Solr

(讲师：程道)

主要课程内容

Solr是一个建立在Lucene基础上的 企业级的 快速的 可扩展的 可部署的 搜索和存储引擎。

- **第一部分：Solr基础回顾**

  什么是solr　solr和Lucene的区别　核心特性　安装方式　管理界面

- **第二部分：Solr配置详解**

  solr目录结构　solrconfig.xml(lib dataDir indexConfig directoryFactory requestHandler)

  schema.xml （ Field Field类型 分词器 ）

- **第三部分：Solr数据操作**

  dataimport(DIH) solrJ 对文档的增删改查 SpringBoot访问Solr

- **第四部分：Solr性能优化**

  配置文件　索引构建　搜索　系统

- **第五部分：SolrCloud**

  SolrCloud简介　结构(物理 逻辑) 搭建(ZK集群 solr集群) SpringBoot访问SolrCloud

一.Solr基础回顾

1.什么是solr

为了在CNET 公司网站上添加搜索功能，Yonik Seely于2004年创建了Solr。并在2006年1月，它成为Apache软件基金会下的一个开源项目。并于2016年发布新版本Solr 6.0，支持并行SQL查询的执行,目前最新的版本是8.6.0。

Solr是一个开源搜索平台，用于构建搜索应用程序。 它建立在Lucene(全文搜索库)之上。Solr不仅限于搜索，也可以用于存储目的。像其他NoSQL数据库一样，它是一种非关系数据存储和处理技术。

Solr是一个企业级的 ，快速的 ，可扩展的，可部署的，搜索和存储引擎，用于优化搜索大量以文本为中心的数据。

2.Solr和Lucene的区别

Lucene是一个开放源代码的全文检索工具包，它不是一个完整的全文检索应用。Lucene仅提供了完整的查询引擎和索引引擎 仍然需要关注数据获取、解析、索引查看等方面的东西。Lucene的目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能，或者以Lucene为基础构建全文检索应用。

Solr的目标是打造一款企业级的搜索引擎系统，它是基于Lucene一个搜索引擎服务，可以独立运行，通过Solr可以非常快速的构建企业的搜索引擎，通过Solr也可以高效的完成站内搜索功能。

优势:

1、solr是已经将整个索引操作功能封装好了的搜索引擎系统(企业级搜索引擎产品)
2、solr可以部署到单独的服务器上(WEB服务)提供服务，我们的业务系统就只要发送请求，接收响应即可，降低了业务系统的负载并且solr的索引库就不会受业务系统服务器存储空间的限制
3、solr支持分布式集群，索引服务的容量和能力可以线性扩展

3.Solr 的核心特性

1.先进的全文搜索功能
2.基于标准的开放接口,Solr搜索服务器支持通过XML、JSON和HTTP查询和获取结果。
3.高度可扩展和容错,能够有效地复制到另外一个Solr搜索服务器
4.Solr可以通过HTML页面管理,使用XML配置达到灵活性和适配性
5.灵活的插件体系 新功能能够以插件的形式方便的添加到Solr服务器上。
6.强大的数据导入功能 数据库和其他结构化数据源现在都可以导入、映射和转化。

4.Solr 的安装方式

方式一

1.下载solr项目包 或者上传压缩包到服务器

直接去官网下载 https://lucene.apache.org/solr/downloads.html 或者

wget https://mirror.bit.edu.cn/apache/lucene/solr/7.7.3/solr-7.7.3.tgz

2.解压

tar -xvf solr-7.7.3.tgz

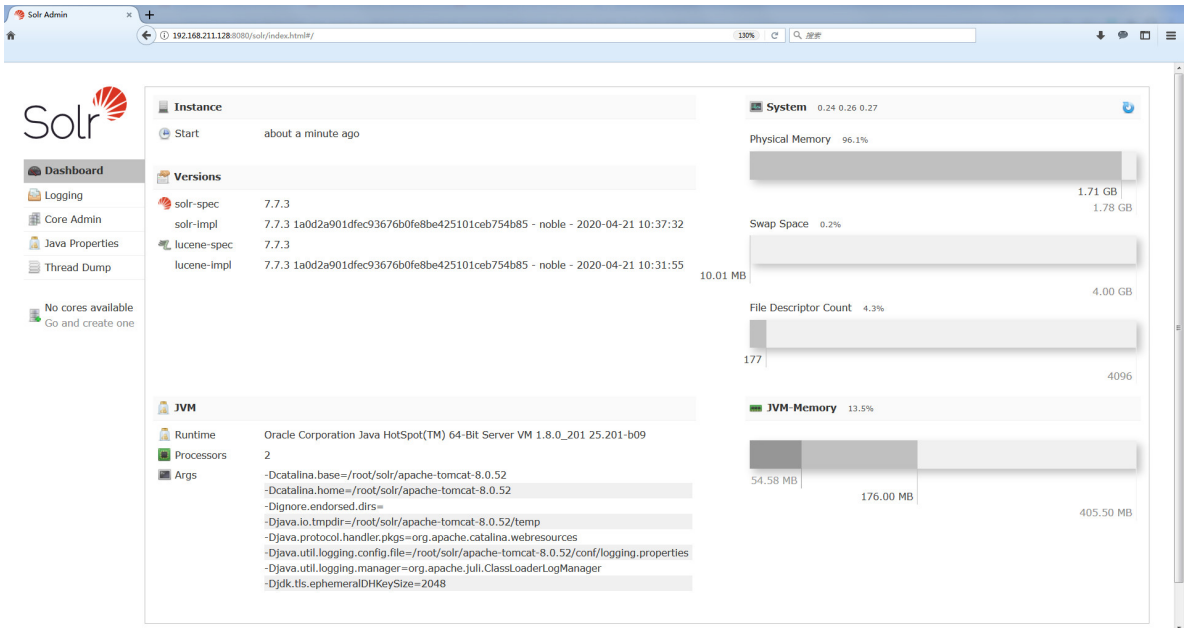
3.启动solr，到solr7.7.3/bin下

./solr start -force -port 端口

默认端口 8983

4.使用浏览器访问

http://主机:端口



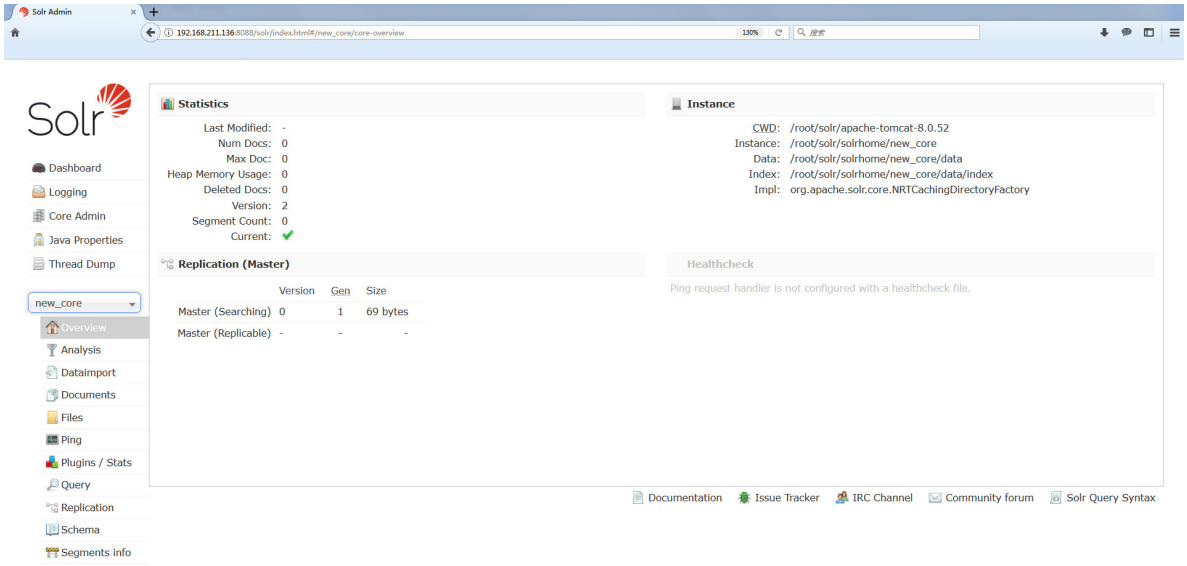
5.配置solr_core　在server/solr 下建立一个文件夹名字任意这里叫 new_core

/root/solr/solr-7.7.3/server/solr/configsets/_default/conf　到 server/solr/new_core

cp /root/solr/solr-7.7.3/server/solr/configsets/_default/conf -rf new_core

修改 managed-schema 为 schema.xml

mv managed-schema schema.xml



方式二

1. Solr 解压后server/solr-webapp下一个webapp目录，它就是Solr的Web项目，把它复制到tomcat的webapps目录下并改名为solr

```
# 进入Solr的server目录下
cd /root/solr/solr-7.7.3/server/solr-webapp/
# 复制webapp目录到tomcat-solr的webapps目录下
cp -r webapp/  /root/solr/apache-tomcat-8.0.52/webapps

# 将Solr的web应用改名
cd /root/solr/apache-tomcat-8.0.52/webapps
# 将webapp重命名为solr
mv webapp  solr
```

2. 复制所需依赖jar包

```
复制solr-7.7.3\server\lib\ext 下的jar包到apache-tomcat-8.0.52\webapps\solr\WEB-INF\lib下（即刚刚复制并重命名为solr的文件夹下）
      cp ext/* /root/solr/apache-tomcat-8.0.52/webapps/solr/WEB-INF/lib/
复制solr-7.7.3\server\lib下所有metrics-开头的jar包（一共有5个）到apache-tomcat-8.0.5/webapps/solr/WEB-INF/lib/
      cp metrics-*    /root/solr/apache-tomcat-8.0.52/webapps/solr/WEB-INF/lib/
```

3. 配置solrhome

这个目录用于存储Solr Core的数据及配置文件

首先创建solrhome存储Solr索引文件

```
mkdir -p  /usr/local/solr/solrhome
```

复制server/solr目录下所有内容到solrhome

```
cp -r * /usr/local/solr/solrhome/
```

4. 配置Tomcat

修改web.xml
修改复制到tomcat中的Solr项目中的`WEB-INF`目录下的`web.xml`配置文件
修改`solr_home`路径指向我们刚刚创建的`solrhome`

```
  <!-- 修改solrhome路径 -->
  <env-entry>
    <env-entry-name>solr/home</env-entry-name>
    <env-entry-value>/usr/local/solr/solrhome</env-entry-value>
    <env-entry-type>java.lang.String</env-entry-type>
  </env-entry>
```

去掉安全认证

```
<!-- Get rid of error message -->
<!--
  <security-constraint>
```

```
    <web-resource-collection>
      <web-resource-name>Disable TRACE</web-resource-name>
      <url-pattern>/</url-pattern>
      <http-method>TRACE</http-method>
    </web-resource-collection>
    <auth-constraint/>
  </security-constraint>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Enable everything but TRACE</web-resource-name>
      <url-pattern>/</url-pattern>
      <http-method-omission>TRACE</http-method-omission>
    </web-resource-collection>
  </security-constraint>
-->
```

5.启动tomcat 访问测试

修改tomcat的执行权限 然后启动

chmod 777 -R bin

./bin/startup.sh

在浏览器上访问solr

http://IP:端口/solr/index.html

6.配置solr_core

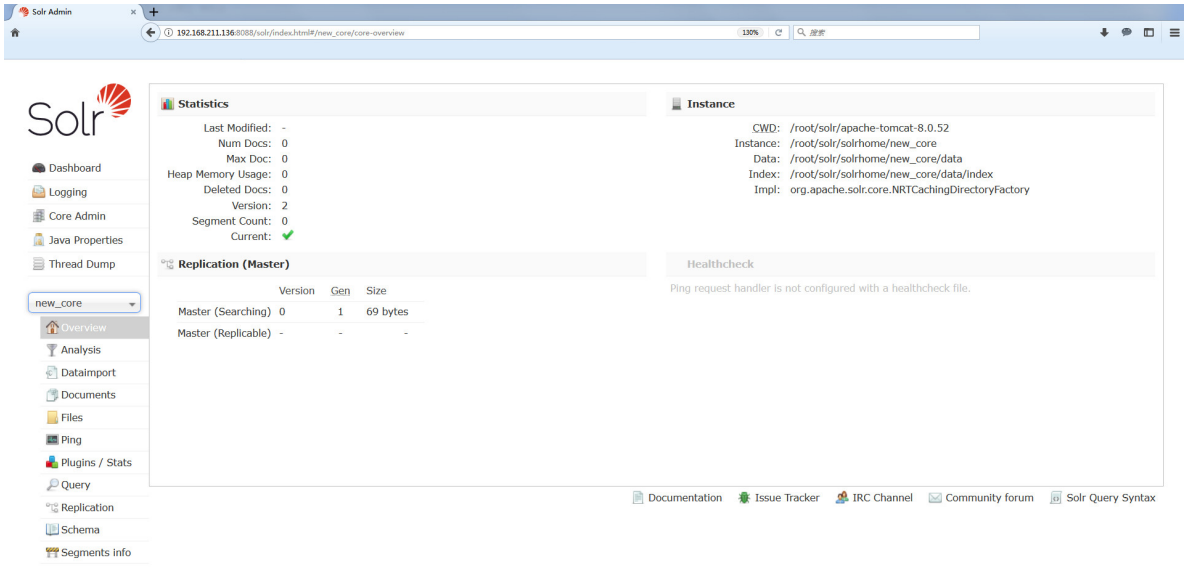
/root/solr/solr-7.7.3/server/solr/configsets/_default/conf 到 solrhome/new_core

cp /root/solr/solr-7.7.3/server/solr/configsets/_default/conf -rf new_core

修改 managed-schema 为 schema.xml

mv managed-schema schema.xml

5.管理界面功能介绍



Dashboard

仪表盘，显示了该Solr实例开始启动运行的时间、版本、系统资源、jvm等信息。

Logging

日志:显示 solr 运行出现的异常或错误

Cloud

Cloud即SolrCloud，即Solr云（集群），当使用Solr Cloud模式运行时会显示此菜单。

Core Admin

Solr Core的管理界面。在这里可以添加SolrCore实例。

```
Core管理: 主要有 Add Core(添加核心)， Unload（卸载核心），Rename（重命名核心），
Reload(重 新加载核心)
Add Core 是添加 core ： 主 要 是 在 instanceDir 对 应 的 文 件 夹 里 生 成 一 个
core.properties 文件
name：给 core 起的名字；
instanceDir：与我们在配置 solr 到 tomcat 里时的 solr_home 里新建的 core 文件夹名一致；

dataDir:确认 Add Core 时，会在 new_core 目录下生成名为 data 的文件夹
config：new_core 下的 conf 下的 config 配置文件(solrconfig.xml)
schema: new_core 下的 conf 下的 schema 文件(schema.xml)
```

java properties

Solr在JVM 运行环境中的属性信息，包括类路径、文件编码、jvm内存设置、jdk等信息。

Thread Dump

显示Solr Server中当前活跃线程信息，同时也可以跟踪线程运行栈信息。

Core selector

可以选择一个SolrCore进行详细操作

Analysis

通过此界面可以测试索引分析器和搜索分析器的执行情况

Dataimport

可以定义数据导入处理器，从关系数据库将数据导入到Solr索引库中。

默认没有配置，需要手工配置。

Documents

通过/update表示更新索引，solr默认根据id（唯一约束）域来更新Document的内容，如果根据id值搜索不到id域则会执行添加操作，如果找到则更新。

overwrite="true" ： solr在做索引的时候，如果文档已经存在，就用xml中的文档进行替换

commitWithin="1000" ： solr 在做索引的时候，每隔1000（1秒）毫秒，做一次文档提交。为了方便测试也可以在Document中立即提交， 后添加 “”

如:name:solr

Ping

```
查看当前核心库还是否工作的以及响应时间
```

Query

通过/select执行搜索索引，必须指定“q”查询条件方可搜索。

Replication

```
显示你当前 Core 的副本，并提供 disable/enable 功能
```

Schema

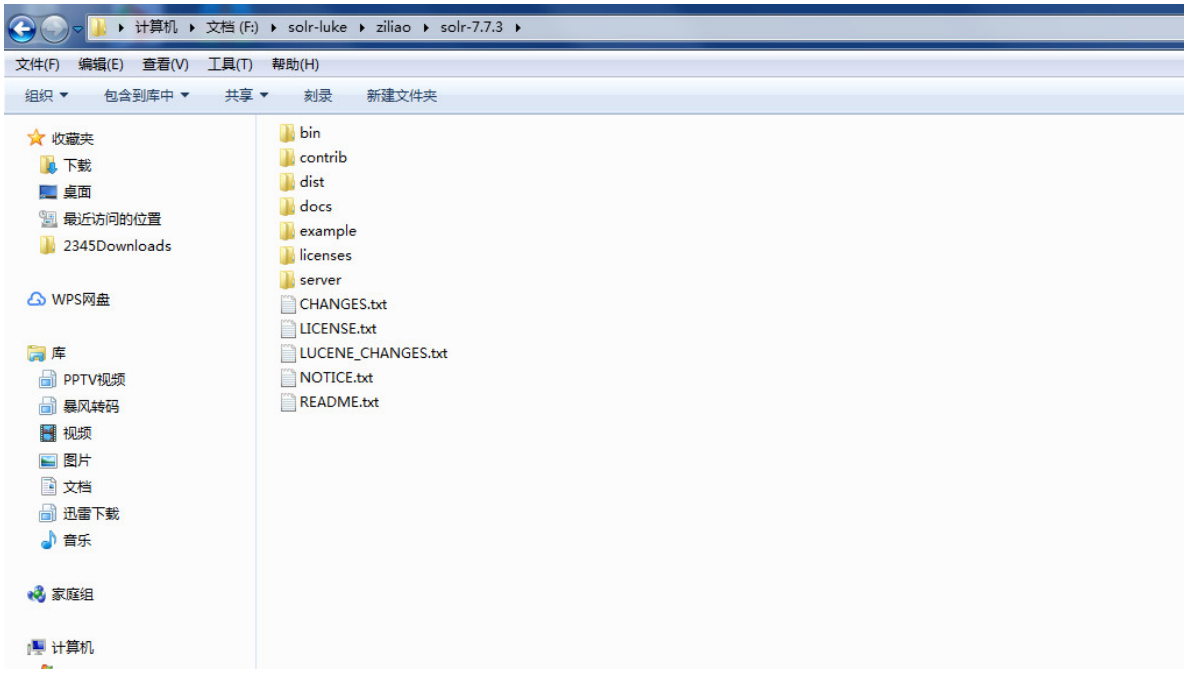
```
展示该 Core 的 shema.xml 文件中的内容
展示该Core的shema数据，如果是用ManagedSchema模式，还可以通过该页面修改，增加，删除schema
的字段
```

Segments info

展示底层Lucence的分段信息。

二. Solr配置详解

1.Solr目录结构说明



bin：Solr控制台管理工具存在该目录下。如：solr 等。
contrib：该文件包含大量关于Solr的扩展。
dist：在这里能找到Solr的核心JAR包和扩展JAR包。当我们试图把Solr嵌入到某个应用程序的时候会用到核心JAR包。
docs：该文件夹里面存放的是Solr文档，离线的静态HTML文件，还有API的描述。
example：包含Solr的简单示例。
licenses：各种许可和协议。
server：solr应用程序的核心,SolrCore核心必要文件都存放在这里。

contrib

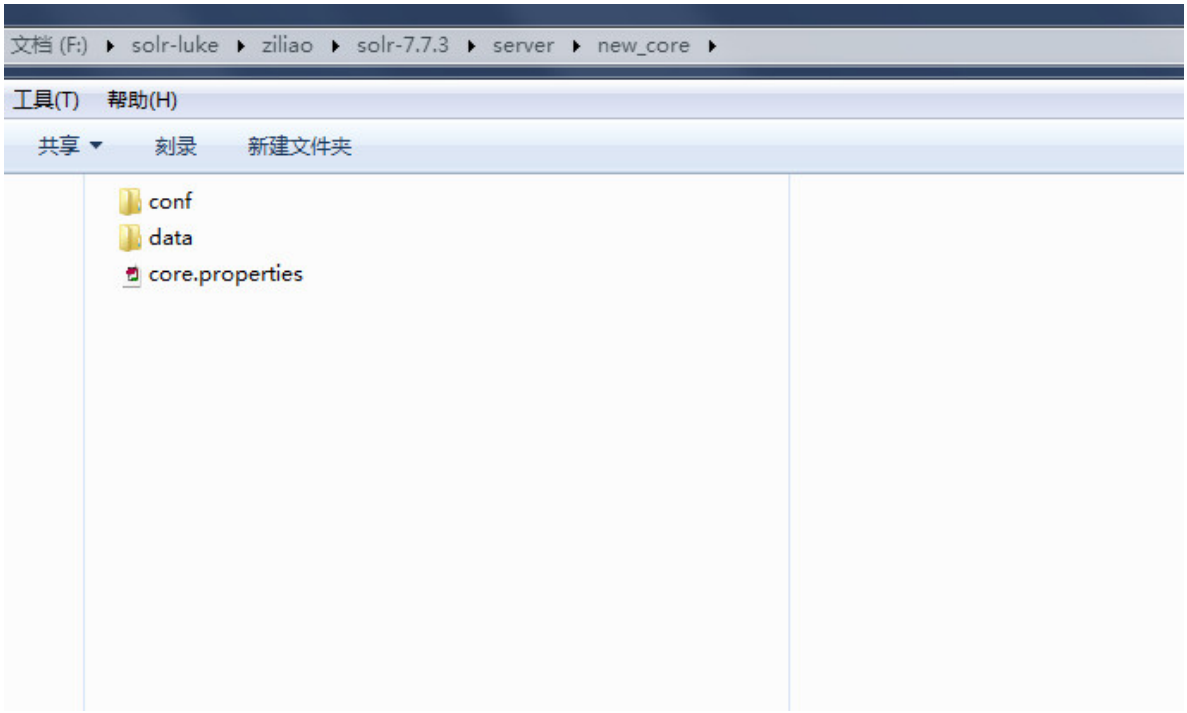
analysis-extras：该目录下面包含一些相互依赖的文本分析组件 分词器相关。
clustering：该目录下有一个用于集群检索结果的引擎。
dataimporthandler：DIH是Solr中一个重要的组件，该组件可以从数据库或者其他数据源导入数据到Solr中。
dataimporthandler-extras：这里面包含了对DIH的扩展。
extraction：集成Apache Tika，用于从普通格式文件中提取文本。
langid：该组件使得Solr拥有在建索引之前识别和检测文档语言的能力。
prometheus-exporter 采集监控数据并通过prometheus监控 solr监控相关。
velocity：包含一个基于Velocity模板语言简单检索UI框架。

server

contexts：启动Solr的Jetty的上下文配置。
etc：Jetty服务器配置文件，在这里可以把默认的8983端口改成其他的。
lib：Jetty服务器程序对应的可执行JAR包和响应的依赖包。
logs：默认情况下，日志将被输出到这个文件夹。
modules：http\https\server\ssl等配置模块。
resources：存放着Log4j的配置文件。这里可以改变输出日志的级别和位置等设置。
scripts：Solr运行的必要脚本。
solr：运行Solr的配置文件都保存在这里。solr.xml文件,zoo.cfg文件，使用SolrCloud的时候有用。子文件　夹configsets存放着Solr的示例配置文件。每创建一个核心Core都会在server目录下生成相应的core 名称
　目录。
solr-webapp：Solr的平台管理界面的站点就存放在这里。
tmp：存放临时文件。

2.SolrCore 结构

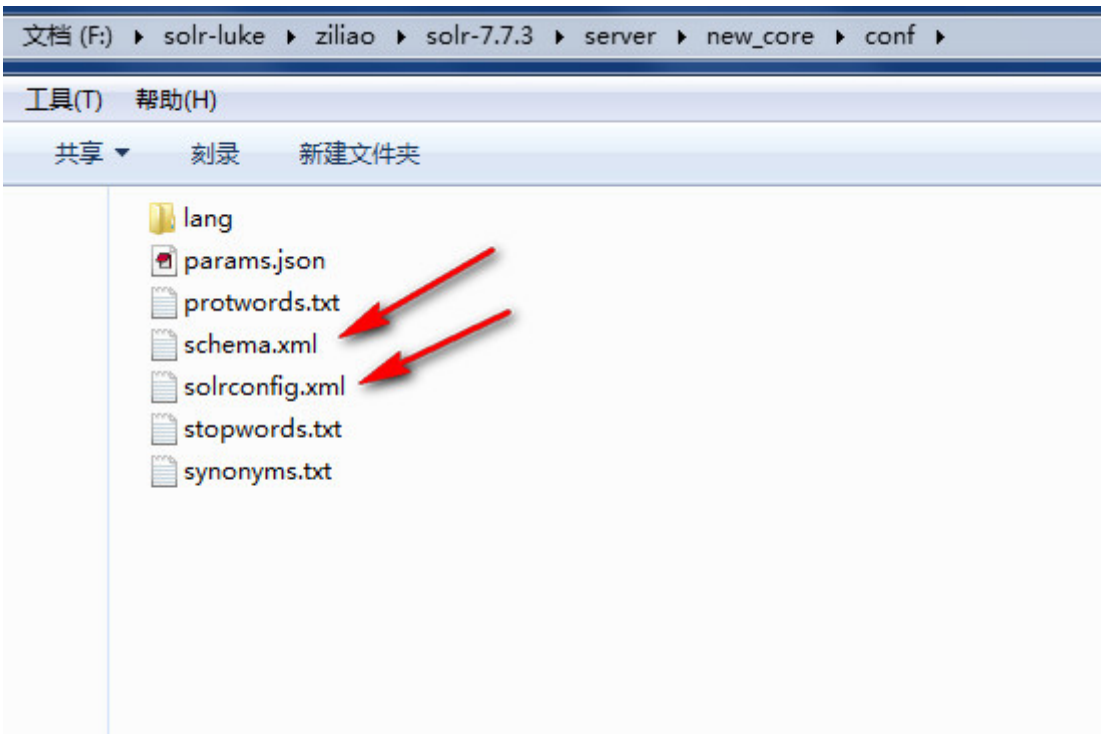
SolrCore内核：是运行在Solr服务器中的具体唯一命名的、可管理和可配置的索引，即内核就是Lucene中说到的索引。一台solr服务器可以托管一个或多个内核。



core.properties

```
#Written by CorePropertiesLocator
#Tue Jul 28 23:27:04 CST 2020
name=new_core
config=solrconfig.xml
schema=schema.xml
dataDir=data
```

conf 中的文件



3.solrconfig.xml

这个文件是来配置SolrCore实例的相关信息。如果使用默认配置可以不用做任何修改。它里面包含了很多标签，例如：lib标签、dataDir标签、requestHandler标签等。

lib 标签

在solrconfig.xml中可以加扩展载一些的jar，如果需要使用，则首先要把这些jar复制到指定的目录，比如把安装包中的contrib和dist文件夹 复制到SolrHome同级目录下 需要把 `../../../..` 修改为 `../..`。

cp contrib/ -rf /usr/local/solr/
cp dist/ -rf /usr/local/solr/

修改solrconfig.xml配置文件加载扩展的jar。

```
  <lib dir="${solr.install.dir:../../../..}/contrib/extraction/lib"
regex=".*\.jar" />
  <lib dir="${solr.install.dir:../../../..}/dist/" regex="solr-cell-\d.*\.jar"
/>
  <lib dir="${solr.install.dir:../../../..}/dist/" regex=".*\.jar" />

  <lib dir="${solr.install.dir:../../../..}/contrib/clustering/lib/"
regex=".*\.jar" />
  <lib dir="${solr.install.dir:../../../..}/dist/" regex="solr-clustering-
\d.*\.jar" />

  <lib dir="${solr.install.dir:../../../..}/contrib/langid/lib/" regex=".*\.jar"
/>
  <lib dir="${solr.install.dir:../../../..}/dist/" regex="solr-langid-\d.*\.jar"
/>
```

dataDir标签

配置SolrCore的data目录。

data目录用来存放SolrCore的索引文件和tlog日志文件

solr.data.dir表示${SolrCore}/data的目录位置

```
  <dataDir>${solr.data.dir:}</dataDir>
```

directoryFactory

```
<directoryFactory name="DirectoryFactory"
    class="${solr.directoryFactory:solr.NRTCachingDirectoryFactory}"/>
```
索引存储方案，共有以下存储方案：

1、solr.StandardDirectoryFactory,这是一个基于文件系统存储目录的工厂，它会试图选择最好的实现基于你当前的操作系统和Java虚拟机版本。

2、solr.SimpleFSDirectoryFactory,适用于小型应用程序，不支持大数据和多线程。

3、solr.NIOFSDirectoryFactory,适用于多线程环境，但是不适用在windows平台（很慢），是因为JVM还存在bug。

4、solr.MMapDirectoryFactory,这个是solr3.1到4.0版本在linux64位系统下默认的实现。它是通过使用虚拟内存和内核特性调用

mmap去访问存储在磁盘中的索引文件。它允许lucene或solr直接访问I/O缓存。如果不需要近实时搜索功能，使用此工厂是个不错的方案。

5、solr.NRTCachingDirectoryFactory,此工厂设计目的是存储部分索引在内存中，从而加快了近实时搜索的速度。

6、solr.RAMDirectoryFactory,这是一个内存存储方案，不能持久化存储，在系统重启或服务器crash时数据会丢失。且不支持索引复制

luceneMatchVersion

```
<luceneMatchVersion>7.7.3</luceneMatchVersion>
```
solr 版本

indexConfig

用于设置索引的底层的属性

```
<!--maxTokenCount即在对某个域分词的时候，最多只提取前10000个Token，后续的域值将被抛弃。-->
<filterclass="solr.LimitTokenCountFilterFactory"maxTokenCount="10000"/>
<!--writeLockTimeout表示IndexWriter实例在获取写锁的时候最大等待超时时间，超过指定的超时时间仍未获取到写锁，则IndexWriter写索引操作将会抛出异常。-->
<writeLockTimeout>1000</writeLockTimeout>
<!--表示创建索引的最大线程数，默认是开辟8个线程来创建索引。-->
<maxIndexingThreads>8</maxIndexingThreads>//
<useCompoundFile>false</useCompoundFile>//solr默认为false。如果为true,索引文件减少，检索性能降低,追求平衡。
<!--表示创建索引时内存缓存大小，单位是MB,默认最大是100M 。-->
<ramBufferSizeMB>100</ramBufferSizeMB>
<!--表示在document写入到硬盘之前，缓存的document最大个数，超过这个最大值会触发索引的flush操作。-->
<maxBufferedDocs>1000</maxBufferedDocs>
<mergePolicyclass="org.apache.lucene.index.TieredMergePolicy">
<intname="maxMergeAtOnce">10</int>
 <intname="segmentsPerTier">10</int>
</mergePolicy>
 //合并策略。
<mergeFactor>10</mergeFactor>//合并因子,每次合并多少个segments。
<mergeSchedulerclass="org.apache.lucene.index.ConcurrentMergeScheduler"/>//合并调度器。
<lockType>${solr.lock.type:native}</lockType>//锁工厂。
<unlockOnStartup>false</unlockOnStartup>//是否启动时先解锁。
<termIndexInterval>128</termIndexInterval>//Luceneloads terms into memory 间隔
<reopenReaders>true</reopenReaders>//重新打开,替代先关闭-再打开。
<deletionPolicyclass="solr.SolrDeletionPolicy">
```

```
 //提交删除策略,必须实现org.apache.lucene.index.IndexDeletionPolicy
<strname="maxCommitsToKeep">1</str>
<strname="maxOptimizedCommitsToKeep">0</str>
<strname="maxCommitAge">30MINUTES</str> OR <strname="maxCommitAge">1DAY</str>
<br>
<infoStream  file="INFOSTREAM.txt">false</infoStream>//相当于把创建索引时的日志输出。
<lockType>${solr.lock.type:native}</lockType>
```

updateHandler

```
<updateHandler>
        <!-- 启用事务日志，用于实时获取，持久化和Solr云副本恢复。日志能够随着未提交的索引增大而增大，因此推荐使用一个确切的 autoCommit
        -->
    <updateLog>
      <str name="dir">${solr.ulog.dir:}</str>
      <int name="numVersionBuckets">${solr.ulog.numVersionBuckets:65536}</int>
    </updateLog>
        <!-- AutoCommit
         满足某种条件时进行提交。
         maxDocs - 触发新的提交前的最大document数量.
         maxTime - 触发新的提交前的最长时间（ms）
         openSearcher-为false时,提交会使最近索引的变化保存下来。但是这些变化对新的searcher不可见
          如果启用 updateLog，则强烈建议使用某种autoCommit，以限制日志大小。
      -->
    <autoCommit>
        <maxTime>${solr.autoCommit.maxTime:15000}</maxTime>
        <openSearcher>false</openSearcher>
    </autoCommit>
    <!-- softAutoCommit 与 autoCommit 类似，但是只保证改变可见，不保证改变同步到硬盘。
         比hard commit 更快，更接近实时搜索 -->
    <autoSoftCommit>
        <!--5秒执行一次软提交-->
        <maxTime>5000</maxTime>
    </autoSoftCommit>

</updateHandler>

其中硬提交是提交数据持久化到磁盘里面，并且能够查询到这条数据。
软提交是提交数据到内存里面，并没有持久化到磁盘，但是他会把提交的记录写到tlog的日志文件里面
```

query

```
<query>
    <!--设置boolean 查询中，最大条件数。在范围搜索或者前缀搜索时，会产生大量的 boolean 条件，
如果条件数达到这个数值时，将抛出异常，限制这个条件数，可以防止条件过多查询等待时间过长。-->
    <maxBooleanClauses>1024</maxBooleanClauses>
        <!-- Solr内部查询缓存
         有两种缓存实现：
         LRUCache - 基于同步的 LinkedHashMap  (LRU Least Recently Used)
         FastLRUCache - 基于 ConcurrentHashMap.
         在单线程中，FastLRUCache 具有较快的 gets，较慢的 puts。因此当缓存命中率
         比较高时（>75%），会比 LRUCache 更快；并且在多CPU环境下也会更快一点
```

```
    -->

    <!-- Filter Cache
         用于缓存未排序的 SolrIndexSearcher 的查询结果集,
         当打开一个新的 searcher 时，可能会使用旧的 searcher 中缓存的值

         Parameters:
           class - SolrCache实现类 (LRUCache or FastLRUCache)
           size - 缓存中的最大实体数
           initialSize - 初始容量 (实体数)
           autowarmCount - 从旧的缓存中导入的预加载的实体数
      -->
    <filterCache class="solr.FastLRUCache"
                 size="512"
                 initialSize="512"
                 autowarmCount="0"/>

    <!-- Query Result Cache 缓存排序后的查询结果
           maxRamMB - 最大缓存容量
     -->
    <queryResultCache class="solr.LRUCache"
                      size="512"
                      initialSize="512"
                      autowarmCount="0"/>

    <!-- Document Cache 缓存文档对象
         Since Lucene internal document ids are transient,
         this cache will not be autowarmed.
      -->
    <documentCache class="solr.LRUCache"
                   size="512"
                   initialSize="512"
                   autowarmCount="0"/>

     <!-- 是否能使用到filtercache关键配置 -->
     <useFilterForSortedQuery>true</useFilterForSortedQuery>
     <!-- queryresult的结果集控制 -->
     <queryResultWindowSize>50</queryResultWindowSize>
     <!-- 是否启用懒加载field -->
     <enableLazyFieldLoading>false</enableLazyFieldLoading>
  </query>
```

requestHandler标签

requestHandler请求处理器，定义了索引和搜索的访问方式。

通过/update维护索引，可以完成索引的添加、修改、删除操作。

```
 <requestHandler name="/update/extract"
                 startup="lazy"
                 class="solr.extraction.ExtractingRequestHandler" >
     <lst name="defaults">
         <str name="lowernames">true</str>
         <str name="fmap.meta">ignored_</str>
         <str name="fmap.content">_text_</str>
     </lst>
  </requestHandler>
```

设置搜索参数完成搜索，搜索参数也可以设置一些默认值，如下：

```
   <requestHandler name="/select" class="solr.SearchHandler">
    <!-- 默认的查询参数，可被请求中的参数覆盖  -->
     <lst name="defaults">
       <str name="echoParams">explicit</str>
       <int name="rows">10</int>
       <!--以指示分布式查询在可用时应首选分片的本地副本。如果查询包含
preferLocalShards=true，
        那么查询控制器将查找本地副本来为查询服务，而不是从整个集群中随机选择副本。-->
       <bool name="preferLocalShards">false</bool>
     </lst>
    </requestHandler>
   <!-- 返回格式化过的（字符串有-->
  <requestHandler name="/query" class="solr.SearchHandler">
     <lst name="defaults">
       <str name="echoParams">explicit</str>
       <str name="wt">json</str>
       <str name="indent">true</str>
       <str name="df">text</str>
     </lst>
  </requestHandler>
```

4. schema.xml 文件

Schema：模式，是内核中字段的定义，让solr知道内核包含哪些字段、字段的数据类型、字段对应的索引存储。

Solr中提供了两种方式来配置schema，两者只能选其一

> - 默认方式，通过Schema API 来实时配置，模式信息存储在内核目录的conf/managed-schema 文件中，一般使用这种方式，特别是在SolrCloud模式下。
> - 传统的手工编辑内核 schema.xml的方式，编辑完后需重载集合/内核才会生效。

两种方式之间是可以切换的，比如用于升级操作，从旧版本到新版本的升级，切换方式如下：

手动编辑切换到API方式： 只需将solrconfig.xml中的<schemaFactory class="ClassicIndexSchemaFactory"/>去掉，或改为ManagedIndexSchemaFactory。Solr重启时，它发现之前存储在schema.xml中但没有存储在 managed-schema中，则它会备份schema.xml，然后改写schema.xml为managed-schema。此后就可以通过Schema API 管理schema了

API切换到手动编辑方式： 首先将managed-schema重命名为schema.xml；然后将solrconfig.xml中schemaFactory 的ManagedIndexSchemaFactory去掉（如果存在）或者改为ClassicIndexSchemaFactory。

managed-schema 文件详解

Solr启动一个服务器实例后，会内置了很多field字段、唯一ID、fieldType字段类型等，这些信息都是在managed-schema文件中定义的。我们先了解下managed-schema文件的大致结构：

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema version="1.6">
    <field .../>
    <dynamicField .../>
    <uniqueKey>id</uniqueKey>
    <copyField .../>
    <fieldType ...>
        <analyzer type="index">
            <tokenizer .../>
            <filter ... />
        </analyzer>
        <analyzer type="query">
            <tokenizer.../>
            <filter ... />
        </analyzer>
    </fieldType>
</schema>
```

field

```
<field name="id" type="string" indexed="true" stored="true" required="true" multiValued="false" />
```

name：属性的名称。
type：字段的数据结构类型，所用到的类型需要在fieldType中设置。
default：默认值。
indexed：是否创建索引只有index=true 的字段才能做facet.field的字段，同时只有index=true该字段才能当做搜索的内容,当然store=true或者false没关系,将不需要被用于搜索的，而只是作为结果返回的field的indexed设置为false
stored：是否存储原始数据（如果不需要存储相应字段值，尽量设为false）,表示是否需要把域值存储到硬盘上，方便你后续查询时能再次提取出来原样显示给用户
docValues：表示此域是否需要添加一个 docValues 域，这对 facet 查询， group 分组，排序，function 查询有好处，尽管这个属性不是必须的，但他能加快索引数据加载，对 NRT 近实时搜索比较友好，且更节省内存，但它也有一些限制，比如当前docValues 域只支持strField,UUIDField,Trie*Field 等域，且要求域的域值是单值不能是多值域

multValued：是否有多个值，比如说一个用户的所有好友id。（对可能存在多值的字段尽量设置为true，避免建索引时抛出错误）
omitNorms：此属性若设置为 true ，即表示将忽略域值的长度标准化，忽略在索引过程中对当前域的权重设置，且会节省内存。只有全文本域或者你需要在索引创建过程中设置域的权重时才需要把这个值设false，对于基本数据类型且不分词的域如intFeild,longField,Stre，否则默认就是 false.
required：添加文档时，该字段必须存在，类似MySQL的not null
termVectors：设置为 true 即表示需要为该 field 存储项向量信息，当你需要MoreLikeThis 功能时，则需要将此属性值设为 true ，这样会带来一些性能提升。
termPositions：是否存储 Term 的起始位置信息，这会增大索引的体积，但高亮功能需要依赖此项设置，否则无法高亮
termOffsets：表示是否存储索引的位置偏移量，高亮功能需要此项配置，当你使用SpanQuery 时，此项配置会影响匹配的结果集

field的定义相当重要，有几个技巧需注意一下，对可能存在多值得字段尽量设置 multiValued属性为true，避免建索引是抛出错误；如果不需要存储相应字段值，尽量将stored属性设为false。

dynamicField（动态域）

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

Name：动态域的名称，是一个表达式，*匹配任意字符，只要域的名称和表达式的规则能够匹配就可以使用。

例如：搜索时查询条件[product_i：钻石]就可以匹配这个动态域，可以直接使用，不用单独再定义一个product_i域。

uniqueKey

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

相当于主键，每个文档中必须有一个id域。

copyField（复制域）

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

可以将多个Field复制到一个Field中，以便进行统一的检索。当创建索引时，solr服务器会自动的将源域的内容复制到目标域中。

source：源域

dest：目标域，搜索时，指定目标域为默认搜索域，可以提高查询效率。

定义目标域：

```
<field name="text" type="text_general" indexed="true" stored="false"
multiValued="true"/>
```

目标域必须要使用：multiValued="true"

fieldType（域类型）

```
<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" />
    <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>
```

```
name：域类型名称，用于Field定义中的type属性引用
class：存放该类型的值来进行索引的字段类名（同lucene中Field的子类）。注意，应以 solr.为前缀，这样solr就可以很快定位到该到哪个包中去查找类，如 solr.TextField 。如果使用的是第三方包的类，则需要用全限定名。
positionIncrementGap：用于多值字段，定义多值间的间隔，来阻止假的短语匹配
autoGeneratePhraseQueries：用于文本字段，如果设为true，solr会自动对该字段的查询生成短语查询，即使搜索文本没带“”
synonymQueryStyle：同义词查询分值计算方式
enableGraphQueries：是否支持图表查询

analyzer：指定分词器。在FieldType定义的时候最重要的就是定义这个类型的数据在建立索引和进行查询的时候要使用的分析器analyzer，包括分词和过滤。
    type：index和query。Index 是创建索引，query是查询索引。
    tokenizer：指定分词器
    filter：指定过滤器
```

分词器

solr7自带分词中文分词器

1、复制jar包

```
 cp contrib/analysis-extras/lucene-libs/lucene-analyzers-smartcn-7.7.3.jar
solr/WEB-INF/lib
```

2.复制新项目的配置文件

```
创建solrcore  mkdir solrhome/new_core
cp -r server/solr/configsets/_default/conf   solrhome/new_core/
```

3.然后到solrhome/new_core/conf目录中打开managed-schema文件，增加如下代码

```
<fieldType name="text_hmm_chinese" class="solr.TextField"
positionIncrementGap="100">
        <analyzer type="index">
            <tokenizer
class="org.apache.lucene.analysis.cn.smart.HMMChineseTokenizerFactory"/>
        </analyzer>
        <analyzer type="query">
            <tokenizer
class="org.apache.lucene.analysis.cn.smart.HMMChineseTokenizerFactory"/>
        </analyzer>
 </fieldType>
```

4、重启tomcat

```
./bin/shutdown.sh
./bin/startup.sh
```

5.访问项目 http://192.168.211.128:8088

测试分词效果

IK中文分词器

1.下载分词器

一般解决分词问题会选择ikanalyzer，因为相对来说ikanalyzer更新的比较的好， solr7 本身提供中文的分词jar包，在此一并讲一下

下载solr7版本的ik分词器，下载地址：http://search.maven.org/#search%7Cga%7C1%7Ccom.github.magese
分词器GitHub源码地址：https://github.com/magese/ik-analyzer-solr7

将下载好的jar包放入 tomcat 对应的项目的 /WEB-INF/lib目录中

2、复制新项目的配置文件
创建solrcore mkdir solrhome/ik
cp -r server/solr/configsets/_default/conf solrhome/ik/

3.然后到server/solr/ik/conf目录中打开managed-schema文件，增加如下代码

```
<!-- ik分词器 -->
<fieldType name="text_ik" class="solr.TextField">
  <analyzer type="index">
      <tokenizer class="org.wltea.analyzer.lucene.IKTokenizerFactory"
useSmart="false" conf="ik.conf"/>
      <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
      <tokenizer class="org.wltea.analyzer.lucene.IKTokenizerFactory"
useSmart="true" conf="ik.conf"/>
      <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

4、重启tomcat

```
./bin/shutdown.sh
./bin/startup.sh
```

5.新增项目ik
访问项目 http://192.168.211.128:8088

停用词和同义词

```
<analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
</analyzer>
```

> - words属性指定停用词文件的绝对路径或相对solr主目录下conf/目录的相对路径
> - 停用词定义语法：一行一个

stopwords.txt

```
hello
like
```

同义词

```
<analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.SynonymGraphFilterFactory" synonyms="mysynonyms.txt"/>
    <filter class="solr.FlattenGraphFilterFactory"/>
</analyzer>
<analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.SynonymGraphFilterFactory" synonyms="mysynonyms.txt"/>
</analyzer>
```

mysynonyms.txt

synonyms属性指定同义词文件的绝对路径或相对solr主目录下conf/目录的相对路径
同义词定义语法：一类一行，如下表示，=>表示查询时标准化为后面的内容

```
couch,sofa,divan
teh => the
huge,ginormous,humungous => large
small => tiny,teeny,weeny
瓷器 => 中国
```

三.Solr数据操作

1.使用dataimport 导入数据库数据

1. 首先找到solr7.7.3/dist/solr-dataimporthandler-7.7.3.jar 和 solr-dataimporthandler-extras-7.7.3.jar，复制到tomcat/webapp/solr/WEB-INF/lib/下，并且找到相应数据库的驱动包，也同样放到该目录。我这里用的是mysql的驱动包。
2. 将mysql的数据库脚本 执行 向mysql中插入数据。

   上传脚本到服务器 然后执行脚本

   mysql> source book.sql
3. 找到solr7.7.3/example/example-DIH/solr/db/conf/db-data-config.xml，把其复制到solrhome/new_core/conf/下，并改名为data-config.xml。
4. 打开并编辑data-config.xml，完整的配置文件如下

```
<dataConfig>
    <!-- 这是mysql的配置  -->
    <dataSource driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/solr_book" user="root" password="123456"/>
    <document>
        <!-- query是一条sql，代表在数据库查找出来的数据 -->
        <entity name="book" query="select * from book">
            <!-- 每一个field映射着数据库中列与文档中的域，column是数据库列，name是
solr的域(必须是在managed-schema文件中配置过的域才行) -->
            <field column="id"  name="id"/>
            <field column="name" name="name"/>
            <field column="price" name="price"/>
            <field column="description" name="desc"/>
        </entity>
    </document>
</dataConfig>
```

5. 找到 solrconfig.xml，并打开，在里面添加一段内容，如下

```
<requestHandler name="/dataimport" class="solr.DataImportHandler">
    <lst name="defaults">
        <str name="config">data-config.xml</str>
    </lst>
</requestHandler>
```

6. 在new_core/conf/managed-schema 配置对应的域

```
<field name="name" type="text_ik" indexed="true" stored="true"/>
<field name="price"  type="pfloat" indexed="true" stored="true"/>
<field name="desc" type="text_ik" indexed="true" stored="false" />
```

7. 重启tomcat，访问solr控制台,找到new_core中的dataimport，点击蓝色的按钮，则开始导入，导入过程依据数量量的大小，需要的时间也不同，可点击右边的Refresh status来刷新状态，可以查看当前导入了多少条。

2.查询基本语法

1.q 查询字符串

```
q:    查询关键字，必须的。
请求的q是字符串，如果查询所有使用*:*
```

2.fq: (filter query)过滤查询

```
作用：在q查询符合结果中同时是fq查询符合的
请求fq是一个数组（多个值）
过滤查询价格从1到120的记录。
price:[1 TO 120]
也可以使用“*”表示无限，例如：
120以上：price:[120 TO *]
120以下：price:[* TO 120]
```

3.sort: 排序，desc代表降序，asc代表升序

如:按照价格升序排 price desc

4.start: 分页显示使用，开始记录下标，从0开始

rows: 指定返回结果最多有多少条记录，配合start来实现分页

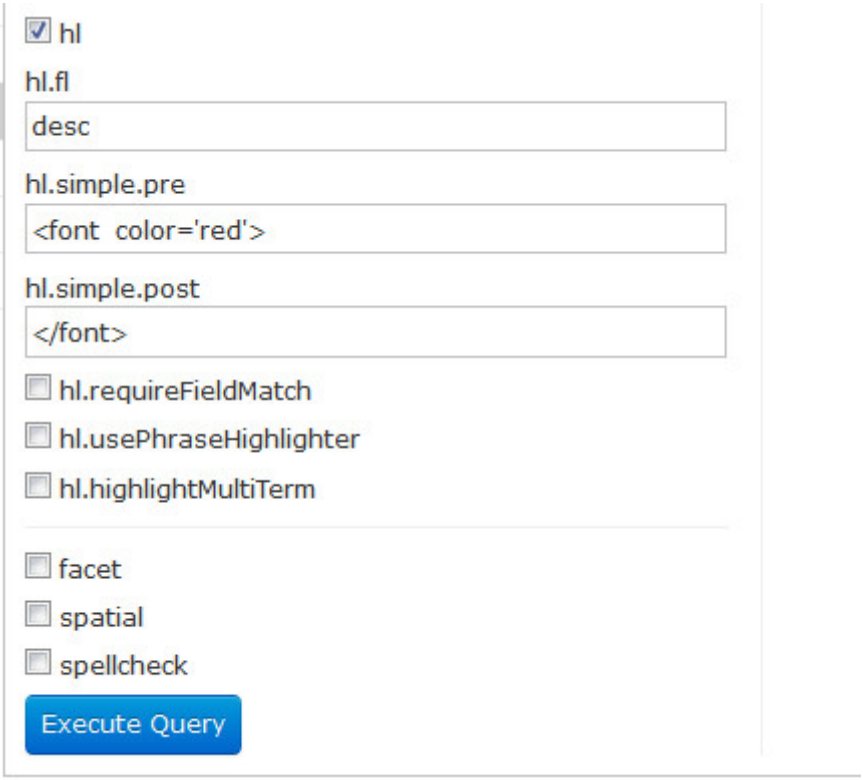
5.fl: (Field List)指定返回那些字段内容，用逗号或空格分隔多个。

name , desc

6.df: 指定默认搜索Field

7.wt: (writer type)指定输出格式，可以有 xml, json, php, phps

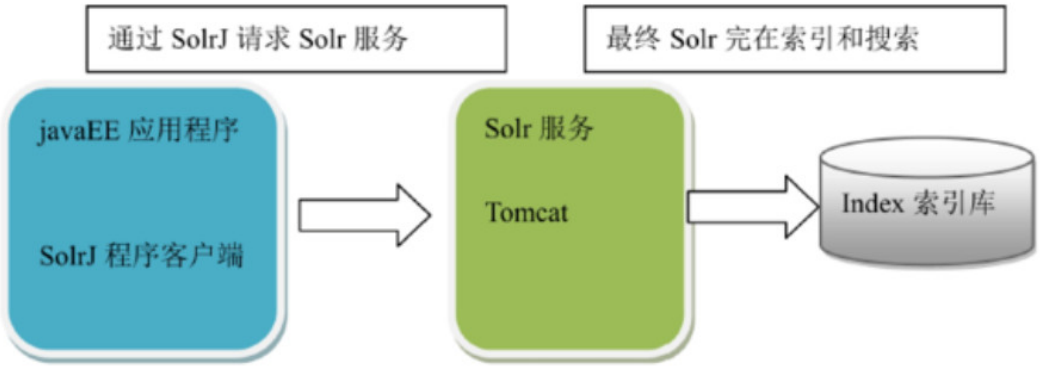
8.hl: 是否高亮 ,设置高亮Field，设置格式前缀和后缀。



3.SolrJ操作Solr

3.1 SolrJ 介绍

solrJ是一个用来访问solr的java客户端，提供了索引和搜索的方法（将一些常用的命令封装进去了），通过solrJ提供的API 接口来操作solr服务。




```
<dependency>
    <groupId>org.apache.solr</groupId>
    <artifactId>solr-solrj</artifactId>
    <version>7.7.3</version>
</dependency>
```

3.2 SolrJ的基本操作

增加和修改文档

根据id（唯一约束）域来更新Document的内容，如果根据id值搜索不到id域则会执行添加操作，如果找到则更新。

```
private String serverUrl = "http://192.168.211.128:8080/solr/new_core";
```

```
@Test
public void testSaveOrUpadteIndex() throws SolrServerException, IOException{

    //创建连接对象
    HttpSolrClient client =  new HttpSolrClient.Builder(serverUrl).build();
    SolrInputDocument doc = new SolrInputDocument();

    doc.addField("id", "41");
    doc.addField("name", "java架构师之路");
    doc.addField("desc", "Lucene Solr ES java架构师都得学 重点ES");
    doc.addField("price",12800);
    client.add(doc);
    client.commit();

    client.close();
}
```

删除文档

```
@Test
public void testDeleteById() throws Exception{
    HttpSolrClient client =  new HttpSolrClient.Builder(serverUrl).build();
    //删除文档  根据id删除文档
    client.deleteById("41");
    //solrServer.deleteByQuery("name:solr");
    //提交
    client.commit();
}
```

查询

```
 @Test
public void testSimpleQuery() throws Exception{
    //1.创建连接
    HttpSolrClient client =  new HttpSolrClient.Builder(serverUrl).build();
    //2.创建查询语句
    SolrQuery query = new SolrQuery();
    //3.设置查询条件
    query.set("q", "id:11");
    //4.执行查询
    QueryResponse queryResponse = client.query(query);
    //5.取文档列表public class SolrDocumentList extends ArrayList<SolrDocument>
    SolrDocumentList documentList = queryResponse.getResults();
    for (SolrDocument solrDocument : documentList) {
        //取各个文档信息
        System.out.println("id:"+solrDocument.get("id")+" ");
        System.out.println("name:"+solrDocument.get("name")+" ");
        System.out.println("desc:"+solrDocument.get("desc")+" ");
        System.out.println("price:"+solrDocument.get("price")+" ");
    }

    client.close();
```

```
}
```

3.3 SolrJ高级查询

查询参数

```
查询参数 与 solr管理界面的条件一致
    q - 查询字符串，如果查询所有*:* (id:1)
    fq - （filter query）过虑查询，过滤条件，基于查询出来的结果
    start - 分页开始
    rows - 分页查询数据
    sort - 排序，格式：sort=<field name>+<desc|asc>[,<field name>+<desc|asc>]… 。
示例：（score desc, price asc）表示先 “score” 降序，再 “price” 升序，默认是相关性降序。
    wt - (writer type)指定输出格式，可以有 xml, json, php, phps。
    fl表示索引显示那些field( *表示所有field,如果想查询指定字段用逗号或空格隔开
       如：name,price,desc或name price desc 注意：字段是严格区分大小写的
    q.op 表示q 中 查询语句的 各条件的逻辑操作 AND(与) OR(或)
    hl 是否高亮 ,如hl=true
    hl.fl 高亮field ,hl.fl=name,desc
    hl.simple.pre 高亮前面的格式
    hl.simple.post 高亮后面的格式
```

检索运算符

```
1. “:” 指定字段查指定值，如返回所有值*:*
2. “?” 表示单个任意字符的通配
3. “*” 表示多个任意字符的通配（不能在检索的项开始使用*或者?符号）
4. “~” 表示模糊检索  如:so~N 匹配 N 个编辑距离以内的内容。
5. 邻近检索，如检索相隔10个单词的”apache”和”jakarta”，”jakarta apache”~10
6. “^” 控制相关度检索，如检索solr apache，同时希望去让”slor”的相关度更加好，那么在其后加上”^”符号和增量值，即solr^4 apache
7. 布尔操作符AND、||
8. 布尔操作符OR、&&
9. 布尔操作符NOT、!、- （排除操作符不能单独与项使用构成查询）
10. “+” 存在操作符，要求符号”+”后的项必须在文档相应的域中存在
11. [ ] 包含范围检索，如检索某时间段记录，包含头尾，price:[120 TO 600]
```

查询和高亮信息获取

```
@Test
public void testComplexQuery() throws Exception {
    // 创建连接对象
    HttpSolrClient client =  new HttpSolrClient.Builder(serverUrl).build();
    //创建一个query对象
    SolrQuery query = new SolrQuery();
    //设置查询条件
    query.setQuery("solr");
    //过滤条件
    query.setFilterQueries("name:solr");
    //排序条件
```

```
    query.setSort("price", SolrQuery.ORDER.asc);
    //分页处理
    query.setStart(0);
    query.setRows(10);
    //结果中域的列表
    query.setFields("id","name","price","desc");
    //设置默认搜索域
    query.set("df", "desc");
    //高亮显示
    query.setHighlight(true);
    //高亮显示的域
    query.addHighlightField("name desc");

    //高亮显示的前缀
    query.setHighlightSimplePre("<font color='red'>");
    //高亮显示的后缀
    query.setHighlightSimplePost("</font>");
    //执行查询
    QueryResponse queryResponse = client.query(query);
    //取查询结果
    SolrDocumentList solrDocumentList = queryResponse.getResults();
    //共查询到书的数量
    System.out.println("共查询到书的数量:" + solrDocumentList.getNumFound());
    //获取高亮
    Map<String, Map<String, List<String>>> highlighting =
        queryResponse.getHighlighting();
    //遍历查询的结果
    for (SolrDocument solrDocument : solrDocumentList) {
        // 取具体的高亮信息
        String bookHL = "";
        List<String> list =
highlighting.get(solrDocument.get("id")).get("name");
        //判断是否有高亮内容
        if (null != list) {
            bookHL = list.get(0);
        } else {
            bookHL = (String) solrDocument.get("name");
        }
        System.out.println("bookHL="+bookHL);
        //取各个文档信息
        System.out.println("id:"+solrDocument.get("id")+" ");
        System.out.println("name:"+solrDocument.get("name")+" ");
        System.out.println("desc:"+solrDocument.get("desc")+" ");
        System.out.println("price:"+solrDocument.get("price")+" ");

    }
    client.close();
}
```

4.SpringBoot 访问Solr

1.Maven依赖

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.0.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-solr</artifactId>
    </dependency>
</dependencies>
```

2.配置Solr连接信息，在application.yml

```
spring:
  data:
    solr:
      host: http://192.168.211.128:8080/solr/new_core
```

3.Book实体类

```
public class Book implements Serializable {
    @Field
    private String id;
    @Field
    private String name;
    @Field("desc")
    private String description;
    @Field
    private double price;

    // get set 方法  构造  toString 省略
}
```

4.BookService

```
@Service("bookService")
public class BookService{
    @Autowired
    SolrClient solrClient;
    // 添加数据
    public void addBook(Book book) {
        /*
        SolrInputDocument document = new SolrInputDocument();
        document.setField("id",book.getId());
        document.setField("name",book.getName());
        document.setField("price",book.getPrice());
        document.setField("desc",book.getDescription());
        solrClient.add(document);
        solrClient.commit();
        */
        try {
```

```
            solrClient.addBean(book);
            solrClient.commit();
        } catch (SolrServerException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


    public List<Book> queryBooks(String query) {
        SolrQuery solrQuery = new SolrQuery();
        //设置默认搜索的域
        solrQuery.set("df", "desc");
        solrQuery.setQuery(query);
        //高亮显示
        solrQuery.setHighlight(true);
        //设置高亮显示的域
        solrQuery.addHighlightField("name desc");
        //高亮显示前缀
        solrQuery.setHighlightSimplePre("<font color='red'>");
        //后缀
        solrQuery.setHighlightSimplePost("</font>");
        try {
            QueryResponse queryResponse = solrClient.query(solrQuery);
            if (queryResponse == null){
                return null;
            }
            //获取高亮
            Map<String, Map<String, List<String>>> map =
queryResponse.getHighlighting();
            // 获取书的列表
            List<Book>  bookList  =  queryResponse.getBeans(Book.class);
            for (Book book : bookList){
                List<String> list = map.get(book.getId()).get("desc");
                if (!CollectionUtils.isEmpty(list)){
                    book.setDescription(list.get(0));
                }
                list = map.get(book.getId()).get("name");
                if (!CollectionUtils.isEmpty(list)){
                    book.setName(list.get(0));
                }
            }
            return bookList;

        } catch (SolrServerException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
       return  null;
    }
}
```

5.从Spring容器中获取服务对象测试

```
@SpringBootApplication
public class ApplicationTest{
    public static void main(String[] args) {
        ApplicationContext
                app =  SpringApplication.run(ApplicationTest.class,args);
        BookService bookService = app.getBean("bookService", BookService.class);
        bookService.addBook(new Book("51","ES","ES就是ElasticSearch",911.1));
        List<Book> books = bookService.queryBooks("name:ES");
        System.out.println(books);
    }
}
```

四.Solr性能优化

1.配置文件

1.schema.xml 配置

schema 配置不合理，往往会导致查询性能低，索引占用磁盘、内存空间大的问题。

1)合理设置域属性

- 域是否要检索(indexed)，是否要存储(stored)，按需配置，避免不必要的空间浪费。
- 域是否需要根据文本长度算分，是否需要在建索引时设置权重，如果不需要，设置 omitNorms=true
- omitPositions、omitTermFreqAndPositions，词频信息和打分相关，位置信息和高亮显示相关，当不需要这些功能，则可设置为 true，节省磁盘空间，提升搜索速度。
- 对于需要排序的字段，使用 docValues，构造 fieldCache 会进行压缩，节省内存使用。

2)使用正确的数据类型

- 对于数值类型，使用能正确存储的最小数值类型，更小的数值类型占用更小的磁盘、内存、CPU 缓存，并且处理时的 CPU 周期也更少。
- 数值类型不要用 string，一个整型占 4 字节，用 string，大小为 1000 以上的整型就占了 4 个字节了。当然，对于只有几个值(比如 0、1、2、3) 的可枚举的，可以用 string。
- 不需要分词的域，用 string，不要用 text，text 默认用标准分词器分词。
- 需要范围查询的数值类型，需要使用 plong、pint 等分精度索引的类型，范围查询性能是不分精度索引的数值类型的 10 倍。当然，也不能滥用，分精度索引的数值类型比较占用空间，如果没有范围查询的需求，则不需要使用。

2.solrconfig.xml 配置

1)索引目录类型

- 采用 NRTCachingDirectoryFactory，这种目录类型，小索引会缓存在内存中，减少磁盘 IO。而且这些小文件往往是频繁变化的，放在内存中，则 reopen 的时候，不需要读磁盘，性能会好很多。
- autoSoftCommit 和 autoCommit
  hardcommit 作用是使索引持久化，会 flush 当前正在索引的段到磁盘，比较重，影响查询性能，时间间隔可以设置得较长。没有 hardcommit，机器挂了 ,重启后，会从 tlog 恢复。

softcommit作用是使索引可见，可根据实时性需要设置适当的长度。需要注意，softcommit会导致searcher层cache失效。索引实时性要求不高的情况下，频率尽可能设置长一点。

2)cache 配置

缓存类型

- queryResultCache，查询结果缓存，key 由 q 参数、fq 参数、sort 参数组成，value 是 docId 和 score（score 可能没有）的有序集合 DocList。只要 q、fq、sort 参数有一个变化了，则属于不同的 key，不会命中结果。
- documentCache，document 的缓存，key 是 docId，value 是 document。
- filterCache，filterQuery 结果缓存，key 是 fq 参数的值，value 是 docId 的无序集合 DocSet。
- fieldCache，正排索引缓存，可通过 docId 获取字段值。排序、facet、group 等需要正排索引的查询需要用到 fieldCache。fieldCache 是基于段的，一个字段的 fieldCache 是在第一次使用的时候加载到内存中的。Lucene 用一个 weakHashMap 存放已加载的段的 fieldCache，key 为段的 indexReader。因此，fieldCache 是常驻内存，不会自动释放的，除非段被合并，不存在了，才会释放掉。使用时，要注意内存的消耗，避免内存不够用，发生 OOM。

缓存注意事项

- queryResultCache、documentCache、filterCache 都是 searcher 级别的缓存，searcher 重新打开(softcommit 会触发)，则缓存失效。其中 queryResultCache、filterCache 可以配置 autowarm 使 searcher 预热时重新加载。documentCache 的 key 是 docId，重新打开后，document 的 id 已经变化，因此 documentCache 不能进行 autowarm。

```
<filterCache class="solr.LRUCache"   size="16384"   initialSize="4096"
autowarmCount="4096"/>
```

- 对于实时查询（比如 softcommit 频率为 1 秒），最好不要配置 cache，因为缓存失效太快，缓存命中率可能比较低，如果配置了 autowarm，还会导致不停的 autowarm，加重服务器负担。除非查询语句都比较集中，缓存条目很少。
- 对于有翻页的查询，可以适当调整 queryResultWindowSize 参数，比如一页的大小为 10，则 queryResultWindowSize 设置为 50，则后面 4 页，都会命中缓存，当然参数设置越大占用内存越多。
- enableLazyFieldLoading 配置为 true，只读取需要的字段。这个属性要配合 documentCache 使用，即开启了 documentCache，才能发挥 lazyLoading 的作用。

> 举个例子，比如查询出来的条目可能只显示简单的信息，点击具体条目，则需要把整个条目的所有信息展示出来。则点击具体条目去查询时，命中了 `cache`，获取到的 `document` 就是有 `lazyField` 的 `document` 了，`Solr` 把它返回给调用者时，就会去加载 `lazyField` 的真实的值。

- 可配置 firstSearcher、newSearcher 来预热耗时的查询，使查询结果缓存起来，使查询性能更平滑。

```
 <listener event="newSearcher" class="solr.MyQuerySenderListener">
```

2.索引构建

建议采用离线方式构建索引 ，离线构建索引的好处：

- 避免在线构建索引对在线服务的影响。
- 不用考虑记录更新的情况，省掉查询老记录是否存在的步骤，性能有提升。
- 可以使用内存大的机器，在内存中（ RAMDirectory ）进行构建（ 内存放不下，可考虑一个段一个段的构建 ），速度往往是在线构建的好几倍。
- 离线构建可以打开多个 indexWriter 进行构建(最好多个进程，因为 IndexWriter 存在类上加锁的情况，多个线程还是存在锁等待)，然后再合并到一个 index 里，虽然在线索引也可以多个线程并发写，但会存在并发锁的问题。
- 离线索引追求的是吞吐量，对响应时间要求不高，可按照高吞吐量调配 jvm 参数，而在线索引还得考虑 jvm 对在线查询的影响。
- 离线构建可减少中间的段的生成，可以调大 ramBufferSzie 和 mergeFactor，避免生成小的段，避免在构建的过程中自动合并，在构建完成后再根据需要触发合并动作，省掉中间的合并过程，速度也有一定的提高。

3.搜索

- 如果不是所有字段都需要返回，则明确指定需要返回的字段，减少系统开销。
- 一次返回的记录数不要太多，深度翻页使用 cursorMark 参数提升性能。

```
 //游标查询
public  void cursorQuery()throws Exception{

    //solr查询封装
    SolrQuery sq =new SolrQuery();
    sq.setRows(2);//设置游标一次读的数量
    sq.set("q", "*:*");//按条件检索
    sq.setSort("id", SolrQuery.ORDER.asc);//根据主键排序
    String cursorMark = CursorMarkParams.CURSOR_MARK_START;//游标初始化
    boolean done = false;
    while (!done) {
        sq.set(CursorMarkParams.CURSOR_MARK_PARAM, cursorMark);//变化游标条件
        QueryResponse rsp =  solrClient.query(sq);//执行多次查询读取
        String nextCursorMark = rsp.getNextCursorMark();//获取下次游标
        //做一些操作数据的事
        for(SolrDocument sd:rsp.getResults()){
            System.out.println(sd.get("id"));
        }
        //如果两次游标一样，说明数据拉取完毕，可以结束循环了
        if (cursorMark.equals(nextCursorMark)) {
            done = true;
        }
        cursorMark = nextCursorMark;
    }

}
```

- 查询条件中包含路由字段的，可以先计算出分片，再从分片中获取机器，并指定 shard 来查询，只查询相应的 shard，避免服务端所有 shard 都查询。

```
String shards = "192.168.1.215:8983/solr/a,192.168.1.214:8983/solr/b";
solrParams.set("shards", shards);//设置shard
```

- 合理使用 filterquery，filterquery 结合 filterCache，能把常用的查询条件缓存起来，提升查询效率。对于不使用 filterCache 的情况下，多个 OR 条件组合起来的查询也应该用 filterquery，因为 filterquery 是不算分的，性能会好很多。主查询中的 OR 查询，对于一个 document 并不是有一个 OR 查询条件命中即返回，接着去查下一个 document，而是所有的 OR 条件都去查一遍，记录命中的 OR 条件的数量，由此来计算得分，所以会比在 filterQuery 中查询慢很多。

4.系统

- 内存分配
  合理设置 jvm 内存大小，Solr 缓存、lucene 缓存、词典文件等都需要加载到内存中，内存设置太小，会造成 gc 频繁。但也不要把内存用得太满，在 jvm 内存足够的情况下，多留点给操作系统做文件缓存，这样索引文件能更多地被操作系统缓存起来，减少磁盘 IO，提升搜索的性能。

```
方法一：直接修改配置文件参数
打开solr\bin目录下的solr.in.sh脚本文件
搜索找到 SOLR_HEAP 或者 SOLR_JAVA_MEM ,然后修改
SOLR_HEAP="1024m" 或者 SOLR_JAVA_MEM="-Xms512m -Xmx512m"

方法二：修改JVM内存存执
打开 tomcat\bin下面的catalina.sh文件，加入
set JAVA_OPTS= -Xms1024m -Xmx1024m

方法三：修改系统环境变量
设置CATALINA_OPTS 或者 JAVA_OPTS
CATALINA_OPTS="-Xms128m -Xmx1024m -XX:PermSize=64M -XX:MaxPermSize=512M "

方法四：在启动时直接设置JVM大小
./solr -m 2g
```

- swap
  系统内存不足时，操作系统会拿交换区(磁盘)来当做内存使用，交换区的访问速度和内存比非常慢，会影响搜索的性能，对 gc 也有较大的影响，jvm 收集交换区的垃圾对象时，常常速度很慢，造成应用停顿。建议内存足够的情况下，把 swap 关掉。如果内存比较紧张，则建议把 swapness 参数的值调小，让操作系统尽量少使用 swap，完全关掉可能会造成操作系统内存不足，进程而被操作系统 kill 掉。

```
如果内存够大，应当告诉 linux 不必太多的使用 SWAP 分区， 可以通过修改 swappiness 的数值。swappiness=0的时候表示最大限度使用物理内存，然后才是 swap空间，swappiness＝100的时候表示积极的使用swap分区，并且把内存上的数据及时的搬运到swap空间里面。
    现在一般1个G的内存可修改为10， 2个G的可改为5， 甚至是0。具体这样做：
    1.查看你的系统里面的swappiness
    $ cat /proc/sys/vm/swappiness

    2.修改swappiness值为10
    $ sudo sysctl vm.swappiness=10
    但是这只是临时性的修改，在你重启系统后会恢复默认的，为长治久安，还要更进一步：
    $ sudo gedit /etc/sysctl.conf
    在这个文档的最后加上这样一行：
    vm.swappiness=10
```

- 磁盘
  搜索引擎对磁盘的随机访问比较多，推荐使用 ssd 磁盘。

五.SolrCloud

1.SolrCloud简介

介绍

SolrCloud 是Solr提供的分布式搜索方案，当索引量很大，搜索请求并发很高，这时需要使用SolrCloud来满足这些需求。当一个系统的索引数据量少的时候是不需要使用SolrCloud的。

SolrCloud是基于Solr和Zookeeper的分布式搜索方案。它的主要思想是使用Zookeeper作为SolrCloud集群的配置信息中心，统一管理solrcloud的配置，比如solrconfig.xml和managed-schema。

场景

SolrCloud(solr集群)是Solr提供的分布式搜索方案，以下场景适合使用SolrCloud

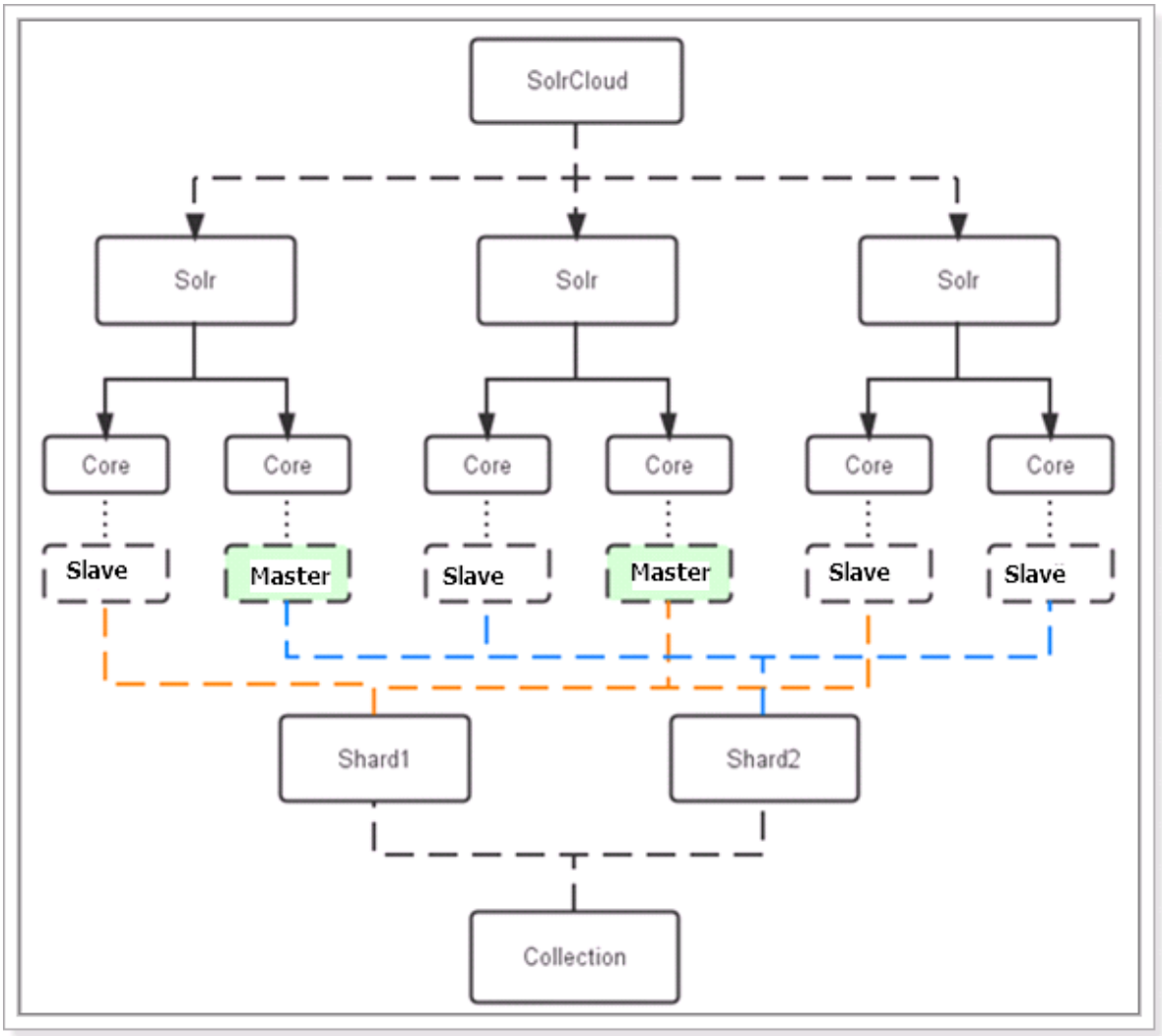
```
当你需要容错，分布式索引和分布式检索能力时使用SolrCloud。
当索引量很大，搜索请求并发很高时，同样需要使用SolrCloud来满足这些需求。
```

特点

```
集中式的配置信息
自动容错
近实时搜索
查询时自动负载均衡
```

2.Solrcloud的结构

Solrcloud结构图



物理结构

三个Solr实例（ 每个实例包括两个Core ），组成一个SolrCloud。

```
core：
    每个Core是Solr中一个独立运行单位，提供 索引和搜索服务。
```

逻辑结构

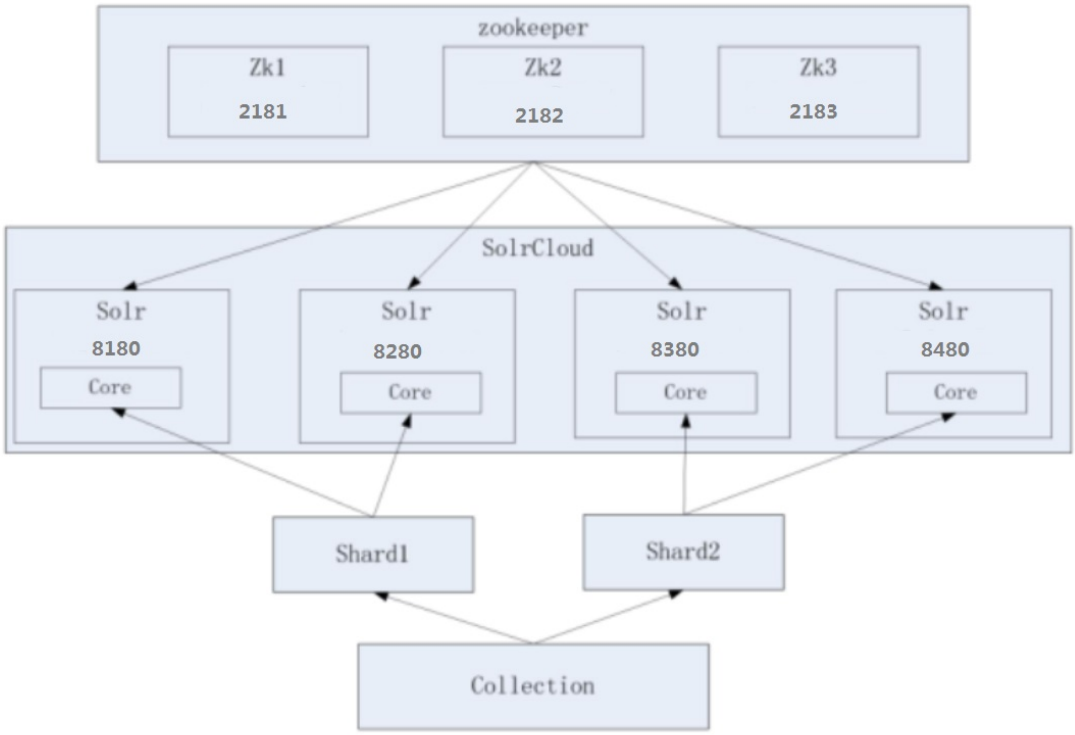
索引集合包括两个Shard（shard1和shard2），shard1和shard2分别由三个Core组成，其中一个Leader两个Replication，Leader是由zookeeper选举产生，zookeeper控制每个shard上三个Core的索引数据一致，解决高可用问题。用户发起索引请求分别从shard1和shard2上获取，解决高并发问题。

Master&Slave：
Master是master-slave构中的主结点（通常说主服务器），Slave是master-slave结构中的从结点（通常说从服务器或备服务器）。
Shard:
一个shard需要由一个Core或多个Core组成。Collection的逻辑分片。每个Shard被化成一个或者多个replication，通过选举确定哪个是Leader。索引集合包括两个Shard（shard1和shard2），shard1和shard2分别由三个Core组成，其中一个Leader两个Replication，Leader是由zookeeper选举产生，zookeeper控制每个shard上三个Core的索引数据一致，解决高可用问题。用户发起索引请求分别从shard1和shard2上获取，解决高并发问题。
collection:
多个shard组成所以collection一般由多个core组成。
Collection在SolrCloud集群中是一个逻辑意义上的完整的索引结构。它常常被划分为一个或多个Shard（分片），它们使用相同的配置信息。
比如：针对商品信息搜索可以创建一个collection。
collection=shard1+shard2+....+shardX

3.环境搭建

系统搭建结构图

solrcloud是基于solr和zookeeper部署，zookeeper是一个集群管理软件，solrcloud需要由多台solr服务器组成，然后由zookeeper来进行协调管理。



搭建ZK集群

1.准备一台centos 7服务器

2.安装

java -version

3.Zookeeper集群安装

第一步：把 zookeeper 的压缩包上传到服务器 解压缩。

第二步：在每个 zookeeper 目录下创建一个 data 目录。

第三步：把 zookeeper 复制三份。

```
[root[@localhost](https://my.oschina.net/u/570656) ~]# mkdir /usr/local/solr-cloud
[root[@localhost](https://my.oschina.net/u/570656) ~]# cp -r zookeeper-3.4.10 /usr/local/solr-cloud/zookeeper01
[root[@localhost](https://my.oschina.net/u/570656) ~]# cp -r zookeeper-3.4.10 /usr/local/solr-cloud/zookeeper02
[root[@localhost](https://my.oschina.net/u/570656) ~]# cp -r zookeeper-3.4.10 /usr/local/solr-cloud/zookeeper03
```

第四步：在 data 目录下创建一个 myid 文件，文件名就叫做“myid”。内容就是每个实例的id。例如1、2、3

[root@localhost data]# echo 1 >> myid (或者使用vim)

第五步：修改配置文件。把 conf 目录下的 zoo_sample.cfg 文件改名为 zoo.cfg

注意其它的zookeeper的dataDir和clientPort

```
cd /usr/local/solr-cloud/zookeeper01/conf/
> mv zoo_sample.cfg zoo.cfg
> vim zoo.cfg
tickTime=2000
dataDir=/usr/local/solr-cloud/zookeeper01/data
clientPort=2181
initLimit=10
syncLimit=5
server.1=192.168.211.137:2881:3881
server.2=192.168.211.137:2882:3882
server.3=192.168.211.137:2883:3883
```

5.启动测试

```
# 在三台机器上分别执行
> zkServer.sh start

# 查看状态
[root@solr-1 zookeeper01]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /usr/local/solr-cloud/zookeeper01/bin/../conf/zoo.cfg
Mode: follower

[root@solr-2 zookeeper02]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /usr/local/solr-cloud/zookeeper02/bin/../conf/zoo.cfg
Mode: leader

[root@solr-3 zookeeper03]# zkServer.sh status
ZooKeeper JMX enabled by default
```

```
Using config: /usr/local/solr-cloud/zookeeper03/bin/../conf/zoo.cfg
Mode: follower
```

搭建Solr集群

第一步: 先构建一个单机版的Solr

```
1.上传tomcat 和 solr 压缩包 到/usr/local/solr-cloud 下,tomcat中建立solr工程
    tar -xvf apache-tomcat-8.5.55.tar.gz
    mv apache-tomcat-8.5.55  tomcat
    tar  -xvf solr-7.7.3.tgz
    cp  /usr/local/solr-cloud/solr-7.7.3/server/solr-webapp/webapp -rf
/usr/local/solr-cloud/tomcat/webapps/solr

2.拷贝jar包
    cd  /usr/local/solr-cloud/solr-7.7.3/server/lib
    cp  ext/*  /usr/local/solr-cloud/tomcat/webapps/solr/WEB-INF/lib/
    cp  metrics-*  /usr/local/solr-cloud/tomcat/webapps/solr/WEB-INF/lib/
3.配置solrhome
  首先创建solrhome存储Solr索引文件
   mkdir  -p    /usr/local/solr-cloud/solrhome
  复制solr-7.7.3/server/solr目录下所有内容到solrhome
   cd    /usr/local/solr-cloud/solr-7.7.3/server/solr
   cp -r  *  /usr/local/solr-cloud/solrhome/
4.需要修改solr的web.xml文件 把solrhome关联起来 和 去掉安全认证。
  <env-entry>
    <env-entry-name>solr/home</env-entry-name>
    <env-entry-value>/usr/local/solr-cloud/solrhome</env-entry-value>
    <env-entry-type>java.lang.String</env-entry-type>
 </env-entry>
 <!--
  <security-constraint>
  ......
  -->
5.启动tomcat测试
./tomcat/bin/startup.sh
systemctl stop firewalld
```

第二步：创建四个tomcat实例。每个tomcat运行在不同的端口。8180、8280、8380、8480

分别修改tomcat中的端口

vi tomcat01/conf/server.xml

把原来的8005 换成 8105 之前的 8009换成 8109 之前的 8080 换成 8180 其它的tomcat 也类似修改

```
<Server port="8105" shutdown="SHUTDOWN"> </Server>
<Connector port="8180" protocol="HTTP/1.1" connectionTimeout="20000"
     redirectPort="8443" />
<Connector port="8109" protocol="AJP/1.3" redirectPort="8443" />
```

第三步:建立classes文件夹 拷贝日志配置文件。

```
mkdir  tomcat01/webapps/solr/WEB-INF/classes
cp   /usr/local/solr-cloud/solr-7.7.3/server/resources/log*  /usr/local/solr-
cloud/tomcat01/webapps/solr/WEB-INF/classes
```

其它的tomcat中的操作 参考上面的修改即可

第四步：把单机版solrhome 复制为solrhome1 solrhome2 solrhome3 solrhome4。

```
 cp   solrhome    -rf    solrhome1
 cp   solrhome    -rf    solrhome2
 cp   solrhome    -rf    solrhome3
 cp   solrhome    -rf    solrhome4
```

第五步：配置solrCloud相关的配置。每个solrhome下都有一个solr.xml，把其中的ip及端口号配置好。

这里重点是 hostPort 要和tomcat 对应 如果是多台机器 ip 设置不同即可

vi solrhome1/solr.xml 其它的solrhome/solr.xml 依次修改

```
<solrcloud>
    <str name="host">192.168.211.137</str>
    <int name="hostPort">8180</int>
    <str name="hostContext">${hostContext:solr}</str>
</solrcloud>
```

第六步：让zookeeper统一管理配置文件。需要把solrhome/configsets/sample_techproducts_configs/conf目录上传到zookeeper。上传任意solrhome中的配置文件即可。要保证zookeeper集群是启动状态。

使用工具上传配置文件：solr-7.7.3/server/scripts/cloud-scripts/zkcli.sh

```
cd  /usr/local/solr-cloud/solr-7.7.3/server/scripts/cloud-scripts

 ./zkcli.sh -zkhost
192.168.211.137:2181,192.168.211.137:2182,192.168.211.137:2183  -cmd upconfig -
confdir     /usr/local/solr-
cloud/solrhome1/configsets/sample_techproducts_configs/conf  -confname myconf
```

第七步：查看zookeeper上的配置文件：
使用zookeeper目录下的bin/zkCli.sh命令查看zookeeper上的配置文件：
[root@localhost bin]# ./zkCli.sh
[zk: localhost:2181(CONNECTED) 0] ls /
[configs, zookeeper]
[zk: localhost:2181(CONNECTED) 1] ls /configs
[myconf]
[zk: localhost:2181(CONNECTED) 2] ls /configs/myconf

第八步：修改tomcat/bin目录下的catalina.sh 文件，关联solr和zookeeper。
把此配置添加到配置文件中：

vi中搜索 umask

JAVA_OPTS="-DzkHost=192.168.211.137:2181,192.168.211.137:2182,192.168.211.137:2183"

第九步:检查每个tomcat中项目的web.xml 看是否已经对应到对应的solrhome 检查没有问题可以启动全部tomcat

tomcat01 中的web.xml 其它对应的修改

vi tomcat01/webapps/solr/WEB-INF/web.xml

```

solr/home
/usr/local/solr-cloud/solrhome1
java.lang.String

```

启动每个tomcat实例。

```
chmod  -R 777  tomcat01/bin
chmod  -R 777  tomcat02/bin
chmod  -R 777  tomcat03/bin
chmod  -R 777  tomcat04/bin
```

vi tomcat_start.sh

```
tomcat01/bin/startup.sh
tomcat02/bin/startup.sh
tomcat03/bin/startup.sh
tomcat04/bin/startup.sh
```

chmod 777 tomcat_start.sh

./tomcat_start.sh

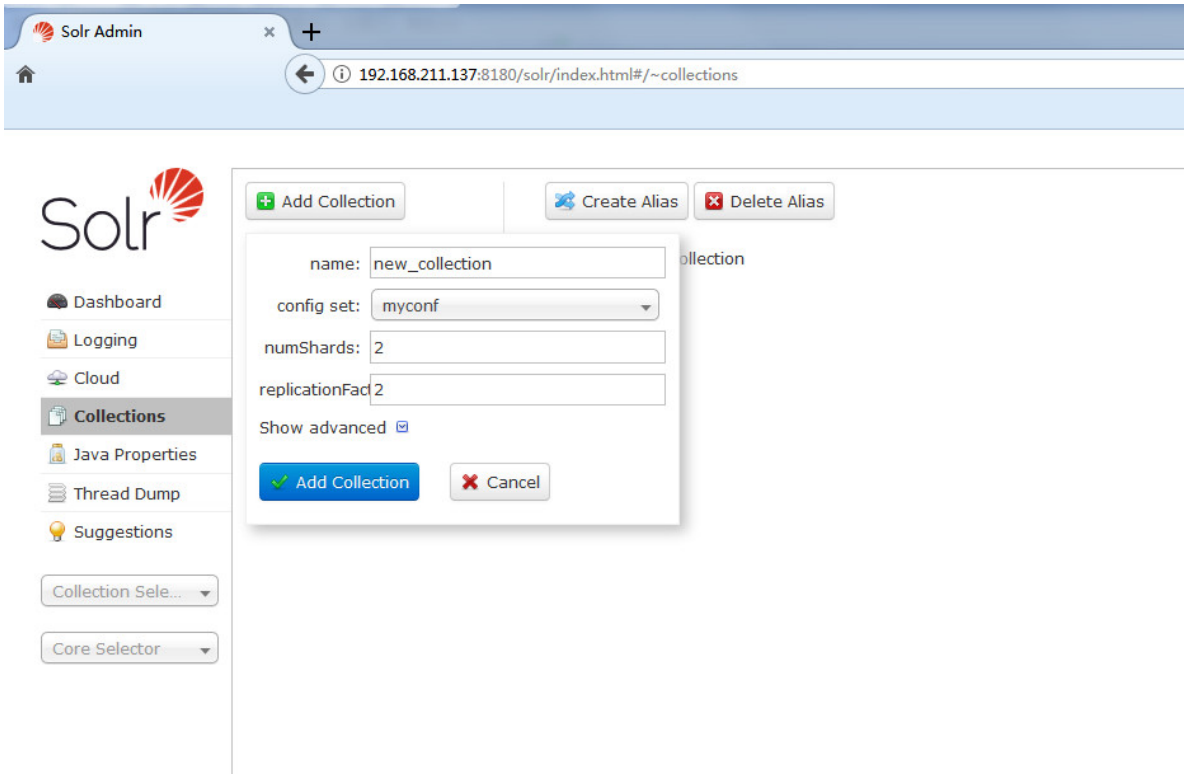
第十步：访问集群

http://192.168.211.137:8180/solr/index.html

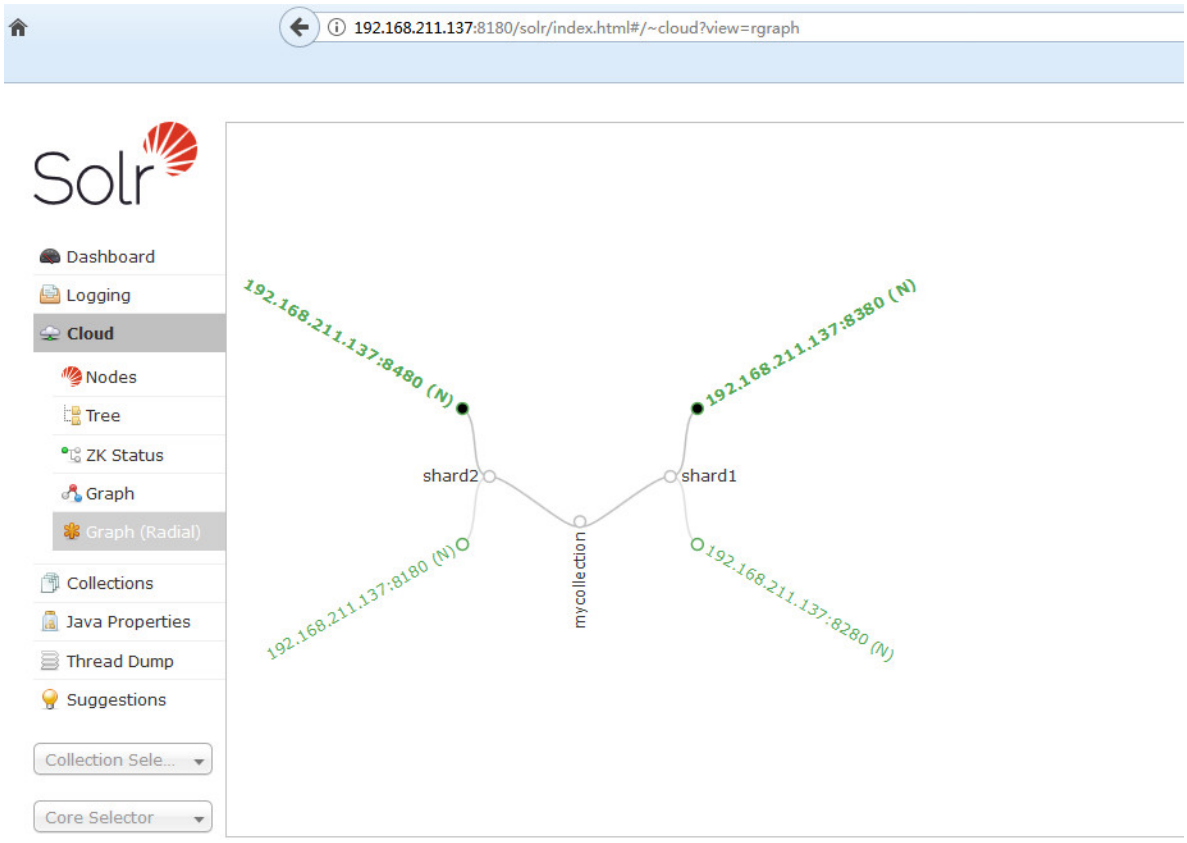
查看里面的 cloud 菜单

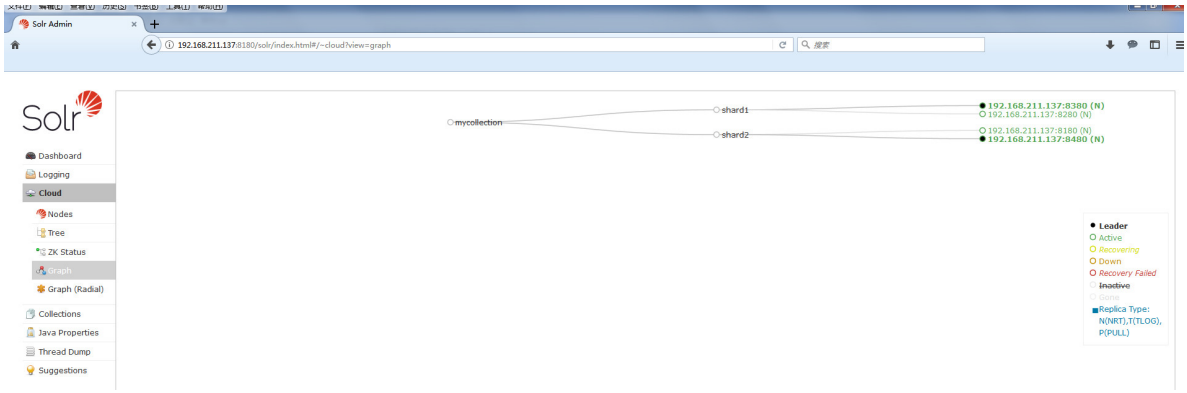
第十一步：创建新的Collection进行分片处理

点击页面的Collections按钮，然后就能添加了，我这里选择的是名字叫mycollection，用自己上传的myconf配置文件，有2片shard，每个shard有2个备份节点一主一备



然后回去看下，这样solr集群就搭建ok了！





4.SpringBoot 访问SolrCloud

配置文件

```
spring:
  data:
    solr:
      zk-host: 192.168.211.137:2181,192.168.211.137:2182,192.168.211.137:2183
```

配置CloudSolrClient

```
@Configuration
public class SolrConfig {

    @Value("${spring.data.solr.zk-host}")
    private String zkHost;

    @Bean
    public CloudSolrClient solrClient() {
        CloudSolrClient  cc  = new
CloudSolrClient.Builder().withZkHost(zkHost).build();
        cc.setDefaultCollection("mycollection");
        return cc;
    }

}
```

服务类中注入 CloudSolrClient

```
@Autowired
CloudSolrClient  solrClient;
// 操作时 可以指定collection的名字
```

注意:boot的版本 和 API细节