苏州大学实验报告

院、系	计算机科学与技术	年级 专业	2020 软件工程	姓名	高歌	学号	2030416018
课程名称	信息检索综合实践					成绩	
指导教师	贡正仙	同组实验者	无		实验日期	2023 年 3 月 22 日	

实 验 名 称　实验 4 倒排索引

一. 实验目的

了解倒排索引相关知识，学习并实现从文档中提取单词并构建倒排索引，供后续实验使用。

二. 实验内容

子任务 1：建索引。从给定的十个文档中提取单词（需要分句、分词、词形还原及去停用词），并按照每行 词条\tDocFreq\tdocID docID… 的格式存储索引文件。

子任务 2：布尔查询。

输入：

- 给定 10 个文档
- 一个布尔查询 Q（And Or 操作）

输出：

- 倒排索引表（存成文件 dict.index）
- Q 的查询结果（界面输出文档名列表）

要求

- 不特别要求做词条变化如 friends->friend 等，直接用空格作为分割符
- 都转成小写 A->a
- 需要停用词
- 查询的合并算法要自己实现，不能直接用 Python 的集合操作

三. 实验步骤和结果

注：代码使用 TypeScript 编写，运行时使用 ts-node。使用 Prettier 与 ESLint 作为代码格式化工具，代码风格遵从 TypeScript ESLint Recommended 标准。

（一）实验步骤

1. 本实验逻辑稍有些复杂，这里对目录结构做一个简单解释：

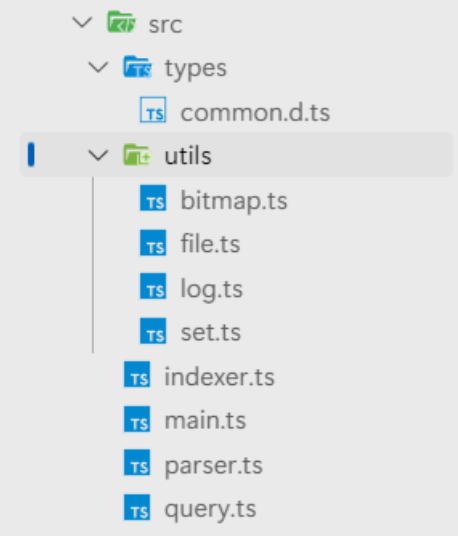

图 1 目录结构

其中，`types`目录下包含了一些必要的类型定义；`utils`目录下包含了一些工具函数，`bitmap.ts`包含了与位图相关的函数，用来辅助集合操作如去重等，`file.ts`包含了一些与文件相关的工具函数，`log.ts`包含了一些与日志相关的工具函数，而`set.ts`包含了基于`bitmap.ts`实现的相关集合操作如交集、并集、差集等（实现中未使用 JS 自带的 Set）；而`indexer.ts`包含了构建与读取索引文件所需的相关函数，`parser.ts`中包含了一个简单的语法分析器（lexer 与 parser），用来将布尔查询表达式转换成抽象语法树以便后续求值，`query.ts`根据抽象语法树与索引字典查询并输出最终结果，其中调用了一些来自`set.ts`中的集合操作工具函数。`main.ts`则为主入口文件。

这里先将几个简单的文件直接解释一下：

①`common.d.ts`文件实际上只包含了一个类型定义`IndexMap`，即索引字典的类型。

```ts
/**
 * The document ID map, where the key is the term and the value is the list
of document IDs.
 */
export type IndexMap = Map<string, number[]>;
```

②`file.ts`仅包含了一个辅助函数，用于判断文件是否存在。该函数用于主入口文件中判断索引文件是否存在，如存在则直接读取，否则先生成索引文件。

```ts
import fs from 'node:fs/promises';

/**
 * Check if a file exists.
 * @param pathname The path to the file.
 * @returns
 */
export const fileExists = async (pathname: string) => {
  try {
    const stat = await fs.stat(pathname);
    return stat.isFile();
  } catch (err) {
    if ((err as NodeJS.ErrnoException).code === 'ENOENT') {
      return false;
```

```
    }
    throw err;
  }
};
```

③`log.ts`仅包含了一个包装器，用于使某一函数调用时可打印日志。该函数的实现与前面几次实验相同，这里不再赘述。

```
/**
 * Log the execution time of a function.
 * @param message The message to be logged.
 * @param fn The function to be executed.
 */
export const logged = <T extends (...args: any[]) => any>(
  message: string,
  fn: T,
): T => {
  return ((...args) => {
    const startTime = Date.now();
    const result = fn(...args);
    if (result instanceof Promise) {
      return result
        .then((res) => {
          console.log(`${message} in ${Date.now() - startTime}ms.`);
          return res;
        })
        .catch((err) => {
          throw err;
        });
    }
    console.log(`${message} in ${Date.now() - startTime}ms.`);
    return result;
  }) as T;
};
```

接下来将一一解释剩余文件中的代码逻辑。

2. `bitmap.ts`中包含了一个简单的位图实现。从效率考虑，位图可以尽量保证 O(n)级别的查找效率，后面的集合操作都使用这里定义的位图实现。

```
/**
 * A data structure that represents a bitmap.
 */
export interface Bitmap {
  value: Uint32Array;
  set: (bitIndex: number) => void;
  clear: (bitIndex: number) => void;
  isSet: (bitIndex: number) => boolean;
```

```
}

/**
 * Create a bitmap.
 * @param size The size of the bitmap.
 * @returns
 */
export const createBitmap = (size: number): Bitmap => ({
  value: new Uint32Array(Math.ceil(size / 32)),
  /**
   * Set a bit in the bitmap.
   * @param index
   */
  set(index: number) {
    const key = Math.floor(index / 32);
    const mask = 1 << index % 32;
    this.value[key] |= mask;
  },
  /**
   * Clear a bit in the bitmap.
   * @param index
   */
  clear(index: number) {
    const key = Math.floor(index / 32);
    const mask = 1 << index % 32;
    this.value[key] &= ~mask;
  },
  /**
   * Check if a bit is set.
   * @param index
   * @returns
   */
  isSet(index: number) {
    const key = Math.floor(index / 32);
    const mask = 1 << index % 32;
    return (this.value[key] & mask) !== 0;
  },
});
```

这段代码应当并不难懂。为了创建任意大小的位图，这里使用了一个稍有些复杂的实现。原本来说，直接使用一个整数当作位图也是可以的，但对于过大的位图，直接使用整数有溢出的可能。这里使用了一个无符号 32 位整数数组作为其内部实现，然后将每一个整数的每一位作为位图的每一位值。该数据结构有三个方法，分别是`set`，用于置位，`clear`，用于清除某一位，及`isSet`，判断某一位是否已置位。相较于传统的导出`setBit`、`clearBit`及`isBitSet`这三个函数的做法，这里采用了稍有些面向对象风格的实现方式。

3. `set.ts`中包含了一些常规的集合操作工具函数。这些工具函数使用刚刚定义的`bitmap.ts`实现。

```
import { createBitmap } from './bitmap.js';
import * as R from 'ramda';

/**
 * Get the union of two arrays.
 * @param arr1
 * @param arr2
 * @returns
 */
export const union = (arr1: number[], arr2: number[]): number[] => {
  const maxElement = Math.max(...arr1, ...arr2);
  const bitmap = createBitmap(maxElement);

  for (const id of arr1) {
    bitmap.set(id);
  }

  for (const id of arr2) {
    bitmap.set(id);
  }

  return R.range(0, maxElement + 1).filter((id) => bitmap.isSet(id));
};

/**
 * Get the intersection of two arrays.
 * @param arr1
 * @param arr2
 * @returns
 */
export const intersection = (arr1: number[], arr2: number[]): number[] =>
{
  const maxElement = Math.max(...arr1, ...arr2);
  const bitmap = createBitmap(maxElement);

  for (const id of arr1) {
    bitmap.set(id);
  }

  return arr2.filter((id) => bitmap.isSet(id));
};

/**
```

```
 * Get the difference of two arrays.
 * @param arr1
 * @param arr2
 * @returns
 */
export const difference = (arr1: number[], arr2: number[]): number[] =>
{
  const maxElement = Math.max(...arr1, ...arr2);
  const bitmap = createBitmap(maxElement);

  for (const id of arr1) {
    bitmap.set(id);
  }

  for (const id of arr2) {
    bitmap.clear(id);
  }

  return R.range(0, maxElement + 1).filter((id) => bitmap.isSet(id));
};
```

这些函数的逻辑都相当清晰，应当不用太多解释。这里使用了`ramda`工具库的`range`函数简化代码，它基本类似于 Python 中的`range`函数，这里只是为了简化代码使用，对函数逻辑并无影响。

4. 然后开始索引相关工具函数的编写。首先在`indexer.ts`中创建三个工具函数。

```
import nlp from 'compromise';

/**
 * Get the document ID from the filename.
 * @param filename
 * @returns
 */
const getDocID = (filename: string): number =>
  Number(filename.replace(/d(\d+)\.txt$/, '$1'));

/**
 * Normalize the text (remove unnecessary spaces, etc.)
 * @param content The text to normalize.
 * @returns
 */
const normalize = (content: string): string =>
  content
    // Remove unnecessary spaces before punctuations
    .replace(/\s+([,.%:;])\s+/g, '$1 ')
    // Remove unnecessary spaces after some punctuations
    .replace(/\s+(\$)\s+/g, ' $1')
```

```
    // Remove unnecessary spaces before and after certain punctuations
    .replace(/\s+(')/g, '$1')
    .replace(/\s(')\s+/g, '$1 ')
    // Remove the unnecessary spaces before the last '.' in the text
    .replace(/\s+\.$/g, '.');

/**
 * Lemmatize a term (verb -> infinitive, noun -> singular).
 * @param term The term to lemmatize.
 * @returns
 */
const lemmatize = (term: string): string => {
  const t = nlp(term);
  if (t.has('#Verb')) {
    return t.verbs().toInfinitive().out('text');
  }
  if (t.has('#Noun')) {
    return t.nouns().toSingular().out('text');
  }
  return term;
};
```

`getDocID`是一个相当简单的函数，它的作用仅仅是从诸如`d10.txt`这样的文件名中提取出文档ID，如对于`d10.txt`就是`10`。

`normalize`的作用是将输入文档正规化。如对于下面这样的文档：

Commercial-vehicle sales in Italy rose 11.4 % in February from a year earlier , to 8,848 units , according to provisional figures from the Italian Association of Auto Makers .

Industrial production in Italy declined 3.4 % in January from a year earlier , the government said .

可以看到，输入文档在标点符号周围加入了不必要的空格。尽管这是为了方便分词考虑，但是对于这里使用的自然语言处理库`compromise`，这样的文档反而不那么容易理解。因此这里使用`normalize`函数，去除这些标点符号周围的空格，使其便于处理。

`lemmatize`函数用于词性还原。它接受一个单词作为输入，若为动词，则返回它的一般形式；若为名词，则返回它的单数形式；对于其他词，则原样返回。

5. 然后编写辅助函数`saveIndex`，它将索引字典存储为文件。

```
import type { IndexMap } from './types/common';
import fs from 'node:fs/promises';
import * as R from 'ramda';

/**
 * Save the index to a file.
 * @param indexMap The index map.
 * @returns
 *
```

```
 * @example
 * ```typescript
 * await saveIndex(new Map([
 *   ['john', [1, 3]],
 *   ['james', [2, 3, 4]],
 *   ['mary', [2, 5]],
 * ])).toFile('index.txt');
 * ```
 */
const saveIndex = (indexMap: IndexMap) => ({
  toFile: async (indexFilePath: string) => {
    await fs.writeFile(
      indexFilePath,
      R.sortBy(R.prop(0), Array.from(indexMap.entries()))
        .map(
          ([word, occurrence]) =>
            `${word}\t${occurrence.length}\t${occurrence.join(' ')}`,
        )
        .join('\n'),
      'utf-8',
    );
  },
});
```

该函数的作用比较简单，即将索引字典排序后按规定的格式保存到指定路径。

6. 然后是`generateIndex`函数，接受一个文件夹路径，读取该文件夹中所有文件并构建索引字典，然后将字典保存到指定路径并返回这个字典。

```
/**
 * Generate the index from the data directory and save it to a file.
 * @param dataDir The data directory.
 * @param pathname The path to the index file.
 * @returns The index map.
 */
export const generateIndex = async (
  dataDir: string,
  pathname: string,
): Promise<IndexMap> => {
  const maxDocID = (await fs.readdir(dataDir)).reduce(
    (maxDocID, filename) => Math.max(maxDocID, getDocID(filename)),
    0,
  );

  const wordOccurs = new Map<string, Bitmap>();

  for (const filename of await fs.readdir(dataDir)) {
```

```
    const docID = getDocID(filename);
    const content = await fs.readFile(path.join(dataDir, filename),
'utf-8');
    const text = normalize(content);

    const doc = nlp(text);

    // Lemmatization
    const words = removeStopwords(
      doc
        .terms()
        .not('#Value')
        .json()
        .map((t: { terms: Array<{ normal: string }> }) =>
t.terms[0].normal)
        .map(lemmatize)
        .filter((w: string) => /\w/.test(w)),
    );

    for (const word of words) {
      if (!wordOccurs.has(word)) {
        wordOccurs.set(word, createBitmap(maxDocID));
      }
      (wordOccurs.get(word) as Bitmap).set(docID);
    }
  }

  // Transform to a map of word -> [docIDs]
  const indexMap = new Map(
    Array.from(wordOccurs.entries()).map(([word, bitmap]) => [
      word,
      R.range(0, maxDocID + 1).filter((docID) => bitmap.isSet(docID)),
    ]),
  );

  // Write to file
  await saveIndex(indexMap).toFile(pathname);

  return indexMap;
};
```

在函数开头，读取文件夹中的所有文件名，然后获取它们的 docID，计算出值最大的 docID，用于下一步构建缓存字典。

然后创建缓存字典`wordOccurs`，它的键是单词，值是一个位图，表示该单词在各文档中是否出现。

然后，遍历每个文件，读取它们的内容，首先使用`normalize`函数正规化文本，然后调用

`compromise`库获取文档中的单词、去除非单词的值（如 8,414）、得到它们的原始形式（如`Commercial-vehicle`变成`commercial`和`vehicle`）、词形还原（动词变成一般时、名词变成单数）、再去除一遍非单词的值以确保无误、最后去除停用词（这里调用了`stopword`这个库提供的`removeStopWords`函数，实际上这个函数仅仅是从单词序列中根据一个停用词表去除这些词而已）。

对于按上述步骤最终得到的每个单词，将其在`wordOccurs`中对应的位图的当前 docID 的值置位（如`wordOccurs`中未包含该单词，则将该单词加入`wordOccurs`并创建对应的位图）。

最后，遍历`wordOccurs`，将它转换成一个键为单词、值为 docID 数组的字典`indexMap`。将该`indexMap`通过`saveIndex`函数保存到文件，并返回该`indexMap`。

7. 类似的，根据索引字典文件保存的格式创建函数`readIndex`，用于将某个路径下的词典文件读取为`IndexMap`。

```
/**
 * Read the index from a file.
 * @param pathname The path to the index file.
 * @returns The index map.
 */
export const readIndex = async (pathname: string): Promise<IndexMap> =>
  (await fs.readFile(pathname, 'utf-8'))
    .split('\n')
    .map((line) => line.split('\t'))
    .map(([word, , docIDs]) => [word, docIDs.split(' ').map(Number)] as
const)
    .reduce((acc, [word, docIDs]) => {
      acc.set(word, docIDs);
      return acc;
    }, new Map());
```

8. 至此，与索引相关的工具函数已经定义完成。下面为布尔查询表达式编写语法分析器。首先在`parser.ts`中创建词法分析器`lexer`。

```
enum TokenType {
  AND = 'AND',
  OR = 'OR',
  NOT = 'NOT',
  LPAREN = 'LPAREN',
  RPAREN = 'RPAREN',
  TERM = 'TERM',
  EOF = 'EOF',
}

interface Token {
  type: TokenType;
  value: string;
}
```

```typescript
/**
 * Tokenize the input string into an array of tokens.
 * @param input The input string to tokenize.
 * @returns An array of tokens.
 */
export const lexer = (input: string): Token[] => {
  const tokens: Token[] = [];
  let currentPosition = 0;

  while (currentPosition < input.length) {
    const char = input[currentPosition];

    if (/\s/.test(char)) {
      currentPosition++;
      continue;
    }

    if (char === '(') {
      tokens.push({ type: TokenType.LPAREN, value: char });
      currentPosition++;
      continue;
    }

    if (char === ')') {
      tokens.push({ type: TokenType.RPAREN, value: char });
      currentPosition++;
      continue;
    }

    const wordMatch = input.slice(currentPosition).match(/^\w+/);
    if (wordMatch) {
      const word = wordMatch[0].toLowerCase();
      currentPosition += word.length;

      switch (word) {
        case 'and':
          tokens.push({ type: TokenType.AND, value: word });
          break;
        case 'or':
          tokens.push({ type: TokenType.OR, value: word });
          break;
        case 'not':
          tokens.push({ type: TokenType.NOT, value: word });
          break;
```

```
        default:
          tokens.push({ type: TokenType.TERM, value: word });
      }
      continue;
    }

    throw new Error(`Unexpected character: ${char}`);
  }

  tokens.push({ type: TokenType.EOF, value: '' });
  return tokens;
};
```

该函数支持对包含以下成分的布尔查询表达式分词：

- AND、OR、NOT 三种基本的逻辑运算符
- 使用左括号（LPAREN）与右括号（RPAREN）手动标注优先级
- 用于查询的其他普通单词

这段代码的逻辑很简单，就是逐一遍历每个字符，然后识别这几个元素将它们转换为 Token，最后输出 Token 数组而已。

9. 然后编写语法分析器`parser`。

```
export type ASTNode = TermNode | AndNode | OrNode | NotNode;

interface TermNode {
  type: 'Term';
  value: string;
}

interface AndNode {
  type: 'And';
  left: ASTNode;
  right: ASTNode;
}

interface OrNode {
  type: 'Or';
  left: ASTNode;
  right: ASTNode;
}

interface NotNode {
  type: 'Not';
  operand: ASTNode;
}
```

```
/**
 * Parse the tokens into an AST.
 * @param tokens The tokens to parse.
 * @returns
 */
export const parser = (tokens: Token[]): ASTNode => {
  let currentTokenIndex = 0;

  const getNextToken = (): Token => tokens[currentTokenIndex++];

  const parseTerm = (token: Token): TermNode => ({
    type: 'Term',
    value: token.value,
  });

  const parseNot = (operand: ASTNode): NotNode => ({
    type: 'Not',
    operand,
  });

  const parseExpression = (): ASTNode => {
    let left = parseFactor();

    // eslint-disable-next-line no-constant-condition
    while (true) {
      const token = getNextToken();

      if (token.type === TokenType.AND) {
        left = { type: 'And', left, right: parseFactor() };
      } else if (token.type === TokenType.OR) {
        left = { type: 'Or', left, right: parseFactor() };
      } else {
        currentTokenIndex--;
        break;
      }
    }

    return left;
  };

  const parseFactor = (): ASTNode => {
    let token = getNextToken();
    if (token.type === TokenType.LPAREN) {
      const expr = parseExpression();
      token = getNextToken();
```

```
      if (token.type !== TokenType.RPAREN) {
        throw new Error('Expected a closing parenthesis');
      }

      return expr;
    }

    if (token.type === TokenType.NOT) {
      const operand = parseFactor();
      return parseNot(operand);
    }

    if (token.type === TokenType.TERM) {
      return parseTerm(token);
    }

    throw new Error(`Unexpected token: ${token.type}`);
  };

  const ast = parseExpression();
  const eofToken = getNextToken();

  if (eofToken.type !== TokenType.EOF) {
    throw new Error('Expected end of input');
  }

  return ast;
};
```

parser 要稍微有些复杂。简单来说，它使用两个函数`parseExpression`和`parseFactor`交替解析 Token 序列。`parseExpression`用于解析包含左值与右值的 AST 节点`AndNode`和`OrNode`，而`parseFactor`用于解析仅包含一个值的节点`TermNode`和`NotNode`，并负责处理括号。

一开始，函数将整个 Token 序列当作一个表达式使用`parseExpression`解析。在`parseExpression`的调用中，它会递归地解析嵌套的`AND`与`OR`，并调用`parseFactor`处理`NOT`、普通单词与括号，最后生成完整的 AST 并返回。这里还对 Token 序列做了一个简单的 EOF 校验，如 Token 序列不以 EOF 结尾，则抛出异常。

10. 至此，关于语法分析器的代码已经编写完成。下面在`query.ts`中编写具体的查询函数。由于已经构建了 AST，该函数的编写比较简单。

```
import { ASTNode, lexer, parser } from './parser.js';
import type { IndexMap } from './types/common';
import { createBitmap } from './utils/bitmap.js';
import { difference, intersection, union } from './utils/set.js';
import * as R from 'ramda';
```

```
/**
 * Get all document IDs from the document ID map.
 * @param indexMap The document ID map, where the key is the term and the
value is the list of document IDs.
 * @returns
 */
const getAllDocIDs = (indexMap: IndexMap): number[] => {
  const maxElement = Math.max(...[...indexMap.values()].flat());
  const bitmap = createBitmap(maxElement);

  for (const docIDs of indexMap.values()) {
    for (const docID of docIDs) {
      bitmap.set(docID);
    }
  }

  return R.range(0, maxElement).filter((docID) => bitmap.isSet(docID));
};

/**
 * Evaluate the AST and return the document IDs that match the query.
 * @param ast The AST.
 * @param indexMap The document ID map, where the key is the term and the
value is the list of document IDs.
 * @returns
 */
const evaluateAst = (
  ast: ASTNode,
  indexMap: IndexMap,
  allDocIDs: number[] = getAllDocIDs(indexMap),
): number[] => {
  switch (ast.type) {
    case 'Term': {
      const term = ast.value;
      return indexMap.get(term) ?? [];
    }
    case 'And': {
      const left = evaluateAst(ast.left, indexMap, allDocIDs);
      const right = evaluateAst(ast.right, indexMap, allDocIDs);
      return intersection(left, right);
    }
    case 'Or': {
      const left = evaluateAst(ast.left, indexMap, allDocIDs);
      const right = evaluateAst(ast.right, indexMap, allDocIDs);
```

```
      return union(left, right);
    }
    case 'Not': {
      const operand = evaluateAst(ast.operand, indexMap, allDocIDs);
      return difference(allDocIDs, operand);
    }
  }
};
```

这里首先定义了一个工具函数`getAllDocIDs`。它看起来有些复杂，实际上很简单，就是从`IndexMap`中获取全部出现过的 docID 而已，这是为了下面`evaluateAst`函数的实现方便编写的辅助函数。

在`evaluateAst`函数中，它接受一个 AST 和一个`IndexMap`，然后对该 AST 根据`IndexMap`递归地进行查询。这里还加入了一个可选参数`allDocIds`，其目的是为了避免在每个函数中都求值一遍全部出现过的 docID，加快查询速度。

现在，为一个布尔查询表达式求值已经很简单了，只需要依次调用`evaluateAst(parser(lexer('aaa and bbb')), indexMap)`即可，但这还是有点不够直观和方便，因此对这个过程稍微封装成一个函数`query`以便使用。

```
/**
 * Get the document IDs that match the query.
 *
 * QUERY ::= TERM | QUERY AND QUERY | QUERY OR QUERY | NOT QUERY
 *
 * TERM ::= \w+
 *
 * AND ::= 'and'
 *
 * OR ::= 'or'
 *
 * NOT ::= 'not'
 *
 * @param input The query string. All terms are case-insensitive.
 * @param docIDMap The document ID map, where the key is the term and the
value is the list of document IDs.
 * @returns
 *
 * @example
 * ```typescript
 * query('john and (james or not mary)').on(
 *   new Map([
 *     ['john', [1, 3]],
 *     ['james', [2, 3, 4]],
 *     ['mary', [2, 5]],
 *   ]),
 * ); // => [1, 3]
```

```
 * ```
 */
export const query = (input: string) => ({
  on: (docIDMap: IndexMap): number[] =>
    evaluateAst(parser(lexer(input)), docIDMap),
});
```

11. 最后在`main.ts`中编写主函数。该函数首先判断索引文件是否存在，如不存在，先构建索引并保存；若存在，则直接读取索引文件。在读取索引文件完成后，等待用户输入布尔查询字符串，然后在索引字典中查询对应的 docID.

```
import { generateIndex, readIndex } from './indexer.js';
import { query } from './query.js';
import { fileExists } from './utils/file.js';
import { logged } from './utils/log.js';
import { stdin, stdout } from 'process';

const DATA_DIR = './data';
const INDEX_PATHNAME = './dict.index';

const loggedReadIndex = logged('Index read', readIndex);
const loggedGenerateIndex = logged('Index generated', generateIndex);

const main = async () => {
  const indexMap = (await fileExists(INDEX_PATHNAME))
    ? await loggedReadIndex(INDEX_PATHNAME)
    : await loggedGenerateIndex(DATA_DIR, INDEX_PATHNAME);

  // Test query
  console.log('Test query: john and (james or not mary)');
  console.log(
    query('john and (james or not mary)').on(
      new Map([
        ['john', [1, 3]],
        ['james', [2, 3, 4]],
        ['mary', [2, 5]],
      ]),
    ),
  );

  stdout.write('Enter a query: ');
  stdin.on('data', (data) => {
    const queryStr = data.toString().trim();
    console.log(query(queryStr).on(indexMap));
    stdout.write('Enter a query: ');
  });
```

```
};

await main();
```

这里写了一个简单的测试查询，在`{‘john’: [1, 3], ‘james’: [2, 3, 4], ‘mary’: [2, 5]}`这个索引字典上查询表达式`john and (james or not mary)`。在查询该表达式时，应首先计算`not mary`，得到`[1, 3, 4]`，然后计算`james or not mary`，得到`[1, 2, 3, 4]`，然后计算`john and (james or not mary)`，得到`[1, 3]`。最终输出结果也符合预期，为`[1, 3]`。这表明查询器执行正常。

```
Index generated in 651ms.
Test query: john and (james or not mary)
[ 1, 3 ]
```

图 2 测试输出

（二）实验结果

运行`npm run dev`，使用 ts-node 执行`./src/main.ts`。依次输入题中要求的测试样例，下为输出结果：

```
Enter a query: john
[ 3, 4, 5 ]
Enter a query: john and james
[ 5 ]
Enter a query: john and aaaaa
[]
Enter a query: john or james
[ 3, 4, 5 ]
Enter a query: john or aaaaa
[ 3, 4, 5 ]
```

图 3 实验结果

四. 实验总结

1. 本次实验使用了 Node.js 上的 NLP 库 compromise 进行分词与词性还原。相比较 Python 中的 NLTK 等传统方案，compromise 提供了非常简洁优雅的接口，但同时也存在一些功能上的局限性，不如传统方案成熟与强大。

2. 本实验使用了位图（Bitmap）实现集合操作与索引表的建立，具有较高的效率。

3. 本实验为布尔查询表达式构建了一个语法解析器，通过使用 lexer 和 parser 将表达式解析为 AST 然后查询的方式以达到理论上可支持任意多层嵌套查询的目的。编写一个这样的简单解析器实际上并不困难，而且使得具体查询功能的实现更加优雅与符合直觉，避免了对太多边界情况的处理。

4. 通过本次实验，了解并实践了倒排索引的构建与查询，为后面的模糊查询实验做了铺垫。