Broadview
www.broadview.com.cn

大型分布式网站架构
设计与实践

陈康贤 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

本书主要介绍了大型分布式网站架构所涉及的一些技术细节，包括 SOA 架构的实现、互联网安全架构、构建分布式网站所依赖的基础设施、系统稳定性保障和海量数据分析等内容；深入地讲述了大型分布式网站架构设计的核心原理，并通过一些架构设计的典型案例，帮助读者了解大型分布式网站设计的一些常见场景及遇到的问题。

作者结合自己在阿里巴巴及淘宝网的实际工作经历展开论述。本书既可供初学者学习，帮助读者了解大型分布式网站的架构，以及解决问题的思路和方法，也可供业界同行参考，给日常工作带来启发。

图书在版编目（CIP）数据

大型分布式网站架构设计与实践 / 陈康贤著. —北京：电子工业出版社，2014.9

ISBN 978-7-121-23885-7

Ⅰ. ①大… Ⅱ. ①陈… Ⅲ. ①网站－建设 Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字（2014）第 169308 号

策划编辑：董　英
责任编辑：陈晓猛
印　　刷：北京中新伟业印刷有限公司
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　　印张：28.75　　字数：640 千字
版　　次：2014 年 9 月第 1 版
印　　次：2014 年 9 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei. com.cn。

服务热线：（010）88258888。

第 2 章

分布式系统基础设施

一个大型、稳健、成熟的分布式系统的背后，往往会涉及众多的支撑系统，我们将这些支撑系统称为分布式系统的基础设施。除了前面所介绍的分布式协作及配置管理系统 ZooKeeper，我们进行系统架构设计所依赖的基础设施，还包括分布式缓存系统、持久化存储、分布式消息系统、搜索引擎，以及 CDN 系统、负载均衡系统、运维自动化系统等，还有后面章节所要介绍的实时计算系统、离线计算系统、分布式文件系统、日志收集系统、监控系统、数据仓库等。

分布式缓存主要用于在高并发环境下，减轻数据库的压力，提高系统的响应速度和并发吞吐。当大量的读、写请求涌向数据库时，磁盘的处理速度与内存显然不在一个量级，因此，在数据库之前加一层缓存，能够显著提高系统的响应速度，并降低数据库的压力。

作为传统的关系型数据库，MySQL 提供完整的 ACID 操作，支持丰富的数据类型、强大的关联查询、where 语句等，能够非常容易地建立查询索引，执行复杂的内连接、外连接、求和、排序、分组等操作，并且支持存储过程、函数等功能，产品成熟度高，功能强大。但是，对于需要应对高并发访问并且存储海量数据的场景来说，出于对性能的考虑，不得不放弃很多传统关系型数据库原本强大的功能，牺牲了系统的易用性，并且使得系统的设计和管理变得更为复杂。这也使得在过去几年中，流行着另一种新的存储解决方案——NoSQL，它与传统的关系型数据库最大的差别在于，它不使用 SQL 作为查询语言来查找数据，而采用 key-value 形式进行查找，提供了更高的查询效率及吞吐，并且能够更加方便地进行扩展，存储海量数据，在数千个节点上进行分区，自动进行数据的复制和备份。

在分布式系统中，消息作为应用间通信的一种方式，得到了十分广泛的应用。消息可以被保存在队列中，直到被接收者取出，由于消息发送者不需要同步等待消息接收者的响应，消息的异步接收降低了系统集成的耦合度，提升了分布式系统协作的效率，使得系统能够更快地响应用户，提供更高的吞吐。当系统处于峰值压力时，分布式消息队列还能够作为缓冲，削峰填谷，缓解集群的压力，避免整个系统被压垮。

垂直化的搜索引擎在分布式系统中是一个非常重要的角色，它既能够满足用户对于全文检索、模糊匹配的需求，解决数据库 like 查询效率低下的问题，又能够解决分布式环境下，由于采用分库分表，或者使用 NoSQL 数据库，导致无法进行多表关联或者进行复杂查询的问题。

本章主要介绍和解决如下问题：

- 分布式缓存 memcache 的使用及分布式策略，包括 Hash 算法的选择。
- 常见的分布式系统存储解决方案，包括 MySQL 的分布式扩展、HBase 的 API 及使用场景、Redis 的使用等。
- 如何使用分布式消息系统 ActiveMQ 来降低系统之间的耦合度，以及进行应用间的通信。
- 垂直化的搜索引擎在分布式系统中的使用，包括搜索引擎的基本原理、Lucene 详细的使用介绍，以及基于 Lucene 的开源搜索引擎工具 Solr 的使用。

2.1 分布式缓存

在高并发环境下，大量的读、写请求涌向数据库，磁盘的处理速度与内存显然不在一个量级，从减轻数据库的压力和提高系统响应速度两个角度来考虑，一般都会在数据库之前加一层缓存。由于单台机器的内存资源和承载能力有限，并且如果大量使用本地缓存，也会使相同的数据被不同的节点存储多份，对内存资源造成较大的浪费，因此才催生出了分布式缓存。

本节将详细介绍分布式缓存的典型代表 memcache，以及分布式缓存的应用场景。最为典型的场景莫过于分布式 session。

2.1.1 memcache 简介及安装

memcache[1]是 danga.com 的一个项目，它是一款开源的高性能的分布式内存对象缓存系统，最早是给 LiveJournal[2]提供服务的，后来逐渐被越来越多的大型网站所采用，用于在应用中减少对数据库的访问，提高应用的访问速度，并降低数据库的负载。

为了在内存中提供数据的高速查找能力，memcache 使用 key-value 形式存储和访问数据，在内存中维护一张巨大的 HashTable，使得对数据查询的时间复杂度降低到 O（1），保证了对数据的高性能访问。内存的空间总是有限的，当内存没有更多的空间来存储新的数据时，memcache 就会使用 LRU（Least Recently Used）算法，将最近不常访问的数据淘汰掉，以腾出空间来存放新的数据。memcache 存储支持的数据格式也是灵活多样的，通过对象的序列化机制，可以将更高层抽象的对象转换成为二进制数据，存储在缓存服务器中，当前端应用需要时，又可以通过二进制内容反序列化，将数据还原成原有对象。

1. memcache 的安装

由于 memcache 使用了 libevent 来进行高效的网络连接处理，因此在安装 memcache 之前，需要先安装 libevent。

下载 libevent[3]，这里采用的是 1.4.14 版本的 libevent。

```
wget https://github.com/downloads/libevent/libevent/libevent-1.4.14b-
stable.tar.gz
```

1 memcache 项目地址为 http://memcached.org。

2 LiveJournal，http://www.livejournal.com。

3 libevent，http://libevent.org。

```
longlong@ubuntu:~/temp$ wget https://github.com/downloads/libevent/libevent/libe
vent-1.4.14b-stable.tar.gz
--2014-03-19 04:52:41--  https://github.com/downloads/libevent/libevent/libevent
-1.4.14b-stable.tar.gz
Resolving github.com (github.com)... 192.30.252.129
Connecting to github.com (github.com)|192.30.252.129|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://cloud.github.com/downloads/libevent/libevent/libevent-1.4.14b-s
table.tar.gz [following]
--2014-03-19 04:52:55--  http://cloud.github.com/downloads/libevent/libevent/lib
event-1.4.14b-stable.tar.gz
Resolving cloud.github.com (cloud.github.com)... 205.251.212.145, 205.251.212.82
, 54.230.126.248, ...
Connecting to cloud.github.com (cloud.github.com)|205.251.212.145|:80... connect
ed.
HTTP request sent, awaiting response... 302 Found
Location: http://218.108.192.119:80/1Q2W3E4R5T6Y7U8I9O0P1Z2X3C4V5B/cloud.github.
com/downloads/libevent/libevent/libevent-1.4.14b-stable.tar.gz [following]
--2014-03-19 04:53:04--  http://218.108.192.119/1Q2W3E4R5T6Y7U8I9O0P1Z2X3C4V5B/c
loud.github.com/downloads/libevent/libevent/libevent-1.4.14b-stable.tar.gz
Connecting to 218.108.192.119:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 474874 (464K) [application/gzip]
Saving to: `libevent-1.4.14b-stable.tar.gz'
```

解压：

```
tar –xf  libevent-1.4.14b-stable.tar.gz
```

```
longlong@ubuntu:~/temp$ tar -xf libevent-1.4.14b-stable.tar.gz
longlong@ubuntu:~/temp$
```

配置、编译、安装 libevent：

```
./configure
```

```
longlong@ubuntu:~/temp/libevent-1.4.14b-stable$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
```

```
make
```

```
longlong@ubuntu:~/temp/libevent-1.4.14b-stable$ make
echo '/* event-config.h' > event-config.h
echo ' * Generated by autoconf; post-processed by libevent.' >> event-config.h
echo ' * Do not edit this file.' >> event-config.h
echo ' * Do not rely on macros in this file existing in later versions.'>> event
-config.h
echo ' */' >> event-config.h
echo '#ifndef _EVENT_CONFIG_H_' >> event-config.h
echo '#define _EVENT_CONFIG_H_' >> event-config.h
sed -e 's/#define /#define _EVENT_/' \
            -e 's/#undef /#undef _EVENT_/' \
            -e 's/#ifndef /#ifndef _EVENT_/' < config.h >> event-config.h
echo "#endif" >> event-config.h
make  all-recursive
```

```
sudo make install
```

```
longlong@ubuntu:~/temp/libevent-1.4.14b-stable$ sudo make install
[sudo] password for longlong:
make  install-recursive
make[1]: Entering directory `/home/longlong/temp/libevent-1.4.14b-stable'
Making install in .
make[2]: Entering directory `/home/longlong/temp/libevent-1.4.14b-stable'
make[3]: Entering directory `/home/longlong/temp/libevent-1.4.14b-stable'
test -z "/usr/local/bin" || /bin/mkdir -p "/usr/local/bin"
 /usr/bin/install -c event_rpcgen.py '/usr/local/bin'
test -z "/usr/local/lib" || /bin/mkdir -p "/usr/local/lib"
 /bin/bash ./libtool   --mode=install /usr/bin/install -c   libevent.la libevent
_core.la libevent_extra.la '/usr/local/lib'
libtool: install: /usr/bin/install -c .libs/libevent-1.4.so.2.2.0 /usr/local/lib
/libevent-1.4.so.2.2.0
libtool: install: (cd /usr/local/lib && { ln -s -f libevent-1.4.so.2.2.0 libeven
t-1.4.so.2 || { rm -f libevent-1.4.so.2 && ln -s libevent-1.4.so.2.2.0 libevent-
1.4.so.2; }; })
libtool: install: (cd /usr/local/lib && { ln -s -f libevent-1.4.so.2.2.0 libeven
t.so || { rm -f libevent.so && ln -s libevent-1.4.so.2.2.0 libevent.so; }; })
libtool: install: /usr/bin/install -c .libs/libevent.lai /usr/local/lib/libevent
```

下载 memcache，并解压：

```
wget  http://www.memcached.org/files/memcached-1.4.17.tar.gz
```

```
longlong@ubuntu:~/temp$ wget http://www.memcached.org/files/memcached-1.4.17.tar
.gz
--2014-03-19 05:02:12--  http://www.memcached.org/files/memcached-1.4.17.tar.gz
Resolving www.memcached.org (www.memcached.org)... 69.46.88.68
Connecting to www.memcached.org (www.memcached.org)|69.46.88.68|:80... connected
.
HTTP request sent, awaiting response... 302 Found
Location: http://218.108.192.145:80/1Q2W3E4R5T6Y7U8I9O0P1Z2X3C4V5B/www.memcached
.org/files/memcached-1.4.17.tar.gz [following]
--2014-03-19 05:02:24--  http://218.108.192.145/1Q2W3E4R5T6Y7U8I9O0P1Z2X3C4V5B/w
ww.memcached.org/files/memcached-1.4.17.tar.gz
Connecting to 218.108.192.145:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 326970 (319K) [application/x-tar]
Saving to: `memcached-1.4.17.tar.gz'

100%[======================================>] 326,970     1.78M/s   in 0.2s

2014-03-19 05:02:24 (1.78 MB/s) - `memcached-1.4.17.tar.gz' saved [326970/326970
]
```

```
tar –xf  memcached-1.4.17.tar.gz
```

```
longlong@ubuntu:~/temp$ tar -xf memcached-1.4.17.tar.gz
longlong@ubuntu:~/temp$
```

配置、编译、安装 memcache：

```
./configure
```

```
longlong@ubuntu:~/temp/memcached-1.4.17$ ./configure
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
```

make

```
longlong@ubuntu:~/temp/memcached-1.4.17$ make
make  all-recursive
make[1]: Entering directory `/home/longlong/temp/memcached-1.4.17'
Making all in doc
make[2]: Entering directory `/home/longlong/temp/memcached-1.4.17/doc'
make  all-am
make[3]: Entering directory `/home/longlong/temp/memcached-1.4.17/doc'
make[3]: Nothing to be done for `all-am'.
make[3]: Leaving directory `/home/longlong/temp/memcached-1.4.17/doc'
make[2]: Leaving directory `/home/longlong/temp/memcached-1.4.17/doc'
make[2]: Entering directory `/home/longlong/temp/memcached-1.4.17'
gcc -std=gnu99 -DHAVE_CONFIG_H -I.  -DNDEBUG   -g -O2 -pthread -pthread -Wall -W
error -pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -M
T memcached-memcached.o -MD -MP -MF .deps/memcached-memcached.Tpo -c -o memcache
d-memcached.o `test -f 'memcached.c' || echo './'`memcached.c
mv -f .deps/memcached-memcached.Tpo .deps/memcached-memcached.Po
gcc -std=gnu99 -DHAVE_CONFIG_H -I.  -DNDEBUG   -g -O2 -pthread -pthread -Wall -W
error -pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -M
T memcached-hash.o -MD -MP -MF .deps/memcached-hash.Tpo -c -o memcached-hash.o `
```

sudo make install

```
longlong@ubuntu:~/temp/memcached-1.4.17$ sudo make install
make  install-recursive
make[1]: Entering directory `/home/longlong/temp/memcached-1.4.17'
Making install in doc
make[2]: Entering directory `/home/longlong/temp/memcached-1.4.17/doc'
make  install-am
make[3]: Entering directory `/home/longlong/temp/memcached-1.4.17/doc'
make[4]: Entering directory `/home/longlong/temp/memcached-1.4.17/doc'
make[4]: Nothing to be done for `install-exec-am'.
test -z "/usr/local/share/man/man1" || /bin/mkdir -p "/usr/local/share/man/man1"
 /usr/bin/install -c -m 644 memcached.1 '/usr/local/share/man/man1'
make[4]: Leaving directory `/home/longlong/temp/memcached-1.4.17/doc'
make[3]: Leaving directory `/home/longlong/temp/memcached-1.4.17/doc'
make[2]: Leaving directory `/home/longlong/temp/memcached-1.4.17/doc'
make[2]: Entering directory `/home/longlong/temp/memcached-1.4.17'
make[3]: Entering directory `/home/longlong/temp/memcached-1.4.17'
test -z "/usr/local/bin" || /bin/mkdir -p "/usr/local/bin"
```

2. 启动与关闭 memcache

启动 memcache 服务：

```
/usr/local/bin/memcached -d -m 10 -u root -l 192.168.136.135 -p 11211 -c 32
-P /tmp/memcached.pid
```

参数的含义如下：

- -d 表示启动的是一个守护进程；
- -m 指定分配给 memcache 的内存数量，单位是 MB，这里指定的是 10 MB。
- -u 指定运行 memcache 的用户，这里指定的是 root；
- -l 指定监听的服务器的 IP 地址；
- -p 设置 memcache 监听的端口，这里指定的是 11211；
- -c 指定最大允许的并发连接数，这里设置为 32；
- -P 指定 memcache 的 pid 文件保存的位置。

```
longlong@ubuntu:~/temp/memcached-1.4.17$ /usr/local/bin/memcached -d -m 10 -u ro
ot -l 192.168.136.135 -p 11211 -c 256 -P /tmp/memcached.pid
longlong@ubuntu:~/temp/memcached-1.4.17$ ps -aux | grep memcached
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
longlong  8132  1.0  0.0  47996   816 ?        Ssl  05:57   0:00 /usr/local/bin/
memcached -d -m 10 -u root -l 192.168.136.135 -p 11211 -c 256 -P /tmp/memcached.
pid
```

关闭 memcache 服务：

```
kill `cat /tmp/memcached.pid`
```

```
longlong@ubuntu:~/temp/memcached-1.4.17$ kill `cat /tmp/memcached.pid`
longlong@ubuntu:~/temp/memcached-1.4.17$ ps -aux | grep memcached
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
```

2.1.2 memcache API 与分布式

memcache 客户端与服务端通过构建在 TCP 协议之上的 memcache 协议[4]来进行通信，协议支持两种数据的传递，这两种数据分别为文本行和非结构化数据。文本行主要用来承载客户端的命令及服务端的响应，而非结构化数据则主要用于客户端和服务端数据的传递。由于非结构化数据采用字节流的形式在客户端和服务端之间进行传输和存储，因此使用方式非常灵活，缓存数据存储几乎没有任何限制，并且服务端也不需要关心存储的具体内容及字节序。

memcache 协议支持通过如下几种方式来读取/写入/失效数据：

4 memcache 协议见 https://github.com/memcached/memcached/blob/master/doc/protocol.txt。

- set 将数据保存到缓存服务器，如果缓存服务器存在同样的 key，则替换之；
- add 将数据新增到缓存服务器，如果缓存服务器存在同样的 key，则新增失败；
- replace 将数据替换缓存服务器中相同的 key，如果缓存服务器不存在同样的 key，则替换失败；
- append 将数据追加到已经存在的数据后面；
- prepend 将数据追加到已经存在的数据前面；
- cas 提供对变量的 cas 操作，它将保证在进行数据更新之前，数据没有被其他人更改；
- get 从缓存服务器获取数据；
- incr 对计数器进行增量操作；
- decr 对计数器进行减量操作；
- delete 将缓存服务器上的数据删除。

memcache 官方提供的 Memcached-Java-Client[5]工具包含了对 memcache 协议的 Java 封装，使用它可以比较方便地与缓存服务端进行通信，它的初始化方式如下：

```
public static void init(){
    String[] servers = {
        "192.168.136.135:11211"
    };
    SockIOPool pool = SockIOPool.getInstance();
    pool.setServers(servers);//设置服务器
    pool.setFailover(true);//容错
    pool.setInitConn(10);//设置初始连接数
    pool.setMinConn(5);//设置最小连接数
    pool.setMaxConn(25); //设置最大连接数
    pool.setMaintSleep(30);//设置连接池维护线程的睡眠时间
    pool.setNagle(false);//设置是否使用 Nagle 算法
    pool.setSocketTO(3000);//设置 socket 的读取等待超时时间
    pool.setAliveCheck(true);//设置连接心跳监测开关
    pool.setHashingAlg(SockIOPool.CONSISTENT_HASH);//设置 Hash 算法
    pool.initialize();
}
```

通过 SockIOPool，可以设置与后端缓存服务器的一系列参数，如服务器地址、是否采用容

5 Memcached-Java-Client，https://github.com/gwhalin/Memcached-Java-Client。

错、初始连接数、最大连接数、最小连接数、线程睡眠时间、是否使用 Nagle 算法、socket 的读取等待超时时间、是否心跳检测、Hash 算法，等等。

使用 Memcached-Java-Client 的 API 设置缓存的值：

```
MemCachedClient memCachedClient = new MemCachedClient();
memCachedClient.add("key", 1);
memCachedClient.set("key", 2);
memCachedClient.replace("key", 3);
```

通过 add()方法新增缓存，如果缓存服务器存在同样的 key，则返回 false；而通过 set()方法将数据保存到缓存服务器，缓存服务器如果存在同样的 key，则将其替换。replace()方法可以用来替换服务器中相同的 key 的值，如果缓存服务器不存在这样的 key，则返回 false。

使用 Memcached-Java-Client 的 API 获取缓存的值：

```
Object value = memCachedClient.get("key");
String[] keys = {"key1","key2"};
Map<String, Object> values = memCachedClient.getMulti(keys);
```

通过 get()方法，可以从服务器获取该 key 对应的数据；而使用 getMulti()方法，则可以一次性从缓存服务器获取一组数据。

对缓存的值进行 append 和 prepend 操作：

```
memCachedClient.set("key-name", "chenkangxian");
memCachedClient.prepend("key-name", "hello");
memCachedClient.append("key-name", "!");
```

通过 prepend()方法，可以在对应 key 的值前面增加前缀；而通过 append()方法，则可以在对应的 key 的值后面追加后缀。

对缓存的数据进行 cas[6]操作：

```
MemcachedItem  item = memCachedClient.gets("key");
memCachedClient.cas("key", (Integer)item.getValue() + 1,
                                    item.getCasUnique());
```

通过 gets()方法获得 key 对应的值和值的版本号，它们包含在 MemcachedItem 对象中；然后使用 cas()方法对该值进行修改，当 key 对应的版本号与通过 gets 取到的版本号（即 item.getCasUnique()）相同时，则将 key 对应的值修改为 item.getValue() + 1，这样可以防止并发修改所带来的问题。

---

6 memcache 的 CAS 有点类似 Java 的 CAS（compare and set）操作，关于 Java 的 CAS 操作，第 4 章会有详细介绍。

对缓存的数据进行增量与减量操作：

```
memCachedClient.incr("key",1);
memCachedClient.decr("key",1);
```

使用 incr()方法可以对 key 对应的值进行增量操作，而使用 decr()方法则可以对 key 对应的值进行减量操作。

memcache 本身并不是一种分布式的缓存系统，它的分布式是由访问它的客户端来实现的。一种比较简单的实现方式是根据缓存的 key 来进行 Hash，当后端有 *N* 台缓存服务器时，访问的服务器为 hash(key)%N，这样可以将前端的请求均衡地映射到后端的缓存服务器，如图 2-1 所示。但这样也会导致一个问题，一旦后端某台缓存服务器宕机，或者是由于集群压力过大，需要新增缓存服务器时，大部分的 key 将会重新分布。对于高并发系统来说，这可能会演变成一场灾难，所有的请求将如洪水般疯狂地涌向后端的数据库服务器，而数据库服务器的不可用，将会导致整个应用的不可用，形成所谓的“雪崩效应”。

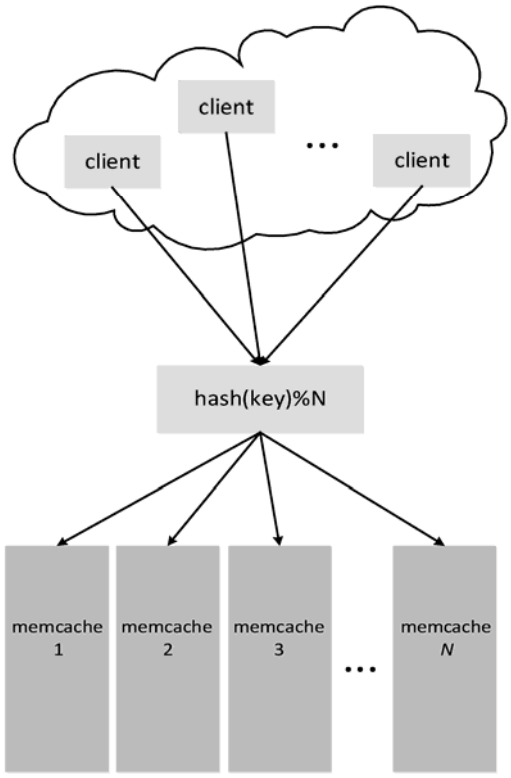


图 2-1　memcache 集群采用 hash(key)%N 进行分布

使用 consistent Hash 算法能够在一定程度上改善上述问题。该算法早在 1997 年就在论文

Consistent hashing and random trees[7]中被提出，它能够在移除/添加一台缓存服务器时，尽可能小地改变已存在的 key 映射关系，避免大量 key 的重新映射。

consistent Hash 的原理是这样的，它将 Hash 函数的值域空间组织成一个圆环，假设 Hash 函数的值域空间为 0～$2^{32}-1$（即 Hash 值是一个 32 位的无符号整型），整个空间按照顺时针方向进行组织，然后对相应的服务器节点进行 Hash，将它们映射到 Hash 环上，假设有 4 台服务器，分别为 node1、node2、node3、node4，它们在环上的位置如图 2-2 所示。

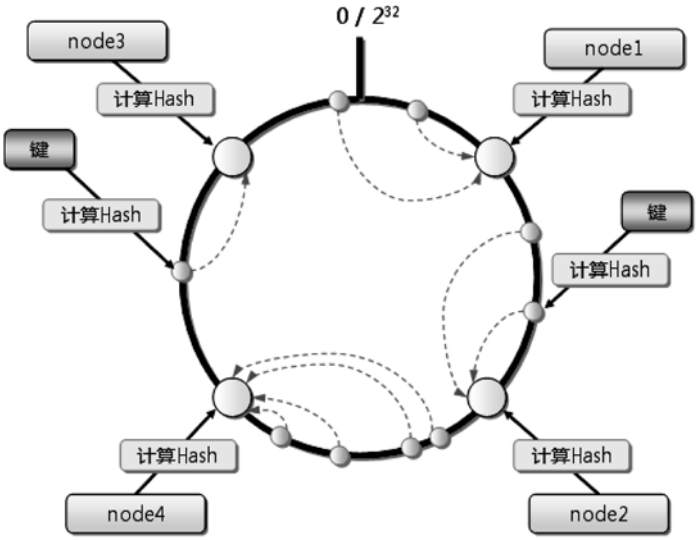


图 2-2　consistent Hash 的原理

接下来使用相同的 Hash 函数，计算出对应的 key 的 Hash 值在环上对应的位置。根据 consistent Hash 算法，按照顺时针方向，分布在 node1 与 node2 之间的 key，它们的访问请求会被定位到 node2，而 node2 与 node4 之间的 key，访问请求会被定为到 node4，以此类推。

假设有新节点 node5 增加进来时，假设它被 Hash 到 node2 和 node4 之间，如图 2-3 所示。那么受影响的只有 node2 和 node5 之间的 key，它们将被重新映射到 node5，而其他 key 的映射关系将不会发生改变，这样便避免了大量 key 的重新映射。

当然，上面描绘的只是一种理想的情况，各个节点在环上分布得十分均匀。正常情况下，当节点数量较少时，节点的分布可能十分不均匀，从而导致数据访问的倾斜，大量的 key 被映射到同一台服务器上。为了避免这种情况的出现，可以引入虚拟节点机制，对每一个服务器节点都计算多个 Hash 值，每一个 Hash 值都对应环上一个节点的位置，该节点称为虚拟节点，而 key 的映射方式不变，只是多了一步从虚拟节点再映射到真实节点的过程。这样，如果虚拟节

7 consistent hash，http://dl.acm.org/citation.cfm?id=258660。

点的数量足够多，即使只有很少的实际节点，也能够使 key 分布得相对均衡。

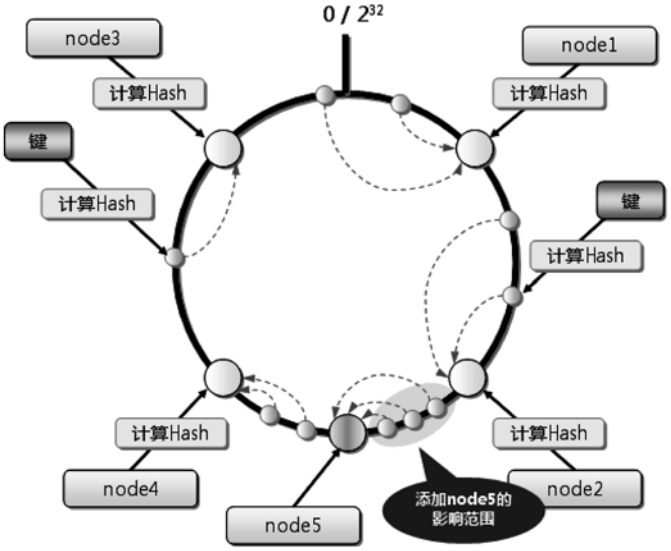


图 2-3　当新节点加入时的情景[8]

2.1.3　分布式 session

传统的应用服务器，如 tomcat、jboss 等，其自身所实现的 session 管理大部分都是基于单机的。对于大型分布式网站来说，支撑其业务的远远不止一台服务器，而是一个分布式集群，请求在不同服务器之间跳转。那么，如何保持服务器之间的 session 同步呢？传统网站一般通过将一部分数据存储在 cookie 中，来规避分布式环境下 session 的操作。这样做的弊端很多，一方面 cookie 的安全性一直广为诟病，另一方面 cookie 存储数据的大小是有限制的。随着移动互联网的发展，很多情况下还得兼顾移动端的 session 需求，使得采用 cookie 来进行 session 同步的方式的弊端更为凸显。分布式 session 正是在这种情况下应运而生的。

对于系统可靠性要求较高的用户，可以将 session 持久化到 DB 中，这样可以保证宕机时会话不易丢失，但缺点也是显而易见的，系统的整体吞吐将受到很大的影响。另一种解决方案便是将 session 统一存储在缓存集群上，如 memcache，这样可以保证较高的读、写性能，这一点对于并发量大的系统来说非常重要；并且从安全性考虑，session 毕竟是有有效期的，使用缓存存储，也便于利用缓存的失效机制。使用缓存的缺点是，一旦缓存重启，里面保存的会话也就丢失了，需要用户重新建立会话。

如图 2-4 所示，前端用户请求经过随机分发之后，可能会命中后端任意的 Web Server，并

8 图片来源 http://blog.charlee.li/content/images/2008/Jul/memcached-0004-05.png。

且 Web Server 也可能会因为各种不确定的原因宕机。在这种情况下，session 是很难在集群间同步的，而通过将 session 以 sessionid 作为 key，保存到后端的缓存集群中，使得不管请求如何分配，即便是 Web Server 宕机，也不会影响其他 Web Server 通过 sessionid 从 Cache Server 中获得 session，这样既实现了集群间的 session 同步，又提高了 Web Server 的容错性。

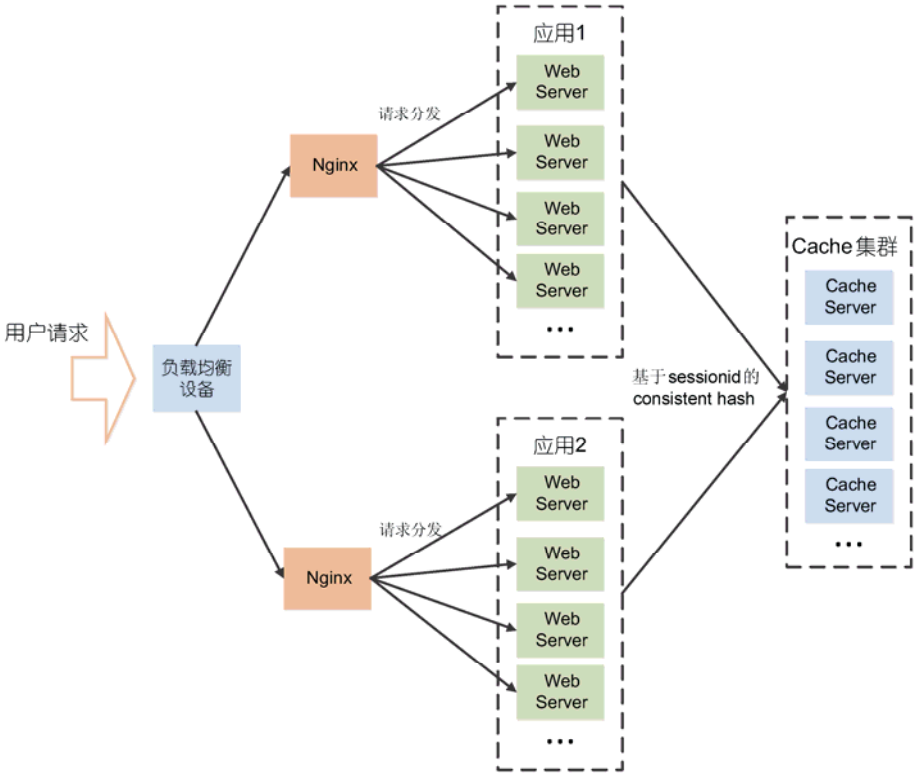


图 2-4　基于缓存的分布式 session 架构

这里以 Tomcat 作为 Web Server 来举例，通过一个简单的工具 memcached-session- manager[9]，实现基于 memcache 的分布式 session。

memcached-session-manager 是一个开源的高可用的 Tomcat session 共享解决方案，它支持 Sticky 模式和 Non-Sticky 模式。Sticky 模式表示每次请求都会被映射到同一台后端 Web Server，直到该 Web Server 宕机，这样 session 可先存放在服务器本地，等到请求处理完成再同步到后端 memcache 服务器；而当 Web Server 宕机时，请求被映射到其他 Web Server，这时候，其他 Web Server 可以从后端 memcache 中恢复 session。对于 Non-Sticky 模式来说，请求每次映射的后端 Web Server 是不确定的，当请求到来时，从 memcache 中加载 session；当请求处理完成时，将

9 memcached-session-manager，https://code.google.com/p/memcached-session-manager。

session 再写回到 memcache。

以 Non-Sticky 模式为例，它需要给 Tomcat 的$CATALINA_HOME/conf/context.xml 文件配置 SessionManager，具体配置如下：

```
<Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
  memcachedNodes="n1:192.168.0.100:11211,n2:192.168.0.101:11211"
  sticky="false"
  sessionBackupAsync="false"
  lockingMode="auto"
  requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
 transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
 />
```

其中：memcachedNodes 指定了 memcache 的节点；sticky 表示是否采用 Sticky 模式；sessionBackupAsync 表示是否采用异步方式备份 session；lockingMode 表示 session 的锁定模式；auto 表示对于只读请求，session 将不会被锁定，如果包含写入请求，则 session 会被锁定；requestUriIgnorePattern 表示忽略的 url；transcoderFactoryClass 用来指定序列化的方式，这里采用的是 Kryo 序列化，也是 memcached-session-manager 比较推荐的一种序列化方式。

memcached-session-manager 依赖于 memcached-session-manager-${version}.jar，如果使用的是 tomcat6，则还需要下载 memcached-session-manager-tc6-${version}.jar，并且它还依赖 memcached-${version}.jar 进行 memcache 的访问。在启动 Tomcat 之前，需要将这些 jar 放在$CATALINA_HOME/lib/目录下。如果使用第三方序列化方式，如 Kryo，还需要在 Web 工程中引入相关的第三方库，Kryo 序列化所依赖的库，包括 kryo-${version}-all.jar、kryo-serializers-${version}.jar 和 msm-kryo-serializer. ${version}.jar。

2.2　持久化存储

随着科技的不断发展，越来越多的人开始参与到互联网活动中来，人们在网络上的活动，如发表心情动态、微博、购物、评论等，这些信息最终被转变成二进制字节的数据存储下来。面对并发访问量的激增和数据量几何级的增长，如何存储正在迅速膨胀并且不断累积的数据，以及应对日益增长的用户访问频次，成为了亟待解决的问题。

传统的 IOE[10]解决方案，使用和扩展的成本越来越高，使得互联网企业不得不思考新的解决方案。开源软件加廉价 PC Server 的分布式架构，得益于社区的支持。在节约成本的同时，也给系统带来了良好的扩展能力，并且由于开源软件的代码透明，使得企业能够以更低的代价定制

10 I 表示 IBM 小型机，O 表示 oracle 数据库，E 表示 EMC 高端存储。

更符合自身使用场景的功能，以提高系统的整体性能。本节将介绍互联网领域常见的三种数据存储方式，包括传统关系型数据库 MySQL、Google 所提出的 bigtable 概念及其开源实现 HBase，以及包含丰富数据类型的 key-value 存储 Redis。

作为传统的关系型数据库，MySQL 提供完整的 ACID 操作，支持丰富的数据类型、强大的关联查询、where 语句等，能够非常容易地建立查询索引，执行复杂的内连接、外连接、求和、排序、分组等操作，并且支持存储过程、函数等功能，产品成熟度高，功能强大。对于大多数中小规模的应用来说，关系型数据库拥有强大完整的功能，以及提供的易用性、灵活性和产品成熟度，地位很难被完全替代。但是，对于需要应对高并发访问并且存储海量数据的场景来说，出于性能的考虑，不得不放弃很多传统关系型数据的功能，如关联查询、事务、数据一致性（由强一致性降为最终一致性）；并且由于对数据存储进行拆分，如分库分表，以及进行反范式设计，以提高系统的查询性能，使得我们放弃了关系型数据库大部分原本强大的功能，牺牲了系统的易用性，并且使得系统的设计和管理变得更为复杂。

过去几年中，流行着一种新的存储解决方案，NoSQL、HBase 和 Redis 作为其中较为典型的代表，各自都得到了较为广泛的使用，它们各自都具有比较鲜明的特性。与传统的关系型数据库相比，HBase 有更好的伸缩能力，更适合于海量数据的存储和处理，并且 HBase 能够支持多个 Region Server 同时写入，并发写入性能十分出色。但 HBase 本身所支持的查询维度有限，难以支持复杂的条件查询，如 group by、order by、join 等，这些特点使它的应用场景受到了限制。对于 Redis 来说，它拥有更好的读/写吞吐能力，能够支撑更高的并发数，而相较于其他的 key-value 类型的数据库，Redis 能够提供更为丰富的数据类型支持，能更灵活地满足业务需求。

2.2.1 MySQL 扩展

随着互联网行业的高速发展，使得采用诸如 IOE 等商用存储解决方案的成本不断攀升，越来越难以满足企业高速发展的需要；因此，开源的存储解决方案开始逐渐受到青睐，并成为互联网企业数据存储的首选方案。

以 MySQL 为例，它作为开源关系型数据库的典范，正越来越广泛地被互联网企业所使用。企业可以根据业务规模的不同的阶段，选择采用不同的系统架构，以应对逐渐增长的访问压力和数据量；并且随着业务的发展，需要提前做好系统的容量规划，在系统的处理能力还未达到极限时，对系统进行扩容，以免带来损失。

1. 业务拆分

业务发展初期为了便于快速迭代，很多应用都采用集中式的架构。随着业务规模的扩展，使系统变得越来越复杂，越来越难以维护，开发效率越来越低，并且系统的资源消耗也越来越大，通过硬件提升性能的成本也越来越高。因此，系统业务的拆分是难以避免的。

举例来说，假设某门户网站，它包含了新闻、用户、帖子、评论等几大块内容，对于数据库来说，它可能包含这样几张表，如 news、users、post、comment，如图 2-5 所示。

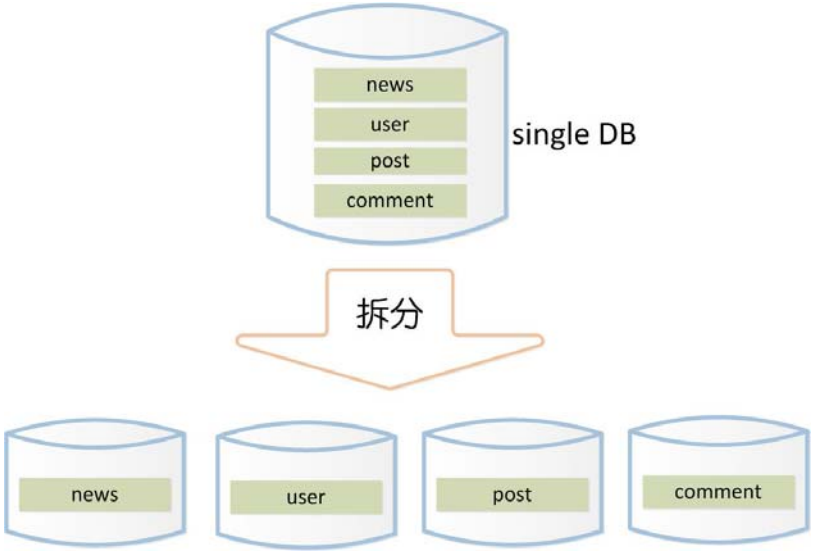


图 2-5　single DB 的拆分

随着业务的不断发展，单个库的访问量越来越大，因此，不得不对业务进行拆分。每一块业务都使用单独的数据库来进行存储，前端不同的业务访问不同的数据库，这样原本依赖单库的服务，变成 4 个库同时承担压力，吞吐能力自然就提高了。

顺带说一句，业务拆分不仅仅提高了系统的可扩展性，也带来了开发工作效率的提升。原来一次简单修改，工程启动和部署可能都需要很长时间，更别说开发测试了。随着系统的拆分，单个系统复杂度降低，减轻了应用多个分支开发带来的分支合并冲突解决的麻烦，不仅大大提高了开发测试的效率，同时也提升了系统的稳定性。

2. 复制策略

架构变化的同时，业务也在不断地发展，可能很快就会发现，随着访问量的不断增加，拆分后的某个库压力越来越大，马上就要达到能力的瓶颈，数据库的架构不得不再次进行变更，这时可以使用 MySQL 的 replication（复制）策略来对系统进行扩展。

通过数据库的复制策略，可以将一台 MySQL 数据库服务器中的数据复制到其他 MySQL 数据库服务器上。当各台数据库服务器上都包含相同数据时，前端应用通过访问 MySQL 集群中任意一台服务器，都能够读取到相同的数据，这样每台 MySQL 服务器所需要承担的负载就会大大降低，从而提高整个系统的承载能力，达到系统扩展的目的。

如图 2-6 所示，要实现数据库的复制，需要开启 Master 服务器端的 Binary log。数据复制的

过程实际上就是 Slave 从 master 获取 binary log，然后再在本地镜像的执行日志中记录的操作。由于复制过程是异步的，因此 Master 和 Slave 之间的数据有可能存在延迟的现象，此时只能够保证数据最终的一致性。

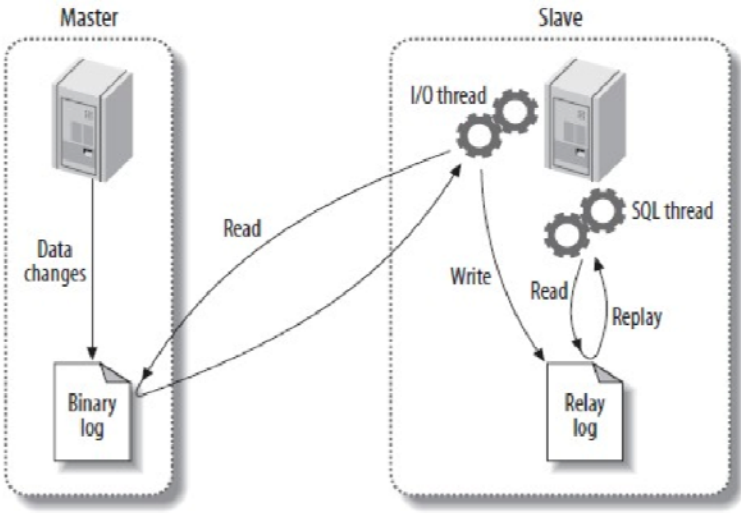


图 2-6　MySQL 的 Master 与 Slave 之间数据同步的过程[11]

MySQL 的复制可以基于一条语句（statement level），也可以基于一条记录（row level）。通过 row level 的复制，可以不记录执行的 SQL 语句相关联的上下文信息，只需要记录数据变更的内容即可。但由于每行的变更都会被记录，这样可能会产生大量的日志内容，而使用 statement level 则只是记录修改数据的 SQL 语句，减少了 binary log 的日志量，节约了 I/O 成本。但是，为了让 SQL 语句在 Slave 端也能够正确地执行，它还需要记录 SQL 执行的上下文信息，以保证所有语句在 Slave 端执行时能够得到在 Master 端执行时的相同结果。

在实际的应用场景中，MySQL 的 Master 与 Slave 之间的复制架构有可能是这样的，如图 2-7 所示。

前端服务器通过 Master 来执行数据写入的操作，数据的更新通过 Binary log 同步到 Slave 集群，而对于数据读取的请求，则交由 Slave 来处理，这样 Slave 集群可以分担数据库读的压力，并且读、写分离还保障了数据能够达到最终一致性。一般而言，大多数站点的读数据库操作要比写数据库操作更为密集。如果读的压力较大，还可以通过新增 Slave 来进行系统的扩展，因此，Master-Slave 的架构能够显著地减轻前面所提到的单库读的压力。毕竟在大多数应用中，读的压力要比写的压力大得多。

11 图片来源 http://hatemysql.com/wp-content/uploads/2013/04/mysql_replication.png。

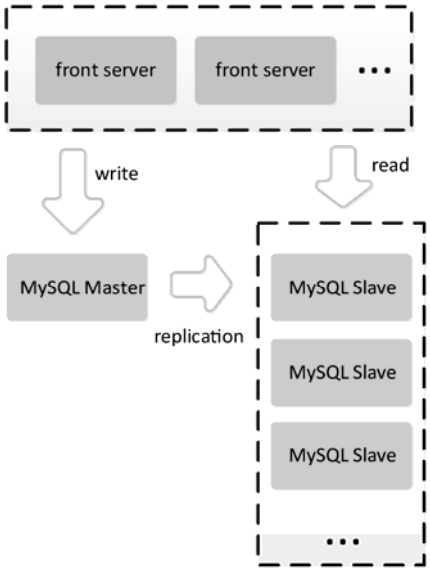


图 2-7　Master-Slaves 复制架构

Master-Slaves 复制架构存在一个问题，即所谓的单点故障。当 Master 宕机时，系统将无法写入，而在某些特定的场景下，也可能需要 Master 停机，以便进行系统维护、优化或者升级。同样的道理，Master 停机将导致整个系统都无法写入，直到 Master 恢复，大部分情况下这显然是难以接受的。为了尽可能地降低系统停止写入的时间，最佳的方式就是采用 Dual-Master 架构，即 Master-Master 架构，如图 2-8 所示。

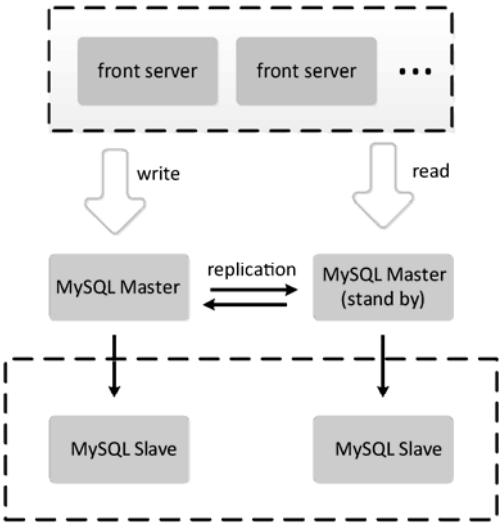


图 2-8 MySQL Dual-Master 架构

所谓的 Dual Master，实际上就是两台 MySQL 服务器互相将对方作为自己的 Master，自己作为对方的 Slave，这样任何一台服务器上的数据变更，都会通过 MySQL 的复制机制同步到另一台服务器。当然，有的读者可能会担心，这样不会导致两台互为 Master 的 MySQL 之间循环复制吗？当然不会，这是由于 MySQL 在记录 Binary log 日志时，记录了当前的 server-id，server-id 在我们配置 MySQL 复制时就已经设置好了。一旦有了 server-id，MySQL 就很容易判断最初的写入是在哪台服务器上发生的，MySQL 不会将复制所产生的变更记录到 Binary log，这样就避免了服务器间数据的循环复制。

当然，我们搭建 Dual-Master 架构，并不是为了让两个 Master 能够同时提供写入服务，这样会导致很多问题。举例来说，假如 Master A 与 Master B 几乎同时对一条数据进行了更新，对 Master A 的更新比对 Master B 的更新早，当对 Master A 的更新最终被同步到 Master B 时，老版本的数据将会把版本更新的数据覆盖，并且不会抛出任何异常，从而导致数据不一致的现象发生。在通常情况下，我们仅开启一台 Master 的写入，另一台 Master 仅仅 stand by 或者作为读库开放，这样可以避免数据写入的冲突，防止数据不一致的情况发生。

在正常情况下，如需进行停机维护，可按如下步骤执行 Master 的切换操作：

（1）停止当前 Master 的所有写入操作。

（2）在 Master 上执行 set global read_only=1，同时更新 MySQL 配置文件中相应的配置，避免重启时失效。

（3）在 Master 上执行 show Master status，以记录 Binary log 坐标。

（4）使用 Master 上的 Binary log 坐标，在 stand by 的 Master 上执行 select Master_pos_wait()，等待 stand by Master 的 Binary log 跟上 Master 的 Binary log。

（5）在 stand by Master 开启写入时，设置 read_only=0。

（6）修改应用程序的配置，使其写入到新的 Master。

假如 Master 意外宕机，处理过程要稍微复杂一点，因为此时 Master 与 stand by Master 上的数据并不一定同步，需要将 Master 上没有同步到 stand by Master 的 Binary log 复制到 Master 上进行 replay，直到 stand by Master 与原 Master 上的 Binary log 同步，才能够开启写入；否则，这一部分不同步的数据就有可能导致数据不一致。

3. 分表与分库

对于大型的互联网应用来说，数据库单表的记录行数可能达到千万级别甚至是亿级，并且数据库面临着极高的并发访问。采用 Master-Slave 复制模式的 MySQL 架构，只能够对数据库的读进行扩展，而对数据的写入操作还是集中在 Master 上，并且单个 Master 挂载的 Slave 也不可

能无限制多，Slave 的数量受到 Master 能力和负载的限制。因此，需要对数据库的吞吐能力进行进一步的扩展，以满足高并发访问与海量数据存储的需要。

对于访问极为频繁且数据量巨大的单表来说，我们首先要做的就是减少单表的记录条数，以便减少数据查询所需要的时间，提高数据库的吞吐，这就是所谓的分表。在分表之前，首先需要选择适当的分表策略，使得数据能够较为均衡地分布到多张表中，并且不影响正常的查询。

对于互联网企业来说，大部分数据都是与用户关联的，因此，用户 id 是最常用的分表字段。因为大部分查询都需要带上用户 id，这样既不影响查询，又能够使数据较为均衡地分布到各个表中[12]，如图 2-9 所示。

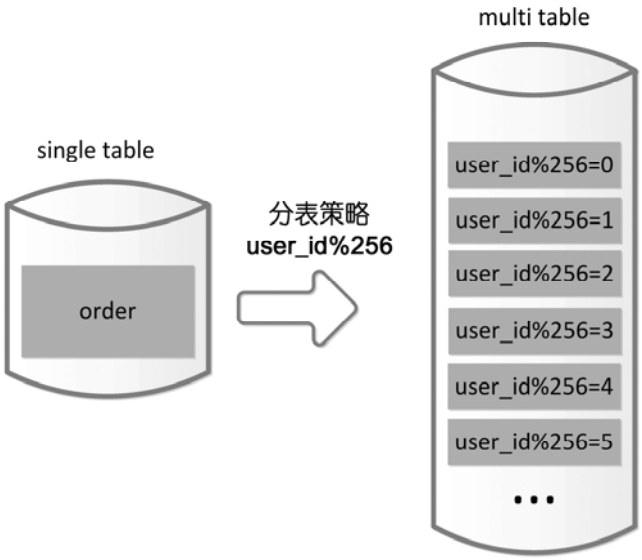


图 2-9　user 表按照 user_id%256 的策略进行分表

假设有一张记录用户购买信息的订单表 order，由于 order 表记录条数太多，将被拆分成 256 张表[13]。拆分的记录根据 user_id%256 取得对应的表进行存储，前台应用则根据对应的 user_id%256，找到对应订单存储的表进行访问。这样一来，user_id 便成为一个必需的查询条件，否则将会由于无法定位数据存储的表而无法对数据进行访问。

假设 user 表的结构如下：

```
create table order(
order_id bigint(20) primary key auto_increment,
```

12 当然，有的场景也可能会出现冷热数据分布不均衡的情况。

13 拆分后表的数量一般为 2 的 *n* 次方。

```
user_id bigint(20),
user_nick varchar(50),
auction_id bigint(20),
auction_title bigint(20),
price bigint(20),
auction_cat varchar(200),
seller_id bigint(20),
seller_nick varchar(50)
);
```

那么分表以后，假设 user_id=257，并且 auction_id=100，需要根据 auction_id 来查询对应的订单信息，则对应的 SQL 语句如下：

```
select * from order_1 where user_id = 257 and auction_id = 100;
```

其中，order_1 根据 257%256 计算得出，表示分表之后的第 1 张 order 表。

分表能够解决单表数据量过大带来的查询效率下降的问题，但是，却无法给数据库的并发处理能力带来质的提升。面对高并发的读写访问，当数据库 Master 服务器无法承载写操作压力时，不管如何扩展 Slave 服务器，此时都没有意义了。因此，我们必须换一种思路，对数据库进行拆分，从而提高数据库写入能力，这就是所谓的分库。

与分表策略相似，分库也可以采用通过一个关键字段取模的方式，来对数据访问进行路由，如图 2-10 所示。

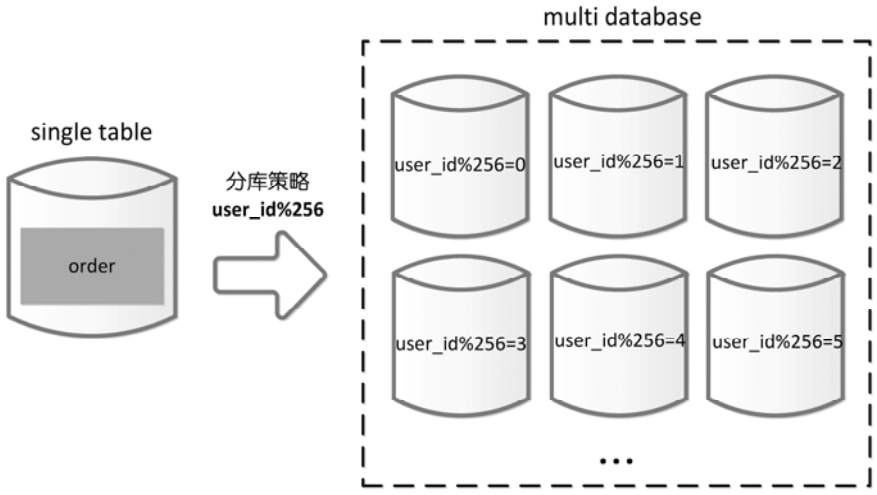


图 2-10　MySQL 分库策略

还是之前的订单表，假设 user_id 字段的值为 257，将原有的单库分为 256 个库，那么应用

程序对数据库的访问请求将被路由到第 1 个库（257%256=1）。

有时数据库可能既面临着高并发访问的压力，又需要面对海量数据的存储问题，这时需要对数据库即采用分库策略，又采用分表策略，以便同时扩展系统的并发处理能力，以及提升单表的查询性能，这就是所谓的分库分表。

分库分表的策略比前面的仅分库或者仅分表的策略要更为复杂，一种分库分表的路由策略如下：

- 中间变量=user_id%（库数量×每个库的表数量）；
- 库=取整（中间变量/每个库的表数量）；
- 表=中间变量%每个库的表数量。

同样采用 user_id 作为路由字段，首先使用 user_id 对库数量×每个库表的数量取模，得到一个中间变量；然后使用中间变量除以每个库表的数量，取整，便得到对应的库；而中间变量对每个库表的数量取模，即得到对应的表。分库分表策略如图 2-11 所示。

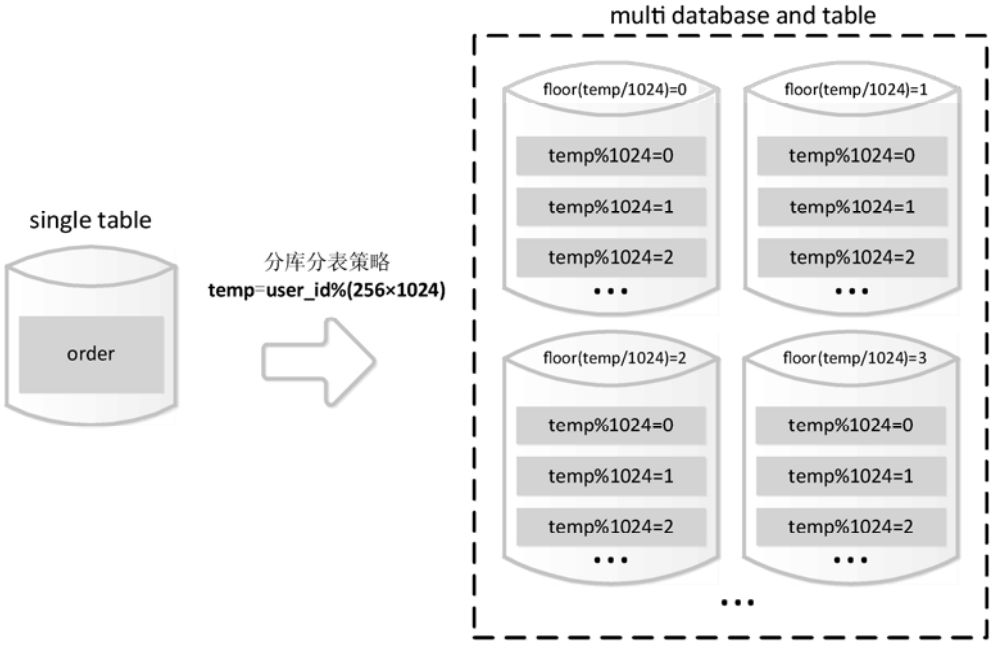


图 2-11　MySQL 分库分表策略

假设将原来的单库单表 order 拆分成 256 个库，每个库包含 1024 个表，那么按照前面所提到的路由策略，对于 user_id=262145 的访问，路由的计算过程如下：

- 中间变量=262145%（256×1024）=1；

- 库=取整（1/1024）=0；
- 表=1%1024=1。

这意味着，对于 user_id=262145 的订单记录的查询和修改，将被路由到第 0 个库的第 1 个表中执行。

数据库经过业务拆分及分库分表之后，虽然查询性能和并发处理能力提高了，但也会带来一系列的问题。比如，原本跨表的事务上升为分布式事务；由于记录被切分到不同的库与不同的表当中，难以进行多表关联查询，并且不能不指定路由字段对数据进行查询。分库分表以后，如果需要对系统进行进一步扩容（路由策略变更），将变得非常不方便，需要重新进行数据迁移。

相较于 MySQL 的分库分表策略，后面要提到的 HBase 天生就能够很好地支持海量数据的存储，能够以更友好、更方便的方式支持表的分区，并且 HBase 还支持多个 Region Server 同时写入，能够较为方便地扩展系统的并发写入能力。而通过后面章节所提到的搜索引擎技术，能够解决采用业务拆分及分库分表策略后，系统无法进行多表关联查询，以及查询时必须带路由字段的问题。搜索引擎能够很好地支持复杂条件的组合查询，通过搜索引擎构建的一张大表，能够弥补一部分数据库拆分所带来的问题。

2.2.2 HBase

HBase[14]是 Apache Hadoop 项目下的一个子项目，它以 Google BigTable[15]为原型，设计实现了高可靠性、高可扩展性、实时读/写的列存储数据库。它的本质实际上是一张稀疏的大表，用来存储粗粒度的结构化数据，并且能够通过简单地增加节点来实现系统的线性扩展。

HBase 运行在分布式文件系统 HDFS[16]之上，利用它可以在廉价的 PC Server 上搭建大规模结构化存储集群。HBase 的数据以表的形式进行组织，每个表由行列组成。与传统的关系型数据库不同的是，HBase 每个列属于一个特定的列族，通过行和列来确定一个存储单元，而每个存储单元又可以有多个版本，通过时间戳来标识，如表 2-1 所示。

表 2-1 HBase 表数据的组织形式

rowkey	column-family1			column-family2		column-family3
	column1	column2	column3	column1	column2	column1
key1	…	…	…	…	…	…
key2	…	…	…	…	…	…

14 HBase 项目地址为 https://hbase.apache.org。

15 著名的 Google BigTable 论文，http://research.google.com/archive/bigtable.html。

16 关于 HDFS 的介绍，请参照第 5.2 节。

key3	…	…	…	…	…	…

HBase 集群中通常包含两种角色，HMaster 和 HRegionServer。当表随着记录条数的增加而不断变大后，将会分裂成一个个 Region，每个 Region 可以由（startkey,endkey）来表示，它包含一个 startkey 到 endkey 的半闭区间。一个 HRegionServer 可以管理多个 Region，并由 HMaster 来负责 HRegionServer 的调度及集群状态的监管。由于 Region 可分散并由不同的 HRegionServer 来管理，因此，理论上再大的表都可以通过集群来处理。HBase 集群布署图如图 2-12 所示。

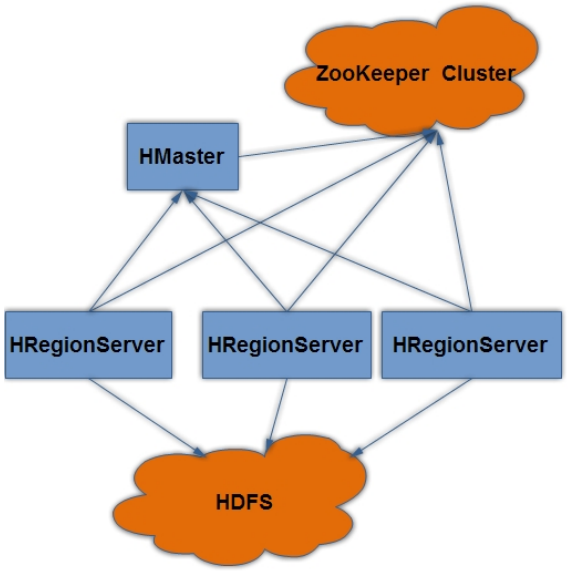


图 2-12　HBase 集群部署图[17]

1. HBase 安装

下载 HBase 的安装包，这里选择的版本是 0.96[18]。

```
wget http://mirror.bit.edu.cn/apache/hbase/hbase-0.96.1.1/hbase-
0.96.1.1-hadoop1-bin.tar.gz
```

17 图片来源 http://dl2.iteye.com/upload/attachment/0073/5412/53da4281-58d4-3f53-8aaf-a09d0c295f05.jpg。

18 HBase 的版本需要与 Hadoop 的版本相兼容，详情请见 http://hbase.apache.org/book/configuration.html# hadoop。

```
longlong@ubuntu:~/temp$ wget http://mirror.bit.edu.cn/apache/hbase/hbase-0.96.1.
1/hbase-0.96.1.1-hadoop1-bin.tar.gz
--2014-04-02 05:41:51--  http://mirror.bit.edu.cn/apache/hbase/hbase-0.96.1.1/hb
ase-0.96.1.1-hadoop1-bin.tar.gz
Resolving mirror.bit.edu.cn (mirror.bit.edu.cn)... 219.143.204.117, 2001:da8:204
:2001:250:56ff:fea1:22
Connecting to mirror.bit.edu.cn (mirror.bit.edu.cn)|219.143.204.117|:80... conne
cted.
HTTP request sent, awaiting response... 200 OK
Length: 73285670 (70M) [application/octet-stream]
Saving to: `hbase-0.96.1.1-hadoop1-bin.tar.gz'

 0% [                                       ] 69,460      8.33K/s  eta 2h 25m
```

解压安装文件：

```
tar -xf hbase-0.96.1.1-hadoop1-bin.tar.gz
```

```
longlong@ubuntu:~/temp$ tar -xf hbase-0.96.1.1-hadoop1-bin.tar.gz
longlong@ubuntu:~/temp$
```

修改配置文件：

编辑{HBASE_HOME}/conf/hbase-env.sh 文件，设置 JAVA_HOME 为 Java 的安装目录。

```
export JAVA_HOME=/usr/java/
```

```
# The java implementation to use.  Java 1.6 required.
export JAVA_HOME=/usr/java/
```

编辑{HBASE_HOME}/conf/hbase-site.xml 文件，增加如下配置，其中 hbase.rootdir 目录用于指定 HBase 的数据存放位置，这里指定的是 HDFS 上的路径，而 hbase.cluster.distributed 则指定了是否运行在分布式模式下。

```
<configuration>
<property>
<name>hbase.cluster.distributed</name>
<value>true</value>
</property>
<property>
<name>hbase.rootdir</name>
<value>hdfs://localhost:9000/hbase</value>
</property>
</configuration>
```

启动 HBase：

完成上述操作后，先启动 Hadoop，再启动 HBase，就可以进行相应的操作了。

```
longlong@ubuntu:/usr/hbase/bin$ ./start-hbase.sh
longlong@localhost's password:
localhost: starting zookeeper, logging to /usr/hbase/bin/../logs/hbase-longlong-
zookeeper-ubuntu.out
starting master, logging to /usr/hbase/bin/../logs/hbase-longlong-master-ubuntu.
out
longlong@localhost's password:
localhost: starting regionserver, logging to /usr/hbase/bin/../logs/hbase-longlo
ng-regionserver-ubuntu.out
```

使用 HBase shell：

```
./hbase shell
```

```
longlong@ubuntu:/usr/hbase/bin$ ./hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.96.1.1-hadoop1, rUnknown, Tue Dec 17 11:52:14 PST 2013

hbase(main):001:0> help
HBase Shell, version 0.96.1.1-hadoop1, rUnknown, Tue Dec 17 11:52:14 PST 2013
Type 'help "COMMAND"', (e.g. 'help "get"' -- the quotes are necessary) for help
on a specific command.
Commands are grouped. Type 'help "COMMAND_GROUP"', (e.g. 'help "general"') for h
elp on a command group.

COMMAND GROUPS:
  Group name: general
  Commands: status, table_help, version, whoami

  Group name: ddl
  Commands: alter, alter_async, alter_status, create, describe, disable, disable
_all, drop, drop_all, enable, enable_all, exists, get_table, is_disabled, is_ena
bled, list, show_filters

  Group name: namespace
  Commands: alter_namespace, create_namespace, describe_namespace, drop_namespac
e, list_namespace, list_namespace_tables
```

查看 HBase 集群状态：

```
status
```

```
hbase(main):002:0> status
1 servers, 0 dead, 2.0000 average load
```

HBase 的基本使用：

创建一个表，并指定列族的名称，create '表名称'、'列族名称 1'、'列族名称 2' ……

例如，create 'user','phone','info'。

```
hbase(main):006:0> create 'user','phone','info'
0 row(s) in 0.5200 seconds

=> Hbase::Table - user
```

创建 user 表，包含两个列族，一个是 phone，一个是 info。

列出已有的表，并查看表的描述：

```
list
```

```
hbase(main):007:0> list
TABLE
user
1 row(s) in 0.1020 seconds

=> ["user"]
```

```
describe ‘表名’
```

例如，describe ‘user’。

```
hbase(main):008:0> describe 'user'
DESCRIPTION                                          ENABLED
 'user', {NAME => 'info', DATA_BLOCK_ENCODING => 'NO true
 NE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0'
 , VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSI
 ONS => '0', TTL => '2147483647', KEEP_DELETED_CELLS
  => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'fa
 lse', BLOCKCACHE => 'true'}, {NAME => 'phone', DATA
 _BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', RE
 PLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSIO
 N => 'NONE', MIN_VERSIONS => '0', TTL => '214748364
 7', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65
 536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
1 row(s) in 0.0600 seconds
```

新增/删除一个列族。

给表新增一个列族：

```
alter '表名',NAME=>'列族名称'
```

例如，alter 'user',NAME=>'class'。

```
hbase(main):010:0> alter 'user',NAME=>'class'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.3880 seconds
```

删除表的一个列族：

```
alter '表名',NAME=>'列族名称',METHOD=>'delete'
```

例如，alter 'user',NAME=>'class',METHOD=>'delete'。

```
hbase(main):012:0> alter 'user',NAME=>'class',METHOD=>'delete'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.3160 seconds
```

删除一个表：

在使用 drop 删除一个表之前，必须先将该表 disable：

```
disable 'user'
drop 'user'
```

```
hbase(main):015:0> disable 'user'
0 row(s) in 1.5500 seconds

hbase(main):016:0> drop 'user'
0 row(s) in 0.2210 seconds
```

如果没有 disable 表而直接使用 drop 删除，则会出现如下提示：

```
hbase(main):014:0> drop 'user'

ERROR: Table user is enabled. Disable it first.'

Here is some help for this command:
Drop the named table. Table must first be disabled: e.g. "hbase> drop 't1'"
```

给表添加记录：

```
put '表名', 'rowkey','列族名称:列名称','值'
```

例如，put 'user','1','info:name','zhangsan'。

```
hbase(main):018:0> put 'user','1','info:name','zhangsan'
0 row(s) in 0.1500 seconds
```

查看数据。

根据 rowkey 查看数据：

```
get '表名称','rowkey'
```

例如，get 'user','1'。

```
hbase(main):019:0> get 'user','1'
COLUMN                CELL
 info:name            timestamp=1396534347948, value=zhangsan
1 row(s) in 0.0330 seconds
```

根据 rowkey 查看对应列的数据：

```
get '表名称','rowkey','列族名称:列名称'
```

例如，get 'user','1','info:name'。

```
hbase(main):028:0> get 'user','1','info:name'
COLUMN                CELL
 info:name            timestamp=1396534347948, value=zhangsan
1 row(s) in 0.0180 seconds
```

查看表中的记录总数：

```
count '表名称'
```

例如，count 'user'。

```
hbase(main):020:0> count 'user'
1 row(s) in 0.0670 seconds

=> 1
```

查看表中所有记录：

```
scan '表名称'
```

例如，scan 'user'。

```
hbase(main):021:0> scan 'user'
ROW                   COLUMN+CELL
 1                    column=info:name, timestamp=1396534347948, value=zhangsan
1 row(s) in 0.0510 seconds
```

查看表中指定列族的所有记录：

```
scan '表名',{COLUMNS => '列族'}
```

例如，scan 'user',{COLUMNS => 'info'}。

```
hbase(main):048:0> scan 'user',{COLUMNS => 'info'}
ROW                   COLUMN+CELL
 1                    column=info:name, timestamp=1396536422440, value=zhangsan
 2                    column=info:name, timestamp=1396536417856, value=zhangsan1
 3                    column=info:name, timestamp=1396535679654, value=zhangsan2
 4                    column=info:name, timestamp=1396536437093, value=zhangsan3
 5                    column=info:name, timestamp=1396536443583, value=zhangsan4
 6                    column=info:name, timestamp=1396536451280, value=zhangsan5
 7                    column=info:name, timestamp=1396536459597, value=zhangsan6
 8                    column=info:name, timestamp=1396536478112, value=zhangsan7
8 row(s) in 0.3570 seconds
```

查看表中指定区间的所有记录：

```
scan '表名称',{COLUMNS => '列族',LIMIT =>记录数, STARTROW => '开始 rowkey',
STOPROW=>'结束 rowkey'}
```

例如，scan 'user',{COLUMNS => 'info',LIMIT =>5, STARTROW => '2',STOPROW=>'7'}。

```
hbase(main):050:0> scan 'user',{COLUMNS => 'info',LIMIT =>5, STARTROW => '2',STOPR
OW=>'7'}
ROW                   COLUMN+CELL
 2                    column=info:name, timestamp=1396536417856, value=zhangsan1
 3                    column=info:name, timestamp=1396535679654, value=zhangsan2
 4                    column=info:name, timestamp=1396536437093, value=zhangsan3
 5                    column=info:name, timestamp=1396536443583, value=zhangsan4
 6                    column=info:name, timestamp=1396536451280, value=zhangsan5
5 row(s) in 0.0240 seconds
```

删除数据。

根据 rowkey 删除列数据：

```
delete '表名称','rowkey' ,'列簇名称'
```

例如，delete 'user','1','info:name'。

```
hbase(main):036:0> delete 'user','1','info:name'
0 row(s) in 0.0140 seconds
```

根据 rowkey 删除一行数据：

```
deleteall  '表名称','rowkey'
```

例如，deleteall 'user','2。

```
hbase(main):037:0> deleteall 'user','2'
0 row(s) in 0.0300 seconds
```

2. HBase API

除了通过 shell 进行操作，HBase 作为分布式数据库，自然也提供程序访问的接口，此处以 Java 为例。

首先，需要配置 HBase 的 HMaster 服务器地址和对应的端口（默认为 60000），以及对应的 ZooKeeper 服务器地址和端口：

```
private static Configuration conf = null;
static {
    conf = HBaseConfiguration.create();
    conf = HBaseConfiguration.create();
    conf.set("hbase.ZooKeeper.property.clientPort", "2181");
    conf.set("hbase.ZooKeeper.quorum", "192.168.136.135");
    conf.set("hbase.master", "192.168.136.135:60000");
}
```

接下来，通过程序来新增 user 表，user 表中有三个列族，分别为 info、class、parent，如果该表已经存在，则先删除该表：

```
public static void createTable() throws Exception {
```

```
    String tableName = "user";
    HBaseAdmin hBaseAdmin = new HBaseAdmin(conf);
    if (hBaseAdmin.tableExists(tableName)) {
        hBaseAdmin.disableTable(tableName);
        hBaseAdmin.deleteTable(tableName);
    }
    HTableDescriptor tableDescriptor = new
            HTableDescriptor(TableName.valueOf(tableName));
    tableDescriptor.addFamily(new HColumnDescriptor("info"));
    tableDescriptor.addFamily(new HColumnDescriptor("class"));
    tableDescriptor.addFamily(new HColumnDescriptor("parent"));
    hBaseAdmin.createTable(tableDescriptor);
    hBaseAdmin.close();
}
```

将数据添加到 user 表，每个列族指定一个列 col，并给该列赋值：

```
public static void putRow() throws Exception {
    String tableName = "user";
    String[] familyNames = {"info","class","parent"};
    HTable table = new HTable(conf, tableName);
    for(int i = 0; i < 20; i ++){
        for (int j = 0; j < familyNames.length; j++) {
            Put put = new Put(Bytes.toBytes(i+""));
            put.add(Bytes.toBytes(familyNames[j]),
                    Bytes.toBytes("col"),
                    Bytes.toBytes("value_"+i+"_"+j));
            table.put(put);
        }
    }
    table.close();
}
```

取得 rowkey 为 1 的行，并将该行打印出来：

```
public static void getRow() throws IOException {
    String tableName = "user";
    String rowKey = "1";
    HTable table = new HTable(conf, tableName);
    Get g = new Get(Bytes.toBytes(rowKey));
```

```
    Result r = table.get(g);
    outputResult(r);
    table.close();
}

public static void outputResult(Result rs){
    List<Cell> list = rs.listCells();
    System.out.println("row key : " +
            new String(rs.getRow()));
    for(Cell cell : list){
        System.out.println("family: " + new String(cell.getFamily())
        + ", col: " + new String(cell.getQualifier())
        + ", value: "  + new String(cell.getValue()) );
    }
}
```

scan 扫描 user 表，并将查询结果打印出来：

```
public static void scanTable() throws Exception {
    String tableName = "user";
    HTable table = new HTable(conf, tableName);
    Scan s = new Scan();
    ResultScanner rs = table.getScanner(s);
    for (Result r : rs) {
        outputResult(r);
    }

    //设置 startrow 和 endrow 进行查询
    s = new Scan("2".getBytes(),"6".getBytes());
    rs = table.getScanner(s);
    for (Result r : rs) {
        outputResult(r);
    }
    table.close();
}
```

删除 rowkey 为 1 的记录：

```
public static void deleteRow( ) throws IOException {
    String tableName = "user";
    String rowKey = "1";
    HTable table = new HTable(conf, tableName);
```

```
        List<Delete> list = new ArrayList<Delete>();
        Delete d = new Delete(rowKey.getBytes());
        list.add(d);
        table.delete(list);
        table.close();
    }
```

3. rowkey 设计

要想访问 HBase 的行，只有三种方式，一种是通过指定 rowkey 进行访问，另一种是指定 rowkey 的 range 进行 scan，再者就是全表扫描。由于全表扫描对于性能的消耗很大，扫描一张上亿行的大表将带来很大的开销，以至于整个集群的吞吐都会受到影响。因此，rowkey 设计的好坏，将在很大程度上影响表的查询性能，是能否充分发挥 HBase 性能的关键。

举例来说，假设使用 HBase 来存储用户的订单信息，我们可能会通过这样几个维度来记录订单的信息，包括购买用户的 id、交易时间、商品 id、商品名称、交易金额、卖家 id 等。假设需要从卖家维度来查看某商品已售出的订单，并且按照下单时间区间来进行查询，那么订单表可以这样设计：

rowkey：`seller_id + auction_id + create_time`

列族：order_info(auction_title,price,user_id)

使用卖家 id+商品 id+交易时间作为表的 rowkey，列族为 order，该列族包含三列，即商品标题、价格、购买者 id，如图 2-13 所示。由于 HBase 的行是按照 rowkey 来排序的，这样通过 rowkey 进行范围查询，可以缩小 scan 的范围。

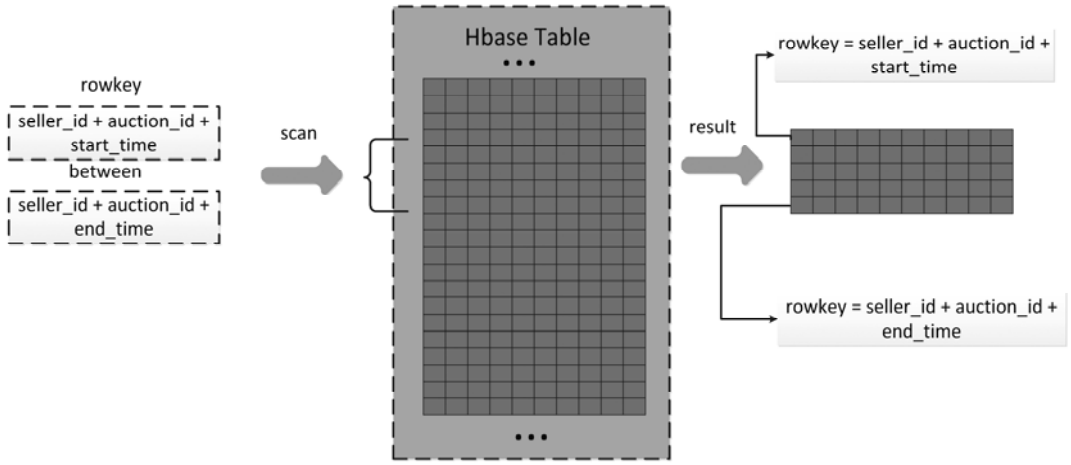


图 2-13　根据 rowkey 进行表的 scan

而假设需要从购买者维度来进行订单数据的查询，展现用户购买过的商品，并且按照购买时间进行查询分页，那么 rowkey 的设计又不同了：

```
rowkey：user_id + create_time
```

列族：order_info(auction_id,auction_title,price,seller_id)

这样通过买家 id+交易时间区间，便能够查到用户在某个时间范围内因购买所产生的订单。

但有些时候，我们既需要从卖家维度来查询商品售出情况，又需要从买家维度来查询商品购买情况，关系型数据库能够很好地支持类似的多条件复杂查询。但对于 HBase 来说，实现起来并不是那么的容易。基本的解决思路就是建立一张二级索引表，将查询条件设计成二级索引表的 rowkey，而存储的数据则是数据表的 rowkey，这样就可以在一定程度上实现多个条件的查询。但是二级索引表也会引入一系列的问题，多表的插入将降低数据写入的性能，并且由于多表之间无事务保障，可能会带来数据一致性的问题[19]。

与传统的关系型数据库相比，HBase 有更好的伸缩能力，更适合于海量数据的存储和处理。由于多个 Region Server 的存在，使得 HBase 能够多个节点同时写入，显著提高了写入性能，并且是可扩展的。但是，HBase 本身能够支持的查询维度有限，难以支持复杂查询，如 group by、order by、join 等，这些特点使得它的应用场景受到了限制。当然，这也并非是不可弥补的硬伤，通过后面章节所介绍的搜索引擎来构建索引，可以在一定程度上解决 HBase 复杂条件组合查询的问题。

2.2.3　Redis

Redis 是一个高性能的 key-value 数据库，与其他很多 key-value 数据库的不同之处在于，Redis 不仅支持简单的键值对类型的存储，还支持其他一系列丰富的数据存储结构，包括 strings、hashs、lists、sets、sorted sets 等，并在这些数据结构类型上定义了一套强大的 API。通过定义不同的存储结构，Redis 可以很轻易地完成很多其他 key-value 数据库难以完成的任务，如排序、去重等。

1. 安装 Redis

下载 Redis 源码安装包：

```
wget http://download.redis.io/releases/redis-2.8.8.tar.gz
```

19 关于 HBase 的二级索引表，华为提供了 hindex 的二级索引解决方案，有兴趣的读者可以参考 https://github.com/Huawei-Hadoop/hindex。

```
longlong@ubuntu:~$ wget http://download.redis.io/releases/redis-2.8.8.tar.gz
--2014-04-06 00:05:22--  http://download.redis.io/releases/redis-2.8.8.tar.gz
Resolving download.redis.io (download.redis.io)... 109.74.203.151
Connecting to download.redis.io (download.redis.io)|109.74.203.151|:80... connec
ted.
HTTP request sent, awaiting response... 200 OK
Length: 1073450 (1.0M) [application/x-gzip]
Saving to: `redis-2.8.8.tar.gz'

19% [======>                              ] 214,588     3.96K/s  eta 4m 2s
```

解压文件：

```
tar -xf redis-2.8.8.tar.gz
```

```
longlong@ubuntu:~/temp$ tar -xf redis-2.8.8.tar.gz
longlong@ubuntu:~/temp$
```

编译安装 Redis：

```
sudo make PREFIX=/usr/local/redis  install
```

```
longlong@ubuntu:~/temp/redis-2.8.8$ sudo make PREFIX=/usr/local/redis install
cd src && make install
make[1]: Entering directory `/home/longlong/temp/redis-2.8.8/src'
rm -rf redis-server redis-sentinel redis-cli redis-benchmark redis-check-dump re
dis-check-aof *.o *.gcda *.gcno *.gcov redis.info lcov-html
(cd ../deps && make distclean)
make[2]: Entering directory `/home/longlong/temp/redis-2.8.8/deps'
(cd hiredis && make clean) > /dev/null || true
(cd linenoise && make clean) > /dev/null || true
(cd lua && make clean) > /dev/null || true
(cd jemalloc && [ -f Makefile ] && make distclean) > /dev/null || true
(rm -f .make-*)
make[2]: Leaving directory `/home/longlong/temp/redis-2.8.8/deps'
(rm -f .make-*)
echo STD=-std=c99 -pedantic >> .make-settings
echo WARN=-Wall >> .make-settings
```

将 Redis 安装到/usr/local/redis 目录，然后，从安装包中找到 Redis 的配置文件，将其复制到安装的根目录。

```
sudo cp redis.conf /usr/local/redis/
```

```
longlong@ubuntu:~/temp/redis-2.8.8$ sudo cp redis.conf /usr/local/redis/
longlong@ubuntu:~/temp/redis-2.8.8$
```

启动 Redis Server：

```
./redis-server ../redis.conf
```

```
longlong@ubuntu:/usr/local/redis/bin$ ./redis-server ../redis.conf
[6535] 06 Apr 00:24:37.800 # You requested maxclients of 10000 requiring at leas
t 10032 max file descriptors.
[6535] 06 Apr 00:24:37.802 # Redis can't set maximum open files to 10032 because
 of OS error: Operation not permitted.
[6535] 06 Apr 00:24:37.802 # Current maximum open files is 1024. maxclients has
been reduced to 4064 to compensate for low ulimit. If you need higher maxclients
 increase 'ulimit -n'.
[6535] 06 Apr 00:24:37.803 # Warning: 32 bit instance detected but no memory lim
it set. Setting 3 GB maxmemory limit with 'noeviction' policy now.

                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 2.8.8 (00000000/0) 32 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in stand alone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 6535
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'
```

使用 redis-cli 进行访问[20]：

```
./redis-cli
```

```
longlong@ubuntu:/usr/local/redis/bin$ ./redis-cli
127.0.0.1:6379> set name chenkangxian
OK
127.0.0.1:6379> get name
"chenkangxian"
127.0.0.1:6379>
```

2. 使用 Redis API

Redis 的 Java client[21]有很多，这里选择比较常用的 Jedis[22]来介绍 Redis 数据访问的 API。

首先，需要对 Redis client 进行初始化：

```
Jedis  redis = new Jedis ("192.168.136.135",6379);
```

Redis 支持丰富的数据类型，如 strings、hashs、lists、sets、sorted sets 等，这些数据类型都有对应的 API 来进行操作。比如，Redis 的 strings 类型实际上就是最基本的 key-value 形式的数据，一个 key 对应一个 value，它支持如下形式的数据访问：

```
redis.set("name", "chenkangxian");//设置 key-value
redis.setex("content", 5, "hello");//设置 key-value 有效期为 5 秒
```

20 更多数据访问的命令请参考 http://redis.io/commands。

21 Redis 的 clien，http://redis.io/clients。

22 Jedis 项目地址为 https://github.com/xetorthio/jedis。

```
redis.mset("class","a","age","25"); //一次设置多个 key-value
redis.append("content", " lucy");//给字符串追加内容
String content = redis.get("content"); //根据 key 获取 value
List<String> list = redis.mget("class","age");//一次取多个 key
```

通过 set 方法，可以给对应的 key 设值；通过 get 方法，可以获取对应 key 的值；通过 setex 方法可以给 key-value 设置有效期；通过 mset 方法，一次可以设置多个 key-value 对；通过 mget 方法，可以一次获取多个 key 对应的 value，这样的好处是，可以避免多次请求带来的网络开销，提高性能；通过 append 方法，可以给已经存在的 key 对应的 value 后追加内容。

Redis 的 hashs 实际上是一个 string 类型的 field 和 value 的映射表，类似于 Map，特别适合存储对象。相较于将每个对象序列化后存储，一个对象使用 hashs 存储将会占用更少的存储空间，并且能够更为方便地存取整个对象：

```
redis.hset("url", "google", "www.google.cn");//给 Hash 添加 key-value
redis.hset("url", "taobao", "www.taobao.com");
redis.hset("url", "sina", "www.sina.com.cn");

Map<String,String> map = new HashMap<String,String>();
map.put("name", "chenkangxian");
map.put("sex", "man");
map.put("age", "100");
redis.hmset("userinfo", map);//批量设置值

String name = redis.hget("userinfo", "name");//取 Hash 中某个 key 的值

//取 Hash 的多个 key 的值
List<String> urllist = redis.hmget("url","google","taobao","sina");

//取 Hash 的所有 key 的值
Map<String,String> userinfo = redis.hgetAll("userinfo");
```

通过 hset 方法，可以给一个 Hash 存储结构添加 key-value 数据；通过 hmset 方法，能够一次性设置多个值，避免多次网络操作的开销；使用 hget 方法，能够取得一个 Hash 结构中某个 key 对应的 value；使用 hmget 方法，则可以一次性获取得多个 key 对应的 value；通过 hgetAll 方法，可以将 Hash 存储对应的所有 key-value 一次性取出。

Redis 的 lists 是一个链表结构，主要的功能是对元素的 push 和 pop，以及获取某个范围内的值等。push 和 pop 操作可以从链表的头部或者尾部插入/删除元素，这使得 lists 既可以作为栈使用，又可以作为队列使用，其中，操作的 key 可以理解为链表的名称：

```
redis.lpush("charlist", "abc");//在 list 首部添加元素
redis.lpush("charlist", "def");
redis.rpush("charlist", "hij");//在 list 尾部添加元素
redis.rpush("charlist", "klm");
List<String> charlist = redis.lrange("charlist", 0, 2);
redis.lpop("charlist");//在 list 首部删除元素
redis.rpop("charlist");//在 list 尾部删除元素
Long charlistSize = redis.llen("charlist");//获得 list 的大小
```

通过 lpush 和 rpush 方法，分别可以在 list 的首部和尾部添加元素；使用 lpop 和 rpop 方法，可以在 list 的首部和尾部删除元素，通过 lrange 方法，可以获取 list 指定区间的元素。

Redis 的 sets 与数据结构的 set 相似，用来存储一个没有重复元素的集合，对集合的元素可以进行添加和删除的操作，并且能够对所有元素进行枚举：

```
redis.sadd("SetMem", "s1");//给 set 添加元素
redis.sadd("SetMem", "s2");
redis.sadd("SetMem", "s3");
redis.sadd("SetMem", "s4");
redis.sadd("SetMem", "s5");
redis.srem("SetMem", "s5");//从 set 中移除元素
Set<String> set = redis.smembers("SetMem");//枚举出 set 的元素
```

sadd 方法用来给 set 添加新的元素，而 srem 则可以对元素进行删除，通过 smembers 方法，能够枚举出 set 中的所有元素。

sorted sets 是 Redis sets 的一个升级版本，它在 sets 的基础之上增加了一个排序的属性，该属性在添加元素时可以指定，sorted sets 将根据该属性来进行排序， 每次新元素增加后，sorted sets 会重新对顺序进行调整。sorted sets 不仅能够通过 range 正序对 set 取值，还能够通过 range 对 set 进行逆序取值，极大地提高了 set 操作的灵活性：

```
redis.zadd("SortSetMem", 1, "5th");//插入 sort set,并指定元素的序号
redis.zadd("SortSetMem", 2, "4th");
redis.zadd("SortSetMem", 3, "3th");
redis.zadd("SortSetMem", 4, "2th");
redis.zadd("SortSetMem", 5, "1th");

//根据范围取 set
Set<String> sortset = redis.zrange("SortSetMem", 2, 4);
```

```
//根据范围反向取 set
Set<String> revsortset = redis.zrevrange("SortSetMem", 1, 2);
```

通过 zadd 方法来给 sorted sets 新增元素，在新增操作的同时，需要指定该元素排序的序号，以便进行排序。使用 zrange 方法可以正序对 set 进行范围取值，而通过 zrevrange 方法，则可以高效率地逆序对 set 进行范围取值。

相较于传统的关系型数据库，Redis 有更好的读/写吞吐能力，能够支撑更高的并发数。而相较于其他的 key-value 类型的数据库，Redis 能够提供更为丰富的数据类型的支持，能够更灵活地满足业务需求。Redis 能够高效率地实现诸如排序取 topN、访问计数器、队列系统、数据排重等业务需求，并且通过将服务器设置为 cache-only，还能够提供高性能的缓存服务。相较于 memcache 来说，在性能差别不大的情况下，它能够支持更为丰富的数据类型。

2.3 消息系统

在分布式系统中，消息系统的应用十分广泛，消息可以作为应用间通信的一种方式。消息被保存在队列中，直到被接收者取出。由于消息发送者不需要同步等待消息接收者的响应，消息的异步接收降低了系统集成的耦合度，提升了分布式系统协作的效率，使得系统能够更快地响应用户，提供更高的吞吐。当系统处于峰值压力时，分布式消息队列还能够作为缓冲，削峰填谷，缓解集群的压力，避免整个系统被压垮。

开源的消息系统有很多，包括 Apache 的 ActiveMQ，Apache 的 Kafka、RabbitMQ、memcacheQ 等，本节将通过 Apache 的 ActiveMQ 来介绍消息系统的使用与集群架构。

2.3.1 ActiveMQ & JMS

ActiveMQ 是 Apache 所提供的一个开源的消息系统，完全采用 Java 来实现，因此，它能够很好地支持 J2EE 提出 JMS 规范。JMS（Java Message Service，即 Java 消息服务）是一组 Java 应用程序接口，它提供消息的创建、发送、接收、读取等一系列服务。JMS 定义了一组公共应用程序接口和相应的语法，类似于 Java 数据库的统一访问接口 JDBC，它是一种与厂商无关的 API，使得 Java 程序能够与不同厂商的消息组件很好地进行通信。

JMS 支持的消息类型包括简单文本（TextMessage）、可序列化的对象（ObjectMessage）、键值对（MapMessage）、字节流（BytesMessage）、流（StreamMessage），以及无有效负载的消息（Message）等。消息的发送是异步的，因此，消息的发布者发送完消息之后，不需要等待消息接收者立即响应，这样便提高了分布式系统协作的效率。

JMS 支持两种消息发送和接收模型。一种称为 Point-to-Point（P2P）模型，即采用点对点的方式发送消息。P2P 模型是基于 queue（队列）的，消息生产者发送消息到队列，消息消费者从队列中接收消息，队列的存在使得消息的异步传输称为可能，P2P 模型在点对点的情况下进行消息传递时采用。另一种称为 Pub/Sub（Publish/Subscribe，即发布/订阅）模型，发布/订阅模型定义了如何向一个内容节点发布和订阅消息，这个内容节点称为 topic（主题）。主题可以认为是消息传递的中介，消息发布者将消息发布到某个主题，而消息订阅者则从主题订阅消息。主题使得消息的订阅者与消息的发布者互相保持独立，不需要进行接触即可保证消息的传递，发布/订阅模型在消息的一对多广播时采用。

如图 2-14 所示，对于点对点消息传输模型来说，多个消息的生产者和消息的消费者都可以注册到同一个消息队列，当消息的生产者发送一条消息之后，只有其中一个消息消费者会接收到消息生产者所发送的消息，而不是所有的消息消费者都会收到该消息。

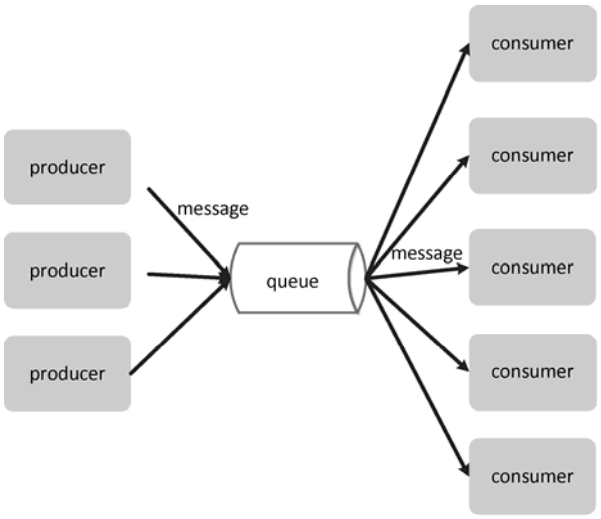


图 2-14　点对点消息传输模型

如图 2-15 所示，对于发布/订阅消息传输模型来说，消息的发布者需将消息投递给 topic，而消息的订阅者则需要在相应的 topic 进行注册，以便接收相应 topic 的消息。与点对点消息传输模型不同的是，消息发布者的消息将被自动发送给所有订阅了该 topic 的消息订阅者。当消息订阅者某段时间由于某种原因断开了与消息发布者的连接时，这个时间段内的消息将会丢失，除非将消息的订阅模式设置为持久订阅（durable subscription），这时消息的发布者将会为消息的订阅者保留这段时间所产生的消息。当消息的订阅者重新连接消息发布者时，消息订阅者仍然可以获得这部分消息，而不至于丢失这部分消息。

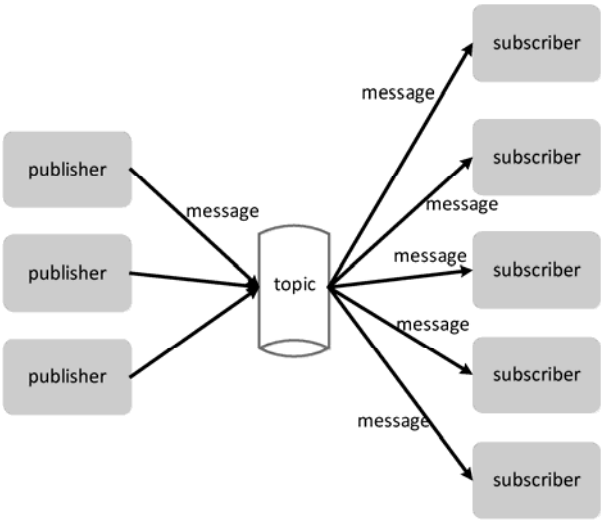


图 2-15　发布/订阅消息传输模型

1. 安装 ActiveMQ

由于 ActiveMQ 是纯 Java 实现的，因此 ActiveMQ 的安装依赖于 Java 环境，关于 Java 环境的安装此处就不详细介绍了，请读者自行查阅相关资料。

下载 ActiveMQ：

```
wget http://apache.dataguru.cn/activemq/apache-activemq/5.9.0/apache-activemq-5.9.0-bin.tar.gz
```

```
longlong@ubuntu:~$ wget http://apache.dataguru.cn/activemq/apache-activemq/5.9.0
/apache-activemq-5.9.0-bin.tar.gz
--2014-04-15 05:34:00--  http://apache.dataguru.cn/activemq/apache-activemq/5.9.
0/apache-activemq-5.9.0-bin.tar.gz
Resolving apache.dataguru.cn (apache.dataguru.cn)...
```

解压安装文件：

```
tar -xf apache-activemq-5.9.0-bin.tar.gz
```

```
longlong@ubuntu:~/temp$ tar -xf apache-activemq-5.9.0-bin.tar.gz
longlong@ubuntu:~/temp$
```

相关的配置放在{ACTIVEMQ_HOME}/conf 目录下，可以对配置文件进行修改：

```
ls /usr/activemq
```

```
longlong@ubuntu:/usr/activemq/conf$ ls
activemq.xml           credentials-enc.properties  jmx.password
broker.ks              credentials.properties      log4j.properties
broker-localhost.cert  groups.properties           logging.properties
broker.ts              jetty-realm.properties      login.config
client.ks              jetty.xml                   users.properties
client.ts              jmx.access
```

启动 ActiveMQ：

```
./activemq  start
```

```
longlong@ubuntu:/usr/activemq/bin$ ./activemq start
INFO: Using default configuration
(you can configure options in one of these file: /etc/default/activemq /home/lon
glong/.activemqrc)

INFO: Invoke the following command to create a configuration file
./activemq setup [ /etc/default/activemq | /home/longlong/.activemqrc ]

INFO: Using java '/usr/java/bin/java'
INFO: Starting - inspect logfiles specified in logging.properties and log4j.prop
erties to get details
INFO: pidfile created : '/usr/activemq/data/activemq-ubuntu.pid' (pid '2953')
```

2. 通过 JMS 访问 ActiveMQ

ActiveMQ 实现了 JMS 规范提供的一系列接口，如创建 Session、建立连接、发送消息等，通过这些接口，能够实现消息发送、消息接收、消息发布、消息订阅的功能。

使用 JMS 来完成 ActiveMQ 基于 queue 的点对点消息发送：

```
ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(
        ActiveMQConnection.DEFAULT_USER,
        ActiveMQConnection.DEFAULT_PASSWORD,
        "tcp://192.168.136.135:61616");
Connection connection = connectionFactory
                        .createConnection();
connection.start();
Session session = connection.createSession
    (Boolean.TRUE,Session.AUTO_ACKNOWLEDGE);
Destination destination = session
                .createQueue("MessageQueue");
MessageProducer producer = session.createProducer(destination);
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

ObjectMessage message = session
    .createObjectMessage("hello everyone!");
producer.send(message);
session.commit();
```

创建一个 ActiveMQConnectionFactory，通过 ActiveMQConnectionFactory 来创建到 ActiveMQ 的连接，通过连接创建 Session。创建 Session 时有两个非常重要的参数，第一个 boolean

类型的参数用来表示是否采用事务消息。如果消息是事务的，对应的该参数设置为 true，此时消息的提交自动由 comit 处理，消息的回滚则自动由 rollback 处理。假如消息不是事务的，则对应的该参数设置为 false，此时分为三种情况，Session.AUTO_ACKNOWLEDGE 表示 Session 会自动确认所接收到的消息；而 Session.CLIENT_ACKNOWLEDGE 则表示由客户端程序通过调用消息的确认方法来确认所收到的消息；Session.DUPS_OK_ACKNOWLEDGE 这个选项使得 Session 将“懒惰”地确认消息，即不会立即确认消息，这样有可能导致消息重复投递。Session 创建好以后，通过 Session 创建一个 queue，queue 的名称为 MessageQueue，消息的发送者将会向这个 queue 发送消息。

基于 queue 的点对点消息接收类似：

```
ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(
        ActiveMQConnection.DEFAULT_USER,
        ActiveMQConnection.DEFAULT_PASSWORD,
        "tcp://192.168.136.135:61616");
Connection connection = connectionFactory
                    .createConnection();
connection.start();
Session session = connection.createSession(Boolean.FALSE,
        Session.AUTO_ACKNOWLEDGE);
Destination destination= session
            .createQueue("MessageQueue");
MessageConsumer consumer = session
                        .createConsumer(destination);

while (true) {
    //取出消息
    ObjectMessage message = (ObjectMessage)consumer.receive(10000);
    if (null != message) {
        String messageContent = (String)message.getObject();
        System.out.println(messageContent);
    } else {
        break;
    }
}
```

创建 ActiveMQConnectionFactory，通过 ActiveMQConnectionFactory 创建连接，通过连接创建 Session，然后创建目的 queue（这里为 MessageQueue），根据目的 queue 创建消息的消费

者，消息消费者通过 receive 方法来接收 Object 消息，然后将消息转换成字符串并打印输出。

还可以通过 JMS 来创建 ActiveMQ 的 topic，并给 topic 发送消息：

```
ConnectionFactory factory = new ActiveMQConnectionFactory(
        ActiveMQConnection.DEFAULT_USER,
        ActiveMQConnection.DEFAULT_PASSWORD,
        "tcp://192.168.136.135:61616");
Connection connection = factory.createConnection();
connection.start();

Session session = connection.createSession(false,
                        Session.AUTO_ACKNOWLEDGE);
Topic topic = session.createTopic("MessageTopic");

MessageProducer producer = session.createProducer(topic);
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

TextMessage message = session.createTextMessage();
message.setText("message_hello_chenkangxian");
producer.send(message);
```

与发送点对点消息一样，首先需要初始化 ActiveMQConnectionFactory，通过 ActiveMQConnectionFactory 创建连接，通过连接创建 Session。然后再通过 Session 创建对应的 topic，这里指定的 topic 为 MessageTopic。创建好 topic 之后，通过 Session 创建对应消息 producer，然后创建一条文本消息，消息内容为 message_hello_chenkangxian，通过 producer 发送。

消息发送到对应的 topic 后，需要将 listener 注册到需要订阅的 topic 上，以便能够接收该 topic 的消息：

```
ConnectionFactory factory = new ActiveMQConnectionFactory(
        ActiveMQConnection.DEFAULT_USER,
        ActiveMQConnection.DEFAULT_PASSWORD,
        "tcp://192.168.136.135:61616");
Connection connection = factory.createConnection();
connection.start();

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic topic = session.createTopic("MessageTopic");

MessageConsumer consumer = session.createConsumer(topic);
```

```
consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        TextMessage tm = (TextMessage) message;
        try {
            System.out.println(tm.getText());
        } catch (JMSException e) {}

    }
});
```

Session 创建好之后，通过 Session 创建对应的 topic，然后通过 topic 来创建消息的消费者，消息的消费者需要在该 topic 上注册一个 listener，以便消息发送到该 topic 之后，消息的消费者能够及时地接收到。

3. ActiveMQ 集群部署

针对分布式环境下对系统高可用的严格要求，以及面临高并发的用户访问，海量的消息发送等场景的挑战，单个 ActiveMQ 实例往往难以满足系统高可用与容量扩展的需求，这时 ActiveMQ 的高可用方案及集群部署就显得十分重要了。

当一个应用被部署到生产环境中，进行容错和避免单点故障是十分重要的，这样可以避免因为单个节点的不可用而导致整个系统的不可用。目前 ActiveMQ 所提供的高可用方案主要是基于 Master-Slave 模式实现的冷备方案，较为常用的包括基于共享文件系统的 Master-Slave 架构和基于共享数据库的 Master-Slave 架构[23]。

如图 2-16 所示，当 Master 启动时，它会获得共享文件系统的排他锁，而其他 Slave 则 stand-by，不对外提供服务，同时等待获取 Master 的排他锁。假如 Master 连接中断或者发生异常，那么它的排他锁则会立即释放，此时便会有另外一个 Slave 能够争夺到 Master 的排他锁，从而成为 Master，对外提供服务。当之前因故障或者连接中断而丢失排他锁的 Master 重新连接到共享文件系统时，排他锁已经被抢占了，它将作为 Slave 等待，直到 Master 再一次发生异常。

23 关于 ActiveMQ 的高可用架构可以参考 http://activemq.apache.org/masterslave.html。

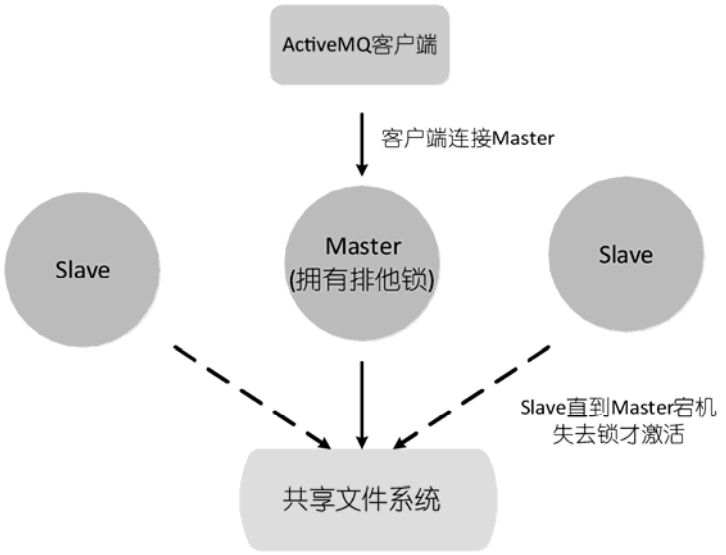


图 2-16　基于共享文件系统的 Master-Slave 架构

基于共享数据库的 Master-Slave 架构同基于共享文件系统的 Master-Slave 架构类似，如图 2-17 所示。当 Master 启动时，会先获取数据库某个表的排他锁，而其他 Slave 则 stand-by，等待表锁，直到 Master 发生异常，连接丢失。这时表锁将释放，其他 Slave 将获得表锁，从而成为 Master 并对外提供服务，Master 与 Slave 自动完成切换，完全不需要人工干预。

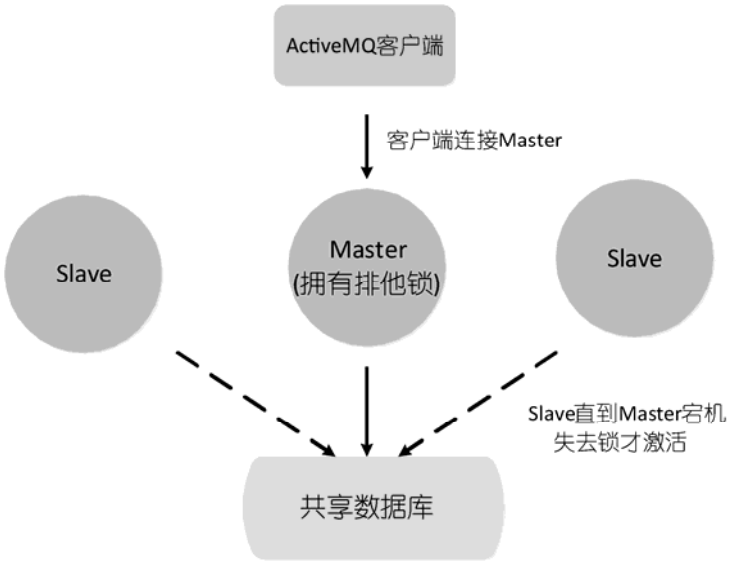


图 2-17　基于共享数据库的 Master-Slave 架构

当然，客户端也需要做一些配置，以便当服务端 Master 与 Slave 切换时，客户无须重启和更改配置就能够进行兼容。在 ActiveMQ 的客户端连接的配置中使用 failover 的方式，可以在 Master 失效的情况下，使客户端自动重新连接到新的 Master：

```
failover:(tcp://master:61616,tcp://slave1:61616,tcp://slave2:61616)
```

假设 Master 失效，客户端能够自动地连接到 Slave1 和 Slave2 两台当中成功获取排他锁的新 Master。

当系统规模不断地发展，产生和消费消息的客户端越来越多，并发的请求数以及发送的消息量不断增加，使得系统逐渐地不堪重负。采用垂直扩展可以提升 ActiveMQ 单 broker 的处理能力。扩展最直接的办法就是提升硬件的性能，如提高 CPU 和内存的能力，这种方式最为简单也最为直接。再者就是就是通过调节 ActiveMQ 本身的一些配置来提升系统并发处理的能力，如使用 nio 替代阻塞 I/O，提高系统处理并发请求的能力，或者调整 JVM 与 ActiveMQ 可用的内存空间等。由于垂直扩展较为简单，此处就不再详细叙述了。

硬件的性能毕竟不能无限制地提升，垂直扩展到一定程度时，必然会遇到瓶颈，这时就需要对系统进行相应的水平扩展。对于 ActiveMQ 来说，可以采用 broker 拆分的方式，将不相关的 queue 和 topic 拆分到多个 broker，来达到提升系统吞吐能力的目的。

假设使用消息系统来处理订单状态的流转，对应的 topic 可能包括订单创建、购买者支付、售卖者发货、购买者确认收货、购买者确认付款、购买者发起退款、售卖者处理退款等，如图 2-18 所示。

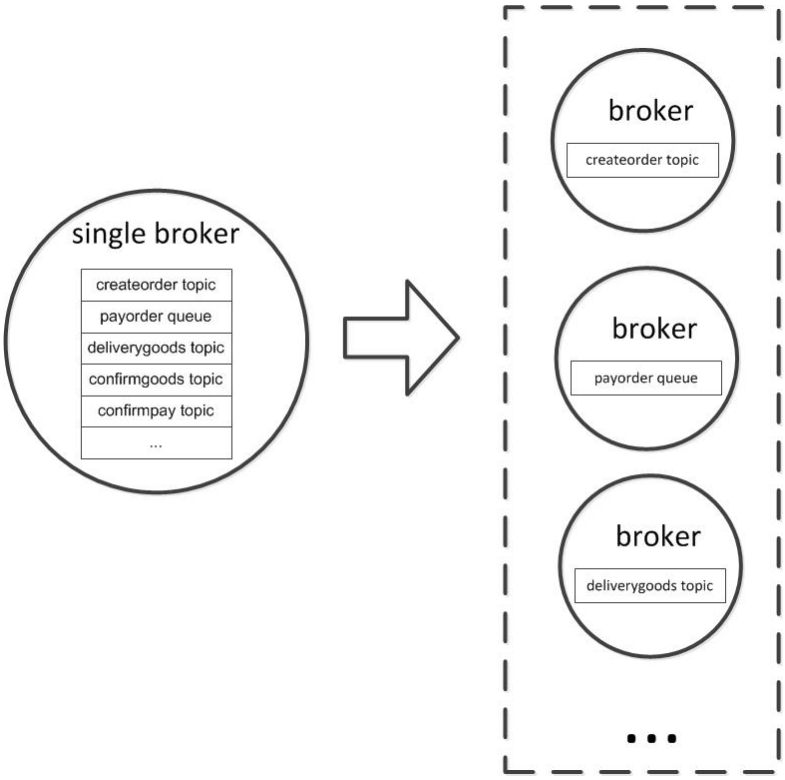


图 2-18　broker 的拆分

原本一个 broker 可以承载多个 queue 或者 topic，现在将不相关的 queue 和 topic 拆出来放到多个 broker 当中，这样可以将一部分消息量大并发请求多的 queue 独立出来单独进行处理，避免了 queue 或者 topic 之间的相互影响，提高了系统的吞吐量，使系统能够支撑更大的并发请求量及处理更多的消息。当然，如有需要，还可以对 queue 和 topic 进行进一步的拆分，类似于数据库的分库分表策略，以提高系统整体的并发处理能力。

2.4　垂直化搜索引擎

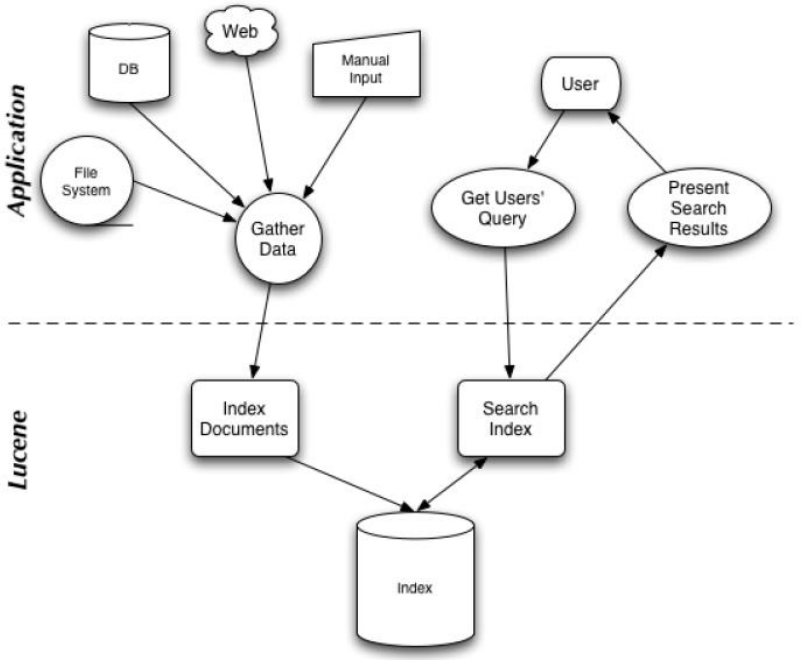
这里所介绍的垂直化搜索引擎，与大家所熟知的 Google 和 Baidu 等互联网搜索引擎存在着一些差别。垂直化的搜索引擎主要针对企业内部的自有数据的检索，而不像 Google 和 Baidu 等搜索引擎平台，采用网络爬虫对全网数据进行抓取，从而建立索引并提供给用户进行检索。在分布式系统中，垂直化的搜索引擎是一个非常重要的角色，它既能满足用户对于全文检索、模糊匹配的需求，解决数据库 like 查询效率低下的问题，又能够解决分布式环境下，由于采用分库分表或者使用 NoSQL 数据库，导致无法进行多表关联或者进行复杂查询的问题。

本节将重点介绍搜索引擎的基本原理和 Apache Lucence 的使用，以及基于 Lucence 的另一个强大的搜索引擎工具 Solr 的一些简单配置。

2.4.1 Lucene 简介

要深入理解垂直化搜索引擎的架构，不得不提到当前全球范围内使用十分广泛的一个开源检索工具——Lucene[24]。Lucene 是 Apache 旗下的一款高性能、可伸缩的开源的信息检索库，最初是由 Doug Cutting[25]开发，并在 SourceForge 的网站上提供下载。从 2001 年 9 月开始，Lucene 作为高质量的开源 Java 产品加入到 Apache 软件基金会，经过多年的不断发展，Lucene 被翻译成 C++、C#、perl、Python 等多种语言，在全球范围内众多知名互联网企业中得到了极为广泛的应用。通过 Lucene，可以十分容易地为应用程序添加文本搜索功能，而不必深入地了解搜索引擎实现的技术细节以及高深的算法，极大地降低了搜索技术推广及使用的门槛。

Lucene 与搜索应用程序之间的关系如图 2-19 所示。



24 Lucene 项目地址为 https://lucene.apache.org。

25 开源领域的重量级人物，创建了多个成功的开源项目，包括 Lucene、Nutch 和 Hadoop。

图 2-19　Lucene 与搜索应用程序之间的关系[26]

在学习使用 Lucene 之前，需要理解搜索引擎的几个重要概念：

倒排索引（inverted index）也称为反向索引，是搜索引擎中最常见的数据结构，几乎所有的搜索引擎都会用到倒排索引。它将文档中的词作为关键字，建立词与文档的映射关系，通过对倒排索引的检索，可以根据词快速获取包含这个词的文档列表，这对于搜索引擎来说至关重要。

分词又称为切词，就是将句子或者段落进行切割，从中提取出包含固定语义的词。对于英语来说，语言的基本单位就是单词，因此分词特别容易，只需要根据空格/符号/段落进行分割，并且排除停止词（stop word），提取词干[27]即可完成。但是对于中文来说，要将一段文字准确地切分成一个个词，就不那么容易了。中文以字为最小单位，多个字连在一起才能构成一个表达具体含义的词。中文会用明显的标点符号来分割句子和段落，唯独词没有一个形式上的分割符，因此，对于支持中文搜索的搜索引擎来说，需要一个合适的中文分词工具，以便建立倒排索引。

停止词（stop word），在英语中包含了 a、the、and 这样使用频率很高的词，如果这些词都被建到索引中进行索引的话，搜索引擎就没有任何意义了，因为几乎所有的文档都会包含这些词。对于中文来说也是如此，中文里面也有一些出现频率很高的词，如“在”、“这”、“了”、“于”等，这些词没有具体含义，区分度低，搜索引擎对这些词进行索引没有任何意义，因此，停止词需要被忽略掉。

排序，当输入一个关键字进行搜索时，可能会命中许多文档，搜索引擎给用户的价值就是快速地找到需要的文档，因此，需要将相关度更大的内容排在前面，以便用户能够更快地筛选出有价值的内容。这时就需要有适当的排序算法。一般来说，命中标题的文档将比命中内容的文档有更高的相关性，命中多次的文档比命中一次的文档有更高的相关性。商业化的搜索引擎的排序规则十分复杂，搜索结果的排序融入了广告、竞价排名等因素，由于涉及的利益广泛，一般属于核心的商业机密。

另外，关于 Lucene 的几个概念也值得关注一下：

文档（Document），在 Lucene 的定义中，文档是一系列域（Field）的组合，而文档的域则代表一系列与文档相关的内容。与数据库表的记录的概念有点类似，一行记录所包含的字段对应的就是文档的域。举例来说，一个文档比如老师的个人信息，可能包括年龄、身高、性别、个人简介等内容。

域（Field），索引的每个文档中都包含一个或者多个不同名称的域，每个域都包含了域的名

26 图片来源 https://www.ibm.com/developerworks/cn/java/j-lo-lucene1/fig001.jpg。

27 提取词干是西方语言特有的处理步骤，比如英文中的单词有单复数的变形，-ing 和-ed 的变形，但是在搜索引擎中，应该当作同一个词。

称和域对应的值，并且域还可以是不同的类型，如字符串、整型、浮点型等。

词（Term），Term 是搜索的基本单元，与 Field 对应，它包括了搜索的域的名称以及搜索的关键词，可以用它来查询指定域中包含特定内容的文档。

查询（Query），最基本的查询可能是一系列 Term 的条件组合，称为 TermQuery，但也有可能是短语查询（PhraseQuery）、前缀查询（PrefixQuery）、范围查询（包括 TermRangeQuery、NumericRangeQuery 等）等。

分词器（Analyzer），文档在被索引之前，需要经过分词器处理，以提取关键的语义单元，建立索引，并剔除无用的信息，如停止词等，以提高查询的准确性。中文分词与西文分词的区别在于，中文对于词的提取更为复杂。常用的中文分词器包括一元分词[28]、二元分词[29]、词库分词[30]等。

如图 2-20 所示，Lucene 索引的构建过程大致分为这样几个步骤，通过指定的数据格式，将 Lucene 的 Document 传递给分词器 Analyzer 进行分词，经过分词器分词之后，通过索引写入工具 IndexWriter 将索引写入到指定的目录。

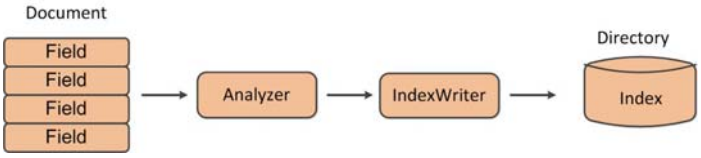


图 2-20　Lucene 索引的构建过程

而对索引的查询，大概可以分为如下几个步骤，如图 2-21 所示。首先构建查询的 Query，通过 IndexSearcher 进行查询，得到命中的 TopDocs。然后通过 TopDocs 的 scoreDocs()方法，拿到 ScoreDoc，通过 ScoreDoc，得到对应的文档编号，IndexSearcher 通过文档编号，使用 IndexReader 对指定目录下的索引内容进行读取，得到命中的文档后

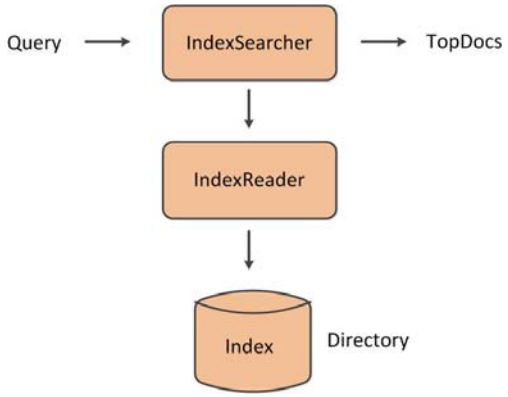


图 2-21　Lucene 索引搜索过程

28 一元分词，即将给定的字符串以一个字为单位进行切割分词，这样分词分我较为精确的缺陷就是语义不准，如“上海”两个字被切割成“上”、“海”，但是包含“上海”、“海上”的文档都会命中。

29 二元分词比一元分词更符合中文的习惯，因为中文的大部分词汇都是两个字，但是问题依然存在。

30 词库分词就是使用词库中定义的词来对字符串进行切分，这样的好处是分词更为准确，但是效率较 *N* 元分词更低，且难以识别互联网世界中层出不穷的新兴词汇。

返回。

2.4.2　Lucene 的使用

Lucene 为搜索引擎提供了强大的、令人惊叹的 API，在企业的垂直化搜索领域得到了极为广泛的应用。为了学习搜索引擎的基本原理，有效地使用 Lucene，并将其引入到我们的应用程序当中，本节将介绍 Lucene 的一些常用的 API 和使用方法，以及索引的优化和分布式扩展。

1. 构建索引

在执行搜索之前，先要构建搜索的索引：

```
Directory dir = FSDirectory.open(new File(indexPath));
Analyzer analyzer = new StandardAnalyzer();
Document doc = new Document();
doc.add(new Field("name","zhansan",Store.YES,Index.ANALYZED));
doc.add(new Field("address","hangzhou",Store.YES,Index.ANALYZED));
doc.add(new Field("sex","man",Store.YES,Index.NOT_ANALYZED));
doc.add(new Field("introduce","i am a coder,my name is zhansan",Store.YES,
Index.NO));
IndexWriter indexWriter = new IndexWriter(dir,analyzer, MaxFieldLength.LIMITED);
indexWriter.addDocument(doc);
indexWriter.close();
```

首先需要构建索引存储的目录 Directory，索引最终将被存放到该目录。然后初始化 Document，给 Document 添加 Field，包括名称、地址、性别和个人介绍信息。Field 的第一个参数为 Field 的名称；第二个参数为 Filed 的值；第三个参数表示该 Field 是否会被存储。Store.NO 表示索引中不存储该 Field；Store.YES 表示索引中存储该 Field；如果是 Store.COMPRESS，则表示压缩存储。最后一个参数表示是否对该字段进行检索。Index.ANALYZED 表示需对该字段进行全文检索，该 Field 需要使用分词器进行分词；Index.NOT_ANALYZED 表示不进行全文检索，因此不需要分词；Index.NO 表示不进行索引。创建一个 IndexWriter，用来写入索引，初始化时需要指定索引存放的目录，以及索引建立时使用的分词器，此处用的是 Lucene 自带的中文分词器 StandardAnalyzer，最后一个参数则用来指定是否限制 Field 的最大长度。

2. 索引更新与删除

很多情况下，在搜索引擎首次构建完索引之后，数据还有可能再次被更改，此时如果不将最新的数据同步到搜索引擎，则有可能检索到过期的数据。遗憾的是，Lucene 暂时还不支持对于 Document 单个 Field 或者整个 Document 的更新，因此这里所说的更新，实际上是删除旧的

Document，然后再向索引中添加新的 Document。所添加的新的 Document 必须包含所有的 Field，包括没有更改的 Field：

```
IndexWriter indexWriter = new IndexWriter(dir,analyzer, MaxFieldLength.LIMITED);
indexWriter.deleteDocuments(new Term("name","zhansan"));
indexWriter.addDocument(doc);
```

IndexWriter 的 deleteDocuments 可以根据 Term 来删除 Document。请注意 Term 匹配的准确性，一个不正确的 Term 可能会导致搜索引擎的大量索引被误删。Lucene 的 IndexWriter 也提供经过封装的 updateDocument 方法，其实质仍然是先删除 Term 所匹配的索引，然后再新增对应的 Document：

```
indexWriter.updateDocument(new Term("name","zhansan"), doc);
```

3. 条件查询

索引构建完之后，就需要对相关的内容进行查询：

```
String queryStr = "zhansan";
String[] fields = {"name","introduce"};

Analyzer analyzer = new StandardAnalyzer();
QueryParser queryPaser = new MultiFieldQueryParser(fields, analyzer);
Query query = queryPaser.parse(queryStr);

IndexSearcher indexSearcher = new IndexSearcher(indexPath);
Filter filter = null;
TopDocs topDocs = indexSearcher.search(query, filter, 10000);

System.out.println("hits :" + topDocs.totalHits );

for(ScoreDoc scoreDoc : topDocs.scoreDocs){
    int docNum = scoreDoc.doc;
    Document doc = indexSearcher.doc(docNum);
    printDocumentInfo(doc);
}
```

查询所使用的字符串为人名 zhansan，查询的 Field 包括 name 和 introduce。构建一个查询 MultiFieldQueryParser 解析器，对查询的内容进行解析，生成 Query；然后通过 IndexSearcher 来对 Query 进行查询，查询将返回 TopDocs，TopDocs 中包含了命中的总条数与命中的 Document

的文档编号；最后通过 IndexSearcher 读取指定文档编号的文档内容，并进行输出。

Lucene 支持多种查询方式，比如针对某个 Field 进行关键字查询：

```
Term term = new Term("name","zhansan");
Query termQuery = new TermQuery(term);
```

Term 中包含了查询的 Field 的名称与需要匹配的文本值，termQuery 将命中名称为 name 的 Field 中包含 zhansan 这个关键字的 Document。

也可以针对某个范围对 Field 的值进行区间查询：

```
NumericRangeQuery numericRangeQuery
    = NumericRangeQuery.newIntRange("size", 2, 100, true, true);
```

假设 Document 包含一个名称为 size 的数值型的 Field，可以针对 size 进行范围查询，指定查询的范围为 2～100，后面两个参数表示是否包含查询的边界值。

还可以通过通配符来对 Field 进行查询：

```
Term wildcardTerm = new Term("name","zhansa?");
WildcardQuery wildcardQuery = new WildcardQuery(wildcardTerm);
```

通配符可以让我们使用不完整、缺少某些字母的项进行查询，但是仍然能够查询到匹配的结果，如指定对 name 的查询内容为“zhansa?”，?表示 0 个或者一个字母，这将命中 name 的值为 zhansan 的 Document，如果使用*，则代表 0 个或者多个字母。

假设某一段落中包含这样一句话“I have a lovely white dog and a black lazy cat”，即使不知道这句话的完整写法，也可以通过 PhraseQuery 查找到包含 dog 和 cat 两个关键字，并且 dog 和 cat 之间的距离不超过 5 个单词的 document：

```
PhraseQuery phraseQuery = new PhraseQuery();
phraseQuery.add(new Term("content","dog"));
phraseQuery.add(new Term("content","cat"));
phraseQuery.setSlop(5);
```

其中，content 为查询对应的 Field，dog 和 cat 分别为查询的短语，而 phraseQuery.setSlop(5) 表示两个短语之间最多不超过 5 个单词，两个 Field 之间所允许的最大距离称为 slop。

除这些之外，Lucene 还支持将不同条件组合起来进行复杂查询：

```
PhraseQuery query1 = new PhraseQuery();
query1.add(new Term("content","dog"));
query1.add(new Term("content","cat"));
query1.setSlop(5);
```

```
Term wildTerm = new Term("name","zhans?");
WildcardQuery query2 = new WildcardQuery(wildTerm);

BooleanQuery booleanQuery = new BooleanQuery();
booleanQuery.add(query1,Occur.MUST);
booleanQuery.add(query2,Occur.MUST);
```

query1 为前面所说的短语查询，而 query2 则为通配符查询，通过 BooleanQuery 将两个查询条件组合起来。需要注意的是，Occur.MUST 表示只有符合该条件的 Document 才会被包含在查询结果中；Occur.SHOULD 表示该条件是可选的；Occur.MUST_NOT 表示只有不符合该条件的 Document 才能够被包含到查询结果中。

4. 结果排序

Lucene 不仅支持多个条件的复杂查询，还支持按照指定的 Field 对查询结果进行排序：

```
String queryStr = "lishi";
String[] fields = {"name","address","size"};
Sort sort = new Sort();
SortField field = new SortField("size",SortField.INT, true);
sort.setSort(field);
Analyzer analyzer = new StandardAnalyzer();
QueryParser queryParse = new MultiFieldQueryParser(fields, analyzer);
Query query = queryParse.parse(queryStr);
IndexSearcher indexSearcher = new IndexSearcher(indexPath);
Filter filter = null;
TopDocs topDocs = indexSearcher.search(query, filter, 100, sort);

for(ScoreDoc scoreDoc : topDocs.scoreDocs){
    int docNum = scoreDoc.doc;
    Document doc = indexSearcher.doc(docNum);
    printDocumentInfo(doc);
}
```

通过新建一个 Sort，指定排序的 Field 为 size，Field 的类型为 SortField.INT，表示按照整数类型进行排序，而不是字符串类型，SortField 的第三个参数用来指定是否对排序结果进行反转。在查询时，使用 IndexSearcher 的一个重构方法，带上 Sort 参数，则能够让查询的结果按照指定的字段进行排序：

```
name : lishi
address : shanghai
sex : man
introduce : i am a dog,my name is coco,and i have a friend,she is a cat
size : 9
name : lishi
address : shanghai
sex : man
introduce : i am a dog,my name is coco,and i have a friend,she is a cat
size : 8
name : lishi
address : shanghai
sex : man
introduce : i am a dog,my name is coco,and i have a friend,she is a cat
size : 7
name : lishi
address : shanghai
sex : man
introduce : i am a dog,my name is coco,and i have a friend,she is a cat
size : 6
```

如果是多个 Field 同时进行查询，可以指定每个 Field 拥有不同的权重，以便匹配时可以按照 Document 的相关度进行排序：

```
String queryStr = "zhansan shanghai";
String[] fields = {"name","address","size"};
Map<String,Float> weights = new HashMap<String, Float>();
weights.put("name", 4f);
weights.put("address", 2f);

Analyzer analyzer = new StandardAnalyzer();
QueryParser queryParse = new MultiFieldQueryParser(fields, analyzer,
weights);

Query query = queryParse.parse(queryStr);
IndexSearcher indexSearcher = new IndexSearcher(indexPath);
Filter filter = null;
TopDocs topDocs = indexSearcher.search(query, filter, 100);

for(ScoreDoc scoreDoc : topDocs.scoreDocs){
    int docNum = scoreDoc.doc;
    Document doc = indexSearcher.doc(docNum);
    printDocumentInfo(doc);
}
```

假设查询串中包含 zhansan 和 shanghai 两个查询串，设置 Field name 的权重为 4，而设置 Field address 的权重为 2，如按照 Field 的权重进行查询排序，那么同时包含 zhansan 和 shanghai 的 Document 将排在最前面，其次是 name 为 zhansan 的 Document，最后是 address 为 shanghai 的 Document：

```
name : zhansan
address : shanghai
sex : man
introduce : i am a dog,my name is coco,and i have a friend,she is a cat
size : 0
name : zhansan
address : hangzhou
sex : man
introduce : i am a dog,my name is coco,and i have a friend,she is a cat
size : 0
name : lishi
address : shanghai
sex : man
introduce : i am a dog,my name is coco,and i have a friend,she is a cat
size : 0
```

5. 高亮

查询到匹配的文档后，需要对匹配的内容进行突出展现，最直接的方式就是对匹配的内容高亮显示。对于搜索 list 来说，由于文档的内容可能比较长，为了控制展示效果，还需要对文档的内容进行摘要，提取相关度最高的内容进行展现，Lucene 都能够很好地满足这些需求：

```
Formatter formatter = new SimpleHTMLFormatter("<font color='red'>","</font>");
Scorer  scorer = new QueryScorer(query);
Highlighter highLight = new Highlighter(formatter, scorer);
Fragmenter fragmenter = new SimpleFragmenter(20);
highLight.setTextFragmenter(fragmenter);
```

通过构建高亮的 Formatter 来指定高亮的 HTML 前缀和 HTML 后缀，这里用的是 font 标签。查询短语在被分词后构建一个 QueryScorer，QueryScorer 中包含需要高亮显示的关键字，Fragmenter 则用来对较长的 Field 内容进行摘要，提取相关度较大的内容，参数 20 表示截取前 20 个字符进行展现。构建一个 Highlighter，用来对 Document 的指定 Field 进行高亮格式化：

```
String hi = highLight.getBestFragment(analyzer, "introduce", doc.get
("introduce"));
```

查询命中相应的 Document 后，通过构建的 Highlighter，对 Document 指定的 Field 进行高亮格式化，并且对相关度最大的一块内容进行摘要，得到摘要内容。假设对 dog 进行搜索，introduce 中如包含有 dog，那么使用 Highlighter 高亮并摘要后的内容如下：

```
introduce : i am a <font color='red'>dog</font>,my name
```

6. 中文分词

Lucene 提供的标准中文分词器 StandardAnalyzer 只能够进行简单的一元分词，一元分词以一个字为单位进行语义切分，这种本来为西文所设计的分词器，用于中文的分词时经常会出现语义不准确的情况。可以通过使用一些其他中文分词器来避免这种情况，常用的中文分词器包括 Lucene 自带的中日韩文分词器 CJKAnalyzer，国内也有一些开源的中文分词器，包括 IK 分词[31]、MM 分词[32]，以及庖丁分词[33]、imdict 分词器[34]等。假设有下面一段文字：

```
String zhContent = "我是一个中国人，我热爱我的国家";
```

分词之后，通过下面一段代码可以将分词的结果打印输出：

```
System.out.println("\n 分词器：" + analyze.getClass());
TokenStream tokenStream = analyze.tokenStream("content", new StringReader(text));
Token token = tokenStream.next();
while(token != null){
    System.out.println(token);
    token = tokenStream.next();
}
```

通过 StandardAnalyzer 分词得到的分词结果如下：

```
Analyzer standarAnalyzer = new StandardAnalyzer(Version.LUCENE_CURRENT);
```

```
分词器：class org.apache.lucene.analysis.standard.StandardAnalyzer
(我,0,1,type=<CJ>)
(是,1,2,type=<CJ>)
(一,2,3,type=<CJ>)
(个,3,4,type=<CJ>)
(中,4,5,type=<CJ>)
(国,5,6,type=<CJ>)
(人,6,7,type=<CJ>)
(我,8,9,type=<CJ>)
(热,9,10,type=<CJ>)
(爱,10,11,type=<CJ>)
(我,11,12,type=<CJ>)
(的,12,13,type=<CJ>)
(国,13,14,type=<CJ>)
(家,14,15,type=<CJ>)
```

由此可以得知，StandardAnalyzer 采用的是一元分词，即字符串以一个字为单位进行切割。

使用 CJKAnalyzer 分词器进行分词，得到的结果如下：

---

31 IK 分词项目地址为 https://code.google.com/p/ik-analyzer。

32 MM 分词项目地址为 https://code.google.com/p/mmseg4j。

33 庖丁分词项目地址为 https://code.google.com/p/paoding。

34 imdict 分词项目地址为 https://code.google.com/p/imdict-chinese-analyzer。

```
Analyzer cjkAnalyzer = new CJKAnalyzer();
```

```
分词器: class org.apache.lucene.analysis.cjk.CJKAnalyzer
(我是,0,2,type=double)
(是一,1,3,type=double)
(一个,2,4,type=double)
(个中,3,5,type=double)
(中国,4,6,type=double)
(国人,5,7,type=double)
(我热,8,10,type=double)
(热爱,9,11,type=double)
(爱我,10,12,type=double)
(我的,11,13,type=double)
(的国,12,14,type=double)
(国家,13,15,type=double)
```

通过分词的结果可以看到，CJKAnalyzer 采用的是二元分词，即字符串以两个字为单位进行切割。

使用开源的 IK 分词的效果如下：

```
Analyzer ikAnalyzer = new IKAnalyzer()
```

```
分词器: class org.wltea.analyzer.lucene.IKAnalyzer
(我,0,1)
(是,1,2)
(一个中国,2,6)
(一个,2,4)
(一,2,3)
(个中,3,5)
(个,3,4)
(中国人,4,7)
(中国,4,6)
(国人,5,7)
(我,8,9)
(热爱,9,11)
(爱我,10,12)
(的,12,13)
(国家,13,15)
```

可以看到，分词的效果比单纯的一元或者二元分词要好很多。

使用 MM 分词器分词的效果如下：

```
Analyzer mmAnalyzer = new MMAnalyzer()
```

```
分词器: class jeasy.analysis.MMAnalyzer
(我是,0,2)
(一个中,2,5)
(国人,5,7)
(我,8,9)
(热爱,9,11)
(我的,11,13)
(国家,13,15)
```

7. 索引优化

Lucene 的索引是由段（segment）组成的，每个段可能又包含多个索引文件，即每个段包含了一个或者多个 Document；段结构使得 Lucene 可以很好地支持增量索引，新增的 Document 将被添加到新的索引段当中。但是，当越来越多的段被添加到索引当中时，索引文件也就越来越多。一般来说，操作系统对于进程打开的文件句柄数是有限的，当一个进程打开太多的文件时，会抛出 too many open files 异常，并且执行搜索任务时，Lucene 必须分别搜索每个段，然后将各个段的搜索结果合并，这样查询的性能就会降低。

为了提高 Lucene 索引的查询性能，当索引段的数量达到设置的上限时，Lucene 会自动进行索引段的优化，将索引段合并成为一个，以提高查询的性能，并减少进程打开的文件句柄数量。但是，索引段的合并需要大量的 I/O 操作，并且需要耗费相当的时间。虽然这样的工作做完以后，可以提高搜索引擎查询的性能，但在索引合并的过程中，查询的性能将受到很大影响，这对于前台应用来说一般是难以接受的。

因此，为了提高搜索引擎的查询性能，需要尽可能地减少索引段的数量，另外，对于需要应对前端高并发查询的应用来说，对索引的自动合并行为也需要进行抑制，以提高查询的性能。

一般来说，在分布式环境下，会安排专门的集群来生成索引，并且生成索引的集群不负责处理前台的查询请求。当索引生成以后，通过索引优化，对索引的段进行合并。合并完以后，将生成好的索引文件分发到提供查询服务的机器供前台应用查询。当然，数据会不断地更新，索引文件如何应对增量的数据更新也是一个挑战。对于少量索引来说，可以定时进行全量的索引重建，并且将索引推送到集群的其他机器，前提是相关业务系统能够容忍数据有一定延迟。但是，当数据量过于庞大时，索引的构建需要很长的时间，延迟的时间可能无法忍受，因此，我们不得不接受索引有一定的瑕疵，即索引同时包含多个索引段，增量的更新请求将不断地发送给查询机器。查询机器可以将索引加载到内存，并以固定的频率回写磁盘，每隔一定的周期，对索引进行一次全量的重建操作，以将增量更新所生成的索引段进行合并。

8. 分布式扩展

与其他的分布式系统架构类似，基于 Lucene 的搜索引擎也会面临扩展的问题，单台机器难以承受访问量不断上升的压力，不得不对其进行扩展。但是，与其他应用不同的是，搜索应用大部分场景都能够接受一定时间的数据延迟，对于数据一致性的要求并不那么高，大部分情况下只要能够保障数据的最终一致性，可以容忍一定时间上的数据不同步，一种扩展的方式如图 2-22 所示。

每个 query server 实例保存一份完整的索引，该索引由 dump server 周期性地生成，并进行索引段的合并，索引生成好之后推送到每台 query server 进行替换，这样避免集群索引 dump 对后端数据存储造成压力。当然，对于增量的索引数据更新，dump server 可以异步地将更新推送到每台 query server，或者是 query server 周期性地到 dump server 进行数据同步，以保证数据最

终的一致性。对于前端的 client 应用来说，通过对请求进行 Hash，将请求均衡地分发到集群中的每台服务器，使得压力能够较为均衡地分布，这样即达到了系统扩展的目的。

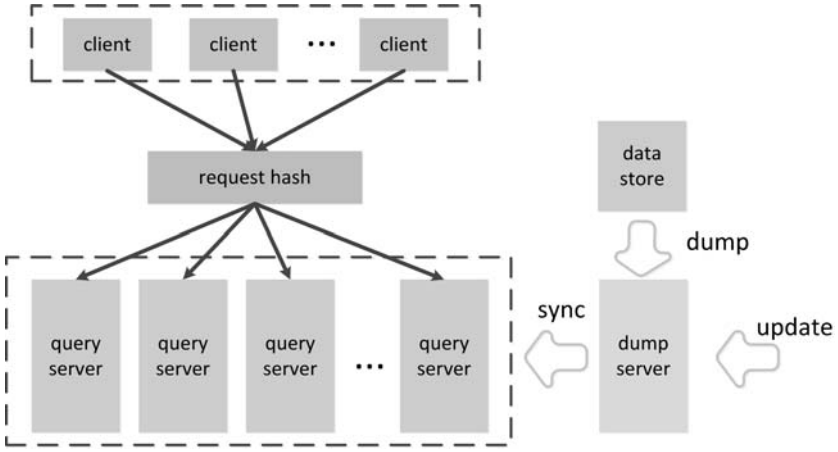


图 2-22 搜索引擎索引的读写分离

索引的读/写分离解决的是请求分布的问题，而对于数据量庞大的搜索引擎来说，单机对索引的存储能力毕竟有限。而且随着索引数量的增加，检索的速度也会随之下降。此时索引本身已经成为系统的瓶颈，需要对索引进行切分，将索引分布到集群的各台机器上，以提高查询性能，降低存储压力，如图 2-23 所示。

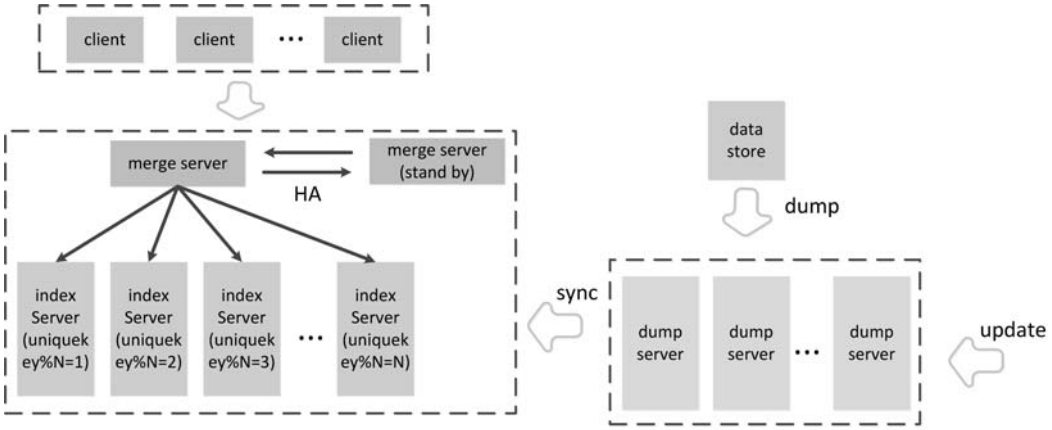


图 2-23 索引的切分

在如图 2-24 所示的架构中，索引依据 uniquekey%N，被切分到多台 index server 中进行存储。client 应用的查询请求提交到 merge server，merge server 将请求分发到 index server 进行检索，最后将查询的结果进行合并后，返回给 client 应用。对于全量的索引构建，可以使用 dump

server 集群，以加快索引构建的速度，并分担存储的压力。而增量的更新请求，可以根据索引的 uniquekey 取模，将索引同步到 index server；为避免 merge server 出现单点，可以对 merge server 进行高可用部署。当然，索引切分的方案并非完美，可能也会带来一些问题。举例来说，假如查询请求需要进行结果排序，当索引没有切分时很好处理，只需要按照查询指定的条件排列即可，但是对切分后的索引来说，排序请求将被分发到每一台 index server 执行排序，排完以后取 topN（出于性能考虑）发送到 merge server 进行合并，合并后的结果与真正的结果很可能存在偏差，这就需要在业务上进行取舍。

有的时候，可能既面临高并发的用户访问请求，又需要对海量的数据集进行索引，这时就需要综合上述的两种方法，即既采用索引读写分离的方式，以支撑更大的并发访问量，又采用索引切分的方式，以解决数据量膨胀所导致的存储压力以及索引性能下降的问题，如图 2-24 所示。

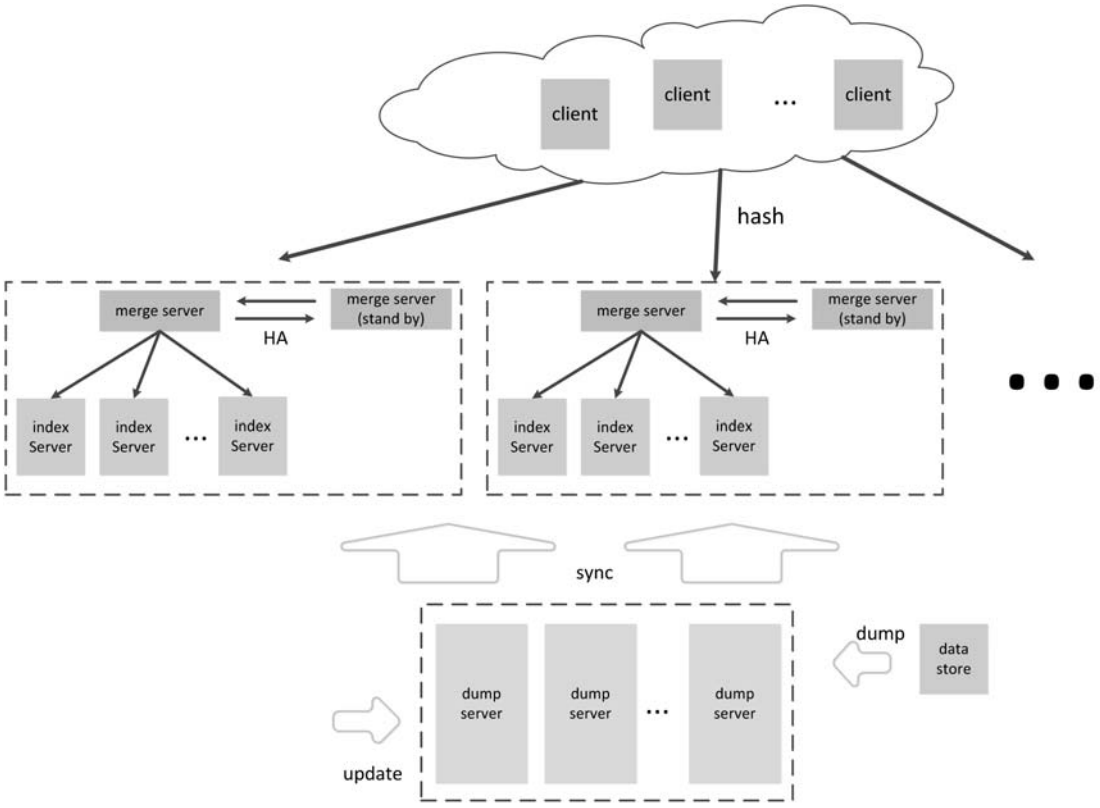


图 2-24　既进行读写分离，又进行索引切分

merge server 与 index server 作为一组基本单元进行复制，而前端应用的请求通过 Hash 被分发到不同的组进行处理；每一组与之前类似，使用 merge server 将请求分发到 index server 进行索引的查询；查询的结果将在 merge server 进行合并，合并完以后，再将结果返回给 client。

2.4.3 Solr

Solr 是一个基于 Lucene、功能强大的搜索引擎工具，它对 Lucene 进行了扩展，提供一系列功能强大的 HTTP 操作接口，支持通过 Data Schema 来定义字段、类型和设置文本分析，使得用户可以通过 HTTP POST 请求，向服务器提交 Document，生成索引，以及进行索引的更新和删除操作。对于复杂的查询条件，Solr 提供了一整套表达式查询语言，能够更方便地实现包括字段匹配、模糊查询、分组统计等功能；同时，Solr 还提供了强大的可配置能力，以及功能完善的后台管理系统。Solr 的架构如图 2-25 所示。

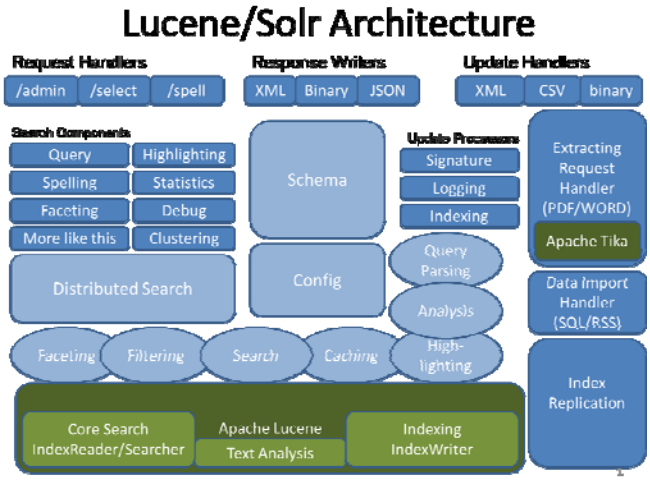


图 2-25 Solr 的架构[35]

1. Solr 的配置

通过 Solr 的官方站点下载 Solr：

```
wget http://apache.fayea.com/apache-mirror/lucene/solr/4.7.2/solr-4.7.2.tgz
```

```
longlong@ubuntu:~/temp$ wget http://apache.fayea.com/apache-mirror/lucene/solr/4
.7.2/solr-4.7.2.tgz
--2014-04-26 22:11:13--  http://apache.fayea.com/apache-mirror/lucene/solr/4.7.2
/solr-4.7.2.tgz
Resolving apache.fayea.com (apache.fayea.com)... 220.166.52.227, 220.166.52.226
Connecting to apache.fayea.com (apache.fayea.com)|220.166.52.227|:80... connecte
d.
HTTP request sent, awaiting response... 200 OK
Length: 151973754 (145M) [application/x-gzip]
Saving to: `solr-4.7.2.tgz'

 0% [                                      ] 1,159,198    451K/s
```

35 图片来源 http://images.cnitblog.com/blog/483523/201308/20142655-8e3153496cf244a280c5e195232ba962.x-png。

解压：

```
tar -xf solr-4.7.2.tgz
```

修改 Tomcat 的 conf/server.xml 中的 Connector 配置，将 URIEncoding 编码设置为 UTF-8，否则中文将会乱码，从而导致搜索查询不到结果。

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" URIEncoding="UTF-8"/>
```

将 Solr 的 dist 目录下的 solr-{version}.war 包复制到 tomcat 的 webapps 目录下，并且重命名为 solr.war。

配置 Solr 的 home 目录，包括 schema 文件、solrconfig 文件及索引文件，如果是第一次配置 Solr，可以直接复制 example 目录下的 Solr 目录作为 Solr 的 home，并通过修改 tomcat 的启动脚本 catalina.sh 来指定 solr.solr.home 变量所代表的 Solr home 路径。

```
CATALINA_OPTS="$CATALINA_OPTS -Dsolr.solr.home=/usr/solr"
```

启动 Tomcat，访问 Solr 的管理页面，如图 2-26 所示。

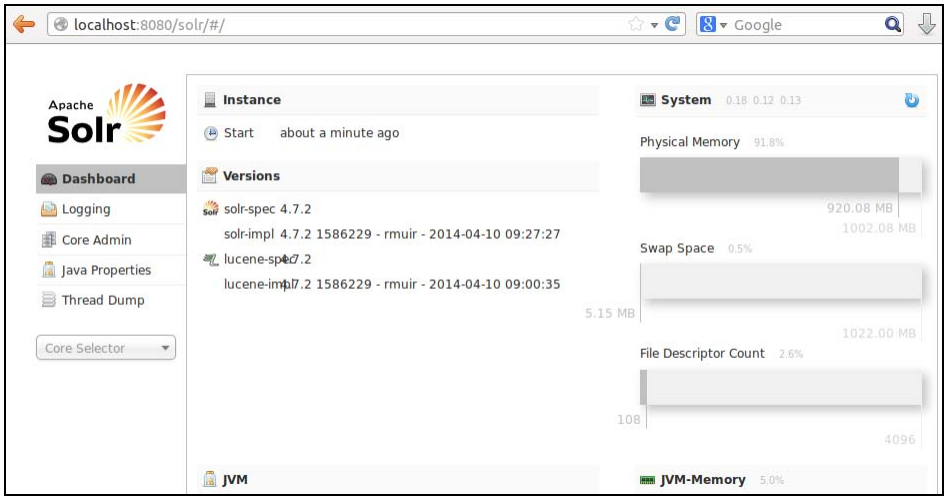


图 2-26　Solr 的管理页面

2. 构建索引

在构建索引之前，首先需要定义好 Document 的 schema。同数据库建表有点类似，即每个 Document 包含哪些 Field，对应的 Field 的 name 是什么，Field 是什么类型，是否被索引，是否被存储，等等。假设我们要构建一个讨论社区，需要对社区内的帖子进行搜索，那么搜索引擎的 Document 中应该包含帖子信息、版块信息、版主信息、发帖人信息、回复总数等内容的聚合，如图 2-27 所示。

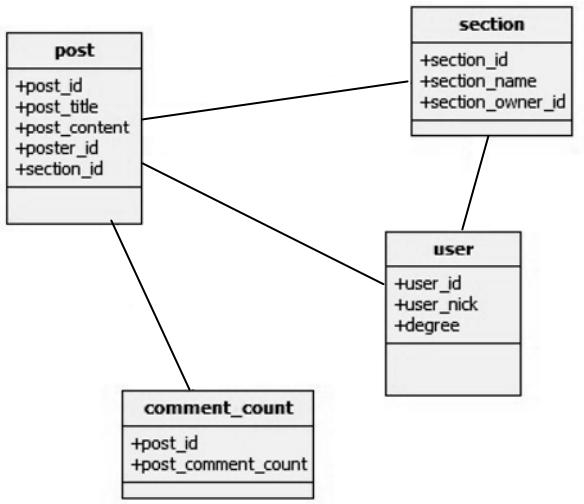


图 2-27　帖子、版块、用户、评论总数的关联关系

其中，post 用来描述用户发布的帖子信息，section 则表示版块信息，user 代表该社区的用户，comment_count 用来记录帖子的评价总数。

对帖子信息建立搜索引擎的好处在于，由于帖子的数据量大，如采用 MySQL 这一类的关系型数据库来进行存储的话，需要进行分库分表。数据经过拆分之后，就难以同时满足多维度复杂条件查询的需求，并且查询可能需要版块、帖子、用户等多个表进行关联查询，导致查询性能下降，甚至回帖总数这样的数据有可能根本就没有存储在关系型数据库当中，而通过搜索引擎，这些需求都能够很好地得到满足。

搜索引擎对应的 schema 文件定义可能是下面这个样子：

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="post" version="1.5">

 <fields>
    <field name="_version_" type="long" indexed="true" stored="true"/>
  <field name="post_id" type="long" indexed="true" stored="true" required=
"true"/>
   <field name="post_title" type="string" indexed="true" stored="true"/>
   <field name="poster_id" type="long" indexed="true" stored="true" />
   <field name="poster_nick" type="string" indexed="true" stored="true"/>
   <field name="post_content" type="text_general" indexed="true" stored=
"true"/>
   <field name="poster_degree" type="int" indexed="true" stored="true"/>
```

```
      <field name="section_id" type="long" indexed="true" stored="true" />
      <field name="section_name" type="string" indexed="true" stored="true" />
      <field name="section_owner_id"  type="long" indexed="true" stored="true"/>
      <field name="section_owner_nick" type="string" indexed="true" stored="true"/>
      <field name="gmt_modified" type="date" indexed="true" stored="true"/>
      <field name="gmt_create" type="date" indexed="true" stored="true"/>
      <field name="comment_count" type="int" indexed="true" stored="true"/>
      <field  name="text"  type="text_general"  indexed="true"  stored="false"
multiValued="true"/>
    </fields>

    <uniqueKey>post_id</uniqueKey>

    <copyField source="post_content" dest="text"/>
    <copyField source="post_content" dest="text"/>
    <copyField source="section_name" dest="text"/>

    <types>

      <fieldType name="string" class="solr.StrField" sortMissingLast="true" />
      <fieldType  name="int"  class="solr.TrieIntField"  precisionStep="0"
positionIncrementGap="0"/>
      <fieldType  name="long"  class="solr.TrieLongField"  precisionStep="0"
positionIncrementGap="0"/>
      <fieldType  name="date"  class="solr.TrieDateField"  precisionStep="0"
positionIncrementGap="0"/>
      <fieldType name="text_general" class="solr.TextField" positionIncrementGap=
"100">
        <analyzer type="index">
          <tokenizer class="solr.StandardTokenizerFactory"/>
        </analyzer>
        <analyzer type="query">
          <tokenizer class="solr.StandardTokenizerFactory"/>
        </analyzer>
      </fieldType>

    </types>
   </schema>
```

fields 标签中所包含的就是定义的这些字段，包括对应的字段名称、字段类型、是否索引、是否存储、是否多值等；uniqueKey 指定了 Document 的唯一键约束；types 标签中则定义了可能用到的数据类型。

使用 HTTP POST 请求可以给搜索引擎添加或者更新已存在的索引：

```
http://hostname:8080/solr/core/update?wt=json
```

POST 的 JSON 内容：

```
{
    "add": {
        "doc": {
            "post_id": "123456",
            "post_title": "Nginx 1.6 稳定版发布，顶级网站用量超越 Apache",
            "poster_id": "340032",
            "poster_nick": "hello123",
            "post_content": "据 W3Techs 统计数据显示，全球 Alexa 排名前 100 万的网站中的 23.3%都在使用 nginx，在排名前 10 万的网站中，这一数据为 30.7%，而在前 1000 名的网站中，nginx 的使用量超过了 Apache，位居第 1 位。",
            "poster_degree": "2",
            "section_id": "422",
            "section_name": "技术",
            "section_owner_id": "232133333",
            "section_owner_nick": "chenkangxian",
            "gmt_modified": "2013-05-07T12:09:12Z",
            "gmt_create": "2013-05-07T12:09:12Z",
            "comment_count": "3"
        },
        "boost": 1,
        "overwrite": true,
        "commitWithin": 1000
    }
}
```

服务端的响应：

```
{
    "responseHeader": {
        "status": 0,
        "QTime": 14
    }
}
```

通过上述的 HTTP POST 请求，便可将 Document 添加到搜索引擎中。

3. 条件查询

比 Lucene 更进一步的是，Solr 支持将复杂条件组装成 HTTP 请求的参数表达式，使得用户能够快速构建复杂多样的查询条件，包括条件查询、过滤查询、仅返回指定字段、分页、排序、高亮、统计等，并且支持 XML、JSON 等格式的输出。举例来说，假如需要根据 post_id（帖子 id）来查询对应的帖子，可以使用下面的查询请求：

```
http://hostname:8080/solr/core/select?q=post_id:123458&wt=json&indent=true
```

返回的 Document 格式如下：

```
{
    "responseHeader": {
        "status": 0,
        "QTime": 0,
        "params": {
            "indent": "true",
            "q": "post_id:123458",
            "wt": "json"
        }
    },
    "response": {
        "numFound": 1,
        "start": 0,
        "docs": [
            {
                "post_id": 123458,
                "post_title": "美军研发光学雷达卫星 可拍三维高分辨率照片",
                "poster_id": 340032,
                "poster_nick": "hello123",
                "post_content": "继广域动态图像、全动态视频和超光谱技术之后，Lidar
技术也受到关注和投资。这是由于上述技术的能力已经在伊拉克和阿富汗得到试验和验证。",
                "poster_degree": 2,
                "section_id": 422,
                "section_name": "技术 1",
                "section_owner_id": 232133333,
                "section_owner_nick": "chenkangxian",
                "gmt_modified": "2013-05-07T12:09:12Z",
                "gmt_create": "2013-05-07T12:09:12Z",
```

```
                "comment_count": 3,
                "_version_": 1467083075564339200
            }
        ]
    }
}
```

假设页面需要根据 poster_id（发帖人 id）和 section_owner_nick（版主昵称）作为条件来进行查询，并且根据 uniqueKey 降序排列，以及根据 section_id（版块 id）进行分组统计，那么查询的条件表达式可以这样写：

```
http://hostname:8080/solr/core/select?q=poster_id:340032+and+section_owner_nick:chenkangxian&sort=post_id+asc&facet=true&facet.field=section_id&wt=json&indent=true
```

其中 q= poster_id:340032+and+section_owner_nick:chenkangxian 表示查询的 post_id 为 340032，section_owner_nick 为 chenkangxian，两个条件使用 and 组合，而 sort=post_id+asc 则表示按照 post_id 进行升序排列，facet=true&facet.field=section_id 表示使用分组统计，并且分组统计字段为 section_id。

当然，Solr 还支持更多复杂的条件查询，此处就不再详细介绍了[36]。

2.5 其他基础设施

除了前面所提到的分布式缓存、持久化存储、分布式消息系统、搜索引擎，大型的分布式系统的背后，还依赖于其他支撑系统，包括后面章节所要介绍的实时计算、离线计算、分布式文件系统、日志收集系统、监控系统、数据仓库等，以及本书没有详细介绍的 CDN 系统、负载均衡系统、消息推送系统、自动化运维系统等[37]。

36 更详细的查询语法介绍请参考 Solr 官方 wiki，http://wiki.apache.org/solr/CommonQueryParameters#head-6522ef80f22d0e50d2f12ec487758577506d6002。

37 这些系统虽然本书虽没进行详细的介绍，但并不代表它们不重要，它们也是分布式系统的重要组成部分，限于篇幅，此处仅一笔带过，读者可自行查阅相关资料。