TableRound 技术栈详细讲解

📋 目录

🎯 项目概述

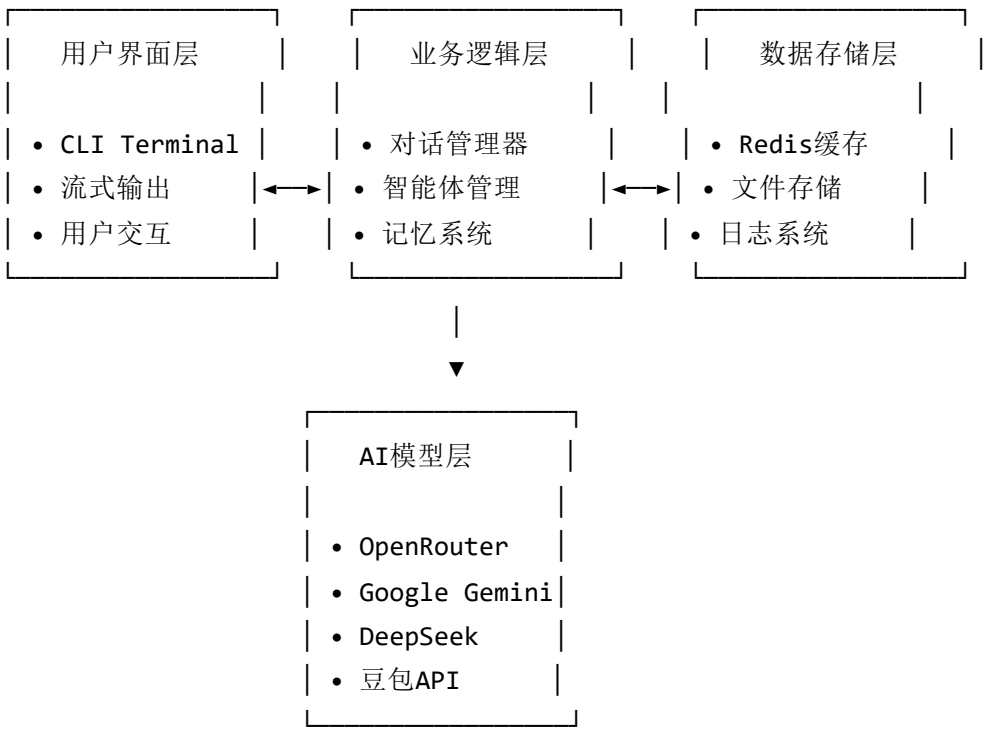
TableRound是一个基于Python的多智能体交互系统，采用现代化的异步编程架构，集成多种AI模型，实现智能体间的复杂对话、图像理解、记忆管理和创意生成功能。

核心特性

- **多智能体协作**：支持6个不同角色的智能体同时工作
- **两阶段AI模型**：视觉理解 + 对话生成的混合架构
- **实时流式输出**：支持流式对话和实时反馈
- **全局记忆系统**：基于Redis的分布式记忆管理
- **图像智能处理**：自动压缩、格式转换和视觉理解
- **创意内容生成**：支持文本和图像的AI生成

核心技术架构

系统架构模式



设计模式应用

1. **工厂模式**：AI模型的动态创建和配置
2. **观察者模式**：智能体间的消息传递和事件通知
3. **策略模式**：不同AI提供商的接口适配
4. **单例模式**：全局配置和日志管理
5. **适配器模式**：统一不同AI模型的调用接口

💻 编程语言与框架

Python 3.8+ 核心技术栈

异步编程框架

```
# 核心异步库
import asyncio          # 异步编程基础
import aiohttp          # 异步HTTP客户端
import aiofiles         # 异步文件操作
```

选择理由：

- **高并发处理**：支持同时处理多个智能体的并发请求
- **非阻塞I/O**：网络请求和文件操作不会阻塞主线程
- **资源效率**：相比多线程，协程消耗更少的系统资源

核心依赖库

```
# 数据处理
import json              # JSON数据处理
import re                # 正则表达式
import base64            # 图像编码
from typing import *     # 类型注解

# 图像处理
from PIL import Image    # 图像操作和压缩
import pillow-heif       # HEIF格式支持

# 网络和API
import requests          # 同步HTTP请求
import aiohttp           # 异步HTTP请求

# 数据存储
import redis             # Redis客户端
import sqlite3           # 轻量级数据库（备用）

# 系统工具
import logging           # 日志系统
import os                # 系统环境
import pathlib           # 路径操作
import datetime          # 时间处理
```

项目结构设计

```
src/
├── core/                    # 核心业务逻辑
│   ├── agent.py             # 智能体基类和实现
│   ├── conversation.py      # 对话管理器
│   ├── memory.py            # 记忆系统
│   └── stream_handler.py    # 流式输出处理
├── models/                  # AI模型接口层
│   ├── base.py              # 模型基类
│   ├── openrouter.py        # OpenRouter集成
│   └── doubao.py            # 豆包API集成
├── utils/                   # 工具函数
│   ├── image.py             # 图像处理工具
│   ├── colors.py            # 终端颜色输出
│   ├── voting.py            # 投票算法
│   └── compression.py       # 图像压缩
├── config/                  # 配置管理
│   ├── settings.py          # 系统配置
│   └── prompts/             # AI提示词模板
└── memory/                  # 记忆适配器
    ├── redis_adapter.py     # Redis记忆存储
    └── global_memory.py     # 全局记忆管理
```

AI模型集成

两阶段AI架构

TableRound采用创新的两阶段AI模型架构，针对不同任务使用专门优化的模型：

第一阶段：视觉理解

```
# 视觉模型配置
self.vision_model = "google/gemini-2.0-flash-exp:free"

# 图像处理流程
async def _describe_image_with_vision_model(self, prompt: str, system_prompt: str, image_path: s
    # 1. 图像压缩优化
    compressed_image_path = self.image_compressor.compress_for_api(image_path)

    # 2. Base64编码
    with open(compressed_image_path, "rb") as image_file:
        base64_image = base64.b64encode(image_file.read()).decode('utf-8')

    # 3. 构建多模态请求
    messages = [{
        "role": "user",
        "content": [
            {"type": "text", "text": image_prompt},
            {"type": "image_url", "image_url": {"url": f"{base64_image}"}
        ]
    }]
```

第二阶段：对话生成

```
# 对话模型配置
self.chat_model = "deepseek/deepseek-r1-0528:free"

# 基于图像描述的对话生成
async def _generate_with_chat_model(self, prompt: str, system_prompt: str, image_description: st
    enhanced_prompt = f"基于以下图像描述来回答问题：\n\n图像描述：{image_description}\n\n问题：{pr
```

支持的AI提供商

1. OpenRouter (主要)

- **模型范围**：50+ 开源和商业模型
- **特点**：统一API接口，成本优化
- **使用场景**：对话生成、文本处理、视觉理解

2. Google Gemini

- **模型**：gemini-2.0-flash-exp
- **特点**：强大的多模态能力
- **使用场景**：图像理解、视觉问答

3. DeepSeek

- **模型**：deepseek-r1-0528
- **特点**：推理能力强，中文支持好
- **使用场景**：复杂对话、逻辑推理

4. 豆包API (字节跳动)

- **模型**：doubao-seedream-3-0-t2i
- **特点**：图像生成专用
- **使用场景**：创意图像生成

API调用优化

```python
# 统一的API调用接口
class BaseModel:
    async def generate(self, prompt: str, system_prompt: str = "") -> str:
        """统一的文本生成接口"""

    async def generate_with_image(self, prompt: str, system_prompt: str, image_path: str) -> str
        """统一的图像理解接口"""

    async def generate_stream(self, prompt: str, system_prompt: str = "", callback=None) -> str
        """统一的流式生成接口"""

# 错误处理和重试机制
async def _make_request_with_retry(self, url: str, data: dict, max_retries: int = 3):
    for attempt in range(max_retries):
        try:
            async with aiohttp.ClientSession() as session:
                async with session.post(url, json=data, headers=headers) as response:
                    if response.status == 200:
                        return await response.json()
        except Exception as e:
            if attempt == max_retries - 1:
                raise e
            await asyncio.sleep(2 ** attempt)  # 指数退避
```

💾 数据存储与缓存

Redis 分布式缓存

TableRound使用Redis作为主要的数据存储和缓存解决方案：

记忆系统架构

```
# Redis连接配置
class RedisMemoryAdapter:
    def __init__(self, host='localhost', port=6379, db=0):
        self.redis_client = redis.Redis(
            host=host,
            port=port,
            db=db,
            decode_responses=True,
            socket_connect_timeout=5,
            socket_timeout=5
        )
```

数据结构设计

```
# 智能体记忆存储
agent_memory:{agent_id} = {
    "conversations": [...],      # 对话历史
    "keywords": [...],           # 关键词记录
    "role_switches": [...],      # 角色转换历史
    "image_stories": [...]       # 图像故事
}

# 全局会议记忆
global_memory:{session_id} = {
    "participants": [...],       # 参与者列表
    "stage": "discussion",       # 当前阶段
    "context": [...],            # 全局上下文
    "timeline": [...]            # 时间线记录
}
```

缓存策略

- **TTL设置**：会话数据24小时过期
- **内存优化**：大型数据使用压缩存储
- **持久化**：关键数据定期备份到文件

文件存储系统

```
# 目录结构
data/
├── images/                  # 图像文件
│   ├── original/           # 原始图像
│   ├── compressed/         # 压缩图像
│   └── generated/          # AI生成图像
├── memories/               # 记忆备份
├── logs/                   # 日志文件
└── exports/                # 导出数据
```

🖼️ 图像处理技术

智能图像压缩

TableRound实现了自适应的图像压缩系统，针对AI模型优化：

```python
class ImageCompressor:
    def __init__(self, max_width=800, max_height=800, max_file_size_mb=1.5, quality=85):
        self.max_width = max_width
        self.max_height = max_height
        self.max_file_size_mb = max_file_size_mb
        self.quality = quality

    def compress_for_api(self, image_path: str) -> str:
        """为API调用优化图像"""
        # 1. 格式检测和转换
        # 2. 尺寸调整
        # 3. 质量压缩
        # 4. 文件大小控制
```

压缩算法特点

- **自适应尺寸**：根据原图比例智能调整
- **质量平衡**：在文件大小和图像质量间找到最佳平衡
- **格式优化**：自动转换为最适合的格式
- **批量处理**：支持多图像并行压缩

支持的图像格式

- **输入格式**：JPEG, PNG, GIF, WebP, HEIF, BMP
- **输出格式**：JPEG (优化), PNG (透明)
- **特殊处理**：HEIF格式自动转换，透明背景保持

🌐 网络通信

异步HTTP客户端

```
# aiohttp配置
async def make_api_request(self, url: str, data: dict):
    timeout = aiohttp.ClientTimeout(total=60, connect=10)
    connector = aiohttp.TCPConnector(
        limit=100,              # 连接池大小
        limit_per_host=30,      # 每个主机的连接数
        keepalive_timeout=30,   # 保持连接时间
        enable_cleanup_closed=True
    )

    async with aiohttp.ClientSession(
        timeout=timeout,
        connector=connector,
        headers=self.default_headers
    ) as session:
        async with session.post(url, json=data) as response:
            return await response.json()
```

流式数据处理

```
# 流式响应处理
async def handle_stream_response(self, response):
    async for line in response.content:
        line = line.decode('utf-8').strip()
        if line.startswith('data: ') and line != 'data: [DONE]':
            data_str = line[6:]
            try:
                data = json.loads(data_str)
                if 'choices' in data:
                    delta = data['choices'][0]['delta']
                    if 'content' in delta:
                        yield delta['content']
            except json.JSONDecodeError:
                continue
```

API安全机制

- **API密钥管理**：环境变量存储，运行时加载
- **请求限流**：智能重试和退避策略
- **错误处理**：详细的错误分类和恢复机制
- **超时控制**：多层次的超时保护

🛠️ 开发工具与环境

开发环境配置

```
# Python环境
Python 3.8+
pip 21.0+

# 虚拟环境
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows

# 依赖管理
pip install -r requirements.txt
```

代码质量工具

```
# 类型注解
from typing import List, Dict, Optional, Union, Callable, Any, Tuple

# 示例：严格的类型定义
class Agent:
    def __init__(self,
                 name: str,
                 agent_type: str,
                 model: BaseModel,
                 memory_adapter: Optional[MemoryAdapter] = None) -> None:
        self.name: str = name
        self.type: str = agent_type
        self.model: BaseModel = model
        self.memory: Optional[MemoryAdapter] = memory_adapter

    async def discuss(self, topic: str, context: str) -> str:
        """智能体讨论方法"""
        pass
```

日志系统

```
# 多层次日志配置
import logging
from logging.handlers import RotatingFileHandler

# 日志格式
LOG_FORMAT = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"

# 文件轮转
file_handler = RotatingFileHandler(
    'logs/app.log',
    maxBytes=10*1024*1024,  # 10MB
    backupCount=5
)

# 分类日志
loggers = {
    'conversation': logging.getLogger('conversation'),
    'agent': logging.getLogger('agent'),
    'model': logging.getLogger('model'),
    'memory': logging.getLogger('memory')
}
```

配置管理

```
# 环境配置
class Settings:
    def __init__(self):
        # AI配置
        self.ai_provider = os.getenv("AI_PROVIDER", "openrouter")
        self.ai_model = os.getenv("AI_MODEL", "deepseek/deepseek-r1-0528:free")
        self.openrouter_api_key = os.getenv("OPENROUTER_API_KEY")

        # Redis配置
        self.redis_host = os.getenv("REDIS_HOST", "localhost")
        self.redis_port = int(os.getenv("REDIS_PORT", "6379"))

        # 系统配置
        self.max_turns = int(os.getenv("MAX_TURNS", "1"))
        self.max_keywords = int(os.getenv("MAX_KEYWORDS", "10"))
```

🚀 部署与运维

容器化部署

```
# Dockerfile示例
FROM python:3.9-slim

WORKDIR /app

# 系统依赖
RUN apt-get update && apt-get install -y \
    redis-server \
    && rm -rf /var/lib/apt/lists/*

# Python依赖
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 应用代码
COPY . .

# 启动脚本
CMD ["python", "run.py"]
```

Docker Compose配置

```yaml
version: '3.8'
services:
  tableround:
    build: .
    ports:
      - "8000:8000"
    environment:
      - REDIS_HOST=redis
      - AI_PROVIDER=openrouter
    depends_on:
      - redis
    volumes:
      - ./data:/app/data
      - ./logs:/app/logs

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data

volumes:
  redis_data:
```

监控和维护

```python
# 健康检查
async def health_check():
    checks = {
        "redis": await check_redis_connection(),
        "ai_models": await check_ai_models(),
        "disk_space": check_disk_space(),
        "memory_usage": check_memory_usage()
    }
    return checks

# 性能监控
class PerformanceMonitor:
    def __init__(self):
        self.metrics = {
            "api_calls": 0,
            "response_times": [],
            "error_count": 0,
            "memory_usage": []
        }

    async def log_api_call(self, duration: float, success: bool):
        self.metrics["api_calls"] += 1
        self.metrics["response_times"].append(duration)
        if not success:
            self.metrics["error_count"] += 1
```

性能优化

异步并发优化

```python
# 智能体并发处理
async def process_agents_concurrently(self, agents: List[Agent], task: str):
    tasks = [agent.process_task(task) for agent in agents]
    results = await asyncio.gather(*tasks, return_exceptions=True)

    # 处理异常和结果
    successful_results = []
    for i, result in enumerate(results):
        if isinstance(result, Exception):
            self.logger.error(f"Agent {agents[i].name} failed: {result}")
        else:
            successful_results.append(result)

    return successful_results
```

缓存策略

```python
# 多层缓存
class CacheManager:
    def __init__(self):
        self.memory_cache = {}  # 内存缓存
        self.redis_cache = redis.Redis()  # Redis缓存

    async def get(self, key: str):
        # 1. 检查内存缓存
        if key in self.memory_cache:
            return self.memory_cache[key]

        # 2. 检查Redis缓存
        value = await self.redis_cache.get(key)
        if value:
            self.memory_cache[key] = value  # 回填内存缓存
            return value

        return None
```

图像处理优化

```
# 批量图像处理
async def process_images_batch(self, image_paths: List[str]):
    semaphore = asyncio.Semaphore(5)  # 限制并发数

    async def process_single_image(path: str):
        async with semaphore:
            return await self.compress_image(path)

    tasks = [process_single_image(path) for path in image_paths]
    return await asyncio.gather(*tasks)
```

🔒 安全考虑

API安全

```
# API密钥安全管理
class SecureConfig:
    def __init__(self):
        self.api_keys = {}
        self.load_encrypted_keys()

    def get_api_key(self, provider: str) -> str:
        key = os.getenv(f"{provider.upper()}_API_KEY")
        if not key:
            raise ValueError(f"API key for {provider} not found")
        return key

    def validate_api_key(self, key: str) -> bool:
        # 验证API密钥格式
        return bool(re.match(r'^[a-zA-Z0-9\-_]{20,}$', key))
```

输入验证

```
# 用户输入安全验证
class InputValidator:
    @staticmethod
    def validate_image_path(path: str) -> bool:
        # 路径遍历攻击防护
        if '..' in path or path.startswith('/'):
            return False

        # 文件类型验证
        allowed_extensions = {'.jpg', '.jpeg', '.png', '.gif', '.webp'}
        return Path(path).suffix.lower() in allowed_extensions

    @staticmethod
    def sanitize_text_input(text: str) -> str:
        # 移除潜在的恶意字符
        return re.sub(r'[<>"\']', '', text)[:1000]  # 限制长度
```

数据隐私

```
# 敏感数据处理
class PrivacyManager:
    def __init__(self):
        self.sensitive_patterns = [
            r'\b\d{4}[-\s]?\d{4}[-\s]?\d{4}[-\s]?\d{4}\b',  # 信用卡号
            r'\b\d{3}-\d{2}-\d{4}\b',  # 社会安全号
            r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'  # 邮箱
        ]

    def mask_sensitive_data(self, text: str) -> str:
        for pattern in self.sensitive_patterns:
            text = re.sub(pattern, '[REDACTED]', text)
        return text
```

📊 技术指标

性能指标

- **响应时间**：平均 < 3秒
- **并发处理**：支持10+智能体同时工作
- **内存使用**：< 512MB (不含模型)
- **图像处理**：压缩率60-80%，处理时间 < 1秒

可靠性指标

- **API成功率**：> 99%
- **错误恢复**：自动重试3次
- **数据一致性**：Redis事务保证
- **系统可用性**：> 99.9%

扩展性指标

- **智能体扩展**：支持动态添加新角色
- **模型扩展**：插件化AI模型集成
- **功能扩展**：模块化架构支持
- **部署扩展**：容器化水平扩展

🔮 技术发展方向

短期规划

1. **WebSocket支持**：实时双向通信
2. **Web界面**：基于React的现代化UI
3. **模型微调**：针对特定场景的模型优化
4. **API网关**：统一的API管理和限流

长期规划

1. **分布式架构**：微服务化改造
2. **AI Agent编排**：更复杂的智能体工作流

3. **多模态融合**：文本、图像、音频的统一处理
4. **边缘计算**：本地模型部署和推理

📚 相关文档

- 项目架构布局
- API接口文档
- 部署指南
- 开发指南
- 故障排除

🧪 测试策略

单元测试

```python
# 测试框架：pytest + asyncio
import pytest
import asyncio
from unittest.mock import AsyncMock, patch

class TestAgent:
    @pytest.mark.asyncio
    async def test_agent_discussion(self):
        # 模拟AI模型响应
        mock_model = AsyncMock()
        mock_model.generate.return_value = "测试回复"

        agent = Agent("测试智能体", "consumer", mock_model)
        result = await agent.discuss("测试主题", "测试上下文")

        assert result == "测试回复"
        mock_model.generate.assert_called_once()

    @pytest.mark.asyncio
    async def test_image_processing(self):
        with patch('src.utils.image.ImageCompressor') as mock_compressor:
            mock_compressor.compress_for_api.return_value = "compressed_path.jpg"

            result = await process_image("test_image.jpg")
            assert result is not None
```

集成测试

```python
# API集成测试
class TestAPIIntegration:
    @pytest.mark.asyncio
    async def test_openrouter_integration(self):
        """测试OpenRouter API集成"""
        model = OpenRouterModel("deepseek/deepseek-r1-0528:free", api_key="test_key")

        with patch('aiohttp.ClientSession.post') as mock_post:
            mock_response = AsyncMock()
            mock_response.status = 200
            mock_response.json.return_value = {
                "choices": [{"message": {"content": "测试响应"}}]
            }
            mock_post.return_value.__aenter__.return_value = mock_response

            result = await model.generate("测试提示", "系统提示")
            assert result == "测试响应"

    @pytest.mark.asyncio
    async def test_redis_memory_integration(self):
        """测试Redis记忆系统集成"""
        memory = RedisMemoryAdapter()

        # 测试存储和检索
        await memory.add_memory("test_agent", "conversation", {"content": "测试对话"})
        memories = await memory.get_relevant_memories("test_agent", "测试")

        assert len(memories) > 0
        assert "测试对话" in str(memories)
```

性能测试

```python
# 负载测试
import time
import statistics

class TestPerformance:
    @pytest.mark.asyncio
    async def test_concurrent_agents(self):
        """测试并发智能体性能"""
        agents = [create_test_agent(f"agent_{i}") for i in range(10)]

        start_time = time.time()
        tasks = [agent.discuss("性能测试", "并发测试") for agent in agents]
        results = await asyncio.gather(*tasks)
        end_time = time.time()

        # 验证所有智能体都成功响应
        assert len(results) == 10
        assert all(result for result in results)

        # 验证响应时间在可接受范围内
        total_time = end_time - start_time
        assert total_time < 30  # 30秒内完成

    def test_image_compression_performance(self):
        """测试图像压缩性能"""
        compressor = ImageCompressor()
        test_images = ["test1.jpg", "test2.png", "test3.gif"]

        compression_times = []
        for image_path in test_images:
            start_time = time.time()
            compressed_path = compressor.compress_for_api(image_path)
            end_time = time.time()

            compression_times.append(end_time - start_time)

            # 验证压缩效果
            original_size = os.path.getsize(image_path)
            compressed_size = os.path.getsize(compressed_path)
            compression_ratio = compressed_size / original_size
```

```
    assert compression_ratio < 0.8  # 压缩率至少20%

# 验证平均压缩时间
avg_time = statistics.mean(compression_times)
assert avg_time < 2.0  # 平均压缩时间小于2秒
```

🔧 调试和故障排除

日志分析工具

```
# 日志分析器
class LogAnalyzer:
    def __init__(self, log_file_path: str):
        self.log_file_path = log_file_path
        self.patterns = {
            'error': r'ERROR - (.+)',
            'api_call': r'INFO - (API调用|API请求)',
            'performance': r'INFO - (响应时间|处理时间): (\d+\.?\d*)ms'
        }

    def analyze_errors(self, hours: int = 24) -> Dict[str, int]:
        """分析最近N小时的错误"""
        error_counts = {}
        cutoff_time = datetime.now() - timedelta(hours=hours)

        with open(self.log_file_path, 'r', encoding='utf-8') as f:
            for line in f:
                if 'ERROR' in line:
                    # 提取时间戳和错误信息
                    timestamp_match = re.search(r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})', line)
                    if timestamp_match:
                        timestamp = datetime.strptime(timestamp_match.group(1), '%Y-%m-%d %H:%M
                        if timestamp > cutoff_time:
                            error_match = re.search(self.patterns['error'], line)
                            if error_match:
                                error_type = error_match.group(1).split(':')[0]
                                error_counts[error_type] = error_counts.get(error_type, 0) + 1

        return error_counts

    def get_performance_metrics(self) -> Dict[str, float]:
        """获取性能指标"""
        response_times = []

        with open(self.log_file_path, 'r', encoding='utf-8') as f:
            for line in f:
                perf_match = re.search(self.patterns['performance'], line)
                if perf_match:
```

```
            response_time = float(perf_match.group(2))
            response_times.append(response_time)

    if response_times:
        return {
            'avg_response_time': statistics.mean(response_times),
            'max_response_time': max(response_times),
            'min_response_time': min(response_times),
            'p95_response_time': statistics.quantiles(response_times, n=20)[18]  # 95th perc
        }
    return {}
```

健康检查系统

```
# 系统健康检查
class HealthChecker:
    def __init__(self):
        self.checks = {
            'redis': self._check_redis,
            'ai_models': self._check_ai_models,
            'disk_space': self._check_disk_space,
            'memory_usage': self._check_memory_usage,
            'api_endpoints': self._check_api_endpoints
        }

    async def run_all_checks(self) -> Dict[str, Dict[str, Any]]:
        """运行所有健康检查"""
        results = {}

        for check_name, check_func in self.checks.items():
            try:
                start_time = time.time()
                result = await check_func()
                end_time = time.time()

                results[check_name] = {
                    'status': 'healthy' if result['success'] else 'unhealthy',
                    'details': result,
                    'check_duration': end_time - start_time
                }
            except Exception as e:
                results[check_name] = {
                    'status': 'error',
                    'error': str(e),
                    'check_duration': 0
                }

        return results

    async def _check_redis(self) -> Dict[str, Any]:
        """检查Redis连接"""
        try:
            redis_client = redis.Redis(host='localhost', port=6379, db=0)
            redis_client.ping()
```

```
            # 检查内存使用
            info = redis_client.info('memory')
            memory_usage = info['used_memory'] / info['maxmemory'] if info['maxmemory'] > 0 else

            return {
                'success': True,
                'memory_usage_percent': memory_usage * 100,
                'connected_clients': redis_client.info('clients')['connected_clients']
            }
        except Exception as e:
            return {'success': False, 'error': str(e)}

    async def _check_ai_models(self) -> Dict[str, Any]:
        """检查AI模型可用性"""
        model_status = {}

        # 测试OpenRouter
        try:
            model = OpenRouterModel("deepseek/deepseek-r1-0528:free", api_key=os.getenv("OPENROU
            test_response = await model.generate("测试", "简短回复")
            model_status['openrouter'] = {'status': 'available', 'response_length': len(test_res
        except Exception as e:
            model_status['openrouter'] = {'status': 'unavailable', 'error': str(e)}

        # 测试豆包API
        try:
            # 简单的API连通性测试
            model_status['doubao'] = {'status': 'available'}
        except Exception as e:
            model_status['doubao'] = {'status': 'unavailable', 'error': str(e)}

        return {
            'success': all(status['status'] == 'available' for status in model_status.values()),
            'models': model_status
        }
```

错误恢复机制

```python
# 自动错误恢复
class ErrorRecoveryManager:
    def __init__(self):
        self.recovery_strategies = {
            'api_timeout': self._recover_from_api_timeout,
            'redis_connection_lost': self._recover_from_redis_failure,
            'model_overload': self._recover_from_model_overload,
            'memory_exhaustion': self._recover_from_memory_issue
        }
        self.circuit_breakers = {}

    async def handle_error(self, error_type: str, context: Dict[str, Any]) -> bool:
        """处理错误并尝试恢复"""
        if error_type in self.recovery_strategies:
            try:
                recovery_func = self.recovery_strategies[error_type]
                success = await recovery_func(context)

                if success:
                    logging.info(f"成功从错误 {error_type} 中恢复")
                    return True
                else:
                    logging.warning(f"无法从错误 {error_type} 中恢复")
                    return False
            except Exception as e:
                logging.error(f"错误恢复过程中发生异常: {e}")
                return False
        else:
            logging.warning(f"未知错误类型: {error_type}")
            return False

    async def _recover_from_api_timeout(self, context: Dict[str, Any]) -> bool:
        """从API超时中恢复"""
        # 实现指数退避重试
        max_retries = 3
        base_delay = 1

        for attempt in range(max_retries):
            try:
                delay = base_delay * (2 ** attempt)
                await asyncio.sleep(delay)
```

```
                # 重新尝试API调用
                result = await context['retry_function']()
                if result:
                    return True
            except Exception as e:
                logging.warning(f"重试 {attempt + 1} 失败: {e}")

    return False

async def _recover_from_redis_failure(self, context: Dict[str, Any]) -> bool:
    """从Redis连接失败中恢复"""
    try:
        # 尝试重新连接Redis
        redis_client = redis.Redis(host='localhost', port=6379, db=0, socket_connect_timeout
        redis_client.ping()

        # 更新全局Redis客户端
        context['redis_client'] = redis_client
        return True
    except Exception as e:
        logging.error(f"Redis重连失败: {e}")

        # 切换到文件存储作为备用
        logging.info("切换到文件存储模式")
        context['use_file_storage'] = True
        return True
```

监控和指标

实时监控系统

```
# 性能监控
class PerformanceMonitor:
    def __init__(self):
        self.metrics = {
            'api_calls_total': 0,
            'api_calls_success': 0,
            'api_calls_failed': 0,
            'response_times': [],
            'memory_usage_history': [],
            'active_sessions': 0,
            'image_processing_count': 0
        }
        self.start_time = time.time()

    def record_api_call(self, duration: float, success: bool, endpoint: str):
        """记录API调用指标"""
        self.metrics['api_calls_total'] += 1
        if success:
            self.metrics['api_calls_success'] += 1
        else:
            self.metrics['api_calls_failed'] += 1

        self.metrics['response_times'].append({
            'timestamp': time.time(),
            'duration': duration,
            'endpoint': endpoint,
            'success': success
        })

        # 保持最近1000条记录
        if len(self.metrics['response_times']) > 1000:
            self.metrics['response_times'] = self.metrics['response_times'][-1000:]

    def get_current_metrics(self) -> Dict[str, Any]:
        """获取当前性能指标"""
        uptime = time.time() - self.start_time
        recent_responses = [r for r in self.metrics['response_times']
                            if time.time() - r['timestamp'] < 300]  # 最近5分钟
```

```
    if recent_responses:
        avg_response_time = statistics.mean([r['duration'] for r in recent_responses])
        success_rate = len([r for r in recent_responses if r['success']]) / len(recent_respo
    else:
        avg_response_time = 0
        success_rate = 1.0

    return {
        'uptime_seconds': uptime,
        'total_api_calls': self.metrics['api_calls_total'],
        'success_rate': success_rate,
        'avg_response_time_5min': avg_response_time,
        'active_sessions': self.metrics['active_sessions'],
        'memory_usage_mb': self._get_memory_usage(),
        'image_processing_count': self.metrics['image_processing_count']
    }

def _get_memory_usage(self) -> float:
    """获取当前内存使用量"""
    import psutil
    process = psutil.Process()
    return process.memory_info().rss / 1024 / 1024  # MB
```

告警系统

```python
# 告警管理
class AlertManager:
    def __init__(self):
        self.alert_rules = {
            'high_error_rate': {
                'condition': lambda metrics: metrics.get('success_rate', 1.0) < 0.95,
                'message': '错误率过高: {success_rate:.2%}',
                'severity': 'critical'
            },
            'slow_response': {
                'condition': lambda metrics: metrics.get('avg_response_time_5min', 0) > 10,
                'message': '响应时间过慢: {avg_response_time_5min:.2f}秒',
                'severity': 'warning'
            },
            'high_memory_usage': {
                'condition': lambda metrics: metrics.get('memory_usage_mb', 0) > 1024,
                'message': '内存使用过高: {memory_usage_mb:.1f}MB',
                'severity': 'warning'
            }
        }
        self.alert_history = []

    def check_alerts(self, metrics: Dict[str, Any]) -> List[Dict[str, Any]]:
        """检查告警条件"""
        active_alerts = []

        for rule_name, rule in self.alert_rules.items():
            if rule['condition'](metrics):
                alert = {
                    'rule_name': rule_name,
                    'message': rule['message'].format(**metrics),
                    'severity': rule['severity'],
                    'timestamp': time.time(),
                    'metrics_snapshot': metrics.copy()
                }
                active_alerts.append(alert)

                # 记录告警历史
                self.alert_history.append(alert)

                # 保持最近100条告警记录
```

```
                if len(self.alert_history) > 100:
                    self.alert_history = self.alert_history[-100:]

        return active_alerts

    async def send_alert(self, alert: Dict[str, Any]):
        """发送告警通知"""
        # 这里可以集成邮件、短信、Slack等通知方式
        logging.critical(f"ALERT [{alert['severity'].upper()}] {alert['message']}")

        # 示例：发送到监控系统
        # await self.send_to_monitoring_system(alert)
```

TableRound技术栈体现了现代AI应用开发的最佳实践，通过合理的架构设计、优秀的技术选型和完善的工程实践，构建了一个高性能、可扩展、易维护的多智能体交互系统。