课前准备

- 准备redis安装包

课堂主题

Redis主从复制和哨兵机制、Redis集群、Redis和lua整合、Redis消息模式、Redis实现分布式锁、常见缓存问题

课堂目标

- 理解lua概念，能够使用Redis和lua整合使用
- 理解redis消息原理
- 掌握redis分布式锁的原理、本质
- 掌握Redisson的原理和实现
- 理解缓存穿透、缓存雪崩、缓存击穿、缓存双写一致性并掌握解决方案

知识要点

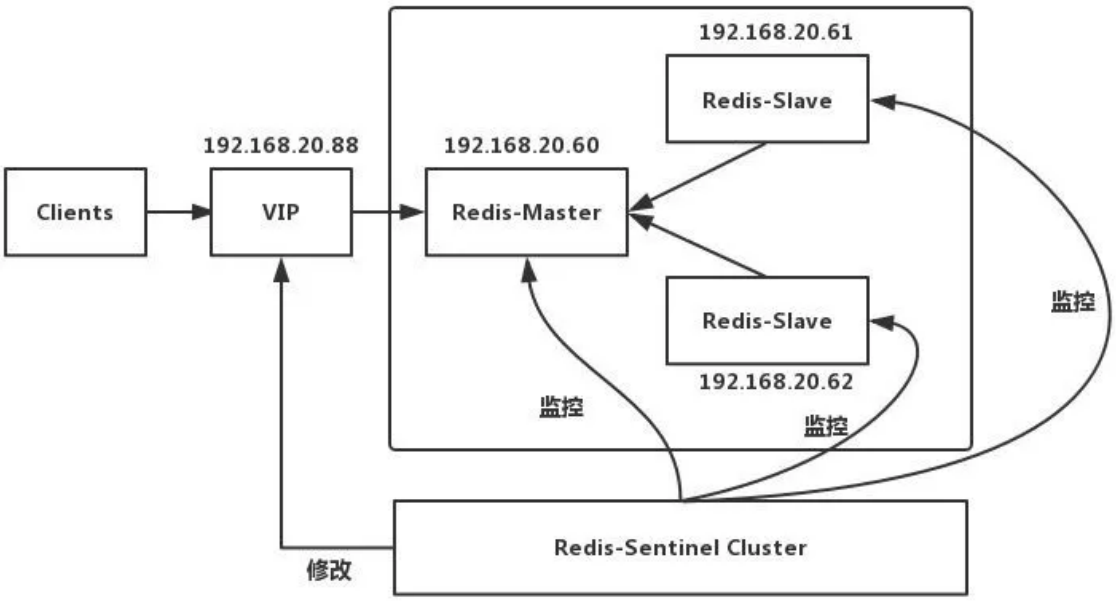
Redis 集群演变

4.1 主从复制

参考主从复制

4.2 Replication+Sentinel * 高可用

这套架构使用的是社区版本推出的原生高可用解决方案，其架构图如下！

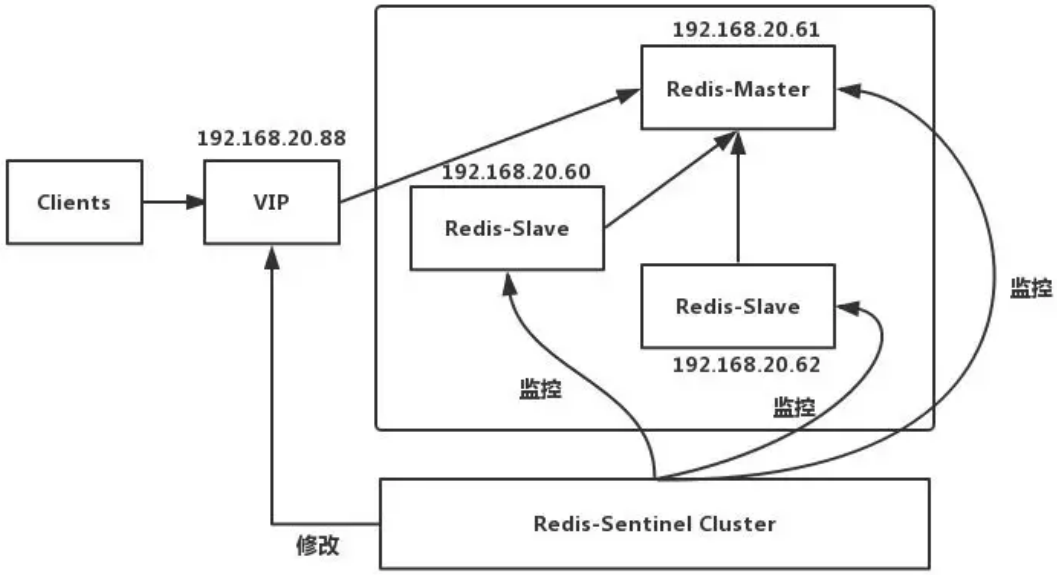


这里Sentinel的作用有三个:

- **监控:**Sentinel 会不断的检查主服务器和从服务器是否正常运行。
- **通知:**当被监控的某个Redis服务器出现问题，Sentinel通过API脚本向管理员或者其他的应用程序发送通知。
- **自动故障转移:**当主节点不能正常工作时，Sentinel会开始一次自动的故障转移操作，它会将与失效主节点是主从关系的其中一个从节点升级为新的主节点，并且将其他的从节点指向新的主节点。

工作原理

当Master宕机的时候，Sentinel会选举出新的Master，并根据Sentinel中client-reconfig-script脚本配置的内容，去动态修改VIP(虚拟IP)，将VIP(虚拟IP)指向新的Master。我们的客户端就连向指定的VIP即可！故障发生后的转移情况，可以理解为下图



缺陷

(1)主从切换的过程中会丢数据
(2)Redis只能单点写，不能水平扩容

常用方案：

内网DNS，VIP和 封装客户端直连Redis Sentinel 端口

内网DNS

底层是 Redis Sentinel 集群，代理着 Redis 主从，Web 端连接内网 DNS 提供服务。内网 DNS 按照一定的规则分配，比如 xxxx.redis.cache/queue.port.xxx.xxx，第一个段表示业务简写，第二个段表示这是 Redis 内网域名，第三个段表示 Redis 类型，cache 表示缓存，queue 表示队列，第四个段表示 Redis 端口，第五、第六个段表示内网主域名。当主节点发生故障，比如机器故障、Redis 节点故障或者网络不可达，Sentinel 集群会调用 client-reconfig- 配置的脚本，修改对应端口的内网域名。对应端口的内网域名指向新的 Redis 主节点。

优点：秒级切换，10秒之内 ；脚本自定义，架构可控；对应用透明，前端不用担心后端发生什么变化

缺点：维护成本高；依赖DNS，存在解析超时；哨兵存在短时间服务不可用；服务时通过外网不可采用

VIP

和第一种方案略有不同，把内网 DNS 换成了虚拟 IP。底层是 Redis Sentinel 集群，代理着 Redis 主从，Web 端通过 VIP 提供服务。在部署 Redis 主从的时候，需要将虚拟 IP 绑定到当前的 Redis 主节点。当主节点发生故障，比如机器故障、Redis 节点故障或者网络不可达，Sentinel 集群会调用 client-reconfig- 配置的脚本，将 VIP 漂移到新的主节点上。

优点：秒级切换，5秒之内 ；脚本自定义，架构可控；对应用透明，前端不用担心后端发生什么变化

缺点：维护成本更高，使用VIP增加维护成本，并存在IP混乱风险

封装客户端直连 Redis Sentinel 端口

这个主要是因为有些业务只能通过外网访问 Redis，于是衍生出了这种方案。Web 使用客户端连接其中一台 Redis Sentinel 集群中的一台机器的某个端口，然后通过这个端口获取到当前的主节点，然后再连接到真实的 Redis 主节点进行相应的业务员操作。需要注意的是，Redis Sentinel 端口和 Redis 主节点均需要开放访问权限。前端业务使用 Java，有 JedisSentinelPool 可以复用。

优点：服务探测故障及时，DBA维护成本低

缺点：依赖客户端支持Sentinel;Sentinel 服务器和 Redis 节点需要开放访问权限;再有 对应用有侵入性

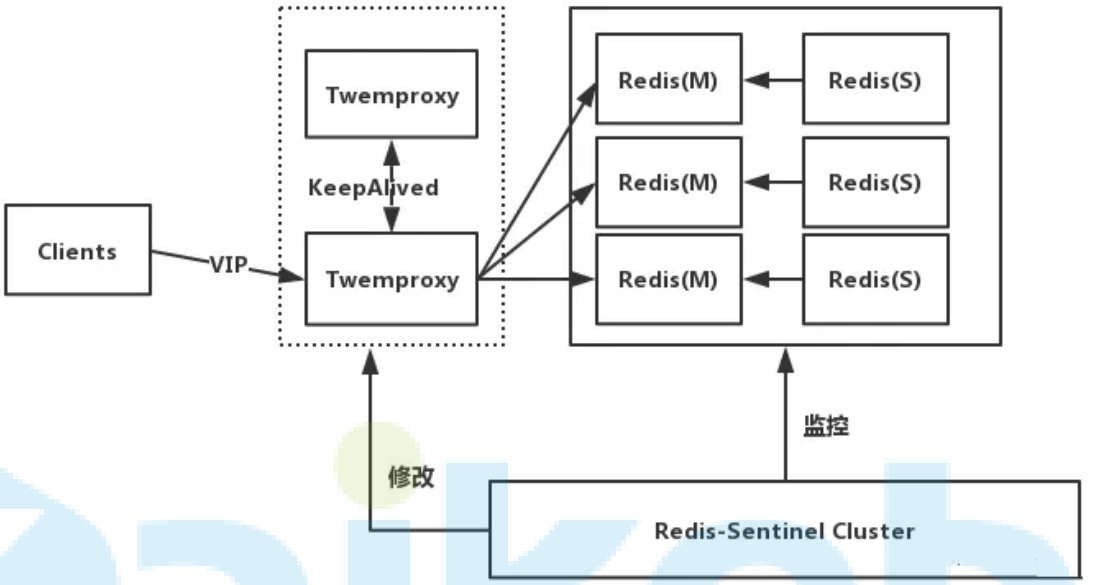
4.3 Proxy+Replication+Sentinel（仅仅了解）

这里的Proxy有两种选择:Codis（豌豆荚）和Twemproxy（推特）。

兔哥经历这套架构的时间为2015年，原因有二:

- 因为Codis开源的比较晚，考虑到更换组件的成本问题。毕竟本来运行好好的东西，你再去换组件，风险是很大的。
- Redis Cluster在2015年还是试用版，不保证会遇到什么问题，因此不敢尝试。

  所以我没接触过Codis，之前一直用的是Twemproxy作为Proxy。
  这里以Twemproxy为例说明，如下图所示



工作原理

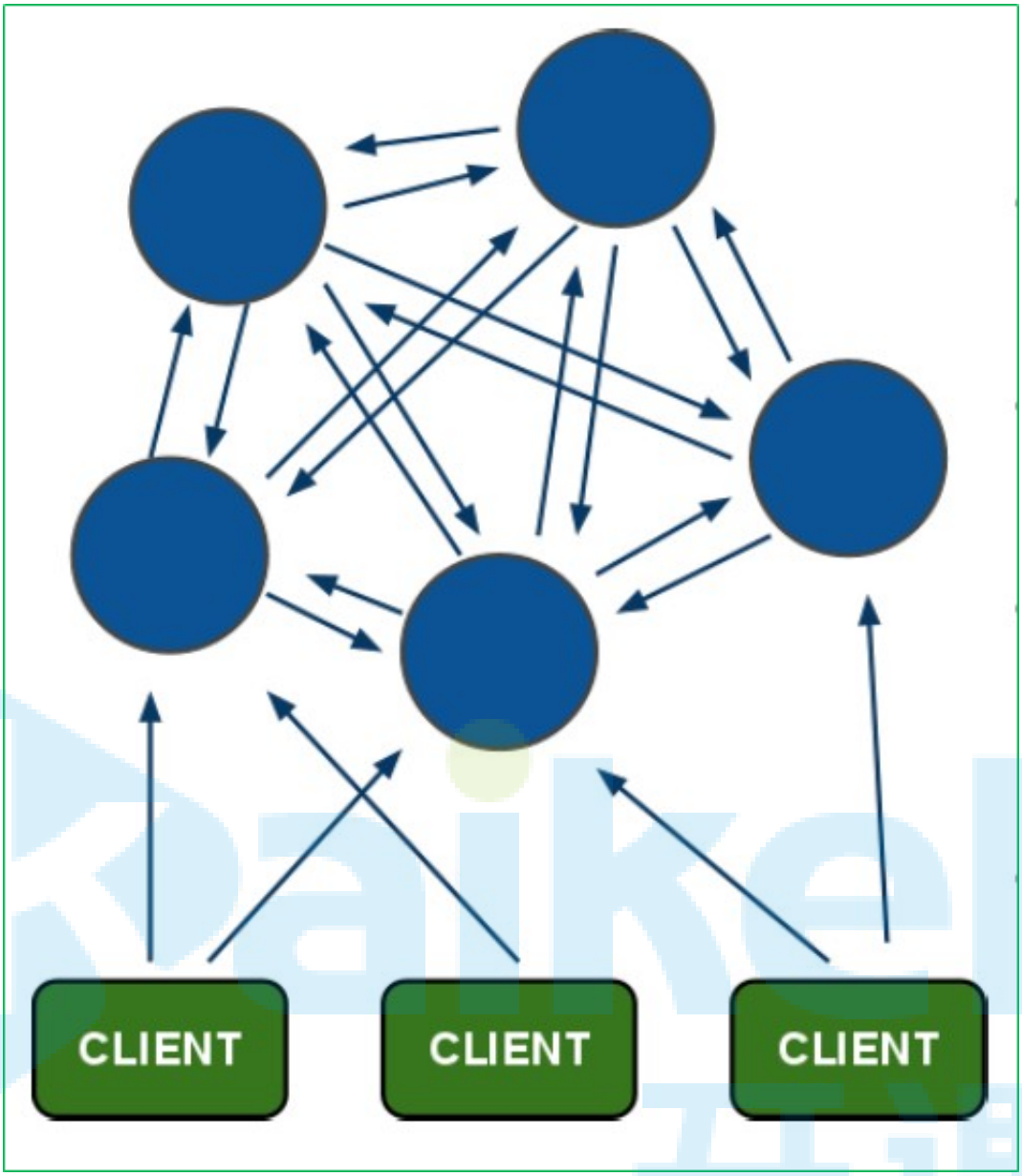
- 前端使用Twemproxy+KeepAlived做代理，将其后端的多台Redis实例分片进行统一管理与分配
- 每一个分片节点的Slave都是Master的副本且只读
- Sentinel持续不断的监控每个分片节点的Master，当Master出现故障且不可用状态时，Sentinel会通知/启动自动故障转移等动作
- Sentinel 可以在发生故障转移动作后触发相应脚本（通过 client-reconfig-script 参数配置），脚本获取到最新的Master来修改Twemproxy配置

缺陷:

(1)部署结构超级复杂
(2)可扩展性差，进行扩缩容需要手动干预
(3)运维不方便

4.4 Redis Cluster 集群（重点）

4.4.1Redis-cluster架构图



架构细节:

(1)所有的redis节点彼此互联(**PING-PONG机制**),内部使用二进制协议优化传输速度和带宽.

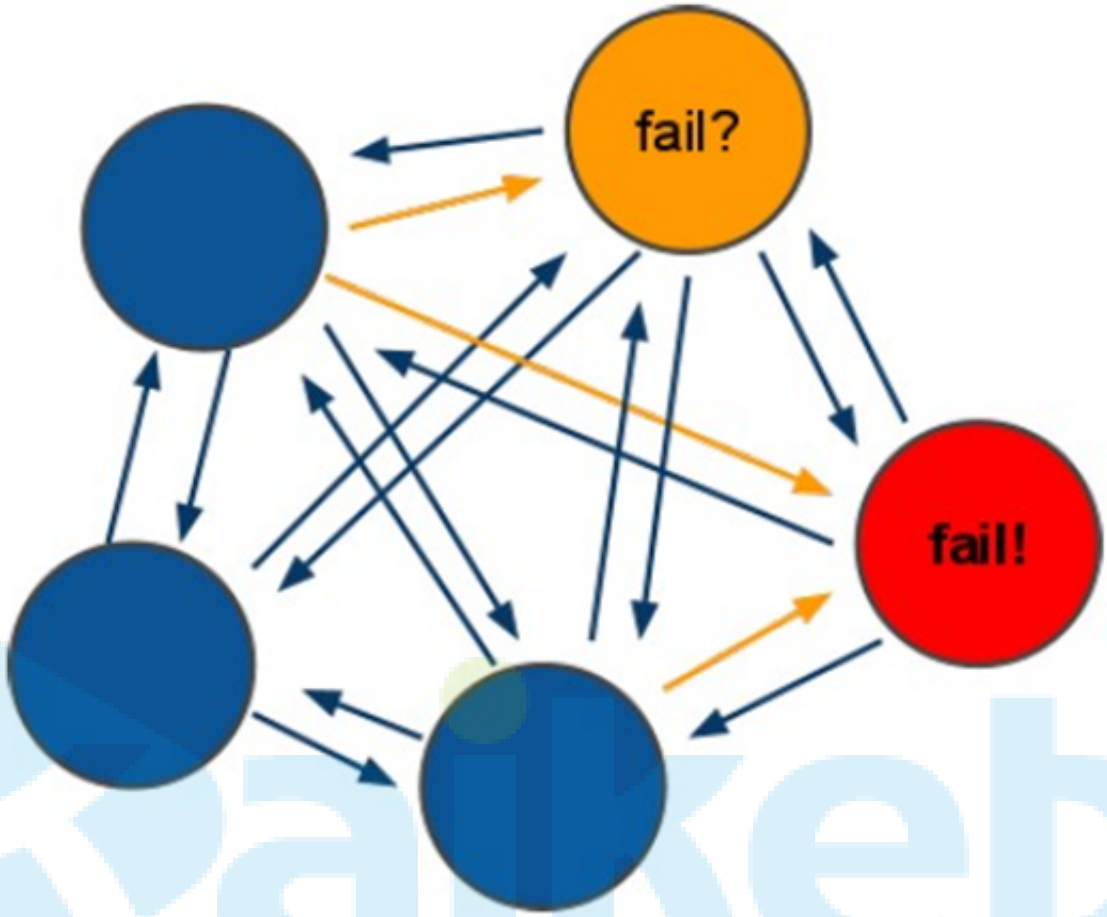
(2)节点的fail是通过集群中超过半数的节点检测失效时才生效.

(3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

(4)redis-cluster把所有的物理节点映射到[0-16383]**slot**上,cluster 负责维护**node<->slot<->value**

```
Redis 集群中内置了 16384个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点
```

4.4.2 Redis-cluster投票:容错



(1)**节点失效判断**：集群中所有master参与投票,如果**半数以上master节点**与其中一个master节点通信超过**(cluster-node-timeout)**,认为该master节点挂掉.

(2)**集群失效判断**:什么时候整个集群不可用(cluster_state:fail)?

- 如果集群任意master挂掉,且当前master没有slave，则集群进入fail状态。也可以理解成集群的[0-16383]slot映射不完全时进入fail状态。
- 如果集群超过半数以上master挂掉，无论是否有slave，集群进入fail状态。

4.4.3 RedisCluster 集群安装

需要开启防火墙，或者直接关闭防火墙。

```
service iptables stop
```

安装RedisCluster

```
Redis集群最少需要三台主服务器，三台从服务器。

端口号分别为：8001~8006
```

- 第一步：创建8001实例，并编辑redis.conf文件，修改port为8001。

  注意：创建实例，即拷贝单机版安装时，生成的bin目录，为8001目录。

- 第二步：修改redis.conf配置文件，打开cluster-enable yes

- 第三步：复制8001，创建8002~8006实例，**注意端口修改**。

```
[root@localhost redis]# cp 8001/ 8002 -r
```

- 第四步：启动所有的实例
- 第五步：创建Redis集群

```
[root@localhost 8001]# ./redis-cli --cluster create 127.0.0.1:7001
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7006 --
cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 127.0.0.1:8005 to 127.0.0.1:8001
Adding replica 127.0.0.1:8006 to 127.0.0.1:8002
Adding replica 127.0.0.1:8004 to 127.0.0.1:8003
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: ab492f8084052c670c59a1594ac19583df6be0a6 127.0.0.1:8001
   slots:[0-5460] (5461 slots) master
M: c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 127.0.0.1:8002
   slots:[5461-10922] (5462 slots) master
M: 40585c9d836cb31aae27ba29d24803ab2f221063 127.0.0.1:8003
   slots:[10923-16383] (5461 slots) master
S: dd19a1123de6b4d7ebe9f5629a45f562bdb3b054 127.0.0.1:8004
   replicates ab492f8084052c670c59a1594ac19583df6be0a6
S: 7daf9dd1c8f8b90f6861717c9f9832dffe8042f6 127.0.0.1:8005
   replicates c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9
S: 009cf3bde8b9d67ba051ddb8b80178bd57d30e4e 127.0.0.1:8006
   replicates 40585c9d836cb31aae27ba29d24803ab2f221063
Can I set the above configuration? (type 'yes' to accept): yse
```

```
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
..
>>> Performing Cluster Check (using node 127.0.0.1:8001)
M: ab492f8084052c670c59a1594ac19583df6be0a6 127.0.0.1:8001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 127.0.0.1:8002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
M: 40585c9d836cb31aae27ba29d24803ab2f221063 127.0.0.1:8003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: 009cf3bde8b9d67ba051ddb8b80178bd57d30e4e 127.0.0.1:8006
   slots: (0 slots) slave
   replicates 40585c9d836cb31aae27ba29d24803ab2f221063
S: 7daf9dd1c8f8b90f6861717c9f9832dffe8042f6 127.0.0.1:8005
   slots: (0 slots) slave
   replicates c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9
S: dd19a1123de6b4d7ebe9f5629a45f562bdb3b054 127.0.0.1:8004
   slots: (0 slots) slave
   replicates ab492f8084052c670c59a1594ac19583df6be0a6
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

命令客户端连接集群

```
./redis-cli –h 127.0.0.1 –p 8001 –c
```

```
[root@localhost 8001]# ./redis-cli -p 8006 -c
```

查看集群的命令

- 查看集群状态

```
127.0.0.1:8006> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
```

```
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:3
cluster_stats_messages_ping_sent:10603
cluster_stats_messages_pong_sent:10272
cluster_stats_messages_meet_sent:3
cluster_stats_messages_sent:20878
cluster_stats_messages_ping_received:10270
cluster_stats_messages_pong_received:10606
cluster_stats_messages_meet_received:2
cluster_stats_messages_received:20878
```

- 查看集群中的节点：

```
127.0.0.1:8006> cluster nodes
40585c9d836cb31aae27ba29d24803ab2f221063 127.0.0.1:8003@18003 master - 0
1590020495000 3 connected 10923-16383
009cf3bde8b9d67ba051ddb8b80178bd57d30e4e 127.0.0.1:8006@18006 myself,slave
40585c9d836cb31aae27ba29d24803ab2f221063 0 1590020494000 6 connected
dd19a1123de6b4d7ebe9f5629a45f562bdb3b054 127.0.0.1:8004@18004 slave
ab492f8084052c670c59a1594ac19583df6be0a6 0 1590020494000 4 connected
7daf9dd1c8f8b90f6861717c9f9832dffe8042f6 127.0.0.1:8005@18005 slave
c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 0 1590020496382 2 connected
ab492f8084052c670c59a1594ac19583df6be0a6 127.0.0.1:8001@18001 master - 0
1590020496000 1 connected 0-5460
c3cb44d45f670b6c16365b54b2a2097ba5b5a1e9 127.0.0.1:8002@18002 master - 0
1590020495376 2 connected 5461-cluster 10922
```

cluster 从节点 默认不可写不可读 仅仅是备份

4.4.4 工作原理

- 客户端与Redis节点直连,不需要中间Proxy层，直接连接任意一个Master节点
- 根据公式HASH_SLOT=CRC16(key) mod 16384，计算出映射到哪个分片上，然后Redis会去相应的节点进行操作

优点:

(1)无需Sentinel哨兵监控，如果Master挂了，Redis Cluster内部自动将Slave切换Master
(2)可以进行水平扩容
(3)支持自动化迁移，当出现某个Slave宕机了，那么就只有Master了，这时候的高可用性就无法很好的保证了，万一Master也宕机了，咋办呢？ 针对这种情况，如果说其他Master有多余的Slave ，集群自动把多余的Slave迁移到没有Slave的Master 中。

缺点:

(1)批量操作是个坑
(2)资源隔离性较差，容易出现相互影响的情况。

4.4.5 主从切换

当集群中节点通过错误检测机制发现某个节点处于fail状态时，会执行主从切换。Redis 还提供了手动切换的方法，即通过执行 cluster failover 命令

自动切换

切换流程如下（假设被切换的主节点为M，执行切换的从节点为S）

```
1．S先更新自己的状态，将声明自己为主节点。并且将S从M中移除
2．由于S需要切换为主节点，所以将S的同步数据相关信息清除（即不再从M同步锁数据）
3．将M提供服务的slot都声明到S中
4．发送一个PONG包，通知集群中其他节点更新状态
```

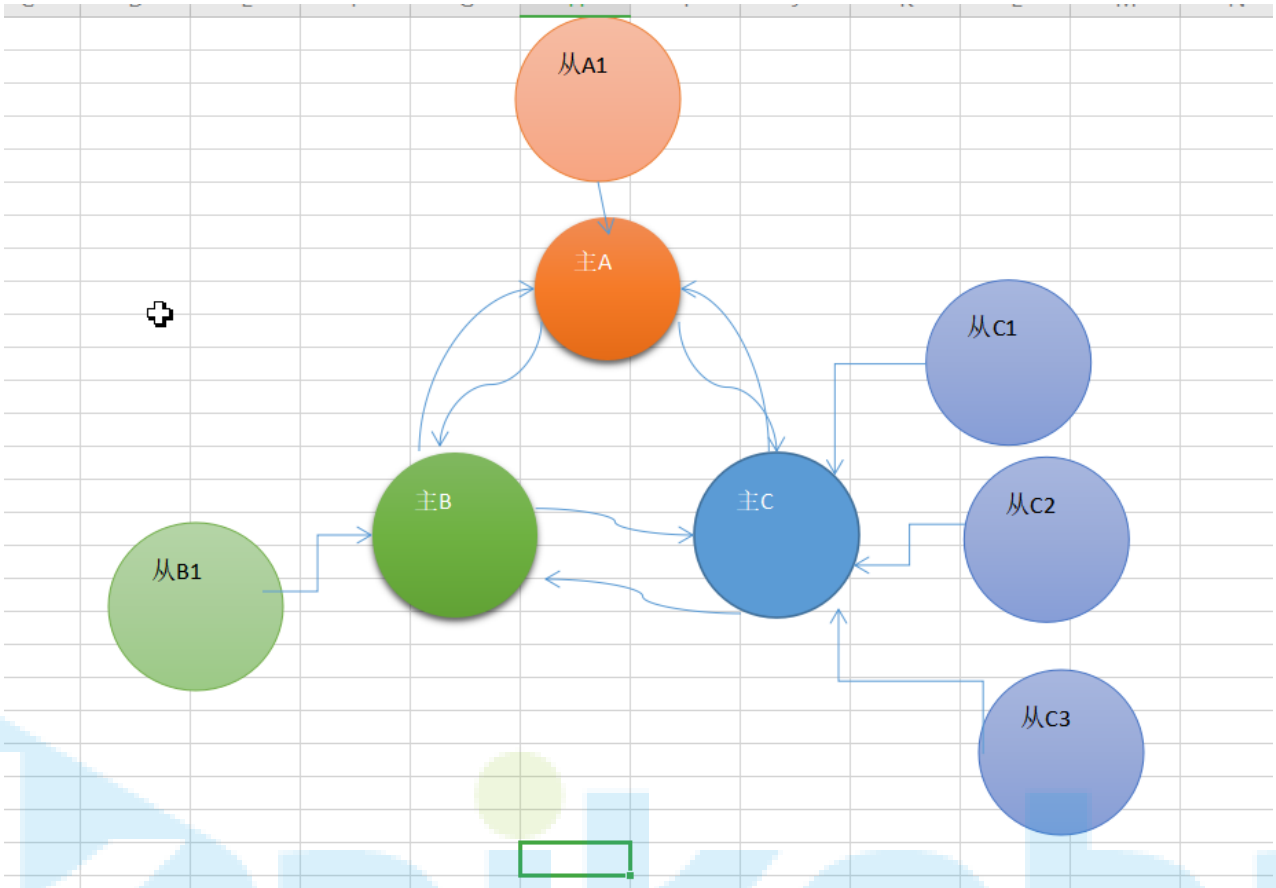
手动切换

当一个节点接受到 cluster failove 命令之后，执行手动切换，流程如下

1. 该从节点首先向主节点发送一个mfstart包。通知主节点从节点开始进行手动切换
2. 主节点会阻塞所有客户端指令的执行。之后主节点在周期函数clusterCron中发送ping 包时会在包头部分做特殊标记
3. 当从节点收到主节点的ping包并且检测到特殊标记之后，会从包头中获取主节点的复制偏移量
4. 从节点在周期函数clusterCron中检测当前处理的复制偏移量与主节点复制偏移量是否相等，当相等时开始执行切换流程
5. 切换完成后，主节点会讲阻塞的所有客户端命令通过发送+MOVED 指令重定向到新的主节点

通过流程可以看到，手动执行主从切换流程时不会丢失任何数据，也不会丢失任何执行命令，只在切换过程中会有暂时的停顿

4.4.6 副本漂移

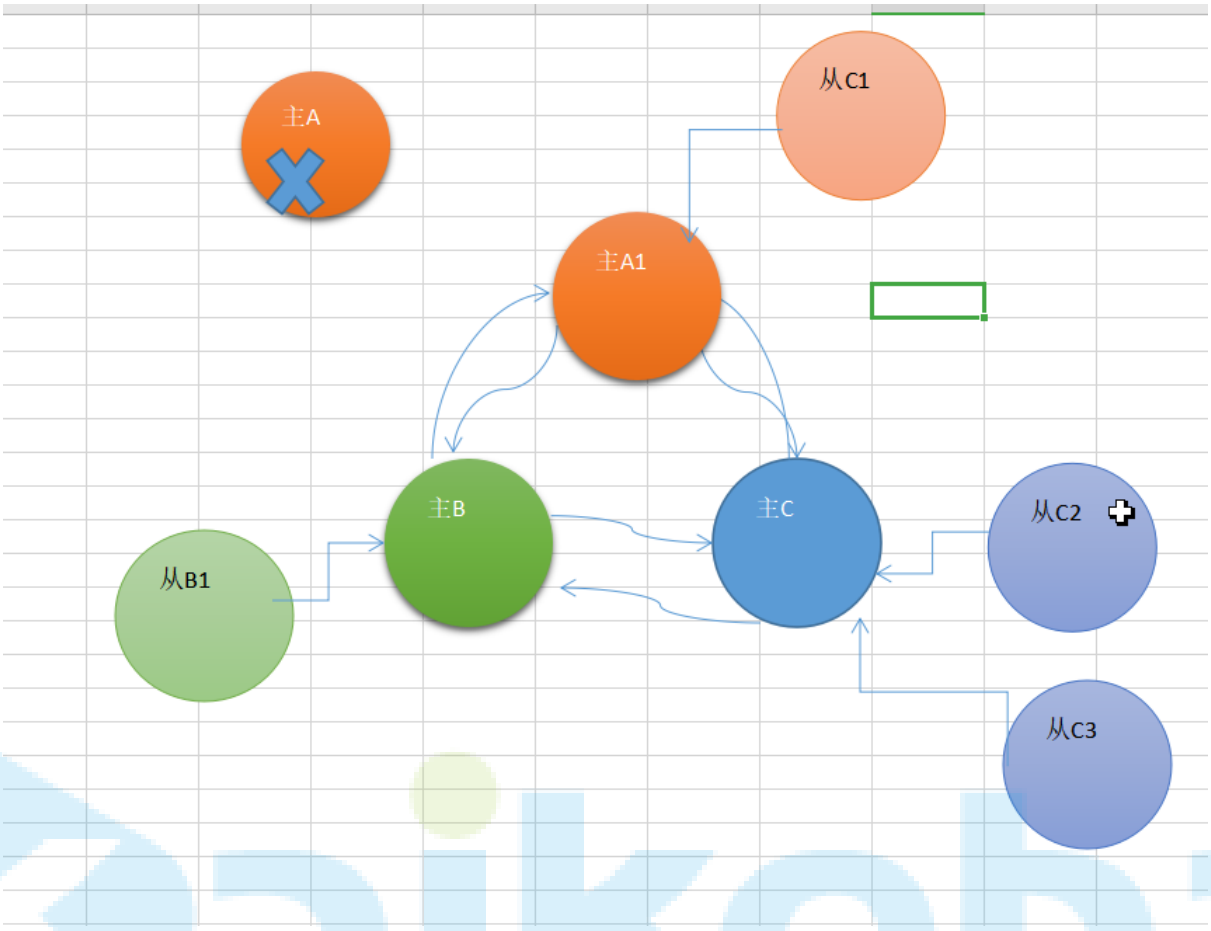


假设A发生故障,主A的A1会执行切换，切换完成后A1变为主A1，此时主A1会出现单点问题。

在周期性调度函数 clusterCron中会定期检查如下条件

1. 是否存在单点的主节点，即主节点没有任何一台可用的从节点
2. 是否存在有两台及以上可用从节点的主节点

   如果以上两个条件都满足，从有最多可用从节点中选择一台从节点执行副本漂移。选择标准为按节点名称从小到大，选择最靠前的一台从节点执行漂移。具体漂移过程



3. 从C的记录中将C1移除
4. 将C1所记录的主节点更改为A1
5. 在A1中添加C1从节点
6. 将C1的数据同步源设置为A1

漂移过程只是更改一些节点所记录的信息，之后会通过心跳包将该信息同步到所有的集群节点。

4.4.7 分片迁移

添加节点

当集群需要扩容时需要向集群中添加新的节点。

我们首先创建一个新的redis实例，端口为8007。其他配置同上。

启动7007节点，然后使用`redis-cli`将新的节点添加到集群中。

```
./redis-cli --cluster add-node 新节点的ip:端口 现有集群ip:端口
```

例如：

```
[root@localhost redis07]# ./redis-cli --cluster add-node 192.168.133.22:8007
192.168.133.22:8001
>>> Adding node 192.168.23.129:7007 to cluster 192.168.23.129:7001
>>> Performing Cluster Check (using node 192.168.23.129:7001)
```

```
S: bc87c70f91a3c01d4fafc731f29905ba7279c7e4 192.168.23.129:7001
   slots: (0 slots) slave
   replicates 4c268aafd7eb918d99d126a8027aae352eb4f914
M: b93055c58f1aa493583cc2c003a6ac5cf12aaa3d 192.168.23.129:7004
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
M: 94217791c8a325ac9cd439f3b2f47d3e115e6fc0 192.168.23.129:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: c4c08a72b7ec58a3d7220c9f1332f7b3973a1290 192.168.23.129:7002
   slots: (0 slots) slave
   replicates b93055c58f1aa493583cc2c003a6ac5cf12aaa3d
S: 717257d8372bda5444b76721b00e9d23d1091ff0 192.168.23.129:7005
   slots: (0 slots) slave
   replicates 94217791c8a325ac9cd439f3b2f47d3e115e6fc0
M: 4c268aafd7eb918d99d126a8027aae352eb4f914 192.168.23.129:7006
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 192.168.23.129:7007 to make it join the cluster.
[OK] New node added correctly.
[root@localhost redis07]#
```

查看集群中的节点信息：

```
192.168.23.129:7006> cluster nodes
c1be46df89b5cb18ad94bfe1033eeb225adaa518 192.168.23.129:7007@17007 master - 0
1581303676120 0 connected
c4c08a72b7ec58a3d7220c9f1332f7b3973a1290 192.168.23.129:7002@17002 slave
b93055c58f1aa493583cc2c003a6ac5cf12aaa3d 0 1581303676000 8 connected
94217791c8a325ac9cd439f3b2f47d3e115e6fc0 192.168.23.129:7003@17003 master - 0
1581303677129 3 connected 10923-16383
717257d8372bda5444b76721b00e9d23d1091ff0 192.168.23.129:7005@17005 slave
94217791c8a325ac9cd439f3b2f47d3e115e6fc0 0 1581303673058 5 connected
4c268aafd7eb918d99d126a8027aae352eb4f914 192.168.23.129:7006@17006
myself,master - 0 1581303675000 7 connected 0-5460
bc87c70f91a3c01d4fafc731f29905ba7279c7e4 192.168.23.129:7001@17001 slave
4c268aafd7eb918d99d126a8027aae352eb4f914 0 1581303676000 7 connected
b93055c58f1aa493583cc2c003a6ac5cf12aaa3d 192.168.23.129:7004@17004 master - 0
1581303675108 8 connected 5461-10922
```

分配slot

我们可以看到新的节点已经添加到集群中了，但是新的节点中并没有任何槽在其上，也就意味着，不可能有数据放到这个节点上。那么我们需要将一部分槽移动到新的节点上。

> ./redis-cli --cluster reshard 192.168.133.22:8001 --cluster-from f3ab5f5c53b82b8a9df24bfd4a8191224cf63597,faa53cbb6bae288d7f3a19fa55f18d7e77c97f6d,d33eab99055e05440a6ba1dd314efb0a25274fb1 --cluster-to 90dcd8e79fd753d24c35ef6ec63e65d79f56600a 3000

--cluster-from：表示slot目前所在的节点的node ID，多个ID用逗号分隔

--cluster-to：表示需要新分配节点的node ID（貌似每次只能分配一个）

--cluster-slots：分配的slot数量

添加从节点

我们向集群中添加了一个新的节点7007，如果只有一个主节点的情况下，高可用性会下降。那么我们就应该给这个节点增加一个备份节点提高可用性。我们再添加一个节点7008，按照之前的配置方法配置，然后启动。

> ./redis-cli --cluster add-node 192.168.133.22:8008 192.168.133.22:8007 --cluster-slave --cluster-master-id 90dcd8e79fd753d24c35ef6ec63e65d79f56600a

add-node: 后面的分别跟着新加入的slave和slave对应的master

cluster-slave：表示加入的是slave节点

--cluster-master-id：表示slave对应的master的node ID

```
[root@localhost redis08]# ./redis-cli --cluster add-node 192.168.133.22:8008
192.168.133.22:8007 --cluster-slave --cluster-master-id
a514f02c7afffd37d29ac2b4d86f407d806963af
>>> Adding node 192.168.23.129:7008 to cluster 192.168.23.129:7007
>>> Performing Cluster Check (using node 192.168.23.129:7007)
M: c1be46df89b5cb18ad94bfe1033eeb225adaa518 192.168.23.129:7007
   slots:[0-1332],[5461-6794],[10923-12255] (4000 slots) master
S: bc87c70f91a3c01d4fafc731f29905ba7279c7e4 192.168.23.129:7001
   slots: (0 slots) slave
   replicates 4c268aafd7eb918d99d126a8027aae352eb4f914
M: 4c268aafd7eb918d99d126a8027aae352eb4f914 192.168.23.129:7006
   slots:[1333-5460] (4128 slots) master
   1 additional replica(s)
M: 94217791c8a325ac9cd439f3b2f47d3e115e6fc0 192.168.23.129:7003
   slots:[12256-16383] (4128 slots) master
   1 additional replica(s)
S: c4c08a72b7ec58a3d7220c9f1332f7b3973a1290 192.168.23.129:7002
   slots: (0 slots) slave
   replicates b93055c58f1aa493583cc2c003a6ac5cf12aaa3d
S: 717257d8372bda5444b76721b00e9d23d1091ff0 192.168.23.129:7005
   slots: (0 slots) slave
```

```
   replicates 94217791c8a325ac9cd439f3b2f47d3e115e6fc0
M: b93055c58f1aa493583cc2c003a6ac5cf12aaa3d 192.168.23.129:7004
   slots:[6795-10922] (4128 slots) master
   1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 192.168.23.129:7008 to make it join the cluster.
Waiting for the cluster to join

>>> Configure node as replica of 192.168.23.129:7007.
[OK] New node added correctly.
[root@localhost redis08]#
```

查看节点状态

```
192.168.23.129:7003> cluster nodes
c1be46df89b5cb18ad94bfe1033eeb225adaa518 192.168.23.129:7007@17007 master - 0
1581309576000 9 connected 0-1332 5461-6794 10923-12255
94217791c8a325ac9cd439f3b2f47d3e115e6fc0 192.168.23.129:7003@17003
myself,master - 0 1581309580000 3 connected 12256-16383
717257d8372bda5444b76721b00e9d23d1091ff0 192.168.23.129:7005@17005 slave
94217791c8a325ac9cd439f3b2f47d3e115e6fc0 0 1581309581000 5 connected
bc87c70f91a3c01d4fafc731f29905ba7279c7e4 192.168.23.129:7001@17001 slave
4c268aafd7eb918d99d126a8027aae352eb4f914 0 1581309580488 7 connected
b93055c58f1aa493583cc2c003a6ac5cf12aaa3d 192.168.23.129:7004@17004 master - 0
1581309581497 8 connected 6795-10922
c4c08a72b7ec58a3d7220c9f1332f7b3973a1290 192.168.23.129:7002@17002 slave
b93055c58f1aa493583cc2c003a6ac5cf12aaa3d 0 1581309578000 8 connected
4c268aafd7eb918d99d126a8027aae352eb4f914 192.168.23.129:7006@17006 master - 0
1581309578000 7 connected 1333-5460
e21ab9b3dbfd20f2c3dd74e4a987366086c3ecb5 192.168.23.129:7008@17008 slave
c1be46df89b5cb18ad94bfe1033eeb225adaa518 0 1581309579000 9 connected
```

收缩集群

下线节点192.168.23.129:7007（master）/192.168.23.129:7008（slave）

（1）首先删除master对应的slave

> ./redis-cli --cluster del-node 192.168.133.22:8008
> 5e8924bef3cfb0a07838d45da77e9a2e0e61d7b9

del-node后面跟着集群中的任意节点的ip及端口号（主要目的是连接到集群，ip:port） 和node ID

（2）清空master的slot

> ./redis-cli --cluster reshard 192.168.133.22:8007 --cluster-from 90dcd8e79fd753d24c35ef6ec63e65d79f56600a --cluster-to f3ab5f5c53b82b8a9df24bfd4a8191224cf63597 --cluster-slots 3000 --cluster-yes

reshard子命令前面已经介绍过了，这里需要注意的一点是，由于我们的集群一共有四个主节点，而每次reshard只能写一个目的节点，因此以上命令需要执行三次（--cluster-to对应不同的目的节点）。

--cluster-yes：不回显需要迁移的slot，直接迁移。

（3）下线（删除）节点

> ./redis-cli --cluster del-node 192.168.133.22:8007 90dcd8e79fd753d24c35ef6ec63e65d79f56600

附录

springboot 连接Redis集群

```
// 1.sentinel模式

server:
  port: 80
spring:
  redis:
    sentinel:
      nodes: 192.168.0.106:26379,192.168.0.106:26380,192.168.0.106:26381 //哨兵的ip和端口
      master: mymaster //这个就是哨兵配置文件中 sentinel monitor mymaster 192.168.0.103 6379 2 配置的mymaster
// 2.Cluster模式
server:
  port: 80
spring:
  redis:
    cluster:
      nodes: 192.168.0.106:7000,192.168.0.106:7001,192.168.0.106:7002,192.168.0.106:7003,192.168.0.106:7004,192.168.0.106:7005
```

Redis版本历史 （增加了解）

1. Redis2.6 Redis2.6在2012年正式发布
2. Redis2.8
   Redis2.8在2013年11月22日正式发布
   Redis Sentinel第二版，相比于Redis2.6的Redis Sentinel，此版本已经变成生产可用。
3. Redis3.0
   Redis3.0在2015年4月1日正式发布

4. Redis3.2
   Redis3.2在2016年5月6日正式发布，集群高可用
5. Redis4.0
   Redis 4.0在2017年7月发布为GA，主要是增加了混合持久化和LFU淘汰策略
6. Redis5.0
   Redis5.0 2018年10月18日正式发布，Stream 是重要新增特性

Redis 5.0 源码清单（对源码感兴趣的，看一下）

1. 基本数据结构
   动态字符串sds.c
   整数集合intset.c
   压缩列表ziplist.c
   快速链表quicklist.c
   字典dict.c
2. Redis数据类型的底层实现
   Redis对象object.c
   字符串t_string.c
   列表t_list.c
   字典t_hash.c
   集合及有序集合t_set.c和t_zset.c
3. Redis数据库的实现
   数据库的底层实现db.c
   持久化rdb.c和aof.c
4. Redis服务端和客户端实现
   事件驱动ae.c和ae_epoll.c
   网络连接anet.c和networking.c
   服务端程序server.c
   客户端程序redis-cli.c
5. 集群相关
   主从复制replication.c
   哨兵sentinel.c
   集群cluster.c
6. 特殊数据类型
   其他数据结构，如hyperloglog.c、geo.c
   数据流t_stream.c
   Streams的底层实现结构listpack.c和rax.c

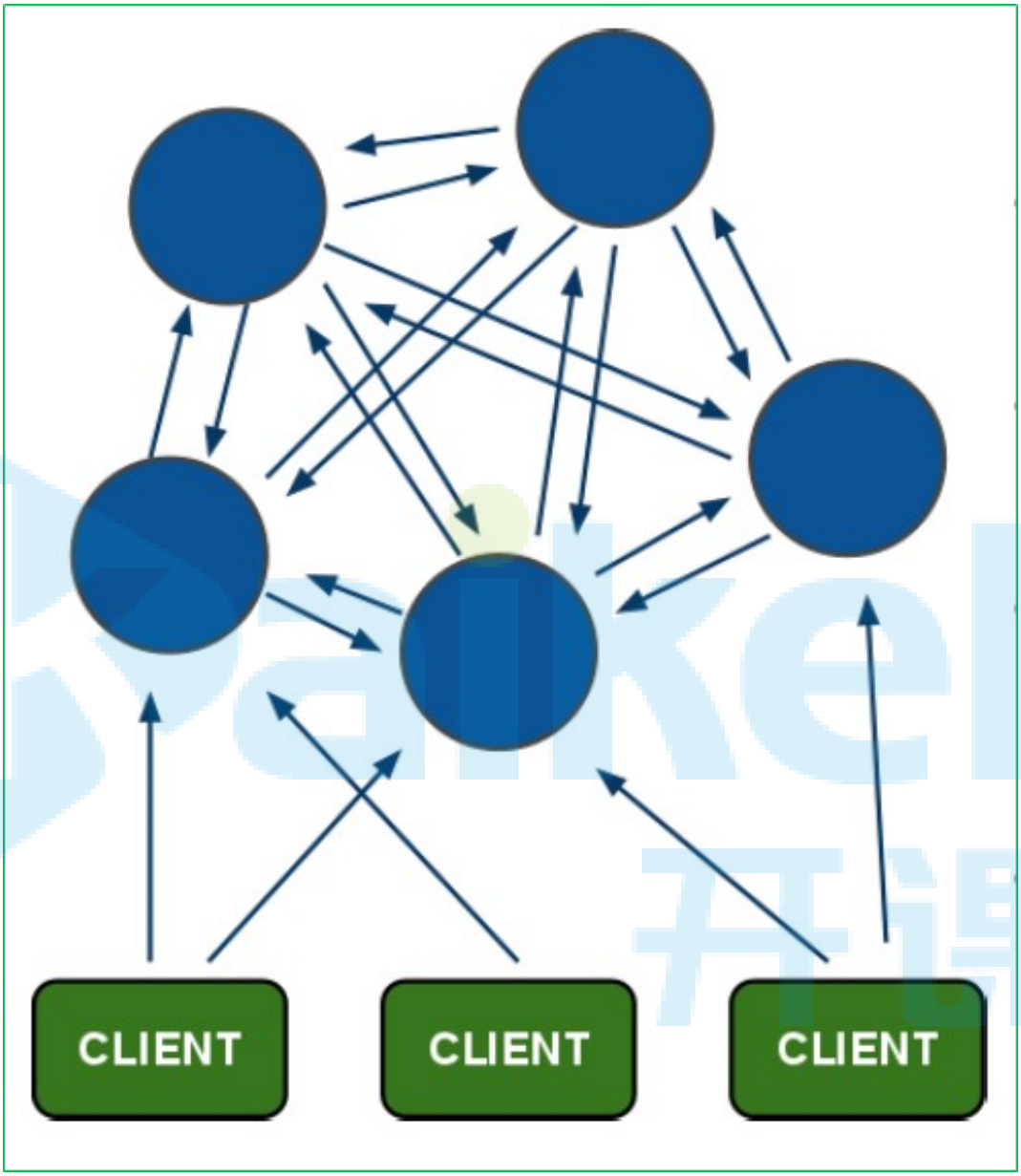
Redis集群

redis3.0以后推出的redis cluster 集群方案，redis cluster集群保证了高可用、高性能、高可扩展性。

Redis的集群策略

- 推特：twemproxy
- 豌豆荚：codis
- 官方：redis cluster

Redis-cluster架构图



架构细节:

(1)所有的redis节点彼此互联(**PING-PONG机制**),内部使用二进制协议优化传输速度和带宽.

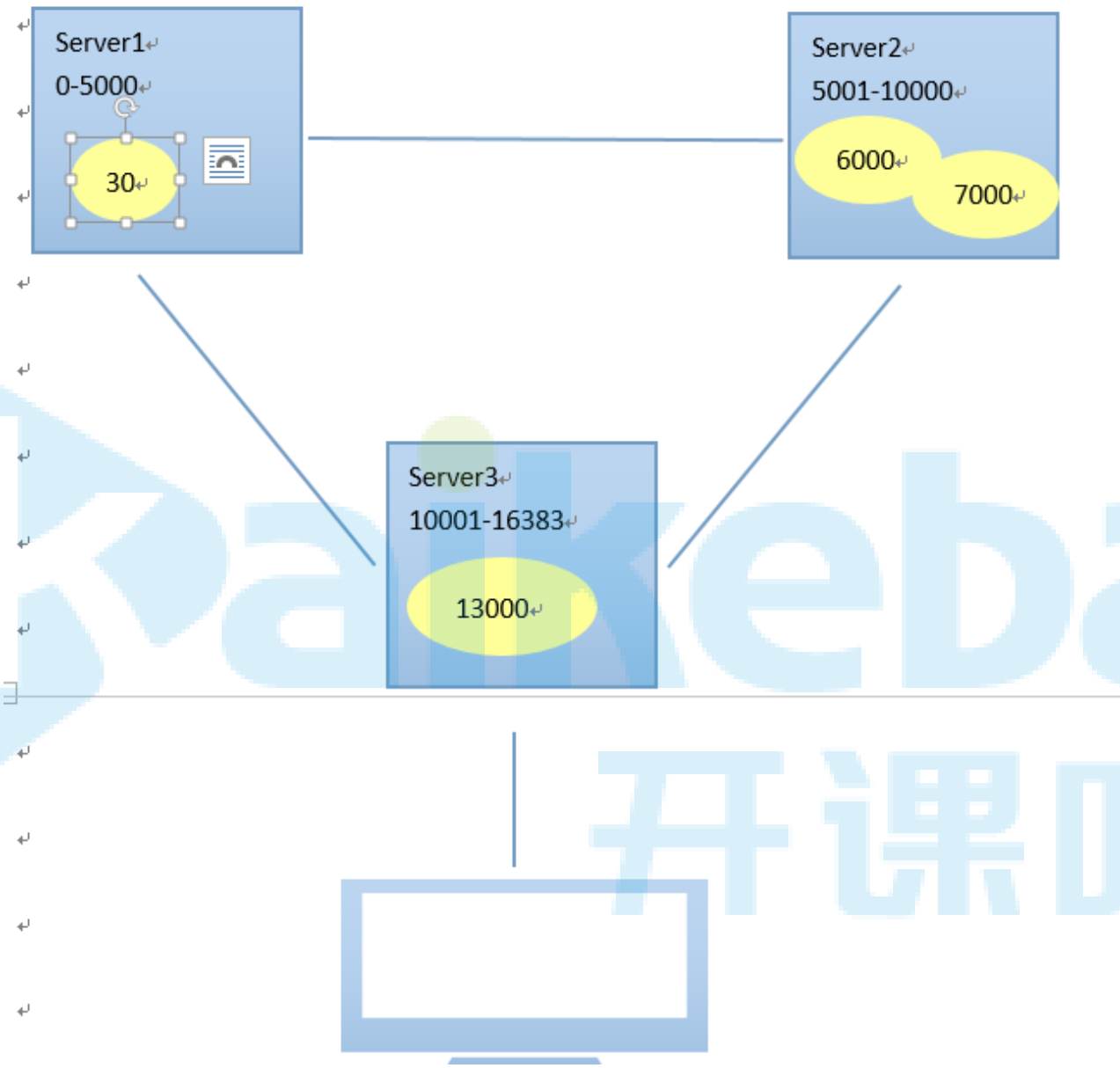
(2)节点的fail是通过集群中超过半数的节点检测失效时才生效.

(3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

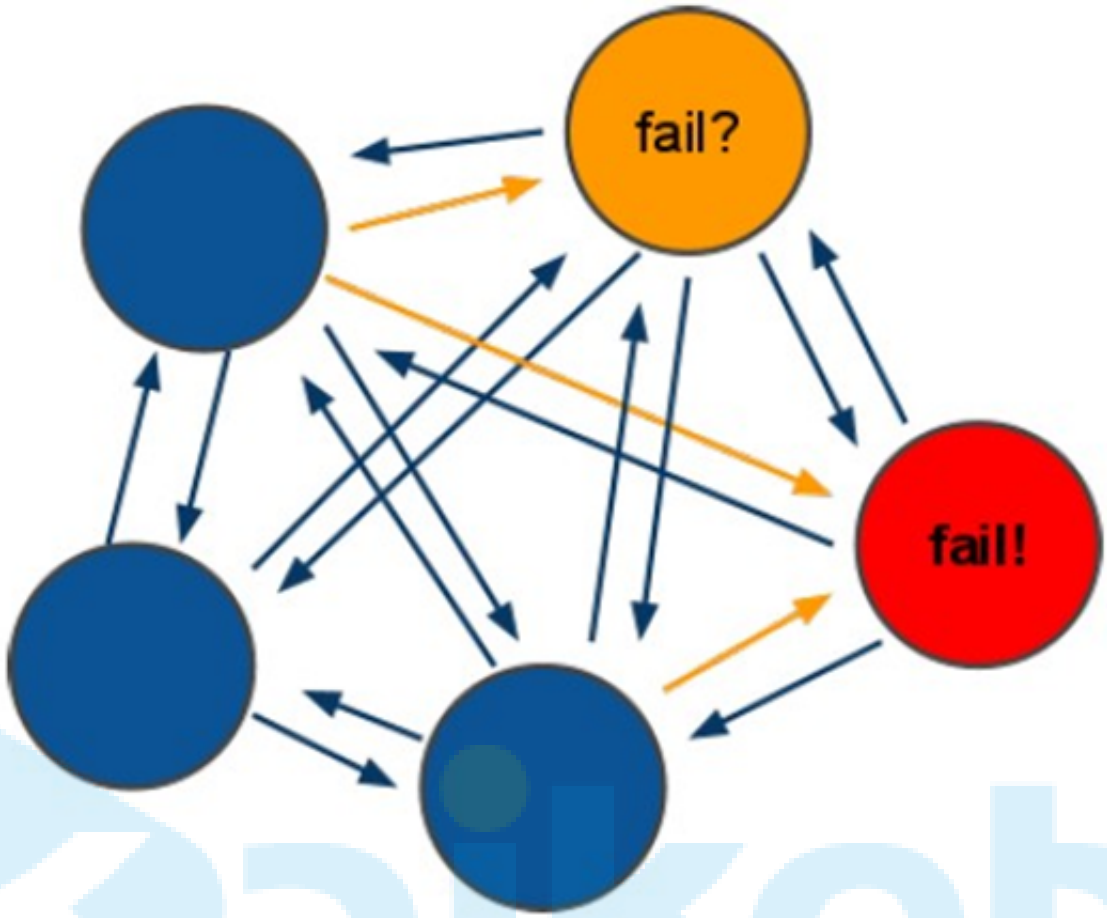
(4)redis-cluster把所有的物理节点映射到[0-16383]**slot**上,cluster 负责维护**node<->slot<->value**

Redis 集群中内置了 16384个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

示例如下：



Redis-cluster投票:容错



(1)**节点失效判断**：集群中所有master参与投票,如果**半数以上master节点**与其中一个master节点通信超过**(cluster-node-timeout)**,认为该master节点挂掉.

(2)**集群失效判断**:什么时候整个集群不可用(cluster_state:fail)?

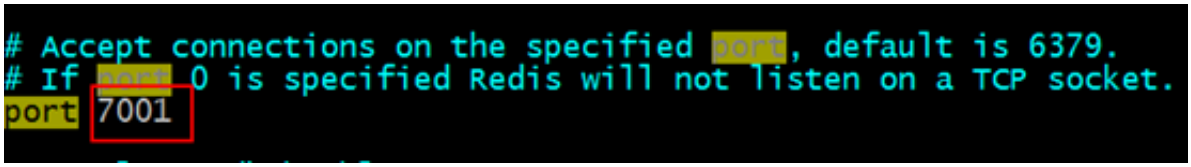
- 如果集群任意master挂掉,且当前master没有slave，则集群进入fail状态。也可以理解成集群的[0-16383]slot映射不完全时进入fail状态。
- 如果集群超过半数以上master挂掉，无论是否有slave，集群进入fail状态。

安装RedisCluster

```
Redis集群最少需要三台主服务器，三台从服务器。

端口号分别为：7001~7006
```

- 第一步：创建7001实例，并编辑redis.conf文件，修改port为7001。

  注意：创建实例，即拷贝单机版安装时，生成的bin目录，为7001目录。



- 第二步：修改redis.conf配置文件，打开Cluster-enable yes

- 第三步：复制7001，创建7002~7006实例，**注意端口修改**。
- 第四步：启动所有的实例

  vim start-cluster.sh

```
cd 7001
./redis-server redis.conf
cd ..
cd 7002
./redis-server redis.conf
cd ..
cd 7003
./redis-server redis.conf
cd ..
cd 7004
./redis-server redis.conf
cd ..
cd 7005
./redis-server redis.conf
cd ..
cd 7006
./redis-server redis.conf
cd ..
```

  chmod u+x start-cluster.sh

- 第五步：创建Redis集群

```
./redis-cli --cluster create 39.105.201.207:7001 39.105.201.207:7002
39.105.201.207:7003 39.105.201.207:7004 39.105.201.207:7005
39.105.201.207:7006 --cluster-replicas 1
>>> Creating cluster
Connecting to node 192.168.10.133:7001: OK
Connecting to node 192.168.10.133:7002: OK
Connecting to node 192.168.10.133:7003: OK
Connecting to node 192.168.10.133:7004: OK
Connecting to node 192.168.10.133:7005: OK
Connecting to node 192.168.10.133:7006: OK
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
192.168.10.133:7001
192.168.10.133:7002
192.168.10.133:7003
Adding replica 192.168.10.133:7004 to 192.168.10.133:7001
Adding replica 192.168.10.133:7005 to 192.168.10.133:7002
Adding replica 192.168.10.133:7006 to 192.168.10.133:7003
M: d8f6a0e3192c905f0aad411946f3ef9305350420 192.168.10.133:7001
   slots:0-5460 (5461 slots) master
M: 7a12bc730ddc939c84a156f276c446c28acf798c 192.168.10.133:7002
   slots:5461-10922 (5462 slots) master
M: 93f73d2424a796657948c660928b71edd3db881f 192.168.10.133:7003
   slots:10923-16383 (5461 slots) master
S: f79802d3da6b58ef6f9f30c903db7b2f79664e61 192.168.10.133:7004
   replicates d8f6a0e3192c905f0aad411946f3ef9305350420
S: 0bc78702413eb88eb6d7982833a6e040c6af05be 192.168.10.133:7005
   replicates 7a12bc730ddc939c84a156f276c446c28acf798c
S: 4170a68ba6b7757e914056e2857bb84c5e10950e 192.168.10.133:7006
   replicates 93f73d2424a796657948c660928b71edd3db881f
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join....
>>> Performing Cluster Check (using node 192.168.10.133:7001)
M: d8f6a0e3192c905f0aad411946f3ef9305350420 192.168.10.133:7001
   slots:0-5460 (5461 slots) master
M: 7a12bc730ddc939c84a156f276c446c28acf798c 192.168.10.133:7002
   slots:5461-10922 (5462 slots) master
M: 93f73d2424a796657948c660928b71edd3db881f 192.168.10.133:7003
   slots:10923-16383 (5461 slots) master
M: f79802d3da6b58ef6f9f30c903db7b2f79664e61 192.168.10.133:7004
   slots: (0 slots) master
   replicates d8f6a0e3192c905f0aad411946f3ef9305350420
M: 0bc78702413eb88eb6d7982833a6e040c6af05be 192.168.10.133:7005
   slots: (0 slots) master
   replicates 7a12bc730ddc939c84a156f276c446c28acf798c
```

```
M: 4170a68ba6b7757e914056e2857bb84c5e10950e 192.168.10.133:7006
   slots: (0 slots) master
   replicates 93f73d2424a796657948c660928b71edd3db881f
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
[root@localhost-0723 redis]#
```

命令客户端连接集群

命令：

```
./redis-cli –h 127.0.0.1 –p 7001 –c
```

注意：**-c** 表示是以redis集群方式进行连接

```
./redis-cli -p 7006 -c
127.0.0.1:7006> set key1 123
-> Redirected to slot [9189] located at 127.0.0.1:7002
OK
127.0.0.1:7002>
```

查看集群的命令

- 查看集群状态

```
127.0.0.1:7003> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:3
cluster_stats_messages_sent:926
cluster_stats_messages_received:926
```

- 查看集群中的节点：

```
127.0.0.1:7003> cluster nodes
7a12bc730ddc939c84a156f276c446c28acf798c 127.0.0.1:7002 master - 0
1443601739754 2 connected 5461-10922
93f73d2424a796657948c660928b71edd3db881f 127.0.0.1:7003 myself,master - 0 0 3
connected 10923-16383
d8f6a0e3192c905f0aad411946f3ef9305350420 127.0.0.1:7001 master - 0
1443601741267 1 connected 0-5460
4170a68ba6b7757e914056e2857bb84c5e10950e 127.0.0.1:7006 slave
93f73d2424a796657948c660928b71edd3db881f 0 1443601739250 6 connected
f79802d3da6b58ef6f9f30c903db7b2f79664e61 127.0.0.1:7004 slave
d8f6a0e3192c905f0aad411946f3ef9305350420 0 1443601742277 4 connected
0bc78702413eb88eb6d7982833a6e040c6af05be 127.0.0.1:7005 slave
7a12bc730ddc939c84a156f276c446c28acf798c 0 1443601740259 5 connected
127.0.0.1:7003>
```

维护节点

集群创建成功后可以继续向集群中添加节点

添加主节点

- 先创建7007节点
- 添加7007结点作为新节点

  执行命令：

```
./redis-cli --cluster add-node 127.0.0.1:7007 127.0.0.1:7001
```

```
[root@server01 7007]# ./redis-trib.rb add-node  192.168.101.3:7007 192.168.101.3:7001
>>> Adding node 192.168.101.3:7007 to cluster 192.168.101.3:7001
Connecting to node 192.168.101.3:7001: OK
Connecting to node 192.168.101.3:7003: OK
Connecting to node 192.168.101.3:7006: OK
Connecting to node 192.168.101.3:7002: OK
Connecting to node 192.168.101.3:7005: OK
Connecting to node 192.168.101.3:7004: OK
>>> Performing Cluster Check (using node 192.168.101.3:7001)
M: cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001
   slots:0-5460 (5461 slots) master
   1 additional replica(s)
M: 1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003
   slots:10923-16383 (5461 slots) master
   1 additional replica(s)
S: 444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006
   slots: (0 slots) slave
   replicates 1a8420896c3ff60b70c716e8480de8e50749ee65
M: 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002
   slots:5461-10922 (5462 slots) master
   1 additional replica(s)
S: d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005
   slots: (0 slots) slave
   replicates 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841
S: 69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004
   slots: (0 slots) slave
   replicates cad9f7413ec6842c971dbcc2c48b4ca959eb5db4
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
Connecting to node 192.168.101.3:7007: OK
>>> Send CLUSTER MEET to node 192.168.101.3:7007 to make it join the cluster.
[OK] New node added correctly.
```

- 查看集群结点发现7007已添加到集群中

```
192.168.101.3:7005> cluster nodes
69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430155626174 4 connected
444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006 slave 1a8420896c3ff60b70c716e8480de8e50749ee65 0 1430155621629 9 connected
4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002 master - 0 1430155627185 2 connected 5461-10922
d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005 myself,slave 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 0 0 5 connected
1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003 master - 0 1430155625164 9 connected 10923-16383
15b809eadae88955e36bcdbb8144f61bbbaf38fb 192.168.101.3:7007 master - 0 1430155628700 0 connected
cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001 master - 0 1430155628197 1 connected 0-5460
```

hash槽重新分配（数据迁移）

添加完主节点需要对主节点进行hash槽分配，这样该主节才可以存储数据。

- **查看集群中槽占用情况**

```
cluster nodes
```

redis集群有16384个槽，集群中的每个结点分配自已槽，通过查看集群结点可以看到槽占用情况。

```
192.168.101.3:7005> cluster nodes
69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430155241550 4 connected
444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006 slave 1a8420896c3ff60b70c716e8480de8e50749ee65 0 1430155240540 9 connected
4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002 master - 0 1430155239532 2 connected 5461-10922
d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005 myself,slave 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 0 0 5 connected
1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003 master - 0 1430155243568 9 connected 10923-16383
cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001 master - 0 1430155242560 1 connected 0-5460
192.168.101.3:7005>
```

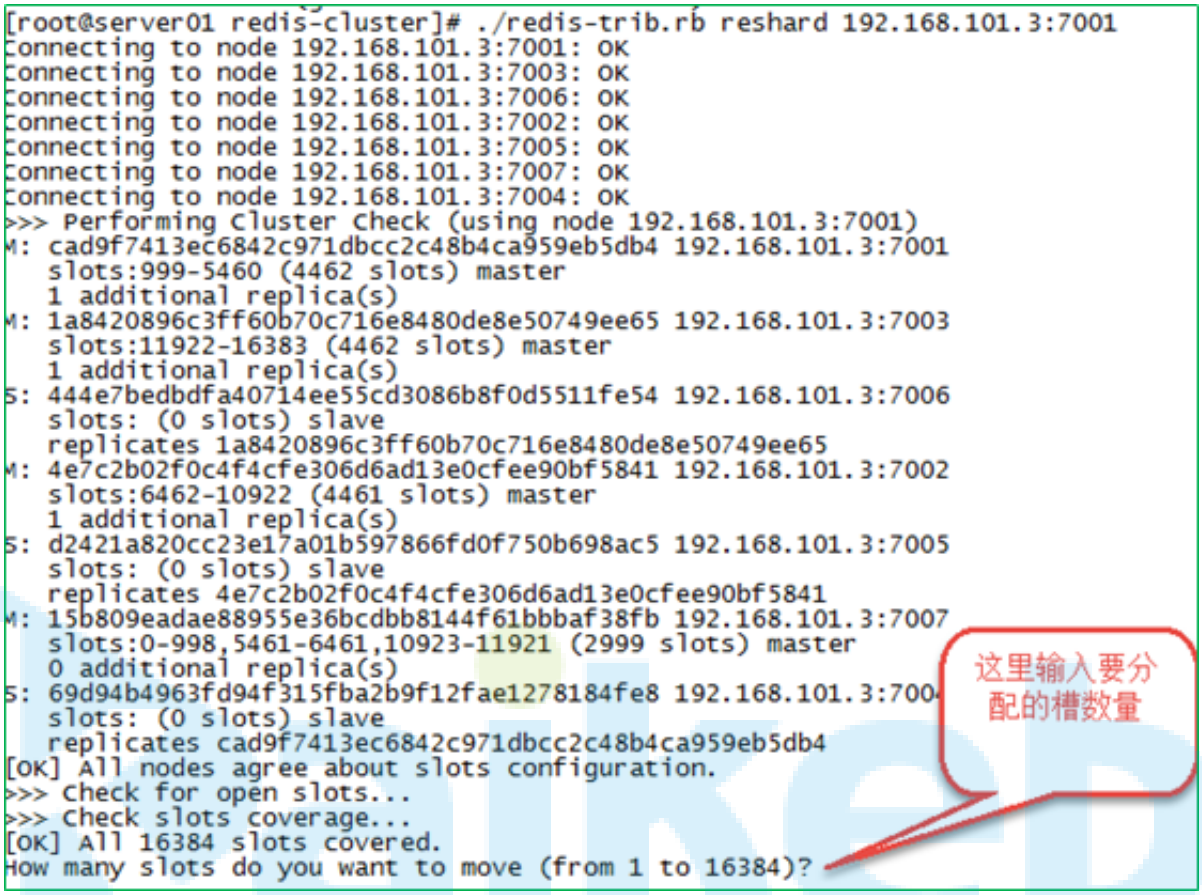
每个master结点都分配了一定数量的槽

给刚添加的7007结点分配槽

- **第一步：连接上集群（连接集群中任意一个可用结点都行）**

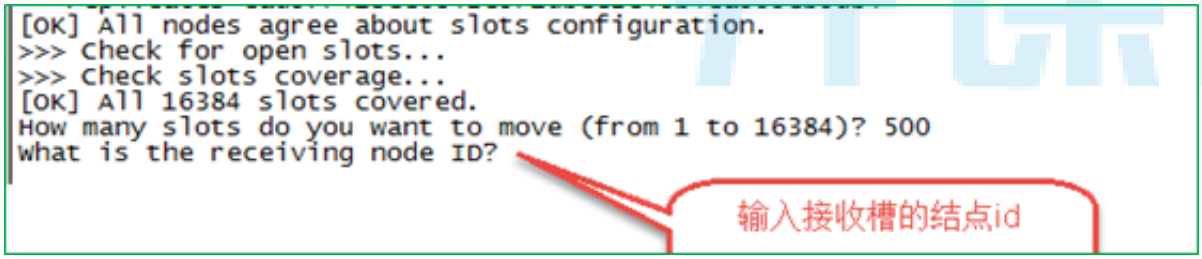
```
./redis-cli --cluster reshard 127.0.0.1:7007
```

- **第二步：输入要分配的槽数量**



输入：**3000**，表示要给目标节点分配3000个槽

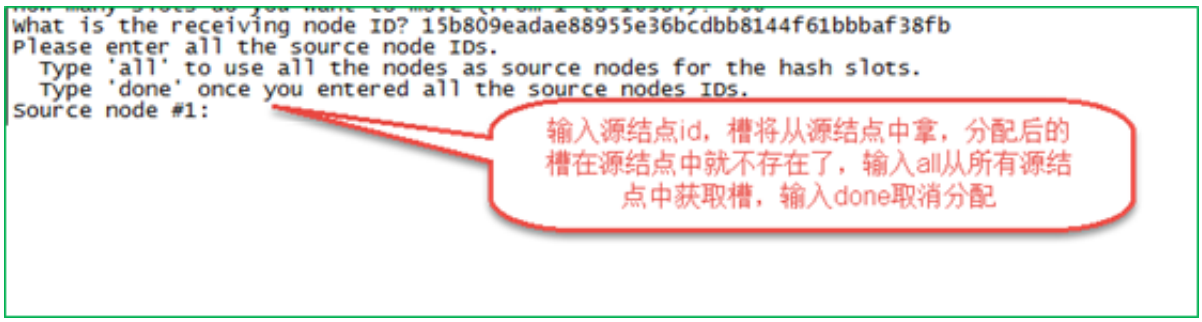
- **第三步：输入接收槽的结点id**



输入：**15b809eadae88955e36bcdbb8144f61bbbaf38fb**

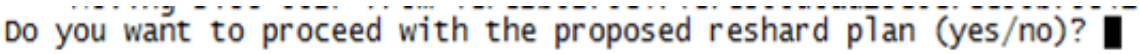
```
PS：这里准备给7007分配槽，通过cluster nodes查看7007结点id为：

15b809eadae88955e36bcdbb8144f61bbbaf38fb
```

- **第四步：输入源结点id**



输入：**all**

- **第五步：输入yes开始移动槽到目标结点id**



输入：**yes**

添加从节点

- **添加7008从结点，将7008作为7007的从结点**

命令：

```
./redis-cli --cluster add-node 新节点的ip和端口    旧节点ip和端口 --cluster-slave --cluster-master-id 主节点id
```

例如：

```
./redis-cli --cluster add-node 127.0.0.1:7008 127.0.0.1:7007 --cluster-slave --cluster-master-id d1ba0092526cdfe66878e8879d446acfdcde25d8
```

d1ba0092526cdfe66878e8879d446acfdcde25d8是7007结点的id，可通过cluster nodes查看。

```
[root@server01 redis-cluster]# ./redis-trib.rb add-node --slave --master-id cad9f7413ec6842c971dbcc2c48b4ca959eb5db4  192.168.101.3:7008 192.168.101.3:7001
>>> Adding node 192.168.101.3:7008 to cluster 192.168.101.3:7001
Connecting to node 192.168.101.3:7001: OK
Connecting to node 192.168.101.3:7003: OK
Connecting to node 192.168.101.3:7006: OK
Connecting to node 192.168.101.3:7002: OK
Connecting to node 192.168.101.3:7005: OK
Connecting to node 192.168.101.3:7007: OK
Connecting to node 192.168.101.3:7004: OK
>>> Performing cluster check (using node 192.168.101.3:7001)
M: cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001
   slots:1166-5460 (4295 slots) master
   1 additional replica(s)
M: 1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003
   slots:12088-16383 (4296 slots) master
   1 additional replica(s)
S: 444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006
   slots: (0 slots) slave
   replicates 1a8420896c3ff60b70c716e8480de8e50749ee65
M: 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002
   slots:6628-10922 (4295 slots) master
   1 additional replica(s)
S: d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005
   slots: (0 slots) slave
   replicates 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841
M: 15b809eadae88955e36bcdbb8144f61bbbaf38fb 192.168.101.3:7007
   slots:0-1165,5461-6627,10923-12087 (3498 slots) master
   0 additional replica(s)
S: 69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004
   slots: (0 slots) slave
   replicates cad9f7413ec6842c971dbcc2c48b4ca959eb5db4
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
Connecting to node 192.168.101.3:7008: OK
>>> Send CLUSTER MEET to node 192.168.101.3:7008 to make it join the cluster.
Waiting for the cluster to join.
>>> Configure node as replica of 192.168.101.3:7001.
[OK] New node added correctly.
```

注意：如果原来该结点在集群中的配置信息已经生成到cluster-config-file指定的配置文件中（如果cluster-config-file没有指定则默认为**nodes.conf**），这时可能会报错：

```
[ERR] Node XXXXXX is not empty. Either the node already knows other nodes
(check with CLUSTER NODES) or contains some key in database 0
```

解决方法是删除生成的配置文件nodes.conf，删除后再执行./redis-trib.rb add-node指令

- **查看集群中的结点，刚添加的7008为7007的从节点：**

```
192.168.101.3:7005> cluster nodes
05dbaf8059630157c245dfe5441dbe9c26b9016d 192.168.101.3:7008 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430157051979 1 connected
69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430157046926 4 connected
444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006 slave 1a8420896c3ff60b70c716e8480de8e50749ee65 0 1430157051878 9 connected
4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002 master - 0 1430157049956 2 connected 6628-10922
d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005 myself,slave 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 0 0 5 connected
1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003 master - 0 1430157050968 9 connected 12088-16383
15b809eadae88955e36bcdbb8144f61bbbaf38fb 192.168.101.3:7007 master - 0 1430157052990 10 connected 0-1165 5461-6627 10923-12087
cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001 master - 0 1430157048946 1 connected 1166-5460
```

删除结点

命令：

```
./redis-cli --cluster del-node 127.0.0.1:7008
41592e62b83a8455f07f7797f1d5c071cffedb50
```

删除已经占有hash槽的结点会失败，报错如下：

```
[ERR] Node 127.0.0.1:7005 is not empty! Reshard data away and try again.
```

需要将该结点占用的hash槽分配出去（参考hash槽重新分配章节）。

Jedis连接集群

需要开启防火墙，或者直接关闭防火墙。

```
service iptables stop
```

代码实现

创建JedisCluster类连接redis集群。

```
@Test
public void testJedisCluster() throws Exception {
  //创建一连接，JedisCluster对象,在系统中是单例存在
  Set<HostAndPort> nodes = new HashSet<>();
  nodes.add(new HostAndPort("192.168.10.133", 7001));
  nodes.add(new HostAndPort("192.168.10.133", 7002));
  nodes.add(new HostAndPort("192.168.10.133", 7003));
  nodes.add(new HostAndPort("192.168.10.133", 7004));
  nodes.add(new HostAndPort("192.168.10.133", 7005));
  nodes.add(new HostAndPort("192.168.10.133", 7006));
  JedisCluster cluster = new JedisCluster(nodes);
  //执行JedisCluster对象中的方法，方法和redis一一对应。
  cluster.set("cluster-test", "my jedis cluster test");
  String result = cluster.get("cluster-test");
  System.out.println(result);
  //程序结束时需要关闭JedisCluster对象
  cluster.close();
}
```

使用spring

Ø 配置applicationContext.xml

```
<!-- 连接池配置 -->
<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
  <!-- 最大连接数 -->
  <property name="maxTotal" value="30" />
  <!-- 最大空闲连接数 -->
  <property name="maxIdle" value="10" />
  <!-- 每次释放连接的最大数目 -->
  <property name="numTestsPerEvictionRun" value="1024" />
```

```
  <!-- 释放连接的扫描间隔（毫秒） -->
  <property name="timeBetweenEvictionRunsMillis" value="30000" />
  <!-- 连接最小空闲时间 -->
  <property name="minEvictableIdleTimeMillis" value="1800000" />
  <!-- 连接空闲多久后释放, 当空闲时间>该值 且 空闲连接>最大空闲连接数 时直接释放 -->
  <property name="softMinEvictableIdleTimeMillis" value="10000" />
  <!-- 获取连接时的最大等待毫秒数,小于零:阻塞不确定的时间,默认-1 -->
  <property name="maxWaitMillis" value="1500" />
  <!-- 在获取连接的时候检查有效性, 默认false -->
  <property name="testOnBorrow" value="true" />
  <!-- 在空闲时检查有效性, 默认false -->
  <property name="testWhileIdle" value="true" />
  <!-- 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true -->
  <property name="blockWhenExhausted" value="false" />
</bean>
<!-- redis集群 -->
<bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">
  <constructor-arg index="0">
    <set>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7001"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7002"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7003"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7004"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7005"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7006"></constructor-arg>
      </bean>
    </set>
  </constructor-arg>
  <constructor-arg index="1" ref="jedisPoolConfig"></constructor-arg>
</bean>
```

Ø 测试代码

```
private ApplicationContext applicationContext;
  @Before
  public void init() {
    applicationContext = new ClassPathXmlApplicationContext(
        "classpath:applicationContext.xml");
  }

  // redis集群
  @Test
  public void testJedisCluster() {
    JedisCluster jedisCluster = (JedisCluster) applicationContext
        .getBean("jedisCluster");

    jedisCluster.set("name", "zhangsan");
    String value = jedisCluster.get("name");
    System.out.println(value);
  }
```

Redis和lua整合

什么是lua

lua是一种轻量小巧的**脚本语言**，用标准**C语言**编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

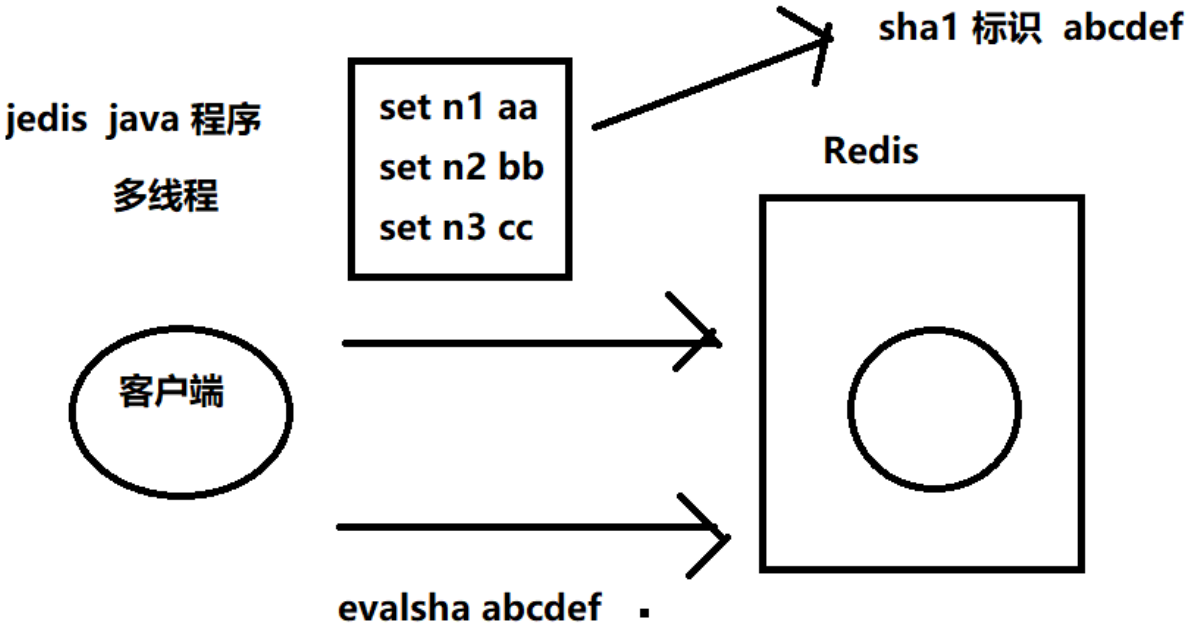
Redis中使用lua的好处

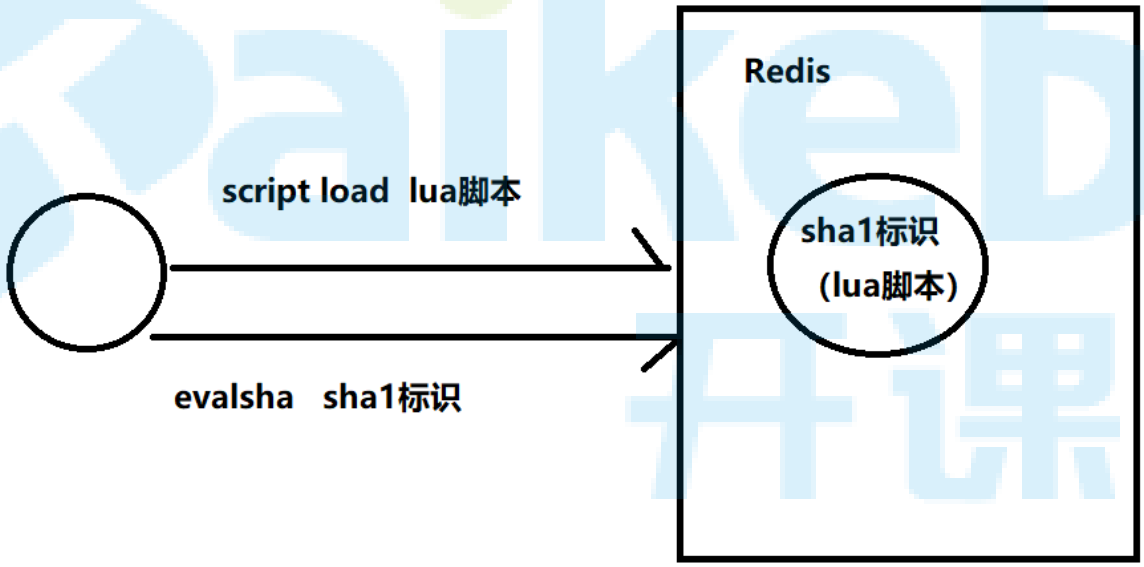
1、减少网络开销，在Lua脚本中可以把多个命令放在同一个脚本中运行

2、原子操作，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。换句话说，编写脚本的过程中无需担心会出现竞态条件

3、复用性，客户端发送的脚本会永远存储在redis中，这意味着其他客户端可以复用这一脚本来完成同样的逻辑

在内存中生成一个sha1 标识 （script load ）





lua的安装（了解）

- 下载

  地址：http://www.lua.org/download.html

  可以本地下载上传到linux，也可以使用curl命令在linux系统中进行在线下载

  ```
  curl -R -O http://www.lua.org/ftp/lua-5.3.5.tar.gz
  ```

- 安装

```
yum -y install readline-devel ncurses-devel
tar -zxvf lua-5.3.5.tar.gz
make linux
make install
```

如果报错，说找不到readline/readline.h, 可以通过yum命令安装

```
yum -y install readline-devel ncurses-devel
```

安装完以后再

```
make linux  / make install
```

最后，直接输入 lua命令即可进入lua的控制台

lua常见语法（了解）

详见http://www.runoob.com/lua/lua-tutorial.html

Redis整合lua脚本

从Redis2.6.0版本开始，通过**内置的lua编译/解释器**，可以使用EVAL命令对lua脚本进行求值。

Redis命令行

EVAL命令

通过执行redis的eval命令，可以运行一段lua脚本。

```
EVAL script numkeys key [key ...] arg [arg ...]
```

命令说明：**

- **script参数**：是一段Lua脚本程序，它会被运行在Redis服务器上下文中，这段脚本不必(也不应该)定义为一个Lua函数。
- **numkeys参数**：用于指定键名参数的个数。
- **key [key ...]参数**： 从EVAL的第三个参数开始算起，使用了numkeys个键（key），表示在脚本中所用到的那些Redis键(key)，这些键名参数可以在Lua中通过全局变量**KEYS**数组，用1为基址的形式访问(KEYS[1] ， KEYS[2] ，以此类推)。
- **arg [arg ...]参数**：可以在Lua中通过全局变量**ARGV**数组访问，访问的形式和KEYS变量类似(ARGV[1] 、 ARGV[2] ，诸如此类)。

```
eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
```

lua脚本中调用Redis命令

- redis.call()：
  - 返回值就是redis命令执行的返回值
  - 如果出错，则返回错误信息，不继续执行
- **redis.pcall()：**
  - 返回值就是redis命令执行的返回值
  - 如果出错，则记录错误信息，继续执行
- **注意事项**
  - 在脚本中，使用return语句将返回值返回给客户端，如果没有return，则返回nil

```
eval "return redis.call('set',KEYS[1],ARGV[1])" 1 n1 zhaoyun
```

SCRIPT命令

- **SCRIPT FLUSH**： 清除所有脚本缓存
- **SCRIPT EXISTS**： 根据给定的脚本校验和，检查指定的脚本是否存在于脚本缓存
- **SCRIPT LOAD**： 将一个脚本装入脚本缓存，**返回SHA1摘要**，但并不立即运行它

```
192.168.24.131:6380> script load "return redis.call('set',KEYS[1],ARGV[1])"
"c686f316aaf1eb01d5a4de1b0b63cd233010e63d"
192.168.24.131:6380> evalsha c686f316aaf1eb01d5a4de1b0b63cd233010e63d 1 n2
zhangfei
OK
192.168.24.131:6380> get n2
```

- **SCRIPT KILL**： 杀死当前正在运行的脚本

EVALSHA

EVAL 命令要求你在每次执行脚本的时候都发送一次脚本主体(script body)。

Redis 有一个内部的缓存机制，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

为了减少带宽的消耗， Redis 实现了 EVALSHA 命令，它的作用和 EVAL 一样，都用于对脚本求值，但它接受的第一个参数不是脚本，而是脚本的 SHA1 校验和(sum)

Linux命令行

redis-cli --eval

直接执行lua脚本

test.lua

```
return redis.call('set',KEYS[1],ARGV[1])

./redis-cli -h 192.168.24.131 -p 6380  --eval test.lua n3 , 'liubei'
```

先去创建list集合

```
127.0.0.1:6380> lpush list 1 2 3 4 5
(integer) 5
127.0.0.1:6380>
```

list.lua

```
local key=KEYS[1]

local list=redis.call("lrange",key,0,-1);

return list;
```

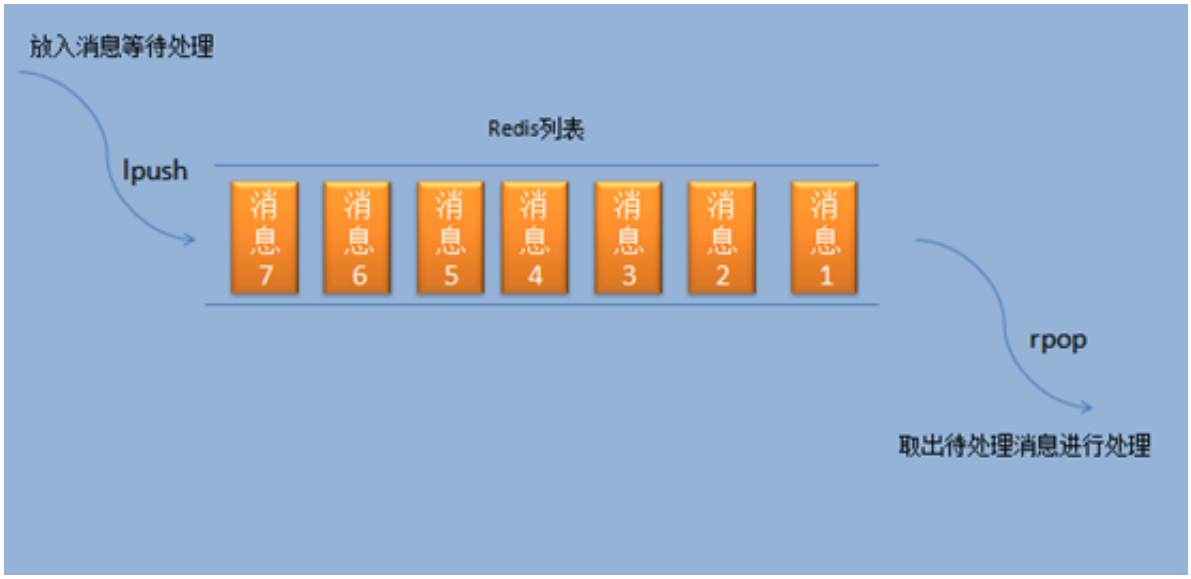
```
./redis-cli --eval list.lua list
```

利用Redis整合Lua，主要是为了性能以及事务的原子性。因为redis帮我们提供的事务功能太差。

Redis消息模式

队列模式（1v1）

使用list类型的lpush和rpop实现消息队列

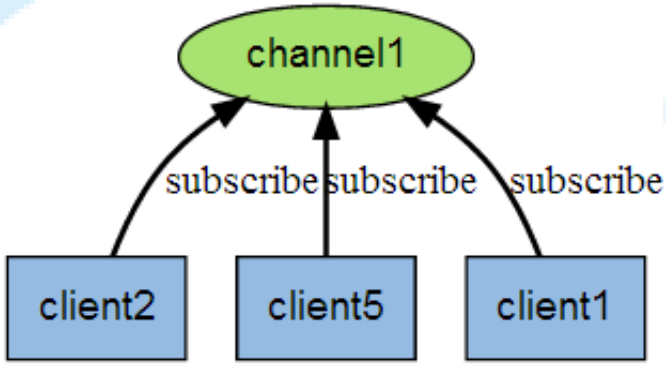


注意事项：

- 消息接收方如果不知道队列中是否有消息，会一直发送rpop命令，如果这样的话，会每一次都建立一次连接，这样显然不好。
- 可以使用**brpop**命令，它如果从队列中取不出来数据，会一直阻塞，在一定范围内没有取出则返回null、

发布订阅模式(1vN)

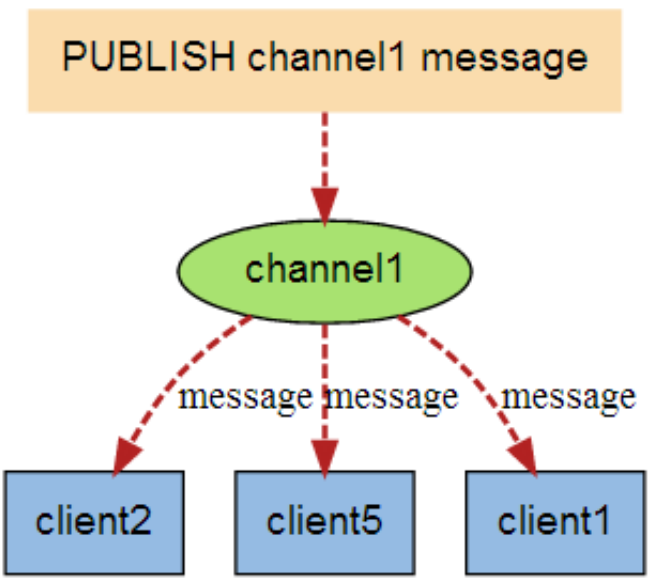
- 订阅消息（**subscribe**）



示例：

```
subscribe kkb-channel
```

- 发布消息（**publish**）




```
publish kkb-channel “我是灭霸詹”
```

- Redis发布订阅命令

Redis 发布订阅命令

下表列出了 redis 发布订阅常用命令：

序号	命令及描述
1	PSUBSCRIBE pattern [pattern ...] 订阅一个或多个符合给定模式的频道。
2	PUBSUB subcommand [argument [argument ...]] 查看订阅与发布系统状态。
3	PUBLISH channel message 将信息发送到指定的频道。
4	PUNSUBSCRIBE [pattern [pattern ...]] 退订所有给定模式的频道。
5	SUBSCRIBE channel [channel ...] 订阅给定的一个或多个频道的信息。
6	UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

四 Redis Stream

Redis 5.0 全新的数据类型：streams，官方把它定义为：以更抽象的方式建模日志的数据结构。Redis的streams主要是一个**append only**（AOF）的数据结构，至少在概念上它是一种在内存中表示的抽象数据类型，只不过它们实现了更强大的操作，以克服日志文件本身的限制。

如果你了解MQ，那么可以把streams当做基于内存的MQ。如果你还了解kafka，那么甚至可以把streams当做基于内存的kafka。listpack存储信息，Rax组织listpack 消息链表

listpack是对ziplist的改进，它比ziplist少了一个定位最后一个元素的属性

另外，这个功能有点类似于redis以前的Pub/Sub，但是也有基本的不同：

- streams支持多个客户端（消费者）等待数据（Linux环境开多个窗口执行XREAD即可模拟），并且每个客户端得到的是完全相同的数据。
- Pub/Sub是发送忘记的方式，并且不存储任何数据；**而streams模式下，所有消息被无限期追加在streams中，除非用于显式执行删除（XDEL）**。 XDEL 只做一个标记位 其实信息和长度还在
- streams的Consumer Groups也是Pub/Sub无法实现的控制方式。

streams数据结构

它主要有消息、生产者、消费者、消费组4组成

streams数据结构本身非常简单，但是streams依然是Redis到目前为止最复杂的类型，其原因是实现的一些额外的功能：一系列的阻塞操作允许消费者等待生产者加入到streams的新数据。另外还有一个称为Consumer Groups的概念，Consumer Group概念最先由kafka提出，Redis有一个类似实现，和kafka的Consumer Groups的目的是一样的：允许一组客户端协调消费相同的信息流！

发布消息

```
127.0.0.1:6379> xadd mystream * message apple
"1589994652300-0"
127.0.0.1:6379> xadd mystream * message orange
"1589994679942-0"
```

读取消息

```
127.0.0.1:6379> xrange mystream - +
1) 1) "1589994652300-0"
   2) 1) "message"
      2) "apple"
2) 1) "1589994679942-0"
   2) 1) "message"
      2) "orange"
```

阻塞读取

```
xread block 0 streams mystream $
```

发布新消息

```
127.0.0.1:6379> xadd mystream * message strawberry
```

创建消费组

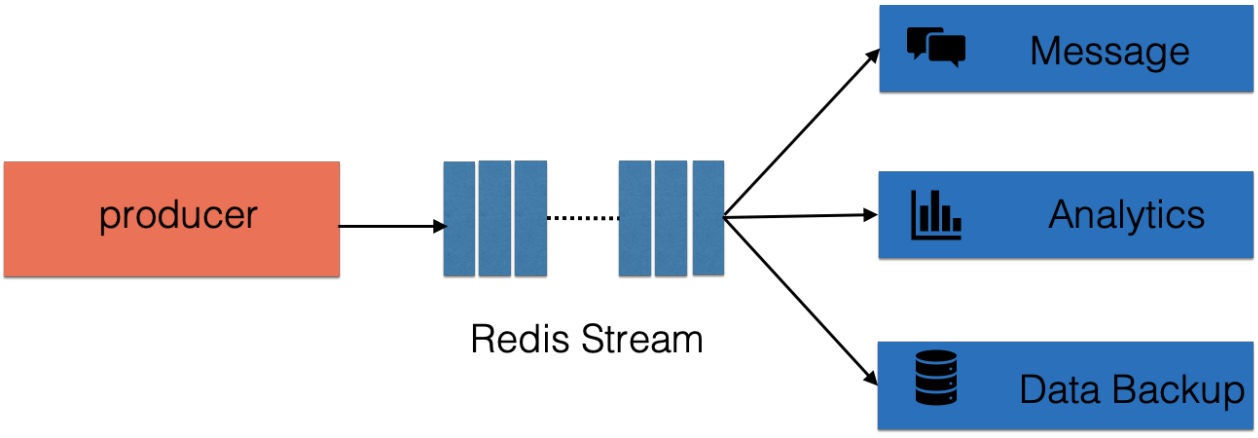
```
127.0.0.1:6379> xgroup create mystream mygroup1 0
OK
127.0.0.1:6379> xgroup create mystream mygroup2 0
OK
```

通过消费组读取消息

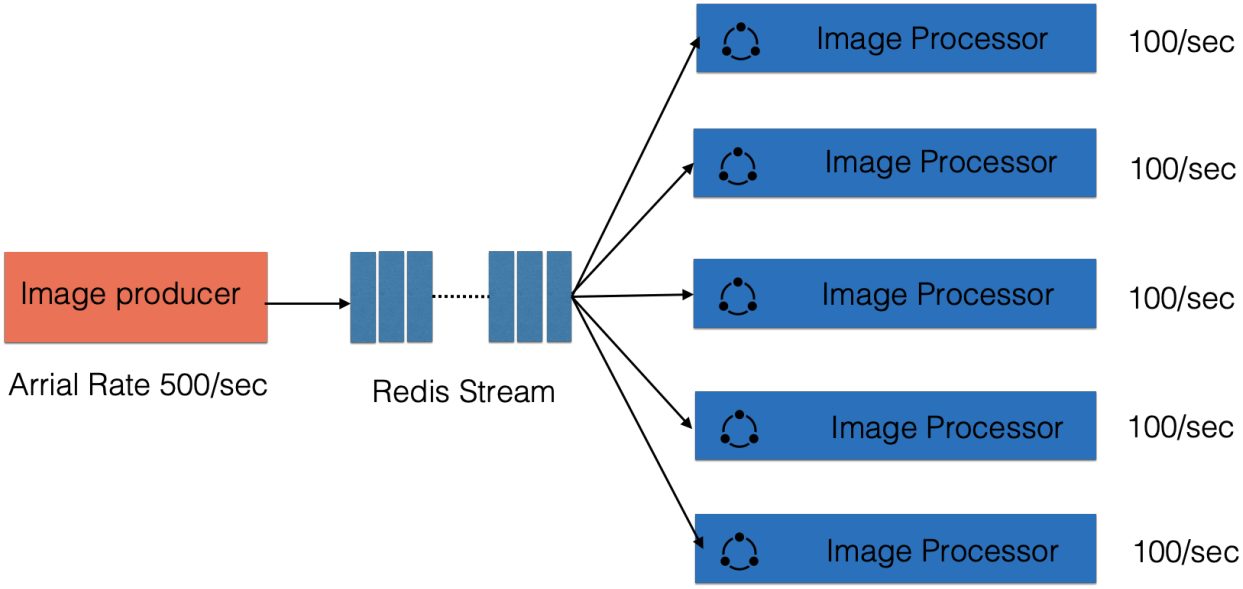
```
127.0.0.1:6379> xreadgroup group mygroup1 zange count 2 streams mystream >
1) 1) "mystream"
   2) 1) 1) "1589994652300-0"
         2) 1) "message"
            2) "apple"
      2) 1) "1589994679942-0"
         2) 1) "message"
            2) "orange"
127.0.0.1:6379> xreadgroup group mugroup1 tuge count 2 streams mystream >
1) 1) "mystream"
   2) 1) 1) "1589995171242-0"
         2) 1) "message"
            2) "strawberry"

127.0.0.1:6379> xreadgroup group mugroup2 tuge count 1 streams mystream >
1) 1) "mystream"
   2) 1) 1) "1589995171242-0"
         2) 1) "message"
            2) "apple"
```

Redis Stream使用场景

可用作时通信等，大数据分析，异地数据备份等



客户端可以平滑扩展，提高处理能力

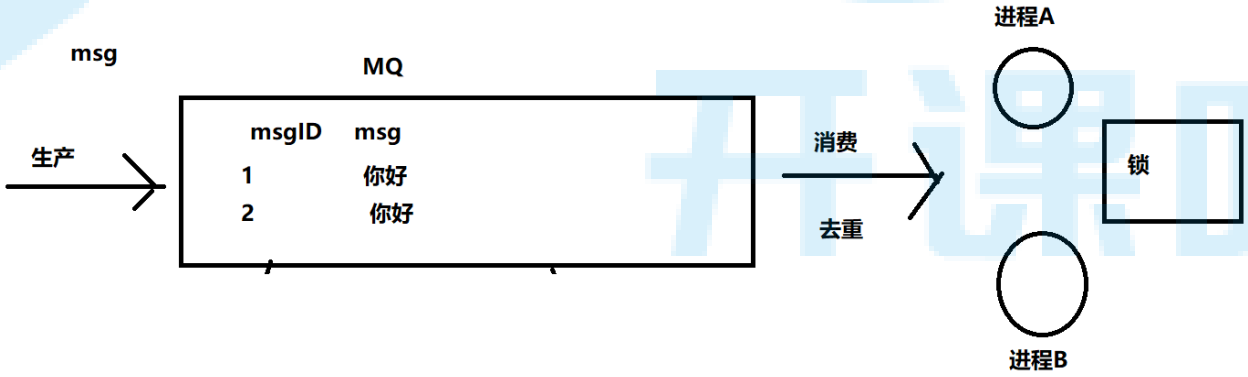


Redis实现分布式锁

业务场景

1、防止用户重复下单

2、MQ消息去重



3、订单操作变更

4、库存超卖

。。。。

分析：

业务场景共性：

共享资源

用户id、订单id、商品id。。。

解决方案

共享资源互斥

共享资源串行化

问题转化

锁的问题 （将需求抽象后得到问题的本质）

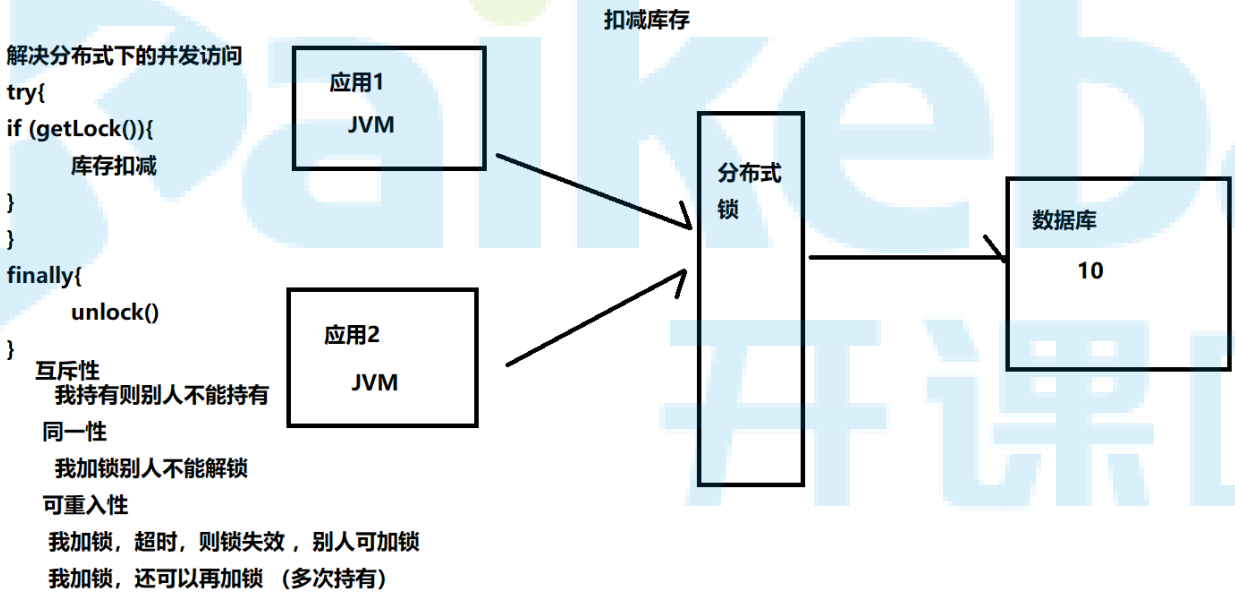
锁的处理

单应用中使用锁：（单进程多线程）

synchronized、ReentrantLock

分布式应用中使用锁：（多进程多线程）

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。



Redis实现分布式锁

原理

利用Redis的单线程特性对共享资源进行串行化处理

实现方式

获取锁

方式1（**使用set命令实现**）--推荐

```
/**
   * 使用redis的set命令实现获取分布式锁
   * @param lockKey      可以就是锁
   * @param requestId    请求ID，保证同一性     uuid+threadID
   * @param expireTime   过期时间，避免死锁
   * @return
   */
  public  boolean getLock(String lockKey,String requestId,int expireTime) {
    //NX:保证互斥性
        // hset    原子性操作
    String result = jedis.set(lockKey, requestId, "NX", "EX", expireTime);
    if("OK".equals(result)) {
      return true;
    }

    return false;
  }
```

方式2（**使用setnx命令实现**） -- 并发会产生问题

```
   public  boolean getLock(String lockKey,String requestId,int expireTime) {
    Long result = jedis.setnx(lockKey, requestId);
    if(result == 1) {
      //成功设置 失效时间
      jedis.expire(lockKey, expireTime);
      return true;
    }

    return false;
  }
```

释放锁

方式1（del命令实现） -- 并发

```
/**
 * 释放分布式锁
 * @param lockKey
 * @param requestId

 */
public static void releaseLock(String lockKey,String requestId) {

    if (requestId.equals(jedis.get(lockKey))) {
        jedis.del(lockKey);
    }
}
```

问题在于如果调用jedis.del()方法的时候，这把锁已经不属于当前客户端的时候会解除他人加的锁。那么是否真的有这种场景？答案是肯定的，比如客户端A加锁，一段时间之后客户端A解锁，在执行jedis.del()之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

方式2（**redis+lua脚本实现**）--推荐

```
public static boolean releaseLock(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 (result.equals(1L)) {
      return true;
    }
    return false;
}
```

常见缓存问题

数据读

缓存穿透

查询缓存中永远不存在的一个值。因为这个值数据库没有。

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。

也就是说，对不存在的key进行高并发访问，导致数据库压力瞬间增大，这就叫做【缓存穿透】。

解决方案：

1、对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。

2、布隆过滤器（位运算） set s1 1111

S1 00000001

缓存雪崩

> 大量缓存集中失效。
> 大量数据访问压力一下怼到数据库身上

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

突然间大量的key失效了或redis重启，大量访问数据库

解决方案:

1、 key的失效期分散开 不同的key设置不同的有效期

2、设置二级缓存

3、高可用

缓存击穿

> 单个缓存有效期到了，失效了。
> 而且在这个时间点有大量的请求进来（导致这些请求都访问到数据库）。

对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

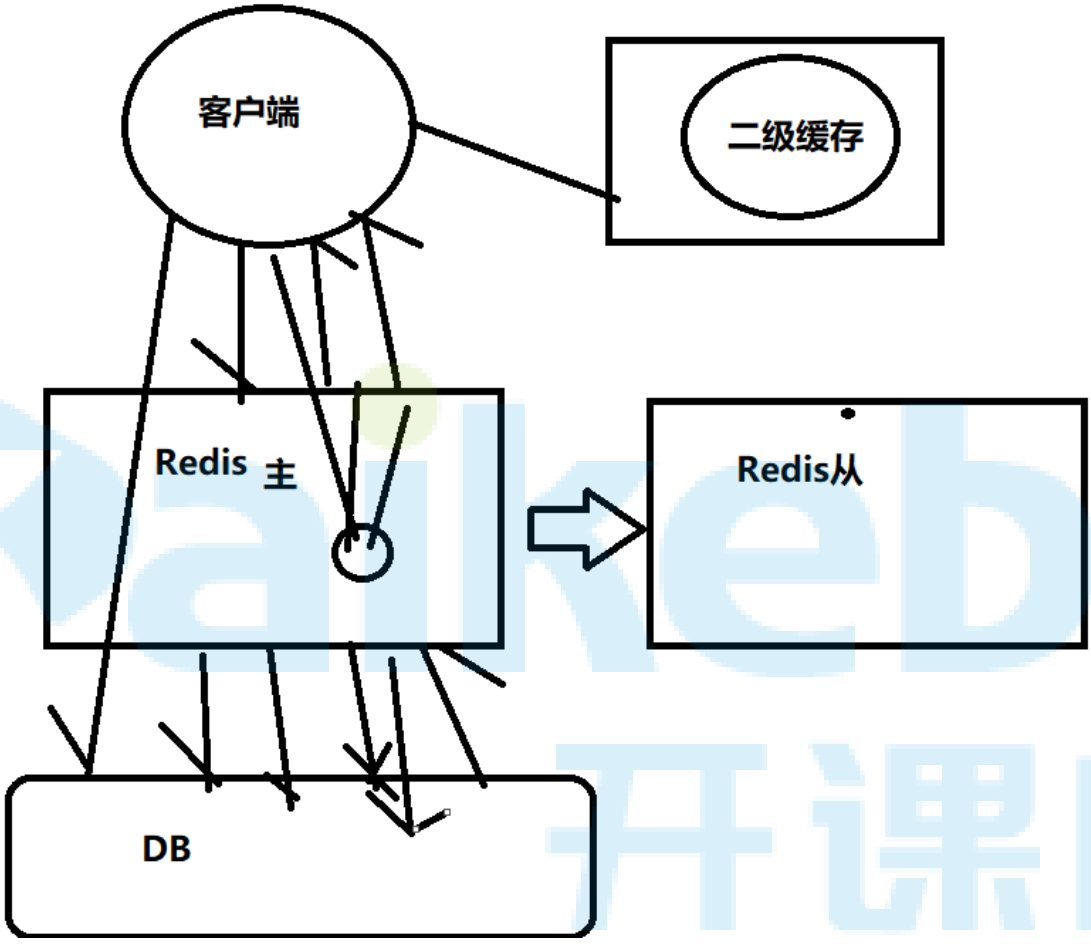
解决方案：

用分布式锁控制访问的线程

使用redis的setnx互斥锁先进行判断，这样其他线程就处于等待状态，保证不会有大并发操作去操作数据库。

```
if(redis.sexnx()==1){
  //先查询缓存
  //查询数据库
  //加入缓存
}
```

不设超时时间，写一致问题



数据写

数据不一致的根源： 数据源不一样

如何解决

强一致性很难，追求最终一致性

互联网业务数据处理的特点

高吞吐量

低延迟

数据敏感性低于金融业

时序控制是否可行?

先更新数据库再更新缓存或者先更新缓存再更新数据库

本质上不是一个原子操作，所以时序控制不可行

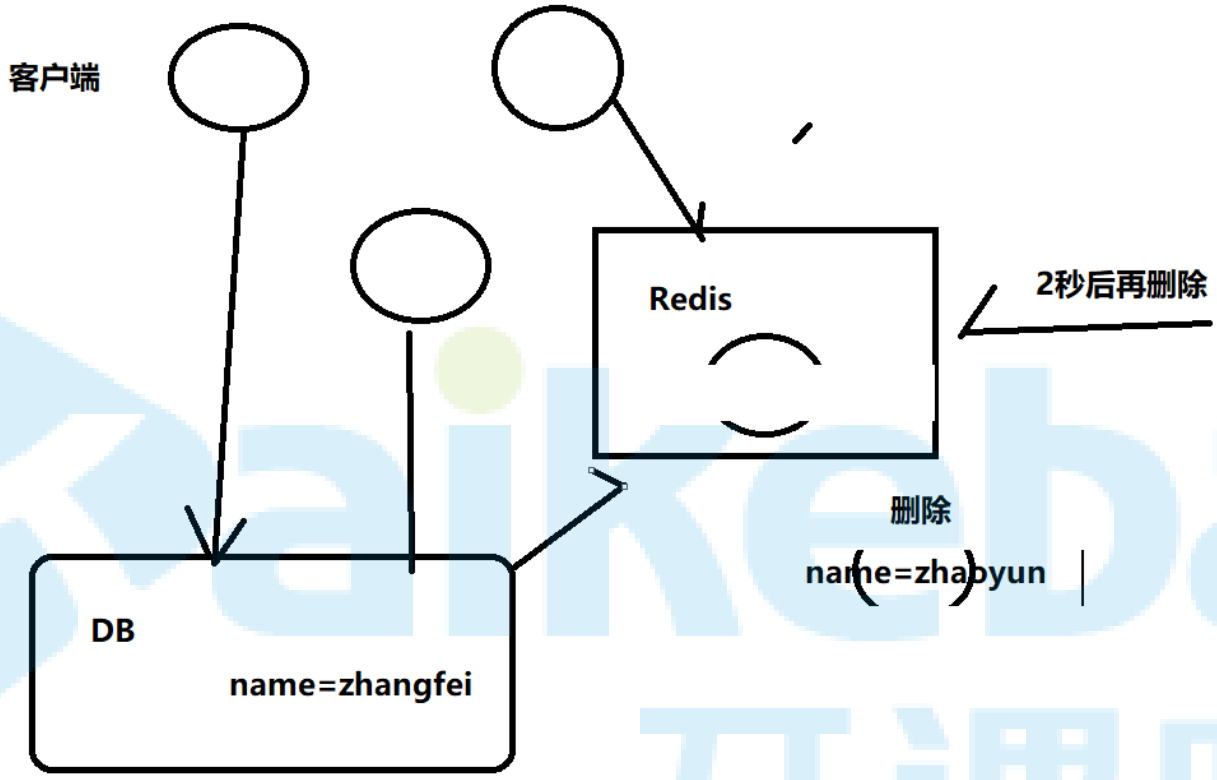
保证数据的最终一致性(延时双删)

1、先更新数据库同时删除缓存项(key)，等读的时候再填充缓存

2、2秒后再删除一次缓存项(key)

3、设置缓存过期时间 Expired Time 比如 10秒 或1小时

4、将缓存删除失败记录到日志中，利用脚本提取失败记录再次删除（缓存失效期过长 7*24）



升级方案

通过数据库的binlog来异步淘汰key，利用工具(canal)将binlog日志采集发送到MQ中，然后通过ACK机制确认处理删除缓存。

二 Redis 的补充数据类型

BitMap

BitMap 就是通过一个 bit 位来表示某个元素对应的值或者状态, 其中的 key 就是对应元素本身，实际上底层也是通过对字符串的操作来实现。Redis 从 2.2 版本之后新增了setbit, getbit, bitcount 等几个 bitmap 相关命令。虽然是新命令，但是本身都是对字符串的操作，我们先来看看语法：

```
SETBIT key offset value
```

其中 offset 必须是数字，value 只能是 0 或者 1

bitop add

bitop not

存储一亿用户 12.5M

```
127.0.0.1:6379> setbit k1 5 1
(integer) 0
127.0.0.1:6379> getbit k1 5 1
(integer) 1
127.0.0.1:6379> getbit k1 4 0
(integer) 0
127.0.0.1:6379> bitcount k1
(integer) 1
127.0.0.1:6379> setbit k1 3 1
(integer) 0
127.0.0.1:6379> bitcount k1
(integer) 2
127.0.0.1:6379> setbit "200522:active"  67 1
(integer) 0
127.0.0.1:6379> setbit "200522:active" 78 1
(integer) 0
```

其中 offset 必须是数字，value 只能是 0 或者 1

通过 bitcount可以很快速的统计，比传统的关系型数据库效率高很多

1、比如统计年活跃用户数量

用户的ID作为offset，当用户在一年内访问过网站，就将对应offset的bit值设置为“1”；

通过bitcount 来统计一年内访问过网站的用户数量

2、比如统计三天内活跃用户数量

时间字符串作为key，比如 “200522:active“ ；

用户的ID就可以作为offset，当用户访问过网站，就将对应offset的bit值设置为“1”；

统计三天的活跃用户，通过bitop or 获取一周内访问过的用户数量

3、连续三天访问的用户数量 bitop and

4、三天内没有访问的用户数量 bitop not

5、统计在线人数 设置在线key：“online：active”，当用户登录时，通过setbit设置

bitmap的优势，以统计活跃用户为例

每个用户id占用空间为1bit，消耗内存非常少，存储1亿用户量只需要12.5M

bitmap 可以做布隆过滤器 确认访问值是否存在 只要在布隆过滤器里值是0 后面的服务就别访问了

HyperLogLog （2.8）

1.基于bitmap 计数

2. 基于概率基数计数 0.87

这个数据结构的命令有三个：PFADD、PFCOUNT、PFMERGE

内部编码主要分稀疏型和密集型

用途：记录网站IP注册数，每日访问的IP数，页面实时UV、在线用户人数

局限性：只能统计数量，没有办法看具体信息

```
127.0.0.1:6379> pfadd h1 b
(integer) 1
127.0.0.1:6379> pfadd h1 a
(integer) 0
127.0.0.1:6379> pfcount h1
(integer) 2
127.0.0.1:6379> pfadd h1 c
(integer) 1
127.0.0.1:6379> pfadd h2 a
(integer) 1
127.0.0.1:6379> pfadd h3 d
(integer) 1
127.0.0.1:6379> pfmerge h3 h1 h2
OK
127.0.0.1:6379> pfcount h3
(integer) 4
```

Geospatial （3.2）

底层数据结构 Zset GEOADD GEODIST GEOHASH GEOPOP GEOPADUIS GEORADIUSBYMEMBER

可以用来保存地理位置，并作位置距离计算或者根据半径计算位置等。有没有想过用Redis来实现附近的人？或者计算最优地图路径？Geo本身不是一种数据结构，它**本质上还是借助于Sorted Set（ZSET）**

GEOADD key 经度 维度 名称

把某个具体的位置信息（经度，纬度，名称）添加到指定的key中，数据将会用一个sorted set存储，以便稍后能使用GEORADIUS和GEORADIUSBYMEMBER命令来根据半径来查询位置信息。

```
127.0.0.1:6379> GEOADD cities 116.404269 39.91582 "beijing" 121.478799
31.235456 "shanghai"
(integer) 2
127.0.0.1:6379> ZRANGE cities 0 -1
1) "shanghai"
2) "beijing"
127.0.0.1:6379> ZRANGE cities 0 -1 WITHSCORES
1) "shanghai"
2) "4054803475356102"
3) "beijing"
4) "4069885555377153"
```

```
127.0.0.1:6379> GEODIST cities beijing shanghai km
"1068.5677"
127.0.0.1:6379> GEOPOS cities beijing shanghai
1) 1) "116.40426903963088989"
   2) "39.91581928642635546"
2) 1) "121.47879928350448608"
   2) "31.23545629441388627"
127.0.0.1:6379> GEOADD cities 120.165036 30.278973 hangzhou
(integer) 1
127.0.0.1:6379> GEORADIUS cities 120 30 500 km
1) "hangzhou"
2) "shanghai"
127.0.0.1:6379> GEORADIUSBYMEMBER cities shanghai 200 km
1) "hangzhou"
2) "shanghai"
127.0.0.1:6379> ZRANGE cities 0 -1
1) "hangzhou"
2) "shanghai"
3) "beijing"
127.0.0.1:6379>
```

一 Redis 性能调优

1. 设计上做优化 设计合理的键值 值 的长度
2. 选择合适的持久化方式 4.0 后 选择混合持久化 只是做缓存，不需要设置持久化

1.1 设置键值的过期时间（惰性删除）

我们应该根据实际的业务情况，对键值设置合理的过期时间，这样 Redis 会帮你自动清除过期的键值对，以节约对内存的占用，以避免键值过多的堆积，频繁的触发内存淘汰策略。

Redis 有四个不同的命令可以用于设置键的生存时间(键可以存在多久)或过期时间(键什么时候会被删除)：

- EXPIRE 命令用于将键key 的生存时间设置为ttl 秒。
- PEXPIRE 命令用于将键key 的生存时间设置为ttl 毫秒。
- EXPIREAT < timestamp> 命令用于将键key 的过期时间设置为timestamp所指定的秒数时间戳。PEXPIREAT < timestamp > 命令用于将键key 的过期时间设置为timestamp所指定的毫秒数时间戳。

```
127.0.0.1:6379> set key1 'value1'
OK
127.0.0.1:6379> expire key1 20
(integer) 1
127.0.0.1:6379> get key1 "value1" 127.0.0.1:6379> get key1 "value1"
127.0.0.1:6379> get key1 (nil) 127.0.0.1:6379>
```

1.2 使用 lazy free 特性

lazy free 特性是 Redis 4.0 新增的一个非常使用的功能，它可以理解为惰性删除或延迟删除。意思是在删除的时候提供异步延时释放键值的功能，把键值释放操作放在 BIO(Background I/O) 单独的子线程处理中，以减少删除删除对 Redis 主线程的阻塞，可以有效地避免删除 big key 时带来的性能和可用性问题。

lazy free 对应了 4 种场景，默认都是关闭的：

```
lazyfree-lazy-eviction no
lazyfree-lazy-expire no
lazyfree-lazy-server-del no
slave-lazy-flush no
```

它们代表的含义如下：

- lazyfree-lazy-eviction：表示当 Redis 运行内存超过最大内存时，是否开启 lazy free 机制删除；
- lazyfree-lazy-expire：表示设置了过期时间的键值，当过期之后是否开启 lazy free 机制删除；
- lazyfree-lazy-server-del：有些指令在处理已存在的键时，会带有一个隐式的 del 键的操作，比如 rename 命令，当目标键已存在，Redis 会先删除目标键，如果这些目标键是一个 big key，就会造成阻塞删除的问题，此配置表示在这种场景中是否开启 lazy free 机制删除；
- slave-lazy-flush：针对 slave(从节点) 进行全量数据同步，slave 在加载 master 的 RDB 文件前，会运行 flushall 来清理自己的数据，它表示此时是否开启 lazy free 机制删除。

建议开启其中的 lazyfree-lazy-eviction、lazyfree-lazy-expire、lazyfree-lazy-server-del 等配置，这样就可以有效的提高主线程的执行效率。

1.3 限制 Redis 内存大小，设置内存淘汰策略

最大缓存

maxmemory 1048576

maxmemory 1048576B

maxmemory 1000KB

maxmemory 100MB

maxmemory 1GB

maxmemory 1000K

maxmemory 100M

maxmemory 1G

没有指定最大缓存，如果有新的数据添加，超过最大内存，则32位会使redis崩溃，所以一定要设置。最佳设置是物理内存的75% ，写操作比较多　60%

- 在 64 位操作系统中 Redis 的内存大小是没有限制的，也就是配置项 `maxmemory` 是被注释掉的，这样就会导致在物理内存不足时，使用 swap 空间既交换空间，而当操作系统将 Redis 所用的内存分页移至 swap 空间时，将会阻塞 Redis 进程，导致 Redis 出现延迟，从而影响 Redis 的整体性能。因此我们需要限制 Redis 的内存大小为一个固定的值，当 Redis 的运行到达此值时会触发内存

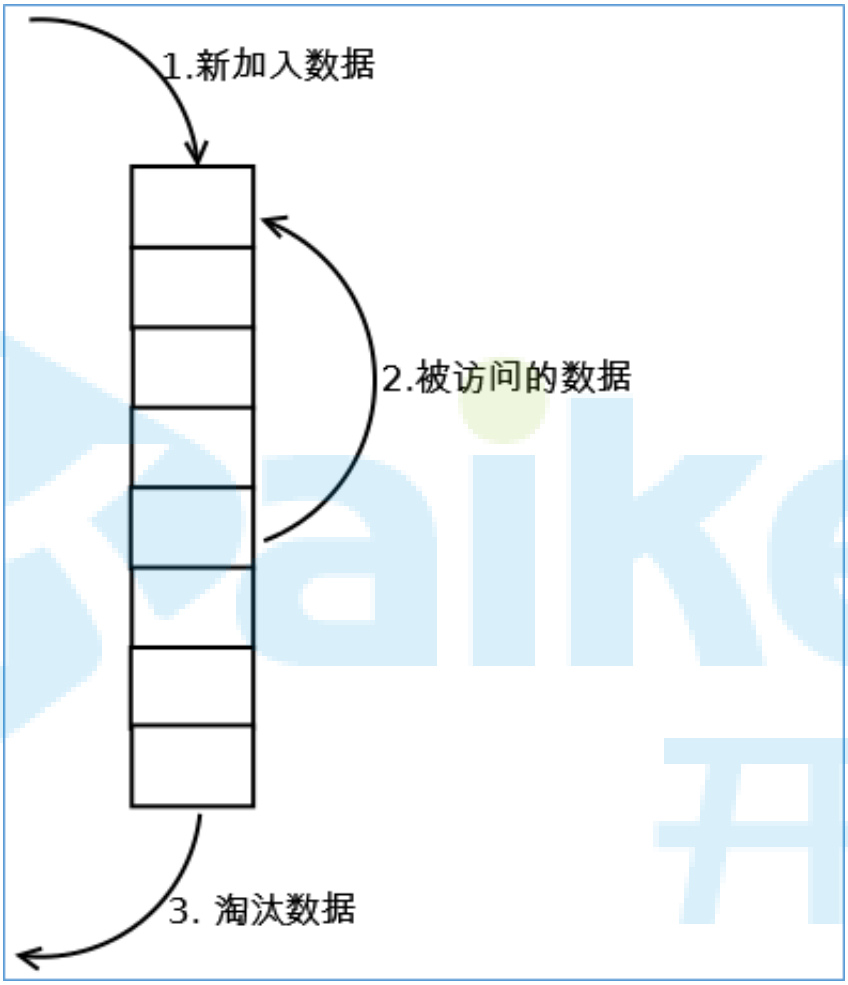
淘汰策略，**内存淘汰策略在 Redis 4.0 之后有 8 种**

FIFO,LRU和LFU

LRU原理

LRU（Least recently used，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：

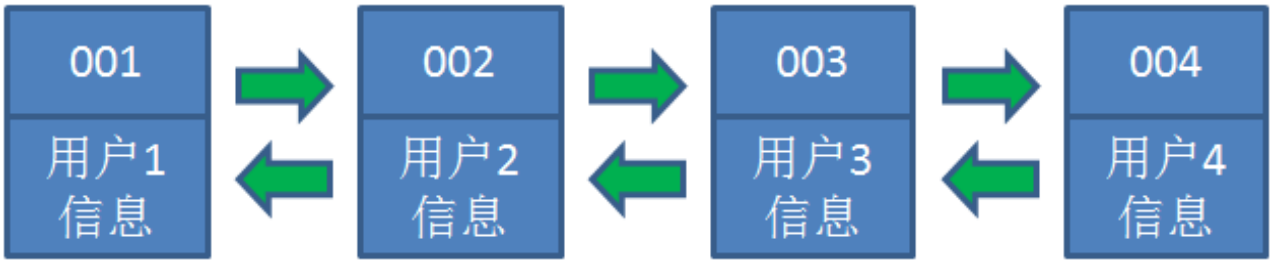


1. 新数据插入到链表头部；
2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

案例分析

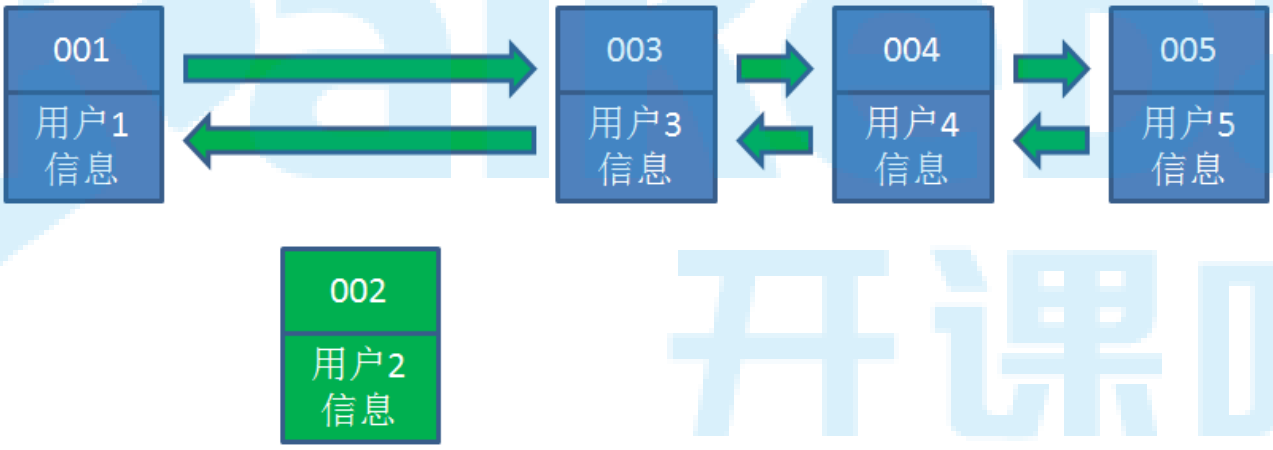
> 让我们以用户信息的需求为例，来演示一下LRU算法的基本思路：map 只是设置了5个

1.假设我们使用哈希链表来缓存用户信息，目前缓存了4个用户，这4个用户是按照时间顺序依次从链表右端插入的。

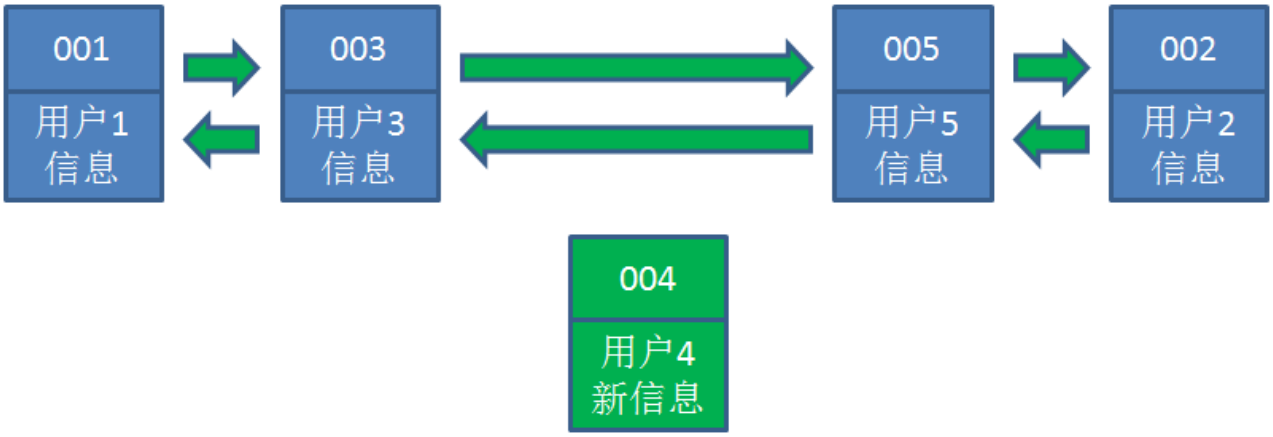


2.此时，业务方访问用户5，由于哈希链表中没有用户5的数据，我们从数据库中读取出来，插入到缓存当中。这时候，链表中最右端是最新访问到的用户5，最左端是最近最少访问的用户1。

3.接下来，业务方访问用户2，哈希链表中存在用户2的数据，我们怎么做呢？我们把用户2从它的前驱节点和后继节点之间移除，重新插入到链表最右端。这时候，链表中最右端变成了最新访问到的用户2，最左端仍然是最近最少访问的用户1。



4.接下来，业务方请求修改用户4的信息。同样道理，我们把用户4从原来的位置移动到链表最右侧，并把用户信息的值更新。这时候，链表中最右端是最新访问到的用户4，最左端仍然是最近最少访问的用户1。



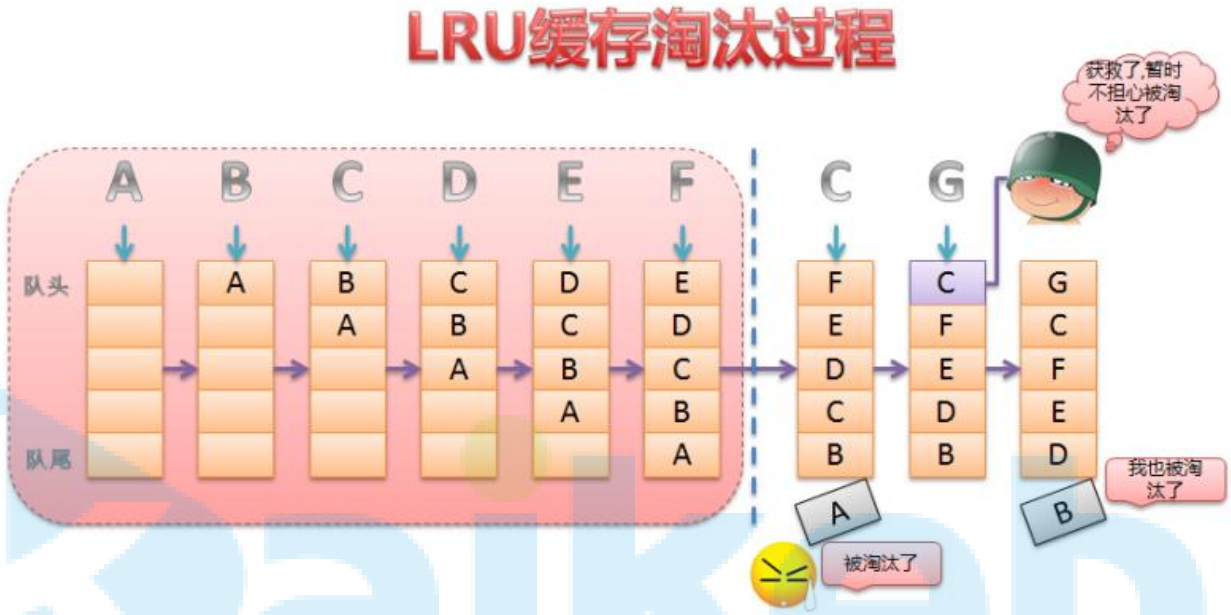
5.后来业务方换口味了，访问用户6，用户6在缓存里没有，需要插入到哈希链表。假设这时候缓存容量已经达到上限，必须先删除最近最少访问的数据，那么位于哈希链表最左端的用户1就会被删除掉，然后再把用户6插入到最右端。

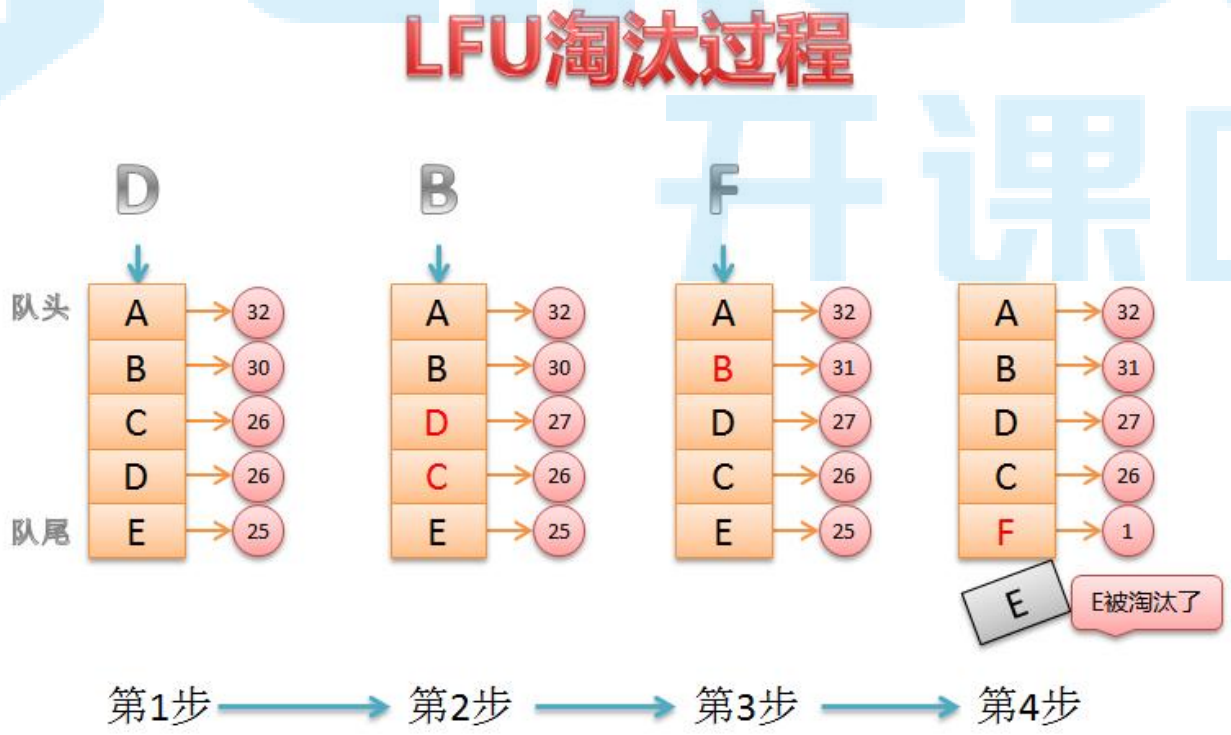
以上，就是LRU算法的基本思路。

https://www.itcodemonkey.com/article/11153.html

LFU原理

LFU,全称是:Least Frequently Used，最不经常使用策略,在一段时间内,数据被使用频次最少的,优先被淘汰。最少使用**（**LFU**）是一种用于管理计算机内存的缓存算法。主要是记录和追踪内存块的使用次数,当缓存已满并且需要更多空间时，系统将以最低内存块使用频率清除内存.采用LFU算法的最简单方法是为每个加载到缓存的块分配一个计数器。每次引用该块时，计数器将增加一。当缓存达到容量并有一个新的内存块等待插入时，系统将搜索计数器最低的块并将其从缓存中删除(本段摘自维基百科)**





LRU和LFU侧重点不同，LRU主要体现在对元素的使用时间上,而LFU主要体现在对元素的使用频次上。LFU的缺陷是：在短期的时间内，对某些缓存的访问频次很高，这些缓存会立刻晋升为热点数据，而保证不会淘汰，这样会驻留在系统内存里面。而实际上，这部分数据只是短暂的高频率访问，之后将会长期不访问,瞬时的高频访问将会造成这部分数据的引用频率加快，而一些新加入的缓存很容易被快速删除，因为它们的引用频率很低。

Redis缓存淘汰策略

redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

> maxmemory-policy voltile-lru，支持热配置 内存淘汰策略在 Redis 4.0 之后有 8 种：

1. **noeviction**：不淘汰任何数据，当内存不足时，新增操作会报错，Redis 默认内存淘汰策略；
2. **allkeys-lru**：淘汰整个键值中最久未使用的键值；
3. **allkeys-random**：随机淘汰任意键值;
4. **volatile-lru**：淘汰所有设置了过期时间的键值中最久未使用的键值；
5. **volatile-random**：随机淘汰设置了过期时间的任意键值；
6. **volatile-ttl**：优先淘汰更早过期的键值。

在 Redis 4.0 版本中又新增了 2 种淘汰策略：

1. **volatile-lfu**：淘汰所有设置了过期时间的键值中，最少使用的键值；
2. **allkeys-lfu**：淘汰整个键值中最少使用的键值。

其中 allkeys-xxx 表示从所有的键值中淘汰数据，而 volatile-xxx 表示从设置了过期键的键值中淘汰数据。

我们可以根据实际的业务情况进行设置，默认的淘汰策略不淘汰任何数据，在新增时会报错。

1.4 禁用长耗时的查询命令

Redis 绝大多数读写命令的时间复杂度都在 O(1) 到 O(N) 之间，在官方文档对每命令都有时间复杂度说明，如下图

其中 O(1) 表示可以安全使用的，而 O(N) 就应该当心了，N 表示不确定，数据越大查询的速度可能会越慢。因为 Redis 只用一个线程来做数据查询，如果这些指令耗时很长，就会阻塞 Redis，造成大量延时。

要避免 O(N) 命令对 Redis 造成的影响，可以从以下几个方面入手改造：

- 决定禁止使用 keys 命令；
- 避免一次查询所有的成员，要使用 scan 命令进行分批的，游标式的遍历；
- 通过机制严格控制 Hash、Set、Sorted Set 等结构的数据大小；
- 将排序、并集、交集等操作放在客户端执行，以减少 Redis 服务器运行压力；
- 删除 (del) 一个大数据的时候，可能会需要很长时间，所以建议用异步删除的方式 **unlink**，它会启动一个新的线程来删除目标数据，而不阻塞 Redis 的主线程。

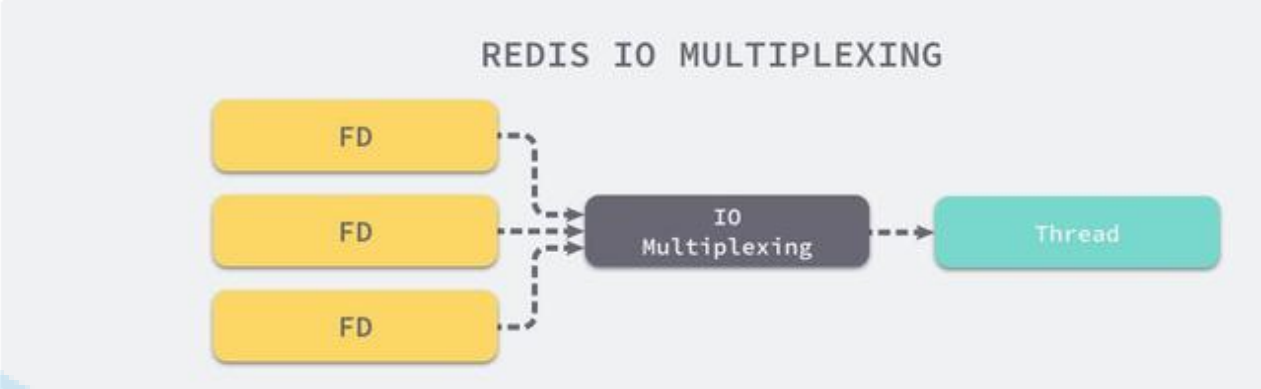
Redis6.0 引入了多线程

主流使用 2.8 3.2 4.0 5.0

1. 为什么 Redis 一开始选择单线程模型（单线程的好处）？

（1）IO多路复用

Redis顶层设计



FD是一个文件描述符，意思是表示当前文件处于可读、可写还是异常状态。使用 I/O 多路复用机制同时监听多个文件描述符的可读和可写状态。**你可以理解为具有了多线程的特点。**

一旦受到网络请求就会在内存中快速处理，由于绝大多数的操作都是纯内存的，所以处理的速度会非常地快。**也就是说在单线程模式下，即使连接的网络处理很多，因为有IO多路复用，依然可以在高速的内存处理中得到忽略。**

（2）可维护性高

多线程模型虽然在某些方面表现优异，但是它却引入了程序执行顺序的不确定性，带来了并发读写的一系列问题。单线程模式下，可以方便地进行调试和测试。

（3）基于内存，单线程状态下效率依然高 读请求11W+ 写请求8.7W+

多线程能够充分利用CPU的资源，但对于Redis来说，由于基于内存速度那是相当的高，能达到在一秒内处理10万个用户请求，如果一秒十万还不能满足，那我们就可以使用Redis分片的技术来交给不同的Redis服务器。这样的做法避免了在同一个 Redis 服务中引入大量的多线程操作。

而且基于内存，除非是要进行AOF备份，否则基本上不会涉及任何的 I/O 操作。这些数据的读写由于只发生在内存中，所以处理速度是非常快的；用多线程模型处理全部的外部请求可能不是一个好的方案。

总结成两句话，基于内存而且使用多路复用技术，单线程速度很快，又保证了多线程的特点。因为没有必要使用多线程。

2. 为什么 Redis 在 6.0 之后加入了多线程（在某些情况下，单线程出现了缺点，多线程可以解决）

因为读写网络的read/write系统调用在Redis执行期间占用了大部分CPU时间，如果把网络读写做成多线程的方式对性能会有很大提升。

Redis可以使用del命令删除一个元素，如果这个元素非常大，可能占据了几十兆或者是几百兆，那么在短时间内是不能完成的，这样一来就需要多线程的异步支持。

使用多线程删除工作可以在后台

总结：Redis 选择使用单线程模型处理客户端的请求主要还是因为 CPU 不是 Redis 服务器的瓶颈，所以使用多线程模型带来的性能提升并不能抵消它带来的开发成本和维护成本，系统的性能瓶颈也主要在网络 I/O 操作上；而 Redis 引入多线程操作也是出于性能上的考虑，对于一些大键值对的删除操作，通过多线程非阻塞地释放内存空间也能减少对 Redis 主线程阻塞的时间，提高执行的效率

4.0 开始已经有异步线程了

1.5 使用 slowlog 优化耗时命令

我们可以使用 slowlog 功能找出最耗时的 Redis 命令进行相关的优化，以提升 Redis 的运行速度，慢查询有两个重要的配置项：

- `slowlog-log-slower-than`：用于设置慢查询的评定时间，也就是说超过此配置项的命令，将会被当成慢操作记录在慢查询日志中，它执行单位是微秒 (1 秒等于 1000000 微秒)；
- `slowlog-max-len`：用来配置慢查询日志的最大记录数。

我们可以根据实际的业务情况进行相应的配置，其中慢日志是按照插入的顺序倒序存入慢查询日志中，我们可以使用 `slowlog get n` 来获取相关的慢查询日志，再找到这些慢查询对应的业务进行相关的优化。

1.6 避免大量数据同时失效

Redis 过期键值删除使用的是贪心策略，它每秒会进行 10 次过期扫描，此配置可在 redis.conf 进行配置，默认值是 `hz 10`，Redis 会随机抽取 20 个值，删除这 20 个键中过期的键，如果过期 key 的比例超过 25% ，重复执行此流程，如下图所示：

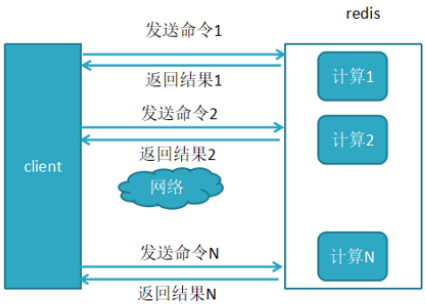
如果在大型系统中有大量缓存在同一时间同时过期，那么会导致 Redis 循环多次持续扫描删除过期字典，直到过期字典中过期键值被删除的比较稀疏为止，而在整个执行过程会导致 Redis 的读写出现明显的卡顿，卡顿的另一种原因是内存管理器需要频繁回收内存页，因此也会消耗一定的 CPU。

为了避免这种卡顿现象的产生，我们需要预防大量的缓存在同一时刻一起过期，就简单的解决方案就是在过期时间的基础上添加一个指定范围的随机数。

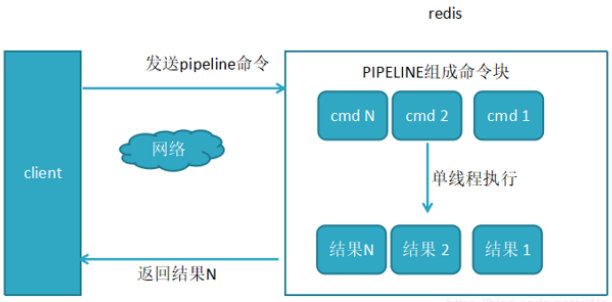
1.7 使用 Pipeline 批量操作数据

Pipeline (管道技术) 是客户端提供的一种批处理技术

可以批量执行一组指令，一次性返回全部结果，
可以减少频繁的请求应答



未使用pipeline执行N条命令



使用了pipeline执行N条命令

1.8 客户端使用优化

在客户端的使用上我们除了要尽量使用 Pipeline 的技术外，还需要注意要尽量使用 Redis 连接池，而不是频繁创建销毁 Redis 连接，这样就可以减少网络传输次数和减少了非必要调用指令。

```
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;
```

1.9 使用分布式架构来增加读写速度

Redis 分布式架构有重要的手段：

- 主从同步
- 哨兵模式
- Redis Cluster 集群

使用主从同步功能我们可以把写入放到主库上执行，把读功能转移到从服务上，因此就可以在单位时间内处理更多的请求，从而提升的 Redis 整体的运行速度。

而哨兵模式是对于主从功能的升级，但当主节点奔溃之后，无需人工干预就能自动恢复 Redis 的正常使用。

Redis Cluster 是 Redis 3.0 正式推出的，Redis 集群是通过将数据库分散存储到多个节点上来平衡各个节点的负载压力。

Redis Cluster 采用虚拟哈希槽分区，所有的键根据哈希函数映射到 0 ~ 16383 整数槽内，计算公式：slot = CRC16(key) & 16383，每一个节点负责维护一部分槽以及槽所映射的键值数据。这样 Redis 就可以把读写压力从一台服务器，分散给多台服务器了，因此性能会有很大的提升。

Redis Cluster 应该是首选的实现方案，它可以把读写压力自动的分担给更多的服务器，并且拥有自动容灾的能力。

数据量不多，并发也不多 我们就用单机

数据量不是很多，并发很多（超过10万的读） 我们一般是使用主从+sentinel 高可用 写是主库，读是从库 每增加10万并发读 加一个从机

数据量大， 我们都是采用cluster 多读写。10个 4G内存 肯定比一个40G 便宜

cluster 把sentinel的功能分摊到主从节点了 监控：master ，主节点是否fail 是主节点选择 选择新的主节点 是从节点决定

强调一点：cluster 的从机 默认是不读不写，就是备份和容灾

1.10 使用物理机而非虚拟机

在虚拟机中运行 Redis 服务器，因为和物理机共享一个物理网口，并且一台物理机可能有多个虚拟机在运行，因此在内存占用上和网络延迟方面都会有很糟糕的表现，我们可以通过 `./redis-cli --intrinsic-latency 100` 命令查看延迟时间，如果对 Redis 的性能有较高要求的话，应尽可能在物理机上直接部署 Redis 服务器。

云服务器：选择真正的云服务器 ，腾讯云-云其实做的虚机处理 2018年的时候

1.11 禁用 THP 特性

Linux kernel 在 2.6.38 内核增加了 Transparent Huge Pages (THP) 特性 ，支持大内存页 2MB 分配，默认开启。

nginx

服务器是数据服务器 redis oracle MongoDB

当开启了 THP 时，fork 的速度会变慢，fork 之后每个内存页从原来 4KB 变为 2MB，会大幅增加重写期间父进程内存消耗。同时每次写命令引起的复制内存页单位放大了 512 倍，会拖慢写操作的执行时间，导致大量写操作慢查询。例如简单的 incr 命令也会出现在慢查询中，因此 Redis 建议将此特性进行禁用，禁用方法如下：

> echo never > /sys/kernel/mm/transparent_hugepage/enabled

为了使机器重启后 THP 配置依然生效，可以在 /etc/rc.local 中追加 `echo never > /sys/kernel/mm/transparent_hugepage/enabled`。

二 缓存常见问题 -不单指Redis

2.1 缓存预热

1. 直接写个缓存刷新页面，上线前手工操作一下
2. 数据量不大的时候，可以在项目启动的时候自动加载
3. 定时刷新缓存

2.2 缓存雪崩

- 什么叫缓存雪崩?

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

- 如何解决?

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。
setRedis（Key，value，time + Math.random() * 10000） 代码

3：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期（此点为补充）

多级缓存 头条的时候 文章-读 热点文章

CDN 和 Nginx 启动THP 拒敌于国门之外

2.3 缓存击穿

- 什么叫缓存击穿?

对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

- 如何解决?

使用redis的setnx互斥锁先进行判断，这样其他线程就处于等待状态，保证不会有大并发操作去操作数据库。

```
if(redis.sexnx()==1){
    //先查询缓存
    //查询数据库
    //加入缓存
}
```

设置监控 当某个热点是超级热点 一般设置为持久化

2.4 缓存穿透

- 什么叫缓存穿透?

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。

也就是说，对不存在的key进行高并发访问，导致数据库压力瞬间增大，这就叫做【缓存穿透】。

- 如何解决?

1.在服务器端，接收参数时业务接口中过滤不合法的值，null，负值，和空值进行检测和空值。

2.bloom filter：类似于哈希表的一种算法，用所有可能的查询条件生成一个bitmap，在进行数据库查询之前会使用这个bitmap进行过滤，如果不在其中则直接过滤，从而减轻数据库层面的压力。
采用的是一票否决　只要有一个认为你不存在　就认为你是不存在的

3.空值缓存：一种比较简单的解决办法，在第一次查询完不存在的数据后，将该key与对应的空值也放入缓存中，只不过设定为较短的失效时间，例如几分钟，这样则可以应对短时间的大量的该key攻击，设置为较短的失效时间是因为该值可能业务无关，存在意义不大，且该次的查询也未必是攻击者发起，无过久存储的必要，故可以早点失效

缓存降级的使用 我们在缓存中没有数据的时候，会给一个随机值。 水贴，随贴。

2.5 缓存降级

当访问量出现剧增、服务出现问题（相应时间慢或者不响应） 或非核心业务影响到核心流程的性能，还需要保证服务的可用性，即便有损服务。

双11的时候 一般会对 退款 降级。

方式： 系统根据一些关键数据进行降级

配置开关实现人工降级

有些服务时无法降级（加入购物车，结算）

参考日志级别： 一般：ex有些服务偶尔网络抖动或者服务正在上线超时，可以自定降级

警告：有些服务在一端时间内有波动（95%-100%），可以自定降级或人工降级，还有发送告警

错误：可利用率低于90%，redis连接池被打爆了，数据库连接池被打爆，或者访问量突然猛增到系统能承受的最大阈值，这时候根据情况自动降级或人工降级

严重错误：比如因为特殊原因数据错误了，需要紧急人工降级。

redis服务出问题了， 不去查数据库，而是直接返回一个默认值（自定义一些随机值）

2.6 缓存更新

自定义的缓存淘汰策略：

1. 定期去清理过期的缓存
2. 当有用户请求过来时，先判断这个请求用到的缓存是否过期，过期的话就去底层系统得到新数据进

行缓存更新

2.7 缓存数据库双写一致性

DB KV

双写 就一定会出现数据一致性问题

一般来说，在读取缓存方面，我们都是先读取缓存，再读取数据库的。

但是，在更新缓存方面，我们是需要先更新缓存，再更新数据库？还是先更新数据库，再更新缓存？还是说有其他的方案？

各种写场景与db redis一致性

1.先更新redis再更新db

按下面步骤会有问题,AB是两个线程

```
A_update_redis
B_update_redis
B_update_db
A_update_db
```

最终db是a值但是redis是b值，不一致

2.先更新db再更新redis

```
A_update_db
B_update_db
B_update_redis
A_update_redis
```

最终db是b值但是redis是a值

3.先更新DB再删除redis

```
A_update_db
B_update_db
B_rm_redis
A_rm_redis
```

是不是不明白。想不出来怎么不一致了？
不是这样的，没这么简单，第二次rm_redis就会保证后面的redis和db是一致的
实际是下面这种形式

```
A_get_data
redis_cache_miss
A_get_db
B_update_db
B_rm_redis
(此时如果拿db是b值，但是redis没有值)
A_update_redis
```

依赖于A_update_redis在B_update_db之后，极端情况
此时redis是old，db是new

4.先删除redis再更新DB

```
A_rm_redis
B_get_data
B_redis_miss
B_get_db
B_update_redis
A_update_db
```

此时redis是old值，db是new值

5.延迟双删

```
rm_redis
update_db
sleep xxx ms
rm_redis
```

这样叫做双删，最后一次sleep一段时间再rm_redis保证再次读请求回溯打到db，用最新值写redis

6.思考变种

上面的3和5情况可以直接变种，即

```
update_db
sleep xxx ms
rm_redis
```

解决了3中的极端情况（靠sleep解决），
并且减少5中第一次不必要的rm redis请求
当然，这个rm_redis还可以考虑异步化（提高吞吐）以及重试（避免异步处理失败），这里不展示

总结

从db回源到redis，需要考虑上面这些极端情况的case

适用场景

当然这些极端情况本身要求同一个key是多写的，这个根据业务需求来看是否需要，比如某些场景本身就是写少读多的

2.8 多个系统同时操作（并发）Redis带来的数据问题

这个问题我以前开发的时候遇到过，其实并发过程中确实会有这样的问题，比如下面这样的情况

系统A、B、C三个系统，分别去操作**Redis**的同一个Key，本来顺序是1，2，3是正常的，但是因为系统A网络突然抖动了一下，B，C在他前面操作了**Redis**，这样数据不就错了么。

就好比下单，支付，退款三个顺序你变了，你先退款，再下单，再支付，那流程就会失败，那数据不就乱了？你订单还没生成你却支付，退款了？明显走不通了，这在线上是很恐怖的事情。

这种情况怎么解决呢?

可以找个管家帮我们管理好数据的嘛!

某个时刻，多个系统实例都去更新某个 key。可以基于 **Zookeeper** 实现分布式锁。每个系统通过 **Zookeeper** 获取分布式锁，确保同一时间，只能有一个系统实例在操作某个 Key，别人都不允许读和写。

要写入缓存的数据，都是从 **MySQL** 里查出来的，都得写入 **MySQL** 中，写入 **MySQL** 中的时候必须保存一个时间戳，从 **MySQL** 查出来的时候，时间戳也查出来

每次要**写之前，先判断**一下当前这个 Value 的时间戳是否比缓存里的 Value 的时间戳要新。如果是的话，那么可以写，否则，就不能用旧的数据覆盖新的数据

三 Redis 常见面试问题

Memcache特点

- MC 处理请求时使用多线程异步 IO 的方式，可以合理利用 CPU 多核的优势，性能非常优秀；
- MC 功能简单，使用内存存储数据
- MC 对缓存的数据可以设置失效期，过期后的数据会被清除；
- 失效的策略采用延迟失效，就是当再次使用数据时检查是否失效；
- 当容量存满时，会对缓存中的数据进行剔除，剔除时，除了会对过期 key 进行清理，还会按 LRU 策略对数据进行剔除。

MC 有一些限制，这些限制在现在的互联网场景下很致命，成为大家选择**Redis**、**MongoDB**的重要原因：

- **key 不能超过 250 个字节**；

- **value 不能超过 1M 字节**；
- key 的最大失效时间是 30 天；
- **只支持 K-V 结构，不提供持久化和主从同步功能。**
- MC没有原生的集群，可以依靠客户端实现往集群中做分片写入数据。

Reids 特点

- 与 MC 不同的是，Redis 采用单线程模式处理请求。这样做的原因有 2 个：一个是因为采用了非阻塞的异步事件处理机制；另一个是缓存数据都是内存操作 IO 时间不会太长，单线程可以避免线程上下文切换产生的代价。
- **Redis** 支持持久化，所以 Redis 不仅仅可以用作缓存，也可以用作 NoSQL 数据库。
- 相比 MC，**Redis** 还有一个非常大的优势，就是除了 K-V 之外，还支持多种数据格式，例如 list、set、sorted set、hash 等。
- **Redis** 提供主从同步机制，以及 **Cluster** 集群部署能力，能够提供高可用服务。

MC和redis的性能对比

存储小数据时候 Redis性能是比MC性能高

100K以上,MC的性能是高于Redis。

MC本身没有集群功能，可以使用客户端做分片

历史面试问题（也是课程的回顾总结）

1. Redis 有哪些数据类型. 8-9
2. 能说一下他们的特性，还有分别的使用场景么？day 03
3. 单机会有瓶颈，怎么解决这个瓶颈？集群
4. 哪他们之间是如何进行数据交互的？Redis是如何进行持久化的？Redis数据都在内存中，一断电或重启不就没有了吗？主从复制-RDB　持久化方式： RDB AOF 4.0后增加了混合持久化。 做了持久化机制
5. 哪你是如何选择持久方式的？4.0之前 只是做缓存 RDB 其他是AOF和RDB都打开，不建议只使用AOF 4.0之后需要使用 就开启混合持久化 。混合持久化是对AOF的一个补充，主要是重写不一样
6. Redis还有其他保证集群**高可用**的方式吗？主从+哨兵
7. 数据传输的时候网络断了或者服务器断了，怎么办？增量同步 全量同步
8. 能说一下Redis的内存淘汰机制么？ LRU LFU day 04
9. 如果的如果，定期没删，我也没查询，那可咋整 ？设置最大内存，还要设置缓存淘汰策略
10. 哨兵机制的原理是什么? 监控-3个任务 判断下线-SDOWN和ODWON 选举 leader 自动故障迁移 day02
11. 哨兵组件的主要功能是什么？监控、通知（告警）、自动故障迁移
12. Redis的事务原理是什么？内存队列，批量处理，具有隔离性
13. Redis事务为什么是“弱”事务？days03 1.不是原子操作 2.不支持回滚
14. Redis为什么没有回滚操作？days03 1.大多数错误都是代码 和数据的问题，可以预判断 2.性能考虑

15. 在Redis中整合lua有几种方式，你认为哪种更好，为什么？写lua文件 redis.call()
16. lua如何解决Redis并发问题？原子性（隔离性）
17. 介绍Redis下的分布式锁实现原理、优势劣势和使用场景? days03
18. Redis-Cluster和Redis主从+哨兵的区别，你认为哪个更好，为什么。看业务

    如果我们读多写少 并发量大 数据只有2G 100W读 主从+哨兵

    如果并发写多 数据量大 40G数据，10W并发 Rediscluster
19. 什么情况下会造成缓存穿透，如何解决？缓存没数据，所有操作访问数据库。 服务器过滤 缓存降级 布隆过滤器 days04
20. 什么情况下会造成缓存雪崩，如何解决？大量key同时失效，又有大量并发访问。 主要两种：一种是过期时间+随机值 多级缓存，CDN和Nginx
21. 什么是缓存击穿，如何解决 热点key失效。持久化 二级缓存 days04
22. 什么情况下会造成数据库与Redis缓存数据不一致，如何解决? 只要双写就会不一致。

    一种最终一致性： 延迟删除 mysql binlog MQ 异步更新KV

    还有一种强一致性：KV-DB 必须保持一致 读请求和写请求串行化，传到内存队列。 比正常情况服务器多使用5.6-5.7倍
23. 那你了解的最经典的KV，DB读写模式什么样

    CAP模式

    读的时候 先读缓存，缓存没有的话 就读数据库，然后把取出的数据放入到缓存 ，同时返回响应

    更新的时候，先更新数据库，然后再删除缓存
24. 为什么是删除缓存，而不是更新缓存?

    用到缓存的时候再去算缓存

    Lazy 思想
25. Redis的线程模型你了解吗?

    文件处理器 由于文件处理器是单线程的 所以我们说redis是单线程。Redis的核心组件：

    Socket

    多路复用程序

    文件命令分配（分派）器

    文件处理器（命令接受器，命令处理器，命令响应处理器）

附录

Redis 5.0 新特性 12个

1. 新增加的Stream（流）数据类型
2. 新的Redis模块api : Times and Cluster api,是一个抽象的集群消息总线，用于方便开发分布式系统。
3. RDB（redis datebase）现在用于存储 LRU（最近最少使用淘汰算法） 和 LFU（最近不经常使用淘汰算法）元数据信息。

4. 集群管理器从ruby（redis-trib.rb）移植到c代码。以前创建集群时候需要通过ruby脚本来创建，现在用c代码重新编写，不用在额外按装ruby了。
5. 新增加有序集合的sorted set4个命令：
    1. ZPOPMAX
        - 作用：删除返回集合中分值最高的元素
        - 用法：ZPOPMAX key [count]
    2. ZPOPMIN
        - 作用：删除返回集合中分值最小的元素
        - 用法：ZPOPMIN key [count]
    3. BZPOPMAX
        - 作用：ZPOPMAX的阻塞版
        - 用法：BZPOPMAX key [key ...] timeout
    4. BZPOPMIN
        - 作用：ZPOPMIN的阻塞版
        - 用法：BZPOPMIN key [key ...] timeout
6. 主动内存碎片整理功能version2版本，依赖于Jemalloc内存分配器。
7. 增强HyperLogLog实现，这个功能是估算集合基数，redis5优化这个算法来节省空间
8. 更好的内存统计报告（碎片整理和内存报告）。

   当我们在redis里存一个key时候，redis会给这个key分配一个存储空间，但是当把这个key删了，redis是不会立即回收这个已经删除key所占用的空间。因此如果反复增加删除key的话，会产生很多内存碎片。这就会影响之后申请大块连续的内存空间，所以进行内存碎片整理很有必要。

   在redis4时候已经有自动整理内存碎片的功能了，不过那时候功能还属于实验阶段。

   redis5是在redis4的基础上将内存碎片自动清理功能进行了完善，现在该功能已经成熟。

   那么这个功能有如下作用：

   1.在redis运行期间自动进行内存碎片清理，可以实时释放内存空间。

   2.通过内存报告来了解整个系统的内存使用情况。

在redis配置文件中查看内存碎片控制相关参数

```
1351 # Enabled active defragmentation
1352  activedefrag yes
```

参数说明：

1） **activedefrag**：内存碎片功能启动配置项，当为yes就表示开启该功能。

2）**active-defrag-ignore-bytes**：当内存浪费小于100M就忽略，大于100M就启动内存碎片整理，这个值可以设置的。

3）**active-defrag-threshold-lower**：当内存浪费小于10%就暂时忽略，大于10%就启动内存碎片整理，这个值可以设置的。

9.许多带有子命令的命令现在都有一个help子命令。

10.客户端断开和连接时候性能更好。

11.错误修复和改进。

12.Jemalloc升级到5.1版本。

Redis 6.0 新特性

2020年5月2号 形成了稳定版本发布

简单介绍一下Redis6.0 有哪些重要新特性。

1.多线程IO

Redis 6引入多线程IO，但多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程。之所以这么设计是不想因为多线程而变得复杂，需要去控制 key、lua、事务，LPUSH/LPOP 等等的并发问题。

2.重新设计了客户端缓存功能

实现了Client-side-caching（客户端缓存）功能。放弃了caching slot，而只使用key names。

Redis server-assisted client side caching

3.RESP3协议

RESP（Redis Serialization Protocol）是 Redis 服务端与客户端之间通信的协议。Redis 5 使用的是 RESP2，而 Redis 6 开始在兼容 RESP2 的基础上，开始支持 RESP3。

推出RESP3的目的：一是因为希望能为客户端提供更多的语义化响应，以开发使用旧协议难以实现的功能；另一个原因是实现 Client-side-caching（客户端缓存）功能。

RESP3

4.支持SSL

连接支持SSL，更加安全。

5.ACL权限控制

1. 支持对客户端的权限控制，实现对不同的key授予不同的操作权限。
2. 有一个新的ACL日志命令，允许查看所有违反ACL的客户机、访问不应该访问的命令、访问不应该访问的密钥，或者验证尝试失败。这对于调试ACL问题非常有用。

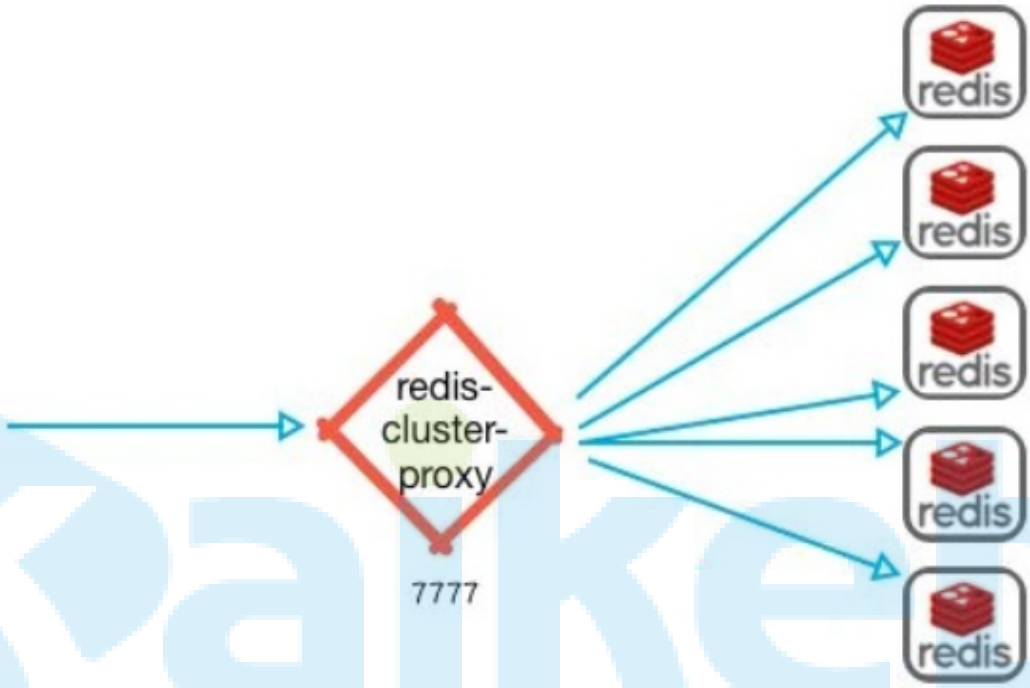
6.提升了RDB日志加载速度

根据文件的实际组成（较大或较小的值），可以预期20/30%的改进。当有很多客户机连接时，信息也更快了，这是一个老问题，现在终于解决了。

7.提供了众多的新模块（modules）API

8.发布官方的Redis集群代理模块 Redis Cluster proxy

在 Redis 集群中，客户端会非常分散，现在为此引入了一个集群代理，可以为客户端抽象 Redis 群集，使其像正在与单个实例进行对话一样。同时在简单且客户端仅使用简单命令和功能时执行多路复用。



安装 6.0

centos 7 直接安装redis6。报错了，需要升级GCC 版本

redis 6.0 需要GCC 8.5 以上

```
$ wget http://download.redis.io/releases/redis-6.0.6.tar.gz
$ tar xzf redis-6.0.6.tar.gz

#升级到 5.3及以上版本
yum -y install centos-release-scl
yum -y install devtoolset-9-gcc devtoolset-9-gcc-c++ devtoolset-9-binutils

scl enable devtoolset-9 bash

#注意：scl命令启用只是临时的，退出xshell或者重启就会恢复到原来的gcc版本。
#如果要长期生效的话，执行如下：
echo "source /opt/rh/devtoolset-9/enable" >>/etc/profile

$ cd redis-6.0.6
$ make
$ make install PREFIX=/usr/local/redis6
```