Supermemory 核心技术实现指南

1. 透明代理机制实现

1.1 代理服务器核心代码

```
// proxy-server.ts
import express from 'express';
import { MemoryManager } from './memory-manager';
import { LLMAdapter } from './llm-adapter';
import { ContextOptimizer } from './context-optimizer';

export class ProxyServer {
  private app: express.Application;
  private memoryManager: MemoryManager;
  private llmAdapter: LLMAdapter;
  private contextOptimizer: ContextOptimizer;

  constructor() {
    this.app = express();
    this.memoryManager = new MemoryManager();
    this.llmAdapter = new LLMAdapter();
    this.contextOptimizer = new ContextOptimizer();
    this.setupMiddleware();
    this.setupRoutes();
  }

  private setupMiddleware() {
    this.app.use(express.json({ limit: '10mb' }));
    this.app.use(this.authMiddleware);
    this.app.use(this.rateLimitMiddleware);
  }

  private setupRoutes() {
    // 透明代理主接口
    this.app.post('/v1/chat/completions', this.handleChatCompletion.
    this.app.post('/v1/completions', this.handleCompletion.bind(this
  }

  private async handleChatCompletion(req: express.Request, res: expr
    try {
      const { messages, model, ...options } = req.body;
      const projectId = req.headers['x-project-id'] as string;
      const sessionId = req.headers['x-session-id'] as string || thi

      // 1. 检索相关记忆
      const relevantMemories = await this.memoryManager.retrieveRele
        projectId,
```

```
        sessionId,
        messages
      );

      // 2. 优化上下文
      const optimizedContext = await this.contextOptimizer.optimizeC
        messages,
        relevantMemories,
        options.max_tokens || 4096
      );

      // 3. 调用LLM
      const response = await this.llmAdapter.callLLM(model, {
        messages: optimizedContext,
        ...options
      });

      // 4. 异步更新记忆
      this.memoryManager.updateMemory(projectId, sessionId, messages
        .catch(error => console.error('Memory update failed:', error

      // 5. 返回响应
      res.json({
        ...response,
        memory_info: {
          memories_used: relevantMemories.length,
          context_optimized: true,
          session_id: sessionId
        }
      });

    } catch (error) {
      console.error('Proxy error:', error);
      res.status(500).json({ error: 'Internal server error' });
    }
  }

  private authMiddleware(req: express.Request, res: express.Response
    const apiKey = req.headers.authorization?.replace('Bearer ', '')
    if (!apiKey || !this.validateApiKey(apiKey)) {
      return res.status(401).json({ error: 'Invalid API key' });
    }
    next();
  }

  private rateLimitMiddleware(req: express.Request, res: express.Res
    // 实现速率限制逻辑
    next();
  }

  private validateApiKey(apiKey: string): boolean {
    // 实现API密钥验证逻辑
```

```
    return true;
  }

  private generateSessionId(): string {
    return `session_${Date.now()}_${Math.random().toString(36).subst
  }
}
```

1.2 记忆管理器实现

```
// memory-manager.ts
import { VectorDatabase } from './vector-database';
import { EmbeddingService } from './embedding-service';
import { DatabaseService } from './database-service';
import { TextSegmenter, SegmentationConfig, TextSegment } from './te
import { TokenCounter } from './token-counter';
import { PrioritySorter, ContentItem, PriorityWeights } from './prio

export interface MemoryChunk {
  id: string;
  content: string;
  embedding: number[];
  relevanceScore: number;
  timestamp: Date;
  chunkType: string;
  chunkIndex: number;
  tokenCount: number;
  metadata: Record<string, any>;
}

export class MemoryManager {
  private vectorDB: VectorDatabase;
  private embeddingService: EmbeddingService;
  private database: DatabaseService;
  private textSegmenter: TextSegmenter;
  private tokenCounter: TokenCounter;
  private prioritySorter: PrioritySorter;

  constructor() {
    this.vectorDB = new VectorDatabase();
    this.embeddingService = new EmbeddingService();
    this.database = new DatabaseService();
    this.tokenCounter = new TokenCounter();

    // 默认配置
    const defaultSegmentationConfig: SegmentationConfig = {
      maxChunkSize: 512,
      overlapSize: 50,
      strategy: 'semantic',
      preserveBoundaries: true
    };
```

```
    const defaultPriorityWeights: PriorityWeights = {
      temporal: 0.3,
      relevance: 0.4,
      importance: 0.2,
      userPreference: 0.1
    };

    this.textSegmenter = new TextSegmenter(defaultSegmentationConfig
    this.prioritySorter = new PrioritySorter(defaultPriorityWeights)
  }

  async retrieveRelevantMemories(
    projectId: string,
    sessionId: string,
    messages: any[],
    maxTokens?: number
  ): Promise<MemoryChunk[]> {
    // 1. 获取查询向量
    const queryText = this.extractQueryText(messages);
    const queryEmbedding = await this.embeddingService.getEmbedding(

    // 2. 向量相似度搜索
    const vectorResults = await this.vectorDB.similaritySearch(
      queryEmbedding,
      {
        projectId,
        sessionId,
        limit: 30,
        threshold: 0.7
      }
    );

    // 3. 转换为ContentItem格式进行优先级排序
    const contentItems: ContentItem[] = vectorResults.map(chunk => (
      id: chunk.id,
      content: chunk.content,
      timestamp: chunk.timestamp,
      relevanceScore: chunk.relevanceScore,
      importanceScore: chunk.metadata.importance || 0.5,
      userInteractionScore: chunk.metadata.userPreference || 0.5,
      tokenCount: chunk.tokenCount,
      metadata: chunk.metadata
    }));

    // 4. 使用优先级排序器
    const sortedItems = this.prioritySorter.sortByPriority(contentIt

    // 5. 根据Token限制或数量限制选择记忆
    let selectedItems = sortedItems;
    if (maxTokens) {
      selectedItems = this.prioritySorter.selectTopItems(sortedItems
```

```
    } else {
      selectedItems = sortedItems.slice(0, 10);
    }

    // 6. 转换回MemoryChunk格式并应用时间衰减
    const selectedChunks = selectedItems.map(item => {
      const chunk = vectorResults.find(c => c.id === item.id)!;
      return chunk;
    });

    // 7. 应用时间衰减
    const rerankedResults = this.applyTimeDecay(selectedChunks);

    return rerankedResults;
  }

  async updateMemory(
    projectId: string,
    sessionId: string,
    messages: any[],
    response: any
  ): Promise<void> {
    try {
      // 1. 保存对话到数据库
      const conversationId = await this.database.saveConversation(
        projectId,
        sessionId,
        messages,
        response
      );

      // 2. 生成记忆块
      const memoryChunks = await this.generateMemoryChunks(
        conversationId,
        messages,
        response
      );

      // 3. 生成嵌入向量
      for (const chunk of memoryChunks) {
        chunk.embedding = await this.embeddingService.getEmbedding(c
      }

      // 4. 存储到向量数据库
      await this.vectorDB.insertMemories(memoryChunks);

      // 5. 清理过期记忆
      await this.cleanupExpiredMemories(projectId);

    } catch (error) {
      console.error('Memory update error:', error);
      throw error;
```

```
  }
}

private extractQueryText(messages: any[]): string {
  // 提取最近几条消息作为查询文本
  return messages
    .slice(-3)
    .map(msg => msg.content)
    .join(' ');
}

private applyTimeDecay(memories: MemoryChunk[]): MemoryChunk[] {
  const now = new Date();

  return memories.map(memory => {
    const ageInHours = (now.getTime() - memory.timestamp.getTime()
    const decayFactor = Math.exp(-ageInHours / 168); // 一周半衰期

    return {
      ...memory,
      relevanceScore: memory.relevanceScore * decayFactor
    };
  }).sort((a, b) => b.relevanceScore - a.relevanceScore);
}

private async generateMemoryChunks(
  conversationId: string,
  messages: any[],
  response: any
): Promise<MemoryChunk[]> {
  const chunks: MemoryChunk[] = [];

  // 为每条消息创建记忆块（使用智能分段）
  for (const message of messages) {
    if (message.content.length > 50) { // 只保存有意义的内容
      // 智能分段
      const segments = await this.textSegmenter.segmentText(messag

      for (const segment of segments) {
        chunks.push({
          id: `chunk_${Date.now()}_${Math.random()}`,
          content: segment.content,
          embedding: [], // 稍后填充
          relevanceScore: 1.0,
          timestamp: new Date(),
          chunkType: segment.type,
          chunkIndex: segment.index,
          tokenCount: segment.metadata.tokenCount,
          metadata: {
            conversationId,
            role: message.role,
            type: 'user_message',
```

```
              importance: segment.metadata.importance,
              startPosition: segment.metadata.startPosition,
              endPosition: segment.metadata.endPosition
            }
          });
        }
      }
    }

    // 为AI响应创建记忆块（使用智能分段）
    if (response.choices?.[0]?.message?.content) {
      const segments = await this.textSegmenter.segmentText(response

      for (const segment of segments) {
        chunks.push({
          id: `chunk_${Date.now()}_${Math.random()}`,
          content: segment.content,
          embedding: [],
          relevanceScore: 1.0,
          timestamp: new Date(),
          chunkType: segment.type,
          chunkIndex: segment.index,
          tokenCount: segment.metadata.tokenCount,
          metadata: {
            conversationId,
            role: 'assistant',
            type: 'ai_response',
            importance: segment.metadata.importance,
            startPosition: segment.metadata.startPosition,
            endPosition: segment.metadata.endPosition
          }
        });
      }
    }

    return chunks;
  }

  private async cleanupExpiredMemories(projectId: string): Promise<v
    // 删除超过30天的记忆
    const cutoffDate = new Date(Date.now() - 30 * 24 * 60 * 60 * 100
    await this.vectorDB.deleteExpiredMemories(projectId, cutoffDate)
  }
}
```

2. 智能分段与检索系统

2.1 文本分段器实现

```typescript
// text-segmenter.ts
export interface SegmentationConfig {
  maxChunkSize: number;
  overlapSize: number;
  strategy: 'semantic' | 'paragraph' | 'fixed' | 'sliding' | 'hybrid
  preserveBoundaries: boolean;
}

export interface TextSegment {
  content: string;
  index: number;
  type: string;
  metadata: {
    startPosition: number;
    endPosition: number;
    tokenCount: number;
    importance: number;
  };
}

export class TextSegmenter {
  private config: SegmentationConfig;

  constructor(config: SegmentationConfig) {
    this.config = config;
  }

  async segmentText(text: string): Promise<TextSegment[]> {
    switch (this.config.strategy) {
      case 'semantic':
        return this.semanticSegmentation(text);
      case 'paragraph':
        return this.paragraphSegmentation(text);
      case 'fixed':
        return this.fixedLengthSegmentation(text);
      case 'sliding':
        return this.slidingWindowSegmentation(text);
      case 'hybrid':
        return this.hybridSegmentation(text);
      default:
        throw new Error(`Unsupported segmentation strategy: ${this.c
    }
  }

  private async semanticSegmentation(text: string): Promise<TextSegm
    const sentences = this.splitIntoSentences(text);
    const segments: TextSegment[] = [];
    let currentSegment = '';
    let currentTokenCount = 0;
    let segmentIndex = 0;

    for (let i = 0; i < sentences.length; i++) {
```

```
      const sentence = sentences[i];
      const sentenceTokens = this.estimateTokens(sentence);

      if (currentTokenCount + sentenceTokens > this.config.maxChunkS
        // 创建当前段落
        segments.push(this.createSegment(currentSegment, segmentInde

        // 开始新段落，包含重叠内容
        const overlapContent = this.getOverlapContent(currentSegment
        currentSegment = overlapContent + sentence;
        currentTokenCount = this.estimateTokens(currentSegment);
        segmentIndex++;
      } else {
        currentSegment += (currentSegment ? ' ' : '') + sentence;
        currentTokenCount += sentenceTokens;
      }
    }

    // 添加最后一个段落
    if (currentSegment) {
      segments.push(this.createSegment(currentSegment, segmentIndex,
    }

    return segments;
  }

  private paragraphSegmentation(text: string): Promise<TextSegment[]
    const paragraphs = text.split(/\n\s*\n/);
    const segments: TextSegment[] = [];
    let segmentIndex = 0;

    for (const paragraph of paragraphs) {
      if (paragraph.trim()) {
        const tokenCount = this.estimateTokens(paragraph);

        if (tokenCount > this.config.maxChunkSize) {
          // 如果段落太长，进一步分割
          const subSegments = this.fixedLengthSegmentation(paragraph
          segments.push(...subSegments.map(seg => ({
            ...seg,
            index: segmentIndex++,
            type: 'paragraph-split'
          })));
        } else {
          segments.push(this.createSegment(paragraph, segmentIndex++
        }
      }
    }

    return Promise.resolve(segments);
  }
```

```
  private fixedLengthSegmentation(text: string): TextSegment[] {
    const segments: TextSegment[] = [];
    const words = text.split(/\s+/);
    let currentSegment = '';
    let currentTokenCount = 0;
    let segmentIndex = 0;

    for (const word of words) {
      const wordTokens = this.estimateTokens(word);

      if (currentTokenCount + wordTokens > this.config.maxChunkSize
        segments.push(this.createSegment(currentSegment, segmentInde

        // 重叠处理
        const overlapWords = this.getOverlapWords(currentSegment);
        currentSegment = overlapWords + word;
        currentTokenCount = this.estimateTokens(currentSegment);
      } else {
        currentSegment += (currentSegment ? ' ' : '') + word;
        currentTokenCount += wordTokens;
      }
    }

    if (currentSegment) {
      segments.push(this.createSegment(currentSegment, segmentIndex,
    }

    return segments;
  }

  private slidingWindowSegmentation(text: string): TextSegment[] {
    const segments: TextSegment[] = [];
    const words = text.split(/\s+/);
    const windowSize = this.config.maxChunkSize;
    const stepSize = windowSize - this.config.overlapSize;

    for (let i = 0; i < words.length; i += stepSize) {
      const windowWords = words.slice(i, i + windowSize);
      const content = windowWords.join(' ');

      if (content.trim()) {
        segments.push(this.createSegment(content, Math.floor(i / ste
      }
    }

    return segments;
  }

  private hybridSegmentation(text: string): Promise<TextSegment[]> {
    // 结合多种策略的混合分段
    const semanticSegments = this.semanticSegmentation(text);
    // 可以添加更多策略的组合逻辑
```

```
    return semanticSegments;
  }

  private splitIntoSentences(text: string): string[] {
    // 简单的句子分割，可以使用更复杂的NLP库
    return text.split(/[.!?]+/).filter(s => s.trim().length > 0);
  }

  private estimateTokens(text: string): number {
    // 简单的token估算，实际应该使用对应模型的tokenizer
    return Math.ceil(text.length / 4);
  }

  private createSegment(content: string, index: number, type: string
    return {
      content: content.trim(),
      index,
      type,
      metadata: {
        startPosition: 0, // 实际实现中应该计算真实位置
        endPosition: content.length,
        tokenCount: this.estimateTokens(content),
        importance: this.calculateImportance(content)
      }
    };
  }

  private getOverlapContent(segment: string): string {
    const words = segment.split(/\s+/);
    const overlapWords = words.slice(-this.config.overlapSize);
    return overlapWords.join(' ') + ' ';
  }

  private getOverlapWords(segment: string): string {
    const words = segment.split(/\s+/);
    const overlapWords = words.slice(-this.config.overlapSize);
    return overlapWords.join(' ') + ' ';
  }

  private calculateImportance(content: string): number {
    // 简单的重要性计算，可以基于关键词密度、长度等
    const keywordCount = (content.match(/\b(重要|关键|核心|主要)\b/g) |
    const lengthScore = Math.min(content.length / 1000, 1);
    return (keywordCount * 0.3 + lengthScore * 0.7);
  }
}
```

1.3 向量数据库接口实现

```
// vector-database.ts
export interface VectorSearchOptions {
  projectId?: string;
  sessionId?: string;
  limit?: number;
  threshold?: number;
  filters?: Record<string, any>;
}

export interface VectorSearchResult {
  id: string;
  content: string;
  embedding: number[];
  score: number;
  metadata: Record<string, any>;
}

export class VectorDatabase {
  private client: any; // Pinecone/Weaviate客户端

  constructor() {
    this.initializeClient();
  }

  private async initializeClient(): Promise<void> {
    // 初始化向量数据库客户端
    // 这里以Pinecone为例
    const { PineconeClient } = require('@pinecone-database/pinecone'
    this.client = new PineconeClient();
    await this.client.init({
      environment: process.env.PINECONE_ENVIRONMENT!,
      apiKey: process.env.PINECONE_API_KEY!
    });
  }

  async store(data: {
    content: string;
    embedding: number[];
    metadata: Record<string, any>;
  }): Promise<string> {
    const id = `vec_${Date.now()}_${Math.random().toString(36).subst

    const index = this.client.Index(process.env.PINECONE_INDEX_NAME!
    await index.upsert({
      upsertRequest: {
        vectors: [{
          id,
          values: data.embedding,
          metadata: {
            content: data.content,
            ...data.metadata
          }
```

```
        }]
      }
    });

    return id;
  }

  async search(
    queryEmbedding: number[],
    options: VectorSearchOptions = {}
  ): Promise<VectorSearchResult[]> {
    const {
      limit = 10,
      threshold = 0.7,
      filters = {}
    } = options;

    const index = this.client.Index(process.env.PINECONE_INDEX_NAME!
    const queryResponse = await index.query({
      queryRequest: {
        vector: queryEmbedding,
        topK: limit,
        includeMetadata: true,
        filter: filters
      }
    });

    return queryResponse.matches
      .filter((match: any) => match.score >= threshold)
      .map((match: any) => ({
        id: match.id,
        content: match.metadata.content,
        embedding: [], // 不返回完整向量以节省带宽
        score: match.score,
        metadata: match.metadata
      }));
  }

  async delete(id: string): Promise<void> {
    const index = this.client.Index(process.env.PINECONE_INDEX_NAME!
    await index.delete1({
      ids: [id]
    });
  }

  async batchStore(items: Array<{
    id?: string;
    content: string;
    embedding: number[];
    metadata: Record<string, any>;
  }>): Promise<string[]> {
    const vectors = items.map(item => ({
      id: item.id || `vec_${Date.now()}_${Math.random().toString(36)
```

```
      values: item.embedding,
      metadata: {
        content: item.content,
        ...item.metadata
      }
    }));

    const index = this.client.Index(process.env.PINECONE_INDEX_NAME!
    await index.upsert({
      upsertRequest: { vectors }
    });

    return vectors.map(v => v.id);
  }

  async similaritySearch(
    queryEmbedding: number[],
    options: {
      projectId: string;
      sessionId?: string;
      limit: number;
      threshold: number;
    }
  ): Promise<MemoryChunk[]> {
    const index = this.client.index(this.indexName);

    const queryRequest = {
      vector: queryEmbedding,
      topK: options.limit,
      filter: {
        projectId: { $eq: options.projectId },
        ...(options.sessionId && { sessionId: { $eq: options.session
      },
      includeMetadata: true,
      includeValues: false
    };

    const results = await index.query(queryRequest);

    return results.matches
      ?.filter(match => (match.score || 0) >= options.threshold)
      .map(match => ({
        id: match.id,
        content: match.metadata?.content as string,
        embedding: [],
        relevanceScore: match.score || 0,
        timestamp: new Date(match.metadata?.timestamp as string),
        chunkType: match.metadata?.chunkType as string,
        chunkIndex: match.metadata?.chunkIndex as number,
        tokenCount: match.metadata?.tokenCount as number,
        metadata: match.metadata as Record<string, any>
      })) || [];
  }
```

```
  async insertMemories(memories: MemoryChunk[]): Promise<void> {
    const index = this.client.index(this.indexName);

    const vectors = memories.map(memory => ({
      id: memory.id,
      values: memory.embedding,
      metadata: {
        content: memory.content,
        timestamp: memory.timestamp.toISOString(),
        ...memory.metadata
      }
    }));

    await index.upsert(vectors);
  }

  async deleteExpiredMemories(projectId: string, cutoffDate: Date):
    const index = this.client.index(this.indexName);

    // Pinecone不支持按时间范围删除，需要先查询再删除
    const expiredIds = await this.findExpiredMemoryIds(projectId, cu

    if (expiredIds.length > 0) {
      await index.deleteMany(expiredIds);
    }
  }

  private async findExpiredMemoryIds(projectId: string, cutoffDate:
    // 实现查找过期记忆ID的逻辑
    return [];
  }
}
```

1.4 嵌入服务实现

```
// embedding-service.ts
import OpenAI from 'openai';

export class EmbeddingService {
  private openai: OpenAI;
  private cache: Map<string, number[]> = new Map();

  constructor() {
    this.openai = new OpenAI({
      apiKey: process.env.OPENAI_API_KEY
    });
  }

  async generateEmbedding(text: string): Promise<number[]> {
    // 检查缓存
```

```
    const cacheKey = this.getCacheKey(text);
    if (this.cache.has(cacheKey)) {
      return this.cache.get(cacheKey)!;
    }

    try {
      const response = await this.openai.embeddings.create({
        model: 'text-embedding-3-small',
        input: text,
        encoding_format: 'float'
      });

      const embedding = response.data[0].embedding;

      // 缓存结果
      this.cache.set(cacheKey, embedding);

      return embedding;
    } catch (error) {
      console.error('生成嵌入向量失败:', error);
      throw error;
    }
  }

  async generateBatchEmbeddings(texts: string[]): Promise<number[][]
    try {
      const response = await this.openai.embeddings.create({
        model: 'text-embedding-3-small',
        input: texts,
        encoding_format: 'float'
      });

      return response.data.map(item => item.embedding);
    } catch (error) {
      console.error('批量生成嵌入向量失败:', error);
      throw error;
    }
  }

  private getCacheKey(text: string): string {
    // 简单的哈希函数用于缓存键
    let hash = 0;
    for (let i = 0; i < text.length; i++) {
      const char = text.charCodeAt(i);
      hash = ((hash << 5) - hash) + char;
      hash = hash & hash; // 转换为32位整数
    }
    return hash.toString();
  }

  // 计算余弦相似度
  cosineSimilarity(a: number[], b: number[]): number {
```

```
    if (a.length !== b.length) {
      throw new Error('向量维度不匹配');
    }

    let dotProduct = 0;
    let normA = 0;
    let normB = 0;

    for (let i = 0; i < a.length; i++) {
      dotProduct += a[i] * b[i];
      normA += a[i] * a[i];
      normB += b[i] * b[i];
    }

    return dotProduct / (Math.sqrt(normA) * Math.sqrt(normB));
  }

  async getEmbedding(text: string): Promise<number[]> {
    // 检查缓存
    const cacheKey = this.hashText(text);
    if (this.cache.has(cacheKey)) {
      return this.cache.get(cacheKey)!;
    }

    try {
      // 文本预处理
      const cleanedText = this.preprocessText(text);

      // 调用OpenAI嵌入API
      const response = await this.openai.embeddings.create({
        model: 'text-embedding-3-small',
        input: cleanedText,
        encoding_format: 'float'
      });

      const embedding = response.data[0].embedding;

      // 缓存结果
      this.cache.set(cacheKey, embedding);

      return embedding;
    } catch (error) {
      console.error('Embedding generation failed:', error);
      throw error;
    }
  }

  async getBatchEmbeddings(texts: string[]): Promise<number[][]> {
    const cleanedTexts = texts.map(text => this.preprocessText(text)

    const response = await this.openai.embeddings.create({
      model: 'text-embedding-3-small',
```

```
      input: cleanedTexts,
      encoding_format: 'float'
    });

    return response.data.map(item => item.embedding);
  }

  private preprocessText(text: string): string {
    return text
      .trim()
      .replace(/\s+/g, ' ')
      .substring(0, 8000); // 限制长度
  }

  private hashText(text: string): string {
    // 简单的哈希函数
    let hash = 0;
    for (let i = 0; i < text.length; i++) {
      const char = text.charCodeAt(i);
      hash = ((hash << 5) - hash) + char;
      hash = hash & hash; // 转换为32位整数
    }
    return hash.toString();
  }
}
```

2. 透明代理实现

2.1 代理服务器核心

```
// proxy-server.ts
import express from 'express';
import { MemoryManager } from './memory-manager';
import { WindowManager, WindowConfig } from './window-manager';
import { TokenCounter } from './token-counter';
import { CompressionEngine, CompressionConfig } from './compression-
import { PrioritySorter, PriorityWeights } from './priority-sorter';

export class ProxyServer {
  private app: express.Application;
  private memoryManager: MemoryManager;
  private windowManager: WindowManager;
  private tokenCounter: TokenCounter;
  private compressionEngine: CompressionEngine;
  private prioritySorter: PrioritySorter;

  constructor() {
    this.app = express();
    this.memoryManager = new MemoryManager();
    this.tokenCounter = new TokenCounter();
```

```
    // 默认窗口配置
    const defaultWindowConfig: WindowConfig = {
      maxContextTokens: 4000,
      reservedTokens: 500,
      systemPromptTokens: 200,
      minUserContentTokens: 100
    };

    // 默认优先级权重
    const defaultPriorityWeights: PriorityWeights = {
      temporal: 0.3,
      relevance: 0.4,
      importance: 0.2,
      userPreference: 0.1
    };

    // 默认压缩配置
    const defaultCompressionConfig: CompressionConfig = {
      enableCompression: true,
      compressionRatio: 0.7,
      summaryModel: 'gpt-3.5-turbo',
      preserveKeywords: true,
      maxCompressionLevel: 3
    };

    this.prioritySorter = new PrioritySorter(defaultPriorityWeights)
    this.compressionEngine = new CompressionEngine(this.tokenCounter
    this.windowManager = new WindowManager(
      defaultWindowConfig,
      this.tokenCounter,
      this.compressionEngine,
      this.prioritySorter
    );

    this.setupMiddleware();
    this.setupRoutes();
  }

  private setupMiddleware(): void {
    this.app.use(express.json({ limit: '10mb' }));
    this.app.use(express.urlencoded({ extended: true }));

    // CORS支持
    this.app.use((req, res, next) => {
      res.header('Access-Control-Allow-Origin', '*');
      res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DE
      res.header('Access-Control-Allow-Headers', 'Origin, X-Requeste
      next();
    });
  }
```

```
  private setupRoutes(): void {
    // 透明代理路由
    this.app.post('/v1/chat/completions', this.handleChatCompletion.
    this.app.post('/v1/completions', this.handleCompletion.bind(this

    // 记忆管理路由
    this.app.get('/v1/memory/search', this.handleMemorySearch.bind(t
    this.app.post('/v1/memory/store', this.handleMemoryStore.bind(th
    this.app.delete('/v1/memory/:id', this.handleMemoryDelete.bind(t

    // Token管理路由
    this.app.get('/v1/token/usage', this.handleTokenUsage.bind(this)
    this.app.post('/v1/token/optimize', this.handleTokenOptimize.bin

    // 配置管理路由
    this.app.get('/v1/config/segmentation', this.handleGetSegmentati
    this.app.post('/v1/config/segmentation', this.handleUpdateSegmen
    this.app.get('/v1/config/priority', this.handleGetPriorityConfig
    this.app.post('/v1/config/priority', this.handleUpdatePriorityCo
  }

  private async handleChatCompletion(req: express.Request, res: expr
    try {
      const { messages, model, max_tokens, temperature, ...otherPara
      const projectId = req.headers['x-project-id'] as string || 'de
      const sessionId = req.headers['x-session-id'] as string || 'de

      // 1. 检索相关记忆
      const relevantMemories = await this.memoryManager.retrieveRele
        projectId,
        sessionId,
        messages,
        max_tokens ? Math.floor(max_tokens * 0.3) : undefined // 为记
      );

      // 2. 使用窗口管理器优化上下文
      const systemMessages = messages.filter((msg: any) => msg.role
      const userMessages = messages.filter((msg: any) => msg.role !=

      const memoryItems = relevantMemories.map(memory => ({
        id: memory.id,
        content: memory.content,
        timestamp: memory.timestamp,
        relevanceScore: memory.relevanceScore,
        importanceScore: memory.metadata.importance || 0.5,
        userInteractionScore: memory.metadata.userPreference || 0.5,
        tokenCount: memory.tokenCount,
        metadata: memory.metadata
      }));

      const optimizedMessages = await this.windowManager.manageConte
        systemMessages,
```

```
        userMessages,
        memoryItems,
        {
          enableCompression: true,
          compressionRatio: 0.7,
          summaryModel: 'gpt-3.5-turbo',
          preserveKeywords: true,
          maxCompressionLevel: 2
        }
      );

      // 3. 调用原始LLM API
      const response = await this.callOriginalAPI('/v1/chat/completi
        messages: optimizedMessages,
        model,
        max_tokens,
        temperature,
        ...otherParams
      });

      // 4. 异步更新记忆（不阻塞响应）
      this.updateMemoryAsync(projectId, sessionId, messages, respons

      // 5. 返回响应
      res.json(response);
    } catch (error) {
      console.error('Chat completion error:', error);
      res.status(500).json({ error: 'Internal server error' });
    }
  }

  private async handleMemorySearch(req: express.Request, res: expres
    try {
      const { query, limit = 10, maxTokens } = req.query;
      const projectId = req.headers['x-project-id'] as string || 'de

      const memories = await this.memoryManager.retrieveMemory(
        query as string,
        parseInt(limit as string),
        maxTokens ? parseInt(maxTokens as string) : undefined
      );

      res.json({ memories });
    } catch (error) {
      console.error('Memory search error:', error);
      res.status(500).json({ error: 'Internal server error' });
    }
  }

  private async handleTokenUsage(req: express.Request, res: express.
    try {
      const { messages } = req.body;
```

```
      const analysis = await this.windowManager.analyzeTokenUsage(me
      res.json(analysis);
    } catch (error) {
      console.error('Token usage analysis error:', error);
      res.status(500).json({ error: 'Internal server error' });
    }
  }

  private async updateMemoryAsync(
    projectId: string,
    sessionId: string,
    messages: any[],
    response: any
  ): Promise<void> {
    try {
      await this.memoryManager.updateMemory(projectId, sessionId, me
    } catch (error) {
      console.error('Memory update error:', error);
    }
  }

  private async callOriginalAPI(endpoint: string, data: any): Promis
    // 这里实现对原始LLM API的调用
    // 可以是OpenAI、Anthropic、或其他LLM提供商
    const fetch = require('node-fetch');

    const response = await fetch(`https://api.openai.com${endpoint}`
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${process.env.OPENAI_API_KEY}`
      },
      body: JSON.stringify(data)
    });

    return await response.json();
  }

  start(port: number = 3000): void {
    this.app.listen(port, () => {
      console.log(`Supermemory proxy server running on port ${port}`
    });
  }
}
```

2.2 嵌入服务实现

```
// embedding-service.ts
import OpenAI from 'openai';

export class EmbeddingService {
```

```
  private openai: OpenAI;
  private cache: Map<string, number[]>;

  constructor() {
    this.openai = new OpenAI({
      apiKey: process.env.OPENAI_API_KEY
    });
    this.cache = new Map();
  }

  async getEmbedding(text: string): Promise<number[]> {
    // 检查缓存
    const cacheKey = this.hashText(text);
    if (this.cache.has(cacheKey)) {
      return this.cache.get(cacheKey)!;
    }

    try {
      // 文本预处理
      const cleanedText = this.preprocessText(text);

      // 调用OpenAI嵌入API
      const response = await this.openai.embeddings.create({
        model: 'text-embedding-3-small',
        input: cleanedText,
        encoding_format: 'float'
      });

      const embedding = response.data[0].embedding;

      // 缓存结果
      this.cache.set(cacheKey, embedding);

      return embedding;
    } catch (error) {
      console.error('Embedding generation failed:', error);
      throw error;
    }
  }

  async getBatchEmbeddings(texts: string[]): Promise<number[][]> {
    const cleanedTexts = texts.map(text => this.preprocessText(text)

    const response = await this.openai.embeddings.create({
      model: 'text-embedding-3-small',
      input: cleanedTexts,
      encoding_format: 'float'
    });

    return response.data.map(item => item.embedding);
  }
```

```
  private preprocessText(text: string): string {
    return text
      .trim()
      .replace(/\s+/g, ' ')
      .substring(0, 8000); // 限制长度
  }

  private hashText(text: string): string {
    // 简单的哈希函数
    let hash = 0;
    for (let i = 0; i < text.length; i++) {
      const char = text.charCodeAt(i);
      hash = ((hash << 5) - hash) + char;
      hash = hash & hash; // 转换为32位整数
    }
    return hash.toString();
  }
}
```

3. 自动Token管理系统

3.1 Token计数器实现

```
// token-counter.ts
import { encoding_for_model } from 'tiktoken';

export class TokenCounter {
  private encoders: Map<string, any> = new Map();

  async countTokens(text: string, model: string = 'gpt-4'): Promise<
    try {
      let encoder = this.encoders.get(model);
      if (!encoder) {
        encoder = encoding_for_model(model as any);
        this.encoders.set(model, encoder);
      }

      const tokens = encoder.encode(text);
      return tokens.length;
    } catch (error) {
      // 回退到估算方法
      return this.estimateTokens(text);
    }
  }

  async countMessagesTokens(messages: any[], model: string = 'gpt-4'
    let totalTokens = 0;

    for (const message of messages) {
      // 每条消息的基础开销
```

```
      totalTokens += 4; // 消息格式开销

      // 角色字段
      totalTokens += await this.countTokens(message.role, model);

      // 内容字段
      if (message.content) {
        totalTokens += await this.countTokens(message.content, model
      }

      // 其他字段
      if (message.name) {
        totalTokens += await this.countTokens(message.name, model);
      }
    }

    totalTokens += 2; // 对话结束标记
    return totalTokens;
  }

  private estimateTokens(text: string): number {
    // 不同语言的token密度估算
    const chineseChars = (text.match(/[\u4e00-\u9fff]/g) || []).leng
    const englishWords = (text.match(/[a-zA-Z]+/g) || []).length;
    const otherChars = text.length - chineseChars - englishWords;

    return Math.ceil(
      chineseChars * 0.7 + // 中文约0.7 token/字符
      englishWords * 0.25 + // 英文约0.25 token/单词
      otherChars * 0.5 // 其他字符约0.5 token/字符
    );
  }

  async batchCountTokens(texts: string[], model: string = 'gpt-4'):
    const results: number[] = [];
    for (const text of texts) {
      results.push(await this.countTokens(text, model));
    }
    return results;
  }
}
```

3.2 压缩策略引擎

```
// compression-engine.ts
export interface CompressionConfig {
  enableCompression: boolean;
  compressionRatio: number;
  summaryModel: string;
  preserveKeywords: boolean;
```

```
  maxCompressionLevel: number;
}

export class CompressionEngine {
  private tokenCounter: TokenCounter;
  private llmAdapter: LLMAdapter;

  constructor(tokenCounter: TokenCounter, llmAdapter: LLMAdapter) {
    this.tokenCounter = tokenCounter;
    this.llmAdapter = llmAdapter;
  }

  async compressContent(
    content: string,
    targetTokens: number,
    config: CompressionConfig
  ): Promise<string> {
    const currentTokens = await this.tokenCounter.countTokens(conten

    if (currentTokens <= targetTokens) {
      return content;
    }

    const compressionRatio = targetTokens / currentTokens;

    if (compressionRatio > 0.8) {
      // 轻度压缩：去除冗余
      return this.lightCompression(content, targetTokens);
    } else if (compressionRatio > 0.5) {
      // 中度压缩：提取关键信息
      return this.mediumCompression(content, targetTokens, config);
    } else {
      // 重度压缩：生成摘要
      return this.heavyCompression(content, targetTokens, config);
    }
  }

  private async lightCompression(content: string, targetTokens: numb
    // 去除多余空格、重复内容
    let compressed = content
      .replace(/\s+/g, ' ')
      .replace(/([.!?])\s*\1+/g, '$1')
      .trim();

    // 如果还是太长，截断到目标长度
    const currentTokens = await this.tokenCounter.countTokens(compre
    if (currentTokens > targetTokens) {
      const ratio = targetTokens / currentTokens;
      const targetLength = Math.floor(compressed.length * ratio);
      compressed = compressed.substring(0, targetLength) + '...';
    }
```

```
    return compressed;
  }

  private async mediumCompression(
    content: string,
    targetTokens: number,
    config: CompressionConfig
  ): Promise<string> {
    // 提取关键句子
    const sentences = content.split(/[.!?]+/).filter(s => s.trim().l
    const keywordSentences = this.extractKeywordSentences(sentences,

    let compressed = keywordSentences.join('. ') + '.';

    // 检查是否达到目标
    const currentTokens = await this.tokenCounter.countTokens(compre
    if (currentTokens > targetTokens) {
      // 进一步压缩
      return this.heavyCompression(compressed, targetTokens, config)
    }

    return compressed;
  }

  private async heavyCompression(
    content: string,
    targetTokens: number,
    config: CompressionConfig
  ): Promise<string> {
    // 使用LLM生成摘要
    const summaryPrompt = `请将以下内容压缩为约${targetTokens}个token的摘

    try {
      const response = await this.llmAdapter.callLLM(config.summaryM
        messages: [
          { role: 'user', content: summaryPrompt }
        ],
        max_tokens: targetTokens,
        temperature: 0.3
      });

      return response.choices[0].message.content;
    } catch (error) {
      console.error('Summary generation failed:', error);
      // 回退到简单截断
      return this.lightCompression(content, targetTokens);
    }
  }

  private extractKeywordSentences(sentences: string[], config: Compr
    // 简单的关键词提取逻辑
    const keywords = ['重要', '关键', '核心', '主要', '必须', '需要', '应
```

```
    return sentences.filter(sentence => {
      const hasKeyword = keywords.some(keyword => sentence.includes(
      const isLongEnough = sentence.trim().length > 10;
      return hasKeyword || isLongEnough;
    });
  }

  async batchCompress(
    contents: string[],
    targetTokens: number,
    config: CompressionConfig
  ): Promise<string[]> {
    const results: string[] = [];
    for (const content of contents) {
      results.push(await this.compressContent(content, targetTokens,
    }
    return results;
  }
}
```

3.3 优先级排序器

```
// priority-sorter.ts
export interface PriorityWeights {
  temporal: number;    // 时间权重
  relevance: number;   // 相关性权重
  importance: number;  // 重要性权重
  userPreference: number; // 用户偏好权重
}

export interface ContentItem {
  id: string;
  content: string;
  timestamp: Date;
  relevanceScore: number;
  importanceScore: number;
  userInteractionScore: number;
  tokenCount: number;
  metadata: Record<string, any>;
}

export class PrioritySorter {
  private weights: PriorityWeights;

  constructor(weights: PriorityWeights) {
    this.weights = weights;
  }

  sortByPriority(items: ContentItem[]): ContentItem[] {
    const now = new Date();
```

```
    const scoredItems = items.map(item => ({
      ...item,
      priorityScore: this.calculatePriorityScore(item, now)
    }));

    return scoredItems.sort((a, b) => b.priorityScore - a.prioritySc
  }

  selectTopItems(items: ContentItem[], maxTokens: number): ContentIt
    const sortedItems = this.sortByPriority(items);
    const selectedItems: ContentItem[] = [];
    let totalTokens = 0;

    for (const item of sortedItems) {
      if (totalTokens + item.tokenCount <= maxTokens) {
        selectedItems.push(item);
        totalTokens += item.tokenCount;
      } else {
        break;
      }
    }

    return selectedItems;
  }

  private calculatePriorityScore(item: ContentItem, now: Date): numb
    // 时间新近度分数 (0-1)
    const ageInHours = (now.getTime() - item.timestamp.getTime()) /
    const temporalScore = Math.exp(-ageInHours / 168); // 一周半衰期

    // 相关性分数 (0-1)
    const relevanceScore = Math.min(item.relevanceScore, 1);

    // 重要性分数 (0-1)
    const importanceScore = Math.min(item.importanceScore, 1);

    // 用户偏好分数 (0-1)
    const userPreferenceScore = Math.min(item.userInteractionScore,

    // 加权计算最终分数
    return (
      this.weights.temporal * temporalScore +
      this.weights.relevance * relevanceScore +
      this.weights.importance * importanceScore +
      this.weights.userPreference * userPreferenceScore
    );
  }

  updateWeights(newWeights: Partial<PriorityWeights>): void {
    this.weights = { ...this.weights, ...newWeights };
```

```
    // 确保权重总和为1
    const totalWeight = Object.values(this.weights).reduce((sum, wei
    if (totalWeight !== 1) {
      Object.keys(this.weights).forEach(key => {
        this.weights[key as keyof PriorityWeights] /= totalWeight;
      });
    }
  }

  analyzeUserPreferences(interactions: any[]): PriorityWeights {
    // 基于用户交互分析偏好
    const analysis = {
      temporal: 0.3,
      relevance: 0.4,
      importance: 0.2,
      userPreference: 0.1
    };

    // 可以基于用户行为调整权重
    return analysis;
  }
}
```

3.4 窗口管理器

```
// window-manager.ts
export interface WindowConfig {
  maxContextTokens: number;
  reservedTokens: number;
  systemPromptTokens: number;
  minUserContentTokens: number;
}

export class WindowManager {
  private config: WindowConfig;
  private tokenCounter: TokenCounter;
  private compressionEngine: CompressionEngine;
  private prioritySorter: PrioritySorter;

  constructor(
    config: WindowConfig,
    tokenCounter: TokenCounter,
    compressionEngine: CompressionEngine,
    prioritySorter: PrioritySorter
  ) {
    this.config = config;
    this.tokenCounter = tokenCounter;
    this.compressionEngine = compressionEngine;
    this.prioritySorter = prioritySorter;
  }
```

```
  async manageContextWindow(
    systemMessages: any[],
    userMessages: any[],
    memoryItems: ContentItem[],
    compressionConfig: CompressionConfig
  ): Promise<any[]> {
    // 1. 计算各部分的token使用量
    const systemTokens = await this.tokenCounter.countMessagesTokens
    const userTokens = await this.tokenCounter.countMessagesTokens(u

    // 2. 计算可用于记忆的token数量
    const availableMemoryTokens = this.config.maxContextTokens
      - this.config.reservedTokens
      - systemTokens
      - userTokens;

    if (availableMemoryTokens <= 0) {
      // 如果没有空间给记忆，只返回系统消息和用户消息
      return [...systemMessages, ...userMessages];
    }

    // 3. 选择和压缩记忆内容
    const selectedMemories = this.prioritySorter.selectTopItems(memo
    const memoryMessages = await this.processMemoryItems(selectedMem

    // 4. 构建最终的上下文
    return this.buildFinalContext(systemMessages, memoryMessages, us
  }

  private async processMemoryItems(
    memoryItems: ContentItem[],
    maxTokens: number,
    compressionConfig: CompressionConfig
  ): Promise<any[]> {
    const memoryMessages: any[] = [];
    let usedTokens = 0;

    for (const item of memoryItems) {
      const remainingTokens = maxTokens - usedTokens;

      if (remainingTokens <= 0) break;

      let content = item.content;

      // 如果内容太长，进行压缩
      if (item.tokenCount > remainingTokens) {
        content = await this.compressionEngine.compressContent(
          content,
          remainingTokens,
          compressionConfig
        );
      }
```

```
    const finalTokens = await this.tokenCounter.countTokens(conten

    if (usedTokens + finalTokens <= maxTokens) {
      memoryMessages.push({
        role: 'system',
        content: `[记忆 ${item.timestamp.toISOString()}] ${content}
        metadata: {
          ...item.metadata,
          compressed: item.tokenCount > remainingTokens,
          originalTokens: item.tokenCount,
          finalTokens
        }
      });

      usedTokens += finalTokens;
    }
  }

  return memoryMessages;
}

private buildFinalContext(
  systemMessages: any[],
  memoryMessages: any[],
  userMessages: any[]
): any[] {
  return [
    ...systemMessages,
    ...memoryMessages,
    ...userMessages
  ];
}

async analyzeTokenUsage(messages: any[]): Promise<{
  totalTokens: number;
  systemTokens: number;
  memoryTokens: number;
  userTokens: number;
  efficiency: number;
}> {
  let systemTokens = 0;
  let memoryTokens = 0;
  let userTokens = 0;

  for (const message of messages) {
    const tokens = await this.tokenCounter.countTokens(message.con

    if (message.role === 'system') {
      if (message.content.includes('[记忆')) {
        memoryTokens += tokens;
      } else {
```

```
          systemTokens += tokens;
        }
      } else {
        userTokens += tokens;
      }
    }

    const totalTokens = systemTokens + memoryTokens + userTokens;
    const efficiency = totalTokens > 0 ? (memoryTokens + userTokens)

    return {
      totalTokens,
      systemTokens,
      memoryTokens,
      userTokens,
      efficiency
    };
  }

  updateConfig(newConfig: Partial<WindowConfig>): void {
    this.config = { ...this.config, ...newConfig };
  }
}
```

4. 上下文优化器

4.1 智能上下文管理

```
// context-optimizer.ts
export class ContextOptimizer {
  private readonly MAX_CONTEXT_TOKENS = 4096;
  private readonly RESERVED_TOKENS = 1000; // 为响应预留的token

  async optimizeContext(
    currentMessages: any[],
    relevantMemories: MemoryChunk[],
    maxTokens: number
  ): Promise<any[]> {
    const availableTokens = Math.min(maxTokens, this.MAX_CONTEXT_TOK

    // 1. 计算当前消息的token数
    const currentTokens = this.estimateTokens(currentMessages);

    // 2. 如果当前消息已经超出限制，进行压缩
    if (currentTokens > availableTokens) {
      return this.compressMessages(currentMessages, availableTokens)
    }

    // 3. 添加相关记忆
    const remainingTokens = availableTokens - currentTokens;
```

```
    const memoryMessages = this.selectMemories(relevantMemories, rem

    // 4. 构建最终上下文
    return this.buildOptimizedContext(currentMessages, memoryMessage
  }

  private estimateTokens(messages: any[]): number {
    // 简单的token估算：1 token ≈ 4 字符
    return messages.reduce((total, msg) => {
      return total + Math.ceil(msg.content.length / 4);
    }, 0);
  }

  private compressMessages(messages: any[], maxTokens: number): any[
    // 保留系统消息和最近的用户消息
    const systemMessages = messages.filter(msg => msg.role === 'syst
    const userMessages = messages.filter(msg => msg.role === 'user')
    const assistantMessages = messages.filter(msg => msg.role === 'a

    let result = [...systemMessages];
    let usedTokens = this.estimateTokens(systemMessages);

    // 从最新的消息开始添加
    const recentMessages = [...userMessages, ...assistantMessages]
      .sort((a, b) => (b.timestamp || 0) - (a.timestamp || 0));

    for (const message of recentMessages) {
      const messageTokens = this.estimateTokens([message]);
      if (usedTokens + messageTokens <= maxTokens) {
        result.push(message);
        usedTokens += messageTokens;
      } else {
        break;
      }
    }

    return result.sort((a, b) => (a.timestamp || 0) - (b.timestamp |
  }

  private selectMemories(memories: MemoryChunk[], maxTokens: number)
    const memoryMessages: any[] = [];
    let usedTokens = 0;

    // 按相关性排序选择记忆
    const sortedMemories = memories.sort((a, b) => b.relevanceScore

    for (const memory of sortedMemories) {
      const memoryMessage = {
        role: memory.metadata.role || 'assistant',
        content: memory.content,
        timestamp: memory.timestamp.getTime()
      };
```

```
      const messageTokens = this.estimateTokens([memoryMessage]);
      if (usedTokens + messageTokens <= maxTokens) {
        memoryMessages.push(memoryMessage);
        usedTokens += messageTokens;
      } else {
        break;
      }
    }

    return memoryMessages;
  }

  private buildOptimizedContext(currentMessages: any[], memoryMessag
    // 1. 系统消息
    const systemMessages = currentMessages.filter(msg => msg.role ==

    // 2. 记忆消息（作为上下文）
    const contextMessages = memoryMessages.map(msg => ({
      ...msg,
      role: 'system',
      content: `[记忆] ${msg.content}`
    }));

    // 3. 当前对话消息
    const dialogMessages = currentMessages.filter(msg => msg.role !=

    return [
      ...systemMessages,
      ...contextMessages,
      ...dialogMessages
    ];
  }
}
```

4. LLM适配器

4.1 多模型支持

```
// llm-adapter.ts
import OpenAI from 'openai';
import Anthropic from '@anthropic-ai/sdk';

export class LLMAdapter {
  private openai: OpenAI;
  private anthropic: Anthropic;

  constructor() {
    this.openai = new OpenAI({
      apiKey: process.env.OPENAI_API_KEY
```

```
    });
    this.anthropic = new Anthropic({
      apiKey: process.env.ANTHROPIC_API_KEY
    });
  }

  async callLLM(model: string, params: any): Promise<any> {
    try {
      if (model.startsWith('gpt-')) {
        return await this.callOpenAI(model, params);
      } else if (model.startsWith('claude-')) {
        return await this.callAnthropic(model, params);
      } else {
        throw new Error(`Unsupported model: ${model}`);
      }
    } catch (error) {
      console.error(`LLM call failed for model ${model}:`, error);
      throw error;
    }
  }

  private async callOpenAI(model: string, params: any): Promise<any>
    const response = await this.openai.chat.completions.create({
      model,
      messages: params.messages,
      temperature: params.temperature || 0.7,
      max_tokens: params.max_tokens,
      stream: params.stream || false,
      ...params
    });

    return response;
  }

  private async callAnthropic(model: string, params: any): Promise<a
    // 转换消息格式
    const messages = this.convertMessagesForAnthropic(params.message

    const response = await this.anthropic.messages.create({
      model,
      messages,
      max_tokens: params.max_tokens || 1000,
      temperature: params.temperature || 0.7,
      ...params
    });

    // 转换响应格式以匹配OpenAI格式
    return this.convertAnthropicResponse(response);
  }

  private convertMessagesForAnthropic(messages: any[]): any[] {
    return messages
```

```
      .filter(msg => msg.role !== 'system') // Anthropic在messages中不
      .map(msg => ({
        role: msg.role === 'assistant' ? 'assistant' : 'user',
        content: msg.content
      }));
  }

  private convertAnthropicResponse(response: any): any {
    return {
      id: response.id,
      object: 'chat.completion',
      created: Math.floor(Date.now() / 1000),
      model: response.model,
      choices: [{
        index: 0,
        message: {
          role: 'assistant',
          content: response.content[0].text
        },
        finish_reason: response.stop_reason
      }],
      usage: {
        prompt_tokens: response.usage.input_tokens,
        completion_tokens: response.usage.output_tokens,
        total_tokens: response.usage.input_tokens + response.usage.o
      }
    };
  }
}
```

5. 性能优化策略

5.1 缓存策略

```
// cache-manager.ts
import Redis from 'ioredis';

export class CacheManager {
  private redis: Redis;
  private readonly EMBEDDING_CACHE_TTL = 7 * 24 * 60 * 60; // 7天
  private readonly RESPONSE_CACHE_TTL = 60 * 60; // 1小时

  constructor() {
    this.redis = new Redis({
      host: process.env.REDIS_HOST,
      port: parseInt(process.env.REDIS_PORT || '6379'),
      password: process.env.REDIS_PASSWORD
    });
  }
```

```
  async cacheEmbedding(text: string, embedding: number[]): Promise<v
    const key = `embedding:${this.hashText(text)}`;
    await this.redis.setex(key, this.EMBEDDING_CACHE_TTL, JSON.strin
  }

  async getCachedEmbedding(text: string): Promise<number[] | null> {
    const key = `embedding:${this.hashText(text)}`;
    const cached = await this.redis.get(key);
    return cached ? JSON.parse(cached) : null;
  }

  async cacheResponse(requestHash: string, response: any): Promise<v
    const key = `response:${requestHash}`;
    await this.redis.setex(key, this.RESPONSE_CACHE_TTL, JSON.string
  }

  async getCachedResponse(requestHash: string): Promise<any | null>
    const key = `response:${requestHash}`;
    const cached = await this.redis.get(key);
    return cached ? JSON.parse(cached) : null;
  }

  private hashText(text: string): string {
    // 使用crypto模块生成更好的哈希
    const crypto = require('crypto');
    return crypto.createHash('sha256').update(text).digest('hex');
  }
}
```

5.2 异步处理队列

```
// queue-manager.ts
import Bull from 'bull';
import { MemoryManager } from './memory-manager';

export class QueueManager {
  private memoryUpdateQueue: Bull.Queue;
  private embeddingQueue: Bull.Queue;
  private memoryManager: MemoryManager;

  constructor() {
    this.memoryManager = new MemoryManager();

    this.memoryUpdateQueue = new Bull('memory-update', {
      redis: {
        host: process.env.REDIS_HOST,
        port: parseInt(process.env.REDIS_PORT || '6379')
      }
    });

    this.embeddingQueue = new Bull('embedding-generation', {
```

```
      redis: {
        host: process.env.REDIS_HOST,
        port: parseInt(process.env.REDIS_PORT || '6379')
      }
    });

    this.setupProcessors();
  }

  private setupProcessors(): void {
    // 记忆更新处理器
    this.memoryUpdateQueue.process('update-memory', async (job) => {
      const { projectId, sessionId, messages, response } = job.data;
      await this.memoryManager.updateMemory(projectId, sessionId, me
    });

    // 嵌入生成处理器
    this.embeddingQueue.process('generate-embedding', async (job) =>
      const { texts, callback } = job.data;
      // 批量生成嵌入向量
      // 实现批量处理逻辑
    });
  }

  async queueMemoryUpdate(
    projectId: string,
    sessionId: string,
    messages: any[],
    response: any
  ): Promise<void> {
    await this.memoryUpdateQueue.add('update-memory', {
      projectId,
      sessionId,
      messages,
      response
    }, {
      attempts: 3,
      backoff: 'exponential',
      delay: 1000
    });
  }

  async queueEmbeddingGeneration(texts: string[]): Promise<void> {
    await this.embeddingQueue.add('generate-embedding', {
      texts
    }, {
      attempts: 2,
      backoff: 'fixed',
      delay: 500
    });
  }
}
```

6. 监控和日志

6.1 性能监控

```
// monitoring.ts
import { createPrometheusMetrics } from 'prom-client';

export class MonitoringService {
  private requestCounter: any;
  private responseTimeHistogram: any;
  private memoryRetrievalHistogram: any;
  private errorCounter: any;

  constructor() {
    this.setupMetrics();
  }

  private setupMetrics(): void {
    const promClient = require('prom-client');

    this.requestCounter = new promClient.Counter({
      name: 'supermemory_requests_total',
      help: 'Total number of requests',
      labelNames: ['method', 'endpoint', 'status']
    });

    this.responseTimeHistogram = new promClient.Histogram({
      name: 'supermemory_response_time_seconds',
      help: 'Response time in seconds',
      labelNames: ['endpoint'],
      buckets: [0.1, 0.5, 1, 2, 5, 10]
    });

    this.memoryRetrievalHistogram = new promClient.Histogram({
      name: 'supermemory_memory_retrieval_time_seconds',
      help: 'Memory retrieval time in seconds',
      buckets: [0.01, 0.05, 0.1, 0.5, 1, 2]
    });

    this.errorCounter = new promClient.Counter({
      name: 'supermemory_errors_total',
      help: 'Total number of errors',
      labelNames: ['type', 'endpoint']
    });
  }

  recordRequest(method: string, endpoint: string, status: number): v
    this.requestCounter.inc({ method, endpoint, status: status.toStr
  }

  recordResponseTime(endpoint: string, duration: number): void {
```

```
    this.responseTimeHistogram.observe({ endpoint }, duration);
  }

  recordMemoryRetrievalTime(duration: number): void {
    this.memoryRetrievalHistogram.observe(duration);
  }

  recordError(type: string, endpoint: string): void {
    this.errorCounter.inc({ type, endpoint });
  }
}
```

7. 部署和配置

7.1 环境配置

```
# .env 文件配置
# OpenAI配置
OPENAI_API_KEY=your_openai_api_key

# 向量数据库配置 (Pinecone)
PINECONE_API_KEY=your_pinecone_api_key
PINECONE_ENVIRONMENT=your_pinecone_environment
PINECONE_INDEX_NAME=supermemory-vectors

# 数据库配置
DATABASE_URL=postgresql://username:password@localhost:5432/supermemo

# Redis配置
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_PASSWORD=your_redis_password

# 服务配置
PORT=3000
NODE_ENV=production

# 监控配置
PROMETHEUS_PORT=9090
LOG_LEVEL=info
```

7.2 Docker部署

```
# Dockerfile
FROM node:18-alpine

WORKDIR /app
```

```
# 复制package文件
COPY package*.json ./

# 安装依赖
RUN npm ci --only=production

# 复制源代码
COPY . .

# 构建应用
RUN npm run build

# 暴露端口
EXPOSE 3000

# 启动应用
CMD ["npm", "start"]
```

```
# docker-compose.yml
version: '3.8'

services:
  supermemory:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DATABASE_URL=postgresql://postgres:password@db:5432/supermem
      - REDIS_HOST=redis
    depends_on:
      - db
      - redis
    volumes:
      - ./logs:/app/logs

  db:
    image: postgres:15
    environment:
      - POSTGRES_DB=supermemory
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql

  redis:
    image: redis:7-alpine
    command: redis-server --appendonly yes
    volumes:
      - redis_data:/data
```

```
  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml

volumes:
  postgres_data:
  redis_data:
```

7.3 Kubernetes部署

```
# k8s-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: supermemory
  labels:
    app: supermemory
spec:
  replicas: 3
  selector:
    matchLabels:
      app: supermemory
  template:
    metadata:
      labels:
        app: supermemory
    spec:
      containers:
      - name: supermemory
        image: supermemory:latest
        ports:
        - containerPort: 3000
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: supermemory-secrets
              key: database-url
        - name: OPENAI_API_KEY
          valueFrom:
            secretKeyRef:
              name: supermemory-secrets
              key: openai-api-key
        resources:
          requests:
            memory: "512Mi"
            cpu: "250m"
```

```
          limits:
            memory: "1Gi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /health
            port: 3000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 3000
          initialDelaySeconds: 5
          periodSeconds: 5

---
apiVersion: v1
kind: Service
metadata:
  name: supermemory-service
spec:
  selector:
    app: supermemory
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: LoadBalancer
```

7.4 性能调优配置

```
// config/performance.ts
export const PerformanceConfig = {
  // 内存管理
  memory: {
    maxMemoryChunks: 10000,
    chunkCleanupInterval: 3600000, // 1小时
    maxCacheSize: 1000,
    cacheCleanupInterval: 1800000 // 30分钟
  },

  // 向量检索
  vectorSearch: {
    defaultTopK: 20,
    maxTopK: 100,
    similarityThreshold: 0.7,
    batchSize: 50
  },
```

```
  // Token管理
  tokenManagement: {
    defaultMaxTokens: 4000,
    reservedTokens: 500,
    compressionThreshold: 0.8,
    maxCompressionLevel: 3
  },

  // 并发控制
  concurrency: {
    maxConcurrentRequests: 100,
    queueTimeout: 30000,
    rateLimitPerMinute: 1000
  },

  // 缓存策略
  cache: {
    embeddingCacheTTL: 7 * 24 * 60 * 60, // 7天
    responseCacheTTL: 60 * 60, // 1小时
    memoryCacheTTL: 24 * 60 * 60 // 24小时
  }
};
```

7.5 监控和告警

```
# prometheus.yml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'supermemory'
    static_configs:
      - targets: ['supermemory:3000']
    metrics_path: '/metrics'
    scrape_interval: 10s

rule_files:
  - "alert_rules.yml"

alerting:
  alertmanagers:
    - static_configs:
        - targets:
          - alertmanager:9093
```

```
# alert_rules.yml
groups:
- name: supermemory_alerts
  rules:
```

```
  - alert: HighErrorRate
    expr: rate(supermemory_errors_total[5m]) > 0.1
    for: 2m
    labels:
      severity: warning
    annotations:
      summary: "High error rate detected"
      description: "Error rate is {{ $value }} errors per second"

  - alert: HighResponseTime
    expr: histogram_quantile(0.95, rate(supermemory_response_time_se
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "High response time detected"
      description: "95th percentile response time is {{ $value }} se

  - alert: MemoryUsageHigh
    expr: process_resident_memory_bytes / 1024 / 1024 > 1000
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "High memory usage"
      description: "Memory usage is {{ $value }}MB"
```

8. 使用示例

8.1 基本使用

```
// 启动Supermemory代理服务器
import { ProxyServer } from './proxy-server';

const server = new ProxyServer();
server.start(3000);

// 客户端使用示例
const response = await fetch('http://localhost:3000/v1/chat/completi
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'X-Project-Id': 'my-project',
    'X-Session-Id': 'session-123'
  },
  body: JSON.stringify({
    model: 'gpt-4',
    messages: [
      { role: 'user', content: '请帮我总结一下我们之前讨论的内容' }
    ],
```

```
    max_tokens: 1000
  })
});

const result = await response.json();
console.log(result);
```

8.2 记忆检索示例

```
// 搜索相关记忆
const memoryResponse = await fetch('http://localhost:3000/v1/memory/
  headers: {
    'X-Project-Id': 'my-project'
  }
});

const memories = await memoryResponse.json();
console.log('相关记忆:', memories);
```

8.3 Token使用分析

```
// 分析Token使用情况
const tokenAnalysis = await fetch('http://localhost:3000/v1/token/us
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    messages: [
      { role: 'system', content: '你是一个AI助手' },
      { role: 'user', content: '请介绍一下人工智能的发展历史' }
    ]
  })
});

const analysis = await tokenAnalysis.json();
console.log('Token使用分析:', analysis);
```

这份技术实现指南提供了Supermemory系统的完整实现方案，包括智能分段与检索系统、自动Token管理系统、透明代理机制、记忆管理、多模型支持、部署配置等关键技术。通过这些代码示例和架构设计，您可以构建一个功能完整、性能优化的AI记忆增强系统。