anet

在 Redis 7.0 中，anet 文件主要用于处理底层网络通信。anet 文件中包含的函数用于实现 socket 的创建、连接、发送和接收数据等操作。

具体而言，anet 文件中包含的函数如下：

```
//anetTcpNonBlockConnect：创建一个非阻塞的TCP连接。
int anetTcpNonBlockConnect(char *err, const char *addr, int port);
//anetTcpNonBlockBestEffortBindConnect：创建一个绑定到特定源地址的非阻塞TCP连接。
int anetTcpNonBlockBestEffortBindConnect(char *err, const char *addr, int port, const char *source_addr);
//anetResolve：将主机名解析为IP地址。
int anetResolve(char *err, char *host, char *ipbuf, size_t ipbuf_len, int flags);
//anetTcpServer：创建一个TCP服务器套接字并开始监听客户端连接请求。
int anetTcpServer(char *err, int port, char *bindaddr, int backlog);
//anetTcp6Server：创建一个IPv6 TCP服务器套接字并开始监听客户端连接请求。
int anetTcp6Server(char *err, int port, char *bindaddr, int backlog);
//anetUnixServer：创建一个Unix域套接字服务器并开始监听客户端连接请求。
int anetUnixServer(char *err, char *path, mode_t perm, int backlog);
//anetTcpAccept：接受客户端连接请求并返回连接的IP地址和端口号。
int anetTcpAccept(char *err, int serversock, char *ip, size_t ip_len, int *port);
//anetUnixAccept：接受客户端连接请求。
int anetUnixAccept(char *err, int serversock);
//anetNonBlock：将套接字设置为非阻塞模式。
int anetNonBlock(char *err, int fd);
//anetBlock：将套接字设置为阻塞模式。
int anetBlock(char *err, int fd);
//anetCloexec：设置套接字的close-on-exec标志。
int anetCloexec(int fd);
//anetEnableTcpNoDelay：启用TCP的Nagle算法，提高小包传输效率。
int anetEnableTcpNoDelay(char *err, int fd);
//anetDisableTcpNoDelay：禁用TCP的Nagle算法。
int anetDisableTcpNoDelay(char *err, int fd);
//anetSendTimeout：设置发送超时时间。
```

```
int anetSendTimeout(char *err, int fd, long long ms);
//anetRecvTimeout：设置接收超时时间。
int anetRecvTimeout(char *err, int fd, long long ms);
//anetFdToString：将套接字的IP地址和端口号转换为字符串形式。
int anetFdToString(int fd, char *ip, size_t ip_len, int *port,
int fd_to_str_type);
//anetKeepAlive：启用TCP的keepalive机制。
int anetKeepAlive(char *err, int fd, int interval);
//anetFormatAddr：格式化IP地址和端口号为字符串形式。
int anetFormatAddr(char *fmt, size_t fmt_len, char *ip, int
port);
//anetFormatFdAddr：格式化套接字的IP地址和端口号为字符串形式。
int anetFormatFdAddr(int fd, char *buf, size_t buf_len, int
fd_to_str_type);
//anetPipe：创建一个管道。
int anetPipe(int fds[2], int read_flags, int write_flags);
//anetSetSockMarkId：为套接字设置mark标志。
int anetSetSockMarkId(char *err, int fd, uint32_t id);
```

1 anetSetBlock

```
int anetSetBlock(char *err, int fd, int non_block) {
    int flags;

    /* Set the socket blocking (if non_block is zero) or non-
blocking.
     * Note that fcntl(2) for F_GETFL and F_SETFL can't be
     * interrupted by a signal. */
    if ((flags = fcntl(fd, F_GETFL)) == -1) {
        anetSetError(err, "fcntl(F_GETFL): %s",
strerror(errno));
        return ANET_ERR;
    }

    if (non_block)
        flags |= O_NONBLOCK;
    else
        flags &= ~O_NONBLOCK;

    if (fcntl(fd, F_SETFL, flags) == -1) {
        anetSetError(err, "fcntl(F_SETFL,O_NONBLOCK): %s",
strerror(errno));
        return ANET_ERR;
    }
```

```
    return ANET_OK;
}
```

```
int anetNonBlock(char *err, int fd) {
    return anetSetBlock(err,fd,1);
}

int anetBlock(char *err, int fd) {
    return anetSetBlock(err,fd,0);
}

```

`anetNonBlock` 函数将一个套接字设置为非阻塞模式，即在读写数据时不会阻塞进程。它接受一个指向错误缓冲区的指针 `err` 和一个文件描述符 `fd`，如果设置成功则返回 `ANET_OK`，否则返回 `ANET_ERR` 并在错误缓冲区中设置错误信息。

`anetBlock` 函数将一个套接字设置为阻塞模式，即在读写数据时会阻塞进程。它接受一个指向错误缓冲区的指针 `err` 和一个文件描述符 `fd`，如果设置成功则返回 `ANET_OK`，否则返回 `ANET_ERR` 并在错误缓冲区中设置错误信息。

2 anetCloexec

`anetCloexec` 函数用于设置给定文件描述符的 `FD_CLOEXEC` 标志位，以使该描述符在调用 `exec()` 系统调用时自动关闭。

```
int anetCloexec(int fd) {
    int r;
    int flags;

    do {
        r = fcntl(fd, F_GETFD);
    } while (r == -1 && errno == EINTR);

    if (r == -1 || (r & FD_CLOEXEC))
        return r;

    flags = r | FD_CLOEXEC;

    do {
        r = fcntl(fd, F_SETFD, flags);
    } while (r == -1 && errno == EINTR);

    return r;
}
```

```
20
```

其中，`fd` 参数是需要设置 `FD_CLOEXEC` 标志位的文件描述符。

该函数的返回值为 `0` 表示设置成功，否则表示设置失败。失败时，错误信息会存储在全局变量 `errno` 中。

在 Redis 中，`anetCloexec` 函数主要在网络事件处理器的初始化中被调用，以确保在执行新的程序时不会将文件描述符泄漏到子进程中。

3 anetKeepAlive

`anetKeepAlive`函数用于在TCP套接字上启用keepalive选项。当在两个对等方之间的连接空闲一段时间后，TCP keepalive机制将发送一些数据包来检查对等方是否仍然存在。这个函数接受一个文件描述符和一个表示时间间隔的整数作为参数，以指定TCP keepalive检查之间的时间间隔。如果该函数返回-1，则表示设置TCP keepalive选项时发生错误，并且在`err`缓冲区中设置了一个相应的错误消息。

```
1  int anetKeepAlive(char *err, int fd, int interval)
2  {
3      int val = 1;
4
5      if (setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &val,
   sizeof(val)) == -1)
6      {
7          anetSetError(err, "setsockopt SO_KEEPALIVE: %s",
   strerror(errno));
8          return ANET_ERR;
9      }
10
11 #ifdef __linux__
12     /* Default settings are more or less garbage, with the
   keepalive time
13      * set to 7200 by default on Linux. Modify settings to
   make the feature
14      * actually useful. */
15
16     /* Send first probe after interval. */
17     val = interval;
18     if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPIDLE, &val,
   sizeof(val)) < 0) {
19         anetSetError(err, "setsockopt TCP_KEEPIDLE: %s\n",
   strerror(errno));
20         return ANET_ERR;
21     }
```

```
22
23      /* Send next probes after the specified interval. Note
    that we set the
24       * delay as interval / 3, as we send three probes before
    detecting
25       * an error (see the next setsockopt call). */
26      val = interval/3;
27      if (val == 0) val = 1;
28      if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPINTVL, &val,
    sizeof(val)) < 0) {
29          anetSetError(err, "setsockopt TCP_KEEPINTVL: %s\n",
    strerror(errno));
30          return ANET_ERR;
31      }
32
33      /* Consider the socket in error state after three we send
    three ACK
34       * probes without getting a reply. */
35      val = 3;
36      if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPCNT, &val,
    sizeof(val)) < 0) {
37          anetSetError(err, "setsockopt TCP_KEEPCNT: %s\n",
    strerror(errno));
38          return ANET_ERR;
39      }
40  #elif defined(__APPLE__)
41      /* Set idle time with interval */
42      val = interval;
43      if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPALIVE, &val,
    sizeof(val)) < 0) {
44          anetSetError(err, "setsockopt TCP_KEEPALIVE: %s\n",
    strerror(errno));
45          return ANET_ERR;
46      }
47  #else
48      ((void) interval); /* Avoid unused var warning for non
    Linux systems. */
49  #endif
50
51      return ANET_OK;
52  }
```

4 anetSetTcpNoDelay

anetSetTcpNoDelay函数是redis中anet网络库中的函数之一，用于设置TCP套接字的TCP_NODELAY选项。

TCP_NODELAY选项表示禁用Nagle算法，该算法用于减少网络流量并提高网络效率，但是在一些特殊情况下（如实时通信）可能会引起延迟问题，因此需要禁用该算法。

函数签名如下：

```
static int anetSetTcpNoDelay(char *err, int fd, int val)
{
    if (setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, &val, sizeof(val)) == -1)
    {
        anetSetError(err, "setsockopt TCP_NODELAY: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}
```

该函数接受三个参数：

- err：指向错误信息缓冲区的指针，如果出现错误则会将错误信息写入到该缓冲区。
- fd：指定需要设置TCP_NODELAY选项的套接字文件描述符。
- val：指定需要设置的选项值，1表示开启TCP_NODELAY选项，0表示关闭TCP_NODELAY选项。

函数返回值为0表示设置成功，否则返回-1表示设置失败。

在redis的源码中，anetSetTcpNoDelay函数主要用于在网络套接字创建之后，设置TCP_NODELAY选项。如果设置失败，则会将错误信息写入到err参数所指向的缓冲区。

总之，anetSetTcpNoDelay函数是用于设置TCP套接字的TCP_NODELAY选项的函数，它可以禁用Nagle算法以减少网络延迟，提高网络效率。

```
int anetEnableTcpNoDelay(char *err, int fd)
{
    return anetSetTcpNoDelay(err, fd, 1);
}

int anetDisableTcpNoDelay(char *err, int fd)
{
    return anetSetTcpNoDelay(err, fd, 0);
}
```

5 anetSendTimeout

anetSendTimeout函数用于设置socket发送数据时的超时时间。

函数原型为：

```
int anetSendTimeout(char *err, int fd, long long ms) {
    struct timeval tv;

    tv.tv_sec = ms/1000;
    tv.tv_usec = (ms%1000)*1000;
    if (setsockopt(fd, SOL_SOCKET, SO_SNDTIMEO, &tv, sizeof(tv)) == -1) {
        anetSetError(err, "setsockopt SO_SNDTIMEO: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}

```

参数说明：

- `err`：用于存储错误信息的字符串缓冲区。
- `fd`：待设置的socket文件描述符。
- `ms`：发送超时时间，单位为毫秒。

返回值为`ANET_OK`表示设置成功，否则表示设置失败。

如果socket在指定的超时时间内没有发送完所有的数据，则会返回一个错误。

这个函数主要用于保证数据发送的及时性，如果数据发送过程中遇到网络拥堵或者其他原因导致发送速度变慢，设置发送超时时间可以让程序及时返回错误信息，避免一直阻塞在发送数据的操作上。

6 anetRecvTimeout

```
/* Set the socket receive timeout (SO_RCVTIMEO socket option) to the specified
 * number of milliseconds, or disable it if the 'ms' argument is zero. */
int anetRecvTimeout(char *err, int fd, long long ms) {
    struct timeval tv;

    tv.tv_sec = ms/1000;
    tv.tv_usec = (ms%1000)*1000;
    if (setsockopt(fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) == -1) {
        anetSetError(err, "setsockopt SO_RCVTIMEO: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}
```

anetRecvTimeout函数用于设置套接字的接收超时时间。该函数的参数包括一个错误缓冲区、一个套接字描述符、以及一个表示超时时间的毫秒数。如果套接字在指定时间内没有接收到数据，函数将返回-1并将错误信息写入err缓冲区。否则，函数将返回接收到的字节数。

如果未设置接收超时时间，套接字将一直等待数据到达。但是，在某些情况下，等待太久可能会导致问题，例如网络故障或处理延迟。因此，在某些应用程序中，设置接收超时时间可以提高系统的可靠性和健壮性。

要使用anetRecvTimeout函数，需要在套接字上首先调用anetNonBlock函数以将其设置为非阻塞模式。如果套接字未设置为非阻塞模式，则接收超时设置将不起作用。

7 anetResolve

```
/* Resolve the hostname "host" and set the string representation of the
 * IP address into the buffer pointed by "ipbuf".
 *
 * If flags is set to ANET_IP_ONLY the function only resolves hostnames
 * that are actually already IPv4 or IPv6 addresses. This turns the function
 * into a validating / normalizing function. */
```

```
int anetResolve(char *err, char *host, char *ipbuf, size_t ipbuf_len,
                        int flags)
{
    struct addrinfo hints, *info;
    int rv;

    memset(&hints,0,sizeof(hints));
    if (flags & ANET_IP_ONLY) hints.ai_flags = AI_NUMERICHOST;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;  /* specify socktype to avoid dups */

    if ((rv = getaddrinfo(host, NULL, &hints, &info)) != 0) {
        anetSetError(err, "%s", gai_strerror(rv));
        return ANET_ERR;
    }
    if (info->ai_family == AF_INET) {
        struct sockaddr_in *sa = (struct sockaddr_in *)info->ai_addr;
        inet_ntop(AF_INET, &(sa->sin_addr), ipbuf, ipbuf_len);
    } else {
        struct sockaddr_in6 *sa = (struct sockaddr_in6 *)info->ai_addr;
        inet_ntop(AF_INET6, &(sa->sin6_addr), ipbuf, ipbuf_len);
    }

    freeaddrinfo(info);
    return ANET_OK;
}
```

anetResolve函数用于将一个主机名解析为一个IPv4或IPv6地址。它接收一个指向错误字符串的指针和一个主机名作为参数，并返回一个IP地址字符串。

如果解析成功，则返回0，错误字符串指针不会被修改。如果解析失败，则返回-1，错误字符串指针被设置为相应的错误消息。

该函数还支持可选的标志参数，用于控制解析方式。例如，传递ANET_IP_ONLY标志将强制使用IPv4地址，而不是IPv6地址。

总之，anetResolve函数用于将主机名解析为IP地址，并返回一个字符串表示该地址。如果解析失败，将返回一个错误码和一个描述错误的错误字符串。

example:

举个例子，假设我们有一个主机名为"example.com"，我们想要获得其IP地址。我们可以使用anetResolve函数来实现：

```
char err[ANET_ERR_LEN];
char ipbuf[INET6_ADDRSTRLEN];
if (anetResolve(err, "example.com", ipbuf, sizeof(ipbuf),
ANET_NONE) == ANET_OK) {
    printf("IP address of example.com is %s\n", ipbuf);
} else {
    printf("Failed to resolve example.com: %s\n", err);
}

```

在上面的代码中，我们传递了主机名"example.com"，并指定了标志ANET_NONE，这将使函数使用系统默认的解析方法（通常是DNS解析）。如果解析成功，将返回ANET_OK并将IP地址存储在ipbuf缓冲区中，然后我们将其打印出来。如果解析失败，则会返回ANET_ERR并将错误信息存储在err缓冲区中。

getaddrinfo

`getaddrinfo()`是一个用于将主机名或服务名转换为网络地址的函数。它可以根据传入的主机名或服务名返回一个或多个网络地址，这些地址可以用于网络通信中的客户端或服务器端。

该函数的使用方式如下：

```
int getaddrinfo(const char *node, const char *service, const
struct addrinfo *hints, struct addrinfo **res);
```

其中，参数`node`和`service`分别指定了主机名和服务名，`hints`指定了一些选项和限制条件，`res`是一个指向`addrinfo`结构体链表头的指针，用于返回解析结果。

函数的返回值为0表示成功，其他值表示失败。在成功时，函数会将解析结果存储在`res`指向的`addrinfo`结构体链表中，每个结构体中包含一个网络地址和一些与地址相关的信息，例如地址族、套接字类型、IP地址、端口号等。

下面是一个简单的使用`getaddrinfo()`函数解析主机名的例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <arpa/inet.h>

```

```c
int main(int argc, char *argv[]) {
    struct addrinfo hints, *res;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET; // 只解析IPv4地址
    hints.ai_socktype = SOCK_STREAM; // 只解析TCP协议的地址
    int status = getaddrinfo("www.baidu.com", NULL, &hints, &res);
    if (status != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
        return 1;
    }
    struct sockaddr_in *addr = (struct sockaddr_in *)res->ai_addr;
    printf("www.baidu.com's IP address is %s\n", inet_ntoa(addr->sin_addr));
    freeaddrinfo(res);
    return 0;
}

```

这个程序通过调用`getaddrinfo()`函数解析主机名"www.baidu.com"，并打印出该主机名对应的IP地址。注意，`hints`结构体中设置了`ai_family`为`AF_INET`和`ai_socktype`为`SOCK_STREAM`，这表示只解析IPv4地址和TCP协议的地址。如果要解析IPv6地址，需要将`ai_family`设置为`AF_INET6`；如果要同时解析IPv4和IPv6地址，需要将`ai_family`设置为`AF_UNSPEC`。

8 anetSetReuseAddr

```c
static int anetSetReuseAddr(char *err, int fd) {
    int yes = 1;
    /* Make sure connection-intensive things like the redis benchmark
     * will be able to close/open sockets a zillion of times */
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes)) == -1) {
        anetSetError(err, "setsockopt SO_REUSEADDR: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}

```

anetSetReuseAddr 函数用于设置 socket 的 SO_REUSEADDR 选项，即允许在同一端口上启动多个监听程序。

函数参数说明：

- `err`：出错信息。
- `fd`：socket 文件描述符。

函数返回值：

- 执行成功返回 `ANET_OK`，失败返回 `ANET_ERR`。

具体实现逻辑为：通过 `setsockopt` 函数设置 `SO_REUSEADDR` 选项。如果执行成功，返回 `ANET_OK`，否则返回 `ANET_ERR`。

9 anetCreateSocket

anetCreateSocket函数的作用是创建一个新的套接字（socket），并根据传入的参数设置其属性（如协议、阻塞/非阻塞等）。

函数原型为：

```
static int anetCreateSocket(char *err, int domain) {
    int s;
    if ((s = socket(domain, SOCK_STREAM, 0)) == -1) {
        anetSetError(err, "creating socket: %s", strerror(errno));
        return ANET_ERR;
    }

    /* Make sure connection-intensive things like the redis benchmark
     * will be able to close/open sockets a zillion of times */
    if (anetSetReuseAddr(err,s) == ANET_ERR) {
        close(s);
        return ANET_ERR;
    }
    return s;
}
```

其中，参数 `err` 是指向保存错误信息的缓冲区的指针；参数 `domain` 是套接字的协议族，如 `AF_INET` 表示 IPv4；参数 `type` 是套接字的类型，如 `SOCK_STREAM` 表示 TCP；参数 `protocol` 是使用的协议，如 0 表示使用默认协议。

函数返回值为新创建的套接字的文件描述符，如果出错则返回 -1，并将错误信息保存在 `err` 缓冲区中

10 anetTcpGenericConnect

anetTcpGenericConnect是Redis网络库中用于创建和连接TCP套接字的通用函数。该函数包含以下参数：

```
static int anetTcpGenericConnect(char *err, const char *addr, int port,
                                 const char *source_addr, int flags)
{
    int s = ANET_ERR, rv;
    char portstr[6];  /* strlen("65535") + 1; */
    struct addrinfo hints, *servinfo, *bservinfo, *p, *b;

    snprintf(portstr,sizeof(portstr),"%d",port);
    memset(&hints,0,sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((rv = getaddrinfo(addr,portstr,&hints,&servinfo)) != 0) {
        anetSetError(err, "%s", gai_strerror(rv));
        return ANET_ERR;
    }
    for (p = servinfo; p != NULL; p = p->ai_next) {
        /* Try to create the socket and to connect it.
         * If we fail in the socket() call, or on connect(), we retry with
         * the next entry in servinfo. */
        if ((s = socket(p->ai_family,p->ai_socktype,p->ai_protocol)) == -1)
            continue;
        if (anetSetReuseAddr(err,s) == ANET_ERR) goto error;
        if (flags & ANET_CONNECT_NONBLOCK && anetNonBlock(err,s) != ANET_OK)
            goto error;
        if (source_addr) {
            int bound = 0;
            /* Using getaddrinfo saves us from self-determining IPv4 vs IPv6 */
            if ((rv = getaddrinfo(source_addr, NULL, &hints, &bservinfo)) != 0)
```

```
            {
                anetSetError(err, "%s", gai_strerror(rv));
                goto error;
            }
            for (b = bservinfo; b != NULL; b = b->ai_next) {
                if (bind(s,b->ai_addr,b->ai_addrlen) != -1) {
                    bound = 1;
                    break;
                }
            }
            freeaddrinfo(bservinfo);
            if (!bound) {
                anetSetError(err, "bind: %s",
strerror(errno));
                goto error;
            }
        }
        if (connect(s,p->ai_addr,p->ai_addrlen) == -1) {
            /* If the socket is non-blocking, it is ok for
connect() to
             * return an EINPROGRESS error here. */
            if (errno == EINPROGRESS && flags &
ANET_CONNECT_NONBLOCK)
                goto end;
            close(s);
            s = ANET_ERR;
            continue;
        }

        /* If we ended an iteration of the for loop without
errors, we
         * have a connected socket. Let's return to the
caller. */
        goto end;
    }
    if (p == NULL)
        anetSetError(err, "creating socket: %s",
strerror(errno));

error:
    if (s != ANET_ERR) {
        close(s);
        s = ANET_ERR;
    }

```

```
end:
    freeaddrinfo(servinfo);

    /* Handle best effort binding: if a binding address was used, but it is
     * not possible to create a socket, try again without a binding address. */
    if (s == ANET_ERR && source_addr && (flags & ANET_CONNECT_BE_BINDING)) {
        return anetTcpGenericConnect(err,addr,port,NULL,flags);
    } else {
        return s;
    }
}

```

其中，各参数的含义如下：

- `err`：输出参数，用于返回函数执行的错误信息（如果有的话）。
- `addr`：目标服务器的 IP 地址或主机名。
- `port`：目标服务器监听的端口号。
- `source_addr`：本地网络接口的 IP 地址。
- `flags`：一些额外的标志，可用于设置套接字的选项。在 `anet.c` 中并未使用该参数。

`anetTcpGenericConnect` 函数的实现过程如下：

1. 调用 `anetCreateSocket` 函数创建一个套接字。
2. 如果指定了本地网络接口的 IP 地址，调用 `anetSetReuseAddr` 和 `anetBind` 函数将该地址与套接字绑定。
3. 将目标服务器的 IP 地址或主机名解析成 IP 地址，并将结果保存在 `server_addr` 变量中。
4. 如果指定了本地网络接口的 IP 地址，将其与套接字绑定的步骤在这里执行。
5. 调用 `connect` 函数与目标服务器建立连接。

在连接建立后，如果需要的话，可以调用 `anetNonBlock` 或 `anetBlock` 函数将套接字设置为非阻塞或阻塞模式。

```
int anetTcpNonBlockConnect(char *err, const char *addr, int
port)
{
    return
anetTcpGenericConnect(err,addr,port,NULL,ANET_CONNECT_NONBLOCK
);
}

int anetTcpNonBlockBestEffortBindConnect(char *err, const char
*addr, int port,
                                         const char
*source_addr)
{
    return anetTcpGenericConnect(err,addr,port,source_addr,
            ANET_CONNECT_NONBLOCK|ANET_CONNECT_BE_BINDING);
}

```

11 anetUnixGenericConnect

，用于与一个Unix域套接字建立连接。其函数原型如下：

```
int anetUnixGenericConnect(char *err, const char *path, int
flags)
{
    int s;
    struct sockaddr_un sa;

    if ((s = anetCreateSocket(err,AF_LOCAL)) == ANET_ERR)
        return ANET_ERR;

    sa.sun_family = AF_LOCAL;
    strncpy(sa.sun_path,path,sizeof(sa.sun_path)-1);
    if (flags & ANET_CONNECT_NONBLOCK) {
        if (anetNonBlock(err,s) != ANET_OK) {
            close(s);
            return ANET_ERR;
        }
    }
    if (connect(s,(struct sockaddr*)&sa,sizeof(sa)) == -1) {
        if (errno == EINPROGRESS &&
            flags & ANET_CONNECT_NONBLOCK)
            return s;

```

```
        anetSetError(err, "connect: %s", strerror(errno));
        close(s);
        return ANET_ERR;
    }
    return s;
}
```

其中，参数err是返回错误信息的指针；path是Unix域套接字的路径名；flags是连接的标志位，可以设置为ANET_CONNECT_NONBLOCK（非阻塞模式）或ANET_CONNECT_BE_BINDING（绑定指定地址）。

函数实现过程如下：

1. 调用socket函数创建一个Unix域套接字。如果失败，返回错误。
2. 如果设置了非阻塞模式，则将套接字设置为非阻塞模式。
3. 如果设置了绑定标志，则调用bind函数将套接字绑定到指定地址上。如果失败，返回错误。
4. 调用connect函数连接到Unix域套接字。如果设置了非阻塞模式，则判断连接是否已经建立；否则，一直阻塞直到连接建立或失败。
5. 如果连接建立失败，则关闭套接字并返回错误。
6. 如果连接建立成功，则返回套接字文件描述符。

总之，anetUnixGenericConnect函数用于连接到Unix域套接字，并且支持非阻塞模式和绑定指定地址。

12 anetListen

anetListen函数用于在指定地址和端口上创建监听套接字。它的原型如下：

```
static int anetListen(char *err, int s, struct sockaddr *sa, socklen_t len, int backlog) {
    if (bind(s,sa,len) == -1) {
        anetSetError(err, "bind: %s", strerror(errno));
        close(s);
        return ANET_ERR;
    }

    if (listen(s, backlog) == -1) {
        anetSetError(err, "listen: %s", strerror(errno));
        close(s);
        return ANET_ERR;
    }
    return ANET_OK;
}

```

参数说明：

- err：错误信息输出缓冲区。
- sockfd：已经创建好的套接字描述符。
- sa：指向sockaddr结构体的指针，表示待监听的地址信息。
- len：sockaddr结构体的长度。
- backlog：监听队列的长度。

anetListen函数将sockfd套接字绑定到指定地址和端口，然后将套接字设置为监听状态。当客户端连接该地址和端口时，内核会将客户端的连接请求排入队列中，等待被服务器进程接受。backlog参数指定队列的长度，表示可以同时等待接受的客户端连接数量。

如果成功，则返回0；否则返回-1，并将错误信息输出到err缓冲区中。

13 anetV6Only

`anetV6Only` 是一个函数，用于设置 IPv6 socket 的 `IPV6_V6ONLY` 选项。IPv6 socket 通常支持 IPv4 和 IPv6 两种协议。启用 `IPV6_V6ONLY` 选项后，IPv6 socket 将只接受 IPv6 连接，忽略 IPv4 连接。如果未启用该选项，则 IPv6 socket 将接受来自 IPv4 或 IPv6 的连接。

函数定义如下：

```
static int anetV6Only(char *err, int s) {
    int yes = 1;
    if (setsockopt(s,IPPROTO_IPV6,IPV6_V6ONLY,&yes,sizeof(yes)) == -1) {
        anetSetError(err, "setsockopt: %s", strerror(errno));
        return ANET_ERR;
    }
    return ANET_OK;
}
```

其中，参数 `fd` 表示要设置的 IPv6 socket 的文件描述符，参数 `err` 为错误信息的缓冲区。如果函数执行成功，返回 0；否则返回 -1，并将错误信息保存在 `err` 缓冲区中。

```
int fd = socket(AF_INET6, SOCK_STREAM, 0);
if (fd == -1) {
    perror("socket");
    exit(1);
}

int yes = 1;
if (setsockopt(fd, IPPROTO_IPV6, IPV6_V6ONLY, &yes,
sizeof(yes)) == -1) {
    perror("setsockopt");
    exit(1);
}

```

以上代码使用 `socket` 函数创建了一个 IPv6 socket，然后使用 `setsockopt` 函数设置了 `IPV6_V6ONLY` 选项。`setsockopt` 函数的第一个参数 `fd` 是要设置的 socket 的文件描述符；第二个参数 `IPPROTO_IPV6` 表示要设置的选项是 IPv6 相关的选项；第三个参数 `IPV6_V6ONLY` 表示要设置的选项是 `IPV6_V6ONLY`；第四个参数 `&yes` 是一个指向要设置的选项值的指针，这里将其设置为 1，表示启用 `IPV6_V6ONLY` 选项；第五个参数 `sizeof(yes)` 表示要设置的选项值的大小，这里是 4 字节，因为 `yes` 是一个 int 类型的变量。如果 `setsockopt` 函数执行成功，`IPV6_V6ONLY` 选项将被启用。

14 _anetTcpServer

```
static int _anetTcpServer(char *err, int port, char *bindaddr,
int af, int backlog)
{
    int s = -1, rv;
    char _port[6];  /* strlen("65535") */
    struct addrinfo hints, *servinfo, *p;

    snprintf(_port,6,"%d",port);
    memset(&hints,0,sizeof(hints));
    hints.ai_family = af;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;    /* No effect if bindaddr
!= NULL */
    if (bindaddr && !strcmp("*", bindaddr))
        bindaddr = NULL;
    if (af == AF_INET6 && bindaddr && !strcmp("::*",
bindaddr))
        bindaddr = NULL;

```

```
16
17      if ((rv = getaddrinfo(bindaddr,_port,&hints,&servinfo)) !=
    0) {
18          anetSetError(err, "%s", gai_strerror(rv));
19          return ANET_ERR;
20      }
21      for (p = servinfo; p != NULL; p = p->ai_next) {
22          if ((s = socket(p->ai_family,p->ai_socktype,p-
    >ai_protocol)) == -1)
23              continue;
24
25          if (af == AF_INET6 && anetV6Only(err,s) == ANET_ERR)
    goto error;
26          if (anetSetReuseAddr(err,s) == ANET_ERR) goto error;
27          if (anetListen(err,s,p->ai_addr,p->ai_addrlen,backlog)
    == ANET_ERR) s = ANET_ERR;
28          goto end;
29      }
30      if (p == NULL) {
31          anetSetError(err, "unable to bind socket, errno: %d",
    errno);
32          goto error;
33      }
34
35  error:
36      if (s != -1) close(s);
37      s = ANET_ERR;
38  end:
39      freeaddrinfo(servinfo);
40      return s;
41  }
```

函数参数：

- `err`：保存错误信息的缓冲区，大小至少为 ANET_ERR_LEN。
- `port`：端口号。
- `bindaddr`：需要监听的 IP 地址，传入 NULL 时监听所有本机 IP 地址。
- `backlog`：连接队列的最大长度。
- `tcp_fastopen`：是否启用 TCP Fast Open。为 0 时不启用，非 0 值时启用。

函数返回值：

- 若出现错误，则返回 ANET_ERR。
- 若成功创建 socket，则返回 socket 文件描述符

函数大致执行流程如下：

1. 将传入的端口号转换成字符串。
2. 调用 getaddrinfo 函数解析 bindaddr 和端口号，获取用于创建 socket 的地址信息（addrinfo 结构体）

```
int anetTcpServer(char *err, int port, char *bindaddr, int backlog)
{
    return _anetTcpServer(err, port, bindaddr, AF_INET, backlog);
}

int anetTcp6Server(char *err, int port, char *bindaddr, int backlog)
{
    return _anetTcpServer(err, port, bindaddr, AF_INET6, backlog);
}

```

15 anetUnixServer

`anetUnixServer` 函数用于在 Unix 域上创建一个服务器 socket，并绑定到指定的地址。

函数定义如下

```
int anetUnixServer(char *err, char *path, mode_t perm, int backlog)
{
    int s;
    struct sockaddr_un sa;

    if (strlen(path) > sizeof(sa.sun_path)-1) {
        anetSetError(err,"unix socket path too long (%zu), must be under %zu", strlen(path), sizeof(sa.sun_path));
        return ANET_ERR;
    }
    if ((s = anetCreateSocket(err,AF_LOCAL)) == ANET_ERR)
        return ANET_ERR;

    memset(&sa,0,sizeof(sa));
    sa.sun_family = AF_LOCAL;
    strncpy(sa.sun_path,path,sizeof(sa.sun_path)-1);
```

```
    if (anetListen(err,s,(struct sockaddr*)&sa,sizeof(sa),backlog) == ANET_ERR)
        return ANET_ERR;
    if (perm)
        chmod(sa.sun_path, perm);
    return s;
}

```

函数参数说明如下：

- `path`：一个字符串指针，表示服务器 socket 绑定的 Unix 域 socket 路径，必须是以 null 结尾的字符串。
- `perm`：一个整数，表示服务器 socket 文件的权限。
- `backlog`：一个整数，表示在监听队列中排队的最大连接数。

函数返回值为服务器 socket 的文件描述符，如果函数执行失败，则返回 `-1`。

代码首先通过 `socket` 函数创建一个 AF_UNIX 套接字，并将其文件描述符保存到变量 `s` 中。接下来，使用 `memset` 函数将 `sa` 结构体清零，并设置 `sa.sun_family` 为 AF_UNIX。然后，使用 `strncpy` 函数将 `path` 复制到 `sa.sun_path` 中，同时确保字符串以 null 结尾。最后，使用 `bind` 函数将服务器 socket 绑定到指定地址。如果绑定失败，则关闭 socket，返回 `ANET_ERR`。如果绑定成功，则使用 `listen` 函数将 socket 转换为监听 socket。如果转换失败，则关闭 socket，返回 `ANET_ERR`。如果转换成功，则使用 `chmod` 函数更改服务器 socket 文件的权限，并返回 socket 文件描述符 `s`。

16 anetGenericAccept

`anetGenericAccept` 函数用于接受客户端的连接请求，它的声明如下：

```
static int anetGenericAccept(char *err, int s, struct sockaddr *sa, socklen_t *len) {
    int fd;
    do {
        /* Use the accept4() call on linux to simultaneously accept and
         * set a socket as non-blocking. */
#ifdef HAVE_ACCEPT4
        fd = accept4(s, sa, len,  SOCK_NONBLOCK | SOCK_CLOEXEC);
#else
        fd = accept(s,sa,len);
#endif
    } while(fd == -1 && errno == EINTR);
```

```
    if (fd == -1) {
        anetSetError(err, "accept: %s", strerror(errno));
        return ANET_ERR;
    }
#ifndef HAVE_ACCEPT4
    if (anetCloexec(fd) == -1) {
        anetSetError(err, "anetCloexec: %s", strerror(errno));
        close(fd);
        return ANET_ERR;
    }
    if (anetNonBlock(err, fd) != ANET_OK) {
        close(fd);
        return ANET_ERR;
    }
#endif
    return fd;
}
```

其中，`err` 为错误信息缓存，`sockfd` 为已经监听的套接字描述符，`sa` 为接受方套接字地址缓存，`len` 为接受方套接字地址缓存长度指针。

该函数主要步骤如下：

1. 调用 `accept()` 函数接受客户端的连接请求，返回连接的套接字描述符。
2. 如果成功接受了连接请求，将客户端的套接字地址信息写入 `sa` 缓存中，`len` 指针更新为 `sa` 缓存的实际长度。
3. 如果 `accept()` 函数调用失败，将错误信息写入 `err` 缓存中，并返回 `-1`。

函数的返回值为成功接受的客户端套接字描述符，如果失败返回 `-1`。

17 anetTcpAccept

anetTcpAccept函数是对anetGenericAccept函数的封装，用于接受一个客户端的连接请求，并返回与客户端连接的套接字描述符。

函数原型如下：

```
int anetTcpAccept(char *err, int serversock, char *ip, size_t
ip_len, int *port) {
    int fd;
    struct sockaddr_storage sa;
    socklen_t salen = sizeof(sa);
    if ((fd = anetGenericAccept(err,serversock,(struct
sockaddr*)&sa,&salen)) == ANET_ERR)
```

```
        return ANET_ERR;

    if (sa.ss_family == AF_INET) {
        struct sockaddr_in *s = (struct sockaddr_in *)&sa;
        if (ip) inet_ntop(AF_INET,(void*)&(s->sin_addr),ip,ip_len);
        if (port) *port = ntohs(s->sin_port);
    } else {
        struct sockaddr_in6 *s = (struct sockaddr_in6 *)&sa;
        if (ip) inet_ntop(AF_INET6,(void*)&(s->sin6_addr),ip,ip_len);
        if (port) *port = ntohs(s->sin6_port);
    }
    return fd;
}

```

函数参数说明：

- `err`：错误信息输出缓冲区。
- `serversock`：服务端套接字描述符。
- `ip`：输出参数，用于存储客户端的IP地址。
- `ip_len`：`ip`缓冲区长度。
- `port`：输出参数，用于存储客户端的端口号。

函数返回值：

- 成功：返回与客户端连接的套接字描述符。
- 失败：返回`ANET_ERR`，并将错误信息写入`err`缓冲区。

函数的实现过程如下：

- 使用`accept`函数接受客户端的连接请求，得到与客户端连接的套接字描述符。
- 使用`getnameinfo`函数获取客户端的IP地址和端口号，并将其保存到`ip`和`port`参数中。
- 返回与客户端连接的套接字描述符。

如果`accept`函数调用失败，则将错误信息写入`err`缓冲区，并返回`ANET_ERR`。

需要注意的是，`anetTcpAccept`函数会阻塞等待客户端的连接请求，直到有客户端连接上来才会返回。如果需要非阻塞地等待客户端的连接请求，可以使用`anetTcpAcceptNonBlock`函数。

18 anetUnixAccept

`anetUnixAccept` 函数用于接受来自 Unix 域套接字的连接请求。

函数签名如下：

```
int anetUnixAccept(char *err, int s) {
    int fd;
    struct sockaddr_un sa;
    socklen_t salen = sizeof(sa);
    if ((fd = anetGenericAccept(err,s,(struct sockaddr*)&sa,&salen)) == ANET_ERR)
        return ANET_ERR;

    return fd;
}
```

参数说明：

- `err`：出错信息存储指针。
- `serversock`：监听 Unix 域套接字的文件描述符。
- `path`：Unix 域套接字绑定的路径。
- `accept_flags`：`accept` 函数的 flags 参数。

函数返回值为新建立连接的文件描述符，出错返回 `-1`。

该函数内部先调用 `accept` 函数接收连接请求，如果出错，则将错误信息写入 `err` 指向的地址，并返回 `-1`。如果成功接收连接，则返回新建立连接的文件描述符。

19 anetFdToString

`anetFdToString` 函数用于将给定的文件描述符转换为可读的字符串形式。它的函数原型如下：

```
int anetFdToString(int fd, char *ip, size_t ip_len, int *port, int fd_to_str_type) {
    struct sockaddr_storage sa;
    socklen_t salen = sizeof(sa);

    if (fd_to_str_type == FD_TO_PEER_NAME) {
        if (getpeername(fd, (struct sockaddr *)&sa, &salen) == -1) goto error;
    } else {
```

```
        if (getsockname(fd, (struct sockaddr *)&sa, &salen) ==
-1) goto error;
    }

    if (sa.ss_family == AF_INET) {
        struct sockaddr_in *s = (struct sockaddr_in *)&sa;
        if (ip) {
            if (inet_ntop(AF_INET,(void*)&(s-
>sin_addr),ip,ip_len) == NULL)
                goto error;
        }
        if (port) *port = ntohs(s->sin_port);
    } else if (sa.ss_family == AF_INET6) {
        struct sockaddr_in6 *s = (struct sockaddr_in6 *)&sa;
        if (ip) {
            if (inet_ntop(AF_INET6,(void*)&(s-
>sin6_addr),ip,ip_len) == NULL)
                goto error;
        }
        if (port) *port = ntohs(s->sin6_port);
    } else if (sa.ss_family == AF_UNIX) {
        if (ip) {
            int res = snprintf(ip, ip_len, "/unixsocket");
            if (res < 0 || (unsigned int) res >= ip_len) goto
error;
        }
        if (port) *port = 0;
    } else {
        goto error;
    }
    return 0;

error:
    if (ip) {
        if (ip_len >= 2) {
            ip[0] = '?';
            ip[1] = '\0';
        } else if (ip_len == 1) {
            ip[0] = '\0';
        }
    }
    if (port) *port = 0;
    return -1;
}

```

函数参数说明：

- `fd`：要转换的文件描述符。
- `buf`：存储可读字符串形式的缓冲区。
- `bufsize`：缓冲区的大小。

函数返回值为存储在缓冲区中的可读字符串形式的文件描述符。如果出现错误，则返回`NULL`。

该函数可用于打印和日志记录等场景中。例如，您可以使用该函数记录连接和断开连接的客户端信息。

20 anetFormatAddr

`anetFormatAddr` 是一个函数，用于将给定的套接字地址格式化为字符串形式。它有以下原型：

```
int anetFormatAddr(char *buf, size_t buf_len, char *ip, int port) {
    return snprintf(buf,buf_len, strchr(ip,':') ?
           "[%s]:%d" : "%s:%d", ip, port);
}

/* Like anetFormatAddr() but extract ip and port from the socket's peer/sockname. */
int anetFormatFdAddr(int fd, char *buf, size_t buf_len, int fd_to_str_type) {
    char ip[INET6_ADDRSTRLEN];
    int port;

    anetFdToString(fd,ip,sizeof(ip),&port,fd_to_str_type);
    return anetFormatAddr(buf, buf_len, ip, port);
}

```

其中：

- `buf`：存储格式化结果的缓冲区。
- `buf_len`：缓冲区的大小。
- `ip`：套接字地址的 IP 地址部分。
- `port`：套接字地址的端口部分。

这个函数可以将 IP 地址和端口号组合成类似 "127.0.0.1:6379" 的格式，方便打印和记录日志。

21 anetPipe

`anetPipe` 是 Redis 的网络库中实现的一个函数，用于创建一个无名管道，并返回其两端的文件描述符。该函数的声明如下

```
int anetPipe(int fds[2], int read_flags, int write_flags) {
    int pipe_flags = 0;
#if defined(__linux__) || defined(__FreeBSD__)
    /* When possible, try to leverage pipe2() to apply flags that are common to both ends.
     * There is no harm to set O_CLOEXEC to prevent fd leaks. */
    pipe_flags = O_CLOEXEC | (read_flags & write_flags);
    if (pipe2(fds, pipe_flags)) {
        /* Fail on real failures, and fallback to simple pipe if pipe2 is unsupported. */
        if (errno != ENOSYS && errno != EINVAL)
            return -1;
        pipe_flags = 0;
    } else {
        /* If the flags on both ends are identical, no need to do anything else. */
        if ((O_CLOEXEC | read_flags) == (O_CLOEXEC | write_flags))
            return 0;
        /* Clear the flags which have already been set using pipe2. */
        read_flags &= ~pipe_flags;
        write_flags &= ~pipe_flags;
    }
#endif

    /* When we reach here with pipe_flags of 0, it means pipe2 failed (or was not attempted),
     * so we try to use pipe. Otherwise, we skip and proceed to set specific flags below. */
    if (pipe_flags == 0 && pipe(fds))
        return -1;

    /* File descriptor flags.
     * Currently, only one such flag is defined: FD_CLOEXEC, the close-on-exec flag. */
    if (read_flags & O_CLOEXEC)
        if (fcntl(fds[0], F_SETFD, FD_CLOEXEC))
            goto error;
```

```
    if (write_flags & O_CLOEXEC)
        if (fcntl(fds[1], F_SETFD, FD_CLOEXEC))
            goto error;

    /* File status flags after clearing the file descriptor flag O_CLOEXEC. */
    read_flags &= ~O_CLOEXEC;
    if (read_flags)
        if (fcntl(fds[0], F_SETFL, read_flags))
            goto error;
    write_flags &= ~O_CLOEXEC;
    if (write_flags)
        if (fcntl(fds[1], F_SETFL, write_flags))
            goto error;

    return 0;

error:
    close(fds[0]);
    close(fds[1]);
    return -1;
}
```

参数 `fds` 是一个长度为 2 的整数数组，用于存放无名管道两端的文件描述符。

无名管道是一个特殊的管道，与命名管道不同，无名管道没有文件名与之对应，因此只能用于相关进程间的通信，一旦关闭管道的文件描述符，其所传递的数据也将被删除。在 Redis 的网络库中，anetPipe 函数主要用于实现单进程中不同线程之间的通信，例如将 I/O 线程中读取到的数据通过管道传递给其他线程进行处理。

22 anetSetSockMarkId

`anetSetSockMarkId`函数是用于设置套接字的SOCK_MARK值的函数。在Linux系统中，可以使用iptables和iproute2等工具进行流量控制和路由控制。这些工具使用Linux内核的Netfilter和Netlink功能，可以根据不同的流量标记（mark）来对流量进行分类和控制。而SOCK_MARK值则是在应用层上设置的，用于标记特定的套接字，以便在之后的流量控制和路由控制中进行分类和处理。

函数原型如下：

```
int anetSetSockMarkId(char *err, int fd, uint32_t id) {
#ifdef HAVE_SOCKOPTMARKID
    if (setsockopt(fd, SOL_SOCKET, SOCKOPTMARKID, (void *)&id, sizeof(id)) == -1) {
        anetSetError(err, "setsockopt: %s", strerror(errno));
```

```
        return ANET_ERR;
    }
    return ANET_OK;
#else
    UNUSED(fd);
    UNUSED(id);
    anetSetError(err,"anetSetSockMarkid unsupported on this
platform");
    return ANET_OK;
#endif
}
```

其中，`fd`为需要设置的套接字文件描述符，`mark`为需要设置的SOCK_MARK值。该函数会将SOCK_MARK值设置到套接字的SO_MARK选项中，并返回设置结果。若设置成功，则返回0；否则返回-1，并设置errno变量来指示错误类型。

需要注意的是，`anetSetSockMarkId`函数仅在Linux系统上可用，且需要root权限才能使用。