信息检索系统

- 项目概述
  - 项目要求
  - 项目架构
- 详细设计
  - 数据爬取
  - 前端设计与实现
  - 后端设计与实现
  - 信息检索服务
  - 多媒体关键词提取服务
  - 检索结果评价
- 优化与创新性
- 环境和社会可持续发展思考
- 实验总结
- 实验分工

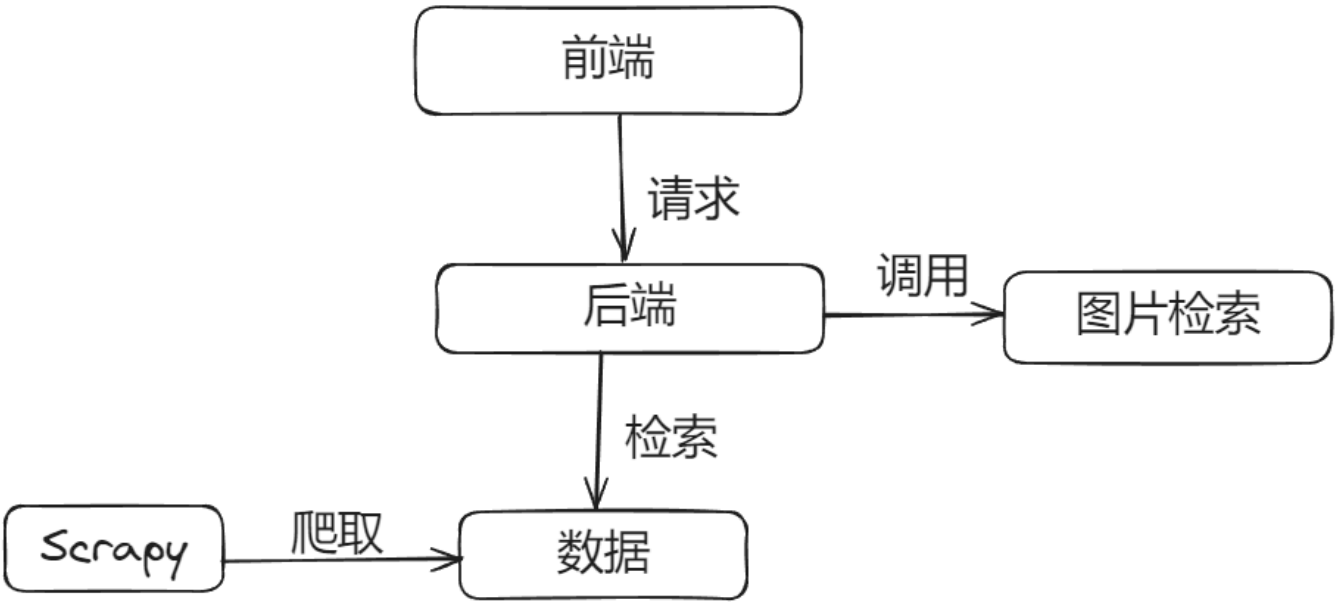
项目概述

项目要求

本实验要求自己动手设计实现一个信息检索系统，中、英文皆可，数据源可以自选，数据通过开源的网络爬虫获取，规模不低于 100 篇文档，进行本地存储。中文可以分词（可用开源代码），也可以不分词，直接使用字作为基本单元。英文可以直接通过空格分隔。构建基本的倒排索引文件。实现基本的向量空间检索模型的匹配算法。用户查询输入可以是自然语言字串，查询结果输出按相关度从大到小排序，列出相关度、题目、主要匹配内容、URL、日期等信息。最好能对检索结果的准确率进行人工评价。界面不做强制要求，可以是命令行，也可以是可操作的界面。提交作业报告和源代码。

项目架构

本项目主要由四个部分组成：数据爬取、前端展示、后端处理数据和图片检索服务，整体架构图和详细内容如下：



- 数据爬取使用 Scrapy 框架爬取网页文章并存储到 json 文件中，方便后续使用；
- 前端展示使用 Vue 框架实现，为用户提供了清晰直观的操作界面；
- 后端使用基于 Go 的 Gin 框架开发，接收用户请求并处理，并且对文章进行初始化处理，如：使用结巴框架进行分词、建立倒排索引、计算 TF-IDF 值等；
- 图片检索服务则使用 Python 进行开发，使用 Flask 框架给后端提供调用接口，使用 Tensorflow 框架对图片进行识别并提取关键词用于检索。

项目运行截图如下：

详细设计

数据爬取

本次实验要求不少于 100 篇文档，所以我们结合自己情况爬取了比较常用的全中文的**OiWiki**和最近在学习的全英文的**The Rust Programming Language**，最终爬取文章数量为中文 **440** 篇，英文 **104** 篇。爬虫框架选取了我们最为熟悉的 Scrapy，使用该框架可以快速爬取网页内容，并且可以方便的进行数据处理。

OiWiki 数据爬取

对于 OiWiki，我们首先爬取文章列表：

```
def parse(self, response):
    sections = response.xpath(
        "//li[@class='md-nav__item']/a[@class='md-nav__link']"
    )
    hrefs = sections.xpath("@href").getall()
    texts = sections.xpath("text()").getall()
    texts = [t.strip() for t in texts]

    for href, section in zip(hrefs, texts):
        url = response.urljoin(href)
        yield scrapy.Request(
            url=url,
            callback=self.parse_section,
            cb_kwargs={"section": section},
        )
```

然后爬取每篇文章内容：

```
def parse_section(self, response, section="Unknown"):
    content = response.xpath(
        '//div[@class="md-content"]//blockquote[1]/preceding-sibling::*[not(self::a)]'
    ).getall()
    keywords = response.xpath(
        '//div[@class="md-content"]//*[self::h1 or self::h2 or self::h3 or self::h4 or self::li
    ).getall()

    self.id = self.id + 1
    yield {
        "id": str(self.id),
        "title": content[0],
        "content": "".join(para for para in content),
        "keywords": "".join(para for para in keywords),
        "url": response.url,
        "date": datetime.date.today().strftime("%Y-%m-%d"),
    }
```

其中，我们将全文内容作为文章内容用于前端展示，文章中的所有文本内容作为关键字用于索引和检索。

The Rust Programming Language 数据爬取

对于 The Rust Programming Language，我们同样地也是先爬取文章所有章节，然后再爬取每个章节内的详细内容：

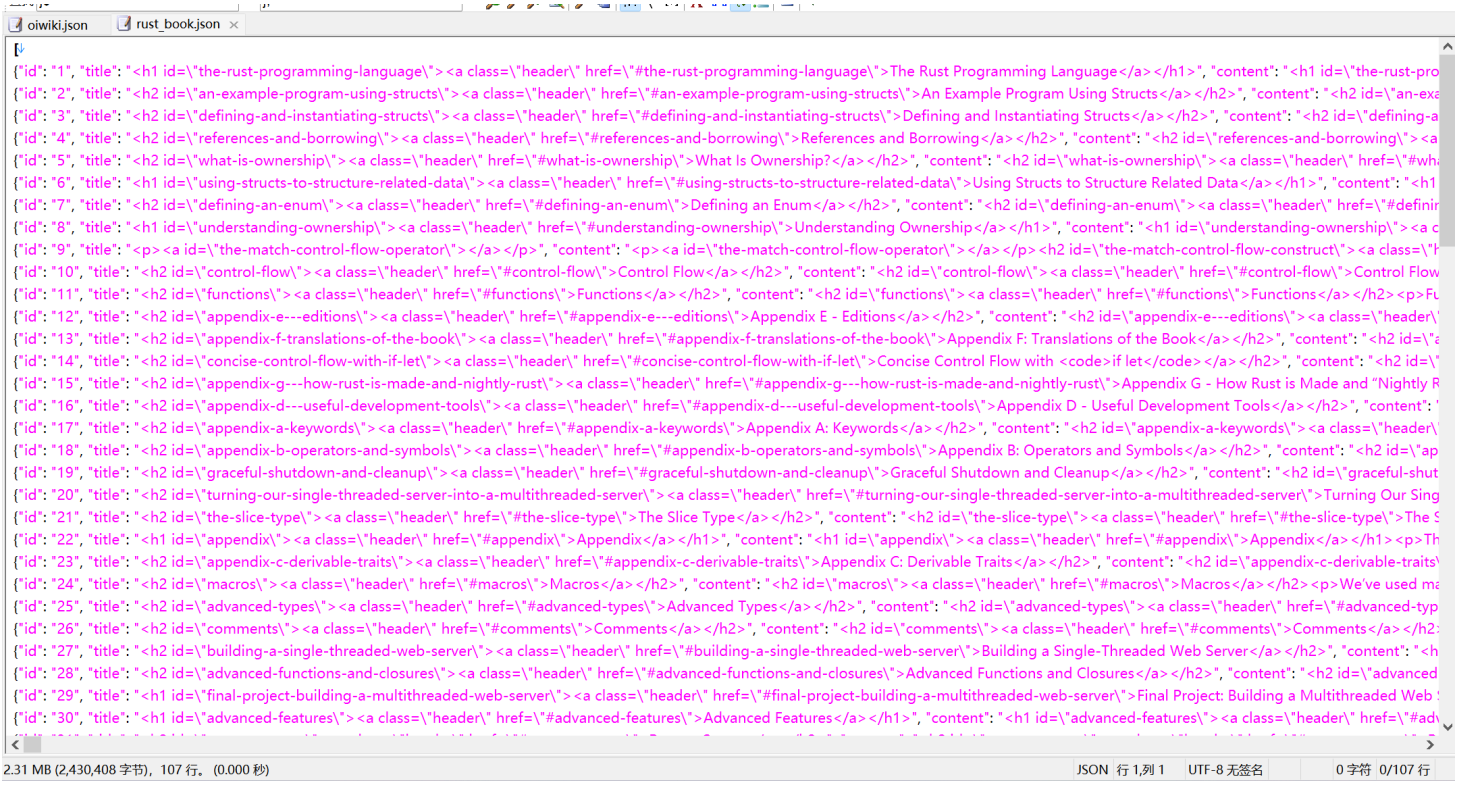
```
def parse(self, response):
    chapters = response.xpath(
        '//ol[@class="chapter"]//li[@class="chapter-item expanded " or @class="chapter-item expa
    )
    hrefs = chapters.xpath("@href").getall()
    texts = chapters.xpath("text()").getall()

    for href, chapter in zip(hrefs, texts):
        url = response.urljoin(href)
        yield scrapy.Request(
            url=url,
            callback=self.parse_chapter,
            cb_kwargs={"chapter": chapter},
        )
def parse_chapter(self, response, chapter="Unknown"):
    content = response.xpath("//main/*")
    keywords = content.xpath("text()").getall()
    content = content.getall()

    self.id = self.id + 1
    yield {
        "id": str(self.id),
        "title": content[0],
        "content": "".join(p for p in content),
        "keywords": "".join(p for p in keywords),
        "url": response.url,
        "date": datetime.date.today().strftime("%Y-%m-%d"),
    }
```

数据存储

为了方便后续索引和检索，我们将所有爬取到的数据都存储成 `json` 文件，每一条的数据的形式如下，保证所有要求的必要信息都会被存储：

```
{
    "id": "1",
    "title": "The Rust Programming Language",
    "content": "The Rust Programming Language",
    "keywords": "The Rust Programming Language",
    "url": "https://doc.rust-lang.org/book/",
    "date": "2022-04-27"
}
```


最终的爬取结果如下：



前端设计与实现

todo hjr 写

后端设计与实现

因为 Go 在各种测试中表现出了优秀的性能水平，所以本次实验后端我使用 Go 语言进行开发，框架使用了 Gin 这一高性能且比较主流的 Web 框架，本部分将介绍本项目的后端接口设计以及相关逻辑实现。

不过 Go 令人最为诟病的一点就是其 `err` 的判断机制，几乎每一次函数调用都要判断函数返回的 `err` 是否需要处理，所以考虑到报告的篇幅长度，我在后续展示 Go 代码时都将删去 `err` 的判断部分，以便带来更加良好的阅读体验。

接口展示

首先展示一下我们此次实验完成的所有接口（包括作业二和作业三）

查询接口

- GET /document 获取文档详细信息
- GET /search 分页查询
- POST /search_by_image 上传图片查询

反馈接口

- POST /entity_feedback 实体反馈
- POST /extract_info_regex_feedback 正则提取反馈 (正则+词性)
- POST /feedback 结果反馈
- POST /hotword_feedback 热词反馈

提取接口

- GET /extract_info 提取关键信息
- GET /extract_info_regex 提取关键信息

```
r.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))

v1 := r.Group("/api/v1")
{
    // Search with keywords
    v1.GET("/search", controller.Search)
    // Fetch SearchResult content details
    v1.GET("/document", controller.GetDocument)
    // Search by image
    v1.POST("/search_by_image", controller.SearchByImage)

    // Entities and hot words
    v1.GET("/extract_info", controller.ExtractInfo)
    // Entity and hot word feedback
    v1.POST("/extract_info_regex", controller.ExtractInfoRegex)

    // Feedback
    v1.POST("/feedback", controller.Feedback)
    // Entity Feedback
    v1.POST("/entity_feedback", controller.EntityFeedback)
    // Hotword Feedback
    v1.POST("/hotword_feedback", controller.HotwordFeedback)
    // Regex Feedback
    v1.POST("/extract_info_regex_feedback", controller.ExtractInfoRegexFeedback)
}
```

关键词搜索

根据关键词时用户提交关键词，请求到达后端后，由 controller 调用对应的 logic 函数实现具体功能。具体实现过程是先查询缓存，缓存未命中则对关键词进行分词，然后调用核心的 `SearchIndex` 函数进行查询， `SearchIndex` 函数的具体实现将会在信息检索服务中详细介绍，查询结束后则将结果放入缓存并向前端返回查询结果。

```go
func Search(q string, page string, resultsPerPage string) (r model.SearchResponse, err error) {
    cacheKey := fmt.Sprintf("%s-%s-%s", q, page, resultsPerPage)

    if cachedResults, found := cache.Get(cacheKey); found {
        return model.SearchResponse{Code: 200, Results: cachedResults}, nil
    }

    intPage, err := strconv.Atoi(page)

    intResultsPerPage, err := strconv.Atoi(resultsPerPage)

    queryWords := WordSplit(q)
    log.Info("queryWords: ", queryWords)

    results, err := SearchIndex(queryWords, intPage, intResultsPerPage)

    cache.Set(cacheKey, results)

    return model.SearchResponse{Code: 200, Results: results}, nil
}
```

图片搜索

图片搜索需要调用 python 写的接口，通过模型识别从图片中提取关键字，然后使用关键词进行搜索。所以在 controller 中，图片搜索首先调用 `SearchByImageLogic` 函数，该函数会调用 python 接口提取图片关键词，使用 python 对图片的处理则会在多媒体关键词提取服务中详细介绍。

```go
func SearchByImageLogic(imagePath string) (string, error) {
    var b bytes.Buffer
    w := multipart.NewWriter(&b)
    f, err := os.Open(imagePath)

    defer f.Close()

    fw, err := w.CreateFormFile("file", filepath.Base(imagePath))
    if _, err = io.Copy(fw, f); err != nil {
        return "", err
    }
    w.Close()

    req, err := http.NewRequest("POST", model.PYTHON_SERVER_URL+"/image_to_keywords", &b)
    req.Header.Set("Content-Type", w.FormDataContentType())

    client := &http.Client{}
    res, err := client.Do(req)

    body, err := ioutil.ReadAll(res.Body)

    var kr KeywordResponse
    err = json.Unmarshal(body, &kr)

    return kr.Keyword, nil
}
```

获取文章内容

考虑到网络传输，每次查询结果都是返回文章概要信息，并不会返回文章的详细内容，所以我另外提供了一个根据文章 `id` 获取文章详细内容的接口，用于返回文章的概要信息和文章的具体内容。

```go
// 直接从map中查询到文章内容并返回即可
func GetFullDoc(id string) (model.Document, bool) {
    doc, ok := idDocMap[id]
    return doc, ok
}
```

信息检索服务

信息检索服务是本次实验的核心内容，在实验中，我在后端服务初始化的过程中加载文档并对其进行预处理。在读入文档之后我对文档进行分词，然后构建倒排索引，同时计算并存储 TF-IDF 值，避免查询时重复计算。在查询时为了提高查询的准确率和性能，我通过计算查询向量和文档向量的余弦相似度作为查询结果的关联度，然后根据关联度对查询结果进行排序并返回结果。

加载文档

首先从指定路径中加载文档，文档格式是之前存的 `json` 文件，同时设置每篇文档的基本信息，如 id、语言等。

```
func LoadDocuments(dir string) ([]Document, error) {
    files, err := ioutil.ReadDir(dir)

    var documents []Document

    for _, file := range files {
        if filepath.Ext(file.Name()) != ".json" {
            continue
        }
        filename := filepath.Join(dir, file.Name())

        f, err := os.Open(filename)
        defer f.Close()

        var docs []Document
        dec := json.NewDecoder(f)
        if err := dec.Decode(&docs); err != nil {
            return nil, err
        }

        setLang(docs, filename)

        documents = append(documents, docs...)

        f.Close()
    }

    for id, doc := range documents {
        doc.Id = strconv.Itoa(id)
        sTitle := strconv.QuoteToASCII(doc.Title)
        doc.Title = sTitle[1 : len(sTitle)-1]
        sUnicode := strconv.QuoteToASCII(doc.Content)
        doc.Content = sUnicode[1 : len(sUnicode)-1]
    }

    return documents, nil
}
```

分词

分词是信息检索服务的基础，对于中英文文档，我均使用 `jieba` 库进行分词并去除停用词防止干扰。

```go
func WordSplit(query string) []string {
    defer func() {
        if panicInfo := recover(); panicInfo != nil {
            log.Errorf("%v, %s", panicInfo, string(debug.Stack()))
        }
    }()

    words := seg.Cut(query, true)

    words = filter(words, stopWords)
    for i := range words {
        words[i] = strings.TrimSpace(words[i])

    }

    return words
}

func filter(slice []string, unwanted []string) []string {
    unwantedSet := make(map[string]any, len(unwanted))
    for _, s := range unwanted {
        unwantedSet[s] = struct{}{}
    }

    var result []string
    for _, s := range slice {
        if _, ok := unwantedSet[s]; !ok {
            result = append(result, s)
        }
    }

    return result
}
```

构建索引

索引是查询的基础，在实验中，我对每篇文档均构建了正向索引和倒排索引，记录文章的 id 映射和该文章中出现的所有词语及其位置信息。具体构建流程如下：

1. 遍历所有文章，构建文章 id 对文章内容的映射关系（正向索引）；
2. 对文章进行分词并去除停用词；
3. 遍历文章中的每个关键词，建立一个关键词到文章内容的索引映射（倒排索引）。

```
    for _, doc := range documents {
        idDocMap[doc.Id] = doc

        words := WordSplit(doc.Keywords)
        for _, word := range words {
            if !isStopWord(word) {
                word = strings.ToLower(word)
                docIndex[word] = append(docIndex[word], doc)
            }
        }
    }
```

计算并存储 TF-IDF 值

本次实验使用 TF-IDF 进行特征提取。词频（term frequency, TF）， 指的是某一个给定的词语在该文件中出现的频率。

$$TF_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

上式中 $n_{i,j}$ 表示第 j 篇文章中出现第 i 个词的频数，而 $\sum_k n_{k,j}$ 表示统计第 j 篇文章中所有词的总数。

逆向文件频率（inverse document frequency, IDF），某一特定词语的 IDF，可以由一类文件中的总文件数目除以该类中包含该词语之文件的数目， 再将得到的商取对数得到。 IDF 是一个词语普遍重要性的度量。

$$IDF_i = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$

上式中 D 表示一个文档的集合，有 $\{d_1, d_2, d_3, \cdots\} \in D$，取模即是计算这个集合中文档的个数。$t_i$ 表示第 i 个词，$j : t_i \in d_i$ 表示第 i 个单词属于文本 d_j，对其取模即是计算包含单词 i 的文本的个数。

TF-IDF 值即是 TF 值与 IDF 值之积， TF-IDF 综合表征了该词在文档中的重要程度和文档区分度。

根据上述算法，我的代码实现如下：

```
// 计算idf
for word := range docIndex {
    if _, ok := idfMap[word]; !ok {
        word = strings.ToLower(word)
        idfMap[word] = math.Log(totalDocs / float64(len(docIndex[word])))
    }
}

// 根据tf-idf计算文档向量，创建tf-idf索引
for _, doc := range documents {
    docVector := buildDocumentVector(doc)
    words := WordSplit(doc.Keywords)

    for _, word := range words {
        word = strings.ToLower(word)
        index[word] = append(index[word], docVector)
    }
}
```

计算文档向量的 `buildDocumentVector` 函数如下：

```
func buildDocumentVector(doc model.Document) DocumentVector {
    vector := make(map[string]float64)
    words := WordSplit(doc.Keywords)
    wordCount := float64(len(words))

    // 计算tf
    for _, word := range words {
        word = strings.ToLower(word)
        vector[word] += 1.0 / wordCount
    }

    // 计算tf-idf
    magnitude := 0.0
    for word, tf := range vector {
        word = strings.ToLower(word)
        tfIdf := tf * idfMap[word]
        vector[word] = tfIdf
        magnitude += tfIdf * tfIdf
    }

    // 归一化
    if magnitude > 0.0 {
        sqrtMagnitude := math.Sqrt(magnitude + epsilon)
        for word := range vector {
            word = strings.ToLower(word)
            vector[word] /= sqrtMagnitude
        }
    }

    return DocumentVector{Doc: doc, Vector: vector}
}
```

检索

本次实验使用余弦相似度来计算两个查询语句和文章的相似度，余弦相似度定义如下：

$$\cos(\theta) = \frac{A \cdot B}{|A|\,|B|}$$

其中 A 和 B 分别是两个向量，θ 是两个向量夹角的弧度。

在项目中我的实现流程如下：

1. 计算查询语句的向量，根据查询词构建 TF-IDF 向量。此过程包括计算每个查询词的 IDF 值，以及归一化查询向量以确保后续相似度计算的准确性；
2. 对于每个查询词，遍历索引中的文档向量，使用 `cosineSimilarity` 函数计算查询向量与文档向量之间的余弦相似度;
3. 根据文档中查询词出现的位置、频率和是否出现在标题中对相似度进行调整，以提高搜索的相关性;
4. 使用协程并行处理这些计算，并通过 channel 收集每个文档的得分，以提高效率；
5. 收集所有得分后，进一步根据查询词在文档中的总出现次数和标题中出现的次数调整最终得分。

详细代码如下：

```
func buildSummaryDocument(doc model.Document) model.SummaryDocument {
    summaryDoc := model.SummaryDocument{
        Id:      doc.Id,
        Title:   doc.Title,
        URL:     doc.URL,
        Date:    doc.Date,
        Content: calculateSummary(doc.Keywords),
    }
    return summaryDoc
}

func SearchIndex(queryWords []string, page, resultsPerPage int) ([]model.SearchResult, error) {
    if len(queryWords) == 0 {
        return nil, errors.New("empty query")
    }

    queryVector := buildQueryVector(queryWords)
    log.Info("queryVector:", queryVector)

    magnigude := 0.0
    for _, tfidf := range queryVector {
        magnigude += tfidf * tfidf
    }
    if magnigude == 0 {
        log.Info("Query made up of words in every or no documents. Returning all documents.")
        results := make([]model.SearchResult, 0, len(docs))
        for _, doc := range docs {
            results = append(results, model.SearchResult{Doc: buildSummaryDocument(doc), Score:
        }

        return results, nil
    }

    vectorCounts := make(map[string]int)
    for _, word := range queryWords {
        word = strings.ToLower(word)
        if vectors, ok := index[word]; ok {
            for _, vector := range vectors {
                vectorCounts[vector.Doc.Id]++
            }
        }
    }
```

```
queryWordCounts := make(map[string]int)
titleQueryWordCounts := make(map[string]int)

var mutex sync.Mutex

scoresChansMap := make(map[string]chan float64)
for id, count := range vectorCounts {
    scoresChansMap[id] = make(chan float64, count)
}

var wg sync.WaitGroup

for _, word := range queryWords {
    word = strings.ToLower(word)
    if vectors, ok := index[word]; ok {
        for _, vector := range vectors {

            wg.Add(1)
            go func(w string, v DocumentVector, scoresChan chan float64) {
                defer wg.Done()

                wi := strings.ToLower(w)

                score := cosineSimilarity(queryVector, v.Vector)

                frequency := float64(len(WordSplit(v.Doc.Keywords)))
                position := float64(strings.Index(v.Doc.Keywords, wi))
                length := float64(len(v.Doc.Keywords))

                adjustment := (1 + math.Log(frequency+1)) * (1 / (1 + math.Log(length+1)) *
                score *= adjustment

                if strings.Contains(v.Doc.Keywords, wi) || strings.Contains(strings.ToLower(
                    mutex.Lock()
                    if strings.Contains(v.Doc.Keywords, wi) {
                        queryWordCounts[v.Doc.Id]++
                    }
                    if strings.Contains(strings.ToLower(v.Doc.Title), wi) {
                        titleQueryWordCounts[v.Doc.Id]++
                    }
                    mutex.Unlock()
                }
                scoresChan <- score
```

```
            }(word, vector, scoresChansMap[vector.Doc.Id])
        }
    }
}

go func() {
    wg.Wait()
    for _, scoresChan := range scoresChansMap {
        close(scoresChan)
    }
}()

scoreMap := make(map[string]*model.SearchResult)
for id, scoresChan := range scoresChansMap {
    totalScore := 0.0
    for score := range scoresChan {
        totalScore += score
    }
    totalScore *= float64(1 + queryWordCounts[id])

    totalScore *= 1.2 * float64(1+titleQueryWordCounts[id])

    summaryDoc := buildSummaryDocument(idDocMap[id])
    scoreMap[id] = &model.SearchResult{Doc: summaryDoc, Score: totalScore}
}

log.Info(len(scoreMap), " results")
log.Debug(">>> scoreMap")
for k, v := range scoreMap {
    log.Debug(k, ":", "Doc:", v.Doc, "Score:", v.Score)
}
log.Debug("<<< scoreMap")

results := make([]model.SearchResult, 0, len(scoreMap))
for _, result := range scoreMap {
    results = append(results, *result)
}

sort.Slice(results, func(i, j int) bool {
    return results[i].Score > results[j].Score
})

start := (page - 1) * resultsPerPage
```

```
    end := start + resultsPerPage
    if start > len(results) {
        start = len(results)
    }
    if end > len(results) {
        end = len(results)
    }

    results = results[start:end]

    return results, nil
}

func buildQueryVector(queryWords []string) map[string]float64 {
    vector := make(map[string]float64)
    wordCount := float64(len(queryWords))

    for _, word := range queryWords {
        word = strings.ToLower(word)
        vector[word] += 1.0 / wordCount
    }

    magnitude := 0.0
    for word, tf := range vector {
        word = strings.ToLower(word)
        idf, ok := idfMap[word]
        if !ok {
            continue
        }
        tfIdf := idf * tf
        vector[word] = tfIdf
        magnitude += tfIdf * tfIdf
    }

    if magnitude > 0.0 {
        sqrtMagnitude := math.Sqrt(magnitude + epsilon)
        for word := range vector {
            vector[word] /= sqrtMagnitude
        }
    }

    return vector
}
```

```
func cosineSimilarity(vector1, vector2 map[string]float64) float64 {
    dotProduct := 0.0
    magnitude1 := 0.0
    magnitude2 := 0.0
    for word, value := range vector1 {
        word = strings.ToLower(word)
        dotProduct += value * vector2[word]
        magnitude1 += value * value
    }
    for _, value := range vector2 {
        magnitude2 += value * value
    }

    sqrtEpsMag1 := math.Sqrt(magnitude1 + epsilon)
    sqrtEpsMag2 := math.Sqrt(magnitude2 + epsilon)
    return dotProduct / (sqrtEpsMag1 * sqrtEpsMag2)
}

func calculateSummary(content string) string {
    if len(content) > 100 {
        return content[:100] + "..."
    }
    return content
}
```

多媒体关键词提取服务

想要实现多媒体的检索服务，首先就要解决从多媒体中提取出关键词的问题，由于 python 在这一方面具有优势，因此我们选择使用 python 完成这一任务。

具体流程是：

1. python 使用 flask 框架接受请求；
2. 将图片输入到 ResNet50 模型中进行对象识别；
3. 将前五个最可能的对象转换成关键词返回；

接口代码如下：

```python
@app.route("/image_to_keywords", methods=["POST"])
def image_to_keywords():
    if "file" not in request.files:
        return "No file part", 400
    file = request.files["file"]

    if file.filename == "":
        return "No selected file", 400
    log.info(file.filename)
    result, code = image_detection.image_to_keywords(file)
    if code != 200:
        log.error(result)
        return result, code
    return result, code
```

模型处理代码如下（错误处理代码则删去不再展示）：

```
config = tf.compat.v1.ConfigProto(
    gpu_options=tf.compat.v1.GPUOptions(allow_growth=True))
sess = tf.compat.v1.Session(config=config)

log = logging.getLogger("ImageToKeywords")

# Image object detection model
model = ResNet50(weights="imagenet")


def image_to_keywords(file: str) -> tuple[str, int]:
    if file.filename == "":
        return ("No selected file", 400)
    log.info(file.filename)

    img = (
        Image.open(io.BytesIO(file.read()))
        .convert("RGB")
        .resize((224, 224))
    )

    x = img_to_array(img)

    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)

    preds = model.predict(x)
    predictions = decode_predictions(preds, top=5)[0]

    if len(predictions) >= 3:
        keywords = [pred[1] for pred in predictions[:3]]
    else:
        keywords = [pred[1] for pred in predictions]
    keywords = " ".join(kw for kw in keywords)
    log.info(keywords)
    return (json.dumps({"keyword": keywords}), 200)
```

检索结果评价

优化与创新性

在本次实验中，通过不断地打磨我们的项目，我们实现了以下优化和创新：

- 缓存优化：后端自己实现了 LRU 缓存，在内存中缓存每次查询的结果，减少了重复的查询计算，提高了查询的性能和效率；
- TF-IDF 归一化：在计算 TF-IDF 值的时候，我对其进行了归一化，减少了文章长度的影响，在使用时我也加了一个小常数 ε 避免 TF-IDF 值为 0；
- 空间向量查询：在检索比较时，我没有使用简单的线性比较，而是通过计算向量余弦值来计算两个空间向量的相似度，这样提高了检索的性能指标；
- 并发优化：在检索时我通过 Go 的协程并行计算每篇文档的得分，这样充分利用了 Go 轻量级协程的优势，大大提高了检索的性能；
- 基于用户反馈动态修改排名：对于每个查询结果，我们都设置用户可以对其进行反馈，并且通过用户反馈修改查询结果的权重，动态地调整查询结果的排名；
- 多媒体检索：在项目中我们引入了图片识别模型，通过识别提取图片关键词，从而实现多媒体检索。

环境和社会可持续发展思考

实验总结

实验分工

	郭晨旭	韩景锐
代码	Go 后端，Scrapy 爬虫，Python 图片提取关键词服务	Vue 前端
报告	项目概述，后端设计与实现，信息检索服务， 多媒体关键词提取服务，优化与创新	前端设计与实现， 环境和社会可持续发展思考