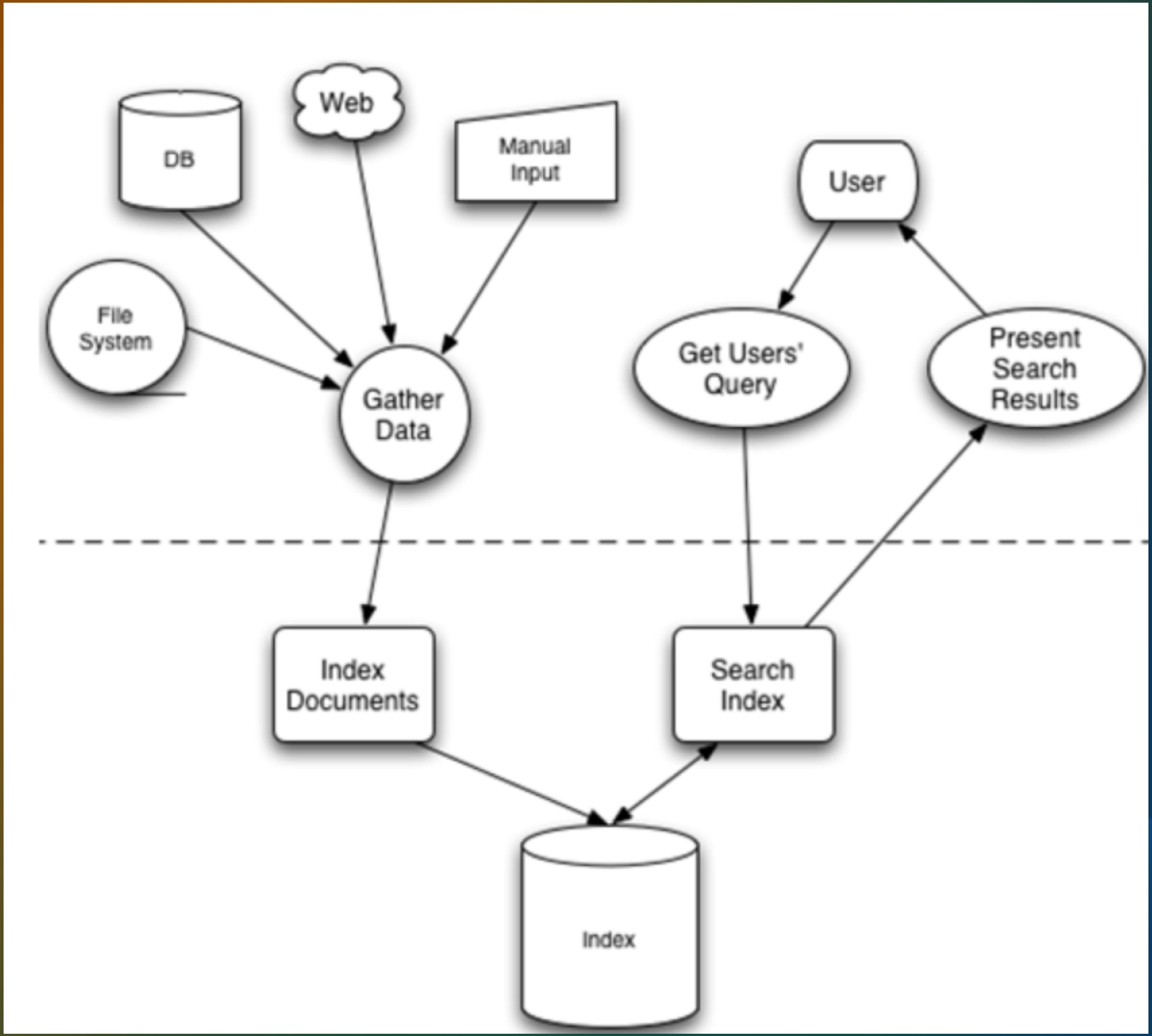
SimpleSearch

袁剑

目录

原理概述


Web
DB
Manual Input
User
File System
Gather Data
Get Users' Query
Present Search Results
Index Documents
Search Index
Index

- 索引创建（Index）

从现实世界中所有的结构化和非结构化数据提取信息，创建索引的过程。

- 搜索索引(Search)

从用户的查询请求,搜索创建的索引,然后返回结果的过程。

- 索引里面究竟存些什么?(Index)
- 如何创建索引?(Indexing)
- 如何对索引进行搜索?(Search)

索引里面究竟存些什么?(Index)

- 正排索引
  - 文档1:面包超人
  - 文档2:超人：解放
  - 文档3:超人 钢铁之躯
- 倒排索引
  - 面包:文档1
  - 超人:文档1，文档2，文档3
  - 解放:文档2
  - 钢铁:文档3
  - 之躯:文档3

如何创建索引?(Indexing)

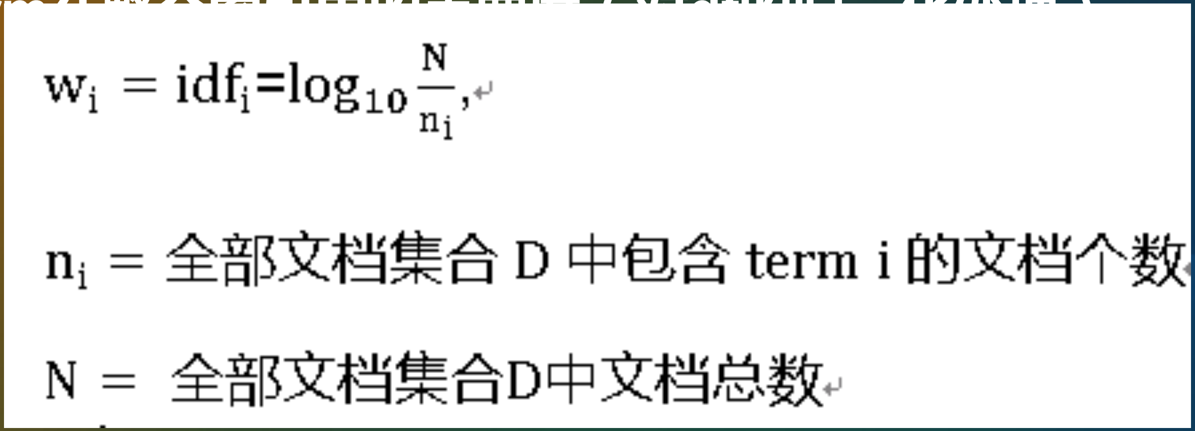
- 分词（Tokenize）
  - 将文档分成一个一个单独的单词。
  - 去除标点符号。
  - 去除停词(Stop word)。
- 语言处理
  - 变为小写(Lowercase)。
  - 将单词缩减为词根形式,如“cars”到“car”
  - 将单词转变为词根形式,如“drove”到“drive”等
- 索引
  - 利用得到的词(Term)创建一个字典
  - 合并相同的词(Term)成为文档倒排集合

如何对索引进行搜索?(Search)

- 搜索
  - 解放超人
- 分词
  - 解放 超人
- 查找反向索引：
  - 解放：文档2
  - 超人：文档1 文档2 文档3
- 根据相关度排序

如何对索引进行搜索?(Search)

- 文档频次df（Document Frequency）
  - 总共有多少文档包含此词(Term)，Df越大说明越不重要
- 词频tf（Term Frequency）
  - 此文档中包了几个此词(Term)，tf 越大说明越重要
- 反向文档频次Idf（inverse document frequency）
  - Term在整个索引中的重要度（对df的归一化处理）



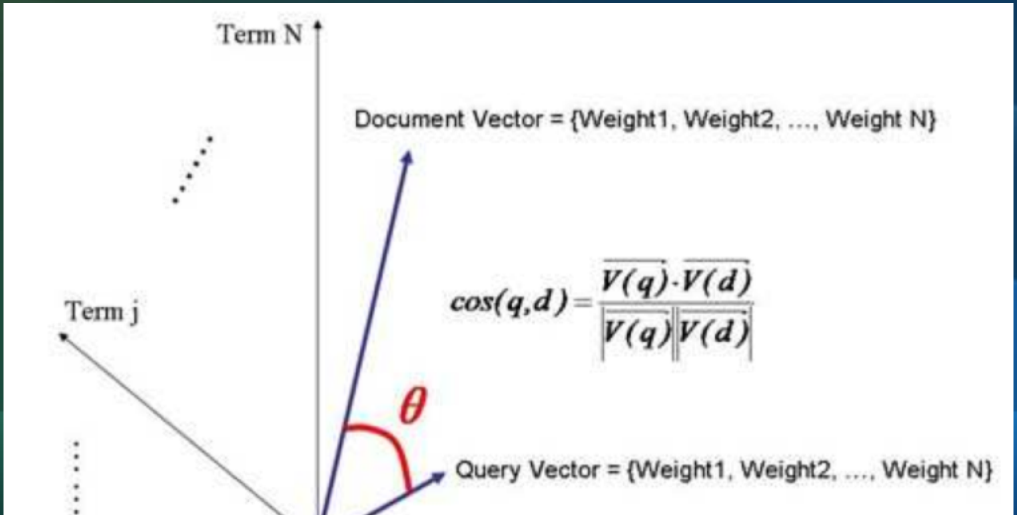
$$w_i = idf_i = \log_{10}\frac{N}{n_i},$$

n_i = 全部文档集合 D 中包含 term i 的文档个数

N = 全部文档集合D中文档总数

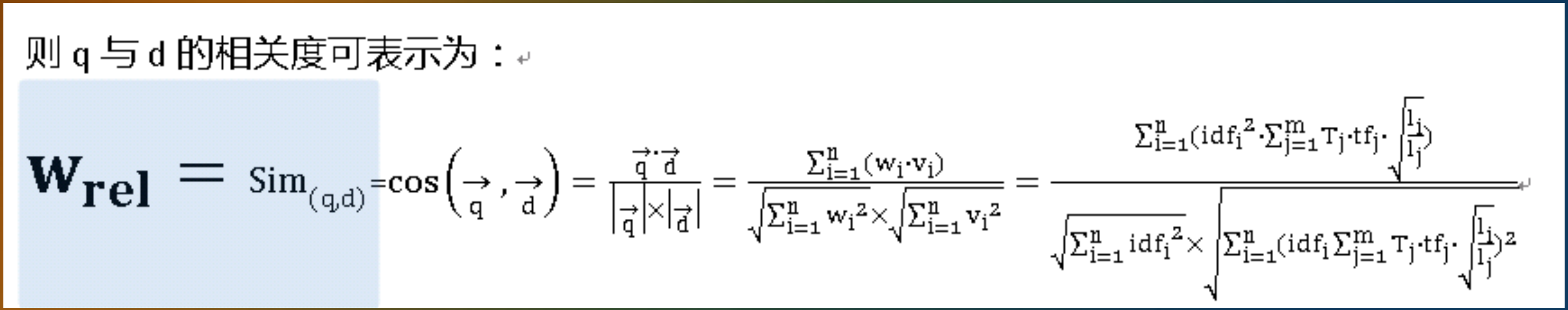
计算相关度

- 文档中词(term)的权重(term weight) 看作一个向量
  - Document = {term1, term2, ... ,term N}
  - Document Vector = {weight1, weight2, ... ,weight N}
- 查询语句的权重也用向量来表示
  - Query = {term1, term 2, ... , term N}
  - Query Vector = {weight1, weight2, ... , weight N}
- 把所有搜索出的文档向量及查询向量放到一个 N 维空间中,每个词(term)是一维，两个向量之间的夹角越小,相关性越大



Term N
Document Vector = {Weight1, Weight2, ..., Weight N}
Term j
$\cos(q,d)=\frac{\overline{V(q)}\cdot\overline{V(d)}}{|\overline{V(q)}||\overline{V(d)}|}$
θ
Query Vector = {Weight1, Weight2, ..., Weight N}

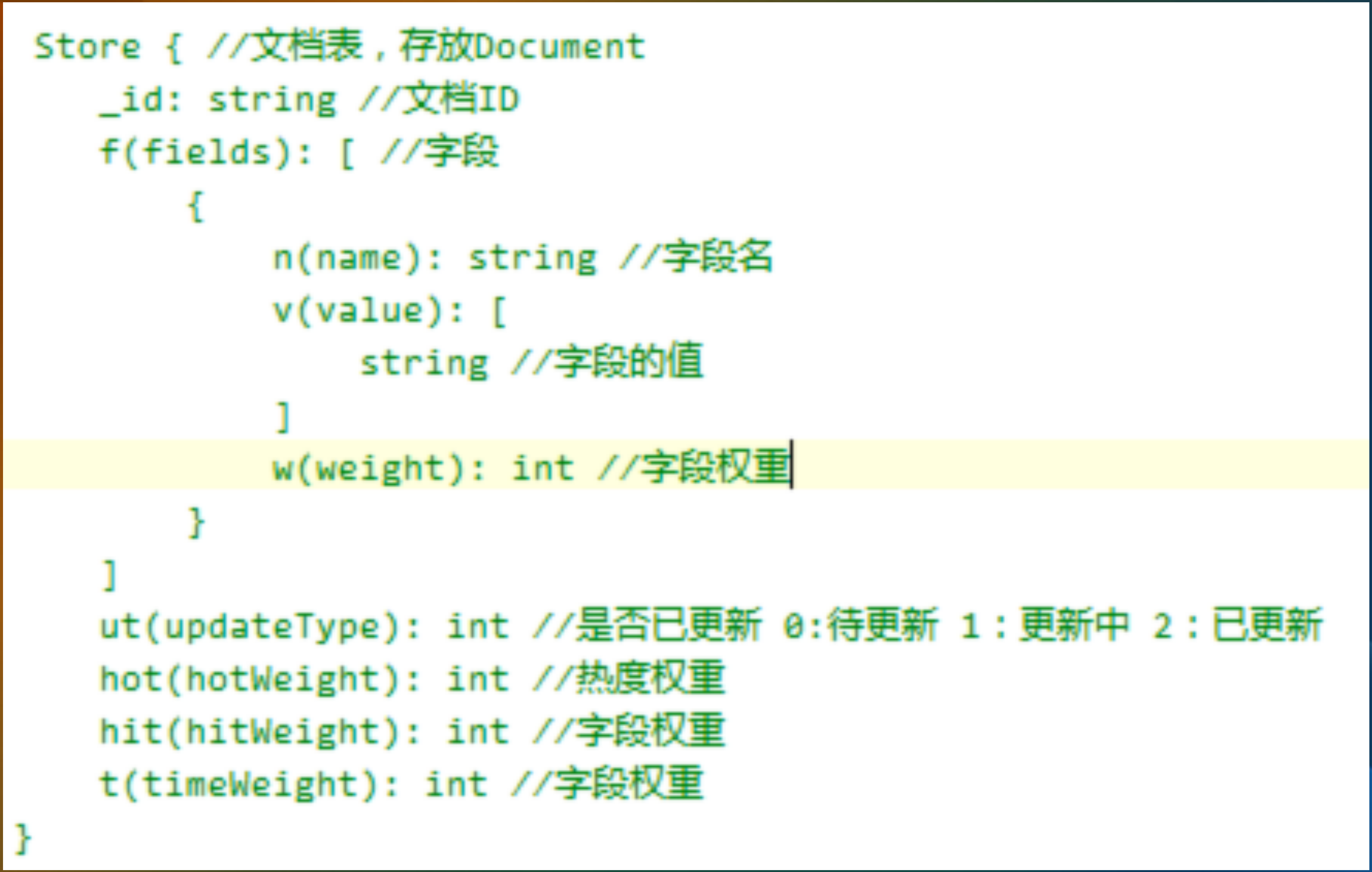
例子，需求文档：4.2.1.2



则 q 与 d 的相关度可表示为：

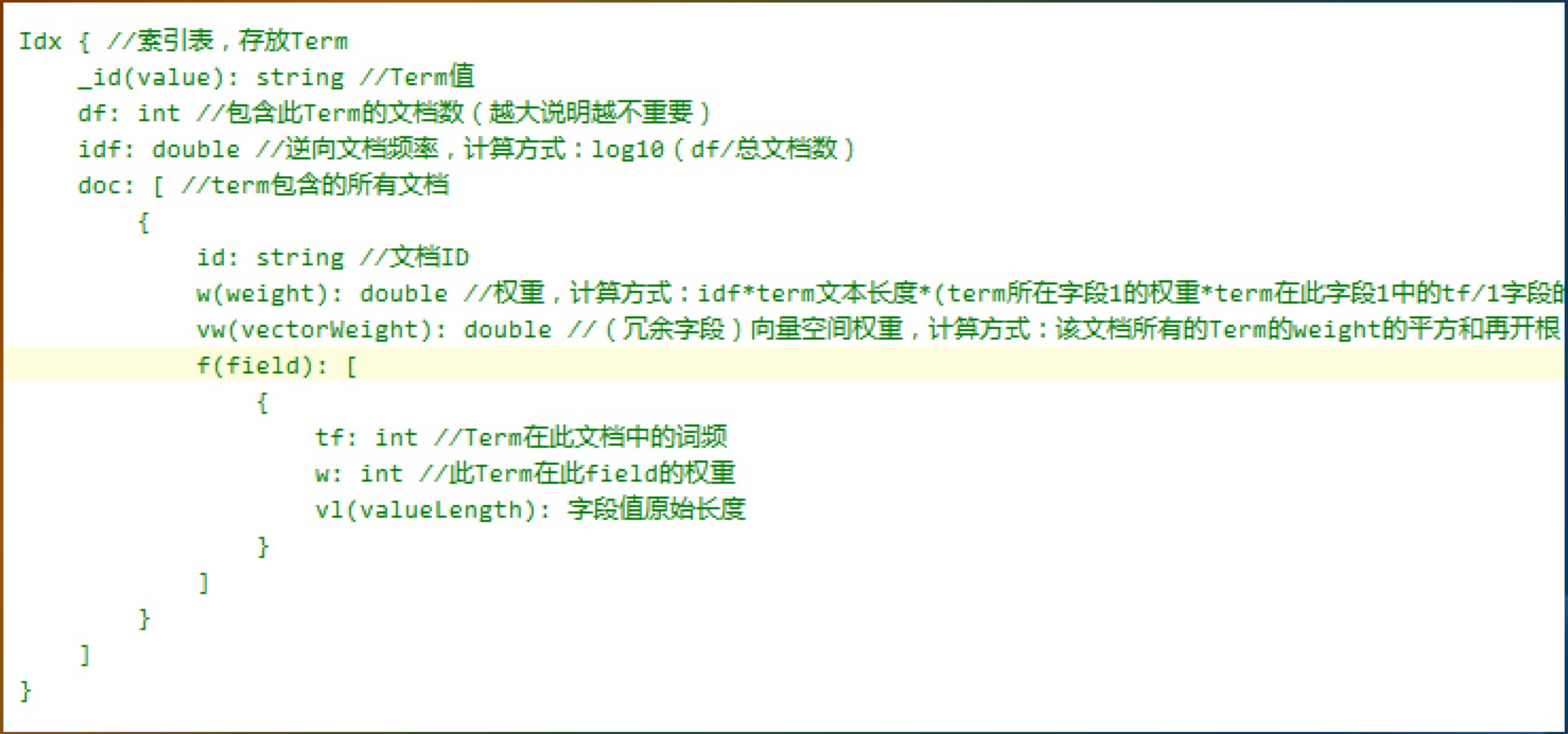
$$W_{rel} = Sim_{(q,d)} = \cos\left(\vec{q}, \vec{d}\right) = \frac{\vec{q}\cdot\vec{d}}{\left|\vec{q}\right|\times\left|\vec{d}\right|} = \frac{\sum_{i=1}^{n}(w_i\cdot v_i)}{\sqrt{\sum_{i=1}^{n} w_i^2}\times\sqrt{\sum_{i=1}^{n} v_i^2}} = \frac{\sum_{i=1}^{n}(idf_i^2\cdot\sum_{j=1}^{m} T_j\cdot tf_j\cdot\sqrt{\frac{l_i}{l_j}})}{\sqrt{\sum_{i=1}^{n} idf_i^2}\times\sqrt{\sum_{i=1}^{n}(idf_i\sum_{j=1}^{m} T_j\cdot tf_j\cdot\sqrt{\frac{l_i}{l_j}})^2}}$$

索引-正向索引



```
Store { //文档表，存放Document
    _id: string //文档ID
    f(fields): [ //字段
        {
            n(name): string //字段名
            v(value): [
                string //字段的值
            ]
            w(weight): int //字段权重
        }
    ]
    ut(updateType): int //是否已更新 0:待更新 1：更新中 2：已更新
    hot(hotWeight): int //热度权重
    hit(hitWeight): int //字段权重
    t(timeWeight): int //字段权重
}
```

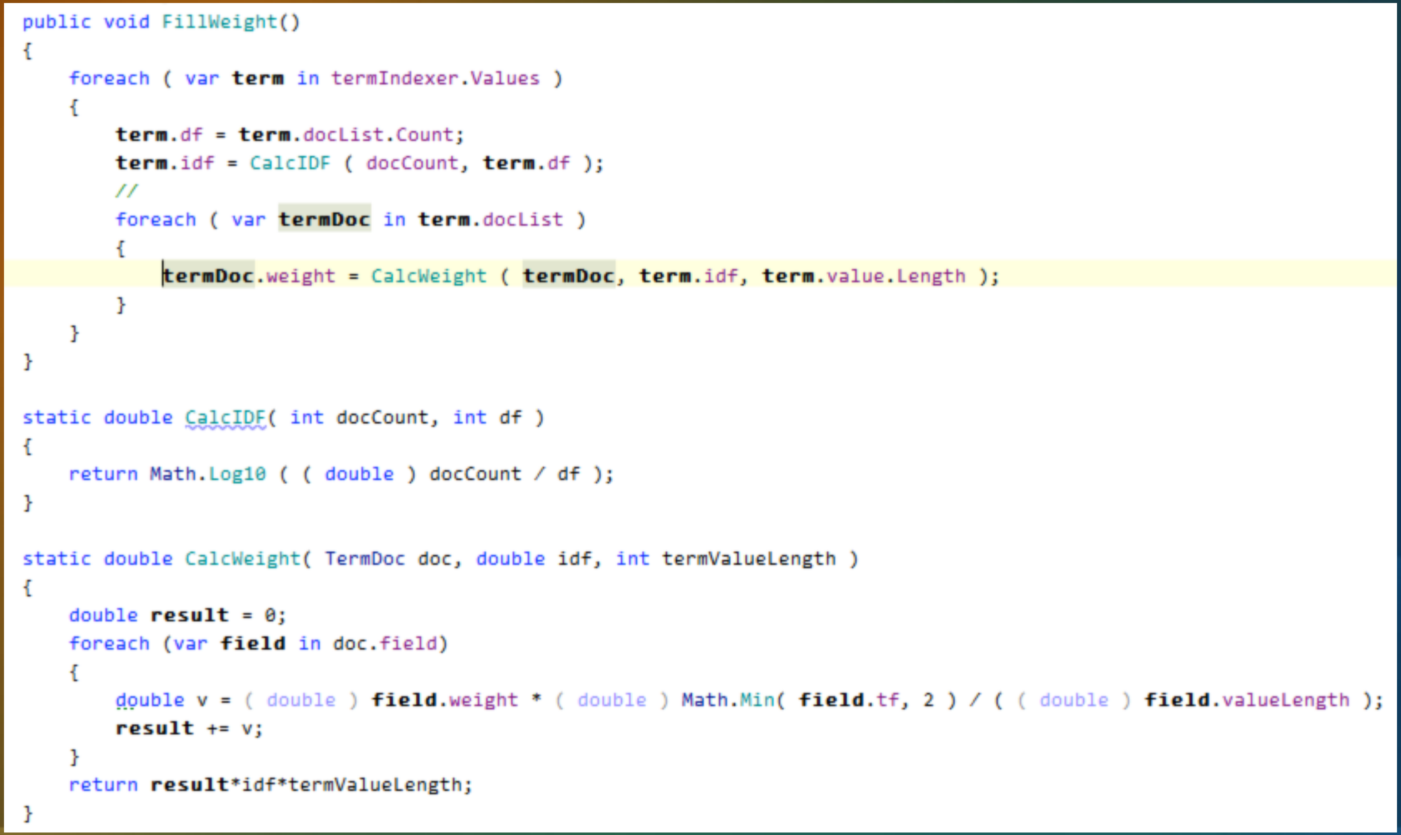
索引-倒排索引



```
Idx { //索引表，存放Term
    _id(value): string //Term值
    df: int //包含此Term的文档数（越大说明越不重要）
    idf: double //逆向文档频率，计算方式：log10（df/总文档数）
    doc: [ //term包含的所有文档
        {
            id: string //文档ID
            w(weight): double //权重，计算方式：idf*term文本长度*(term所在字段1的权重*term在此字段1中的tf/1字段
            vw(vectorWeight): double //（冗余字段）向量空间权重，计算方式：该文档所有的Term的weight的平方和再开根
            f(field): [
                {
                    tf: int //Term在此文档中的词频
                    w: int //此Term在此field的权重
                    vl(valueLength): 字段值原始长度
                }
            ]
        }
    ]
}
```

倒排索引创建过程

- 遍历全部文档
- 对各个文档中的每个索引字段进行分词成Term字典
- 遍历全部Term，计算idf及文档Weight

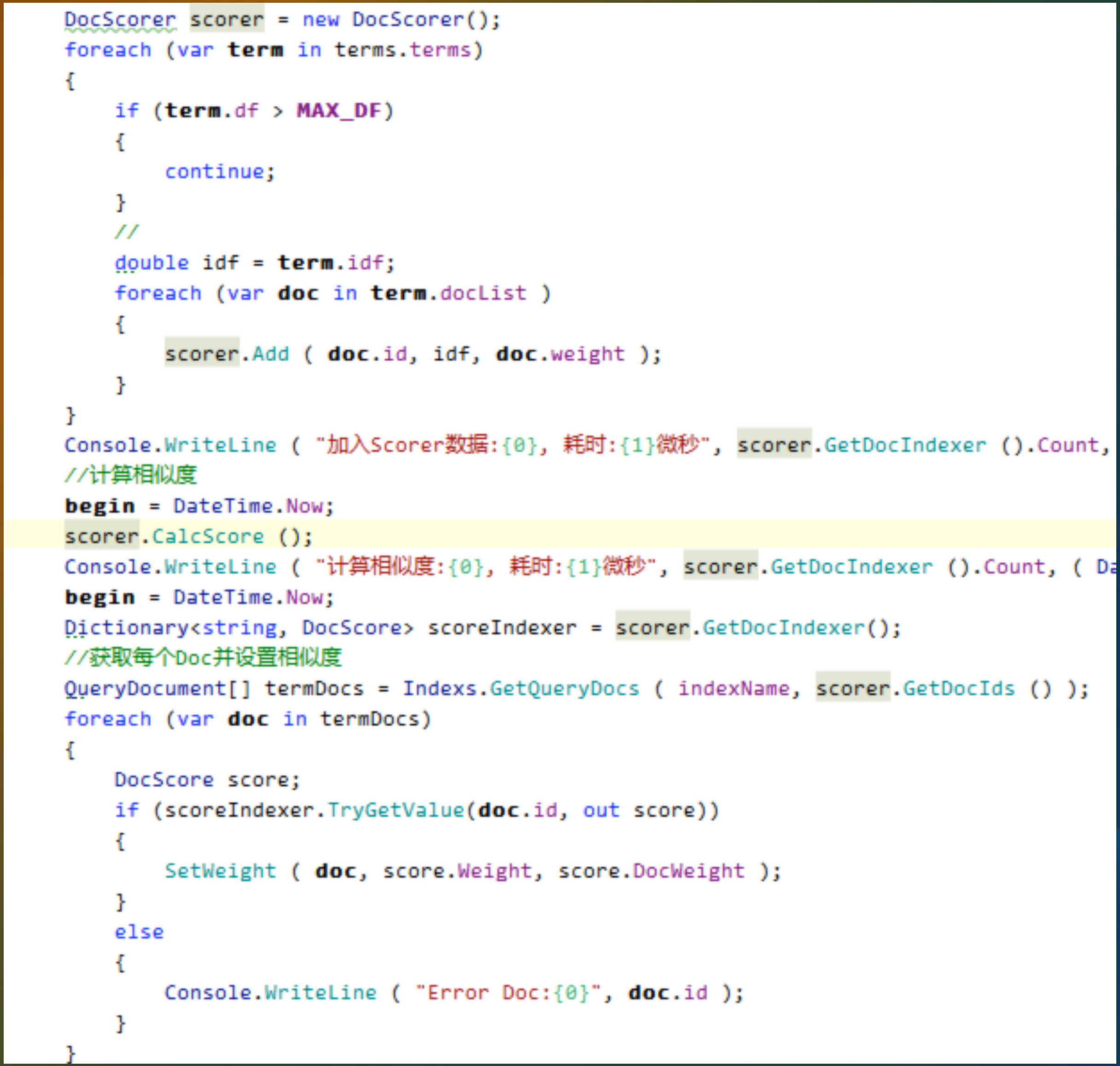


```
public void FillWeight()
{
    foreach ( var term in termIndexer.Values )
    {
        term.df = term.docList.Count;
        term.idf = CalcIDF ( docCount, term.df );
        //
        foreach ( var termDoc in term.docList )
        {
            termDoc.weight = CalcWeight ( termDoc, term.idf, term.value.Length );
        }
    }
}

static double CalcIDF( int docCount, int df )
{
    return Math.Log10 ( ( double ) docCount / df );
}

static double CalcWeight( TermDoc doc, double idf, int termValueLength )
{
    double result = 0;
    foreach (var field in doc.field)
    {
        double v = ( double ) field.weight * ( double ) Math.Min( field.tf, 2 ) / ( ( double ) field.valueLength );
        result += v;
    }
    return result*idf*termValueLength;
}
```

Term查询

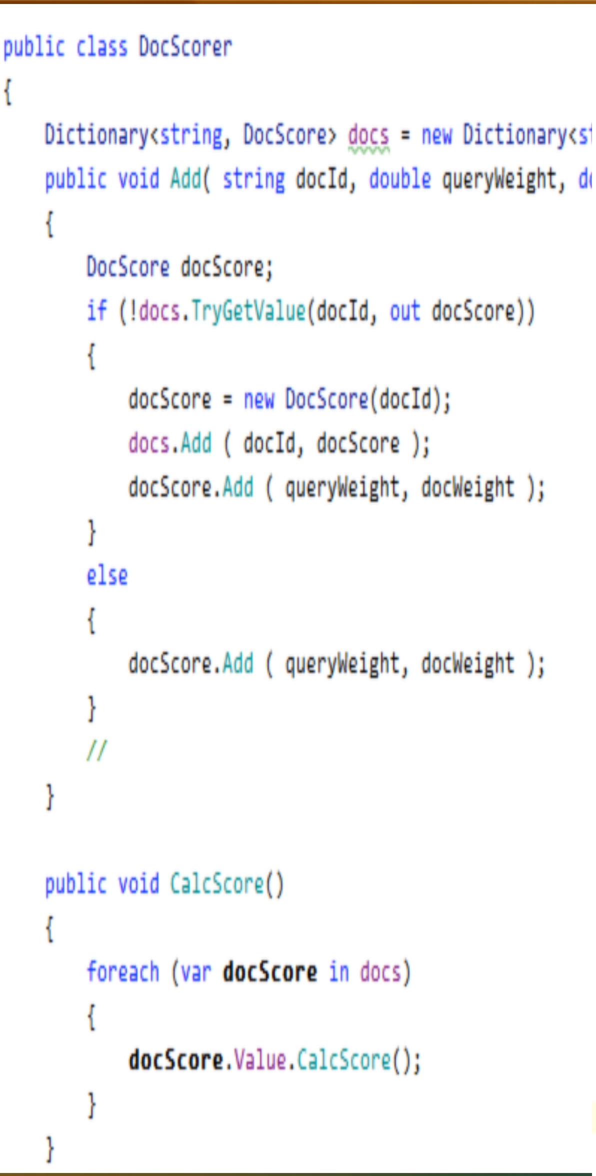
- 遍历Query中的所有Term
- 从倒排索引中获取所有相关的Term并计算其中每个文档的相关度（打分）
- 从正向索引中获取相关文档并设置其相关度
- 把所有相关文档按（相关度、热度、点击、时间权重）等进行排序并返回

Term查询

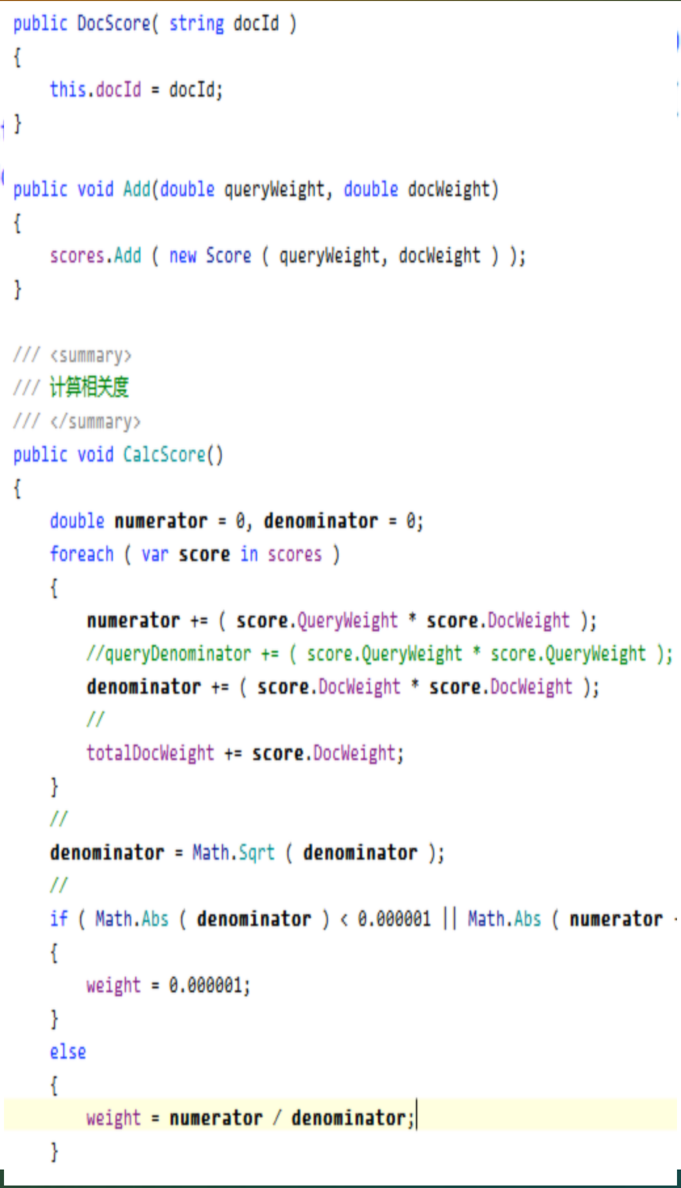


```
DocScorer scorer = new DocScorer();
foreach (var term in terms.terms)
{
    if (term.df > MAX_DF)
    {
        continue;
    }
    //
    double idf = term.idf;
    foreach (var doc in term.docList )
    {
        scorer.Add ( doc.id, idf, doc.weight );
    }
}
Console.WriteLine ( "加入Scorer数据:{0}, 耗时:{1}微秒", scorer.GetDocIndexer ().Count,
//计算相似度
begin = DateTime.Now;
scorer.CalcScore ();
Console.WriteLine ( "计算相似度:{0}, 耗时:{1}微秒", scorer.GetDocIndexer ().Count, ( Da
begin = DateTime.Now;
Dictionary<string, DocScore> scoreIndexer = scorer.GetDocIndexer();
//获取每个Doc并设置相似度
QueryDocument[] termDocs = Indexs.GetQueryDocs ( indexName, scorer.GetDocIds () );
foreach (var doc in termDocs)
{
    DocScore score;
    if (scoreIndexer.TryGetValue(doc.id, out score))
    {
        SetWeight ( doc, score.Weight, score.DocWeight );
    }
    else
    {
        Console.WriteLine ( "Error Doc:{0}", doc.id );
    }
}
```

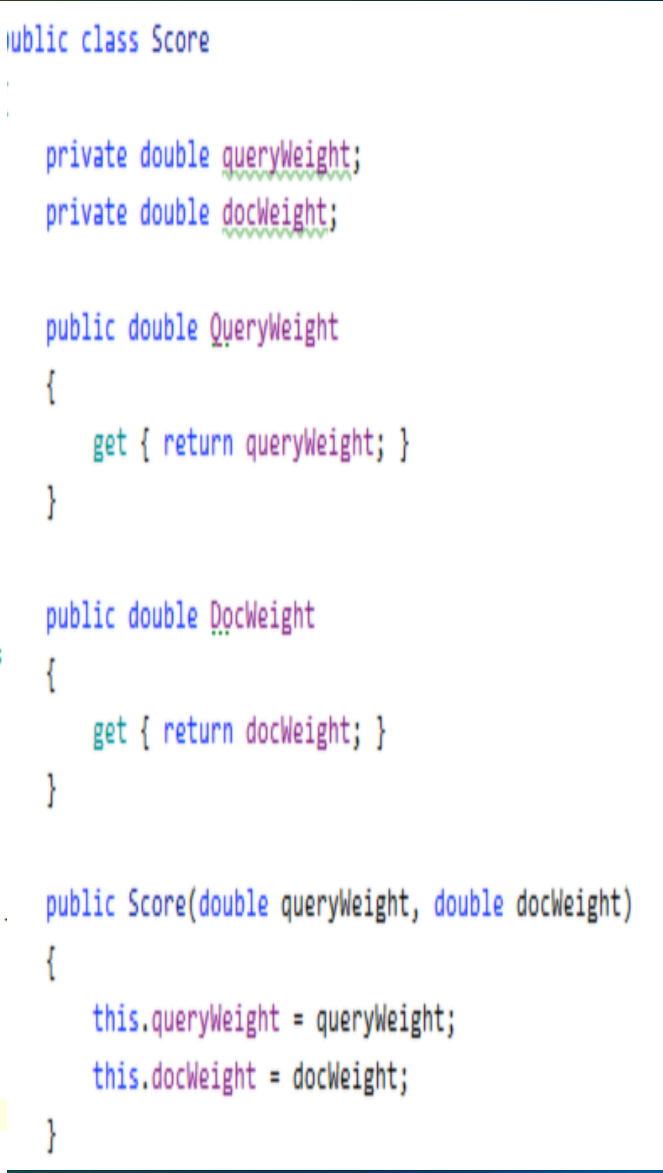
Term查询



```
public class DocScorer
{
    Dictionary<string, DocScore> docs = new Dictionary<s
    public void Add( string docId, double queryWeight, d
    {
        DocScore docScore;
        if (!docs.TryGetValue(docId, out docScore))
        {
            docScore = new DocScore(docId);
            docs.Add ( docId, docScore );
            docScore.Add ( queryWeight, docWeight );
        }
        else
        {
            docScore.Add ( queryWeight, docWeight );
        }
        //
    }

    public void CalcScore()
    {
        foreach (var docScore in docs)
        {
            docScore.Value.CalcScore();
        }
    }
```



```
public DocScore( string docId )
{
    this.docId = docId;
}

public void Add(double queryWeight, double docWeight)
{
    scores.Add ( new Score ( queryWeight, docWeight ) );
}

/// <summary>
/// 计算相关度
/// </summary>
public void CalcScore()
{
    double numerator = 0, denominator = 0;
    foreach ( var score in scores )
    {
        numerator += ( score.QueryWeight * score.DocWeight );
        //queryDenominator += ( score.QueryWeight * score.QueryWeight );
        denominator += ( score.DocWeight * score.DocWeight );
        //
        totalDocWeight += score.DocWeight;
    }
    //
    denominator = Math.Sqrt ( denominator );
    //
    if ( Math.Abs ( denominator ) < 0.000001 || Math.Abs ( numerator
    {
        weight = 0.000001;
    }
    else
    {
        weight = numerator / denominator;
    }
```



```
ublic class Score

    private double queryWeight;
    private double docWeight;

    public double QueryWeight
    {
        get { return queryWeight; }
    }

    public double DocWeight
    {
        get { return docWeight; }
    }

    public Score(double queryWeight, double docWeight)
    {
        this.queryWeight = queryWeight;
        this.docWeight = docWeight;
    }
```

DEMO

- git@192.168.50.30:web/search