Redis 实战

红丸出品

第一章 Redis 快速入门

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。

1.1 Key-Value 存储系统简介

Key-Value Store 是当下比较流行的话题，尤其在构建诸如搜索引擎、IM、P2P、游戏服务器、SNS 等大型互联网应用以及提供云计算服务的时候，怎样保证系统在海量数据环境下的高性能、高可靠性、高扩展性、高可用性、低成本成为所有系统架构们挖苦心思考虑的重点，而怎样解决数据库服务器的性能瓶颈是最大的挑战。

按照分布式领域的 CAP 理论（Consistency、 Availability、Tolerance to network Partitions 这三部分在任何系统架构实现时只可能同时满足其中二点，没法三者兼顾）来衡量，传统的关系数据库的 ACID 只满足了 Consistency、Availability，因此在 Partition tolerance 上就很难做得好。另外传统的关系数据库处理海量数据、分布式架构时候在 Performance、Scalability、Availability 等方面也存在很大的局限性。

而 Key-Value Store 更加注重对海量数据存取的性能、分布式、扩展性支持上，并不需要传统关系数据库的一些特征，例如：Schema、事务、完整 SQL 查询支持等等，因此在分布式环境下的性能相对于传统的关系数据库有较大的提升。

Key-Value 数据库分为很多种类，具体如下图:

项目名称	语言	存储性	持久性存储介质	客户端协议	数据模型	文档	赞助商/社区
Project Voldemort	Java	分区，复制，read-repair	Pluggable: BerkleyDB, Mysql	Java API	Structured / blob / text	A	Linkedin, no
Ringo	Erlang	分区，复制，immutable	Custom on-disk (append only log)	HTTP	blob	B	诺基亚, no
Scalaris	Erlang	分区，复制，paxos	In-memory only	Erlang, Java, HTTP	blob	B	OnScale, no
Kai	Erlang	分区，复制?	On-disk Dets file	Memcached	blob	C	no
Dynomite	Erlang	分区，复制	Pluggable: couch, dets	Custom ascii, Thrift	blob	D+	Powerset, no
MemcacheDB	C	复制	BerkleyDB	Memcached	blob	B	新浪网, some
ThruDB	C++	复制	Pluggable: BerkleyDB, Custom, Mysql, S3	Thrift	Document oriented	C+	Third rail, unsure
CouchDB	Erlang	复制，分区?	Custom on-disk	HTTP, json	Document oriented (json)	A	Apache, yes
Cassandra	Java	复制，分区	Custom on-disk	Thrift	Bigtable meets Dynamo	F	Facebook, no
HBase	Java	复制，分区	Custom on-disk	Custom API, Thrift, Rest	Bigtable	A	Apache, yes
Hypertable	C++	复制，分区	Custom on-disk (HDFS, KFS)	Thrift, other	Bigtable	A	Zvents, 百度, yes
Tokyo Tyrant	C	复制	Tokyo Cabinet	Memcached,	blob	A	mixi.jp, no

这些 Key-Value 数据库，有的是用 C/C++编写的，有的是用 Java 编写的，还有的是用 Erlang 编写的，每个都有自己的独到之处，我们从中挑选一些比较有特色且应用广泛的产品学习和了解一下。

1.1.1 Voldemort

Voldemort 是一个分布式 Key/Value 存储系统，它具有以下特点：

- 数据自动在多个服务器之间复制;
- 数据自动分区，因此每个服务器只包括整体数据的一个子集;
- 服务器故障处理是透明的;
- 支持插入式序列化，允许丰富的 Key 和 Value 类型，包括列表和元组，也可以集成常见的序列化框架，如 Protocol Buffers，Thrift，Avro 和 Java Serialization
- 数据项支持版本化，即使在故障情况下，数据完整性也可以得到保障;
- 每个节点都是独立的，无需其他节点协调，因此也没有中央节点;
- 单节点性能优秀：根据机器配置、网络、磁盘系统和数据复制因素的不同，每秒可以执行 10-20k 操作;
- 支持地理分散式部署。

1.1.2 Dynamo

Dynamo 是亚马逊的 key-value 模式的存储平台，可用性和扩展性都很好，性能也不错：读写访问中 99.9%的响应时间都在 300ms 内。

接下来对 Dynamo 需要的一些特性做一下简要的描述：

- Cost-effectiveness - 省钱！Dynamo 不像一些商用数据库产品，需要昂贵的服务器来得到良好的性能，而且可能增加 5%的访问量会需要你花 2 万美刀去买一台新服务器。而在 Dynamo 上，由于是利用一堆廉价机器来存数据，于是你可能只需要花个 500 刀买个破机器加入到集群里就行了。
- Dynamo 是一个 Key-Value 存储 - 因此他不支持外键和关联查询什么的。其 Value 值是二进制存储的，所以查询条件也只能作用在 Key 上。
- 配置简单的分布式存储 - 这是由于 Dynamo 是去中心化地设计，在集群中它的每一台机器都是对等的，不像 MongoDB 这样的中心化设计，于是它也不会有单点问题。

1.1.3 memcachedb

memcachedb 是 一个由新浪网的开发人员开放出来的开源项目，给 memcached 分布式缓存服务器添加了 Berkeley DB 的持久化存储机制和异步主辅复制机制，让 memcached 具备了事务恢复能力、持久化能力和分布式复制能力，非常适合于需要超高性能读写速度，但是 不需要严格事务约束，能够被持久化保存的应用场景，例如 memcachedb 被应用在新浪博客上面。

1.1.4 Cassandra

Apache Cassandra 是一套开源分布式 Key-Value 存储系统。它最初由 Facebook 开发，用于储存特别大的数据。Facebook 目前在使用此系统。
主要特性：

- 分布式
- 基于 column 的结构化
- 高伸展性

Cassandra 的主要特点就是它不是一个数据库，而是由一堆数据库节点共同构成的一个分布式网络服务，对 Cassandra 的一个写操作，会被复制到其他节点上去，对 Cassandra 的读操作，也会被路由到某个节点上面去读取。对于一个 Cassandra 群集来说，扩展性能 是比较简单的事情，只管在群集里面添加节点就可以了。

Cassandra 是一个混合型的非关系的数据库，类似于 Google 的 BigTable。其主要功能比 Dynomite（分布式的 Key-Value 存 储系统）更丰富，但支持度却不如文档存储 MongoDB（介于关系数据库和非关系数据库之间的开源产品，是非关系数据库当中功能最丰富，最像关系数据库 的。支持的数据结构非常松散，是类似 json 的 bjson 格式，因此可以存储比较复杂的数据类型。）Cassandra 最初由 Facebook 开发，后转变成了开源项目。它是一个网络社交云计算方面理想的数据库。以 Amazon 专有的完全分布式的 Dynamo 为基础，结合了 Google BigTable 基于列族（Column Family）的数据模型。P2P 去中心化的存储。很多方面都可以称之为 Dynamo 2.0。

和其他数据库比较，有几个突出特点：

- 模式灵活 ：使用 Cassandra，像文档存储，你不必提前解决记录中的字段。你可以在系统运行时随意的添加或移除字段。这是一个惊人的效率提升，特别是在大型部 署上。
- 真正的可扩展性 ：Cassandra 是纯粹意义上的水平扩展。为给集群添加更多容量，可以指向另一台电脑。你不必重启任何进程，改变应用查询，或手动迁移任何数据。
- 多数据中心识别 ：你可以调整你的节点布局来避免某一个数据中心起火，一个备用的数据中心将至少有每条记录的完全复制。
- 范围查询 ：如果你不喜欢全部的键值查询，则可以设置键的范围来查询。
- 列表数据结构 ：在混合模式可以将超级列添加到 5 维。对于每个用户的索引，这是非常方便的。
- 分布式写操作 ：有可以在任何地方任何时间集中读或写任何数据。并且不会有任何单点失败。

1.1.5 memcached

memcached 是一套分布式的快取系统，当初是 Danga Interactive 为了 LiveJournal 所发展的，但目前被许多软件（如 MediaWiki）所使用。这是一套开放源代码软件，以 BSD license 授权释出。

memcached 缺乏认证以及安全管制，这代表应该将 memcached 服务器放置在防火墙后。

memcached 的 API 使用三十二位元的循环冗余校验（CRC-32）计算键值后，将资料分散在不同的机器上。当表格满了以后，接下来新增的资料会以 LRU 机制替换掉。由于 memcached 通常只是当作快取系统使用，所以使用 memcached 的应用程式在写回较慢的系统时（像是后端的数据库）需要额外的程式码更新 memcached 内的资料。

memcached 具有多种语言的客户端开发包，包括：Perl/PHP/JAVA/C/Python/Ruby/C#/MySQL/

1.1.6 Hypertable

Hypertable 是一个开源、高性能、可伸缩的数据库，它采用与 Google 的 Bigtable 相似的模型。在过去数年中，Google 为在 PC 集群 上运行的可伸缩计算基础设施设计建造了三个关键部分。第一个关键的基础设施是 Google File System（GFS），这是一个高可用的文件系统，提供了一个全局的命名空间。它通过跨机器（和跨机架）的文件数据复制来达到高可用性，并因此免受传统 文件存储系统无法避免的许多失败的影响，比如电源、内存和网络端口等失败。第二个基础设施是名为 Map-Reduce 的计算框架，它与 GFS 紧密协作，帮 助处理收集到的海量数据。第三个基础设施是 Bigtable，它是传统数据库的替代。Bigtable 让你可以通过一些主键来组织海量数据，并实现高效的 查询。Hypertable 是 Bigtable 的一个开源实现，并且根据我们的想法进行了一些改进。

1.2 为什么选择 Key-Value Store

大量的互联网用户选择 Key-Value Store 的原因具体是什么呢？ 主要分为下面的 2 个主要原因：

1.2.1 大规模的互联网应用

对于 google，ebay 这样的互联网企业，每时每刻都有无数的用户在使用它们提供的互联网服务，这些服务带来的就是大量的数据吞吐量，在同一时间，会并发的有成千上万的连接对数据库进行操作。在这种情况下，单台服务器或者几台服务器远远不能满足这些数据处理的需求，简单的升级服务器性能这样的 scale up 的方式也不行，所以唯一可以采用的办法就是 scale out 了。scale out 的方法有很多种，但大致分为两类：一类仍然采用 RDBMS，然后通过对数据库的垂直和水平切割将整个数据库部署到一个集群上，这种方法的优点在于可以采用 RDBMS 这种熟悉的技术，但缺点在于它是针对特定应用的，就是说，由于应用的不同，切割的方法是不一样的。

还有一类就是 google 所采用的方法，抛弃 RDBMS，采用 key-value 形式的存储，这样可 以极大的增强系统的可扩展性（scalability），如果要处理的数据持续增大，多加机器就 可以了。事实上，key-value 的存储就是由于 BigTable 等相关论文的发表慢慢进入人们的 视野的。

1.2.2 云存储

如果说上一个问题还有可以替代的解决方案（切割数据库）的话，那么对于云存储来说，也许 key-value 的 store 就是唯一的解决方案了。云存储简单点说就是构建一个大型的存储平台给别人用，这也就意味着在这上面运行的应用其实是不可控的。如果其中某个客户的应用随着用户的增长而不断增长时，云存储供应商是没有办法通过数据库的切割来达到 scale 的，因为这个数据是客户的，供应商不了解这个数据自然就没法作出切割。在这种情况下，key-value 的 store 就是唯一的选择了，因为这种条件下的 scalability 必须是自动完成的，不能有人工干预。这也是为什么几乎所有的现有的云存储都是 key-value 形式的，例如 Amazon 的 smipleDB，底层实现就是 key-value，还有 google 的 GoogleAppEngine，采用的是 BigTable 的存储形式。也许唯一可能例外的是 MS 的解决方案，我在 Qcon 大会上听说 MS 的 Azure 平台的下一个版本中就会推出基于 RDBMS 的云存储，尽管我本人仍然对此保持怀疑。

Key-Value Store 最大的特点就是它的可扩展性，这也就是它最大的优势。所谓的可扩展性，在我看来这里包括了两方面内容。一方面，是指 Key-Value Store 可以支持极大的数据的存储，它的分布式的架构决定了只要有更多的机器，就能够保证存储更多的数据。另一方面，是指它可以支持数量很多的并发的查询。对于 RDBMS，一般几百个并发的查询就可以让它很吃力了，而一个 Key-Value Store，可以很轻松的支持上千的并发查询。下面而简单的罗列了一些特点：

- Key-value store：一个 key-value 数据存储系统，只支持一些基本操作，如：SET(key, value)

和 GET(key) 等；

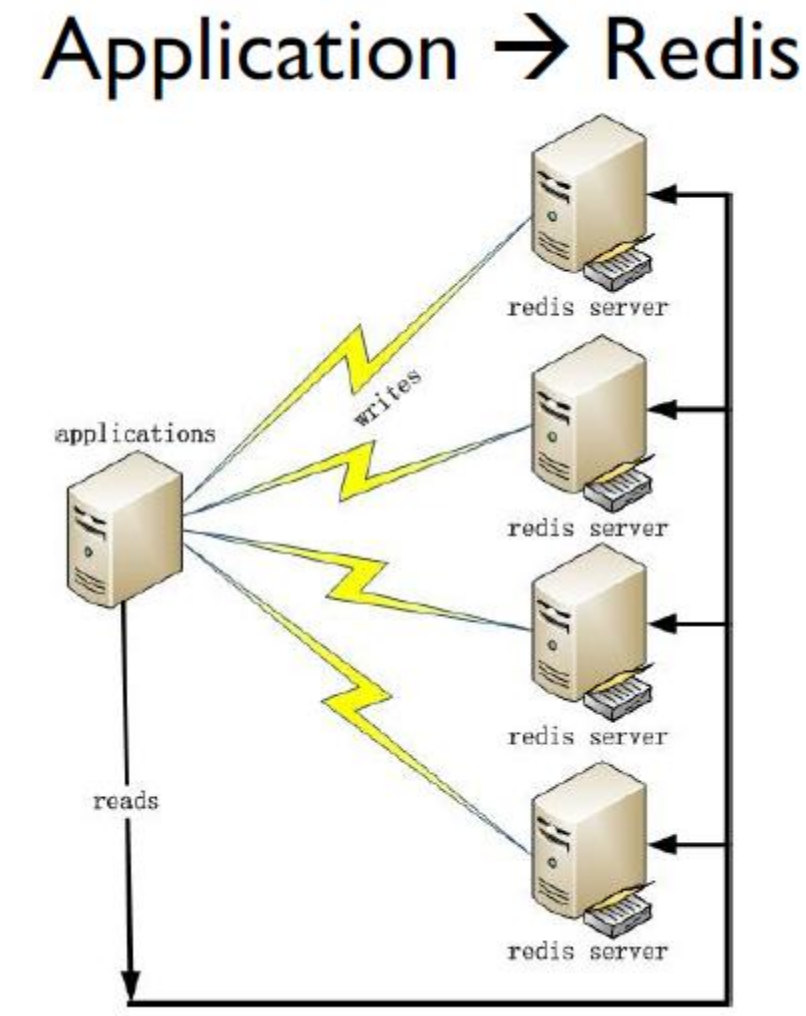
- 分布式：多台机器（nodes）同时存储数据和状态，彼此交换消息来保持数据一致，可视为一个完整的存储系统。
- 数据一致：所有机器上的数据都是同步更新的、不用担心得到不一致的结果；
- 冗余：所有机器（nodes）保存相同的数据，整个系统的存储能力取决于单台机器（node）的能力；
- 容错：如果有少数 nodes 出错，比如重启、当机、断网、网络丢包等各种 fault/fail 都不影响整个系统的运行；
- 高可靠性：容错、冗余等保证了数据库系统的可靠性。

1.2.3 Redis 实际应用案例

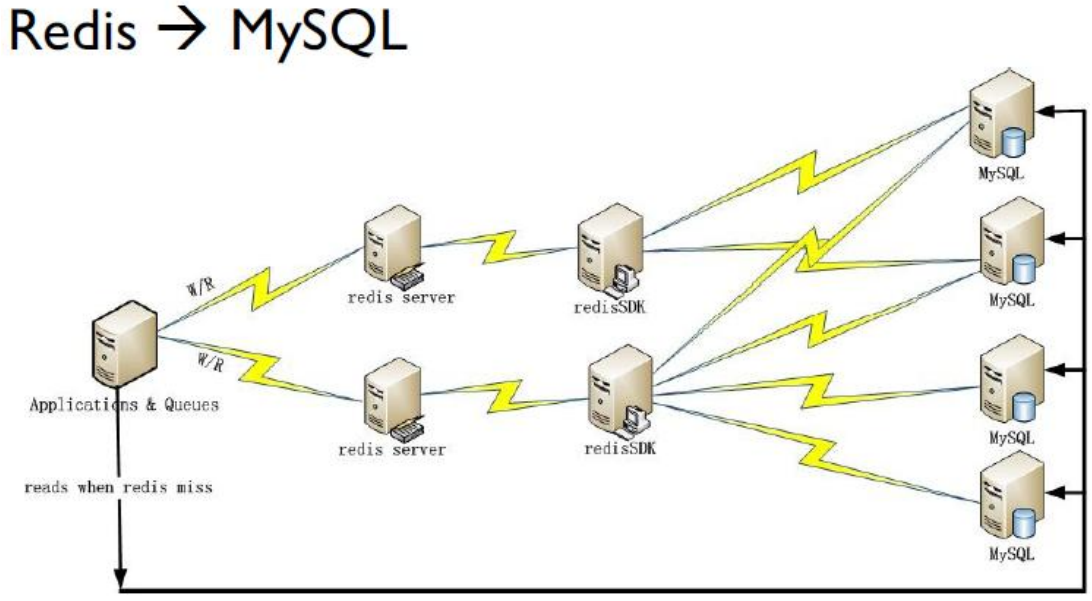
目前全球最大的 Redis 用户是新浪微博，在新浪有 200 多台物理机，400 多个端口正在运行着 Redis，有+4G 的数据跑在 Redis 上来为微博用户提供服务。

在新浪微博 Redis 的部署场景很多，大概分为如下的 2 种:

第一种是应用程序直接访问 Redis 数据库



第二种是应用程序直接访问 Redis，只有当 Redis 访问失败时才访问 MySQL



同时，Digg 的一项新功能，添加了对文章浏览数的显示，这一功能的一大卖点是其实时性。

而支持此实时浏览量计数的，正是 Redis。

1.3 初识 Redis

Redis 是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。从 2010 年 3 月 15 日起，Redis 的开发工作由 VMware 主持。

1.3.1 数据类型

作为 Key-value 型数据库，Redis 也提供了键（Key）和键值（Value）的映射关系。但是，除了常规的数值或字符串，Redis 的键值还可以是以下形式之一：

- Lists （列表）
- Sets （集合）
- Sorted sets （有序集合）
- Hashes （哈希表）

键值的数据类型决定了该键值支持的操作。Redis 支持诸如列表、集合或有序集合的交集、并集、查集等高级原子操作；同时，如果键值的类型是普通数字，Redis 则提供自增等原子操作。

1.3.2 持久化

通常，Redis 将数据存储于内存中，或被配置为使用虚拟内存。通过两种方式可以实现数据持久化：使用截图的方式，将内存中的数据不断写入磁盘；或使用类似 MySQL 的日志方式，记录每次更新的日志。前者性能较高，但是可能会引起一定程度的数据丢失；后者相反。

1.3.3 主从同步

Redis 支持将数据同步到多台从库上，这种特性对提高读取性能非常有益。

1.3.4 性能

相比需要依赖磁盘记录每个更新的数据库，基于内存的特性无疑给 Redis 带来了非常优秀的性能。读写操作之间有显著的性能差异。

1.3.5 提供 API 的语言

- C
- C++
- C#
- Clojure
- Common Lisp
- Erlang
- Haskell
- Java
- Javascript
- Lua
- Objective-C
- Perl
- PHP
- Python
- Ruby
- Scala
- Go
- Tcl

1.3.6 适用场合

毫无疑问，Redis 开创了一种新的数据存储思路，使用 Redis，我们不用在面对功能单调的数据库时，把精力放在如何把大象放进冰箱这样的问题上，而是利用 Redis 灵活多变的数据结构和数据操作，为不同的大象构建不同的冰箱。希望你喜欢这个比喻。

下面是 Redis 适用的一些场景:

1、取最新 N 个数据的操作

比如典型的取你网站的最新文章，通过下面方式，我们可以将最新的 5000 条评论的 ID 放在 Redis 的 List 集合中，并将超出集合部分从数据库获取。

使用 LPUSH latest.comments<ID>命令，向 list 集合中插入数据
插入完成后再用 LTRIM latest.comments 0 5000 命令使其永远只保存最近 5000 个 ID
然后我们在客户端获取某一页评论时可以用下面的逻辑

```
FUNCTION get_latest_comments(start,num_items):
```

```
    id_list = redis.lrange("latest.comments",start,start+num_items-1)
    IF id_list.length < num_items
        id_list = SQL_DB("SELECT ... ORDER BY time LIMIT ...")
    END
    RETURN id_list
END
```

如果你还有不同的筛选维度，比如某个分类的最新 N 条，那么你可以再建一个按此分类的 List，只存 ID 的话，Redis 是非常高效的。

2、排行榜应用，取 TOP N 操作

这个需求与上面需求的不同之处在于，前面操作以时间为权重，这个是以某个条件为权重，比如按顶的次数排序，这时候就需要我们的 sorted set 出马了，将你要排序的值设置成 sorted set 的 score，将具体的数据设置成相应的 value，每次只需要执行一条 ZADD 命令即可。

3、需要精准设定过期时间的应用

比如你可以把上面说到的 sorted set 的 score 值设置成过期时间的时间戳，那么就可以简单地通过过期时间排序，定时清除过期数据了，不仅是清除 Redis 中的过期数据，你完全可以把 Redis 里这个过期时间当成是对数据库中数据的索引，用 Redis 来找出哪些数据需要过期删除，然后再精准地从数据库中删除相应的记录。

4、计数器应用

Redis 的命令都是原子性的，你可以轻松地利用 INCR，DECR 命令来构建计数器系统。

5、Uniq 操作，获取某段时间所有数据排重值

这个使用 Redis 的 set 数据结构最合适了，只需要不断地将数据往 set 中扔就行了，set 意为集合，所以会自动排重。

6、实时系统，反垃圾系统

通过上面说到的 set 功能，你可以知道一个终端用户是否进行了某个操作，可以找到其操作的集合并进行分析统计对比等。没有做不到，只有想不到。

7、Pub/Sub 构建实时消息系统

Redis 的 Pub/Sub 系统可以构建实时的消息系统，比如很多用 Pub/Sub 构建的实时聊天系统的例子。

8、构建队列系统

使用 list 可以构建队列系统，使用 sorted set 甚至可以构建有优先级的队列系统。

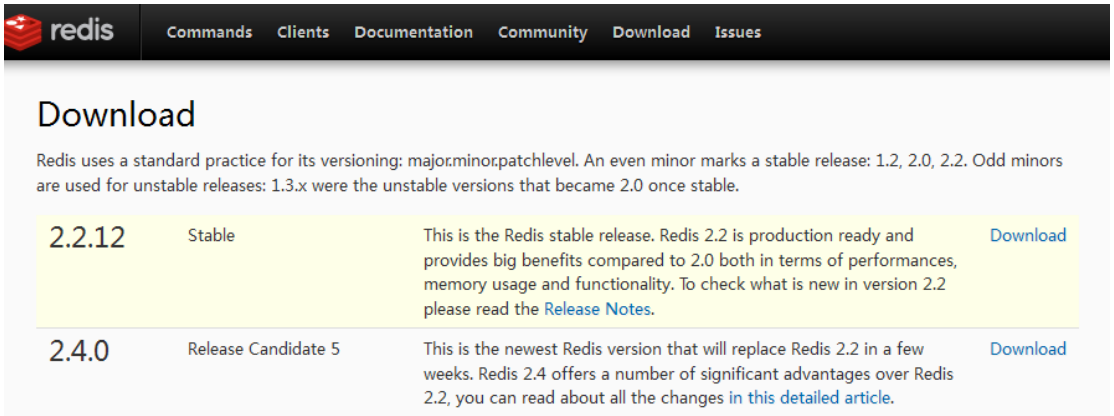
9、缓存

这个不必说了，性能优于 Memcached，数据结构更多样化。

1.4 快速入门

1.4.1 安装 Redis

Redis 的官方下载站是 http://redis.io/download，可以去上面下载最新的安装程序下来，我写此文章时的的稳定版本是 2.2.12。



怎么安装 Redis 数据库呢？下面将介绍 Linux 版本的安装方法

步骤一：下载 Redis

下载安装包：wget http://redis.googlecode.com/files/redis-2.2.12.tar.gz

```
[root@localhost 4setup]# wget http://redis.googlecode.com/files/redis-2.2.12.tar.gz
--19:06:56--  http://redis.googlecode.com/files/redis-2.2.12.tar.gz
正在解析主机 redis.googlecode.com... 74.125.71.82
Connecting to redis.googlecode.com|74.125.71.82|:80... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度：455240 (445K) [application/x-gzip]
Saving to: `redis-2.2.12.tar.gz'
100%[======================================>] 455,240       34.8K/s   in 13s
19:07:16 (34.8 KB/s) - `redis-2.2.12.tar.gz' saved [455240/455240]
[root@localhost 4setup]#
```

步骤二：编译源程序

```
[root@localhost 4setup]# ll
总计 29168
-rw-r--r-- 1 root root      455240 2011-07-22 redis-2.2.12.tar.gz
[root@localhost 4setup]# tar xzf redis-2.2.12.tar.gz
[root@localhost 4setup]# cd redis-2.2.12
[root@localhost redis-2.2.12]# make
cd src && make all
make[1]: Entering directory `/root/4setup/redis-2.2.12/src'
```

步骤三：启动 Redis 服务

src/redis-server

```
[root@localhost redis-2.2.12]# src/redis-server
[6246] 05 Aug 19:17:22 # Warning: no config file specified, using the default config. In order to specify a config file use 'redis-server /path/to/redis.conf'
[6246] 05 Aug 19:17:22 * Server started, Redis version 2.2.12
[6246] 05 Aug 19:17:22 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
[6246] 05 Aug 19:17:22 * The server is now ready to accept connections on port 6379
[6246] 05 Aug 19:17:22 - 0 clients connected (0 slaves), 539544 bytes in use
```

Redis 服务端的默认连接端口是 6379

步骤四：将 Redis 作为 Linux 服务随机启动

vi /etc/rc.local，使用 vi 编辑器打开随机启动配置文件，并在其中加入下面一行代码

/root/4setup/redis-2.2.12/src/redis-server

步骤五：客户端连接验证

新打开一个 Session 输入：src/redis-cli，如果出现下面提示，那么您就可以开始 Redis 之旅了

```
[root@localhost redis-2.2.12]# src/redis-cli
redis 127.0.0.1:6379>
```

步骤六：查看 Redis 日志

查看服务器端 session，即可对 Redis 的运行状况进行查看或分析了

```
[6246] 05 Aug 19:24:33 - 0 clients connected (0 slaves), 539544 bytes in use
[6246] 05 Aug 19:24:37 - Accepted 127.0.0.1:51381
[6246] 05 Aug 19:24:38 - 1 clients connected (0 slaves), 547372 bytes in use
```

以上的几个步骤就 OK 了!!这样一个简单的 Redis 数据库就可以畅通无阻地运行起来了。

步骤七：停止 Redis 实例

最简单的方法是在启动实例的 session 中，直接使用 Control-C 来将实例停止。

我们还可以用客户端来停止服务，如可以用 shutdown 来停止 Redis 实例，具体如下:

```
[root@localhost redis-2.2.12]# src/redis-cli shutdown
```

1.4.2 配置 Redis

如果是一个专业的 DBA，那么实例启动时会加很多的参数以便使系统运行的非常稳定，这样就可能会在启动时在 Redis 后面加一个参数，以指定配置文件的路径，就象 mysql 一样的读取启动配置文件的方式来启动数据库。源码编译完成后，在 redis-2.2.12 目录下有一个 redis.conf 文件，这个文件即是 Redis 的配置文件，用配置文件来启动 Redis 的方法如下:

```
[root@localhost redis-2.2.12]# src/redis-server redis.conf
[6353] 05 Aug 19:36:45 * Server started, Redis version 2.2.12
[6353] 05 Aug 19:36:45 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
[6353] 05 Aug 19:36:45 * The server is now ready to accept connections on port 6379
[6353] 05 Aug 19:36:45 - 0 clients connected (0 slaves), 539540 bytes in use
```

Redis 支持很多的参数，但都有默认值。

- **daemonize:**
  默认情况下，redis 不是在后台运行的，如果需要在后台运行，把该项的值更改为 yes
- **pidfile**
  当 Redis 在后台运行的时候，Redis 默认会把 pid 文件放在/var/run/redis.pid，你可以配置到其他地址。当运行多个 redis 服务时，需要指定不同的 pid 文件和端口
- **bind**
  指定 Redis 只接收来自于该 IP 地址的请求，如果不进行设置，那么将处理所有请求，在生产环境中最好设置该项
- **port**
  监听端口，默认为 6379
- **timeout**
  设置客户端连接时的超时时间，单位为秒。当客户端在这段时间内没有发出任何指令，那么关闭该连接
- **loglevel**
  log 等级分为 4 级，debug, verbose, notice，和 warning。生产环境下一般开启 notice
- **logfile**
  配置 log 文件地址，默认使用标准输出，即打印在命令行终端的窗口上
- **databases**
  设置数据库的个数，可以使用 SELECT <dbid>命令来切换数据库。默认使用的数据库是 0
- **save**
  设置 Redis 进行数据库镜像的频率。
  if(在 60 秒之内有 10000 个 keys 发生变化时){
  　　进行镜像备份
  }else if(在 300 秒之内有 10 个 keys 发生了变化){
  　　进行镜像备份
  }else if(在 900 秒之内有 1 个 keys 发生了变化){
  　　进行镜像备份
  }
- **rdbcompression**
  在进行镜像备份时，是否进行压缩
- **dbfilename**
  镜像备份文件的文件名
- **dir**
  数据库镜像备份的文件放置的路径。这里的路径跟文件名要分开配置是因为 Redis 在进

行备份时，先会将当前数据库的状态写入到一个临时文件中，等备份完成时，再把该该临时文件替换为上面所指定的文件，而这里的临时文件和上面所配置的备份文件都会放在这个指定的路径当中

- **slaveof**
  设置该数据库为其他数据库的从数据库
- **masterauth**
  当主数据库连接需要密码验证时，在这里指定
- **requirepass**
  设置客户端连接后进行任何其他指定前需要使用的密码。警告：因为 redis 速度相当快，所以在一台比较好的服务器下，一个外部的用户可以在一秒钟进行 150K 次的密码尝试，这意味着你需要指定非常非常强大的密码来防止暴力破解。
- **maxclients**
  限制同时连接的客户数量。当连接数超过这个值时，redis 将不再接收其他连接请求，客户端尝试连接时将收到 error 信息。
- **maxmemory**
  设置 redis 能够使用的最大内存。当内存满了的时候，如果还接收到 set 命令，redis 将先尝试剔除设置过 expire 信息的 key，而不管该 key 的过期时间还没有到达。在删除时，将按照过期时间进行删除，最早将要被过期的 key 将最先被删除。如果带有 expire 信息的 key 都删光了，那么将返回错误。这样，redis 将不再接收写请求，只接收 get 请求。maxmemory 的设置比较适合于把 redis 当作于类似 memcached 的缓存来使用。
- **appendonly**
  默认情况下，redis 会在后台异步的把数据库镜像备份到磁盘，但是该备份是非常耗时的，而且备份也不能很频繁，如果发生诸如拉闸限电、拔插头等状况，那么将造成比较大范围的数据丢失。所以 redis 提供了另外一种更加高效的数据库备份及灾难恢复方式。开启 append only 模式之后，redis 会把所接收到的每一次写操作请求都追加到 appendonly.aof 文件中，当 redis 重新启动时，会从该文件恢复出之前的状态。但是这样会造成 appendonly.aof 文件过大，所以 redis 还支持了 BGREWRITEAOF 指令，对 appendonly.aof 进行重新整理。所以我认为推荐生产环境下的做法为关闭镜像，开启 appendonly.aof，同时可以选择在访问较少的时间每天对 appendonly.aof 进行重写一次。
- **appendfsync**
  设置对 appendonly.aof 文件进行同步的频率。always 表示每次有写操作都进行同步，everysec 表示对写操作进行累积，每秒同步一次。这个需要根据实际业务场景进行配置
- **vm-enabled**
  是否开启虚拟内存支持。因为 redis 是一个内存数据库，而且当内存满的时候，无法接收新的写请求，所以在 redis 2.0 中，提供了虚拟内存的支持。但是需要注意的是，redis 中，所有的 key 都会放在内存中，在内存不够时，只会把 value 值放入交换区。这样保证了虽然使用虚拟内存，但性能基本不受影响，同时，你需要注意的是你要把 vm-max-memory 设置到足够来放下你的所有的 key
- **vm-swap-file**
  设置虚拟内存的交换文件路径
- **vm-max-memory**
  这里设置开启虚拟内存之后，redis 将使用的最大物理内存的大小。默认为 0，redis 将把他所有的能放到交换文件的都放到交换文件中，以尽量少的使用物理内存。在生产环境下，需要根据实际情况设置该值，最好不要使用默认的 0

- **vm-page-size**
  设置虚拟内存的页大小，如果你的 value 值比较大，比如说你要在 value 中放置博客、新闻之类的所有文章内容，就设大一点，如果要放置的都是很小的内容，那就设小一点。
- **vm-pages**
  设置交换文件的总的 page 数量，需要注意的是，page table 信息会放在物理内存中，每 8 个 page 就会占据 RAM 中的 1 个 byte。总的虚拟内存大小 ＝ vm-page-size * vm-pages
- **vm-max-threads**
  设置 VM IO 同时使用的线程数量。因为在进行内存交换时，对数据有编码和解码的过程，所以尽管 IO 设备在硬件上本上不能支持很多的并发读写，但是还是如果你所保存的 vlaue 值比较大，将该值设大一些，还是能够提升性能的
- **glueoutputbuf**
  把小的输出缓存放在一起，以便能够在一个 TCP packet 中为客户端发送多个响应，具体原理和真实效果我不是很清楚。所以根据注释，你不是很确定的时候就设置成 yes
- **hash-max-zipmap-entries**
  在 redis 2.0 中引入了 hash 数据结构。当 hash 中包含超过指定元素个数并且最大的元素没有超过临界时，hash 将以一种特殊的编码方式（大大减少内存使用）来存储，这里可以设置这两个临界值
- **activerehashing**
  开启之后，redis 将在每 100 毫秒时使用 1 毫秒的 CPU 时间来对 redis 的 hash 表进行重新 hash，可以降低内存的使用。当你的使用场景中，有非常严格的实时性需要，不能够接受 Redis 时不时的对请求有 2 毫秒的延迟的话，把这项配置为 no。如果没有这么严格的实时性要求，可以设置为 yes，以便能够尽可能快的释放内存

1.4.3 操作数据库

下面我们来简单的操作一下数据库。

插入数据

```
redis 127.0.0.1:6379> set name wwl
OK
```

设置一个 key-value 对

查询数据

```
redis 127.0.0.1:6379> get name
"wwl"
```

取出 key 所对应的 value

删除键值

```
redis 127.0.0.1:6379> del name
```

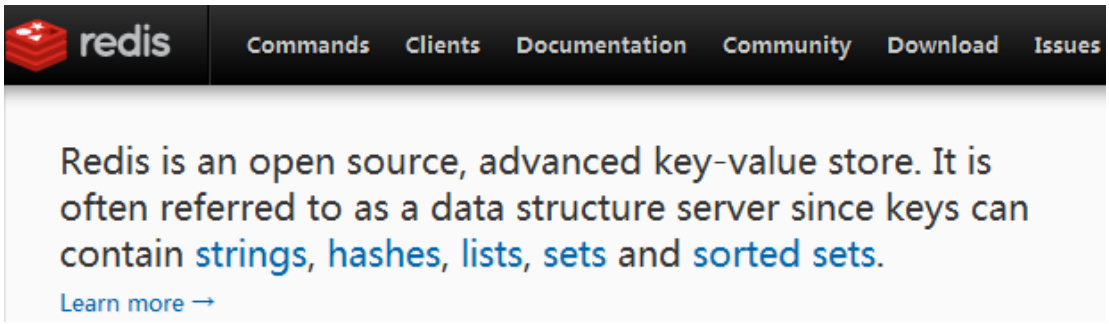
删除这个 key 及对应的 value

验证键是否存在

```
redis 127.0.0.1:6379> exists name
(integer) 0
```

其中 0，代表此 key 不存在；1 代表存在

第二章 Redis 数据类型及操作



2.1 前言

Redis 的作者 antirez（Salvatore Sanfilippo）曾经发表了一篇名为 Redis 宣言（Redis Manifesto）的文章，文中列举了 Redis 的七个原则，以向大家阐明 Redis 的思想。

1、Redis 是一个操作数据结构的语言工具，它提供基于 TCP 的协议以操作丰富的数据结构。在 Redis 中，数据结构这个词的意义不仅表示在某种数据结构上的操作，更包括了结构本身及这些操作的时间空间复杂度。

2、Redis 定位于一个内存数据库，正是由于内存的快速访问特性，才使得 Redis 能够有如此高的性能，才使得 Redis 能够轻松处理大量复杂的数据结构，Redis 会尝试其它的存储方面的选择，但是永远不会改变它是一个内存数据库的角色。

3、Redis 使用基础的 API 操作基础的数据结构，Redis 的 API 与数据结构一样，都是一些最基础的元素，你几乎可以将任何信息交互使用此 API 格式表示。作者调侃说，如果有其它非人类的智能生物存在，他们也能理解 Redis 的 API。因为它是如此的基础。

4、Redis 有着诗一般优美的代码，经常有一些不太了解 Redis 有的人会建议 Redis 采用一些其它人的代码，以实现一些 Redis 未实现的功能，但这对我们来说就像是非要给《红楼梦》接上后四十回一样。

5、Redis 始终避免复杂化，我们认为设计一个系统的本质，就是与复杂化作战。我们不会为了一个小功能而往源码里添加上千行代码，解决复杂问题的方法就是让复杂问题永远不要提复杂的问题。

6、Redis 支持两个层成的 API，第一个层面包含部分操作 API，但它支持用于分布式环境下的 Redis。第二个层面的 API 支持更复杂的 multi-key 操作。它们各有所长，但是我们不会推出两者都支持的 API，但我们希望能够提供实例间数据迁移的命令，并执行 multi-key 操作。

7、我们以优化代码为乐，我们相信编码是一件辛苦的工作，唯一对得起这辛苦的就是去享受它。如果我们在编码中失去了乐趣，那最好的解决办法就是停下来。我们决不会选择让 Redis 不好玩的开发模式。

Redis 的作者 antirez 曾笑称 Redis 为一个数据结构服务器（data structures server），我认为这是一个非常准确的表述，Redis 的所有功能就是将数据以其固有的几种结构来保存，并提供给用户操作这几种结构的接口。本文将介绍 Redis 支持的各种数据类型及其操作接口。

2.2 strings 类型及操作

string 是最简单的类型，你可以理解成与 Memcached 是一模一样的类型，一个 key 对应一个 value，其上支持的操作与 Memcached 的操作类似。但它的功能更丰富。

string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据，比如 jpg 图片或者序列化的对象。从内部实现来看其实 string 可以看作 byte 数组，最大上限是 1G 字节，下面是 string 类型的定义:

```
struct sdshdr {
   long len;
   long free;
   char buf[];
};
```

len 是 buf 数组的长度。
free 是数组中剩余可用字节数，由此可以理解为什么 string 类型是二进制安全的了，因为它本质上就是个 byte 数组，当然可以包含任何数据了
buf 是个 char 数组用于存贮实际的字符串内容，其实 char 和 c#中的 byte 是等价的，都是一个字节。
另外 string 类型可以被部分命令按 int 处理.比如 incr 等命令，如果只用 string 类型，redis 就可以被看作加上持久化特性的 memcached。当然 redis 对 string 类型的操作比 memcached 还是多很多的，具体操作方法如下：

2.2.1 set

设置 key 对应的值为 string 类型的 value。
例如我们添加一个 name= HongWan 的键值对，可以这样做:

```
redis 127.0.0.1:6379> set name HongWan
OK
redis 127.0.0.1:6379>
```

2.2.2 setnx

设置 key 对应的值为 string 类型的 value。如果 key 已经存在，返回 0，nx 是 not exist 的意思。

例如我们添加一个 name= HongWan_new 的键值对，可以这样做:

```
redis 127.0.0.1:6379> get name
"HongWan"
redis 127.0.0.1:6379> setnx name HongWan_new
(integer) 0
redis 127.0.0.1:6379> get name
"HongWan"
redis 127.0.0.1:6379>
```

由于原来 name 有一个对应的值，所以本次的修改不生效，且返回码是 0。

2.2.3 setex

设置 key 对应的值为 string 类型的 value，并指定此键值对应的有效期。

例如我们添加一个 haircolor= red 的键值对，并指定它的有效期是 10 秒，可以这样做:

```
redis 127.0.0.1:6379> setex haircolor 10 red
OK
redis 127.0.0.1:6379> get haircolor
"red"
redis 127.0.0.1:6379> get haircolor
(nil)
redis 127.0.0.1:6379>
```

可见由于最后一次的调用是 10 秒以后了，所以取不到 haicolor 这个键对应的值。

2.2.4 setrange

设置指定 key 的 value 值的子字符串。

例如我们希望将 HongWan 的 126 邮箱替换为 gmail 邮箱，那么我们可以这样做:

```
redis 127.0.0.1:6379> get name
"HongWan@126.com"
redis 127.0.0.1:6379> setrange name 8 gmail.com
(integer) 17
redis 127.0.0.1:6379> get name
"HongWan@gmail.com"
redis 127.0.0.1:6379>
```

其中的 8 是指从下标为 8（包含 8）的字符开始替换

2.2.5 mset

一次设置多个 key 的值，成功返回 ok 表示所有的值都设置了，失败返回 0 表示没有任何值被设置。

```
redis 127.0.0.1:6379> mset key1 HongWan1 key2 HongWan2
```

```
OK
redis 127.0.0.1:6379> get key1
"HongWan1"
redis 127.0.0.1:6379> get key2
"HongWan2"
redis 127.0.0.1:6379>
```

2.2.6 msetnx

一次设置多个 key 的值，成功返回 ok 表示所有的值都设置了，失败返回 0 表示没有任何值被设置，但是不会覆盖已经存在的 key。

```
redis 127.0.0.1:6379> get key1
"HongWan1"
redis 127.0.0.1:6379> get key2
"HongWan2"
redis 127.0.0.1:6379> msetnx key2 HongWan2_new key3 HongWan3
(integer) 0
redis 127.0.0.1:6379> get key2
"HongWan2"
redis 127.0.0.1:6379> get key3
(nil)
```

可以看出如果这条命令返回 0，那么里面操作都会回滚，都不会被执行。

2.2.7 get

获取 key 对应的 string 值,如果 key 不存在返回 nil。
例如我们获取一个库中存在的键 name，可以很快得到它对应的 value

```
redis 127.0.0.1:6379> get name
"HongWan"
redis 127.0.0.1:6379>
```

我们获取一个库中不存在的键 name1，那么它会返回一个 nil 以表时无此键值对

```
redis 127.0.0.1:6379> get name1
(nil)
redis 127.0.0.1:6379>
```

2.2.8 getset

设置 key 的值，并返回 key 的旧值。

```
redis 127.0.0.1:6379> get name
"HongWan"
```

```
redis 127.0.0.1:6379> getset name HongWan_new
"HongWan"
redis 127.0.0.1:6379> get name
"HongWan_new"
redis 127.0.0.1:6379>
```

接下来我们看一下如果 key 不存的时候会什么样儿?

```
redis 127.0.0.1:6379> getset name1 aaa
(nil)
redis 127.0.0.1:6379>
```

可见，如果 key 不存在，那么将返回 nil

2.2.9 getrange

获取指定 key 的 value 值的子字符串。
具体样例如下:

```
redis 127.0.0.1:6379> get name
"HongWan@126.com"
redis 127.0.0.1:6379> getrange name 0 6
"HongWan"
redis 127.0.0.1:6379>
```

字符串左面下标是从 0 开始的

```
redis 127.0.0.1:6379> getrange name -7 -1
"126.com"
redis 127.0.0.1:6379>
```

字符串右面下标是从-1 开始的

```
redis 127.0.0.1:6379> getrange name 7 100
"@126.com"
redis 127.0.0.1:6379>
```

当下标超出字符串长度时，将默认为是同方向的最大下标

2.2.10 mget

一次获取多个 key 的值，如果对应 key 不存在，则对应返回 nil。
具体样例如下:

```
redis 127.0.0.1:6379> mget key1 key2 key3
1) "HongWan1"
2) "HongWan2"
3) (nil)
redis 127.0.0.1:6379>
```

key3 由于没有这个键定义，所以返回 nil。

2.2.11 incr

对 key 的值做加加操作,并返回新的值。注意 incr 一个不是 int 的 value 会返回错误，incr 一个不存在的 key，则设置 key 为 1

```
redis 127.0.0.1:6379> set age 20
OK
redis 127.0.0.1:6379> incr age
(integer) 21
redis 127.0.0.1:6379> get age
"21"
redis 127.0.0.1:6379>
```

2.2.12 incrby

同 incr 类似，加指定值 ，key 不存在时候会设置 key，并认为原来的 value 是 0

```
redis 127.0.0.1:6379> get age
"21"
redis 127.0.0.1:6379> incrby age 5
(integer) 26
redis 127.0.0.1:6379> get name
"HongWan@gmail.com"
redis 127.0.0.1:6379> get age
"26"
redis 127.0.0.1:6379>
```

2.2.13 decr

对 key 的值做的是减减操作，decr 一个不存在 key，则设置 key 为-1

```
redis 127.0.0.1:6379> get age
"26"
redis 127.0.0.1:6379> decr age
(integer) 25
redis 127.0.0.1:6379> get age
"25"
redis 127.0.0.1:6379>
```

2.2.14 decrby

同 decr，减指定值。

```
redis 127.0.0.1:6379> get age
"25"
redis 127.0.0.1:6379> decrby age 5
(integer) 20
redis 127.0.0.1:6379> get age
"20"
redis 127.0.0.1:6379>
```

decrby 完全是为了可读性，我们完全可以通过 incrby 一个负值来实现同样效果，反之一样。

```
redis 127.0.0.1:6379> get age
"20"
redis 127.0.0.1:6379> incrby age -5
(integer) 15
redis 127.0.0.1:6379> get age
"15"
redis 127.0.0.1:6379>
```

2.2.15 append

给指定 key 的字符串值追加 value,返回新字符串值的长度。
例如我们向 name 的值追加一个@126.com 字符串，那么可以这样做:

```
redis 127.0.0.1:6379> append name @126.com
(integer) 15
redis 127.0.0.1:6379> get name
"HongWan@126.com"
redis 127.0.0.1:6379>
```

2.2.16 strlen

取指定 key 的 value 值的长度。

```
redis 127.0.0.1:6379> get name
"HongWan_new"
redis 127.0.0.1:6379> strlen name
(integer) 11
redis 127.0.0.1:6379> get age
"15"
redis 127.0.0.1:6379> strlen age
(integer) 2
redis 127.0.0.1:6379>
```

2.3 hashes 类型及操作

Redis hash 是一个 string 类型的 field 和 value 的映射表.它的添加、删除操作都是 O(1)（平均）。hash 特别适合用于存储对象。相较于将对象的每个字段存成单个 string 类型。将一个对象存储在 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。省内存的原因是新建一个 hash 对象时开始是用 zipmap（又称为 small hash）来存储的。这个 zipmap 其实并不是 hash table，但是 zipmap 相比正常的 hash 实现可以节省不少 hash 本身需要的一些元数据存储开销。尽管 zipmap 的添加，删除，查找都是 O(n)，但是由于一般对象的 field 数量都不太多。所以使用 zipmap 也是很快的,也就是说添加删除平均还是 O(1)。如果 field 或者 value 的大小超出一定限制后，Redis 会在内部自动将 zipmap 替换成正常的 hash 实现. 这个限制可以在配置文件中指定

hash-max-zipmap-entries 64 #配置字段最多 64 个

hash-max-zipmap-value 512 #配置 value 最大为 512 字节

2.3.1 hset

设置 hash field 为指定值，如果 key 不存在，则先创建。

```
redis 127.0.0.1:6379> hset myhash field1 Hello
(integer) 1
redis 127.0.0.1:6379>
```

2.3.2 hsetnx

设置 hash field 为指定值，如果 key 不存在，则先创建。如果 field 已经存在，返回 0，nx 是 not exist 的意思。

```
redis 127.0.0.1:6379> hsetnx myhash field "Hello"
(integer) 1
redis 127.0.0.1:6379> hsetnx myhash field "Hello"
(integer) 0
redis 127.0.0.1:6379>
```

第一次执行是成功的，但第二次执行相同的命令失败，原因是 field 已经存在了。

2.3.3 hmset

同时设置 hash 的多个 field。

```
redis 127.0.0.1:6379> hmset myhash field1 Hello field2 World
OK
redis 127.0.0.1:6379>
```

2.3.4 hget

获取指定的 hash field。

```
redis 127.0.0.1:6379> hget myhash field1
"Hello"
redis 127.0.0.1:6379> hget myhash field2
"World"
redis 127.0.0.1:6379> hget myhash field3
(nil)
redis 127.0.0.1:6379>
```

由于数据库没有 field3，所以取到的是一个空值 nil

2.3.5 hmget

获取全部指定的 hash filed。

```
redis 127.0.0.1:6379> hmget myhash field1 field2 field3
1) "Hello"
2) "World"
3) (nil)
redis 127.0.0.1:6379>
```

由于数据库没有 field3，所以取到的是一个空值 nil

2.3.6 hincrby

指定的 hash filed 加上给定值。

```
redis 127.0.0.1:6379> hset myhash field3 20
(integer) 1
redis 127.0.0.1:6379> hget myhash field3
"20"
redis 127.0.0.1:6379> hincrby myhash field3 -8
(integer) 12
redis 127.0.0.1:6379> hget myhash field3
"12"
redis 127.0.0.1:6379>
```

在本例中我们将 field3 的值从 20 降到了 12，即做了一个减 8 的操作。

2.3.7 hexists

测试指定 field 是否存在。

```
redis 127.0.0.1:6379> hexists myhash field1
(integer) 1
```

```
redis 127.0.0.1:6379> hexists myhash field9
(integer) 0
redis 127.0.0.1:6379>
```

通过上例可以说明 field1 存在，但 field9 是不存在的。

2.3.8 hlen

返回指定 hash 的 field 数量。

```
redis 127.0.0.1:6379> hlen myhash
(integer) 4
redis 127.0.0.1:6379>
```

通过上例可以看到 myhash 中有 4 个 field。

2.3.9 hdel

返回指定 hash 的 field 数量。

```
redis 127.0.0.1:6379> hlen myhash
(integer) 4
redis 127.0.0.1:6379> hdel myhash field1
(integer) 1
redis 127.0.0.1:6379> hlen myhash
(integer) 3
redis 127.0.0.1:6379>
```

2.3.10 hkeys

返回 hash 的所有 field。

```
redis 127.0.0.1:6379> hkeys myhash
1) "field2"
2) "field"
3) "field3"
redis 127.0.0.1:6379>
```

说明这个 hash 中有 3 个 field

2.3.11 hvals

返回 hash 的所有 value。

```
redis 127.0.0.1:6379> hvals myhash
1) "World"
2) "Hello"
3) "12"
```

```
redis 127.0.0.1:6379>
```

说明这个 hash 中有 3 个 field

2.3.12 hgetall

获取某个 hash 中全部的 filed 及 value。

```
redis 127.0.0.1:6379> hgetall myhash
1) "field2"
2) "World"
3) "field"
4) "Hello"
5) "field3"
6) "12"
redis 127.0.0.1:6379>
```

可见，一下子将 myhash 中所有的 field 及对应的 value 都取出来了。

2.4 lists 类型及操作

list 是一个链表结构，主要功能是 push、pop、获取一个范围的所有值等等，操作中 key 理解为链表的名字。

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。链表的最大长度是(2 的 32 次方)。我们可以通过 push,pop 操作从链表的头部或者尾部添加删除元素。这使得 list 既可以用作栈，也可以用作队列。

有意思的是 list 的 pop 操作还有阻塞版本的，当我们[lr]pop 一个 list 对象时，如果 list 是空，或者不存在，会立即返回 nil。但是阻塞版本的 b[lr]pop 可以则可以阻塞，当然可以加超时时间，超时后也会返回 nil。为什么要阻塞版本的 pop 呢，主要是为了避免轮询。举个简单的例子如果我们用 list 来实现一个工作队列。执行任务的 thread 可以调用阻塞版本的 pop 去获取任务这样就可以避免轮询去检查是否有任务存在。当任务来时候工作线程可以立即返回，也可以避免轮询带来的延迟。说了这么多，接下来看一下实际操作的方法吧:

2.4.1 lpush

在 key 对应 list 的头部添加字符串元素

```
redis 127.0.0.1:6379> lpush mylist "world"
(integer) 1
redis 127.0.0.1:6379> lpush mylist "hello"
(integer) 2
redis 127.0.0.1:6379> lrange mylist 0 -1
1) "hello"
```

```
2) "world"
redis 127.0.0.1:6379>
```

在此处我们先插入了一个 world，然后在 world 的头部插入了一个 hello。其中 lrange 是用于取 mylist 的内容。

2.4.2 rpush

在 key 对应 list 的尾部添加字符串元素

```
redis 127.0.0.1:6379> rpush mylist2 "hello"
(integer) 1
redis 127.0.0.1:6379> rpush mylist2 "world"
(integer) 2
redis 127.0.0.1:6379> lrange mylist2 0 -1
1) "hello"
2) "world"
redis 127.0.0.1:6379>
```

在此处我们先插入了一个 hello，然后在 hello 的尾部插入了一个 world。

2.4.3 linsert

在 key 对应 list 的特定位置之前或之后添加字符串元素

```
redis 127.0.0.1:6379> rpush mylist3 "hello"
(integer) 1
redis 127.0.0.1:6379> rpush mylist3 "world"
(integer) 2
redis 127.0.0.1:6379> linsert mylist3 before "world" "there"
(integer) 3
redis 127.0.0.1:6379> lrange mylist3 0 -1
1) "hello"
2) "there"
3) "world"
redis 127.0.0.1:6379>
```

在此处我们先插入了一个 hello，然后在 hello 的尾部插入了一个 world，然后又在 world 的前面插入了 there。

2.4.4 lset

设置 list 中指定下标的元素值(下标从 0 开始)

```
redis 127.0.0.1:6379> rpush mylist4 "one"
(integer) 1
redis 127.0.0.1:6379> rpush mylist4 "two"
```

```
(integer) 2
redis 127.0.0.1:6379> rpush mylist4 "three"
(integer) 3
redis 127.0.0.1:6379> lset mylist4 0 "four"
OK
redis 127.0.0.1:6379> lset mylist4 -2 "five"
OK
redis 127.0.0.1:6379> lrange mylist4 0 -1
1) "four"
2) "five"
3) "three"
redis 127.0.0.1:6379>
```

在此处我们依次插入了 one,two,three，然后将标是 0 的值设置为 four，再将下标是-2 的值设置为 five。

2.5.5 lrem

从 key 对应 list 中删除 count 个和 value 相同的元素。

count>0 时，按从头到尾的顺序删除，具体如下:

```
redis 127.0.0.1:6379> rpush mylist5 "hello"
(integer) 1
redis 127.0.0.1:6379> rpush mylist5 "hello"
(integer) 2
redis 127.0.0.1:6379> rpush mylist5 "foo"
(integer) 3
redis 127.0.0.1:6379> rpush mylist5 "hello"
(integer) 4
redis 127.0.0.1:6379> lrem mylist5 2 "hello"
(integer) 2
redis 127.0.0.1:6379> lrange mylist5 0 -1
1) "foo"
2) "hello"
redis 127.0.0.1:6379>
```

count<0 时，按从尾到头的顺序删除，具体如下:

```
redis 127.0.0.1:6379> rpush mylist6 "hello"
(integer) 1
redis 127.0.0.1:6379> rpush mylist6 "hello"
(integer) 2
redis 127.0.0.1:6379> rpush mylist6 "foo"
(integer) 3
redis 127.0.0.1:6379> rpush mylist6 "hello"
(integer) 4
```

```
redis 127.0.0.1:6379> lrem mylist6 -2 "hello"
(integer) 2
redis 127.0.0.1:6379> lrange mylist6 0 -1
1) "hello"
2) "foo"
redis 127.0.0.1:6379>
```

count=0 时，删除全部，具体如下:

```
redis 127.0.0.1:6379> rpush mylist7 "hello"
(integer) 1
redis 127.0.0.1:6379> rpush mylist7 "hello"
(integer) 2
redis 127.0.0.1:6379> rpush mylist7 "foo"
(integer) 3
redis 127.0.0.1:6379> rpush mylist7 "hello"
(integer) 4
redis 127.0.0.1:6379> lrem mylist7 0 "hello"
(integer) 3
redis 127.0.0.1:6379> lrange mylist7 0 -1
1) "foo"
redis 127.0.0.1:6379>
```

2.4.6 ltrim

保留指定 key 的值范围内的数据

```
redis 127.0.0.1:6379> rpush mylist8 "one"
(integer) 1
redis 127.0.0.1:6379> rpush mylist8 "two"
(integer) 2
redis 127.0.0.1:6379> rpush mylist8 "three"
(integer) 3
redis 127.0.0.1:6379> rpush mylist8 "four"
(integer) 4
redis 127.0.0.1:6379> ltrim mylist8 1 -1
OK
redis 127.0.0.1:6379> lrange mylist8 0 -1
1) "two"
2) "three"
3) "four"
redis 127.0.0.1:6379>
```

2.4.7 lpop

从 list 的头部删除元素，并返回删除元素

```
redis 127.0.0.1:6379> lrange mylist 0 -1
1) "hello"
2) "world"
redis 127.0.0.1:6379> lpop mylist
"hello"
redis 127.0.0.1:6379> lrange mylist 0 -1
1) "world"
redis 127.0.0.1:6379>
```

2.4.8 rpop

从 list 的尾部删除元素，并返回删除元素

```
redis 127.0.0.1:6379> lrange mylist2 0 -1
1) "hello"
2) "world"
redis 127.0.0.1:6379> rpop mylist2
"world"
redis 127.0.0.1:6379> lrange mylist2 0 -1
1) "hello"
redis 127.0.0.1:6379>
```

2.4.9 rpoplpush

从第一个 list 的尾部移除元素并添加到第二个 list 的头部,最后返回被移除的元素值，整个操作是原子的.如果第一个 list 是空或者不存在返回 nil

```
redis 127.0.0.1:6379> lrange mylist5 0 -1
1) "three"
2) "foo"
3) "hello"
redis 127.0.0.1:6379> lrange mylist6 0 -1
1) "hello"
2) "foo"
redis 127.0.0.1:6379> rpoplpush mylist5 mylist6
"hello"
redis 127.0.0.1:6379> lrange mylist5 0 -1
1) "three"
2) "foo"
redis 127.0.0.1:6379> lrange mylist6 0 -1
```

```
1) "hello"
2) "hello"
3) "foo"
redis 127.0.0.1:6379>
```

2.4.10 lindex

返回名称为 key 的 list 中 index 位置的元素

```
redis 127.0.0.1:6379> lrange mylist5 0 -1
1) "three"
2) "foo"
redis 127.0.0.1:6379> lindex mylist5 0
"three"
redis 127.0.0.1:6379> lindex mylist5 1
"foo"
redis 127.0.0.1:6379>
```

2.4.11 llen

返回 key 对应 list 的长度

```
redis 127.0.0.1:6379> llen mylist5
(integer) 2
redis 127.0.0.1:6379>
```

2.5 sets 类型及操作

set 是集合，和我们数学中的集合概念相似，对集合的操作有添加删除元素，有对多个集合求交并差等操作，操作中 key 理解为集合的名字。

Redis 的 set 是 string 类型的无序集合。set 元素最大可以包含(2 的 32 次方)个元素。

set 的是通过 hash table 实现的，所以添加、删除和查找的复杂度都是 O(1)。hash table 会随着添加或者删除自动的调整大小。需要注意的是调整 hash table 大小时候需要同步（获取写锁）会阻塞其他读写操作，可能不久后就会改用跳表（skip list）来实现，跳表已经在 sorted set 中使用了。关于 set 集合类型除了基本的添加删除操作，其他有用的操作还包含集合的取并集(union)，交集(intersection)，差集(difference)。通过这些操作可以很容易的实现 sns 中的好友推荐和 blog 的 tag 功能。下面详细介绍 set 相关命令:

2.5.1 sadd

向名称为 key 的 set 中添加元素

```
redis 127.0.0.1:6379> sadd myset "hello"
(integer) 1
redis 127.0.0.1:6379> sadd myset "world"
(integer) 1
redis 127.0.0.1:6379> sadd myset "world"
(integer) 0
redis 127.0.0.1:6379> smembers myset
1) "world"
2) "hello"
redis 127.0.0.1:6379>
```

本例中，我们向 myset 中添加了三个元素，但由于第三个元素跟第二个元素是相同的，所以第三个元素没有添加成功，最后我们用 smembers 来查看 myset 中的所有元素。

2.5.2 srem

删除名称为 key 的 set 中的元素 member

```
redis 127.0.0.1:6379> sadd myset2 "one"
(integer) 1
redis 127.0.0.1:6379> sadd myset2 "two"
(integer) 1
redis 127.0.0.1:6379> sadd myset2 "three"
(integer) 1
redis 127.0.0.1:6379> srem myset2 "one"
(integer) 1
redis 127.0.0.1:6379> srem myset2 "four"
(integer) 0
redis 127.0.0.1:6379> smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379>
```

本例中，我们向 myset2 中添加了三个元素后，再调用 srem 来删除 one 和 four，但由于元素中没有 four 所以，此条 srem 命令执行失败。

2.5.3 spop

随机返回并删除名称为 key 的 set 中一个元素

```
redis 127.0.0.1:6379> sadd myset3 "one"
(integer) 1
redis 127.0.0.1:6379> sadd myset3 "two"
(integer) 1
redis 127.0.0.1:6379> sadd myset3 "three"
(integer) 1
```

```
redis 127.0.0.1:6379> spop myset3
"three"
redis 127.0.0.1:6379> smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379>
```

本例中，我们向 myset3 中添加了三个元素后，再调用 spop 来随机删除一个元素，可以看到 three 元素被删除了。

2.5.4 sdiff

返回所有给定 key 与第一个 key 的差集

```
redis 127.0.0.1:6379>   smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379>   smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379> sdiff myset2 myset3
1) "three"
redis 127.0.0.1:6379>
```

本例中，我们可以看到 myset2 中的元素与 myset3 中不同的只是 three，所以只有 three 被查出来了，而不是 three 和 one，因为 one 是 myset3 的元素。

我们也可以将 myset2 和 myset3 换个顺序来看一下结果:

```
redis 127.0.0.1:6379> sdiff myset3 myset2
1) "one"
redis 127.0.0.1:6379>
```

这个结果中只显示了，myset3 中的元素与 myset2 中不同的元素。

2.5.5 sdiffstore

返回所有给定 key 与第一个 key 的差集，并将结果存为另一个 key

```
redis 127.0.0.1:6379> smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379> smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379> sdiffstore myset4 myset2 myset3
(integer) 1
redis 127.0.0.1:6379> smembers myset4
```

```
1) "three"
redis 127.0.0.1:6379>
```

2.5.6 sinter

返回所有给定 key 的交集

```
redis 127.0.0.1:6379> smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379> smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379> sinter myset2 myset3
1) "two"
redis 127.0.0.1:6379>
```

通过本例的结果可以看出, myset2 和 myset3 的交集 two 被查出来了

2.5.7 sinterstore

返回所有给定 key 的交集，并将结果存为另一个 key

```
redis 127.0.0.1:6379> smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379> smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379> sinterstore myset5 myset2 myset3
(integer) 1
redis 127.0.0.1:6379> smembers myset5
1) "two"
redis 127.0.0.1:6379>
```

通过本例的结果可以看出, myset2 和 myset3 的交集被保存到 myset5 中了

2.5.8 sunion

返回所有给定 key 的并集

```
redis 127.0.0.1:6379> smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379> smembers myset3
1) "two"
```

```
2) "one"
redis 127.0.0.1:6379> sunion myset2 myset3
1) "three"
2) "one"
3) "two"
redis 127.0.0.1:6379>
```

通过本例的结果可以看出, myset2 和 myset3 的并集被查出来了

2.5.9 sunionstore

返回所有给定 key 的并集，并将结果存为另一个 key

```
redis 127.0.0.1:6379> smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379> smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379> sunionstore myset6 myset2 myset3
(integer) 3
redis 127.0.0.1:6379> smembers myset6
1) "three"
2) "one"
3) "two"
redis 127.0.0.1:6379>
```

通过本例的结果可以看出, myset2 和 myset3 的并集被保存到 myset6 中了

2.5.10 smove

从第一个 key 对应的 set 中移除 member 并添加到第二个对应 set 中

```
redis 127.0.0.1:6379> smembers myset2
1) "three"
2) "two"
redis 127.0.0.1:6379> smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379> smove myset2 myset7 three
(integer) 1
redis 127.0.0.1:6379> smembers myset7
1) "three"
redis 127.0.0.1:6379>
```

通过本例可以看到，myset2 的 three 被移到 myset7 中了

2.5.11 scard

返回名称为 key 的 set 的元素个数

```
redis 127.0.0.1:6379> scard myset2
(integer) 1
redis 127.0.0.1:6379>
```

通过本例可以看到，myset2 的成员数量为 1

2.5.12 sismember

测试 member 是否是名称为 key 的 set 的元素

```
redis 127.0.0.1:6379> smembers myset2
1) "two"
redis 127.0.0.1:6379> sismember myset2 two
(integer) 1
redis 127.0.0.1:6379> sismember myset2 one
(integer) 0
redis 127.0.0.1:6379>
```

通过本例可以看到，two 是 myset2 的成员，而 one 不是。

2.5.13 srandmember

随机返回名称为 key 的 set 的一个元素，但是不删除元素

```
redis 127.0.0.1:6379> smembers myset3
1) "two"
2) "one"
redis 127.0.0.1:6379> srandmember myset3
"two"
redis 127.0.0.1:6379> srandmember myset3
"one"
redis 127.0.0.1:6379>
```

2.6 sorted sets 类型及操作

sorted set 是 set 的一个升级版本，它在 set 的基础上增加了一个顺序属性，这一属性在添加修改元素的时候可以指定，每次指定后，zset 会自动重新按新的值调整顺序。可以理解为有两列的 mysql 表，一列存 value，一列存顺序。操作中 key 理解为 zset 的名字。

和 set 一样 sorted set 也是 string 类型元素的集合，不同的是每个元素都会关联一个 double 类型的 score。sorted set 的实现是 skip list 和 hash table 的混合体。

当元素被添加到集合中时，一个元素到 score 的映射被添加到 hash table 中，所以给定一个元素获取 score 的开销是 O(1),另一个 score 到元素的映射被添加到 skip list，并按照 score 排序，所以就可以有序的获取集合中的元素。添加，删除操作开销都是 O(log(N))和 skip list 的开销一致,redis 的 skip list 实现用的是双向链表,这样就可以逆序从尾部取元素。sorted set 最经常的使用方式应该是作为索引来使用.我们可以把要排序的字段作为 score 存储，对象的 id 当元素存储。下面是 sorted set 相关命令

2.6.1 zadd

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

```
redis 127.0.0.1:6379> zadd myzset 1 "one"
(integer) 1
redis 127.0.0.1:6379> zadd myzset 2 "two"
(integer) 1
redis 127.0.0.1:6379> zadd myzset 3 "two"
(integer) 0
redis 127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "3"
redis 127.0.0.1:6379>
```

本例中我们向 myzset 中添加了 one 和 two，并且 two 被设置了 2 次，那么将以最后一次的设置为准，最后我们将所有元素都显示出来并显示出了元素的 score。

2.6.2 zrem

删除名称为 key 的 zset 中的元素 member

```
redis 127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "3"
redis 127.0.0.1:6379> zrem myzset two
(integer) 1
redis 127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "one"
2) "1"
redis 127.0.0.1:6379>
```

可以看到 two 被删除了

2.6.3 zincrby

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

```
redis 127.0.0.1:6379> zadd myzset2 1 "one"
(integer) 1
redis 127.0.0.1:6379> zadd myzset2 2 "two"
(integer) 1
redis 127.0.0.1:6379> zincrby myzset2 2 "one"
"3"
redis 127.0.0.1:6379> zrange myzset2 0 -1 withscores
1) "two"
2) "2"
3) "one"
4) "3"
redis 127.0.0.1:6379>
```

本例中将 one 的 score 从 1 增加了 2，增加到了 3

2.6.4 zrank

返回名称为 key 的 zset 中 member 元素的排名(按 score 从小到大排序)即下标

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
7) "five"
8) "5"
redis 127.0.0.1:6379> zrank myzset3 two
(integer) 1
redis 127.0.0.1:6379>
```

本例中将 two 的下标是 1，我这里取的是下标，而不是 score

2.6.5 zrevrank

返回名称为 key 的 zset 中 member 元素的排名(按 score 从大到小排序)即下标

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
```

```
3) "two"
4) "2"
5) "three"
6) "3"
7) "five"
8) "5"
redis 127.0.0.1:6379> zrevrank myzset3 two
(integer) 2
redis 127.0.0.1:6379>
```

按从大到小排序的话 two 是第三个元素，下标是 2

2.6.6 zrevrange

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

```
redis 127.0.0.1:6379> zrevrange myzset3 0 -1 withscores
1) "five"
2) "5"
3) "three"
4) "3"
5) "two"
6) "2"
7) "one"
8) "1"
redis 127.0.0.1:6379>
```

首先按 score 从大到小排序，再取出全部元素

2.6.7 zrangebyscore

返回集合中 score 在给定区间的元素

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
7) "five"
8) "5"
redis 127.0.0.1:6379> zrangebyscore myzset3 2 3 withscores
1) "two"
2) "2"
3) "three"
```

```
4) "3"
redis 127.0.0.1:6379>
```

本例中，返回了 score 在 2~3 区间的元素

2.6.8 zcount

返回集合中 score 在给定区间的数量

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
7) "five"
8) "5"
redis 127.0.0.1:6379> zcount myzset3 2 3
(integer) 2
redis 127.0.0.1:6379>
```

本例中，计算了 score 在 2~3 之间的元素数目

2.6.9 zcard

返回集合中元素个数

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
7) "five"
8) "5"
redis 127.0.0.1:6379> zcard myzset3
(integer) 4
redis 127.0.0.1:6379>
```

从本例看出 myzset3 这个集全的元素数量是 4

2.6.10 zscore

返回给定元素对应的 score

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
7) "five"
8) "5"
redis 127.0.0.1:6379> zscore myzset3 two
"2"
redis 127.0.0.1:6379>
```

此例中我们成功的将 two 的 score 取出来了。

2.6.11 zremrangebyrank

删除集合中排名在给定区间的元素

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
7) "five"
8) "5"
redis 127.0.0.1:6379> zremrangebyrank myzset3 3 3
(integer) 1
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
redis 127.0.0.1:6379>
```

在本例中我们将 myzset3 中按从小到大排序结果的下标为 3 的元素删除了。

2.6.12 zremrangebyscore

删除集合中 score 在给定区间的元素

```
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
```

```
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
redis 127.0.0.1:6379> zremrangebyscore myzset3 1 2
(integer) 2
redis 127.0.0.1:6379> zrange myzset3 0 -1 withscores
1) "three"
2) "3"
redis 127.0.0.1:6379>
```

在本例中我们将 myzset3 中按从小到大排序结果的 score 在 1~2 之间的元素删除了。

第三章、Redis 常用命令

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，比如各类语言包，这些命令都有对应的方法。下面将 Redis 提供的命令做一总结。

3.1 键值相关命令

3.1.1 keys

返回满足给定 pattern 的所有 key

```
redis 127.0.0.1:6379> keys *
 1) "myzset2"
 2) "myzset3"
 3) "mylist"
 4) "myset2"
 5) "myset3"
 6) "myset4"
 7) "k_zs_1"
 8) "myset5"
 9) "myset6"
10) "myset7"
11) "myhash"
12) "myzset"
13) "age"
```

```
14) "myset"
15) "mylist5"
16) "mylist6"
17) "mylist7"
18) "mylist8"
```

用表达式*，代表取出所有的 key

```
redis 127.0.0.1:6379> keys mylist*
1) "mylist"
2) "mylist5"
3) "mylist6"
4) "mylist7"
5) "mylist8"
redis 127.0.0.1:6379>
```

用表达式 mylist*，代表取出所有以 mylist 开头的 key

3.1.2 exists

确认一个 key 是否存在

```
redis 127.0.0.1:6379> exists HongWan
(integer) 0
redis 127.0.0.1:6379> exists age
(integer) 1
redis 127.0.0.1:6379>
```

从结果来数据库中不存在 HongWan 这个 key，但是 age 这个 key 是存在的

3.1.3 del

删除一个 key

```
redis 127.0.0.1:6379> del age
(integer) 1
redis 127.0.0.1:6379> exists age
(integer) 0
redis 127.0.0.1:6379>
```

从结果来数据库中不存在 HongWan 这个 key，但是 age 这个 key 是存在的

3.1.4 expire

设置一个 key 的过期时间(单位:秒)

```
redis 127.0.0.1:6379> expire addr 10
```

```
(integer) 1
redis 127.0.0.1:6379> ttl addr
(integer) 8
redis 127.0.0.1:6379> ttl addr
(integer) 1
redis 127.0.0.1:6379> ttl addr
(integer) -1
redis 127.0.0.1:6379>
```

在本例中，我们设置 addr 这个 key 的过期时间是 10 秒，然后我们不断的用 ttl 来获取这个 key 的有效时长，直至为-1 说明此值已过期

3.1.5 move

将当前数据库中的 key 转移到其它数据库中

```
redis 127.0.0.1:6379> select 0
OK
redis 127.0.0.1:6379> set age 30
OK
redis 127.0.0.1:6379> get age
"30"
redis 127.0.0.1:6379> move age 1
(integer) 1
redis 127.0.0.1:6379> get age
(nil)
redis 127.0.0.1:6379> select 1
OK
redis 127.0.0.1:6379[1]> get age
"30"
redis 127.0.0.1:6379[1]>
```

在本例中，我先显式的选择了数据库 0，然后在这个库中设置一个 key，接下来我们将这个 key 从数据库 0 移到数据库 1，之后我们确认在数据库 0 中无此 key 了，但在数据库 1 中存在这个 key，说明我们转移成功了

3.1.6 persist

移除给定 key 的过期时间

```
redis 127.0.0.1:6379[1]> expire age 300
(integer) 1
redis 127.0.0.1:6379[1]> ttl age
(integer) 294
redis 127.0.0.1:6379[1]> persist age
```

```
(integer) 1
redis 127.0.0.1:6379[1]> ttl age
(integer) -1
redis 127.0.0.1:6379[1]>
```

在这个例子中，我们手动的将未到过期时间的 key，成功设置为过期

3.1.7 randomkey

随机返回 key 空间的一个 key

```
redis 127.0.0.1:6379> randomkey
"mylist7"
redis 127.0.0.1:6379> randomkey
"mylist5"
redis 127.0.0.1:6379>
```

通过结果可以看到取 key 的规则是随机的

3.1.8 rename

重命名 key

```
redis 127.0.0.1:6379[1]> keys *
1) "age"
redis 127.0.0.1:6379[1]> rename age age_new
OK
redis 127.0.0.1:6379[1]> keys *
1) "age_new"
redis 127.0.0.1:6379[1]>
```

age 成功的被我们改名为 age_new 了

3.1.9 type

返回值的类型

```
redis 127.0.0.1:6379> type addr
string
redis 127.0.0.1:6379> type myzset2
zset
redis 127.0.0.1:6379> type mylist
list
redis 127.0.0.1:6379>
```

这个方法可以非常简单的判断出值的类型

3.2 服务器相关命令

3.2.1 ping

测试连接是否存活

```
redis 127.0.0.1:6379> ping
PONG
//执行下面命令之前，我们停止 redis 服务器
redis 127.0.0.1:6379> ping
Could not connect to Redis at 127.0.0.1:6379: Connection refused
//执行下面命令之前，我们启动 redis 服务器
not connected> ping
PONG
redis 127.0.0.1:6379>
```

第一个 ping 时，说明此连接正常
第二个 ping 之前，我们将 redis 服务器停止，那么 ping 是失败的
第三个 ping 之前，我们将 redis 服务器启动，那么 ping 是成功的

3.2.2 echo

在命令行打印一些内容

```
redis 127.0.0.1:6379> echo HongWan
"HongWan"
redis 127.0.0.1:6379>
```

3.2.3 select

选择数据库。Redis 数据库编号从 0~15，我们可以选择任意一个数据库来进行数据的存取。

```
redis 127.0.0.1:6379> select 1
OK
redis 127.0.0.1:6379[1]> select 16
(error) ERR invalid DB index
redis 127.0.0.1:6379[16]>
```

当选择 16 时，报错，说明没有编号为 16 的这个数据库

3.2.4 quit

退出连接。

```
redis 127.0.0.1:6379> quit
[root@localhost redis-2.2.12]#
```

3.2.5 dbsize

返回当前数据库中 key 的数目。

```
redis 127.0.0.1:6379> dbsize
(integer) 18
redis 127.0.0.1:6379>
```

结果说明此库中有 18 个 key

3.2.6 info

获取服务器的信息和统计。

```
redis 127.0.0.1:6379> info
redis_version:2.2.12
redis_git_sha1:00000000
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
process_id:28480
uptime_in_seconds:2515
uptime_in_days:0
.
.
.
redis 127.0.0.1:6379>
```

此结果用于说明服务器的基础信息，包括版本、启动时间等。

3.2.7 monitor

实时转储收到的请求。

```
redis 127.0.0.1:6379> config get dir
1) "dir"
2) "/root/4setup/redis-2.2.12"
redis 127.0.0.1:6379>
```

从结果可以看出，此服务器目前接受了命令"keys *"和"get addr"。

3.2.8 config get

获取服务器配置信息。

```
redis 127.0.0.1:6379> config get dir
1) "dir"
2) "/root/4setup/redis-2.2.12"
```

```
redis 127.0.0.1:6379>
```

本例中我们获取了 dir 这个参数配置的值，如果想获取全部参数据的配置值也很简单，只需执行”config get *”即可将全部的值都显示出来。

3.2.9 flushdb

删除当前选择数据库中的所有 key。

```
redis 127.0.0.1:6379> dbsize
(integer) 18
redis 127.0.0.1:6379> flushdb
OK
redis 127.0.0.1:6379> dbsize
(integer) 0
redis 127.0.0.1:6379>
```

在本例中我们将 0 号数据库中的 key 都清除了。

3.2.10 flushall

删除所有数据库中的所有 key。

```
redis 127.0.0.1:6379[1]> dbsize
(integer) 1
redis 127.0.0.1:6379[1]> select 0
OK
redis 127.0.0.1:6379> flushall
OK
redis 127.0.0.1:6379> select 1
OK
redis 127.0.0.1:6379[1]> dbsize
(integer) 0
redis 127.0.0.1:6379[1]>
```

在本例中我们先查看了一个 1 号数据库中有一个 key，然后我切换到 0 号库执行 flushall 命令，结果 1 号库中的 key 也被清除了，说是此命令工作正常。

第四章 Redis 高级实用特性

4.1 安全性

设置客户端连接后进行任何其他指定前需要使用的密码。

警告：因为 redis 速度相当快，所以在一台比较好的服务器下，一个外部的用户可以在一秒钟进行 150K 次的密码尝试，这意味着你需要指定非常非常强大的密码来防止暴力破解。

下面我们做一个实验，说明 redis 的安全性是如何实现的。

```
# requirepass foobared
requirepass beijing
```

我们设置了连接的口令是 beijing

那么们启动一个客户端试一下:

```
[root@localhost redis-2.2.12]# src/redis-cli
redis 127.0.0.1:6379> keys *
(error) ERR operation not permitted
redis 127.0.0.1:6379>
```

说明权限太小，我们可以当前的这个窗口中设置口令

```
redis 127.0.0.1:6379> auth beijing
OK
redis 127.0.0.1:6379> keys *
1) "name"
redis 127.0.0.1:6379>
```

我们还可以在连接到服务器期间就指定一个口令，如下:

```
[root@localhost redis-2.2.12]# src/redis-cli -a beijing
redis 127.0.0.1:6379> keys *
1) "name"
redis 127.0.0.1:6379>
```

可以看到我们在连接的时候就可以指定一个口令。

4.2 主从复制

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。

4.2.1 redis 主从复制特点:

(1)、master 可以拥有多个 slave
(2)、多个 slave 可以连接同一个 master 外，还可以连接到其他 slave
(3)、主从复制不会阻塞 master，在同步数据时，master 可以继续处理 client 请求
(4)、提高系统的伸缩性

4.2.2 redis 主从复制过程:

当配置好 slave 后，slave 与 master 建立连接，然后发送 sync 命令。无论是第一次连接还是

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。后台进程完成写文件后，master 就发送文件给 slave，slave 将文件保存到硬盘上，再加载到内存中，接着 master 就会把缓存的命令转发给 slave，后续 master 将收到的写命令发送给 slave。如果 master 同时收到多个 slave 发来的同步连接命令，master 只会启动一个进程来写数据库镜像，然后发送给所有的 slave。

4.2.3 如何配置

配置 slave 服务器很简单，只需要在 slave 的配置文件中加入如下配置

```
slaveof 192.168.1.1 6379    #指定 master 的 ip 和端口
```

下面我们做一个实验来演示如何搭建一个主从环境:

```
# slaveof <masterip> <masterport>
slaveof localhost 6379
```

我们在一台机器上启动主库(端口 6379)，从库(端口 6378)

启动后主库控制台日志如下:

```
[root@localhost redis-2.2.12]# src/redis-server redis.conf
[7064] 09 Aug 20:13:12 * Server started, Redis version 2.2.12
[7064] 09 Aug 20:13:12 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
[7064] 09 Aug 20:13:12 * The server is now ready to accept connections on port 6379
[7064] 09 Aug 20:13:13 - 0 clients connected (0 slaves), 539512 bytes in use
[7064] 09 Aug 20:13:18 - 0 clients connected (0 slaves), 539512 bytes in use
[7064] 09 Aug 20:13:20 - Accepted 127.0.0.1:37789
[7064] 09 Aug 20:13:20 * Slave ask for synchronization
[7064] 09 Aug 20:13:20 * Starting BGSAVE for SYNC
[7064] 09 Aug 20:13:20 * Background saving started by pid 7067
[7067] 09 Aug 20:13:20 * DB saved on disk
[7064] 09 Aug 20:13:20 * Background saving terminated with success
[7064] 09 Aug 20:13:20 * Synchronization with slave succeeded
[7064] 09 Aug 20:13:23 - 0 clients connected (1 slaves), 547380 bytes in use
```

启动后从库控制台日志如下:

```
[root@localhost redis-2.2.12]# src/redis-server redis.slave
[7066] 09 Aug 20:13:20 * Server started, Redis version 2.2.12
[7066] 09 Aug 20:13:20 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
[7066] 09 Aug 20:13:20 * The server is now ready to accept connections on port 6378
[7066] 09 Aug 20:13:20 - 0 clients connected (0 slaves), 539548 bytes in use
[7066] 09 Aug 20:13:20 * Connecting to MASTER...
```

```
[7066] 09 Aug 20:13:20 * MASTER <-> SLAVE sync started: SYNC sent
[7066] 09 Aug 20:13:20 * MASTER <-> SLAVE sync: receiving 10 bytes from master
[7066] 09 Aug 20:13:20 * MASTER <-> SLAVE sync: Loading DB in memory
[7066] 09 Aug 20:13:20 * MASTER <-> SLAVE sync: Finished with success
[7068] 09 Aug 20:13:20 * SYNC append only file rewrite performed
[7066] 09 Aug 20:13:20 * Background append only file rewriting started by pid 7068
[7066] 09 Aug 20:13:21 * Background append only file rewriting terminated with success
[7066] 09 Aug 20:13:21 * Parent diff flushed into the new append log file with success (0 bytes)
[7066] 09 Aug 20:13:21 * Append only file successfully rewritten.
[7066] 09 Aug 20:13:21 * The new append only file was selected for future appends.
[7066] 09 Aug 20:13:25 - 1 clients connected (0 slaves), 547396 bytes in use
```

我们在主库上设置一对键值对

```
redis 127.0.0.1:6379> set name HongWan
OK
redis 127.0.0.1:6379>
```

在从库上取一下这个键

```
redis 127.0.0.1:6378> get name
"HongWan"
redis 127.0.0.1:6378>
```

说明主从是同步正常的.

那么我们如何判断哪个是主哪个是从呢?我们只需调用 info 这个命令就可以得到主从的信息了,我们在从库上执行 info 命令

```
redis 127.0.0.1:6378> info
.
.
.
role:slave
master_host:localhost
master_port:6379
master_link_status:up
master_last_io_seconds_ago:10
master_sync_in_progress:0
db0:keys=1,expires=0
redis 127.0.0.1:6378>
```

里面有一个角色标识,来判断是主库还是从库,对于本例是一个从库，同时还有一个 master_link_status 用于标明主从是否异步，如果此值=up，说明同步正常；如果此值=down，说明同步异步；

db0:keys=1,expires=0， 用于说明数据库有几个 key,以及过期 key 的数量。

4.3 事务控制

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。

4.3.1 简单事务控制

下面可以看一个例子

```
redis 127.0.0.1:6379> get age
"33"
redis 127.0.0.1:6379> multi
OK
redis 127.0.0.1:6379> set age 10
QUEUED
redis 127.0.0.1:6379> set age 20
QUEUED
redis 127.0.0.1:6379> exec
1) OK
2) OK
redis 127.0.0.1:6379> get age
"20"
redis 127.0.0.1:6379>
```

从这个例子我们可以看到 2 个 set age 命令发出后并没执行而是被放到了队列中。调用 exec 后 2 个命令才被连续的执行，最后返回的是两条命令执行后的结果。

4.3.2 如何取消一个事务

我们可以调用 discard 命令来取消一个事务，让事务回滚。接着上面例子

```
redis 127.0.0.1:6379> get age
"20"
redis 127.0.0.1:6379> multi
OK
redis 127.0.0.1:6379> set age 30
QUEUED
redis 127.0.0.1:6379> set age 40
QUEUED
```

```
redis 127.0.0.1:6379> discard
OK
redis 127.0.0.1:6379> get age
"20"
redis 127.0.0.1:6379>
```

可以发现这次 2 个 set age 命令都没被执行。discard 命令其实就是清空事务的命令队列并退出事务上下文，也就是我们常说的事务回滚。

4.3.3 乐观锁复杂事务控制

在本小节开始前，我们有必要向读者朋友简单介绍一下乐观锁的概念，并举例说明乐观锁是怎么工作的。

乐观锁：大多数是基于数据版本（version）的记录机制实现的。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表添加一个“version”字段来实现读取出数据时，将此版本号一同读出，之后更新时，对此版本号加 1。此时，将提交数据的版本号与数据库表对应记录的当前版本号进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

乐观锁实例：假设数据库中帐户信息表中有一个 version 字段，当前值为 1；而当前帐户余额字段（balance）为$100。下面我们将用时序表的方式来为大家演示乐观锁的实现原理：

操作员 A	操作员 B
(1)、操作员 A 此时将用户信息读出（此时 version=1），并准备从其帐户余额中扣除$50（$100-$50）	(2)、在操作员 A 操作的过程中，操作员 B 也读入此用户信息（此时 version=1），并准备从其帐户余额中扣除$20（$100-$20）
(3)、操作员 A 完成了修改工作，将数据版本号加 1（此时 version=2），连同帐户扣除后余额（balance=$50），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2	
	(4)、操作员 B 完成了操作，也将版本号加 1（version=2）并试图向数据库提交数据（balance=$80），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2，数据库记录当前版本也为 2，不满足“提交版本必须大于记录当前版本才能执行更新”的乐观锁策略，因此，操作员 B 的提交被驳回

这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果来覆盖操作员 A 的操作结果的可能。

即然乐观锁比悲观锁要好很多，redis 是否也支持呢？答案是支持, redis 从 2.1.0 开始就支持

乐观锁了，可以显式的使用 watch 对某个 key 进行加锁，避免悲观锁带来的一系列问题。

Redis 乐观锁实例：假设有一个 age 的 key，我们开 2 个 session 来对 age 进行赋值操作，我们来看一下结果如何。

Session 1	Session 2
(1)第 1 步 redis 127.0.0.1:6379> get age "10" redis 127.0.0.1:6379> **watch age** OK redis 127.0.0.1:6379> multi OK redis 127.0.0.1:6379>	
	(2)第 2 步 redis 127.0.0.1:6379> **set age 30** OK redis 127.0.0.1:6379> get age "30" redis 127.0.0.1:6379>
(3)第 3 步 redis 127.0.0.1:6379> **set age 20** QUEUED redis 127.0.0.1:6379> exec (nil) redis 127.0.0.1:6379> get age **"30"** redis 127.0.0.1:6379>	

从以上实例可以看到在

第一步，Session 1 还没有来得及对 age 的值进行修改

第二步，Session 2 已经将 age 的值设为 30

第三步，Session 1 希望将 age 的值设为 20，但结果一执行返回是 nil，说明执行失败，之后我们再取一下 age 的值是 30，这是由于 Session 1 中对 age 加了乐观锁导致的。

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。下面将以一个实例的例子来说明这个问题:

```
redis 127.0.0.1:6379> get age
"30"
redis 127.0.0.1:6379> get name
```

```
"HongWan"
redis 127.0.0.1:6379> multi
OK
redis 127.0.0.1:6379> incr age
QUEUED
redis 127.0.0.1:6379> incr name
QUEUED
redis 127.0.0.1:6379> exec
1) (integer) 31
2) (error) ERR value is not an integer or out of range
redis 127.0.0.1:6379> get age
"31"
redis 127.0.0.1:6379> get name
"HongWan"
redis 127.0.0.1:6379>
```

从这个例子中可以看到，age 由于是个数字，那么它可以有自增运算，但是 name 是个字符串，无法对其进行自增运算，所以会报错，如果按传统关系型数据库的思路来讲，整个事务都会回滚，但是我们看到 redis 却是将可以执行的命令提交了，所以这个现象对于习惯于关系型数据库操作的朋友来说是很别扭的，这一点也是 redis 今天需要改进的地方。

4.4 持久化机制

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:

4.4.1 snapshotting 方式

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 的实时复制机制（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变更数据。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。
下面将演示各种场景的数据库持久化情况

```
redis 127.0.0.1:6379> set name HongWan
OK
redis 127.0.0.1:6379> get name
"HongWan"
redis 127.0.0.1:6379> shutdown
redis 127.0.0.1:6379> quit
```

我们先设置了一个 name 的键值对，然后正常关闭了数据库实例，数据是否被保存到磁盘了呢？我们来看一下服务器端是否有消息被记录下来了:

```
[6563] 09 Aug 18:58:58 * The server is now ready to accept connections on port 6379
[6563] 09 Aug 18:58:58 - 0 clients connected (0 slaves), 539540 bytes in use
[6563] 09 Aug 18:59:02 - Accepted 127.0.0.1:58005
[6563] 09 Aug 18:59:03 - 1 clients connected (0 slaves), 547368 bytes in use
[6563] 09 Aug 18:59:08 - 1 clients connected (0 slaves), 547424 bytes in use
[6563] 09 Aug 18:59:12 # User requested shutdown...
[6563] 09 Aug 18:59:12 * Saving the final RDB snapshot before exiting.
[6563] 09 Aug 18:59:12 * DB saved on disk
[6563] 09 Aug 18:59:12 # Redis is now ready to exit, bye bye...
[root@localhost redis-2.2.12]#
```

从日志可以看出，数据库做了一个存盘的操作，将内存的数据写入磁盘了。正常的话，磁盘上会产生一个 dump 文件，用于保存数据库快照，我们来验证一下:

```
[root@localhost redis-2.2.12]# ll
总计 188
-rw-rw-r-- 1 root root   9602 2011-07-22 00-RELEASENOTES
-rw-rw-r-- 1 root root     55 2011-07-22 BUGS
-rw-rw-r-- 1 root root 84050 2011-07-22 Changelog
drwxrwxr-x 2 root root   4096 2011-07-22 client-libraries
-rw-rw-r-- 1 root root    671 2011-07-22 CONTRIBUTING
-rw-rw-r-- 1 root root   1487 2011-07-22 COPYING
drwxrwxr-x 4 root root   4096 2011-07-22 deps
drwxrwxr-x 2 root root   4096 2011-07-22 design-documents
drwxrwxr-x 2 root root 12288 2011-07-22 doc
```

```
-rw-r--r-- 1 root root      26 08-09 18:59 dump.rdb
-rw-rw-r-- 1 root root     652 2011-07-22 INSTALL
-rw-rw-r-- 1 root root     337 2011-07-22 Makefile
-rw-rw-r-- 1 root root    1954 2011-07-22 README
-rw-rw-r-- 1 root root 19067 08-09 18:48 redis.conf
drwxrwxr-x 2 root root    4096 08-05 19:12 src
drwxrwxr-x 7 root root    4096 2011-07-22 tests
-rw-rw-r-- 1 root root     158 2011-07-22 TODO
drwxrwxr-x 2 root root    4096 2011-07-22 utils
[root@localhost redis-2.2.12]#
```

硬盘上已经产生了一个数据库快照了。这时侯我们再将 redis 启动，看键值还是否真的持久化到硬盘了。

```
redis 127.0.0.1:6379> keys *
1) "name"
redis 127.0.0.1:6379> get name
"HongWan"
redis 127.0.0.1:6379>
```

数据被完全持久化到硬盘了。

4.4.2 aof 方式

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        //收到写命令就立即写入磁盘，最慢，但是保证完全的持久化
appendfsync everysec        //每秒钟写入磁盘一次，在性能和持久化方面做了很好的折中
# appendfsync no            //完全依赖 os，性能最好,持久化没保证
```

接下来我们以实例说明用法:

```
redis 127.0.0.1:6379> set name HongWan
OK
redis 127.0.0.1:6379> set age 20
OK
redis 127.0.0.1:6379> keys *
1) "age"
```

```
2) "name"
redis 127.0.0.1:6379> shutdown
redis 127.0.0.1:6379>
```

我们先设置 2 个键值对，然后我们看一下系统中有没有产生 appendonly.aof 文件

```
[root@localhost redis-2.2.12]# ll
总计 184
-rw-rw-r-- 1 root root    9602 2011-07-22 00-RELEASENOTES
-rw-r--r-- 1 root root       0 08-09 19:37 appendonly.aof
-rw-rw-r-- 1 root root      55 2011-07-22 BUGS
-rw-rw-r-- 1 root root 84050 2011-07-22 Changelog
drwxrwxr-x 2 root root    4096 2011-07-22 client-libraries
-rw-rw-r-- 1 root root     671 2011-07-22 CONTRIBUTING
-rw-rw-r-- 1 root root    1487 2011-07-22 COPYING
drwxrwxr-x 4 root root    4096 2011-07-22 deps
drwxrwxr-x 2 root root    4096 2011-07-22 design-documents
drwxrwxr-x 2 root root 12288 2011-07-22 doc
-rw-rw-r-- 1 root root     652 2011-07-22 INSTALL
-rw-rw-r-- 1 root root     337 2011-07-22 Makefile
-rw-rw-r-- 1 root root    1954 2011-07-22 README
-rw-rw-r-- 1 root root 19071 08-09 19:24 redis.conf
drwxrwxr-x 2 root root    4096 08-05 19:12 src
drwxrwxr-x 7 root root    4096 2011-07-22 tests
-rw-rw-r-- 1 root root     158 2011-07-22 TODO
drwxrwxr-x 2 root root    4096 2011-07-22 utils
[root@localhost redis-2.2.12]#
```

结果证明产生了，接着我们将 redis 再次启动后来看一下数据是否还在

```
[root@localhost redis-2.2.12]# src/redis-cli
redis 127.0.0.1:6379> keys *
1) "age"
2) "name"
redis 127.0.0.1:6379>
```

数据还存在系统中，说明系统是在启动时执行了一下从磁盘到内存的 load 数据的过程。

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。接来我们看一下实际的例子:

我们先调用 5 次 incr age 命令:

```
redis 127.0.0.1:6379> incr age
(integer) 21
redis 127.0.0.1:6379> incr age
(integer) 22
redis 127.0.0.1:6379> incr age
(integer) 23
redis 127.0.0.1:6379> incr age
(integer) 24
redis 127.0.0.1:6379> incr age
(integer) 25
redis 127.0.0.1:6379>
```

接下来我们看一下日志文件的大小

```
[root@localhost redis-2.2.12]# ll
总计 188
-rw-rw-r-- 1 root root    9602 2011-07-22 00-RELEASENOTES
-rw-r--r-- 1 root root     259 08-09 19:43 appendonly.aof
-rw-rw-r-- 1 root root      55 2011-07-22 BUGS
-rw-rw-r-- 1 root root 84050 2011-07-22 Changelog
```

大小为 259 个字节，接下来我们调用一下 bgrewriteaof 命令将内存中的数据重新刷到磁盘的日志文件中

```
redis 127.0.0.1:6379> bgrewriteaof
Background append only file rewriting started
redis 127.0.0.1:6379>
```

再看一下磁盘上的日志文件大小

```
[root@localhost redis-2.2.12]# ll
总计 188
-rw-rw-r-- 1 root root    9602 2011-07-22 00-RELEASENOTES
-rw-r--r-- 1 root root     127 08-09 19:45 appendonly.aof
-rw-rw-r-- 1 root root      55 2011-07-22 BUGS
```

```
-rw-rw-r-- 1 root root 84050 2011-07-22 Changelog
```

日志文件大小变为 127 个字节了，说明原来日志中的重复记录已被刷新掉了。

4.5 发布及订阅消息

发布订阅(pub/sub)是一种消息通信模式，主要的目的是解耦消息发布者和消息订阅者之间的耦合，这点和设计模式中的观察者模式比较相似。pub/sub 不仅仅解决发布者和订阅者直接代码级别耦合也解决两者在物理部署上的耦合。redis 作为一个 pub/sub 的 server，在订阅者和发布者之间起到了消息路由的功能。订阅者可以通过 subscribe 和 psubscribe 命令向 redis server 订阅自己感兴趣的消息类型，redis 将消息类型称为通道(channel)。当发布者通过 publish 命令向 redis server 发送特定类型的消息时。订阅该消息类型的全部 client 都会收到此消息。这里消息的传递是多对多的。一个 client 可以订阅多个 channel,也可以向多个 channel 发送消息。

下面做个实验。这里使用 3 不同的 client，client1 用于订阅 tv1 这个 channel 的消息，client2 用于订阅 tv1 和 tv2 这 2 个 chanel 的消息，client3 用于发布 tv1 和 tv2 的消息。

Client 1	Client 2	Client 3
redis 127.0.0.1:6379> **subscribe tv1** Reading messages... (press Ctrl-C to quit) 1) "subscribe" 2) "tv1" 3) (integer) 1	redis 127.0.0.1:6379> **subscribe tv1 tv2** Reading messages... (press Ctrl-C to quit) 1) "subscribe" 2) "tv1" 3) (integer) 1 1) "subscribe" 2) "tv2" 3) (integer) 2	
		redis 127.0.0.1:6379> **publish tv1 program1** (integer) 2 redis 127.0.0.1:6379>
redis 127.0.0.1:6379> subscribe tv1 Reading messages... (press Ctrl-C to quit) 1) "subscribe" 2) "tv1" 3) (integer) 1 **1) "message"** **2) "tv1"** **3) "program1"**	redis 127.0.0.1:6379> subscribe tv1 tv2 Reading messages... (press Ctrl-C to quit) 1) "subscribe" 2) "tv1" 3) (integer) 1 1) "subscribe" 2) "tv2" 3) (integer) 2 **1) "message"** **2) "tv1"** **3) "program1"**	
		redis 127.0.0.1:6379> **publish tv2 program2** (integer) 1 redis 127.0.0.1:6379>
redis 127.0.0.1:6379> subscribe tv1	redis 127.0.0.1:6379> subscribe tv1 tv2	

Reading messages... (press Ctrl-C to quit) 1) "subscribe" 2) "tv1" 3) (integer) 1 1) "message" 2) "tv1" 3) "program1"	Reading messages... (press Ctrl-C to quit) 1) "subscribe" 2) "tv1" 3) (integer) 1 1) "subscribe" 2) "tv2" 3) (integer) 2 1) "message" 2) "tv1" 3) "program1" **1) "message"** **2) "tv2"** **3) "program2"**	

下面将详细的解释一下上面的例子

1、client1 订阅了 tv1 这个 channel 这个频道的消息，client2 订阅了 tv1 和 tv2 这 2 个频道的消息

2、client3 是用于发布 tv1 和 tv2 这 2 个频道的消息发布者

3、接下来我们在 client3 发布了一条消息”publish tv1 program1”，大家可以看到这条消息是发往 tv1 这个频道的

4、理所当然的 client1 和 client2 都接收到了这个频道的消息

5、 然后 client3 又发布了一条消息”publish tv2 program2”，这条消息是发往 tv2 的，由于 client1 并没有订阅 tv1，所以 client1 的结果中并没有显示出任何结果,但 client2 订阅了这个频道，所以 client2 是会有返回结果的。

我们也可以用 psubscribe tv*的方式批量订阅以 tv 开头的频道的内容。

看完这个小例子后应该对 pub/sub 功能有了一个感性的认识。需要注意的是当一个连接通过 subscribe 或者 psubscribe 订阅通道后就进入订阅模式。在这种模式除了再订阅额外的通道或者用 unsubscribe 或者 punsubscribe 命令退出订阅模式，就不能再发送其他命令。另外使用 psubscribe 命令订阅多个通配符通道，如果一个消息匹配上了多个通道模式的话，会多次收到同一个消息。

4.6 Pipeline 批量发送请求

redis 是一个 cs 模式的 tcp server，使用和 http 类似的请求响应协议。一个 client 可以通过一个 socket 连接发起多个请求命令。每个请求命令发出后 client 通常会阻塞并等待 redis 服务处理，redis 处理完后请求命令后会将结果通过响应报文返回给 client。基本的通信过程如下:

```
Client: INCR X
Server: 1
Client: INCR X
Server: 2
Client: INCR X
Server: 3
```

```
Client: INCR X
Server: 4
```

基本上四个命令需要 8 个 tcp 报文才能完成。由于通信会有网络延迟,假如从 client 和 server 之间的包传输时间需要 0.125 秒。那么上面的四个命令 8 个报文至少会需要 1 秒才能完成。这样即使 redis 每秒能处理 100 个命令，而我们的 client 也只能一秒钟发出四个命令。这显示没有充分利用 redis 的处理能力，怎么样解决这个问题呢? 我们可以利用 pipeline 的方式从 client 打包多条命令一起发出，不需要等待单条命令的响应返回，而 redis 服务端会处理完多条命令后会将多条命令的处理结果打包到一起返回给客户端。通信过程如下

```
Client: INCR X
Client: INCR X
Client: INCR X
Client: INCR X
Server: 1
Server: 2
Server: 3
Server: 4
```

假设不会因为 tcp 报文过长而被拆分。可能两个 tcp 报文就能完成四条命令,client 可以将四个 incr 命令放到一个 tcp 报文一起发送，server 则可以将四条命令的处理结果放到一个 tcp 报文返回。通过 pipeline 方式当有大批量的操作时候，我们可以节省很多原来浪费在网络延迟的时间，需要注意到是用 pipeline 方式打包命令发送，redis 必须在处理完所有命令前先缓存起所有命令的处理结果。打包的命令越多，缓存消耗内存也越多。所以并不是打包的命令越多越好。具体多少合适需要根据具体情况测试。下面是个 Java 使用 pipeline 的实验:

```
import org.jredis.JRedis;
import org.jredis.connector.ConnectionSpec;
import org.jredis.ri.alphazero.JRedisClient;
import org.jredis.ri.alphazero.JRedisPipelineService;
import org.jredis.ri.alphazero.connection.DefaultConnectionSpec;

public class TestPipeline {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        //采用 pipeline 方式发送指令
        usePipeline();
        long end = System.currentTimeMillis();
        System.out.println("用 pipeline 方式耗时： " + (end - start) + "毫秒");

        start = System.currentTimeMillis();
        //普通方式发送指令
        withoutPipeline();
        end = System.currentTimeMillis();
        System.out.println("普通方式耗时： " + (end - start) + "毫秒");
```

```
	}

	//采用 pipeline 方式发送指令
	private static void usePipeline() {
		try {
			ConnectionSpec spec = DefaultConnectionSpec.newSpec(
					"192.168.115.170", 6379, 0, null);
			JRedis jredis = new JRedisPipelineService(spec);

			for (int i = 0; i < 100000; i++) {
				jredis.incr("test2");
			}

			jredis.quit();
		} catch (Exception e) {
		}
	}

	//普通方式发送指令
	private static void withoutPipeline() {
		try {
			JRedis jredis = new JRedisClient("192.168.115.170", 6379);

			for (int i = 0; i < 100000; i++) {
				jredis.incr("test2");
			}

			jredis.quit();
		} catch (Exception e) {
		}
	}
}
```

执行结果如下:

```
-- JREDIS -- INFO: Pipeline thread <response-handler> started.
-- JREDIS -- INFO: Pipeline <org.jredis.ri.alphazero.connection.SynchPipelineConnection@1bf73fa> connected
用 pipeline 方式耗时：11531 毫秒
-- JREDIS -- INFO: Pipeline <org.jredis.ri.alphazero.connection.SynchPipelineConnection@1bf73fa> disconnected
-- JREDIS -- INFO: Pipeline thread <response-handler> stopped.
普通方式耗时：15985 毫秒
```

所以用两种方式发送指令，耗时是不一样的，具体是否使用 pipeline 必须要基于大家手中的网络情况来决定，不能一切都按最新最好的技术来实施，因为它有可能不是最适合你的。

4.7 虚拟内存的使用

首先说明下 redis 的虚拟内存与操作系统的虚拟内存不是一码事，但是思路和目的都是相同的。就是暂时把不经常访问的数据从内存交换到磁盘中，从而腾出宝贵的内存空间用于其他需要访问的数据。尤其是对于 redis 这样的内存数据库，内存总是不够用的。除了可以将数据分割到多个 redis server 外。另外的能够提高数据库容量的办法就是使用虚拟内存把那些不经常访问的数据交换的磁盘上。如果我们的存储的数据总是有少部分数据被经常访问，大部分数据很少被访问，对于网站来说确实总是只有少量用户经常活跃。当少量数据被经常访问时，使用虚拟内存不但能提高单台 redis server 数据库的容量，而且也不会对性能造成太多影响。

redis 没有使用操作系统提供的虚拟内存机制而是自己在实现了自己的虚拟内存机制，主要的理由有两点:
1、操作系统的虚拟内存是已 4k 页面为最小单位进行交换的。而 redis 的大多数对象都远小于 4k，所以一个操作系统页面上可能有多个 redis 对象。另外 redis 的集合对象类型如 list,set 可能存在与多个操作系统页面上。最终可能造成只有 10%key 被经常访问，但是所有操作系统页面都会被操作系统认为是活跃的，这样只有内存真正耗尽时操作系统才会交换页面。

2、相比于操作系统的交换方式，redis 可以将被交换到磁盘的对象进行压缩,保存到磁盘的对象可以去除指针和对象元数据信息,一般压缩后的对象会比内存中的对象小 10 倍,这样 redis 的虚拟内存会比操作系统虚拟内存能少做很多 io 操作。

下面是 vm 相关配置

```
vm-enabled yes                          #开启 vm 功能
vm-swap-file /tmp/redis.swap            #交换出来的 value 保存的文件路径
vm-max-memory 1000000                   #redis 使用的最大内存上限
vm-page-size 32                         #每个页面的大小 32 个字节
vm-pages 134217728                      #最多使用多少页面
vm-max-threads 4                        #用于执行 value 对象换入换出的工作线程数量
```

redis 的虚拟内存在设计上为了保证 key 的查找速度，只会将 value 交换到 swap 文件中。所以如果是内存问题是由于太多 value 很小的 key 造成的，那么虚拟内存并不能解决，和操作系统一样 redis 也是按页面来交换对象的。redis 规定同一个页面只能保存一个对象。但是一个对象可以保存在多个页面中。在 redis 使用的内存没超过 vm-max-memory 之前是不会交换任何 value 的。当超过最大内存限制后，redis 会选择较过期的对象。如果两个对象一样过期会优先交换比较大的对象，精确的公式 swappability = age*log(size_in_memory)。对于 vm-page-size 的设置应该根据自己的应用将页面的大小设置为可以容纳大多数对象的大小，太大了会浪费磁盘空间,太小了会造成交换文件出现碎片。对于交换文件中的每个页面,redis 会在内存中对应一个 1bit 值来记录页面的空闲状态。所以像上面配置中页面数量(vm-pages 134217728)会占用 16M 内存用来记录页面空闲状态。vm-max-threads 表示用做交换任务的线程数量。如果大于 0 推荐设为服务器的 cpu 内核的数量，如果是 0 则交换过程在主线程进

行。