文 档

Coreseek 全文检索服务器 2.0 (Sphinx 0.9.8)参考手册

文档版本：v0.9

目录

1. 简介

1.1. 什么是 Sphinx

Sphinx 是一个在 GPLv2 下发布的一个全文检索引擎，商业授权（例如, 嵌入到其他程序中）需要联系我们（Sphinxsearch.com）以获得商业授权。

一般而言，Sphinx 是一个独立的搜索引擎，意图为其他应用提供高速、低空间占用、高结果相关度的全文搜索功能。Sphinx 可以非常容易的与 SQL 数据库和脚本语言集成。

当前系统内置 MySQL 和 PostgreSQL 数据库数据源的支持，也支持从标准输入读取特定格式的 XML 数据。通过修改源代码，用户可以自行增加新的数据源（例如：其他类型的 DBMS 的原生支持）。

搜索 API 支持 PHP、Python、Perl、Rudy 和 Java，并且也可以用作 MySQL 存储引擎。搜索 API 非常简单，可以在若干个小时之内移植到新的语言上。

Sphinx 是 SQL Phrase Index 的缩写，但不幸的和 CMU 的 Sphinx 项目重名。

Coreseek 全文检索服务器 2.0 是在 Sphinx 基础上开发的全文检索软件，按照 GPLv2 协议发行。Coreseek (http://www.coreseek.com) 为 sphinx 在中国地区的用户提供支持服务，如果您不希望纠缠与琐碎的技术细节，请直接联系我。

本文可能存在潜在的翻译错误，如果您发现本文的翻译错误，请联系我：

我的联系方式：

coreseek@gmail.com 李沫南

1.2. Sphinx 的特性

- 高速的建立索引(在当代 CPU 上，峰值性能可达到 10 MB/秒);
- 高性能的搜索(在 2 – 4GB 的文本数据上，平均每次检索响应时间小于 0.1 秒);
- 可处理海量数据(目前已知可以处理超过 100 GB 的文本数据, 在单一 CPU 的系统上可处理 100 M 文档);
- 提供了优秀的相关度算法，基于短语相似度和统计（BM25）的复合 Ranking 方法;

- 支持分布式搜索;
- provides document exceprts generation;
- 可作为 MySQL 的存储引擎提供搜索服务;
- 支持布尔、短语、词语相似度等多种检索模式;
- 文档支持多个全文检索字段(最大不超过 32 个);
- 文档支持多个额外的属性信息(例如：分组信息，时间戳等);
- 停止词查询;
- 支持单一字节编码和 UTF-8 编码;
- 原生的 MySQL 支持(同时支持 MyISAM 和 InnoDB);
- 原生的 PostgreSQL 支持.

1.3. 如何获得 Sphinx

Sphinx 可以从官方网站 http://www.sphinxsearch.com/ 下载，支持中文分词的 Sphinx 可以从 http://www.coreseek.com/ 下载。

目前，Sphinx 的发布包包括如下软件：

- `indexer`: 用于创建全文索引;
- `search`: 一个简单的命令行(CLI) 的测试程序，用于测试全文索引;
- `searchd`: 一个守护进程，其他软件可以通过这个守护进程进行全文检索;
- `sphinxapi`: 一系列 searchd 的客户端 API 库，用于流行的 Web 脚本开发语言(PHP, Python, Perl, Ruby)。

1.4. 许可 协议

1.5. 作者和贡献者

作者

Sphinx 的最初作者和目前的主要开发人员:

- Andrew Aksyonoff, <shodan(at)shodan.ru>

贡献者

为 Sphinx 的开发出过力的人员和他们的贡献如下（以下排名不分先后）:

- Robert "coredev" Bengtsson (Sweden), initial version of PostgreSQL data source;
- Len Kranendonk, Perl API
- Dmytro Shteflyuk, Ruby API

此外，还有许多人提出了宝贵的想法、错误报告以及修正。在此一并致谢。

1.6. 历史

Sphinx 的开发工作可以上溯到 2001 年，当时作者试图为一个数据库驱动的网站寻找一个可接受的搜索的解决方案，但是当时没有任何方案能够满足要求。事实上，主要是如下问题：

- 搜索质量(例如：有效的相关度算法)
  - 单纯的统计学方法的效果非常糟糕，特别是在大量的短篇文档的集合上，例如：论坛、博客等等。
- 搜索速度
  - 特别是当搜索的短语包括“停止词”时，表现的尤其明显，例如："to be or not to be"
- 建立索引时，可控的磁盘和 CPU 消耗
  - 在虚拟主机的环境下，这一点的重要性要超过对索引构造速度的要求

年复一年，其他的解决方案有了很多改进，但是，我个人认为仍然没有一种解决方案足够的好，能让我将搜索平台迁移过去。

去年，Sphinx 的用户给了我很多正面的反馈，因此，显而易见的，Sphinx 的开发过程将会继续（也许将持续到世界末日）。

2. 安装

2.1. 支持的操作系统

在绝大多数现代的 Unix 类操作系统上，只需要一个 C++编译器就可以编译并运行 Sphinx，而不需要对源码进行任何改动。

目前，Sphinx 可以在以下系统上运行:

- Linux 2.4.x, 2.6.x (包括各种发行版)
- Windows 2000, XP
- FreeBSD 4.x, 5.x, 6.x
- NetBSD 1.6, 3.0
- Solaris 9, 11
- Mac OS X

支持的 CPU 种类包括 X86, X86-64, SPARC64。

我希望 Sphinx 也能够在其他的 Unix 平台上工作，如果你运行 Sphinx 使用的操作系统不在上面的名单中，请告诉我。

目前的阶段，Sphinx 的 Windows 版可用于测试和调试，但不建议用于生产系统。最突出的两个问题是：1）缺少并发查询的支持；2）缺少索引数据热切换的支持。虽然目前已经有成功的生产环境克服了这两个问题，仍然不推荐在 Windows 下运行 Sphinx 提供高强度的搜索服务。

2.2. 依赖的工具

在 UNIX 平台上，你需要以下的工具用来编译和安装 Sphinx:

- C++编译器。GNU gcc 就能够干这个活。
- make 程序。GNU make 就能够干这个活。

在 Windows 平台上，你需要 Microsoft Visual C/C++ Studio .NET 2003 or 2005。其他的编译器/开发环境也许也能搞定这件事，但你可能需要自己手工制作他们所需的 Makefile 或者工程文件。

2.3. 安装 Sphinx

1. 将你下载的 tar 包解压，并进入 `sphinx` 子目录:

   ```
   $ tar xzvf sphinx-0.9.7.tar.gz
   $ cd sphinx
   ```

2. 运行 configuration 程序：

   ```
   $ ./configure
   ```

   configure 程序有很多运行选项。完整的列表可以通过使用 `--help` 开关得到。最重要的如下：

   - `--prefix`, 定义将 Sphinx 安装到何处;
   - `--with-mysql`, 当自动检测失败时，指出在那里能找到 MySQL 头文件和库文件;
   - `--with-pgsql`, 指出在那里能找到 PostgreSQL 头文件和库文件。

3. 制作二进制程序：

   ```
   $ make
   ```

4. 按照二进制程序到你选好的目录下：

   ```
   $ make install
   ```

2.4. 已知的问题和解决方法

如果 `configure` 程序没有找到 MySQL 的头文件和库文件，请试图检查 `mysql-devel` 这个包是否安装了。在有些系统上，默认安装包括这个包。

如果 make 程序给出如下错误提示，

```
/bin/sh: g++: command not found
make[1]: *** [libsphinx_a-sphinx.o] Error 127
```

请检查 `gcc-c++` 这个包是否安装了。

如果你在编译时得到如下错误

```
sphinx.cpp:67: error: invalid application of `sizeof' to
   incomplete type `Private::SizeError<false>'
```

这意味着某些编译时的类型检查失败了，一个最有可能的原因是在你的系统上类型 off_t 的长度小于 64bit。一个快速的修复手段是，你可以修改 sphinx.h ，将在定义类型 SphOffset_t 处，将 off_t 替换成 DWORD，需要注意，这种改动将使你的全文索引文件不能超过 2GB。

即便这种修改有用，也请汇报这一问题，在汇报中请包括具体的错误信息以及操作系统编译器的配置情况。这样，我可能能够在下一个版本中解决这一问题。

如何你遇到了其他的任何问题，或者前面的建议对你没有帮助，别犹豫，请立刻联系我。

2.5. Sphinx 快速入门教程

以下所有的例子都假设你将 sphinx 安装在目录 `/usr/local/sphinx` 中了。

为了使用 Sphinx，你需要：

1. 创建配置文件

   缺省的配置文件名为 `sphinx.conf`。全部的 `Sphinx` 提供的程序默认都在当前工作的目录下寻找该文件。

   由 `configure` 程序生成的示例配置文件 `sphinx.conf.dist` 中包括全部选项的注释，复制并编辑这个文件使之适用于你的具体情况：

   ```
   $ cd /usr/local/sphinx/etc
   $ cp sphinx.conf.dist sphinx.conf
   $ vi sphinx.conf
   ```

   在示例配置文件中，将试图对 MySQL 数据库 `test` 中的 `documents` 表建立索引；因此在这里还提供了 `example.sql` 用于给测试表增加数据用于测试：

   ```
   $ mysql -u test < /usr/local/sphinx/etc/example.sql
   ```

2. 运行 indexer 创建你的全文索引：

   ```
   $ cd /usr/local/sphinx/etc
   $ /usr/local/sphinx/bin/indexer
   ```

3. 检索你新创建的索引！

你可以使用 search 实用程序可以从命令行对索引进行检索：

```
$ cd /usr/local/sphinx/etc
$ /usr/local/sphinx/bin/search test
```

如果要从 PHP 脚本检索索引，你需要：

1. 运行守护进程 searchd，PHP 脚本需要连接到 searchd 上进行检索 ：

   ```
   $ cd /usr/local/sphinx/etc
   $ /usr/local/sphinx/bin/searchd
   ```

2. 运行 PHP API 附带的 test 脚本（运行之前请确认 searchd 守护进程已启动）：

   ```
   $ cd sphinx/api
   $ php test.php test
   ```

3. 将 API 文件(位于 `api/sphinxapi.php`) 包含进你自己的脚本，开始编程。

搜索愉快！

3. 建立索引

3.1. 数据源

索引的数据可以来自各种各样不同的来源：SQL 数据库、纯文本、HTML 文件、邮件等等。从 Sphinx 的视角看，索引数据是一个结构化的文档的集合，其中每个文档是字段的集合，这和 SQL 数据库的视角有所不同，在那里，每一行代表一个文档，每一列代表一个字段。

由于数据来源的不同，需要不同的代码来获取数据、处理数据以供 Sphinx 进行索引的建立。这种代码被称之为数据源驱动程序（简称：驱动或数据源）。

在本文撰写时，Sphinx 中包括 MySQL 和 PostgreSQL 数据源的驱动程序，这些驱动使用数据库系统提供的 C/C++原生接口连接到数据库服务器并获取数据。此外，Sphinx 还提供了额外的被成为 xmlpipe 的数据源驱动，该驱动运行某个具体的命令，并从该命令的输出中读入数据。数据的格式在3.8, “xmlpipe 数据源” 中有介绍。

如果确有必要，一个索引的数据可以来自多个数据源。这些数据将严格按照配置文件中定义的顺序进行处理。所有从这些数据源获取到的文档将被合并，共同产生一个索引，如同他们来源于同一个数据源一样。

3.2. 属性

属性是附加在每个文档上的额外的信息（值），可以在搜索的时候用于过滤和排序。

搜索结果通常不仅仅是进行文档的匹配和相关度的排序，经常还需要根据其他与文档相关联的值，对结果进行额外的处理。例如，用户可能需要对新闻检索结果依次按日期和相关度排序，检索特定价格范围内的产品，检索某些特定用户的 blog 日志，或者将检索结果按月分组。为了高效地完成上述工作，Sphinx 允许给文档附加一些额外的值，并把这些值存储在全文索引中，以便在对全文匹配结果进行过滤、排序或分组时使用。

论坛帖子表是一个很好的例子。假设只有帖子的标题和内容这两个字段需要全文检索，但是有时检索结果需要被限制在某个特定的作者的帖子或者属于某个子论坛的帖子中（也就是说，只检索在 SQL 表的 author_id 和 forum_id 这两个列上有特定值的那些行），或者需要按 post_date 列对匹配的结果排序，或者根据 post_date 列对帖子按月份分组，并对每组中的帖子计数。

为实现这些功能，可以将上述各列（除了标题和内容列）作为属性做索引，之后即可使用 API 调用来设置过滤、排序和分组。以下是一个例子：

示例：sphinx.conf 片断:

```
...
sql_query = SELECT id, title, content, \
       author_id, forum_id, post_date FROM my_forum_posts
sql_attr_uint = author_id
sql_attr_uint = forum_id
sql_attr_timestamp = post_date
...
```

示例： 应用程序代码 (PHP):

```
// only search posts by author whose ID is 123
$cl->SetFilter ( "author_id", array ( 123 ) );
```

```
// only search posts in sub-forums 1, 3 and 7
$cl->SetFilter ( "forum_id", array ( 1,3,7 ) );

// sort found posts by posting date in descending order
$cl->SetSortMode ( SPH_SORT_ATTR_DESC, "post_date" );
```

可以通过名字来指示特定的属性，并且这个名字是大小写无关的（注意：直到目前为止，Sphinx 还不支持中文作为属性的名称）。属性并不会被全文索引，他们只是按原封不动的存储在索引文件中。目前支持的属性类型如下：

- 无符号整数（1-32 位宽）
- UNIX 时间戳（timestamps）
- 浮点值（32 位，IEEE 754 单精度）
- 字符串叙述 (尤其是计算出的整数值);
- 多值属性 MVA（multi-value attributes）（32 位无符号整形值的变长序列）.

由各个文档的全部的属性信息构成了一个集合，它也被称为文档信息（docinfo），docinfo 可以按如下两种方式之一存储：

- 与全文索引数据分开存储（“外部存储”，在.spa 文件中存储）
- 在全文索引数据中，每出现一次文档 ID 就出现相应的文档信息（“内联存储”，在.spd 文件中存储）.

当采用外部存储方式时，searchd 总是在 RAM 中保持一份.spa 文件的拷贝（该文件包含所有文档的所有文档信息）。这是主要是为了提高性能，因为磁盘的随机访问太慢了。相反，内联存储并不需要任何额外的 RAM，但代价是索引文件的体积大大地增加了；请注意，**全部**属性值在文档 ID 出现的**每一处**都被复制了一份，而文档 ID 出现的次数恰是文档中不同关键字的数目。仅当有一个很小的属性集、庞大的数据集和受限的 RAM 时，内联存储才是一个可考虑的选择。在大多数情况下，外部存储可令建立索引和检索的效率都**大幅**提高。

检索时采用外部存储方式产生的的内存需求为 (1+number_of_attrs)*number_of_docs*4 字节，也就是说，带有两个属性和一个时间戳的 1 千万篇文档会消耗(1+2+1)*10M*4 = 160 MB 的 RAM。这是**每个检索的守护进程（daemon）**消耗的量，而不是每次查询，searchd 仅在启动时分配 160MB 的内存，读入数据并在不同的查询之间保持这些数据。子进程并**不会**对这些数据做额外的拷贝。

3.3. 多值属性 MVA (multi-valued attributes)

多值属性 MVA (multi-valued attributes)是文档属性的一种重要的特例，MVA 使得向文档附加一系列的值作为属性的想法成为可能。这对文章的 tags，产品类别等等非常有用。MVA 属性支持过滤和分组（但不支持分组排序）。

目前 MVA 列表项的值被限制为 32 位无符号整数。列表的长度不受限制，只要有足够的 RAM，任意个数的值都可以被附加到文档上（包含 MVA 值的.spm 文件会被 searchd 预缓冲到 RAM 中）。源数据既可以来自一个单独的查询，也可以来自文档属性，参考 sql_attr_multi 中的源类型。在第一种情况中，该查询须返回文档 ID 和 MVA 值的序对；而在第二种情况中，该字段被分析为整型值。对于多值属性的输入数据的顺序没有任何限制，在索引过程中这些值会自动按文档 ID 分组（而相同文档 ID 下的数据也会排序）。

在过滤过程中，MVA 属性中的**任何一个**值满足过滤条件，则文档与过滤条件匹配（因此通

过排他性过滤的文档不会包含任何被禁止的值）。按 MVA 属性分组时，一篇文档会被分到与多个不同 MVA 值对应的多个组。例如，如果文档集只包含一篇文档，它有一个叫做 tag 的 MVA 属性，该属性的值是 5、7 和 11，那么按 tag 的分组操作会产生三个组，它们的 @count 都是 1，@groupby 键值分别是 5、7 和 11。还要注意，按 MVA 分组可能会导致结果集中有重复的文档：因为每篇文文档可能属于不同的组，而且它可能在多个组中被选为最佳结果，这会导致重复的 ID。由于历史原因，PHP API 对结果集的行进行按文档 ID 的有序 hash，因此用 PHP API 进行对 MVA 属性的分组操作时你还需要使用 SetArrayResult()

3.4. 索引

为了快速地相应查询，Sphinx 需要从文本数据中建立一种为查询做优化的特殊的数据结构。这种数据结构被称为**索引（index）**；而建立索引的过程也叫做**索引**或**建立索引**（indexing）。

不同的索引类型是为不同的任务设计的。比如，基于磁盘的 B-Tree 存储结构的索引可以更新起来比较简单（容易向已有的索引插入新的文档），但是搜起来就相当慢。因此 Sphinx 的程序架构允许实现多种不同类型的**索引**。

目前在 Sphinx 中实现的唯一一种索引类型是为最优化建立索引和检索的速度而设计的。随之而来的代价是更新索引相当的很慢。理论上讲，更新这种索引甚至可能比从头重建索引还要慢。不过大多数情况下这可以靠建立多个索引来解决索引更新慢的问题，细节请参考 节 3.10, “实时索引更新”

实现更多的索引类型支持，已列入计划，其中包括一种可以实时更新的类型。

> **每个配置文件都可以按需配置足够多的索引。indexer 工具可以将它们同时重新索引（如果使用了--all 选项）或者仅更新明确指出的一个。 Searchd 工具会为所有被指明的索引提供检索服务，而客户端可以在运行时指定使用那些索引进行检索。**

3.5 源数据的限制

Sphinx 索引的源数据有一些限制，其中最重要的一条是：

所有文档的 ID 必须是唯一的无符号非零整数（根据 Sphinx 构造时的选项，可能是 32 位或 64 位）

如果不满足这个要求，各种糟糕的情况都可能发生。例如，Sphinx 建立索引时可能在突然崩溃，或者由于冲突的文档 ID 而在索引结果中产生奇怪的结果。也可能，一只重达 1000 磅的大猩猩最后跳出你的电脑，向你扔木桶。我告诉你咯！

3. 6 字符集、大小写转换和转换表

当建立索引时，Sphinx 从指定的数据源获得文本文档，将文本分成词的集合，再对每个词做大小写转换，于是“Abc”，“ABC”和“abc”都被当作同一个词（word，或者更学究一点，词项 term）

为了正确完成上述工作，Sphinx 需要知道：

- 源文本是什么编码的
- 那些字符是字母，哪些不是
- 哪些字符需要被转换，以及被转换成什么

这些都可以用 charset_type 和 charset_table 选项为每个索引单独配置。

charset_type 指定文档的编码是单字节的（SBCS）还是 UTF-8 的（注意：Sphinx 目前只支持这两种编码，GBK 及 BIG5 的编码需要预先转成 UTF-8 的）。charset_table 则指定了字母类字符到它们的大小写转换版本的对应表，没有在这张表中出现的字符被认为是非字母类字符，并且在建立索引和检索时被当作词的分割符来看待。

注意，尽管默认的转换表并不包含空格符 （ASCII 0x20, Unicode U+0020），但这么做是**完全合法**的。这在某些情况下可能有用，比如在对 tag 云构造索引的时候，这样一个用空格分开的词集就可以被当作一个**单独的**查询项了。

默认转换表目前包括英文和俄文字符。请您提交您为其他语言撰写的转换表！

3.7. SQL 数据源 (MySQL, PostgreSQL)

对于所有的 SQL 驱动，建立索引的过程如下：

- 连接到数据库
- 执行预查询 (参见 节 8.1.9, “sql_query_pre”)，以便完成所有必须的初始设置，比如为 MySQL 连接设置编码
- 执行主查询（参见节 8.1.10, “sql_query”），其返回的的数据将被索引
- 执行后查询 (参见 节 8.1.19, “sql_query_post”)，以便完成所有必须的清理工作
- 关闭到数据库的连接
- 对短语进行排序 (或者学究一点, 索引类型相关的后处理)
- 再次建立到数据库的连接
- 执行后索引查询 (参见 节 8.1.20, “sql_query_post_index”)，以便完成所有最终的清理工作
- 再次关闭到数据库的连接

大多数参数是很直观的，例如数据库的用户名、主机、密码。不过，还有一些细节上的问题需要讨论。

分区查询

Sphinx 需要通过主查询来获取全部的文档信息，一种简单的实现是将整个表的数据读入内存，但是这可能导致整个表被锁定并使得其他操作被阻止（例如：在 MyISAM 格式上的 INSERT 操作），同时，将浪费大量内存用于存储查询结果，诸如此类的问题吧。为了避免出现这种情况，Sphinx 支持一种被称作“分区查询”的技术。首先，Sphinx 从数据库中取出文档 ID 的最小值和最大值，将由最大值和最小值定义自然数区间分成若干份，一次获取数据，建立索引。现举例如下：

例 **1.** 使用分区查询的例子

```
# in sphinx.conf

sql_query_range = SELECT MIN(id),MAX(id) FROM documents
sql_range_step = 1000
sql_query = SELECT * FROM documents WHERE id>=$start AND id<=$end
```

如果这个表（documents）中，字段 ID 的最小值和最大值分别是 1 和 2345，则 sql_query 将执行 3 次：

1. 将$start 替换为 1，并且将$end 替换为 1000;

2. 将$start 替换为 1001，并且将 $end 替换为 2000;
3. 将$start 替换为 2000，并且将 $end 替换为 2345.

显然，这对于只有 2000 行的表，分区查询与整个读入没有太大区别，但是当表的规模扩大到千万级（特别是对于 MyISAM 格式的表），分区查询将提供一些帮助。

后查询（sql_post） vs. 索引后查询（sql_post_index）

后查询和索引后查询的区别在于，当 Sphinx 获取到全部文档数据后，立即执行后查询，但是构建索引的过程仍然可能因为某种原因失败。在另一方面，当索引后查询被执行时，可以理所当然的认为索引已经成功构造完了。因为构造索引可能是个漫长的过程，因此对与数据库的连接在执行后索引操作后被关闭，在执行索引后操作前被再次打开。

3.8. xmlpipe 数据源

xmlpipe 数据源是处于让用户能够将现有数据嵌入 Sphinx 而无需开发新的数据源驱动的目的被设计和提供的。它将每篇文档限制为只能包括两个可全文索引的字段，以及只能包括两个属性。xmlpipe 数据源已经被废弃，在节 3.9, “xmlpipe2 数据源” 中描述了 xmlpipe 的替代品 xmlpipe2 数据源。对于新的数据，建议采用 xmlpipe2。.

为了使用 xmlpipe，需要将配置文件改为类似如下的样子：

```
source example_xmlpipe_source
{
    type = xmlpipe
    xmlpipe_command = perl /www/mysite.com/bin/sphinxpipe.pl
}
```

indexer 实用程序将要运行 xmlpipe_command 所指定的命令，而后读取其向标准输出上输出的数据，并对之进行解析并建立索引。严格的说，是 Sphinx 打开了一个与指定命令相连的管道，并从这个管道读取数据。

indexer 实用程序假定在从标准输入读入的 XML 格式的数据中中存在一个或更多的文档。下面是一个包括两个文档的文档数据流的例子：

Example 2. XMLpipe 文档数据流

```
<document>
<id>123</id>
<group>45</group>
<timestamp>1132223498</timestamp>
<title>test title</title>
<body>
this is my document body
</body>
</document>

<document>
<id>124</id>
<group>46</group>
<timestamp>1132223498</timestamp>
<title>another test</title>
<body>
this is another document
</body>
</document>
```

遗留的 xmlpipe 的遗留的数据驱动使用内置的解析器来解析 xml 文档，这个解析器的速度非常快，但是并没有提供对 XML 格式完整支持。这个解析器需要文档中包括全部的字段，并且**严格**按照例子中给出的格式给出，而且字段的出现顺序需要**严格**按照例子中给出的顺序。仅有一个字段 timestamp 是可选的，它的缺省值为 1。

3.9. xmlpipe2 数据源

xmlpipe2 使你可以用另一种自定义的 XML 格式向 Sphinx 传输任意文本数据和属性数据。数据模式（即数据字段的集合或者属性集）可以由 XML 流本身指定，也可以在配置文件中数据源的配置部分中指定。

Xmlpipe2

在对 xmlpipe2 数据源做索引时，索引器运行指定的命令，打开一个连接前述命令标准输出的管道，并等待接受具有正确格式的 XML 数据流。以下是一个数据流的样本：

示例 **3 xmlpipe2** 文档流

```
<?xml version="1.0" encoding="utf-8"?>
<sphinx:docset>

<sphinx:schema>
<sphinx:field name="subject"/>
<sphinx:field name="content"/>
<sphinx:attr name="published" type="timestamp"/>
<sphinx:attr name="author_id" type="int" bits="16" default="1"/>
</sphinx:schema>

<sphinx:document id="1234">
<content>this is the main content <![CDATA[[and this <cdata> entry must be
handled properly by xml parser lib]]></content>
<published>1012325463</published>
<subject>note how field/attr tags can be in <b class="red">randomized</b>
order</subject>
<misc>some undeclared element</misc>
</sphinx:document>

<!-- ... more documents here ... -->

</sphinx:docset>
```

任意多的数据字段和属性都是允许的。数据字段和属性在同一文档元素中出现的先后顺序没有特别要求。。单一字段数据的最大长度有限制，超过 2MB 的数据会被截短到 2MB（但这个限制可以在配置文件中数据源部分修改）

数据模式，即数据字段和属性的完整列表，必须在任何文档被分析之前就确定。这既可以在配置文件中用 xmlpipe_field 和 xmlpipe_attr_xxx 选项指定，也可以就在数据流中用 <sphinx:schema>元素指定。 <sphinx:schema>元素是可选的，但如果出现，就必须是 <sphinx:docset>元素的第一个子元素。如果没有在数据流中内嵌的数据模式定义，配置文件中的相关设置就会生效，否则数据流内嵌的设置被优先采用。

未知类型的标签（既不是数据字段，也不是属性的标签）会被忽略，但会给出警告。在上面的例子中，<misc>标签会被忽略。所有嵌入在其他标签中的标签及其属性都会被无视（例如上述例子中嵌入在<subject>标签中的<b>标签）

支持输入数据流的何种字符编码取决于系统中是否安装了 iconv。xmlpipe2 是用 libexpat 解析

器解析的，该解析器内置对 US-ASCII，ISO-8859-1，UTF-8 和一些 UTF-16 变体的支持。Sphinx 的 configure 脚本也会检查 libiconv 是否存在并使用它来处理其他的字符编码。libexpat 也隐含的要求在 Sphinx 端使用 UTF-8，因为它返回的分析过的数据总是 UTF-8 的。

xmlpipe2 可以识别的 XML 元素（标签）（以及前述元素可用的属性）如下：

sphinx:docset

顶级元素，用于标明并包括 xmlpipe2 文档。

Sphinx:schema

可选元素，它要么是 sphinx:docset 的第一个子元素，要么干脆不出现。声明文档的模式。包括数据字段和属性的声明。若此元素出现，则它会覆盖配置文件中对数据源的设定。

Sphinx:field

可选元素，sphinx:schema 的子元素。声明一个全文数据字段。唯一可识别的属性是“name”，它指定了字段的名称，后续数据文档中具有此名称的元素的数据都被当作待检索的全文数据对待。

sphinx:attr

可选元素，sphinx:schema 的子元素。用于声明具体属性。其已知的属性有：

- “name”，设定该属性名称，后续文档中具有该名称的元素应被当作一个属性对待。
- ”type”，设定该属性的类型。可能的类型包括“int”，“timestamp”，“str2ordinal”，“bool”和“float”
- “bits”，设定“int”型属性的宽度，有效值为 1 到 32
- “default”，设定该属性的默认值，若后续文档中没有指定这个属性，则使用此默认值。

Sphinx:document

必须出现的元素，必须是 sphinx:docset 的子元素。包含任意多的其他元素，这些子元素带有待索引的数据字段和属性值，而这些数据字段或属性值既可以是用 sphinx:field 和 sphinx:attr 元素声明的，也可以在配置文件中声明。唯一的已知属性是“id”，它必须包含一个唯一的整型的文档 ID。

3.10. 实时索引更新

有这么一种常见的情况：整个数据集非常大，以至于难于经常性的重建索引，但是每次新增的记录却相当地少。一个典型的例子是：一个论坛有 1000000 个已经归档的帖子，但每天只有 1000 个新帖子。

在这种情况下可以用所谓的“主索引＋增量索引”（main+delta）模式来实现“近实时”的索引更新。

这种方法的基本思路是设置两个数据源和两个索引，对很少更新或根本不更新的数据建立主索引，而对新增文档建立增量索引。在上述例子中，那 1000000 个已经归档的帖子放在主索引中，而每天新增的 1000 个帖子则放在增量索引中。增量索引更新的频率可以非常快，而文档可以在出现几分种内就可以被检索到。

确定具体某一文档的分属那个索引的分类工作可以自动完成。一个可选的方案是，建立一个计数表，记录将文档集分成两部分的那个文档 ID，而每次重新构建主索引时，这个表都会

被更新。

示例 **4** 全自动的即时更新

```
# in MySQL
CREATE TABLE sph_counter
(
    counter_id INTEGER PRIMARY KEY NOT NULL,
    max_doc_id INTEGER NOT NULL
);

# in sphinx.conf
source main
{
    # ...
    sql_query_pre = REPLACE INTO sph_counter SELECT 1, MAX(id) FROM documents
    sql_query = SELECT id, title, body FROM documents \
        WHERE id<=( SELECT max_doc_id FROM sph_counter WHERE counter_id=1 )
}

source delta : main
{
    sql_query_pre =
    sql_query = SELECT id, title, body FROM documents \
        WHERE id>( SELECT max_doc_id FROM sph_counter WHERE counter_id=1 )
}

index main
{
    source = main
    path = /path/to/main
    # ... all the other settings
}

# note how all other settings are copied from main,
# but source and path are overridden (they MUST be)
index delta : main
{
    source = delta
    path = /path/to/delta
}
```

3.11. 索引合并

合并两个已有的索引比重新对所有数据做索引更有效率，而且有时候必须这样做（例如在“主索引＋增量索引”分区模式中应合并主索引和增量索引，而不是简单地重新索引“主索引对应的数据）。因此 indexer 有这个选项。合并索引一般比重新索引快，但在大型索引上仍然**不是**一蹴而就。基本上，待合并的两个索引都会被读入内存一次，而合并后的内容需要写入磁盘一次。例如，合并 100GB 和 1GB 的两个索引将导致 202GB 的 IO 操作（但很可能还是比重新索引少）

基本的命令语法如下：

```
indexer --merge DSTINDEX SRCINDEX [--rotate]
```

SRCINDEX 的内容被合并到 DSTINDEX 中，因此只有 DSTINDEX 索引会被改变。若 DSTINDEX 已经被 searchd 用于提供服务，则--rotate 参数是必须的。最初设计的使用模式是，将小量的更新从 SRCINDEX 合并到 DSTINDEX 中。因此，当属性被合并时，一旦出现了重复的文档 ID，SRCINDEX 中的属性值更优先（会覆盖 DSTINDEX 中的值）。不过要注意，“旧的”关键字并不会被自动删除。例如，在 DSTINDEX 中有一个叫做“old”的关键字与文档 123 相关联，而在 SRCINDEX 中则有关键字“new”与同一个文档相关，那么在合并后用这两个关键字都能找到文档 123。您可以给出一个显式条件来将文档从 DSTINDEX 中移除，以便应对这种情况，相关的开关是--merge-dst-range:

```
indexer --merge main delta --merge-dst-range deleted 0 0
```

这个开关允许您在合并过程中对目标索引实施过滤。过滤器可以有多个，只有满足全部过滤条件的文档才会在最终合并后的索引中出现。在上述例子中，过滤器只允许“deleted”为 0 的那些条件通过，而去除所有标记为已删除（“deleted”）的记录（可以通过调用 UpdateAttributes() 设置文档的属性）。

4. 搜索

4.1. 匹配模式

有如下可选的匹配模式：

- SPH_MATCH_ALL, 匹配所有查询词（默认模式）
- SPH_MATCH_ANY, 匹配查询词中的任意一个
- SPH_MATCH_PHRASE, 将整个查询看作一个词组，要求按顺序完整匹配
- SPH_MATCH_BOOLEAN, 将查询看作一个布尔表达式 （参见 节 4.2, “布尔查询语法”
- SPH_MATCH_EXTENDED, 将查询看作一个 Sphinx 内部查询语言的表达式（参见 节 4.3, “扩展的查询语法”）

还有一个特殊的“完整扫描”模式，当如下条件满足时，该模式被自动激活：

1. 查询串是空的（即长度为零）
2. docinfo 存储方式为 extern

在完整扫描模式中，全部已索引的文档都被看作是匹配的。这类匹配仍然会被过滤、排序或分组，但是并不会做任何真正的全文检索。这种模式可以用来统一全文检索和非全文检索的代码，或者减轻 SQL 服务器的负担（有些时候 Sphinx 扫描的速度要优于类似的 MySQL 查询）

4.2. 布尔查询

布尔查询允许使用下列特殊操作符：

- 显式的与（AND）操作符

```
hello & world
```

- 或（OR）操作符

```
hello | world
```

- 非（NOT）操作符

```
hello -world
hello !world
```

- 分组（grouping）

```
( hello world )
```

以下是一个使用了如上全部操作符的例子：

示例 5 布尔查询示例

```
( cat -dog ) | ( cat -mouse)
```

与(AND)操作符为默认操作，所以“hello world”其实就是“hello & world”

或(OR)操作符的优先级高于与操作符，因此“lookingfor cat | dog | mouse”意思是"looking for (cat | dog | mouse)" 而不是 "(looking for cat) | dog | mouse"

像“-dog”这种查询不能被执行，因为它差不多包括索引所有文档。这既有技术上的原因，也有性能上的原因。从技术上说，Sphinx 并不总是保持一个全部文档 ID 的列表。性能方面，当文档集非常大的时候（即 10-100M 个文档），对这种执行查询可能需要很长的时间。

4.3. 扩展查询

在扩展查询模式中可以使用如下特殊操作符：

- 或（OR）操作符

```
hello | world
```

- 非（NOT）操作符

```
hello -world
hello !world
```

- 字段（field）搜索符：

```
@title hello @body world
```

- 词组搜索符

```
"hello world"
```

- 近似搜索符

```
"hello world"~10
```

- 阀值匹配符

```
"the world is a wonderful place"/3
```

下例使用了上述大多数操作符：

示例 6 扩展查询示例

```
"hello world" @title "example program"~5 @body python -(php|perl)
```

与(AND)操作为默认操作，因此“hello world”意思是“hello”和“world”必须同时存在文档才

能匹配。

或（OR）操作符的优先级要高于与操作符，因此"looking for cat | dog | mouse" 意思是"looking for (cat | dog | mouse)" 而**不是**"(looking for cat) | dog | mouse"；

近似距离以词为单位，随词数变化而变化，并应用于引号中的全部词。举个例子，"cat dog mouse"~5 这个查询的意思是必须有一个少于 8 个词的词串，它要包含全部的三个词，也就是说"CAT aaa bbb ccc DOG eee fff MOUSE" 这个文档**不会**匹配这个查询，因为这个词串正好是 8 个词。

阀值匹配符引入了一种模糊匹配。它允许至少含有某个阈值数量个匹配词的文档通过。上述例子（"the world is a wonderful place"/3）会匹配含有指定的六个词中的至少三个的那些文档。

类似如下查询中嵌套的括号

```
aaa | ( bbb ccc | ( ddd eee ) )
```

目前还是不允许的，但是这会在今后得以改进。

取否（也就是非（NOT）操作符）只允许用在最外一层且不能在括号（也就是分组）里。这个特性不会改变，因为支持嵌套的非操作会大大提高词组相关度计算实现的复杂性。

4.4. 权值计算

采用何种权值计算函数（目前）取决于查询的模式。

There are these major parts which are used in the weighting functions:

权值计算函数进行如下两部分主要部分：

1. 词组评分
2. 统计学评分

词组评分根据文档和查询的最长公共子串（LCS，longest common subsequence）的长度进行。因此如果文档对查询词组有一个精确匹配（即文档直接包含该词组），那么它的词组评分就取得了可能的最大值，也就是查询中词的个数。

统计学评分基于经典的 BM25 函数，该函数仅考虑词频。如果某词在整个数据库中很少见（即文档集上的低频词）或者在某个特定文档中被经常提及（即特定文档上的高频词），那么它就得到一个较高的权重。最终的 BM25 权值是一个 0 到 1 之间的浮点数。

在所有模式中，数据字段的词组评分是 LCS 乘以用户指定的数据字段权值。数据字段权值是整数，默认为 1，且字段的权值必须不小于 1。

在 SPH_MATCH_BOOLEAN 模式中，不做任何权重估计，每一个匹配项的权重都是 1。

在 SPH_MATCH_ALL 和 SPH_MATCH_PHRASE 模式中，最终的权值是词组评分的加权和。

在 SPH_MATCH_ANY 模式中，于前面述两模式的基本思想类似，只是每个数据字段的权重都再加上一个匹配词数目。在那之前，带权的词组相关度被额外乘以一个足够大的数，以便确保任何一个有较大词组评分的数据字段都会使整个匹配的相关度较高，即使该数据字段的权重比较低。

在 SPH_MATCH_EXTENDED 模式中，最终的权值是带权的词组评分和 BM25 权重的和，再乘以 1000 并四舍五入到整数。

这个行为将会被修改，以便使 MATCH_ALL 和 MATCH_ANY 这两个模式也能使用 BM25 算

法。这将使词组评分相同的搜索结果片断得到改进，这在只有一个词的查询中尤其有用。

关键的思想（对于除布尔模式以外的全部模式中）是子词组的匹配越好则评分越高，精确匹配（匹配整个词组）评分最高。作者的经验是，这种基于词组相似性的评分方法可以提供比任何单纯的统计模型（比如其他搜索引擎中广泛使用的 BM25）明显更高的搜索质量。

4.5. 排序模式

可使用如下模式对搜索结果排序：

- SPH_SORT_RELEVANCE 模式, 按相关度降序排列（最好的匹配排在最前面）
- SPH_SORT_ATTR_DESC 模式, 按属性降序排列 （属性值越大的越是排在前面）
- SPH_SORT_ATTR_ASC 模式, 按属性升序排列（属性值越小的越是排在前面）
- SPH_SORT_TIME_SEGMENTS 模式, 先按时间段（最近一小时/天/周/月）降序，再按相关度降序
- SPH_SORT_EXTENDED 模式, 按一种类似 SQL 的方式将列组合起来，升序或降序排列。
- SPH_SORT_EXPR 模式，按某个算术表达式排序。

SPH_SORT_RELEVANCE 忽略任何附加的参数，永远按相关度评分排序。所有其余的模式都要求额外的排序子句，子句的语法跟具体的模式有关。SPH_SORT_ATTR_ASC, SPH_SORT_ATTR_DESC 以及 SPH_SORT_TIME_SEGMENTS 这三个模式仅要求一个属性名。SPH_SORT_RELEVANCE 模式等价于在扩展模式中按"@weight DESC, @id ASC"排序，SPH_SORT_ATTR_ASC 模式等价于"attribute ASC, @weight DESC, @id ASC"，而 SPH_SORT_ATTR_DESC 等价于"attribute DESC, @weight DESC, @id ASC"。

SPH_SORT_TIME_SEGMENTS 模式

在 SPH_SORT_TIME_SEGMENTS 模式中，属性值被分割成“时间段”，然后先按时间段排序，再按相关度排序。

时间段是根据搜索发生时的**当前时间戳**计算的，因此结果随时间而变化。所说的时间段有如下这些值：

- 最近一小时
- 最近一天
- 最近一星期
- 最近一个月
- 最近三个月
- 其他值

时间段的分法固化在搜索程序中了，但如果需要，也可以比较容易地改变（需要修改源码）。

这种模式是为了方便对 Blog 日志和新闻提要等的搜索而增加的。使用这个模式时，处于更近时间段的记录会排在前面，但是在同一时间段中的记录又根据相关度排序－这不同于单纯按时间戳排序而不考虑相关度。

SPH_SORT_EXTENDED 模式

在 SPH_SORT_EXTENDED 模式中，您可以指定一个类似 SQL 的排序表达式，但涉及的属

性（包括内部属性）不能超过 5 个，例如：

```
@relevance DESC, price ASC, @id DESC
```

只要做了相关设置，不管是内部属性（引擎动态计算出来的那些属性）还是用户定义的属性就都可以使用。内部属性的名字必须用特殊符号@开头，用户属性按原样使用就行了。在上面的例子里，@relevance 和@id 是内部属性，而 price 是用户定义属性。

已知的内部属性：

- @id (match ID)
- @weight (match weight)
- @rank (match weight)
- @relevance (match weight)

- @id（匹配的 ID）
- @weight（匹配权值）
- @rank（匹配权值）
- @relevance（匹配权值）

@rank 和@relevance 只是@weight 的额外别名。

SPH_SORT_EXPR 模式

表达式排序模式使您可以对匹配项按任何算术表达式排序，表达式中的项可以是属性值，内部属性（@id 和@weight），算术运算符和一些内建的函数。例如：

```
$cl->SetSortMode ( SPH_SORT_EXPR,
       "@weight + ( user_karma + ln(pageviews) )*0.1" );
```

支持的运算符和函数如下。它们是模仿 MySQL 设计的。函数接受参数，参数的数目根据具体函数的不同而不同。

- Operators: +, -, *, /, <, > <=, >=, =, <>.
- Unary (1-argument) functions: abs(), ceil(), floor(), sin(), cos(), ln(), log2(), log10(), exp(), sqrt().
- Binary (2-argument) functions: min(), max(), pow().
- Ternary (3-argument) functions: if().

- 运算符： +, -, *, /, <, > <=, >=, =, <>.
- 一元函数（一个参数）：abs(), ceil(), floor(), sin(), cos(), ln(), log2(), log10(), exp(), sqrt().
- 二元函数（两个参数）：min(), max(), pow().
- 三元函数（三个参数）：if().

全部的计算都以单精度 32 位 IEEE754 浮点数进行。比较操作符（比如=和<=）在条件为真时返回 1.0，否则返回 0.0。例如 (a=b)+3 在属性“a”与属性“b”相等时返回 4，否则返回 3。与 MySQL 不同，相等性比较符（即=和<>）中引入了一个小的阈值（默认是 1e-6）。如果被比较的两个值的差异在阈值之内，则二者被认为相等。

全部的一元和二元函数的意义都很明确，他们的行为跟在数学中的定义一样。但 IF()的行为

需要点详细的解释。它接受 3 个参数，检查第一个参数是否为 0.0，若非零则返回第二个参数，为零时则返回第三个参数。注意，与比较操作符不同，IF()并不使用阈值！因此在第一个参数中使用比较结果是安全的，但使用算术运算符则可能产生意料之外的结果。比如，下面两个调用会产生不同的结果，虽然在逻辑上他们是等价的：

```
IF ( sqrt(3)*sqrt(3)-3<>0, a, b )
IF ( sqrt(3)*sqrt(3)-3, a, b )
```

在第一种情况下，由于有阈值，比较操作符<>返回 0.0（逻辑假），于是 IF()总是返回‘b’。在第二种情况下，`IF()`函数亲自在没有阈值的情况下将同样的`sqrt(3)*sqrt(3)-3`与零值做比较。但由于浮点数运算的精度问题，该表达式的结果与`0`值会有微小的差异，因此该值与零值的相等比较不会通过，上述第二种情况中`IF()`会返回‘a’做为结果。

4.6. 结果分组(聚类)

有时将搜索结果分组（或者说“聚类”）并对每组中的结果计数是很有用的一例如画个漂亮的图来展示每个月有多少的 blog 日志，或者把 Web 搜索结果按站点分组，或者把找到的论坛帖子按其作者分组。

理论上，这可以分两步实现：首先在 Sphinx 中做全文检索，再在 SQL 服务器端对得到的 ID 分组。但是现实中在大结果集（10K 到 10M 个匹配）上这样做通常会严重影响性能。

为避免上述问题，Sphinx 提供了一种“分组模式”，可以用 API 调用 SetGroupBy()来开启。在分组时，根据 group-by 值给匹配项赋以一个分组。这个值用下列内建函数之一根据特定的属性值计算：

- SPH_GROUPBY_DAY，从时间戳中按 YYYYMMDD 格式抽取年、月、日
- SPH_GROUPBY_WEEK，从时间戳中按 YYYYNNN 格式抽取年份和指定周数（自年初计起）的第一天
- SPH_GROUPBY_MONTH，从时间戳中按 YYYYMM 格式抽取月份
- SPH_GROUPBY_YEAR，从时间戳中按 YYYY 格式抽取年份
- SPH_GROUPBY_ATTR，使用属性值自身进行分组

最终的搜索结果中每组包含一个最佳匹配。分组函数值和每组的匹配数目分别以“虚拟”属性**@group** 和**@count** 的形式返回。

结果集按 group-by 排序子句排序，语法与 SPH_SORT_EXTENDED 排序子句的语法相似。除了@id 和@weight，分组排序子句还包括：

- @group（groupby 函数值）
- @count（组中的匹配数目）

默认模式是根据 groupby 函数值降序排列，即按照“@group desc”

排序完成时，结果参数 total_found 会包含在整个索引上匹配的组的总数目。

注意：分组操作在固定的内存中执行，因此它给出的是近似结果；所以 total_found 报告的数目可能比实际给出的个分组数目的和多。@count 也可能被低估。要降低不准确性，应提高 max_matches。如果 max_matches 允许存储找到的全部分组，那结果就是百分之百准确的。

例如，如果按相关度排序，同时用 SPH_GROUPBY_DAY 函数按属性“published”分组，那么：

- 结果中包含每天的匹配结果中最相关的那一个，如果那天有记录匹配的话。
- 结果中还附加给出天的编号和每天的匹配数目
- 结果以天的编号降序排列（即最近的日子在前面）

4.7. 分布式搜索

为提高可伸缩性，Sphnix 提供了分布式检索能力。分布式检索可以改善查询延迟问题（即缩短查询时间）和提高多服务器、多 CPU 或多核环境下的吞吐率（即每秒可以完成的查询数）。这对于大量数据（即十亿级的记录数和 TB 级的文本量）上的搜索应用来说是很关键的。

其关键思想是将待搜索数据做水平分区（HP，Horizontally partition），然后并行处理。

分区不能自动完成，您需要：

- 在不同服务器上设置 Sphinx 程序集（indexer 和 searchd）的多个实例
- 让这些实例对数据的不同部分做索引（并检索）
- 在 searchd 的一些实例上配置一个特殊的分布式索引
- 然后对这个索引进行查询

这个特殊索引只包括对其他本地或远程索引的引用，因此不能对它执行重新建立索引的操作，相反，如果要对这个特殊索引进行重建，要重建的是那些被这个索引被引用到的索引。

当 searchd 收到一个对分布式索引的查询时，它做如下操作：

1. 连接到远程代理
2. 执行查询
3. （在远程代理执行搜索的同时）对本地索引进行查询
4. 接收来自远程代理的搜索结果
5. 将所有结果合并，删除重复项
6. 将合并后的结果返回给客户端

在应用程序看来，普通索引和分布式索引完全没有区别。

任一个 searchd 实例可以同时做为主控端（master，对搜索结果做聚合）和从属端（只做本地搜索）。这有如下几点好处：

1. 集群中的每台机器都可以做为主控端来搜索整个集群，搜索请求可以在主控端之间获得负载平衡，相当于实现了一种 HA（high availability，高可用性），可以应对某个节点失效的情况。
2. 如果在单台多 CPU 或多核机器上使用，一个做为代理对本机进行搜索的 searchd 实例就可以利用到全部的 CPU 或者核。

更好的 HA 支持已在计划之中，到时将允许指定哪些代理之间互相备份、有效性检查、跟踪运行中的代理、对检索请求进行负载均衡，等等。

4.8. `searchd` 日志格式

searchd 将全部成功执行的搜索查询都记录在查询日志文件中。以下是一个类似记录文件的例子：

```
[Fri Jun 29 21:17:58 2007] 0.004 sec [all/0/rel 35254 (0,20)] [lj] test
[Fri Jun 29 21:20:34 2007] 0.024 sec [all/0/rel 19886 (0,20) @channel_id] [lj]
test
```

日志格式如下

```
[query-date] query-time [match-mode/filters-count/sort-mode
   total-matches (offset,limit) @groupby-attr] [index-name] query
```

匹配模式（match-mode）可以是如下值之一：

- "all" 代表 SPH_MATCH_ALL 模式;
- "any" 代表 SPH_MATCH_ANY 模式;
- "phr" 代表 SPH_MATCH_PHRASE 模式;
- "bool" 代表 SPH_MATCH_BOOLEAN 模式;
- "ext" 代表 SPH_MATCH_EXTENDED 模式.

排序模式（sort-mode）可以取如下值之一：

- "rel" 代表 SPH_SORT_RELEVANCE 模式;
- "attr-" 代表 SPH_SORT_ATTR_DESC 模式;
- "attr+" 代表 SPH_SORT_ATTR_ASC 模式;
- "tsegs" 代表 SPH_SORT_TIME_SEGMENTS 模式;
- "ext" 代表 SPH_SORT_EXTENDED 模式.

5. API 参考

Sphnix 有几种不同编程语言的 searchd 客户端 API 的实现。在本文完成之时，我们对我们自己的 PHP，Python 和 java 实现提供官方支持。此外，也有一些针对 Perl，Ruby 和 C++的第三方免费、开源 API 实现。

API 的参考实现是用 PHP 写成的，因为（我们相信）较之其他语言，Sphinx 在 PHP 中应用最广泛。因此这份参考文档基于 PHP API 的参考，而且这节中的所有的代码样例都用 PHP 给出。

当然，其他所有 API 都提供相同的方法，也使用完全相同的网络协议。因此这份文档对他们同样适用。在方法命名习惯方面或者具体数据结构的使用上可能会有小的差别。但不同语言的 API 提供的功能上绝不会有差异。

5.1. 通用 API 方法

5.1.1. GetLastError

原型： function GetLastError()

以人类可读形式返回最近的错误描述信息。如果前一次 API 调用没有错误，返回空字符串。

任何其他函数（如 Query()）失败后（函数失败一般返回 false），都应该调用这个函数，它将返回错误的描述。

此函数本身并不重置对错误描述，因此如有必要，可以多次调用。

5.1.2. GetLastWarning

原型: function GetLastWarning ()

Returns last warning message, as a string, in human readable format. If there were no warnings during the previous API call, empty string is returned.

以人类可读格式返回最近的警告描述信息。如果前一次 API 调用没有警告，返回空字符串。

您应该调用这个函数来确认您的请求（如 Query()）是否虽然完成了但产生了警告。例如，即使几个远程代理超时了，对分布式索引的搜索查询也可能成功完成。这时会产生一个警告信息。

此函数本身**不会**重置警告信息，因此如有必要，可以多次调用。

5.1.3. SetServer

原型： function SetServer ($host, $port)

设置 searchd 的主机名和 TCP 端口。此后的所有请求都使用新的主机和端口设置。默认的主机和端口分别是“localhost”和 3312。

5.1.4. SetRetries

原型: function SetRetries ($count, $delay=0)

设置分布式搜索重试的次数和延迟时间。

对于暂时的失败，searchd 对每个代理重试至多$count 次。$delay 是两次重试之间延迟的时间，以毫秒为单位。默认情况下，重试是禁止的。注意，这个调用**不会**使 API 本身对暂时失败进行重试，它只是让 searchd 这样做。目前暂时失败包括 connect()调用的各种失败和远程代理超过最大连接数（过于繁忙）的情况。

5.1.5. SetArrayResult

原型： function SetArrayResult ($arrayresult)

PHP 专用。控制搜索结果集的返回格式（匹配项按数组返回还是按 hash 返回）

$arrayresult 参数应为布尔型。如果$arrayresult 为假（默认），匹配项以 PHP hash 格式返回，文档 ID 为键，其他信息（权重、属性）为值。如果$arrayresult 为真，匹配项以普通数组返回，包括匹配项的全部信息（含文档 ID）

这个调用是对 MVA 属性引入分组支持时同时引入的。对 MVA 分组的结果可能包含重复的文档 ID。因此需要将他们按普通数组返回，因为 hash 对每个文档 ID 仅能保存一个记录。

5.2. 通用搜索设置

5.2.1. SetLimits

原型: function SetLimits ($offset, $limit, $max_matches=0, $cutoff=0)

给服务器端结果集设置一个偏移量（$offset）和从那个偏移量起向客户端返回的匹配项数目限制（$limit）。并且可以在服务器端设定当前查询的结果集大小（$max_matches），另有一个阈值（$cutoff），当找到的匹配项达到这个阀值时就停止搜索。全部这些参数都必须是非负整数。

前两个参数的行为与 MySQL LIMIT 子句中参数的行为相同。他们令 searchd 从编号为$offset 的匹配项开始返回最多$limit 个匹配项。偏移量($offset)和结果数限制($limit)的默认值分别是 0 和 20，即返回前 20 个匹配项。

max_match 这个设置控制搜索过程中 searchd 在内存中所保持的匹配项数目。一般来说，即使设置了 max_matches 为 1，**全部**的匹配文档也都会被处理、评分、过滤和排序。但是任一时刻只有最优的 N 个文档会被存储在内存中，这是为了性能和内存使用方面的原因，这个设置正是控制这个 N 的大小。注意，max_matches 在**两个**地方设置。针对单个查询的限制由这个 API 调用指定。但还有一个针对整个服务器的限制，那是由配置文件中的 max_matches 设置控制的。为防止滥用内存，服务器不允许单个查询的限制高于服务器的限制。

在客户端不可能收到超过 max_matches 个匹配项。默认的限制是 1000，您应该不会遇到需要设置得更高的情况。1000 个记录足够向最终用户展示了。如果您是想将结果传输给应用程序以便做进一步排序或过滤，那么请注意，在 Sphinx 端完成效率要高得多。

$cutoff 设置是为高级性能优化而提供的。它告诉 searchd 在找到并处理$cutoff 个匹配后就强制停止。

5.2.2. SetMaxQueryTime

原型: function SetMaxQueryTime ($max_query_time)

设置最大搜索时间，以毫秒为单位。参数必须是非负整数。默认值为 0，意思是不做限制。

这个设置与 SetLimits()中的$cutoff 相似，不过这个设置限制的是查询时间，而不是处理的匹配数目。一旦处理时间已经太久，本地搜索查询会被停止。注意，如果一个搜索查询了多个本地索引，那这个限制独立地作用于这几个索引。

5.3. 全文搜索设置

5.3.1. SetMatchMode

原型: function SetMatchMode ($mode)

设置全文查询的匹配模式，参见节 4.1 “匹配模式”中的描述。参数必须是一个与某个已知模式对应的常数。

警告：（仅 PHP）查询模式常量**不能**包含在引号中，那给出的是一个字符串而不是一个常量。

```
$cl->SetMatchMode ( "SPH_MATCH_ANY" ); // INCORRECT! will not work as expected
$cl->SetMatchMode ( SPH_MATCH_ANY ); // correct, works OK
```

5.3.2. SetRankingMode

原型: function SetRankingMode ($ranker)

设置评分模式。目前只在 SPH_MATCH_EXTENDED2 这个匹配模式中提供。参数必须是与某个已知模式对应的常数。

Sphinx 默认计算两个对最终匹配权重有用的因子。主要是查询词组与文档文本的相似度。其次是称之为 BM25 的统计函数，该函数值根据关键字文档中的频率（高频导致高权重）和在整个索引中的频率（低频导致高权重）在 0 和 1 之间取值。

然而，有时可能需要换一种计算权重的方法——或者可能为了提高性能而根本不计算权值，结果集用其他办法排序。这个目的可以通过设置合适的相关度计算模式来达到。

已经实现的模式包括：

- SPH_RANK_PROXIMITY_BM25， 默认模式，同时使用词组评分和 BM25 评分，并且将二者结合。
- SPH_RANK_BM25，统计相关度计算模式，仅使用 BM25 评分计算（与大多数全文检索引擎相同）。这个模式比较快，但是可能使包含多个词的查询的结果质量下降。
- SPH_RANK_NONE，禁用评分的模式，这是最快的模式。实际上这种模式与布尔搜索相同。所有的匹配项都被赋予权重 1。

5.3.3. SetSortMode

原型: function SetSortMode ($mode, $sortby="")

设置匹配项的排序模式，见节 4.5, “排序模式” 中的描述。参数必须为与某个已知模式对应的常数。

警告：（仅 PHP）查询模式常量**不能**包含在引号中，那给出的是一个字符串而不是一个常量。

```
$cl->SetSortMode ( "SPH_SORT_ATTR_DESC" ); // INCORRECT! will not work as
expected
$cl->SetSortMode ( SPH_SORT_ATTR_ASC ); // correct, works OK
```

5.3.4. SetWeights

原型: function SetWeights ($weights)

按在索引中出现的先后顺序给字段设置权重。**不推荐**，请使用 SetFieldWeights() 。

5.3.5. SetFieldWeights

原型: function SetFieldWeights ($weights)

按字段名称设置字段的权值。参数必须是一个 hash（关联数组），该 hash 将代表字段名字的字符串映射到一个整型的权值上。

字段权重影响匹配项的评级。节 4.4, “权值计算” 解释了词组相似度如何影响评级。这个调用用于给不同的全文数据字段指定不同于默认值的权值。

给定的权重必须是正的 32 位整数。最终的权重也是个 32 位的整数。默认权重为 1。未知的属性名会被忽略。

目前对权重没有强制的最大限制。但您要清楚，设定过高的权值可能会导致出现 32 位整数的溢出问题。例如，如果设定权值为 10000000 并在扩展模式中进行搜索，那么最大可能的权值为 10M（您设的值）乘以 1000（BM25 的内部比例系数，参见节 4.4, “权值计算”）再乘以 1 或更多（词组相似度评级）。上述结果最少是 100 亿，这在 32 位整数里面没法存储，将导致意想不到的结果。

5.3.6. SetIndexWeights

原型: function SetIndexWeights ($weights)

设置索引的权重，并启用不同索引中匹配结果权重的加权和。参数必须为在代表索引名的字符串与整型权值之间建立映射关系的 hash（关联数组）。默认值是空数组，意思是关闭带权

加和。

当在不同的本地索引中都匹配到相同的文档 ID 时，Sphinx 默认选择查询中指定的最后一个索引。这是为了支持部分重叠的分区索引。

然而在某些情况下索引并不仅仅是被分区了，您可能想将不同索引中的权值加在一起，而不是简单地选择其中的一个。SetIndexWeights()允许您这么做。当开启了加和功能后，最后的匹配权值是各个索引中的权值的加权合，各索引的权由本调用指定。也就是说，如果文档 123 在索引 A 被找到，权值是 2，在 B 中也可找到，权值是 3，而且您调用了 `SetIndexWeights ( array ( "A"=>100, "B"=>10 ) )`，那么文档 123 最终返回给客户端的权值为 `2*100+3*10 = 230`。

5. 4. 结果集过滤设置

5.4.1. SetIDRange

原型: function SetIDRange ($min, $max)

设置接受的文档 ID 范围。参数必须是整数。默认是 0 和 0，意思是不限制范围。

此调用执行后，只有 ID 在$min 和$max（包括$min 和$max）之间的文档会被匹配。

5.4.2. SetFilter

原型: function SetFilter ($attribute, $values, $exclude=false)

增加整数值过滤器。

此调用在已有的过滤器列表中添加新的过滤器。$attribute 是属性名。$values 是整数数组。$exclude 是布尔值，它控制是接受匹配的文档（默认模式，即$exclude 为假时）还是拒绝它们。

只有当索引中$attribute 列的值与$values 中的任一值匹配时文档才会被匹配（或者拒绝，如果$exclude 值为真）

5.4.3. SetFilterRange

原型 : function SetFilterRange ($attribute, $min, $max, $exclude=false)

添加新的整数范围过滤器。

此调用在已有的过滤器列表中添加新的过滤器。`$attribute` 是属性名，`$min`、`$max` 定义了一个整数闭区间，`$exclude` 布尔值，它控制是接受匹配的文档（默认模式，即`$exclude` 为假时）还是拒绝它们。

只有索引中$attribute 列的值落在$min 和$max 之间（包括$min 和$max），文档才会被匹配（或者拒绝，如果$exclude 值为真）。

5.4.4. SetFilterFloatRange

原型: function SetFilterFloatRange ($attribute, $min, $max, $exclude=false)

增加新的浮点数范围过滤器。

此调用在已有的过滤器列表中添加新的过滤器。$attribute 是属性名，$min、$max 定义了一个浮点数闭区间，$exclude 必须是布尔值，它控制是接受匹配的文档（默认模式，即 $exclude 为假时）还是拒绝它们。

只有当索引中$attribute 列的值落在$min 和$max 之间（包括$min 和$max），文档才会被匹配（或者拒绝，如果$exclude 值为真）。

5.4.5. SetGeoAnchor

原型: function SetGeoAnchor ($attrlat, $attrlong, $lat, $long)

为地表距离计算设置锚点，并且允许使用它们。

$attrlat 和$attrlong 是字符串，分别指定了对应经度和纬度的属性名称。$lat 和$long 是浮点值，指定了锚点的经度和纬度值，以角度为单位。

一旦设置了锚点，您就可以在您的过滤器和/或排序表达式中使用@geodist 特殊属性。Sphinx 将在每一次全文检索中计算给定经纬度与锚点之前的地表距离，并把此距离附加到匹配结果上去。SetGeoAnchor 和索引属性数据中的经纬度值都是角度。而结果会以米为单位返回，因此地表距离 1000.0 代表 1 千米。一英里大约是 1609.344 米。

5.5. GROUP BY 设置

5.5.1. SetGroupBy

原型: function SetGroupBy ($attribute, $func, $groupsort="@group desc")

设置进行分组的属性、函数和组间排序模式，并启用分组（参考节 4.6, “结果分组（聚类）”中的描述）。

$attribute 是字符串，为进行分组的属性名。$func 为常数，它指定内建函数，该函数以前面所述的分组属性的值为输入，目前的可选的值为：

SPH_GROUPBY_DAY， SPH_GROUPBY_WEEK， SPH_GROUPBY_MONTH， SPH_GROUPBY_YEAR， SPH_GROUPBY_ATTR 。

$groupsort 是控制分组如何排序的子句。其语法与节 4.5, “SPH_SORT_EXTENDED 模式”中描述的相似。

分组与 SQL 中的 GROUP BY 子句本质上相同。此函数调用产生的结果与下面伪代码产生的结果相同。

```
SELECT ... GROUP BY $func($attribute) ORDER BY $groupsort
```

注意，影响最终结果集中匹配项顺序的是$groupsort。排序模式（见节 5.3.3,“SetSortMode”）影响每个分组内的顺序，即每组内哪些匹配项被视为最佳匹配。比如，组之间可以根据每组中的匹配项数量排序的同时每组组内又根据相关度排序。

5.5.2. SetGroupDistinct

原型: function SetGroupDistinct ($attribute)

设置分组中需要计算不同取值数目的属性名。只在分组查询中有效。

$attribute 是包含属性名的字符串。每个组的这个属性的取值都会被储存起来（只要内存允许），其后此属性在此组中不同值的总数会被计算出来并返回给客户端。这个特性与标准

SQL 中的 COUNT（DISTINCT）子句类似。因此如下 Sphinx 调用

```
$cl->SetGroupBy ( "category", SPH_GROUPBY_ATTR, "@count desc" );
$cl->SetGroupDistinct ( "vendor" );
```

等价于如下的 SQL 语句：

```
SELECT id, weight, all-attributes,
       COUNT(DISTINCT vendor) AS @distinct,
       COUNT(*) AS @count
FROM products
GROUP BY category
ORDER BY @count DESC
```

在上述示例伪代码中，SetGroupDistinct()调用只与 COUNT（DISTINCT vendor）对应。GROUP BY，ORDER By 和 COUNT(*)子句则与 SetGroupBY()调用等价。两个查询都会在每类中返回一个匹配的行。除了索引中的属性，匹配项还可以包含每类的匹配项计数和每类中不同 vendor ID 的计数。

5.6. Querying

5.6.1. Query

原型: function Query ($query, $index="*")

连接到 searchd 服务器，根据服务器的当前设置执行给定的查询，取得并返回结果集。

$query 是查询字串，$index 是包含一个或多个索引名的字符串。一旦发生一般错误，则返回假并设置 GetLastError()信息。若成功则返回搜索的结果集。

$index 的默认值是“*”，意思是对全部本地索引做查询。索引名中允许的字符包括拉丁字母（a-z），数字（0-9），减号（-）和下划线（_），其他字符均视为分隔符。因此，下面的示例调用都是有效的，而且会搜索相同的两个索引。

```
$cl->Query ( "test query", "main delta" );
$cl->Query ( "test query", "main;delta" );
$cl->Query ( "test query", "main, delta" );
```

给出多个索引时的顺序是有意义的。如果同一个文档 ID 的文档在多个索引中找到，那么权值和属性值会取最后一个索引中所存储的作为该文档 ID 的权值和属性值，用于排序、过滤，并返回给客户端（除非用 SetIndexWeights()显式改变默认行为）。因此在上述示例中，索引“delta”中的匹配项总是比索引“main”中的更优先。

如果搜索成功，Query()返回的结果集包含找到的全部匹配项中的一部分（根据 SetLimits()之设定）和与查询相关的统计数据。结果集是 hash（仅 PHP，其他语言的 API 可能使用其他数据结构）

，包含如下键和值：

"matches":

是一个 hash 表，存储文档 ID 以及其对应的另一个包含文档权重和属性值的 hash 表（或者是数组，如果启用了 SetArrayResult()）。

"total":

此查询在**服务器**检索所得的匹配文档总数（即服务器端结果集的大小）。这是在当前设置下，用当前查询可以从服务器端获得的匹配文档数目的上限。

"total_found":
（服务器上找到和处理了的）索引中匹配文档的总数。

"words":
一个 hash，它将查询关键字（关键字已经过大小写转换，取词干和其他处理）映射到一个包含关于关键字的统计数据（“docs”——在多少文档中出现，“hits”——共出现了多少次）的小 hash 表上。

"error":
searchd 报告的错误信息（人类可读的字符串）。若无错误则为空字符串。

"warning":
searchd 报告的警告信息（人类可读字符串）。若无警告则为空串。

5.6.2. AddQuery

原型: function AddQuery ($query, $index="*")

向批量查询增加一个查询。$query 为查询串。$index 为包含一个或多个索引名的字符串。返回 RunQueries()返回的数组中的一个下标。

批量查询（或多查询）使 searchd 能够进行可能的内部优化，并且无论在任何情况下都会减少网络连接和进程创建方面的开销。相对于单独的查询，批量查询不会引入任何额外的开销。因此当您的 Web 页运行几个不同的查询时，一定要考虑使用批量查询。

例如，多次运行同一个全文查询，但使用不同的排序或分组设置，这会使 searchd 仅运行一次开销昂贵的全文检索和相关度计算，然后在此基础上产生多个分组结果。

有时您不仅需要简单地显示搜索结果，而且要显示一些与类别相关的计数信息，例如按制造商分组后的产品数目，此时批量查询会节约大量的开销。若无批量查询，您会必须将这些本质上几乎相同的查询运行多次并取回相同的匹配项，最后产生不同的结果集。若使用批量查询，您只须将这些查询简单地组成一个批量查询，Sphinx 会在内部优化掉这些冗余的全文搜索。

AddQuery()在内部存储全部当前设置状态以及查询，您也可在后续的 SubQuery()调用中改变设置。早先加入的查询不会被影响，实际上没有任何办法可以改变它们。下面是一个示例：

```
$cl->SetSortMode ( SPH_SORT_RELEVANCE );
$cl->AddQuery ( "hello world", "documents" );

$cl->SetSortMode ( SPH_SORT_ATTR_DESC, "price" );
$cl->AddQuery ( "ipod", "products" );

$cl->AddQuery ( "harry potter", "books" );

$results = $cl->RunQueries ();
```

用上述代码，第一个查询会在“documents”索引上查询“hello world”并将结果按相关度排序，第二个查询会在“products”索引上查询“ipod”并将结果按价格排序，第三个查询在“books”索引上搜索“harry potter”，结果仍按价格排序。注意，第二个 SetSortMode()调用并不会影响第一个查询（因为它已经被添加了），但后面的两个查询都会受影响。

AddQuery()并不修改当前状态。也就是说，现有的全部排序、过滤和分组设置都不会因这个

调用而发生改变，因此后续的查询很容易地复用现有设置。

AddQuery()返回 RunQueries()返回的数组中的一个下标。它是一个从 0 开始的递增整数，即，第一次调用返回 0，第二次返回 1，以此类推。这个方便的特性使你在需要这些下标的时候不用手工记录它们。

5.6.3. RunQueries

原型: function RunQueries ()

连接到 searchd，运行由 AddQuery()添加的全部查询，获取并返回它们的结果集。若发生一般错误（例如网络 I/O 失败）则返回假并设置 GetLastError()信息。若成功则返回结果集的简单数组。

该数组中的每一个结果集都跟 `Query()` 返回的结果集完全相同。注意，批量查询请求自身几乎总是成功——除非有网络错误、正在进行索引轮换，或者其他导致整个查询无法被处理的因素。

然而其中的单个的查询很可能失败。此时与之对应的结果集只包含一个非空的“错误”信息，而没有关于匹配或查询的统计信息。在极端情况下，批量查询中的所有单个查询可能都失败。但这仍然不会导致报告一般错误，因为 API 已经成功地连接到 searchd，提交了批量查询并得到返回结果——但每个结果集都只包含特定的错误信息。

5.6.4. ResetFilters

原型: function ResetFilters ()

清除当前设置的过滤器。

通常此调用在使用批量查询的时候会用到。您可能需要为批量查询中的不同查询提供不同的过滤器，为达到这个目的，您需要调用 ResetFilters()然后用其他调用增加新的过滤器。

5.6.5. ResetGroupBy

原型: function ResetGroupBy ()

清除现有的全部分组设置，并关闭分组。

通常此调用在使用批量查询的时候会用到。单独的分组设置可以用 SetGroupBy()和 SetGroupDistinct()来改变，但它们不能关闭分组。ResetGroupBy()将之前的分组设置彻底重置并在当前状态下关闭分组模式，因此后续的 AddQuery()可以进行无分组的搜索。

5.7. 额外的方法

5.7.1. BuildExcerpts

原型: function BuildExcerpts ($docs, $index, $words, $opts=array())

该函数用来产生文档片段（摘要）。连接到 searchd，要求它从指定文档中产生片段（摘要），并返回结果。

$docs 为包含各文档内容的数组。$index 为包含索引名字的字符串。给定索引的不同设置（例如字符集、形态学、词形等方面的设置）会被使用。$words 为包含需要高亮的关键字的字符串。它们会按索引的设置被处理。例如，如果英语取词干（stemming）在索引中被设置

为允许，那么即使关键词是“shoe”，“shoes”这个词也会被高亮。$opts 为包含其他可选的高亮参数的 hash：

"before_match":
在匹配的关键字前面插入的字符串。默认为“<b>”

"chunk_separator":
在摘要块（段落）之间插入的字符串。默认为“...”

"limit":
摘要最多包含的符号（码点）数。整数，默认为 256.

"around":
每个关键词块左右选取的词的数目。整数，默认为 5.

"exact_phrase":
是否仅高亮精确匹配的整个查询词组，而不是单独的关键词。布尔值，默认为假。

"single_passage":
是否仅抽取最佳的一个段落。布尔值，默认为否。

失败时返回假。成功时返回包含有片段（摘要）字符串的数组。

5.7.2. UpdateAttributes

原型: function UpdateAttributes ($index, $attrs, $values)

立即更新指定文档的指定属性值。成功则返回实际被更新的文档数目（0 或更多），失败则返回-1。

$index 为待更新的（一个或多个）索引名。$attrs 为属性名字符串的数组，其所列的属性会被更新。$values 为 hash 表，$values 表的键为文档 ID，$values 表的值为新的属性值的简单数组。

$index 既可以是一个单独的索引名，也可以是一个索引名的列表，就像 Query()的参数。与 Query()不同的是不允许通配符，全部待更新的索引必须明确指出。索引名列表可以包含分布式索引。对分布式索引，更新会同步到全部代理上。

只有在 docinfo=extern 这个存储策略下才可以运行更新。更新非常快，因为操作完全在内存中进行，但它们也可以变成持久的，更新会在 searchd 干净关闭时（收到 SIGTERM 信号时）被写入磁盘。

使用示例：

```
$cl->UpdateAttributes ( "test1", array("group_id"), array(1=>array(456)) );
$cl->UpdateAttributes ( "products", array ( "price", "amount_in_stock" ),
      array ( 1001=>array(123,5), 1002=>array(37,11), 1003=>(25,129) ) );
```

第一条示例语句会更新索引“test1”中的文档 1，设置“group_id”为 456.第二条示例语句则更新索引“products”中的文档 1001，1002 和 1003。文档 1001 的“price”会被更新为 123，“amount_in_stock”会被更新为 5；文档 1002，“price”变为 37 而“amount_in_storage”变为 11，等等。

6. MySQL storage engine (SphinxSE)

6. MySQL 存储引擎 (SphinxSE)

6.1. SphinxSE 概览

SphinxSE 是一个可以编译进 MySQL 5.x 版本的 MySQL 存储引擎，它利用了该版本 MySQL 的插件式体系结构。SphinxSE 不能用于 MySQL 4.x 系列，它需要 MySQL 5.0.22 或更高版本；或 MySQL 5.1.12 或更高版本。

尽管被称作“存储引擎”，SphinxSE 自身其实并不存储任何数据。它其实是一个允许 MySQL 服务器与 searchd 交互并获取搜索结果的嵌入式客户端。所有的索引和搜索都发生在 MySQL 之外。

显然，SphinxSE 的适用于：

- 使将 MySQL FTS 应用程序移植到 Sphinx
- 使没有 Sphinx API 的那些语言也可以使用 Sphinx
- 当需要在 MySQL 端对 Sphinx 结果集做额外处理（例如对原始文档表做 JOIN，MySQL 端的额外过滤等等）时提供优化

6.2.安装 SphinxSE

您需要搞到一份 MySQL 的源码，并重新编译 MySQL。MySQL 源码（mysql-5.x.yy.tar.gz）可在 dev.mysql.com 网站获得。

针对某些版本的 MySQL，Sphinx 网站提供了包含支持 SphinxSE 的打过补丁 tarball 压缩包。将这些文件解压出来替换原始文件，就可以配置(configure)、构建(build)以生成带有内建 Shpinx 支持的 MySQL 了。

如果网站上没有对应版本的 tarball，或者由于某种原因无法工作，那您可能需要手工准备这些文件。您需要一份安装好的 GUN Autotools 框架（autoconf，automake 和 libtool）来完成这项任务。

6.2.1. 在 MySQL 5.0.x 上编译 SphinxSE

如果使用我们事先做好的打过补丁的 tarball，那请跳过步骤 1-3

1. 将 sphinx5.0.yy.diff 补丁文件复制到 MySQL 源码目录并运行

```
patch -p1 < sphinx.5.0.yy.diff
```

   如果没有与您的 MySQL 版本完全匹配的.diff 文件，请尝试一个最接近版本的.diff 文件。确保补丁顺利应用，没有 rejects。

2. 在 MySQL 源码目录中运行

```
sh BUILD/autorun.sh
```

3. 在 MySQL 源码目录中建立 sql/sphinx 目录，并把 Sphinx 源码目录中 mysqlse 目录下的全部文件拷贝到这个目录。

```
cp
-R /root/builds/sphinx-0.9.7/mysqlse /root/builds/mysql-5.0.24/sql/sphinx
```

4. 配置（configure）MySQL，启用 Sphinx 引擎

```
./configure --with-sphinx-storage-engine
```

5. 构建（build）并安装 MySQL

```
make
make install
```

6.2.2. 在 MySQL 5.1.x 上编译 SphinxSE

如果使用我们事先做好的打过补丁的 tarball，那请跳过步骤 1-3

1. 在 MySQL 源码目录中建立 storage/sphinx 目录，并将 Sphinx 源码目录中的 mysqlse 目录下的全部文件拷贝到这个目录。

```
cp
-R /root/builds/sphinx-0.9.7/mysqlse /root/builds/mysql-5.1.14/storage/sph
inx
```

2. 在 MySQL 源码目录运行

```
sh BUILD/autorun.sh
```

3. 配置（configure）MySQL，启用 Sphinx 引擎

```
./configure --with-plugins=sphinx
```

4. 构建（build）并安装 MySQL

```
make
make install
```

6.2.3. SphinxSE 安装测试

为了检查 SphinxSE 是否成功地编入了 MySQL，启动新编译出的 MySQL 服务器，运行 mysql 客户端，执行 SHOW ENGLINES 查询，这会显示一个全部可用引擎的列表。Sphinx 应该出现在这个列表中，而且在“Support”列上显示“YES”

```
mysql> show engines;
+------------+----------
+-------------------------------------------------------------------+
| Engine     | Support  | Comment
|
+------------+----------
+-------------------------------------------------------------------+
| MyISAM     | DEFAULT  | Default engine as of MySQL 3.23 with great performance
|
  ...
| SPHINX     | YES      | Sphinx storage engine
|
  ...
+------------+----------
+-------------------------------------------------------------------+
13 rows in set (0.00 sec)
```

6.3. 使用 SphinxSE

要通过 SphinxSE 搜索，您需要建立特殊的 ENGINE=SPHINX 的“搜索表”，然后使用 SELECT 语句从中检索，把全文查询放在 WHERE 子句中。

让我们从一个 create 语句和搜索查询的例子开始：

```
CREATE TABLE t1
(
   id          INTEGER NOT NULL,
   weight      INTEGER NOT NULL,
   query       VARCHAR(3072) NOT NULL,
   group_id    INTEGER,
   INDEX(query)
) ENGINE=SPHINX CONNECTION="sphinx://localhost:3312/test";

SELECT * FROM t1 WHERE query='test it;mode=any';
```

搜索表前三列的类型**必须**是 INTEGER，INTEGER 和 VARCHAR，这三列分别对应文档 ID，匹配权值和搜索查询。查询列必须被索引，其他列必须无索引。列的名字会被忽略，所以可以任意命名。

额外的列的类型必须是 INTEGER 或 TIMESPTAMP 之一。它们必须与 Sphinx 结果集中提供的属性按名称绑定，即它们的名字必须与 sphinx.conf 中指定的属性名一一对应。如果 Sphinx 搜索结果中没有某个属性名，该列的值就为 NULL.

特殊的“虚拟”属性名也可以与 SphinxSE 列绑定。但特殊符号@用_sph_代替。例如，要取得@group 和@count 虚属性，列名应使用_sph_group 和_sph_count。

可以使用字符串参数 CONNECTION 来指定用这个表搜索时的默认搜索主机、端口号和索引。如果 CREATE TABLE 中没有使用连接（connection）串，那么默认使用索引名“*”（搜索所有索引）和 localhost:3312。连接串的语法如下：

```
CONNECTION="sphinx://HOST:PORT/INDEXNAME"
```

默认的连接串也可以日后改变：

```
ALTER TABLE t1 CONNECTION="sphinx://NEWHOST:NEWPORT/NEWINDEXNAME";
```

也可以在查询中覆盖全部这些选项。

如例子所示，查询文本和搜索选项都应放在 WHERE 子句中对 query 列的限制中（即第三列），选项之间用分号分隔，选项名与选项值用等号隔开。可以指定任意数目的选项。可用的选项如下：

- query —— 查询文本
- mode —— 匹配模式.必须是“all”， “any”， “phrase”， “boolean” 或 “extended 之一，默认为“All”
- sort —— 匹配项排序模式 必须是“relevance”， “attr_desc”， “attr_asc”， “time_segments”或“extended”之一。除了“relevance”模式，其他模式中还必须在一个冒号后附上属性名（或“extended”模式中的排序子句）。

  ```
  ... WHERE query='test;sort=attr_asc:group_id';
  ... WHERE query='test;sort=extended:@weight desc, group_id asc';
  ```

- offset —— 结果集中的偏移量，默认是 0。

- limit —— 从结果集中获取的匹配项数目，默认为 20。
- Index —— 待搜索的索引：

```
... WHERE query='test;index=test1;';
... WHERE query='test;index=test1,test2,test3;';
```

- minid, maxid —— 匹配文档 ID 的最小值和最大值
- weights —— 逗号分隔的列表，指定 Sphinx 全文数据字段的权值

```
... WHERE query='test;weights=1,2,3;';
```

- filter, !filter —— 逗号分隔的列表，指定一个属性名和一系列可匹配的属性值：

```
# only include groups 1, 5 and 19
... WHERE query='test;filter=group_id,1,5,19;';

# exclude groups 3 and 11
... WHERE query='test;!filter=group_id,3,11;';
```

- range, !range —— 逗号分隔的列表，指定一个属性名和该属性可匹配的最小值和最大值：

```
# include groups from 3 to 7, inclusive
... WHERE query='test;range=group_id,3,7;';

# exclude groups from 5 to 25
... WHERE query='test;!range=group_id,5,25;';
```

- maxmatches —— 此查询最大匹配的数量：

```
... WHERE query='test;maxmatches=2000;';
```

- groupby —— 分组（group-by）函数和属性：

```
... WHERE query='test;groupby=day:published_ts;';
... WHERE query='test;groupby=attr:group_id;';
```

- groupsort —— 分组（group-by）排序子句

```
... WHERE query='test;groupsort=@count desc;';
```

- indexweights —— 逗号分隔的列表，指定一系列索引名和搜索时这些索引对应的权值

```
... WHERE query='test;indexweights=idx_exact,2,idx_stemmed,1;';
```

非常重要的注意事项：让 Sphinx 来对结果集执行排序、过滤和切片（slice）要比提高最大匹配项数量然后在 MySQL 端用 WHERE、ORDER BY 和 LIMIT 子句完成对应的功能来得高效**得多**。这有两方面的原因。首先，Sphinx 对这些操作做了一些优化，比 MySQL 效率更高一些。其次，searchd 可以打包更少的数据，SphinxSE 也可以传输和解包更少的数据。

除了结果集，额外的查询信息可以用 SHOW ENGINE SPHINX STATUS 语句获得：

```
mysql> SHOW ENGINE SPHINX STATUS;
+--------+-------+----------------------------------------------------+
| Type   | Name  | Status                                             |
+--------+-------+----------------------------------------------------+
| SPHINX | stats | total: 25, total found: 25, time: 126, words: 2 |
| SPHINX | words | sphinx:591:1256 soft:11076:15945                |
+--------+-------+----------------------------------------------------+
2 rows in set (0.00 sec)
```

可以对 SphinxSE 搜索表和其他引擎的表之间使用 JOIN，以下是一个例子，例中“documents”来自 example.sql：

```
mysql> SELECT content, date_added FROM test.documents docs
-> JOIN t1 ON (docs.id=t1.id)
-> WHERE query="one document;mode=any";
+------------------------------------+---------------------+
| content                            | docdate             |
+------------------------------------+---------------------+
| this is my test document number two | 2006-06-17 14:04:28 |
| this is my test document number one | 2006-06-17 14:04:28 |
+------------------------------------+---------------------+
2 rows in set (0.00 sec)

mysql> SHOW ENGINE SPHINX STATUS;
+--------+-------+----------------------------------------------+
| Type   | Name  | Status                                       |
+--------+-------+----------------------------------------------+
| SPHINX | stats | total: 2, total found: 2, time: 0, words: 2 |
| SPHINX | words | one:1:2 document:2:2                         |
+--------+-------+----------------------------------------------+
2 rows in set (0.00 sec)
```

7. 报告 bugs

很不幸，Sphinx 还没有达到 100%无 bug（尽管我们正向这个目标努力），因此您可能偶尔遇到些问题。

对于每个问题的报告越详细越好，这很重要——因为要想修复 bug，我们必须重现 bug 并调试它，或者根据您提供的信息来推断出产生 bug 的原因。因此在此提供一些如何报告 bug 的指导。

构建时问题

如果 Sphinx 构建失败，请您：

1. 确认您的 DBMS 的头文件和库文件都正确安装了（例如，检查 mysql-devel 包已经安装）
2. 报告 Sphinx 的版本和配置文件（别忘了删除数据库连接密码），MySQL（或 PostgreSQL）配置文件信息，gcc 版本，操作系统版本和 CPU 类型（例如 x86、x86-64、PowerPC 等）

```
mysql_config
gcc --version
uname -a
```

3. 报告 configure 脚本或者 gcc 给出的错误信息（只需错误信息本身，不必附上整个构建日志）

运行时问题

如果 Sphinx 已经成功构建并能运行，但运行过程中出现了问题，请您：

1. 描述您遇到的 bug（即，您预期的行为和实际发生的行为），以及重现您遇到的问题需要的步骤。
2. 附带 Sphinx 的版本和配置文件（别忘了删除密码），MySQL（或 PostgreSQL）配置文件信息，gcc 版本，操作系统版本和 CPU 类型（例如 x86、x86-64、PowerPC 等）

```
mysql --version
gcc --version
uname -a
```

3. 构建、安装和运行调试版本的全部 Sphinx 程序（这会启用很多内部校验，或称断言）

```
make distclean
./configure --with-debug
make install
killall -TERM searchd
```

4. 重新索引，检查是否有断言被触发（如果是，那很可能是索引损坏了并造成了问题）
5. 如果 bug 在调试版本中没有重现，请回到非调试版本并在报告中说明这个情况。
6. 如果 bug 可以在您的数据库的很小的子集（1-100 条记录）上重现，请提供一个该子集的 gzip 压缩包。
7. 如果问题与 searchd 有关，请在 bug 报告中提供 searchd.log 和 query.log 中的相关条目。
8. 如果问题与 searchd 有关，请尝试在 console 模式下运行它并检查它是否因断言失败而退出。

```
./searchd --console
```

9. 如果任何一个程序因断言失败而退出，请提供断言信息。

调试断言，崩溃和停止响应

如果任何一个程序因断言失败而退出，崩溃或停止响应，您可以额外生成一个内存转储文件并检查它。

1. 启用内存转储。在大多数 Linux 系统上，可以用 ulimit 命令启用它。

```
ulimit -c 32768
```

2. 运行程序，尝试重现 bug。
3. 如果程序崩溃（可能有断言失败的情况也可能没有），在当前目录下找到内存转储文件（一般会打印“Segmentation fault (core dumped)”消息）
4. 如果程序停止响应，在另一个控制台上用 kill -SEGV 强制退出并获得内存转储：

```
kill -SEGV HANGED-PROCESS-ID
```

5. 使用 gdb 检查转储文件，查看 backtrace

```
gdb ./CRASHED-PROGRAM-FILE-NAME CORE-DUMP-FILE-NAME
(gdb) bt
(gdb) quit
```

提示： HANGED-PROCESS-ID（停止响应的进程 ID）, CRASHED-PROGRAM-FILE-NAME（崩溃程序的文件名） and CORE-DUMP-FILE-NAME（核心转储文件的文件名）应

该被换成具体的数字和文件名。例如，一次对停止响应的 searchd 的调试会话看起来应该像下面这样：

```
# kill -SEGV 12345
# ls *core*
core.12345
# gdb ./searchd core.12345
(gdb) bt
...
(gdb) quit
```

注意 ulimit 并不是整个服务器范围的，而是仅影响当前的 shell 会话。因此您不必还原任何服务器范围的限制——但是一旦重新登陆，您就需要再次设置 ulimit。

核心内存转储文件会存放在当前工作目录下（Sphinx 的各个程序不会改变工作目录），因此它们就在那。

不要立刻删除转储文件，从它里面可能获得更多有用的信息。您不需要把这个文件发送给我们（因为调试信息与您的系统本身紧密相关），但我们可能会向您询问一些与之相关的问题。

8. sphinx.conf 选项参考

8. 1. 数据源配置选项

8.1.1. type

数据源类型。必须选项，无默认值。已知的类型包括 mysql，pgsql，xmlpipe 和 xmlpipe2。

所有其他与数据源相关的选项都依赖于这个选项指定的源类型。与 SQL 数据源（即 MySQL 和 PostgreSQL）相关的选项以“sql_”开头，而与 xmlpipe 和 xmlpipe2 数据源相关的选项则以“xmlpipe_”开头。

示例：

```
type = mysql
```

8.1.2. sql_host

要连接的 SQL 服务器主机地址。必须选项，无默认值。仅对 SQL 数据源（mysql 和 pgsql）有效。

最简单的情形下，Sphinx 与 MySQL 或 PostgreSQL 服务器安装在同一台主机上，此时您只须设置为 localhost 即可。注意，MySQL 客户端库根据主机名决定是通过 TCP/IP 还是 UNIX socket 连接到服务器。一般来说，“localhost”使之强制使用 UNIX socket 连接（这是默认的也是推荐的模式），而“127.0.01”会强制使用 TCP/IP。细节请参考 MySQL 文档。

示例：

```
sql_host = localhost
```

8.1.3. sql_port

要连接的 SQL 服务器的 IP 端口。可选选项，默认值为 mysql 端口 3306，pgsql 端口 5432。

仅适用于 SQL 数据源（mysql 和 pgsql）。注意，此选项是否实际被使用依赖于 sql_host 选项。

示例:

```
sql_port = 3306
```

8.1.4. sql_user

连接到 sql_host 时使用的 SQL 用户名。必须选项，无默认值。仅适用于 SQL 数据源（mysql 和 pgsql）。

示例:

```
sql_user = test
```

8.1.5. sql_pass

连接到 sql_host 时使用的 SQL 用户密码。必须选项，无默认值。仅适用于 SQL 数据源（mysql 和 pgsql）。

示例:

```
sql_pass = mysecretpassword
```

8.1.6. sql_db

连接到 SQL 数据源之后使用的 SQL 数据库，此后的查询均在此数据库上进行。必须选项，无默认值。仅适用于 SQL 数据源（mysql 和 pgsql）。

示例:

```
sql_db = test
```

8.1.7. sql_sock

连接到本地 SQL 服务器时使用的 UNIX socket 名称。可选选项，默认值为空（使用客户端库的默认设置）。仅适用于 SQL 数据源（mysql 和 pgsql）。

在 Linux 上，通常是`/var/lib/mysql/mysql.sock`。 而在 FreeBSD 上通常是`/tmp/mysql.sock`。注意此选项是否实际被使用依赖与 `sql_host` 的设置。

示例:

```
sql_sock = /tmp/mysql.sock
```

8.1.8. mysql_connect_flags

MySQL 客户端的连接标志（connection flags）。可选选项，默认值为 0（不设置任何标志）。仅适用于 SQL 数据源（mysql 和 pgsql）。

此选项必须包含各标志相加所得的整型值。此整数将被原样传递给 mysql_real_connect() 。可用的标志在 mysql_com.h 中列举。下面列举的是几个与索引相关的标志和它们的值：

- CLIENT_COMPRESS = 32; 允许使用压缩协议
- CLIENT_SSL = 2048; 握手后切换到 SSL
- CLIENT_SECURE_CONNECTION = 32768; 新的 4.1 版本身份认证

例如，标志 2080（2048+32）代表同时使用压缩和 SSL，32768 代表仅使用新的身份验证。起初这个选项是为了在 indexer 和 mysql 位于不同主机的情况下使用压缩协议而引入的。尽管降低了网络带宽消耗，但不管在理论上还是在现实中，在 1Gbps 的链路上启用压缩很可能恶化索引时间。然而在 100Mbps 的连输上启用压缩可能会明显地改善索引时间（有报告说总的索引时间降低了 20-30%）。根据您网络的连接情况，您获得的改善程度可能会有所不同。

示例：

```
mysql_connect_flags = 32 # enable compression
```

8.1.9. sql_query_pre

取前查询（pre-fetch query），或预查询（pre-query）。多值选项，可选选项，默认为一个空的查询列表。仅适用于 SQL 数据源（mysql 和 pgsql）。

多值意思是您可以指定多个预查询。它们在主查询之前执行，而且会严格按照在配置文件中出现的顺序执行。预查询的结果会被忽略。

预查询在很多时候有用。它们被用来设置字符编码，标记待索引的记录，更新内部计数器，设置 SQL 服务器连接选项和变量等等。

也许预查询最常用的一个应用就是用来指定服务器返回行时使用的字符编码。这**必须**与 Sphinx 期待的编码相同（在 charset_type 和 charset_table 选项中设置）。以下是两个与 MySQL 有关的设置示例：

```
sql_query_pre = SET CHARACTER_SET_RESULTS=cp1251
sql_query_pre = SET NAMES utf8
```

对于 MySQL 数据源，在预查询中禁用查询缓冲（query cache）（仅对 indexer 连接）是有用的，因为索引查询一般并会频繁地重新运行，缓冲它们的结果是没有意义的。这可以按如下方法实现：

```
sql_query_pre = SET SESSION query_cache_type=OFF
```

示例：

```
sql_query_pre = SET NAMES=utf8
sql_query_pre = SET SESSION query_cache_type=OFF
```

8.1.10. sql_query

获取文档的主查询。必须的选项，无默认选项。仅适用于 SQL 数据源（mysql 和 pgsql）。

只能有一个主查询。它被用来从 SQL 服务器获取文档（文档列表）。可以指定多达 32 个全文数据字段（严格来说是在 sphinx.h 中定义的 SPH_MAX_FIELDS 个）和任意多个属性。所有既不是文档 ID（第一列）也不是属性的列的数据会被用于建立全文索引。

文档 ID **必须**是第一列，而且**必须是唯一的正整数值（不能是 0 也不能是负数）**，既可以是 32 位的也可以是 64 位的，这要根据 Sphinx 是如何被构建的，默认情况下文档 ID 是 32 位的，

但在运行 configure 脚本时指定--enable-id64 选项会打开 64 位文档 ID 和词 ID 的支持。

示例：

```
sql_query = \
      SELECT id, group_id, UNIX_TIMESTAMP(date_added) AS date_added, \
            title, content \
      FROM documents
```

8.1.11. sql_query_range

分区查询设置。可选选项，默认为空。仅适用于 SQL 数据源（mysql 和 pgsql）。

设置这个选项会启用文档的分区查询（参看节 3.7, “分区查询”）。分区查询有助于避免在索引大量数据时发生 MyISAM 表臭名昭著的死锁问题。（同样有助于解决其他不那么声名狼藉的问题，比如大数据集上的性能下降问题，或者 InnoDB 对多个大型读事务（read transactions）进行序列化时消耗额外资源的问题。）

此选项指定的查询语句必须获取用于分区的最小和最大文档 ID。它必须返回正好两个整数字段，先是最小 ID 然后是最大 ID，字段的名字会被忽略。

当启用了分区查询时，sql_query 要求包括$start 和$end 宏（因为重复多次索引整个表显然是个错误）。注意，$start..$end 所指定的区间不会重叠，因此不会在查询中删除 ID 正好等于 $start 或$end 的文档。节 3.7, “分区查询”中的例子解释了这个问题，注意大于等于或小于等于比较操作是如何被使用的。

示例：

```
sql_query_range = SELECT MIN(id),MAX(id) FROM documents
```

8.1.12. sql_range_step

分区查询的步进。可选选项，默认为 1024。仅适用于 SQL 数据源（mysql 和 pgsql）。

仅当启用 ranged queries 时有效。用 sql_query_range 取得的文档 ID 区间会被以这个不小的间隔步数跳跃遍历。例如，如果取得的最小和最大 ID 分别是 12 和 3456，而间隔步数是 1000，那么 indexer 会用下面这些值重复调用几次 sql_query()

- $start=12, $end=1011
- $start=1012, $end=2011
- $start=2012, $end=3011
- $start=3012, $end=3456

示例：

```
sql_range_step = 1000
```

8.1.13. sql_attr_uint

声明无符号整数属性(attribute)。可声明同一类型的多个不同名称的属性，可选项。仅适用于 SQL 数据源（mysql 和 pgsql）。

被声明的列的值必须在 32 位无符号整型可表示的范围内。超出此范围的值也会被接受，但

会溢出。例如-1 会变成 2^32-1 或者说 4,294,967,295。

您可以在属性名后面附加“:BITCOUNT”（见下面的示例）以便指定整型属性的位数。属性小于默认 32 位（此时称为位域）会有损性能。但它们在外部存储（extern storage）模式下可以节约内存：这些位域被组合成 32 位的块存储在.spa 属性数据文件中。如果使用内联存储（inline storage），则位宽度的设置会被忽略。

示例：

```
sql_attr_uint = group_id
sql_attr_uint = forum_id:9 # 9 bits for forum_id
```

8.1.14. sql_attr_bool

声明布尔属性（attribute）。可声明同一类型的多个不同名称的属性，可选项。仅适用于 SQL 数据源（mysql 和 pgsql）。等价于用 sql_attr_uint 声明为 1 位。

示例：

```
sql_attr_bool = is_deleted # will be packed to 1 bit
```

8.1.15. sql_attr_timestamp

声明 UNIX 时间戳属性（attribute）。可声明同一类型的多个不同名称的属性，可选项。仅适用于 SQL 数据源（mysql 和 pgsql）。

这个列的值必须是 UNIX 格式的时间戳，即 32 位无符号整数表示的自格林尼治平时 1970 年 1 月 1 日午夜起过去的秒数。时间戳在内部是按整数值存储和处理的。但除了将时间戳按整数使用，还可以对它们使用多种与日期相关的函数——比如时间段排序模式，或为分组（GROUP BY）抽取天/星期/月/年。注意 MySQL 中的 DATE 和 DATETIME 列类型**不能**直接作为时间戳使用，必须使用 UNIX_TIMESTAMP 函数将这些列做显式转换。

示例：

```
sql_attr_timestamp = UNIX_TIMESTAMP(added_datetime) AS added_ts
```

8.1.16. sql_attr_str2ordinal

声明字符串序数属性（attribute）。可声明同一类型的多个不同名称的属性，可选项。仅适用于 SQL 数据源（mysql 和 pgsql）。

这个属性类型（简称为字串序数）的设计是为了允许按字符串值排序，但不存储字符串本身。对字串序数做索引时，字符串值从数据库中取出、暂存、排序然后用它们在该有序数组中的序数代替它们自身，因此字串序数是个整型，对它们的大小比较与在原字串上做字典序比较结果相同。

早期版本上，对字串序数做索引可能消耗大量的 RAM。自 r1112 起，字串序数的积累和排序也可在固定大小的内存中解决了（代价是额外的临时磁盘空间），并受 mem_limit 设置限制。

理想中字符串可以根据字符编码和本地选项（locale）排序。例如，如果已知字符串为 KOI8R 编码下的俄语字串，那么对字节 0xE0,0xE1 和 0xE2 排序结果应为 0xE1,0xE2 和 0xE0，因为 0xE0 在 KOI8R 中代表的字符明显应在 0xE1 和 0xE2 之后。但很不幸，Sphinx 目

前不支持这个功能，而是简单地按字节值大小排序。

示例：

```
sql_attr_str2ordinal = author_name
```

8.1.17. sql_attr_float

声明浮点型属性（attribute）。可声明同一类型的多个不同名称的属性，可选项。仅适用于 SQL 数据源（mysql 和 pgsql）。

属性值按单精度 32 位 IEEE754 格式存储。可表示的范围大约是 1e-38 到 1e+38。可精确表示的小数大约是 7 位。浮点属性的一个重要应用是存储经度和纬度值（以角度为单位），经纬度值在查询时的地表距离计算中有用。

示例：

```
sql_attr_float = lat_radians
sql_attr_float = long_radians
```

8.1.18. sql_attr_multi

声明多值属性（MVA，Multi-valued attribute）。可声明同一类型的多个不同名称的属性，可选项。仅适用于 SQL 数据源（mysql 和 pgsql）。

简单属性每篇文档只允许一个值。然而有很多情况（比如 tags 或者类别）需要将多个值附加给同一个属性，而且要对这个属性值列表做过滤或者分组。

声明格式如下（用反斜线只是为了清晰，您仍可以在一行之内完成声明）：

```
sql_attr_multi = ATTR-TYPE ATTR-NAME 'from' SOURCE-TYPE \
       [;QUERY] \
       [;RANGE-QUERY]
```

其中

- ATTR-TYPE 是 'uint' 或 'timestamp'之一
- SOURCE-TYPE 是 'field', 'query', 或 'ranged-query'之一
- QUERY 是用来取得全部（文档 ID，属性值）序对的 SQL 查询
- RANGE-QUERY 是用来取得文档 ID 的最小值与最大值的 SQL 查询，与'sql_query_range'类似

示例：

```
sql_attr_multi = uint tag from query; SELECT id, tag FROM tags
sql_attr_multi = uint tag from ranged-query; \
       SELECT id, tag FROM tags WHERE id>=$start AND id<=$end; \
       SELECT MIN(id), MAX(id) FROM tags
```

8.1.19. sql_query_post

取后查询。可选项，默认值为空。仅适用于 SQL 数据源（mysql 和 pgsql）。

此查询在 sql_query 成功执行后立即执行。如果取后查询产生了错误，该错误被当作警告被

报告，但索引**不会**因此终止。取后查询的结果会被忽略。注意当取后查询执行时索引还**尚未**完成，而后面的索引仍然可能失败。因此在这个查询中不应进行任何永久性的更新。例如，不应在此查询中更新辅助表中存储的最近成功索引的文档 ID 值，请在后索引查询（post-index query）中操作。

示例：

```
sql_query_post = DROP TABLE my_tmp_table
```

8.1.20. sql_query_post_index

后索引查询。可选项，默认值为空。仅适用于 SQL 数据源（mysql 和 pgsql）。

此查询在索引完全成功结束后执行。如果此查询产生错误，该错误会被当作警告报告，但索引不会因此而终止。该查询的结果集被忽略。此查询中可以使用宏$maxid，它会被扩展为索引过程中实际得到的最大的文档 ID。

示例：

```
sql_query_post_index = REPLACE INTO counters ( id, val ) \
   VALUES ( 'max_indexed_id', $maxid )
```

8.1.21. sql_ranged_throttle

分区查询的间隔时间（throttling），单位是毫秒。可选选项，默认值为 0（无间隔时间）。仅适用于 SQL 数据源（mysql 和 pgsql）。

此选项旨在避免 indexer 对数据库服务器构成了太大的负担。它会使 indexer 在每个分区查询的步之后休眠若干毫秒。休眠无条件执行，并在取结果的查询之前执行。

示例：

```
sql_ranged_throttle = 1000 # sleep for 1 sec before each query step
```

8.1.22. sql_query_info

文档信息查询。可选选项，默认为空。仅对 mysql 数据源有效。

仅被 CLI 搜索所用，用来获取和显示文档信息，目前仅对 MySQL 有效，且仅用于调试目的。此查询为每个文档 ID 获取 CLI 搜索工具要显示的文档信息。

示例：

```
sql_query_info = SELECT * FROM documents WHERE id=$id
```

8.1.23. xmlpipe_command

调用 xmlpipe 流提供者的 Shell 命令。必须选项。仅对 xml_pipe 和 xml_pipe2 数据源有效。

指定的命令会被运行，其输出被当作 XML 文档解析。具体格式描述请参考节 3.8, “xmlpipe 数据源”或节 3.9, “xmlpipe2 数据源”

示例：

```
xmlpipe_command = cat /home/sphinx/test.xml
```

8.1.24. xmlpipe_field

声明 xmlpipe 数据字段。可声明同一类型的多个不同名称的属性，可选项。仅对 xmlpipe2 数据源有效。参考节 3.9, “xmlpipe2 数据源”

示例：

```
xmlpipe_field = subject
xmlpipe_field = content
```

8.1.25. xmlpipe_attr_uint

声明 xmlpipe 整型属性。可声明同一类型的多个不同名称的属性，可选项。仅对 xmlpipe2 数据源有效。语法与 sql_attr_uint 完全相同。

示例：

```
xmlpipe_attr_uint = author
```

8.1.26. xmlpipe_attr_bool

声明 xmlpipe 布尔型属性。可声明同一类型的多个不同名称的属性，可选项。仅对 xmlpipe2 数据源有效。语法与 sql_attr_bool 完全相同。

示例：

```
xmlpipe_attr_bool = is_deleted # will be packed to 1 bit
```

8.1.27. xmlpipe_attr_timestamp

声明 xmlpipe UNIX 时间戳属性。可声明同一类型的多个不同名称的属性，可选项。仅对 xmlpipe2 数据源有效。语法与 sql_attr_timestamp 完全相同。

示例：

```
xmlpipe_attr_timestamp = published
```

8.1.28. xmlpipe_attr_str2ordinal

声明 xmlpipe 字符序数属性。可声明同一类型的多个不同名称的属性，可选项。仅对 xmlpipe2 数据源有效。语法与 sql_attr_str2ordinal 完全相同。

示例：

```
xmlpipe_attr_str2ordinal = author_sort
```

8.1.29. xmlpipe_attr_float

声明 xmlpipe 浮点型属性。可声明同一类型的多个不同名称的属性，可选项。仅对 xmlpipe2 数据源有效。语法与 sql_attr_float 完全相同。

示例：

```
xmlpipe_attr_float = lat_radians
xmlpipe_attr_float = long_radians
```

8.1.30. xmlpipe_attr_multi

声明 xmlpipe MVA 属性。可声明同一类型的多个不同名称的属性，可选项。仅对 xmlpipe2 数据源有效。

这个选项为 xmlpipe2 流声明一个 MVA 属性标签。该标签的内容会被试图解析成一个整型值的列表（数组），此列表构成一个 MVA 属性值，这与把 MVA 属性的数据源设置为“字段”时 sql_attr_multi 分析 SQL 列内容的方式类似。

示例：

```
xmlpipe_attr_multi = taglist
```

8. 2. 索引配置选项

8.2.1. type

索引类型。可选选项，默认值为空（索引为简单本地索引）。可用的值包括空字符串或“distributed”

Sphinx 支持两种不同的索引类型：本地——在本机上存储和处理，和远程——不仅涉及本地搜索，而且同时通过网络向远程 searchd 实力做查询。索引类似选项使您可以选择使用何种索引。索引默认是本地型。指定“distributed”索引类型会运行分布式搜索，参看节 4.7, “分布式搜索”

示例：

```
type = distributed
```

8.2.2. source

向本地索引增加文档源。可以出现多次，必须选项。

为当前索引指定一个从中可以获取文档的文档源。必须至少有一个文档源。可以有多个文档源，任何数据源类型都可接受：即您可以从 MySQL 服务器中获取一部分数据，从 PostgreSQL 中获取另一部分，再在文件系统上使用 xmlpipe2 wrapper 获取一部分。

然而，对源数据却有一些限制。首先，文档 ID 必须在所有源的总体上是唯一的。如果这个条件不满足，那可能导致非预期的搜索结果。其次，源的模式必须相同，以便在同一个索引

中存储。

数据来源的 ID 不会被自动存储。因此，为了获知匹配的文档是从哪个数据源中来的，需要手工存储一些额外的信息。通常有两种方法：

1. 修改文档 ID，将源 ID 编码进去：

```
source src1
{
       sql_query = SELECT id*10+1, ... FROM table1
       ...
}

source src2
{
       sql_query = SELECT id*10+2, ... FROM table2
       ...
}
```

2. 将数据来源存储为一个属性

```
source src1
{
       sql_query = SELECT id, 1 AS source_id FROM table1
       sql_attr_uint = source_id
       ...
}

source src2
{
       sql_query = SELECT id, 2 AS source_id FROM table2
       sql_attr_uint = source_id
       ...
}
```

示例：

```
source = srcpart1
source = srcpart2
source = srcpart3
```

8.2.3. path

索引文件的路径和文件名（不包括扩展名）。必须选项。

path 既包括文件夹也包括文件名，但不包括扩展名。indexer 在产生永久和临时索引文件的最终名字时会附加上不同的扩展名。永久数据文件有几个不同的扩展名，都以“.sp”开头，临时文件的扩展名以“.tmp”开头。如果 indexer 没有成功地自动删除.tmp*文件，手工删除是安全的。

以下是不同索引文件所存储的数据种类，供参考：

- .spa 存储文档属性（仅在 extern_docinfo 存储模式中使用）
- .spd 存储每个词 ID 可匹配的文档 ID 列表
- .sph 存储索引头信息
- .spi 存储词列表（词 ID 和指向.spd 文件的指针）

- .spm 存储 MVA 数据
- .spp 存储每个词 ID 的命中（或者说记账，或者词的出现）列表

示例：

```
path = /var/data/test1
```

8.2.4. docinfo

文档信息(docinfo)的存储模式。可选选项，默认是“extern”已知的值包括'none', 'extern' 和 'inline'.

此选项确切定义了文档信息在磁盘和 RAM 中的物理存储方式。“none”意思是根本不存储文档信息（没有任何属性）。通常并不需要显式设置为“none”因为当没有配置任何属性时 Sphinx 会自动选择“none”。“inline”代表文档信息与文档 ID 列表一同存储在在.spd 文件中。“extern”代表文档信息与文档 ID 分开（在外部）存储（在.spa 文件中）。

基本上，外部存储的文档信息在查询时必须保持在内存中。这是性能的原因。因此有时候“inline”是唯一的选择。然而这种情况并不多见，文档信息默认是“extern”存储的。深入的探讨和 RAM 使用的估计请参见节 3.2, “属性” 。

示例：

```
docinfo = inline
```

8.2.5. mlock

已缓冲数据的内存锁定。可选选项，默认为 0（不调用 mlock()）

为提高性能，searchd 将.spa 和.spi 文件预取到内存中，并一直在内存中保存它们的拷贝。但如果一段时间内没有对该索引的搜索，则对这份缓冲的拷贝没有内存访问，而操作系统可能会决定将它交换到磁盘上去。对这些“冷却”了的索引的访问会导致其交换回内存并得到一个较大的延迟。

将 mlock 选项设置为 1 会使 Sphinx 使用 mlock(2)系统调用将存储上述缓冲了的数据的系统内存锁定，这将避免内存交换（详情参见 man 2 mlock）。mlock(2)是特权调用，因此可能需要 searchd 以 root 账户运行或通过其他办法赋予足够的权限。如果 mlock()失败会发出警告，但索引会继续进行。

示例：

```
mlock = 1
```

8.2.6. morphology

词形处理器的列表。可选选项，默认为空（不使用任何词形处理器）。

词形处理器可以将待索引的词从各种形态变成基本的规则的形态。例如，英语词干提取器（English stemmer）可以将“dogs”和“dog”都变成“dog”，这使搜索这两个词的结果都相同。

内置的词形处理器包括英语词干提取器，俄语词干提取器（支持 UTF-8 和 Windows-1251 编码），Soundex 和 Metaphone。后面两个会将词替换成特殊的语音编码，这会使发音相近的词表示形式相同。Snowball 项目的 libstemmer 库提供的额外词干提取器可以通过在编译期对

configure 脚本使用—with-libstemmer 选项来启用。内建的英语和俄语词干提取器要比它们在 libstemmer 中的对应物运行更快，但它们的结果可能略有不同，因为内建的版本基于较旧的版本。Metaphone 基于 Double Metaphone 算法实现。

在 morphology 选项中可使用的内建值包括“none”，“stem_en”，“stem_ru”，“stem_enru”，“soundex”和“metaphone”。libstemmer 提供的额外值格式为“libstemmer_XXX”，XXX 指 libstemmer 算法的代号。（完整列表参见 libstemmer_c/libstemmer/modules.txt）

可以指定多个词干提取器（以逗号分隔）。这些提取器按列出的顺序应用于输入词串，整个处理会在第一个真正修改了原词的词干提取器之后停止。另外，当 wordforms 特性启用时，词会现在词形字典中查询，如果词典中有对应的条目，那么词干提取器干脆不会被使用。换个说法，wordforms 选项可以用来补充指定词干提取器的例外情况。

示例：

```
morphology = stem_en, libstemmer_sv
```

8.2.7. stopwords

停用词文件列表（空格分隔）。可选选项，默认为空。

停用词是不被索引的词。停用词表一般包括最常用的高频词，因为它们对搜索结果没有多大帮助却消耗很多处理资源。

可以指定多个文件名，用空格分隔。所有文件都会被载入。停用词文件的格式是简单的纯文本。其编码必须与 charset_type 选项所指定的索引编码相匹配。文件数据会根据 charset_type 选项的设置，分词，因此您可以使用与待索引数据相同的分隔符。词干提取器也会在停用词文件的分析中使用。

尽管停用词不会被索引，它们却影响关键词的位置。例如，假设“the”是一个停用词，文档 1 包含一行“in office”，而文档 2 包含“in the office”。将“in office”作为确切词组搜索则只会得到文档 1，虽然文档 2 里的 the 是停用的。

示例：

```
stopwords = /usr/local/sphinx/data/stopwords.txt
stopwords = stopwords-ru.txt stopwords-en.txt
```

8.2.8. wordforms

词形字典。可选选项，默认为空。

词形字典在输入文档根据 charset_table 切碎后使用。本质上，它使您可以将一个词替换成另一个。这通常被用来将不同的词形变成一个单一的标准形式（即将词的各种形态如“walks”，“walked”，“walking”变为标准形式“walk”）。也可以用来实现取词根的例外情况，因为词形字典中可以找到的词不会经过词干提取器的处理。

索引和搜索中的输入词都会利用词典做规则化。因此要使词形字典的更改起作用，需要重新索引并重启 searchd。

Sphnix 的词形支持被设计成可以很好地支持很大的字典。它们轻微地影响索引速度：例如，1M 个条目的字典会使索引速度下降 1.5 倍。搜索速度则完全不受影响。额外的内存占用大体上等于字典文件的大小，而且字典是被多个索引共享的，即如果一个 50MB 的词形字典文

件被 10 个不同的索引使用了，那么额外的 searchd 内存占用就是大约 50MB。

字典文件的格式是简单的纯文本。每行包括一个源词形和一个目标词形，编码应与 charset_type 选项所指定的完全相同，二者用大于号分隔。文件载入时会经过 charset_table 选项指定的规则的处理。因此基本上在大小写是否敏感这个问题上它是与待索引的全文数据相同的，即通常是大小写无关的。一下是个文件内容的例子：

```
walks > walk
walked > walk
walking > walk
```

我们提供了一个 spelldump 工具，它可以帮您从 ispell 格式的.dict 和.aff 字典文件生成 Sphinix 可接受的格式。

示例：

```
wordforms = /usr/local/sphinx/data/wordforms.txt
```

8.2.9. exceptions

Token 特例文件。可选选项，默认为空。

对于中文用户，这一选项无效。Coreseek 为 Sphinx 贡献的中文分词法内置了 Token 特例化支持，具体参阅 Coreseek MMSeg 分词法的文档。不过，值得高兴的是，Token 特例化的文件格式两者是同样的。

此选项允许将一个或多个 Token（Token 中，可以包括在正常情况下会被过滤的字符）映射成一个单独的关键词。exceptions 选项与 wordforms 选项很类似，它们都代表某种映射，但有一些重要的不同点。

这些不同点简要总结如下：

- exceptions 大小写敏感, wordforms 大小写无关
- exceptions 允许检测一串记号 wordforms 仅对单独的词有效
- exceptions 可以使用 charset_table 中**没有**的特殊符号 wordforms 完全遵从 charset_table
- exceptions 在大字典上性能会下降 wordforms 则对百万级的条目应对自如

输入文件的格式仍然是纯文本，每行一个分词例外，而行的格式如下：

```
map-from-tokens => map-to-token
```

示例文件：

```
AT & T => AT&T
AT&T => AT&T
Standarten   Fuehrer => standartenfuhrer
Standarten Fuhrer => standartenfuhrer
MS Windows => ms windows
Microsoft Windows => ms windows
C++ => cplusplus
c++ => cplusplus
C plus plus => cplusplus
```

这里全部的记号都是大小写敏感的：它们不会按 charset_table 选项的规则处理。因此，在上述例外文件下，“At&t”会被分成两个关键字“at”和“t”，因为其中的小写字母。而“AT&T”却被精确匹配并产生一个单独的关键字“AT&T”。

需要注意的是，前述映射文件的目标关键词（右侧）a)总是被解释成一个**单独的**词，而且 b)不仅是大小写敏感的，而且是空白符号敏感的！在上述样例中，查询“ms windows”不会匹配包含“MS Windows”的文档。这个查询会被解释成两个词“ms”和“Windows”。而“MS Windows”映射到的是一个单独的关键字“ms windows”，包括中间的空格。另一方面“standartenfuhrer”会取回带有“Standarten Fuhrer”或者“Standarten Fuehrer”内容的文档（大写字母必须与此处列出的完全相同），或者关键词本身大小写随意的任何版本，例如“staNdarTenfUhreR”。（然而“standarten fuhrer”不会匹配。这段文本无法与列出的任何一个例外相匹配，因为大小写不同。因此被索引为两个分开的关键字）

映射源部分的空白符（white space）不会被忽略，但空白符的数量无所谓。任何数量的空白符都匹配已索引的文档或者查询中的任意数量的空白符。例如映射源部分（“=>”左端）的“AT□&□T”可以匹配“AT□□&□T”，不管被映射部分或已索引全文数据中实际有几个空格。根据上述例子中的第一条，上述文本会作为“AT&T”关键字被索引。

对于中文用户，这个特性目前尚不被支持。Coreseek 将在后续版本支持这个特性。

exceptions 选项也允许特殊字符（这是通用 charset_table 选项规则的例外（exception），此选项因而得名）。假设您一般不想把“+”当作有效的字符，但仍想搜索一些例外情况，比如“C++”。上述例子正好可以做到这点，完全不管哪些字符在表中，哪些字符不在。

exceptions 选项被应用于原始输入文档和索引、搜索时的查询数据。因此要使文件的改动生效，需要重建索引并重启 searchd。

示例：

```
exceptions = /usr/local/sphinx/data/exceptions.txt
```

8.2.10. min_word_len

最小索引词长度。可选选项，默认为 1（索引任何词）

只有长度不小于这个最小索引词长度的词会被索引。例如，如果 min_word_len 为 4，那么“the”这个词不会被索引，但“they”会。

示例：

```
min_word_len = 4
```

8.2.11. charset_type

字符集编码类型。可选选项，默认为“sbcs”。已知的值包括“sbcs”和“utf-8”。

对于中文用户，可选的值还可以有“zh_cn.utf-8 ”和“zh_cn.gbk”。当设置 charset_type 值为上面的两种时，系统默认您开启了中文分词特性。

不同的编码将它们的内部字符代码映射到特殊字节序列的方法不同。目前两个最常见的方法是单字节编码和 UTF-8。它们对应的 charset_type 值分别是“sbcs”（代表 Single Byte Character Set 单字节字符集）和“utf-8”。选定的编码类型会在搜索被使用的任何情况下使用：索引数据时，对索引查询时，产生摘要时，等等。

注意，尽管“utf-8”暗示解码出来的值应按 unicode 码点数值对待，“sbcs”却对应一系列不同的编码，它们对不同字节值的处理不同，这要在 charset_table 设置中正确地反应出来。例如，同一个值 244（十六进制 0xE0）根据使用的是 koi-8r 还是 windows-1251 编码而映射到不同

的俄语字符。

示例：

```
charset_type = utf-8
```

8.2.12. charset_table

接受的字符表和大小写转换规则。可选选项，默认值与 charset_type 选项的值有关。

对于中文用户，Coreseek 提供的 MMseg 分词法内置了可接受的字符表，并且用户不可修改。当启用分词功能时，自动开启。

charset_table 频繁应用于 Sphinx 的分词过程，即从文档文本或查询文本中抽取关键字的过程。它控制哪些字符被当作有效字符接受，哪些相反，还有接受了的字符如何转换（例如大小写信息保留还是去除）。

可以把 charset_table 想成一个对超过 100K 个 Unicode 字符中每一个的映射关系的大表（或者一个 256 个字符的小表，如果你使用 SBCS）。默认每个字符都对应 0，这表示它不在关键字中出现，应被视为分隔符。一旦在此表中被提及，字符就映射到另一个字符（通常是它自身或者自身的小写版本），同时被当作一个可以出现在关键字中的有效字符。

值的格式是逗号分隔的映射列表。两种最简单的映射分别是声明一个字符为有效和将一个简单字符映射为另一个字符。但用这种格式指定整个表会导致其体积臃肿、无法管理。因此提供了一些语法上的快捷方式，用它们可以一次指定一定范围的字符。详细的列表如下：

A->a
: 单个字符映射，声明源字符“A”为允许出现在关键字中的字符，并将之映射到目的字符“a”（这并**没有**声明“a”是允许的）

A..Z->a..z
: 范围映射，声明源范围中的全部字符允许出现在关键字中，并将它们映射到目的范围。并**不**声明目的范围是允许的。

a
: 单一字符映射，声明一个允许的字符，将它映射到它自身。相当于单个字符映射 a->a。

a..z
: 杂散范围映射，声明范围中的全部字符为允许，将它们映射到自身。相当于范围映射 a..z->a..z

A..Z/2
: 棋盘范围映射。映射每相邻两个字符到其中的第二个。形式化地说，声明范围中的奇数字符为允许，将它们映射到偶数字符上。同时允许偶数字符并映射到其自身。例如，A..Z/2 相当于 A->B, B->B, C->D, D->D, ..., Y->Z, Z->Z。这个映射接将捷径方便声明大小写字符交替而非大小写字符分别连续的 Unicode 块。

编码为 0 到 31 之间的控制字符总是被视作分隔符。编码 32 到 127 的字符即 7 位 ASCII 字符可以原样使用在映射中。为避免配置文件的编码问题，8 位 ASCII 字符和 Unicode 字符必须以 U+xxx 形式指定，“xxx”是码点对应的十六进制数。也可以用这个形式编码 7 位 ASCII 编码中的特殊字符，例如用 U+20 来编码空格符，U+2E 来编码句点，U+2C 来编码逗号。

示例：

```
# 'sbcs' defaults for English and Russian
charset_table = 0..9, A..Z->a..z, _, a..z, \
      U+A8->U+B8, U+B8, U+C0..U+DF->U+E0..U+FF, U+E0..U+FF

# 'utf-8' defaults for English and Russian
charset_table = 0..9, A..Z->a..z, _, a..z, \
      U+410..U+42F->U+430..U+44F, U+430..U+44F
```

8.2.13. ignore_chars

忽略字符表。可选选项，默认为空。

有些字符，如软断字符(U+00AD)，不是仅仅要当作分隔符，而且应该被完全忽略。例如，如果“-”只是不在 charset_table 里，那么“abc-def”会被当作两个关键字“abc”和“def”来索引。相反，如果将“-”加到 ignore_char 列表中，那么相同的文本会被当作一个单独的关键字“abcdef”索引。

此选项的语法与 charset_table 相同，但只允许声明字符，不允许映射它们。另外，忽略的字符不能出现在 charset_table 里。

示例：

```
ignore_chars = U+AD
```

8.2.14. min_prefix_len

索引的最小前缀长度。可选选项，默认为 0（不索引前缀）。

前缀索引使实现“wordstart*”形式的通配符成为可能（通配符语法的细节请参考 enable_star 选项）。当最小前缀长度被设置为正值，indexer 除了关键字本身还会索引所有可能的前缀（即词的开头部分）。太短的前缀（小于允许的最小值）不会被索引。

例如，在 min_prefix_len=3 设置下索引关键字“example”会导致产生 5 个索引项“exa”, “exam”, “examp”, “exampl”和该词本身。对这个索引搜索“exam”会得到包含“example”的文档，即使该文档中没有“exam”自身。然而，前缀索引会使索引体积急剧增大（因为待索引关键字增多了很多），而且索引和搜索的时间皆会恶化。

在前缀索引中没有自动的办法可以提高精确匹配（整个词完全匹配）的评分，但有一些技巧可以实现这个功能。首先，可以建立两个索引，一个带有前缀索引，另一个没有，同时在这两个索引中搜索，然后用 SetIndexWeights() 来设置二者的权重。其次，可以启用星号语法并重写扩展模式的查询。

```
# in sphinx.conf
enable_star = 1

// in query
$cl->Query ( "( keyword | keyword* ) other keywords" );
```

示例：

```
min_prefix_len = 3
```

8.2.15. min_infix_len

索引的最小中缀长度。可选选项，默认为 0（不索引中缀）。

中缀索引是实现“start*”, “*end”, and “*middle*”等形式的通配符成为可能（通配符语法的细节请参考 enable_star 选项）。当最小中缀长度设置为正值，indexer 除了对关键字本身还会对所有可能的中缀（即子字符串）做索引。太短的中缀（短于允许的最小长度）不会被索引。

例如，在 min_infix_len=2 设置下索引关键字“test”会导致产生 6 个索引项 "te", "es", "st", "tes", "est"等中缀和词本身。对此索引搜索“es”会得到包含“test”的文档，即使它并不包含“es”本身。然而，中缀索引会使索引体积急剧增大（因为待索引关键字增多了很多），而且索引和搜索的时间皆会恶化。

在中缀索引中没有自动的办法可以提高精确匹配（整个词完全匹配）的评分，但可以使用与 prefix indexes 选项中相同的技巧。

示例：

```
min_infix_len = 3
```

8.2.16. prefix_fields

做前缀索引的字段列表。可选选项，默认为空（所有字段均为前缀索引模式）。

因为前缀索引对索引和搜索性能均有影响，可能需要将它限制在某些特定的全文数据字段：例如，对 URL 提供前缀索引，但对页面内容不提供。prefix_fields 指定哪些字段要提供前缀索引，其他字段均会使用普通模式。值的格式是逗号分隔的字段名字列表。

示例：

```
prefix_fields = url, domain
```

8.2.17. infix_fields

做中缀索引的字段列表。可选选项，默认为空（所有字段均为中缀索引模式）。

与 prefix_fields 选项类似，但限制的是哪些字段做中缀索引。

示例：

```
infix_fields = url, domain
```

8.2.18. enable_star

允许前缀/中缀索引上的星号语法（或称通配符）。可选选项，默认为 0（不使用通配符），这是为了与 0.9.7 版本的兼容性。已知的值为 0 和 1。

此特性启用搜索前缀或中缀索引时的“星号语法”，或者说通配符语法。仅影响搜索，因此要使改变生效只须重启 searchd，而不需要重新索引。

默认值为 0，意思是禁止星号语法，所有关键字都根据索引时的 min_prefix_len 和 min_infix_len 设置被视为前缀或者中缀。取值 1 的意思是星号（“*”）可以用在关键字的前面或后面。星号与零个或多个字符匹配。

例如，假设某索引启用了中缀索引，且 enable_star 值为 1。搜索过程按如下工作：

1. 查询“abcdef” 仅匹配确切含有“abcdef”这个词的文档
2. 查询“abc*”可匹配含有以“abc”开头的词的文档（包括精确匹配词“abc”的文档）。
3. 查询“*cde*”匹配在任何地方含有“cde”的词的文档（包括精确匹配词“cde”的文档）
4. 查询“*def”匹配含有以“def”结束的词的文档（包括精确匹配词“def”的文档）

示例：

```
enable_star = 1
```

8.2.19. ngram_len

n-gram 索引的 n-gram 长度。可选选项，默认为 0（禁用 n-gram 索引）已知的值是 0 和 1（其他长度尚未实现）

对于中文用户，在启用了中文分词的情况下，本节内容可忽略

n-gram 提供对未分词 CJK（Chinese, Japanse, Koreasn 中日韩）文本的基本支持。CJK 搜索的问题在于词与词之前没有清晰的界限。理想中，文本可以通过一个称作分词程序（segmenter）的特殊程序的过滤，之后分隔符即被加入到适当位置。然而分词过程缓慢而易错，因此通常会转而索引连续的一组 N 个字符，或称 n-gram。

启用此特性，CJK 字符流会被当作 n-gram 索引。例如，如果输入文本为“ABCDEF”（A 到 F 均代表 CJK 字符），而此选项设置的长度为 1，那它们会被当作“A B C D E F”而索引。（如果此选项设置的长度是 2，那会产生“AB BC CD DE EF”，但目前仅支持 1）。只有那些在 ngram_chars 选项表中列出的字符会这样分割，其他不受影响。

注意，如果搜索查询是已分词的，即单独的词之间有分隔符分隔，那么在扩展模式中将这些词放入引号中搜索会得到正确的匹配结果，即使文档没有分词。例如，假设原查询为“BC DEF”，在应用程序端用引号将索引包起来，看起来是“BC” “DEF”（**包括**引号），这个查询被传给 Sphinx 并在其内部分割成 1-gram，查询变成“B C” “D E F”，仍然包括作为词组查询操作符的引号。该查询会匹配正确的文本，即使文本中没有相应的分隔符。

即使搜索查询没有分词，Sphinx 也可以返回较好的结果，这要感谢基于词组的相关度计算：它会使相近的词组匹配（在 n-gram 中 CJK 词相当于多个字符的词匹配）排在前面。

示例：

```
ngram_len = 1
```

8.2.20. ngram_chars

n-gram 字符列表。可选选项，默认为空。

对于中文用户，在启用了中文分词的情况下，本节内容可忽略

与 ngram_len 选项联用，此列表定义了从中抽取 n-gram 的字符序列。其他字符组成的词不受 n-gram 索引特性的影响。值的格式与 charset_table 相同。

示例：

```
ngram_chars = U+3000..U+2FA1F
```

8.2.21. phrase_boundary

词组边界符列表。可选选项，默认为空。

此列表控制哪些字符被视作分隔不同词组的边界，每到一个这样的边界，其后面的词的“位置”值都会被加入一个额外的增量，可以借此用近似搜索符来模拟词组搜索。语法与 charset_table 选项相似，但没有字符之间的映射关系，而且这些词组边界符不能重复出现在其他任何设置选项中。

自每个词组边界起，后面的词的“位置”都会被加入一个额外的增量（由 phrase_boundary_step 定义）。这使通过近似搜索符实现词组搜索成为可能：不同词组中的词之间的距离肯定大于 phrase_boundary_step，因此相似距离小于 phrase_boundary_step 的近似搜索其实是在搜索在一个词组范围内出现了全部给定查询词的情况，相当于词组搜索。

只有词组边界符后面紧跟着一个分隔符时，词组边界才被激活，这是为了避免 S.T.A.L.K.E.R 或 URLs 等缩写被错当成若干个连续的词组（因为“.”属于词组边界符）。

这一选项可以用于发现潜在的某些麻烦制造者。

示例：

```
phrase_boundary = ., ?, !, U+2026 # horizontal ellipsis
```

8.2.22. phrase_boundary_step

词组边界上词位置的增量。可选选项，默认为 0.

在词组边界上，当前词位置会加上此选项设置的额外增量。详细请参考 phrase_boundary 选项。

Example:

示例：

```
phrase_boundary_step = 100
```

8.2.23. html_strip

是否从输入全文数据中去除 HTML 标记。可选标记，默认为 0。已知值包括 0（禁用）和 1（启用）。

此特性对 xmlpipe 数据源无效（建议升级到 xmlpipe2）。这个去除 HTML 标记的模块应该很好地工作于正确格式化的 HTML 和 XHTML，但就像大多数浏览器一样，对于格式错误的文本（例如带有无法配对的<和>的 HTML）可能产生不希望的输出。

只有 HTML 标签和 HTML 注释会被删除。要同时删除标签的内容（例如要删除内嵌的脚本），请参考 html_remove_elements 选项。标签名没有限制，即任何看起来像有效的标签开头、结束或者注释的内容都会被删除。

示例：

```
html_strip = 1
```

8.2.24. html_index_attrs

去除 HTML 标记时要索引的标记语言属性列表。可选选项，默认为空（不索引标记语言属性）。

指定被保留并索引的 HTML 标记语言属性，即使其他 HTML 标记被删除。格式是对每个标记列举可以索引的属性，请看下例：

示例：

```
html_index_attrs = img=alt,title; a=title;
```

8.2.25. html_remove_elements

HTML 元素列表，不仅这些元素本身会被删除，它们的中间包括的文字内容也会被删除。可选选项，默认为空串（不删除任何元素的内容）。

此特性允许删除元素的内容，即在开始标记和结束标记之间的所有东西。用于删除内嵌的脚本或 CSS 等。短格式的元素（即
）被适当地支持，即，这种标记后面的内容**不会**被删除。

值为逗号分隔的标签名称列表。标签名大小写无关。

示例：

```
html_remove_elements = style, script
```

8.2.26. local

分布式索引（distributed index）中的本地索引声明。可以出现多次，可选选项，默认为空。

此设置用于声明分布式索引被搜索时要搜索的本地索引。全部本地索引会被**依次**搜索，仅使用 1 个 CPU 或核。要并行处理，可以配置 searchd 查询它自身（细节参考节 8.2.27, “代理”）。可以为每个分布式索引声明多个本地索引。每个本地索引可以在其他分布式索引中多次引用。

示例：

```
local = chunk1
local = chunk2
```

8.2.27. agent

分布式索引（distributed index）中的远程代理和索引声明。可以出现多次，可选选项，默认为空。

此设置用来声明搜索分布式索引时要搜索的远程代理。代理可以看作网络指针，它指定了主机、端口和索引名。在最基本的情况下，代理可以与远程物理主机对应。更严格的来说，这不一定总是正确：可以将多个代理指向同一台远程主机，甚至指向同一个 searchd 示例（以便利用多个 CPU 或核）

值的格式如下：

```
agent = hostname:port:remote-indexes-list
```

“hostname”是远程主机名，“port”是远程 TCP 端口，而“remote-index-list”是一个逗号分隔的远程索引列表。

全部代理会被并行搜索。然而同一个代理的多个索引是依次搜索的。这使您可以根据硬件来优化配置。例如，如果两个远程索引存储在一个相同的硬盘上，最好是配置一个带有多个按顺序搜索的索引，避免频繁的磁头寻址。如果这些索引存储在不同的硬盘上，那配置两个代理会更有利，因为这可以使工作完全并行。对于 CPU 也是如此，虽然在两个进程间切换对性能的影响比较小而且常常被完全忽略。

在有多个 CPU 和硬盘的机器上，代理可以指向相同的机器以便并行地使用硬件，降低查询延迟。并不需要为此设置多个 searchd 实例，一个实例与自身通信是合法的。以下是一个示例设置，它是为一台有 4 个 CPU 的机器准备的，可以并行地使用 4 个 CPU，各处理一个查询。

```
index dist
{
       type = distributed
       local = chunk1
       agent = localhost:3312:chunk2
       agent = localhost:3312:chunk3
       agent = localhost:3312:chunk4
}
```

注意其中一块是本地搜索的，而同一个 searchd 示例又向本身查询，以便并行地启动其他三个搜索。

示例：

```
agent = localhost:3312:chunk2 # contact itself
agent = searchbox2:3312:chunk3,chunk4 # search remote indexes
```

8.2.28. agent_connect_timeout

远程代理的连接超时时间，单位为毫秒。可选选项，默认为 1000（即 1 秒）。

连接到远程代理时，searchd 最多花这些时间等待 connet()调用成功完成。如果达到了超时时间 connect()仍没有完成，而 retries 选项是启用的，那么将开始重试。

示例：

```
agent_connect_timeout = 300
```

8.2.29. agent_query_timeout

Remote agent query timeout, in milliseconds. Optional, default is 3000 (ie. 3 seconds).

远程代理查询超时时间，以毫秒为单位。可选选项，默认为 3000（即 3 秒）。

连接后，searchd 最多花这这些时间等到远程查询完成。这个超时时间与连接超时时间是完全独立的。因此一个远程代理最多能造成的延迟为 agent_connection_timeout 与 agent_query_timeout 之和。如果超时时间已到，查询**不会**再重试，同时产生警告。

示例：

```
agent_query_timeout = 10000 # our query can be long, allow up to 10 sec
```

8.2.30. preopen

预先打开全部索引文件还是每次查询时再打开索引。可选选项，默认为 0（不预先打开）。

此选项令 searchd 在启动时（或轮换索引时）预先开打全部索引文件并在运行过程中保持打开。目前，默认是不预先打开这些文件（此行为可能在未来改变）。预先打开的每个索引文件会占用若干（目前是两个）文件描述符。但每次查询可以节约两个 open()调用而且不会受高负载情况下索引轮换过程中可能发生的微妙的竞争条件（race condition）的影响。另一方面，当提供很多索引服务（几百到几千）时，必须每次查询时打开索引文件以便节约文件描述符。

示例：

```
preopen = 1
```

8.2.31. charset_dictpath

指明分词法读取词典文件的位置，当启用分词法时，为必填项。在使用 LibMMSeg 作为分词库时，需要确保词典文件 uni.lib 在指定的目录下。

这个参数是 coreseek 对 Sphinx 的增强。你需要使用 Coreseek 为中文用户修改过的 Sphinx 或直接使用 Coreseek 全文检索服务器，此参数才有效。

示例：

```
charset_dictpath = dict
```

8.3. indexer 程序配置选项

8.3.1. mem_limit

索引过程内存使用限制。可选选项，默认 32M。

这是 indexer 不会超越的强制内存限制。可以以字节、千字节（以 K 为后缀）或兆字节（以 M 为后缀）为单位。参见示例。当过小的值导致 I/O 缓冲低于 8KB 时该限制会自动提高，此值的最低限度依赖于待索引数据的大小。如果缓冲低于 256KB，会产生警告。

最大可能的限制是 2047M。太低的值会影响索引速度，但 256M 到 1024M 对绝大多数数据集（如果不是全部）来说应该足够了。这个值设得太高可能导致 SQL 服务器连接超时。在文档收集阶段，有时内存缓冲的一部分会被排序，而与数据库的通信会暂停，于是数据库服务器可能超时。这可以通过提高 SQL 服务器端的超时时间或降低 mem_limit 来解决。

示例：

```
mem_limit = 256M
# mem_limit = 262144K # same, but in KB
# mem_limit = 268435456 # same, but in bytes
```

8.3.2. max_iops

每秒最大 I/O 操作次数，用于限制 I/O 操作。可选选项，默认为 0（无限制）。

与 I/O 节流有关的选项。它限制了每秒钟最大的 I/O 操作（读或写）的次数。值 0 意思是不

加限制。

indexer 在索引时可能导致突发的密集磁盘 I/O，因此需要限制它磁盘活动（给同一台机器上运行的其他程序留出一些资源，比如 searchd）。I/O 节流就是用来实现上述功能的。它的工作原理是，在 indexer 的连续磁盘 I/O 操作之间强制增加一个保证的延迟。现代 SATA 硬盘每秒钟可以执行多达 70-100 以上次的 I/O 操作（主要受磁头寻道时间的限制）。将索引 I/O 限制为上述数值的几分之一可以减轻由索引带来的搜索性能下降。

示例：

```
max_iops = 40
```

8.3.3. max_iosize

最大允许的 I/O 操作大小，以字节为单位，用于 I/O 节流。可选选项，默认为 0（不限制）。

与 I/O 节流有关的选项。它限制 indexer 文件 I/O 操作的单次最大大小。值 0 代表不加限制。超过限制的读写操作会被分成几个小的操作，并被 max_iops 计为多次。在本文写作时，全部 I/O 操作都被限制在 256KB 以下（默认的内部缓冲大小），因此大于 256KB 的 max_iosize 值没有任何作用。

示例：

```
max_iosize = 1048576
```

8.4. searchd 程序配置选项

8.4.1. address

要绑定的接口 IP 地址。可选项，默认为 0.0.0.0（即在所有接口上监听）。

此设置指定 searchd 在哪个接口上绑定、监听和接受输入的网络连接。默认值为 0.0.0.0，意思是在所有接口上监听。目前**不能**指定多个接口。

示例：

```
address = 192.168.0.1
```

8.4.2. port

searchd 的 TCP 端口号。必选项，默认为 3312。

示例：

```
port = 3312
```

8.4.3. log

日志文件名。可选项，默认为“searchd.log”。全部 searchd 运行时事件会被记录在这个日志文件中。

示例：

```
log = /var/log/searchd.log
```

8.4.4. query_log

查询日志文件名。可选项，默认为空（不记录查询日志）。全部搜索查询会被记录在此文件中。其格式在节 4.8,“searchd 查询日志格式”中描述

示例：

```
query_log = /var/log/query.log
```

8.4.5. read_timeout

网络客户端请求的读超时时间，单位是秒。可选项，默认是 5 秒。searchd 强制关闭在此时间内未能成功发出查询的客户端连接。

示例：

```
read_timeout = 1
```

8.4.6. max_children

子进程的最大数量（或者说，并行执行的搜索的数目）。可选项，默认为 0，不限制。

用来控制服务器负载。任何时候不可能有比此设置值更多的搜索同时运行。当达到限制时，新的输入客户端会被用临时失败（SEARCH_RETRY）状态码驳回，同时给出一个声明服务器已到最大连接限制的消息。

示例：

```
max_children = 10
```

8.4.7. pid_file

searchd 进程 ID 文件名。必选项。

PID 文件会在启动时重建（并锁定）。主守护进程运行时它含有该进程的 ID，而当守护进程退出时该文件会被删除。这个选项是必须的，因为 Sphinx 在内部使用它做如下事：检查是否已有一个 searchd 示例；停止 searchd；通知 searchd 应该轮换索引了。也可以被各种不同的外部自动化脚本所利用。

Example:

示例：

```
pid_file = /var/run/searchd.pid
```

8.4.8. max_matches

守护进程在内存中为每个索引所保持并返回给客户端的匹配数目的最大值。可选选项，默认值为 1000.

引入此选项是为了控制和限制内存使用，max_matches 设置定义了搜索每个索引时有多少匹配项会保存在内存中。每个找到的匹配项**都会被处理**，但只有它们中最佳的 N 个会在内存中保持并最终返回给客户端。假设索引中包括 2000000 个当前查询的匹配项，你几乎总是不需要它们中的**全部**。通常您需要扫描它们并根据某种条件（即按相关度排序、或者价格、或者其他什么）选出最好的那些，比如 500 个，并以在页面上显示 20 到 100 项。只跟踪最好的 500 个匹配要比保持全部的 2000000 个匹配项大大地节约内存和 CPU，之后可以对这些最佳匹配排序，然后丢弃除了要在页面上要显式的 20 项之外的结果。max_matches 控制“最佳 N 个匹配”中的 N。

此参数明显影响每个查询消耗的内存和 CPU。1000 到 10000 的值通常就可以满足需求，但更高的值要小心使用。粗心地把 max_matches 增加到 1000000 意味着 searchd 被迫为**每一个查询**分配 1M 条匹配项的缓冲。这会明显增大查询的内存消耗，有时会明显影响性能。

特别注意！此限制还可在**另一个**地方指定。max_matches 可以通过对应的 API 调用实时降低，该调用的默认值**也**是 1000。因此要使应用程序获取超过 1000 个匹配结果，必须修改配置文件，重启 searchd，再用 SetLimits()调用设置合适的限制。还要注意，API 调用设置的限制不能大于.conf 文件中的设置，这是为了预防恶意的或错误的请求。

示例：

```
max_matches = 10000
```

8.4.9. seamless_rotate

防止 searchd 轮换在需要预取大量数据的索引时停止响应。可选选项，默认为 1（启用无缝（seamless）轮换）。

索引可能包含某些需要预取到内存中的数据。目前.spa，.spi 和.spm 文件会被完全预取到内存中（它们分别包含属性数据，MVA 数据和关键字索引）。若无无缝轮换，轮换索引时会尽量使用较小的内存，并如下工作：

1. 新的查询暂时被拒绝（用“retry”错误码）
2. searchd 等待目前正在运行的查询结束
3. 旧的索引被释放，文件被重命名
4. 新的索引文件被重命名，分配所需的内存
5. 新的索引属性和字典数据预调进内存
6. searchd 恢复为新索引提供查询服务

然而，如果有大量的属性或字典数据，那么预调数据的步骤可能消耗大量的时间——预调 1.5GB 的文件可能需要几分钟的时间。

当启用了无缝轮换，轮换按如下工作：

1. 为新索引分配内存
2. 新索引的属性和字典数据异步地预调进内存
3. 如果成功，旧的索引被释放，新旧索引文件被重命名
4. 如果失败，释放新索引
5. 在任意时刻，查询服务都正常运行——或者使用旧索引，或者使用新索引

无缝轮换以轮换过程中更大的峰值内存消耗为代价（因为当预调新索引时.spa/.spi/.spm 数据的新旧拷贝需要同时保持在内存中）。平均内存耗用不变。

示例：

```
seamless_rotate = 1
```

8.4.10. preopen_indexes

是否在启动是强制重新打开所有索引文件。可选选项，默认为 0（不重新打开）。对所有提供服务的索引强制打开 preopen 选项，免得对每个索引手工指定了。

示例：

```
preopen_indexes = 1
```

8.4.11. unlink_old

索引轮换成功之后，是否删除以.old 为扩展名的索引拷贝。可选选项，默认为 1（删除这些索引拷贝）。

示例：

```
unlink_old = 0
```