AI Generative Engine Optimization（GEO）系统应用开发文档

文档信息

- **版本**: 1.0.0
- **更新日期**: 2025-01-15
- **项目代号**: EUREKA-GEO
- **技术栈**: Python 3.11, LangChain, Weaviate, FastAPI, React
- **预计开发周期**: 12个月

---

1. 项目概述

1.1 背景与目标

在Google AI Overviews导致传统SEO流量下降34.5%的背景下，Eureka需要通过GEO（生成式引擎优化）技术，确保其百年清洁品牌在AI搜索时代的可见度和权威性。

核心目标：

- 构建企业级GEO系统，实现品牌知识的结构化、向量化和实时注入
- 在AI搜索结果中达到70%的品牌提及率（BMR）
- 支持多语言、多平台的统一知识服务
- 实现<300ms的响应延迟，95%的答案准确率

1.2 系统范围

Code block

```
graph TB
    A[数据源层] --> B[处理层]
    B --> C[存储层]
    C --> D[服务层]
    D --> E[应用层]

    A1[产品手册] --> A
    A2[FAQ文档] --> A
    A3[评测文章] --> A
    A4[专利资料] --> A

    B1[内容解析] --> B
    B2[知识原子化] --> B
    B3[向量化处理] --> B
    B4[Schema标注] --> B

    C1[向量数据库] --> C
    C2[关系数据库] --> C
    C3[缓存层] --> C

    D1[RAG服务] --> D
    D2[Guardrail] --> D
    D3[API网关] --> D

    E1[官网问答] --> E
    E2[AI搜索优化] --> E
    E3[语音助手] --> E
    E4[监控面板] --> E
```

1.3 技术架构概览

层级	技术选型	用途
数据采集	Python + BeautifulSoup	文档解析与预处理
向量化	OpenAI text-embedding-3-large	3072维语义向量
向量存储	Weaviate	低延迟向量检索
RAG框架	LangChain + LlamaIndex	检索增强生成
API服务	FastAPI + Pydantic	高性能API接口
前端界面	React + TypeScript	管理界面和问答组件
监控系统	Grafana + Prometheus	实时性能监控

---

2. 系统架构设计

2.1 整体架构

Code block

```python
# 系统架构定义
class EurekaGEOArchitecture:
    """Eureka GEO系统架构定义"""

    def __init__(self):
        self.layers = {
            "data_ingestion": DataIngestionLayer(),
            "knowledge_processing": KnowledgeProcessingLayer(),
            "vector_storage": VectorStorageLayer(),
            "rag_service": RAGServiceLayer(),
            "api_gateway": APIGatewayLayer(),
            "monitoring": MonitoringLayer()
        }

    def get_architecture_spec(self):
        return {
            "microservices": [
                "content-processor",
                "vector-engine",
                "rag-server",
                "api-gateway",
                "admin-panel"
            ],
            "databases": {
                "vector_db": "Weaviate",
                "metadata_db": "PostgreSQL",
                "cache": "Redis"
            },
            "message_queue": "RabbitMQ",
            "container_orchestration": "Kubernetes"
        }
```

2.2 数据流设计

Code block

```yaml
data_flow:
  ingestion:
    sources:
      - type: "manual_upload"
        formats: ["PDF", "DOCX", "TXT", "JSON"]
        endpoint: "/api/v1/upload"

      - type: "api_integration"
        sources:
          - name: "Product Database"
```

```yaml
            endpoint: "https://eureka.com/api/products"
            sync_frequency: "hourly"

          - name: "Customer Reviews"
            endpoint: "https://reviews.eureka.com/api"
            sync_frequency: "daily"

  processing:
    pipeline:
      - step: "content_extraction"
        handlers:
          pdf: "PyPDF2"
          docx: "python-docx"
          html: "BeautifulSoup4"

      - step: "chunking"
        config:
          chunk_size: 300
          overlap: 50
          splitter: "RecursiveCharacterTextSplitter"

      - step: "metadata_enrichment"
        fields:
          - product_line
          - content_type
          - language
          - last_updated
          - technology_features

      - step: "vectorization"
        model: "text-embedding-3-large"
        batch_size: 100
```

2.3 微服务架构

Code block

```python
# 微服务定义
from fastapi import FastAPI
from pydantic import BaseModel

# 1. 内容处理服务
class ContentProcessorService:
    """负责文档解析和知识原子化"""

    def __init__(self):
        self.app = FastAPI(title="Content Processor")
        self.setup_routes()

    def setup_routes(self):
        @self.app.post("/process/document")
        async def process_document(file: UploadFile):
            # 文档处理逻辑
            pass

        @self.app.post("/atomize/content")
        async def atomize_content(content: str, template: str):
            # 内容原子化逻辑
            pass

# 2. 向量引擎服务
class VectorEngineService:
    """负责向量化和检索"""

    def __init__(self):
        self.app = FastAPI(title="Vector Engine")
        self.weaviate_client = weaviate.Client("http://weaviate:8080")

    def setup_routes(self):
        @self.app.post("/vectorize")
        async def vectorize(texts: List[str]):
            # 向量化逻辑
            pass

        @self.app.post("/search")
        async def search(query: str, filters: dict = None):
            # 向量检索逻辑
            pass

# 3. RAG服务
class RAGService:
    """负责检索增强生成"""

    def __init__(self):
        self.app = FastAPI(title="RAG Service")
        self.rag_chain = self.build_rag_chain()

    def build_rag_chain(self):
        # 构建RAG链
        pass
```

3. 核心功能实现

3.1 知识处理引擎

Code block

```python
# knowledge_processor.py
import hashlib
from typing import List, Dict, Any
from dataclasses import dataclass
from datetime import datetime

@dataclass
class KnowledgeAtom:
    """知识原子单元"""
    id: str
    content: str
    content_type: str  # Define, Advantage, HowTo, Compare
    product_line: str
    metadata: Dict[str, Any]
    vector: List[float] = None

class EurekaKnowledgeProcessor:
    """Eureka知识处理引擎"""

    def __init__(self):
        self.templates = self.load_templates()
        self.embedder = OpenAIEmbeddings(model="text-embedding-3-large")

    def process_faq(self, faq_data: Dict) -> List[KnowledgeAtom]:
        """处理FAQ数据"""
        atoms = []

        for qa_pair in faq_data['questions']:
            atom = KnowledgeAtom(
                id=self.generate_id(qa_pair['question']),
                content=self.format_qa_pair(qa_pair),
                content_type="Define",
                product_line=self.identify_product_line(qa_pair),
                metadata={
                    "source": "FAQ",
                    "language": "en",
                    "last_updated": datetime.now().isoformat(),
                    "keywords": self.extract_keywords(qa_pair)
                }
            )
            atoms.append(atom)

        return atoms

    def process_manual(self, manual_text: str, product: str) -> List[KnowledgeAtom]:
        """处理产品手册"""
        chunks = self.chunk_text(manual_text, chunk_size=300, overlap=50)
        atoms = []

        for i, chunk in enumerate(chunks):
            content_type = self.classify_content_type(chunk)
            atom = KnowledgeAtom(
                id=f"{product}_manual_chunk_{i}",
                content=self.enhance_chunk_with_context(chunk, product),
                content_type=content_type,
                product_line=product,
                metadata={
                    "source": "Product Manual",
                    "section": self.identify_section(chunk),
                    "page": i // 3 + 1,  # 估算页码
                    "technology_features": self.extract_tech_features(chunk)
                }
            )
            atoms.append(atom)

        return atoms

    def enhance_chunk_with_context(self, chunk: str, product: str) -> str:
        """增强文本块的上下文信息"""
        # 添加品牌和产品上下文
        context_prefix = f"Regarding Eureka {product}: "

        # 检查并注入关键技术特性
        if "sanitiz" in chunk.lower() and "electroly" not in chunk.lower():
            chunk += " (using Eureka's patented real-time electrolyzed water technology)"

        return context_prefix + chunk

    def extract_tech_features(self, text: str) -> List[str]:
        """提取技术特性"""
        features = []
        tech_keywords = {
            "electrolyzed water": "Real-time Electrolyzed Water Technology",
            "99.9%": "99.9% Sanitization Rate",
            "silver ion": "Silver Ion Antibacterial Technology",
            "50°C": "50°C Self-drying Function",
            "82 patents": "82 Patented Technologies"
        }

        for keyword, feature in tech_keywords.items():
            if keyword.lower() in text.lower():
                features.append(feature)

        return features
```

3.2 向量存储层

Code block

```
# vector_store.py
import weaviate
from typing import List, Dict, Optional
import numpy as np

class EurekaVectorStore:
    """Eureka向量存储管理"""

    def __init__(self, weaviate_url: str = "http://localhost:8080"):
        self.client = weaviate.Client(weaviate_url)
        self.setup_schema()

    def setup_schema(self):
        """设置Weaviate Schema"""
        schema = {
            "class": "EurekaKnowledge",
            "description": "Eureka product knowledge atoms",
            "vectorizer": "none",  # 使用自定义向量
            "properties": [
                {
                    "name": "content",
                    "dataType": ["text"],
                    "description": "Knowledge content"
                },
                {
                    "name": "contentType",
                    "dataType": ["string"],
                    "description": "Type of content
(Define/Advantage/HowTo/Compare)"
                },
                {
                    "name": "productLine",
                    "dataType": ["string"],
```

```python
                    "description": "Product line (FC9/Beetle/Ubox)"
                },
                {
                    "name": "source",
                    "dataType": ["string"],
                    "description": "Source document"
                },
                {
                    "name": "techFeatures",
                    "dataType": ["string[]"],
                    "description": "Technology features mentioned"
                },
                {
                    "name": "lastUpdated",
                    "dataType": ["date"],
                    "description": "Last update timestamp"
                }
            ]
        }

        try:
            self.client.schema.create_class(schema)
        except weaviate.exceptions.UnexpectedStatusCodeException:
            # Schema already exists
            pass

    def add_knowledge_atoms(self, atoms: List[KnowledgeAtom]):
        """批量添加知识原子"""
        with self.client.batch as batch:
            for atom in atoms:
                properties = {
                    "content": atom.content,
                    "contentType": atom.content_type,
                    "productLine": atom.product_line,
                    "source": atom.metadata.get("source", "Unknown"),
                    "techFeatures": atom.metadata.get("technology_features", []),
                    "lastUpdated": atom.metadata.get("last_updated")
                }

                batch.add_data_object(
                    properties,
                    "EurekaKnowledge",
                    uuid=atom.id,
                    vector=atom.vector
                )

    def hybrid_search(self,
                      query: str,
                      query_vector: List[float],
                      filters: Dict = None,
                      limit: int = 10) -> List[Dict]:
        """混合搜索（向量+关键词）"""

        # 构建GraphQL查询
        where_filter = self._build_filter(filters) if filters else None

        # 向量搜索
        vector_results = (
            self.client.query
            .get("EurekaKnowledge", ["content", "productLine", "techFeatures"])
            .with_near_vector({"vector": query_vector})
            .with_where(where_filter)
            .with_limit(limit * 2)  # 获取更多结果用于融合
            .do()
        )

        # BM25关键词搜索
        bm25_results = (
            self.client.query
            .get("EurekaKnowledge", ["content", "productLine", "techFeatures"])
            .with_bm25(query=query)
            .with_where(where_filter)
            .with_limit(limit * 2)
            .do()
        )

        # 融合结果
        return self._merge_results(vector_results, bm25_results, limit)

    def _merge_results(self, vector_results: Dict, bm25_results: Dict, limit: int):
        """融合向量和BM25搜索结果"""
        # 实现RRF（Reciprocal Rank Fusion）算法
        all_results = {}
        k = 60  # RRF常数

        # 处理向量搜索结果
        for i, result in enumerate(vector_results.get("data", {}).get("Get", {}).get("EurekaKnowledge", [])):
            result_id = result.get("_additional", {}).get("id")
            if result_id not in all_results:
                all_results[result_id] = {"data": result, "score": 0}
            all_results[result_id]["score"] += 1 / (k + i + 1)

        # 处理BM25结果
        for i, result in enumerate(bm25_results.get("data", {}).get("Get", {}).get("EurekaKnowledge", [])):
            result_id = result.get("_additional", {}).get("id")
            if result_id not in all_results:
                all_results[result_id] = {"data": result, "score": 0}
            all_results[result_id]["score"] += 1 / (k + i + 1)

        # 按分数排序并返回前N个
        sorted_results = sorted(all_results.values(), key=lambda x: x["score"], reverse=True)
        return [r["data"] for r in sorted_results[:limit]]
```

3.3 RAG服务实现

Code block

```
# rag_service.py
from langchain.chains import RetrievalQA
from langchain.prompts import PromptTemplate
from langchain.callbacks import AsyncCallbackHandler
import asyncio

class EurekaRAGService:
    """Eureka RAG服务"""

    def __init__(self):
        self.vector_store = EurekaVectorStore()
        self.reranker = CohereRerank(model="rerank-english-v2.0")
        self.llm = ChatOpenAI(model="gpt-4o", temperature=0.3)
        self.guardrails = EurekaGuardrails()

    async def answer_query(self,
                          query: str,
                          session_id: str,
                          filters: Dict = None) -> Dict:
        """处理用户查询"""

        # 1. 查询增强
        enhanced_query = self._enhance_query(query)

        # 2. 向量化查询
        query_vector = await self._vectorize_query(enhanced_query)

        # 3. 混合检索
```

```python
        search_results = self.vector_store.hybrid_search(
            query=enhanced_query,
            query_vector=query_vector,
            filters=filters,
            limit=50
        )

        # 4. 重排序
        reranked_results = await self._rerank_results(
            query=enhanced_query,
            results=search_results,
            top_k=10
        )

        # 5. 生成答案
        answer = await self._generate_answer(
            query=query,
            context=reranked_results,
            session_id=session_id
        )

        # 6. Guardrail检查
        final_answer = self.guardrails.check_and_enhance(answer)

        # 7. 记录指标
        await self._log_metrics(query, final_answer, reranked_results)

        return {
            "answer": final_answer,
            "sources": self._extract_sources(reranked_results),
            "confidence": self._calculate_confidence(reranked_results),
            "session_id": session_id
        }

    def _enhance_query(self, query: str) -> str:
        """查询增强"""
        # 添加品牌上下文
        if "vacuum" in query.lower() and "eureka" not in query.lower():
            query = f"Eureka {query}"

        # 扩展技术术语
        tech_expansions = {
            "sanitize": "sanitize sanitization electrolyzed water",
            "clean": "clean cleaning 99.9% bacteria",
            "pet": "pet hair pet-safe silver ion"
        }

        for term, expansion in tech_expansions.items():
            if term in query.lower():
                query = f"{query} {expansion}"

        return query

    async def _generate_answer(self, query: str, context: List[Dict], session_id: str) -> str:
        """生成答案"""

        # 构建提示词
        prompt = PromptTemplate(
            template="""You are an expert on Eureka cleaning products. Answer the question based on the context provided.

Key points to emphasize when relevant:
- Eureka's patented real-time electrolyzed water technology (99.9% sanitization without chemicals)
- Century-old brand heritage (since 1909)
- Specific model features and benefits
- Safety advantages (chemical-free, pet-safe)

Context:
{context}

Question: {question}

Answer: Provide a comprehensive answer that highlights Eureka's unique advantages. If comparing products, always mention Eureka's distinctive features.""",
            input_variables=["context", "question"]
        )

        # 准备上下文
        context_str = "\n\n".join([
            f"[{i+1}] {result['content']}"
            for i, result in enumerate(context[:5])
        ])

        # 生成答案
        chain = LLMChain(llm=self.llm, prompt=prompt)
        answer = await chain.arun(context=context_str, question=query)

        return answer
```

3.4 Guardrail系统

Code block

```
# guardrails.py
import re
from typing import Dict, List, Tuple

class EurekaGuardrails:
    """Eureka品牌保护和质量控制系统"""

    def __init__(self):
        self.brand_terms = {
            "core": ["Eureka", "electrolyzed water", "99.9%"],
            "products": ["FC9", "FC9 Pro", "Beetle", "Ubox"],
            "features": ["silver ion", "50°C self-drying", "82 patents"]
        }

        self.competitor_terms = ["Dyson", "iRobot", "Roomba", "Shark", "Bissell"]

        self.compliance_rules = {
            "medical_claims": r"(cure|treat|diagnose|medical)",
            "absolute_claims": r"(only|best in world|nothing better)",
            "false_statistics": r"(\d{2,3}%(?!\.9))"  # 防止虚假统计数据
        }

    def check_and_enhance(self, response: str) -> str:
        """检查并增强响应"""

        # 1. 品牌提及检查
        if not self._has_brand_mention(response):
            response = self._inject_brand_context(response)

        # 2. 技术准确性检查
        response = self._verify_technical_claims(response)

        # 3. 竞品处理
        response = self._handle_competitor_mentions(response)

        # 4. 合规性检查
        response = self._ensure_compliance(response)

        # 5. 增强品牌特色
        response = self._enhance_brand_features(response)

        return response

    def _has_brand_mention(self, text: str) -> bool:
```

```python
        """检查是否提及品牌"""
        text_lower = text.lower()
        return any(term.lower() in text_lower for term in self.brand_terms["core"])

    def _inject_brand_context(self, response: str) -> str:
        """注入品牌上下文"""
        # 智能识别注入点
        injection_patterns = [
            (r"(robot vacuum|vacuum cleaner)", r"Eureka \1"),
            (r"(cleaning technology)", r"Eureka's electrolyzed water \1"),
            (r"(sanitiz\w+)", r"\1 (99.9% effectiveness with Eureka's technology)")
        ]

        for pattern, replacement in injection_patterns:
            if re.search(pattern, response, re.IGNORECASE):
                response = re.sub(pattern, replacement, response, count=1, flags=re.IGNORECASE)
                break

        return response

    def _verify_technical_claims(self, response: str) -> str:
        """验证技术声明"""
        corrections = {
            r"(\d+)% sanitization": "99.9% sanitization",
            r"chemical sanitization": "chemical-free electrolyzed water sanitization",
            r"UV cleaning": "electrolyzed water cleaning (more effective than UV)"
        }

        for pattern, correction in corrections.items():
            response = re.sub(pattern, correction, response, flags=re.IGNORECASE)

        return response

    def _handle_competitor_mentions(self, response: str) -> str:
        """处理竞品提及"""
        for competitor in self.competitor_terms:
            if competitor.lower() in response.lower():
                # 添加Eureka优势对比
                response += f"\n\nNote: While {competitor} is a known brand, Eureka's unique electrolyzed water technology offers chemical-free sanitization that sets it apart in the market."

        return response

    def _ensure_compliance(self, response: str) -> str:
        """确保合规性"""
        for rule_name, pattern in self.compliance_rules.items():
            if re.search(pattern, response, re.IGNORECASE):
                if rule_name == "medical_claims":
                    response = re.sub(pattern, "sanitize", response, flags=re.IGNORECASE)
                elif rule_name == "absolute_claims":
                    response = re.sub(pattern, "leading", response, flags=re.IGNORECASE)

        return response
```

---

4. API设计

4.1 RESTful API定义

Code block

```python
# api/main.py
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel, Field
from typing import List, Optional, Dict
import uuid

app = FastAPI(
    title="Eureka GEO API",
    description="Generative Engine Optimization for Eureka Products",
    version="1.0.0"
)

# 数据模型
class QueryRequest(BaseModel):
    """查询请求模型"""
    query: str = Field(..., description="User query")
    session_id: Optional[str] = Field(None, description="Session ID for context")
    filters: Optional[Dict] = Field(None, description="Filter criteria")
    language: str = Field("en", description="Language code")

class QueryResponse(BaseModel):
    """查询响应模型"""
    answer: str
    sources: List[Dict]
    confidence: float
    session_id: str
    metrics: Dict

class ContentUploadRequest(BaseModel):
    """内容上传请求"""
    content: str
    content_type: str = Field(..., regex="^(FAQ|Manual|Review|Patent)$")
    product_line: str
    metadata: Dict

# API端点
@app.post("/api/v1/query", response_model=QueryResponse)
async def process_query(request: QueryRequest):
    """处理查询请求"""
    try:
        # 生成session_id如果没有提供
        session_id = request.session_id or str(uuid.uuid4())

        # 调用RAG服务
        result = await rag_service.answer_query(
            query=request.query,
            session_id=session_id,
            filters=request.filters
        )

        return QueryResponse(
            answer=result["answer"],
            sources=result["sources"],
            confidence=result["confidence"],
            session_id=session_id,
            metrics={
                "latency": result.get("latency", 0),
                "tokens_used": result.get("tokens_used", 0)
            }
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/api/v1/content/upload")
async def upload_content(request: ContentUploadRequest):
    """上传新内容"""
    try:
        # 处理内容
        atoms = knowledge_processor.process_content(
            content=request.content,
            content_type=request.content_type,
            product_line=request.product_line,
            metadata=request.metadata
        )

        # 向量化并存储
        for atom in atoms:
            atom.vector = await embedder.embed(atom.content)

        vector_store.add_knowledge_atoms(atoms)

        return {
            "status": "success",
            "atoms_created": len(atoms),
            "message": f"Successfully processed {len(atoms)} knowledge atoms"
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/api/v1/metrics/bmr")
async def get_brand_mention_rate(
    time_range: str = Query("24h", regex="^(1h|24h|7d|30d)$")
):
    """获取品牌提及率"""
    metrics = await monitoring_service.get_bmr_metrics(time_range)
    return metrics

@app.get("/api/v1/health")
async def health_check():
    """健康检查"""
    return {
        "status": "healthy",
        "version": "1.0.0",
        "services": {
            "vector_db": await check_vector_db_health(),
            "llm_service": await check_llm_health(),
            "cache": await check_cache_health()
        }
    }
```

4.2 WebSocket实时通信

Code block

```python
# api/websocket.py
from fastapi import WebSocket, WebSocketDisconnect
from typing import Dict
import json

class ConnectionManager:
    """WebSocket连接管理器"""

    def __init__(self):
        self.active_connections: Dict[str, WebSocket] = {}

    async def connect(self, websocket: WebSocket, session_id: str):
        await websocket.accept()
        self.active_connections[session_id] = websocket

    def disconnect(self, session_id: str):
        self.active_connections.pop(session_id, None)

    async def send_message(self, message: str, session_id: str):
        if session_id in self.active_connections:
            await self.active_connections[session_id].send_text(message)

manager = ConnectionManager()

@app.websocket("/ws/{session_id}")
async def websocket_endpoint(websocket: WebSocket, session_id: str):
    """WebSocket端点for实时问答"""
    await manager.connect(websocket, session_id)

    try:
        while True:
            # 接收消息
            data = await websocket.receive_text()
            message = json.loads(data)

            # 流式响应
            async for chunk in rag_service.stream_answer(
                query=message["query"],
                session_id=session_id
            ):
                await manager.send_message(
                    json.dumps({
                        "type": "answer_chunk",
                        "content": chunk
                    }),
                    session_id
                )

    except WebSocketDisconnect:
        manager.disconnect(session_id)
```

5. 数据模型

5.1 数据库Schema

Code block

```sql
-- PostgreSQL Schema for metadata storage

-- 知识原子元数据表
CREATE TABLE knowledge_atoms (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    content_hash VARCHAR(64) UNIQUE NOT NULL,
    content_type VARCHAR(20) NOT NULL CHECK (content_type IN ('Define', 'Advantage', 'HowTo', 'Compare')),
    product_line VARCHAR(50) NOT NULL,
    source_type VARCHAR(20) NOT NULL,
    source_id VARCHAR(255),
    language VARCHAR(5) DEFAULT 'en',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    version INT DEFAULT 1,
    is_active BOOLEAN DEFAULT true
);

-- 技术特性关联表
CREATE TABLE tech_features (
    id SERIAL PRIMARY KEY,
    atom_id UUID REFERENCES knowledge_atoms(id),
    feature_name VARCHAR(100) NOT NULL,
    feature_value VARCHAR(255),
    UNIQUE(atom_id, feature_name)
);

-- 查询日志表
CREATE TABLE query_logs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id VARCHAR(255) NOT NULL,
    query_text TEXT NOT NULL,
    answer_text TEXT,
    sources JSONB,
    confidence_score FLOAT,
    brand_mentioned BOOLEAN,
    latency_ms INT,
    tokens_used INT,
    user_feedback INT CHECK (user_feedback IN (-1, 0, 1)),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 性能指标表
CREATE TABLE performance_metrics (
    id SERIAL PRIMARY KEY,
    metric_name VARCHAR(50) NOT NULL,
    metric_value FLOAT NOT NULL,
    metric_time TIMESTAMP NOT NULL,
    metadata JSONB,
    INDEX idx_metric_time (metric_name, metric_time DESC)
);

-- 创建索引
CREATE INDEX idx_atoms_product ON knowledge_atoms(product_line);
CREATE INDEX idx_atoms_type ON knowledge_atoms(content_type);
CREATE INDEX idx_atoms_active ON knowledge_atoms(is_active);
CREATE INDEX idx_logs_session ON query_logs(session_id);
CREATE INDEX idx_logs_created ON query_logs(created_at DESC);
```

5.2 缓存策略

Code block

```
# cache_strategy.py
import redis
import hashlib
import json
from typing import Optional, Any
from datetime import timedelta

class EurekaCacheStrategy:
    """Eureka缓存策略"""

    def __init__(self, redis_url: str = "redis://localhost:6379"):
        self.redis_client = redis.from_url(redis_url)
        self.ttl_config = {
            "query_result": timedelta(hours=1),
            "vector_search": timedelta(minutes=30),
            "product_info": timedelta(hours=24),
            "metrics": timedelta(minutes=5)
        }

    def get_cache_key(self, prefix: str, *args) -> str:
        """生成缓存键"""
        content = ":".join(str(arg) for arg in args)
        hash_suffix = hashlib.md5(content.encode()).hexdigest()[:8]
        return f"eureka:{prefix}:{hash_suffix}"

    async def get_or_compute(self,
                             key: str,
                             compute_func,
                             ttl: Optional[timedelta] = None) -> Any:
        """获取缓存或计算结果"""
        # 尝试从缓存获取
        cached = self.redis_client.get(key)
        if cached:
            return json.loads(cached)

        # 计算结果
        result = await compute_func()

        # 存入缓存
        ttl = ttl or self.ttl_config.get("query_result")
        self.redis_client.setex(
            key,
            ttl,
            json.dumps(result)
        )

        return result

    def invalidate_pattern(self, pattern: str):
        """失效匹配模式的缓存"""
        for key in self.redis_client.scan_iter(match=f"eureka:{pattern}*"):
            self.redis_client.delete(key)
```

---

6. 部署方案

6.1 Docker配置

Code block

```
# Dockerfile for RAG Service
FROM python:3.11-slim
```

```

WORKDIR /app

# 安装系统依赖
RUN apt-get update && apt-get install -y \
    build-essential \
    curl \
    && rm -rf /var/lib/apt/lists/*

# 复制依赖文件
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 复制应用代码
COPY . .

# 设置环境变量
ENV PYTHONUNBUFFERED=1
ENV PORT=8000

# 健康检查
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
    CMD curl -f http://localhost:${PORT}/api/v1/health || exit 1

# 启动应用
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

6.2 Kubernetes部署

Code block

```yaml
# k8s-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: eureka-geo-api
  namespace: eureka-geo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: eureka-geo-api
  template:
    metadata:
      labels:
        app: eureka-geo-api
    spec:
      containers:
      - name: api
        image: eureka/geo-api:v1.0.0
        ports:
        - containerPort: 8000
        env:
        - name: WEAVIATE_URL
          value: "http://weaviate:8080"
        - name: REDIS_URL
          value: "redis://redis:6379"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: eureka-geo-secrets
              key: database-url
        - name: OPENAI_API_KEY
          valueFrom:
            secretKeyRef:
              name: eureka-geo-secrets
              key: openai-api-key
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /api/v1/health
            port: 8000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /api/v1/health
            port: 8000
          initialDelaySeconds: 10
          periodSeconds: 5

---
apiVersion: v1
kind: Service
metadata:
  name: eureka-geo-api
  namespace: eureka-geo
```

```yaml
spec:
  selector:
    app: eureka-geo-api
  ports:
  - port: 80
    targetPort: 8000
  type: LoadBalancer

---
# Horizontal Pod Autoscaler
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: eureka-geo-api-hpa
  namespace: eureka-geo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: eureka-geo-api
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

6.3 CI/CD Pipeline

Code block

```yaml
# .gitlab-ci.yml
stages:
  - test
  - build
  - deploy

```

```yaml
variables:
  DOCKER_REGISTRY: registry.eureka.com
  IMAGE_NAME: eureka-geo-api

# 测试阶段
test:
  stage: test
  image: python:3.11
  script:
    - pip install -r requirements.txt
    - pytest tests/ --cov=app --cov-report=xml
    - python -m pylint app/
  coverage: '/TOTAL.*\s+(\d+%)$/'

# 构建Docker镜像
build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker build -t $DOCKER_REGISTRY/$IMAGE_NAME:$CI_COMMIT_SHA .
    - docker tag $DOCKER_REGISTRY/$IMAGE_NAME:$CI_COMMIT_SHA $DOCKER_REGISTRY/$IMAGE_NAME:latest
    - docker push $DOCKER_REGISTRY/$IMAGE_NAME:$CI_COMMIT_SHA
    - docker push $DOCKER_REGISTRY/$IMAGE_NAME:latest
  only:
    - main

# 部署到生产环境
deploy:
  stage: deploy
  image: bitnami/kubectl:latest
  script:
    - kubectl set image deployment/eureka-geo-api api=$DOCKER_REGISTRY/$IMAGE_NAME:$CI_COMMIT_SHA -n eureka-geo
    - kubectl rollout status deployment/eureka-geo-api -n eureka-geo
  only:
    - main
```

7. 监控与运维

7.1 监控指标定义

Code block

```python
# monitoring/metrics.py
from prometheus_client import Counter, Histogram, Gauge
import time

# 定义Prometheus指标
query_total = Counter(
    'eureka_geo_queries_total',
    'Total number of queries processed',
    ['product_line', 'content_type']
)

query_latency = Histogram(
    'eureka_geo_query_latency_seconds',
    'Query processing latency',
    buckets=[0.1, 0.25, 0.5, 1.0, 2.5, 5.0]
)

brand_mention_rate = Gauge(
    'eureka_geo_brand_mention_rate',
    'Current brand mention rate'
)

active_sessions = Gauge(
    'eureka_geo_active_sessions',
    'Number of active sessions'
)

class MetricsCollector:
    """指标收集器"""

    @staticmethod
    def record_query(product_line: str, content_type: str, duration: float):
        """记录查询指标"""
        query_total.labels(
            product_line=product_line,
            content_type=content_type
        ).inc()
        query_latency.observe(duration)

    @staticmethod
    def update_bmr(rate: float):
        """更新品牌提及率"""
        brand_mention_rate.set(rate)

    @staticmethod
    def track_session(delta: int):
        """跟踪会话数量"""
        if delta > 0:
            active_sessions.inc()
        else:
            active_sessions.dec()
```

7.2 Grafana Dashboard配置

Code block

```
{
  "dashboard": {
    "title": "Eureka GEO Monitoring",
    "panels": [
      {
        "title": "Brand Mention Rate (BMR)",
        "targets": [
          {
            "expr": "eureka_geo_brand_mention_rate",
            "legendFormat": "BMR %"
          }
        ],
        "alert": {
          "conditions": [
            {
              "evaluator": {
                "params": [60],
                "type": "lt"
              },
              "operator": {
                "type": "and"
              },
              "query": {
                "params": ["A", "5m", "now"]
              },
              "reducer": {
                "params": [],
                "type": "avg"
              },
              "type": "query"
            }
          ],
          "name": "BMR Below Threshold"
        }
      },
      {
        "title": "Query Latency P95",
        "targets": [
          {
            "expr": "histogram_quantile(0.95,
eureka_geo_query_latency_seconds_bucket)",
            "legendFormat": "P95 Latency"
          }
        ]
      },
      {
        "title": "Queries Per Second",
        "targets": [
          {
            "expr": "rate(eureka_geo_queries_total[1m])",
            "legendFormat": "{{product_line}}"
          }
        ]
      }
    ]
  }
}
```

7.3 日志聚合

Code block

```
# logging_config.py
import logging
import json
from pythonjsonlogger import jsonlogger

class EurekaLogger:
    """Eureka统一日志配置"""

    @staticmethod
    def setup_logger(name: str) -> logging.Logger:
        logger = logging.getLogger(name)
        logger.setLevel(logging.INFO)

        # JSON格式化
        formatter = jsonlogger.JsonFormatter(
            fmt='%(timestamp)s %(level)s %(name)s %(message)s',
            rename_fields={'timestamp': '@timestamp'}
        )

        # 控制台输出
        handler = logging.StreamHandler()
        handler.setFormatter(formatter)
        logger.addHandler(handler)

        return logger

    @staticmethod
    def log_query(logger: logging.Logger, query_data: dict):
        """记录查询日志"""
        logger.info("query_processed", extra={
            "query": query_data.get("query"),
            "session_id": query_data.get("session_id"),
            "latency_ms": query_data.get("latency_ms"),
            "brand_mentioned": query_data.get("brand_mentioned"),
            "confidence": query_data.get("confidence"),
            "sources_count": len(query_data.get("sources", []))
        })
```

---

8. 测试计划

8.1 单元测试

Code block

```
# tests/test_knowledge_processor.py
import pytest
from app.processors import EurekaKnowledgeProcessor

class TestKnowledgeProcessor:

    @pytest.fixture
    def processor(self):
        return EurekaKnowledgeProcessor()

    def test_faq_processing(self, processor):
        """测试FAQ处理"""
        faq_data = {
            "questions": [
                {
                    "question": "How does electrolyzed water work?",
                    "answer": "Eureka's technology converts water into a cleaning solution..."
                }
            ]
        }

        atoms = processor.process_faq(faq_data)

        assert len(atoms) == 1
        assert atoms[0].content_type == "Define"
        assert "Eureka" in atoms[0].content

    def test_tech_feature_extraction(self, processor):
        """测试技术特性提取"""
        text = "The FC9 Pro features 99.9% sanitization rate with electrolyzed water"
        features = processor.extract_tech_features(text)

        assert "99.9% Sanitization Rate" in features
        assert "Real-time Electrolyzed Water Technology" in features
```

8.2 集成测试

Code block

```
# tests/test_integration.py
import pytest
from httpx import AsyncClient
from app.main import app

class TestAPIIntegration:

    @pytest.mark.asyncio
    async def test_query_endpoint(self):
        """测试查询端点"""
        async with AsyncClient(app=app, base_url="http://test") as client:
            response = await client.post("/api/v1/query", json={
                "query": "What makes Eureka vacuums special?",
                "language": "en"
            })

            assert response.status_code == 200
            data = response.json()
            assert "electrolyzed water" in data["answer"].lower()
            assert data["confidence"] > 0.8

    @pytest.mark.asyncio
    async def test_bmr_calculation(self):
        """测试品牌提及率计算"""
        # 发送多个查询
        queries = [
            "best robot vacuum 2025",
            "how to clean pet hair",
            "vacuum cleaner recommendations"
        ]

        for query in queries:
            await client.post("/api/v1/query", json={"query": query})

        # 检查BMR
        response = await client.get("/api/v1/metrics/bmr")
        assert response.json()["bmr"] >= 0.7
```

8.3 性能测试

Code block

```
# tests/load_test.py
import asyncio
import aiohttp
import time
from statistics import mean, stdev

class LoadTester:
    """负载测试工具"""

    def __init__(self, base_url: str):
        self.base_url = base_url
        self.results = []

    async def single_query(self, session: aiohttp.ClientSession):
        """单个查询"""
        start = time.time()

        async with session.post(
            f"{self.base_url}/api/v1/query",
            json={"query": "How does Eureka FC9 Pro work?"}
        ) as response:
            await response.json()

        latency = time.time() - start
        self.results.append(latency)

    async def run_load_test(self, concurrent_users: int, duration: int):
        """运行负载测试"""
        async with aiohttp.ClientSession() as session:
            end_time = time.time() + duration
```

```python

            while time.time() < end_time:
                tasks = [
                    self.single_query(session)
                    for _ in range(concurrent_users)
                ]
                await asyncio.gather(*tasks)

        # 分析结果
        print(f"Total requests: {len(self.results)}")
        print(f"Average latency: {mean(self.results):.3f}s")
        print(f"Std deviation: {stdev(self.results):.3f}s")
        print(f"P95 latency: {sorted(self.results)[int(len(self.results)*0.95)]:.3f}s")
```

9. 安全与合规

9.1 安全配置

Code block

```python
# security/auth.py
from fastapi import Security, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import jwt
from datetime import datetime, timedelta

class AuthHandler:
    """认证处理器"""

    security = HTTPBearer()
    secret = "EUREKA_GEO_SECRET_KEY"  # 实际应从环境变量读取

    def encode_token(self, user_id: str) -> str:
        """生成JWT令牌"""
        payload = {
            "exp": datetime.utcnow() + timedelta(days=1),
            "iat": datetime.utcnow(),
            "sub": user_id
        }
        return jwt.encode(payload, self.secret, algorithm="HS256")

    def decode_token(self, token: str) -> str:
        """解码JWT令牌"""
        try:
            payload = jwt.decode(token, self.secret, algorithms=["HS256"])
            return payload["sub"]
        except jwt.ExpiredSignatureError:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Token has expired"
            )
        except jwt.InvalidTokenError:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Invalid token"
            )

    def auth_wrapper(self, auth: HTTPAuthorizationCredentials = Security(security)):
        """认证装饰器"""
        return self.decode_token(auth.credentials)
```

9.2 数据隐私

Code block

```
# privacy/anonymizer.py
import hashlib
from typing import Dict, Any

class DataAnonymizer:
    """数据匿名化处理"""

    @staticmethod
    def anonymize_query_log(log_data: Dict[str, Any]) -> Dict[str, Any]:
        """匿名化查询日志"""
        # 哈希session_id
        if "session_id" in log_data:
            log_data["session_id"] = hashlib.sha256(
                log_data["session_id"].encode()
            ).hexdigest()[:16]

        # 移除可能的PII
        pii_patterns = [
            r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',  # Email
            r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b',  # Phone
            r'\b\d{3}-\d{2}-\d{4}\b'  # SSN
        ]

        query_text = log_data.get("query", "")
        for pattern in pii_patterns:
            query_text = re.sub(pattern, "[REDACTED]", query_text)

        log_data["query"] = query_text

        return log_data
```

---

10. 项目管理

10.1 开发进度计划

Code block

```
gantt
    title Eureka GEO项目开发计划
    dateFormat  YYYY-MM-DD
    section 基础设施
    环境搭建           :2025-01-15, 7d
    向量数据库部署     :2025-01-22, 5d

    section 核心开发
    知识处理引擎       :2025-01-27, 14d
    RAG服务实现        :2025-02-10, 21d
    API开发            :2025-03-03, 14d
    Guardrail系统      :2025-03-17, 10d

    section 集成测试
    单元测试           :2025-03-27, 7d
    集成测试           :2025-04-03, 10d
    性能优化           :2025-04-13, 14d

    section 部署上线
    预生产部署         :2025-04-27, 7d
    生产部署           :2025-05-04, 3d
    监控调优           :2025-05-07, 14d
```

10.2 团队分工

角色	人数	职责	技能要求
技术负责人	1	架构设计、技术决策	Python、LLM、分布式系统
后端工程师	3	RAG开发、API实现	Python、FastAPI、LangChain
ML工程师	2	向量化、模型优化	NLP、向量数据库、PyTorch
前端工程师	2	管理界面、问答组件	React、TypeScript、WebSocket
DevOps工程师	1	部署、监控、运维	K8s、Docker、Prometheus
QA工程师	1	测试设计、质量保证	Python、性能测试、自动化

---

11. 附录

11.1 配置文件示例

Code block

```yaml
# config/production.yaml
app:
  name: "Eureka GEO"
  version: "1.0.0"
  environment: "production"

api:
  host: "0.0.0.0"
  port: 8000
  workers: 4
  cors_origins:
    - "https://eureka.com"
    - "https://admin.eureka.com"

database:
  postgres:
    host: "${DB_HOST}"
    port: 5432
    database: "eureka_geo"
    user: "${DB_USER}"
    password: "${DB_PASSWORD}"

  redis:
    url: "${REDIS_URL}"

  weaviate:
    url: "${WEAVIATE_URL}"

```

```yaml
llm:
  openai:
    api_key: "${OPENAI_API_KEY}"
    model: "gpt-4o"
    embedding_model: "text-embedding-3-large"
    temperature: 0.3

  cohere:
    api_key: "${COHERE_API_KEY}"
    rerank_model: "rerank-english-v2.0"

monitoring:
  prometheus:
    enabled: true
    port: 9090

  grafana:
    enabled: true
    dashboards:
      - "bmr_monitoring"
      - "latency_tracking"
      - "error_rates"
```

11.2 故障排除指南

Code block

```markdown
## 常见问题与解决方案

### 1. 向量检索延迟过高
**症状**: P95延迟超过500ms
**解决方案**:
- 检查Weaviate索引配置
- 增加缓存层
- 优化查询向量维度
- 考虑使用HNSW索引

### 2. 品牌提及率下降
**症状**: BMR低于60%
**解决方案**:
- 检查Guardrail规则
- 分析未提及品牌的查询
- 增强查询重写逻辑
- 更新品牌注入策略

### 3. LLM响应不一致
```

```
**症状**：相同查询返回不同答案
**解决方案**：
- 降低temperature参数
- 增加few-shot示例
- 实施响应缓存
- 考虑模型微调
```

---

12. 版本历史

版本	日期	主要更新	负责人
0.1.0	1/15/2025	初始架构设计	技术负责人
0.2.0	2/1/2025	RAG服务实现	后端团队
0.3.0	3/1/2025	API完整实现	后端团队
0.4.0	4/1/2025	性能优化	ML团队
1.0.0	5/1/2025	正式发布	全体团队

---

13. 参考资料

1. **GEO理论基础**
   - Generative Engine Optimization白皮书
   - Athena & Profound案例研究
2. **技术文档**
   - LangChain Documentation
   - Weaviate Vector Database
   - FastAPI Framework
3. **Eureka产品资料**
   - 产品手册和技术规格
   - 专利文档（CN202110384521.6）
   - 用户评测和反馈数据
4. **行业标准**
   - Schema.org结构化数据规范
   - OpenAI API最佳实践

- GDPR合规指南

文档结束

本开发文档为 GEO系统的完整实施指南，涵盖从架构设计到生产部署的全部技术细节。请根据实际开发进度持续更新本文档。