Redis尝试

如果是用apt-get或者yum install安装的redis，可以直接通过下面的命令停止/启动/重启redis

1、/etc/init.d/redis-server stop

2、/etc/init.d/redis-server start

3、/etc/init.d/redis-server restart

sudo ps -ef | grep redis

Redis事务错误处理的两种类型

事务先组队（multi）然后执行（exec）也可以取消执行（discard）

1. 组队阶段出错，如果使用exec去执行，那么所有的都不会执行
2. 如果执行阶段出错，那么只有出错的那条命令不会执行，别的都会执行

Watch关键字

在执行multi之前，先执行watch key1[key2]，可以监视一个（或多个）key，如果在事务执行之前这个（或这些）key被其他命令所改动，那么事务将被打断；

Unwatch（需要用在multi之前才会生效）

取消watch命令对所有key的监视

如果在执行watch命令之后，exec命令或discard命令先被执行了的话那么就不需要在执行unwatch了

Redis事务三特性

1. 单独的隔离操作，事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令所打断。
2. 没有隔离级别的概念：队列中的命令没有提交之前都不会被实际执行，因为事务提交前任何指令都不会被实际执行。
3. 不保证原子性：事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

在秒杀问题中，Redis连接超时

//通过连接池来获取Jedis连接，用来解决连接超时问题

在秒杀问题中，超买和超卖问题（使用乐观锁来进行解决）

监视库存，使用Redis里的watch监视库存的key，然后，将库存减一和将用户ID加入数据库的操作，加入事务，调用jedis里面的multi方法，生产multi对象，然后组队，然后执行返回一个list，如果list为空或者size为0就表示事务存在冲突，返回秒杀失败；

秒杀问题中的库存遗留问题（乐观锁造成库存遗留问题）

使用悲观锁来解决，lua脚本

Redis持久化操作的两种方式RDB（Redis DataBase）和AOF（Append Of File）

1. RDB

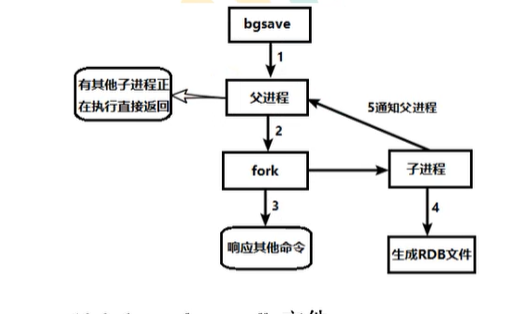
在指定的时间间隔内，将内存中的数据集快照写入到磁盘中，也就是行话讲的Snapshot快照，它恢复时是讲快照文件直接读到内存里

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件，整个过程中，主进程是不会进行任何io操作的，这就确保了极高的性能，如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失

Fork的作用是复制一个与当前进程一样的进程，新进程的所有数据数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

在linux程序中，fork()会产生一个和父进程完全相同的子进程，但子进程在此后多会exec系统调用，出于效率考虑，Linux中引入了“写时复制技术”

一般情况父进程和子进程会共用同一段物理内存，只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程



Dump.rdb 在redis.conf中配置文件名称，默认为dump.rdb

Rdb方式持久化的优势

适合大规模的数据恢复

对数据完整性和一致性要求不高更适合使用

节省磁盘空间

恢复速度快

RDB劣势

Fork的时候，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑

虽然redis在fork时使用了写时拷贝技术，但是如果数据庞大时还是比较消耗性能

在备份周期在一定间隔时间做一次备份，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更改

Redis持久化之Aof

以日志的形式记录每个操作（增量保存，读操作不记录），只许追加文件不可以改写文件

注意如果rdb和aof同时开启，系统默认取aof的数据（数据不会存在丢失）

在配置文件中，appendonly yes打开此功能，是通过将执行过的写指令记录下俩，数据恢复时，把所以AOF记录的指令从头到尾再执行一遍，如果有写操作就将该命令追加到Aof问价的末尾，默认AOF持久化策略（可以通过配置appendfsync来更改同步频率，always、everysec、no），时每秒钟fsync一次（就是把缓存中的写指令记录到磁盘中），也可以保证redis性能，即使故障，也只会丢失一秒钟的数据。

如果在追加日志时，遇到磁盘空间满、inode满或断电等情况导致日志写入不完整，redis提供了redis-check-aof（redis-check-aof –fix appendonly.aof）工具，可以用来进行日志修复。

因为AOF采用了追加机制，如果不做任何处理的话，AOF文件会越来越大，对于该种情况，redis提供了AOF文件重写（rewrite），即当AOF文件的大小超过了所设定的阈值时，redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集

重写机制的原理：

如果一个命令执行了100次，如果不适用重写，那我们就要存储100次，重写可以把这100条指令合并成一条set指令，这就是重写机制的原理

在进行AOF重写时，仍然是采用先写临时文件，全部完成后在替换的流程，全部完成后再替换

AOF相较与RDB的劣势：

AOF文件比RDB文件大，占用空间， AOF方式的恢复速度要慢于RDB方式

Rewrite压缩

中间怎么做不关注，只关注最后的结果，用较少的指令，达到同样的效果

Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍（配置文件中对应于auto-aof-rewrite-percentage 2）且文件大于64M（配置文件中auto-aof-rewrite-mini-size 64）时触发

Aof的持久化流程

客户端的请求写命令会被append追加到aof缓冲区

Aof缓冲区会根据aof持久化策略[always，everysec，no]将操作sync同步到磁盘的aof文件中

Aof文件大小超过重写策略或手动重写时，会对aof文件rewrite重写，压缩aof文件容量

Redis服务重启时，会重新load加载aof文件中的写操作达到数据恢复的目的

Aof的优势 备份机制更加稳健，丢失数据概率更低

可读的日志文件，通过操作aof文件，可以处理误操作

劣势

比起rdb占用更多的磁盘空间

恢复备份速度要慢

每次读写都同步的话，有一定的性能压力

存在个别 bug造成一些问题

总结：

用哪个好

官方推荐两个都启用

如果对数据不敏感，可以选单独用rdb

不建议单独用aof，因为可能会出现bug

如果只是做纯内存缓存，可以都不用

Redis主从复制

主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写为主，Slave以读为主

好处

1. 读写分离
2. 容灾快速恢复

Redis主从复制的配置

1. 创建/myredis文件夹
2. 复制redis.conf到该文件夹中
3. 配置一主两从，创建三个配置文件redis6379.conf、redis6380.conf、redis6381.conf
4. 在三个配置文件中写入内容

include /myredis/redis.conf

pidfile /var/run/redis\_6379.pid

port 6379

dbfilename dump6379.rdb

5、通过三个配置文件启动三个redis服务

查看三台主机运行情况（info replication redis-cli -p 6379）

6、在从机上执行 slaveof 主机ip 端口号

当从服务器连接上主服务器之后，从服务器向主服务器发送进行数据同步消息

主服务器接到从服务器发送过来同步消息，把主服务器数据进行持久化，rdb文件，把rdb文件发送给从服务器，从服务器拿到rdb进行读取

每次主服务器进行写操作之后，和从服务器进行数据同步

薪火相传

主服务器下面的从服务器可以再有从服务器

反客为主

当有从服务器的从服务器，它的主服务器挂掉了，那么它就可以成为主服务器（slaveof

no one），反客为主

反客为主的自动化（哨兵模式）

哨兵模式配置过程

1. 自定义的/myredis目录下新建sentinel.conf文件，名字绝不能错
2. 配置文件中填写的内容为：sentinel monitor mymaster 127.0.0.1 6379 1

其中mymaster为给监控对象起的一个服务器名称，1为至少有多少个哨兵同意迁移的数量

如果主服务器挂掉，哨兵检测到，切换主服务器，同时设置原来挂掉的主服务器为新主服务器的从服务器

复制延时

所有的写都是在master上操作，然后同步更新到slave上，所以从master同步到slave机器有一定延迟，当系统繁忙的时候，延迟问题会更加严重，slave机器数量的增加也会使这个问题更加严重

新主选举：

选择优先级靠前的，选择偏移量最大的，选择runid最小的从服务

Vi中 :set nu!取消行号，:set nu显示行号

优先级在redis.conf中默认：replica-priority 100 值越小优先级越高

偏移量是指获得原主机数据最全的

每个redis实例启动后都会随机生成一个40位的runid

Redis集群

什么情况下会用到集群

容量不够，redis如何进行扩容？

并发写操作，redis如何分摊？

另外，主从模式，薪火相传模式，主机宕机，导致ip地址发生变化，应用程序中配置需要修改对应的主机地址、端口等信息

之前通过代理主机来解决，但是redis3.0中提供了解决方案，就是无中心化集群配置

代理主机：

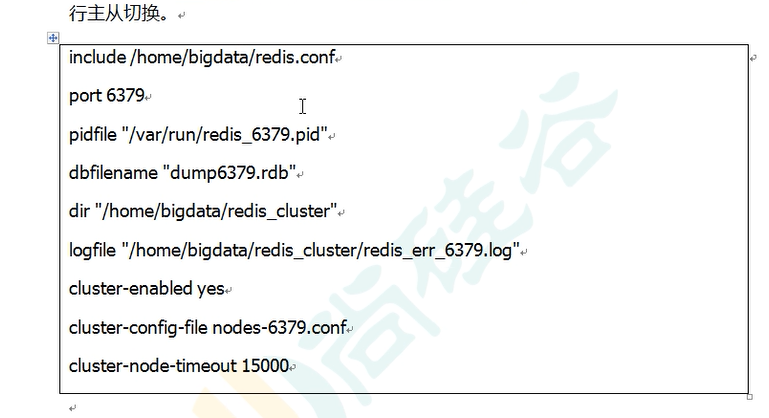
Redis集群搭建

在要创建集群的服务的conf文件中加入

cluster-enabled yes 打开集群模式

cluster-config-file nodes-6379.conf 设置节点配置文件名

cluster-node-timeout 15000 设置节点失联时间，超过该时间（毫秒），集群自动进行主从切换



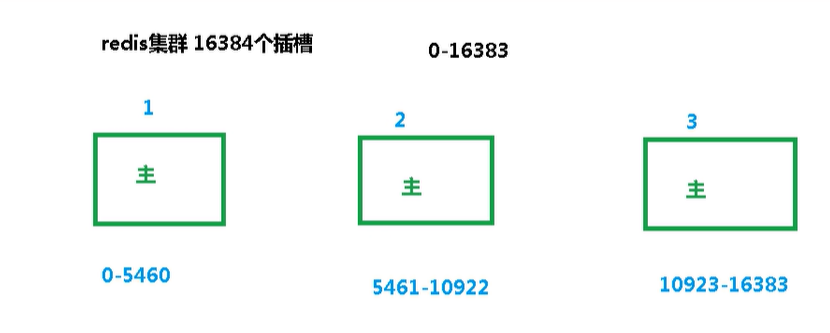
启动6个redis服务

将6个服务合并成一个集群

创建集群命令：redis-cli --cluster create --cluster-replicas 1 192.168.64.188:6379 192.168.64.188:6380 192.168.64.188:6381 192.168.64.188:6389 192.168.64.188:6390 192.168.64.188:6391

All 16384 slots covered.

一个集群包含16384个插槽，一个主机负责一个范围的插槽，如下图



集群Set值的过程 set k1 value1 拿k1去计算k1所在的插槽向插槽位置中加入值

计算方法为CRC(key)%16383

集群的连接方式：redis-cli -c -p 6379

通过cluster nodes查看集群信息

Redis cluster如何分配这六个节点

一个集群至少要有三个主节点

选项—cluster-replicas 1表示我们希望为集群中的每个主节点创建一个从节点。分配原则尽量保证每个主数据库运行在不同的ip地址，每个从库和主库不在一个ip地址上

Redis集群一次性加入多个值

使用组的概念：

mset name{user} lucy age{user} 20 address{user} china

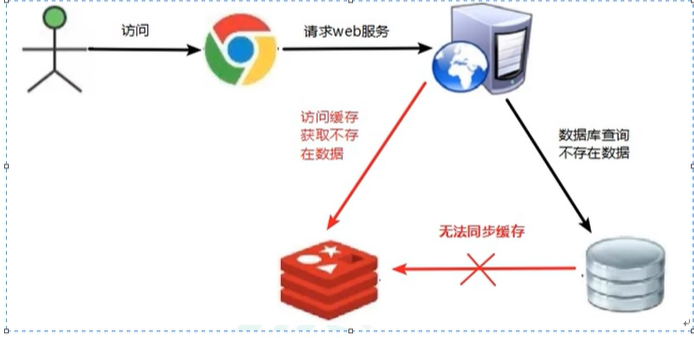
计算key的插槽值 cluster keyslot key

计算一个插槽有几个键 cluster countkeysinslot （插槽值）（注：每台集群服务器只能看自己所负责插槽范围的插槽值下有几个key）

集群故障恢复，主机会变为从机

如果某段插槽主机和从机全部挂掉，并且redis.conf配置文件cluster-require-full-coverage为yes，那么整个集群都挂掉，而如果为no，那么，该插槽数据全都不能使用，也无法存储。

Redis缓存穿透（缓存没有起到作用（被穿透了）而是一直去访问数据库）



现象：

1. 应用服务器压力增大
2. 请求增大时，Redis命中率较低，会频繁查询数据库，造成数据库压力增加（数据查询先查Redis没有数据时再去查数据库）

原因：

1. Redis查询不到数据
2. 出现很多非正常的url访问

解决方案：

一个一定不存在缓存及查询不到的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层面查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查，失去了缓存的意义

1. 对空值缓存：如果一个查询返回的数据为空（不管数据是否存在），我们仍然把这个空结果（null）进行缓存，设置空结果的过期时间很短最长不超过五分钟。
2. 设置可访问名单（白名单）：使用bitmaps类型定义一个可以访问的名单，名单id作为bitmaps的偏移量，每次访问和bitmaps里面的id进行比较，如果访问id不在bitmaps里面，进行拦截，不允许访问。
3. 采用布隆过滤器：（布隆过滤器（Bloom Filter）是1970年由布隆提出的。它实际上是一个很长的二进制向量（位图）和一系列随机映射函数（哈希函数））。
4. 进行实时监控：当发现redis的命中率开始急速降低，需要排查访问对象和访问的数据和运维人员配合，可以设置黑名单限制服务。

Redis缓存击穿

现象：

1. 数据的访问压力瞬时间增加了
2. Redis里面没有出现大量key过期
3. Redis正常运行

原因：

1. redis某个key过期了，大量访问使用这个key，这个key是个热门访问的key

解决：

1. 预先设置热门数据：在redis高峰访问之前，把一些热门数据提前存入到redis里面，加大这些热门数据key的时长。
2. 实时调整：线程监控哪些数据热门，实时调整key的过期时长
3. 使用锁：
4. 就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db
5. 先使用缓存工具的某些带成功操作返回值的操作（比如redis的setnx）去set一个mutext key
6. 当操作返回成功时候，再进行load db的操作，并回设缓存，最后删除mutex key；
7. 当操作返回失败，证明有线程在load db，当前线程睡眠一段时间再重试整个get缓存的方法

Redis缓存雪崩

雪崩效应对底层系统的冲击非常可怕

现象：

1. 数据库压力变大，服务器崩溃

原因：

1. 在极短的时间段，查询大量key的集中过期情况

解决：

1. 构建多级缓存架构：nginx缓存+redis缓存+其他缓存（ehcache等）
2. 使用锁或队列：用加锁或者队列的方式保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。不适用高并发情况（效率低）
3. 设置过期标志更新缓存：记录缓存数据是否过期（设置提前量），如果过期会触发通知另外的线程在后台去更新实际key的缓存。
4. 将缓存失效时间分散开：可在原有时间的基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件

分布式锁：

问题描述：原单体单机部署的系统被演化成分布式集群系统后，由于分布式系统多线程、多进程并且分布在不同机器上，这将使原单机部署情况下的并发控制锁策略失效，单纯的java api并不能提供分布式锁的能力。为了解决这个问题就需要一中跨jvm的互斥机制来控制共享资源的访问，这就是分布式锁要解决的问题

主流实现方案：

1. 基于数据库实现分布式锁
2. 基于缓存（Redis等）
3. 基于Zookeeper

每一种分布式锁解决方案的优缺点：

1. 性能：redis最高
2. 可靠性：zookeeper最高

基于redis实现分布式锁

Redis命令

#set sku:1:info “OK” NX PX 10000

EX second : 设置键的过期时间为second秒，set key value EX second效果等同于

SETEX key second value

第一张方案 setnx key value 第一次设值可以设置返回为1，第二次及之后的设值都不可以返回为0 。setnx key value为加锁，再将key删掉为释放锁(del key)

会产生的问题：如果某个程序长时间不释放锁，那么会阻塞

解决方案：给锁设置过期时间（expire users 10设置过期时间为10s）

该种方案会产生的另一个问题：由于redis的操作不是原子操作，会发生redis值设置成功，但是设置过期时间的失败

解决方案：上锁的时候同时设置过期时间（实现原子操作）set users 10 nx ex 40设置超时时间为40秒

为了防止自己释放其他服务的锁使用uuid进行优化，set key uuid nx ex 10，在释放锁的时候判断其值与自己的uuid是否一样，一样则释放，不一样则不释放