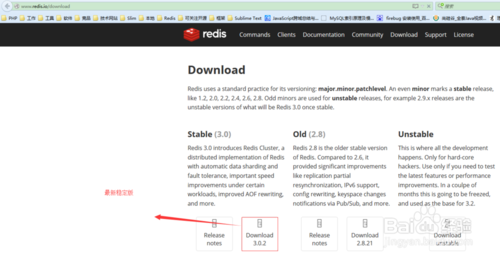
* Redis源码获取

1、进入Redis官网获取Redis最新稳定版下载地址

2、通过wget命令下载 Redis 源代码。



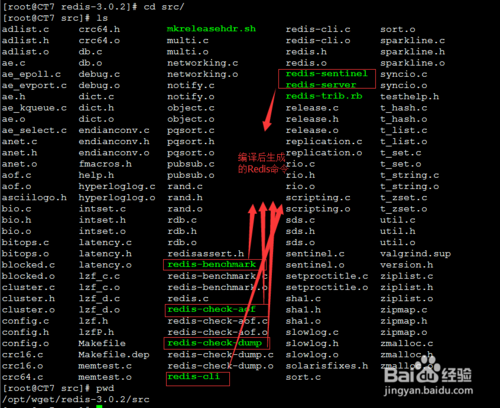


* Redis编译

1、通过tar -xvf redis-3.0.2.tar.gz命令解压下载Redis源码压缩包redis-3.0.2.tar.gz；

2、编译Redis。通过cd redis-3.0.2/进入Redis源码目录内，执行make编译Redis；

注意：make命令执行完成编译后，会在src目录下生成6个可执行文件，分别是redis-server、redis-cli、redis-benchmark、redis-check-aof、redis-check-dump、redis-sentinel。

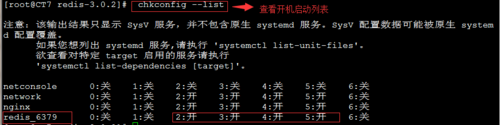


* Redis安装配置

1、安装Redis，执行make install。会将make编译生成的可执行文件拷贝到/usr/local/bin目录下；

2、执行./utils/install\_server.sh配置Redis配置之后Redis能随系统启动。





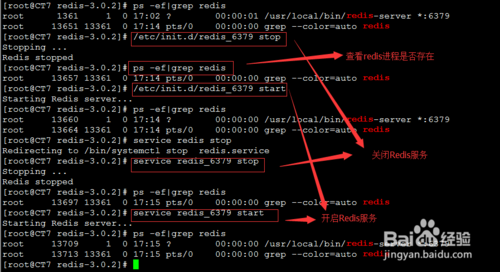
* 4

Redis服务查看、开启、关闭

1、通过ps -ef|grep redis命令查看Redis进程；

2、开启Redis服务操作通过/etc/init.d/redis\_6379 start命令，也可通过（service redis\_6379 start）；

3、关闭Redis服务操作通过/etc/init.d/redis\_6379 stop命令，也可通过（service redis\_6379 stop）；



END

1.  [**MySQL**](http://lib.csdn.net/base/mysql)+Memcached[**架构**](http://lib.csdn.net/base/architecture)的问题

Memcached采用客户端-服务器的架构，客户端和服务器端的通讯使用自定义的协议标准，只要满足协议格式要求，客户端Library可以用任何语言实现。

Memcached服务器使用基于Slab的内存管理方式，有利于减少内存碎片和频繁分配销毁内存所带来的开销。各个Slab按需动态分配一个page的内存（和4Kpage的概念不同，这里默认page为1M），page内部按照不同slab class的尺寸再划分为内存chunk供服务器存储KV键值对使用（slab机制相当于内存池机制, 实现从[**操作系统**](http://lib.csdn.net/base/operatingsystem)分配一大块内存, 然后 memcached 自己管理这块内存, 负责分配与回收。）

关于memcached的内存分配机制：[Memcached 内存分配机制介绍](http://blog.csdn.net/hguisu/article/details/7353482)

      实际MySQL是适合进行海量数据存储的，通过Memcached将热点数据加载到cache，加速访问，很多公司都曾经使用过这样的架构，但随着业务数据量的不断增加，和访问量的持续增长，我们遇到了很多问题：

　　1.MySQL需要不断进行拆库拆表，Memcached也需不断跟着扩容，扩容和维护工作占据大量开发时间。

　　2.Memcached与MySQL[**数据库**](http://lib.csdn.net/base/mysql)数据一致性问题。

　　3.Memcached数据命中率低或down机，大量访问直接穿透到DB，MySQL无法支撑。

　　4.跨机房cache同步问题。

       关于memcached问题：[Memcache存储大数据的问题](http://blog.csdn.net/hguisu/article/details/6163621)

　　众多NoSQL百花齐放，如何选择

　　最近几年，业界不断涌现出很多各种各样的NoSQL产品，那么如何才能正确地使用好这些产品，最大化地发挥其长处，是我们需要深入研究和思考的问题，实际归根结底最重要的是了解这些产品的定位，并且了解到每款产品的tradeoffs，在实际应用中做到扬长避短，总体上这些NoSQL主要用于解决以下几种问题

　　1.少量数据存储，高速读写访问。此类产品通过数据全部in-momery 的方式来保证高速访问，同时提供数据落地的功能，实际这正是[**Redis**](http://lib.csdn.net/base/redis)最主要的适用场景。

　　2.海量数据存储，分布式系统支持，数据一致性保证，方便的集群节点添加/删除。

　　3.这方面最具代表性的是dynamo和bigtable 2篇论文所阐述的思路。前者是一个完全无中心的设计，节点之间通过gossip方式传递集群信息，数据保证最终一致性，后者是一个中心化的方案设计，通过类似一个分布式锁服务来保证强一致性,数据写入先写[内存](http://product.it168.com/list/b/0205_1.shtml)和redo log，然后定期compat归并到磁盘上，将随机写优化为顺序写，提高写入性能。

　　4.Schema free，auto-sharding等。比如目前常见的一些文档数据库都是支持schema-free的，直接存储json格式数据，并且支持auto-sharding等功能，比如[**MongoDB**](http://lib.csdn.net/base/mongodb)。

　　面对这些不同类型的NoSQL产品,我们需要根据我们的业务场景选择最合适的产品。

       Redis最适合所有数据in-momory的场景，虽然Redis也提供持久化功能，但实际更多的是一个disk-backed的功能，跟传统意义上的持久化有比较大的差别，那么可能大家就会有疑问，似乎Redis更像一个加强版的Memcached，那么何时使用Memcached,何时使用Redis呢?

memcache和redis的比较：

* 性能方面：没有必要过多的关心性能，因为二者的性能都已经足够高了。由于Redis只使用单核，而Memcached可以使用多核，所以在比较上，平均每一个核上Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis，虽然Redis最近也在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色。说了这么多，结论是，无论你使用哪一个，每秒处理请求的次数都不会成为瓶颈。（比如瓶颈可能会在网卡）
* 内存使用效率：使用简单的key-value存储的话，Memcached的内存利用率更高，而如果Redis采用hash结构来做key-value存储，由于其组合式的压缩，其内存利用率会高于Memcached。当然，这和你的应用场景和数据特性有关。
* 数据持久化：如果你对数据持久化和数据同步有所要求，那么推荐你选择Redis，因为这两个特性Memcached都不具备。即使你只是希望在升级或者重启系统后缓存数据不会丢失，选择Redis也是明智的。
* 数据结构:当然，最后还得说到你的具体应用需求。Redis相比Memcached来说，拥有更多的数据结构和并支持更丰富的数据操作，通常在Memcached里，你需要将数据拿到客户端来进行类似的修改再set回去。这大大增加了网络IO的次数和数据体积。在Redis中，这些复杂的操作通常和一般的GET/SET一样高效。所以，如果你需要缓存能够支持更复杂的结构和操作，那么Redis会是不错的选择。
* 网络IO模型方面：Memcached是多线程，分为监听线程、worker线程，引入锁，带来了性能损耗。Redis使用单线程的IO复用模型，将速度优势发挥到最大，也提供了较简单的计算功能
* 内存管理方面：Memcached使用预分配的内存池的方式，带来一定程度的空间浪费 并且在内存仍然有很大空间时，新的数据也可能会被剔除，而Redis使用现场申请内存的方式来存储数据，不会剔除任何非临时数据 Redis更适合作为存储而不是cache
* 数据的一致性方面：Memcached提供了cas命令来保证.而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

    如果简单地比较Redis与Memcached的区别，大多数都会得到以下观点：

     1 、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等[**数据结构**](http://lib.csdn.net/base/datastructure)的存储。

     2 、Redis支持数据的备份，即master-slave模式的数据备份。

     3 、Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

     4、Redis可以实现主从复制，实现故障恢复。

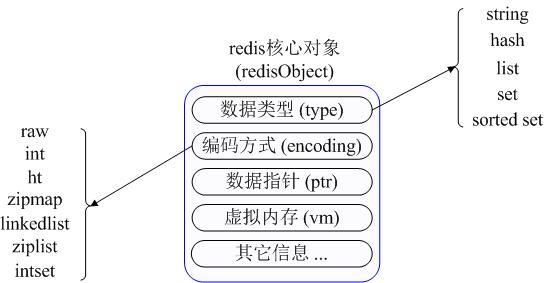
     5、Redis的Sharding技术： 很容易将数据分布到多个Redis实例中

2.  Redis常用数据类型

Redis最为常用的数据类型主要有以下：

* String
* Hash
* List
* Set
* Sorted set
* pub/sub
* Transactions

在具体描述这几种数据类型之前，我们先通过一张图了解下Redis内部内存管理中是如何描述这些不同数据类型的：



         首先Redis内部使用一个redisObject对象来表示所有的key和value,redisObject最主要的信息如上图所示：

         type代表一个value对象具体是何种数据类型，

         encoding是不同数据类型在redis内部的存储方式，

         比如：type=string代表value存储的是一个普通字符串，那么对应的encoding可以是raw或者是int,如果是int则代表实际redis内部是按数值型类存储和表示这个字符串的，当然前提是这个字符串本身可以用数值表示，比如:"123" "456"这样的字符串。

       这里需要特殊说明一下vm字段，只有打开了Redis的虚拟内存功能，此字段才会真正的分配内存，该功能默认是关闭状态的，该功能会在后面具体描述。通过上图我们可以发现Redis使用redisObject来表示所有的key/value数据是比较浪费内存的，当然这些内存管理成本的付出主要也是为了给Redis不同数据类型提供一个统一的管理接口，实际作者也提供了多种方法帮助我们尽量节省内存使用，我们随后会具体讨论。

## 

3.  各种数据类型应用和实现方式

下面我们先来逐一的分析下这7种数据类型的使用和内部实现方式:

### String:

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。

应用场景：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.即可以完全实现目前 Memcached 的功能，并且效率更高。还可以享受Redis的定时持久化，操作日志及 Replication等功能。除了提供与 Memcached 一样的get、set、incr、decr 等操作外，Redis还提供了下面一些操作：

* + - 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

实现方式：String在redis内部存储默认就是一个字符串，被redisObject所引用，当遇到incr,decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。

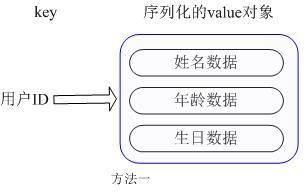
### Hash

常用命令：hget,hset,hgetall 等。

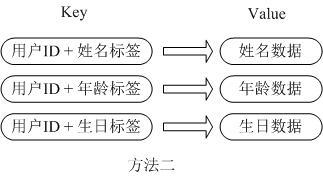
应用场景：在Memcached中，我们经常将一些结构化的信息打包成HashMap，在客户端序列化后存储为一个字符串的值，比如用户的昵称、年龄、性别、积分等，这时候在需要修改其中某一项时，通常需要将所有值取出反序列化后，修改某一项的值，再序列化存储回去。这样不仅增大了开销，也不适用于一些可能并发操作的场合（比如两个并发的操作都需要修改积分）。而Redis的Hash结构可以使你像在数据库中Update一个属性一样只修改某一项属性值。

        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：

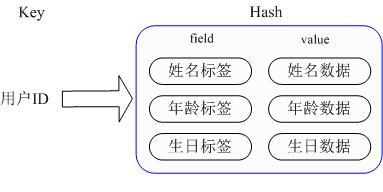


第一种方式将用户ID作为查找key,把其他信息封装成一个对象以序列化的方式存储，这种方式的缺点是，增加了序列化/反序列化的开销，并且在需要修改其中一项信息时，需要把整个对象取回，并且修改操作需要对并发进行保护，引入CAS等复杂问题。



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。

那么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。很好的解决了问题。

这里同时需要注意，Redis提供了接口(hgetall)可以直接取到全部的属性数据,但是如果内部Map的成员很多，那么涉及到遍历整个内部Map的操作，由于Redis单线程模型的缘故，这个遍历操作可能会比较耗时，而另其它客户端的请求完全不响应，这点需要格外注意。

实现方式：

上面已经说到Redis Hash对应Value内部实际就是一个HashMap，实际这里会有2种不同实现，这个Hash的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，对应的value redisObject的encoding为zipmap,当成员数量增大时会自动转成真正的HashMap,此时encoding为ht。

### List

常用命令：lpush,rpush,lpop,rpop,lrange等。

应用场景：

Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现。

Lists 就是链表，相信略有数据结构知识的人都应该能理解其结构。使用Lists结构，我们可以轻松地实现最新消息排行等功能。Lists的另一个应用就是消息队列，  
可以利用Lists的PUSH操作，将任务存在Lists中，然后工作线程再用POP操作将任务取出进行执行。Redis还提供了操作Lists中某一段的api，你可以直接查询，删除Lists中某一段的元素。

实现方式：

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。

### Set

常用命令：

sadd,spop,smembers,sunion 等。

应用场景：

Redis 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。

Sets 集合的概念就是一堆不重复值的组合。利用Redis提供的Sets数据结构，可以存储一些集合性的数据，比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。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，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。

实现方式：

set 的内部实现是一个 value永远为null的HashMap，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。

### Sorted Set

常用命令：

zadd,zrange,zrem,zcard等

使用场景：

Redis 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，比如twitter 的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。

另外还可以用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。

实现方式：

Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

### Pub/Sub

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在Redis中，你可以设定对某一个key值进行消息发布及消息订阅，当一个key值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

### Transactions

谁说NoSQL都不支持事务，虽然Redis的Transactions提供的并不是严格的ACID的事务（比如一串用EXEC提交执行的命令，在执行中服务器宕机，那么会有一部分命令执行了，剩下的没执行），但是这个Transactions还是提供了基本的命令打包执行的功能（在服务器不出问题的情况下，可以保证一连串的命令是顺序在一起执行的，中间有会有其它客户端命令插进来执行）。Redis还提供了一个Watch功能，你可以对一个key进行Watch，然后再执行Transactions，在这过程中，如果这个Watched的值进行了修改，那么这个Transactions会发现并拒绝执行。

## 

4.  Redis实际应用场景

        Redis在很多方面与其他数据库解决方案不同：它使用内存提供主存储支持，而仅使用硬盘做持久性的存储；它的数据模型非常独特，用的是单线程。另一个大区别在于，你可以在开发环境中使用Redis的功能，但却不需要转到Redis。

转向Redis当然也是可取的，许多开发者从一开始就把Redis作为首选数据库；但设想如果你的开发环境已经搭建好，应用已经在上面运行了，那么更换数据库框架显然不那么容易。另外在一些需要大容量数据集的应用，Redis也并不适合，因为它的数据集不会超过系统可用的内存。所以如果你有[**大数据**](http://lib.csdn.net/base/spark)应用，而且主要是读取访问模式，那么Redis并不是正确的选择。

        然而我喜欢Redis的一点就是你可以把它融入到你的系统中来，这就能够解决很多问题，比如那些你现有的数据库处理起来感到缓慢的任务。这些你就可以通过Redis来进行优化，或者为应用创建些新的功能。在本文中，我就想探讨一些怎样将Redis加入到现有的环境中，并利用它的原语命令等功能来解决 传统环境中碰到的一些常见问题。在这些例子中，Redis都不是作为首选数据库。

### 1、显示最新的项目列表

下面这个语句常用来显示最新项目，随着数据多了，查询毫无疑问会越来越慢。

**[sql]** [view plain](http://blog.csdn.net/hguisu/article/details/8836819) [copy](http://blog.csdn.net/hguisu/article/details/8836819)

[print?](http://blog.csdn.net/hguisu/article/details/8836819)

1. SELECT \* FROM foo WHERE ... ORDER BY time DESC LIMIT 10

        在Web应用中，“列出最新的回复”之类的查询非常普遍，这通常会带来可扩展性问题。这令人沮丧，因为项目本来就是按这个顺序被创建的，但要输出这个顺序却不得不进行排序操作。

        类似的问题就可以用Redis来解决。比如说，我们的一个Web应用想要列出用户贴出的最新20条评论。在最新的评论边上我们有一个“显示全部”的链接，点击后就可以获得更多的评论。

        我们假设数据库中的每条评论都有一个唯一的递增的ID字段。

        我们可以使用分页来制作主页和评论页，使用Redis的模板，每次新评论发表时，我们会将它的ID添加到一个Redis列表：

**[plain]** [view plain](http://blog.csdn.net/hguisu/article/details/8836819) [copy](http://blog.csdn.net/hguisu/article/details/8836819)

[print?](http://blog.csdn.net/hguisu/article/details/8836819)

1. LPUSH latest.comments <ID>

       我们将列表裁剪为指定长度，因此Redis只需要保存最新的5000条评论：

       LTRIM latest.comments 0 5000

      每次我们需要获取最新评论的项目范围时，我们调用一个函数来完成（使用伪代码）：

**[plain]** [view plain](http://blog.csdn.net/hguisu/article/details/8836819) [copy](http://blog.csdn.net/hguisu/article/details/8836819)

[print?](http://blog.csdn.net/hguisu/article/details/8836819)

1. FUNCTION get\_latest\_comments(start, num\_items):
2. id\_list = redis.lrange("latest.comments",start,start+num\_items - 1)
3. IF id\_list.length < num\_items
4. id\_list = SQL\_DB("SELECT ... ORDER BY time LIMIT ...")
5. END
6. RETURN id\_list
7. END

      这里我们做的很简单。在Redis中我们的最新ID使用了常驻缓存，这是一直更新的。但是我们做了限制不能超过5000个ID，因此我们的获取ID函数会一直询问Redis。只有在start/count参数超出了这个范围的时候，才需要去访问数据库。

        我们的系统不会像传统方式那样“刷新”缓存，Redis实例中的信息永远是一致的。SQL数据库（或是硬盘上的其他类型数据库）只是在用户需要获取“很远”的数据时才会被触发，而主页或第一个评论页是不会麻烦到硬盘上的数据库了。

### 2、删除与过滤

      我们可以使用LREM来删除评论。如果删除操作非常少，另一个选择是直接跳过评论条目的入口，报告说该评论已经不存在。

       有些时候你想要给不同的列表附加上不同的过滤器。如果过滤器的数量受到限制，你可以简单的为每个不同的过滤器使用不同的Redis列表。毕竟每个列表只有5000条项目，但Redis却能够使用非常少的内存来处理几百万条项目。

### 3、排行榜相关

      另一个很普遍的需求是各种数据库的数据并非存储在内存中，因此在按得分排序以及实时更新这些几乎每秒钟都需要更新的功能上数据库的性能不够理想。

      典型的比如那些在线游戏的排行榜，比如一个Facebook的游戏，根据得分你通常想要：

         - 列出前100名高分选手

         - 列出某用户当前的全球排名

      这些操作对于Redis来说小菜一碟，即使你有几百万个用户，每分钟都会有几百万个新的得分。

      模式是这样的，每次获得新得分时，我们用这样的代码：

      ZADD leaderboard  <score>  <username>

     你可能用userID来取代username，这取决于你是怎么设计的。

      得到前100名高分用户很简单：ZREVRANGE leaderboard 0 99。

      用户的全球排名也相似，只需要：ZRANK leaderboard <username>。

### 4、按照用户投票和时间排序

      排行榜的一种常见变体模式就像Reddit或Hacker News用的那样，新闻按照类似下面的公式根据得分来排序：

       score = points / time^alpha

      因此用户的投票会相应的把新闻挖出来，但时间会按照一定的指数将新闻埋下去。下面是我们的模式，当然[**算法**](http://lib.csdn.net/base/datastructure)由你决定。

      模式是这样的，开始时先观察那些可能是最新的项目，例如首页上的1000条新闻都是候选者，因此我们先忽视掉其他的，这实现起来很简单。

      每次新的新闻贴上来后，我们将ID添加到列表中，使用LPUSH + LTRIM，确保只取出最新的1000条项目。

      有一项后台任务获取这个列表，并且持续的计算这1000条新闻中每条新闻的最终得分。计算结果由ZADD命令按照新的顺序填充生成列表，老新闻则被清除。这里的关键思路是排序工作是由后台任务来完成的。

### 5、处理过期项目

      另一种常用的项目排序是按照时间排序。我们使用unix时间作为得分即可。

      模式如下：

       - 每次有新项目添加到我们的非Redis数据库时，我们把它加入到排序集合中。这时我们用的是时间属性，current\_time和time\_to\_live。

       - 另一项后台任务使用ZRANGE…SCORES查询排序集合，取出最新的10个项目。如果发现unix时间已经过期，则在数据库中删除条目。

### 6、计数

       Redis是一个很好的计数器，这要感谢INCRBY和其他相似命令。

       我相信你曾许多次想要给数据库加上新的计数器，用来获取统计或显示新信息，但是最后却由于写入敏感而不得不放弃它们。

       好了，现在使用Redis就不需要再担心了。有了原子递增（atomic increment），你可以放心的加上各种计数，用GETSET重置，或者是让它们过期。

       例如这样操作：

         INCR user:<id> EXPIRE

         user:<id> 60

       你可以计算出最近用户在页面间停顿不超过60秒的页面浏览量，当计数达到比如20时，就可以显示出某些条幅提示，或是其它你想显示的东西。

### 7、特定时间内的特定项目

        另一项对于其他数据库很难，但Redis做起来却轻而易举的事就是统计在某段特点时间里有多少特定用户访问了某个特定资源。比如我想要知道某些特定的注册用户或IP地址，他们到底有多少访问了某篇文章。

      每次我获得一次新的页面浏览时我只需要这样做：

       SADD page:day1:<page\_id> <user\_id>

      当然你可能想用unix时间替换day1，比如time()-(time()%3600\*24)等等。

      想知道特定用户的数量吗？只需要使用SCARD page:day1:<page\_id>。

       需要[**测试**](http://lib.csdn.net/base/softwaretest)某个特定用户是否访问了这个页面？SISMEMBER page:day1:<page\_id>。

### 8、实时分析正在发生的情况，用于数据统计与防止垃圾邮件等

        我们只做了几个例子，但如果你研究Redis的命令集，并且组合一下，就能获得大量的实时分析方法，有效而且非常省力。使用Redis原语命令，更容易实施垃圾邮件过滤系统或其他实时跟踪系统。

### 9、Pub/Sub

       Redis的Pub/Sub非常非常简单，运行稳定并且快速。支持模式匹配，能够实时订阅与取消频道。

### 10、队列

        你应该已经注意到像list push和list pop这样的Redis命令能够很方便的执行队列操作了，但能做的可不止这些：比如Redis还有list pop的变体命令，能够在列表为空时阻塞队列。

       现代的互联网应用大量地使用了消息队列（Messaging）。消息队列不仅被用于系统内部组件之间的通信，同时也被用于系统跟其它服务之间的交互。消息队列的使用可以增加系统的可扩展性、灵活性和用户体验。非基于消息队列的系统，其运行速度取决于系统中最慢的组件的速度（注：短板效应）。而基于消息队列可以将系统中各组件解除耦合，这样系统就不再受最慢组件的束缚，各组件可以异步运行从而得以更快的速度完成各自的工作。

    此外，当服务器处在高并发操作的时候，比如频繁地写入日志文件。可以利用消息队列实现异步处理。从而实现高性能的并发操作。

### 11、缓存

        Redis的缓存部分值得写一篇新文章，我这里只是简单的说一下。Redis能够替代memcached，让你的缓存从只能存储数据变得能够更新数据，因此你不再需要每次都重新生成数据了。

此部分内容的原文地址：http://antirez.com/post/take-advantage-of-redis-adding-it-to-your-stack.html

5.  国内外三个不同领域巨头分享的Redis实战经验及使用场景

     随着应用对高性能需求的增加，NoSQL逐渐在各大名企的系统架构中生根发芽。这里我们将为大家分享社交巨头新浪微博、传媒巨头Viacom及图片分享领域佼佼者Pinterest带来的Redis实践，首先我们看新浪微博 [@启盼cobain](http://weibo.com/1319158547/AcmcMuPM5)的Redis实战经验分享：

### 一、新浪微博：史上最大的Redis集群

Tape is Dead，Disk is Tape，Flash is Disk，RAM Locality is King. — Jim Gray

Redis不是比较成熟的memcache或者Mysql的替代品，是对于大型互联网类应用在架构上很好的补充。现在有越来越多的应用也在纷纷基于Redis做架构的改造。首先简单公布一下Redis平台实际情况：

* 2200+亿 commands/day 5000亿Read/day 500亿Write/day
* 18TB+ Memory
* 500+ Servers in 6 IDC 2000+instances

应该是国内外比较大的Redis使用平台，今天主要从应用角度谈谈Redis服务平台。

#### Redis使用场景

1.Counting（计数）

计数的应用在另外一篇文章里较详细的描述，计数场景的优化 <http://www.xdata.me/?p=262>这里就不多加描述了。

可以预见的是，有很多同学认为把计数全部存在内存中成本非常高，我在这里用个图表来表达下我的观点：

很多情况大家都会设想纯使用内存的方案会很有很高成本，但实际情况往往会有一些不一样：

* COST，对于有一定吞吐需求的应用来说，肯定会单独申请DB、Cache资源，很多担心DB写入性能的同学还会主动将DB更新记入异步队列，而这三块的资源的利用率一般都不会太高。资源算下来，你惊异的发现：反而纯内存的方案会更精简！
* KISS原则，这对于开发是非常友好的，我只需要建立一套连接池，不用担心数据一致性的维护，不用维护异步队列。
* Cache穿透风险，如果后端使用DB，肯定不会提供很高的吞吐能力，cache宕机如果没有妥善处理，那就悲剧了。
* 大多数的起始存储需求，容量较小。

2.Reverse cache（反向cache）

面对微博常常出现的热点，如最近出现了较为火爆的短链，短时间有数以万计的人点击、跳转，而这里会常常涌现一些需求，比如我们向快速在跳转时判定用户等级，是否有一些账号绑定，性别爱好什么的，已给其展示不同的内容或者信息。

普通采用memcache+Mysql的解决方案，当调用id合法的情况下，可支撑较大的吞吐。但当调用id不可控，有较多垃圾用户调用时，由于memcache未有命中，会大量的穿透至Mysql服务器，瞬间造成连接数疯长，整体吞吐量降低，响应时间变慢。

这里我们可以用redis记录全量的用户判定信息，如string key:uid int:type，做一次反向的cache，当用户在redis快速获取自己等级等信息后，再去Mc+Mysql层去获取全量信息。如图：

当然这也不是最优化的场景，如用Redis做bloomfilter，可能更加省用内存。

3.Top 10 list

产品运营总会让你展示最近、最热、点击率最高、活跃度最高等等条件的top list。很多更新较频繁的列表如果使用MC+MySQL维护的话缓存失效的可能性会比较大，鉴于占用内存较小的情况，使用Redis做存储也是相当不错的。

4.Last Index

用户最近访问记录也是redis list的很好应用场景，lpush lpop自动过期老的登陆记录，对于开发来说还是非常友好的。

5.Relation List/Message Queue

这里把两个功能放在最后，因为这两个功能在现实问题当中遇到了一些困难，但在一定阶段也确实解决了我们很多的问题，故在这里只做说明。

Message Queue就是通过list的lpop及lpush接口进行队列的写入和消费，由于本身性能较好也能解决大部分问题。

6.Fast transaction with Lua

Redis 的Lua的功能扩展实际给Redis带来了更多的应用场景，你可以编写若干command组合作为一个小型的非阻塞事务或者更新逻辑，如：在收到message推送时，同时1.给自己的增加一个未读的对话 2.给自己的私信增加一个未读消息 3.最后给发送人回执一个完成推送消息，这一层逻辑完全可以在Redis Server端实现。

但是，需要注意的是Redis会将lua script的全部内容记录在aof和传送给slave，这也将是对磁盘，网卡一个不小的开销。

7.Instead of Memcache

1. 很多测试和应用均已证明，
2. 在性能方面Redis并没有落后memcache多少，而单线程的模型给Redis反而带来了很强的扩展性。
3. 在很多场景下，Redis对同一份数据的内存开销是小于memcache的slab分配的。
4. Redis提供的数据同步功能，其实是对cache的一个强有力功能扩展。

#### Redis使用的重要点

1.rdb/aof Backup!

我们线上的Redis 95%以上是承担后端存储功能的，我们不仅用作cache，而更为一种k-v存储，他完全替代了后端的存储服务（MySQL），故其数据是非常重要的，如果出现数据污染和丢失，误操作等情况，将是难以恢复的。所以备份是非常必要的！为此，我们有共享的hdfs资源作为我们的备份池，希望能随时可以还原业务所需数据。

2.Small item & Small instance!

由于Redis单线程（严格意义上不是单线程，但认为对request的处理是单线程的）的模型，大的数据结构list,sorted set,hash set的批量处理就意味着其他请求的等待，故使用Redis的复杂数据结构一定要控制其单key-struct的大小。

另外，Redis单实例的内存容量也应该有严格的限制。单实例内存容量较大后，直接带来的问题就是故障恢复或者Rebuild从库的时候时间较长，而更糟糕的是，Redis rewrite aof和save rdb时，将会带来非常大且长的系统压力，并占用额外内存，很可能导致系统内存不足等严重影响性能的线上故障。我们线上96G/128G内存服务器不建议单实例容量大于20/30G。

3.Been Available!

业界资料和使用比较多的是Redis sentinel（哨兵）

<http://www.huangz.me/en/latest/storage/redis_code_analysis/sentinel.html>

<http://qiita.com/wellflat/items/8935016fdee25d4866d9>

2000行C实现了服务器状态检测，自动故障转移等功能。

但由于自身实际架构往往会复杂，或者考虑的角度比较多，为此 [@许琦eryk](http://weibo.com/eryk86)和我一同做了hypnos项目。

hypnos是神话中的睡神，字面意思也是希望我们工程师无需在休息时间处理任何故障。:-)

其工作原理示意如下：

Talk is cheap, show me your code! 稍后将单独写篇博客细致讲下Hypnos的实现。

4.In Memory or not?

发现一种情况，开发在沟通后端资源设计的时候，常常因为习惯使用和错误了解产品定位等原因，而忽视了对真实使用用户的评估。也许这是一份历史数据，只有最近一天的数据才有人进行访问，而把历史数据的容量和最近一天请求量都抛给内存类的存储现实是非常不合理的。

所以当你在究竟使用什么样的数据结构存储的时候，请务必先进行成本衡量，有多少数据是需要存储在内存中的？有多少数据是对用户真正有意义的。因为这其实对后端资源的设计是至关重要的，1G的数据容量和1T的数据容量对于设计思路是完全不一样的

#### Plans in future?

1.slave sync改造

全部改造线上master-slave数据同步机制，这一点我们借鉴了MySQL Replication的思路，使用rdb+aof+pos作为数据同步的依据，这里简要说明为什么官方提供的psync没有很好的满足我们的需求：

假设A有两个从库B及C，及 A `— B&C，这时我们发现master A服务器有宕机隐患需要重启或者A节点直接宕机，需要切换B为新的主库，如果A、B、C不共享rdb及aof信息，C在作为B的从库时，仍会清除自身数据，因为C节点只记录了和A节点的同步状况。

故我们需要有一种将A`–B&C 结构切换切换为A`–B`–C结构的同步机制，psync虽然支持断点续传，但仍无法支持master故障的平滑切换。

实际上我们已经在我们定制的Redis计数服务上使用了如上功能的同步，效果非常好，解决了运维负担，但仍需向所有Redis服务推广，如果可能我们也会向官方Redis提出相关sync slave的改进。

2.更适合redis的name-system Or proxy

细心的同学发现我们除了使用DNS作为命名系统，也在zookeeper中有一份记录，为什么不让用户直接访问一个系统，zk或者DNS选择其一呢？

其实还是很简单，命名系统是个非常重要的组件，而dns是一套比较完善的命名系统，我们为此做了很多改进和试错，zk的实现还是相对复杂，我们还没有较强的把控粒度。我们也在思考用什么做命名系统更符合我们需求。

3.后端数据存储

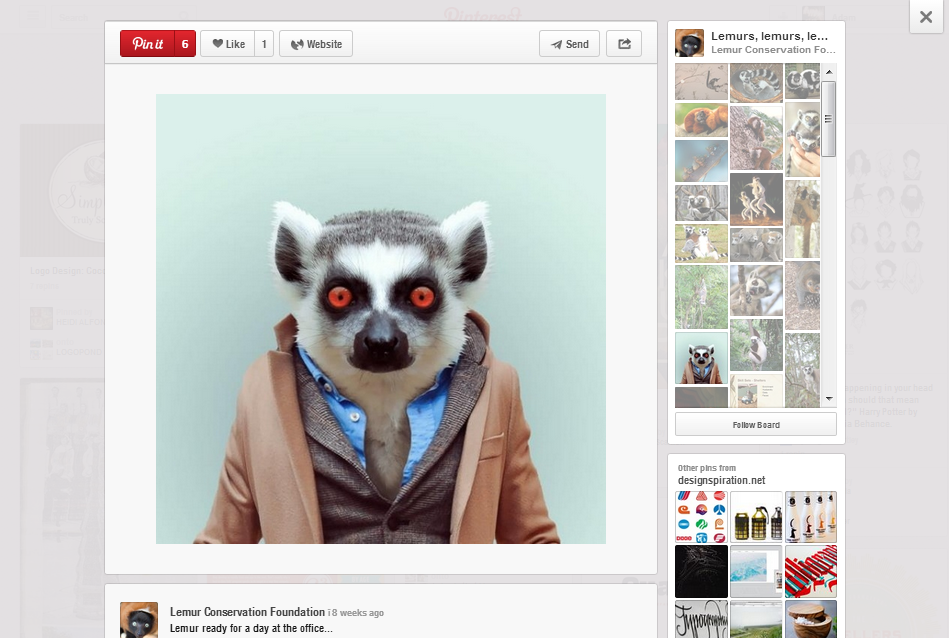
大内存的使用肯定是一个重要的成本优化方向，flash盘及分布式的存储也在我们未来计划之中。（原文链接： [Largest Redis Clusters Ever](http://www.xdata.me/?p=301)）

### 二、Pinterest：Reids维护上百亿的相关性

      Pinterest已经成为硅谷最疯故事之一，在2012年，他们基于PC的业务增加1047%，移动端采用增加1698%， [该年3月其独立访问数量更飙升至533亿](http://www.forbes.com/sites/jjcolao/2013/05/08/why-is-pinterest-a-2-5-billion-company-an-early-investor-explains/)。在Pinterest，人们关注的事物以百亿记——每个用户界面都会查询某个board或者是用户是否关注的行为促成了异常复杂的工程问题。这也让Redis获得了用武之地。经过数年的发展，Pinterest已经成为媒体、社交等多个领域的佼佼者，其辉煌战绩如下：

* 获得的推荐流量高于Google+、YouTube及LinkedIn三者的总和
* 与Facebook及Twitter一起成为最流行的三大社交网络
* 参考Pinterest进行购买的用户比其它网站更高（ [更多详情](http://corp.wishpond.com/pinterest-marketing-resources-for-business/15-facts-you-need-to-know-about-pinterest/)）

如您所想，基于其独立访问数，Pinterest的高规模促成了一个非常高的IT基础设施需求。



通过缓存来优化用户体验

近日，Pinterest工程经理Abhi Khune对其公司的用户体验需求及Redis的使用经验 [进行了分享](http://engineering.pinterest.com/post/55272557617/building-a-follower-model-from-scratch)。即使是滋生的应用程序打造者，在分析网站的细节之前也不会理解这些特性，因此先大致的理解一下使用场景：首先，为每个粉丝进行提及到的预检查；其次，UI将准确的显示用户的粉丝及关注列表分页。高效的执行这些操作，每次点击都需要非常高的性能架构。

不能免俗，Pinterest的软件工程师及架构师已经使用了MySQL及memcache，但是缓存解决方案仍然达到了他们的瓶颈；因此为了拥有更好的用户体验，缓存必须被扩充。而在实际操作过程中，工程团队已然发现缓存只有当用户sub-graph已经在缓存中时才会起到作用。因此。任何使用这个系统的人都需要被缓存，这就导致了整个图的缓存。同时，最常见的查询“用户A是否关注了用户B”的答案经常是否定的，然而这却被作为了缓存丢失，从而促成一个数据库查询，因此他们需要一个新的方法来扩展缓存。最终，他们团队决定使用Redis来存储整个图，用以服务众多的列表。

使用Redis存储大量的Pinterest列表

Pinterest使用了Redis作为解决方案，并将性能推至了内存数据库等级，为用户保存多种类型列表：

* 关注者列表
* 你所关注的board列表
* 粉丝列表
* 关注你board的用户列表
* 某个用户中board中你没有关注的列表
* 每个board的关注者及非关注者

Redis为其7000万用户存储了以上的所有列表，本质上讲可以说是储存了所有粉丝图，通过用户ID分片。鉴于你可以通过类型来查看以上列表的数据，分析概要信息被用看起来更像事务的系统储存及访问。Pinterest当下的用户like被限制为10万，初略进行统计：如果每个用户关注25个board，将会在用户及board间产生17.5亿的关系。同时更加重要的是，这些关系随着系统的使用每天都会增加。

Pinterest的Reids架构及运营

通过Pinterest的一个创始人了解到，Pinterest开始使用[**Python**](http://lib.csdn.net/base/python)及订制的Django编写应用程序，并一直持续到其拥有1800万用户级日410TB用户数据的时候。虽然使用了多个存储对数据进行储存，工程师根据用户id使用了8192个虚拟分片，每个分片都运行在一个Redis DB之上，同时1个Redis实例将运行多个Redis DB。为了对CPU核心的充分使用，同一台主机上同时使用多线程和单线程Redis实例。

鉴于整个数据集运行在内存当中，Redis在Amazon EBS上对每秒传输进来的写入都会进行持久化。扩展主要通过两个方面进行：第一，保持50%的利用率，通过主从转换，机器上运行的Redis实例一半会转译到一个新机器上；第二，扩展节点和分片。整个Redis集群都会使用一个主从配置，从部分将被当做一个热备份。一旦主节点失败，从部分会立刻完成主的转换，同时一个新的从部分将会被添加，ZooKeeper将完成整个过程。同时他们每个小时都会在Amazon S3上运行BGsave做更持久的储存——这项Reids操作会在后端进行，之后Pinterest会使用这些数据做MapReduce和分析作业。（更多内容见原文）

### 三、Viacom：Redis在系统中的用例盘点

Viacom是全球最大的传媒集体之一，同时也遭遇了当下最大的数据难题之一：如何处理日益剧增的动态视频内容。

着眼这一挑战的上升趋势，我们会发现：[2010年世界上所有数据体积达到ZB级](http://blogs.vmware.com/vfabric/2013/03/why-every-database-must-be-broken-soon.html)，而单单2012这一年，互联网产生的数据就增加了2.8个ZB，其中大部分的数据都是非结构化的，包括了视频和图片。

覆盖MVN（以前称为MTV Networks、Paramount及BET），Viacom是个名副其实的传媒巨头，支持众多人气站点，其中包括The Daily Show、osh.0、South Park Studios、GameTrailers.com等。作为媒体公司，这些网站上的文档、图片、视频短片都在无时无刻的更新。长话短说，下面就进入Viacom高级架构师Michael Venezia 分享的Redis实践：

Viacom的网站架构背景

对于Viacom，横跨多个站点传播内容让必须专注于规模的需求，同时为了将内容竟可能快的传播到相应用户，他们还必须聚焦内容之间的关系。然而即使The Daily Show、Nickelodeon、Spike或者是VH1 这些单独的网站上，日平均PV都可以达到千万，峰值时流量更会达到平均值的20-30倍。同时基于对实时的需求，动态的规模及速度已成为架构的基础之一。

除去动态规模之外，服务还必须基于用户正在浏览的视频或者是地理位置来推测用户的喜好。比如说，某个页面可能会将一个独立的视频片段与本地的促销，视频系列的额外部分，甚至是相关视频联系起来。为了能让用户能在网站上停留更长的时间，他们建立了一个能基于详细元数据自动建立页面的软件引擎，这个引擎可以根据用户当下兴趣推荐额外的内容。鉴于用于兴趣的随时改变，数据的类型非常广泛——类似graph-like，实际上做的是大量的join。

这样做有利于减少类似视频的大体积文件副本数，比如数据存储中一个独立的记录是Southpark片段“Cartman gets an Anal Probe”，这个片段可能也会出现在德语的网站上。虽然视频是一样的，但是英语用户搜索的可能就是另一个不同的词语。元数据的副本转换成搜索结果，并指向相同的视频。因此在美国用户搜索真实标题的情况下，德国浏览者可能会使用转译的标题——德国网站上的“Cartman und die Analsonde”。

这些元数据覆盖了其它记录或者是对象，同时还可以根据使用环境来改变内容，通过不同的规则集来限制不同地理位置或者是设备请求的内容。

Viacom的实现方法

尽管许多机构通过使用ORM及传统关系型数据库来解决这个问题，Viacom却使用了一个迥然不同的方法。

本质上，他们完全承担不了对数据库的直接访问。首先，他们处理的大部分都是流数据，他们偏向于使用Akamai从地理上来分配内容。其次，基于页面的复杂性可能会取上万个对象。取如此多的数据显然会影响到性能，因此JSON在1个数据服务中投入了使用。当然，这些JSON对象的缓存将直接影响到网站性能。同时，当内容或者是内容之间的关系发生改变时，缓存还需要动态的进行更新。

Viacom依靠对象基元和超类解决这个问题，继续以South Park为例：一个私有的“episode”类包含了所有该片段相关信息，一个“super object”将有助于发现实际的视频对象。超类这个思想确实非常有益于建设低延迟页面的自动建设，这些超类可以帮助到基元对象到缓存的映射及保存。

Viacom为什么要使用Redis

每当Viacom上传一个视频片段，系统将建立一个私有的对象，并于1个超类关联。每一次修改，他们都需要重估私有对象的每个改变，并更新所有复合对象。同时，系统还需要无效Akamail中的URL请求。系统现有架构的组合及更[**敏捷**](http://lib.csdn.net/base/agile)的管理方法需求将Viacom推向了Redis。

基于Viacom主要基于[**PHP**](http://lib.csdn.net/base/php)，所以这个解决方案必须支持PHP。他们首先选择了memcached做对象存储，但是它并不能很好的支持hashmap；同时他们还需要一个更有效的进行无效步骤的重估，即更好的理解内容的依赖性。本质上说，他们需要时刻跟进无效步骤中的依赖性改变。因此他们选择了Redis及Predis的组合来解决这个问题。

他们团队使用Redis给southparkstudios.com和thedailyshow.com两个网站建设依赖性图，在取得了很大的成功后他们开始着眼Redis其它适合场景。

Redis的其它使用场景

显而易见，如果有人使用Redis来建设依赖性图，那么使用它来做对象处理也是说得通的。同样，这也成了架构团队为Redis选择的第二使用场景。Redis的复制及持久化特性同时也征服了Viacom的运营团队，因此在几个开发周期后，Redis成为他们网站的主要数据及依赖性储存。

后两个用例则是行为追踪及浏览计数的缓冲，改变后的架构是Redis每几分钟向MySQL中储存一次，而浏览计数则通过Redis进行存储及计数。同时Redis还被用来做人气的计算，一个基于访问数及访问时间的得分系统——如果某个视频最近被访问的次数越多，它的人气就越高。在如此多内容上每隔10-15分钟做一次计算绝对不是类似MySQL这样传统关系型数据库的强项，Viacom使用Redis的理由也非常简单——在1个存储浏览信息的Redis实例上运行Lua批处理作业，计算出所有的得分表。信息被拷贝到另一个Redis实例上，用以支持相关的产品查询。同时还在MySQL上做了另一个备份，用以以后的分析，这种组合会将这个过程耗费的时间降低60倍。

Viacom还使用Redis存储一步作业信息，这些信息被插入一个列表中，工作人员则使用[BLPOP](http://redis.io/commands/blpop)命令行在队列中抓取顶端的任务。同时zsets被用于从众多社交网络（比如Twitter及Tumblr）上综合内容，Viacom通过Brightcove视频播放器来同步多个内容管理系统。

横跨这些用例，几乎所有的Redis命令都被使用——sets、lists、zlists、hashmaps、scripts、counters等。同时，Redis也成为Viacom可扩展架构中不可或缺的一环。