RAG_Chunking_Techniques

RAG分塊技術完整指南

什麼是分塊？

分塊（Chunking） 是將大型文檔分解成較小、可管理片段的過程。就像把一整本厚重的百科全書撕成單獨的頁面，這樣你就能更容易地找到特定的信息。

為什麼分塊很重要？

向量資料庫限制：大多數嵌入模型有輸入長度限制

檢索精確度：較小的塊更容易精確匹配查詢

計算效率：處理小塊比處理整個文檔更快

記憶體管理：避免在生成時超載context window

1. 固定大小分塊（Fixed-Size Chunking）

比喻

想像你在切一條長麵包。你用尺子測量，每次都切出完全相同厚度的片。不管麵包的質地或內容如何變化，每一片都是固定的大小。

特點

最簡單的分塊方法

每個塊都有相同的字符數或token數

可能會在句子中間截斷

Python實現

python

```
def fixed_size_chunking(text, chunk_size=1000, overlap=200):
    """
    固定大小分塊

    Args:
```

```
    text: 輸入文本
    chunk_size: 每個塊的大小（字符數）
    overlap: 塊之間的重疊部分
    """
    chunks = []
    start = 0

    while start < len(text):
        # 計算結束位置
        end = start + chunk_size

        # 提取塊
        chunk = text
        chunks.append(chunk)

        # 移動到下一個位置（考慮重疊）
        start += chunk_size - overlap

        # 避免無限循環
        if start >= len(text):
            break

    return chunks
```

使用範例

```
text = """人工智能是21世紀最重要的技術革命之一。它正在改變我們的工作方式、
生活方式和思考方式。從自動駕駛汽車到智能語音助手，AI技術無處不在。
機器學習和深度學習是AI的核心技術..."""

chunks = fixed_size_chunking(text, chunk_size=50, overlap=10)
for i, chunk in enumerate(chunks):
```

```
    print(f"塊 {i+1}: {chunk}")
```

優缺點

優點：實現簡單、處理速度快 缺點：可能破壞語義連貫性、在句子中間截斷

2. 遞歸字符文本分割（Recursive Character Text Splitting）

比喻

這就像是一個聰明的廚師在切蔬菜。首先嘗試在自然的分隔點（像洋蔥的層）切開，如果還是太大，就在下一個最好的地方切，依此類推，直到每一塊都是合適的大小。

特點

嘗試在自然邊界分割（段落、句子、詞語）

遞歸地應用分割規則

保持語義完整性

Python實現

python

```
import re

class RecursiveCharacterTextSplitter:
    def __init__(self, chunk_size=1000, chunk_overlap=200):
        self.chunk_size = chunk_size
        self.chunk_overlap = chunk_overlap
        # 分隔符優先順序：段落 -> 句子 -> 詞語 -> 字符
        self.separators = ['\n\n', '\n', '。', '！', '？', '，', '']

    def split_text(self, text):
        """遞歸分割文本"""
        return self._split_text_recursive(text, self.separators)

    def _split_text_recursive(self, text, separators):
        """遞歸分割邏輯"""
        if len(text) <= self.chunk_size:
```

```
    return [text] if text else []

# 嘗試當前分隔符
separator = separators[0] if separators else ''

if separator:
    splits = text.split(separator)
else:
    splits = list(text) # 字符級分割

# 重新組合分割後的部分
chunks = []
current_chunk = ""

for split in splits:
    if separator and split:
        split = split + separator

    # 檢查添加這個分割是否會超過大小限制
    if len(current_chunk + split) <= self.chunk_size:
        current_chunk += split
    else:
        # 保存當前塊
        if current_chunk:
            chunks.append(current_chunk)

        # 如果單個分割還是太大，遞歸處理
        if len(split) > self.chunk_size:
            sub_chunks = self._split_text_recursive(split, separators[1:])
            chunks.extend(sub_chunks)
        else:
```

```
                current_chunk = split

        # 添加最後一個塊
        if current_chunk:
            chunks.append(current_chunk)

        return self._add_overlap(chunks)

    def _add_overlap(self, chunks):
        """添加重疊部分"""
        if not chunks or len(chunks) == 1:
            return chunks

        overlapped_chunks = [chunks[0]]

        for i in range(1, len(chunks)):
            # 從前一個塊的末尾取overlap長度的內容
            prev_chunk = chunks
            current_chunk = chunks

            if len(prev_chunk) > self.chunk_overlap:
                overlap_text = prev_chunk
                combined_chunk = overlap_text + current_chunk
            else:
                combined_chunk = current_chunk

            overlapped_chunks.append(combined_chunk)

        return overlapped_chunks
```

使用範例

```
splitter = RecursiveCharacterTextSplitter(chunk_size=200, chunk_overlap=50)
```

text = """

人工智能的發展歷程可以追溯到1950年代。那時，科學家們開始思考機器是否能夠思考。

Alan Turing提出了著名的Turing測試，這成為了判斷機器智能的標準之一。

在接下來的幾十年裡，AI經歷了多次興衰。1980年代的專家系統，1990年代的機器學習，

以及2010年代的深度學習革命，每一個階段都推動了技術的進步。

今天，我們看到AI在各個領域的應用：醫療診斷、自動駕駛、自然語言處理等等。

"""

chunks = splitter.split_text(text)

for i, chunk in enumerate(chunks):

    print(f"=== 塊 {i+1} ===")

    print(chunk)

    print(f"長度: {len(chunk)}")

    print()

3. 語義分塊（Semantic Chunking）

比喻

這就像一個圖書館管理員根據書籍的主題和內容來組織書架，而不是僅僅按照書的厚度。相關的概念被放在一起，即使它們的"大小"不同。

特點

基於內容的語義相似性進行分塊

使用嵌入模型計算句子間的相似性

在語義邊界處分割

Python實現

python

import numpy as np

from sentence_transformers import SentenceTransformer

```
from sklearn.metrics.pairwise import cosine_similarity
import re

class SemanticChunker:
    def __init__(self, model_name='all-MiniLM-L6-v2', similarity_threshold=0.7):
        self.model = SentenceTransformer(model_name)
        self.similarity_threshold = similarity_threshold

    def split_into_sentences(self, text):
        """將文本分割成句子"""
        # 使用正則表達式分割中文句子
        sentences = re.split(r'[。！？\n]+', text)
        return [s.strip() for s in sentences if s.strip()]

    def get_embeddings(self, sentences):
        """獲取句子嵌入"""
        return self.model.encode(sentences)

    def find_semantic_breaks(self, embeddings):
        """找到語義分界點"""
        breaks = []

        for i in range(len(embeddings) - 1):
            # 計算相鄰句子的相似性
            similarity = cosine_similarity(
                embeddings[i].reshape(1, -1),
                embeddings[i + 1].reshape(1, -1)

            # 如果相似性低於閾值，這是一個分界點
            if similarity < self.similarity_threshold:
                breaks.append(i + 1)
```

```
        return breaks

    def chunk_by_semantic_breaks(self, sentences, breaks):
        """根據語義分界點創建塊"""
        chunks = []
        start = 0

        for break_point in breaks + [len(sentences)]:
            chunk_sentences = sentences
            chunk_text = '。'.join(chunk_sentences) + '。'
            chunks.append(chunk_text)
            start = break_point

        return chunks

    def semantic_chunking(self, text):
        """語義分塊主函數"""
        # 1. 分割成句子
        sentences = self.split_into_sentences(text)

        # 2. 獲取嵌入
        embeddings = self.get_embeddings(sentences)

        # 3. 找到語義分界點
        breaks = self.find_semantic_breaks(embeddings)

        # 4. 創建塊
        chunks = self.chunk_by_semantic_breaks(sentences, breaks)

        return chunks
```

使用範例

```
chunker = SemanticChunker(similarity_threshold=0.6)

text = """
機器學習是人工智能的一個分支。它使電腦能夠從數據中學習和改進。
監督學習使用標記的數據來訓練模型。無監督學習則從未標記的數據中發現模式。

深度學習是機器學習的一個子集。它使用神經網絡來模擬人腦的學習過程。
卷積神經網絡特別適合圖像識別任務。遞歸神經網絡則擅長處理序列數據。

自然語言處理是AI的另一個重要領域。它使機器能夠理解和生成人類語言。
Transformer架構徹底改變了NLP領域。BERT和GPT是基於transformer的著名模型。
"""

semantic_chunks = chunker.semantic_chunking(text)
for i, chunk in enumerate(semantic_chunks):
    print(f"=== 語義塊 {i+1} ===")
    print(chunk)
    print()
```

4. 重疊窗口分塊（Sliding Window Chunking）

比喻

想像你在拍攝一個長長的風景照。你不是拍一張全景，而是拍攝一系列重疊的照片，每張照片都與前一張有一部分重疊。這樣，即使某個重要的景物正好在兩張照片的邊界上，它也會完整地出現在其中一張照片中。

特點

塊之間有重疊區域

確保重要信息不會在邊界處丟失

提高檢索召回率

Python實現

python

```
class SlidingWindowChunker:
```

```
def __init__(self, window_size=1000, step_size=800):
    self.window_size = window_size
    self.step_size = step_size
    self.overlap_size = window_size - step_size

def sliding_window_chunk(self, text):
    """滑動窗口分塊"""
    chunks = []
    start = 0

    while start < len(text):
        end = min(start + self.window_size, len(text))
        chunk = text

        # 嘗試在單詞邊界結束（對於中文，在標點符號處）
        if end < len(text):
            # 向後查找最近的句號或其他標點
            punctuation_pos = self._find_nearest_punctuation(chunk)
            if punctuation_pos > len(chunk) * 0.8: # 如果標點在塊的後80%
                chunk = chunk[:punctuation_pos + 1]

        chunks.append({
            'text': chunk,
            'start_pos': start,
            'end_pos': start + len(chunk),
            'overlap_with_previous': min(self.overlap_size, start) if start > 0 else 0
        })

        start += self.step_size

        if end >= len(text):
```

```
            break

        return chunks

    def _find_nearest_punctuation(self, text):
        """找到最近的標點符號位置"""
        punctuation_marks = ['。', '！', '？', '\n', '，', '；']
        last_punct_pos = -1

        for i in range(len(text) - 1, -1, -1):
            if text[i] in punctuation_marks:
                last_punct_pos = i
                break

        return last_punct_pos if last_punct_pos != -1 else len(text) - 1
```

視覺化重疊示例

```
def visualize_overlapping_chunks(chunks):
    """視覺化重疊塊"""
    print("重疊窗口視覺化：")
    print("=" * 60)

    for i, chunk_info in enumerate(chunks):
        print(f"塊 {i+1}:")
        print(f" 位置: {chunk_info['start_pos']}-{chunk_info['end_pos']}")
        print(f" 重疊: {chunk_info['overlap_with_previous']} 字符")
        print(f" 內容預覽: {chunk_info['text'][:50]}...")

        # 顯示重疊區域
        if i > 0:
            overlap_start = chunk_info
```

```
        overlap_end = overlap_start + chunk_info

        print(f"  重  疊  區  域  ({overlap_start}-{overlap_end}):  {chunk_info['text'][:chunk_info['overlap_with_previous']]}")


    print()
```

使用範例

chunker = SlidingWindowChunker(window_size=100, step_size=70)

text = """自然語言處理技術的發展經歷了多個階段。早期的規則基礎方法依賴於手工編寫的語法規則。

統計方法的引入帶來了新的突破，使系統能夠從大量數據中學習語言模式。

深度學習的興起徹底改變了NLP領域，特別是transformer架構的出現。

現在的大型語言模型展現出驚人的語言理解和生成能力。"""

chunks = chunker.sliding_window_chunk(text)

visualize_overlapping_chunks(chunks)

5. 基於句子的分塊（Sentence-Based Chunking）

比喻

這就像是在整理一本詩集。你不會在詩句中間分頁，而是確保每一頁都包含完整的詩句，這樣讀者就能理解完整的思想。

特點

在句子邊界分割，保持語義完整

結合多個句子直到達到大小限制

避免截斷重要的語義單位

Python實現

python

```
import jieba
import re

class SentenceBasedChunker:
```

```
def __init__(self, max_chunk_size=1000, min_chunk_size=100):
    self.max_chunk_size = max_chunk_size
    self.min_chunk_size = min_chunk_size

def split_into_sentences(self, text):
    """將文本分割成句子"""
    # 處理中文句子分割
    sentence_pattern = r'[。！？\n]+'
    sentences = re.split(sentence_pattern, text)

    # 清理和過濾空句子
    sentences = [s.strip() for s in sentences if s.strip()]
    return sentences

def sentence_based_chunking(self, text):
    """基於句子的分塊"""
    sentences = self.split_into_sentences(text)
    chunks = []
    current_chunk = ""
    current_sentences = []

    for sentence in sentences:
        # 檢查添加這個句子是否會超過大小限制
        potential_chunk = current_chunk + sentence + "。"

        if len(potential_chunk) <= self.max_chunk_size:
            current_chunk = potential_chunk
            current_sentences.append(sentence)
        else:
            # 保存當前塊（如果不為空且達到最小大小）
            if current_chunk and len(current_chunk) >= self.min_chunk_size:
```

```
                chunks.append({
                    'text': current_chunk,
                    'sentence_count': len(current_sentences),
                    'char_count': len(current_chunk)
                })

                # 開始新塊
                current_chunk = sentence + "。"
                current_sentences =

        # 添加最後一個塊
        if current_chunk and len(current_chunk) >= self.min_chunk_size:
            chunks.append({
                'text': current_chunk,
                'sentence_count': len(current_sentences),
                'char_count': len(current_chunk)
            })

        return chunks
```

進階：智能句子合併

```
class SmartSentenceChunker(SentenceBasedChunker):
    def __init__(self, max_chunk_size=1000, min_chunk_size=100, similarity_threshold=0.8):
        super().__init__(max_chunk_size, min_chunk_size)
        self.similarity_threshold = similarity_threshold
        try:
            from sentence_transformers import SentenceTransformer
            self.model = SentenceTransformer('all-MiniLM-L6-v2')
            self.use_semantic = True
        except ImportError:
```

```
            self.use_semantic = False
            print("警告: 未安裝sentence-transformers，將使用基本分塊")

    def smart_sentence_chunking(self, text):
        """智能句子分塊，考慮語義相似性"""
        sentences = self.split_into_sentences(text)

        if not self.use_semantic:
            return self.sentence_based_chunking(text)

        # 獲取句子嵌入
        embeddings = self.model.encode(sentences)

        chunks = []
        current_sentences = [sentences[0]]
        current_embeddings = [embeddings[0]]

        for i in range(1, len(sentences)):
            sentence = sentences
            embedding = embeddings

            # 計算與當前塊的平均相似性
            current_avg_embedding = np.mean(current_embeddings, axis=0)
            similarity = cosine_similarity(
                embedding.reshape(1, -1),
                current_avg_embedding.reshape(1, -1)

            # 檢查是否可以添加到當前塊
            potential_text = "。".join(current_sentences + [sentence]) + "。"

            if (len(potential_text) <= self.max_chunk_size and
                similarity >= self.similarity_threshold):
```

```
            current_sentences.append(sentence)
            current_embeddings.append(embedding)
        else:
            # 保存當前塊
            chunk_text = "。".join(current_sentences) + "。"
            chunks.append({
                'text': chunk_text,
                'sentence_count': len(current_sentences),
                'avg_similarity': np.mean([
                    cosine_similarity(
                        current_embeddings[j].reshape(1, -1),
                        current_avg_embedding.reshape(1, -1)
                    )[0][0] for j in range(len(current_embeddings))
                ])
            })

            # 開始新塊
            current_sentences =
            current_embeddings =

    # 添加最後一個塊
    if current_sentences:
        chunk_text = "。".join(current_sentences) + "。"
        chunks.append({
            'text': chunk_text,
            'sentence_count': len(current_sentences),
            'avg_similarity': 1.0 if len(current_sentences) == 1 else np.mean([
                cosine_similarity(
                    current_embeddings[j].reshape(1, -1),
                    np.mean(current_embeddings, axis=0).reshape(1, -1)
```

```
                )[0][0] for j in range(len(current_embeddings))
            ])
        })

    return chunks
```

使用範例

```
chunker = SmartSentenceChunker(max_chunk_size=300, similarity_threshold=0.7)
text = """
機器學習算法可以分為幾大類。監督學習使用標記的訓練數據。
常見的監督學習算法包括決策樹、隨機森林和支持向量機。

無監督學習不需要標記數據。聚類算法是無監督學習的一個例子。
K-means和層次聚類是兩種流行的聚類方法。

強化學習通過試錯來學習最優策略。智能體在環境中採取行動並接收獎勵。
Q-learning和policy gradient是強化學習的核心算法。

深度學習使用多層神經網絡。反向傳播算法用於訓練這些網絡。
dropout和batch normalization是防止過擬合的技術。
"""

chunks = chunker.smart_sentence_chunking(text)
for i, chunk in enumerate(chunks):
    print(f"=== 智能塊 {i+1} ===")
    print(f"句子數: {chunk['sentence_count']}")
    print(f"平均相似性: {chunk['avg_similarity']:.3f}")
    print(f"內容: {chunk['text']}")
    print()
```

6. 文檔結構感知分塊（Document-Aware Chunking）

比喻

這就像一個經驗豐富的編輯在處理一本書。他們理解章節、段落和小節的結構，不會在標題中間分頁，也不會把章節標題與其內容分開。

特點

識別文檔結構（標題、段落、列表等）

保持結構完整性

根據內容類型調整分塊策略

Python實現

python

```
import re
from dataclasses import dataclass
from typing import List, Dict, Any

@dataclass
class DocumentElement:
    """文檔元素"""
    type: str # 'title', 'paragraph', 'list', 'code', etc.
    content: str
    level: int = 0 # 標題級別
    metadata: Dict[str, Any] = None

class DocumentAwareChunker:
    def __init__(self, max_chunk_size=1500, preserve_structure=True):
        self.max_chunk_size = max_chunk_size
        self.preserve_structure = preserve_structure

    def parse_document_structure(self, text):
        """解析文檔結構"""
        elements = []
        lines = text.split('\n')
```

```
    for line in lines:
        line = line.strip()
        if not line:
            continue

        element = self._classify_line(line)
        elements.append(element)

    return elements

def _classify_line(self, line):
    """分類文本行"""
    # 檢查是否為標題（Markdown格式）
    if line.startswith('#'):
        level = len(line) - len(line.lstrip('#'))
        content = line.lstrip('#').strip()
        return DocumentElement('title', content, level)

    # 檢查是否為列表項
    if re.match(r'^\s*[-+]\s+', line) or re.match(r'^\s*\d+.\s+', line):
        return DocumentElement('list_item', line)

    # 檢查是否為代碼塊
    if line.startswith('```') or line.startswith('    '):
        return DocumentElement('code', line)

    # 默認為段落
    return DocumentElement('paragraph', line)

def structure_aware_chunking(self, text):
    """結構感知分塊"""
```

```
elements = self.parse_document_structure(text)
chunks = []
current_chunk = []
current_size = 0

i = 0
while i < len(elements):
    element = elements

    # 對於標題，嘗試將其與後續內容組合
    if element.type == 'title':
        chunk_group = self._group_title_with_content(elements, i)
        chunk_text = self._elements_to_text(chunk_group)

        # 如果組合後太大，分別處理
        if len(chunk_text) > self.max_chunk_size:
            # 先保存當前塊
            if current_chunk:
                chunks.append(self._create_chunk(current_chunk))
                current_chunk = []
                current_size = 0

            # 處理大的標題組
            sub_chunks = self._split_large_group(chunk_group)
            chunks.extend(sub_chunks)
        else:
            # 檢查是否可以添加到當前塊
            if current_size + len(chunk_text) <= self.max_chunk_size:
                current_chunk.extend(chunk_group)
                current_size += len(chunk_text)
```

```
            else:
                # 保存當前塊，開始新塊
                if current_chunk:
                    chunks.append(self._create_chunk(current_chunk))
                current_chunk = chunk_group
                current_size = len(chunk_text)

        i += len(chunk_group)
    else:
        # 處理非標題元素
        element_size = len(element.content)

        if current_size + element_size <= self.max_chunk_size:
            current_chunk.append(element)
            current_size += element_size
        else:
            # 保存當前塊
            if current_chunk:
                chunks.append(self._create_chunk(current_chunk))

            # 如果單個元素太大，分割它
            if element_size > self.max_chunk_size:
                sub_chunks = self._split_large_element(element)
                chunks.extend(sub_chunks)
                current_chunk = []
                current_size = 0
            else:
                current_chunk =
                current_size = element_size
```

```
            i += 1

        # 添加最後一個塊
        if current_chunk:
            chunks.append(self._create_chunk(current_chunk))

        return chunks

    def _group_title_with_content(self, elements, title_index):
        """將標題與其內容組合"""
        group = [elements[title_index]]
        title_level = elements[title_index].level

        i = title_index + 1
        while i < len(elements):
            element = elements

            # 如果遇到同級或更高級的標題，停止
            if element.type == 'title' and element.level <= title_level:
                break

            group.append(element)
            i += 1

        return group

    def _elements_to_text(self, elements):
        """將元素列表轉換為文本"""
        text_parts = []
        for element in elements:
            if element.type == 'title':
                text_parts.append('#' * element.level + ' ' + element.content)
```

```
            else:
                text_parts.append(element.content)
        return '\n'.join(text_parts)

    def _create_chunk(self, elements):
        """創建塊對象"""
        text = self._elements_to_text(elements)
        return {
            'text': text,
            'elements': elements,
            'char_count': len(text),
            'element_count': len(elements),
            'structure_info': self._analyze_structure(elements)
        }

    def _analyze_structure(self, elements):
        """分析塊的結構信息"""
        structure = {
            'has_title': any(e.type == 'title' for e in elements),
            'title_levels': [e.level for e in elements if e.type == 'title'],
            'element_types': [e.type for e in elements],
            'has_code': any(e.type == 'code' for e in elements),
            'has_lists': any(e.type == 'list_item' for e in elements)
        }
        return structure

    def _split_large_element(self, element):
        """分割過大的單個元素"""
        # 對於過大的段落，使用固定大小分割
        if element.type == 'paragraph':
```

```
        chunks = []
        text = element.content
        start = 0

        while start < len(text):
            end = start + self.max_chunk_size
            chunk_text = text

            # 嘗試在詞邊界結束
            if end < len(text):
                last_space = chunk_text.rfind(' ')
                last_punct = max(chunk_text.rfind('。'), chunk_text.rfind('，'))
                boundary = max(last_space, last_punct)

                if boundary > len(chunk_text) * 0.8:
                    chunk_text = chunk_text[:boundary + 1]

            chunks.append({
                'text': chunk_text,
                'elements': [DocumentElement('paragraph', chunk_text)],
                'char_count': len(chunk_text),
                'element_count': 1,
                'is_split_element': True
            })

            start += len(chunk_text)

        return chunks

    # 對於其他類型，保持原樣
    return [self._create_chunk([element])]
```

```
    def _split_large_group(self, group):
        """分割過大的標題組"""
        # 如果組太大，在子標題處分割
        chunks = []
        current_group = []
        current_size = 0

        for element in group:
            element_size = len(element.content)

            if current_size + element_size <= self.max_chunk_size:
                current_group.append(element)
                current_size += element_size
            else:
                if current_group:
                    chunks.append(self._create_chunk(current_group))
                current_group =
                current_size = element_size

        if current_group:
            chunks.append(self._create_chunk(current_group))

        return chunks
```

使用範例

```
text = """
```

機器學習基礎

什麼是機器學習？

機器學習是人工智能的一個分支，它使電腦能夠在沒有明確編程的情況下學習和改進。

主要類型

機器學習主要分為三種類型：

- 監督學習：使用標記數據進行訓練
- 無監督學習：從未標記數據中發現模式
- 強化學習：通過與環境互動學習

常用算法

監督學習算法

決策樹是一種直觀的監督學習算法。它通過一系列是/否問題來做決策。

隨機森林組合多個決策樹，提高預測準確性。

無監督學習算法

K-means聚類將數據點分組到k個簇中。

主成分分析(PCA)用於降維和數據可視化。

深度學習

深度學習是機器學習的一個子集，使用具有多個隱藏層的神經網絡。

"""

chunker = DocumentAwareChunker(max_chunk_size=300)

chunks = chunker.structure_aware_chunking(text)

for i, chunk in enumerate(chunks):

    print(f"=== 結構感知塊 {i+1} ===")

    print(f"字符數: {chunk['char_count']}")

    print(f"元素數: {chunk['element_count']}")

    print(f"結構信息: {chunk['structure_info']}")

    print(f"內容:\n{chunk['text']}")

```
    print("-" * 40)
```

7. 分層分塊（Hierarchical Chunking）

比喻

這就像是俄羅斯套娃。最大的娃娃代表整個文檔，裡面的娃娃代表章節，更小的娃娃代表段落，最小的娃娃代表句子。每一層都包含更細粒度的信息。

特點

創建多層次的塊層次結構

支持不同粒度的檢索

可以在不同層次間導航

Python實現

python

```
from dataclasses import dataclass, field
from typing import List, Optional
import uuid

@dataclass
class HierarchicalChunk:
    """分層塊"""
    id: str
    content: str
    level: int
    parent_id: Optional[str] = None
    children_ids: List[str] = field(default_factory=list)
    metadata: Dict[str, Any] = field(default_factory=dict)

    def add_child(self, child_id: str):
        """添加子塊"""
        if child_id not in self.children_ids:
            self.children_ids.append(child_id)
```

```
class HierarchicalChunker:
    def __init__(self, chunk_sizes=[3000, 1000, 300]):
        """
        初始化分層分塊器

        Args:
            chunk_sizes: 每層的最大塊大小，從粗到細
        """
        self.chunk_sizes = chunk_sizes
        self.chunks = {} # chunk_id -> HierarchicalChunk

    def hierarchical_chunking(self, text, document_id=None):
        """執行分層分塊"""
        if document_id is None:
            document_id = str(uuid.uuid4())

        # 創建根節點（整個文檔）
        root_chunk = HierarchicalChunk(
            id=document_id,
            content=text,
            level=0,
            metadata={'type': 'document', 'size': len(text)}
        )
        self.chunks[document_id] = root_chunk

        # 遞歸創建子層次
        self._create_sublevel(text, document_id, 1)

        return self.chunks
```

```
def _create_sublevel(self, text, parent_id, level):
    """創建子層次"""
    if level >= len(self.chunk_sizes):
        return

    chunk_size = self.chunk_sizes

    # 根據層次選擇分割策略
    if level == 1: # 章節級別
        sub_chunks = self._split_by_sections(text, chunk_size)
    elif level == 2: # 段落級別
        sub_chunks = self._split_by_paragraphs(text, chunk_size)
    else: # 句子級別
        sub_chunks = self._split_by_sentences(text, chunk_size)

    # 創建子塊
    for i, chunk_text in enumerate(sub_chunks):
        chunk_id = f"{parent_id}_L{level}_{i}"

        chunk = HierarchicalChunk(
            id=chunk_id,
            content=chunk_text,
            level=level,
            parent_id=parent_id,
            metadata={
                'type': f'level_{level}',
                'size': len(chunk_text),
                'position': i
            }
        )
```

```
            self.chunks[chunk_id] = chunk
            self.chunks[parent_id].add_child(chunk_id)

            # 遞歸創建更深層次
            if len(chunk_text) > chunk_size and level < len(self.chunk_sizes) - 1:
                self._create_sublevel(chunk_text, chunk_id, level + 1)

    def _split_by_sections(self, text, max_size):
        """按章節分割"""
        # 簡單的章節檢測（基於空行）
        sections = re.split(r'\n\s*\n', text)
        chunks = []
        current_chunk = ""

        for section in sections:
            if len(current_chunk + section) <= max_size:
                current_chunk += section + "\n\n"
            else:
                if current_chunk:
                    chunks.append(current_chunk.strip())
                current_chunk = section + "\n\n"

        if current_chunk:
            chunks.append(current_chunk.strip())

        return chunks

    def _split_by_paragraphs(self, text, max_size):
        """按段落分割"""
        paragraphs = [p.strip() for p in text.split('\n') if p.strip()]
        chunks = []
```

```
    current_chunk = ""

    for paragraph in paragraphs:
        if len(current_chunk + paragraph + '\n') <= max_size:
            current_chunk += paragraph + '\n'
        else:
            if current_chunk:
                chunks.append(current_chunk.strip())
            current_chunk = paragraph + '\n'

    if current_chunk:
        chunks.append(current_chunk.strip())

    return chunks

def _split_by_sentences(self, text, max_size):
    """按句子分割"""
    sentences = re.split(r'[。！？]', text)
    sentences = [s.strip() + '。' for s in sentences if s.strip()]

    chunks = []
    current_chunk = ""

    for sentence in sentences:
        if len(current_chunk + sentence) <= max_size:
            current_chunk += sentence
        else:
            if current_chunk:
                chunks.append(current_chunk)
            current_chunk = sentence

    if current_chunk:
```

```
            chunks.append(current_chunk)

        return chunks

    def get_chunk_hierarchy(self, chunk_id):
        """獲取塊的層次結構"""
        if chunk_id not in self.chunks:
            return None

        chunk = self.chunks
        hierarchy = {
            'current': chunk,
            'parent': self.chunks.get(chunk.parent_id) if chunk.parent_id else None,
            'children': [self.chunks[child_id] for child_id in chunk.children_ids],
            'siblings': []
        }

        # 獲取兄弟節點
        if chunk.parent_id:
            parent = self.chunks
            hierarchy['siblings'] = [
                self.chunks[sibling_id] for sibling_id in parent.children_ids
                if sibling_id != chunk_id
            ]

        return hierarchy

    def visualize_hierarchy(self):
        """視覺化層次結構"""
        print("文檔層次結構：")
        print("=" * 50)
```

```
        # 找到根節點
        root_chunks = [chunk for chunk in self.chunks.values() if chunk.level == 0]

        for root in root_chunks:
            self._print_chunk_tree(root, 0)

    def _print_chunk_tree(self, chunk, indent):
        """打印塊樹"""
        prefix = "  " * indent + "  ├── " if indent > 0 else ""
        print(f"{prefix}[L{chunk.level}] {chunk.id}")
        print(f"{'  ' * (indent + 1)}  {len(chunk.content)} 字符")
        print(f"{'  ' * (indent + 1)}  {chunk.content[:50]}...")

        # 遞歸打印子節點
        for child_id in chunk.children_ids:
            if child_id in self.chunks:
                self._print_chunk_tree(self.chunks[child_id], indent + 1)
```

高級檢索功能

```
class HierarchicalRetriever:
    def __init__(self, chunker: HierarchicalChunker):
        self.chunker = chunker

    def multi_level_search(self, query, level=None):
        """多層次搜索"""
        results = []

        for chunk_id, chunk in self.chunker.chunks.items():
            if level is None or chunk.level == level:
                # 簡單的關鍵詞匹配（實際應用中使用向量相似性）
                if self._matches_query(chunk.content, query):
```

```
                results.append({
                    'chunk': chunk,
                    'hierarchy': self.chunker.get_chunk_hierarchy(chunk_id),
                    'relevance_score': self._calculate_relevance(chunk.content, query)
                })

        # 按相關性排序
        results.sort(key=lambda x: x['relevance_score'], reverse=True)
        return results

    def _matches_query(self, text, query):
        """檢查文本是否匹配查詢"""
        query_terms = query.lower().split()
        text_lower = text.lower()
        return any(term in text_lower for term in query_terms)

    def _calculate_relevance(self, text, query):
        """計算相關性分數"""
        query_terms = query.lower().split()
        text_lower = text.lower()
        matches = sum(1 for term in query_terms if term in text_lower)
        return matches / len(query_terms)
```

使用範例

```
text = """
```

人工智能概述

人工智能(AI)是計算機科學的一個分支，致力於創建能夠執行通常需要人類智能的任務的機器。

機器學習

機器學習是AI的一個重要子領域。它使電腦能夠從數據中學習，而無需明確編程。

監督學習

監督學習使用標記的訓練數據。算法從輸入-輸出對中學習模式。

常見的監督學習任務包括分類和回歸。

分類算法

決策樹通過一系列決策規則進行分類。支持向量機找到最優的分類邊界。

回歸算法

線性回歸找到輸入變數和輸出之間的線性關係。

多項式回歸可以捕捉非線性關係。

無監督學習

無監督學習處理未標記的數據。聚類和降維是主要的無監督學習任務。

深度學習

深度學習使用多層神經網絡來學習複雜的模式。

卷積神經網絡擅長處理圖像數據。

遞歸神經網絡適合處理序列數據。

"""

創建分層分塊器

chunker = HierarchicalChunker(chunk_sizes=[1000, 400, 150])

hierarchy = chunker.hierarchical_chunking(text)

視覺化層次結構

chunker.visualize_hierarchy()

測試多層次檢索

retriever = HierarchicalRetriever(chunker)

results = retriever.multi_level_search("監督學習算法")

print("\n" + "=" * 50)

print("檢索結果：")

for i, result in enumerate(results[:3]): # 顯示前3個結果

chunk = result

print(f"\n結果 {i+1} (相關性: {result['relevance_score']:.2f}):")

print(f"層次: Level {chunk.level}")

print(f"內容: {chunk.content[:100]}...")

8. Token感知分塊（Token-Aware Chunking）

比喻

這就像是一個經驗豐富的翻譯員在處理文檔。他們知道不同語言的"詞"的概念不同，所以會根據目標語言的特性來分割文本，確保每一部分都在翻譯系統的處理能力範圍內。

特點

基於模型的tokenizer進行分塊

精確控制token數量

適配特定的語言模型

Python實現

python

import tiktoken

from transformers import AutoTokenizer

class TokenAwareChunker:

def __init__(self, model_name="gpt-3.5-turbo", max_tokens=1000, overlap_tokens=100):

self.model_name = model_name

self.max_tokens = max_tokens

```
        self.overlap_tokens = overlap_tokens

        # 初始化tokenizer
        if "gpt" in model_name.lower():
            self.tokenizer = tiktoken.encoding_for_model(model_name)
            self.is_tiktoken = True
        else:
            self.tokenizer = AutoTokenizer.from_pretrained(model_name)
            self.is_tiktoken = False

    def count_tokens(self, text):
        """計算文本的token數量"""
        if self.is_tiktoken:
            return len(self.tokenizer.encode(text))
        else:
            return len(self.tokenizer.encode(text))

    def encode_text(self, text):
        """編碼文本為tokens"""
        if self.is_tiktoken:
            return self.tokenizer.encode(text)
        else:
            return self.tokenizer.encode(text)

    def decode_tokens(self, tokens):
        """解碼tokens為文本"""
        if self.is_tiktoken:
            return self.tokenizer.decode(tokens)
        else:
            return self.tokenizer.decode(tokens)
```

```
def token_aware_chunking(self, text):
    """基於token的分塊"""
    # 編碼整個文本
    tokens = self.encode_text(text)
    total_tokens = len(tokens)

    if total_tokens <= self.max_tokens:
        return [{
            'text': text,
            'tokens': tokens,
            'token_count': total_tokens,
            'start_token': 0,
            'end_token': total_tokens
        }]

    chunks = []
    start = 0

    while start < total_tokens:
        # 計算結束位置
        end = min(start + self.max_tokens, total_tokens)

        # 提取token片段
        chunk_tokens = tokens

        # 解碼為文本
        chunk_text = self.decode_tokens(chunk_tokens)

        # 嘗試在句子邊界結束
        if end < total_tokens:
            chunk_text = self._adjust_to_sentence_boundary(chunk_text)
```

```
            # 重新編碼調整後的文本以獲得實際token數
            actual_tokens = self.encode_text(chunk_text)
            chunk_tokens = actual_tokens

        chunks.append({
            'text': chunk_text,
            'tokens': chunk_tokens,
            'token_count': len(chunk_tokens),
            'start_token': start,
            'end_token': start + len(chunk_tokens),
            'overlap_tokens': min(self.overlap_tokens, start) if start > 0 else 0
        })

        # 移動到下一個位置（考慮重疊）
        start += len(chunk_tokens) - self.overlap_tokens

        if start >= total_tokens:
            break

    return chunks

def _adjust_to_sentence_boundary(self, text):
    """調整到句子邊界"""
    # 找到最後一個句號、感嘆號或問號
    last_punct = -1
    for i in range(len(text) - 1, -1, -1):
        if text[i] in '。！？':
            last_punct = i
            break

    # 如果找到標點且位置合理，在那裡截斷
```

```
        if last_punct > len(text) * 0.7: # 至少保留70%的內容
            return text[:last_punct + 1]

        return text

    def analyze_token_distribution(self, chunks):
        """分析token分佈"""
        token_counts = [chunk['token_count'] for chunk in chunks]
        overlap_counts = [chunk['overlap_tokens'] for chunk in chunks]

        analysis = {
            'total_chunks': len(chunks),
            'avg_tokens_per_chunk': sum(token_counts) / len(token_counts),
            'min_tokens': min(token_counts),
            'max_tokens': max(token_counts),
            'total_tokens': sum(token_counts),
            'avg_overlap': sum(overlap_counts) / len(overlap_counts) if overlap_counts else 0,
            'token_efficiency': (sum(token_counts) - sum(overlap_counts)) / sum(token_counts)
        }

        return analysis
```

進階：自適應token分塊

```
class AdaptiveTokenChunker(TokenAwareChunker):
    def __init__(self, model_name="gpt-3.5-turbo", target_tokens=800, tolerance=0.2):
        super().__init__(model_name, target_tokens)
        self.target_tokens = target_tokens
        self.tolerance = tolerance
        self.min_tokens = int(target_tokens * (1 - tolerance))
        self.max_tokens = int(target_tokens * (1 + tolerance))
```

```
def adaptive_chunking(self, text):
    """自適應token分塊"""
    sentences = self._split_into_sentences(text)
    chunks = []
    current_chunk_sentences = []
    current_tokens = 0

    for sentence in sentences:
        sentence_tokens = self.count_tokens(sentence)

        # 檢查添加這個句子是否會超過限制
        if current_tokens + sentence_tokens <= self.max_tokens:
            current_chunk_sentences.append(sentence)
            current_tokens += sentence_tokens
        else:
            # 如果當前塊達到最小大小，保存它
            if current_tokens >= self.min_tokens:
                chunk_text = ''.join(current_chunk_sentences)
                chunks.append({
                    'text': chunk_text,
                    'token_count': current_tokens,
                    'sentence_count': len(current_chunk_sentences),
                    'quality_score': self._calculate_chunk_quality(chunk_text)
                })

                current_chunk_sentences =
                current_tokens = sentence_tokens
            else:
                # 當前塊太小，繼續添加
                current_chunk_sentences.append(sentence)
```

```
            current_tokens += sentence_tokens

    # 處理最後一個塊
    if current_chunk_sentences:
        chunk_text = ''.join(current_chunk_sentences)
        chunks.append({
            'text': chunk_text,
            'token_count': current_tokens,
            'sentence_count': len(current_chunk_sentences),
            'quality_score': self._calculate_chunk_quality(chunk_text)
        })

    return chunks

def _split_into_sentences(self, text):
    """分割成句子"""
    sentences = re.split(r'([。！？])', text)
    result = []
    for i in range(0, len(sentences) - 1, 2):
        if i + 1 < len(sentences):
            sentence = sentences[i] + sentences[i + 1]
            if sentence.strip():
                result.append(sentence)
    return result

def _calculate_chunk_quality(self, text):
    """計算塊的質量分數"""
    # 基於多個因素計算質量
    factors = {
        'length_score': min(len(text) / self.target_tokens, 1.0),
```

```
        'sentence_completeness': 1.0 if text.endswith(('。', '！', '？')) else 0.7,
        'information_density': len(set(text.split())) / len(text.split()) if text.split() else 0
    }

    # 加權平均
    weights = {'length_score': 0.3, 'sentence_completeness': 0.4, 'information_density': 0.3}
    quality_score = sum(factors[key] * weights[key] for key in factors)

    return quality_score
```

使用範例

```
print("=== Token感知分塊示例 ===")
```

基本token分塊

```
basic_chunker = TokenAwareChunker(model_name="gpt-3.5-turbo", max_tokens=100, overlap_tokens=20)

text = """
自然語言處理的發展可以分為幾個重要階段。早期的符號主義方法依賴手工編寫的規則和專家知識。
統計方法的引入標誌著一個重要轉折，系統開始能夠從大量語料庫中自動學習語言模式。
機器學習技術的應用進一步提升了NLP系統的性能，特別是在語音識別和機器翻譯領域。
深度學習革命帶來了突破性進展，Transformer架構的出現徹底改變了整個領域。
大型語言模型如GPT系列展現出前所未有的語言理解和生成能力。
"""

chunks = basic_chunker.token_aware_chunking(text)
analysis = basic_chunker.analyze_token_distribution(chunks)

print("基本Token分塊結果：")
for i, chunk in enumerate(chunks):
    print(f"塊 {i+1}: {chunk['token_count']} tokens")
```

```
    print(f"重疊: {chunk['overlap_tokens']} tokens")
    print(f"內容: {chunk['text'][:80]}...")
    print()

print("Token分佈分析：")
for key, value in analysis.items():
    print(f"{key}: {value:.2f}")

print("\n" + "=" * 50)
```

自適應token分塊

```
adaptive_chunker = AdaptiveTokenChunker(model_name="gpt-3.5-turbo", target_tokens=80)
adaptive_chunks = adaptive_chunker.adaptive_chunking(text)

print("自適應Token分塊結果：")
for i, chunk in enumerate(adaptive_chunks):
    print(f"塊 {i+1}:")
    print(f"  Tokens: {chunk['token_count']}")
    print(f"  句子數: {chunk['sentence_count']}")
    print(f"  質量分數: {chunk['quality_score']:.3f}")
    print(f"  內容: {chunk['text'][:60]}...")
    print()
```

9. 最佳實踐和選擇指南

選擇分塊策略的決策樹

```
python
class ChunkingStrategySelector:
    """分塊策略選擇器"""

    @staticmethod
    def recommend_strategy(document_type, document_size, use_case, available_resources):
```

```
    """推薦最適合的分塊策略"""

    recommendations = []

    # 基於文檔類型
    if document_type == 'technical_doc':
        recommendations.append({
            'strategy': 'document_aware',
            'reason': '技術文檔有清晰的結構層次',
            'priority': 9
        })

    elif document_type == 'narrative':
        recommendations.append({
            'strategy': 'semantic',
            'reason': '敘事文本需要保持語義連貫性',
            'priority': 8
        })

    elif document_type == 'conversational':
        recommendations.append({
            'strategy': 'sentence_based',
            'reason': '對話文本以句子為自然單位',
            'priority': 7
        })

    # 基於文檔大小
    if document_size < 10000: # 小文檔
        recommendations.append({
            'strategy': 'fixed_size',
            'reason': '小文檔使用簡單策略即可',
```

```
            'priority': 6
        })

    elif document_size > 100000: # 大文檔
        recommendations.append({
            'strategy': 'hierarchical',
            'reason': '大文檔需要層次化處理',
            'priority': 9
        })

    # 基於使用場景
    if use_case == 'qa_system':
        recommendations.append({
            'strategy': 'sliding_window',
            'reason': '問答系統需要避免信息丟失',
            'priority': 8
        })

    elif use_case == 'summarization':
        recommendations.append({
            'strategy': 'semantic',
            'reason': '摘要需要理解內容主題',
            'priority': 9
        })

    elif use_case == 'search':
        recommendations.append({
            'strategy': 'token_aware',
            'reason': '搜索需要精確的token控制',
            'priority': 7
```

```
        })

    # 基於可用資源
    if available_resources == 'limited':
        recommendations.append({
            'strategy': 'fixed_size',
            'reason': '資源有限時使用簡單方法',
            'priority': 8
        })

    # 按優先級排序
    recommendations.sort(key=lambda x: x['priority'], reverse=True)

    return recommendations
```

綜合分塊策略

```
class HybridChunker:
    """混合分塊器 - 結合多種策略的優點"""

    def __init__(self):
        self.strategies = {
            'fixed': TokenAwareChunker(),
            'semantic': SemanticChunker() if 'SemanticChunker' in globals() else None,
            'document': DocumentAwareChunker(),
            'hierarchical': HierarchicalChunker()
        }

    def adaptive_chunking(self, text, context=None):
        """根據內容特征自適應選擇策略"""

        # 分析文本特征
```

```
    features = self._analyze_text_features(text)

    # 根據特征選擇策略
    strategy = self._select_strategy(features, context)

    # 執行分塊
    chunker = self.strategies
    if strategy == 'fixed':
        chunks = chunker.token_aware_chunking(text)
    elif strategy == 'semantic' and chunker:
        chunks = chunker.semantic_chunking(text)
    elif strategy == 'document':
        chunks = chunker.structure_aware_chunking(text)
    elif strategy == 'hierarchical':
        hierarchy = chunker.hierarchical_chunking(text)
        chunks = [chunk for chunk in hierarchy.values() if chunk.level == 1]

    return {
        'chunks': chunks,
        'strategy_used': strategy,
        'text_features': features,
        'chunk_count': len(chunks)
    }

def _analyze_text_features(self, text):
    """分析文本特征"""
    features = {
        'length': len(text),
        'has_headers': bool(re.search(r'^#+\s', text, re.MULTILINE)),
        'has_lists': bool(re.search(r'^\s[- +]\s', text, re.MULTILINE)),
```

```
        'has_code': bool(re.search(r'```', text)),
        'paragraph_count': len(re.split(r'\n\s*\n', text)),
        'sentence_count': len(re.split(r'[。！？]', text)),
        'avg_sentence_length': len(text) / max(len(re.split(r'[。！？]', text)), 1),
        'structure_score': self._calculate_structure_score(text)
    }

    return features

def _calculate_structure_score(self, text):
    """計算結構化程度分數"""
    score = 0

    # 檢查各種結構元素
    if re.search(r'^#+\s', text, re.MULTILINE):
        score += 3 # 有標題

    if re.search(r'^\s*[-+]\s', text, re.MULTILINE):
        score += 2 # 有列表

    if re.search(r'\n\s*\n', text):
        score += 1 # 有段落分隔

    if re.search(r'```', text):
        score += 2 # 有代碼塊

    return min(score, 5) # 最大分數為5

def _select_strategy(self, features, context):
    """選擇最適合的策略"""

    # 如果文檔有清晰結構，使用文檔感知
```

```
    if features['structure_score'] >= 3:
        return 'document'

    # 如果文檔很大，使用分層
    if features['length'] > 10000:
        return 'hierarchical'

    # 如果句子很長或內容複雜，使用語義分塊
    if (features['avg_sentence_length'] > 100 and
        self.strategies['semantic'] is not None):
        return 'semantic'

    # 默認使用token感知
    return 'fixed'
```

使用範例和比較

```
def compare_chunking_strategies(text):
    """比較不同分塊策略的效果"""

    print("=== 分塊策略比較 ===")
    print(f"原文長度: {len(text)} 字符")
    print()

    # 1. 固定大小分塊
    fixed_chunker = TokenAwareChunker(max_tokens=150, overlap_tokens=30)
    fixed_chunks = fixed_chunker.token_aware_chunking(text)

    print(f"1. Token感知分塊: {len(fixed_chunks)} 個塊")
    print(f"   平均token數: {sum(c['token_count'] for c in fixed_chunks) / len(fixed_chunks):.1f}")

    # 2. 文檔感知分塊
    doc_chunker = DocumentAwareChunker(max_chunk_size=400)
```

```
doc_chunks = doc_chunker.structure_aware_chunking(text)

print(f"2. 文檔感知分塊: {len(doc_chunks)} 個塊")
print(f" 平均字符數: {sum(c['char_count'] for c in doc_chunks) / len(doc_chunks):.1f}")

# 3. 分層分塊
hier_chunker = HierarchicalChunker(chunk_sizes=[800, 300])
hier_chunks = hier_chunker.hierarchical_chunking(text)
level_1_chunks = [c for c in hier_chunks.values() if c.level == 1]

print(f"3. 分層分塊: {len(level_1_chunks)} 個一級塊")
print(f" 總層次數: {max(c.level for c in hier_chunks.values()) + 1}")

# 4. 混合策略
hybrid_chunker = HybridChunker()
hybrid_result = hybrid_chunker.adaptive_chunking(text)

print(f"4. 自適應混合: {len(hybrid_result['chunks'])} 個塊")
print(f" 選擇的策略: {hybrid_result['strategy_used']}")
print(f" 文本特征分數: {hybrid_result['text_features']['structure_score']}")
```

測試文本

```
test_text = """
```

深度學習在計算機視覺中的應用

卷積神經網絡基礎

卷積神經網絡(CNN)是專門用於處理網格狀數據（如圖像）的深度學習架構。

CNN的核心組件包括卷積層、池化層和全連接層。

卷積層

卷積層使用可學習的濾波器在輸入上滑動，提取局部特征。

每個濾波器檢測特定的模式，如邊緣、紋理或形狀。

池化層

池化層降低特征圖的空間維度，減少計算量和參數數量。

最大池化和平均池化是兩種常用的池化操作。

經典CNN架構

LeNet

LeNet是最早的CNN架構之一，由Yann LeCun在1990年代提出。

它主要用於手寫數字識別任務，結構相對簡單。

AlexNet

AlexNet在2012年ImageNet比賽中大放異彩，標誌著深度學習時代的開始。

它引入了ReLU激活函數和dropout正則化技術。

ResNet

殘差網絡(ResNet)解決了深層網絡的梯度消失問題。

通過引入跳躍連接，ResNet能夠訓練非常深的網絡。

"""

compare_chunking_strategies(test_text)

總結：何時使用哪種策略

策略 最適用場景 優點 缺點

固定大小 簡單文檔、原型開發 實現簡單、速度快 可能破壞語義

遞歸字符 通用文本處理 平衡了簡單性和質量 需要調參

語義分塊 內容理解任務 保持語義完整性 計算成本高

滑動窗口 問答系統、檢索 避免信息丟失 重疊導致冗餘

句子基礎 對話、評論分析 自然語言單位 大小不均勻

文檔感知 結構化文檔 保持文檔結構 複雜度高

分層分塊 大型文檔、企業知識庫 多粒度檢索 實現複雜

Token感知 LLM應用、API限制 精確控制 需要特定tokenizer

實用建議

從簡單開始：先嘗試固定大小或遞歸字符分塊

考慮下游任務：檢索任務偏向重疊，生成任務偏向語義完整

測試和迭代：用實際數據測試不同策略的效果

監控性能：追蹤檢索質量和計算成本

組合策略：根據文檔類型使用不同策略

記住，最好的分塊策略往往是針對你的特定用例和數據調優的組合方法！