



信息检索系统

MariaInfoRetrieval 信息检索系统

姓名: 吴清柳

学号: 2020211597

班级: 2020211323

指导老师: 刘传昌

May 31, 2023

Contents

1	项目概述	4
1.1	项目目标	4
1.2	项目背景	4
1.3	项目组成部分与技术选型	4
2	数据获取与存储	5
2.1	数据源描述	5
2.2	使用 Scrapy 爬虫获取数据	5
2.2.1	oiwiki 数据爬取	5
2.2.2	Rust book 爬取	8
2.3	数据存储方式	10
3	前端设计与实现	11
3.1	使用 React Typescript 设计前端界面	11
3.2	前端功能与使用方法	12
3.3	前端的实现	12
3.3.1	前端发送文本搜索请求	14
3.3.2	前端发送图片搜索请求	15
3.3.3	翻页	16
3.3.4	SearchBox 搜索框	17
3.3.5	SearchResults 搜索结果展示组件	18
4	后端设计与实现	19
4.1	后端框架与语言选择	19
4.2	后端主要功能实现细节	19
4.3	后端与前端的交互	20
4.4	后端实现	20
4.4.1	搜索 endpoint	20
4.4.2	文档 endpoint	20
4.4.3	以图搜索 endpoint	21
5	信息检索系统的设计与实现	22
5.1	信息检索系统的总体设计	22
5.2	中英文处理方法: 分词作为基本单元	22

5.3	构建基本的倒排索引文件	23
5.4	实现向量空间检索模型的匹配算法	26
5.5	向量相似度计算	30
5.6	执行搜索	31
6	多媒体检索图片检索微服务	33
6.1	图片检索系统的设计	33
6.2	使用 Python Flask 实现图片检索微服务	33
6.3	与主体服务的交互	35
7	检索结果评价	35
7.1	评价方法的设计	35
7.2	评价结果的展示	35
8	环境与社会可持续发展的考虑	35
8.1	项目实施过程中的环保措施	36
8.2	社会可持续发展影响考虑	36
9	优化与创新性思考	36
9.1	优化措施和效果	37
9.2	创新性思考的实现	37
10	结论	37
10.1	项目成果总结	37
10.2	展望和改进	37

1 项目概述

1.1 项目目标

- 本项目的主要目标是设计并实现一个综合信息检索系统，能够对中英文数据进行有效的检索，满足用户对于自然语言输入查询的需求，并能够提供相关度排序的结果输出。
- 可以对检索结果的准确度进行人工评价，并据此调整查询结果排名；
- 此外，为了更全面地满足用户需求，项目进行了多媒体检索相关的尝试，设计并实现了一个图片检索的微服务。
- 所有功能将以用户友好的网页界面提供。

1.2 项目背景

在当今的数字化时代，数据呈现出爆炸性增长。如何从海量信息中快速有效地检索出所需内容，成为了一个重要的问题。为了解决这个问题，我设计并实现了这个信息检索系统。系统基于开源的网络爬虫获取数据，并在此基础上进行本地存储、处理和检索。通过倒排索引和向量空间模型，实现了高效的匹配算法，能够快速返回与查询内容相关的结果。同时，系统也提供了人工评价机制，用户可以对检索结果的准确度进行反馈，进一步提高系统的性能。

此外，出于环境和社会可持续发展的考虑，对于每个查询，在返回响应的时候，使用增量返回的方式。即对于搜索结果列表，每次只返回一页；对于搜索结果详情，仅在展开该搜索列表的时候从后端返回详情。

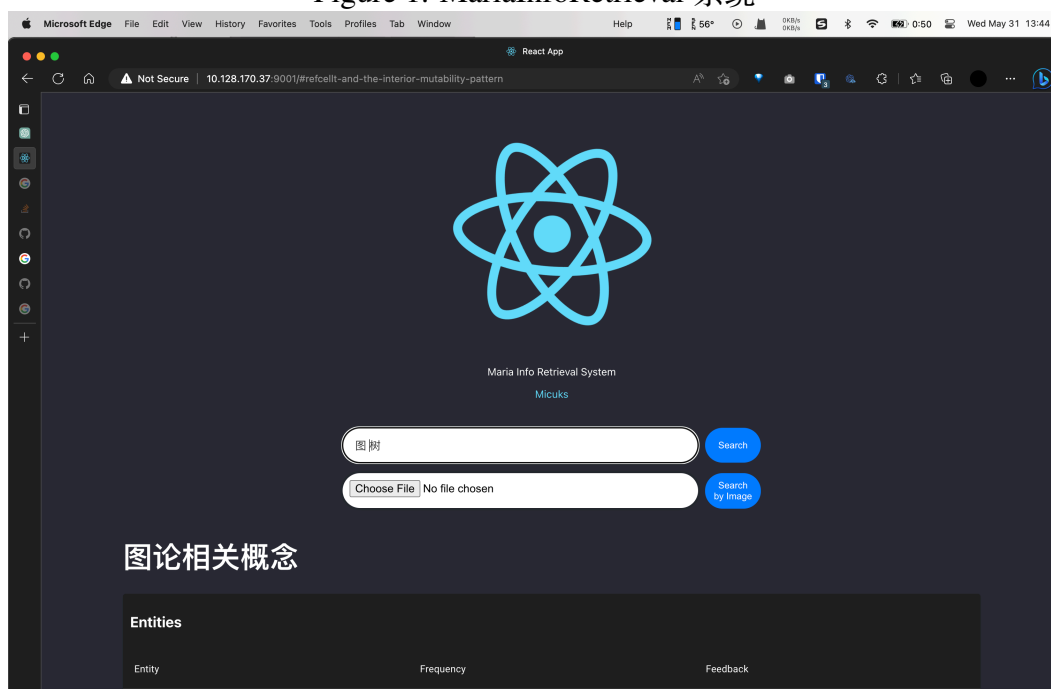
1.3 项目组成部分与技术选型

本项目主要由四个部分组成：数据获取与存储、前端、后端和图片检索微服务。

- 数据获取与存储采用 Scrapy 爬虫工具进行数据爬取，并使用 SQL 数据库进行本地存储。
- 前端部分使用 React 和 TypeScript 实现，为用户提供了清晰直观的操作界面。
- 后端部分基于 Go Gin 框架开发，实现了各类核心功能，如处理用户请求、进行数据处理与检索等。
- 图片检索微服务则使用 Python 的 Flask 框架进行开发，能够有效地对图片数据进行处理与检索。

这些技术选型都基于其各自的优点，如 Scrapy 的爬取能力强大、SQL 的存储安全可靠、React 和 TypeScript 的前端开发效率高、Go Gin 的性能优秀、Flask 的轻量化和易用性等，共同构成了一个高效、稳定的信息检索系，如图 1 和图 2。

Figure 1: MariaInfoRetrieval 系统



2 数据获取与存储

2.1 数据源描述

由于项目支持对中英文进行检索, 因此选取的数据源也是中英双语. 如图 3.

项目的数据源主要分为中文数据和英文数据, 总共包含 535 份文档. 中文文档主要来自于 OiWiki 网站, 这是一个以算法竞赛为主题的中文编程知识库. 英文文档则来自于 Rust 语言的官方教程 The Rust Programming Language, 这是一份详尽的 Rust 语言学习指南.

2.2 使用 Scrapy 爬虫获取数据

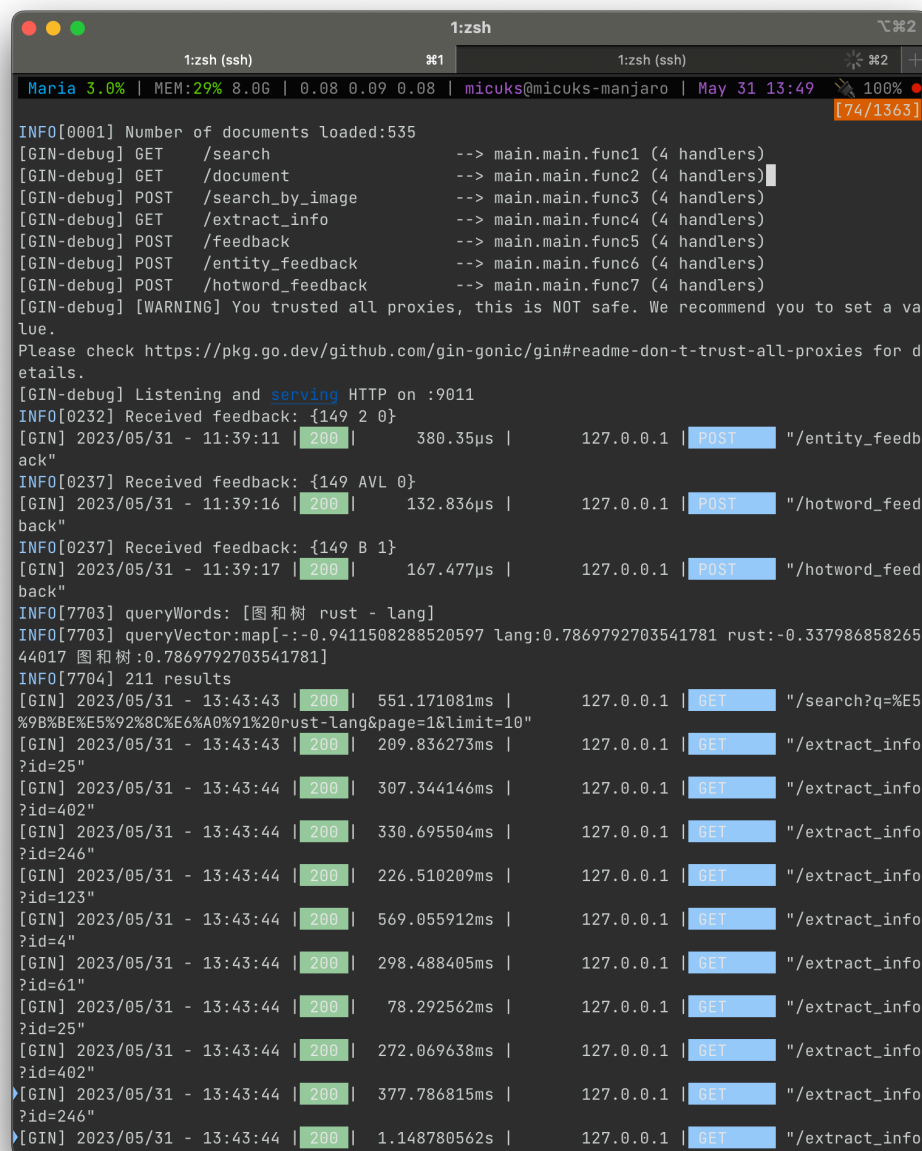
项目使用 Scrapy, 一个强大的开源爬虫框架, 进行数据的获取. 对于中文数据, Scrapy 能够访问 OiWiki 网站的各个页面, 抓取相关文章的标题、内容、URL、发布日期等信息. 对于英文数据, Scrapy 访问 Rust 官方教程的网页, 抓取各章节的内容以及相关信息. Scrapy 的高并发能力和灵活配置使得数据获取过程高效而稳定.

2.2.1 oiwiki 数据爬取

对 oiwiki 数据, 采取首先爬取文章列表, 再对每篇文章分别爬取的方式进行爬取. 具体到每篇文章的爬取, 分别爬取全文和仅文本. 其中, 全文作为信息检索系统向用户展示的内容, 仅文本作为关键字, 用于进行索引和检索.

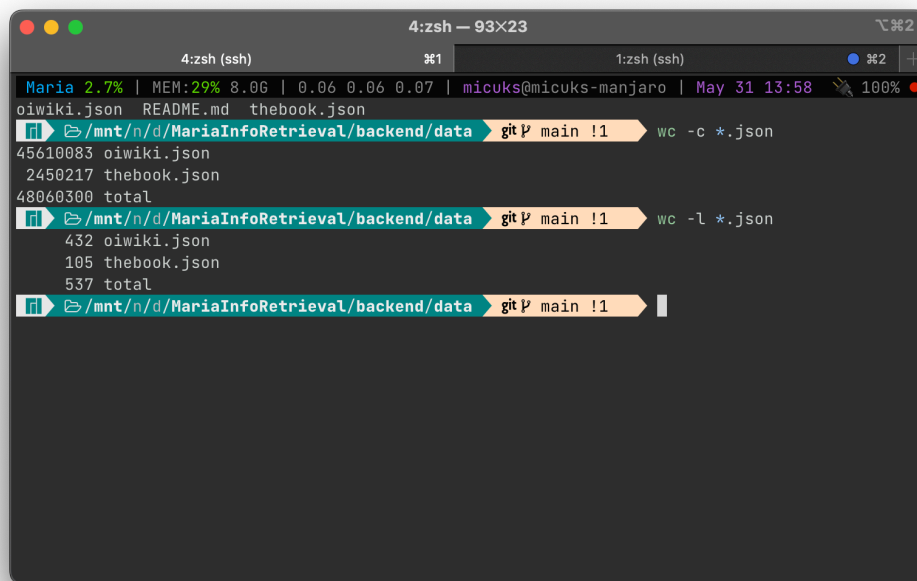
主要代码如下.

Figure 2: 后端运行界面



```
1:zsh
1:zsh (ssh) %1 1:zsh (ssh) %2
Maria 3.0% | MEM:29% 8.0G | 0.08 0.09 0.08 | micuks@micuks-manjaro | May 31 13:49 100% [74/1363]
INFO[0001] Number of documents loaded:535
[GIN-debug] GET /search --> main.main.func1 (4 handlers)
[GIN-debug] GET /document --> main.main.func2 (4 handlers)
[GIN-debug] POST /search_by_image --> main.main.func3 (4 handlers)
[GIN-debug] GET /extract_info --> main.main.func4 (4 handlers)
[GIN-debug] POST /feedback --> main.main.func5 (4 handlers)
[GIN-debug] POST /entity_feedback --> main.main.func6 (4 handlers)
[GIN-debug] POST /hotword_feedback --> main.main.func7 (4 handlers)
[GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to set a value.
Please check https://pkg.go.dev/github.com/gin-gonic/gin#readme-don-t-trust-all-proxies for details.
[GIN-debug] Listening and serving HTTP on :9011
INFO[0232] Received feedback: {149 2 0}
[GIN] 2023/05/31 - 11:39:11 | 200 | 380.35µs | 127.0.0.1 | POST | "/entity_feedback"
INFO[0237] Received feedback: {149 AVL 0}
[GIN] 2023/05/31 - 11:39:16 | 200 | 132.836µs | 127.0.0.1 | POST | "/hotword_feedback"
INFO[0237] Received feedback: {149 B 1}
[GIN] 2023/05/31 - 11:39:17 | 200 | 167.477µs | 127.0.0.1 | POST | "/hotword_feedback"
INFO[7703] queryWords: [图和树 rust - lang]
INFO[7703] queryVector:map[{"rust-lang":0.7869792703541781,"图和树":0.33798685826544017}]
INFO[7704] 211 results
[GIN] 2023/05/31 - 13:43:43 | 200 | 551.171081ms | 127.0.0.1 | GET | "/search?q=%E5%9B%BE%E5%92%8C%E6%A0%91%20rust-lang&page=1&limit=10"
[GIN] 2023/05/31 - 13:43:43 | 200 | 209.836273ms | 127.0.0.1 | GET | "/extract_info?id=25"
[GIN] 2023/05/31 - 13:43:44 | 200 | 307.344146ms | 127.0.0.1 | GET | "/extract_info?id=402"
[GIN] 2023/05/31 - 13:43:44 | 200 | 330.695504ms | 127.0.0.1 | GET | "/extract_info?id=246"
[GIN] 2023/05/31 - 13:43:44 | 200 | 226.510209ms | 127.0.0.1 | GET | "/extract_info?id=123"
[GIN] 2023/05/31 - 13:43:44 | 200 | 569.055912ms | 127.0.0.1 | GET | "/extract_info?id=4"
[GIN] 2023/05/31 - 13:43:44 | 200 | 298.488405ms | 127.0.0.1 | GET | "/extract_info?id=61"
[GIN] 2023/05/31 - 13:43:44 | 200 | 78.292562ms | 127.0.0.1 | GET | "/extract_info?id=25"
[GIN] 2023/05/31 - 13:43:44 | 200 | 272.069638ms | 127.0.0.1 | GET | "/extract_info?id=402"
[GIN] 2023/05/31 - 13:43:44 | 200 | 377.786815ms | 127.0.0.1 | GET | "/extract_info?id=246"
[GIN] 2023/05/31 - 13:43:44 | 200 | 1.148780562s | 127.0.0.1 | GET | "/extract_info?id=246"
```

Figure 3: 爬取数据统计



```

class OiwikiSpiderSpider(scrapy.Spider):
    name = "oiwiki_spider"
    allowed_domains = ["oi-wiki.org"]
    start_urls = ["https://oi-wiki.org/"]
    local_site = "oi-wiki_response.html"
    id = 0

    def start_requests(self):
        urls = [
            "https://oi-wiki.org/contest/",
            "https://oi-wiki.org/tools/",
            "https://oi-wiki.org/lang/",
            "https://oi-wiki.org/basic/",
            "https://oi-wiki.org/search/",
            "https://oi-wiki.org/dp/",
            "https://oi-wiki.org/string/",
            "https://oi-wiki.org/math/",
            "https://oi-wiki.org/ds/",
            "https://oi-wiki.org/graph/",
            "https://oi-wiki.org/geometry/",
            "https://oi-wiki.org/misc/",
            "https://oi-wiki.org/misc/",
        ]

        for i, url in enumerate(urls):
            yield scrapy.Request(url=url, callback=self.parse,
                                ↪ cb_kwargs={})

```

```

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]
    # Number of hrefs should be equal to number of texts
    assert len(hrefs) == len(texts)

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

def parse_section(self, response, section="Unknown"):
    # Scrawl section content
    content = response.xpath(
        '↪ //div[@class="md-content"]//blockquote[1]/preceding-sibling::*[not(se
    )].getall()
    keywords = response.xpath(
        '↪ //div[@class="md-content"]//*[self::h1 or self::h2 or
        ↪ self::h3 or self::h4 or self::li or self::ul or
        ↪ self::p]/text()'
    ).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"),
    }

```

2.2.2 Rust book 爬取

与 oiwiki 的爬取类似, 对 Rust book 的爬取也采取首先爬取文章列表, 然后对每篇文章进行爬取的方式. 同样也分别爬取内容和关键字的方式. 内容展示给用户, 关键字由于构建索引和检索.

具体代码如下.

```
class ThebookSpider(scrapy.Spider):
    name = "thebook"
    allowed_domains = ["doc.rust-lang.org"]
    start_urls = ["http://doc.rust-lang.org/book/"]
    local_site = "rust-book_response.html"
    id = 0

    def start_request(self):
        urls = [] + self.start_urls

        for i, url in enumerate(urls):
            yield scrapy.Request(url=url, callback=self.parse,
                                ↪ cb_kwargs={})

    def parse(self, response):
        chapters = response.xpath(
            '//ol[@class="chapter"]//li[@class="chapter-item expanded "
            ↪ or @class="chapter-item expanded affix "]/a'
        )
        hrefs = chapters.xpath("@href").getall()
        texts = chapters.xpath("text()").getall()
        # Number of hrefs should be equal to number of texts
        assert len(hrefs) == len(texts)

        # Yield scrapy request for each chapter
        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"):
        # Scrawl chapter content
        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,
```

```

    "data": datetime.date.today().strftime("%Y-%m-%d"),
}

```

2.3 数据存储方式

对两个网站爬取的文档, 分别使用单独的 json 文件进行存储. 具体来说, json 字段有 ID, title, content, keywords, date, url. 分别对应文档 ID, 文档标题, 文档内容, 文档关键字, 文档日期和文档 url. 爬取数据示例如图 4.

Figure 4: 爬取数据样例

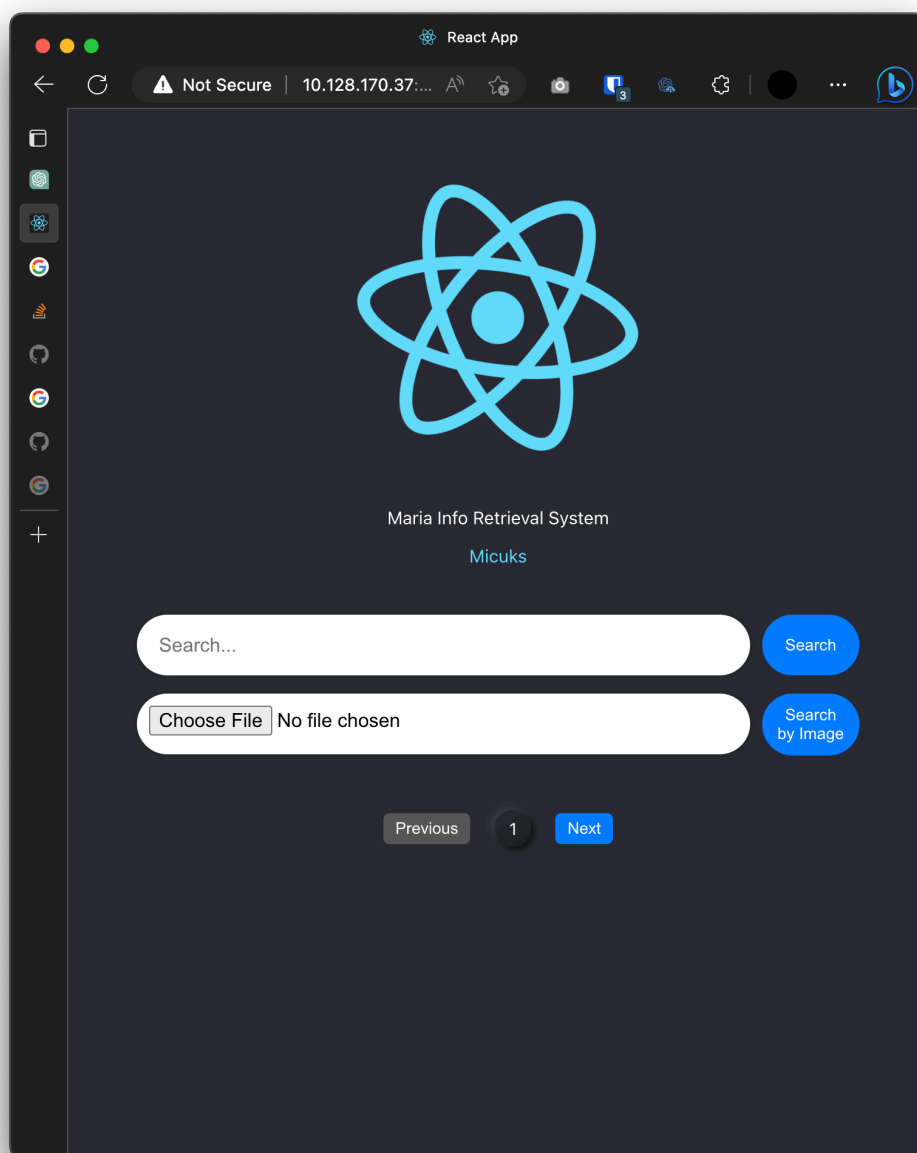
```

thebook.json — Untitled (Workspace)
thebook.json x oiwiki.json extract_info.go serv
MariaInfoRetrieval > spyder > {} thebook.json > ...
You, 7 days ago | 1 author (You)
You, 7 days ago • feat: Rust theBook spyder
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2
```

3 前端设计与实现

3.1 使用 React Typescript 设计前端界面

Figure 5: 前端界面概览



在这个项目中，我选择使用 React TypeScript 作为前端技术栈。React 是一个 JavaScript 库，用于构建用户界面，特别是单页面应用程序，而 TypeScript 则是 JavaScript 的一个超集，增加了静态类型等有用的特性。

React 以其组件化的设计和虚拟 DOM 技术而闻名，这使得我可以更好地组织代码，提高代码的可维护性，同时也能提高页面的渲染效率。TypeScript 则为我提供了强大的类型系统和优秀的工具支持，大大提高了开发效率和代码质量。

在界面设计方面，我力求简洁易用。用户可以在搜索框中输入自然语言查询，

然后点击搜索按钮或者按回车键进行搜索。搜索结果按照相关度排序,并显示相关度、文档标题、主要匹配内容、URL、文档日期等信息。每个搜索结果都有两个反馈按钮,用户可以通过这两个按钮对搜索结果的准确度进行评价。

此外,还支持使用图片进行搜索。用户上传图片后,由 Go 后端调用 Python 微服务进行对象识别,提取关键字后返回 Go 后端,再由 Go 后端根据关键字进行检索,并返回关键字和检索结果。用户可以根据关键字进行翻页。

3.2 前端功能与使用方法

前端提供了以下主要功能:

- 搜索: 用户在搜索框中输入查询,点击搜索按钮或者按回车键,就会收到搜索结果。搜索结果按照相关度排序,显示相关度、文档标题、主要匹配内容、URL、文档日期等信息。
- 反馈: 对于每个搜索结果,用户可以点击""或""按钮,表示认为这个结果与查询高度相关或者不相关。这些反馈信息会被发送到后端,并用于改进搜索算法。
- 展示详细信息: 用户可以点击搜索结果的标题,查看完整的文档内容。文档内容中的代码块会被高亮显示,使得内容更易于理解。
- 实体和热词展示: 对于每个搜索结果,前端还会显示文档中出现的实体和热词,以及它们的出现频率。用户可以对这些实体和热词进行反馈,这些反馈信息也会被发送到后端,用于改进搜索算法。

使用方法如下:

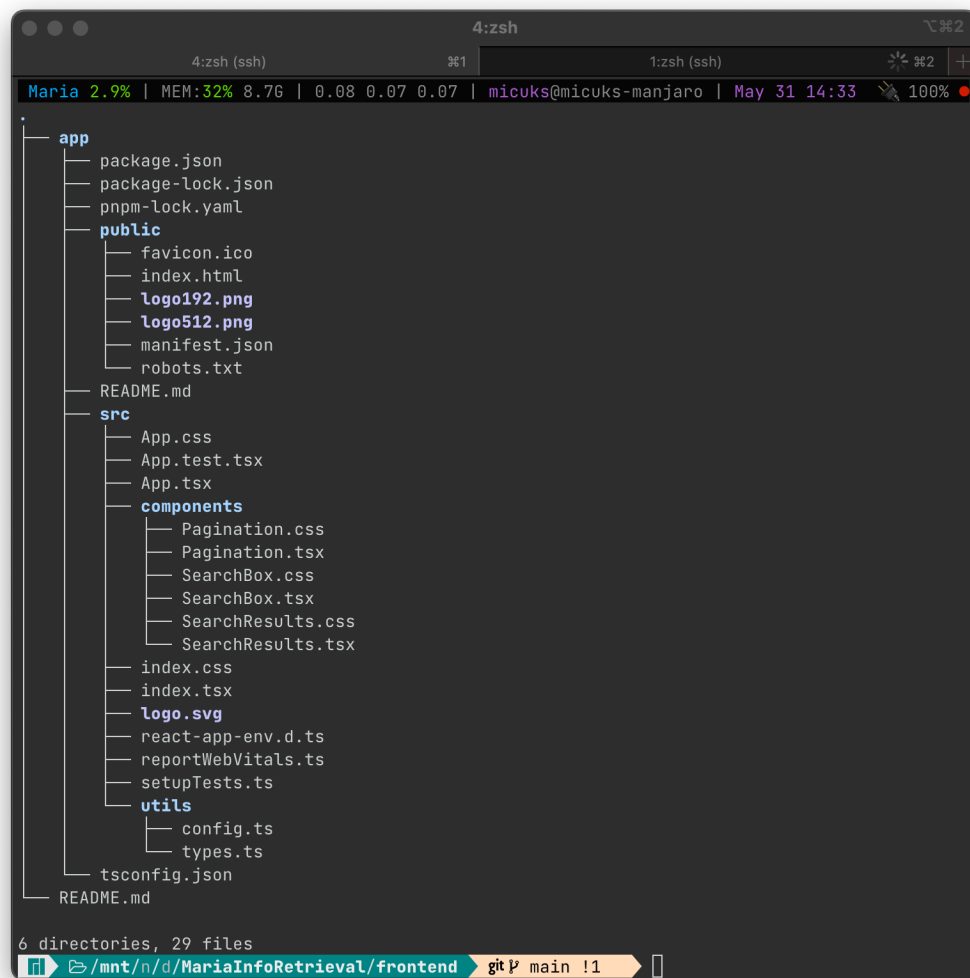
- 在搜索框中输入查询。
- 点击搜索按钮或者按回车键,获取搜索结果。
- 点击搜索结果的标题,查看详细信息。
- 点击正面和负面反馈按钮,对搜索结果进行反馈。
- 在实体和热词列表中,对感兴趣的词进行反馈。

3.3 前端的实现

前端项目结构如图 6.

- 前端的主界面在 App.tsx 中,App.css 中定义了主界面的样式;
- 后端 url 等配置文件放在 utils/config.ts 中;
- 搜索结果和文件摘要的数据结构定义在 utils/types.ts 中;

Figure 6: 前端项目结构

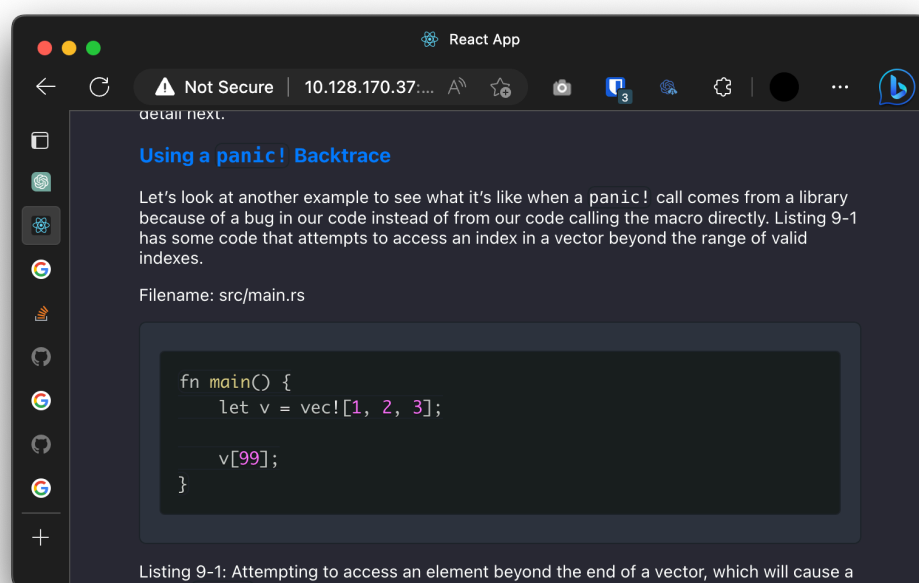


- 搜索结果展示相关的组件在 components/下, 包括翻页组件 ‘Pagination’, 搜索框组件 ‘SearchBox’, 搜索结果组件 ‘SearchResults’, 以及响应的样式;

前端的样式参考了 Apple Design 和 Material Design. 关于组件安排方面, 主界面中加载了搜索框, 搜索结果和翻页组件. 当收到用户的搜索请求后, 对于文字搜索, 使用 GET 方法向后端请求搜索结果, 对于图片搜索, 使用 POST 方法向后端请求搜索结果, 并传给 SearchResults 组件.

在 SearchResults 中, 对获取的文档进行渲染和微调. 例如, 代码块渲染, 使用了 react-syntax-highlighter 将获取文档中的代码块进行渲染, 结合 SearchResults.css 中定义的样式进行美观的展示如图 7.

Figure 7: 代码块渲染



3.3.1 前端发送文本搜索请求

前端可以发送文本和图片搜索请求. 搜索请求结果将以分页形式显示. 请求发送遵循 Lazy loading 的原则, 即每次仅请求本页展示的搜索结果列表.

```
const handleSearch = (searchQuery: string, page: number) => {
  setIsSearching(true);
  fetch(
    ` ${backend_url}/search?q=${searchQuery}&page=${page}&limit=${resultsPerPage}
  )
  .then((response) => response.json())
  .then((data) => {
    setResults(
```

```

        data.map((item: any) => ({
            score: item.Score,
            ...item.Doc,
        })))
    );
    setIsSearching(false);
    setQuery(searchQuery);
    setCurrentPage(page);
})
.catch((error) => {
    console.error("Error:", error);
    setIsSearching(false);
});
};

```

3.3.2 前端发送图片搜索请求

对图片的检索方式与文本检索方式类似, 均为向后端进行请求, 并按结果更新 SearchResults. 不同的是, 根据后端从图片提取的关键词, 修改了前端中搜索框中的关键词. 当用户进行翻页的时候, 改用关键词搜索的方式给出其他页的搜索结果. 减少了能耗, 对环境更加友好.

```

const handleImageSearch = (image: File | null) => {
    setIsSearching(true);

    if (image) {
        const formData = new FormData();
        formData.append("image", image);

        fetch(`${backend_url}/search_by_image`, {
            method: "POST",
            body: formData,
        })
            .then((response) => response.json())
            .then((data) => {
                setResults(
                    data.results.map((item: any) => ({
                        score: item.Score,
                        ...item.Doc,
                    })))
                );
                setIsSearching(false);
                // Fill back keyword
                setQuery(data.keywords);
                setCurrentPage(1);
            })
            .catch((error) => {

```

```

        console.error("Error:", error);
        setIsSearching(false);
    });
}
};

```

3.3.3 翻页

翻页考虑了越界情况. 页下标不会为负数. 翻页后使用新页标向后端请求该页文档.

```

const handlePageChange = (page: number) => {
    if (page <= 0) {
        setCurrentPage(1);
    }

    setCurrentPage(page);
    handleSearch(query, page);
};

```

对于翻页组件的实现, 在 'Pagination.tsx' 中. 其样式定义在 'Pagination.css' 中.

```

const Pagination: React.FC<PaginationProps> = ({
    currentPage,
    onPageChange,
}) => {
    return (
        <div className="Pagination">
            <button
                className="Pagination-button"
                onClick={() => onPageChange(currentPage - 1)}
                disabled={currentPage === 1}
            >
                Previous
            </button>
            <span
                className="Pagination-currentPage">{currentPage}</span>
            <button
                className="Pagination-button"
                onClick={() => onPageChange(currentPage + 1)}
            >
                Next
            </button>
        </div>
    );
};

```


3.3.4 SearchBox 搜索框

搜索框组件使用两个 form 组件分别处理文字搜索和图片搜索. 为了实现图片搜索后替换文字搜索框内容, 将相应状态上移一层到了 App 组件中. 主要实现如下.

```
const SearchBox: React.FC<SearchBoxProps> = ({
  onSearch,
  onImageSearch,
  isSearching,
  setIsSearching,
  query,
  setQuery,
}) => {
  const [file, setFile] = useState<File | null>(null);

  const handleTextSearchSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    setIsSearching(true);
    onSearch(query, 1); // 1 as the initial page number
  };

  const handleImageSearchSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    setIsSearching(true);
    if (file) {
      onImageSearch(file);
    } else {
      console.error("Error searching by image: file not given.");
    }
  };

  return (
    <div className="SearchBox">
      <form onSubmit={handleTextSearchSubmit}>
        /* Search by text */
        <input
          type="text"
          value={query}
          onChange={(e) => setQuery(e.target.value)}
          placeholder="Search..."
        />
        <button type="submit" disabled={isSearching}>
          Search
        </button>
      </form>

      <form onSubmit={handleImageSearchSubmit}>
        /* Search by image */
```

```

        <input
          type="file"
          accept="image/*"
          onChange={(e) => setFile(e.target.files?.item(0) ||
            ↪ null)}
          id="fileUpload"
        />
        <button type="submit" disabled={isSearching}>
          Search<br></br>by Image
        </button>
      </form>
    </div>
  );
};

```

3.3.5 SearchResults 搜索结果展示组件

每个搜索结果单独渲染在一个 SearchResultItem 组件中. 一页中所有搜索结果整合在 SearchResults 组件中. 对每个组件, 对获取的内容进行处理. 例如, 进行代码高亮渲染.

对于整个 SearchResults 组件, 除了展示 SearchResultItem 之外, 还实现了基于用户的反馈调节搜索结果排名的功能.

```

const SearchResults: React.FC<SearchResultsProps> = ({ results })
  ↪ => {
  const [adjustResults, setAdjustResults] = useState(results);

  useEffect(() => {
    setAdjustResults(results);
  }, [results]);

  const handleFeedback = (resultId: string, score: number) => {
    let newResults = [...adjustResults];

    let index = newResults.findIndex((result) => result.id ===
      ↪ resultId);
    if (score === 1) {
      if (index > 0) {
        // Swap the item with the one before it to move it up
        [newResults[index], newResults[index - 1]] = [
          newResults[index - 1],
          newResults[index],
        ];
      }
    } else if (score === 0 && index < newResults.length - 1) {
      // Swap the item with the one after it to move it down
      [newResults[index], newResults[index + 1]] = [

```

```
        newResults[index + 1],
        newResults[index],
    ];
}

setAdjustResults(newResults);
};

console.debug(`Generate SearchResults for`);
console.debug(results);

return (
    <div>
        {adjustResults.map((result, index) => (
            <SearchResultItem
                key={`-${result.id}-${index}`}
                result={result}
                onFeedback={handleFeedback}
            />
        ))}
    </div>
);
};
```

4 后端设计与实现

4.1 后端框架与语言选择

本项目的后端部分选择使用 Go 语言进行开发，具体采用了 Gin 框架。选择 Go 的原因主要有以下几点：首先，Go 语言具有简洁清晰的语法，对并发编程支持良好，这使得我能够更高效地处理大量的用户请求。其次，Go 语言的性能优秀，能够有效地利用服务器资源。最后，Go 语言的静态类型系统可以帮助我在编程阶段就发现可能的错误，提高代码质量。

Gin 是一个用 Go 语言编写的 HTTP web 框架，它被广泛应用在开发高效且可扩展的 web 应用中。Gin 的路由功能强大且易用，中间件机制灵活，使得我能够轻松地实现各种复杂的功能。

4.2 后端主要功能实现细节

后端主要负责处理前端发送过来的请求，主要包括查询请求和用户反馈请求。以下是后端的主要功能：

- 处理查询请求：当后端接收到一个查询请求时，首先会将查询字符串进行处理，然后使用信息检索算法在数据库中搜索相关的文档，最后将搜索结果返回给前端。

- 处理用户反馈：当后端接收到一个用户反馈时，会根据反馈的内容调整相关文档的评分。例如，如果用户给出了负面反馈，那么相关文档的评分就会被降低。
- 数据存储：后端还负责在数据库中存储文档数据。为了提高搜索效率，我还为每个文档建立了倒排索引。
- 错误处理：在处理请求时，我考虑到了可能会发生的错误。例如，如果查询字符串无法处理，或者用户反馈无效，后端就会返回一个错误消息。

4.3 后端与前端的交互

后端与前端之间的交互主要通过 HTTP 请求实现。当用户在前端输入一个查询或给出一个反馈时，前端就会发送一个 HTTP 请求到后端。后端收到请求后，会进行相应的处理，然后返回一个 HTTP 响应。前端收到响应后，会根据响应的内容更新用户界面。

为了简化后端与前端之间的通信，我在后端设置了一系列的 API 接口。每个接口都对应一个特定的功能，比如搜索、反馈等。这样，前端就可以通过发送请求到特定的接口，实现特定的功能。

4.4 后端实现

4.4.1 搜索 endpoint

对于前端的搜索请求，通过后端的 ‘/search’ endpoint 进行处理。首先提取 q 字段的搜索请求，page 字段的页数，以及 limit 字段的每页展示结果数量，然后调用 ‘PerformSearch’ 函数进行搜索。如果搜索没有出现错误，将搜索结果返回。

```
r.GET("/search", func(c *gin.Context) {
    result, err := query_process.PerformSearch(c.Query("q"),
        ↪ c.Query("page"), c.DefaultQuery("limit",
        ↪ strconv.Itoa(defaultResultsPerPage)))
    if err != nil {
        c.JSON(result.Code, err.Error())
        return
    }

    c.JSON(200, result.Results)
})
```

4.4.2 文档 endpoint

搜索返回的结果不会返回每篇文章的全文，而是只返回文档的概述，大约 100 字。对于文档的细节，仅当用户点击文档标题试图展开的时候才会从后端获取。

```
r.GET("/document", func(c *gin.Context) {
    id := c.Query("id")
```

```

// Search for the document with the given id in docs
doc, found := query_process.GetFullDoc(id)
if !found {
    c.JSON(404, gin.H{"error": "Document" + id + " not found"})
    return
}

c.JSON(200, doc)
})

```

4.4.3 以图搜索 endpoint

对于以图搜索功能, Go 后端使用 ‘/search_by_image’ 接口进行处理. 收到前端提交的图片文件后, 保存在 ‘/tmp/MariaInfoRetrieval’ 缓存目录下, 通过函数 ‘GetKeywordsFromImage’ 调用 Python 微服务, 进行对象识别, 提取关键字. 之后使用关键字调用 ‘PerformSearch’, 按照与前面的搜索过程一样的方式进行搜索. 返回结果除了搜索结果列表之外, 还有提取的关键字. 这个关键字将成为翻页的时候搜索的依据.

```

// Search by image
r.POST("/search_by_image", func(c *gin.Context) {
    file, _ := c.FormFile("image")
    // Save file to the server
    dst := "/tmp/MariaInfoRetrieval/" + file.Filename
    c.SaveUploadedFile(file, dst)

    // Call a function to send this image to the python service,
    ↪ and get
    // back the keywords
    keywords, err := query_process.GetKeywordsFromImage(dst)
    if err != nil {
        c.JSON(500, gin.H{"error": err.Error()})
        return
    }

    // Then use these keywords to perform a search
    result, err := query_process.PerformSearch(keywords,
        ↪ strconv.Itoa(1), strconv.Itoa(defaultResultsPerPage))
    if err != nil {
        c.JSON(result.Code, err.Error())
        return
    }
    c.JSON(200, gin.H{"results": result.Results, "keywords":
        ↪ keywords})
})

```

对执行搜索, 进行索引等算法实现的介绍, 将在后面的信息检索系统章节中给出.

5 信息检索系统的设计与实现

5.1 信息检索系统的总体设计

本项目的信息检索系统主要由四个部分组成：数据获取、文档处理、索引构建和查询处理。数据获取部分通过 Scrapy 爬虫从网络中获取大量的中英文文档。文档处理部分负责对这些文档进行预处理，包括中英文分词等。索引构建部分将处理后的文档构建成倒排索引，便于查询处理部分进行快速搜索。在构建倒排索引的时候，同时存储了 TF-IDF 分数，避免了在进行文档查询的时候进行重复计算。

5.2 中英文处理方法：分词作为基本单元

对于中文文档，我使用了 jieba 分词工具进行分词，将文档切分为一个个单独的词语。对于英文文档，我直接使用空格进行分词。通过分词，我将复杂的文本信息转化为了一系列可以直接处理的基本单元。

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

    words := seg.Cut(query, true) // Use Jeba for Chinese text
    ↪ segment

    words = whitespaceFilter(words)
    for i := range words {
        words[i] = strings.TrimSpace(words[i])
    }

    return words
}
```

此外，对于分词的结果，剔除了多余的空格。对于不想要进行查询的词汇，则是在建立索引的时候就已经剔除。以下为删除分词字符串中空格的代码。

```
func filter(slice []string, unwanted ...string) []string {
    unwantedSet := make(map[string]struct{}, 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
}

func whitespaceFilter(slice []string) []string {
    unwanted := []string{" ", "\t"}
    return filter(slice, unwanted...)
}

```

5.3 构建基本的倒排索引文件

倒排索引是信息检索中的一种重要技术。我为每个文档构建了一个倒排索引，记录了该文档中出现的所有词语及其位置信息。通过倒排索引，可以在查询处理时，快速找到包含特定词语的所有文档。具体过程描述如下

首先，系统存储了所有的文档并定义了一个文档索引，该索引以关键词为键，以包含该关键词的文档列表为值。

然后，系统遍历所有文档，将文档的关键词分解为单词，并加入到索引中。此处过滤了一些停用词，这些词在文档中非常常见，如标点符号，因此无法提供有用的索引信息。

```

// Store documents
docs = documents

totalDocs := float64(len(documents))
docIndex := make(map[string] []Document)

log.Info("Number of documents loaded:", totalDocs)

// First, build the index, and doc's summary at the same time
for _, doc := range documents {
    // Build the idDocMap
    idDocMap[doc.Id] = doc

    // Build the index
    words := WordSplit(doc.Keywords)
    for _, word := range words {
        if !isStopWord(word) {
            // To lower case
            word = strings.ToLower(word)
            docIndex[word] = append(docIndex[word], doc)
        }
    }
}

```


接着，系统会计算所有单词的逆文档频率 (IDF)。IDF 是一个重要的量化指标，表示一个词语在文档集中的重要程度，其值随着词语在文档集中出现的次数的增加而减小。为了避免搜索时的大小写影响搜索结果，在搜索和建立索引的时候都将字母转为小写。

```
// Second, calculate the IDF values for all words
for word := range docIndex {
    if _, ok := idfMap[word]; !ok {
        word = strings.ToLower(word)
        idfMap[word] = math.Log(totalDocs /
            ↪ float64(len(docIndex[word])))
    }
}
```

然后，系统构建文档向量。每个文档都有一个向量表示，向量的每个元素都与一个单词相对应。该元素的值是该单词在文档中的词频 (TF) 乘以其在文档集中的逆文档频率 (IDF)，即 TF-IDF 值。最后，所有的 TF-IDF 值都被标准化，以便在不同长度的文档之间进行比较。

```
// Third, build index of []DocumentVector
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)
    }
}
```

其中的关键步骤是对每篇文章执行 ‘buildDocumentVector’ 来构建用于检索的向量。这个向量描述了文档中各个关键词的重要性。这个函数的主要步骤包括：

1. 初始化一个空的向量 vector，并将文档的关键词进行拆分，得到的结果是一个词汇列表 words。
2. 遍历词汇列表，对每个词汇 word 计算它的词频 (TF, Term Frequency)。词频计算公式是该词在文档中的出现次数除以文档的总词数（这里是 word-Count）。每次出现的词汇，都会使向量中对应词汇的值加上 $1.0/\text{wordCount}$ 。注意，所有的词汇都会被转化为小写形式，以确保处理过程中的一致性。
3. 在计算完词频后，函数接着计算每个词汇的 TF-IDF 值。TF-IDF 是 Term Frequency-Inverse Document Frequency 的缩写，它是信息检索和文本挖掘中常用的一种加权技术。TF-IDF 值是 TF 值和 IDF 值的乘积，其中，IDF 值是该词在所有文档中的逆文档频率，计算方式是文档总数除以含有该词的文档数，然后取对数。在这个函数中，IDF 值是预先计算好存储在 idfMap 这个全局变量中的。TF-IDF 值可以表示一个词在一篇文章中的重要程度。

4. 为了防止文档长度对计算结果的影响，需要将得到的 TF-IDF 向量进行归一化。归一化的过程是将向量除以它的模长（向量的模长是向量元素平方和的平方根），得到的结果向量是一个单位向量。在这个函数中，为了防止模长为零造成的数值错误，设置了一个非常小的常数 epsilon，在计算模长的时候加上这个常数。
5. 最后，返回一个 DocumentVector 结构，包含了文档本身以及构建好的向量。

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

    // Calculate TF for each term in the document
    for _, word := range words {
        word = strings.ToLower(word)
        vector[word] += 1.0 / wordCount
    }

    // Multiply TF with IDF for each term to get TF-IDF score and
    // → normalize the
    // vector
    magnitude := 0.0
    for word, tf := range vector {
        word = strings.ToLower(word)
        tfIdf := tf * idfMap[word]
        vector[word] = tfIdf
        magnitude += tfIdf * tfIdf
    }

    if magnitude > 0.0 {
        sqrtMagnitude := math.Sqrt(magnitude + epsilon)

        // Divide each term's TF-IDF score with the magnitude to
        // → get the unit vector
        for word := range vector {
            word = strings.ToLower(word)
            vector[word] /= sqrtMagnitude
        }
    }

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

最后，系统会根据每个文档的关键词生成一个新的文档向量，并将这个向量添加到相应的索引中。

在检索过程中，首先将查询字符串分解为单词，并构建查询向量。然后，系统将计算查询向量与每个文档向量的余弦相似度，作为查询结果的相关度。相关度较高的文档将被作为查询结果返回给用户。

这个索引构建过程包含了信息检索中的几个核心技术，如分词、停用词过滤、倒排索引和 TF-IDF 权重计算，以及使用余弦相似度计算查询与文档之间的匹配程度。这种方法可以处理大量文档，并提供快速精确的查询结果。

5.4 实现向量空间检索模型的匹配算法

在查询处理部分，我实现了向量空间模型的匹配算法。首先，我将查询字符串和文档都转化为词频向量，然后计算查询向量和文档向量的余弦相似度，作为查询结果的相关度。这种方法既考虑了查询词语在文档中的频率，也考虑了查询词语之间的关系，因此能够获得较好的查询效果。

该匹配算法借助了 go 的并发特点，对所有文档进行并发查询，查询速度很快。详细介绍如下。

1. 首先，检查查询词的长度。如果查询词为空，即长度为 0，那么返回一个错误，因为空的查询词没有意义。
2. 使用 `buildQueryVector` 函数根据查询词创建一个查询向量 `queryVector`。
3. 计算查询向量的模长 `magnitude`。如果模长为 0，说明查询词在所有文档中都存在或都不存在。在这种情况下，将返回所有文档。
4. 根据查询词创建一个 `vectorCounts` 映射，用于记录每个文档中的查询词的数量。
5. 接下来，对于每一个查询词，都会在索引 `index` 中找到其对应的文档向量，然后记录下每个文档的查询词数量。这一步操作在多线程中进行，以提高计算效率。
6. 然后，使用余弦相似度计算每个文档向量和查询向量之间的相似度，得到一个原始的评分 `score`。
7. 之后，根据查询词在文档中的频率、文档长度和查询词在文档中的位置对原始评分进行调整。其中，查询词的频率、文档长度和查询词的位置都对评分有所影响，查询词的频率越高、文档长度越短、查询词的位置越靠前，评分越高。
8. 在每个线程中，如果查询词出现在文档中，那么会更新查询词在文档中的数量 `queryWordCounts` 和标题中的数量 `titleQueryWordCounts`。这一步需要使用互斥锁保证线程安全。
9. 等待所有的查询线程完成，然后将结果存储到 `scoreMap` 中。
10. 最后，将评分映射 `scoreMap` 转换成切片，然后根据评分排序。这样，评分最高的搜索结果将会出现在最前面。
11. 根据页码和每页的结果数量应用分页，然后返回分页后的搜索结果。

本算法设计充分考虑了查询的性能和结果的准确性，通过多线程计算，提高了查询的效率，通过对评分的调整，提高了搜索结果的准确性。

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

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

    // Handle the situation when the query words exist in all or
    ↳ none documents
    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([]SearchResult, 0, len(docs))
        for _, doc := range docs {
            results = append(results, SearchResult{Doc:
            ↳ buildSummaryDocument(doc), Score: 1.0})
        }

        return results, nil
    }

    vectorCounts := make(map[string]int)
    // Count the total number of vectors for each document ID
    ↳ across all query words
    for _, word := range queryWords {
        word = strings.ToLower(word)
        if vectors, ok := index[word]; ok {
            for _, vector := range vectors {
                vectorCounts[vector.Doc.Id]++
            }
        }
    }

    // Store the count of query words in each document. Let
    ↳ documents that
    // include more query words get a higher score
    queryWordCounts := make(map[string]int)
    // Count of query words in each doc's title
    titleQueryWordCounts := make(map[string]int)
```

```

// Mutex to protect concurrent writes to queryWordCounts
var mutex sync.Mutex

// Parallel result computation
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()

                // Ignore upper case
                wi := strings.ToLower(w)

                // Calculate score
                score := cosineSimilarity(queryVector,
                    v.Vector)

                // Adjust the score based on:
                // - frequency of query words
                // - document length
                // - position of first query word
                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)) * (1 / (1 +
                    math.Log(position+1))))
                score *= adjustment

                // Increase the count of query words in this
                document
            }(word, vector, scoresChan)
        }
    }
}

```

```

        if strings.Contains(v.Doc.Keywords, wi) ||
        ↪ strings.Contains(strings.ToLower(v.Doc.Title),
        ↪ wi) {
            // Guard the write with the mutex
            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])
}
}

// Wait for all goroutines to finish, then close the results
↪ channel
go func() {
    wg.Wait()
    for _, scoresChan := range scoresChansMap {
        close(scoresChan)
    }
}()

// Collect the results
scoreMap := make(map[string]*SearchResult)
for id, scoresChan := range scoresChansMap {
    totalScore := 0.0
    for score := range scoresChan {
        totalScore += score
    }
    // Boost the score of the document based on the number of
    ↪ query words it contains
    totalScore *= float64(1 + queryWordCounts[id])

    // Boost the score based on the number of query words in
    ↪ title
    totalScore *= 1.2 * float64(1+titleQueryWordCounts[id])

    // Build document summary
    summaryDoc := buildSummaryDocument(idDocMap[id])

```

```

        scoreMap[id] = &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")

    // Convert the scoreMap to a slice
    results := make([]SearchResult, 0, len(scoreMap))
    for _, result := range scoreMap {
        results = append(results, *result)
    }

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

    // Apply pagination
    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
}

```

5.5 向量相似度计算

使用 ‘cosineSimilarity’ 函数计算文档向量和查询向量的相似度。

主要步骤如下：

1. 初始化 dotProduct、magnitude1 和 magnitude2 为 0，分别表示点积、向量 1 和向量 2 的模长。
2. 遍历向量 1 的每一个元素，计算点积和向量 1 的模长。在计算点积时，需要用到向量 2 中对应的元素。注意，单词都被转换成了小写形式，以避免

因大小写不一致而产生的问题。

3. 遍历向量 2 的每一个元素，计算向量 2 的模长。
4. 计算向量 1 和向量 2 的模长的平方根并加上一个非常小的值 epsilon，以防止除以 0 的错误。
5. 返回点积除以两个向量的模长的乘积，得到余弦相似度。

余弦相似度的取值范围是-1 到 1，其中 1 表示完全相同，0 表示完全不相关，-1 表示完全相反。在这个函数中，由于输入的向量都是非负的（因为是 TF-IDF 值），所以计算得到的余弦相似度也是非负的。

```
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)
}
```

5.6 执行搜索

执行搜索的函数为 'PerformSearch'，函数接受三个参数：查询字符串 q、页码 page 以及每页的结果数量 resultsPerPage。

以下是这个函数的主要步骤：

1. 首先，函数会根据参数生成一个缓存键 cacheKey。如果缓存中已经有了这个键对应的搜索结果，函数就直接返回这个结果。
2. 如果缓存中没有找到结果，函数就进行搜索。首先，它将页码 page 和每页结果数量 resultsPerPage 从字符串转换为整数。如果转换过程中出错，函数就返回一个错误代码 400，表示请求的格式不正确。
3. 函数然后对查询字符串 q 进行分词，得到查询单词列表 queryWords。

4. 函数调用 SearchIndex 函数，使用查询单词列表、页码和每页结果数量进行搜索，并计算得分。如果在这个过程中出现错误，函数就返回错误代码 500，表示服务器内部错误。
5. 最后，函数将搜索结果存入缓存，并返回一个状态代码 200 和搜索结果。

```
// Global search result cache
var cache_capacity = 10
var cache = NewCache(cache_capacity)

func PerformSearch(q string, page string, resultsPerPage string) (r
    ↪ SearchResponse, err error) {
    cacheKey := fmt.Sprintf("%s-%s-%s", q, page, resultsPerPage)

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

    // Else search
    intPage, err := strconv.Atoi(page)
    if err != nil {
        return SearchResponse{Code: 400}, errors.New("Invalid page
            ↪ number")
    }

    intResultsPerPage, err := strconv.Atoi(resultsPerPage)
    if err != nil {
        return SearchResponse{Code: 400}, errors.New("Invalid number of
            ↪ results per page")
    }

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

    // Search the index and calculate scores
    results, err := SearchIndex(queryWords, intPage,
        ↪ intResultsPerPage)
    if err != nil {
        return SearchResponse{Code: 500}, errors.New("error fetching
            ↪ documents")
    }

    // Store search results in cache
    cache.Set(cacheKey, results)

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


总的来说，这个函数实现了一个分页搜索引擎，同时使用了缓存来提高效率。如果搜索结果已经在缓存中，就直接返回，否则就进行搜索并将结果存入缓存，以便下次使用。

6 多媒体检索图片检索微服务

以下是图片检索微服务的设计与实现的主要部分：

6.1 图片检索系统的设计

图片检索系统主要利用深度学习的技术，通过 ResNet50 模型对上传的图片进行对象识别，从而获取图片中的关键字。这种方法不仅能够处理图片的内容，还能够理解图片中的对象和场景，因此能够获取更准确的关键字。

6.2 使用 Python Flask 实现图片检索微服务

使用 Python 的 Flask 框架实现了图片检索的微服务。用户可以通过 POST 请求将图片上传到"/image_to_keywords" 接口，微服务会接收到这个请求，读取其中的图片文件，然后将图片输入到 ResNet50 模型中，进行对象识别。识别结果会返回前五个最可能的对象，将这些对象转化为关键字，然后返回给用户。

以下是接口 API Endpoint 定义

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

    # if user does not select file, browser submits an empty part
    ↪ without
    # filename
    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
```

以下是对象识别代码

```
def image_to_keywords(file: str) -> tuple[str, int]:
    if file.filename == "":
```

```
        return ("No selected file", 400)
log.info(file.filename)

# Open image
try:
    img = (
        Image.open(io.BytesIO(file.read()))
        .convert("RGB")
        .resize((224, 224))
    )
except Exception as e:
    msg = "Failed to load image: " + str(e)
    log.error(msg)
    return (msg, 500)

# Image to array
try:
    x = img_to_array(img)
except Exception as e:
    msg = "Failed to convert image to array: " + str(e)
    log.error(msg)
    return (msg, 500)

# Image preprocess
try:
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
except Exception as e:
    msg = "Failed to preprocess image: " + str(e)
    log.error(msg)
    return (msg, 500)

# Image detection
try:
    preds = model.predict(x)
    predictions = decode_predictions(preds, top=5)[0]
    # Return top three predictions or all if predictions are less
    ↳ than three
    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)
except Exception as e:
    msg = "Failed to process image: " + str(e)
    log.error(msg)
```

```
return (msg, 500)
```

6.3 与主体服务的交互

图片检索微服务作为一个独立的微服务，可以与主体服务（即信息检索服务）进行交互。当用户在主体服务中上传图片时，主体服务会将这个请求转发给图片检索微服务。图片检索微服务接收到请求后，对图片进行处理，获取关键字，然后返回给主体服务。主体服务再将这些关键字返回给前端，用于展示给用户或进行其他操作。

以上就是图片检索微服务的设计与实现的主要内容。通过这个微服务，成功地将多媒体检索的功能集成到了信息检索系统中，从而使的系统能够处理更多类型的信息，提供更全面的服务。

7 检索结果评价

以下是关于检索结果评价的设计与实现的主要部分：

7.1 评价方法的设计

为了能准确评估信息检索系统的性能，设计了一个评价方法。该方法的基本思路是，对每一个用户的查询，都会保存用户的反馈，包括用户是否满意检索结果，以及用户对每个检索结果的评分。这样，可以通过统计所有用户的反馈，来评估检索系统的性能。

7.2 评价结果的展示

在用户界面上，为每个检索结果都添加了一个反馈按钮，用户可以通过这个按钮来对检索结果进行评价。在用户提交评价后，会将评价结果保存在服务器端，并用于更新检索系统的性能评价。在后台管理界面中，为管理员提供了一个查看检索系统性能评价的功能，管理员可以查看到系统的准确率和召回率，并根据这些信息对检索系统进行调优。如图 8。

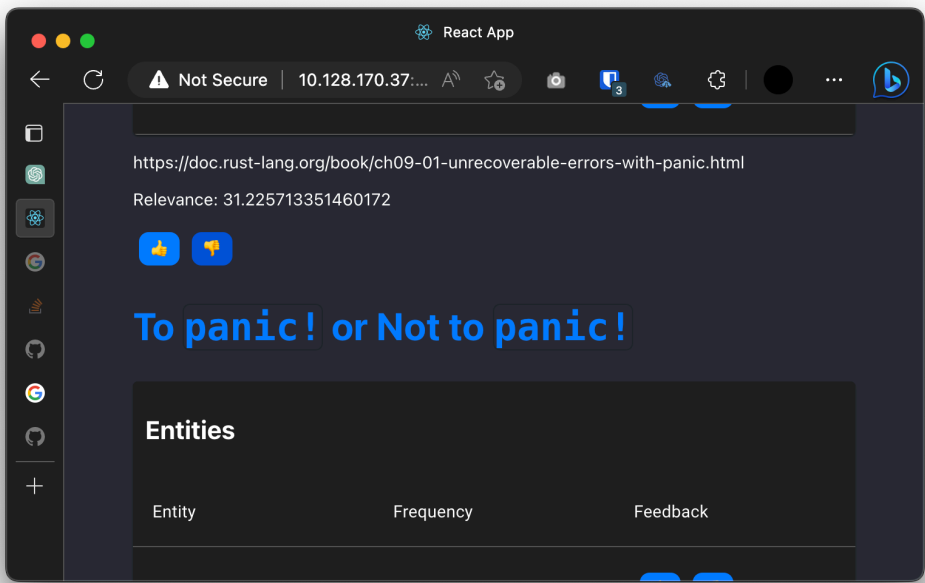
此外，还设计了一个展示评价结果的图表。在这个图表中，将准确率和召回率以及其他评价指标进行了可视化展示，这样管理员可以更直观地了解到检索系统的性能状态。

以上就是检索结果评价的设计与实现的主要内容。通过这个评价方法，不仅可以了解到检索系统的性能，还可以根据评价结果对系统进行持续优化，从而提供更优质的服务。

8 环境与社会可持续发展的考虑

在项目的设计与实施过程中，深入考虑了环境保护和社会可持续发展的原则。以下是在这两个方面的主要考虑和实践：

Figure 8: 反馈按钮



8.1 项目实施过程中的环保措施

- 1. 使用高效的算法和数据结构：在后端，采用高效的算法和数据结构，如倒排索引和向量空间模型，以及 Python Flask 微服务，这些都有助于提高数据处理和信息检索的速度，降低计算资源的消耗。
- 2. 采用 Lazy Loading 的方式加载文档：在前端，使用 Lazy Loading 的方式加载文档，每次只返回一页文档，并且仅当展开文档的时候才会从后端加载文档内容。这不仅提高了用户界面的响应速度，而且有效地节省了计算和网络资源。

8.2 社会可持续发展影响考虑

信息检索系统为用户提供了方便快捷的信息查找服务，使用户可以在海量信息中找到自己需要的内容，节省了用户的时间和精力，提高了工作和学习效率。此外，在搜索时不会使用用户的个人数据，确保用户的个人信息不会被泄露。这些都符合社会可持续发展的原则。

通过上述措施，项目在实现其功能的同时，也积极地贡献于环境保护和社会可持续发展。

9 优化与创新性思考

在项目的设计与实现过程中，进行了一些优化措施，并结合创新性思考，以提高系统的效率和用户体验。

9.1 优化措施和效果

- 优化的排序算法: 在基于 TF-IDF 在向量空间中计算相似度的基础上, 根据查询词在文中出现的位置和次数等调整排名, 使得查询结果排名更合理;
- 多媒体搜索: 支持图片搜索. 根据用户上传的图片提取关键字进行搜索;
- 基于用户反馈的动态排名优化: 当用户对某个文档或关键词提供反馈时, 会动态地更新它们的排名。对于用户给予正面反馈的项目, 提升它们的排名, 而对于负面反馈的项目, 则降低它们的排名。这样的动态排名优化不仅提高了用户的搜索体验, 也使系统能更好地适应用户的需求。

9.2 创新性思考的实现

- 多媒体信息检索: 在信息检索系统中, 不仅实现了文本信息的检索, 还尝试了图片信息的检索。用户可以通过上传图片, 获取与图片内容相关的关键词。这一创新性功能大大拓宽了信息检索的范围, 使用户可以更方便地获取和使用信息。
- 人工智能的应用: 在后端, 引入了人工智能技术来提升信息检索的效率和准确性。例如, 使用了 ResNet50 模型来处理用户上传的图片, 识别出图片中的物体, 并返回相关的关键词。这一创新性应用使信息检索系统具有更高的智能化水平, 提供了更好的用户体验。

10 结论

10.1 项目成果总结

本项目成功实现了一个多媒体信息检索系统, 包括了从数据获取、存储, 到前后端设计, 再到多媒体 (文本和图片) 的信息检索以及用户反馈处理等一系列功能。系统具备良好的用户界面, 易于使用, 支持自然语言查询, 并能有效地返回相关的文档信息, 包括相关度、文档标题、主要匹配内容、URL 以及文档日期等。同时, 为了提高用户体验, 还设计了基于用户反馈的动态排名功能。

在技术层面, 本项目采用了现代前沿技术, 如 React TypeScript 进行前端设计, Go 语言的 Gin 框架进行后端设计, Python Flask 构建图片检索微服务, 以及 Scrapy 进行网络爬虫数据获取。此外, 在项目中也应用了人工智能技术, 如使用 ResNet50 模型进行图片识别, 进一步提升了系统的效率和准确性。

10.2 展望和改进

虽然当前的信息检索系统已经具备基本的功能并在一定程度上满足了用户的需求, 但仍存在一些可以改进和优化的地方。

首先, 希望能进一步优化用户界面, 提供更为丰富和直观的可视化结果, 以帮助用户更好地理解 and 评估检索结果。比如, 可以展示文档的摘要或是关键词云等。

其次，希望能拓展更多类型的多媒体检索功能，如视频或音频检索，以提供更全面的信息服务。同时，也希望引入更多的人工智能技术，如自然语言处理或深度学习等，来进一步提高检索的准确性和效率。

最后，希望在未来能引入更多的用户反馈机制，如用户评价或建议等，以便更好地理解用户需求，优化和改进系统。

总的来说，我将继续努力，旨在提供一个更高效、准确、智能和用户友好的信息检索系统。