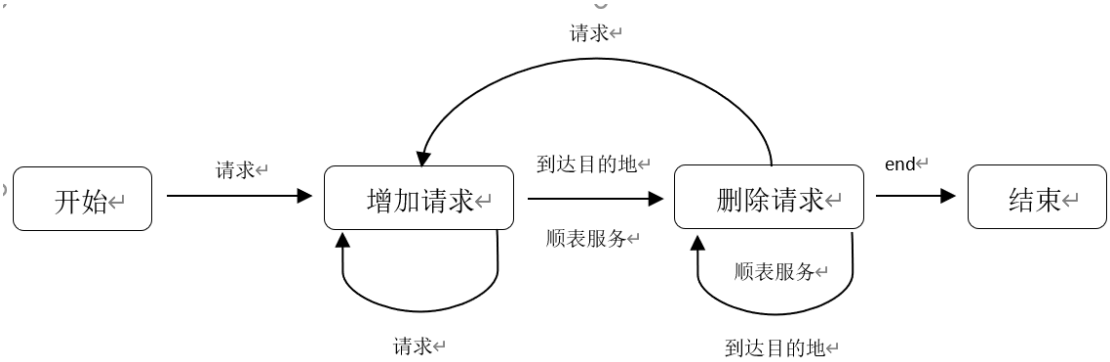
概要设计

一.自动机模型（状态转移图）



B．请求的状态转移图

本系统假设测试用例保证end是最后一条指令，后续不需要再输出。如图，在公交车静止时，接收到请求指令，则状态转换为增加请求；在增加请求状态下，接收到请求继续增加请求，在到达目的地或顺便服务时，状态转换为删除请求；在删除请求状态下，在到达目的地或顺便服务时继续删除请求，在接收到请求时，状态转换为增加请求，在接收到end指令时结束。

对于增加请求和删除请求，可以近一步细化为clockwise,counterclockwise,target三种请求的增加和删除。

对于计算机控制的公交车，它接收的指令是从键盘输入或从文件读入的。

二.高层数据结构设计

1、常量定义

（1）调度策略

```
#define FCFS 0
#define SSTF 1
#define SCAN 2
```

（2）请求类型

```
#define TARGET 0
#define CLOCKWISE 1
#define COUNTERCLOCKWISE 2
```

（3）行驶方向

```
#define WISE 1
#define COUNTER 2
```

2、数据结构定义

（1）公交车的结构

```
struct Bus
{
    int position;//公交车的位置
    int direction;//公交车的行驶方向
};
```

（2）车站的结构

```
struct Station

{
    char *clockwise;//站台顺时针请求
    char *counterclockwise;//逆时针请求
    char *target;//站台请求
};
```

（3）目的地的结构

```
struct Destination
{
    int position;//目的地的位置
    int requestType;//目的地对应的请求类型
}
```

（4）配置参数的结构

```
 struct Dict
{
    int TOTAL_STATION;
    int STATERGY;
    int DISTANCE;
};
```

（5）所有请求的链表

```
struct Request
{
    int position;
    int requestType;
    struct Request*nextPtr;
}
```

SSTF策略额外补充:

```
struct conHead
{
    struct convenient* pNext;
    int num;
}
struct convenient
{
    int position;
    struct Request*nextPtr;
}
```

```
struct reqHead
{
    struct request* pNext;
    int num;
}
struct request
{
    int position;
    int requestType;
    struct Request*nextPtr;
}
```

三、系统模块划分

（一）软件结构图

本系统程序部分划分为`main.c`、`dict.c`、`fcfs.c`、`ssfc.c`、`scan.c`、`busControl.c`、`stationControl.c`、`input.c`、`output.c`

1. 模块名称：`main.c`

   模块功能简要描述：主函数，运行各个模块与调用函数；根据调度策略，处理用户输入的指令；控制公交车的运行状态，进行相应的输出。

2. 模块名称：`dict.c`

   模块功能简要描述：去掉带#的注释；读取配置文件的三个参数 （TOTAL_STATION、STATERGY、DISTANCE）

3. 模块名称：`fcfs.c`

   模块功能简要描述：先来先服务策略

4. 模块名称：`ssfc.c`

   模块功能简要描述：最短寻找时间策略

5. 模块名称：`scan.c`

   模块功能简要描述：顺便服务策略

6. 模块名称：`input.c`

   模块功能简要描述：输入指令

7. 模块名称：`output.c`

   模块功能简要描述：按要求输出各个状态

8. 模块名称：`control.c`

   模块功能简要描述：控制并改变Bus、Station、Request中的各种参数

9. 模块名称：`check.c`

   模块功能简要描述：检查各个参数的状态，检测是否到达目的地的，检测请求链表中是否已有相同的请求，检测是否到达某个目的地，检测请求链表中存在请求

（二）文件及函数组成

1.源文件

源文件	源文件说明	函数名称	函数功能
`dict.c`	根据配置文件配置参数	`void deleteNotes(char*fileContent )`	打开配置文件，把删除#注释后的内容放入temp字符串中
		`void configurePara (char*fileContent,struct Dict*dict)`	配置三参数
		`void configureStation (struct Station*station,struct Dict* dict)`	为station中的请求开辟内存空间，并初始化
`input.c`	输入指令	`int receiveInstruction ( char*command, struct Destination* destination )`	接受指令并判断指令的类型；若是请求则储存到destination中
`output.c`	按要求输出各个状态	`void outputResults ( struct Bus* bus, struct Station*station)`	输出当前状态
`control.c`	控制并改变Bus、Station、Request中的各种参数	`void recordStation (struct Station*station, struct Destination*destination )`	增加station中的请求
		`void deleteStation ( struct Station*station, struct Destination*destination)`	删除station中的已完成请求
		`void changePosition (struct Bus* bus)`	根据行驶方向改变公交车的位置
		`void findDestination ( struct Request*request,struct Destination*destination )`	通过目的地判断汽车的行驶方向（改变Bus中的direction）
`check.c`	检查各个参数的状态	`int checkSameRquest (struct Station*Request, struct Destination*destination )`	检查是否有相同的请求
		`int arriveDestination ( struct Bus* bus, struct Destination*destination )`	检测公交车是否到达了当前的目的地
		`int checkRemain ( struct Station* station)`	检测当前是否有未完成的请求

源文件	源文件说明	函数名称	函数功能
		`void judgeDirection (struct Destination*destination, struct Bus* bus)`	通过目的地判断并改变汽车的行驶方向（改变Bus中的direction）
`fcfs.c`	先来先服务策略	`int checkNextPosition( struct Destination*destination, struct Request*request )`	判断下一个请求是否与当前到达的目的地位置相同
		`int insertListEnd(struct Destiantion value, struct Request** lastptrptr)`	把新的请求插入请求链表的尾部
`sstf.c`	最短寻找时间策略	`void moveCar(struct conHead*pCon,struct reqHead*pReq,struct Dict*dict, struct Bus*bus,struct Destination* destination)`	调用自身和里面包含的函数对车进行移动
		`void Think(struct conHead*pCon,struct reqHead*pReq,struct station*pSta, struct Bus* bus)`	判断命令是否需要添加到目标链表或顺便服务链表中
		`int GetDistance(int a,int b)`	得到两站点距离函数
		`void GetShortestDistance(struct Bus*pBus,struct conHead*pCon,struct reqHead*pReq, struct station*pSta)`	寻找最短路径
		`int isConvenient(struct Bus*pBus,struct station*pSta,struct conHead*pCon,struct reqHead*pReq)`	判断是否顺便服务
		`int isExist(struct station*pSta,struct conHead*pCon,struct reqHead*pReq)`	判断新命令是否已经存在request或convenient中;
		`int isOvercome(struct Bus*pBus)`	判断车位置是否越界;
		`int isAllFinish(struct station*pSta,struct conHead*pCon,struct reqHead*pReq)`	判断是否所有任务(request和convenient)都完成;

源文件	源文件说明	函数名称	函数功能
`scan.c`	顺便服务策略	`LISTNODE* createListHead-SCAN(void)`	创建空结点作为链表头
		`void addList-SCAN (struct Request * request, LISTNODE * headPtr)`	将新请求插入链表合适位置
		`int checkStation-SCAN(struct Request, struct Station *station)`	检查新请求对应站点是否已有其他请求
		`int judgeDirection-SCAN(LISTNODE * headPtr, struct Destination *destination, struct Bus * bus )`	判断是否改变方向
		`void switchDirection-SCAN (struct Bus * bus)`	改变方向
		`void getDestination-SCAN (stuct Destination * destination, LISTNODE * headPtr, stuct Bus * bus)`	根据当前方向确定下一目标
		`void renewStation-SCAN (struct Request *request, struct Station *station)`	更新站点请求信息

2.函数说明

序号	函数原型	功能	参数	返回值
1	`void deleteNotes ( char*fileContent )`	打开配置文件，把删除#注释后的内容放入temp字符串中	fileContent是用于存放去掉注释后的文件内容的字符串	void
2	`void configurePara (char*fileContent,struct Dict*dict)`	配置三参数	fileContent是配置文件的内容，dict是三参数的结构体指针	void
3	`void configureStation (struct Station*station,struct Dict* dict)`	为station中的请求开辟内存空间，并初始化	station是车站请求的结构体指针，dict是配置参数的结构体指针	void
4	`int receiveInstruction ( char*command, struct Destination* destination )`	接受指令并判断指令的类型；若是请求则储存到destination中	command是从键盘输入的指令，destination是destination	根据指令的类型返回相应的值0到4
5	`void outputResults ( struct Bus* bus, struct Station* station)`	输出当前状态	bus, station分别是Bus和Station的结构体	void
6	`void recordStation (struct Station*station, struct Destination*destination )`	增加station中的请求	station,destination分别是Station和Destination的结构体	void
7	`void deleteStation ( struct Station*station, struct Destination*destination)`	删除station中的已完成请求	station,destination分别是Station和Destination的结构体	void
8	`int checkSameRquest (struct Station*Request, struct Destination*destination )`	检查是否有相同的请求	request, station分别是Request和Station的结构体	有相同的请求，返回1；否则返回0
9	`int arriveDestination ( struct Bus* bus, struct Destination*destination )`	检测公交车是否到达了当前的目的地	bus, destination分别是Bus和Destination的结构体	到达了目的地返回1；未到达返回0

序号	函数原型	功能	参数	返回值
10	`int checkRemain ( struct Station* station)`	检测当前是否有未完成的请求	station是Station存请求的结构体	判断是否还有未完成的请求，有返回1；无返回0
11	`void changePosition (struct Bus* bus)`	根据行驶方向改变公交车的位置	bus是Bus的结构体	void
12	`int checkNextPosition ( struct Destination* destination, struct Request* request )`	判断下一个请求是否与当前到达的目的地位置相同（用于FCFS策略）	request,destination分别是Request和Destination的结构体	若是，返回1；若不是返回0
13	`int creatListHead(struct Request**headPtrPtr, struct Request**lastPtrPtr)`	创建表头以及空结点尾指针	headPtrPtr是链表头的二级指针；lastPtrPtr是尾节点的二级指针	创建成功返回1；创建失败返回0
14	`int insertListEnd(struct Destiantion value, struct Request** lastptrptr)`	把新的请求插入请求链表的尾部（即按先后顺序）用于FSCFS	value是等待插入链表的数据，lastPtrPtr是尾节点的二级指针	插入成功返回0；链表不存在返回-1；分配空间失败返回-2
15	`void findDestination(struct Request*request,struct Destination*destination )`	从请求链表中读取公交车当前状态目的地，放入destination中	request是请求链表，destination是当前的目的地	void
16	`void judgeDirection (struct Destination*destination, struct Bus* bus)`	通过目的地判断汽车的行驶方向（改变Bus中的direction）	destination是当前的目的地，bus是Bus的结构体	void
17	`int releaseList(LISTNODEPTR* head, LISTNODEPTR* lastptr)`	清空所有结点并释放链表	head是头节点，lastptr是尾指针	成功返回0；失败返回-1

序号	函数原型	功能	参数	返回值
18	`LISTNODE* createListHead-SCAN(void)`	创建空结点作为链表头	void	指向头结点的指针headPtr
19	`void addList-SCAN (struct Request * request, LISTNODE * headPtr)`	将新请求插入链表合适位置	request是新请求的指针，headPtr是链表头指针	void
20	`int checkStation-SCAN(struct Request, struct Station *station)`	检查新请求对应站点是否已有其他请求	request, station分别是Request和Station的结构体	有返回1，无返回0
21	`int judgeDirection-SCAN(LISTNODE * headPtr, struct Destination *destination, struct Bus * bus )`	判断是否改变方向	headPtr为指向链表的头指针，destination为当前目标的指针，bus为Bus的指针	是返回1否返回0
22	`void switchDirection-SCAN (struct Bus * bus)`	改变方向	bus为Bus的指针	void
23	`void getDestination-SCAN (stuct Destination * destination, LISTNODE * headPtr, stuct Bus * bus)`	根据当前方向确定下一目标	headPtr为指向链表的头指针，destination为当前目标的指针，bus为Bus的指针	void
24	`void renewStation-SCAN (struct Request *request, struct Station *station)`	更新站点请求信息	request为请求的指针，station为Station的指针	void
3	`void moveCar(struct conHead*pCon,struct reqHead*pReq,struct Dict*dict, struct Bus*bus,struct Destination* destination)`	调用自身和里面包含的函数对车进行移动	主目的地链表指针，顺便服务链表指针，指令结构体指针	void
4	`void outputResults ( struct Bus* bus, struct Station* station)`	在收到clock指令后输出当前状态	bus, station分别是Bus和Station的结构体指针	void
5	`void Think(struct conHead*pCon,struct reqHead*pReq,struct station*pSta, struct Bus* bus)`	判断命令是否需要添加到目标链表或顺便服务链表中	主目的地链表指针，顺便服务链表指针，指令结构体指针;bus, station分别是Bus和Station的结构体指针	void
6	`int GetDistance(int a,int b)`	得到两站点距离函数	a.b为两个坐标	int距离

序号	函数原型	功能	参数	返回值
7	`void GetShortestDistance(struct Bus*pBus,struct conHead*pCon,struct reqHead*pReq, struct station*pSta)`	寻找最短路径	主目的地链表指针，顺便服务链表指针，指令结构体指针;bus, station分别是Bus和Station的结构体指针	void
8	`int isConvenient(struct Bus*pBus,struct station*pSta,struct conHead*pCon,struct reqHead*pReq)`	判断是否顺便服务	主目的地链表指针，顺便服务链表指针，指令结构体指针;bus, station分别是Bus和Station的结构体指针	0或1
9	`int isExist(struct station*pSta,struct conHead*pCon,struct reqHead*pReq)`	判断新命令是否已经存在request或convenient中;	主目的地链表指针，顺便服务链表指针，指令结构体指针;station分别是Station的结构体指针	0或1
10	`int isOvercome(struct Bus*pBus)`	判断车位置是否越界;	bus分别是Bus的结构体指针	0或1
11	`int isAllFinish(struct station*pSta,struct conHead*pCon,struct reqHead*pReq)`	判断是否所有任务(request和convenient)都完成;	主目的地链表指针，顺便服务链表指针，指令结构体指针;bus, station分别是Bus和Station的结构体指针	0或1

2.高层算法设计

SSTF初步框架:

```
int main ()
{
    int 状态 = 静止阶段;
    while(没有接收到End指令)
    {
        switch case
        {
            case 静止阶段:
                {
                    ...;
                }
                break;
            case 调度阶段:
                {
                    ...;
                }
                break;
```

```
            case 车运动阶段:
                {
                    ...;
                }
                break;
        }
    }
    return 0;
}

```

```
case 静止阶段:
{
    收到request:
        为第一个请求申请一个request类型的链节，并接到头节点上;
        case = 调度阶段;
    收到clock:
        原地不动;
        case = 静止阶段;
}
```

```
case 调度阶段://上一个命令是request请求
{
    接收到request:
        调度函数;
        case = 调度阶段;
    接收到clock:
        车运动函数;
        case = 车运动阶段;
}
```

```
case 车运动阶段://上一个命令是clock请求
{
    接收到request:
        调度函数;
        case = 调度阶段;
    接受到clock:
        车运动函数;
        case = 车运动阶段;
}
```

```
车运动函数
{
    //运动区域
    判断车运动的方向;
    if(逆时针)
    {
        车位置--;
        判断车位置是否越界;
        if(越界)
        {
            车位置变为(车站*距离)-1;
        }
    }else{
        车位置++;
```

```
        判断车位置是否越界;
        if(越界)
        {
            车位置变1;
        }
    }
    //判断区域
    判断有无顺便服务;
    if(有顺便服务)
    {
        判断是否顺便服务;
        if(可以顺便服务)
        {
            停留一秒;
            删除convenient第一个节点;
            对convenient进行排序;
        }else{
            判断是否到达request的第一个节点
                if(是)
                {
                    停留1秒;
                    删除request的第一个节点;
                    对request进行排序,寻找下一目标站;
                }
        }

    }else{
        判断是否达到request的第一个节点;
        if(是)
        {
            停留1秒;
            删除request的第一个节点;
            对request进行排序,寻找下一目标站;
        }
    }
    判断是否所有任务(request和convenient)都完成;
    if(是)
    {
        case = 静止阶段;
    }
}
```

```
调度函数
{
    判断新命令是否已经存在request或convenient中;
    if(存在)
    {
        ;
    }else{
        判断是否可以顺便服务;
        if(可以)
        {
            创建一个链节，并添加到convenient链表中;
            对convenient链表进行排序;
        }else
        {
            创建一个链节,添加到request中;
```

```
        }
    }
}
```