配置文件补充

昨天回去想了一想，配置文件还有一个非常重要的知识点没说到位，下面我在给大家补充补充这块，非常重要，非常重要。

```
int main(int argc , char* argv[])
{
    //...
     initServerConfig()
    //...
      loadServerConfig(server.configfile, config_from_stdin, options);
    //...
}
```

在执行loadServerConfig函数之前呢，还需要执行initServerConfig函数，该函数的作用如下：

> `initServerConfig()`函数的作用是将Redis服务器的配置结构体`struct redisServer`的各个成员变量初始化为默认值。该函数在Redis服务器启动时被调用，以确保服务器的配置在开始时被正确设置。在该函数中，所有配置选项的默认值都被设置，例如服务器监听端口、日志文件名、持久化选项等等。

该函数代码我粘贴给大家看看：

```

void initServerConfig(void) {
    int j;
    char *default_bindaddr[CONFIG_DEFAULT_BINDADDR_COUNT] = CONFIG_DEFAULT_BINDADDR;

    initConfigValues();
    updateCachedTime(1);
    getRandomHexChars(server.runid,CONFIG_RUN_ID_SIZE);
    server.runid[CONFIG_RUN_ID_SIZE] = '\0';
    changeReplicationId();
    clearReplicationId2();
    server.hz = CONFIG_DEFAULT_HZ; /* Initialize it ASAP, even if it may get
                                      updated later after loading the config.
```

```
                                        This value may be used
before the server
                                        is initialized. */
    server.timezone = getTimeZone(); /* Initialized by
tzset(). */
    server.configfile = NULL;
    server.executable = NULL;
    server.arch_bits = (sizeof(long) == 8) ? 64 : 32;
    server.bindaddr_count = CONFIG_DEFAULT_BINDADDR_COUNT;
    for (j = 0; j < CONFIG_DEFAULT_BINDADDR_COUNT; j++)
        server.bindaddr[j] = zstrdup(default_bindaddr[j]);
   /*

        //相关参数的初始化，我就不粘贴了


   */

    /* Client output buffer limits */
    for (j = 0; j < CLIENT_TYPE_OBUF_COUNT; j++)
        server.client_obuf_limits[j] =
clientBufferLimitsDefaults[j];

    /* Linux OOM Score config */
    for (j = 0; j < CONFIG_OOM_COUNT; j++)
        server.oom_score_adj_values[j] =
configOOMScoreAdjValuesDefaults[j];

    /* Double constants initialization */
    R_Zero = 0.0;
    R_PosInf = 1.0/R_Zero;
    R_NegInf = -1.0/R_Zero;
    R_Nan = R_Zero/R_Zero;

    /* Command table -- we initialize it here as it is part of
the
     * initial configuration, since command names may be
changed via
     * redis.conf using the rename-command directive. */
    server.commands = dictCreate(&commandTableDictType);
    server.orig_commands = dictCreate(&commandTableDictType);
    populateCommandTable();

    /* Debugging */
    server.watchdog_period = 0;
```

```
53 }
```

1 代码说明-默认绑定

```
1   char *default_bindaddr[CONFIG_DEFAULT_BINDADDR_COUNT] =
CONFIG_DEFAULT_BINDADDR;
```

这行代码定义了一个指针数组`default_bindaddr`，其每个元素都是一个指向字符数组的指针，每个字符数组存储了Redis默认绑定的IP地址。

具体来说，`CONFIG_DEFAULT_BINDADDR`定义在`src/server.h`中，其内容如下：

```
1 #define CONFIG_DEFAULT_BINDADDR_COUNT 2
2 #define CONFIG_DEFAULT_BINDADDR { "*", "-::*" }
```

2 代码说明- initConfigValues

```
1  void initConfigValues() {
2      configs = dictCreate(&sdsHashDictType);
3      dictExpand(configs, sizeof(static_configs) /
sizeof(standardConfig));
4      for (standardConfig *config = static_configs; config->name
!= NULL; config++) {
5          if (config->interface.init) config-
>interface.init(config);
6          /* Add the primary config to the dictionary. */
7          int ret = registerConfigValue(config->name, config,
0);
8          serverAssert(ret);
9
10         /* Aliases are the same as their primary counter
parts, but they
11          * also have a flag indicating they are the alias. */
12         if (config->alias) {
13             int ret = registerConfigValue(config->alias,
config, ALIAS_CONFIG);
14             serverAssert(ret);
15         }
16     }
17 }
```

`configs`是一个字典类型的全局变量，在src/config.c里面定义

```
dict *configs = NULL; /* Runtime config values *
```

> `configs` 是一个 Redis 服务器配置项字典，用于存储 Redis 服务器的各种配置项。在 `initConfigValues()` 函数中，通过调用 `dictCreate()` 函数创建了这个字典，并且使用了 `sizeof()` 函数计算了静态配置项数组的长度，然后调用了 `dictExpand()` 函数进行了字典的扩容。
>
> 在之后的循环中，将静态配置项数组中的每个配置项添加到 `configs` 字典中，以便后续使用。同时，如果配置项具有别名，则还会将别名添加到字典中，并将它们的类型标记为 `ALIAS_CONFIG`。通过这种方式，可以方便地将别名配置项映射到其主配置项。

`initConfigValues()`函数用于初始化Redis的配置参数，并将这些参数加入到Redis的参数字典中，方便后续使用。具体实现过程如下：

1. 首先创建了一个字典结构`configs`，用于存储配置参数。
2. 然后通过`dictExpand()`函数扩展字典的大小，以容纳所有静态配置参数（即在源码中直接定义的参数）。
3. 接着遍历静态配置参数数组`static_configs`，对于每个配置参数，分别执行以下操作：
    1. 如果该参数的接口有初始化函数，则调用该函数进行初始化。
    2. 调用`registerConfigValue()`函数将该参数添加到参数字典中。
    3. 如果该参数有别名，则同样调用`registerConfigValue()`函数将别名也添加到参数字典中，并设置标志位`ALIAS_CONFIG`表示该参数为别名。
4. 遍历完所有静态配置参数后，所有配置参数都已经被添加到了参数字典中，初始化工作完成。

总之，`initConfigValues()`函数主要作用是初始化Redis的配置参数，并将这些参数添加到Redis的参数字典中，方便后续使用。

我给大家截了一张静态数组源码的图，大家可以看一看：

```
standardConfig static_configs[] = {
    /* Bool configs */
    createBoolConfig("rdbchecksum", NULL, IMMUTABLE_CONFIG, server.rdb_checksum, 1, NULL, NULL),
    createBoolConfig("daemonize", NULL, IMMUTABLE_CONFIG, server.daemonize, 0, NULL, NULL),
```

每一个元素的类型都为`standardConfig`类型，其定义如下：

```c
struct standardConfig {
    const char *name; /* The user visible name of this config */
    const char *alias; /* An alias that can also be used for this config */
    unsigned int flags; /* Flags for this specific config */
    typeInterface interface; /* The function pointers that define the type interface */
    typeData data; /* The type specific data exposed used by the interface */
    configType type; /* The type of config this is. */
    void *privdata; /* privdata for this config, for module configs this is a ModuleConfig struct */
};
```

作用为：

> - name：该配置项的名称；
> - alias：该配置项的别名，也可以用来代替该配置项的名称；
> - flags：标志位，用于指示该配置项的特性；
> - interface：一个 typeInterface 结构体，定义了该配置项的操作接口；
> - data：该配置项的数据；
> - type：该配置项的数据类型；
> - privdata：私有数据，用于某些特殊的配置项，比如模块配置。
>
> 该结构体中的字段描述了一个 Redis 的配置项的各种属性，包括名称、别名、数据类型、操作接口等。在 Redis 的配置文件中，配置项就是由这个结构体来定义的。

```c
int ret = registerConfigValue(config->name, config, 0);
```

这行代码调用了`registerConfigValue`函数，将一个配置项注册到Redis的全局配置字典中。其中，`config->name`是配置项的名称，`config`是该配置项对应的`standardConfig`结构体指针，`0`是该配置项的标志。这个函数返回一个整数值，代表配置项是否注册成功。

```c
serverAssert(ret);
#define serverAssert(_e) ((_e)?(void)0 : (_serverAssert(#_e,__FILE__,__LINE__),redis_unreachable()))
```

`serverAssert(ret);`是 Redis 中的断言语句，用于在运行时检查程序逻辑是否正确。如果`ret`的值为 0，也就是注册配置项失败，`serverAssert()`会抛出一个运行时错误并终止程序的执行，用于帮助开发人员快速定位错误并进行修复。

3 代码说明- updateCachedTime(1)

```
void updateCachedTime(int update_daylight_info) {
    const long long us = ustime();
    updateCachedTimeWithUs(update_daylight_info, us);
}
static inline void updateCachedTimeWithUs(int
update_daylight_info, const long long ustime) {
    server.ustime = ustime;
    server.mstime = server.ustime / 1000;
    time_t unixtime = server.mstime / 1000;
    atomicSet(server.unixtime, unixtime);

    /* To get information about daylight saving time, we need
to call
     * localtime_r and cache the result. However calling
localtime_r in this
     * context is safe since we will never fork() while here,
in the main
     * thread. The logging function will call a thread safe
version of
     * localtime that has no locks. */
    if (update_daylight_info) {
        struct tm tm;
        time_t ut = server.unixtime;
        localtime_r(&ut,&tm);
        server.daylight_active = tm.tm_isdst;
    }
}
```

`updateCachedTime(1)`函数是用来更新服务器的内部时间缓存的。该函数接受一个布尔值作为参数，指示是否应该将当前时间设置为服务器的unixtime，如果参数为1，则设置为unixtime。该函数的作用是确保服务器内部的时间缓存始终与系统时间保持同步，以便服务器能够正确地处理各种过期和定时事件。

4 代码说明- getRandomHexChars(server.runid,CONFIG_RUN_ID_SIZE)

```
void getRandomHexChars(char *p, size_t len) {
    char *charset = "0123456789abcdef";
    size_t j;

    getRandomBytes((unsigned char*)p,len);
    for (j = 0; j < len; j++) p[j] = charset[p[j] & 0x0F];
}
#define CONFIG_RUN_ID_SIZE 40
```

`getRandomHexChars(server.runid, CONFIG_RUN_ID_SIZE)` 是 Redis 在初始化时生成一个唯一的运行 ID。它的作用是为了让多个 Redis 实例在进行集群操作时进行区分。

这个函数定义在 `util.c` 文件中。它使用 `getRandomBytes()` 函数生成一个随机数序列，然后将其转换成一个包含 40 个随机十六进制字符的字符串。

`server.runid` 是一个全局变量，表示 Redis 实例的运行 ID。它在 `serverCron()` 函数中被更新，以确保它始终保持最新状态。运行 ID 通常被用于构建集群中节点之间的通信，因为它是唯一的，可以用于标识每个 Redis 实例。

```
  server.runid[CONFIG_RUN_ID_SIZE] = '\0';
```

这行代码的作用是给 `server.runid` 字符数组的末尾添加一个 null 字符，即将字符串的结尾标记为 `'\0'`。这是为了确保 `server.runid` 是一个以 null 字符结尾的 C 字符串，以便可以使用字符串处理函数对其进行操作。如果不添加 null 字符，那么在使用字符串处理函数时会出现未定义行为。

```
  changeReplicationId();
  clearReplicationId2();
```

这两个函数是用于设置Redis复制的ID的。`changeReplicationId()`会生成一个新的ID并设置到`server.runid`中，而`clearReplicationId2()`会将`server.replid2`设置为空字符串。

在Redis复制中，每个主服务器都有一个唯一的复制ID（replid），所有的从服务器会记录它们最后一次成功复制的replid。当从服务器重新连接到主服务器时，它将发送自己的replid，主服务器会检查它是否比它所记录的从服务器最后一次成功复制的replid更新，如果是，主服务器会将缺失的数据同步到从服务器。这种方式可以确保从服务器与主服务器数据的一致性。

5 配置项说明

```
server.hz = CONFIG_DEFAULT_HZ;
```

这行代码是在 Redis 服务器启动时初始化服务器的 hz 值，即 Redis 服务器每秒执行定时器事件的次数，它影响到 Redis 服务器的时间精度。如果在配置文件中没有设置 hz 值，那么就使用默认值 CONFIG_DEFAULT_HZ（100），将其赋值给服务器的 hz 属性。

```
    for (j = 0; j < CONFIG_DEFAULT_BINDADDR_COUNT; j++)
         server.bindaddr[j] = zstrdup(default_bindaddr[j]);
```

这段代码是将默认的绑定地址配置复制到服务器实例的 `server.bindaddr` 数组中，这个数组是一个字符串数组，用于记录 Redis 服务器监听的地址和端口信息。具体来说，这个循环会遍历默认绑定地址数组 `default_bindaddr`，将其中的每个地址字符串复制到 `server.bindaddr` 数组的对应位置上，通过 `zstrdup` 函数来完成字符串的复制操作。

```
 server.ipfd.count = 0;
    server.tlsfd.count = 0;
    server.sofd = -1;
    server.active_expire_enabled = 1;
    server.skip_checksum_validation = 0;
    server.loading = 0;
    server.async_loading = 0;
    server.loading_rdb_used_mem = 0;
    server.aof_state = AOF_OFF;
    server.aof_rewrite_base_size = 0;
    server.aof_rewrite_scheduled = 0;
    server.aof_flush_sleep = 0;
    server.aof_last_fsync = time(NULL);
    server.aof_cur_timestamp = 0;
    atomicSet(server.aof_bio_fsync_status,C_OK);
    server.aof_rewrite_time_last = -1;
    server.aof_rewrite_time_start = -1;
    server.aof_lastbgrewrite_status = C_OK;
    server.aof_delayed_fsync = 0;
    server.aof_fd = -1;
    server.aof_selected_db = -1; /* Make sure the first time
will not match */
    server.aof_flush_postponed_start = 0;
    server.aof_last_incr_size = 0;
    server.active_defrag_running = 0;
    server.notify_keyspace_events = 0;
    server.blocked_clients = 0;
    memset(server.blocked_clients_by_type,0,
           sizeof(server.blocked_clients_by_type));
    server.shutdown_asap = 0;
```

```
    server.shutdown_flags = 0;
    server.shutdown_mstime = 0;
    server.cluster_module_flags = CLUSTER_MODULE_FLAG_NONE;
    server.migrate_cached_sockets =
dictCreate(&migrateCacheDictType);
    server.next_client_id = 1; /* Client IDs, start from 1 .*/
    server.page_size = sysconf(_SC_PAGESIZE);
    server.pause_cron = 0;

    server.latency_tracking_info_percentiles_len = 3;
    server.latency_tracking_info_percentiles =
zmalloc(sizeof(double)*
(server.latency_tracking_info_percentiles_len));
    server.latency_tracking_info_percentiles[0] = 50.0;  /*
p50 */
    server.latency_tracking_info_percentiles[1] = 99.0;  /*
p99 */
    server.latency_tracking_info_percentiles[2] = 99.9;  /*
p999 */
```

- `server.ipfd.count`和`server.tlsfd.count`表示监听套接字的数量，初始化为0；
- `server.sofd`表示持久化进程的文件描述符，初始化为-1；
- `server.active_expire_enabled`表示是否启用过期键检查功能，初始化为1；
- `server.skip_checksum_validation`表示是否跳过对AOF文件和RDB文件的校验和验证，初始化为0；
- `server.loading`和`server.async_loading`表示是否正在加载持久化文件，初始化为0；
- `server.aof_state`表示AOF持久化状态，初始化为AOF_OFF，表示未启用AOF；
- `server.aof_last_fsync`表示上次AOF缓冲区同步到磁盘的时间戳，初始化为当前时间；
- `server.aof_cur_timestamp`表示AOF文件当前的时间戳，初始化为0；
- `server.aof_rewrite_time_last`和`server.aof_rewrite_time_start`分别表示AOF重写上次执行的时间和开始执行的时间，初始化为-1；
- `server.aof_fd`表示AOF文件的文件描述符，初始化为-1；
- `server.active_defrag_running`表示是否正在进行主动内存碎片整理，初始化为0；
- `server.notify_keyspace_events`表示订阅了哪些键空间通知，初始化为0；
- `server.blocked_clients`表示当前阻塞的客户端数量，初始化为0；

- `server.shutdown_asap`表示是否立即关闭服务器，初始化为0；
- `server.shutdown_flags`表示关闭服务器的标志，初始化为0；
- `server.cluster_module_flags`表示集群模块的标志，初始化为CLUSTER_MODULE_FLAG_NONE；
- `server.migrate_cached_sockets`表示缓存的迁移套接字，初始化为一个空字典；
- `server.next_client_id`表示下一个客户端的ID，初始化为1；
- `server.page_size`表示操作系统的内存页大小，初始化为从操作系统获取的值；
- `server.pause_cron`表示是否暂停定时任务，初始化为0；
- `server.latency_tracking_info_percentiles`表示延迟跟踪的百分比值，初始化为50、99和99.9三个值。

```
1  unsigned int lruclock = getLRUClock();
```

`getLRUClock()`是Redis中的一个函数，它返回一个递增的时钟值，用于LRU算法中的时间戳。这个时钟值会在Redis的每个操作中进行更新，用于记录最近一次访问的时间戳。在这段代码中，`lruclock`是一个无符号整数类型的变量，存储的是通过调用`getLRUClock()`函数获取的LRU时钟值。

```
1  resetServerSaveParams();
2
3      appendServerSaveParams(60*60,1);  /* save after 1 hour and
   1 change */
4      appendServerSaveParams(300,100);  /* save after 5 minutes
   and 100 changes */
5      appendServerSaveParams(60,10000); /* save after 1 minute
   and 10000 changes */
```

这段代码是Redis中关于持久化的参数配置，resetServerSaveParams()函数是用来清空Redis服务器当前已有的保存配置信息的。接下来，使用appendServerSaveParams()函数对Redis服务器进行三次持久化配置设置：

1. save after 1 hour and 1 change（1小时1次变更后保存）。
2. save after 5 minutes and 100 changes（5分钟100次变更后保存）。
3. save after 1 minute and 10000 changes（1分钟10000次变更后保存）。

这三个配置参数定义了Redis服务器进行自动持久化的条件。在Redis运行过程中，每当满足其中一个条件时，Redis服务器就会将当前内存中的数据保存到磁盘上，以避免数据在服务器崩溃时的丢失。

```
1  /* Replication related */
2      server.masterhost = NULL;
```

```
    server.masterport = 6379;
    server.master = NULL;
    server.cached_master = NULL;
    server.master_initial_offset = -1;
    server.repl_state = REPL_STATE_NONE;
    server.repl_transfer_tmpfile = NULL;
    server.repl_transfer_fd = -1;
    server.repl_transfer_s = NULL;
    server.repl_syncio_timeout = CONFIG_REPL_SYNCIO_TIMEOUT;
    server.repl_down_since = 0; /* Never connected, repl is
down since EVER. */
    server.master_repl_offset = 0;

    /* Replication partial resync backlog */
    server.repl_backlog = NULL;
    server.repl_no_slaves_since = time(NULL);

    /* Failover related */
    server.failover_end_time = 0;
    server.force_failover = 0;
    server.target_replica_host = NULL;
    server.target_replica_port = 0;
    server.failover_state = NO_FAILOVER;

```

这段代码是Redis的复制（replication）和故障转移（failover）相关的配置。其中：

- `server.masterhost` 和 `server.masterport` 指定了Redis主节点的地址和端口号。
- `server.master` 是一个指向主节点状态的指针，指向`clusterNode`结构体。
- `server.cached_master` 也是一个指向主节点状态的指针，指向`clusterNode`结构体。它在复制过程中被用来缓存主节点的信息，以避免频繁地访问主节点状态。
- `server.master_initial_offset` 指定了从节点初始同步时的偏移量，通常为-1表示从头开始同步。
- `server.repl_state` 记录了复制状态，有`REPL_STATE_NONE`、`REPL_STATE_CONNECT`、`REPL_STATE_CONNECTING`、`REPL_STATE_RECEIVE_PONG`、`REPL_STATE_SEND_AUTH`、`REPL_STATE_RECEIVE_AUTH`、`REPL_STATE_SEND_PORT`、`REPL_STATE_RECEIVE_PORT`、`REPL_STATE_SEND_CAPA`、`REPL_STATE_RECEIVE_CAPA`、`REPL_STATE_SEND_PSYNC`、

`REPL_STATE_RECEIVE_PSYNC`、`REPL_STATE_SEND_FILE`、`REPL_STATE_RECEIVE_FILE`、`REPL_STATE_SEND_BULK`、`REPL_STATE_RECEIVE_BULK`、`REPL_STATE_ONLINE`、`REPL_STATE_CATCHUP`、`REPL_STATE_CONNECTED`、`REPL_STATE_WAIT_BGSAVE_START`、`REPL_STATE_WAIT_BGSAVE_END` 等状态。

- `server.repl_transfer_tmpfile` 用于保存复制过程中传输的临时文件的文件名。
- `server.repl_transfer_fd` 是传输文件的文件描述符。
- `server.repl_transfer_s` 是传输文件的网络连接状态。
- `server.repl_syncio_timeout` 是复制同步操作的超时时间，默认值为5秒。
- `server.repl_down_since` 记录了主节点失效的时间戳，单位是秒。
- `server.master_repl_offset` 是从节点当前的复制偏移量。

在这段代码中，还定义了与故障转移相关的变量，例如`server.failover_end_time`表示故障转移的结束时间戳，`server.failover_state`表示故障转移的状态，`server.target_replica_host`和`server.target_replica_port`是新的主节点的地址和端口号。

```
/* Client output buffer limits */
    for (j = 0; j < CLIENT_TYPE_OBUF_COUNT; j++)
        server.client_obuf_limits[j] =
clientBufferLimitsDefaults[j];

    /* Linux OOM Score config */
    for (j = 0; j < CONFIG_OOM_COUNT; j++)
        server.oom_score_adj_values[j] =
configOOMScoreAdjValuesDefaults[j];

```

这段代码初始化了服务器中客户端输出缓冲区的限制以及Linux OOM（Out-Of-Memory） Score值的配置。

在第一个循环中，使用默认值初始化了客户端输出缓冲区的限制，CLIENT_TYPE_OBUF_COUNT表示不同类型的客户端输出缓冲区，clientBufferLimitsDefaults是默认的客户端输出缓冲区限制。

在第二个循环中，使用默认值初始化了服务器的OOM Score值，OOM Score是Linux内核的一个功能，用于根据系统内存使用情况调整进程的优先级。CONFIG_OOM_COUNT是不同类型的OOM Score的数量，configOOMScoreAdjValuesDefaults是默认的OOM Score值。

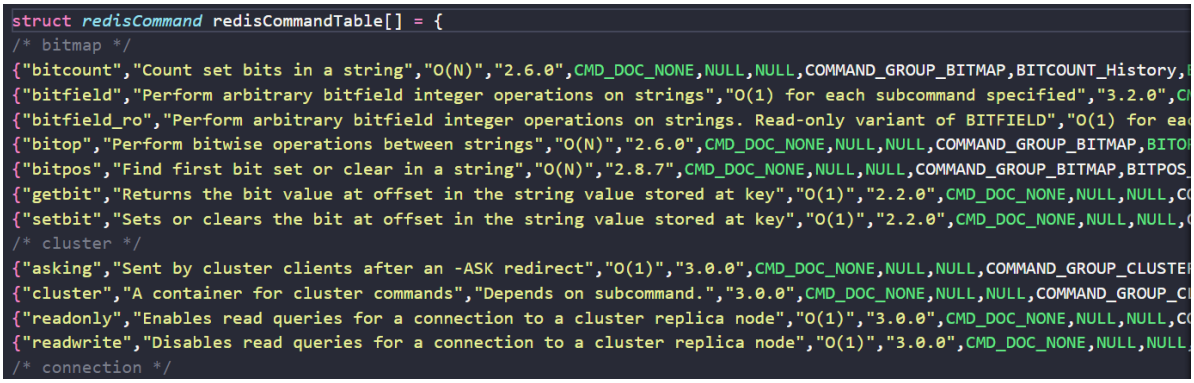
```
  server.commands = dictCreate(&commandTableDictType);
   server.orig_commands = dictCreate(&commandTableDictType);
   populateCommandTable();
```

这段代码创建了两个字典结构 `server.commands` 和 `server.orig_commands`，用于存储 Redis 命令。然后通过调用 `populateCommandTable()` 函数来填充 Redis 命令表，将命令添加到字典中。其中，命令表的定义是通过 `struct redisCommand` 结构体实现的，结构体中包含命令名、命令函数等信息

6 填充redis命令表

```
void populateCommandTable(void) {
    int j;
    struct redisCommand *c;

    for (j = 0;; j++) {
        c = redisCommandTable + j;
        if (c->declared_name == NULL)
            break;

        int retval1, retval2;

        c->fullname = sdsnew(c->declared_name);
        if (populateCommandStructure(c) == C_ERR)
            continue;

        retval1 = dictAdd(server.commands, sdsdup(c->fullname), c);
        /* Populate an additional dictionary that will be unaffected
         * by rename-command statements in redis.conf. */
        retval2 = dictAdd(server.orig_commands, sdsdup(c->fullname), c);
        serverAssert(retval1 == DICT_OK && retval2 == DICT_OK);
    }
}

```

命令表部分截图如下：



`redisCommand`结构如下所示：

```
1  struct redisCommand {
2      /* Declarative data */
3      const char *declared_name; /* A string representing the
   command declared_name.
4                                    * It is a const char * for
   native commands and SDS for module commands. */
5      const char *summary; /* Summary of the command (optional).
   */
6      const char *complexity; /* Complexity description
   (optional). */
7      const char *since; /* Debut version of the command
   (optional). */
8      int doc_flags; /* Flags for documentation (see CMD_DOC_*).
   */
9      const char *replaced_by; /* In case the command is
   deprecated, this is the successor command. */
10     const char *deprecated_since; /* In case the command is
   deprecated, when did it happen? */
11     redisCommandGroup group; /* Command group */
12     commandHistory *history; /* History of the command */
13     const char **tips; /* An array of strings that are meant
   to be tips for clients/proxies regarding this command */
14     redisCommandProc *proc; /* Command implementation */
15     int arity; /* Number of arguments, it is possible to use -
   N to say >= N */
16     uint64_t flags; /* Command flags, see CMD_*. */
17     uint64_t acl_categories; /* ACl categories, see
   ACL_CATEGORY_*. */
18     keySpec key_specs_static[STATIC_KEY_SPECS_NUM]; /* Key
   specs. See keySpec */
19     /* Use a function to determine keys arguments in a command
   line.
20      * Used for Redis Cluster redirect (may be NULL) */
21     redisGetKeysProc *getkeys_proc;
```

```
    /* Array of subcommands (may be NULL) */
    struct redisCommand *subcommands;
    /* Array of arguments (may be NULL) */
    struct redisCommandArg *args;

    /* Runtime populated data */
    long long microseconds, calls, rejected_calls,
failed_calls;
    int id;     /* Command ID. This is a progressive ID
starting from 0 that
                   is assigned at runtime, and is used in
order to check
                   ACLs. A connection is able to execute a
given command if
                   the user associated to the connection has
this command
                   bit set in the bitmap of allowed commands.
*/
    sds fullname; /* A SDS string representing the command
fullname. */
    struct hdr_histogram* latency_histogram; /*points to the
command latency command histogram (unit of time nanosecond) */
    keySpec *key_specs;
    keySpec legacy_range_key_spec; /* The legacy
(first,last,step) key spec is
                                    * still maintained (if
applicable) so that
                                    * we can still support
the reply format of
                                    * COMMAND INFO and
COMMAND GETKEYS */
    int num_args;
    int num_history;
    int num_tips;
    int key_specs_num;
    int key_specs_max;
    dict *subcommands_dict; /* A dictionary that holds the
subcommands, the key is the subcommand sds name
                             * (not the fullname), and the
value is the redisCommand structure pointer. */
    struct redisCommand *parent;
    struct RedisModuleCommand *module_cmd; /* A pointer to the
module command data (NULL if native command) */
};
```

这是一个 Redis 命令结构体的定义，它包含了 Redis 命令的各种属性和信息。具体来说，这个结构体包含了以下属性：

- declared_name：一个字符串，表示该命令的名称，对于原生命令来说是 const char * 类型，对于模块命令来说是 SDS 类型。
- summary：命令的摘要信息（可选）。
- complexity：命令的复杂度描述（可选）。
- since：命令的起始版本号（可选）。
- doc_flags：用于文档的标志位（见 CMD_DOC_*）。
- replaced_by：在命令被弃用的情况下，表示其后继命令。
- deprecated_since：在命令被弃用的情况下，表示它被弃用的时间。
- group：命令所属的组别。
- history：命令的历史记录。
- tips：一个字符串数组，用于提示客户端或代理有关该命令的信息。
- proc：命令的实现函数。
- arity：命令的参数个数，可以使用 -N 来表示 >= N。
- flags：命令的标志位，见 CMD_*。
- acl_categories：ACL 类别，见 ACL_CATEGORY_*。
- key_specs_static：keySpec 类型的数组，用于描述键值对的信息，详见 keySpec 结构体的定义。
- getkeys_proc：一个函数指针，用于在命令行中确定键参数，用于 Redis Cluster 重定向（可能为空）。
- subcommands：子命令的结构体指针数组（可能为空）。
- args：命令的参数结构体指针数组（可能为空）。

除此之外，这个结构体还包含了一些运行时需要的信息，例如：

- microseconds：命令的总执行时间（单位为微秒）。
- calls：命令被调用的次数。
- rejected_calls：命令被拒绝执行的次数。
- failed_calls：命令执行失败的次数。
- id：命令的 ID，用于检查 ACL。
- fullname：命令的完整名称（SDS 字符串类型）。
- latency_histogram：指向命令延迟直方图的指针（时间单位为纳秒）。
- key_specs：keySpec 类型的指针数组，用于描述键值对的信息。
- legacy_range_key_spec：用于支持 COMMAND INFO 和 COMMAND GETKEYS 的回复格式。
- num_args：参数个数。
- num_history：历史记录个数。
- num_tips：提示个数。
- key_specs_num：keySpec 的数量。
- key_specs_max：keySpec 的最大数量。

- subcommands_dict：一个字典，用于存储子命令的 SDS 名称和 redisCommand 结构体指针。
- parent：父命令的结构体指针。
- module_cmd：指向模块命令数据的指针（如果是原生命令则为 NULL）。

```
1   if (populateCommandStructure(c) == C_ERR)
```

`populateCommandStructure(c)` 函数的作用是初始化 Redis 命令结构体 `redisCommand`，该结构体定义了 Redis 支持的所有命令的信息，包括命令名、参数个数、参数类型、命令实现函数、命令标识等。Redis 的命令注册是在 Redis 启动时进行的，这个过程会解析 Redis 配置文件中定义的命令以及 Redis 模块中定义的命令，并将它们注册到 Redis 服务器的命令表中，使得 Redis 服务器能够响应客户端发来的命令请求。

`populateCommandStructure(c)` 函数接受一个 `redisCommand` 结构体指针 `c` 作为参数，根据 `c` 中声明的命令名、命令参数、命令实现函数等信息，初始化 `redisCommand` 结构体，这些信息可以在 Redis 启动时从配置文件和模块中读取。函数执行成功时返回 `C_OK`，否则返回 `C_ERR`。

该函数是 Redis 命令注册过程中的一个重要环节，确保 Redis 服务器能够正确地识别并响应客户端的请求。

7 redis看们狗

"Watchdog"一词源于狗的领域。在过去，人们常常训练一些狗来守卫家园、牲畜或其他财产。这些狗通常是非常警觉和忠诚的，可以保护主人的财产免受入侵和损坏。因此，当计算机系统出现故障或意外关闭时，类似于这些狗的软件程序会被称为"看门狗"，**以暗示它们的任务是监视系统并在必要时采取行动来保护系统的完整性和稳定性。**

```
1   server.watchdog_period = 0;
```

这行代码的作用是将Redis服务器的看门狗（watchdog）定期检查的时间间隔设置为0。看门狗是Redis中的一个后台线程，用于检测Redis服务器是否处于正常运行状态。当看门狗检测到Redis服务器未响应时，它将尝试自动重启服务器，以保证Redis服务器的可用性。

通常情况下，看门狗定期检查的时间间隔是1秒钟。将该时间间隔设置为0意味着禁用了看门狗的自动重启功能。这通常在某些情况下很有用，比如在进行调试时，我们可能会频繁地重启Redis服务器，此时禁用看门狗可以避免不必要的麻烦。但是需要注意的是，禁用看门狗可能会对Redis服务器的可用性产生潜在的风险，因此在生产环境下不应该将看门狗完全禁用。