使用 Claude 编程指南

获得更高代码质量与更准确结果的完整手册

Claude Sonnet 4.6 · 2026

第一章：对话 vs 项目——核心区别

理解「对话」与「项目」的区别，是高效使用 Claude 的第一步。错误地使用对话来做长期开发，会导致 Claude 反复「失忆」，浪费大量时间重新说明背景。

1.1 直观对比

维度	□ 对话 (Chat)	□ 项目 (Project)
记忆范围	仅限本次会话，关闭后清空	跨对话持久保存，长期可用
系统提示词	无（或每次手动粘贴）	可设置项目级系统提示，始终生效
上传文件	仅在当前对话中可见	文件持久存储，所有对话共享
适用场景	一次性问答、临时任务	长期项目、代码库、团队协作
推荐用于编程？	一般性提问 ✓	持续开发项目 **强烈推荐** ✓✓

1.2 何时用对话

- 快速验证一个算法思路
- 询问某个 API 的用法
- 临时的代码转换（如 JSON 格式化）
- 一次性的 debug 问题

1.3 何时用项目（强烈推荐用于编程）

- 任何持续超过一次会话的开发任务
- 有固定代码库、架构文档需要 Claude 长期理解的项目
- 需要统一编码规范、代码风格的团队项目
- 需要上传 API 文档、数据库 Schema 等参考资料

□ **最佳实践** 在项目中上传核心文件（README、架构图、主要模块），Claude 将在整个项目生命周期中持续引用它们。

第二章：项目系统提示词——让 Claude 了解你的项目

项目系统提示词是项目模式的核心优势。它在每次对话开始时自动注入，Claude 始终知道你的项目背景、技术栈和规范要求。

2.1 推荐的系统提示词模板

```
## 项目背景
本项目是一个基于 FastAPI + PostgreSQL 的 SaaS 后端服务。
Python 3.11，使用 Poetry 管理依赖，部署在 AWS Lambda。

## 编码规范
- 所有函数必须有类型注解（PEP 484）
- 使用 Pydantic v2 做数据验证
- 遵循 Google Python Style Guide
- 测试框架：pytest + pytest-asyncio

## 输出要求
- 直接给出完整可运行代码，不省略
- 每个函数附带 docstring
- 涉及数据库操作时，同时提供回滚方案
```

2.2 系统提示词的关键要素

- 技术栈全貌：语言版本、框架、主要依赖库
- 代码风格：命名规范、注释语言（中/英）、格式化工具
- 输出格式：是否需要测试、文档字符串、错误处理
- 项目约束：不能引入的依赖、性能要求、安全限制
- 团队上下文：是独立开发还是需要 Code Review 友好的风格

第三章：编写高质量编程 Prompt

Prompt 质量直接决定代码质量。以下是经过验证的技巧与反模式对照。

3.1 质量对照表

技巧	✕低质量示例	☑高质量示例
明确语言和框架	帮我写个排序函数	用 Python 3.11 写一个归并排序函数，需要类型注解和单元测试

描述约束条件	优化这段代码	优化这段代码的时间复杂度，不引入新依赖，保持 API 不变
指定输出格式	解释这个错误	解释此错误原因，给出修复代码，并说明如何预防
分步骤思考	重构这个模块	先分析现有问题，再提出重构方案，最后给出完整代码

3.2 结构化 Prompt 模板

对于复杂的编程任务，推荐使用以下四段式结构：

① 背景（Context）

```
我正在开发一个电商平台的订单服务，使用 Django 5.0 + DRF。
当前 OrderSerializer 在处理嵌套数据时性能低下（N+1 查询问题）。
```

② 目标（Goal）

```
优化订单详情 API 的查询性能，目标响应时间 < 200ms（当前约 800ms）。
```

③ 约束（Constraints）

```
- 保持现有 API 响应格式不变（不破坏前端）
- 不能使用 Celery 或异步任务（当前架构限制）
- Django ORM only，不写原生 SQL
```

④ 输出要求（Output）

```
请给出：1) 问题分析  2) 优化后的完整代码  3) 性能对比说明
```

✅**关键原则** Claude 处理「约束明确」的任务比「开放式」任务表现好得多。约束越具体，代码越贴合实际需求。

第四章：获得更高代码质量的技巧

4.1 要求 Claude 先分析再编码

直接要求代码往往得到「能跑但不够好」的结果。改为分两步：

- 第一步：「分析这段代码存在哪些问题，不要给代码，只给分析」
- 第二步：确认分析准确后，「现在基于你的分析给出重构后的完整代码」

□ **原因** 分步骤让 Claude 先建立问题模型，再生成解决方案，避免直接输出表面修复。

4.2 要求完整代码，明确禁止省略

Claude 有时会用注释替代实现。使用以下指令强制完整输出：

```
请给出完整代码。不要使用「# ... 其余代码保持不变」
```

```
或「# 省略其他方法」等占位符。所有方法都必须实现。
```

4.3 要求 Claude 进行自我审查

在得到代码后，追加提问：

```
请以 Senior Engineer 的视角审查你刚才写的代码：
1. 是否有边界条件未处理？
2. 是否存在潜在的性能问题？
3. 是否符合我们的编码规范？
如有问题，请直接给出修正版本。
```

4.4 提供上下文代码

如果要修改现有功能，始终提供：

- 相关函数的完整代码（不要只截取片段）
- 调用方式或测试用例
- 期望的输入/输出示例
- 当前出现的错误信息（完整的 stack trace）

第五章：使用 Claude 进行高效调试

5.1 提供完整错误信息

模糊的错误描述是调试失败的主要原因。标准格式：

```
【环境】Python 3.11, FastAPI 0.104, Ubuntu 22.04
【操作】调用 POST /api/orders 接口
【期望】返回 201 Created 和订单 ID
【实际】返回 500，完整 traceback 如下：
  sqlalchemy.exc.IntegrityError: (psycopg2.errors.NotNullViolation)
  null value in column "user_id" violates not-null constraint
  [SQL: INSERT INTO orders (product_id, amount) VALUES (%s, %s)]
```

5.2 橡皮鸭调试法（升级版）

当你已经知道大概方向但不确定时：

```
我怀疑问题出在数据库事务的提交时机上，
以下是我的推断过程：[你的分析]
请告诉我这个推断是否正确，如果错误请指出真正原因。
```

□ **效果** 引导 Claude 验证你的思路，比直接问「为什么报错」更快得到准确答案。

5.3 逐步缩小范围

- 先问「这个错误最可能的 3 个原因是什么？」

- 根据回答添加日志或断点，获得更多信息
- 再次提问时附上新的信息，逐步聚焦

第六章：长期项目的上下文管理

6.1 上传到项目的核心文件

- README.md（项目概览、安装、使用说明）
- 架构图或模块划分说明
- 数据库 Schema（ERD 或 models.py）
- API 文档或 OpenAPI spec
- 编码规范文档（如 CONTRIBUTING.md）
- 常见问题 FAQ（避免 Claude 重复犯同样的错误）

6.2 维护「项目日志」文件

在项目中维护一个 CLAUDE_NOTES.md 文件，记录：

```
# 已解决的关键问题
- [2024-01] 订单查询 N+1 问题：使用 select_related 解决
- [2024-02] Redis 连接池耗尽：调整 max_connections=50

# 已知限制（不要再尝试）
- 不能用 async SQLAlchemy（当前连接池不支持）
- 不能升级 Pydantic 到 v3（依赖冲突）

# 架构决策记录
- 使用 UUID 而非自增 ID：原因是多数据中心同步需求
```

□ **效果** 每次对话开始时，将此文件内容粘贴给 Claude，可以避免重复踩坑，显著提升准确率。

6.3 对话过长时的处理策略

当单次对话超过 20-30 轮时，Claude 对早期上下文的记忆会逐渐衰减：

- 新建对话，重新粘贴系统提示词和关键代码
- 用一段话总结上一次对话的结论和当前任务状态
- 只保留最相关的代码片段，而非完整文件

附录：高频指令速查

场景	推荐指令
需要完整代码	给出完整实现，不要省略任何部分
代码审查	以 Senior Engineer 视角找出所有潜在问题
性能优化	分析时间/空间复杂度，给出优化版本并对比
写测试	为以上代码写 pytest 单元测试，覆盖边界条件
解释代码	逐行解释这段代码，重点说明设计意图
重构建议	不改变功能，重构以提高可读性和可维护性
安全审查	从安全角度审查，找出 SQL 注入/XSS 等风险
先分析后编码	先分析问题，获得确认后再给代码
