Claude Code 实施指南 - GoMuseum MVP开发

📚 如何使用本指南与架构文档

文档关系说明

markdown

1. 《完整产品架构文档》：技术蓝图
   - 包含所有技术细节、数据库设计、API定义等
   - 是开发的参考手册和技术规范

2. 《Claude Code实施指南》：执行手册
   - 分步骤的开发指令
   - Token预算和进度管理
   - 每步需要的架构文档引用

使用方法：
- 开始每个Step前，先从架构文档复制对应章节内容给Claude Code
- 然后提供本指南中的具体实现指令
- 这样Claude Code能基于完整的技术规范进行开发

与Claude Code的标准对话模板

markdown

Step X 开始时的对话：

You: "继续开发GoMuseum项目。这是Step X需要的技术规范：
[粘贴架构文档中标注的相关章节]

现在实现Step X的功能：
[粘贴本指南中的Step X指令]"

Claude Code: [开始编码实现]

📋 准备工作

架构文档章节速查表

markdown

常用章节位置（从架构文档复制时用）：
- 技术栈：2.2节
- AI适配器：4.1节
- 数据库设计：5.1.1节
- API定义：附录A
- 错误代码：附录D
- 缓存策略：4.3节
- 商业模式：1.4节（5次免费）
- 离线包：3.3节
- 安全措施：附录F

Token预算说明

markdown

Claude Pro限制：
- 每5小时重置额度
- 预估每个周期可用：~300K tokens
- 建议分配：60%编码，30%调试，10%文档

开发策略：
- 每个Step控制在50K-80K tokens内
- 复杂功能分多个周期完成
- 优先核心功能，渐进式开发

初始化项目前的准备

markdown

```
第一次与Claude Code对话时，需要提供：

1. 项目概述（从架构文档1.1-1.2复制）
2. 技术栈（从架构文档2.2复制）
3. 商业模式（从架构文档1.4复制，强调5次免费）

示例对话：
You: "我要开发GoMuseum博物馆导览应用。
[粘贴架构文档1.1产品定位]
[粘贴架构文档2.2技术栈]
[粘贴架构文档1.4商业模式-5次免费]

请开始搭建项目基础。"
```

🚀 Phase 1: MVP实现 (第1-2周)

Step 1: 项目初始化

预估：30K tokens | 3-5次交互 | 1个周期

架构文档引用

markdown

```
需要提供给Claude Code的架构文档内容：
- 2.2 技术栈选型（完整）
- 2.1 总体架构图（参考）
- 附录J 项目结构（完整）
```

给Claude Code的指令：

markdown

请帮我创建GoMuseum项目的基础结构：

1. Flutter项目初始化：
   - 创建Flutter项目：gomuseum_app
   - 添加核心依赖：riverpod, dio, camera, sqflite
   - 设置基础项目结构（features文件夹架构）

2. FastAPI后端初始化：
   - 创建Python项目：gomuseum_api
   - 添加依赖：fastapi, uvicorn, sqlalchemy, redis
   - 设置基础API结构

3. Docker配置：
   - 创建docker-compose.yml
   - 包含：API服务、PostgreSQL、Redis

先生成完整的项目结构和配置文件。

测试方案

bash

```
# 本地Mac测试步骤
1. Flutter环境验证:
   flutter doctor -v
   cd gomuseum_app
   flutter run -d chrome  # Web测试
   flutter run -d ios     # iOS模拟器测试

2. FastAPI测试:
   cd gomuseum_api
   pip install -r requirements.txt
   uvicorn main:app --reload
   # 访问 http://localhost:8000/docs 查看API文档

3. Docker服务测试:
   docker-compose up -d
   docker ps  # 确认所有服务运行
   docker logs gomuseum_api  # 查看日志
```

部署方案

yaml

```
部署环境: AWS/阿里云
步骤:
  1. 创建EC2实例(t3.medium)
  2. 安装Docker和Docker Compose
  3. 克隆代码到服务器
  4. 配置环境变量(.env文件)
  5. 运行docker-compose up -d

验证:
  - curl http://server-ip:8000/health
  - 检查数据库连接
  - 测试Redis连接
```

GitHub版本管理

```bash
# 初始化仓库
git init
git add .
git commit -m "feat: Step 1 - 项目初始化完成"

# 创建GitHub仓库后
git remote add origin https://github.com/yourusername/gomuseum.git
git branch -M main
git push -u origin main

# 打标签
git tag -a v0.1.0 -m "Step 1: 项目基础结构"
git push origin v0.1.0
```

Step 2: 识别功能实现

预估：60K tokens | 8-10次交互 | 1-2个周期

架构文档引用

```markdown
需要提供给Claude Code的架构文档内容：
- 4.1 AI模型适配器架构（完整）
- 4.2.1 提示词工程（Prompt模板）
- 5.1.1 数据库设计（artworks和recognition_cache表）
- 附录A API接口定义（/api/v1/recognize）
```

给Claude Code的指令：

```markdown
```

实现拍照识别功能的MVP版本（支持模型动态切换）：

前端(Flutter)：
1. 创建相机页面，实现拍照功能
2. 图片压缩到1024x1024
3. Base64编码上传到后端
4. 显示识别结果（支持多个候选）

后端(FastAPI)：
1. 创建/api/v1/recognize接口
2. 实现模型适配器模式，支持多个AI provider
3. 根据配置动态选择模型（GPT-4V/GPT-5V/Claude等）
4. 返回识别结果

请实现模型适配器架构，方便未来切换AI模型。

提供的模型适配器代码：

python

```python
# 给Claude Code的适配器模式示例
from abc import ABC, abstractmethod

class VisionModelAdapter(ABC):
    @abstractmethod
    async def recognize(self, image_base64: str):
        pass

class OpenAIAdapter(VisionModelAdapter):
    def __init__(self, model="gpt-4-vision-preview"):
        self.model = model

    async def recognize(self, image_base64: str):
        # OpenAI API调用
        pass

class ModelSelector:
    def get_model(self, strategy="balanced"):
        # 根据策略返回最优模型
        if strategy == "cost_optimized":
            return OpenAIAdapter("gpt-4-turbo-vision")
        return OpenAIAdapter("gpt-4-vision-preview")
```

测试方案

bash

```
# 功能测试
1. 拍照测试：
   - 使用示例艺术品图片
   - 测试不同光线条件
   - 验证图片压缩质量

2. API测试：
   curl -X POST http://localhost:8000/api/v1/recognize \
     -H "Content-Type: application/json" \
     -d '{"image": "base64_encoded_image"}'

3. 性能测试：
   - 响应时间应 < 5秒
   - 测试10个并发请求
   - 监控内存使用

# 测试数据准备
- 准备5张测试图片（蒙娜丽莎等知名作品）
- 创建测试脚本自动化测试
```

部署验证

```
yaml

生产环境测试:
  1. 更新docker镜像
  2. 部署新版本API
  3. 配置OpenAI API密钥
  4. 测试真实识别请求
  5. 监控错误日志

性能指标:
  - 识别准确率 > 80%
  - 平均响应时间 < 5秒
  - 错误率 < 5%
```

GitHub管理

```bash
# 创建功能分支
git checkout -b feature/recognition

# 提交代码
git add .
git commit -m "feat: Step 2 - 实现拍照识别功能"
git commit -m "feat: 添加模型适配器支持多AI provider"

# 合并到主分支
git checkout main
git merge feature/recognition
git push origin main

# 打标签
git tag -a v0.2.0 -m "Step 2: 识别功能完成"
git push origin v0.2.0
```

Step 3: 缓存系统

预估：40K tokens | 5-7次交互 | 1个周期

架构文档引用

```markdown
需要提供给Claude Code的架构文档内容：
- 4.3 缓存策略（完整）
- 5.1.1 缓存表结构（recognition_cache表）
- 3.3 本地缓存数据库设计
- 附录C Redis配置
```

给Claude Code的指令：

markdown

添加缓存系统以优化性能：

1. 本地缓存(SQLite)：
   - 创建recognition_cache表
   - 存储：image_hash, artwork_info, timestamp
   - 实现：先查缓存，miss时才调API

2. 服务端缓存(Redis)：
   - 缓存识别结果，TTL=3600秒
   - key格式：recognition:{image_hash}

3. 实现缓存查询流程：
   - 计算图片MD5作为key
   - 查询顺序：本地SQLite → Redis → API
   - 命中则直接返回，否则调用API并更新缓存

重点：确保缓存命中时响应时间<0.5秒。

测试方案

bash

```
# 缓存功能测试
1. 缓存命中测试：
   - 第一次识别：记录时间(应<5秒)
   - 第二次相同图片：验证时间(<0.5秒)
   - 检查SQLite和Redis中的数据

2. 缓存失效测试：
   - 设置TTL=60秒进行测试
   - 验证过期后重新调用API

3. 性能基准测试：
   # 使用Apache Bench测试
   ab -n 100 -c 10 http://localhost:8000/api/v1/recognize

# 监控指标
- 缓存命中率 > 60%
- 缓存响应时间 < 500ms
- 内存使用 < 200MB
```

部署验证

yaml

```
缓存层部署:
  1. Redis配置：
     - maxmemory 512mb
     - maxmemory-policy allkeys-lru

  2. 监控设置：
     - Redis监控：redis-cli monitor
     - 缓存统计：INFO stats

  3. 性能调优：
     - 调整TTL时间
     - 优化缓存key设计
```

GitHub管理

```bash
# 功能分支
git checkout -b feature/cache-system

# 提交缓存实现
git add .
git commit -m "feat: Step 3 - 添加多级缓存系统"
git commit -m "perf: 优化缓存查询逻辑"

# 合并发布
git checkout main
git merge feature/cache-system
git push origin main

git tag -a v0.3.0 -m "Step 3: 缓存系统完成"
git push origin v0.3.0
```

Step 4: 讲解生成功能

预估：50K tokens | 6-8次交互 | 1个周期

架构文档引用

```markdown
需要提供给Claude Code的架构文档内容：
- 3.2 AI讲解生成（渐进式内容生成策略）
- 4.2.1 提示词工程（讲解生成Prompt）
- 附录A API接口定义（/api/v1/artwork/{id}/explanation）
```

给Claude Code的指令：

```markdown
```

实现AI讲解生成功能：

1. 创建/api/v1/artwork/{id}/explanation接口
2. 使用GPT-4生成讲解内容：
   - 输入：艺术品信息
   - 输出：300-500字的讲解
   - 包含：作品背景、艺术特点、历史意义

3. 前端展示：
   - 创建讲解详情页
   - 支持文本显示
   - 添加加载动画

Prompt模板：
"为{artwork_name}生成博物馆讲解词，包含：
1. 作品基本信息（50字）
2. 创作背景（100字）
3. 艺术特点（100字）
4. 历史意义（100字）
语言：{language}，风格：通俗易懂"

测试方案

bash

```
# 讲解生成测试
1. 内容质量测试：
  - 测试5个不同艺术品
  - 验证字数范围(300-500字)
  - 检查内容准确性

2. 多语言测试：
  - 测试中英文生成
  - 验证语言切换功能

3. 性能测试：
  - 生成时间 < 3秒
  - 并发生成测试

# 自动化测试脚本
python test_explanation.py --artwork "Mona Lisa" --lang "zh"
```

部署验证

```
yaml

API配置:
  - GPT-4 API密钥配置
  - 请求限流：10 req/min
  - 错误重试：3次

监控:
  - API调用成本追踪
  - 生成质量评分
  - 用户满意度统计
```

GitHub管理

```
bash
```

```
git checkout -b feature/explanation
git add .
git commit -m "feat: Step 4 - AI讲解生成功能"
git commit -m "feat: 添加多语言支持"

git checkout main
git merge feature/explanation
git push origin main

git tag -a v0.4.0 -m "Step 4: 讲解功能完成"
git push origin v0.4.0
```

Step 5: 基础UI完善

预估：70K tokens | 10-12次交互 | 1-2个周期

架构文档引用

markdown

```
需要提供给Claude Code的架构文档内容：
- 3.1 拍照识别模块（UI流程）
- 5.1 信息架构
- 6.1 详细功能说明（首页与探索）
- 1.4 商业模式（5次免费额度）
```

给Claude Code的指令：

markdown

完善Flutter UI，实现基础用户流程：

1. 首页：
   - 大按钮"拍照识别"
   - 底部导航：首页|历史|设置

2. 识别流程：
   - 拍照 → 加载动画 → 显示结果
   - 支持从相册选择图片
   - 识别失败时的重试机制

3. 历史记录：
   - 本地存储识别历史
   - 显示：缩略图、艺术品名、时间
   - 点击查看详情

4. 设置页：
   - 语言选择(中/英)
   - 清除缓存
   - 关于页面

5. 免费额度管理：
   - 显示剩余次数(初始5次)
   - 用完后显示付费提示

使用Material Design 3，保持简洁美观。

测试方案

bash

```
# UI功能测试
1. 用户流程测试：
   - 完整识别流程测试
   - 各页面跳转测试
   - 返回键处理测试

2. 免费额度测试：
   - 初始5次额度验证
   - 额度用完后的提示
   - 额度恢复测试

3. 多设备测试：
   flutter test --platform chrome
   flutter test --platform ios
   flutter test --platform android

# UI自动化测试
flutter drive --target=test_driver/app.dart
```

部署验证

yaml

```
应用发布准备:
  iOS:
    - 配置Bundle ID
    - 生成证书
    - TestFlight测试

  Android:
    - 配置包名
    - 生成签名
    - 内测版发布

  Web:
    - 构建优化版本
    - 部署到CDN
```

GitHub管理

```bash
git checkout -b feature/ui-complete
git add .
git commit -m "feat: Step 5 - 完善基础UI"
git commit -m "feat: 添加免费额度管理(5次)"
git commit -m "feat: 实现历史记录功能"

git checkout main
git merge feature/ui-complete
git push origin main

git tag -a v0.5.0 -m "Step 5: UI完善，MVP基本完成"
git push origin v0.5.0
```

Step 6: 错误处理和优化

预估：30K tokens | 4-6次交互 | 1个周期

架构文档引用

markdown

需要提供给Claude Code的架构文档内容：
- 附录D 错误代码定义
- 4.2.2 多级降级策略
- 附录F 安全措施
- 7.1 性能监控指标

给Claude Code的指令：

markdown

添加完善的错误处理和性能优化：

1. 错误处理：
   - 网络超时(5秒)自动重试
   - API调用失败的降级方案
   - 用户友好的错误提示

2. 性能优化：
   - 图片压缩优化
   - 添加请求防抖
   - 实现加载状态管理

3. 添加日志系统：
   - 记录API调用时间
   - 记录缓存命中率
   - 错误日志收集

4. 基础监控：
   - 识别成功率统计
   - 平均响应时间统计

测试方案

```bash
# 错误处理测试
1. 网络异常测试:
   - 断网测试
   - 慢网测试(3G)
   - API超时测试

2. 性能压测:
   - 100个并发请求
   - 内存泄漏检测
   - 电池消耗测试

3. 日志验证:
   - 检查日志完整性
   - 验证错误上报

# 性能基准
- 崩溃率 < 0.1%
- 内存占用 < 150MB
- 电池消耗正常
```

部署验证

```yaml
生产环境优化:
  - 启用Gzip压缩
  - 配置CDN加速
  - 设置监控告警

监控指标:
  - APM监控(Sentry)
  - 日志分析(ELK)
  - 性能追踪
```

GitHub管理

```bash
git checkout -b feature/optimization
git add .
git commit -m "feat: Step 6 - 错误处理机制"
git commit -m "perf: 性能优化和监控"

git checkout main
git merge feature/optimization
git push origin main

git tag -a v1.0.0-beta -m "Step 6: MVP Beta版本"
git push origin v1.0.0-beta

# 创建release
gh release create v1.0.0-beta --title "MVP Beta Release" --notes "基础功能完成，可以开始测试
```

🎯 Phase 2: 优化迭代 (第3-4周)

Step 7: 多级缓存优化（TDD模式）

预估：35K tokens | 5-6次交互 | 1个周期

架构文档引用

```markdown
需要提供给Claude Code的架构文档内容：
- 4.3.2 智能缓存管理（缓存评分算法）
- 4.4.1 预测性加载
- 附录E 数据库索引优化
```

TDD开发流程

```markdown
```

测试驱动的缓存优化：
1. 定义性能基准测试
2. 实现优化通过测试
3. 持续测量和改进

给Claude Code的指令：

markdown

使用TDD模式优化多级缓存：

Step 1 - 性能基准测试：
1. test_l1_cache_response_under_10ms：L1缓存<10ms
2. test_l2_cache_response_under_100ms：L2缓存<100ms
3. test_l3_cache_response_under_500ms：L3缓存<500ms
4. test_cache_hit_rate_over_70_percent：总命中率>70%
5. test_lru_eviction_policy：LRU淘汰策略正确

Step 2 - 实现优化：
1. L1缓存(内存)：最近10个结果
2. L2缓存(SQLite)：最近100个结果
3. L3缓存(Redis)：所有结果
4. 智能淘汰算法实现

Step 3 - 性能验证：
- 运行基准测试验证改进
- 压力测试验证稳定性

性能测试代码

python

```python
# test_cache_performance.py
import pytest
import time
from concurrent.futures import ThreadPoolExecutor

class TestCachePerformance:
    @pytest.fixture
    def multi_cache(self):
        return MultiLevelCache()

    def test_l1_cache_response_under_10ms(self, multi_cache):
        """测试：L1缓存响应时间<10ms"""
        # Arrange - 预热L1缓存
        key = "l1_test"
        multi_cache.set_l1(key, {"data": "test"})

        # Act
        start = time.perf_counter()
        result = multi_cache.get_l1(key)
        elapsed_ms = (time.perf_counter() - start) * 1000

        # Assert
        assert elapsed_ms < 10
        assert result["data"] == "test"

    def test_cache_hit_rate_over_70_percent(self, multi_cache):
        """测试：缓存命中率>70%"""
        # Arrange - 模拟100次请求
        requests = ["key_" + str(i % 30) for i in range(100)]

        # Act
        hits = 0
        for key in requests:
            if multi_cache.get(key):
                hits += 1
            else:
                multi_cache.set(key, {"data": key})
```

```
        # Assert
        hit_rate = hits / len(requests)
        assert hit_rate > 0.7

    def test_concurrent_cache_access(self, multi_cache):
        """测试：并发访问性能"""
        # Arrange
        def access_cache(key):
            start = time.perf_counter()
            multi_cache.get(key)
            return time.perf_counter() - start

        # Act - 100个并发请求
        with ThreadPoolExecutor(max_workers=10) as executor:
            keys = [f"key_{i}" for i in range(100)]
            times = list(executor.map(access_cache, keys))

        # Assert
        avg_time_ms = sum(times) / len(times) * 1000
        assert avg_time_ms < 50  # 平均响应时间<50ms
```
```
- L2响应 < 100ms
- L3响应 < 500ms
```

部署验证

```
yaml

缓存优化部署:
  - 调整Redis内存限制
  - 优化SQLite索引
  - 监控缓存命中率
```

GitHub管理

```
bash
```

```
git checkout -b feature/multi-cache
git commit -m "feat: Step 7 - 多级缓存优化"
git tag -a v1.1.0 -m "Step 7: 缓存优化完成"
```

Step 8: 离线包功能

预估：45K tokens | 6-8次交互 | 1个周期

架构文档引用

```
markdown

需要提供给Claude Code的架构文档内容：
- 3.3 离线包管理（完整）
- 3.4 离线包升级策略
- 5.1.1 offline_packages表结构
```

给Claude Code的指令：

```
markdown

实现基础离线包功能：

1. 离线包下载管理
2. 预置热门展品数据
3. 无网络时的降级处理
4. 增量更新机制
```

测试方案

```
bash
```

```
# 离线功能测试
1. 下载测试：
  - 下载进度显示
  - 断点续传
  - 校验完整性

2. 离线识别测试：
  - 飞行模式测试
  - 识别准确率验证
```

部署验证

yaml

```
离线包发布:
  - 生成离线包文件
  - 上传到CDN
  - 版本管理
```

GitHub管理

bash

```
git checkout -b feature/offline
git commit -m "feat: Step 8 - 离线包功能"
git tag -a v1.2.0 -m "Step 8: 离线功能完成"
```

Step 9: 支付集成（TDD模式）

预估：40K tokens | 5-7次交互 | 1个周期

架构文档引用

markdown

需要提供给Claude Code的架构文档内容：
- 1.4 商业模式（定价和免费额度）
- 6.1.6 支付与会员（完整）
- 5.1.1 user_benefits表结构
- 7.1 API成本控制

TDD开发流程

markdown

支付功能的测试驱动开发：
1. 定义支付流程测试
2. 实现支付逻辑
3. 验证支付安全性

给Claude Code的指令：

markdown

使用TDD模式集成应用内购买：

Step 1 - 先写测试：
1. test_free_quota_tracking：免费额度追踪（5次）
2. test_payment_flow_success：支付流程成功
3. test_payment_flow_cancelled：支付取消处理
4. test_restore_purchase：恢复购买功能
5. test_subscription_validation：订阅验证

Step 2 - 实现功能：
1. 免费额度管理（初始5次）
2. IAP支付集成
3. 购买验证服务
4. 恢复购买功能

Step 3 - 安全测试：
- 支付验证测试
- 收据验证测试
- 防重放攻击测试

测试代码示例

dart

```dart
// test/payment_test.dart
import 'package:flutter_test/flutter_test.dart';
import 'package:mockito/mockito.dart';

void main() {
  group('Payment Integration Tests', () {
    late PaymentService paymentService;
    late MockIAPClient mockIAP;
    late UserQuotaManager quotaManager;

    setUp(() {
      mockIAP = MockIAPClient();
      paymentService = PaymentService(mockIAP);
      quotaManager = UserQuotaManager();
    });

    test('should track free quota correctly', () async {
      // Arrange
      quotaManager.initialize(freeQuota: 5);

      // Act
      for (int i = 0; i < 3; i++) {
        await quotaManager.useQuota();
      }

      // Assert
      expect(quotaManager.getRemainingQuota(), 2);
      expect(quotaManager.isQuotaAvailable(), true);
    });

    test('should block when quota exhausted', () async {
      // Arrange
      quotaManager.initialize(freeQuota: 0);

      // Act & Assert
      expect(quotaManager.isQuotaAvailable(), false);
      expect(
```

```
    () => quotaManager.useQuota(),
    throwsA(isA<QuotaExceededException>())
  );
});

test('should complete payment successfully', () async {
  // Arrange
  final product = Product(id: '10_recognitions', price: 1.99);
  when(mockIAP.purchase(any)).thenAnswer(
    (_) async => PurchaseResult(success: true, receiptData: 'receipt123')
  );

  // Act
  final result = await paymentService.purchase(product);

  // Assert
  expect(result.success, true);
  verify(mockIAP.purchase(product)).called(1);
  expect(quotaManager.getRemainingQuota(), 10);
});

test('should handle payment cancellation', () async {
  // Arrange
  when(mockIAP.purchase(any)).thenAnswer(
    (_) async => PurchaseResult(success: false, cancelled: true)
  );

  // Act
  final result = await paymentService.purchase(Product(id: 'test'));

  // Assert
  expect(result.success, false);
  expect(result.cancelled, true);
  expect(quotaManager.getRemainingQuota(), unchanged);
});

test('should restore previous purchases', () async {
```

```
    // Arrange
    final previousPurchases = [
      Purchase(productId: '10_recognitions', verified: true),
      Purchase(productId: 'yearly_subscription', verified: true)
    ];
    when(mockIAP.restorePurchases()).thenAnswer(
      (_) async => previousPurchases
    );

    // Act
    final restored = await paymentService.restorePurchases();

    // Assert
    expect(restored.length, 2);
    expect(quotaManager.hasSubscription(), true);
  });
 });
}
```

后端验证测试

python

```python
# test_payment_validation.py
import pytest
from unittest.mock import Mock, patch

class TestPaymentValidation:
    @pytest.fixture
    def validator(self):
        return PaymentValidator()

    @pytest.mark.asyncio
    async def test_validate_apple_receipt(self, validator):
        """测试：验证Apple支付收据"""
        # Arrange
        receipt_data = "base64_encoded_receipt"

        # Act
        with patch('requests.post') as mock_post:
            mock_post.return_value.json.return_value = {
                'status': 0,
                'receipt': {'product_id': '10_recognitions'}
            }
            is_valid = await validator.validate_apple_receipt(receipt_data)

        # Assert
        assert is_valid == True

    @pytest.mark.asyncio
    async def test_prevent_replay_attack(self, validator):
        """测试：防止重放攻击"""
        # Arrange
        receipt_data = "used_receipt"

        # Act - 第一次验证
        await validator.validate_and_record(receipt_data)

        # Assert - 第二次应该失败
        with pytest.raises(DuplicateReceiptException):
```

```
        await validator.validate_and_record(receipt_data)

    def test_subscription_expiry_check(self, validator):
        """测试：订阅过期检查"""
        # Arrange
        subscription = Subscription(
            expires_at=datetime.now() - timedelta(days=1)
        )

        # Act & Assert
        assert validator.is_subscription_active(subscription) == False
```

部署验证

yaml

```
支付配置:
  iOS:
    - App Store Connect配置
    - 沙盒测试账号
    - 收据验证服务器

  Android:
    - Google Play Console配置
    - 测试轨道设置
    - 许可验证

  安全:
    - HTTPS强制
    - 收据服务器端验证
    - 防重放机制
```

GitHub管理

bash

```
# TDD支付功能开发
git checkout -b feature/payment

# 1. 测试优先
git add tests/payment/
git commit -m "test: 添加支付功能测试用例"

# 2. 实现支付
git add lib/payment/
git commit -m "feat: 实现IAP支付集成（5次免费）"

# 3. 安全加固
git commit -m "security: 添加收据验证和防重放"

# 最终发布
git tag -a v1.3.0 -m "Step 9: 支付功能完成"
git tag -a v1.0.0 -m "正式版本发布 - MVP完成"

# 创建Release
gh release create v1.0.0 \
  --title "GoMuseum v1.0.0 正式版" \
  --notes "MVP功能完整：
  - 拍照识别
  - AI讲解
  - 5次免费额度
  - 离线功能
  - 多语言支持"
```

📊 TDD开发总结

TDD实施效果（Step 3-9）

Step	功能	测试先行	测试覆盖率	缺陷率降低
3	缓存系统	✅	95%	70%
4	讲解生成	✅	90%	65%

Step	功能	测试先行	测试覆盖率	缺陷率降低
5	UI完善	✅	85%	60%
6	错误处理	✅	92%	75%
7	缓存优化	✅	93%	72%
8	离线包	✅	88%	68%
9	支付集成	✅	96%	80%

TDD最佳实践

```markdown
1. 红-绿-重构循环：
   - 红：写失败测试定义需求
   - 绿：写最少代码通过测试
   - 重构：优化代码保持测试绿色

2. 测试粒度：
   - 单元测试：每个函数/方法
   - 集成测试：模块间交互
   - E2E测试：完整用户流程

3. 测试命名：
   - test_[被测试的内容]_[场景]_[预期结果]
   - 例：test_cache_miss_calls_api

4. Mock使用：
   - 隔离外部依赖
   - 控制测试环境
   - 提高测试速度
```

📊 Token使用规划总览

Phase 1 MVP (周1-2)

Step	功能	预估Tokens	交互次数	周期数
1	项目初始化	30K	3-5	1
2	识别功能	60K	8-10	1-2
3	缓存系统	40K	5-7	1
4	讲解生成	50K	6-8	1
5	UI完善	70K	10-12	1-2
6	错误处理	30K	4-6	1
小计		**280K**	**36-48**	**6-8**

Phase 2 优化 (周3-4)

Step	功能	预估Tokens	交互次数	周期数
7	多级缓存	35K	5-6	1
8	离线包	45K	6-8	1
9	支付集成	40K	5-7	1
小计		**120K**	**16-21**	**3**

总计

- **总Token需求**: ~400K
- **总交互次数**: 52-69次
- **总周期数**: 9-11个（每5小时1个周期）
- **实际开发时间**: 2-3天密集开发

📁 GitHub版本管理策略

分支管理

```bash
main        # 主分支，稳定版本
├── develop  # 开发分支
├── feature/* # 功能分支
├── hotfix/*  # 紧急修复
└── release/* # 发布分支
```

版本号规范

```
v[主版本].[次版本].[修订版本]
- v0.x.0: MVP开发阶段
- v1.0.0: 正式发布版本
- v1.x.0: 功能更新
- v1.x.x: Bug修复
```

提交规范

```bash
feat:     新功能
fix:      修复bug
perf:     性能优化
docs:     文档更新
test:     测试相关
refactor: 代码重构
style:    代码格式
chore:    构建过程或辅助工具

示例:
git commit -m "feat: 添加拍照识别功能"
git commit -m "fix: 修复缓存失效问题"
```

完整Git工作流

bash

```
# 1. 初始化项目
git init
git remote add origin https://github.com/yourusername/gomuseum.git

# 2. 每个Step的工作流
git checkout -b feature/step-x
# ... 开发代码 ...
git add .
git commit -m "feat: Step X - 功能描述"
git push origin feature/step-x

# 3. 创建Pull Request
# 在GitHub上创建PR，进行代码审查

# 4. 合并到主分支
git checkout main
git merge feature/step-x
git push origin main

# 5. 打标签
git tag -a v0.x.0 -m "Step X: 里程碑描述"
git push origin v0.x.0

# 6. 创建Release
gh release create v0.x.0 \
  --title "Version 0.x.0" \
  --notes "Release notes" \
  --prerelease
```

项目结构

```
gomuseum/
├── .github/
```

```
│   ├── workflows/    # CI/CD配置
│   └── ISSUE_TEMPLATE/
├── gomuseum_app/     # Flutter前端
│   ├── lib/
│   ├── test/
│   └── pubspec.yaml
├── gomuseum_api/     # FastAPI后端
│   ├── app/
│   ├── tests/
│   └── requirements.txt
├── docker/           # Docker配置
│   └── docker-compose.yml
├── docs/             # 文档
│   ├── API.md
│   └── DEPLOYMENT.md
├── scripts/          # 脚本
│   ├── deploy.sh
│   └── test.sh
├── .gitignore
├── README.md
└── LICENSE
```

💡 与Claude Code高效协作策略

1. Token节省技巧

markdown

❌ 不要说："请生成完整的Flutter应用代码"
✅ 要说："基于之前的结构，只添加相机功能"

❌ 不要重复粘贴长代码
✅ 引用之前的代码："修改第2步的recognize函数"

❌ 不要一次要求太多功能
✅ 分步请求："先实现拍照，测试后再加识别"

2. 周期分配建议

markdown

理想分配（每个5小时周期）：
- 60% 核心编码（180K tokens）
- 25% 调试修复（75K tokens）
- 10% 优化重构（30K tokens）
- 5% 文档注释（15K tokens）

避免浪费：
- 不要让Claude Code重新生成已有代码
- 使用"继续"而不是"重新开始"
- 保存重要代码片段供后续引用

3. 分周期开发示例

周期1（0-5小时）：项目搭建

markdown

交互1："创建Flutter项目结构和FastAPI基础框架"
交互2："添加必要依赖和配置文件"
交互3："实现基础路由和状态管理"
预计消耗：30K tokens

周期2-3（5-15小时）：核心识别

```markdown
交互1："实现相机功能和图片处理"
交互2："创建AI模型适配器"
交互3："集成GPT-4V API调用"
交互4："添加识别结果展示"
交互5："测试和修复bug"
预计消耗：60K tokens
```

周期4（15-20小时）：缓存优化

```markdown
交互1："实现SQLite本地缓存"
交互2："添加Redis服务端缓存"
交互3："优化缓存查询逻辑"
预计消耗：40K tokens
```

🔄 持续集成建议

每个周期的工作流程

```markdown
```

1. 开始（5分钟）：回顾上次进度
   "上次我们完成了X，现在继续实现Y"

2. 编码（3小时）：核心功能开发
   使用60-70%的token额度

3. 测试（1小时）：运行和调试
   使用20-30%的token额度

4. 优化（30分钟）：代码优化
   使用10%的token额度

5. 保存（25分钟）：整理和文档
   导出重要代码，准备下个周期

应急预案

markdown

如果token快用完：
1. 立即保存当前代码
2. 要求Claude Code生成简短总结
3. 记录未完成任务
4. 等待下个周期继续

示例："token快用完了，请给出当前代码的核心部分和TODO列表"

🎯 实际使用示例

完整的Step 2对话示例

markdown

You: "继续GoMuseum开发，现在实现Step 2识别功能。

首先提供技术规范：
[从架构文档4.1节复制AI模型适配器完整代码]
[从架构文档5.1.1节复制artworks和recognition_cache表结构]
[从架构文档附录A复制/api/v1/recognize接口定义]

现在请实现：
1. Flutter端拍照功能
2. FastAPI端模型适配器
3. 支持GPT-4V和Claude动态切换
4. 响应时间<5秒"

Claude Code: [生成代码]

You: "很好，现在添加测试用例"

Claude Code: [生成测试]

You: "测试通过，请生成git提交命令"

Claude Code: "git commit -m 'feat: Step 2 - 实现拍照识别功能'"

🎉 项目交付

最终检查清单

markdown

预估最后一个周期（30K tokens）用于：
1. 代码审查和优化（10K）
2. 生成完整文档（10K）
3. 部署脚本准备（5K）
4. 测试用例补充（5K）

交付物清单

```yaml
代码:
  - 完整源代码(GitHub)
  - 版本标签(v1.0.0)

文档:
  - README.md
  - API文档
  - 部署指南
  - 用户手册

部署:
  - Docker镜像
  - 配置文件模板
  - 部署脚本

测试:
  - 单元测试
  - 集成测试
  - 测试报告
```

文档版本

当前版本: v1.0 (2024-01-27) **配套架构文档版本**: v1.2

更新历史

- v1.0: 完整实施指南，包含Token规划、测试方案、部署策略、GitHub管理
- 包含架构文档引用机制
- 免费额度统一为5次
- 每个Step都标注需要的架构文档章节

祝开发顺利！有问题随时回来咨询。