## 一、Redis概述

Redis 是一款开源的，基于 BSD 许可的，高级键值 (key-value) 缓存 (cache) 和存储 (store) 系统。由于 Redis 的键包括 string，hash，list，set，sorted set，bitmap 和 hyperloglog，所以常常被称为数据结构服务器。你可以在这些类型上面运行原子操作，例如，追加字符串，增加哈希中的值，加入一个元素到列表，计算集合的交集、并集和差集，或者是从有序集合中获取最高排名的元素。

为了满足高性能，Redis 采用内存 (in-memory) 数据集 (dataset)。根据你的使用场景，你可以通过每隔一段时间转储数据集到磁盘，或者追加每条命令到日志来持久化。持久化也可以被禁用，如果你只是需要一个功能丰富，网络化的内存缓存。

Redis 还支持主从异步复制，非常快的非阻塞初次同步、网络断开时自动重连局部重同步。 其他特性包括：

* 事务
* 订阅/发布
* Lua 脚本
* 带 TTL 的键
* LRU 回收健
* 自动故障转移 (failover)

你可以通过多种语言来使用 Redis。

Redis 是由 ANSI C 语言编写的，在无需额外依赖下，运行于大多数 POSIX 系统，如 Linux、\*BSD、OS X。Redis 是在 Linux 和 OS X 两款操作系统下开发和充分测试的，我们推荐 Linux 为部署环境。Redis 也可以运行在 Solaris 派生系统上，如 SmartOS，但是支持有待加强。没有官方支持的 Windows 构建版本，但是微软开发和维护了一个 64 位 Windows 的版本。

## 二、数据类型初探

2.1 字符串 (Strings)

字符串是 Redis 最基本的数据类型。Redis 字符串是二进制安全的，也就是说，一个 Redis 字符串可以包含任意类型的数据，例如一张 JPEG 图像，或者一个序列化的 Ruby 对象。

一个字符串最大为 512M 字节。

你可以使用 Redis 的字符串类型做很多有意思的事情，例如，你可以：

使用 INCR 命令族 (INCR，DECR，INCRBY)，将字符串作为原子计数器。

使用 APPEND 命令追加字符串。

使用 GETRANGE 和 SETRANGE 命令，使字符串作为随机访问向量 (vectors)。

编码大量数据到很小的空间，或者使用 GETBIT 和 SETBIT 命令，创建一个基于 Redis 的布隆 (Bloom) 过滤器。

2.2

## 三、Redis持久化

持久化是将内存数据存储到硬盘的过程。

Redis提供两种持久化策略：快照(snapshotting)、只追加文件(append-only file,AOF)。

### 3.1 快照

快照可以将存在某一时刻的所有数据都写入到硬盘里面。

默认情况下，Redis将快照数据保存到磁盘的dump.rdb文件中，可以在配置文件中配置：save N M(N秒内至少有M次数据集改动时保存数据)，也可以手动调用save或bgSave命令触发。

save命令是在主线程中保存快照，这种方式会阻塞所有客户端请求。

bgsave会fork一个子进程来处理数据快照，不影响主线程对客户端请求的处理。

#### 3.1.1 快照的过程

1) 触发快照时，Redis会fork一个子进程

2) 父进程进行处理客户端请求，子进程负责将内存内容写入到临时文件。由于操作系统的写时复制机制(copy on write)，父子进程会共享相同的物理页面。

父进程处理请求时，操作系统会为父进程要修改的页面创建副本，而不是写共享的页面；

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) 子进程将快照写入到临时文件完毕后，会用临时文件替换原来的快照文件，然后子进程退出。

Remark：

1) 每次快照时，都是将内存数据完整写入到磁盘。

2) 写入时复制(Copy-on-write，简称COW)是一种计算机程序设计领域的优化策略。核心思想是:如果有多个调用者（callers）同时要求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的（transparently）。此作法主要的优点是如果调用者没有修改该资源，就不会有副本（private copy）被创建，因此多个调用者只是读取操作时可以共享同一份资源。

3) fork的是进程，不是线程。

#### 3.1.2 快照的优缺点

优点：

1) 快照后的文件dump.rdb可以用于数据备份。

2) 通过父子进程的方式，父进程不需要执行磁盘IO。

3) RDB在重启保存了大数据集的实例时，比AOF要快。

缺点：

1) 如果Redis停止工作，会丢失最近一个快照到停止时间点新写入的数据。

2) 当需要快照的数据集非常大并且CPU性能不够强大时，Redis会停止服务客户端几毫米甚至一秒。

### 3.2 AOF

AOF会在执行写命令时，将被执行的写命令复制到磁盘。

AOF记录的是数据发生的变化，要恢复数据，只需要重新执行一次AOF文件包含的所有写命令即可。

AOF同步的频率有：always(每个写命令同步一次)、everysec(一秒同步一次)、no(让操作系统决定)。

#### 3.2.1 文件同步

在向硬盘写入文件时，写入的内容首先会被存储到缓冲区，操作系统在某个时候将缓冲区内容写入到硬盘。写入到硬盘才算真正的持久化完成。

因此AOF方式的持久化也可能会丢失部分数据。

#### 3.2.2重新/压缩AOF文件

AOF的方式会导致持久化文件越来越大，在对同个键进行多次写操作的时候，只需要保存之后一次写操作，为了压缩AOF文件，Redis提供bgRewriteAOF命令进行压缩AOF文件，也可以通过配置文件的相关配置项，配置压缩AOF的触发条件。

压缩的过程如下：

1) 触发压缩时，Redis会fork一个子进程

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) 当子进程完成重写文件，父进程收到一个信号，追加内存缓冲区到子进程创建的文件末尾。

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

Remark：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。

#### 3.2.3 AOF的优缺点

优点：

1) AOF根据配置的策略，丢失的数据会很少

2) 即使AOF文件损坏，可以通过redis-check-aof工具进行修复

3) AOF文件有重写机制

4) 即使不小心的FlushAll了，也可以在重写之前，停止Redis服务，然后将文件内的该命令删除，得以恢复数据

缺点：

1) 同等数量集，AOF文件通常要大于等级的RDB文件

### 3.3 RDB和AOF相关的配置

#### 3.3.1 RDB相关配置

# 可以用如下的指令来让数据保存到磁盘上，即控制RDB快照功能

# 禁用RDB持久化的策略，只要不设置任何save指令就可以，或者给save传入一个空字符串参数也可以达到相同效果

save <seconds> <changes>

save ""

stop-writes-on-bgsave-error yes

# 对于存储到磁盘中的快照，可以设置是否进行压缩存储

# 如果是的话，redis会采用LZF算法进行压缩

# 如果不想消耗CPU来进行压缩的话，可以设置为关闭此功能，但是存储在磁盘上的快照会比较大

rdbcompression yes

# 在存储快照后，可以让redis使用CRC64算法来进行数据校验，但是这样做会增加大约10%的性能消耗

# 如果希望获取到最大的性能提升，可以关闭此功能

rdbchecksum yes

# 设置快照文件的名称，默认配置为

dbfilename dump.rdb

# 设置这个快照文件存放的路径

# 比如默认设置就是当前文件夹

dir ./

#### 3.3.2 AOF相关配置

# 默认情况下，redis会异步的将数据持久化到磁盘

# 这种模式在大部分应用程序中已被验证是很有效的，但是在一些问题发生时，比如断电，则这种机制可能会导致数分钟的写请求丢失。

# 建议大家，AOF机制和RDB机制可以同时使用，不会有任何冲突。

appendonly no

# 可以设置aof文件的名称：

appendfilename "appendonly.aof"

# fsync()调用，用来告诉操作系统立即将缓存的指令写入磁盘

# 一些操作系统会“立即”进行，而另外一些操作系统则会“尽快”进行。

# redis支持三种不同的模式：

# 1.no：不调用fsync()。而是让操作系统自行决定sync的时间。这种模式下，redis的性能会最快。

# 2.always：在每次写请求后都调用fsync()。这种模式下，redis会相对较慢，但数据最安全。

# 3.everysec：每秒钟调用一次fsync()。这是性能和安全的折衷。

# 默认情况下为everysec

appendfsync everysec

# 当fsync方式设置为always或everysec时，如果后台持久化进程需要执行一个很大的磁盘IO操作，那么redis可能会在fsync()调用时卡住

#目前尚未修复这个问题，这是因为即使我们在另一个新的线程中去执行fsync()，也会阻塞住同步写调用

# 为了缓解这个问题，我们可以使用下面的配置项，这样的话，当BGSAVE或BGWRITEAOF运行时，fsync()在主进程中的调用会被阻止

# 这意味着当另一路进程正在对AOF文件进行重构时，redis的持久化功能就失效了，就好像我们设置了“appendsync none”一样

# 如果你的redis有时延问题，那么请将下面的选项设置为yes。否则请保持no，因为这是保证数据完整性的最安全的选择。

no-appendfsync-on-rewrite no

# 允许redis自动重写aof

# 当aof增长到一定规模时，redis会隐式调用BGREWRITEAOF来重写log文件，以缩减文件体积。

# redis是这样工作的：redis会记录上次重写时的aof大小。假如redis自启动至今还没有进行过重写，那么启动时aof文件的大小会被作为基准值。这个基准值会和当前的aof大小进行比较。如果当前aof大小超出所设置的增长比例，则会触发重写。另外，你还需要设置一个最小大小，是为了防止在aof很小时就触发重写

# 如果设置auto-aof-rewrite-percentage为0，则会关闭此重写功能

auto-aof-rewrite-percentage 100

auto-aof-rewrite-min-size 64mb

## 四.Redis部署

### 4.1 哨兵部署



### 4.2 集群部署



### 4.3 主从部署



### 4.4 主从节点数据同步



## 五、Redis应用

### 5.1 限制访问频率

原理：这一应用场景主要使用到了Redis数据有过期时间这个属性。可以有两种实现：需要限定的可以是某个IP或某个用户的唯一标识，如userId。

基于要限定的对象，设定key为[限定对象]:[YYYYMMDDHHmm],该键对应的值为访问次数，这种限制只能是单个单位的时间，如1秒、1分钟、1小时、1天等。并且当相邻两个时间段时，可以发生超出限定频率的访问。

另外一种是基于列表进行，将需要限定的对象的唯一标识作为key，如userId、Ip,键对应的值为该对象每次的访问时间列表，如列表的元素个数超出了限定频率，则判断最早的时间和当前的时间差是不是超过指定的时间段，若不是，则将最早的时间去掉，把当前时间加进去，否则，访问频率超过了限定。

基于Zset的访问限制：

使用zset的score来圈出时间窗口,每次操作都把时间窗口的数据砍掉，然后再比较Zset的数量是否超过限制频率。

### 5.2 分布式锁

资料来源: <http://www.importnew.com/27477.html>

分布式锁一般有三种实现方式：

1). 数据库乐观锁；

2). 基于Redis的分布式锁；

3). 基于ZooKeeper的分布式锁；

#### 可靠性

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1).**互斥性**。在任意时刻，只有一个客户端能持有锁。

2).**不会发生死锁**。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。

3).**具有容错性**。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。

4).**解铃还须系铃人**。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

#### 组件依赖

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

#### 加锁代码

public class RedisTool {

private static final String LOCK\_SUCCESS = "OK";

private static final String SET\_IF\_NOT\_EXIST = "NX";

private static final String SET\_WITH\_EXPIRE\_TIME = "PX";

/\*\*

\* 尝试获取分布式锁

\* @param jedis Redis客户端

\* @param lockKey 锁

\* @param requestId 请求标识

\* @param expireTime 超期时间

\* @return 是否获取成功

\*/

public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {

String result = jedis.set(lockKey, requestId, SET\_IF\_NOT\_EXIST, SET\_WITH\_EXPIRE\_TIME, expireTime);

if (LOCK\_SUCCESS.equals(result)) {

return true;

}

return false;

}

}

jedis.set(String key, String value, String nxxx, String expx, int time)，这个set()方法一共有五个形参：

第一个为key，我们使用key来当锁，因为key是唯一的。

第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。

第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；

第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。

第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

#### 解锁代码

public class RedisTool {

private static final Long RELEASE\_SUCCESS = 1L;

/\*\*

\* 释放分布式锁

\* @param jedis Redis客户端

\* @param lockKey 锁

\* @param requestId 请求标识

\* @return 是否释放成功

\*/

public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {

String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));

if (RELEASE\_SUCCESS.equals(result)) {

return true;

}

return false;

}

}

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。关于非原子性会带来什么问题，可以阅读【解锁代码-错误示例2】 。那么为什么执行eval()方法可以确保原子性，源于Redis的特性，下面是官网对eval命令的部分解释：

简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命令。

### 5.3 附近的人

### 5.4 队列

## 六、底层原理

### 6.1 过期策略

### 6.2 线程IO模型

### 6.3 String设计

### 6.4 Hash设计

### 6.5 List设计

### 6.6 Set设计

### 6.7 Zset设计