第 9 周：系统架构与分组开发

把所有模块组合起来

北京石油化工学院\人工智能研究院\王文通

北京石油化工学院
BEIJING INSTITUTE OF PETROCHEMICAL TECHNOLOGY

北京石油化工学院
人工智能研究院

2025-2026 学年

课程概览

本周内容：

- 项目回顾：已学模块快速串联
- 系统架构设计：分层架构与管道模式
- 模块接口设计规范
- 图像处理流水线实战
- AI 辅助编程工具实战
- 团队协作与 Git 工作流
- 调试工具与项目框架

Live Coding 实战：

1. 搭建项目框架（AI 辅助）
2. 实现图像处理流水线
3. Git 分支管理与协作
4. 调试工具函数使用
5. 团队分组与任务分配

本周目标

搭建开发环境，完成 70% 项目框架，开始模块开发

预备知识

课前 5 分钟视频

观看：软件架构基础、Python 面向对象编程、Git 基本操作

本周需要的前置知识：

- **软件工程基础**：什么是架构、模块化思想
- **Python 面向对象**：类、继承、接口概念
- **Git 基础**：add、commit、push 操作

没有基础？

- 课前观看预备知识视频
- 课堂上先观察架构图
- 小组讨论时向组员请教
- 课后查阅推荐资源

不同背景的学习建议：

专业背景	学习建议

本周时间分配（135 分钟 = 3 学时）

第 1 学时（45 分钟）：

00:05 项目回顾（5min）

00:15 架构设计概述（10min）

00:25 分层架构与管道模式（10min）

00:40 Live Coding：搭建项目框架（15min）

00:45 小组讨论：确定分工（5min）

课间休息（10 分钟）：

00:55 休息与交流

第 2 学时（40 分钟）：

01:15 Live Coding：流水线实现（20min）

01:30 实践环节：学生跟随编写（15min）

第 3 学时（40 分钟）：

01:35-01:50 AI 辅助编程演示（15min）

01:50-02:05 分组任务分配（15min）

02:05-02:10 调试工具快速演示（5min）

02:10-02:15 总结与作业（5min）

课后延伸

- Git 工作流：观看视频教程（课后）
- 案例分析：阅读材料（课后）
- 代码规范：参考文档（课后）

时间控制提示

本周分组策略

分组原则：

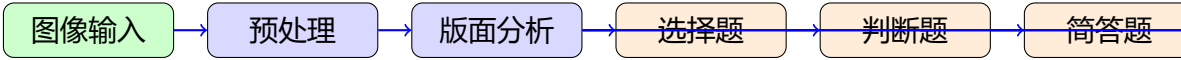
- 每组 3-4 人
- 包含不同专业背景
- 设立组长和技术负责人

角色分工：

角色	职责	适合
组长	统筹协调、进度管理	组织能力强的
技术负责人	架构设计、核心算法	CS/EE 专业
模块开发 A	选择题 + 判断题实现	有编程基础的
模块开发 B	简答题 + 评分实现	有编程基础的

本周协作任务

智能阅卷系统全景



本周任务：将所有模块整合，构建完整系统

已学模块快速回顾

基础模块（第 1-4 周）：

- 图像基础：像素、通道、颜色空间
- 预处理：去噪、增强、二值化
- 几何变换：透视矫正
- 版面分析：边缘检测、轮廓提取

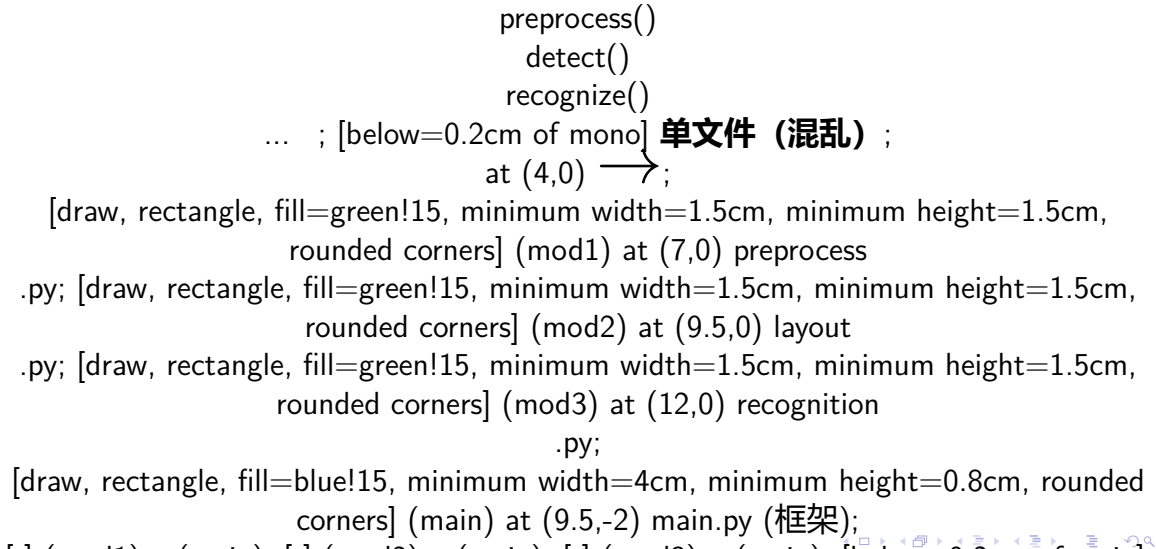
识别模块（第 5-8 周）：

- 选择题：填涂检测、像素密度
- 判断题：符号识别、模板匹配
- 简答题：OCR 识别、PaddleOCR

本周目标

将散落的模块组装成完整的智能阅卷系统

从” 单文件” 到” 模块化”:


preprocess()
detect()
recognize()
... ; [below=0.2cm of mono] 单文件（混乱）;
at (4,0) ⟶;
[draw, rectangle, fill=green!15, minimum width=1.5cm, minimum height=1.5cm, rounded corners] (mod1) at (7,0) preprocess
.py; [draw, rectangle, fill=green!15, minimum width=1.5cm, minimum height=1.5cm, rounded corners] (mod2) at (9.5,0) layout
.py; [draw, rectangle, fill=green!15, minimum width=1.5cm, minimum height=1.5cm, rounded corners] (mod3) at (12,0) recognition
.py;
[draw, rectangle, fill=blue!15, minimum width=4cm, minimum height=0.8cm, rounded corners] (main) at (9.5,-2) main.py (框架);
[-] (mod1) – (main); [-] (mod2) – (main); [-] (mod3) – (main); [below=0.2cm of main]

什么是软件架构?

定义 (软件架构)

软件架构是系统的高级结构，包括软件元素、元素的外部可见属性，以及元素之间的关系。

架构的核心要素:

- **组件 (Components)**: 系统的功能单元，如模块、服务、类库
- **连接器 (Connectors)**: 组件之间的交互机制，如 API 调用、消息传递、事件
- **配置 (Configuration)**: 组件和连接器的拓扑结构，即系统的组织方式
- **约束 (Constraints)**: 设计和实现的规则，如技术选型、性能要求、安全规范

架构 vs 设计

架构关注" 系统由什么组成" 和" 它们如何交互"; 设计关注" 如何实现每个组件"。架构是蓝图，设计是施工图。

架构决策的层次

各层次决策内容：

- 业务架构：确定系统要解决什么问题，业务流程如何
- 数据架构：确定数据如何组织、存储、流转
- 应用架构：确定系统如何分解为可管理的模块
- 技术架构：确定使用什么技术栈，如何部署

架构设计的重要性

好的架构带来：

- 易于理解和维护
- 便于团队协作
- 支持系统演化
- 降低技术风险
- 提高开发效率
- 支持快速迭代

糟糕的架构导致：

- 技术债务累积
- 修改牵一发而动全身
- 团队协作困难
- 系统难以扩展
- 维护成本高昂
- 新人难以入手

架构决策的影响

架构决策一旦做出，后期修改成本极高。前期投入时间设计架构是值得的。记住：修房子时改地基比建好后拆墙成本高得多。

架构设计原则：SOLID

原则	名称	核心思想
S	单一职责	一个类只负责一件事
O	开闭原则	对扩展开放，对修改关闭
L	里氏替换	子类可以替换父类
I	接口隔离	客户端不应该依赖它不需要的接口
D	依赖倒置	依赖抽象，而非具体实现

实际应用

- SRP：图像预处理类只负责预处理，不负责识别
- OCP：新增识别算法时，不修改现有代码
- LSP：所有识别器可以互换使用
- ISP：按需定义小接口

架构设计原则：KISS & DRY

KISS 原则

Keep **I**t **S**hort and **S**imple

- 简单优于复杂
- 避免过度设计
- 选择最直接的解决方案
- 代码应易于理解
- 警惕" 聪明的" 代码

DRY 原则

Don't **R**epeat **Y**ourself

- 消除重复代码
- 单一事实来源
- 变化只需修改一处
- 提高可维护性
- 抽象重复逻辑

简单是终极的复杂——达芬奇

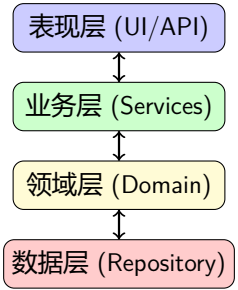
架构风格分类

架构风格	特点	适用场景
单体架构	所有功能在一个应用中	小型项目、快速原型
分层架构	按层次组织	企业应用、Web 应用
微服务架构	服务拆分、独立部署	大型系统、团队协作
事件驱动	基于事件通信	实时系统、流处理
管道-过滤器	数据流处理	数据处理、编译器

选择建议

没有最好的架构，只有最适合的架构。根据团队规模、项目复杂度、维护周期选择。对于我们的智能阅卷系统，建议采用分层 + 管道过滤器混合架构。

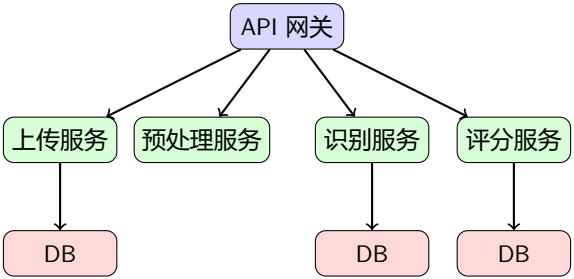
架构风格：分层架构详解



分层架构的优点：

- 关注点分离，职责清晰
- 易于测试（每层可独立测试）
- 便于维护（修改一层不影响其他层）
- 支持替换（如更换数据库）
- 新人容易理解代码结构

架构风格：微服务架构



微服务特点：

- 每个服务独立部署、独立扩展
- 服务间通过 API 通信
- 每个服务可以有独立的数据库
- 技术选型可以不同
- 适合大型团队

架构风格：管道-过滤器架构

管道-过滤器特点：

- 数据流驱动，单向流动
- 每个过滤器独立，可复用
- 易于扩展，新增过滤器
- 支持并行处理
- 天然适合图像处理流程

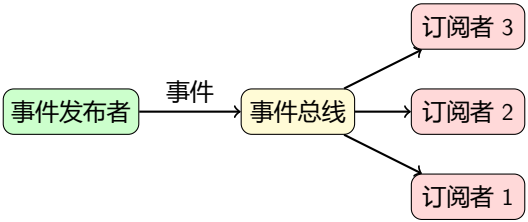
我们的应用：

图像输入 → 去噪 → 增强 → 二值化 → 矫正 → 版面分析

架构风格：事件驱动架构

定义 (事件驱动架构)

系统组件通过发布和订阅事件进行通信，实现松耦合的交互模式。



适用场景：

- 异步处理：如图像处理完成后通知评分模块
- 解耦系统：发布者和订阅者互不感知
- 多端同步：Web、移动端同时接收更新
- 审计追踪：所有事件可记录

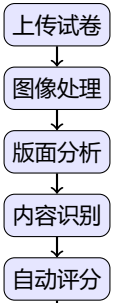
智能阅卷系统的业务场景

主要用户：

- 教师：上传试卷、查看结果、导出成绩
- 学生：查看成绩、提交复查申请
- 管理员：系统管理、用户管理

用户痛点：

- 人工阅卷耗时费力
- 容易出现人工错误
- 统计报表生成麻烦
- 无法追溯阅卷过程



系统功能需求分析

核心功能：

- 图像导入与格式转换
- 图像预处理（去噪、矫正）
- 答题卡区域定位
- 选择题自动识别
- 判断题符号识别
- 简答题文字识别
- 答案比对与评分
- 成绩统计与导出

辅助功能：

- 标准答案管理
- 人工复核接口
- 识别结果校对
- 批量处理
- 历史记录查询
- 权限管理
- 日志记录

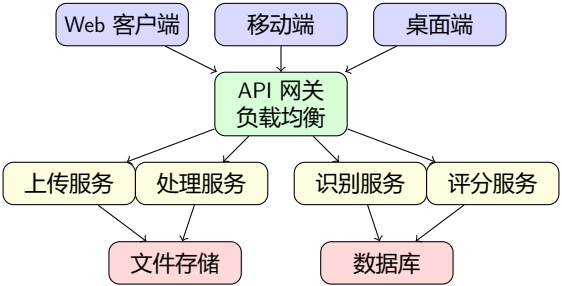
非功能性需求

需求类型	具体要求	实现策略
性能	单张试卷 <3 秒	算法优化、并行处理、缓存
准确率	选择题 >99%	多算法融合、人工复核机制
可扩展性	支持新题型	插件化架构、模块化设计
可用性	7×24 小时	容错设计、异常恢复
安全性	数据加密	传输加密、访问控制
可维护性	易于升级	清晰文档、单元测试
可用性	简单易用	清晰 UI、完善帮助

性能要求详解

- 图像预处理：<500ms/张
- 区域定位：<300ms/张
- 识别响应：<1.5s/题

典型智能阅卷系统架构案例



这是一个典型的分层微服务架构，支持水平扩展和高可用。

案例对比分析

维度	小型系统	中型系统	大型系统
用户规模	<1000	1000-10 万	10 万 +
试卷量级	10 份/天	1 万份/天	100 万份/天
架构风格	单体分层	微服务	分布式微服务
部署方式	单机部署	集群部署	云原生
识别算法	模板匹配	机器学习	深度学习
扩展性	低	中	高

选型建议

学生课程项目建议采用：单体分层架构 + 管道过滤器模式足够简单支撑当前需求，同时为未来扩展预留空间。

敏捷开发方法论简介

定义 (敏捷开发)

敏捷开发是一种以人为核心、迭代、循序渐进的软件开发方法。强调快速交付、持续改进和适应变化。

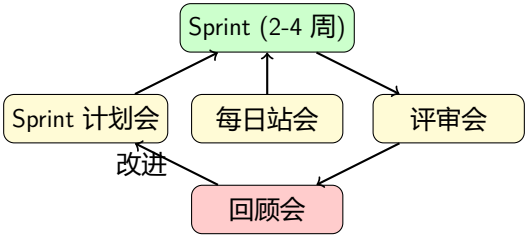
敏捷宣言四大价值观：

1. 个体和互动高于流程和工具
2. 工作的软件高于详尽的文档
3. 客户合作高于合同谈判
4. 响应变化高于遵循计划

常用敏捷方法

- **Scrum**：迭代开发、每日站会、冲刺计划
- **Kanban**：可视化工作流、限制在制品

Scrum 核心概念



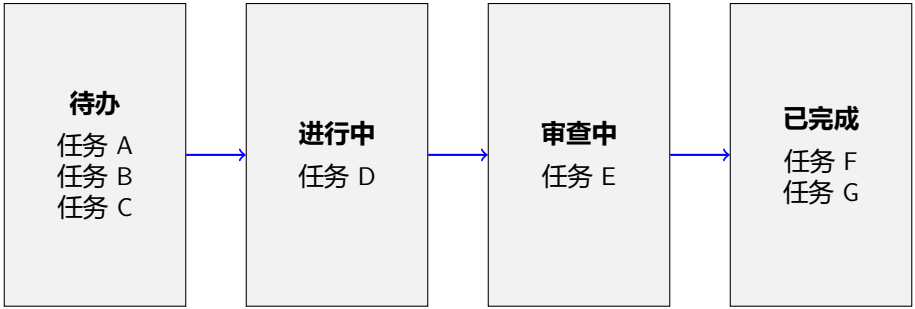
Scrum 角色：

- Product Owner：产品负责人，定义需求优先级
- Scrum Master：流程专家，保证流程执行
- Team：开发团队，执行开发工作

核心工件：

- Product Backlog：产品待办列表
- Sprint Backlog：冲刺待办列表
- Increment：可交付的增量

Kanban 可视化看板



Kanban 核心原则：

- 可视化工作流
- 限制在制品（WIP Limit）
- 管理流动
- 持续改进

团队协作工具与流程

版本控制：

- Git：分布式版本控制
- GitHub/GitLab：代码托管
- 分支策略：Git Flow / GitHub Flow

沟通协作：

- Slack/飞书：即时通讯
- 腾讯会议：视频会议
- 石墨文档：协作文档

项目管理：

- Jira：企业级项目管理
- Trello：轻量级看板
- Notion：知识库 + 任务管理

CI/CD 工具：

- GitHub Actions
- Jenkins
- GitLab CI

代码规范与版本控制

Python 代码规范（PEP 8）：

- 缩进：4 个空格（不要用 Tab）
- 行宽：每行不超过 79 字符
- 命名：模块小写、类驼峰、函数下划线
- 注释：文档字符串、行内注释
- 导入：每行一个导入、按标准库/第三方/本地分组

Git 提交规范

- feat：新功能
- fix：修复 bug
- docs：文档更新
- style：格式调整
- refactor：重构
- test：测试相关

团队沟通与冲突解决

有效沟通原则

- **清晰表达**：明确目标、背景、期望
- **主动同步**：定期汇报进度、暴露问题
- **倾听反馈**：理解他人观点、接受建议
- **文档留痕**：重要决策书面记录

常见冲突及解决：

冲突类型	解决策略
技术方案分歧	数据说话、原型验证、投票决定
任务分配不均	明确角色、轮换任务、互相帮助
代码风格不一致	统一规范、自动化检查、互相审查
进度延误	及时沟通、调整计划、寻求帮助
资源竞争	优先级排序、错峰使用

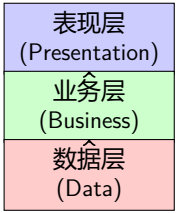
分层架构的概念与优势

定义 (分层架构)

将系统按职责划分为多个层次，每层只与相邻层交互，上层依赖下层，下层不依赖上层。

经典三层架构：

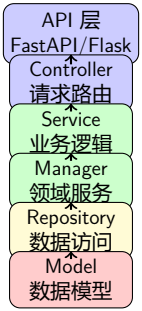
1. **表现层**：UI、API 接口、用户交互
2. **业务层**：业务逻辑、业务规则
3. **数据层**：数据访问、持久化存储



分层架构的优势：

- 关注点分离，职责清晰
- 易于测试（每层可独立测试）
- 便于维护（修改一层不影响其他层）

分层架构的详细设计



各层职责：

- API/Controller：请求接收、参数校验、响应格式化
- Service：编排业务逻辑、事务管理
- Manager/Domain：核心业务规则
- Repository：数据 CRUD 操作
- Model：数据结构定义

层间通信与依赖管理

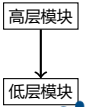
依赖方向：

- 依赖只能从上到下
- 上层调用下层接口
- 下层通过回调/事件通知上层

依赖倒置原则（DIP）应用：

不好的做法：

- 上层直接依赖下层实现
- 紧耦合
- 难以替换实现



好的做法：

- 依赖抽象接口
- 低层实现接口
- 灵活替换

分层架构的适用场景

适用场景	说明
企业应用	ERP、CRM 等业务系统，业务逻辑复杂
Web 应用	前后端分离的 Web 系统
桌面应用	具有复杂业务逻辑的单机软件
移动端应用	具有本地业务处理的 App
课程项目	小型到中型项目，快速开发

不适用场景

- 简单脚本工具（过度设计）
- 高性能计算（层间开销）

为什么选择管道-过滤器?

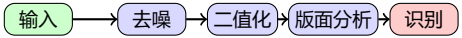
定义 (管道-过滤器架构)

数据从数据源流入，经过一系列处理组件（过滤器）转换，最终输出到数据汇。

非常适合图像处理的原因：

- 图像处理本质是" 流式" 转换
- 每个步骤独立可测试
- 易于组合和替换
- 天然支持并行处理

智能阅卷流水线示例：



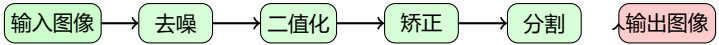
驱动，单向流动

过滤器组合与复用

组合方式：

1. **线性管道**：$A \rightarrow B \rightarrow C$
2. **分支管道**：$A \rightarrow B/C \rightarrow D$
3. **合并管道**：$A/B \rightarrow C$
4. **反馈管道**：输出反馈到输入

图像处理流水线示例：



复用策略：

- 将常用过滤器组合封装为"宏过滤器"
- 提供预定义的流水线模板
- 支持通过配置切换不同过滤器组合

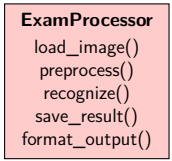
单一职责原则（SRP）

定义 (单一职责原则)

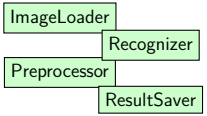
一个类应该只有一个引起它变化的原因。换句话说，一个类应该只负责一件事。

违反 SRP 的例子：

违反 SRP



改进方案：



- SRP 使得类更容易理解
- 修改只影响少量类
- 单元测试更容易编写

OCP 的好处：

- 新功能通过添加新代码实现
- 现有代码不会被意外破坏
- 系统可维护性大大提高

定义 (开闭原则)

软件实体应该对扩展开放，对修改关闭。

实现方式：

- 抽象基类 + 具体实现
- 策略模式
- 依赖注入
- 插件架构

里氏替换原则（LSP）

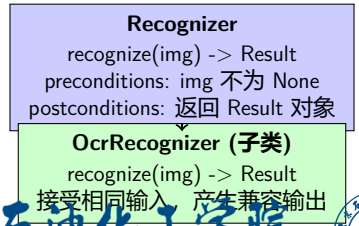
定义 (里氏替换原则)

子类型必须能够替换其基类型，而程序的行为不会改变。

关键要求：

- 子类不改变父类的前置条件（输入）
- 子类不强化父类的后置条件（输出）
- 子类保持父类的不变性

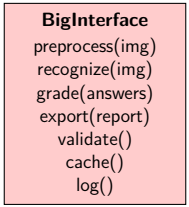
正确示例：



定义 (接口隔离原则)

客户端不应该被迫依赖它不使用的方法。应该将大接口拆分为小接口。

不好的设计：



改进设计：

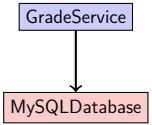
依赖倒置原则（DIP）

定义 (依赖倒置原则)

高层模块不应该依赖低层模块，两者都应该依赖抽象。抽象不应该依赖细节，细节应该依赖抽象。

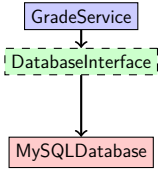
示例对比：

传统方式（违反 DIP）：



- 高层依赖低层实现
- 更换数据库需要修改高层

改进方式（遵循 DIP）：



- 依赖抽象接口
- 可轻松替换实现

定义 (工厂模式)

定义一个创建对象的接口，但让子类决定实例化哪个类。工厂方法让类将实例化推迟到子类。

使用示例：

```
class RecognizerFactory:
 @staticmethod
 def create_recognizer(type_name):
 if type_name == "choice":
 return ChoiceRecognizer()
 elif type_name == "judge":
 return JudgeRecognizer()
```

定义 (策略模式)

定义一系列算法，将每个算法封装起来，使它们可以互相替换。策略让算法独立于使用它的客户端。

应用场景：

- 多种识别算法切换
- 不同评分策略
- 多种输出格式

优点：

- 算法可以自由切换

定义 (观察者模式)

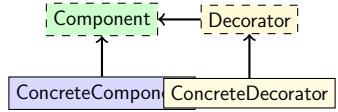
定义对象间的一种一对多依赖关系，当一个对象状态改变时，其所有依赖者都会收到通知并自动更新。

应用场景：

- 识别完成通知评分模块
- 处理进度更新 UI
- 日志记录事件

定义 (装饰器模式)

动态地给一个对象添加额外的职责，比生成子类更灵活。

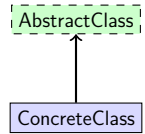


使用示例：

```
class LoggingDecorator(RecognizerInterface):
 def __init__(self, recognizer):
 self._recognizer = recognizer

 def recognize(self, image):
 print(f" 开始识别")
 result = self._recognizer.recognize(image)
```

定义 (模板方法模式)

在抽象类中定义算法的骨架，将某些步骤的具体实现延迟到子类。



示例：图像处理流程

```
class ImageProcessor:
 def process(self, image):
 image = self.load(image)
 image = self.preprocess(image)
 image = self.analyze(image)
 return self.format_result(image)
```

设计模式总结对比

类型	模式	解决的问题
创建型	工厂模式	对象创建逻辑复杂，类型需要运行时决定
	单例模式	需要确保某个类只有一个实例
	建造者模式	构建复杂对象，参数多，需分步骤构建
结构型	适配器模式	接口不兼容，需要适配
	装饰器模式	动态添加功能，不改变原有类
	代理模式	控制访问，添加代理逻辑
行为型	观察者模式	一对多通知，状态变化需广播
	策略模式	算法可互换，需要运行时切换
	模板方法	算法骨架固定，部分步骤可变

为什么选择管道-过滤器架构?

定义 (管道-过滤器架构)

数据从数据源流入，经过一系列处理组件（过滤器）转换，最终输出到数据池。

图像处理的特点：

- 数据流明确：输入 → 处理 → 输出
- 步骤独立：去噪、增强、分割可独立工作
- 可组合：不同场景需要不同处理组合
- 可复用：过滤器可在多个流程中使用

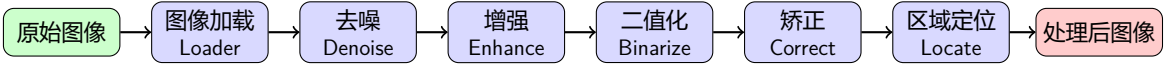
管道-过滤器的优势：

- 高内聚：每个过滤器职责单一
- 低耦合：过滤器间通过标准接口连接
- 易扩展：添加新过滤器不影响现有
- 易测试：每个过滤器可独立测试

适用场景

编译器、数据处理管道、音视频处理、图像处理、ETL 流程

流水线整体架构



设计要点：

- 每个过滤器职责单一
- 过滤器之间松耦合
- 支持动态组合
- 中间结果可追溯

数据流转设计

定义 (ImageData 数据类)

流水线中传递的数据封装，包含图像本身和处理过程中的元数据。

ImageData 结构

- `image: np.ndarray` - 原始/处理后图像
- `metadata: Dict` - 处理元数据（步骤、参数）
- `step_name: str` - 当前处理步骤
- `confidence: float` - 处理置信度

元数据内容：

- 处理步骤历史
- 每步参数配置
- 中间结果引用

过滤器接口设计

定义 (FilterInterface 抽象基类)

所有过滤器必须实现的接口规范。

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Any, Dict

@dataclass
class ImageData:
 """ 图像数据封装"""
 image: np.ndarray
 metadata: Dict[str, Any]

class FilterInterface(ABC):
 """ 过滤器接口"""

 @abstractmethod
 def process(self, data: ImageData) -> ImageData:
 """ 处理图像数据，返回处理后的数据"""
```

去噪过滤器实现

```
class DenoiseFilter(FilterInterface):
 """ 去噪过滤器：高斯滤波"""

 def __init__(self, kernel_size=5):
 self.kernel_size = kernel_size
 self.name = "DenoiseFilter"

 def get_name(self) -> str:
 return self.name

 def process(self, data: ImageData) -> ImageData:
 denoised = cv2.GaussianBlur(
 data.image,
 (self.kernel_size, self.kernel_size), 0
 )
 data.metadata['denoise_applied'] = True
 data.metadata['kernel_size'] = self.kernel_size
 return ImageData(denoised, data.metadata)

 def configure(self, params: Dict[str, Any]) -> None:
```

二值化过滤器实现

```
class BinarizeFilter(FilterInterface):
 """ 自适应二值化过滤器"""

 def __init__(self, method='adaptive', block_size=11):
 self.method = method
 self.block_size = block_size
 self.name = "BinarizeFilter"

 def process(self, data: ImageData) -> ImageData:
 gray = cv2.cvtColor(data.image, cv2.COLOR_BGR2GRAY)

 if self.method == 'adaptive':
 binary = cv2.adaptiveThreshold(
 gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
 cv2.THRESH_BINARY, self.block_size, 2
 )
 else:
 _, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)

 data.metadata['binarize_method'] = self.method
```

透视矫正过滤器实现

```
class PerspectiveCorrectionFilter(FilterInterface):
 """ 透视矫正过滤器"""

 def __init__(self, target_points=None):
 self.target_points = target_points
 self.name = "PerspectiveCorrectionFilter"

 def process(self, data: ImageData) -> ImageData:
 # 从元数据获取角点
 corners = data.metadata.get('corners')
 if corners is None:
 return data # 无需矫正

 # 计算透视变换矩阵
 h, w = data.image.shape[:2]
 target = np.float32([[0, 0], [w, 0], [w, h], [0, h]])
 matrix = cv2.getPerspectiveTransform(corners, target)

 # 应用变换
 corrected = cv2.warpPerspective(data.image, matrix, (w, h))
```

流水线执行与监控

```
def execute_with_monitoring(self, image: np.ndarray) -> Dict:
""" 带监控的执行"""
import time
data = ImageData(image, {'pipeline': self.name})
execution_log = []

for filter_obj in self.filters:
start_time = time.time()
data = filter_obj.process(data)
elapsed = time.time() - start_time

execution_log.append({
'filter': filter_obj.get_name(),
'time_ms': elapsed * 1000,
'image_shape': data.image.shape
})

return {
'result': data,
'execution_log': execution_log,
```

流水线使用示例

```
# 构建预处理流水线
pipeline = ImagePipeline("exam_preprocess")

pipeline \
 .add_filter(DenoiseFilter(kernel_size=5)) \
 .add_filter(BinarizeFilter(method='adaptive')) \
 .add_filter(PerspectiveCorrectionFilter())

# 执行处理
result = pipeline.execute(image)
processed_image = result.image

# 带监控执行
monitored = pipeline.execute_with_monitoring(image)
print(f" 总耗时: {monitored['total_time_ms']:.2f}ms")
for log in monitored['execution_log']:
 print(f" {log['filter']}: {log['time_ms']:.2f}ms")
```

中间结果缓存

```
class CachedPipeline(ImagePipeline):
 """ 带缓存的流水线"""

 def __init__(self, name: str = "cached", cache_dir: str = "./cache"):
 super().__init__(name)
 self.cache_dir = Path(cache_dir)
 self.cache_dir.mkdir(exist_ok=True)

 def _get_cache_key(self, image: np.ndarray, step: int) -> str:
 """ 生成缓存键"""
 import hashlib
 image_hash = hashlib.md5(image.tobytes()).hexdigest()
 return f"cache_{self.name}_step_{step}_image_{image_hash}.jpg"

 def execute(self, image: np.ndarray, use_cache: bool = True) -> ImageData:
 """ 支持缓存的执行"""
 # 检查缓存...
 # 命中则直接返回，否则执行并缓存
```

异常处理与恢复

```
class RobustPipeline(ImagePipeline):
 """ 带异常处理的流水线"""

 def execute(self, image: np.ndarray) -> ImageData:
 data = ImageData(image, {'pipeline': self.name})

 for i, filter_obj in enumerate(self.filters):
 try:
 data = filter_obj.process(data)
 except Exception as e:
 print(f" 过滤器 {filter_obj.get_name()} 失败: {e}")

 # 策略 1: 跳过当前过滤器
 if self.skip_on_error:
 continue

 # 策略 2: 使用备用过滤器
 if i in self.fallback_filters:
 fallback = self.fallback_filters[i]
 data = fallback.process(data)
```

过滤器组合模式

```
class CompositeFilter(FilterInterface):
 """ 组合过滤器：将多个过滤器视为一个"""

 def __init__(self, name: str):
 self.name = name
 self.filters: List[FilterInterface] = []

 def add(self, filter_obj: FilterInterface) -> 'CompositeFilter':
 self.filters.append(filter_obj)
 return self

 def process(self, data: ImageData) -> ImageData:
 for f in self.filters:
 data = f.process(data)
 return data

 def get_name(self) -> str:
 return self.name

# 使用：创建可复用的预处理组合
```

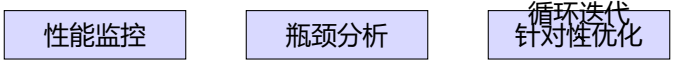
并行流水线

```
class ParallelPipeline(ImagePipeline):
 """ 支持并行处理的流水线"""

 def __init__(self, name: str = "parallel", max_workers: int = 4):
 super().__init__(name)
 self.max_workers = max_workers

 def execute_parallel(self, images: List[np.ndarray]) -> List[ImageData]:
 """ 并行处理多张图像"""
 from concurrent.futures import ThreadPoolExecutor

 with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
 results = list(executor.map(self.execute, images))

 return results

 def execute_filter_parallel(self, data: ImageData) -> ImageData:
 """ 过滤器内部并行（如分块处理）"""
 # 实现分块并行处理逻辑
 pass
```

流水线性能优化策略

优化方向：

1. **惰性计算**：跳过不必要的处理步骤
2. **缓存机制**：缓存中间结果，避免重复计算
3. **并行处理**：多图像/多区域并行
4. **算法优化**：选择更高效的算法
5. **资源复用**：复用滤波器内核、连接池等



性能指标：

- 吞吐量：每秒处理图像数
- 延迟：单张图像处理时间
- 资源占用：CPU、内存使用率

为什么需要模块化识别引擎?

- **问题 1：代码耦合严重**
  - 图像预处理与识别逻辑混在一起
  - 新增题型需要修改核心代码
  - 难以进行单元测试
- **问题 2：算法无法替换**
  - 固定使用某种边缘检测算法
  - 无法根据图像特点自适应选择

- **模块化解决思路**
  - 定义统一接口规范
  - 解耦预处理与识别逻辑
  - 支持运行时动态加载算法
- **设计目标**
  - 开闭原则：对扩展开放，对修改关闭
  - 依赖倒置：依赖于抽象而非具体

识别器接口定义

所有识别器必须实现统一接口：

识别器接口规范

- `detect_question_type(image) -> str`：检测图像中的题型
- `preprocess(image) -> image`：图像预处理
- `recognize(image) -> dict`：执行识别，返回结构化结果
- `get_confidence(result) -> float`：计算识别置信度

接口设计原则

1. 输入输出标准化：统一使用 NumPy 数组格式
2. 错误处理统一：返回空字典表示识别失败
3. 配置外部化：通过参数传入而非硬编码

选择题识别器实现

选择题识别流程

1. 题目区域定位：使用连通域分析找到题目块
2. 选项分割：投影法分离各选项区域
3. 填涂判断：计算选项内部填充比例
4. 结果生成：选择填充比例最大的选项

关键代码逻辑

```
如果填充比例 > 0.6 且方差 < 阈值:
    判定为已填涂
```

- 使用 Otsu 方法自适应确定二值化阈值
- 形态学闭操作填补小空洞
- 轮廓面积过滤去除噪点

判断题符号识别算法

判断依据

- 正确 (✓)：斜向线条组合
- 错误 (×)：交叉十字形

识别步骤

1. 霍夫变换检测直线
2. 分析直线角度分布
3. 计算交叉点位置
4. 输出判断结果

Algorithm 1 判断题符号识别

检测所有直线段存在两条交叉直线且夹角 $\in [60°, 120°]$ × 存在两条夹角 $< 45°$ 的直线 ✓

算法选择策略

自动算法选择机制

根据图像特征自动选择最优算法：

- **图像复杂度评估**：计算梯度幅值和边缘密度
- **噪声水平检测**：分析邻域像素差异
- **动态策略选择**
  - 噪声高：使用中值滤波 + Canny 边缘检测
  - 背景简单：直接使用 Otsu 二值化
  - 目标模糊：使用自适应阈值

策略模式实现

```
根据图像复杂度 _ 等级选择对应识别器
case 低: 选用简单二值化
case 中: 选用自适应阈值
```

结果融合与置信度评估

多识别器融合

当多个识别器给出结果时，采用加权投票策略：

- 每个识别器输出结果和置信度
- 加权平均计算综合置信度
- 置信度超过阈值则输出结果

置信度计算公式

综合置信度 = $\sum$（识别器置信度 × 权重）/ $\sum$ 权重

- 选择题识别器权重：0.8
- 判断题识别器权重：0.7
- 位置校验器权重：0.5

置信度阈值与降级策略

三级置信度阈值

高置信度 > 0.9 ： 直接输出，无需复核

中置信度 0.7-0.9 ： 输出但标记为待复核

低置信度 < 0.7 ： 触发降级策略或人工复核

降级策略

1. 更换备用识别算法重新识别
2. 尝试多角度/多阈值识别
3. 返回结果附带警告信息
4. 降级日志记录

关键阈值设置

引擎使用完整示例

引擎初始化与调用

1. 创建引擎实例并注册识别器
2. 加载配置参数
3. 传入图像执行识别
4. 获取并解析结果

标准调用流程

1. `engine = RecognitionEngine()`
2. `engine.register("choice", ChoiceRecognizer())`
3. `engine.register("judge", JudgeRecognizer())`
4. `result = engine.recognize(image)`
5. `if result.confidence > 0.7: print(result.value)`

性能监控与优化

关键性能指标

- 识别耗时：单张图像处理时间
- 内存占用：运行时的内存峰值
- 准确率：正确识别样本/总样本
- 召回率：检出目标/实际目标
- 识别成功率：成功返回结果/总调用
- 降级触发率：触发降级次数/总调用
- 置信度分布：各置信度区间的占比
- 算法切换频率：切换算法的次数

优化策略

- **预处理优化**：使用查表法替代复杂计算
- **识别器懒加载**：首次使用时才初始化

开发环境搭建

步骤 1：创建项目目录

```
mkdir auto_grading_system
```

```
cd auto_grading_system
```

```
git init
```

步骤 2：创建虚拟环境

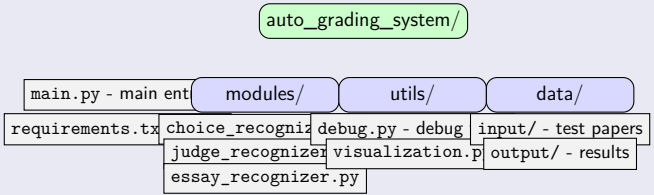
```
python -m venv venv
source venv/bin/activate # Linux/Mac
venv # Windows
```

步骤 3：安装依赖

```
pip install opencv-python paddlepaddle paddleocr numpy pillow
```

项目目录结构

智能阅卷系统完整结构



Course Materials

utils/debug.py and utils/visualization.py are provided (70% framework code)

Function: save_debug_image(image, name, output_dir="debug")

Purpose: Save debug images for troubleshooting

- Creates output directory automatically
- Adds timestamp to filename for uniqueness
- Saves as PNG format

Usage Example:

Function: draw_debug_boxes(image, boxes, labels, color)

Purpose: Visualize detection results on images

- Draws bounding boxes on the image
- Optional labels for each box
- Customizable BGR color

Usage Example:

Class: DebugPipeline - Pipeline Step Visualization

Key Methods:

- `_init_(output_dir="debug")` - Initialize with output directory
- `add_step(name, image, info)` - Add processing step
- `save_all_steps()` - Save all step images
- `visualize()` - Display all steps in windows

Usage Example:

```
pipeline.add_step("blur", blurred)
pipeline.add_step("edges", edges)

pipeline.save_all_steps();
```

Interface Contract

- Input: image (numpy.ndarray) or ROI region
- Output: RecognitionResult (text, confidence, question_type)
- Exception: Return result with confidence=0, no exceptions

Recognizer Interface Definition:

```
    def recognize(self, image, roi=None) -> RecognitionResult:
            """Perform recognition, return result"""
                                pass

       def get_supported_types(self) -> List[QuestionType]:
            """Return supported question types"""
pass ;
```

Data Structure Definition:

```
            self.text = text # Recognized text content
    self.confidence = confidence # 0.0-1.0, recognition confidence
            self.question_type = type # QuestionType enum
        self.bounding_box = None # Optional: ROI coordinates
            self.metadata = {} # Optional: extra info

                    def is_valid(self) -> bool:
                """Check if result is valid for grading"""
return self.confidence > 0.6 ;
```

AI 编程工具推荐

课程推荐工具（按优先级）：

Cursor - 首选

- AI 原生 IDE
- Ctrl+K 原地编辑
- Ctrl+L 上下文对话
- 最适合初学者

Claude Code

- 命令行 AI 助手
- 适合高级用户
- 强大的代码理解

通义灵码

- JetBrains 插件
- 国内访问友好
- 免费使用

Cursor 实战演示

Cursor 核心快捷键：

- `Ctrl+K` - 原地编辑：选中代码 → 按 Ctrl+K → 输入指令
- `Ctrl+L` - 上下文对话：询问代码逻辑、解释错误原因
- `Ctrl+I` - 生成新代码：描述需求 → 自动生成代码框架

AI Prompt 模板：RTF 框架

RTF 框架示例 - 生成接口代码：

Role-Task-Format

- **Role**: 你是一个 Python 架构师
- **Task**: 设计图像预处理流水线的 Filter 接口
- **Format**: 包含类定义、方法签名、类型注解、注释

思维链引导示例： 请一步步思考：首先确定输入输出数据结构，然后定义过滤器接口，最后实现流水线引擎

少样本提示示例： 请参考以下接口设计，为二值化过滤器创建类似的接口：

```
class DenoiseFilter:
def process(self, data: ImageData) -> ImageData:
```

AI 辅助调试三部曲

遇到错误时，这样问 AI：

1. **Traceback 复制**：粘贴完整的错误信息
2. **环境说明**：Python 版本、使用的库
3. **数据贴出**：关键变量值或输入数据

示例 Prompt

我遇到了这个错误：`TypeError: 'NoneType' object...`
环境：Python 3.10, OpenCV 4.8
代码在第 56 行，请帮我分析原因并修复

调试工具函数：save_debug_image

utils/debug.py

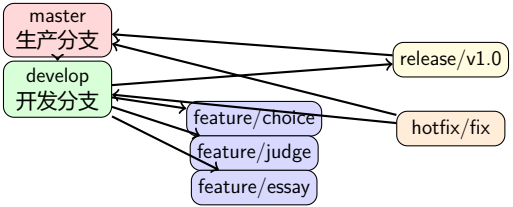
```
%============================================================
% 05_development.tex - 开发指导
%============================================================

\section{开发指导}

\begin{frame}{开发环境搭建}
    \begin{block}{步骤1: 创建项目目录}
        \begin{center}
            \begin{tikzpicture}[scale=0.8, transform shape]
                \node[draw, rectangle, fill=green!20] at (0,0) {\texttt{mkdir auto\_grading\_system}};
                \node[draw, rectangle, fill=blue!20] at (0,-1) {\texttt{cd auto\_grading\_system}};
                \node[draw, rectangle, fill=yellow!20] at (0,-2) {\texttt{git init}};
            \end{tikzpicture}
        \end{center}
    \end{block}

    \begin{block}{步骤2: 创建虚拟环境}
        \texttt{python -m v
```

调试工具函数：draw_debug_boxes

绘制调试框

```
   \end{block}
\end{frame}

\begin{frame}{项目目录结构}
   \begin{block}{智能阅卷系统完整结构}
      \begin{center}
         \begin{tikzpicture}[scale=0.65, transform shape,
            dir/.style={draw, rectangle, rounded corners, fill=blue!15, minimum width=3cm, minimum
                height=0.6cm, align=center},
            file/.style={draw, rectangle, fill=gray!10, minimum width=2.5cm, minimum height=0.5cm,
                align=left, font=\small},
            arrow/.style={->, thick}]

            \node[dir, fill=green!20] (root) at (5,3) {auto\_grading\_system/};

            \node[file] (main) at (0,1.5) {\texttt{main.py} - main entry};
            \node[file] (req) at (0,0.8) {\texttt{requirements.txt} - deps};

            \node[dir] (modules) at (3,-5) {modules/};
```

分支策略：Git Flow 详解



分支说明：

- **master**：生产分支，只接受合并，永远保持可发布状态
- **develop**：开发分支，功能集成分支
- **feature/***：功能分支，从 develop 创建
- **release/***：发布分支，准备上线
- **hotfix/***：紧急修复，从 master 创建

分支命名规范

分支类型	命名规范
功能分支	feature/模块-功能描述 如：feature/choice-recognizer
发布分支	release/v 版本号 如：release/v1.0.0
修复分支	hotfix/问题描述 如：hotfix/correct-bug-001
实验分支	dev/实验描述 如：dev/ocr-experiment

命名原则

- 使用小写字母
- 使用连字符（-）而非下划线

开发任务分解

高优先级（P0）- 本周必须完成：

选择题识别器

- ChoiceRecognizer 类框架
- 填涂区域检测
- 像素密度统计
- 置信度计算
- 单元测试

判断题识别器

- JudgeRecognizer 类框架
- 符号轮廓提取
- 形状特征分析
- √/× 分类
- 单元测试

中优先级（P1）- 下周完成：

- 简答题识别器（集成 PaddleOCR）
- 评分模块
- 模块集成测试

低优先级（P2）- 选做：

角色任务清单

角色	本周任务	交付物
组长	协调分工、进度跟踪、代码审查、主程序整合	项目周报
技术负责人	架构设计、技术难点攻关、代码审查	架构设计文档
模块开发 A	选择题识别器实现、单元测试	choice_recognizer.py
模块开发 B	判断题识别器实现、单元测试	judge_recognizer.py

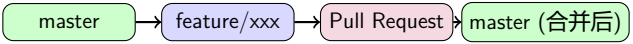
协作要求

每日站会（15 分钟）：各自汇报进度、遇到的问题

GitHub Flow 简化工作流

适合小团队的简化流程：

1. **主分支保护**：master 分支受保护，禁止直接推送
2. **功能分支开发**：从 master 创建 feature 分支
3. **提交 Pull Request**：完成功能后提交 PR
4. **代码审查**：至少 1 人 review 通过
5. **合并到主分支**：CI 通过后合并
6. **删除功能分支**：保持仓库整洁



Git 命令实战

创建功能分支

- `git checkout -b feature/choice-recognizer` - 创建并切换
- `git add .` - 添加修改
- `git commit -m "feat: 实现选择题识别器"` - 提交

推送与同步

- `git push origin feature/choice-recognizer` - 推送到远程
- `git fetch origin` - 拉取远程
- `git rebase origin/develop` - 变基到最新

代码审查流程

代码审查（Code Review）原则

- 所有代码必须经过 review 才能合并
- 审查关注：功能、性能、安全、风格
- 提出问题要具体，给出改进建议
- 保持尊重，对代码不对人
- 及时响应审查意见

审查清单：

检查项	内容
功能性	代码是否实现了需求？边界条件处理了吗？
可读性	命名是否清晰？注释是否充分？
可维护性	是否遵循设计原则？是否高内聚低耦合？
性能	是否有明显性能问题？算法复杂度如何？

审查评论规范

好的评论：

- " 这个变量名建议改为 total_score，更清晰表达含义"
- " 这里建议添加边界检查，防止数组越界"
- " 这个逻辑可以用列表推导式简化，更 Pythonic"

避免的评论：

- " 这写得不对"（太笼统）
- " 重写这段代码"（没有说明原因）
- " 这么简单都写错"（人身攻击）

审查评论标记

- **[suggestion]**：建议性改进
- **[nitpick]**：小的格式问题
- **[required]**：必须修改的问题
- **[blocking]**：阻塞性问题

冲突解决与合并

避免冲突的最佳实践

- 频繁提交，小步迭代
- 开发前先从主分支拉取最新代码
- 不要长时间在分支上开发
- 与团队成员沟通，避免同时修改同一文件
- 使用相同的代码格式化工具

解决冲突步骤：

版本发布管理

语义化版本（Semantic Versioning）：

MAJOR.MINOR.PATCH

- **MAJOR**：不兼容的 API 修改
- **MINOR**：向下兼容的功能新增
- **PATCH**：向下兼容的问题修复

版本示例

- v0.1.0（内测版）
- v0.9.0（公测版）
- v1.0.0（正式版）
- v1.1.0（功能更新）
- v1.1.1（Bug 修复）

标签操作：

版本标签命令

- git tag -a v1.0.0 -m "正式版发布" 创建标签

PEP 8 代码规范详解

命名规范：

类型	规范	示例
模块/包	小写，短名称	`recognition`, `utils`
类	驼峰命名	`ChoiceRecognizer`
函数/方法	小写下划线	`recognize_choice()`
变量	小写下划线	`image_data`
常量	大写下划线	`MAX_IMAGE_SIZE`
私有变量	前导下划线	`_internal_data`
保护成员	双前导下划线	`__init__`

代码布局：

- 缩进：4 个空格（禁用 Tab）
- 行宽：最大 79 字符
- 空行：顶级函数/类间 2 行，方法间 1 行
- 导入：标准库 [illegible]

Python 代码风格检查

使用工具保证代码质量：

black - 代码格式化

- 强制统一代码风格
- 减少代码审查中的格式争论
- 配置简单

安装使用

- `pip install black`
- `black src/`

flake8 - 代码检查

- 检查代码风格
- 发现潜在问题

安装使用

- `pip install flake8`
- `flake8 src/`

类型注解与文档字符串

类型注解示例

- `from typing import List, Optional, Dict`
- `import numpy as np`
- 参数：`image: np.ndarray`
- 返回：`-> Dict[str, str]`

文档字符串规范

- **Args**: 参数说明
- **Returns**: 返回值说明
- **Raises**: 异常说明
- **Example**: 使用示例

接口文档生成

使用工具自动生成文档：

Sphinx：

- Python 标准文档工具
- 支持 reStructuredText
- 生成 HTML/PDF/ePub
- 支持自动生成 API 文档

MkDocs：

- 基于 Markdown
- 配置简单
- 美观的主题
- 适合项目文档

pdoc 文档生成

- `pip install pdoc`
- `pdoc --http : mymodule` - 启动文档服务器
- `pdoc --html mymodule` - 生成 HTML

README 文档规范

项目 README 应包含：

章节	内容
项目名称	清晰的项目标题
项目简介	一句话描述项目目的
功能特性	主要功能列表
技术栈	使用的技术、库、版本
安装部署	环境要求、安装步骤
使用说明	快速上手示例
目录结构	项目文件组织
贡献指南	如何参与开发
许可证	开源协议

任务分配与跟踪

任务分解原则（WBS）：

- 每个任务可独立完成（1-3 天）
- 任务有明确的验收标准
- 任务间依赖关系清晰
- 预留缓冲时间应对风险

任务模板：

字段	示例
任务名称	实现选择题识别器
任务描述	识别答题卡上选择题的填涂答案
验收标准	准确率 >95%，有单元测试
负责人	张三
计划工时	3 天
截止时间	第 10 周周三
优先级	高
依赖任务	图像预处理模块

每日站会与进度同步

每日站会（Daily Stand-up）

- 时间：固定时间，15 分钟以内
- 形式：站立进行，保持高效
- 每人回答三个问题：
  1. 昨天完成了什么？
  2. 今天计划做什么？
  3. 有什么阻碍？

进度同步工具：

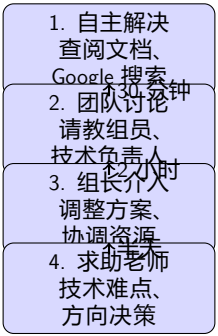
- 看板（Kanban）：直观展示任务状态
- 燃尽图（Burndown）：跟踪进度趋势
- 甘特图（Gantt）：展示任务时间安排

会议记录模板：

- 日期和参会人员
- 各成员进度汇报
- 遇到的问题
- 解决方案

问题沟通与解决

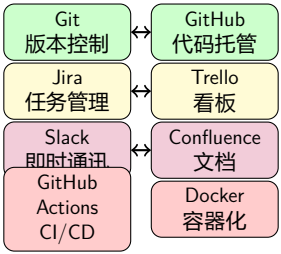
问题升级路径：



沟通原则

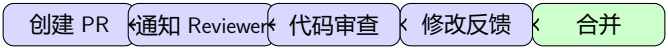
遇到问题不要憋太久，及时求助。但同时也要先自己尝试解决，带着思考去提问。

团队协作工具链


Git
版本控制
GitHub
代码托管
Jira
任务管理
Trello
看板
Slack
即时通讯
Confluence
文档
GitHub
Actions
CI/CD
Docker
容器化

代码评审最佳实践

评审流程：



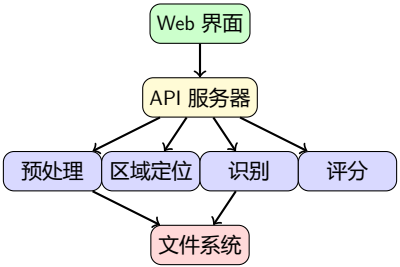
PR 描述模板：

字段	内容
描述	实现选择题识别功能
关联 Issue	#123
改动内容	新增 ChoiceRecognizer 类
测试用例	test_choice_recognition.py

案例 1：小型阅卷系统架构

项目背景：

- 用户：单个班级（约 30 人）
- 试卷量：每周 100-500 份
- 功能：选择题 + 判断题识别
- 团队：1-2 人



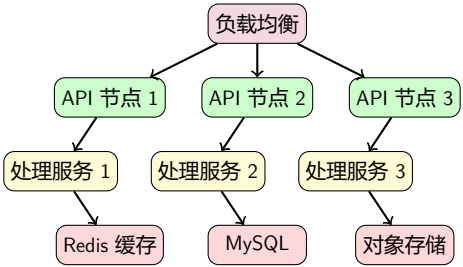
架构特点：

- 单体应用，架构简单

案例 2：中型阅卷系统架构

项目背景：

- 用户：多班级/多学校（1000+ 人）
- 试卷量：每日 10000+ 份
- 功能：全题型支持
- 团队：5-10 人



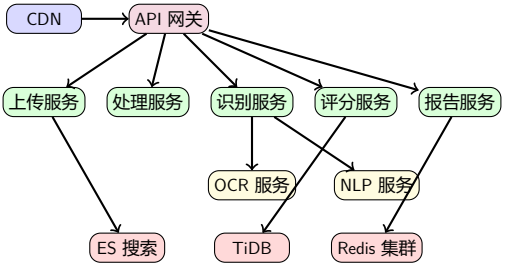
架构特点：

- 微服务架构

案例 3：大型企业级阅卷系统

项目背景：

- 用户：全国范围（百万用户）
- 试卷量：每日百万级
- 功能：全题型 +AI 辅助
- 团队：50+ 人，多团队协作



架构特点：

- 云原生架构

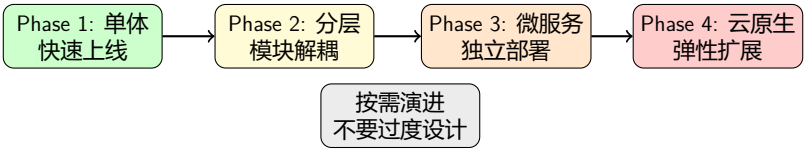
案例对比分析

维度	小型	中型	大型
用户规模	<1000	1000-10 万	10 万 +
试卷量级	10 份/天	1 万份/天	100 万份/天
团队规模	1-2 人	5-10 人	50+ 人
架构风格	单体分层	微服务	云原生
部署方式	单机部署	集群部署	K8s+ 多活
扩展策略	垂直扩展	水平扩展	自动扩缩容
数据存储	SQLite	MySQL	TiDB+ES

选型建议

学生课程项目建议采用：单体分层架构 + 管道过滤器模式足够简单支撑当前需求，同时为未来扩展预留空间。

架构演进路线



演进原则：

- 业务驱动：扩展跟不上业务时再演进
- 渐进式：逐步拆分，而非一次性重写
- 可逆性：保持模块可独立部署能力

微服务架构最佳实践

服务拆分原则：

1. 按业务能力拆分（不是按技术）
2. 每个服务独立数据库
3. 服务间通过 API 通信
4. 避免分布式事务

服务治理：

必须：

- 服务注册发现
- 负载均衡
- 熔断降级
- 链路追踪

建议：

- API 网关
- 配置中心
- 消息队列
- 限流

常见问题

- 服务拆分过细：运维复杂

领域驱动设计（DDD）实践

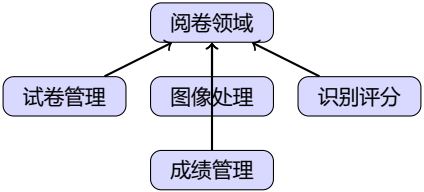
核心概念：

战略设计：

- 限界上下文（Bounded Context）
- 领域（Domain）
- 子域（Subdomain）
- 上下文映射（Context Map）

战术设计：

- 实体（Entity）
- 值对象（Value Object）
- 聚合（Aggregate）
- 领域服务（Domain Service）



DevOps 与 CI/CD 最佳实践

CI/CD 流水线：

GitHub Actions 示例：

YAML 配置示例

```
name: CI
on: [push, pull_request]

jobs:
 build:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3
 - name: Run tests
 run: |
```

测试策略最佳实践

测试金字塔：

测试分层

单元测试（70%）

`test_preprocessor.py, test_recognizer.py, ...`

集成测试（20%）

`test_pipeline_integration.py, test_engine_integration.py`

端到端测试（10%）

`test_full_workflow.py, test_api_endpoints.py`

快速、稳定　　慢、复杂

测试原则：

- 测试覆盖核心业务逻辑
- Mock 外部依赖

Quiz 1：架构设计判断

判断正误：

1. 微服务比单体架构更好，所以应该一开始就采用微服务。
2. 分层架构中，上层可以跳过中间层直接调用下层。
3. SOLID 原则中的" 开闭原则" 指的是软件应该对扩展开放，对修改关闭。
4. 管道-过滤器架构适合处理图像数据流。
5. 识别器注册表可以让我们在不修改引擎代码的情况下添加新的识别算法。

答案

1. 错 2. 错 3. 对 4. 对 5. 对

Quiz 2：设计模式选择

场景匹配：

场景	设计模式
需要在运行时切换不同的识别算法	
需要为一个类动态添加日志功能	
需要确保全局只有一个配置实例	
需要将复杂对象的构建分步骤进行	
需要在识别完成后通知评分模块	

参考答案

Quiz 3：架构问题诊断

案例分析：

问题描述

小明的智能阅卷系统运行一年后，代码变得难以维护。每次添加新题型都需要修改识别引擎的核心代码，而且经常导致原有功能出错。

问题诊断：

1. 识别引擎违反了哪个设计原则？
2. 如何用设计模式解决这个问题？
3. 如何重构代码使其更易扩展？

参考答案

1. 违反了开闭原则（OCP）和单一职责原则（SRP）
2. 应用策略模式，将识别算法抽象为接口
3. 使用工厂模式创建识别器，使用注册表管理识别器

Quiz 4：场景设计

设计练习：

题目

请为以下场景设计解决方案：

1. 需要支持多种预处理算法（高斯滤波、中值滤波、双边滤波），用户可以选择使用哪种算法。
2. 识别系统需要能够处理多种图片格式（PNG、JPG、TIFF）。
3. 处理过程中需要记录每个步骤的详细信息，便于调试。

设计要点：

1. 策略模式 + 工厂模式
2. 适配器模式
3. 观察者模式 + 日志

讨论：架构决策

讨论题目

在实际项目中，架构设计常常面临取舍：

1. 简单 vs 可扩展：对于课程项目，什么程度的架构设计是" 恰到好处"?
2. 规范 vs 速度：严格遵循代码规范是否会拖慢开发进度?
3. 个人 vs 团队：在小团队中是否有必要使用复杂的版本控制流程?

讨论要点：

支持严格规范：

- 长远看节省时间
- 减少沟通成本
- 代码更易维护

适度灵活：

- 根据项目阶段调整
- 平衡成本和收益
- 保持核心原则

最佳实践投票（实时互动）

手机投票说明

问卷星实时投票：请拿出手机，扫描二维码参与投票

- 投票结果实时显示在主屏
- 看看其他小组的选择
- 准备分享你们组的决策理由

投票题目：

1. 你们组打算采用什么分支策略？
   - A. Git Flow（规范）
   - B. GitHub Flow（简洁）
   - C. 主干开发（极致简化）
2. 识别算法通过什么方式扩展？
3. 代码审查的频率？
   - A. 每次 PR 都审查（严格）
   - B. 每日审查（平衡）
   - C. 合并前审查（宽松）

思考题

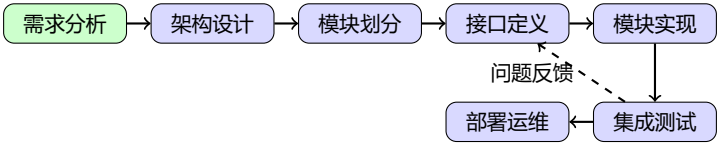
课后思考

1. 在识别引擎中，如果某个识别算法的准确率突然下降，如何快速定位问题？
2. 如果要支持实时识别（边扫描边识别），流水线架构需要如何调整？
3. 在团队协作中，如何平衡代码审查的严格程度和开发效率？

提示

- 考虑添加监控和日志
- 考虑使用流式处理
- 考虑自动化检查工具

系统架构设计完整流程



核心要点：

- 架构设计要面向需求，而非面向技术
- 模块划分遵循高内聚低耦合原则
- 接口设计要稳定，实现可以变化
- 持续重构，保持架构活力

架构风格速查表

架构风格	核心思想	适用场景	实现技术
分层架构	按职责分层	企业应用	Spring, Django
微服务	服务拆分独立部署	大型分布式系统	Docker, K8s
管道-过滤器	数据流处理	图像/数据处理	OpenCV, FFmpeg
事件驱动	异步消息通信	实时系统	Kafka, RabbitMQ

选择原则：

- 小团队、简单业务：单体/分层架构
- 大团队、复杂业务：微服务架构
- 数据处理流水线：管道-过滤器
- 高并发、实时性：事件驱动

设计模式速查表

类型	模式	解决问题	应用场景
创建型	工厂模式	对象创建与使用解耦	根据类型创建不同对象
	单例模式	全局唯一实例	配置管理、数据库连接
	建造者模式	复杂对象构建	多参数对象构造
结构型	适配器模式	接口不兼容转换	集成第三方库
	装饰器模式	动态添加功能	日志、缓存、权限
	代理模式	控制对象访问	延迟加载、远程代理
行为型	策略模式	算法 interchangeable	多种识别算法切换
	观察者模式	状态变化通知	事件监听、消息订阅
	模板方法	算法骨架复用	通用处理流程

SOLID 原则速查

原则	核心	实践要点
SRP	单一职责	一个类只做一件事，修改原因只有一个
OCP	开闭原则	扩展新功能不修改已有代码
LSP	里氏替换	子类可以无缝替换父类
ISP	接口隔离	客户端不依赖不需要的接口
DIP	依赖倒置	依赖抽象接口，而非具体实现

记忆口诀

"职责单一要封闭，替换子类没问题，接口隔离倒依赖"

常见问题与解决方案

问题	解决方案
模块耦合严重，修改一处影响全局	应用 DIP，引入接口抽象；使用依赖注入
代码重复，复制粘贴多	提取公共逻辑，应用 DRY 原则
扩展困难，新增功能需要修改多处	应用 OCP，使用策略/工厂模式
类职责不清，功能臃肿	应用 SRP，拆分小类
团队协作冲突多	规范 Git 工作流，明确代码审查流程
接口频繁变更	接口版本化，向下兼容
性能瓶颈	引入缓存、异步处理、并行计算

作业要求详解

作业 1：架构设计文档（必做）

为你的小组项目设计系统架构：

1. 绘制系统架构图（模块划分、接口关系）
2. 说明选择的架构风格及理由
3. 定义各模块的接口（输入、输出、职责）
4. 说明应用了哪些设计模式

提交格式： Markdown 文档 + 架构图（PNG/PDF）

作业 2：流水线实现（必做）

实现图像预处理流水线：

1. 定义 FilterInterface 接口

作业评分标准

评分项	优秀 (90-100)	良好 (75-89)	及格 (60-74)
架构完整性	设计合理，扩展性强	设计合理	基本可用
代码规范	规范 + 类型注解 + 文档	规范	基本规范
设计模式应用	恰当使用 3+ 种	使用 2 种	使用 1 种
测试覆盖	单元测试覆盖主要逻辑	有测试	无测试
团队协作	Git 提交规范，PR 清晰	提交较规范	提交混乱

提交要求

推荐书籍与论文

架构设计：

- 《软件架构设计：大型网站技术架构与业务架构融合之道》—余洪春
- 《实现领域驱动设计》—Vaughn Vernon
- 《微服务设计》—Sam Newman
- 《软件架构模式》—Mark Richards（免费电子书）

设计模式：

- 《设计模式：可复用面向对象软件的基础》—GoF
- 《Head First 设计模式》—Eric Freeman
- 《Python 设计模式》—Brandon Rhodes

代码规范：

- 《代码整洁之道》—Robert C. Martin
- 《重构：改善既有代码的设计》—Martin Fowler

在线课程与教程

中文资源：

- 极客时间《从 0 开始学架构》—李运华
- 极客时间《设计模式之美》—王争
- B 站《软件架构设计》系列课程
- 慕课网《Python 设计模式》

英文资源：

- Coursera: Software Architecture for the Internet of Things
- Udemy: Software Architecture Technology of Large-Scale Systems
- YouTube: System Design Primer 系列
- Refactoring.Guru: 设计模式教程（有中文）

官方文档：

- PEP 8 – Python 代码风格指南
- Git 官方文档与 Pro Git 电子书

开源项目参考

推荐学习项目：

项目	语言	学习点
OCRmyPDF	Python	图像处理流水线、插件架构
OpenCV-Python	Python/C++	计算机视觉算法实现
FastAPI	Python	现代 Web 框架设计
Celery	Python	分布式任务队列
Pytest	Python	测试框架设计

参与开源：

- 从阅读源码开始，理解项目架构
- 修复 good-first-issue 标签的问题
- 参与代码审查，学习最佳实践

谢谢！

有问题随时交流

下节预告：第 10 周核心开发与调试

目标：完成所有模块开发与集成