一 从一条命令说起之配置文件

当我们谈到 Redis 服务端启动程序时，可能有很多人会想到 `redis-server` 这个命令。作为 Redis 的核心组件，`redis-server` 不仅仅是 Redis 服务端启动的命令，还包括了 Redis 的许多配置选项和功能。在这篇博客中，我们将从这个简单的命令入手，一步步了解 Redis 服务端启动的过程和相关的配置选项。无论你是 Redis 初学者还是有一定经验的开发者，相信这篇博客都会对你有所启发和帮助。

1 redis-server

`redis-server` 是 Redis 的服务端启动程序。它通过监听端口，接收客户端发送的请求，并执行相应的 Redis 命令。在默认情况下，`redis-server` 启动后会监听 127.0.0.1:6379 这个地址和端口，这意味着只有本地能够连接到 Redis 服务器。

`redis-server` 命令可以带有多个参数和选项，用于配置 Redis 服务器的不同功能和行为。例如，你可以通过 `-p` 选项来指定 Redis 监听的端口，通过 `-a` 选项来设置 Redis 访问密码等等。这些选项和参数将在后续的博客中逐一介绍。

在启动 `redis-server` 命令之前，你需要确保已经安装了 Redis，并且 Redis 的配置文件已经正确配置。如果你还没有安装 Redis，可以参考官方文档进行安装和配置。

1.1 redis配置文件（简单介绍）

Redis 配置文件是一个文本文件，包含了 Redis 服务器的配置选项和参数。在启动 Redis 服务器时，可以通过命令行参数指定配置文件的路径，例如：

```
┌──(root㉿kali)-[/opt/local/myconfig]
└─# redis-server redis.conf
 
┌──(root㉿kali)-[/opt/local/myconfig]
└─# ps aux |grep redis-server
root        8632  0.5  0.1  55480 10268 ?        Ssl  14:59   0:00 redis-server 127.0.0.1:6379
```

通过ps aux命令我们可以很清楚的看到redis服务器已经启动且其监听在本机的6379的端口号上。

下面我会简单的介绍一下redis配置文件中的一点点信息，以便于我们能快速的配置出不同端口的服务器，后续的博客中我们将会详细的聊一聊这个话题，如过对这阔比较感兴趣的话，记得点赞点赞哦，废话不多说，下面我们来看看redis的配置文件：

bind：服务器启动的ip地址；

prot：服务器所监听的端口号；

requirepass：认证信息

Redis 配置文件支持 `include` 模块，可以在一个配置文件中引用另一个配置文件。这个功能非常有用，可以将一些通用的配置选项和参数抽象出来，单独存放在一个文件中，然后在需要的配置文件中通过 `include` 引用即可。

使用 `include` 模块非常简单，只需要在配置文件中使用 `include` 指令即可。例如，下面是一个示例：

```
# Redis 配置文件示例

# 引用通用配置文件
include redis.cong

# 监听地址和端口
bind 127.0.0.1
port 6380

# 认证密码
requirepass mypassword

# 日志文件路径
logfile /var/log/redis.log
```

在这个示例中，`include` 指令引用了另一个配置文件 `redis.conf`，这个文件包含了一些通用的配置选项和参数。在当前配置文件中，只需要指定一些特定的配置选项即可。

需要注意的是，`include` 指令可以出现在任何位置，而且可以多次引用不同的配置文件。在解析配置文件时，Redis 会按照 `include` 指令的顺序逐一引用配置文件，以此来构建最终的配置选项和参数列表。

我们启动看看：

```
└─# redis-server myredis.conf
```

```
┌──(root㉿kali)-[/opt/local/myconfig]
└─# ps aux |grep redis
root        8632  0.4  0.1  55480 10232 ?        Ssl  14:59   0:03 redis-server 127.0.0.1:6379
root        8757  0.5  0.1  55480 10220 ?        Ssl  15:12   0:00 redis-server 127.0.0.1:6380
root        8766  0.0  0.0   9436  2196 pts/0    S+   15:12   0:00 grep --color=auto redis
```

我们可以明显的看到redis服务器如我们预期所料的一样，在6380号端口启动了。

2 执行了redis-server后，redis会做什么

本文章采用的redis版本为7.0版本

2.1 配置文件处理

服务器启动时会调用 loadServerConfig() 函数，该函数定义在 src/config.c 文件中，用于加载配置文件并解析参数。

通过查找我们可以在src/server.c中发现下述代码

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

在 Redis 服务器启动时，会读取命令行参数，其中包括配置文件的路径。`server.configfile` 就是配置文件的路径，它是在 Redis 服务器启动时设置的。

在加载配置文件时，Redis 服务器调用了 `loadServerConfig` 函数，这个函数会读取配置文件中的选项和参数，并将它们保存在 Redis 服务器的内存中。`config_from_stdin` 和 `options` 参数表示是否从标准输入读取配置文件和其他选项，这些选项可以用于在 Redis 服务器启动时进行调试和测试。

在加载配置文件的过程中，Redis 会检查配置文件中的语法错误和非法选项，并在出现错误时输出错误信息。如果配置文件中存在错误，Redis 服务器将无法正常启动。

2.3 来看看loadServerConfig

`loadServerConfig` 是 Redis 服务器加载配置文件的函数。这个函数的定义位于 `src/config.c` 文件中。

`loadServerConfig` 函数的参数包括配置文件的路径、是否从标准输入读取配置文件和其他选项等。函数的主要功能是读取配置文件中的选项和参数，并将它们保存在 Redis 服务器的内存中，以供 Redis 服务器后续的操作使用。

具体来说，`loadServerConfig` 函数会调用 `loadServerConfigFromString` 函数读取配置文件内容，并将配置文件内容转换为一个字典对象，其中每个键值对表示一个配置选项和参数。然后，函数会逐一遍历字典对象中的键值对，将其转换为相应的 Redis 服务器配置选项和参数，并保存在 Redis 服务器的内存中。

在读取配置文件时，`loadServerConfig` 函数会检查配置文件的语法错误和非法选项，并在出现错误时输出错误信息。如果配置文件中存在错误，Redis 服务器将无法正常启动。

总之，`loadServerConfig` 函数是 Redis 服务器加载配置文件的核心函数，它确保了 Redis 服务器能够正确地读取和解析配置文件，并将配置选项和参数保存在内存中，以供 Redis 服务器后续的操作使用。

该函数的原型如下：

```
void loadServerConfig(char *filename, char config_from_stdin,
char *options)
```

该函数的三个参数的作用分别如下：

1. `filename`：配置文件的路径，如果 `config_from_stdin` 参数为假（即不从标准输入读取配置文件），则 Redis 服务器将从该路径读取配置文件。
2. `config_from_stdin`：标志变量，如果该参数为真，Redis 服务器将从标准输入读取配置文件。
3. `options`：其他选项，目前只有一个选项，即 `-test-conf`，用于测试配置文件的语法正确性。

下面是几个 `loadServerConfig` 函数的例子：

1. 从文件加载配置文件

   如果要从文件中加载配置文件，可以将 `config_from_stdin` 参数设为假，然后指定配置文件的路径。例如：

   ```
   loadServerConfig("/path/to/redis.conf", 0, NULL);
   ```

2. 从标准输入加载配置文件

   如果要从标准输入中加载配置文件，可以将 `config_from_stdin` 参数设为真，并将 `filename` 参数设为 `NULL`。例如：

   ```
   loadServerConfig(NULL, 1, NULL);
   ```

3. 测试配置文件的语法正确性

   如果要测试配置文件的语法正确性，可以将 `options` 参数设为 `-test-conf`。例如：

   ```
   loadServerConfig("/path/to/redis.conf", 0, "-test-
   conf");
   ```

总之，`loadServerConfig`函数的参数和作用比较简单，主要用于指定配置文件的路径和读取方式，并提供了一个用于测试配置文件语法正确性的选项。

我么在细看细看该函数：

```
void loadServerConfig(char *filename, char config_from_stdin,
char *options)
{
  sds config = sdsempty();
  char buf[CONFIG_READ_LEN+1];
  FILE *fp;
  glob_t globbuf;
  if(filename)
  {
      //....
  }
   if(config_from_stdin)
   {
       //...
   }
    if(option)
    {
        //...
    }
    loadServerConfigFromString(config);
    sdsfree(config);
}
```

三个if循环代表了我们传入的三个选项，该函数会自动进行选择要进行的操作，并将解析的信息放入到config里面，供 loadServerConfigFromString使用。

接下来我们会具体分析分析这三个条件判断分别干了什么事

2.4 filename

废话不多说，先看代码：

```
  if (filename) {
        if (strchr(filename, '*') || strchr(filename, '?') ||
strchr(filename, '[')) {
            /* A wildcard character detected in filename, so
let us use glob */
            if (glob(filename, 0, NULL, &globbuf) == 0) {

```

```
                for (size_t i = 0; i < globbuf.gl_pathc; i++) {
                    if ((fp = fopen(globbuf.gl_pathv[i], "r")) == NULL) {
                        serverLog(LL_WARNING,
                                  "Fatal error, can't open config file '%s': %s",
                                  globbuf.gl_pathv[i], strerror(errno));
                        exit(1);
                    }
                    while(fgets(buf,CONFIG_READ_LEN+1,fp) != NULL)
                        config = sdscat(config,buf);
                    fclose(fp);
                }

                globfree(&globbuf);
            }
        } else {
            /* No wildcard in filename means we can use the original logic to read and
             * potentially fail traditionally */
            if ((fp = fopen(filename, "r")) == NULL) {
                serverLog(LL_WARNING,
                          "Fatal error, can't open config file '%s': %s",
                          filename, strerror(errno));
                exit(1);
            }
            while(fgets(buf,CONFIG_READ_LEN+1,fp) != NULL)
                config = sdscat(config,buf);
            fclose(fp);
        }
    }
```

我么要想分析这段代码之前，必须了解了解以下结构，不要着急哈~~

1 glob_t

`glob_t` 是一个 POSIX 标准结构体类型，用于存储 `glob()` 函数匹配到的文件路径列表。该结构体定义在 `<glob.h>` 头文件中，具体定义如下：

```
  typedef struct {
                size_t   gl_pathc;    /* Count of paths matched so far  */
                char   **gl_pathv;    /* List of matched pathnames.  */
                size_t   gl_offs;     /* Slots to reserve in gl_pathv.  */
            } glob_t;

```

其中，各成员的含义如下：

- `gl_pathc`：匹配到的文件路径数量。
- `gl_pathv`：匹配到的文件路径列表，是一个指向 `char*` 类型指针的指针。
- `gl_offs`：`gl_pathv` 列表中保留的额外空间数量。

在 Redis 服务器中，`glob_t` 结构体类型被用于存储 `glob()` 函数匹配到的配置文件路径列表。具体地，在 Redis 服务器启动时，`loadServerConfig`函数解析配置文件，而在解析配置文件时，如果遇到 `include` 选项，则会调用 `glob()` 函数匹配指定目录下的所有配置文件，将匹配到的文件路径添加到配置文件路径列表中，以供后续的解析使用。在 `glob()` 函数返回时，返回结果会存储在 `glob_t` 结构体类型的变量中，即 `globbuf` 变量。

2 strchr

```
 if (strchr(filename, '*') || strchr(filename, '?') || strchr(filename, '['))
 {
     //...
 }
```

这行代码的作用是判断传入的文件名是否包含通配符，如果包含则返回真，否则返回假。

具体地，`strchr()` 函数是 C 标准库中的一个字符串处理函数，用于查找字符串中是否包含指定的字符，并返回该字符在字符串中第一次出现的位置。如果字符串中不包含该字符，则返回空指针。

在这行代码中，`strchr()` 函数被用来判断 `filename` 字符串中是否包含 `*`、`?` 或 `[` 这三个通配符字符，如果存在，则说明 `filename` 可能是一个包含通配符的文件名，需要使用 `glob()` 函数进行文件名扩展（globbing）。

例如，如果 `filename` 参数为 `"/etc/redis/*.conf"`，则该代码会返回真，因为其中包含了 `*` 通配符，表示匹配 `/etc/redis/` 目录下的所有以 `.conf` 结尾的文件。

3 glob

```
1  if (glob(filename, 0, NULL, &globbuf) == 0)
2  {
3      //...
4  }
```

这行代码的作用是使用 `glob()` 函数对传入的文件名进行扩展，将符合通配符匹配规则的文件名全部存储在 `globbuf` 结构体中的 `gl_pathv` 字段中，**并返回扩展后的文件名个数**（存储在 `gl_pathc` 字段中）。

具体地，`glob()` 函数是 C 标准库中的一个文件名扩展函数，用于根据指定的匹配规则对文件名进行扩展。该函数的第一个参数是匹配规则字符串，其中可以包含 `*`、`?`、`[]` 等通配符，用于匹配符合规则的文件名。第二个参数指定额外的标志，第三个参数指定一个错误处理函数，用于处理文件名扩展出错时的情况。最后一个参数是用于存储扩展后的文件名的 `glob_t` 结构体指针。

在这行代码中，`glob()` 函数被用来对传入的 `filename` 文件名进行扩展，如果扩展成功，则将扩展后的文件名存储在 `globbuf` 结构体中的 `gl_pathv` 字段中，同时返回扩展后的文件名个数（存储在 `gl_pathc` 字段中）。如果扩展失败，则会返回非零值，表示出现了错误。

例如，如果 `filename` 参数为 `"/etc/redis/*.conf"`，则该代码会调用 `glob()` 函数对 `/etc/redis/` 目录下所有以 `.conf` 结尾的文件进行扩展，并将扩展后的文件名存储在 `globbuf` 结构体中的 `gl_pathv` 字段中。如果扩展成功，那么 `glob()` 函数将返回零值，表示成功扩展了若干个文件名。

4 合并核心代码

```
1  for (size_t i = 0; i < globbuf.gl_pathc; i++) {
2                      if ((fp = fopen(globbuf.gl_pathv[i], "r"))
   == NULL) {
3                          serverLog(LL_WARNING,
4                                    "Fatal error, can't open
   config file '%s': %s",
5                                    globbuf.gl_pathv[i],
   strerror(errno));
6                          exit(1);
7                      }
```

```
                    while(fgets(buf,CONFIG_READ_LEN+1,fp) !=
NULL)
                        config = sdscat(config,buf);
                    fclose(fp);
                }

                globfree(&globbuf);
            }
```

这段代码的作用是将通过文件名扩展获得的多个配置文件读取并合并成一个完整的配置字符串。

首先，循环遍历 `globbuf.gl_pathv` 中存储的每一个文件名，**打开文件并逐行读取，将每一行的内容追加到 `config` 字符串的末尾。如果打开文件失败，则会输出错误信息，并直接退出程序。**

然后，通过调用 `globfree()` 函数释放 `globbuf` 结构体中动态分配的内存空间。

具体地，该函数的执行过程可以分为以下几个步骤：

1.遍历 `globbuf.gl_pathv` 中的每一个文件名：

```
for (size_t i = 0; i < globbuf.gl_pathc; i++) {
    ...
}
```

2.对于每一个文件名，尝试以只读方式打开文件：

```
if ((fp = fopen(globbuf.gl_pathv[i], "r")) == NULL) {
    ...
}
```

3.逐行读取文件内容，并将每一行的内容追加到 `config` 字符串的末尾：

```
while(fgets(buf,CONFIG_READ_LEN+1,fp) != NULL)
    config = sdscat(config,buf);
```

4.关闭已打开的文件：

```
fclose(fp);
```

5.循环结束后，释放 `globbuf` 结构体中动态分配的内存空间：

```
globfree(&globbuf);
```

这段代码的作用是将多个配置文件读取并合并成一个完整的配置字符串。

例如，如果传入的文件名为 `/etc/redis/redis.conf`，那么该函数会尝试打开该文件并读取其中的内容，如果该文件中包含 `include /etc/redis/*.conf` 这一行配置，则该函数会通过文件名扩展函数 `glob()` 扩展 `/etc/redis/` 目录下所有以 `.conf` 结尾的文件，并将这些文件中的内容逐行读取并合并到一个字符串中。最终，该函数会返回包含所有配置信息的字符串。

5 不需要合并文件代码

```
else {
            /* No wildcard in filename means we can use the original logic to read and
             * potentially fail traditionally */
            if ((fp = fopen(filename, "r")) == NULL) {
                serverLog(LL_WARNING,
                          "Fatal error, can't open config file '%s': %s",
                          filename, strerror(errno));
                exit(1);
            }
            while(fgets(buf,CONFIG_READ_LEN+1,fp) != NULL)
                config = sdscat(config,buf);
            fclose(fp);
        }
```

假如你看懂了合并的核心代码，那么你就会发现这两种几乎一样！

2.5 config_from_stdin

```
 if (config_from_stdin) {
       serverLog(LL_WARNING,"Reading config from stdin");
       fp = stdin;
       while(fgets(buf,CONFIG_READ_LEN+1,fp) != NULL)
           config = sdscat(config,buf);
   }
```

这段代码的作用是从标准输入读取配置信息，并将读取到的内容追加到 `config` 字符串的末尾。

首先，判断 `config_from_stdin` 参数是否为真（即是否需要从标准输入读取配置信息），如果为真，则输出一条日志信息提示正在从标准输入中读取配置信息。然后，将标准输入设置为输入流 `fp`，逐行读取标准输入中的内容，并将每一行的内容追加到 `config` 字符串的末尾。

具体地，该函数的执行过程可以分为以下几个步骤：

1. 判断是否需要从标准输入读取配置信息：

```
if (config_from_stdin) {
    ...
}
```

2.输出日志信息提示正在从标准输入中读取配置信息：

```
serverLog(LL_WARNING,"Reading config from stdin");
```

3.将标准输入设置为输入流 `fp`，逐行读取标准输入中的内容，并将每一行的内容追加到 `config` 字符串的末尾：

```
fp = stdin;
while(fgets(buf,CONFIG_READ_LEN+1,fp) != NULL)
    config = sdscat(config,buf);
```

该代码块的作用是：将标准输入设置为输入流 `fp`，并使用 `fgets()` 函数从标准输入中逐行读取内容，每次读取一行内容，存储到字符数组 `buf` 中，并将 `buf` 中的内容追加到 `config` 字符串的末尾。该循环会一直运行，直到读取完标准输入中的所有内容。

这段代码的作用是：如果 `config_from_stdin` 参数为真，则从标准输入读取配置信息，并将读取到的内容追加到 `config` 字符串的末尾。这样，即使没有通过命令行参数或配置文件传递配置信息，用户仍然可以通过标准输入传递配置信息。

2.6 options

```
  if (options) {
        config = sdscat(config,"\n");
        config = sdscat(config,options);
    }
```

这段代码的作用是将 `options` 参数指定的配置信息追加到 `config` 字符串的末尾。

首先，判断 `options` 参数是否非空，如果非空，则在 `config` 字符串的末尾插入一个换行符，然后将 `options` 参数指定的配置信息追加到 `config` 字符串的末尾。

具体地，该函数的执行过程可以分为以下几个步骤：

1. 判断是否需要将 `options` 参数指定的配置信息追加到 `config` 字符串的末尾：

```
if (options) {
    ...
}
```

2.在 `config` 字符串的末尾插入一个换行符：

```
config = sdscat(config,"\n");
```

3.将 `options` 参数指定的配置信息追加到 `config` 字符串的末尾：

```
config = sdscat(config,options);
```

该代码块的作用是：如果 `options` 参数非空，则在 `config` 字符串的末尾插入一个换行符，并将 `options` 参数指定的配置信息追加到 `config` 字符串的末尾。

这样，用户既可以通过命令行参数指定配置信息，也可以通过配置文件或标准输入指定配置信息。

2.6 loadServerConfigFromString

`loadServerConfigFromString` 函数的作用是将给定的字符串解析成配置项，并设置到 Redis 服务器的配置中。

具体地，`loadServerConfigFromString` 函数接受一个参数 `config`，表示要解析的字符串。函数的实现比较简单，它会将字符串解析成一个包含多个配置项的数组，然后逐个配置项地设置到 Redis 服务器的配置中。

以下是 `loadServerConfigFromString` 函数的伪代码实现：

```
loadServerConfigFromString(config):
    # 将字符串按行拆分成多个配置项
    lines = split(config, "\n")

    # 逐个配置项地解析并设置到 Redis 服务器的配置中
    for line in lines:
        # 解析配置项，得到配置项名称和值
        name, value = parseConfigLine(line)

        # 如果解析成功，则将配置项设置到 Redis 服务器的配置中
        if name and value:
            setConfigOption(name, value)
```

其中，`parseConfigLine` 函数用于解析单个配置项，将配置项名称和值解析出来。`setConfigOption` 函数用于将解析出来的配置项设置到 Redis 服务器的配置中。

总之，`loadServerConfigFromString` 函数是 Redis 服务器启动时加载配置信息的重要组成部分，它允许用户通过字符串直接指定配置信息，比如从配置文件中读取配置信息并解析成字符串，然后传给 `loadServerConfigFromString` 函数，即可将配置信息设置到 Redis 服务器中。

完整代码如下，安装惯例，我还是会分部分讲解代码：

```
void loadServerConfigFromString(char *config) {
    deprecatedConfig deprecated_configs[] = {
        {"list-max-ziplist-entries", 2, 2},
        {"list-max-ziplist-value", 2, 2},
        {"lua-replicate-commands", 2, 2},
        {NULL, 0},
    };
    char buf[1024];
    const char *err = NULL;
    int linenum = 0, totlines, i;
    sds *lines;


    reading_config_file = 1;
    lines = sdssplitlen(config,strlen(config),"\n",1,&totlines);

    for (i = 0; i < totlines; i++) {
        //***
    }

    if (server.logfile[0] != '\0') {
       //***
    }

    /* Sanity checks. */
    if (server.cluster_enabled && server.masterhost) {
       //***
    }

    /* To ensure backward compatibility and work while hz is out of range */
    if (server.config_hz < CONFIG_MIN_HZ) server.config_hz = CONFIG_MIN_HZ;
```

```
    if (server.config_hz > CONFIG_MAX_HZ) server.config_hz = CONFIG_MAX_HZ;

    sdsfreesplitres(lines,totlines);
    reading_config_file = 0;
    return;

loaderr:
    fprintf(stderr, "\n*** FATAL CONFIG FILE ERROR (Redis %s) ***\n",
        REDIS_VERSION);
    if (i < totlines) {
        fprintf(stderr, "Reading the configuration file, at line %d\n", linenum);
        fprintf(stderr, ">>> '%s'\n", lines[i]);
    }
    fprintf(stderr, "%s\n", err);
    exit(1);
}

```

1 解析config

```
sds *sdssplitlen(const char *s, ssize_t len, const char *sep, int seplen, int *count)
lines = sdssplitlen(config,strlen(config),"\n",1,&totlines);
```

在 Redis 中，`sdssplitlen` 函数用于将字符串按照指定分隔符分割成多个字符串，并将结果保存到一个字符串数组中。该函数的参数及作用如下：

- `s`：要被分割的字符串。
- `len`：要被分割的字符串的长度。
- `sep`：分隔符，是一个 C 字符串。
- `seplen`：分隔符的长度。
- `count`：返回被分割成的子字符串数量。
- `eachlen`：返回一个数组，其中包含每个子字符串的长度。

在 `loadServerConfigFromString` 函数中，代码 `sdssplitlen(config,strlen(config),"\n",1,&totlines)` 将配置字符串 `config` 按照换行符 `"\n"` 进行分割，并将结果保存到 `lines` 数组中，同时记录被分割成的子字符串数量并保存到 `totlines` 变量中。此时 `lines` 数组的每个元素即为配置文件中的一行。

需要注意的是，`sdssplitlen` 函数返回的 `lines` 数组是动态分配的，因此在使用完毕后需要手动释放内存，以避免内存泄漏。在 Redis 中，可以通过调用 `sdsfreesplitres(lines, totlines)` 函数来释放 `lines` 数组。

2 for循环

```
for (i = 0; i < totlines; i++) {
        sds *argv;
        int argc;

        linenum = i+1;
        lines[i] = sdstrim(lines[i]," \t\r\n");

        /* Skip comments and blank lines */
        if (lines[i][0] == '#' || lines[i][0] == '\0') continue;

        /* Split into arguments */
        argv = sdssplitargs(lines[i],&argc);
        if (argv == NULL) {
            err = "Unbalanced quotes in configuration line";
            goto loaderr;
        }

        /* Skip this line if the resulting command vector is empty. */
        if (argc == 0) {
            sdsfreesplitres(argv,argc);
            continue;
        }
        sdstolower(argv[0]);

        /* Iterate the configs that are standard */
        standardConfig *config = lookupConfig(argv[0]);
        if (config) {
            /* For normal single arg configs enforce we have a single argument.
             * Note that MULTI_ARG_CONFIGs need to validate arg count on their own */
            if (!(config->flags & MULTI_ARG_CONFIG) && argc != 2) {
                err = "wrong number of arguments";
                goto loaderr;
            }

```

```
35              if ((config->flags & MULTI_ARG_CONFIG) && argc ==
2 && sdslen(argv[1])) {
36                  /* For MULTI_ARG_CONFIGs, if we only have one
argument, try to split it by spaces.
37                   * Only if the argument is not empty,
otherwise something like --save "" will fail.
38                   * So that we can support something like --
config "arg1 arg2 arg3". */
39                  sds *new_argv;
40                  int new_argc;
41                  new_argv = sdssplitargs(argv[1], &new_argc);
42                  if (!config->interface.set(config, new_argv,
new_argc, &err)) {
43                      goto loaderr;
44                  }
45                  sdsfreesplitres(new_argv, new_argc);
46              } else {
47                  /* Set config using all arguments that
follows */
48                  if (!config->interface.set(config, &argv[1],
argc-1, &err)) {
49                      goto loaderr;
50                  }
51              }
52
53              sdsfreesplitres(argv,argc);
54              continue;
55          } else {
56              int match = 0;
57              for (deprecatedConfig *config =
deprecated_configs; config->name != NULL; config++) {
58                  if (!strcasecmp(argv[0], config->name) &&
59                      config->argc_min <= argc &&
60                      argc <= config->argc_max)
61                  {
62                      match = 1;
63                      break;
64                  }
65              }
66              if (match) {
67                  sdsfreesplitres(argv,argc);
68                  continue;
69              }
70          }
71
```

```
72          /* Execute config directives */
73          if (!strcasecmp(argv[0],"include") && argc == 2) {
74              loadServerConfig(argv[1], 0, NULL);
75          } else if (!strcasecmp(argv[0],"rename-command") &&
argc == 3) {
76              struct redisCommand *cmd =
lookupCommandBySds(argv[1]);
77              int retval;
78
79              if (!cmd) {
80                  err = "No such command in rename-command";
81                  goto loaderr;
82              }
83
84              /* If the target command name is the empty string
we just
85               * remove it from the command table. */
86              retval = dictDelete(server.commands, argv[1]);
87              serverAssert(retval == DICT_OK);
88
89              /* Otherwise we re-add the command under a
different name. */
90              if (sdslen(argv[2]) != 0) {
91                  sds copy = sdsdup(argv[2]);
92
93                  retval = dictAdd(server.commands, copy, cmd);
94                  if (retval != DICT_OK) {
95                      sdsfree(copy);
96                      err = "Target command name already
exists"; goto loaderr;
97                  }
98              }
99          } else if (!strcasecmp(argv[0],"user") && argc >= 2)
{
100             int argc_err;
101             if (ACLAppendUserForLoading(argv,argc,&argc_err)
== C_ERR) {
102                 const char *errmsg = ACLSetUserStringError();
103                 snprintf(buf,sizeof(buf),"Error in user
declaration '%s': %s",
104                     argv[argc_err],errmsg);
105                 err = buf;
106                 goto loaderr;
107             }
```

```
        } else if (!strcasecmp(argv[0],"loadmodule") && argc >= 2) {
            queueLoadModule(argv[1],&argv[2],argc-2);
        } else if (strchr(argv[0], '.')) {
            if (argc < 2) {
                err = "Module config specified without value";
                goto loaderr;
            }
            sds name = sdsdup(argv[0]);
            sds val = sdsdup(argv[1]);
            for (int i = 2; i < argc; i++)
                val = sdscatfmt(val, " %S", argv[i]);
            if (!dictReplace(server.module_configs_queue, name, val)) sdsfree(name);
        } else if (!strcasecmp(argv[0],"sentinel")) {
            /* argc == 1 is handled by main() as we need to enter the sentinel
             * mode ASAP. */
            if (argc != 1) {
                if (!server.sentinel_mode) {
                    err = "sentinel directive while not in sentinel mode";
                    goto loaderr;
                }
                queueSentinelConfig(argv+1,argc-1,linenum,lines[i]);
            }
        } else {
            err = "Bad directive or wrong number of arguments"; goto loaderr;
        }
        sdsfreesplitres(argv,argc);
    }

```

3 代码说明1：

```
if (lines[i][0] == '#' || lines[i][0] == '\0') continue;
```

这段代码的作用是跳过以#号开头的行以及空行。

具体来说，它检查lines数组中第i行的第一个字符是否为#号或者是空字符\0（表示这一行没有内容）。如果是，则直接跳过这一行，继续执行下一行代码；如果不是，则继续执行后面的代码。这段代码通常用于读取文本文件，并且忽略掉注释行和空行。

4 代码说明2

```
 argv = sdssplitargs(lines[i],&argc);
        if (argv == NULL) {
            err = "Unbalanced quotes in configuration line";
            goto loaderr;
        }

        /* Skip this line if the resulting command vector is
empty. */
        if (argc == 0) {
            sdsfreesplitres(argv,argc);
            continue;
        }
        sdstolower(argv[0]);

```

这些代码的作用是把`lines[i]`这一行字符串按照空格符分割成若干个字符串，并将这些字符串存储在`argv`数组中。同时，将分割后的字符串的数量存储在`argc`变量中。假如参数为空或者解析失败，就跳过这一行，然后将第一个参数转换为小写字母。

具体来说，`sdssplitargs`是一个函数，它的第一个参数是要分割的字符串，第二个参数是一个指向整数的指针，用于存储分割后的字符串数量。这个函数返回一个指向字符串数组的指针，这个数组包含了分割后的所有字符串。

5 代码说明3

```
standardConfig *config = lookupConfig(argv[0]);
```

这段代码的作用是根据配置项的名称从预定义的配置表中查找并返回对应的配置项结构体指针。

具体来说，`argv[0]`存储了当前处理的配置项名称，`lookupConfig`函数会在一个预定义的配置表中查找这个名称对应的配置项结构体，并返回结构体指针。这个配置表可能是一个哈希表、数组等数据结构，具体实现方式取决于代码的设计。返回的结构体指针可以用于后续的配置项处理，例如读取、修改、写入等操作。

```
  if (!(config->flags & MULTI_ARG_CONFIG) && argc != 2) {
                err = "wrong number of arguments";
                goto loaderr;
            }
```

这段代码用于检查当前解析到的配置项是否是**单参数配置项（single-argument config）**并且是否提供了正确数量的参数。

其中 `config->flags` 是 `standardConfig` 结构体中一个整数类型的变量，用于存储当前配置项的标志位（flag），包括是否是单参数配置项、是否需要特殊处理等等。按位与运算符 `&` 用于检查标志位中是否包含某个特定的标志。具体来说，`config->flags & MULTI_ARG_CONFIG` 检查当前配置项是否是多参数配置项（multi-argument config）。

如果当前配置项是单参数配置项，但是提供的参数数量不等于 2，那么将错误信息设置为 "wrong number of arguments" 并跳转到 `loaderr` 标签处处理错误。

```
 if ((config->flags & MULTI_ARG_CONFIG) && argc == 2 &&
sdslen(argv[1])) {

                sds *new_argv;
                int new_argc;
                new_argv = sdssplitargs(argv[1], &new_argc);
                if (!config->interface.set(config, new_argv,
new_argc, &err)) {
                    goto loaderr;
                }
                sdsfreesplitres(new_argv, new_argc);
             }
```

如果一个配置项是MULTI_ARG_CONFIG类型的，并且传入的参数个数为2，且第二个参数不为空，则尝试将第二个参数按空格拆分成多个参数。如果成功拆分，则使用新的参数调用接口的set方法来设置这个配置项。最后释放新的参数数组的内存空间。

```
  else {
                /* Set config using all arguments that follows
*/
                if (!config->interface.set(config, &argv[1],
argc-1, &err)) {
                    goto loaderr;
                }
            }
```

如果一个配置项不需要多个参数，那么如果参数数量不等于2就会出错。如果配置项需要多个参数，并且只有一个参数，则尝试按空格拆分这个参数。如果拆分出来的参数数量不为零，则使用这些参数设置配置项。否则，使用跟随参数中的所有参数设置配置项。

如果找到了，则根据配置项的参数类型和数量对输入参数进行验证，然后根据参数数量调用对应的回调函数进行设置。如果配置项是多参数类型，则还需要进行参数分割。最后，释放`argv`指针所指向的内存，继续循环读取下一个命令行参数。如果没有找到对应的配置项，则忽略该命令行参数，继续循环读取下一个命令行参数。

6 代码说明4

```
if (!strcasecmp(argv[0],"include") && argc == 2) {
            loadServerConfig(argv[1], 0, NULL);
        }
```

这段代码是解析 Redis 配置文件中 `include` 配置项的逻辑。如果 `argv[0]` 的值是 `"include"`，且 `argc` 的值为 2，就调用 `loadServerConfig` 函数，读取指定文件名的配置文件并将其应用到 Redis 服务器中。这样就可以在一个配置文件中嵌套其他的配置文件，从而更加方便地管理 Redis 服务器的配置。

```
else if (!strcasecmp(argv[0],"rename-command") && argc == 3) {
            struct redisCommand *cmd =
lookupCommandBySds(argv[1]);
            int retval;

            if (!cmd) {
                err = "No such command in rename-command";
                goto loaderr;
            }

            /* If the target command name is the empty string
we just
             * remove it from the command table. */
            retval = dictDelete(server.commands, argv[1]);
            serverAssert(retval == DICT_OK);

            /* Otherwise we re-add the command under a
different name. */
            if (sdslen(argv[2]) != 0) {
                sds copy = sdsdup(argv[2]);

                retval = dictAdd(server.commands, copy, cmd);
```

```
                if (retval != DICT_OK) {
                    sdsfree(copy);
                    err = "Target command name already
exists"; goto loaderr;
                }
            }
```

这段代码是处理`rename-command`命令的逻辑，首先根据给定的命令名字（即`argv[1]`）查询出对应的`redisCommand`结构体指针，如果没有找到则报错。然后判断目标命令名字（即`argv[2]`）是否为空，如果为空则删除原有命令，否则将原有命令添加到新的命令名下（即`argv[2]`），如果新命令名字已经存在则报错。

```
if (!strcasecmp(argv[0],"user") && argc >= 2) {
            int argc_err;
            if (ACLAppendUserForLoading(argv,argc,&argc_err)
== C_ERR) {
                const char *errmsg = ACLSetUserStringError();
                snprintf(buf,sizeof(buf),"Error in user
declaration '%s': %s",
                    argv[argc_err],errmsg);
                err = buf;
                goto loaderr;
            }
        }
```

这段代码是用于解析和加载Redis配置文件中的用户声明。如果配置文件中包含"user"字段且后面跟随至少两个参数，那么将调用"ACLAppendUserForLoading"函数来处理这些参数。如果该函数返回C_ERR，则说明用户声明中有错误，此时会通过"ACLSetUserStringError"函数获取错误信息，然后将错误信息和发生错误的参数位置组合成一个字符串，最终将该字符串赋值给"err"变量并跳转到"loaderr"标签处。如果用户声明中的参数没有错误，则可以成功地加载该用户，并将其添加到Redis的用户列表中。

```
if (!strcasecmp(argv[0],"loadmodule") && argc >= 2) {
            queueLoadModule(argv[1],&argv[2],argc-2);
        }
```

这段代码是Redis用于加载模块的命令处理逻辑。当用户执行`loadmodule`命令时，Redis会将指定的模块加入加载队列，该队列会在后续的异步加载中被处理，直到所有的模块都被加载完成。其中，`argv[1]`是指定的模块路径，`&argv[2]`是传递给模块的参数列表，`argc-2`表示参数的数量。

```
 if (strchr(argv[0], '.')) {
            if (argc < 2) {
                err = "Module config specified without value";
                goto loaderr;
            }
            sds name = sdsdup(argv[0]);
            sds val = sdsdup(argv[1]);
            for (int i = 2; i < argc; i++)
                val = sdscatfmt(val, " %S", argv[i]);
            if (!dictReplace(server.module_configs_queue,
name, val)) sdsfree(name);
        }
```

这段代码是处理 Redis 模块的配置选项，它允许在 Redis 启动时为模块传递配置选项，类似于传递命令行参数一样。如果模块的名字中包含 `.`，那么就说明这个模块有配置选项需要设置。此时会检查是否提供了对应的配置值，如果没有提供，就会报错。如果提供了对应的配置值，则会将配置项的名字和值作为键值对存储到 Redis 的 `module_configs_queue` 字典中。其中，键名为配置项的名字，键值为配置项的值。

```
if (!strcasecmp(argv[0],"sentinel")) {
            /* argc == 1 is handled by main() as we need to
enter the sentinel
             * mode ASAP. */
            if (argc != 1) {
                if (!server.sentinel_mode) {
                    err = "sentinel directive while not in
sentinel mode";
                    goto loaderr;
                }
                queueSentinelConfig(argv+1,argc-
1,linenum,lines[i]);
            }
        }
```

这段代码是用于处理Redis配置文件中的`sentinel`指令。当`argv[0]`为`sentinel`时，如果`argc`为1，则表示需要立即进入Sentinel模式，否则将`argv[1]`及其之后的参数作为配置信息添加到Sentinel的配置队列中。如果当前Redis不是Sentinel模式，则无法使用该指令，会将错误信息存入`err`并跳转到`loaderr`标签处处理。

```
 else {
            err = "Bad directive or wrong number of arguments";
goto loaderr;
        }
```

这段代码是在 Redis 服务器加载配置文件时，处理非特定指令的情况。如果当前指令不属于任何特定指令，并且参数数量不正确，那么将返回一个错误信息。可以理解为这是一个默认的错误处理分支。

7 代码说明5

```
 if (server.logfile[0] != '\0') {
        FILE *logfp;

        /* Test if we are able to open the file. The server
will not
         * be able to abort just for this problem later... */
        logfp = fopen(server.logfile,"a");
        if (logfp == NULL) {
            err = sdscatprintf(sdsempty(),
                               "Can't open the log file: %s",
strerror(errno));
            goto loaderr;
        }
        fclose(logfp);
    }

```

这段代码的作用是检查是否可以成功打开 Redis 的日志文件。如果日志文件路径(server.logfile)不为空字符串，代码会尝试以追加模式("a")打开该文件。如果打开文件失败，则会设置一个错误信息(err)并跳转到错误处理流程(loaderr)。如果打开文件成功，则会立即关闭文件。

这段代码的目的是在 Redis 启动前检查日志文件是否可用，以避免在之后的运行过程中出现日志写入失败导致的问题。

```
 if (server.cluster_enabled && server.masterhost) {
        err = "replicaof directive not allowed in cluster mode";
        goto loaderr;
    }

    /* To ensure backward compatibility and work while hz is out of range */
    if (server.config_hz < CONFIG_MIN_HZ) server.config_hz = CONFIG_MIN_HZ;
    if (server.config_hz > CONFIG_MAX_HZ) server.config_hz = CONFIG_MAX_HZ;
```

这段代码中有两个功能：

1. 防止在集群模式下使用 `replicaof` 指令，因为在集群模式下 Redis 的主从复制机制是不同的。
2. 检查服务器配置中的 `config_hz` 参数是否在 Redis 支持的最小和最大范围内。如果该参数值小于 Redis 支持的最小值 `CONFIG_MIN_HZ`，则将其设置为 `CONFIG_MIN_HZ`；如果该参数值大于 Redis 支持的最大值 `CONFIG_MAX_HZ`，则将其设置为 `CONFIG_MAX_HZ`。这是为了确保 Redis 在不同的硬件环境下都能够正常运行。

8 代码说明6

```
loaderr:
    fprintf(stderr, "\n*** FATAL CONFIG FILE ERROR (Redis %s) ***\n",
        REDIS_VERSION);
    if (i < totlines) {
        fprintf(stderr, "Reading the configuration file, at line %d\n", linenum);
        fprintf(stderr, ">>> '%s'\n", lines[i]);
    }
    fprintf(stderr, "%s\n", err);
    exit(1);
}
```

这段代码是在处理 Redis 的配置文件时，如果遇到错误，会输出错误信息并且退出程序。具体实现是通过 goto 语句跳转到 `loaderr` 标签处，输出错误信息，然后调用 `exit()` 函数退出程序。这段代码是保证 Redis 在启动时能够正确读取配置文件，如果配置文件存在错误，能够及时发现并且终止程序，避免出现不可预知的问题。

3 总结

Redis的配置文件是一个文本文件，用于配置Redis服务器的各种参数和选项。它是启动Redis时必需的一部分，Redis将在启动时读取该文件并使用它来设置服务器的配置参数。

Redis的配置文件是以行为单位进行解析的，每行包含一个指令或一个指令和其对应的参数。指令以英文单词或缩写表示，参数可以是字符串、整数、布尔值等不同类型的值。

Redis的配置文件可以通过编辑redis.conf文件来进行修改。在该文件中，每个指令都有一个默认值，但可以通过手动更改配置文件中的值来覆盖这些默认值。

Redis配置文件中的指令和参数可以分为以下几类：

1. 服务器配置：如服务器端口号、IP地址、数据持久化方式、日志文件等。
2. 数据库配置：如数据库数量、数据库的最大内存使用量、超时时间等。
3. 安全配置：如密码、访问控制等。
4. 性能配置：如最大客户端连接数、缓存大小等。
5. 集群配置：如集群模式下的主从复制、节点分布等。

Redis的配置文件在服务器启动时被读取，并根据其中的指令和参数设置服务器的配置参数。可以通过修改redis.conf文件来更改Redis的配置参数。

今天的文章就到这个，如果你觉得我写的好的话，记得点赞哦，可以转载，但要附上原创信息哦，后面呢我会给大家介绍：

1. 初始化数据结构：Redis 服务器会初始化一些全局数据结构，例如服务器状态结构体、事件循环结构体等等。
2. 打开监听端口：Redis 服务器会打开指定的监听端口，并开始监听客户端连接。
3. 进入事件循环：Redis 服务器会进入事件循环，等待事件的到来，并根据不同的事件类型调用相应的事件处理器进行处理。

和配置文件一样精彩哦！！！！