory` 从会话中浏览和打开记忆文件。

使用 `/memory` 查看和编辑

`/memory` 命令列出在当前会话中加载的所有 CLAUDE.md 和规则文件，让您切换自动记忆开或关，并提供打开自动记忆文件夹的链接。选择任何文件在您的编辑器中打开它。

当您要求 Claude 记住某些内容时，例如”始终使用 pnpm，而不是 npm”或”记住 API 测试需要本地 Redis 实例”，Claude 将其保存到自动记忆。要改为向 CLAUDE.md 添加指令，直接要求 Claude，例如”将其添加到 CLAUDE.md”，或通过 `/memory` 自己编辑文件。

故障排除记忆问题

这些是 CLAUDE.md 和自动记忆最常见的问题，以及调试步骤。

Claude 不遵循我的 CLAUDE.md

CLAUDE.md 是上下文，不是强制。Claude 读取它并尝试遵循它，但没有严格遵守的保证，特别是对于模糊或冲突的指令。

要调试：

- 运行 `/memory` 验证您的 CLAUDE.md 文件正在加载。如果文件未列出，Claude 看不到它。
- 检查相关 CLAUDE.md 是否在为您的会话加载的位置（请参阅 [选择 CLAUDE.md 文件的位置](#)）。
- 使指令更具体。“使用 2 空格缩进”比”格式化代码很好”效果更好。
- 查找跨 CLAUDE.md 文件的冲突指令。如果两个文件为相同行为提供不同的指导，Claude 可能会任意选择一个。

Tip:

使用 `InstructionsLoaded` hook 记录加载的确切指令文件、加载时间和原因。这对于调试特定路径规则或子目录中的延迟加载文件很有用。

我不知道自动记忆保存了什么

运行 `/memory` 并选择自动记忆文件夹来浏览 Claude 保存的内容。一切都是纯 markdown，您可以读取、编辑或删除。

我的 CLAUDE.md 太大了

超过 200 行的文件消耗更多上下文并可能降低遵守度。将详细内容移到使用 `@path` 导入引用的单独文件中（请参阅 [导入其他文件](#)），或将您的指令拆分到 `.claude/rules/` 文件中。

在 `/compact` 后指令似乎丢失了

CLAUDE.md 完全幸存压缩。在 `/compact` 后，Claude 从磁盘重新读取您的 CLAUDE.md 并将其新鲜重新注入会话。如果指令在压缩后消失，它仅在对话中给出，未写入 CLAUDE.md。将其添加到 CLAUDE.md 以使其在会话中持久化。

有关大小、结构和具体性的指导，请参阅 [编写有效的指令](#)。

相关资源

- [Skills](#)：打包按需加载的可重复工作流
- [Settings](#)：使用设置文件配置 Claude Code 行为
- [管理会话](#)：管理上下文、恢复对话和运行并行会话
- [Subagent 记忆](#)：让 subagents 维护自己的自动记忆

Claude Code 设置

使用全局和项目级设置以及环境变量配置 Claude Code。

Claude Code 提供多种设置来配置其行为以满足您的需求。您可以在使用交互式 REPL 时运行 `/config` 命令来配置 Claude Code，这会打开一个选项卡式设置界面，您可以在其中查看状态信息并修改配置选项。

配置作用域

Claude Code 使用**作用域系统**来确定配置应用的位置以及与谁共享。了解作用域可以帮助您决定如何为个人使用、团队协作或企业部署配置 Claude Code。

可用作用域

作用域	位置	影响范围	与团队共享?
Managed	服务器管理的设置、plist / 注册表或系统级 <code>managed-settings.json</code>	机器上的所有用户	是（由 IT 部署）
User	<code>~/.claude/</code> 目录	您，跨所有项目	否
Project	存储库中的 <code>.claude/</code>	此存储库上的所有协作者	是（提交到 git）
Local	<code>.claude/settings.local.json</code>	您，仅在此存储库中	否（gitignored）

何时使用每个作用域

Managed 作用域用于：

- 必须在整个组织范围内强制执行的安全策略
- 无法被覆盖的合规要求
- 由 IT/DevOps 部署的标准化配置

User 作用域最适合：

- 您想在任何地方使用的个人偏好（主题、编辑器设置）
- 您在所有项目中使用的工具和插件

- API 密钥和身份验证（安全存储）

Project 作用域最适合：

- 团队共享的设置（权限、hooks、MCP servers）
- 整个团队应该拥有的插件
- 跨协作者标准化工具

Local 作用域最适合：

- 特定项目的个人覆盖
- 在与团队共享之前测试配置
- 对其他人不适用的特定于机器的设置

作用域如何相互作用

当在多个作用域中配置相同的设置时，更具体的作用域优先：

1. **Managed**（最高） - 无法被任何内容覆盖
2. **命令行参数** - 临时会话覆盖
3. **Local** - 覆盖项目和用户设置
4. **Project** - 覆盖用户设置
5. **User**（最低） - 当没有其他内容指定设置时应用

例如，如果在用户设置中允许某个权限，但在项目设置中拒绝，则项目设置优先，权限被阻止。

哪些功能使用作用域

作用域适用于许多 Claude Code 功能：

功能	User 位置	Project 位置	Local 位置
Settings	~/.claude/settings.json	.claude/settings.json	.claude/settings.local.json
Subagents	~/.claude/agents/	.claude/agents/	—
MCP servers	~/.claude.json	.mcp.json	~/.claude.json（每个项目）
Plugins	~/.claude/settings.json	.claude/settings.json	.claude/settings.local.json

功能	User 位置	Project 位置	Local 位置
CLAUDE.m d	~/ .claude/CLAUDE.md	CLAUDE.md 或 .claude/ CLAUDE.md	—

设置文件

`settings.json` 文件是我们用于通过分层设置配置 Claude Code 的官方机制：

- 用户设置在 `~/ .claude/settings.json` 中定义，适用于所有项目。
- 项目设置保存在您的项目目录中：
 - `.claude/settings.json` 用于检入源代码管理并与您的团队共享的设置
 - `.claude/settings.local.json` 用于未检入的设置，适用于个人偏好和实验。
Claude Code 将在创建 `.claude/settings.local.json` 时配置 git 以忽略它。
- **Managed 设置：**对于需要集中控制的组织，Claude Code 支持多种 managed 设置的交付机制。所有机制都使用相同的 JSON 格式，无法被用户或项目设置覆盖：
 - **服务器管理的设置：**通过 Claude.ai 管理员控制台从 Anthropic 的服务器交付。请参阅[服务器管理的设置](#)。
 - **MDM/OS 级别策略：**通过 macOS 和 Windows 上的本机设备管理交付：
 - macOS：`com.anthropic.claudecode` 管理首选项域（通过 Jamf、Kandji 或其他 MDM 工具中的配置文件部署）
 - Windows：`HKLM\SOFTWARE\Policies\ClaudeCode` 注册表项，带有包含 JSON 的 `Settings` 值（REG_SZ 或 REG_EXPAND_SZ）（通过组策略或 Intune 部署）
 - Windows（用户级）：`HKCU\SOFTWARE\Policies\ClaudeCode`（最低策略优先级，仅在不存在管理员级源时使用）
 - **基于文件：**`managed-settings.json` 和 `managed-mcp.json` 部署到系统目录：
 - macOS：`/Library/Application Support/ClaudeCode/`
 - Linux 和 WSL：`/etc/claude-code/`
 - Windows：`C:\Program Files\ClaudeCode\`

请参阅[managed 设置](#)和 [Managed MCP 配置](#)了解详情。

Note:

Managed 部署也可以使用 `strictKnownMarketplaces` 限制插件市场添加。有关更多信息，请参阅 [Managed 市场限制](#)。

- 其他配置存储在 `~/.claude.json` 中。此文件包含您的偏好（主题、通知设置、编辑器模式）、OAuth 会话、[MCP server](#) 配置（用于用户和本地作用域）、每个项目的状态（允许的工具、信任设置）和各种缓存。项目作用域的 MCP servers 单独存储在 `.mcp.json` 中。

Note:

Claude Code 自动创建配置文件的时间戳备份，并保留最近五个备份以防止数据丢失。

```
{
  "$schema": "https://json.schemastore.org/claude-code-settings.json",
  "permissions": {
    "allow": [
      "Bash(npm run lint)",
      "Bash(npm run test *)",
      "Read(~/.zshrc)"
    ],
    "deny": [
      "Bash(curl *)",
      "Read(./env)",
      "Read(./env.*)",
      "Read(./secrets/**)"
    ]
  },
  "env": {
    "CLAUDE_CODE_ENABLE_TELEMETRY": "1",
    "OTEL_METRICS_EXPORTER": "otlp"
  },
  "companyAnnouncements": [
    "Welcome to Acme Corp! Review our code guidelines at docs.acme.com",
    "Reminder: Code reviews required for all PRs",
    "New security policy in effect"
  ]
}
```

上面示例中的 `$schema` 行指向 Claude Code 设置的[官方 JSON 架构](#)。将其添加到您的 `settings.json` 可在 VS Code、Cursor 和任何其他支持 JSON 架构验证的编辑器中启用自动完成和内联验证。

可用设置

`settings.json` 支持多个选项：

键	描述	示例
<code>apiKeyHelper</code>	自定义脚本，在 <code>/bin/sh</code> 中执行，以生成身份验证值。此值将作为 <code>X-Api-Key</code> 和 <code>Authorization: Bearer</code> 标头发送用于模型请求	<code>/bin/ generate_temp_api_key .sh</code>
<code>cleanupPeriodDays</code>	非活跃超过此期间的会话在启动时被删除。设置为 <code>0</code> 立即删除所有会话。（默认：30 天）	<code>20</code>
<code>companyAnnouncements</code>	在启动时显示给用户的公告。如果提供多个公告，它们将随机循环显示。	<code>["Welcome to Acme Corp! Review our code guidelines at docs.acme.com"]</code>
<code>env</code>	将应用于每个会话的环境变量	<code>{"F00": "bar"}</code>
<code>attribution</code>	自定义 git 提交和拉取请求的归属。请参阅 归属设置	<code>{"commit": " Generated with Claude Code", "pr": ""}</code>
<code>includeCoAuthoredBy</code>	已弃用 ：改用 <code>attribution</code> 。是否在 git 提交和拉取请求中包含 <code>co-authored-by Claude</code> 署名（默认： <code>true</code> ）	<code>false</code>
<code>includeGitInstructions</code>	在 Claude 的系统提示中包含内置提交和 PR 工作流说明（默认： <code>true</code> ）。设置为 <code>false</code> 以删除这些说明，例如在使用您自己的 git 工作流技能时。当设置时， <code>CLAUDE_CODE_DISABLE_GIT_INSTRUCTIONS</code> 环境变量优先于此设置	<code>false</code>
<code>permissions</code>	请参阅下表了解权限的结构。	
<code>hooks</code>	配置自定义命令以在生命周期事件处运行。请参阅 hooks 文档 了解格式	请参阅 hooks
<code>disableAllHooks</code>	禁用所有 hooks 和任何自定义 状态行	<code>true</code>

键	描述	示例
<code>allowManagedHooksOnly</code>	(仅 Managed 设置) 防止加载用户、项目和插件 hooks。仅允许 managed hooks 和 SDK hooks。请参阅 Hook 配置	<code>true</code>
<code>allowedHttpHookUrls</code>	HTTP hooks 可能针对的 URL 模式的允许列表。支持 <code>*</code> 作为通配符。设置后，具有不匹配 URL 的 hooks 被阻止。未定义 = 无限制，空数组 = 阻止所有 HTTP hooks。数组跨设置源合并。请参阅 Hook 配置	<code>["https://hooks.example.com/*"]</code>
<code>httpHookAllowedEnvVars</code>	HTTP hooks 可能插入到标头中的环境变量名称的允许列表。设置后，每个 hook 的有效 <code>allowedEnvVars</code> 是与此列表的交集。未定义 = 无限制。数组跨设置源合并。请参阅 Hook 配置	<code>["MY_TOKEN", "HOOK_SECRET"]</code>
<code>allowManagedPermissionsRulesOnly</code>	(仅 Managed 设置) 防止用户和项目设置定义 <code>allow</code> 、 <code>ask</code> 或 <code>deny</code> 权限规则。仅应用 managed 设置中的规则。请参阅 Managed 专用设置	<code>true</code>
<code>allowManagedMcpServersOnly</code>	(仅 Managed 设置) 仅尊重来自 managed 设置的 <code>allowedMcpServers</code> 。 <code>deniedMcpServers</code> 仍从所有源合并。用户仍可以添加 MCP servers，但仅应用管理员定义的允许列表。请参阅 Managed MCP 配置	<code>true</code>
<code>model</code>	覆盖 Claude Code 使用的默认模型	<code>"claude-sonnet-4-6"</code>
<code>availableModels</code>	限制用户可以通过 <code>/model</code> 、 <code>--model</code> 、Config 工具或 <code>ANTHROPIC_MODEL</code> 选择的模型。不影响默认选项。请参阅 限制模型选择	<code>["sonnet", "haiku"]</code>

键	描述	示例
<code>modelOverrides</code>	将 Anthropic 模型 ID 映射到特定于提供商的模型 ID，例如 Bedrock 推理配置文件 ARN。每个模型选择器条目在调用提供商 API 时使用其映射值。请参阅 按版本覆盖模型 ID	<pre>{"claude-opus-4-6": "arn:aws:bedrock:..." }</pre>
<code>otelHeadersHelper</code>	生成动态 OpenTelemetry 标头的脚本。在启动时和定期运行（请参阅 动态标头 ）	<pre>/bin/ generate_otel_headers .sh</pre>
<code>statusLine</code>	配置自定义状态行以显示上下文。请参阅 statusLine 文档	<pre>{"type": "command", "command": "~/.claude/ statusline.sh"}</pre>
<code>fileSuggestion</code>	为 @ 文件自动完成配置自定义脚本。请参阅 文件建议设置	<pre>{"type": "command", "command": "~/.claude/file- suggestion.sh"}</pre>
<code>respectGitignore</code>	控制 @ 文件选择器是否尊重 <code>.gitignore</code> 模式。当为 <code>true</code> （默认）时，匹配 <code>.gitignore</code> 模式的文件被排除在建议之外	<pre>false</pre>
<code>outputStyle</code>	配置输出样式以调整系统提示。请参阅 输出样式文档	<pre>"Explanatory"</pre>
<code>forceLoginMethod</code>	使用 <code>claudeai</code> 限制登录到 Claude.ai 账户， <code>console</code> 限制登录到 Claude Console（API 使用计费）账户	<pre>claudeai</pre>
<code>forceLoginOrgUUID</code>	指定组织的 UUID 以在登录期间自动选择它，绕过组织选择步骤。需要设置 <code>forceLoginMethod</code>	<pre>"xxxxxxxx-xxxx-xxxx- xxxx-xxxxxxxxxxxx"</pre>
<code>enableAllProjectMcpServers</code>	自动批准项目 <code>.mcp.json</code> 文件中定义的所有 MCP servers	<pre>true</pre>
<code>enabledMcpjsonServers</code>	要批准的 <code>.mcp.json</code> 文件中特定 MCP servers 的列表	<pre>["memory", "github"]</pre>

键	描述	示例
<code>disabledMcpjsonServers</code>	要拒绝的 <code>.mcp.json</code> 文件中特定 MCP servers 的列表	<code>["filesystem"]</code>
<code>allowedMcpServers</code>	在 <code>managed-settings.json</code> 中设置时，用户可以配置的 MCP servers 的允许列表。未定义 = 无限制，空数组 = 锁定。适用于所有作用域。拒绝列表优先。请参阅 Managed MCP 配置	<code>[{"serverName": "github"}]</code>
<code>deniedMcpServers</code>	在 <code>managed-settings.json</code> 中设置时，明确阻止的 MCP servers 的拒绝列表。适用于所有作用域，包括 managed servers。拒绝列表优先于允许列表。请参阅 Managed MCP 配置	<code>[{"serverName": "filesystem"}]</code>
<code>strictKnownMarketplaces</code>	在 <code>managed-settings.json</code> 中设置时，用户可以添加的插件市场的允许列表。未定义 = 无限制，空数组 = 锁定。仅适用于市场添加。请参阅 Managed 市场限制	<code>[{"source": "github", "repo": "acme-corp/plugins"}]</code>
<code>blockedMarketplaces</code>	（仅 Managed 设置）市场源的阻止列表。在下载前检查被阻止的源，因此它们永远不会接触文件系统。请参阅 Managed 市场限制	<code>[{"source": "github", "repo": "untrusted/plugins"}]</code>
<code>pluginTrustMessage</code>	（仅 Managed 设置）在安装前显示的插件信任警告中附加的自定义消息。使用此消息添加组织特定的上下文，例如确认来自您的内部市场的插件已被审查。	<code>"All plugins from our marketplace are approved by IT"</code>
<code>awsAuthRefresh</code>	修改 <code>.aws</code> 目录的自定义脚本（请参阅 高级凭证配置 ）	<code>aws sso login --profile myprofile</code>
<code>awsCredentialExport</code>	输出包含 AWS 凭证的 JSON 的自定义脚本（请参阅 高级凭证配置 ）	<code>/bin/generate_aws_grant.sh</code>
<code>alwaysThinkingEnabled</code>	为所有会话默认启用 扩展思考 。通常通过 <code>/config</code> 命令而不是直接编辑来配置	<code>true</code>

键	描述	示例
<code>plansDirectory</code>	自定义计划文件的存储位置。路径相对于项目根目录。默认： <code>~/.claude/plans</code>	<code>"./plans"</code>
<code>showTurnDuration</code>	在响应后显示轮次持续时间消息（例如“Cooked for 1m 6s”）。设置为 <code>false</code> 以隐藏这些消息	<code>true</code>
<code>spinnerVerbs</code>	自定义在微调器和轮次持续时间消息中显示的操作动词。将 <code>mode</code> 设置为 <code>"replace"</code> 以仅使用您的动词，或 <code>"append"</code> 以将它们添加到默认值	<code>{"mode": "append", "verbs": ["Pondering", "Crafting"]}</code>
<code>language</code>	配置 Claude 的首选响应语言（例如 <code>"japanese"</code> 、 <code>"spanish"</code> 、 <code>"french"</code> ） Claude 将默认以此语言响应	<code>"japanese"</code>
<code>autoUpdatesChannel</code>	遵循更新的发布渠道。使用 <code>"stable"</code> 获取通常约一周前的版本并跳过有重大回归的版本，或使用 <code>"latest"</code> （默认）获取最新版本	<code>"stable"</code>
<code>spinnerTipsEnabled</code>	在 Claude 工作时在微调器中显示提示。设置为 <code>false</code> 以禁用提示（默认： <code>true</code> ）	<code>false</code>
<code>spinnerTipsOverride</code>	用自定义字符串覆盖微调器提示。 <code>tips</code> ：提示字符串数组。 <code>excludeDefault</code> ：如果为 <code>true</code> ，仅显示自定义提示；如果为 <code>false</code> 或不存在，自定义提示与内置提示合并	<code>{ "excludeDefault": true, "tips": ["Use our internal tool X"] }</code>
<code>terminalProgressBarEnabled</code>	启用终端进度条，在 Windows Terminal 和 iTerm2 等支持的终端中显示进度（默认： <code>true</code> ）	<code>false</code>
<code>prefersReducedMotion</code>	减少或禁用 UI 动画（微调器、闪烁、闪光效果）以实现可访问性	<code>true</code>

键	描述	示例
<code>fastModePerSessionOptIn</code>	当为 <code>true</code> 时，快速模式不会跨会话持续。每个会话以快速模式关闭开始，需要用户使用 <code>/fast</code> 启用它。用户的快速模式偏好仍被保存。请参阅 需要每个会话的选择加入	<code>true</code>
<code>teammateMode</code>	<code>agent team</code> 队友如何显示： <code>auto</code> （在 <code>tmux</code> 或 <code>iTerm2</code> 中选择分割窗口，否则进程内）、 <code>in-process</code> 或 <code>tmux</code> 。请参阅 设置 agent teams	<code>"in-process"</code>

权限设置

键	描述	示例
<code>allow</code>	允许工具使用的权限规则数组。请参阅下面的 权限规则语法 了解模式匹配详情	<code>["Bash(git diff *)"]</code>
<code>ask</code>	要求在工具使用时确认的权限规则数组。请参阅下面的 权限规则语法	<code>["Bash(git push *)"]</code>
<code>deny</code>	拒绝工具使用的权限规则数组。使用此选项从 Claude Code 访问中排除敏感文件。请参阅 权限规则语法 和 Bash 权限限制	<code>["WebFetch", "Bash(curl *)", "Read(./env)", "Read(./secrets/**)"]</code>
<code>additionalDirectories</code>	Claude 有权访问的额外 工作目录	<code>["../docs/"]</code>
<code>defaultMode</code>	打开 Claude Code 时的默认 权限模式	<code>"acceptEdits"</code>
<code>disableBypassPermissionsMode</code>	设置为 <code>"disable"</code> 以防止激活 <code>bypassPermissions</code> 模式。这禁用 <code>--dangerously-skip-permissions</code> 命令行标志。请参阅 managed 设置	<code>"disable"</code>

权限规则语法

权限规则遵循格式 `Tool` 或 `Tool(specifier)`。规则按顺序评估：首先是拒绝规则，然后是询问，最后是允许。第一个匹配的规则获胜。

快速示例：

规则	效果
<code>Bash</code>	匹配所有 Bash 命令
<code>Bash(npm run *)</code>	匹配以 <code>npm run</code> 开头的命令
<code>Read(./env)</code>	匹配读取 <code>.env</code> 文件
<code>WebFetch(domain:example.com)</code>	匹配对 <code>example.com</code> 的获取请求

有关完整的规则语法参考，包括通配符行为、Read、Edit、WebFetch、MCP 和 Agent 规则的工具特定模式，以及 Bash 模式的安全限制，请参阅[权限规则语法](#)。

沙箱设置

配置高级沙箱行为。沙箱将 bash 命令与您的文件系统和网络隔离。请参阅[沙箱](#)了解详情。

键	描述	示例
<code>enabled</code>	启用 bash 沙箱（macOS、Linux 和 WSL2）。默认：false	<code>true</code>
<code>autoAllowBashIfSandboxed</code>	沙箱化时自动批准 bash 命令。默认：true	<code>true</code>
<code>excludedCommands</code>	应在沙箱外运行的命令	<code>["git", "docker"]</code>
<code>allowUnsandboxedCommands</code>	允许命令通过 <code>dangerouslyDisableSandbox</code> 参数在沙箱外运行。当设置为 <code>false</code> 时， <code>dangerouslyDisableSandbox</code> 逃生舱完全禁用，所有命令必须沙箱化（或在 <code>excludedCommands</code> 中）。对于需要严格沙箱的企业策略很有用。默认：true	<code>false</code>
<code>filesystem.allowWrite</code>	沙箱化命令可以写入的额外路径。数组跨所有设置作用域合并：用户、项目和 managed 路径组合，不替换。也与 <code>Edit(...)</code> 允许权限规则中的路径合并。请参阅下面的 路径前缀 。	<code>["//tmp/build", "~/.kube"]</code>

键	描述	示例
<code>filesystem.denyWrite</code>	沙箱化命令无法写入的路径。数组跨所有设置作用域合并。也与 <code>Edit(...)</code> 拒绝权限规则中的路径合并。	<code>["//etc", "//usr/local/bin"]</code>
<code>filesystem.denyRead</code>	沙箱化命令无法读取的路径。数组跨所有设置作用域合并。也与 <code>Read(...)</code> 拒绝权限规则中的路径合并。	<code>["~/.aws/credentials"]</code>
<code>network.allowUnixSockets</code>	沙箱中可访问的 Unix 套接字路径（用于 SSH 代理等）	<code>["~/.ssh/agent-socket"]</code>
<code>network.allowAllUnixSockets</code>	允许沙箱中的所有 Unix 套接字连接。默认: false	<code>true</code>
<code>network.allowLocalBinding</code>	允许绑定到 localhost 端口（仅 macOS）。默认: false	<code>true</code>
<code>network.allowedDomains</code>	允许出站网络流量的域数组。支持通配符（例如 <code>*.example.com</code> ）。	<code>["github.com", "*.npmjs.org"]</code>
<code>network.allowManagedDomainsOnly</code>	（仅 Managed 设置）仅尊重来自 managed 设置的 <code>allowedDomains</code> 和 <code>WebFetch(domain:...)</code> 允许规则。来自用户、项目和本地设置的域被忽略。非允许的域自动被阻止，不提示用户。拒绝的域仍从所有源受尊重。默认: false	<code>true</code>
<code>network.httpProxyPort</code>	如果您想自带代理，使用的 HTTP 代理端口。如果未指定，Claude 将运行自己的代理。	<code>8080</code>
<code>network.socksProxyPort</code>	如果您想自带代理，使用的 SOCKS5 代理端口。如果未指定，Claude 将运行自己的代理。	<code>8081</code>
<code>enableWeakerNestedSandbox</code>	为无特权 Docker 环境启用较弱的沙箱（仅 Linux 和 WSL2）。 降低安全性 。默认: false	<code>true</code>

键	描述	示例
<code>enableWeakerNetworkIsolation</code>	（仅 macOS）允许在沙箱中访问系统 TLS 信任服务（ <code>com.apple.trustd.agent</code> ）。对于 Go 基础工具（如 <code>gh</code> 、 <code>gcloud</code> 和 <code>terraform</code> ）在使用 <code>httpProxyPort</code> 与 MITM 代理和自定义 CA 时验证 TLS 证书是必需的。 通过打开潜在的数据泄露路径降低安全性。 默认：false	<code>true</code>

沙箱路径前缀

`filesystem.allowWrite`、`filesystem.denyWrite` 和 `filesystem.denyRead` 中的路径支持这些前缀：

前缀	含义	示例
<code>//</code>	从文件系统根目录的绝对路径	<code>//tmp/build</code> 变为 <code>/tmp/build</code>
<code>~/</code>	相对于主目录	<code>~/.kube</code> 变为 <code>\$HOME/.kube</code>
<code>/</code>	相对于设置文件的目录	<code>/build</code> 变为 <code>\$SETTINGS_DIR/build</code>
<code>./</code> 或无前缀	相对路径（由沙箱运行时解析）	<code>./output</code>

配置示例：

```
{
  "sandbox": {
    "enabled": true,
    "autoAllowBashIfSandboxed": true,
    "excludedCommands": ["docker"],
    "filesystem": {
      "allowWrite": ["/tmp/build", "~/kube"],
      "denyRead": ["/.aws/credentials"]
    },
    "network": {
      "allowedDomains": ["github.com", "*.npmjs.org", "registry.yarnpkg.com"],
      "allowUnixSockets": [
        "/var/run/docker.sock"
      ],
      "allowLocalBinding": true
    }
  }
}
```

文件系统和网络限制可以通过两种合并在一起的方式配置：

- **sandbox.filesystem** 设置（如上所示）：在 OS 级沙箱边界处控制路径。这些限制适用于所有子进程命令（例如 `kubectl`、`terraform`、`npm`），而不仅仅是 Claude 的文件工具。
- **权限规则**：使用 `Edit` 允许/拒绝规则控制 Claude 的文件工具访问，`Read` 拒绝规则阻止读取，`WebFetch` 允许/拒绝规则控制网络域。这些规则中的路径也合并到沙箱配置中。

归属设置

Claude Code 为 git 提交和拉取请求添加归属。这些分别配置：

- 提交默认使用 [git trailers](#)（如 `Co-Authored-By`），可以自定义或禁用
- 拉取请求描述是纯文本

键	描述
<code>commit</code>	git 提交的归属，包括任何 trailers。空字符串隐藏提交归属
<code>pr</code>	拉取请求描述的归属。空字符串隐藏拉取请求归属

默认提交归属：

```
 Generated with [Claude Code](https://claude.com/claude-code)

Co-Authored-By: Claude Sonnet 4.6 <noreply@anthropic.com>
```

默认拉取请求归属：

```
 Generated with [Claude Code](https://claude.com/claude-code)
```

示例：

```
{
  "attribution": {
    "commit": "Generated with AI\n\nCo-Authored-By: AI <ai@example.com>",
    "pr": ""
  }
}
```

Note:

attribution 设置优先于已弃用的 **includeCoAuthoredBy** 设置。要隐藏所有归属，将 **commit** 和 **pr** 设置为空字符串。

文件建议设置

为 @ 文件路径自动完成配置自定义命令。内置文件建议使用快速文件系统遍历，但大型 monorepos 可能受益于项目特定的索引，例如预构建的文件索引或自定义工具。

```
{
  "fileSuggestion": {
    "type": "command",
    "command": "~/ .claude/file-suggestion.sh"
  }
}
```

该命令使用与 [hooks](#) 相同的环境变量运行，包括 `CLAUDE_PROJECT_DIR`。它通过 stdin 接收包含 `query` 字段的 JSON：

```
{"query": "src/comp"}
```

将换行符分隔的文件路径输出到 stdout（当前限制为 15）：

```
src/components/Button.tsx
src/components/Modal.tsx
src/components/Form.tsx
```

示例：

```
#!/bin/bash
query=$(cat | jq -r '.query')
your-repo-file-index --query "$query" | head -20
```

Hook 配置

这些设置控制允许运行哪些 hooks 以及 HTTP hooks 可以访问什么。

`allowManagedHooksOnly` 设置只能在 [managed 设置](#) 中配置。URL 和环境变量允许列表可以在任何设置级别设置并跨源合并。

当 `allowManagedHooksOnly` 为 `true` 时的行为：

- 加载 Managed hooks 和 SDK hooks
- 用户 hooks、项目 hooks 和插件 hooks 被阻止

限制 HTTP hook URL：

限制 HTTP hooks 可以针对的 URL。支持 `*` 作为匹配的通配符。定义数组后，针对不匹配 URL 的 HTTP hooks 被静默阻止。

```
{
  "allowedHttpHookUrls": ["https://hooks.example.com/*", "http://localhost:*"]
}
```

限制 HTTP hook 环境变量：

限制 HTTP hooks 可以插入到头值中的环境变量名称。每个 hook 的有效 `allowedEnvVars` 是其自己列表与此设置的交集。

```
{
  "httpHookAllowedEnvVars": ["MY_TOKEN", "HOOK_SECRET"]
}
```

设置优先级

设置按优先级顺序应用。从最高到最低：

1. Managed 设置（[服务器管理](#)、[MDM/OS 级别策略](#)或 [managed 设置](#)）

- 由 IT 通过服务器交付、MDM 配置文件、注册表策略或 managed 设置文件部署的策略
- 无法被任何其他级别覆盖，包括命令行参数
- 在 managed 层内，优先级为：server-managed > MDM/OS 级别策略 > `managed-settings.json` > HKCU 注册表（仅 Windows）。仅使用一个 managed 源；源不合并。

2. 命令行参数

- 特定会话的临时覆盖

3. 本地项目设置（`.claude/settings.local.json`）

- 个人项目特定设置

4. 共享项目设置（`.claude/settings.json`）

- 源代码管理中的团队共享项目设置

5. 用户设置（`~/.claude/settings.json`）

- 个人全局设置

此层次结构确保组织策略始终被强制执行，同时仍允许团队和个人自定义其体验。

例如，如果您的用户设置允许 `Bash(npm run *)`，但项目的共享设置拒绝它，则项目设置优先，命令被阻止。

Note:

数组设置跨作用域合并。 当相同的数组值设置（例如 `sandbox.filesystem.allowWrite` 或 `permissions.allow`）出现在多个作用域中时，数组被**连接和去重**，而不是替换。这意味着较低优先级的作用域可以添加条目而不覆盖由较高优先级作用域设置的条目，反之亦然。例如，如果 `managed` 设置将 `allowWrite` 设置为 `["//opt/company-tools"]`，用户添加 `["~/.kube"]`，则最终配置中包含两个路径。

验证活跃设置

在 Claude Code 中运行 `/status` 以查看哪些设置源处于活跃状态以及它们来自何处。输出显示每个配置层（`managed`、`user`、`project`）及其来源，例如 `Enterprise managed settings (remote)`、`Enterprise managed settings (plist)`、`Enterprise managed settings (HKLM)` 或 `Enterprise managed settings (file)`。如果设置文件包含错误，`/status` 会报告问题，以便您可以修复它。

配置系统的关键点

- **内存文件（`CLAUDE.md`）**：包含 Claude 在启动时加载的说明和上下文
- **设置文件（JSON）**：配置权限、环境变量和工具行为
- **Skills**：可以使用 `/skill-name` 调用或由 Claude 自动加载的自定义提示
- **MCP servers**：使用额外的工具和集成扩展 Claude Code
- **优先级**：更高级别的配置（Managed）覆盖较低级别的配置（User/Project）
- **继承**：设置被合并，更具体的设置添加到或覆盖更广泛的设置

系统提示

Claude Code 的内部系统提示未发布。要添加自定义说明，请使用 `CLAUDE.md` 文件或 `--append-system-prompt` 标志。

排除敏感文件

要防止 Claude Code 访问包含敏感信息（如 API 密钥、秘密和环境文件）的文件，请在您的 `.claude/settings.json` 文件中使用 `permissions.deny` 设置：

```
{
  "permissions": {
    "deny": [
      "Read(./env)",
      "Read(./env.*)",
      "Read(/secrets/**)",
      "Read(/config/credentials.json)",
      "Read(/build)"
    ]
  }
}
```

这替代了已弃用的 `ignorePatterns` 配置。匹配这些模式的文件被排除在文件发现和搜索结果之外，这些文件上的读取操作被拒绝。

Subagent 配置

Claude Code 支持可以在用户和项目级别配置的自定义 AI subagents。这些 subagents 存储为带有 YAML frontmatter 的 Markdown 文件：

- **用户 subagents**：`~/.claude/agents/` - 在所有项目中可用
- **项目 subagents**：`.claude/agents/` - 特定于您的项目，可以与您的团队共享

Subagent 文件定义具有自定义提示和工具权限的专门 AI 助手。在 [subagents 文档](#) 中了解有关创建和使用 subagents 的更多信息。

插件配置

Claude Code 支持一个插件系统，让您可以使用 skills、agents、hooks 和 MCP servers 扩展功能。插件通过市场分发，可以在用户和存储库级别配置。

插件设置

`settings.json` 中的插件相关设置：

```
{
  "enabledPlugins": {
    "formatter@acme-tools": true,
    "deployer@acme-tools": true,
    "analyzer@security-plugins": false
  },
  "extraKnownMarketplaces": {
    "acme-tools": {
      "source": "github",
      "repo": "acme-corp/claude-plugins"
    }
  }
}
```

enabledPlugins

控制启用哪些插件。格式： "plugin-name@marketplace-name": true/false

作用域：

- 用户设置（`~/.claude/settings.json`）：个人插件偏好
- 项目设置（`.claude/settings.json`）：与团队共享的项目特定插件
- 本地设置（`.claude/settings.local.json`）：每台机器的覆盖（未提交）

示例：

```
{
  "enabledPlugins": {
    "code-formatter@team-tools": true,
    "deployment-tools@team-tools": true,
    "experimental-features@personal": false
  }
}
```

extraKnownMarketplaces

定义应为存储库提供的额外市场。通常在存储库级设置中使用，以确保团队成员可以访问所需的插件源。

当存储库包含 `extraKnownMarketplaces` 时：

1. 当团队成员信任该文件夹时，会提示他们安装市场
2. 然后提示团队成员从该市场安装插件
3. 用户可以跳过不需要的市场或插件（存储在用户设置中）
4. 安装尊重信任边界并需要明确同意

示例：

```
{
  "extraKnownMarketplaces": {
    "acme-tools": {
      "source": {
        "source": "github",
        "repo": "acme-corp/claude-plugins"
      }
    },
    "security-plugins": {
      "source": {
        "source": "git",
        "url": "https://git.example.com/security/plugins.git"
      }
    }
  }
}
```

市场源类型：

- `github`：GitHub 存储库（使用 `repo`）
- `git`：任何 git URL（使用 `url`）
- `directory`：本地文件系统路径（使用 `path`，仅用于开发）
- `hostPattern`：正则表达式模式以匹配市场主机（使用 `hostPattern`）

`strictKnownMarketplaces`

仅 Managed 设置：控制用户可以添加哪些插件市场。此设置只能在 [managed 设置](#) 中配置，为管理员提供对市场源的严格控制。

Managed 设置文件位置：

- **macOS**：`/Library/Application Support/ClaudeCode/managed-settings.json`

- **Linux 和 WSL** : `/etc/claude-code/managed-settings.json`
- **Windows** : `C:\Program Files\ClaudeCode\managed-settings.json`

关键特征：

- 仅在 managed 设置 (`managed-settings.json`) 中可用
- 无法被用户或项目设置覆盖 (最高优先级)
- 在网络/文件系统操作之前强制执行 (被阻止的源永远不会执行)
- 对源规范使用精确匹配 (包括 git 源的 `ref`、`path`)，除了 `hostPattern`，它使用正则表达式匹配

允许列表行为：

- `undefined` (默认)：无限制 - 用户可以添加任何市场
- 空数组 `[]`：完全锁定 - 用户无法添加任何新市场
- 源列表：用户只能添加与之精确匹配的市场

所有支持的源类型：

允许列表支持七种市场源类型。大多数源使用精确匹配，而 `hostPattern` 使用正则表达式匹配市场主机。

1. GitHub 存储库：

```
{ "source": "github", "repo": "acme-corp/approved-plugins" }
{ "source": "github", "repo": "acme-corp/security-tools", "ref": "v2.0" }
{ "source": "github", "repo": "acme-corp/plugins", "ref": "main", "path": "marketplace" }
```

字段：`repo` (必需)、`ref` (可选：分支/标签/SHA)、`path` (可选：子目录)

1. Git 存储库：

```
{ "source": "git", "url": "https://gitlab.example.com/tools/plugins.git" }
{ "source": "git", "url": "https://bitbucket.org/acme-corp/plugins.git", "ref": "production" }
{ "source": "git", "url": "ssh://git@git.example.com/plugins.git", "ref": "v3.1", "path": "approved" }
```

字段：`url` (必需)、`ref` (可选：分支/标签/SHA)、`path` (可选：子目录)

1. 基于 URL 的市场：

```
{ "source": "url", "url": "https://plugins.example.com/marketplace.json" }
{ "source": "url", "url": "https://cdn.example.com/marketplace.json", "headers":
{ "Authorization": "Bearer ${TOKEN}" } }
```

字段：`url`（必需）、`headers`（可选：用于身份验证访问的 HTTP 标头）

Note:

基于 URL 的市场仅下载 `marketplace.json` 文件。它们不从服务器下载插件文件。基于 URL 的市场中的插件必须使用外部源（GitHub、npm 或 git URL），而不是相对路径。对于具有相对路径的插件，改用基于 Git 的市场。请参阅[故障排除](#)了解详情。

1. NPM 包：

```
{ "source": "npm", "package": "@acme-corp/claude-plugins" }
{ "source": "npm", "package": "@acme-corp/approved-marketplace" }
```

字段：`package`（必需，支持作用域包）

1. 文件路径：

```
{ "source": "file", "path": "/usr/local/share/claude/acme-marketplace.json" }
{ "source": "file", "path": "/opt/acme-corp/plugins/marketplace.json" }
```

字段：`path`（必需：marketplace.json 文件的绝对路径）

1. 目录路径：

```
{ "source": "directory", "path": "/usr/local/share/claude/acme-plugins" }
{ "source": "directory", "path": "/opt/acme-corp/approved-marketplaces" }
```

字段：`path`（必需：包含 `.claude-plugin/marketplace.json` 的目录的绝对路径）

1. 主机模式匹配：

```
{ "source": "hostPattern", "hostPattern": "^github\\.example\\.com$" }
{ "source": "hostPattern", "hostPattern": "^gitlab\\.internal\\.example\\.com$" }
```

字段: `hostPattern` (必需: 与市场主机匹配的正则表达式模式)

当您想允许来自特定主机的所有市场而不枚举每个存储库时, 使用主机模式匹配。这对于具有内部 GitHub Enterprise 或 GitLab 服务器的组织很有用, 开发人员在其中创建自己的市场。

按源类型的主机提取:

- `github`: 始终与 `github.com` 匹配
- `git`: 从 URL 提取主机名 (支持 HTTPS 和 SSH 格式)
- `url`: 从 URL 提取主机名
- `npm`、`file`、`directory`: 不支持主机模式匹配

配置示例:

示例: 仅允许特定市场:

```
{
  "strictKnownMarketplaces": [
    {
      "source": "github",
      "repo": "acme-corp/approved-plugins"
    },
    {
      "source": "github",
      "repo": "acme-corp/security-tools",
      "ref": "v2.0"
    },
    {
      "source": "url",
      "url": "https://plugins.example.com/marketplace.json"
    },
    {
      "source": "npm",
      "package": "@acme-corp/compliance-plugins"
    }
  ]
}
```

示例 - 禁用所有市场添加:

```
{
  "strictKnownMarketplaces": []
}
```

示例：允许来自内部 git 服务器的所有市场：

```
{
  "strictKnownMarketplaces": [
    {
      "source": "hostPattern",
      "hostPattern": "^github\\.example\\.com$"
    }
  ]
}
```

精确匹配要求：

市场源必须**精确**匹配以允许用户的添加。对于基于 git 的源（`github` 和 `git`），这包括所有可选字段：

- `repo` 或 `url` 必须精确匹配
- `ref` 字段必须精确匹配（或两者都未定义）
- `path` 字段必须精确匹配（或两者都未定义）

不匹配的源示例：

```
// 这些是不同的源：
{ "source": "github", "repo": "acme-corp/plugins" }
{ "source": "github", "repo": "acme-corp/plugins", "ref": "main" }

// 这些也是不同的：
{ "source": "github", "repo": "acme-corp/plugins", "path": "marketplace" }
{ "source": "github", "repo": "acme-corp/plugins" }
```

与 `extraKnownMarketplaces` 的比较：

方面	<code>strictKnownMarketplaces</code>	<code>extraKnownMarketplaces</code>
目的	组织策略强制执行	团队便利
设置文件	仅 <code>managed-settings.json</code>	任何设置文件
行为	阻止非允许列表的添加	自动安装缺失的市场
何时强制执行	在网络/文件系统操作之前	在用户信任提示之后
可以被覆盖	否（最高优先级）	是（由更高优先级设置）
源格式	直接源对象	具有嵌套源的命名市场
用例	合规、安全限制	入职、标准化

格式差异：

`strictKnownMarketplaces` 使用直接源对象：

```
{
  "strictKnownMarketplaces": [
    { "source": "github", "repo": "acme-corp/plugins" }
  ]
}
```

`extraKnownMarketplaces` 需要命名市场：

```
{
  "extraKnownMarketplaces": {
    "acme-tools": {
      "source": { "source": "github", "repo": "acme-corp/plugins" }
    }
  }
}
```

重要说明：

- 限制在任何网络请求或文件系统操作之前检查
- 被阻止时，用户看到清晰的错误消息，指示源被 `managed` 策略阻止
- 限制仅适用于添加新市场；以前安装的市场保持可访问

- Managed 设置具有最高优先级，无法被覆盖

请参阅 [Managed 市场限制](#) 了解面向用户的文档。

管理插件

使用 `/plugin` 命令以交互方式管理插件：

- 浏览来自市场的可用插件
- 安装/卸载插件
- 启用/禁用插件
- 查看插件详情（提供的命令、agents、hooks）
- 添加/删除市场

在 [plugins 文档](#) 中了解有关插件系统的更多信息。

环境变量

Claude Code 支持以下环境变量来控制其行为：

Note:

所有环境变量也可以在 `settings.json` 中配置。这是自动为每个会话设置环境变量或为整个团队或组织推出一组环境变量的有用方式。

变量	目的	
<code>ANTHROPIC_API_KEY</code>	作为 <code>X-API-Key</code> 标头发送的 API 密钥，通常用于 Claude SDK（对于交互式使用，运行 <code>/login</code> ）	
<code>ANTHROPIC_AUTH_TOKEN</code>	<code>Authorization</code> 标头的自定义值（您在此处设置的值将以 <code>Bearer</code> 为前缀）	
<code>ANTHROPIC_CUSTOM_HEADERS</code>	要添加到请求的自定义标头（ <code>Name: Value</code> 格式，多个标头用换行符分隔）	
<code>ANTHROPIC_DEFAULT_HAIKU_MODEL</code>	请参阅 模型配置	
<code>ANTHROPIC_DEFAULT_OPUS_MODEL</code>	请参阅 模型配置	

变量	目的	
<code>ANTHROPIC_DEFAULT_SONNET_MODEL</code>	请参阅 模型配置	
<code>ANTHROPIC_FOUNDRY_API_KEY</code>	Microsoft Foundry 身份验证的 API 密钥（请参阅 Microsoft Foundry ）	
<code>ANTHROPIC_FOUNDRY_BASE_URL</code>	Foundry 资源的完整基础 URL（例如 <code>https://my-resource.services.ai.azure.com/anthropic</code> ）。 <code>ANTHROPIC_FOUNDRY_RESOURCE</code> 的替代方案（请参阅 Microsoft Foundry ）	
<code>ANTHROPIC_FOUNDRY_RESOURCE</code>	Foundry 资源名称（例如 <code>my-resource</code> ）。如果未设置 <code>ANTHROPIC_FOUNDRY_BASE_URL</code> ，则为必需（请参阅 Microsoft Foundry ）	
<code>ANTHROPIC_MODEL</code>	要使用的模型设置的名称（请参阅 模型配置 ）	
<code>ANTHROPIC_SMALL_FAST_MODEL</code>	[已弃用] Haiku 级模型用于后台任务 的名称	
<code>ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION</code>	使用 Bedrock 时覆盖 Haiku 级模型的 AWS 区域	
<code>AWS_BEARER_TOKEN_BEDROCK</code>	Bedrock API 密钥用于身份验证（请参阅 Bedrock API 密钥 ）	
<code>BASH_DEFAULT_TIMEOUT_MS</code>	长时间运行的 bash 命令的默认超时	
<code>BASH_MAX_OUTPUT_LENGTH</code>	bash 输出中的最大字符数，超过此数量后会进行中间截断	
<code>BASH_MAX_TIMEOUT_MS</code>	模型可以为长时间运行的 bash 命令设置的最大超时	

变量	目的	
<code>CLAUDE_AUTOCOMPACT_PERCENT_OVERRIDE</code>	设置上下文容量的百分比（1-100），在该百分比处自动压缩触发。默认情况下，自动压缩在大约 95% 容量时触发。使用较低的值（如 <code>50</code> ）以更早压缩。高于默认阈值的值无效。适用于主对话和 subagents。此百分比与 状态行 中可用的 <code>context_window.used_percentage</code> 字段对齐	
<code>CLAUDE_BASH_MAINTAIN_PROJECT_WORKING_DIR</code>	在每个 Bash 命令后返回到原始工作目录	
<code>CLAUDE_CODE_ACCOUNT_UUID</code>	已认证用户的账户 UUID。由 SDK 调用者使用以同步提供账户信息，避免早期遥测事件缺少账户元数据的竞态条件。需要同时设置 <code>CLAUDE_CODE_USER_EMAIL</code> 和 <code>CLAUDE_CODE_ORGANIZATION_UUID</code>	
<code>CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD</code>	设置为 <code>1</code> 以从使用 <code>--add-dir</code> 指定的目录加载 CLAUDE.md 文件。默认情况下，额外目录不加载内存文件	<code>1</code>
<code>CLAUDE_CODE_API_KEY_HELPER_TTL_MS</code>	应刷新凭证的间隔（以毫秒为单位）（使用 <code>apiKeyHelper</code> 时）	
<code>CLAUDE_CODE_CLIENT_CERT</code>	mTLS 身份验证的客户端证书文件的路径	
<code>CLAUDE_CODE_CLIENT_KEY</code>	mTLS 身份验证的客户端私钥文件的路径	
<code>CLAUDE_CODE_CLIENT_KEY_PASSPHRASE</code>	加密 CLAUDE_CODE_CLIENT_KEY 的密码（可选）	
<code>CLAUDE_CODE_DISABLE_1M_CONTEXT</code>	设置为 <code>1</code> 以禁用 1M 上下文窗口 支持。设置后，1M 模型变体在模型选择器中不可用。对于具有合规要求的企业环境很有用	

变量	目的	
<code>CLAUDE_CODE_DISABLE_ADAPTIVE_THINKING</code>	设置为 <code>1</code> 以禁用 Opus 4.6 和 Sonnet 4.6 的自适应推理。禁用时，这些模型回退到由 <code>MAX_THINKING_TOKENS</code> 控制的固定思考预算	
<code>CLAUDE_CODE_DISABLE_AUTO_MEMORY</code>	设置为 <code>1</code> 以禁用自动内存。设置为 <code>0</code> 以在逐步推出期间强制启用自动内存。禁用时，Claude 不创建或加载自动内存文件	
<code>CLAUDE_CODE_DISABLE_GIT_INSTRUCTIONS</code>	设置为 <code>1</code> 以从 Claude 的系统提示中删除内置提交和 PR 工作流说明。在使用您自己的 git 工作流技能时很有用。当设置时优先于 <code>includeGitInstructions</code> 设置	
<code>CLAUDE_CODE_DISABLE_BACKGROUND_TASKS</code>	设置为 <code>1</code> 以禁用所有后台任务功能，包括 Bash 和 subagent 工具上的 <code>run_in_background</code> 参数、自动后台处理和 Ctrl+B 快捷键	
<code>CLAUDE_CODE_DISABLE_CRON</code>	设置为 <code>1</code> 以禁用计划任务。 <code>/loop</code> 技能和 cron 工具变为不可用，任何已计划的任务停止触发，包括已在会话中运行的任务	
<code>CLAUDE_CODE_DISABLE_EXPERIMENTAL_BETAS</code>	设置为 <code>1</code> 以禁用 Anthropic API 特定的 <code>anthropic-beta</code> 标头。如果使用具有第三方提供商的 LLM 网关时遇到“Unexpected value(s) for the <code>anthropic-beta</code> header”之类的问题，请使用此选项	
<code>CLAUDE_CODE_DISABLE_FAST_MODE</code>	设置为 <code>1</code> 以禁用快速模式	
<code>CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY</code>	设置为 <code>1</code> 以禁用“Claude 表现如何？”“会话质量调查。在使用第三方提供商或禁用遥测时也禁用。请参阅会话质量调查	

变量	目的	
<code>CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC</code>	等同于设置 <code>DISABLE_AUTOUPDATER</code> 、 <code>DISABLE_BUG_COMMAND</code> 、 <code>DISABLE_ERROR_REPORTING</code> 和 <code>DISABLE_TELEMETRY</code>	
<code>CLAUDE_CODE_DISABLE_TERMINAL_TITLE</code>	设置为 <code>1</code> 以禁用基于对话上下文的自动终端标题更新	
<code>CLAUDE_CODE_EFFORT_LEVEL</code>	为支持的模型设置努力级别。值： <code>low</code> 、 <code>medium</code> 、 <code>high</code> 。较低的努力更快且更便宜，较高的努力提供更深入的推理。在 Opus 4.6 和 Sonnet 4.6 上支持。请参阅 调整努力级别	
<code>CLAUDE_CODE_ENABLE_PROMPT_SUGGESTION</code>	设置为 <code>false</code> 以禁用提示建议（ <code>/config</code> 中的”提示建议”切换）。这些是在 Claude 响应后出现在您的提示输入中的灰显预测。请参阅 提示建议	
<code>CLAUDE_CODE_ENABLE_TASKS</code>	设置为 <code>false</code> 以临时恢复到之前的 TODO 列表而不是任务跟踪系统。默认： <code>true</code> 。请参阅 任务列表	
<code>CLAUDE_CODE_ENABLE_TELEMETRY</code>	设置为 <code>1</code> 以启用 OpenTelemetry 数据收集以获取指标和日志。在配置 OTEL 导出器之前需要。请参阅 监控	
<code>CLAUDE_CODE_EXIT_AFTER_STOP_DELAY</code>	查询循环变为空闲后自动退出前等待的时间（以毫秒为单位）。对于使用 SDK 模式的自动化工作流和脚本很有用	
<code>CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS</code>	设置为 <code>1</code> 以启用 agent teams 。Agent teams 是实验性的，默认禁用	
<code>CLAUDE_CODE_FILE_READ_MAX_OUTPUT_TOKENS</code>	覆盖文件读取的默认令牌限制。在需要完整读取较大文件时很有用	
<code>CLAUDE_CODE_HIDE_ACCOUNT_INFO</code>	设置为 <code>1</code> 以从 Claude Code UI 中隐藏您的电子邮件地址和组织名称。在流式传输或录制时很有用	

变量	目的	
<code>CLAUDE_CODE_IDE_SKIP_AUTO_INSTALL</code>	跳过 IDE 扩展的自动安装	
<code>CLAUDE_CODE_MAX_OUTPUT_TOKENS</code>	为大多数请求设置最大输出令牌数。默认：32,000。最大：64,000。增加此值会减少在 自动压缩 触发之前可用的有效上下文窗口。	
<code>CLAUDE_CODE_ORGANIZATION_UUID</code>	已认证用户的组织 UUID。由 SDK 调用者使用以同步提供账户信息。需要同时设置 <code>CLAUDE_CODE_ACCOUNT_UUID</code> 和 <code>CLAUDE_CODE_USER_EMAIL</code>	
<code>CLAUDE_CODE_OTEL_HELPERS_HELPER_DEBOUNCE_MS</code>	刷新动态 OpenTelemetry 标头的间隔（以毫秒为单位）（默认：1740000 / 29 分钟）。请参阅 动态标头	
<code>CLAUDE_CODE_PLAN_MODE_REQUIRED</code>	自动设置为 <code>true</code> 在需要计划批准的 agent team 队友上。只读：在生成队友时由 Claude Code 设置。请参阅 需要计划批准	
<code>CLAUDE_CODE_PLUGIN_GIT_TIMEOUT_MS</code>	安装或更新插件时 git 操作的超时（以毫秒为单位）（默认：120000）。对于大型存储库或慢速网络连接，增加此值。请参阅 Git 操作超时	
<code>CLAUDE_CODE_PROXY_RESOLVES_HOSTS</code>	设置为 <code>true</code> 以允许代理执行 DNS 解析而不是调用者。对于代理应处理主机名解析的环境选择加入	
<code>CLAUDE_CODE_SHELL</code>	覆盖自动 shell 检测。在您的登录 shell 与您的首选工作 shell 不同时很有用（例如 <code>bash</code> vs <code>zsh</code> ）	
<code>CLAUDE_CODE_SHELL_PREFIX</code>	命令前缀以包装所有 bash 命令（例如用于日志记录或审计）。示例： <code>/path/to/logger.sh</code> 将执行 <code>/path/to/logger.sh <command></code>	

变量	目的	
<code>CLAUDE_CODE_SIMPLE</code>	设置为 <code>1</code> 以使用最小系统提示和仅 Bash、文件读取和文件编辑工具运行。禁用 MCP 工具、附件、hooks 和 CLAUDE.md 文件	
<code>CLAUDE_CODE_SKIP_BEDROCK_AUTH</code>	跳过 Bedrock 的 AWS 身份验证（例如在使用 LLM 网关时）	
<code>CLAUDE_CODE_SKIP_FOUNDRY_AUTH</code>	跳过 Microsoft Foundry 的 Azure 身份验证（例如在使用 LLM 网关时）	
<code>CLAUDE_CODE_SKIP_VERTEX_AUTH</code>	跳过 Vertex 的 Google 身份验证（例如在使用 LLM 网关时）	
<code>CLAUDE_CODE_SUBAGENT_MODEL</code>	请参阅 模型配置	
<code>CLAUDE_CODE_TASK_LIST_ID</code>	跨会话共享任务列表。在多个 Claude Code 实例中设置相同的 ID 以协调共享任务列表。请参阅 任务列表	
<code>CLAUDE_CODE_TEAM_NAME</code>	此队友所属的 agent team 的名称。在 agent team 成员上自动设置	
<code>CLAUDE_CODE_TMPDIR</code>	覆盖用于内部临时文件的临时目录。Claude Code 将 <code>/claude/</code> 附加到此路径。默认：Unix/macOS 上的 <code>/tmp</code> ，Windows 上的 <code>os.tmpdir()</code>	
<code>CLAUDE_CODE_USER_EMAIL</code>	已认证用户的电子邮件地址。由 SDK 调用者使用以同步提供账户信息。需要同时设置 <code>CLAUDE_CODE_ACCOUNT_UUID</code> 和 <code>CLAUDE_CODE_ORGANIZATION_UUID</code>	
<code>CLAUDE_CODE_USE_BEDROCK</code>	使用 Bedrock	
<code>CLAUDE_CODE_USE_FOUNDRY</code>	使用 Microsoft Foundry	

变量	目的	
<code>CLAUDE_CODE_USE_VERTEX</code>	使用 Vertex	
<code>CLAUDE_CONFIG_DIR</code>	自定义 Claude Code 存储其配置和数据文件的位置	
<code>DISABLE_AUTOUPDATER</code>	设置为 <code>1</code> 以禁用自动更新。	
<code>DISABLE_BUG_COMMAND</code>	设置为 <code>1</code> 以禁用 <code>/bug</code> 命令	
<code>DISABLE_COST_WARNINGS</code>	设置为 <code>1</code> 以禁用成本警告消息	
<code>DISABLE_ERROR_REPORTING</code>	设置为 <code>1</code> 以选择退出 Sentry 错误报告	
<code>DISABLE_INSTALLATION_CHECKS</code>	设置为 <code>1</code> 以禁用安装警告。仅在手动管理安装位置时使用，因为这可能会掩盖标准安装的问题	
<code>DISABLE_NON_ESSENTIAL_MODEL_CALLS</code>	设置为 <code>1</code> 以禁用非关键路径（如风味文本）的模型调用	
<code>DISABLE_PROMPT_CACHING</code>	设置为 <code>1</code> 以禁用所有模型的提示缓存（优先于每个模型的设置）	
<code>DISABLE_PROMPT_CACHING_HAIKU</code>	设置为 <code>1</code> 以禁用 Haiku 模型的提示缓存	
<code>DISABLE_PROMPT_CACHING_OPUS</code>	设置为 <code>1</code> 以禁用 Opus 模型的提示缓存	
<code>DISABLE_PROMPT_CACHING_SONNET</code>	设置为 <code>1</code> 以禁用 Sonnet 模型的提示缓存	
<code>DISABLE_TELEMETRY</code>	设置为 <code>1</code> 以选择退出 Statsig 遥测（注意 Statsig 事件不包括用户数据，如代码、文件路径或 bash 命令）	
<code>ENABLE_CLAUDEAI_MCP_SERVERS</code>	设置为 <code>false</code> 以禁用 Claude Code 中的 claude.ai MCP servers 。对于已登录的用户默认启用	

变量	目的	
<code>ENABLE_TOOL_SEARCH</code>	控制 MCP 工具搜索 。值： <code>auto</code> （默认，在 10% 上下文处启用）、 <code>auto:N</code> （自定义阈值，例如 <code>auto:5</code> 表示 5%）、 <code>true</code> （始终开启）、 <code>false</code> （禁用）	
<code>FORCE_AUTOUPDATE_PLUGINS</code>	设置为 <code>true</code> 以强制插件自动更新，即使主自动更新程序通过 <code>DISABLE_AUTOUPDATER</code> 禁用	
<code>HTTP_PROXY</code>	为网络连接指定 HTTP 代理服务器	
<code>HTTPS_PROXY</code>	为网络连接指定 HTTPS 代理服务器	
<code>IS_DEMO</code>	设置为 <code>true</code> 以启用演示模式：从 UI 隐藏电子邮件和组织，跳过入职，隐藏内部命令。对于流式传输或录制会话很有用	
<code>MAX_MCP_OUTPUT_TOKENS</code>	MCP 工具响应中允许的最大令牌数。当输出超过 10,000 个令牌时，Claude Code 显示警告（默认：25000）	
<code>MAX_THINKING_TOKENS</code>	覆盖 扩展思考 令牌预算。思考默认在最大预算（31,999 个令牌）处启用。使用此选项限制预算（例如 <code>MAX_THINKING_TOKENS=10000</code> ）或完全禁用思考（ <code>MAX_THINKING_TOKENS=0</code> ）。对于 Opus 4.6，思考深度由 努力级别 控制，此变量被忽略，除非设置为 <code>0</code> 以禁用思考。	
<code>MCP_CLIENT_SECRET</code>	需要 预配置凭证 的 MCP servers 的 OAuth 客户端密钥。在添加具有 <code>--client-secret</code> 的服务器时避免交互式提示	
<code>MCP_OAUTH_CALLBACK_PORT</code>	OAuth 重定向回调的固定端口，作为在添加具有 预配置凭证 的 MCP server 时 <code>--callback-port</code> 的替代方案	

变量	目的	
<code>MCP_TIMEOUT</code>	MCP 服务器启动的超时（以毫秒为单位）	
<code>MCP_TOOL_TIMEOUT</code>	MCP 工具执行的超时（以毫秒为单位）	
<code>NO_PROXY</code>	域和 IP 列表，对其的请求将直接发出，绕过代理	
<code>SLASH_COMMAND_TOOL_CHARACTER_BUDGET</code>	覆盖为 Skill 工具 显示的技能元数据的字符预算。预算在上下文窗口的 2% 处动态缩放，回退为 16,000 个字符。为了向后兼容保留旧名称	
<code>USE_BUILTIN_RIPGREP</code>	设置为 <code>0</code> 以使用系统安装的 <code>rg</code> 而不是 Claude Code 附带的 <code>rg</code>	
<code>VERTEX_REGION_CLAUDE_3_5_HAIKU</code>	使用 Vertex AI 时覆盖 Claude 3.5 Haiku 的区域	
<code>VERTEX_REGION_CLAUDE_3_7_SONNET</code>	使用 Vertex AI 时覆盖 Claude 3.7 Sonnet 的区域	
<code>VERTEX_REGION_CLAUDE_4_0_OPUS</code>	使用 Vertex AI 时覆盖 Claude 4.0 Opus 的区域	
<code>VERTEX_REGION_CLAUDE_4_0_SONNET</code>	使用 Vertex AI 时覆盖 Claude 4.0 Sonnet 的区域	
<code>VERTEX_REGION_CLAUDE_4_1_OPUS</code>	使用 Vertex AI 时覆盖 Claude 4.1 Opus 的区域	

Claude 可用的工具

Claude Code 可以访问一组强大的工具，帮助它理解和修改您的代码库：

工具	描述	需要权限
Agent	生成一个 subagent ，具有自己的上下文窗口来处理任务	否
AskUserQuestion	提出多选问题以收集需求或澄清歧义	否

工具	描述	需要权限
Bash	在您的环境中执行 shell 命令。请参阅 Bash 工具行为	是
CronCreate	在当前会话中计划重复或一次性提示（Claude 退出时消失）。请参阅 计划任务	否
CronDelete	按 ID 取消计划任务	否
CronList	列出会话中的所有计划任务	否
Edit	对特定文件进行有针对性的编辑	是
EnterPlanMode	切换到计划模式以在编码前设计方法	否
EnterWorktree	创建一个隔离的 git worktree 并切换到它	否
ExitPlanMode	提出计划以供批准并退出计划模式	是
ExitWorktree	退出 worktree 会话并返回到原始目录	否
Glob	基于模式匹配查找文件	否
Grep	在文件内容中搜索模式	否
ListMcpResourcesTool	列出连接的 MCP servers 公开的资源	否
LSP	通过语言服务器的代码智能。在文件编辑后自动报告类型错误和警告。还支持导航操作：跳转到定义、查找引用、获取类型信息、列出符号、查找实现、跟踪调用层次结构。需要 code intelligence plugin 及其语言服务器二进制文件	否
NotebookEdit	修改 Jupyter notebook 单元格	是
Read	读取文件的内容	否
ReadMcpResourceTool	按 URI 读取特定 MCP 资源	否
Skill	在主对话中执行 skill	是

工具	描述	需要权限
TaskCreate	在任务列表中创建新任务	否
TaskGet	检索特定任务的完整详情	否
TaskList	列出所有任务及其当前状态	否
TaskOutput	从后台任务检索输出	否
TaskStop	按 ID 杀死运行中的后台任务	否
TaskUpdate	更新任务状态、依赖项、详情或删除任务	否
TodoWrite	管理会话任务清单。在非交互模式和 Agent SDK 中可用；交互式会话改用 TaskCreate、TaskGet、TaskList 和 TaskUpdate	否
ToolSearch	当启用 tool search 时搜索和加载延迟工具	否
WebFetch	从指定的 URL 获取内容	是
WebSearch	执行网络搜索	是
Write	创建或覆盖文件	是

权限规则可以使用 `/allowed-tools` 或在[权限设置](#)中配置。另请参阅[工具特定权限规则](#)。

Bash 工具行为

Bash 工具使用以下持久性行为执行 shell 命令：

- **工作目录持续：**当 Claude 更改工作目录（例如 `cd /path/to/dir`）时，后续 Bash 命令将在该目录中执行。您可以使用 `CLAUDE_BASH_MAINTAIN_PROJECT_WORKING_DIR=1` 在每个命令后重置为项目目录。
- **环境变量不持续：**在一个 Bash 命令中设置的环境变量（例如 `export MY_VAR=value`）不在后续 Bash 命令中可用。每个 Bash 命令在新的 shell 环境中运行。

要使环境变量在 Bash 命令中可用，您有**三个选项**：

选项 1：在启动 Claude Code 前激活环境（最简单的方法）

在启动 Claude Code 前在您的终端中激活您的虚拟环境：

```
conda activate myenv
## 或: source /path/to/venv/bin/activate
claude
```

这适用于 shell 环境，但在 Claude 的 Bash 命令中设置的环境变量不会在命令之间持续。

选项 2：在启动 Claude Code 前设置 `CLAUDE_ENV_FILE`（持久环境设置）

导出包含您的环境设置的 shell 脚本的路径：

```
export CLAUDE_ENV_FILE=/path/to/env-setup.sh
claude
```

其中 `/path/to/env-setup.sh` 包含：

```
conda activate myenv
## 或: source /path/to/venv/bin/activate
## 或: export MY_VAR=value
```

Claude Code 将在每个 Bash 命令前获取此文件，使环境在所有命令中持续。

选项 3：使用 `SessionStart` hook（项目特定配置）

在 `.claude/settings.json` 中配置：

```
{
  "hooks": {
    "SessionStart": [{
      "matcher": "startup",
      "hooks": [{
        "type": "command",
        "command": "echo 'conda activate myenv' >> \"\$CLAUDE_ENV_FILE\""
      }]
    }]
  }
}
```

hook 写入 `$CLAUDE_ENV_FILE`，然后在每个 Bash 命令前获取。这对于团队共享的项目配置是理想的。

请参阅 [SessionStart hooks](#) 了解有关选项 3 的更多详情。

使用 hooks 扩展工具

您可以使用 [Claude Code hooks](#) 在任何工具执行前或后运行自定义命令。

例如，您可以在 Claude 修改 Python 文件后自动运行 Python 格式化程序，或通过阻止对某些路径的 Write 操作来防止修改生产配置文件。

另请参阅

- [权限](#)：权限系统、规则语法、工具特定模式和 managed 策略
- [身份验证](#)：设置用户对 Claude Code 的访问
- [故障排除](#)：常见配置问题的解决方案

模型配置

了解 Claude Code 模型配置，包括模型别名如 `opusplan`

可用模型

对于 Claude Code 中的 `model` 设置，你可以配置以下任一项：

- 一个模型别名
- 一个模型名称
 - Anthropic API：完整的**模型名称**
 - Bedrock：推理配置文件 ARN
 - Foundry：部署名称
 - Vertex：版本名称

模型别名

模型别名提供了一种便捷的方式来选择模型设置，无需记住确切的版本号：

模型别名	行为
<code>default</code>	推荐的模型设置，取决于你的账户类型
<code>sonnet</code>	使用最新的 Sonnet 模型（当前为 Sonnet 4.6）用于日常编码任务
<code>opus</code>	使用最新的 Opus 模型（当前为 Opus 4.6）用于复杂推理任务
<code>haiku</code>	使用快速高效的 Haiku 模型用于简单任务
<code>sonnet[1m]</code>	使用 Sonnet 和100 万 token 上下文窗口用于长会话
<code>opusplan</code>	特殊模式，在计划模式中使用 <code>opus</code> ，然后在执行时切换到 <code>sonnet</code>

别名始终指向最新版本。要固定到特定版本，请使用完整模型名称（例如 `claude-opus-4-6`）或设置相应环境变量，如 `ANTHROPIC_DEFAULT_OPUS_MODEL`。

设置你的模型

你可以通过多种方式配置模型，按优先级顺序列出：

1. 在会话期间 - 使用 `/model <alias|name>` 在会话中切换模型

2. 启动时 - 使用 `claude --model <alias|name>` 启动
3. 环境变量 - 设置 `ANTHROPIC_MODEL=<alias|name>`
4. 设置 - 在设置文件中使用 `model` 字段永久配置。

使用示例：

```
## 使用 Opus 启动
claude --model opus

## 在会话期间切换到 Sonnet
/model sonnet
```

设置文件示例：

```
{
  "permissions": {
    ...
  },
  "model": "opus"
}
```

限制模型选择

企业管理员可以在[托管或策略设置](#)中使用 `availableModels` 来限制用户可以选择的模型。

设置 `availableModels` 后，用户无法通过 `/model`、`--model` 标志、Config 工具或 `ANTHROPIC_MODEL` 环境变量切换到列表中不存在的模型。

```
{
  "availableModels": ["sonnet", "haiku"]
}
```

默认模型行为

模型选择器中的”默认”选项不受 `availableModels` 影响。它始终保持可用，并代表系统的运行时默认值[基于用户的订阅层级](#)。

即使使用 `availableModels: []`，用户仍然可以使用其层级的默认模型来使用 Claude Code。

控制用户运行的模型

要完全控制模型体验，请将 `availableModels` 与 `model` 设置一起使用：

- **availableModels**：限制用户可以切换到的内容
- **model**：设置显式模型覆盖，优先于默认值

此示例确保所有用户运行 Sonnet 4.6，并且只能在 Sonnet 和 Haiku 之间选择：

```
{
  "model": "sonnet",
  "availableModels": ["sonnet", "haiku"]
}
```

合并行为

当 `availableModels` 在多个级别设置时，例如用户设置和项目设置，数组会被合并并去重。要强制执行严格的允许列表，请在托管或策略设置中设置 `availableModels`，这具有最高优先级。

特殊模型行为

`default` 模型设置

`default` 的行为取决于你的账户类型：

- **Max 和 Team Premium**：默认为 Opus 4.6
- **Pro 和 Team Standard**：默认为 Sonnet 4.6
- **Enterprise**：Opus 4.6 可用但不是默认值

如果你在使用 Opus 时达到使用阈值，Claude Code 可能会自动回退到 Sonnet。

`opuslan` 模型设置

`opuslan` 模型别名提供了一种自动化的混合方法：

- **在计划模式中** - 使用 `opus` 进行复杂推理和架构决策
- **在执行模式中** - 自动切换到 `sonnet` 进行代码生成和实现

这为你提供了两全其美的方案：Opus 的卓越推理能力用于规划，Sonnet 的效率用于执行。

调整努力级别

努力级别控制自适应推理，根据任务复杂性动态分配思考。较低的努力级别对于直接任务更快更便宜，而较高的努力级别为复杂问题提供更深入的推理。

有三个级别可用：**low**、**medium** 和 **high**。Opus 4.6 对于 Max 和 Team 订阅者默认为中等努力级别。

设置努力级别：

- 在 `/model` 中：选择模型时使用左右箭头键调整努力级别滑块
- 环境变量：设置 `CLAUDE_CODE_EFFORT_LEVEL=low|medium|high`
- 设置：在设置文件中设置 `effortLevel`

Opus 4.6 和 Sonnet 4.6 支持努力级别。当选择支持的模型时，努力级别滑块会出现在 `/model` 中。当前努力级别也会显示在徽标和旋转器旁边（例如“with low effort”），因此你可以确认哪个设置处于活动状态，而无需打开 `/model`。

要在 Opus 4.6 和 Sonnet 4.6 上禁用自适应推理并恢复到之前的固定思考预算，请设置 `CLAUDE_CODE_DISABLE_ADAPTIVE_THINKING=1`。禁用时，这些模型使用由 `MAX_THINKING_TOKENS` 控制的固定预算。请参阅[环境变量](#)。

扩展上下文

Opus 4.6 和 Sonnet 4.6 支持[100 万 token 上下文窗口](#)用于包含大型代码库的长会话。

Note:

100 万 token 上下文窗口目前处于测试阶段。功能、定价和可用性可能会改变。

扩展上下文可用于：

- **API 和按使用量付费用户**：完全访问 100 万 token 上下文
- **Pro、Max、Teams 和 Enterprise 订阅者**：可通过[额外使用](#)获得

要完全禁用 100 万 token 上下文，请设置 `CLAUDE_CODE_DISABLE_1M_CONTEXT=1`。这会从模型选择器中删除 100 万 token 模型变体。请参阅[环境变量](#)。

选择 100 万 token 模型不会立即改变计费。你的会话使用标准费率，直到超过 200K token 的上下文。超过 200K token 后，请求按[长上下文定价](#)收费，具有专用[速率限制](#)。对于订阅者，超过 200K 的 token 作为额外使用而不是通过订阅计费。

如果你的账户支持 100 万 token 上下文，该选项会出现在最新版本 Claude Code 的模型选择器（`/model`）中。如果你看不到它，请尝试重新启动你的会话。

你也可以将 `[1m]` 后缀与模型别名或完整模型名称一起使用：

```
## 使用 sonnet[1m] 别名
/model sonnet[1m]

## 或将 [1m] 附加到完整模型名称
/model claude-sonnet-4-6[1m]
```

检查你的当前模型

你可以通过多种方式查看你当前使用的模型：

- 1. 在状态行中（如果已配置）
- 2. 在 /status 中，它也显示你的账户信息。

环境变量

你可以使用以下环境变量，这些变量必须是完整的模型名称（或你的 API 提供商的等效项），以控制别名映射到的模型名称。

环境变量	描述
ANTHROPIC_DEFAULT_OPUS_MODEL	用于 opus 的模型，或在 Plan Mode 活跃时用于 opusplan 的模型。
ANTHROPIC_DEFAULT_SONNET_MODEL	用于 sonnet 的模型，或在 Plan Mode 不活跃时用于 opusplan 的模型。
ANTHROPIC_DEFAULT_HAIKU_MODEL	用于 haiku 的模型，或用于后台功能
CLAUDE_CODE_SUBAGENT_MODEL	用于 subagents 的模型

注意：ANTHROPIC_SMALL_FAST_MODEL 已弃用，改为使用 ANTHROPIC_DEFAULT_HAIKU_MODEL 。

为第三方部署固定模型

通过 Bedrock 、 Vertex AI 或 Foundry 部署 Claude Code 时，在向用户推出前固定模型版本。

不固定模型，Claude Code 会使用模型别名（ sonnet 、 opus 、 haiku ），这些别名会解析为最新版本。当 Anthropic 发布新模型时，其账户未启用新版本的用户将无声地中断。

Warning:

在初始设置中将所有三个模型环境变量设置为特定版本 ID。跳过此步骤意味着 Claude Code 更新可能会在你没有任何操作的情况下破坏你的用户。

对你的提供商使用以下环境变量和特定版本的模型 ID：

提供商	示例
Bedrock	<code>export ANTHROPIC_DEFAULT_OPUS_MODEL='us.anthropic.claude-opus-4-6-v1'</code>
Vertex AI	<code>export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-6'</code>
Foundry	<code>export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-6'</code>

对 `ANTHROPIC_DEFAULT_SONNET_MODEL` 和 `ANTHROPIC_DEFAULT_HAIKU_MODEL` 应用相同的模式。有关所有提供商的当前和旧版模型 ID，请参阅[模型概览](#)。要将用户升级到新模型版本，请更新这些环境变量并重新部署。

Note:

使用第三方提供商时，`settings.availableModels` 允许列表仍然适用。过滤匹配模型别名（`opus`、`sonnet`、`haiku`），而不是提供商特定的模型 ID。

按版本覆盖模型 ID

上面的家族级环境变量为每个家族别名配置一个模型 ID。如果你需要将同一家族中的多个版本映射到不同的提供商 ID，请改用 `modelOverrides` 设置。

`modelOverrides` 将单个 Anthropic 模型 ID 映射到 Claude Code 发送给你的提供商 API 的提供商特定字符串。当用户在 `/model` 选择器中选择映射的模型时，Claude Code 使用你配置的值而不是内置默认值。

这让企业管理员可以将每个模型版本路由到特定的 Bedrock 推理配置文件 ARN、Vertex AI 版本名称或 Foundry 部署名称，用于治理、成本分配或区域路由。

在你的[设置文件](#)中设置 `modelOverrides`：

```
{
  "modelOverrides": {
    "claude-opus-4-6": "arn:aws:bedrock:us-east-2:123456789012:application-
inference-profile/opus-prod",
    "claude-opus-4-5-20251101": "arn:aws:bedrock:us-
east-2:123456789012:application-inference-profile/opus-45-prod",
    "claude-sonnet-4-6": "arn:aws:bedrock:us-east-2:123456789012:application-
inference-profile/sonnet-prod"
  }
}
```

键必须是[模型概览](#)中列出的 Anthropic 模型 ID。对于带日期的模型 ID，请包含日期后缀，完全按照其显示的方式。未知的键会被忽略。

覆盖替换了支持 `/model` 选择器中每个条目的内置模型 ID。在 Bedrock 上，覆盖优先于 Claude Code 在启动时自动发现的任何推理配置文件。你通过 `ANTHROPIC_MODEL`、`--model` 或 `ANTHROPIC_DEFAULT_*_MODEL` 环境变量直接提供的值会按原样传递给提供商，不会被 `modelOverrides` 转换。

`modelOverrides` 与 `availableModels` 一起工作。允许列表针对 Anthropic 模型 ID 进行评估，而不是覆盖值，因此 `availableModels` 中的条目如 `"opus"` 即使 Opus 版本映射到 ARN 也会继续匹配。

Prompt caching 配置

Claude Code 自动使用 [prompt caching](#) 来优化性能并降低成本。你可以全局禁用 prompt caching 或针对特定模型层级禁用：

环境变量	描述
<code>DISABLE_PROMPT_CACHING</code>	设置为 <code>1</code> 以禁用所有模型的 prompt caching（优先于按模型设置）
<code>DISABLE_PROMPT_CACHING_HAIKU</code>	设置为 <code>1</code> 以仅禁用 Haiku 模型的 prompt caching
<code>DISABLE_PROMPT_CACHING_SONNET</code>	设置为 <code>1</code> 以仅禁用 Sonnet 模型的 prompt caching
<code>DISABLE_PROMPT_CACHING_OPUS</code>	设置为 <code>1</code> 以仅禁用 Opus 模型的 prompt caching

这些环境变量为你提供了对 prompt caching 行为的细粒度控制。全局

`DISABLE_PROMPT_CACHING` 设置优先于特定模型的设置，允许你在需要时快速禁用所有缓存。按模型的设置对于选择性控制很有用，例如在调试特定模型或与可能具有不同缓存实现的云提供商合作时。

输出样式

将 Claude Code 适配用于软件工程之外的用途

输出样式允许你将 Claude Code 用作任何类型的代理，同时保留其核心功能，例如运行本地脚本、读取/写入文件和跟踪 TODO。

内置输出样式

Claude Code 的默认输出样式是现有的系统提示，旨在帮助你高效地完成软件工程任务。

还有两种额外的内置输出样式，专注于教你了解代码库和 Claude 的运作方式：

- **Explanatory**：在帮助你完成软件工程任务的同时提供教育性的” Insights”。帮助你理解实现选择和代码库模式。
- **Learning**：协作式的边学边做模式，Claude 不仅会在编码时分享” Insights”，还会要求你自己贡献小的、战略性的代码片段。Claude Code 将在你的代码中添加 `TODO(human)` 标记供你实现。

输出样式如何工作

输出样式直接修改 Claude Code 的系统提示。

- 所有输出样式都排除了高效输出的说明（例如简洁回复）。
- 自定义输出样式排除了编码说明（例如用测试验证代码），除非 `keep-coding-instructions` 为 true。
- 所有输出样式都在系统提示的末尾添加了自己的自定义说明。
- 所有输出样式都会在对话期间触发提醒，让 Claude 遵守输出样式说明。

更改你的输出样式

运行 `/config` 并选择**输出样式**从菜单中选择一种样式。你的选择会保存到本地项目级别的 `.claude/settings.local.json`。

要在不使用菜单的情况下设置样式，直接编辑设置文件中的 `outputStyle` 字段：


```
{
  "outputStyle": "Explanatory"
}
```

由于输出样式是在会话开始时在系统提示中设置的，更改将在你下次启动新会话时生效。这使系统提示在整个对话中保持稳定，以便 prompt caching 可以降低延迟和成本。

创建自定义输出样式

自定义输出样式是包含 frontmatter 和将添加到系统提示的文本的 Markdown 文件：

```
---
name: My Custom Style
description:
  A brief description of what this style does, to be displayed to the user
---

## Custom Style Instructions

You are an interactive CLI tool that helps users with software engineering
tasks. [Your custom instructions here...]

### Specific Behaviors

[Define how the assistant should behave in this style...]
```

你可以在用户级别（`~/.claude/output-styles`）或项目级别（`.claude/output-styles`）保存这些文件。

Frontmatter

输出样式文件支持 frontmatter 来指定元数据：

Frontmatter	目的	默认值
<code>name</code>	输出样式的名称，如果不是文件名	从文件名继承
<code>description</code>	输出样式的描述，在 <code>/config</code> 选择器中显示	无

Frontmatter	目的	默认值
<code>keep-coding-instructions</code>	是否保留 Claude Code 系统提示中与编码相关的部分。	false

与相关功能的比较

输出样式 vs. CLAUDE.md vs. --append-system-prompt

输出样式完全”关闭”了 Claude Code 默认系统提示中特定于软件工程的部分。CLAUDE.md 和 `--append-system-prompt` 都不会编辑 Claude Code 的默认系统提示。CLAUDE.md 将内容作为用户消息添加到 Claude Code 默认系统提示_之后_。 `--append-system-prompt` 将内容附加到系统提示。

输出样式 vs. Agents

输出样式直接影响主代理循环，仅影响系统提示。Agents 被调用来处理特定任务，可以包括额外的设置，如要使用的模型、可用的工具以及有关何时使用代理的一些上下文。

输出样式 vs. Skills

输出样式修改 Claude 的响应方式（格式、语气、结构），一旦选择就始终处于活动状态。Skills 是特定于任务的提示，你可以使用 `/skill-name` 调用或 Claude 在相关时自动加载。使用输出样式来实现一致的格式化偏好；使用 skills 来实现可重用的工作流和任务。

使用快速模式加快响应速度

通过切换快速模式在 Claude Code 中获得更快的 Opus 4.6 响应。

Note:

快速模式处于[研究预览](#)阶段。该功能、定价和可用性可能会根据反馈而改变。

快速模式是 Claude Opus 4.6 的高速配置，使模型速度提高 2.5 倍，但每个令牌的成本更高。当您需要快速进行交互式工作（如快速迭代或实时调试）时，使用 `/fast` 切换它打开，当成本比延迟更重要时，将其切换关闭。

快速模式不是一个不同的模型。它使用相同的 Opus 4.6，但采用不同的 API 配置，优先考虑速度而不是成本效率。您获得相同的质量和功能，只是响应速度更快。

Note:

快速模式需要 Claude Code v2.1.36 或更高版本。使用 `claude --version` 检查您的版本。

需要了解的内容：

- 使用 `/fast` 在 Claude Code CLI 中切换快速模式。也可通过 Claude Code VS Code 扩展中的 `/fast` 使用。
- Opus 4.6 快速模式定价从 \$30/150 MTok 开始。快速模式在所有计划上享受 50% 折扣，直到太平洋时间 2 月 16 日晚上 11:59。
- 可供订阅计划（Pro/Max/Team/Enterprise）上的所有 Claude Code 用户和 Claude Console 使用。
- 对于订阅计划（Pro/Max/Team/Enterprise）上的 Claude Code 用户，快速模式仅通过额外使用提供，不包含在订阅速率限制中。

本页涵盖如何[切换快速模式](#)、其[成本权衡](#)、[何时使用](#)、[要求](#)、[每个会话选择加入](#)和[速率限制行为](#)。

切换快速模式

通过以下任一方式切换快速模式：

- 输入 `/fast` 并按 Tab 键切换打开或关闭

- 在您的[用户设置文件](#)中设置 `"fastMode": true`

默认情况下，快速模式在会话之间保持。管理员可以配置快速模式在每个会话时重置。有关详细信息，请参阅[要求每个会话选择加入](#)。

为了获得最佳成本效率，在会话开始时启用快速模式，而不是在对话中途切换。有关详细信息，请参阅[了解成本权衡](#)。

启用快速模式时：

- 如果您使用的是不同的模型，Claude Code 会自动切换到 Opus 4.6
- 您将看到确认消息：“Fast mode ON”
- 快速模式活跃时，提示旁边会出现一个小的  图标
- 随时再次运行 `/fast` 以检查快速模式是否打开或关闭

使用 `/fast` 再次禁用快速模式时，您仍然保持在 Opus 4.6 上。模型不会恢复到您之前的模型。要切换到不同的模型，请使用 `/model`。

了解成本权衡

快速模式的每个令牌定价高于标准 Opus 4.6：

模式	输入 (MTok)	输出 (MTok)
Opus 4.6 上的快速模式 (<200K)	\$30	\$150
Opus 4.6 上的快速模式 (>200K)	\$60	\$225

快速模式与 1M 令牌扩展上下文窗口兼容。

当您在对话中途切换到快速模式时，您需要为整个对话上下文支付完整的快速模式未缓存输入令牌价格。这比从一开始就启用快速模式的成本更高。

决定何时使用快速模式

快速模式最适合响应延迟比成本更重要的交互式工作：

- 快速迭代代码更改
- 实时调试会话
- 时间敏感的工作，有紧迫的截止日期

标准模式更适合：

- 速度不那么重要的长期自主任务
- 批处理或 CI/CD 管道

- 成本敏感的工作负载

快速模式与努力级别

快速模式和努力级别都会影响响应速度，但方式不同：

设置	效果
快速模式	相同的模型质量，更低的延迟，更高的成本
较低的努力级别	更少的思考时间，更快的响应，在复杂任务上可能质量较低

您可以结合两者：在直接任务上使用快速模式和较低的努力级别以获得最大速度。

要求

快速模式需要以下所有条件：

- **第三方云提供商上不可用：**快速模式在 Amazon Bedrock、Google Vertex AI 或 Microsoft Azure Foundry 上不可用。快速模式可通过 Anthropic Console API 和使用额外使用的 Claude 订阅计划获得。
- **启用额外使用：**您的账户必须启用额外使用，这允许在您的计划包含的使用之外进行计费。对于个人账户，在您的 [Console 计费设置](#) 中启用此功能。对于团队和企业，管理员必须为组织启用额外使用。

Note:

快速模式使用直接计入额外使用，即使您的计划上还有剩余使用。这意味着快速模式令牌不计入您的计划包含的使用，并从第一个令牌开始按快速模式费率收费。

- **团队和企业的管理员启用：**快速模式默认对团队和企业组织禁用。管理员必须明确 [启用快速模式](#)，用户才能访问它。

Note:

如果您的管理员尚未为您的组织启用快速模式，`/fast` 命令将显示”Fast mode has been disabled by your organization.”

为您的组织启用快速模式

管理员可以在以下位置启用快速模式：

- **Console（API 客户）：**[Claude Code 偏好设置](#)

- **Claude AI**（团队和企业）：[管理员设置 > Claude Code](#)

另一个完全禁用快速模式的选项是设置 `CLAUDE_CODE_DISABLE_FAST_MODE=1`。请参阅[环境变量](#)。

要求每个会话选择加入

默认情况下，快速模式在会话之间保持：如果用户启用快速模式，它在未来的会话中保持打开。[团队](#)或[企业](#)计划上的管理员可以通过在[托管设置](#)或[服务器托管设置](#)中将

`fastModePerSessionOptIn` 设置为 `true` 来防止这种情况。这会导致每个会话以快速模式关闭开始，要求用户使用 `/fast` 明确启用它。

```
{
  "fastModePerSessionOptIn": true
}
```

这对于在用户运行多个并发会话的组织中控制成本很有用。用户在需要速度时仍然可以使用 `/fast` 启用快速模式，但它在每个新会话开始时重置。用户的快速模式偏好仍然被保存，因此删除此设置会恢复默认的持久行为。

处理速率限制

快速模式与标准 Opus 4.6 有单独的速率限制。当您达到快速模式速率限制或用完额外使用额度时：

1. 快速模式自动回退到标准 Opus 4.6
2.  图标变灰以指示冷却
3. 您继续以标准速度和定价工作
4. 冷却过期时，快速模式自动重新启用

要手动禁用快速模式而不是等待冷却，请再次运行 `/fast`。

研究预览

快速模式是一个研究预览功能。这意味着：

- 该功能可能会根据反馈而改变
- 可用性和定价可能会改变
- 底层 API 配置可能会演变

通过您通常的 Anthropic 支持渠道报告问题或反馈。

另请参阅

- [模型配置](#)：切换模型并调整努力级别
- [有效管理成本](#)：跟踪令牌使用情况并降低成本
- [状态行配置](#)：显示模型和上下文信息

有效管理成本

跟踪令牌使用情况，设置团队支出限制，并通过上下文管理、模型选择、扩展思考设置和预处理 hooks 来降低 Claude Code 成本。

Claude Code 在每次交互中消耗令牌。成本因代码库大小、查询复杂性和对话长度而异。平均成本为每个开发者每天 \$6，90% 的用户每日成本保持在 \$12 以下。

对于团队使用，Claude Code 按 API 令牌消耗收费。平均而言，使用 Sonnet 4.6 的 Claude Code 成本约为每个开发者每月 \$100-200，但根据用户运行的实例数量以及是否在自动化中使用，成本差异很大。

本页面介绍如何[跟踪成本](#)、[管理团队成本](#)和[减少令牌使用](#)。

跟踪成本

使用 `/cost` 命令

Note:

`/cost` 命令显示 API 令牌使用情况，适用于 API 用户。Claude Max 和 Pro 订阅者的使用情况包含在订阅中，因此 `/cost` 数据与计费无关。订阅者可以使用 `/stats` 查看使用模式。

`/cost` 命令为您的当前会话提供详细的令牌使用统计：

```
Total cost:           $0.55
Total duration (API):  6m 19.7s
Total duration (wall): 6h 33m 10.2s
Total code changes:    0 lines added, 0 lines removed
```

管理团队成本

使用 Claude API 时，您可以在 Claude Console 上[设置工作区支出限制](#)，以限制总体 Claude Code 工作区支出。管理员可以在 Console 中[查看成本和使用情况报告](#)。

Note:

当您首次使用 Claude Console 账户对 Claude Code 进行身份验证时，会自动为您创建一个名为“Claude Code”的工作区。此工作区为您的组织中的所有 Claude Code 使用情况提供集中式成本跟踪和管理。您无法为此工作区创建 API 密钥；它专门用于 Claude Code 身份验证和使用。

在 Bedrock、Vertex 和 Foundry 上，Claude Code 不会从您的云中发送指标。为了获取成本指标，几家大型企业报告使用 [LiteLLM](#)，这是一个开源工具，可帮助公司[按密钥跟踪支出](#)。此项目与 Anthropic 无关，尚未进行安全审计。

速率限制建议

为团队设置 Claude Code 时，请根据您的组织规模考虑这些每用户的令牌/分钟 (TPM) 和请求/分钟 (RPM) 建议：

团队规模	每用户 TPM	每用户 RPM
1-5 用户	200k-300k	5-7
5-20 用户	100k-150k	2.5-3.5
20-50 用户	50k-75k	1.25-1.75
50-100 用户	25k-35k	0.62-0.87
100-500 用户	15k-20k	0.37-0.47
500+ 用户	10k-15k	0.25-0.35

例如，如果您有 200 个用户，您可能会为每个用户请求 20k TPM，或总共 400 万 TPM (200*20,000 = 400 万)。

随着团队规模的增长，每用户的 TPM 会减少，因为在较大的组织中，往往较少的用户同时使用 Claude Code。这些速率限制在组织级别应用，而不是按个人用户应用，这意味着当其他人未积极使用该服务时，个人用户可以暂时消耗超过其计算份额的资源。

Note:

如果您预期会出现异常高的并发使用情况（例如与大型团体进行的实时培训会话），您可能需要更高的每用户 TPM 分配。

Agent 团队令牌成本

[Agent 团队](#)生成多个 Claude Code 实例，每个实例都有自己的上下文窗口。令牌使用情况随活跃队友的数量和每个队友运行的时间长度而扩展。

为了保持 agent 团队成本可控：

- 为队友使用 Sonnet。它为协调任务平衡了能力和成本。
- 保持团队规模小。每个队友运行自己的上下文窗口，因此令牌使用大致与团队规模成正比。
- 保持生成提示的重点。队友会自动加载 CLAUDE.md、MCP servers 和 skills，但生成提示中的所有内容都会从一开始就添加到其上下文中。
- 工作完成后清理团队。活跃的队友即使处于空闲状态也会继续消耗令牌。
- Agent 团队默认被禁用。在您的 `settings.json` 或环境中设置 `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1` 以启用它们。请参阅[启用 agent 团队](#)。

减少令牌使用

令牌成本随上下文大小而扩展：Claude 处理的上下文越多，您使用的令牌就越多。Claude Code 通过 prompt caching（减少重复内容（如系统提示）的成本）和 auto-compact（在接近上下文限制时总结对话历史）自动优化成本。

以下策略可帮助您保持上下文较小并降低每条消息的成本。

主动管理上下文

使用 `/cost` 检查您当前的令牌使用情况，或[配置您的状态行](#)以连续显示它。

- **在任务之间清除：**使用 `/clear` 在切换到不相关的工作时重新开始。陈旧的上下文会在随后的每条消息上浪费令牌。在清除之前使用 `/rename` 以便您稍后可以轻松找到会话，然后使用 `/resume` 返回到它。
- **添加自定义 compaction 指令：**`/compact Focus on code samples and API usage` 告诉 Claude 在总结期间保留什么。

您还可以在 CLAUDE.md 中自定义 compaction 行为：

```
## Compact instructions
```

```
When you are using compact, please focus on test output and code changes
```

选择正确的模型

Sonnet 处理大多数编码任务效果很好，成本低于 Opus。为复杂的架构决策或多步推理保留 Opus。使用 `/model` 在会话中途切换模型，或在 `/config` 中设置默认值。对于简单的 subagent 任务，在您的 [subagent 配置](#) 中指定 `model: haiku`。

减少 MCP server 开销

每个 MCP server 都会向您的上下文添加工具定义，即使处于空闲状态。运行 `/context` 查看占用空间的内容。

- **在可用时优先使用 CLI 工具：** `gh`、`aws`、`gcloud` 和 `sentry-cli` 等工具比 MCP servers 更节省上下文，因为它们不添加持久工具定义。Claude 可以直接运行 CLI 命令，无需开销。
- **禁用未使用的 servers：** 运行 `/mcp` 查看配置的 servers 并禁用您未积极使用的任何 servers。
- **工具搜索是自动的：** 当 MCP 工具描述超过您的上下文窗口的 10% 时，Claude Code 会自动延迟它们并通过 [工具搜索](#) 按需加载工具。由于延迟的工具仅在实际使用时进入上下文，较低的阈值意味着较少的空闲工具定义消耗空间。使用 `ENABLE_TOOL_SEARCH=auto:<N>` 设置较低的阈值（例如，`auto:5` 在工具超过您的上下文窗口的 5% 时触发）。

为类型化语言安装代码智能插件

[代码智能插件](#) 为 Claude 提供精确的符号导航，而不是基于文本的搜索，减少在探索不熟悉的代码时不必要的文件读取。单个“转到定义”调用替代了可能需要的 `grep` 后跟读取多个候选文件。已安装的语言服务器还会在编辑后自动报告类型错误，因此 Claude 无需运行编译器即可捕获错误。

将处理卸载到 hooks 和 skills

自定义 [hooks](#) 可以在 Claude 看到数据之前对其进行预处理。Claude 不是读取 10,000 行日志文件来查找错误，hook 可以 `grep ERROR` 并仅返回匹配的行，将上下文从数万个令牌减少到数百个。

[skill](#) 可以为 Claude 提供领域知识，这样它就不必进行探索。例如，“codebase-overview” skill 可以描述您的项目架构、关键目录和命名约定。当 Claude 调用该 skill 时，它会立即获得此上下文，而不是花费令牌读取多个文件来理解结构。

例如，此 PreToolUse hook 过滤测试输出以仅显示失败：

settings.json

将此添加到您的 [settings.json](#) 以在每个 Bash 命令之前运行 hook：

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "~/.claude/hooks/filter-test-output.sh"
          }
        ]
      }
    ]
  }
}
```

filter-test-output.sh

hook 调用此脚本，该脚本检查命令是否为测试运行器并修改它以仅显示失败：

```
#!/bin/bash
input=$(cat)
cmd=$(echo "$input" | jq -r '.tool_input.command')

## If running tests, filter to show only failures
if [[ "$cmd" =~ ^(npm test|pytest|go test) ]]; then
  filtered_cmd="$cmd 2>&1 | grep -A 5 -E '(FAIL|ERROR|error:)' | head -100"
  echo "{\"hookSpecificOutput\":{\"hookEventName\":\"PreToolUse\",\"permissionDecision\":\"allow\",\"updatedInput\":{\"command\":\"$filtered_cmd\"}}}"
else
  echo "{}"
fi
```

将指令从 CLAUDE.md 移动到 skills

您的 [CLAUDE.md](#) 文件在会话开始时加载到上下文中。如果它包含特定工作流的详细指令（如 PR 审查或数据库迁移），即使您在做不相关的工作时，这些令牌也会存在。[Skills](#) 仅在调用时按需加载，因此将专门指令移动到 skills 中可以保持您的基础上下文较小。目标是通过仅包含必要内容来将 CLAUDE.md 保持在约 500 行以下。

调整扩展思考

扩展思考默认启用，预算为 31,999 个令牌，因为它显著改进了复杂规划和推理任务的性能。但是，思考令牌作为输出令牌计费，因此对于不需要深度推理的更简单任务，您可以通过在 `/model` 中为 Opus 4.6 降低**努力级别**、在 `/config` 中禁用思考或降低预算（例如，`MAX_THINKING_TOKENS=8000`）来降低成本。

将冗长的操作委托给 subagents

运行测试、获取文档或处理日志文件可能会消耗大量上下文。将这些委托给 [subagents](#)，以便冗长的输出保留在 subagent 的上下文中，而只有摘要返回到您的主对话。

管理 agent 团队成本

当队友在 plan mode 中运行时，Agent 团队使用的令牌大约是标准会话的 7 倍，因为每个队友维护自己的上下文窗口并作为单独的 Claude 实例运行。保持团队任务小且独立，以限制每个队友的令牌使用。有关详细信息，请参阅[agent 团队](#)。

编写具体的提示

模糊的请求（如“改进此代码库”）会触发广泛扫描。具体的请求（如“向 auth.ts 中的登录函数添加输入验证”）让 Claude 能够以最少的文件读取高效地工作。

高效处理复杂任务

对于较长或更复杂的工作，这些习惯有助于避免因走错路而浪费的令牌：

- **对复杂任务使用 plan mode：**按 Shift+Tab 进入 [plan mode](#)，然后再进行实现。Claude 探索代码库并提出一个方法供您批准，防止当初始方向错误时的昂贵返工。
- **尽早纠正方向：**如果 Claude 开始朝错误的方向发展，按 Escape 立即停止。使用 `/rewind` 或双击 Escape 将对话和代码恢复到之前的 checkpoint。
- **给出验证目标：**在您的提示中包含测试用例、粘贴屏幕截图或定义预期输出。当 Claude 可以验证自己的工作时，它会在您需要请求修复之前捕获问题。
- **增量测试：**编写一个文件，测试它，然后继续。这会在问题便宜时尽早捕获问题。

后台令牌使用

Claude Code 即使在空闲时也会为某些后台功能使用令牌：

- **对话总结：**为 `claude --resume` 功能总结以前对话的后台作业
- **命令处理：**某些命令（如 `/cost`）可能会生成请求以检查状态

这些后台进程即使没有活跃交互也会消耗少量令牌（通常每个会话不到 \$0.04）。

了解 Claude Code 行为的变化

Claude Code 定期接收可能改变功能工作方式的更新，包括成本报告。运行 `claude --version` 检查您的当前版本。如有具体计费问题，请通过您的[Console 账户](#)联系 Anthropic 支持。对于团队部署，在更广泛的推出之前，从一个小的试点团体开始以建立使用模式。

Part 5: Extensibility

Hooks 参考

Claude Code hook 事件、配置架构、JSON 输入/输出格式、退出代码、异步 hooks、HTTP hooks、提示 hooks 和 MCP 工具 hooks 的参考。

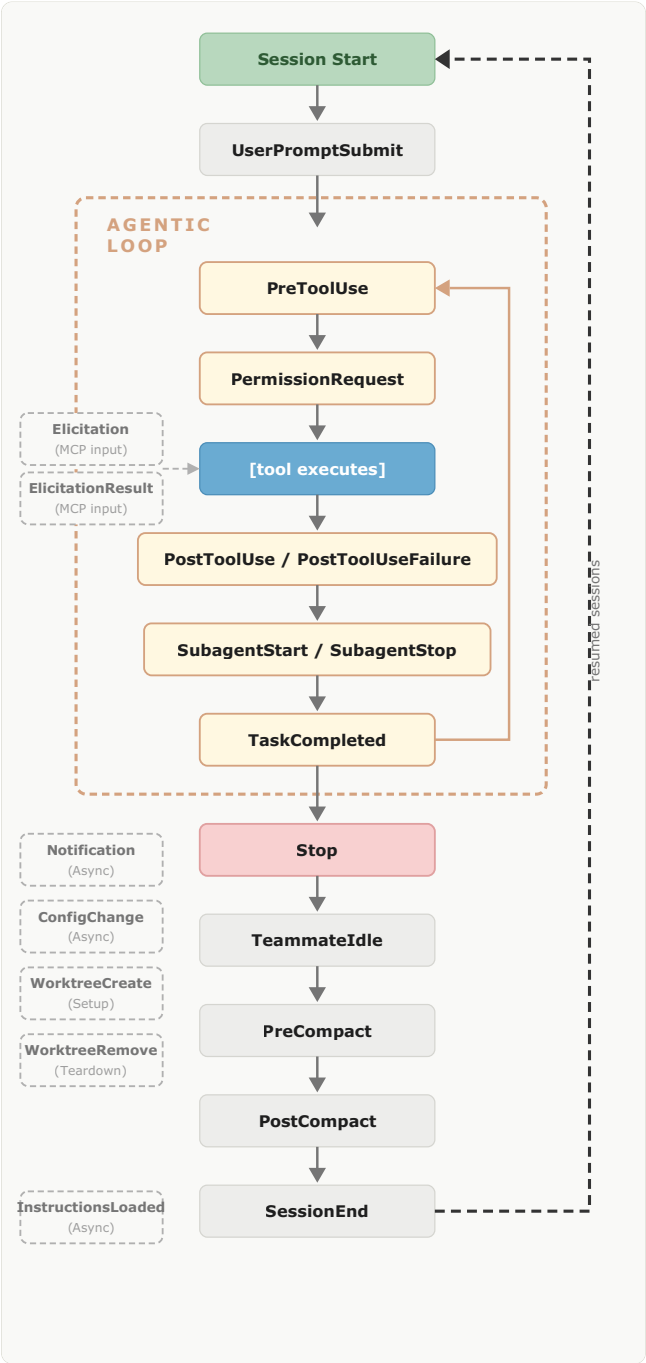
Tip:

有关包含示例的快速入门指南，请参阅 [使用 hooks 自动化工作流](#)。

Hooks 是用户定义的 shell 命令、HTTP 端点或 LLM 提示，在 Claude Code 生命周期中的特定点自动执行。使用此参考查找事件架构、配置选项、JSON 输入/输出格式以及异步 hooks、HTTP hooks 和 MCP 工具 hooks 等高级功能。如果您是第一次设置 hooks，请改为从 [指南](#) 开始。

Hook 生命周期

Hooks 在 Claude Code 会话期间的特定点触发。当事件触发且匹配器匹配时，Claude Code 会将关于该事件的 JSON 上下文传递给您的 hook 处理程序。对于命令 hooks，输入通过 stdin 到达。对于 HTTP hooks，它作为 POST 请求体到达。您的处理程序随后可以检查输入、采取行动并可选地返回决定。某些事件每个会话触发一次，而其他事件在代理循环内重复触发：



Hook 生命周期图，显示从 SessionStart 通过代理循环到 SessionEnd 的 hooks 序列，WorktreeCreate、WorktreeRemove 和 InstructionsLoaded 作为独立异步事件

下表总结了每个事件何时触发。[Hook 事件](#) 部分记录了每个事件的完整输入架构和决定控制选项。

Event	When it fires
SessionStart	When a session begins or resumes
UserPromptSubmit	When you submit a prompt, before Claude processes it
PreToolUse	Before a tool call executes. Can block it
PermissionRequest	When a permission dialog appears
PostToolUse	After a tool call succeeds
PostToolUseFailure	After a tool call fails
Notification	When Claude Code sends a notification
SubagentStart	When a subagent is spawned
SubagentStop	When a subagent finishes
Stop	When Claude finishes responding
TeammateIdle	When an agent team teammate is about to go idle
TaskCompleted	When a task is being marked as completed
InstructionsLoaded	When a CLAUDE.md or <code>.claude/rules/*.md</code> file is loaded into context. Fires at session start and when files are lazily loaded during a session
ConfigChange	When a configuration file changes during a session
WorktreeCreate	When a worktree is being created via <code>--worktree</code> or <code>isolation: "worktree"</code> . Replaces default git behavior
WorktreeRemove	When a worktree is being removed, either at session exit or when a subagent finishes
PreCompact	Before context compaction
PostCompact	After context compaction completes

Event	When it fires
Elicitation	When an MCP server requests user input during a tool call
ElicitationResult	After a user responds to an MCP elicitation, before the response is sent back to the server
SessionEnd	When a session terminates

Hook 如何解析

要了解这些部分如何组合在一起，请考虑这个 `PreToolUse` hook，它阻止破坏性 shell 命令。该 hook 在每次 Bash 工具调用之前运行 `block-rm.sh`：

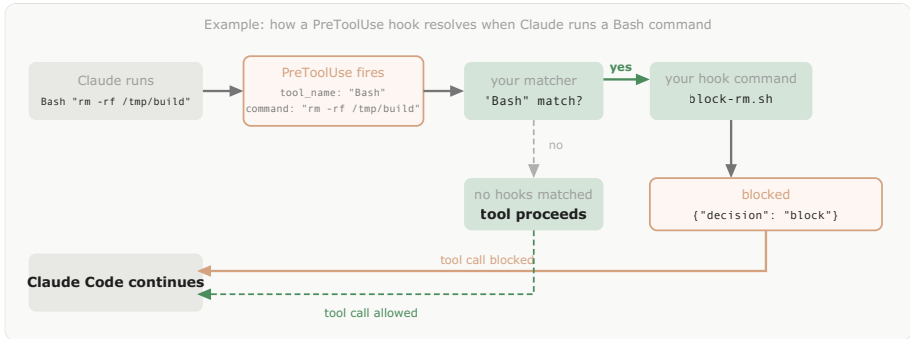
```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": ".claude/hooks/block-rm.sh"
          }
        ]
      }
    ]
  }
}
```

该脚本从 stdin 读取 JSON 输入，提取命令，如果包含 `rm -rf`，则返回 `permissionDecision` 为 `"deny"`：

```
#!/bin/bash
## .claude/hooks/block-rm.sh
COMMAND=$(jq -r '.tool_input.command')

if echo "$COMMAND" | grep -q 'rm -rf'; then
  jq -n '{
    hookSpecificOutput: {
      hookEventName: "PreToolUse",
      permissionDecision: "deny",
      permissionDecisionReason: "Destructive command blocked by hook"
    }
  }'
else
  exit 0 # allow the command
fi
```

现在假设 Claude Code 决定运行 **Bash "rm -rf /tmp/build"**。以下是发生的情况：



Hook 解析流：PreToolUse 事件触发，匹配器检查 Bash 匹配，hook 处理程序运行，结果返回到 Claude Code

Step 1: 事件触发

PreToolUse 事件触发。Claude Code 将工具输入作为 JSON 通过 stdin 发送到 hook：

```
{ "tool_name": "Bash", "tool_input": { "command": "rm -rf /tmp/build" }, ... }
```

Step 2: 匹配器检查

匹配器 `"Bash"` 与工具名称匹配，因此 `block-rm.sh` 运行。如果您省略匹配器或使用 `"*"`，hook 在事件的每次出现时运行。仅当定义了匹配器且不匹配时，hooks 才会跳过。

Step 3: Hook 处理程序运行

脚本从输入中提取 `"rm -rf /tmp/build"` 并找到 `rm -rf`，因此它将决定打印到 stdout：

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Destructive command blocked by hook"
  }
}
```

如果命令是安全的（如 `npm test`），脚本将改为执行 `exit 0`，这告诉 Claude Code 允许工具调用而无需进一步操作。

Step 4: Claude Code 对结果采取行动

Claude Code 读取 JSON 决定，阻止工具调用，并向 Claude 显示原因。

下面的 [配置](#) 部分记录了完整的架构，每个 [hook 事件](#) 部分记录了您的命令接收的输入以及它可以返回的输出。

配置

Hooks 在 JSON 设置文件中定义。配置有三个嵌套级别：

1. 选择要响应的 [hook 事件](#)，如 `PreToolUse` 或 `Stop`
2. 添加 [匹配器组](#) 来过滤何时触发，如“仅针对 Bash 工具”
3. 定义一个或多个 [hook 处理程序](#) 在匹配时运行

有关完整的演练和带注释的示例，请参阅上面的 [Hook 如何解析](#)。

Note:

此页面为每个级别使用特定术语：**hook 事件**表示生命周期点，**匹配器组**表示过滤器，**hook 处理程序**表示运行的 shell 命令、HTTP 端点、提示或代理。“Hook”本身指的是一般功能。

Hook 位置

您定义 hook 的位置决定了其范围：

位置	范围	可共享
<code>~/.claude/ settings.json</code>	您的所有项目	否，仅限于您的计算机
<code>.claude/ settings.json</code>	单个项目	是，可以提交到仓库
<code>.claude/ settings.local.json</code>	单个项目	否，gitignored
托管策略设置	组织范围	是，由管理员控制
<code>Plugin hooks/ hooks.json</code>	启用插件时	是，与插件捆绑
<code>Skill</code> 或 <code>agent</code> frontmatter	组件活跃时	是，在组件文件中定义

有关设置文件解析的详细信息，请参阅 [settings](#)。企业管理员可以使用 `allowManagedHooksOnly` 来阻止用户、项目和插件 hooks。请参阅 [Hook 配置](#)。

匹配器模式

`matcher` 字段是一个正则表达式字符串，用于过滤 hooks 何时触发。使用 `"*"`、`""` 或完全省略 `matcher` 来匹配所有出现。每个事件类型在不同的字段上匹配：

事件	匹配器过滤的内容	示例匹配器值
<code>PreToolUse</code> 、 <code>PostToolUse</code> 、 <code>PostToolUseFailure</code> 、 <code>PermissionRequest</code>	工具名称	<code>Bash</code> 、 <code>Edit Write</code> 、 <code>mc</code> __.*
<code>SessionStart</code>	会话如何启动	<code>startup</code> 、 <code>resume</code> 、 <code>clear</code> 、 <code>compact</code>
<code>SessionEnd</code>	会话为何结束	<code>clear</code> 、 <code>logout</code> 、 <code>prompt_input_exit</code> 、 <code>bypass_permissions_disabled</code> 、 <code>other</code>
<code>Notification</code>	通知类型	<code>permission_prompt</code> 、 <code>idle_prompt</code> 、 <code>auth_success</code> 、 <code>elicitation_dialog</code>

事件	匹配器过滤的内容	示例匹配器值
SubagentStart	代理类型	Bash 、 Explore 、 Plan 或自定义代理名称
PreCompact	触发压缩的原因	manual 、 auto
SubagentStop	代理类型	与 SubagentStart 相同的值
ConfigChange	配置源	user_settings 、 project_settings 、 local_settings 、 policy_settings 、 skills
UserPromptSubmit 、 Stop 、 TeammateIdle 、 TaskCompleted 、 WorktreeCreate 、 WorktreeRemove 、 InstructionsLoaded	不支持匹配器	总是在每次出现时触发

匹配器是一个正则表达式，所以 `Edit|Write` 匹配任一工具，`Notebook.*` 匹配任何以 Notebook 开头的工具。匹配器针对 Claude Code 在 stdin 上发送给您的 hook 的 [JSON 输入](#) 中的字段运行。对于工具事件，该字段是 `tool_name`。每个 [hook 事件](#) 部分列出了该事件的完整匹配器值集和输入架构。

此示例仅在 Claude 写入或编辑文件时运行 linting 脚本：

```

{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/lint-check.sh"
          }
        ]
      }
    ]
  }
}

```

`UserPromptSubmit`、`Stop`、`TeammateIdle`、`TaskCompleted`、`WorktreeCreate`、`WorktreeRemove` 和 `InstructionsLoaded` 不支持匹配器，总是在每次出现时触发。如果您向这些事件添加 `matcher` 字段，它会被静默忽略。

匹配 MCP 工具

[MCP](#) 服务器工具在工具事件中显示为常规工具（`PreToolUse`、`PostToolUse`、`PostToolUseFailure`、`PermissionRequest`），因此您可以像匹配任何其他工具名称一样匹配它们。

MCP 工具遵循命名模式 `mcp__<server>__<tool>`，例如：

- `mcp__memory__create_entities`：Memory 服务器的创建实体工具
- `mcp__filesystem__read_file`：Filesystem 服务器的读取文件工具
- `mcp__github__search_repositories`：GitHub 服务器的搜索工具

使用正则表达式模式来针对特定 MCP 工具或工具组：

- `mcp__memory__.*` 匹配来自 `memory` 服务器的所有工具
- `mcp__.*__write.*` 匹配来自任何服务器的任何包含“write”的工具

此示例记录所有内存服务器操作并验证来自任何 MCP 服务器的写入操作：

```

{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "mcp__memory__.*",
        "hooks": [
          {
            "type": "command",
            "command": "echo 'Memory operation initiated' >> ~/mcp-operations.log"
          }
        ]
      },
      {
        "matcher": "mcp__.*__write.*",
        "hooks": [
          {
            "type": "command",
            "command": "/home/user/scripts/validate-mcp-write.py"
          }
        ]
      }
    ]
  }
}

```

Hook 处理程序字段

内部 `hooks` 数组中的每个对象都是一个 hook 处理程序：当匹配器匹配时运行的 shell 命令、HTTP 端点、LLM 提示或代理。有四种类型：

- **命令 hooks** (`type: "command"`)：运行 shell 命令。您的脚本在 stdin 上接收事件的 [JSON 输入](#)，并通过退出代码和 stdout 传回结果。
- **HTTP hooks** (`type: "http"`)：将事件的 JSON 输入作为 HTTP POST 请求发送到 URL。端点通过使用与命令 hooks 相同的 [JSON 输出格式](#) 的响应体传回结果。
- **提示 hooks** (`type: "prompt"`)：向 Claude 模型发送提示进行单轮评估。模型返回 yes/no 决定作为 JSON。请参阅 [基于提示的 hooks](#)。
- **代理 hooks** (`type: "agent"`)：生成一个可以使用 Read、Grep 和 Glob 等工具来验证条件的 subagent，然后返回决定。请参阅 [基于代理的 hooks](#)。

通用字段

这些字段适用于所有 hook 类型：

字段	必需	描述
type	是	"command"、"http"、"prompt" 或 "agent"
timeout	否	取消前的秒数。默认值：命令 600，提示 30，代理 60
statusMessage	否	hook 运行时显示的自定义加载程序消息
once	否	如果为 true，每个会话仅运行一次，然后被移除。仅限 Skills，不是代理。请参阅 Skills 和代理中的 Hooks

命令 hook 字段

除了 [通用字段](#) 外，命令 hooks 还接受这些字段：

字段	必需	描述
command	是	要执行的 shell 命令
async	否	如果为 true，在后台运行而不阻止。请参阅 在后台运行 hooks

HTTP hook 字段

除了 [通用字段](#) 外，HTTP hooks 还接受这些字段：

字段	必需	描述
url	是	发送 POST 请求的 URL

字段	必需	描述
<code>headers</code>	否	其他 HTTP 标头作为键值对。值支持使用 <code>\$VAR_NAME</code> 或 <code>\${VAR_NAME}</code> 语法的环境变量插值。仅解析 <code>allowedEnvVars</code> 中列出的变量
<code>allowedEnvVars</code>	否	可能被插值到标头值中的环境变量名称列表。对未列出变量的引用被替换为空字符串。任何环境变量插值都需要此项

Claude Code 将 hook 的 [JSON 输入](#) 作为 POST 请求体发送，`Content-Type: application/json`。响应体使用与命令 hooks 相同的 [JSON 输出格式](#)。

错误处理与命令 hooks 不同：非 2xx 响应、连接失败和超时都会产生非阻止错误，允许执行继续。要阻止工具调用或拒绝权限，返回 2xx 响应，其 JSON 体包含 `decision: "block"` 或 `hookSpecificOutput` 与 `permissionDecision: "deny"`。

此示例将 `PreToolUse` 事件发送到本地验证服务，使用来自 `MY_TOKEN` 环境变量的令牌进行身份验证：

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "http",
            "url": "http://localhost:8080/hooks/pre-tool-use",
            "timeout": 30,
            "headers": {
              "Authorization": "Bearer $MY_TOKEN"
            },
            "allowedEnvVars": ["MY_TOKEN"]
          }
        ]
      }
    ]
  }
}
```

Note:

HTTP hooks 必须通过直接编辑设置 JSON 来配置。 `/hooks` 交互菜单仅支持添加命令 hooks。

提示和代理 hook 字段

除了 [通用字段](#) 外，提示和代理 hooks 还接受这些字段：

字段	必需	描述
<code>prompt</code>	是	要发送给模型的提示文本。 使用 <code>\$ARGUMENTS</code> 作为 hook 输入 JSON 的占位符
<code>model</code>	否	用于评估的模型。默认为快速模型

所有匹配的 hooks 并行运行，相同的处理程序会自动去重。命令 hooks 按命令字符串去重，HTTP hooks 按 URL 去重。处理程序在当前目录中运行，使用 Claude Code 的环境。`$CLAUDE_CODE_REMOTE` 环境变量在远程 web 环境中设置为 `"true"`，在本地 CLI 中未设置。

按路径引用脚本

使用环境变量来引用相对于项目或插件根目录的 hook 脚本，无论 hook 运行时的工作目录如何：

- `$CLAUDE_PROJECT_DIR`：项目根目录。用引号包装以处理包含空格的路径。
- `${CLAUDE_PLUGIN_ROOT}`：插件的根目录，用于与 `plugin` 捆绑的脚本。

项目脚本

此示例使用 `$CLAUDE_PROJECT_DIR` 在任何 `Write` 或 `Edit` 工具调用后从项目的 `.claude/hooks/` 目录运行样式检查器：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/check-style.sh"
          }
        ]
      }
    ]
  }
}
```

插件脚本

在 `hooks/hooks.json` 中定义插件 hooks，带有可选的顶级 `description` 字段。启用插件时，其 hooks 与您的用户和项目 hooks 合并。

此示例运行与插件捆绑的格式化脚本：

```

{
  "description": "Automatic code formatting",
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/format.sh",
            "timeout": 30
          }
        ]
      }
    ]
  }
}

```

有关创建插件 hooks 的详细信息，请参阅 [插件组件参考](#)。

Skills 和代理中的 Hooks

除了设置文件和插件外，hooks 还可以使用 frontmatter 直接在 [skills](#) 和 [subagents](#) 中定义。这些 hooks 的范围限于组件的生命周期，仅在该组件活跃时运行。

支持所有 hook 事件。对于 subagents，**Stop** hooks 会自动转换为 **SubagentStop**，因为这是 subagent 完成时触发的事件。

Hooks 使用与基于设置的 hooks 相同的配置格式，但范围限于组件的生命周期，并在其完成时清理。

此 skill 定义了一个 **PreToolUse** hook，在每个 **Bash** 命令之前运行安全验证脚本：

```

---
name: secure-operations
description: Perform operations with security checks
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/security-check.sh"
---

```

代理在其 YAML frontmatter 中使用相同的格式。

`/hooks` 菜单

在 Claude Code 中输入 `/hooks` 以打开交互式 hooks 管理器，您可以在其中查看、添加和删除 hooks，而无需直接编辑设置文件。有关分步演练，请参阅指南中的 [设置您的第一个 hook](#)。

菜单中的每个 hook 都标有括号前缀，指示其来源：

- `[User]`：来自 `~/.claude/settings.json`
- `[Project]`：来自 `.claude/settings.json`
- `[Local]`：来自 `.claude/settings.local.json`
- `[Plugin]`：来自插件的 `hooks/hooks.json`，只读

禁用或移除 hooks

要移除 hook，请从设置 JSON 文件中删除其条目，或使用 `/hooks` 菜单并选择 hook 来删除它。

要临时禁用所有 hooks 而不移除它们，请在设置文件中设置 `"disableAllHooks": true` 或使用 `/hooks` 菜单中的切换。没有办法在保持 hook 在配置中的同时禁用单个 hook。

`disableAllHooks` 设置遵循托管设置层次结构。如果管理员通过托管策略设置配置了 hooks，则在用户、项目或本地设置中设置的 `disableAllHooks` 无法禁用这些托管 hooks。仅在托管设置级别设置的 `disableAllHooks` 可以禁用托管 hooks。

对设置文件中 hooks 的直接编辑不会立即生效。Claude Code 在启动时捕获 hooks 的快照，并在整个会话中使用它。这可以防止恶意或意外的 hook 修改在会话中途生效，而无需您的审查。如果 hooks 在外部被修改，Claude Code 会警告您，并要求在 `/hooks` 菜单中进行审查，然后更改才能应用。

Hook 输入和输出

命令 `hooks` 通过 `stdin` 接收 JSON 数据，并通过退出代码、`stdout` 和 `stderr` 传回结果。HTTP `hooks` 接收相同的 JSON 作为 POST 请求体，并通过 HTTP 响应体传回结果。本部分涵盖所有事件通用的字段和行为。每个事件在 [Hook 事件](#) 下的部分包括其特定的输入架构和决定控制选项。

通用输入字段

所有 `hook` 事件都接收这些字段作为 JSON，除了每个 [hook 事件](#) 部分中记录的事件特定字段。对于命令 `hooks`，此 JSON 通过 `stdin` 到达。对于 HTTP `hooks`，它作为 POST 请求体到达。

字段	描述
<code>session_id</code>	当前会话标识符
<code>transcript_path</code>	对话 JSON 的路径
<code>cwd</code>	调用 <code>hook</code> 时的当前工作目录
<code>permission_mode</code>	当前 权限模式 ： <code>"default"</code> 、 <code>"plan"</code> 、 <code>"acceptEdits"</code> 、 <code>"dontAsk"</code> 或 <code>"bypassPermissions"</code>
<code>hook_event_name</code>	触发的事件名称

使用 `--agent` 运行或在 `subagent` 内部时，包括两个额外字段：

字段	描述
<code>agent_id</code>	<code>subagent</code> 的唯一标识符。仅当 <code>hook</code> 在 <code>subagent</code> 调用内触发时存在。使用此来区分 <code>subagent hook</code> 调用和主线程调用。
<code>agent_type</code>	代理名称（例如， <code>"Explore"</code> 或 <code>"security-reviewer"</code> ）。当会话使用 <code>--agent</code> 或 <code>hook</code> 在 <code>subagent</code> 内触发时存在。对于 <code>subagents</code> ， <code>subagent</code> 的类型优先于会话的 <code>--agent</code> 值。

例如，Bash 命令的 `PreToolUse` `hook` 在 `stdin` 上接收：

```
{
  "session_id": "abc123",
  "transcript_path": "/home/user/.claude/projects/.../transcript.jsonl",
  "cwd": "/home/user/my-project",
  "permission_mode": "default",
  "hook_event_name": "PreToolUse",
  "tool_name": "Bash",
  "tool_input": {
    "command": "npm test"
  }
}
```

`tool_name` 和 `tool_input` 字段是事件特定的。每个 [hook 事件](#) 部分记录了该事件的额外字段。

退出代码输出

来自您的 hook 命令的退出代码告诉 Claude Code 该操作是应该继续、被阻止还是被忽略。

退出 0 表示成功。Claude Code 解析 stdout 以获取 [JSON 输出字段](#)。JSON 输出仅在退出 0 时处理。对于大多数事件，stdout 仅在详细模式（`Ctrl+0`）中显示。例外是 `UserPromptSubmit` 和 `SessionStart`，其中 stdout 作为 Claude 可以看到和作用的上下文添加。

退出 2 表示阻止错误。Claude Code 忽略 stdout 和其中的任何 JSON。相反，stderr 文本被反馈给 Claude 作为错误消息。效果取决于事件：`PreToolUse` 阻止工具调用，`UserPromptSubmit` 拒绝提示，等等。有关完整列表，请参阅 [退出代码 2 行为](#)。

任何其他退出代码 是非阻止错误。stderr 在详细模式（`Ctrl+0`）中显示，执行继续。

例如，一个 hook 命令脚本，阻止危险的 Bash 命令：


```
#!/bin/bash
## 从 stdin 读取 JSON 输入，检查命令
command=$(jq -r '.tool_input.command' < /dev/stdin)

if [[ "$command" = rm* ]]; then
    echo "Blocked: rm commands are not allowed" >&2
    exit 2 # 阻止错误：工具调用被阻止
fi

exit 0 # 成功：工具调用继续
```

每个事件的退出代码 2 行为

退出代码 2 是 hook 发出”停止，不要这样做”的方式。效果取决于事件，因为某些事件代表可以被阻止的操作（如尚未发生的工具调用），而其他事件代表已经发生或无法防止的事情。

Hook 事件	可以阻止?	退出 2 时发生的情况
PreToolUse	是	阻止工具调用
PermissionRequest	是	拒绝权限
UserPromptSubmit	是	阻止提示处理并从上下文中删除提示
Stop	是	防止 Claude 停止，继续对话
SubagentStop	是	防止 subagent 停止
TeammateIdle	是	防止队友空闲（队友继续工作）
TaskCompleted	是	防止任务被标记为已完成
ConfigChange	是	阻止配置更改生效（除 policy_settings 外）
PostToolUse	否	向 Claude 显示 stderr（工具已运行）

Hook 事件	可以阻止?	退出 2 时发生的情况
PostToolUseFailure	否	向 Claude 显示 stderr (工具已失败)
Notification	否	仅向用户显示 stderr
SubagentStart	否	仅向用户显示 stderr
SessionStart	否	仅向用户显示 stderr
SessionEnd	否	仅向用户显示 stderr
PreCompact	否	仅向用户显示 stderr
WorktreeCreate	是	任何非零退出代码都会导致 worktree 创建失败
WorktreeRemove	否	失败仅在调试模式下记录
InstructionsLoaded	否	退出代码被忽略

HTTP 响应处理

HTTP hooks 使用 HTTP 状态代码和响应体而不是退出代码和 stdout：

- **2xx 带空体**：成功，等同于退出代码 0 且无输出
- **2xx 带纯文本体**：成功，文本作为上下文添加
- **2xx 带 JSON 体**：成功，使用与命令 hooks 相同的 [JSON 输出](#) 架构解析
- **非 2xx 状态**：非阻止错误，执行继续
- **连接失败或超时**：非阻止错误，执行继续

与命令 hooks 不同，HTTP hooks 无法仅通过状态代码发出阻止错误信号。要阻止工具调用或拒绝权限，返回 2xx 响应，其 JSON 体包含适当的决定字段。

JSON 输出

退出代码让您允许或阻止，但 JSON 输出提供更细粒度的控制。与其使用代码 2 退出来阻止，不如退出 0 并将 JSON 对象打印到 stdout。Claude Code 从该 JSON 读取特定字段以控制行为，包括 [决定控制](#) 来阻止、允许或升级给用户。

Note:

您必须为每个 hook 选择一种方法，而不是两种：要么单独使用退出代码进行信号，要么退出 0 并打印 JSON 以进行结构化控制。Claude Code 仅在退出 0 时处理 JSON。如果您退出 2，任何 JSON 都会被忽略。

您的 hook 的 stdout 必须仅包含 JSON 对象。如果您的 shell 配置文件在启动时打印文本，它可能会干扰 JSON 解析。请参阅故障排除指南中的 [JSON 验证失败](#)。

JSON 对象支持三种字段：

- **通用字段**，如 `continue`，在所有事件中工作。这些列在下表中。
- **顶级 `decision` 和 `reason`** 由某些事件用来阻止或提供反馈。
- **`hookSpecificOutput`** 是一个嵌套对象，用于需要更丰富控制的事件。它需要一个 `hookEventName` 字段设置为事件名称。

字段	默认	描述
<code>continue</code>	<code>true</code>	如果为 <code>false</code> ，Claude 在 hook 运行后完全停止处理。优先于任何事件特定的决定字段
<code>stopReason</code>	无	hook 运行后向用户显示的消息，当 <code>continue</code> 为 <code>false</code> 时。不向 Claude 显示
<code>suppressOutput</code>	<code>false</code>	如果为 <code>true</code> ，从详细模式输出中隐藏 stdout
<code>systemMessage</code>	无	向用户显示的警告消息

要完全停止 Claude，无论事件类型如何：

```
{ "continue": false, "stopReason": "Build failed, fix errors before continuing" }
```

决定控制

并非每个事件都支持通过 JSON 阻止或控制行为。支持的事件各自使用不同的字段集来表达该决定。在编写 hook 之前，使用此表作为快速参考：

事件	决定模式	关键字段
UserPromptSubmit 、 PostToolUse 、 PostToolUseFailure 、 Stop 、SubagentStop 、 ConfigChange	顶级 <code>decision</code>	<code>decision: "block"</code> 、 <code>reason</code>
TeammateIdle 、 TaskCompleted	退出代码或 <code>continue: false</code>	退出代码 2 用 <code>stderr</code> 反馈 阻止操作。JSON <code>{"continue": false, "stopReason": "..."} 也停止队友，匹配 <code>Stop</code> hook 行为</code>
PreToolUse	<code>hookSpecificOutput</code>	<code>permissionDecision</code> (<code>allow/deny/ask</code>) 、 <code>permissionDecisionReason</code>
PermissionRequest	<code>hookSpecificOutput</code>	<code>decision.behavior</code> (<code>allow/deny</code>)
WorktreeCreate	stdout 路径	Hook 打印创建的 <code>worktree</code> 的绝对路径。非 零退出失败创建
WorktreeRemove 、 Notification 、 SessionEnd 、 PreCompact 、 InstructionsLoaded	无	无决定控制。用于日志记录 或清理等副作用

以下是每种模式的实际示例：

顶级决定

由 `UserPromptSubmit` 、 `PostToolUse` 、 `PostToolUseFailure` 、 `Stop` 、 `SubagentStop` 和 `ConfigChange` 使用。唯一的值是 `"block"` 。要允许操作继续，从您的 JSON 中省略 `decision` ，或退出 0 而不带任何 JSON ：

```
{
  "decision": "block",
  "reason": "Test suite must pass before proceeding"
}
```

PreToolUse

使用 `hookSpecificOutput` 以获得更丰富的控制：允许、拒绝或升级给用户。您还可以在运行前修改工具输入或为 Claude 注入额外上下文。有关完整的选项集，请参阅 [PreToolUse 决定控制](#)。

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Database writes are not allowed"
  }
}
```

PermissionRequest

使用 `hookSpecificOutput` 代表用户允许或拒绝权限请求。允许时，您还可以修改工具的输入或应用权限规则，以便用户不会再次被提示。有关完整的选项集，请参阅 [PermissionRequest 决定控制](#)。

```
{
  "hookSpecificOutput": {
    "hookEventName": "PermissionRequest",
    "decision": {
      "behavior": "allow",
      "updatedInput": {
        "command": "npm run lint"
      }
    }
  }
}
```

有关扩展示例，包括 Bash 命令验证、提示过滤和自动批准脚本，请参阅指南中的 [您可以自动化的内容](#) 和 [Bash 命令验证器参考实现](#)。

Hook 事件

每个事件对应于 Claude Code 生命周期中 hooks 可以运行的一个点。下面的部分按照生命周期的顺序排列：从会话设置通过代理循环到会话结束。每个部分描述事件何时触发、它支持什么匹配器、它接收的 JSON 输入以及如何通过输出控制行为。

SessionStart

在 Claude Code 启动新会话或恢复现有会话时运行。对于加载开发上下文（如现有问题或代码库的最近更改）或设置环境变量很有用。对于不需要脚本的静态上下文，请改用 [CLAUDE.md](#)。

SessionStart 在每个会话上运行，所以保持这些 hooks 快速。仅支持 `type: "command"` hooks。

匹配器值对应于会话的启动方式：

匹配器	何时触发
<code>startup</code>	新会话
<code>resume</code>	<code>--resume</code> 、 <code>--continue</code> 或 <code>/resume</code>
<code>clear</code>	<code>/clear</code>
<code>compact</code>	自动或手动压缩

SessionStart 输入

除了 [通用输入字段](#) 外，SessionStart hooks 还接收 `source`、`model` 和可选的 `agent_type`。 `source` 字段指示会话如何启动：新会话为 `"startup"`，恢复会话为 `"resume"`，`/clear` 后为 `"clear"`，压缩后为 `"compact"`。 `model` 字段包含模型标识符。如果您使用 `claude --agent <name>` 启动 Claude Code，`agent_type` 字段包含代理名称。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SessionStart",
  "source": "startup",
  "model": "claude-sonnet-4-6"
}
```

SessionStart 决定控制

您的 hook 脚本打印到 stdout 的任何文本都作为 Claude 的上下文添加。除了所有 hooks 可用的 [JSON 输出字段](#) 外，您还可以返回这些事件特定字段：

字段	描述
<code>additionalContext</code>	添加到 Claude 上下文的字符串。多个 hooks 的值被连接

```
{
  "hookSpecificOutput": {
    "hookEventName": "SessionStart",
    "additionalContext": "My additional context here"
  }
}
```

持久化环境变量

SessionStart hooks 可以访问 `CLAUDE_ENV_FILE` 环境变量，它提供一个文件路径，您可以在其中为后续 Bash 命令持久化环境变量。

要设置单个环境变量，请将 `export` 语句写入 `CLAUDE_ENV_FILE`。使用追加 (`>>`) 来保留由其他 hooks 设置的变量：

```
#!/bin/bash

if [ -n "$CLAUDE_ENV_FILE" ]; then
    echo 'export NODE_ENV=production' >> "$CLAUDE_ENV_FILE"
    echo 'export DEBUG_LOG=true' >> "$CLAUDE_ENV_FILE"
    echo 'export PATH="$PATH:./node_modules/.bin"' >> "$CLAUDE_ENV_FILE"
fi

exit 0
```

要捕获设置命令中的所有环境更改，请比较之前和之后导出的变量：

```
#!/bin/bash

ENV_BEFORE=$(export -p | sort)

## 运行修改环境的设置命令
source ~/.nvm/nvm.sh
nvm use 20

if [ -n "$CLAUDE_ENV_FILE" ]; then
    ENV_AFTER=$(export -p | sort)
    comm -13 <(echo "$ENV_BEFORE") <(echo "$ENV_AFTER") >> "$CLAUDE_ENV_FILE"
fi

exit 0
```

写入此文件的任何变量都将在会话期间 Claude Code 执行的所有后续 Bash 命令中可用。

Note:

`CLAUDE_ENV_FILE` 可用于 SessionStart hooks。其他 hook 类型无法访问此变量。

InstructionsLoaded

当 `CLAUDE.md` 或 `.claude/rules/*.md` 文件加载到上下文中时触发。此事件在会话启动时为急切加载的文件触发，稍后当文件被懒加载时再次触发，例如当 Claude 访问包含嵌套 `CLAUDE.md` 的子目录或条件规则与 `paths: frontmatter` 匹配时。hook 不支持阻止或决定控制。它异步运行以用于可观测性目的。

InstructionsLoaded 不支持匹配器，在每次加载时触发。

InstructionsLoaded 输入

除了 [通用输入字段](#) 外，InstructionsLoaded hooks 还接收这些字段：

字段	描述
<code>file_path</code>	加载的指令文件的绝对路径
<code>memory_type</code>	文件的范围: <code>"User"</code> 、 <code>"Project"</code> 、 <code>"Local"</code> 或 <code>"Managed"</code>
<code>load_reason</code>	文件被加载的原因: <code>"session_start"</code> 、 <code>"nested_traversal"</code> 、 <code>"path_glob_match"</code> 或 <code>"include"</code>
<code>globs</code>	文件的 <code>paths: frontmatter</code> 中的路径 glob 模式（如果有）。仅对 <code>path_glob_match</code> 加载存在
<code>trigger_file_path</code>	触发此加载的文件的相对路径，用于懒加载
<code>parent_file_path</code>	包含此文件的父指令文件的路径，用于 <code>include</code> 加载

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../transcript.jsonl",
  "cwd": "/Users/my-project",
  "permission_mode": "default",
  "hook_event_name": "InstructionsLoaded",
  "file_path": "/Users/my-project/CLAUDE.md",
  "memory_type": "Project",
  "load_reason": "session_start"
}
```

InstructionsLoaded 决定控制

InstructionsLoaded hooks 没有决定控制。它们无法阻止或修改指令加载。使用此事件进行审计日志记录、合规性跟踪或可观测性。

UserPromptSubmit

在用户提交提示时运行，在 Claude 处理之前。这允许您根据提示/对话添加额外上下文、验证提示或阻止某些类型的提示。

UserPromptSubmit 输入

除了 [通用输入字段](#) 外，UserPromptSubmit hooks 还接收包含用户提交的文本的 `prompt` 字段。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "UserPromptSubmit",
  "prompt": "Write a function to calculate the factorial of a number"
}
```

UserPromptSubmit 决定控制

`UserPromptSubmit` hooks 可以控制用户提示是否被处理并添加上下文。所有 [JSON 输出字段](#) 都可用。

在退出代码 0 时有两种方式向对话添加上下文：

- **纯文本 `stdout`**：写入 `stdout` 的任何非 JSON 文本都作为上下文添加
- **带 `additionalContext` 的 JSON**：使用下面的 JSON 格式以获得更多控制。
`additionalContext` 字段作为上下文添加

纯 `stdout` 在成绩单中显示为 hook 输出。`additionalContext` 字段更谨慎地添加。

要阻止提示，返回一个 JSON 对象，其中 `decision` 设置为 `"block"`：

字段	描述
<code>decision</code>	<code>"block"</code> 防止提示被处理并从上下文中删除。省略以允许提示继续

字段	描述
<code>reason</code>	当 <code>decision</code> 为 <code>"block"</code> 时向用户显示。不添加到上下文
<code>additionalContext</code>	添加到 Claude 上下文的字符串

```
{
  "decision": "block",
  "reason": "Explanation for decision",
  "hookSpecificOutput": {
    "hookEventName": "UserPromptSubmit",
    "additionalContext": "My additional context here"
  }
}
```

Note:

JSON 格式对于简单用例不是必需的。要添加上下文，您可以使用退出代码 0 将纯文本打印到 stdout。当您需要阻止提示或想要更结构化的控制时，使用 JSON。

PreToolUse

在 Claude 创建工具参数后和处理工具调用之前运行。在工具名称上匹配：`Bash`、`Edit`、`Write`、`Read`、`Glob`、`Grep`、`Agent`、`WebFetch`、`WebSearch` 和任何 [MCP 工具名称](#)。

使用 [PreToolUse 决定控制](#) 来允许、拒绝或要求使用工具的权限。

PreToolUse 输入

除了 [通用输入字段](#) 外，PreToolUse hooks 还接收 `tool_name`、`tool_input` 和 `tool_use_id`。 `tool_input` 字段取决于工具：

Bash

执行 shell 命令。

字段	类型	示例	描述
<code>command</code>	string	<code>"npm test"</code>	要执行的 shell 命令
<code>description</code>	string	<code>"Run test suite"</code>	命令执行操作的可选描述

字段	类型	示例	描述
<code>timeout</code>	number	<code>120000</code>	可选超时（毫秒）
<code>run_in_background</code>	boolean	<code>false</code>	是否在后台运行命令

Write

创建或覆盖文件。

字段	类型	示例	描述
<code>file_path</code>	string	<code>"/path/to/file.txt"</code>	要写入的文件的绝对路径
<code>content</code>	string	<code>"file content"</code>	要写入文件的内容

Edit

替换现有文件中的字符串。

字段	类型	示例	描述
<code>file_path</code>	string	<code>"/path/to/file.txt"</code>	要编辑的文件的绝对路径
<code>old_string</code>	string	<code>"original text"</code>	要查找和替换的文本
<code>new_string</code>	string	<code>"replacement text"</code>	替换文本
<code>replace_all</code>	boolean	<code>false</code>	是否替换所有出现

Read

读取文件内容。

字段	类型	示例	描述
<code>file_path</code>	string	<code>"/path/to/file.txt"</code>	要读取的文件的绝对路径

字段	类型	示例	描述
<code>offset</code>	number	<code>10</code>	可选的开始读取的行号
<code>limit</code>	number	<code>50</code>	可选的要读取的行数

Glob

查找与 glob 模式匹配的文件。

字段	类型	示例	描述
<code>pattern</code>	string	<code>"**/*.ts"</code>	要匹配文件的 glob 模式
<code>path</code>	string	<code>"/path/to/dir"</code>	可选的搜索目录。默认为当前工作目录

Grep

使用正则表达式搜索文件内容。

字段	类型	示例	描述
<code>pattern</code>	string	<code>"TODO.*fix"</code>	要搜索的正则表达式模式
<code>path</code>	string	<code>"/path/to/dir"</code>	可选的要搜索的文件或目录
<code>glob</code>	string	<code>"*.ts"</code>	可选的 glob 模式来过滤文件
<code>output_mode</code>	string	<code>"content"</code>	<code>"content"</code> 、 <code>"files_with_matches"</code> 或 <code>"count"</code> 。默认为 <code>"files_with_matches"</code>
<code>-i</code>	boolean	<code>true</code>	不区分大小写的搜索
<code>multiline</code>	boolean	<code>false</code>	启用多行匹配

WebFetch

获取和处理 web 内容。

字段	类型	示例	描述
<code>url</code>	string	<code>"https://example.com/api"</code>	要获取内容的 URL
<code>prompt</code>	string	<code>"Extract the API endpoints"</code>	在获取的内容上运行的提示

WebSearch

搜索网络。

字段	类型	示例	描述
<code>query</code>	string	<code>"react hooks best practices"</code>	搜索查询
<code>allowed_domains</code>	array	<code>["docs.example.com"]</code>	可选：仅包含来自这些域的结果
<code>blocked_domains</code>	array	<code>["spam.example.com"]</code>	可选：排除来自这些域的结果

Agent

生成一个 [subagent](#)。

字段	类型	示例	描述
<code>prompt</code>	string	<code>"Find all API endpoints"</code>	代理要执行的任务
<code>description</code>	string	<code>"Find API endpoints"</code>	任务的简短描述
<code>subagent_type</code>	string	<code>"Explore"</code>	要使用的专门代理的类型
<code>model</code>	string	<code>"sonnet"</code>	可选的模型别名来覆盖默认值

PreToolUse 决定控制

`PreToolUse` hooks 可以控制工具调用是否继续。与使用顶级 `decision` 字段的其他 hooks 不同，`PreToolUse` 在 `hookSpecificOutput` 对象内返回其决定。这给了它更丰富的控制：三个结果（允许、拒绝或询问）加上在执行前修改工具输入的能力。

字段	描述
<code>permissionDecision</code>	"allow" 绕过权限系统，"deny" 防止工具调用，"ask" 提示用户确认
<code>permissionDecisionReason</code>	对于 "allow" 和 "ask"，向用户显示但不向 Claude 显示。 对于 "deny"，向 Claude 显示
<code>updatedInput</code>	在执行前修改工具的输入参数。与 "allow" 结合以自动批准， 或与 "ask" 结合以向用户显示修改后的输入
<code>additionalContext</code>	在工具执行前添加到 Claude 上下文的字符串

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "allow",
    "permissionDecisionReason": "My reason here",
    "updatedInput": {
      "field_to_modify": "new value"
    },
    "additionalContext": "Current environment: production. Proceed with caution."
  }
}
```

Note:

`PreToolUse` 之前使用顶级 `decision` 和 `reason` 字段，但这些对此事件已弃用。改用 `hookSpecificOutput.permissionDecision` 和 `hookSpecificOutput.permissionDecisionReason`。弃用的值 `"approve"` 和 `"block"` 分别映射到 `"allow"` 和 `"deny"`。 `PostToolUse` 和 `Stop` 等其他事件继续使用顶级 `decision` 和 `reason` 作为其当前格式。

PermissionRequest

在向用户显示权限对话框时运行。使用 [PermissionRequest 决定控制](#) 代表用户允许或拒绝。

在工具名称上匹配，与 PreToolUse 相同的值。

PermissionRequest 输入

PermissionRequest hooks 接收 `tool_name` 和 `tool_input` 字段，如 PreToolUse hooks，但没有 `tool_use_id`。可选的 `permission_suggestions` 数组包含用户通常在权限对话框中看到的“总是允许”选项。区别在于 hook 何时触发：PermissionRequest hooks 在权限对话框即将显示给用户时运行，而 PreToolUse hooks 在工具执行前运行，无论权限状态如何。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PermissionRequest",
  "tool_name": "Bash",
  "tool_input": {
    "command": "rm -rf node_modules",
    "description": "Remove node_modules directory"
  },
  "permission_suggestions": [
    { "type": "toolAlwaysAllow", "tool": "Bash" }
  ]
}
```

PermissionRequest 决定控制

`PermissionRequest` hooks 可以允许或拒绝权限请求。除了所有 hooks 可用的 [JSON 输出 字段](#) 外，您的 hook 脚本可以返回一个 `decision` 对象，其中包含这些事件特定字段：

字段	描述
<code>behavior</code>	<code>"allow"</code> 授予权限， <code>"deny"</code> 拒绝它
<code>updatedInput</code>	仅对 <code>"allow"</code> ：在执行前修改工具的输入参数

字段	描述
<code>updatedPermissions</code>	仅对 <code>"allow"</code> ：应用权限规则更新，等同于用户选择“总是允许”选项
<code>message</code>	仅对 <code>"deny"</code> ：告诉 Claude 为什么权限被拒绝
<code>interrupt</code>	仅对 <code>"deny"</code> ：如果为 <code>true</code> ，停止 Claude

```
{
  "hookSpecificOutput": {
    "hookEventName": "PermissionRequest",
    "decision": {
      "behavior": "allow",
      "updatedInput": {
        "command": "npm run lint"
      }
    }
  }
}
```

PostToolUse

在工具成功完成后立即运行。

在工具名称上匹配，与 `PreToolUse` 相同的值。

PostToolUse 输入

`PostToolUse` hooks 在工具已经成功执行后触发。输入包括 `tool_input`（发送给工具的参数）和 `tool_response`（它返回的结果）。两者的确切架构取决于工具。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PostToolUse",
  "tool_name": "Write",
  "tool_input": {
    "file_path": "/path/to/file.txt",
    "content": "file content"
  },
  "tool_response": {
    "filePath": "/path/to/file.txt",
    "success": true
  },
  "tool_use_id": "toolu_01ABC123..."
}
```

PostToolUse 决定控制

PostToolUse hooks 可以在工具执行后向 Claude 提供反馈。除了所有 hooks 可用的 [JSON 输出字段](#) 外，您的 hook 脚本可以返回这些事件特定字段：

字段	描述
decision	"block" 用 reason 提示 Claude。省略以允许操作继续
reason	当 decision 为 "block" 时向 Claude 显示的解釋
additionalContext	Claude 要考虑的额外上下文
updatedMCPToolOutput	仅对 MCP 工具：用提供的值替换工具的输出

```
{
  "decision": "block",
  "reason": "Explanation for decision",
  "hookSpecificOutput": {
    "hookEventName": "PostToolUse",
    "additionalContext": "Additional information for Claude"
  }
}
```

PostToolUseFailure

当工具执行失败时运行。此事件对于抛出错误或返回失败结果的工具调用触发。使用此来记录失败、发送警报或向 Claude 提供纠正反馈。

在工具名称上匹配，与 PreToolUse 相同的值。

PostToolUseFailure 输入

PostToolUseFailure hooks 接收与 PostToolUse 相同的 `tool_name` 和 `tool_input` 字段，以及作为顶级字段的错误信息：

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PostToolUseFailure",
  "tool_name": "Bash",
  "tool_input": {
    "command": "npm test",
    "description": "Run test suite"
  },
  "tool_use_id": "toolu_01ABC123...",
  "error": "Command exited with non-zero status code 1",
  "is_interrupt": false
}
```

字段	描述
<code>error</code>	描述出错的字符串
<code>is_interrupt</code>	可选的布尔值，指示失败是否由用户中断引起

PostToolUseFailure 决定控制

`PostToolUseFailure` hooks 可以在工具失败后向 Claude 提供上下文。除了所有 hooks 可用的 [JSON 输出字段](#) 外，您的 hook 脚本可以返回这些事件特定字段：

字段	描述
<code>additionalContext</code>	Claude 要与错误一起考虑的额外上下文

```
{
  "hookSpecificOutput": {
    "hookEventName": "PostToolUseFailure",
    "additionalContext": "Additional information about the failure for Claude"
  }
}
```

Notification

在 Claude Code 发送通知时运行。在通知类型上匹配：`permission_prompt`、`idle_prompt`、`auth_success`、`elicitation_dialog`。省略匹配器以为所有通知类型运行 hooks。

使用单独的匹配器根据通知类型运行不同的处理程序。此配置在 Claude 需要权限批准时触发权限特定的警报脚本，在 Claude 空闲时触发不同的通知：

```

{
  "hooks": {
    "Notification": [
      {
        "matcher": "permission_prompt",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/permission-alert.sh"
          }
        ]
      },
      {
        "matcher": "idle_prompt",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/idle-notification.sh"
          }
        ]
      }
    ]
  }
}

```

Notification 输入

除了 [通用输入字段](#) 外，Notification hooks 还接收包含通知文本的 `message`、可选的 `title` 和 `notification_type` 指示哪个类型触发。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "Notification",
  "message": "Claude needs your permission to use Bash",
  "title": "Permission needed",
  "notification_type": "permission_prompt"
}
```

Notification hooks 无法阻止或修改通知。除了所有 hooks 可用的 [JSON 输出字段](#) 外，您可以返回 `additionalContext` 来向对话添加上下文：

字段	描述
<code>additionalContext</code>	添加到 Claude 上下文的字符串

SubagentStart

当通过 Agent 工具生成 Claude Code subagent 时运行。支持匹配器来按代理类型名称过滤（内置代理如 `Bash`、`Explore`、`Plan` 或来自 `.claude/agents/` 的自定义代理名称）。

SubagentStart 输入

除了 [通用输入字段](#) 外，SubagentStart hooks 还接收 `agent_id` 与 subagent 的唯一标识符和 `agent_type` 与代理名称（内置代理如 `"Bash"`、`"Explore"`、`"Plan"` 或自定义代理名称）。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SubagentStart",
  "agent_id": "agent-abc123",
  "agent_type": "Explore"
}
```

SubagentStart hooks 无法阻止 subagent 创建，但它们可以向 subagent 注入上下文。除了所有 hooks 可用的 [JSON 输出字段](#) 外，您可以返回：

字段	描述
<code>additionalContext</code>	添加到 subagent 上下文的字符串

```
{
  "hookSpecificOutput": {
    "hookEventName": "SubagentStart",
    "additionalContext": "Follow security guidelines for this task"
  }
}
```

SubagentStop

在 Claude Code subagent 完成响应时运行。在代理类型上匹配，与 SubagentStart 相同的值。

SubagentStop 输入

除了 [通用输入字段](#) 外，SubagentStop hooks 还接收 `stop_hook_active`、`agent_id`、`agent_type`、`agent_transcript_path` 和 `last_assistant_message`。 `agent_type` 字段是用于匹配器过滤的值。 `transcript_path` 是主会话的成绩单，而 `agent_transcript_path` 是 subagent 自己的成绩单，存储在嵌套的 `subagents/` 文件夹中。 `last_assistant_message` 字段包含 subagent 最终响应的文本内容，因此 hooks 可以访问它而无需解析成绩单文件。

```
{
  "session_id": "abc123",
  "transcript_path": "~/claude/projects/.../abc123.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SubagentStop",
  "stop_hook_active": false,
  "agent_id": "def456",
  "agent_type": "Explore",
  "agent_transcript_path": "~/claude/projects/.../abc123/subagents/agent-
def456.jsonl",
  "last_assistant_message": "Analysis complete. Found 3 potential issues..."
}
```

SubagentStop hooks 使用与 [Stop hooks](#) 相同的决定控制格式。

Stop

在主 Claude Code 代理完成响应时运行。如果停止是由于用户中断，则不运行。

Stop 输入

除了 [通用输入字段](#) 外，Stop hooks 还接收 `stop_hook_active` 和

`last_assistant_message`。 `stop_hook_active` 字段在 Claude Code 已经作为 stop hook 的结果继续时为 `true`。检查此值或处理成绩单以防止 Claude Code 无限运行。

`last_assistant_message` 字段包含 Claude 最终响应的文本内容，因此 hooks 可以访问它而无需解析成绩单文件。

```
{
  "session_id": "abc123",
  "transcript_path": "~/claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "Stop",
  "stop_hook_active": true,
  "last_assistant_message": "I've completed the refactoring. Here's a summary..."
}
```


Stop 决定控制

`Stop` 和 `SubagentStop` hooks 可以控制 Claude 是否继续。除了所有 hooks 可用的 [JSON 输出字段](#) 外，您的 hook 脚本可以返回这些事件特定字段：

字段	描述
<code>decision</code>	<code>"block"</code> 防止 Claude 停止。省略以允许 Claude 停止
<code>reason</code>	当 <code>decision</code> 为 <code>"block"</code> 时必需。告诉 Claude 为什么它应该继续

```
{
  "decision": "block",
  "reason": "Must be provided when Claude is blocked from stopping"
}
```

TeammateIdle

在 [agent team](#) 队友在完成其轮次后即将空闲时运行。使用此来在队友停止工作之前强制执行质量门，例如要求通过 lint 检查或验证输出文件存在。

当 `TeammateIdle` hook 以代码 2 退出时，队友接收 `stderr` 消息作为反馈并继续工作而不是空闲。要完全停止队友而不是重新运行它，返回 JSON `{"continue": false, "stopReason": "..."}` 。 `TeammateIdle` hooks 不支持匹配器，在每次出现时触发。

TeammateIdle 输入

除了 [通用输入字段](#) 外， `TeammateIdle` hooks 还接收 `teammate_name` 和 `team_name`。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "TeammateIdle",
  "teammate_name": "researcher",
  "team_name": "my-project"
}
```

字段	描述
<code>teammate_name</code>	即将空闲的队友的名称
<code>team_name</code>	团队名称

TeammateIdle 决定控制

TeammateIdle hooks 支持两种方式来控制队友行为：

- **退出代码 2：**队友接收 `stderr` 消息作为反馈并继续工作而不是空闲。
- **JSON `{"continue": false, "stopReason": "..."}:`** 完全停止队友，匹配 `Stop` hook 行为。`stopReason` 向用户显示。

此示例检查构建工件是否存在，然后允许队友空闲：

```
#!/bin/bash

if [ ! -f "./dist/output.js" ]; then
    echo "Build artifact missing. Run the build before stopping." >&2
    exit 2
fi

exit 0
```

TaskCompleted

当任务被标记为已完成时运行。这在两种情况下触发：当任何代理通过 `TaskUpdate` 工具显式标记任务为已完成时，或当 `agent team` 队友完成其轮次且有进行中的任务时。使用此来强制执行完成标准，如通过测试或 `lint` 检查，然后任务才能关闭。

当 `TaskCompleted` hook 以代码 2 退出时，任务不被标记为已完成，`stderr` 消息作为反馈反馈给模型。要完全停止队友而不是重新运行它，返回 JSON `{"continue": false, "stopReason": "..."}.` `TaskCompleted` hooks 不支持匹配器，在每次出现时触发。

TaskCompleted 输入

除了 [通用输入字段](#) 外，`TaskCompleted` hooks 还接收 `task_id`、`task_subject` 和可选的 `task_description`、`teammate_name` 和 `team_name`。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "TaskCompleted",
  "task_id": "task-001",
  "task_subject": "Implement user authentication",
  "task_description": "Add login and signup endpoints",
  "teammate_name": "implementer",
  "team_name": "my-project"
}
```

字段	描述
<code>task_id</code>	被完成的任务的标识符
<code>task_subject</code>	任务的标题
<code>task_description</code>	任务的详细描述。可能不存在
<code>teammate_name</code>	完成任务的队友的名称。可能不存在
<code>team_name</code>	团队名称。可能不存在

TaskCompleted 决定控制

TaskCompleted hooks 支持两种方式来控制任务完成：

- **退出代码 2**：任务不被标记为已完成，stderr 消息作为反馈反馈给模型。
- **JSON** `{"continue": false, "stopReason": "...}"`：完全停止队友，匹配 `Stop` hook 行为。`stopReason` 向用户显示。

此示例运行测试并在失败时阻止任务完成：

```
#!/bin/bash
INPUT=$(cat)
TASK_SUBJECT=$(echo "$INPUT" | jq -r '.task_subject')

## 运行测试套件
if ! npm test 2>&1; then
    echo "Tests not passing. Fix failing tests before completing: $TASK_SUBJECT" >&2
    exit 2
fi

exit 0
```

ConfigChange

在会话期间配置文件更改时运行。使用此来审计设置更改、强制执行安全策略或阻止对配置文件的未授权修改。

ConfigChange hooks 对设置文件、托管策略设置和 skill 文件的更改触发。输入中的 `source` 字段告诉您哪种类型的配置更改，可选的 `file_path` 字段提供更改文件的路径。

匹配器在配置源上过滤：

匹配器	何时触发
<code>user_settings</code>	<code>~/.claude/settings.json</code> 更改
<code>project_settings</code>	<code>.claude/settings.json</code> 更改
<code>local_settings</code>	<code>.claude/settings.local.json</code> 更改
<code>policy_settings</code>	托管策略设置更改
<code>skills</code>	<code>.claude/skills/</code> 中的 skill 文件更改

此示例记录所有配置更改以进行安全审计：

```
{
  "hooks": {
    "ConfigChange": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/audit-config-
change.sh"
          }
        ]
      }
    ]
  }
}
```

ConfigChange 输入

除了 [通用输入字段](#) 外，ConfigChange hooks 还接收 `source` 和可选的 `file_path`。
`source` 字段指示哪种配置类型更改，`file_path` 提供被修改的特定文件的路径。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "ConfigChange",
  "source": "project_settings",
  "file_path": "/Users/.../my-project/.claude/settings.json"
}
```

ConfigChange 决定控制

ConfigChange hooks 可以阻止配置更改生效。使用退出代码 2 或 JSON `decision` 来防止更改。被阻止时，新设置不应用于运行中的会话。

字段	描述
<code>decision</code>	<code>"block"</code> 防止配置更改被应用。省略以允许更改
<code>reason</code>	当 <code>decision</code> 为 <code>"block"</code> 时向用户显示的解釋

```
{
  "decision": "block",
  "reason": "Configuration changes to project settings require admin approval"
}
```

`policy_settings` 更改无法被阻止。Hooks 仍然对 `policy_settings` 源触发，因此您可以使用它们进行审计日志记录，但任何阻止决定都被忽略。这确保企业管理的设置总是生效。

WorktreeCreate

当您运行 `claude --worktree` 或 `subagent` 使用 `isolation: "worktree"` 时，Claude Code 使用 `git worktree` 创建隔离的工作副本。如果您配置 WorktreeCreate hook，它替换默认的 git 行为，让您使用不同的版本控制系统，如 SVN、Perforce 或 Mercurial。

hook 必须在 stdout 上打印创建的 worktree 目录的绝对路径。Claude Code 使用此路径作为隔离会话的工作目录。

此示例创建 SVN 工作副本并打印路径供 Claude Code 使用。用您自己的替换仓库 URL：

```
{
  "hooks": {
    "WorktreeCreate": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "bash -c 'NAME=$(jq -r .name); DIR=\"$HOME/.claude/worktrees/$NAME\"; svn checkout https://svn.example.com/repo/trunk \"$DIR\" >&2 && echo \"$DIR\"'"
          }
        ]
      }
    ]
  }
}
```

hook 从 stdin 上的 JSON 输入读取 `worktree name`，将新副本检出到新目录，并打印目录路径。最后一行的 `echo` 是 Claude Code 读取的 `worktree` 路径。将任何其他输出重定向到 `stderr`，以便它不会干扰路径。

WorktreeCreate 输入

除了 [通用输入字段](#) 外，`WorktreeCreate` hooks 还接收 `name` 字段。这是新 `worktree` 的 slug 标识符，由用户指定或自动生成（例如，`bold-oak-a3f2`）。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "hook_event_name": "WorktreeCreate",
  "name": "feature-auth"
}
```

WorktreeCreate 输出

hook 必须在 `stdout` 上打印创建的 `worktree` 目录的绝对路径。如果 hook 失败或不产生输出，`worktree` 创建失败并出现错误。

WorktreeCreate hooks 不使用标准的允许/阻止决定模型。相反，hook 的成功或失败决定结果。仅支持 `type: "command"` hooks。

WorktreeRemove

[WorktreeCreate](#) 的清理对应物。此 hook 在 worktree 被移除时触发，要么当您退出 `--worktree` 会话并选择移除它时，要么当具有 `isolation: "worktree"` 的 subagent 完成时。对于基于 git 的 worktrees，Claude 使用 `git worktree remove` 自动处理清理。如果您为非 git 版本控制系统配置了 WorktreeCreate hook，将其与 WorktreeRemove hook 配对以处理清理。没有它，worktree 目录留在磁盘上。

Claude Code 将 WorktreeCreate 在 stdout 上打印的路径作为 `worktree_path` 在 hook 输入中传递。此示例读取该路径并移除目录：

```
{
  "hooks": {
    "WorktreeRemove": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "bash -c 'jq -r .worktree_path | xargs rm -rf'"
          }
        ]
      }
    ]
  }
}
```

WorktreeRemove 输入

除了 [通用输入字段](#) 外，WorktreeRemove hooks 还接收 `worktree_path` 字段，这是被移除的 worktree 的绝对路径。


```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "hook_event_name": "WorktreeRemove",
  "worktree_path": "/Users/.../my-project/.claude/worktrees/feature-auth"
}
```

WorktreeRemove hooks 没有决定控制。它们无法阻止 worktree 移除，但可以执行清理任务，如移除版本控制状态或存档更改。Hook 失败仅在调试模式下记录。仅支持 `type: "command"` hooks。

PreCompact

在 Claude Code 即将运行压缩操作之前运行。

匹配器值指示压缩是手动还是自动触发：

匹配器	何时触发
<code>manual</code>	<code>/compact</code>
<code>auto</code>	当上下文窗口满时自动压缩

PreCompact 输入

除了 [通用输入字段](#) 外，PreCompact hooks 还接收 `trigger` 和 `custom_instructions`。对于 `manual`，`custom_instructions` 包含用户传入 `/compact` 的内容。对于 `auto`，`custom_instructions` 为空。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PreCompact",
  "trigger": "manual",
  "custom_instructions": ""
}
```

SessionEnd

在 Claude Code 会话结束时运行。对于清理任务、记录会话统计或保存会话状态很有用。支持匹配器来按退出原因过滤。

hook 输入中的 `reason` 字段指示会话为何结束：

原因	描述
<code>clear</code>	会话用 <code>/clear</code> 命令清除
<code>logout</code>	用户登出
<code>prompt_input_exit</code>	用户在提示输入可见时退出
<code>bypass_permissions_disabled</code>	绕过权限模式被禁用
<code>other</code>	其他退出原因

SessionEnd 输入

除了 [通用输入字段](#) 外，SessionEnd hooks 还接收 `reason` 字段，指示会话为何结束。有关所有值，请参阅上面的 [原因表](#)。

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SessionEnd",
  "reason": "other"
}
```

SessionEnd hooks 没有决定控制。它们无法阻止会话终止，但可以执行清理任务。

基于提示的 Hooks

除了命令和 HTTP hooks 外，Claude Code 还支持基于提示的 hooks（`type: "prompt"`），使用 LLM 来评估是否允许或阻止操作，以及代理 hooks（`type: "agent"`），生成具有工具访问权限的代理验证器。并非所有事件都支持每种 hook 类型。

支持所有四种 hook 类型（`command`、`http`、`prompt` 和 `agent`）的事件：

- `PermissionRequest`
- `PostToolUse`
- `PostToolUseFailure`
- `PreToolUse`
- `Stop`
- `SubagentStop`
- `TaskCompleted`
- `UserPromptSubmit`

仅支持 `type: "command"` hooks 的事件：

- `ConfigChange`
- `InstructionsLoaded`
- `Notification`
- `PreCompact`
- `SessionEnd`
- `SessionStart`
- `SubagentStart`

- `TeammateIdle`
- `WorktreeCreate`
- `WorktreeRemove`

基于提示的 Hooks 如何工作

基于提示的 hooks 不执行 Bash 命令，而是：

1. 将 hook 输入和您的提示发送到 Claude 模型，默认为 Haiku
2. LLM 用包含决定的结构化 JSON 响应
3. Claude Code 自动处理决定

提示 hook 配置

将 `type` 设置为 `"prompt"` 并提供 `prompt` 字符串而不是 `command`。使用 `$ARGUMENTS` 占位符将 hook 的 JSON 输入数据注入到您的提示文本中。Claude Code 将组合的提示和输入发送到快速 Claude 模型，该模型返回 JSON 决定。

此 `Stop` hook 要求 LLM 在允许 Claude 完成之前评估是否应该停止：

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "Evaluate if Claude should stop: $ARGUMENTS. Check if all
tasks are complete."
          }
        ]
      }
    ]
  }
}
```

字段	必需	描述
<code>type</code>	是	必须是 <code>"prompt"</code>

字段	必需	描述
<code>prompt</code>	是	要发送给 LLM 的提示文本。使用 <code>\$ARGUMENTS</code> 作为 hook 输入 JSON 的占位符。如果 <code>\$ARGUMENTS</code> 不存在，输入 JSON 被追加到提示
<code>model</code>	否	用于评估的模型。默认为快速模型
<code>timeout</code>	否	超时（秒）。默认值：30

响应架构

LLM 必须用包含以下内容的 JSON 响应：

```
{
  "ok": true | false,
  "reason": "Explanation for the decision"
}
```

字段	描述
<code>ok</code>	<code>true</code> 允许操作， <code>false</code> 防止它
<code>reason</code>	当 <code>ok</code> 为 <code>false</code> 时必需。向 Claude 显示的解釋

示例：多条件 Stop hook

此 `Stop` hook 使用详细提示在允许 Claude 停止之前检查三个条件。如果 `"ok"` 为 `false`，Claude 继续工作，提供的原因作为其下一条指令。`SubagentStop` hooks 使用相同的格式来评估 `subagent` 是否应该停止：

```

{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "You are evaluating whether Claude should stop working.
Context: $ARGUMENTS\n\nAnalyze the conversation and determine if:\n1. All user-
requested tasks are complete\n2. Any errors need to be addressed\n3. Follow-up
work is needed\n\nRespond with JSON: {\"ok\": true} to allow stopping, or
{\"ok\": false, \"reason\": \"your explanation\"} to continue working.",
            "timeout": 30
          }
        ]
      }
    ]
  }
}

```

基于代理的 Hooks

基于代理的 hooks（`type: "agent"`）类似于基于提示的 hooks，但具有多轮工具访问。代理 hook 不是单个 LLM 调用，而是生成一个可以读取文件、搜索代码和检查代码库来验证条件的 subagent。代理 hooks 支持与基于提示的 hooks 相同的事件。

基于代理的 Hooks 如何工作

当代理 hook 触发时：

1. Claude Code 生成一个 subagent，带有您的提示和 hook 的 JSON 输入
2. subagent 可以使用 Read、Grep 和 Glob 等工具来调查
3. 在最多 50 轮后，subagent 返回结构化的 `{ "ok": true/false }` 决定
4. Claude Code 以与提示 hook 相同的方式处理决定

代理 hooks 在验证需要检查实际文件或测试输出时很有用，而不仅仅是单独评估 hook 输入数据。

代理 hook 配置

将 `type` 设置为 `"agent"` 并提供 `prompt` 字符串。配置字段与 [提示 hooks](#) 相同，超时更长：

字段	必需	描述
<code>type</code>	是	必须是 <code>"agent"</code>
<code>prompt</code>	是	描述要验证的内容的提示。 使用 <code>\$ARGUMENTS</code> 作为 hook 输入 JSON 的占位符
<code>model</code>	否	要使用的模型。默认为快速 模型
<code>timeout</code>	否	超时（秒）。默认值：60

响应架构与提示 hooks 相同：`{ "ok": true }` 允许或 `{ "ok": false, "reason": "..."}` 阻止。

此 `Stop` hook 在允许 Claude 完成之前验证所有单元测试通过：

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "agent",
            "prompt": "Verify that all unit tests pass. Run the test suite and  
check the results. $ARGUMENTS",
            "timeout": 120
          }
        ]
      }
    ]
  }
}
```

在后台运行 Hooks

默认情况下，hooks 阻止 Claude 的执行，直到它们完成。对于长时间运行的任务，如部署、测试套件或外部 API 调用，设置 `"async": true` 以在后台运行 hook，同时 Claude 继续工作。异步 hooks 无法阻止或控制 Claude 的行为：响应字段如 `decision`、`permissionDecision` 和 `continue` 无效，因为它们会控制的操作已经完成。

配置异步 hook

向命令 hook 的配置添加 `"async": true` 以在后台运行它而不阻止 Claude。此字段仅在 `type: "command"` hooks 上可用。

此 hook 在每个 `Write` 工具调用后运行测试脚本。Claude 立即继续工作，同时 `run-tests.sh` 执行最多 120 秒。脚本完成时，其输出在下一个对话轮次上传递：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/run-tests.sh",
            "async": true,
            "timeout": 120
          }
        ]
      }
    ]
  }
}
```

`timeout` 字段设置后台进程的最大时间（秒）。如果未指定，异步 hooks 使用与同步 hooks 相同的 10 分钟默认值。

异步 Hooks 如何执行

当异步 hook 触发时，Claude Code 启动 hook 进程并立即继续，不等待它完成。hook 在 `stdin` 上接收与同步 hook 相同的 JSON 输入。

后台进程退出后，如果 hook 产生了带有 `systemMessage` 或 `additionalContext` 字段的 JSON 响应，该内容在下一个对话轮次作为上下文传递给 Claude。

示例：文件更改后运行测试

此 hook 在 Claude 写入文件时在后台启动测试套件，然后在测试完成时将结果报告回 Claude。将此脚本保存到项目中的 `.claude/hooks/run-tests-async.sh` 并使用 `chmod +x` 使其可执行：

```
#!/bin/bash
## run-tests-async.sh

## 从 stdin 读取 hook 输入
INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // empty')

## 仅对源文件运行测试
if [[ "$FILE_PATH" != *.ts && "$FILE_PATH" != *.js ]]; then
    exit 0
fi

## 运行测试并通过 systemMessage 报告结果
RESULT=$(npm test 2>&1)
EXIT_CODE=$?

if [ $EXIT_CODE -eq 0 ]; then
    echo "{\"systemMessage\": \"Tests passed after editing $FILE_PATH\"}"
else
    echo "{\"systemMessage\": \"Tests failed after editing $FILE_PATH: $RESULT\"}"
fi
```

然后将此配置添加到项目根目录中的 `.claude/settings.json`。 `async: true` 标志让 Claude 在测试运行时继续工作：

```

{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "\\\"$CLAUDE_PROJECT_DIR\\\"/.claude/hooks/run-tests-async.sh",
            "async": true,
            "timeout": 300
          }
        ]
      }
    ]
  }
}

```

限制

异步 hooks 与同步 hooks 相比有几个限制：

- 仅 `type: "command"` hooks 支持 `async`。基于提示的 hooks 无法异步运行。
- 异步 hooks 无法阻止工具调用或返回决定。当 hook 完成时，触发操作已经进行。
- Hook 输出在下一个对话轮次上传递。如果会话空闲，响应等待直到下一个用户交互。
- 每次执行创建一个单独的后台进程。同一异步 hook 的多次触发之间没有去重。

安全考虑

免责声明

命令 hooks 以您的系统用户的完整权限运行。

Warning:

命令 hooks 以您的完整用户权限执行 shell 命令。它们可以修改、删除或访问您的用户帐户可以访问的任何文件。在将任何 hook 命令添加到您的配置之前，请审查并测试它们。

安全最佳实践

编写 hooks 时请记住这些实践：

- **验证和清理输入**：永远不要盲目信任输入数据
- **始终引用 shell 变量**：使用 `"$VAR"` 而不是 `$VAR`
- **阻止路径遍历**：检查文件路径中的 `..`
- **使用绝对路径**：为脚本指定完整路径，使用 `"$CLAUDE_PROJECT_DIR"` 作为项目根目录
- **跳过敏感文件**：避免 `.env`、`.git/`、密钥等

调试 Hooks

运行 `claude --debug` 以查看 hook 执行详细信息，包括哪些 hooks 匹配、它们的退出代码和输出。使用 `Ctrl+O` 切换详细模式以在成绩单中查看 hook 进度。

```
[DEBUG] Executing hooks for PostToolUse:Write
[DEBUG] Getting matching hook commands for PostToolUse with query: Write
[DEBUG] Found 1 hook matchers in settings
[DEBUG] Matched 1 hooks for query "Write"
[DEBUG] Found 1 hook commands to execute
[DEBUG] Executing hook command: <Your command> with timeout 600000ms
[DEBUG] Hook command completed with status 0: <Your stdout>
```

有关故障排除常见问题，如 hooks 不触发、无限 Stop hook 循环或配置错误，请参阅指南中的 [限制和故障排除](#)。

使用 hooks 自动化 workflow

当 Claude Code 编辑文件、完成任务或需要输入时自动运行 shell 命令。格式化代码、发送通知、验证命令并强制执行项目规则。

Hooks 是用户定义的 shell 命令，在 Claude Code 生命周期中的特定点执行。它们对 Claude Code 的行为提供确定性控制，确保某些操作始终发生，而不是依赖 LLM 选择运行它们。使用 hooks 来强制执行项目规则、自动化重复任务，并将 Claude Code 与现有工具集成。

对于需要判断而不是确定性规则的决策，你也可以使用 [prompt-based hooks](#) 或 [agent-based hooks](#)，它们使用 Claude 模型来评估条件。

有关扩展 Claude Code 的其他方式，请参阅 [skills](#)，用于为 Claude 提供额外的指令和可执行命令，[subagents](#) 用于在隔离的上下文中运行任务，以及 [plugins](#) 用于打包要在项目间共享的扩展。

Tip:

本指南涵盖常见用例和入门方法。有关完整的事件架构、JSON 输入/输出格式和高级功能（如异步 hooks 和 MCP 工具 hooks），请参阅 [Hooks 参考](#)。

设置你的第一个 hook

在 Claude Code 中创建 hook 的最快方法是通过 `/hooks` 交互式菜单。本演练创建一个桌面通知 hook，这样每当 Claude 等待你的输入而不是监视终端时，你都会收到警报。

Step 1: 打开 hooks 菜单

在 Claude Code CLI 中输入 `/hooks`。你将看到所有可用 hook 事件的列表，以及禁用所有 hooks 的选项。每个事件对应 Claude 生命周期中的一个点，你可以在该点运行自定义代码。选择 `Notification` 来创建一个在 Claude 需要你关注时触发的 hook。

Step 2: 配置匹配器

菜单显示匹配器列表，这些匹配器过滤 hook 何时触发。将匹配器设置为 `*` 以在所有通知类型上触发。你可以稍后通过将匹配器更改为特定值（如 `permission_prompt` 或 `idle_prompt`）来缩小范围。

Step 3: 添加你的命令

选择 **+ Add new hook...**。菜单会提示你输入事件触发时要运行的 shell 命令。Hooks 可以运行你提供的任何 shell 命令，因此你可以使用平台的内置通知工具。复制适用于你的操作系统的命令：

macOS

使用 **osascript** 通过 AppleScript 触发原生 macOS 通知：

```
osascript -e 'display notification "Claude Code needs your attention" with title "Claude Code"'
```

Linux

使用 **notify-send**，它在大多数带有通知守护程序的 Linux 桌面上预装：

```
notify-send 'Claude Code' 'Claude Code needs your attention'
```

Windows (PowerShell)

使用 PowerShell 通过 .NET 的 Windows Forms 显示原生消息框：

```
powershell.exe -Command "[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms'); [System.Windows.Forms.MessageBox]::Show('Claude Code needs your attention', 'Claude Code')"
```

Step 4: 选择存储位置

菜单询问在哪里保存 hook 配置。选择 **User settings** 将其存储在 `~/.claude/settings.json` 中，这会将 hook 应用于所有项目。你也可以选择 **Project settings** 将其限制在当前项目。有关所有可用范围，请参阅 [配置 hook 位置](#)。

Step 5: 测试 hook

按 **Esc** 返回 CLI。要求 Claude 做需要权限的事情，然后切换离开终端。你应该会收到桌面通知。

你可以自动化什么

Hooks 让你在 Claude Code 生命周期中的关键点运行代码：编辑后格式化文件、在执行前阻止命令、在 Claude 需要输入时发送通知、在会话开始时注入上下文等。有关完整的 hook 事件列表，请参阅 [Hooks 参考](#)。

每个示例都包含一个现成的配置块，你可以将其添加到 [settings 文件](#)。最常见的模式：

- [在 Claude 需要输入时获得通知](#)
- [编辑后自动格式化代码](#)
- [阻止对受保护文件的编辑](#)
- [压缩后重新注入上下文](#)
- [审计配置更改](#)

在 Claude 需要输入时获得通知

每当 Claude 完成工作并需要你的输入时获得桌面通知，这样你可以切换到其他任务而无需检查终端。

此 hook 使用 `Notification` 事件，当 Claude 等待输入或权限时触发。下面的每个选项卡使用平台的原生通知命令。将其添加到 `~/.claude/settings.json`，或使用上面的 [交互式演练](#) 通过 `/hooks` 配置它：

macOS

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "osascript -e 'display notification \"Claude Code needs your attention\" with title \"Claude Code\"'"
          }
        ]
      }
    ]
  }
}
```

Linux

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "notify-send 'Claude Code' 'Claude Code needs your
attention'"
          }
        ]
      }
    ]
  }
}
```

Windows (PowerShell)

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "powershell.exe -Command \"[System.Reflection.Assembly]::LoadWithPartialName('System.Windows.Forms'); [System.Windows.Forms.MessageBox]::Show('Claude Code needs your attention', 'Claude Code')\"";
          }
        ]
      }
    ]
  }
}
```

编辑后自动格式化代码

在 Claude 编辑的每个文件上自动运行 [Prettier](#)，以便格式保持一致而无需手动干预。

此 hook 使用 `PostToolUse` 事件和 `Edit|Write` 匹配器，因此它仅在文件编辑工具之后运行。该命令使用 `jq` 提取编辑的文件路径并将其传递给 Prettier。将其添加到项目根目录中的 `.claude/settings.json`：


```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.file_path' | xargs npx prettier --
write"
          }
        ]
      }
    ]
  }
}
```

Note:

本页上的 Bash 示例使用 `jq` 进行 JSON 解析。使用 `brew install jq` (macOS) 、 `apt-get install jq` (Debian/Ubuntu) 安装它，或参阅 [jq 下载](#)。

阻止对受保护文件的编辑

防止 Claude 修改敏感文件，如 `.env` 、 `package-lock.json` 或 `.git/` 中的任何内容。Claude 会收到解释编辑被阻止原因的反馈，因此它可以调整其方法。

此示例使用 hook 调用的单独脚本文件。该脚本根据受保护模式列表检查目标文件路径，并以代码 2 退出以阻止编辑。

Step 1: 创建 hook 脚本

将其保存到 `.claude/hooks/protect-files.sh`：

```
#!/bin/bash
## protect-files.sh

INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // empty')

PROTECTED_PATTERNS=( ".env" "package-lock.json" ".git/" )

for pattern in "${PROTECTED_PATTERNS[@]}; do
  if [ [ "$FILE_PATH" == *"$pattern"* ]; then
    echo "Blocked: $FILE_PATH matches protected pattern '$pattern'" >&2
    exit 2
  fi
done

exit 0
```

Step 2: 使脚本可执行 (macOS/Linux)

Hook 脚本必须可执行才能让 Claude Code 运行它们：

```
chmod +x .claude/hooks/protect-files.sh
```

Step 3: 注册 hook

将 `PreToolUse` hook 添加到 `.claude/settings.json`，在任何 `Edit` 或 `Write` 工具调用之前运行脚本：

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "\\\"$CLAUDE_PROJECT_DIR\\\"/.claude/hooks/protect-files.sh"
          }
        ]
      }
    ]
  }
}
```

压缩后重新注入上下文

当 Claude 的上下文窗口填满时，压缩会总结对话以释放空间。这可能会丢失重要细节。使用带有 `compact` 匹配器的 `SessionStart` hook 在每次压缩后重新注入关键上下文。

你的命令写入 stdout 的任何文本都会添加到 Claude 的上下文中。此示例提醒 Claude 项目约定和最近的工作。将其添加到项目根目录中的 `.claude/settings.json`：

```

{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "compact",
        "hooks": [
          {
            "type": "command",
            "command": "echo 'Reminder: use Bun, not npm. Run bun test before committing. Current sprint: auth refactor.'"
          }
        ]
      }
    ]
  }
}

```

你可以用任何产生动态输出的命令替换 `echo`，如 `git log --oneline -5` 来显示最近的提交。有关在每个会话开始时注入上下文，请考虑改用 [CLAUDE.md](#)。有关环境变量，请参阅参考中的 [CLAUDE_ENV_FILE](#)。

审计配置更改

跟踪会话期间设置或技能文件何时更改。`ConfigChange` 事件在外部进程或编辑器修改配置文件时触发，因此你可以记录更改以进行合规性检查或阻止未授权的修改。

此示例将每个更改附加到审计日志。将其添加到 `~/.claude/settings.json`：

```
{
  "hooks": {
    "ConfigChange": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "jq -c '{timestamp: now | todate, source: .source,
file: .file_path}' >> ~/claude-config-audit.log"
          }
        ]
      }
    ]
  }
}
```

匹配器按配置类型过滤：`user_settings`、`project_settings`、`local_settings`、`policy_settings` 或 `skills`。要阻止更改生效，以代码 2 退出或返回 `{"decision": "block"}`。有关完整的输入架构，请参阅 [ConfigChange 参考](#)。

Hooks 如何工作

Hook 事件在 Claude Code 中的特定生命周期点触发。当事件触发时，所有匹配的 hooks 并行运行，相同的 hook 命令会自动去重。下表显示每个事件及其触发时间：

Event	When it fires
<code>SessionStart</code>	When a session begins or resumes
<code>UserPromptSubmit</code>	When you submit a prompt, before Claude processes it
<code>PreToolUse</code>	Before a tool call executes. Can block it
<code>PermissionRequest</code>	When a permission dialog appears
<code>PostToolUse</code>	After a tool call succeeds
<code>PostToolUseFailure</code>	After a tool call fails
<code>Notification</code>	When Claude Code sends a notification

Event	When it fires
SubagentStart	When a subagent is spawned
SubagentStop	When a subagent finishes
Stop	When Claude finishes responding
TeammateIdle	When an agent team teammate is about to go idle
TaskCompleted	When a task is being marked as completed
InstructionsLoaded	When a CLAUDE.md or <code>.claude/rules/*.md</code> file is loaded into context. Fires at session start and when files are lazily loaded during a session
ConfigChange	When a configuration file changes during a session
WorktreeCreate	When a worktree is being created via <code>--worktree</code> or <code>isolation: "worktree"</code> . Replaces default git behavior
WorktreeRemove	When a worktree is being removed, either at session exit or when a subagent finishes
PreCompact	Before context compaction
PostCompact	After context compaction completes
Elicitation	When an MCP server requests user input during a tool call
ElicitationResult	After a user responds to an MCP elicitation, before the response is sent back to the server
SessionEnd	When a session terminates

每个 hook 都有一个 `type` 来确定它如何运行。大多数 hooks 使用 `"type": "command"`，它运行 shell 命令。还有三种其他类型可用：

- `"type": "http"`：将事件数据 POST 到 URL。请参阅 [HTTP hooks](#)。
- `"type": "prompt"`：单轮 LLM 评估。请参阅 [Prompt-based hooks](#)。
- `"type": "agent"`：具有工具访问权限的多轮验证。请参阅 [Agent-based hooks](#)。

读取输入并返回输出

Hooks 通过 stdin、stdout、stderr 和退出代码与 Claude Code 通信。当事件触发时，Claude Code 将事件特定的数据作为 JSON 传递到脚本的 stdin。你的脚本读取该数据，完成其工作，并通过退出代码告诉 Claude Code 接下来要做什么。

Hook 输入

每个事件都包含常见字段，如 `session_id` 和 `cwd`，但每个事件类型添加不同的数据。例如，当 Claude 运行 Bash 命令时，`PreToolUse` hook 在 stdin 上接收类似以下内容：

```
{
  "session_id": "abc123",           // 此会话的唯一 ID
  "cwd": "/Users/sarah/myproject", // 事件触发时的工作目录
  "hook_event_name": "PreToolUse", // 哪个事件触发了此 hook
  "tool_name": "Bash",             // Claude 即将使用的工具
  "tool_input": {                  // Claude 传递给工具的参数
    "command": "npm test"          // 对于 Bash，这是 shell 命令
  }
}
```

你的脚本可以解析该 JSON 并对任何这些字段进行操作。`UserPromptSubmit` hooks 获取 `prompt` 文本，`SessionStart` hooks 获取 `source`（`startup`、`resume`、`clear`、`compact`），等等。有关共享字段，请参阅参考中的 [常见输入字段](#)，以及每个事件的部分了解事件特定的架构。

Hook 输出

你的脚本通过写入 stdout 或 stderr 并以特定代码退出来告诉 Claude Code 接下来要做什么。例如，一个想要阻止命令的 `PreToolUse` hook：

```
#!/bin/bash
INPUT=$(cat)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command')

if echo "$COMMAND" | grep -q "drop table"; then
    echo "Blocked: dropping tables is not allowed" >&2 # stderr 变成 Claude 的反馈
    exit 2 # exit 2 = 阻止操作
fi

exit 0 # exit 0 = 让它继续
```

退出代码确定接下来会发生什么：

- **Exit 0**：操作继续。对于 `UserPromptSubmit` 和 `SessionStart` hooks，你写入 stdout 的任何内容都会添加到 Claude 的上下文中。
- **Exit 2**：操作被阻止。写入原因到 stderr，Claude 会收到它作为反馈，以便它可以调整。
- **任何其他退出代码**：操作继续。Stderr 被记录但不显示给 Claude。使用 `Ctrl+O` 切换详细模式以在记录中查看这些消息。

结构化 JSON 输出

退出代码给你两个选项：允许或阻止。为了获得更多控制，以 exit 0 退出并改为将 JSON 对象打印到 stdout。

Note:

使用 exit 2 通过 stderr 消息阻止，或使用 exit 0 和 JSON 进行结构化控制。不要混合它们：Claude Code 在你以 exit 2 退出时忽略 JSON。

例如，`PreToolUse` hook 可以拒绝工具调用并告诉 Claude 为什么，或将其升级给用户以获得批准：


```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Use rg instead of grep for better performance"
  }
}
```

Claude Code 读取 `permissionDecision` 并取消工具调用，然后将 `permissionDecisionReason` 反馈给 Claude。这三个选项特定于 `PreToolUse`：

- `"allow"`：继续而不显示权限提示
- `"deny"`：取消工具调用并将原因发送给 Claude
- `"ask"`：照常向用户显示权限提示

其他事件使用不同的决策模式。例如，`PostToolUse` 和 `Stop` hooks 使用顶级 `decision: "block"` 字段，而 `PermissionRequest` 使用 `hookSpecificOutput.decision.behavior`。有关按事件的完整分解，请参阅参考中的 [摘要表](#)。

对于 `UserPromptSubmit` hooks，改用 `additionalContext` 将文本注入到 Claude 的上下文中。Prompt-based hooks（`type: "prompt"`）处理输出的方式不同：请参阅 [Prompt-based hooks](#)。

使用匹配器过滤 hooks

没有匹配器，hook 会在其事件的每次出现时触发。匹配器让你缩小范围。例如，如果你只想在文件编辑后运行格式化程序（而不是在每个工具调用后），将匹配器添加到你的 `PostToolUse` hook：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          { "type": "command", "command": "prettier --write ..." }
        ]
      }
    ]
  }
}
```

`"Edit|Write"` 匹配器是一个与工具名称匹配的正则表达式模式。hook 仅在 Claude 使用 `Edit` 或 `Write` 工具时触发，而不是在它使用 `Bash`、`Read` 或任何其他工具时触发。

每个事件类型在特定字段上匹配。匹配器支持精确字符串和正则表达式模式：

事件	匹配器过滤的内容	示例匹配器值
<code>PreToolUse</code> 、 <code>PostToolUse</code> 、 <code>PostToolUseFailure</code> 、 <code>PermissionRequest</code>	工具名称	<code>Bash</code> 、 <code>Edit Write</code> 、 <code>mc__.*</code>
<code>SessionStart</code>	会话如何开始	<code>startup</code> 、 <code>resume</code> 、 <code>clear</code> 、 <code>compact</code>
<code>SessionEnd</code>	会话为什么结束	<code>clear</code> 、 <code>logout</code> 、 <code>prompt_input_exit</code> 、 <code>bypass_permissions_disabled</code> 、 <code>other</code>
<code>Notification</code>	通知类型	<code>permission_prompt</code> 、 <code>idle_prompt</code> 、 <code>auth_success</code> 、 <code>elicitation_dialog</code>
<code>SubagentStart</code>	代理类型	<code>Bash</code> 、 <code>Explore</code> 、 <code>Plan</code> 或自定义代理名称
<code>PreCompact</code>	什么触发了压缩	<code>manual</code> 、 <code>auto</code>

事件	匹配器过滤的内容	示例匹配器值
SubagentStop	代理类型	与 SubagentStart 相同的值
ConfigChange	配置源	user_settings 、 project_settings 、 local_settings 、 policy_settings 、 skills
UserPromptSubmit 、 Stop 、 TeammateIdle 、 TaskCompleted 、 WorktreeCreate 、 WorktreeRemove	不支持匹配器	始终在每次出现时触发

显示不同事件类型上匹配器的更多示例：

记录每个 Bash 命令

仅匹配 Bash 工具调用并将每个命令记录到文件。PostToolUse 事件在命令完成后触发，因此 tool_input.command 包含运行的内容。hook 在 stdin 上接收事件数据作为 JSON，jq -r '.tool_input.command' 仅提取命令字符串，>> 将其附加到日志文件：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.command' >> ~/.claude/command-log.txt"
          }
        ]
      }
    ]
  }
}
```

匹配 MCP 工具

MCP 工具使用与内置工具不同的命名约定：`mcp__<server>__<tool>`，其中 `<server>` 是 MCP 服务器名称，`<tool>` 是它提供的工具。例如，`mcp__github__search_repositories` 或 `mcp__filesystem__read_file`。使用正则表达式匹配器来针对来自特定服务器的所有工具，或使用 `mcp__.*__write.*` 之类的模式跨服务器匹配。有关完整的示例列表，请参阅参考中的 [匹配 MCP 工具](#)。

下面的命令使用 `jq` 从 hook 的 JSON 输入中提取工具名称，并将其写入 `stderr`，其中它显示在详细模式（`Ctrl+O`）中：

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "mcp__github__.*",
        "hooks": [
          {
            "type": "command",
            "command": "echo \"GitHub tool called: $(jq -r '.tool_name')\" >&2"
          }
        ]
      }
    ]
  }
}
```

在会话结束时清理

`SessionEnd` 事件支持会话结束原因的匹配器。此 hook 仅在 `clear` 时触发（当你运行 `/clear` 时），而不是在正常退出时：

```
{
  "hooks": {
    "SessionEnd": [
      {
        "matcher": "clear",
        "hooks": [
          {
            "type": "command",
            "command": "rm -f /tmp/claude-scratch-*.txt"
          }
        ]
      }
    ]
  }
}
```

有关完整的匹配器语法，请参阅 [Hooks 参考](#)。

配置 hook 位置

你添加 hook 的位置决定了其范围：

位置	范围	可共享
<code>~/.claude/settings.json</code>	所有你的项目	否，本地到你的机器
<code>.claude/settings.json</code>	单个项目	是，可以提交到仓库
<code>.claude/settings.local.json</code>	单个项目	否，gitignored
托管策略设置	组织范围	是，管理员控制
<code>Plugin hooks/hooks.json</code>	启用插件时	是，与插件捆绑
<code>Skill</code> 或 <code>agent frontmatter</code>	技能或代理活跃时	是，在组件文件中定义

你也可以在 Claude Code 中使用 `/hooks` 菜单来交互式地添加、删除和查看 hooks。要一次禁用所有 hooks，使用 `/hooks` 菜单底部的切换或在设置文件中设置

```
"disableAllHooks": true
```

通过 `/hooks` 菜单添加的 Hooks 立即生效。如果你在 Claude Code 运行时直接编辑设置文件，更改不会生效，直到你在 `/hooks` 菜单中查看它们或重新启动会话。

Prompt-based hooks

对于需要判断而不是确定性规则的决策，使用 `type: "prompt"` hooks。Claude Code 不运行 shell 命令，而是将你的提示和 hook 的输入数据发送到 Claude 模型（默认为 Haiku）来做出决策。如果你需要更多能力，可以使用 `model` 字段指定不同的模型。

模型的唯一工作是返回一个是/否决策作为 JSON：

- `"ok": true`：操作继续
- `"ok": false`：操作被阻止。模型的 `"reason"` 被反馈给 Claude，以便它可以调整。

此示例使用 `Stop` hook 来询问模型是否所有请求的任务都已完成。如果模型返回 `"ok": false`，Claude 继续工作并使用 `reason` 作为其下一条指令：

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "Check if all tasks are complete. If not, respond with\n{\\\"ok\\\": false, \\\"reason\\\": \\\"what remains to be done\\\"}."
          }
        ]
      }
    ]
  }
}
```

有关完整的配置选项，请参阅参考中的 [Prompt-based hooks](#)。

Agent-based hooks

当验证需要检查文件或运行命令时，使用 `type: "agent"` hooks。与只进行单个 LLM 调用的 prompt hooks 不同，agent hooks 生成一个 subagent，它可以读取文件、搜索代码和使用其他工具来在返回决策前验证条件。

Agent hooks 使用与 prompt hooks 相同的 `"ok"` / `"reason"` 响应格式，但默认超时更长（60 秒）且最多 50 个工具使用轮次。

此示例在允许 Claude 停止前验证测试通过：

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "agent",
            "prompt": "Verify that all unit tests pass. Run the test suite and check the results. $ARGUMENTS",
            "timeout": 120
          }
        ]
      }
    ]
  }
}
```

当 hook 输入数据本身足以做出决策时使用 prompt hooks。当你需要根据代码库的实际状态验证某些内容时使用 agent hooks。

有关完整的配置选项，请参阅参考中的 [Agent-based hooks](#)。

HTTP hooks

使用 `type: "http"` hooks 将事件数据 POST 到 HTTP 端点，而不是运行 shell 命令。端点接收命令 hook 在 stdin 上接收的相同 JSON，并通过使用相同 JSON 格式的 HTTP 响应体返回结果。

HTTP hooks 在你想要 web 服务器、云函数或外部服务处理 hook 逻辑时很有用：例如，一个跨团队记录工具使用事件的共享审计服务。

此示例将每个工具使用 POST 到本地日志服务：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "hooks": [
          {
            "type": "http",
            "url": "http://localhost:8080/hooks/tool-use",
            "headers": {
              "Authorization": "Bearer $MY_TOKEN"
            },
            "allowedEnvVars": ["MY_TOKEN"]
          }
        ]
      }
    ]
  }
}
```

端点应使用与命令 hooks 相同的 [输出格式](#) 返回 JSON 响应体。要阻止工具调用，返回 2xx 响应和适当的 `hookSpecificOutput` 字段。HTTP 状态代码本身无法阻止操作。

标头值支持使用 `$VAR_NAME` 或 `${VAR_NAME}` 语法的环境变量插值。仅解析 `allowedEnvVars` 数组中列出的变量；所有其他 `$VAR` 引用保持为空。

Note:

HTTP hooks 必须通过直接编辑设置 JSON 来配置。 `/hooks` 交互式菜单仅支持添加命令 hooks。

有关完整的配置选项和响应处理，请参阅参考中的 [HTTP hooks](#)。

限制和故障排除

限制

- 命令 hooks 仅通过 stdout、stderr 和退出代码通信。它们无法直接触发命令或工具调用。HTTP hooks 改为通过响应体通信。

- Hook 超时默认为 10 分钟，可通过 `timeout` 字段（以秒为单位）按 hook 配置。
- `PostToolUse` hooks 无法撤销操作，因为工具已经执行。
- `PermissionRequest` hooks 不在 [非交互模式](#)（`-p`）中触发。对于自动化权限决策，使用 `PreToolUse` hooks。
- `Stop` hooks 在 Claude 完成响应时触发，而不仅仅在任务完成时。它们不在用户中断时触发。

Hook 未触发

Hook 已配置但从不执行。

- 运行 `/hooks` 并确认 hook 出现在正确的事件下
- 检查匹配器模式是否与工具名称完全匹配（匹配器区分大小写）
- 验证你是否触发了正确的事件类型（例如，`PreToolUse` 在工具执行前触发，`PostToolUse` 在之后触发）
- 如果在非交互模式（`-p`）中使用 `PermissionRequest` hooks，改用 `PreToolUse`

Hook 输出中的错误

你在记录中看到类似” `PreToolUse hook error: ...` “的消息。

- 你的脚本意外以非零代码退出。通过管道传递示例 JSON 来手动测试它：

```
echo '{"tool_name":"Bash","tool_input":{"command":"ls"}}' | ./my-hook.sh
echo $? # 检查退出代码
```

- 如果你看到” `command not found`”，使用绝对路径或 `$CLAUDE_PROJECT_DIR` 来引用脚本
- 如果你看到” `jq: command not found`”，安装 `jq` 或使用 Python/Node.js 进行 JSON 解析
- 如果脚本根本没有运行，使其可执行：`chmod +x ./my-hook.sh`

`/hooks` 显示未配置 hooks

你编辑了设置文件但 hooks 不出现在菜单中。

- 重新启动会话或打开 `/hooks` 以重新加载。通过 `/hooks` 菜单添加的 Hooks 立即生效，但手动文件编辑需要重新加载。
- 验证你的 JSON 有效（不允许尾随逗号和注释）
- 确认设置文件在正确的位置：`~/.claude/settings.json` 用于项目 hooks，`~/.claude/settings.json` 用于全局 hooks

Stop hook 永远运行

Claude 在无限循环中继续工作而不是停止。

你的 Stop hook 脚本需要检查它是否已经触发了继续。从 JSON 输入中解析

`stop_hook_active` 字段，如果为 `true` 则提前退出：

```
#!/bin/bash
INPUT=$(cat)
if [ "$(echo "$INPUT" | jq -r '.stop_hook_active')" = "true" ]; then
    exit 0 # 允许 Claude 停止
fi
## ... 你的 hook 逻辑的其余部分
```

JSON 验证失败

Claude Code 显示 JSON 解析错误，即使你的 hook 脚本输出有效的 JSON。

当 Claude Code 运行 hook 时，它生成一个 shell，该 shell 源你的配置文件（`~/.zshrc` 或 `~/.bashrc`）。如果你的配置文件包含无条件的 `echo` 语句，该输出会被添加到你的 hook 的 JSON 前面：

```
Shell ready on arm64
{"decision": "block", "reason": "Not allowed"}
```

Claude Code 尝试将其解析为 JSON 并失败。要修复此问题，在你的 shell 配置文件中包装 `echo` 语句，使其仅在交互式 shell 中运行：

```
## 在 ~/.zshrc 或 ~/.bashrc 中
if [[ $- == *i* ]]; then
    echo "Shell ready"
fi
```

`$-` 变量包含 shell 标志，`i` 表示交互式。Hooks 在非交互式 shell 中运行，因此 `echo` 被跳过。

调试技术

使用 `Ctrl+O` 切换详细模式以在记录中查看 hook 输出，或运行 `claude --debug` 以获取完整的执行详情，包括哪些 hooks 匹配及其退出代码。

了解更多

- [Hooks 参考](#)：完整的事件架构、JSON 输出格式、异步 hooks 和 MCP 工具 hooks
- [安全考虑](#)：在共享或生产环境中部署 hooks 前查看
- [Bash 命令验证器示例](#)：完整的参考实现

通过 MCP 将 Claude Code 连接到工具

了解如何使用 Model Context Protocol 将 Claude Code 连接到您的工具。

Claude Code 可以通过 [Model Context Protocol \(MCP\)](#)（一个用于 AI 工具集成的开源标准）连接到数百个外部工具和数据源。MCP 服务器为 Claude Code 提供对您的工具、数据库和 API 的访问权限。

使用 MCP 可以做什么

连接 MCP 服务器后，您可以要求 Claude Code：

- **从问题跟踪器实现功能：**“添加 JIRA 问题 ENG-4521 中描述的功能，并在 GitHub 上创建 PR。”
- **分析监控数据：**“检查 Sentry 和 Statsig 以检查 ENG-4521 中描述的功能的使用情况。”
- **查询数据库：**“根据我们的 PostgreSQL 数据库，查找使用功能 ENG-4521 的 10 个随机用户的电子邮件。”
- **集成设计：**“根据在 Slack 中发布的新 Figma 设计更新我们的标准电子邮件模板”
- **自动化工作流：**“创建 Gmail 草稿，邀请这 10 个用户参加关于新功能的反馈会议。”

流行的 MCP 服务器

以下是一些您可以连接到 Claude Code 的常用 MCP 服务器：

Warning:

使用第三方 MCP 服务器需自担风险 - Anthropic 尚未验证所有这些服务器的正确性或安全性。请确保您信任正在安装的 MCP 服务器。使用可能获取不受信任内容的 MCP 服务器时要特别小心，因为这些可能会使您面临提示注入风险。

Server	Description	Command
Notion	Connect your Notion workspace to search, update, and power workflows across tools	<pre>claude mcp add --transport http notion https://mcp.notion.com/mcp</pre>

Server	Description	Command
Canva	Search, create, autofill, and export Canva designs	<code>claude mcp add --transport http canva https://mcp.canva.com/mcp</code>
Figma	Generate diagrams and better code from Figma context	<code>claude mcp add --transport http figma-remote-mcp https://mcp.figma.com/mcp</code>
Atlassian	Access Jira & Confluence from Claude	<code>claude mcp add --transport http atlassian https://mcp.atlassian.com/v1/mcp</code>
Linear	Manage issues, projects & team workflows in Linear	<code>claude mcp add --transport http linear https://mcp.linear.app/mcp</code>
monday.com	Manage projects, boards, and workflows in monday.com	<code>claude mcp add --transport http monday https://mcp.monday.com/mcp</code>
Intercom	Access to Intercom data for better customer insights	<code>claude mcp add --transport http intercom https://mcp.intercom.com/mcp</code>
Vercel	Analyze, debug, and manage projects and deployments	<code>claude mcp add --transport http vercel https://mcp.vercel.com</code>
Granola	The AI notepad for meetings	<code>claude mcp add --transport http granola https://mcp.granola.ai/mcp</code>

Server	Description	Command
Asana	Connect to Asana to coordinate tasks, projects, and goals	<code>claude mcp add --transport streamable-http asana https://mcp.asana.com/v2/mcp</code>
Miro	Access and create new content on Miro boards	<code>claude mcp add --transport http miro https://mcp.miro.com/</code>
Sentry	Search, query, and debug errors intelligently	<code>claude mcp add --transport http sentry https://mcp.sentry.dev/mcp</code>
Supabase	Manage databases, authentication, and storage	<code>claude mcp add --transport http supabase https://mcp.supabase.com/mcp</code>
Hugging Face	Access the Hugging Face Hub and thousands of Gradio Apps	<code>claude mcp add --transport http hugging-face https://huggingface.co/mcp</code>
Context7	Up-to-date docs for LLMs and AI code editors	<code>claude mcp add --transport http context7 https://mcp.context7.com/mcp</code>
Stripe	Payment processing and financial infrastructure tools	<code>claude mcp add --transport http stripe https://mcp.stripe.com</code>
Microsoft Learn	Search trusted Microsoft docs to power your development	<code>claude mcp add --transport http microsoft-learn https://learn.microsoft.com/api/mcp</code>

Server	Description	Command
Clay	Find prospects. Research accounts. Personalize outreach	<code>claude mcp add --transport http clay https://api.clay.com/v3/mcp</code>
Webflow	Manage Webflow CMS, pages, assets and sites	<code>claude mcp add --transport http webflow https://mcp.webflow.com/mcp</code>
Cloudflare	Build applications with compute, storage, and AI	<code>claude mcp add --transport http cloudflare https://bindings.mcp.cloudflare.com/mcp</code>
Ramp	Search, access, and analyze your Ramp financial data	<code>claude mcp add --transport http ramp https://ramp-mcp-remote.ramp.com/mcp</code>
ZoomInfo	Enrich contacts & accounts with GTM intelligence	<code>claude mcp add --transport http zoominfo https://mcp.zoominfo.com/mcp</code>
Netlify	Create, deploy, manage, and secure websites on Netlify	<code>claude mcp add --transport http netlify https://netlify-mcp.netlify.app/mcp</code>
Make	Run Make scenarios and manage your Make account	<code>claude mcp add --transport http make https://mcp.make.com</code>
GoDaddy	Search domains and check availability	<code>claude mcp add --transport http godaddy https://api.godaddy.com/v1/domains/mcp</code>

Server	Description	Command
Google Cloud BigQuery	BigQuery: Advanced analytical insights for agents	<pre>claude mcp add -- transport http bigquery https:// bigquery.googleapis.c om/mcp</pre>
PayPal	Access PayPal payments platform	<pre>claude mcp add -- transport http paypal https:// mcp.paypal.com/mcp</pre>
PostHog	Query, analyze, and manage your PostHog insights	<pre>claude mcp add -- transport http posthog https:// mcp.posthog.com/mcp</pre>
Similarweb	Real time web, mobile app, and market data	<pre>claude mcp add -- transport http similarweb https:// mcp.similarweb.com</pre>
Crypto.com	Real time prices, orders, charts, and more for crypto	<pre>claude mcp add -- transport http crypto.com https:// mcp.crypto.com/ market-data/mcp</pre>
Attio	Search, manage, and update your Attio CRM from Claude	<pre>claude mcp add -- transport http attio https:// mcp.attio.com/mcp</pre>
Trivago	Find your ideal hotel at the best price	<pre>claude mcp add -- transport http trivago https:// mcp.trivago.com/mcp</pre>
Jam	Record screen and collect automatic context for issues	<pre>claude mcp add -- transport http jam https://mcp.jam.dev/ mcp</pre>

Server	Description	Command
Consensus	Explore scientific research	<code>claude mcp add --transport http consensus https://mcp.consensus.app/mcp</code>
Clockwise	Advanced scheduling and time management for work	<code>claude mcp add --transport http clockwise https://mcp.getclockwise.com/mcp</code>
Square	Search and manage transaction, merchant, and payment data	<code>claude mcp add --transport sse square https://mcp.squareup.com/sse</code>
Egnyte	Securely access and analyze Egnyte content	<code>claude mcp add --transport http egnyte https://mcp-server.egnyte.com/mcp</code>
Pylon	Search and manage Pylon support issues	<code>claude mcp add --transport http pylon https://mcp.usepylon.com/</code>
Honeycomb	Query and explore observability data and SLOs	<code>claude mcp add --transport http honeycomb https://mcp.honeycomb.io/mcp</code>

Note:

需要特定的集成？ [在 GitHub 上查找数百个更多 MCP 服务器](#)，或使用 [MCP SDK](#) 构建您自己的服务器。

安装 MCP 服务器

MCP 服务器可以根据您的需求以三种不同的方式进行配置：

选项 1：添加远程 HTTP 服务器

HTTP 服务器是连接到远程 MCP 服务器的推荐选项。这是云服务最广泛支持的传输方式。

```
## 基本语法
claude mcp add --transport http <name> <url>

## 真实示例：连接到 Notion
claude mcp add --transport http notion https://mcp.notion.com/mcp

## 带有 Bearer 令牌的示例
claude mcp add --transport http secure-api https://api.example.com/mcp \
  --header "Authorization: Bearer your-token"
```

选项 2：添加远程 SSE 服务器

Warning:

SSE (Server-Sent Events) 传输已弃用。请在可用的地方使用 HTTP 服务器。

```
## 基本语法
claude mcp add --transport sse <name> <url>

## 真实示例：连接到 Asana
claude mcp add --transport sse asana https://mcp.asana.com/sse

## 带有身份验证标头的示例
claude mcp add --transport sse private-api https://api.company.com/sse \
  --header "X-API-Key: your-key-here"
```

选项 3：添加本地 stdio 服务器

Stdio 服务器作为您机器上的本地进程运行。它们非常适合需要直接系统访问或自定义脚本的工具。

基本语法

```
claude mcp add [options] <name> -- <command> [args...]
```

真实示例：添加 Airtable 服务器

```
claude mcp add --transport stdio --env AIRTABLE_API_KEY=YOUR_KEY airtable \  
-- npx -y airtable-mcp-server
```

Note:

重要：选项顺序

所有选项（`--transport`、`--env`、`--scope`、`--header`）必须在服务器名称之前。然后 `--`（双破折号）将服务器名称与传递给 MCP 服务器的命令和参数分开。

例如：

- `claude mcp add --transport stdio myserver -- npx server` → 运行 `npx server`
- `claude mcp add --transport stdio --env KEY=value myserver -- python server.py --port 8080` → 运行 `python server.py --port 8080`，环境中有 `KEY=value`

这可以防止 Claude 的标志与服务器标志之间的冲突。

管理您的服务器

配置后，您可以使用这些命令管理您的 MCP 服务器：

列出所有配置的服务器

```
claude mcp list
```

获取特定服务器的详细信息

```
claude mcp get github
```

删除服务器

```
claude mcp remove github
```

（在 Claude Code 中）检查服务器状态

```
/mcp
```

动态工具更新

Claude Code 支持 MCP `list_changed` 通知，允许 MCP 服务器动态更新其可用工具、提示和资源，而无需您断开连接并重新连接。当 MCP 服务器发送 `list_changed` 通知时，Claude Code 会自动刷新来自该服务器的可用功能。

Tip:

提示：

- 使用 `--scope` 标志指定配置的存储位置：
- `local`（默认）：仅在当前项目中对您可用（在较旧版本中称为 `project`）
- `project`：通过 `.mcp.json` 文件与项目中的每个人共享
- `user`：在所有项目中对您可用（在较旧版本中称为 `global`）
- 使用 `--env` 标志设置环境变量（例如，`--env KEY=value`）
- 使用 `MCP_TIMEOUT` 环境变量配置 MCP 服务器启动超时（例如，`MCP_TIMEOUT=10000` `claude` 设置 10 秒超时）
- 当 MCP 工具输出超过 10,000 个令牌时，Claude Code 将显示警告。要增加此限制，请设置 `MAX_MCP_OUTPUT_TOKENS` 环境变量（例如，`MAX_MCP_OUTPUT_TOKENS=50000`）
- 使用 `/mcp` 对需要 OAuth 2.0 身份验证的远程服务器进行身份验证

Warning:

Windows 用户：在本机 Windows（不是 WSL）上，使用 `npm` 的本地 MCP 服务器需要 `cmd /c` 包装器以确保正确执行。

```
## 这创建了 Windows 可以执行的 command="cmd"
claude mcp add --transport stdio my-server -- cmd /c npm -y @some/package
```

没有 `cmd /c` 包装器，您会遇到“连接已关闭”错误，因为 Windows 无法直接执行 `npm`。（有关 `--` 参数的说明，请参阅上面的注释。）

插件提供的 MCP 服务器

插件可以捆绑 MCP 服务器，在启用插件时自动提供工具和集成。插件 MCP 服务器的工作方式与用户配置的服务器相同。

插件 MCP 服务器的工作原理：

- 插件在插件根目录的 `.mcp.json` 中或在 `plugin.json` 中内联定义 MCP 服务器
- 启用插件时，其 MCP 服务器会自动启动
- 插件 MCP 工具与手动配置的 MCP 工具一起出现

- 插件服务器通过插件安装进行管理（不是 `/mcp` 命令）

示例插件 MCP 配置：

在插件根目录的 `.mcp.json` 中：

```
{
  "database-tools": {
    "command": "${CLAUDE_PLUGIN_ROOT}/servers/db-server",
    "args": ["--config", "${CLAUDE_PLUGIN_ROOT}/config.json"],
    "env": {
      "DB_URL": "${DB_URL}"
    }
  }
}
```

或在 `plugin.json` 中内联：

```
{
  "name": "my-plugin",
  "mcpServers": {
    "plugin-api": {
      "command": "${CLAUDE_PLUGIN_ROOT}/servers/api-server",
      "args": ["--port", "8080"]
    }
  }
}
```

插件 MCP 功能：

- **自动生命周期：**服务器在插件启用时启动，但您必须重新启动 Claude Code 以应用 MCP 服务器更改（启用或禁用）
- **环境变量：**对插件相对路径使用 `${CLAUDE_PLUGIN_ROOT}`
- **用户环境访问：**访问与手动配置的服务器相同的环境变量
- **多种传输类型：**支持 stdio、SSE 和 HTTP 传输（传输支持可能因服务器而异）

查看插件 MCP 服务器：

```
## 在 Claude Code 中，查看所有 MCP 服务器，包括插件服务器  
/mcp
```

插件服务器在列表中出现，并带有指示它们来自插件的指示符。

插件 MCP 服务器的优势：

- **捆绑分发：**工具和服务器打包在一起
- **自动设置：**无需手动 MCP 配置
- **团队一致性：**安装插件时每个人都获得相同的工具

有关使用插件捆绑 MCP 服务器的详细信息，请参阅[插件组件参考](#)。

MCP 安装范围

MCP 服务器可以在三个不同的范围级别进行配置，每个级别都用于管理服务器可访问性和共享的不同目的。了解这些范围可以帮助您确定为特定需求配置服务器的最佳方式。

本地范围

本地范围的服务器代表默认配置级别，存储在您项目路径下的 `~/.claude.json` 中。这些服务器对您保持私密，仅在当前项目目录中工作时可访问。此范围非常适合个人开发服务器、实验配置或包含不应共享的敏感凭据的服务器。

Note:

MCP 服务器的“本地范围”术语与一般本地设置不同。MCP 本地范围的服务器存储在 `~/.claude.json`（您的主目录）中，而一般本地设置使用 `.claude/settings.local.json`（在项目目录中）。有关设置文件位置的详细信息，请参阅[设置](#)。

```
## 添加本地范围的服务器（默认）  
claude mcp add --transport http stripe https://mcp.stripe.com  
  
## 显式指定本地范围  
claude mcp add --transport http stripe --scope local https://mcp.stripe.com
```

项目范围

项目范围的服务器通过在项目根目录中存储配置在 `.mcp.json` 文件中来启用团队协作。此文件设计为检入版本控制，确保所有团队成员都可以访问相同的 MCP 工具和服务。添加项目范围的服务器时，Claude Code 会自动创建或更新此文件，使用适当的配置结构。

```
## 添加项目范围的服务器
```

```
claude mcp add --transport http paypal --scope project https://mcp.paypal.com/mcp
```

生成的 `.mcp.json` 文件遵循标准化格式：

```
{
  "mcpServers": {
    "shared-server": {
      "command": "/path/to/server",
      "args": [],
      "env": {}
    }
  }
}
```

出于安全原因，Claude Code 在使用来自 `.mcp.json` 文件的项目范围的服务器之前会提示批准。如果您需要重置这些批准选择，请使用 `claude mcp reset-project-choices` 命令。

用户范围

用户范围的服务器存储在 `~/.claude.json` 中，并提供跨项目可访问性，使其在您机器上的所有项目中可用，同时对您的用户帐户保持私密。此范围适用于个人实用程序服务器、开发工具或您在不同项目中经常使用的服务。

```
## 添加用户服务器
```

```
claude mcp add --transport http hubspot --scope user https://mcp.hubspot.com/anthropic
```

选择正确的范围

根据以下条件选择您的范围：

- **本地范围：**个人服务器、实验配置或特定于一个项目的敏感凭据
- **项目范围：**团队共享的服务器、项目特定的工具或协作所需的服务

- **用户范围：**跨多个项目需要的个人实用程序、开发工具或经常使用的服务

Note:

MCP 服务器存储在哪里？

- **用户和本地范围：** `~/.claude.json` （在 `mcpServers` 字段或项目路径下）
- **项目范围：**项目根目录中的 `.mcp.json` （检入源代码控制）
- **托管：**系统目录中的 `managed-mcp.json` （请参阅[托管 MCP 配置](#)）

范围层次结构和优先级

MCP 服务器配置遵循清晰的优先级层次结构。当具有相同名称的服务器存在于多个范围时，系统通过首先优先考虑本地范围的服务器、其次是项目范围的服务器，最后是用用户范围的服务器来解决冲突。此设计确保个人配置可以在需要时覆盖共享配置。

`.mcp.json` 中的环境变量扩展

Claude Code 支持 `.mcp.json` 文件中的环境变量扩展，允许团队共享配置，同时为特定于机器的路径和 API 密钥等敏感值保持灵活性。

支持的语法：

- `${VAR}` - 扩展为环境变量 `VAR` 的值
- `${VAR:-default}` - 如果设置了 `VAR`，则扩展为 `VAR`，否则使用 `default`

扩展位置：环境变量可以在以下位置扩展：

- `command` - 服务器可执行文件路径
- `args` - 命令行参数
- `env` - 传递给服务器的环境变量
- `url` - 对于 HTTP 服务器类型
- `headers` - 对于 HTTP 服务器身份验证

带有变量扩展的示例：


```
{
  "mcpServers": {
    "api-server": {
      "type": "http",
      "url": "${API_BASE_URL:-https://api.example.com}/mcp",
      "headers": {
        "Authorization": "Bearer ${API_KEY}"
      }
    }
  }
}
```

如果未设置所需的环境变量且没有默认值，Claude Code 将无法解析配置。

实际示例

{/* ### 示例：使用 Playwright 自动化浏览器测试

```
claude mcp add --transport stdio playwright -- npx -y @playwright/mcp@latest
```

然后编写并运行浏览器测试：

```
Test if the login flow works with test@example.com
```

```
Take a screenshot of the checkout page on mobile
```

```
Verify that the search feature returns results
``` */}

示例：使用 Sentry 监控错误

```bash theme={null}
claude mcp add --transport http sentry https://mcp.sentry.dev/mcp
```

使用您的 Sentry 帐户进行身份验证：

```
/mcp
```

然后调试生产问题：

过去 24 小时内最常见的错误是什么？

显示我错误 ID abc123 的堆栈跟踪

哪个部署引入了这些新错误？

示例：连接到 GitHub 进行代码审查

```
claude mcp add --transport http github https://api.githubcopilot.com/mcp/
```

如果需要，通过为 GitHub 选择”身份验证”进行身份验证：

```
/mcp
```

然后使用 GitHub：

审查 PR #456 并建议改进

为我们刚发现的错误创建新问题

显示分配给我的所有开放 PR

示例：查询您的 PostgreSQL 数据库

```
claude mcp add --transport stdio db -- npx -y @bytebase/dbhub \
--dsn "postgresql://readonly:pass@prod.db.com:5432/analytics"
```

然后自然地查询您的数据库：

本月我们的总收入是多少？

显示订单表的架构

查找 90 天内未进行购买的客户

使用远程 MCP 服务器进行身份验证

许多基于云的 MCP 服务器需要身份验证。Claude Code 支持 OAuth 2.0 以实现安全连接。

Step 1: 添加需要身份验证的服务器

例如：

```
claude mcp add --transport http sentry https://mcp.sentry.dev/mcp
```

Step 2: 在 Claude Code 中使用 /mcp 命令

在 Claude Code 中，使用命令：

```
/mcp
```

然后按照浏览器中的步骤登录。

Tip:

提示：

- 身份验证令牌安全存储并自动刷新
- 使用 `/mcp` 菜单中的“清除身份验证”撤销访问权限
- 如果您的浏览器没有自动打开，请复制提供的 URL 并手动打开
- 如果浏览器重定向在身份验证后失败并出现连接错误，请将浏览器地址栏中的完整回调 URL 粘贴到 Claude Code 中出现的 URL 提示中
- OAuth 身份验证适用于 HTTP 服务器

使用固定的 OAuth 回调端口

某些 MCP 服务器需要预先注册的特定重定向 URI。默认情况下，Claude Code 为 OAuth 回调选择随机可用端口。使用 `--callback-port` 固定端口，使其与 `http://localhost:PORT/callback` 形式的预注册重定向 URI 匹配。

您可以单独使用 `--callback-port`（使用动态客户端注册）或与 `--client-id` 一起使用（使用预配置的凭据）。

```
## 使用动态客户端注册的固定回调端口
claude mcp add --transport http \
  --callback-port 8080 \
  my-server https://mcp.example.com/mcp
```

使用预配置的 OAuth 凭据

某些 MCP 服务器不支持自动 OAuth 设置。如果您看到类似“不兼容的身份验证服务器：不支持动态客户端注册”的错误，服务器需要预配置的凭据。首先通过服务器的开发者门户注册 OAuth 应用，然后在添加服务器时提供凭据。

Step 1: 使用服务器注册 OAuth 应用

通过服务器的开发者门户创建应用，并记下您的客户端 ID 和客户端密钥。

许多服务器还需要重定向 URI。如果是这样，请选择一个端口并以 `http://localhost:PORT/callback` 的格式注册重定向 URI。在下一步中使用该相同的端口与 `--callback-port`。

Step 2: 使用您的凭据添加服务器

选择以下方法之一。用于 `--callback-port` 的端口可以是任何可用的端口。它只需要与您在上一步中注册的重定向 URI 匹配。

claude mcp add

使用 `--client-id` 传递您的应用的客户端 ID。 `--client-secret` 标志使用掩盖的输入提示输入密钥：

```
claude mcp add --transport http \
  --client-id your-client-id --client-secret --callback-port 8080 \
  my-server https://mcp.example.com/mcp
```

claude mcp add-json

在 JSON 配置中包含 `oauth` 对象，并将 `--client-secret` 作为单独的标志传递：

```
claude mcp add-json my-server \  
  '{"type":"http","url":"https://mcp.example.com/mcp","oauth":{"clientId":"your-  
client-id","callbackPort":8080}}' \  
  --client-secret
```

claude mcp add-json（仅回调端口）

使用 `--callback-port` 而不使用客户端 ID 来固定端口，同时使用动态客户端注册：

```
claude mcp add-json my-server \  
  '{"type":"http","url":"https://mcp.example.com/mcp","oauth":  
{"callbackPort":8080}}'
```

CI / 环境变量

通过环境变量设置密钥以跳过交互式提示：

```
MCP_CLIENT_SECRET=your-secret claude mcp add --transport http \  
  --client-id your-client-id --client-secret --callback-port 8080 \  
  my-server https://mcp.example.com/mcp
```

Step 3: 在 Claude Code 中进行身份验证

在 Claude Code 中运行 `/mcp` 并按照浏览器登录流程。

Tip:

提示：

- 客户端密钥安全地存储在您的系统钥匙链（macOS）或凭据文件中，而不是在您的配置中
- 如果服务器使用没有密钥的公共 OAuth 客户端，仅使用 `--client-id` 而不使用 `--client-secret`
- `--callback-port` 可以与或不与 `--client-id` 一起使用
- 这些标志仅适用于 HTTP 和 SSE 传输。它们对 stdio 服务器没有影响
- 使用 `claude mcp get <name>` 验证为服务器配置了 OAuth 凭据

覆盖 OAuth 元数据发现

如果您的 MCP 服务器在标准 OAuth 元数据端点（`/.well-known/oauth-authorization-server`）上返回错误，但公开了工作的 OIDC 端点，您可以告诉 Claude Code 直接从您指定的 URL 获取 OAuth 元数据，绕过标准发现链。

在您的服务器配置中的 `.mcp.json` 的 `oauth` 对象中设置 `authServerMetadataUrl`：

```
{
  "mcpServers": {
    "my-server": {
      "type": "http",
      "url": "https://mcp.example.com/mcp",
      "oauth": {
        "authServerMetadataUrl": "https://auth.example.com/.well-known/openid-configuration"
      }
    }
  }
}
```

URL 必须使用 `https://`。此选项需要 Claude Code v2.1.64 或更高版本。

从 JSON 配置添加 MCP 服务器

如果您有 MCP 服务器的 JSON 配置，您可以直接添加它：

Step 1: 从 JSON 添加 MCP 服务器

基本语法

```
claude mcp add-json <name> '<json>'
```

示例：添加带有 JSON 配置的 HTTP 服务器

```
claude mcp add-json weather-api '{"type":"http","url":"https://api.weather.com/mcp","headers":{"Authorization":"Bearer token"}}'
```

示例：添加带有 JSON 配置的 stdio 服务器

```
claude mcp add-json local-weather '{"type":"stdio","command":"/path/to/weather-cli","args":["--api-key","abc123"],"env":{"CACHE_DIR":"/tmp"}}'
```

示例：添加带有预配置 OAuth 凭据的 HTTP 服务器

```
claude mcp add-json my-server '{"type":"http","url":"https://mcp.example.com/mcp","oauth":{"clientId":"your-client-id","callbackPort":8080}}' --client-secret
```

Step 2: 验证服务器已添加

```
claude mcp get weather-api
```

Tip:

提示：

- 确保 JSON 在您的 shell 中正确转义
- JSON 必须符合 MCP 服务器配置架构
- 您可以使用 `--scope user` 将服务器添加到您的用户配置而不是项目特定的配置

从 Claude Desktop 导入 MCP 服务器

如果您已在 Claude Desktop 中配置了 MCP 服务器，您可以导入它们：

Step 1: 从 Claude Desktop 导入服务器

基本语法

```
claude mcp add-from-claude-desktop
```

Step 2: 选择要导入的服务器

运行命令后，您将看到一个交互式对话框，允许您选择要导入的服务器。

Step 3: 验证服务器已导入

```
claude mcp list
```

Tip:

提示：

- 此功能仅在 macOS 和 Windows Subsystem for Linux (WSL) 上有效
- 它从这些平台上的标准位置读取 Claude Desktop 配置文件
- 使用 `--scope user` 标志将服务器添加到您的用户配置
- 导入的服务器将具有与 Claude Desktop 中相同的名称
- 如果具有相同名称的服务器已存在，它们将获得数字后缀（例如，`server_1`）

使用来自 Claude.ai 的 MCP 服务器

如果您已使用 [Claude.ai](#) 帐户登录 Claude Code，您在 Claude.ai 中添加的 MCP 服务器会自动在 Claude Code 中可用：

Step 1: 在 Claude.ai 中配置 MCP 服务器

在 [claude.ai/settings/connectors](#) 添加服务器。在 Team 和 Enterprise 计划上，仅管理员可以添加服务器。

Step 2: 对 MCP 服务器进行身份验证

在 Claude.ai 中完成任何必需的身份验证步骤。

Step 3: 在 Claude Code 中查看和管理服务器

在 Claude Code 中，使用命令：

```
/mcp
```

Claude.ai 服务器在列表中出现，并带有指示它们来自 Claude.ai 的指示符。

要在 Claude Code 中禁用 claude.ai MCP 服务器，请将 `ENABLE_CLAUDEAI_MCP_SERVERS` 环境变量设置为 `false`：


```
ENABLE_CLAUDEAI_MCP_SERVERS=false claude
```

将 Claude Code 用作 MCP 服务器

您可以将 Claude Code 本身用作 MCP 服务器，其他应用程序可以连接到它：

```
## 启动 Claude 作为 stdio MCP 服务器
claude mcp serve
```

您可以通过将此配置添加到 `claude_desktop_config.json` 在 Claude Desktop 中使用它：

```
{
  "mcpServers": {
    "claude-code": {
      "type": "stdio",
      "command": "claude",
      "args": ["mcp", "serve"],
      "env": {}
    }
  }
}
```

Warning:

配置可执行文件路径： `command` 字段必须引用 Claude Code 可执行文件。如果 `claude` 命令不在您的系统 PATH 中，您需要指定可执行文件的完整路径。

要查找完整路径：

```
which claude
```

然后在您的配置中使用完整路径：

```
{
  "mcpServers": {
    "claude-code": {
      "type": "stdio",
      "command": "/full/path/to/claude",
      "args": ["mcp", "serve"],
      "env": {}
    }
  }
}
```

没有正确的可执行文件路径，您会遇到类似 `spawn claude ENOENT` 的错误。

Tip:

提示：

- 服务器提供对 Claude 的工具（如 View、Edit、LS 等）的访问权限。
- 在 Claude Desktop 中，尝试要求 Claude 读取目录中的文件、进行编辑等。
- 请注意，此 MCP 服务器仅向您的 MCP 客户端公开 Claude Code 的工具，因此您自己的客户端负责为单个工具调用实现用户确认。

MCP 输出限制和警告

当 MCP 工具产生大量输出时，Claude Code 可帮助管理令牌使用情况，以防止压倒您的对话上下文：

- **输出警告阈值：**当任何 MCP 工具输出超过 10,000 个令牌时，Claude Code 显示警告
- **可配置限制：**您可以使用 `MAX_MCP_OUTPUT_TOKENS` 环境变量调整最大允许的 MCP 输出令牌
- **默认限制：**默认最大值为 25,000 个令牌

要为产生大量输出的工具增加限制：

```
## 为 MCP 工具输出设置更高的限制
export MAX_MCP_OUTPUT_TOKENS=50000
claude
```

这在使用以下 MCP 服务器时特别有用：

- 查询大型数据集或数据库

- 生成详细的报告或文档
- 处理广泛的日志文件或调试信息

Warning:

如果您经常遇到特定 MCP 服务器的输出警告，请考虑增加限制或配置服务器以分页或过滤其响应。

使用 MCP 资源

MCP 服务器可以公开资源，您可以使用 @ 提及来引用，类似于您引用文件的方式。

引用 MCP 资源

Step 1: 列出可用资源

在您的提示中键入 @ 以查看来自所有连接的 MCP 服务器的可用资源。资源与文件一起出现在自动完成菜单中。

Step 2: 引用特定资源

使用格式 `@server:protocol://resource/path` 来引用资源：

```
Can you analyze @github:issue://123 and suggest a fix?
```

```
Please review the API documentation at @docs:file://api/authentication
```

Step 3: 多个资源引用

您可以在单个提示中引用多个资源：

```
Compare @postgres:schema://users with @docs:file://database/user-model
```

Tip:

提示：

- 资源在引用时会自动获取并作为附件包含
- 资源路径在 @ 提及自动完成中可进行模糊搜索
- Claude Code 在服务器支持时自动提供列出和读取 MCP 资源的工具

- 资源可以包含 MCP 服务器提供的任何类型的内容（文本、JSON、结构化数据等）

使用 MCP 工具搜索进行扩展

当您配置了许多 MCP 服务器时，工具定义可能会消耗您的上下文窗口的很大一部分。MCP 工具搜索通过动态按需加载工具而不是预加载所有工具来解决此问题。

工作原理

当您的 MCP 工具描述会消耗超过 10% 的上下文窗口时，Claude Code 会自动启用工具搜索。您可以[调整此阈值](#)或完全禁用工具搜索。触发时：

1. MCP 工具被延迟而不是预先加载到上下文中
2. Claude 使用搜索工具在需要时发现相关的 MCP 工具
3. 仅将 Claude 实际需要的工具加载到上下文中
4. MCP 工具从您的角度继续完全按照之前的方式工作

对于 MCP 服务器作者

如果您正在构建 MCP 服务器，启用工具搜索时服务器说明字段会变得更有用。服务器说明可帮助 Claude 了解何时搜索您的工具，类似于 [skills](#) 的工作方式。

添加清晰、描述性的服务器说明，说明：

- 您的工具处理的任务类别
- Claude 应何时搜索您的工具
- 您的服务器提供的关键功能

配置工具搜索

工具搜索默认以自动模式运行，这意味着仅当您的 MCP 工具定义超过上下文阈值时才激活。如果您的工具很少，它们会正常加载，无需工具搜索。此功能需要支持 `tool_reference` 块的模型：Sonnet 4 及更高版本，或 Opus 4 及更高版本。Haiku 模型不支持工具搜索。

使用 `ENABLE_TOOL_SEARCH` 环境变量控制工具搜索行为：

值	行为
<code>auto</code>	当 MCP 工具超过 10% 的上下文时激活（默认）
<code>auto:<N></code>	在自定义阈值激活，其中 <code><N></code> 是百分比（例如， <code>auto:5</code> 表示 5%）
<code>true</code>	始终启用

值	行为
false	禁用，所有 MCP 工具预先加载

```
## 使用自定义 5% 阈值
ENABLE_TOOL_SEARCH=auto:5 claude

## 完全禁用工具搜索
ENABLE_TOOL_SEARCH=false claude
```

或在您的settings.json env 字段中设置值。

您也可以使用 disallowedTools 设置专门禁用 MCPSearch 工具：

```
{
  "permissions": {
    "deny": ["MCPSearch"]
  }
}
```

将 MCP 提示用作命令

MCP 服务器可以公开在 Claude Code 中作为命令可用的提示。

执行 MCP 提示

Step 1: 发现可用的提示

键入 / 以查看所有可用的命令，包括来自 MCP 服务器的命令。MCP 提示以 / mcp_servername_promptname 的格式出现。

Step 2: 执行不带参数的提示

```
/mcp_github_list_prs
```

Step 3: 执行带参数的提示

许多提示接受参数。在命令后面用空格分隔传递它们：

```
/mcp_github__pr_review 456
```

```
/mcp_jira__create_issue "Bug in login flow" high
```

Tip:

提示:

- MCP 提示从连接的服务器动态发现
- 参数根据提示的定义参数进行解析
- 提示结果直接注入到对话中
- 服务器和提示名称被规范化（空格变为下划线）

托管 MCP 配置

对于需要对 MCP 服务器进行集中控制的组织，Claude Code 支持两个配置选项：

1. 使用 **managed-mcp.json** 的独占控制：部署用户无法修改或扩展的固定 MCP 服务器集
2. 使用允许列表/拒绝列表的基于策略的控制：允许用户添加自己的服务器，但限制允许的服务器

这些选项允许 IT 管理员：

- **控制员工可以访问哪些 MCP 服务器**：在整个组织中部署一组标准化的已批准 MCP 服务器
- **防止未授权的 MCP 服务器**：限制用户添加未批准的 MCP 服务器
- **完全禁用 MCP**：如果需要，完全删除 MCP 功能

选项 1：使用 managed-mcp.json 的独占控制

部署 **managed-mcp.json** 文件时，它对所有 MCP 服务器进行**独占控制**。用户无法添加、修改或使用此文件中定义的任何 MCP 服务器以外的任何 MCP 服务器。这是希望完全控制的组织的最简单方法。

系统管理员将配置文件部署到系统范围的目录：

- macOS：`/Library/Application Support/ClaudeCode/managed-mcp.json`
- Linux 和 WSL：`/etc/claude-code/managed-mcp.json`
- Windows：`C:\Program Files\ClaudeCode\managed-mcp.json`

Note:

这些是系统范围的路径（不是像 `~/Library/...` 这样的用户主目录），需要管理员权限。它们设计为由 IT 管理员部署。

`managed-mcp.json` 文件使用与标准 `.mcp.json` 文件相同的格式：

```
{
  "mcpServers": {
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/"
    },
    "sentry": {
      "type": "http",
      "url": "https://mcp.sentry.dev/mcp"
    },
    "company-internal": {
      "type": "stdio",
      "command": "/usr/local/bin/company-mcp-server",
      "args": ["--config", "/etc/company/mcp-config.json"],
      "env": {
        "COMPANY_API_URL": "https://internal.company.com"
      }
    }
  }
}
```

选项 2：使用允许列表和拒绝列表的基于策略的控制

管理员可以允许用户配置自己的 MCP 服务器，而不是进行独占控制，同时对允许的服务器进行限制。此方法在[托管设置文件](#)中使用 `allowedMcpServers` 和 `deniedMcpServers`。

Note:

在选项之间选择：当您想要部署一组固定的服务器而不进行用户自定义时，使用选项 1（`managed-mcp.json`）。当您想要允许用户在策略约束内添加自己的服务器时，使用选项 2（允许列表/拒绝列表）。

限制选项

允许列表或拒绝列表中的每个条目可以通过三种方式限制服务器：

1. 按服务器名称 (`serverName`)：匹配服务器的配置名称
2. 按命令 (`serverCommand`)：匹配用于启动 stdio 服务器的确切命令和参数
3. 按 URL 模式 (`serverUrl`)：匹配带有通配符支持的远程服务器 URL

重要：每个条目必须恰好具有 `serverName` 、 `serverCommand` 或 `serverUrl` 之一。

示例配置

```
{
  "allowedMcpServers": [
    // 按服务器名称允许
    { "serverName": "github" },
    { "serverName": "sentry" },

    // 按确切命令允许（对于 stdio 服务器）
    { "serverCommand": ["npx", "-y", "@modelcontextprotocol/server-filesystem"] },
    { "serverCommand": ["python", "/usr/local/bin/approved-server.py"] },

    // 按 URL 模式允许（对于远程服务器）
    { "serverUrl": "https://mcp.company.com/*" },
    { "serverUrl": "https://*.internal.corp/*" }
  ],
  "deniedMcpServers": [
    // 按服务器名称阻止
    { "serverName": "dangerous-server" },

    // 按确切命令阻止（对于 stdio 服务器）
    { "serverCommand": ["npx", "-y", "unapproved-package"] },

    // 按 URL 模式阻止（对于远程服务器）
    { "serverUrl": "https://*.untrusted.com/*" }
  ]
}
```


基于命令的限制如何工作

精确匹配：

- 命令数组必须**精确**匹配 - 命令和所有参数的顺序正确
- 示例：`["npx", "-y", "server"]` 将**不**匹配 `["npx", "server"]` 或 `["npx", "-y", "server", "--flag"]`

Stdio 服务器行为：

- 当允许列表包含**任何** `serverCommand` 条目时，stdio 服务器**必须**匹配其中一个命令
- Stdio 服务器在存在命令限制时无法仅按名称通过
- 这确保管理员可以强制执行允许运行哪些命令

非 stdio 服务器行为：

- 远程服务器（HTTP、SSE、WebSocket）在允许列表中存在 `serverUrl` 条目时使用基于 URL 的匹配
- 如果不存在 URL 条目，远程服务器回退到基于名称的匹配
- 命令限制不适用于远程服务器

基于 URL 的限制如何工作

URL 模式使用 `*` 支持通配符以匹配任何字符序列。这对于允许整个域或子域很有用。

通配符示例：

- `https://mcp.company.com/*` - 允许特定域上的所有路径
- `https://*.example.com/*` - 允许 example.com 的任何子域
- `http://localhost:*/*` - 允许 localhost 上的任何端口

远程服务器行为：

- 当允许列表包含**任何** `serverUrl` 条目时，远程服务器**必须**匹配其中一个 URL 模式
- 远程服务器在存在 URL 限制时无法仅按名称通过
- 这确保管理员可以强制执行允许哪些远程端点

```
{
  "allowedMcpServers": [
    { "serverUrl": "https://mcp.company.com/*" },
    { "serverUrl": "https://*.internal.corp/*" }
  ]
}
```

结果：

- `https://mcp.company.com/api` 处的 HTTP 服务器：✅ 允许（匹配 URL 模式）
- `https://api.internal.corp/mcp` 处的 HTTP 服务器：✅ 允许（匹配通配符子域）
- `https://external.com/mcp` 处的 HTTP 服务器：❌ 阻止（不匹配任何 URL 模式）
- 任何命令的 Stdio 服务器：❌ 阻止（没有名称或命令条目可匹配）

```
{
  "allowedMcpServers": [
    { "serverCommand": ["npx", "-y", "approved-package"] }
  ]
}
```

结果：

- 带有 `["npx", "-y", "approved-package"]` 的 Stdio 服务器：✅ 允许（匹配命令）
- 带有 `["node", "server.js"]` 的 Stdio 服务器：❌ 阻止（不匹配命令）
- 名为 `my-api` 的 HTTP 服务器：❌ 阻止（没有名称条目可匹配）

```
{
  "allowedMcpServers": [
    { "serverName": "github" },
    { "serverCommand": ["npx", "-y", "approved-package"] }
  ]
}
```

结果：

- 名为 `local-tool`、带有 `["npx", "-y", "approved-package"]` 的 Stdio 服务器：✅ 允许（匹配命令）

- 名为“local-tool”、带有 `["node", "server.js"]` 的 Stdio 服务器：❌ 阻止（命令条目存在但不匹配）
- 名为“github”、带有 `["node", "server.js"]` 的 Stdio 服务器：❌ 阻止（当命令条目存在时，stdio 服务器必须匹配命令）
- 名为“github”的 HTTP 服务器：✅ 允许（匹配名称）
- 名为“other-api”的 HTTP 服务器：❌ 阻止（名称不匹配）

```
{
  "allowedMcpServers": [
    { "serverName": "github" },
    { "serverName": "internal-tool" }
  ]
}
```

结果：

- 名为“github”、任何命令的 Stdio 服务器：✅ 允许（没有命令限制）
- 名为“internal-tool”、任何命令的 Stdio 服务器：✅ 允许（没有命令限制）
- 名为“github”的 HTTP 服务器：✅ 允许（匹配名称）
- 任何名为“other”的服务器：❌ 阻止（名称不匹配）

允许列表行为 (`allowedMcpServers`)

- `undefined`（默认）：无限制 - 用户可以配置任何 MCP 服务器
- 空数组 `[]`：完全锁定 - 用户无法配置任何 MCP 服务器
- 条目列表：用户只能配置按名称、命令或 URL 模式匹配的服务器

拒绝列表行为 (`deniedMcpServers`)

- `undefined`（默认）：没有服务器被阻止
- 空数组 `[]`：没有服务器被阻止
- 条目列表：指定的服务器在所有范围内被显式阻止

重要说明

- **选项 1 和选项 2 可以组合**：如果 `managed-mcp.json` 存在，它具有独占控制，用户无法添加服务器。允许列表/拒绝列表仍然适用于托管服务器本身。
- **拒绝列表具有绝对优先级**：如果服务器匹配拒绝列表条目（按名称、命令或 URL），即使它在允许列表上，它也会被阻止

- 基于名称、基于命令和基于 URL 的限制一起工作：如果服务器匹配**任何**名称条目、命令条目或 URL 模式，它就会通过（除非被拒绝列表阻止）

Note:

使用 `managed-mcp.json` 时：用户无法通过 `claude mcp add` 或配置文件添加 MCP 服务器。`allowedMcpServers` 和 `deniedMcpServers` 设置仍然适用于过滤实际加载的托管服务器。

使用 skills 扩展 Claude

创建、管理和共享 skills 以在 Claude Code 中扩展 Claude 的功能。包括自定义命令和捆绑 skills。

Skills 扩展了 Claude 能做的事情。创建一个 `SKILL.md` 文件，其中包含说明，Claude 会将其添加到其工具包中。Claude 在相关时使用 skills，或者你可以使用 `/skill-name` 直接调用一个。

Note:

对于内置命令（如 `/help` 和 `/compact`），请参阅[交互模式](#)。

自定义命令已合并到 skills 中。`.claude/commands/deploy.md` 中的文件和 `.claude/skills/deploy/SKILL.md` 中的 skill 都会创建 `/deploy` 并以相同的方式工作。你现有的 `.claude/commands/` 文件继续工作。Skills 添加了可选功能：支持文件的目录、控制你或 Claude 是否调用它们的 frontmatter，以及 Claude 在相关时自动加载它们的能力。

Claude Code skills 遵循 [Agent Skills](#) 开放标准，该标准适用于多个 AI 工具。Claude Code 使用额外功能扩展了该标准，如[调用控制](#)、[subagent 执行](#)和[动态上下文注入](#)。

捆绑 skills

捆绑 skills 随 Claude Code 一起提供，在每个会话中都可用。与[内置命令](#)不同，内置命令直接执行固定逻辑，捆绑 skills 是基于提示的：它们为 Claude 提供详细的操作手册，让它使用其工具来编排工作。这意味着捆绑 skills 可以生成并行代理、读取文件并适应你的代码库。

你调用捆绑 skills 的方式与调用任何其他 skill 相同：输入 `/` 后跟 skill 名称。

- `/simplify`：审查你最近更改的文件以查找代码重用、质量和效率问题，然后修复它们。在实现功能或错误修复后运行它来清理你的工作。它并行生成三个审查代理（代码重用、代码质量、效率），汇总他们的发现，并应用修复。传递可选文本以专注于特定问题：`/simplify focus on memory efficiency`。
- `/batch <instruction>`：在整个代码库中并行编排大规模更改。提供更改的描述，`/batch` 研究代码库，将工作分解为 5 到 30 个独立单元，并提出计划供你批准。批准后，它为每个单元生成一个后台代理，每个都在隔离的 [git worktree](#) 中。每个代理实现其单元、运行测试并打开拉取请求。需要 git 存储库。示例：`/batch migrate src/ from Solid to React`。

- `/debug [description]`：通过读取会话调试日志来排查你当前的 Claude Code 会话。可选地描述问题以专注分析。
- `/loop [interval] <prompt>`：在会话保持打开状态时按间隔重复运行提示。Claude 解析间隔、安排循环 cron 任务并确认节奏。对于轮询部署、监视 PR 或定期重新运行另一个 skill 很有用。示例：`/loop 5m check if the deploy finished`。请参阅[按计划运行提示](#)。
- `/claude-api`：为你的项目语言（Python、TypeScript、Java、Go、Ruby、C#、PHP 或 cURL）加载 Claude API 参考资料，以及 Python 和 TypeScript 的 Agent SDK 参考。涵盖工具使用、流式传输、批处理、结构化输出和常见陷阱。当你的代码导入 `anthropic`、`@anthropic-ai/sdk` 或 `claude_agent_sdk` 时也会自动激活。

入门

创建你的第一个 skill

此示例创建一个 skill，教 Claude 使用视觉图表和类比来解释代码。由于它使用默认 frontmatter，Claude 可以在你询问某事如何工作时自动加载它，或者你可以使用 `/explain-code` 直接调用它。

Step 1: 创建 skill 目录

在你的个人 skills 文件夹中为 skill 创建一个目录。个人 skills 在你的所有项目中都可用。

```
mkdir -p ~/.claude/skills/explain-code
```

Step 2: 编写 SKILL.md

每个 skill 都需要一个 `SKILL.md` 文件，包含两部分：YAML frontmatter（在 `---` 标记之间）告诉 Claude 何时使用该 skill，以及包含 Claude 在调用该 skill 时遵循的说明的 markdown 内容。`name` 字段变成 `/slash-command`，`description` 帮助 Claude 决定何时自动加载它。

创建 `~/.claude/skills/explain-code/SKILL.md`：

```
---
name: explain-code
description: Explains code with visual diagrams and analogies. Use when explaining
how code works, teaching about a codebase, or when the user asks "how does this
work?"
---

When explaining code, always include:

1. **Start with an analogy**: Compare the code to something from everyday life
2. **Draw a diagram**: Use ASCII art to show the flow, structure, or relationships
3. **Walk through the code**: Explain step-by-step what happens
4. **Highlight a gotcha**: What's a common mistake or misconception?

Keep explanations conversational. For complex concepts, use multiple analogies.
```

Step 3: 测试 skill

你可以通过两种方式测试它：

让 Claude 自动调用它，通过询问与描述匹配的内容：

```
How does this code work?
```

或直接使用 skill 名称调用它：

```
/explain-code src/auth/login.ts
```

无论哪种方式，Claude 都应该在其解释中包含类比和 ASCII 图表。

Skills 的位置

你存储 skill 的位置决定了谁可以使用它：

位置	路径	适用于
企业	请参阅 托管设置	你的组织中的所有用户
个人	<code>~/.claude/skills/<skill-name>/SKILL.md</code>	你的所有项目

位置	路径	适用于
项目	<code>.claude/skills/<skill-name>/SKILL.md</code>	仅此项目
插件	<code><plugin>/skills/<skill-name>/SKILL.md</code>	启用插件的位置

当 skills 在各个级别共享相同名称时，更高优先级的位置获胜：企业 > 个人 > 项目。插件 skills 使用 `plugin-name:skill-name` 命名空间，因此它们不能与其他级别冲突。如果你在 `.claude/commands/` 中有文件，它们的工作方式相同，但如果 skill 和命令共享相同名称，skill 优先。

从嵌套目录自动发现

当你在子目录中处理文件时，Claude Code 会自动从嵌套的 `.claude/skills/` 目录中发现 skills。例如，如果你正在编辑 `packages/frontend/` 中的文件，Claude Code 也会在 `packages/frontend/.claude/skills/` 中查找 skills。这支持 monorepo 设置，其中包含自己的 skills。

每个 skill 都是一个以 `SKILL.md` 作为入口点的目录：

```
my-skill/
├─ SKILL.md           # 主要说明（必需）
├─ template.md        # Claude 要填写的模板
├─ examples/
│   └─ sample.md      # 显示预期格式的示例输出
└─ scripts/
    └─ validate.sh     # Claude 可以执行的脚本
```

`SKILL.md` 包含主要说明并且是必需的。其他文件是可选的，让你构建更强大的 skills：Claude 要填写的模板、显示预期格式的示例输出、Claude 可以执行的脚本或详细的参考文档。从你的 `SKILL.md` 引用这些文件，以便 Claude 知道它们包含什么以及何时加载它们。有关更多详细信息，请参阅[添加支持文件](#)。

Note:

`.claude/commands/` 中的文件仍然有效并支持相同的 `frontmatter`。建议使用 Skills，因为它们支持额外功能，如支持文件。

来自其他目录的 skills

在通过 `--add-dir` 添加的目录中的 `.claude/skills/` 中定义的 skills 会自动加载并由实时更改检测拾取，因此你可以在会话期间编辑它们而无需重新启动。

Note:

来自 `--add-dir` 目录的 `CLAUDE.md` 文件默认不加载。要加载它们，请设置 `CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1`。请参阅[从其他目录加载](#)。

配置 skills

Skills 通过 `SKILL.md` 顶部的 YAML frontmatter 和随后的 markdown 内容进行配置。

Skill 内容的类型

Skill 文件可以包含任何说明，但思考你想如何调用它们有助于指导要包含的内容：

参考内容添加 Claude 应用于你当前工作的知识。约定、模式、风格指南、领域知识。此内容内联运行，以便 Claude 可以将其与你的对话上下文一起使用。

```
---
name: api-conventions
description: API design patterns for this codebase
---

When writing API endpoints:
- Use RESTful naming conventions
- Return consistent error formats
- Include request validation
```

任务内容为 Claude 提供特定操作的分步说明，如部署、提交或代码生成。这些通常是你想使用 `/skill-name` 直接调用的操作，而不是让 Claude 决定何时运行它们。添加 `disable-model-invocation: true` 以防止 Claude 自动触发它。

```
---
name: deploy
description: Deploy the application to production
context: fork
disable-model-invocation: true
---

Deploy the application:
1. Run the test suite
2. Build the application
3. Push to the deployment target
```

你的 `SKILL.md` 可以包含任何内容，但思考你想如何调用该 skill（由你、由 Claude 或两者）以及你想在哪里运行它（内联或在 subagent 中）有助于指导要包含的内容。对于复杂的 skills，你也可以[添加支持文件](#)来保持主 skill 的专注。

Frontmatter 参考

除了 markdown 内容外，你可以使用 `SKILL.md` 文件顶部 `---` 标记之间的 YAML frontmatter 字段来配置 skill 行为：

```
---
name: my-skill
description: What this skill does
disable-model-invocation: true
allowed-tools: Read, Grep
---

Your skill instructions here...
```

所有字段都是可选的。建议使用 `description`，以便 Claude 知道何时使用该 skill。

字段	必需	描述
<code>name</code>	否	Skill 的显示名称。如果省略，使用目录名称。仅小写字母、数字和连字符（最多 64 个字符）。

字段	必需	描述
<code>description</code>	推荐	Skill 的功能以及何时使用它。Claude 使用它来决定何时应用该 skill。如果省略，使用 markdown 内容的第一段。
<code>argument-hint</code>	否	自动完成期间显示的提示，指示预期的参数。示例：[<code>issue-number</code>] 或 [<code>filename</code>] [<code>format</code>]。
<code>disable-model-invocation</code>	否	设置为 <code>true</code> 以防止 Claude 自动加载此 skill。用于你想使用 <code>/name</code> 手动触发的工作流。默认值： <code>false</code> 。
<code>user-invocable</code>	否	设置为 <code>false</code> 以从 <code>/</code> 菜单中隐藏。用于用户不应直接调用的背景知识。默认值： <code>true</code> 。
<code>allowed-tools</code>	否	当此 skill 处于活动状态时，Claude 可以使用而无需请求权限的工具。
<code>model</code>	否	当此 skill 处于活动状态时要使用的模型。
<code>context</code>	否	设置为 <code>fork</code> 以在分叉的 subagent 上下文中运行。
<code>agent</code>	否	当设置 <code>context: fork</code> 时要使用的 subagent 类型。
<code>hooks</code>	否	限定于此 skill 生命周期的 hooks。有关配置格式，请参阅 Skills 和代理中的 Hooks 。

可用的字符串替换

Skills 支持 skill 内容中动态值的字符串替换：

变量	描述
<code>\$ARGUMENTS</code>	调用 skill 时传递的所有参数。如果 <code>\$ARGUMENTS</code> 不在内容中，参数作为 <code>ARGUMENTS: <value></code> 追加。
<code>\$ARGUMENTS[N]</code>	按 0 基索引访问特定参数，如 <code>\$ARGUMENTS[0]</code> 表示第一个参数。
<code>\$N</code>	<code>\$ARGUMENTS[N]</code> 的简写，如 <code>\$0</code> 表示第一个参数或 <code>\$1</code> 表示第二个参数。
<code>\$ {CLAUDE_SESSION_ID }</code>	当前会话 ID。用于日志记录、创建会话特定文件或将 skill 输出与会话关联。
<code>\$ {CLAUDE_SKILL_DIR}</code>	包含 skill 的 <code>SKILL.md</code> 文件的目录。对于插件 skills，这是插件内 skill 的子目录，而不是插件根目录。在 bash 注入命令中使用它来引用与 skill 捆绑的脚本或文件，无论当前工作目录如何。

使用替换的示例：

```
---
name: session-logger
description: Log activity for this session
---

Log the following to logs/${CLAUDE_SESSION_ID}.log:

$ARGUMENTS
```

添加支持文件

Skills 可以在其目录中包含多个文件。这使 `SKILL.md` 专注于要点，同时让 Claude 仅在需要时访问详细的参考资料。大型参考文档、API 规范或示例集合不需要在每次 skill 运行时加载到上下文中。

```
my-skill/
├─ SKILL.md (required - overview and navigation)
├─ reference.md (detailed API docs - loaded when needed)
├─ examples.md (usage examples - loaded when needed)
└─ scripts/
    └─ helper.py (utility script - executed, not loaded)
```

从 **SKILL.md** 引用支持文件，以便 Claude 知道每个文件包含什么以及何时加载它：

```
### Additional resources

- For complete API details, see [reference.md](reference.md)
- For usage examples, see [examples.md](examples.md)
```

Tip:

将 **SKILL.md** 保持在 500 行以下。将详细的参考资料移到单独的文件中。

控制谁调用 skill

默认情况下，你和 Claude 都可以调用任何 skill。你可以输入 `/skill-name` 直接调用它，Claude 可以在与你的对话相关时自动加载它。两个 frontmatter 字段让你限制这一点：

- **disable-model-invocation: true**：只有你可以调用该 skill。用于有副作用或你想控制时间的工作流，如 `/commit`、`/deploy` 或 `/send-slack-message`。你不希望 Claude 因为你的代码看起来准备好了就决定部署。
- **user-invocable: false**：只有 Claude 可以调用该 skill。用于不可作为命令操作的背景知识。`legacy-system-context` skill 解释了旧系统的工作原理。Claude 在相关时应该知道这一点，但 `/legacy-system-context` 对用户来说不是一个有意义的操作。

此示例创建一个只有你可以触发的部署 skill。 **disable-model-invocation: true** 字段防止 Claude 自动运行它：

```
---
name: deploy
description: Deploy the application to production
disable-model-invocation: true
---

Deploy $ARGUMENTS to production:

1. Run the test suite
2. Build the application
3. Push to the deployment target
4. Verify the deployment succeeded
```

以下是两个字段如何影响调用和上下文加载：

Frontmatter	你可以调用	Claude 可以调用	何时加载到上下文
(默认)	是	是	描述始终在上下文中，调用时加载完整 skill
<code>disable-model-invocation: true</code>	是	否	描述不在上下文中，你调用时加载完整 skill
<code>user-invocable: false</code>	否	是	描述始终在上下文中，调用时加载完整 skill

Note:

在常规会话中，skill 描述被加载到上下文中，以便 Claude 知道什么可用，但完整 skill 内容仅在调用时加载。[具有预加载 skills 的 Subagents](#) 的工作方式不同：完整 skill 内容在启动时注入。

限制工具访问

使用 `allowed-tools` 字段来限制当 skill 处于活动状态时 Claude 可以使用哪些工具。此 skill 创建一个只读模式，其中 Claude 可以浏览文件但不能修改它们：

```

---
name: safe-reader
description: Read files without making changes
allowed-tools: Read, Grep, Glob
---
```

将参数传递给 skills

你和 Claude 都可以在调用 skill 时传递参数。参数可通过 `$ARGUMENTS` 占位符获得。

此 skill 按编号修复 GitHub 问题。`$ARGUMENTS` 占位符被替换为 skill 名称后面的任何内容：

```

---
name: fix-issue
description: Fix a GitHub issue
disable-model-invocation: true
---
```

Fix GitHub issue \$ARGUMENTS following our coding standards.

1. Read the issue description
2. Understand the requirements
3. Implement the fix
4. Write tests
5. Create a commit

当你运行 `/fix-issue 123` 时，Claude 收到” Fix GitHub issue 123 following our coding standards...”

如果你使用参数调用 skill 但 skill 不包含 `$ARGUMENTS`，Claude Code 会将 `ARGUMENTS: <your input>` 追加到 skill 内容的末尾，以便 Claude 仍然看到你输入的内容。

要按位置访问单个参数，使用 `$ARGUMENTS[N]` 或较短的 `$N`：

```
---  
name: migrate-component  
description: Migrate a component from one framework to another  
---  
  
Migrate the $ARGUMENTS[0] component from $ARGUMENTS[1] to $ARGUMENTS[2].  
Preserve all existing behavior and tests.
```

运行 `/migrate-component searchBar React Vue` 将 `$ARGUMENTS[0]` 替换为 `SearchBar`，`$ARGUMENTS[1]` 替换为 `React`，`$ARGUMENTS[2]` 替换为 `Vue`。使用 `$N` 简写的相同 skill：

```
---  
name: migrate-component  
description: Migrate a component from one framework to another  
---  
  
Migrate the $0 component from $1 to $2.  
Preserve all existing behavior and tests.
```

高级模式

注入动态上下文

`! command` `` 语法在将 skill 内容发送给 Claude 之前运行 shell 命令。命令输出替换占位符，所以 Claude 接收实际数据，而不是命令本身。

此 skill 通过使用 GitHub CLI 获取实时 PR 数据来总结拉取请求。`! gh pr diff` `` 和其他命令首先运行，它们的输出被插入到提示中：


```
---
name: pr-summary
description: Summarize changes in a pull request
context: fork
agent: Explore
allowed-tools: Bash(gh *)
---

### Pull request context
- PR diff: !`gh pr diff`
- PR comments: !`gh pr view --comments`
- Changed files: !`gh pr diff --name-only`

### Your task
Summarize this pull request...
```

当此 skill 运行时：

1. 每个 `! command`` 立即执行（在 Claude 看到任何东西之前）
2. 输出替换 skill 内容中的占位符
3. Claude 接收带有实际 PR 数据的完全呈现的提示

这是预处理，不是 Claude 执行的东西。Claude 只看到最终结果。

Tip:

要在 skill 中启用[扩展思考](#)，在你的 skill 内容中的任何地方包含单词“ultrathink”。

在 subagent 中运行 skills

当你想让 skill 在隔离中运行时，在你的 frontmatter 中添加 `context: fork`。skill 内容变成驱动 subagent 的提示。它将无法访问你的对话历史。

Warning:

`context: fork` 仅对具有明确说明的 skills 有意义。如果你的 skill 包含“使用这些 API 约定”之类的指南而没有任务，subagent 接收指南但没有可操作的提示，并返回而没有有意义的输出。

Skills 和 [subagents](#) 以两个方向协同工作：

方法	系统提示	任务	也加载
带有 <code>context: fork</code> 的 Skill	来自代理类型（ <code>Explore</code> 、 <code>Plan</code> 等）	SKILL.md 内容	CLAUDE.md
带有 <code>skills</code> 字段的 Subagent	Subagent 的 markdown 正文	Claude 的委派消息	预加载的 skills + CLAUDE.md

使用 `context: fork`，你在你的 skill 中编写任务并选择一个代理类型来执行它。对于反向（定义使用 skills 作为参考资料的自定义 subagent），请参阅 [Subagents](#)。

示例：使用 `Explore` 代理的研究 skill

此 skill 在分叉的 `Explore` 代理中运行研究。skill 内容变成任务，代理提供针对代码库探索优化的只读工具：

```
---
name: deep-research
description: Research a topic thoroughly
context: fork
agent: Explore
---

Research $ARGUMENTS thoroughly:

1. Find relevant files using Glob and Grep
2. Read and analyze the code
3. Summarize findings with specific file references
```

当此 skill 运行时：

- 1. 创建新的隔离上下文
- 2. Subagent 接收 skill 内容作为其提示（“Research \$ARGUMENTS thoroughly...”）
- 3. `agent` 字段确定执行环境（模型、工具和权限）
- 4. 结果被总结并返回到你的主对话

`agent` 字段指定使用哪个 subagent 配置。选项包括内置代理（`Explore`、`Plan`、`general-purpose`）或来自 `.claude/agents/` 的任何自定义 subagent。如果省略，使用 `general-purpose`。

限制 Claude 的 skill 访问

默认情况下，Claude 可以调用任何没有设置 `disable-model-invocation: true` 的 skill。定义 `allowed-tools` 的 Skills 在 skill 处于活动状态时授予 Claude 对这些工具的访问权限，无需每次使用批准。你的[权限设置](#)仍然管理所有其他工具的基线批准行为。内置命令如 `/compact` 和 `/init` 不能通过 Skill 工具获得。

三种方式来控制 Claude 可以调用哪些 skills：

通过在 `/permissions` 中拒绝 Skill 工具来禁用所有 skills：

```
## Add to deny rules:
Skill
```

使用[权限规则](#)允许或拒绝特定 skills：

```
## Allow only specific skills
Skill(commit)
Skill(review-pr *)

## Deny specific skills
Skill(deploy *)
```

权限语法：`Skill(name)` 用于精确匹配，`Skill(name *)` 用于带有任何参数的前缀匹配。

通过在其 frontmatter 中添加 `disable-model-invocation: true` 来隐藏单个 skills。这将 skill 从 Claude 的上下文中完全删除。

Note:

`user-invocable` 字段仅控制菜单可见性，不控制 Skill 工具访问。使用 `disable-model-invocation: true` 来阻止程序调用。

共享 skills

Skills 可以根据你的受众在不同范围内分发：

- **项目 skills**：将 `.claude/skills/` 提交到版本控制
- **插件**：在你的[插件](#)中创建 `skills/` 目录
- **托管**：通过[托管设置](#)部署组织范围内

生成视觉输出

Skills 可以捆绑并运行任何语言的脚本，为 Claude 提供超越单个提示可能的功能。一个强大的模式是生成视觉输出：在浏览器中打开的交互式 HTML 文件，用于探索数据、调试或创建报告。

此示例创建一个代码库浏览器：一个交互式树视图，你可以在其中展开和折叠目录、一目了然地查看文件大小，并按颜色识别文件类型。

创建 Skill 目录：

```
mkdir -p ~/.claude/skills/codebase-visualizer/scripts
```

创建 `~/.claude/skills/codebase-visualizer/SKILL.md`。描述告诉 Claude 何时激活此 Skill，说明告诉 Claude 运行捆绑脚本：

```

---
name: codebase-visualizer
description: Generate an interactive collapsible tree visualization of your
codebase. Use when exploring a new repo, understanding project structure, or
identifying large files.
allowed-tools: Bash(python *)
---

## Codebase Visualizer

Generate an interactive HTML tree view that shows your project's file structure
with collapsible directories.

### Usage

Run the visualization script from your project root:

```bash
python ~/.claude/skills/codebase-visualizer/scripts/visualize.py .
```text

This creates `codebase-map.html` in the current directory and opens it in your
default browser.

### What the visualization shows

- **Collapsible directories**: Click folders to expand/collapse
- **File sizes**: Displayed next to each file
- **Colors**: Different colors for different file types
- **Directory totals**: Shows aggregate size of each folder

```

创建 `~/.claude/skills/codebase-visualizer/scripts/visualize.py`。此脚本扫描目录树并生成一个自包含的 HTML 文件，包含：

- 一个**摘要侧边栏**，显示文件计数、目录计数、总大小和文件类型数量
- 一个**条形图**，按文件类型（按大小排名前 8）分解代码库
- 一个**可折叠树**，你可以在其中展开和折叠目录，带有颜色编码的文件类型指示器

该脚本需要 Python，但仅使用内置库，因此无需安装包：

```
#!/usr/bin/env python3

"""Generate an interactive collapsible tree visualization of a codebase."""

from pathlib import Path
from collections import Counter

IGNORE = {''.git', 'node_modules', '__pycache__', '.venv', 'venv', 'dist', 'build'}

def scan(path: Path, stats: dict) → dict:
    result = {"name": path.name, "children": [], "size": 0}
    try:
        for item in sorted(path.iterdir()):
            if item.name in IGNORE or item.name.startswith('.'):
                continue
            if item.is_file():
                size = item.stat().st_size
                ext = item.suffix.lower() or '(no ext)'
                result["children"].append({"name": item.name, "size": size,
"ext": ext})

                result["size"] += size
                stats["files"] += 1
                stats["extensions"][ext] += 1
                stats["ext_sizes"][ext] += size
            elif item.is_dir():
                stats["dirs"] += 1
                child = scan(item, stats)
                if child["children"]:
                    result["children"].append(child)
                    result["size"] += child["size"]
    except PermissionError:
        pass
    return result

def generate_html(data: dict, stats: dict, output: Path) → None:
    ext_sizes = stats["ext_sizes"]
    total_size = sum(ext_sizes.values()) or 1
    sorted_exts = sorted(ext_sizes.items(), key=lambda x: -x[1])[:8]
    colors = {
```

```

        '.js': '#f7df1e', '.ts': '#3178c6', '.py': '#3776ab', '.go': '#00add8',
        '.rs': '#dea584', '.rb': '#cc342d', '.css': '#264de4', '.html': '#e34c26',
        '.json': '#6b7280', '.md': '#083fa1', '.yaml': '#cb171e', '.yml': '#cb171e
    ',

        '.mdx': '#083fa1', '.tsx': '#3178c6', '.jsx': '#61dafb', '.sh': '#4eaa25',
    }

    lang_bar = "".join(
        f'<div class="bar-row"><span class="bar-label">{ext}</span>'
        f'<div class="bar" style="width:{{(size/total_size)*100}}%;background:{{color
s.get(ext,"#6b7280"}}"></div>'
        f'<span class="bar-pct">{{(size/total_size)*100:.1f}}%</span></div>'
        for ext, size in sorted_exts
    )

    def fmt(b):
        if b < 1024: return f"{b} B"
        if b < 1048576: return f"{b/1024:.1f} KB"
        return f"{b/1048576:.1f} MB"

    html = f'''<!DOCTYPE html>
<html><head>
<meta charset="utf-8"><title>Codebase Explorer</title>
<style>
    body {{ font: 14px/1.5 system-ui, sans-serif; margin: 0; background: #1a1a2e;
color: #eee; }}
    .container {{ display: flex; height: 100vh; }}
    .sidebar {{ width: 280px; background: #252542; padding: 20px; border-right:
1px solid #3d3d5c; overflow-y: auto; flex-shrink: 0; }}
    .main {{ flex: 1; padding: 20px; overflow-y: auto; }}
    h1 {{ margin: 0 0 10px 0; font-size: 18px; }}
    h2 {{ margin: 20px 0 10px 0; font-size: 14px; color: #888; text-transform:
uppercase; }}
    .stat {{ display: flex; justify-content: space-between; padding: 8px 0;
border-bottom: 1px solid #3d3d5c; }}
    .stat-value {{ font-weight: bold; }}
    .bar-row {{ display: flex; align-items: center; margin: 6px 0; }}
    .bar-label {{ width: 55px; font-size: 12px; color: #aaa; }}
    .bar {{ height: 18px; border-radius: 3px; }}
    .bar-pct {{ margin-left: 8px; font-size: 12px; color: #666; }}
    .tree {{ list-style: none; padding-left: 20px; }}

```

```

    details {{ cursor: pointer; }}
    summary {{ padding: 4px 8px; border-radius: 4px; }}
    summary:hover {{ background: #2d2d44; }}
    .folder {{ color: #ffd700; }}
    .file {{ display: flex; align-items: center; padding: 4px 8px; border-radius:
4px; }}
    .file:hover {{ background: #2d2d44; }}
    .size {{ color: #888; margin-left: auto; font-size: 12px; }}
    .dot {{ width: 8px; height: 8px; border-radius: 50%; margin-right: 8px; }}
</style>
</head><body>
  <div class="container">
    <div class="sidebar">
      <h1> Summary</h1>
      <div class="stat"><span>Files</span><span class="stat-
value">{stats["files"]},</span></div>
      <div class="stat"><span>Directories</span><span class="stat-value">{stats["d
irs"]},</span></div>
      <div class="stat"><span>Total size</span><span class="stat-
value">{fmt(data["size"])}</span></div>
      <div class="stat"><span>File types</span><span class="stat-value">{len(stats
["extensions"])}</span></div>
      <h2>By file type</h2>
      {lang_bars}
    </div>
    <div class="main">
      <h1> {data["name"]}</h1>
      <ul class="tree" id="root"></ul>
    </div>
  </div>
<script>
  const data = {json.dumps(data)};
  const colors = {json.dumps(colors)};
  function fmt(b) {{ if (b < 1024) return b + ' B'; if (b < 1048576) return (b/
1024).toFixed(1) + ' KB'; return (b/1048576).toFixed(1) + ' MB'; }}
  function render(node, parent) {{
    if (node.children) {{
      const det = document.createElement('details');
      det.open = parent === document.getElementById('root');

```



```

        det.innerHTML = `<summary><span class="folder">📁 ${node.name}</span><span class="size">${fmt(node.size)}</span></summary>`;
        const ul = document.createElement('ul'); ul.className = 'tree';
        node.children.sort((a,b) => (b.children?1:0)-(a.children?1:0) || a.name.localeCompare(b.name));
        node.children.forEach(c => render(c, ul));
        det.appendChild(ul);
        const li = document.createElement('li'); li.appendChild(det);
        parent.appendChild(li);
    } else {
        const li = document.createElement('li'); li.className = 'file';
        li.innerHTML = `<span class="dot" style="background:${colors[node.ext]}|'| #6b7280'></span>${node.name}<span class="size">${fmt(node.size)}</span>`;
        parent.appendChild(li);
    }
}

data.children.forEach(c => render(c, document.getElementById('root')));
</script>
</body></html>'''
    output.write_text(html)

if __name__ == '__main__':
    target = Path(sys.argv[1] if len(sys.argv) > 1 else '.').resolve()
    stats = {"files": 0, "dirs": 0, "extensions": Counter(), "ext_sizes": Counter()}
    data = scan(target, stats)
    out = Path('codebase-map.html')
    generate_html(data, stats, out)
    print(f'Generated {out.absolute()}')
    webbrowser.open(f'file://{out.absolute()}')

```

要测试，在任何项目中打开 Claude Code 并询问” Visualize this codebase.” Claude 运行脚本，生成 `codebase-map.html`，并在浏览器中打开它。

这个模式适用于任何视觉输出：依赖关系图、测试覆盖率报告、API 文档或数据库架构可视化。捆绑脚本做繁重的工作，而 Claude 处理编排。

故障排除

Skill 未触发

如果 Claude 在预期时不使用你的 skill：

1. 检查描述是否包含用户会自然说的关键字
2. 验证 skill 是否出现在 `What skills are available?` 中
3. 尝试重新表述你的请求以更接近描述
4. 如果 skill 是用户可调用的，使用 `/skill-name` 直接调用它

Skill 触发过于频繁

如果 Claude 在你不想要时使用你的 skill：

1. 使描述更具体
2. 如果你只想手动调用，添加 `disable-model-invocation: true`

Claude 看不到我的所有 skills

Skill 描述被加载到上下文中，以便 Claude 知道什么可用。如果你有许多 skills，它们可能超过字符预算。预算在上下文窗口的 2% 处动态扩展，回退为 16,000 个字符。运行 `/context` 来检查有关排除的 skills 的警告。

要覆盖限制，设置 `SLASH_COMMAND_TOOL_CHAR_BUDGET` 环境变量。

相关资源

- [Subagents](#)：将任务委派给专门的代理
- [Plugins](#)：打包和分发 skills 与其他扩展
- [Hooks](#)：围绕工具事件自动化工作流
- [Memory](#)：管理 CLAUDE.md 文件以获得持久上下文
- [Interactive mode](#)：内置命令和快捷键
- [Permissions](#)：控制工具和 skill 访问

创建插件

创建自定义插件以使用 skills、agents、hooks 和 MCP servers 扩展 Claude Code。

Plugins 让你能够使用自定义功能扩展 Claude Code，这些功能可以在项目和团队中共享。本指南涵盖如何使用 skills、agents、hooks 和 MCP servers 创建自己的插件。

想要安装现有插件？请参阅[发现和安装插件](#)。有关完整的技术规范，请参阅[插件参考](#)。

何时使用插件与独立配置

Claude Code 支持两种方式来添加自定义 skills、agents 和 hooks：

方法	Skill 名称	最适合
独立（ <code>.claude/</code> 目录）	<code>/hello</code>	个人工作流、项目特定的自定义、快速实验
插件（包含 <code>.claude-plugin/plugin.json</code> 的目录）	<code>/plugin-name:hello</code>	与团队成员共享、分发到社区、版本化发布、跨项目重用

在以下情况下使用独立配置：

- 你正在为单个项目自定义 Claude Code
- 配置是个人的，不需要共享
- 你在打包 skills 或 hooks 之前进行实验
- 你想要简短的 skill 名称，如 `/hello` 或 `/deploy`

在以下情况下使用插件：

- 你想与团队或社区共享功能
- 你需要在多个项目中使用相同的 skills/agents
- 你想要版本控制和轻松更新扩展
- 你通过市场分发
- 你可以接受命名空间化的 skills，如 `/my-plugin:hello`（命名空间可防止插件之间的冲突）

Tip:

从 `.claude/` 中的独立配置开始进行快速迭代，然后在准备好共享时[转换为插件](#)。

快速开始

本快速开始将引导你创建一个带有自定义 skill 的插件。你将创建一个清单（定义插件的配置文件）、添加一个 skill，并使用 `--plugin-dir` 标志在本地测试它。

前置条件

- Claude Code [已安装并已认证](#)
- Claude Code 版本 1.0.33 或更高版本（运行 `claude --version` 检查）

Note:

如果你没有看到 `/plugin` 命令，请将 Claude Code 更新到最新版本。有关升级说明，请参阅[故障排除](#)。

创建你的第一个插件

Step 1: 创建插件目录

每个插件都位于其自己的目录中，包含清单和你的 skills、agents 或 hooks。现在创建一个：

```
mkdir my-first-plugin
```

Step 2: 创建插件清单

位于 `.claude-plugin/plugin.json` 的清单文件定义了你的插件的身份：其名称、描述和版本。Claude Code 使用此元数据在插件管理器中显示你的插件。

在你的插件文件夹内创建 `.claude-plugin` 目录：

```
mkdir my-first-plugin/.claude-plugin
```

然后使用以下内容创建 `my-first-plugin/.claude-plugin/plugin.json`：

```
{
  "name": "my-first-plugin",
  "description": "A greeting plugin to learn the basics",
  "version": "1.0.0",
  "author": {
    "name": "Your Name"
  }
}
```

字段	目的
name	唯一标识符和 skill 命名空间。Skills 以此为前缀（例如 <code>/my-first-plugin:hello</code> ）。
description	在浏览或安装插件时在插件管理器中显示。
version	使用 语义版本控制 跟踪发布。
author	可选。有助于归属。

有关 `homepage`、`repository` 和 `license` 等其他字段，请参阅[完整清单架构](#)。

Step 3: 添加 skill

Skills 位于 `skills/` 目录中。每个 skill 是一个包含 `SKILL.md` 文件的文件夹。文件夹名称成为 skill 名称，以插件的命名空间为前缀（在名为 `my-first-plugin` 的插件中的 `hello/` 创建 `/my-first-plugin:hello`）。

在你的插件文件夹中创建一个 skill 目录：

```
mkdir -p my-first-plugin/skills/hello
```

然后使用以下内容创建 `my-first-plugin/skills/hello/SKILL.md`：

```
---  
description: Greet the user with a friendly message  
disable-model-invocation: true  
---  
  
Greet the user warmly and ask how you can help them today.
```

Step 4: 测试你的插件

使用 `--plugin-dir` 标志运行 Claude Code 以加载你的插件：

```
claude --plugin-dir ./my-first-plugin
```

Claude Code 启动后，尝试你的新 skill：

```
/my-first-plugin:hello
```

你将看到 Claude 用问候语回应。运行 `/help` 以查看你的 skill 在插件命名空间下列出。

Note:

为什么要命名空间？ 插件 skills 总是命名空间化的（如 `/greet:hello`），以防止多个插件具有相同名称的 skills 时发生冲突。

要更改命名空间前缀，请更新 `plugin.json` 中的 `name` 字段。

Step 5: 添加 skill 参数

通过接受用户输入使你的 skill 动态化。 `$ARGUMENTS` 占位符捕获用户在 skill 名称后提供的任何文本。

更新你的 `SKILL.md` 文件：

```
---  
description: Greet the user with a personalized message  
---  
  
## Hello Skill  
  
Greet the user named "$ARGUMENTS" warmly and ask how you can help them today. Make  
the greeting personal and encouraging.
```

运行 `/reload-plugins` 以获取更改，然后尝试使用你的名字的 skill：

```
/my-first-plugin:hello Alex
```

Claude 将按名字问候你。有关向 skills 传递参数的更多信息，请参阅 [Skills](#)。

你已成功创建并测试了一个包含以下关键组件的插件：

- **插件清单**（`.claude-plugin/plugin.json`）：描述你的插件的元数据
- **Skills 目录**（`skills/`）：包含你的自定义 skills
- **Skill 参数**（`$ARGUMENTS`）：捕获用户输入以实现动态行为

Tip:

`--plugin-dir` 标志对开发和测试很有用。当你准备好与他人共享你的插件时，请参阅[创建和分发插件市场](#)。

插件结构概览

你已创建了一个带有 skill 的插件，但插件可以包含更多内容：自定义 agents、hooks、MCP servers 和 LSP servers。

Warning:

常见错误：不要将 `commands/`、`agents/`、`skills/` 或 `hooks/` 放在 `.claude-plugin/` 目录内。只有 `plugin.json` 应该在 `.claude-plugin/` 内。所有其他目录必须在插件根级别。

目录	位置	目的
<code>.claude-plugin/</code>	插件根	包含 <code>plugin.json</code> 清单 (如果组件使用默认位置， 则可选)
<code>commands/</code>	插件根	作为 Markdown 文件的 Skills
<code>agents/</code>	插件根	自定义 agent 定义
<code>skills/</code>	插件根	带有 <code>SKILL.md</code> 文件的 Agent Skills
<code>hooks/</code>	插件根	<code>hooks.json</code> 中的事件处 理程序
<code>.mcp.json</code>	插件根	MCP server 配置
<code>.lsp.json</code>	插件根	用于代码智能的 LSP server 配置
<code>settings.json</code>	插件根	启用插件时应用的默认 设置

Note:

后续步骤：准备好添加更多功能了吗？跳转到[开发更复杂的插件](#)以添加 agents、hooks、MCP servers 和 LSP servers。有关所有插件组件的完整技术规范，请参阅[插件参考](#)。

开发更复杂的插件

一旦你对基本插件感到满意，你可以创建更复杂的扩展。

向你的插件添加 Skills

插件可以包含 [Agent Skills](#) 以扩展 Claude 的功能。Skills 是模型调用的：Claude 根据任务上下文自动使用它们。

在你的插件根目录添加一个 `skills/` 目录，其中包含包含 `SKILL.md` 文件的 Skill 文件夹：


```
my-plugin/  
├── .claude-plugin/  
│   └── plugin.json  
└── skills/  
    └── code-review/  
        └── SKILL.md
```

每个 `SKILL.md` 需要包含 `name` 和 `description` 字段的 frontmatter，后跟说明：

```
---  
name: code-review  
description: Reviews code for best practices and potential issues. Use when  
reviewing code, checking PRs, or analyzing code quality.  
---  
  
When reviewing code, check for:  
1. Code organization and structure  
2. Error handling  
3. Security concerns  
4. Test coverage
```

安装插件后，运行 `/reload-plugins` 以加载 Skills。有关完整的 Skill 编写指南，包括渐进式披露和工具限制，请参阅 [Agent Skills](#)。

向你的插件添加 LSP servers

Tip:

对于 TypeScript、Python 和 Rust 等常见语言，请从官方市场安装预构建的 LSP 插件。仅当你需要支持尚未涵盖的语言时，才创建自定义 LSP 插件。

LSP (Language Server Protocol) 插件为 Claude 提供实时代码智能。如果你需要支持没有官方 LSP 插件的语言，你可以通过向你的插件添加 `.lsp.json` 文件来创建自己的：

```
{
  "go": {
    "command": "gopls",
    "args": ["serve"],
    "extensionToLanguage": {
      ".go": "go"
    }
  }
}
```

安装你的插件的用户必须在其机器上安装语言服务器二进制文件。

有关完整的 LSP 配置选项，请参阅 [LSP servers](#)。

使用你的插件提供默认设置

插件可以在插件根目录包含一个 `settings.json` 文件，以在启用插件时应用默认配置。目前仅支持 `agent` 键。

设置 `agent` 激活插件的[自定义 agents](#)之一作为主线程，应用其系统提示、工具限制和模型。这让插件在启用时通过改变 Claude Code 的默认行为方式。

```
{
  "agent": "security-reviewer"
}
```

此示例激活在插件的 `agents/` 目录中定义的 `security-reviewer` agent。来自 `settings.json` 的设置优先于在 `plugin.json` 中声明的 `settings`。未知键被静默忽略。

组织复杂的插件

对于具有许多组件的插件，按功能组织你的目录结构。有关完整的目录布局和组织模式，请参阅[插件目录结构](#)。

在本地测试你的插件

使用 `--plugin-dir` 标志在开发期间测试插件。这会直接加载你的插件，无需安装。

```
claude --plugin-dir ./my-plugin
```

当你对插件进行更改时，运行 `/reload-plugins` 以获取更新，无需重新启动。对 LSP server 配置的更改仍需要完全重新启动。测试你的插件组件：

- 使用 `/plugin-name:skill-name` 尝试你的 skills
- 检查 agents 是否出现在 `/agents` 中
- 验证 hooks 是否按预期工作

Tip:

你可以通过多次指定标志来一次加载多个插件：

```
claude --plugin-dir ./plugin-one --plugin-dir ./plugin-two
```

调试插件问题

如果你的插件不按预期工作：

1. **检查结构**：确保你的目录在插件根目录，而不是在 `.claude-plugin/` 内
2. **单独测试组件**：分别检查每个命令、agent 和 hook
3. **使用验证和调试工具**：有关 CLI 命令和故障排除技术，请参阅[调试和开发工具](#)

共享你的插件

当你的插件准备好共享时：

1. **添加文档**：包含一个 `README.md`，其中包含安装和使用说明
2. **版本化你的插件**：在你的 `plugin.json` 中使用[语义版本控制](#)
3. **创建或使用市场**：通过[插件市场](#)分发以供安装
4. **与他人测试**：在更广泛分发之前让团队成员测试插件

一旦你的插件在市场中，其他人可以使用[发现和安装插件](#)中的说明安装它。

向官方市场提交你的插件

要向官方 Anthropic 市场提交插件，请使用以下应用内提交表单之一：

- **Claude.ai**：claude.ai/settings/plugins/submit
- **Console**：platform.claude.com/plugins/submit

Note:

有关完整的技术规范、调试技术和分发策略，请参阅[插件参考](#)。

将现有配置转换为插件

如果你已经在 `.claude/` 目录中有 `skills` 或 `hooks`，你可以将它们转换为插件，以便更轻松地共享和分发。

迁移步骤

Step 1: 创建插件结构

创建一个新的插件目录：

```
mkdir -p my-plugin/.claude-plugin
```

在 `my-plugin/.claude-plugin/plugin.json` 处创建清单文件：

```
{
  "name": "my-plugin",
  "description": "Migrated from standalone configuration",
  "version": "1.0.0"
}
```

Step 2: 复制你现有的文件

将你现有的配置复制到插件目录：

```
## Copy commands
cp -r .claude/commands my-plugin/

## Copy agents (if any)
cp -r .claude/agents my-plugin/

## Copy skills (if any)
cp -r .claude/skills my-plugin/
```

Step 3: 迁移 hooks

如果你在设置中有 `hooks`，请创建一个 `hooks` 目录：

```
mkdir my-plugin/hooks
```

使用你的 hooks 配置创建 `my-plugin/hooks/hooks.json`。从你的 `.claude/settings.json` 或 `settings.local.json` 复制 `hooks` 对象，因为格式相同。命令在 stdin 上接收 hook 输入作为 JSON，所以使用 `jq` 提取文件路径：

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [{ "type": "command", "command":
          "jq -r '.tool_input.file_path' | xargs npm run lint:fix" }]
      }
    ]
  }
}
```

Step 4: 测试你迁移的插件

加载你的插件以验证一切正常：

```
claude --plugin-dir ./my-plugin
```

测试每个组件：运行你的命令、检查 agents 是否出现在 `/agents` 中，并验证 hooks 是否正确触发。

迁移时的变化

独立 (<code>.claude/</code>)	插件
仅在一个项目中可用	可以通过市场共享
<code>.claude/commands/</code> 中的文件	<code>plugin-name/commands/</code> 中的文件
<code>settings.json</code> 中的 Hooks	<code>hooks/hooks.json</code> 中的 Hooks
必须手动复制以共享	使用 <code>/plugin install</code> 安装

Note:

迁移后，你可以从 `.claude/` 中删除原始文件以避免重复。加载时插件版本将优先。

后续步骤

现在你了解了 Claude Code 的插件系统，以下是针对不同目标的建议路径：

对于插件用户

- [发现和安装插件](#)：浏览市场并安装插件
- [配置团队市场](#)：为你的团队设置存储库级别的插件

对于插件开发者

- [创建和分发市场](#)：打包和共享你的插件
- [插件参考](#)：完整的技术规范
- 深入了解特定的插件组件：
 - [Skills](#)：skill 开发详情
 - [Subagents](#)：agent 配置和功能
 - [Hooks](#)：事件处理和自动化
 - [MCP](#)：外部工具集成

Plugins 参考

Claude Code 插件系统的完整技术参考，包括架构、CLI 命令和组件规范。

Tip:

想要安装插件？请参阅[发现和安装插件](#)。如需创建插件，请参阅[Plugins](#)。如需分发插件，请参阅[Plugin marketplaces](#)。

本参考提供了 Claude Code 插件系统的完整技术规范，包括组件架构、CLI 命令和开发工具。

plugin 是一个自包含的组件目录，用于扩展 Claude Code 的自定义功能。插件组件包括 skills、agents、hooks、MCP servers 和 LSP servers。

Plugin 组件参考

Skills

Plugins 向 Claude Code 添加 skills，创建可由您或 Claude 调用的 `/name` 快捷方式。

位置：插件根目录中的 `skills/` 或 `commands/` 目录

文件格式：Skills 是包含 `SKILL.md` 的目录；commands 是简单的 markdown 文件

Skill 结构：

```
skills/
├── pdf-processor/
│   ├── SKILL.md
│   ├── reference.md (可选)
│   └── scripts/ (可选)
└── code-reviewer/
    └── SKILL.md
```

集成行为：

- 安装插件时会自动发现 Skills 和 commands
- Claude 可以根据任务上下文自动调用它们

- Skills 可以在 SKILL.md 旁边包含支持文件

有关完整详情，请参阅[Skills](#)。

Agents

Plugins 可以为特定任务提供专门的 subagents，Claude 可以在适当时自动调用。

位置：插件根目录中的 `agents/` 目录

文件格式：描述 agent 功能的 Markdown 文件

Agent 结构：

```
---
name: agent-name
description: 该 agent 的专长以及 Claude 应何时调用它
---
```

详细的系统提示，描述 agent 的角色、专业知识和行为。

集成点：

- Agents 出现在 `/agents` 界面中
- Claude 可以根据任务上下文自动调用 agents
- Agents 可以由用户手动调用
- Plugin agents 与内置 Claude agents 一起工作

有关完整详情，请参阅[Subagents](#)。

Hooks

Plugins 可以提供事件处理程序，自动响应 Claude Code 事件。

位置：插件根目录中的 `hooks/hooks.json`，或在 `plugin.json` 中内联

格式：具有事件匹配器和操作的 JSON 配置

Hook 配置：


```

{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/format-code.sh"
          }
        ]
      }
    ]
  }
}

```

可用事件：

- **PreToolUse**：Claude 使用任何工具之前
- **PostToolUse**：Claude 成功使用任何工具之后
- **PostToolUseFailure**：Claude 工具执行失败之后
- **PermissionRequest**：显示权限对话框时
- **UserPromptSubmit**：用户提交提示时
- **Notification**：Claude Code 发送通知时
- **Stop**：Claude 尝试停止时
- **SubagentStart**：subagent 启动时
- **SubagentStop**：subagent 尝试停止时
- **SessionStart**：会话开始时
- **SessionEnd**：会话结束时
- **TeammateIdle**：agent 团队队友即将空闲时
- **TaskCompleted**：任务被标记为已完成时
- **PreCompact**：对话历史被压缩之前

Hook 类型：

- **command**：执行 shell 命令或脚本
- **prompt**：使用 LLM 评估提示（使用 **\$ARGUMENTS** 占位符表示上下文）

- **agent**：运行具有工具的 agentic 验证器以完成复杂验证任务

MCP servers

Plugins 可以捆绑 Model Context Protocol (MCP) servers 以将 Claude Code 与外部工具和服务连接。

位置：插件根目录中的 `.mcp.json`，或在 `plugin.json` 中内联

格式：标准 MCP server 配置

MCP server 配置：

```
{
  "mcpServers": {
    "plugin-database": {
      "command": "${CLAUDE_PLUGIN_ROOT}/servers/db-server",
      "args": ["--config", "${CLAUDE_PLUGIN_ROOT}/config.json"],
      "env": {
        "DB_PATH": "${CLAUDE_PLUGIN_ROOT}/data"
      }
    },
    "plugin-api-client": {
      "command": "npx",
      "args": ["@company/mcp-server", "--plugin-mode"],
      "cwd": "${CLAUDE_PLUGIN_ROOT}"
    }
  }
}
```

集成行为：

- 启用插件时，Plugin MCP servers 会自动启动
- Servers 在 Claude 的工具包中显示为标准 MCP 工具
- Server 功能与 Claude 的现有工具无缝集成
- Plugin servers 可以独立于用户 MCP servers 进行配置

LSP servers

Tip:

想要使用 LSP plugins？从官方市场安装它们：在 `/plugin Discover` 选项卡中搜索“lsp”。本部分记录了如何为官方市场未涵盖的语言创建 LSP plugins。

Plugins 可以提供 [Language Server Protocol \(LSP\)](#) servers，在处理代码库时为 Claude 提供实时代码智能。

LSP 集成提供：

- **即时诊断**：Claude 在每次编辑后立即看到错误和警告
- **代码导航**：转到定义、查找引用和悬停信息
- **语言感知**：代码符号的类型信息和文档

位置：插件根目录中的 `.lsp.json`，或在 `plugin.json` 中内联

格式：将语言服务器名称映射到其配置的 JSON 配置

.lsp.json 文件格式：

```
{
  "go": {
    "command": "gopls",
    "args": ["serve"],
    "extensionToLanguage": {
      ".go": "go"
    }
  }
}
```

在 `plugin.json` 中内联：

```

{
  "name": "my-plugin",
  "lspServers": {
    "go": {
      "command": "gopls",
      "args": ["serve"],
      "extensionToLanguage": {
        ".go": "go"
      }
    }
  }
}

```

必需字段：

字段	描述
<code>command</code>	要执行的 LSP 二进制文件（必须在 PATH 中）
<code>extensionToLanguage</code>	将文件扩展名映射到语言标识符

可选字段：

字段	描述
<code>args</code>	LSP server 的命令行参数
<code>transport</code>	通信传输： <code>stdio</code> （默认）或 <code>socket</code>
<code>env</code>	启动 server 时要设置的环境变量
<code>initializationOptions</code>	在初始化期间传递给 server 的选项
<code>settings</code>	通过 <code>workspace/didChangeConfiguration</code> 传递的设置
<code>workspaceFolder</code>	server 的工作区文件夹路径
<code>startupTimeout</code>	等待 server 启动的最长时间（毫秒）
<code>shutdownTimeout</code>	等待正常关闭的最长时间（毫秒）
<code>restartOnCrash</code>	server 崩溃时是否自动重启

字段	描述
<code>maxRestarts</code>	放弃前的最大重启尝试次数

Warning:

您必须单独安装语言服务器二进制文件。LSP plugins 配置 Claude Code 如何连接到语言服务器，但它们不包括服务器本身。如果在 `/plugin Errors` 选项卡中看到 `Executable not found in $PATH`，请为您的语言安装所需的二进制文件。

可用的 LSP plugins :

Plugin	语言服务器	安装命令
<code>pyright-lsp</code>	Pyright (Python)	<code>pip install pyright</code> 或 <code>npm install -g pyright</code>
<code>typescript-lsp</code>	TypeScript Language Server	<code>npm install -g typescript-language-server typescript</code>
<code>rust-lsp</code>	rust-analyzer	参阅 rust-analyzer 安装

首先安装语言服务器，然后从市场安装 plugin。

Plugin 安装范围

安装 plugin 时，您选择一个范围，确定 plugin 的可用位置以及谁可以使用它：

范围	设置文件	用例
<code>user</code>	<code>~/.claude/settings.json</code>	在所有项目中可用的个人 plugins（默认）
<code>project</code>	<code>.claude/settings.json</code>	通过版本控制共享的团队 plugins
<code>local</code>	<code>.claude/settings.local.json</code>	项目特定的 plugins，gitignored
<code>managed</code>	Managed settings	托管 plugins（只读，仅更新）

Plugins 使用与其他 Claude Code 配置相同的范围系统。有关安装说明和范围标志，请参阅[安装 plugins](#)。有关范围的完整说明，请参阅[Configuration scopes](#)。

Plugin 清单架构

`.claude-plugin/plugin.json` 文件定义了您的 plugin 的元数据和配置。本部分记录了所有支持的字段和选项。

清单是可选的。如果省略，Claude Code 会自动发现[默认位置](#)中的组件，并从目录名称派生 plugin 名称。当您需要提供元数据或自定义组件路径时，使用清单。

完整架构

```
{
  "name": "plugin-name",
  "version": "1.2.0",
  "description": "Brief plugin description",
  "author": {
    "name": "Author Name",
    "email": "author@example.com",
    "url": "https://github.com/author"
  },
  "homepage": "https://docs.example.com/plugin",
  "repository": "https://github.com/author/plugin",
  "license": "MIT",
  "keywords": ["keyword1", "keyword2"],
  "commands": ["/custom/commands/special.md"],
  "agents": "/custom/agents/",
  "skills": "/custom/skills/",
  "hooks": "/config/hooks.json",
  "mcpServers": "/mcp-config.json",
  "outputStyles": "/styles/",
  "lspServers": "/.lsp.json"
}
```

必需字段

如果包含清单，`name` 是唯一必需的字段。

字段	类型	描述	示例
<code>name</code>	string	唯一标识符（kebab-case，无空格）	<code>"deployment-tools"</code>

此名称用于命名空间组件。例如，在 UI 中，名为 `plugin-dev` 的 plugin 的 agent `agent-creator` 将显示为 `plugin-dev:agent-creator`。

元数据字段

字段	类型	描述	示例
<code>version</code>	string	语义版本。如果也在市场条目中设置， <code>plugin.json</code> 优先。您只需在一个地方设置它。	<code>"2.1.0"</code>
<code>description</code>	string	plugin 目的的简要说明	<code>"Deployment automation tools"</code>
<code>author</code>	object	作者信息	<code>{"name": "Dev Team", "email": "dev@company.com"}</code>
<code>homepage</code>	string	文档 URL	<code>"https://docs.example.com"</code>
<code>repository</code>	string	源代码 URL	<code>"https://github.com/user/plugin"</code>
<code>license</code>	string	许可证标识符	<code>"MIT"、"Apache-2.0"</code>
<code>keywords</code>	array	发现标签	<code>["deployment", "ci-cd"]</code>

组件路径字段

字段	类型	描述	示例
commands	string array	其他命令文件/目录	<code>"/custom/cmd.md" 或 <code>["./cmd1.md"]</code></code>
agents	string array	其他 agent 文件	<code>"/custom/agents/reviewer.md"</code>
skills	string array	其他 skill 目录	<code>"/custom/skills/"</code>
hooks	string array object	Hook 配置路径或内联配置	<code>"/my-extra-hooks.json"</code>
mcpServers	string array object	MCP 配置路径或内联配置	<code>"/my-extra-mcp-config.json"</code>
outputStyles	string array	其他输出样式文件/目录	<code>"/styles/"</code>
lspServers	string array object	Language Server Protocol 配置用于代码智能（转到定义、查找引用等）	<code>"/.lsp.json"</code>

路径行为规则

重要：自定义路径补充默认目录 - 它们不替换默认目录。

- 如果 `commands/` 存在，除了自定义命令路径外，它也会被加载
- 所有路径必须相对于 plugin 根目录，并以 `./` 开头
- 来自自定义路径的命令使用相同的命名和命名空间规则
- 可以将多个路径指定为数组以获得灵活性

路径示例：


```
{
  "commands": [
    "./specialized/deploy.md",
    "./utilities/batch-process.md"
  ],
  "agents": [
    "./custom-agents/reviewer.md",
    "./custom-agents/tester.md"
  ]
}
```

环境变量

`${CLAUDE_PLUGIN_ROOT}`：包含 plugin 目录的绝对路径。在 hooks、MCP servers 和脚本中使用此变量，以确保无论安装位置如何都能使用正确的路径。

```
{
  "hooks": {
    "PostToolUse": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/process.sh"
          }
        ]
      }
    ]
  }
}
```

Plugin 缓存和文件解析

Plugins 通过以下两种方式之一指定：

- 通过 `claude --plugin-dir`，用于会话期间。
- 通过市场，为将来的会话安装。

出于安全和验证目的，Claude Code 将 `_市场_` plugins 复制到用户的本地 **plugin 缓存**（`~/.claude/plugins/cache`），而不是就地使用它们。在开发引用外部文件的 plugins 时，理解此行为很重要。

路径遍历限制

已安装的 plugins 无法引用其目录外的文件。遍历 plugin 根目录外的路径（例如 `../shared-utils`）在安装后将不起作用，因为这些外部文件不会被复制到缓存中。

使用外部依赖

如果您的 plugin 需要访问其目录外的文件，您可以在 plugin 目录中创建指向外部文件的符号链接。在复制过程中会遵守符号链接：

```
## 在您的 plugin 目录内
ln -s /path/to/shared-utils ./shared-utils
```

符号链接的内容将被复制到 plugin 缓存中。这在保持缓存系统安全优势的同时提供了灵活性。

Plugin 目录结构

标准 plugin 布局

完整的 plugin 遵循此结构：

```

enterprise-plugin/
├── .claude-plugin/           # 元数据目录 (可选)
│   ├── plugin.json         # plugin 清单
│   └── commands/           # 默认命令位置
│       ├── status.md
│       └── logs.md
├── agents/                  # 默认 agent 位置
│   ├── security-reviewer.md
│   ├── performance-tester.md
│   └── compliance-checker.md
├── skills/                  # Agent Skills
│   ├── code-reviewer/
│   │   └── SKILL.md
│   └── pdf-processor/
│       ├── SKILL.md
│       └── scripts/
├── hooks/                   # Hook 配置
│   ├── hooks.json          # 主 hook 配置
│   └── security-hooks.json # 其他 hooks
├── settings.json            # plugin 的默认设置
├── .mcp.json                # MCP server 定义
├── .lsp.json                # LSP server 配置
├── scripts/                 # Hook 和实用脚本
│   ├── security-scan.sh
│   ├── format-code.py
│   └── deploy.js
├── LICENSE                  # 许可证文件
└── CHANGELOG.md             # 版本历史

```

Warning:

`.claude-plugin/` 目录包含 `plugin.json` 文件。所有其他目录 (`commands/`、`agents/`、`skills/`、`hooks/`) 必须在 `plugin` 根目录，而不是在 `.claude-plugin/` 内。

文件位置参考

组件	默认位置	目的
清单	<code>.claude-plugin/plugin.json</code>	Plugin 元数据和配置（可选）
命令	<code>commands/</code>	Skill Markdown 文件（遗留；新 skills 使用 <code>skills /</code> ）
Agents	<code>agents/</code>	Subagent Markdown 文件
Skills	<code>skills/</code>	具有 <code><name>/SKILL.md</code> 结构的 Skills
Hooks	<code>hooks/hooks.json</code>	Hook 配置
MCP servers	<code>.mcp.json</code>	MCP server 定义
LSP servers	<code>.lsp.json</code>	语言服务器配置
设置	<code>settings.json</code>	启用 plugin 时应用的默认配置。目前仅支持 <code>agent</code> 设置

CLI 命令参考

Claude Code 提供了用于非交互式 plugin 管理的 CLI 命令，对脚本和自动化很有用。

plugin install

从可用市场安装 plugin。

```
claude plugin install <plugin> [options]
```

参数：

- `<plugin>`：Plugin 名称或 `plugin-name@marketplace-name` 用于特定市场

选项：

选项	描述	默认值
<code>-s, --scope <scope></code>	安装范围: <code>user</code> 、 <code>project</code> 或 <code>local</code>	<code>user</code>
<code>-h, --help</code>	显示命令帮助	

范围确定将已安装的 plugin 添加到哪个设置文件。例如，`--scope project` 写入 `.claude/settings.json` 中的 `enabledPlugins`，使 plugin 对克隆项目存储库的每个人都可用。

示例：

```
## 安装到用户范围（默认）
claude plugin install formatter@my-marketplace

## 安装到项目范围（与团队共享）
claude plugin install formatter@my-marketplace --scope project

## 安装到本地范围（gitignored）
claude plugin install formatter@my-marketplace --scope local
```

plugin uninstall

删除已安装的 plugin。

```
claude plugin uninstall <plugin> [options]
```

参数：

- `<plugin>` : Plugin 名称或 `plugin-name@marketplace-name`

选项：

选项	描述	默认值
<code>-s, --scope <scope></code>	从范围卸载: <code>user</code> 、 <code>project</code> 或 <code>local</code>	<code>user</code>
<code>-h, --help</code>	显示命令帮助	

别名: `remove`、`rm`

plugin enable

启用已禁用的 plugin。

```
claude plugin enable <plugin> [options]
```

参数:

- **<plugin>** : Plugin 名称或 `plugin-name@marketplace-name`

选项:

选项	描述	默认值
<code>-s, --scope <scope></code>	要启用的范围: <code>user</code> 、 <code>project</code> 或 <code>local</code>	<code>user</code>
<code>-h, --help</code>	显示命令帮助	

plugin disable

禁用 plugin 而不卸载它。

```
claude plugin disable <plugin> [options]
```

参数:

- **<plugin>** : Plugin 名称或 `plugin-name@marketplace-name`

选项:

选项	描述	默认值
<code>-s, --scope <scope></code>	要禁用的范围: <code>user</code> 、 <code>project</code> 或 <code>local</code>	<code>user</code>
<code>-h, --help</code>	显示命令帮助	

plugin update

将 plugin 更新到最新版本。

```
claude plugin update <plugin> [options]
```

参数:

- `<plugin>` : Plugin 名称或 `plugin-name@marketplace-name`

选项:

选项	描述	默认值
<code>-s, --scope <scope></code>	要更新的范围: <code>user</code> 、 <code>project</code> 、 <code>local</code> 或 <code>managed</code>	<code>user</code>
<code>-h, --help</code>	显示命令帮助	

调试和开发工具

调试命令

使用 `claude --debug` (或 TUI 中的 `/debug`) 查看 plugin 加载详情:

这显示:

- 正在加载哪些 plugins
- plugin 清单中的任何错误
- 命令、agent 和 hook 注册
- MCP server 初始化

常见问题

问题	原因	解决方案
Plugin 未加载	无效的 <code>plugin.json</code>	使用 <code>claude plugin validate</code> 或 <code>/plugin validate</code> 验证 JSON 语法
命令未出现	目录结构错误	确保 <code>commands/</code> 在根目录, 而不是在 <code>.claude-plugin/</code> 中
Hooks 未触发	脚本不可执行	运行 <code>chmod +x script.sh</code>
MCP server 失败	缺少 <code>\${CLAUDE_PLUGIN_ROOT}</code>	对所有 plugin 路径使用变量

问题	原因	解决方案
路径错误	使用了绝对路径	所有路径必须是相对的，并以 <code>./</code> 开头
<code>LSP Executable not found in \$PATH</code>	语言服务器未安装	安装二进制文件（例如， <code>npm install -g typescript-language-server typescript</code> ）

示例错误消息

清单验证错误：

- `Invalid JSON syntax: Unexpected token } in JSON at position 142`：检查缺少的逗号、多余的逗号或未引用的字符串
- `Plugin has an invalid manifest file at .claude-plugin/plugin.json. Validation errors: name: Required`：缺少必需字段
- `Plugin has a corrupt manifest file at .claude-plugin/plugin.json. JSON parse error: ...`：JSON 语法错误

Plugin 加载错误：

- `Warning: No commands found in plugin my-plugin custom directory: ./cmds. Expected .md files or SKILL.md in subdirectories.`：命令路径存在但不包含有效的命令文件
- `Plugin directory not found at path: ./plugins/my-plugin. Check that the marketplace entry has the correct path.`：marketplace.json 中的 `source` 路径指向不存在的目录
- `Plugin my-plugin has conflicting manifests: both plugin.json and marketplace entry specify components.`：删除重复的组件定义或删除 marketplace 条目中的 `strict: false`

Hook 故障排除

Hook 脚本未执行：

- 检查脚本是否可执行：`chmod +x ./scripts/your-script.sh`
- 验证 shebang 行：第一行应该是 `#!/bin/bash` 或 `#!/usr/bin/env bash`
- 检查路径是否使用 `${CLAUDE_PLUGIN_ROOT}`：`"command": "${CLAUDE_PLUGIN_ROOT}/scripts/your-script.sh"`
- 手动测试脚本：`./scripts/your-script.sh`

Hook 未在预期事件上触发：

1. 验证事件名称是否正确（区分大小写）：`PostToolUse`，而不是 `postToolUse`
2. 检查匹配器模式是否与您的工具匹配：`"matcher": "Write|Edit"` 用于文件操作
3. 确认 hook 类型有效：`command`、`prompt` 或 `agent`

MCP server 故障排除

Server 未启动：

1. 检查命令是否存在且可执行
2. 验证所有路径是否使用 `${CLAUDE_PLUGIN_ROOT}` 变量
3. 检查 MCP server 日志：`claude --debug` 显示初始化错误
4. 在 Claude Code 外手动测试 server

Server 工具未出现：

1. 确保 server 在 `.mcp.json` 或 `plugin.json` 中正确配置
2. 验证 server 是否正确实现 MCP 协议
3. 检查调试输出中的连接超时

目录结构错误

症状：Plugin 加载但组件（命令、agents、hooks）缺失。

正确结构：组件必须在 plugin 根目录，而不是在 `.claude-plugin/` 内。只有 `plugin.json` 属于 `.claude-plugin/`。

```
my-plugin/
├── .claude-plugin/
│   └── plugin.json    ← 仅清单在此处
├── commands/         ← 在根级别
├── agents/            ← 在根级别
└── hooks/             ← 在根级别
```

如果您的组件在 `.claude-plugin/` 内，请将它们移到 plugin 根目录。

调试清单：

1. 运行 `claude --debug` 并查找“loading plugin”消息
2. 检查每个组件目录是否在调试输出中列出
3. 验证文件权限允许读取 plugin 文件

分发和版本管理参考

版本管理

遵循语义版本控制进行 plugin 发布：

```
{  
  "name": "my-plugin",  
  "version": "2.1.0"  
}
```

版本格式： MAJOR.MINOR.PATCH

- **MAJOR**: 破坏性更改（不兼容的 API 更改）
- **MINOR**: 新功能（向后兼容的添加）
- **PATCH**: 错误修复（向后兼容的修复）

最佳实践：

- 从 1.0.0 开始进行第一个稳定版本
- 在分发更改之前更新 plugin.json 中的版本
- 在 CHANGELOG.md 文件中记录更改
- 使用预发布版本，如 2.0.0-beta.1 进行测试

Warning:

Claude Code 使用版本来确定是否更新您的 plugin。如果您更改了 plugin 的代码但没有在 plugin.json 中提升版本，您的 plugin 的现有用户由于缓存而看不到您的更改。

如果您的 plugin 在 [市场](#) 目录中，您可以通过 marketplace.json 管理版本，而不是从 plugin.json 中省略 version 字段。

另请参阅

- [Plugins](#) - 教程和实际用法
- [Plugin marketplaces](#) - 创建和管理市场
- [Skills](#) - Skill 开发详情
- [Subagents](#) - Agent 配置和功能
- [Hooks](#) - 事件处理和自动化

- [MCP](#) - 外部工具集成
- [Settings](#) - Plugins 的配置选项

创建和分发 plugin marketplace

构建和托管 plugin marketplace，以在团队和社区中分发 Claude Code 扩展。

plugin marketplace 是一个目录，让你能够将 plugins 分发给他人。Marketplace 提供集中式发现、版本跟踪、自动更新以及对多种源类型（git 存储库、本地路径等）的支持。本指南展示了如何创建自己的 marketplace，与你的团队或社区共享 plugins。

想要从现有 marketplace 安装 plugins？请参阅[发现和安装预构建的 plugins](#)。

概述

创建和分发 marketplace 涉及：

1. **创建 plugins**：使用命令、agents、hooks、MCP servers 或 LSP servers 构建一个或多个 plugins。本指南假设你已经要有要分发的 plugins；有关如何创建 plugins 的详细信息，请参阅[创建 plugins](#)。
2. **创建 marketplace 文件**：定义一个 `marketplace.json`，列出你的 plugins 及其位置（请参阅[创建 marketplace 文件](#)）。
3. **托管 marketplace**：推送到 GitHub、GitLab 或其他 git 主机（请参阅[托管和分发 marketplaces](#)）。
4. **与用户共享**：用户使用 `/plugin marketplace add` 添加你的 marketplace 并安装单个 plugins（请参阅[发现和安装 plugins](#)）。

一旦你的 marketplace 上线，你可以通过推送更改到你的存储库来更新它。用户使用 `/plugin marketplace update` 刷新他们的本地副本。

演练：创建本地 marketplace

此示例创建一个包含一个 plugin 的 marketplace：一个用于代码审查的 `/quality-review` skill。你将创建目录结构、添加 skill、创建 plugin manifest 和 marketplace 目录，然后安装并测试它。

Step 1: 创建目录结构

```
mkdir -p my-marketplace/.claude-plugin
mkdir -p my-marketplace/plugins/quality-review-plugin/.claude-plugin
mkdir -p my-marketplace/plugins/quality-review-plugin/skills/quality-review
```

Step 2: 创建 skill

创建一个 `SKILL.md` 文件，定义 `/quality-review` skill 的功能。

```
---  
description: Review code for bugs, security, and performance  
disable-model-invocation: true  
---  
  
Review the code I've selected or the recent changes for:  
- Potential bugs or edge cases  
- Security concerns  
- Performance issues  
- Readability improvements  
  
Be concise and actionable.
```

Step 3: 创建 plugin manifest

创建一个 `plugin.json` 文件，描述该 plugin。manifest 位于 `.claude-plugin/` 目录中。

```
{  
  "name": "quality-review-plugin",  
  "description": "Adds a /quality-review skill for quick code reviews",  
  "version": "1.0.0"  
}
```

Step 4: 创建 marketplace 文件

创建列出你的 plugin 的 marketplace 目录。

```
{
  "name": "my-plugins",
  "owner": {
    "name": "Your Name"
  },
  "plugins": [
    {
      "name": "quality-review-plugin",
      "source": "./plugins/quality-review-plugin",
      "description": "Adds a /quality-review skill for quick code reviews"
    }
  ]
}
```

Step 5: 添加和安装

添加 marketplace 并安装 plugin。

```
/plugin marketplace add ./my-marketplace
/plugin install quality-review-plugin@my-plugins
```

Step 6: 尝试一下

在编辑器中选择一些代码并运行你的新命令。

```
/review
```

要了解更多关于 plugins 可以做什么的信息，包括 hooks、agents、MCP servers 和 LSP servers，请参阅 [Plugins](#)。

Note:

plugins 如何安装：当用户安装 plugin 时，Claude Code 将 plugin 目录复制到缓存位置。这意味着 plugins 无法使用 `../shared-utils` 之类的路径引用其目录外的文件，因为这些文件不会被复制。

如果你需要在 plugins 之间共享文件，请使用符号链接（在复制期间会被跟踪）。有关详细信息，请参阅 [Plugin 缓存和文件解析](#)。

创建 marketplace 文件

在你的存储库根目录中创建 `.claude-plugin/marketplace.json`。此文件定义你的 marketplace 的名称、所有者信息以及包含其源的 plugins 列表。

每个 plugin 条目至少需要一个 `name` 和 `source`（从哪里获取它）。有关所有可用字段，请参阅下面的[完整架构](#)。

```
{
  "name": "company-tools",
  "owner": {
    "name": "DevTools Team",
    "email": "devtools@example.com"
  },
  "plugins": [
    {
      "name": "code-formatter",
      "source": "./plugins/formatter",
      "description": "Automatic code formatting on save",
      "version": "2.1.0",
      "author": {
        "name": "DevTools Team"
      }
    },
    {
      "name": "deployment-tools",
      "source": {
        "source": "github",
        "repo": "company/deploy-plugin"
      },
      "description": "Deployment automation tools"
    }
  ]
}
```

Marketplace 架构

必需字段

字段	类型	描述	示例
<code>name</code>	string	Marketplace 标识符（kebab-case，无空格）。这是面向公众的：用户在安装 plugins 时会看到它（例如， <code>/plugin install my-tool@your-marketplace</code> ）。	<code>"acme-tools"</code>
<code>owner</code>	object	Marketplace 维护者信息（ 见下面的字段 ）	
<code>plugins</code>	array	可用 plugins 列表	见下文

Note:

保留名称：以下 marketplace 名称为 Anthropic 官方使用保留，第三方 marketplaces 无法使用：`claude-code-marketplace`、`claude-code-plugins`、`claude-plugins-official`、`anthropic-marketplace`、`anthropic-plugins`、`agent-skills`、`life-sciences`。冒充官方 marketplaces 的名称（如 `official-claude-plugins` 或 `anthropic-tools-v2`）也被阻止。

所有者字段

字段	类型	必需	描述
<code>name</code>	string	是	维护者或团队的名称
<code>email</code>	string	否	维护者的联系电子邮件

可选元数据

字段	类型	描述
<code>metadata.description</code>	string	简短的 marketplace 描述

字段	类型	描述
<code>metadata.version</code>	string	Marketplace 版本
<code>metadata.pluginRoot</code>	string	前置到相对 plugin 源路径的基目录（例如， <code>./plugins</code> 让你写 <code>"source": "formatter"</code> 而不是 <code>"source": "./plugins/formatter"</code> ）

Plugin 条目

`plugins` 数组中的每个 plugin 条目描述一个 plugin 及其位置。你可以包含 [plugin manifest 架构](#) 中的任何字段（如 `description`、`version`、`author`、`commands`、`hooks` 等），加上这些 marketplace 特定的字段：`source`、`category`、`tags` 和 `strict`。

必需字段

字段	类型	描述
<code>name</code>	string	Plugin 标识符（kebab-case，无空格）。这是面向公众的：用户在安装时会看到它（例如， <code>/plugin install my-plugin@marketplace</code> ）。
<code>source</code>	string object	从哪里获取 plugin（见下面的 Plugin 源 ）

可选 plugin 字段

标准元数据字段：

字段	类型	描述
<code>description</code>	string	简短的 plugin 描述
<code>version</code>	string	Plugin 版本
<code>author</code>	object	Plugin 作者信息（ <code>name</code> 必需， <code>email</code> 可选）

字段	类型	描述
homepage	string	Plugin 主页或文档 URL
repository	string	源代码存储库 URL
license	string	SPDX 许可证标识符（例如，MIT、Apache-2.0）
keywords	array	用于 plugin 发现和分类的标签
category	string	Plugin 类别以供组织
tags	array	用于可搜索性的标签
strict	boolean	控制 <code>plugin.json</code> 是否是组件定义的权威（默认：true）。见下面的 Strict 模式 。

组件配置字段：

字段	类型	描述
commands	string array	命令文件或目录的自定义路径
agents	string array	agent 文件的自定义路径
hooks	string object	自定义 hooks 配置或 hooks 文件的路径
mcpServers	string object	MCP server 配置或 MCP 配置的路径
lspServers	string object	LSP server 配置或 LSP 配置的路径

Plugin 源

Plugin 源告诉 Claude Code 在你的 marketplace 中列出的每个单独 plugin 从哪里获取。这些在 `marketplace.json` 中每个 plugin 条目的 `source` 字段中设置。

一旦 plugin 被克隆或复制到本地机器，它就会被复制到本地版本化 plugin 缓存中，位置为 `~/.claude/plugins/cache`。

源	类型	字段	注释
相对路径	<code>string</code> (例如 <code>"./my-plugin"</code>)	—	marketplace repo 中的本地目录。必须以 <code>./</code> 开头
<code>github</code>	object	<code>repo</code> 、 <code>ref?</code> 、 <code>sha?</code>	
<code>url</code>	object	<code>url</code> (必须以 <code>.git</code> 结尾)、 <code>ref?</code> 、 <code>sha?</code>	Git URL 源
<code>git-subdir</code>	object	<code>url</code> 、 <code>path</code> 、 <code>ref?</code> 、 <code>sha?</code>	git repo 中的子目录。稀疏克隆以最小化大型 monorepos 的带宽
<code>npm</code>	object	<code>package</code> 、 <code>version?</code> 、 <code>registry?</code>	通过 <code>npm install</code> 安装
<code>pip</code>	object	<code>package</code> 、 <code>version?</code> 、 <code>registry?</code>	通过 pip 安装

Note:

Marketplace 源与 plugin 源： 这些是控制不同事物的不同概念。

- **Marketplace 源** — 从哪里获取 `marketplace.json` 目录本身。在用户运行 `/plugin marketplace add` 或在 `extraKnownMarketplaces` 设置中设置。支持 `ref` (分支/标签) 但不支持 `sha`。
- **Plugin 源** — 从哪里获取 marketplace 中列出的单个 plugin。在 `marketplace.json` 内每个 plugin 条目的 `source` 字段中设置。支持 `ref` (分支/标签) 和 `sha` (精确提交)。

例如，托管在 `acme-corp/plugin-catalog` 的 marketplace (marketplace 源) 可以列出从 `acme-corp/code-formatter` 获取的 plugin (plugin 源)。marketplace 源和 plugin 源指向不同的存储库，并独立固定。

相对路径

对于同一存储库中的 plugins：

```
{
  "name": "my-plugin",
  "source": "./plugins/my-plugin"
}
```

Note:

相对路径仅在用户通过 Git（GitHub、GitLab 或 git URL）添加你的 marketplace 时有效。如果用户通过直接 URL 添加你的 marketplace 到 `marketplace.json` 文件，相对路径将无法正确解析。对于基于 URL 的分发，请改用 GitHub、npm 或 git URL 源。有关详细信息，请参阅[故障排除](#)。

GitHub 存储库

```
{
  "name": "github-plugin",
  "source": {
    "source": "github",
    "repo": "owner/plugin-repo"
  }
}
```

你可以固定到特定的分支、标签或提交：

```
{
  "name": "github-plugin",
  "source": {
    "source": "github",
    "repo": "owner/plugin-repo",
    "ref": "v2.0.0",
    "sha": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0"
  }
}
```

字段	类型	描述
repo	string	必需。owner/repo 格式的 GitHub 存储库
ref	string	可选。Git 分支或标签（默认为存储库默认分支）
sha	string	可选。完整的 40 字符 git 提交 SHA 以固定到精确版本

Git 存储库

```
{
  "name": "git-plugin",
  "source": {
    "source": "url",
    "url": "https://gitlab.com/team/plugin.git"
  }
}
```

你可以固定到特定的分支、标签或提交：

```
{
  "name": "git-plugin",
  "source": {
    "source": "url",
    "url": "https://gitlab.com/team/plugin.git",
    "ref": "main",
    "sha": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0"
  }
}
```

字段	类型	描述
url	string	必需。完整的 git 存储库 URL（必须以 .git 结尾）

字段	类型	描述
<code>ref</code>	string	可选。Git 分支或标签（默认为存储库默认分支）
<code>sha</code>	string	可选。完整的 40 字符 git 提交 SHA 以固定到精确版本

Git 子目录

使用 `git-subdir` 指向位于 git 存储库子目录中的 plugin。Claude Code 使用稀疏的部分克隆来仅获取子目录，最小化大型 monorepos 的带宽。

```
{
  "name": "my-plugin",
  "source": {
    "source": "git-subdir",
    "url": "https://github.com/acme-corp/monorepo.git",
    "path": "tools/claude-plugin"
  }
}
```

你可以固定到特定的分支、标签或提交：

```
{
  "name": "my-plugin",
  "source": {
    "source": "git-subdir",
    "url": "https://github.com/acme-corp/monorepo.git",
    "path": "tools/claude-plugin",
    "ref": "v2.0.0",
    "sha": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0"
  }
}
```

`url` 字段也接受 GitHub 简写（`owner/repo`）或 SSH URL（`git@github.com:owner/repo.git`）。

字段	类型	描述
<code>url</code>	string	必需。Git 存储库 URL 、GitHub <code>owner/repo</code> 简写 或 SSH URL
<code>path</code>	string	必需。repo 中包含 plugin 的子目录路径（例如， <code>"tools/claude-plugin"</code> ）
<code>ref</code>	string	可选。Git 分支或标签（默认为存储库默认分支）
<code>sha</code>	string	可选。完整的 40 字符 git 提交 SHA 以固定到精确版本

npm 包

作为 npm 包分发的 Plugins 使用 `npm install` 安装。这适用于公共 npm registry 上的任何包或你的团队托管的私有 registry。

```
{
  "name": "my-npm-plugin",
  "source": {
    "source": "npm",
    "package": "@acme/claude-plugin"
  }
}
```

要固定到特定版本，请添加 `version` 字段：

```
{
  "name": "my-npm-plugin",
  "source": {
    "source": "npm",
    "package": "@acme/claude-plugin",
    "version": "2.1.0"
  }
}
```

要从私有或内部 registry 安装，请添加 `registry` 字段：

```
{
  "name": "my-npm-plugin",
  "source": {
    "source": "npm",
    "package": "@acme/claude-plugin",
    "version": "^2.0.0",
    "registry": "https://npm.example.com"
  }
}
```

字段	类型	描述
<code>package</code>	string	必需。包名称或作用域包（例如， <code>@org/plugin</code> ）
<code>version</code>	string	可选。版本或版本范围（例如， <code>2.1.0</code> 、 <code>^2.0.0</code> 、 <code>~1.5.0</code> ）
<code>registry</code>	string	可选。自定义 npm registry URL。默认为系统 npm registry（通常为 <code>npmjs.org</code> ）

高级 plugin 条目

此示例显示了使用许多可选字段的 plugin 条目，包括命令、agents、hooks 和 MCP servers 的自定义路径：


```

{
  "name": "enterprise-tools",
  "source": {
    "source": "github",
    "repo": "company/enterprise-plugin"
  },
  "description": "Enterprise workflow automation tools",
  "version": "2.1.0",
  "author": {
    "name": "Enterprise Team",
    "email": "enterprise@example.com"
  },
  "homepage": "https://docs.example.com/plugins/enterprise-tools",
  "repository": "https://github.com/company/enterprise-plugin",
  "license": "MIT",
  "keywords": ["enterprise", "workflow", "automation"],
  "category": "productivity",
  "commands": [
    "./commands/core/",
    "./commands/enterprise/",
    "./commands/experimental/preview.md"
  ],
  "agents": ["/agents/security-reviewer.md", "/agents/compliance-checker.md"],
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/validate.sh"
          }
        ]
      }
    ]
  },
  "mcpServers": {
    "enterprise-db": {

```

```
    "command": "${CLAUDE_PLUGIN_ROOT}/servers/db-server",
    "args": ["--config", "${CLAUDE_PLUGIN_ROOT}/config.json"]
  },
  "strict": false
}
```

需要注意的关键事项：

- **commands 和 agents**：你可以指定多个目录或单个文件。路径相对于 plugin 根目录。
- **`${CLAUDE_PLUGIN_ROOT}`**：在 hooks 和 MCP server 配置中使用此变量来引用 plugin 安装目录中的文件。这是必要的，因为 plugins 在安装时被复制到缓存位置。
- **`strict: false`**：由于这设置为 `false`，plugin 不需要自己的 `plugin.json`。marketplace 条目定义了一切。见下面的 [Strict 模式](#)。

Strict 模式

`strict` 字段控制 `plugin.json` 是否是组件定义（命令、agents、hooks、skills、MCP servers、输出样式）的权威。

值	行为
<code>true</code> （默认）	<code>plugin.json</code> 是权威。marketplace 条目可以用额外的组件补充它，两个源都被合并。
<code>false</code>	marketplace 条目是完整的定义。如果 plugin 也有声明组件的 <code>plugin.json</code> ，那就是冲突，plugin 无法加载。

何时使用每种模式：

- **`strict: true`**：plugin 有自己的 `plugin.json` 并管理自己的组件。marketplace 条目可以在顶部添加额外的命令或 hooks。这是默认值，适用于大多数 plugins。
- **`strict: false`**：marketplace 操作员想要完全控制。plugin repo 提供原始文件，marketplace 条目定义这些文件中的哪些被公开为命令、agents、hooks 等。当 marketplace 以不同于 plugin 作者意图的方式重组或策划 plugin 的组件时很有用。

托管和分发 marketplaces

在 GitHub 上托管（推荐）

GitHub 提供最简单的分发方法：

1. **创建存储库**：为你的 marketplace 设置一个新存储库

2. 添加 marketplace 文件：使用你的 plugin 定义创建 `.claude-plugin/marketplace.json`

3. 与团队共享：用户使用 `/plugin marketplace add owner/repo` 添加你的 marketplace

优点：内置版本控制、问题跟踪和团队协作功能。

在其他 git 服务上托管

任何 git 托管服务都可以工作，例如 GitLab、Bitbucket 和自托管服务器。用户使用完整的存储库 URL 添加：

```
/plugin marketplace add https://gitlab.com/company/plugins.git
```

私有存储库

Claude Code 支持从私有存储库安装 plugins。对于手动安装和更新，Claude Code 使用你现有的 git 凭证助手。如果 `git clone` 对你终端中的私有存储库有效，它在 Claude Code 中也有效。常见的凭证助手包括用于 GitHub 的 `gh auth login`、macOS Keychain 和 `git-credential-store`。

后台自动更新在启动时运行，不使用凭证助手，因为交互式提示会阻止 Claude Code 启动。要为私有 marketplaces 启用自动更新，请在你的环境中设置适当的身份验证令牌：

提供商	环境变量	注释
GitHub	<code>GITHUB_TOKEN</code> 或 <code>GH_TOKEN</code>	个人访问令牌或 GitHub App 令牌
GitLab	<code>GITLAB_TOKEN</code> 或 <code>GL_TOKEN</code>	个人访问令牌或项目令牌
Bitbucket	<code>BITBUCKET_TOKEN</code>	应用密码或存储库访问令牌

在你的 shell 配置中设置令牌（例如，`.bashrc`、`.zshrc`）或在运行 Claude Code 时传递它：

```
export GITHUB_TOKEN=ghp_XXXXXXXXXXXXXXXXXXXX
```

Note:

对于 CI/CD 环境，将令牌配置为秘密环境变量。GitHub Actions 自动为同一组织中的存储库提供 `GITHUB_TOKEN`。

在分发前本地测试

在共享前本地测试你的 marketplace：

```
/plugin marketplace add ./my-local-marketplace  
/plugin install test-plugin@my-local-marketplace
```

有关完整的添加命令范围（GitHub、Git URL、本地路径、远程 URL），请参阅[添加 marketplaces](#)。

为你的团队要求 marketplaces

你可以配置你的存储库，以便当团队成员信任项目文件夹时，他们会自动被提示安装你的 marketplace。将你的 marketplace 添加到 `.claude/settings.json`：

```
{  
  "extraKnownMarketplaces": {  
    "company-tools": {  
      "source": {  
        "source": "github",  
        "repo": "your-org/claude-plugins"  
      }  
    }  
  }  
}
```

你也可以指定默认应启用哪些 plugins：

```
{  
  "enabledPlugins": {  
    "code-formatter@company-tools": true,  
    "deployment-tools@company-tools": true  
  }  
}
```

有关完整的配置选项，请参阅 [Plugin 设置](#)。

托管 marketplace 限制

对于需要严格控制 plugin 源的组织，管理员可以使用托管设置中的 `strictKnownMarketplaces` 设置限制用户允许添加哪些 plugin marketplaces。

当在托管设置中配置 `strictKnownMarketplaces` 时，限制行为取决于值：

值	行为
未定义（默认）	无限制。用户可以添加任何 marketplace
空数组 <code>[]</code>	完全锁定。用户无法添加任何新 marketplaces
源列表	用户只能添加与允许列表完全匹配的 marketplaces

常见配置

禁用所有 marketplace 添加：

```
{
  "strictKnownMarketplaces": []
}
```

仅允许特定 marketplaces：

```
{
  "strictKnownMarketplaces": [
    {
      "source": "github",
      "repo": "acme-corp/approved-plugins"
    },
    {
      "source": "github",
      "repo": "acme-corp/security-tools",
      "ref": "v2.0"
    },
    {
      "source": "url",
      "url": "https://plugins.example.com/marketplace.json"
    }
  ]
}
```

使用主机上的正则表达式模式匹配允许来自内部 git 服务器的所有 marketplaces：

```
{
  "strictKnownMarketplaces": [
    {
      "source": "hostPattern",
      "hostPattern": "^github\\.example\\.com$"
    }
  ]
}
```

使用路径上的正则表达式模式匹配允许来自特定目录的基于文件系统的 marketplaces：

```
{
  "strictKnownMarketplaces": [
    {
      "source": "pathPattern",
      "pathPattern": "^/opt/approved/"
    }
  ]
}
```

使用 `"*"` 作为 `pathPattern` 来允许任何文件系统路径，同时仍然使用 `hostPattern` 控制网络源。

限制如何工作

限制在 plugin 安装过程的早期进行验证，在任何网络请求或文件系统操作之前。这可以防止未授权的 marketplace 访问尝试。

允许列表对大多数源类型使用精确匹配。要允许 marketplace，所有指定的字段必须完全匹配：

- 对于 GitHub 源： `repo` 是必需的，如果在允许列表中指定， `ref` 或 `path` 也必须匹配
- 对于 URL 源：完整 URL 必须完全匹配
- 对于 `hostPattern` 源：marketplace 主机与正则表达式模式匹配
- 对于 `pathPattern` 源：marketplace 的文件系统路径与正则表达式模式匹配

因为 `strictKnownMarketplaces` 在[托管设置](#)中设置，个别用户和项目配置无法覆盖这些限制。

有关完整的配置详细信息，包括所有支持的源类型和与 `extraKnownMarketplaces` 的比较，请参阅[strictKnownMarketplaces 参考](#)。

版本解析和发布渠道

Plugin 版本确定缓存路径和更新检测。你可以在 plugin manifest (`plugin.json`) 或 marketplace 条目 (`marketplace.json`) 中指定版本。

Warning:

如果可能，避免在两个地方都设置版本。plugin manifest 总是无声地获胜，这可能导致 marketplace 版本被忽略。对于相对路径 plugins，在 marketplace 条目中设置版本。对于所有其他 plugin 源，在 plugin manifest 中设置它。

设置发布渠道

要为你的 plugins 支持”稳定”和”最新”发布渠道，你可以设置两个指向同一 repo 的不同 refs 或 SHAs 的 marketplaces。然后，你可以通过[托管设置](#)将两个 marketplaces 分配给不同的用户组。

Warning:

plugin 的 `plugin.json` 必须在每个固定的 ref 或提交处声明不同的 `version`。如果两个 refs 或提交具有相同的 manifest 版本，Claude Code 会将它们视为相同并跳过更新。

示例

```
{
  "name": "stable-tools",
  "plugins": [
    {
      "name": "code-formatter",
      "source": {
        "source": "github",
        "repo": "acme-corp/code-formatter",
        "ref": "stable"
      }
    }
  ]
}
```



```
{
  "name": "latest-tools",
  "plugins": [
    {
      "name": "code-formatter",
      "source": {
        "source": "github",
        "repo": "acme-corp/code-formatter",
        "ref": "latest"
      }
    }
  ]
}
```

将渠道分配给用户组

通过托管设置将每个 marketplace 分配给适当的用户组。例如，稳定组接收：

```
{
  "extraKnownMarketplaces": {
    "stable-tools": {
      "source": {
        "source": "github",
        "repo": "acme-corp/stable-tools"
      }
    }
  }
}
```

早期访问组改为接收 `latest-tools`：

```
{
  "extraKnownMarketplaces": {
    "latest-tools": {
      "source": {
        "source": "github",
        "repo": "acme-corp/latest-tools"
      }
    }
  }
}
```

验证和测试

在共享前测试你的 marketplace。

验证你的 marketplace JSON 语法：

```
claude plugin validate .
```

或从 Claude Code 内：

```
/plugin validate .
```

添加 marketplace 进行测试：

```
/plugin marketplace add ./path/to/marketplace
```

安装测试 plugin 以验证一切正常：

```
/plugin install test-plugin@marketplace-name
```

有关完整的 plugin 测试工作流，请参阅[本地测试你的 plugins](#)。有关技术故障排除，请参阅[Plugins 参考](#)。

故障排除

Marketplace 未加载

症状：无法添加 marketplace 或从中看到 plugins

解决方案：

- 验证 marketplace URL 是否可访问
- 检查 `.claude-plugin/marketplace.json` 是否存在于指定路径
- 使用 `claude plugin validate` 或 `/plugin validate` 确保 JSON 语法有效
- 对于私有存储库，确认你有访问权限

Marketplace 验证错误

从你的 marketplace 目录运行 `claude plugin validate .` 或 `/plugin validate .` 来检查问题。常见错误：

错误	原因	解决方案
<code>File not found: .claude-plugin/marketplace.json</code>	缺少 manifest	使用必需字段创建 <code>.claude-plugin/marketplace.json</code>
<code>Invalid JSON syntax: Unexpected token...</code>	JSON 语法错误	检查缺少的逗号、多余的逗号或未引用的字符串
<code>Duplicate plugin name "x" found in marketplace</code>	两个 plugins 共享相同的名称	给每个 plugin 一个唯一的 <code>name</code> 值
<code>plugins[0].source: Path traversal not allowed</code>	源路径包含 <code>..</code>	使用相对于 marketplace 根目录的路径，不包含 <code>..</code>

警告（非阻止）：

- `Marketplace has no plugins defined`：将至少一个 plugin 添加到 `plugins` 数组
- `No marketplace description provided`：添加 `metadata.description` 以帮助用户理解你的 marketplace

Plugin 安装失败

症状：Marketplace 出现但 plugin 安装失败

解决方案：

- 验证 plugin 源 URL 是否可访问
- 检查 plugin 目录是否包含必需的文件
- 对于 GitHub 源，确保存储库是公开的或你有访问权限
- 通过手动克隆/下载来测试 plugin 源

私有存储库身份验证失败

症状：从私有存储库安装 plugins 时出现身份验证错误

解决方案：

对于手动安装和更新：

- 验证你已使用你的 git 提供商进行身份验证（例如，对于 GitHub 运行 `gh auth status`）
- 检查你的凭证助手是否配置正确：`git config --global credential.helper`
- 尝试手动克隆存储库以验证你的凭证有效

对于后台自动更新：

- 在你的环境中设置适当的令牌：`echo $GITHUB_TOKEN`
- 检查令牌是否具有所需的权限（对存储库的读取访问权限）
- 对于 GitHub，确保令牌对私有存储库具有 `repo` 范围
- 对于 GitLab，确保令牌至少具有 `read_repository` 范围
- 验证令牌未过期

Git 操作超时

症状：Plugin 安装或 marketplace 更新失败，出现超时错误，如” Git clone timed out after 120s” 或” Git pull timed out after 120s”。

原因：Claude Code 对所有 git 操作使用 120 秒超时，包括克隆 plugin 存储库和拉取 marketplace 更新。大型存储库或缓慢的网络连接可能超过此限制。

解决方案：使用 `CLAUDE_CODE_PLUGIN_GIT_TIMEOUT_MS` 环境变量增加超时。该值以毫秒为单位：

```
export CLAUDE_CODE_PLUGIN_GIT_TIMEOUT_MS=300000 # 5 minutes
```

相对路径 Plugins 在基于 URL 的 marketplaces 中失败

症状：通过 URL（如 `https://example.com/marketplace.json`）添加了 marketplace，但具有相对路径源（如 `../plugins/my-plugin`）的 plugins 无法安装，出现“path not found”错误。

原因：基于 URL 的 marketplaces 仅下载 `marketplace.json` 文件本身。它们不从服务器下载 plugin 文件。marketplace 条目中的相对路径引用远程服务器上未下载的文件。

解决方案：

- **使用外部源：**将 plugin 条目更改为使用 GitHub、npm 或 git URL 源而不是相对路径：

```
{ "name": "my-plugin", "source": { "source": "github", "repo": "owner/repo" } }
```

- **使用基于 Git 的 marketplace：**在 Git 存储库中托管你的 marketplace 并使用 git URL 添加它。基于 Git 的 marketplaces 克隆整个存储库，使相对路径有效。

安装后文件未找到

症状：Plugin 安装但对文件的引用失败，特别是 plugin 目录外的文件

原因：Plugins 被复制到缓存目录而不是就地使用。引用 plugin 目录外文件的路径（如 `../shared-utils`）不会工作，因为这些文件不会被复制。

解决方案：见 [Plugin 缓存和文件解析](#) 了解解决方法，包括符号链接和目录重组。

有关其他调试工具和常见问题，请参阅[调试和开发工具](#)。

另见

- [发现和安装预构建的 plugins](#) - 从现有 marketplaces 安装 plugins
- [Plugins](#) - 创建你自己的 plugins
- [Plugins 参考](#) - 完整的技术规范和架构
- [Plugin 设置](#) - Plugin 配置选项
- [strictKnownMarketplaces 参考](#) - 托管 marketplace 限制

通过市场发现和安装预构建插件

从市场发现和安装插件，以使用新命令、代理和功能扩展 Claude Code。

插件通过 skills、agents、hooks 和 MCP servers 扩展 Claude Code。插件市场是帮助您发现和安装这些扩展的目录，无需自己构建。

想要创建和分发自己的市场？请参阅[创建和分发插件市场](#)。

市场如何工作

市场是他人创建和共享的插件目录。使用市场是一个两步过程：

Step 1: 添加市场

这会向 Claude Code 注册目录，以便您可以浏览可用内容。尚未安装任何插件。

Step 2: 安装单个插件

浏览目录并安装您想要的插件。

可以将其视为添加应用商店：添加商店让您可以访问浏览其集合，但您仍然需要单独选择要下载的应用。

官方 Anthropic 市场

官方 Anthropic 市场（`claude-plugins-official`）在您启动 Claude Code 时自动可用。运行 `/plugin` 并转到发现选项卡以浏览可用内容。

要从官方市场安装插件：

```
/plugin install plugin-name@claude-plugins-official
```

Note:

官方市场由 Anthropic 维护。要向官方市场提交插件，请使用应用内提交表单之一：

- **Claude.ai:** claude.ai/settings/plugins/submit
- **Console:** platform.claude.com/plugins/submit

要独立分发插件，请[创建您自己的市场](#)并与用户共享。

官方市场包括多个插件类别：

代码智能

代码智能插件启用 Claude Code 的内置 LSP 工具，使 Claude 能够跳转到定义、查找引用并在编辑后立即查看类型错误。这些插件配置[语言服务器协议](#)连接，这是为 VS Code 代码智能提供支持的相同技术。

这些插件需要在您的系统上安装语言服务器二进制文件。如果您已经安装了语言服务器，当您打开项目时，Claude 可能会提示您安装相应的插件。

语言	插件	所需二进制文件
C/C++	clangd-lsp	clangd
C#	csharp-lsp	csharp-ls
Go	gopls-lsp	gopls
Java	jdtls-lsp	jdtls
Kotlin	kotlin-lsp	kotlin-language-server
Lua	lua-lsp	lua-language-server
PHP	php-lsp	intelephense
Python	pyright-lsp	pyright-langserver
Rust	rust-analyzer-lsp	rust-analyzer
Swift	swift-lsp	sourcekit-lsp
TypeScript	typescript-lsp	typescript-language-server

您也可以[为其他语言创建自己的 LSP 插件](#)。

Note:

如果在安装插件后在 `/plugin` 错误选项卡中看到 `Executable not found in $PATH`，请从上表安装所需的二进制文件。

Claude 从代码智能插件获得的功能

安装代码智能插件并且其语言服务器二进制文件可用后，Claude 获得两项功能：

- **自动诊断：**在 Claude 进行的每次文件编辑后，语言服务器分析更改并自动报告错误和警告。Claude 看到类型错误、缺失导入和语法问题，无需运行编译器或 linter。如果 Claude 引入错误，它会注意到并在同一轮中修复问题。这不需要除安装插件外的任何配置。当“发现诊断”指示器出现时，您可以按 **Ctrl+O** 来内联查看诊断。
- **代码导航：**Claude 可以使用语言服务器跳转到定义、查找引用、获取悬停时的类型信息、列出符号、查找实现和追踪调用层次结构。这些操作为 Claude 提供比基于 grep 的搜索更精确的导航，尽管可用性可能因语言和环境而异。

如果遇到问题，请参阅[代码智能故障排除](#)。

外部集成

这些插件捆绑预配置的 [MCP servers](#)，以便您可以连接 Claude 到外部服务，无需手动设置：

- **源代码控制：** [github](#) 、 [gitlab](#)
- **项目管理：** [atlassian](#)（Jira/Confluence） 、 [asana](#) 、 [linear](#) 、 [notion](#)
- **设计：** [figma](#)
- **基础设施：** [vercel](#) 、 [firebase](#) 、 [supabase](#)
- **通信：** [slack](#)
- **监控：** [sentry](#)

开发工作流

为常见开发任务添加命令和代理的插件：

- **commit-commands：**Git 提交工作流，包括提交、推送和 PR 创建
- **pr-review-toolkit：**用于审查拉取请求的专门代理
- **agent-sdk-dev：**使用 Claude Agent SDK 构建的工具
- **plugin-dev：**用于创建您自己的插件的工具包

输出样式

自定义 Claude 的响应方式：

- **explanatory-output-style：**关于实现选择的教育见解
- **learning-output-style：**用于技能构建的交互式学习模式

尝试：添加演示市场

Anthropic 还维护一个[演示插件市场](#)（`claude-code-plugins`），其中包含展示插件系统可能性的示例插件。与官方市场不同，您需要手动添加此市场。

Step 1: 添加市场

在 Claude Code 中，为 `anthropics/claude-code` 市场运行 `plugin marketplace add` 命令：

```
/plugin marketplace add anthropics/claude-code
```

这会下载市场目录并使其插件对您可用。

Step 2: 浏览可用插件

运行 `/plugin` 打开插件管理器。这会打开一个选项卡式界面，有四个选项卡，您可以使用 **Tab** 循环切换（或使用 **Shift+Tab** 向后切换）：

- **发现**：从所有市场浏览可用插件
- **已安装**：查看和管理已安装的插件
- **市场**：添加、删除或更新已添加的市场
- **错误**：查看任何插件加载错误

转到**发现**选项卡以查看您刚添加的市场中的插件。

Step 3: 安装插件

选择一个插件以查看其详细信息，然后选择安装范围：

- **用户范围**：在所有项目中为自己安装
- **项目范围**：为此存储库上的所有协作者安装
- **本地范围**：仅在此存储库中为自己安装

例如，选择 **commit-commands**（添加 git 工作流命令的插件）并将其安装到您的用户范围。

您也可以从命令行直接安装：

```
/plugin install commit-commands@anthropics-claude-code
```

请参阅[配置范围](#)以了解有关范围的更多信息。

Step 4: 使用您的新插件

安装后，插件的命令立即可用。插件命令由插件名称命名空间，因此 **commit-commands** 提供诸如 `/commit-commands:commit` 之类的命令。

通过对文件进行更改并运行来尝试：

```
/commit-commands:commit
```

这会暂存您的更改、生成提交消息并创建提交。

每个插件的工作方式不同。检查**发现**选项卡中的插件描述或其主页以了解它提供的命令和功能。

本指南的其余部分涵盖了添加市场、安装插件和管理配置的所有方式。

添加市场

使用 `/plugin marketplace add` 命令从不同来源添加市场。

Tip:

快捷方式：您可以使用 `/plugin market` 代替 `/plugin marketplace`，以及使用 `rm` 代替 `remove`。

- **GitHub 存储库：** `owner/repo` 格式（例如，`anthropics/claude-code`）
- **Git URL：** 任何 git 存储库 URL（GitLab、Bitbucket、自托管）
- **本地路径：** 目录或 `marketplace.json` 文件的直接路径
- **远程 URL：** 托管 `marketplace.json` 文件的直接 URL

从 GitHub 添加

使用 `owner/repo` 格式添加包含 `.claude-plugin/marketplace.json` 文件的 GitHub 存储库，其中 `owner` 是 GitHub 用户名或组织，`repo` 是存储库名称。

例如，`anthropics/claude-code` 指的是由 `anthropics` 拥有的 `claude-code` 存储库：

```
/plugin marketplace add anthropics/claude-code
```

从其他 Git 主机添加

通过提供完整 URL 添加任何 git 存储库。这适用于任何 Git 主机，包括 GitLab、Bitbucket 和自托管服务器：

使用 HTTPS：

```
/plugin marketplace add https://gitlab.com/company/plugins.git
```

使用 SSH：

```
/plugin marketplace add git@gitlab.com:company/plugins.git
```

要添加特定分支或标签，请在 **#** 后附加 ref：

```
/plugin marketplace add https://gitlab.com/company/plugins.git#v1.0.0
```

从本地路径添加

添加包含 `.claude-plugin/marketplace.json` 文件的本地目录：

```
/plugin marketplace add ./my-marketplace
```

您也可以添加 `marketplace.json` 文件的直接路径：

```
/plugin marketplace add ./path/to/marketplace.json
```

从远程 URL 添加

通过 URL 添加远程 `marketplace.json` 文件：

```
/plugin marketplace add https://example.com/marketplace.json
```

Note:

与基于 Git 的市场相比，基于 URL 的市场有一些限制。如果在安装插件时遇到“路径未找到”错误，请参阅[故障排除](#)。

安装插件

添加市场后，您可以直接安装插件（默认安装到用户范围）：

```
/plugin install plugin-name@marketplace-name
```

要选择不同的[安装范围](#)，请使用交互式 UI：运行 `/plugin`，转到[发现](#)选项卡，然后在插件上按 **Enter**。您将看到以下选项：

- **用户范围**（默认）：在所有项目中为自己安装
- **项目范围**：为此存储库上的所有协作者安装（添加到 `.claude/settings.json`）
- **本地范围**：仅在此存储库中为自己安装（不与协作者共享）

您也可能看到具有[托管](#)范围的插件——这些由管理员通过[托管设置](#)安装，无法修改。

运行 `/plugin` 并转到[已安装](#)选项卡以查看按范围分组的插件。

Warning:

在安装插件之前，请确保您信任该插件。Anthropic 不控制插件中包含的 MCP servers、文件或其他软件，也无法验证它们是否按预期工作。检查每个插件的主页以获取更多信息。

管理已安装的插件

运行 `/plugin` 并转到[已安装](#)选项卡以查看、启用、禁用或卸载您的插件。键入以按插件名称或描述筛选列表。

您也可以使用直接命令管理插件。

禁用插件而不卸载：

```
/plugin disable plugin-name@marketplace-name
```

重新启用已禁用的插件：

```
/plugin enable plugin-name@marketplace-name
```

完全删除插件：

```
/plugin uninstall plugin-name@marketplace-name
```

`--scope` 选项允许您使用 CLI 命令针对特定范围：

```
claude plugin install formatter@your-org --scope project
claude plugin uninstall formatter@your-org --scope project
```

应用插件更改而不重启

当您在会话期间安装、启用或禁用插件时，某些更改（如新命令和 hooks）会立即生效。其他更改（包括 LSP 服务器更新）需要重启。

要激活所有待处理的插件更改而不重启，请运行：

```
/reload-plugins
```

Claude Code 重新加载所有活跃插件并报告加载的内容。如果添加或更新了任何 LSP 服务器，它会让您知道这些需要重启才能生效。

管理市场

您可以通过交互式 `/plugin` 界面或 CLI 命令管理市场。

使用交互式界面

运行 `/plugin` 并转到**市场**选项卡以：

- 查看所有已添加的市场及其来源和状态
- 添加新市场
- 更新市场列表以获取最新插件
- 删除您不再需要的市场

使用 CLI 命令

您也可以使用直接命令管理市场。

列出所有配置的市场：

```
/plugin marketplace list
```

刷新市场的插件列表：

```
/plugin marketplace update marketplace-name
```

删除市场：

```
/plugin marketplace remove marketplace-name
```

Warning:

删除市场将卸载您从中安装的任何插件。

配置自动更新

Claude Code 可以在启动时自动更新市场及其已安装的插件。为市场启用自动更新后，Claude Code 会刷新市场数据并将已安装的插件更新到最新版本。如果任何插件已更新，您将看到提示您运行 `/reload-plugins` 的通知。

通过 UI 为单个市场切换自动更新：

1. 运行 `/plugin` 打开插件管理器
2. 选择**市场**
3. 从列表中选择市场
4. 选择**启用自动更新**或**禁用自动更新**

官方 Anthropic 市场默认启用自动更新。第三方和本地开发市场默认禁用自动更新。

要完全禁用 Claude Code 和所有插件的所有自动更新，请设置 `DISABLE_AUTOUPDATER` 环境变量。有关详细信息，请参阅[自动更新](#)。

要在禁用 Claude Code 自动更新的同时保持插件自动更新启用，请设置

`FORCE_AUTOUPDATE_PLUGINS=true` 以及 `DISABLE_AUTOUPDATER`：

```
export DISABLE_AUTOUPDATER=true
export FORCE_AUTOUPDATE_PLUGINS=true
```

当您想手动管理 Claude Code 更新但仍接收自动插件更新时，这很有用。

配置团队市场

团队管理员可以通过将市场配置添加到 `.claude/settings.json` 来为项目设置自动市场安装。当团队成员信任存储库文件夹时，Claude Code 会提示他们安装这些市场和插件。

将 `extraKnownMarketplaces` 添加到您项目的 `.claude/settings.json`：

```
{
  "extraKnownMarketplaces": {
    "my-team-tools": {
      "source": {
        "source": "github",
        "repo": "your-org/claude-plugins"
      }
    }
  }
}
```

有关完整配置选项（包括 `extraKnownMarketplaces` 和 `enabledPlugins`），请参阅[插件设置](#)。

安全性

插件和市场是高度受信任的组件，可以使用您的用户权限在您的机器上执行任意代码。仅从您信任的来源安装插件和添加市场。组织可以使用[托管市场限制](#)限制用户允许添加的市场。

故障排除

/plugin 命令无法识别

如果您看到“未知命令”或 `/plugin` 命令未出现：

1. **检查您的版本：**运行 `claude --version`。插件需要版本 1.0.33 或更高版本。
2. **更新 Claude Code：**
 - **Homebrew：** `brew upgrade claude-code`
 - **npm：** `npm update -g @anthropic-ai/claude-code`
 - **本地安装程序：**从[设置](#)重新运行安装命令
3. **重启 Claude Code：**更新后，重启您的终端并再次运行 `claude`。

常见问题

- **市场未加载：**验证 URL 是否可访问以及 `.claude-plugin/marketplace.json` 是否存在于该路径
- **插件安装失败：**检查插件源 URL 是否可访问以及存储库是否公开（或您有访问权限）
- **安装后找不到文件：**插件被复制到缓存，因此引用插件目录外文件的路径将不起作用

- **插件 skills 未出现：**使用 `rm -rf ~/.claude/plugins/cache` 清除缓存，重启 Claude Code，然后重新安装插件。

有关详细的故障排除和解决方案，请参阅市场指南中的[故障排除](#)。有关调试工具，请参阅[调试和开发工具](#)。

代码智能问题

- **语言服务器未启动：**验证二进制文件已安装且在您的 `$PATH` 中可用。检查 `/plugin` 错误选项卡以获取详细信息。
- **高内存使用：**`rust-analyzer` 和 `pyright` 等语言服务器在大型项目上可能消耗大量内存。如果您遇到内存问题，请使用 `/plugin disable <plugin-name>` 禁用插件，并改为依赖 Claude 的内置搜索工具。
- **monorepos 中的误报诊断：**如果工作区配置不正确，语言服务器可能会报告内部包的未解析导入错误。这些不会影响 Claude 编辑代码的能力。

后续步骤

- **构建您自己的插件：**请参阅[插件](#)以创建 skills、agents 和 hooks
- **创建市场：**请参阅[创建插件市场](#)以将插件分发给您的团队或社区
- **技术参考：**请参阅[插件参考](#)以获取完整规范

Part 6: Advanced & Automation

创建自定义 subagents

在 Claude Code 中创建和使用专门的 AI subagents，用于特定任务的工作流和改进的上下文管理。

Subagents 是处理特定类型任务的专门 AI 助手。每个 subagent 在自己的 context window 中运行，具有自定义系统提示、特定的工具访问权限和独立的权限。当 Claude 遇到与 subagent 描述相匹配的任务时，它会委托给该 subagent，该 subagent 独立工作并返回结果。

Note:

如果您需要多个代理并行工作并相互通信，请参阅 [agent teams](#)。Subagents 在单个会话中工作；agent teams 跨多个会话进行协调。

Subagents 可以帮助您：

- **保留上下文**，通过将探索和实现保持在主对话之外
- **强制执行约束**，通过限制 subagent 可以使用的工具
- **跨项目重用配置**，使用用户级 subagents
- **专门化行为**，为特定领域使用专注的系统提示
- **控制成本**，通过将任务路由到更快、更便宜的模型（如 Haiku）

Claude 使用每个 subagent 的描述来决定何时委托任务。创建 subagent 时，请编写清晰的描述，以便 Claude 知道何时使用它。

Claude Code 包括几个内置 subagents，如 **Explore**、**Plan** 和 **general-purpose**。您也可以创建自定义 subagents 来处理特定任务。本页涵盖 [内置 subagents](#)、[如何创建您自己的](#)、[完整配置选项](#)、[使用 subagents 的模式](#) 和 [示例 subagents](#)。

内置 subagents

Claude Code 包括内置 subagents，Claude 在适当时会自动使用。每个都继承父对话的权限，并有额外的工具限制。

Explore

一个快速的、只读的代理，针对搜索和分析代码库进行了优化。

- **模型**：Haiku（快速、低延迟）
- **工具**：只读工具（拒绝访问 Write 和 Edit 工具）
- **目的**：文件发现、代码搜索、代码库探索

当 Claude 需要搜索或理解代码库而不进行更改时，它会委托给 Explore。这样可以将探索结果保持在主对话上下文之外。

调用 Explore 时，Claude 指定一个彻底程度级别：**quick** 用于有针对性的查找，**medium** 用于平衡的探索，或 **very thorough** 用于全面分析。

Plan

在 [plan mode](#) 期间用于在呈现计划之前收集上下文的研究代理。

- **模型**：继承自主对话
- **工具**：只读工具（拒绝访问 Write 和 Edit 工具）
- **目的**：用于规划的代码库研究

当您处于 plan mode 并且 Claude 需要理解您的代码库时，它会将研究委托给 Plan subagent。这可以防止无限嵌套（subagents 无法生成其他 subagents），同时仍然收集必要的上下文。

General-purpose

一个能够处理复杂、多步骤任务的代理，这些任务需要探索和操作。

- **模型**：继承自主对话
- **工具**：所有工具
- **目的**：复杂研究、多步骤操作、代码修改

当任务需要探索和修改、复杂推理来解释结果或多个相关步骤时，Claude 会委托给 general-purpose。

Other

Claude Code 包括用于特定任务的其他辅助代理。这些通常会自动调用，因此您不需要直接使用它们。

代理	模型	Claude 何时使用它
Bash	继承	在单独的上下文中运行终端命令

代理	模型	Claude 何时使用它
statusline-setup	Sonnet	当您运行 <code>/statusline</code> 来配置您的状态行时
Claude Code Guide	Haiku	当您提出关于 Claude Code 功能的问题时

除了这些内置 subagents，您可以创建自己的，具有自定义提示、工具限制、权限模式、hooks 和 skills。以下部分展示了如何开始和自定义 subagents。

快速入门：创建您的第一个 subagent

Subagents 在带有 YAML frontmatter 的 Markdown 文件中定义。您可以 [手动创建它们](#) 或使用 `/agents` 命令。

本演练指导您使用 `/agent` 命令创建用户级 subagent。该 subagent 审查代码并为代码库建议改进。

Step 1: 打开 subagents 界面

在 Claude Code 中，运行：

```
/agents
```

Step 2: 创建新的用户级代理

选择 **Create new agent**，然后选择 **User-level**。这会将 subagent 保存到 `~/.claude/agents/`，以便在所有项目中可用。

Step 3: 使用 Claude 生成

选择 **Generate with Claude**。出现提示时，描述 subagent：

```
A code improvement agent that scans files and suggests improvements for readability, performance, and best practices. It should explain each issue, show the current code, and provide an improved version.
```

Claude 生成系统提示和配置。如果您想自定义它，请按 `e` 在编辑器中打开它。

Step 4: 选择工具

对于只读审查者，取消选择除 **Read-only tools** 之外的所有内容。如果您保持所有工具被选中，subagent 会继承主对话可用的所有工具。

Step 5: 选择模型

选择 subagent 使用的模型。对于此示例代理，选择 **Sonnet**，它在分析代码模式的能力和速度之间取得平衡。

Step 6: 选择颜色

为 subagent 选择背景颜色。这可以帮助您在 UI 中识别哪个 subagent 正在运行。

Step 7: 保存并尝试

保存 subagent。它立即可用（无需重启）。尝试它：

```
Use the code-improver agent to suggest improvements in this project
```

Claude 委托给您的新 subagent，它扫描代码库并返回改进建议。

现在您有了一个 subagent，可以在机器上的任何项目中使用它来分析代码库并建议改进。

您也可以手动创建 subagents 作为 Markdown 文件、通过 CLI 标志定义它们，或通过插件分发它们。以下部分涵盖所有配置选项。

配置 subagents

使用 `/agents` 命令

`/agents` 命令提供了一个交互式界面来管理 subagents。运行 `/agents` 来：

- 查看所有可用的 subagents（内置、用户、项目和插件）
- 使用引导式设置或 Claude 生成创建新 subagents
- 编辑现有 subagent 配置和工具访问
- 删除自定义 subagents
- 查看当存在重复项时哪些 subagents 处于活动状态

这是创建和管理 subagents 的推荐方式。对于手动创建或自动化，您也可以直接添加 subagent 文件。

要从命令行列出所有配置的 subagents 而不启动交互式会话，请运行 `claude agents`。这显示按来源分组的代理，并指示哪些被更高优先级的定义覆盖。

选择 subagent 范围

Subagents 是带有 YAML frontmatter 的 Markdown 文件。根据范围将它们存储在不同的位置。当多个 subagents 共享相同的名称时，更高优先级的位置获胜。

位置	范围	优先级	如何创建
<code>--agents</code> CLI 标志	当前会话	1（最高）	启动 Claude Code 时传递 JSON
<code>.claude/agents/</code>	当前项目	2	交互式或手动
<code>~/.claude/agents/</code>	所有项目	3	交互式或手动
插件的 <code>agents/</code> 目录	启用插件的位置	4（最低）	与 plugins 一起安装

项目 subagents (`.claude/agents/`) 非常适合特定于代码库的 subagents。将它们检入版本控制，以便您的团队可以协作使用和改进它们。

用户 subagents (`~/.claude/agents/`) 是在所有项目中可用的个人 subagents。

CLI 定义的 subagents 在启动 Claude Code 时作为 JSON 传递。它们仅存在于该会话中，不会保存到磁盘，使其对快速测试或自动化脚本很有用：

```
claude --agents '{
  "code-reviewer": {
    "description": "Expert code reviewer. Use proactively after code changes.",
    "prompt": "You are a senior code reviewer. Focus on code quality, security,
and best practices.",
    "tools": ["Read", "Grep", "Glob", "Bash"],
    "model": "sonnet"
  }
}'
```

`--agents` 标志接受与基于文件的 subagents 相同的 `frontmatter` 字段的 JSON：`description`、`prompt`、`tools`、`disallowedTools`、`model`、`permissionMode`、`mcpServers`、`hooks`、`maxTurns`、`skills` 和 `memory`。对系统提示使用 `prompt`，等同于基于文件的 subagents 中的 markdown 正文。有关完整的 JSON 格式，请参阅 [CLI 参考](#)。

插件 subagents 来自您已安装的 [plugins](#)。它们与您的自定义 subagents 一起出现在 `/agents` 中。有关创建插件 subagents 的详细信息，请参阅 [插件组件参考](#)。

编写 subagent 文件

Subagent 文件使用 YAML frontmatter 进行配置，后跟 Markdown 中的系统提示：

Note:

Subagents 在会话启动时加载。如果您通过手动添加文件来创建 subagent，请重启会话或使用 `/agents` 立即加载它。

```
---
name: code-reviewer
description: Reviews code for quality and best practices
tools: Read, Glob, Grep
model: sonnet
---

You are a code reviewer. When invoked, analyze the code and provide
specific, actionable feedback on quality, security, and best practices.
```

Frontmatter 定义 subagent 的元数据和配置。正文成为指导 subagent 行为的系统提示。Subagents 仅接收此系统提示（加上基本环境详细信息，如工作目录），而不是完整的 Claude Code 系统提示。

支持的 frontmatter 字段

以下字段可以在 YAML frontmatter 中使用。仅 `name` 和 `description` 是必需的。

字段	必需	描述
<code>name</code>	是	使用小写字母和连字符的唯一标识符
<code>description</code>	是	Claude 何时应委托给此 subagent
<code>tools</code>	否	subagent 可以使用的 工具 。如果省略，继承所有工具
<code>disallowedTools</code>	否	要拒绝的工具，从继承或指定的列表中删除

字段	必需	描述
<code>model</code>	否	要使用的 模型 ： <code>sonnet</code> 、 <code>opus</code> 、 <code>haiku</code> 或 <code>inherit</code> 。默 认为 <code>inherit</code>
<code>permissionMode</code>	否	权限模式 ： <code>default</code> 、 <code>ac</code> <code>ceptEdits</code> 、 <code>dontAsk</code> 、 <code>bypassPermissions</code> 或 <code>plan</code>
<code>maxTurns</code>	否	subagent 停止前的最大代 理轮数
<code>skills</code>	否	在启动时加载到 subagent 上下文中的 Skills 。注入完 整的技能内容，而不仅仅是 可用于调用。Subagents 不继承来自父对话的技能
<code>mcpServers</code>	否	此 subagent 可用的 MCP servers 。每个条目要么是 引用已配置服务器的服务器 名称（例如 <code>"slack"</code> ）， 要么是内联定义，其中服务 器名称为键，完整的 MCP 服务器配置 为值
<code>hooks</code>	否	限定于此 subagent 的 生命 周期 hooks
<code>memory</code>	否	持久内存范围 ： <code>user</code> 、 <code>project</code> 或 <code>local</code> 。启 用跨会话学习
<code>background</code>	否	设置为 <code>true</code> 以始终将此 subagent 作为 后台任务 运 行。默认值： <code>false</code>

字段	必需	描述
<code>isolation</code>	否	设置为 <code>worktree</code> 以在临时 <code>git worktree</code> 中运行 subagent，为其提供存储库的隔离副本。如果 subagent 不进行任何更改， <code>worktree</code> 会自动清理

选择模型

`model` 字段控制 subagent 使用的 [AI 模型](#)：

- **模型别名**：使用可用的别名之一：`sonnet`、`opus` 或 `haiku`
- **inherit**：使用与主对话相同的模型
- **省略**：如果未指定，默认为 `inherit`（使用与主对话相同的模型）

控制 subagent 功能

您可以通过工具访问、权限模式和条件规则来控制 subagents 可以做什么。

可用工具

Subagents 可以使用 Claude Code 的任何 [内部工具](#)。默认情况下，subagents 继承主对话的所有工具，包括 MCP 工具。

要限制工具，请使用 `tools` 字段（允许列表）或 `disallowedTools` 字段（拒绝列表）：

```
---
name: safe-researcher
description: Research agent with restricted capabilities
tools: Read, Grep, Glob, Bash
disallowedTools: Write, Edit
---
```

限制可以生成的 subagents

当代理作为主线程运行时，使用 `claude --agent`，它可以使用 Agent 工具生成 subagents。要限制它可以生成的 subagent 类型，请在 `tools` 字段中使用 `Agent(agent_type)` 语法。

Note:

在版本 2.1.63 中，Task 工具被重命名为 Agent。设置和代理定义中的现有 `Task(...)` 引用仍然作为别名工作。

```
---
name: coordinator
description: Coordinates work across specialized agents
tools: Agent(worker, researcher), Read, Bash
---
```

这是一个允许列表：只有 `worker` 和 `researcher` subagents 可以被生成。如果代理尝试生成任何其他类型，请求失败，代理在其提示中仅看到允许的类型。要在允许所有其他类型的同时阻止特定代理，请改用 `permissions.deny`。

要允许生成任何 subagent 而不受限制，请使用不带括号的 `Agent`：

```
tools: Agent, Read, Bash
```

如果 `Agent` 完全从 `tools` 列表中省略，代理无法生成任何 subagents。此限制仅适用于作为主线程运行的代理，使用 `claude --agent`。Subagents 无法生成其他 subagents，因此 `Agent(agent_type)` 在 subagent 定义中无效。

权限模式

`permissionMode` 字段控制 subagent 如何处理权限提示。Subagents 继承主对话的权限上下文，但可以覆盖模式。

模式	行为
<code>default</code>	标准权限检查，带有提示
<code>acceptEdits</code>	自动接受文件编辑
<code>dontAsk</code>	自动拒绝权限提示（显式允许的工具仍然有效）
<code>bypassPermissions</code>	跳过所有权限检查
<code>plan</code>	Plan mode（只读探索）

Warning:

谨慎使用 `bypassPermissions`。它跳过所有权限检查，允许 subagent 执行任何操作而无需批准。

如果父级使用 `bypassPermissions`，这优先并且无法被覆盖。

将 skills 预加载到 subagents

使用 `skills` 字段在启动时将 skill 内容注入到 subagent 的上下文中。这为 subagent 提供领域知识，而无需在执行期间发现和加载 skills。

```
---
name: api-developer
description: Implement API endpoints following team conventions
skills:
  - api-conventions
  - error-handling-patterns
---

Implement API endpoints. Follow the conventions and patterns from the preloaded skills.
```

每个 skill 的完整内容被注入到 subagent 的上下文中，而不仅仅是可用于调用。Subagents 不继承来自父对话的 skills；您必须明确列出它们。

Note:

这与 [在 subagent 中运行 skill](#) 相反。使用 subagent 中的 `skills`，subagent 控制系统提示并加载 skill 内容。使用 skill 中的 `context: fork`，skill 内容被注入到您指定的代理中。两者都使用相同的底层系统。

启用持久内存

`memory` 字段为 subagent 提供一个在对话中存活的持久目录。Subagent 使用此目录随时间积累知识，例如代码库模式、调试见解和架构决策。

```
---
name: code-reviewer
description: Reviews code for quality and best practices
memory: user
---
```

You are a code reviewer. As you review code, update your agent memory with patterns, conventions, and recurring issues you discover.

根据内存应该应用的广泛程度选择范围：

范围	位置	何时使用
user	~/.claude/agent-memory/<name-of-agent>/	subagent 应该在所有项目中记住学习
project	.claude/agent-memory/<name-of-agent>/	subagent 的知识是特定于项目的并可通过版本控制共享
local	.claude/agent-memory-local/<name-of-agent>/	subagent 的知识是特定于项目的但不应检入版本控制

启用内存时：

- subagent 的系统提示包括读取和写入内存目录的说明。
- subagent 的系统提示还包括内存目录中 MEMORY.md 的前 200 行，以及如果超过 200 行则策划 MEMORY.md 的说明。
- Read、Write 和 Edit 工具会自动启用，以便 subagent 可以管理其内存文件。

持久内存提示

- user 是推荐的默认范围。当 subagent 的知识仅与特定代码库相关时，使用 project 或 local。
- 要求 subagent 在开始工作前查阅其内存：“Review this PR, and check your memory for patterns you’ve seen before.”
- 要求 subagent 在完成任务后更新其内存：“Now that you’re done, save what you learned to your memory.” 随着时间的推移，这会建立一个知识库，使 subagent 更有效。
- 直接在 subagent 的 markdown 文件中包含内存说明，以便它主动维护自己的知识库：

Update your agent memory as you discover codepaths, patterns, library locations, and key architectural decisions. This builds up institutional knowledge across conversations. Write concise notes about what you found and where.

使用 hooks 的条件规则

为了更动态地控制工具使用，请使用 `PreToolUse` hooks 在执行前验证操作。当您需要允许工具的某些操作同时阻止其他操作时，这很有用。

此示例创建一个仅允许只读数据库查询的 subagent。 `PreToolUse` hook 在每个 Bash 命令执行前运行 `command` 中指定的脚本：

```
---
name: db-reader
description: Execute read-only database queries
tools: Bash
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/validate-readonly-query.sh"
---
```

Claude Code [通过 stdin 将 hook 输入作为 JSON 传递](#) 给 hook 命令。验证脚本读取此 JSON，提取 Bash 命令，并 [以代码 2 退出](#) 以阻止写入操作：

```
#!/bin/bash

## ./scripts/validate-readonly-query.sh

INPUT=$(cat)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // empty')

## Block SQL write operations (case-insensitive)
if echo "$COMMAND" | grep -iE '\b(INSERT|UPDATE|DELETE|DROP|CREATE|ALTER|TRUNCATE)\b' > /dev/null; then
    echo "Blocked: Only SELECT queries are allowed" >&2
    exit 2
fi

exit 0
```

有关完整的输入架构，请参阅 [Hook input](#)，有关退出代码如何影响行为，请参阅 [exit codes](#)。

禁用特定 subagents

您可以通过将 subagents 添加到 [settings](#) 中的 `permissions.deny` 数组来防止 Claude 使用特定 subagents。使用格式 `Agent(subagent-name)`，其中 `subagent-name` 与 subagent 的 `name` 字段匹配。

```
{
  "permissions": {
    "deny": ["Agent(Explore)", "Agent(my-custom-agent)"]
  }
}
```

这适用于内置和自定义 subagents。您也可以使用 `--disallowedTools` CLI 标志：

```
claude --disallowedTools "Agent(Explore)"
```

有关权限规则的更多详细信息，请参阅 [Permissions 文档](#)。

为 subagents 定义 hooks

Subagents 可以定义在 subagent 生命周期期间运行的 [hooks](#)。有两种方式来配置 hooks：

- 1. 在 **subagent 的 frontmatter 中**：定义仅在该 subagent 活动时运行的 hooks
- 2. 在 **settings.json 中**：定义在 subagents 启动或停止时在主会话中运行的 hooks

Subagent frontmatter 中的 Hooks

直接在 subagent 的 markdown 文件中定义 hooks。这些 hooks 仅在该特定 subagent 活动时运行，并在其完成时清理。

所有 [hook 事件](#) 都受支持。subagents 最常见的事件是：

事件	匹配器输入	何时触发
PreToolUse	工具名称	在 subagent 使用工具之前
PostToolUse	工具名称	在 subagent 使用工具之后
Stop	(无)	当 subagent 完成时（在运行时转换为 SubagentStop）

此示例使用 **PreToolUse** hook 验证 Bash 命令，并使用 **PostToolUse** 在文件编辑后运行 linter：

```
---
name: code-reviewer
description: Review code changes with automatic linting
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/validate-command.sh $TOOL_INPUT"
  PostToolUse:
    - matcher: "Edit|Write"
      hooks:
        - type: command
          command: "./scripts/run-linter.sh"
---
```

Frontmatter 中的 **Stop** hooks 会自动转换为 **SubagentStop** 事件。

用于 subagent 事件的项目级 hooks

在 **settings.json** 中配置 hooks，以响应主会话中的 subagent 生命周期事件。

事件	匹配器输入	何时触发
SubagentStart	代理类型名称	当 subagent 开始执行时
SubagentStop	代理类型名称	当 subagent 完成时

两个事件都支持匹配器以按名称针对特定代理类型。此示例仅在 **db-agent** subagent 启动时运行设置脚本，并在任何 subagent 停止时运行清理脚本：

```

{
  "hooks": {
    "SubagentStart": [
      {
        "matcher": "db-agent",
        "hooks": [
          { "type": "command", "command": "./scripts/setup-db-connection.sh" }
        ]
      }
    ],
    "SubagentStop": [
      {
        "hooks": [
          { "type": "command", "command": "./scripts/cleanup-db-connection.sh" }
        ]
      }
    ]
  }
}

```

有关完整的 hook 配置格式，请参阅 [Hooks](#)。

使用 subagents

理解自动委托

Claude 根据您的请求中的任务描述、subagent 配置中的 `description` 字段和当前上下文自动委托任务。为了鼓励主动委托，在 subagent 的 `description` 字段中包含“use proactively”之类的短语。

您也可以明确请求特定的 subagent：

```

Use the test-runner subagent to fix failing tests
Have the code-reviewer subagent look at my recent changes

```


在前台或后台运行 subagents

Subagents 可以在前台（阻塞）或后台（并发）运行：

- **前台 subagents** 阻塞主对话直到完成。权限提示和澄清问题（如 [AskUserQuestion](#)）会传递给您。
- **后台 subagents** 在您继续工作时并发运行。启动前，Claude Code 会提示您输入 subagent 需要的任何工具权限，确保它具有必要的批准。一旦运行，subagent 继承这些权限并自动拒绝任何未预先批准的内容。如果后台 subagent 需要提出澄清问题，该工具调用失败，但 subagent 继续。

如果后台 subagent 由于缺少权限而失败，您可以 [恢复它](#) 在前台以使用交互式提示重试。

Claude 根据任务决定是否在前台或后台运行 subagents。您也可以：

- 要求 Claude “run this in the background”
- 按 **Ctrl+B** 将运行中的任务放在后台

要禁用所有后台任务功能，请将 `CLAUDE_CODE_DISABLE_BACKGROUND_TASKS` 环境变量设置为 [1](#)。请参阅 [Environment variables](#)。

常见模式

隔离高容量操作

subagents 最有效的用途之一是隔离产生大量输出的操作。运行测试、获取文档或处理日志文件可能会消耗大量上下文。通过将这些委托给 subagent，详细输出保留在 subagent 的上下文中，而只有相关摘要返回到主对话。

```
Use a subagent to run the test suite and report only the failing tests with their
error messages
```

并行运行研究

对于独立调查，生成多个 subagents 以同时工作：

```
Research the authentication, database, and API modules in parallel using separate
subagents
```

每个 subagent 独立探索其区域，然后 Claude 综合这些发现。当研究路径不相互依赖时，这效果最好。

Warning:

当 subagents 完成时，它们的结果返回到主对话。运行许多 subagents，每个都返回详细结果，可能会消耗大量上下文。

对于需要持续并行性或超过上下文窗口的任务，[agent teams](#) 为每个工作者提供自己的独立上下文。

链接 subagents

对于多步骤工作流，要求 Claude 按顺序使用 subagents。每个 subagent 完成其任务并将结果返回给 Claude，然后将相关上下文传递给下一个 subagent。

```
Use the code-reviewer subagent to find performance issues, then use the optimizer subagent to fix them
```

在 subagents 和主对话之间选择

在以下情况下使用 **主对话**：

- 任务需要频繁的来回或迭代细化
- 多个阶段共享重要上下文（规划 → 实现 → 测试）
- 您正在进行快速、有针对性的更改
- 延迟很重要。Subagents 从头开始，可能需要时间来收集上下文

在以下情况下使用 **subagents**：

- 任务产生您不需要在主上下文中的详细输出
- 您想强制执行特定的工具限制或权限
- 工作是自包含的，可以返回摘要

当您想要可重用的提示或在主对话上下文中运行的工作流而不是隔离的 subagent 上下文时，请改为考虑 [Skills](#)。

对于关于对话中已有内容的快速问题，请使用 `/btw` 而不是 subagent。它看到您的完整上下文但没有工具访问权限，答案被丢弃而不是添加到历史记录。

Note:

Subagents 无法生成其他 subagents。如果您的工作流需要嵌套委托，请使用 [Skills](#) 或从主对话 [链接 subagents](#)。

管理 subagent 上下文

恢复 subagents

每个 subagent 调用都会创建一个具有新鲜上下文的新实例。要继续现有 subagent 的工作而不是重新开始，请要求 Claude 恢复它。

恢复的 subagents 保留其完整的对话历史，包括所有以前的工具调用、结果和推理。Subagent 从停止的地方继续，而不是从头开始。

当 subagent 完成时，Claude 接收其代理 ID。要恢复 subagent，请要求 Claude 继续之前的工作：

```
Use the code-reviewer subagent to review the authentication module
[Agent completes]

Continue that code review and now analyze the authorization logic
[Claude resumes the subagent with full context from previous conversation]
```

您也可以要求 Claude 提供代理 ID（如果您想明确引用它），或在 `~/.claude/projects/{project}/{sessionId}/subagents/` 的转录文件中找到 ID。每个转录存储为 `agent-{agentId}.jsonl`。

Subagent 转录独立于主对话持久化：

- **主对话压缩**：当主对话压缩时，subagent 转录不受影响。它们存储在单独的文件中。
- **会话持久性**：Subagent 转录在其会话中持久化。您可以通过恢复同一会话来在重启 Claude Code 后 [恢复 subagent](#)。
- **自动清理**：转录根据 `cleanupPeriodDays` 设置进行清理（默认：30 天）。

自动压缩

Subagents 支持使用与主对话相同的逻辑进行自动压缩。默认情况下，自动压缩在大约 95% 容量时触发。要更早触发压缩，请将 `CLAUDE_AUTOCOMPACT_PCT_OVERRIDE` 设置为较低的百分比（例如 50）。有关详细信息，请参阅 [environment variables](#)。

压缩事件记录在 subagent 转录文件中：

```
{
  "type": "system",
  "subtype": "compact_boundary",
  "compactMetadata": {
    "trigger": "auto",
    "preTokens": 167189
  }
}
```

preTokens 值显示压缩发生前使用了多少令牌。

示例 subagents

这些示例演示了构建 subagents 的有效模式。将它们用作起点，或使用 Claude 生成自定义版本。

Tip:

最佳实践：

- **设计专注的 subagents**：每个 subagent 应该在一个特定任务中表现出色
- **编写详细的描述**：Claude 使用描述来决定何时委托
- **限制工具访问**：仅授予必要的权限以确保安全和专注
- **检入版本控制**：与您的团队共享项目 subagents

代码审查者

一个只读 subagent，审查代码而不修改它。此示例展示了如何设计一个专注的 subagent，具有有限的工具访问权限（无 Edit 或 Write）和详细的提示，指定确切要查找的内容以及如何格式化输出。

```
---  
name: code-reviewer  
description: Expert code review specialist. Proactively reviews code for quality,  
security, and maintainability. Use immediately after writing or modifying code.  
tools: Read, Grep, Glob, Bash  
model: inherit  
---
```

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is clear and readable
- Functions and variables are well-named
- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

调试器

一个可以分析和修复问题的 subagent。与代码审查者不同，这个包括 Edit，因为修复错误需要修改代码。提示提供了从诊断到验证的清晰 workflow。

```
---  
name: debugger  
description: Debugging specialist for errors, test failures, and unexpected  
behavior. Use proactively when encountering any issues.  
tools: Read, Edit, Bash, Grep, Glob  
---
```

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not the symptoms.

数据科学家

一个用于数据分析工作的特定领域 subagent。此示例展示了如何为典型编码任务之外的专门工作流创建 subagents。它明确设置 `model: sonnet` 以进行更强大的分析。

```
---  
name: data-scientist  
description: Data analysis expert for SQL queries, BigQuery operations, and data  
insights. Use proactively for data analysis tasks and queries.  
tools: Bash, Read, Write  
model: sonnet  
---
```

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:

1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

数据库查询验证器

一个允许 Bash 访问但验证命令以仅允许只读 SQL 查询的 subagent。此示例展示了如何使用 `PreToolUse` hooks 进行条件验证，当您需要比 `tools` 字段更精细的控制时。

```

---
name: db-reader
description: Execute read-only database queries. Use when analyzing data or
generating reports.
tools: Bash
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/validate-readonly-query.sh"
---

```

You are a database analyst with read-only access. Execute SELECT queries to answer questions about the data.

When asked to analyze data:

1. Identify which tables contain the relevant data
2. Write efficient SELECT queries with appropriate filters
3. Present results clearly with context

You cannot modify data. If asked to INSERT, UPDATE, DELETE, or modify schema, explain that you only have read access.

Claude Code [通过 stdin 将 hook 输入作为 JSON 传递](#) 给 hook 命令。验证脚本读取此 JSON，提取正在执行的命令，并根据 SQL 写入操作列表检查它。如果检测到写入操作，脚本 [以代码 2 退出](#) 以阻止执行并通过 stderr 向 Claude 返回错误消息。

在项目中的任何位置创建验证脚本。路径必须与 hook 配置中的 `command` 字段匹配：


```
#!/bin/bash

## Blocks SQL write operations, allows SELECT queries

## Read JSON input from stdin
INPUT=$(cat)

## Extract the command field from tool_input using jq
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // empty')

if [ -z "$COMMAND" ]; then
    exit 0
fi

## Block write operations (case-insensitive)
if echo "$COMMAND" | grep -iE '\b(INSERT|UPDATE|DELETE|DROP|CREATE|ALTER|TRUNCATE|REPLACE|MERGE)\b' > /dev/null; then
    echo "Blocked: Write operations not allowed. Use SELECT queries only." >&2
    exit 2
fi

exit 0
```

使脚本可执行：

```
chmod +x ./scripts/validate-readonly-query.sh
```

Hook 通过 stdin 接收 JSON，Bash 命令在 `tool_input.command` 中。退出代码 2 阻止操作并将错误消息反馈给 Claude。有关退出代码和 [Hook input](#) 的完整输入架构的详细信息，请参阅 [Hooks](#)。

后续步骤

现在您了解了 subagents，请探索这些相关功能：

- [使用插件分发 subagents](#) 以在团队或项目中共享 subagents
- [以编程方式运行 Claude Code](#)，使用 Agent SDK 进行 CI/CD 和自动化
- [使用 MCP servers](#) 为 subagents 提供对外部工具和数据的访问

协调 Claude Code 会话团队

协调多个 Claude Code 实例作为一个团队一起工作，具有共享任务、代理间消息传递和集中管理。

Warning:

Agent teams 是实验性功能，默认禁用。通过将 `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS` 添加到你的 `settings.json` 或环境变量来启用它们。Agent teams 在 [已知限制](#) 中存在关于会话恢复、任务协调和关闭行为的问题。

Agent teams 让你协调多个 Claude Code 实例一起工作。一个会话充当团队负责人，协调工作、分配任务和综合结果。队友独立工作，每个都在自己的 context window 中，并直接相互通信。

与 [subagents](#) 不同，subagents 在单个会话中运行，只能向主代理报告，你也可以直接与个别队友互动，无需通过负责人。

Note:

Agent teams 需要 Claude Code v2.1.32 或更高版本。使用 `claude --version` 检查你的版本。

本页涵盖：

- [何时使用 agent teams](#)，包括最佳用例以及它们与 subagents 的比较
- [启动团队](#)
- [控制队友](#)，包括显示模式、任务分配和委派
- [并行工作的最佳实践](#)

何时使用 agent teams

Agent teams 最适合用于并行探索能增加真实价值的任务。有关完整场景，请参阅 [用例示例](#)。最强的用例是：

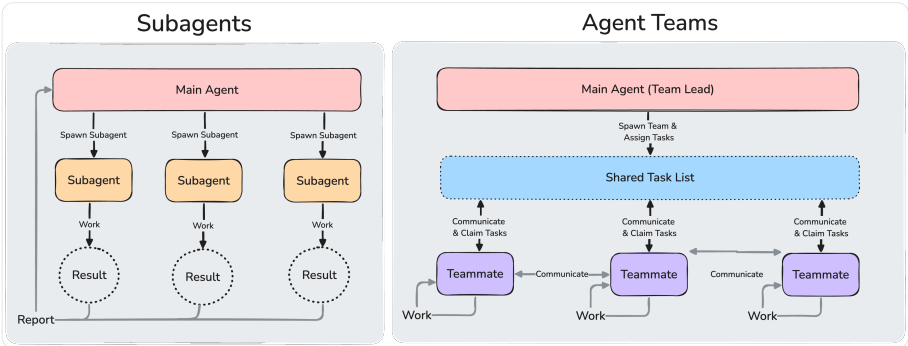
- **研究和审查：**多个队友可以同时调查问题的不同方面，然后分享和质疑彼此的发现
- **新模块或功能：**队友可以各自拥有一个独立的部分，不会相互干扰
- **使用竞争假设进行调试：**队友并行测试不同的理论，更快地收敛到答案

- **跨层协调**：跨越前端、后端和测试的更改，每个由不同的队友负责

Agent teams 增加了协调开销，使用的令牌数量明显多于单个会话。当队友可以独立运作时，它们效果最好。对于顺序任务、同一文件编辑或有许多依赖关系的工作，单个会话或 [subagents](#) 更有效。

与 subagents 比较

Agent teams 和 [subagents](#) 都让你并行化工作，但它们的运作方式不同。根据你的工作人员是否需要相互通信来选择：



比较 subagent 和 agent team 架构的图表。Subagents 由主代理生成、执行工作并报告结果。Agent teams 通过共享任务列表进行协调，队友彼此直接通信。

	Subagents	Agent teams
Context	自己的 context window；结果返回给调用者	自己的 context window；完全独立
通信	仅向主代理报告结果	队友直接相互发送消息
协调	主代理管理所有工作	具有自我协调的共享任务列表
最适合	只有结果重要的专注任务	需要讨论和协作的复杂工作
令牌成本	较低：结果汇总回主 context	较高：每个队友是一个独立的 Claude 实例

当你需要快速、专注的工作人员报告结果时，使用 subagents。当队友需要分享发现、相互质疑和自我协调时，使用 agent teams。

启用 agent teams

Agent teams 默认禁用。通过将 `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS` 环境变量设置为 `1`，在你的 shell 环境中或通过 `settings.json` 来启用它：

```
{
  "env": {
    "CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS": "1"
  }
}
```

启动你的第一个 agent team

启用 agent teams 后，告诉 Claude 创建一个 agent team，并用自然语言描述你想要的任务和团队结构。Claude 创建团队、生成队友并根据你的提示协调工作。

这个例子效果很好，因为三个角色是独立的，可以在不相互等待的情况下探索问题：

```
I'm designing a CLI tool that helps developers track TODO comments across
their codebase. Create an agent team to explore this from different angles: one
teammate on UX, one on technical architecture, one playing devil's advocate.
```

从那里，Claude 创建一个具有 [共享任务列表](#) 的团队，为每个角度生成队友，让他们探索问题，综合发现，并在完成时尝试 [清理团队](#)。

负责人的终端列出所有队友及其正在处理的工作。使用 Shift+Down 循环浏览队友并直接向他们发送消息。在最后一个队友之后，Shift+Down 会回到负责人。

如果你想让每个队友在自己的分割窗格中，请参阅 [选择显示模式](#)。

控制你的 agent team

用自然语言告诉负责人你想要什么。它根据你的指示处理团队协调、任务分配和委派。

选择显示模式

Agent teams 支持两种显示模式：

- **In-process**：所有队友在你的主终端内运行。使用 Shift+Down 循环浏览队友并输入以直接向他们发送消息。在任何终端中工作，无需额外设置。
- **Split panes**：每个队友获得自己的窗格。你可以同时看到每个人的输出，并点击窗格直接交互。需要 tmux 或 iTerm2。

Note:

`tmux` 在某些操作系统上有已知限制，传统上在 macOS 上效果最好。在 iTerm2 中使用 `tmux -CC` 是进入 `tmux` 的建议入口点。

默认值是 `"auto"`，如果你已经在 `tmux` 会话中运行，则使用分割窗格，否则使用 `in-process`。 `"tmux"` 设置启用分割窗格模式，并根据你的终端自动检测是使用 `tmux` 还是 `iTerm2`。要覆盖，在你的 `settings.json` 中设置 `teammateMode`：

```
{
  "teammateMode": "in-process"
}
```

要为单个会话强制 `in-process` 模式，将其作为标志传递：

```
claude --teammate-mode in-process
```

分割窗格模式需要 `tmux` 或 `iTerm2` 与 `it2 CLI`。手动安装：

- **tmux**：通过你的系统包管理器安装。有关特定于平台的说明，请参阅 [tmux wiki](#)。
- **iTerm2**：安装 `it2 CLI`，然后在 `iTerm2` → **Settings** → **General** → **Magic** → **Enable Python API** 中启用 Python API。

指定队友和模型

Claude 根据你的任务决定要生成的队友数量，或者你可以指定你想要的确切内容：

```
Create a team with 4 teammates to refactor these modules in parallel.
Use Sonnet for each teammate.
```

要求队友的计划批准

对于复杂或有风险的任务，你可以要求队友在实施前进行规划。队友在只读计划模式下工作，直到负责人批准他们的方法：

```
Spawn an architect teammate to refactor the authentication module.
Require plan approval before they make any changes.
```

当队友完成规划时，它向负责人发送计划批准请求。负责人审查计划并批准或拒绝并提供反馈。如果被拒绝，队友保持在计划模式，根据反馈进行修订并重新提交。一旦批准，队友退出计划模式并开始实施。

负责人自主做出批准决定。要影响负责人的判断，在你的提示中给出标准，例如”仅批准包括测试覆盖的计划”或”拒绝修改数据库架构的计划”。

直接与队友交谈

每个队友都是一个完整的、独立的 Claude Code 会话。你可以直接向任何队友发送消息，以提供额外的指示、提出后续问题或改变他们的方法。

- **In-process 模式：**使用 Shift+Down 循环浏览队友，然后输入向他们发送消息。按 Enter 查看队友的会话，然后按 Escape 中断他们的当前轮次。按 Ctrl+T 切换任务列表。
- **Split-pane 模式：**点击队友的窗格以直接与他们的会话交互。每个队友都有自己终端的完整视图。

分配和认领任务

共享任务列表协调整个团队的工作。负责人创建任务，队友完成它们。任务有三种状态：待处理、进行中和已完成。任务也可以依赖其他任务：具有未解决依赖关系的待处理任务在这些依赖关系完成之前无法被认领。

负责人可以显式分配任务，或队友可以自我认领：

- **负责人分配：**告诉负责人将哪个任务分配给哪个队友
- **自我认领：**完成任务后，队友自己选择下一个未分配、未阻止的任务

任务认领使用文件锁定来防止多个队友同时尝试认领同一任务时的竞态条件。

关闭队友

要优雅地结束队友的会话：

```
Ask the researcher teammate to shut down
```

负责人发送关闭请求。队友可以批准并优雅地退出，或拒绝并提供解释。

清理团队

完成后，要求负责人清理：

Clean up the team

这会删除共享的团队资源。当负责人运行清理时，它会检查活跃的队友，如果仍有任何队友在运行，则失败，所以先关闭他们。

Warning:

始终使用负责人进行清理。队友不应该运行清理，因为他们的团队 context 可能无法正确解析，可能会使资源处于不一致的状态。

使用 hooks 强制质量门

使用 [hooks](#) 在队友完成工作或任务完成时强制执行规则：

- **TeammateIdle**：当队友即将空闲时运行。以代码 2 退出以发送反馈并保持队友工作。
- **TaskCompleted**：当任务被标记为完成时运行。以代码 2 退出以防止完成并发送反馈。

Agent teams 如何工作

本部分涵盖 agent teams 背后的架构和机制。如果你想开始使用它们，请参阅上面的 [控制你的 agent team](#)。

Claude 如何启动 agent teams

Agent teams 有两种启动方式：

- **你请求一个团队**：给 Claude 一个受益于并行工作的任务，并明确要求一个 agent team。Claude 根据你的指示创建一个。
- **Claude 提议一个团队**：如果 Claude 确定你的任务将受益于并行工作，它可能会建议创建一个团队。你在它继续之前确认。

在这两种情况下，你都保持控制。Claude 不会在没有你的批准的情况下创建团队。

架构

Agent team 由以下部分组成：

组件	角色
Team lead	创建团队、生成队友并协调工作的主 Claude Code 会话
Teammates	各自处理分配任务的独立 Claude Code 实例
Task list	队友认领和完成的共享工作项列表

组件	角色
Mailbox	代理之间通信的消息系统

有关显示配置选项，请参阅 [选择显示模式](#)。队友消息自动到达负责人。

系统自动管理任务依赖关系。当队友完成其他任务依赖的任务时，被阻止的任务会自动解除阻止。

团队和任务存储在本地：

- **Team config** : `~/.claude/teams/{team-name}/config.json`
- **Task list** : `~/.claude/tasks/{team-name}/`

团队配置包含一个 `members` 数组，其中包含每个队友的名称、代理 ID 和代理类型。队友可以读取此文件以发现其他团队成员。

权限

队友从负责人的权限设置开始。如果负责人使用 `--dangerously-skip-permissions` 运行，所有队友也会这样做。生成后，你可以更改个别队友模式，但在生成时无法设置每个队友的模式。

Context 和通信

每个队友都有自己的 context window。生成时，队友加载与常规会话相同的项目 context：CLAUDE.md、MCP servers 和 skills。它还接收来自负责人的生成提示。负责人的对话历史不会继承。

队友如何共享信息：

- **自动消息传递**：当队友发送消息时，它们会自动传递给收件人。负责人不需要轮询更新。
- **空闲通知**：当队友完成并停止时，他们会自动通知负责人。
- **共享任务列表**：所有代理都可以看到任务状态并认领可用工作。

队友消息传递：

- **message**：向一个特定的队友发送消息
- **broadcast**：同时发送给所有队友。谨慎使用，因为成本随团队规模而增加。

令牌使用

Agent teams 使用的令牌数量明显多于单个会话。每个队友都有自己的 context window，令牌使用量随活跃队友数量而增加。对于研究、审查和新功能工作，额外的令牌通常是值得的。对于日常任务，单个会话更具成本效益。有关使用指导，请参阅 [agent team 令牌成本](#)。

用例示例

这些示例展示了 agent teams 如何处理并行探索增加价值的任务。

运行并行代码审查

单个审查者往往一次只关注一种类型的问题。将审查标准分解为独立的领域意味着安全性、性能和测试覆盖都同时获得彻底的关注。提示为每个队友分配一个不同的视角，以便他们不重叠：

```
Create an agent team to review PR #142. Spawn three reviewers:
- One focused on security implications
- One checking performance impact
- One validating test coverage
Have them each review and report findings.
```

每个审查者从同一个 PR 工作，但应用不同的过滤器。负责人在他们完成后综合所有三个的发现。

使用竞争假设进行调查

当根本原因不清楚时，单个代理往往会找到一个看似合理的解释并停止寻找。提示通过让队友明确对抗来对抗这一点：每个队友的工作不仅是调查自己的理论，还要质疑其他队友的理论。

```
Users report the app exits after one message instead of staying connected.
Spawn 5 agent teammates to investigate different hypotheses. Have them talk to
each other to try to disprove each other's theories, like a scientific
debate. Update the findings doc with whatever consensus emerges.
```

辩论结构是这里的关键机制。顺序调查受到锚定的影响：一旦探索了一个理论，后续调查就会偏向于它。

有多个独立的调查者积极尝试相互反驳，存活下来的理论更有可能是实际的根本原因。

最佳实践

给队友足够的 context

队友自动加载项目 context，包括 CLAUDE.md、MCP servers 和 skills，但他们不继承负责人的对话历史。有关详细信息，请参阅 [Context 和通信](#)。在生成提示中包含特定于任务的详细信息：

```
Spawn a security reviewer teammate with the prompt: "Review the authentication module at src/auth/ for security vulnerabilities. Focus on token handling, session management, and input validation. The app uses JWT tokens stored in httpOnly cookies. Report any issues with severity ratings."
```

选择适当的团队规模

队友数量没有硬限制，但实际限制适用：

- **令牌成本线性增加**：每个队友都有自己的 context window 并独立消耗令牌。有关详细信息，请参阅 [agent team 令牌成本](#)。
- **协调开销增加**：更多队友意味着更多通信、任务协调和潜在冲突
- **收益递减**：超过一定点，额外的队友不会按比例加快工作

对于大多数工作流，从 3-5 个队友开始。这平衡了并行工作和可管理的协调。本指南中的示例使用 3-5 个队友，因为该范围在不同任务类型中效果很好。

每个队友有 5-6 个 [tasks](#) 可以让每个人保持生产力，而不会过度的上下文切换。如果你有 15 个独立任务，3 个队友是一个很好的起点。

仅当工作真正受益于队友同时工作时才扩展。三个专注的队友通常胜过五个分散的队友。

适当调整任务大小

- **太小**：协调开销超过收益
- **太大**：队友长时间工作而不进行检查，增加浪费努力的风险
- **恰到好处**：自包含的单位，产生清晰的可交付成果，例如函数、测试文件或审查

Tip:

负责人将工作分解为任务并自动分配给队友。如果它没有创建足够的任务，要求它将工作分成更小的部分。每个队友有 5-6 个任务可以让每个人保持生产力，并让负责人在有人卡住时重新分配工作。

等待队友完成

有时负责人开始自己实施任务，而不是等待队友。如果你注意到这一点：

```
Wait for your teammates to complete their tasks before proceeding
```

从研究和审查开始

如果你是 agent teams 的新手，从具有明确边界且不需要编写代码的任务开始：审查 PR、研究库或调查错误。这些任务展示了并行探索的价值，而不会带来并行实施所带来的协调挑战。

避免文件冲突

两个队友编辑同一文件会导致覆盖。分解工作，使每个队友拥有不同的文件集。

监控和指导

检查队友的进度，重定向不起作用的方法，并在发现时综合发现。让团队无人值守运行太长时间会增加浪费努力的风险。

故障排除

队友未出现

如果你要求 Claude 创建团队后队友没有出现：

- 在 in-process 模式中，队友可能已经在运行但不可见。按 Shift+Down 循环浏览活跃的队友。
- 检查你给 Claude 的任务是否足够复杂以保证一个团队。Claude 根据任务决定是否生成队友。
- 如果你明确要求分割窗格，请确保 tmux 已安装并在你的 PATH 中可用：

```
which tmux
```

- 对于 iTerm2，验证 `it2` CLI 已安装，并在 iTerm2 偏好设置中启用了 Python API。

过多权限提示

队友权限请求冒泡到负责人，这可能会造成摩擦。在生成队友之前，在你的 [权限设置](#) 中预批准常见操作，以减少中断。

队友在错误后停止

队友可能在遇到错误后停止，而不是恢复。在 in-process 模式中使用 Shift+Down 或在分割模式中点击窗格来检查他们的输出，然后：

- 直接给他们额外的指示
- 生成一个替代队友来继续工作

负责人在工作完成前关闭

负责人可能会在所有任务实际完成之前决定团队已完成。如果发生这种情况，告诉它继续。你也可以告诉负责人在继续之前等待队友完成，如果它开始做工作而不是委派。

孤立的 tmux 会话

如果 tmux 会话在团队结束后仍然存在，它可能没有被完全清理。列出会话并杀死由团队创建的会话：

```
tmux ls
tmux kill-session -t <session-name>
```

限制

Agent teams 是实验性的。需要注意的当前限制：

- **In-process 队友没有会话恢复：** `/resume` 和 `/rewind` 不会恢复 in-process 队友。恢复会话后，负责人可能会尝试向不再存在的队友发送消息。如果发生这种情况，告诉负责人生成新队友。
- **任务状态可能滞后：** 队友有时无法将任务标记为已完成，这会阻止依赖任务。如果任务似乎卡住，检查工作是否实际完成，并手动更新任务状态或告诉负责人推动队友。
- **关闭可能很慢：** 队友在关闭前完成他们的当前请求或工具调用，这可能需要时间。
- **每个会话一个团队：** 负责人一次只能管理一个团队。在启动新团队之前清理当前团队。
- **没有嵌套团队：** 队友无法生成自己的团队或队友。只有负责人可以管理团队。
- **负责人是固定的：** 创建团队的会话在其生命周期内是负责人。你无法将队友提升为负责人或转移领导权。
- **权限在生成时设置：** 所有队友从负责人的权限模式开始。你可以在生成后更改个别队友模式，但在生成时无法设置每个队友的模式。
- **分割窗格需要 tmux 或 iTerm2：** 默认 in-process 模式在任何终端中工作。VS Code 的集成终端、Windows Terminal 或 Ghostty 不支持分割窗格模式。

Tip:

CLAUDE.md 正常工作：队友从他们的工作目录读取 **CLAUDE.md** 文件。使用这个为所有队友提供项目特定的指导。

后续步骤

探索用于并行工作和委派的相关方法：

- **轻量级委派：**[subagents](#) 在你的会话中为研究或验证生成辅助代理，更适合不需要代理间协调的任务
- **手动并行会话：**[Git worktrees](#) 让你自己运行多个 Claude Code 会话，无需自动化团队协调
- **比较方法：**有关并排分解，请参阅 [subagent vs agent team](#) 比较

以编程方式运行 Claude Code

使用 Agent SDK 从 CLI、Python 或 TypeScript 以编程方式运行 Claude Code。

[Agent SDK](#) 为您提供了与 Claude Code 相同的工具、agent 循环和上下文管理。它可作为 CLI 用于脚本和 CI/CD，或作为 [Python](#) 和 [TypeScript](#) 包供完整的编程控制。

Note:

CLI 之前称为“headless mode”。`-p` 标志和所有 CLI 选项的工作方式相同。

要从 CLI 以编程方式运行 Claude Code，请使用 `-p` 传递您的提示和任何 [CLI 选项](#)：

```
claude -p "Find and fix the bug in auth.py" --allowedTools "Read,Edit,Bash"
```

本页面介绍通过 CLI (`claude -p`) 使用 Agent SDK。对于具有结构化输出、工具批准回调和本机消息对象的 Python 和 TypeScript SDK 包，请参阅 [完整 Agent SDK 文档](#)。

基本用法

将 `-p`（或 `--print`）标志添加到任何 `claude` 命令以非交互方式运行它。所有 [CLI 选项](#) 都适用于 `-p`，包括：

- `--continue` 用于 [继续对话](#)
- `--allowedTools` 用于 [自动批准工具](#)
- `--output-format` 用于 [获取结构化输出](#)

此示例询问 Claude 关于您的代码库的问题并打印响应：

```
claude -p "What does the auth module do?"
```

示例

这些示例突出了常见的 CLI 模式。

获取结构化输出

使用 `--output-format` 控制响应的返回方式：

- `text`（默认）：纯文本输出
- `json`：包含结果、会话 ID 和元数据的结构化 JSON
- `stream-json`：用于实时流式传输的换行符分隔的 JSON

此示例以 JSON 格式返回项目摘要以及会话元数据，文本结果在 `result` 字段中：

```
claude -p "Summarize this project" --output-format json
```

要获得符合特定架构的输出，请使用 `--output-format json` 与 `--json-schema` 和 [JSON Schema](#) 定义。响应包括有关请求的元数据（会话 ID、使用情况等），结构化输出在 `structured_output` 字段中。

此示例从 `auth.py` 中提取函数名称并将其作为字符串数组返回：

```
claude -p "Extract the main function names from auth.py" \
  --output-format json \
  --json-schema '{"type":"object","properties":{"functions":
{"type":"array","items":{"type":"string"}},"required":["functions"]}'
```

Tip:

使用 `jq` 之类的工具来解析响应并提取特定字段：

```
## Extract the text result
claude -p "Summarize this project" --output-format json | jq -r '.result'

## Extract structured output
claude -p "Extract function names from auth.py" \
  --output-format json \
  --json-schema '{"type":"object","properties":{"functions":
{"type":"array","items":{"type":"string"}},"required":["functions"]}' \
  | jq '.structured_output'
```

流式传输响应

使用 `--output-format stream-json` 与 `--verbose` 和 `--include-partial-messages` 来接
收生成的令牌。每一行都是代表一个事件的 JSON 对象：

```
claude -p "Explain recursion" --output-format stream-json --verbose --include-partial-messages
```

以下示例使用 `jq` 来过滤文本增量并仅显示流式文本。`-r` 标志输出原始字符串（无引号），`-j` 不带换行符连接，以便令牌连续流式传输：

```
claude -p "Write a poem" --output-format stream-json --verbose --include-partial-messages | \
jq -rj 'select(.type == "stream_event" and .event.delta.type? == "text_delta")
| .event.delta.text'
```

有关具有回调和消息对象的编程流式传输，请参阅 Agent SDK 文档中的 [实时流式传输响应](#)。

自动批准工具

使用 `--allowedTools` 让 Claude 使用某些工具而无需提示。此示例运行测试套件并修复失败，允许 Claude 执行 Bash 命令和读取/编辑文件而无需请求权限：

```
claude -p "Run the test suite and fix any failures" \
--allowedTools "Bash,Read,Edit"
```

创建提交

此示例审查暂存的更改并创建具有适当消息的提交：

```
claude -p "Look at my staged changes and create an appropriate commit" \
--allowedTools "Bash(git diff *),Bash(git log *),Bash(git status *),Bash(git
commit *)"
```

`--allowedTools` 标志使用 [权限规则语法](#)。尾部的 `*` 启用前缀匹配，因此 `Bash(git diff *)` 允许任何以 `git diff` 开头的命令。空格在 `*` 之前很重要：没有它，`Bash(git diff*)` 也会匹配 `git diff-index`。

Note:

用户调用的 [skills](#) 如 `/commit` 和 [内置命令](#) 仅在交互模式中可用。在 `-p` 模式中，改为描述您想要完成的任务。

自定义系统提示

使用 `--append-system-prompt` 添加指令同时保持 Claude Code 的默认行为。此示例将 PR diff 传递给 Claude 并指示它审查安全漏洞：

```
gh pr diff "$1" | claude -p \  
  --append-system-prompt "You are a security engineer. Review for  
vulnerabilities." \  
  --output-format json
```

有关更多选项（包括 `--system-prompt` 以完全替换默认提示），请参阅 [系统提示标志](#)。

继续对话

使用 `--continue` 继续最近的对话，或使用 `--resume` 与会话 ID 继续特定对话。此示例运行审查，然后发送后续提示：

```
## First request  
claude -p "Review this codebase for performance issues"  
  
## Continue the most recent conversation  
claude -p "Now focus on the database queries" --continue  
claude -p "Generate a summary of all issues found" --continue
```

如果您运行多个对话，请捕获会话 ID 以恢复特定对话：

```
session_id=$(claude -p "Start a review" --output-format json | jq -r  
'$.session_id')  
claude -p "Continue that review" --resume "$session_id"
```

后续步骤

- [Agent SDK 快速入门](#)：使用 Python 或 TypeScript 构建您的第一个 agent
- [CLI 参考](#)：所有 CLI 标志和选项
- [GitHub Actions](#)：在 GitHub 工作流中使用 Agent SDK
- [GitLab CI/CD](#)：在 GitLab 管道中使用 Agent SDK

使用 Remote Control 从任何设备继续本地会话

使用 Remote Control 从您的手机、平板电脑或任何浏览器继续本地 Claude Code 会话。适用于 claude.ai/code 和 Claude 移动应用。

Note:

Remote Control 在所有计划中都可用。Team 和 Enterprise 管理员必须首先在[管理员设置](#)中启用 Claude Code。

Remote Control 将 claude.ai/code 或 Claude 应用（iOS 和 Android）连接到在您的机器上运行的 Claude Code 会话。在您的办公桌上启动任务，然后从沙发上的手机或另一台计算机上的浏览器继续。

当您在机器上启动 Remote Control 会话时，Claude 会在整个过程中在本地运行，因此没有任何内容会移到云端。使用 Remote Control，您可以：

- **远程使用您的完整本地环境：**您的文件系统、[MCP servers](#)、工具和项目配置都保持可用
- **同时在两个界面上工作：**对话在所有连接的设备上保持同步，因此您可以从终端、浏览器和手机交替发送消息
- **应对中断：**如果您的笔记本电脑进入睡眠状态或网络断开，当您的机器重新上线时，会话会自动重新连接

与在云基础设施上运行的 [Claude Code on the web](#) 不同，Remote Control 会话直接在您的机器上运行并与您的本地文件系统交互。Web 和移动界面只是该本地会话的一个窗口。

Note:

Remote Control 需要 Claude Code v2.1.51 或更高版本。使用 `claude --version` 检查您的版本。

本页涵盖设置、如何启动和连接到会话，以及 Remote Control 与 Claude Code on the web 的比较。

要求

在使用 Remote Control 之前，请确认您的环境满足以下条件：

- **订阅**：在 Pro、Max、Team 和 Enterprise 计划中可用。Team 和 Enterprise 管理员必须首先在[管理员设置](#)中启用 Claude Code。不支持 API 密钥。
- **身份验证**：运行 `claude` 并使用 `/login` 通过 claude.ai 登录（如果您还没有登录）。
- **工作区信任**：在您的项目目录中至少运行一次 `claude` 以接受工作区信任对话框。

启动 Remote Control 会话

您可以直接在 Remote Control 中启动新会话，或连接已经运行的会话。

新会话

导航到您的项目目录并运行：

```
claude remote-control
```

该过程在您的终端中保持运行，等待远程连接。它显示一个会话 URL，您可以使用该 URL 从另一个设备[连接](#)，您可以按空格键显示 QR 码以从您的手机快速访问。当远程会话处于活动状态时，终端显示连接状态和工具活动。

此命令支持以下标志：

- `--name "My Project"`：设置在 claude.ai/code 的会话列表中可见的自定义会话标题。您也可以将名称作为位置参数传递：`claude remote-control "My Project"`
- `--verbose`：显示详细的连接和会话日志
- `--sandbox` / `--no-sandbox`：启用或禁用[沙箱](#)以在会话期间进行文件系统和网络隔离。默认情况下沙箱处于关闭状态。

从现有会话

如果您已经在 Claude Code 会话中并想远程继续它，请使用 `/remote-control`（或 `/rc`）命令：

```
/remote-control
```

将名称作为参数传递以设置自定义会话标题：

```
/remote-control My Project
```

这启动一个 Remote Control 会话，该会话继承您当前的对话历史记录，并显示一个会话 URL 和 QR 码，您可以使用它从另一个设备[连接](#)。 `--verbose`、`--sandbox` 和 `--no-sandbox` 标志不适用于此命令。

从另一个设备连接

一旦 Remote Control 会话处于活动状态，您有几种方式从另一个设备连接：

- **打开会话 URL** 在任何浏览器中直接转到 claude.ai/code 上的会话。 `claude remote-control` 和 `/remote-control` 都在终端中显示此 URL。
- **扫描 QR 码** 显示在会话 URL 旁边，直接在 Claude 应用中打开它。使用 `claude remote-control` 时，按空格键切换 QR 码显示。
- **打开 claude.ai/code 或 Claude 应用**，在会话列表中按名称查找会话。Remote Control 会话在联机时显示带有绿色状态点的计算机图标。

远程会话从 `--name` 参数（或传递给 `/remote-control` 的名称）、您的最后一条消息、您的 `/rename` 值或“Remote Control session”（如果没有对话历史记录）获取其名称。如果环境已经有活动会话，系统会询问您是否继续它或启动新会话。

如果您还没有 Claude 应用，请在 Claude Code 中使用 `/mobile` 命令显示 [iOS](#) 或 [Android](#) 的下载 QR 码。

为所有会话启用 Remote Control

默认情况下，Remote Control 仅在您显式运行 `claude remote-control` 或 `/remote-control` 时激活。要为每个会话自动启用它，请在 Claude Code 中运行 `/config` 并将**为所有会话启用 Remote Control** 设置为 `true`。将其设置回 `false` 以禁用。

每个 Claude Code 实例一次支持一个远程会话。如果您运行多个实例，每个实例都有自己的环境和会话。

连接和安全

您的本地 Claude Code 会话仅发出出站 HTTPS 请求，从不在您的机器上打开入站端口。当您启动 Remote Control 时，它向 Anthropic API 注册并轮询工作。当您从另一个设备连接时，服务器通过流连接在 Web 或移动客户端与您的本地会话之间路由消息。

所有流量都通过 TLS 上的 Anthropic API 传输，与任何 Claude Code 会话的传输安全相同。连接使用多个短期凭证，每个凭证的范围限定为单一目的并独立过期。

Remote Control 与 Claude Code on the web 的比较

Remote Control 和 [Claude Code on the web](#) 都使用 `claude.ai/code` 界面。关键区别在于会话运行的位置：Remote Control 在您的机器上执行，因此您的本地 MCP servers、工具和项目配置保持可用。Claude Code on the web 在 Anthropic 管理的云基础设施中执行。

当您在本地工作中途并想从另一个设备继续时，使用 Remote Control。当您想在没有任何本地设置的情况下启动任务、处理您没有克隆的存储库或并行运行多个任务时，使用 Claude Code on the web。

限制

- **一次一个远程会话：**每个 Claude Code 会话支持一个远程连接。
- **终端必须保持打开：**Remote Control 作为本地进程运行。如果您关闭终端或停止 `claude` 进程，会话将结束。再次运行 `claude remote-control` 以启动新会话。
- **扩展网络中断：**如果您的机器处于唤醒状态但无法在大约 10 分钟以上的时间内到达网络，会话将超时并且进程退出。再次运行 `claude remote-control` 以启动新会话。

相关资源

- [Claude Code on the web](#)：在 Anthropic 管理的云环境中运行会话，而不是在您的机器上
- [身份验证](#)：设置 `/login` 并管理 `claude.ai` 的凭证
- [CLI 参考](#)：包括 `claude remote-control` 的标志和命令的完整列表
- [安全](#)：Remote Control 会话如何适应 Claude Code 安全模型
- [数据使用](#)：在本地和远程会话期间通过 Anthropic API 流动的数据

按计划运行提示词

使用 `/loop` 和 `cron` 调度工具在 Claude Code 会话中重复运行提示词、轮询状态或设置一次性提醒。

Note:

计划任务需要 Claude Code v2.1.72 或更高版本。使用 `claude --version` 检查您的版本。

计划任务让 Claude 按间隔自动重新运行提示词。使用它们来轮询部署、监督 PR、检查长时间运行的构建，或在会话中稍后提醒自己做某事。

任务是会话范围的：它们存在于当前 Claude Code 进程中，当您退出时就会消失。对于需要在重启后继续运行且无需活跃终端会话的持久调度，请参阅 [Desktop 计划任务](#) 或 [GitHub Actions](#)。

使用 `/loop` 计划重复提示词

`/loop` [bundled skill](#) 是计划重复提示词的最快方式。传递一个可选的间隔和一个提示词，Claude 会设置一个在后台运行的 `cron` 作业，同时会话保持打开。

```
/loop 5m check if the deployment finished and tell me what happened
```

Claude 解析间隔，将其转换为 `cron` 表达式，计划作业，并确认节奏和作业 ID。

间隔语法

间隔是可选的。您可以在开头使用它们、在末尾使用它们，或完全省略它们。

形式	示例	解析的间隔
前导令牌	<code>/loop 30m check the build</code>	每 30 分钟
尾部 <code>every</code> 子句	<code>/loop check the build every 2 hours</code>	每 2 小时
无间隔	<code>/loop check the build</code>	默认为每 10 分钟

支持的单位是 **s** 表示秒、**m** 表示分钟、**h** 表示小时、**d** 表示天。秒数向上舍入到最近的分钟，因为 cron 的粒度为一分钟。不能均匀分割其单位的间隔，例如 **7m** 或 **90m**，会舍入到最近的整洁间隔，Claude 会告诉您它选择了什么。

循环另一个命令

计划的提示词本身可以是命令或 skill 调用。这对于重新运行您已经打包的工作流很有用。

```
/loop 20m /review-pr 1234
```

每次作业触发时，Claude 都会运行 **/review-pr 1234**，就像您输入了它一样。

设置一次性提醒

对于一次性提醒，用自然语言描述您想要的内容，而不是使用 **/loop**。Claude 计划一个单次触发的任务，在运行后删除自己。

```
remind me at 3pm to push the release branch
```

```
in 45 minutes, check whether the integration tests passed
```

Claude 使用 cron 表达式将触发时间固定到特定的分钟和小时，并确认何时触发。

管理计划任务

用自然语言要求 Claude 列出或取消任务，或直接引用底层工具。

```
what scheduled tasks do I have?
```

```
cancel the deploy check job
```

在幕后，Claude 使用这些工具：

工具	目的
CronCreate	计划新任务。接受 5 字段 cron 表达式、要运行的提示词以及是否重复或仅触发一次。

工具	目的
<code>CronList</code>	列出所有计划任务及其 ID、计划和提示词。
<code>CronDelete</code>	按 ID 取消任务。

每个计划任务都有一个 8 字符的 ID，您可以将其传递给 `CronDelete`。一个会话最多可以同时保存 50 个计划任务。

计划任务如何运行

调度程序每秒检查一次到期的任务，并以低优先级将其入队。计划的提示词在您的回合之间触发，而不是在 Claude 正在响应时。如果 Claude 在任务到期时忙碌，提示词会等到当前回合结束。

所有时间都在您的本地时区中解释。像 `0 9 * * *` 这样的 cron 表达式意味着 9am 在您运行 Claude Code 的任何地方，而不是 UTC。

抖动

为了避免每个会话在同一个挂钟时刻击中 API，调度程序会向触发时间添加一个小的确定性偏移：

- 重复任务最多晚触发其周期的 10%，上限为 15 分钟。每小时的作业可能在 `:00` 到 `:06` 之间的任何时间触发。
- 为小时顶部或底部计划的一次性任务最多提前 90 秒触发。

偏移是从任务 ID 派生的，所以相同的任务总是获得相同的偏移。如果精确的时间很重要，选择不是 `:00` 或 `:30` 的分钟，例如 `3 9 * * *` 而不是 `0 9 * * *`，一次性抖动将不适用。

三天过期

重复任务在创建后 3 天自动过期。任务最后触发一次，然后删除自己。这限制了被遗忘的循环可以运行多长时间。如果您需要重复任务持续更长时间，请在过期前取消并重新创建它，或使用 [Desktop 计划任务](#) 进行持久调度。

Cron 表达式参考

`CronCreate` 接受标准 5 字段 cron 表达式：`minute hour day-of-month month day-of-week`。所有字段都支持通配符 (`*`)、单个值 (`5`)、步长 (`*/15`)、范围 (`1-5`) 和逗号分隔的列表 (`1,15,30`)。

示例	含义
<code>* / 5 * * * *</code>	每 5 分钟
<code>0 * * * *</code>	每小时整点
<code>7 * * * *</code>	每小时的第 7 分钟
<code>0 9 * * *</code>	每天本地时间 9am
<code>0 9 * * 1-5</code>	工作日本地时间 9am
<code>30 14 15 3 *</code>	3 月 15 日本地时间下午 2:30

星期几使用 `0` 或 `7` 表示星期日，`6` 表示星期六。不支持扩展语法如 `L`、`W`、`?` 和名称别名如 `MON` 或 `JAN`。

当月份日期和星期几都受到限制时，如果任一字段匹配，日期就匹配。这遵循标准的 `vixie-cron` 语义。

禁用计划任务

在您的环境中设置 `CLAUDE_CODE_DISABLE_CRON=1` 以完全禁用调度程序。`cron` 工具和 `/loop` 变得不可用，任何已计划的任务都停止触发。有关禁用标志的完整列表，请参阅 [环境变量](#)。

限制

会话范围的调度有固有限制：

- 任务仅在 Claude Code 运行且空闲时触发。关闭终端或让会话退出会取消所有内容。
- 没有错过触发的追赶。如果任务的计划时间在 Claude 忙于长时间运行的请求时经过，它会在 Claude 变为空闲时触发一次，而不是每个错过的间隔触发一次。
- 没有跨重启的持久性。重启 Claude Code 会清除所有会话范围的任务。

对于需要无人值守运行的 `cron` 驱动自动化，使用带有 `schedule` 触发器的 [GitHub Actions 工作流](#)，或如果您想要图形化设置流程，使用 [Desktop 计划任务](#)。

代码审查

设置自动化 PR 审查，使用多代理分析您的完整代码库来捕获逻辑错误、安全漏洞和回归问题

Note:

Code Review 处于研究预览阶段，可用于 [Teams](#) 和 [Enterprise](#) 订阅。对于启用了 [Zero Data Retention](#) 的组织，不可用。

Code Review 分析您的 GitHub pull request，并在发现问题的代码行上发布内联注释。一队专门的代理在您完整代码库的上下文中检查代码更改，寻找逻辑错误、安全漏洞、破损的边界情况和微妙的回归问题。

发现结果按严重程度标记，不会批准或阻止您的 PR，因此现有的审查工作流保持不变。您可以通过向存储库添加 `CLAUDE.md` 或 `REVIEW.md` 文件来调整 Claude 标记的内容。

要在您自己的 CI 基础设施中运行 Claude 而不是使用此托管服务，请参阅 [GitHub Actions](#) 或 [GitLab CI/CD](#)。

本页涵盖：

- [审查工作原理](#)
- [设置](#)
- [自定义审查](#)，使用 `CLAUDE.md` 和 `REVIEW.md`
- [定价](#)

审查工作原理

一旦管理员为您的组织 [启用 Code Review](#)，当 pull request 打开或更新时，审查会自动运行。多个代理在 Anthropic 基础设施上并行分析差异和周围代码。每个代理寻找不同类别的问题，然后验证步骤根据实际代码行为检查候选项以过滤掉误报。结果被去重、按严重程度排序，并作为内联注释发布在发现问题的特定行上。如果未发现问题，Claude 会在 PR 上发布简短的确认注释。

审查成本随 PR 大小和复杂性而扩展，平均在 20 分钟内完成。管理员可以通过 [分析仪表板](#) 监控审查活动和支出。

严重程度级别

每个发现都标记有严重程度级别：

标记	严重程度	含义
	正常	应在合并前修复的错误
	小问题	轻微问题，值得修复但不阻止
	预先存在	代码库中存在但不是由此 PR 引入的错误

发现包括可折叠的扩展推理部分，您可以展开以了解 Claude 为什么标记该问题以及它如何验证问题。

Code Review 检查的内容

默认情况下，Code Review 专注于正确性：会破坏生产的错误，而不是格式偏好或缺失的测试覆盖。您可以通过 [添加指导文件](#) 到您的存储库来扩展它检查的内容。

设置 Code Review

管理员为组织启用一次 Code Review，并选择要包含的存储库。

Step 1: 打开 Claude Code 管理员设置

转到 claude.ai/admin-settings/claude-code 并找到 Code Review 部分。您需要对 Claude 组织的管理员访问权限和在 GitHub 组织中安装 GitHub Apps 的权限。

Step 2: 开始设置

点击 **Setup**。这开始 GitHub App 安装流程。

Step 3: 安装 Claude GitHub App

按照提示将 Claude GitHub App 安装到您的 GitHub 组织。该应用请求这些存储库权限：

- **Contents**：读和写
- **Issues**：读和写
- **Pull requests**：读和写

Code Review 使用对内容的读访问权限和对 pull request 的写访问权限。更广泛的权限集也支持 [GitHub Actions](#)，如果您稍后启用它。

Step 4: 选择存储库

选择要为 Code Review 启用的存储库。如果您看不到存储库，请确保在安装期间给了 Claude GitHub App 访问权限。您可以稍后添加更多存储库。

Step 5: 为每个存储库设置审查触发器

设置完成后，Code Review 部分在表格中显示您的存储库。对于每个存储库，使用下拉菜单选择审查何时运行：

- **仅在 PR 创建后：**当 PR 打开或标记为准备审查时审查运行一次
- **在每次推送到 PR 分支后：**在每次推送时运行审查，在 PR 演变时捕获新问题，并在您修复标记的问题时自动解决线程

在每次推送时审查会运行更多审查并花费更多。从仅 PR 创建开始，对于您想要持续覆盖和自动线程清理的存储库切换到推送时。

存储库表还显示每个存储库基于最近活动的平均审查成本。使用行操作菜单为每个存储库打开或关闭 Code Review，或完全删除存储库。

要验证设置，请打开测试 PR。在几分钟内会出现名为 **Claude Code Review** 的检查运行。如果没有，请确认存储库在您的管理员设置中列出，并且 Claude GitHub App 有权访问它。

自定义审查

Code Review 从您的存储库读取两个文件来指导它标记的内容。两者都是在默认正确性检查之上的附加：

- **CLAUDE.md**：共享项目说明，Claude Code 用于所有任务，不仅仅是审查。当指导也适用于交互式 Claude Code 会话时使用它。
- **REVIEW.md**：仅审查指导，在代码审查期间专门读取。对于严格关于在审查期间标记或跳过什么的规则，以及会使您的常规 **CLAUDE.md** 混乱的规则，使用它。

CLAUDE.md

Code Review 读取您的存储库的 **CLAUDE.md** 文件，并将新引入的违规视为小问题级别的发现。这是双向工作的：如果您的 PR 以使 **CLAUDE.md** 语句过时的方式更改代码，Claude 会标记文档需要更新。

Claude 在目录层次结构的每个级别读取 **CLAUDE.md** 文件，因此子目录的 **CLAUDE.md** 中的规则仅适用于该路径下的文件。有关 **CLAUDE.md** 如何工作的更多信息，请参阅 [内存文档](#)。

对于您不想应用于常规 Claude Code 会话的仅审查指导，请改用 **REVIEW.md**。

REVIEW.md

将 **REVIEW.md** 文件添加到您的存储库根目录以获取仅审查规则。使用它来编码：

- 公司或团队风格指南：“优先使用早期返回而不是嵌套条件”
- 语言或框架特定的约定，不被 linter 覆盖

- Claude 应始终标记的内容：“任何新 API 路由必须有集成测试”
- Claude 应跳过的内容：“不要对 `/gen/` 下生成的代码中的格式进行注释”

示例 `REVIEW.md`：

```
## 代码审查指南

### 始终检查
- 新 API 端点有相应的集成测试
- 数据库迁移向后兼容
- 错误消息不会向用户泄露内部详情

### 风格
- 优先使用 `match` 语句而不是链式 `instance` 检查
- 使用结构化日志，而不是日志调用中的 f 字符串插值

### 跳过
- `src/gen/` 下的生成文件
- `*.lock` 文件中仅格式的更改
```

Claude 自动发现存储库根目录中的 `REVIEW.md`。无需配置。

查看使用情况

转到 claude.ai/analytics/code-review 以查看整个组织的 Code Review 活动。仪表板显示：

部分	显示内容
审查的 PR	所选时间范围内审查的 pull request 的每日计数
每周成本	Code Review 的每周支出
反馈	因开发人员解决问题而自动解决的审查注释计数
存储库分解	每个存储库审查的 PR 计数和解决的注释

管理员设置中的存储库表也显示每个存储库的平均审查成本。

定价

Code Review 根据令牌使用情况计费。审查平均 \$15-25，随 PR 大小、代码库复杂性和需要验证的问题数量而扩展。Code Review 使用通过 [额外使用](#) 单独计费，不计入您的计划包含的使用。

您选择的审查触发器影响总成本：

- **仅在 PR 创建后：**每个 PR 运行一次
- **在每次推送时：**在每次提交时运行，将成本乘以推送次数

无论您的组织是否为其他 Claude Code 功能使用 AWS Bedrock 或 Google Vertex AI，成本都会出现在您的 Anthropic 账单上。要为 Code Review 设置每月支出上限，请转到 claude.ai/admin-settings/usage 并为 Claude Code Review 服务配置限制。

通过 [分析](#) 中的每周成本图表或管理员设置中的每个存储库平均成本列监控支出。

相关资源

Code Review 设计用于与 Claude Code 的其余部分一起工作。如果您想在打开 PR 之前在本地运行审查、需要自托管设置或想更深入地了解 [CLAUDE.md](#) 如何在工具中塑造 Claude 的行为，这些页面是很好的下一步：

- [Plugins](#)：浏览插件市场，包括用于在推送前本地运行按需审查的 [code-review](#) 插件
- [GitHub Actions](#)：在您自己的 GitHub Actions 工作流中运行 Claude，用于超越代码审查的自定义自动化
- [GitLab CI/CD](#)：GitLab 管道的自托管 Claude 集成
- [Memory](#)：[CLAUDE.md](#) 文件如何在 Claude Code 中工作
- [Analytics](#)：跟踪超越代码审查的 Claude Code 使用情况

Part 7: CI/CD & Integrations

Claude Code GitHub Actions

了解如何将 Claude Code 集成到您的开发工作流中，使用 Claude Code GitHub Actions

Claude Code GitHub Actions 为您的 GitHub 工作流带来了 AI 驱动的自动化。只需在任何 PR 或 issue 中简单地提及 `@claude`，Claude 就可以分析您的代码、创建拉取请求、实现功能和修复错误 - 所有这些都遵循您项目的标准。如需在每个 PR 上自动发布评论而无需触发器，请参阅 [GitHub Code Review](#)。

Note:

Claude Code GitHub Actions 建立在 [Claude Agent SDK](#) 之上，该 SDK 支持将 Claude Code 以编程方式集成到您的应用程序中。您可以使用该 SDK 构建超越 GitHub Actions 的自定义自动化工作流。

Info:

Claude Opus 4.6 现已推出。 Claude Code GitHub Actions 默认使用 Sonnet。要使用 Opus 4.6，请配置 `model` 参数以使用 `claude-opus-4-6`。

为什么使用 Claude Code GitHub Actions？

- **即时 PR 创建：**描述您需要什么，Claude 会创建一个包含所有必要更改的完整 PR
- **自动化代码实现：**通过单个命令将 issue 转换为可工作的代码
- **遵循您的标准：**Claude 尊重您的 `CLAUDE.md` 指南和现有代码模式
- **简单设置：**通过我们的安装程序和 API 密钥在几分钟内开始使用
- **默认安全：**您的代码保留在 Github 的运行器上

Claude 可以做什么？

Claude Code 提供了一个强大的 GitHub Action，改变了您处理代码的方式：

Claude Code Action

这个 GitHub Action 允许您在 GitHub Actions 工作流中运行 Claude Code。您可以使用它在 Claude Code 之上构建任何自定义工作流。

[查看仓库 →](#)

设置

快速设置

设置此 action 的最简单方法是通过终端中的 Claude Code。只需打开 claude 并运行 `/install-github-app`。

此命令将指导您完成 GitHub 应用和所需密钥的设置。

Note:

- 您必须是仓库管理员才能安装 GitHub 应用并添加密钥
- GitHub 应用将请求对内容、Issue 和拉取请求的读写权限
- 此快速启动方法仅适用于直接 Claude API 用户。如果您使用 AWS Bedrock 或 Google Vertex AI，请参阅 [使用 AWS Bedrock](#) 和 [Google Vertex AI](#) 部分。

手动设置

如果 `/install-github-app` 命令失败或您更喜欢手动设置，请按照以下手动设置说明进行操作：

1. **安装 Claude GitHub 应用**到您的仓库：<https://github.com/apps/claude>

Claude GitHub 应用需要以下仓库权限：

- **Contents**：读写（用于修改仓库文件）
- **Issues**：读写（用于响应 issue）
- **Pull requests**：读写（用于创建 PR 和推送更改）

有关安全和权限的更多详情，请参阅 [安全文档](#)。

2. **添加 ANTHROPIC_API_KEY** 到您的仓库密钥（[了解如何在 GitHub Actions 中使用密钥](#)）
3. **复制工作流文件**从 [examples/claude.yml](#) 到您的仓库的 `.github/workflows/`

Tip:

完成快速启动或手动设置后，通过在 issue 或 PR 评论中标记 `@claude` 来测试该 action。

从 Beta 升级

Warning:

Claude Code GitHub Actions v1.0 引入了重大更改，需要更新您的工作流文件才能从 beta 版本升级到 v1.0。

如果您当前使用 Claude Code GitHub Actions 的 beta 版本，我们建议您更新工作流以使用 GA 版本。新版本简化了配置，同时添加了强大的新功能，如自动模式检测。

基本更改

所有 beta 用户必须对其工作流文件进行这些更改才能升级：

- 更新 action 版本：**将 `@beta` 更改为 `@v1`
- 删除模式配置：**删除 `mode: "tag"` 或 `mode: "agent"`（现在自动检测）
- 更新提示输入：**将 `direct_prompt` 替换为 `prompt`
- 移动 CLI 选项：**将 `max_turns`、`model`、`custom_instructions` 等转换为 `claude_args`

重大更改参考

旧 Beta 输入	新 v1.0 输入
<code>mode</code>	(已删除 - 自动检测)
<code>direct_prompt</code>	<code>prompt</code>
<code>override_prompt</code>	<code>prompt</code> 带 GitHub 变量
<code>custom_instructions</code>	<code>claude_args: --append-system-prompt</code>
<code>max_turns</code>	<code>claude_args: --max-turns</code>
<code>model</code>	<code>claude_args: --model</code>
<code>allowed_tools</code>	<code>claude_args: --allowedTools</code>
<code>disallowed_tools</code>	<code>claude_args: --disallowedTools</code>
<code>claude_env</code>	<code>settings</code> JSON 格式

前后示例

Beta 版本:

```
- uses: anthropics/claude-code-action@beta
  with:
    mode: "tag"
    direct_prompt: "Review this PR for security issues"
    anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }
    custom_instructions: "Follow our coding standards"
    max_turns: "10"
    model: "claude-sonnet-4-6"
```

GA 版本 (v1.0):

```
- uses: anthropics/claude-code-action@v1
  with:
    prompt: "Review this PR for security issues"
    anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }
    claude_args: |
      --append-system-prompt "Follow our coding standards"
      --max-turns 10
      --model claude-sonnet-4-6
```

Tip:

该 action 现在根据您的配置自动检测是在交互模式（响应 [@claude](#) 提及）还是自动化模式（立即使用提示运行）下运行。

示例用例

Claude Code GitHub Actions 可以帮助您完成各种任务。[examples 目录](#)包含针对不同场景的现成工作流。

基本工作流

```
name: Claude Code
on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
jobs:
  claude:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }}
          # Responds to @claude mentions in comments
```

使用 skills

```
name: Code Review
on:
  pull_request:
    types: [opened, synchronize]
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }}
          prompt: "Review this pull request for code quality, correctness, and
security. Analyze the diff, then post your findings as review comments."
          claude_args: "--max-turns 5"
```

使用提示的自定义自动化

```
name: Daily Report
on:
  schedule:
    - cron: "0 9 * * *"
jobs:
  report:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }
          prompt: "Generate a summary of yesterday's commits and open issues"
          claude_args: "--model opus"
```

常见用例

在 issue 或 PR 评论中：

```
@claude implement this feature based on the issue description
@claude how should I implement user authentication for this endpoint?
@claude fix the TypeError in the user dashboard component
```

Claude 将自动分析上下文并做出适当的响应。

最佳实践

CLAUDE.md 配置

在您的仓库根目录创建一个 **CLAUDE.md** 文件来定义代码风格指南、审查标准、项目特定规则和首选模式。此文件指导 Claude 对您的项目标准的理解。

安全考虑

Warning:

永远不要直接将 API 密钥提交到您的仓库。

有关全面的安全指导，包括权限、身份验证和最佳实践，请参阅 [Claude Code Action 安全文档](#)。

始终为 API 密钥使用 GitHub Secrets：

- 将您的 API 密钥添加为名为 `ANTHROPIC_API_KEY` 的仓库密钥
- 在工作流中引用它：`anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }`
- 将 action 权限限制为仅必要的权限
- 在合并前审查 Claude 的建议

始终使用 GitHub Secrets（例如，`${ secrets.ANTHROPIC_API_KEY }`）而不是直接在工作流文件中硬编码 API 密钥。

优化性能

使用 issue 模板提供上下文，保持您的 `CLAUDE.md` 简洁和专注，并为您的工作流配置适当的超时。

CI 成本

使用 Claude Code GitHub Actions 时，请注意相关成本：

GitHub Actions 成本：

- Claude Code 在 GitHub 托管的运行器上运行，这会消耗您的 GitHub Actions 分钟数
- 有关详细的定价和分钟限制，请参阅 [GitHub 的计费文档](#)

API 成本：

- 每次 Claude 交互都会根据提示和响应的长度消耗 API 令牌
- 令牌使用量因任务复杂性和代码库大小而异
- 有关当前令牌费率，请参阅 [Claude 的定价页面](#)

成本优化提示：

- 使用特定的 `@claude` 命令来减少不必要的 API 调用
- 在 `claude_args` 中配置适当的 `--max-turns` 以防止过度迭代
- 设置工作流级别的超时以避免失控的作业
- 考虑使用 GitHub 的并发控制来限制并行运行

配置示例

Claude Code Action v1 使用统一参数简化了配置：

```
- uses: anthropics/claude-code-action@v1
  with:
    anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }
    prompt: "Your instructions here" # Optional
    claude_args: "--max-turns 5" # Optional CLI arguments
```

关键功能：

- **统一提示界面** - 对所有说明使用 `prompt`
- **Skills** - 直接从提示调用已安装的 `skills`
- **CLI 传递** - 通过 `claude_args` 的任何 Claude Code CLI 参数
- **灵活的触发器** - 适用于任何 GitHub 事件

访问 [examples 目录](#) 获取完整的工作流文件。

Tip:

当响应 issue 或 PR 评论时，Claude 会自动响应 @claude 提及。对于其他事件，使用 `prompt` 参数提供说明。

使用 AWS Bedrock 和 Google Vertex AI

对于企业环境，您可以将 Claude Code GitHub Actions 与您自己的云基础设施一起使用。这种方法让您可以控制数据驻留和计费，同时保持相同的功能。

前置条件

在使用云提供商设置 Claude Code GitHub Actions 之前，您需要：

对于 Google Cloud Vertex AI：

1. 启用了 Vertex AI 的 Google Cloud 项目
2. 为 GitHub Actions 配置的工作负载身份联合
3. 具有所需权限的服务账户
4. GitHub 应用（推荐）或使用默认 GITHUB_TOKEN

对于 AWS Bedrock：

1. 启用了 Amazon Bedrock 的 AWS 账户
2. 在 AWS 中配置的 GitHub OIDC 身份提供商
3. 具有 Bedrock 权限的 IAM 角色

4. GitHub 应用（推荐）或使用默认 GITHUB_TOKEN

Step 1: 创建自定义 GitHub 应用（推荐用于第三方提供商）

为了在使用 Vertex AI 或 Bedrock 等第三方提供商时获得最佳控制和安全性，我们建议创建您自己的 GitHub 应用：

1. 转到 <https://github.com/settings/apps/new>

2. 填写基本信息：

- **GitHub App 名称：**选择唯一的名称（例如，“YourOrg Claude Assistant”）
- **主页 URL：**您的组织网站或仓库 URL

1. 配置应用设置：

- **Webhooks：**取消选中” Active”（此集成不需要）

1. 设置所需的权限：

- **仓库权限：**
- Contents：读写
- Issues：读写
- Pull requests：读写

1. 点击” Create GitHub App”

2. 创建后，点击” Generate a private key” 并保存下载的 `.pem` 文件

3. 从应用设置页面记下您的应用 ID

4. 将应用安装到您的仓库：

- 从您的应用设置页面，点击左侧边栏中的” Install App”
- 选择您的账户或组织
- 选择” Only select repositories” 并选择特定仓库
- 点击” Install”

1. 将私钥添加为仓库密钥：

- 转到您的仓库的 Settings → Secrets and variables → Actions
- 创建一个名为 `APP_PRIVATE_KEY` 的新密钥，内容为 `.pem` 文件的内容

1. 将应用 ID 添加为密钥：

- 创建一个名为 `APP_ID` 的新密钥，值为您的 GitHub 应用的 ID

Note:

此应用将与 [actions/create-github-app-token](#) action 一起使用，以在您的工作流中生成身份验证令牌。

Claude API 的替代方案或如果您不想设置自己的 Github 应用： 使用官方 Anthropic 应用：

1. 从以下位置安装：<https://github.com/apps/claude>
2. 无需额外的身份验证配置

Step 2: 配置云提供商身份验证

选择您的云提供商并设置安全身份验证：

配置 AWS 以允许 GitHub Actions 安全地进行身份验证，而无需存储凭证。

安全说明： 使用特定于仓库的配置并仅授予最少所需的权限。

所需设置：

1. 启用 Amazon Bedrock：

- 请求在 Amazon Bedrock 中访问 Claude 模型
- 对于跨区域模型，请在所有必需的区域中请求访问

1. 设置 GitHub OIDC 身份提供商：

- 提供商 URL：`https://token.actions.githubusercontent.com`
- 受众：`sts.amazonaws.com`

1. 为 GitHub Actions 创建 IAM 角色：

- 受信任的实体类型：Web 身份
- 身份提供商：`token.actions.githubusercontent.com`
- 权限：`AmazonBedrockFullAccess` 策略
- 为您的特定仓库配置信任策略

所需值：

设置后，您需要：

- `AWS_ROLE_TO_ASSUME`：您创建的 IAM 角色的 ARN

Tip:

OIDC 比使用静态 AWS 访问密钥更安全，因为凭证是临时的并自动轮换。

有关详细的 OIDC 设置说明，请参阅 [AWS 文档](#)。

配置 Google Cloud 以允许 GitHub Actions 安全地进行身份验证，而无需存储凭证。

安全说明： 使用特定于仓库的配置并仅授予最少所需的权限。

所需设置：

1. 在您的 Google Cloud 项目中启用 API：

- IAM Credentials API
- Security Token Service (STS) API
- Vertex AI API

1. 创建工作负载身份联合资源：

- 创建工作负载身份池
- 添加 GitHub OIDC 提供商，具有：
- 发行者：`https://token.actions.githubusercontent.com`
- 仓库和所有者的属性映射
- **安全建议：** 使用特定于仓库的属性条件

1. 创建服务账户：

- 仅授予 `Vertex AI User` 角色
- **安全建议：** 为每个仓库创建专用服务账户

1. 配置 IAM 绑定：

- 允许工作负载身份池模拟服务账户
- **安全建议：** 使用特定于仓库的主体集

所需值：

设置后，您需要：

- **GCP_WORKLOAD_IDENTITY_PROVIDER：** 完整的提供商资源名称
- **GCP_SERVICE_ACCOUNT：** 服务账户电子邮件地址

Tip:

工作负载身份联合消除了对可下载服务账户密钥的需求，提高了安全性。

有关详细的设置说明，请参阅 [Google Cloud 工作负载身份联合文档](#)。

Step 3: 添加所需的密钥

将以下密钥添加到您的仓库（Settings → Secrets and variables → Actions）：

对于 Claude API（直接）：

1. 对于 API 身份验证：

- `ANTHROPIC_API_KEY`：您的 Claude API 密钥，来自 console.anthropic.com

1. 对于 GitHub 应用（如果使用您自己的应用）：

- `APP_ID`：您的 GitHub 应用的 ID
- `APP_PRIVATE_KEY`：私钥 (.pem) 内容

对于 Google Cloud Vertex AI

1. 对于 GCP 身份验证：

- `GCP_WORKLOAD_IDENTITY_PROVIDER`
- `GCP_SERVICE_ACCOUNT`

1. 对于 GitHub 应用（如果使用您自己的应用）：

- `APP_ID`：您的 GitHub 应用的 ID
- `APP_PRIVATE_KEY`：私钥 (.pem) 内容

对于 AWS Bedrock

1. 对于 AWS 身份验证：

- `AWS_ROLE_TO_ASSUME`

1. 对于 GitHub 应用（如果使用您自己的应用）：

- `APP_ID`：您的 GitHub 应用的 ID
- `APP_PRIVATE_KEY`：私钥 (.pem) 内容

Step 4: 创建工作流文件

创建与您的云提供商集成的 GitHub Actions 工作流文件。下面的示例显示了 AWS Bedrock 和 Google Vertex AI 的完整配置：

前置条件：

- 启用了 AWS Bedrock 访问权限，具有 Claude 模型权限
- GitHub 在 AWS 中配置为 OIDC 身份提供商
- 具有 Bedrock 权限的 IAM 角色，信任 GitHub Actions

所需的 GitHub 密钥：

密钥名称	描述
AWS_ROLE_TO_ASSUME	Bedrock 访问的 IAM 角色的 ARN
APP_ID	您的 GitHub 应用 ID（来自应用设置）
APP_PRIVATE_KEY	您为 GitHub 应用生成的私钥

```

name: Claude PR Action

permissions:
  contents: write
  pull-requests: write
  issues: write
  id-token: write

on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
  issues:
    types: [opened, assigned]

jobs:
  claude-pr:
    if: |
      (github.event_name == 'issue_comment' && contains(github.event.comment.body,
        '@claude')) ||
      (github.event_name == 'pull_request_review_comment' &&
        contains(github.event.comment.body, '@claude')) ||
      (github.event_name == 'issues' && contains(github.event.issue.body,
        '@claude'))
    runs-on: ubuntu-latest
    env:
      AWS_REGION: us-west-2
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Generate GitHub App token
        id: app-token
        uses: actions/create-github-app-token@v2
        with:
          app-id: ${ secrets.APP_ID }
          private-key: ${ secrets.APP_PRIVATE_KEY }

```

```
- name: Configure AWS Credentials (OIDC)
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: ${ secrets.AWS_ROLE_TO_ASSUME }
    aws-region: us-west-2

- uses: anthropics/claude-code-action@v1
  with:
    github_token: ${ steps.app-token.outputs.token }
    use_bedrock: "true"
    claude_args: '--model us.anthropic.claude-sonnet-4-6 --max-turns 10'
```

Tip:
Bedrock 的模型 ID 格式包括区域前缀（例如，`us.anthropic.claude-sonnet-4-6`）。

前置条件：

- 在您的 GCP 项目中启用了 Vertex AI API
- 为 GitHub 配置了工作负载身份联合
- 具有 Vertex AI 权限的服务账户

所需的 GitHub 密钥：

密钥名称	描述
GCP_WORKLOAD_IDENTITY_PROVIDER	工作负载身份提供商资源名称
GCP_SERVICE_ACCOUNT	具有 Vertex AI 访问权限的服务账户电子邮件
APP_ID	您的 GitHub 应用 ID（来自应用设置）
APP_PRIVATE_KEY	您为 GitHub 应用生成的私钥

```

name: Claude PR Action

permissions:
  contents: write
  pull-requests: write
  issues: write
  id-token: write

on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
  issues:
    types: [opened, assigned]

jobs:
  claude-pr:
    if: |
      (github.event_name == 'issue_comment' && contains(github.event.comment.body,
        '@claude')) ||
      (github.event_name == 'pull_request_review_comment' &&
        contains(github.event.comment.body, '@claude')) ||
      (github.event_name == 'issues' && contains(github.event.issue.body,
        '@claude'))
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Generate GitHub App token
        id: app-token
        uses: actions/create-github-app-token@v2
        with:
          app-id: ${ secrets.APP_ID }
          private-key: ${ secrets.APP_PRIVATE_KEY }

      - name: Authenticate to Google Cloud

```

```

    id: auth
    uses: google-github-actions/auth@v2
    with:
      workload_identity_provider: $
    {{ secrets.GCP_WORKLOAD_IDENTITY_PROVIDER }}
    service_account: ${{ secrets.GCP_SERVICE_ACCOUNT }}

- uses: anthropics/claude-code-action@v1
  with:
    github_token: ${{ steps.app-token.outputs.token }}
    trigger_phrase: "@claude"
    use_vertex: "true"
    claude_args: '--model claude-sonnet-4@20250514 --max-turns 10'
  env:
    ANTHROPIC_VERTEX_PROJECT_ID: ${{ steps.auth.outputs.project_id }}
    CLOUD_ML_REGION: us-east5
    VERTEX_REGION_CLAUDE_3_7_SONNET: us-east5

```

Tip:

项目 ID 从 Google Cloud 身份验证步骤自动检索，因此您无需对其进行硬编码。

故障排除

Claude 不响应 @claude 命令

验证 GitHub 应用是否正确安装，检查工作流是否已启用，确保 API 密钥在仓库密钥中设置，并确认评论包含 @claude（不是 /claude）。

CI 不在 Claude 的提交上运行

确保您使用的是 GitHub 应用或自定义应用（不是 Actions 用户），检查工作流触发器是否包含必要的事件，并验证应用权限是否包括 CI 触发器。

身份验证错误

确认 API 密钥有效且具有足够的权限。对于 Bedrock/Vertex，检查凭证配置并确保密钥在工作流中正确命名。

高级配置

Action 参数

Claude Code Action v1 使用简化的配置：

参数	描述	必需
<code>prompt</code>	Claude 的说明（纯文本或 skill 名称）	否*
<code>claude_args</code>	传递给 Claude Code 的 CLI 参数	否
<code>anthropic_api_key</code>	Claude API 密钥	是**
<code>github_token</code>	用于 API 访问的 GitHub 令牌	否
<code>trigger_phrase</code>	自定义触发短语（默认：“@claude”）	否
<code>use_bedrock</code>	使用 AWS Bedrock 而不是 Claude API	否
<code>use_vertex</code>	使用 Google Vertex AI 而不是 Claude API	否

*提示是可选的 - 当对 issue/PR 评论省略时，Claude 响应触发短语

**对于直接 Claude API 是必需的，对于 Bedrock/Vertex 不是必需的

传递 CLI 参数

`claude_args` 参数接受任何 Claude Code CLI 参数：

```
claude_args: "--max-turns 5 --model claude-sonnet-4-6 --mcp-config /path/to/config.json"
```

常见参数：

- `--max-turns`：最大对话轮数（默认：10）
- `--model`：要使用的模型（例如，`claude-sonnet-4-6`）
- `--mcp-config`：MCP 配置的路径
- `--allowed-tools`：允许的工具的逗号分隔列表
- `--debug`：启用调试输出

替代集成方法

虽然 `/install-github-app` 命令是推荐的方法，但您也可以：

- **自定义 GitHub 应用：**对于需要品牌用户名或自定义身份验证流的组织。创建您自己的 GitHub 应用，具有所需的权限（contents、issues、pull requests），并使用 `actions/create-github-app-token` action 在您的工作流中生成令牌。
- **手动 GitHub Actions：**直接工作流配置以获得最大灵活性
- **MCP 配置：**Model Context Protocol 服务器的动态加载

有关身份验证、安全和高级配置的详细指南，请参阅 [Claude Code Action 文档](#)。

自定义 Claude 的行为

您可以通过两种方式配置 Claude 的行为：

1. **CLAUDE.md：**在您的仓库根目录的 `CLAUDE.md` 文件中定义编码标准、审查标准和项目特定规则。Claude 在创建 PR 和响应请求时将遵循这些指南。查看我们的 [Memory 文档](#) 了解更多详情。
2. **自定义提示：**在工作流文件中使用 `prompt` 参数提供工作流特定的说明。这允许您为不同的工作流或任务自定义 Claude 的行为。

Claude 在创建 PR 和响应请求时将遵循这些指南。

Claude Code GitLab CI/CD

了解如何将 Claude Code 集成到您的 GitLab CI/CD 开发工作流中

Info:

Claude Code for GitLab CI/CD 目前处于测试阶段。随着我们完善体验，功能和特性可能会发生变化。

此集成由 GitLab 维护。如需支持，请参阅以下 [GitLab issue](#)。

Note:

此集成基于 [Claude Code CLI and Agent SDK](#) 构建，可在您的 CI/CD 作业和自定义自动化工作流中以编程方式使用 Claude。

为什么在 GitLab 中使用 Claude Code？

- **即时 MR 创建：**描述您的需求，Claude 会提议一个完整的 MR，包含更改和说明
- **自动化实现：**使用单个命令或提及将问题转化为可工作的代码
- **项目感知：**Claude 遵循您的 `CLAUDE.md` 指南和现有代码模式
- **简单设置：**向 `.gitlab-ci.yml` 添加一个作业和一个掩码 CI/CD 变量
- **企业就绪：**选择 Claude API、AWS Bedrock 或 Google Vertex AI 以满足数据驻留和采购需求
- **默认安全：**在您的 GitLab runners 中运行，具有您的分支保护和批准

工作原理

Claude Code 使用 GitLab CI/CD 在隔离的作业中运行 AI 任务，并通过 MR 将结果提交回来：

1. **事件驱动的编排：**GitLab 监听您选择的触发器（例如，在问题、MR 或审查线程中提及 `@claude` 的评论）。该作业从线程和存储库收集上下文，从该输入构建提示，并运行 Claude Code。
2. **提供商抽象：**使用适合您环境的提供商：
 - Claude API (SaaS)

- AWS Bedrock（基于 IAM 的访问、跨区域选项）
 - Google Vertex AI（GCP 原生、Workload Identity Federation）
3. **沙箱执行**：每次交互都在具有严格网络和文件系统规则的容器中运行。Claude Code 强制执行工作区范围的权限以限制写入。每项更改都通过 MR 流动，以便审查者可以看到差异，批准仍然适用。

选择区域端点以降低延迟并满足数据主权要求，同时使用现有的云协议。

Claude 可以做什么？

Claude Code 支持强大的 CI/CD 工作流，改变您处理代码的方式：

- 从问题描述或评论创建和更新 MR
- 分析性能回归并提议优化
- 直接在分支中实现功能，然后打开 MR
- 修复由测试或评论识别的错误和回归
- 响应后续评论以迭代所请求的更改

设置

快速设置

最快的入门方式是向您的 `.gitlab-ci.yml` 添加一个最小作业，并将您的 API 密钥设置为掩码变量。

1. 添加掩码 CI/CD 变量

- 转到 **Settings** → **CI/CD** → **Variables**
- 添加 `ANTHROPIC_API_KEY`（掩码，根据需要保护）

2. 向 `.gitlab-ci.yml` 添加 Claude 作业

```

stages:
  - ai

claude:
  stage: ai
  image: node:24-alpine3.21
  # 调整规则以适应您想要触发作业的方式:
  # - 手动运行
  # - 合并请求事件
  # - 当评论包含 '@claude' 时的 web/API 触发
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  variables:
    GIT_STRATEGY: fetch
  before_script:
    - apk update
    - apk add --no-cache git curl bash
    - curl -fsSL https://claude.ai/install.sh | bash
  script:
    # 可选: 如果您的设置提供了 GitLab MCP 服务器, 请启动它
    - /bin/gitlab-mcp-server || true
    # 通过 web/API 触发器使用 AI_FLOW_* 变量调用时使用上下文有效负载
    - echo "$AI_FLOW_INPUT for $AI_FLOW_CONTEXT on $AI_FLOW_EVENT"
    - >

    claude
    -p "${AI_FLOW_INPUT:-'Review this MR and implement the requested changes'}"
    --permission-mode acceptEdits
    --allowedTools "Bash Read Edit Write mcp_gitlab"
    --debug

```

添加作业和您的 `ANTHROPIC_API_KEY` 变量后, 通过从 **CI/CD** → **Pipelines** 手动运行作业进行测试, 或从 MR 触发它, 让 Claude 在分支中提议更新并在需要时打开 MR。

Note:

要改为在 AWS Bedrock 或 Google Vertex AI 上运行而不是 Claude API, 请参阅下面的 [Using with AWS Bedrock & Google Vertex AI](#) 部分, 了解身份验证和环境设置。

手动设置（建议用于生产）

如果您更喜欢更受控的设置或需要企业提供商：

1. 配置提供商访问：

- **Claude API**：创建并将 `ANTHROPIC_API_KEY` 存储为掩码 CI/CD 变量
- **AWS Bedrock**：Configure GitLab → AWS OIDC 并为 Bedrock 创建 IAM 角色
- **Google Vertex AI**：Configure Workload Identity Federation for GitLab → GCP

2. 为 GitLab API 操作添加项目凭证：

- 默认使用 `CI_JOB_TOKEN`，或创建具有 `api` 范围的项目访问令牌
- 如果使用 PAT，则存储为 `GITLAB_ACCESS_TOKEN`（掩码）

3. 向 `.gitlab-ci.yml` 添加 Claude 作业（请参阅下面的示例）

4. （可选）启用提及驱动的触发器：

- 为” Comments (notes) “添加项目 webhook 到您的事件监听器（如果您使用）
- 当评论包含 `@claude` 时，让监听器使用 `AI_FLOW_INPUT` 和 `AI_FLOW_CONTEXT` 等变量调用管道触发 API

示例用例

将问题转化为 MR

在问题评论中：

```
@claude implement this feature based on the issue description
```

Claude 分析问题和代码库，在分支中编写更改，并打开 MR 供审查。

获取实现帮助

在 MR 讨论中：

```
@claude suggest a concrete approach to cache the results of this API call
```

Claude 提议更改，添加具有适当缓存的代码，并更新 MR。

快速修复错误

在问题或 MR 评论中：

```
@claude fix the TypeError in the user dashboard component
```

Claude 定位错误，实现修复，并更新分支或打开新 MR。

使用 AWS Bedrock 和 Google Vertex AI

对于企业环境，您可以在云基础设施上完全运行 Claude Code，具有相同的开发者体验。

AWS Bedrock

前置条件

在使用 AWS Bedrock 设置 Claude Code 之前，您需要：

1. 具有对所需 Claude 模型的 Amazon Bedrock 访问权限的 AWS 账户
2. 在 AWS IAM 中配置为 OIDC 身份提供商的 GitLab
3. 具有 Bedrock 权限和信任策略的 IAM 角色，限制为您的 GitLab 项目/refs
4. 用于角色假设的 GitLab CI/CD 变量：

- `AWS_ROLE_TO_ASSUME`（角色 ARN）
- `AWS_REGION`（Bedrock 区域）

设置说明

配置 AWS 以允许 GitLab CI 作业通过 OIDC 假设 IAM 角色（无静态密钥）。

必需的设置：

1. 启用 Amazon Bedrock 并请求访问您的目标 Claude 模型
2. 如果尚未存在，为 GitLab 创建 IAM OIDC 提供商
3. 创建由 GitLab OIDC 提供商信任的 IAM 角色，限制为您的项目和受保护的 refs
4. 为 Bedrock 调用 API 附加最小权限

需要存储在 CI/CD 变量中的必需值：

- `AWS_ROLE_TO_ASSUME`
- `AWS_REGION`

在 Settings → CI/CD → Variables 中添加变量：

```
## 对于 AWS Bedrock :
- AWS_ROLE_TO_ASSUME
- AWS_REGION
```

使用上面的 AWS Bedrock 作业示例在运行时交换 GitLab 作业令牌以获取临时 AWS 凭证。

Google Vertex AI

前置条件

在使用 Google Vertex AI 设置 Claude Code 之前，您需要：

1. 具有以下条件的 Google Cloud 项目：
 - 启用了 Vertex AI API
 - 配置了 Workload Identity Federation 以信任 GitLab OIDC
1. 仅具有所需 Vertex AI 角色的专用服务账户
2. 用于 WIF 的 GitLab CI/CD 变量：
 - `GCP_WORKLOAD_IDENTITY_PROVIDER`（完整资源名称）
 - `GCP_SERVICE_ACCOUNT`（服务账户电子邮件）

设置说明

配置 Google Cloud 以允许 GitLab CI 作业通过 Workload Identity Federation 模拟服务账户。

必需的设置：

1. 启用 IAM Credentials API、STS API 和 Vertex AI API
2. 为 GitLab OIDC 创建 Workload Identity Pool 和提供商
3. 创建具有 Vertex AI 角色的专用服务账户
4. 授予 WIF 主体权限以模拟服务账户

需要存储在 CI/CD 变量中的必需值：

- `GCP_WORKLOAD_IDENTITY_PROVIDER`
- `GCP_SERVICE_ACCOUNT`

在 Settings → CI/CD → Variables 中添加变量：

```
## 对于 Google Vertex AI:
- GCP_WORKLOAD_IDENTITY_PROVIDER
- GCP_SERVICE_ACCOUNT
- CLOUD_ML_REGION (例如, us-east5)
```

使用上面的 Google Vertex AI 作业示例在不存储密钥的情况下进行身份验证。

配置示例

以下是您可以适配到管道的现成代码片段。

基本 .gitlab-ci.yml (Claude API)

```
stages:
  - ai

claude:
  stage: ai
  image: node:24-alpine3.21
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  variables:
    GIT_STRATEGY: fetch
  before_script:
    - apk update
    - apk add --no-cache git curl bash
    - curl -fsSL https://claude.ai/install.sh | bash
  script:
    - /bin/gitlab-mcp-server || true
    - >

    claude
    -p "${AI_FLOW_INPUT:-'Summarize recent changes and suggest improvements'}"
    --permission-mode acceptEdits
    --allowedTools "Bash Read Edit Write mcp_gitlab"
    --debug

# Claude Code 将使用 CI/CD 变量中的 ANTHROPIC_API_KEY
```


AWS Bedrock 作业示例 (OIDC)

前置条件：

- 启用了 Amazon Bedrock 并可访问您选择的 Claude 模型
- 在 AWS 中配置了 GitLab OIDC，具有信任您的 GitLab 项目和 refs 的角色
- 具有 Bedrock 权限的 IAM 角色（建议最小权限）

必需的 CI/CD 变量：

- `AWS_ROLE_TO_ASSUME`：用于 Bedrock 访问的 IAM 角色的 ARN
- `AWS_REGION`：Bedrock 区域（例如，`us-west-2`）

```

claude-bedrock:
  stage: ai
  image: node:24-alpine3.21
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
  before_script:
    - apk add --no-cache bash curl jq git python3 py3-pip
    - pip install --no-cache-dir awscli
    - curl -fsSL https://claude.ai/install.sh | bash
    # 交换 GitLab OIDC 令牌以获取 AWS 凭证
    - export AWS_WEB_IDENTITY_TOKEN_FILE="${CI_JOB_JWT_FILE:-/tmp/oidc_token}"
    - if [ -n "${CI_JOB_JWT_V2}" ]; then printf "%s" "${CI_JOB_JWT_V2}" >
"$AWS_WEB_IDENTITY_TOKEN_FILE"; fi
    - >
      aws sts assume-role-with-web-identity
      --role-arn "${AWS_ROLE_TO_ASSUME}"
      --role-session-name "gitlab-claude-$(date +%s)"
      --web-identity-token "file://$AWS_WEB_IDENTITY_TOKEN_FILE"
      --duration-seconds 3600 > /tmp/aws_creds.json
    - export AWS_ACCESS_KEY_ID="$(jq -r .Credentials.AccessKeyId /tmp/
aws_creds.json)"
    - export AWS_SECRET_ACCESS_KEY="$(jq -r .Credentials.SecretAccessKey /tmp/
aws_creds.json)"
    - export AWS_SESSION_TOKEN="$(jq -r .Credentials.SessionToken /tmp/
aws_creds.json)"
  script:
    - /bin/gitlab-mcp-server || true
    - >
      claude
      -p "${AI_FLOW_INPUT:-'Implement the requested changes and open an MR'}"
      --permission-mode acceptEdits
      --allowedTools "Bash Read Edit Write mcp__gitlab"
      --debug
  variables:
    AWS_REGION: "us-west-2"

```

Note:

Bedrock 的模型 ID 包括特定于区域的前缀（例如，`us.anthropic.claude-sonnet-4-6`）。

如果您的 workflow 支持，通过您的作业配置或提示传递所需的模型。

Google Vertex AI 作业示例（Workload Identity Federation）

前置条件：

- 在您的 GCP 项目中启用了 Vertex AI API
- 配置了 Workload Identity Federation 以信任 GitLab OIDC
- 具有 Vertex AI 权限的服务账户

必需的 CI/CD 变量：

- `GCP_WORKLOAD_IDENTITY_PROVIDER`：完整的提供商资源名称
- `GCP_SERVICE_ACCOUNT`：服务账户电子邮件
- `CLOUD_ML_REGION`：Vertex 区域（例如，`us-east5`）

```

claude-vertex:
  stage: ai
  image: gcr.io/google.com/cloudsdktool/google-cloud-cli:slim
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
  before_script:
    - apt-get update && apt-get install -y git && apt-get clean
    - curl -fsSL https://claude.ai/install.sh | bash
    # 通过 WIF 向 Google Cloud 进行身份验证 (无下载的密钥)
    - >
      gcloud auth login --cred-file=<(cat <<EOF
      {
        "type": "external_account",
        "audience": "${GCP_WORKLOAD_IDENTITY_PROVIDER}",
        "subject_token_type": "urn:ietf:params:oauth:token-type:jwt",
        "service_account_impersonation_url": "https://
iamcredentials.googleapis.com/v1/projects/-/serviceAccounts/${
GCP_SERVICE_ACCOUNT}:generateAccessToken",
        "token_url": "https://sts.googleapis.com/v1/token"
      }
      EOF
    )
    - gcloud config set project "$(gcloud projects list --
format='value(projectId)' --filter="name:${CI_PROJECT_NAMESPACE}" | head -n1)" ||
true
  script:
    - /bin/gitlab-mcp-server || true
    - >
      CLOUD_ML_REGION="${CLOUD_ML_REGION:-us-east5}"
      claude
      -p "${AI_FLOW_INPUT:-'Review and update code as requested'}"
      --permission-mode acceptEdits
      --allowedTools "Bash Read Edit Write mcp__gitlab"
      --debug
  variables:
    CLOUD_ML_REGION: "us-east5"

```

Note:

使用 Workload Identity Federation，您无需存储服务账户密钥。使用特定于存储库的信任条件和最小权限服务账户。

最佳实践

CLAUDE.md 配置

在存储库根目录创建 `CLAUDE.md` 文件以定义编码标准、审查标准和项目特定规则。Claude 在运行期间读取此文件，并在提议更改时遵循您的约定。

安全考虑

永远不要将 API 密钥或云凭证提交到您的存储库。始终使用 GitLab CI/CD 变量：

- 将 `ANTHROPIC_API_KEY` 添加为掩码变量（如果需要，保护它）
- 尽可能使用提供商特定的 OIDC（无长期密钥）
- 限制作业权限和网络出口
- 像审查任何其他贡献者一样审查 Claude 的 MR

优化性能

- 保持 `CLAUDE.md` 专注和简洁
- 提供清晰的问题/MR 描述以减少迭代
- 配置合理的作业超时以避免失控运行
- 在可能的情况下在 runners 中缓存 npm 和包安装

CI 成本

在 GitLab CI/CD 中使用 Claude Code 时，请注意相关成本：

- **GitLab Runner 时间：**
 - Claude 在您的 GitLab runners 上运行并消耗计算分钟数
 - 有关详细信息，请参阅您的 GitLab 计划的 runner 计费
- **API 成本：**
 - 每次 Claude 交互根据提示和响应大小消耗令牌
 - 令牌使用因任务复杂性和代码库大小而异
 - 有关详细信息，请参阅 [Anthropic 定价](#)

• 成本优化提示：

- 使用特定的 `@claude` 命令以减少不必要的轮次
- 设置适当的 `max_turns` 和作业超时值
- 限制并发以控制并行运行

安全和治理

- 每个作业都在具有受限网络访问的隔离容器中运行
- Claude 的更改通过 MR 流动，以便审查者可以看到每个差异
- 分支保护和批准规则适用于 AI 生成的代码
- Claude Code 使用工作区范围的权限来限制写入
- 成本保持在您的控制下，因为您带来自己的提供商凭证

故障排除

Claude 不响应 `@claude` 命令

- 验证您的管道是否被触发（手动、MR 事件或通过注释事件监听器/webhook）
- 确保 CI/CD 变量（`ANTHROPIC_API_KEY` 或云提供商设置）存在且未掩码
- 检查评论是否包含 `@claude`（不是 `/claude`）以及您的提及触发器是否已配置

作业无法写入评论或打开 MR

- 确保 `CI_JOB_TOKEN` 对项目具有足够的权限，或使用具有 `api` 范围的项目访问令牌
- 检查 `mcp_gitlab` 工具是否在 `--allowedTools` 中启用
- 确认作业在 MR 的上下文中运行或通过 `AI_FLOW_*` 变量有足够的上下文

身份验证错误

- 对于 Claude API：确认 `ANTHROPIC_API_KEY` 有效且未过期
- 对于 Bedrock/Vertex：验证 OIDC/WIF 配置、角色模拟和密钥名称；确认区域和模型可用性

高级配置

常见参数和变量

Claude Code 支持这些常用输入：

- `prompt` / `prompt_file`：内联提供说明（`-p`）或通过文件
- `max_turns`：限制来回迭代的次数

- `timeout_minutes`：限制总执行时间
- `ANTHROPIC_API_KEY`：Claude API 所需（不用于 Bedrock/Vertex）
- 提供商特定的环境：`AWS_REGION`、Vertex 的项目/区域变量

Note:

确切的标志和参数可能因 `@anthropic-ai/claude-code` 的版本而异。在您的作业中运行 `claude --help` 以查看支持的选项。

自定义 Claude 的行为

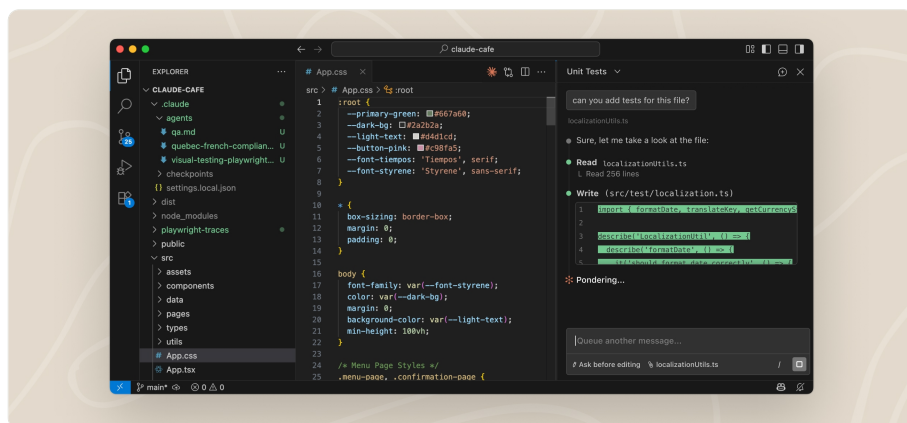
您可以通过两种主要方式指导 Claude：

1. **CLAUDE.md**：定义编码标准、安全要求和项目约定。Claude 在运行期间读取此文件并遵循您的规则。
2. **自定义提示**：通过作业中的 `prompt` / `prompt_file` 传递特定于任务的说明。为不同的作业使用不同的提示（例如，审查、实现、重构）。

Part 8: IDE & Platform Integration

在 VS Code 中使用 Claude Code

安装和配置 VS Code 的 Claude Code 扩展。获得 AI 编码协助，包括内联差异、@-提及、计划审查和快捷键。



VS Code 编辑器，右侧打开 Claude Code 扩展面板，显示与 Claude 的对话

VS Code 扩展为 Claude Code 提供了原生图形界面，直接集成到您的 IDE 中。这是在 VS Code 中使用 Claude Code 的推荐方式。

使用该扩展，您可以在接受 Claude 的计划之前审查和编辑它们、在进行编辑时自动接受、从您的选择中 @-提及具有特定行范围的文件、访问对话历史记录，以及在单独的选项卡或窗口中打开多个对话。

前置条件

安装前，请确保您拥有：

- VS Code 1.98.0 或更高版本
- Anthropic 账户（首次打开扩展时您将登录）。如果您使用第三方提供商（如 Amazon Bedrock 或 Google Vertex AI），请改为参阅[使用第三方提供商](#)。

Tip:

该扩展包括 CLI（命令行界面），您可以从 VS Code 的集成终端访问它以获得高级功能。有关详细信息，请参阅 [VS Code 扩展与 Claude Code CLI](#)。

安装扩展

点击您的 IDE 的链接以直接安装：

- [为 VS Code 安装](#)
- [为 Cursor 安装](#)

或者在 VS Code 中，按 **Cmd+Shift+X**（Mac）或 **Ctrl+Shift+X**（Windows/Linux）打开扩展视图，搜索“Claude Code”，然后点击**安装**。

Note:

如果安装后扩展没有出现，请重启 VS Code 或从命令面板运行“Developer: Reload Window”。

开始使用

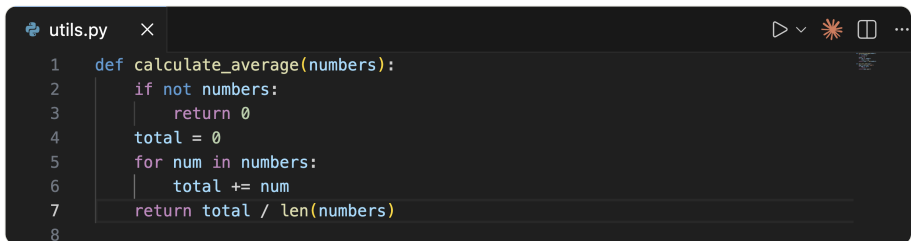
安装后，您可以通过 VS Code 界面开始使用 Claude Code：

Step 1: 打开 Claude Code 面板

在整个 VS Code 中，Spark 图标表示 Claude Code：



打开 Claude 的最快方式是点击**编辑器工具栏**（编辑器右上角）中的 Spark 图标。该图标仅在您打开文件时出现。



VS Code 编辑器显示编辑器工具栏中的 Spark 图标

打开 Claude Code 的其他方式：

- **活动栏：** 点击左侧边栏中的 Spark 图标打开会话列表。点击任何会话以将其作为完整编辑器选项卡打开，或开始新的会话。此图标在活动栏中始终可见。
- **命令面板：** `Cmd+Shift+P` (Mac) 或 `Ctrl+Shift+P` (Windows/Linux)，输入“Claude Code”，然后选择一个选项，如“在新选项卡中打开”
- **状态栏：** 点击窗口右下角的** Claude Code**。即使没有打开文件，这也有效。

首次打开面板时，会出现**学习 Claude Code** 检查清单。通过点击**显示给我**来完成每一项，或用 X 关闭它。要稍后重新打开它，请在 VS Code 设置中的扩展 → Claude Code 下取消选中**隐藏入门**。

您可以拖动 Claude 面板在 VS Code 中重新定位它。有关详细信息，请参阅[自定义您的工作流](#)。

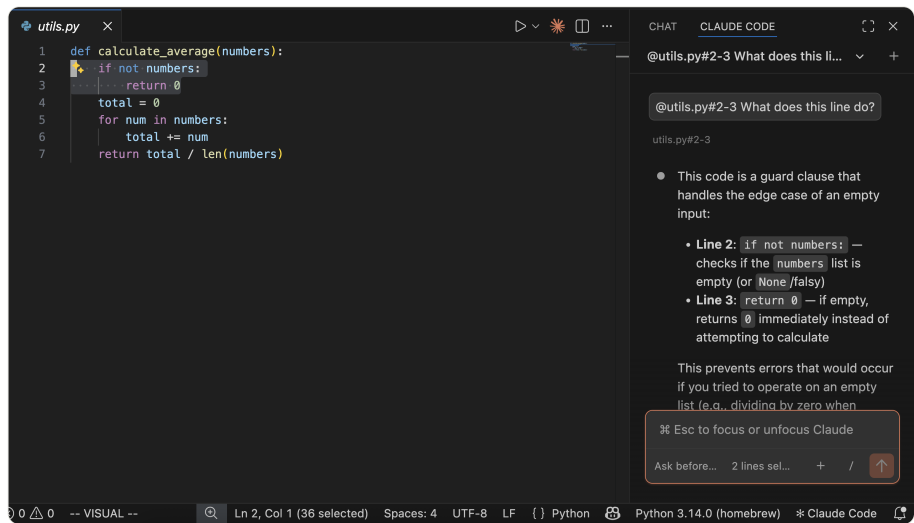
Step 2: 发送提示

要求 Claude 帮助您处理代码或文件，无论是解释某些内容的工作原理、调试问题还是进行更改。

Tip:

Claude 会自动看到您选择的文本。按 `Option+K` (Mac) / `Alt+K` (Windows/Linux) 也可以在您的提示中插入 @-提及引用 (如 `@file.ts#5-10`)。

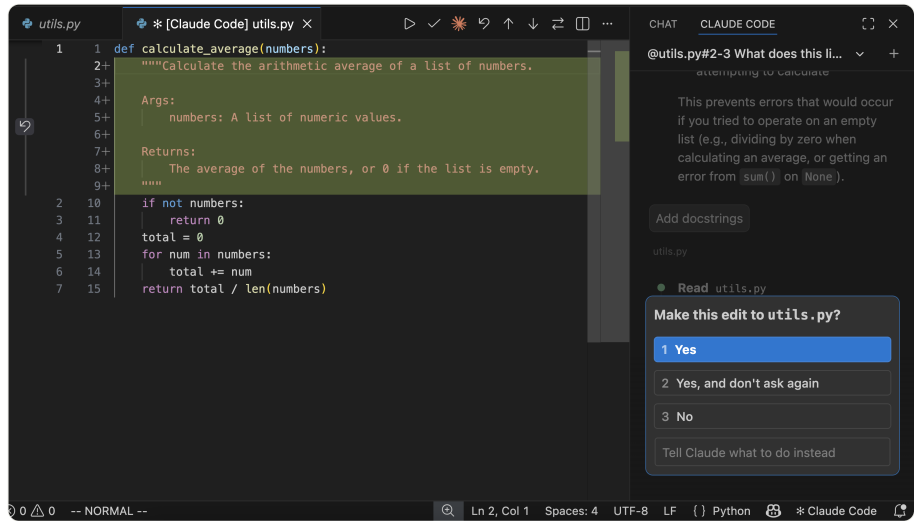
以下是询问文件中特定行的示例：



VS Code 编辑器，Python 文件中的第 2-3 行被选中，Claude Code 面板显示关于这些行的问题，带有 @-提及引用

Step 3: 审查更改

当 Claude 想要编辑文件时，它会显示原始内容和建议更改的并排比较，然后请求许可。您可以接受、拒绝或告诉 Claude 改为做什么。



VS Code 显示 Claude 建议更改的差异，带有权限提示，询问是否进行编辑

有关您可以使用 Claude Code 做什么的更多想法，请参阅[常见工作流](#)。

Tip:

从命令面板运行“Claude Code: Open Walkthrough”以获得基础知识的引导式教程。

使用提示框

提示框支持多个功能：

- **权限模式**：点击提示框底部的模式指示器以切换模式。在正常模式下，Claude 在每个操作前请求许可。在 Plan Mode 中，Claude 描述它将做什么，并在进行更改前等待批准。VS Code 会自动将计划作为完整的 markdown 文档打开，您可以添加内联注释以在 Claude 开始前提供反馈。在自动接受模式下，Claude 进行编辑而不询问。在 VS Code 设置中的 `claudeCode.initialPermissionMode` 下设置默认值。
- **命令菜单**：点击 `/` 或输入 `/` 以打开命令菜单。选项包括附加文件、切换模型、切换扩展思考和查看计划使用情况（`/usage`）。自定义部分提供对 MCP servers、hooks、memory、permissions 和 plugins 的访问。带有终端图标的项目在集成终端中打开。
- **上下文指示器**：提示框显示您使用了多少 Claude 的 context window。Claude 在需要时自动压缩，或者您可以手动运行 `/compact`。
- **扩展思考**：让 Claude 花更多时间推理复杂问题。通过命令菜单（`/`）切换它。有关详细信息，请参阅[扩展思考](#)。
- **多行输入**：按 `Shift+Enter` 添加新行而不发送。这也适用于问题对话框的“其他”自由文本输入。

引用文件和文件夹

使用 @-提及为 Claude 提供有关特定文件或文件夹的上下文。当您输入 @ 后跟文件或文件夹名称时，Claude 会读取该内容，并可以回答有关它的问题或对其进行更改。Claude Code 支持模糊匹配，因此您可以输入部分名称来找到您需要的内容：

```
> Explain the logic in @auth (fuzzy matches auth.js, AuthService.ts, etc.)
> What's in @src/components/ (include a trailing slash for folders)
```

对于大型 PDF，您可以要求 Claude 读取特定页面而不是整个文件：单个页面、范围（如第 1-10 页）或开放式范围（如第 3 页及以后）。

当您在编辑器中选择文本时，Claude 可以自动看到您突出显示的代码。提示框页脚显示选择了多少行。按 `Option+K`（Mac）/ `Alt+K`（Windows/Linux）插入带有文件路径和行号的 @-提及（例如 `@app.ts#5-10`）。点击选择指示器以切换 Claude 是否可以看到您突出显示的文本 - 眼睛斜线图标表示选择对 Claude 隐藏。

您也可以在将文件拖到提示框时按住 **Shift** 以将它们添加为附件。点击任何附件上的 X 以将其从上下文中删除。

恢复过去的对话

点击 Claude Code 面板顶部的下拉菜单以访问您的对话历史记录。您可以按关键字搜索或按时间浏览（今天、昨天、过去 7 天等）。点击任何对话以使用完整的消息历史记录恢复它。将鼠标悬停在会话上以显示重命名和删除操作：重命名以给它一个描述性标题，或删除以将其从列表中删除。有关恢复会话的更多信息，请参阅[常见工作流](#)。

从 Claude.ai 恢复远程会话

如果您使用[网络上的 Claude Code](#)，您可以直接在 VS Code 中恢复这些远程会话。这需要您使用 **Claude.ai Subscription** 登录，而不是 Anthropic Console。

Step 1: 打开过去的对话

点击 Claude Code 面板顶部的**过去的对话**下拉菜单。

Step 2: 选择远程选项卡

对话框显示两个选项卡：本地和远程。点击**远程**以查看来自 claude.ai 的会话。

Step 3: 选择要恢复的会话

浏览或搜索您的远程会话。点击任何会话以下载它并在本地继续对话。

Note:

只有使用 GitHub 存储库启动的网络会话才会出现在远程选项卡中。恢复会在本地加载对话历史记录；更改不会同步回 claude.ai。

自定义您的工作流

一旦您启动并运行，您可以重新定位 Claude 面板、运行多个会话或切换到终端模式。

选择 Claude 的位置

您可以拖动 Claude 面板在 VS Code 中重新定位它。抓住面板的选项卡或标题栏并将其拖到：

- **次级边栏**：窗口的右侧。在您编码时保持 Claude 可见。
- **主边栏**：左侧边栏，带有资源管理器、搜索等图标。
- **编辑器区域**：将 Claude 作为选项卡打开，与您的文件并排。适用于辅助任务。

Tip:

为您的主 Claude 会话使用边栏，并为辅助任务打开其他选项卡。Claude 会记住您首选的位置。活动栏会话列表图标与 Claude 面板分开：会话列表在活动栏中始终可见，而 Claude 面板图标仅在面板停靠到左侧边栏时出现在那里。

运行多个对话

从命令面板使用**在新选项卡中打开**或**在新窗口中打开**来启动其他对话。每个对话维护自己的历史记录和上下文，允许您并行处理不同的任务。

使用选项卡时，spark 图标上的小彩色点表示状态：蓝色表示权限请求待处理，橙色表示 Claude 在选项卡隐藏时完成。

切换到终端模式

默认情况下，扩展打开图形聊天面板。如果您更喜欢 CLI 风格的界面，打开[使用终端设置](#)并选中该框。

您也可以打开 VS Code 设置（Mac 上为 **Cmd+,** 或 Windows/Linux 上为 **Ctrl+,**），转到扩展 → Claude Code，然后选中**使用终端**。

管理 plugins

VS Code 扩展包括用于安装和管理 [plugins](#) 的图形界面。在提示框中输入 **/plugins** 以打开**管理 plugins** 界面。

安装 plugins

plugin 对话框显示两个选项卡：**Plugins** 和 **Marketplaces**。

在 Plugins 选项卡中：

- **已安装的 plugins** 显示在顶部，带有切换开关以启用或禁用它们
- **可用的 plugins** 来自您配置的 marketplaces，显示在下方
- 搜索以按名称或描述过滤 plugins
- 点击任何可用 plugin 上的**安装**

当您安装 plugin 时，选择安装范围：

- **为您安装**：在您的所有项目中可用（用户范围）
- **为此项目安装**：与项目协作者共享（项目范围）
- **本地安装**：仅适用于您，仅在此存储库中（本地范围）

管理 marketplaces

切换到 **Marketplaces** 选项卡以添加或删除 plugin 源：

- 输入 GitHub 存储库、URL 或本地路径以添加新的 marketplace
- 点击刷新图标以更新 marketplace 的 plugin 列表
- 点击垃圾桶图标以删除 marketplace

进行更改后，横幅会提示您重启 Claude Code 以应用更新。

Note:

VS Code 中的 plugin 管理在幕后使用相同的 CLI 命令。您在扩展中配置的 plugins 和 marketplaces 也可在 CLI 中使用，反之亦然。

有关 plugin 系统的更多信息，请参阅 [Plugins](#) 和 [Plugin marketplaces](#)。

使用 Chrome 自动化浏览器任务

将 Claude 连接到您的 Chrome 浏览器以测试网络应用、使用控制台日志进行调试，以及在不离开 VS Code 的情况下自动化浏览器工作流。这需要 [Chrome 中的 Claude 扩展](#) 版本 1.0.36 或更高版本。

在提示框中输入 `@browser`，然后输入您想让 Claude 做的事情：

```
@browser go to localhost:3000 and check the console for errors
```

您也可以打开附件菜单以选择特定的浏览器工具，如打开新选项卡或读取页面内容。

Claude 为浏览器任务打开新选项卡并共享您的浏览器登录状态，因此它可以访问您已登录的任何网站。

有关设置说明、完整的功能列表和故障排除，请参阅[使用 Claude Code 与 Chrome](#)。

VS Code 命令和快捷键

打开命令面板（Mac 上为 `Cmd+Shift+P` 或 Windows/Linux 上为 `Ctrl+Shift+P`）并输入” Claude Code”以查看 Claude Code 扩展的所有可用 VS Code 命令。

某些快捷键取决于哪个面板”获得焦点”（接收键盘输入）。当您的光标在代码文件中时，编辑器获得焦点。当您的光标在 Claude 的提示框中时，Claude 获得焦点。使用 `Cmd+Esc` / `Ctrl+Esc` 在它们之间切换。

Note:

这些是用于控制扩展的 VS Code 命令。并非所有内置 Claude Code 命令都在扩展中可用。有关详细信息，请参阅 [VS Code 扩展与 Claude Code CLI](#)。

命令	快捷键	描述
焦点输入	<code>Cmd+Esc</code> (Mac) / <code>Ctrl+Esc</code> (Windows/Linux)	在编辑器和 Claude 之间切换焦点
在边栏中打开	-	在左侧边栏中打开 Claude
在终端中打开	-	在终端模式下打开 Claude
在新选项卡中打开	<code>Cmd+Shift+Esc</code> (Mac) / <code>Ctrl+Shift+Esc</code> (Windows/Linux)	将新对话作为编辑器选项卡打开
在新窗口中打开	-	在单独的窗口中打开新对话
新对话	<code>Cmd+N</code> (Mac) / <code>Ctrl+N</code> (Windows/Linux)	开始新对话（需要 Claude 获得焦点）
插入 @-提及引用	<code>Option+K</code> (Mac) / <code>Alt+K</code> (Windows/Linux)	插入对当前文件和选择的引用（需要编辑器获得焦点）
显示日志	-	查看扩展调试日志
登出	-	登出您的 Anthropic 账户

配置设置

扩展有两种类型的设置：

- **扩展设置**在 VS Code 中：控制扩展在 VS Code 中的行为。使用 `Cmd+,` (Mac) 或 `Ctrl+,` (Windows/Linux) 打开，然后转到扩展 → Claude Code。您也可以输入 `/` 并选择**常规配置**以打开设置。
- **Claude Code 设置**在 `~/.claude/settings.json` 中：在扩展和 CLI 之间共享。用于允许的命令、环境变量、hooks 和 MCP servers。有关详细信息，请参阅[设置](#)。

Tip:

将 `"$schema": "https://json.schemastore.org/claude-code-settings.json"` 添加到您的 `settings.json` 以在 VS Code 中直接获得所有可用设置的自动完成和内联验证。

扩展设置

设置	默认值	描述
<code>selectedModel</code>	<code>default</code>	新对话的模型。使用 <code>/model</code> 按会话更改。
<code>useTerminal</code>	<code>false</code>	以终端模式而不是图形面板启动 Claude
<code>initialPermissionMode</code>	<code>default</code>	控制批准提示： <code>default</code> （每次询问）、 <code>plan</code> 、 <code>acceptEdits</code> 或 <code>bypassPermissions</code>
<code>preferredLocation</code>	<code>panel</code>	Claude 打开的位置： <code>sidebar</code> （右侧）或 <code>panel</code> （新选项卡）
<code>autosave</code>	<code>true</code>	在 Claude 读取或写入文件前自动保存文件
<code>useCtrlEnterToSend</code>	<code>false</code>	使用 <code>Ctrl/Cmd+Enter</code> 而不是 <code>Enter</code> 发送提示
<code>enableNewConversationShortcut</code>	<code>true</code>	启用 <code>Cmd/Ctrl+N</code> 以开始新对话
<code>hideOnboarding</code>	<code>false</code>	隐藏入门检查清单（毕业帽图标）
<code>respectGitIgnore</code>	<code>true</code>	从文件搜索中排除 <code>.gitignore</code> 模式
<code>environmentVariables</code>	<code>[]</code>	为 Claude 进程设置环境变量。对于共享配置，改用 Claude Code 设置。
<code>disableLoginPrompt</code>	<code>false</code>	跳过身份验证提示（用于第三方提供商设置）。

设置	默认值	描述
<code>allowDangerouslySkipPermissions</code>	<code>false</code>	绕过所有权限提示。 谨慎使用。
<code>claudeProcessWrapper</code>	-	用于启动 Claude 进程的可执行文件路径

VS Code 扩展与 Claude Code CLI

Claude Code 既可作为 VS Code 扩展（图形面板）也可作为 CLI（终端中的命令行界面）使用。某些功能仅在 CLI 中可用。如果您需要仅限 CLI 的功能，请在 VS Code 的集成终端中运行 `claude`。

功能	CLI	VS Code 扩展
命令和 skills	全部	子集（输入 <code>/</code> 以查看可用的）
MCP server 配置	是	部分（通过 CLI 添加 servers；使用聊天面板中的 <code>/mcp</code> 管理现有 servers）
Checkpoints	是	是
 bash 快捷方式	是	否
制表符完成	是	否

使用 checkpoints 进行 Rewind

VS Code 扩展支持 checkpoints，它们跟踪 Claude 的文件编辑并让您 rewind 到之前的状态。将鼠标悬停在任何消息上以显示 rewind 按钮，然后从三个选项中选择：

- **从此处分叉对话：**从此消息开始新的对话分支，同时保持所有代码更改完整
- **将代码 Rewind 到此处：**将文件更改恢复到对话中的此点，同时保持完整的对话历史记录
- **分叉对话并 Rewind 代码：**开始新的对话分支并将文件更改恢复到此点

有关 checkpoints 如何工作及其限制的完整详细信息，请参阅 [Checkpointing](#)。

在 VS Code 中运行 CLI

要在 VS Code 中使用 CLI，请打开集成终端（Windows/Linux 上为 `Ctrl+`` 或 Mac 上为 `Cmd+``）并运行 `claude`。CLI 会自动与您的 IDE 集成，以获得差异查看和诊断共享等功能。

如果使用外部终端，请在 Claude Code 中运行 `/ide` 以将其连接到 VS Code。

在扩展和 CLI 之间切换

扩展和 CLI 共享相同的对话历史记录。要在 CLI 中继续扩展对话，请在终端中运行 `claude --resume`。这会打开一个交互式选择器，您可以在其中搜索和选择您的对话。

在提示中包含终端输出

使用 `@terminal:name` 在您的提示中引用终端输出，其中 `name` 是终端的标题。这让 Claude 看到命令输出、错误消息或日志，而无需复制粘贴。

监控后台进程

当 Claude 运行长时间运行的命令时，扩展在状态栏中显示进度。但是，与 CLI 相比，后台任务的可见性有限。为了获得更好的可见性，让 Claude 输出命令，以便您可以在 VS Code 的集成终端中运行它。

使用 MCP 连接到外部工具

MCP（Model Context Protocol）servers 为 Claude 提供对外部工具、数据库和 API 的访问。

要添加 MCP server，请打开集成终端（`Ctrl+`` 或 `Cmd+``）并运行：

```
claude mcp add --transport http github https://api.githubcopilot.com/mcp/
```

配置后，要求 Claude 使用这些工具（例如，“Review PR #456”）。

要不在离开 VS Code 的情况下管理 MCP servers，请在聊天面板中输入 `/mcp`。MCP 管理对话框让您启用或禁用 servers、重新连接到 server 以及管理 OAuth 身份验证。有关可用 servers，请参阅 [MCP 文档](#)。

使用 git

Claude Code 与 git 集成以帮助直接在 VS Code 中进行版本控制工作流。要求 Claude 提交更改、创建拉取请求或跨分支工作。

创建提交和拉取请求

Claude 可以暂存更改、编写提交消息并根据您的工作创建拉取请求：

```
> commit my changes with a descriptive message
> create a pr for this feature
> summarize the changes I've made to the auth module
```

创建拉取请求时，Claude 根据实际代码更改生成描述，并可以添加有关测试或实现决策的上下文。

使用 git worktrees 进行并行任务

使用 `--worktree` (`-w`) 标志在隔离的 `worktree` 中启动 Claude，该 `worktree` 具有自己的文件和分支：

```
claude --worktree feature-auth
```

每个 `worktree` 维护独立的文件状态，同时共享 git 历史记录。这可以防止 Claude 实例在处理不同任务时相互干扰。有关更多详细信息，请参阅[使用 Git worktrees 运行并行 Claude Code 会话](#)。

使用第三方提供商

默认情况下，Claude Code 直接连接到 Anthropic 的 API。如果您的组织使用 Amazon Bedrock、Google Vertex AI 或 Microsoft Foundry 来访问 Claude，请配置扩展以改用您的提供商：

Step 1: 禁用登录提示

打开[禁用登录提示设置](#)并选中该框。

您也可以打开 VS Code 设置（Mac 上为 `Cmd+,` 或 Windows/Linux 上为 `Ctrl+,`），搜索“Claude Code login”，然后选中[禁用登录提示](#)。

Step 2: 配置您的提供商

按照您的提供商的设置指南：

- [Amazon Bedrock 上的 Claude Code](#)
- [Google Vertex AI 上的 Claude Code](#)
- [Microsoft Foundry 上的 Claude Code](#)

这些指南涵盖在 `~/.claude/settings.json` 中配置您的提供商，这确保您的设置在 VS Code 扩展和 CLI 之间共享。

安全和隐私

您的代码保持私密。Claude Code 处理您的代码以提供协助，但不使用它来训练模型。有关数据处理的详细信息以及如何选择退出日志记录，请参阅[数据和隐私](#)。

启用自动编辑权限后，Claude Code 可以修改 VS Code 配置文件（如 `settings.json` 或 `tasks.json`），VS Code 可能会自动执行。要在处理不受信任的代码时降低风险：

- 为不受信任的工作区启用 [VS Code 受限模式](#)
- 使用手动批准模式而不是自动接受编辑
- 在接受更改前仔细审查它们

修复常见问题

扩展无法安装

- 确保您拥有兼容的 VS Code 版本（1.98.0 或更高版本）
- 检查 VS Code 是否有权安装扩展
- 尝试从 [VS Code Marketplace](#) 直接安装

Spark 图标不可见

Spark 图标在**编辑器工具栏**（编辑器右上角）中出现，当您打开文件时。如果您看不到它：

1. **打开文件：**该图标需要打开文件。仅打开文件夹是不够的。
2. **检查 VS Code 版本：**需要 1.98.0 或更高版本（帮助 → 关于）
3. **重启 VS Code：**从命令面板运行”Developer: Reload Window”
4. **禁用冲突的扩展：**临时禁用其他 AI 扩展（Cline、Continue 等）
5. **检查工作区信任：**扩展在受限模式下不工作

或者，点击**状态栏**（右下角）中的“**Claude Code**”。即使没有打开文件，这也有效。您也可以使用**命令面板**（`Cmd+Shift+P` / `Ctrl+Shift+P`）并输入“Claude Code”。

Claude Code 从不响应

如果 Claude Code 没有响应您的提示：

1. **检查您的互联网连接：**确保您有稳定的互联网连接
2. **开始新对话：**尝试开始新的对话以查看问题是否仍然存在

3. 尝试 CLI：从终端运行 `claude` 以查看是否获得更详细的错误消息

如果问题仍然存在，请在 [GitHub 上提交问题](#)，并提供有关错误的详细信息。

卸载扩展

要卸载 Claude Code 扩展：

1. 打开扩展视图（Mac 上为 `Cmd+Shift+X` 或 Windows/Linux 上为 `Ctrl+Shift+X`）
2. 搜索“Claude Code”
3. 点击**卸载**

要也删除扩展数据并重置所有设置：

```
rm -rf ~/.vscode/globalStorage/anthropic.claude-code
```

有关其他帮助，请参阅[故障排除指南](#)。

后续步骤

现在您已在 VS Code 中设置了 Claude Code：

- [探索常见工作流](#)以充分利用 Claude Code
- [设置 MCP servers](#)以使用外部工具扩展 Claude 的功能。使用 CLI 添加 servers，然后使用聊天面板中的 `/mcp` 管理它们。
- [配置 Claude Code 设置](#)以自定义允许的命令、hooks 等。这些设置在扩展和 CLI 之间共享。

JetBrains IDEs

在 JetBrains IDE（包括 IntelliJ、PyCharm、WebStorm 等）中使用 Claude Code

Claude Code 通过专用插件与 JetBrains IDE 集成，提供交互式差异查看、选择上下文共享等功能。

支持的 IDE

Claude Code 插件适用于大多数 JetBrains IDE，包括：

- IntelliJ IDEA
- PyCharm
- Android Studio
- WebStorm
- PhpStorm
- GoLand

功能

- **快速启动：**使用 `Cmd+Esc`（Mac）或 `Ctrl+Esc`（Windows/Linux）直接从编辑器打开 Claude Code，或点击 UI 中的 Claude Code 按钮
- **差异查看：**代码更改可以直接在 IDE 差异查看器中显示，而不是在终端中显示
- **选择上下文：**IDE 中的当前选择/标签页会自动与 Claude Code 共享
- **文件引用快捷方式：**使用 `Cmd+Option+K`（Mac）或 `Alt+Ctrl+K`（Linux/Windows）插入文件引用（例如，@File#L1-99）
- **诊断共享：**IDE 中的诊断错误（lint、语法等）在您工作时会自动与 Claude 共享

安装

市场安装

从 JetBrains 市场查找并安装 [Claude Code 插件](#)，然后重启您的 IDE。

如果您还没有安装 Claude Code，请参阅[我们的快速入门指南](#)了解安装说明。

Note:

安装插件后，您可能需要完全重启 IDE 才能使其生效。

使用

从您的 IDE

从 IDE 的集成终端运行 `claude`，所有集成功能都将处于活跃状态。

从外部终端

在任何外部终端中使用 `/ide` 命令将 Claude Code 连接到您的 JetBrains IDE 并激活所有功能：

```
claude
```

```
/ide
```

如果您希望 Claude 能够访问与 IDE 相同的文件，请从与 IDE 项目根目录相同的目录启动 Claude Code。

配置

Claude Code 设置

通过 Claude Code 的设置配置 IDE 集成：

1. 运行 `claude`
2. 输入 `/config` 命令
3. 将差异工具设置为 `auto` 以进行自动 IDE 检测

插件设置

通过转到 **Settings → Tools → Claude Code [Beta]** 配置 Claude Code 插件：

常规设置

- **Claude 命令：**指定自定义命令来运行 Claude（例如，`claude`、`/usr/local/bin/claude` 或 `npx @anthropic/claude`）
- **抑制 Claude 命令未找到的通知：**跳过有关找不到 Claude 命令的通知

- **启用使用 Option+Enter 进行多行提示**（仅 macOS）：启用后，Option+Enter 在 Claude Code 提示中插入新行。如果遇到 Option 键被意外捕获的问题，请禁用（需要终端重启）
- **启用自动更新**：自动检查并安装插件更新（在重启时应用）

Tip:

对于 WSL 用户：将 `wsl -d Ubuntu -- bash -lic "claude"` 设置为您的 Claude 命令（将 `Ubuntu` 替换为您的 WSL 发行版名称）

ESC 键配置

如果 ESC 键在 JetBrains 终端中无法中断 Claude Code 操作：

1. 转到 **Settings → Tools → Terminal**
2. 执行以下任一操作：
 - 取消选中”使用 Escape 将焦点移动到编辑器”，或
 - 点击”配置终端快捷键”并删除”切换焦点到编辑器”快捷方式
3. 应用更改

这将允许 ESC 键正确中断 Claude Code 操作。

特殊配置**远程开发****Warning:**

使用 JetBrains 远程开发时，您必须通过 **Settings → Plugin (Host)** 在远程主机上安装插件。

插件必须安装在远程主机上，而不是在您的本地客户端计算机上。

WSL 配置**Warning:**

WSL 用户可能需要额外配置才能使 IDE 检测正常工作。有关详细设置说明，请参阅我们的 [WSL 故障排除指南](#)。

WSL 配置可能需要：

- 正确的终端配置
- 网络模式调整
- 防火墙设置更新

故障排除

插件不工作

- 确保您从项目根目录运行 Claude Code
- 检查 JetBrains 插件在 IDE 设置中是否已启用
- 完全重启 IDE（您可能需要多次执行此操作）
- 对于远程开发，确保插件已安装在远程主机上

IDE 未检测到

- 验证插件已安装并启用
- 完全重启 IDE
- 检查您是否从集成终端运行 Claude Code
- 对于 WSL 用户，请参阅 [WSL 故障排除指南](#)

命令未找到

如果点击 Claude 图标显示”命令未找到”：

1. 验证 Claude Code 已安装：`npm list -g @anthropic-ai/claude-code`
2. 在插件设置中配置 Claude 命令路径
3. 对于 WSL 用户，使用配置部分中提到的 WSL 命令格式

安全考虑

当 Claude Code 在启用自动编辑权限的 JetBrains IDE 中运行时，它可能能够修改可由您的 IDE 自动执行的 IDE 配置文件。这可能会增加在自动编辑模式下运行 Claude Code 的风险，并允许绕过 Claude Code 对 bash 执行的权限提示。

在 JetBrains IDE 中运行时，请考虑：

- 对编辑使用手动批准模式
- 特别小心确保 Claude 仅与受信任的提示一起使用
- 了解 Claude Code 有权修改哪些文件

如需更多帮助，请参阅我们的[故障排除指南](#)。

使用 Claude Code Desktop

充分利用 Claude Code Desktop：通过 Git 隔离的并行会话、可视化 diff 审查、应用预览、PR 监控、权限模式、连接器和企业配置。

Claude Desktop 应用中的 Code 选项卡让你可以通过图形界面而不是终端来使用 Claude Code。

Desktop 在标准 Claude Code 体验的基础上添加了这些功能：

- [可视化 diff 审查](#)，支持内联注释
- [实时应用预览](#)，支持开发服务器
- [GitHub PR 监控](#)，支持自动修复和自动合并
- [并行会话](#)，支持自动 Git worktree 隔离
- [计划任务](#)，按定期计划运行 Claude
- [连接器](#)，支持 GitHub、Slack、Linear 等
- 本地、[SSH](#) 和 [云](#)环境

Tip:

初次使用 Desktop？从[快速开始](#)开始安装应用并进行第一次编辑。

本页涵盖[使用代码](#)、[管理会话](#)、[扩展 Claude Code](#)、[计划任务](#)和[配置](#)。它还包括 [CLI 比较](#) 和 [故障排除](#)。

启动会话

在发送第一条消息之前，在提示区域配置四件事：

- **环境**：选择 Claude 运行的位置。选择[本地](#)用于你的机器，[远程](#)用于 Anthropic 托管的云会话，或[SSH 连接](#)用于你管理的远程机器。请参阅[环境配置](#)。
- **项目文件夹**：选择 Claude 工作的文件夹或存储库。对于远程会话，你可以添加[多个存储库](#)。
- **模型**：从发送按钮旁的下拉菜单中选择一个[模型](#)。会话启动后，模型将被锁定。
- **权限模式**：从[模式选择器](#)中选择 Claude 拥有多少自主权。你可以在会话期间更改此设置。

输入你的任务并按 **Enter** 启动。每个会话独立跟踪其自己的上下文和更改。

使用代码

为 Claude 提供正确的上下文，控制它自己做多少工作，并审查它更改的内容。

使用提示框

输入你想让 Claude 做的事情，然后按 **Enter** 发送。Claude 读取你的项目文件，进行更改，并根据你的[权限模式](#)运行命令。你可以随时中断 Claude：点击停止按钮或输入你的更正并按 **Enter**。Claude 停止正在做的事情并根据你的输入进行调整。

提示框旁的 + 按钮让你可以访问文件附件、[skills](#)、[连接器](#)和[插件](#)。

向提示添加文件和上下文

提示框支持两种方式来引入外部上下文：

- **@mention 文件**：输入 @ 后跟文件名，将文件添加到对话上下文。Claude 然后可以读取和引用该文件。
- **附加文件**：使用附件按钮将图像、PDF 和其他文件附加到你的提示，或直接将文件拖放到提示中。这对于共享错误的屏幕截图、设计模型或参考文档很有用。

选择权限模式

权限模式控制 Claude 在会话期间拥有多少自主权：它是否在编辑文件、运行命令或两者之前询问。你可以随时使用发送按钮旁的模式选择器切换模式。从”询问权限”开始，以准确了解 Claude 的操作，然后随着你变得更舒适，转向”自动接受编辑”或”Plan Mode”。

模式	设置键	行为
询问权限	<code>default</code>	Claude 在编辑文件或运行命令之前询问。你会看到一个 diff，可以接受或拒绝每个更改。推荐给新用户。
自动接受编辑	<code>acceptEdits</code>	Claude 自动接受文件编辑，但在运行终端命令之前仍然询问。当你信任文件更改并想要更快的迭代时，使用此选项。
Plan Mode	<code>plan</code>	Claude 分析你的代码并创建计划，而不修改文件或运行命令。适合复杂任务，你想先审查方法。

模式	设置键	行为
绕过权限	<code>bypassPermissions</code>	Claude 运行时没有任何权限提示，等同于 CLI 中的 <code>-dangerously-skip-permissions</code> 。在设置 → Claude Code 中的“允许绕过权限模式”下启用。仅在沙箱容器或虚拟机中使用。企业管理员可以禁用此选项。

`dontAsk` 权限模式仅在 [CLI](#) 中可用。

Tip:

在 Plan Mode 中启动复杂任务，以便 Claude 在进行更改之前制定方法。一旦你批准计划，切换到“自动接受编辑”或“询问权限”来执行它。有关此工作流的更多信息，请参阅[先探索，然后计划，然后编码](#)。

远程会话支持”自动接受编辑”和”Plan Mode”。“询问权限”不可用，因为远程会话默认自动接受文件编辑，“绕过权限”不可用，因为远程环境已经是沙箱化的。

企业管理员可以限制哪些权限模式可用。有关详细信息，请参阅[企业配置](#)。

预览你的应用

Claude 可以启动开发服务器并打开嵌入式浏览器来验证其更改。这适用于前端 Web 应用以及后端服务器：Claude 可以测试 API 端点、查看服务器日志并迭代它发现的问题。在大多数情况下，Claude 在编辑项目文件后自动启动服务器。你也可以随时要求 Claude 预览。默认情况下，Claude [自动验证](#) 每次编辑后的更改。

从预览面板，你可以：

- 在嵌入式浏览器中直接与你运行的应用交互
- 观看 Claude 自动验证其自己的更改：它拍摄屏幕截图、检查 DOM、点击元素、填充表单并修复它发现的问题
- 从会话工具栏中的 **Preview** 下拉菜单启动或停止服务器
- 通过在下拉菜单中选择 **Persist sessions** 来在服务器重启时保持 cookie 和本地存储，这样你就不必在开发期间重新登录
- 编辑服务器配置或一次停止所有服务器

Claude 根据你的项目创建初始服务器配置。如果你的应用使用自定义开发命令，编辑 `.claude/launch.json` 以匹配你的设置。有关完整参考，请参阅[配置预览服务器](#)。

要清除保存的会话数据，在设置 → Claude Code 中切换 **Persist preview sessions** 关闭。要完全禁用预览，在设置 → Claude Code 中切换 **Preview** 关闭。

使用 diff 视图审查更改

Claude 对你的代码进行更改后，diff 视图让你在创建拉取请求之前逐个文件审查修改。

当 Claude 更改文件时，会出现一个 diff 统计指示器，显示添加和删除的行数，例如 `+12 -1`。点击此指示器打开 diff 查看器，它在左侧显示文件列表，在右侧显示每个文件的更改。

要对特定行进行注释，点击 diff 中的任何行以打开注释框。输入你的反馈并按 **Enter** 添加注释。在多行添加注释后，一次提交所有注释：

- **macOS**：按 **Cmd+Enter**
- **Windows**：按 **Ctrl+Enter**

Claude 读取你的注释并进行请求的更改，这些更改显示为你可以审查的新 diff。

审查你的代码

在 diff 视图中，点击右上角工具栏中的 **Review code** 来要求 Claude 在你提交之前评估更改。Claude 检查当前 diff 并直接在 diff 视图中留下注释。你可以回复任何注释或要求 Claude 修改。

审查侧重于高信号问题：编译错误、明确的逻辑错误、安全漏洞和明显的错误。它不标记样式、格式、预先存在的问题或 linter 会捕获的任何内容。

监控拉取请求状态

打开拉取请求后，CI 状态栏出现在会话中。Claude Code 使用 GitHub CLI 轮询检查结果并显示失败。

- **自动修复**：启用后，Claude 通过读取失败输出并迭代来自动尝试修复失败的 CI 检查。
- **自动合并**：启用后，Claude 在所有检查通过后合并 PR。合并方法是 squash。自动合并必须要在你的 GitHub 存储库设置中[启用](#)才能工作。

使用 CI 状态栏中的 **Auto-fix** 和 **Auto-merge** 切换来启用任一选项。Claude Code 还在 CI 完成时发送桌面通知。

Note:

PR 监控需要在你的机器上安装并验证 [GitHub CLI \(gh \)](#)。如果未安装 `gh`，Desktop 会在你第一次尝试创建 PR 时提示你安装它。

管理会话

每个会话是一个独立的对话，拥有自己的上下文和更改。你可以并行运行多个会话或将工作发送到云。

使用会话并行工作

点击侧边栏中的 **+ New session** 来并行处理多个任务。对于 Git 存储库，每个会话使用 [Git worktrees](#) 获得自己的隔离项目副本，因此一个会话中的更改不会影响其他会话，直到你提交它们。

Worktrees 默认存储在 `<project-root>/ .claude/worktrees/` 中。你可以在设置 → Claude Code 中的”Worktree location”下将其更改为自定义目录。你也可以设置一个分支前缀，该前缀会添加到每个 worktree 分支名称前面，这对于保持 Claude 创建的分支有组织很有用。要在完成后删除 worktree，请将鼠标悬停在侧边栏中的会话上并点击存档图标。

Note:

会话隔离需要 Git。大多数 Mac 默认包含 Git。在终端中运行 `git --version` 来检查。在 Windows 上，Git 是 Code 选项卡工作所必需的：[下载 Git for Windows](#)，安装它，然后重启应用。如果你遇到 Git 错误，请尝试 Cowork 会话来帮助排除你的设置。

使用侧边栏顶部的过滤器图标按状态（活跃、已存档）和环境（本地、云）过滤会话。要重命名会话或检查上下文使用情况，点击活跃会话顶部工具栏中的会话标题。当上下文填满时，Claude 自动总结对话并继续工作。你也可以输入 `/compact` 来更早触发总结并释放上下文空间。有关压缩工作原理的详细信息，请参阅[上下文窗口](#)。

远程运行长时间运行的任务

对于大型重构、测试套件、迁移或其他长时间运行的任务，在启动会话时选择 **Remote** 而不是 **Local**。远程会话在 Anthropic 的云基础设施上运行，即使你关闭应用或关闭计算机，也会继续运行。随时检查进度或引导 Claude 朝不同方向发展。你也可以从 [claude.ai/code](#) 或 Claude iOS 应用监控远程会话。

远程会话也支持多个存储库。选择云环境后，点击 repo pill 旁的 + 按钮向会话添加其他存储库。每个 repo 都有自己的分支选择器。这对于跨越多个代码库的任务很有用，例如更新共享库及其使用者。

有关远程会话如何工作的更多信息，请参阅[网络上的 Claude Code](#)。

在另一个表面继续

Continue in 菜单，可从会话工具栏右下角的 VS Code 图标访问，让你将会话移动到另一个表面：

- **网络上的 Claude Code**：将你的本地会话发送到远程继续运行。Desktop 推送你的分支，生成对话摘要，并创建具有完整上下文的新远程会话。然后你可以选择存档本地会话或保留它。这需要干净的工作树，对于 SSH 会话不可用。
- **你的 IDE**：在当前工作目录的支持的 IDE 中打开你的项目。

扩展 Claude Code

连接外部服务、添加可重用工作流、自定义 Claude 的行为并配置预览服务器。

连接外部工具

对于本地和 [SSH](#) 会话，点击提示框旁的 + 按钮并选择 **Connectors** 来添加集成，如 Google Calendar、Slack、GitHub、Linear、Notion 等。你可以在会话之前或期间添加连接器。远程会话不支持连接器。

要管理或断开连接器，请在桌面应用中转到设置 → Connectors，或从提示框中的 Connectors 菜单中选择 **Manage connectors**。

连接后，Claude 可以读取你的日历、发送消息、创建问题并直接与你的工具交互。你可以询问 Claude 在你的会话中配置了哪些连接器。

连接器是 [MCP servers](#)，具有图形设置流程。使用它们快速与支持的服务集成。对于 Connectors 中未列出的集成，通过 [设置文件](#) 手动添加 MCP servers。你也可以 [创建自定义连接器](#)。

使用 skills

[Skills](#) 扩展 Claude 可以做的事情。Claude 在相关时自动加载它们，或者你可以直接调用一个：在提示框中输入 `/` 或点击 + 按钮并选择 **Slash commands** 来浏览可用的内容。这包括 [内置命令](#)、你的 [自定义 skills](#)、来自你的代码库的项目 skills 以及来自任何 [已安装插件](#) 的 skills。选择一个，它会在输入字段中突出显示。在它之后输入你的任务并照常发送。

安装插件

[Plugins](#) 是可重用的包，为 Claude Code 添加 skills、agents、hooks、MCP servers 和 LSP 配置。你可以从桌面应用安装插件，而无需使用终端。

对于本地和 [SSH](#) 会话，点击提示框旁的 + 按钮并选择 **Plugins** 来查看你已安装的插件及其命令。要添加插件，从子菜单中选择 **Add plugin** 来打开插件浏览器，它显示来自你配置的 [市场](#) 的可用插件，包括官方 Anthropic 市场。选择 **Manage plugins** 来启用、禁用或卸载插件。

插件可以限定到你的用户账户、特定项目或仅本地。远程会话不支持插件。有关完整的插件参考，包括创建你自己的插件，请参阅[插件](#)。

配置预览服务器

Claude 自动检测你的开发服务器设置并将配置存储在启动会话时选择的文件夹根目录的 `.claude/launch.json` 中。Preview 使用此文件夹作为其工作目录，因此如果你选择了父文件夹，具有自己开发服务器的子文件夹将不会自动检测。要使用子文件夹的服务器，要么直接在该文件夹中启动会话，要么手动添加配置。

要自定义服务器的启动方式，例如使用 `yarn dev` 而不是 `npm run dev` 或更改端口，手动编辑文件或点击 Preview 下拉菜单中的 **Edit configuration** 在你的代码编辑器中打开它。该文件支持带注释的 JSON。

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "my-app",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "dev"],
      "port": 3000
    }
  ]
}
```

你可以定义多个配置来从同一项目运行不同的服务器，例如前端和 API。请参阅下面的[示例](#)。

自动验证更改

启用 `autoVerify` 时，Claude 在编辑文件后自动验证代码更改。它拍摄屏幕截图、检查错误并在完成响应之前确认更改有效。

自动验证默认打开。通过在 `.claude/launch.json` 中添加 `"autoVerify": false` 来按项目禁用它，或从 **Preview** 下拉菜单切换它。

```
{
  "version": "0.0.1",
  "autoVerify": false,
  "configurations": [...]
}
```

禁用时，预览工具仍然可用，你可以随时要求 Claude 验证。自动验证使其在每次编辑后自动进行。

配置字段

`configurations` 数组中的每个条目接受以下字段：

字段	类型	描述
<code>name</code>	string	此服务器的唯一标识符
<code>runtimeExecutable</code>	string	要运行的命令，例如 <code>npm</code> 、 <code>yarn</code> 或 <code>node</code>
<code>runtimeArgs</code>	string[]	传递给 <code>runtimeExecutable</code> 的参数，例如 <code>["run", "dev"]</code>
<code>port</code>	number	你的服务器监听的端口。默认为 3000
<code>cwd</code>	string	相对于你的项目根目录的工作目录。默认为项目根目录。使用 <code>\${workspaceFolder}</code> 显式引用项目根目录
<code>env</code>	object	其他环境变量作为键值对，例如 <code>{ "NODE_ENV": "development" }</code> 。不要在这里放置秘密，因为此文件被提交到你的 repo。在你的 shell 配置文件中设置的秘密会自动继承。
<code>autoPort</code>	boolean	如何处理端口冲突。见下文

字段	类型	描述
<code>program</code>	string	用 <code>node</code> 运行的脚本。请参阅 何时使用 <code>program</code> vs <code>runtimeExecutable</code>
<code>args</code>	string[]	传递给 <code>program</code> 的参数。仅在设置 <code>program</code> 时使用

何时使用 `program` vs `runtimeExecutable`

使用 `runtimeExecutable` 和 `runtimeArgs` 通过包管理器启动开发服务器。例如，`"runtimeExecutable": "npm"` 和 `"runtimeArgs": ["run", "dev"]` 运行 `npm run dev`。

当你有一个想直接用 `node` 运行的独立脚本时，使用 `program`。例如，`"program": "server.js"` 运行 `node server.js`。使用 `args` 传递其他标志。

端口冲突

`autoPort` 字段控制当你的首选端口已在使用时会发生什么：

- `true`：Claude 自动查找并使用空闲端口。适合大多数开发服务器。
- `false`：Claude 失败并出现错误。当你的服务器必须使用特定端口时使用此选项，例如 OAuth 回调或 CORS 允许列表。
- **未设置（默认）**：Claude 询问服务器是否需要该确切端口，然后保存你的答案。

当 Claude 选择不同的端口时，它通过 `PORT` 环境变量将分配的端口传递给你的服务器。

示例

这些配置显示了不同项目类型的常见设置：

Next.js

此配置使用 Yarn 在端口 3000 上运行 Next.js 应用：

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "web",
      "runtimeExecutable": "yarn",
      "runtimeArgs": ["dev"],
      "port": 3000
    }
  ]
}
```

多个服务器

对于具有前端和 API 服务器的 monorepo，定义多个配置。前端使用 `autoPort: true`，因此如果 3000 被占用，它会选择空闲端口，而 API 服务器需要端口 8080：

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "frontend",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "dev"],
      "cwd": "apps/web",
      "port": 3000,
      "autoPort": true
    },
    {
      "name": "api",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "start"],
      "cwd": "server",
      "port": 8080,
      "env": { "NODE_ENV": "development" },
      "autoPort": false
    }
  ]
}
```

Node.js 脚本

要直接运行 Node.js 脚本而不是使用包管理器命令，使用 `program` 字段：

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "server",
      "program": "server.js",
      "args": ["--verbose"],
      "port": 4000
    }
  ]
}
```

计划定期任务

计划任务在你选择的时间和频率自动启动新的本地会话。使用它们进行定期工作，如每日代码审查、依赖项更新检查或从你的日历和收件箱提取的早晨简报。

任务在你的机器上运行，因此桌面应用必须打开且你的计算机处于唤醒状态才能触发。有关错过的运行和追赶行为的详细信息，请参阅[计划任务如何运行](#)。

Note:

默认情况下，计划任务针对你的工作目录的任何状态运行，包括未提交的更改。在提示输入中启用 `worktree` 切换，为每次运行提供自己的隔离 Git `worktree`，与[并行会话](#)相同。

要创建计划任务，点击侧边栏中的 **Schedule**，然后点击 **+ New task**。配置这些字段：

字段	描述
Name	任务的标识符。转换为小写 kebab-case 并用作磁盘上的文件夹名称。必须在你的任务中唯一。
Description	任务列表中显示的简短摘要。
Prompt	任务运行时发送给 Claude 的指令。以你在提示框中编写任何消息的相同方式编写此内容。提示输入还包括模型、权限模式、工作文件夹和 <code>worktree</code> 的控制。
Frequency	任务运行的频率。请参阅下面的 频率选项 。

你也可以通过在任何会话中描述你想要的内容来创建任务。例如，“设置一个每天早上 9 点运行的每日代码审查。”

频率选项

- **Manual**: 无计划，仅在你点击 **Run now** 时运行。对于保存你按需触发的提示很有用
- **Hourly**: 每小时运行一次。每个任务从整点获得最多 10 分钟的固定偏移，以错开 API 流量
- **Daily**: 显示时间选择器，默认为上午 9:00 本地时间
- **Weekdays**: 与 Daily 相同，但跳过星期六和星期日
- **Weekly**: 显示时间选择器和日期选择器

对于选择器不提供的间隔（每 15 分钟、每月第一天等），在任何 Desktop 会话中询问 Claude 来设置计划。使用纯语言；例如，“安排一个任务每 6 小时运行所有测试。”

计划任务如何运行

计划任务在你的机器上本地运行。Desktop 在应用打开时每分钟检查一次计划，并在任务到期时启动新会话，独立于你打开的任何手动会话。每个任务在计划时间后获得最多 10 分钟的固定延迟，以错开 API 流量。延迟是确定性的：同一任务总是在相同的偏移处启动。

当任务触发时，你会收到桌面通知，新会话出现在侧边栏的 **Scheduled** 部分下。打开它以查看 Claude 做了什么、审查更改或回复权限提示。会话像任何其他会话一样工作：Claude 可以编辑文件、运行命令、创建提交和打开拉取请求。

任务仅在桌面应用运行且你的计算机处于唤醒状态时运行。如果你的计算机在计划时间睡眠，运行将被跳过。要防止空闲睡眠，在设置中的 **Desktop app** → **General** 下启用 **Keep computer awake**。关闭笔记本电脑盖仍会使其睡眠。

错过的运行

当应用启动或你的计算机唤醒时，Desktop 检查每个任务是否在过去七天内错过了任何运行。如果有，Desktop 为最近错过的时间启动恰好一次追赶运行，并丢弃任何更旧的。一个错过六天的每日任务在唤醒时运行一次。Desktop 在追赶运行启动时显示通知。

在编写提示时记住这一点。计划在上午 9 点的任务可能在晚上 11 点运行，如果你的计算机整天睡眠。如果时间很重要，向提示本身添加护栏，例如：“仅审查今天的提交。如果已经是下午 5 点后，跳过审查，只发布错过的摘要。”

计划任务的权限

每个任务都有自己的权限模式，你在创建或编辑任务时设置。来自 `~/.claude/settings.json` 的允许规则也适用于计划任务会话。如果任务在询问模式下运行并需要运行它没有权限的工具，运行将停滞，直到你批准它。会话保持在侧边栏中打开，以便你稍后可以回答。

要避免停滞，在创建任务后点击 **Run now**，观察权限提示，并为每个提示选择“总是允许”。该任务的未来运行自动批准相同的工具，无需提示。你可以从任务的详细页面审查和撤销这些批准。

管理计划任务

点击 **Schedule** 列表中的任务来打开其详细页面。从这里你可以：

- **Run now**：立即启动任务，无需等待下一个计划时间
- **Toggle repeats**：暂停或恢复计划运行，无需删除任务
- **Edit**：更改提示、频率、文件夹或其他设置
- **Review history**：查看每次过去的运行，包括因你的计算机睡眠而被跳过的运行
- **Review allowed permissions**：从 **Always allowed** 面板查看和撤销此任务的保存工具批准
- **Delete**：删除任务并存档它创建的所有会话

你也可以通过在任何 Desktop 会话中询问 Claude 来管理任务。例如，“暂停我的 dependency-audit 任务”、“删除 standup-prep 任务”或“显示我的计划任务。”

要在磁盘上编辑任务的提示，打开 `~/.claude/scheduled-tasks/<task-name>/SKILL.md`（或在设置 `CLAUDE_CONFIG_DIR` 时在其下）。该文件对 `name` 和 `description` 使用 YAML frontmatter，提示作为正文。更改在下一次运行时生效。计划、文件夹、模型和启用状态不在此文件中：通过编辑表单或询问 Claude 来更改它们。

环境配置

你在[启动会话](#)时选择的环境决定了 Claude 执行的位置以及你如何连接：

- **Local**：在你的机器上运行，直接访问你的文件
- **Remote**：在 Anthropic 的云基础设施上运行。即使你关闭应用，会话也会继续。
- **SSH**：在你通过 SSH 连接的远程机器上运行，例如你自己的服务器、云虚拟机或开发容器

本地会话

本地会话从你的 shell 继承环境变量。如果你需要其他变量，在你的 shell 配置文件中设置它们，例如 `~/.zshrc` 或 `~/.bashrc`，然后重启桌面应用。有关支持的变量的完整列表，请参阅[环境变量](#)。

扩展思考 默认启用，这改进了复杂推理任务的性能，但使用额外的令牌。要完全禁用思考，在你的 shell 配置文件中设置 `MAX_THINKING_TOKENS=0`。在 Opus 上，`MAX_THINKING_TOKENS` 被忽略，除了 `0`，因为自适应推理控制思考深度。

远程会话

远程会话即使在你关闭应用后也会在后台继续。使用计入你的[订阅计划限制](#)，没有单独的计算费用。

你可以创建具有不同网络访问级别和环境变量的自定义云环境。在启动远程会话时选择环境下拉菜单，然后选择 **Add environment**。有关配置网络访问和环境变量的详细信息，请参阅[云环境](#)。

SSH 会话

SSH 会话让你在远程机器上运行 Claude Code，同时使用桌面应用作为你的界面。这对于使用存在于云虚拟机、开发容器或具有特定硬件或依赖项的服务器上的代码库很有用。

要添加 SSH 连接，在启动会话之前点击环境下拉菜单并选择 **+ Add SSH connection**。对话框要求：

- **Name**：此连接的友好标签
- **SSH Host**：`user@hostname` 或在 `~/.ssh/config` 中定义的主机
- **SSH Port**：如果留空默认为 22，或使用你的 SSH 配置中的端口
- **Identity File**：你的私钥的路径，例如 `~/.ssh/id_rsa`。留空以使用默认密钥或你的 SSH 配置。

添加后，连接出现在环境下拉菜单中。选择它在该机器上启动会话。Claude 在远程机器上运行，可以访问其文件和工具。

Claude Code 必须安装在远程机器上。连接后，SSH 会话支持权限模式、连接器、插件和 MCP servers。

企业配置

Teams 或 Enterprise 计划上的组织可以通过管理员控制台控制、托管设置文件和设备管理策略来管理桌面应用行为。

管理员控制台控制

这些设置通过[管理员设置控制台](#)配置：

- **启用或禁用 Code 选项卡**：控制你的组织中的用户是否可以在桌面应用中访问 Claude Code
- **禁用绕过权限模式**：防止你的组织中的用户启用绕过权限模式
- **禁用网络上的 Claude Code**：为你的组织启用或禁用远程会话

托管设置

托管设置覆盖项目和用户设置，并在 Desktop 生成 CLI 会话时应用。你可以在你的组织的[托管设置](#)文件中设置这些键，或通过管理员控制台远程推送它们。

键	描述
<code>disableBypassPermissionsMode</code>	设置为 <code>"disable"</code> 以防止用户启用绕过权限模式。请参阅 托管设置 。

有关托管专用设置的完整列表，包括 `allowManagedPermissionRulesOnly` 和 `allowManagedHooksOnly`，请参阅[托管专用设置](#)。

通过管理员控制台上传的远程托管设置目前仅适用于 CLI 和 IDE 会话。对于 Desktop 特定的限制，使用上面的管理员控制台控制。

设备管理策略

IT 团队可以通过 macOS 上的 MDM 或 Windows 上的组策略管理桌面应用。可用的策略包括启用或禁用 Claude Code 功能、控制自动更新和设置自定义部署 URL。

- **macOS:** 通过 `com.anthropic.Claude` 偏好域使用 Jamf 或 Kandji 等工具配置
- **Windows:** 通过 `SOFTWARE\Policies\Claude` 处的注册表配置

身份验证和 SSO

企业组织可以要求所有用户使用 SSO。有关计划级别的详细信息，请参阅[身份验证](#)，有关 SAML 和 OIDC 配置，请参阅[设置 SSO](#)。

数据处理

Claude Code 在本地会话中本地处理你的代码，或在远程会话中在 Anthropic 的云基础设施上处理。对话和代码上下文被发送到 Anthropic 的 API 进行处理。有关数据保留、隐私和合规性的详细信息，请参阅[数据处理](#)。

部署

Desktop 可以通过企业部署工具分发：

- **macOS:** 通过 MDM（如 Jamf 或 Kandji）使用 `.dmg` 安装程序分发
- **Windows:** 通过 MSIX 包或 `.exe` 安装程序部署。有关企业部署选项（包括静默安装），请参阅[为 Windows 部署 Claude Desktop](#)

有关网络配置，如代理设置、防火墙允许列表和 LLM 网关，请参阅[网络配置](#)。

有关完整的企业配置参考，请参阅[企业配置指南](#)。

来自 CLI ？

如果你已经使用 Claude Code CLI ， Desktop 运行相同的底层引擎，具有图形界面。你可以在同一机器上同时运行两者，甚至在同一项目上。每个维护单独的会话历史，但它们通过 CLAUDE.md 文件共享配置和项目内存。

要将 CLI 会话移动到 Desktop ，在终端中运行 `/desktop` 。 Claude 保存你的会话并在桌面应用中打开它，然后退出 CLI 。此命令仅在 macOS 和 Windows 上可用。

Tip:

何时使用 Desktop vs CLI ：当你想要可视化 diff 审查、文件附件或侧边栏中的会话管理时，使用 Desktop 。当你需要脚本、自动化、第三方提供商或更喜欢终端 workflows 时，使用 CLI 。

CLI 标志等效项

此表显示了常见 CLI 标志的桌面应用等效项。未列出的标志没有桌面等效项，因为它们是为脚本或自动化设计的。

CLI	Desktop 等效项
<code>--model sonnet</code>	发送按钮旁的模型下拉菜单，在启动会话之前
<code>--resume</code> , <code>--continue</code>	点击侧边栏中的会话
<code>--permission-mode</code>	发送按钮旁的模式选择器
<code>--dangerously-skip-permissions</code>	绕过权限模式。在设置 → Claude Code → “允许绕过权限模式” 中启用。企业管理员可以禁用此设置。
<code>--add-dir</code>	在远程会话中使用 + 按钮添加多个 repos
<code>--allowedTools</code> , <code>--disallowedTools</code>	在 Desktop 中不可用
<code>--verbose</code>	不可用。检查系统日志： macOS 上的 Console.app ， Windows 上的事件查看器 → Windows 日志 → 应用程序
<code>--print</code> , <code>--output-format</code>	不可用。Desktop 仅是交互式的。
<code>ANTHROPIC_MODEL</code> env var	发送按钮旁的模型下拉菜单
<code>MAX_THINKING_TOKENS</code> env var	在 shell 配置文件中设置；适用于本地会话。请参阅 环境配置 。

共享配置

Desktop 和 CLI 读取相同的配置文件，因此你的设置会转移：

- [CLAUDE.md](#) 文件在你的项目中被两者使用
- [MCP servers](#) 在 `~/.claude.json` 或 `.mcp.json` 中配置在两者中工作
- [Hooks](#) 和 [skills](#) 在设置中定义适用于两者
- [Settings](#) 在 `~/.claude.json` 和 `~/.claude/settings.json` 中是共享的。权限规则、允许的工具和 `settings.json` 中的其他设置适用于 Desktop 会话。
- **Models**：Sonnet、Opus 和 Haiku 在两者中都可用。在 Desktop 中，在启动会话之前从发送按钮旁的下拉菜单中选择模型。你无法在活跃会话期间更改模型。

Note:

MCP servers：桌面聊天应用 vs Claude Code：在 `claude_desktop_config.json` 中为 Claude Desktop 聊天应用配置 MCP servers 与 Claude Code 分开，不会出现在 Code 选项中。要在 Claude Code 中使用 MCP servers，在 `~/.claude.json` 或你的项目的 `.mcp.json` 文件中配置它们。有关详细信息，请参阅 [MCP 配置](#)。

功能比较

此表比较了 CLI 和 Desktop 之间的核心功能。有关 CLI 标志的完整列表，请参阅 [CLI 参考](#)。

功能	CLI	Desktop
权限模式	所有模式，包括 <code>dontAsk</code>	询问权限、自动接受编辑、Plan Mode 和通过设置的绕过权限
<code>--dangerously-skip-permissions</code>	CLI 标志	绕过权限模式。在设置 → Claude Code → “允许绕过权限模式” 中启用
第三方提供商	Bedrock、Vertex、Foundry	不可用。Desktop 直接连接到 Anthropic 的 API。
MCP servers	在设置文件中配置	本地和 SSH 会话的连接器 UI，或设置文件
Plugins	<code>/plugin</code> 命令	插件管理器 UI
@mention 文件	基于文本	带自动完成

功能	CLI	Desktop
文件附件	不可用	图像、PDF
会话隔离	<code>--worktree</code> 标志	自动 worktrees
多个会话	单独的终端	侧边栏选项卡
定期任务	cron 作业、CI 管道	计划任务
脚本和自动化	<code>--print</code> 、 Agent SDK	不可用

Desktop 中不可用的内容

以下功能仅在 CLI 或 VS Code 扩展中可用：

- **第三方提供商**：Desktop 直接连接到 Anthropic 的 API。改用 [CLI](#) 与 Bedrock、Vertex 或 Foundry。
- **Linux**：桌面应用仅在 macOS 和 Windows 上可用。
- **内联代码建议**：Desktop 不提供自动完成风格的建议。它通过对话提示和显式代码更改工作。
- **Agent teams**：多 agent 编排通过 [CLI](#) 和 [Agent SDK](#) 可用，不在 Desktop 中。

故障排除

检查你的版本

要查看你运行的桌面应用版本：

- **macOS**：点击菜单栏中的 **Claude**，然后点击 **About Claude**
- **Windows**：点击 **Help**，然后点击 **About**

点击版本号将其复制到你的剪贴板。

Code 选项卡中的 403 或身份验证错误

如果在使用 Code 选项卡时看到 **Error 403: Forbidden** 或其他身份验证失败：

1. 从应用菜单中注销并重新登录。这是最常见的修复。
2. 验证你有活跃的付费订阅：Pro、Max、Teams 或 Enterprise。
3. 如果 CLI 工作但 Desktop 不工作，完全退出桌面应用，而不仅仅是关闭窗口，然后重新打开并登录。
4. 检查你的互联网连接和代理设置。

启动时屏幕空白或卡住

如果应用打开但显示空白或无响应的屏幕：

1. 重启应用。
2. 检查待处理的更新。应用在启动时自动更新。
3. 在 Windows 上，在 **Windows 日志** → **应用程序** 下的事件查看器中检查崩溃日志。

“Failed to load session”

如果你看到 **Failed to load session**，选定的文件夹可能不再存在，Git 存储库可能需要未安装的 Git LFS，或文件权限可能阻止访问。尝试选择不同的文件夹或重启应用。

会话找不到已安装的工具

如果 Claude 找不到 **npm**、**node** 或其他 CLI 命令等工具，验证工具在你的常规终端中工作，检查你的 shell 配置文件是否正确设置 PATH，并重启桌面应用以重新加载环境变量。

Git 和 Git LFS 错误

在 Windows 上，Git 是启动本地会话的 Code 选项卡所必需的。如果你看到“Git is required”，安装 [Git for Windows](#) 并重启应用。

如果你看到“Git LFS is required by this repository but is not installed”，从 git-lfs.com 安装 Git LFS，运行 **git lfs install**，然后重启应用。

Windows 上的 MCP servers 不工作

如果 MCP server 切换不响应或服务器在 Windows 上连接失败，检查服务器在你的设置中是否正确配置，重启应用，验证服务器进程在任务管理器中运行，并查看服务器日志以获取连接错误。

应用无法退出

- **macOS**：按 Cmd+Q。如果应用不响应，使用 Cmd+Option+Esc 强制退出，选择 Claude，然后点击强制退出。
- **Windows**：使用 Ctrl+Shift+Esc 的任务管理器来结束 Claude 进程。

Windows 特定问题

- **安装后 PATH 未更新**：打开新的终端窗口。PATH 更新仅适用于新的终端会话。
- **并发安装错误**：如果你看到关于另一个安装正在进行的错误，但实际上没有，尝试以管理员身份运行安装程序。
- **ARM64**：Windows ARM64 设备完全支持。

Intel Mac 上的 Cowork 选项卡不可用

Cowork 选项卡在 macOS 上需要 Apple Silicon (M1 或更高版本)。在 Windows 上，Cowork 在所有支持的硬件上可用。Chat 和 Code 选项卡在 Intel Mac 上正常工作。

在 CLI 中打开时” Branch doesn’ t exist yet”

远程会话可以创建在你的本地机器上不存在的分支。点击会话工具栏中的分支名称来复制它，然后在本地获取它：

```
git fetch origin <branch-name>
git checkout <branch-name>
```

仍然卡住？

- 在 [GitHub Issues](#) 上搜索或提交错误
- 访问 [Claude 支持中心](#)

提交错误时，包括你的桌面应用版本、你的操作系统、确切的错误消息和相关日志。在 macOS 上，检查 Console.app。在 Windows 上，检查事件查看器 → Windows 日志 → 应用程序。

在 Chrome 中使用 Claude Code（测试版）

将 Claude Code 连接到 Chrome 浏览器，以测试网络应用、使用控制台日志进行测试、自动填充表单以及从网页中提取数据。

Claude Code 与 Claude in Chrome 浏览器扩展程序集成，为您提供从 CLI 或 [VS Code 扩展程序](#) 进行浏览器自动化的功能。构建您的代码，然后在浏览器中测试和调试，无需切换上下文。

Claude 为浏览器任务打开新标签页，并共享您浏览器的登录状态，因此它可以访问您已登录的任何网站。浏览器操作在实时可见的 Chrome 窗口中运行。当 Claude 遇到登录页面或 CAPTCHA 时，它会暂停并要求您手动处理。

Note:

Chrome 集成处于测试版阶段，目前仅适用于 Google Chrome。尚不支持 Brave、Arc 或其他基于 Chromium 的浏览器。也不支持 WSL（Windows 子系统 for Linux）。

功能

连接 Chrome 后，您可以在单个工作流中链接浏览器操作和编码任务：

- **实时调试：**直接读取控制台错误和 DOM 状态，然后修复导致这些错误的代码
- **设计验证：**从 Figma 模型构建 UI，然后在浏览器中打开它以验证它是否匹配
- **网络应用测试：**测试表单验证、检查视觉回归或验证用户流程
- **已认证的网络应用：**与 Google Docs、Gmail、Notion 或您已登录的任何应用交互，无需 API 连接器
- **数据提取：**从网页中提取结构化信息并将其保存到本地
- **任务自动化：**自动化重复的浏览器任务，如数据输入、表单填充或多站点工作流
- **会话录制：**将浏览器交互录制为 GIF，以记录或分享发生的情况

前置条件

在使用 Claude Code 与 Chrome 之前，您需要：

- [Google Chrome](#) 浏览器
- [Claude in Chrome 扩展程序](#) 版本 1.0.36 或更高版本
- [Claude Code](#) 版本 2.0.73 或更高版本

- 直接 Anthropic 计划（Pro、Max、Team 或 Enterprise）

Note:

Chrome 集成不可通过 Amazon Bedrock、Google Cloud Vertex AI 或 Microsoft Foundry 等第三方提供商获得。如果您仅通过第三方提供商访问 Claude，则需要单独的 claude.ai 账户来使用此功能。

在 CLI 中开始

Step 1: 使用 Chrome 启动 Claude Code

使用 `--chrome` 标志启动 Claude Code：

```
claude --chrome
```

你也可以通过在现有会话中运行 `/chrome` 来从现有会话中启用 Chrome。

Step 2: 要求 Claude 使用浏览器

此示例导航到页面、与其交互并报告其发现，全部来自您的终端或编辑器：

```
Go to code.claude.com/docs, click on the search box,  
type "hooks", and tell me what results appear
```

随时运行 `/chrome` 以检查连接状态、管理权限或重新连接扩展程序。

对于 VS Code，请参阅 [VS Code 中的浏览器自动化](#)。

默认启用 Chrome

为了避免每个会话都传递 `--chrome`，运行 `/chrome` 并选择“默认启用”。

在 [VS Code 扩展程序](#) 中，只要安装了 Chrome 扩展程序，Chrome 就可用。无需额外标志。

Note:

在 CLI 中默认启用 Chrome 会增加上下文使用，因为浏览器工具始终被加载。如果您注意到上下文消耗增加，请禁用此设置，仅在需要时使用 `--chrome`。

管理网站权限

网站级权限从 Chrome 扩展程序继承。在 Chrome 扩展程序设置中管理权限，以控制 Claude 可以浏览、点击和输入的网站。

示例工作流

这些示例展示了将浏览器操作与编码任务结合的常见方式。运行 `/mcp` 并选择 `claude-in-chrome` 以查看可用浏览器工具的完整列表。

测试本地网络应用

在开发网络应用时，要求 Claude 验证您的更改是否正常工作：

```
I just updated the login form validation. Can you open localhost:3000,
try submitting the form with invalid data, and check if the error
messages appear correctly?
```

Claude 导航到您的本地服务器、与表单交互并报告其观察到的内容。

使用控制台日志进行调试

Claude 可以读取控制台输出以帮助诊断问题。告诉 Claude 要查找的模式，而不是要求所有控制台输出，因为日志可能很冗长：

```
Open the dashboard page and check the console for any errors when
the page loads.
```

Claude 读取控制台消息，可以过滤特定模式或错误类型。

自动填充表单

加快重复数据输入任务的速度：

```
I have a spreadsheet of customer contacts in contacts.csv. For each row,
go to the CRM at crm.example.com, click "Add Contact", and fill in the
name, email, and phone fields.
```

Claude 读取您的本地文件、导航网络界面并为每条记录输入数据。

在 Google Docs 中起草内容

使用 Claude 直接在您的文档中写入，无需 API 设置：

```
Draft a project update based on the recent commits and add it to my  
Google Doc at docs.google.com/document/d/abc123
```

Claude 打开文档、点击编辑器并输入内容。这适用于您已登录的任何网络应用：Gmail、Notion、Sheets 等。

从网页中提取数据

从网站中提取结构化信息：

```
Go to the product listings page and extract the name, price, and  
availability for each item. Save the results as a CSV file.
```

Claude 导航到页面、读取内容并将数据编译成结构化格式。

运行多站点工作流

协调多个网站之间的任务：

```
Check my calendar for meetings tomorrow, then for each meeting with  
an external attendee, look up their company website and add a note  
about what they do.
```

Claude 跨标签页工作以收集信息并完成工作流。

录制演示 GIF

创建浏览器交互的可共享录制：

```
Record a GIF showing how to complete the checkout flow, from adding  
an item to the cart through to the confirmation page.
```

Claude 录制交互序列并将其保存为 GIF 文件。

故障排除

未检测到扩展程序

如果 Claude Code 显示”未检测到 Chrome 扩展程序”：

1. 验证 Chrome 扩展程序已安装并在 `chrome://extensions` 中启用

2. 通过运行 `claude --version` 验证 Claude Code 是最新的
3. 检查 Chrome 是否正在运行
4. 运行 `/chrome` 并选择”重新连接扩展程序”以重新建立连接
5. 如果问题仍然存在, 请重新启动 Claude Code 和 Chrome

第一次启用 Chrome 集成时, Claude Code 会安装本机消息传递主机配置文件。Chrome 在启动时读取此文件, 因此如果扩展程序在您的第一次尝试中未被检测到, 请重新启动 Chrome 以获取新配置。

如果连接仍然失败, 请验证主机配置文件是否存在于:

- **macOS** : `~/Library/Application Support/Google/Chrome/NativeMessagingHosts/com.anthropic.claude_code_browser_extension.json`
- **Linux** : `~/.config/google-chrome/NativeMessagingHosts/com.anthropic.claude_code_browser_extension.json`
- **Windows**: 检查 Windows 注册表中的 `HKCU\Software\Google\Chrome\NativeMessagingHosts\`

浏览器无响应

如果 Claude 的浏览器命令停止工作:

1. 检查是否有模态对话框 (alert、confirm、prompt) 阻止页面。JavaScript 对话框阻止浏览器事件并防止 Claude 接收命令。手动关闭对话框, 然后告诉 Claude 继续。
2. 要求 Claude 创建新标签页并重试
3. 通过在 `chrome://extensions` 中禁用并重新启用来重新启动 Chrome 扩展程序

长会话期间连接断开

Chrome 扩展程序的 service worker 在扩展会话期间可能会进入空闲状态, 这会破坏连接。如果浏览器工具在一段时间不活动后停止工作, 请运行 `/chrome` 并选择”重新连接扩展程序”。

Windows 特定问题

在 Windows 上, 您可能会遇到:

- **命名管道冲突 (EADDRINUSE)**: 如果另一个进程正在使用相同的命名管道, 请重新启动 Claude Code。关闭任何可能使用 Chrome 的其他 Claude Code 会话。
- **本机消息传递主机错误**: 如果本机消息传递主机在启动时崩溃, 请尝试重新安装 Claude Code 以重新生成主机配置。

常见错误消息

这些是最常见的错误及其解决方法：

错误	原因	修复
“浏览器扩展程序未连接”	本机消息传递主机无法到达扩展程序	重新启动 Chrome 和 Claude Code，然后运行 <code>/chrome</code> 以重新连接
“未检测到扩展程序”	Chrome 扩展程序未安装或已禁用	在 <code>chrome://extensions</code> 中安装或启用扩展程序
“没有可用的标签页”	Claude 在标签页准备好之前尝试操作	要求 Claude 创建新标签页并重试
“接收端不存在”	扩展程序 service worker 进入空闲状态	运行 <code>/chrome</code> 并选择”重新连接扩展程序“

另请参阅

- [在 VS Code 中使用 Claude Code](#)：VS Code 扩展程序中的浏览器自动化
- [CLI 参考](#)：命令行标志，包括 `--chrome`
- [常见工作流](#)：更多使用 Claude Code 的方式
- [数据和隐私](#)：Claude Code 如何处理您的数据
- [Claude in Chrome 入门](#)：Chrome 扩展程序的完整文档，包括快捷键、计划和权限

Slack 中的 Claude Code

直接从 Slack 工作区委派编码任务

Slack 中的 Claude Code 将 Claude Code 的强大功能直接引入您的 Slack 工作区。当您使用编码任务提及 `@Claude` 时，Claude 会自动检测意图并在网络上创建 Claude Code 会话，允许您在不离开团队对话的情况下委派开发工作。

此集成基于现有的 Claude for Slack 应用程序构建，但为与编码相关的请求添加了到网络上 Claude Code 的智能路由。

用例

- **Bug 调查和修复：**要求 Claude 在 Slack 频道中报告 Bug 时立即调查和修复。
- **快速代码审查和修改：**让 Claude 根据团队反馈实现小功能或重构代码。
- **协作调试：**当团队讨论提供关键背景信息（例如错误重现或用户报告）时，Claude 可以使用该信息来指导其调试方法。
- **并行任务执行：**在 Slack 中启动编码任务，同时继续其他工作，完成时收到通知。

前置条件

在使用 Slack 中的 Claude Code 之前，请确保您具有以下条件：

要求	详情
Claude 计划	Pro、Max、Team 或 Enterprise，具有 Claude Code 访问权限（高级席位）
网络上的 Claude Code	必须启用对 网络上的 Claude Code 的访问
GitHub 账户	连接到网络上的 Claude Code，至少有一个存储库已认证
Slack 认证	您的 Slack 账户通过 Claude 应用程序链接到您的 Claude 账户

在 Slack 中设置 Claude Code

Step 1: 在 Slack 中安装 Claude 应用程序

工作区管理员必须从 Slack 应用程序市场安装 Claude 应用程序。访问 [Slack 应用程序市场](#) 并单击” Add to Slack” 开始安装过程。

Step 2: 连接您的 Claude 账户

安装应用程序后，认证您的个人 Claude 账户：

- 1. 通过单击您的应用程序部分中的” Claude” 在 Slack 中打开 Claude 应用程序
- 2. 导航到应用程序主页选项卡
- 3. 单击” Connect” 将您的 Slack 账户与您的 Claude 账户链接
- 4. 在浏览器中完成认证流程

Step 3: 配置网络上的 Claude Code

确保您网络上的 Claude Code 已正确配置：

- 访问 claude.ai/code 并使用您连接到 Slack 的同一账户登录
- 如果尚未连接，请连接您的 GitHub 账户
- 认证至少一个您希望 Claude 使用的存储库

Step 4: 选择您的路由模式

连接您的账户后，配置 Claude 如何在 Slack 中处理您的消息。导航到 Slack 中的 Claude 应用程序主页以找到**路由模式**设置。

模式	行为
仅代码	Claude 将所有 @mentions 路由到 Claude Code 会话。最适合仅将 Claude 用于 Slack 中开发任务的团队。
代码 + 聊天	Claude 分析每条消息并在 Claude Code（用于编码任务）和 Claude Chat（用于写作、分析和常见问题）之间智能路由。最适合希望为所有类型工作提供单一 @Claude 入口点的团队。

Note:

在代码 + 聊天模式下，如果 Claude 将消息路由到聊天但您想要编码会话，您可以单击” Retry as Code” 来创建 Claude Code 会话。类似地，如果它被路由到代码但您想要聊天会话，您可以在该线程中选择该选项。

工作原理

自动检测

当您在 Slack 频道或线程中提及 @Claude 时，Claude 会自动分析您的消息以确定它是否是编码任务。如果 Claude 检测到编码意图，它将把您的请求路由到网络上的 Claude Code，而不是作为常规聊天助手响应。

您也可以明确告诉 Claude 将请求作为编码任务处理，即使它没有自动检测到。

Note:

Slack 中的 Claude Code 仅在频道（公开或私有）中工作。它在直接消息 (DM) 中不起作用。

上下文收集

来自线程：当您在线程中 @mention Claude 时，它会从该线程中的所有消息收集上下文以理解完整的对话。

来自频道：当直接在频道中提及，Claude 会查看最近的频道消息以获取相关上下文。

此上下文帮助 Claude 理解问题、选择适当的存储库并指导其任务方法。

Warning:

当在 Slack 中调用 @Claude 时，Claude 可以访问对话上下文以更好地理解您的请求。Claude 可能会遵循上下文中其他消息的指示，因此用户应确保仅在受信任的 Slack 对话中使用 Claude。

会话流程

1. **启动：**您使用编码请求 @mention Claude
2. **检测：**Claude 分析您的消息并检测编码意图
3. **会话创建：**在 claude.ai/code 上创建新的 Claude Code 会话
4. **进度更新：**Claude 在工作进行时向您的 Slack 线程发布状态更新
5. **完成：**完成后，Claude @mentions 您并提供摘要和操作按钮
6. **审查：**单击“View Session”查看完整记录，或单击“Create PR”打开拉取请求

用户界面元素

应用程序主页

应用程序主页选项卡显示您的连接状态，并允许您连接或断开您的 Claude 账户与 Slack 的连接。

消息操作

- **View Session：**在浏览器中打开完整的 Claude Code 会话，您可以在其中查看所有执行的工作、继续会话或提出其他请求。
- **Create PR：**直接从会话的更改创建拉取请求。

- **Retry as Code**: 如果 Claude 最初作为聊天助手响应但您想要编码会话，请单击此按钮将请求重试为 Claude Code 任务。
- **Change Repo**: 如果 Claude 选择不正确，允许您选择不同的存储库。

存储库选择

Claude 根据 Slack 对话中的上下文自动选择存储库。如果多个存储库可能适用，Claude 可能会显示一个下拉菜单，允许您选择正确的存储库。

访问和权限

用户级访问

访问类型	要求
Claude Code 会话	每个用户在其自己的 Claude 账户下运行会话
使用情况和速率限制	会话计入个人用户的计划限制
存储库访问	用户只能访问他们个人连接的存储库
会话历史	会话出现在您在 <code>claude.ai/code</code> 上的 Claude Code 历史中

工作区管理员权限

Slack 工作区管理员控制 Claude 应用程序是否可以在工作区中安装。然后，各个用户使用自己的 Claude 账户进行认证以使用集成。

什么可以在哪里访问

在 Slack 中: 您将看到状态更新、完成摘要和操作按钮。完整记录被保留并始终可访问。

在网络上: 完整的 Claude Code 会话，包含完整的对话历史、所有代码更改、文件操作以及继续会话或创建拉取请求的能力。

最佳实践

编写有效的请求

- **具体说明**: 在相关时包括文件名、函数名或错误消息。
- **提供上下文**: 如果从对话中不清楚，请提及存储库或项目。
- **定义成功**: 解释“完成”的样子——Claude 应该编写测试吗？更新文档？创建 PR？
- **使用线程**: 在讨论 Bug 或功能时在线程中回复，以便 Claude 可以收集完整的上下文。

何时使用 Slack 与网络

在以下情况下使用 Slack：上下文已存在于 Slack 讨论中，您想异步启动任务，或者您正在与需要可见性的团队成员协作。

直接在网络上使用：当您需要上传文件、想要在开发过程中进行实时交互或处理更长、更复杂的任务时。

故障排除

会话未启动

1. 验证您的 Claude 账户在 Claude 应用程序主页中已连接
2. 检查您是否启用了网络上的 Claude Code 访问权限
3. 确保您至少有一个 GitHub 存储库连接到 Claude Code

存储库未显示

1. 在 claude.ai/code 的网络上的 Claude Code 中连接存储库
2. 验证您对该存储库的 GitHub 权限
3. 尝试断开并重新连接您的 GitHub 账户

选择了错误的存储库

1. 单击“Change Repo”按钮选择不同的存储库
2. 在您的请求中包括存储库名称以获得更准确的选择

认证错误

1. 在应用程序主页中断开并重新连接您的 Claude 账户
2. 确保您在浏览器中登录到正确的 Claude 账户
3. 检查您的 Claude 计划是否包括 Claude Code 访问权限

会话过期

1. 会话在网络上的 Claude Code 历史中保持可访问
2. 您可以从 claude.ai/code 继续或参考过去的会话

当前限制

- **仅 GitHub：**目前支持 GitHub 上的存储库。
- **一次一个 PR：**每个会话可以创建一个拉取请求。
- **速率限制适用：**会话使用您的个人 Claude 计划的速率限制。

- **需要网络访问：**用户必须具有网络上的 Claude Code 访问权限；没有它的用户将只获得标准 Claude 聊天响应。

相关资源

- [网络上的 Claude Code](#) 了解有关网络上的 Claude Code 的更多信息
- [Claude for Slack](#) Claude for Slack 常规文档
- [Slack 应用程序市场](#) 从 Slack 市场安装 Claude 应用程序
- [Claude 帮助中心](#) 获取额外支持

Claude Code on the web

在安全的云基础设施上异步运行 Claude Code 任务

Note:

Claude Code on the web 目前处于研究预览阶段。

Claude Code on the web 是什么？

Claude Code on the web 让开发者可以从 Claude 应用中启动 Claude Code。这非常适合：

- **回答问题：**询问代码架构和功能实现方式
- **错误修复和日常任务：**不需要频繁调整的明确定义的任务
- **并行工作：**同时处理多个错误修复
- **不在本地机器上的存储库：**处理你没有本地检出的代码
- **后端更改：**Claude Code 可以编写测试，然后编写代码来通过这些测试

Claude Code 也可在 Claude 应用中用于 [iOS](#) 和 [Android](#)，用于随时启动任务和监控进行中的工作。

你可以从终端使用 `--remote` 在网络上启动新任务，或将网络会话传送回你的终端以在本地继续。要在运行 Claude Code 在自己的机器上而不是云基础设施上使用网络界面，请参阅[远程控制](#)。

谁可以使用 Claude Code on the web？

Claude Code on the web 在研究预览中可供以下用户使用：

- **Pro 用户**
- **Max 用户**
- **Team 用户**
- **Enterprise 用户**（拥有高级席位或 Chat + Claude Code 席位）

入门

1. 访问 claude.ai/code

2. 连接你的 GitHub 账户
3. 在你的存储库中安装 Claude GitHub 应用
4. 选择你的默认环境
5. 提交你的编码任务
6. 在 diff 视图中审查更改，通过评论进行迭代，然后创建拉取请求

工作原理

当你在 Claude Code on the web 上启动任务时：

1. **存储库克隆**：你的存储库被克隆到 Anthropic 管理的虚拟机
2. **环境设置**：Claude 准备一个安全的云环境，包含你的代码，然后运行你的[设置脚本](#)（如果已配置）
3. **网络配置**：根据你的设置配置互联网访问
4. **任务执行**：Claude 分析代码、进行更改、运行测试并检查其工作
5. **完成**：你会收到完成通知，可以使用更改创建 PR
6. **结果**：更改被推送到一个分支，准备好创建拉取请求

使用 diff 视图审查更改

Diff 视图让你在创建拉取请求之前看到 Claude 更改的确切内容。与其点击”创建 PR”在 GitHub 中审查更改，不如直接在应用中查看 diff 并与 Claude 迭代，直到更改准备好。

当 Claude 对文件进行更改时，会出现一个 diff 统计指示器，显示添加和删除的行数（例如，**+12 -1**）。选择此指示器打开 diff 查看器，左侧显示文件列表，右侧显示每个文件的更改。

从 diff 视图中，你可以：

- 逐个文件审查更改
- 对特定更改进行评论以请求修改
- 根据你看到的内容继续与 Claude 迭代

这让你可以通过多轮反馈来完善更改，而无需创建草稿 PR 或切换到 GitHub。

在网络和终端之间移动任务

你可以从终端在网络上启动新任务，或将网络会话拉入你的终端以在本地继续。网络会话即使在你关闭笔记本电脑后也会持续，你可以从任何地方（包括 Claude 移动应用）监控它们。

Note:

会话切换是单向的：你可以将网络会话拉入你的终端，但不能将现有的终端会话推送到网络。`--remote` 标志为你的当前存储库创建一个新的网络会话。

从终端到网络

使用 `--remote` 标志从命令行启动网络会话：

```
claude --remote "Fix the authentication bug in src/auth/login.ts"
```

这在 `claude.ai` 上创建一个新的网络会话。任务在云中运行，而你继续在本地工作。使用 `/tasks` 检查进度，或在 `claude.ai` 或 Claude 移动应用上打开会话以直接交互。从那里你可以引导 Claude、提供反馈或回答问题，就像任何其他对话一样。

远程任务的提示

在本地规划，远程执行：对于复杂的任务，在 Plan Mode 中启动 Claude 以协作制定方法，然后将工作发送到网络：

```
claude --permission-mode plan
```

在 Plan Mode 中，Claude 只能读取文件和探索代码库。一旦你对计划满意，为自主执行启动远程会话：

```
claude --remote "Execute the migration plan in docs/migration-plan.md"
```

这种模式让你可以控制策略，同时让 Claude 在云中自主执行。

并行运行任务：每个 `--remote` 命令创建自己的网络会话，独立运行。你可以启动多个任务，它们都将在单独的会话中同时运行：

```
claude --remote "Fix the flaky test in auth.spec.ts"
claude --remote "Update the API documentation"
claude --remote "Refactor the logger to use structured output"
```

使用 `/tasks` 监控所有会话。当会话完成时，你可以从网络界面创建 PR 或 [传送](#) 会话到你的终端以继续工作。

从网络到终端

有几种方法可以将网络会话拉入你的终端：

- **使用 `/teleport`**：在 Claude Code 中，运行 `/teleport`（或 `/tp`）以查看你的网络会话的交互式选择器。如果你有未提交的更改，系统会提示你先隐藏它们。
- **使用 `--teleport`**：从命令行，运行 `claude --teleport` 以获得交互式会话选择器，或 `claude --teleport <session-id>` 以直接恢复特定会话。
- **从 `/tasks`**：运行 `/tasks` 以查看你的后台会话，然后按 `t` 传送到其中一个
- **从网络界面**：点击”在 CLI 中打开”以复制可以粘贴到你的终端中的命令

当你传送一个会话时，Claude 验证你在正确的存储库中，从远程会话获取并检出分支，并将完整的对话历史加载到你的终端中。

传送的要求

传送在恢复会话之前检查这些要求。如果任何要求未满足，你会看到错误或被提示解决问题。

要求	详情
干净的 git 状态	你的工作目录必须没有未提交的更改。如果需要，传送会提示你隐藏更改。
正确的存储库	你必须从同一存储库的检出运行 <code>--teleport</code> ，而不是从分叉运行。
分支可用	网络会话中的分支必须已被推送到远程。传送会自动获取并检出它。
相同账户	你必须认证到网络会话中使用的相同 Claude.ai 账户。

共享会话

要共享会话，请根据下面的账户类型切换其可见性。之后，按原样共享会话链接。打开你的共享会话的收件人将在加载时看到会话的最新状态，但收件人的页面不会实时更新。

从 Enterprise 或 Teams 账户共享

对于 Enterprise 和 Teams 账户，两个可见性选项是**私有**和**团队**。团队可见性使会话对你的 Claude.ai 组织的其他成员可见。默认情况下启用存储库访问验证，基于连接到收件人账户的 GitHub 账户。你的账户显示名称对所有有访问权限的收件人可见。[Claude in Slack](#) 会话会自动以团队可见性共享。

从 Max 或 Pro 账户共享

对于 Max 和 Pro 账户，两个可见性选项是**私有**和**公开**。公开可见性使会话对任何登录到 claude.ai 的用户可见。

在共享之前检查你的会话是否包含敏感内容。会话可能包含来自私有 GitHub 存储库的代码和凭证。默认情况下不启用存储库访问验证。

通过转到设置 > Claude Code > 共享设置来启用存储库访问验证和/或从你的共享会话中隐藏你的名称。

管理会话

归档会话

你可以归档会话以保持你的会话列表有序。归档的会话从默认会话列表中隐藏，但可以通过筛选已归档会话来查看。

要归档会话，请在侧边栏中悬停在会话上并点击归档图标。

删除会话

删除会话会永久删除会话及其数据。此操作无法撤销。你可以通过两种方式删除会话：

- **从侧边栏：**筛选已归档会话，然后悬停在你想删除的会话上并点击删除图标
- **从会话菜单：**打开会话，点击会话标题旁的下拉菜单，然后选择删除

删除会话前会要求你确认。

云环境

默认镜像

我们构建并维护一个通用镜像，其中预装了常见的工具链和语言生态系统。此镜像包括：

- 流行的编程语言和运行时
- 常见的构建工具和包管理器
- 测试框架和 linters

检查可用工具

要查看你的环境中预装了什么，请要求 Claude Code 运行：

```
check-tools
```

此命令显示：

- 编程语言及其版本
- 可用的包管理器
- 已安装的开发工具

特定于语言的设置

通用镜像包括以下预配置的环境：

- **Python**：Python 3.x，带有 pip、poetry 和常见的科学库
- **Node.js**：最新 LTS 版本，带有 npm、yarn、pnpm 和 bun
- **Ruby**：版本 3.1.6、3.2.6、3.3.6（默认：3.3.6），带有 gem、bundler 和 rbenv 用于版本管理
- **PHP**：版本 8.4.14
- **Java**：OpenJDK，带有 Maven 和 Gradle
- **Go**：最新稳定版本，带有模块支持
- **Rust**：Rust 工具链，带有 cargo
- **C++**：GCC 和 Clang 编译器

数据库

通用镜像包括以下数据库：

- **PostgreSQL**：版本 16
- **Redis**：版本 7.0

环境配置

当你在 Claude Code on the web 中启动会话时，以下是幕后发生的情况：

1. **环境准备**：我们克隆你的存储库并运行任何已配置的[设置脚本](#)。存储库将使用你的 GitHub 存储库上的默认分支进行克隆。如果你想检出特定分支，可以在提示中指定。
2. **网络配置**：我们为代理配置互联网访问。默认情况下互联网访问受限，但你可以根据需要配置环境以禁用互联网或完全互联网访问。
3. **Claude Code 执行**：Claude Code 运行以完成你的任务，编写代码、运行测试并检查其工作。你可以通过网络界面在整个会话中指导和引导 Claude。Claude 尊重你在 `CLAUDE.md` 中定义的上文。
4. **结果**：当 Claude 完成其工作时，它将推送分支到远程。你将能够为分支创建 PR。

Note:

Claude 完全通过环境中可用的终端和 CLI 工具运行。它使用通用镜像中的预装工具和你通过 hooks 或依赖管理安装的任何其他工具。

要添加新环境：选择当前环境以打开环境选择器，然后选择”添加环境”。这将打开一个对话框，你可以在其中指定环境名称、网络访问级别、环境变量和[设置脚本](#)。

要更新现有环境：选择当前环境，在环境名称的右侧，然后选择设置按钮。这将打开一个对话框，你可以在其中更新环境名称、网络访问、环境变量和设置脚本。

要从终端选择你的默认环境：如果你配置了多个环境，运行 `/remote-env` 以选择在使用 `--remote` 从终端启动网络会话时使用哪个环境。使用单个环境时，此命令显示你的当前配置。

Note:

环境变量必须指定为键值对，采用 `.env` 格式。例如：

```
API_KEY=your_api_key
DEBUG=true
```

设置脚本

设置脚本是一个 Bash 脚本，在新的云会话启动时运行，在 Claude Code 启动之前。使用设置脚本来安装依赖、配置工具或准备云环境需要的任何东西，这些东西不在默认镜像中。

脚本在 Ubuntu 24.04 上以 root 身份运行，所以 `apt install` 和大多数语言包管理器都可以工作。

Tip:

要在将其添加到你的脚本之前检查已安装的内容，请要求 Claude 在云会话中运行 `check-tools`。

要添加设置脚本，请打开环境设置对话框并在**设置脚本**字段中输入你的脚本。

此示例安装 `gh` CLI，它不在默认镜像中：

```
#!/bin/bash
apt update && apt install -y gh
```

设置脚本仅在创建新会话时运行。恢复现有会话时会跳过它们。

如果脚本以非零值退出，会话将无法启动。将 `|| true` 附加到非关键命令以避免在不稳定的安装上阻止会话。

Note:

安装包的设置脚本需要网络访问才能到达注册表。默认网络访问允许连接到[常见包注册表](#)，包括 npm、PyPI、RubyGems 和 crates.io。如果你的环境禁用了网络访问，脚本将无法安装包。

设置脚本与 SessionStart hooks

使用设置脚本来安装云需要但你的笔记本电脑已有的东西，如语言运行时或 CLI 工具。使用 [SessionStart hook](#) 进行应该在任何地方运行的项目设置，云和本地，如 `npm install`。

两者都在会话开始时运行，但它们属于不同的地方：

	设置脚本	SessionStart hooks
附加到	云环境	你的存储库
配置在	云环境 UI	你的存储库中的 <code>.claude/settings.json</code>
运行	在 Claude Code 启动之前，仅在新会话上	在 Claude Code 启动之后，在每个会话上，包括已恢复的
范围	仅云环境	本地和云

SessionStart hooks 也可以在你的用户级 `~/.claude/settings.json` 中本地定义，但用户级设置不会传送到云会话。在云中，只有提交到存储库的 hooks 运行。

依赖管理

自定义环境镜像和快照尚不支持。使用 [设置脚本](#) 在会话启动时安装包，或使用 [SessionStart hooks](#) 进行也应在本地环境中运行的依赖安装。SessionStart hooks 有 [已知限制](#)。

要使用设置脚本配置自动依赖安装，请打开你的环境设置并添加脚本：

```
#!/bin/bash
npm install
pip install -r requirements.txt
```

或者，你可以在你的存储库的 `.claude/settings.json` 文件中使用 SessionStart hooks 进行也应在本地环境中运行的依赖安装：

```
{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "startup",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/scripts/install_pkgs.sh"
          }
        ]
      }
    ]
  }
}
```

在 `scripts/install_pkgs.sh` 创建相应的脚本：

```
#!/bin/bash

## Only run in remote environments
if [ "$CLAUDE_CODE_REMOTE" ≠ "true" ]; then
  exit 0
fi

npm install
pip install -r requirements.txt
exit 0
```

使其可执行： `chmod +x scripts/install_pkgs.sh`

持久化环境变量

SessionStart hooks 可以通过写入 `CLAUDE_ENV_FILE` 环境变量中指定的文件来为后续 Bash 命令持久化环境变量。有关详情，请参阅 hooks 参考中的[SessionStart hooks](#)。

依赖管理限制

- **Hooks 对所有会话触发**：SessionStart hooks 在本地和远程环境中都运行。没有 hook 配置来仅将 hook 限定到远程会话。要跳过本地执行，请在你的脚本中检查 `CLAUDE_CODE_REMOTE` 环境变量，如上所示。
- **需要网络访问**：安装命令需要网络访问才能到达包注册表。如果你的环境配置为”无互联网”访问，这些 hooks 将失败。使用”受限”（默认）或”完全”网络访问。[默认允许列表](#)包括常见的注册表，如 npm、PyPI、RubyGems 和 crates.io。
- **代理兼容性**：远程环境中的所有出站流量都通过[安全代理](#)。某些包管理器不能与此代理正确配合使用。Bun 是一个已知的例子。
- **在每个会话启动时运行**：Hooks 在每次会话启动或恢复时运行，增加启动延迟。通过在重新安装之前检查依赖是否已存在来保持安装脚本快速。

网络访问和安全

网络策略

GitHub 代理

为了安全起见，所有 GitHub 操作都通过专用代理服务进行，该服务透明地处理所有 git 交互。在沙箱内，git 客户端使用自定义构建的作用域凭证进行身份验证。此代理：

- 安全地管理 GitHub 身份验证 - git 客户端在沙箱内使用作用域凭证，代理验证并将其转换为你的实际 GitHub 身份验证令牌
- 限制 git push 操作到当前工作分支以确保安全
- 启用无缝克隆、获取和 PR 操作，同时维护安全边界

安全代理

环境在 HTTP/HTTPS 网络代理后面运行，用于安全和滥用防止目的。所有出站互联网流量都通过此代理，该代理提供：

- 防止恶意请求
- 速率限制和滥用防止
- 增强安全性的内容过滤

访问级别

默认情况下，网络访问仅限于[允许列表域](#)。

你可以配置自定义网络访问，包括禁用网络访问。

默认允许的域

使用”受限”网络访问时，默认允许以下域：

Anthropic 服务

- api.anthropic.com
- statsig.anthropic.com
- platform.claude.com
- code.claude.com
- claude.ai

版本控制

- github.com
- www.github.com
- api.github.com
- npm.pkg.github.com
- raw.githubusercontent.com
- pkg-npm.githubusercontent.com
- objects.githubusercontent.com
- codeload.github.com
- avatars.githubusercontent.com
- camo.githubusercontent.com
- gist.github.com
- gitlab.com
- www.gitlab.com
- registry.gitlab.com
- bitbucket.org
- www.bitbucket.org
- api.bitbucket.org

容器注册表

- registry-1.docker.io
- auth.docker.io
- index.docker.io
- hub.docker.com

- www.docker.com
- production.cloudflare.docker.com
- download.docker.com
- gcr.io
- *.gcr.io
- ghcr.io
- mcr.microsoft.com
- *.data.mcr.microsoft.com
- public.ecr.aws

云平台

- cloud.google.com
- accounts.google.com
- gcloud.google.com
- *.googleapis.com
- storage.googleapis.com
- compute.googleapis.com
- container.googleapis.com
- azure.com
- portal.azure.com
- microsoft.com
- www.microsoft.com
- *.microsoftonline.com
- packages.microsoft.com
- dotnet.microsoft.com
- dot.net
- visualstudio.com
- dev.azure.com
- *.amazonaws.com
- *.api.aws
- oracle.com
- www.oracle.com
- java.com
- www.java.com

- java.net
- www.java.net
- download.oracle.com
- yum.oracle.com

包管理器 - JavaScript/Node

- registry.npmjs.org
- www.npmjs.com
- www.npmjs.org
- npmjs.com
- npmjs.org
- yarnpkg.com
- registry.yarnpkg.com

包管理器 - Python

- pypi.org
- www.pypi.org
- files.pythonhosted.org
- pythonhosted.org
- test.pypi.org
- pypi.python.org
- pypa.io
- www.pypa.io

包管理器 - Ruby

- rubygems.org
- www.rubygems.org
- api.rubygems.org
- index.rubygems.org
- ruby-lang.org
- www.ruby-lang.org
- rubyforge.org
- www.rubyforge.org
- rubyonrails.org

- www.rubyonrails.org
- rvm.io
- get.rvm.io

包管理器 - Rust

- crates.io
- www.crates.io
- index.crates.io
- static.crates.io
- rustup.rs
- static.rust-lang.org
- www.rust-lang.org

包管理器 - Go

- proxy.golang.org
- sum.golang.org
- index.golang.org
- golang.org
- www.golang.org
- goproxy.io
- pkg.go.dev

包管理器 - JVM

- maven.org
- repo.maven.org
- central.maven.org
- repo1.maven.org
- jcenter.bintray.com
- gradle.org
- www.gradle.org
- services.gradle.org
- plugins.gradle.org
- kotlin.org
- www.kotlin.org

- spring.io
- repo.spring.io

包管理器 - 其他语言

- packagist.org (PHP Composer)
- www.packagist.org
- repo.packagist.org
- nuget.org (.NET NuGet)
- www.nuget.org
- api.nuget.org
- pub.dev (Dart/Flutter)
- api.pub.dev
- hex.pm (Elixir/Erlang)
- www.hex.pm
- cpan.org (Perl CPAN)
- www.cpan.org
- metacpan.org
- www.metacpan.org
- api.metacpan.org
- cocoapods.org (iOS/macOS)
- www.cocoapods.org
- cdn.cocoapods.org
- haskell.org
- www.haskell.org
- hackage.haskell.org
- swift.org
- www.swift.org

Linux 发行版

- archive.ubuntu.com
- security.ubuntu.com
- ubuntu.com
- www.ubuntu.com
- *.ubuntu.com

- ppa.launchpad.net
- launchpad.net
- www.launchpad.net

开发工具和平台

- dl.k8s.io (Kubernetes)
- pkgs.k8s.io
- k8s.io
- www.k8s.io
- releases.hashicorp.com (HashiCorp)
- apt.releases.hashicorp.com
- rpm.releases.hashicorp.com
- archive.releases.hashicorp.com
- hashicorp.com
- www.hashicorp.com
- repo.anaconda.com (Anaconda/Conda)
- conda.anaconda.org
- anaconda.org
- www.anaconda.com
- anaconda.com
- continuum.io
- apache.org (Apache)
- www.apache.org
- archive.apache.org
- downloads.apache.org
- eclipse.org (Eclipse)
- www.eclipse.org
- download.eclipse.org
- nodejs.org (Node.js)
- www.nodejs.org

云服务和监控

- statsig.com
- www.statsig.com

- `api.statsig.com`
- `sentry.io`
- `*.sentry.io`
- `http-intake.logs.datadoghq.com`
- `*.datadoghq.com`
- `*.datadoghq.eu`

内容交付和镜像

- `sourceforge.net`
- `*.sourceforge.net`
- `packagecloud.io`
- `*.packagecloud.io`

架构和配置

- `json-schema.org`
- www.json-schema.org
- `json.schemastore.org`
- www.schemastore.org

Model Context Protocol

- `*.modelcontextprotocol.io`

Note:

标记为 `*` 的域表示通配符子域匹配。例如，`*.gcr.io` 允许访问 `gcr.io` 的任何子域。

自定义网络访问的安全最佳实践

1. **最小权限原则**：仅启用所需的最小网络访问
2. **定期审计**：定期审查允许的域
3. **使用 HTTPS**：始终优先使用 HTTPS 端点而不是 HTTP

安全和隔离

Claude Code on the web 提供强大的安全保证：

- **隔离的虚拟机**：每个会话在隔离的、Anthropic 管理的 VM 中运行
- **网络访问控制**：网络访问默认受限，可以禁用

Note:

在禁用网络访问的情况下运行时，Claude Code 被允许与 Anthropic API 通信，这可能仍然允许数据从隔离的 Claude Code VM 中退出。

- **凭证保护**：敏感凭证（如 git 凭证或签名密钥）永远不会在沙箱内与 Claude Code 一起。身份验证通过使用作用域凭证的安全代理处理
- **安全分析**：代码在隔离的 VM 内分析和修改，然后创建 PR

定价和速率限制

Claude Code on the web 与你账户内所有其他 Claude 和 Claude Code 使用共享速率限制。并行运行多个任务将按比例消耗更多速率限制。

限制

- **存储库身份验证**：你只能在认证到相同账户时将会话从网络移动到本地
- **平台限制**：Claude Code on the web 仅适用于托管在 GitHub 中的代码。GitLab 和其他非 GitHub 存储库不能与云会话一起使用

最佳实践

1. **自动化环境设置**：使用[设置脚本](#)在 Claude Code 启动之前安装依赖和配置工具。对于更高级的场景，配置[SessionStart hooks](#)。
2. **记录要求**：在你的 `CLAUDE.md` 文件中清楚地指定依赖和命令。如果你有 `AGENTS.md` 文件，你可以在你的 `CLAUDE.md` 中使用 `@AGENTS.md` 来源它以维护单一的真实来源。

相关资源

- [Hooks 配置](#)
- [设置参考](#)
- [安全](#)
- [数据使用](#)

Part 9: Security & Privacy

安全性

了解 Claude Code 的安全防护措施和安全使用的最佳实践。

我们如何处理安全性

安全基础

您的代码安全至关重要。Claude Code 以安全为核心构建，按照 Anthropic 的全面安全计划开发。在 [Anthropic Trust Center](#) 了解更多信息并访问资源（SOC 2 Type 2 报告、ISO 27001 证书等）。

基于权限的架构

Claude Code 默认使用严格的只读权限。当需要额外操作时（编辑文件、运行测试、执行命令），Claude Code 会请求明确的权限。用户可以控制是否批准一次性操作或自动允许操作。

我们设计 Claude Code 是为了透明和安全。例如，我们要求在执行 bash 命令前获得批准，让您拥有直接控制权。这种方法使用户和组织能够直接配置权限。

有关详细的权限配置，请参阅 [Permissions](#)。

内置保护

为了降低代理系统中的风险：

- **沙箱化 bash 工具：**使用文件系统和网络隔离的 [Sandbox](#) bash 命令，减少权限提示同时保持安全性。使用 `/sandbox` 启用以定义 Claude Code 可以自主工作的边界
- **写入访问限制：**Claude Code 只能写入启动它的文件夹及其子文件夹——它不能在明确权限的情况下修改父目录中的文件。虽然 Claude Code 可以读取工作目录外的文件（对于访问系统库和依赖项很有用），但写入操作严格限制在项目范围内，创建了清晰的安全边界
- **提示疲劳缓解：**支持按用户、按代码库或按组织的白名单常用安全命令
- **Accept Edits 模式：**批量接受多个编辑，同时为具有副作用的命令保持权限提示

用户责任

Claude Code 只拥有您授予它的权限。您负责在批准前审查建议的代码和命令的安全性。

防止提示注入

提示注入是一种攻击者试图通过插入恶意文本来覆盖或操纵 AI 助手指令的技术。Claude Code 包括针对这些攻击的多项防护措施：

核心保护

- **权限系统**：敏感操作需要明确批准
- **上下文感知分析**：通过分析完整请求来检测潜在有害指令
- **输入清理**：通过处理用户输入来防止命令注入
- **命令黑名单**：默认阻止从网络获取任意内容的风险命令，如 `curl` 和 `wget`。显式允许时，请注意 [权限模式限制](#)

隐私保护

我们实施了多项保护措施来保护您的数据，包括：

- 敏感信息的有限保留期（请参阅 [Privacy Center](#) 了解更多）
- 对用户会话数据的受限访问
- 用户对数据训练偏好的控制。消费者用户可以随时更改其 [隐私设置](#)。

有关完整详情，请查看我们的 [Commercial Terms of Service](#)（适用于 Team、Enterprise 和 API 用户）或 [Consumer Terms](#)（适用于 Free、Pro 和 Max 用户）以及 [Privacy Policy](#)。

其他保护措施

- **网络请求批准**：进行网络请求的工具默认需要用户批准
- **隔离的上下文窗口**：Web fetch 使用单独的上下文窗口以避免注入潜在恶意提示
- **信任验证**：首次代码库运行和新 MCP servers 需要信任验证
 - 注意：使用 `-p` 标志以非交互方式运行时，信任验证被禁用
- **命令注入检测**：即使之前已白名单，可疑的 bash 命令也需要手动批准
- **故障关闭匹配**：不匹配的命令默认需要手动批准
- **自然语言描述**：复杂的 bash 命令包括用户理解的说明
- **安全凭证存储**：API 密钥和令牌已加密。请参阅 [Credential Management](#)

Warning:

Windows WebDAV 安全风险：在 Windows 上运行 Claude Code 时，我们建议不要启用 WebDAV 或允许 Claude Code 访问可能包含 WebDAV 子目录的路径，如 `\\*`。WebDAV 已被 Microsoft 弃用，原因是安全风险。启用 WebDAV 可能允许 Claude Code 触发对远程主机的网络请求，绕过权限系统。

处理不受信任内容的最佳实践：

1. 在批准前审查建议的命令
2. 避免直接将不受信任的内容通过管道传递给 Claude
3. 验证对关键文件的建议更改
4. 使用虚拟机 (VM) 运行脚本并进行工具调用，特别是在与外部 Web 服务交互时
5. 使用 `/bug` 报告可疑行为

Warning:

虽然这些保护措施大大降低了风险，但没有系统完全免疫所有攻击。在使用任何 AI 工具时，始终保持良好的安全实践。

MCP 安全性

Claude Code 允许用户配置 Model Context Protocol (MCP) servers。允许的 MCP servers 列表在您的源代码中配置，作为 Claude Code 设置的一部分，工程师将其检入源代码控制。

我们鼓励编写您自己的 MCP servers 或使用来自您信任的提供商的 MCP servers。您能够为 MCP servers 配置 Claude Code 权限。Anthropic 不管理或审计任何 MCP servers。

IDE 安全性

有关在 IDE 中运行 Claude Code 的更多信息，请参阅 [VS Code security and privacy](#)。

云执行安全性

使用 [Claude Code on the web](#) 时，会实施额外的安全控制：

- **隔离的虚拟机：**每个云会话在隔离的、由 Anthropic 管理的 VM 中运行
- **网络访问控制：**网络访问默认受限，可以配置为禁用或仅允许特定域
- **凭证保护：**身份验证通过安全代理处理，该代理在沙箱内使用作用域凭证，然后转换为您的实际 GitHub 身份验证令牌
- **分支限制：**Git push 操作限制在当前工作分支

- **审计日志**：云环境中的所有操作都被记录以用于合规和审计目的
- **自动清理**：会话完成后，云环境会自动终止

有关云执行的更多详情，请参阅 [Claude Code on the web](#)。

Remote Control 会话的工作方式不同：Web 界面连接到在您本地机器上运行的 Claude Code 进程。所有代码执行和文件访问都保持本地，任何本地 Claude Code 会话期间流动的相同数据通过 TLS 上的 Anthropic API 传输。不涉及云 VM 或沙箱。连接使用多个短期的、范围狭窄的凭证，每个凭证限制于特定目的并独立过期，以限制任何单个受损凭证的影响范围。

安全最佳实践

处理敏感代码

- 在批准前审查所有建议的更改
- 为敏感存储库使用项目特定的权限设置
- 考虑使用 [devcontainers](#) 以获得额外隔离
- 使用 `/permissions` 定期审计您的权限设置

团队安全

- 使用 [managed settings](#) 来强制执行组织标准
- 通过版本控制共享批准的权限配置
- 培训团队成员了解安全最佳实践
- 通过 [OpenTelemetry metrics](#) 监控 Claude Code 使用情况
- 使用 [ConfigChange hooks](#) 审计或阻止会话期间的设置更改

报告安全问题

如果您在 Claude Code 中发现安全漏洞：

1. 不要公开披露
2. 通过我们的 [HackerOne program](#) 报告
3. 包括详细的复现步骤
4. 在公开披露前给我们时间来解决问题

相关资源

- [Sandboxing](#) - bash 命令的文件系统和网络隔离
- [Permissions](#) - 配置权限和访问控制
- [Monitoring usage](#) - 跟踪和审计 Claude Code 活动

- [Development containers](#) - 安全、隔离的环境
- [Anthropic Trust Center](#) - 安全认证和合规

Sandboxing

了解 Claude Code 的沙箱 bash 工具如何提供文件系统和网络隔离，以实现更安全、更自主的代理执行。

概述

Claude Code 具有原生沙箱功能，为代理执行提供更安全的环境，同时减少了对持续权限提示的需求。Claude Code 不是要求对每个 bash 命令进行权限批准，而是预先创建定义的边界，使 Claude Code 能够以降低的风险更自由地工作。

沙箱 bash 工具使用操作系统级原语来强制执行文件系统和网络隔离。

为什么沙箱很重要

传统的基于权限的安全性需要对 bash 命令进行持续的用户批准。虽然这提供了控制，但可能导致：

- **批准疲劳：**重复点击“批准”可能导致用户对他们批准的内容关注度降低
- **生产力降低：**持续的中断会减慢开发工作流程
- **自主性受限：**当等待批准时，Claude Code 无法高效工作

沙箱通过以下方式解决这些挑战：

1. **定义清晰的边界：**精确指定 Claude Code 可以访问的目录和网络主机
2. **减少权限提示：**沙箱内的安全命令不需要批准
3. **维护安全性：**尝试访问沙箱外的资源会触发立即通知
4. **启用自主性：**Claude Code 可以在定义的限制内更独立地运行

Warning:

有效的沙箱需要**同时**进行文件系统和网络隔离。没有网络隔离，被破坏的代理可能会泄露敏感文件，如 SSH 密钥。没有文件系统隔离，被破坏的代理可能会后门系统资源以获得网络访问权限。配置沙箱时，重要的是确保配置的设置不会在这些系统中创建绕过。

工作原理

文件系统隔离

沙箱 bash 工具将文件系统访问限制在特定目录：

- **默认写入行为：**对当前工作目录及其子目录的读写访问
- **默认读取行为：**对整个计算机的读取访问，除了某些被拒绝的目录
- **被阻止的访问：**无法在没有明确权限的情况下修改当前工作目录外的文件
- **可配置：**通过设置定义自定义允许和拒绝的路径

您可以使用设置中的 `sandbox.filesystem.allowWrite` 向其他路径授予写入访问权限。这些限制在操作系统级别强制执行（macOS 上的 Seatbelt，Linux 上的 bubblewrap），因此它们适用于所有子进程命令，包括 `kubectl`、`terraform` 和 `npm` 等工具，而不仅仅是 Claude 的文件工具。

网络隔离

网络访问通过在沙箱外运行的代理服务器进行控制：

- **域名限制：**只能访问批准的域名
- **用户确认：**新的域名请求会触发权限提示（除非启用了 `allowManagedDomainsOnly`，它会自动阻止非允许的域名）
- **自定义代理支持：**高级用户可以在出站流量上实现自定义规则
- **全面覆盖：**限制适用于所有脚本、程序和由命令生成的子进程

操作系统级强制执行

沙箱 bash 工具利用操作系统安全原语：

- **macOS：**使用 Seatbelt 进行沙箱强制执行
- **Linux：**使用 `bubblewrap` 进行隔离
- **WSL2：**使用 bubblewrap，与 Linux 相同

不支持 WSL1，因为 bubblewrap 需要仅在 WSL2 中可用的内核功能。

这些操作系统级限制确保由 Claude Code 命令生成的所有子进程都继承相同的安全边界。

入门

前置条件

在 **macOS** 上，沙箱使用内置的 Seatbelt 框架开箱即用。

在 **Linux 和 WSL2** 上，首先安装所需的包：

Ubuntu/Debian

```
sudo apt-get install bubblewrap socat
```

Fedora

```
sudo dnf install bubblewrap socat
```

启用沙箱

您可以通过运行 `/sandbox` 命令来启用沙箱：

```
/sandbox
```

这会打开一个菜单，您可以在其中选择沙箱模式。如果缺少所需的依赖项（例如 Linux 上的 `bubblewrap` 或 `socat`），菜单会显示您平台的安装说明。

沙箱模式

Claude Code 提供两种沙箱模式：

自动允许模式： Bash 命令将尝试在沙箱内运行，并自动允许而无需权限。无法沙箱化的命令（例如需要访问非允许主机的网络访问的命令）会回退到常规权限流程。您配置的显式询问/拒绝规则始终被尊重。

常规权限模式： 所有 bash 命令都通过标准权限流程，即使是沙箱化的。这提供了更多控制，但需要更多批准。

在两种模式中，沙箱都强制执行相同的文件系统和网络限制。区别仅在于沙箱化命令是自动批准还是需要明确权限。

Info:

自动允许模式独立于您的权限模式设置工作。即使您不在“接受编辑”模式中，启用自动允许时沙箱化的 bash 命令也会自动运行。这意味着在沙箱边界内修改文件的 bash 命令将执行而不提示，即使文件编辑工具通常需要批准。

配置沙箱

通过 `settings.json` 文件自定义沙箱行为。有关完整的配置参考，请参阅 [Settings](#)。

向特定路径授予子进程写入访问权限

默认情况下，沙箱化命令只能写入当前工作目录。如果子进程命令（如 `kubectl`、`terraform` 或 `npm`）需要在项目目录外写入，请使用 `sandbox.filesystem.allowWrite` 向特定路径授予访问权限：

```
{
  "sandbox": {
    "enabled": true,
    "filesystem": {
      "allowWrite": ["~/.kube", "//tmp/build"]
    }
  }
}
```

这些路径在操作系统级别强制执行，因此在沙箱内运行的所有命令（包括其子进程）都尊重它们。当工具需要对特定位置的写入访问时，这是推荐的方法，而不是使用 `excludedCommands` 将工具从沙箱中排除。

当在多个 `settings scopes` 中定义 `allowWrite`（或 `denyWrite` / `denyRead`）时，数组被合并，这意味着来自每个范围的路径被组合，而不是替换。例如，如果托管设置允许写入 `//opt/company-tools`，用户在其个人设置中添加 `~/.kube`，则两个路径都包含在最终沙箱配置中。这意味着用户和项目可以扩展列表而无需复制或覆盖由更高优先级范围设置的路径。

路径前缀控制路径的解析方式：

前缀	含义	示例
//	从文件系统根目录的绝对路径	//tmp/build 变为 /tmp/build
~/	相对于主目录	~/.kube 变为 \$HOME/.kube
/	相对于设置文件的目录	/build 变为 \$SETTINGS_DIR/build
./ 或无前缀	相对路径（由沙箱运行时解析）	./output

您也可以使用 `sandbox.filesystem.denyWrite` 和 `sandbox.filesystem.denyRead` 拒绝写入或读取访问。这些与来自 `Edit(...)` 和 `Read(...)` 权限规则的任何路径合并。

Tip:

并非所有命令都与沙箱开箱即用兼容。一些可能帮助您充分利用沙箱的注意事项：

- 许多 CLI 工具需要访问某些主机。当您使用这些工具时，它们会请求访问某些主机的权限。授予权限将允许它们现在和将来访问这些主机，使它们能够在沙箱内安全执行。
- `watchman` 与在沙箱中运行不兼容。如果您运行 `jest`，请考虑使用 `jest --no-watchman`
- `docker` 与在沙箱中运行不兼容。考虑在 `excludedCommands` 中指定 `docker` 以强制其在沙箱外运行。

Note:

Claude Code 包括一个有意的逃生舱机制，允许命令在必要时在沙箱外运行。当命令由于沙箱限制（例如网络连接问题或不兼容的工具）失败时，Claude 会被提示分析失败，并可能使用 `dangerouslyDisableSandbox` 参数重试命令。使用此参数的命令通过需要用户权限执行的常规 Claude Code 权限流程。这允许 Claude Code 处理某些工具或网络操作无法在沙箱约束内运行的边界情况。

您可以通过在 [sandbox settings](#) 中设置 `"allowUnsandboxedCommands": false` 来禁用此逃生舱。禁用时，`dangerouslyDisableSandbox` 参数被完全忽略，所有命令必须沙箱化运行或在 `excludedCommands` 中明确列出。

安全优势

防止提示注入

即使攻击者通过提示注入成功操纵 Claude Code 的行为，沙箱也确保您的系统保持安全：

文件系统保护：

- 无法修改关键配置文件，如 `~/.bashrc`
- 无法修改 `/bin/` 中的系统级文件
- 无法读取在您的 [Claude 权限设置](#) 中被拒绝的文件

网络保护：

- 无法向攻击者控制的服务器泄露数据
- 无法从未授权的域下载恶意脚本
- 无法向未批准的服务进行意外的 API 调用
- 无法联系任何未明确允许的域

监控和控制：

- 所有在沙箱外的访问尝试都在操作系统级别被阻止
- 当边界被测试时，您会收到立即通知
- 您可以选择拒绝、允许一次或永久更新您的配置

减少攻击面

沙箱限制了以下可能造成的损害：

- **恶意依赖项**：具有有害代码的 NPM 包或其他依赖项
- **被破坏的脚本**：具有安全漏洞的构建脚本或工具
- **社会工程**：欺骗用户运行危险命令的攻击
- **提示注入**：欺骗 Claude 运行危险命令的攻击

透明操作

当 Claude Code 尝试访问沙箱外的网络资源时：

1. 操作在操作系统级别被阻止
2. 您会收到立即通知
3. 您可以选择：
 - 拒绝请求
 - 允许一次
 - 更新您的沙箱配置以永久允许它

安全限制

- **网络沙箱限制**：网络过滤系统通过限制进程允许连接的域来运行。它不会以其他方式检查通过代理的流量，用户负责确保他们只在其策略中允许受信任的域。

Warning:

用户应该意识到允许广泛域名（如 github.com）可能允许数据泄露的潜在风险。此外，在某些情况下，可能可以通过 [domain fronting](#) 绕过网络过滤。

- **通过 Unix Sockets 的权限提升**：`allowUnixSockets` 配置可能会无意中授予对可能导致沙箱绕过的强大系统服务的访问权限。例如，如果它用于允许访问 `/var/run/docker.sock`，这将有效地通过利用 docker socket 授予对主机系统的访问权限。鼓励用户仔细考虑他们通过沙箱允许的任何 unix sockets。

- 文件系统权限提升：过于宽泛的文件系统写入权限可能导致权限提升攻击。允许写入包含 `$PATH` 中的可执行文件、系统配置目录或用户 shell 配置文件（`.bashrc`、`.zshrc`）的目录可能导致当其他用户或系统进程访问这些文件时在不同的安全上下文中执行代码。
- Linux 沙箱强度：Linux 实现提供强大的文件系统和网络隔离，但包括一个 `enableWeakerNestedSandbox` 模式，使其能够在 Docker 环境中工作而无需特权命名空间。此选项大大削弱了安全性，应仅在其他隔离被强制执行的情况下使用。

沙箱如何与权限相关

沙箱和 [permissions](#) 是互补的安全层，协同工作：

- 权限控制 Claude Code 可以使用哪些工具，在任何工具运行之前进行评估。它们适用于所有工具：Bash、Read、Edit、WebFetch、MCP 和其他工具。
- 沙箱提供操作系统级强制执行，限制 Bash 命令在文件系统和网络级别可以访问的内容。它仅适用于 Bash 命令及其子进程。

文件系统和网络限制通过沙箱设置和权限规则进行配置：

- 使用 `sandbox.filesystem.allowWrite` 向工作目录外的路径授予子进程写入访问权限
- 使用 `sandbox.filesystem.denyWrite` 和 `sandbox.filesystem.denyRead` 阻止子进程访问特定路径
- 使用 `Read` 和 `Edit` 拒绝规则阻止访问特定文件或目录
- 使用 `WebFetch` 允许/拒绝规则控制域名访问
- 使用沙箱 `allowedDomains` 控制 Bash 命令可以到达的域名

来自 `sandbox.filesystem` 设置和权限规则的路径被合并到最终沙箱配置中。

此 [repository](#) 包括常见部署场景的启动设置配置，包括沙箱特定的示例。使用这些作为起点，并根据您的需求调整它们。

高级用法

自定义代理配置

对于需要高级网络安全的组织，您可以实现自定义代理以：

- 解密和检查 HTTPS 流量
- 应用自定义过滤规则
- 记录所有网络请求
- 与现有安全基础设施集成

```
{
  "sandbox": {
    "network": {
      "httpProxyPort": 8080,
      "socksProxyPort": 8081
    }
  }
}
```

与现有安全工具的集成

沙箱 bash 工具与以下工具配合使用：

- **权限规则：**与 [permission settings](#) 结合以实现深度防御
- **开发容器：**与 [devcontainers](#) 一起使用以获得额外隔离
- **企业策略：**通过 [managed settings](#) 强制执行沙箱配置

最佳实践

1. **从限制性开始：**从最小权限开始，根据需要扩展
2. **监控日志：**查看沙箱违规尝试以了解 Claude Code 的需求
3. **使用环境特定的配置：**开发与生产环境的不同沙箱规则
4. **与权限结合：**将沙箱与 IAM 策略一起使用以实现全面安全
5. **测试配置：**验证您的沙箱设置不会阻止合法工作流程

开源

沙箱运行时作为开源 npm 包提供，供您在自己的代理项目中使用。这使更广泛的 AI 代理社区能够构建更安全、更安全的自主系统。这也可以用于沙箱化您可能希望运行的其他程序。例如，要沙箱化 MCP 服务器，您可以运行：

```
npx @anthropic-ai/sandbox-runtime <command-to-sandbox>
```

有关实现细节和源代码，请访问 [GitHub repository](#)。

限制

- **性能开销：**最小，但某些文件系统操作可能稍慢

- **兼容性：**某些需要特定系统访问模式的工具可能需要配置调整，或者甚至可能需要在沙箱外运行
- **平台支持：**支持 macOS、Linux 和 WSL2。不支持 WSL1。计划支持原生 Windows。

另请参阅

- [Security](#) - 全面的安全功能和最佳实践
- [Permissions](#) - 权限配置和访问控制
- [Settings](#) - 完整的配置参考
- [CLI reference](#) - 命令行选项

checkpointing

自动跟踪和回退 Claude 的编辑，快速恢复不需要的更改。

Claude Code 自动跟踪 Claude 在工作过程中的文件编辑，允许您快速撤销更改并回退到之前的状态，以防任何事情出现偏差。

checkpointing 如何工作

当您与 Claude 合作时，checkpointing 会自动捕获每次编辑前代码的状态。这个安全网让您可以放心地执行雄心勃勃的大规模任务，因为您始终可以返回到之前的代码状态。

自动跟踪

Claude Code 跟踪其文件编辑工具所做的所有更改：

- 每个用户提示都会创建一个新的 checkpoint
- Checkpoints 在会话之间持久存在，因此您可以在恢复的对话中访问它们
- 在 30 天后自动清理（可配置）

回退更改

按两次 **Esc** (**Esc** + **Esc**) 或使用 `/rewind` 命令打开回退菜单。您可以选择恢复：

- **仅对话**：回退到用户消息，同时保留代码更改
- **仅代码**：恢复文件更改，同时保留对话
- **代码和对话**：将两者都恢复到会话中的先前点

常见用例

Checkpoints 在以下情况下特别有用：

- **探索替代方案**：尝试不同的实现方法，而不会丢失起点
- **从错误中恢复**：快速撤销引入错误或破坏功能的更改
- **迭代功能**：进行变体实验，同时知道您可以恢复到工作状态

限制

Bash 命令更改未被跟踪

Checkpointing 不跟踪由 bash 命令修改的文件。例如，如果 Claude Code 运行：

```
rm file.txt  
mv old.txt new.txt  
cp source.txt dest.txt
```

这些文件修改无法通过回退撤销。只有通过 Claude 的文件编辑工具进行的直接文件编辑才会被跟踪。

外部更改未被跟踪

Checkpointing 仅跟踪在当前会话中已编辑的文件。您在 Claude Code 外部对文件所做的手动更改以及来自其他并发会话的编辑通常不会被捕获，除非它们碰巧修改了与当前会话相同的文件。

不是版本控制的替代品

Checkpoints 设计用于快速的会话级恢复。对于永久版本历史和协作：

- 继续使用版本控制（例如 Git）进行提交、分支和长期历史
- Checkpoints 补充但不替代适当的版本控制
- 将 checkpoints 视为”本地撤销”，将 Git 视为”永久历史”

另请参阅

- [Interactive mode](#) - 快捷键和会话控制
- [Built-in commands](#) - 使用 `/rewind` 访问 checkpoints
- [CLI reference](#) - 命令行选项

数据使用

了解 Anthropic 对 Claude 数据使用的政策

数据政策

数据训练政策

消费者用户（Free、Pro 和 Max 计划）： 我们给您选择是否允许您的数据用于改进未来的 Claude 模型。当此设置打开时，我们将使用来自 Free、Pro 和 Max 账户的数据来训练新模型（包括当您从这些账户使用 Claude Code 时）。

商业用户：（Team 和 Enterprise 计划、API、第三方平台和 Claude Gov）维持现有政策：除非客户选择向我们提供数据以改进模型（例如，[开发者合作伙伴计划](#)），否则 Anthropic 不会使用商业条款下发送到 Claude Code 的代码或提示来训练生成模型。

开发者合作伙伴计划

如果您明确选择加入通过[开发者合作伙伴计划](#)等方式向我们提供训练材料的方法，我们可能会使用这些提供的材料来训练我们的模型。组织管理员可以明确选择为其组织加入开发者合作伙伴计划。请注意，此计划仅适用于 Anthropic 第一方 API，不适用于 Bedrock 或 Vertex 用户。

使用 `/bug` 命令的反馈

如果您选择使用 `/bug` 命令向我们发送有关 Claude Code 的反馈，我们可能会使用您的反馈来改进我们的产品和服务。通过 `/bug` 共享的记录保留 5 年。

会话质量调查

当您在 Claude Code 中看到“Claude 在本次会话中表现如何？”“提示时，对此调查的回应（包括选择“关闭”）仅记录您的数字评分（1、2、3 或关闭）。作为此调查的一部分，我们不收集或存储任何对话记录、输入、输出或其他会话数据。与竖起大拇指/竖起大拇指向下反馈或 `/bug` 报告不同，此会话质量调查是一个简单的产品满意度指标。您对此调查的回应不会影响您的数据训练偏好，也不能用于训练我们的 AI 模型。

要禁用这些调查，请设置 `CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY=1`。当使用第三方提供商（Bedrock、Vertex、Foundry）或禁用遥测时，调查也会自动禁用。

数据保留

Anthropic 根据您的账户类型和偏好保留 Claude Code 数据。

消费者用户（Free、Pro 和 Max 计划）：

- 允许数据用于模型改进的用户：5 年保留期，以支持模型开发和安全改进
- 不允许数据用于模型改进的用户：30 天保留期
- 隐私设置可以随时在 claude.ai/settings/data-privacy-controls 更改。

商业用户（Team、Enterprise 和 API）：

- 标准：30 天保留期
- **零数据保留**：适用于 Claude for Enterprise 上的 Claude Code。ZDR 按组织启用；每个新组织必须由您的账户团队单独启用 ZDR
- 本地缓存：Claude Code 客户端可能会在本地存储会话长达 30 天，以启用会话恢复（可配置）

您可以随时删除网络上的单个 Claude Code 会话。删除会话会永久删除该会话的事件数据。有关如何删除会话的说明，请参阅[管理会话](#)。

在我们的[隐私中心](#)了解更多关于数据保留实践的信息。

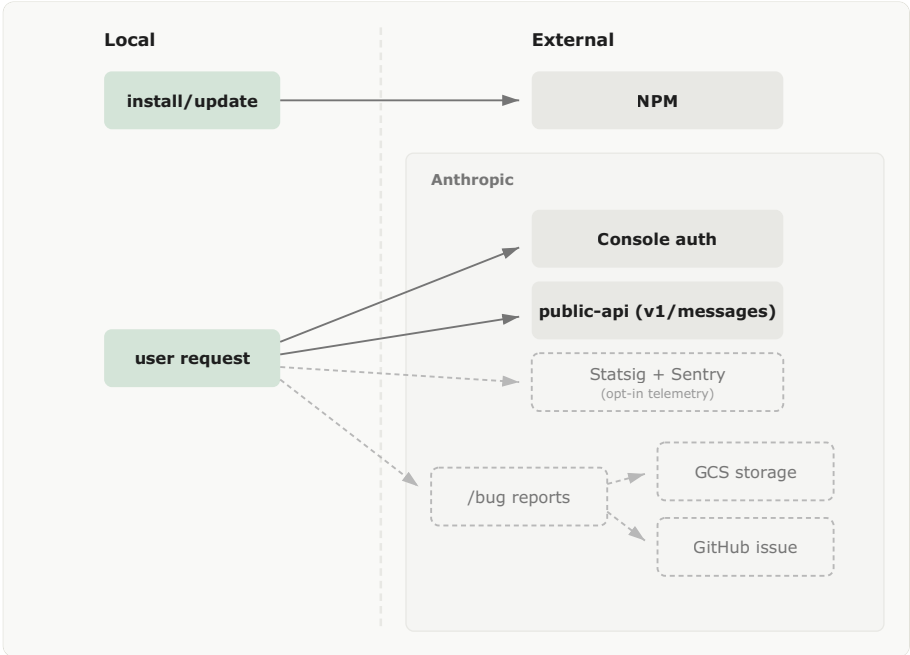
有关完整详情，请查看我们的[商业服务条款](#)（适用于 Team、Enterprise 和 API 用户）或[消费者条款](#)（适用于 Free、Pro 和 Max 用户）和[隐私政策](#)。

数据访问

对于所有第一方用户，您可以了解更多关于为[本地 Claude Code](#)和[远程 Claude Code](#)记录的数据。[远程控制](#)会话遵循本地数据流，因为所有执行都发生在您的机器上。请注意，对于远程 Claude Code，Claude 访问您启动 Claude Code 会话的存储库。Claude 不访问您已连接但未在其中启动会话的存储库。

本地 Claude Code：数据流和依赖关系

下面的图表显示了 Claude Code 在安装和正常操作期间如何连接到外部服务。实线表示必需的连接，而虚线表示可选或用户启动的数据流。



显示 Claude Code 外部连接的图表：安装/更新连接到 NPM，用户请求连接到 Anthropic 服务，包括 Console 身份验证、public-api，以及可选的 Statsig、Sentry 和错误报告

Claude Code 从 [NPM](#) 安装。Claude Code 在本地运行。为了与 LLM 交互，Claude Code 通过网络发送数据。此数据包括所有用户提示和模型输出。数据通过 TLS 在传输中加密，在静止时未加密。Claude Code 与大多数流行的 VPN 和 LLM 代理兼容。

Claude Code 基于 Anthropic 的 API 构建。有关我们 API 的安全控制的详情，包括我们的 API 日志记录程序，请参阅 [Anthropic 信任中心](#) 中提供的合规工件。

云执行：数据流和依赖关系

使用 [网络上的 Claude Code](#) 时，会话在 Anthropic 管理的虚拟机中运行，而不是在本地运行。在云环境中：

- **代码和数据存储：**您的存储库被克隆到隔离的 VM。代码和会话数据受您的账户类型的保留和使用政策约束（请参阅上面的数据保留部分）
- **凭证：**GitHub 身份验证通过安全代理处理；您的 GitHub 凭证永远不会进入沙箱
- **网络流量：**所有出站流量都通过安全代理进行审计日志记录和滥用防止
- **会话数据：**提示、代码更改和输出遵循与本地 Claude Code 使用相同的数据政策

有关云执行的安全详情，请参阅 [安全](#)。

遥测服务

Claude Code 从用户的机器连接到 Statsig 服务，以记录操作指标，例如延迟、可靠性和使用模式。此日志记录不包括任何代码或文件路径。数据使用 TLS 在传输中加密，使用 256 位 AES 加密在静止时加密。在 [Statsig 安全文档](#) 中了解更多。要选择退出 Statsig 遥测，请设置 `DISABLE_TELEMETRY` 环境变量。

Claude Code 从用户的机器连接到 Sentry 以进行操作错误日志记录。数据使用 TLS 在传输中加密，使用 256 位 AES 加密在静止时加密。在 [Sentry 安全文档](#) 中了解更多。要选择退出错误日志记录，请设置 `DISABLE_ERROR_REPORTING` 环境变量。

当用户运行 `/bug` 命令时，他们的完整对话历史记录（包括代码）的副本被发送到 Anthropic。数据在传输中和静止时加密。可选地，在我们的公共存储库中创建 Github 问题。要选择退出错误报告，请设置 `DISABLE_BUG_COMMAND` 环境变量。

按 API 提供商的默认行为

默认情况下，当使用 Bedrock、Vertex 或 Foundry 时，我们禁用所有非必需的流量（包括错误报告、遥测、错误报告功能和会话质量调查）。您也可以通过设置 `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` 环境变量一次选择退出所有这些。以下是完整的默认行为：

服务	Claude API	Vertex API	Bedrock API	Foundry API
Statsig (指标)	默认开启。 <code>DISABLE_TELEMETRY=1</code> 禁用。	默认关闭。 <code>CLAUDE_CODE_USE_VERTEX</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_BEDROCK</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_FOUNDRY</code> 必须为 1。
Sentry (错误)	默认开启。 <code>DISABLE_ERROR_REPORTING=1</code> 禁用。	默认关闭。 <code>CLAUDE_CODE_USE_VERTEX</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_BEDROCK</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_FOUNDRY</code> 必须为 1。
Claude API (<code>/bug</code> 报告)	默认开启。 <code>DISABLE_BUG_COMMAND=1</code> 禁用。	默认关闭。 <code>CLAUDE_CODE_USE_VERTEX</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_BEDROCK</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_FOUNDRY</code> 必须为 1。
会话质量调查	默认开启。 <code>CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY=1</code> 禁用。	默认关闭。 <code>CLAUDE_CODE_USE_VERTEX</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_BEDROCK</code> 必须为 1。	默认关闭。 <code>CLAUDE_CODE_USE_FOUNDRY</code> 必须为 1。

所有环境变量都可以检查到 `settings.json` ([了解更多](#)) 。

法律和合规

Claude Code 的法律协议、合规认证和安全信息。

法律协议

许可证

您对 Claude Code 的使用受以下条款约束：

- [商业条款](#) - 适用于 Team、Enterprise 和 Claude API 用户
- [消费者条款](#) - 适用于 Free、Pro 和 Max 用户

商业协议

无论您是直接使用 Claude API（1P）还是通过 AWS Bedrock 或 Google Vertex（3P）访问它，您现有的商业协议将适用于 Claude Code 使用，除非我们相互另有约定。

合规

医疗保健合规（BAA）

如果客户与我们签订了业务关联协议（BAA），并想使用 Claude Code，如果客户已执行 BAA 并激活了零数据保留（ZDR），BAA 将自动扩展以覆盖 Claude Code。BAA 将适用于通过 Claude Code 流动的该客户的 API 流量。

安全和信任

信任和安全

您可以在 [Anthropic 信任中心](#) 和 [透明度中心](#) 中找到更多信息。

安全漏洞报告

Anthropic 通过 HackerOne 管理我们的安全计划。 [使用此表单报告漏洞](#)。

© Anthropic PBC。版权所有。使用受适用的 Anthropic 服务条款约束。

零数据保留

了解 Claude for Enterprise 上 Claude Code 的零数据保留 (ZDR)，包括范围、禁用功能以及如何请求启用。

零数据保留 (ZDR) 在通过 Claude for Enterprise 使用 Claude Code 时可用。启用 ZDR 后，Claude Code 会话期间生成的提示和模型响应会实时处理，在返回响应后不会由 Anthropic 存储，除非需要遵守法律或防止滥用。

Claude for Enterprise 上的 ZDR 为企业客户提供了使用 Claude Code 并实现零数据保留的能力，同时可以访问管理功能：

- 按用户的成本控制
- [分析](#)仪表板
- [服务器管理的设置](#)
- 审计日志

Claude for Enterprise 上 Claude Code 的 ZDR 仅适用于 Anthropic 的直接平台。对于在 AWS Bedrock、Google Vertex AI 或 Microsoft Foundry 上的 Claude 部署，请参考这些平台的数据保留政策。

ZDR 范围

ZDR 涵盖 Claude for Enterprise 上的 Claude Code 推理。

Warning:

ZDR 在每个组织的基础上启用。每个新组织都需要由您的 Anthropic 账户团队单独启用 ZDR。ZDR 不会自动应用于在同一账户下创建的新组织。请联系您的账户团队为任何新组织启用 ZDR。

ZDR 涵盖的内容

ZDR 涵盖通过 Claude for Enterprise 上的 Claude Code 进行的模型推理调用。当您在终端中使用 Claude Code 时，您发送的提示和 Claude 生成的响应不会由 Anthropic 保留。这适用于无论使用哪个 Claude 模型。

ZDR 不涵盖的内容

ZDR 不适用于以下内容，即使对于启用了 ZDR 的组织也是如此。这些功能遵循[标准数据保留政策](#)：

功能	详情
claude.ai 上的聊天	通过 Claude for Enterprise 网络界面的聊天对话不受 ZDR 保护。
Cowork	Cowork 会话不受 ZDR 保护。
Claude Code 分析	不存储提示或模型响应，但收集生产力元数据，如账户电子邮件和使用统计。对于 ZDR 组织，贡献指标不可用； 分析仪表板 仅显示使用指标。
用户和席位管理	管理数据，如账户电子邮件和席位分配，根据标准政策保留。
第三方集成	由第三方工具、MCP servers 或其他外部集成处理的数据不受 ZDR 保护。请独立审查这些服务的数据处理实践。

ZDR 下禁用的功能

当为 Claude for Enterprise 上的 Claude Code 组织启用 ZDR 时，某些需要存储提示或完成的功能会在后端级别自动禁用：

功能	原因
网络上的 Claude Code	需要服务器端存储对话历史。
来自 Desktop 应用的 远程会话	需要包含提示和完成的持久会话数据。
反馈提交 (/feedback)	提交反馈会将对话数据发送给 Anthropic。

这些功能在后端被阻止，无论客户端显示如何。如果您在启动期间在 Claude Code 终端中看到禁用的功能，尝试使用它会返回一个错误，指示组织的政策不允许该操作。

如果未来的功能需要存储提示或完成，它们也可能被禁用。

政策违规的数据保留

即使启用了 ZDR，Anthropic 也可能在法律要求或解决使用政策违规时保留数据。如果会话因政策违规而被标记，Anthropic 可能会保留相关的输入和输出长达 2 年，与 Anthropic 的标准 ZDR 政策一致。

请求 ZDR

要为 Claude for Enterprise 上的 Claude Code 请求 ZDR，请联系您的 Anthropic 账户团队。您的账户团队将在内部提交请求，Anthropic 将在确认符合条件后在您的组织上审查并启用 ZDR。所有启用操作都会被审计记录。

如果您当前通过按使用量付费的 API 密钥使用 Claude Code 的 ZDR，您可以过渡到 Claude for Enterprise 以获得对管理功能的访问权限，同时为 Claude Code 保持 ZDR。请联系您的账户团队以协调迁移。

Part 10: Enterprise & Monitoring

使用分析跟踪团队使用情况

在分析仪表板中查看 Claude Code 使用指标、跟踪采用情况并衡量工程速度。

Claude Code 提供分析仪表板，帮助组织了解开发者使用模式、跟踪贡献指标，并衡量 Claude Code 对工程速度的影响。访问您计划的仪表板：

计划	仪表板 URL	包含内容	了解更多
Claude for Teams / Enterprise	claude.ai/analytics/claude-code	使用指标、带 GitHub 集成的贡献指标、排行榜、数据导出	详情
API (Claude Console)	platform.claude.com/claude-code	使用指标、支出跟踪、团队洞察	详情

访问 Teams 和 Enterprise 分析

导航到 claude.ai/analytics/claude-code。管理员和所有者可以查看仪表板。

Teams 和 Enterprise 仪表板包括：

- **使用指标：**接受的代码行数、建议接受率、日活跃用户和会话数
- **贡献指标：**使用 Claude Code 协助的 PR 和已发布的代码行数，带有 [GitHub 集成](#)
- **排行榜：**按 Claude Code 使用情况排名的顶级贡献者
- **数据导出：**将贡献数据下载为 CSV 格式以进行自定义报告

启用贡献指标

Note:

贡献指标处于公开测试版，可用于 Claude for Teams 和 Claude for Enterprise 计划。这些指标仅涵盖您 claude.ai 组织内的用户。通过 Claude Console API 或第三方集成的使用不包括在内。

使用和采用数据可用于所有 Claude for Teams 和 Claude for Enterprise 账户。贡献指标需要额外设置来连接您的 GitHub 组织。

您需要所有者角色来配置分析设置。GitHub 管理员必须安装 GitHub 应用。

Warning:

启用了 [Zero Data Retention](#) 的组织无法使用贡献指标。分析仪表板将仅显示使用指标。

Step 1: 安装 GitHub 应用

GitHub 管理员在您组织的 GitHub 账户上安装 Claude GitHub 应用，地址为 github.com/apps/claude。

Step 2: 启用 Claude Code 分析

Claude 所有者导航到 claude.ai/admin-settings/claude-code 并启用 Claude Code 分析功能。

Step 3: 启用 GitHub 分析

在同一页面上，启用”GitHub 分析”切换。

Step 4: 使用 GitHub 进行身份验证

完成 GitHub 身份验证流程并选择要包含在分析中的 GitHub 组织。

启用后，数据通常在 24 小时内出现，并进行每日更新。如果没有数据出现，您可能会看到以下消息之一：

- **“GitHub 应用必需”**：安装 GitHub 应用以查看贡献指标
- **“数据处理进行中”**：几天后重新检查，如果数据未出现，请确认 GitHub 应用已安装

贡献指标支持 GitHub Cloud 和 GitHub Enterprise Server。

查看摘要指标

Note:

这些指标故意保守，代表对 Claude Code 实际影响的低估。仅计算有高度信心涉及 Claude Code 的代码行和 PR。

仪表板在顶部显示这些摘要指标：

- **带 CC 的 PR**：包含至少一行使用 Claude Code 编写的代码的已合并拉取请求的总计数
- **带 CC 的代码行**：所有已合并 PR 中使用 Claude Code 协助编写的代码行总数。仅计算”有效行”：规范化后超过 3 个字符的行，不包括空行和仅包含括号或琐碎标点符号的行。

- **带 Claude Code 的 PR (%)**：包含 Claude Code 协助代码的所有已合并 PR 的百分比
- **建议接受率**：用户接受 Claude Code 代码编辑建议的次数百分比，包括 Edit、Write 和 NotebookEdit 工具使用
- **接受的代码行**：Claude Code 编写且用户在其会话中接受的代码行总数。这包括被拒绝的建议，也不跟踪后续删除。

探索图表

仪表板包括多个图表来可视化一段时间内的趋势。

跟踪采用

采用图表显示每日使用趋势：

- **用户**：日活跃用户
- **会话**：每天的活跃 Claude Code 会话数

衡量每个用户的 PR

此图表显示一段时间内的个人开发者活动：

- **每个用户的 PR**：每天合并的 PR 总数除以日活跃用户
- **用户**：日活跃用户

使用此功能了解随着 Claude Code 采用增加，个人生产力如何变化。

查看拉取请求分解

拉取请求图表显示已合并 PR 的每日分解：

- **带 CC 的 PR**：包含 Claude Code 协助代码的拉取请求
- **不带 CC 的 PR**：不包含 Claude Code 协助代码的拉取请求

切换到**代码行**视图以按代码行而不是 PR 计数查看相同的分解。

查找顶级贡献者

排行榜显示按贡献量排名的前 10 个用户。在以下之间切换：

- **拉取请求**：显示每个用户的带 Claude Code 的 PR 与所有 PR
- **代码行**：显示每个用户的带 Claude Code 的行与所有行

单击**导出所有用户**以将所有用户的完整贡献数据下载为 CSV 文件。导出包括所有用户，而不仅仅是显示的前 10 个。

PR 归属

启用贡献指标后，Claude Code 会分析已合并的拉取请求，以确定哪些代码是使用 Claude Code 协助编写的。这是通过将 Claude Code 会话活动与每个 PR 中的代码进行匹配来完成的。

标记标准

如果 PR 包含在 Claude Code 会话期间编写的至少一行代码，则将其标记为”带 Claude Code”。系统使用保守匹配：仅计算有高度信心涉及 Claude Code 的代码。

归属过程

当拉取请求被合并时：

1. 从 PR diff 中提取添加的行
2. 识别在时间窗口内编辑匹配文件的 Claude Code 会话
3. 使用多种策略将 PR 行与 Claude Code 输出进行匹配
4. 计算 AI 协助行和总行的指标

在比较之前，行被规范化：空格被修剪、多个空格被折叠、引号被标准化、文本被转换为小写。

包含 Claude Code 协助行的已合并拉取请求在 GitHub 中被标记为 `claude-code-assisted`。

时间窗口

PR 合并日期前 21 天到后 2 天的会话被考虑用于归属匹配。

排除的文件

某些文件会自动从分析中排除，因为它们是自动生成的：

- 锁定文件：package-lock.json、yarn.lock、Cargo.lock 等
- 生成的代码：Protobuf 输出、构建工件、缩小的文件
- 构建目录：dist/、build/、node_modules/、target/
- 测试夹具：快照、磁带、模拟数据
- 超过 1,000 个字符的行，可能是缩小或生成的

归属说明

在解释归属数据时，请记住这些额外的细节：

- 由开发者大幅重写的代码（差异超过 20%）不归属于 Claude Code
- 不考虑 21 天窗口外的会话

- 该算法在执行归属时不考虑 PR 源或目标分支

从分析中获得最大收益

使用贡献指标来展示 ROI、识别采用模式，并找到可以帮助他人入门的团队成员。

监控采用

跟踪采用图表和用户计数以识别：

- 可以分享最佳实践的活跃用户
- 整个组织的整体采用趋势
- 可能表示摩擦或问题的使用下降

衡量 ROI

贡献指标帮助回答”这个工具值得投资吗？“，使用来自您自己代码库的数据：

- 随着采用增加，跟踪一段时间内每个用户的 PR 变化
- 比较使用和不使用 Claude Code 发布的 PR 和代码行
- 与 [DORA 指标](#)、冲刺速度或其他工程 KPI 一起使用，以了解采用 Claude Code 的变化

识别超级用户

排行榜帮助您找到具有高 Claude Code 采用率的团队成员，他们可以：

- 与团队分享提示技术和工作流
- 提供关于什么运行良好的反馈
- 帮助新用户入门

以编程方式访问数据

要通过 GitHub 查询此数据，请搜索标记为 `claude-code-assisted` 的 PR。

访问 API 客户的分析

使用 Claude Console 的 API 客户可以在 platform.claude.com/claude-code 访问分析。您需要 UsageView 权限来访问仪表板，该权限授予开发者、计费、管理员、所有者和主要所有者角色。

Note:

贡献指标与 GitHub 集成目前不可用于 API 客户。Console 仪表板仅显示使用和支出指标。

Console 仪表板显示：

- **接受的代码行：** Claude Code 编写且用户在其会话中接受的代码行总数。这包括被拒绝的建议，也不跟踪后续删除。
- **建议接受率：** 用户接受代码编辑工具使用的次数百分比，包括 Edit、Write 和 NotebookEdit 工具。
- **活动：** 图表上显示的日活跃用户和会话。
- **支出：** 每日 API 成本（美元）与用户计数一起显示。

查看团队洞察

团队洞察表显示每个用户的指标：

- **成员：** 所有已向 Claude Code 进行身份验证的用户。API 密钥用户按密钥标识符显示，OAuth 用户按电子邮件地址显示。
- **本月支出：** 每个用户当前月份的 API 成本总计。
- **本月代码行：** 每个用户当前月份接受的代码行总数。

Note:

Console 仪表板中的支出数字是用于分析目的的估计值。有关实际成本，请参阅您的计费页面。

相关资源

- [使用 OpenTelemetry 进行监控](#)：将实时指标和事件导出到您的可观测性堆栈
- [有效管理成本](#)：设置支出限制并优化令牌使用
- [权限](#)：配置角色和权限

监控

了解如何为 Claude Code 启用和配置 OpenTelemetry。

Claude Code 支持 OpenTelemetry (OTel) 指标和事件，用于监控和可观测性。

所有指标都是通过 OpenTelemetry 的标准指标协议导出的时间序列数据，事件通过 OpenTelemetry 的日志/事件协议导出。用户有责任确保其指标和日志后端配置正确，并且聚合粒度满足其监控要求。

快速开始

使用环境变量配置 OpenTelemetry：

```
## 1. 启用遥测
export CLAUDE_CODE_ENABLE_TELEMETRY=1

## 2. 选择导出器（两者都是可选的 - 仅配置您需要的）
export OTEL_METRICS_EXPORTER=otlp      # 选项: otlp、prometheus、console
export OTEL_LOGS_EXPORTER=otlp        # 选项: otlp、console

## 3. 配置 OTLP 端点（用于 OTLP 导出器）
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317

## 4. 设置身份验证（如果需要）
export OTEL_EXPORTER_OTLP_HEADERS="Authorization=Bearer your-token"

## 5. 用于调试：减少导出间隔
export OTEL_METRIC_EXPORT_INTERVAL=10000 # 10 秒（默认：60000ms）
export OTEL_LOGS_EXPORT_INTERVAL=5000    # 5 秒（默认：5000ms）

## 6. 运行 Claude Code
claude
```

Note:

默认导出间隔为指标 60 秒和日志 5 秒。在设置期间，您可能希望使用更短的间隔用于调试目的。请记住为生产使用重置这些值。

有关完整配置选项，请参阅 [OpenTelemetry 规范](#)。

管理员配置

管理员可以通过 [托管设置文件](#) 为所有用户配置 OpenTelemetry 设置。这允许在整个组织中集中控制遥测设置。有关设置如何应用的更多信息，请参阅 [设置优先级](#)。

示例托管设置配置：

```
{
  "env": {
    "CLAUDE_CODE_ENABLE_TELEMETRY": "1",
    "OTEL_METRICS_EXPORTER": "otlp",
    "OTEL_LOGS_EXPORTER": "otlp",
    "OTEL_EXPORTER_OTLP_PROTOCOL": "grpc",
    "OTEL_EXPORTER_OTLP_ENDPOINT": "http://collector.company.com:4317",
    "OTEL_EXPORTER_OTLP_HEADERS": "Authorization=Bearer company-token"
  }
}
```

Note:

托管设置可以通过 MDM（移动设备管理）或其他设备管理解决方案分发。在托管设置文件中定义的环境变量具有高优先级，用户无法覆盖。

配置详情

常见配置变量

环境变量	描述	示例值
CLAUDE_CODE_ENABLE_TELEMETRY	启用遥测收集（必需）	1
OTEL_METRICS_EXPORTER	指标导出器类型（逗号分隔）	console 、 otlp 、 prometheus

环境变量	描述	示例值
OTEL_LOGS_EXPORTER	日志/事件导出器类型（逗号分隔）	console 、 otlp
OTEL_EXPORTER_OTLP_PROTOCOL	OTLP 导出器的协议（所有信号）	grpc 、 http/json 、 http/protobuf
OTEL_EXPORTER_OTLP_ENDPOINT	OTLP 收集器端点（所有信号）	http://localhost:4317
OTEL_EXPORTER_OTLP_METRICS_PROTOCOL	指标协议（覆盖常规）	grpc 、 http/json 、 http/protobuf
OTEL_EXPORTER_OTLP_METRICS_ENDPOINT	OTLP 指标端点（覆盖常规）	http://localhost:4318/v1/metrics
OTEL_EXPORTER_OTLP_LOGS_PROTOCOL	日志协议（覆盖常规）	grpc 、 http/json 、 http/protobuf
OTEL_EXPORTER_OTLP_LOGS_ENDPOINT	OTLP 日志端点（覆盖常规）	http://localhost:4318/v1/logs
OTEL_EXPORTER_OTLP_HEADERS	OTLP 的身份验证标头	Authorization=Bearer token
OTEL_EXPORTER_OTLP_METRICS_CLIENT_KEY	mTLS 身份验证的客户端密钥	客户端密钥文件的路径
OTEL_EXPORTER_OTLP_METRICS_CLIENT_CERTIFICATE	mTLS 身份验证的客户端证书	客户端证书文件的路径
OTEL_METRIC_EXPORT_INTERVAL	导出间隔（毫秒）（默认：60000）	5000 、 60000
OTEL_LOGS_EXPORT_INTERVAL	日志导出间隔（毫秒）（默认：5000）	1000 、 10000
OTEL_LOG_USER_PROMPTS	启用户提示内容的日志记录（默认：禁用）	1 启用

环境变量	描述	示例值
CLAUDE_CODE_OTEL_HEADERS_HELPER_DEBOUNCE_MS	刷新动态标头的间隔（默认：1740000ms / 29 分钟）	900000

指标基数控制

以下环境变量控制指标中包含哪些属性以管理基数：

环境变量	描述	默认值	禁用示例
OTEL_METRICS_INCLUDE_SESSION_ID	在指标中包含 session.id 属性	true	false
OTEL_METRICS_INCLUDE_APP_VERSION	在指标中包含 app.version 属性	false	true
OTEL_METRICS_INCLUDE_USER_ACCOUNT_UUID	在指标中包含 user.account_uuid 属性	true	false

这些变量有助于控制指标的基数，这会影响指标后端中的存储要求和查询性能。较低的基数通常意味着更好的性能和更低的存储成本，但分析的数据粒度较低。

动态标头

对于需要动态身份验证的企业环境，您可以配置脚本来动态生成标头：

设置配置

添加到您的 `.claude/settings.json`：

```
{
  "otelHeadersHelper": "/bin/generate_opentelemetry_headers.sh"
}
```


脚本要求

脚本必须输出有效的 JSON，其中包含表示 HTTP 标头的字符串键值对：

```
#!/bin/bash
## 示例：多个标头
echo '{"Authorization": \"Bearer $(get-token.sh)\", \"X-API-Key\": \"$(get-api-key.sh)\"}'
```

刷新行为

标头助手脚本在启动时运行，之后定期运行以支持令牌刷新。默认情况下，脚本每 29 分钟运行一次。使用 `CLAUDE_CODE_OTEL_HEADERS_HELPER_DEBOUNCE_MS` 环境变量自定义间隔。

多团队组织支持

具有多个团队或部门的组织可以使用 `OTEL_RESOURCE_ATTRIBUTES` 环境变量添加自定义属性以区分不同的组：

```
## 添加自定义属性用于团队识别
export OTEL_RESOURCE_ATTRIBUTES="department=engineering,team.id=platform,cost_center=eng-123"
```

这些自定义属性将包含在所有指标和事件中，允许您：

- 按团队或部门过滤指标
- 按成本中心跟踪成本
- 创建特定于团队的仪表板
- 为特定团队设置警报

Warning:

OTEL_RESOURCE_ATTRIBUTES 的重要格式要求：

`OTEL_RESOURCE_ATTRIBUTES` 环境变量遵循 [W3C Baggage 规范](#)，具有严格的格式要求：

- **不允许空格**：值不能包含空格。例如，`user.organizationName=My Company` 无效
- **格式**：必须是逗号分隔的键=值对：`key1=value1,key2=value2`
- **允许的字符**：仅 US-ASCII 字符，不包括控制字符、空格、双引号、逗号、分号和反斜杠
- **特殊字符**：超出允许范围的字符必须进行百分比编码

示例：

❌ 无效 - 包含空格

```
export OTEL_RESOURCE_ATTRIBUTES="org.name=John's Organization"
```

✅ 有效 - 改用下划线或驼峰式大小写

```
export OTEL_RESOURCE_ATTRIBUTES="org.name=Johns_Organization"
```

```
export OTEL_RESOURCE_ATTRIBUTES="org.name=JohnsOrganization"
```

✅ 有效 - 如果需要，对特殊字符进行百分比编码

```
export OTEL_RESOURCE_ATTRIBUTES="org.name=John%27s%20Organization"
```

注意：用引号包装值不会转义空格。例如，`org.name="My Company"` 会导致字面值 `"My Company"`（包括引号），而不是 `My Company`。

示例配置

控制台调试 (1 秒间隔)

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=console
export OTEL_METRIC_EXPORT_INTERVAL=1000
```

OTLP/gRPC

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
```

Prometheus

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=prometheus
```

多个导出器

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=console,otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=http/json
```

指标和日志的不同端点/后端

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=otlp
export OTEL_LOGS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_METRICS_PROTOCOL=http/protobuf
export OTEL_EXPORTER_OTLP_METRICS_ENDPOINT=http://metrics.company.com:4318
export OTEL_EXPORTER_OTLP_LOGS_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_LOGS_ENDPOINT=http://logs.company.com:4317
```

仅指标 (无事件/日志)

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
```

仅事件/日志 (无指标)

```
export CLAUDE_CODE_ENABLE_TELEMETRY=1
```

```
export OTEL_LOGS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
```

可用的指标和事件

标准属性

所有指标和事件共享这些标准属性：

属性	描述	控制方式
session.id	唯一的会话标识符	OTEL_METRICS_INCLUDE_SESSION_ID（默认：true）
app.version	当前 Claude Code 版本	OTEL_METRICS_INCLUDE_VERSION（默认：false）
organization.id	组织 UUID（已认证时）	可用时始终包含
user.account_uuid	账户 UUID（已认证时）	OTEL_METRICS_INCLUDE_ACCOUNT_UUID（默认：true）
terminal.type	终端类型（例如，iTerm.app、vscode、cursor、tmux）	检测到时始终包含

指标

Claude Code 导出以下指标：

指标名称	描述	单位
claude_code.session.count	启动的 CLI 会话计数	count
claude_code.lines_of_code.count	修改的代码行数计数	count
claude_code.pull_request.count	创建的拉取请求数	count
claude_code.commit.count	创建的 git 提交数	count

指标名称	描述	单位
<code>claude_code.cost.usage</code>	Claude Code 会话的成本	USD
<code>claude_code.token.usage</code>	使用的令牌数	tokens
<code>claude_code.code_edit_tool.decision</code>	代码编辑工具权限决策计数	count
<code>claude_code.active_time.total</code>	总活跃时间（秒）	s

指标详情

会话计数器

在每个会话开始时递增。

属性：

- 所有 [标准属性](#)

代码行计数器

当添加或删除代码时递增。

属性：

- 所有 [标准属性](#)
- `type`：（`"added"`、`"removed"`）

拉取请求计数器

通过 Claude Code 创建拉取请求时递增。

属性：

- 所有 [标准属性](#)

提交计数器

通过 Claude Code 创建 git 提交时递增。

属性：

- 所有 [标准属性](#)

成本计数器

在每个 API 请求后递增。

属性：

- 所有 [标准属性](#)
- `model`：模型标识符（例如，“claude-sonnet-4-5-20250929”）

令牌计数器

在每个 API 请求后递增。

属性：

- 所有 [标准属性](#)
- `type`：（`"input"`、`"output"`、`"cacheRead"`、`"cacheCreation"`）
- `model`：模型标识符（例如，“claude-sonnet-4-5-20250929”）

代码编辑工具决策计数器

当用户接受或拒绝 Edit、Write 或 NotebookEdit 工具使用时递增。

属性：

- 所有 [标准属性](#)
- `tool`：工具名称（`"Edit"`、`"Write"`、`"NotebookEdit"`）
- `decision`：用户决策（`"accept"`、`"reject"`）
- `language`：编辑文件的编程语言（例如，`"TypeScript"`、`"Python"`、`"JavaScript"`、`"Markdown"`）。对于无法识别的文件扩展名，返回 `"unknown"`。

活跃时间计数器

跟踪实际花费在积极使用 Claude Code 上的时间（不是空闲时间）。此指标在用户交互期间递增，例如输入提示或接收响应。

属性：

- 所有 [标准属性](#)

事件

Claude Code 通过 OpenTelemetry 日志/事件导出以下事件（当配置了 `OTEL_LOGS_EXPORTER` 时）：

用户提示事件

当用户提交提示时记录。

事件名称：`claude_code.user_prompt`

属性：

- 所有 [标准属性](#)
- `event.name`：`"user_prompt"`
- `event.timestamp`：ISO 8601 时间戳
- `prompt_length`：提示的长度
- `prompt`：提示内容（默认为已编辑，使用 `OTEL_LOG_USER_PROMPTS=1` 启用）

工具结果事件

当工具完成执行时记录。

事件名称：`claude_code.tool_result`

属性：

- 所有 [标准属性](#)
- `event.name`：`"tool_result"`
- `event.timestamp`：ISO 8601 时间戳
- `tool_name`：工具的名称
- `success`：`"true"` 或 `"false"`
- `duration_ms`：执行时间（毫秒）
- `error`：错误消息（如果失败）
- `decision`：`"accept"` 或 `"reject"`
- `source`：决策来源 -
`"config"`、`"user_permanent"`、`"user_temporary"`、`"user_abort"` 或 `"user_reject"`
- `tool_parameters`：包含工具特定参数的 JSON 字符串（如果可用）
 - 对于 Bash 工具：包括 `bash_command`、`full_command`、`timeout`、`description`、`sandbox`

API 请求事件

为每个对 Claude 的 API 请求记录。

事件名称：`claude_code.api_request`

属性：

- 所有 [标准属性](#)
- `event.name` : "api_request"
- `event.timestamp` : ISO 8601 时间戳
- `model` : 使用的模型（例如，“claude-sonnet-4-5-20250929”）
- `cost_usd` : USD 估计成本
- `duration_ms` : 请求持续时间（毫秒）
- `input_tokens` : 输入令牌数
- `output_tokens` : 输出令牌数
- `cache_read_tokens` : 从缓存读取的令牌数
- `cache_creation_tokens` : 用于缓存创建的令牌数

API 错误事件

当对 Claude 的 API 请求失败时记录。

事件名称：`claude_code.api_error`

属性：

- 所有 [标准属性](#)
- `event.name` : "api_error"
- `event.timestamp` : ISO 8601 时间戳
- `model` : 使用的模型（例如，“claude-sonnet-4-5-20250929”）
- `error` : 错误消息
- `status_code` : HTTP 状态代码（如果适用）
- `duration_ms` : 请求持续时间（毫秒）
- `attempt` : 尝试次数（对于重试的请求）

工具决策事件

当做出工具权限决策（接受/拒绝）时记录。

事件名称：`claude_code.tool_decision`

属性：

- 所有 [标准属性](#)
- `event.name` : "tool_decision"
- `event.timestamp` : ISO 8601 时间戳

- `tool_name`：工具的名称（例如，“Read”、“Edit”、“Write”、“NotebookEdit”）
- `decision`：“accept” 或 “reject”
- `source`：决策来源 -
“config”、“user_permanent”、“user_temporary”、“user_abort” 或 “user_reject”

解释指标和事件数据

Claude Code 导出的指标提供了对使用模式和生产力的宝贵见解。以下是您可以创建的一些常见可视化和分析：

使用情况监控

指标	分析机会
<code>claude_code.token.usage</code>	按 <code>type</code> （输入/输出）、用户、团队或模型分解
<code>claude_code.session.count</code>	跟踪随时间推移的采用和参与度
<code>claude_code.lines_of_code.count</code>	通过跟踪代码添加/删除来衡量生产力
<code>claude_code.commit.count</code> & <code>claude_code.pull_request.count</code>	了解对开发工作流的影响

成本监控

`claude_code.cost.usage` 指标有助于：

- 跟踪团队或个人的使用趋势
- 识别高使用会话以进行优化

Note:

成本指标是近似值。有关官方计费数据，请参阅您的 API 提供商（Claude 控制台、AWS Bedrock 或 Google Cloud Vertex）。

警报和分段

要考虑的常见警报：

- 成本激增
- 异常的令牌消耗
- 来自特定用户的高会话量

所有指标都可以按 `user.account_uuid` 、 `organization.id` 、 `session.id` 、 `model` 和 `app.version` 进行分段。

事件分析

事件数据提供了对 Claude Code 交互的详细见解：

工具使用模式： 分析工具结果事件以识别：

- 最常用的工具
- 工具成功率
- 平均工具执行时间
- 按工具类型的错误模式

性能监控： 跟踪 API 请求持续时间和工具执行时间以识别性能瓶颈。

后端考虑事项

您选择的指标和日志后端决定了您可以执行的分析类型：

对于指标

- **时间序列数据库（例如，Prometheus）：** 速率计算、聚合指标
- **列式存储（例如，ClickHouse）：** 复杂查询、唯一用户分析
- **全功能可观测性平台（例如，Honeycomb、Datadog）：** 高级查询、可视化、警报

对于事件/日志

- **日志聚合系统（例如，Elasticsearch、Loki）：** 全文搜索、日志分析
- **列式存储（例如，ClickHouse）：** 结构化事件分析
- **全功能可观测性平台（例如，Honeycomb、Datadog）：** 指标和事件之间的关联

对于需要日活跃用户/周活跃用户/月活跃用户 (DAU/WAU/MAU) 指标的组织，请考虑支持高效唯一值查询的后端。

服务信息

所有指标和事件都使用以下资源属性导出：

- `service.name` : `claude-code`
- `service.version` : 当前 Claude Code 版本
- `os.type` : 操作系统类型（例如，`linux`、`darwin`、`windows`）
- `os.version` : 操作系统版本字符串
- `host.arch` : 主机架构（例如，`amd64`、`arm64`）
- `wsl.version` : WSL 版本号（仅在 Windows Subsystem for Linux 上运行时出现）
- 仪表名称: `com.anthropic.claude_code`

ROI 测量资源

有关测量 Claude Code 投资回报率的综合指南，包括遥测设置、成本分析、生产力指标和自动化报告，请参阅 [Claude Code ROI 测量指南](#)。此存储库提供了现成的 Docker Compose 配置、Prometheus 和 OpenTelemetry 设置，以及用于生成与 Linear 等工具集成的生产力报告的模板。

安全/隐私考虑事项

- 遥测是可选的，需要显式配置
- 敏感信息（如 API 密钥或文件内容）永远不会包含在指标或事件中
- 用户提示内容默认为已编辑 - 仅记录提示长度。要启用用户提示日志记录，请设置 `OTEL_LOG_USER_PROMPTS=1`

在 Amazon Bedrock 上监控 Claude Code

有关 Amazon Bedrock 的 Claude Code 使用情况监控指南的详细信息，请参阅 [Claude Code 监控实现 \(Bedrock\)](#)。

企业部署概述

了解 Claude Code 如何与各种第三方服务和基础设施集成，以满足企业部署需求。

本页面提供了可用部署选项的概述，并帮助您为组织选择正确的配置。

提供商对比

功能 Anthropic Amazon Bedrock Google Vertex AI Microsoft Foundry

区域 支持的[国家](#) 多个 AWS [区域](#) 多个 GCP [区域](#) 多个 Azure [区域](#)

提示缓存 默认启用 默认启用 默认启用 默认启用

身份验证 API 密钥 API 密钥或 AWS 凭证 GCP 凭证 API 密钥或 Microsoft Entra ID

成本跟踪 仪表板 AWS Cost Explorer GCP Billing Azure Cost Management

企业功能 团队、使用情况监控 IAM 策略、CloudTrail IAM 角色、Cloud Audit Logs RBAC 策略、Azure Monitor

云提供商

- [Amazon Bedrock](#) 通过 AWS 基础设施使用 Claude 模型，支持 API 密钥或基于 IAM 的身份验证以及 AWS 原生监控
- [Google Vertex AI](#) 通过 Google Cloud Platform 访问 Claude 模型，具有企业级安全性和合规性
- [Microsoft Foundry](#) 通过 Azure 访问 Claude，支持 API 密钥或 Microsoft Entra ID 身份验证以及 Azure 计费

企业基础设施

- [企业网络](#) 配置 Claude Code 以与您组织的代理服务器和 SSL/TLS 需求配合使用
- [LLM 网关](#) 部署集中式模型访问，具有使用情况跟踪、预算编制和审计日志

配置概述

Claude Code 支持灵活的配置选项，允许您组合不同的提供商和基础设施：

Note:

了解以下内容的区别：

- **企业代理**：用于路由流量的 HTTP/HTTPS 代理（通过 `HTTPS_PROXY` 或 `HTTP_PROXY` 设置）
- **LLM 网关**：处理身份验证并提供提供商兼容端点的服务（通过 `ANTHROPIC_BASE_URL` 、
`ANTHROPIC_BEDROCK_BASE_URL` 或 `ANTHROPIC_VERTEX_BASE_URL` 设置）

两种配置可以同时使用。

将 Bedrock 与企业代理结合使用

通过企业 HTTP/HTTPS 代理路由 Bedrock 流量：

```
## 启用 Bedrock
export CLAUDE_CODE_USE_BEDROCK=1
export AWS_REGION=us-east-1

## 配置企业代理
export HTTPS_PROXY='https://proxy.example.com:8080'
```

将 Bedrock 与 LLM 网关结合使用

使用提供 Bedrock 兼容端点的网关服务：

```
## 启用 Bedrock
export CLAUDE_CODE_USE_BEDROCK=1

## 配置 LLM 网关
export ANTHROPIC_BEDROCK_BASE_URL='https://your-llm-gateway.com/bedrock'
export CLAUDE_CODE_SKIP_BEDROCK_AUTH=1 # 如果网关处理 AWS 身份验证
```

将 Foundry 与企业代理结合使用

通过企业 HTTP/HTTPS 代理路由 Azure 流量：

```
## 启用 Microsoft Foundry
export CLAUDE_CODE_USE_FOUNDRY=1
export ANTHROPIC_FOUNDRY_RESOURCE=your-resource
export ANTHROPIC_FOUNDRY_API_KEY=your-api-key # 或省略以使用 Entra ID 身份验证

## 配置企业代理
export HTTPS_PROXY='https://proxy.example.com:8080'
```

将 Foundry 与 LLM 网关结合使用

使用提供 Azure 兼容端点的网关服务：

```
## 启用 Microsoft Foundry
export CLAUDE_CODE_USE_FOUNDRY=1

## 配置 LLM 网关
export ANTHROPIC_FOUNDRY_BASE_URL='https://your-llm-gateway.com'
export CLAUDE_CODE_SKIP_FOUNDRY_AUTH=1 # 如果网关处理 Azure 身份验证
```

将 Vertex AI 与企业代理结合使用

通过企业 HTTP/HTTPS 代理路由 Vertex AI 流量：

```
## 启用 Vertex
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=us-east5
export ANTHROPIC_VERTEX_PROJECT_ID=your-project-id

## 配置企业代理
export HTTPS_PROXY='https://proxy.example.com:8080'
```

将 Vertex AI 与 LLM 网关结合使用

将 Google Vertex AI 模型与 LLM 网关结合，实现集中式管理：

```
## 启用 Vertex
export CLAUDE_CODE_USE_VERTEX=1

## 配置 LLM 网关
export ANTHROPIC_VERTEX_BASE_URL='https://your-llm-gateway.com/vertex'
export CLAUDE_CODE_SKIP_VERTEX_AUTH=1 # 如果网关处理 GCP 身份验证
```

身份验证配置

Claude Code 在需要时使用 `ANTHROPIC_AUTH_TOKEN` 作为 `Authorization` 标头。
`SKIP_AUTH` 标志（`CLAUDE_CODE_SKIP_BEDROCK_AUTH`、`CLAUDE_CODE_SKIP_VERTEX_AUTH`）
用于 LLM 网关场景，其中网关处理提供商身份验证。

选择正确的部署配置

选择部署方法时，请考虑以下因素：

直接提供商访问

最适合以下组织：

- 希望最简单的设置
- 已有 AWS 或 GCP 基础设施
- 需要提供商原生监控和合规性

企业代理

最适合以下组织：

- 有现有的企业代理需求
- 需要流量监控和合规性
- 必须通过特定网络路径路由所有流量

LLM 网关

最适合以下组织：

- 需要跨团队的使用情况跟踪
- 希望在模型之间动态切换
- 需要自定义速率限制或预算
- 需要集中式身份验证管理

调试

调试部署时：

- 使用 `claude /status` [斜杠命令](#)。此命令提供对任何应用的身份验证、代理和 URL 设置的可见性。
- 设置环境变量 `export ANTHROPIC_LOG=debug` 以记录请求。

组织的最佳实践

1. 投资文档和记忆

我们强烈建议投资文档，以便 Claude Code 理解您的代码库。组织可以在多个级别部署 CLAUDE.md 文件：

- **组织范围：**部署到系统目录，如 `/Library/Application Support/ClaudeCode/CLAUDE.md`（macOS），用于公司范围的标准
- **存储库级别：**在存储库根目录中创建 `CLAUDE.md` 文件，包含项目架构、构建命令和贡献指南。将这些文件检入源代码控制，以便所有用户受益 [了解更多](#)。

2. 简化部署

如果您有自定义开发环境，我们发现创建一种“一键”安装 Claude Code 的方式是在组织中推动采用的关键。

3. 从引导式使用开始

鼓励新用户尝试 Claude Code 进行代码库问答，或在较小的错误修复或功能请求上使用。要求 Claude Code 制定计划。检查 Claude 的建议，如果偏离轨道，请提供反馈。随着时间的推移，当用户更好地理解这种新范式时，他们将更有效地让 Claude Code 更自主地运行。

4. 配置安全策略

安全团队可以配置托管权限，以确定 Claude Code 允许和不允许做什么，这不能被本地配置覆盖。 [了解更多](#)。

5. 利用 MCP 进行集成

MCP 是为 Claude Code 提供更多信息的好方法，例如连接到票证管理系统或错误日志。我们建议一个中央团队配置 MCP 服务器，并将 `.mcp.json` 配置检入代码库，以便所有用户受益。 [了解更多](#)。

在 Anthropic，我们信任 Claude Code 为每个 Anthropic 代码库的开发提供动力。我们希望您像我们一样享受使用 Claude Code。

后续步骤

- [设置 Amazon Bedrock](#) 用于 AWS 原生部署
- [配置 Google Vertex AI](#) 用于 GCP 部署
- [设置 Microsoft Foundry](#) 用于 Azure 部署
- [配置企业网络](#) 用于网络需求
- [部署 LLM 网关](#) 用于企业管理
- [设置](#) 用于配置选项和环境变量

配置服务器管理的设置（公开测试版）

通过 Claude.ai 上基于网络的界面为您的组织集中配置 Claude Code，无需设备管理基础设施。

服务器管理的设置允许管理员通过 Claude.ai 上的网络界面集中配置 Claude Code。Claude Code 客户端在用户使用其组织凭证进行身份验证时自动接收这些设置。这种方法专为没有设备管理基础设施的组织或需要为非托管设备上的用户管理设置的组织而设计。

Note:

服务器管理的设置处于公开测试版，可供 [Claude for Teams](#) 和 [Claude for Enterprise](#) 客户使用。功能可能在正式发布前进行演变。

要求

要使用服务器管理的设置，您需要：

- Claude for Teams 或 Claude for Enterprise 计划
- Claude for Teams 的 Claude Code 版本 2.1.38 或更高版本，或 Claude for Enterprise 的版本 2.1.30 或更高版本
- 对 api.anthropic.com 的网络访问

在服务器管理和端点管理的设置之间选择

Claude Code 支持两种集中配置方法。服务器管理的设置从 Anthropic 的服务器传递配置。[端点管理的设置](#)通过本机操作系统策略（macOS 托管首选项、Windows 注册表）或托管设置文件直接部署到设备。

方法	最适合	安全模型
服务器管理的设置	没有 MDM 的组织，或非托管设备上的用户	在身份验证时从 Anthropic 的服务器传递的设置
端点管理的设置	具有 MDM 或端点管理的组织	通过 MDM 配置文件、注册表策略或托管设置文件部署到设备的设置

如果您的设备已在 MDM 或端点管理解决方案中注册，端点管理的设置提供更强的安全保障，因为设置文件可以在操作系统级别受到保护，防止用户修改。

配置服务器管理的设置

Step 1: 打开管理控制台

在 [Claude.ai](#) 中，导航到 **Admin Settings > Claude Code > Managed settings**。

Step 2: 定义您的设置

将您的配置添加为 JSON。支持 `settings.json` 中可用的所有设置，包括[仅限托管的设置](#)，如 `disableBypassPermissionsMode`。

此示例强制执行权限拒绝列表并防止用户绕过权限：

```
{
  "permissions": {
    "deny": [
      "Bash(curl *)",
      "Read(./env)",
      "Read(./env.*)",
      "Read(./secrets/**)"
    ]
  },
  "disableBypassPermissionsMode": "disable"
}
```

Step 3: 保存并部署

保存您的更改。Claude Code 客户端在下次启动或每小时轮询周期时接收更新的设置。

验证设置传递

要确认设置正在被应用，请要求用户重新启动 Claude Code。如果配置包含触发[安全批准对话框](#)的设置，用户会在启动时看到描述托管设置的提示。您还可以通过让用户运行 `/permissions` 来验证托管权限规则是否处于活动状态，以查看其有效的权限规则。

访问控制

以下角色可以管理服务器管理的设置：

- 主要所有者
- 所有者

限制对受信任人员的访问，因为设置更改适用于组织中的所有用户。

当前限制

服务器管理的设置在测试版期间有以下限制：

- 设置统一应用于组织中的所有用户。尚不支持按组配置。
- [MCP 服务器配置](#)无法通过服务器管理的设置分发。

设置传递

设置优先级

服务器管理的设置和[端点管理的设置](#)都占据 Claude Code [设置层次结构](#)中的最高层。没有其他设置级别可以覆盖它们，包括命令行参数。当两者都存在时，服务器管理的设置优先，端点管理的设置不被使用。

获取和缓存行为

Claude Code 在启动时从 Anthropic 的服务器获取设置，并在活动会话期间每小时轮询一次更新。

首次启动而无缓存的设置：

- Claude Code 异步获取设置
- 如果获取失败，Claude Code 继续运行而不使用托管设置
- 在设置加载之前有一个简短的窗口，其中限制尚未被强制执行

后续启动且有缓存的设置：

- 缓存的设置启动后立即应用
- Claude Code 在后台获取新鲜设置
- 缓存的设置通过网络故障持久化

Claude Code 自动应用设置更新而无需重新启动，除了高级设置（如 OpenTelemetry 配置）需要完全重新启动才能生效。

安全批准对话框

某些可能带来安全风险设置在应用前需要明确的用户批准：

- **Shell 命令设置：**执行 shell 命令的设置
- **自定义环境变量：**不在已知安全允许列表中的变量
- **Hook 配置：**任何 hook 定义

当这些设置存在时，用户会看到一个安全对话框，解释正在配置的内容。用户必须批准才能继续。如果用户拒绝设置，Claude Code 会退出。

Note:

在使用 `-p` 标志的非交互模式下，Claude Code 跳过安全对话框并在没有用户批准的情况下应用设置。

平台可用性

服务器管理的设置需要与 `api.anthropic.com` 的直接连接，在使用第三方模型提供商时不可用：

- Amazon Bedrock
- Google Vertex AI
- Microsoft Foundry
- 通过 `ANTHROPIC_BASE_URL` 或 [LLM 网关](#) 的自定义 API 端点

审计日志

设置更改的审计日志事件可通过合规 API 或审计日志导出获得。请联系您的 Anthropic 账户团队以获取访问权限。

审计事件包括执行的操作类型、执行操作的账户和设备，以及对先前值和新值的引用。

安全考虑

服务器管理的设置提供集中的策略强制执行，但它们作为客户端控制运行。在非托管设备上，具有管理员或 `sudo` 访问权限的用户可以修改 Claude Code 二进制文件、文件系统或网络配置。

场景	行为
用户编辑缓存的设置文件	篡改的文件在启动时应用，但正确的设置在下次服务器获取时恢复
用户删除缓存的设置文件	首次启动行为发生：设置异步获取，有一个简短的未强制执行的窗口
API 不可用	如果可用，缓存的设置应用，否则托管设置在下次成功获取前不被强制执行
用户使用不同的组织进行身份验证	不为托管组织外的账户传递设置

场景	行为
用户设置非默认的 <code>ANTHROPIC_BASE_URL</code>	使用第三方 API 提供商时，服务器管理的设置被绕过

要检测运行时配置更改，请使用 `ConfigChange hooks` 来记录修改或在未授权的更改生效前阻止它们。

为了获得更强的强制执行保证，请在已在 MDM 解决方案中注册的设备上使用[端点管理的设置](#)。

另请参阅

用于管理 Claude Code 配置的相关页面：

- [Settings](#)：完整的配置参考，包括所有可用的设置
- [Endpoint-managed settings](#)：由 IT 部门部署到设备的托管设置
- [Authentication](#)：设置用户对 Claude Code 的访问
- [Security](#)：安全保障和最佳实践

Part 11: Cloud Providers

Amazon Bedrock 上的 Claude Code

了解如何通过 Amazon Bedrock 配置 Claude Code，包括设置、IAM 配置和故障排除。

前置条件

在使用 Bedrock 配置 Claude Code 之前，请确保您拥有：

- 启用了 Bedrock 访问权限的 AWS 账户
- 在 Bedrock 中访问所需的 Claude 模型（例如 Claude Sonnet 4.6）
- 已安装并配置 AWS CLI（可选 - 仅在您没有其他获取凭证的机制时需要）
- 适当的 IAM 权限

Note:

如果您要将 Claude Code 部署给多个用户，请[固定您的模型版本](#)，以防止在 Anthropic 发布新模型时出现破损。

设置

1. 提交用例详情

Anthropic 模型的首次用户需要在调用模型之前提交用例详情。这是每个账户执行一次的操作。

1. 确保您拥有正确的 IAM 权限（请参阅下面的更多信息）
2. 导航到 [Amazon Bedrock 控制台](#)
3. 选择聊天/文本游乐场
4. 选择任何 Anthropic 模型，您将被提示填写用例表单

2. 配置 AWS 凭证

Claude Code 使用默认的 AWS SDK 凭证链。使用以下方法之一设置您的凭证：

选项 A：AWS CLI 配置


```
aws configure
```

选项 B：环境变量（访问密钥）

```
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```

选项 C：环境变量（SSO 配置文件）

```
aws sso login --profile=<your-profile-name>

export AWS_PROFILE=your-profile-name
```

选项 D：AWS 管理控制台凭证

```
aws login
```

[了解更多](#)关于 `aws login`。

选项 E：Bedrock API 密钥

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

Bedrock API 密钥提供了一种更简单的身份验证方法，无需完整的 AWS 凭证。[了解更多关于 Bedrock API 密钥](#)。

高级凭证配置

Claude Code 支持 AWS SSO 和企业身份提供商的自动凭证刷新。将这些设置添加到您的 Claude Code 设置文件（请参阅[设置](#)了解文件位置）。

当 Claude Code 检测到您的 AWS 凭证已过期（基于本地时间戳或当 Bedrock 返回凭证错误时），它将自动运行您配置的 `awsAuthRefresh` 和/或 `awsCredentialExport` 命令来获取新凭证，然后重试请求。

示例配置

```
{
  "awsAuthRefresh": "aws sso login --profile myprofile",
  "env": {
    "AWS_PROFILE": "myprofile"
  }
}
```

配置设置说明

awsAuthRefresh：用于修改 `.aws` 目录的命令，例如更新凭证、SSO 缓存或配置文件。命令的输出显示给用户，但不支持交互式输入。这适用于基于浏览器的 SSO 流，其中 CLI 显示 URL 或代码，您在浏览器中完成身份验证。

awsCredentialExport：仅在您无法修改 `.aws` 且必须直接返回凭证时使用。输出被静默捕获，不显示给用户。命令必须以此格式输出 JSON：

```
{
  "Credentials": {
    "AccessKeyId": "value",
    "SecretAccessKey": "value",
    "SessionToken": "value"
  }
}
```

3. 配置 Claude Code

设置以下环境变量以启用 Bedrock：

```
## 启用 Bedrock 集成
export CLAUDE_CODE_USE_BEDROCK=1
export AWS_REGION=us-east-1 # 或您首选的区域

## 可选：覆盖小型/快速模型（Haiku）的区域
export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2
```

为 Claude Code 启用 Bedrock 时，请记住以下几点：

- **AWS_REGION** 是必需的环境变量。Claude Code 不会从 `.aws` 配置文件中读取此设置。

- 使用 Bedrock 时，`/login` 和 `/logout` 命令被禁用，因为身份验证通过 AWS 凭证处理。
- 您可以使用设置文件来处理环境变量，如 `AWS_PROFILE`，您不希望泄露给其他进程。请参阅[设置](#)了解更多信息。

4. 固定模型版本

Warning:

为每个部署固定特定的模型版本。如果您使用模型别名（`sonnet`、`opus`、`haiku`）而不固定版本，Claude Code 可能会尝试使用您的 Bedrock 账户中不可用的较新模型版本，在 Anthropic 发布更新时破坏现有用户。

将这些环境变量设置为特定的 Bedrock 模型 ID：

```
export ANTHROPIC_DEFAULT_OPUS_MODEL='us.anthropic.claude-opus-4-6-v1'
export ANTHROPIC_DEFAULT_SONNET_MODEL='us.anthropic.claude-sonnet-4-6'
export ANTHROPIC_DEFAULT_HAIKU_MODEL='us.anthropic.claude-haiku-4-5-20251001-v1:0'
```

这些变量使用跨区域推理配置文件 ID（带有 `us.` 前缀）。如果您使用不同的区域前缀或应用推理配置文件，请相应调整。有关当前和旧版模型 ID，请参阅[模型概览](#)。请参阅[模型配置](#)了解完整的环境变量列表。

当未设置固定变量时，Claude Code 使用这些默认模型：

模型类型	默认值
主模型	<code>global.anthropic.claude-sonnet-4-6</code>
小型/快速模型	<code>us.anthropic.claude-haiku-4-5-20251001-v1:0</code>

要进一步自定义模型，请使用以下方法之一：

```
## 使用推理配置文件 ID
export ANTHROPIC_MODEL='global.anthropic.claude-sonnet-4-6'
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-haiku-4-5-20251001-v1:0'

## 使用应用推理配置文件 ARN
export ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-
inference-profile/your-model-id'

## 可选: 如果需要, 禁用 prompt caching
export DISABLE_PROMPT_CACHING=1
```

Note:

[Prompt caching](#) 可能在所有区域都不可用。

将每个模型版本映射到推理配置文件

`ANTHROPIC_DEFAULT_*_MODEL` 环境变量为每个模型系列配置一个推理配置文件。如果您的组织需要在 `/model` 选择器中公开同一系列的多个版本, 每个版本路由到其自己的应用推理配置文件 ARN, 请改用[设置文件](#)中的 `modelOverrides` 设置。

此示例将三个 Opus 版本映射到不同的 ARN, 以便用户可以在它们之间切换, 而无需绕过您组织的推理配置文件:

```
{
  "modelOverrides": {
    "claude-opus-4-6": "arn:aws:bedrock:us-east-2:123456789012:application-
inference-profile/opus-46-prod",
    "claude-opus-4-5-20251101": "arn:aws:bedrock:us-
east-2:123456789012:application-inference-profile/opus-45-prod",
    "claude-opus-4-1-20250805": "arn:aws:bedrock:us-
east-2:123456789012:application-inference-profile/opus-41-prod"
  }
}
```

当用户在 `/model` 中选择其中一个版本时, Claude Code 使用映射的 ARN 调用 Bedrock。没有覆盖的版本回退到内置的 Bedrock 模型 ID 或启动时发现的任何匹配推理配置文件。请参阅[按版本覆盖模型 ID](#)了解覆盖如何与 `availableModels` 和其他模型设置交互的详情。

IAM 配置

创建具有 Claude Code 所需权限的 IAM 策略：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowModelAndInferenceProfileAccess",
      "Effect": "Allow",
      "Action": [
        "bedrock:InvokeModel",
        "bedrock:InvokeModelWithResponseStream",
        "bedrock:ListInferenceProfiles"
      ],
      "Resource": [
        "arn:aws:bedrock:*:*:inference-profile/*",
        "arn:aws:bedrock:*:*:application-inference-profile/*",
        "arn:aws:bedrock:*:*:foundation-model/*"
      ]
    },
    {
      "Sid": "AllowMarketplaceSubscription",
      "Effect": "Allow",
      "Action": [
        "aws-marketplace:ViewSubscriptions",
        "aws-marketplace:Subscribe"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:CalledViaLast": "bedrock.amazonaws.com"
        }
      }
    }
  ]
}
```

为了获得更严格的权限，您可以将资源限制为特定的推理配置文件 ARN。

有关详情，请参阅 [Bedrock IAM 文档](#)。

Note:

为 Claude Code 创建一个专用的 AWS 账户，以简化成本跟踪和访问控制。

AWS Guardrails

[Amazon Bedrock Guardrails](#) 让您为 Claude Code 实现内容过滤。在 [Amazon Bedrock 控制台](#) 中创建 Guardrail，发布一个版本，然后将 Guardrail 标头添加到您的 [设置文件](#)。如果您使用跨区域推理配置文件，请在您的 Guardrail 上启用跨区域推理。

示例配置：

```
{
  "env": {
    "ANTHROPIC_CUSTOM_HEADERS": "X-Amzn-Bedrock-GuardrailIdentifier: your-guardrail-id\nX-Amzn-Bedrock-GuardrailVersion: 1"
  }
}
```

故障排除

如果您遇到区域问题：

- 检查模型可用性：`aws bedrock list-inference-profiles --region your-region`
- 切换到支持的区域：`export AWS_REGION=us-east-1`
- 考虑使用推理配置文件进行跨区域访问

如果您收到错误“不支持按需吞吐量”：

- 将模型指定为[推理配置文件](#) ID

Claude Code 使用 Bedrock [Invoke API](#)，不支持 Converse API。

其他资源

- [Bedrock 文档](#)
- [Bedrock 定价](#)
- [Bedrock 推理配置文件](#)
- [Amazon Bedrock 上的 Claude Code：快速设置指南](#)

- [Claude Code 监控实现 \(Bedrock\)](#)

Google Vertex AI 上的 Claude Code

了解如何通过 Google Vertex AI 配置 Claude Code，包括设置、IAM 配置和故障排除。

前置条件

在使用 Vertex AI 配置 Claude Code 之前，请确保您拥有：

- 启用了计费的 Google Cloud Platform (GCP) 账户
- 启用了 Vertex AI API 的 GCP 项目
- 对所需 Claude 模型的访问权限（例如，Claude Sonnet 4.5）
- 已安装并配置的 Google Cloud SDK (`gcloud`)
- 在所需 GCP 区域中分配的配额

区域配置

Claude Code 可以与 Vertex AI [全球](#)和区域端点一起使用。

Note:

Vertex AI 可能不支持所有区域上的 Claude Code 默认模型。您可能需要切换到[支持的区域或模型](#)。

Note:

Vertex AI 可能不支持全球端点上的 Claude Code 默认模型。您可能需要切换到区域端点或[支持的模型](#)。

设置

1. 启用 Vertex AI API

在您的 GCP 项目中启用 Vertex AI API：


```
## 设置您的项目 ID
gcloud config set project YOUR-PROJECT-ID

## 启用 Vertex AI API
gcloud services enable aiplatform.googleapis.com
```

2. 请求模型访问权限

请求在 Vertex AI 中访问 Claude 模型：

1. 导航到 [Vertex AI Model Garden](#)
2. 搜索“Claude”模型
3. 请求访问所需的 Claude 模型（例如，Claude Sonnet 4.5）
4. 等待批准（可能需要 24-48 小时）

3. 配置 GCP 凭证

Claude Code 使用标准的 Google Cloud 身份验证。

有关更多信息，请参阅 [Google Cloud 身份验证文档](#)。

Note:

进行身份验证时，Claude Code 将自动使用 `ANTHROPIC_VERTEX_PROJECT_ID` 环境变量中的项目 ID。要覆盖此设置，请设置以下环境变量之一：`GCPLOUD_PROJECT`、`GOOGLE_CLOUD_PROJECT` 或 `GOOGLE_APPLICATION_CREDENTIALS`。

4. 配置 Claude Code

设置以下环境变量：

```
## 启用 Vertex AI 集成
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=global
export ANTHROPIC_VERTEX_PROJECT_ID=YOUR-PROJECT-ID

## 可选: 如果需要, 禁用 prompt caching
export DISABLE_PROMPT_CACHING=1

## 当 CLOUD_ML_REGION=global 时, 为不支持的模型覆盖区域
export VERTEX_REGION_CLAUDE_3_5_HAIKU=us-east5

## 可选: 为其他特定模型覆盖区域
export VERTEX_REGION_CLAUDE_3_5_SONNET=us-east5
export VERTEX_REGION_CLAUDE_3_7_SONNET=us-east5
export VERTEX_REGION_CLAUDE_4_0_OPUS=europe-west1
export VERTEX_REGION_CLAUDE_4_0_SONNET=us-east5
export VERTEX_REGION_CLAUDE_4_1_OPUS=europe-west1
```

Note:

当您指定 `cache_control` 临时标志时, `prompt caching` 会自动支持。要禁用它, 请设置 `DISABLE_PROMPT_CACHING=1`。如需提高速率限制, 请联系 Google Cloud 支持。

Note:

使用 Vertex AI 时, `/login` 和 `/logout` 命令被禁用, 因为身份验证通过 Google Cloud 凭证处理。

5. 模型配置

Claude Code 为 Vertex AI 使用这些默认模型:

模型类型	默认值
主模型	claude-sonnet-4-5@20250929
小型/快速模型	claude-haiku-4-5@20251001

Note:

对于 Vertex AI 用户，Claude Code 不会自动从 Haiku 3.5 升级到 Haiku 4.5。要手动切换到较新的 Haiku 模型，请将 `ANTHROPIC_DEFAULT_HAIKU_MODEL` 环境变量设置为完整模型名称（例如，`claude-haiku-4-5@20251001`）。

要自定义模型：

```
export ANTHROPIC_MODEL='claude-opus-4-1@20250805'
export ANTHROPIC_SMALL_FAST_MODEL='claude-haiku-4-5@20251001'
```

IAM 配置

分配所需的 IAM 权限：

`roles/aiplatform.user` 角色包括所需的权限：

- `aiplatform.endpoints.predict` - 模型调用和令牌计数所需

对于更严格的权限，请创建仅包含上述权限的自定义角色。

有关详细信息，请参阅 [Vertex IAM 文档](#)。

Note:

我们建议为 Claude Code 创建一个专用的 GCP 项目，以简化成本跟踪和访问控制。

1M token context window

Claude Sonnet 4 和 Sonnet 4.5 在 Vertex AI 上支持 [1M token context window](#)。

Note:

1M token context window 目前处于测试版。要使用扩展的 context window，请在您的 Vertex AI 请求中包含 `context-1m-2025-08-07` 测试版标头。

故障排除

如果您遇到配额问题：

- 通过 [Cloud Console](#) 检查当前配额或请求增加配额

如果您遇到”模型未找到” 404 错误：

- 确认模型在 [Model Garden](#) 中已启用
- 验证您有权访问指定的区域
- 如果使用 `CLOUD_ML_REGION=global`，请检查您的模型是否在 [Model Garden](#) 中的”支持的功能”下支持全球端点。对于不支持全球端点的模型，请执行以下任一操作：
 - 通过 `ANTHROPIC_MODEL` 或 `ANTHROPIC_SMALL_FAST_MODEL` 指定支持的模型，或
 - 使用 `VERTEX_REGION=<MODEL_NAME>` 环境变量设置区域端点

如果您遇到 429 错误：

- 对于区域端点，请确保主模型和小型/快速模型在您选择的区域中受支持
- 考虑切换到 `CLOUD_ML_REGION=global` 以获得更好的可用性

其他资源

- [Vertex AI 文档](#)
- [Vertex AI 定价](#)
- [Vertex AI 配额和限制](#)

Claude Code on Microsoft Foundry

了解如何通过 Microsoft Foundry 配置 Claude Code，包括设置、配置和故障排除。

前置条件

在使用 Microsoft Foundry 配置 Claude Code 之前，请确保您拥有：

- 具有 Microsoft Foundry 访问权限的 Azure 订阅
- 创建 Microsoft Foundry 资源和部署的 RBAC 权限
- 已安装并配置 Azure CLI（可选 - 仅在您没有其他获取凭证机制时需要）

设置

1. 配置 Microsoft Foundry 资源

首先，在 Azure 中创建 Claude 资源：

1. 导航到 [Microsoft Foundry 门户](#)
2. 创建新资源，记下您的资源名称
3. 为 Claude 模型创建部署：
 - Claude Opus
 - Claude Sonnet
 - Claude Haiku

2. 配置 Azure 凭证

Claude Code 支持两种 Microsoft Foundry 身份验证方法。选择最适合您安全要求的方法。

选项 A：API 密钥身份验证

1. 在 Microsoft Foundry 门户中导航到您的资源
2. 转到端点和密钥部分
3. 复制 **API 密钥**
4. 设置环境变量：

```
export ANTHROPIC_FOUNDRY_API_KEY=your-azure-api-key
```

选项 B：Microsoft Entra ID 身份验证

当未设置 `ANTHROPIC_FOUNDRY_API_KEY` 时，Claude Code 会自动使用 Azure SDK [默认凭证链](#)。这支持多种方法来验证本地和远程工作负载。

在本地环境中，您通常可以使用 Azure CLI：

```
az login
```

Note:

使用 Microsoft Foundry 时，`/login` 和 `/logout` 命令被禁用，因为身份验证通过 Azure 凭证处理。

3. 配置 Claude Code

设置以下环境变量以启用 Microsoft Foundry。请注意，您的部署名称设置为 Claude Code 中的模型标识符（如果使用建议的部署名称，可能是可选的）。

```
## 启用 Microsoft Foundry 集成
export CLAUDE_CODE_USE_FOUNDRY=1

## Azure 资源名称（将 {resource} 替换为您的资源名称）
export ANTHROPIC_FOUNDRY_RESOURCE={resource}
## 或提供完整的基础 URL：
## export ANTHROPIC_FOUNDRY_BASE_URL=https://{resource}.services.ai.azure.com

## 将模型设置为您的资源的部署名称
export ANTHROPIC_DEFAULT_SONNET_MODEL='claude-sonnet-4-5'
export ANTHROPIC_DEFAULT_HAIKU_MODEL='claude-haiku-4-5'
export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-1'
```

有关模型配置选项的更多详情，请参阅[模型配置](#)。

Azure RBAC 配置

`Azure AI User` 和 `Cognitive Services User` 默认角色包括调用 Claude 模型所需的所有权限。

对于更严格的权限，请创建具有以下内容的自定义角色：

```
{
  "permissions": [
    {
      "dataActions": [
        "Microsoft.CognitiveServices/accounts/providers/*"
      ]
    }
  ]
}
```

有关详情，请参阅 [Microsoft Foundry RBAC 文档](#)。

故障排除

如果您收到错误” Failed to get token from azureADTokenProvider: ChainedTokenCredential authentication failed”：

- 在环境中配置 Entra ID，或设置 `ANTHROPIC_FOUNDRY_API_KEY`。

其他资源

- [Microsoft Foundry 文档](#)
- [Microsoft Foundry 模型](#)
- [Microsoft Foundry 定价](#)

LLM gateway 配置

了解如何配置 Claude Code 以使用 LLM gateway 解决方案。涵盖网关要求、身份验证配置、模型选择和特定提供商的端点设置。

LLM gateway 提供了 Claude Code 和模型提供商之间的集中代理层，通常提供以下功能：

- **集中身份验证** - API 密钥管理的单一入口
- **使用情况跟踪** - 监控团队和项目的使用情况
- **成本控制** - 实施预算和速率限制
- **审计日志** - 跟踪所有模型交互以实现合规性
- **模型路由** - 无需更改代码即可在提供商之间切换

网关要求

为了使 LLM gateway 与 Claude Code 配合使用，它必须满足以下要求：

API 格式

网关必须向客户端公开以下至少一种 API 格式：

1. **Anthropic Messages**: `/v1/messages` , `/v1/messages/count_tokens`
 - 必须转发请求头: `anthropic-beta` 、 `anthropic-version`
2. **Bedrock InvokeModel**: `/invoke` , `/invoke-with-response-stream`
 - 必须保留请求体字段: `anthropic_beta` 、 `anthropic_version`
3. **Vertex rawPredict**: `:rawPredict` 、 `:streamRawPredict` 、 `/count-tokens:rawPredict`
 - 必须转发请求头: `anthropic-beta` 、 `anthropic-version`

未能转发请求头或保留请求体字段可能导致功能减少或无法使用 Claude Code 功能。

Note:

Claude Code 根据 API 格式确定要启用的功能。当使用 Bedrock 或 Vertex 的 Anthropic Messages 格式时，您可能需要设置环境变量

`CLAUDE_CODE_DISABLE_EXPERIMENTAL_BETAS=1` 。

配置

模型选择

默认情况下，Claude Code 将为选定的 API 格式使用标准模型名称。

如果您在网关中配置了自定义模型名称，请使用 [模型配置](#) 中记录的环境变量来匹配您的自定义名称。

LiteLLM 配置

Note:

LiteLLM 是第三方代理服务。Anthropic 不认可、维护或审计 LiteLLM 的安全性或功能。本指南仅供参考，可能会过时。请自行判断使用。

前置条件

- Claude Code 更新到最新版本
- LiteLLM Proxy Server 已部署且可访问
- 通过您选择的提供商访问 Claude 模型

基本 LiteLLM 设置

配置 Claude Code :

身份验证方法

静态 API 密钥

使用固定 API 密钥的最简单方法：

```
## 在环境中设置
export ANTHROPIC_AUTH_TOKEN=sk-litellm-static-key

## 或在 Claude Code 设置中
{
  "env": {
    "ANTHROPIC_AUTH_TOKEN": "sk-litellm-static-key"
  }
}
```

此值将作为 `Authorization` 请求头发送。

使用辅助程序的动态 API 密钥

用于轮换密钥或按用户身份验证：

1. 创建 API 密钥辅助程序脚本：

```
#!/bin/bash
## ~/bin/get-litellm-key.sh

## 示例：从保险库获取密钥
vault kv get -field=api_key secret/litellm/claude-code

## 示例：生成 JWT 令牌
jwt encode \
  --secret="${JWT_SECRET}" \
  --exp="+1h" \
  '{"user":"'${USER}'","team":"engineering"}'
```

1. 配置 Claude Code 设置以使用辅助程序：

```
{
  "apiKeyHelper": "~/bin/get-litellm-key.sh"
}
```

1. 设置令牌刷新间隔：

```
## 每小时刷新一次（3600000 毫秒）
export CLAUDE_CODE_API_KEY_HELPER_TTL_MS=3600000
```

此值将作为 `Authorization` 和 `X-API-Key` 请求头发送。`apiKeyHelper` 的优先级低于 `ANTHROPIC_AUTH_TOKEN` 或 `ANTHROPIC_API_KEY`。

统一端点（推荐）

使用 LiteLLM 的 [Anthropic 格式端点](#)：

```
export ANTHROPIC_BASE_URL=https://litellm-server:4000
```

统一端点相对于直通端点的优势：

- 负载均衡
- 故障转移
- 对成本跟踪和最终用户跟踪的一致支持

特定提供商的直通端点（替代方案）

通过 LiteLLM 的 Claude API

使用 [直通端点](#)：

```
export ANTHROPIC_BASE_URL=https://litellm-server:4000/anthropic
```

通过 LiteLLM 的 Amazon Bedrock

使用 [直通端点](#)：

```
export ANTHROPIC_BEDROCK_BASE_URL=https://litellm-server:4000/bedrock
export CLAUDE_CODE_SKIP_BEDROCK_AUTH=1
export CLAUDE_CODE_USE_BEDROCK=1
```

通过 LiteLLM 的 Google Vertex AI

使用 [直通端点](#)：

```
export ANTHROPIC_VERTEX_BASE_URL=https://litellm-server:4000/vertex_ai/v1
export ANTHROPIC_VERTEX_PROJECT_ID=your-gcp-project-id
export CLAUDE_CODE_SKIP_VERTEX_AUTH=1
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=us-east5
```

有关更多详细信息，请参阅 [LiteLLM 文档](#)。

其他资源

- [LiteLLM 文档](#)
- [Claude Code 设置](#)
- [企业网络配置](#)
- [第三方集成概述](#)

企业网络配置

为企业环境配置 Claude Code，支持代理服务器、自定义证书颁发机构 (CA) 和相互传输层安全 (mTLS) 身份验证。

Claude Code 通过环境变量支持各种企业网络和安全配置。这包括通过公司代理服务器路由流量、信任自定义证书颁发机构 (CA)，以及使用相互传输层安全 (mTLS) 证书进行身份验证以增强安全性。

Note:

本页面显示的所有环境变量也可以在 `settings.json` 中配置。

代理配置

环境变量

Claude Code 遵守标准代理环境变量：

```
## HTTPS 代理 (推荐)
export HTTPS_PROXY=https://proxy.example.com:8080

## HTTP 代理 (如果 HTTPS 不可用)
export HTTP_PROXY=http://proxy.example.com:8080

## 绕过特定请求的代理 - 空格分隔格式
export NO_PROXY="localhost 192.168.1.1 example.com .example.com"
## 绕过特定请求的代理 - 逗号分隔格式
export NO_PROXY="localhost,192.168.1.1,example.com,.example.com"
## 绕过所有请求的代理
export NO_PROXY="*"

```

Note:

Claude Code 不支持 SOCKS 代理。

基本身份验证

如果您的代理需要基本身份验证，请在代理 URL 中包含凭据：

```
export HTTPS_PROXY=http://username:password@proxy.example.com:8080
```

Warning:

避免在脚本中硬编码密码。改用环境变量或安全凭据存储。

Tip:

对于需要高级身份验证（NTLM、Kerberos 等）的代理，请考虑使用支持您的身份验证方法的 LLM Gateway 服务。

自定义 CA 证书

如果您的企业环境使用自定义 CA 进行 HTTPS 连接（无论是通过代理还是直接 API 访问），请配置 Claude Code 以信任它们：

```
export NODE_EXTRA_CA_CERTS=/path/to/ca-cert.pem
```

mTLS 身份验证

对于需要客户端证书身份验证的企业环境：

```
## 用于身份验证的客户端证书
export CLAUDE_CODE_CLIENT_CERT=/path/to/client-cert.pem

## 客户端私钥
export CLAUDE_CODE_CLIENT_KEY=/path/to/client-key.pem

## 可选：加密私钥的密码短语
export CLAUDE_CODE_CLIENT_KEY_PASSPHRASE="your-passphrase"
```

网络访问要求

Claude Code 需要访问以下 URL：

- `api.anthropic.com`：Claude API 端点
- `claude.ai`：claude.ai 账户的身份验证
- `platform.claude.com`：Anthropic Console 账户的身份验证

确保这些 URL 在您的代理配置和防火墙规则中被列入白名单。这在容器化或受限网络环境中使用 Claude Code 时尤为重要。

其他资源

- [Claude Code 设置](#)
- [环境变量参考](#)
- [故障排除指南](#)

Part 12: Environment Setup

开发容器

了解 Claude Code 开发容器，适用于需要一致、安全环境的团队。

参考 [devcontainer 设置](#) 和相关的 [Dockerfile](#) 提供了一个预配置的开发容器，你可以按原样使用，也可以根据需要进行自定义。此 devcontainer 与 Visual Studio Code [Dev Containers 扩展](#) 和类似工具兼容。

容器的增强安全措施（隔离和防火墙规则）允许你运行 `claude --dangerously-skip-permissions` 来绕过权限提示，以实现无人值守操作。

Warning:

虽然 devcontainer 提供了实质性的保护，但没有任何系统能够完全免疫所有攻击。当使用 `--dangerously-skip-permissions` 执行时，devcontainer 不会阻止恶意项目泄露 devcontainer 中可访问的任何内容，包括 Claude Code 凭证。我们建议仅在使用受信任的存储库进行开发时使用 devcontainer。始终保持良好的安全实践并监控 Claude 的活动。

主要功能

- **生产就绪的 Node.js**：基于 Node.js 20 构建，包含必要的开发依赖项
- **安全设计**：自定义防火墙限制网络访问仅限于必要的服务
- **开发者友好的工具**：包括 git、ZSH 及生产力增强功能、fzf 等
- **无缝 VS Code 集成**：预配置的扩展和优化的设置
- **会话持久性**：在容器重启之间保留命令历史和配置
- **随处可用**：兼容 macOS、Windows 和 Linux 开发环境

4 步快速开始

1. 安装 VS Code 和 Remote - Containers 扩展
2. 克隆 [Claude Code 参考实现](#) 存储库
3. 在 VS Code 中打开存储库
4. 出现提示时，点击”在容器中重新打开”（或使用命令面板：Cmd+Shift+P → “Remote-Containers: Reopen in Container”）

配置分解

devcontainer 设置由三个主要组件组成：

- [devcontainer.json](#)：控制容器设置、扩展和卷挂载
- [Dockerfile](#)：定义容器镜像和已安装的工具
- [init-firewall.sh](#)：建立网络安全规则

安全功能

容器通过其防火墙配置实现了多层安全方法：

- **精确访问控制**：将出站连接限制为仅白名单域（npm 注册表、GitHub、Claude API 等）
- **允许的出站连接**：防火墙允许出站 DNS 和 SSH 连接
- **默认拒绝策略**：阻止所有其他外部网络访问
- **启动验证**：在容器初始化时验证防火墙规则
- **隔离**：创建与主系统分离的安全开发环境

自定义选项

devcontainer 配置设计为可适应你的需求：

- 根据你的工作流添加或删除 VS Code 扩展
- 为不同的硬件环境修改资源分配
- 调整网络访问权限
- 自定义 shell 配置和开发者工具

示例用例

安全的客户端工作

使用 devcontainer 隔离不同的客户端项目，确保代码和凭证不会在环境之间混合。

团队入职

新团队成员可以在几分钟内获得完全配置的开发环境，所有必要的工具和设置都已预装。

一致的 CI/CD 环境

在 CI/CD 管道中镜像你的 devcontainer 配置，以确保开发和生产环境匹配。

相关资源

- [VS Code devcontainer 文档](#)
- [Claude Code 安全最佳实践](#)
- [企业网络配置](#)

优化您的终端设置

Claude Code 在终端配置正确时效果最佳。请按照这些指南来优化您的体验。

主题和外观

Claude 无法控制您的终端主题。这由您的终端应用程序处理。您可以随时通过 `/config` 命令将 Claude Code 的主题与您的终端相匹配。

为了进一步自定义 Claude Code 界面本身，您可以配置一个[自定义状态行](#)来显示上下文信息，如当前模型、工作目录或 git 分支在您的终端底部。

换行符

您有多个选项可以在 Claude Code 中输入换行符：

- **快速转义**：输入 `\` 后按 Enter 键创建新行
- **Shift+Enter**：在 iTerm2、WezTerm、Ghostty 和 Kitty 中开箱即用
- **键盘快捷键**：在其他终端中设置键绑定以插入新行

为其他终端设置 Shift+Enter

在 Claude Code 中运行 `/terminal-setup` 以自动为 VS Code、Alacritty、Zed 和 Warp 配置 Shift+Enter。

Note:

`/terminal-setup` 命令仅在需要手动配置的终端中可见。如果您使用的是 iTerm2、WezTerm、Ghostty 或 Kitty，您将看不到此命令，因为 Shift+Enter 已经原生工作。

设置 Option+Enter（VS Code、iTerm2 或 macOS Terminal.app）

对于 Mac Terminal.app：

1. 打开设置 → 配置文件 → 键盘
2. 勾选“使用 Option 作为 Meta 键”

对于 iTerm2 和 VS Code 终端：

1. 打开设置 → 配置文件 → 键
2. 在常规下，将左/右 Option 键设置为“Esc+”

通知设置

通过正确的通知配置，永远不会错过 Claude 完成任务的时刻：

iTerm 2 系统通知

对于任务完成时的 iTerm 2 警报：

1. 打开 iTerm 2 偏好设置
2. 导航到配置文件 → 终端
3. 启用“静音铃声”和过滤警报 → “发送转义序列生成的警报”
4. 设置您首选的通知延迟

请注意，这些通知特定于 iTerm 2，在默认的 macOS 终端中不可用。

自定义通知钩子

对于高级通知处理，您可以创建[通知钩子](#)来运行您自己的逻辑。

处理大型输入

处理大量代码或长指令时：

- **避免直接粘贴**：Claude Code 可能难以处理非常长的粘贴内容
- **使用基于文件的工作流**：将内容写入文件并要求 Claude 读取它
- **注意 VS Code 的限制**：VS Code 终端特别容易截断长粘贴

Vim 模式

Claude Code 支持可以通过 `/vim` 启用或通过 `/config` 配置的 Vim 键绑定子集。

支持的子集包括：

- 模式切换：`Esc`（到 NORMAL）、`i/I`、`a/A`、`o/O`（到 INSERT）
- 导航：`h/j/k/l`、`w/e/b`、`0/$/^`、`gg/G`、`f/F/t/T` 带 `;`，重复
- 编辑：`x`、`dw/de/db/dd/D`、`cw/ce/cb/cc/C`、`.`（重复）
- 复制/粘贴：`yy/Y`、`yw/ye/yb`、`p/P`
- 文本对象：`iw/aw`、`iW/aW`、`i"/a"`、`i'/a'`、`i(/a(`、`i[/a[`、`i{/a{`
- 缩进：`>>` / `<<`
- 行操作：`J`（连接行）

请参阅[交互模式](#)获取完整参考。

自定义你的状态行

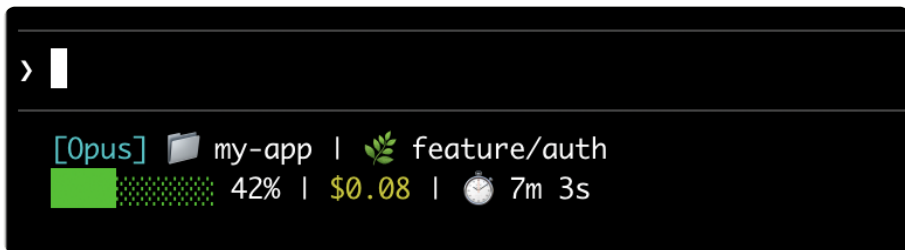
配置自定义状态栏以监控 Claude Code 中的上下文窗口使用情况、成本和 git 状态

状态行是 Claude Code 底部的可自定义栏，可以运行你配置的任何 shell 脚本。它通过 stdin 接收 JSON 会话数据，并显示你的脚本打印的任何内容，为你提供一个持久的、一目了然的上下文使用情况、成本、git 状态或任何其他你想跟踪的内容的视图。

状态行在以下情况下很有用：

- 你想在工作时监控上下文窗口使用情况
- 你需要跟踪会话成本
- 你在多个会话中工作，需要区分它们
- 你希望 git 分支和状态始终可见

这是一个 [多行状态行](#) 的示例，它在第一行显示 git 信息，在第二行显示颜色编码的上下文栏。



一个多行状态行，显示第一行上的模型名称、目录、git 分支，第二行上的上下文使用进度条、成本和持续时间

本页面介绍了 [设置基本状态行](#)，解释了 [数据如何从 Claude Code 流向你的脚本](#)，列出了 [你可以显示的所有字段](#)，并提供了 [常见模式的现成示例](#)，如 git 状态、成本跟踪和进度条。

设置状态行

使用 `/statusline` 命令让 Claude Code 为你生成脚本，或 [手动创建脚本](#) 并将其添加到你的设置中。

使用 `/statusline` 命令

`/statusline` 命令接受描述你想显示的内容的自然语言指令。Claude Code 在 `~/.claude/` 中生成脚本文件并自动更新你的设置：

```
/statusline show model name and context percentage with a progress bar
```

手动配置状态行

将 `statusLine` 字段添加到你的用户设置（`~/.claude/settings.json`，其中 `~` 是你的主目录）或[项目设置](#)。将 `type` 设置为 `"command"` 并将 `command` 指向脚本路径或内联 shell 命令。有关创建脚本的完整演练，请参阅[逐步构建状态行](#)。

```
{
  "statusLine": {
    "type": "command",
    "command": "~/.claude/statusline.sh",
    "padding": 2
  }
}
```

`command` 字段在 shell 中运行，所以你也可以使用内联命令而不是脚本文件。此示例使用 `jq` 解析 JSON 输入并显示模型名称和上下文百分比：

```
{
  "statusLine": {
    "type": "command",
    "command": "jq -r '\"[\\$(.model.display_name)] \\$(.context_window.used_percent  
age // 0)% context\\\"'"
  }
}
```

可选的 `padding` 字段为状态行内容添加额外的水平间距（以字符为单位）。默认为 `0`。此填充是在界面的内置间距之外的，所以它控制相对缩进而不是距离终端边缘的绝对距离。

禁用状态行

运行 `/statusline` 并要求它删除或清除你的状态行（例如，`/statusline delete`、`/statusline clear`、`/statusline remove it`）。你也可以手动从 `settings.json` 中删除 `statusLine` 字段。

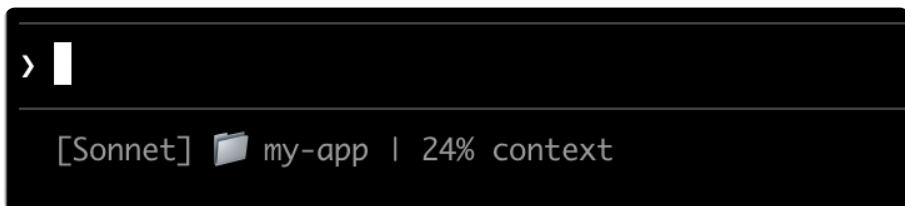
逐步构建状态行

本演练展示了通过手动创建显示当前模型、工作目录和上下文窗口使用百分比的状态行来了解幕后发生的情况。

Note:

使用 `/statusline` 和你想要的内容的描述会自动为你配置所有这些。

这些示例使用 Bash 脚本，在 macOS 和 Linux 上工作。在 Windows 上，请参阅[Windows 配置](#)以获取 PowerShell 和 Git Bash 示例。



一个状态行，显示模型名称、目录和上下文百分比

Step 1: 创建一个读取 JSON 并打印输出的脚本

Claude Code 通过 stdin 向你的脚本发送 JSON 数据。此脚本使用 `jq`（一个你可能需要安装的命令行 JSON 解析器）来提取模型名称、目录和上下文百分比，然后打印格式化的行。

将其保存到 `~/.claude/statusline.sh`（其中 `~` 是你的主目录，例如 macOS 上的 `/Users/username` 或 Linux 上的 `/home/username`）：

```
#!/bin/bash

## Read JSON data that Claude Code sends to stdin
input=$(cat)

## Extract fields using jq
MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')
## The "/" provides a fallback if the field is null
PCT=$(echo "$input" | jq -r '.context_window.used_percentage // 0' | cut -d. -f1)

## Output the status line - ${DIR##*/} extracts just the folder name
echo "[$MODEL] 📁 ${DIR##*/} | ${PCT}% context"
```

Step 2: 使其可执行

将脚本标记为可执行，以便你的 shell 可以运行它：

```
chmod +x ~/.claude/statusline.sh
```

Step 3: 添加到设置

告诉 Claude Code 运行你的脚本作为状态行。将此配置添加到 `~/.claude/settings.json`，它将 `type` 设置为 `"command"`（意思是“运行此 shell 命令”）并将 `command` 指向你的脚本：

```
{
  "statusLine": {
    "type": "command",
    "command": "~/.claude/statusline.sh"
  }
}
```

你的状态行出现在界面的底部。设置会自动重新加载，但更改在你与 Claude Code 的下次交互之前不会出现。

状态行如何工作

Claude Code 运行你的脚本并通过 stdin 向其传输 [JSON 会话数据](#)。你的脚本读取 JSON，提取它需要的内容，并将文本打印到 stdout。Claude Code 显示你的脚本打印的任何内容。

何时更新

你的脚本在每条新的助手消息之后、权限模式更改时或 vim 模式切换时运行。更新在 300ms 处进行防抖，这意味着快速更改会批处理在一起，你的脚本在事情稳定后运行一次。如果在你的脚本仍在运行时触发新的更新，则会取消正在进行的执行。如果你编辑你的脚本，更改在你与 Claude Code 的下一交互触发更新之前不会出现。

你的脚本可以输出什么

- **多行**：每个 `echo` 或 `print` 语句显示为单独的行。请参阅[多行示例](#)。
- **颜色**：使用[ANSI 转义码](#)，如 `\033[32m` 表示绿色（终端必须支持它们）。请参阅[git 状态示例](#)。
- **链接**：使用[OSC 8 转义序列](#)使文本可点击（macOS 上为 Cmd+click，Windows/Linux 上为 Ctrl+click）。需要支持超链接的终端，如 iTerm2、Kitty 或 WezTerm。请参阅[可点击链接示例](#)。

Note:

状态行在本地运行，不消耗 API 令牌。在某些 UI 交互期间，它会临时隐藏，包括自动完成建议、帮助菜单和权限提示。

可用数据

Claude Code 通过 stdin 向你的脚本发送以下 JSON 字段：

字段	描述
<code>model.id</code> , <code>model.display_name</code>	当前模型标识符和显示名称
<code>cwd</code> , <code>workspace.current_dir</code>	当前工作目录。两个字段包含相同的值；为了与 <code>workspace.project_dir</code> 保持一致，首选 <code>workspace.current_dir</code> 。
<code>workspace.project_dir</code>	启动 Claude Code 的目录，如果在会话期间工作目录更改，可能与 <code>cwd</code> 不同
<code>cost.total_cost_usd</code>	总会话成本（美元）

字段	描述
<code>cost.total_duration_ms</code>	自会话开始以来的总挂钟时间（毫秒）
<code>cost.total_api_duration_ms</code>	等待 API 响应的总时间（毫秒）
<code>cost.total_lines_added</code> , <code>cost.total_lines_removed</code>	更改的代码行数
<code>context_window.total_input_tokens</code> , <code>context_window.total_output_tokens</code>	整个会话中的累积令牌计数
<code>context_window.context_window_size</code>	最大上下文窗口大小（令牌）。默认为 200000，或对于具有扩展上下文的模型为 1000000。
<code>context_window.used_percentage</code>	预计算的已使用上下文窗口百分比
<code>context_window.remaining_percentage</code>	预计算的剩余上下文窗口百分比
<code>context_window.current_usage</code>	来自最后一次 API 调用的令牌计数，在 上下文窗口字段 中描述
<code>exceeds_200k_tokens</code>	最近一次 API 响应中的总令牌计数（输入、缓存和输出令牌合并）是否超过 200k。这是一个固定阈值，与实际上下文窗口大小无关。
<code>session_id</code>	唯一的会话标识符
<code>transcript_path</code>	对话记录文件的路径
<code>version</code>	Claude Code 版本
<code>output_style.name</code>	当前输出样式的名称
<code>vim.mode</code>	启用 vim 模式 时的当前 vim 模式（ <code>NORMAL</code> 或 <code>INSERT</code> ）
<code>agent.name</code>	使用 <code>--agent</code> 标志或配置的代理设置运行时的代理名称
<code>worktree.name</code>	活跃 <code>worktree</code> 的名称。仅在 <code>--worktree</code> 会话期间出现
<code>worktree.path</code>	<code>worktree</code> 目录的绝对路径

字段	描述
<code>worktree.branch</code>	worktree 的 Git 分支名称（例如， <code>"worktree-my-feature"</code> ）。对于基于钩子的 worktree 不存在
<code>worktree.original_cwd</code>	进入 worktree 之前 Claude 所在的目录
<code>worktree.original_branch</code>	进入 worktree 之前检出的 Git 分支。对于基于钩子的 worktree 不存在

你的状态行命令通过 stdin 接收此 JSON 结构：

```

{
  "cwd": "/current/working/directory",
  "session_id": "abc123...",
  "transcript_path": "/path/to/transcript.jsonl",
  "model": {
    "id": "claude-opus-4-6",
    "display_name": "Opus"
  },
  "workspace": {
    "current_dir": "/current/working/directory",
    "project_dir": "/original/project/directory"
  },
  "version": "1.0.80",
  "output_style": {
    "name": "default"
  },
  "cost": {
    "total_cost_usd": 0.01234,
    "total_duration_ms": 45000,
    "total_api_duration_ms": 2300,
    "total_lines_added": 156,
    "total_lines_removed": 23
  },
  "context_window": {
    "total_input_tokens": 15234,
    "total_output_tokens": 4521,
    "context_window_size": 200000,
    "used_percentage": 8,
    "remaining_percentage": 92,
    "current_usage": {
      "input_tokens": 8500,
      "output_tokens": 1200,
      "cache_creation_input_tokens": 5000,
      "cache_read_input_tokens": 2000
    }
  },
  "exceeds_200k_tokens": false,
  "vim": {

```

```

    "mode": "NORMAL"
  },
  "agent": {
    "name": "security-reviewer"
  },
  "worktree": {
    "name": "my-feature",
    "path": "/path/to/.claude/worktrees/my-feature",
    "branch": "worktree-my-feature",
    "original_cwd": "/path/to/project",
    "original_branch": "main"
  }
}

```

可能不存在的字段（不在 JSON 中）：

- `vim`：仅在启用 vim 模式时出现
- `agent`：仅在使用 `--agent` 标志或配置的代理设置运行时出现
- `worktree`：仅在 `--worktree` 会话期间出现。存在时，`branch` 和 `original_branch` 对于基于钩子的 worktree 也可能不存在

可能为 `null` 的字段：

- `context_window.current_usage`：在会话中第一次 API 调用之前为 `null`
- `context_window.used_percentage`，`context_window.remaining_percentage`：在会话早期可能为 `null`

在你的脚本中使用条件访问处理缺失字段，使用回退默认值处理 `null` 值。

上下文窗口字段

`context_window` 对象提供了两种跟踪上下文使用情况的方式：

- **累积总计**（`total_input_tokens`，`total_output_tokens`）：整个会话中所有令牌的总和，用于跟踪总消耗
- **当前使用情况**（`current_usage`）：来自最近一次 API 调用的令牌计数，使用此来获得准确的上下文百分比，因为它反映了实际的上下文状态

`current_usage` 对象包含：

- `input_tokens`：当前上下文中的输入令牌
- `output_tokens`：生成的输出令牌

- `cache_creation_input_tokens`：写入缓存的令牌
- `cache_read_input_tokens`：从缓存读取的令牌

`used_percentage` 字段仅从输入令牌计算：

`input_tokens + cache_creation_input_tokens + cache_read_input_tokens`。它不包括 `output_tokens`。

如果你从 `current_usage` 手动计算上下文百分比，使用相同的仅输入公式来匹配 `used_percentage`。

`current_usage` 对象在会话中第一次 API 调用之前为 `null`。

示例

这些示例展示了常见的状态行模式。要使用任何示例：

1. 将脚本保存到文件，如 `~/.claude/statusline.sh`（或 `.py / .js`）
2. 使其可执行：`chmod +x ~/.claude/statusline.sh`
3. 将路径添加到你的[设置](#)

Bash 示例使用 `jq` 来解析 JSON。Python 和 Node.js 具有内置的 JSON 解析。

上下文窗口使用情况

显示当前模型和上下文窗口使用情况，带有可视进度条。每个脚本从 `stdin` 读取 JSON，提取 `used_percentage` 字段，并构建一个 10 字符的栏，其中填充的块 (■) 代表使用情况：



一个状态行，显示模型名称和带有百分比的进度条

```
#!/bin/bash

## Read all of stdin into a variable
input=$(cat)

## Extract fields with jq, "// 0" provides fallback for null
MODEL=$(echo "$input" | jq -r '.model.display_name')
PCT=$(echo "$input" | jq -r '.context_window.used_percentage // 0' | cut -d. -f1)

## Build progress bar: printf creates spaces, tr replaces with blocks
BAR_WIDTH=10
FILLED=$((PCT * BAR_WIDTH / 100))
EMPTY=$((BAR_WIDTH - FILLED))
BAR=""

[ "$FILLED" -gt 0 ] && BAR=$(printf "%${FILLED}s" | tr ' ' '█')
[ "$EMPTY" -gt 0 ] && BAR="${BAR}$(printf "%${EMPTY}s" | tr ' ' '░')"

echo "[${MODEL}] $BAR ${PCT%}"
```

```
#!/usr/bin/env python3
import json, sys

## json.load reads and parses stdin in one step
data = json.load(sys.stdin)
model = data['model']['display_name']
## "or 0" handles null values
pct = int(data.get('context_window', {}).get('used_percentage', 0) or 0)

## String multiplication builds the bar
filled = pct * 10 // 100
bar = '█' * filled + '░' * (10 - filled)

print(f"[{model}] {bar} {pct}%")
```

```
#!/usr/bin/env node
// Node.js reads stdin asynchronously with events
let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
  const data = JSON.parse(input);
  const model = data.model.display_name;
  // Optional chaining (?.) safely handles null fields
  const pct = Math.floor(data.context_window?.used_percentage || 0);

  // String.repeat() builds the bar
  const filled = Math.floor(pct * 10 / 100);
  const bar = '█'.repeat(filled) + '░'.repeat(10 - filled);

  console.log(`[${model}] ${bar} ${pct}%`);
});
```

Git 状态与颜色

显示 git 分支，带有暂存和修改文件的颜色编码指示器。此脚本使用[ANSI 转义码](#)表示终端颜色：`\033[32m` 是绿色，`\033[33m` 是黄色，`\033[0m` 重置为默认值。

```
> █
```

```
[Sonnet] 📁 my-app | 🌿 feature/auth +3~2
```

一个状态行，显示模型、目录、git 分支和暂存和修改文件的彩色指示器

每个脚本检查当前目录是否是 git 存储库，计算暂存和修改文件，并显示颜色编码的指示器：

```
#!/bin/bash

input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')

GREEN='\033[32m'
YELLOW='\033[33m'
RESET='\033[0m'

if git rev-parse --git-dir > /dev/null 2>&1; then
    BRANCH=$(git branch --show-current 2>/dev/null)
    STAGED=$(git diff --cached --numstat 2>/dev/null | wc -l | tr -d ' ')
    MODIFIED=$(git diff --numstat 2>/dev/null | wc -l | tr -d ' ')

    GIT_STATUS=""
    [ "$STAGED" -gt 0 ] && GIT_STATUS="${GREEN}+${STAGED}${RESET}"
    [ "$MODIFIED" -gt 0 ] && GIT_STATUS="${GIT_STATUS}${YELLOW}~${MODIFIED}${RESET}"

    echo -e "[$MODEL] 📁 ${DIR##*/} | 🌿 $BRANCH $GIT_STATUS"
else
    echo "[$MODEL] 📁 ${DIR##*/}"
fi
```



```
#!/usr/bin/env python3
import json, sys, subprocess, os

data = json.load(sys.stdin)
model = data['model']['display_name']
directory = os.path.basename(data['workspace']['current_dir'])

GREEN, YELLOW, RESET = '\033[32m', '\033[33m', '\033[0m'

try:
    subprocess.check_output(['git', 'rev-parse', '--git-dir'], stderr=subprocess.D
EVNULL)
    branch = subprocess.check_output(['git', 'branch', '--show-current'], text=Tru
e).strip()
    staged_output = subprocess.check_output(['git', 'diff', '--cached', '--
numstat'], text=True).strip()
    modified_output = subprocess.check_output(['git', 'diff', '--numstat'], text=T
rue).strip()
    staged = len(staged_output.split('\n')) if staged_output else 0
    modified = len(modified_output.split('\n')) if modified_output else 0

    git_status = f"{GREEN}+{staged}{RESET}" if staged else ""
    git_status += f"{YELLOW}~{modified}{RESET}" if modified else ""

    print(f"[{model}] 📁 {directory} | 🌿 {branch} {git_status}")
except:
    print(f"[{model}] 📁 {directory}")
```

```
#!/usr/bin/env node

const { execSync } = require('child_process');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
  const data = JSON.parse(input);
  const model = data.model.display_name;
  const dir = path.basename(data.workspace.current_dir);

  const GREEN = '\x1b[32m', YELLOW = '\x1b[33m', RESET = '\x1b[0m';

  try {
    execSync('git rev-parse --git-dir', { stdio: 'ignore' });
    const branch = execSync('git branch --show-current', { encoding:
'utf8' }).trim();
    const staged = execSync('git diff --cached --numstat', { encoding: 'utf8'
}).trim().split('\n').filter(Boolean).length;
    const modified = execSync('git diff --numstat', { encoding:
'utf8' }).trim().split('\n').filter(Boolean).length;

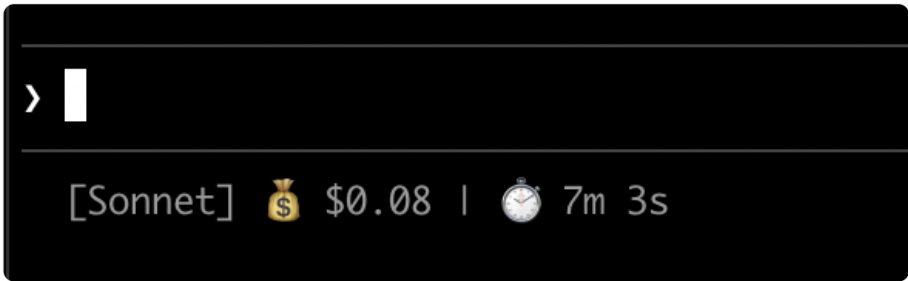
    let gitStatus = staged ? `${GREEN}+${staged}${RESET}` : '';
    gitStatus += modified ? `${YELLOW}~${modified}${RESET}` : '';

    console.log(`[${model}] 📁 ${dir} | 🌿 ${branch} ${gitStatus}`);
  } catch {
    console.log(`[${model}] 📁 ${dir}`);
  }
});
```

成本和持续时间跟踪

跟踪你的会话的 API 成本和经过的时间。`cost.total\_cost\_usd` 字段累积当前会话中所有 API 调用的成本。`cost.total\_duration\_ms` 字段测量自会话开始以来的总经过时间，而 `cost.total\_api\_duration\_ms` 仅跟踪等待 API 响应的时间。

每个脚本将成本格式化为货币，并将毫秒转换为分钟和秒：



一个状态行，显示模型名称、会话成本和持续时间

```
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
COST=$(echo "$input" | jq -r '.cost.total_cost_usd // 0')
DURATION_MS=$(echo "$input" | jq -r '.cost.total_duration_ms // 0')

COST_FMT=$(printf '%.2f' "$COST")
DURATION_SEC=$((DURATION_MS / 1000))
MINS=$((DURATION_SEC / 60))
SECS=$((DURATION_SEC % 60))

echo "[${MODEL}] 💰 ${COST_FMT} | 🕒 ${MINS}m ${SECS}s"
```

```
#!/usr/bin/env python3
import json, sys

data = json.load(sys.stdin)
model = data['model']['display_name']
cost = data.get('cost', {}).get('total_cost_usd', 0) or 0
duration_ms = data.get('cost', {}).get('total_duration_ms', 0) or 0

duration_sec = duration_ms // 1000
mins, secs = duration_sec // 60, duration_sec % 60

print(f"[{model}] 💰 ${cost:.2f} | 🕒 {mins}m {secs}s")
```

```
#!/usr/bin/env node
let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
  const data = JSON.parse(input);
  const model = data.model.display_name;
  const cost = data.cost?.total_cost_usd || 0;
  const durationMs = data.cost?.total_duration_ms || 0;

  const durationSec = Math.floor(durationMs / 1000);
  const mins = Math.floor(durationSec / 60);
  const secs = durationSec % 60;

  console.log(`[${model}] 💰 $$${cost.toFixed(2)} | ⌚ ${mins}m ${secs}s`);
});
```

显示多行

你的脚本可以输出多行来创建更丰富的显示。每个 `echo` 语句在状态区域中产生单独的行。

```
> |
[Opus] 📁 my-app | 🌿 feature/auth
██████████ 42% | 💰 $0.08 | ⌚ 7m 3s
```

一个多行状态行，显示第一行上的模型名称、目录、git 分支，第二行上的上下文使用进度条、成本和持续时间

此示例结合了几种技术：基于阈值的颜色（70% 以下为绿色，70-89% 为黄色，90%+ 为红色）、进度条和 git 分支信息。每个 `print` 或 `echo` 语句创建单独的行：

```
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')
COST=$(echo "$input" | jq -r '.cost.total_cost_usd // 0')
PCT=$(echo "$input" | jq -r '.context_window.used_percentage // 0' | cut -d. -f1)
DURATION_MS=$(echo "$input" | jq -r '.cost.total_duration_ms // 0')

CYAN='\033[36m'; GREEN='\033[32m'; YELLOW='\033[33m'; RED='\033[31m'; RESET='\033[
0m'

## Pick bar color based on context usage
if [ "$PCT" -ge 90 ]; then BAR_COLOR="$RED"
elif [ "$PCT" -ge 70 ]; then BAR_COLOR="$YELLOW"
else BAR_COLOR="$GREEN"; fi

FILLED=$((PCT / 10)); EMPTY=$((10 - FILLED))
BAR=$(printf "%${FILLED}s" | tr ' ' '█')$(printf "%${EMPTY}s" | tr ' ' '░')

MINS=$((DURATION_MS / 60000)); SECS=$((DURATION_MS % 60000) / 1000)

BRANCH=""
git rev-parse --git-dir > /dev/null 2>&1 && BRANCH=" | 🌿 $(git branch --show-
current 2>/dev/null)"

echo -e "${CYAN}[$MODEL]${RESET} 📁 ${DIR##*/}${BRANCH}"
COST_FMT=$(printf '%.2f' "$COST")
echo -e "${BAR_COLOR}${BAR}${RESET} ${PCT}% | ${YELLOW}${COST_FMT}${RESET} | ⌚ $
{MINS}m ${SECS}s"
```

```
#!/usr/bin/env python3
import json, sys, subprocess, os

data = json.load(sys.stdin)
model = data['model']['display_name']
directory = os.path.basename(data['workspace']['current_dir'])
cost = data.get('cost', {}).get('total_cost_usd', 0) or 0
pct = int(data.get('context_window', {}).get('used_percentage', 0) or 0)
duration_ms = data.get('cost', {}).get('total_duration_ms', 0) or 0

CYAN, GREEN, YELLOW, RED, RESET = '\033[36m', '\033[32m', '\033[33m', '\033[31m',
'\033[0m'

bar_color = RED if pct ≥ 90 else YELLOW if pct ≥ 70 else GREEN
filled = pct // 10
bar = '█' * filled + '░' * (10 - filled)

mins, secs = duration_ms // 60000, (duration_ms % 60000) // 1000

try:
    branch = subprocess.check_output(['git', 'branch', '--show-current'], text=True,
stderr=subprocess.DEVNULL).strip()
    branch = f" | 🌿 {branch}" if branch else ""
except:
    branch = ""

print(f"{CYAN}[{model}]{RESET} 📁 {directory}{branch}")
print(f"{bar_color}{bar}{RESET} {pct}% | {YELLOW}${cost:.2f}{RESET} | ⌚ {mins}m
{secs}s")
```

```
#!/usr/bin/env node
const { execSync } = require('child_process');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
  const data = JSON.parse(input);
  const model = data.model.display_name;
  const dir = path.basename(data.workspace.current_dir);
  const cost = data.cost?.total_cost_usd || 0;
  const pct = Math.floor(data.context_window?.used_percentage || 0);
  const durationMs = data.cost?.total_duration_ms || 0;

  const CYAN = '\x1b[36m', GREEN = '\x1b[32m', YELLOW = '\x1b[33m', RED = '\x1b[31m', RESET = '\x1b[0m';

  const barColor = pct >= 90 ? RED : pct >= 70 ? YELLOW : GREEN;
  const filled = Math.floor(pct / 10);
  const bar = '█'.repeat(filled) + '░'.repeat(10 - filled);

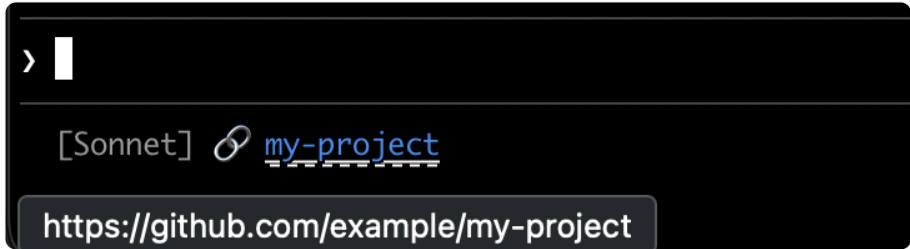
  const mins = Math.floor(durationMs / 60000);
  const secs = Math.floor((durationMs % 60000) / 1000);

  let branch = '';
  try {
    branch = execSync('git branch --show-current', { encoding: 'utf8', stdio:
['pipe', 'pipe', 'ignore'] }).trim();
    branch = branch ? ` | 🌿 ${branch}` : '';
  } catch {}

  console.log(`${CYAN}[${model}]${RESET} 📁 ${dir}${branch}`);
  console.log(`${barColor}${bar}${RESET} ${pct}% | ${YELLOW}$$${cost.toFixed(2)}${RESET} | 🕒 ${mins}m ${secs}s`);
});
```

可点击链接

此示例创建指向你的 GitHub 存储库的可点击链接。它读取 git 远程 URL，使用 `sed` 将 SSH 格式转换为 HTTPS，并将存储库名称包装在 OSC 8 转义码中。按住 Cmd (macOS) 或 Ctrl (Windows/Linux) 并单击以在浏览器中打开链接。



一个状态行，显示指向 GitHub 存储库的可点击链接

每个脚本获取 git 远程 URL，将 SSH 格式转换为 HTTPS，并将存储库名称包装在 OSC 8 转义码中。Bash 版本使用 `printf '%b'`，它比 `echo -e` 更可靠地跨不同 shell 解释反斜杠转义：

```
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')

## Convert git SSH URL to HTTPS
REMOTE=$(git remote get-url origin 2>/dev/null | sed 's/git@github.com:/https:\/\/github.com\/' | sed 's\/.git$\/')

if [ -n "$REMOTE" ]; then
    REPO_NAME=$(basename "$REMOTE")
    # OSC 8 format: \e]8;;URL\a then TEXT then \e]8;;\a
    # printf %b interprets escape sequences reliably across shells
    printf '%b' "[$MODEL] 
```



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

data = json.load(sys.stdin)
model = data['model']['display_name']

## Get git remote URL
try:
    remote = subprocess.check_output(
        ['git', 'remote', 'get-url', 'origin'],
        stderr=subprocess.DEVNULL, text=True
    ).strip()
    # Convert SSH to HTTPS format
    remote = re.sub(r'^git@github\.com:', 'https://github.com/', remote)
    remote = re.sub(r'\.git$', '', remote)
    repo_name = os.path.basename(remote)
    # OSC 8 escape sequences
    link = f"\033]8;;{remote}\a{repo_name}\033]8;;\a"
    print(f"[{model}] link")
except:
    print(f"[{model}]")
```

```
#!/usr/bin/env node
const { execSync } = require('child_process');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
  const data = JSON.parse(input);
  const model = data.model.display_name;

  try {
    let remote = execSync('git remote get-url origin', { encoding: 'utf8', stdio: ['pipe', 'pipe', 'ignore'] }).trim();
    // Convert SSH to HTTPS format
    remote = remote.replace(/^git@github\.com:/, 'https://github.com/').replace(/\.git$/, '');
    const repoName = path.basename(remote);
    // OSC 8 escape sequences
    const link = `\x1b]8;;${remote}\x07${repoName}\x1b]8;;\x07`;
    console.log(`[${model}] 🔗 \${link}`);
  } catch {
    console.log(`[${model}]`);
  }
});
```

缓存昂贵的操作

你的状态行脚本在活跃会话期间频繁运行。像 `git status` 或 `git diff` 这样的命令可能很慢，特别是在大型存储库中。此示例将 git 信息缓存到临时文件，并仅每 5 秒刷新一次。

为缓存文件使用稳定的固定文件名，如 `/tmp/statusline-git-cache`。每个状态行调用作为新进程运行，所以像 `$$`、`os.getpid()` 或 `process.pid` 这样的基于进程的标识符每次都产生不同的值，缓存永远不会被重用。

每个脚本在运行 git 命令之前检查缓存文件是否缺失或早于 5 秒：

```
#!/bin/bash

input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')

CACHE_FILE="/tmp/statusline-git-cache"
CACHE_MAX_AGE=5 # seconds

cache_is_stale() {
    [ ! -f "$CACHE_FILE" ] || \
    # stat -f %m is macOS, stat -c %Y is Linux
    [ $((($(date +%s) - $(stat -f %m "$CACHE_FILE" 2>/dev/null || stat -c %Y "$CACHE_FILE" 2>/dev/null || echo 0))) -gt $CACHE_MAX_AGE ]
}

if cache_is_stale; then
    if git rev-parse --git-dir > /dev/null 2>&1; then
        BRANCH=$(git branch --show-current 2>/dev/null)
        STAGED=$(git diff --cached --numstat 2>/dev/null | wc -l | tr -d ' ')
        MODIFIED=$(git diff --numstat 2>/dev/null | wc -l | tr -d ' ')
        echo "$BRANCH|$STAGED|$MODIFIED" > "$CACHE_FILE"
    else
        echo "||" > "$CACHE_FILE"
    fi
fi

IFS='|' read -r BRANCH STAGED MODIFIED < "$CACHE_FILE"

if [ -n "$BRANCH" ]; then
    echo "[ $MODEL ] 📁 ${DIR##*/} | 🌿 $BRANCH +$STAGED ~$MODIFIED"
else
    echo "[ $MODEL ] 📁 ${DIR##*/}"
fi
```

```
#!/usr/bin/env python3
import json, sys, subprocess, os, time

data = json.load(sys.stdin)
model = data['model']['display_name']
directory = os.path.basename(data['workspace']['current_dir'])

CACHE_FILE = "/tmp/statusline-git-cache"
CACHE_MAX_AGE = 5 # seconds

def cache_is_stale():
    if not os.path.exists(CACHE_FILE):
        return True
    return time.time() - os.path.getmtime(CACHE_FILE) > CACHE_MAX_AGE

if cache_is_stale():
    try:
        subprocess.check_output(['git', 'rev-parse', '--git-dir'], stderr=subprocess.DEVNULL)
        branch = subprocess.check_output(['git', 'branch', '--show-current'],
text=True).strip()
        staged = subprocess.check_output(['git', 'diff', '--cached', '--numstat'],
text=True).strip()
        modified = subprocess.check_output(['git', 'diff', '--numstat'],
text=True).strip()
        staged_count = len(staged.split('\n')) if staged else 0
        modified_count = len(modified.split('\n')) if modified else 0
        with open(CACHE_FILE, 'w') as f:
            f.write(f"{branch}|{staged_count}|{modified_count}")
    except:
        with open(CACHE_FILE, 'w') as f:
            f.write("||")

with open(CACHE_FILE) as f:
    branch, staged, modified = f.read().strip().split('|')

if branch:
```

```
print(f"[{model}] 📁 {directory} | 🌿 {branch} +{staged} ~{modified}")  
  
else:  
    print(f"[{model}] 📁 {directory}")
```

```
#!/usr/bin/env node
const { execSync } = require('child_process');
const fs = require('fs');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
  const data = JSON.parse(input);
  const model = data.model.display_name;
  const dir = path.basename(data.workspace.current_dir);

  const CACHE_FILE = '/tmp/statusline-git-cache';
  const CACHE_MAX_AGE = 5; // seconds

  const cacheIsStale = () => {
    if (!fs.existsSync(CACHE_FILE)) return true;
    return (Date.now() / 1000) - fs.statSync(CACHE_FILE).mtimeMs / 1000 >
    CACHE_MAX_AGE;
  };

  if (cacheIsStale()) {
    try {
      execSync('git rev-parse --git-dir', { stdio: 'ignore' });
      const branch = execSync('git branch --show-current', { encoding: 'utf8'
' }).trim();
      const staged = execSync('git diff --cached --numstat', { encoding: 'ut
f8' }).trim().split('\n').filter(Boolean).length;
      const modified = execSync('git diff --numstat', { encoding:
'utf8' }).trim().split('\n').filter(Boolean).length;
      fs.writeFileSync(CACHE_FILE, `${branch}|${staged}|${modified}`);
    } catch {
      fs.writeFileSync(CACHE_FILE, '||');
    }
  }

  const [branch, staged, modified] = fs.readFileSync(CACHE_FILE,
'utf8').trim().split('|');
```

```

    if (branch) {
        console.log(`[${model}] 📁 ${dir} | 🌿 ${branch} +${staged} ~${modified}
`);
    } else {
        console.log(`[${model}] 📁 ${dir}`);
    }
}
});

```

Windows 配置

在 Windows 上，Claude Code 通过 Git Bash 运行状态行命令。你可以从该 shell 调用 PowerShell：

```

{
  "statusLine": {
    "type": "command",
    "command": "powershell -NoProfile -File C:/Users/username/.claude/
statusline.ps1"
  }
}

```

```

$input_json = $input | Out-String | ConvertFrom-Json
$cwd = $input_json.cwd
$model = $input_json.model.display_name
$used = $input_json.context_window.used_percentage
$dirname = Split-Path $cwd -Leaf

if ($used) {
    Write-Host "$dirname [$model] ctx: $used%"
} else {
    Write-Host "$dirname [$model]"
}

```

或直接运行 Bash 脚本：

```
{
  "statusLine": {
    "type": "command",
    "command": "~/claude/statusline.sh"
  }
}
```

```
#!/usr/bin/env bash
input=$(cat)
cwd=$(echo "$input" | grep -o '"cwd": "[^"]*"' | cut -d'"' -f4)
model=$(echo "$input" | grep -o '"display_name": "[^"]*"' | cut -d'"' -f4)
dirname="${cwd##*/}"
echo "$dirname [$model]"
```

提示

- **使用模拟输入测试：** `echo '{"model":{"display_name":"Opus"},"context_window":{"used_percentage":25}}' | ./statusline.sh`
- **保持输出简短：** 状态栏的宽度有限，所以长输出可能会被截断或换行不当
- **缓存慢速操作：** 你的脚本在活跃会话期间频繁运行，所以像 `git status` 这样的命令可能会导致延迟。请参阅[缓存示例](#)以了解如何处理此问题。

社区项目如 [ccstatusline](#) 和 [starship-claude](#) 提供带有主题和其他功能的预构建配置。

故障排除

状态行未出现

- 验证你的脚本是可执行的： `chmod +x ~/.claude/statusline.sh`
- 检查你的脚本输出到 `stdout`，而不是 `stderr`
- 手动运行你的脚本以验证它产生输出
- 如果 `disableAllHooks` 在你的设置中设置为 `true`，状态行也会被禁用。删除此设置或将其设置为 `false` 以重新启用。
- 运行 `claude --debug` 以记录会话中第一个状态行调用的退出代码和 `stderr`
- 要求 Claude 读取你的设置文件并直接执行 `statusLine` 命令以显示错误

状态行显示 `--` 或空值

- 在第一个 API 响应完成之前，字段可能为 `null`
- 在你的脚本中使用回退处理 `null` 值，如 `jq` 中的 `// 0`
- 如果多条消息后值仍为空，请重新启动 Claude Code

上下文百分比显示意外值

- 使用 `used_percentage` 获得准确的上下文状态，而不是累积总计
- `total_input_tokens` 和 `total_output_tokens` 在整个会话中累积，可能超过上下文窗口大小
- 上下文百分比可能与 `/context` 输出不同，因为每个的计算时间不同

OSC 8 链接不可点击

- 验证你的终端支持 OSC 8 超链接（iTerm2、Kitty、WezTerm）
- Terminal.app 不支持可点击链接
- SSH 和 `tmux` 会话可能根据配置剥离 OSC 序列
- 如果转义序列显示为文字文本，如 `\e]8;;`，使用 `printf '%b'` 而不是 `echo -e` 以获得更可靠的转义处理

转义序列显示故障

- 复杂的转义序列（ANSI 颜色、OSC 8 链接）如果与其他 UI 更新重叠，偶尔会导致输出混乱
- 如果你看到损坏的文本，尝试简化你的脚本为纯文本输出
- 带有转义码的多行状态行比单行纯文本更容易出现渲染问题

脚本错误或挂起

- 以非零代码退出或不产生输出的脚本会导致状态行变为空白
- 慢速脚本会阻止状态行更新，直到它们完成。保持脚本快速以避免陈旧输出。
- 如果在慢速脚本运行时触发新的更新，正在进行的脚本会被取消
- 在配置之前使用模拟输入独立测试你的脚本

通知共享状态行行

- 系统通知，如 MCP 服务器错误、自动更新和令牌警告，显示在与你的状态行相同行的右侧
- 启用详细模式会向此区域添加令牌计数器
- 在窄终端上，这些通知可能会截断你的状态行输出

Part 13: Troubleshooting & Changelog

故障排除

发现 Claude Code 安装和使用中常见问题的解决方案。

故障排除安装问题

Tip:

如果您想完全跳过终端，[Claude Code 桌面应用](#)让您可以通过图形界面安装和使用 Claude Code。下载 [macOS](#) 或 [Windows](#) 版本，无需任何命令行设置即可开始编码。

找到您看到的错误消息或症状：

您看到的内容	解决方案
<code>command not found: claude</code> 或 <code>'claude' is not recognized</code>	修复您的 PATH
<code>syntax error near unexpected token '<'</code>	安装脚本返回 HTML
<code>curl: (56) Failure writing output to destination</code>	先下载脚本，然后运行
Linux 上安装期间 <code>Killed</code>	为低内存服务器添加交换空间
<code>TLS connect error</code> 或 <code>SSL/TLS secure channel</code>	更新 CA 证书
<code>Failed to fetch version</code> 或无法访问下载服务器	检查网络和代理设置
<code>irm is not recognized</code> 或 <code>&& is not valid</code>	为您的 shell 使用正确的命令
<code>Claude Code on Windows requires git-bash</code>	安装或配置 Git Bash

您看到的内容	解决方案
Error loading shared library	您的系统安装了错误的二进制变体
Linux 上的 Illegal instruction	架构不匹配
macOS 上的 dyld: cannot load 或 Abort trap	二进制不兼容
Invoke-Expression: Missing argument in parameter list	安装脚本返回 HTML
App unavailable in region	Claude Code 在您的国家/地区不可用。请参阅 支持的国家和地区 。
unable to get local issuer certificate	配置企业 CA 证书
OAuth error 或 403 Forbidden	修复身份验证

如果您的问题未列出，请按照这些诊断步骤进行操作。

调试安装问题

检查网络连接

安装程序从 `storage.googleapis.com` 下载。验证您可以访问它：

```
curl -sI https://storage.googleapis.com
```

如果失败，您的网络可能阻止了连接。常见原因：

- 企业防火墙或代理阻止 Google Cloud Storage
- 区域网络限制：尝试使用 VPN 或替代网络
- TLS/SSL 问题：更新您系统的 CA 证书，或检查是否配置了 `HTTPS_PROXY`

如果您在企业代理后面，在安装前设置 `HTTPS_PROXY` 和 `HTTP_PROXY` 为您的代理地址。如果您不知道代理 URL，请向您的 IT 团队询问，或检查您的浏览器代理设置。

此示例设置两个代理变量，然后通过您的代理运行安装程序：

```
export HTTP_PROXY=http://proxy.example.com:8080
export HTTPS_PROXY=http://proxy.example.com:8080
curl -fsSL https://claude.ai/install.sh | bash
```

验证您的 PATH

如果安装成功但运行 `claude` 时出现 `command not found` 或 `not recognized` 错误，则安装目录不在您的 PATH 中。您的 shell 在 PATH 中列出的目录中搜索程序，安装程序在 macOS/Linux 上将 `claude` 放在 `~/.local/bin/claude`，或在 Windows 上放在 `%USERPROFILE%\local\bin\claude.exe`。

通过列出您的 PATH 条目并过滤 `local/bin` 来检查安装目录是否在您的 PATH 中：

macOS/Linux

```
echo $PATH | tr ':' '\n' | grep local/bin
```

如果没有输出，则目录缺失。将其添加到您的 shell 配置：

```
## Zsh (macOS 默认)
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc

## Bash (Linux 默认)
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

或者，关闭并重新打开您的终端。

验证修复是否有效：

```
claude --version
```

Windows PowerShell

```
$env:PATH -split ';' | Select-String 'local\\bin'
```

如果没有输出，将安装目录添加到您的用户 PATH：

```
$currentPath = [Environment]::GetEnvironmentVariable('PATH', 'User')  
[Environment]::SetEnvironmentVariable('PATH', "$currentPath;$env:USERPROFILE\.local\bin", 'User')
```

重启您的终端以使更改生效。

验证修复是否有效：

```
claude --version
```

Windows CMD

```
echo %PATH% | findstr /i "local\bin"
```

如果没有输出，打开系统设置，转到环境变量，并将 `%USERPROFILE%\local\bin` 添加到您的用户 PATH 变量。重启您的终端。

验证修复是否有效：

```
claude --version
```

检查冲突的安装

多个 Claude Code 安装可能导致版本不匹配或意外行为。检查已安装的内容：

macOS/Linux

列出在您的 PATH 中找到的所有 `claude` 二进制文件：

```
which -a claude
```

检查是否存在本机安装程序和 npm 版本：

```
ls -la ~/.local/bin/claude
```

```
ls -la ~/.claude/local/
```

```
npm -g ls @anthropic-ai/claude-code 2>/dev/null
```

Windows PowerShell

```
where.exe claude  
Test-Path "$env:LOCALAPPDATA\Claude Code\claude.exe"
```

如果您找到多个安装，只保留一个。建议使用 `~/.local/bin/claude` 处的本机安装。删除任何额外的安装：

卸载 npm 全局安装：

```
npm uninstall -g @anthropic-ai/claude-code
```

在 macOS 上删除 Homebrew 安装：

```
brew uninstall --cask claude-code
```

检查目录权限

安装程序需要对 `~/.local/bin/` 和 `~/.claude/` 的写入权限。如果安装失败并出现权限错误，请检查这些目录是否可写：

```
test -w ~/.local/bin && echo "writable" || echo "not writable"  
test -w ~/.claude && echo "writable" || echo "not writable"
```

如果任一目录不可写，创建安装目录并将您的用户设置为所有者：

```
sudo mkdir -p ~/.local/bin  
sudo chown -R $(whoami) ~/.local
```

验证二进制文件是否有效

如果 `claude` 已安装但在启动时崩溃或挂起，运行这些检查以缩小原因范围。

确认二进制文件存在且可执行：

```
ls -la $(which claude)
```

在 Linux 上，检查缺失的共享库。如果 `ldd` 显示缺失的库，您可能需要安装系统包。在 Alpine Linux 和其他基于 musl 的发行版上，请参阅 [Alpine Linux 设置](#)。

```
ldd $(which claude) | grep "not found"
```

运行快速健全性检查以确认二进制文件可以执行：

```
claude --version
```

常见安装问题

这些是最常见的安装问题及其解决方案。

安装脚本返回 HTML 而不是 shell 脚本

运行安装命令时，您可能会看到以下错误之一：

```
bash: line 1: syntax error near unexpected token `<'  
bash: line 1: `<!DOCTYPE html>'
```

在 PowerShell 上，同样的问题显示为：

```
Invoke-Expression: Missing argument in parameter list.
```

这意味着安装 URL 返回了 HTML 页面而不是安装脚本。如果 HTML 页面显示“App unavailable in region”，Claude Code 在您的国家/地区不可用。请参阅[支持的国家/地区](#)。

否则，这可能由网络问题、区域路由或临时服务中断引起。

解决方案：

1. 使用替代安装方法：

在 macOS 或 Linux 上，通过 Homebrew 安装：

```
brew install --cask claude-code
```

在 Windows 上，通过 WinGet 安装：

```
winget install Anthropic.ClaudeCode
```

1. 几分钟后重试：问题通常是临时的。等待并重试原始命令。

安装后 `command not found: claude`

安装完成但 `claude` 不起作用。确切的错误因平台而异：

平台	错误消息
macOS	<code>zsh: command not found: claude</code>
Linux	<code>bash: claude: command not found</code>
Windows CMD	<code>'claude' is not recognized as an internal or external command</code>
PowerShell	<code>claude : The term 'claude' is not recognized as the name of a cmdlet</code>

这意味着安装目录不在您的 shell 搜索路径中。请参阅[验证您的 PATH](#) 以获取每个平台上的修复。

`curl: (56) Failure writing output to destination`

`curl ... | bash` 命令下载脚本并使用管道 (`|`) 将其直接传递给 Bash 执行。此错误意味着连接在脚本完成下载前中断。常见原因包括网络中断、下载被中途阻止或系统资源限制。

解决方案：

1. 检查网络稳定性： Claude Code 二进制文件托管在 Google Cloud Storage 上。测试您是否可以访问它：

```
curl -fsSL https://storage.googleapis.com -o /dev/null
```


如果命令无声地完成，您的连接良好，问题可能是间歇性的。重试安装命令。如果您看到错误，您的网络可能阻止了下载。

1. 尝试替代安装方法：

在 macOS 或 Linux 上：

```
brew install --cask claude-code
```

在 Windows 上：

```
winget install Anthropic.ClaudeCode
```

TLS 或 SSL 连接错误

诸如 `curl: (35) TLS connect error`、`schannel: next InitializeSecurityContext failed` 或 PowerShell 的 `Could not establish trust relationship for the SSL/TLS secure channel` 之类的错误表示 TLS 握手失败。

解决方案：

1. 更新您的系统 CA 证书：

在 Ubuntu/Debian 上：

```
sudo apt-get update && sudo apt-get install ca-certificates
```

在 macOS 上通过 Homebrew：

```
brew install ca-certificates
```

1. 在 Windows 上，在运行安装程序前在 PowerShell 中启用 TLS 1.2：

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12  
irm https://claude.ai/install.ps1 | iex
```

1. **检查代理或防火墙干扰：**执行 TLS 检查的企业代理可能导致这些错误，包括 `unable to get local issuer certificate`。将 `NODE_EXTRA_CA_CERTS` 设置为您的企业 CA 证书包：

```
export NODE_EXTRA_CA_CERTS=/path/to/corporate-ca.pem
```

如果您没有证书文件，请向您的 IT 团队询问。您也可以尝试直接连接以确认代理是原因。

Failed to fetch version from storage.googleapis.com

安装程序无法访问下载服务器。这通常意味着 `storage.googleapis.com` 在您的网络上被阻止。

解决方案：

1. 直接测试连接：

```
curl -sI https://storage.googleapis.com
```

1. 如果在代理后面，设置 `HTTPS_PROXY` 以便安装程序可以通过它路由。有关详细信息，请参阅[代理配置](#)。

```
export HTTPS_PROXY=http://proxy.example.com:8080
curl -fsSL https://claude.ai/install.sh | bash
```

1. 如果在受限网络上，尝试不同的网络或 VPN，或使用替代安装方法：
在 macOS 或 Linux 上：

```
brew install --cask claude-code
```

在 Windows 上：

```
winget install Anthropic.ClaudeCode
```

Windows：`irm` 或 `&&` 未被识别

如果您看到 `'irm' is not recognized` 或 `The token '&&' is not valid`，您在错误的 shell 中运行命令。

- **`irm` 未被识别：**您在 CMD 中，而不是 PowerShell。您有两个选项：
通过在开始菜单中搜索“PowerShell”打开 PowerShell，然后运行原始安装命令：

```
irm https://claude.ai/install.ps1 | iex
```

或留在 CMD 中并改用 CMD 安装程序：

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del  
install.cmd
```

- **&& 无效：**您在 PowerShell 中但运行了 CMD 安装程序命令。使用 PowerShell 安装程序：

```
irm https://claude.ai/install.ps1 | iex
```

低内存 Linux 服务器上安装被杀死

如果在 VPS 或云实例上安装期间看到 **Killed**：

```
Setting up Claude Code...  
Installing Claude Code native build latest...  
bash: line 142: 34803 Killed    "$binary_path" install ${TARGET:+"$TARGET"}
```

Linux OOM 杀手终止了该进程，因为系统内存不足。Claude Code 需要至少 4 GB 的可用 RAM。

解决方案：

1. **添加交换空间**（如果您的服务器 RAM 有限）。交换使用磁盘空间作为溢出内存，让安装即使在低物理 RAM 的情况下也能完成。

创建 2 GB 交换文件并启用它：

```
sudo fallocate -l 2G /swapfile  
sudo chmod 600 /swapfile  
sudo mkswap /swapfile  
sudo swapon /swapfile
```

然后重试安装：

```
curl -fsSL https://claude.ai/install.sh | bash
```

1. 关闭其他进程以在安装前释放内存。
2. 使用更大的实例（如果可能）。Claude Code 需要至少 4 GB 的 RAM。

Docker 中安装挂起

在 Docker 容器中安装 Claude Code 时，以 root 身份安装到 `/` 可能导致挂起。

解决方案：

1. 在运行安装程序前设置工作目录。从 `/` 运行时，安装程序扫描整个文件系统，导致过度的内存使用。设置 `WORKDIR` 将扫描限制在小目录：

```
WORKDIR /tmp
RUN curl -fsSL https://claude.ai/install.sh | bash
```

1. 增加 Docker 内存限制（如果使用 Docker Desktop）：

```
docker build --memory=4g .
```

Windows：Claude Desktop 覆盖 `claude` CLI 命令

如果您安装了较旧版本的 Claude Desktop，它可能在 `WindowsApps` 目录中注册一个 `Claude.exe`，其 PATH 优先级高于 Claude Code CLI。运行 `claude` 会打开桌面应用而不是 CLI。

更新 Claude Desktop 到最新版本以修复此问题。

Windows：“Claude Code on Windows requires git-bash”

Windows 上的 Claude Code 需要 [Git for Windows](#)，其中包括 Git Bash。

如果未安装 Git，从 git-scm.com/downloads/win 下载并安装。在设置期间，选择”Add to PATH”。安装后重启您的终端。

如果已安装 Git 但 Claude Code 仍然找不到它，在您的 `settings.json` 文件中设置路径：

```
{
  "env": {
    "CLAUDE_CODE_GIT_BASH_PATH": "C:\\Program Files\\Git\\bin\\bash.exe"
  }
}
```

如果您的 Git 安装在其他地方，通过在 PowerShell 中运行 `where.exe git` 找到路径，并使用该目录中的 `bin\\bash.exe` 路径。

Linux：安装了错误的二进制变体（musl/glibc 不匹配）

如果在安装后看到关于缺失共享库的错误，如 `libstdc++.so.6` 或 `libgcc_s.so.1`，安装程序可能为您的系统下载了错误的二进制变体。

```
Error loading shared library libstdc++.so.6: No such file or directory
```

这可能发生在安装了 musl 交叉编译包的基于 glibc 的系统上，导致安装程序将系统误检测为 musl。

解决方案：

1. 检查您的系统使用哪个 libc：

```
ldd /bin/ls | head -1
```

如果显示 `linux-vdso.so` 或对 `/lib/x86_64-linux-gnu/` 的引用，您在 glibc 上。如果显示 `musl`，您在 musl 上。

1. 如果您在 glibc 上但获得了 musl 二进制文件，删除安装并重新安装。您也可以从 GCS 存储桶 <https://storage.googleapis.com/claude-code-dist-86c565f3-f756-42ad-8dfa-d59b1c096819/claude-code-releases/{VERSION}/manifest.json> 手动下载正确的二进制文件。使用 `ldd /bin/ls` 和 `ls /lib/libc.musl*` 的输出提交 [GitHub 问题](#)。

2. 如果您实际上在 musl 上（Alpine Linux），安装所需的包：

```
apk add libgcc libstdc++ ripgrep
```

Linux 上的 `Illegal instruction`

如果安装程序打印 `Illegal instruction` 而不是 `OOM Killed` 消息，下载的二进制文件与您的 CPU 架构不匹配。这通常发生在接收 x86 二进制文件的 ARM 服务器上，或在缺少所需指令集的较旧 CPU 上。

```
bash: line 142: 2238232 Illegal instruction   "$binary_path" install ${TARGET:+"$TARGET"}
```

解决方案：

1. 验证您的架构：

```
uname -m
```

`x86_64` 表示 64 位 Intel/AMD，`aarch64` 表示 ARM64。如果二进制文件不匹配，请使用输出 [提交 GitHub 问题](#)。

1. 尝试替代安装方法，同时解决架构问题：

```
brew install --cask claude-code
```

macOS 上的 `dyld: cannot load`

如果在安装期间看到 `dyld: cannot load` 或 `Abort trap: 6`，二进制文件与您的 macOS 版本或硬件不兼容。

```
dyld: cannot load 'claude-2.1.42-darwin-x64' (load command 0x80000034 is unknown)
Abort trap: 6
```

解决方案：

1. **检查您的 macOS 版本：** Claude Code 需要 macOS 13.0 或更高版本。打开 Apple 菜单并选择“关于本 Mac”以检查您的版本。
2. **更新 macOS**（如果您在较旧版本上）。二进制文件使用较旧 macOS 版本不支持的加载命令。
3. **尝试 Homebrew** 作为替代安装方法：

```
brew install --cask claude-code
```

Windows 安装问题：WSL 中的错误

您可能在 WSL 中遇到以下问题：

OS/平台检测问题：如果在安装期间收到错误，WSL 可能使用 Windows `npm`。尝试：

- 在安装前运行 `npm config set os linux`
- 使用 `npm install -g @anthropic-ai/claude-code --force --no-os-check` 安装。不要使用 `sudo`。

Node 未找到错误：如果运行 `claude` 时看到 `exec: node: not found`，您的 WSL 环境可能使用 Windows 安装的 Node.js。您可以使用 `which npm` 和 `which node` 确认这一点，它们应该指向以 `/usr/` 开头的 Linux 路径，而不是 `/mnt/c/`。要修复此问题，请尝试通过您的 Linux 发行版的包管理器或通过 `nvm` 安装 Node。

nvm 版本冲突：如果您在 WSL 和 Windows 中都安装了 `nvm`，在 WSL 中切换 Node 版本时可能会遇到版本冲突。这是因为 WSL 默认导入 Windows PATH，导致 Windows `nvm/``npm` 优先于 WSL 安装。

您可以通过以下方式识别此问题：

- 运行 `which npm` 和 `which node` - 如果它们指向 Windows 路径（以 `/mnt/c/` 开头），则使用 Windows 版本
- 在 WSL 中使用 `nvm` 切换 Node 版本后遇到功能损坏

要解决此问题，修复您的 Linux PATH 以确保 Linux `node/npm` 版本优先：

主要解决方案：确保 `nvm` 在您的 shell 中正确加载

最常见的原因是 `nvm` 在非交互式 shell 中未加载。将以下内容添加到您的 shell 配置文件（`~/.bashrc`、`~/.zshrc` 等）：

```
## 如果存在，加载 nvm
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"
```

或在您的当前会话中直接运行：

```
source ~/.nvm/nvm.sh
```

替代方案：调整 PATH 顺序

如果 nvm 正确加载但 Windows 路径仍然优先，您可以在 shell 配置中显式将 Linux 路径添加到 PATH 的前面：

```
export PATH="$HOME/.nvm/versions/node/$(node -v)/bin:$PATH"
```

Warning:

避免通过 `appendWindowsPath = false` 禁用 Windows PATH 导入，因为这会破坏从 WSL 调用 Windows 可执行文件的能力。同样，如果您在 Windows 开发中使用 Node.js，避免从 Windows 卸载它。

WSL2 sandbox 设置

[Sandboxing](#) 在 WSL2 上受支持，但需要安装额外的包。如果运行 `/sandbox` 时看到”Sandbox requires socat and bubblewrap”之类的错误，请安装依赖项：

Ubuntu/Debian

```
sudo apt-get install bubblewrap socat
```

Fedora

```
sudo dnf install bubblewrap socat
```

WSL1 不支持 sandboxing。如果您看到”Sandboxing requires WSL2”，您需要升级到 WSL2 或在不使用 sandboxing 的情况下运行 Claude Code。

安装期间的权限错误

如果本机安装程序因权限错误失败，目标目录可能不可写。请参阅[检查目录权限](#)。

如果您之前使用 npm 安装并遇到 npm 特定的权限错误，请切换到本机安装程序：

```
curl -fsSL https://claude.ai/install.sh | bash
```


权限和身份验证

这些部分涉及登录失败、令牌问题和权限提示行为。

重复的权限提示

如果您发现自己反复批准相同的命令，您可以使用 `/permissions` 命令允许特定工具无需批准即可运行。请参阅[权限文档](#)。

身份验证问题

如果您遇到身份验证问题：

1. 运行 `/logout` 完全注销
2. 关闭 Claude Code
3. 使用 `claude` 重新启动并再次完成身份验证过程

如果浏览器在登录期间不会自动打开，按 `c` 将 OAuth URL 复制到您的剪贴板，然后手动将其粘贴到您的浏览器中。

OAuth 错误：无效代码

如果您看到 `OAuth error: Invalid code. Please make sure the full code was copied`，登录代码已过期或在复制粘贴期间被截断。

解决方案：

- 在浏览器打开后按 Enter 重试并快速完成登录
- 如果浏览器不会自动打开，输入 `c` 复制完整 URL
- 如果使用远程/SSH 会话，浏览器可能在错误的机器上打开。复制终端中显示的 URL 并在您的本地浏览器中打开它。

登录后 403 Forbidden

如果登录后看到

```
API Error: 403 {"error":{"type":"forbidden","message":"Request not allowed"}} :
```

- **Claude Pro/Max 用户：**在 claude.ai/settings 验证您的订阅是否有效
- **Console 用户：**确认您的账户已由您的管理员分配“Claude Code”或“Developer”角色
- **在代理后面：**企业代理可能干扰 API 请求。有关代理设置，请参阅[网络配置](#)。

OAuth 登录在 WSL2 中失败

如果 WSL 无法打开您的 Windows 浏览器，WSL2 中基于浏览器的登录可能失败。设置 `BROWSER` 环境变量：

```
export BROWSER="/mnt/c/Program Files/Google/Chrome/Application/chrome.exe"
claude
```

或手动复制 URL：当登录提示出现时，按 **c** 复制 OAuth URL，然后将其粘贴到您的 Windows 浏览器中。

“未登录”或令牌已过期

如果 Claude Code 在会话后提示您再次登录，您的 OAuth 令牌可能已过期。

运行 `/login` 重新进行身份验证。如果这种情况经常发生，请检查您的系统时钟是否准确，因为令牌验证取决于正确的时间戳。

配置文件位置

Claude Code 在多个位置存储配置：

文件	目的
<code>~/.claude/settings.json</code>	用户设置（权限、hooks、模型覆盖）
<code>.claude/settings.json</code>	项目设置（检入源代码控制）
<code>.claude/settings.local.json</code>	本地项目设置（未提交）
<code>~/.claude.json</code>	全局状态（主题、OAuth、MCP 服务器）
<code>.mcp.json</code>	项目 MCP 服务器（检入源代码控制）
<code>managed-mcp.json</code>	托管 MCP 服务器
托管设置	托管设置 （服务器管理、MDM/OS 级别策略或基于文件）

在 Windows 上，`~` 指您的用户主目录，例如 `C:\Users\YourName`。

有关配置这些文件的详细信息，请参阅[设置](#)和[MCP](#)。

重置配置

要将 Claude Code 重置为默认设置，您可以删除配置文件：

```
## 重置所有用户设置和状态
```

```
rm ~/.claude.json
```

```
rm -rf ~/.claude/
```

```
## 重置项目特定设置
```

```
rm -rf .claude/
```

```
rm .mcp.json
```

Warning:

这将删除您的所有设置、MCP 服务器配置和会话历史记录。

性能和稳定性

这些部分涵盖与资源使用、响应性和搜索行为相关的问题。

高 CPU 或内存使用

Claude Code 设计用于大多数开发环境，但在处理大型代码库时可能消耗大量资源。如果您遇到性能问题：

1. 定期使用 `/compact` 以减少上下文大小
2. 在主要任务之间关闭并重启 Claude Code
3. 考虑将大型构建目录添加到您的 `.gitignore` 文件

命令挂起或冻结

如果 Claude Code 似乎无响应：

1. 按 Ctrl+C 尝试取消当前操作
2. 如果无响应，您可能需要关闭终端并重新启动

搜索和发现问题

如果搜索工具、`@file` 提及、自定义代理和自定义 skills 不起作用，请安装系统

`ripgrep`：

```
## macOS (Homebrew)
brew install ripgrep

## Windows (winget)
winget install BurntSushi.ripgrep.MSVC

## Ubuntu/Debian
sudo apt install ripgrep

## Alpine Linux
apk add ripgrep

## Arch Linux
pacman -S ripgrep
```

然后在您的[环境](#)中设置 `USE_BUILTIN_RIPGREP=0`。

WSL 上的搜索速度缓慢或结果不完整

在 WSL 上[跨文件系统工作](#)时的磁盘读取性能损失可能导致使用 Claude Code 在 WSL 上时搜索匹配数少于预期。搜索仍然有效，但返回的结果少于本机文件系统。

Note:

在这种情况下，`/doctor` 将显示搜索为正常。

解决方案：

1. **提交更具体的搜索：**通过指定目录或文件类型来减少搜索的文件数：“在 auth-service 包中搜索 JWT 验证逻辑”或“在 JS 文件中查找 md5 哈希的使用”。
2. **将项目移到 Linux 文件系统：**如果可能，确保您的项目位于 Linux 文件系统（`/home/`）而不是 Windows 文件系统（`/mnt/c/`）。
3. **改用本机 Windows：**考虑在 Windows 上本机运行 Claude Code 而不是通过 WSL，以获得更好的文件系统性能。

IDE 集成问题

如果 Claude Code 未连接到您的 IDE 或在 IDE 终端中表现异常，请尝试以下解决方案。

WSL2 上未检测到 JetBrains IDE

如果您在 WSL2 上使用 Claude Code 和 JetBrains IDE，并收到” No available IDEs detected” 错误，这可能是由于 WSL2 的网络配置或 Windows 防火墙阻止连接。

WSL2 网络模式

WSL2 默认使用 NAT 网络，这可能阻止 IDE 检测。您有两个选项：

选项 1：配置 Windows 防火墙（推荐）

1. 找到您的 WSL2 IP 地址：

```
wsl hostname -I  
## 示例输出: 172.21.123.45
```

1. 以管理员身份打开 PowerShell 并创建防火墙规则：

```
New-NetFirewallRule -DisplayName "Allow WSL2 Internal Traffic" -Direction Inbound  
-Protocol TCP -Action Allow -RemoteAddress 172.21.0.0/16 -LocalAddress 172.21.0.0/  
16
```

根据步骤 1 中的 WSL2 子网调整 IP 范围。

1. 重启您的 IDE 和 Claude Code

选项 2：切换到镜像网络

在您的 Windows 用户目录中添加到 `.wslconfig`：

```
[wsl2]  
networkingMode=mirrored
```

然后从 PowerShell 使用 `wsl --shutdown` 重启 WSL。

Note:

这些网络问题仅影响 WSL2。WSL1 直接使用主机的网络，不需要这些配置。

有关其他 JetBrains 配置提示，请参阅 [JetBrains IDE 指南](#)。

报告 Windows IDE 集成问题

如果您在 Windows 上遇到 IDE 集成问题，请使用以下信息[创建问题](#)：

- 环境类型：本机 Windows (Git Bash) 或 WSL1/WSL2
- WSL 网络模式（如适用）：NAT 或镜像
- IDE 名称和版本
- Claude Code 扩展/插件版本
- Shell 类型：Bash、Zsh、PowerShell 等

JetBrains IDE 终端中的 Escape 键不起作用

如果您在 JetBrains 终端中使用 Claude Code 且 **Esc** 键不会按预期中断代理，这可能是由于 JetBrains 默认快捷键的冲突。

要修复此问题：

1. 转到设置 → 工具 → 终端
2. 要么：
 - 取消选中” Move focus to the editor with Escape” ，或
 - 单击” Configure terminal keybindings” 并删除” Switch focus to Editor” 快捷键
3. 应用更改

这允许 **Esc** 键正确中断 Claude Code 操作。

Markdown 格式问题

Claude Code 有时生成 markdown 文件，代码围栏上缺少语言标签，这可能影响 GitHub、编辑器和文档工具中的语法突出显示和可读性。

代码块中缺少语言标签

如果您在生成的 markdown 中注意到这样的代码块：

```
```  

function example() {
 return "hello";
}
```:text
```

而不是正确标记的块，如：

```
```javascript
function example() {
 return "hello";
}
```text
```

解决方案：

1. **要求 Claude 添加语言标签：**请求” Add appropriate language tags to all code blocks in this markdown file.”
2. **使用后处理 hooks：**设置自动格式化 hooks 以检测和添加缺失的语言标签。有关示例，请参阅[编辑后自动格式化代码](#)的 PostToolUse 格式化 hook。
3. **手动验证：**生成 markdown 文件后，查看它们以确保正确的代码块格式，如果需要，请求更正。

不一致的间距和格式

如果生成的 markdown 有过多的空行或不一致的间距：

解决方案：

1. **请求格式更正：**要求 Claude” Fix spacing and formatting issues in this markdown file.”
2. **使用格式化工具：**设置 hooks 以在生成的 markdown 文件上运行 markdown 格式化程序，如 `prettier` 或自定义格式化脚本。
3. **指定格式偏好：**在您的提示或项目[内存](#)文件中包含格式要求。

减少 markdown 格式问题

要最小化格式问题：

- **在请求中明确：**要求” properly formatted markdown with language-tagged code blocks”
- **使用项目约定：**在 `CLAUDE.md` 中记录您首选的 markdown 风格
- **设置验证 hooks：**使用后处理 hooks 自动验证和修复常见格式问题

获取更多帮助

如果您遇到此处未涵盖的问题：

1. 在 Claude Code 中使用 `/bug` 命令直接向 Anthropic 报告问题
2. 检查 [GitHub 存储库](#) 以了解已知问题

3. 运行 `/doctor` 以诊断问题。它检查：

- 安装类型、版本和搜索功能
- 自动更新状态和可用版本
- 无效的设置文件（格式错误的 JSON、不正确的类型）
- MCP 服务器配置错误
- 快捷键配置问题
- 上下文使用警告（大型 CLAUDE.md 文件、高 MCP 令牌使用、无法访问的权限规则）
- 插件和代理加载错误

4. 直接向 Claude 询问其功能和特性 - Claude 可以内置访问其文档