二 初始化Server

hello，又到了本期的博客了，这一期我将会给大家介绍启动时redis是如何初始化网络状态的，大家一起快乐的学习吧！！

先看一看初始化server在main函数被调用的代码：

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

当将配置文件加载到全局变量server中时，这时redis就会根据配置文件中的内容去初始化服务端状态了，server在全局中的定义如下：

```
/* Global vars */
struct redisServer server; /* Server global state */
```

接下来我们来看看初始化server具体干了哪些事情。

1 信号

```
    signal(SIGHUP, SIG_IGN);
    signal(SIGPIPE, SIG_IGN);
    setupSignalHandlers();
    makeThreadKillable();
```

> `signal(SIGHUP, SIG_IGN)`用于`忽略SIGHUP信号`，它是一种在Unix和类Unix操作系统中广泛使用的信号。在Unix系统中，当控制终端挂起时，会向进程组中的所有进程发送SIGHUP信号，通常用于重新初始化进程。通常情况下，当一个进程接收到SIGHUP信号时，它会终止执行，但使用`signal(SIGHUP,`

> `SIG_IGN)`可以忽略这个信号，从而避免进程被终止。在上面的代码中，当Redis服务器接收到SIGHUP信号时，它不会做任何事情。
>
> `signal(SIGPIPE, SIG_IGN)`的作用是`忽略对于管道/套接字等读取端已经关闭的写入操作而产生的`SIGPIPE`信号`。在使用网络套接字进行通信时，如果对方断开连接，而当前套接字仍然在写数据，那么就会产生`SIGPIPE`信号，程序默认情况下会退出。通过将`SIGPIPE`信号的处理函数设置为`SIG_IGN`，程序将忽略该信号，避免程序异常退出。

```
1
2  void setupSignalHandlers(void) {
3      struct sigaction act;
4
5      /* When the SA_SIGINFO flag is set in sa_flags then
   sa_sigaction is used.
6       * Otherwise, sa_handler is used. */
7      sigemptyset(&act.sa_mask);
8      act.sa_flags = 0;
9      act.sa_handler = sigShutdownHandler;
10     sigaction(SIGTERM, &act, NULL);
11     sigaction(SIGINT, &act, NULL);
12
13     sigemptyset(&act.sa_mask);
14     act.sa_flags = SA_NODEFER | SA_RESETHAND | SA_SIGINFO;
15     act.sa_sigaction = sigsegvHandler;
16     if(server.crashlog_enabled) {
17         sigaction(SIGSEGV, &act, NULL);
18         sigaction(SIGBUS, &act, NULL);
19         sigaction(SIGFPE, &act, NULL);
20         sigaction(SIGILL, &act, NULL);
21         sigaction(SIGABRT, &act, NULL);
22     }
23     return;
24 }
```

`setupSignalHandlers()`函数用于设置Redis进程的信号处理程序。在Unix/Linux系统中，信号是一种异步通信机制，用于处理进程之间的异步事件。在Redis中，有多种信号可以被处理，比如`SIGTERM`表示终止进程，`SIGINT`表示中断进程等。通过设置信号处理程序，Redis可以对这些异步事件做出响应，例如在`SIGTERM`信号到来时进行清理工作并优雅地关闭Redis进程。

首先，使用 `sigemptyset` 函数初始化一个信号集，将其存储在 `act.sa_mask` 中，以便在信号处理器运行时阻塞其他信号。然后将 `act.sa_flags` 设置为 0，表明使用默认行为，即不使用 `SA_RESTART` 重新启动系统调用。最后，将 `act.sa_handler` 设置为 `sigShutdownHandler` 函数，表示在接收到信号时调用 `sigShutdownHandler` 函数进行处理。此函数用于处理 `SIGTERM` 和 `SIGINT` 信号，以使 Redis 正常退出。

首先，`sigemptyset(&act.sa_mask)` 会将 `act.sa_mask` 初始化为空集，这是为了在信号处理函数运行期间防止进程收到其他信号。

接着，`act.sa_flags = SA_NODEFER | SA_RESETHAND | SA_SIGINFO;` 设置了信号处理选项。其中，`SA_NODEFER` 表示在信号处理函数执行期间禁止该信号被阻塞，`SA_RESETHAND` 表示在信号处理函数执行完毕后将该信号处理方式重置为默认值，`SA_SIGINFO` 表示使用带有三个参数的 `sa_sigaction` 处理函数。

最后，`act.sa_sigaction = sigsegvHandler` 设置了信号处理函数。如果进程收到SIGSEGV信号，就会调用 `sigsegvHandler` 函数。

捕获SIGSEGV、SIGBUS、SIGFPE、SIGILL和SIGABRT信号。如果服务器的crashlog_enabled选项设置为true，则向这些信号注册sigsegvHandler()函数作为信号处理程序。如果这些信号被触发，它们将调用sigsegvHandler()函数。此函数是Redis的默认崩溃日志记录器，它将记录崩溃时的上下文信息并尝试将其写入服务器的日志文件。

```
makeThreadKillable()
```

`makeThreadKillable()` 函数的作用是将当前线程标记为可被取消的状态。在使用线程时，当线程处于某些关键代码段时，如果接收到取消请求，可能会造成资源泄漏等问题，因此需要将线程标记为可被取消的状态，以便在接收到取消请求时，线程可以安全地停止执行。此函数通常在执行长时间操作的线程中使用，例如执行文件I/O或网络I/O的线程。

2 日志

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

```

这段代码的作用是在启用syslog的情况下，调用openlog函数打开syslog服务，并设置syslog_ident、syslog_facility等参数。syslog是一个系统日志服务，用于记录操作系统或者应用程序的日志信息。在Redis中，如果启用了syslog，Redis就会把一些重要的日志信息输出到syslog中，以方便用户查看和分析。openlog是syslog服务提供的一个函数，用于打开syslog服务，并设置相关参数。参数LOG_PID表示在每条日志信息中加入当前进程ID，参数LOG_NDELAY表示打开syslog服务时不会等待，参数LOG_NOWAIT表示在写入日志信息时，不会等待syslog进程返回。

3 初始化数据结构和变量

```
  server.aof_state = server.aof_enabled ? AOF_ON : AOF_OFF;
    server.hz = server.config_hz;
    server.pid = getpid();
    server.in_fork_child = CHILD_TYPE_NONE;
    server.main_thread_id = pthread_self();
    server.current_client = NULL;
    server.errors = raxNew();
    server.fixed_time_expire = 0;
    server.in_nested_call = 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.clients_waiting_acks = listCreate();
    server.get_ack_from_slaves = 0;
    server.client_pause_type = CLIENT_PAUSE_OFF;
    server.client_pause_end_time = 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;
     = NULL;
```

这是 Redis 服务器在启动时初始化的一些变量和数据结构。以下是每个变量的简要说明：

- server.aof_state: 标志是否启用 AOF（Append Only File）持久化，初始化为启用或禁用状态。
- server.hz: Redis 服务器的运行频率，即每秒执行的事件循环次数。
- server.pid: Redis 服务器的进程 ID。
- server.in_fork_child: 标记当前进程是否为子进程，初始为 NONE。
- server.main_thread_id: Redis 服务器的主线程 ID。
- server.current_client: 当前客户端，初始为 NULL。
- server.errors: 存储 Redis 服务器发生的错误，以及错误发生的次数和时间戳等信息。
- server.fixed_time_expire: 指定某些键的过期时间是否为固定时间。
- server.in_nested_call: 标记 Redis 服务器是否处于嵌套调用状态，初始为 0。
- server.clients: 存储所有已连接的客户端。
- server.clients_index: 用于快速查找客户端。
- server.clients_to_close: 存储待关闭的客户端。
- server.slaves: 存储所有从服务器。
- server.monitors: 存储所有 MONITOR 客户端。
- server.clients_pending_write: 存储需要写入数据的客户端。
- server.clients_pending_read: 存储需要读取数据的客户端。
- server.clients_timeout_table: 存储客户端的超时时间。
- server.replication_allowed: 标志是否允许复制，初始为允许。
- server.slaveseldb: 从服务器的当前数据库，初始化为 -1。
- server.unblocked_clients: 存储非阻塞客户端。
- server.ready_keys: 存储已就绪的键。
- server.tracking_pending_keys: 存储需要追踪的键。
- server.clients_waiting_acks: 存储等待客户端应答的客户端。
- server.get_ack_from_slaves: 标志是否等待从服务器的应答。
- server.client_pause_type: 客户端暂停状态，初始化为未暂停。
- server.client_pause_end_time: 客户端暂停结束时间，初始化为 0。
- server.client_pause_per_purpose: 存储客户端暂停的原因和时间戳。
- server.postponed_clients: 存储已推迟处理的客户端。

- server.events_processed_while_blocked: 存储在阻塞状态下处理的事件数。
- server.system_memory_size: Redis 服务器可用的系统内存大小。
- server.blocked_last_cron: 记录 Redis 服务器最后一次被阻塞的时间。
- server.blocking_op_nesting: 阻塞操作的嵌套深度，初始为 0。
- server.thp_enabled: 是否启用 Transparent Huge Pages。
- server.cluster_drop_packet_filter: 集群节点间通信中过滤器的编号，初始化为 -1。
- server.reply_buffer_peak_reset_time: Redis 回复缓冲区的峰值重置时间
- server.client_mem_usage_buckets跟踪客户端的内存使用情况,如果为`NULL`表示未启用内存使用统计。

4 重置服务器缓冲区

```
void resetReplicationBuffer(void) {
    server.repl_buffer_mem = 0;
    server.repl_buffer_blocks = listCreate();
    listSetFreeMethod(server.repl_buffer_blocks, (void (*)
(void*))zfree);
}
```

resetReplicationBuffer()是Redis服务器中的一个函数，用于重置服务器的复制缓冲区。

复制缓冲区是Redis用来**存储复制操作期间生成的命令的缓冲区**。这些命令会被发送给从服务器，以便它们可以复制主服务器上执行的操作。**在复制期间，如果从服务器断开连接或出现其他错误，那么Redis将重置复制缓冲区以避免数据丢失或混淆。**

5 TLS

```
 if ((server.tls_port || server.tls_replication ||
server.tls_cluster)
                && tlsConfigure(&server.tls_ctx_config) ==
C_ERR) {
        serverLog(LL_WARNING, "Failed to configure TLS. Check
logs for more info.");
        exit(1);
    }
```

这段代码片段是在服务器启动时检查是否需要配置 TLS，并尝试进行配置。如果服务器配置了 TLS 端口、TLS 复制或 TLS 集群，就会调用 `tlsConfigure` 函数进行配置。如果配置失败，会记录一条警告日志并退出服务器。这里使用了 `exit` 函数，因此服务器无法继续运行。

> TLS是Transport Layer Security（传输层安全协议）的缩写，是一种加密协议，用于保护计算机网络通信的安全。它是SSL（Secure Sockets Layer，安全套接字层）的继任者。TLS协议通过对数据进行加密、认证和完整性保护等手段来保证通信的安全性。在网络通信中，常用的应用层协议，如HTTP、SMTP、POP3等，都可以在TLS协议的基础上实现安全通信。

6 创建共享对象

```
1
2  void createSharedObjects(void) {
3      int j;
4
5      /* Shared command responses */
6      shared.crlf = createObject(OBJ_STRING,sdsnew("\r\n"));
7      shared.ok = createObject(OBJ_STRING,sdsnew("+OK\r\n"));
8      shared.emptybulk =
   createObject(OBJ_STRING,sdsnew("$0\r\n\r\n"));
9      shared.czero = createObject(OBJ_STRING,sdsnew(":0\r\n"));
10     shared.cone = createObject(OBJ_STRING,sdsnew(":1\r\n"));
11     shared.emptyarray =
   createObject(OBJ_STRING,sdsnew("*0\r\n"));
12     shared.pong =
   createObject(OBJ_STRING,sdsnew("+PONG\r\n"));
13     shared.queued =
   createObject(OBJ_STRING,sdsnew("+QUEUED\r\n"));
14     shared.emptyscan =
   createObject(OBJ_STRING,sdsnew("*2\r\n$1\r\n0\r\n*0\r\n"));
15     shared.space = createObject(OBJ_STRING,sdsnew(" "));
16     shared.plus = createObject(OBJ_STRING,sdsnew("+"));
17
18     /* Shared command error responses */
19     shared.wrongtypeerr = createObject(OBJ_STRING,sdsnew(
20         "-WRONGTYPE Operation against a key holding the wrong
   kind of value\r\n"));
21     shared.err = createObject(OBJ_STRING,sdsnew("-ERR\r\n"));
22     shared.nokeyerr = createObject(OBJ_STRING,sdsnew(
23         "-ERR no such key\r\n"));
24     shared.syntaxerr = createObject(OBJ_STRING,sdsnew(
25         "-ERR syntax error\r\n"));
```

```
    shared.sameobjecterr = createObject(OBJ_STRING,sdsnew(
        "-ERR source and destination objects are the
same\r\n"));
    shared.outofrangeerr = createObject(OBJ_STRING,sdsnew(
        "-ERR index out of range\r\n"));
    shared.noscripterr = createObject(OBJ_STRING,sdsnew(
        "-NOSCRIPT No matching script. Please use
EVAL.\r\n"));
    shared.loadingerr = createObject(OBJ_STRING,sdsnew(
        "-LOADING Redis is loading the dataset in
memory\r\n"));
    shared.slowevalerr = createObject(OBJ_STRING,sdsnew(
        "-BUSY Redis is busy running a script. You can only
call SCRIPT KILL or SHUTDOWN NOSAVE.\r\n"));
    shared.slowscripterr = createObject(OBJ_STRING,sdsnew(
        "-BUSY Redis is busy running a script. You can only
call FUNCTION KILL or SHUTDOWN NOSAVE.\r\n"));
    shared.slowmoduleerr = createObject(OBJ_STRING,sdsnew(
        "-BUSY Redis is busy running a module
command.\r\n"));
    shared.masterdownerr = createObject(OBJ_STRING,sdsnew(
        "-MASTERDOWN Link with MASTER is down and replica-
serve-stale-data is set to 'no'.\r\n"));
    shared.bgsaveerr = createObject(OBJ_STRING,sdsnew(
        "-MISCONF Redis is configured to save RDB snapshots,
but it's currently unable to persist to disk. Commands that
may modify the data set are disabled, because this instance
is configured to report errors during writes if RDB
snapshotting fails (stop-writes-on-bgsave-error option).
Please check the Redis logs for details about the RDB
error.\r\n"));
    shared.roslaveerr = createObject(OBJ_STRING,sdsnew(
        "-READONLY You can't write against a read only
replica.\r\n"));
    shared.noautherr = createObject(OBJ_STRING,sdsnew(
        "-NOAUTH Authentication required.\r\n"));
    shared.oomerr = createObject(OBJ_STRING,sdsnew(
        "-OOM command not allowed when used memory >
'maxmemory'.\r\n"));
    shared.execaborterr = createObject(OBJ_STRING,sdsnew(
        "-EXECABORT Transaction discarded because of previous
errors.\r\n"));
    shared.noreplicaserr = createObject(OBJ_STRING,sdsnew(
        "-NOREPLICAS Not enough good replicas to
write.\r\n"));
```

```
    shared.busykeyerr = createObject(OBJ_STRING,sdsnew(
        "-BUSYKEY Target key name already exists.\r\n"));

    /* The shared NULL depends on the protocol version. */
    shared.null[0] = NULL;
    shared.null[1] = NULL;
    shared.null[2] =
createObject(OBJ_STRING,sdsnew("$-1\r\n"));
    shared.null[3] =
createObject(OBJ_STRING,sdsnew("_\r\n"));

    shared.nullarray[0] = NULL;
    shared.nullarray[1] = NULL;
    shared.nullarray[2] =
createObject(OBJ_STRING,sdsnew("*-1\r\n"));
    shared.nullarray[3] =
createObject(OBJ_STRING,sdsnew("_\r\n"));

    shared.emptymap[0] = NULL;
    shared.emptymap[1] = NULL;
    shared.emptymap[2] =
createObject(OBJ_STRING,sdsnew("*0\r\n"));
    shared.emptymap[3] =
createObject(OBJ_STRING,sdsnew("%0\r\n"));

    shared.emptyset[0] = NULL;
    shared.emptyset[1] = NULL;
    shared.emptyset[2] =
createObject(OBJ_STRING,sdsnew("*0\r\n"));
    shared.emptyset[3] =
createObject(OBJ_STRING,sdsnew("~0\r\n"));

    for (j = 0; j < PROTO_SHARED_SELECT_CMDS; j++) {
        char dictid_str[64];
        int dictid_len;

        dictid_len =
ll2string(dictid_str,sizeof(dictid_str),j);
        shared.select[j] = createObject(OBJ_STRING,
            sdscatprintf(sdsempty(),
                "*2\r\n$6\r\nSELECT\r\n$%d\r\n%s\r\n",
                dictid_len, dictid_str));
    }
    shared.messagebulk =
createStringObject("$7\r\nmessage\r\n",13);
```

```
    shared.pmessagebulk =
createStringObject("$8\r\npmessage\r\n",14);
    shared.subscribebulk =
createStringObject("$9\r\nsubscribe\r\n",15);
    shared.unsubscribebulk =
createStringObject("$11\r\nunsubscribe\r\n",18);
    shared.ssubscribebulk =
createStringObject("$10\r\nssubscribe\r\n", 17);
    shared.sunsubscribebulk =
createStringObject("$12\r\nsunsubscribe\r\n", 19);
    shared.smessagebulk =
createStringObject("$8\r\nsmessage\r\n", 14);
    shared.psubscribebulk =
createStringObject("$10\r\npsubscribe\r\n",17);
    shared.punsubscribebulk =
createStringObject("$12\r\npunsubscribe\r\n",19);

    /* Shared command names */
    shared.del = createStringObject("DEL",3);
    shared.unlink = createStringObject("UNLINK",6);
    shared.rpop = createStringObject("RPOP",4);
    shared.lpop = createStringObject("LPOP",4);
    shared.lpush = createStringObject("LPUSH",5);
    shared.rpoplpush = createStringObject("RPOPLPUSH",9);
    shared.lmove = createStringObject("LMOVE",5);
    shared.blmove = createStringObject("BLMOVE",6);
    shared.zpopmin = createStringObject("ZPOPMIN",7);
    shared.zpopmax = createStringObject("ZPOPMAX",7);
    shared.multi = createStringObject("MULTI",5);
    shared.exec = createStringObject("EXEC",4);
    shared.hset = createStringObject("HSET",4);
    shared.srem = createStringObject("SREM",4);
    shared.xgroup = createStringObject("XGROUP",6);
    shared.xclaim = createStringObject("XCLAIM",6);
    shared.script = createStringObject("SCRIPT",6);
    shared.replconf = createStringObject("REPLCONF",8);
    shared.pexpireat = createStringObject("PEXPIREAT",9);
    shared.pexpire = createStringObject("PEXPIRE",7);
    shared.persist = createStringObject("PERSIST",7);
    shared.set = createStringObject("SET",3);
    shared.eval = createStringObject("EVAL",4);

    /* Shared command argument */
    shared.left = createStringObject("left",4);
    shared.right = createStringObject("right",5);
```

```
126     shared.pxat = createStringObject("PXAT", 4);
127     shared.time = createStringObject("TIME",4);
128     shared.retrycount = createStringObject("RETRYCOUNT",10);
129     shared.force = createStringObject("FORCE",5);
130     shared.justid = createStringObject("JUSTID",6);
131     shared.entriesread =
    createStringObject("ENTRIESREAD",11);
132     shared.lastid = createStringObject("LASTID",6);
133     shared.default_username =
    createStringObject("default",7);
134     shared.ping = createStringObject("ping",4);
135     shared.setid = createStringObject("SETID",5);
136     shared.keepttl = createStringObject("KEEPTTL",7);
137     shared.absttl = createStringObject("ABSTTL",6);
138     shared.load = createStringObject("LOAD",4);
139     shared.createconsumer =
    createStringObject("CREATECONSUMER",14);
140     shared.getack = createStringObject("GETACK",6);
141     shared.special_asterick = createStringObject("*",1);
142     shared.special_equals = createStringObject("=",1);
143     shared.redacted = makeObjectShared(createStringObject("
    (redacted)",10));
144
145     for (j = 0; j < OBJ_SHARED_INTEGERS; j++) {
146         shared.integers[j] =
147             makeObjectShared(createObject(OBJ_STRING,(void*)
    (long)j));
148         shared.integers[j]->encoding = OBJ_ENCODING_INT;
149     }
150     for (j = 0; j < OBJ_SHARED_BULKHDR_LEN; j++) {
151         shared.mbulkhdr[j] = createObject(OBJ_STRING,
152             sdscatprintf(sdsempty(),"*%d\r\n",j));
153         shared.bulkhdr[j] = createObject(OBJ_STRING,
154             sdscatprintf(sdsempty(),"$%d\r\n",j));
155         shared.maphdr[j] = createObject(OBJ_STRING,
156             sdscatprintf(sdsempty(),"%%%d\r\n",j));
157         shared.sethdr[j] = createObject(OBJ_STRING,
158             sdscatprintf(sdsempty(),"~%d\r\n",j));
159     }
160     /* The following two shared objects, minstring and
    maxstring, are not
161      * actually used for their value but as a special object
    meaning
162      * respectively the minimum possible string and the
    maximum possible
```

```
     * string in string comparisons for the ZRANGEBYLEX
command. */
    shared.minstring = sdsnew("minstring");
    shared.maxstring = sdsnew("maxstring");
}
```

```
/* Our shared "common" objects */
struct sharedObjectsStruct shared;
struct sharedObjectsStruct {
    robj *crlf, *ok, *err, *emptybulk, *czero, *cone, *pong,
*space,
    *queued, *null[4], *nullarray[4], *emptymap[4],
*emptyset[4],
    *emptyarray, *wrongtypeerr, *nokeyerr, *syntaxerr,
*sameobjecterr,
    *outofrangeerr, *noscripterr, *loadingerr,
    *slowevalerr, *slowscripterr, *slowmoduleerr, *bgsaveerr,
    *masterdownerr, *roslaveerr, *execaborterr, *noautherr,
*noreplicaserr,
    *busykeyerr, *oomerr, *plus, *messagebulk, *pmessagebulk,
*subscribebulk,
    *unsubscribebulk, *psubscribebulk, *punsubscribebulk,
*del, *unlink,
    *rpop, *lpop, *lpush, *rpoplpush, *lmove, *blmove,
*zpopmin, *zpopmax,
    *emptyscan, *multi, *exec, *left, *right, *hset, *srem,
*xgroup, *xclaim,
    *script, *replconf, *eval, *persist, *set, *pexpireat,
*pexpire,
    *time, *pxat, *absttl, *retrycount, *force, *justid,
*entriesread,
    *lastid, *ping, *setid, *keepttl, *load, *createconsumer,
    *getack, *special_asterick, *special_equals,
*default_username, *redacted,
    *ssubscribebulk,*sunsubscribebulk, *smessagebulk,
    *select[PROTO_SHARED_SELECT_CMDS],
    *integers[OBJ_SHARED_INTEGERS],
    *mbulkhdr[OBJ_SHARED_BULKHDR_LEN], /* "*<value>\r\n" */
    *bulkhdr[OBJ_SHARED_BULKHDR_LEN],  /* "$<value>\r\n" */
    *maphdr[OBJ_SHARED_BULKHDR_LEN],   /* "%<value>\r\n" */
    *sethdr[OBJ_SHARED_BULKHDR_LEN];   /* "~<value>\r\n" */
    sds minstring, maxstring;
};
```

`struct sharedObjectsStruct shared;` 定义了一个名为 `shared` 的结构体变量，该结构体用于存储 Redis 中一些共享的字符串对象，如 `OK`、`ERR` 等。这些共享对象在 Redis 内部被广泛使用，可以减少内存占用和提高性能。
`createSharedObjects()` 函数则是初始化这些共享对象。

共享对象是Redis中已经预先创建好的一些固定字符串对象，它们的值已经提前在Redis启动时被设置好了。由于这些字符串对象是共享的，可以被多个客户端共同使用，所以它们在Redis内存中只有一份拷贝，这样就可以节省很多内存空间。

在Redis中，共享对象是由struct sharedObjectsStruct结构体中的成员变量表示的。这个结构体包含了很多的共享对象，例如空字符串对象、OK字符串对象、错误信息对象、空列表对象、空哈希对象等等。在Redis启动时，会调用createSharedObjects()函数创建这些共享对象，并将它们存储在sharedObjectsStruct结构体中，以便后续使用。

使用共享对象可以提高Redis的性能和内存使用效率，因为它们不需要在每次使用时重新创建，而是可以直接引用。

7 文件描述符限制

```
1  void adjustOpenFilesLimit(void) {
2      rlim_t maxfiles =
   server.maxclients+CONFIG_MIN_RESERVED_FDS;
3      struct rlimit limit;
4
5      if (getrlimit(RLIMIT_NOFILE,&limit) == -1) {
6          serverLog(LL_WARNING,"Unable to obtain the current
   NOFILE limit (%s), assuming 1024 and setting the max clients
   configuration accordingly.",
7              strerror(errno));
8          server.maxclients = 1024-CONFIG_MIN_RESERVED_FDS;
9      } else {
10         rlim_t oldlimit = limit.rlim_cur;
11
12         /* Set the max number of files if the current limit is
   not enough
13          * for our needs. */
14         if (oldlimit < maxfiles) {
15             rlim_t bestlimit;
16             int setrlimit_error = 0;
17
18             /* Try to set the file limit to match 'maxfiles'
   or at least
19              * to the higher value supported less than
   maxfiles. */
```

```
            bestlimit = maxfiles;
            while(bestlimit > oldlimit) {
                rlim_t decr_step = 16;

                limit.rlim_cur = bestlimit;
                limit.rlim_max = bestlimit;
                if (setrlimit(RLIMIT_NOFILE,&limit) != -1)
break;
                setrlimit_error = errno;

                /* We failed to set file limit to 'bestlimit'.
Try with a
                 * smaller limit decrementing by a few FDs per
iteration. */
                if (bestlimit < decr_step) {
                    bestlimit = oldlimit;
                    break;
                }
                bestlimit -= decr_step;
            }

            /* Assume that the limit we get initially is still
valid if
             * our last try was even lower. */
            if (bestlimit < oldlimit) bestlimit = oldlimit;

            if (bestlimit < maxfiles) {
                unsigned int old_maxclients =
server.maxclients;
                server.maxclients = bestlimit-
CONFIG_MIN_RESERVED_FDS;
                /* maxclients is unsigned so may overflow: in
order
                 * to check if maxclients is now logically
less than 1
                 * we test indirectly via bestlimit. */
                if (bestlimit <= CONFIG_MIN_RESERVED_FDS) {
                    serverLog(LL_WARNING,"Your current 'ulimit
-n' "
                        "of %llu is not enough for the server
to start. "
                        "Please increase your open file limit
to at least "
                        "%llu. Exiting.",
                        (unsigned long long) oldlimit,
```

```
                        (unsigned long long) maxfiles);
                    exit(1);
                }
                serverLog(LL_WARNING,"You requested maxclients of %d "
                    "requiring at least %llu max file descriptors.",
                    old_maxclients,
                    (unsigned long long) maxfiles);
                serverLog(LL_WARNING,"Server can't set maximum open files "
                    "to %llu because of OS error: %s.",
                    (unsigned long long) maxfiles, strerror(setrlimit_error));
                serverLog(LL_WARNING,"Current maximum open files is %llu. "
                    "maxclients has been reduced to %d to compensate for "
                    "low ulimit. "
                    "If you need higher maxclients increase 'ulimit -n'.",
                    (unsigned long long) bestlimit, server.maxclients);
            } else {
                serverLog(LL_NOTICE,"Increased maximum number of open files "
                    "to %llu (it was originally set to %llu).",
                    (unsigned long long) maxfiles,
                    (unsigned long long) oldlimit);
            }
        }
    }
}

```

`adjustOpenFilesLimit()` 函数用于检查系统对于Redis进程的打开文件数限制，如果系统限制过低，则尝试将Redis进程的文件打开数限制增加到一个更高的值。

这个函数首先会调用 `getrlimit()` 函数获取当前系统对于进程的文件打开数限制，然后计算出 Redis 目前已经打开的文件数和 Redis 配置文件中指定的最大打开文件数之间的较小值。如果当前系统的文件打开数限制比这个值小，则会将 Redis 的文件打开数限制调整为这个值。

在 Linux 系统中，每个进程可以同时打开的文件数是有限制的，这个限制可以通过 `ulimit` 命令来查看和修改。当 Redis 需要同时打开大量的文件（例如在进行 RDB 持久化时），如果当前的文件打开数限制过低，就会导致 Redis 进程无法正常工作。因此，这个函数的作用就是检查文件打开数限制是否过低，并尝试将其增加到一个足够大的值，保证 Redis 进程可以正常工作。

```
    rlim_t maxfiles = server.maxclients+CONFIG_MIN_RESERVED_FDS;
```

这行代码的作用是根据最大客户端数和配置文件中指定的最小保留文件描述符数来计算出当前操作系统允许的最大文件描述符数。在Redis中，每个客户端都会使用一个文件描述符，如果超过了操作系统允许的最大文件描述符数，就会出现文件描述符耗尽的情况，导致Redis无法正常工作。因此，这个函数的作用是调整Redis进程的文件描述符限制，确保Redis进程能够支持足够数量的客户端连接。

```
 if (getrlimit(RLIMIT_NOFILE,&limit) == -1) {
        serverLog(LL_WARNING,"Unable to obtain the current NOFILE limit (%s), assuming 1024 and setting the max clients configuration accordingly.",
            strerror(errno));
        server.maxclients = 1024-CONFIG_MIN_RESERVED_FDS;
 }
```

这段代码是在获取系统打开文件描述符限制失败时，设置服务器的最大客户端数量为1024减去一些保留文件描述符的数量。它使用了getrlimit系统调用来获取当前的限制，并将其存储在limit结构中。如果getrlimit调用失败，则会打印一条警告消息，并将server.maxclients设置为默认值1024-CONFIG_MIN_RESERVED_FDS。这是为了确保服务器不会超出系统限制而崩溃。

8 初始化时钟

```
const char * monotonicInit() {
    #if defined(USE_PROCESSOR_CLOCK) && defined(__x86_64__) &&
defined(__linux__)
    if (getMonotonicUs == NULL) monotonicInit_x86linux();
    #endif

    #if defined(USE_PROCESSOR_CLOCK) && defined(__aarch64__)
    if (getMonotonicUs == NULL) monotonicInit_aarch64();
    #endif

    if (getMonotonicUs == NULL) monotonicInit_posix();

    return monotonic_info_string;
}
```

`monotonicInit()` 函数是 Redis 在启动时初始化时钟的函数，用于确保 Redis 在不同的操作系统上都能正确使用单调时钟（monotonic clock）。该函数返回一个字符串指针，指向一个字符串，用于记录时钟初始化的详细信息。在日志记录和调试时可能会用到这个信息。

函数首先通过宏定义判断当前环境是否支持`USE_PROCESSOR_CLOCK`，如果是x86_64架构的Linux系统，则会调用`monotonicInit_x86linux()`函数进行初始化；如果是aarch64架构，则调用`monotonicInit_aarch64()`进行初始化；否则，调用`monotonicInit_posix()`进行初始化。最终返回一个字符串类型的单调递增时间戳信息。

9 核心组件-事件循环

```
 server.el =
aeCreateEventLoop(server.maxclients+CONFIG_FDSET_INCR);
```

这行代码创建了一个事件循环（event loop），并将其赋值给了 Redis 服务器状态中的 `el` 变量。事件循环是 Redis 服务器的核心组件之一，它会监听一组文件描述符上的事件（例如可读、可写或异常事件），并在这些事件发生时调用相应的回调函数。事件循环通常由系统提供的 I/O 多路复用机制来实现，例如 Linux 中的 `epoll` 或 BSD 中的 `kqueue`。在 Redis 中，事件循环的具体实现是由第三方库 `ae` 提供的。

`aeCreateEventLoop` 函数的参数 `server.maxclients+CONFIG_FDSET_INCR` 表示要为事件循环分配的文件描述符集合（fdset）的大小。文件描述符集合是事件循环的重要组成部分，它用于记录事件循环要监听的文件描述符。在 Redis 中，fdset 的大小默认为 `FD_SETSIZE`，通常是 1024。但由于 Redis 的客户端连接数可能会很大，因此在这里使用了 `server.maxclients+CONFIG_FDSET_INCR` 来扩大

fdset 的大小，以便可以同时监听更多的文件描述符。其中，`server.maxclients` 是 Redis 服务器配置文件中设置的最大客户端连接数，`CONFIG_FDSET_INCR` 是 Redis 配置文件中的另一个参数，用于控制文件描述符集合的增量大小，它的默认值是 32。因此，`server.maxclients+CONFIG_FDSET_INCR` 表示将最大客户端连接数与增量大小相加，作为事件循环所需的 fdset 大小。

该函数时redis事件分发的核心函数，在后面我会详细的剖析她。

10 分配数据库数组

```
server.db = zmalloc(sizeof(redisDb)*server.dbnum);
```

这行代码是在Redis服务器启动时分配了一个`redisDb`类型的数组，用于存储Redis数据库的相关信息。

在Redis服务器中，每个数据库对应一个`redisDb`结构体，其中包含了该数据库中所有的键值对信息，以及一些其他的相关配置信息，如过期时间等。

`server.dbnum`变量是在`redis.conf`配置文件中指定的，表示Redis服务器中要使用的数据库数量。这行代码通过调用`zmalloc`函数分配了一个长度为`server.dbnum`的`redisDb`数组，用于存储所有的Redis数据库。这些数据库在服务器启动时被创建，并在内存中持久化存储，以便随时提供快速的数据读写操作。

值得注意的是，每个`redisDb`结构体中都有一个`dict`类型的成员变量，用于存储键值对信息。Redis的字典数据结构在内存中以哈希表的形式实现，提供了非常高效的键值对存储和查找操作。

11 设置监听端口

```c
if (server.port != 0 &&
        listenToPort(server.port,&server.ipfd) == C_ERR) {
        /* Note: the following log text is matched by the test
suite. */
        serverLog(LL_WARNING, "Failed listening on port %u
(TCP), aborting.", server.port);
        exit(1);
    }
    if (server.tls_port != 0 &&
        listenToPort(server.tls_port,&server.tlsfd) == C_ERR)
{
        /* Note: the following log text is matched by the test
suite. */
        serverLog(LL_WARNING, "Failed listening on port %u
(TLS), aborting.", server.tls_port);
        exit(1);
    }
```

这段代码是用来监听 Redis 服务器的 TCP 和 TLS 端口的。如果监听失败，则会记录日志并退出程序。具体来说，它首先检查服务器配置中的 `port` 和 `tls_port` 是否为零，如果不为零，则分别调用 `listenToPort` 函数来监听对应的端口，并将对应的文件描述符分别存储在 `server.ipfd` 和 `server.tlsfd` 中。如果监听失败，则记录一条日志，并通过调用 `exit` 函数退出程序。

```c
int listenToPort(int port, socketFds *sfd) {
    int j;
    char **bindaddr = server.bindaddr;

    /* If we have no bind address, we don't listen on a TCP
socket */
    if (server.bindaddr_count == 0) return C_OK;

    for (j = 0; j < server.bindaddr_count; j++) {
        char* addr = bindaddr[j];
        int optional = *addr == '-';
        if (optional) addr++;
        if (strchr(addr,':')) {
            /* Bind IPv6 address. */
            sfd->fd[sfd->count] =
anetTcp6Server(server.neterr,port,addr,server.tcp_backlog);
        } else {
            /* Bind IPv4 address. */
            sfd->fd[sfd->count] =
anetTcpServer(server.neterr,port,addr,server.tcp_backlog);
```

```
        }
        if (sfd->fd[sfd->count] == ANET_ERR) {
            int net_errno = errno;
            serverLog(LL_WARNING,
                "Warning: Could not create server TCP listening socket %s:%d: %s",
                addr, port, server.neterr);
            if (net_errno == EADDRNOTAVAIL && optional)
                continue;
            if (net_errno == ENOPROTOOPT     || net_errno == EPROTONOSUPPORT ||
                net_errno == ESOCKTNOSUPPORT || net_errno == EPFNOSUPPORT ||
                net_errno == EAFNOSUPPORT)
                continue;

            /* Rollback successful listens before exiting */
            closeSocketListeners(sfd);
            return C_ERR;
        }
        if (server.socket_mark_id > 0) anetSetSockMarkId(NULL, sfd->fd[sfd->count], server.socket_mark_id);
        anetNonBlock(NULL,sfd->fd[sfd->count]);
        anetCloexec(sfd->fd[sfd->count]);
        sfd->count++;
    }
    return C_OK;
}

```

12 本地unix套接字

```
 if (server.unixsocket != NULL) {
        unlink(server.unixsocket); /* don't care if this fails */
        server.sofd = anetUnixServer(server.neterr,server.unixsocket,
            (mode_t)server.unixsocketperm, server.tcp_backlog);
        if (server.sofd == ANET_ERR) {
            serverLog(LL_WARNING, "Failed opening Unix socket: %s", server.neterr);
            exit(1);
        }
        anetNonBlock(NULL,server.sofd);
        anetCloexec(server.sofd);
    }
```

这段代码是在 Redis 服务器启动时，尝试创建一个 Unix 套接字，用于本地通信。具体实现是通过调用 `anetUnixServer` 函数创建一个 Unix 套接字，如果创建失败则会记录错误信息并终止服务器进程。如果创建成功，则会将套接字设置为非阻塞模式并且设置文件描述符标志 `FD_CLOEXEC`。这样，服务器就可以通过 Unix 套接字与本地客户端进行通信了。需要注意的是，如果 Redis 服务器配置文件中没有指定 Unix 套接字路径，则不会执行这段代码。

13 检测套接字

```
  /* Abort if there are no listening sockets at all. */
    if (server.ipfd.count == 0 && server.tlsfd.count == 0 && server.sofd < 0) {
        serverLog(LL_WARNING, "Configured to not listen anywhere, exiting.");
        exit(1);
    }
```

这段代码的作用是检查是否存在正在监听的套接字。如果没有任何一个监听套接字，则记录一个警告日志并退出程序。这是一个安全保护措施，因为如果Redis服务器没有打开任何端口或套接字，那么它将无法提供服务，也无法接受客户端连接。

14 创建Redis数据库

```
 /* Create the Redis databases, and initialize other internal state. */
    for (j = 0; j < server.dbnum; j++) {
        server.db[j].dict = dictCreate(&dbDictType);
```

```
        server.db[j].expires = dictCreate(&dbExpiresDictType);
        server.db[j].expires_cursor = 0;
        server.db[j].blocking_keys =
dictCreate(&keylistDictType);
        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;
        server.db[j].defrag_later = listCreate();
        server.db[j].slots_to_keys = NULL; /* Set by
clusterInit later on if necessary. */
        listSetFreeMethod(server.db[j].defrag_later,(void (*)
(void*))sdsfree);
    }
```

这段代码创建了 Redis 数据库，并初始化了一些内部状态。具体来说，它使用了一个循环来创建指定数量的 Redis 数据库，每个数据库都包含了以下属性：

- dict：用于保存键值对的字典（底层实现为哈希表）；
- expires：用于过期键的字典（底层实现为哈希表），用于实现键的 TTL 功能；
- expires_cursor：游标，用于在过期键字典中定期地删除过期键；
- blocking_keys：用于阻塞操作的键的列表（底层实现为字典）；
- ready_keys：用于保存已准备好的键（底层实现为字典）；
- watched_keys：用于保存被监视的键（底层实现为字典）；
- id：数据库的 ID；
- avg_ttl：键的平均 TTL（Time To Live）值；
- defrag_later：等待进行碎片整理操作的键的列表；
- slots_to_keys：用于保存槽和键之间映射关系的字典（用于 Redis 集群）。

在循环内，还使用了 listCreate() 函数创建了一个 defrag_later 列表，用于保存需要进行碎片整理的键。其中，listSetFreeMethod() 函数用于设置 defrag_later 列表的释放函数，这里使用了 sdsfree() 函数来释放 sds 字符串。

后面会详细的介绍。

15 初始化LRU

```
void evictionPoolAlloc(void) {
    struct evictionPoolEntry *ep;
    int j;

    ep = zmalloc(sizeof(*ep)*EVPOOL_SIZE);
```

```
    for (j = 0; j < EVPOOL_SIZE; j++) {
        ep[j].idle = 0;
        ep[j].key = NULL;
        ep[j].cached = sdsnewlen(NULL,EVPOOL_CACHED_SDS_SIZE);
        ep[j].dbid = 0;
    }
    EvictionPoolLRU = ep;
}


```

`evictionPoolAlloc()` 函数的作用是分配内存来初始化用于 LRU eviction 的 key pool。

具体地，它通过调用 `zmalloc()` 分配 `sizeof(*ep)*EVPOOL_SIZE)` 大小的内存，其中 `ep` 是一个 `struct evictionPoolEntry` 类型的数组。然后它遍历 `ep` 数组的每个元素，并对其进行初始化，将 `idle` 域设置为 0，`key` 域设置为 NULL，`cached` 域使用 `sdsnewlen(NULL,EVPOOL_CACHED_SDS_SIZE)` 函数初始化为空的 sds 字符串，`dbid` 域设置为 0。最后，将 `ep` 赋值给全局变量 `EvictionPoolLRU`。

这个 key pool 用于在 Redis 的 LRU eviction 中缓存一些被淘汰的 key 的信息，以便在需要时可以快速从缓存中获取这些信息而无需重新计算。

16 内部状态初始化

```
server.pubsub_channels = dictCreate(&keylistDictType);
    server.pubsub_patterns = dictCreate(&keylistDictType);
    server.pubsubshard_channels =
dictCreate(&keylistDictType);
    server.cronloops = 0;
    server.in_exec = 0;
    server.busy_module_yield_flags = BUSY_MODULE_YIELD_NONE;
    server.busy_module_yield_reply = NULL;
    server.core_propagates = 0;
    server.propagate_no_multi = 0;
    server.module_ctx_nesting = 0;
    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_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.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;
```

这是Redis服务器在启动时初始化自身的一系列变量和数据结构，其中包括：

- server.pubsub_channels、server.pubsub_patterns、server.pubsubshard_channels：三个字典，用于维护Redis的发布/订阅功能。
- server.cronloops：一个计数器，记录Redis的定时任务已经执行的次数。
- server.in_exec：一个标志，表示是否正在执行Redis的事务操作。
- server.busy_module_yield_flags、server.busy_module_yield_reply：在Redis执行长时间的阻塞操作时，用于指示是否可以中断阻塞操作。
- server.core_propagates、server.propagate_no_multi：用于Redis的主从复制功能。
- server.module_ctx_nesting：一个计数器，记录Redis当前嵌套的模块上下文数量。
- server.client_pause_in_transaction：一个标志，表示是否在Redis事务操作中暂停了客户端。
- server.child_pid、server.child_type、server.rdb_child_type：用于Redis的持久化功能，记录持久化子进程的相关信息。
- server.rdb_pipe_conns、server.rdb_pipe_numconns、server.rdb_pipe_numconns_writing、server.rdb_pipe_buff、server.rdb_pipe_bufflen：用于Redis进行RDB持久化时，缓存数据写入管道的相关信息。
- server.rdb_bgsave_scheduled：一个标志，表示是否已经安排了RDB持久化操作。
- server.child_info_pipe、server.child_info_nread：用于与持久化子进程进行通信。
- server.aof_buf：一个缓冲区，用于Redis进行AOF持久化时，缓存待写入AOF文件的数据。
- server.lastsave、server.lastbgsave_try、server.rdb_save_time_last、server.rdb_save_time_start、server.rdb_last_load_keys_expired、server.rdb_last_load_keys_loaded：用于记录Redis的持久化状态。
- server.dirty：一个计数器，记录Redis的脏键数量。
- server.stat_starttime、server.stat_peak_memory、server.stat_current_cow_peak、server.stat_current_cow_bytes、server.stat_current_cow_updated、server.stat_current_save_keys_processed、server.stat_current_save_keys_total、server.stat_rdb_cow_bytes、server.stat_aof_cow_bytes、server.stat_module_cow_bytes、server.stat_module_progress、server.stat_clients_type_memory、server.stat_cluster_links_memory：一系列统计信息，用于记录Redis的运行状态。
- server.cron_malloc_stats：一个结构体，用于记录Redis的内存使用情况。
- server.lastbgsave_status、server.aof_last_write_status、server.aof_last_write_errno、server.repl_good_slaves_count、

server.last_sig_received：用于Redis的持久化和主从复制功能。

17 时间事件

```
if (aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL) == AE_ERR) {
        serverPanic("Can't create event loop timers.");
        exit(1);
    }
long long aeCreateTimeEvent(aeEventLoop *eventLoop, long long milliseconds,
        aeTimeProc *proc, void *clientData,
        aeEventFinalizerProc *finalizerProc)
{
    long long id = eventLoop->timeEventNextId++;
    aeTimeEvent *te;

    te = zmalloc(sizeof(*te));
    if (te == NULL) return AE_ERR;
    te->id = id;
    te->when = getMonotonicUs() + milliseconds * 1000;
    te->timeProc = proc;
    te->finalizerProc = finalizerProc;
    te->clientData = clientData;
    te->prev = NULL;
    te->next = eventLoop->timeEventHead;
    te->refcount = 0;
    if (te->next)
        te->next->prev = te;
    eventLoop->timeEventHead = te;
    return id;
}
```

这段代码是Redis服务器启动时的一部分，主要功能是创建一个定时器事件，并将其添加到事件循环中。这个定时器事件每秒会执行一次serverCron函数，用于执行一些周期性的任务，例如检查过期键值对、清理过期数据等。

如果创建定时器事件失败（返回AE_ERR），那么服务器将调用serverPanic函数进入崩溃状态，并退出程序。这种情况应该极为罕见，通常是由于系统资源不足或其他严重问题导致的。

`aeCreateTimeEvent` 函数用于在 `eventLoop` 上创建一个指定时间间隔后触发的时间事件。

参数解释如下：

- `eventLoop`：事件循环。
- `milliseconds`：指定时间间隔，单位是毫秒。
- `proc`：事件处理函数。
- `clientData`：事件处理函数的参数。
- `finalizerProc`：事件结束时执行的函数。

函数的返回值是时间事件的 ID。如果创建事件失败，返回 AE_ERR。

该函数先为时间事件分配内存，然后初始化各个字段。其中，`when` 字段表示事件的触发时间，是当前时间加上指定的时间间隔。然后将事件加入事件循环的时间事件链表 `timeEventHead` 中，并返回时间事件的 ID。

18 监听TCP连接请求

```
1  if (createSocketAcceptHandler(&server.ipfd, acceptTcpHandler)
!= C_OK) {
2          serverPanic("Unrecoverable error creating TCP socket
accept handler.");
3      }
```

这段代码是创建一个监听TCP连接请求的套接字，并将该套接字上的事件添加到事件循环中，以便在有新的连接请求到达时触发相应的处理函数 `acceptTcpHandler()`。

具体来说，`createSocketAcceptHandler()` 函数会创建一个套接字，将其绑定到指定的 IP 地址和端口上，并将其设置为监听状态。然后，它将该套接字上的可读事件添加到事件循环中，当该事件被触发时，事件循环会调用相应的处理函数 `acceptTcpHandler()`，来接受新的连接请求并进行处理

```c
int createSocketAcceptHandler(socketFds *sfd, aeFileProc
*accept_handler) {
    int j;

    for (j = 0; j < sfd->count; j++) {
        if (aeCreateFileEvent(server.el, sfd->fd[j],
AE_READABLE, accept_handler,NULL) == AE_ERR) {
            /* Rollback */
            for (j = j-1; j >= 0; j--)
aeDeleteFileEvent(server.el, sfd->fd[j], AE_READABLE);
            return C_ERR;
        }
    }
    return C_OK;
}

```

该函数的作用是创建用于监听 socket 连接请求的事件处理器，以及将处理器注册到事件循环中。该函数的输入参数 `sfd` 是一个结构体，用于保存多个监听 socket 的文件描述符。函数还接受一个名为 `accept_handler` 的回调函数指针作为输入参数，用于处理新连接的请求。

在函数内部，使用了一个 for 循环遍历 `sfd->fd` 数组中的每个文件描述符，将每个文件描述符注册为可读事件，并将 `accept_handler` 回调函数指针作为参数传入注册函数 `aeCreateFileEvent` 中。如果某个文件描述符的注册失败，就会执行 for 循环的回滚操作，删除已经注册的文件描述符事件处理器。最终，如果所有文件描述符都成功注册了事件处理器，则函数返回 `C_OK`（表示执行成功），否则返回 `C_ERR`（表示执行失败）。

`server.ipfd` 是一个 `socketFds` 结构体类型的变量，用于存储服务器监听的 TCP 套接字描述符。这个结构体定义如下：

```c
typedef struct socketFds {
    int *fd; /* 监听套接字描述符数组 */
    int count; /* 监听套接字数量 */
} socketFds;
```

这个结构体包含两个字段，一个是 `fd` 数组用于存储监听套接字描述符，另一个是 `count` 表示监听套接字数量。在服务器启动时，服务器会创建并监听多个 TCP 套接字，并将所有的套接字描述符存储在 `server.ipfd` 变量中，以便后续操作。

19 Unix

```
if (server.sofd > 0 &&
aeCreateFileEvent(server.el,server.sofd,AE_READABLE,
        acceptUnixHandler,NULL) == AE_ERR)
serverPanic("Unrecoverable error creating server.sofd file
event.");
```

这段代码是在创建一个监听 Unix domain socket（AF_UNIX）的文件事件，用于接收客户端的连接请求。具体来说，它使用了 `aeCreateFileEvent` 函数创建一个文件事件，当 `server.sofd` 变为可读时，就会调用 `acceptUnixHandler` 函数处理新的连接请求。如果创建文件事件出错，就会触发服务器宕机，即调用 `serverPanic` 函数。

20 管道

```
 if (aeCreateFileEvent(server.el, server.module_pipe[0],
AE_READABLE,
        modulePipeReadable,NULL) == AE_ERR) {
            serverPanic(
                "Error registering the readable event for the
module pipe.");
    }
```

这段代码是创建一个管道的可读事件，当管道有数据可读时会触发一个回调函数`modulePipeReadable`。

具体来说，`aeCreateFileEvent`函数会向事件循环`server.el`注册一个文件事件，用于监听`server.module_pipe[0]`这个文件描述符的可读事件。如果事件创建失败，那么`serverPanic`函数会输出一条错误信息并终止程序。

在这里，`modulePipeReadable`函数被注册为回调函数，当`server.module_pipe[0]`有数据可读时，事件循环将自动调用它。

21 事件循环睡眠前后

```
aeSetBeforeSleepProc(server.el,beforeSleep);
aeSetAfterSleepProc(server.el,afterSleep);

```

`aeSetBeforeSleepProc`和`aeSetAfterSleepProc`是设置事件循环在进入睡眠前和睡眠后要执行的函数。

`aeSetBeforeSleepProc`用于注册一个在进入事件循环睡眠之前被调用的函数，该函数可以用于在处理事件循环之前执行特定任务。例如，可以在此处调用`redisCheckAofBackgroundWriting()`检查是否需要执行 AOF 文件的后台写入。

`aeSetAfterSleepProc`用于注册一个在事件循环进入睡眠后被调用的函数，该函数可以用于在进入事件循环之前执行特定任务。例如，在此处可以更新 Redis 的内部状态或执行一些清理任务。

这两个函数可以通过`aeMain`函数中的相应调用设置，并在每个事件循环周期中执行。

22 服务器位数

```
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;
    }
```

这段代码是在判断服务器实例的位数是否为32位，如果是32位，则设置一个3GB的最大内存限制，并将最大内存策略设置为不驱逐策略。这是因为32位的服务器进程无法使用大于4GB的内存，因此如果没有设置最大内存限制，则可能导致内存不足并导致服务器崩溃。为了避免这种情况，这里设置一个最大内存限制，以确保服务器进程不会超过3GB的内存使用量。同时，将最大内存策略设置为不驱逐策略，以确保在达到最大内存限制时，不会驱逐任何键值对，而是拒绝写入新的键值对，从而避免因驱逐键值对而导致数据丢失的风险。

23 其他函数

```
if (server.cluster_enabled) clusterInit();
    scriptingInit(1);
    functionsInit();
    slowlogInit();
    latencyMonitorInit();
```

这段代码主要做了以下事情：

1. 如果Redis开启了集群模式，就调用clusterInit()函数进行集群初始化；
2. 调用scriptingInit()函数初始化脚本系统；
3. 调用functionsInit()函数初始化Redis内置函数；
4. 调用slowlogInit()函数初始化慢查询日志系统；
5. 调用latencyMonitorInit()函数初始化延迟监视器。

这些初始化函数的作用是为Redis提供不同的功能和服务，如集群支持、脚本支持、内置函数、慢查询日志和延迟监视器等。通过这些初始化操作，Redis可以更好地服务于用户的不同需求和场景。

24 更新默认用户密码

```
ACLUpdateDefaultUserPassword(server.requirepass);
```

`ACLUpdateDefaultUserPassword` 函数是 Redis 用于更新默认用户密码的函数。在 Redis 中，默认情况下，使用 `requirepass` 参数所设置的密码作为用户认证密码。当 `requirepass` 参数被修改时，如果默认用户已经存在，那么它的密码需要被更新。`ACLUpdateDefaultUserPassword` 函数就是用来更新这个默认用户密码的。

25 看门狗

```
applyWatchdogPeriod()
```

`applyWatchdogPeriod()`函数用于启用或禁用Redis的看门狗程序。看门狗程序是一个定期的任务，用于检查Redis是否处于假死状态，如果是，则通过发送SIGUSR1信号重启Redis进程。

在该函数中，首先会检查配置文件中的`watchdog-period`参数是否被设置为0，如果是，则禁用看门狗程序；否则，根据该参数的值来设置看门狗程序的执行周期。执行周期由`server.watchdog_period`变量来表示，以毫秒为单位。

如果在运行Redis的操作系统中没有实现定时器（例如OpenVZ），则会禁用看门狗程序并打印警告信息。

该函数在Redis服务器启动时被调用。

26 客户端内存限制

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

这段代码是在检查服务器是否设置了客户端的最大内存使用限制（`maxmemory_clients`），如果设置了，就会调用`initServerClientMemUsageBuckets()` 函数来初始化一个用于记录客户端内存使用情况的数据结构。该函数会在 `dict.c` 文件中定义。在启用了客户端内存限制后，服务器会定期检查客户端的内存使用情况，并在客户端使用的内存超出限制时，通过断开与客户端的连接来保证服务器的稳定性。

总结

今天呢，我们看了看初始化服务端的代码，是不是很懵，对的。但是我们一步一步的从外部向内剥去，最终一定会吃透他的。加油