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 2 月 22 日	

实 验 名 称　　实验 1 单词计数

一. 实验目的

初步了解信息检索的基本知识。

二. 实验内容

统计所给数据文件中每一个单词的出现次数。

1. 将给定的文本文件 sample-en.txt 保存到本地文件夹。
2. 将文件中的单词和对应的出现次数按要求保存到词典文件 dict.index 中。
3. 完成步骤 2 后，等待用户输入

- 用户每输入一个单词，屏幕输出该单词的次数
- 如果输入单词不在词典文件中，则输出-1
- 输入###，程序退出

将步骤 2 生成的词典文件放在运行程序的当前路径下，并命名为 dict.index

- 在读取 sample-en.txt 内容时，直接用空白符作为单词的分割符
- 大写字母都转成小写，如 A->a

要求 dict.index 中的数据按行存放，每行格式：词条\t 频次

测试样例：

- 输入：from 输出：3
- 输入：china 输出：-1
- 输入：government 输出：1
- 输入：### 程序退出

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 JavaScript Standard Style 标准。使用 Jest 编写测试代码。

（一）实验步骤

1. 在`./src`目录下创建`utils.ts`文件，用于导出若干辅助函数。

```
import fs from 'node:fs/promises'
import crypto from 'node:crypto'


...
```

2. 创建第一个函数`parseSourceFileToMap()`，它接收一个 string 类型的参数`filePath`，将指定的源数据文件转换成一个 Map，其中键为单词，值为该单词的出现次数。该函数统一将单词转换为小写，使用正则表达式`/[a-zA-Z]+/g`找出文件中的所有单词。这里使用了 Node.js 环境中的`node:fs/promises`包读取文件，使用 JavaScript 中的 Map 存储单词信息。

下面的代码应当不难理解，唯一可能需要解释的或许是这里的`??`运算符，该运算符判断前面的值是否为 null 或 undefined，若是，则返回后面的值，否则返回前面的值。`x ?? y`可以理解为`x === undefined || x === null ? x : y`的简写形式。这是 JavaScript 标准 ES2020 中新加入的运算符，近几年已被各编程语言广泛采用。

```
/**
 * Read data from source file and return a map
 * which maps words to the number of occurrences
 * @param filePath
 */
export const parseSourceFileToMap = async (
  filePath: string
): Promise<Map<string, number>> => {
  const word2numMap = new Map<string, number>()
  const fileContent = await fs.readFile(filePath, 'utf-8')
  const words = fileContent.match(/[a-zA-Z]+/g) ?? []
  for (const word of words) {
    word2numMap.set(word.toLowerCase(), (word2numMap.get(word) ?? 0) + 1)
  }
  return word2numMap
}
```

3. 创建第二个函数`checkDictFileExists()`，该函数接收一个 string 类型的参数`filePath`，判断词典文件是否存在。该函数的功能比较简单，无需过多解释。

```
/**
 * Check if the dict file exists
 * @param filePath
 */
export const checkDictFileExists = async (
  filePath: string
): Promise<boolean> => {
  return await fs
    .access(filePath)
    .then(() => true)
```

```
        .catch(() => false)
    }
```

4. 创建第三个函数`cacheMapToDictFile()`，该函数接收三个参数，第一个是 string 类型的参数`dictFilePath`，第二个是 string 类型的参数`sourceFilePath`，第三个是 Map<string, number>类型的参数 word2numMap，即上文提到的 Map。

首先，该函数计算源数据文件的 MD5 值，并保存到文件首行。**其目的是为了后续判断源数据文件是否更改，以此更新缓存词典文件，防止缓存过时。**

然后，该函数将这个 Map 按单词出现次数从大到小排序后转换成字符串，与源数据文件的 MD5 值一同保存到文件中。

```
    /**
     * Save the word2numMap to dict file
     * The first line of the dict file is the hash of the source file
     * @param dictFilePath
     * @param sourceFilePath
     * @param word2numMap
     */
    export const cacheMapToDictFile = async (
      dictFilePath: string,
      sourceFilePath: string,
      word2numMap: Map<string, number>
    ): Promise<void> => {
      // Get MD5 hash of the source file
      const hash = crypto.createHash('md5')
      const sourceFileContent = await fs.readFile(sourceFilePath, 'utf-8')
      hash.update(sourceFileContent)
      const hashResult = hash.digest('hex')

      // Write the word2numMap to the dict file
      const result = Array.from(word2numMap.entries())
        .sort((a, b) => b[1] - a[1])
        .map(([word, num]) => `${word}\t${num}`)
        .join('\n')

      await fs.writeFile(dictFilePath, `${hashResult}\n${result}`)
    }
```

5. 创建第四个函数`parseDictFileToMap()`，该函数接收一个 string 类型的参数`filePath`，读取该缓存的词典文件，并将其解析为上面提到的键为单词、值为单词出现次数的 Map。

值得注意的是，由于`cacheMapToDictFile()`函数将源数据文件的 MD5 值放在缓存词典文件的首行，所以这里需要跳过第一行。

```
/**
 * Read data from dict file and return a map
 * which maps words to the number of occurrences
 * @param filePath
 */
export const parseDictFileToMap = async (
  filePath: string
): Promise<Map<string, number>> => {
  const dictContent = await fs.readFile(filePath, 'utf-8')
  const word2numMap = new Map<string, number>()
  // Skip the first line because it is the hash of the source file
  for (const line of dictContent.split('\n').slice(1)) {
    const [word, num] = line.split('\t')
    word2numMap.set(word, Number(num))
  }
  return word2numMap
}
```

6. 最后，编写函数`compareFileHash()`，该函数接受两个 string 类型的参数，分别表示缓存词典文件的路径与源数据文件的路径。该函数首先读取源数据文件的内容，计算其 MD5 值，然后读取缓存词典文件的第一行，获取该词典文件对应的源数据文件的 MD5 值，并比较这两个值是否相同。

该函数用于判断源数据文件是否已经更改，以供更新缓存词典文件。

```
/**
 * Compare the hash of the source file and the dict file (from the first line)
 * @param dictFilePath
 * @param sourceFilePath
 */
export const compareFileHash = async (
  dictFilePath: string,
  sourceFilePath: string
): Promise<boolean> => {
  // Get MD5 hash of the source file
  const sourceFileHash = crypto.createHash('md5')
  const sourceFileContent = await fs.readFile(sourceFilePath, 'utf-8');
  sourceFileHash.update(sourceFileContent)
  const sourceFileResult = sourceFileHash.digest('hex')

  // Get the hash from the first line of the dict file
  const dictFileContent = await fs.readFile(dictFilePath, 'utf-8')
  const dictFileResult = dictFileContent.split('\n')[0]

  return sourceFileResult === dictFileResult
}
```

7. 然后，在`./tests`目录下新建`utils.test.ts`，使用 Jest 为这些函数编写相应的测试。经测试全部通过。

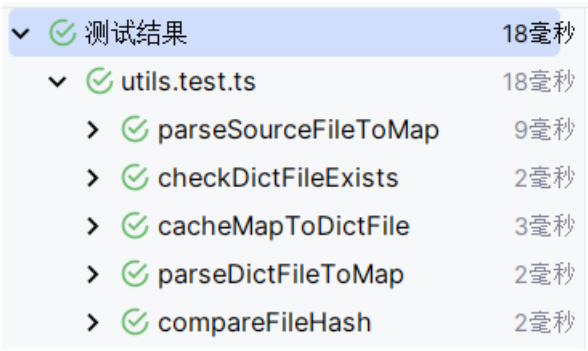

图 1 测试结果

下为测试代码。

```
import fs from 'node:fs/promises'
import crypto from 'node:crypto'
import {
  cacheMapToDictFile,
  checkDictFileExists,
  compareFileHash,
  parseDictFileToMap,
  parseSourceFileToMap
} from '../src/utils'

describe('parseSourceFileToMap', () => {
  // Mock the readFile function
  beforeAll(() => {
    jest
      .spyOn(fs, 'readFile')
      .mockImplementation(() => Promise.resolve('foo foo bar'))
  })

  // Restore the original readFile function
  afterAll(() => {
    jest.restoreAllMocks()
  })

  it('should return a map', async () => {
    const result = await parseSourceFileToMap('')
    expect(result).toBeInstanceOf(Map)
  })

  it('should return a map with correct data', async () => {
    const result = await parseSourceFileToMap('')
    expect(result.get('foo')).toBe(2)
    expect(result.get('bar')).toBe(1)
    expect(result.get('baz')).toBeUndefined()
  })
```

```
})

describe('checkDictFileExists', () => {
  // Mock the access function
  beforeAll(() => {
    jest.spyOn(fs, 'access').mockImplementation(() => Promise.resolve())
  })

  // Restore the original access function
  afterAll(() => {
    jest.restoreAllMocks()
  })

  it('should return true if the file exists', async () => {
    const result = await checkDictFileExists('')
  })

  it('should return false if the file does not exist', async () => {
    jest.restoreAllMocks()
    await jest.spyOn(fs, 'access').mockImplementation(() => Promise.reject())
    const result = await checkDictFileExists('')
  })
})

describe('cacheMapToDictFile', () => {
  // Mock the writeFile function
  beforeAll(() => {
    jest.spyOn(fs, 'writeFile').mockImplementation(() => Promise.resolve())
    jest.spyOn(fs, 'readFile').mockImplementation(() => Promise.resolve(''))
  })

  // Restore the original writeFile function
  afterAll(() => {
    jest.restoreAllMocks()
  })

  it('should call writeFile function', async () => {
    const result = await cacheMapToDictFile('', '', new Map())
    expect(fs.writeFile).toBeCalled()
  })

  it('should call writeFile function with correct data', async () => {
    const result = await cacheMapToDictFile(
      '',
      '',
```

```
      new Map([
        ['foo', 2],
        ['bar', 1]
      ])
    )
    expect(fs.writeFile).toBeCalledWith('', `${
      crypto.createHash('md5').update('').digest('hex')
    }\nfoo\t2\nbar\t1`)
  })
})


describe('parseDictFileToMap', () => {
  // Mock the readFile function
  beforeAll(() => {
    jest
      .spyOn(fs, 'readFile')
      .mockImplementation(() => Promise.resolve('1234\nfoo\t2\nbar\t1'))
  })

  // Restore the original readFile function
  afterAll(() => {
    jest.restoreAllMocks()
  })

  it('should return a map', async () => {
    const result = await parseDictFileToMap('')
    expect(result).toBeInstanceOf(Map)
  })

  it('should return a map with correct data', async () => {
    const result = await parseDictFileToMap('')
    expect(result.get('foo')).toBe(2)
    expect(result.get('bar')).toBe(1)
    expect(result.get('baz')).toBeUndefined()
  })
})

describe('compareFileHash', () => {
  // Mock the readFile function
  beforeAll(() => {
    const sourceFileContentOriginal = 'foo foo bar'
    const originalHash = crypto
      .createHash('md5')
      .update(sourceFileContentOriginal)
```

```
        .digest('hex')
      const sourceFileContentModified = 'foo foo bar baz'

      // mock the readFile function with two different results
      jest
        .spyOn(fs, 'readFile')
        .mockImplementationOnce(() => Promise.resolve(sourceFileContentOriginal))
        .mockImplementationOnce(() => Promise.resolve(`${originalHash}\nfoo\t2\nbar\t1`))
        .mockImplementationOnce(() => Promise.resolve(sourceFileContentModified))
        .mockImplementationOnce(() =>
  Promise.resolve(`${originalHash}\nfoo\t2\nbar\t1\nbaz\t1`))
    })

    // Restore the original readFile function
    afterAll(() => {
      jest.restoreAllMocks()
    })

    it('should return true if the hash is the same', async () => {
      const result = await compareFileHash('', '')
      expect(result).toBe(true)
    })

    it('should return false if the hash is different', async () => {
      const result = await compareFileHash('', '')
      expect(result).toBe(false)
    })
  })
```

测试代码整体上比较简单。值得注意的是这里使用 Jest 的 Mock 模拟了 Node.js 与文件读取相关的几个函数，主要是为了测试时不必创建真实的文件。

8. 在`src`目录下编写主函数`main.ts`，代码如下。

```
import {
  cacheMapToDictFile,
  checkDictFileExists,
  compareFileHash,
  parseDictFileToMap,
  parseSourceFileToMap
} from './utils.js'

const SOURCE_FILE_PATH = 'data/sample-en.txt'
const DICT_FILE_PATH = 'dict.index'

async function main() {
  let word2numMap: Map<string, number>
  // Read data from source file or dict file
```

```
  const isDictFileExists = await checkDictFileExists(DICT_FILE_PATH)
  if (isDictFileExists && await compareFileHash(DICT_FILE_PATH, SOURCE_FILE_PATH)) {
    // If the source file exists and has not been modified, read data from dict file
    console.log('Reading data from cache dict file...')
    const startTime = Date.now()
    word2numMap = await parseDictFileToMap(DICT_FILE_PATH)
    console.log(`Data loaded in ${Date.now() - startTime}ms`)
  } else {
    // If the source file does not exist or has been modified, read data from source file,
    if (isDictFileExists) {
      console.log('Source file has been modified, reading data from source file...')
    } else {
      console.log(`Dict file not found at ${DICT_FILE_PATH}`)
    }
    console.log('Reading data from source file...')
    const startTime1 = Date.now()
    word2numMap = await parseSourceFileToMap(SOURCE_FILE_PATH)
    console.log(`Data loaded in ${Date.now() - startTime1}ms`)

    // Save the word2numMap to dict file
    console.log('Caching the dict to speed up the next run...')
    const startTime2 = Date.now()
    await cacheMapToDictFile(DICT_FILE_PATH, SOURCE_FILE_PATH, word2numMap)
    console.log(`Dict saved in ${Date.now() - startTime2}ms`)
    console.log(`Dict file saved at ${DICT_FILE_PATH}`)
  }

  // Wait for the user to input words until the user inputs '###'
  // Each time the user inputs a word, print the number of occurrences of the word
  console.log('Please input a word to search (input ### to exit):')
  const stdin = process.openStdin()
  stdin.addListener('data', input => {
    const word = input.toString().trim()
    if (word === '###') {
      process.exit()
    }
    console.log(word2numMap.get(word) ?? -1)
  })
}

await main()
```

在程序开头创建了两个常量`SOURCE_FILE_PATH`和`DICT_FILE_PATH`，分别表示源数据文件的路径及缓存词典文件的保存路径。

主程序先判断缓存词典文件是否存在。若存在，再次使用`compareFileHash()`函数**判断源数据文件是否与缓存的词典文件对应**。若对应，则源数据文件未更改，读取词典文件，存到 word2numMap 中；

若文件不存在或不对应，从源数据文件中读取 word2numMap，并缓存词典文件。

在此之后，打开标准输入，监听用户输入。若用户输入`###`，退出程序，否则每次用户输入单词，则仓 word2numMap 中查询单词的出现次数，若不存在，则打印-1。

（二）实验结果

运行`npm run dev`，使用 ts-node 执行`main.ts`，在词典文件不存在时，输出如下。

```
[nodemon] starting `npx ts-node ./src/main.ts`
Dict file not found at dict.index
Reading data from source file...
Data loaded in 1ms
Caching the dict to speed up the next run...
Dict saved in 2ms
Dict file saved at dict.index
Please input a word to search (input ### to exit):
from
3
china
-1
government
1
###
[nodemon] clean exit - waiting for changes before restart
```

图 2 测试样例 1

词典文件存在时（即第二次及之后运行时，**且未改变源数据文件时**），输出如下。

```
[nodemon] starting `npx ts-node ./src/main.ts`
Reading data from cache dict file...
Data loaded in 1ms
Please input a word to search (input ### to exit):
from
3
china
-1
government
1
###
[nodemon] clean exit - waiting for changes before restart
```

图 3 测试样例 2

词典文件存在，**但源数据文件更改时（这里将文件中的一个 from 改成了 to）**，输出如下：

```
Source file has been modified, reading data from source file...
Reading data from source file...
Data loaded in 0ms
Caching the dict to speed up the next run...
Dict saved in 2ms
Dict file saved at dict.index
Please input a word to search (input ### to exit):
from
2
china
-1
government
1
###
[nodemon] clean exit - waiting for changes before restart
```

图 4 测试样例 3

注意，这里将源数据文件中的一个 from 改成了 to，因此**这里的显示是正确**的，说明程序正确处理了源数据文件变动后更新缓存词典文件的情况。

四. 实验总结

1. 观察到在所给的 sample-en.txt 文件中，与常见英文不同的是，文件中的标点符号在左右两侧都添加了空格，如`11.4 %`而非`11.4%`，`units , according`而非`units, according`。

2. 在本次实验中，使用 Node.js 环境编写了基本的单词计数程序，实现了对文件中单词出现次数的统计，并将其缓存到词典文件中以加速下次运行时的读取速度，**且实现了检查源数据文件是否更改，并更新缓存词典文件的功能**。通过本次实验，建立了对信息检索基本方式的认识，了解了信息检索的定义、目标、要求等相关知识点。