

三 从server.main再次看redis启动流程

今天呢，我们从整体的角度来看看redis的启动过程，前面两章内容已经将redis启动时重要的过程详细的讲了一遍，现在呢，我们从整体的过程来看看redis的启动。

main函数是程序启动的入口，现在呢，我们一步一步的去分析他，挖掘他。

1 变量定义

main函数的前三行

```
1 struct timeval tv;
2     int j;
3     char config_from_stdin = 0;
4
```

这段代码定义了一个名为 `tv` 的 `timeval` 结构体变量和一个 `int` 类型的变量 `j`，以及一个 `char` 类型的变量 `config_from_stdin`，初值为 0。

`timeval` 是 C 语言标准库中的一个结构体，表示时间值，由两个成员组成：`tv_sec` 表示秒数，`tv_usec` 表示微秒数。在这段代码中，我们定义了一个 `timeval` 类型的变量 `tv`，通常用于计算时间差和设置超时等场景。

`int` 类型的变量 `j` 可以用于计数，或者存储整数值。

`char` 类型的变量 `config_from_stdin` 用于标记程序是否需要从标准输入读取配置。该变量的初值为 0，表示程序不需要从标准输入读取配置。当该变量的值为 1 时，表示程序需要从标准输入读取配置。

```
1 NAME
2     timeval - time in seconds and microseconds
3
4 LIBRARY
5     Standard C library (libc)
6
7 SYNOPSIS
8     #include <sys/time.h>
9
10    struct timeval {
11        time_t      tv_sec;    /* Seconds */
```

```

12         suseconds_t  tv_usec;  /* Microseconds */
13     };
14
15     DESCRIPTION
16         Describes times in seconds and microseconds.
17
18

```

看到这里，你是不是想起前面读取配置的三种方式了，其中是否从终端中读取就在此处定义了。

2 预编译指令-条件测试

```

1  #ifdef REDIS_TEST
2      if (argc >= 3 && !strcasecmp(argv[1], "test")) {
3          int flags = 0;
4          for (j = 3; j < argc; j++) {
5              char *arg = argv[j];
6              if (!strcasecmp(arg, "--accurate")) flags |=
REDIS_TEST_ACCURATE;
7              else if (!strcasecmp(arg, "--large-memory")) flags
|= REDIS_TEST_LARGE_MEMORY;
8          }
9
10         if (!strcasecmp(argv[2], "all")) {
11             int numtests = sizeof(redisTests)/sizeof(struct
redisTest);
12             for (j = 0; j < numtests; j++) {
13                 redisTests[j].failed =
(redisTests[j].proc(argc,argv,flags) != 0);
14             }
15
16             /* Report tests result */
17             int failed_num = 0;
18             for (j = 0; j < numtests; j++) {
19                 if (redisTests[j].failed) {
20                     failed_num++;
21                     printf("[failed] Test - %s\n",
redisTests[j].name);
22                 } else {
23                     printf("[ok] Test - %s\n",
redisTests[j].name);
24                 }
25             }
26

```

```

27         printf("%d tests, %d passed, %d failed\n",
numtests,
28             numtests-failed_num, failed_num);
29
30         return failed_num == 0 ? 0 : 1;
31     } else {
32         redisTestProc *proc = getTestProcByName(argv[2]);
33         if (!proc) return -1; /* test not found */
34         return proc(argc,argv,flags);
35     }
36
37     return 0;
38 }
39 #endif

```

这段代码是一段预编译指令，主要用于判断当前是否处于测试模式下。

`#ifdef` 指令是一个条件编译指令，用于判断某个宏是否已经被定义过。在这里，`REDIS_TEST` 宏是否被定义过将决定 `#ifdef REDIS_TEST` 到 `#endif` 之间的代码段是否需要被编译。

如果 `REDIS_TEST` 宏已经被定义过，则该代码段将被编译。否则，该代码段将被忽略。

条件编译指令常用于编译时针对不同情况进行不同处理，例如调试模式和发布模式下的代码处理不同，或者针对不同平台进行处理等。在 Redis 中，测试模式下的代码与正式发布版本的代码可能存在一些差异，因此需要使用条件编译指令进行区分。

条件测试这个模块我们放到后面进行讲解。

3 预编译指令-进程标题

```

1  #ifdef INIT_SETPROCTITLE_REPLACEMENT
2      spt_init(argc, argv);
3  #endif

```

在这里，`#ifdef INIT_SETPROCTITLE_REPLACEMENT` 到 `#endif` 之间的代码段将会被编译，如果在编译 Redis 时定义了 `INIT_SETPROCTITLE_REPLACEMENT` 宏。该宏在 Redis 的 Makefile 中定义，用于告知编译器使用一个自定义的进程标题设置函数 `spt_init()`。

进程标题是指进程在操作系统中显示的名称，通常用于标识进程的用途。在 Redis 中，进程标题默认为 Redis 的运行参数（如端口号、工作目录等）的组合，但是在某些情况下（如在服务端部署时），我们可能希望进程标题能够更直观地反映 Redis 的用途，这时可以使用自定义的进程标题设置函数来实现。

`INIT_SETPROCTITLE_REPLACEMENT` 宏的定义与具体实现有关，可以参考 Redis 的源代码和 Makefile 文件来了解。在这里，我们假设该宏定义了一个值为 1 的整数常量，表示需要启用进程标题的自定义设置。如果该宏被定义，那么就会调用 `spt_init()` 函数来初始化进程标题，该函数的参数为 `argc` 和 `argv`，表示命令行参数的数量和内容。

4 设置程序本地信息

```
1 | setlocale(LC_COLLATE, "");
```

这行代码使用了 C 标准库中的 `setlocale` 函数，用于设置程序的本地化 (locale) 信息。

`locale` 是一个 C 语言标准库中的概念，用于描述一组与特定地区和语言相关的参数和规则，例如时间格式、货币符号、字符编码等等。程序的本地化信息通常由操作系统决定，但是程序也可以通过调用 `setlocale` 函数来主动设置本地化信息，以便更好地适应不同的用户环境。

在这里，`setlocale` 函数的第一个参数是 `LC_COLLATE`，表示需要设置的本地化分类信息。`LC_COLLATE` 是一个用于排序和比较字符串的本地化分类，用于定义字符排序的规则。当设置 `LC_COLLATE` 为一个空字符串时，表示使用默认的本地化分类信息，这通常是指操作系统的默认设置。

该函数的第二个参数为一个空字符串，表示使用默认的本地化信息。

在 Redis 中，该代码的作用是设置程序的本地化信息，以便更好地适应不同的语言和地区。这在 Redis 中尤其重要，因为 Redis 是一个全球性的软件，需要支持不同的语言和字符编码。

该函数接受两个参数，分别是本地化分类信息 `category` 和需要设置的本地化信息 `locale`。`category` 表示需要设置的本地化分类信息，它可以是以下常量之一：

```
SYNOPSIS
#include <locale.h>

char *setlocale(int category, const char *locale);

DESCRIPTION
The setlocale() function is used to set or query the program's current locale.

If locale is not NULL, the program's current locale is modified according to the arguments. The argument category determines which parts of the program's current locale should be modified.

Category      Governs
LC_ALL         All of the locale
LC_ADDRESS     Formatting of addresses and geography-related items (*)
LC_COLLATE     String collation
LC_CTYPE       Character classification
LC_IDENTIFICATION Metadata describing the locale (*)
LC_MEASUREMENT Settings related to measurements (metric versus US customary) (*)
LC_MESSAGES    Localizable natural-language messages
LC_MONETARY    Formatting of monetary values
LC_NAME        Formatting of salutations for persons (*)
LC_NUMERIC     Formatting of nonmonetary numeric values
LC_PAPER       Settings related to the standard paper size (*)
LC_TELEPHONE   Formats to be used with telephone services (*)
LC_TIME        Formatting of date and time values
```

```
1 #include <locale.h>
2
3 char *setlocale(int category, const char *locale);
```

- `LC_ALL`：表示设置所有本地化分类信息。
- `LC_COLLATE`：表示设置字符排序和比较的本地化分类信息。
- `LC_CTYPE`：表示设置字符分类和转换的本地化分类信息。
- `LC_MONETARY`：表示设置货币格式的本地化分类信息。
- `LC_NUMERIC`：表示设置数字格式的本地化分类信息。
- `LC_TIME`：表示设置时间格式的本地化分类信息。

`locale` 表示需要设置的本地化信息，它的格式通常是一个字符串，用于指定语言、地区、字符编码等信息。例如，`"en_US.UTF-8"` 表示英语语言、美国地区、UTF-8 编码。

`setlocale` 函数返回一个字符串指针，指向当前的本地化信息字符串。如果设置本地化信息成功，返回值将与第二个参数 `locale` 相同，否则返回一个空指针。

在程序中调用 `setlocale` 函数可以设置本地化信息，以便更好地适应不同的语言和地区。例如，如果程序需要处理多语言的用户界面，就可以根据用户的语言偏好来设置本地化信息，以便正确地显示日期、时间、货币符号等内容。

5 初始化时区信息

```
1 tzset(); /* Populates 'timezone' global. */.
```

`tzset()` 函数是 C 标准库中的一个函数，用于初始化时区信息。

在 Unix/Linux 系统中，时间和时区是密切相关的。当系统初始化时，需要读取环境变量 `TZ`，以确定当前所在时区。`tzset()` 函数的作用就是根据 `TZ` 环境变量的值来设置时区信息，并将相应的信息保存到全局变量 `timezone` 和 `daylight` 中。

`timezone` 是一个整型变量，表示当前时区与 UTC（协调世界时）的时差，以秒为单位。如果当前时区比 UTC 早，那么 `timezone` 将是一个正值；如果当前时区比 UTC 晚，那么 `timezone` 将是一个负值。

`daylight` 是一个布尔型变量，表示当前时区是否使用夏令时。如果当前时区使用夏令时，那么 `daylight` 的值为非零（通常为 1）；否则 `daylight` 的值为零。

在调用 `tzset()` 函数之后，程序就可以通过访问全局变量 `timezone` 和 `daylight` 来获取当前的时区信息了。需要注意的是，`tzset()` 函数只需要在程序启动时调用一次即可，通常不需要重复调用。

6 内存分配失败回调函数

```
1 zmalloc_set_oom_handler(redisOutOfMemoryHandler);
2 void redisOutOfMemoryHandler(size_t allocation_size) {
3     serverLog(LL_WARNING, "Out of Memory allocating %zu bytes!",
4         allocation_size);
5     serverPanic("Redis aborting for OUT OF MEMORY. Allocating
6         %zu bytes!",
7         allocation_size);
8 }
```

`zmalloc_set_oom_handler` 函数是 Redis 中一个用于处理内存分配失败的回调函数，它用于设置内存分配失败时的处理函数。

在 Redis 中，内存分配失败时会调用 `zmalloc` 函数，如果分配失败，就会触发 Redis 的 out of memory (OOM) 事件。默认情况下，Redis 的 OOM 事件处理方式是直接调用 `exit(1)` 函数，退出 Redis 进程。但是，如果你希望在发生内存分配失败时执行一些特定的操作，例如记录日志或者释放一些资源，就可以使用 `zmalloc_set_oom_handler` 函数来设置自定义的处理函数。

例如，`zmalloc_set_oom_handler(redisOutOfMemoryHandler)` 的作用是将 `redisOutOfMemoryHandler` 函数注册为 Redis 内存分配失败时的处理函数。如果 Redis 在内存分配时遇到问题，将会自动调用该处理函数，并执行其中的逻辑。这样，你就可以在 Redis 发生内存分配失败时进行特定的操作了。

```
1 static void (*zmalloc_oom_handler)(size_t) =  
    zmalloc_default_oom;  
2 static void zmalloc_default_oom(size_t size) {  
3     fprintf(stderr, "zmalloc: Out of memory trying to allocate  
    %zu bytes\n",  
4         size);  
5     fflush(stderr);  
6     abort();  
7 }
```

```
1 void zmalloc_set_oom_handler(void (*oom_handler)(size_t)) {  
2     zmalloc_oom_handler = oom_handler;  
3 }
```

Redis 实现在内存分配失败时能够执行错误处理函数的关键在于两个方面：

1. 使用了 `zmalloc` 函数库作为 Redis 的内存分配器：Redis 使用 `zmalloc` 作为内存分配器而不是 C 语言标准库中的 `malloc` 函数。`zmalloc` 函数库是一个基于 `malloc` 的封装，它在 `malloc` 的基础上添加了一些特性，例如内存分配跟踪、内存对齐等。同时，`zmalloc` 还提供了 `zmalloc_set_oom_handler` 函数，它能够让开发者注册一个函数，当 Redis 在内存分配失败时会调用该函数。
2. 在 `zmalloc` 函数库中，重写了 `malloc` 和 `free` 函数：`zmalloc` 函数库重写了 C 语言标准库中的 `malloc` 和 `free` 函数，以实现自己的内存分配器。在 `zmalloc` 中，重写的 `malloc` 函数在内存分配失败时不是返回 `NULL`，而是调用了错误处理函数。而且在 `zmalloc` 中还提供了一个 `zmalloc_oom` 函数，**它会触发 Redis 的 OOM 事件，这个事件将会使 Redis 执行错误处理函数。**

因此，当 Redis 在分配内存时发现内存不足，就会调用错误处理函数。这个错误处理函数可以是开发者自己编写的、记录错误日志、尝试释放内存或者其他操作，以确保 Redis 不会因为内存不足而崩溃。

在 `zmalloc` 函数库中，重写的 `malloc` 函数被称为 `zmalloc` 函数，它是 Redis 中用于分配内存的函数。

`zmalloc` 函数与标准的 `malloc` 函数非常相似，但有一些区别。主要区别如下：

1. `zmalloc` 函数会对分配的内存进行额外的跟踪和处理，例如记录分配的内存大小和地址，内存对齐等。
2. `zmalloc` 函数会在分配内存时检查是否有足够的内存可用，如果没有足够的内存，它会调用 Redis 的错误处理函数，而不是返回 `NULL`。
3. `zmalloc` 函数会维护一个内存池，用于提高分配和释放内存的效率。

4. `zmalloc` 函数还提供了其他一些功能，例如重置内存池、获取内存池的统计信息等。

因此，通过使用 `zmalloc` 函数库，Redis 能够更好地控制内存的分配和释放，并且在内存分配失败时能够执行错误处理函数。

```
1 void *zmalloc(size_t size) {  
2     void *ptr = ztrymalloc_usable(size, NULL);  
3     if (!ptr) zmalloc_oom_handler(size);  
4     return ptr;  
5 }
```

这段代码是 Redis 中的 `zmalloc` 函数的实现。它的主要作用是分配一块指定大小的内存，并返回一个指向该内存块的指针。

具体来说，`zmalloc` 函数首先调用 `ztrymalloc_usable` 函数尝试分配指定大小的内存。如果分配成功，则直接返回指向该内存块的指针。如果分配失败，则调用 `zmalloc_oom_handler` 函数来处理内存不足的情况。

`zmalloc_oom_handler` 函数是 Redis 中的内存不足处理函数。它的主要作用是执行一些错误处理操作，例如打印错误消息、释放一些已分配的内存等。这样可以确保 Redis 不会在内存不足的情况下崩溃，而是能够优雅地处理该问题。

总之，`zmalloc` 函数是 Redis 中用于分配内存的核心函数之一，它确保了 Redis 在分配内存时的安全性和可靠性。

7 初始化随机数发生器

```
1 gettimeofday(&tv, NULL);  
2 srand(time(NULL)^getpid()^tv.tv_usec);  
3 srandom(time(NULL)^getpid()^tv.tv_usec);  
4 init_genrand64(((long long) tv.tv_sec * 1000000 +  
tv.tv_usec) ^ getpid());  
5 crc64_init();
```

这段代码主要用于初始化随机数发生器。具体来说，它执行以下操作：

1. 调用 `gettimeofday` 函数获取当前时间，并存储到 `tv` 结构体中。
2. 使用当前进程的进程 ID、时间戳、以及微秒数作为种子，通过 `srand` 和 `srandom` 函数初始化随机数发生器。
3. 调用 `init_genrand64` 函数，使用当前时间的秒数和微秒数、以及进程 ID 作为种子，初始化一个 64 位随机数发生器。
4. 调用 `crc64_init` 函数初始化 CRC64 校验表。

这些操作的目的是为了在 Redis 运行期间产生各种随机数，以及在需要进行校验和计算时使用 CRC64 校验表。这样可以增加 Redis 的安全性和可靠性，确保 Redis 在各种情况下都能正常运行。

8 设置文件权限

```
1 | umask(server.umask = umask(0777));
```

`umask` 函数用于设置当前进程的文件创建屏蔽字。在 Redis 中，该语句的作用是将当前进程的文件创建屏蔽字设置为 `0777`，并将返回值赋值给 `server.umask` 变量。

`umask` 函数的作用是屏蔽掉进程的某些权限，比如读、写、执行等权限，使其不能被文件创建系统调用所继承。例如，如果文件创建屏蔽字设置为 `022`，则表示屏蔽掉组和其他用户的写权限，使得创建的文件默认权限为 `-rw-r--r--`。

在 Redis 中，通过将文件创建屏蔽字设置为 `0777`，可以确保 Redis 在创建文件时不会受到任何限制，这对于一些需要频繁创建和操作文件的场景非常重要，比如 AOF 持久化和 RDB 持久化等。

9 哈希种子

```
1 | uint8_t hashseed[16];
2 | getRandomBytes(hashseed, sizeof(hashseed));
3 | dictSetHashFunctionSeed(hashseed)
```

在这段代码中，Redis 使用 `getRandomBytes` 函数生成一个 16 字节的随机字节数组 `hashseed`，然后将其作为种子调用 `dictSetHashFunctionSeed` 函数，设置哈希函数的种子。哈希函数的种子是一个常量，它影响到哈希函数的散列结果，用于减少哈希冲突，提高哈希表的效率。

在 Redis 中，哈希表被广泛用于实现各种数据结构，如字典、集合、有序集合等，哈希函数的性能对于 Redis 的性能有很大的影响。为了防止哈希碰撞，Redis 在启动时生成一个随机的种子，并将其用作哈希函数的种子。这样，即使在同一组键值对中，如果键的散列值相同，那么哈希函数也会以不同的方式处理这些键值对，减少哈希碰撞的可能性，提高 Redis 的性能。

10 检测哨兵模式

```

1 char *exec_name = strrchr(argv[0], '/');
2 if (exec_name == NULL) exec_name = argv[0];
3 server.sentinel_mode = checkForSentinelMode(argc,argv,
exec_name);
4

```

1. 通过 `strrchr()` 函数获取程序名 `argv[0]` 中最后一个 `/` 字符之后的字符串（即程序名），并将其保存到 `exec_name` 变量中。
2. 如果程序名中没有 `/` 字符，说明 `argv[0]` 本身就是程序名，将其直接保存到 `exec_name` 变量中。
3. 调用 `checkForSentinelMode()` 函数检查是否以 Sentinel 模式运行，并将检查结果保存到 `server.sentinel_mode` 变量中。

```

1 int checkForSentinelMode(int argc, char **argv, char
*exec_name) {
2     if (strstr(exec_name,"redis-sentinel") != NULL) return 1;
3
4     for (int j = 1; j < argc; j++)
5         if (!strcmp(argv[j],"--sentinel")) return 1;
6     return 0;
7 }

```

具体实现是，首先根据参数 `argv[0]` 中的可执行文件名（不包括路径）判断是否以 `redis-sentinel` 命名，如果是，返回1；否则，遍历命令行参数 `argv`，如果发现 `--sentinel` 参数，返回1，否则返回0。

11 初始化服务器配置

```

1 initServerConfig();

```

熟悉吧，这就是我们前两章讲得配置文件,在这里我就不详细描述了，简单的概述一下

`initServerConfig()` 是Redis中用来初始化服务器配置结构体的函数。在该函数中，会将服务器配置结构体的各个成员变量设置为默认值，然后根据启动参数以及配置文件的设置修改这些默认值。

具体来说，该函数会先将所有的配置项都设置为默认值，然后读取 `redis.conf` 配置文件中的配置项，如果启动参数中存在与配置文件中相同的配置项，则以启动参数中的值为准。最后，根据 `sentinel_mode` 变量的值来决定是否进行哨兵模式下的配置。

12 ACL权限控制模块

```
1 | ACLInit()
```

`ACLInit()` 函数是Redis中用来初始化ACL权限控制模块的函数。在该函数中，会初始化一些数据结构，以支持后续的ACL操作。

具体来说，该函数会初始化一个全局的ACL权限列表，以及多个用于ACL操作的数据结构，如命令表（command table）、用户表（user table）、角色表（role table）等。在初始化过程中，该函数会从配置文件中读取默认的ACL规则，并将其加入到全局的ACL权限列表中。同时，该函数还会将 `defaultUser` 用户和 `defaultPassword` 密码设置为默认的登录凭据，以便在没有指定具体凭据的情况下，使用这些凭据进行登录。

这一块具体的代码还是比较复杂的，在后续中，我会详细的为大家进行讲解。

13 Redis初始化模块系统

```
1 | moduleInitModulesSystem();
```

`moduleInitModulesSystem()` 是 Redis 5.0 引入的新函数，它的作用是初始化 Redis 的模块系统。

Redis 模块系统允许开发者通过动态加载模块，扩展 Redis 的功能。模块系统允许开发者通过 Redis 提供的 API 进行模块开发，同时模块也可以在 Redis 运行时加载、卸载，而无需停止 Redis 服务。

`moduleInitModulesSystem()` 函数在 Redis 启动时被调用，用于初始化模块系统。这个函数会在 Redis 的服务器状态中创建一个模块列表，然后遍历加载所有已经在 Redis 配置文件中配置的模块。

如果开发者希望编写自己的 Redis 模块，可以通过调用 Redis 模块 API 进行编写，并将编写好的模块编译成动态库文件。在 Redis 启动时，将这些动态库文件加载到 Redis 的模块系统中，就可以在 Redis 中使用自己编写的模块。

14 TLS层初始化

```
1 | tlsInit();
```

`tlsInit()` 函数是Redis服务器TLS（传输层安全性）支持的一部分。它在服务器初始化期间调用，用于设置TLS上下文。

在以下是TLS初始化的步骤概述：

1. 调用 OpenSSL 库的初始化函数 `SSL_library_init()` 和 `openssl_add_all_algorithms()`，以初始化和加载 OpenSSL 支持库。
2. 调用 `SSL_CTX_new()` 函数创建一个新的 TLS 上下文对象，该对象用于 TLS 连接的创建和配置。
3. 调用 `SSL_CTX_set_ecdh_auto()` 函数以启用自动选择椭圆曲线密钥交换算法，这是一种更安全的方式来生成加密密钥。
4. 调用 `SSL_CTX_set_session_cache_mode()` 函数启用 TLS 会话缓存，以提高 TLS 连接的性能。
5. 调用 `SSL_CTX_set_verify()` 函数设置 TLS 连接的验证模式，可以配置为验证服务器证书或客户端证书，或者不进行证书验证。
6. 调用 `SSL_CTX_use_certificate_chain_file()` 和 `SSL_CTX_use_PrivateKey_file()` 函数加载服务器证书和私钥，以便使用 TLS 进行加密通信。
7. 调用 `SSL_CTX_set_cipher_list()` 函数设置 TLS 连接支持的加密算法列表，以确保 TLS 连接的安全性。
8. 最后，为了提供更好的性能和安全性，Redis 使用 `setsockopt()` 函数启用 TCP Fast Open (TFO) 特性，这将允许连接更快地建立。

通过以上步骤，`tlsInit()` 函数初始化了 Redis 服务器的 TLS 支持，并为加密通信创建了安全的 TLS 上下文。

15 保存服务器信息

```
1  /* Store the executable path and arguments in a safe place in
   order
2      * to be able to restart the server later. */
3      server.executable = getAbsolutePath(argv[0]);
4      server.exec_argv = zmalloc(sizeof(char*)*(argc+1));
5      server.exec_argv[argc] = NULL;
6      for (j = 0; j < argc; j++) server.exec_argv[j] =
zstrdup(argv[j]);
```

这段代码用于保存 Redis 服务器启动时的可执行文件路径和命令行参数，以便在重启服务器时使用。具体步骤如下：

首先通过调用 `getAbsolutePath` 函数获取当前可执行文件的绝对路径，然后将其保存在 `server.executable` 变量中。接着使用 `zmalloc` 函数动态分配内存，分配 `argc+1` 个指向 `char` 类型的指针，将其保存在 `server.exec_argv` 变量中。最后使用 `zstrdup` 函数为每个命令行参数动态分配内存，并将指针保存在 `server.exec_argv` 数组中，最后将 `server.exec_argv` 数组的最后一个元素设置为 `NULL`。

这样做的目的是为了在 Redis 服务器异常退出后，可以通过重新执行可执行文件并传入相同的命令行参数，来恢复 Redis 服务器的状态。这个机制通常用于自动重启 Redis 服务器，以提高 Redis 服务器的可用性。

16 哨兵模式启动

```
1 if (server.sentinel_mode) {
2     initSentinelConfig();
3     initSentinel();
4 }
```

这段代码在判断 Redis 是否是运行在 Sentinel 模式下。如果是，那么会先执行 `initSentinelConfig()` 来初始化 Sentinel 相关的配置信息，再调用 `initSentinel()` 来初始化 Sentinel 相关的数据结构和网络连接等。否则，不做任何处理，直接进入正常的 Redis 模式。这里所说的 Sentinel 是 Redis 高可用方案中的一个组件，用于实现自动故障转移等功能。

这个功能将会在后续的哨兵处详细讲解

17 检测RDB或者AOF文件

```
1 if (strstr(exec_name,"redis-check-rdb") != NULL)
2     redis_check_rdb_main(argc,argv,NULL);
3 else if (strstr(exec_name,"redis-check-aof") != NULL)
4     redis_check_aof_main(argc,argv);
```

这段代码主要是根据可执行文件名 `exec_name` 是否包含 `redis-check-rdb` 或者 `redis-check-aof` 字符串来决定执行对应的函数，即 `redis_check_rdb_main()` 或者 `redis_check_aof_main()` 函数。

当 `redis-server` 启动时，会传入参数 `argv`，其中第一个参数是可执行文件名，例如 `/usr/local/bin/redis-server`。因此，可以通过查找 `argv[0]` 中是否包含特定的字符串来判断当前程序的运行模式。对于 `redis-check-rdb` 和 `redis-check-aof` 这两个程序来说，它们都是用来检查 Redis 数据文件的工具程序，因此需要单独的逻辑来处理。

如果 `exec_name` 包含 `redis-check-rdb`，则会调用 `redis_check_rdb_main()` 函数；如果包含 `redis-check-aof`，则会调用 `redis_check_aof_main()` 函数。这两个函数都是用来检查 Redis 数据文件的工具程序，用于检查 RDB 文件和 AOF 文件的正确性和完整性。

需要注意的是，这段代码是在 `if (server.sentinel_mode)` 的外层，因此它只有在 Redis Server 不是以 Sentinel 模式启动时才会执行。如果以 Sentinel 模式启动，那么不会执行这段代码，而是在 `initSentinel()` 函数中根据 Sentinel 模式启动的参数来决定是否执行数据文件检查的逻辑。

18 检测用户命令行参数

18.1 version

```
1  if (strcmp(argv[1], "-v") == 0 ||
2      strcmp(argv[1], "--version") == 0) version();
```

这段代码是用于检查用户是否传入了 `-v` 或 `--version` 参数，如果传入了这些参数，则调用 `version()` 函数打印 Redis 的版本信息。

```
1  void version(void) {
2      printf("Redis server v=%s sha=%s:%d malloc=%s bits=%d
3          build=%11x\n",
4              REDIS_VERSION,
5              redisGitSHA1(),
6              atoi(redisGitDirty()) > 0,
7              ZMALLOC_LIB,
8              sizeof(long) == 4 ? 32 : 64,
9              (unsigned long long) redisBuildId());
10     exit(0);
11 }
```

这段代码实现了 `redis-server` 命令的版本号展示功能。当在命令行中执行 `redis-server -v` 或 `redis-server --version` 时，会触发 `version()` 函数，该函数会输出当前 Redis 服务器的版本号、Git SHA1 值、是否为脏数据版本、Redis 服务器使用的内存分配器、运行在 32 位还是 64 位机器上、以及构建 ID 等信息，然后使用 `exit()` 函数退出程序。

18.2 help

```
1  if (strcmp(argv[1], "--help") == 0 ||
2      strcmp(argv[1], "-h") == 0) usage();
```

这段代码是判断用户是否输入了 `--help` 或 `-h` 参数，并在用户输入时调用 `usage()` 函数来显示 Redis 的帮助信息。如果用户输入了这两个参数中的任意一个，`strcmp()` 函数会返回 0，进入 if 语句，执行 `usage()` 函数，否则继续执行 Redis 的主逻辑。

```

1 void usage(void) {
2     fprintf(stderr, "Usage: ./redis-server
[/path/to/redis.conf] [options] [-]\n");
3     fprintf(stderr, "        ./redis-server - (read config from
stdin)\n");
4     fprintf(stderr, "        ./redis-server -v or --version\n");
5     fprintf(stderr, "        ./redis-server -h or --help\n");
6     fprintf(stderr, "        ./redis-server --test-memory
<megabytes>\n");
7     fprintf(stderr, "        ./redis-server --check-system\n");
8     fprintf(stderr, "\n");
9     fprintf(stderr, "Examples:\n");
10    fprintf(stderr, "        ./redis-server (run the server with
default conf)\n");
11    fprintf(stderr, "        echo 'maxmemory 128mb' | ./redis-
server -\n");
12    fprintf(stderr, "        ./redis-server
/etc/redis/6379.conf\n");
13    fprintf(stderr, "        ./redis-server --port 7777\n");
14    fprintf(stderr, "        ./redis-server --port 7777 --
replicaof 127.0.0.1 8888\n");
15    fprintf(stderr, "        ./redis-server /etc/myredis.conf --
loglevel verbose -\n");
16    fprintf(stderr, "        ./redis-server /etc/myredis.conf --
loglevel verbose\n\n");
17    fprintf(stderr, "Sentinel mode:\n");
18    fprintf(stderr, "        ./redis-server /etc/sentinel.conf -
-sentinel\n");
19    exit(1);
20 }

```

`usage()` 函数是用来打印Redis服务器的使用帮助信息的。在命令行参数解析过程中，如果用户指定了 `--help` 或 `-h` 选项，则会调用该函数来打印帮助信息。

该函数会向标准错误输出流（`stderr`）打印一些使用示例和选项说明。用户可以通过命令行参数来设置Redis的配置选项，例如指定配置文件路径、指定日志级别等。最后，函数通过调用 `exit()` 函数退出程序，返回状态码1。

18.3 test-memory


```

1  if (strcmp(argv[1], "--test-memory") == 0) {
2      if (argc == 3) {
3          memtest(atoi(argv[2]),50);
4          exit(0);
5      } else {
6          fprintf(stderr,"Please specify the amount of
memory to test in megabytes.\n");
7          fprintf(stderr,"Example: ./redis-server --
test-memory 4096\n\n");
8          exit(1);
9      }
10 }

```

这段代码是处理 `--test-memory` 参数的，这个参数用于在启动 Redis 服务器之前测试系统内存的可用性。如果用户在命令行中指定了 `--test-memory` 参数并且提供了一个数字，那么 Redis 将测试系统中该数字大小的内存是否可用。如果内存可用，Redis 将打印一条消息并退出程序。如果用户没有提供数字或提供了错误的数字，则 Redis 将打印错误消息并退出程序。

```

(root@ kali)-[~]
# redis-server --test-memory
Please specify the amount of memory to test in megabytes.
Example: ./redis-server --test-memory 4096

(root@ kali)-[~]
# redis-server --test-memory 10240
Unable to allocate 10240 megabytes: Cannot allocate memory

(root@ kali)-[~]
#

```

18.4 系统检查

```

1  if (strcmp(argv[1], "--check-system") == 0) {
2      exit(syscheck() ? 0 : 1);
3  }

```

这段代码是在检查是否需要执行 Redis 系统检查（system check）。如果命令行参数中包含了 `--check-system` 参数，则执行系统检查，如果通过检查，则程序返回 0，否则返回 1，该返回值将被操作系统当作程序的返回值。如果没有包含 `--check-system` 参数，则不进行系统检查，程序继续向下执行。

```

1  int syscheck(void) {
2      check *cur_check = checks;
3      int ret = 1;

```

```

4     sds err_msg = NULL;
5     while (cur_check->check_fn) {
6         int res = cur_check->check_fn(&err_msg);
7         printf("[%s]...", cur_check->name);
8         if (res == 0) {
9             printf("skipped\n");
10        } else if (res == 1) {
11            printf("OK\n");
12        } else {
13            printf("WARNING:\n");
14            printf("%s\n", err_msg);
15            sdsfree(err_msg);
16            ret = 0;
17        }
18        cur_check++;
19    }
21    return ret;
22 }
23

```

这段代码实现了 Redis 服务器启动时的系统检查功能，主要是检查服务器运行所需的一些系统资源和配置参数是否满足要求。

当执行 `./redis-server --check-system` 命令时，程序会调用 `syscheck` 函数进行系统检查。该函数首先会遍历全局数组 `checks`，该数组中存储了多个系统检查函数的指针，每个函数用于检查一个系统资源或参数。然后逐个调用每个检查函数，并根据检查结果输出相应的信息，最后返回整个系统检查的结果。

其中，每个检查函数都需要返回一个整型值，表示检查的结果。如果返回值为 0，表示该检查函数被跳过；如果返回值为 1，表示该检查函数检查通过；如果返回值为其他非零值，表示该检查函数检查失败，此时需要将错误信息保存在 `err_msg` 指针指向的缓冲区中，并将返回值作为该函数的检查结果。

需要注意的是，在遍历 `checks` 数组并调用检查函数时，程序会先输出一个类似 `[xxx]...` 的提示信息，其中 `xxx` 表示当前正在进行的检查任务的名称，方便用户了解当前检查任务的进展情况。如果某个检查任务被跳过，则程序不会输出任何信息；如果该任务检查通过，则程序输出 `OK` 字样；如果该任务检查失败，则程序输出 `WARNING` 字样，并将错误信息打印出来。最后，如果所有检查任务都检查通过，则 `syscheck` 函数返回 1；否则返回 0。

```

[root@ kali:~]# ./redis-server --check-system
[slow-clocksourc]... OK
[xen-clocksourc]... OK
[overcommit]... WARNING:
Memory overcommit must be enabled! Without it, a background save or replication may fail under low memory condition. Being disabled, it can can also cause failures wi
thout low memory condition, see https://github.com/jemalloc/jemalloc/issues/1328. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot
or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
[THP]... WARNING:
You have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command '
echo madvise > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be res
tarted after THP is disabled (set to 'madvise' or 'never').

```

18.5 检查配置文件

```
1  if (argv[1][0] != '-') {
2      /* Replace the config file in server.exec_argv with
   its absolute path. */
3      server.configfile = getAbsolutePath(argv[1]);
4      zfree(server.exec_argv[1]);
5      server.exec_argv[1] = zstrdup(server.configfile);
6      j = 2; // Skip this arg when parsing options
7  }
```

这段代码主要是处理Redis服务器的命令行参数。如果第一个参数不是以“-”开头，那么这个参数应该是配置文件的路径。此时，服务器会将配置文件的绝对路径存储在 `server.configfile` 中，并替换 `server.exec_argv` 数组中的配置文件路径参数为绝对路径。最后，`j` 被设置为2，这是为了在后续解析选项时跳过这个参数。

19 命令行配置

```
1  while(j < argc) {
2      /* Either first or last argument - Should we read
   config from stdin? */
3      if (argv[j][0] == '-' && argv[j][1] == '\0' && (j
   == 1 || j == argc-1)) {
4          config_from_stdin = 1;
5      }
6      else if (handled_last_config_arg && argv[j][0] ==
   '-' && argv[j][1] == '-') {
7          /*
8              *****
9              */
10         } else {
11             /* Option argument */
12             options =
   sdscatrepr(options,argv[j],strlen(argv[j]));
13             options = sdscat(options," ");
14             handled_last_config_arg = 1;
15         }
16         j++;
17     }
18
19 }
```

19.1 if

```
1  if (argv[j][0] == '-' && argv[j][1] == '\0' && (j == 1 || j ==  
    argc-1)) {  
2      config_from_stdin = 1;  
3  }
```

如果当前命令行参数的第一个字符为 '-', 第二个字符为 '\0', 且该参数是第一个或最后一个参数, 那么就表示这是一个 "-" 参数, 表示从标准输入中读取配置文件。

这个判断是用来检查命令行参数是否是一个单独的 -, 如果是的话, 表示程序需从标准输入中读取配置。例如, 在 Linux 命令行中运行 Redis 的命令为 `redis-server -`, 这将使 Redis 从标准输入中读取配置。在这种情况下, 程序将通过检查命令行参数是否为单个 - 来判断是否需要从标准输入中读取配置。

```
(root@kali)-[~]  
# redis-server -  
1336:C 10 Apr 2023 15:46:24.972 # Reading config from stdin
```

19.2 else if

```
1  else if (handled_last_config_arg && argv[j][0] == '-' &&  
    argv[j][1] == '-')
```

如果已经处理完最后一个配置参数, 而且当前参数以双破折线 (--) 开头, 则进入以下分支。

```
1  if (sdslen(options)) options = sdscat(options, "\n");
```

这行代码是将 `options` 变量的内容末尾添加一个换行符, 以确保下一行输出的内容和当前行的选项显示在不同行上, 以提高输出的可读性。其中 `options` 变量是在解析命令行参数时用来记录指定的选项的一个字符串。

```
1  options = sdscat(options, argv[j]+2);  
2  options = sdscat(options, " ");
```

`argv[j]` 是一个指向命令行参数字符串的指针, `argv[j][2]` 表示这个字符串的第三个字符, 因为C语言数组的下标从0开始。所以 `argv[j]+2` 表示从字符串的第三个字符开始的子字符串。这里的目的是为了跳过命令行参数的前缀 --。

这段代码的作用是将输入的参数解析成redis的配置选项, 例如 `--port 6379`。在代码中, 当检测到参数以 -- 开头时, 将该参数作为选项的一部分, 并将它加入一个字符串变量 `options` 中。

首先，该段代码通过 `sdslen()` 函数获取 `options` 字符串的长度，若该字符串不为空，则在其末尾加上一个换行符 `\n`，用于将不同的选项区分开来。

然后，该段代码通过 `sdslen()` 函数获取当前参数的长度，将其作为选项的一部分加入 `options` 字符串，并在字符串末尾加上一个空格，以便将该选项与后面的参数分隔开来。最终生成的 `options` 字符串就是redis的配置选项。

```
1 | argv_tmp = sdssplitargs(argv[j], &argc_tmp);
```

`sdssplitargs` 函数是 Redis 中的一个字符串操作函数，其作用是将给定的字符串按照空格分隔符分成多个子字符串，并返回一个字符串数组，同时修改给定的参数 `argc_tmp` 来表示字符串数组的长度。

在这里，`argv[j]` 表示一个命令行参数，它需要按照空格分隔符进行分割。分割后得到的字符串数组 `argv_tmp` 将在接下来的代码中被处理。

```
1 | if (argc_tmp == 1) {
2 |     /* Means that we only have one option name, like --
   | port or "--port " */
3 | }
```

```
1 | handled_last_config_arg = 0;
```

这行代码的作用是将 `handled_last_config_arg` 变量的值设为0，它被用来跟踪上一个参数是否是配置文件的路径名，以便下一个参数能够被正确地解析。当它被设置为0时，下一个参数将被视为一个新的配置项或选项，而不是上一个配置文件路径名的一部分。

```
1 | if ((j != argc-1) && argv[j+1][0] == '-' && argv[j+1][1] ==
   | '-' &&
2 |         !strcasecmp(argv[j], "--save"))
3 |     {
4 |         options = sdscat(options, "\\\"");
5 |         handled_last_config_arg = 1;
6 |     }
```

这段代码是处理特殊情况的。如果当前处理的参数是 `--save`，并且后面一个参数以 `--` 开头（即后面一个参数也是一个选项），那么就将 `handled_last_config_arg` 标志重置为0，然后在选项字符串 `options` 中加入一个空字符串 `""`，以便将其与后面的选项分离开来。这个特殊处理的原因是为了与早期版本的 Redis 兼容，因为有些用户会从一个数组生成一个命令行，而当它为空时就会产生这种情况。

```

1  else if ((j == argc-1) && !strcasecmp(argv[j], "--save")) {
2      options = sdscat(options, "\\");
3  }

```

如果命令行最后一个参数是"--save", 且没有任何配置信息紧随其后, 这里会将一个空字符串 "" 追加到 options 字符串的末尾。这是为了使其与上面提到的特殊情况保持一致, 从而可以处理空配置参数的情况。例如, "--save"选项被使用而没有任何配置信息紧随其后。

```

1  else if ((j != argc-1) && argv[j+1][0] == '-' && argv[j+1][1]
    == '-' &&
2      !strcasecmp(argv[j], "--sentinel"))
3      {
4          options = sdscat(options, "");
5          handled_last_config_arg = 1;
6      }

```

这段代码是在解析 Redis 的命令行参数时, 处理一些特殊的选项的情况。在这个 else if 语句中, 它会检查是否遇到了 --sentinel 这个选项, 并且如果下一个参数也是以 -- 开头, 就将 handled_last_config_arg 标志位设置为 1。

--sentinel 是一个伪配置选项, 它没有值。如果下一个参数也是以 -- 开头, 就说明它不是 --sentinel 选项的值, 这时就需要将 handled_last_config_arg 标志位重置, 以便后面的参数可以被正确解析。options = sdscat(options, ""); 的作用是向 options 字符串中添加一个空字符串, 表示 --sentinel 选项的存在。

```

1  else if ((j == argc-1) && !strcasecmp(argv[j], "--sentinel"))
2      {
3          options = sdscat(options, "");
4      }

```

这段代码是针对解析Redis服务器启动参数的逻辑。在这个逻辑中, Redis使用 argv[]数组存储启动参数, 并根据参数类型进行不同的处理。

具体来说, 当解析到一个形如"--config"的参数时, Redis需要将该参数作为一个配置文件路径处理。如果该参数后面紧跟着的是另一个参数, 则说明该参数并不是最后一个参数, Redis需要将下一个参数作为一个普通参数处理, 并将该参数设置为“已处理完最后一个配置文件参数”。如果下一个参数也是一个"--"类型的参数, 则说明该参数后面没有其他参数了, 因此Redis需要将该参数设置为最后一个参数, 并将该参数设置为“已处理完最后一个配置文件参数”。

当解析到一个形如"--save"或"--sentinel"的参数时，Redis需要将该参数作为一个配置项名处理。如果该参数后面紧跟着的是另一个参数，则说明该参数并不是最后一个参数，Redis需要将下一个参数作为该配置项的值处理，并将该参数设置为“已处理完最后一个配置文件参数”。如果下一个参数也是一个"--"类型的参数，则说明该参数后面没有其他参数了，因此Redis需要将该参数设置为最后一个参数，并将该参数设置为“已处理完最后一个配置文件参数”。如果该参数是最后一个参数，则Redis需要将该参数设置为该配置项的值。

```
1  else {
2
3      handled_last_config_arg = 1;
4  }
5  sdsfreesplitres(argv_tmp, argc_tmp);
6  }
```

如果当前参数以"--"开头，且同时包含配置选项名和选项值（如"--port 6380"），则需要将 handled_last_config_arg 标志位设置为 1，以表明当前参数是一个新的配置选项。

19.3 else

```
1  else {
2      /* Option argument */
3      options =
4      sdscatrepr(options,argv[j],strlen(argv[j]));
5      options = sdscat(options," ");
6      handled_last_config_arg = 1;
7  }
```

这段代码是在处理命令行参数时的一种情况，即当前参数既不是配置文件的参数，也不是选项参数，而是一个选项参数的值。在这种情况下，将当前参数作为选项参数的值添加到 options 字符串中，并将 handled_last_config_arg 标记设置为 1，表示已处理完上一个配置参数，可以继续处理下一个参数。

```
1  sds *argv_tmp;
2  int argc_tmp;
3  int handled_last_config_arg = 1;
4  while(j < argc) {
5      /* Either first or last argument - Should we read
6      config from stdin? */
7      if (argv[j][0] == '-' && argv[j][1] == '\0' && (j
8      == 1 || j == argc-1)) {
9          config_from_stdin = 1;
10     }
```



```

37         handled_last_config_arg = 1;
38     }
39     else if ((j == argc-1) &&
!strcasecmp(argv[j], "--save")) {
40         /* Special case: when empty save is
the last argument.
41         * In this case, we append an empty ""
config value to the options,
42         * so it will become `--save ""` and
will follow the same reset thing. */
43         options = sdscat(options, "\\\"");
44     }
45     else if ((j != argc-1) && argv[j+1][0] ==
'- ' && argv[j+1][1] == '-' &&
46         !strcasecmp(argv[j], "--sentinel"))
47     {
48         /* Special case: handle some things
like `--sentinel --config value`.
49         * It is a pseudo config option with
no value. In this case, if next
50         * argument starts with `--`, we will
reset handled_last_config_arg flag.
51         * We are doing it to be compatible
with pre 7.0 behavior (which we
52         * break it in #10660, 7.0.1). */
53         options = sdscat(options, "");
54         handled_last_config_arg = 1;
55     }
56     else if ((j == argc-1) &&
!strcasecmp(argv[j], "--sentinel")) {
57         /* Special case: when --sentinel is
the last argument.
58         * It is a pseudo config option with
no value. In this case, do nothing.
59         * We are doing it to be compatible
with pre 7.0 behavior (which we
60         * break it in #10660, 7.0.1). */
61         options = sdscat(options, "");
62     }
63     } else {
64         /* Means that we are passing both config
name and it's value in the same arg,
65         * like "--port 6380", so we need to reset
handled_last_config_arg flag. */
66         handled_last_config_arg = 1;

```

```

67         }
68         sdsfreesplitres(argv_tmp, argc_tmp);
69     } else {
70         /* Option argument */
71         options =
sdsatrepr(options,argv[j],strlen(argv[j]));
72         options = sdscat(options," ");
73         handled_last_config_arg = 1;
74     }
75     j++;
76 }

```

20 loadServerConfig

```

1 loadServerConfig(server.configfile, config_from_stdin,
options);

```

这段函数大家也比较熟悉了，我在简单概述一下，详细请参考前面文章

`loadServerConfig()` 函数用于加载 Redis 服务器的配置。它接受三个参数：

- `configfile`：Redis 配置文件的路径。
- `config_from_stdin`：是否从标准输入中读取配置。这个值会在解析命令行参数时被设置。
- `options`：Redis 配置选项。这个字符串包含了从命令行读取到的 Redis 配置选项和对应的值。

当 Redis 服务器启动时，它需要读取一个配置文件来确定一些基本配置参数。Redis 支持从命令行传递参数来覆盖这些基本配置参数，也支持从标准输入中读取配置，但是这些配置会覆盖命令行参数和配置文件参数。`loadServerConfig()` 函数的作用就是根据这些参数来加载 Redis 服务器的配置。

21 哨兵模式运行

```

1 if (server.sentinel_mode) loadSentinelConfigFromQueue();
2 sdsfree(options);

```

这段代码主要是检查Redis是否在sentinel模式下运行，如果是，则调用 `loadSentinelConfigFromQueue()` 函数从sentinel.conf中加载配置。接着，释放 options 字符串的内存空间，因为在配置加载后，这个字符串就不再需要了。

哨兵模式会在后面详细介绍

21 检查哨兵配置文件

```
1 | if (server.sentinel_mode) sentinelCheckConfigFile();
```

`sentinelCheckConfigFile()` 函数是用于检查 Sentinel 配置文件的函数。在 Redis Sentinel 模式下，Redis 实例是由 Sentinel 管理的，它需要一个 Sentinel 配置文件。该函数用于检查 Sentinel 配置文件是否正确配置。如果文件中没有配置 Redis Sentinel 的运行条件，则该函数会输出错误信息并导致 Redis Sentinel 启动失败。

具体来说，`sentinelCheckConfigFile()` 函数首先检查配置文件是否存在。如果不存在，则函数会输出错误消息并导致 Sentinel 无法启动。如果存在，则函数会读取配置文件并检查其内容。配置文件应该至少包含以下内容：

1. Sentinel 运行的端口号。
2. 连接到 Redis 服务器的地址和端口。
3. 在启动 Sentinel 之前，需要 Sentinel 所监控的 Redis 实例数量。
4. Sentinel 的其他配置参数。

如果配置文件中存在上述内容，则函数会返回 1。如果缺少其中任何一个内容，则函数会输出相应的错误消息并返回 0，导致 Sentinel 启动失败。

总之，`sentinelCheckConfigFile()` 函数用于检查 Sentinel 配置文件的正确性，以确保 Sentinel 可以正确地监控 Redis 实例。

```
1 | void sentinelCheckConfigFile(void) {
2 |     if (server.configfile == NULL) {
3 |         serverLog(LL_WARNING,
4 |             "Sentinel needs config file on disk to save state.
Exiting...");
5 |         exit(1);
6 |     } else if (access(server.configfile,W_OK) == -1) {
7 |         serverLog(LL_WARNING,
8 |             "Sentinel config file %s is not writable: %s.
Exiting...",
9 |             server.configfile,strerror(errno));
10 |        exit(1);
11 |    }
12 | }
```

22 监控模式运行

```
1 | server.supervised =
redisIsSupervised(server.supervised_mode);
```

这行代码会根据 `server.supervised_mode` 的值来设置 `server.supervised` 的值，即 Redis 是否以监控模式运行。监控模式是指在启动 Redis 时将其交给一个监控进程（如 `systemd`）来管理，由监控进程负责自动重启 Redis 服务，以保证 Redis 服务的高可用性。

具体来说，`redisIsSupervised` 函数会检查当前 Redis 进程是否被某个监控进程管理，检查方式因平台而异，Linux 上的实现是检查当前进程的父进程是否为 1（`init` 进程）。如果是，说明 Redis 进程被启动在监控模式下，否则不是。

23 服务端后台运行

```
1 int background = server.daemonize && !server.supervised;
2     if (background) daemonize();
```

这段代码的作用是判断当前 Redis 是否需要在后台运行，并在需要时执行 `daemonize()` 函数使其在后台运行。

首先，通过 `server.daemonize` 变量来判断是否需要在后台运行。如果该变量的值为 1，表示需要在后台运行，否则为 0。

其次，如果 Redis 是以 `supervised` 模式运行的，即 `server.supervised` 变量的值为 1，表示 Redis 在被 `supervisord` 进程管理，因此不需要在此处进行后台运行，因为 `supervisord` 会控制 Redis 进程的生命周期。如果 `server.supervised` 变量的值为 0，则表示 Redis 不是以 `supervised` 模式运行的，此时需要在后台运行，因此执行 `daemonize()` 函数将 Redis 进程 `daemonize` 到后台。

最终，如果 Redis 需要在后台运行，则 `background` 变量的值为 1，否则为 0。

```
1 void daemonize(void) {
2     int fd;
3
4     if (fork() != 0) exit(0); /* parent exits */
5     setsid(); /* create a new session */
6
7     /* Every output goes to /dev/null. If Redis is daemonized
8     but
9     * the 'logfile' is set to 'stdout' in the configuration
10    file
11    * it will not log at all. */
12    if ((fd = open("/dev/null", O_RDWR, 0)) != -1) {
13        dup2(fd, STDIN_FILENO);
14        dup2(fd, STDOUT_FILENO);
15        dup2(fd, STDERR_FILENO);
16        if (fd > STDERR_FILENO) close(fd);
17    }
```

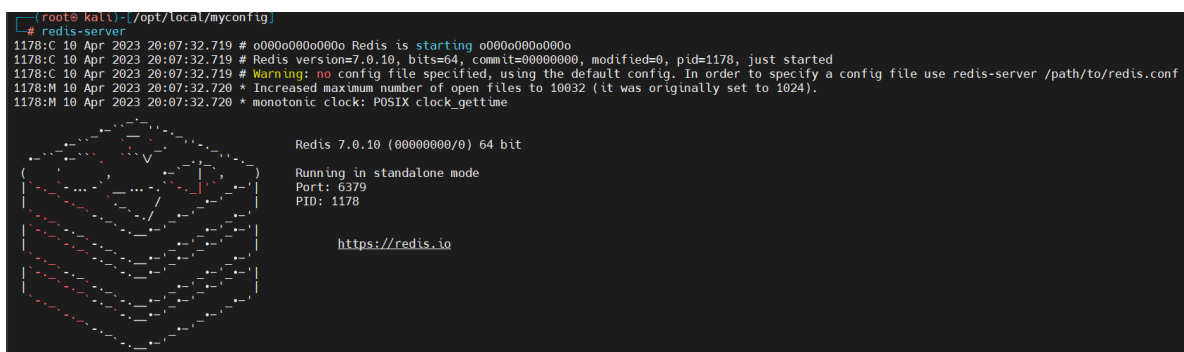
24 服务端输出启动信息

```

1  serverLog(LL_WARNING, "o000o000o000o Redis is starting
   o000o000o000o");
2  serverLog(LL_WARNING,
3      "Redis version=%s, bits=%d, commit=%s, modified=%d,
   pid=%d, just started",
4      REDIS_VERSION,
5      (sizeof(long) == 8) ? 64 : 32,
6      redisGitSHA1(),
7      strtol(redisGitDirty(), NULL, 10) > 0,
8      (int) getpid());
9

```

这段代码是 Redis 服务器启动时输出的一些启动信息，包括 Redis 版本、当前机器是 32 位还是 64 位、Redis 代码的 git commit ID、Redis 代码是否被修改过以及 Redis 服务器的进程 ID。这些信息可以帮助管理员在出现问题时更快地定位问题所在，也可以方便管理员确认 Redis 服务器启动后的状态。



```

[roots@kali:~/opt/local/myconfig]
# redis-server
1178:C 10 Apr 2023 20:07:32.719 # o000o000o000o Redis is starting o000o000o000o
1178:C 10 Apr 2023 20:07:32.719 # Redis version=7.0.10, bits=64, commit=00000000, modified=0, pid=1178, just started
1178:C 10 Apr 2023 20:07:32.719 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
1178:M 10 Apr 2023 20:07:32.720 * Increased maximum number of open files to 10032 (it was originally set to 1024).
1178:M 10 Apr 2023 20:07:32.720 * monotonic clock: POSIX clock_gettime

Redis 7.0.10 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 1178

https://redis.io

```

```

1  if (argc == 1) {
2      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]);
3  } else {
4      serverLog(LL_WARNING, "Configuration loaded");
5  }

```

这段代码是用来记录 Redis 的启动信息的。如果在命令行中没有指定配置文件路径，则记录警告消息，表示使用默认配置。如果指定了配置文件，则记录“Configuration loaded”信息。

25 initServer

```
1 | initServer();
```

这个函数也十分熟悉了，我不在赘述了。

`initServer()` 函数用于初始化服务器相关的结构体和参数，其主要执行以下操作：

1. 初始化服务器状态结构体 `server` 的各个字段，如各个数据库的状态结构体、命令表、统计信息等；
2. 设置服务器的各种默认参数值，如最大客户端数量、默认的最大内存限制等；
3. 初始化随机数生成器，以便后续使用；
4. 读取系统的配置参数，包括最大打开文件数量限制等；
5. 设置崩溃处理函数，以便程序发生错误时能够及时处理；
6. 初始化慢查询日志；
7. 加载用户自定义的扩展命令；
8. 初始化事件循环机制。

总的来说，`initServer()` 函数为服务器的正常运行打下了基础，后续的代码将会在此基础上进一步执行初始化和启动服务器的操作。

26 创建PID文件

```
1 | if (background || server.pidfile) createPidFile();
```

这段代码用于创建PID文件。PID文件是一个文本文件，其中包含Redis服务器进程的进程ID。通过检查该文件，其他程序可以确定Redis服务器正在运行的进程ID，以便进行管理。如果服务器被配置为以守护进程方式运行，或者服务器需要将PID写入文件，则该函数将创建一个PID文件。如果服务器正在以监视模式运行，则不会创建PID文件。


```

1 void createPidFile(void) {
2     /* If pidfile requested, but no pidfile defined, use
3      * default pidfile path */
4     if (!server.pidfile) server.pidfile =
zstrdup(CONFIG_DEFAULT_PID_FILE);
5
6     /* Try to write the pid file in a best-effort way. */
7     FILE *fp = fopen(server.pidfile,"w");
8     if (fp) {
9         fprintf(fp,"%d\n",(int)getpid());
10        fclose(fp);
11    }
12 }

```

27 修改进程名称

```

1 if (server.set_proc_title) redisSetProcTitle(NULL);

```

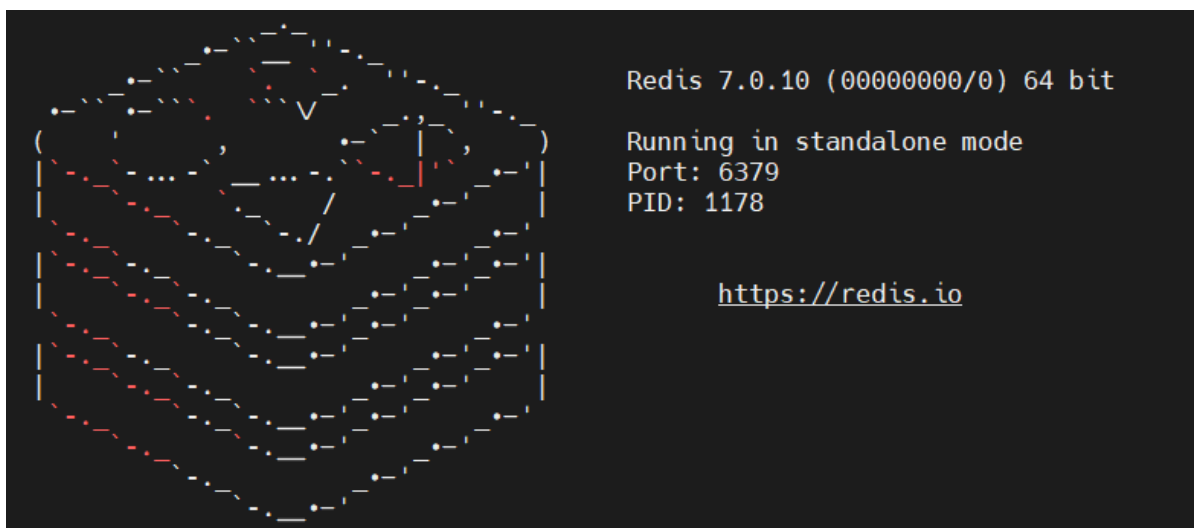
这行代码是在调用redisSetProcTitle()函数，用来修改当前进程的名称，以便于更好地标识当前正在执行的程序。redisSetProcTitle()函数在不同的平台上会有不同的实现，这里不再赘述。在这里，如果server.set_proc_title变量的值为1，则执行redisSetProcTitle(NULL)函数，将当前进程的名称设置为NULL，即不显示名称。

28 ASCII艺术字符画

```

1 redisAsciiArt();

```



redisAsciiArt() 是Redis启动时打印的ASCII艺术字符画，类似于Logo。这个函数主要是为了让Redis启动时输出更加美观，增加用户体验。

29 检查当前backlog

```
1 | checkTcpBacklogSettings();
```

```
1 void checkTcpBacklogSettings(void) {
2     #if defined(HAVE_PROC_SOMAXCONN)
3         FILE *fp = fopen("/proc/sys/net/core/somaxconn", "r");
4         char buf[1024];
5         if (!fp) return;
6         if (fgets(buf, sizeof(buf), fp) != NULL) {
7             int somaxconn = atoi(buf);
8             if (somaxconn > 0 && somaxconn < server.tcp_backlog) {
9                 serverLog(LL_WARNING, "WARNING: The TCP backlog
10                setting of %d cannot be enforced because
11                /proc/sys/net/core/somaxconn is set to the lower value of
12                %d.", server.tcp_backlog, somaxconn);
13            }
14        }
15        fclose(fp);
16    #elif defined(HAVE_SYSCTL_KIPC_SOMAXCONN)
17        int somaxconn, mib[3];
18        size_t len = sizeof(int);
19
20        mib[0] = CTL_KERN;
21        mib[1] = KERN_IPC;
22        mib[2] = KIPC_SOMAXCONN;
23
24        if (sysctl(mib, 3, &somaxconn, &len, NULL, 0) == 0) {
25            if (somaxconn > 0 && somaxconn < server.tcp_backlog) {
26                serverLog(LL_WARNING, "WARNING: The TCP backlog
27                setting of %d cannot be enforced because kern.ipc.somaxconn is
28                set to the lower value of %d.", server.tcp_backlog,
29                somaxconn);
30            }
31        }
32    #elif defined(HAVE_SYSCTL_KERN_SOMAXCONN)
33        int somaxconn, mib[2];
34        size_t len = sizeof(int);
35
36        mib[0] = CTL_KERN;
37        mib[1] = KERN_SOMAXCONN;
38
39        if (sysctl(mib, 2, &somaxconn, &len, NULL, 0) == 0) {
40            if (somaxconn > 0 && somaxconn < server.tcp_backlog) {
```

```

35         serverLog(LL_WARNING, "WARNING: The TCP backlog
        setting of %d cannot be enforced because kern.somaxconn is set
        to the lower value of %d.", server.tcp_backlog, somaxconn);
36     }
37 }
38 #elif defined(SOMAXCONN)
39     if (SOMAXCONN < server.tcp_backlog) {
40         serverLog(LL_WARNING, "WARNING: The TCP backlog setting
        of %d cannot be enforced because SOMAXCONN is set to the lower
        value of %d.", server.tcp_backlog, SOMAXCONN);
41     }
42 #endif
43 }

```

在Redis中，当客户端连接Redis服务器时，Redis会创建一个TCP套接字以进行通信。在Linux系统中，这个套接字有一个backlog参数，用于指定允许挂起的未完成连接的最大数量。

在 `checkTcpBacklogSettings` 函数中，Redis会检查当前backlog设置是否过低，如果过低，将打印一条警告日志。

这是为了避免在客户端连接数增加时可能发生的连接超时或拒绝连接等问题。建议在高流量场景下将backlog设置得较高。

30 初始化完成日志

```

1  if (!server.sentinel_mode) {
2      /* Things not needed when running in Sentinel mode. */
3      serverLog(LL_WARNING, "Server initialized");

```

这段代码中，首先通过判断 `server.sentinel_mode` 是否为真来确定是否是在 Sentinel 模式下运行 Redis，如果不是，就打印一条日志，表示 Redis 服务器已初始化完毕。在 Sentinel 模式下，该语句不会被执行，因为在 Sentinel 模式下 Redis 服务器只会执行 Sentinel 相关的操作，不会执行普通 Redis 服务器的操作。

31 Linux检查

```

1  #ifdef __linux__
2      linuxMemoryWarnings();
3      sds err_msg = NULL;
4      if (checkXenClocksource(&err_msg) < 0) {
5          serverLog(LL_WARNING, "WARNING %s", err_msg);
6          sdsfree(err_msg);

```

```

7         }
8     #if defined (__arm64__)
9         int ret;
10        if ((ret = checkLinuxMadvFreeForkBug(&err_msg)) <= 0)
11    {
12        if (ret < 0) {
13            serverLog(LL_WARNING, "WARNING %s", err_msg);
14            sdsfree(err_msg);
15        } else
16            serverLog(LL_WARNING, "Failed to test the
17            kernel for a bug that could lead to data corruption during
18            background save. "
19            "Your system could be
20            affected, please report this error.");
21        if (!checkIgnorewarning("ARM64-COW-BUG")) {
22            serverLog(LL_WARNING, "Redis will now exit to
23            prevent data corruption. "
24            "Note that it is possible
25            to suppress this warning by setting the following config:
26            ignore-warnings ARM64-COW-BUG");
27            exit(1);
28        }
29    }
30    }
31    #endif /* __arm64__ */

```

这段代码是对Linux操作系统下一些特定的问题进行检查和预警，其中包括：

- linuxMemoryWarnings：在内存使用达到某个阈值时发出警告
- checkXenClocksource：检查系统时钟源是否正确，如果不正确则可能导致性能下降
- checkLinuxMadvFreeForkBug：检查系统是否受到特定内核bug影响，可能导致在后台保存数据时出现数据损坏

如果发现问题，会记录日志，并在某些情况下终止Redis的运行。

32 Redis模块

```
1 | moduleInitModulesSystemLast();
```

`moduleInitModulesSystemLast()` 是一个Redis模块的函数，它会在Redis服务器启动时执行，主要用于初始化所有已加载的Redis模块，并在Redis服务器成功启动时调用各个模块的回调函数。

该函数主要完成以下操作：

1. 遍历Redis模块列表，并调用每个模块的 `RedisModule_OnLoad()` 函数进行初始化。
2. 在Redis服务器成功启动后，遍历Redis模块列表，并调用每个模块的 `RedisModule_OnAfterFork()` 函数进行初始化。
3. 在Redis服务器成功启动后，遍历Redis模块列表，并调用每个模块的 `RedisModule_EventuallyFree()` 函数进行资源释放。

这些模块的回调函数是通过Redis模块API在模块的代码中定义的，并在模块初始化时通过 `RedisModuleTypeMethods` 结构体注册到Redis服务器中。每个模块的回调函数的实现不同，可以在模块代码中自定义。

该函数的作用是确保所有的Redis模块都已经正确初始化，并且已经被成功加载到Redis服务器中。这是Redis模块化架构的核心之一，使得Redis可以轻松地扩展和添加新功能。

33 加载模块

```
1 | moduleLoadFromQueue();
```

`moduleLoadFromQueue()` 函数用于从队列中加载模块。在Redis服务器启动时，会从服务器配置文件中读取模块的相关配置，然后根据配置来加载模块。模块加载时，会检查模块的版本信息和依赖关系，并将模块的API函数注册到Redis服务器中，从而让模块可以被调用。

在Redis服务器运行期间，如果需要加载新的模块或卸载现有的模块，可以通过发送 `MODULE LOAD` 或 `MODULE UNLOAD` 命令来实现。这些命令会将指定的模块信息发送到队列中，然后由 `moduleLoadFromQueue()` 函数来处理。如果加载或卸载成功，函数会返回相应的成功信息，否则会返回错误信息。

34 ACL用户信息

```
1 | ACLLoadUsersAtStartup();
```

`ACLLoadUsersAtStartup()` 是 Redis 在启动时加载 ACL 用户信息的函数。Redis 的 ACL (Access Control List) 功能可以用于控制用户对 Redis 实例的访问权限，从而保障 Redis 实例的安全性。

`ACLLoadUsersAtStartup()` 的作用是读取 Redis 配置文件中指定的 ACL 配置，然后根据配置信息加载所有的用户和用户组到 Redis 实例中。如果没有配置 ACL，则默认使用 "default" 用户和 "allkeys" 用户组。

在 Redis 实例启动时调用 `ACLLoadUsersAtStartup()` 可以确保 Redis 实例在加载完所有需要的用户和用户组之后再开始对外提供服务，从而保障 Redis 实例的安全性。

35 InitServerLast

```
1 | InitServerLast();
```

`InitServerLast()` 是 Redis 初始化的最后一个步骤，它完成了一些初始化工作：

- 初始化慢日志，通过调用 `slowlogInit()` 函数完成。
- 初始化服务器统计信息，通过调用 `statsMetricInit()` 函数完成。
- 初始化时间事件，通过调用 `aeCreateTimeEvent()` 函数完成。
- 初始化事件循环，通过调用 `aeMain()` 函数完成。

总之，`InitServerLast()` 函数是 Redis 初始化的最后一步，在这里，Redis 完成了许多重要的初始化工作，准备好接受客户端的连接并开始工作。

36 AOF元数据

```
1 | aofLoadManifestFromDisk();
```

`aofLoadManifestFromDisk()` 是 Redis 服务器启动后，加载持久化 AOF 模式的元数据文件的函数。AOF 是 Redis 的一种持久化方式，会将每个修改 Redis 数据库的命令追加到 AOF 文件的末尾，以保证数据的持久化。

该函数会在服务器启动时调用，用于读取 `redis.aof_manifest` 文件中的数据，并将数据解析成 AOF 文件的元数据结构。

这个元数据包含了 AOF 文件的大小、最后一次执行 AOF 重写的时间、AOF 文件的存储位置等信息。这些信息对于服务器进行 AOF 重写操作非常重要，因为服务器需要根据这些信息来决定是否需要执行 AOF 重写操作，以及应该如何进行 AOF 重写操作。

在函数内部，首先尝试打开 `redis.aof_manifest` 文件，如果文件打开失败，说明当前服务器并不是第一次启动，也没有执行过 AOF 重写操作，可以直接返回。如果文件打开成功，则从文件中读取元数据信息，并根据这些信息来更新服务器状态。

最后，如果有必要，函数会将新的 AOF 文件移动到指定位置。这样，AOF 文件就已经加载完毕，并准备好接收新的命令追加。

37 从磁盘回复文件

```
1 | loadDataFromDisk();
```

`loadDataFromDisk()` 是 Redis 在启动时从磁盘中加载数据的函数，主要功能是读取 RDB 文件（如果存在）并将其中的数据恢复到 Redis 内存中。

具体来说，该函数首先尝试从配置文件中指定的 RDB 文件路径中加载 RDB 文件，如果找不到该文件，则尝试从自动备份文件中加载 RDB 文件。如果找到 RDB 文件，则执行 `rdLoad()` 函数将其中的数据读取到 Redis 内存中。

需要注意的是，如果 Redis 配置中开启了 AOF 持久化，那么在 `loadDataFromDisk()` 函数加载 RDB 文件之后，还需要调用 `aofRewriteIfNeeded()` 函数执行 AOF 重写操作。该操作会根据 AOF 文件中的数据重新生成 RDB 文件，并将其中的数据写入到新的 AOF 文件中。这样可以保证 RDB 文件和 AOF 文件中的数据一致。

38 是否打开aof?

```
1 | aofOpenIfNeededOnServerStart();
```

`aofOpenIfNeededOnServerStart()` 函数主要用于在 Redis 服务器启动时打开 AOF 文件，如果服务器当前没有打开 AOF 持久化，则会根据配置项进行决定是否需要打开。

具体来说，该函数会根据以下几个方面进行判断：

- 是否在 AOF 重写过程中，如果是，则不需要打开 AOF 文件。
- 是否开启了 AOF 持久化功能，如果没有，则不需要打开 AOF 文件。
- 是否开启了 AOF 自动重写功能，如果开启了，则根据条件判断是否需要触发 AOF 自动重写，如果需要，则不需要打开 AOF 文件。
- 如果以上条件都不满足，则需要打开 AOF 文件。

在打开 AOF 文件之前，该函数还会尝试从 AOF 文件中载入数据到内存中。

总的来说，`aofOpenIfNeededOnServerStart()` 函数的作用就是在 Redis 服务器启动时恢复 AOF 持久化所保存的数据，使得 Redis 服务器能够恢复到上一次停机时的状态。

39 删除旧的AOF文件

```
1 | aofDelHistoryFiles()
```

函数 `aofDelHistoryFiles()` 的作用是删除旧的 AOF 文件。在 AOF 持久化模式下，Redis 会将所有写入命令都追加到 AOF 文件中，以保证数据持久化。当 AOF 文件过大时，Redis 会对其进行压缩和重写，生成新的 AOF 文件。此时，旧的 AOF 文件可以被删除，以释放磁盘空间。

在 Redis 启动时，`aofDelHistoryFiles()` 函数会检查是否开启了 AOF 持久化模式，并且是否启用了 AOF 压缩。如果开启了 AOF 压缩，那么 Redis 会保留多个历史版本的 AOF 文件，以便出现问题时可以回滚到之前的版本。此时，`aofDelHistoryFiles()` 函数会删除旧的 AOF 文件，只保留最近几个版本的 AOF 文件，以释放磁盘空间。

需要注意的是，删除旧的 AOF 文件并不会影响 Redis 的正常运行。在下一次 AOF 压缩时，Redis 会自动清理不再使用的 AOF 文件。

40 集群模式检查0号数据库

```
1  if (server.cluster_enabled) {
2      if (verifyClusterConfigWithData() == C_ERR) {
3          serverLog(LL_WARNING,
4              "You can't have keys in a DB different than
DB 0 when in "
5              "cluster mode. Exiting.");
6          exit(1);
7      }
8  }
```

这段代码的作用是在 Redis 以集群模式启动时，检查是否有非 0 号数据库中存数据。在 Redis 集群模式下，每个节点只维护数据库 0，因此如果在其他数据库中有数据，则无法将该节点添加到集群中。如果发现存在非 0 号数据库中存数据，程序将输出错误信息并退出。

41 来连接吧！

```
1  if (server.ipfd.count > 0 || server.tlsfd.count > 0)
2      serverLog(LL_NOTICE, "Ready to accept connections");
```

这段代码是在 Redis 启动成功后，当 IP 套接字或 TLS 套接字的数量大于 0 时，打印一个日志记录，表示 Redis 已经准备好接受客户端连接了。其中，`LL_NOTICE` 是日志级别，表示普通提示信息。

```
1  if (server.sofd > 0)
2      serverLog(LL_NOTICE, "The server is now ready to
accept connections at %s", server.unixsocket);
```

这段代码的作用是在 Redis 启动成功后，如果 Redis 实例绑定了 TCP 端口或 TLS 端口，或者绑定了 Unix socket，就会在日志中打印出相应的提示信息，表明 Redis 已经准备好接收连接。在这里，`LL_NOTICE` 级别的日志表示该信息是一个正常的通知信息，表明 Redis 启动成功并已准备好接收连接。

```

1  if (server.supervised_mode == SUPERVISED_SYSTEMD) {
2      if (!server.masterhost) {
3          redisCommunicateSystemd("STATUS=Ready to
accept connections\n");
4      } else {
5          redisCommunicateSystemd("STATUS=Ready to
accept connections in read-only mode. Waiting for MASTER <->
REPLICA sync\n");
6      }
7      redisCommunicateSystemd("READY=1\n");
8  }
9  } else {
10     ACLLoadUsersAtStartup();
11     InitServerLast();
12     sentinelIsRunning();
13     if (server.supervised_mode == SUPERVISED_SYSTEMD) {
14         redisCommunicateSystemd("STATUS=Ready to accept
connections\n");
15         redisCommunicateSystemd("READY=1\n");
16     }
17 }

```

这段代码是在 Redis 服务器启动后，执行各种初始化工作之后，最终准备好接受客户端连接之前执行的。在这段代码中，根据 Redis 是否在 systemd 监管模式下运行，服务器会与 systemd 进行通信以通知其已准备好接受连接。在非 systemd 监管模式下，Redis 会加载 ACL 用户信息，然后执行 InitServerLast 函数，该函数执行了一些其他的初始化工作，包括激活模块、开启 RDB/AOF 持久化、初始化 Sentinel 等。最后，Redis 会检查 Sentinel 是否正在运行，如果正在运行，将通过 Sentinel 的 API 向其它 Sentinel 实例发送消息，以通知它们当前实例已经启动。如果服务器在 systemd 监管模式下运行，则服务器也会通过 systemd 的 API 与其通信以通知其已准备好接受连接。

42 内存限制

```

1  if (server.maxmemory > 0 && server.maxmemory < 1024*1024) {
2      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);
3  }

```

这段代码是在检查服务器是否设置了最大内存限制，如果设置了限制但限制值小于 1MB，会输出一个警告信息。警告信息中会显示当前设置的最大内存限制值。

其中, `server.maxmemory` 表示设置的最大内存限制值, 单位为字节。在这里, 用 `1024*1024` 表示 1MB 的字节数。如果 `server.maxmemory` 小于 1MB 的字节数, 就会输出警告信息。输出信息使用了 `serverLog()` 函数, 以 `LL_WARNING` 级别记录在日志中。

43 CPU亲和性

```
1 | redisSetCpuAffinity(server.server_cpulist);
```

`redisSetCpuAffinity` 是 Redis 中的一个函数, 用于设置 Redis 进程的 CPU 亲和性, 即指定进程在哪些 CPU 上运行。

在调用该函数之前, Redis 会从配置文件中读取 `server_cpulist` 参数, 该参数的值是一个以逗号分隔的 CPU 核心列表, 例如 `"0,1,2,3"`。如果该参数为空, 则表示不设置 CPU 亲和性, Redis 进程可以在任何 CPU 上运行。

在 Linux 系统中, CPU 亲和性可以通过 `sched_setaffinity` 系统调用来实现。Redis 通过该系统调用将进程绑定到指定的 CPU 核心。具体来说, `redisSetCpuAffinity` 函数会先检查当前系统是否支持 `sched_setaffinity` 系统调用, 如果支持则根据 `server_cpulist` 参数的值调用 `sched_setaffinity` 函数, 将 Redis 进程绑定到指定的 CPU 核心上。如果不支持, 则该函数不执行任何操作。

设置 CPU 亲和性可以提高 Redis 的性能, 因为它可以避免进程在不同的 CPU 核心之间频繁切换, 从而减少上下文切换的开销。另外, 设置 CPU 亲和性还可以避免 CPU 缓存的失效, 从而进一步提高 Redis 的性能。

44 Redis在内存不足的稳定性

```
1 | setOOMScoreAdj(-1);
```

`setOOMScoreAdj(-1)` 是 Redis 在启动时设置进程的 OOM Score (out-of-memory score) 的值为 -1, 这个值表示系统不应该轻易地杀死这个进程。OOM Score 是 Linux 内核用来确定一个进程在内存不足的情况下是否应该被杀死的一个指标, 它的值越低, 表示这个进程越不容易被杀死。在 Redis 中, 通过设置 OOM Score 为 -1, 可以使得 Redis 进程在系统内存不足的情况下更难被系统杀死, 从而提高 Redis 的稳定性和可靠性。

45 Redis事件循环

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

```

1 void aeMain(aeEventLoop *eventLoop) {
2     eventLoop->stop = 0;
3     while (!eventLoop->stop) {
4         aeProcessEvents(eventLoop, AE_ALL_EVENTS |
5                         AE_CALL_BEFORE_SLEEP |
6                         AE_CALL_AFTER_SLEEP);
7     }
8 }
9

```

`aeMain(server.el)` 是Redis事件循环的主要函数，它通过循环遍历所有注册的文件事件和时间事件来实现事件的处理。参数 `server.el` 是Redis的事件状态结构体，其中包含了事件循环的状态信息。`aeMain()` 函数会阻塞当前线程，直到有事件就绪，然后调用相应的事件处理函数。在事件处理完毕后，`aeMain()` 函数会继续循环处理其他事件。这个函数在Redis启动后一直运行，直到Redis进程被关闭。

46 END

```

1 aeDeleteEventLoop(server.el);

```

```

1
2 void aeDeleteEventLoop(aeEventLoop *eventLoop) {
3     aeApiFree(eventLoop);
4     zfree(eventLoop->events);
5     zfree(eventLoop->fired);
6
7     /* Free the time events list. */
8     aeTimeEvent *next_te, *te = eventLoop->timeEventHead;
9     while (te) {
10         next_te = te->next;
11         zfree(te);
12         te = next_te;
13     }
14     zfree(eventLoop);
15 }

```

函数 `aeDeleteEventLoop(server.el)` 用于删除事件循环，释放事件循环所占用的内存。

在Redis中，事件循环采用了第三方库 `ae` 提供的事件驱动框架。Redis的事件循环被封装在 `aeEventLoop` 结构体中，而函数 `aeDeleteEventLoop(server.el)` 则用于删除这个结构体。

在Redis中，当进程结束时，会调用 `aeDeleteEventLoop(server.el)` 来释放事件循环所占用的内存。此外，当Redis进程需要进行重启时，也会先调用 `aeDeleteEventLoop(server.el)`，再创建新的事件循环。

需要注意的是，在Redis中，如果开启了AOF持久化功能，那么在Redis进程启动时，也会调用函数 `aeDeleteEventLoop(server.el)` 来删除旧的事件循环，然后重新创建新的事件循环。这个过程是在函数 `redisServeCommands()` 中完成的。

47 总结

Redis服务器启动过程大致可以分为以下几步：

1. 初始化服务器的各个配置选项、日志系统和随机数生成器等，加载服务器所需的所有模块和库，并执行模块的初始化操作。
2. 加载服务器的数据，包括配置文件中指定的数据、AOF文件、RDB文件等。
3. 检查服务器是否在集群模式下运行，并进行相关的配置检查。
4. 设置服务器的CPU亲和性，使其运行在指定的CPU上。
5. 设置服务器的OOM分数，以控制内存使用。
6. 如果服务器是在supervised模式下启动的，与systemd通信以报告服务器已准备好接受连接。
7. 如果开启了maxmemory选项，检查其值是否大于1MB。
8. 启动事件循环，开始处理客户端请求和其他事件。
9. 在服务器关闭时，清理资源并释放内存。

其中，步骤3和步骤8是比较关键的，因为它们涉及到了Redis的核心功能。在检查服务器是否在集群模式下运行时，如果服务器配置有误，将会强制退出。在启动事件循环后，服务器将开始监听客户端请求，并根据请求类型调用相应的处理函数进行处理，最终返回响应结果。在事件循环过程中，如果遇到了其他事件（如定时任务等），也会根据事件类型调用相应的处理函数进行处理。

后面呢，我还会继续向大家详细的介绍里面重要的函数。希望大家继续关注哦。