第 10 周：核心开发与调试

让系统真正跑起来

北京石油化工学院\人工智能研究院\王文通

通选课

2025-2026 学年

北京石油化工学院

BEIJING INSTITUTE OF PETROCHEMICAL TECHNOLOGY

课程概览

前置知识自检

学习本课程前，请确认已掌握以下知识：

Python 基础

- 变量、数据类型、运算符
- 条件判断（if/elif/else）
- 循环结构（for/while）
- 函数定义与调用
- 类与对象基础

文件操作

- 读取/写入文本文件
- 路径处理（os.path）

版本控制基础

- Git 基本概念（仓库、提交）
- 基本命令（add/commit/push）
- 分支概念（可选）

数学基础

- 矩阵/数组基本概念
- 百分比计算
- 集合论基础（可选）

如需补充学习

三条学习路径，适合不同基础的你：

观察者：重点学习模块 1-2，直接运行代码示例

使用者：完成全部模块，重点实践模块 3-4

创造者：深入学习全部内容，完成模块 4 挑战任务

;

本周预习资源

课前 5 分钟视频（必须观看）

- **视频 1：**软件工程基础（模块化与接口设计）
  - 什么是模块化？为什么要模块化?
  - 接口设计的基本原则
- **视频 2：**团队协作基础（Git 工作流与代码规范）
  - Git 基本概念：仓库、分支、提交
  - 代码规范的重要性（PEP 8 简介）

预习目标：

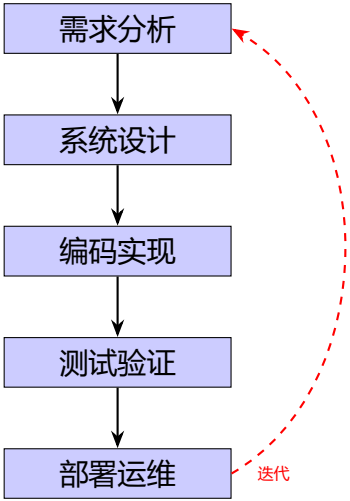
- 理解模块化设计的好处（高内聚低耦合）
- 掌握 Git 基本工作流（add/commit/push/pull）
- 了解代码规范的重要性（PEP 8）

预习检查：

软件开发生命周期（SDLC）

经典阶段：

1. **需求分析**：明确系统功能与性能要求
2. **系统设计**：架构设计与模块划分
3. **编码实现**：将设计转化为代码
4. **测试验证**：确保质量符合预期
5. **部署运维**：交付并持续维护



开发模型对比

瀑布模型

- 阶段清晰，顺序执行
- 文档驱动，易于管理
- 变更成本高
- 适合需求明确的项目

适用场景：

- 政府、金融项目
- 安全关键系统

敏捷开发

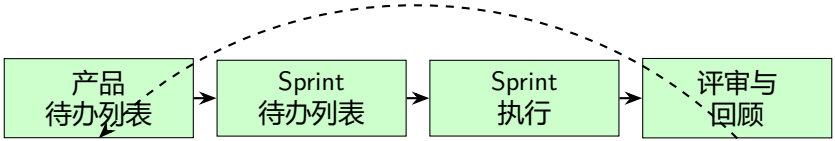
- 迭代增量，快速交付
- 拥抱变化，持续反馈
- 轻量文档，重视协作
- 适合需求易变项目

适用场景：

- 互联网产品
- 创新实验项目

敏捷开发核心实践

Scrum 框架：

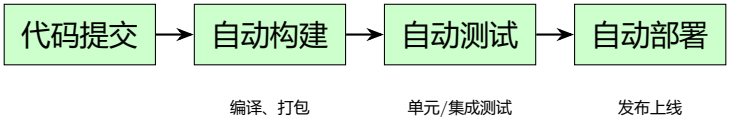


关键概念：

- **Sprint**：2-4 周的迭代周期
- **每日站会**：15 分钟同步进展
- **迭代评审**：演示成果，收集反馈
- **回顾会议**：持续改进流程

持续集成与持续部署（CI/CD）

核心理念：自动化一切可以自动化的



关键实践：

- **版本控制**：Git 管理代码，分支策略规范
- **自动化构建**：Jenkins、GitHub Actions、GitLab CI
- **自动化测试**：每次提交触发测试套件
- **环境一致性**：Docker 容器化部署

版本控制最佳实践

Git 分支策略：

Git Flow 模型：

- `main/master`：生产环境
- `develop`：开发集成分支
- `feature/*`：功能开发分支
- `release/*`：发布准备分支
- `hotfix/*`：紧急修复分支

Commit Message 规范：

- `feat`：新功能
- `fix`：修复 bug
- `docs`：文档更新
- `refactor`：重构
- `test`：测试相关
- `chore`：构建/工具

示例：规范的 Commit

```
feat(answer-sheet): 添加答题卡区域检测算法
- 实现基于轮廓的区域分割
```

代码规范与风格

为什么需要代码规范?

- **可读性**：代码被阅读的次数远多于编写
- **可维护性**：团队协作用统一" 语言"
- **减少错误**：规范规避常见陷阱

Python 代码规范（PEP 8）:

- 缩进：4 个空格（禁用 Tab）
- 行长度：每行不超过 79 字符
- 命名：函数/变量用小写下划线，类用大驼峰
- 空行：类和函数间空两行，方法间空一行

命名规范示例:

- `class AnswerSheetDetector:` (类名大驼峰)
- `def detect_regions(image):` (函数名小写 + 下划线)
- `MAX_SCORE = 100` (常量大写)

代码审查（Code Review）

审查内容：

- 代码正确性
- 逻辑清晰度
- 性能影响
- 安全漏洞
- 测试覆盖
- 文档完整性

审查原则：

- 对事不对人
- 小步快跑（PR 要小）
- 及时响应（24 小时内）
- 记录决策（为什么修改）

代码审查清单示例

- ☐ 是否处理了边界情况?
- ☐ 是否有适当的错误处理?
- ☐ 命名是否清晰表达意图?
- ☐ 是否有重复代码可以提取?

技术债务管理

什么是技术债务?

> ”为了快速交付而采取的非最优技术方案，未来需要额外成本来偿还。”
> ——*Ward Cunningham*

技术债务类型:

- **有意债务**：战略性地选择快速方案，计划偿还
- **无意债务**：由于缺乏经验或知识而引入
- **过期债务**：原本合理的方案随时间变得过时

管理策略:

- 债务可视化：记录和追踪技术债务
- 定期偿还：每个 Sprint 预留 20% 时间重构
- 防止累积：代码审查时识别新增债务

重构与优化

重构定义：

”在不改变代码外部行为的前提下，改善其内部结构。”——*Martin Fowler*

重构时机：

- **规则一**：添加功能时
- **规则二**：修复 bug 时
- **规则三**：代码审查时

常见重构技术：

- 提取函数（Extract Method）
- 内联函数（Inline Method）
- 提取类（Extract Class）
- 搬移函数（Move Method）
- 重命名（Rename）
- 引入参数对象
- 替换魔法数字
- 简化条件表达式

调试的基本概念

调试（Debugging）：识别、定位和修复软件缺陷的过程

常见错误类型：

- **语法错误**：编译/解析失败
- **运行时错误**：异常、崩溃
- **逻辑错误**：结果不正确
- **性能错误**：运行过慢
- **并发错误**：时序相关、难复现

调试的本质：

- 建立假设
- 收集证据
- 验证假设
- 修正理解

调试策略与方法

系统化调试方法：

1. **复现问题**
   - 找到稳定复现问题的步骤
   - 最小化复现案例（Minimal Reproduction）
2. **定位根因**
   - 二分法缩小范围
   - 使用调试器或日志追踪
3. **验证假设**
   - 临时修改测试假设
   - 添加断言验证状态
4. **修复并验证**
   - 修复问题
   - 确保修复不引入新问题

日志与监控

日志记录的重要性：

- 生产环境调试的唯一手段
- 系统行为的历史记录
- 问题诊断的关键证据

日志级别：

级别	用途	示例
DEBUG	详细调试信息	变量值、执行路径
INFO	正常操作记录	服务启动、配置加载
WARNING	警告，可能有问题	资源即将耗尽
ERROR	错误，功能受影响	文件读取失败
CRITICAL	严重错误，系统崩溃	数据库连接断开

日志最佳实践

性能监控与分析

关键性能指标（KPI）：

响应时间

- 平均响应时间
- P95/P99 延迟
- 超时率

吞吐量

- QPS/TPS
- 并发用户数
- 峰值处理能力

资源使用

- CPU 使用率
- 内存占用
- 磁盘 IO
- 网络流量

错误率

- 异常比例
- 失败请求率
- 服务可用性

监控工具链：

- Prometheus + Grafana：指标采集与可视化

模块 00-背景知识：要点速查

核心概念

- **SDLC**：软件开发生命周期（需求 → 设计 → 编码 → 测试 → 部署）
- **敏捷开发**：迭代增量，快速反馈（2-4 周 Sprint）
- **CI/CD**：持续集成与持续部署（自动化构建、测试、发布）
- **Code Review**：代码审查，对事不对人，小步快跑
- **重构**：不改变外部行为，改善内部结构

关键公式

- **高内聚** = 模块内元素紧密相关
- **低耦合** = 模块间依赖最小

下一步学习

模块化设计原则

为什么需要模块化?

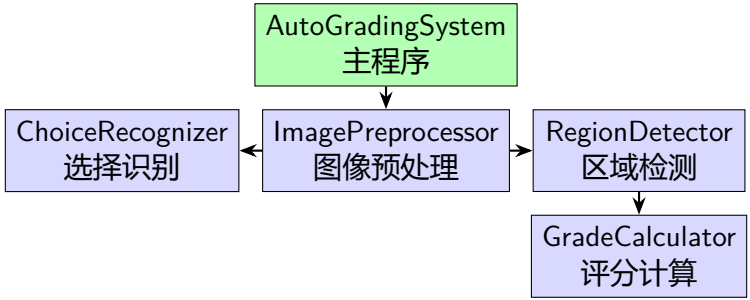
- **降低复杂度**：分而治之，理解局部而非整体
- **提高复用性**：独立模块可在不同项目使用
- **便于维护**：修改局部不影响全局
- **支持并行开发**：不同开发者负责不同模块

高内聚低耦合

高内聚：模块内部元素紧密相关，职责单一
低耦合：模块之间依赖最小化，接口清晰

智能阅卷系统的模块划分

系统模块结构：



模块间关系：

- 主程序调用各模块
- 预处理 → 区域检测 → 识别 → 评分
- 各模块独立，便于单独开发和测试

接口设计原则

良好的接口应该：

1. **简洁明了**：只做一件事，命名清晰
2. **隐藏实现**：调用者无需知道内部细节
3. **稳定可靠**：接口定义一旦发布尽量少变更
4. **完备一致**：错误处理、边界情况都有定义

接口示例：选择题识别

```
def recognize_choice(image, options, threshold=0.5):
    """ 识别选择题答案"""
    Args:
        image: 题目区域图像
        options: 选项位置列表 ['A', 'B', 'C', 'D']
        threshold: 填涂判定阈值
    Returns: str: 识别结果，如'A'，或 None
```

面向对象设计：封装

封装（Encapsulation）：隐藏内部实现，暴露必要接口

封装的好处：

- 保护内部状态不被意外修改
- 降低模块间的耦合
- 便于修改内部实现
- 提供清晰的使用方式

Python 中的封装：

- `_var`：建议私有（约定）
- `__var`：名称改写（伪私有）
- `property`：属性访问控制

示例：AnswerSheet 类

```
class AnswerSheet:
    def __init__(self):
        self.__image = None # 私有属性
        self._threshold = 0.5
    @property
```

面向对象设计：继承与多态

继承（Inheritance）：

- 代码复用，提取公共行为
- 建立类型层次关系
- 支持扩展和定制

多态（Polymorphism）：

- 同一接口，不同实现
- 运行时动态绑定
- 提高代码灵活性

识别器基类

```
class BaseRecognizer:
    def recognize(self, image):
        raise NotImplementedError
class OMRRecognizer(BaseRecognizer):
    def recognize(self, image):
```

SOLID 设计原则

面向对象设计的五大原则：

S - 单一职责原则 类只负责一项功能

O - 开闭原则 对扩展开放，对修改封闭

L - 里氏替换原则 子类可替换父类使用

I - 接口隔离原则 客户端不依赖不需要的接口

D - 依赖倒置原则 依赖抽象而非具体实现

应用示例：阅卷系统

- **S**：ImagePreprocessor 只处理图像，不负责识别
- **O**：新增题型识别器，无需修改主程序
- **D**：依赖 BaseRecognizer 接口，不依赖具体实现

函数式编程思想

核心概念：

纯函数

- 相同输入永远得到相同输出
- 无副作用（不修改外部状态）
- 便于测试和推理

高阶函数

- 函数可作为参数传递
- 函数可返回函数
- `map`, `filter`, `reduce`

不可变数据

- 数据创建后不可修改
- 避免共享状态问题
- 简化并发编程

函数组合

- 小函数组合成复杂功能
- 管道式数据处理
- `result = f(g(h(x)))`

Python 示例

```
images = [load(p) for p in paths] # 列表推导
```

Python 异常层次结构

标准异常类型：

常见异常：

- `Exception`：所有异常的基类
- `ValueError`：值不正确
- `TypeError`：类型错误
- `FileNotFoundError`：文件不存在
- `KeyError` / `IndexError`：访问错误

自定义异常：

```
class
GradingError(Exception):
    pass
class
ImageLoadError(GradingError):
    pass
class
RecognitionError(GradingError):
    pass
```

异常捕获层次

错误处理策略

防御式编程：

- 提前验证输入条件
- 断言不变式
- 失败快速（Fail Fast）

异常转换与封装：

- 捕获底层异常，转换为业务异常
- 保留原始异常信息（异常链）
- 提供更有意义的错误消息

异常转换示例

```
try:
    img = cv2.imread(path)
    if img is None:
        raise ImageLoadError(f" 无法加载: {path}")
except cv2.error as e:
```

日志记录最佳实践

结构化日志配置：

```
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```

日志记录建议：

- DEBUG：详细变量值、执行路径
- INFO：重要操作里程碑
- WARNING：可恢复的异常情况
- ERROR：功能受影响的错误

断点调试技巧

断点类型：

行断点

- 最常用，执行到指定行暂停
- IDE 中点击行号设置
- 临时查看程序状态

条件断点

- 满足条件时才暂停
- 右键断点 → 编辑条件
- 示例：`i == 100`

异常断点

- 发生异常时自动暂停
- 捕获未被处理的异常
- 快速定位异常源头

日志断点

- 不暂停，输出日志
- 用于性能敏感代码
- 避免中断程序执行

Python 代码中断点

```
import pdb; pdb.set_trace() # 设置断点
```

单步执行与变量查看

调试操作：

Resume	继续执行到下一断点
Step Over	执行当前行，不进入函数
Step Into	进入函数内部
Step Out	跳出当前函数

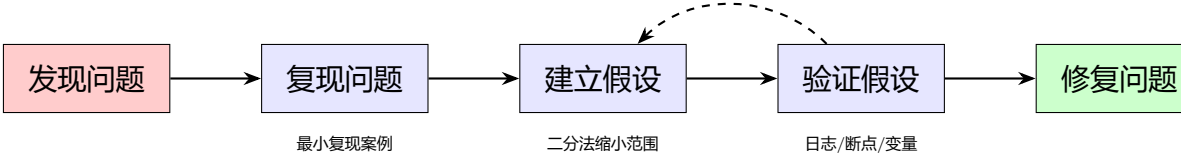
变量查看：

- Variables 面板：所有局部变量
- Watches：自定义表达式
- 鼠标悬停：快速查看
- 控制台：执行表达式

调试技巧

- 从预期结果倒推：哪里可能导致这个结果？
- 检查中间状态：变量是否符合预期？
- 修改变量值：测试修复是否有效
- 使用条件断点：跳过不感兴趣的循环

调试流程可视化



调试本质：科学排查流程

1. **建立假设**：基于理解，推测可能出错的位置
2. **收集证据**：通过日志、断点、变量值获取数据
3. **验证假设**：检查证据是否支持假设
4. **修正理解**：根据结果调整假设，循环直到找到根因

关键洞察

调试不是乱试，而是**科学实验**……假设、验证、迭代的过程

性能分析工具

cProfile - CPU 性能分析：

```
import cProfile
cProfile.run('process_image(path)')
```

line_profiler - 逐行分析：

```
@profile
def process(image):
    result = detect(image) # 显示此行耗时
```

memory_profiler - 内存分析：

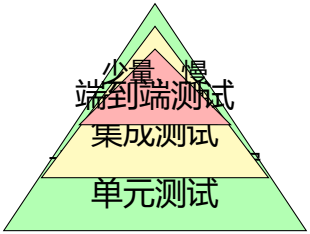
```
@profile
def load_images(paths):
    images = [load(p) for p in paths]
```

运行命令

```
kernprof -l -v script.py # line_profiler
```

测试金字塔

测试类型层次：

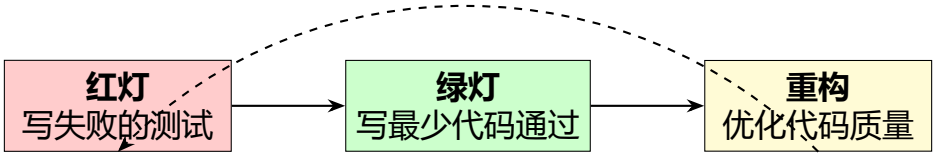


测试策略：

- 底层测试多，高层测试少
- 单元测试运行快，反馈及时
- E2E 测试保证关键路径正确

测试驱动开发（TDD）

TDD 循环：红-绿-重构



TDD 的好处：

- 保证代码可测试
- 充当使用文档
- 减少 bug 数量
- 提高设计质量

pytest 高级特性

Fixture 机制：

```
@pytest.fixture
def sample_image():
    return create_test_image()
def test_recognize(sample_image):
    result = recognize(sample_image)
    assert result == 'A'
```

参数化测试：

```
@pytest.mark.parametrize("input,expected", [
    (filled_img, 'A'),
    (empty_img, None),
    (partial_img, None)
])
def test_recognize(input, expected):
```

Mock 与测试隔离

为什么要 Mock?

- 隔离外部依赖（文件、网络、数据库）
- 加速测试执行
- 模拟异常情况
- 确保测试可重复

Mock 示例

```
from unittest.mock import patch, Mock
@patch('cv2.imread')
def test_load(mock_imread):
    mock_imread.return_value = test_img
    result = load('any_path.jpg')
    assert result is not None
    mock_imread.assert_called_once()
```

测试覆盖率

覆盖率指标：

语句覆盖

- 执行到的代码行比例
- 基础指标

分支覆盖

- if/else 各分支执行情况
- 更严格

使用 coverage.py：

```
pip install coverage
pytest-cov
pytest --cov=module
--cov-report=html
open htmlcov/index.html
```

覆盖率建议

- 核心业务逻辑：$> 80\%$
- 工具类：$> 90\%$
- 注意：高覆盖率不等于高质量
- 关注边界条件和异常路径

模块 01-核心理论：要点速查

设计原则

- **SOLID 原则**：单一职责、开闭原则、里氏替换、接口隔离、依赖倒置
- **封装**：隐藏实现细节，暴露必要接口
- **继承与多态**：代码复用，灵活扩展
- **纯函数**：相同输入 $\rightarrow$ 相同输出，无副作用

调试流程

发现问题 → 复现问题 → 建立假设 → 验证假设 → 修复问题

测试金字塔

大量单元测试（70%）+ 适量集成测试（20%）+ 少量 E2E 测试（10%）

概念补充：什么是重构?

生活类比

整理房间

- 房间很乱（代码可读性差）
- 不改变房间里的东西（功能不变）
- 只是重新整理收纳（优化结构）
- 目的是更容易找到东西（更容易维护）

代码示例

重构前

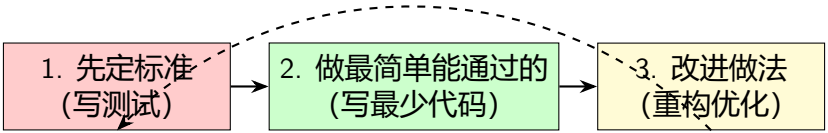
```
def f(x):
    # 100 行代码
```

重构后

```
def process():
    load()
    process_data()
    save()
```

概念补充：什么是测试驱动开发？

想象学做菜的过程：



好处

- 确保做出来的东西符合要求（测试通过）
- 避免过度设计（只写最少代码）
- 持续改进质量（每次都优化一点）

IDE 选择与对比

PyCharm

- 专业 Python IDE
- 智能代码补全
- 强大的调试器
- 集成测试工具
- 专业版收费

VS Code

- 轻量编辑器
- 插件生态丰富
- Python 扩展强大
- 免费开源
- 多语言支持

Jupyter

- 交互式环境
- 适合数据探索
- 可视化集成
- 文档混排
- 教学演示友好

推荐选择

- 大型项目开发：PyCharm（专业版）
- 日常/多语言开发：VS Code
- 数据分析/教学：Jupyter Notebook

调试工具详解

Python 内置调试器 pdb：

```
import pdb

def process_image(image_path):
    image = load_image(image_path)
    pdb.set_trace()  \#设置断点
    result = analyze(image)
    return result

\#常用pdb命令
\#n - 下一行（Step Over）
\#s - 进入函数（Step Into）
\#c - 继续执行
\#p var - 打印变量
\#q - 退出调试
```

ipdb 增强版：

- 语法高亮
- Tab 自动补全

性能分析工具

cProfile - 标准库性能分析：

```
import cProfile
import pstats

\#分析代码
profiler = cProfile.Profile()
profiler.enable()

\#要分析的代码
process_batch(images)

profiler.disable()
stats = pstats.Stats(profiler)
stats.sort_stats('cumulative')
stats.print_stats(20)  \#打印前20个热点
```

line_profiler - 逐行分析：

- 精确到每行代码的耗时
- 使用 @profile 装饰器标记

代码质量工具

静态检查工具

- **pylint**：全面代码检查
- **flake8**：PEP 8 + 复杂度检查
- **mypy**：静态类型检查
- **bandit**：安全漏洞扫描

代码格式化工具

- **black**：严格格式化，无配置
- **autopep8**：PEP 8 自动修复
- **isort**：import 排序
- **yapf**：Google 出品，可配置

```
\#使用示例
$ flake8 myproject/          \#检查代码风格
$ black myproject/           \#自动格式化
$ mypy myproject/            \#类型检查
$ bandit -r myproject/       \#安全扫描
```

pytest 测试框架

为什么选择 pytest?

- 简洁的断言语法（无需 `assertEqual`）
- 强大的 Fixture 机制
- 丰富的插件生态
- 详细的失败报告

unittest 风格:

```
class TestRecognizer(unittest.TestCase):
  def test(self):
    assert result == 'A'
```

pytest 风格:

```
def test_recognize():
  assert result == 'A'
```

Mock 与测试隔离

为什么要 Mock?

- 隔离被测代码，避免外部依赖
- 控制测试环境，确保可重复
- 模拟异常情况，提高覆盖率
- 加速测试执行（避免真实 IO）

Mock 对象示例:

```
from unittest.mock import Mock
mock = Mock()
mock.method.return_value = 42
assert mock.method() == 42
```

Patch 装饰器:

```
@patch('module.function')
def test(mocked_func):
    mocked_func.return_value = 42
```

测试覆盖率

coverage.py 使用：

```
$ pip install coverage pytest-cov
$ coverage run -m pytest
$ coverage report
$ coverage html
$ open htmlcov/index.html
```

覆盖率目标：

- 核心逻辑：80% 以上
- 工具类：90% 以上
- 但高覆盖率不等于高质量，关键是有效测试

Git 高级操作

常用高级命令：

- **交互式暂存：** `git add -p`
- **修改提交：** `git commit --amend`
- **储存修改：** `git stash` / `git stash pop`
- **变基整理：** `git rebase -i HEAD~3`
- **二分调试：** `git bisect start`

Git 工作流

Feature Branch 工作流

- 主分支保持稳定
- 每个功能新建分支
- 完成后 Code Review 合并
- 适合大多数团队

Forking 工作流

- 开发者 fork 仓库
- 在自己的仓库开发
- 通过 Pull Request 贡献
- 适合开源项目

Commit Message 规范：

- `feat`: 新功能
- `fix`: 修复 bug
- `docs`: 文档更新
- `refactor`: 重构
- `test`: 测试相关
- `chore`: 构建/工具

AI 编程工具概览

AI 改变编程方式：

传统编程流程：

1. 查阅文档
2. 编写代码
3. 调试测试
4. 优化重构

AI 辅助编程流程：

1. 描述需求
2. AI 生成代码
3. 验证测试
4. AI 辅助优化

主流 AI 编程工具：

- **Cursor**：AI 原生 IDE，内置 GPT-4
- **Claude Code**：Anthropic 官方 CLI 工具
- **GitHub Copilot**：代码自动补全

Cursor：AI 原生 IDE

核心特性：

- **AI Chat**：内置对话界面，无需切换窗口
- **代码生成**：根据描述生成完整函数
- **代码解释**：选中代码即可获得详细解释
- **重构建议**：自动识别 Code Smell 并优化

快捷键：

- `Cmd+L`：打开 AI Chat
- `Cmd+K`：生成/编辑代码
- `Cmd+I`：询问当前文件

适用场景：

- 快速原型开发
- 代码理解学习
- 重构优化代码

Claude Code：命令行 AI 助手

Claude Code 特点：

优势：

- 长上下文窗口（200K tokens）
- 精准代码理解
- 多文件编辑能力
- 命令行无缝集成

使用方式：

```
\#安装
npm install -g @anthropic-ai/claude-
```

```
\#使用
claude  \#启动交互会话
```

实战示例：

```
\#用户：解释这个检测填涂的函数
[选中代码]
```

```
\#Claude：这个函数通过灰度化和阈值判断...
\#- 先转为灰度图
```

GitHub Copilot：智能代码补全

Copilot 工作原理：

- 基于 OpenAI Codex 模型
- 从 GitHub 公开代码学习
- 根据上下文自动补全

使用场景：

最佳场景：

- 编写样板代码
- 生成测试用例
- API 调用示例
- 正则表达式

使用技巧：

- 写注释描述意图
- 函数命名要清晰
- Tab 接受建议
- Esc 忽略建议

Copilot 示例

```
\#注释：计算答题卡填涂区域密度
```

AI 辅助代码审查

用 AI 进行 Code Review：

Prompt 模板：

请审查这段代码的质量：

1. 代码规范问题
2. 潜在bug
3. 性能优化建议
4. 重构建议

[粘贴代码]

AI 会检查：

- 命名是否规范
- 是否有重复代码
- 错误处理是否完善
- 边界条件是否考虑
- 性能是否可优化

实战案例：

\#原始代码

```
def proc(img,p):
```

AI 辅助调试实战

场景：代码运行报错，看不懂错误信息

错误信息：

```
cv2.error: OpenCV(4.8.0) :-1: error: (-5:Bad argument)
in function 'threshold'
> Overwhelming requirement: (img.depth() == CV_8U ||
 img.depth() == CV_32F)
```

向 AI 提问：

这段OpenCV代码报错，是什么原因？如何修复？

[粘贴错误信息和相关代码]

AI 诊断：

- **问题**：图像深度不符合要求

模块 02-工具环境：要点速查

开发工具选择

- 大型项目：PyCharm
- 多语言/轻量：VS Code
- 数据分析：Jupyter

代码质量工具

- 检查：flake8、pylint
- 格式化：black
- 类型检查：mypy

AI 编程工具

- Cursor：AI 原生 IDE（`Cmd+L` 对话）
- Claude Code：命令行助手
- Copilot：代码补全

识别 Code Smells

什么是 Code Smell?

- 代码中潜在的糟糕设计指示
- 不是 bug，但可能导致问题
- 需要重构的信号

常见 Code Smells:

- 长函数（Long Method）
- 大类（Large Class）
- 重复代码（Duplicated Code）
- 过长参数列表（Long Parameter List）
- 发散式变化（Divergent Change）
- 特性依恋（Feature Envy）

示例：长函数

```
def process_answer_sheet(path):
    # 200行代码...
    image = load(path)
    gray = cv2.cvtColor(...)
    binary = cv2.threshold(...)
    contours = cv2.findContours(...)
    # ...更多代码
```

重构：提取函数

重构前：

```
def process_answer_sheet(path):
    image = cv2.imread(path)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)[1]
    contours = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APP
    # ... 混杂的处理逻辑
```

重构后：

```
def process_answer_sheet(path):
    image = load_image(path)
    binary = binarize(image)
    regions = find_regions(binary)
    return recognize(regions)
```

重构：提取类

重构前：所有逻辑在一个类中

```
class AnswerSheetProcessor:
    def process(self, path):
        # 加载
        # 预处理
        # 检测
        # 识别
        # 评分
        # 全部在这里...
```

重构后：职责分离

```
class ImagePreprocessor:
    def load(self, path): pass
    def normalize(self, image): pass
    def enhance(self, image): pass
```

重构：引入参数对象

重构前：参数过多

```
def recognize(image, x, y, width, height, threshold,
              options, algorithm, debug_mode):
    # 9个参数，难以理解
    pass

# 调用时容易出错
result = recognize(img, 100, 200, 50, 30, 0.5,
                   ['A','B','C','D'], 'omr', True)
```

重构后：参数对象

```
@dataclass
class RecognitionConfig:
    region: Rectangle
    threshold: float = 0.5
```

重构：替换魔法数字

重构前：代码中有未命名的常量

```
if density > 0.7:
    answer = 'A'
elif density > 0.5:
    answer = 'B'
elif density > 0.3:
    answer = 'C'
else:
    answer = 'D'

for i in range(100):
    process(i)
```

重构后：使用有意义的常量

```
# 常量定义
```

模块化重构实战

单体结构（避免）：

```
# 所有功能在一个文件中
# grading_system.py (1000行)
class AutoGradingSystem:
    def load(self): pass
    def preprocess(self): pass
    def detect(self): pass
    def recognize(self): pass
    def grade(self): pass
    def export(self): pass
    # ... 更多方法
```

模块化结构（推荐）：

```
# 清晰的模块划分
grading_system/
```

自定义异常层次

定义异常体系:

```
# 基类异常
class GradingError(Exception):
    """阅卷系统基础异常"""
    pass

# 具体异常
class ImageLoadError(GradingError):
    """图像加载失败"""
    pass

class RegionDetectionError(GradingError):
    """区域检测失败"""
    pass
```

异常捕获策略

策略一：在合适层级捕获异常

```
# 低层函数：抛出异常
def load_image(path):
    img = cv2.imread(path)
    if img is None:
        raise ImageLoadError(f"无法加载：{path}")
    return img

# 中层函数：转换异常
def process_sheet(path):
    try:
        img = load_image(path)
    except ImageLoadError as e:
        logger.error(f"加载失败：{e}")
```

防御式编程实战

输入验证：

```
def recognize_choice(image, options, threshold=0.5):
    # 输入验证
    if image is None:
        raise ValueError("图像不能为空")
    if image.size == 0:
        raise ValueError("图像尺寸无效")
    if not options:
        raise ValueError("选项列表不能为空")
    if not 0 < threshold < 1:
        raise ValueError(f"阈值必须在0-1之间，实际：{threshold}")

    # 业务逻辑
    result = _do_recognize(image, options, threshold)
```

日志系统配置

完整日志配置：

```
import logging
from logging.handlers import RotatingFileHandler

def setup_logging(name):
    logger = logging.getLogger(name)
    logger.setLevel(logging.DEBUG)

    # 文件处理器（带轮转）
    file_handler = RotatingFileHandler(
        'grading.log', maxBytes=10*1024*1024, backupCount=5
    )
    file_handler.setLevel(logging.INFO)
```

结构化日志输出

结构化日志:

```
import json
import logging

class StructuredLogger:
    def __init__(self, name):
        self.logger = logging.getLogger(name)

    def log_event(self, level, event, **context):
        log_data = {
            'event': event,
            'timestamp': datetime.now().isoformat(),
            **context
        }
```

条件断点实战

场景：循环中只关心特定情况

问题代码：

```
results = []
for i in range(10000):
    region = detect_region(i)
    result = recognize(region)
    results.append(result)
    # 只有i=5000时出错
```

条件断点设置：

```
# 在IDE中设置：
# 行号：  "result = recognize(region)"
# 条件：  i == 5000

# 或使用代码：
for i in range(10000):
    region = detect_region(i)
    if i == 5000:
        breakpoint()  # 只在这里暂停
    result = recognize(region)
```

复杂条件示例：

异常断点使用

自动在异常时暂停：
IDE 设置：

- PyCharm：Run → View Breakpoints → Python Exception
- VS Code：调试面板 → Breakpoints → 异常

选择捕获：

- Raise：抛出异常时
- Caught：捕获异常时

代码中的异常捕获：

```
# 在可疑代码块外
try:
    complex_operation()
except Exception as e:
    # 自动暂停并查看
    breakpoint()
    raise  # 重新抛出
```

使用场景

- 代码中某个位置抛出异常，但不知道原因
- 异常被上层捕获，难以定位源头

二分法定位问题

场景：长流程中某处出错

```
def process_exam(image_path):
    # 第一步
    image = load_image(image_path)
    print(f"[1] 图像加载: {image.shape if image is not None else 'None'}

    # 第二步
    preprocessed = preprocess(image)
    print(f"[2] 预处理: {preprocessed.shape}")

    # 第三步
    regions = detect_regions(preprocessed)
    print(f"[3] 检测到区域: {len(regions)}")
```

性能分析与优化

使用 cProfile 分析性能：

```
import cProfile
import pstats

def profile_process():
    profiler = cProfile.Profile()
    profiler.enable()

    # 执行要分析的代码
    result = process_exam("test.jpg")

    profiler.disable()

    # 输出统计
```

用 AI 理解复杂代码

场景：不理解 OpenCV 函数参数含义

向 AI 提问（RTF 框架）：

```
角色：OpenCV专家
任务：解释这个函数的参数
格式：分点说明，举例说明

cv2.findContours(
    binary, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE
)
```

AI 会解释：

- **binary**: 输入的二值图像
- **RETR_EXTERNAL**: 只检测外轮廓
- **CHAIN_APPROX_SIMPLE**: 压缩轮廓存储

AI 还会提供：

- 返回值说明（轮廓列表、层次结构）
- 使用示例代码
- 常见错误及解决方案

用 AI 生成测试用例

场景：为填涂识别函数生成测试

代码函数：

```
def recognize_filled(image):
    gray = cv2.cvtColor(image,
        cv2.COLOR_BGR2GRAY)
    density = np.sum(gray > 127) /
              gray.size
    return density > 0.5
```

Prompt 模板：

为这个函数生成pytest测试用例
覆盖以下场景：
1. 填涂区域（应返回True）
2. 空白区域（应返回False）
3. 部分填涂（边界情况）
4. 空图像（异常处理）

AI 生成的测试：

```
@pytest.mark.parametrize("image,expected", [
    (create_filled_img(), True),
    (create_empty_img(), False),
    (create_partial_img(0.6), True),
```

用 AI 辅助重构

场景：识别长函数中的 Code Smell
原始代码（有坏味道）：

```
def process_sheet(path):
    img = cv2.imread(path)
    gray = cv2.cvtColor(img,
        cv2.COLOR_BGR2GRAY)
    binary = cv2.threshold(
        gray, 127, 255,
        cv2.THRESH_BINARY)[1]
    # ...100+行混杂逻辑
```

AI 重构建议：

- 提取 `load_image()` 函数
- 提取 `binarize()` 函数
- 提取 `find_regions()` 函数

向 AI 提问：

这段代码有什么Code Smell？
请重构为更清晰的模块化结构
要求：
1. 提取独立函数
2. 添加文档字符串
3. 处理错误情况
4. 使用类型注解

AI 辅助错误诊断

场景：遇到难以理解的错误信息
错误信息：

```
OpenCV Error: Assertion failed
((npoints == prevpoints.size()) && ...
```

有效的 Prompt：

```
我遇到了OpenCV错误：
[粘贴完整错误信息和相关代码]

请告诉我：
1. 这个错误的可能原因是什么？
2. 如何定位具体问题？
3. 有哪些解决方案？
```

AI 诊断流程：

1. 分析错误类型（断言失败）
2. 指出可能原因（点集数量变化）
3. 提供检查步骤（打印变量值）
4. 给出修复方案（检查预处理逻辑）

AI 辅助调试最佳实践

何时使用 AI 辅助：

场景	最佳实践
理解 API 文档	提供”角色 + 任务 + 格式”的 Prompt
生成测试代码	指定测试框架和覆盖场景
重构建议	说明具体重构目标和约束
错误诊断	提供完整错误信息和上下文
性能优化	描述当前性能和优化目标

Prompt 工程技巧：

- **RTF**：Role（角色）、Task（任务）、Format（格式）
- **上下文**：提供足够的代码和错误信息
- **约束**：明确输出格式和具体要求
- **迭代**：根据结果逐步细化 Prompt

TDD 循环实践

场景：实现填涂识别功能

第一步：红灯（写失败的测试）

```
def test_filled_recognized():
    image = create_filled_image()
    result = recognize_filled(image)
    assert result == True
    # 运行：失败！函数不存在
```

第二步：绿灯（最少代码通过）

```
def recognize_filled(image):
    # 最简单的实现
    return True
    # 测试通过！
```

第三步：重构（优化实现）

```
# 添加更多测试
def test_empty_not_recognized():
    image = create_empty_image()
    assert recognize_filled(image) == False

# 重构实现
```

参数化测试

减少重复代码：

```
# 传统方式：重复
def test_threshold_0_3():
    assert recognize(img, 0.3) == 'A'

def test_threshold_0_5():
    assert recognize(img, 0.5) == 'A'

def test_threshold_0_7():
    assert recognize(img, 0.7) == None

# 参数化：简洁
@pytest.mark.parametrize("threshold,expected", [
    (0.3, 'A'),
```

Mock 实战：隔离外部依赖

场景：测试图像加载，但不依赖真实文件

问题：测试依赖文件系统

```
def test_load_image():
    img = load_image("test.jpg")
    # 需要真实文件
    # 文件可能不存在
    # 测试不稳定
```

解决：Mock cv2.imread

```
from unittest.mock import patch

@patch('cv2.imread')
def test_load_image(mock_imread):
    # 设置返回值
    mock_imread.return_value = fake_

    # 测试
    result = load_image("any_path.jp

    # 验证
    assert result is not None
```

测试夹具（Fixture）

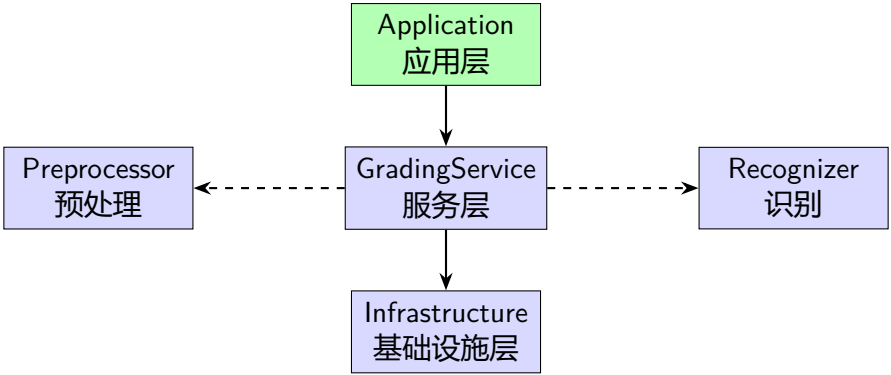
复用测试数据：

```
@pytest.fixture
def sample_image():
    """创建标准测试图像"""
    return np.ones((100, 100, 3), dtype=np.uint8) * 255

@pytest.fixture
def sample_regions():
    """创建标准测试区域"""
    return [
        Rectangle(10, 10, 20, 20),
        Rectangle(40, 10, 20, 20),
        Rectangle(70, 10, 20, 20),
    ]
```

系统架构实现

智能阅卷系统架构：



开发顺序：

1. 先实现基础设施层（图像加载、文件操作）
2. 再实现核心功能层（预处理、检测、识别）

核心功能实现：预处理模块

```
class ImagePreprocessor:
    """图像预处理模块"""

    def __init__(self, config: PreprocessConfig):
        self.config = config
        self.logger = setup_logging(self.__class__.__name__)

    def load(self, path: str) -> np.ndarray:
        """加载图像"""
        self.logger.info(f"加载图像: {path}")
        img = cv2.imread(path)
        if img is None:
            raise ImageLoadError(f"无法加载: {path}")
        return img
```

核心功能实现：识别模块

```
class ChoiceRecognizer:
    """选择题识别模块"""

    def __init__(self, config: RecognitionConfig):
        self.config = config
        self.logger = setup_logging(self.__class__.__name__)

    def recognize(self, image: np.ndarray,
                  options: List[str]) -> Optional[str]:
        """识别选择题答案"""
        if image is None:
            raise ValueError("图像不能为空")

        self.logger.debug(f"识别选项: {options}")
```

系统组装与集成

```
class GradingService:
    """阅卷服务"""

    def __init__(self, config: SystemConfig):
        self.prep = ImagePreprocessor(config.preprocess)
        self.detector = RegionDetector(config.detect)
        self.recognizer = ChoiceRecognizer(config.recognize)
        self.calculator = GradeCalculator(config.grade)
        self.logger = setup_logging(self.__class__.__name__)

    def process(self, image_path: str) -> GradingResult:
        """处理答题卡"""
        try:
            # 加载
```

单元测试与集成测试

单元测试：

- 测试各模块独立功能
- 使用 Mock 隔离依赖
- 快速、稳定、可重复

集成测试：

- 测试模块间协作
- 验证接口契约
- 使用真实或测试数据

测试策略

- **Preprocessor**: 单元测试，Mock cv2 函数
- **Recognizer**: 单元测试 + 参数化测试
- **GradingService**: 集成测试，验证完整流程
- **端到端**: 测试真实图像文件处理

模块 03-Live Coding：要点速查

重构技术

- 提取函数（长函数 → 多个小函数）
- 提取类（职责分离）
- 参数对象（过多参数 → 配置类）
- 替换魔法数字（有意义的常量）

异常处理策略

- 自定义异常层次
- 捕获底层异常 → 转换业务异常
- 防御式编程：前置条件检查

TDD 循环

现场练习：识别 Bug

任务：找出以下代码中的 Bug（3 分钟）

def calculate$_density(regions) : results = []for region in regions : if region is not None : density = sum(region)/len(region) results.append(density) return results$

使用 img = None result = calculate$_density([img]) print(result)$

提示：运行这段代码会发生什么？

Bug 分析：

- None 对象无法参与运算
- 应在循环内检查，而非循环前
- 缺少异常处理

现场练习：完善代码

任务：完善以下代码，添加错误处理（5 分钟）

待完善代码

def recognize$_c hoice(image, options) : results = foroptioninoptions : region = image[option]density = np.sum(region > 127)/region.sizeresults[option] = densityreturnresults$

需要处理的情况：

- image 为 None
- option 不在 image 中
- region 为空

参考解答要点： if image is None: raise ValueError(...) if option not in image: raise KeyError(...) if region.size == 0: continue 或 return

案例 1：小型阅卷系统开发

项目背景：

- 范围：单班级，30-50 份试卷
- 题型：选择题（20 题）+ 判断题（10 题）
- 时间：2 周开发周期
- 团队：3 名学生

架构设计：

- 单机版 Python 应用
- OpenCV + NumPy 处理图像
- CSV 格式存储结果
- 命令行界面

经验总结：

- 先做原型验证核心算法
- 用真实试卷数据测试

案例 2：企业级阅卷系统开发

项目背景：

- 范围：全校考试，单次 10,000+ 份试卷
- 题型：选择题、判断题、填空题、简答题
- 时间：3 个月开发周期
- 团队：8 人（前后端、算法、测试）

架构设计：

- 微服务架构（识别服务、评分服务、存储服务）
- 消息队列处理批量任务
- 分布式存储（图片、结果）
- Web 管理界面 + REST API

案例对比与启示

维度	小型系统	企业级系统
架构复杂度	单体应用	微服务
开发周期	2 周	3 个月
团队规模	3 人	8 人
代码规范	基本规范	严格规范 + 审查
测试覆盖	关键功能测试	全面测试 + 自动化
部署方式	手动运行	CI/CD 自动化
监控运维	日志查看	完整监控告警

关键启示：

- 规模决定架构，不要过度设计
- 小型项目也要考虑扩展性
- 自动化是规模化的关键

性能瓶颈诊断

问题：处理一张试卷需要 30 秒

诊断过程：

1. **定位瓶颈**：使用 cProfile 发现 75% 时间在 OCR
2. **分析原因**：每次调用都重新加载模型
3. **解决方案**：模型单例化，只加载一次
4. **优化效果**：处理时间从 30s 降到 3s

```
# 优化前：每次创建新实例
def recognize_text(image):
    ocr = PaddleOCR()  # 耗时操作
    return ocr.ocr(image)

# 优化后：单例模式
_ocr_instance = None
def get_ocr():
    global _ocr_instance
    if _ocr_instance is None:
        _ocr_instance = PaddleOCR()
```

内存泄漏排查

问题：批量处理时内存持续增长，最终崩溃

诊断过程：

1. 使用 `tracemalloc` 跟踪内存分配
2. 发现图像数组未释放
3. 确认是循环中加载了大量高分辨率图片

解决方案：

```python
def process_batch(image_paths):
    for path in image_paths:
        # 使用上下文管理器确保资源释放
        with ImageProcessor(path) as processor:
            result = processor.process()
            save_result(result)
        # 显式垃圾回收
        if processed_count % 100 == 0:
            gc.collect()
```

并发访问问题

问题：多线程处理时出现随机错误

根本原因：

- OpenCV 部分函数非线程安全
- 共享状态被多个线程修改
- 资源竞争导致数据损坏

解决方案：

```python
from concurrent.futures import ProcessPoolExecutor
import multiprocessing as mp

def process_images_parallel(image_paths, max_workers=4):
    """使用进程池而非线程池"""
    # OpenCV更适合多进程
    with ProcessPoolExecutor(max_workers=max_workers) as executor:
        results = list(executor.map(process_single, image_paths))
    return results

# 或者使用队列+工作进程模式
```

代码重构挑战

挑战 1：改进这个函数

```
def process(data):
    if data is not None:
        if len(data) > 0:
            result = []
            for i in range(len(data)):
                if data[i] > 0:
                    result.append(data[i] * 2)
            if len(result) > 0:
                return sum(result) / len(result)
    return 0
```

问题：

1. 这段代码有哪些坏味道？
2. 如何重构使其更清晰？

Bug 诊断 Quiz

下面的代码有什么问题?

```
class Recognizer:
    def __init__(self):
        self.threshold = 0.5
        self.results = []  # 注意这里!

    def recognize(self, image):
        result = self._process(image)
        self.results.append(result)
        return result

# 使用
rec1 = Recognizer()
rec2 = Recognizer()
rec1.recognize(img1)
print(rec2.results)  # 竟然也有数据? !
```

答案: 可变默认参数陷阱! `self.results = []` 在类定义时创建, 所有实例共享同一列表。

最佳实践投票

以下哪种做法更好?

选项 A:

```
def process(image):
    try:
        return recognize(image)
    except:
        return None
```

选项 B:

```
def process(image):
    if image is None:
        raise ValueError("图像不能为空")
    try:
        return recognize(image)
    except RecognitionError as e:
        logger.error("识别失败: %s", e)
        raise ProcessingError("处理失败") from e
```

经验分享讨论

讨论话题：

1. **最难忘的 Bug**：你遇到最难排查的问题是什么？
2. **重构经历**：是否有过"拯救烂代码"的经历？
3. **调试技巧**：有什么独门调试技巧？
4. **工具推荐**：最喜欢的开发工具是什么？

分享格式：

- 问题描述（1 分钟）
- 解决过程（2 分钟）
- 经验教训（1 分钟）

课堂 Quiz 1：识别 Code Smell

以下函数有什么问题？（单选）

代码

```
def p(img, thresh=127, mode=1, debug=False):
    处理图片 50 行代码...  如果调试模式 if debug: print(" 处理中...") return result
```

选项

A. 语法错误
B. 参数太多
C. 函数名不规范
D. 以上都是

vspace0.5cm
答案：D

(函数名 p 应改为 process_image，参数过多应封装为配置类)

课堂 Quiz 2：异常处理

以下异常处理方式最好的是?

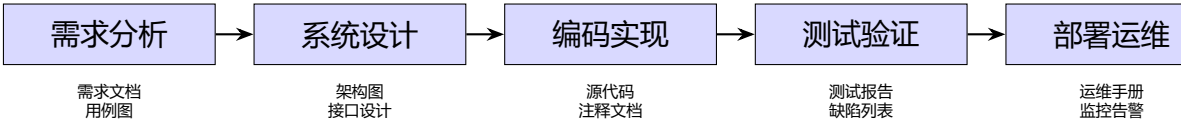
选项 A try: f() except: pass

选项 B try: f() except ValueError as e: logger.error(e) raise except Exception: logger.exception()

答案：B

原因：1. 不捕获所有异常（避免掩盖 bug）
2. 记录错误信息
3. 重新抛出异常让上层处理

系统开发完整流程



核心原则：

- 需求驱动，文档先行
- 迭代增量，持续交付
- 测试保护，重构无忧

调试技术速查表

问题定位方法

- 二分法：快速缩小范围
- 假设验证：科学排查思路
- 最小复现：剥离干扰因素
- 日志追踪：生产环境必备

调试黄金法则：

1. 先复现，再定位
2. 先假设，再验证
3. 先理解，再修复
4. 先测试，再提交

调试工具选择

- IDE 调试器：单步、断点、变量
- pdb/ipdb：命令行调试
- print/logging：简单直接
- 性能分析：cProfile、line_profiler

测试类型	粒度	速度	范围
单元测试	函数/类	极快（毫秒）	单个组件
集成测试	模块间	较快（秒）	接口交互
端到端测试	完整流程	慢（分钟）	用户场景

测试金字塔原则：

- 大量单元测试（70%）
- 适量集成测试（20%）
- 少量端到端测试（10%）

常见问题解决方案汇总

问题类型	解决方案
性能瓶颈	分析热点、算法优化、缓存、异步处理
内存泄漏	检查循环引用、及时释放资源、gc 监控
并发问题	锁机制、线程安全、进程隔离
识别不准	调参、数据增强、模型优化
模块集成失败	检查接口、版本兼容、依赖管理
测试不稳定	隔离依赖、Mock 外部服务、固定随机种子

任务层级（可选路径）

三条路径，适合不同基础的同学：

观察者路径（基础 60 分）

- 理解代码架构
- 运行框架代码
- 调整参数功能

适合：
编程基础较弱

使用者路径（进阶 +20 分）

- 补充模块代码
- 编写单元测试
- 优化准确率 80%+

适合：
有编程基础

创造者路径（挑战 +20 分）

- 设计新功能
- AI 辅助重构
- 性能优化

适合：编程能力较强

本周核心任务

所有路径必须完成（P0 任务）

1. **完成模块开发**（至少选择题 + 判断题）
2. **完成模块集成**
3. **通过基本功能测试**

各路径额外任务：

- **观察者**：完成基础任务即可获得 60 分
- **使用者**：补充核心模块 + 测试，额外获得 20 分
- **创造者**：新功能 + 重构 + 优化，额外获得 20 分

作业详细要求

代码质量要求：

- 函数长度不超过 50 行
- 每个函数有清晰文档字符串
- 关键逻辑有注释说明
- 命名清晰，见名知意

测试要求：

- 核心函数有单元测试
- 测试覆盖正常和异常情况
- 提供测试运行说明

提交格式：

- 源码压缩包（包含 README）
- 测试样例和预期结果
- 运行演示视频（可选加分）

评分项	分值
功能完整性	40 分
代码质量	20 分
测试覆盖	15 分
文档完整性	15 分
演示效果	10 分
总分	**100 分**

加分项：

- 使用 TDD 开发（+5 分）
- 性能优化有数据支撑（+5 分）
- 额外的错误处理（+3 分）

推荐学习资源

书籍推荐：

- 《代码大全》- Steve McConnell（软件工程圣经）
- 《重构》- Martin Fowler（重构技术权威）
- 《代码整洁之道》- Robert C. Martin（Clean Code）
- 《Python 编程：从入门到实践》- 基础入门

在线资源：

- Python 官方文档：https://docs.python.org/zh-cn/3/
- pytest 文档：https://docs.pytest.org/
- Real Python：https://realpython.com/（高质量教程）
- 廖雪峰 Python 教程（中文入门）

下节预告

第 11 周：成果展示与总结

每组 5 分钟演示 + 2 分钟答辩

展示内容：

- 系统功能演示
- 技术架构介绍
- 开发过程分享
- 问题与解决方案

准备好你们的展示！

加油！

最后一周冲刺

让系统真正跑起来！

前置知识自检

学习本课程前，请确认已掌握以下知识：

Python 基础

- 变量、数据类型、运算符
- 条件判断（if/elif/else）
- 循环结构（for/while）
- 函数定义与调用
- 类与对象基础

文件操作

- 读取/写入文本文件
- 路径处理（os.path）

版本控制基础

- Git 基本概念（仓库、提交）
- 基本命令（add/commit/push）
- 分支概念（可选）

数学基础

- 矩阵/数组基本概念
- 百分比计算
- 集合论基础（可选）

如需补充学习

三条学习路径，适合不同基础的你：

观察者：重点学习模块 1-2，直接运行代码示例

使用者：完成全部模块，重点实践模块 3-4

创造者：深入学习全部内容，完成模块 4 挑战任务

;

核心术语速查

术语	含义
模块化	将系统分解为独立、可复用的部分
重构	不改变功能，优化代码内部结构
高内聚低耦合	模块内部紧密相关，模块间依赖最小
TDD	测试驱动开发：先写测试，再写代码
Mock	模拟对象，隔离外部依赖进行测试
Code Smell	代码中的设计问题指示信号
断点调试	在指定行暂停，检查程序状态

概念补充：什么是重构?

生活类比

整理房间

- 房间很乱（代码可读性差）
- 不改变房间里的东西（功能不变）
- 只是重新整理收纳（优化结构）
- 目的是更容易找到东西（更容易维护）

代码示例

重构前

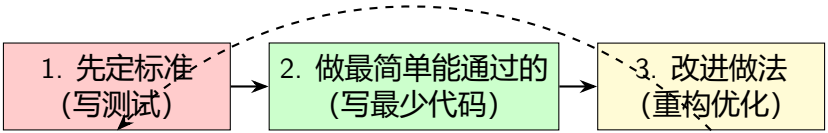
```
def f(x):
## 100行代码
```

重构后

```
def process():
load()
process_data()
save()
```

概念补充：什么是测试驱动开发？

想象学做菜的过程：



好处

- 确保做出来的东西符合要求（测试通过）
- 避免过度设计（只写最少代码）
- 持续改进质量（每次都优化一点）

课堂 Quiz 1：识别 Code Smell

以下函数有什么问题？（单选）

代码

答案：D

(函数名 p 应改为 process$_{image}$)

课堂 Quiz 2：异常处理

以下异常处理方式最好的是?

选项 A try: f() except: pass

选项 B try: f() except ValueError as e: logger.error(e) raise except Exception: logger.exception()

答案：B

原因：1. 不捕获所有异常（避免掩盖 bug）
2. 记录错误信息
3. 重新抛出异常让上层处理

现场练习：识别 Bug

任务：找出以下代码中的 Bug（3 分钟）

def calculate$_density(regions) : results = []for region in regions : if region is not None : density = sum(region)/len(region) results.append(density) return results$

使用 img = None result = calculate$_density([img]) print(result)$

提示：运行这段代码会发生什么？

Bug 分析：

- None 对象无法参与运算
- 应在循环内检查，而非循环前
- 缺少异常处理

现场练习：完善代码

任务：完善以下代码，添加错误处理（5 分钟）

待完善代码

def recognize$_c hoice(image, options) : results = foroptioninoptions : region = image[option]density = np.sum(region > 127)/region.sizeresults[option] = densityreturnresults$

需要处理的情况：

- image 为 None
- option 不在 image 中
- region 为空

参考解答要点： if image is None: raise ValueError(...) if option not in image: raise KeyError(...) if region.size == 0: continue 或 return

模块 00-背景知识：要点速查

核心概念

- **SDLC**：软件开发生命周期（需求 → 设计 → 编码 → 测试 → 部署）
- **敏捷开发**：迭代增量，快速反馈（2-4 周 Sprint）
- **CI/CD**：持续集成与持续部署（自动化构建、测试、发布）
- **Code Review**：代码审查，对事不对人，小步快跑
- **重构**：不改变外部行为，改善内部结构

关键公式

- **高内聚** = 模块内元素紧密相关
- **低耦合** = 模块间依赖最小

下一步学习

模块 01-核心理论：要点速查

设计原则

- **SOLID 原则**：单一职责、开闭原则、里氏替换、接口隔离、依赖倒置
- **封装**：隐藏实现细节，暴露必要接口
- **继承与多态**：代码复用，灵活扩展
- **纯函数**：相同输入 $\rightarrow$ 相同输出，无副作用

调试流程

发现问题 → 复现问题 → 建立假设 → 验证假设 → 修复问题

测试金字塔

大量单元测试 (70%) + 适量集成测试 (20%) + 少量 E2E 测试 (10%)

模块 02-工具环境：要点速查

开发工具选择

- 大型项目：PyCharm
- 多语言/轻量：VS Code
- 数据分析：Jupyter

代码质量工具

- 检查：flake8、pylint
- 格式化：black
- 类型检查：mypy

AI 编程工具

- Cursor：AI 原生 IDE（Cmd+L 对话）
- Claude Code：命令行助手
- Copilot：代码补全

模块 03-Live Coding：要点速查

重构技术

- 提取函数（长函数 → 多个小函数）
- 提取类（职责分离）
- 参数对象（过多参数 → 配置类）
- 替换魔法数字（有意义的常量）

异常处理策略

- 自定义异常层次
- 捕获底层异常 → 转换业务异常
- 防御式编程：前置条件检查

TDD 循环