AI驱动的代码扫描自动化工作流程

🎯 流程概述

本流程通过MCP协议和AI智能分析，实现OneDrive共享文档与GitHub代码扫描结果的双向同步，自动发现代码库变化并维护最新的组件清单，最终部署到远程服务器。

📁 文件路径定义

```
# 本地GitHub仓库
LOCAL_REPO="/Users/jiedong/scan_tool"

# 输入文件（OneDrive同步后的本地文件）
INPUT_FILE="/Users/jiedong/scan_tool/uitoolkit_scan_tool_resourse/File_DLT
.xlsx"

# OneDrive源文件
ONEDRIVE_FILE="/Users/jiedong/Library/CloudStorage/OneDrive-
Cisco/Mac_Component_Report.xlsx"

# Python处理脚本
MAIN_SCRIPT="/Users/jiedong/scan_tool/main.py"

# 输出文件（动态日期）
OUTPUT_FILE="/Users/jiedong/scan_tool/uitoolkit_scan_tool_resourse/DLT_sta
ts_macOS_All_{YYYYMMDD}.xlsx"

# Git远程仓库
REMOTE_REPO="git@sqbu-github.cisco.com:jiedong/scan_tool.git"

# 远程服务器配置
REMOTE_SERVER="production-server.company.com"
REMOTE_REPO_PATH="/opt/scan_tool"
```

🤖 MCP服务架构

1. IntelligentExcelProcessor MCP服务

```
methods:
  - analyze_table_structure(file_path)
  - smart_column_mapping(source_file, target_file)
  - copy_data_by_mapping(source, target, mapping)
  - clear_content_keep_headers(file_path)
  - compare_excel_files(file1, file2)
```

2. SmartGitOperations MCP服务

```
methods:
  - analyze_file_changes(before_file, after_file)
  - generate_smart_commit_message(analysis_result)
  - commit_and_push(message)
  - create_pull_request()
  - monitor_pr_status()
  - get_changed_files_list()
```

3. IntelligentFileManager MCP服务

```
methods:
  - find_latest_dlt_file(directory, pattern)
  - backup_file(file_path)
  - verify_file_existence(file_path)
```

4. RemoteServerManager MCP服务

```
methods:
  - connect_to_server(host, username, key_path)
  - execute_remote_commands(commands_list)
  - sync_repository(repo_path)
  - deploy_files(file_mapping)
  - verify_deployment(expected_files)
  - check_file_permissions(file_paths)
```

🔄 完整自动化流程

阶段一：智能数据同步（每日8:00触发）

步骤1.1：清理本地文件

```
# MCP调用
IntelligentExcelProcessor.clear_content_keep_headers(INPUT_FILE)
```

步骤1.2：AI分析OneDrive表格结构

```
# AI Prompt 1: 表格结构识别
prompt_1 = """
你是一个Excel数据分析专家。请分析这个OneDrive Excel文件的表格结构:

文件路径: {ONEDRIVE_FILE}
表格内容预览:
```

```
{excel_content_preview}

请识别以下信息：
1. 哪一列包含组件名称（component names）- 通常包含软件包、库、工具等名称
2. 哪一列包含DLT分类信息 - 通常是分类标签或状态
3. 哪一列包含文件路径信息 - 通常是GitHub仓库中的文件位置

返回严格的JSON格式：
{
  "component_column": "列号或列名",
  "dlt_column": "列号或列名",
  "path_column": "列号或列名",
  "confidence": "高/中/低",
  "notes": "识别依据说明"
}
"""

# AI分析结果
ai_mapping = claude_analyze(prompt_1, onedrive_content)
```

步骤1.3：执行智能数据映射

```
# 根据AI分析结果进行数据复制
mapping_rules = {
    "source_path_col": ai_mapping["path_column"],      # OneDrive第X列 → File_DLT第1列
    "source_dlt_col": ai_mapping["dlt_column"],        # OneDrive第X列 → File_DLT第2列
    "source_component_col": ai_mapping["component_column"] # OneDrive第X列 → File_DLT第3列
}

# MCP调用
IntelligentExcelProcessor.copy_data_by_mapping(ONEDRIVE_FILE, INPUT_FILE, mapping_rules)
```

阶段二：执行代码扫描

步骤2.1：运行Python脚本

```
# MCP调用
execution_result = ScriptRunner.execute_python_script(MAIN_SCRIPT)

if execution_result.success:
    print(f"脚本执行成功，耗时：{execution_result.duration}秒")
else:
    print(f"脚本执行失败：{execution_result.error}")
    # 触发异常处理流程
```

步骤2.2：AI智能文件查找

```
# AI Prompt 2: 智能文件识别
prompt_2 = """
你是一个文件管理专家。请在指定目录中找到最新生成的DLT统计文件：

目录路径：/Users/jiedong/scan_tool/uitoolkit_scan_tool_resourse/
文件名模式：DLT_stats_macOS_All_YYYYMMDD.xlsx （其中YYYYMMDD是年月日）

当前日期：{current_date}
目录文件列表：
{directory_listing}

请返回：
1. 最新的DLT统计文件完整路径
2. 文件的创建时间
3. 文件大小（用于验证完整性）

返回JSON格式：
{
  "latest_file_path": "完整文件路径",
  "created_time": "创建时间",
  "file_size": "文件大小",
  "confidence": "高/中/低"
}
"""

# AI查找结果
latest_dlt_file = claude_analyze(prompt_2, directory_info)
OUTPUT_FILE = latest_dlt_file["latest_file_path"]
```

阶段三：AI驱动的变更分析

步骤3.1：对比分析两个文件

```
# AI Prompt 3: 深度变更分析
prompt_3 = """
你是一个代码资产分析专家。请深度对比两个Excel文件，分析GitHub代码库的实际情况与人工维护
清单的差异：

**输入文件（人工清单）：** {INPUT_FILE}
- 来源：OneDrive人工维护的组件清单
- 行数：{input_row_count}
- 代表：团队认为存在的组件情况

**输出文件（扫描结果）：** {OUTPUT_FILE}
- 来源：main.py扫描GitHub代码库生成
- 行数：{output_row_count}
```

```
- 代表：GitHub仓库的真实情况

请进行以下分析：

### 1. 数量变化统计
- 新发现组件：在GitHub中存在但人工清单中没有记录的组件
- 已移除组件：人工清单中有但GitHub扫描中未发现的组件
- 保持不变：两边都存在且信息一致的组件

### 2. 内容变化分析
- DLT分类变更：同一组件的DLT分类发生变化的数量
- 路径变更：组件文件路径发生变化的数量
- 其他属性变更：其他字段的变化情况

### 3. 重要发现
- 是否有重要的新增组件？
- 是否有关键组件被意外移除？
- 是否有大量的分类变更？

### 4. 数据质量评估
- 数据一致性如何？
- 是否有异常或可疑的变化？

基于以上分析，生成以下输出：

```json
{
  "summary": {
    "new_components": 数量,
    "removed_components": 数量,
    "unchanged_components": 数量,
    "dlt_changes": 数量,
    "path_changes": 数量
  },
  "key_findings": [
    "关键发现1",
    "关键发现2",
    "关键发现3"
  ],
  "commit_message": "简洁的git提交信息（50字以内）",
  "detailed_report": "详细的变更报告（用于通知团队）"
}
```
```

文件对比数据： {file_comparison_data} """

AI分析结果

change_analysis = claude_analyze(prompt_3, comparison_data)

步骤3.2：生成智能Commit消息
```python
# 使用AI分析结果生成commit消息
commit_message = change_analysis["commit_message"]
detailed_report = change_analysis["detailed_report"]

# 示例输出：
# commit_message: "GitHub scan: +12 new components, -3 removed, 8 DLT
updates"
# detailed_report: "发现12个新组件主要集中在UI toolkit模块..."
```

阶段四：版本控制操作

步骤4.1：Git提交和推送

```python
# 获取当前日期用于commit消息
current_date = datetime.now().strftime("%m/%d")
final_commit_message = f"{commit_message} on {current_date}"

# MCP调用
SmartGitOperations.commit_and_push(final_commit_message)
SmartGitOperations.create_pull_request({
    "title": f"Auto scan update - {current_date}",
    "body": detailed_report,
    "reviewers": ["team-lead", "tech-reviewer"]
})
```

步骤4.2：PR状态监控

```python
# 定期检查PR状态
pr_status = SmartGitOperations.monitor_pr_status()

if pr_status == "merged":
    # 触发下一阶段：远程服务器部署
    trigger_stage_five()
```

阶段五：远程服务器部署（PR合并后触发）

步骤5.1：AI智能识别需要部署的文件

```python
# AI Prompt 4: 智能文件部署识别
prompt_4 = """
你是一个部署管理专家。请分析GitHub仓库中的变更，识别需要部署到远程服务器的文件：
```

```
仓库路径：{LOCAL_REPO}
最近提交的变更文件列表：
{changed_files_list}

请识别以下类型的文件需要部署：
1. DLT统计文件：DLT_stats_macOS_All_*.xlsx
2. 配置文件：*.json, *.yaml, *.conf
3. 脚本文件：*.py, *.sh （如果有更新）
4. 其他重要的输出文件

服务器部署规则：
- DLT统计文件 → /opt/scan_results/latest/
- 配置文件 → /opt/scan_tool/config/
- 脚本文件 → /opt/scan_tool/scripts/
- 其他文件 → /opt/scan_tool/data/

返回JSON格式：
{
  "deploy_files": [
    {
      "source_path": "本地文件相对路径",
      "target_path": "服务器目标路径",
      "file_type": "文件类型",
      "priority": "high/medium/low"
    }
  ],
  "total_files": "需要部署的文件总数",
  "estimated_size": "预估传输大小"
}
"""

# AI分析需要部署的文件
deploy_plan = claude_analyze(prompt_4, git_changes_info)
```

步骤5.2：远程服务器同步操作

```
# MCP调用：RemoteServerManager服务
class RemoteServerManager:
    def __init__(self, server_config):
        self.host = server_config["host"]
        self.username = server_config["username"]
        self.key_path = server_config["ssh_key_path"]
        self.repo_path = server_config["remote_repo_path"]

    def sync_repository(self):
        """在远程服务器执行git pull操作"""
        commands = [
            f"cd {self.repo_path}",
            "git stash",  # 暂存本地变更（如果有）
            "git pull origin main",
            "git stash pop",  # 恢复本地变更
```

```
        ]
        return self.execute_remote_commands(commands)

    def deploy_files(self, deploy_plan):
        """根据AI分析结果部署文件"""
        for file_info in deploy_plan["deploy_files"]:
            source = os.path.join(self.repo_path,
file_info["source_path"])
            target = file_info["target_path"]

            # 创建目标目录（如果不存在）
            target_dir = os.path.dirname(target)
            self.execute_remote_commands([f"mkdir -p {target_dir}"])

            # 复制文件到目标位置
            copy_cmd = f"cp {source} {target}"
            result = self.execute_remote_commands([copy_cmd])

            if result.success:
                logger.info(f"✅ 文件部署成功: {file_info['source_path']} →
{target}")
            else:
                logger.error(f"❌ 文件部署失败: {file_info['source_path']}")

# 服务器配置
server_config = {
    "host": "production-server.company.com",
    "username": "deploy_user",
    "ssh_key_path": "/Users/jiedong/.ssh/id_rsa",
    "remote_repo_path": "/opt/scan_tool"
}

# 执行远程部署
remote_manager = RemoteServerManager(server_config)

# 1. 先同步GitHub仓库
sync_result = remote_manager.sync_repository()

if sync_result.success:
    # 2. 根据AI分析部署特定文件
    deploy_result = remote_manager.deploy_files(deploy_plan)

    if deploy_result.success:
        logger.info("🎉 远程服务器部署完成")
    else:
        logger.error("❌ 文件部署过程中出现错误")
else:
    logger.error("❌ Git同步失败，跳过文件部署")
```

步骤5.3：部署验证与状态确认

```
# AI Prompt 5: 部署结果验证
prompt_5 = """
你是一个系统运维专家。请分析远程服务器的部署结果：

部署计划：
{deploy_plan}

执行结果：
{deployment_results}

服务器文件状态：
{server_file_status}

请验证：
1. 所有计划部署的文件是否都成功复制到目标位置？
2. 文件大小和时间戳是否正确？
3. 是否有权限问题？
4. 是否有文件冲突或覆盖问题？

生成部署报告：
{
  "deployment_status": "success/partial/failed",
  "successful_files": 数量,
  "failed_files": 数量,
  "issues_found": ["问题列表"],
  "recommendations": ["建议措施"],
  "next_actions": ["后续动作"]
}
"""

# AI验证部署结果
deployment_verification = claude_analyze(prompt_5, deployment_info)
```

阶段六：结果反馈与通知

步骤6.1：清理OneDrive源文件

```
# 清空OneDrive文件内容，保留表头
IntelligentExcelProcessor.clear_content_keep_headers(ONEDRIVE_FILE)
```

步骤6.2：回写最新结果

```
# 将GitHub扫描的最新结果复制到OneDrive
# 这样团队成员就能看到最新的、准确的组件清单
IntelligentExcelProcessor.copy_all_data(OUTPUT_FILE, ONEDRIVE_FILE)
```

步骤6.3：发送完成通知

```
# AI生成流程完成报告
completion_report = f"""
🎉 完整自动化部署流程已完成!

📊 本次更新统计:
- 新发现组件: {change_analysis['summary']['new_components']}个
- 已移除组件: {change_analysis['summary']['removed_components']}个
- DLT分类更新: {change_analysis['summary']['dlt_changes']}个
- 路径变更: {change_analysis['summary']['path_changes']}个

📝 关键发现:
{chr(10).join(change_analysis['key_findings'])}

🚀 部署状态:
- 远程服务器同步: ✅ 成功
- 文件部署数量: {deployment_verification['successful_files']}个
- 部署失败文件: {deployment_verification['failed_files']}个
- 整体部署状态: {deployment_verification['deployment_status']}

📁 文件更新:
- OneDrive清单已更新为最新扫描结果
- GitHub PR已成功合并
- 远程服务器文件已同步部署
- 所有数据已同步完成

⚠ 注意事项:
{chr(10).join(deployment_verification.get('recommendations', []))}

下次自动扫描时间: 明日8:00 AM
"""

# 发送通知（邮件、Slack、Teams等）
NotificationService.send_completion_report(completion_report)
```

🚨 异常处理机制

1. 表格结构识别失败

```
if ai_mapping["confidence"] == "低":
    # 使用默认映射规则作为备选
    fallback_mapping = {
        "component_column": "A",
        "dlt_column": "B",
        "path_column": "C"
    }
    # 记录日志并通知管理员
```

2. 文件找不到或损坏

```
if not latest_dlt_file["confidence"] == "高":
    # 手动指定最可能的文件
    # 发送告警通知
    # 记录详细日志用于调试
```

3. main.py执行失败

```
if not execution_result.success:
    # 重试机制（最多3次）
    # 发送紧急通知
    # 保存执行日志用于调试
```

4. 远程服务器Git同步失败

```
if not sync_result.success:
    # SSH连接问题诊断
    connection_test = remote_manager.test_connection()

    if not connection_test.success:
        # 网络或认证问题
        NotificationService.send_urgent_alert("远程服务器连接失败")
        # 尝试备用服务器或手动干预
    else:
        # Git仓库问题，尝试重置
        remote_manager.execute_remote_commands([
            f"cd {repo_path}",
            "git reset --hard HEAD",
            "git clean -fd",
            "git pull origin main"
        ])
```

5. 文件部署失败

```
if deployment_verification["deployment_status"] != "success":
    # 分析失败原因
    for issue in deployment_verification["issues_found"]:
        if "permission" in issue.lower():
            # 权限问题，尝试修复
            remote_manager.fix_file_permissions()
        elif "space" in issue.lower():
            # 磁盘空间不足
            remote_manager.cleanup_old_files()
        else:
```

```
            # 其他问题，记录并通知
            logger.error(f"部署问题: {issue}")
```

📈 成功指标

- **准确率**：AI表格结构识别准确率 > 95%
- **效率**：整个流程执行时间 < 15分钟（包含远程部署）
- **可靠性**：成功执行率 > 98%
- **智能化**：Commit消息描述准确性 > 90%
- **部署成功率**：远程文件部署成功率 > 95%

🔧 部署要求

技术栈

- **MCP SDK**：用于服务间通信
- **Python 3.8+**：主要开发语言
- **openpyxl**：Excel文件处理
- **GitPython**：Git操作自动化
- **paramiko/fabric**：SSH远程连接和操作
- **Claude API**：AI分析服务

环境配置

- **本地Git仓库**：已配置SSH密钥
- **OneDrive同步**：已设置本地同步文件夹
- **远程服务器访问**：SSH密钥配置，sudo权限
- **定时任务**：系统cron或任务调度器
- **网络访问**：GitHub API、Claude API、远程服务器访问权限

📋 监控和日志

日志记录

```
logging_config = {
    "level": "INFO",
    "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s",
    "handlers": [
        "console",
        "file:/var/log/ai-scan-workflow.log",
        "slack://webhook_url"
    ]
}
```

关键监控指标

- AI分析响应时间

- 文件同步成功率
- Git操作成功率
- 远程服务器连接稳定性
- 文件部署成功率
- 整体流程完成时间
- 异常发生频率

该工作流程通过AI智能分析和MCP协议，实现了从手动维护到全自动化的革命性转变，包含完整的远程部署能力，让团队专注于更有价值的工作。