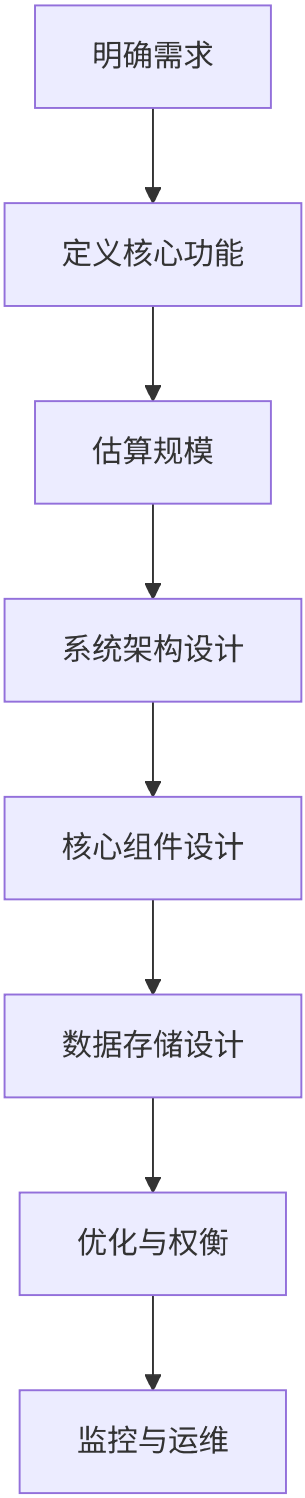
第16章 系统设计题

> 系统设计题考察工程思维和架构能力

16.1 系统设计答题框架

16.1.1 答题步骤



系统设计模板：

```python
class SystemDesignTemplate:
    """
    系统设计答题模板
    """
    def design_system(self, problem):
        """
        系统设计标准流程
        """
        # 1. 需求澄清（5分钟）
        requirements = self.clarify_requirements(problem)

        # 2. 规模估算（5分钟）
        scale = self.estimate_scale(requirements)

        # 3. 高层设计（10分钟）
        architecture = self.design_architecture(requirements, scale)

        # 4. 详细设计（15分钟）
        components = self.design_components(architecture)

        # 5. 优化讨论（10分钟）
        optimizations = self.discuss_optimizations(components)

        return {
            "requirements": requirements,
            "scale": scale,
            "architecture": architecture,
            "components": components,
            "optimizations": optimizations
        }

    def clarify_requirements(self, problem):
        """
        澄清需求

        必问的问题：
        - 用户规模？（DAU/MAU）
        - 核心功能？（MVP）
        - 性能要求？（延迟/吞吐量）
        - 可靠性要求？（SLA）
        - 扩展性要求？（未来规模）
        """
        questions = [
            "What is the expected user scale?",
            "What are the core features?",
            "What are the latency requirements?",
            "What is the target availability?",
            "Do we need to support multiple languages/regions?"
```

```
        ]

        return questions

    def estimate_scale(self, requirements):
        """
        规模估算

        常见估算:
        - QPS = DAU * 平均请求数 / 86400
        - 存储 = 用户数 * 平均数据大小
        - 带宽 = QPS * 平均响应大小
        """
        estimations = {
            "users": {
                "DAU": "100M",
                "peak_QPS": "100k",
                "avg_QPS": "10k"
            },
            "storage": {
                "total": "100TB",
                "daily_growth": "1TB"
            },
            "network": {
                "bandwidth": "10Gbps"
            }
        }

        return estimations
```

16.2 经典题目：设计对话系统

16.2.1 需求分析

题目： 设计一个基于大模型的对话系统，支持百万级DAU

需求澄清：

面试官：设计一个AI对话系统。

候选人：
1. 功能需求：
   - 支持多轮对话吗？ → 是
   - 需要记忆上下文吗？ → 是，至少10轮
   - 支持流式输出吗？ → 是
   - 需要支持插件/工具吗？ → 后续支持

2. 非功能需求：

- DAU多少？ → 1M
- 延迟要求？ → 首Token < 500ms，整体 < 5s
- 可用性？ → 99.9%
- 并发？ → 峰值10k QPS

3. 其他：
- 使用哪个模型？ → GPT-4或类似
- 需要审核吗？ → 是，内容安全

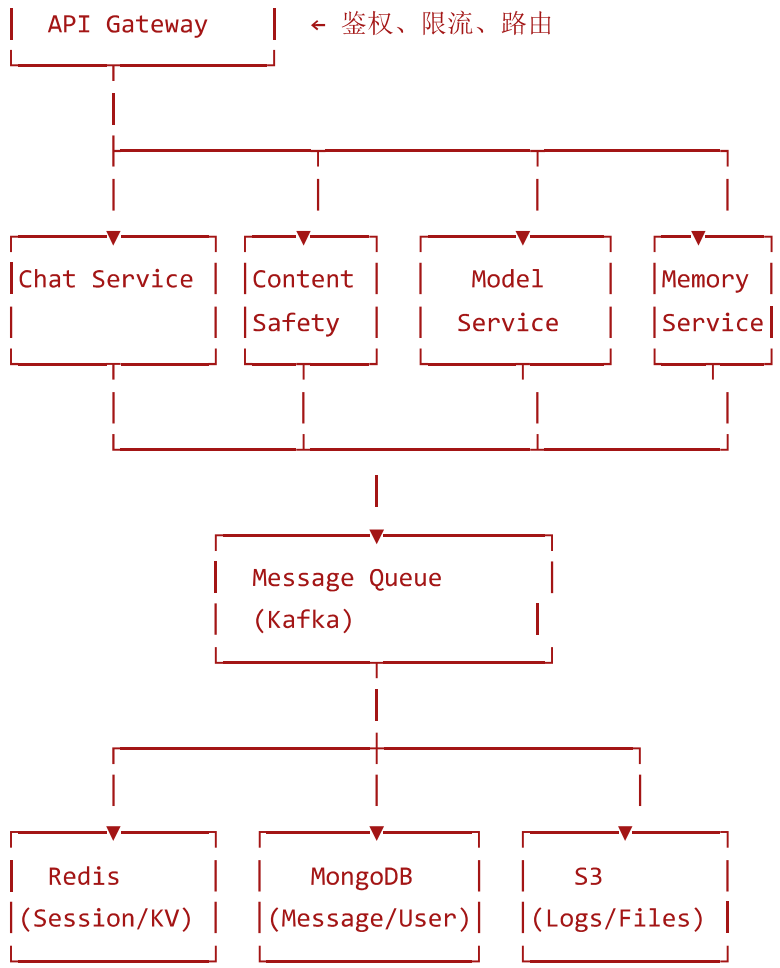
16.2.2 规模估算

```python
class ScaleEstimation:
    """
    对话系统规模估算
    """
    def __init__(self):
        self.dau = 1_000_000  # 100万DAU
        self.sessions_per_user = 3  # 每用户每天3次会话
        self.messages_per_session = 10  # 每次会话10条消息
        self.avg_tokens_per_message = 50  # 每条消息50个token

    def estimate_qps(self):
        """
        估算QPS
        """
        # 每天总消息数
        daily_messages = self.dau * self.sessions_per_user * self.messages_per_session
        # 30M 条/天

        # 平均QPS
        avg_qps = daily_messages / 86400  # ≈ 347 QPS

        # 峰值QPS（假设峰值是平均的3倍）
        peak_qps = avg_qps * 3  # ≈ 1000 QPS

        return {
            "daily_messages": daily_messages,
            "avg_qps": avg_qps,
            "peak_qps": peak_qps
        }

    def estimate_storage(self):
        """
        估算存储
        """
        # 每条消息存储
        bytes_per_message = 500  # 500字节（包含元数据）
```

```
        # 每天新增
        daily_messages = self.dau * self.sessions_per_user * self.messages_per_session
        daily_storage = daily_messages * bytes_per_message  # ≈ 15GB/天

        # 保留1年
        yearly_storage = daily_storage * 365  # ≈ 5.5TB

        return {
            "daily": f"{daily_storage / 1e9:.2f} GB",
            "yearly": f"{yearly_storage / 1e12:.2f} TB"
        }

    def estimate_compute(self):
        """
        估算计算资源
        """
        # 平均每个请求需要的token数
        # 输入：历史10轮 * 50 tokens = 500 tokens
        # 输出：100 tokens
        tokens_per_request = 600

        # 每秒token数
        peak_qps = 1000
        tokens_per_second = peak_qps * tokens_per_request  # 600k tokens/s

        # GPT-4速度约 40 tokens/s/GPU
        # 需要GPU数 ≈ 600k / 40 = 15,000 GPU等效
        # 实际使用批处理可以降低

        return {
            "tokens_per_second": tokens_per_second,
            "estimated_gpus": tokens_per_second / 40
        }
```

16.2.3 系统架构

```
"""
对话系统架构

┌─────────────┐
│   Client    │
└──────┬──────┘
       │
       ▼
┌──────────────────────────────────────────┐
│          Load Balancer (Nginx)           │
└──────┬──────────────────────────────────┘
       │
       ▼
┌──────────────
```




```
"""

class ChatSystemArchitecture:
    """
    对话系统架构设计
    """
    def __init__(self):
        self.components = {
            "frontend": ["Web", "Mobile", "Desktop"],
            "gateway": ["API Gateway", "Load Balancer"],
            "services": [
                "Chat Service",
                "Model Service",
                "Memory Service",
                "Content Safety",
                "Analytics"
            ],
            "infrastructure": [
                "Redis (Cache)",
                "MongoDB (Storage)",
                "Kafka (Queue)",
                "S3 (Objects)"
            ]
        }

    def design_chat_service(self):
```

```
        """
        对话服务设计
        """
        service = {
            "responsibilities": [
                "接收用户消息",
                "管理会话状态",
                "调用模型服务",
                "返回流式响应",
                "记录对话历史"
            ],

            "api_endpoints": {
                "/chat/create": "创建新会话",
                "/chat/send": "发送消息",
                "/chat/stream": "流式接收",
                "/chat/history": "获取历史"
            },

            "tech_stack": {
                "language": "Python/Go",
                "framework": "FastAPI/Gin",
                "async": "asyncio/goroutines"
            }
        }

        return service

    def design_model_service(self):
        """
        模型服务设计
        """
        service = {
            "responsibilities": [
                "模型推理",
                "批处理优化",
                "负载均衡",
                "模型版本管理"
            ],

            "architecture": {
                "framework": "vLLM",
                "serving": "Multiple replicas",
                "gpu": "A100 * N",
                "batching": "Dynamic batching"
            },

            "optimization": [
                "KV Cache",
```

```
                "Flash Attention",
                "Continuous Batching",
                "Speculative Decoding"
            ]
        }

        return service
```

16.2.4 核心代码设计

```
from fastapi import FastAPI, WebSocket
from fastapi.responses import StreamingResponse
import asyncio
import json

class ChatService:
    """
    对话服务实现
    """
    def __init__(self):
        self.app = FastAPI()
        self.memory_service = MemoryService()
        self.model_service = ModelService()
        self.safety_service = SafetyService()

        self.setup_routes()

    def setup_routes(self):
        """
        设置路由
        """
        @self.app.post("/chat/create")
        async def create_session(user_id: str):
            """创建新会话"""
            session_id = generate_session_id()

            # 初始化会话
            await self.memory_service.create_session(session_id, user_id)

            return {"session_id": session_id}

        @self.app.post("/chat/send")
        async def send_message(
            session_id: str,
            message: str
        ):
            """
            发送消息（流式响应）
```

```
        """
        # 1. 安全检查
        is_safe, reason = await self.safety_service.check(message)
        if not is_safe:
            return {"error": reason}

        # 2. 获取上下文
        history = await self.memory_service.get_history(session_id)

        # 3. 生成响应（流式）
        async def generate():
            full_response = ""

            async for chunk in self.model_service.generate_stream(
                message, history
            ):
                full_response += chunk
                yield f"data: {json.dumps({'chunk': chunk})}\n\n"

            # 4. 保存到历史
            await self.memory_service.add_message(
                session_id,
                {"role": "user", "content": message}
            )
            await self.memory_service.add_message(
                session_id,
                {"role": "assistant", "content": full_response}
            )

            yield f"data: {json.dumps({'done': True})}\n\n"

        return StreamingResponse(
            generate(),
            media_type="text/event-stream"
        )


class MemoryService:
    """
    记忆服务：管理对话上下文
    """
    def __init__(self):
        self.redis_client = redis.Redis()
        self.mongo_client = motor.motor_asyncio.AsyncIOMotorClient()

    async def create_session(self, session_id, user_id):
        """
        创建会话
        """
```

```
        # Redis中缓存（快速访问）
        await self.redis_client.hset(
            f"session:{session_id}",
            mapping={
                "user_id": user_id,
                "created_at": time.time(),
                "message_count": 0
            }
        )

        # 设置过期时间（24小时）
        await self.redis_client.expire(f"session:{session_id}", 86400)

    async def get_history(self, session_id, limit=10):
        """
        获取对话历史（最近N轮）

        策略：
        1. 先从Redis尝试获取
        2. 未命中则从MongoDB加载
        3. 加载到Redis
        """
        # 1. 尝试从Redis获取
        cache_key = f"history:{session_id}"
        cached = await self.redis_client.get(cache_key)

        if cached:
            return json.loads(cached)

        # 2. 从MongoDB加载
        db = self.mongo_client["chatbot"]
        messages = await db.messages.find(
            {"session_id": session_id}
        ).sort("timestamp", -1).limit(limit * 2).to_list(None)

        # 反转顺序
        messages.reverse()

        # 3. 缓存到Redis
        await self.redis_client.setex(
            cache_key,
            300,  # 5分钟
            json.dumps(messages)
        )

        return messages

    async def add_message(self, session_id, message):
        """
```

```
        添加消息
        """
        # 1. 保存到MongoDB（持久化）
        db = self.mongo_client["chatbot"]
        await db.messages.insert_one({
            "session_id": session_id,
            "role": message["role"],
            "content": message["content"],
            "timestamp": time.time()
        })

        # 2. 清除Redis缓存（下次重新加载）
        await self.redis_client.delete(f"history:{session_id}")


class ModelService:
    """
    模型服务：封装模型推理
    """
    def __init__(self):
        # 使用vLLM进行推理
        from vllm import LLM, SamplingParams

        self.llm = LLM(
            model="gpt-4",
            tensor_parallel_size=4,  # 4卡并行
            max_num_seqs=256,  # 最大batch size
        )

        self.sampling_params = SamplingParams(
            temperature=0.7,
            top_p=0.9,
            max_tokens=1024
        )

    async def generate_stream(self, message, history):
        """
        流式生成
        """
        # 构建prompt
        prompt = self._build_prompt(message, history)

        # 流式生成
        async for output in self.llm.generate_stream(
            prompt,
            self.sampling_params
        ):
            # 提取新生成的token
            new_text = output.outputs[0].text
```

```
            yield new_text

    def _build_prompt(self, message, history):
        """
        构建prompt
        """
        prompt = "You are a helpful AI assistant.\n\n"

        # 添加历史
        for msg in history[-10:]:  # 最近10轮
            role = msg["role"]
            content = msg["content"]
            prompt += f"{role.capitalize()}: {content}\n"

        # 当前消息
        prompt += f"User: {message}\nAssistant:"

        return prompt


class SafetyService:
    """
    内容安全服务
    """
    def __init__(self):
        from transformers import pipeline

        self.classifier = pipeline(
            "text-classification",
            model="unitary/toxic-bert"
        )

    async def check(self, text):
        """
        检查内容安全

        Returns:
            is_safe: bool
            reason: str
        """
        # 1. 关键词检查（快速）
        if self._keyword_filter(text):
            return False, "包含敏感词"

        # 2. 模型检测（准确）
        result = self.classifier(text)[0]

        if result["label"] == "toxic" and result["score"] > 0.8:
            return False, "内容可能不当"
```

```
        return True, ""

    def _keyword_filter(self, text):
        """
        关键词过滤
        """
        # 敏感词库
        sensitive_words = ["暴力", "色情", ...]  # 实际应从配置加载

        for word in sensitive_words:
            if word in text:
                return True

        return False
```

16.2.5 扩展性与优化

```
class SystemOptimization:
    """
    系统优化方案
    """
    def __init__(self):
        self.optimizations = {}

    def latency_optimization(self):
        """
        延迟优化
        """
        strategies = {
            "缓存": {
                "热点会话缓存": "Redis缓存频繁访问的会话",
                "模型输出缓存": "缓存常见问题的回答",
                "CDN": "静态资源CDN加速"
            },

            "模型优化": {
                "Quantization": "INT8量化降低延迟",
                "Speculative Decoding": "投机采样加速生成",
                "Flash Attention": "优化Attention计算"
            },

            "架构优化": {
                "流式输出": "边生成边返回",
                "异步处理": "异步IO减少阻塞",
                "连接池": "复用数据库连接"
            }
        }
```

```
        return strategies

    def scalability_design(self):
        """
        扩展性设计
        """
        design = {
            "水平扩展": {
                "无状态服务": "所有服务设计为无状态，易于扩展",
                "分片": {
                    "会话分片": "按session_id分片到不同节点",
                    "用户分片": "按user_id分片数据库"
                },
                "自动伸缩": "基于CPU/QPS自动扩缩容"
            },

            "垂直扩展": {
                "GPU升级": "使用更强GPU（A100→H100）",
                "内存扩容": "增加缓存容量"
            },

            "读写分离": {
                "主从复制": "MongoDB主从，读写分离",
                "CQRS": "命令查询职责分离"
            }
        }

        return design

    def reliability_design(self):
        """
        可靠性设计
        """
        design = {
            "高可用": {
                "多副本": "每个服务3+副本",
                "跨AZ部署": "跨可用区部署",
                "故障转移": "自动故障检测和切换"
            },

            "容错": {
                "熔断": "Circuit Breaker防止雪崩",
                "限流": "Rate Limiting保护后端",
                "降级": "模型不可用时降级到简单回复"
            },

            "数据可靠": {
                "备份": "数据库每日备份",
```

```
            "多副本": "Redis/MongoDB多副本",
            "WAL": "Write-Ahead Log保证不丢数据"
        },

        "监控告警": {
            "指标": "QPS、延迟、错误率、GPU利用率",
            "日志": "ELK收集分析日志",
            "追踪": "分布式追踪（Jaeger）",
            "告警": "异常自动告警（PagerDuty）"
        }
    }

    return design
```

16.3 其他经典系统设计题

16.3.1 设计内容审核系统

```
class ContentModerationSystem:
    """
    内容审核系统设计

    需求：
    - 支持文本、图像、视频审核
    - 机器预审 + 人工复审
    - 实时审核 < 100ms
    - 99.9%准确率
    """
    def __init__(self):
        self.ml_models = {
            "text": TextModerationModel(),
            "image": ImageModerationModel(),
            "video": VideoModerationModel()
        }

        self.human_review_queue = Queue()

    async def moderate_content(self, content, content_type):
        """
        审核内容

        流程：
        1. 机器预审（快速）
        2. 置信度低的→人工队列
        3. 返回结果
        """
        # 1. 机器预审
```

```
        ml_result = await self._ml_moderate(content, content_type)

        # 2. 判断是否需要人工
        if ml_result["confidence"] < 0.8:
            # 加入人工审核队列
            await self._queue_for_human_review(content, ml_result)
            return {"status": "pending", "estimated_time": "5min"}

        # 3. 直接返回
        return {
            "status": "completed",
            "result": ml_result["label"],  # safe/unsafe
            "confidence": ml_result["confidence"]
        }

    async def _ml_moderate(self, content, content_type):
        """
        机器学习审核
        """
        model = self.ml_models[content_type]

        # 多模型集成
        results = []

        # 模型1: 规则引擎（快速）
        rule_result = self._rule_based_check(content)
        results.append(rule_result)

        # 模型2: 深度学习模型（准确）
        dl_result = await model.predict(content)
        results.append(dl_result)

        # 模型3: 大模型（理解能力强）
        llm_result = await self._llm_moderate(content)
        results.append(llm_result)

        # 集成结果
        final_result = self._ensemble(results)

        return final_result
```

16.3.2 设计推荐系统

```
class RecommendationSystem:
    """
    大模型驱动的推荐系统

    特点:
```

```
- 召回+排序两阶段
- 使用Embedding做召回
- LLM做精排和解释
"""
def __init__(self):
    self.embedding_model = SentenceTransformer()
    self.ranking_llm = RankingLLM()
    self.vector_db = Milvus()

async def recommend(self, user_id, context, k=10):
    """
    推荐流程
    """
    # 1. 理解用户意图
    user_intent = await self._understand_intent(context)

    # 2. 召回候选（快速，从大量item中筛选）
    candidates = await self._recall(user_id, user_intent, k=100)

    # 3. 精排（准确，LLM排序）
    ranked = await self._rank(user_id, candidates, user_intent, k=k)

    # 4. 生成解释
    explanations = await self._explain(ranked, user_intent)

    return {
        "items": ranked,
        "explanations": explanations
    }

async def _recall(self, user_id, intent, k):
    """
    召回阶段：向量检索
    """
    # 用户兴趣embedding
    user_embedding = await self._get_user_embedding(user_id)

    # 向量召回
    candidates = await self.vector_db.search(
        user_embedding,
        top_k=k
    )

    return candidates

async def _rank(self, user_id, candidates, intent, k):
    """
    排序阶段：LLM精排
    """
```

```
        prompt = f"""
给定用户意图和候选items，请排序推荐最相关的{k}个。

【用户意图】
{intent}

【候选items】
{json.dumps(candidates, ensure_ascii=False)}

请输出排序后的item IDs（JSON数组）：
"""

        ranked_ids = await self.ranking_llm.generate(prompt)

        return [c for c in candidates if c["id"] in ranked_ids]
```

16.4 答题技巧总结

16.4.1 时间分配

阶段	时间	要点
需求澄清	5min	主动提问，确认范围
规模估算	5min	快速估算，数量级正确即可
高层设计	10min	画架构图，说明组件
详细设计	15min	核心流程，数据结构
优化讨论	10min	扩展性、可靠性

16.4.2 常见陷阱

❌ **过早优化**：不要一开始就讲细节优化 ❌ **缺少trade-off**：要说明设计的权衡 ❌ **忽略非功能需求**：延迟、可用性等 ❌ **不和面试官沟通**：一个人闷头想

✅ **正确做法**：

- 边想边说，让面试官跟上思路
- 先整体后局部
- 说明为什么这样设计
- 讨论方案的优缺点

16.5 本章小结

本章介绍了系统设计题的答题方法：

✅ **标准流程**：需求→估算→架构→详细设计→优化 ✅ **对话系统**：完整的设计案例 ✅ **其他系统**：审核、推荐等 ✅ **答题技巧**：时间分配、沟通技巧

关键要点：

- 结构化思考
- 主动沟通
- 权衡取舍
- 工程思维

下一章预告： 第17章场景题与开放题已完成，第18章将介绍公司与岗位分析。