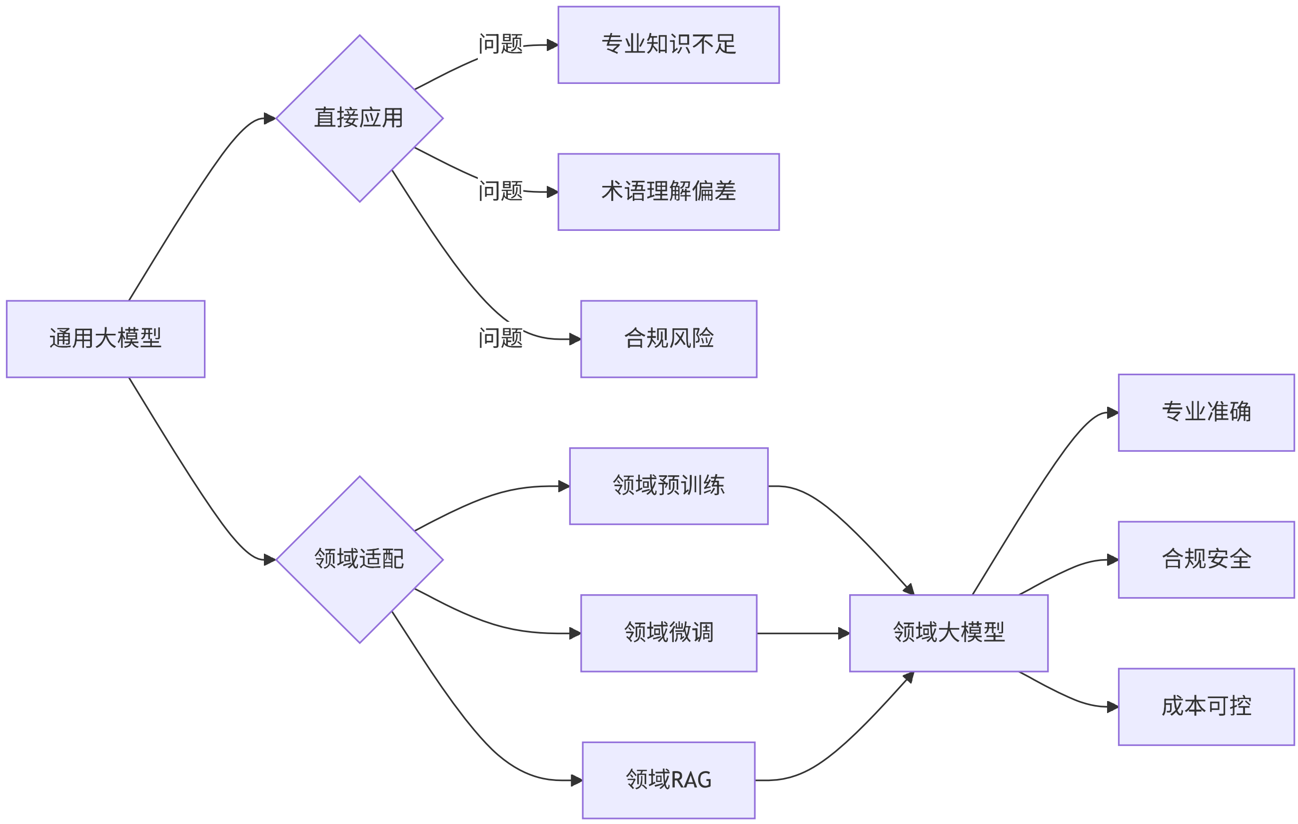
第13章 垂直领域应用

> 通用模型+领域知识=垂直大模型

13.1 领域大模型概述

13.1.1 为什么需要领域模型



通用模型 vs 领域模型对比：

维度	通用模型	领域模型	提升幅度
专业知识	60-70%	85-95%	+25-35%
术语理解	有偏差	精准	显著提升
合规性	需人工审核	内置规则	风险降低
推理成本	高	中等	降低30-50%
响应速度	中等	快	提升2-3倍

13.2 金融领域

13.2.1 金融大模型架构

```
class FinanceLLM:
    """
```

```
    金融领域大模型
    """
    def __init__(self, base_model, domain_knowledge):
        self.base_model = base_model
        self.domain_knowledge = domain_knowledge

        # 领域组件
        self.finance_rag = FinanceRAG()
        self.risk_checker = RiskChecker()
        self.compliance_filter = ComplianceFilter()

    def generate_response(self, query, context=None):
        """
        生成金融领域回答
        """
        # 1. 风险预检
        risk_level = self.risk_checker.assess(query)
        if risk_level == "high":
            return self._generate_safe_response(query)

        # 2. 检索相关知识
        relevant_docs = self.finance_rag.retrieve(query)

        # 3. 增强prompt
        enhanced_prompt = self._build_finance_prompt(
            query, relevant_docs, context
        )

        # 4. 生成回答
        response = self.base_model.generate(enhanced_prompt)

        # 5. 合规过滤
        filtered_response = self.compliance_filter.filter(response)

        # 6. 添加免责声明
        final_response = self._add_disclaimer(filtered_response)

        return final_response

    def _build_finance_prompt(self, query, docs, context):
        """
        构建金融领域prompt
        """
        prompt = f"""
你是一个专业的金融顾问助手。请基于以下信息回答问题。

【参考资料】
{docs}
```

```
【问题】
{query}

【回答要求】
1. 准确引用监管政策和法规
2. 使用专业金融术语
3. 提供数据支撑（如有）
4. 说明风险因素
5. 保持中立客观

【回答】
"""
        return prompt

    def _add_disclaimer(self, response):
        """
        添加免责声明
        """
        disclaimer = "\n\n【免责声明】以上内容仅供参考，不构成投资建议。投资有风险，入市需谨慎
        return response + disclaimer
```

13.2.2 金融知识库构建

```python
class FinanceKnowledgeBase:
    """
    金融知识库
    """
    def __init__(self):
        self.sources = {
            "regulations": [],    # 监管文件
            "policies": [],       # 政策文件
            "reports": [],        # 研究报告
            "news": [],           # 财经新闻
            "company_data": [],   # 公司数据
        }

    def build_knowledge_base(self):
        """
        构建知识库
        """
        # 1. 收集监管文件
        self._collect_regulations()

        # 2. 收集研报
        self._collect_research_reports()

        # 3. 收集实时数据
        self._collect_market_data()
```

```python
        # 4. 处理和向量化
        self._process_and_embed()

    def _collect_regulations(self):
        """
        收集监管文件

        数据源:
        - 中国证监会公告
        - 银保监会文件
        - 央行政策
        - 交易所规则
        """
        sources = [
            "http://www.csrc.gov.cn/",  # 证监会
            "http://www.cbirc.gov.cn/", # 银保监会
            "http://www.pbc.gov.cn/",   # 央行
        ]

        for source in sources:
            documents = self._crawl_documents(source)
            self.sources["regulations"].extend(documents)

    def _process_and_embed(self):
        """
        处理文档并向量化
        """
        from langchain.text_splitter import RecursiveCharacterTextSplitter
        from langchain.embeddings import HuggingFaceEmbeddings

        # 分块
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=500,
            chunk_overlap=50,
            separators=["\n\n", "\n", "。", "；"]
        )

        all_docs = []
        for source_type, documents in self.sources.items():
            for doc in documents:
                chunks = text_splitter.split_text(doc["content"])

                for chunk in chunks:
                    all_docs.append({
                        "text": chunk,
                        "source": doc["source"],
                        "type": source_type,
                        "date": doc["date"]
```

```
                })

        # 向量化
        embeddings = HuggingFaceEmbeddings(
            model_name="BAAI/bge-large-zh-v1.5"  # 中文金融embedding
        )

        # 存入向量数据库
        from langchain.vectorstores import Milvus

        self.vector_store = Milvus.from_documents(
            all_docs,
            embeddings,
            collection_name="finance_knowledge"
        )
```

13.2.3 金融应用场景

场景1: 智能投研助手

```
class InvestmentResearchAssistant:
    """
    智能投研助手
    """
    def analyze_company(self, company_name, stock_code):
        """
        分析上市公司
        """
        # 1. 收集公司数据
        company_data = self._get_company_data(stock_code)

        # 2. 财务分析
        financial_analysis = self._analyze_financials(company_data)

        # 3. 行业对比
        industry_comparison = self._compare_with_industry(company_data)

        # 4. 风险评估
        risks = self._assess_risks(company_data)

        # 5. 生成研报
        report = self._generate_report(
            company_name,
            financial_analysis,
            industry_comparison,
            risks
        )
```

```python
        return report

    def _analyze_financials(self, data):
        """
        财务分析
        """
        analysis = {
            "盈利能力": {
                "ROE": data["roe"],
                "ROA": data["roa"],
                "净利率": data["net_margin"],
                "评价": self._evaluate_profitability(data)
            },
            "偿债能力": {
                "资产负债率": data["debt_ratio"],
                "流动比率": data["current_ratio"],
                "评价": self._evaluate_solvency(data)
            },
            "成长能力": {
                "营收增长率": data["revenue_growth"],
                "利润增长率": data["profit_growth"],
                "评价": self._evaluate_growth(data)
            }
        }

        return analysis

    def _generate_report(self, company, financials, industry, risks):
        """
        使用LLM生成研报
        """
        prompt = f"""
请基于以下数据，生成一份专业的投资研究报告：

【公司基本信息】
公司名称：{company}

【财务分析】
{json.dumps(financials, ensure_ascii=False, indent=2)}

【行业对比】
{json.dumps(industry, ensure_ascii=False, indent=2)}

【风险因素】
{json.dumps(risks, ensure_ascii=False, indent=2)}

请生成包含以下部分的研报：
1. 执行摘要
2. 公司概况
```

```
3. 财务分析
4. 行业地位
5. 投资亮点
6. 风险提示
7. 投资建议

要求：
- 数据驱动
- 逻辑清晰
- 结论明确
- 风险提示充分
"""

        report = self.llm.generate(prompt)
        return report
```

场景2: 智能风控

```python
class IntelligentRiskControl:
    """
    智能风控系统
    """
    def __init__(self):
        self.risk_models = {
            "credit_risk": CreditRiskModel(),
            "market_risk": MarketRiskModel(),
            "fraud_detection": FraudDetectionModel(),
        }

    def assess_loan_application(self, application):
        """
        评估贷款申请
        """
        # 1. 基础信息检查
        basic_check = self._basic_validation(application)
        if not basic_check["passed"]:
            return {"approved": False, "reason": basic_check["reason"]}

        # 2. 信用评分
        credit_score = self.risk_models["credit_risk"].score(application)

        # 3. 反欺诈检测
        fraud_score = self.risk_models["fraud_detection"].detect(application)

        # 4. 综合评估
        decision = self._make_decision(credit_score, fraud_score, application)

        # 5. 生成解释
```

```python
        explanation = self._generate_explanation(decision, application)

        return {
            "approved": decision["approved"],
            "credit_score": credit_score,
            "fraud_score": fraud_score,
            "loan_amount": decision["loan_amount"],
            "interest_rate": decision["interest_rate"],
            "explanation": explanation
        }

    def _generate_explanation(self, decision, application):
        """
        生成可解释的决策理由
        """
        prompt = f"""
基于以下贷款评估结果，生成一份客户可理解的决策说明：

【评估结果】
- 是否批准：{"是" if decision["approved"] else "否"}
- 信用评分：{decision["credit_score"]}/100
- 批准额度：{decision.get("loan_amount", "不适用")}

【客户信息】
- 月收入：{application["income"]}
- 工作年限：{application["work_years"]}
- 信用历史：{application["credit_history"]}

请生成：
1. 决策结果说明（用通俗语言）
2. 主要考虑因素
3. 如何改善信用（如被拒）

要求：
- 语言友好
- 逻辑清晰
- 具有建设性
"""

        explanation = self.llm.generate(prompt)
        return explanation
```

13.3 医疗健康领域

13.3.1 医疗大模型特点

```python
class MedicalLLM:
    """
    医疗领域大模型
    """
    def __init__(self):
        self.base_model = load_medical_model()

        # 医疗知识库
        self.knowledge_bases = {
            "guidelines": MedicalGuidelinesDB(),  # 诊疗指南
            "drugs": DrugDatabase(),              # 药品库
            "diseases": DiseaseDatabase(),        # 疾病库
            "literature": MedicalLiteratureDB()   # 医学文献
        }

        # 安全组件
        self.safety_checker = MedicalSafetyChecker()
        self.disclaimer_generator = DisclaimerGenerator()

    def diagnose_assistant(self, symptoms, patient_info):
        """
        辅助诊断（注意：不能替代医生）
        """
        # 1. 安全检查
        if self.safety_checker.is_emergency(symptoms):
            return {
                "urgency": "紧急",
                "recommendation": "请立即就医或拨打120！",
                "symptoms": symptoms
            }

        # 2. 检索相关疾病
        possible_diseases = self._match_diseases(symptoms)

        # 3. 生成建议
        advice = self._generate_medical_advice(
            symptoms,
            possible_diseases,
            patient_info
        )

        # 4. 添加免责声明
        advice["disclaimer"] = self.disclaimer_generator.get_medical_disclaimer()

        return advice

    def _generate_medical_advice(self, symptoms, diseases, patient_info):
        """
```

```
        生成医疗建议
        """
        prompt = f"""
你是一个医疗知识助手。基于以下信息提供建议。

【患者信息】
- 年龄：{patient_info.get('age')}
- 性别：{patient_info.get('gender')}
- 既往病史：{patient_info.get('medical_history', '无')}

【症状】
{symptoms}

【可能的疾病】
{diseases}

请提供：
1. 症状分析
2. 可能的原因
3. 建议的检查项目
4. 生活建议
5. 就医建议（科室）

重要提示：
- 这不是诊断，仅供参考
- 严重症状请立即就医
- 不要自行用药
"""

        response = self.base_model.generate(prompt)

        return {
            "analysis": response,
            "urgency_level": self._assess_urgency(symptoms),
            "recommended_department": self._recommend_department(diseases)
        }
```

13.3.2 医疗知识图谱

```
class MedicalKnowledgeGraph:
    """
    医疗知识图谱
    """
    def __init__(self):
        self.graph = nx.DiGraph()

    def build_graph(self):
        """
```

```python
        构建医疗知识图谱

        实体类型：
        - 疾病（Disease）
        - 症状（Symptom）
        - 药物（Drug）
        - 检查（Examination）
        - 科室（Department）

        关系类型：
        - has_symptom: 疾病-症状
        - treated_by: 疾病-药物
        - diagnosed_by: 疾病-检查
        - belongs_to: 疾病-科室
        """
        # 添加实体
        self._add_entities()

        # 添加关系
        self._add_relations()

    def _add_entities(self):
        """
        添加实体
        """
        # 疾病
        diseases = [
            {"id": "D001", "name": "感冒", "type": "Disease"},
            {"id": "D002", "name": "流感", "type": "Disease"},
            # ...
        ]

        # 症状
        symptoms = [
            {"id": "S001", "name": "发热", "type": "Symptom"},
            {"id": "S002", "name": "咳嗽", "type": "Symptom"},
            # ...
        ]

        # 添加到图中
        for disease in diseases:
            self.graph.add_node(
                disease["id"],
                name=disease["name"],
                type="Disease"
            )

        for symptom in symptoms:
            self.graph.add_node(
```

```python
                symptom["id"],
                name=symptom["name"],
                type="Symptom"
            )

    def _add_relations(self):
        """
        添加关系
        """
        relations = [
            ("D001", "S001", "has_symptom"),  # 感冒-发热
            ("D001", "S002", "has_symptom"),  # 感冒-咳嗽
            # ...
        ]

        for source, target, rel_type in relations:
            self.graph.add_edge(source, target, relation=rel_type)

    def query_diseases_by_symptoms(self, symptoms):
        """
        根据症状查询可能的疾病
        """
        symptom_ids = [self._get_symptom_id(s) for s in symptoms]

        # 找到所有关联的疾病
        disease_scores = {}

        for symptom_id in symptom_ids:
            # 找到有该症状的疾病
            for disease_id in self.graph.predecessors(symptom_id):
                if self.graph.nodes[disease_id]["type"] == "Disease":
                    disease_scores[disease_id] = disease_scores.get(disease_id, 0) + 1

        # 按匹配度排序
        sorted_diseases = sorted(
            disease_scores.items(),
            key=lambda x: x[1],
            reverse=True
        )

        return [
            {
                "disease": self.graph.nodes[d_id]["name"],
                "match_score": score / len(symptoms),
                "symptoms": self._get_disease_symptoms(d_id)
            }
            for d_id, score in sorted_diseases[:5]
        ]
```

13.3.3 电子病历理解

```python
def extract_medical_entities(medical_record):
    """
    从电子病历中提取实体
    """
    from transformers import pipeline

    # 使用医疗NER模型
    ner_model = pipeline(
        "ner",
        model="emilyalsentzer/Bio_ClinicalBERT"
    )

    entities = ner_model(medical_record)

    # 分类整理
    extracted = {
        "symptoms": [],
        "diseases": [],
        "drugs": [],
        "examinations": []
    }

    for entity in entities:
        entity_type = entity["entity"]
        text = entity["word"]

        if "SYMPTOM" in entity_type:
            extracted["symptoms"].append(text)
        elif "DISEASE" in entity_type:
            extracted["diseases"].append(text)
        elif "DRUG" in entity_type:
            extracted["drugs"].append(text)
        # ...

    return extracted
```

13.4 法律领域

13.4.1 法律大模型

```python
class LegalLLM:
    """
    法律领域大模型
    """
    def __init__(self):
```

```python
        self.base_model = load_legal_model()

        # 法律知识库
        self.law_database = LawDatabase()      # 法律法规
        self.case_database = CaseDatabase()    # 判例库
        self.precedent_retriever = PrecedentRetriever()

    def legal_consultation(self, question, case_details=None):
        """
        法律咨询
        """
        # 1. 识别法律问题类型
        question_type = self._classify_legal_question(question)

        # 2. 检索相关法律条文
        relevant_laws = self.law_database.search(question)

        # 3. 检索相似案例
        similar_cases = self.case_database.find_similar(case_details)

        # 4. 生成法律意见
        legal_opinion = self._generate_legal_opinion(
            question,
            question_type,
            relevant_laws,
            similar_cases
        )

        # 5. 添加法律免责
        legal_opinion["disclaimer"] = "本意见仅供参考，不构成正式法律意见。具体问题请咨询执业

        return legal_opinion

    def _generate_legal_opinion(self, question, q_type, laws, cases):
        """
        生成法律意见
        """
        prompt = f"""
你是一个专业的法律助手。请基于法律法规和判例，回答以下法律问题。

【问题类型】
{q_type}

【问题】
{question}

【相关法律条文】
{self._format_laws(laws)}
```

```
【相似案例】
{self._format_cases(cases)}

请提供:
1. 法律分析
2. 适用的法律条文
3. 类似案例的判决要点
4. 可能的法律后果
5. 建议的处理方式

要求:
- 准确引用法律条文
- 逻辑严密
- 风险提示清晰
"""

        response = self.base_model.generate(prompt)

        return {
            "analysis": response,
            "applicable_laws": [law["name"] for law in laws],
            "similar_cases": [case["id"] for case in cases],
            "risk_level": self._assess_legal_risk(question, laws)
        }

    def contract_review(self, contract_text):
        """
        合同审查
        """
        # 1. 提取关键条款
        key_clauses = self._extract_clauses(contract_text)

        # 2. 风险识别
        risks = self._identify_risks(key_clauses)

        # 3. 生成审查报告
        report = self._generate_review_report(
            contract_text,
            key_clauses,
            risks
        )

        return report

    def _identify_risks(self, clauses):
        """
        识别合同风险
        """
        risks = []
```

```python
        risk_patterns = {
            "权利义务不对等": ["单方", "甲方有权", "乙方应当"],
            "违约责任不明确": ["违约", "未明确金额", "未明确期限"],
            "争议解决条款缺失": ["争议", "仲裁", "管辖"],
            "关键条款缺失": ["标的", "价款", "履行期限"]
        }

        for risk_type, keywords in risk_patterns.items():
            # 检查是否存在风险
            # ...
            pass

        return risks
```

13.5 领域模型训练流程

13.5.1 领域数据准备

```python
class DomainDataPreparation:
    """
    领域数据准备
    """
    def __init__(self, domain):
        self.domain = domain

    def prepare_domain_corpus(self):
        """
        准备领域语料
        """
        corpus = []

        # 1. 公开数据源
        corpus.extend(self._collect_public_data())

        # 2. 专业文献
        corpus.extend(self._collect_literature())

        # 3. 行业报告
        corpus.extend(self._collect_reports())

        # 4. 质量过滤
        corpus = self._filter_quality(corpus)

        # 5. 去重
        corpus = self._deduplicate(corpus)
```

```python
        # 6. 格式化
        formatted = self._format_for_training(corpus)

        return formatted

    def create_instruction_dataset(self):
        """
        创建指令数据集

        格式:
        {
            "instruction": "问题或任务描述",
            "input": "输入（可选）",
            "output": "期望的输出"
        }
        """
        dataset = []

        # 方法1: 从FAQ转换
        faqs = self._collect_domain_faqs()
        for faq in faqs:
            dataset.append({
                "instruction": faq["question"],
                "input": "",
                "output": faq["answer"]
            })

        # 方法2: 从文档生成
        documents = self._collect_domain_docs()
        for doc in documents:
            qa_pairs = self._generate_qa_from_doc(doc)
            dataset.extend(qa_pairs)

        # 方法3: 专家标注
        expert_data = self._collect_expert_annotations()
        dataset.extend(expert_data)

        return dataset

    def _generate_qa_from_doc(self, document):
        """
        从文档生成问答对

        使用GPT-4生成高质量的问答对
        """
        prompt = f"""
基于以下{self.domain}领域文档，生成5个高质量的问答对。

【文档】
```

```
{document}

要求：
1. 问题要具体、实用
2. 答案要准确、专业
3. 覆盖文档的关键信息

输出格式（JSON）：
[
    {{"question": "...", "answer": "..."}},
    ...
]
"""

        qa_pairs = self.llm.generate(prompt)
        return json.loads(qa_pairs)
```

13.5.2 领域模型微调

```python
def fine_tune_domain_model(base_model, domain_dataset, domain_name):
    """
    领域模型微调
    """
    from transformers import (
        AutoModelForCausalLM,
        AutoTokenizer,
        TrainingArguments,
        Trainer
    )
    from peft import get_peft_model, LoraConfig

    # 1. 加载基础模型
    model = AutoModelForCausalLM.from_pretrained(base_model)
    tokenizer = AutoTokenizer.from_pretrained(base_model)

    # 2. LoRA配置
    lora_config = LoraConfig(
        r=16,
        lora_alpha=32,
        target_modules=["q_proj", "v_proj"],
        lora_dropout=0.05,
        bias="none",
        task_type="CAUSAL_LM"
    )

    model = get_peft_model(model, lora_config)

    # 3. 训练配置
```

```python
    training_args = TrainingArguments(
        output_dir=f"./output/{domain_name}-model",
        num_train_epochs=3,
        per_device_train_batch_size=4,
        gradient_accumulation_steps=4,
        learning_rate=2e-5,
        fp16=True,
        logging_steps=10,
        save_strategy="epoch",
        # 领域特定配置
        warmup_ratio=0.1,
        weight_decay=0.01,
    )

    # 4. 训练
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=domain_dataset["train"],
        eval_dataset=domain_dataset["eval"],
        tokenizer=tokenizer,
    )

    trainer.train()

    # 5. 保存
    model.save_pretrained(f"./models/{domain_name}-lora")

    return model
```

13.6 本章小结

本章介绍了垂直领域大模型的构建和应用：

✅ **金融领域**：投研助手、智能风控 ✅ **医疗领域**：辅助诊断、病历理解 ✅ **法律领域**：法律咨询、合同审查 ✅ **领域适配**：数据准备、模型微调

关键要点：

- 领域知识库是核心
- 安全合规放在首位
- 人机协作而非替代
- 持续迭代优化

下一章预告： 第14章将介绍多模态大模型。