## 使用说明

公共配置文件作用

有配置文件

> //master监听的端口
>
> //master_port传送数据端口
>
> //master_port3紧急端口
>
> //ｍａｓｔｅｒ的ｉｐ
>
> //ｃｌｉｅｎｔ开放端口
>
> //初始化５０－１００

master配置文件作用

> //存储文件的路径

client配置文件作用

> //脚本名字
>
> //存储路径
>
> //压缩路径
>
> //间隔运行时间

## master

宏定义：pot 3 开放3个端口

端口１用于监听

端口２用于接受报警信息

结构体说明

```
typedef struct master{
    int fd_server[pot];//门卫套接字
    int listen_port[pot];//收节点监听端口,数据传送端口  紧急端口
    socklen_t len_addr_client;
    int client_port;
    char *path;//文件存储路径
    char *prename;//开始ip
    int  start;//开始ip
    int finish;//结束ip
    struct sockaddr_in addr_server[3];
}MASTER;
```

数据结构说明

**链表数组，有头结点**

存放客户端信息，客户端套接字，脚本名字等

```
//1,链表节点
typedef struct Node{
    int fd_client;
    //用来创建master端给客户端发送信息的套接字
    struct sockaddr_in addr_client;
    //保存客户端信息
    socklen_t len_addr_client;
    //标记addr——client长度
    char filename[scripe][max_size];
    //脚本的名字
    struct Node *next;
    //指向下一个节点
}Node;
typedef struct linkedlist{
    Node head;
    //虚拟节点
    int length;
    //链表长度
    int index;
}linkedlist;

//链表初始化
void init_linkedlist(linkedlist *p, int ind) {
    p->index = ind;
    p->head.next = NULL;
    p->length = 0;
    return ;
}

//申请链表节点
Node *get_new_node() {
    Node *p = (Node *)calloc(sizeof(Node), 1);
```

```
    p->len_addr_client = sizeof(struct sockaddr_in);
    return p;
}

//在链表某个位置插入一个节点
void insert_list(linkedlist *l, Node *new_node, int ind) {
    Node *p = &(l->head);
    while (ind--) {
        p->next;
        if (p == NULL) return ;
    }
    new_node->next = p->next;
    p->next = new_node;
    l->length += 1;
    return ;
}

//删除链表节点
void delete_node(linkedlist *l, Node *node) {
    if (node == NULL) return ;
    Node *current_node = &(l->head);
    while (current_node != NULL && current_node->next != node) {
        current_node = current_node->next;
    }
    Node *delete_node = current_node->next;
    current_node->next = delete_node->next;
    free(delete_node);
    l->length -= 1;
    return ;
}

//遍历链表寻找元素最少的那个链表
int find_min(linkedlist *l) {
    int ret = 0x3f3f3f3f, ind = 0;
    for (int i = 0; i < INS; i++) {
        if (l[i].length < ret) {
            ret = l[i].length;
            ind = i;
        }
    }
    return ind;
}
void output(linkedlist *l) {
    printf ("[");
    printf("index :%d length :%d\n", l->index, l->length);
    for (Node *node = l->head.next; node != NULL; node = node->next) {
        printf("%s ", inet_ntoa(node->addr_client.sin_addr));
    }
    printf("]\n");
    return ;
}
```

**hash哈希结构**

结构功能:  控制接受ｉｐ范围，防止重复插入

```
//2hash结构
//hash结构定义
typedef struct HashTable{
    int data[hashsize];
    int flag[hashsize];
}HashTable;

//hash函数
int hashfunc(int value) {
    return value &0x7fffffff;
}

//2次探测法
int search(HashTable *h, int value) {
    int pos = hashfunc(value);
    int ind = pos % hashsize;
    int time = 1;
    while (h->flag[ind] != -1 && h->flag[ind] != value) {
        ind = (ind + time * time) % hashsize;
        time++;
    }
    if (h->flag[ind] == value && h->data[ind] == 1) {
        //插入过
        return 1;
    } else if (h->flag[ind] == value && h->data[ind] == -1){
        //没插入过
        return 0;
    }
    //不存在,无法插入
    return -1;
}

//插入hash
int insert_hash(HashTable * h, int value) {
    int pos = hashfunc(value);
    int ind = pos % hashsize;
    int time = 1;
    while (h->flag[ind] != -1 && h->flag[ind] != value) {
        ind = (ind + time * time) % hashsize;
        time++;
    }
    if (h->flag[ind] == value) {
        h->data[ind] = 1;
        return ind;
    }
    return -1;
}
//hash初始化flag
void insert_flag(HashTable *h, int value) {
    int pos = hashfunc(value);
    int ind = pos % hashsize;
```

```
    int time = 1;
    while (h->flag[ind] != -1) {
        ind = (ind + time * time) % hashsize;
        time++;
    }
    h->flag[ind] = value;
    return ;
}

//初始化hash
void init_hash(HashTable * h, MASTER * mast) {
   memset(h, -1, sizeof(HashTable));
    char buffer[max_size];
    memset(buffer, 0, sizeof(buffer));
    strcpy(buffer, mast->prename);//192.168.1.
    int len = strlen(buffer);
    for (int i = mast->start; i <= mast->finish; i++) {
        sprintf(buffer + len, "%d", i);
        insert_flag(h, inet_addr(buffer));
    }
    return ;
}
```

**初始化**

读取配置文件，获取存储路径，监听端口，ｃｌｉｅｎｔ监听端口，初始化各个数据结构

**操作**

开启子线程并行度为５，同时会接受５个客户端的信息

打开所有监听端口，

主线程等待链接，如果有链接就插入数组

**子线程操作**

遍历链表操作，

对于每个接单创建与客户端的６次链接，并接受数据，存放到响应文件夹下，每个数据的第一个字符是标志位，标识数据的种类

**报警信息**

部分解释和说明

注释

## client

读取配置文件，读取键值，存放在数组中

初始化客户端信息，ｍａｓｔｅｒ端ｉｐ地址，监听信息，（警报信息），文件存储路径，压缩路径

心跳函数，监听端口，

初始化各个脚本信息

初始化互斥锁

ｍａｉｎ函数

初始化

开启监听线程

开启检测文件的线程

开启脚本运行线程

主线程开始线条

用于监听，如果有链接就循环６次发送６个脚本的信息

并看是否有警报信息如果有警报信息就个ｍａｓｔｅｒ发送报警信息