# 查看内存使用情况(info)

[root@localhost bin]# redis-cli

127.0.0.1:6379> info

# Clients

connected\_clients:1 #查看客户端连接数

client\_recent\_max\_input\_buffer:2

client\_recent\_max\_output\_buffer:0

blocked\_clients:0

# Memory

used\_memory:832784 # Redis分配的内存总量

used\_memory\_human:813.27K

used\_memory\_rss:1896448 # Redis分配的内存总量(包括内存碎片)

used\_memory\_peak:832760

used\_memory\_peak\_human:813.24K #Redis所用内存的高峰值

# redis线程模型

原文：https://blog.csdn.net/y277an/article/details/98342442

Redis基于Reactor模式开发了网络事件处理器，这个处理器被称为文件事件处理器。它的组成结构为4部分：多个socket、IO多路复用程序、文件事件分派器、事件处理器（包括连接应答处理器、命令请求处理器、命令回复处理器）。

多个socket可能会并发产生不同的操作，每个操作对应不同的文件事件。IO多路复用程序同时监听多个socket，然后将产生事件的socket压入内存队列中。事件分派器每次从队列中取出一个socket，根据socket 的事件类型交给对应的事件处理器进行处理。因为文件事件分派器队列的消费是单线程的，而redis基于它工作，所以Redis才叫单线程模型。

要明白，进程通信是通过socket来完成的，不懂的同学可以先看一看socket网络编程

流程如下：

1）建立socket连接

客户端socket01向redis的ServerSocket请求建立连接，此时ServerSocket会产生一个AE\_READABLE事件，IO多路复用程序监听到ServerSocket产生的事件后，将该事件压入队列中。文件事件分派器从队列中获取该事件，并交给连接应答处理器。连接应答处理器会创建一个能与客户端通信的socket01，并将该socket01的AE\_READABLE事件与命令请求处理器关联。（图中队列的ss）

2）发送set key value请求

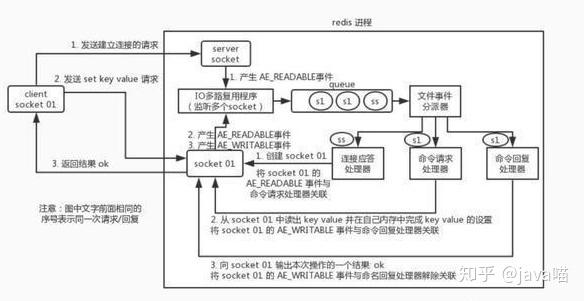
假设此时客户端发送set key value请求，此时redis中的socket01会产生AE\_READABLE事件，IO多路复用程序将事件压入队列中，此时事件分派器从队列中获取到该事件，由于前面 socket01 的 AE\_READABLE 事件已经与命令请求处理器关联，因此事件分派器将事件交给命令请求处理器来处理。命令请求处理器读取 socket01 的 key value 并在自己内存中完成 key value 的设置。操作完成后，它会将 socket01 的 AE\_WRITABLE 事件与命令回复处理器关联。（图中队列中间的s1）

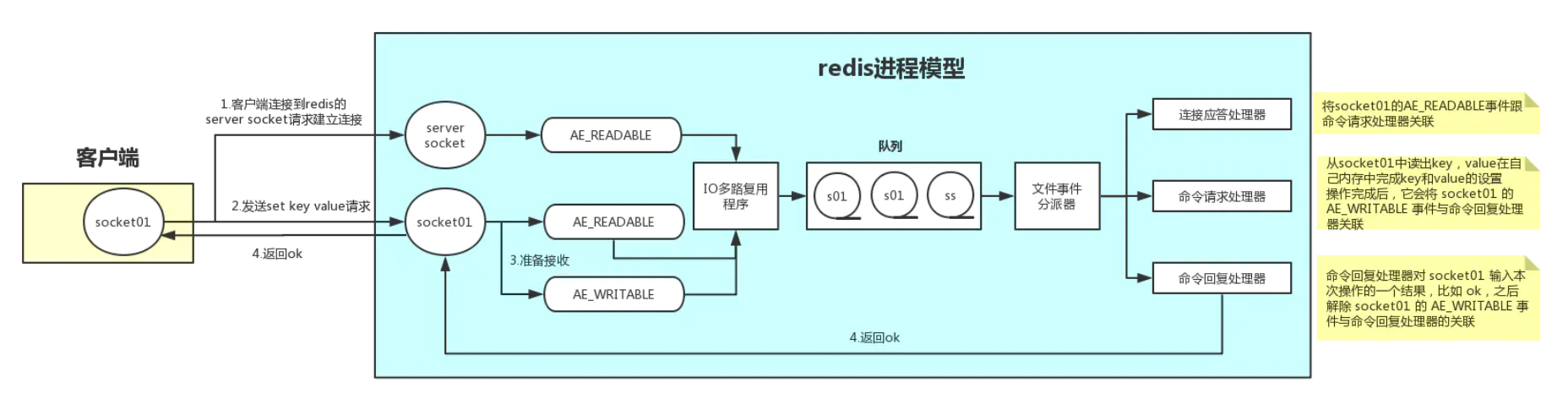
3）准备接收回复

如果此时客户端准备好接收返回结果了，那么 redis 中的 socket01 会产生一个 AE\_WRITABLE 事件，同样压入队列中，事件分派器找到相关联的命令回复处理器，由命令回复处理器对 socket01 输入本次操作的一个结果，比如 ok，之后解除 socket01 的 AE\_WRITABLE 事件与命令回复处理器的关联。

这样便完成了一次通信

客户端与redis的一次通信过程：





问：为啥redis单线程模型也能效率这么高？

纯内存操作；

核心是基于非阻塞的IO多路复用机制；

单线程反而避免了多线程的频繁上下文切换问题。

# redis QPS性能测试

单节点TPS达到8万/秒，QPS达到10万/秒。

QPS：应用系统每秒钟最大能接受的用户访问量。每秒钟处理完请求的次数，注意这里是处理完，具体是指发出请求到服务器处理完成功返回结果。可以理解在server中有个counter，每处理一个请求加1，1秒后counter=QPS。

TPS：每秒钟最大能处理的请求数。每秒钟处理完的事务次数，一个应用系统1s能完成多少事务处理，一个事务在分布式处理中，可能会对应多个请求，对于衡量单个接口服务的处理能力，用QPS比较合理

redis自带性能测试工具：redis-benchmark，可用它计算QPS。

格式：> redis-benchmark [option] [option value]

实例：100个客户端连接，发出10000个请求

> ./redis-benchmark -h 127.0.0.1 -p 6379 -c 100 -n 10000

实例：100个客户端连接，发出10000个请求，每个请求数据为100个字节的value

> ./redis-benchmark -h 127.0.0.1 -p 6379 -c 100 -n 10000 -d 100

**示例：测试服务端性能，50个并发连接，发出100000个请求，每个请求数据为2KB**

# 查看redis服务

[root@localhost bin]# pwd

/usr/local/bin

# 查看redis进程

[root@localhost bin]# ps -ef | grep redis

root 1051 1 0 12:24 ? 00:00:33 /usr/local/bin/redis-server 127.0.0.1:6379

root 1790 1489 0 14:06 pts/0 00:00:00 grep --color=auto redis

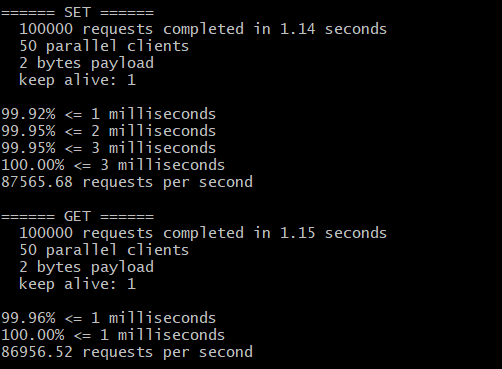
# 对该redis服务进行性能测试

[root@localhost bin]# ./redis-benchmark -h 127.0.0.1 -p 6379 -c 50 -n 100000 -d 2

说明：

会对ping、set、get、incr、lpush、rpush、lpop、rpop、sadd、hset、lrange分别进行性能测试。

我们关注最后一行：每秒87565.68个请求，即有8.7万QPS；这里的数据都只是测试数据，测出来的QPS不能代表实际生产的处理能力。



# 生产环境的redis部署

问题：需要了解公司的redis生产环境的部署架构，redis的主从架构？集群架构？用了哪种集群方案？有没有做高可用？有没有开启持久化确保数据恢复？线上redis给了几个G的内存？设置了哪些参数？压测后你们redis集群承载多少QPS？

描述：Redis Cluster，10台机器，5台机器部署了redis主节点，另外5台机器部署了redis从节点，每个主节点挂了一个从节点，5个节点对外提供读写服务，每个节点的读写高峰QPS可以达到每秒5万，5台机器最多是每秒25万读写请求。

机器是什么配置？8和CPU+32G内核+1T磁盘，但是分配给redis进程的是10g内存，一般线上生成环境，redis的内存尽量不要超过10g，超过10g可能会有问题。

每个主节点都挂了一个从节点，所以是高可用的，任何一个主节点宕机，都会自动故障迁移，redis从节点会自动变成主节点继续提供读写服务。

你往内存里写的是什么数据？每条数据的大小是多少？商品详细数据，每条数据是10kb。100条数据是1mb，10万条数据是1g。常驻内存的是200万条商品数据，占用内存是20g，仅仅不到总内存的50%。目前高峰期每秒就是3500左右的请求量。

# Redis的并发竞争问题和解决方案？

描述：多个客户端同时获取一个key，修改值后再写回去，但是顺序错了，数据就错了。如集群环境下多个子系统去set一个key，如某个key1，系统A设置key1=valueA，系统B设置key1=valueB，系统C设置key1=valueC。期望key1的顺序为valuaA->valueB->valueC。

解决1：基于zookeeper实现分布式锁。

每个系统通过zookeeper获取分布式锁，确保同一时间只能由一个系统在操作某个key，其它都不允许读和写。

解决2：若为自增/自减这类的操作，可使用redis自带的incr/desc命令

解决3：利用消息队列，将set方法变成串行化访问

解决4：基于lua实现cas乐观锁方案

1）set key value操作：KEYS[1]为key，ARGV[1]对应要比较的值，值相同则更新成 ARGV[2]，并返回 1，否则返回 0。

if redis.call("get", KEYS[1]) == ARGV[1] then

redis.call("set", KEYS[1], ARGV[2])

return 1

else

return 0

end

2）hset key field value操作：KEYS[1]为key，ARGV[1] 对应 Hash 的 field，ARGV[2]对应要比较的值，值相同则更新成 ARGV[3]，并返回 1，否则返回 0。

if redis.call("hget", KEYS[1], ARGV[1]) == ARGV[2] then

redis.call("hset", KEYS[1], ARGV[1], ARGV[3])

return 1

else

return 0

end

# 缓存和数据库双写一致性？

读请求：先读缓存->无则读数据库->更新缓存

写请求：先更新数据库，后删除缓存。（删除的数据在进行读请求后会重新加入缓存）

# 高并发场景中读请求长时间阻塞

描述：内存队列中可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分担一些数据的更新操作。如果一个内存队列里积压了100个商品库存修改操作，每个库存修改操作要耗费10ms去完成，那么队列最后有一个读操作，可能等待10\*100=1000ms=1s后，才能得到数据，这个时候导致读请求的长时间阻塞。

解决：如果一个内存队列中可能积压的更新操作特别多，那么你就要加机器，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少。

其实根据之前的项目经验，一般来说，数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的。像这种针对读高并发、读缓存架构的项目，一般来说写请求是非常少的，每秒的 QPS 能到几百就不错了。

我们来实际粗略测算一下。

如果一秒有 500 的写操作，如果分成 5 个时间片，每 200ms 就 100 个写操作，放到 20 个内存队列中，每个内存队列，可能就积压 5 个写操作。每个写操作性能测试后，一般是在 20ms 左右就完成，那么针对每个内存队列的数据的读请求，也就最多 hang 一会儿，200ms 以内肯定能返回了。

经过刚才简单的测算，我们知道，单机支撑的写 QPS 在几百是没问题的，如果写 QPS 扩大了 10 倍，那么就扩容机器，扩容 10 倍的机器，每个机器 20 个队列。

# Redis Cluster集群

## Redis cluster介绍

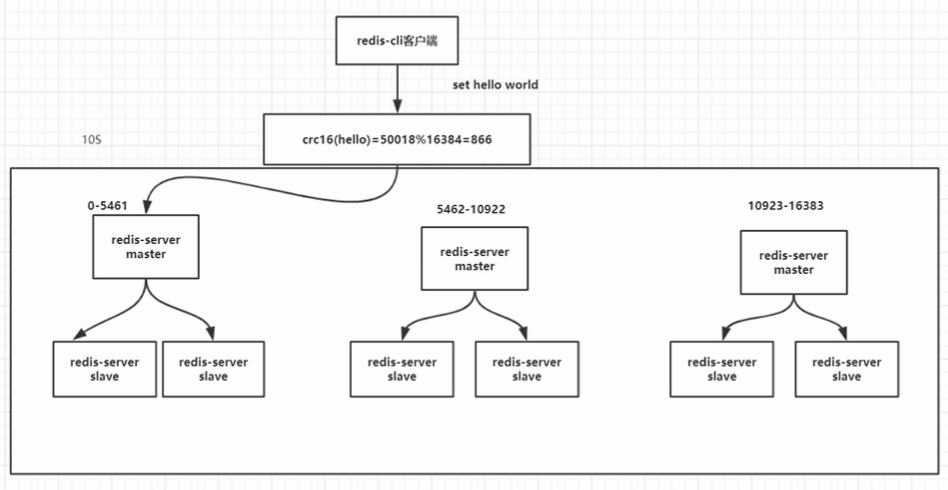
自动将数据进行分片，每个master上放一部分数据。

提供内置的高可用支持，部分master节点不可用时，其它master节点还是可以继续工作的。

redis cluster 功能强大，直接集成了 replication 和sentinel 的功能。

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加 1w 的端口号，比如 16379。16379 端口号是用来进行节点间通信的，也就是 cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。

cluster bus 用了另外一种二进制的协议-gossip协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。



## Redis cluster节点间通信-gossip协议

redis cluster 节点间采用 gossip 协议进行通信。

redis 维护集群元数据（节点信息、故障等等）采用gossip 协议，所有节点都持有一份元数据，不同的节点若出现了元数据的变更，就不断将元数据发送给其它的节点，让其它节点也进行元数据的变更。

gossip 好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续打到所有节点上去更新，降低了压力；不好在于，元数据的更新有延时，可能导致集群中的一些操作会有一些滞后。

Gossip的通信端口：每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如7001，那么用于节点间通信的就是 17001 端口。每个节点每隔一段时间都会往另外几个节点发送 ping 消息，同时其它几个节点接收到 ping 之后返回 pong。

交换的信息：信息包括故障信息，节点的增加和删除，hash slot 信息等等。

gossip 协议包含多种消息，包含 ping、pong、meet、fail 等等。

1）meet：某个节点发送 meet 给新加入的节点，让新节点加入集群中，然后新节点就会开始与其它节点进行通信。redis-trib.rb add-node其实内部就是发送了一个 gossip meet 消息给新加入的节点，通知那个节点去加入我们的集群。

2）ping：每个节点都会频繁给其它节点发送 ping，其中包含自己的状态还有自己维护的集群元数据，互相通过 ping 交换元数据。

3）pong：返回ping和meeet，包括自己的状态和其他信息，也用于信息广播和更新。

4）fail：某个节点判断另一个节点fail之后，发送fail给其他节点，通知其他节点说，某个节点宕机啦。

## Redis cluster的哈希槽(hash slot)算法

分布式系统寻址算法大致有三种：

简单hash取模算法（增加节点需要重建）

一致性hash算法（自动缓存迁移）+虚拟节点（自动负载均衡）

Redis cluster的哈希槽（hash slot）算法

Redis cluster有固定的16384个哈希槽（hash slot），对每个key计算CRC16值，然后对16384取模，可以获取key对应的哈希槽。

redis cluster 中每个 master 都会持有部分 slot，比如有 3 个 master，那么可能每个 master 持有5000 多个 hash slot。hash slot 让 node 的增加和移除很简单，增加一个 master，就将其他 master的 hash slot 移动部分过去，减少一个 master，就将它的 hash slot 移动到其他 master 上去。移动hash slot 的成本是非常低的。客户端的 api，可以对指定的数据，让他们走同一个 hash slot，通过 hash tag 来实现。

任何一台机器宕机，另外两个节点，不影响的。因为 key 找的是 hash slot，不是机器。

转移槽位：可以以下两种方式

》/usr/local/bin/redis-cli --cluster reshard 192.168.2.101:7002

》./redis-trib.rb reshard --from bdd63e1f522d78eb1bb2574b2461a7302e14944a  --to 1d204c88a14a76dc30abb05025135f7e850f2a5d  --slots 5461 --yes 192.168.112.36:6004

## Redis cluster的高可用性与主备切换原理

Redis Cluster的高可用原理，几乎跟哨兵是类似的。

**判断节点宕机：**

如果一个节点认为另外一个节点宕机，那么就是 pfail，主观宕机。如果多个节点都认为另外一个节点宕机了，那么就是 fail，客观宕机，跟哨兵的原理几乎一样，sdown，odown。

在 cluster-node-timeout 内，某个节点一直没有返回 pong，那么就被认为 pfail。

如果一个节点认为某个节点 pfail 了，那么会在 gossip ping 消息中，ping 给其他节点，如果超过半数的节点都认为 pfail 了，那么就会变成 fail。

**从节点选举：**

对宕机的 master node，从其所有的 slave node 中，选择一个切换成 master node。

每个从节点，都根据自己对 master 复制数据的 offset，来设置一个选举时间，offset 越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举。

所有的 master node 开始 slave 选举投票，给要进行选举的 slave 进行投票，如果大部分 master node（N/2 + 1）都投票给了某个从节点，那么选举通过，那个从节点可以切换成 master。

从节点执行主备切换，从节点切换为主节点。