[Redis笔记 3](#_Toc15183)

[后Web2.0时代的发展要求遇到的问题 3](#_Toc26996)

[数据库的主要特性 3](#_Toc24902)

[NoSQL无与伦比的特点 3](#_Toc6719)

[Redis特点 4](#_Toc22644)

[Redis中Key(键)常见使用 4](#_Toc17188)

[Keys pattern 4](#_Toc12570)

[Del key 4](#_Toc17244)

[Exists key 4](#_Toc1432)

[Expire key seconds/Expire key timestamp 4](#_Toc27534)

[Persist key 5](#_Toc11423)

[TTL key 5](#_Toc26049)

[Move key db 5](#_Toc1482)

[Rename key Newkey 5](#_Toc16912)

[Type key 5](#_Toc31757)

[Redis中常用数据类型 5](#_Toc30167)

[String 5](#_Toc29860)

[SET key value 5](#_Toc18801)

[GET key 6](#_Toc22111)

[STRLEN KEY 6](#_Toc20674)

[INCR KEY / INCRBY Key Increment 6](#_Toc29832)

[DECR Key/DECRBY KEY Decrement 6](#_Toc11544)

[APPEND Key value 6](#_Toc16817)

[Hash哈希 6](#_Toc2870)

[HSET key field value 7](#_Toc4649)

[HMSET key field1 value1 [field2 value2 ]  7](#_Toc9852)

[HVALS key 8](#_Toc13037)

[HKEYS key 8](#_Toc8844)

[HLEN key 8](#_Toc333)

[HGETALL key 8](#_Toc22236)

[HINCRBY key field increment 8](#_Toc20646)

[HINCRBYFLOAT key field increment 8](#_Toc17429)

[List链表 8](#_Toc5842)

[BLPOP key1 [key2 ] timeout 9](#_Toc26928)

[BRPOP key1 [key2 ] timeout 9](#_Toc23876)

[BRPOPLPUSH source destination timeout 9](#_Toc7854)

[LINDEX key index 9](#_Toc9199)

[LINSERT key BEFORE|AFTER pivot value 9](#_Toc4191)

[LLEN key 10](#_Toc3170)

[LPOP key 10](#_Toc29332)

[LPUSH key value1 [value2] 10](#_Toc5821)

[LPUSHX key value 10](#_Toc13746)

[LRANGE key start stop 10](#_Toc18292)

[LREM key count value 10](#_Toc21370)

[LSET key index value 10](#_Toc1014)

[LTRIM key start stop 10](#_Toc27529)

[RPOP key 10](#_Toc9645)

[RPOPLPUSH source destination 11](#_Toc20697)

[RPUSH key value1 [value2] 11](#_Toc32252)

[RPUSHX key value 11](#_Toc30797)

[Set 集合 11](#_Toc6944)

[SADD key member1 [member2] 11](#_Toc30856)

[SCARD key 11](#_Toc23837)

[SDIFF key1 [key2] 11](#_Toc3493)

[SDIFFSTORE destination key1 [key2] 12](#_Toc8997)

[SINTER key1 [key2] 12](#_Toc26508)

[SINTERSTORE destination key1 [key2] 12](#_Toc23829)

[SISMEMBER key member 12](#_Toc32749)

[SMEMBERS key 12](#_Toc27134)

[SMOVE source destination member 12](#_Toc23857)

[SPOP key 12](#_Toc13268)

[SRANDMEMBER key [count] 12](#_Toc4580)

[SREM key member1 [member2] 12](#_Toc14778)

[SUNION key1 [key2] 13](#_Toc1114)

[Sorted Set 有序集合 13](#_Toc1410)

[ZADD key score1 member1 [score2 member2] 13](#_Toc10404)

[ZCARD key 13](#_Toc11866)

[ZCOUNT key min max 13](#_Toc26389)

[ZINCRBY key increment member 13](#_Toc22977)

[ZINTERSTORE destination numkeys key [key ...] 13](#_Toc31704)

[ZLEXCOUNT key min max 13](#_Toc30003)

[ZRANGE key start stop [WITHSCORES] 14](#_Toc16498)

[ZRANGEBYLEX key min max [LIMIT offset count] 14](#_Toc30258)

[ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT] 14](#_Toc5673)

[ZRANK key member 14](#_Toc7059)

[ZREM key member [member ...] 14](#_Toc8342)

[ZREMRANGEBYLEX key min max 14](#_Toc16954)

[ZREMRANGEBYRANK key start stop 14](#_Toc14841)

[ZREMRANGEBYSCORE key min max 14](#_Toc12455)

[ZREVRANGE key start stop [WITHSCORES] 14](#_Toc5899)

[ZREVRANGEBYSCORE key max min [WITHSCORES] 15](#_Toc6150)

[ZREVRANK key member 15](#_Toc32340)

[ZSCORE key member 15](#_Toc16028)

[ZUNIONSTORE destination numkeys key [key ...] 15](#_Toc22362)

[ZSCAN key cursor [MATCH pattern] [COUNT count] 15](#_Toc9932)

# Redis笔记

http://www.cnblogs.com/edisonchou/p/3821228.html

## 后Web2.0时代的发展要求遇到的问题

1.对数据库高并发读写的需求 (高并发)

2.对海量数据的高效率存储和访问的需求（大数据存取，高效率）

3.对数据库的高可扩展性和高可用性的需求 （高拓展与高可用）

## 数据库的主要特性

1.数据库事务一致性需求 （一致性）

2.数据库的写实时性和读实时性需求 （实时性）

3.对复杂的SQL查询，特别是多表关联查询的需求 （复杂查询，WEB2.0会尽量减少复杂查询）

总结:关系数据库在这些越来越多的应用场景下显得不那么合适了，为了解决这类问题的非关系数据库应运而生。NoSQL 是非关系型数据存储的广义定义。它打破了长久以来关系型数据库与ACID理论大一统的局面。NoSQL 数据存储不需要固定的表结构（例如以键值对存储，它的结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销），通常也不存在连接操作。[特点:数据结构不统一]

## NoSQL无与伦比的特点

在大数据存取上具备关系型数据库无法比拟的性能优势，例如：

　　（1）易扩展

　　NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。**数据之间无关系**，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

　　（2）大数据量，高性能

　　NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。

　　（3）灵活的数据模型

　　NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。这点在大数据量的Web2.0时代尤其明显。

　　（4）高可用

　　NoSQL在不太影响性能的情况，就可以方便的实现高可用的架构。比如Cassandra，HBase模型，通过复制模型也能实现高可用。

综上所述，NoSQL的非关系特性使其成为了后Web2.0时代的宠儿，助力大型Web2.0网站的再次起飞，是一项全新的数据库革命性运动。

## Redis特点

高性能，持久存储，适应高并发的应用场景

和Memcached类似，Redis支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，Redis支持各种不同方式的排序。与Memcached一样，为了保证效率，数据都是缓存在内存中。区别的是Redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步（数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。）。

　　因此，Redis的出现，很大程度补偿了Memcached这类key/value存储的不足，在部分场合可以对关系数据库起到很好的补充作用

## Redis中Key(键)常见使用

### Keys pattern

查找所有符合给定模式( pattern)的 key , pattern为正则表达式的内容

一般查询当前所属的redis库的所有键时，使用: keys \*

### Del key

该命令用于在 key 存在时删除 key , 例如:del abc;

### Exists key

检查给定 key 是否存在

### Expire key seconds/Expire key timestamp

为给定 key 设置过期时间。

### Persist key

移除 key 的过期时间，key 将持久保持

### TTL key

以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)

### Move key db

将当前数据库的 key 移动到给定的数据库 db 当中,例如:move abc 2;

### Rename key Newkey

修改 key 的名称

### Type key

返回 key 所储存的值的类型。

## Redis中常用数据类型

## String

String是最常用的一种数据类型，普通的key/value存储都可以归为此类 。一个Key对应一个Value，string类型是二进制安全的。Redis的string可以包含任何数据，比如jpg图片(生成二进制)或者序列化的对象

### SET key value

设置指定 key 的值

### GET key

设置指定 key 的值

### STRLEN KEY

返回 key 所储存的字符串值的长度。

### INCR KEY / INCRBY Key Increment

将 key 中储存的数字值增一。/将 key 所储存的值加上给定的增量值（increment）

### DECR Key/DECRBY KEY Decrement

将 key 中储存的数字值减一。/key 所储存的值减去给定的减量值（decrement） 。

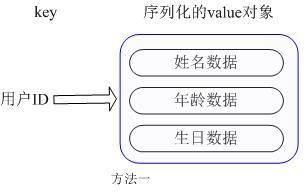
### APPEND Key value

如果 key 已经存在并且是一个字符串， APPEND 命令将 value 追加到 key 原来的值的末尾。

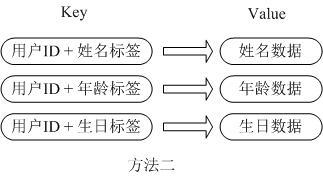
## Hash哈希

Hash是一个string 类型的field和value的映射表。Hash特别适合存储对象，相对于将对象的每个字段存成单个string 类型。一个对象存储在Hash类型中会占用更少的内存，并且可以更方便的存取整个对象。

　　这里借用[群叔](http://www.cnblogs.com/qunshu/p/3196972.html" \t "http://www.cnblogs.com/edisonchou/p/_blank)的描述，我们简单举个实例来描述下Hash的应用场景，比如我们要存储一个用户信息对象数据，包含以下信息：用户ID为查找的key，存储的value用户对象包含姓名，年龄，生日等信息，如果用普通的key/value结构来存储，主要有以下2种存储方式：

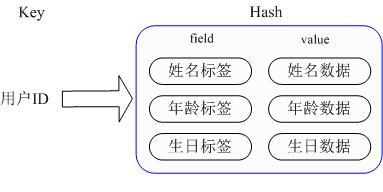


　　第一种方式将用户ID作为查找key,把其他信息封装成一个对象以序列化的方式存储，这种方式的缺点是，增加了序列化/反序列化的开销，并且在需要修改其中一项信息时，需要把整个对象取回，并且修改操作需要对并发进行保护，引入CAS等复杂问题。



　　第二种方法是这个用户信息对象有多少成员就存成多少个key-value对儿，用用户ID+对应属性的名称作为唯一标识来取得对应属性的值，虽然省去了序列化开销和并发问题，但是用户ID为重复存储，如果存在大量这样的数据，内存浪费还是非常可观的。

　　因此，基于以上两种方式的缺陷，Redis提供的Hash很好的解决了这个问题，Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，如下图：



也就是说，Key仍然是用户ID, value是一个Map，这个Map的key是成员的属性名，value是属性值，这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field), 也就是通过key(用户ID) + field(属性标签) 就可以操作对应属性数据了，既不需要重复存储数据，也不会带来序列化和并发修改控制的问题，也就很好的解决了问题。

### HSET key field value

将哈希表 key 中的字段 field 的值设为 value 。

### HMSET key field1 value1 [field2 value2 ]

同时将**多个** field-value (域-值)对设置到哈希表 key 中。

### HVALS key

获取哈希表中所有值

### HKEYS key

获取所有哈希表中的字段

### HLEN key

获取哈希表中字段的数量

### HGETALL key

获取在哈希表中指定 key 的所有字段和值

### HINCRBY key field increment

为哈希表 key 中的指定字段的整数值加上增量 increment 。

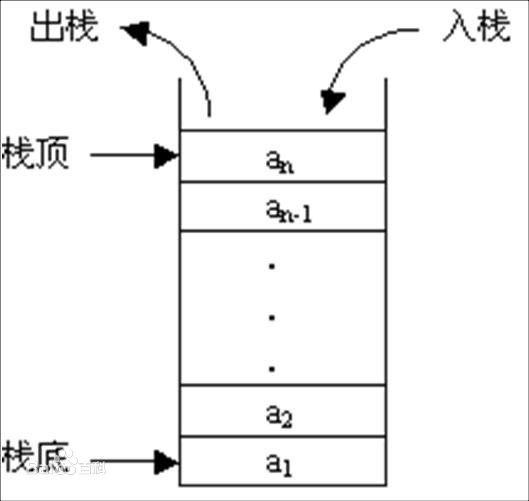
### HINCRBYFLOAT key field increment

为哈希表 key 中的指定字段的浮点数值加上增量 increment 。

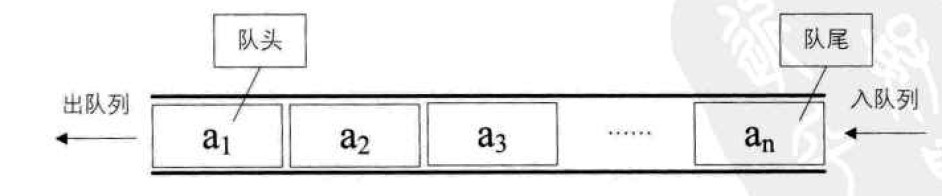
## List链表

Lst是一个链表结构，主要功能是push与pop，获取一个范围的所有的值等，操作中key理解为链表名字。 Redis的List类型其实就是一个每个子元素都是string类型的双向链表，我们可以通过push或pop操作从链表的头部或者尾部添加删除元素，这样List既可以作为栈，又可以作为队列。它即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

　　（1）现在我们首先来看看List作为（Stack）栈类型的使用：



（2）下面我们来看看List作为（Queue）队列的使用：



### BLPOP key1 [key2 ] timeout

移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

### BRPOP key1 [key2 ] timeout

移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

### BRPOPLPUSH source destination timeout

从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

### LINDEX key index

通过索引获取列表中的元素

### LINSERT key BEFORE|AFTER pivot value

在列表的元素前或者后插入元素

### LLEN key

获取列表长度

### LPOP key

移出并获取列表的第一个元素

### LPUSH key value1 [value2]

将一个或多个值插入到列表头部

### LPUSHX key value

将一个值插入到已存在的列表头部

### LRANGE key start stop

获取列表指定范围内的元素

### LREM key count value

移除列表元素

### LSET key index value

通过索引设置列表元素的值

### LTRIM key start stop

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

### RPOP key

移除并获取列表最后一个元素

### RPOPLPUSH source destination

移除列表的最后一个元素，并将该元素添加到另一个列表并返回

### RPUSH key value1 [value2]

在列表中添加一个或多个值

### RPUSHX key value

为已存在的列表添加值

## Set 集合

Set是string类型的无序集合。set是通过hash table实现的，添加，删除和查找，对集合我们可以取并集，交集，差集，可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

与List比较而言，set对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的。

### SADD key member1 [member2]

向集合添加一个或多个成员

### SCARD key

获取集合的成员数

### SDIFF key1 [key2]

返回给定所有集合的差集

### SDIFFSTORE destination key1 [key2]

返回给定所有集合的差集并存储在 destination 中

### SINTER key1 [key2]

返回给定所有集合的交集

### SINTERSTORE destination key1 [key2]

返回给定所有集合的交集并存储在 destination 中

### SISMEMBER key member

判断 member 元素是否是集合 key 的成员

### SMEMBERS key

返回集合中的所有成员

### SMOVE source destination member

将 member 元素从 source 集合移动到 destination 集合

### SPOP key

移除并返回集合中的一个随机元素

### SRANDMEMBER key [count]

返回集合中一个或多个随机数

### SREM key member1 [member2]

移除集合中一个或多个成员

### SUNION key1 [key2]

返回所有给定集合的并集

### SUNIONSTORE destination key1 [key2]

所有给定集合的并集存储在 destination 集合中

## Sorted Set 有序集合

Sorted Set 是set的一个升级版本，又被称为ZSet，它在set的基础上增加了一个顺序的属性，这一属性在添加修改。元素的时候可以指定，每次指定后，zset(表示有序集合)会自动重新按新的值调整顺序。可以理解为有列的表，一列存 value，一列存顺序。操作中key理解为zset的名字。

　　sorted set的使用场景与set类似，区别是set不是自动有序的，而sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set数据结构。此外，还可以用Sorted Sets来做带权重的队列，比如普通消息的score为1，重要消息的score为2，然后工作线程可以选择按score的倒序来获取工作任务。让重要的任务优先执行。

### ZADD key score1 member1 [score2 member2]

向有序集合添加一个或多个成员，或者更新已存在成员的分数

### ZCARD key

获取有序集合的成员数

### ZCOUNT key min max

计算在有序集合中指定区间分数的成员数

### ZINCRBY key increment member

有序集合中对指定成员的分数加上增量 increment

### ZINTERSTORE destination numkeys key [key ...]

计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中

### ZLEXCOUNT key min max

在有序集合中计算指定字典区间内成员数量

### ZRANGE key start stop [WITHSCORES]

通过索引区间返回有序集合成指定区间内的成员

### ZRANGEBYLEX key min max [LIMIT offset count]

通过字典区间返回有序集合的成员

### ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]

通过分数返回有序集合指定区间内的成员

### ZRANK key member

返回有序集合中指定成员的索引

### ZREM key member [member ...]

移除有序集合中的一个或多个成员

### ZREMRANGEBYLEX key min max

移除有序集合中给定的字典区间的所有成员

### ZREMRANGEBYRANK key start stop

移除有序集合中给定的排名区间的所有成员

### ZREMRANGEBYSCORE key min max

移除有序集合中给定的分数区间的所有成员

### ZREVRANGE key start stop [WITHSCORES]

返回有序集中指定区间内的成员，通过索引，分数从高到底

### ZREVRANGEBYSCORE key max min [WITHSCORES]

返回有序集中指定分数区间内的成员，分数从高到低排序

### ZREVRANK key member

返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序

### ZSCORE key member

返回有序集中，成员的分数值

### ZUNIONSTORE destination numkeys key [key ...]

计算给定的一个或多个有序集的并集，并存储在新的 key 中

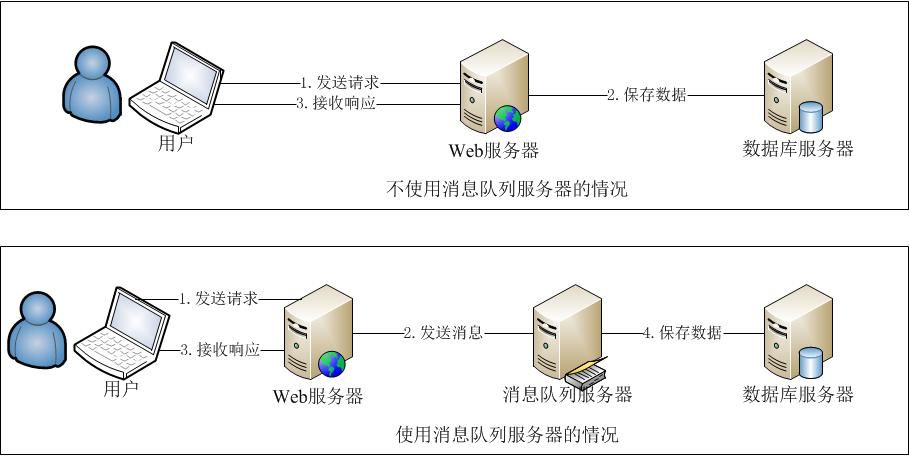
### ZSCAN key cursor [MATCH pattern] [COUNT count]

迭代有序集合中的元素（包括元素成员和元素分值）

# 消息队列

<http://www.cnblogs.com/edisonchou/p/3825682.html>

## 消息队列场景简介

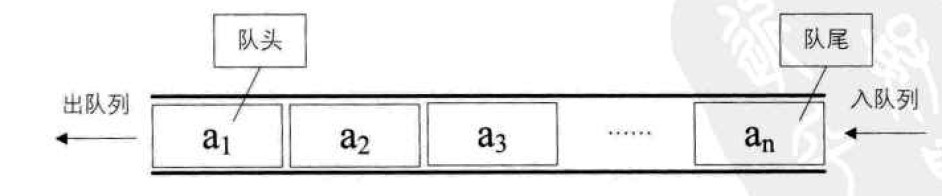


高并发的用户请求是网站成长过程中必不可少的过程，也是一个必须要解决的难题。在众多的实践当中，除了增加服务器数量配置服务器集群实现伸缩性架构设计之外，异步操作也被广泛采用。而异步操作中最核心的就是使用消息队列，通过消息队列，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务，改善网站系统的性能。在京东之类的电子商务网站促销活动中，**合理地使用消息队列，可以有效地抵御促销活动刚开始就开始大量涌入的订单对系统造成的冲击**。

记得我在实习期间，成都市XXXX局的一个价格信息采集发布系统项目中有一个采集任务发布的模块，其中每个任务都是一个事务，这个事务中需要向数据库中不断地插入行，每个任务发布时都要往表中插入几百行甚至几千行的任务数据（比如价格采集日报，往往需要发布2-3年的任务数据，每一天都是一个任务，所以大约有2,3千行任务期号数据，还要发给很多个区县的监测中心，因此数据库写操作量很大，更别说同时发布的并发操作），由于业务逻辑的处理比较复杂和往数据库的写操作量交大，所以在没有采用消息队列时点击“发布”按钮后往往需要等待1分钟左右的时间才提示“发布成功”，用户体验极不友好。



这时，我们就可以使用消息队列的思想来重构这个发布模块，在用户点击“发布”按钮后，系统只需要把往数据库插入的这个事务信息插入到指定的任务发布消息队列里边去（入队操作，这里一般有一台独立的消息队列服务器来单独存储和处理），然后系统就可以立即对用户的这个发布请求进行响应（比如给出一个发布成功的操作提示，这里暂不考虑消息队列服务操作失败的情形，如果失败了，可以考虑采用给用户发送邮件、短信或站内消息，让其重新进行发布操作）。

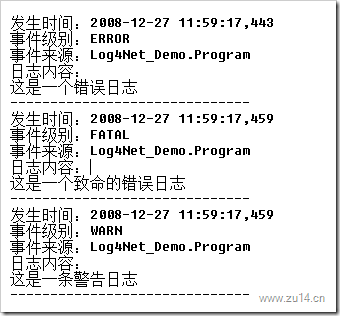


最后，消息队列服务器中有**一个进程单独对消息队列进行处理**，首先判断消息队列中是否有待处理的消息，如果有，则将其取出（出队操作，坚持“先进先出”的顺序，保证事务的准确性）进行相应地处理（比如这里是进行保存数据的操作，将数据插入到数据库服务器中的指定数据库里边，实质还是文件的IO操作）。就这样，通过消息队列将高并发用户请求进行异步操作，然后一一对消息队列进行出队的同步操作，也避免了并发控制的难题。

　　说到这里，大家可能会想到这尼玛不就是生产者消费者模式么？对的，么么嗒，消息队列就是生产者消费者模式的典型场景。简单地说，客户端不同用户发送的操作请求就是生产者，他们将要处理的事务存储到消息队列中，然后消息队列服务器的某个进程不停地将要处理的单个事务从消息队列中一个一个地取出来进行相应地处理，这就是消费者消费的过程。

　　下面我们将以异常日志为案例，介绍在.Net中如何采用消息队列的思想解决并发问题。当然，消息队列只是解决并发问题的其中一种方式，在实际中往往需要结合多种不同的技术方式来共同解决，比如负载均衡、反向代理、集群等方案。这里，虽然以异常日志为案例，但是“麻雀虽小五脏俱全”，日志写入文件的高并发操作也同样适用于数据库的高并发，所以，研究这个案例是具有实际意义的。

## 使用预置类型实现异常日志队列



在日常的Web应用中，异常日志的记录是一个十分重要的要点。因为，人无完人，系统也一样，难免会在什么时候出一个测试阶段未能完全测试到的异常。这时候，不能将异常信息直接显示给客户，那样既不友好也不安全。所以，一般都采用将异常信息记录到日志文件中（比如某个txt文件，数据库中某个表等），然后技术支持人员通过查看异常日志，分析异常原因，改进BUG重新发布，保障系统正常运行。

在用户的各种操作中，如果出现异常的时间一致，那么记录异常日志的操作就会成为并发操作，而记录异常日志又属于文件的IO操作（其实数据库的读写归根结底也是对文件即对磁盘进行的IO操作），因此很有可能带来并发控制的一系列问题。在以往的编码实践中，我们可以通过给不同的IO请求进行加锁（C#中的lock），等第一个请求完成写入后释放锁，第二个请求再获得锁，进行IO操作，然后释放掉，一直到第N个请求释放后结束。这种方式，虽然解决了并发操作带来的问题，但是通过加锁延迟了用户响应请求的时间（比如第一个正在IO写入操作时，后面的均处于等待状态），并且加锁也会给服务器带来一定的性能负担，造成服务器性能的下降。基于以上原因，我们采用消息队列的思想将异常日志的记录操作改为队列版.

通过使该类继承HandlerErrorAttribute并使其覆写OnException这个事件，代表在异常发生时可以进行的操作。而我们在这儿主要通过一个异常队列将获取的异常写入队列，然后跳转到自定义错误页：~/Common/CommonError.html，这个错误页很简单，就是简单的显示“系统发生错误，5秒后自动跳转到首页”

走到这里，生产者消费者模式中生产者的任务已经完成了，接下来消费者就需要开始消费了。也就是说，消息队列已经建好了，我们什么时候从队列中去任务，在哪里执行？怎么样执行？通过上面的介绍，我们知道，在专门的消息队列服务器中有一个进程在始终不停地监视消息队列，如果有需要待办的任务信息，则会立即从队列中取出来执行相应的操作，直到队列为空为止。于是，思路有了，我们马上来实现以下。这个消息监视的操作也是一个全局操作，在系统启动时就会一直运行，于是它也应该写在Application\_Start这个全局起始事件里边，于是按照标准的配置写法，我们在Application\_Start中添加了如下代码：MessageQueueConfig.RegisterExceptionLogQueue()；



现在，让我们来看看这段代码：

　　①首先定义Log文件存放的文件夹目录，这里我们一般放到App\_Data里边，因为放到这里边外网是无法访问到的，可以防止下载操作；

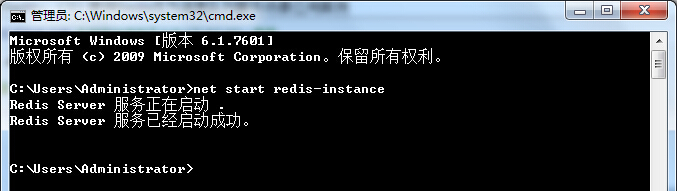
　　②其次通过线程池ThreadPool开启一个线程，不停地监听消息队列里边的待办事项个数，如果个数>0，则进行出队（FIFO，先入队的先出队）操作。这里主要是取出具体的异常实例对象，并将异常的具体堆栈信息追加写入到指定命名格式的文件中。

③如果该线程检测到消息队列中无待办事项，则使用Thread.Sleep使线程“休息”一会，避免了CPU空转（从理论上来说，CPU资源是很珍贵的，应该尽量提高CPU的利用率）。

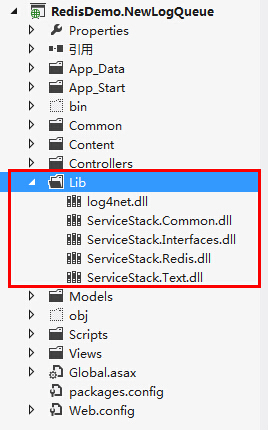
PS：许多应用程序创建的线程都要在休眠状态中消耗大量时间，以等待事件发生。其他线程可能进入休眠状态，只被定期唤醒以轮询更改或更新状态信息。线程池通过为应用程序提供一个由系统管理的辅助线程池使您可以更为有效地使用线程。关于线程池的更多信息请访问：http://msdn.microsoft.com/zh-cn/library/system.threading.threadpool(v=VS.90).aspx

## 使用Redis重构异常日志队列

（1）第一步，开启Redis的服务，这里我们使用命令开启Redis服务（之前已经将Redis注册到了Windows系统服务中了嘛，么么嗒）：net start redis-instance，当然，也可以通过在Windows服务列表中开启。



　　（2）第二步，在刚刚的版本1的Demo中新建一个文件夹，命名为Lib，将ServiceStack.Redis的dll和Log4Net的dll都拷贝进去。然后，在引用中添加对Lib文件夹中所有dll的引用。



　　（3）第三步，重写MyExceptionFilterAttribute这个全局异常信息过滤器。这里使用到了Redis的客户端连接池，每次连接时都是从池中取，不需要每次都创建，节省了时间和资源，提高了资源利用率。对于，多台Redis服务器组成的集群而言，这里需要指定多个形如 IP地址:端口号 的字符串数组。





# Redis的特性

Strings：Strings 数据结构是简单的key-value类型，value其实不仅是String，也可以是数字.  
常用命令: set,get,decr,incr,mget 等。  
常用方法：

获取字符串长度

往字符串append内容

设置和获取字符串的某一段内容

设置及获取字符串的某一位（bit）

批量设置一系列字符串的内容



Hashs：Redis Hash对应Value内部实际就是一个HashMap，常用命令：hget,hset,hgetall 等。



Lists：Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。  
常用命令：lpush,rpush,lpop,rpop,lrange等。



Sets：Sets 集合的概念就是一堆不重复值的组合。Redis还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好、二度好友等功能。  
常用命令：sadd,spop,smembers,sunion 等。



Sorted Sets：Redis sorted set的使用场景与set类似，区别是set不是自动有序的。sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。  
常用命令：zadd,zrange,zrem,zcard等



下面就根据这几个特性，也就是Redis支持的数据类型，来完成以下场景的实现。

