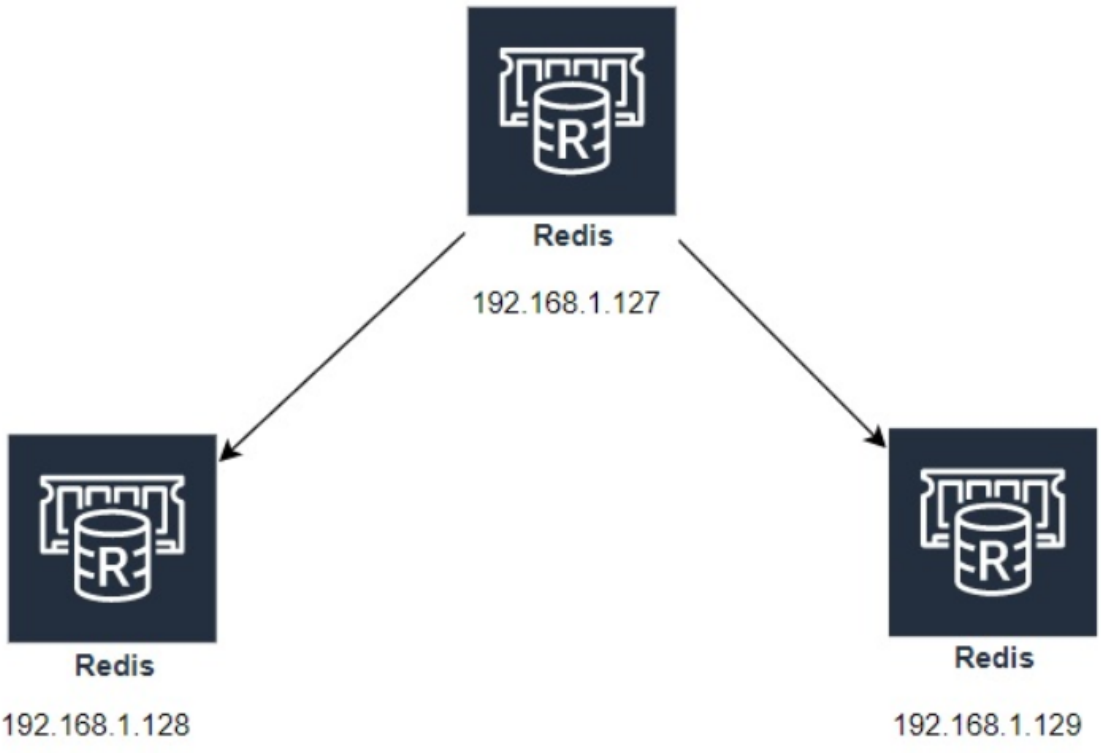
Redis的主从复制

零声学院：https://0voice.ke.qq.com/?tuin=137bb271

1、主从复制的概念

redis为了实现高可用（比如解决单点故障的问题），会把数据复制多个副本部署到其他节点上，通过复制，实现Redis的高可用性，实现对数据的冗余备份，保证数据和服务的可靠性。比如：



如何配置的：

- 配置文件： 在从服务器的配置文件中加入：slaveof
- 启动命令： redis-server启动命令后加入 --slaveof

- 客户端命令： Redis服务器启动后，直接通过客户端执行命令：slaveof ，则该Redis实例成为从节点。

> PS：通过 info replication 命令可以看到复制的一些信息

```
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:192.168.1.127
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:0
master_link_down_since_seconds:1575946149
slave_priority:100
slave_read_only:1
connected_slaves:0
master_replid:43994f8eb09cfdc95dbbcbdbec9cbfc39c99567c
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6379>
```

主从复制的作用

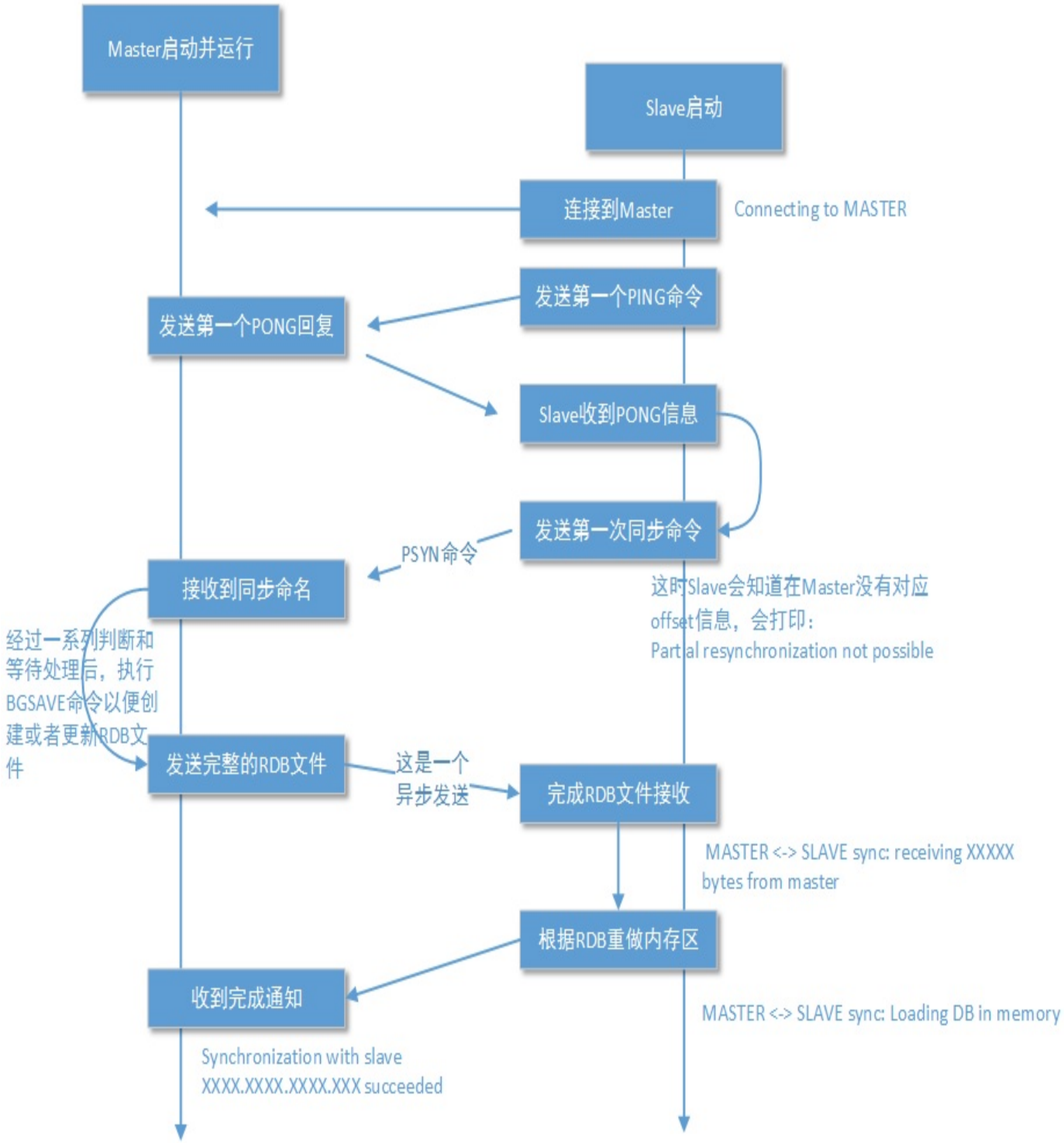
- 数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
- 故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；实际上是一种服务的冗余。
- 负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写Redis数据时应用连接主节点，读Redis数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高Redis服务器的并发量。
- 读写分离：可以用于实现读写分离，主库写、从库读，读写分离不仅可以提高服务器的负载能力，同时可根据需求的变化，改变从库的数量；
- 高可用基石：除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是Redis高可用的基础。

2、主从复制的机制

Redis的主从复制功能除了支持一个Master节点对应多个Slave节点的同时进行复制外，还支持Slave节点向其它多个Slave节点进行复制。这样就使得架构师能够灵活组织业务缓存数据的传播，例如使用多个Slave作为数据读取服务的同时，专门使用一个Slave节点为流式分析工具服务。Redis的主从复制功能分为两种数据同步模式进行：全量数据同步和增量数据同步。

2.1、全量数据同步（full resyncchrozation）

先执行一次全同步 — 请求master BgSave出自己的一个RDB Snapshot文件发给slave，slave接收完毕后，清除掉自己的旧数据，然后将RDB载入内存。

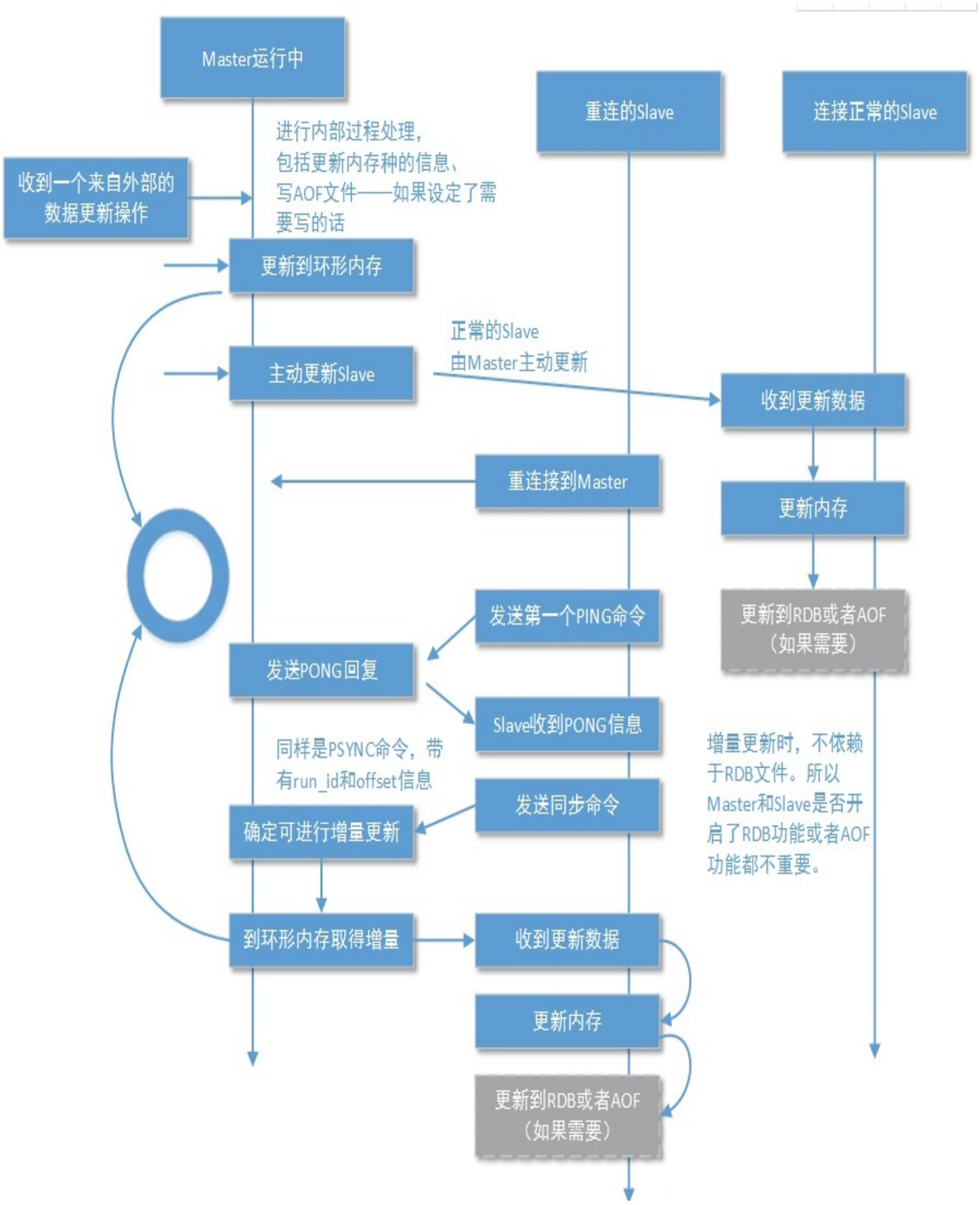


上图简要说明了Redis中Master节点到Slave节点的全量数据同步过程。当Slave节点给定的replication id和Master的replication id不一致时，或者Slave给定的上一次增量同步的offset的位置在Master的环形内存中(replication backlog)无法定位时（后文会提到），Master就会对Slave发起全量同步操作。这时无论您是否在Master打开了RDB快照功能，它和Slave节点的每一次全量同步操作过程都会更新/创建Master上的RDB文件。在Slave连接到Master，并完成第一次全量数据同步后，接下来Master到Slave的数据同步过程一般就是增量同步形式了（也称为部分同步）。增量同步过程不再主要依赖RDB文件，

Master会将新产生的数据变化操作存放在replication backlog这个内存缓存区，这个内存区域是一个环形缓冲区，也就是说是一个FIFO的队列。

2.2、 增量同步

进行增量同步 — master作为一个普通的client连入slave，将所有写操作转发给slave，没有特殊的同步协议。具体过程如下：



为什么在Master上新增的数据除了根据Master节点上RDB或者AOF的设置进行日志文件更新外，还会同时将数据变化写入一个环形内存结构（replication backlog），并以后者为依据进行Slave节点的增量更新呢？主要原因有以下几个：

- 由于网络环境的不稳定，网络抖动/延迟都可能造成Slave和Master暂时断开连接，这种情况要远远多于新的Slave连接到Master的情况。如果以上所有情况都使用全量更新，就会大大增加Master的负载压力——写RDB文件是有大量I/O过程的，虽然Linux Page Cahe特性会减少性能消耗。
- 另外在数据量达到一定规模的情况下，使用全量更新进行和Slave的第一次同步是一个不得已的选择——因为要尽快减少Slave节点和Master节点的数据差异。所以只能占用Master节点的资源和网络带宽资源。
- 使用内存记录数据增量操作，可以有效减少Master节点在这方面付出的I/O代价。而做成环形内存的原因，是为了保证在满足数据记录需求的情况下尽可能减少内存的占用量。这个环形内存的大小，可以通过repl-backlog-size参数进行设置。

Slave重连后会向Master发送之前接收到的Master replication id信息和上一次完成部分同步的offset的位置信息。如果Master能够确定这个replication id和自己的replication id(有两个)一致且能够在环形内存中找到这个offset的位置，Master就会发送从offset的位置开始向Slave发送增量数据。那么连接正常的各个Slave节点如何接受新数据呢？连接正常的Slave节点将会在Master节点将数据写入环形内存后，主动接收到来自Master的数据复制信息。

这里就有一个问题了，**我们的replication backlog的size设置为多大合适?**

redis为replication backlog设置的默认大小为1M（repl-backlog-size），但是这个值是可以调整的，如果主服务器需要执行大量的写命令，又或者主从服务器之间断线后重连的时间比较长，那么这个大小也许并不合适。如果replication backlog的大小设置不恰当，那么PSYNC命令的复制同步模式就不能正常发挥作用，因此，正确估算和设置replication backlog的size非常重要。

reconnect_time_second * write_size_per_second*2

- reconnect_time_second：重连时间，以秒未单位
- write_size_per_second：每秒写入的命令大小

3、主从复制的实现

先来看一段log：

```
1440:M 09 Dec 2019 22:27:19.963 * Replica 192.168.1.128:6379 asks for synchronization
1440:M 09 Dec 2019 22:27:19.963 * Partial resynchronization not accepted: Replication ID mismatch (Replica asked for '8dfda0b4f573438e5f35131df02752da598bb124', my replication IDs are '55812dacdc7d3354661404725708b8d96f29c80f' and '0948cdd260edff4623ec7342b37497b49c0bb8ca')
1440:M 09 Dec 2019 22:27:19.964 * Starting BGSAVE for SYNC with target: disk
1440:M 09 Dec 2019 22:27:19.964 * Background saving started by pid 2302
2302:C 09 Dec 2019 22:27:19.967 * DB saved on disk
2302:C 09 Dec 2019 22:27:19.967 * RDB: 6 MB of memory used by copy-on-write
```

```
1440:M 09 Dec 2019 22:27:19.981 * Background saving terminated with success
1440:M 09 Dec 2019 22:27:19.982 * Synchronization with replica 192.168.1.128:6379 succeeded
```

上文的log是我在从机器上输入slaveof 192.168.1.127 6379后，master机器上的一段日志。下面这段是slava机器上的log

```
908:S 09 Dec 2019 22:27:19.394 * REPLICAOF 192.168.1.127:6379 enabled (user request from 'id=3 addr=127.0.0.1:53158 fd=7 name= age=18 idle=0 flags=N db=0 sub=0 psub=0 multi=-1 qbuf=47 qbuf-free=32721 obl=0 oll=0 omem=0 events=r cmd=slaveof')
908:S 09 Dec 2019 22:27:19.974 * Connecting to MASTER 192.168.1.127:6379
908:S 09 Dec 2019 22:27:19.975 * MASTER <-> REPLICA sync started
908:S 09 Dec 2019 22:27:19.975 * Non blocking connect for SYNC fired the event.
908:S 09 Dec 2019 22:27:19.975 * Master replied to PING, replication can continue...
908:S 09 Dec 2019 22:27:19.976 * Trying a partial resynchronization (request 8dfda0b4f573438e5f35131df02752da598bb124:1).
908:S 09 Dec 2019 22:27:19.979 * Full resync from master: 15489bcd0b74269cad5f767ebc67c7510506778a:0
908:S 09 Dec 2019 22:27:19.979 * Discarding previously cached master state.
908:S 09 Dec 2019 22:27:19.995 * MASTER <-> REPLICA sync: receiving 12801 bytes from master
908:S 09 Dec 2019 22:27:19.995 * MASTER <-> REPLICA sync: Flushing old data
908:S 09 Dec 2019 22:27:19.996 * MASTER <-> REPLICA sync: Loading DB in memory
908:S 09 Dec 2019 22:27:19.996 * MASTER <-> REPLICA sync: Finished with success
```

首先，在redis内部，所有的命令都维护在一张表格里：

```
struct redisCommand redisCommandTable[] = {
    {"get",getCommand,2,
     "read-only fast @string",
     0,NULL,1,1,1,0,0,0},
    /* Note that we can't flag set as fast, since it may perform
```

```
    an
6       * implicit DEL of a large key. */
7      {"set",setCommand,-3,
8       "write use-memory @string",
9       0,NULL,1,1,1,0,0,0},
10     {"setnx",setnxCommand,3,
11      "write use-memory fast @string",
12      0,NULL,1,1,1,0,0,0},
13     {"slaveof",replicaofCommand,3,
14      "admin no-script ok-stale",
15      0,NULL,0,0,0,0,0,0},
16
17      {"sync",syncCommand,1,
18      "admin no-script",
19      0,NULL,0,0,0,0,0,0},
20     {"psync",syncCommand,3,
21      "admin no-script",
22      0,NULL,0,0,0,0,0,0},
23
24      ...
25 }
```

PS：当然所有的这些command，在启动的时候会调用populateCommandTable函数进行分析，然后把所有的redis command放在server.commands，它是一个字典。

```
dict commands;          / Command table */
```

当我们在客户端执行slaveof 的，在客户端这一侧会调用replicaofCommand这个函数。

3.1、 主从关系的建立

主从复制建立的方式有三种：

- 在redis.conf文件中配置slaveof 选项，然后指定该配置文件启动Redis生效。
- 在redis-server启动命令后加上--slaveof 启动生效。
- 直接使用 slaveof 命令在从节点执行生效。

无论是通过哪一种方式来建立主从复制，都是从节点来执行slaveof命令，那么从节点执行了这个命令到底做了什么，我们上源码：

```
1 void replicaofCommand(client *c) {
2     /* SLAVEOF is not allowed in cluster mode as replication is a
  utomatically
3      * configured using the current address of the master node.
   */
```

```
    // 如果是cluster模式（集群模式），则不能执行这个命令
    if (server.cluster_enabled) {
        addReplyError(c,"REPLICAOF not allowed in cluster mode.")
;
        return;
    }
    /* The special host/port combination "NO" "ONE" turns the ins
tance
     * into a master. Otherwise the new master address is set. */
     // 如果是slaveof no one，则该节点会变成master节点，此时会调用replic
ationUnsetMaster取消原来的复制操作
    if (!strcasecmp(c->argv[1]->ptr,"no") &&
        !strcasecmp(c->argv[2]->ptr,"one")) {
        if (server.masterhost) {
            replicationUnsetMaster();
            sds client = catClientInfoString(sdsempty(),c);
            serverLog(LL_NOTICE,"MASTER MODE enabled (user reques
t from '%s')",
                client);
            sdsfree(client);
            /* Restart the AOF subsystem in case we shut it down
 during a sync when
             * we were still a slave. */
            if (server.aof_enabled && server.aof_state == AOF_OFF
) restartAOFAfterSYNC();
        }
    } else {
        long port;
        if (c->flags & CLIENT_SLAVE)
        {
            /* If a client is already a replica they cannot run t
his command,
             * because it involves flushing all replicas (includi
ng this
             * client) */
            addReplyError(c, "Command is not valid when client is
a replica.");
            return;
        }
        if ((getLongFromObjectOrReply(c, c->argv[2], &port, NULL)
```

```
!= C_OK))
35              return;
36          /* Check if we are already attached to the specified slav
    e */
37          if (server.masterhost && !strcasecmp(server.masterhost,c-
    >argv[1]->ptr)
38              && server.masterport == port) {
39              serverLog(LL_NOTICE,"REPLICAOF would result into sync
    hronization "
40                                  "with the master we are already c
    onnected "
41                                  "with. No operation performed.");
42              addReplySds(c,sdsnew("+OK Already connected to specif
    ied "
43                                   "master\r\n"));
44              return;
45          }
46          /* There was no previous master or the user specified a d
    ifferent one,
47           * we can continue. */
48          // 第一次执行设置端口和ip，或者是重新设置端口和IP
49          // 设置服务器复制操作的主节点IP和端口
50          replicationSetMaster(c->argv[1]->ptr, port);
51          sds client = catClientInfoString(sdsempty(),c);
52          serverLog(LL_NOTICE,"REPLICAOF %s:%d enabled (user reques
    t from '%s')",
53              server.masterhost, server.masterport, client);
54          sdsfree(client);
55      }
56      addReply(c,shared.ok);
57  }
```

而SLAVEOF命令做的操作并不多，主要以下三步：

- 判断当前环境是否在集群模式下，因为集群模式下不行执行该命令。
- 是否执行的是SLAVEOF NO ONE命令，该命令会断开主从的关系，设置当前节点为主节点服务器。
- 设置从节点所属主节点的IP和port。调用了replicationSetMaster()函数。

```
1 // 设置复制操作的主节点IP和端口
2 void replicationSetMaster(char *ip, int port) {
3     // 按需清除原来的主节点信息
```

```
    sdsfree(server.masterhost);
    // 设置ip和端口
    server.masterhost = sdsnew(ip);
    server.masterport = port;
    // 如果有其他的主节点，在释放
    // 例如服务器1是服务器2的主节点，现在服务器2要同步服务器3，服务器3要成为服务器2的主节点，因此要释放服务器1
    if (server.master) freeClient(server.master);
    // 解除所有客户端的阻塞状态
    disconnectAllBlockedClients(); /* Clients blocked in master, now slave. */
    // 关闭所有从节点服务器的连接，强制从节点服务器进行重新同步操作
    disconnectSlaves(); /* Force our slaves to resync with us as well. */
    // 释放主节点结构的缓存，不会执行部分重同步PSYNC
    replicationDiscardCachedMaster(); /* Don't try a PSYNC. */
    // 释放复制积压缓冲区
    freeReplicationBacklog(); /* Don't allow our chained slaves to PSYNC. */
    // 取消执行复制操作
    cancelReplicationHandshake();
    // 设置复制必须重新连接主节点的状态
    server.repl_state = REPL_STATE_CONNECT;
    // 初始化复制的偏移量
    server.master_repl_offset = 0;
    // 清零连接断开的时长
    server.repl_down_since = 0;
}
```

由代码知，replicationSetMaster()函数执行操作的也很简单，总结为两步：

- 清理之前所属的主节点的信息。
- 设置新的主节点IP和port等。 因为，当前从节点有可能之前从属于另外的一个主节点服务器，因此要清理所有关于之前主节点的缓存、关闭旧的连接等等。然后设置该从节点的新主节点，设置了IP和port，还设置了以下状态：

> // 设置复制必须重新连接主节点的状态 server.repl_state = REPL_STATE_CONNECT; // 初始化全局复制的偏移量 server.master_repl_offset = 0;

slaveof命令是一个异步命令，执行命令时，从节点保存主节点的信息，确立主从关系后就会立即返回，后续的复制流程在节点内部异步执行。那么，如何触发复制的执行呢?

周期性执行的函数：replicationCron()函数，该函数被服务器的时间事件的回调函数serverCron()所调用，而serverCron()函数在Redis服务器初始化时，被设置为时间事件的处理函数。

```
// void initServer(void) Redis服务器初始化
aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL)
```

replicationCron这个函数每秒才执行一次，以下代码执行到serverCron函数的实现：

```
/* Replication cron function -- used to reconnect to master,
     * detect transfer failures, start background RDB transfers an
  d so forth. */
    run_with_period(1000) replicationCron();
```

主从关系建立后，从节点服务器的server.repl_state被设置为REPL_STATE_CONNECT，而replicationCron()函数会被每秒执行一次，该函数会发现我（从节点）现在有主节点了，而且我要的状态是要连接主节点（REPL_STATE_CONNECT）。
replicationCron()函数处理这以情况的代码如下：

```
/* Check if we should connect to a MASTER */
// 如果处于要必须连接主节点的状态，尝试连接
if (server.repl_state == REPL_STATE_CONNECT) {
    serverLog(LL_NOTICE,"Connecting to MASTER %s:%d",
        server.masterhost, server.masterport);
    // 以非阻塞的方式连接主节点
    if (connectWithMaster() == C_OK) {
        serverLog(LL_NOTICE,"MASTER <-> SLAVE sync started");
    }
}
```

replicationCron()函数根据从节点的状态，调用connectWithMaster()非阻塞连接主节点。代码如下：

```
// 以非阻塞的方式连接主节点
int connectWithMaster(void) {
    int fd;
    // 连接主节点
    fd = anetTcpNonBlockBestEffortBindConnect(NULL,
        server.masterhost,server.masterport,NET_FIRST_BIND_ADDR);
    if (fd == -1) {
        serverLog(LL_WARNING,"Unable to connect to MASTER: %s",
            strerror(errno));
        return C_ERR;
    }
```

```
    // 监听主节点fd的可读和可写事件的发生，并设置其处理程序为syncWithMaster
    if (aeCreateFileEvent(server.el,fd,AE_READABLE|AE_WRITABLE,syncWithMaster,NULL) == AE_ERR)
    {
        close(fd);
        serverLog(LL_WARNING,"Can't create readable event for SYNC");
        return C_ERR;
    }
    // 最近一次读到RDB文件内容的时间
    server.repl_transfer_lastio = server.unixtime;
    // 从节点和主节点的同步套接字
    server.repl_transfer_s = fd;
    // 处于和主节点正在连接的状态
    server.repl_state = REPL_STATE_CONNECTING;
    return C_OK;
}
```

connectWithMaster()函数执行的操作可以总结为：
根据IP和port非阻塞的方式连接主节点，得到主从节点进行通信的文件描述符fd，并保存到从节点服务器server.repl_transfer_s中，并且将刚才的REPL_STATE_CONNECT状态设置为REPL_STATE_CONNECTING。 监听fd的可读和可写事件，并且设置事件发生的处理程序syncWithMaster()函数。 至此，主从网络建立就完成了，逐步进入了协商交互阶段，使用的是状态机处理的，详见syncWithMaster函数的处理，这些状态机的定义如下：

```
/* Slave replication state. Used in server.repl_state for slaves to remember
* what to do next. */
#define REPL_STATE_NONE 0 /* No active replication */
#define REPL_STATE_CONNECT 1 /* Must connect to master */
#define REPL_STATE_CONNECTING 2 /* Connecting to master */
/* --- Handshake states, must be ordered --- */
#define REPL_STATE_RECEIVE_PONG 3 /* Wait for PING reply */
#define REPL_STATE_SEND_AUTH 4 /* Send AUTH to master */
#define REPL_STATE_RECEIVE_AUTH 5 /* Wait for AUTH reply */
#define REPL_STATE_SEND_PORT 6 /* Send REPLCONF listening-port */
#define REPL_STATE_RECEIVE_PORT 7 /* Wait for REPLCONF reply */
#define REPL_STATE_SEND_IP 8 /* Send REPLCONF ip-address */
#define REPL_STATE_RECEIVE_IP 9 /* Wait for REPLCONF reply */
#define REPL_STATE_SEND_CAPA 10 /* Send REPLCONF capa */
```

```
#define REPL_STATE_RECEIVE_CAPA 11 /* Wait for REPLCONF reply */
#define REPL_STATE_SEND_PSYNC 12 /* Send PSYNC */
#define REPL_STATE_RECEIVE_PSYNC 13 /* Wait for PSYNC reply */
/* --- End of handshake states --- */
#define REPL_STATE_TRANSFER 14 /* Receiving .rdb from master */
#define REPL_STATE_CONNECTED 15 /* Connected to master */
/* State of slaves from the POV of the master. Used in client->replstate.
* In SEND_BULK and ONLINE state the slave receives new updates
* in its output queue. In the WAIT_BGSAVE states instead the server is waiting
* to start the next background saving in order to send updates to it. */
#define SLAVE_STATE_WAIT_BGSAVE_START 6 /* We need to produce a new RDB file. */
#define SLAVE_STATE_WAIT_BGSAVE_END 7 /* Waiting RDB file creation to finish. */
#define SLAVE_STATE_SEND_BULK 8 /* Sending RDB file to slave. */
#define SLAVE_STATE_ONLINE 9 /* RDB file transmitted, sending just updates. */
```

就不再那么啰嗦了，我们主要关注以下点：

- 在PSYNC阶段之前，master主节点都是调用replconfCommand去处理的
- 收到PSYNC命令时，调用syncCommand函数去处理。