myshell源代码

include

inc/config.h

```
// 程序：命令行解释器
// 作者：邱日宏 3200105842

/**
 * @file config.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 配置文件
 * @version 1.0
 * @date 2022-07-03
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _CONFIG_H_
#define _CONFIG_H_

static constexpr int BUFFER_SIZE = 1024;         // 缓冲区大小
static constexpr int MAX_PROCESS_NUMBER = 1024; // 最大进程数量
static constexpr int MAX_ARGUMENT_NUMBER = 128; // 最大参数数量

enum sh_err_t   //  shell错误类型
{
    SH_SUCCESS = 0, // 正常
    SH_FAILED,      // 失败
    SH_UNDEFINED,   // 未定义
    SH_ARGS,        // 参数错误
    SH_EXIT,        // 退出
};

enum job_state                                  // 进程状态
{
    Running,                                    // 正在运行
    Stopped,                                    // 停止运行
    Done,                                       // 完成运行
    Terminated                                  // 终止运行
};

constexpr unsigned int hash_prime = 33u;   // 相乘质数
constexpr unsigned int hash_basis = 5381u; // 偏移

/**
 * @brief 字符串散列，用于将字符串转为正整数，在编译时进行
 *
 * @param input 需要转换的字符串
 * @return unsigned constexpr 散列后的哈希值
```

```
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-19
 * @copyright Copyright (c) 2022
 */
unsigned int constexpr String_Hash(char const *input, unsigned int prime =
hash_prime, unsigned int basis = hash_basis)
{
    return *input ?                                                    //
是否达到字符串的结尾
        static_cast<unsigned int>(*input) + prime * String_Hash(input + 1) : //
还未达到，递归求和继续
        basis;                                                         //
到达末尾，返回一个质数哈希
}

#endif
```

inc/BinaryHeap.h

```
/**
 * @file BinaryHeap.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 二叉堆
 * @version 1.0
 * @date 2022-07-20
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _BINARY_HEAP_H_
#define _BINARY_HEAP_H_

#include "Heap.h"
#include <assert.h>
#include <exception>

static constexpr size_t HeapBlockSize = 1024;   // 默认堆大小

// template <class T>
// static constexpr T INF = -0x7f7f7f7f;         // 负无穷

/**
 * @brief 二项堆，小根堆
 *
 * @tparam T
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-20
 * @copyright Copyright (c) 2022
```

```
 */
template <class T>
class BinaryHeap : public Heap<T>
{
    // 由于父类是模板类，因此子类在使用时必须使用using引入命名空间，
    // 或者用this指针实现多态，这样才能正确构造父类的模板类函数
    using Heap<T>::size_;

    public:
        BinaryHeap(size_t heap_capacity = HeapBlockSize)
        : Heap<T>(), capacity_(heap_capacity)
        {
            assert(heap_capacity > 0);

            node = new T[heap_capacity+1];  // 分配内存
            if (node == NULL)               // 异常处理
                throw OutOfMemory();
        }

        BinaryHeap(T data[], size_t size, size_t heap_capacity = HeapBlockSize)
        : Heap<T>(), capacity_(heap_capacity)
        {
            node = new T[(size>heap_capacity?size:heap_capacity) + 1];  // 分配内
存
            if (node == NULL)               // 异常处理
                throw OutOfMemory();

            size_ = size;
            for (size_t i = 1; i <= size; ++i) // 数组拷贝
                node[i] = data[i-1];

            build_heap();                          // 建堆
        }

        virtual ~BinaryHeap()
        {
            delete [] node;
        }

        virtual void build(T data[], size_t size)
        {
            while (capacity_ < size)    // 内存不够则分配空间
                AllocMoreSpace();
            size_ = size;
            for (size_t i = 0; i < size; ++i)   // 拷贝数组
                node[i+1] = data[i];

            for (size_t i = (size_>>1); i>0; --i)    //从 n/2 开始
            {
                size_t p, child;
                T X = node[i];
                for (p = i; (p<<1) <= size_; p = child) //下滤
                {
                    child=(p<<1);   //寻找最小的孩子
                    if (child != size_ && node[child+1] < node[child])
```

```
                    ++child;

                if (X > node[child])
                    node[p] = node[child];
                else
                    break;
            }
            node[p] = X;
        }
    }

    virtual void insert(T value)
    {
        if (size_ + 2 >= capacity_)
        {
            AllocMoreSpace();    // 内存不够则分配空间
        }

        int p;
        for (p = ++size_; node[p>>1] > value && p > 1; p = p>>1)    //下滤
            node[p] = node[p>>1];   //避免使用swap交换
        node[p] = value; //将节点插入在正确的位置上
    }

    virtual T top() const
    {
        if (size_ == 0)
            throw ExtractEmptyHeap();
        return node[1];
    }

    virtual T extract()
    {
        if (size_ == 0)   //如果堆是空的
            throw ExtractEmptyHeap();    //那么为异常

        T top, last;
        top = node[1];
        last = node[size_--];

        size_t p, child;
        for (p = 1; (p<<1) <= size_; p = child)  //下滤
        {
            child = (p<<1);   //寻找最小的孩子
            if (child != size_ && node[child+1] < node[child])
                ++child;

            if (last >node[child])    // 如果未到合适的位置
                node[p]=node[child];  // 将孩子提上来
            else
                break;
        }

        node[p] = last;
        return top;
```

```
        }

    protected:
        size_t capacity_;  // 最大容量
        T *node;           // 数据

        class ExtractEmptyHeap : public std::exception {};
        class OutOfMemory : public std::exception {};

        void AllocMoreSpace()   // 动态数组分配空间
        {
            capacity_<<=1;      // 容量翻倍
            T *newNode = new T[capacity_];
            if (newNode == NULL)
            {
                throw OutOfMemory();    // 堆内存不足异常
            }

            for (size_t i = 0; i < size_; ++i)
                std::swap(node[i], newNode[i]); // 直接地址交换，提高效率
            delete [] node;
            node = std::move(newNode);          // 移动拷贝，效率更佳
        }

    private:
        void build_heap()
        {
            for (size_t i = (size_>>1); i>0; --i)    //从 n/2 开始
            {
                size_t p, child;
                T X = node[i];
                for (p = i; (p<<1) <= size_; p = child) //percolate down
                {
                    child=(p<<1);   //寻找最小的孩子
                    if (child != size_ && node[child+1] < node[child])
                        ++child;

                    if (X > node[child])    // 如果未到合适的位置
                        node[p] = node[child];  // 将孩子提上来
                    else
                        break;
                }
                node[p] = X;  // 找到了合适的位置
            }
        }
};

#endif
```

inc/common.h

```
// 程序：命令行解释器
// 作者：邱日宏 3200105842

/**
 * @file common.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 共享函数库
 * @version 1.0
 * @date 2022-07-15
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _COMMON_H_
#define _COMMON_H_

#include <cmath>
#include <string>
#include <cassert>
#include <sstream>

// 错误判断与信息提示
#define ASSERT(expr, message) assert((expr) && (message))

/**
 * @brief 命令参数打印
 * 首行显示传入参数个数
 * 接下来一行一次显示命令行中的各个参数，以空格分开
 *
 * @param argc 参数个数
 * @param argv 参数列表
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-15
 * @copyright Copyright (c) 2022
 */
void Argument_Display(const int argc, char* const argv[]);

/**
 * @brief 二分搜索查找，查询范围为[l, r)
 *
 * @tparam T
 * @tparam Tp
 * @param lelt 查找左区间，包含
 * @param right 查找右区间，不包含
 * @param val 查找变量
 * @param array 搜索数组
 * @param cmp 比较方法
 * @return int 返回对应元素下标，若没有找到则返回-1
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-17
```

```
 * @copyright Copyright (c) 2022
 */
template<typename T>
int Binary_Search(int left, int right, T val, T array[], int cmp(T a, T b))
{
    while (left < right)
    {
        int mid = (left + right) >> 1;
        int compare_result = cmp(val, array[mid]);
        if (compare_result == 0)    // 找到了
            return mid;
        else if (compare_result > 0)    // 查找结果在右半区间
            left = mid + 1;
        else                            // 查找结果在左半区间
            right = mid;
    }

    return -1;                      // 没找到
}

/**
 * @brief 去掉字符串两端空格
 *
 * @param s 需要去除空格的字符串
 * @return std::string& 去除完空格的字符串
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-17
 * @copyright Copyright (c) 2022
 */
std::string& String_Trim(std::string &s);

/**
 * @brief 将字符串转换成任意类型变量
 *
 * @tparam Type 返回类型
 * @param str 提取的字符串
 * @return Type 转化后的类型变量
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-18
 * @copyright Copyright (c) 2022
 */
template <class Type>
Type String_to_Number(const std::string& str)
{
    std::istringstream iss(str);
    Type num;
    iss >> num;
    return num;
}

/** 比较取小 */
template <typename T>
inline T Min(const T& a, const T& b)
```

```cpp
{
    return a < b ? a : b;
}

/** 比较取大 */
template <typename T>
inline T Max(const T& a, const T& b)
{
    return a > b ? a : b;
}

/**
 * @brief 八进制转十进制
 *
 * @tparam T
 * @param octalNumber 八进制数
 * @return T 十进制数
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-19
 * @copyright Copyright (c) 2022
 */
template <typename T>
T Octal_to_Decimal(T octalNumber)
{
    T decimalNumber = 0, i = 0, remainderNumber;
    while (octalNumber != 0)
    {
        remainderNumber = octalNumber % 10; // 余数
        octalNumber /= 10;                  // 退位
        decimalNumber += remainderNumber * pow(8, i);   // 幂乘
        ++i;
    }
    return decimalNumber;
}

/**
 * @brief 十进制转八进制
 *
 * @tparam T
 * @param decimalNumber 十进制数
 * @return T 八进制数
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-19
 * @copyright Copyright (c) 2022
 */
template <typename T>
T Decimal_to_Octal(T decimalNumber)
{
    T remainderNumber, i = 1, octalNumber = 0;
    while (decimalNumber != 0)
    {
        remainderNumber = decimalNumber % 8; // 余数
        decimalNumber /= 8;                  // 退位
```

```
        octalNumber += remainderNumber * i;  // 幂乘
        i *= 10;
    }
    return octalNumber;
}

/**
 * @brief 十六进制转十进制
 *
 * @tparam T
 * @param hexadecimalNumber 十六进制数
 * @return T 十进制数
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-19
 * @copyright Copyright (c) 2022
 */
template <typename T>
T Hexadecimal_to_Decimal(T hexadecimalNumber)
{
    T decimalNumber = 0, i = 0, remainderNumber;
    while (hexadecimalNumber != 0)
    {
        remainderNumber = hexadecimalNumber % 10; // 余数
        hexadecimalNumber /= 10;                  // 退位
        decimalNumber += remainderNumber * pow(16, i);  // 幂乘
        ++i;
    }
    return decimalNumber;
}

/**
 * @brief 十进制转十六进制
 *
 * @tparam T
 * @param decimalNumber 十进制数
 * @return T 十六进制数
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-19
 * @copyright Copyright (c) 2022
 */
template <typename T>
T Decimal_to_Hexadecimal(T decimalNumber)
{
    T remainderNumber, i = 1, hexadecimalNumber = 0;
    while (decimalNumber != 0)
    {
        remainderNumber = decimalNumber % 16; // 余数
        decimalNumber /= 16;                  // 退位
        hexadecimalNumber += remainderNumber * i;  // 幂乘
        i *= 10;
    }
    return hexadecimalNumber;
}
```

```
/**
 * @brief timespec时间比较
 *
 * @param time1 时间1
 * @param time2 时间2
 * @return true 如果time1的时间晚于time2的时间
 * @return false 如果time1的时间不晚于time2的时间
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-20
 * @copyright Copyright (c) 2022
 */
inline bool test_timespec_newer(struct timespec& time1, struct timespec& time2)
{
    if (time1.tv_sec > time2.tv_sec)    // 先比较秒
        return true;
    else if (time1.tv_sec < time2.tv_sec)
        return false;
    else
        return time1.tv_nsec > time2.tv_nsec;   // 再比较纳秒
}

/**
 * @brief timespec时间比较
 *
 * @param time1 时间1
 * @param time2 时间2
 * @return true 如果time1的时间早于time2的时间
 * @return false 如果time1的时间不早于time2的时间
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-20
 * @copyright Copyright (c) 2022
 */
inline bool test_timespec_older(struct timespec& time1, struct timespec& time2)
{
    if (time1.tv_sec < time2.tv_sec)    // 先比较秒
        return true;
    else if (time1.tv_sec > time2.tv_sec)
        return false;
    else
        return time1.tv_nsec < time2.tv_nsec;   // 再比较纳秒
}

#endif
```

inc/Console.h

```
/**
 * @file Console.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 控制台
```

```
 * @version 1.0
 * @date 2022-07-03
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _CONSOLE_H_
#define _CONSOLE_H_

#include "config.h"

#include <sys/stat.h>
#include <sys/types.h>

class ProcessManager;   // 为了加快编译速度，这里不引用头文件而只是声明

/**
 * @brief 信号控制与处理
 *
 * @param signal_
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-21
 * @copyright Copyright (c) 2022
 */
void SignalHandler(int signal_);

/**
 * @brief 控制台
 * 存储必要的环境变量以及渲染用户前端所需要的数据
 *
 */
class Console
{
    private:
        // 显示模块
        char user_name[BUFFER_SIZE];                    // 用户名称
        char host_name[BUFFER_SIZE];                    // 主机名称
        char current_working_dictionary[BUFFER_SIZE];   // 当前工作目录

        char home[BUFFER_SIZE];                         // 主目录

        // 环境变量
        char shell_path_env[BUFFER_SIZE];               // shell的完整路径

        // 进程管理
        pid_t process_id;                               // 当前进程pid
        static pid_t child_process_id;                  // 子进程pid
        ProcessManager* process_manager;                // 进程管理器

        // 文件描述符
        int input_file_descriptor;                      // 输入文件描述符
        int output_file_descriptor;                     // 输出文件描述符
        int error_file_descriptor;                      // 错误文件描述符
```

```
    // 标准输入、输出与错误输出
    static int input_std_fd;                                // 标准输入备份
    static int output_std_fd;                               // 标准输出备份

    static int error_std_fd;                                // 标准错误备份

    // 重定向标志
    bool redirect_input;                                     // 输入重定向状态
    bool redirect_output;                                    // 输出重定向状态
    bool redirect_error;                                     // 错误重定向状态

    // 掩码
    mode_t umask_;                                           // 文件掩码

    int argc;                                                // 当前命令参数个数
    char argv[MAX_ARGUMENT_NUMBER][BUFFER_SIZE];             // 当前命令参数列表

public:
    Console(/* args */);

    virtual ~Console();

    /* 初始化 */
    int init();

    /* 打印进程列表 */
    void ConsoleJobList() const;

    /* 打印已完成的进程列表 */
    void ConsoleJobListDone();

    /* 添加进程 */
    unsigned int AddJob(int pid, job_state state, int argc, char *argv[]);

    // void RemoveJob();

    void ResetChildPid() { child_process_id = -1; }

    /* 设置文件描述符 */
    void SetInputFD(int _fd)  { input_file_descriptor = _fd; }
    /* 设置文件描述符 */
    void SetOutputFD(int _fd) { output_file_descriptor = _fd; }
    /* 设置文件描述符 */
    void SetErrorFD(int _fd)  { error_file_descriptor = _fd; }

    /* 获取文件描述符 */
    int GetInputFD() const  { return input_file_descriptor; }
    /* 获取文件描述符 */
    int GetOutputFD() const { return output_file_descriptor; }
    /* 获取文件描述符 */
    int GetErrorFD() const  { return error_file_descriptor; }

    /* 设置重定向状态 */
    void SetInputRedirect()  { redirect_input = true; }
```

```
        /* 设置重定向状态 */
        void SetOutputRedirect() { redirect_output = true; }
        /* 设置重定向状态 */
        void SetErrorRedirect()  { redirect_error = true; }

        /* 重置重定向状态 */
        void ResetInputRedirect()  { redirect_input = false; }
        /* 重置重定向状态 */
        void ResetOutputRedirect() { redirect_output = false; }
        /* 重置重定向状态 */
        void ResetErrorRedirect()  { redirect_error = false; }

        /* 获取重定向状态 */
        bool GetInputRedirect() const  { return redirect_input ; }
        /* 获取重定向状态 */
        bool GetOutputRedirect() const { return redirect_output; }
        /* 获取重定向状态 */
        bool GetErrorRedirect() const  { return redirect_error ; }

        /* 获取标注输入、输出、错误输出 */
        int GetSTDIN()  const { return input_std_fd;  }
        /* 获取标注输入、输出、错误输出 */
        int GetSTDOUT() const { return output_std_fd; }
        /* 获取标注输入、输出、错误输出 */
        int GetSTDERR() const { return error_std_fd;  }

        /* 设置掩码 */
        void SetMask(mode_t _mask) { umask_ = _mask; }
        /* 获取掩码 */
        mode_t GetMask() const { return umask_; }

        friend class Display;
        friend class Executor;
        friend class ProcessManager;
        friend void SignalHandler(int);
};

#endif
```

inc/Display.h

```
/**
 * @file Display.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 显示器
 * @version 1.0
 * @date 2022-07-03
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _DISPLAY_H_
```

```
#define _DISPLAY_H_

class Console;

#include <string>

class Display
{
    private:

        Console* console_;

        bool perform;   // 是否显示提示符的标志

    protected:
        std::string buffer_;

    public:
        Display(Console* console);

        virtual ~Display();

        /**
         * @brief 命令行输入控制
         * @return 正数表示正常退出，返回读入的字符数；
         * 返回0表示读到EOF，返回负数表示出现错误
         */
        int InputCommand(char *input, const int len);

        /** @brief 命令行提示符显示模块 */
        void render();

        /** @brief 继续输入提示 */
        void prompt() const;

        /** @brief 打印信息msg与显示器 */
        void message(const char * msg);

        /** @brief 将所有打印信息统一显示在终端 */
        void show() const;

        /** @brief 清空缓冲区 */
        void clear() { buffer_ = ""; }
};

#endif
```

inc/Executor.h

```
/**
 * @file Executor.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 解释器
```

```
 * @version 1.0
 * @date 2022-07-04
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _EXECUTOR_H_
#define _EXECUTOR_H_

#include "config.h"

class Console;
class Display;

static constexpr int FunctionNumber = 16;

class Executor
{
    private:

        Console *console_;  /** @see 控制台 */

        Display *display_;  /** @see 显示器 */

    protected:

        /** 选择执行函数并执行 */
        sh_err_t shell_function(const int argc, char * const argv[], char *
const env[]) const;

        /** 更改目录 */
        sh_err_t execute_cd(const int argc, char * const argv[], char * const
env[]) const;

        /** 显示当前目录 */
        sh_err_t execute_pwd(const int argc, char * const argv[], char * const
env[]) const;

        /** 显示当前日期 */
        sh_err_t execute_time(const int argc, char * const argv[], char * const
env[]) const;

        /** 清屏 */
        sh_err_t execute_clr(const int argc, char * const argv[], char * const
env[]) const;

        /* 列出目录内容 */
        sh_err_t execute_dir(const int argc, char * const argv[], char * const
env[]) const;

        /* 列出所有环境变量 */
        sh_err_t execute_set(const int argc, char * const argv[], char * const
env[]) const;
```

```
        /* 回声 */
        sh_err_t execute_echo(const int argc, char * const argv[], char * const
env[]) const;

        /* 显示帮助手册 */
        sh_err_t execute_help(const int argc, char * const argv[], char * const
env[]) const;

        /** 退出shell */
        sh_err_t execute_exit(const int argc, char * const argv[], char * const
env[]) const;

        /** 显示当前日期 */
        sh_err_t execute_date(const int argc, char * const argv[], char * const
env[]) const;

        /** 清屏 */
        sh_err_t execute_clear(const int argc, char * const argv[], char * const
env[]) const;

        /** 获取系统环境变量 */
        sh_err_t execute_env(const int argc, char * const argv[], char * const
env[]) const;

        /** 获取当前登入用户信息 */
        sh_err_t execute_who(const int argc, char * const argv[], char * const
env[]) const;

        /** 创建新目录 */
        sh_err_t execute_mkdir(const int argc, char * const argv[], char * const
env[]) const;

        /** 移除空目录 */
        sh_err_t execute_rmdir(const int argc, char * const argv[], char * const
env[]) const;

        /** 将被挂起的作业转到后台 */
        sh_err_t execute_bg(const int argc, char * const argv[], char * const
env[]) const;

        /** 将后台作业转到前台 */
        sh_err_t execute_fg(const int argc, char * const argv[], char * const
env[]) const;

        /** 显示所有作业 */
        sh_err_t execute_jobs(const int argc, char * const argv[], char * const
env[]) const;

        /** 执行命令替换当前进程 */
        sh_err_t execute_exec(const int argc, char * const argv[], char * const
env[]) const;

        /** 检测命令执行结构 */
        sh_err_t execute_test(const int argc, char * const argv[], char * const
env[]) const;
```

```
        /** 设置掩码 */
        sh_err_t execute_umask(const int argc, char * const argv[], char * const
env[]) const;

            /** myshell */
        sh_err_t execute_myshell(const int argc, char * const argv[], char *
const env[]) const;

        /** 从命令到对应函数的映射，采用红黑树的STL实现 */

        /** 定义函数指针类型 */
        typedef sh_err_t (Executor::*MemFuncPtr)(const int argc, char * const
argv[], char * const env[]) const;
        /** 创建函数指针数组 */
        MemFuncPtr FunctionArray[FunctionNumber];

        /** 文件测试 */
        static bool test_file_state(const int argc, const char * const argv[]);
        /** 文件测试 */
        static bool test_number_compare(const int argc, const char * const
argv[]);
        /** 文件测试 */
        static bool test_string_compare(const int argc, const char * const
argv[]);

    public:
        Executor(Console *model, Display *view);

        virtual ~Executor();

        /**
         * @brief 执行器命令执行函数
         *
         * @param argc 传入参数个数
         * @param argv 传入具体参数
         * @param env  环境变量
         * @return sh_err_t 返回执行情况
         * @version 1.0
         * @author 邱日宏 (3200105842@zju.edu.cn)
         * @date 2022-07-04
         * @copyright Copyright (c) 2022
         */
        sh_err_t execute(const int argc, char * const argv[], char * const
env[]) const;
};


#endif
```

inc/Heap.h

```
/**
 * @file Heap.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 堆，抽象类
 * @version 1.0
 * @date 2022-07-20
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _HEAP_H_
#define _HEAP_H_

#include <assert.h>
#include <stddef.h>

/**
 * @brief 抽象堆
 *
 * @tparam T
 * @version 1.0
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @date 2022-07-20
 * @copyright Copyright (c) 2022
 */
template <class T>
class Heap
{
    public:
        Heap() : size_(0) {};

        /**
         * @brief Destroy the Heap object
         * Heap的析构函数。由于我们的链接库是静态库，因此无法将析构函数定义成纯虚函数。
         * 如果使用动态链接库的话则能够较好的实现多态，这里暂且将其定义为空函数以便链接。
         *
         * @version 1.0
         * @author 邱日宏 (3200105842@zju.edu.cn)
         * @date 2022-07-20
         * @copyright Copyright (c) 2022
         */
        virtual ~Heap() {};

        size_t size() const { return size_; }

        virtual void build(T data[], size_t size) = 0;

        virtual void insert(T value)
        {
            assert(false && "insert not implemented.");
        }
```

```
        virtual T top() const
        {
            assert(false && "top not implemented.");
            return 0;
        }

        virtual T extract()
        {
            assert(false && "extract not implemented.");
            return 0;
        }

    protected:
        size_t size_;      // 当前容量
};


#endif
```

inc/Parser.h

```
/**
 * @file Parser.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 语法分析
 * @version 1.0
 * @date 2022-07-19
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _PARSER_H_
#define _PARSER_H_

class Console;
class Display;
class Executor;

class Parser
{
    private:
        enum {SUCCESS = 0, EXIT = 1};

        /**
         * @brief 执行shell
         *
         * @param model
         * @param view
         * @param controller
         * @param argc
         * @param argv
         * @param env
```

```
         * @return true
         * @return false
         * @version 1.0
         * @author 邱日宏 (3200105842@zju.edu.cn)
         * @date 2022-07-19
         * @copyright Copyright (c) 2022
         */
        static bool shell_execute(Console *model, Display* view, Executor*
controller, int& argc, char *argv[], char *env[]);

    public:
        Parser(/* args */) {};

        /**
         * @brief Destroy the Parser object
         *
         * @version 1.0
         * @author 邱日宏 (3200105842@zju.edu.cn)
         * @date 2022-07-19
         * @copyright Copyright (c) 2022
         */
        virtual ~Parser() = 0;  // 抽象类，纯虚函数

        static bool shell_pipe(Console *model, Display* view, Executor*
controller, int& argc, char *argv[], char *env[]);

        static int shell_parser(Console *model, Display* view, Executor*
controller, int& argc, char *argv[], char *env[]);
};

#endif
```

inc/ProcessManager.h

```
/**
 * @file ProcessManager.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 进程管理
 * @version 1.0
 * @date 2022-07-20
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _PROCESS_MANAGER_H_
#define _PROCESS_MANAGER_H_

#include "Heap.h"
#include "config.h"

#include <set>
#include <unistd.h>
```

```
// 置于config.h头文件中，加快编译速度
// enum job_state                                    // 进程状态
// {
//     Running,                                      // 正在运行
//     Stopped,                                      // 停止运行
//     Done,                                         // 完成运行
//     Terminated                                    // 终止运行
// };

class job_unit
{
    public:
        job_unit(unsigned int _id, int _pid, job_state _state, int _argc, char *
_argv[]);

        // ~job_unit();

        void PrintJob(int output_fd = STDOUT_FILENO);

        /* 为了使用集合，我们需要重载job unit的大小比较运算符 */
        bool operator== ( const job_unit& rhs ) const
        {
            return id == rhs.id;
        }

        bool operator!= ( const job_unit& rhs ) const
        {
            return !(*this == rhs);
        }

        bool operator< ( const job_unit& rhs ) const
        {
            return id < rhs.id;
        }

        bool operator> ( const job_unit& rhs ) const
        {
            return rhs < *this;
        }

        bool operator<= ( const job_unit& rhs ) const
        {
            return !(rhs < *this);
        }

        bool operator>= ( const job_unit& rhs ) const
        {
            return !(*this < rhs);
        }

    // private:
        unsigned int id;                              // 进程列表id
        pid_t pid;                                    // 进程列表pid
        job_state state;                              // 进程列表状态
```

```
        int argc;                                       // 进程列表参数
        char argv[MAX_ARGUMENT_NUMBER][BUFFER_SIZE];// 进程列表参数
};

class ProcessManager
{
    private:
        // 进程控制
        Heap<unsigned int> *job_heap;                    // 工作id分配堆
        std::set<class job_unit> jobs;                  // 进程列表，采用STL红黑树实现

    public:
        ProcessManager(/* args */);
        virtual ~ProcessManager();

        void PrintJobList(int output_fd = STDOUT_FILENO) const; // 打印进程列表

        void PrintJobListDone(int output_fd = STDOUT_FILENO); // 打印已完成的进程列
表


        /**
         * @brief 添加进程
         *
         * @param pid 进程号
         * @param state 状态
         * @param argc 参数个数
         * @param argv 参数列表
         * @return unsigned int 进程作业号，0表示添加失败
         * @version 1.0
         * @author 邱日宏 (3200105842@zju.edu.cn)
         * @date 2022-07-20
         * @copyright Copyright (c) 2022
         */
        unsigned int JobInsert(int pid, job_state state, int argc, char
*argv[]);

        /**
         * @brief 删除进程
         *
         * @param job
         * @version 1.0
         * @author 邱日宏 (3200105842@zju.edu.cn)
         * @date 2022-07-21
         * @copyright Copyright (c) 2022
         */
        void JobRemove(job_unit * job);
        void JobRemove(std::set<job_unit>::iterator& job);

        int ForeGround(unsigned int jobid);
        int BackGround(unsigned int jobid);
};

#endif
```

inc/myshell.h

```
// 程序：命令行解释器
// 作者：邱日宏 3200105842

/**
 * @file myshell.h
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief myshell头文件
 * 包含了myshell.cpp中所需要引用的所有自定义类
 *
 * @version 1.0
 * @date 2022-07-02
 *
 * @copyright Copyright (c) 2022
 *
 */

#ifndef _MYSHELL_H_
#define _MYSHELL_H_

/* 配置文件 */
#include "config.h"

/* 类的声明 */
class Console;
class Display;
class Executor;

namespace SHELL
{
    /** @brief 启动shell */
    int shell_setup(int argc, char *argv[], char *env[]);

    /** @brief 进入shell循环 */
    int shell_loop(Console* model, Display* view, Executor* controller, char
*env[]);

} // namespace SHELL


#endif
```

source

src/common.cpp

```
/**
 * @file common.cpp
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 共享函数库
 * @version 1.0
```

```
 * @date 2022-07-15
 *
 * @copyright Copyright (c) 2022
 *
 */

#include "common.h"

#include <stdio.h>

void Argument_Display(const int argc, char* const argv[])
{
    printf("argc: %d\n", argc);
    for (int i = 0; i < argc; ++i)
    {
        printf("%s ", argv[i]);
    }
    putchar('\n');
    return;
}

std::string& String_Trim(std::string &s)
{
    if (s.empty()) // 如果s为空
    {
        return s;   // 则不必处理
    }

    s.erase(0,s.find_first_not_of(" "));    // 去除字符串前的空格
    s.erase(s.find_last_not_of(" ") + 1);   // 去除字符串后的空格
    return s;
}
```

src/Console.cpp

```
/**
 * @file Console.cpp
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 控制台
 * @version 1.0
 * @date 2022-07-03
 *
 * @copyright Copyright (c) 2022
 *
 */

#include "Console.h"
#include "BinaryHeap.h"
#include "ProcessManager.h"

#include <pwd.h>
#include <wait.h>
#include <assert.h>
```

```cpp
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <iostream>
#include <exception>

int Console::input_std_fd;
int Console::output_std_fd;
int Console::error_std_fd;
pid_t Console::child_process_id = -1;
static Console* cp = nullptr;   // 为了能够在友元函数中引用控制台，在此处设置本地变量以利于信号处理

Console::Console(/* args */)
{
    [[maybe_unused]] int ret;
    ret = init();       // 初始化
    assert(ret == 0);   // 判断初始化是否成功

    process_manager = new ProcessManager();
    cp = this;

    return;
}

Console::~Console()
{
    delete process_manager;
}

void SignalHandler(int signal_)
{
    switch (signal_)
    {
        case SIGINT:    // Ctrl C 交互注意信号
            #ifdef _DEBUG_
            printf("Ctrl + C\n");
            #endif
            if (write(STDOUT_FILENO, "\n", 1) < 0)
                throw std::exception();
            // 当kill的pid < 0时  取|pid|发给对应进程组。
            // kill(-getpid(), SIGINT);

            // 子进程的CTRL C重置了，由子进程处理中断
            break;

        case SIGTSTP:    // Ctrl Z 键盘中断
            #ifdef _DEBUG_
            printf("Ctrl + Z\n");
            #endif
            if (write(STDOUT_FILENO, "\n", 1) < 0)
                throw std::exception();

            if (Console::child_process_id >= 0)
```

```
            {
                setpgid(Console::child_process_id, 0);
                kill(Console::child_process_id, SIGTSTP);

                unsigned int jobid = cp->AddJob(Console::child_process_id,
Stopped, cp->argc, (char **)cp->argv);

                // 打印当前进程
                char buffer[BUFFER_SIZE];
                snprintf(buffer, BUFFER_SIZE-1, "[%u] %d\n", jobid,
Console::child_process_id);
                if (write(cp->output_std_fd, buffer, strlen(buffer)) == -1)
                    throw std::exception();

                snprintf(buffer, BUFFER_SIZE-1, "[%u]%c\tStopped\t\t\t\t\t",
jobid, ' ');
                if (write(cp->output_std_fd, buffer, strlen(buffer)) == -1)
                    throw std::exception();

                // 参数打印
                if (cp->argc > 0)
                {
                    // 确保行末无多余的空格
                    if (write(cp->output_std_fd, cp->argv[0], strlen(cp-
>argv[0])) == -1)
                        throw std::exception();
                    for (int i = 1; i < cp->argc; ++i)
                    {
                        if (write(cp->output_std_fd, " ", 1) == -1)    // 打印空格
                            throw std::exception();

                        // 打印参数
                        if (write(cp->output_std_fd, cp->argv[i], strlen(cp-
>argv[i])) == -1)
                            throw std::exception();
                    }
                }
                if (write(cp->output_std_fd, "\n", 1) == -1)
                    throw std::exception();

                Console::child_process_id = -1;
            }
            break;

        case SIGCHLD:   // 子进程结束
            // 父进程收到子进程退出命令后，回收子进程
            // waitpid(-1, NULL, WNOHANG);
            cp->ResetChildPid();
            break;

        default:
            break;
    }

}
```

```
int Console::init()
{
    try
    {
        // 获取用户名称
        struct passwd *pw = getpwuid(getuid());
        if (pw == nullptr)
        {
            throw "get user database entry error";
        }
        memset(user_name, 0, BUFFER_SIZE);
        strncpy(user_name, pw->pw_name, BUFFER_SIZE-1);

        // 获取主机名称
        int ret;
        ret = gethostname(host_name, BUFFER_SIZE-1);
        if (ret != 0)
        {
            throw "Error when getting host name";
        }

        // 获取当前工作目录
        char *result;
        result = getcwd(current_working_dictionary, BUFFER_SIZE);
        if (result == NULL)
        {
            throw "Error when getting current working dictionary";
        }

        // 获取主目录
        memset(home, 0, BUFFER_SIZE);
        strncpy(home, getenv("HOME"), BUFFER_SIZE-1);

        // 设置shell环境变量
        strncpy(shell_path_env, current_working_dictionary, BUFFER_SIZE);
        strncat(shell_path_env, "/myshell", BUFFER_SIZE);
        setenv("shell", shell_path_env, 1);

        // 设置掩码
        umask_ = umask(022);  // 获取默认掩码
        umask(umask_); // 改回原来掩码

        // 设置默认文件描述符
        input_file_descriptor = STDIN_FILENO;
        output_file_descriptor = STDOUT_FILENO;
        error_file_descriptor = STDERR_FILENO;

        // 设置重定向状态
        redirect_input = false;
        redirect_output = false;
        redirect_error = false;

        // 备份STD IO
        input_std_fd = dup(STDIN_FILENO);
```

```
        output_std_fd = dup(STDOUT_FILENO);
        error_std_fd = dup(STDERR_FILENO);

        // 获取进程
        process_id = getpid();
        child_process_id = -1;  // 暂无子进程

        // signal(SIGINT, SignalHandler);  // Ctrl + C
        signal(SIGTSTP, SignalHandler); // Ctrl + Z
        signal(SIGCHLD, SignalHandler); // 子进程结束时发送给父进程的信号

        // 屏幕shell从后台调用tcsetpcgrp时收到的信号
        signal(SIGTTIN, SIG_IGN);   // 屏蔽SIGTTIN信号
        signal(SIGTTOU, SIG_IGN);   // 屏蔽SIGTTOU信号
    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << '\n';
        return 1;
    }

    return 0;
}

void Console::ConsoleJobList() const
{
    /* 显示工作列表，以打印与重定向处。 */
    process_manager->PrintJobList(STDOUT_FILENO);
}

void Console::ConsoleJobListDone()
{
    /* 输出应显示在屏幕上，无论如何重定向。 */
    process_manager->PrintJobListDone(output_std_fd);
}

unsigned int Console::AddJob(int pid, job_state state, int argc, char *argv[])
{
    return process_manager->JobInsert(pid, state, argc, argv);
}
```

src/Display.cpp

```
/**
 * @file Display.cpp
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 显示器
 * @version 1.0
 * @date 2022-07-03
 *
 * @copyright Copyright (c) 2022
 *
 */
```

```cpp
#include "Display.h"
#include "Console.h"

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

Display::Display(Console* console)
: console_(console), perform(true), buffer_("")
{
}

Display::~Display()
{
}

int Display::InputCommand(char *input, const int len)
{
    tcsetpgrp(STDIN_FILENO, getpid());

    // 初始化输入缓冲器与相关变量
    char ch;
    int i = 0;
    memset(input, 0, len);

    // 循环读入字符
    do
    {
        ssize_t state = read(console_->input_file_descriptor, &ch, 1);
        if (state == 0)
        {
            // 读到了EOF，结束
            if (i == 0) // 如果此时缓冲器中什么内容也没有
                return 0;   // 直接返回
            else    // 这是文本未加入换行的最后一行
            {
                input[i++] = '\n';  // 手动加入换行
                return i;           // 将最后一行命令处理完毕
            }
        }
        else if (state == -1)
        {
            throw "Read Input Error";
        }


        if (ch == '\\') // 如果读到换行输入\命令就跳过继续
        {
            ch = getchar(); // 将随后的换行符读入
            continue;
        }

        if (ch == ';')  // 将；视为换行符，便于lexer和parser处理
```

```
        {
            ch = '\n';
            perform = false;
        }

        input[i++] = ch;

        if (i == len)   // 达到最大长度了
        {
            buffer_ = "\e[1;31mERROR\e[0m input compand exceeds maximum length.
输入命令的长度超过了允许的最大长度";
            memset(input, 0, len);  // 清空缓冲区输入
            return -1;
        }
    } while (ch != '\n');

    #ifdef _DEBUG_
    printf("input: %s", input);
    #endif

    return i;
}

void Display::render()
{
    buffer_ = "";   // 每轮循环前将输出缓冲区清空

    // 如果不是从标准输入中输入或是不是将内容输出到标准输出的话，
    if (console_->input_file_descriptor != STDIN_FILENO ||
        console_->output_file_descriptor != STDOUT_FILENO)
        return; // 就不需要打印提示符

    if (!perform)
    {
        perform = true;
        return;
    }

    int sret = 0;
    const size_t len = strlen(console_->home);
    if (strlen(console_->current_working_dictionary) >= len)
    {
        size_t i = 0;
        while (i < len)
        {
            if (console_->current_working_dictionary[i] != console_->home[i])
                break;
            ++i;
        }
        if (i == len)
            sret = i;
    }

    char buffer[BUFFER_SIZE];   // 打印缓冲区
    sret = sret
```

```
        ? snprintf(buffer, BUFFER_SIZE, "\e[1;32m%s@%s\e[0m:\e[1;34m~%s\e[0m> ",
\
        console_->user_name, console_->host_name, console_-
>current_working_dictionary+sret)
        : snprintf(buffer, BUFFER_SIZE, "\e[1;32m%s@%s\e[0m:\e[1;34m%s\e[0m> ",
\
        console_->user_name, console_->host_name, console_-
>current_working_dictionary);
    if (sret == -1)
    {
        throw "Error when writing into output buffer";
    }

    ssize_t ret;
    ret = write(console_->output_file_descriptor, buffer, strlen(buffer));
    if (ret == -1)
    {
        throw "Error when writing from buffer";
    }

    return;
}

void Display::prompt() const
{
    if (write(console_->output_file_descriptor, "> ", 2) == -1)
    {
        throw std::exception();
    }
}

void Display::message(const char * msg)
{
    buffer_ += std::string(msg);
}

void Display::show() const
{
    ssize_t ret;
    ret = write(console_->output_file_descriptor, buffer_.c_str(),
buffer_.length());
    if (ret == -1)
    {
        throw "Error when showing buffer in Display";
    }
}
```

src/Executor.cpp

```
/**
 * @file Executor.cpp
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 解释器
```

```
 * @version 1.0
 * @date 2022-07-04
 *
 * @copyright Copyright (c) 2022
 *
 */

#include "common.h"
#include "myshell.h"
#include "Console.h"
#include "Display.h"
#include "Executor.h"
#include "ProcessManager.h"

#include <vector>
#include <sstream>

#include <fcntl.h>
#include <assert.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>

#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/types.h>

/** 测试终端联系 */
static inline bool test_tty(const char * file_name);

/** 定义命令字符串数组 */
static const char* OperandArray[] =
{
    "bg", "cd", "clr", "dir", "echo", "exec", "exit", "fg",
    "help", "jobs", "myshell", "pwd", "set", "test", "time", "umask"
};

Executor::Executor(Console *model, Display *view)
: console_(model), display_(view)
{
    assert(console_ != nullptr);
    assert(display_ != nullptr);

    /** 定义函数指针数组 */
    int i = 0;

    FunctionArray[i] = &Executor::execute_bg;       ++i;
    FunctionArray[i] = &Executor::execute_cd;       ++i;
    FunctionArray[i] = &Executor::execute_clr;      ++i;
    FunctionArray[i] = &Executor::execute_dir;      ++i;
    FunctionArray[i] = &Executor::execute_echo;     ++i;
    FunctionArray[i] = &Executor::execute_exec;     ++i;
    FunctionArray[i] = &Executor::execute_exit;     ++i;
    FunctionArray[i] = &Executor::execute_fg;       ++i;
```

```
    FunctionArray[i] = &Executor::execute_help;     ++i;
    FunctionArray[i] = &Executor::execute_jobs;     ++i;
    FunctionArray[i] = &Executor::execute_myshell;  ++i;
    FunctionArray[i] = &Executor::execute_pwd;      ++i;
    FunctionArray[i] = &Executor::execute_set;      ++i;
    FunctionArray[i] = &Executor::execute_test;     ++i;
    FunctionArray[i] = &Executor::execute_time;     ++i;
    FunctionArray[i] = &Executor::execute_umask;    ++i;

    return;
}

Executor::~Executor()
{
}

sh_err_t Executor::execute(const int argc, char * const argv[], char * const
env[]) const
{
    if (argc == 0)
        return SH_SUCCESS;  // 没有输入命令则无需处理
    else if (argv == nullptr || argv[0] == nullptr)
    {
        assert(false);
        return SH_FAILED;   // 解析可能产生了错误
    }

    /* 挂起命令处理 */
    int& argc_ = const_cast<int&>(argc);
    if (strcmp(argv[argc - 1], "&") == 0)    // 后台挂起
    {
        --argc_;
        if (argc == 0)  // 参数错误
            return SH_ARGS;

        pid_t pid;
        if ((pid = fork()) < 0)
        {
            /* 错误处理 */
            throw "Fork Error, 错误终止";
        }
        else if (pid == 0)
        {
            /* 子进程 */
            setenv("parent", console_->shell_path_env, 1);  // 设置调用子进程的父进
程
            Console::child_process_id = getpid();

            #ifdef _DEBUG_
            printf("child pid: %d\n", console_->process_id);
            #endif

            setpgid(0, 0);
            signal(SIGINT, SIG_DFL);    // 恢复Ctrl C信号
            signal(SIGTSTP, SIG_DFL);   // 恢复Ctrl Z信号
```

```
            char **&argv_ = const_cast<char **&>(argv);
            argv_[argc] = NULL;
            #ifdef _DEBUG_
            Argument_Display(argc, argv);
            #endif

            // 执行命令
            shell_function(argc, argv, env);

            // 执行完成时退出
            return SH_EXIT;
        }
        else
        {
            /* 父进程 */
            #ifdef _DEBUG_
            printf("parent pid: %d\n", pid);
            #endif

            // 添加进程列表
            char **&argv_ = const_cast<char **&>(argv);
            argv_[argc] = NULL;
            unsigned int jobid = console_->AddJob(pid, Running, argc_, argv_);
            // console_->process_id = getpid();    // 可以看到，这里的pid是没有改变的
            console_->child_process_id = pid;

            // 打印当前进程
            char buffer[32];
            snprintf(buffer, 32, "[%u] %d\n", jobid, pid);
            if (write(console_->output_std_fd, buffer, strlen(buffer)) == -1)
                throw std::exception();

            // setpgid(pid, pid);

            // // 将前端设置为子进程
            // tcsetpgrp(STDIN_FILENO, pid);
            // tcsetpgrp(STDOUT_FILENO, pid);
            // tcsetpgrp(STDERR_FILENO, pid);

            // int status;
            // waitpid(pid, &status, WNOHANG);

            // // 将前端设置为父进程
            // tcsetpgrp(STDIN_FILENO, getpid());
            // tcsetpgrp(STDOUT_FILENO, getpid());
            // tcsetpgrp(STDERR_FILENO, getgid());

            return SH_SUCCESS;
        }
    }

    return shell_function(argc, argv, env);
}
```

```
sh_err_t Executor::shell_function(const int argc, char * const argv[], char *
const env[]) const
{
    const char *op = argv[0];
    console_->argc = argc;
    for (int i = 0; i < argc; ++i)
        strncpy(console_->argv[i], argv[i], BUFFER_SIZE);

#ifdef _DEBUG_
    Argument_Display(argc, argv);

    // 以下命令不要求实现，仅供练习使用
    if (strcmp(op, "date") == 0)
    {
        return execute_date(argc, argv, env);
    }
    else if (strcmp(op, "clear") == 0)
    {
        return execute_clear(argc, argv, env);
    }
    else if (strcmp(op, "env") == 0)
    {
        return execute_env(argc, argv, env);
    }
    else if (strcmp(op, "who") == 0)
    {
        return execute_who(argc, argv, env);
    }
    else if (strcmp(op, "mkdir") == 0)
    {
        return execute_mkdir(argc, argv, env);
    }
    else if (strcmp(op, "rmdir") == 0)
    {
        return execute_rmdir(argc, argv, env);
    }
#endif

    /** 二分查找匹配内部命令 */
    int index = Binary_Search(0, sizeof(OperandArray)/sizeof(OperandArray[0]),
op, OperandArray, strcmp);
#ifdef _DEBUG_
    printf("index: %d op: %s\n", index, OperandArray[index>=0?index:0]);
#endif

    if (index >= 0 && index < FunctionNumber)   // 找到了
    {
        MemFuncPtr FunctionPointer = FunctionArray[index];  // 找到对应的函数指针
        return (*this.*FunctionPointer)(argc, argv, env);   // 执行内部函数
    }


    // 其他的命令行输入被解释为程序调用，
    // shell 创建并执行这个程序，并作为自己的子进程
    pid_t pid = getpid(); // 获取当前进程id，用于处理父进程行为
```

```
    if ((pid = vfork()) < 0)
    {
        /* 错误处理 */
        throw "Fork Error, 错误终止";
    }
    else if (pid == 0)
    {
        /* 子进程 */
        setenv("parent", console_->shell_path_env, 1);  // 设置调用子进程的父进程
        int status_code = execvp(argv[0], argv);        // 在子进程之中执行

        if (status_code == -1)
        {
            throw "Execvp Error, terminated incorrectly";
        }

        return SH_UNDEFINED; // 未识别的命令
    }
    else
    {
        /* 父进程 */
        console_->child_process_id = pid;   // 设置子进程pid，用于Ctrl+Z信号处理
        wait(NULL); // 等待子进程结束后再继续执行，保证执行顺序不混乱
        console_->child_process_id = -1;
        return SH_SUCCESS;
    }

    return SH_FAILED;
}

sh_err_t Executor::execute_cd(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "cd")==0 && "unexpected node type");

    std::string path;
    if (argc == 1)
    {
        // 默认无参数时为主目录
        path = console_->home;
    }
    else if (argc == 2)
    {
        path = argv[1];

        #ifdef _DEBUG_
        printf("char: %c %d\n", path[0], (path[0] == '~'));
        #endif

        if (path[0] == '~') // 对于~目录需要特殊判断
        {
            // 将~替换为主目录
            path.replace(0, 1, console_->home);
        }
```

```
        #ifdef _DEBUG_
        printf("Argv: %s\nHome: %s\nPath: %s\n", argv[1], console_->home,
path.c_str());
        #endif
    }
    else
    {
        return SH_ARGS; // 参数错误
    }

    // 更改目录
    int ret = chdir(path.c_str());
    if (ret != 0)   // 打开目录异常
    {
        throw ((std::string)"cd: 无法打开路径 " + path);
    }

    // 重新设置控制台环境与系统环境变量
    if (getcwd(console_->current_working_dictionary, BUFFER_SIZE) != nullptr )
        setenv("PWD", console_->current_working_dictionary, 1);
    else
        throw "get cwd error";

    return SH_SUCCESS;
}

sh_err_t Executor::execute_pwd(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "pwd")==0 && "unexpected node type");
    display_->message(console_->current_working_dictionary);
    display_->message("\n");
    return SH_SUCCESS;
}

sh_err_t Executor::execute_time(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "time")==0 && "unexpected node type");

    // 相当于env命令，从兼容性的角度出发，这里选择调用env命令以兼容Linux用户需求以及后续开发
    return execute_date(argc, argv, env);
}

sh_err_t Executor::execute_clr(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "clr")==0 && "unexpected node type");

    // 相当于clear命令，从兼容性的角度出发，这里选择调用env命令以兼容Linux用户需求以及后续开
发
    return execute_clear(argc, argv, env);
}
```

```
sh_err_t Executor::execute_dir(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "dir")==0 && "unexpected node type");

    std::string real_path;
    if (argc == 1)
    {
        // 默认无参数时为当前目录
        real_path = console_->current_working_dictionary;
    }
    else if (argc == 2)
    {
        real_path = argv[1];
        if (real_path[0] == '~') // 对于~目录需要特殊判断
        {
            // 将~替换为主目录
            real_path.replace(0, 1, console_->home);
        }
    }
    else
    {
        return SH_ARGS; // 参数错误
    }

    int ret;                 // 用于接受返回值
    DIR *direction_pointer; // 目录指针
    if ((direction_pointer = opendir(real_path.c_str())) == NULL)
    {
        throw ((std::string)"dir: 无法打开路径 " + real_path);
    }

    // 临时将进程目录调整为指定目录
    ret = chdir(real_path.c_str());
    if (ret != 0)   // 打开目录异常
    {
        throw ((std::string)"dir: 无法打开路径 " + real_path);
    }

    struct dirent *entry;   // 目录内容
    while ((entry = readdir(direction_pointer)) != NULL)
    {
        struct stat stat_buffer;    // 存储stat结构
        lstat(entry->d_name, &stat_buffer); // 根据文件名获得文件stat结构

        char buffer[BUFFER_SIZE];
        if (S_ISDIR(stat_buffer.st_mode))   // 检测该数据项是否是一个目录
        {
            // 数据项是一个目录

            if (strcmp(".", entry->d_name) == 0 ||
                strcmp("..", entry->d_name) == 0)
            {
                // 如果是.或者..目录，则不显示
                continue;
```

```
            }

            // 目录用蓝色显示
            snprintf(buffer, BUFFER_SIZE, "\033[34m%s\033[0m  ", entry->d_name);
            if (console_->redirect_output == false)
                display_->message(buffer);
            else
            {
                display_->message(entry->d_name);
                display_->message("  ");
            }
        }
        else
        {
            // 普通文件
            switch (entry->d_type)
            {
                case DT_UNKNOWN:    // 目录文件未知文件用红色
                    snprintf(buffer, BUFFER_SIZE, "\033[31m%s\033[0m  ", entry-
>d_name);
                    break;

                case DT_REG:        // 目录文件普通文件用白色
                    if (access(entry->d_name, X_OK) == 0)    // 可执行文件除外，用绿
色
                        snprintf(buffer, BUFFER_SIZE, "\033[32m%s\033[0m  ",
entry->d_name);
                    else
                        snprintf(buffer, BUFFER_SIZE, "\033[37m%s\033[0m  ",
entry->d_name);
                    break;

                default:            // 其他文件用青色
                    snprintf(buffer, BUFFER_SIZE, "\033[36m%s\033[0m  ", entry-
>d_name);
                    break;
            }
            if (console_->redirect_output == false)
                display_->message(buffer);
            else
            {
                display_->message(entry->d_name);
                display_->message("  ");
            }
        }
    }
    display_->message("\n");

    // 结束时将目录更改回当前目录
    ret = chdir(console_->current_working_dictionary);
    if (ret != 0)   // 打开目录异常
    {
        throw ((std::string)"dir: 无法打开路径 " + real_path);
    }
```

```
    ret = closedir(direction_pointer);
    if (ret == -1)  // 关闭目录流异常
    {
        throw "dir: 关闭目录流异常";
    }

    return SH_SUCCESS;
}

sh_err_t Executor::execute_set(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "set")==0 && "unexpected node type");

    // 相当于env命令，从兼容性的角度出发，这里选择调用env命令以兼容Linux用户需求
    return execute_env(argc, argv, env);
}

sh_err_t Executor::execute_echo(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "echo")==0 && "unexpected node type");

    for (int i = 1; i < argc; ++i)
    {
        // 多个空格和制表符被缩减为一个空格
        if (i > 1)
            display_->message(" ");

        display_->message(argv[i]);
    }
    display_->message("\n");

    return SH_SUCCESS;
}

sh_err_t Executor::execute_help(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "help")==0 && "unexpected node type");

    FILE* fp = fopen("README.md", "r");
    if (fp == nullptr)
    {
        return SH_FAILED;
    }

    char buffer[BUFFER_SIZE*2];
    if (fgets(buffer, BUFFER_SIZE, fp) == nullptr)
        return SH_FAILED;    // 忽略首行

    size_t size = fread(buffer, 1, BUFFER_SIZE*2, fp);
    if (size < 0)
    {
        return SH_FAILED;
```

```
    }

    if (fclose(fp) == -1)
    {
        throw std::exception();
    }

    display_->message(buffer);
    display_->message("\n");

    return SH_SUCCESS;
}

sh_err_t Executor::execute_exit(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "exit")==0 && "unexpected node type");
    return SH_EXIT;
}

sh_err_t Executor::execute_date(const int argc, char * const argv[], char *
const env[]) const
{
    // assert(strcmp(argv[0], "date")==0 && "unexpected node type");

    //获取当前时间
    time_t t = time(NULL);
    struct tm *ptr = localtime(&t);

    //生成返回信息
    // char weekday[16], month[16];
    char date[256];
    // strftime(weekday, 16, "%A", ptr);
    // strftime(month, 16, "%B", ptr);
    strftime(date, 256, "%c", ptr);

    // char buffer[BUFFER_SIZE];
    // snprintf(buffer, BUFFER_SIZE, "%s %s %s\n", weekday, month, date);

    // display_->message(buffer);
    display_->message(date);
    display_->message("\n");

    return SH_SUCCESS;
}

sh_err_t Executor::execute_clear(const int argc, char * const argv[], char *
const env[]) const
{
    display_->message("\x1b[H\x1b[2J");    // 输出清屏控制 \x1b[H\x1b[2J

    return SH_SUCCESS;
}
```

```
sh_err_t Executor::execute_env(const int argc, char * const argv[], char * const
env[]) const
{
    extern char **environ; //env variables
    char ***update_env = const_cast<char ***>(&env);
    *update_env = environ;

    while(*env)
    {
        char buffer[BUFFER_SIZE];
        snprintf(buffer, BUFFER_SIZE, "%s\n", *env++);
        display_->message(buffer);
    }

    return SH_SUCCESS;
}

sh_err_t Executor::execute_who(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "who")==0 && "unexpected node type");
    display_->message(console_->user_name);
    display_->message("\n");
    return SH_SUCCESS;
}

sh_err_t Executor::execute_mkdir(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "mkdir")==0 && "unexpected node type");

    const char * path = argv[1];
    if (mkdir(path, S_IRWXU) == 0)
    {
        return SH_SUCCESS;
    }
    else
    {
        return SH_FAILED;
    }

}

sh_err_t Executor::execute_rmdir(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "rmdir")==0 && "unexpected node type");

    if (rmdir(argv[1]) == 0)
    {
        return SH_SUCCESS;
    }
    else
    {
        return SH_FAILED;
```

```
    }
}

sh_err_t Executor::execute_bg(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "bg")==0 && "unexpected node type");

    if (argc == 1)
        return SH_SUCCESS;

    unsigned int job_id = String_to_Number<unsigned int>(argv[1]);
    int id = console_->process_manager->BackGround(job_id);
    if (id == 0)
    {
        char buffer[BUFFER_SIZE];
        snprintf(buffer, BUFFER_SIZE, "bg: job %u already in background\n",
job_id);
        display_->message(buffer);
    }

    if (id == -1)
    {
        char buffer[BUFFER_SIZE];
        snprintf(buffer, BUFFER_SIZE, "bg: %u : no such job\n", job_id);
        display_->message(buffer);
    }

    return SH_SUCCESS;
}

sh_err_t Executor::execute_fg(const int argc, char * const argv[], char * const
env[]) const
{
    assert(strcmp(argv[0], "fg")==0 && "unexpected node type");

    if (argc == 1)
        return SH_SUCCESS;

    unsigned int job_id = String_to_Number<unsigned int>(argv[1]);
    int id = console_->process_manager->ForeGround(job_id);
    if (id == -1)
    {
        char buffer[BUFFER_SIZE];
        snprintf(buffer, BUFFER_SIZE, "bg: %u : no such job\n", job_id);
        display_->message(buffer);
    }

    return SH_SUCCESS;
}

sh_err_t Executor::execute_jobs(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "jobs")==0 && "unexpected node type");
```

```
    console_->ConsoleJobList();

    return SH_SUCCESS;
}

sh_err_t Executor::execute_exec(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "exec")==0 && "unexpected node type");

    // 只有一个参数时不做处理
    if (argc == 1)
        return SH_SUCCESS;

    int status_code = execvp(argv[1], argv+1);        // 在子进程之中执行

    if (status_code == -1)
    {
        throw "Execvp Error, terminated incorrectly";
    }

    return SH_UNDEFINED; // 未识别的命令
}

sh_err_t Executor::execute_test(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "test")==0 && "unexpected node type");

    bool ret = false;
    if (argc == 1)  // 空字符串，false
    {
        ret = false;
    }
    else if (argc == 2) // 单目运算
    {
        if (strcmp(argv[1], "!") == 0 || strcmp(argv[1], "-z") == 0)
            ret = true;
        else
            ret = false;
    }
    else
    {
        if (strcmp(argv[1], "!"))   // 第二个参数不是!
        {
            if (argc == 3 || argc == 4) // 双目运算
            {
                // 文件测试 与 部分字符串测试
                ret = Executor::test_file_state(argc, argv)
                    | Executor::test_number_compare(argc, argv)
                    | Executor::test_string_compare(argc, argv);
            }
            else
            {
```

```
                return SH_ARGS;
            }
        }
    }

    if (console_->GetOutputRedirect() == false) // 如果是在终端显示就产生正误提示
    {
        if (ret)
            display_->message("true\n");
        else
            display_->message("false\n");
    }

    return SH_SUCCESS;
}

sh_err_t Executor::execute_umask(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "umask")==0 && "unexpected node type");

    if (argc == 1)
    {
        // 没有参数，显示当前掩码
        char buffer[16];
        snprintf(buffer, 16, "%04o\n", console_->umask_);   // 以八进制显示，0补齐
        display_->message(buffer);
    }
    else if (argc == 2) // 有一个输入的参数
    {
        // 使用函数模板实现进制转换
        console_->umask_ = String_to_Number<mode_t>(argv[1]);

        if (argv[1][0] == '0')
        {
            if (strlen(argv[1]) >= 2 && argv[1][1] == 'x')  // 十六进制
                console_->umask_ = Hexadecimal_to_Decimal(console_->umask_);
            else                                            // 八进制
                console_->umask_ = Octal_to_Decimal(console_->umask_);
        }

        #ifdef _DEBUG_
        printf("mask: %04u %04o\n", console_->umask_, console_->umask_);
        #endif
        umask(console_->umask_);
    }
    else
    {
        return SH_ARGS; // 参数错误
    }

    return SH_SUCCESS;
}
```

```
sh_err_t Executor::execute_myshell(const int argc, char * const argv[], char *
const env[]) const
{
    assert(strcmp(argv[0], "myshell")==0 && "unexpected node type");

    std::vector<std::string> FileList;
    if (argc == 1)
    {
        /*如果shell 被调用时没有使用参数，
        它会在屏幕上显示提示符请求用户输入*/
        while (1)   // 循环直到用户有输入
        {
            display_->prompt();

            int len;
            char input[BUFFER_SIZE];
            len = display_->InputCommand(input, BUFFER_SIZE);

            if (len == 1 || len < 0)
                continue;   // 非有效输入
            if (len == 0)
                return SH_EXIT; // EOF

            #ifdef _DEBUG_
            printf("len: %d\n", len);
            #endif
            input[len-1] = '\0'; // 去掉末尾的\n

            int& argc_ = const_cast<int&>(argc);        // 引用
            // char **argv_ = const_cast<char **>(argv);   // 指针

            std::istringstream line(input); // 字符串流
            std::string word;               // 分割出的字符串

            while (std::getline(line, word, ' '))
            {
                word = String_Trim(word);   // 裁剪
                if (word == "")
                    continue;

                ++argc_;
                FileList.emplace_back(word);  // 需要深拷贝
            }

            if (argc == 1)
                continue;   // 未能读到有效输入

            #ifdef _DEBUG_
            Argument_Display(argc, argv);   // 调试
            #endif

            break;
        }
    }
    else
```

```
    {
        for (int i = 1; i < argc; ++i) // 顺序执行
            FileList.push_back(argv[i]);    // 将参数加入向量列表
    }

    assert(argc > 1);   // 判断

    int input_fd = console_->input_file_descriptor; // 存储当前控制台的输入fd
    for (std::string File : FileList)
    {
        try
        {
            int fd = open(File.c_str(), O_RDONLY);   // 打开文件
            if (fd < 0) // 打开错误处理
            {
                throw std::exception();
            }

            #ifdef _DEBUG_
            fprintf(stdout, "FD: %d Input: %d Output: %d\n", fd, console_-
>input_file_descriptor, console_->output_file_descriptor);
            #endif

            console_->input_file_descriptor = fd;   // 更改输入

            // 执行循环
            SHELL::shell_loop(console_, display_, const_cast<Executor *>(this),
const_cast<char **>(env));

            int state_code = close(fd); // 关闭文件
            if (state_code != 0)    // 关闭错误处理
            {
                throw std::exception();
            }
        }
        catch(...)
        {
            puts("every thing");
            std::string msg = "\e[1;31m[ERROR]\e[0m";
            msg = msg + "myshell" + ": (" + File + ") " + strerror(errno) +
"\n";
            display_->message(msg.c_str());
        }

    }

    console_->input_file_descriptor = input_fd; // 恢复控制台的input fd

    return SH_SUCCESS;
}

static inline bool test_tty(const char * file_name)
{
    try
    {
```

```
        int fd = open(file_name, S_IREAD);  // 打开文件获取文件描述符
        bool tty = isatty(fd);              // 判断是否为终端
        close(fd);                          // 关闭文件
        return tty;
    }
    catch(...)
    {
        return false;                       // 如果有任何中断则返回false
    }
}

bool Executor::test_file_state(const int argc, const char * const argv[])
{
    assert(argc == 3 || argc == 4);

    if (argc == 3)
    {
        struct stat file_stat;

        if (lstat(argv[2], &file_stat) < 0) // 文件不存在
        {
            return false;   // 不存在一定是false
        }

        // 对文件测试参数进行判断
        switch (String_Hash(argv[1]))   // 为了形式上的优雅，使用switch语句
        {
            /* 存在性判断 */
            case String_Hash("-e"):  // 存在判断
                return true;

            /* 文件类型判断 */
            case String_Hash("-f"):  // 普通文件
                return S_ISREG(file_stat.st_mode);

            case String_Hash("-d"):  // 目录文件
                return S_ISDIR(file_stat.st_mode);

            case String_Hash("-c"):  // 字符特殊文件
                return S_ISCHR(file_stat.st_mode);

            case String_Hash("-b"):  // 块特殊文件
                return S_ISBLK(file_stat.st_mode);

            case String_Hash("-p"):  // 管道文件
                return S_ISFIFO(file_stat.st_mode);

            case String_Hash("-L"):  // 符号链接文件
                return S_ISLNK(file_stat.st_mode);

            case String_Hash("-S"):  // 套接字文件
                return S_ISSOCK(file_stat.st_mode);

            /* 文件权限判断 */
            case String_Hash("-r"):  // 可读文件
```

```
                return access(argv[1], R_OK);

            case String_Hash("-w"):  // 可写文件
                return access(argv[1], W_OK);

            case String_Hash("-x"):  // 可执行文件
                return access(argv[1], X_OK);

            case String_Hash("-O"):  // 所有者文件
                return file_stat.st_uid == getuid();

            case String_Hash("-G"):  // 组 文件
                return file_stat.st_gid == getgid();

            /* 文件属性判断 */
            case String_Hash("-u"):  // 用户位属性SUID
                return S_ISUID & file_stat.st_mode;

            case String_Hash("-g"):  // 组位属性GUID
                return S_ISGID & file_stat.st_mode;

            case String_Hash("-k"):  // Sticky bit属性
                return S_ISVTX & file_stat.st_mode;

            case String_Hash("-s"):  // 文件长度非0
                return file_stat.st_size > 0;

            case String_Hash("-t"):  // 文件描述符联系终端
                return test_tty(argv[2]);

            /* 其他情况返回错误 */
            default:
                return false;
        }
    }
    else
    {
        struct stat file_stat1, file_stat2;

        if (lstat(argv[1], &file_stat1) < 0) // 文件不存在
        {
            return false;   // 不存在一定是false
        }
        if (lstat(argv[3], &file_stat2) < 0) // 文件不存在
        {
            return false;   // 不存在一定是false
        }

        // 对文件测试参数进行判断
        switch (String_Hash(argv[2]))   // 为了形式上的优雅，使用switch语句
        {
            case String_Hash("-nt"):    // 判断file1是否比file2新
                return test_timespec_newer(file_stat1.st_mtim,
file_stat2.st_mtim);
```

```
            case String_Hash("-ot"):    // 判断file1是否比file2旧
                return test_timespec_older(file_stat1.st_mtim,
file_stat2.st_mtim);

            case String_Hash("-ef"):    // 判断file1与file2是否为同一个文件
                return file_stat1.st_ino == file_stat2.st_ino;

            default:                    /* 其他情况返回错误 */
                return false;
        }
    }
}

bool Executor::test_number_compare(const int argc, const char * const argv[])
{
    if (argc != 4)
        return false;


    int number1 = String_to_Number<int>(argv[1]);
    int number2 = String_to_Number<int>(argv[3]);

    // 对整数测试参数进行判断
    switch (String_Hash(argv[2]))   // 为了形式上的优雅，使用switch语句
    {
        case String_Hash("-eq"):
        case String_Hash("=="):     // 判断number1是否与number2相等
            return number1 == number2;

        case String_Hash("-ne"):
        case String_Hash("!="):     // 判断number1是否与number2不相等
            return number1 != number2;

        case String_Hash("-ge"):
        case String_Hash(">="):     // 判断number1是否大于等于number2
            return number1 >= number2;

        case String_Hash("-gt"):
        case String_Hash(">"):      // 判断number1是否大于number2
            return number1 > number2;

        case String_Hash("-le"):
        case String_Hash("<="):     // 判断number1是否小于等于number2
            return number1 <= number2;

        case String_Hash("-lt"):
        case String_Hash("<"):      // 判断number1是否小于number2
            return number1 < number2;

        default:                    /* 其他情况返回错误 */
            return false;
    }

}
```

```
bool Executor::test_string_compare(const int argc, const char * const argv[])
{
    assert(argc == 3 || argc == 4);

    if (argc == 3)
    {
        // 对字符串测试参数进行判断
        switch (String_Hash(argv[1]))   // 为了形式上的优雅，使用switch语句
        {
            /* 存在性判断 */
            case String_Hash("-n"):  // 存在判断
                return true;

            /* 其他情况返回错误 */
            default:
                return false;
        }
    }
    else
    {
        // 对字符串测试参数进行判断
        switch (String_Hash(argv[2]))   // 为了形式上的优雅，使用switch语句
        {
            case String_Hash("="):     // 判断string1是否与string2相等
                return !strcmp(argv[1], argv[3]);

            case String_Hash("!="):     // 判断string1是否与string2不相等
                return strcmp(argv[1], argv[3]);

            case String_Hash("\\>"):      // 判断string1是否大于string2
                return strcmp(argv[1], argv[3]) > 0;

            case String_Hash("\\<"):      // 判断string1是否小于string2
                return strcmp(argv[1], argv[3]) < 0;

            default:
                return false;
        }
    }
}
```

src/lexer.l

```
%{
    #define MAX_ARGUMENT_NUMBER 128
    char *_argvector[MAX_ARGUMENT_NUMBER];  // 解析参数
    int _argcounter = 0;                    // 参数个数
%}

WORD     [a-zA-Z0-9\/\.\-~]+
STRINGLITERAL \"(\\.|[^\\"])*\"
REDIRCT  [0-9><]+
OPERATOR [<>=!]=
```

```
SPECIAL  [()|&*!]

%%
    _argcounter = 0;
    _argvector[0] = NULL;

{WORD}|{SPECIAL}|{REDIRCT}|{OPERATOR}|{STRINGLITERAL} {
    if(_argcounter < MAX_ARGUMENT_NUMBER-1)
    {
        _argvector[_argcounter++] = (char *)strdup(yytext);
        _argvector[_argcounter] = NULL;
    }
}

\n  return (int)_argvector; // 解析到换行符时结束

[ \t]+

\#[^\n]*    ;   // 忽略以#开头的注释

. {
  char str[128] = {0};
  sprintf(str, "Unrecognized token [%s] in input sql.", yytext);
//   ParserSetError(str);
}

%%

int yywrap()
{
    return 1;
}

int yy_lexer(int *argc, char ***argv)
{
    yylex();

    *argc = _argcounter;
    *argv = _argvector;

    return 0;
}
```

src/Parser.cpp

```
/**
 * @file Parser.cpp
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 语法分析
 * @version 1.0
 * @date 2022-07-19
 *
```

```
 * @copyright Copyright (c) 2022
 *
 */

#include "common.h"
#include "Parser.h"
#include "Console.h"
#include "Display.h"
#include "Executor.h"

#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <exception>
#include <sys/wait.h>

/** @brief 根据错误类型给出错误信息 */
static const char * shell_error_message(sh_err_t err);

bool Parser::shell_pipe(Console *model, Display* view, Executor* controller,
int& argc, char *argv[], char *env[])
{
    int count = 0;
    char *args[MAX_ARGUMENT_NUMBER];

    int input_fd = model->GetInputFD();     // 记录下原始输入
    int output_fd = model->GetOutputFD();   // 记录下原始输出

    int i = 0;
    do
    {
        if (strcmp(argv[i], "|") != 0)   // 不是管道符
        {
            args[count] = argv[i];
            count++;
        }
        else
        {
            args[count] = NULL; // 命令结束

            int channel[2];
            // channel[0] : read
            // channel[1] : write
            if (pipe(channel) == -1)
                throw "Pipe Error, 错误终止";

            #ifdef _DEBUG_
            printf("channel: read %d write %d\n", channel[0], channel[1]);
            #endif

            pid_t pid = fork(); // 分裂进程，fork返回的是子进程的pid
            if (pid < 0)
            {
                /* 错误处理 */
                throw "Fork Error, 错误终止";
```

```
            }
            else if (pid == 0)
            {
                /* 子进程 */
                setenv("parent", getenv("shell"), 1);  // 设置调用子进程的父进程
                close(channel[0]);  // 关闭读进程
                int fd = channel[1];

                model->SetOutputFD(fd);
                model->SetOutputRedirect();

                dup2(fd, STDOUT_FILENO); // 重定向标准输出至channel[1]

                shell_execute(model, view, controller, count, args, env);

                /* 在shell execute里包含了重定向后文件的关闭等操作，此处可以不必再次关闭
*/

                return EXIT;
            }
            else
            {
                /* 父进程 */
                wait(NULL); // 为保持逻辑，需要等子进程输出之后再继续

                close(channel[1]);
                int fd = channel[0];

                model->SetInputFD(fd);
                model->SetInputRedirect();

                dup2(fd, STDIN_FILENO); // 重定向标准输入至fd

                count = 0;
            }

        }

        ++i;
    } while (i < argc);

    #ifdef _DEBUG_
    printf("Parent Process\n");
    #endif
    /* 最后一条命令也要执行 */
    args[count] = NULL; // 命令结束
    bool exit_state = shell_execute(model, view, controller, count, args, env);

    #ifdef _DEBUG_
    printf("pipe: Input %d Output %d Error %d\n", model->GetInputFD(), model-
>GetOutputFD(), model->GetErrorFD());
    #endif

    /* 无论重定向如何发生，最后将其还原为本来的状态
       可能执行时已关闭文件，但未正确设置原始状态 */
```

```
    if (model->GetInputFD() != input_fd)    // 如果发生了输入重定向且未关闭
    {
        // dup2(model->GetSTDIN(), STDIN_FILENO);
        model->SetInputFD(input_fd);  // 恢复输入
        // model->ResetInputRedirect();    // 恢复状态
    }

    if (model->GetOutputFD() != output_fd)    // 如果发生了输出重定向
    {
        // dup2(model->GetSTDOUT(), STDOUT_FILENO);
        model->SetOutputFD(output_fd);  // 恢复输出
        // model->ResetOutputRedirect();    // 恢复状态
    }

    return exit_state;
}

int Parser::shell_parser(Console *model, Display* view, Executor* controller,
int& argc, char *argv[], char *env[])
{
    if (argc == 0 || strcmp(argv[0], "test") == 0)
        return 0;   // 无参，不需处理

    for (int index = argc-1; index > 0; --index)    // 从末尾开始往前扫描，第一个不必
扫
    {
        std::string arg(argv[index]);    // 使用string类处理

        /** 重定向处理 */

        /** 标准输入重定向 */
        if (arg == "<" || arg == "0<")
        {
            if (index + 1 == argc)  // 重定向符号是最后一个输入
            {
                throw "语法解析错误";
            }

            if (model->GetInputRedirect())      // 如果已经设置了重定向状态了
            {
                throw "多重重定向错误";
            }

            const char * input_file = argv[index + 1];
            int fd = open(input_file, O_RDONLY);
            if (fd < 0)
                throw std::exception();

            model->SetInputFD(fd);
            model->SetInputRedirect();

            dup2(fd, STDIN_FILENO); // 重定向标准输入至fd

            for (int jump = index + 2; jump < argc; ++jump)
                argv[jump-2] = argv[jump];
```

```
            argc = argc - 2;
            argv[argc] = NULL;
        }

        /** 标准输出重定向 */
        if (arg == ">" || arg == "1>")
        {
            if (index + 1 == argc)  // 重定向符号是最后一个输入
            {
                throw "语法解析错误";
            }

            if (model->GetOutputRedirect())      // 如果已经设置了重定向状态了
            {
                throw "多重重定向错误";
            }

            const char * output_file = argv[index + 1];
            int fd = open(output_file, O_WRONLY | O_TRUNC | O_CREAT, 0777&
(~model->GetMask()));
            if (fd < 0)
                throw std::exception();

            model->SetOutputFD(fd);
            model->SetOutputRedirect();

            dup2(fd, STDOUT_FILENO); // 重定向标准输出至fd

            for (int jump = index + 2; jump < argc; ++jump)
                argv[jump-2] = argv[jump];
            argc = argc - 2;
            argv[argc] = NULL;
        }

        /** 标准错误输出重定向 */
        if (arg == "2>")
        {
            if (index + 1 == argc)  // 重定向符号是最后一个输入
            {
                throw "语法解析错误";
            }

            if (model->GetErrorRedirect())      // 如果已经设置了重定向状态了
            {
                throw "多重重定向错误";
            }

            const char * output_file = argv[index + 1];
            int fd = open(output_file, O_WRONLY | O_TRUNC | O_CREAT, 0777&
(~model->GetMask()));
            if (fd < 0)
                throw std::exception();

            model->SetErrorFD(fd);
            model->SetErrorRedirect();
```

```
            dup2(fd, STDERR_FILENO); // 重定向标准错误输出至fd

            for (int jump = index + 2; jump < argc; ++jump)
                argv[jump-2] = argv[jump];
            argc = argc - 2;
            argv[argc] = NULL;
        }

        /** 追加 */
        if (arg == ">>" || arg == "1>>")
        {
            /* 词法解析时应该识别<和>符号的闭包 */
            #ifdef _DEBUG_
            Argument_Display(argc, argv);
            #endif

            if (index + 1 == argc)  // 重定向符号是最后一个输入
            {
                throw "语法解析错误";
            }

            if (model->GetOutputRedirect())      // 如果已经设置了重定向状态了
            {
                throw "多重重定向错误";
            }

            const char * output_file = argv[index + 1];
            int fd = open(output_file, O_WRONLY | O_APPEND | O_CREAT, 0777&
(~model->GetMask()));
            if (fd < 0)
                throw std::exception();

            model->SetOutputFD(fd);
            model->SetOutputRedirect();

            dup2(fd, STDOUT_FILENO); // 重定向标准输出至fd

            for (int jump = index + 2; jump < argc; ++jump)
                argv[jump-2] = argv[jump];
            argc = argc - 2;
            argv[argc] = NULL;
        }
    }

    return 0;
}

bool Parser::shell_execute(Console *model, Display* view, Executor* controller,
int& argc, char *argv[], char *env[])
{
    // Argument_Display(argc, argv);

    int input_fd = model->GetInputFD();     // 记录下原始输入
    int output_fd = model->GetOutputFD();   // 记录下原始输出
```

```
    int error_fd = model->GetErrorFD();     // 记录下原始错误输出

    // 执行命令
    try
    {
        // Parser 语法分析
        Parser::shell_parser(model, view, controller, argc, argv, env);

        // 执行命令
        sh_err_t err = controller->execute(argc, argv, env);

        // 根据返回状态判断
        if (err == SH_EXIT)
        {
            view->show(); // 将退出信息显示
            return true;
        }
        else if (err != SH_SUCCESS)
        {
            throw err;
        }

        view->show();       // 显示输出信息
        view->clear();      // 清空结果
    }
    catch(const std::exception& e)
    {
        fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno),
e.what());
    }
    catch(const sh_err_t e)
    {
        fprintf(stderr, "\e[1;31m[ERROR]\e[0m MyShell: %s\n",
shell_error_message(e));
    }
    catch(const char * message)
    {
        fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno),
message);
    }
    catch(...)
    {
        fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s\n", strerror(errno));
    }

    if (model->GetInputRedirect())    // 如果发生了输入重定向
    {
        int state_code = close(model->GetInputFD()); // 关闭文件
        if (state_code != 0)                         // 关闭错误处理
            throw std::exception();

        dup2(model->GetSTDIN(), STDIN_FILENO);
        model->SetInputFD(input_fd);  // 恢复输入
        model->ResetInputRedirect();    // 恢复状态
    }
```

```
    if (model->GetOutputRedirect())    // 如果发生了输出重定向
    {
        int state_code = close(model->GetOutputFD()); // 关闭文件
        if (state_code != 0)                          // 关闭错误处理
            throw std::exception();

        dup2(model->GetSTDOUT(), STDOUT_FILENO);
        model->SetOutputFD(output_fd);  // 恢复输出
        model->ResetOutputRedirect();    // 恢复状态
    }

    if (model->GetErrorRedirect())    // 如果发生了错误输出重定向
    {
        int state_code = close(model->GetErrorFD()); // 关闭文件
        if (state_code != 0)                          // 关闭错误处理
            throw std::exception();

        dup2(model->GetSTDERR(), STDERR_FILENO);
        model->SetErrorFD(error_fd);  // 恢复错误输出
        model->ResetErrorRedirect();    // 恢复状态
    }

    return false;
}

/** @brief 根据错误类型给出错误信息 */
static const char * shell_error_message(sh_err_t err)
{
    switch (err)
    {
        case SH_FAILED:
            return "Shell Failed. 错误";
        case SH_UNDEFINED:
            return "Undifined command. 未定义的命令";
        case SH_ARGS:
            return "Argument error. 参数错误";

        default:
            return "Unknown error. 未知错误";
    }
}
```

src/ProcessManager.cpp

```
/**
 * @file ProcessManager.cpp
 * @author your name (you@domain.com)
 * @brief 进程管理器
 * @version 1.0
 * @date 2022-07-23
 *
 * @copyright Copyright (c) 2022
```

```
 *
 */

#include "config.h"
#include "Console.h"
#include "BinaryHeap.h"
#include "ProcessManager.h"

#include <stdio.h>
#include <string.h>
#include <iostream>
#include <sys/wait.h>
#include <sys/types.h>

job_unit::job_unit(unsigned int _id, int _pid, job_state _state, int _argc, char
* _argv[])
            : id(_id), pid(_pid), state(_state), argc(_argc)
{
    // argv 必须进行深拷贝，否则释放argv后将不再有，还会造成段错误
    assert(argc < MAX_ARGUMENT_NUMBER);
    for (int i = 0; i < argc; ++i)
        strncpy(argv[i], _argv[i], BUFFER_SIZE);
}

void job_unit::PrintJob(int output_fd)
{
    // if (argc <= 0)  // 参数错误
    // {
    //     assert(false && "argument error");
    //     return;
    // }

    const char *State_;
    switch (state)  // 状态映射
    {
        case Running:                                       // 正在运行
            State_ = "Running";
            break;
        case Stopped:                                       // 停止运行
            State_ = "Stopped";
            break;
        case Done:                                          // 完成运行
            State_ = "Done";
            break;
        case Terminated:                                    // 终止运行
            State_ = "Terminated";
            break;
    }

    // 状态打印
    char buffer[BUFFER_SIZE];
    ssize_t write_state;
    snprintf(buffer, BUFFER_SIZE-1, "[%u]%c\t%s\t\t\t\t\t\t", id, ' ', State_);
    write_state = write(output_fd, buffer, strlen(buffer));
    if (write_state == -1)
```

```
        throw std::exception();

    // 参数打印
    if (argc > 0)
    {
        write_state = write(output_fd, argv[0], strlen(argv[0])); // 确保行末无多余
的空格
        if (write_state == -1)
            throw std::exception();
        for (int i = 1; i < argc; ++i)
        {
            write_state = write(output_fd, " ", 1);   // 打印空格
            if (write_state == -1)
                throw std::exception();

            write_state = write(output_fd, argv[i], strlen(argv[i])); // 打印参数
            if (write_state == -1)
                throw std::exception();
        }
    }

    write_state = write(output_fd, "\n", 1);   // 打印换行符
    if (write_state == -1)
        throw std::exception();
}

ProcessManager::ProcessManager(/* args */)
{
    unsigned int job_id[MAX_PROCESS_NUMBER];
    for (unsigned int i = 1; i <= MAX_PROCESS_NUMBER; ++i)
        job_id[i-1] = i;        // 初始化工作进程id池
    job_heap = new BinaryHeap<unsigned int>(job_id, MAX_PROCESS_NUMBER);

    #ifdef _DEBUG_
    for (unsigned int i = 1; i <= MAX_PROCESS_NUMBER; ++i)
        printf("heap: %u\n", job_heap->extract());
    #endif
}

ProcessManager::~ProcessManager()
{
    delete job_heap;
}

void ProcessManager::PrintJobList(int output_fd) const
{
    for (auto job : jobs)
    {
        job.PrintJob(output_fd);
    }
}

void ProcessManager::PrintJobListDone(int output_fd)
{
    job_unit *pre_job = nullptr;
```

```
    for (auto job : jobs)
    {
        #ifdef _DEBUG_
        printf("Id: %u pid: %d\n", job.id, job.pid);
        #endif
        if (pre_job != nullptr)     // 内存回收
        {
            this->JobRemove(pre_job);
            pre_job = nullptr;
        }

        /* waitpid 在WNOHANG参数下 如果子进程已经结束，则返回子进程的pid；
        如果子进程还未结束，则返回0； 如果发生错误，则返回-1 */
        int stat_loc, wait_pid = waitpid(job.pid, &stat_loc, WNOHANG);
        #ifdef _DEBUG_
        printf("id: %u pid: %d wait: %d stat: %d\n", job.id, job.pid, wait_pid,
stat_loc);
        #endif
        if (wait_pid == job.pid) // 已经结束
        {
            job.state = Done;
            job.PrintJob();
            pre_job = &job;
        }
        else if (wait_pid < 0)  // 发生错误
        {
            throw std::exception();
        }
    }

    if (pre_job != nullptr)     // 内存回收
        this->JobRemove(pre_job);
}

unsigned int ProcessManager::JobInsert(int pid, job_state state, int argc, char
*argv[])
{
    try
    {
        unsigned int id = job_heap->extract();  // 从id池取出最小的id
        job_unit* newJob = new job_unit(id, pid, state, argc, argv);
        #ifdef _DEBUG_
        newJob->PrintJob();
        #endif
        jobs.emplace(*newJob);   // 加入集合
        return id;
    }
    catch (std::exception& e)
    {
        std::cerr << e.what() << '\n';
        return 0;
    }
}
```

```
void ProcessManager::JobRemove(job_unit * job)
{
    assert(job->id > 0);
    job_heap->insert(job->id);  // 将id放回id池中
    jobs.erase(*job);             // 移出集合
    // delete job;  // 因为在set里面存放的不是指针了，在erase set的时候已经完成了析构
    return;
}

void ProcessManager::JobRemove(std::set<job_unit>::iterator& job)
{
    job_heap->insert(job->id);  // 将id放回id池中
    jobs.erase(*job);             // 移出集合
    // delete job;  // 因为在set里面存放的不是指针了，在erase set的时候已经完成了析构
    return;
}

int ProcessManager::ForeGround(unsigned int jobid)
{
    for (auto job : jobs)
    {
        if (job.id == jobid)
        {
            Console::child_process_id = job.pid;
            setpgid(job.pid, getgid());

            // 将前端设置为子进程
            tcsetpgrp(STDIN_FILENO, job.pid);
            tcsetpgrp(STDOUT_FILENO, job.pid);
            tcsetpgrp(STDERR_FILENO, job.pid);

            job.state = Running;    // 继续运行

            kill(job.pid, SIGCONT); // 向子进程发送SIGCONT信号
            while(waitpid(Console::child_process_id, NULL, WNOHANG) == 0 &&
Console::child_process_id >= 0);

            Console::child_process_id = -1; // 等待子进程结束，并且将子进程状态清除
            JobRemove(&job);

            return jobid;
        }
    }

    return -1;  // 未找到该子进程
}


int ProcessManager::BackGround(unsigned int jobid)
{
    for (auto job : jobs)
    {
        if (job.id == jobid)
        {
            if (job.state == Running)
```

```
                return 0;

            job.state = Running;    // 修改子进程状态

            kill(job.pid, SIGCONT); // 发送继续执行信号SIGCONT

            return jobid;
        }
    }

    return -1;  // 未找到该子进程
}
```

src/myshell.cpp

```
// 程序：命令行解释器
// 作者：邱日宏 3200105842

/**
 * @file myshell.cpp
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief myshell程序的main函数，负责调用各个接口并实现myshell功能
 * @version 1.0
 * @date 2022-07-02
 *
 * @copyright Copyright (c) 2022
 *
 */

// #define _DEBUG_

extern "C"
{
    #include "lexer.h"
    int yy_lexer(int *argc, char ***argv);
}

#include "myshell.h"
#include "common.h"
#include "Parser.h"
#include "Console.h"
#include "Display.h"
#include "Executor.h"

#include <exception>


namespace SHELL
{
    int shell_setup(int argc, char *argv[], char *env[])
    {
        // 创建模型
        Console *model = new Console;
```

```
        if (model == nullptr)
        {
            fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno),
"Out of Space for Console model");
            return 1;
        }

        // 创建视图
        Display *view = new Display(model);
        if (view == nullptr)
        {
            fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno),
"Out of Space for Display view");
            return 1;
        }

        // 创建控制
        Executor *controller = new Executor(model, view);
        if (controller == nullptr)
        {
            fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno),
"Out of Space for Executor controller");
            return 1;
        }

        SHELL::shell_loop(model, view, controller, env);

        // 回收内存，MVC模型
        delete model;
        delete view;
        delete controller;

        return 0;
    }

    int shell_loop(Console* model, Display* view, Executor* controller, char
*env[])
    {
        try
        {
            while (1)
            {
                // 显示提示符
                view->render();

                // 从输入读入命令
                char input[BUFFER_SIZE];
                int input_len = view->InputCommand(input, BUFFER_SIZE);

                if (input_len == 0)   // 输入完毕
                    return 0;
                if (input_len < 0)   // 输入异常
                    continue;

                // 从输入中创建buffer
```

```
                YY_BUFFER_STATE bp = yy_scan_string(input);
                if (bp == nullptr)
                {
                    throw "Failed to create yy buffer state.";
                }

                yy_switch_to_buffer(bp);

                // 进行分词解析处理
                int argument_counter = 0;
                char **argument_vector = nullptr;
                yy_lexer(&argument_counter, &argument_vector);

                #ifdef _DEBUG_
                Argument_Display(argument_counter, argument_vector);
                #endif

                model->ResetChildPid();

                /* 输出完成的进程，即使是空指令也应如此。 */
                model->ConsoleJobListDone();

                if (argument_counter == 0)
                    continue;

                bool exit_state = Parser::shell_pipe(model, view, controller,
argument_counter, argument_vector, env);

                // view->show();   // 显示输出信息

                yylex_destroy();    // 释放词法分析器占用的空间，防止内存泄露

                if (exit_state == true)
                    break;
            }
        }
        catch(const char * message)
        {
            fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno),
message);
        }
        catch(const std::exception& e)
        {
            fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno),
e.what());
        }
        catch(...)
        {
            fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s\n", strerror(errno));
        }

        return 0;
    }

}
```

src/Makefile

```
CC = g++
OBJS = common.o Console.o Display.o Executor.o lexer.o myshell.o Parser.o
ProcessManager.o
INC = ../inc
OUT = ../lib
CFLAG = -I$(INC) -O3 -Wall -MMD
DEPS = $(OBJS:.o=.d)
LIB = myshell

lib$(LIB).a: $(OBJS)
    ar -rcs $(OUT)/$@ $^

-include $(DEPS)

%.o: %.cpp
    $(CC) $(CFLAG) -c $<

%.o: %.c
    gcc -I$(INC) -O2 -w -MMD -c $<

lexer.c: lexer.l
    lex --header-file=../inc/lexer.h --outfile=./lexer.c lexer.l

clean:
    -rm $(OBJS) $(DEPS)
    -rm ../inc/lexer.h lexer.c
```

main

main.cpp

```
// 程序：命令行解释器
// 作者：邱日宏 3200105842

/**
 * @file main.cpp
 * @author 邱日宏 (3200105842@zju.edu.cn)
 * @brief 主函数
 * @version 1.0
 * @date 2022-07-17
 *
 * @copyright Copyright (c) 2022
 *
 */

#include "myshell.h"

#include <stdio.h>
```

```cpp
int main(int argc, char *argv[], char **env)
{
    // 开头输出判断程序是否正常开始，仅在调试时使用
    // puts("Welcome to MyShell ! \n");

    if (SHELL::shell_setup(argc, argv, env) != 0)
        puts("shell setup failed.");

    // 末尾输出判断程序是否正常结束，仅在调试时使用
    // puts("Bye~");

    return 0;
}
```

Makefile

```makefile
# 程序：命令行解释器
# 作者：邱日宏 3200105842

CC = g++
INC = ./inc
SRC = ./src
LIB = ./lib
CFLAG = -I$(INC) -L$(LIB) -O3 -Wall -MMD
OBJS = main.o
LIBS = myshell
DEPS = $(OBJS:.o=.d)
RUN = myshell

$(RUN): $(OBJS) lib$(LIBS).a
    $(CC) $(CFLAG) $< -l$(LIBS) -o $@

lib$(LIBS).a:
    make -C $(SRC)

-include $(DEPS)

.cpp.o:
    $(CC) $(CFLAG) -c -o $@ $<

debug: lib$(LIBS).a
    -rm $(OBJS) $(DEPS) $(RUN)
    $(CC) $(CFLAG) -D_DEBUG_ -c -o main.o main.cpp
    $(CC) $(CFLAG) $(OBJS) -l$(LIBS) -o myshell

clean:
    -rm $(OBJS) $(DEPS) $(RUN)

cleanall:
    -rm $(OBJS) $(DEPS) $(RUN) $(LIB)/lib$(LIBS).a
    make clean -C $(SRC)
```