

完整 RAG 实施方案 - 详细步骤与代码

根据你的项目结构，我将提供完整的实现步骤、代码和文件位置。

📋 实施计划总览

复制阶段 1: 数据库准备 (Day 1-2)
阶段 2: 依赖安装与配置 (Day 2)
阶段 3: 核心工具函数 (Day 3-4)
阶段 4: API 路由实现 (Day 5-6)
阶段 5: 测试与优化 (Day 7)

📁 阶段 1: 数据库准备

步骤 1.1: 安装 pgvector 扩展

根据 pgvector 官方文档，有多种安装方式：

方案 A: 使用 Docker (推荐)

```
复制# 使用带 pgvector 的 PostgreSQL 镜像
docker pull pgvector/pgvector:pg16

# 运行容器
docker run -d \
  --name postgres-vector \
  -e POSTGRES_PASSWORD=your_password \
  -e POSTGRES_DB=ai_chat \
  -p 5432:5432 \
  pgvector/pgvector:pg16
```

方案 B: 云服务 (最简单)

如果使用 Supabase 或 Neon，pgvector 已预装，直接在 SQL 编辑器执行：

```
复制-- 启用 pgvector 扩展
CREATE EXTENSION IF NOT EXISTS vector;

-- 验证安装
SELECT * FROM pg_extension WHERE extname = 'vector';
```

方案 C: 本地安装 (Linux/Mac)

```
复制# Ubuntu/Debian
sudo apt install postgresql-16-pgvector

# macOS (Homebrew)
brew install pgvector

# 然后在 PostgreSQL 中启用
psql -U postgres -d ai_chat -c "CREATE EXTENSION vector;"
```

步骤 1.2: 修改 Prisma Schema

文件位置: `prisma/schema.prisma`

```
复制// =====
// Prisma Schema - 添加 RAG 支持
// =====

generator client {
  provider = "prisma-client-js"
  previewFeatures = ["postgresqlExtensions"] // ✅ 启用扩展支持
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
  extensions = [vector] // ✅ 声明使用 pgvector
}

// =====
// 现有模型（保持不变）
// =====

model User {
  id          Int      @id @default(autoincrement())
  email       String   @unique
  passwordHash String
  name        String?
  role        String   @default("user")
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt

  conversations Conversation[]
  pdfs         PDF[]

  @@map("users")
}
```

```

model Conversation {
    id          Int          @id @default(autoincrement())
    userId      Int
    title       String
    model       String       @default("openai/gpt-4o")
    createdAt   DateTime     @default(now())
    updatedAt   DateTime     @updatedAt

    user        User         @relation(fields: [userId], references: [id], onDelete: Cascade)
    messages    Message[]

    @@index([userId])
    @@map("conversations")
}

model Message {
    id          Int          @id @default(autoincrement())
    conversationId Int
    role        String
    content      String      @db.Text
    createdAt    DateTime    @default(now())

    conversation Conversation @relation(fields: [conversationId], references: [id], onDelete: Cascade)

    @@index([conversationId])
    @@map("messages")
}

// =====
// PDF 模型（修改）
// =====

model PDF {
    id          Int          @id @default(autoincrement())
    userId      Int
    name        String
    fileName    String
    filePath    String
    size        Int

    // ✅ 新增字段
    status      String       @default("processing") // processing/ready/failed
    totalChunks Int          @default(0)
    totalPages  Int?
    processedAt DateTime?
    errorMessage String?    @db.Text

    createdAt   DateTime     @default(now())
    updatedAt   DateTime     @updatedAt

    user        User         @relation(fields: [userId], references: [id], onDelete: Cascade)
    chunks      DocumentChunk[] // ✅ 关联文档块

```

```

    @@index([userId])
    @@index([status])
    @@map("pdfs")
}

// =====
// ✅ 新增：文档块模型（核心）
// =====

model DocumentChunk {
    id          Int          @id @default(autoincrement())
    pdfId       Int

    // 内容信息
    chunkIndex Int          // 第几块（从 0 开始）
    content     String       @db.Text

    // ✅ 向量字段（使用 Unsupported 类型）
    embedding   Unsupported("vector(1536)")? // OpenAI embedding 维度

    // 元数据
    pageNumber Int?         // 来源页码
    startChar   Int?         // 起始字符位置
    endChar     Int?         // 结束字符位置
    tokenCount  Int?         // token 数量

    metadata    Json?        // 其他元信息

    createdAt   DateTime @default(now())

    pdf PDF @relation(fields: [pdfId], references: [id], onDelete: Cascade)

    @@index([pdfId])
    @@index([chunkIndex])
    @@map("document_chunks")
}

```

关键说明：

- `Unsupported("vector(1536)")`: Prisma 对 pgvector 的支持方式
- `1536`: OpenAI text-embedding-3-small 的向量维度
- `@@index`: 加速查询性能

步骤 1.3: 创建数据库迁移

```
复制# 生成迁移文件
npx prisma migrate dev --name add_rag_support

# 如果遇到问题，可以先重置
# npx prisma migrate reset

# 生成 Prisma Client
npx prisma generate
```

步骤 1.4: 创建向量索引 (SQL)

文件位置: 创建新文件 `prisma/migrations/create_vector_index.sql`

```
复制-- =====
-- 创建向量相似度搜索索引
-- =====

-- 使用 IVFFlat 索引（适合中小规模数据）
CREATE INDEX IF NOT EXISTS document_chunks_embedding_idx
ON document_chunks
USING ivfflat (embedding vector_cosine_ops)
WITH (lists = 100);

-- 说明：
-- - ivfflat: 倒排文件索引，适合 10K-1M 向量
-- - vector_cosine_ops: 余弦相似度（推荐）
-- - lists = 100: 聚类数量（建议为行数的平方根）

-- 如果数据量更大 (> 1M)，使用 HNSW 索引：
-- CREATE INDEX document_chunks_embedding_idx
-- ON document_chunks
-- USING hnsw (embedding vector_cosine_ops);
```

手动执行:

```
复制# 连接数据库执行
psql -U your_user -d ai_chat -f prisma/migrations/create_vector_index.sql
```



阶段 2: 依赖安装与配置

步骤 2.1: 安装 NPM 包

```
复制# 进入项目目录
cd ai-chat-app

# 安装 LangChain 相关包
npm install langchain @langchain/openai @langchain/community

# 安装文本处理工具
npm install tiktoken # Token 计数

# 安装 pgvector 客户端（可选，Prisma 已支持）
npm install pgvector


# 验证安装
npm list langchain
```

步骤 2.2: 更新环境变量

文件位置: `.env.local`

```
复制# =====
# 现有配置（保持不变）
# =====

DATABASE_URL="postgresql://user:password@localhost:5432/ai_chat"
NEXTAUTH_SECRET="your-secret-key"
NEXTAUTH_URL="http://localhost:3000"
OPENAI_API_KEY="sk-or-v1-xxxxx" # OpenRouter API Key

# =====
#  新增: RAG 配置
# =====

# Embedding 模型配置
OPENAI_EMBEDDING_MODEL="text-embedding-3-small"
OPENAI_EMBEDDING_DIMENSIONS=1536

# 文档分块配置
CHUNK_SIZE=1000           # 每块字符数
CHUNK_OVERLAP=200        # 重叠字符数
MAX_CHUNKS_PER_PDF=500   # 单个 PDF 最大块数

# 检索配置
RETRIEVAL_TOP_K=5         # 返回最相关的 K 个块
SIMILARITY_THRESHOLD=0.7  # 相似度阈值（0-1）

# 处理配置
```

```
ENABLE_ASYNC_PROCESSING=true    # 启用异步处理
MAX_PDF_SIZE_MB=20              # 最大 PDF 大小
```

🔧 阶段 3: 核心工具函数

步骤 3.1: 创建 Embedding 工具

文件位置: `lib/rag/embeddings.js`

```
复制/**
 * =====
 * Embedding 工具 (lib/rag/embeddings.js)
 * =====
 *
 * 功能:
 *   1. 文本向量化 (单个/批量)
 *   2. Token 计数
 *   3. 成本估算
 *
 * 使用:
 *   import { embedText, embedBatch } from '@lib/rag/embeddings';
 * =====
 */

import { OpenAIEmbeddings } from '@langchain/openai';
import { encoding_for_model } from 'tiktoken';

// =====
// 初始化 Embedding 模型
// =====

const embeddings = new OpenAIEmbeddings({
  openAIApiKey: process.env.OPENAI_API_KEY,
  modelName: process.env.OPENAI_EMBEDDING_MODEL || 'text-embedding-3-small',
  dimensions: parseInt(process.env.OPENAI_EMBEDDING_DIMENSIONS || '1536'),
  configuration: {
    baseUrl: 'https://openrouter.ai/api/v1', // 如果使用 OpenRouter
  },
});

// =====
// Token 计数器
// =====

let tokenizer;
try {
  tokenizer = encoding_for_model('text-embedding-3-small');
} catch (error) {
  console.warn('⚠️ Tiktoken 初始化失败, 使用估算方法');
}
```

```

/**
 * 计算文本 token 数量
 * @param {string} text - 输入文本
 * @returns {number} token 数量
 */
export function countTokens(text) {
  if (!text) return 0;

  if (tokenizer) {
    try {
      const tokens = tokenizer.encode(text);
      return tokens.length;
    } catch (error) {
      console.error('Token 计数失败:', error);
    }
  }

  // 回退：粗略估算 (1 token ≈ 4 字符)
  return Math.ceil(text.length / 4);
}

// =====
// 单个文本向量化
// =====

/**
 * 将单个文本转换为向量
 * @param {string} text - 输入文本
 * @returns {Promise<number[]>} 向量数组
 */
export async function embedText(text) {
  if (!text || !text.trim()) {
    throw new Error('文本不能为空');
  }

  try {
    console.log('🔄 开始向量化, 文本长度:', text.length);
    const startTime = Date.now();

    const vector = await embeddings.embedQuery(text);

    const duration = Date.now() - startTime;
    console.log('✅ 向量化完成, 耗时: ${duration}ms, 维度: ${vector.length}`);

    return vector;
  } catch (error) {
    console.error('❌ 向量化失败:', error);
    throw new Error(`向量化失败: ${error.message}`);
  }
}

// =====
// 批量文本向量化

```

```
// =====

/**
 * 批量向量化（更高效）
 * @param {string[]} texts - 文本数组
 * @param {Object} options - 配置选项
 * @returns {Promise<number[][]>} 向量数组
 */
export async function embedBatch(texts, options = {}) {
  const {
    batchSize = 100,      // 每批处理数量
    showProgress = true,  // 显示进度
  } = options;

  if (!texts || texts.length === 0) {
    return [];
  }

  console.log(`🔄 批量向量化开始，总数：${texts.length}`);
  const startTime = Date.now();

  const results = [];

  // 分批处理
  for (let i = 0; i < texts.length; i += batchSize) {
    const batch = texts.slice(i, i + batchSize);

    if (showProgress) {
      console.log(`📊 处理进度：${i + batch.length}/${texts.length}`);
    }

    try {
      const vectors = await embeddings.embedDocuments(batch);
      results.push(...vectors);
    } catch (error) {
      console.error(`❌ 批次 ${i}-${i + batch.length} 失败:`, error);

      // 失败时逐个重试
      for (const text of batch) {
        try {
          const vector = await embedText(text);
          results.push(vector);
        } catch (retryError) {
          console.error('单个重试也失败:', retryError);
          results.push(null); // 标记失败
        }
      }
    }
  }

  // 避免 API 限流
  if (i + batchSize < texts.length) {
    await new Promise(resolve => setTimeout(resolve, 100));
  }
}
```

```

    }

    const duration = Date.now() - startTime;
    console.log(`✅ 批量向量化完成, 耗时: ${duration}ms`);

    return results;
}

// =====
// 成本估算
// =====

/**
 * 估算向量化成本
 * @param {number} tokenCount - Token 数量
 * @returns {Object} 成本信息
 */
export function estimateCost(tokenCount) {
    // OpenAI text-embedding-3-small: $0.02 / 1M tokens
    const costPerMillion = 0.02;
    const cost = (tokenCount / 1000000) * costPerMillion;

    return {
        tokens: tokenCount,
        cost: cost.toFixed(6),
        costUSD: `$$${cost.toFixed(6)}`,
        costCNY: `¥${(cost * 7.2).toFixed(4)}`, // 假设汇率 1:7.2
    };
}

// =====
// 导出
// =====

export default {
    embedText,
    embedBatch,
    countTokens,
    estimateCost,
};

```

步骤 3.2: 创建文本分块工具

文件位置: `lib/rag/chunking.js`

```

复制/**
 * =====
 * 文本分块工具 (lib/rag/chunking.js)
 * =====
 *
 * 功能:

```

```

* 1. 递归字符分割
* 2. 保持语义完整
* 3. 添加元数据
*
* 使用:
* import { chunkText } from '@lib/rag/chunking';
* =====
*/

import { RecursiveCharacterTextSplitter } from 'langchain/text_splitter';
import { countTokens } from './embeddings';

// =====
// 配置
// =====

const DEFAULT_CHUNK_SIZE = parseInt(process.env.CHUNK_SIZE || '1000');
const DEFAULT_CHUNK_OVERLAP = parseInt(process.env.CHUNK_OVERLAP || '200');

// =====
// 创建分块器
// =====

/**
 * 创建文本分块器
 * @param {Object} options - 配置选项
 * @returns {RecursiveCharacterTextSplitter}
 */
export function createSplitter(options = {}) {
  const {
    chunkSize = DEFAULT_CHUNK_SIZE,
    chunkOverlap = DEFAULT_CHUNK_OVERLAP,
    separators = ['\n\n', '\n', '。', '!', '?', ';', ':', ', ', ' ', ' ', ' ', ' '],
  } = options;

  return new RecursiveCharacterTextSplitter({
    chunkSize,
    chunkOverlap,
    separators,
    lengthFunction: (text) => text.length, // 按字符计数
  });
}

// =====
// 文本分块（核心函数）
// =====

/**
 * 将文本分块
 * @param {string} text - 输入文本
 * @param {Object} metadata - 元数据
 * @param {Object} options - 配置选项
 * @returns {Promise<Array>} 分块结果

```

```

*/
export async function chunkText(text, metadata = {}, options = {}) {
  if (!text || !text.trim()) {
    console.warn('⚠️ 输入文本为空');
    return [];
  }

  console.log('🔪 开始文本分块...');
  console.log('📏 原始文本长度:', text.length);

  const startTime = Date.now();

  try {
    // 创建分块器
    const splitter = createSplitter(options);

    // 执行分块
    const docs = await splitter.createDocuments([text], [metadata]);

    // 处理结果
    const chunks = docs.map((doc, index) => {
      const content = doc.pageContent;
      const tokens = countTokens(content);

      return {
        chunkIndex: index,
        content: content,
        tokenCount: tokens,
        charCount: content.length,
        metadata: {
          ...doc.metadata,
          ...metadata,
        },
      };
    });

    const duration = Date.now() - startTime;

    console.log('✅ 分块完成');
    console.log('📊 统计信息:', {
      totalChunks: chunks.length,
      avgChunkSize: Math.round(text.length / chunks.length),
      totalTokens: chunks.reduce((sum, c) => sum + c.tokenCount, 0),
      duration: `${duration}ms`,
    });

    return chunks;
  } catch (error) {
    console.error('❌ 分块失败:', error);
    throw new Error(`文本分块失败: ${error.message}`);
  }
}

```

```

// =====
// 智能分块（按页码）
// =====

/**
 * 按页码分块（适合 PDF）
 * @param {Object} pdfData - PDF 解析数据
 * @param {Object} options - 配置选项
 * @returns {Promise<Array>} 分块结果
 */
export async function chunkByPages(pdfData, options = {}) {
  const { text, numpages, metadata } = pdfData;

  console.log('📄 按页码分块，总页数:', numpages);

  // 如果有页码信息，按页分块
  if (metadata?.pageTexts && Array.isArray(metadata.pageTexts)) {
    const allChunks = [];

    for (const page of metadata.pageTexts) {
      const pageChunks = await chunkText(
        page.text,
        {
          pageNumber: page.page,
          source: 'pdf',
        },
        options
      );

      allChunks.push(...pageChunks);
    }

    return allChunks;
  }

  // 否则整体分块
  return chunkText(text, { source: 'pdf', totalPages: numpages }, options);
}

// =====
// 导出
// =====

export default {
  chunkText,
  chunkByPages,
  createSplitter,
};

```

步骤 3.3: 创建向量检索工具

文件位置: `lib/rag/retrieval.js`

```
复制/**
 * =====
 * 向量检索工具 (lib/rag/retrieval.js)
 * =====
 *
 * 功能:
 *   1. 相似度搜索
 *   2. 混合检索 (向量 + 关键词)
 *   3. 结果重排序
 *
 * 使用:
 *   import { searchSimilarChunks } from '@lib/rag/retrieval';
 * =====
 */

import { prisma } from '@lib/prisma';
import { embedText } from '../embeddings';

// =====
// 配置
// =====

const DEFAULT_TOP_K = parseInt(process.env.RETRIEVAL_TOP_K || '5');
const DEFAULT_THRESHOLD = parseFloat(process.env.SIMILARITY_THRESHOLD || '0.7');

// =====
// 向量相似度搜索
// =====

/**
 * 搜索相似文档块
 * @param {string} query - 查询文本
 * @param {Object} options - 配置选项
 * @returns {Promise<Array>} 相似文档块
 */
export async function searchSimilarChunks(query, options = {}) {
  const {
    pdfId = null,           // 限制在特定 PDF
    topK = DEFAULT_TOP_K,   // 返回数量
    threshold = DEFAULT_THRESHOLD, // 相似度阈值
    includeMetadata = true, // 包含元数据
  } = options;

  console.log('🔍 开始相似度搜索...');
  console.log('📄 查询:', query);

  try {
    // 1. 将查询向量化
    const queryVector = await embedText(query);
```

```

console.log('✅ 查询向量化完成');

// 2. 构建 SQL 查询
// 使用 pgvector 的余弦相似度运算符 <=>
const vectorString = `[${queryVector.join(',')}]`;

let sql = `
  SELECT
    dc.id,
    dc."pdfId",
    dc."chunkIndex",
    dc.content,
    dc."pageNumber",
    dc."tokenCount",
    dc.metadata,
    p.name as "pdfName",
    p."filePath" as "pdfPath",
    1 - (dc.embedding <=> $1::vector) as similarity
  FROM document_chunks dc
  JOIN pdfs p ON dc."pdfId" = p.id
  WHERE dc.embedding IS NOT NULL
`;

const params = [vectorString];

// 3. 添加过滤条件
if (pdfId) {
  sql += ` AND dc."pdfId" = ${params.length + 1}`;
  params.push(pdfId);
}

// 4. 添加相似度阈值
sql += ` AND (1 - (dc.embedding <=> $1::vector)) >= ${params.length + 1}`;
params.push(threshold);

// 5. 排序和限制
sql += `
  ORDER BY dc.embedding <=> $1::vector
  LIMIT ${params.length + 1}
`;
params.push(topK);

console.log('📄 执行数据库查询...');

// 6. 执行查询
const results = await prisma.$queryRawUnsafe(sql, ...params);

console.log(`✅ 找到 ${results.length} 个相似块`);

// 7. 格式化结果
const formattedResults = results.map(row => ({
  id: row.id,
  pdfId: row.pdfId,

```

```

        pdfName: row.pdfName,
        pdfPath: row.pdfPath,
        chunkIndex: row.chunkIndex,
        content: row.content,
        pageNumber: row.pageNumber,
        tokenCount: row.tokenCount,
        similarity: parseFloat(row.similarity.toFixed(4)),
        metadata: includeMetadata ? row.metadata : undefined,
    }));

// 8. 打印结果摘要
if (formattedResults.length > 0) {
    console.log('📄 检索结果摘要:');
    formattedResults.forEach((r, i) => {
        console.log(` ${i + 1}. 相似度: ${r.similarity}, 页码: ${r.pageNumber || 'N/A'}, 长
度: ${r.content.length}`);
    });
} else {
    console.log('⚠️ 未找到满足条件的结果');
}

return formattedResults;

} catch (error) {
    console.error('❌ 检索失败:', error);
    throw new Error(`向量检索失败: ${error.message}`);
}
}

// =====
// 混合检索（向量 + 关键词）
// =====

/**
 * 混合检索：结合向量相似度和关键词匹配
 * @param {string} query - 查询文本
 * @param {Object} options - 配置选项
 * @returns {Promise<Array>} 检索结果
 */
export async function hybridSearch(query, options = {}) {
    const {
        pdfId = null,
        topK = DEFAULT_TOP_K,
        vectorWeight = 0.7,    // 向量检索权重
        keywordWeight = 0.3,    // 关键词权重
    } = options;

    console.log('🔍 混合检索开始...');

    try {
        // 1. 向量检索
        const vectorResults = await searchSimilarChunks(query, {
            pdfId,

```

```

    topK: topK * 2, // 获取更多候选
    threshold: 0.5, // 降低阈值
  });

// 2. 关键词检索（使用 PostgreSQL 全文搜索）
const keywords = query.split(/\s+/).filter(k => k.length > 1);

let keywordResults = [];
if (keywords.length > 0) {
  const keywordQuery = keywords.map(k => `%${k}%`).join(' ');

  keywordResults = await prisma.documentChunk.findMany({
    where: {
      pdfId: pdfId || undefined,
      content: {
        contains: keywordQuery,
        mode: 'insensitive',
      },
    },
    include: {
      pdf: {
        select: {
          name: true,
          filePath: true,
        },
      },
    },
    take: topK * 2,
  });
}

// 3. 合并结果并计算综合得分
const combinedMap = new Map();

// 处理向量结果
vectorResults.forEach(result => {
  combinedMap.set(result.id, {
    ...result,
    vectorScore: result.similarity,
    keywordScore: 0,
    finalScore: result.similarity * vectorWeight,
  });
});

// 处理关键词结果
keywordResults.forEach(result => {
  const keywordScore = calculateKeywordScore(result.content, keywords);

  if (combinedMap.has(result.id)) {
    const existing = combinedMap.get(result.id);
    existing.keywordScore = keywordScore;
    existing.finalScore =
      existing.vectorScore * vectorWeight +

```

```

        keywordScore * keywordweight;
    } else {
        combinedMap.set(result.id, {
            ...result,
            pdfName: result.pdf.name,
            pdfPath: result.pdf.filePath,
            vectorScore: 0,
            keywordScore: keywordScore,
            finalScore: keywordScore * keywordweight,
        });
    }
});

// 4. 排序并返回 Top K
const finalResults = Array.from(combinedMap.values())
    .sort((a, b) => b.finalScore - a.finalScore)
    .slice(0, topK);

console.log(`✅ 混合检索完成, 返回 ${finalResults.length} 个结果`);

return finalResults;

} catch (error) {
    console.error('❌ 混合检索失败:', error);
    throw error;
}
}

// =====
// 辅助函数: 计算关键词匹配得分
// =====

function calculateKeywordScore(content, keywords) {
    if (!keywords || keywords.length === 0) return 0;

    const lowerContent = content.toLowerCase();
    let matchCount = 0;

    keywords.forEach(keyword => {
        const regex = new RegExp(keyword.toLowerCase(), 'g');
        const matches = lowerContent.match(regex);
        if (matches) {
            matchCount += matches.length;
        }
    });

    // 归一化到 0-1
    return Math.min(matchCount / (keywords.length * 3), 1);
}

// =====
// 导出
// =====

```

```
export default {
  searchSimilarChunks,
  hybridSearch,
};
```

阶段 4: API 路由实现

步骤 4.1: PDF 处理 API

文件位置: `app/api/pdf/process/route.js`

复制/**

```
* =====
* PDF 处理 API (app/api/pdf/process/route.js)
* =====
*
* 功能:
*   1. 解析 PDF 文本
*   2. 文本分块
*   3. 向量化并存储
*
* 路由: POST /api/pdf/process
* =====
*/

import { NextResponse } from 'next/server';
import { auth } from '@app/api/auth/[...nextauth]/route';
import { prisma } from '@lib/prisma';
import fs from 'fs/promises';
import path from 'path';
import pdf from 'pdf-parse';

import { chunkText } from '@lib/rag/chunking';
import { embedBatch, estimateCost, countTokens } from '@lib/rag/embeddings';

/**
 * POST - 处理 PDF 文件
 */
export async function POST(request) {
  console.log('🚀 开始处理 PDF...');

  try {
    // 1. 身份验证
    const session = await auth();
    if (!session?.user) {
      return NextResponse.json({ error: '请先登录' }, { status: 401 });
    }

    // 2. 获取参数
    const { pdfId } = await request.json();
```

```
if (!pdfId) {
  return NextResponse.json({ error: '缺少 PDF ID' }, { status: 400 });
}

console.log('📄 PDF ID:', pdfId);

// 3. 查询 PDF 记录
const pdfRecord = await prisma.pdf.findFirst({
  where: {
    id: pdfId,
    userId: session.user.id,
  },
});

if (!pdfRecord) {
  return NextResponse.json({ error: 'PDF 不存在' }, { status: 404 });
}

console.log('✅ 找到 PDF:', pdfRecord.name);

// 4. 更新状态为处理中
await prisma.pdf.update({
  where: { id: pdfId },
  data: { status: 'processing' },
});

// 5. 读取 PDF 文件
const filePath = path.join(process.cwd(), 'public', pdfRecord.filePath);
console.log('📁 文件路径:', filePath);

const dataBuffer = await fs.readFile(filePath);
const pdfData = await pdf(dataBuffer);

console.log('📄 PDF 信息:', {
  pages: pdfData.numpages,
  textLength: pdfData.text.length,
});

// 6. 文本分块
console.log('✂️ 开始分块...');
const chunks = await chunkText(pdfData.text, {
  pdfId: pdfId,
  pdfName: pdfRecord.name,
  totalPages: pdfData.numpages,
});

console.log(`✅ 分块完成, 共 ${chunks.length} 块`);

// 7. 批量向量化
console.log('🔄 开始向量化...');
const contents = chunks.map(c => c.content);
const vectors = await embedBatch(contents, { showProgress: true });
```

```

// 8. 保存到数据库
console.log('📁 保存到数据库...');

const chunksToInsert = chunks.map((chunk, index) => ({
  pdfId: pdfId,
  chunkIndex: index,
  content: chunk.content,
  embedding: vectors[index] ? `[${vectors[index].join(',')}]` : null,
  pageNumber: chunk.metadata?.pageNumber || null,
  tokenCount: chunk.tokenCount,
  metadata: chunk.metadata,
}));

// 使用事务批量插入
await prisma.$transaction(async (tx) => {
  // 删除旧的块（如果重新处理）
  await tx.documentChunk.deleteMany({
    where: { pdfId: pdfId },
  });

  // 插入新块
  await tx.documentChunk.createMany({
    data: chunksToInsert,
  });

  // 更新 PDF 状态
  await tx.pdf.update({
    where: { id: pdfId },
    data: {
      status: 'ready',
      totalChunks: chunks.length,
      totalPages: pdfData.numpages,
      processedAt: new Date(),
    },
  });
});

console.log('✅ 处理完成! ');

// 9. 计算成本
const totalTokens = chunks.reduce((sum, c) => sum + c.tokenCount, 0);
const cost = estimateCost(totalTokens);

return NextResponse.json({
  success: true,
  message: 'PDF 处理完成',
  data: {
    pdfId: pdfId,
    totalChunks: chunks.length,
    totalPages: pdfData.numpages,
    totalTokens: totalTokens,
    cost: cost,
  },
});

```

```

    },
  });

  } catch (error) {
    console.error('❌ 处理失败:', error);

    // 更新状态为失败
    try {
      const { pdfId } = await request.json();
      await prisma.pdf.update({
        where: { id: pdfId },
        data: {
          status: 'failed',
          errorMessage: error.message,
        },
      });
    } catch (updateError) {
      console.error('更新状态失败:', updateError);
    }

    return NextResponse.json(
      { error: `处理失败: ${error.message}` },
      { status: 500 }
    );
  }
}

```

步骤 4.2: RAG 问答 API

文件位置: `app/api/chat-pdf-rag/route.js`

复制/**

```

* =====
* RAG 问答 API (app/api/chat-pdf-rag/route.js)
* =====
*
* 功能:
*   1. 接收用户问题
*   2. 向量检索相关文档块
*   3. 构建增强提示词
*   4. 调用 LLM 生成答案
*
* 路由: POST /api/chat-pdf-rag
* =====
*/

```

```

import { NextResponse } from 'next/server';
import { auth } from '@app/api/auth/[...nextauth]/route';
import { ChatOpenAI } from '@langchain/openai';
import { searchSimilarChunks } from '@lib/rag/retrieval';

```

```

export async function POST(request) {
  console.log('🗨️ RAG 问答请求');

  try {
    // 1. 身份验证
    const session = await auth();
    if (!session?.user) {
      return NextResponse.json({ error: '请先登录' }, { status: 401 });
    }

    // 2. 解析参数
    const { message, pdfId, history, model } = await request.json();

    if (!message?.trim()) {
      return NextResponse.json({ error: '消息不能为空' }, { status: 400 });
    }

    if (!pdfId) {
      return NextResponse.json({ error: '请选择 PDF' }, { status: 400 });
    }

    console.log('👤 用户问题:', message);
    console.log('📄 PDF ID:', pdfId);

    // 3. 向量检索
    console.log('🔍 开始检索相关内容...');
    const relevantChunks = await searchSimilarChunks(message, {
      pdfId: pdfId,
      topK: 5,
      threshold: 0.7,
    });

    if (relevantChunks.length === 0) {
      console.log('⚠️ 未找到相关内容');
      return NextResponse.json({
        error: '未在文档中找到相关内容，请尝试换个问法',
      }, { status: 404 });
    }

    console.log(`✅ 找到 ${relevantChunks.length} 个相关块`);

    // 4. 构建增强提示词
    const contextText = relevantChunks
      .map((chunk, index) => {
        return `【来源: ${chunk.pdfName} 第${chunk.pageNumber} || '?'页】\n${chunk.content}`;
      })
      .join('\n\n---\n\n');

    const systemPrompt = `你是一个专业的文档问答助手。请基于以下参考内容回答用户问题。

## 参考内容:
${contextText}

```

回答要求:

1. **仅基于参考内容回答**，不要编造信息
2. 如果参考内容不足以回答问题，明确告知用户
3. 引用时标注来源（如：根据第3页...）
4. 使用 **Markdown** 格式美化回答
5. 保持专业、准确、友好的语气

引用格式示例:

根据文档第5页的内容，机器学习是...[^1]

参考来源:

```
${relevantChunks.map((c, i) => `[^${i+1}]: ${c.pdfName} 第${c.pageNumber} || '?'}页`)  
).join('\n')}`;
```

```
// 5. 构建消息历史
```

```
const messages = [  
  { role: 'system', content: systemPrompt },  
  ...(history || []),  
  { role: 'user', content: message },  
];
```

```
// 6. 调用 LLM
```

```
console.log('🤖 调用 AI 模型...');  
const llm = new ChatOpenAI({  
  modelName: model || 'openai/gpt-4o',  
  openAIApiKey: process.env.OPENAI_API_KEY,  
  configuration: {  
    baseUrl: 'https://openrouter.ai/api/v1',  
  },  
  streaming: true,  
  temperature: 0.3, // 降低创造性，提高准确性  
});
```

```
const stream = await llm.stream(messages);
```

```
// 7. 返回流式响应
```

```
const encoder = new TextEncoder();  
const readable = new ReadableStream({  
  async start(controller) {  
    try {  
      for await (const chunk of stream) {  
        if (chunk.content) {  
          controller.enqueue(  
            encoder.encode(`data: ${JSON.stringify({ content: chunk.content })}\n\n`)  
          );  
        }  
      }  
    }  
  }  
});
```

```
// 发送引用信息
```

```
controller.enqueue(  
  encoder.encode(`data: ${JSON.stringify({  
    citations: relevantChunks.map(c => ({
```

```

        pdfName: c.pdfName,
        pageNumber: c.pageNumber,
        similarity: c.similarity,
    )))
    }}}\n\n`)
);

    controller.enqueue(encoder.encode('data: [DONE]\n\n'));
    controller.close();
} catch (error) {
    console.error('流式响应错误:', error);
    controller.error(error);
}
},
});

return new Response(readable, {
    headers: {
        'Content-Type': 'text/event-stream',
        'Cache-Control': 'no-cache',
        'Connection': 'keep-alive',
    },
});

} catch (error) {
    console.error('❌ RAG 问答失败:', error);
    return NextResponse.json(
        { error: `问答失败: ${error.message}` },
        { status: 500 }
    );
}
}
}

```

步骤 4.3: 修改上传 API

文件位置: `app/api/upload/route.js` (修改现有文件)

在现有上传成功后, 添加自动触发处理:

```

复制// 在文件上传成功后添加 (约第 80 行)

// 保存记录到数据库
const pdfRecord = await prisma.pdf.create({
    data: {
        userId: session.user.id,
        name: originalName,
        fileName: fileName,
        filePath: fileUrl,
        size: file.size,
        status: 'processing', // ✅ 初始状态为处理中
    },
});

```

```
// ✅ 新增：自动触发处理
if (process.env.ENABLE_ASYNC_PROCESSING === 'true') {
  // 异步调用处理 API（不等待）
  fetch(`${process.env.NEXTAUTH_URL}/api/pdf/process`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ pdfId: pdfRecord.id }),
  }).catch(err => console.error('触发处理失败:', err));

  console.log('✅ 已触发异步处理');
}

return NextResponse.json({
  success: true,
  message: '上传成功，正在处理中...',
  data: pdfRecord,
});
```

阶段 5: 测试与验证

步骤 5.1: 创建测试脚本

文件位置: `scripts/test-rag.js`

```
复制/**
 * RAG 功能测试脚本
 * 运行: node scripts/test-rag.js
 */

import { embedText, embedBatch } from '../lib/rag/embeddings.js';
import { chunkText } from '../lib/rag/chunking.js';
import { searchSimilarChunks } from '../lib/rag/retrieval.js';

async function testEmbedding() {
  console.log('🔧 测试向量化...');

  const text = '人工智能是计算机科学的一个分支';
  const vector = await embedText(text);

  console.log('✅ 向量维度:', vector.length);
  console.log('✅ 前5个值:', vector.slice(0, 5));
}

async function testChunking() {
  console.log('\n🔧 测试分块...');

  const longText = '这是一段很长的文本...'.repeat(100);
  const chunks = await chunkText(longText);

  console.log('✅ 分块数量:', chunks.length);
```

```

    console.log('✅ 第一块长度:', chunks[0].content.length);
  }

  async function testRetrieval() {
    console.log('\n🔍 测试检索...');

    const query = '什么是机器学习? ';
    const results = await searchSimilarChunks(query, { topK: 3 });

    console.log('✅ 检索结果数:', results.length);
    if (results.length > 0) {
      console.log('✅ 最高相似度:', results[0].similarity);
    }
  }

  async function runTests() {
    try {
      await testEmbedding();
      await testChunking();
      await testRetrieval();

      console.log('\n✅ 所有测试通过! ');
    } catch (error) {
      console.error('\n❌ 测试失败:', error);
    }
  }

  runTests();

```

运行测试:

复制

```
node scripts/test-rag.js
```

步骤 5.2: 端到端测试流程

1. 上传 PDF

复制

```
curl -X POST http://localhost:3000/api/upload \
-H "Cookie: next-auth.session-token=YOUR_TOKEN" \
-F "file=@test.pdf"
```

1. 检查处理状态

复制

```
SELECT id, name, status, totalChunks FROM pdfs ORDER BY id DESC LIMIT 1;
```

1. 查看文档块

```
复制SELECT COUNT(*) FROM document_chunks WHERE "pdfId" = 1;
SELECT "chunkIndex", LENGTH(content), "pageNumber"
FROM document_chunks
WHERE "pdfId" = 1
LIMIT 5;
```

1. 测试检索

```
复制curl -X POST http://localhost:3000/api/chat-pdf-rag \
-H "Content-Type: application/json" \
-H "Cookie: next-auth.session-token=YOUR_TOKEN" \
-d '{
  "message": "这个文档讲了什么？",
  "pdfId": 1,
  "model": "openai/gpt-4o"
}'
```

性能监控

步骤 6.1: 添加日志记录

文件位置: `lib/rag/logger.js`

```
复制/**
 * RAG 操作日志记录
 */

export function logProcessing(pdfId, stats) {
  console.log('📊 处理统计:', {
    pdfId,
    chunks: stats.totalChunks,
    tokens: stats.totalTokens,
    cost: stats.cost,
    duration: stats.duration,
  });

  // 可以扩展为写入数据库或文件
}

export function logRetrieval(query, results, duration) {
  console.log('🔍 检索统计:', {
    query: query.substring(0, 50),
    resultsCount: results.length,
    avgSimilarity: results.reduce((sum, r) => sum + r.similarity, 0) / results.length,
    duration: `${duration}ms`,
  });
}
```

总结与检查清单

完成的功能：

- ✓ pgvector 扩展安装
- ✓ Prisma Schema 更新 (DocumentChunk 模型)
- ✓ 向量化工具 (embeddings.js)
- ✓ 文本分块工具 (chunking.js)
- ✓ 向量检索工具 (retrieval.js)
- ✓ PDF 处理 API (/api/pdf/process)
- ✓ RAG 问答 API (/api/chat-pdf-rag)
- ✓ 测试脚本

文件结构总览：

```
复制ai-chat-app/  
├─ app/  
│   └─ api/  
│       └─ pdf/  
│           └─ process/route.js      ✓ 新增  
│           └─ chat-pdf-rag/route.js ✓ 新增  
│           └─ upload/route.js      ✓ 修改  
├─ lib/  
│   └─ rag/                          ✓ 新增目录  
│       └─ embeddings.js            ✓ 新增  
│       └─ chunking.js              ✓ 新增  
│       └─ retrieval.js             ✓ 新增  
│       └─ logger.js                ✓ 新增  
├─ prisma/  
│   └─ schema.prisma                ✓ 修改  
│   └─ migrations/  
│       └─ create_vector_index.sql  ✓ 新增  
├─ scripts/  
│   └─ test-rag.js                  ✓ 新增  
└─ .env.local                       ✓ 修改
```

下一步操作：

1. 安装依赖： `npm install`
2. 数据库迁移： `npx prisma migrate dev`
3. 测试向量化： `node scripts/test-rag.js`
4. 上传测试 PDF
5. 验证检索功能

参考资料:

: pgvector GitHub - <https://github.com/pgvector/pgvector>

: pgvector 安装指南 - <https://medium.com/@besttechreads/step-by-step-guide-to-installing-pgvector-and-loading-data-in-postgresql-f2cffb5dec43>

: PostgreSQL pgvector 入门 - <https://www.yugabyte.com/blog/postgresql-pgvector-getting-started/>

: 将 PostgreSQL 转为向量数据库 - <https://www.tigerdata.com/learn/postgresql-extensions-pgvector>

需要我详细解释某个步骤或提供前端集成代码吗? 🚀