Redis服务器启动过程源码分析

Redis服务器的启动逻辑由`server.c/main()`函数实现，当用户调用`redis-server`程序启动Redis服务器的时候，它就会被调用：

```
int main(int argc, char **argv) {
    // ...
}
```

为了将整个服务器设置就绪并能够随时接收命令，`main()`函数需要做非常多的工作，下面将对它们一一进行介绍。

首先，服务器需要调用一系列函数，为其设置时区和时间，并初始化之后将要用到的各类哈希算法和函数：

```
// 设置时区以及内存超限处理器
tzset(); /* Populates 'timezone' global. */
zmalloc_set_oom_handler(redisOutOfMemoryHandler);

// 初始化服务器时间以及各类哈希函数的随机生成器
/* To achieve entropy, in case of containers, their time() and getpid() can
 * be the same. But value of tv_usec is fast enough to make the difference */
gettimeofday(&tv,NULL);
srand(time(NULL)^getpid()^tv.tv_usec);
srandom(time(NULL)^getpid()^tv.tv_usec);
init_genrand64(((long long) tv.tv_sec * 1000000 + tv.tv_usec) ^ getpid());
crc64_init();

// 初始化哈希算法种子
uint8_t hashseed[16];
getRandomBytes(hashseed,sizeof(hashseed));
dictSetHashFunctionSeed(hashseed);
```

接着，服务器会记录启动时的`umask`值和文件名以便在之后使用：

```
// 记录mask值，以便在将来创建文件等场景中使用
/* Store umask value. Because umask(2) only offers a set-and-get API we have
 * to reset it and restore it back. We do this early to avoid a potential
 * race condition with threads that could be creating files or directories.
 */
umask(server.umask = umask(0777));

// 记录程序的执行名
char *exec_name = strrchr(argv[0], '/');
if (exec_name == NULL) exec_name = argv[0];
```

然后服务器会调用`checkForSentinelMode()`函数，检查自己是否以哨兵模式启动：

```
// 检查服务器是否以哨兵模式启动
server.sentinel_mode = checkForSentinelMode(argc,argv, exec_name);
```

`checkForSentinel()`函数的实现非常简单，只需要检查服务器启动时的名字是否为`redis-sentinel`又或者启动服务器时给定的参数是否包含了`--sentinel`即可：

```c
// 检查服务器是否以哨兵模式启动，是的话返回1，否则返回0
int checkForSentinelMode(int argc, char **argv, char *exec_name) {
    // 服务器以"redis-sentinel"的名字启动
    if (strstr(exec_name,"redis-sentinel") != NULL) return 1;

    // 启动服务器时给定的参数中包含了"--sentinel"
    for (int j = 1; j < argc; j++)
        if (!strcmp(argv[j],"--sentinel")) return 1;

    // 服务器未以哨兵模式启动
    return 0;
}
```

再然后，服务器会调用`initServerConfig()`函数，对服务器的各项属性进行初始化：

```c
initServerConfig();
```

这个函数的主要作用，就是为服务器结构`server.h/redisServer`中的各项属性初始化一个默认值：

```c
void initServerConfig(void) {
    // ...
    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.dbg_assert_keysizes = 0; /* Disabled by default */
    server.bindaddr_count = CONFIG_DEFAULT_BINDADDR_COUNT;
    for (j = 0; j < CONFIG_DEFAULT_BINDADDR_COUNT; j++)
        server.bindaddr[j] = zstrdup(default_bindaddr[j]);
    memset(server.listeners, 0x00, sizeof(server.listeners));
    server.active_expire_enabled = 1;
    server.allow_access_expired = 0;
    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) * 1000;
    server.aof_cur_timestamp = 0;
    // ...
}
```

之后，服务器会调用`acl.c/ACLInit()`、`module.c/moduleInitModulesSystem()`和`connection.c/connTypeInitialize()`三个函数，分别对服务器的ACL子系统、模块子系统和网络连接进行初始化：

```
ACLInit(); /* The ACL subsystem must be initialized ASAP because the
              basic networking code and client creation depends on it. */
moduleInitModulesSystem();
connTypeInitialize();
```

接着，服务器会将启动时的绝对路径和各项给定参数分别记录到服务器结构的`executable`属性和`exec_argv`属性中，以便将来使用：

```
server.executable = getAbsolutePath(argv[0]);
server.exec_argv = zmalloc(sizeof(char*)*(argc+1));
server.exec_argv[argc] = NULL;
for (j = 0; j < argc; j++) server.exec_argv[j] = zstrdup(argv[j]);
```

之后，如果服务器是以哨兵模式启动的话，那么服务器将调用`sentinel.c/initSentinelConfig()`函数和`sentinel.c/initSentinel()`函数，为服务器结构中哨兵相关的各项属性设置初始值：

```
if (server.sentinel_mode) {
    initSentinelConfig();
    initSentinel();
}
```

以下是这两个被调用函数的定义：

```
// 这个函数会使用哨兵特定的默认值覆盖一些普通模式下的配置变量
void initSentinelConfig(void) {
    server.port = REDIS_SENTINEL_PORT;
    server.protected_mode = 0; /* Sentinel must be exposed. */
}

// 执行哨兵模式初始化，为服务器结构中哨兵相关的属性设置初始值
void initSentinel(void) {
    /* Initialize various data structures. */
    sentinel.current_epoch = 0;
    sentinel.masters = dictCreate(&instancesDictType);
    sentinel.tilt = 0;
    sentinel.tilt_start_time = 0;
    sentinel.total_tilt = 0;
    sentinel.previous_time = mstime();
    sentinel.running_scripts = 0;
    sentinel.scripts_queue = listCreate();
    sentinel.announce_ip = NULL;
    sentinel.announce_port = 0;
    sentinel.simfailure_flags = SENTINEL_SIMFAILURE_NONE;
    sentinel.deny_scripts_reconfig = SENTINEL_DEFAULT_DENY_SCRIPTS_RECONFIG;
    sentinel.sentinel_auth_pass = NULL;
    sentinel.sentinel_auth_user = NULL;
    sentinel.resolve_hostnames = SENTINEL_DEFAULT_RESOLVE_HOSTNAMES;
    sentinel.announce_hostnames = SENTINEL_DEFAULT_ANNOUNCE_HOSTNAMES;
    memset(sentinel.myid,0,sizeof(sentinel.myid));
    server.sentinel_config = NULL;
}
```

此后，由于检查Redis持久化文件完整性的`redis-check-rdb`程序和`redis-check-aof`程序都跟Redis哨兵一样，是Redis服务器的一种特殊运行模式，所以服务器在启动的时候也会检查自己是否以该模式运行——如果是的话，那么就会执行相应的文件完整性检查：

```
if (strstr(exec_name,"redis-check-rdb") != NULL)
    // 执行RDB文件完整性检查
    redis_check_rdb_main(argc,argv,NULL);
else if (strstr(exec_name,"redis-check-aof") != NULL)
    // 执行AOF文件完整性检查
    redis_check_aof_main(argc,argv);
```

之后，服务器会解析并载入启动时通过配置文件、配置参数和标准输入三种形式给定的配置选项：

```
// 解析命令行参数，获取启动服务器时给定的配置选项
if (argc >= 2) {
    j = 1; /* First option to parse in argv[] */
    sds options = sdsempty();

    // 处理--help、--version等特殊选项
    if (strcmp(argv[1], "-v") == 0 ||
        strcmp(argv[1], "--version") == 0)
    {
        // ...
    }
    if (strcmp(argv[1], "--help") == 0 ||
        strcmp(argv[1], "-h") == 0) usage();
    if (strcmp(argv[1], "--test-memory") == 0) {
        // ...
    } if (strcmp(argv[1], "--check-system") == 0) {
        exit(syscheck() ? 0 : 1);
    }

    // 获取给定的配置文件
    if (argv[1][0] != '-') {
        /* Replace the config file in server.exec_argv with its absolute path. */
        server.configfile = getAbsolutePath(argv[1]);
        zfree(server.exec_argv[1]);
        server.exec_argv[1] = zstrdup(server.configfile);
        j = 2; // Skip this arg when parsing options
    }

    // 获取给定的配置选项
    sds *argv_tmp;
    int argc_tmp;
    int handled_last_config_arg = 1;
    while(j < argc) {
        /* Either first or last argument - Should we read config from stdin? */
        // 判断是否需要从标准输入读入配置选项
        if (argv[j][0] == '-' && argv[j][1] == '\0' && (j == 1 || j == argc-1)) {
            config_from_stdin = 1;
        }
        // 所有给定的配置选项都会被解析并在概念上拼接至给定的配置文件内容之后。
        // 比如给定--port 6380将生成字符串"port 6380\n"，并在配置文件和标准输入之后被解析。
        else if (handled_last_config_arg && argv[j][0] == '-' && argv[j][1] == '-') {
            // ...
        } else {
            // 将解析所得的配置选项追加至options变量（SDS）中
            options = sdscatrepr(options,argv[j],strlen(argv[j]));
            options = sdscat(options," ");
            handled_last_config_arg = 1;
        }
        j++;
    }

    // 载入服务器配置选项
    loadServerConfig(server.configfile, config_from_stdin, options);
    // 在以哨兵模式启动时，也载入哨兵相关的配置选项
    if (server.sentinel_mode) loadSentinelConfigFromQueue();
    // 配置选项解析和载入完毕，释放记录配置选项的options变量
    sdsfree(options);
}
// 在以哨兵模式启动时，检查刚刚载入的哨兵相关配置选项
if (server.sentinel_mode) sentinelCheckConfigFile();
```

然后，服务器会检查正在使用的系统，判断是否需要对特定系统进行设置：

```
#ifdef __linux__
    linuxMemoryWarnings();
    sds err_msg = NULL;
    if (checkXenClocksource(&err_msg) < 0) {
        serverLog(LL_WARNING, "WARNING %s", err_msg);
        sdsfree(err_msg);
    }
#if defined (__arm64__)
    int ret;
    if ((ret = checkLinuxMadvFreeForkBug(&err_msg)) <= 0) {
        if (ret < 0) {
            serverLog(LL_WARNING, "WARNING %s", err_msg);
            sdsfree(err_msg);
        } else
            serverLog(LL_WARNING, "Failed to test the kernel for a bug that could lead to
data corruption during background save. "
                                  "Your system could be affected, please report this
error.");
        if (!checkIgnoreWarning("ARM64-COW-BUG")) {
            serverLog(LL_WARNING,"Redis will now exit to prevent data corruption. "
                                 "Note that it is possible to suppress this warning by
setting the following config: ignore-warnings ARM64-COW-BUG");
            exit(1);
        }
    }
#endif /* __arm64__ */
#endif /* __linux__ */
```

之后，服务器会检查是否需要以守护进程的方式启动，并打印服务器日志：

```
// 检查是否需要以守护进程方式启动
server.supervised = redisIsSupervised(server.supervised_mode);
int background = server.daemonize && !server.supervised;
if (background) daemonize();

// 打印日志
serverLog(LL_NOTICE, "oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo");
serverLog(LL_NOTICE,
    "Redis version=%s, bits=%d, commit=%s, modified=%d, pid=%d, just started",
        REDIS_VERSION,
        (sizeof(long) == 8) ? 64 : 32,
        redisGitSHA1(),
        strtol(redisGitDirty(),NULL,10) > 0,
        (int)getpid());

if (argc == 1) {
    serverLog(LL_WARNING, "Warning: no config file specified, using the default config. In
order to specify a config file use %s /path/to/redis.conf", argv[0]);
} else {
    serverLog(LL_NOTICE, "Configuration loaded");
}
```

至此，启动服务器所需的前期准备工作已经完成，服务器将调用`initServer()`函数以初始化服务器，

```
// 初始化服务器
initServer();
if (background || server.pidfile) createPidFile();
if (server.set_proc_title) redisSetProcTitle(NULL);
// 打印Redis LOGO
redisAsciiArt();
checkTcpBacklogSettings();
```

`initServer()`函数将继续`initServerConfig()`函数的工作，对`redisServer`结构中的多项属性进行设置，并创建服务器所需的各项数据结构——比如为服务器记录所有客户端的`clients`属性创建链表，为存储服务器数据的`db`属性创建并初始化各个数据库，为服务器的事件循环状态`el`属性绑定各项处理器等等：

```
void initServer(void) {
    int j;

    // 设置信号处理器
    signal(SIGHUP, SIG_IGN);
    signal(SIGPIPE, SIG_IGN);
    setupSignalHandlers();
```

```c
    // 初始化线程管理器
    ThreadsManager_init();
    makeThreadKillable();

    if (server.syslog_enabled) {
        openlog(server.syslog_ident, LOG_PID | LOG_NDELAY | LOG_NOWAIT,
            server.syslog_facility);
    }

    /* Initialization after setting defaults from the config system. */
    // 根据配置系统设置的默认值进行初始化
    server.aof_state = server.aof_enabled ? AOF_ON : AOF_OFF;
    server.fsynced_reploff = server.aof_enabled ? 0 : -1;
    server.hz = server.config_hz;
    server.pid = getpid();
    server.in_fork_child = CHILD_TYPE_NONE;
    server.rdb_pipe_read = -1;
    server.rdb_child_exit_pipe = -1;
    server.main_thread_id = pthread_self();
    server.current_client = NULL;
    server.errors = raxNew();
    server.errors_enabled = 1;
    server.execution_nesting = 0;
    server.clients = listCreate();
    server.clients_index = raxNew();
    server.clients_to_close = listCreate();
    server.slaves = listCreate();
    server.monitors = listCreate();
    server.clients_pending_write = listCreate();
    server.clients_pending_read = listCreate();
    server.clients_timeout_table = raxNew();
    server.replication_allowed = 1;
    server.slaveseldb = -1; /* Force to emit the first SELECT command. */
    server.unblocked_clients = listCreate();
    server.ready_keys = listCreate();
    server.tracking_pending_keys = listCreate();
    server.pending_push_messages = listCreate();
    server.clients_waiting_acks = listCreate();
    server.get_ack_from_slaves = 0;
    server.paused_actions = 0;
    memset(server.client_pause_per_purpose, 0,
           sizeof(server.client_pause_per_purpose));
    server.postponed_clients = listCreate();
    server.events_processed_while_blocked = 0;
    server.system_memory_size = zmalloc_get_memory_size();
    server.blocked_last_cron = 0;
    server.blocking_op_nesting = 0;
    server.thp_enabled = 0;
    server.cluster_drop_packet_filter = -1;
    server.reply_buffer_peak_reset_time = REPLY_BUFFER_DEFAULT_PEAK_RESET_TIME;
    server.reply_buffer_resizing_enabled = 1;
    server.client_mem_usage_buckets = NULL;
    resetReplicationBuffer();
    /* Make sure the locale is set on startup based on the config file. */
    // 基于配置文件，设置本地化变量
    if (setlocale(LC_COLLATE,server.locale_collate) == NULL) {
        serverLog(LL_WARNING, "Failed to configure LOCALE for invalid locale name.");
        exit(1);
    }

    // 创建服务器共享对象
    createSharedObjects();

    // 调整可打开文件的上限
    adjustOpenFilesLimit();

    const char *clk_msg = monotonicInit();
    serverLog(LL_NOTICE, "monotonic clock: %s", clk_msg);

    // 创建事件循环状态结构
    server.el = aeCreateEventLoop(server.maxclients+CONFIG_FDSET_INCR);
    if (server.el == NULL) {
        serverLog(LL_WARNING,
            "Failed creating the event loop. Error message: '%s'",
            strerror(errno));
        exit(1);
    }

    // 创建数据库结构
    server.db = zmalloc(sizeof(redisDb)*server.dbnum);

    /* Create the Redis databases, and initialize other internal state. */
    // 初始化各个数据库的内部状态
    int slot_count_bits = 0;
    int flags = KVSTORE_ALLOCATE_DICTS_ON_DEMAND;
    if (server.cluster_enabled) {
        slot_count_bits = CLUSTER_SLOT_MASK_BITS;
        flags |= KVSTORE_FREE_EMPTY_DICTS;
    }
    for (j = 0; j < server.dbnum; j++) {
        server.db[j].keys = kvstoreCreate(&dbDictType, slot_count_bits, flags |
KVSTORE_ALLOC_META_KEYS_HIST);
        server.db[j].expires = kvstoreCreate(&dbExpiresDictType, slot_count_bits, flags);
        server.db[j].subexpires = estoreCreate(&subexpiresBucketsType, slot_count_bits);
        server.db[j].expires_cursor = 0;
        server.db[j].blocking_keys = dictCreate(&keylistDictType);
        server.db[j].blocking_keys_unblock_on_nokey =
dictCreate(&objectKeyPointerValueDictType);
        server.db[j].ready_keys = dictCreate(&objectKeyPointerValueDictType);
        server.db[j].watched_keys = dictCreate(&keylistDictType);
        server.db[j].id = j;
        server.db[j].avg_ttl = 0;
    }

    // 初始化LRU回收算法
    evictionPoolAlloc(); /* Initialize the LRU keys pool. */

    /* Note that server.pubsub_channels was chosen to be a kvstore (with only one dict,
which
     * seems odd) just to make the code cleaner by making it be the same type as
server.pubsubshard_channels
     * (which has to be kvstore), see pubsubtype.serverPubSubChannels */
    server.pubsub_channels = kvstoreCreate(&objToDictDictType, 0,
KVSTORE_ALLOCATE_DICTS_ON_DEMAND);
    server.pubsub_patterns = dictCreate(&objToDictDictType);
    server.pubsubshard_channels = kvstoreCreate(&objToDictDictType, slot_count_bits,
KVSTORE_ALLOCATE_DICTS_ON_DEMAND | KVSTORE_FREE_EMPTY_DICTS);
    server.pubsub_clients = 0;
    server.watching_clients = 0;
    server.cronloops = 0;
    server.in_exec = 0;
    server.busy_module_yield_flags = BUSY_MODULE_YIELD_NONE;
    server.busy_module_yield_reply = NULL;
    server.client_pause_in_transaction = 0;
    server.child_pid = -1;
    server.child_type = CHILD_TYPE_NONE;
    server.rdb_child_type = RDB_CHILD_TYPE_NONE;
    server.rdb_pipe_conns = NULL;
    server.rdb_pipe_numconns = 0;
    server.rdb_pipe_numconns_writing = 0;
    server.rdb_pipe_buff = NULL;
    server.rdb_pipe_bufflen = 0;
    server.rdb_bgsave_scheduled = 0;
    server.child_info_pipe[0] = -1;
    server.child_info_pipe[1] = -1;
    server.child_info_nread = 0;
    server.aof_buf = sdsempty();
    server.lastsave = time(NULL); /* At startup we consider the DB saved. */
    server.lastbgsave_try = 0;    /* At startup we never tried to BGSAVE. */
    server.rdb_save_time_last = -1;
    server.rdb_save_time_start = -1;
    server.rdb_last_load_keys_expired = 0;
    server.rdb_last_load_keys_loaded = 0;
    server.dirty = 0;
    resetServerStats();
    /* A few stats we don't want to reset: server startup time, and peak mem. */
    server.stat_starttime = time(NULL);
    server.stat_peak_memory = 0;
    server.stat_peak_memory_time = server.unixtime;
    server.stat_current_cow_peak = 0;
    server.stat_current_cow_bytes = 0;
    server.stat_current_cow_updated = 0;
    server.stat_current_save_keys_processed = 0;
    server.stat_current_save_keys_total = 0;
    server.stat_rdb_cow_bytes = 0;
    server.stat_aof_cow_bytes = 0;
    server.stat_module_cow_bytes = 0;
    server.stat_module_progress = 0;
    for (int j = 0; j < CLIENT_TYPE_COUNT; j++)
        server.stat_clients_type_memory[j] = 0;
    server.stat_cluster_links_memory = 0;
    server.cron_malloc_stats.zmalloc_used = 0;
    server.cron_malloc_stats.process_rss = 0;
    server.cron_malloc_stats.allocator_allocated = 0;
    server.cron_malloc_stats.allocator_active = 0;
    server.cron_malloc_stats.allocator_resident = 0;
    server.repl_current_sync_attempts = 0;
    server.lastbgsave_status = C_OK;
    server.aof_last_write_status = C_OK;
    server.aof_last_write_errno = 0;
    server.repl_good_slaves_count = 0;
    server.last_sig_received = 0;
    memset(server.io_threads_clients_num, 0, sizeof(server.io_threads_clients_num));
    atomicSetWithSync(server.running, 0);

    /* Initiate acl info struct */
    // 初始化ACL信息结构
    server.acl_info.invalid_cmd_accesses = 0;
    server.acl_info.invalid_key_accesses  = 0;
    server.acl_info.user_auth_failures = 0;
    server.acl_info.invalid_channel_accesses = 0;

    /* Create the timer callback, this is our way to process many background
     * operations incrementally, like clients timeout, eviction of unaccessed
     * expired keys and so forth. */
    // 创建定时器回调，这是服务器处理客户端超时、驱逐过期键等增量式后台操作的方法。
    if (aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL) == AE_ERR) {
        serverPanic("Can't create event loop timers.");
        exit(1);
    }

    /* Register a readable event for the pipe used to awake the event loop
     * from module threads. */
    // 为模块线程唤醒事件循环的管线注册一个可读事件
    if (aeCreateFileEvent(server.el, server.module_pipe[0], AE_READABLE,
        modulePipeReadable,NULL) == AE_ERR) {
            serverPanic(
                "Error registering the readable event for the module pipe.");
    }

    /* Register before and after sleep handlers (note this needs to be done
     * before loading persistence since it is used by processEventsWhileBlocked. */
    // 为休眠前和休眠后分别绑定处理器
    // 由于读取持久化文件需要用到processEventsWhileBlocked()函数，
    // 所以这项工作必须在读取持久化文件之前完成
    aeSetBeforeSleepProc(server.el,beforeSleep);
    aeSetAfterSleepProc(server.el,afterSleep);

    // 设置服务器可用内存上限
    /* 32 bit instances are limited to 4GB of address space, so if there is
     * no explicit limit in the user provided configuration we set a limit
     * at 3 GB using maxmemory with 'noeviction' policy'. This avoids
     * useless crashes of the Redis instance for out of memory. */
    if (server.arch_bits == 32 && server.maxmemory == 0) {
        serverLog(LL_WARNING,"Warning: 32 bit instance detected but no memory limit set.
Setting 3 GB maxmemory limit with 'noeviction' policy now.");
        server.maxmemory = 3072LL*(1024*1024); /* 3 GB */
        server.maxmemory_policy = MAXMEMORY_NO_EVICTION;
    }

    // 初始化Lua脚本环境
    luaEnvInit();
    scriptingInit(1);

    // 初始化服务器端函数环境
    if (functionsInit() == C_ERR) {
        serverPanic("Functions initialization failed, check the server logs.");
        exit(1);
    }

    // 初始化慢查询子系统
    slowlogInit();

    // 初始化延迟检测子系统
    latencyMonitorInit();

    /* Initialize ACL default password if it exists */
    // 为ACL子系统设置默认密码（如果有的话）
    ACLUpdateDefaultUserPassword(server.requirepass);

    applyWatchdogPeriod();

    if (server.maxmemory_clients != 0)
        initServerClientMemUsageBuckets();
}
```

在`initServer()`函数之后，如果服务器是以集群模式或者哨兵模式启动的话，那么还要继续调用其特有的初始化函数以实现初始化：

```
if (server.cluster_enabled) {
    // 集群初始化
    clusterInit();
}
if (!server.sentinel_mode) {
    // 哨兵初始化
    moduleInitModulesSystemLast();
    moduleLoadInternalModules();
    moduleLoadFromQueue();
}
```

之后，服务器还要载入ACL用户，初始化服务器的监听端口，并初始化后台线程：

```
// 载入ACL用户
ACLLoadUsersAtStartup();
// 初始化监听端口
initListeners();
if (server.cluster_enabled) {
    clusterInitLast();
}
// 初始化后台线程
InitServerLast();
```

接下来，服务器还要再判断一次自己是否运行在哨兵模式中，如果是的话就载入哨兵配置，准备启动哨兵，否则的话就从持久化文件中载入数据，然后分配监听器准备接受网络连接：

```
if (!server.sentinel_mode) {
    /* Things not needed when running in Sentinel mode. */
    serverLog(LL_NOTICE,"Server initialized");
    // 载入持久化数据
    aofLoadManifestFromDisk();
    loadDataFromDisk();
    aofOpenIfNeededOnServerStart();
    aofDelHistoryFiles();
    /* While loading data, we delay applying "appendonly" config change.
     * If there was a config change while we were inside loadDataFromDisk()
     * above, we'll apply it here. */
    applyAppendOnlyConfig();

    if (server.cluster_enabled) {
        serverAssert(verifyClusterConfigWithData() == C_OK);
    }

    // 分配监听器，准备好接受连接
    for (j = 0; j < CONN_TYPE_MAX; j++) {
        connListener *listener = &server.listeners[j];
        if (listener->ct == NULL)
            continue;

        serverLog(LL_NOTICE,"Ready to accept connections %s", listener->ct->get_type(NULL));
    }

    if (server.supervised_mode == SUPERVISED_SYSTEMD) {
        if (!server.masterhost) {
            redisCommunicateSystemd("STATUS=Ready to accept connections\n");
        } else {
            redisCommunicateSystemd("STATUS=Ready to accept connections in read-only mode.
Waiting for MASTER <-> REPLICA sync\n");
        }
        redisCommunicateSystemd("READY=1\n");
    }
} else {
    // 载入哨兵配置，准备启动哨兵
    sentinelIsRunning();
    if (server.supervised_mode == SUPERVISED_SYSTEMD) {
        redisCommunicateSystemd("STATUS=Ready to accept connections\n");
        redisCommunicateSystemd("READY=1\n");
    }
}
```

在载入数据或者启动哨兵之后，服务器还会检查`maxmemory`选项的设置，并为服务器进程设置CPU亲和性以及调整OOM评分：

```
// 对可疑的maxmemory设置进行提醒
if (server.maxmemory > 0 && server.maxmemory < 1024*1024) {
    serverLog(LL_WARNING,"WARNING: You specified a maxmemory value that is less than 1MB
(current value is %llu bytes). Are you sure this is what you really want?",
server.maxmemory);
}

// 设置CPU亲和性
redisSetCpuAffinity(server.server_cpulist);
// 调整OOM评分
setOOMScoreAdj(-1);
```

至此，一切已经准备就绪，接下来服务器将调用`ae.c/aeMain()`函数，开启服务器的事件主循环：

```
aeMain(server.el);
```

`aeMain()`函数将接收并处理服务器运行期间发生的一切事件，直到服务器主动或被动退出事件循环为止：

```
void aeMain(aeEventLoop *eventLoop) {
    // 初始化事件状态
    eventLoop->stop = 0;
    // 监听并处理事件，直到主动或被动退出为止
    while (!eventLoop->stop) {
        aeProcessEvents(eventLoop, AE_ALL_EVENTS|
                                   AE_CALL_BEFORE_SLEEP|
                                   AE_CALL_AFTER_SLEEP);
    }
}
```

在服务器正常退出，`aeMain()`函数正常结束的情况下，服务器将继续执行`main()`函数的最后两行代码——它们将清理事件循环相关的内存和数据结构，然后正常退出服务器：

```
// 清理事件循环相关的内存和数据结构
aeDeleteEventLoop(server.el);
// 服务器退出
return 0;
```

至此，整个`main()`函数执行完毕。

黄健宏
2025.8.26