Redis缓存技术学习

# 什么是Redis

Redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)和zset(有序集合)。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。

# 性能怎么样

Redis是一个高性能的key-value内存数据库。

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。它提供了Python，Ruby，Erlang，PHP客户端，使用很方便.

官方性能测试结果：

set操作每秒110000次，get操作每秒81000次。

# Redis与Memcache的比较

* 1. **网络IO模型模型**

**Memcached**是多线程，非阻塞IO复用的网络模型，分为监听主线程和worker子线程，监听线程监听网络连接，接受请求后，将连接描述字pipe 传递给worker线程，进行读写IO, 网络层使用libevent封装的事件库，多线程模型可以发挥多核作用，但是引入了cache coherency和锁的问题，比如，Memcached最常用的stats 命令，实际Memcached所有操作都要对这个全局变量加锁，进行计数等工作，带来了性能损耗。

**Redis**使用单线程的IO复用模型，自己封装了一个简单的AeEvent事件处理框架，主要实现了epoll、kqueue和select，对于单纯只 有IO操作来说，单线程可以将速度优势发挥到最大，但是Redis也提供了一些简单的计算功能，比如排序、聚合等，对于这些操作，单线程模型实际会严重影 响整体吞吐量，CPU计算过程中，整个IO调度都是被阻塞住的。

* 1. **内存管理方面**

**Memcached**使用预分配的内存池的方式，使用slab和大小不同的chunk来管理内存，Item根据大小选择合适的chunk存储，内存池的方式可以省去申请/释放内存的开销，并且能 减小内存碎片产生，但这种方式也会带来一定程度上的空间浪费，并且在内存仍然有很大空间时，新的数据也可能会被剔除，原因可以参考Timyang的文 章：http://timyang.net/data/Memcached-lru-evictions/

**Redis**使用现场申请内存的方式来存储数据，并且很少使用free-list等方式来优化内存分配，会在一定程度上存在内存碎片，Redis跟据存储命 令参数，会把带过期时间的数据单独存放在一起，并把它们称为临时数据，非临时数据是永远不会被剔除的，即便物理内存不够，导致swap也不会剔除任何非临 时数据(但会尝试剔除部分临时数据)，这点上Redis更适合作为存储而不是cache。

* 1. **数据一致性问题**

**Memcached**提供了cas命令，可以保证多个并发访问操作同一份数据的一致性问题。

CAS是Memcached中比较方便的一种防止竞争修改资源的方法。CAS实现需要为每个cache key设置一个隐藏的cas token，cas相当value版本号，每次set会token需要递增，因此带来CPU和内存的双重开销，虽然这些开销很小，但是到单机10G+ cache以及QPS上万之后这些开销就会给双方相对带来一些细微性能差别

**Redis**没有提供cas 命令，并不能保证这点，不过Redis提供了事务的功能，可以保证一串 命令的原子性，中间不会被任何操作打断。

* 1. **存储方式及其他方面**

**Memcached**基本只支持简单的key-value存储，不支持枚举，不支持持久化和复制等功能

**Redis**除key/value之外，还支持list,set,sorted set,hash等众多数据结构，提供了KEYS进行枚举操作，但不能在线上使用，如果需要枚举线上数据，Redis提供了工具可以直接扫描其dump文件，枚举出所有数据，Redis还同时提供了持久化和复制等功能。

* 1. **关于不同语言的客户端支持**

在不同语言的客户端方面，Memcached和Redis都有丰富的第三方客户端可供选择，不过因为Memcached发展的时间更久一些，目前看在 客户端支持方面，Memcached的很多客户端更加成熟稳定，而Redis由于其协议本身就比Memcached复杂，加上作者不断增加新的功能等，对应第三方客户端跟进速度可能会赶不上，有时可能需要自己在第三方客户端基础上做些修改才能更好的使用

**总结：**

　　1.Redis使用最佳方式是全部数据in-memory。

　　2.Redis更多场景是作为Memcached的替代者来使用。

　　3.当需要除key/value之外的更多数据类型支持时，使用Redis更合适。

　　4.当存储的数据不能被剔除时，使用Redis更合适。

# 单台Redis的存放数据必须比物理内存小

Redis的数据全部放在内存带来了高速的性能，但是也带来一些不合理之处。比如一个中型网站有100万注册用户，如果这些资料要用Redis来存储，内存的容量必须能够容纳这100万用户。但是业务实际情况是100万用户只有5万活跃用户，1周来访问过1次的也只有15万用户，因此全部100万用户的数据都放在内存有不合理之处，RAM需要为冷数据买单。

这跟操作系统非常相似，操作系统所有应用访问的数据都在内存，但是如果物理内存容纳不下新的数据，操作系统会智能将部分长期没有访问的数据交换到磁盘，为新的应用留出空间。现代操作系统给应用提供的并不是物理内存，而是虚拟内存(Virtual Memory)的概念。

基于相同的考虑，Redis 2.0也增加了VM特性。让Redis数据容量突破了物理内存的限制。并实现了数据冷热分离。

# Redis的VM实现是重复造轮子

Redis的VM依照之前的epoll实现思路依旧是自己实现。但是在前面操作系统的介绍提到OS也可以自动帮程序实现冷热数据分离，Redis只需要OS申请一块大内存，OS会自动将热数据放入物理内存，冷数据交换到硬盘，另外一个知名的“理解了现代操作系统”的Varnish就是这样实现，也取得了非常成功的效果。

Redis自己实现VM的原因。主要OS的VM换入换出是基于Page概念，比如OS VM1个Page是4K, 4K中只要还有一个元素即使只有1个字节被访问，这个页也不会被SWAP, 换入也同样道理，读到一个字节可能会换入4K无用的内存。而Redis自己实现则可以达到控制换入的粒度。另外访问操作系统SWAP内存区域时block进程，也是导致Redis要自己实现VM原因之一。

# 用get/set方式使用Redis

作为一个key value存在，很多开发者自然的使用set/get方式来使用Redis，实际上这并不是最优化的使用方法。尤其在未启用VM情况下，Redis全部数据需要放入内存，节约内存尤其重要。

假如一个key-value单元需要最小占用512字节，即使只存一个字节也占了512字节。这时候就有一个设计模式，可以把key复用，几个key-value放入一个key中，value再作为一个set存入，这样同样512字节就会存放10-100倍的容量。

这就是为了节约内存，建议使用hashset而不是set/get的方式来使用Redis

# 使用aof代替snapshot

Redis有两种存储方式，默认是snapshot方式，实现方法是定时将内存的快照(snapshot)持久化到硬盘，这种方法缺点是持久化之后如果出现crash则会丢失一段数据。因此在完美主义者的推动下作者增加了aof方式。aof即append only mode，在写入内存数据的同时将操作命令保存到日志文件，在一个并发更改上万的系统中，命令日志是一个非常庞大的数据，管理维护成本非常高，恢复重建时间会非常长，这样导致失去aof高可用性本意。另外更重要的是Redis是一个内存数据结构模型，所有的优势都是建立在对内存复杂数据结构高效的原子操作上，这样就看出aof是一个非常不协调的部分。

其实aof目的主要是数据可靠性及高可用性，在Redis中有另外一种方法来达到目的：Replication。由于Redis的高性能，复制基本没有延迟。这样达到了防止单点故障及实现了高可用。

# Redis是否支持集群

支持

Redis主从复制配置和使用都非常简单。通过主从复制可以允许多个slave server拥有和master server相同的数据库副本。下面是关于Redis主从复制的一些特点

1.master可以有多个slave

2.除了多个slave连到相同的master外，slave也可以连接其他slave形成图状结构

3.主从复制不会阻塞master。也就是说当一个或多个slave与master进行初次同步数据时，master可以继续处理client发来的请求。相反slave在初次同步数据时则会阻塞不能处理client的请求。

4.主从复制可以用来提高系统的可伸缩性,我们可以用多个slave 专门用于client的读请求，比如sort操作可以使用slave来处理。也可以用来做简单的数据冗余

5.可以在master禁用数据持久化，只需要注释掉master 配置文件中的所有save配置，然后只在slave上配置数据持久化。

下面介绍下主从复制的过程

当设置好slave服务器后，slave会建立和master的连接，然后发送sync命令。无论是第一次同步建立的连接还是连接断开后的重新连 接，master都会启动一个后台进程，将数据库快照保存到文件中，同时master主进程会开始收集新的写命令并缓存起来。后台进程完成写文件 后，master就发送文件给slave，slave将文件保存到磁盘上，然后加载到内存恢复数据库快照到slave上。接着master就会把缓存的命 令转发给slave。而且后续master收到的写命令都会通过开始建立的连接发送给slave。从master到slave的同步数据的命令和从 client发送的命令使用相同的协议格式。当master和slave的连接断开时slave可以自动重新建立连接。如果master同时收到多个 slave发来的同步连接命令，只会使用启动一个进程来写数据库镜像，然后发送给所有slave。

# 在多台服务器上简单实现Redis的数据主从复制

Redis的主从复制功能非常强大，一个master可以拥有多个slave，而一个slave又可以拥有多个slave，如此下去，形成了强大的多级服务器集群架构。下面我演示下怎样在多台服务器上进行Redis数据主从复制。这里我假设有两台服务器，一台是Windows操作系统（局域网IP：192.168.3.82），一台是Linux操作系统（局域网IP：192.168.3.90），在两个操作系统都安装Redis，Windows操作系统使用cygwin工具进行安装，命令为：

view sourceprint?

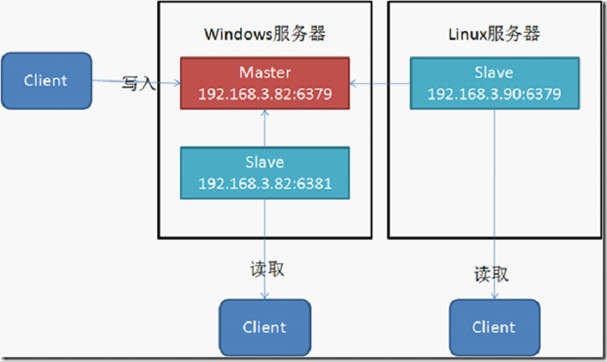
$ tar xzf Redis-2.2.2.tar.gz

$ cd Redis-2.2.2

$ make

可以通过”make test”命令判断是否安装成功。

这里我使用1个master以及2个slave（master在Windows下，一个slave在Windows下，一个slave在Linux下），基本流程是：



1. 在Windows服务器上创建两个目录，Demo1，Demo2，其中Demo1用来存放Master服务，Demo2用来存放Slave服务，

在Master服务中的配置文件修改：

view sourceprint?

bind 192.168.3.82

在Slave服务中的配置文件修改：

view sourceprint?

port 6381（服务端口号要分开）

bind 192.168.3.82

slaveof 192.168.3.82 6379 （设置master的Host以及Port）

2. 在Linux服务器上创建一个目录，Demo，Demo存放Slave服务，在服务中的配置文件修改：

view sourceprint?

bind 192.168.3.90

slaveof 192.168.3.82 6379（设置master的Host以及Port）

这样就完成了所有的配置。

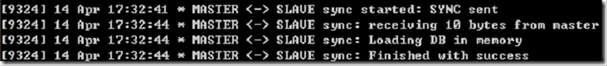
3. 现在运行这3个服务，通过命令：

view sourceprint?

./Redis-server Redis.conf

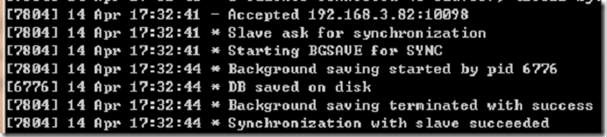
来启动Redis服务。

注意到，当我启动master，然后启动一个slave的时候，可以发现slave上：



会发送一个SYNC请求，从Master上面进行相应，而且它支持自动重连，即当master掉线的情况下，它会处于等待请求的状态。

而Master上：



能够接受Slave的应答，并且开始持久化操作，说明在Slave每次去连接Master的时候，都会去持久化磁盘。

4. 现在开始写一个客户端程序，使用到ServiceStack.Redis.dll的.NET组件：

view sourceprint?

using ServiceStack.Redis;

static void Main(string[] args)

{

IRedisClientFactory factory = new RedisCacheClientFactory();

IRedisClient client = factory.CreateRedisClient(“192.168.3.82″, 6379);

client.Set<STRING>(“username”, “leepy”);

string username = client.Get<STRING>(“username”);

client.Save();

Console.WriteLine(“username: {0}”, username);

Console.ReadLine();

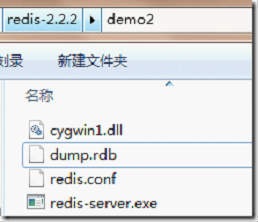
}

运行结果：



数据Set的时候，数据保存在内存中，当调用Save方法时候，将数据保存在磁盘中。

其中你会发现在3个服务目录中，都出现了dump.rdb，说明Master的文件都同步到Slave中去了。





用UE编辑器打开文件查看：

  
从Redis源码中，可以发现rdb文件采用的是lzf压缩算法进行实现，默认lzf压缩算法是开启的。

# 安装与使用

* 1. **Win下安装**
  2. **Linux下安装**

**下载地址：**<http://www.redis.net.cn/download/>，下载最新文档版本。

本教程使用的最新文档版本为 2.8.17，下载并安装：

1. $ wget http://download.redis.io/releases/redis-2.8.17.tar.gz
2. $ tar xzf redis-2.8.17.tar.gz
3. $ cd redis-2.8.17
4. $ make

make完后 redis-2.8.17目录下会出现编译后的redis服务程序redis-server,还有用于测试的客户端程序redis-cli

下面启动redis服务.

1. $ ./redis-server

注意这种方式启动redis 使用的是默认配置。也可以通过启动参数告诉redis使用指定配置文件使用下面命令启动。

1. $ ./redis-server redis.conf

redis.conf是一个默认的配置文件。我们可以根据需要使用自己的配置文件。

启动redis服务进程后，就可以使用测试客户端程序redis-cli和redis服务交互了。 比如：

1. $ ./redis-cli
2. redis> set foo bar
3. OK
4. redis> get foo
5. "bar"
   1. **Ubuntu下安装**

要配置参数的意义：

· daemonize：是否以后台daemon方式运行

· pidfile：pid文件位置

· port：监听的端口号

· timeout：请求超时时间

· loglevel：log信息级别

· logfile：log文件位置

· databases：开启数据库的数量

· save \* \*：保存快照的频率，第一个\*表示多长时间，第三个\*表示执行多少次写操作。在一定时间内执行一定数量的写操作时，自动保存快照。可设置多个条件。

· rdbcompression：是否使用压缩

· dbfilename：数据快照文件名（只是文件名，不包括目录）

· dir：数据快照的保存目录（这个是目录）

· appendonly：是否开启appendonlylog，开启的话每次写操作会记一条log，这会提高数据抗风险能力，但影响效率。

· appendfsync：appendonlylog如何同步到磁盘（三个选项，分别是每次写都强制调用fsync、每秒启用一次fsync、不调用fsync等待系统自己同步）

* 1. **Redis命令总结**

Redis提供了丰富的命令（command）对数据库和各种数据类型进行操作，这些command可以在Linux终端使用。在编程时，比如使用Redis 的Java语言包，这些命令都有对应的方法，比如上面例子中使用的sadd方法，就是对集合操作中的SADD命令。下面将Redis提供的命令做一总结。

### 连接操作相关的命令

|  |  |  |
| --- | --- | --- |
| 编号 | 命令 | 解释 |
| 1 | quit | 关闭连接（connection） |
| 2 | auth | 简单密码认证 |

### 对value操作的命令

|  |  |  |
| --- | --- | --- |
| 编号 | 命令 | 解释 |
| 1 | exists(key) | 确认一个key是否存在 |
| 2 | del(key) | 删除一个key |
| 3 | type(key) | 返回值的类型 |
| 4 | keys(pattern) | 返回满足给定pattern的所有key |
| 5 | randomkey | 随机返回key空间的一个key |
| 6 | rename(oldname, newname) | 将key由oldname重命名为newname，若newname存在则删除newname表示的key |
| 7 | dbsize | 返回当前数据库中key的数目 |
| 8 | expire | 设定一个key的活动时间（s） |
| 9 | ttl | 获得一个key的活动时间 |
| 10 | select(index) | 按索引查询 |
| 11 | move(key, dbindex) | 将当前数据库中的key转移到有dbindex索引的数据库 |
| 12 | flushdb | 删除当前选择数据库中的所有key |
| 13 | flushall | 删除所有数据库中的所有key |

### 对String操作的命令

|  |  |
| --- | --- |
| 命令 | 解释 |
| set(key, value) | 给数据库中名称为key的string赋予值value |
| get(key) | 返回数据库中名称为key的string的value |
| getset(key, value) | 给名称为key的string赋予上一次的value |
| mget(key1, key2,…, key N) | 返回库中多个string（它们的名称为key1，key2…）的value |
| setnx(key, value) | 如果不存在名称为key的string，则向库中添加string，名称为key，值为value |
| setex(key, time, value) | 向库中添加string（名称为key，值为value）同时，设定过期时间time |
| mset(key1, value1, key2, value2,…key N, value N) | 同时给多个string赋值，名称为key i的string赋值value i |
| msetnx(key1, value1, key2, value2,…key N, value N) | 如果所有名称为key i的string都不存在，则向库中添加string，名称key i赋值为value i |
| incr(key) | 名称为key的string增1操作 |
| incrby(key, integer) | 名称为key的string增加integer |
| decr(key) | 名称为key的string减1操作 |
| decrby(key, integer) | 名称为key的string减少integer |
| append(key, value) | 名称为key的string的值附加value |
| substr(key, start, end) | 返回名称为key的string的value的子串 |

### 对List操作的命令

rpush(key, value)：在名称为key的list尾添加一个值为value的元素

lpush(key, value)：在名称为key的list头添加一个值为value的 元素

llen(key)：返回名称为key的list的长度

lrange(key, start, end)：返回名称为key的list中start至end之间的元素（下标从0开始，下同）

ltrim(key, start, end)：截取名称为key的list，保留start至end之间的元素

lindex(key, index)：返回名称为key的list中index位置的元素

lset(key, index, value)：给名称为key的list中index位置的元素赋值为value

lrem(key, count, value)：删除count个名称为key的list中值为value的元素。count为0，删除所有值为value的元素，count>0从头至尾删除count个值为value的元素，count<0从尾到头删除|count|个值为value的元素。

lpop(key)：返回并删除名称为key的list中的首元素

rpop(key)：返回并删除名称为key的list中的尾元素

blpop(key1, key2,… key N, timeout)：lpop命令的block版本。即当timeout为0时，若遇到名称为key i的list不存在或该list为空，则命令结束。如果timeout>0，则遇到上述情况时，等待timeout秒，如果问题没有解决，则对key i+1开始的list执行pop操作。

brpop(key1, key2,… key N, timeout)：rpop的block版本。参考上一命令。

rpoplpush(srckey, dstkey)：返回并删除名称为srckey的list的尾元素，并将该元素添加到名称为dstkey的list的头部

### 对Set操作的命令

sadd(key, member)：向名称为key的set中添加元素member

srem(key, member) ：删除名称为key的set中的元素member

spop(key) ：随机返回并删除名称为key的set中一个元素

smove(srckey, dstkey, member) ：将member元素从名称为srckey的集合移到名称为dstkey的集合

scard(key) ：返回名称为key的set的基数

sismember(key, member) ：测试member是否是名称为key的set的元素

sinter(key1, key2,…key N) ：求交集

sinterstore(dstkey, key1, key2,…key N) ：求交集并将交集保存到dstkey的集合

sunion(key1, key2,…key N) ：求并集

sunionstore(dstkey, key1, key2,…key N) ：求并集并将并集保存到dstkey的集合

sdiff(key1, key2,…key N) ：求差集

sdiffstore(dstkey, key1, key2,…key N) ：求差集并将差集保存到dstkey的集合

smembers(key) ：返回名称为key的set的所有元素

srandmember(key) ：随机返回名称为key的set的一个元素

### 对zset（sorted set）操作的命令

zadd(key, score, member)：向名称为key的zset中添加元素member，score用于排序。如果该元素已经存在，则根据score更新该元素的顺序。

zrem(key, member) ：删除名称为key的zset中的元素member

zincrby(key, increment, member) ：如果在名称为key的zset中已经存在元素member，则该元素的score增加increment；否则向集合中添加该元素，其score的值为increment

zrank(key, member) ：返回名称为key的zset（元素已按score从小到大排序）中member元素的rank（即index，从0开始），若没有member元素，返回“nil”

zrevrank(key, member) ：返回名称为key的zset（元素已按score从大到小排序）中member元素的rank（即index，从0开始），若没有member元素，返回“nil”

zrange(key, start, end)：返回名称为key的zset（元素已按score从小到大排序）中的index从start到end的所有元素

zrevrange(key, start, end)：返回名称为key的zset（元素已按score从大到小排序）中的index从start到end的所有元素

zrangebyscore(key, min, max)：返回名称为key的zset中score >= min且score <= max的所有元素

zcard(key)：返回名称为key的zset的基数

zscore(key, element)：返回名称为key的zset中元素element的score

zremrangebyrank(key, min, max)：删除名称为key的zset中rank >= min且rank <= max的所有元素

zremrangebyscore(key, min, max) ：删除名称为key的zset中score >= min且score <= max的所有元素

zunionstore / zinterstore(dstkeyN, key1,…,keyN, WEIGHTS w1,…wN, AGGREGATE SUM|MIN|MAX)：对N个zset求并集和交集，并将最后的集合保存在dstkeyN中。对于集合中每一个元素的score，在进行AGGREGATE运算前，都要乘以对于的WEIGHT参数。如果没有提供WEIGHT，默认为1。默认的AGGREGATE是SUM，即结果集合中元素的score是所有集合对应元素进行SUM运算的值，而MIN和MAX是指，结果集合中元素的score是所有集合对应元素中最小值和最大值。

### 对Hash操作的命令

hset(key, field, value)：向名称为key的hash中添加元素field<—>value

hget(key, field)：返回名称为key的hash中field对应的value

hmget(key, field1, …,field N)：返回名称为key的hash中field i对应的value

hmset(key, field1, value1,…,field N, value N)：向名称为key的hash中添加元素field i<—>value i

hincrby(key, field, integer)：将名称为key的hash中field的value增加integer

hexists(key, field)：名称为key的hash中是否存在键为field的域

hdel(key, field)：删除名称为key的hash中键为field的域

hlen(key)：返回名称为key的hash中元素个数

hkeys(key)：返回名称为key的hash中所有键

hvals(key)：返回名称为key的hash中所有键对应的value

hgetall(key)：返回名称为key的hash中所有的键（field）及其对应的value

### 持久化

save：将数据同步保存到磁盘

bgsave：将数据异步保存到磁盘

lastsave：返回上次成功将数据保存到磁盘的Unix时戳

shundown：将数据同步保存到磁盘，然后关闭服务

### 远程服务控制

info：提供服务器的信息和统计

monitor：实时转储收到的请求

slaveof：改变复制策略设置

config：在运行时配置Redis服务器

在192.168.134.96 上安装了Redis的master

在192.168.134.97 上安装了slave 绑定了192.168.134.96

# Redis事务

Redis对事务的支持目前还比较简单。Redis只能保证一个client发起的事务中的命令可以连续的执行，而中间不会插入其他client的命令。 由于Redis是单线程来处理所有client的请求的所以做到这点是很容易的。一般情况下Redis在接受到一个client发来的命令后会立即处理并 返回处理结果，但是当一个client在一个连接中发出multi命令有，这个连接会进入一个事务上下文，该连接后续的命令并不是立即执行，而是先放到一 个队列中。当从此连接受到exec命令后，Redis会顺序的执行队列中的所有命令。并将所有命令的运行结果打包到一起返回给client.然后此连接就 结束事务上下文。下面可以看一个例子

Redis> multi  
OK  
Redis> incr a  
QUEUED  
Redis> incr b  
QUEUED  
Redis> exec  
1. (integer) 1  
2. (integer) 1

从这个例子我们可以看到incr a ,incr b命令发出后并没执行而是被放到了队列中。调用exec后俩个命令被连续的执行，最后返回的是两条命令执行后的结果  
我们可以调用discard命令来取消一个事务。接着上面例子

Redis> multi  
OK  
Redis> incr a  
QUEUED  
Redis> incr b  
QUEUED  
Redis> discard  
OK  
Redis> get a  
“1”  
Redis> get b  
“1”

可以发现这次incr a incr b都没被执行。discard命令其实就是清空事务的命令队列并退出事务上下文。  
虽说Redis事务在本质上也相当于序列化隔离级别的了。但是由于事务上下文的命令只排队并不立即执行，所以事务中的写操作不能依赖事务中的读操作结果。看下面例子

Redis> multi  
OK  
Redis> get a  
QUEUED  
Redis> get b  
QUEUED  
Redis> exec  
1. “1”  
2. “1”

发现问题了吧。假如我们想用事务实现incr操作怎么办？可以这样做吗？

Redis> get a  
“1”  
Redis> multi  
OK  
Redis> set a 2  
QUEUED  
Redis> exec  
1. OK  
Redis> get a，  
“2”

结论很明显这样是不行的。这样和 get a 然后直接set a是没区别的。很明显由于get a 和set a并不能保证两个命令是连续执行的(get操作不在事务上下文中)。很可能有两个client同时做这个操作。结果我们期望是加两次a从原来的1变成3. 但是很有可能两个client的get a，取到都是1，造成最终加两次结果却是2。主要问题我们没有对共享资源a的访问进行任何的同步  
也就是说Redis没提供任何的加锁机制来同步对a的访问。  
还好Redis 2.1后添加了watch命令，可以用来实现乐观锁。看个正确实现incr命令的例子,只是在前面加了watch a

Redis> watch a  
OK  
Redis> get a  
“1”  
Redis> multi  
OK  
Redis> set a 2  
QUEUED  
Redis> exec  
1. OK  
Redis> get a，  
“2”

watch 命令会监视给定的key,当exec时候如果监视的key从调用watch后发生过变化，则整个事务会失败。也可以调用watch多次监视多个key.这 样就可以对指定的key加乐观锁了。注意watch的key是对整个连接有效的，事务也一样。如果连接断开，监视和事务都会被自动清除。当然了 exec,discard,unwatch命令都会清除连接中的所有监视.  
Redis的事务实现是如此简单，当然会存在一些问题。第一个问题是Redis只能保证事务的每个命令连续执行，但是如果事务中的一个命令失败了，并不回滚其他命令，比如使用的命令类型不匹配。

Redis> set a 5  
OK  
Redis> lpush b 5  
(integer) 1  
Redis> set c 5  
OK  
Redis> multi  
OK  
Redis> incr a  
QUEUED  
Redis> incr b  
QUEUED  
Redis> incr c  
QUEUED  
Redis> exec  
1. (integer) 6  
2. (error) ERR Operation against a key holding the wrong kind of value  
3. (integer) 6

可以看到虽然incr b失败了，但是其他两个命令还是执行了。  
最 后一个十分罕见的问题是 当事务的执行过程中，如果Redis意外的挂了。很遗憾只有部分命令执行了，后面的也就被丢弃了。当然如果我们使用的append-only file方式持久化，Redis会用单个write操作写入整个事务内容。即是是这种方式还是有可能只部分写入了事务到磁盘。发生部分写入事务的情况 下，Redis重启时会检测到这种情况，然后失败退出。可以使用Redis-check-aof工具进行修复，修复会删除部分写入的事务内容。修复完后就 能够重新启动了。

# Redis的持久化

Redis是一个支持持久化的内存数据库，也就是说Redis需要经常将内存中的数据同步到磁盘来保证持久化。Redis支持两种持久化方式，一种是 Snapshotting（快照）也是默认方式，另一种是Append-only file（缩写aof）的方式。下面分别介绍  
Snapshotting  
快照是默认的持久化方式。这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。可以通过配置设置自动做快照持久 化的方式。我们可以配置Redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置  
save 900 1 #900秒内如果超过1个key被修改，则发起快照保存  
save 300 10 #300秒内容如超过10个key被修改，则发起快照保存  
save 60 10000  
下面介绍详细的快照保存过程  
1.Redis调用fork,现在有了子进程和父进程。  
2. 父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的写时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数 据是fork时刻整个数据库的一个快照。  
3.当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。  
client 也可以使用save或者bgsave命令通知Redis做一次快照持久化。save操作是在主线程中保存快照的，由于Redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不 是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io操作，可能会严重影响性能。  
另外由于快照方式是在一定间隔时间做一次的，所以如果Redis意外down掉的话，就会丢失最后一次快照后的所有修改。如果应用要求不能丢失任何修改的话，可以采用aof持久化方式。下面介绍  
Append-only file  
aof 比快照方式有更好的持久化性，是由于在使用aof持久化方式时,Redis会将每一个收到的写命令都通过write函数追加到文件中(默认是 appendonly.aof)。当Redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉Redis我们想要 通过fsync函数强制os写入到磁盘的时机。有三种方式如下（默认是：每秒fsync一次）  
appendonly yes //启用aof持久化方式  
# appendfsync always //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用  
appendfsync everysec //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐  
# appendfsync no //完全依赖os，性能最好,持久化没保证  
aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。为了压缩aof的持久化文件。Redis提供了bgrewriteaof命令。收到此命令Redis将使用与快照类似的方式将内存中的数据 以命令的方式保存到临时文件中，最后替换原来的文件。具体过程如下  
1. Redis调用fork ，现在有父子两个进程  
2. 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令  
3.父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。  
4.当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。  
5.现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。  
需要注意到是重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件,这点和快照有点类似。