Apache Solr Reference Guide

Covering Apache Solr 6.0

Apache Solr 参考指南

Table of Contents

关于本书

翻译自 apache solr ref guide 6.0

术语表

虽然目标是要翻译 Solr 参考指南，但遗憾的是很多术语不太容易精确的翻译过来。这个表格列出了 solr 一些重要的术语，及如何翻译成中文

英文	译文
solr	solr
core	core
collection	collection，集合
document	document，文档
shard	shard，分片
replica	replica，副本
leader	leader
node	node，节点
term	词条
token	词元

文档 字段 **schema** 设计

这一章探讨 solr 如何组织数据到文档和字段，及如何和 schema 一起工作

文档 字段 **schema** 概述

solr 基本前提是简单的。你给它很多信息，然后你可以问它问题并找到想要的信息。你提交所有信息的部分称为索引，你问问题称为查询。

理解 solr 如何工作的一个方法是想象一个食谱活页本。每次你添加食谱到本上，都会更新封底的索引。你列出每个原料和刚刚添加的食谱的页码。假设你添加了 100 个食谱，使用这个索引，你可以很快的找到使用了鹰嘴豆或洋蓟，或咖啡做原料的食谱。使用这个索引比一个接一个的查看食谱要快得多。想象一个有 1000 或 1,000,000 食谱的本本。

solr 让你创建一个有很多不同字段或很多类型条目的索引。上面的例子演示了如何创建只有一个字段 `ingredients` 的索引。你的索引应该还有其他字段，如烹调方式，例如 `asian`，`cajun` 或 `vegan`，以及索引字段准备时间。solr 可以回答诸如 "哪种卡津样式的食谱是以血橙作原料，准备时间少于 30 分钟" 这样的问题。

schema 就是你告诉 solr 应该如何从输入的文档创建索引的地方

solr 如何看这个世界

solr 的基本信息单位是文档(document)，用于描述某样事物的一组数据。一个食谱的文档包含原料，操作指南，准备时间，烹调时间，需要的工具，等等。举个例子，关于一个人的文档，应该包含这个人的名字，档案，喜爱的颜色，鞋的尺码。书的文档可包含标题，作者，出版年份，页数，等等。

在 solr 的宇宙里，文档由字段组成，字段是更明确的信息块。鞋码可以是个字段，姓和名可以是字段。

字段能包含不同种类的数据。例如，名字字段是文本(字符型数据)。鞋码字段可能是浮点数，值可能是 6 或 9.5。显然，字段的定义是灵活的(例如，你可以定义鞋码字段为文本而非浮点数)，但是，如果你正确的定义字段，solr 将正确的解释它们，你的用户执行查询时将得到更好的结果

你可以通过指定字段类型来告诉 solr 关于一类数据为一个字段。字段类型告诉 solr 如何解释字段及其如何被查询。

当你添加一个文档，solr 获取文档各字段的信息并添加这些信息到索引。当你执行一个查询，solr 就能迅速翻阅索引并返回匹配的文档。

字段解析

字段解析告诉 solr 创建索引时如何处理输入的数据。这个过程更精确的名称应该是处理或吸收，不过官方名称是解析。

例如，考虑一个人的文档里的档案字段。档案里的每个字都应该被索引，这样你就能迅速找到那些生活里与番茄酱，或星座，或密码有关的人。

但是，档案里可能包含很多你不在乎，也不想堵塞索引的字——例如 "the","a","to" 等等。此外，假设档案包含了 "Ketchup"(番茄酱) 这个字，首字母是大写的。如果有用户查询 "ketchup"，你想要 solr 告诉你档案包含了大写字母的人。

这些问题的解决方案就是字段解析。对于档案字段，你可以告诉 solr 怎样把档案分词，你可以告诉 solr 你想要把所有词转成小写，你也可以告诉 solr 移除重音标记。

字段解析是字段类型的重要部分。`Understanding Analyzers, Tokenizers, and Filters` 是字段解析的详细描述。

solr 的 schema 文件

solr 保存它期望理解的字段和字段类型的细节在 schema 文件。这个文件的名字和位置基于你如何配置 solr 而变化。

- `managed-schema` 是 solr 默认使用的名字，支持运行时通过 schema api 改变，或无 schema 模式。也可以配置一个新文件名，不过文件的内容仍然是由 solr 自动更新的。
- `schema.xml` 是 schema 文件的传统名字，可以被用户手工修改，使用 `ClassicSimilarityFactory`
- 如果使用的是 solrcloud，可能在本地文件系统找不到以上命名的文件。只能用 schema api(如果开启) 查看 schema，或通过 solr 管理界面的 `cloud screen`

不管你的 solr 用的是哪个名字的文件，文件的结构是不变滴。但是，你和文件交互的方式将改变。如果你在使用 managed schema，你只能用 shcema api 与之交互，不能手工编辑。如果不用 managed schema，只能手工修改该文件，不支持用 schema api 修改。

注意，如果你在用 solrcloud 而没有用 schema api，你可以用 upconfig 和 downconfig 命令创建一个 `schema.xml` 的本地拷贝并上传你的改变到 ZooKeeper。

solr 字段类型

字段类型定义了 solr 如何解释字段的数据及字段如何被查询。solr 默认包含了很多字段类型，而且也可以自定义。

定义和属性

字段类型的定义可包含 4 类信息

- 字段名(必须)
- 实现类名(必须)
- 如果字段类型是 `TextField`，字段解析的说明
- 字段类型属性 - 依赖于实现类，有些属性可能是必须的

schema.xml 里的字段定义

字段类型在 `schema.xml` 里定义。每个字段类型定义在 `fieldType` 元素之间。它们可以用 `types` 元素集中。下面是一个字段类型 `text_general` 定义的例子

```
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"/>
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymFilterFactory" synonyms="index_synonyms.txt" ignoreCase
="true" expand="false"/>
    -->
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"/>
    <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
      ignoreCase="true" expand="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

上面例子里的第一行包含了字段类型的名字，`text_general`，以及实现类的名字，`solr.TextField`。其余的定义是关于字段解析，参考 `Understanding Analyzers, Tokenizers, and Filters`

实现类确保字段被正确处理。在 `schema.xml` 里的类名，字符串 `solr` 是 `org.apache.solr.schema` 或 `org.apache.solr.analysis` 的简写。因此，`solr.TextField` 实际是 `org.apache.solr.schema.TextField` 。

字段类型属性

字段类型的 `class` 决定了字段类型大多数行为，但是仍然可以定义可选的属性。例如，下面的日期字段类型定义了 2 个属性， `sortMissingLast` 和 `omitNorms`

```
<fieldType name="date" class="solr.TrieDateField" sortMissingLast="true" omitNorms="tr
ue"/>
```

字段类型分为 3 个大类

- 字段类型的类
- 通用属性 所有字段类型都支持
- 字段默认属性 在字段类型上指定后，可以被字段继承替换字段默认行为的属性(存疑：如果字段上定义的属性和字段类型不同，应该是以字段上定义的为准)

通用属性

属性	描述	值
name	字段类型名字。这个值在字段定义时用于 "type" 属性。强烈建议名字以字母或下划线而非数字开头。目前并非强制要求	
class	存储和索引该类型数据的类。注意，类名可以用 "solr" 前缀，solr 会自动识别该类在哪个包 - 所以 "solr.TextField" 有效。如果使用第三方的类，使用完整的类名。"solr.TextField" 等效于 "org.apache.solr.schema.TextField""	
positionIncrementGap	对于多值字段，指定多值之间的距离，这能避免"虚假"的短语匹配	integer
autoGeneratePhraseQueries	用于文本字段。如果为 true，solr 自动对相邻的词条生成短语查询。如为 false，只有用双引号包围的词条才会作为短语处理	true 或 false
docValuesFormat	定义一个自定义 `DocValuesFormat` 的类型字段 ，需要一个 schema 感知的编解码器，诸如 `SchemaCodecFactory` 已在 `solrconfig.xml` 里配置好	n/a
postingsFormat	定义一个自定义 `PostingsFormat` 的类型字段，需要一个 schema 感知的编解码器，诸如 `SchemaCodecFactory` 已在 `solrconfig.xml` 里配置好	n/a

> Lucene 索引向后兼容只支持默认的编解码器。如果在 schema.xml 里自定义 `postingsFormat` 或 `docValuesFormat`，升级到新版本的 solr 之前可能要求你切换到默认的编解码器并优化索引以重新写入默认编解码器，或者在升级之后从零开始重建索引。

字段默认属性

这些属性要么在字段类型指定，要么在特定字段上覆盖字段类型的指定。每个属性的默认值依赖于对应的 `FieldType` 类，取决于 `<schema/>` 的 `version` 属性值。下面的表格包含了 solr 提供的大多数 `FieldType` 实现的默认值，如果 `schema.xml` 的 `version="1.6"`

属性	描述	值	隐含默认值
indexed	如为 true，字段值能用于查询检索匹配的文档	true/false	true
stored	如为 true，字段值可以被查询获取	true/false	true
docValues	如为 true，字段值将放入面向列的 `DocValues` 结构	true/false	false
sortMissingFirst sortMissingLast	当一个索引字段未提供时，控制文档(在索引)的位置(是在最前/最后)	true/false	false
multiValued	如为 true，单个文档在这个字段可能包含多个值	true/false	false
omitNorms	如为 true，省略与此字段关联的加权基准(这会关闭长度标准化，及索引时对该字段的加权，从而节省内存占用)。对所有原始(无需解析)的字段类型，诸如 int，float，date，bool，和 string，默认为 true。只有全文本字段或需要索引时加权的字段才需要加权基准(norms)	true/false	*
omitTermFreqAndPositions	如为 true，在提交字段时省略词条的频率/次数，位置，载荷。对于不需要这些信息的字段能提升性能，同时还减少了索引需要的存储空间。依赖位置信息的查询会无声的失败。对于所有非文本字段这个属性默认为 true	true/false	*
omitPositions	和 `omitTermFreqAndPositions` 类似，但是会保留词条频率/次数信息	true/false	*
termVectors termPositions termOffsets termPayloads	这些选项命令 solr 保持每个文档完整的词条向量，随意的包括在那些向量里每个词条的位置，偏移，及载荷信息。这些可用于加快高亮和其他辅助功能，但会极大增加索引的尺寸。在 solr 的典型应用里是没有必要的	true/false	false
required	如为 true，solr 拒绝添加一个该字段没有值的文档	true/false	false
useDocValuesAsStored	如果该字段开启了 `docValues`，设为 true 将允许该字段在 `fl=*` 时像一个存储的字段一样返回(即使该字段的 `stored=false`)	true/false	true

> term 的 `Vector` 实际上就是由 term 的 `positions`， `offset`， `payloads` 组成的

字段类型相似性

字段类型可以指定 `<similarity/>` ，可用于当给一个有该字段类型字段的文档评分时，只要 collection 的 "global" 相似性允许。默认情况，所有没有定义 similarity 的字段类型使用 `BM25Similarity` 。要了解更多细节，及配置 global 和 每个类型的 similarity，参考 `Other Schema Elements`

solr 自带字段类型

下面表格列出了 solr 可用的字段类型。`org.apache.solr.schema` 包包含列出的所有类

类	描述
BinaryField	二进制数据
BoolField	包含 true 或 false。"1"， "t"， "T" 为首字母视同 true，其他首字母的视同 false
CollationField	索引和范围查询支持 Unicode 排序规则。如果使用 ICU4J，ICUCollationField 是个更好的选择。参考 `Unicode Collation`
CurrencyField	支持货币和汇率
DateRangeField	支持日期范围索引，也包括时间点
ExternalFileField	从磁盘文件里拉取数值
EnumField	对于那些不容易以字母或数字顺序来保存的(例如一个严重程度的列表)，可定义一个值的枚举集。这个字段类型需要一个配置文件，用来有序的记录字段值
ICUCollationField	索引和范围查询支持 Unicode 排序规则。参考 `Unicode Collation`
LatLonType	相联系的纬度，经度对，纬度在前
PointType	任意维度的点，在蓝图或CAD绘图等资源里搜索时有用
PreAnalyzedField	提供了一种向 solr 发送连续的词元流的方法，词元流可附带独立的存储字段，并将此信息存储和索引而不做任何额外的处理
RandomSortField	该字段类型不包含值(?)查询如果在这个字段上做排序将随机排列。使用动态字段来使用这个特性
SpatialRecursivePrefixTreeFieldType	(简写为 RPT) 维度+逗号+经度字符串或其他的 WKT 格式
StrField	字符串(UTF8 编码或 Unicode)

TextField	文本，通常是多个单词或词元
TrieDateField	日期 `precisionStep="0"` 可按日期排序，索引最小 `precisionStep="8"` 默认值，可范围查询
TrieDoubleField	double，8字节 `precisionStep="0"` 可按数字排序，索引最小 `precisionStep="8"` 默认值，可范围查询
TrieField	这个类型必须指定 type 属性，可选的值为 `integer` ， `long` ， `float` ， `double` ， `date` 。等效于对应的 Trie 字段 `precisionStep="0"` 可按数字排序，索引最小 `precisionStep="8"` 默认值，可范围查询
TrieFloatField	float，4字节 `precisionStep="0"` 可按数字排序，索引最小 `precisionStep="8"` 默认值，可范围查询
TrieIntField	int，4字节 `precisionStep="0"` 可按数字排序，索引最小 `precisionStep="8"` 默认值，可范围查询
TrieLongField	long，8字节 `precisionStep="0"` 可按数字排序，索引最小 `precisionStep="8"` 默认值，可范围查询
UUIDField	通用唯一标识符，传入 "NEW"，solr 会创建一个 UUID。注意：在 SolrCloud 模式，对于大多数用户配置一个 UUIDField 默认值为 "NEW" 是不明智的(而且，如果 UUID 值配置为唯一键，也不可能)，因为这会导致每个 replica 的每个 document 都有一个唯一的 UUID 值。建议用 UUIDUpdateProcessorFactory 来生成 UUID 值

货币和汇率

`Currency` 字段类型支持以下特性

- 点查询
- 范围查询
- 函数范围查询
- 排序
- 使用货币代码或者符号解析货币
- 对称和非对称汇率

货币配置

`Currency` 字段类型在 `schema.xml` 定义，默认配置如下

```
<fieldType name="currency" class="solr.CurrencyField" precisionStep="8"
    defaultCurrency="USD" currencyConfig="currency.xml" />
```

这个例子里，我们定义了字段类型的名字和类，定义了 `defaultCurrency` 为 "USD"，即美元。还定义了 `currencyConfig`，使用名为 `currency.xml` 的文件。这个是默认货币到其他货币的汇率文件。还有一个代替的实现允许下载货币数据

索引期间，金额字段能被索引为本地货币。例如，一个欧洲电子商务网站上的商品，索引其价格字段为 "100,EUR" 是很合理的。价格和货币之间用逗号分隔，价格应该被编码为一个浮点值。

查询时，范围和点查询都可以支持。

汇率

通过指定提供者来配置汇率。Solr 自带 2 类提供者：`FileExchangeRateProvider` 或 `OpenExchangeRatesOrgProvider`

FileExchangeRateProvider

这个提供者需要你提供一个汇率文件。这是默认的，意思是要用这个提供者你只需要指定文件的路径和名字到 `currencyConfig`

这里有一个 solr 自带的示例 `currency.xml` 文件，和 `shema.xml` 文件在同一个目录，下面是该文件的一小段

```
<currencyConfig version="1.0">
  <rates>
    <!-- Updated from http://www.exchangerate.com/ at 2011-09-27 -->
    <rate from="USD" to="ARS" rate="4.333871" comment="ARGENTINA Peso" />
    <rate from="USD" to="AUD" rate="1.025768" comment="AUSTRALIA Dollar" />
    <rate from="USD" to="EUR" rate="0.743676" comment="European Euro" />
    <rate from="USD" to="CAD" rate="1.030815" comment="CANADA Dollar" />

    <!-- Cross-rates for some common currencies -->
    <rate from="EUR" to="GBP" rate="0.869914" />
    <rate from="EUR" to="NOK" rate="7.800095" />
    <rate from="GBP" to="NOK" rate="8.966508" />

    <!-- Asymmetrical rates -->
    <rate from="EUR" to="USD" rate="0.5" />
  </rates>
</currencyConfig>
```

OpenExchangeRatesOrgProvider

你可以配置 solr 从 OpenExchangeRates.org 下载汇率，可以每小时更新美元和 170 种货币的汇率。

这个案例里，你需要指定 `providerClass` ，并注册以获得一个 API key，如下所示

```
<fieldType
  name="currency"
  class="solr.CurrencyField"
  precisionStep="8"
  providerClass="solr.OpenExchangeRatesOrgProvider"
  refreshInterval="60"
  ratesFileLocation="http://www.openexchangerates.org/api/latest.json?app_id=yourPerso
nalAppIdKey"/>
```

`refreshInterval` 以分钟为单位，所以上面例子会每隔 60 分钟下载最新的汇率。这个刷新间隔可以加大，但不能减少。

日期

日期格式化

solr 的 日期字段 (`TrieDateField` 和 `DateRangeField`) 表现为一个毫秒精度的时间点。其所用格式是 XML Schema specification(ISO-8601 的一个有限子集)的 `dateTime` 的权威表述的有限格式(这一段大概意思就是并不支持各种日期格式，只支持有限的一种或几种)。如果熟悉 Java 8，solr 使用 DateTimeFormatter.ISO_INSTANT 来格式化和解析。

```
YYYY-MM-DDThh:mm:ssZ
```

- `YYYY` 年
- `MM` 月
- `DD` 月份里的日期
- `hh` 24小时制的小时
- `mm` 分钟
- `ss` 秒
- `Z` 就是字符 'Z'，表示这个字符串用 UTC 表示日期

注意，不能指定时区，表示时间的字符串总是以 UTC 来表示，示例如下

```
1972-05-20T17:33:18Z
```

如果你愿意，还可以包含秒的小数，但是任何超出毫秒的精度都会被忽略。示例如下

- `1972-05-20T17:33:18.772Z`
- `1972-05-20T17:33:18.77Z`
- `1972-05-20T17:33:18.7Z`

在 0000 年之前的日期，必须以 `'-'` 开头，`'+'` 开头表示 9999 年以后。0000 年 视为 公元前 1 年(year 1 BC)，没有 公元前/后 0 年这种事。

> 查询时可能要转义
>
> 如你所见，日期格式包含了冒号来分隔时，分和秒。由于对于 solr 大多数的查询解析器来说，冒号是个特殊字符，所以有时候需要转义
>
> 这是个无效查询
>
> `datefield:1972-05-20T17:33:18.772Z`
>
> 这些是有效查询
>
> `datefield:1972-05-20T17\:33\:18.772Z`
>
> `datefield:"1972-05-20T17:33:18.772Z"`
>
> `datefield:[1972-05-20T17:33:18.772 TO *]`

日期范围格式化

solr 的 `DateRangeField` 支持前面所述的时刻语法(和下面要讲的数学日期一起)和日期范围表示。一个例子是截断日期，表示整个日期跨度。其他例子是范围语法(`[ TO ]`)，这里有些例子

- `2000-11` – 2000 年 整个 11 月
- `2000-11T13` – 同上，不过是当日 13 时(下午 1-2 时)
- `-0009` – 公元前 10 年。0 年，是公元 0 年，也是公元前 1 年
- `[2000-11-01 TO 2014-12-01]` – 不解释(霸气~)
- `[2014 TO 2014-12-01]` – 从 2014 到 2014-12-01
- `[* TO 2014-12-01]` – 从可表示的最早时间到 2014-12-01

局限：范围语法不支持内嵌数学日期。如果你指定一个用数学日期截断的 `TrieDateField` 日期实例，如 `NOW/DAY` ，你得到的依然是当天的第一毫秒，而不是整天的范围。开区间范围(使用 `{` 和 `}`)在查询时可用，但不能用来做范围索引。

数学日期

solr 日期字段类型也支持数学日期表达式，这使得创建一个相对于某确定时刻的时间更容易了，包括用特殊的值 " `NOW` " 表示当前时间。

数学日期语法

数学日期表达式不是以特定单位增加时间，就是以特定单位对当前时间取整。表达式可以是链式的，从左至右执行

例如：从现在到 2 个月后的时刻

```
NOW+2MONTHS
```

一天以前

```
NOW-1DAY
```

斜杠表示取整，下面表示当前小时的开始

```
NOW/HOUR
```

下面例子计算(毫秒精度) 6 个月 3 天后并按天取整，即那天开始的时刻

```
NOW+6MONTHS+3DAYS/DAY
```

注意，数学日期最常和 NOW 一起用，与此同时，也可以和任何确定的时刻一起

```
1972-05-20T17:33:18.772Z+6MONTHS+3DAYS/DAY
```

影响数学日期的请求参数

NOW

在一个分布式请求里，solr 内部使用 `NOW` 参数来确保数学日期表达式跨越多个节点解析的一致性。但是，它也可以被一个任意的时刻(过去或未来)覆盖。

如果要设定 `NOW` 的值，必须为从 epoch(1970年1月1日00:00:00 UTC) 开始的毫秒数

示例

```
q=solr&fq=start_date:[* TO NOW]&NOW=1384387200000
```

TZ

默认情况下，所有数学日期表达式都被认为是相对于 UTC 时区，但是，可以设定 `TZ` 参数来覆盖默认行为。

例如，下面的请求在 UTC 当前月的每一天使用范围 faceting

```
http://localhost:8983/solr/my_collection/select?\
q=*:*&facet.range=my_date_field&facet=true&facet.range.start=NOW/MONTH&\
facet.range.end=NOW/MONTH%2B1MONTH&facet.range.gap=%2B1DAY
```

```
<int name="2013-11-01T00:00:00Z">0</int>
<int name="2013-11-02T00:00:00Z">0</int>
<int name="2013-11-03T00:00:00Z">0</int>
<int name="2013-11-04T00:00:00Z">0</int>
<int name="2013-11-05T00:00:00Z">0</int>
<int name="2013-11-06T00:00:00Z">0</int>
<int name="2013-11-07T00:00:00Z">0</int>
...
```

下面这个例子，使用指定的时区

```
http://localhost:8983/solr/my_collection/select?\
q=*:*&facet.range=my_date_field&facet=true&facet.range.start=NOW/MONTH&\
facet.range.end=NOW/MONTH%2B1MONTH&facet.range.gap=%2B1DAY&TZ=America/Los_Angeles
```

```
<int name="2013-11-01T07:00:00Z">0</int>
<int name="2013-11-02T07:00:00Z">0</int>
<int name="2013-11-03T07:00:00Z">0</int>
<int name="2013-11-04T08:00:00Z">0</int>
<int name="2013-11-05T08:00:00Z">0</int>
<int name="2013-11-06T08:00:00Z">0</int>
<int name="2013-11-07T08:00:00Z">0</int>
...
```

更多 DataRangeField 细节

几乎所有使用 `TrieDateField` 的地方都可以随时用 `DateRangeFiled` 替换。唯一的不同是 solr 的 xml 或 solrj 里显示其存储的数据格式为字符串而不是日期型。索引的数据会大一点。按时间单位秒对齐的查询要比 TrieDateField 更快，尤其是用 UTC。但主要的是，正如其名，允许按日期范围索引。为此，只需要按上面格式提交字符串。它还支持在索引数据和查询范围间的 3 种不同的断言：`Intersects` (默认)，`Contains`，`Within`。可以在查询里用本地参数 `op` 来指定断言，如下

```
fq={!field f=dateRange op=Contains}[2013 TO 2018]
```

这个例子里，会查找索引范围包含(或等于) 2013 到 2018 的文档。

DateRangeFiled 示例和其他信息，参阅 solr's community wiki

枚举字段

枚举字段类型可以定义一个取值范围是一个闭集的字段，且值的排序顺序是预定义好的，而非按字母或数字排序。例如，严重性列表，风险列表。

在 schema.xml 里定义枚举字段

枚举字段类型的定义很简单，如下面的例子，定义了严重级别和风险级别的枚举

```
<fieldType name="priorityLevel" class="solr.EnumField"
  enumsConfig="enumsConfig.xml" enumName="priority"/>
<fieldType name="riskLevel" class="solr.EnumField"
  enumsConfig="enumsConfig.xml" enumName="risk" />
```

除了 `name` 和 `class`，这个类型还有 2 个额外的参数

- `enumsConfig`：包含了该字段类型要用到的 `<enum/>` 字段值列表及其顺序的配置文件名。如果没有指定路径，这个文件应该在 collection 的 `conf` 目录
- `enumName`：枚举的名字，在 `enumsConfig` 文件里要用。

枚举字段配置文件

`enumConfig` 参数指定的文件，可以包含多个枚举值列表，各有不同的名字，可用于你的 solr schema 有多个枚举字段类型。

这个例子里，定义了 2 个枚举值列表，每个都在 `enum` 标签内

```
<?xml version="1.0" ?>
<enumsConfig>
  <enum name="priority">
    <value>Not Available</value>
    <value>Low</value>
    <value>Medium</value>
    <value>High</value>
    <value>Urgent</value>
  </enum>

  <enum name="risk">
    <value>Unknown</value>
    <value>Very Low</value>
    <value>Low</value>
    <value>Medium</value>
    <value>High</value>
    <value>Critical</value>
  </enum>
</enumsConfig>
```

> 修改枚举值
>
> 除非重建索引，不能修改枚举值的顺序，或移除在 `<enum/>` 里的值，也不能在末尾添加新的值。

外部文件及处理

字段属性用例

这是一个公共用例的概要，以及字段或字段类型要支持该用例所需要的属性。表格里的 true 或 false 表示为了功能正确选项必须设为给定的值。如果输入未提供，则属性值对用例没有影响。

用例	**indexed**	**stored**	**multiValued**	**omitNorms**	**termVectors**
search within field	true				
retrieve contents		true			
use as unique key	true		false		
sort on field	true(7)		false	true(1)	
use field boosts				false	
document boosts affect searches within field				false	
highlighting	true(4)	true			true(2)
facting(5)	true(7)				
add multiple values,maintaining order			true		
field length affects doc score				false	
MoreLikeThis					true(6)

注意

1. 推荐，但不是必须的
2. 如果提供了就会使用，非必须
3. if termVectors=true
4. 该字段必须定义 tokenizer，但不必是 indexed 的
5. faceting，参考 `Understanding Analyzers, Tokenizers, and Filters`
6. 这里 Term vectors 不是强制的。如果不是 true，那么一个 stored 字段被分析。所以，term vector 是建议的，但仅当 stored=false 时才是必须的
7. indexed 和 docValues 都必须是 true，但都不是必须的。DocValues 在很多场景更有效

定义字段

字段在 `schema.xml` 的 `field` 元素里定义。定义好字段类型后，定义字段很简单。

例子

下面例子定义了一个名为 `price` ，类型名为 `float` ，默认值为 `0.0` 的字段， `indexed` 和 `stored` 属性明确设定为 `true` ，同时，其他的属性都继承自 `float` 字段类型

```
<field name="price" type="float" default="0.0" indexed="true" stored="true"/>
```

字段属性

属性	描述
name	字段名。应该是字母数字下划线组成，不能以数字开头。名字前后都是下划线的(如 `_version_`)为保留字。每个字段都要有名字
type	这个字段的类型名。是在 `fieldType` 里定义的 `name` 。每个字段都要有类型名
default	索引时，如果该字段没有值，一个默认值会自动添加到文档里的该字段。如果未指定则没有默认值

可选的字段类型覆盖属性

字段可以有和很多字段类型一样的属性。下表所列属性如在字段里设定，将覆盖其在字段类型 `<fieldType/>` 里明确设定的值或默认值。

属性	描述	值	默认值
indexed	如为 true，字段值能用于查询检索匹配的文档	true/false	true
stored	如为 true，字段值可以被查询获取	true/false	true
docValues	如为 true，字段值将放入面向列的 `DocValues` 结构	true/false	false
sortMissingFirst sortMissingLast	当一个索引字段未提供时，控制文档(在索引)的位置(是在最前/最后)	true/false	false
multiValued	如为 true，单个文档在这个字段可能包含多个值	true/false	false
omitNorms	如为 true，省略与此字段关联的加权基准(这会关闭长度标准化，及索引时对该字段的加权，从而节省内存占用)。对所有原始(无需解析)的字段类型，诸如 int，float，date，bool，和 string，默认为 true。只有全文本字段或需要索引时加权的字段才需要加权基准(norms)	true/false	*
omitTermFreqAndPositions	如为 true，在提交字段时省略词条的频率/次数，位置，载荷。对于不需要这些信息的字段能提升性能，同时还减少了索引需要的存储空间。依赖位置信息的查询会无声的失败。对于所有非文本字段这个属性默认为 true	true/false	*
omitPositions	和 `omitTermFreqAndPositions` 类似，但是会保留词条频率/次数信息	true/false	*
termVectors termPositions termOffsets termPayloads	这些选项命令 solr 保持每个文档完整的词条向量，随意的包括在那些向量里每个词条的位置，偏移，及载荷信息。这些可用于加快高亮和其他辅助功能，但会极大增加索引的尺寸。在 solr 的典型应用里是没有必要的	true/false	false
required	如为 true，solr 拒绝添加一个该字段没有值的文档	true/false	false
useDocValuesAsStored	如果该字段开启了 `docValues`，设为 true 将允许该字段在 `fl=*` 时像一个存储的字段一样返回(即使该字段的 `stored=false`)	true/false	true

复制字段

你可能想要用多种方式诠释文档里的某些字段。solr 有一种创建字段副本的机制，这样你就能对单一的输入信息应用多种不同的字段类型了。

你要复制的字段是 源*(source)*，副本是 目标*(destination)*。在 `schema.xml` 里创建字段副本很简单：

```
<copyField source="cat" dest="text" maxChars="30000" />
```

这个例子里，我们想要 solr 把 `cat` 字段复制到 `text` 字段。字段在解析完成前复制，这意味着你可以有 2 个来源内容完全一样的字段，使用不同的解析流程，保存在不同的索引里。

上面例子里，如果目标字段 `text` 在输入文档里有其数据，`cat` 字段的内容将作为附加值添加 - 正如所有的值最初都是来自于客户端指定。不要忘记配置你的字段为 `multivalued="true"`，如果他们最终会有多个值(要么是来自于一个多值的源，要么是来自于多个 `copyField` 配置)

这个功能一个通常的用途是创建单个搜索字段作为默认的查询字段，当用户或客户端未在查询里指定字段时。例如，`title`，`author`，`keyword`，和 `body` 可能是默认都应被搜索的字段，将每个字段都复制到一个 `catchall` 字段(仅作示例，可以用任何名字)。稍后在 `solrconfig.xml` 设定默认搜索字段为 `catchall`。一个警告是使用复制字段索引大小的增长。对于你，还有最终的大小来说，这是否一个问题，取决于要被复制的源字段数量，目标字段的数量，要使用的解析流程，以及可用的磁盘空间。

`maxChars` 参数，`int` 类型，设定了从源字段复制到目标字段的字符数上限。这个限制对于想要从源字段复制数据又想控制索引文件大小的场景有用。

`copyField` 的源字段和目标字段，都可以以星号(`*`)开头或结尾，表示匹配任意字符。例如，下面这行将复制所有匹配 `*_t` 的字段内容到 `text` 字段

```
<copyField source="*_t" dest="text" maxChars="25000" />
```

> `copyField` 命令仅当 `source` 参数也有一个通配符(`*`)时，可以在 `dest` 参数使用通配符(`*`)。

动态字段

动态字段允许 solr 索引那些你没有在 schema 里明确定义的字段。如果你忘记定义某些字段，这就有用了。动态字段通过提升添加文档到 solr 的灵活性，使你的应用更健壮。

动态字段除了名字里有通配符以外，和标准字段一样。索引文档时，不匹配任何明确定义字段的字段可匹配动态字段。

例如，假设你的 schema 包含了名为 `*_i` 的动态字段，如果你想索引一个有 `cost_i` 字段的文档，但没有在 schema 里明确定义 `cost_i` 字段，那么这个 `cost_i` 字段将作为 `*_i` 字段解析

和标准字段一样，动态字段有 名字，字段类，和其他的选项

```
<dynamicField name="*_i" type="int" indexed="true" stored="true"/>
```

推荐在你的 `schema.xml` 里包含基本的动态字段映射(如上所示)，这将会很有用。

其他 schema 元素

Unique Key 唯一键

`uniqueKey` 元素指示文档的唯一标识是哪个字段。虽然 `uniqueKey` 不是必须的，但几乎是你的应用程序设计中总是要保证的。例如，如果想要更新索引里的文档应该用到 `uniqueKey` 定义

```
<uniqueKey>id</uniqueKey>
```

`copyField` 不能用于 `uniqueKey` 字段。也不能用 `UUIDUpdateProcessorFactory` 自动生成 `uniqueKey` 值

如果字段是多值的或继承自多值字段类型，用作 `uniqueKey` 字段会导致操作失败。但是，`uniqueKey` 会继续工作，只要该字段被正确的使用。

默认的搜索字段和查询操作

虽然已不建议使用并会在某个时候弃用，solr 仍然支持在 schema 配置 `<defaultSearchField/>` (被 `df` 参数代替) 和 `solrQueryParserDefaultOperator="OR"/>` (被 `q.op` 参数代替)

如果你在 schema 设定了这些选项，强烈推荐用请求参数(或默认请求参数)替换它们，因为未来的 solr 发行版本可能会移除。

Similarity 相似性

Similarity 是搜索时给文档评分的 Lucene 类。

每个 collection 都有一个全局的 similarity，默认情况下，solr 使用自带的 `SchemaSimilarityFactory`，允许特定字段类型配置其特定的 Similarity，并为所有未明确指定 Similarity 的字段类型使用 `BM25Similarity`。

在 `schema.xml` 里，任何字段类型的定义之外，声明一个顶级的 `<similarity/>` 元素，可覆盖默认行为。这个 similarity 声明要么直接引用一个有无参构造函数的类，例如下面的例子

```
<similarity class="solr.BM25Similarity"/>
```

要么引用一个 `SimilarityFactory` 实现，接受可选的初始化参数

```
<similarity class="solr.DFRSimilarityFactory">
  <str name="basicModel">P</str>
  <str name="afterEffect">L</str>
  <str name="normalization">H2</str>
  <float name="c">7</float>
</similarity>
```

大多数场景下，如果你的 `schema.xml` 包含了字段类型里的 `<similarity/>` 声明，如上面那样指定全局的 similarity 会导致错误。一个关键的异常就是你可能明确声明了 `SchemaSimilarityFactory`，并设定所有字段类型的默认行为，但是这个默认行为没有声明为一个明确的 similarity，即已配置了 similarity 的字段类型的名字(用 `defaultSimFromFieldType` 设定)(这段话很绕，意思就是，如果你已经在字段类型里配置了 similarity，要想配置一个全局的 similarity，很可能会出错。这是因为你的全局 similarity 是用的 `SchemaSimilarityFactory`，但是你没有配置这个 factory 的 `defaultSimFromFieldType` 为那些个已配置了 similarity 字段的其中之一的名字)

```
<similarity class="solr.SchemaSimilarityFactory">
  <str name="defaultSimFromFieldType">text_dfr</str>
<similarity>
<fieldType name="text_dfr" class="solr.TextField">
  <analyzer ... />
  <similarity class="solr.DFRSimilarityFactory">
    <str name="basicModel">I(F)</str>
    <str name="afterEffect">B</str>
    <str name="normalization">H3</str>
    <float name="mu">900</float>
  </similarity>
</fieldType>
<fieldType name="text_ib">
  <analyzer ... />
  <similarity class="solr.IBSimilarityFactory">
    <str name="distribution">SPL</str>
    <str name="lambda">DF</str>
    <str name="normalization">H2</str>
  </similarity>
</fieldType>
<fieldType name="text_other">
  <analyzer ... />
</fieldType>
```

上面例子里，`IBSimilarityFactory`(使用基于信息的模式)将被用于类型为 `text_ib` 的所有字段，同时 `DFRSimilarityFactory`(随机偏离)将既用于类型为 `ext_dfr` 的所有字段，又用于任何未在字段类型里设定 `<similarity/>` 的字段(即，字段类型为 `text_other` 的字段，和字段类型为 `text_+dfr` 的字段一样使用 `IBSimilarityFactory`)

如果 `SchemaSimilarityFactory` 明确声明，且没有配置 `defaultSimFromFieldType` ，那么 `BM25Similarity` 会用作默认值。

除了这里提到的几种 factory 以外，还有一些其他 Similarity 实现，诸如 `SweetSpotSimilarityFactory` ， `ClassicSimilarityFactory` 等等，可以使用，详细信息，参阅 solr 的 similarity factory java 文档

搜索

查询语法及解析

普通查询参数

参数	说明
defType	默认=lucene（标准查询解析器），用于指定查询解析器
sort	返回结果如何排序：字段名 asc或desc
start	默认=0，返回结果集的起始位置
rows	默认=10，返回多少行
fq	针对查询结果进行过滤的查询
fl	返回哪些字段，逗号或空格分隔多个字段名
debug	
explainOther	
timeAllowed	
segmentTerminateEarly	
omitHeader	默认=false，返回结果是否忽略header
wt	返回结果的格式，例如：csv，json，php，xml，.....
logParamsList	
echoParams	-

标准查询解析器

solr 默认查询解析器为 “lucene” 解析器

参数

参数	说明
q	使用标准查询语法的查询语句，该参数为必须的
q.op	查询表达式默认操作，取值为 AND , OR，取代 solrconfig.xml里的配置
df	默认字段， 取代 solrconfig.xml里的配置

响应

略

规格化词条

标准查询解析器把查询分解为词条和操作符，有 2 种词条：单个词条和短语

- 单个词条：单个的单词，例如 "test"，"hello"
- 短语：双引号包围的多个单词，例如 "hello dolly"

多个词条可以被逻辑操作符连接起来组成复杂查询

> 重要的是，查询和索引时使用相同的分析器来分解词条；否则，搜索时就可能产生意料之外的结果

词条编辑器

通配符搜索

类型	通配符	示例
匹配单个字符	?	te?t，匹配 test 和 text
匹配多个字符	*	tes*，匹配 test testing tester

模糊搜索

solr 标准查询解析器支持模糊搜索，基于 Damerau-Levenshtein Distance 或 Edit Distance 算法。要执行模糊搜索，在单个词条末尾使用波浪符~，示例如下

> roam~ 匹配 roams，foam，foams，也匹配 roam 本身

还可以附带一个距离参数在波浪符后面，取值为 0~2，默认=2，示例如下

> roam~1 匹配 roams，foam，但是不匹配 foams，除非把距离修改为 2

临近搜索

临近搜索查找那些间隔一定距离的词条

要执行临近搜索，使用波浪符 ~ ，并在要搜索的短语末尾添加一个数字，示例如下

> "jakarta apache"~10，表示在文档里搜索 apache 和 jakarta，互相之间间隔 10 个单词

范围搜索

示例

> mod_date:[20020101 TO 20030101]

范围查询不限于日期或数字，字符串也支持，此时按字典数序，示例：

> title:{Aida TO Carmen}

范围区间

- {x TO y}，表示 x 与 y 之间，不包含 x 和 y
- [x TO y]，表示 x 与 y 之间，且包含 x 和 y
- [x TO y}，也是可以滴

注意：* 号可以表示无限，在范围区间的两侧都可以使用

词条加权搜索

使用插入符号 ^ 来指定词条的相关性权重，在插入符后的数字是权重因子，例如：搜索 "jarkarta apache"，想要 "jakarta" 有更多的相关性，可以提高它的权重，如下

> jakarta^4 apache

也可以为短语加权，如下

> "jakarta apache"^4 "jakarta lucene"

默认的权重因子=1，虽然权重因子必须是正数，但是可以小于1，例如0.2

固定分数搜索

相同的查询条件，在不同的文档里有着不同的匹配度，该匹配度用分数表示。固定分数查询可以使一个查询对于任何匹配的文档都有一个固定的分数，格式如下

> <查询语句>^=<分数>

如果你仅需要匹配词条，而不关注匹配的词条在文档里出现的频率\/次数及词条在文档所有词条所占比例（这也是影响匹配度的因子），就可以使用固定分数查询。示例

> (description:blue OR color:blue)^=1.0 text:shoes

注意，匹配度由下述情况来决定

- 文档本身的权重，越大分数越大
- 字段的权重，越大分数越大
- 文档里的词条数量，越多分数越小
- 匹配的词条在文档里出现了多少次，越多分数越高
- 匹配的词条数目占文档总词条数目的比例，比例越大分数越高

规格化字段

数据在 solr 里是按字段进行索引的。搜索时可以指定字段进行搜索。在 schema 里定义了默认的字段，如果搜索时未指定字段，则会在默认字段里进行搜索

示例

> title:"The Right Way" AND text:go

如果 text 字段是默认字段，也可以这样

> title:"Do it right" AND go

字段只对第一个词条生效，后续的词条将在默认字段里搜索，如下

> title:Do it right，等同于 title:Do OR text: it right

逻辑操作符

逻辑操作符	符号	说明
AND	&&	操作符 2 边所有词条都要在匹配中出现
NOT	!	后续词条不得出现
OR	||	操作符 2 边任意词条在匹配中出现
	+	后续词条需要出现
	-	禁止后续词条，即匹配的字段或文档里木有这个词条。和 ! 操作符是一样一样滴

注意

- 逻辑操作符全部大写
- 标准查询解析器支持所有逻辑操作符：AND，NOT，OR，+，-；DisMax 查询解析器只支持 +，-
- OR 操作符是默认的连接符号，这意味着如果 2 个词条之间木有逻辑操作符，那就表示有个隐含的 OR 操作符

特殊字符转义

如下字符出现在查询中时是有特别含义滴，若要查询字符本身需要转义

```
&& \|\| ! \( \) { } \[ \] ^ " ~ \* ? : \/
```

转义符为反斜杠 \，查询 (1+1):2示例如下

```
\\(1+1\\)\:2
```

组合词条成子查询

solr 支持用圆括号来组合查询，示例如下

```
\(jakarta OR apache\) AND website
```

表示查询须匹配 website，且 jakarta 或 apache 至少一个

针对字段的组合查询

在单个字段上使用多个逻辑操作符，用圆括号来组合，示例

```
title:\(+return +"pink panther"\)
```

注释

solr 支持 c 语言样式的注释，即 \/* 注释 *\/。注释可以内嵌在查询中，示例

```
"jakarta apache" \/\* this is a comment in the middle of a normal query string \*\/ OR jakarta
```

Lucene 查询解析器 和 solr 标准查询解析器差异

solr 标准查询解析器在如下方面和 lucene 查询解析器不同

- 范围查询的 * 号
  - `field:\[\* TO 100\]` 表示所有小于等于 100 的值
  - `field:\[100 TO \*\]` 表示所有大于等于 100 的值
  - `field:\[\* TO \*\]` 表示所有值
- 在最顶层查询中，支持纯的负查询（所有条件都是禁止的查询）
  - `-inStock:false` 表示所有 inStock 不为 false 的值
  - `-field:\[\* TO \*\]` 表示 field 值为空的
- FunctionQuery 语法挂钩：需要用引号将带圆括号的函数包围，如下第二例
  - `\_val\_:myfield`
  - `\_val\_:"recip\(rord\(myfield\),1,2,3\)"`
- 内嵌的查询语句支持任意类型的查询解析器
  - `inStock:true OR {!dismax qf='name manu' v='ipod'}`
- 支持特殊的 filter(...) 语法来指明某些查询子句应该被缓存到过滤器缓存(filter cache)，例如下面 3 个例子里的 inStock:true 会被缓存和重用

  `q=features:songs OR filter\(inStock:true\)`

  `q=+manu:Apple +filter\(inStock:true\)`

  `q=+manu:Apple & fq=inStock:true`
- 范围查询 ("[a TO z]")，前缀查询 ("a*")，通配符查询 ("a*b") 都是固定分数的，即所有匹配的文档都有相同的分数

规格化日期和时间

字段类型为 TrieDateField 的，可以使用如下的语法

- `timestamp:\[\* TO NOW\]`
- `createdate:\[1976-03-06T23:59:59.999Z TO \*\]`
- `createdate:\[1995-12-31T23:59:59.999Z TO 2007-03-06T00:00:00Z\]`
- `pubdate:\[NOW-1YEAR\/DAY TO NOW\/DAY+1DAY\]`
- `createdate:\[1976-03-06T23:59:59.999Z TO 1976-03-06T23:59:59.999Z+1YEAR\]`
- `createdate:\[1976-03-06T23:59:59.999Z\/YEAR TO 1976-03-06T23:59:59.999Z\]`

DisMax 查询解析器

DisMax 查询解析器被设计来处理简单短语和多个不同权重的字段。

通常，DisMax 查询解析器更像 google 而不是标准查询解析器。

DisMax 支持 Lucene 查询解析器语法的精简子集

好奇 DisMax 这个名字的来源？DisMax 代表 Maximum Disjunction。不论是否能记住这个名字的含义，只要记住 DisMax 是易于使用，能接受任何输入且不会出错的。

参数

参数	说明
q	查询的原始输入
q.alt	如果 q 参数未使用，这个参数用来呼叫标准查询解析器解析输入
qf	query fields，指定执行查询时所需要的索引字段，如果未指定则为 df 定义的字段
mm	最小匹配数，指定查询最少要匹配几个条件，如果没有指定该参数，则依赖于 q.op 参数（不论该参数是在查询条件里指定，还是在 solrconfig.xml 里指定的默认值，或 schema.xml 里通过 defaultOperator 选项指定）：q.op = AND 则 mm = 100%；q.op = OR 则 mm=1。用户也可以在 solrconfig.xml 里指定 mm 的默认值。该参数还允许在表达式里混入空格，例如 " 3 < -25% 10 < -3\n", " \n-25%\n", " \n3\n "
pf	phrase fields：增加文档的分数以防 q 参数的所有词条出现在附近 （神马意思?）
ps	phrase slop：指定 2 个词条分开的位置数，以匹配一个特定的短语
qs	query phrase slop：含义同 ps，一般和 qf 参数一起使用
tie	Tie Breaker：一个远小于 1 的浮点值，在 DisMax 查询里作为决定性因素 （这 tmd 又是神马意思？）
bq	boost query：指定一个因子，对于匹配时需要加强重要性的那些词条或短语
bf	Boost Functions：指定一个函数来加权

qf

字段列表，每个字段都有一个权重因子来增强或减弱该字段在查询中的重要性，示例

```
qf="fieldOne^2.3 fieldTwo fieldThree^0.4"
```

上面例子表示 fieldOne 的权重因子为2.3，fieldTwo 为默认的权重因子，fieldThree 的权重因子为0.4mm

mm

Lucene/Solr 在处理查询时，有 3 类子句：强制的，禁止的，可选的。默认情况下，q 参数里的所有词条或短语都被作为可选的，除非前面有 + 或者 -。在处理可选的子句时，mm 参数表示最少有多少可选的子句要被匹配，DisMax 查询解析器对于如何指定 mm 提供了极大的灵活性

语法	示例	说明
正整数	3	最少有几个子句需要匹配，不论总共有多少子句
负整数	-2	从全部子句里减去多少个子句
百分比	75%	全部子句总数的百分比，向下取整
负百分比	-25%	全部子句里可以不用匹配的子句所占的百分比，向下取整
表达式：正整数后跟>或<和另一个值	3<90%	如果全部子句数量等于或小于该正整数，那么就都是强制的；但是如果是大于这个正整数，那么就只需要满足百分比
多项表达式，包含>或<	2<-25% 9<-3	多个表达式，任一个要生效，必须是其数字比前一个的数字更大。左边的例子里，如果有 1~2 个子句，则都是强制的；如果有 3~9 个子句，则 -25% 是强制的；如果超过 9 个子句，则 -3 是必须的

默认的 mm=100%

pf

格式如同 qf，如果该字段匹配短语而非几个词条，那么会导致加权因子生效。

例如，查询 q=hello world，那么 "hello world" 显然比 "hello this is a world" 更匹配，因为其不仅匹配了 hello 和 world，还同时将输入作为单个短语进行了匹配

ps

貌似是对 pf 里的短语指定一个“间距”，如果在这个间距内也算是短语

tie

如果一个查询在多个字段都匹配，tie 可以用来决定最终的得分，最终得分的计算公式为

```
分数最大值 + tie * (其他分数)
```

示例如下

两个 field: A 和 B，query 分别在 A 和 B 中都能命中，其得分如下：

文档	字段**A**得分	字段**B**得分	最大得分
doc1	0.5	0.8	0.8
doc2	0.8	0.1	0.8

这时候两个 doc 在不同 field 上的 max 得分相同，但是我们其实更希望 doc1 得分更高，那么 tie 就可以派上用场了

假设 tie = 0.1，则各自的最终分为

```
doc1 = 0.8 + 0.5 * 0.1 = 0.85
```

```
doc2 = 0.8 + 0.1 * 0.1 = 0.81
```

doc1 胜出

可见，若 tie = 0，则就是分数最大的字段的得分作为最终得分，若 tie = 1，则就是将所有字段得分简单相加作为最终得分；一般设定 tie = 0.1

bq

bq 参数指定了一个附加的、可选的查询子句，会附加到用户的主查询中以影响结果的分数；可以使用多个 bq 参数

bf

同 bq，不过 bf 是指定一个函数查询，且可以附带一个可选的权重因子，可以使用任何 solr 支持的函数查询，示例

```
recip(rord(myfield),1,2,3)^1.5
```

bf 的函数本质上是 bq 参数组合了 {!func} 的一个简写，例如，如果想要最近的文档排在前面，如下的查询是一样一样滴

```
bf=recip(rord(creationDate),1,1000,1000)  bq=
{!func}recip(rord(creationDate),1,1000,1000)
```

扩展的 **DisMax** 查询解析器

扩展的 DisMax（eDisMax）是一个 DisMax 的优化版本，除了支持 DisMax 的所有参数之外，eDisMax

- 支持完整的 Lucene 查询语法
- 支持 AND, OR, NOT, +, -
- Lucene 语法模式下，and, or 被视同为 AND, OR
- 支持魔法字段 `_val_`，`_query_`。这2个字段不是在 schema 里定义的真实的字段，但是如果使用的话，可以用来干一些特别的事：

函数查询

函数查询允许用一个或多个数值字段生成一个相关度的分数，标准查询解析器、DisMax、eDisMax都支持函数查询

函数查询使用函数，函数可以是一个常数（数字或字符串），一个字段，另一个函数，或参数替换参数（不懂...），可以用函数来修改结果的排名(Ranking)，可以用于对于根据用户的位置，或其他的考虑，来改变结果的排名

使用函数查询

函数表现为函数调用，例如 `sum(a,b)` 而不是 `a+b`

在 solr 查询里，有几种使用函数查询的方法

- 通过一个明确的、期待函数参数的 QParser，如 `func` 或 `frange`，例如
  - `q={!func}div(popularity,price)&fq={!frange l=1000}customer_ratings`
- 在一个排序表达式中，例如
  - `sort=div(popularity,price) desc, score desc`
- 将函数结果值作为查询结果文档的伪字段（区别与真实字段），例如
  - `&fl=sum(x, y),id,a,b,c,score`
- 明确需要函数的参数，如 eDisMax 的 `boost` 参数，或 DisMax 的 `bf` 参数（注意 bf 参数接收多个以空格分隔的函数查询，每个函数查询有一个可选的 boost 因子，确保在单个函数内的空格被排除），例如
  - `q=dismax&bf="ord(popularity)^0.5 recip(rord(price),1,1000,1000)^0.3"`
- 使用 Lucene QParser 的 _val_ 关键字引入函数查询，例如
  - `q=_val_:mynumericfield _val_:"recip(rord(myfield),1,2,3)"`

只推荐可快速随机存取的函数（？）

可用的函数

函数	说明	语法示例
abs	绝对值	`abs(x)` `abs(-5)`
and	仅当所有操作数都为 true 时返回 true	`and(not(exists(popularity)),exists(price` 任何文档，字段 price 有值，且字段 popu 有值时返回 true
"constant"	浮点常数	1.5

def	default 的简写，若值 1 不存在则返回值 2	`def(rating,5)` ：如果 rating 存在则返回之 返回 5 `def(myfield, 1.0)` ：如果字段 myfield 存 值，否则返回 1.0
div	除法，div(x,y) 表示 x 除以 y	`div(1,y)` `div(sum(x,100),max(y,1))`
dist	返回 n 维空间里 2 个矢量的距离	`dist(2, x, y, 0, 0)` ：(0,0) 和 (x,y) 之间 离 `dist(1, x, y, 0, 0)`：(0,0) 和 (x,y) 之间的 距离 `dist(2, x,y,z,0,0,0)` ：(0,0,0) 和 (x,y,z) 氏距离 `dist(1,x,y,z,e,f,g)`：(x,y,z) 和 (e,f,g) 之 顿距离，每个字符是一个字段名
docfreq(field,val)	返回这个字段里包含这个值的文档数量，这是个常数	`docfreq(text,'solr')` `defType=func&q=docfreq(text,$myterm)&myt`
exists	如果任意字段存在就返回 true	`exists(author)` 任何文档在 author 字段有 回 true `exists(query(price:5.00))` price 匹配 5.0 true

field	返回字段的 docValues 或 indexed 值（数量？），对于 docValues，可以添加可选参数 min 或 max，返回 0 表示文档木有该字段	如下示例是等价的
		`myFloatFieldName`
		`field(myFloatFieldName)`
		`field("myFloatFieldName")`
		如果字段名是非典型的（例如包含了空格）种写法
		对于多值的 docValues 字段，示例如下
		`field(myMultiValuedFloatField,min)`
		`field(myMultiValuedFloatField,max)`

hsin		
idf		
if	条件判断，语法为 `if(test,value1,value2)` `test` 是个逻辑值或逻辑表达式 `value1` test = true 时返回 `values` test = false 时返回 表达式可以是任意返回逻辑值的函数，或是返回数值的函数，此时 0 表示 false，或是 string，此时空串表示 false	`if(termfreq(cat,'electronics'),popularit` 该函数检查每一文档，字段 cat 是否包含 "electronics"，如果包含，返回 popularity 值，否则返回 42
	`m*x+c` 的函数形式，m、c 为常数，x 为变量	

linear	等价于 `sum(product(m,x),c)` 但是性能好些，因为只有一次函数调用	`linear(x,m,c)` `linear(x,2,4)` 返回 `2*x+4`
log	对数，以 10 为底数	

map	根据输入是否落在某个范围来返回值，范围参数 min，max 必须是常数。如果输入 x 不在 min 和 max 之间，返回值为 x 或 default（如果指定了该参数）	`map(x,min,max,target)`
		`map(x,0,0,1)` - 将 0 改为 1，处理默认值
		`map(x,min,max,target,default)`
		`map(x,0,100,1,-1)` - 将 0~100 之间的值改
		其他值 改为 -1

max	返回最大值 `max(x,y,...)`	`max(myfield,myotherfield,0)`
maxdoc	返回索引的文档数量，包括已标记为删除但还没有物理删除的文档，这是个常数，对于索引里的任何文档都是同样的值	`maxdoc()`
min	返回最小值	

ms	返回参数之间的时间差异，以毫秒为单位，参数可以是索引的 TrieDateField 字段，或 NOW，或基于日期常数的日期计算
	`ms()` 等价于 `ms(NOW)` 当前时间与 epoch 之间的毫秒差
	epoch 1970-1-1 00:00:00.000
	`ms(a)` a 与 epoch 之间的毫秒差
	ms(a,b) : 时间 a 与 b 之间的毫秒差，注意 a 发生在 b 之后，即 a - b
		`ms(NOW/DAY)`
		`ms(2000-01-01T00:00:00Z)`
		`ms(mydatefield)`
		`ms(NOW,mydatefield)`
		`ms(mydatefield,2000-01-01T00:00:00Z)`
		`ms(datefield1,datefield2)`

norm(field)		
not	逻辑非	
numdocs	返回索引的文档数量，不包括已标记为删除但还没有物理删除的文档，这是个常数，对于索引里的任何文档都是同样的值	`numdocs()`
		`or(value1,value2)` value1 和 value2 都为

		返回 true
ord	返回索引字段的值的序号，从 1 开始。排序是依字典顺序，该字段为单值字段。若字段没有值返回 0	

pow	幂值，`pow(x,y) =x^y`	`pow(x,y)`
		`pow(x,log(y))`
		`pow(x,0.5)` 等价于 `sqrt`

product	乘积，参数为逗号分隔的值或函数，等价于 `mul(...)`	`product(x,y,...)`
query	返回指定的子查询的分数，如果子查询未匹配到文档的话返回一个默认值	`query(subquery, default)` `q=product(popularity, query({!dismaxv='s rocks'})` 返回 popularity 和 DisMax 查询的乘积 `q=product(popularity, query($qq))&qq={!dismax}solr rocks`：等价于上面的查询，参数引用 `q=product(popularity, query($qq,0.1))&qq {!dismax}solr rocks`：指定了不匹配 DisMa 文档的默认的分数为 0.1
recip	`recip(x,m,a,b) = a/(m*x+b)` m,a,b 为常数, x 是个函数	`recip(myfield,m,a,b)` `recip(rord(creationDate),1,1000,1000)`
rord	ord 的反转	`rord(myDateField)`
scale	将输入 x 缩放到 minTarget 和 maxTarget 之间，当前的实现会遍历所有的 x 值来确定 min 和 max 的值 如果文档被删除或者没有值，则会视同为0	`scale(x,minTarget,maxTarget)` `scale(x,1,2): scales the values of x suc all values will be between 1 and 2 inclus`
sqedist	欧氏距离的平方，	`sqedist(x_td, y_td, 0, 0)`
sqrt	平方根	`sqrt(x)sqrt(100)sqrt(sum(x,100))`
strdist	计算 2 个字符串的距离，格式为 `strdist(string1, string2, distance measure)`	`strdist("SOLR",id,edit)` distance measure 取值为 `jw`：Jaro-Winkler `edit`：Levenstein or Edit distance `ngram`：NGramDistance, 可以指定一个可 ngram size 参数，默认为 2 `FQN`：实现了 StringDistance 接口的类名，须有一个无参的构造函数
sub	减法，sub(x,y) 即为 x - y	`sub(myfield,myfield2)`
sum	求和，add() 可以作为别名使用	`sum(x,y,...)`

	名使用	
sumtotaltermfreq	返回 `totaltermfreq` 之和	`sttf()`
termfreq	词条在字段里出现的次数	`termfreq(text,'memory')`
tf	term frequence，返回一个给定词条的词条频率因子	`tf(text,'solr')`
top	貌似是在顶层索引里取值，同一个值在某个段的序号和其在整个索引里的序号是不一样的	`ord()` ，`rord()` 实际是隐含了 `top()` 调 `ord()` 等价于 `top(rod())`
totaltermfreq	整个索引里，词条在字段里出现的次数	`ttf(text,'memory')`
xor()	逻辑异或	`xor(field1,field2)` field1 ，field2 都为 tr 回 false，否则返回 true

有 2 个文档，doc1，doc2，都有字段 fieldX，其中

- doc1.fieldX = [A B C]
- doc2.fieldX = [A A A A]

则

- `docFreq(fieldX:A) = 2` A 在 2 个文档里出现
- `freq(doc2, fieldX:A) = 4` A 在 doc2 里出现 4 次
- `totalTermFreq(fieldX:A) = 5` A 在所有文档里出现 5 次
- `sumTotalTermFreq(fieldX) = 7` 对于 fieldX，5 个 A，1 个 B，1 个 C

函数查询示例

假设一个索引，字段名为 `boxnames` ，保存了一些名字随意，尺寸(x,y,z)以米为单位的盒子，你要搜索名字匹配 `findbox` 且按体积排序的盒子，查询参数如下

```
q=boxname:findbox _val_:"product(x,y,z)"
```

这个查询将结果按盒子的体积排序，如果想要知道体积，你需要请求分数（score，包含了体积）

```
&fl=*,score
```

假设还有一个字段保存着重量，`weight` ，要按盒子的密度排序且返回密度大小，可以提交下面的查询

用函数排序

可以用函数的输出对查询结果排序，例如，要按距离排序

```
http://localhost:8983/solr/collection_name/select?q=*:*&sort=dist(2, point1, point2) desc
```

函数排序也支持伪字段：字段是动态生成且像一个正常字段那样返回结果，例如

`&fl=id,sum(x, y),score` 将会返回

```
<str name="id">foo</str>
```

```
<float name="sum(x,y)">40</float>
```

```
<float name="score">0.343</float>
```

查询中的局部参数

局部参数是 solr 请求参数的参数（？），提供了一种添加元数据到某个参数的方法

局部参数被指定为参数前缀（？），如下例的查询

```
q= solr recks
```

可以在查询前置局部参数来为标准查询解析器提供更多的信息，例如：修改默认操作类型为 AND，默认字段为 title

```
q={!q.op=AND df=title}solr rocks
```

局部参数基本语法

要指定一个局部参数，在要修改的参数前插入

- 以 `{!` 开头
- 任意个数的键值对，即 key=value，以空格分隔
- 以 `}` 结尾，且后面紧跟原查询参数

每个参数仅可指定一个局部参数，键值对里的值可以用单引号或双引号包围，反斜线作为引号的转义符

查询类型简写

如果一个局部参数看上去只有值而没有名字，表明使用了隐含的名字 `type` 。表明使用哪种查询解析器来解析查询语句，即

`q={!dismax qf=myfield}solr rocks` 等价于 `q={!type=dismax qf=myfield}solr rocks`

如果未指定 `type` 那么 lucene parser 是默认的解析器，即

`fq={!df=summary}solr rocks` 等价于 `fq={!type=lucene df=summary}solr rocks`

用'v'键指定参数值

局部参数里的特殊键 `v` ，可以用来代替原查询参数，即

`q={!dismax qf=myfield}solr rocks` 等价于 `q={!type=dismax qf=myfield v='solr rocks'}`

参数无关

参数无关或间接让你使用另一个参数的值而不是直接引用它（？）

`q={!dismax qf=myfield}solr rocks` 等价于 `q={!type=dismax qf=myfield v=$qq}&qq=solr rocks`

其他解析器

建议

Solr 建议组件(SuggestComponet)为用户在查询词条时提供自动的建议。在你自己的应用程序使用它来实现强大的自动建议特性。

虽然使用语法检查(Spell checking)功能提供自动建议是可行的，但是 Solr 专门设计了 SuggestComponet 来实现该功能。这个方法利用了 Lucene 的建议实现且支持 Lucene 提供的所有查找实现

建议的主要特性

- 可插拔的查找实现
- 可插拔的词典，带来选择词典实现的灵活性
- 分布式支持

在 Solr 的 `techproducts` 示例里已在 `solrconfig.xml` 里配置了新的建议实现。了解更多的搜索组件，参考 `RequestHandlers and SearchComponents in SolrConfig`

在 solrconfig.xml 里配置建议

`techproducts` 示例的 `solrconfig.xml` 已配置了 `suggest` 搜索组件和一个 `/suggest` 请求处理器。以此为你的配置基础，或按下面描述的从头创建配置

添加建议搜索组件

首先是在 `solrconfig.xml` 里添加一个搜索组件，并让它使用建议组件，如下

```
<searchComponent name="suggest" class="solr.SuggestComponent">
  <lst name="suggester">
  <str name="name">mySuggester</str>
  <str name="lookupImpl">FuzzyLookupFactory</str>
  <str name="dictionaryImpl">DocumentDictionaryFactory</str>
  <str name="field">cat</str>
  <str name="weightField">price</str>
  <str name="suggestAnalyzerFieldType">string</str>
  <str name="buildOnStartup">false</str>
  </lst>
</searchComponent>
```

建议搜索组件参数

建议搜索组件有几个配置参数。查找实现(`lookupImpl` ，如何在建议词典里找到词条)和词典实现(`dictionaryImpl` ，如何在建议词典里保存词条)的选择基于几个必须的参数。不管是哪种查找或词典的实现，下面列出的是主要的参数

参数	说明
searchComponentName	搜索组件名称
name	建议的名称，可以在 URL 参数和 SearchHandler 配置里引用该名称。可以有多个建议(?)
lookupImpl	查找实现。有几个可用的实现，参考 `Lookup Implementations` 。如果未设置，默认为 JaspellLookupFactory
dictionaryImpl	词典实现。有几个可用的实现，参考 `Dictionary Implementations` 。如果未设置，默认的词典实现为 HighFrequencyDictionaryFactory，如果 `sourceLocation` 使用的话，则为 FileDictionaryFactory

field	索引里的一个字段，用来作为建议词条的基础，如果 `sourceLocation` 为空(意味着 FileDictionaryFactory 之外的词典实现)那么这个字段里索引的词条被使用。
	要作为建议的基础，这个字段必须是存储的。你也许想要用 copyField 来创建一个包含文档里其他字段的建议字段。无论如何，你希望该字段的分析最小化，所以，一个额外的选项是在你的 schema 创建一个只使用基本的分词器或过滤器的字段，如下面的一个字段
	
	`<fieldType class="solr.TextField" name="textSuggest" positionIncrementGap="100">`
	`<analyzer>`
	`<tokenizer class="solr.StandardTokenizerFactory"/>`
	`<filter class="solr.StandardFilterFactory"/>`
	`<filter class="solr.LowerCaseFilterFactory"/>`
	`</analyzer>`
	`</fieldType>`
	
	但是，如果你想要在词条上作更多的分析，如果使用 AnalyzingLookupFactory 作为查找实现，你就有了在索引和查询时分析的字段选项

sourceLocation	词典文件路径，用于 FileDictionaryFactory。如果该值为空，那么主索引将被用于词条和权重的来源
storeDir	保存词典文件的路径
buildOnCommit 或 buildOnOptimize	如果为 true，查找数据结构在软提交后会被重建。默认为 false，查找数据仅在 URL 参数 `suggest.build=true` 时重建。 `buildOnCommit` 会在每次软提交时重建词典， `buildOnOptimize` 仅在索引优化时重建词典。某些查找实现创建(建议)非常耗时，尤其是对大索引，这种场景下，不推荐在有高频的软提交时使用 `buildOnCommit` 或 `buildOnOptimize` ，替代的是，推荐低频的手工创建建议，使用 `suggest.build=true` 发送请求
	true 表示在 Solr 启动时或 core 重载时创建查找数据结构。

buildOnStartup	如果该参数未指定，会检查查找数据结构是否存在于磁盘，如果不存在则创建。设为 true 会导致 core 在加载(或重载)时更耗时，因为需要创建建议数据结构，而这是个耗时的操作。通常把这个设置设为 false，并手工创建建议，使用 `suggest.build=true` 发送请求

查找实现

`lookupImpl` 参数定义了在建议索引里查找词条的算法。有几种实现可供选择，有些还需要额外的配置参数

AnalyzingLookupFactory

该实现首先分析输入文本，然后将分析过的结果添加到一个加权的 FST(?)，在查找时同样这么做。

这个实现使用下面列出的额外的属性

- suggestAnalyzerFieldType: 在构建和查询时分析建议所用的字段类型
- exactMatchFirst: 默认=true，首先返回准确的建议，即便其前缀或其他字符串在 FST 里有更高的权重。(这个应该是表示优先返回完全匹配的建议)
- preserveSep: 默认=true，保留词元之间的分隔符。这表示对分词敏感(例如，baseball 和 base ball 是不同的)
- preservePositionIncrements: 如果为 true，建议会保留位置增量。这表示构建建议时保留位置间隔的分词过滤器更受尊重(?)。默认=false

FuzzyLookupFactory

AnalyzingSuggester 的扩展，但是模糊性。Levenshtein 算法用于衡量相似性。

该实现使用下面的额外属性

- exactMatchFirst: 默认=true，首先返回准确的建议，即便其前缀或其他字符串在 FST 里有更高的权重。(这个应该是表示优先返回完全匹配的建议)
- preserveSep: 默认=true，保留词元之间的分隔符。这表示对分词敏感(例如，baseball 和 base ball 是不同的)
- maxSurfaceFormsPerAnalyzedForm: Maximum number of surface forms to keep for a single analyzed form. When there are too many surface forms we discard the lowest weighted ones.
- maxGraphExpansions: When building the FST ("index-time"), we add each path through the tokenstream graph as an individual entry. This places an upper-bound on how many expansions will be added for a single suggestion. The default is -1 which means there is no limit.
- preservePositionIncrements: 如果为 true，建议会保留位置增量。这表示构建建议时保留位置间隔的分词过滤器更受尊重(?)。默认=false

- maxEdits: 可编辑的字符串最大数值。系统最大限制为 2，默认为 1
- transpositions: If true, the default, transpositions should be treated as a primitive edit operation.
- nonFuzzyPrefix: The length of the common non fuzzy prefix match which must match a suggestion. The default is 1.
- minFuzzyLength: The minimum length of query before which any string edits will be allowed. The default is 3.
- unicodeAware: 默认=false，如果为 true，则 maxEdits, minFuzzyLength, transpositions 和 nonFuzzyPrefix 参数以 unicode code 字符来计算而不是字节

AnalyzingInfixLookupFactory

BlendedInfixLookupFactory

FreeTextLookupFactory

FSTLookupFactory

TSTLookupFactory

简单紧凑的三叉树(Ternary Trie)的查找

WFSTLookupFactory

JaspellLookupFactory

来自 JsSpell 项目，基于三叉树但更加复杂的查找。如果想要更加复杂的匹配结果使用这个实现

词典实现

词典实现定义了词条如何存储。有多个选项，如果有必要的话，单个请求可以使用多个词典。

DocumentDictionaryFactory

词典包含：词条，权重和可选的来自索引的负荷(payload)

参数

- weightField: 一个存储的字段，或DocValue 数值字段。该字段可选
- payloadField: 一个存储的字段，可选的
- contextField: 上下文过滤字段，注意只有某些查找实现支持过滤

DocumentExpressionDictionaryFactory

与 DocumentDictionaryFactory 一样，但是允许指定一个任意的表达式。

参数

- weightExpression: 用于对建议评分的任意表达式，必须使用数值字段。该字段是必须滴
- payloadField: 一个存储的字段，可选的
- contextField: 上下文过滤字段，注意只有某些查找实现支持过滤

HighFrequencyDictionaryFactory

FileDictionaryFactory

多重词典

单个建议组件里可以包含多个词典实现。只需简单的分别定义建议，示例如下

```
<searchComponent name="suggest" class="solr.SuggestComponent">
  <lst name="suggester">
    <str name="name">mySuggester</str>
    <str name="lookupImpl">FuzzyLookupFactory</str>
    <str name="dictionaryImpl">DocumentDictionaryFactory</str>
    <str name="field">cat</str>
    <str name="weightField">price</str>
    <str name="suggestAnalyzerFieldType">string</str>
  </lst>
  <lst name="suggester">
    <str name="name">altSuggester</str>
    <str name="dictionaryImpl">DocumentExpressionDictionaryFactory</str>
    <str name="lookupImpl">FuzzyLookupFactory</str>
    <str name="field">product_name</str>
    <str name="weightExpression">((price * 2) + ln(popularity))</str>
    <str name="sortField">weight</str>
    <str name="sortField">price</str>
    <str name="storeDir">suggest_fuzzy_doc_expr_dict</str>
    <str name="suggestAnalyzerFieldType">text_en</str>
  </lst>
</searchComponent>
```

在查询里使用建议时，可以在请求里定义多个 'suggest.dictionary' 参数，引用搜索组件里定义的每个建议的名称即可。响应里会包含每个建议的小节。

添加建议请求处理器(Suggest Request Handler)

添加搜索组件以后，必须在 `solrconfig.xml` 里添加请求处理器(request handler)。这个请求处理器和其他的请求处理器一样，允许你配置默认参数来为建议请求服务。这个请求处理器的定义里必须包含之前定义的建议搜索组件。

```
<requestHandler name="/suggest" class="solr.SearchHandler" startup="lazy">
  <lst name="defaults">
    <str name="suggest">true</str>
    <str name="suggest.count">10</str>
  </lst>
  <arr name="components">
    <str>suggest</str>
  </arr>
</requestHandler>
```

建议请求处理器参数

参数	说明
suggest=true	该参数必须为 true
suggest.dictionary	在搜索组件里配置的词典组件的名称。该参数是必须滴，可以在请求处理器里设置，也可以在查询时通过参数传递
suggest.q	查找建议时用的查询
suggest.count	指定 Solr 返回的建议数量
suggest.cfq	cfq = Context Filter Query，如果支持的话，用于过滤建议的上下文字段
suggest.build	true 表示构建建议索引。仅在初次请求时有用；你应该不会想在每次请求时构建词典，尤其是在生产环境。如果想要保持词典最新，应该在搜索组件上使用 `buildOnCommit` 或 `buildOnOptimize` 参数
suggest.reload	true 表示重载建议索引
suggest.buildAll	true 表示构建所有的建议索引
suggest.reloadAll	true 表示重载所有的建议索引

> 上下文过滤
>
> 上下文过滤(suggest.cfg)当前仅 AnalyzingInfixLookupFactory 和 BlendedInfixLookupFactory 支持，且仅当词典实现为 Document*Dictionary。其他实现将忽视过滤请求，返回未过滤的匹配。

用法示例

根据权重获取建议

这是最基本的建议，使用单个词典和单个 Slor core，示例

```
http://localhost:8983/solr/techproducts/suggest?suggest=true&suggest.build=true&sugges
t.dictionary=mySuggester&wt=json&suggest.q=elec
```

在这个例子里，有一个简单的请求：,suggest.q 参数指定字符串 'elec'，suggest.build 参数指定构建建议词典(注意，你不会想要每次请求都构建建议词典，取而代之的是使用 buildOnCommit 或 buildOnOptimize，如果你定期改变文档)

示例的响应

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 35
  },
  "command": "build",
  "suggest": {
    "mySuggester": {
      "elec": {
        "numFound": 3,
        "suggestions": [
          {
            "term": "electronics and computer1",
            "weight": 2199,
            "payload": ""
          },
          {
            "term": "electronics",
            "weight": 649,
            "payload": ""
          },
          {
            "term": "electronics and stuff2",
            "weight": 279,
            "payload": ""
          }
        ]
      }
    }
  }
}
```

多重词典

如果定义了多个词典，可以在查询里使用它们。示例查询

```
http://localhost:8983/solr/techproducts/suggest?suggest=true&suggest.dictionary=mySugg
ester&suggest.dictionary=altSuggester&wt=json&suggest.q=elec
```

在这个例子里，发送了 suggest.q 参数为字符串 'elec' 和 2 个 suggest.dictionary

示例的响应

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 3
  },
  "suggest": {
    "mySuggester": {
      "elec": {
        "numFound": 1,
        "suggestions": [
          {
            "term": "electronics and computer1",
            "weight": 100,
            "payload": ""
          }
        ]
      }
    },
    "altSuggester": {
      "elec": {
        "numFound": 1,
        "suggestions": [
          {
            "term": "electronics and computer1",
            "weight": 10,
            "payload": ""
          }
        ]
      }
    }
  }
}
```

内容过滤

上下文过滤让你使用分离的上下文字段来过滤建议，例如分类，部门或其他。AnalyzingInfixLookupFactory 和 BlendedInfixLookupFactory 当前支持这个特性，当后台是用的 DocumentDictionaryFactory。

在你的建议配置里添加 `contextField` 。这个示例将建议名称且按分类过滤

```
<!--solrconfig.xml-->

<searchComponent name="suggest" class="solr.SuggestComponent">
  <lst name="suggester">
    <str name="name">mySuggester</str>
    <str name="lookupImpl">AnalyzingInfixLookupFactory</str>
    <str name="dictionaryImpl">DocumentDictionaryFactory</str>
    <str name="field">name</str>
    <str name="weightField">price</str>
    <str name="contextField">cat</str>
    <str name="suggestAnalyzerFieldType">string</str>
    <str name="buildOnStartup">false</str>
  </lst>
</searchComponent>
```

示例的内容过滤建议查询

```
http://localhost:8983/solr/techproducts/suggest?suggest=true&suggest.build=true& \
suggest.dictionary=mySuggester&wt=json&suggest.q=c&suggest.cfq=memory
```

该建议将只返回产品标记为 cat=memory 的建议

配置良好的 Solr 实例

如何调整 solr 以达到最佳性能？

配置 solrconfig.xml

`solrconfig.xml` 是拥有最多参数的 solr 配置文件。当配置 solr 时，通常是通过 `solrconfig.xml`，无论是直接编辑或是使用 Config API 创建配置覆盖(`configoverlay.json`)来覆盖 `solrconfig.xml` 里的值。

在 `solrconfig.xml` 里，配置如下的重要特性

- 请求处理器(request handler)，处理发送到 solr 的请求，诸如添加文档到索引的请求，或请求查询结果的请求
- 监听器(listener)，监听查询相关的特定事件，可用来触发特定代码的执行，诸如调用一些公共的查询来预热缓存
- http 请求转发
- 管理页面
- 复制、重复相关的参数

`solrconfig.xml` 文件在每个集合的 `conf/` 目录。在 `server/solr/configsets` 目录下可以发现几个注释完善的示例文件，示范了几种不同安装的最佳实践

在 solr 配置文件里使用属性变量

在配置文件 `solrconfig.xml` 可以使用属性变量作为占位符，语法是 `${属性名:可选的默认值}`。以下是设定属性值的方法

JVM 系统属性

启动 JVM 时，通过 `-D` 标识设定 JVM 系统属性，可以用在配置文件里。

例如，在 `solrconfig.xml` 文件里，可以这样定义锁定类型

```
<lockType>${solr.lock.type:native}</lockType>
```

这表示锁定类型(`lockType`)默认为 `native`，但是在启动 JVM 时，可以用 JVM 系统属性来覆盖该默认值

```
bin/solr start -Dsolr.lock.type=none
```

通常，你可以传递任意 java 系统属性给 `bin/solr` 脚本，使用标准的 `-Dproperty=value` 语法，或者，添加公共的系统属性到 solr 包含文件(`bin/solr.in.sh`)里定义的 `SOLR_OPTS` 环境变量。

solrcore.properties

如果在 solr core 的配置目录包含 `solrcore.properties` 文件，该文件可以定义任意的属性名及其值，这些属性都可以在配置文件里作为变量使用

例如，在 `conf/` 目录创建如下的 `solrcore.properties` 文件，来覆盖 `lockType`

```
# conf/solrcore.properties
solr.lock.type=none
```

> `solrcore.properties` 文件的路径和文件名，可以在 `core.properties` 里使用 `properties` 属性重新设定

core.properties 里的用户定义属性

如果你在随 `solr.xml` 一起使用 `solr.properties` 文件，该文件里用户定义的属性也可以在 xml 格式的配置文件里作为占位符使用

例如，如下的 `solr.properties` 文件

```
# core.properties
name=collection2
my.custom.prop=edismax
```

`my.custom.prop` 属性可以作为一个变量在 `solrconfig.xml` 使用

```
<requestHandler name="/select">
  <lst name="defaults">
    <str name="defType">${my.custom.prop}</str>
  </lst>
</requestHandler>
```

隐含的 core 属性

solr core 有一些隐含属性可以作为占位符使用，不论其潜在的值是在哪里或怎样初始化的。举个例子，无论 core 的名字是在 `sore.properties` 里明确设定，或是根据目录名来推断，隐含的属性 `solr.core.name` 都可以在配置文件里使用

```
<requestHandler name="/select">
  <lst name="defaults">
    <str name="collection_name">${solr.core.name}</str>
  </lst>
</requestHandler>
```

所有这些隐含属性都以 `solr.core` 作为前缀，如下

- `solr.core.name`
- `solr.core.config`
- `solr.core.schema`
- `solr.core.dataDir`
- `solr.core.transient`
- `solr.core.loadOnStartup`

DataDir 和 DirectoryFactory

用 DataDir 参数设定索引数据的位置

Solr 默认将索引数据存储在 solr home 下的 `/data` 目录。如果你想用一个不同的目录来存储索引数据，在 `solrconfig.xml` 里用 `<dataDir>` 参数。可以用绝对路径或 SolrCore 实例目录的相对路径。

示例

```
<dataDir>/var/data/solr/</dataDir>
```

如果使用 solr 的主从复制功能来复制索引数据，那么从节点的 `<dataDir>` 要和主节点一致

为索引指定 DirectoryFactory

默认的 `solr.StandardDirectoryFactory` 基于文件系统，并尝试挑选当前 JVM 和平台最优的实现。可以强行指定一种： `solr.MMapDirectoryFactory`， `solr.NIOFSDirectoryFactory`，或 `solr.SimpleFSDirectoryFactory`

```
<directoryFactory name="DirectoryFactory"
    class="${solr.directoryFactory:solr.StandardDirectoryFactory}"/>
```

`solr.RAMDirectoryFactory` 是基于内存的实现，不做持久化，在复制上不能工作。使用 DirectoryFactory 保存索引到 RAM.

```
<directoryFactory class="org.apache.solr.core.RAMDirectoryFactory"/>
```

> 如果使用 Hadoop 且把索引存储到 HDFS，可以使用 `solr.HdfsDirectoryFactory` 替换以上列举的实现

Lib Directives

可以在 `solrconfig.xml` 里定义 `<lib/>` 来加载插件

插件按出现在 `solrconfig.xml` 的顺序加载，如果有依赖的库，先列出最底层的依赖库的 jar 包

可以使用正则表达式来加载同一个目录下的其他依赖的 jar 包。所有目录都是相对 Solr `instanceDir` 目录的

> `instanceDir` 即 core 的目录，一般是 `solr/server/solr/${core_name}`，`solrconfig.xml` 文件就在其下的 `conf/` 目录

```
<lib dir="../../../contrib/extraction/lib" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-cell-\d.*\.jar" />
<lib dir="../../../contrib/clustering/lib/" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-clustering-\d.*\.jar" />
<lib dir="../../../contrib/langid/lib/" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-langid-\d.*\.jar" />
<lib dir="../../../contrib/velocity/lib" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-velocity-\d.*\.jar" />
```

Schema Factory

Solr 的 Schema API 使远程客户端通过 REST 接口访问 schema 信息和修改 schema

Schema API 支持读取所有类型的 schema，修改 schema 则取决于 `<schemaFactory/>`

Managed Schema Default

若 `<schemaFactory/>` 未在 `solrconfig.xml` 明确指定，solr 隐含的会使用 `ManagedIndexSchemaFactory`，默认为 `mutable`，且将 schema 信息保存在 `managed-chema` 文件

```
<!-- An example of Solr's implicit default behavior if no
  no schemaFactory is explicitly defined.
-->
<schemaFactory class="ManagedIndexSchemaFactory">
  <bool name="mutable">true</bool>
  <str name="managedSchemaResourceName">managed-schema</str>
</schemaFactory>
```

选项

- `mutable`：能否对 schema 数据进行改变。必须设为 **true** 来允许 Schema API 进行修改
- `managedSchemaResourceName`：可选，默认为 `managed-schema`，可以定义为除了 `schema.xml` 以外的任何名称

使用上面的默认设置，就可以用 Schema API 来修改 schema。如果想要锁定 schema，阻止以后的修改，可以将 `mutable` 设为 **false**

classic schema.xml

另一种使用可管理的 schema 的方式是明确的配置 `ClassicIndexSchemaFactory`，`ClassicIndexSchemaFactory` 需要 `schema.xml`，并且不允许运行时对 schema 的修改。`schema.xml` 必须手工编辑，且仅随 collection 加载时加载

```
<schemaFactory class="ClassicIndexSchemaFactory"/>
```

从 **schema.xml** 切换到 **managed schema**

如果一个已存在的 collection 使用 `ClassicIndexSchemaFactory`，想要转换成使用 managed schema，可以简单的修改 `solrconfig.xml`，指定使用 `ManagedIndexSchemaFactory`。

solr 重启时，检测到 `schema.xml` 文件存在，且 `managedSchemaResourceName` 指定的文件(如 `managed-schema`)不存在，则存在的 `schema.xml` 文件会被改名为 `schema.xml.bak`，且内容会重写到 `managed-schema`，文件顶部如下

```
<!-- Solr managed schema - automatically generated - DO NOT EDIT -->
```

现在可以自由地用 Schema API 来修改 schema，并可以删除掉 `schema.xml.bak`

改为手工编辑的 **schema.xml**

如果 solr 启动为使用 managed schema，想要切换到手工编辑 `schema.xml`，按如下步骤

1. 将 `managed-schema.xml` 更名为 `schema.xml`
2. 修改 `solrconfig.xml` 替换 `schemaFactory` 类
    i. 清除 `ManagedIndexSchemaFactory` 的定义
    ii. 添加 `ClassicIndexSchemaFactory` 的定义
3. 重载 core

如果使用的是 SolrCloud，需要修改 zk 上的文件

IndexConfig

`solrconfig.xml` 里的 `indexConfig` 定义了 Lucene 索引写入器的行为。在 solr 包含的示例 `solrconfig.xml` 里其设置默认是注释的，表示采用默认设置。多数场景下，默认设置很好

```
<indexConfig>
  ...
</indexConfig>
```

writing new segment 写入新段

ramBufferSizeMB

一旦累计的文档更新超过了这个存储空间(以 MB 为单位)，那么挂起的更新会立即刷入(这意思是对文档的更新先在内存里执行，等到内存里的数据达到阈值，再统一从内存写入磁盘文件...吧...)。这也可能创建一个新段(segment)或触发一次合并。使用这个设置要比 `maxBufferedDocs` 好。如果 `maxBufferedDocs` 和 `ramBufferSizeMB` 都在 `solrconfig.xml` 设置，那么任意一个限制达到都会触发一次刷入(flush)。默认值是 `100MB`

```
<ramBufferSizeMB>100</ramBufferSizeMB>
```

maxBufferedDocs

设置在刷入到新段前在内存里缓冲多少个文档。这也可能会触发一次合并。solr 默认使用 `ramBufferSizeMB` 设置

```
<maxBufferedDocs>1000</maxBufferedDocs>
```

useCompoundFile

最近的写入(尚未 merge)索引是否使用 `Compound File Segment` 格式，默认是 `false`

```
<useCompoundFile>false</useCompoundFile>
```

merging index segment 合并索引段

mergePolicyFactory

定义合并段怎么做。solr 默认使用 `TieredMergePolicy`，合并段时使用相同的大小。其他可用的策略有 `LogByteSizeMergePolicy` 和 `LogDocMergePolicy`

```
<mergePolicyFactory class="org.apache.solr.index.TieredMergePolicyFactory">
  <int name="maxMergeAtOnce">10</int>
  <int name="segmentsPerTier">10</int>
</mergePolicyFactory>
```

Controlling Segment Sizes: Merge Factors

compound file segment 复合文件段

index locks 索引锁

other indexing settings 其他索引设置

RequestHandlers 和 SearchComponents

在 `solrconfig.xml` 的 `<query>` 之后配置 request handler(请求处理器) 和 search component(搜索组件)

一个 search component 是搜索的一个特性，诸如高亮或 Faceting。search component 在 `solrconfig.xml` 里定义，与 request handler 分开，并一起注册

request handler

每个 request handler 都有名字和(Java)类。request handler 的名字被 solr 请求所引用，通常是个路径。举个例子，如果 solr 安装在 `http://localhost:8983/solr/` 且有个 collection 名为 `gettingstared`，可以发送如下的请求

```
http://localhost:8983/solr/gettingstarted/select?q=solr
```

这个查询将被名为 `/select` 的 request handler 处理。这里只使用了 `q` 参数，包含了一个简单的关键字 `solr` 作为查询的词条。如果 request handler 定义了更多的参数，它们将被用于任何我们发送到这个 request handler 的查询，除非它们被客户端发送的查询覆盖()意思是可以在 `solrconfig.xml` 里定义 request handler 时指定参数的值，但客户端发送查询时可以用自己的参数值来覆盖定义在配置文件里的值)

如果定义了另一个 request handler，应使用其名字来发送请求。举个例子，`/update` 是个处理索引更新(例如，发送新的文档来创建索引)的 request handler。`/select` 是默认用来处理查询请求的 request handler。

request handler 也可以处理嵌入其路径名的请求，例如，一个请求使用 `/myhandler/extrapath` 可能被名为 `/myhandler` 的 request handler 处理。如果有个 request handler 明确的定义了名字为 `/myhandler/extrapath` 那就优先处理这个嵌入的路径。这些都是假定你在使用 solr 自带的 request handler 类；如果使用自定义的 request handler，要确保其能处理嵌入路径。

可以用 `initParams` 配置 request handler 的默认值。这些默认值可以在每个独立的请求里作为通用的属性。例如，如果你想要创建返回相同 field 列表的多个 request handler，可以用 `initParams` 配置 field 列表。

SearchHandlers

solr 默认定义的主要的 request handler 是 SearchHandler，用于处理搜索查询。这个 request handler 已定义，且有一系列默认值定义在 `defaults` 列

例如，在默认的 `solrconfig.xml` ，第一个 request handler 是这样定义的

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
  </lst>
</requestHandler>
```

这个例子定义了 `rows` 参数为 10，表示返回多少查询结果。`echoParams` 参数，表示如果有 debug 信息返回的话，是否在查询结果里返回。同时，注意在 list 里定义默认值的方式，如果参数是 string，integer，或其他类型。

所有这些在 `searching` 章节里描述过的参数，可以被作为默认值定义在任一个 SearchHandler 里

除了 `defaults` ，SearchHandler 还有其他选项，如下

- appends：定义要添加到用户查询里的参数。可以是过滤查询，或其他应该添加到每个查询的查询规则。Solr 没有一个允许客户端覆盖这些额外的参数的机制，所以，你要绝对确信你总是想要在查询里应用这些参数

```
<lst name="appends">
    <str name="fq">inStock:true</str>
</lst>
```

在这个例子里，过滤查询 fq=inStock:true 将总是被加到每一个查询

- invariants：定义不能被客户端覆盖的参数，在 `invariants` 里定义的值将总是被使用，而不论用户，客户端传递或通过 `defaults` 或 `appends` 指定的值

```
<lst name="invariants">
  <str name="facet.field">cat</str>
  <str name="facet.field">manu_exact</str>
  <str name="facet.query">price:[* TO 500]</str>
  <str name="facet.query">price:[500 TO *]</str>
</lst>
```

在这个例子里，用于限制 facet 返回结果的字段已经被定义。如果客户端请求 facet，那么在这个配置里定义的 facet 是他/她唯一能看到的 facet。

reques hander 最后是 `componets`，定义一系列可以被 request hander 使用的 search component，只能和 request handler 一起注册。后面会讨论如何定义一个 search component。`components` 元素只能在 SearchHandler 这个 request handler 里使用。

`solrconfig.xml` 里包含了许多其他的 SearchHandler 例子，如有必要可以修改

UpdateRequestHandlers

UpdateReqeustHandler 是更新索引的 request handler。

在本指南里，这些 handler 的细节在章节 `Uploading Data with Index Handlers`

ShardHandlers

可以配置一个 request handler 跨越集群的多个 shard 做搜索，使用分布式搜索。更多有关分布式查询和配置 ShardHandler 的信息在章节 `Distributed Search with Index Sharding`

Other Request Handlers

`solrconfig.xml` 定义了其他 request handler，在以下章节

- RealTime Get
- Index Replication
- Ping

Search Components

search components 定义了 SearchHandlers 执行查询的逻辑

Default Components

有几个和所有 SearchHandler 一起工作的默认的 search component 不需要任何附加的配置。如果没有用 `first-components` 和 `last-components` 定义的 component，默认会按下面所列出的顺序执行

Component 名称	类名	更多信息
query	solr.QueryComponet	
facet	solr.FacetComponet	
mlt	solr.MoreLikeThisComponet	
highlight	solr.HighlightComponet	
stats	solr.StatsComponet	
debug	solr.DebugComponet	
expand	solr.ExpandComponet	-

如果你用这些默认名称之一注册了一个新的 search componen，新定义的 component 将取代默认的

First-Components and Last-Components

可以定义一些 component 在上面列出的默认 components 之前(`first-components`)或之后(`last-components`)被使用

> `first-components` 和/或 `last-components` 只能和默认 components 联合使用。如果自定义了 `components` ，默认 components 将不会执行，且 `first-components` 和 `last-components` 也无效

```
<arr name="first-components">
  <str>mycomponent</str>
</arr>
<arr name="last-components">
  <str>spellcheck</str>
</arr>
```

Components

如果自定义 `components` ，前述的默认 components 将不会执行，且 `first-components` 和 `last-components` 也无效

```
<arr name="components">
  <str>mycomponent</str>
  <str>query</str>
  <str>debug</str>
</arr>
```

Other Userful Components

本指南的其他章节描述了很多其他有用的 components，它们是

- `SpellCheckComponent`
- `TermVectorComponent`
- `QueryElevationComponent`
- `TermsComponent`

InitParams

`solrconfig.xml` 里的 `<initParams>` 使你可在 handler 配置之外定义 request handler 参数。

用例是

- 一些 handler 是隐含定义的(应该是说某些属性在源码里硬编码了默认值)，应该有一个新增/添加/覆盖这些隐含属性的方法
- 少数属性用于多个 handler。这样可以仅定义一次这些属性并应用于多个 handler

例如，如果想要几个 handler 返回相同的字段列表，可以创建一个 `<initParams>` 元素而不用在每一个 handler 里定义相同的参数。如果有单个 handler 要返回不同的字段，可以在 `<requestHandler>` 里单独定义参数来覆盖

`<initParams>` 里的属性和配置镜像了 request handler 的属性和配置(这应该是说其 xml 节点和 `<requestHandler>` 一样)。它能包含 defaults，appends，invariants，和任何 request handler 一样。

例如，在 `data_driven_config` 例子里的一个 `<initParams>` 定义

```
<initParams path="/update/**,/query,/select,/tvrh,/elevate,/spell,/browse">
  <lst name="defaults">
    <str name="df">_text_</str>
  </lst>
</initParams>
```

这为 path 里的全部 request handler 设置了默认的搜索字段("df")为 "_text_"。如果以后想要 `/query` request handler 默认搜索一个不同的字段，可以在 `<requestHandler>` 里定义这个参数覆盖 `<initParams>`

语法和 `<requestHandler>` 类似，如下

属性	描述
path	逗号分隔的路径列表，可以用通配符来定义嵌入路径，如下文所述
name	这一批参数的名称。如果没有一个确定的路径，这个名字可以直接用在 request handler 里。如果给 `<initParams>` 一个名称，就可以在 `<requestHandler>` 里引用这个名称 例如，一个 `<initParams>` 的名称为 "myParams"，可以在定义 request handler 时使用 `<requestHandler name="/dump1" class="DumpRequestHandler" initParams="myParams"/>`

通配符

`<initParams>` 支持用通配符定义嵌套路径。单个星号(*)指示嵌套路径深一层，2 个星号(**)指示所有深度的嵌套路径。

例如，有如下的 `<initParams>`

```
<initParams name="myParams" path="/myhandler,/root/*,/root1/**">
  <lst name="defaults">
    <str name="fl">_text_</str>
  </lst>
  <lst name="invariants">
    <str name="rows">10</str>
  </lst>
  <lst name="appends">
    <str name="df">title</str>
  </lst>
</initParams>
```

定义了 3 个路径

- `/myhandler` 声明了一个明确的路径
- `/root/*` 表示 `/root` 下的路径，不含子目录
- `/root1/**` `/root`` 开头的所有路径

当定义 request handler 时，通配符将如下工作

```
<requestHandler name="/myhandler" class="SearchHandler"/>
```

`/myhandler` 类在 `<initParams>` 里定义了 `path`，所以会使用那些参数

```
<requestHandler name="/root/search5" class="SearchHandler"/>
```

`/root` 下定义了一层深度，所以这个 request handler 将使用这些参数，但是，下面这个不止一层深度就不会使用

```
<requestHandler name="/root/search5/test" class="SearchHandler"/>
```

想要定义所有深度的嵌套路径，应该使用双星号，如同 `/root1/**`

```
<requestHandler name="/root1/search/tests" class="SearchHandler"/>
```

`/root1` 下的任意路径，不论是否在 request handler 里明确定义，都会使用 `initParams` 里定义的参数

UpdateHandlers

`solrconfig.xml` 的 `<updateHandler>` 元素的配置对索引更新有影响，其设置影响的是内部的更新是如何完成的，对 request hander 处理客户端的更新请求的的高级配置没有影响

```
<updateHandler class="solr.DirectUpdateHandler2">
  ...
</updateHandler>
```

commit 提交

发送到 solr 的数据直到被提交到索引才可搜索。原因是在某些场景下提交较慢，且应与其他的提交请求隔离以避免覆盖数据。所以，在数据提交时最好进行控制。控制提交的时机有几个选项

commit and softCommit 提交和软提交

在 solr 里，一个 `提交(commit)` 是要求 solr `提交` 修改到 Lucene 索引文件的行动。默认情况下，提交行动的结果是硬提交——所有 Lucene 索引文件到稳定的存储(磁盘)。当客户端更新请求包含了 `commit=true` 参数，这会确保被更新里的新增、删除影响的所有索引段会在索引更新结束时立即写入到磁盘。

如果指定了附加的标记 `softCommit=true` ，solr 执行软提交，意思是 solr 会立即提交修改到 Lucene 数据结构但不保证 Lucene 文件写入到磁盘。这是近实时(Near Real Time)存储的一个实现，促进文档可见的一个特性，你不需要等待后台的合并和存储(如果使用 SolrCloud，则是对于 zk)结束。完整的提交意味着，如果服务器崩溃，solr 会精确的知道你的数据保存在哪；软提交意味着数据已存储，但是其位置信息尚未存储。由于不需要等待后台的合并结束，软提交给你更快的可见性。

关于 Near Real Time 操作的更多信息，参考 `Near Real Time Searching`

autoCommit 自动提交

这些设置控制挂起的更新自动推送到索引的频率。可以用 `commitWithin` 取代 `autoCommit` ，该参数可以在发送更新请求到 solr 时定义，或在一个 updateRequestRequest 里定义

设置	说明
maxDocs	从上次提交后至今发生的更新的数量
maxTime	最早的未提交更新开始至今的毫秒数
openSearcher	当执行一次提交时是否开启一个新的 searcher。 默认为false，本次提交会把最近的索引变更刷入磁盘，但不会开启一个新的 searcher 来使这些变更可见

如果 `maxDocs` 或 `maxTime` 任一个满足，solr 自动执行提交。如果没有 `autoCommit` 标签，那么只有明确的提交会更新索引。是否使用自动提交取决于你的应用。

决定最好的自动提交参数，是在性能和准确性(这应该是搜索结果的准确性)之间做平衡。频繁的更新会提升准确性，因为新的内容会更快的被搜索到，但会降低性能。降低频率能提升性能但需要更久才能在查询里显示更新

```
<autoCommit>
  <maxDocs>10000</maxDocs>
  <maxTime>1000</maxTime>
  <openSearcher>false</openSearcher>
</autoCommit>
```

可以像设定自动提交一样来设定软自动提交，只需要把 `autoCommit` 替换成 `autoSoftCommit`

```
<autoSoftCommit>
  <maxTime>1000</maxTime>
</autoSoftCommit>
```

cmomitWithin

`commitWithin` 设置可强制文档在一个定义的时间段提交。多用于 `Near Real Time Searching`，默认是执行软提交。在主从架构下，不会复制新文档到从服务器。可以添加一个参数来强制硬提交，如下

```
<commitWithin>
  <softCommit>false</softCommit>
</commitWithin>
```

这样配置后，每次当你的更新信息的一部分是 `commitWithin`，就会自动执行硬提交

Event Listeners 事件监听器

UpdateHandler 可配置更新相关的时间监听器。可以在任何提交(`event="postCommit"`)以后或者仅在优化指令(`event="postOptimize"`)之后被触发

可以编写自己的更新事件监听器，但一般是通过 `RunExecutableListerer` 来运行外部程序

设置	说明
exe	要运行的程序名称，应该是相对于 solr home 的路径名
dir	工作目录，默认是 "."
wait	强制调用线程等待，直到程序返回响应。默认=true
args	传送给程序的任意参数，默认为空
env	任意要设置的环境变量，默认为空

Transaction Log 事务日志

在 `RealTime Get` 里描述过，事务日志是那个特性必须的。在 `solrconfig.xml` 的 `updateHandler` 里配置

实时获取当前依赖更新日志，默认是开启滴。配置如下

```
<updateLog>
  <str name="dir">${solr.ulog.dir:}</str>
</updateLog>
```

有 3 个专家级的配置项可影响索引性能及一个 replica 在必须进入全量恢复前可以落后于更新多远——参考 写入侧容错 获取更新信息

设置名称	类型	默认值	说明
numRecordsToKeep	int	100	每个日志多少条更新记录
maxNumLogsToKeep	int	10	最多几个日志
numVersionBuckets	int	65536	翻译不能...

一个例子， `solrconfig.xml` 要包含在 `<config><updateHandler>` ，使用如下高级设置

```
<updateLog>
  <str name="dir">${solr.ulog.dir:}</str>
  <int name="numRecordsToKeep">500</int>
  <int name="maxNumLogsToKeep">20</int>
  <int name="numVersionBuckets">65536</int>
</updateLog>
```

Query Settings

本章节的设置影响了 solr 处理和响应请求的方式。这些设置全都在 `solrconfig.xml` 的 `<query>` 元素下

```
<query>
  ...
</query>
```

cache

solr cache 和确定的 index searcher 实例关联，一个 searcher 生命期里不会改变的索引的明确视图。只要那个 index searcher 被使用，它的 cache 里的任何项目都是有效和可重用的。solr caching 和其他许多应用的 caching 不同，solr 里缓存的对象不会在一段时间间隔后过期；它们会在 index searcher 生命期里保持有效

当一个新的 searcher 被开启，当前的 searcher 继续为请求服务，同时新的 searcher 自动预热其缓存。新的 searcher 用当前 searcher 的缓存来预填充自己的。当新的 searcher 就绪，它注册成为当前的 searcher 并开始处理所有新的搜索请求。旧的 searcher 完成所有的请求后被关闭

solr 有 3 个 cache 实现：`solr.search.LRUCache`，`solr.search.FastLRUCache`，`solr.search.LFUCache`

缩写 LRU 代表 Least Recently Used(最近最少使用)，当一个 LRU cache 充满，最后访问的最老的对象会被清理，给新的对象提供空间。结果就是频繁访问的对象会留在 cache，同时那些不是频繁访问的对象被清理，如果需要的话会重新从索引里取到 cache

`FastLRUCache`，solr 1.4 已引入，无锁设计(lock-free，相对于 `LRUCache` 可以做到读写不加锁)，所以很适合缓存那些在一个请求里被命中多次的对象

`LRUCache` 和 `FastLRUCache` 在预热时，用一个整数或百分比来评估相对于当前 cache 大小的的自动预热数(意思是初始化时预加载多少数据吗?)

`LFUCache` 引用 Least Frequently Used(最近最不常用) cache。工作方式类似 LRU cache，除了当 cache 充满时最少使用的对象会被淘汰。

> **LRU 和 LFU 区别**
>
> - LRU 根据最后一次命中时间来淘汰数据
> - LFU 淘汰在一个时间段内命中次数最少的
>
> 例如，一个空闲很久的对象上一秒刚刚被命中，LRU 很大概率保留该对象，LFU 很大概率淘汰该对象
>
> by goosebaobao

solr 管理界面的统计页会显示所有活动 cache 的执行信息，可以帮助你调整适合应用的 cache 尺寸。当一个 searcher 结束，其 cache 的使用概要会写入日志。

每个 cache 都有定义初始尺寸(`initialSize`)，最大尺寸(`size`)，预热时的对象数量(`autowarmCount`)的设置。 LRU 和 FastLRUCache 实现可以用百分比来代替 `autowarmCount` 的绝对数量值

以下是各个 cache 的细节描述

filterCache

`SolrIndexSearcher` 用这个 cache 过滤匹配一个查询的所有未排序的文档集。数值属性用来控制 cache 里的对象数量。

solr 用 `filterCache` 来缓存使用 `fq` 参数的查询结果。后续使用相同参数的查询命中并从 cache 返回结果。参考 搜索 了解 `fq` 参数详情。

当配置参数 `facet.method` 设置为 `fc` ，solr 也把这个用于缓存 faceting。

```
<filterCache class="solr.LRUCache"
    size="512"
    initialSize="512"
    autowarmCount="128"/>
```

documentCache

介个 cache 保存 Lucene 文档对象(每个文档的存储字段)。由于 Lucene 内部的文档 id 是临时的，这个 cache 不会自动预热。 `documentCache` 的大小应该总是大于 `max_results` 乘以 `max_concurrent_queries` ，以确保不需要在一个请求里重新获取文档。你的文档存储的字段越多，这个 cache 占用的内存越多。

```
<documentCache class="solr.LRUCache"
    size="512"
    initialSize="512"
    autowarmCount="0"/>
```

使用自定义 cache

你也可以定义一个用你自己的代码的 cache。可以调用 `SolrIndexSearch` 的方法 `getCache()`，`cacheLookup()`，`cacheInsert()` 拿到你的 cache 对象并使用它。

```
<cache name="myUserCache" class="solr.LRUCache"
    size="4096"
    initialSize="1024"
    autowarmCount="1024"
    regenerator="org.mycompany.mypackage.MyRegenerator" />
```

如果想要自动预热你的 cache，包含一个 `regenerator` 属性，值为 `solr.search.CacheRegenerator` 实现类的全路径名。也可以使用 `NoOpRegenerator`，用旧的项目简单填充 cache。在 `regenerator` 参数里定义：`regenerator="solr.NoOpRegenerator"`

Query Sizing and Warming

maxBooleanClauses

一个 boolean 查询最多有几个从句。可影响范围或前缀查询扩展的大量 boolean 查询。如果超过限制会抛出异常

```
<maxBooleanClauses>1024</maxBooleanClauses>
```

> 该选项对所有 core 生效。如果多个 `solrconfig.xml` 文件里该属性不一致，最后一个初始化的 core 的值被采用

enableLazyFieldLoading

若设为 true 则非直接请求的字段会延迟加载。如果大多数一般查询只需要一个小的字段子集，尤其是很少访问的字段有大的尺寸时可提升性能

```
<enableLazyFieldLoading>true</enableLazyFieldLoading>
```

useFilterForStortedQuery

配置 solr 使用过滤器来满足一个搜索。如果请求的排序不包含 "score"，`filterCache` 将用过滤器来匹配该查询。多数场景下，如果相同的搜索请求经常用不同的排序选项且都不用 "score" 才有用。

```
<useFilterForSortedQuery>true</useFilterForSortedQuery>
```

QueryResultWindowSize

和 `queryResultCache` 一起使用，将 cache 超出请求的文档的 id。例如，如果一个搜索请求文档 10 到 19，且 `QueryResultWindowSize` 为 50，文档 0 到 49 将被 cache。

```
<queryResultWindowSize>20</queryResultWindowSize>
```

queryResultMaxDocsCached

这个参数设置 `queryResultCache` 里任意 key 缓存的文档最大数量

```
<queryResultMaxDocsCached>200</queryResultMaxDocsCached>
```

useColdSearcher

若当前没有注册的 searcher，搜索请求应该等待一个新的 searcher 预热(false)还是立即执行(true)。当设为 false，请求会阻塞，直到 searcher 完成其 cache 的预热

```
<useColdSearcher>false</useColdSearcher>
```

maxWarmingSearchers

任何时刻，后台正在预热的 searcher 最多可以有几个。超过该限制会抛出错误(error)。对只读的从机(slave)，2 是合理的。主机(master)应高一点点

```
<maxWarmingSearchers>2</maxWarmingSearchers>
```

Query-Related Listeners 查询相关监听器

在 cache 章节已描述过，新的 Index Searcher 是缓存的。使用监听器触发执行查询相关的任务是可能的。最通用的是在定义查询推动 Index Searcher 启动时预热。这个做法其中一个好处是字段 cache 会预填充以加速排序。

良好的查询选择器(?)是这类监听器的关键。最好选择你最常用和/或最重的查询，并且不仅包括用到的关键字，还有任何其他参数诸如排序或过滤请求。

有 2 类事件可以触发监听器。当一个新的 searcher 正在准备但当前没有已注册的 searcher 来处理请求或来获取预热的数据(如 solr 启动)时一个 `firstSearch` 事件发生。一个新的 searcher 正在准备且当前有 searcher 可处理请求时一个 `newSearcher` 事件发生。

下面的例子可在 solr 包含的 `sample_techproducts_configs` 配置集 的 `solrconfig.xml` 文件里找到，演示了 `solr.QuerySenderListener` 类预热一些特定查询的用法

```
<listener event="newSearcher" class="solr.QuerySenderListener">
  <arr name="queries">
  <!--
    <lst><str name="q">solr</str><str name="sort">price asc</str></lst>
    <lst><str name="q">rocks</str><str name="sort">weight asc</str></lst>
  -->
  </arr>
</listener>

<listener event="firstSearcher" class="solr.QuerySenderListener">
  <arr name="queries">
    <lst><str name="q">static firstSearcher warming in solrconfig.xml</str></lst>
  </arr>
</listener>
```

> 上面的代码来自一个实例 `solrconfig.xml`。一个关键的最佳实践是在部署你的应用到生产环节前修改这些默认值，但请注意：在 `newSearcher` 事件里的查询被注释而 `firstSearcher` 事件里的没有。如果和你的搜索应用无关的话，你的 index searcher 自动预热 "static firstSearcher warming in solrconfig.xml" 没有意义。

RequestDispatcher

`solrconfig.xml` 的 `requestDispatcher` 元素控制了 solr 的 HTTP `RequestDispatcher` 实现如何响应请求。其参数定义了是否要处理 `/select` url，是否应支持远程流，上传文件的最大尺寸，及它如何响应 HTTP cache 头

handleSelect

> `handlerSelect` 是为了向后兼容，新的 solr 爱好者没有必要修改默认的配置

`<requestDispatcher>` 元素的首个配置项是 `handleSelect`。取值为 "true" 或 "false"。它控制着 solr 怎样响应 `/select?qt=XXX` 这样的请求。默认值为 "false" 表示如果一个 requestHandler 没有明确的注册到 `/select` 则忽略 `/select`。"true" 将路由查询请求到 `qt` 定义的解析器

在 solr 最新版本里，`/select` requestHandler 是默认定义的，所以 `false` 设置会很好的生效。参考 RequestHandler 和 SearchComponent 获取更多信息

```
<requestDispatcher handleSelect="true" >
  ...
</requestDispatcher>
```

requestParsers

`requestParsers>` 控制解析请求。这是个空 xml 元素，没有内容只有属性。

`enableRemoteStreaming` 属性控制是否允许远程内容流。`false` 不允许流，默认为 `true` 让你指定内容的位置，使用 `stream.file` 或 `stream.url` 参数

如果允许远程流，确保开启验证。否则，其他人可能通过任意 url 访问你的内容。把 solr 放在防火墙后面来防止不受信任的客户端访问也是个好主意。

`multipartUploadLimitInKB` 设定了通过 HTTP POST 请求提交的文档的最大尺寸，单位为 KB。

`formDataUploadLimitInKB` 设定了 HTTP POST 提交的表单数据(application/x-www-form-urlencoded)的最大尺寸，KB

`addHttpRequestToContext` 用于表明原始的 `HttpServletRequest` 对象应该被包含在 `SolrQueryRequest` 的上下文里，使用 `httpRequest` 为 key。这个 `HttpServletRequest` 不会被任何 Solr component 使用，但开发自定义的插件时可能会有用。

```
<requestParsers enableRemoteStreaming="true"
    multipartUploadLimitInKB="2048000"
    formdataUploadLimitInKB="2048"
    addHttpRequestToContext="false" />
```

httpCaching

`httpCaching` 控制 HTTP cache 控制头。不要和 solr 内部的 cache 配置混淆，这个元素控制 HTTP 响应的 caching 是在 W3C HTTP 规范里定义的

这个元素有 3 个属性和 1 个子元素。`<httpCaching>` 控制 GET 请求是否返回一个 304，如果是，响应应该是啥。当一个 HTTP 客户端应用发布一个 GET，如果资源在它上次获取以后没有被修改，会指定一个 304 响应

参数	描述
never304	`true` 表示 GET 请求永远不会返回 304，即便请求的资源没有被修改。同时其他 2 个属性被忽略。设为 true 是为了方便开发
lastModFrom	取值为 `openTime` (默认) 或 `dirLastMod` ， `openTime` 表示最后修改时间，即用来与客户端发送的 If-Modified-Since 头信息比较的时间，是 searcher 启动的时间。如果你想要索引最后在磁盘更新的精确时间，使用 `dirLastMod`
etagSeed	`ETag` 头信息的值。修改这个值有助于强制客户端重新获取内容，即使索引没有改变——例如，如果对配置做了修改。

```
<httpCaching never304="false"
      lastModFrom="openTime"
      etagSeed="Solr">
  <cacheControl>max-age=30, public</cacheControl>
</httpCaching>
```

cacheControl

上面的属性之外，`<httpCaching>` 有个子元素： `<cacheControl>` 。这个元素的内容将作为 HTTP 响应头 Cache-Control 的值。这个头用来修改请求客户端的默认的 caching 行为。Cache-Control 头可能的取值在 HTTP 1.1 规范的 14.9 章节定义

设置 max-age 字段控制客户端在重新请求之前可以重用 cache 响应多久。设置这个时间间隔应根据你多久更新你的索引，且无论你的应用程序使用的内容有点点过期。

`must-revalidate` 设置要求客户端在重用前向服务器确认其 cache 副本依然有效。这会确保使用更及时的结果，避免不必要的第二次获取

Update Request Processors

solr 接收的每个更新请求都运行在插件 Update Request Processor 的链上。这很有用，例如，要添加一个字段到索引的文档或修改某个字段的值或因提交的文档不符合规则而放弃更新。事实上，solr 大量的特性是作为 Update Processor 来实现的，因此理解该插件如何工作和哪里配置很有必要。

Anatomy and life cycle 结构和生命周期

一个 Update Request Processor 作为一个或多个 update processor 链的一部分创建。solr 创建了一个默认的 update request processor 链实现一些基本的特性。这个默认的链用来处理每一个 update request 除非用户配置和指定一个不同的自定义的链

最简单的了解 update request processor 的方法是查看抽象类 UpdateRequestProcessor 的 javadoc 文档。每个 UpdateRequestProcessor 应该有一个相应的继承自 UpdateRequestProcessorFactory 的工厂类。solr 用这个工厂类创建该插件的一个新实例。这种设计有 2 个好处

1. 一个 update request processor 不需要是线程安全的，因为它只会被一个请求线程使用并在请求结束后销毁
2. 工厂类可以接受配置参数并可能需要在请求间维持任意的状态。工厂类必须是线程安全的。

每个 update request processor 链在 solr core 加载期间构建并缓存到该 core 卸载。链里的每个 UpdateRequestProcessorFactory 也用 `solrconfig.xml` 里的配置实例化和初始化。

当一个更新请求被 solr 接收，它查找更新链用于该请求。链里的每个 UpdateRequestProcessor 的新实例被相应的工厂创建。这个更新请求被相应的 UpdateCommand 对象解析。每个 UpdateRequestProcessor 实例负责调起链上的下一个插件。不调起下一个处理器会导致链式处理短路(提前结束链式处理)，甚至可以抛出异常来放弃后续处理

> 单个更新请求可能包含一批多个文档要保存或删除，因此 UpdateRequestProcessor 相应的 processXXX 方法会被调用多次。但是，可以保证是单个线程连续调用这些方法()

Configuration 配置

update request processor 链的创建，既可以在 solrconfig.xml 里直接创建完整的链，也可以在 solrconfig.xml 里个别的创建 update processor，然后在运行时用请求参数指定所有处理器动态的创建

但是，在我们理解如何配置 update processor 链之前，我们必须先学习默认的 update processor 链，因为它提供了在自定义的请求处理器链也需要的基本的特性

The default update request processor chain 默认更新请求处理器链

如果 solrconfig.xml 里没有配置 update processor 链，solr 自动创建默认的 update processor 链作用于所有更新请求。这个默认的 update processor 链由下面的处理器按顺序组成

1.LogUpdateProcessorFactory - 在请求期间跟踪命令的处理并记下日志
2.DistributedUpdateProcessorFactory - 负责将更新请求分发到正确的节点。例如，路由请求到正确的 shard 的 leader，且将请求从 leader 分发到每个 replica。这个 processor 仅在 SolrCloud 模式才会激活 3.RunUpdateProcessorFactory - 使用 solr 内部的 api 来执行更新

每个 processor 执行一个基本功能，且自定义链通常会包含全部这些 processor。在自定义链里 RunUpdateProcessorFactory 通常是最后一个 processor

Custom update request processor chain 自定义更新请求处理器链

下面的例子演示了如何在 solrconfig.xml 配置自定义链

```
<!-- updateRequestProcessorChain -->

<updateRequestProcessorChain name="dedupe">
  <processor class="solr.processor.SignatureUpdateProcessorFactory">
    <bool name="enabled">true</bool>
    <str name="signatureField">id</str>
    <bool name="overwriteDupes">false</bool>
    <str name="fields">name,features,cat</str>
    <str name="signatureClass">solr.processor.Lookup3Signature</str>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

上面的例子，用 SignatureUpdateProcessorFactory， LogUpdateProcessorFactory，RunUpdateProcessorFactory 新建了一个名为 "dedupe" 的更新处理器链。SignatureUpdateProcessorFactory 有几个参数，诸如 "signatureField"， "overwriteDupes" 等等。这个链示例了 solr 如何配置对文档去重，通过计算用 name， features， cat 字段的值

的签名用作 id 字段。你可能注意到，这个例子没有指定 DistributedUpdateProcessorFactory - 因为这个处理器对 solr 正确运作及其重要，solr 将自动插入 DistributedUpdateProcessorFactory 到不包含它的链里，刚好在 RunUpdateProcessorFactory 之前。

> **RunUpdateProcessorFactory** 不要忘记将 RunUpdateProcessFactory 添加到你在 solrconfig.xml 里定义的任何链的末尾，否则那个链的更新请求将不会真正地更新索引数据

配置顶级处理器

在 solrconfig.xml 也可以配置独立于链的 update request processor

```
<!-- updateProcessor -->

<updateProcessor class="solr.processor.SignatureUpdateProcessorFactory" name="signatur
e">
  <bool name="enabled">true</bool>
  <str name="signatureField">id</str>
  <bool name="overwriteDupes">false</bool>
  <str name="fields">name,features,cat</str>
  <str name="signatureClass">solr.processor.Lookup3Signature</str>
</updateProcessor>

<updateProcessor class="solr.RemoveBlankFieldUpdateProcessorFactory" name="remove_blan
ks"/>
```

这个例子里，配置了名为 "signature" 的 SignatureUpdateProcessorFactory 和名为 "remove_blanks" 的 RemoveBlankFieldUpdateProcessorFactory。在 solrconfig.xml 如上定义后，就可以在 update request processor 链里引用他们，如下

```
<!-- updateRequestProcessorChains and updateProcessors -->

<updateProcessorChain name="custom" processor="remove_blanks,signature">
  <processor class="solr.RunUpdateProcessorFactory" />
</updateProcessorChain>
```

Update processors in SolrCloud

solr 单机模式，每个更新只会被链里的更新处理器执行一次。在 SolrCloud 模式下更新处理器的行为值得特别考虑。

SolrCloud 的一项关键功能是路由和分发请求 - 对于更新请求这是由 DistributedUpdateRequestProcessor 实现的，且这个处理器因其重要功能被 solr 给与了特殊地位。

在分布式 SolrCloud 场景，链里的所有 DistributedUpdateProcessor 之前 的处理器在接收客户端请求的第一个节点(node)上运行，不论这个节点是 leader 或 replica。然后 DistributedUpdateProcessor 转发更新到恰当的 shard 的 leader(或多个 leader，如果该更新影响多个文档，例如删除或提交)。Shard Leader 通过事务日志来运用原子更新，然后转发更新到 shard 的所有 replica。Leader 和每个 replica 执行链上的所有在 DistributedUpdateProcessor 之后 的处理器。

例如，考虑上节看到的 "dedupe" 链。假定 SolrCloud 集群由 3 个节点组成，NodeA 是 shard1 Leader，NodeB 是 shard2 Leader，NodeC 是 shard2 replica。一个更新请求发送到 NodeA，由于这个更新属于 shard2，所以 NodeA 转发给 NodeB，NodeB 分发给自己的 replica NodeC。我们看看每个 node 上都发生了什么

- NodeA：执行更新从 SignatureUpdateProcessor(计算签名并赋值给 id 字段)，然后是 LogUpdateProcessor 和 DistributeUpdateProcessor。这个处理器判断这个更新实际属于 NodeB，转发更新请求给 NodeB。不再进行后续的处理。这是必须的，因为下一个处理器即 RunUpdateProcessor 将不会在 shard1 上执行更新以避免 shard1 和 shard2 上有重复的数据。
- NodeB：接收到更新请求，发现是从另一个 node 转发过来的(不是从客户端直接过来的)。这个更新被直接发送给 DistributeUpdateProcessor，因为它已经在 NodeA 上通过了 SignatureUpdateProcessor，再计算签名就没有必要了。DistributeUpdateProcessor 判断这个更新的确属于当前 node，分发更新请求到 NodeC 上的 replica，然后转发更新请求到链上后续的 RunUpdateProcessor
- NodeC：接收到更新请求，发现是从 leader 分发过来的。这个更新请求直接发送到 DistributeUpdateProcessor，执行一些一致性检查，转发给链上后续的 RunUpdateProcessor

总结

1. 在 DistributeUpdateProcessor 之前的所有处理器只会在接收到更新请求的第一个 node 上运行，不论它是转发 node(上面的 NodeA)或 leader(NodeB)。我们称之为预处理器
2. 在 DistributeUpdateProcessor 之后的所有处理器只会在 leader 和 replica 节点上运行。不会在转发 node 上执行。这些处理器称为后处理器

上一节里，我们看到 updateRequestProcessorChain 配置有 `processor="remove_blanks,signature"` 。这表示这些处理器是第一类处理器(预处理器)，只在转发节点上运行。类似的，我们可以将它们配置为第二类处理器，只要用 "post-processor" 属性来指定，如下所示

```
<!-- post-processors -->

<updateProcessorChain name="custom" processor="signature"
    post-processor="remove_blanks">
  <processor class="solr.RunUpdateProcessorFactory" />
</updateProcessorChain>
```

总之，在 SolrCloud 集群上通过负载均衡随机的发送请求，使其仅在转发节点上运行一个处理器是分配高代价的计算诸如删除重复数据的一个伟大的方法。否则这个高代价的计算将会在 leader 和 replica 节点重复执行。

> 预处理器和原子更新 DistributeUpdateProcessor 负责在 leader 节点上处理所有文档的`原子更新`，这意味着仅在转发节点执行的预处理器只能处理特定的部分文档。如果你的处理器必须处理全部文档，那么唯一的选择是指定为后处理器。

Using custom chains 使用自定义链

update.chain request parameter 请求参数 update.chain

请求参数 update.chain 可用于任意更新请求，选择一个在 solrconfig.xml 里的自定义链。例如，要选择前述章节里的 "dedupe" 链，可以如下发送请求

```
# update.chain

curl "http://localhost:8983/solr/gettingstarted/update/json?update.chain=dedupe&commit
=true" -H 'Content-type: application/json' -d '
[
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  },
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  }
]'
```

上面的例子，2 个同样的文档将只会索引其中一个

processor & post-processor request parameters 请求参数 processor 和 post-processor

使用 "processor" 和 "post-processor" 请求参数，我们可以动态构造一个自定义的更新请求处理器链。多个处理器可以用逗号分隔，示例如下

```
# Constructing a chain at request time

# Executing processors configured in solrconfig.xml as (pre)-processors
curl "http://localhost:8983/solr/gettingstarted/update/json?processor=remove_blanks,signature&commit=true" -H 'Content-type: application/json' -d '
[
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  },
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  }
]'

# Executing processors configured in solrconfig.xml as pre and post processors
curl "http://localhost:8983/solr/gettingstarted/update/json?processor=remove_blanks&post-processor=signature&commit=true" -H 'Content-type: application/json' -d '
[
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  },
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  }
]'
```

第一个例子，solr 动态创建的链有 "signature" 和 "remove_blanks" 预处理器，仅在转发节点执行；第二个例子，"remove_blanks" 是预处理器，"signature" 在 leader 和 replica 上作为后处理执行

configuring a custom chain as a default 配置自定义链为默认

我们也可以指定一个自定义链为所有请求的默认，取代在每个请求参数里指定链名称

可以在指定路径里添加 "update.chain" 或 "processor" 和 "post-processor" 作为默认参数，参考 iniParams；或作为默认处理器，参考 defaults

下面例子是用 initParams 设定使用自定义更新链处理 "/update/" 开头的请求

```
<!-- initParams -->

<initParams path="/update/**">
  <lst name="defaults">
    <str name="update.chain">add-unknown-fields-to-the-schema</str>
  </lst>
</initParams>
```

如下所示，使用 "defaults" 也可以实现类似效果

```
<!-- defaults -->

<requestHandler name="/update/extract" startup="lazy"
    class="solr.extraction.ExtractingRequestHandler" >
  <lst name="defaults">
    <str name="update.chain">add-unknown-fields-to-the-schema</str>
  </lst>
</requestHandler>
```

Update Request processor Factories

FieldMutatingUpdateProcessorFactory derived factories

Update Processor factories that can be loaded as plugins

Update Processor factories you should *not* modify or remove.

Codec Factory 编解码器工厂

`solrconfig.xml` 的 `CodecFactory` 用于确定在写入索引到磁盘时使用哪个 Lucene 的 Codec

如果不指定，使用 Lucene 默认的 Codec，但是 solr.SchemaCodecFactory 也是可用的，它支持 2 个特性

1. 基于 Schema 定义每个字段类型，`docValuesFormat` 和 `postingsFormat`，参考 FieldType Definitions and Properties
2. `compressionMode` 选项
   - `BEST_SPEED` 默认，为搜索速度优化
   - `BEST_COMPRESSION` 为磁盘空间占用优化

示例

```
<codecFactory class="solr.SchemaCodecFactory">
  <str name="compressionMode">BEST_COMPRESSION</str>
</codecFactory>
```

solr cores 及 solr.xml

在 solr 里，术语 *core* 指的是单个的索引及关联的事务日志和配置文件(其中包括 `solrconfig.xml` ，schema 文件)。如果有必要，你的 solr 可以有多个core，让你可以在同一个服务器索引不同结构的数据，并控制你的数据如何展示给不同的用户。在 solrcloud 模式你可能对术语 *collection* 更熟悉。一个 collection 在后台是由一个或多个 core 组成的。

可用 `bin/solr` 脚本创建 core，或作为 solrcloud collection 的一部分使用 API 创建。core 的属性(诸如索引数据目录，配置文件目录，名称，及其他)在 `core.properties` 文件里定义。你的 solr 安装目录下的任何目录(或是在 `solr_home` 定义的目录)下的任何 `core.properties` 文件会被 solr 发现，其中定义的属性将会被用于文件里命名的 core。

在单机模式， `solr_home` 目录必须有 `solr.xml` 。在 solrcloud 模式，如果存在， `solr.xml` 会从 zk 加载

> 在 solr 旧版本，必须在 `solr.xml` 的 `<core>` 里预先定义 core 让 solr 知道他们。现在，solr 支持自动发现 core，它们再不用被明确定义。推荐用 API 动态的创建 cores/collections

solr.xml 格式

本章节将描述包含在 solr 里的默认的 `solr.xml` 文件，及如何修改。

定义 Solr.xml

你可以在 solr 主目录或 zk 找到 `solr.xml`。默认的 `solr.xml` 看上去是这样滴

```
<solr>
  <solrcloud>
    <str name="host">${host:}</str>
    <int name="hostPort">${jetty.port:8983}</int>
    <str name="hostContext">${hostContext:solr}</str>
    <int name="zkClientTimeout">${zkClientTimeout:15000}</int>
    <bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
  </solrcloud>
  <shardHandlerFactory name="shardHandlerFactory" class="HttpShardHandlerFactory">
    <int name="socketTimeout">${socketTimeout:0}</int>
    <int name="connTimeout">${connTimeout:0}</int>
  </shardHandlerFactory>
</solr>
```

如你所见，solr 配置是 solrcloud 友好的。但是，`<solrcloud>` 元素的出现不代表 solr 实例正在运行 solrcloud 模式。除非启动时指定了 `-DzkHost` 或 `-DzkRun`，否则该元素会被忽略

solr.xml 参数

`<solr>`

`<solr>` 标签没有属性，这是 `solr.xml` 的根元素。下表是每个 xml 元素的子节点

节点	描述
adminHandler	如果使用的话，这个属性应类(继承自 CoreAdminHandler)的全路径名 FQN(Fully qualified name).例如，adminHandler="com.myorg.MyAdminHandler" 将配置自定义 admin handler(MyAdminHandler) 来处理管理请求。如果这个属性未设置，solr 使用默认的 admin handler，org.apache.solr.handler.admin.CoreAdminHandler。该参数的更多信息，参考 wiki 页面
collectionsHandler	如上，可配置自定义的 CollectionHandler 实现
infoHandler	如上，配置自定义 InfoHandler 实现
coreLoadThreads	加载 core 同时分配的线程数
coreRootDirectory	发现 core 的树的根目录，默认是 SOLR_HOME
managementPath	当前未用
sharedLib	所有 core 都共享的公共库目录的路径。这个目录的任何 JAR 文件都会被添加到搜索路径。该路径是相对于 solr 主目录
shareSchema	若设为 true，要确保多个 core 指向同一个 schema 资源文件，指向同一个 IndexSchema 对象。共享 IndexSchema 对象会加速 core 的加载。如果使用这个特性，确保在 schema 文件没有使用 core 的属性
transientCacheSize	有多少个设置为 transient=true 的 core 在最近最少使用的 core 被新的 core 交换前被加载
configSetBaseDir	solr core 的 configsets 目录，默认是 SOLR_HOME/configsets

`<solrcloud>`

除非启动时指定了 `-DzkHost` 或 `-DzkRun`，否则该元素会被忽略

节点	描述
distribUpdateConnTimeout	为集群内部的更新设置底层的 "connTimeout"
distribUpdateSoTimeout	为集群内部的更新设置底层的 "SocketTimeout"
host	solr 用来访问 core 的主机名
hostContext	url 上下文路径
hostPort	solr 访问 core 使用的端口。在默认的 `solr.xml` 文件，设置为 `${jetty.port:8983}`
leaderVoteWait	当 solrcloud 启动，在假定任何没有汇报的 node 已经挂掉之前，等待 node 多久去发现 shard 的所有已知的 replica
leaderConflictResolveWait	当尝试选举一个 shard 的 leader，这个属性设置一个 replica 将要等着看冲突状态的解决的最大时间；临时的冲突状态可能发生在重启时，尤其是 Overseer 所在的主机重启。典型的，默认值 180000 ms 足够冲突得以解决；如果你的 solrcloud 有成百上千的小 collection，你也许需要增加这个值
zkClientTimeout	用于 solrcloud，到 zk 服务器的连接的超时时间
zkHost	在 solrcloud 模式，solr 用来连接到 zk 主机的 url
genericCoreNames	若为 `TRUE` ，node 名字不是基于 node 的地址，而是基于通用名来标识 core。当一个不同的机器接管了 core 将更易理解
zkCredentialsProvider & zkACLProvider	可选参数，如果使用了 ZooKeeper Access Control

`<logging>`

节点	描述
class	记录日志的类。相应的文件必须可用，可在 `solrconfig.xml` 用 `<lib>` 指定
enabled	true/false - 是否启用日志

`<logging><watcher>`

节点	描述
size	缓冲的 log 事件数量
threshold	日志级别，以上的将被记录。例如，使用 log4j 可以被设定为 DEBUG，WARN， INFO 等等

`<shardHandlerFactory>`

自定义 Shard handler

```
<shardHandlerFactory name="ShardHandlerFactory" class="qualified.class.name">
```

由于这是自定义的 shard handler，子元素取决于其实现

在 solr.xml 里的 jvm 系统属性

`solr.xml` 支持 JVM 系统属性值，运行时设定配置选项。语法是 `${属性名:[可选的默认值]}`。可以定义一个在运行时能被覆盖的默认值。如果默认值未设定，那么运行时必须指定属性值，否则 `solr.xml` 会在解析时报错。

启动 JVM 时使用 `-D` 设定的任意属性都可以用在 `solr.xml` 文件。

例如，在下面所示的 `solr.xml` 文件， `socketTimeout` 和 `connTimeout` 值都被设为 '0'。但是，如果你启动 solr 时使用 `"bin/solr -DsocketTimeout=1000"` ， `HttpShardHandlerFactory` 的 `socketTimeout` 选项会用 1000ms 覆盖，同时 `connTimeout` 继续使用默认值 '0'

```
<solr>
  <shardHandlerFactory name="shardHandlerFactory" class="HttpShardHandlerFactory">
    <int name="socketTimeout">${socketTimeout:0}</int>
    <int name="connTimeout">${connTimeout:0}</int>
  </shardHandlerFactory>
</solr>
```

定义 core.properties

core discovery(core 发现/感知)表示创建一个 core 就和磁盘上的 `core.properties` 一样简单(这 tmd 什么比喻...)。`core.properties` 文件是个简单的 JAVA 属性文件，每行是一个 k-v 对，例如 `name=core1`。注意不需要引号

一个最小化的 `core.properties` 文件看上去是这样的(但是，也可以是空的)

```
name=my_core_name
```

core.properties 位置

在 `solr.home` 的子目录下放置名为 `core.properties` 的文件来配置 solr core。树的深度没有限制，可定义的 core 的数量也没有限制。core 可以在树的任何地方，但不能定义在一个已存在的 core 下。即，如下是不允许的

```
./cores/core1/core.properties
./cores/core1/coremore/core5/core.properties
```

这个例子里，将在 core1 停止枚举(应该是说，solr 不会在 core1 的子目录里查看是否有 core.properties 文件存在)

如下是合法滴

```
./cores/somecores/core1/core.properties
./cores/somecores/core2/core.properties
./cores/othercores/core3/core.properties
./cores/extracores/deepertree/core4/core.properties
```

solr 可以被分割为多个 core，每个 core 都有自己的配置和索引。core 可以专注于单个应用或另一个完全不同的，但是所有的管理操作都是通过一个公共的管理界面。你可以创建新的 core，停止 core，甚至用一个 core 替换另一个正在运行的 core，都不需要停止或重启 solr。

如有必要，你的 `core.properties` 文件可以是空的。假设一个空的 `core.properties` 文件，位于 `./cores/core1` (相对于 `solr_home`)，这个案例里，core 的名字假设为 "core1"，instanceDir 将是包含 `core.properties` 文件的目录(`./cores/core1`)，dataDir 将是 `../cores/core1/data` ，等等

> 可不配置 core 来运行 solr

定义 core.properties 文件

最小化的 `core.properties` 文件是空文件，所有的属性都使用默认值。

java 属性文件可以用 "#" 或 "!" 注释一行

下面的表格定义了有效的属性

属性	描述
name	solr core 的名字。当用 CoreAdminHandler 运行命令时用这个名字来指代 core
config	给定的 core 的配置文件的名字，默认是 `solrconfig.xml`
shema	给定的 core 的 schema 文件名，默认是 `schema.xml`，但是注意，如果使用 "managed schema"(默认行为)，那么任何不匹配 `managedSchemaRespurceName` 的属性值都只会读取一次，便转换成 managed shema。
dataDir	core 的数据目录(存储索引的目录)，一个绝对路径或者相对于 `instanceDir` 的目录，默认为 `data`
configSet	配置集的名字，用来配置 core
properties	core 的属性文件的名字。绝对路径或者是相对(`instanceDir`)路径
transient	如为 **true**，core 会在 solr 达到 `transientCacheSize` 时被卸载。如果未指定，默认为 **fasle**。最近最少使用的 core 首先被卸载。不建议在 *solrcloud* 模式设为 ***true***
loadOnStartup	如为 **true**，未指定时的默认值，core 在 solr 启动时加载。不建议在 *solrcloud* 模式设为 ***false***
coreNodeName	只用于 solrcloud，这是承载 replica 的 node 的唯一标识。默认情况下，coreNodeName 自动生成，但是明确设定该属性允许你手工分配一个新 core 来代替一个已存在的 replica。例如：在新机器上用新的主机名或端口，用备份还原来取代一个硬件故障的机器
ulogDir	core(solrcloud) 的更新日志(update log)的绝对或相对路径
shard	分配给这个 core 的 shard (solrcloud)
collection	这个 core 所属的 collection (solrcloud)
roles	solrcloud 将来的参数，或用户用来标记 node 为自用的

SolrCloud

Apache Solr 包含了创建集群来结合容错和高可用的能力，即 **SolrCloud**，提供了分布式索引和查询的能力，支持如下的特性

- 集中配置整个集群
- 查询的自动负载均衡及故障切换
- 集成 Zookeeper 来协调和配置集群

SolrCloud 是一个灵活的、分布的查询和索引，无需中心节点来分配节点、分片和复制。Solr 使用 ZooKeeper ，基于配置文件和 schemas 来管理这些场景。文档可以发送到任何服务器上，ZooKeeper 总能找到它。

SolrCloud 入门

SolrCloud 设计目标是提供一个高可用，容错的环境，将索引过的内容和查询请求分发到多台服务器上。

这是一个数据被组织为多份或多片，存储在多个服务器上，复制副本来以提供扩展性和容错的系统；使用 ZooKeeper 服务器管理所有属性，以使索引和查询请求可被正确路由。

本小节解释了 SolrCloud 及其内部工作细节，但是在你深入之前，最好是搞清楚你要达成什么目标。本页提供了一个简单的启动 Solr cloud 模式的操作说明，使你能对solr 分片在索引和查询时如何互相协作有个感性认识。为此，我们将用一个简单的例子，在单个服务器上配置 SolrCloud，显然这不是一个真正的，有多台服务器或虚拟机的生产环境。

SolrCloud 示例

交互式启动

`bin/solr` 脚本使 SolrCloud 入门很简单，它引导你 cloud 模式启动 solr 并添加 collection，输入下述命令开始

```
$ bin/solr -e cloud
```

这会启动一个交互式的会话，引导你创建一个使用内置的 ZooKeeper 的、简单的 SolrCloud 集群。脚本询问你想要在本地集群创建几个节点（node），默认是 2 个

```
Welcome to the SolrCloud example!

This interactive session will help you launch a SolrCloud cluster on your local workst
ation.
To begin, how many Solr nodes would you like to run in your local cluster? (specify 1-
4 nodes) [2]
```

脚本支持启动 4 个节点，但我们推荐 2 个。这些节点在一台机器上，使用不同的端口来模拟在不同机器上。接下来，脚本提示你每个节点绑定的端口

```
Please enter the port for node1 [8983]
```

为每个节点选择可用的端口，默认第一个节点是 8983，第二个节点是 7574。脚本将顺序启动各节点，并显示启动服务器的命令，如下

```
solr start -cloud -s example/cloud/node1/solr -p 8983
```

第一个节点将同时启动一个内置的 ZooKeeper，绑定 9983 端口，第一个节点的 solr home 目录为 `example/cloud/node1/solr`，由 `-s` 选项指定

在集群的所有节点都启动后，脚本提示你输入要创建的 collection 的名字

```
Please provide a name for your new collection: [gettingstarted]
```

默认的名字是 "gettingstarted"，你也可以选择一个更恰当的名字

然后，脚本会提示你这个 collection 跨越的分片的数量。

然后，脚本会提示你每个分片的副本数量。

最后，脚本会提示你的 collection 的配置目录的名字，你可以选择 **basic_configs**，**data_driven_schema_configs**， 或 **sample_techproducts_configs**。配置目录是从 `server/solr/configsets/` 拉取的，所以你可以事先审核这些配置。如果你要为你的文档定义 schema，且需要一些灵活性，默认配置 **data_driven_schema_configs** 是很有用的。

此刻，你在本地的 Solr 集群里新建了一个 collection。为了验证这一点，可以执行 status 命令

```
$ bin/solr status
```

如果在这个过程中，碰到了任何的错误，在 `example/cloud/node1/logs` 和 `example/cloud/node2/logs` 查看 solr 的日志文件。

可以在 Solr 管理界面的 cloud 面板查看你的 collection 是如何发布到集群的：http://localhost:8983/solr/#/~cloud，Solr 也提供一个 healthcheck 命令来为 collection 执行简单的诊断

```
$ bin/solr healthcheck -c gettingstarted
```

healthcheck 命令收集 collection 里每个副本的基本信息，例如文档数量，当前状态（active，down，等），地址（副本在集群的位置）

现在可以用 post 工具将文档添加到 SolrCloud

要停止 solr，使用 stop 命令，如下

```
$ bin/solr stop -all
```

-noprompt 启动选项

以默认配置启动 SolrCloud，而不是交互式会话模式

```
$ bin/solr -e cloud -noprompt
```

重启节点

使用 `bin/solr` 脚本重启 SolrCloud 节点，例如，重启端口为 8989 的节点 1

```
$ bin/solr restart -c -p 8983 -s example/cloud/node1/solr
```

重启端口为 7574 的节点 2

```
$ bin/solr restart -c -p 7574 -z localhost:9983 -s example/cloud/node2/solr
```

注意，重启节点 2 时，你需要指定 ZooKeeper 的地址（-z localhost:9983），这样节点 2 才能加入到集群

向集群添加节点

向一个已存在的集群添加节点，有一点高级。。。和更多一点对 solr 的理解。当你使用 startup 脚本启动了一个 SolrCloud 集群后，可以用下面的命令添加节点

```
$ mkdir <solr.home for new solr node>
$ cp <existing solr.xml path> <new solr.home>
$ bin/solr start -cloud -s solr.home/solr -p <port num> -z <zk hosts string>
```

注意，上面的命令需要你创建 solr home 目录，你需要复制 `solr.xml` 到 `solr_home` 目录，或在 ZooKeeper `/solr.xml`

示例如下

```
$ mkdir -p example/cloud/node3/solr
$ cp server/solr/solr.xml example/cloud/node3/solr
$ bin/solr start -cloud -s example/cloud/node3/solr -p 8987 -z localhost:9983
```

上面的命令将在 8987 端口启动一个 solr 节点，其 home 目录为 `example/cloud/node3/solr`。新的节点将日志文件写入到 `example/cloud/node3/logs`

SolrCloud 如何工作

SolrCloud 关键概念

一个 SolrCloud 集群由一些物理概念之上的逻辑概念组成

逻辑概念

- 一个集群(cluster)可以承载多个文档(document)的集合(collection)
- 一个集合(collection)可以被划分为多个分片(shard)，每一个分片(shard)都包含了集合(collection)的全部文档(document)的子集
- 决定集合(collection)划分为几个分片(shard)的是
  - 集合(collection)包含多少文档(document)，这里说的是理论上限
  - 并发的查询请求数量

物理概念

- 一个集群(cluster)由一个或多个节点(node)组成，每个节点(node)都运行着 Solr 服务进程实例
- 每个节点(node)可以承载多个核(core)
- 集群(cluster)中的每个核(core)是一个逻辑分片(shard)的物理副本(replica)
- 集合(collection)里的每个副本(replica)使用相同的配置
- 决定每个分片(shard)里有几个副本(replica)的是
  - 构建集合(collection)的冗余度，及集群(cluster)里的某几个节点(node)不可用时的容错能力
  - 在重负载时，可以同时处理的查询请求，这里说的是理论上限

在 SolrCloud 里分片和索引数据

当你的数据对一个节点来说太大时，你可以创建多个分片，将数据分散存储于其中。

在 SolrCloud 之前，Solr 支持分布式查询，即一个查询跨越多个分片，整个索引上执行而不会遗漏任何文档。所以将核划分成分片并非 SolrCloud 独有的概念。但是，有几个需要 SolrCloud 改善的分布式的问题：

1. 将 core 分片是手工滴
2. 不支持分布式索引，这意味着你要把文档发送到一个特定的分片，而 Solr 自己不知道要把文档发送到哪个分片
3. 没有负载均衡或故障切换，如果你有大量的查询，你需要知道发送给谁

SolrCloud 解决了这些问题。它支持分布式索引和查询，ZooKeeper 提供了故障切换和负载均衡。此外，每个分片可以有多个副本来提升健壮性

在 SolrCloud 里没有主(master)或从(slave)，取而代之的是，领导(leader)和副本(replica)。领导是自动选举的，初始是基于先来先服务原则，然后就是基于 ZooKeeper 的选举

如果一个领导挂了，它的某个副本会被自动选为新的领导。如果每个节点都启动了，会分配给最少副本的分片，如果有个限制，会分配给 shard id 最小的分片。（这一段貌似是说哪个分片做 leader）

当一个文档发送给机器来索引，系统首先判断该机器是否领导或副本

- 如果该机器是个副本，文档转发给领导来处理
- 如果该机器是个领导，SolrCloud 判断文档应去往哪个分片，转发文档到分片的领导，为这个分片做索引，并且转发索引笔记到自己和所有副本

文档路由

创建一个 collection 时，用 `router.name` 参数来指定路由器实现。如果你使用 `"compositeId"` 路由器，可以在文档 id 中添加前缀来计算 hash 值，solr 使用这个值来决定文档被发送到哪个分片去索引。前缀可以是任何你喜欢的东东（不一定要用分片的名字），但应该是一致的。例如，如果你想要把一个客户的文档放在一起，可以把客户的 id 或名字作为前缀。如果客户名字是 "IBM"，有个文档 id 为 "12345"，可以把前缀插入 id 字段，即 "IBM!12345"，感叹号 ("!") 很重要，它表明了用于决定文档所处分片的前缀。

然后，在查询时，使用 `_route_` 参数将前缀包含在查询中（例如，`q=solr&_route_=IBM!`）来定位分片。在某些场景下，这会改善查询性能，因为在查询所有分片时不用考虑网络因素

> `_route_` 参数取代了 `shard.keys`，后者已废弃并将在 Solr 的后续版本被移除

`compositeId` 路由器支持 2 层前缀。例如，一个前缀首先路由到区域，然后是客户："USA!IBM!12345"

另一个用例，客户 "IBM" 有很多文档，你想要把他们分割到多个分片。语法是这样的："shard_key/num!document_id"，/num 是用于 hash 的shard_key 的比特位的数量

所以，"IBM/3!12345" 会为 shard_key 占用 3 位，文档 id 占用 29 位。这样就将分片划分为 8 份。如果 num 为 2，那么就是将文档划分为 4 份。

查询时，用 `_route_` 参数来指定分片，如 `q=solr&_route_=IBM/3!`

如果不想影响文档如何存储，就不要在文档 id 里指定前缀

如果创建 collection 时，使用 "implicit" 路由器，`router.field` 参数可以指定一个字段来识别文档属于哪个分片，木有这个字段的文档会被拒绝，同样可以用 `_route_` 参数来命名指定的分片

切分分片

在 SolrCloud 里创建 collection 时，你要决定初始的分片数目。很难确定需要多少个分片，特别是需求随时会变化，并且事后发现数目不对的代价很高，牵涉到创建新的核和重新索引所有数据

分片要使用 Collection API，当前允许将 1 个分片切分为 2 份。已存在的分片依然保留，所以这个切分动作实际上是创建了 2 个副本作为分片。当你准备好时，可以删除老的分片。

忽略客户端提交

大多数场景下，在运行 SolrCloud 时，索引客户端不应发送明确的提交请求。相反的，应该用 `openSearcher=false` 配置自动提交，自动软提交，以使最新的更新能在查询中可见。这能确保在及群里自动提交是按预定计划发生的。为了强调客户端不应该明确提交的策略，应该更新所有索引数据到 SolrCloud 的客户端。不过这并非总是可行的，所以 Solr 提供了 IgnoreCommitOptimizeUpdateProcessorFactory，允许你忽略明确的提交，并且/或者优化客户端请求而不需要重构客户端代码。要激活这个请求处理器，你需要将下面内容添加到 solrconfig.xml

```
<updateRequestProcessorChain name="ignore-commit-from-client" default="true">
  <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
    <int name="statusCode">200</int>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.DistributedUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

如上面的例子，处理器返回 200 给客户端，但是会忽略提交/优化请求。注意你需要把 SolrCloud 必需的隐含处理器串联起来，毕竟这个自定义的处理器链会取代默认的处理器链

下面的例子会抛出一个代码为 403 的异常，及定制的错误消息

```
<updateRequestProcessorChain name="ignore-commit-from-client" default="true">
  <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
    <int name="statusCode">403</int>
    <str name="responseMessage">Thou shall not issue a commit!</str>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.DistributedUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

最后，你也可以配置为忽略优化，让提交通过

```
<updateRequestProcessorChain name="ignore-optimize-only-from-client-403">
  <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
    <str name="responseMessage">Thou shall not issue an optimize, but commits are OK!</
str>
    <bool name="ignoreOptimizeOnly">true</bool>
  </processor>
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

分布式请求

当一个 Solr 节点接收到搜索请求，这个请求在后台被路由到被搜索集合的某个分片的一个副本上。这个被选中的副本担当一个聚合器：创建内部请求到集合的所有分片的某个随机的副本，协调所有响应，发出任何必要的后续内部请求（例如，改善 facet 值，或请求额外的存储字段），构建返回给客户端的最终响应

限制哪些分片(shards)被查询

使用 SolrCloud 的一个好处是，多个分片组成的大集合，但在某些案例里，你只对分片的子集返回的结果感兴趣。你可以选择查询所有还是部分分片。

在集合的所有分片上查询，看上去眼熟，就好像没有运行 SolrCloud（意思是在所有分片上查询的语法和单核）

```
http://localhost:8983/solr/gettingstarted/select?q=*:*
```

另一方面，如果你想只在一个分片上搜索，你可以指定分片的逻辑 id，如

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=shard1
```

如果想在一组分片上搜索，可以一起指定

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=shard1,shard2
```

在上面的例子里，分片 id 用来选择该分片的一个随机的副本

你也可以明确指定想要的副本取代分片 id

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=localhost:7574/solr/ge
ttingstarted,localhost:8983/solr/gettingstarted
```

或者你也可以用管道符(|)指定一个分片的副本列表来选择

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=localhost:7574/solr/ge
ttingstarted|localhost:7500/solr/gettingstarted
```

当然，你可以指定一个逗号分隔的分片列表，每一个都由管道符(|)分隔多个副本，下面例子里，2 个分片被查询，第一个分片是随机选择的副本，第二个分片是在管道符(|)限定的副本里随机一个

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=shard1,localhost:7574/solr/gettingstarted|localhost:7500/solr/gettingstarted
```

配置 ShardHandlerFactory

你可以在 Solr 分布式查询里配置并发和线程池，更好的细粒度控制和调整，默认的配置有利于吞吐量。

配置标准的处理器(handler)，在 solrconfig.xml 里按如下例子配置

```
<requestHandler name="standard" class="solr.SearchHandler" default="true">
  <!-- other params go here -->
  <shardHandler class="HttpShardHandlerFactory">
    <int name="socketTimeOut">1000</int>
    <int name="connTimeOut">5000</int>
  </shardHandler>
</requestHandler>
```

可用的参数列表

参数	默认值	含义
`socketTimeout`	0 (使用操作系统默认值)	socket 等待的时间，毫秒
`connTimeout`	0 (使用操作系统默认值)	接受绑定或连接 socket 的时间，毫秒
`maxConnectionsPerHost`	20	分布式查询时每个分片最大的并发连接数
`maxConnections`	10000	分布式查询时总的并发连接上限数
`corePoolSize`	0	分布式查询时线程数下限
`maximumPoolSize`	Integer.MAX_VALUE	分布式查询时线程数最大值
`maxThreadIdleTime`	5	线程在回收前等待的时间，秒
`sizeOfQueue`	-1	如果指定，线程池将用一个队列来替代直接切换缓冲区。高吞吐量的系统设为 -1 来使用直接切换缓冲区，希望更好的延迟的系统配置一个合理的队列大小来处理变化的请求
`fairnessPolicy`	false	为 true，则分布式查询使用先进先出方式来消费，false 则吞吐量是延迟处理

配置 statsCache(分布式 IDF)

为计算关联度（匹配度？）需要统计文档和词条。Solr 提供 4 种开箱即用的实现

- `LocalStatsCache`：只用局部词条和文档来统计关联度。对于分片上的统一术语，这个很不错
- `ExactStatsCache`：使用全局值来统计文档频率
- `ExactSharedStatsCache`：和 ExactStatsCache 一样，但是相同词条的后续请求不被统计
- `LRUStatsCache`：用一个 LRU(Least Recently Used，最近最少使用) 算法的 cache 来保存全局统计

在 `solrconfig.xml` 文件里，使用 `<statsCache>` 标签来配置。示例：

```
<statsCache class="org.apache.solr.search.stats.ExactStatsCache"/>
```

避免分布式死锁

每个分片接受到顶层查询请求，然后创建子查询请求到其他的所有分片。应确保处理请求的最大线程数大于来自顶层的客户端请求和来自其他分片的请求数。否则就可能导致分布式死锁

例如，死锁可能发生在这样的场景：2 个分片，每个都只有一个线程来处理 HTTP 请求。每个线程都接收到一个顶层请求，并创建一个子查询请求到另一个线程。由于没有剩余的线程来处理请求，新来的请求将被阻塞直到其他处理中的请求完成，但是这些请求将无法完成，因为它们正在等待子查询请求的结果。确保 Solr 配置了足够的线程数，以避免类似的死锁场景。

优先本地分片

Solr 允许你传递一个可选的逻辑参数 `preferLocalShards` 来表明一个分布式查询应该尽可能优先本地的分片副本。也就是说，如果一个查询包含 `preferLocalShards=true` ，那么查询控制器会查找本地的副本来处理查询，而不是在集群里随机选择一个副本。这对于要返回很多字段或大的字段的查询请求很有用，因为本地查询避免了在网络上传输大量数据。而且，这个特性对于有性能问题的副本影响最小化也有用，健康的副本选中该问题副本的可能性也降低了

读写侧容错

SolrCloud 支持伸缩性，高可用和读写容错。这意味着，如果你有个大的集群，总是可以发送请求到集群：只要有可能，读请求总是能返回结果，即使某些节点宕机；写请求将收到回执。你不会丢失数据

读容错

SolrCloud 集群里，读请求是集合里的节点来均衡负载的。你仍然需要一个外部的负载均衡器来，或者一个智能客户端，能明白如何读 ZooKeeper 里的元数据来发现应向哪个节点发送请求。

Solr 提供了一个 Java 的客户端 CloudSolrClient

zkConnected

一个 Solr 节点会在它能联系到每个分片的至少一个副本时返回查询结果，甚至是在它收到请求时无法联系到 ZooKeeper。为了容错这通常是优先的做法，但是可能会得到错误的结果或者脏数据，如果整个集合的结构发生了大的变化而该节点还没有能从 ZooKeeper 得到通知。例如，新增或移除了分片，或者某个分片切分成子分片

每个查询结果都包含一个 `zkConnected` 头来指示节点是否在处理请求时连接到 ZooKeeper

```
{
  "responseHeader": {
    "status": 0,
    "zkConnected": true,
    "QTime": 20,
    "params": {
      "q": "*:*"
    }
  },
  "response": {
    "numFound": 107,
    "start": 0,
    "docs": [ ... ]
  }
}
```

shards.tolerant

如果一个或多个分片不可用，Solr 默认的行为是请求失败。但是，很多场景部分结果是可接受的，所以 Solr 提供一个逻辑参数 `shards.tolerant` (默认 `false`)。如果 `shards.tolerant=true` ，会返回部分结果。如果返回的结果不包含所有分片，返回头里有一个特殊的标识 `partialResults` 。客户端可以同时指定 `shards.info` 参数来获取更多的细节

```
{
  "responseHeader": {
    "status": 0,
    "zkConnected": true,
    "partialResults": true,
    "QTime": 20,
    "params": {
      "q": "*:*"
    }
  },
  "response": {
    "numFound": 77,
    "start": 0,
    "docs": [ ... ]
  }
}
```

写容错

SolrCloud 被设计为复制文档来确保数据的冗余，并允许发送更新请求到集群内任何节点。节点会判断它是否承载了分片的领导(leader)，如果没有的话会转发请求给领导(leader)，领导会转发请求到所有存在的副本，使用版本控制来确保所有副本都有最新的版本。如果领导(leader)宕机，某个其他副本会取代它。这个架构保证你的数据能在灾难时恢复，即使你在使用近实时搜索。

Recovery

每个节点都有一个事务日志，每一次对内容或组织的修改都会被记录。这个日志被用来判断节点的哪些内容应该包含到副本。当一个新副本被创建，它会引用领导(leader)和事务日志来明白要包含哪些内容。如果失败的话会重试。

事务日志是由更新记录组成，提供了索引的健壮性，如果索引被中断，它可以重做未提交的索引

如果领导(leader)宕机，它可能已发送了请求到某些副本而未发送给其他副本。当一个新的领导(leader)选出，会运行一个同步进程，同步其他副本数据。如果运行成功，所有都会一致，领导(leader)注册为活跃的，如果一个副本无法同步，系统会要求一次全量复制

核在重新加载 schema，某些成功了，另一些还没有成功，会导致一个更新失败，领导(leader)将告诉节点并启动一个恢复过程

Achieved Replication Factor

当使用一个大于 1 的复制因子，一个更新请求可能在分片领导(leader)上成功而在一个或多个副本上失败。例如，一个集合有 1 个分片，复制因子为 3，这样你就有 1 个分片领导(leader)和 2 个额外的副本，如果一个更新请求在领导(leader)上成功，但是在副本上都失败，不论是因为什么原因，这个更新请求在客户端看来是成功的。副本恢复后会从领导(leader)同步错过的数据。

在幕后，这表明 Solr 仅在一个节点上(这里是领导(leader))进行了更新。Solr 在更新时，支持可选的参数 `min_rf` ，使服务器返回成功复制因子。上例中，如果客户端包含了 `min_rf >= 1` ，Solr 将在结果的头信息里返回 `rf=1` ，因为只有领导(leader)更新成功。`min_rf` 参数并不强制 Solr 必须最少在几个副本上更新，因为 Solr 并不支持在更新成功的副本上回滚。这个参数仅表示客户端希望知道更新请求的成功复制因子是什么。

在客户端，如果成功复制因子小于可接受的水平，可以为这个降级的状态做一个额外的评估。例如，客户端可以记录日志，当集合降级时，哪些更新请求被发送，在问题解决后重新发送这些更新请求。简而言之，`min_rf` 是集合处于一种降级状态时，对于更新请求返回给客户端的一种可选的警告机制

SolrCloud 配置和参数

创建外部 ZooKeeper

虽然 Solr 附带了一个 ZooKeeper，你应该考虑使用内置的 ZooKeeper 的不利之处：关闭一个多余的 Solr 实例将同时关闭其附带的 ZooKeeper，而这个 ZooKeeper 也许并非多余。由于 ZooKeeper 集群在任何时候都需要至少其服务器数量的一半以上的节点在运行，这可能会是一个麻烦。

解决方案是一个外部的 ZooKeeper。

> 需要多少 **ZooKeeper?**
>
> 要保证一个 ZooKeeper 服务可用，需要多数的机器能彼此通信。如果可以容忍有 F 台机器处于故障状态，那么你至少要有 2*F+1 台机器。因此，3 台机器可以有 1 台故障，5 台机器可以有 2 台故障。要注意 6 台机器只能有 2 台故障，因为 3 台机器并非多数。
>
> 基于这个理由，ZooKeeper 部署一般是奇数台机器。

当计划需要配置多少 zk 节点时，记住主要的原理是保证有多数的服务器在提供服务。通常建议有奇数台 zk 服务器组成你的 zk 服务。例如，如果你只有 2 个 zk 节点且其中一个宕机，50% 的可用服务器并非多数，所以 zk 将不能提供服务。但是，如果你有 3 个 zk 节点且其中一个宕机，你有 66% 的可用服务器，zk 将保持正常直到你修复宕机的节点。如果你有 5 节点，你可以在 2 个节点宕机的情况下继续运作。

下载 Apache ZooKeeper

第一步是下载 Apache ZooKeeper，下载地址

> 使用一个独立的 zk 时，你应注意保持 zk 的版本和 Solr 的要求一致。因为你的 zk 不会随着 Solr 的升级而自动升级版本。(相对于使用 Solr 内置的 zk 而言)
>
> Solr 当前使用的 zk 版本为 v3.4.6

安装配置单个 ZooKeeper

创建实例

解压文件到指定的目录

配置实例

接下来是配置你的 zk 实例：创建如下的文件 `<ZOOKEEPER_HOME>/conf/zoo.cfg`，并添加如下的内容

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
```

参数如下

tickTime：以毫秒为单位，一个 tick 的时长。tick 是 zk 的一个时间单位，在 zk 检测服务器是否正常工作时(心跳)，最小的会话超时时间是 2 tick。

dataDir：zk 用这个目录来保存数据。该目录初始应该为空

clientPort：Solr 用来访问 zk 的端口

一旦这个文件就绪，你就已准备好启动 zk 实例了。

运行实例

使用 `ZOOKEEPER_HOME/bin/zkServer.sh` 脚本来运行 zk 实例，命令为 `zkServer.sh start`

zk 提供了其他很多强大的配置，但是深入研究这些内容已超出了这个说明的范围。对于本文的示例，默认的配置已足够好了。

在 Solr 中使用 zk 实例

在 Solr 里使用 zk 实例很简单：在 bin/solr 脚本使用 `-z` 参数。例如，要在 solr 里指向 你启动在 2181 端口的 zk，只需如此

随同已运行在 2181 端口的 zk 启动 `cloud` 例子(其他均为默认选项)

```
bin/solr start -e cloud -z localhost:2181 -noprompt
```

添加一个指向 2181 端口的 zk 的节点

```
bin/solr start -cloud -s <path to solr home for new node> -p 8987 -z localhost:2181
```

注意：当你不是在启动一个 solr 示例，确保在创建集合前已将配置上传到 zk

关闭 ZooKeeper

要关闭 zk，使用 stop 命令来运行 zkServer：`zkServer.sh stop`

安装配置 ZooKeeper 集群

对于集群来说，只需要比单个 zk 更加小心一些

不同之处在于，你需要首先配置多个 zk 互相认知和交谈。所以你的 `zoo.cfg` 看上去是这样滴

```
clientPort=2181
initLimit=5
syncLimit=2
dataDir=/var/lib/zookeeperdata/1
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

这里有 3 个新的参数

initLimit：以 tick 为单位，follower 连接到 leader 并同步数据的时间。本例里，是 5 个 tick，每个 2000 毫秒，所以服务器将等待 10 秒连接到 leader 并同步。

syncLimit：以 tick 为单位，follower 与 zk 同步数据的时间。如果 follower 离 leader 太远，将会被剔除。这个貌似是 leader 与 follower 之间的心跳检测超时时间，心跳是 leader 主动发送给 follower 的。

server.X：集群里的所有服务器的 id 和位置信息。Server ID 必须额外的保存在 `<dataDir>/myid` 文件里，且在每个 zk 实例的 `dataDir` 目录下。ID 用来标识每个服务器，在本例中，第一个实例应该有 `/var/lib/zookeeperdata/1/myid` 文件，内容为 '1'。右边可以配置两个端口，第一个端口用于F和L之间的数据同步和其它通信，第二个端口用于Leader选举过程中投票通信。

现在，随同 solr 你需要创建全新的目录来运行多个实例，为了一个 zk 实例你所要做的所有工作，即便是为了测试而运行在同一个机器，就是一个新的配置文件。为了完成这个例子，你将创建 2 个配置文件

`<ZOOKEEPER_HOME>/conf/zoo2.cfg` 内容如下

```
tickTime=2000
dataDir=c:/sw/zookeeperdata/2
clientPort=2182
initLimit=5
syncLimit=2
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

你还需要创建 `<ZOOKEEPER_HOME>/conf/zoo3.cfg`

```
tickTime=2000
dataDir=c:/sw/zookeeperdata/3
clientPort=2183
initLimit=5
syncLimit=2
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

最终，在每个 `dataDir` 目录创建你的 `myid` 文件，以便每个服务器都知道自己是哪个实例。`myid` 文件里的 id 必须匹配 'server.X' 的定义。这样，上例中的 zk 实例(或机器) 'server.1'，应该有一个 `myid` 文件包含了内容 1。`myid` 文件可以是任意的 1 至 255 之间的整数，且必须匹配 `zoo.cfg` 文件里分配的服务器 ID。

要启动服务，如下引用配置文件

```
cd <ZOOKEEPER_HOME>
bin/zkServer.sh start zoo.cfg
bin/zkServer.sh start zoo2.cfg
bin/zkServer.sh start zoo3.cfg
```

一旦这些服务器启动，就可以在 solr 里引用他们

```
bin/solr start -e cloud -z localhost:2181,localhost:2182,localhost:2183 -noprompt
```

ZooKeeper 连接安全

你也许想要保证 Solr 和 ZooKeeper 之间的通信安全，参考 ACL(ZooKeeper Access Control)

用 ZooKeeper 管理配置文件

使用 SolrCloud，你的配置文件保存在 zk。这些文件用以下方式之一上传到 zk

- 用 bin/solr 脚本启动 SolrCloud 示例应用
- 用 bin/solr 脚本创建一个集合(collection)
- 直接将配置上传到 zk

启动引导

当你用 bin/solr -e cloud 第一次启动 SolrCloud时，相关的配置自动上传到 zk 并连接到最近创建的集合(collection)

下面的命令启动 SolrCloud，连接到默认的集合(gettingstarted)，并上传默认的配置(data_driven_schema_configs)

```
$ bin/solr -e cloud -noprompt
```

使用 bin/solr 创建集合(collection)时，用 `-d` 选项直接上传一个配置目录

```
$ bin/solr create -c mycollection -d data_driven_schema_configs
```

create 命令会上传 `data_driven_schema_configs` 配置目录的一个副本到 zk 的 `/configs/mycollection`

一旦一个配置目录被上传到 zk，你就可以用 zkCLI(ZooKeeper Command Line Interface) 来更新它们。

> 最好是将这些文件置于版本控制之下

用 zkcli 或 SolrJ 上传配置

在生产环境，也可以用 solr 的 zkcli.sh 脚本，或者 java 方法 CloudSlorClient.updataConfig() 上传配置

下面的命令使用 zkcli 脚本上传一个新的配置

```
$ sh zkcli.sh -cmd upconfig -zkhost <host:port> -confname <name for configset> -solrho
me <solrhome> -confdir <path to directory with configset>
```

管理 SolrCloud 配置文件

要更新或更换 SolrCloud 配置文件

1. 从 zk 下载最新的配置文件，使用源码控制的检出
2. 进行变更
3. 提交更新后的文件到源码控制
4. 将变更推送到 zk
5. 重载集合(collection)以使变更生效

在第一个集群启动前准备 ZooKeeper

如果你要和其他应用程序共享一个 zk 实例，你应该使用 *chroot*。

集群需要配置，且部分配置对集群的正常工作是关键性的，你应该在第一次启动 Solr 集群之前就把这些配置上传到 zk。例如如下的配置文件(不限于)

`solr.xml` ， `security.json` ， `clusterprops.json`

例如，你要把 `solr.xml` 保存到 zk 以避免将其复制到每个节点的 `solr_home` 目录，你可以用 `zkcli.sh` 工具来推送它到 zk

```
zkcli.sh -zkhost localhost:2181 -cmd putfile /solr.xml /path/to/solr.xml
```

ZooKeeper 访问控制

ACLs:ZooKeeper access control lists

关于 ZooKeeper ACLs

SolrCloud 使用 ZooKeeper 来共享信息和协调

默认情况下，Solr 的 zk 节点是开放的，不安全的，不需认证的。本小节讲述了如何配置 Solr 来对其在 zk 上的创建的内容进行访问控制，及如何告诉 Solr 在访问这些内容时需要的认证信息。如果要在你的 zk 节点上使用 ACLs，你必须激活这个功能

变更 zk 上的 Solr 相关内容，可能会对 SolrCloud 集群造成损害。例如

- 变更配置可导致 Solr 失败或意料之外的行为
- 将集群状态信息错误的变更或不一致，可导致 SolrCloud 集群奇怪的行为
- 添加一个删除集群的工作在观察者(Overseer)上运行，可导致集群的数据被删除

你可能想要在 Solr 上开启 ZooKeeper ACLs，以授权你不信任的实体访问你的 zk，或降低恶意操作风险，例如

- 你系统上的恶意软件
- 使用同一个 zk 集群的其他应用程序

你可能想要限制读取操作，如果你觉得 zk 上有些内容不应对所有人开放

保护 ZooKeeper 本身意味着很大的不同。本小节是关于保护 **zk** 上的 **Solr** 相关内容的。zk 的内容基本上是在硬盘，部分在 zk 进程的内存里。本小节也不是关于保护存储在硬盘上的 **zk** 数据或者 **zk** 进程级别的数据。-这是 ZooKeeper 要对付的

这些内容也可以对于使用 zk api 的外来者也是可用的。外部进程可以连接到 zk 并创建，更新，删除，读取内容；例如，一个 SolrCloud 节点想要创建，更新，删除，读取，一个 SolrJ 客户端想要读取。

如何启用 ACLs

你想要做到

1. 控制 Solr 使用该证书来连接 zk，证书是用来获得 zk 上的操作许可的
2. 控制哪些 ACLs 可以被 Solr 添加到它创建的 zk 节点
3. 控制外部访问，以避免开启一个 Solr 节点时不得不修改或重编译它

Solr 节点，客户端，工具(如 ZkCLI)总是使用 java 类 SolrZkClient 来处理 zk 的工作。

控制证书

在 `solr.xml` 的 `<solrcloud>` 节使用 `zkCredentialsProvider` 属性来指定类名字或类路径以控制使用哪个证书提供者，类需要实现下面的接口

```
package org.apache.solr.common.cloud;

public interface ZkCredentialsProvider {
  public class ZkCredentials {
    String scheme;
    byte[] auth;
    public ZkCredentials(String scheme, byte[] auth) {
      super();
      this.scheme = scheme;
      this.auth = auth;
    }
    String getScheme() {
      return scheme;
    }
    byte[] getAuth() {
      return auth;
    }
  }
  Collection<ZkCredentials> getCredentials();
}
```

Solr 调用给定的证书提供者的 `getCredentials()` 方法来决定使用哪个证书。如果未配置提供者，会使用默认的实现，`DefaultZkCredentialsProvider`

开箱即用的实现

你可以自己实现，但 Solr 附带了 2 个实现

- `org.apache.solr.common.cloud.DefaultZkCredentialsProvider`：它的 `getCredentials()` 方法返回一个空的 list，表示没有可用的证书。如果没有在 `solr.xml` 里配置提供者，默认就是这个
- `org.apache.solr.common.cloud.VMParamsSingleSetCredentialsDigestZkCredentialsProvider`：它让你用系统属性来自定义证书。它支持最多一组证书
  - schema 为 "digest". 用户名和密码由系统变量 " `zkDigestUsername` " and " `zkDigestPassword` " 分别定义。如果用户名和密码都提供的话，这组证书通过 `getCredentials()` 方法返回。
  - 如果上述这组证书没有返回，将会使用默认的行为返回一个空列表，即 `DefaultZkCredentialsProvider`

控制 ACLs

在 `solr.xml` 的 `<solrcloud>` 节使用 `zkACLProvider` 属性来指定类名字或类路径以控制哪些 ACLs 将被添加，类需要实现下面的接口

```
package org.apache.solr.common.cloud;

public interface ZkACLProvider {
  List<ACL> getACLsToAdd(String zNodePath);
}
```

当 Solr 要添加一个新的 zk 节点(znode)，调用给定的 acl 提供者的 `getACLsToAdd()` 方法来决定哪些 ACLs 添加到这个 znode。如果未配置提供者，使用默认的实现，`DefaultZkACLProvider`

开箱即用的实现

你可以自己实现，但 Solr 附带了

- `org.apache.solr.common.cloud.DefaultZkACLProvider`：返回只有一个元素的列表：`zNodePath`。这个唯一的 ACL 记录 是开放且不安全的。如果未在 `solr.xml` 里配置提供者，这就是默认值
- `org.apache.solr.common.cloud.VMParamsAllAndReadonlyDigestZkACLProvider`：让你用户系统属性自定义 ACLs。它的 `getACLsToAdd()` 实现不使用 `zNodePath`，所以所有的 znode 得到相同的 ACLs，可添加如下的选项
  - 可以做任何事的用户
    - 权限为 "`ALL`" (相当于所有的 `CREATE`，`READ`，`WRITE`，`DELETE`，`ADMIN`)，且 schema 为 "digest"
    - 用户名和密码分别用系统属性 "`zkDigestUsername`" 和 "`zkDigestPassword`" 定义
    - 除非用户名和密码都提供，否则该 ACL 不会被添加到 ACLs 里
  - 只读的用户
    - 权限为 "`READ`"，且 schema 为 "digest".
    - 用户名和密码分别用系统属性 "`zkDigestUsername`" 和 "`zkDigestPassword`" 定义
    - 除非用户名和密码都提供，否则该 ACL 不会被添加到 ACLs 里

    如果上述的 ACLs 都没有添加到列表，那么 `DefaultZkACLProvider` 的空 ACL 列表将作为默认值

注意系统属性名和证书提供者 `VMParamsSingleSetCredentialsDigestZkCredentialsProvider` 重名了。这是为了让这 2 个提供者以一种可能的方式协作：我们总是用 2 种限制来保护对内容的访问——一个管理员用户和一个只读的用户，同时我们总是用这个管理员用户连接到 zk，这样我们就可以在内容和 我们自己创建的 znode 上做任何事了

你可以将只读证书授予 SolrCloud 集群的客户，例如 SolrJ 客户端。它们将能够读取任何需要的内容，但不能修改任何内容。

变更 ACL Schemes

在你的 Solr 集群的运行生命期，也许想要从一个不安全的 zk 迁移到一个安全的 zk。变更 `solr.xml` 里的 `zkACLProvider` 配置能确保最新创建的节点是安全的，但不能保护已存在的数据。要修改所有存在的 ACLs，你可以用 `ZkCLI -cmd updateacls /zk-path`

只有你的 SolrCloud 集群停止时，才能变更 zk 里的 ACLs。强行在 Solr 运行时变更会导致状态不一致和一些节点不可达。要配置新的 ACLs，运行 ZkCli，附带如下 vm 属性：`-DzkACLProvider=...`，`-DzkCredentialsProvider=....`

- 证书提供者必须在当前节点上有管理员权限。如果省略，该操作将不使用证书(适用于不安全的配置)
- ACL 提供者将用来计算新的 ACLs。如果省略，该操作会将所有权限赋予任意用户，移除现有的安全控制

你可以用之前介绍过的 `VMParamsSingleSetCredentialsDigestZkCredentialsProvider` 和 `VMParamsAllAndReadonlyDigestZkACLProvider` 作为上面的属性

变更 zk ACLs后，确保 `solr.xml` 内容匹配。

例子

你想要保护 zk 里所有 Solr 相关的内容。你想要一个 'admin' 用户可以对 zk 里的内容做任何事，该用户被用于在 zk 初始化 Solr 内容，及服务端的 Solr 节点。你还想要一个 'readonly' 用户，只能读取 zk 里的内容，这个用户由客户端使用

下面的例子

- 'admin' 用户的用户名和密码为 admin-user/admin-password
- 'readonly' 用户的用户名和密码为 readonly-user/readonly-password

提供者的类名必须第一个在 `solr.xml` 里配置

```
...
<solrcloud>
...
  <str name="zkCredientialsProvider">org.apache.solr.common.cloud.VMParamsSingleSetCre
dentialsDigestZkCredentialsProvider</str>
  <str name="zkACLProvider">org.apache.solr.common.cloud.VMParamsAllAndReadonlyDigestZ
kACLProvider</str>
```

使用 ZkCli

```
SOLR_ZK_CREDS_AND_ACLS="-DzkDigestUsername=admin-user
-DzkDigestPassword=admin-password \
-DzkDigestReadonlyUsername=readonly-user
-DzkDigestReadonlyPassword=readonly-password"

java ... $SOLR_ZK_CREDS_AND_ACLS ... org.apache.solr.cloud.ZkCLI -cmd ...
```

要使用 **bin/solr**，添加下面内容到 **bin/solr.in.sh** 末尾

```
SOLR_ZK_CREDS_AND_ACLS="-DzkDigestUsername=admin-user
-DzkDigestPassword=admin-password \
-DzkDigestReadonlyUsername=readonly-user
-DzkDigestReadonlyPassword=readonly-password"

SOLR_OPTS="$SOLR_OPTS $SOLR_ZK_CREDS_AND_ACLS"
```

要使用客户端**(SolrJ)**

```
SOLR_ZK_CREDS_AND_ACLS="-DzkDigestUsername=readonly-user
-DzkDigestPassword=readonly-password"

java ... $SOLR_ZK_CREDS_AND_ACLS ...
```

或许你使用自己开发的 SolrZkClient 客户端，你可以覆盖提供者的代码

集合(collection) API

集合 api 让你能够创建，移除，重载集合，但是在 SolrCloud 环境，你也能用它创建指定数目的分片和副本的集合

api 入口

下面的 api 都基于 url `http://<hostname>:<port>/solr`

`/admin/collections?action=CREATE` ：创建集合

`/admin/collections?action=MODIFYCOLLECTION` ：修改集合某些属性

`/admin/collections?action=RELOAD` ：重载集合

`/admin/collections?action=SPLITSHARD` ：将一个分片切分为 2 个新的分片

`/admin/collections?action=CREATESHARD` ：创建一个新的分片

`/admin/collections?action=DELETESHARD` ：删除一个不活动的分片

`/admin/collections?action=CREATEALIAS` ：创建或修改集合的别名

`/admin/collections?action=DELETEALIAS` ：删除集合别名

`/admin/collections?action=DELETE` ：删除集合

`/admin/collections?action=DELETEREPLICA` ：删除分片的一个副本

`/admin/collections?action=ADDREPLICA` ：给分片添加一个副本

`/admin/collections?action=CLUSTERPROP` ：添加/修改/删除一个集群级别的属性

`/admin/collections?action=MIGRATE` ：迁移文档到另一个集合

`/admin/collections?action=ADDROLE` ：添加一个特定的角色到集群的一个节点

`/admin/collections?action=REMOVEROLE` ：移除一个已分配的角色

`/admin/collections?action=OVERSEERSTATUS` ：获得观察者的状态和统计信息

`/admin/collections?action=CLUSTERSTATUS` ：获得集群的状态

`/admin/collections?action=REQUESTSTATUS` ：获取一个之前的异步请求的状态

`/admin/collections?action=DELETESTATUS` ：删除之前的异步请求的已存储的状态

`/admin/collections?action=LIST`：列出所有集合

`/admin/collections?action=ADDREPLICAPROP`：添加一个属性到集合/分片/副本指定的副本

`/admin/collections?action=DELETEREPLICAPROP`：从集合/分片/副本指定的副本上删除一个属性

`/admin/collections?action=BALANCESHARDUNIQUE`：发布一个属性到集合的节点上的每一个分片

`/admin/collections?action=REBALANCELEADERS`：基于已分配的 'preferredLeader' 发布一个领导角色

`/admin/collections?action=FORCELEADER`：如果领导已遗失，强制发起一个领导选举

`/admin/collections?action=MIGRATESTATEFORMAT`：从已分片的 clusterstate.json 迁移集合到每个集合 state.json(?)

参数指南

命令行工具

Solr 管理页面(默认在 `http://hostname:8983/solr/`)，提供了索引监控，执行统计，索引分布信息，副本信息，及 jvm 的线程信息。

此外，SolrCloud 也提供了管理页面(`http://localhost:8983/solr/#/~cloud`)，及 zk 命令行工具(CLI)。CLI 脚本在 `server/scripts/cloud-scripts` ，让你上传配置信息到 zk。还有一些其他命令来集合集到集合，创建或清除 zk 路径，从 zk 下载配置到本地。

> Solr 的zkcli.sh 对比 Zookeeper 的 zkCli.sh
>
> `zkcli.sh` 由 solr 提供，和 ZooKeeper 发行版所含的 `zkCli.sh` 不一样。
>
> zk 的 `zkCli.sh` 提供了完整和通用的 zk 数据操作。 solr 的 `zkcli.sh` 为 solr 特有，用来处理 zk 里的 solr 数据

使用 Solr 的 zk CLI

支持的命令行选项

简写	参数用法	含义
	`-cmd <arg>`	要执行的指令： `bootstrap` , `upconfig` , `downconfig` , `linkconfig` , `makepath` , `get` , `getfile` , `put` , `putfile` , `list` , `clear` 或 `clusterprop` 。该参数是必须滴
`-z`	`-zhhost <locations>`	zk 地址，该参数对所有指令都是必须滴
`-c`	`-collection <name>`	用于 `linkconfig` ：collection 的名称
`-d`	`-confdir <path>`	用于 `upconfig` ：配置文件的目录。 用于 `downconfig` ：从 zk 拉取的文件保存的目标目录
`-h`	`help`	显示帮助信息
`-n`	`-confname <arg>`	用于 `upconfig` , `linkconfig` , `downconfig` ：配置集的名称
`-r`	`-runzk <port>`	仅用于集群运行在单个机器上，传递 solr 运行的端口来运行内置的 zk
`-s`	`-solrhome <path>`	当运行 `-runzk` 时，或用于 `bootstrap` ：solrhome 的位置，必须滴
	-name	用于 `clusterprop` ：集群的属性名，必须滴
	-val	用于 `clusterprop` ：集群的属性值，如果未指定则为 null

简写时用一个破折号(例如，`-c mycollection`)，完整格式时可以用 1 个破折号或 2 个破折号(例如，`-collection mycollection` 或 `--collection mycollection`)

zk CLI 示例

下面的例子，均假设你已启动了 SolrCloud(`bin/solr -e cloud -noprompt`)

上传一个配置目录

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
-cmd upconfig -confname my_new_config -confdir
server/solr/configsets/basic_configs/conf
```

引导 zk 从已存在的 SOLR_HOME

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:2181 \
-cmd bootstrap -solrhome /var/solr/data
```

> **Bootstrap with chroot**
>
> 使用 `bootstrap` 指令，及一个zk 的 chroot，例如 `-zkhost 127.0.0.1:2181/solr`，将在上传配置前自动的创建 chroot 路径

将任意数据写入新的 zk 文件

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
-cmd put /my_zk_file.txt 'some data'
```

将本地文件写入 zk 文件

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
-cmd putfile /my_zk_file.txt /tmp/my_local_file.txt
```

将一个集合连接到一个配置集

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
-cmd linkconfig -collection gettingstarted -confname my_new_config
```

创建一个新的 zk 路径

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:2181 \
-cmd makepath /solr
```

设置一个集群属性

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:2181 \
-cmd clusterprop -name urlScheme -val https
```

示例

```
zkcli.sh -zkhost localhost:9983 -cmd bootstrap -solrhome /opt/solr
zkcli.sh -zkhost localhost:9983 -cmd upconfig -confdir /opt/solr/collection1/conf -con
fname myconf
zkcli.sh -zkhost localhost:9983 -cmd downconfig -confdir /opt/solr/collection1/conf -c
onfname myconf
zkcli.sh -zkhost localhost:9983 -cmd linkconfig -collection collection1 -confname myco
nf
zkcli.sh -zkhost localhost:9983 -cmd makepath /apache/solr
zkcli.sh -zkhost localhost:9983 -cmd put /solr.conf 'conf data'
zkcli.sh -zkhost localhost:9983 -cmd putfile /solr.xml /User/myuser/solr/solr.xml
zkcli.sh -zkhost localhost:9983 -cmd get /solr.xml
zkcli.sh -zkhost localhost:9983 -cmd getfile /solr.xml solr.xml.file
zkcli.sh -zkhost localhost:9983 -cmd clear /solr
zkcli.sh -zkhost localhost:9983 -cmd list
zkcli.sh -zkhost localhost:9983 -cmd clusterprop -name urlScheme -val https
zkcli.sh -zkhost localhost:9983 -cmd updateacls /solr
```

遗留的配置文件

客户端 **api**

使用 SolrJ

SolrJ 使 Java 程序更容易地和 Solr 通信。SolrJ 隐藏了大量连接到 Solr 的细节，允许你的程序和 Solr 之间用简单的高级方法交互。

SolrJ 的中心是 `org.apache.solr.client.solrj` 包，仅包含 5 个主要的类。创建一个 `SolrClient`，代表你想要使用的 Solr 实例，然后发送 `SolrRequests` 或 `SolrQuerys` 并获取 `SolrResponses`。

`SolrClient` 是个抽象类，所以要连接到远程的 Solr 实例，你实际创建的是 `HttpSolrClient` 或 `CloudSolrClient` 之一。都是通过 HTTP 与 Solr 通信，不同的是 `HttpSolrClient` 配置为使用一个确定的 Solr URL，而 `CloudSolrClient` 配置为使用 SolrCloud 集群的 zkHost 串。

```
// 单节点 Solr 客户端

String urlString = "http://localhost:8983/solr/techproducts";
SolrClient solr = new HttpSolrClient(urlString);
```

```
// SolrCloud 客户端

String zkHostString = "zkServerA:2181,zkServerB:2181,zkServerC:2181/solr";
SolrClient solr = new CloudSolrClient(zkHostString);
```

有了 `SolrClient` 后，就可以调用 `query()`，`add()`，和 `commint()` 方法。

创建和运行 SolrJ 应用程序

SolrJ api 包含在 Solr 里，所以你无需再下载或安装。但是，为了创建和运行 SolrJ 应用程序，需要添加一些库到类路径。

在创建阶段，本小节展示的例子需要 `solr-solrj-x.y.z.jar` 位于类路径。

在运行阶段，本小节展示的例子需要这些库在 'dist/solrj-lib' 目录。

使用 maven，将下面代码放入项目的 `pom.xml`

```
<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-solrj</artifactId>
  <version>x.y.z</version>
</dependency>
```

如果你担心 SolrJ 库会增加你的应用的大小，你可用代码混淆器(例如 ProGuard)移除你没有用到的 api。

设置 XMLResponseParser

SolrJ 使用二进制格式而不是 XML 作为默认的响应格式。如果你使用 Solr 1.x 版本和 SolrJ 3.x+ 版本，那么你必须使用 XML 响应。SolrJ 响应二进制格式在 3.x 版本变更过，导致了与 Solr 1.x 完全不兼容。

下面代码将返回格式设为 XML

```
server.setParser(new XMLResponseParser());
```

执行查询

使用 `query()` 让 Solr 搜索结果。你必须传递一个描述查询的 `SolrQuery` 对象，并获得一个 QueryResponse (来自于 `org.apache.solr.client.solrj.response` 包)。

`SolrQuery` 有一些方法，可以让选择请求处理器和发送参数更简单。这里有一个很简单的例子，使用默认的请求处理器，设置查询串

```
SolrQuery query = new SolrQuery();
query.setQuery(mQueryString);
```

要选择一个不同的请求处理器，在 SolrJ 4.0 和以后版本有个特定的方法

```
query.setRequestHandler("/spellCheckCompRH");
```

你也可以在查询里设置任意参数。下面代码的前 2 行是互相等价的，第 3 行演示了如果在查询串里用一个 `q` 参数

```
query.set("fl", "category,title,price");
query.setFields("category", "title", "price");
query.set("q", "category:books");
```

准备好你的 `SolrQuery` 后，用 `query()` 提交它

```
QueryResponse response = solr.query(query);
```

客户端创建一个网络连接并发送该查询。Solr 处理查询，发送响应，由一个 `QueryResponse` 解析。

`QueryResponse` 是满足查询的文档的集合。你可以用 `getResults()` 来获取文档，还可以用其他方法来获取高亮和 facet 信息

```
SolrDocumentList list = response.getResults();
```

索引文档

其他操作也是一样滴简单。要索引(添加)文档，你要做的是创建一个 `SolrInputDocument` 传递给 `SolrClient` 的 `add()` 方法。下面例子假设 SolrClient 对象 'solr' 已经创建

```
SolrInputDocument document = new SolrInputDocument();
document.addField("id", "552199");
document.addField("name", "Gouda cheese wheel");
document.addField("price", "49.99");
UpdateResponse response = solr.add(document);

// 记得提交你的改变
solr.commit();
```

以 XML 或二进制格式更新内容

SolrJ 让你以二进制格式上传内容而不是默认的 XML 格式。使用下面的代码以二进制上传，和 SolrJ 获取结果的格式一样。如果你同时使用 1.x 的 Solr 和 3.x+ 的 SolrJ，那么你必须使用 XML 格式提交请求。

```
server.setRequestWriter(new BinaryRequestWriter());
```

使用 ConcurrentUpdateSolrClient

当用 java 应用程序加载大量文档，`ConcurrentUpdateSolrClient` 代替 `HttpSolrClient` 是另一个选择。`ConcurrentUpdateSolrClient` 缓冲了所有新的文档并写入到一个打开的 HTTP 连接。这个类是线程安全的。

对于 `/update` 请求，虽然任何 SolrClient 请求都可以实现，但仅推荐使用 `ConcurrentUpdateSolrClient`

内嵌的 Solr 服务器

`EmbeddedSolrServer` 类可在你的应用内运行 Solr。这个内嵌的做法在大多场景下都不推荐，而且支持的特性有限：特别是不支持 SolrCloud 和索引复制。`EmbeddedSolrServer` 的存在主要是帮助测试。

了解更多 `EmbeddedSolrServer` 信息，请参考 Solr 源码 `org.apache.solr.client.solrj.embedded` 包里的 SolrJ 单元测试