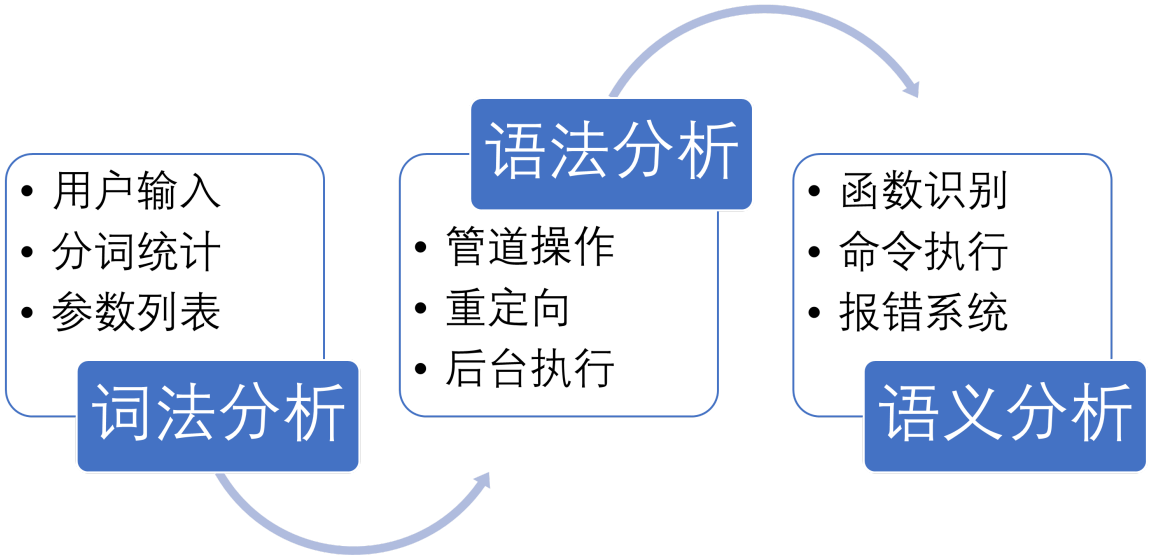
My Shell

引言

命令行解释器作为操作系统中最基本的用户接口，是用户与操作系统底层直接打交道的必经之处。本实验要求通过程序设计语言实现shell 程序的基本功能，设计一个My Shell命令行解释器。

编译器前端构建

为了对用户潜在的各类输入进行解析并抽象出合理的语法结构供解释器执行shell命令，我们需要在解释器执行命令前先将用户的输入进行词法分析和语法分析，最后才能交由解释器进行语义分析和执行。



词法分析

本shell的编译器前端的词法分析器是使用flex进行搭建的。将二十六个英文字符大小写、十个阿拉伯数字以及一些常常会与字符连用的特殊字符作为一个字进行读取，对于在shell中具有特殊含义的管道符（|），重定向符（<, >, >>）等再单独处理，对单行注释符合（#）之后的内容直接予以忽略等，由此写出的lex前端可以较好的实现对用户输入文本的分词和解析，按照规定的语法将输入划分成不同的词组供后续进一步的语法分析。

通过flex编译我们所编写的lex语法可以自动解析出C语言的词法分析器的众多实用函数以及头文件。我们在定义好规则之后直接在需要的地方调用词法分析器即可实现词法分析功能。

此外，对于用户可能输入的一些非法的无法解析的字符（比如中文字符，非法的不可见字符等），词法分析器会在此步骤直接产生较为精确的报错信息以供用户审阅，报错信息较为友善。

与直接在输入中根据空格划分词组相比，实用lex文法编写的词法分析器具有以下优势：

- 允许**特殊含义字符**与其他词组**连写**而中间不加入空格
- 能够适应用户输入命令的**任意空白符分隔**

- 自适应去除**文本注释**，为后续处理带来方便
- 合理的**报错信息**帮助用户定位词法分析错误

当然，由于flex直接生成的词法分析器将由C语言构建，而本shell的主要实现由C++实现。因此，在撰写Makefile时需要专门为其添加相应的规则，并且在使用时需要使用extern "C"的说明。例如：

```
extern "C"
{
    #include "lexer.h"
    /* 其他引用的C语言头文件 */
}
```

语法分析

在完成了用户输入语句的词法分析之后，我们获得了按照语义分隔的参数向量列表。虽然可以通过yacc更进一步的对用户的输入内容进行语法分析，但一方面shell的语法相对较为简单，嵌套语法并不多见；另一方面为了能够更好地在语法分析的过程中对进程调度，重定向等进行设置，我们直接采用C++程序对获取的参数列表进行语法分析。

在我所设计的语法分析器中，语法分析一共分为四个环节进行：**管道符分隔**、**重定向定位**、**挂起操作判别**以及**命令识别**。

语法分析的第一个环节是**管道符**分隔。在获取用户输入的命令列表之后，我们首先在命令列表中根据管道符将一串的命令分割为几个子命令，并且将这些子命令按照管道连接的方式分别顺序执行。每个子命令进入下一步解析，同时前一个子命令执行的输出是下一个子命令的输入，第一个子命令由终端输入，最后一个子命令由终端输出，由此可以完成管道操作。

语法分析的第二个环节是**重定向**定位。对于管道分隔后产生的子命令，语法分析器将再次进行扫描，查找其中出现的重定向符号：<, >, >>, 0<, 1>, 2>, 1>>, 2>>。并对这些重定向符号进行一定的语义判别：比如重定向符号之后一定需要跟一个文件名，如果用户在重定向符号之后没有接一个重定向的文件，或者重定向之后的文件并不满足重定向的要求，那么语法分析器会直接在这一步向用户给出错误信息，帮助用户识别潜在的语义输入的错误。如果用户给出的输入一切合法且正常，那么此时语法分析器会将程序所要执行输入和输出的文件描述符进行重定向，并在命令中去除掉这些重定向符号以免影响接下来环节的解析。

语法分析的第三个环节是**后台执行**判别，即挂起操作。如果经过上述解析之后用户输入命令的最后一个字符仍然为&，那么表明用户希望此命令作为后台命令执行。我们通过分裂子进程的方式完成后台执行的要求，并且将用户输入的&符号去除后交由下一环节继续解析。如果未检测到&符号则作为前台命令直接交给下一环节。

语法分析的最后一个环节是**函数命令识别**。我们需要根据用户输入的指令选择合适的函数执行。由于我们的函数种类非常多，如果顺序解析比较的话需要花费较长的时间才能够完成。为了提高命令解析识别的效率，我们采用了**函数数组**和**二分搜索**的方式进行命令识别。

总所周知，在C语言之中所有的函数均是指针，我们可以通过指针的方式调用函数，也可以将函数指针按照正常指针存储一样存储在其他数据结构里。函数数组就是包含了一组函数指针的数组，在定义了函数指针类型之后的使用与其他变量并不太大差异。例如在我们的程序中如下定义了函数指针数组FunctionArray[]：

```
/** 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];
```

有了函数指针数组之后，我们就可以在数组上进行搜索了。因为用户输入的命令簇为一组字符串，第一个字符串代表了本命令所要执行的函数。而**字符串**是**有序**且可以比较的，因此，我们就在字符串比较的基础上对用户输入的命令进行二分搜索，寻找最合适匹配的命令。如果用户输入的命令在内部命令的函数库中，则直接调用内部命令函数执行，否则就创建一个子进程并调用系统函数执行。

二分搜索是一类高效在有序数组上搜索的技巧，其通过每次将搜索空间减半进行查找可以达到$O(logN)$数量级的搜索时间复杂度。由于二分搜索的实现在许多地方具有高度的复用性，因此我们将其写为模板函数实现。相关伪代码如下：

```
/**
 * @brief 二分搜索查找
 *
 * @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
 */
template<typename T>
int Binary_Search(int left, int right, T val, T array[], int cmp(T a, T b))
{
    while (left < right)
        mid = (left + right) / 2;
        int compare_result = cmp(val, array[mid]);
        if val == array[mid]
            return mid;
        else
            if val > array[mid]
                left = mid + 1;
            else
                right = mid;

    return -1;
}
```

通过二分搜索找到合适的函数，并根据数据下标就可以快速定位到正确的函数并执行了。

解释器后端架构

解释器主要负责具体命令的执行，为用户的错误提供足够的报错信息以及处理程序运行过程中用户可能产生的各类中断信号。由于具体命令执行的部分将在其他章节重点讲述，这里着重只阐述解释器中的报错系统以及中断处理部分。

报错系统

由于用户的使用和输入可能具有很大的不确定性，系统中文件的打开状态，一些命令访问执行的结果也可能存在隐患，因此，我们必须对于用户输入命令的执行情况给予尽可能多详细的反馈来提高系统的稳健性，避免shell轻易奔溃。

在实现上，我们在代码中使用了大量的try……catch……代码块处理各种异常，对于所有系统调用的函数返回值都有进行详细的检查。一旦发生意外，我们就会向用户发出提示并告知错误的原因，帮助用户进行正确的输入和排查错误。对于系统调用发生的错误，我们主要通过errno的记录进行识别；对于内部命令的执行情况，我们定义了如下的错误列表，定位用户的各类错误信息与执行情况。

```
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
};
```

当用户发生某一类型内部命令执行的错误时，系统就会根据报错列表的内容进行提示并给予个性化的判别，为用户输入提供更好的体验。

中断处理

虽然在绝大多数时候，程序只需要根据用户输入的命令顺序执行单进程运行就可以完成绝大多数的任务了，然而，有时候部分内容可能会因为程序运行时间较长，用户并不希望这部分进程继续在前台执行，而需要将某些进程终止或者转移到后台。这时候，用户就会向控制台发送中断信号。

在C语言中，`signal()`函数可以指定如何处理每一个中断信号的函数句柄。在没有设定的情况下，这些信号将以默认的方式进行处理。不过，为了让shell能够处理后台执行的一系列函数以及实现中断等功能，我们需要自己手动来处理这些中断信号，设置相应的信号处理句柄。

目前，我们的shell中可以独立处理的中断信号包括SIGINT，SIGTSTP，SIGCHLD，SIGTTIN和SIGTTOU信号。

SIGINT信号主要由用户按下Ctrl+C引起，在我们的设定中：父进程对于用户输入的Ctrl+C信号不做处理，继续保持循环执行；而执行命令的子进程接收到Ctrl+C信号后按照默认发生中断。

SIGTSTP信号主要由用户按下Ctrl+Z引起。与SIGINT信号类似，对于一直在执行循环的父进程而言，Ctrl+Z信号并不能对其产生实际影响，但却需要将其对应的子进程挂起并在后台暂定执行，同时父进程要在进程管理中记录下被挂起的子进程的相关信息以便查询。

SIGCHLD信号是子进程结束时发送给父进程的信号，标志了该父进程所创建的子进程的结束。SIGTTIN信号和SIGTTOU信号是我们在将后台命令使用`tcsetpcgrp()`函数调至前台时产生的信号，如果不做处理这些信号会将程序中断，因此我们选择将这些信号忽略以保证shell循环能够一直运行下去。

内部命令实现

在最新的版本中，shell 程序支持以下内部命令：bg、cd 、clr、dir、echo 、exec 、exit 、fg 、help、jobs 、pwd 、set 、test 、time 、umask。这些命令的实现介绍如下。

bg

bg命令的格式为：bg

bg命令用于将被挂起的进程转到后台。如果没有参数，则默认将当前进程放到后台运行，相当于在命令后加上&符号。

bg命令处理的主要是由Ctrl+Z控制信号所引起的在后台暂定运行的程序。在接收到bg命令后，进程管理器会查询相应的进程单元，获取用户需要在后台运行的进程pid，并且向该进程发送SIGCONT信号让其在后台继续运行。同时，将进程管理器中记录下的该进程的运行状态从Stopped改为Running。

cd

cd命令的格式为：cd

该命令可以把当前默认目录改变为。如果没有参数，则显示主目录。如该目录不存在，会出现相应的错误信息。

这个命令主要是通过`chdir()`函数来实现的。该函数可以将当前进程的工作目录切换到输入参数对应目录下，如果对应目录存在且能够正常打开，则会返回0，否则会返回相应的异常状态。我们通过该命令可以切换进程的工作目录，并且根据该函数的返回信息判断相应的错误情况给予用户以提示。

同时，这个命令也可以改变PWD环境变量，通过`setenv()`函数设置PWD的环境变量为切换后的工作目录，让环境变量与我们的shell中的工作目录保持逻辑上的一致性。

当然，其中有一项内容需要特殊处理，那就是用户输入中的**“~”符号**。因为“~”符号代表了用户的主目录，如果用户输入的目录以“~”开头的话，我们需要在更改目录前先将用户的输入的“~”字符转变为用户主目录的路径，然后再进行接下来的操作。

clr

clr 是一个清屏命令。clr命令等价与普通shell中的clear命令，其作用是将显示屏幕上的内容全部清除干净。

依托丰富的不可见控制字符，Linux的输出控制变得十分简单而有趣了。通过向屏幕或者是输出文件上输出一些不可见的控制字符，我们可以很容易地改变屏幕或是这些输出文件里的输出的位置、输出字符的颜色、当前屏幕或文件的状态等等。而在清屏功能上亦是如此。

看上去将整个屏幕清除似乎不是十分容易，但其实通过向输出文件中打印**"\x1b[H\x1b[2J"**字符之后，所有原来文件中的内容就自动被清楚了，而在屏幕之上表现的效果便是清屏。

dir

dir命令的命令格式为：dir

该命令的功能是列出目录的内容。如果没有参数，则显示当前目录内容。在实现上，dir命令与cd命令有许多异曲同工之处，二者都是对目录进行操作。

与cd命令类似，dir命令之中也需要对用户可能查询命令中的“~”字符进行特殊的替换处理。将“~”字符替换为用户的主目录后可以保证程序以正确的方式继续运行。

echo

echo命令的格式为：echo

echo命令用于在屏幕上显示并换行，其中多个空格和制表符被缩减为一个空格。

由于在前期flex解析的时候已经将用户所有的输入根据空白符进行划分进行了解析，因此在实现echo命令时，我们只需要将echo命令之后的参数逐一输出，中间以一个空格分开，在所有内容输出完毕之后再输出一个换行符即可。

exec

exec命令的命令格式为：exec

exec命令用于执行命令，并且将该命令对应的代码覆盖当前运行这个命令的进程的代码，以执行新的命令替换 当前的shell进程。

在C语言的库中包含有exec一系列的函数可以帮助我们在C程序中实现执行exec命令。我们可以使用运行指令向量的`execvp()`函数实现调用exec命令。

需要注意的是，在我们使用execvp函数时，此时我们解析语法中的命令第一个命令为exec，因此我们向函数传参时要去掉第一个命令，从第二个参数开始传给execvp。此外，如果用户输入中只有一个exec命令而没有其他参数，那么shell应该不做任何处理，相当于什么命令都没有执行。

exit

exit命令可以退出当前shell。

在My Shell命令行解释器中，所有执行的命令最后都会带回一个返回状态。其中设置的一个状态就是SH_EXIT状态用于判断是否执行的是退出命令。当用户输入exit命令后，该命令正常执行完会返回SH_EXIT状态，主程序循环的shell检测到上一条命令执行完成时的状态为退出状态后便会自动结束shell，退出程序。

fg

fg命令的命令格式为：fg

fg命令用于将后台运行或挂起的作业切换到前台运行。如果该作业原本在后台运行，则其运行将变为前台；如果该作业原本在后台暂定了，则会先启动该进程再放至前台运行。

从后台转为前台主要使用的命令为库函数中的 `tcsetpgrp()` 函数。该函数可以设置某一进程为前台进程。同时，执行fg命令的进程也需要向该进程发送SIGCONT命令使其继续运行，并且记录下改变后子进程的相应状态，自身转变为后台进程运行。

help

help 命令可以显示用户手册。

在程序所在的doc目录下存放了供用户阅读的用户手册，当用户输入help命令后，程序会读取doc目录下的用户手册并将其显示在屏幕上。使用more 命令可以过滤用户手册的信息。

同时，使用 `doxygen` 工具可以帮助我们根据程序注释生成相应的man指南，该指南也能够用于为用户提供帮助。

jobs

jobs ——显示所有挂起的和后台进程的作业号及状态。如果没有列表，则显示当前进程的状态。

在用户通过&操作符或者Ctrl Z控制信号引起一个子进程被挂起到后台时，我们就在父进程中记录下被挂起的子进程的相关信息，包括其所在的工作号（job id），进程号（process id），执行状态（如Running，Stopped，Done，Terminated），当前执行的命令等。所有的这些信息记录在一棵红黑树之中。当用户调用job命令查找时，shell可以快速打印出所有进程的信息或是查找到某一进程的状态信息。

pwd

pwd 命令的功能是显示当前目录。

pwd命令的实现非常简单，只要将控制台存储的当前工作目录输出到显示处，让负责处理输出的显示终端根据需要在对应的输出文件上打出当前工作目录的输出即可。

set

set命令用于列出系统中所有的环境变量。

在C语言程序运行时，系统会自动将输入给C语言程序的参数格式argc，参数列表argv，以及对应的所有环境变量列表env传递传递给C语言程序的main函数。因此，一个标准的main函数可以写成如下的形式：

```c
int main(int argc, char argv[], char **env)
{
    /*some main part*/
    return 0;
}
```

在这个程序里，我们可以直接通过传入main函数的env变量来获取当前程序所处在系统中的所有环境变量，并通过一个while循环将所有环境变量逐一打印便可列出系统中所有的环境变量了。

当然，有时候随着我们程序的运行，我们可能会在环境中设置新的环境变量。而这时原本通过main函数传入的env变量可能还没有来得及进行更新，我们可以通过引用C库中定义的全局变量environ来更新env变量所指向的环境变量的值，确保能够读取到最新的环境变量。

test

test命令的命令格式为：test

test命令可以检测用户输入的表达式返回的结果是true还是false。为了使命令测试的结果更加可视化，如果test命令返回的结果是正确的且输出在终端中，那么终端上将显示“true”；如果test命令返回的结果是错误的且输出在终端中，那么终端上将显示“false”。

test命令支持丰富的命令格式，根据用户输入的参数不同，test命令可以完成文件测试、整数测试或者是字符串测试。目前test命令中支持的各类测试参数及其含义如下。

1. 文件测试

类型	表达式	含义
文件存在判断	-e file	判断文件是否存在
文件类型判断	-f file	是否为普通文件
	-d file	是否为目录文件
	-c file	是否为字符特殊文件
	-b file	是否为块特殊文件
	-p file	是否为管道文件
	-L file	是否为符号链接文件
	-S file	是否为套接字文件
文件权限判断	-r file	是否具有可读属性
	-w file	是否具有可写属性
	-x file	是否具有可执行属性
	-O file	是否为所有者文件
	-G file	是否为组文件
文件属性判断	-u file	是否具有用户位属性
	-g file	是否具有组位属性
	-k file	是否具有粘滞键属性
	-s file	文件长度是否非0
	-t file	文件描述符是否联系终端
文件时间比较	file1 -nt file2	file1是否比file2新
	file1 -ot file2	file1是否比file2旧
	file1 -ef file2	file1与file2是否为同一文件

2. 整数测试

类型	表达式	含义
数字大小比较	number1 -eq number2 number1 == number2	number1是否与number2相等
	number1 -ne number2 number1 != number2	number1是否与number2不相等
	number1 -ge number2 number1 >= number2	number1是否大于等于number2
	number1 -gt number2 number1 > number2	number1是否大于number2
	number1 -le number2 number1 <= number2	number1是否小于等于number2
	number1 -lt number2 number1 < number2	number1是否小于number2

3. 字符串测试

类型	表达式	含义
字符长度判断	-n string -z string	字符串长度是否非0 字符串长度是否为0
数字大小比较	string1 -eq string2 string1 -ne string2 string1 -ge string2 string1 -gt string2 string1 -le string2 string1 -lt string2	string1是否与string2相等 string1是否与string2不相等 string1是否大于等于string2 string1是否大于string2 string1是否小于等于string2 string1是否小于string2

文件测试实现主要通过程序的系统调用。在C语言的系统调用库中，也提供了与文件状态信息查询相关的丰富的库函数。在sys/stat库内，有`fstat()`，`stat()`，`lstat()`函数可以提供文件信息。其中，fstat返回与打开的文件描述符相关的文件的状态信息；stat返回通过文件名查询到的文件的状态信息；lstat返回的也是通过文件名查询到的状态信息，但是当文件是符号链接时，lstat返回的时符号链接本身的信息，而stat返回的是该链接指向的文件的信息。

在test文件测试的判断当中，我们主要使用到的是`lstat()`函数对文件状态进行判断。因为在我们的判断里函数需要同时能够处理符号链接相关的内容。`lstat()`函数能够返回一个`struct stat`的结构体，并将文件所有的相关内容都存储在该结构体之中。通过访问`lstat()`函数返回的文件结构信息，test便可以对文件测试进行判断。

而整数测试和字符串测试便十分简单。由于C语言中已经内置了整数类型和字符串类型，我们将用户的输入转化为相应类型的变量数值，通过程序自身的类型序关系比较以及字符串库中所带的`strcmp()`等函数便可以获得关于整数和字符串类型的判别结果。

time

time 命令用于显示当前时间。需要注意的是，这里的time命令并不是指Linux当中用于计时的time命令，而是date命令——显示当前的日期与时间等相关信息。

在C语言gcc编译器提供的glibc等许多库里包含丰富的系统调用函数，其中在time.h的头文件里定义了time_t的时间获取指针，tm的结构体以及相应的函数能够帮助我们快速系统时间。

与此同时，time库里面还包含了`strftime()`的函数能够形式化的打印时间结构体中的各种参数，以任何我们所期望的形式。比如通过%c的控制符就可以打印出符合当前区域设置的首选日期和时间表现形式的日期格式。我们将打印出的信息传给显示器进行显示即可。关于strftime函数的具体格式说明可以参考下面这张图片：

umask

umask ——设定新创建文件或目录的访问特权。如果没有参数，则显示当前设置的掩码。

是位的掩码，通常用八进制表示。掩码位为1表示新创建的文件相应的访问特权应该被关闭。在输入时，用户可以以任意进制的方式输入，只是需要在以除了十进制之外的进制输入时需要加上对应进制表示的前缀。例如在输入八进制数前应该加上字符‘0’，在输入十六进制数前应该加上字符‘0x’。

在Linux操作系统中C语言底层的sys库内，包含一个`umask()`函数能够帮助我们对系统中的掩码进行设置。`umask()`函数会将系统umask值设成参数mask&0777后的值, 然后将先前的umask值返回。当然，这里数字一般都是用八进制数表示的，因此，我们对于用户输入的内容也需要进行判断，如果用户以八进制或者十六进制输入数据的话我们需要对输入的内容进行转换之后再存储。

在使用open()建立新文件时, 该参数mode 并非真正建立文件的权限, 而是 (mode&~umask)的权限值。设置完掩码之后用户便可以利用该掩码创建新的符合文件权限需求的文件了。

环境变量设置

shell 运行时的环境变量将包含shell环境变量，其格式如下所示，其中/myshell 是可执行程序shell 的完整路径。

```
shell=<pathname>/myshell
```

当shell中遇到了上述提到的内部命令之外的**其他的命令行输入**时，这些命令会被解释为**程序调用**，shell创建并执行这个程序，并作为自己的子进程。在子程序的执行的环境里，系统环境变量将包含parent环境变量，其格式如下所示。

```
parent=<pathname>/myshell。
```

在C语言中，设置环境变量并不是一件十分复杂的事情。在stdlib库里有`setenv()`的函数可以帮助我们设置环境变量。在程序运行开始时我们设置shell环境变量，在每次创建子进程时在子进程中设置parent环境变量即可实现上述功能。

批处理文件执行

除了直接从终端的命令行中读入用户一条条命令的输入之外，shell也支持从文件中提取命令行输入。在shell中输入myshell加上所要执行的批处理文件名便可以从这些文件中读入命令行输入。

例如，用户可以在shell中使用以下命令行调用批处理：

```
$ myshell batchfile
```

在这个命令中，batchfile是我们要执行的一个批处理文件，这个批处理文件可以包含一组命令集。调用此命令后shell会依次执行该命令集中的每条命令，当到达文件结尾时shell 退出返回到直接调用myshell时的状态。

如果shell 被调用时没有使用参数，则会在屏幕上显示提示符'> '请求用户输入批处理文件。

shell支持**同时处理多个批处理文件**，这些批处理文件会按照输入的顺序逐一解析执行。

批处理文件的执行主要涉及到文件操作以及输入的重定向。在解析到用户输入了myshell命令之后，执行器首先要判断之后是否有输入参数。如果没有输入参数的话需要在终端显示提示符并不断读取用户输入，直到用户提供了正确的输入。此时我们还需要对用户的正确输入进行解析以确保能够获取正确的文件名。

在用户输入解析完成后，shell对逐一打开用户输入的各组文件，并将这些**文件**作为**输入的重定向**传入给另一个shell循环之中，对从这些文件读入的内容按照正常终端读入的方式读取和执行。在所有文件执行结束后需要将文件关闭。

I/O 重定向

为了适应用户在不同环境中输入与输出的需求，shell 支持stdin 和stdout 的I/O 重定向。

例如命令行为：
programname arg1 arg2 < inputfile > outputfile
使用arg1 和arg2 执行程序programname，输入文件流被替换为inputfile，输出文件流被替换为outputfile。
stdout 重定向应该支持以下内部命令：dir、environ、echo、help。
使用输出重定向时，如果重定向字符是>，则创建输出文件，如果存在则覆盖之；如果重定向字符为>>，也会创建输出文件，如果存在则添加到文件尾。

同时，输入输出重定向也支持加入重定向的文件描述符的标识，例如，标准输入重定向可以写为 0< ；标准输出重定向可以写为 1> 等。

在Linux操作系统中，所有的东西都是以文件的形式组织的，包括标准输入、标准输出、标准错误输出等也其实都是一个个文件。通常，标准输入指向的是用户键盘的输入，用户键盘的输入都已文件内容的形式在Linux程序中被共享；标准输出和标准错误输出练习的是显示屏幕，Linux程序都过向这些文件中输出就可以最终在用户的显示屏上显示相应的内容。

因此，要改变程序执行结果的输入和输出，实际上也就是改变这些输入和输出时所对应的文件描述符。当一个程序输入或者是输出的文件描述符被替换为其他文件的文件描述符时，该程序执行所需要的输入就会从输入文件描述符所对应的文件中获取，而该程序执行结果的输出也会打印到输出文件描述符所对应的文件之中。

为了实现重定向功能，一方面在程序shell中所有的输入与输出都是采用**read函数**或者**write函数**来实现输入输出的。而shell中的输入输出控制由显示器类所控制，所有的输入输出都是使用控制台中记录下的输入文件描述符、输出文件描述符、错误输出文件描述符所控制。当发生重定向时，这些文件描述符就会发生改变，变为需要重定向的文件的文件描述符，从而实现重定向的功能。

另一方面，在解析到命令中含有重定向的指令之后，我们也通过调用`dup()`**系列函数**来迁移重定向。因为我们自身所实现的显示器类只能够在程序依然由我们自身的shell所控制时实现输入输出的重定向，但是当程序需要使用`exec()`类函数簇发生系统调用时，程序的输入输出将不再由显示器类所控制，所以我们必须将程序本身的输入输出重定向在系统层面改变，这样才能确保即使程序将执行权交给了`exec()`类函数之后用户所定义的重定向依然能够继续进行。

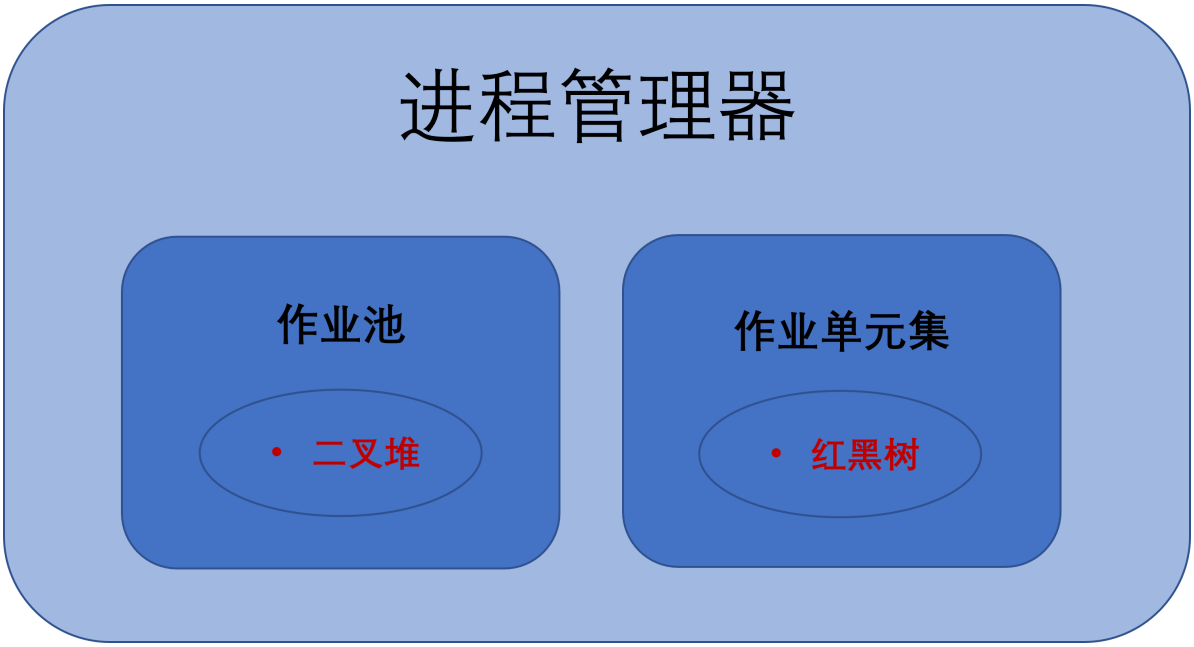
后台程序执行

shell 能够支持后台程序执行。如果在一条命令行的末尾添加了&字符，那么该命令将在后台运行，而前台可以继续执行其他程序。当后台程序运行完成之后会将结果显示在前台并给予任务完成的信息提示。

在我们的shell中，后台运行的程序具有四种不同的状态。它们分别是正在运行（Running）、停止运行（Stopped）、完成运行（Done）和终止运行（Terminated）。在显示程序运行状态时会对这些内容给予显示。对于这些状态的定义我们是采用一个枚举类型来实现的，这样可以保证进程状态具有高度的可扩展性和可迁移性。

```
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
};
```

后台程序执行主要由控制台类中的**进程管理器**类实现。进程管理器包含两个主要部分——**作业池**和**作业单元集**。其中作业池负责在用户请求新的后台作业时向该进程分配一个当前可用作业号中最小的一个作业号；作业单元集负责存储和管理当前所有正在后台运行的进程。



作业池

作业池的作用为在用户请求新的后台作业时向该进程**分配**一个当前可用作业号中最小的一个作业号，同时在一个进程结束后**回收**为该进程所分配的作业号。这是一个需要不断插入和删除的有序的数据结构，但我们每次的查询只需要取其中最小的一个数使用即可，因此，使用**堆**来实现该数据结构是较为合适的。

堆能够近似维护一组数据的序关系同时降低仅仅查询当前序关系中最小数的时间成本。通常，一个堆的数据结构具有如下的形式。为了较好的实现各类数据结构，我们采用了**模板类**的方式定义此堆，并使用**抽象类**的架构实现以便于随时可以替换堆的实现方式，实现更高性能的突破。

```
template <class T>
class Heap
{
    public:
        Heap() : size_(0) {};
        virtual ~Heap() = 0;

        virtual void build(T data[], size_t size);
        virtual void insert(T value);
        virtual T top() const;
        virtual T extract();
        size_t size() const { return size_; }

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

通常来说，在一个系统上同时在后台执行的作业并不会太多，我们需要处理的并不是超大规模级的数据。由于计算机系统中对于连续存储内存空间中数据读取可以产生较高的命中率与较低的失配率，在较小规模的数据下，由于二叉堆具有**连续存储**的内存空间，因此**二叉堆**在数据规模不太大时对于插入和删除操作的性能是远远优于其他使用包含指针实现的各类堆结构的。所以，我们在作业池中的堆采用二叉堆来实现。

二叉堆是一类完全二元树或者是近似完全二元树，其满足堆特性，即父节点的键值总是保持固定的序关系于任何一个子节点的键值，且每个节点的左子树和右子树都是一个二叉堆。二叉堆具有$O(logN)$的插入效率以及$O(logN)$的删除效率，同时将N个数同时插入建立成一个二叉堆也只需要$O(N)$的时间复杂度，是能够较好的实现shell中作业池分配任务的功能的。

作业单元集

作业单元集是一类用于存储当前所有后台运行进程状态的数据集。每个被挂起的作业进程是一个作业进程单元。在作业进程单元中，我们需要存储作业的id，进程号pid，当前运行状态以及其执行的命令。其中，作业**id**是作为每一个被挂起的进程的**唯一标识**，可用于识别一个进程，其有序的正整数的序关系也使得其能够进行排序比较。因此，作业单元集可以以id作为比较的键值并重载运算符来实现一类基于比较的搜索数据结构以满足有时需要**查询**，**插入**或**删除**特定作业单元的需求。

搜索树最为一种灵活的数据结构能够很好地支持动态插入新的数据，查找在当前条件下某一数据以及删除指定数据的需求。通常，我们会使用二叉搜索树(Binary Search Tree)来实现这一功能，理想情况下，使用二叉搜索树插入，查找和删除的效率都是$O(logN)$的。但由于二叉搜索树在一定的条件下可能会非常的不平衡，在极端情况下，二叉搜索树甚至可能退化成一个链表，这时我们的插入，查找和删除的时间复杂度将会退化到$O(N)$，这是一个非常坏的情况。

二叉搜索树出现由平均$O(logN)$到最坏 $O(N)$时间复杂度的转变是二叉搜索树中可能出现的不平衡所导致的。由于在插入有序数据进行二叉搜索树的时候二叉搜索树会形成一个链表，极端倾斜不平衡的树使得我们的查找最坏的时间复杂度大大提升了。

为了解决这一问题，我们需要采用一种可以在插入和删除过程中维护树的动态平衡的数据结构。红黑树作为一种优秀的平衡搜索树，能够很好地满足这一要求。

红黑树是一种特殊的二叉搜索树，其必须满足下面**五条性质**：

1. 每一个节点要么是红色，要么是黑色
2. 根节点的颜色为黑色
3. 每一个叶子节点（NIL）是黑色的
4. 如果一个节点是红色的，那么这个节点的孩子都是黑色的
5. 对于任一节点，从该节点到该节点后代的叶子节点的所有简单路径都含有相同数量的黑色节点

红黑树的这5条性质可以保证红黑树在很大程度上是一个十分平衡的树。使用红黑树进行插入、查询、删除等操作都可以在最坏$O(logN)$的时间复杂度内完成。利用这个特性，我们就能够实现在最坏$O(logN)$的时间复杂度内实现对作业单元的插入，查询或者删除操作了。

管道操作

shell命令行支持管道（“|”）操作，可以将上一条命令执行的结果作为下一条命令的输入传递。多个进程之间可以互相进行管道通信，两个需要共享的命令之间使用管道符“|”分隔。

管道通信的实现是以文件系统为基础的。当一个写进程向管道的共享文件中输入数据时，接收管道输出的读进程就能够通过该共享文件从管道中接收到从写进程发送来的数据。

管道操作的实现在Linux基于文件系统的架构上十分轻松，在C程序库中，通过使用`pipe()`函数就能够打开一个**管道通信文件**，让两个文件描述符可以指向一个共同的管道文件并从中输出或者输入。配合上`fork()`函数**分裂进程**产生子进程之后，我们再结合**重定向**的技术，让子进程执行命令的输出重定向至管道文件；同时父进程的输入也重定向至该管道文件。这样，我们就可以非常容易的实现一个管道操作了。

路径显示

当用户在终端进入shell且用户命令的输出需要在终端显示时，命令行提示符将会显示当前使用用户的用户名称，用户所使用的机器名称以及用户当前所处在的工作目录的路径。其显示格式如下：

```
<username>@<hostname>:<current working dictionary>
```

由于用户绝大多数的操作都是在用户自身的主目录下完成的，而许多用户的主目录往往又不是十分精简。因此，在shell终端工作目录的显示中如果路径以用户的主目录开头的话，那么则用户的主目录将被~符号替代显示。

以上这些所需的相关信息基本上都已经在**环境变量**中列出了，我们在初始化控制台的时候通过调用函数从环境变量中将这些内容读取便可以得到这些信息。

环境变量	相关函数
用户名称	getpwuid()
主机名称	gethostname()
工作目录	getcwd()

为了让用户终端的显示更加丰富多彩，shell命令行的输出被赋予了丰富的颜色。shell终端的显示控制与各类不可见字符有关，通过向终端中输出一些不可见字符可以改变终端中输出内容的颜色。这里列出了在终端中常见颜色的ASCII码：30 (黑色), 31 (红色), 32 (绿色), 33 (黄色), 34 (蓝色), 35 (洋红), 36 (青色), 37 (白色)。通过调节终端的颜色输出，我们就可以获得更好的显示效果。

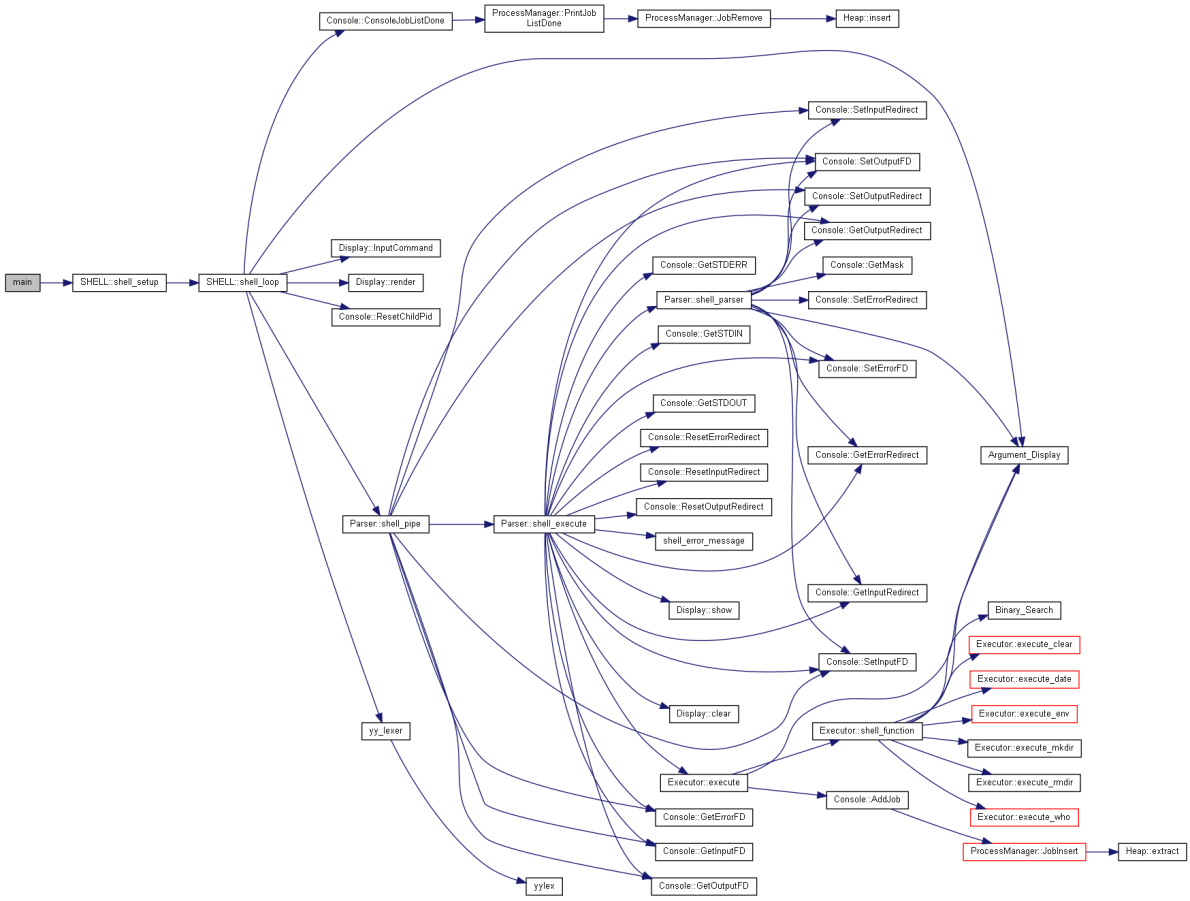
用户主目录的替换则是使用一次字符串扫描与匹配则能够完成。在每次显示目录时同时从开头开始线性扫描当前工作目录的字符串以及用户主目录的字符串，如果完全匹配的话则将用户的主目录删去改为~符号，否则就按照原目录格式显示。

完成上述内容的替换之后，我们就得到了一个较为好看的路径显示样式了。下图是一个命令行提示符的样例，展现了我们命令行提示符的效果。

总结

本项目采用了面向对象程序设计的思想，综合运用封装、继承、多态等技术，构建了从词法分析器、语法分析器到控制台、显示器、解释器等多种类，并且在许多类之下又有包含许多子类，代码具有较高的可复用性和可扩展性。同时，本工程项目合理的文件层次和结构既较好地做到了技术的封装和隐藏，又对调用程序的用户提供了友好的函数接口，方便用户根据头文件和我们编译所生成的静态链接库进行函数调用。

为更好地理清本项目中各个类之间的逻辑关系，下图展示了本项目中部分类之间的函数调用关系，其中红色框部分内容为不要求在本项目中实现的内部命令功能，只有在DEBUG模式下才会被调用。



附录

shell内部指令列表

1. bg ——将被挂起的作业转到后台。如果没有参数，则默认将当前进程放到后台运行。
2. cd ——把当前默认目录改变为。如果没有参数，则显示主目录。
3. clr ——清屏。
4. dir ——列出目录的内容。如果没有参数，则显示当前目录内容。
5. echo ——在屏幕上显示并换行，多个空格和制表符将被缩减为一个空格。
6. exec ——执行命令替换当前运行这个命令的进程。
7. exit ——退出shell。
8. fg ——将后台运行或挂起的作业切换到前台运行。如果没有参数，则默认将当前进程放到后台运行。
9. help ——显示用户手册，即本文档。
10. jobs ——显示所有挂起的和后台进程的作业号及状态。如果没有列表，则显示当前进程的状态。
11. pwd ——显示当前目录。
12. set ——列出所有的环境变量。
13. test ——检测表达式返回true还是false。
14. time ——显示当前时间。
15. umask ——设定新创建文件或目录的访问特权。如果没有参数，则显示当前设置的掩码。