# Redis

Redis是一个开源，先进的key-value存储，并用于构建高性能，可扩展的Web应用程序的完美解决方案。

Redis从它的许多竞争继承来的三个主要特点：

* Redis数据库完全在内存中，使用磁盘仅用于持久性。
* 相比许多键值数据存储，Redis拥有一套较为丰富的数据类型。
* Redis可以将数据复制到任意数量的从服务器。

Redis 优势 :

* 异常快速：Redis的速度非常快，每秒能执行约11万集合，每秒约81000+条记录。
* 支持丰富的数据类型：Redis支持最大多数开发人员已经知道像列表，集合，有序集合，散列数据类型。这使得它非常容易解决各种各样的问题，因为我们知道哪些问题是可以处理通过它的数据类型更好。
* 操作都是原子性：所有Redis操作是原子的，这保证了如果两个客户端同时访问的Redis服务器将获得更新后的值。
* 多功能实用工具：Redis是一个多实用的工具，可以在多个用例如缓存，消息，队列使用(Redis原生支持发布/订阅)，任何短暂的数据，应用程序，如Web应用程序会话，网页命中计数等。

## 安装

Redis对于Linux是官方支持的, 但却不支持windows，如果想在windows折腾下redis，则可在<https://github.com/MSOpenTech/redis>上找到windows安装包，它是Microsoft Open Tech group 在 GitHub上开发了一个Win64的版本。

|  |
| --- |
| 下载解压后可以看到如下一些文件：  redis-benchmark.exe #基准测试  redis-check-aof.exe # aof  redis-check-dump.exe # dump  redis-cli.exe # 客户端  redis-server.exe # 服务器  redis.windows.conf # 配置文件 |
| 启动：  D:\workSoft\redis\Redis-x64-2.8.2400>redis-server redis.windows.conf |
| 客户端连接：  双击打开 **redis-cli.exe** |

## 命令

更多请参见：http://doc.redisfans.com/index.html

### Key

|  |
| --- |
| 127.0.0.1:6379> exists name //是否存在key  (integer) 1 |
| 127.0.0.1:6379> keys \*am\* //查看给定模式的key  1) "names"  2) "name"  KEYS \* 匹配数据库中所有 key 。  KEYS h?llo 匹配 hello ， hallo 和 hxllo 等。  KEYS h\*llo 匹配 hllo 和 heeeeello 等。  KEYS h[ae]llo 匹配 hello 和 hallo ，但不匹配 hillo 。 |
| 127.0.0.1:6379> type name //key 的类型  string  127.0.0.1:6379> type set  set  127.0.0.1:6379> type names  Hash  none (key不存在)  string (字符串)  list (列表)  set (集合)  zset (有序集)  hash (哈希表) |

### String

Redis字符串是字节序列。Redis字符串是二进制安全的，这意味着他们有一个已知的长度没有任何特殊字符终止，所以你可以存储任何东西，512兆为上限。

|  |
| --- |
| 127.0.0.1:6379> set name "cuihongtao" //设值  OK  **SET key value [EX seconds] [PX milliseconds] [NX|XX]**  EX second ：设置键的过期时间为 second 秒。 SET key value EX second 效果等同于 SETEX key second value 。  PX millisecond ：设置键的过期时间为 millisecond 毫秒。 SET key value PX millisecond 效果等同于 PSETEX key millisecond value 。  NX ：只在键不存在时，才对键进行设值操作。 SET key value NX 效果等同于 SETNX key value 。  XX ：只在键已经存在时，才对键进行设值操作。 |
| 127.0.0.1:6379> get name //取值  "cuihongtao" |
| 127.0.0.1:6379> strlen name  (integer) 10 |
| 127.0.0.1:6379> set name "tianpingping" //覆盖  OK  127.0.0.1:6379> get name  "tianpingping" |
| 127.0.0.1:6379> set name 10  OK  127.0.0.1:6379> incr name  (integer) 11 |

### Hash

Redis的哈希是键值对的集合。 Redis的哈希值是字符串字段和字符串值之间的映射，因此hash被用来表示对象

|  |
| --- |
| **127.0.0.1:6379> hset names cui "cuihongtao" //设置hashmap 某个key值**  **(integer) 1**  **HSET key field value** |
| **127.0.0.1:6379> hget names cui //获取hashmap 某个key值 存在返回值，不存在返回nil**  **"cuihongtao"**  **127.0.0.1:6379> hget names tian**  **(nil)** |
| **127.0.0.1:6379> hmset names tian "tianpingping" zhang "zhangsan" //设置多个key值**  **OK** |
| **127.0.0.1:6379> hmget names cui tian //取多个key值**  **1) "cuihongtao"**  **2) "tianpingping"**  **127.0.0.1:6379> hmget names cui tian zhang**  **1) "cuihongtao"**  **2) "tianpingping"**  **3) "zhangsan"** |
| **127.0.0.1:6379> hkeys names //取key 列表**  **1) "cui"**  **2) "tian"**  **3) "zhang"** |
| **127.0.0.1:6379> hvals names //取value列表**  **1) "cuihongtao"**  **2) "tianpingping"**  **3) "zhangsan"** |
| **127.0.0.1:6379> hlen names //长度**  **(integer) 3** |

### List

Redis的列表是简单的字符串列表，排序插入顺序。

|  |
| --- |
| # 对空列表(key 不存在)进行 LSET  127.0.0.1:6379> lset list 0 item  (error) ERR no such key  怎么办?  127.0.0.1:6379> lpush list item  (integer) 1  再次lset  127.0.0.1:6379> lset list 1 item  (error) ERR index out of range //  //用push  127.0.0.1:6379> lpush list item  (integer) 2  我们发现lset仅可对已存在的list，已有位置的元素进行设值操作 |
| Lpush list item item //没有key，则创建并存值，存在仅设值  127.0.0.1:6379> lpush list item item  (integer) 6 |
| 127.0.0.1:6379> lpop list //移除尾部，最后的元素  "item" |
| 127.0.0.1:6379> lindex list 5 //从0开始  "item" |
| 127.0.0.1:6379> lrange list 0 0 //  1) "item1"  127.0.0.1:6379> lrange list 0 5  1) "item1"  2) "item"  3) "item"  4) "item"  5) "item"  6) "item"  **LRANGE key start stop**  下标(index)参数 start 和 stop 都以 0 为底，也就是说，以 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。  你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素，以此类推。 |
| 127.0.0.1:6379> llen list  (integer) 6 |

### Set

Redis的集合是字符串的无序集合。

|  |
| --- |
| 127.0.0.1:6379> sadd set "abc"  (integer) 1  127.0.0.1:6379> sadd set "abc" //重复添加无效  (integer) 0 |
| 127.0.0.1:6379> spop set //随机删除一个元素  "abc" |
| 127.0.0.1:6379> scard set //集合大小  (integer) 4 |
| 127.0.0.1:6379> smembers set  1) "hij"  2) "lmn"  3) "efg"  4) "abc" |
| 127.0.0.1:6379> sismember set "abc" //是否含有成员 abc  (integer) 1 |
| 127.0.0.1:6379> srem set abc //移除一个元素  (integer) 1  127.0.0.1:6379> srem set dddd //移除不存在元素  (integer) 0  127.0.0.1:6379> srem set efg hij //移除多个元素  (integer) 2 |

### SortedSet

Redis的有序集合类似于Redis的集合，字符串不重复的集合。不同的是，一个有序集合的每个成员用分数，以便采取有序set命令，从最小的到最大的成员分数有关。虽然成员具有唯一性，但分数可能会重复。

|  |
| --- |
| 127.0.0.1:6379> zadd page\_rank 10 google.com  (integer) 1  127.0.0.1:6379> zadd page\_rank 9 baidu.com 8 bing.com  (integer) 2 |
| 127.0.0.1:6379> zcard page\_rank  (integer) 3 |
| 127.0.0.1:6379> zrange page\_rank 0 -1 //从小到大的顺序列出元素  1) "bing.com"  2) "baidu.com"  3) "google.com"  127.0.0.1:6379> zrange page\_rank 0 -1 withscores //从小到大的顺序列出元素 并显示其分  1) "bing.com"  2) "8"  3) "baidu.com"  4) "9"  5) "google.com"  6) "10" |
| 127.0.0.1:6379> zrank page\_rank baidu.com  (integer) 1 //第二名 |
| 127.0.0.1:6379> zrem page\_rank bing.com //移除元素  (integer) 1  127.0.0.1:6379> zrem page\_rank bing.com baidu.com //移除多个元素 不存在忽略  (integer) 1 |

### Pub/sub

### Transaction

|  |
| --- |
| **MULTI**  标记一个事务块的开始。  事务块内的多条命令会按照先后顺序被放进一个队列当中，最后由 [EXEC](http://doc.redisfans.com/transaction/exec.html#exec) 命令原子性(atomic)地执行。  Watch  监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。  UnWatch  取消 [WATCH](http://doc.redisfans.com/transaction/watch.html#watch) 命令对所有 key 的监视。  如果在执行 [WATCH](http://doc.redisfans.com/transaction/watch.html#watch) 命令之后， [EXEC](http://doc.redisfans.com/transaction/exec.html#exec) 命令或 [DISCARD](http://doc.redisfans.com/transaction/discard.html#discard) 命令先被执行了的话，那么就不需要再执行 [UNWATCH](http://doc.redisfans.com/transaction/unwatch.html#unwatch) 了。  因为 [EXEC](http://doc.redisfans.com/transaction/exec.html#exec) 命令会执行事务，因此 [WATCH](http://doc.redisfans.com/transaction/watch.html#watch) 命令的效果已经产生了；而 [DISCARD](http://doc.redisfans.com/transaction/discard.html#discard) 命令在取消事务的同时也会取消所有对 key 的监视，因此这两个命令执行之后，就没有必要执行 [UNWATCH](http://doc.redisfans.com/transaction/unwatch.html#unwatch) 了。  **DISCARD**  取消事务，放弃执行事务块内的所有命令。  如果正在使用 [*WATCH*](http://doc.redisfans.com/transaction/watch.html#watch) 命令监视某个(或某些) key，那么取消所有监视，等同于执行命令 [*UNWATCH*](http://doc.redisfans.com/transaction/unwatch.html#unwatch) 。  **EXEC**  执行所有事务块内的命令。  假如某个(或某些) key 正处于 [WATCH](http://doc.redisfans.com/transaction/watch.html#watch) 命令的监视之下，且事务块中有和这个(或这些) key 相关的命令，那么 [EXEC](http://doc.redisfans.com/transaction/exec.html#exec) 命令只在这个(或这些) key 没有被其他命令所改动的情况下执行并生效，否则该事务被打断(abort)。  127.0.0.1:6379> multi  OK  127.0.0.1:6379> incr name  QUEUED  127.0.0.1:6379> incr name  QUEUED  127.0.0.1:6379> exec  1) (integer) 12  2) (integer) 13  127.0.0.1:6379> |

### Script

### Connection

### Server

## Java

Redis的Java客户端Jedis的八种调用方式

### 直连方式

|  |
| --- |
| @Test  public void test1Normal() {      Jedis jedis = new Jedis("localhost");      long start = System.currentTimeMillis();      for (int i = 0; i < 100000; i++) {          String result = jedis.set("n" + i, "n" + i);      }      long end = System.currentTimeMillis();      System.out.println("Simple SET: " + ((end - start)/1000.0) + " seconds");      jedis.disconnect();  } |

### 事务方式

|  |
| --- |
| @Test  public void test2Trans() {      Jedis jedis = new Jedis("localhost");      long start = System.currentTimeMillis();      Transaction tx = jedis.multi();      for (int i = 0; i < 100000; i++) {          tx.set("t" + i, "t" + i);      }      List<Object> results = tx.exec();      long end = System.currentTimeMillis();      System.out.println("Transaction SET: " + ((end - start)/1000.0) + " seconds");      jedis.disconnect();  } |

### 管道方式

有时，我们需要采用异步方式，一次发送多个指令，不同步等待其返回结果。这样可以取得非常好的执行效率。

|  |
| --- |
| @Test  public void test3Pipelined() {      Jedis jedis = new Jedis("localhost");      Pipeline pipeline = jedis.pipelined();      long start = System.currentTimeMillis();      for (int i = 0; i < 100000; i++) {          pipeline.set("p" + i, "p" + i);      }      List<Object> results = pipeline.syncAndReturnAll();      long end = System.currentTimeMillis();      System.out.println("Pipelined SET: " + ((end - start)/1000.0) + " seconds");      jedis.disconnect();  } |

### 管道中调用事务

|  |
| --- |
| @Test  public void test4combPipelineTrans() {      jedis = new Jedis("localhost");      long start = System.currentTimeMillis();      Pipeline pipeline = jedis.pipelined();      pipeline.multi();      for (int i = 0; i < 100000; i++) {          pipeline.set("" + i, "" + i);      }      pipeline.exec();      List<Object> results = pipeline.syncAndReturnAll();      long end = System.currentTimeMillis();      System.out.println("Pipelined transaction: " + ((end - start)/1000.0) + " seconds");      jedis.disconnect();  } |

### 分布式直连同步调用

|  |
| --- |
| @Test  public void test5shardNormal() {      List<JedisShardInfo> shards = Arrays.asList(              new JedisShardInfo("localhost",6379),              new JedisShardInfo("localhost",6380));       ShardedJedis sharding = new ShardedJedis(shards);       long start = System.currentTimeMillis();      for (int i = 0; i < 100000; i++) {          String result = sharding.set("sn" + i, "n" + i);      }      long end = System.currentTimeMillis();      System.out.println("Simple@Sharing SET: " + ((end - start)/1000.0) + " seconds");       sharding.disconnect();  }   这个是分布式直接连接，并且是同步调用，每步执行都返回执行结果。 |

### 分布式直连异步调用

|  |
| --- |
| @Test  public void test6shardpipelined() {      List<JedisShardInfo> shards = Arrays.asList(              new JedisShardInfo("localhost",6379),              new JedisShardInfo("localhost",6380));       ShardedJedis sharding = new ShardedJedis(shards);       ShardedJedisPipeline pipeline = sharding.pipelined();      long start = System.currentTimeMillis();      for (int i = 0; i < 100000; i++) {          pipeline.set("sp" + i, "p" + i);      }      List<Object> results = pipeline.syncAndReturnAll();      long end = System.currentTimeMillis();      System.out.println("Pipelined@Sharing SET: " + ((end - start)/1000.0) + " seconds");       sharding.disconnect();  } |

### 分布式连接池同步调用

如果，你的分布式调用代码是运行在线程中，那么上面两个直连调用方式就不合适了，因为直连方式是非线程安全的，这个时候，你就必须选择连接池调用。

|  |
| --- |
| @Test  public void test7shardSimplePool() {      List<JedisShardInfo> shards = Arrays.asList(              new JedisShardInfo("localhost",6379),              new JedisShardInfo("localhost",6380));       ShardedJedisPool pool = new ShardedJedisPool(new JedisPoolConfig(), shards);       ShardedJedis one = pool.getResource();       long start = System.currentTimeMillis();      for (int i = 0; i < 100000; i++) {          String result = one.set("spn" + i, "n" + i);      }      long end = System.currentTimeMillis();      pool.returnResource(one);      System.out.println("Simple@Pool SET: " + ((end - start)/1000.0) + " seconds");       pool.destroy();  } |

### 分布式连接池异步调用

如果，你的分布式调用代码是运行在线程中，那么上面两个直连调用方式就不合适了，因为直连方式是非线程安全的，这个时候，你就必须选择连接池调用。

|  |
| --- |
| @Test  public void test8shardPipelinedPool() {      List<JedisShardInfo> shards = Arrays.asList(              new JedisShardInfo("localhost",6379),              new JedisShardInfo("localhost",6380));       ShardedJedisPool pool = new ShardedJedisPool(new JedisPoolConfig(), shards);       ShardedJedis one = pool.getResource();       ShardedJedisPipeline pipeline = one.pipelined();       long start = System.currentTimeMillis();      for (int i = 0; i < 100000; i++) {          pipeline.set("sppn" + i, "n" + i);      }      List<Object> results = pipeline.syncAndReturnAll();      long end = System.currentTimeMillis();      pool.returnResource(one);      System.out.println("Pipelined@Pool SET: " + ((end - start)/1000.0) + " seconds");      pool.destroy();  } |

事务和管道都是异步模式。在事务和管道中不能同步查询结果

事务和管道都是异步的，个人感觉，在管道中再进行事务调用，没有必要，不如直接进行事务模式。

分布式中，连接池的性能比直连的性能略好。

分布式调用中不支持事务。

因为事务是在服务器端实现，而在分布式中，每批次的调用对象都可能访问不同的机器，所以，没法进行事务。