# Lab: Xv6 and Unix utilities

Github仓库链接：[OS-xv6-labs/lab1 at main · Randomly-git/OS-xv6-labs](https://github.com/Randomly-git/OS-xv6-labs/tree/main/lab1)

## 1.Boot xv6 ([easy](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1. 实验目的

成功编译并运行xv6内核。

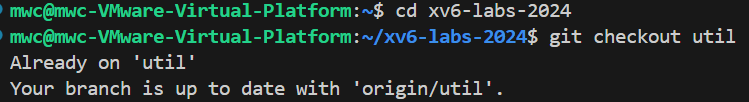
1. 实验步骤

·获取实验室的xv6源的git存储库并切换到本实验的分支（util）

git clone git://g.csail.mit.edu/xv6-labs-2024

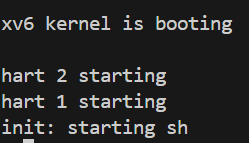
cd xv6-labs-2024

git checkout util

成功进入了该实验分支：  


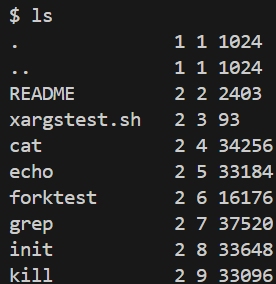
·构建并运行xv6（make qemu）

运行指令，可以看到xv6的启动信息如下，并进入shell模式：



·在xv6 shell中输入命令，比如ls（查看xv6初始文件系统列表）

若想退出shell 模式，输入ctrl-a再按下x即可。

**分析**：xv6成功启动，可以开始后续实验了。  


1. 实验中遇到的问题和解决办法

第一个实验主要是在环境配置的基础上做的。2024年版实验环境搭建本身不难（tools中只需执行一条指令，然后测试即可），但唯一的要求是要求ubuntu24以上的版本。之前没有注意这一点，导致qemu始终运行出错（版本太低），重新下载最新的ubuntu24.02之后问题得到了解决。

1. 实验心得

实验环境的搭建要注意版本适配问题，尤其是比较新的实验环境需要注意虚拟机自身提供的库可能无法兼容。

## 2.sleep ([easy](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1）实验目的

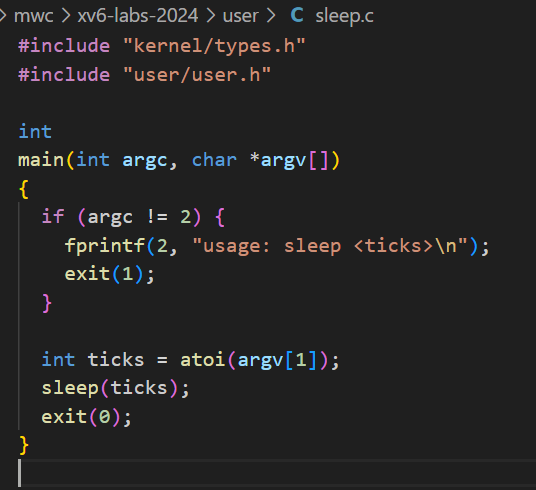
创建并调用user/sleep.c文件，程序接受一个ticks参数（休眠时钟数），

实现用户态的sleep命令。若用户没有传参数则报错。

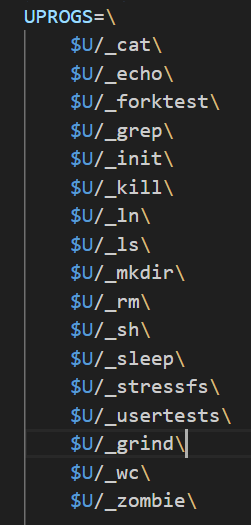
1. 实验步骤

·创建并编写user/sleep.c文件，并注意错误处理  
首先进入分支的user文件夹下用touch指令创建sleep.c，并打开

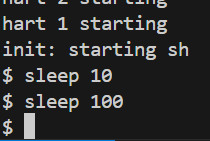
编写代码如下（全部代码）：



使用字符串的动态数组方式传输入内容，参数argc进行错误处理

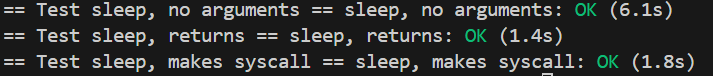
·把sleep.c程序添加到Makefile的UPROGS中  
返回根目录打开Makefile文件，找到UPROGS进行添加：  


·运行make qemu并在sv6 shell中测试



**分析：**发现输入sleep 10之后系统会过一段时间才给出反应，而且输入的参数越大时间越长，说明我们的程序起作用了

·运行make grade验证程序是否通过：

输入指令make GRADEFLAGS=sleep grade，看到：  


测试通过（包括无输入的错误处理），实验结束

1. 实验中遇到的问题和解决办法

本次实验主要难点是如何将自己编译的程序加入构建系统。具体方法已经在实验步骤中给出，即按照Makefile文件的格式去添加文件\_sleep，要严格按照xv6的书写规范来做。

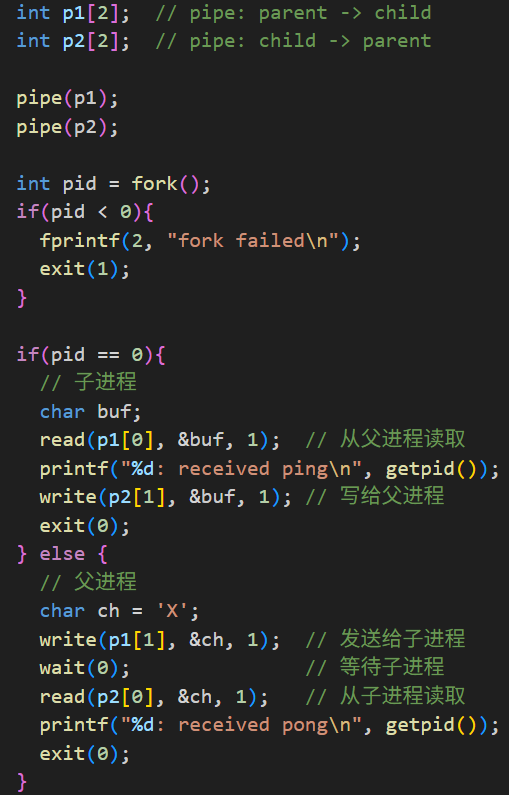
1. 实验心得  
   本实验在学到如何在该环境下加入自己的程序之外，代码编写上用到了一些基本操作：atoi()把字符串转换成整数，sleep()系统调用，用exit()结束用户程序；以及如何测试自己的程序。

## 3.pingpong ([easy](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1）实验目的

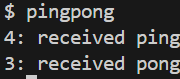
通过自己编写一个程序，使用pipe方式实现父子进程之间的反复通信：将一个字节流像打乒乓球一样在父子进程之间传来传去（要看得见对方的收到回复），加深对于理论课上关于进程间通信方式的理解。

1. 实验步骤

·创建文件user/pingpong.c，写入核心代码并编辑Makefile。过程与Sleep实验相近，这里展示核心代码（LINE：9）：  


分别定义了两个管道：都是单工的。子进程只能拿读p1并写p2，父进程反之。用pid来标识两个进程。

·进入shell模式测试程序，看到如下输出：



**分析：**这里的4和3分别是系统给子进程与父进程提供的pid，分别表示子进程收到父进程的发送，以及父进程收到回复。pid为什么是3、4将在最后一部分讲述。

·运行make GRADEFLAGS=pingpong grade验证：  


测试通过

1. 实验中遇到的问题和解决办法

编写程序中最重要的问题在于对Pipe方式的理解：它是单向通信的，建立之后只允许一个进程在管道不空的时候从中接收消息，拎一个进程在管道不满的时候发送消息。编程时通过定义两个管道和规定两个进程的行为来实现。

1. 实验心得

本实验加深了对管道通信方式的理解：单向传输比特流，每个通道都规定一个读端和一个写端。

关于控制台中输出的pid为何是3和4的问题，理论课中的内容能够很好地回答：在linux系统中0代表调度进程/闲逛，1为用户进程的始祖进程，2为内核进程的始祖进程，它们都是保留的。

## 4.primes ([moderate](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))/([hard](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1）实验目的

通过实现一个基于管道的并发素数筛选算法（prime sieve），理解Unix用管道连接小程序以完成复杂任务的思想，熟悉使用read()/write()操作二进制数据而非文本格式，理解进程创建、退出、资源清理（特别是文件描述符关闭）的过程和并发编程中资源限制管理的重要性。

1. 实验步骤

·创建user/primes.c文件并添加到UPROGS列表（过程不再赘述）

·prime sieve的并发结构分析

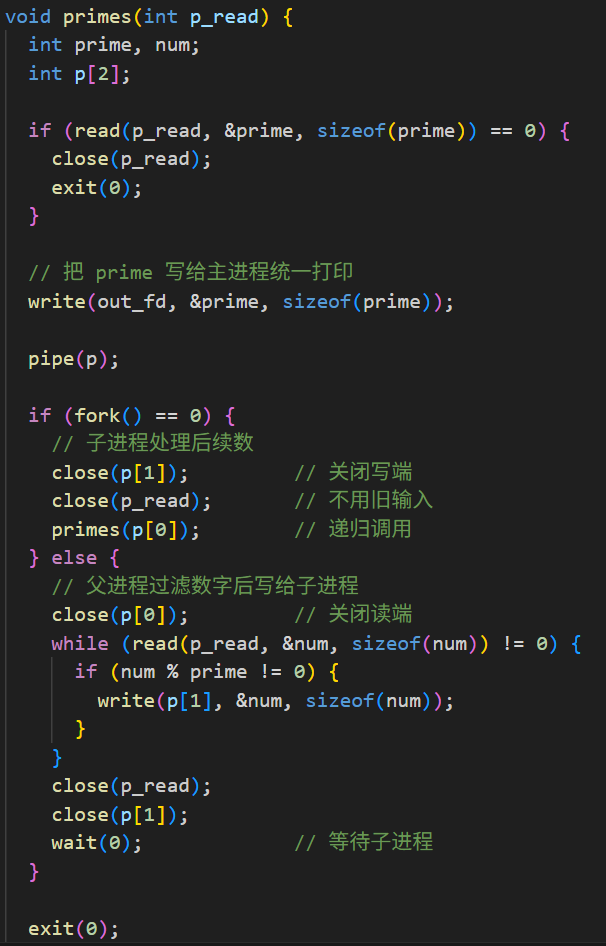
进程管理通过这样一个逻辑链来实现：  
main ──> pipe ──> primes(2) ──> pipe ──> primes(3) ──> pipe ──> ...

每一个prime(n)进程都读取来自上游管道的数字流，打印它遇到的第一个数字n（即第一个素数），忽略所有能被n整除的数，把剩下的数字写入下一个管道并fork处下一个primes进程。

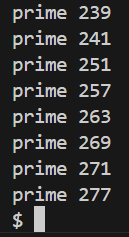
·通过递归函数来实现prime sieve算法：

在users/primes.c中编写一个递归函数primes()，实现这样一个递归过程，形参是为父进程提供数据的管道p\_read。

每个父进程都打印自己获取的第一个数prime，同时创建一个新的管道和自己的子进程。父进程负责向管道中写数据，而子进程负责读数据，因此它们在运行时分别需要关闭读端和写端，避免资源泄露或关都不结束的情况。父进程把能被prime整除的截住，剩下的发给子进程；子进程再递归调用primes()，直到一个进程收不到任何数字，递归结束。递归链条中的每个进程都需要等到自己子进程结束之后才能结束。具体的代码实现如下图所示（LINE：8）：



·程序测试（shell中和make grade）与分析：





**分析：**程序有序地输出了范围之内的所有素数，这一过程在递归进入的过程中就进行了，通过递归的方式保证了输出逻辑上的有序（至于实际输出的有序，在下一部分会提到如何实现）。

1. 实验中遇到的问题和解决办法

·运行过程中首先遇到了来自编译器的发难，编译器怀疑我的递归函数有无限递归的可能，因此给出了一个warning，这在vscode的严格编译要求下会拒绝执行。于是我声明了：



在这之后编译器允许我的程序继续运行

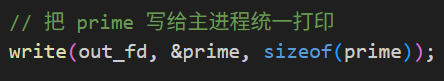
·程序在运行过程中反复出现了输出乱码的情况：

prime 37 )+/5;=CGIprime 183 OSYaegkmq���prime 203 prime 271 pri�me �2�79� $ ���������

一开始我怀疑是父进程的终止逻辑不对，于是我改成需等待所有子孙进程停止才可以停止，但没有解决问题。其实原先的逻辑是正确的：递归地终止进程，子进程结束时所有子孙进程都结束了。

后来我分析输出的乱码才发现了问题所在。在乱码中，乱码内容和正常内容相互交叠，这很符合输出竞争的特点。由于进程的并发性，可能多个进程在同一时间都进行printf系统调用，输出相互干扰而形成乱码。这种情况再一开始进程数较少的时候不会出现，因此可以正常打印前面的一些部分。

于是我将分配给各个进程的打印过程整合到一个统一的输出进程中，这样最终解决了输出乱码的问题。



1. 实验心得

编写代码的基础是对prime sieve算法做深入了解，在此基础上进行资源的调配。在多进程通信中，管道关闭的时间是否重要，否则会出现信息泄露或者保持敞开的情况。

并发编程的重点是资源的合理调配，尤其应该注意I/O资源的特殊性，像这个实验中的输出调用就必须由一个进程来统一完成，否则会出现乱码。本次实验使我对并发编程的复杂性与对资源管理的高要求有了初步的认识，为后续实验打下良好基础。

## 5.find ([moderate](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1）实验目的

通过实现一个简化版的UNIX find程序——在指定目录树中查找所有名字为指定目标的文件（递归搜索子目录）——加深对于UNIX文件系统管理方式的理解。

1. 实验步骤

·功能需求分析

程序输入两个参数：find [path] [target\_name]

其中path为搜索的起始路径，如（.）

target\_name为需要查找的文件名

输出为所有匹配的问价路径，格式如：./a/b/file.txt

·程序设计思路

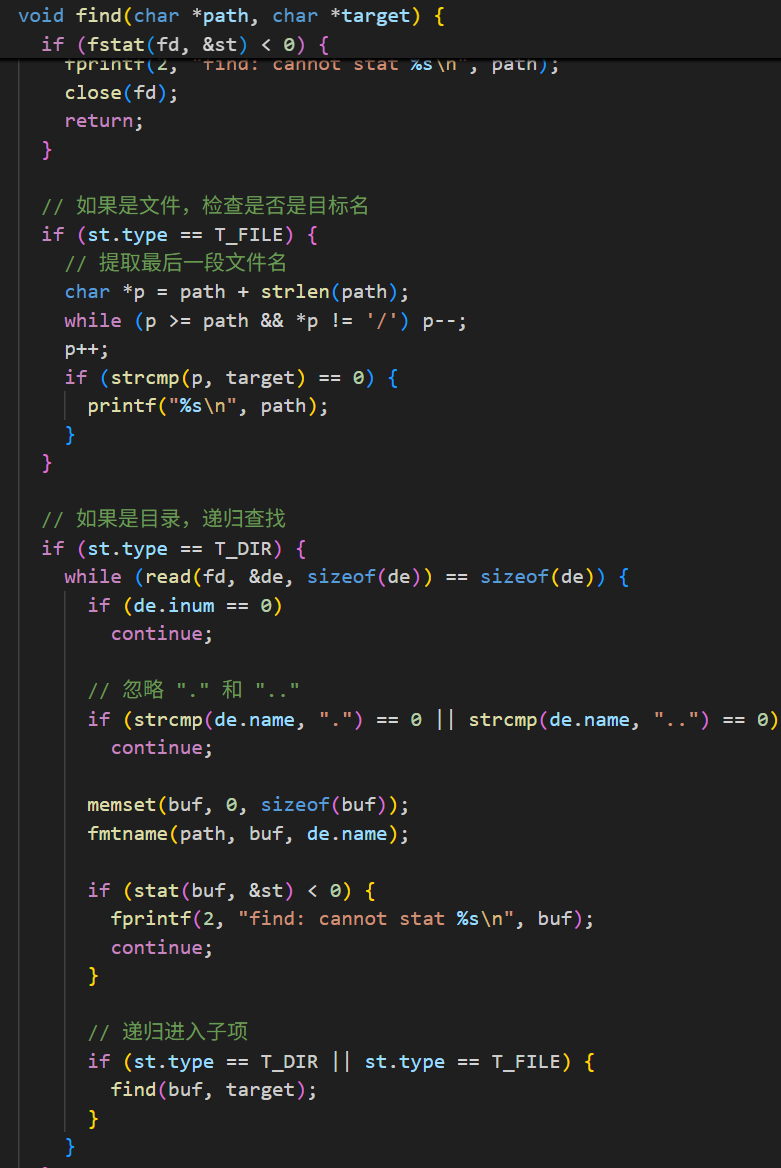
采用递归遍历法实现文件搜索，定义递归函数void find(\*path, \*target)

如果打开的是文件，则判断它的名字是否等于target；如果是目录（文件夹），则需要递归地遍历目录项，拼接完整路径路径newpath再递归地调用find(newpath,target)。

其中有几个关键的功能模块：打开目录或文件路径open()，获取文件类型fstat()，遍历目录项read()，比较文件名是否相同strcmp()。

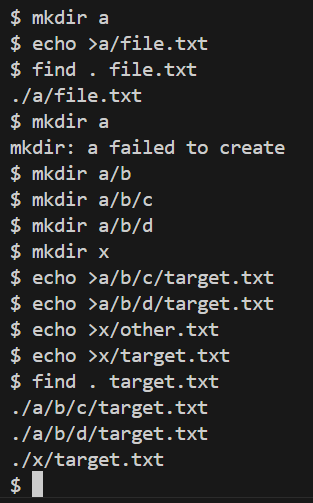
·添加并实现文件user/find.c

根据上述思路完成编程。下面是递归函数find实现（LINE：16）：



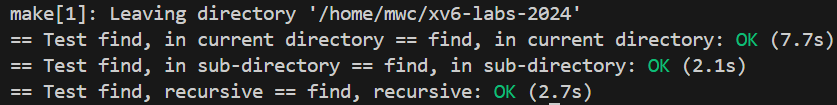
·shell测试与make grade提交测试：

在shell环境中，我们构造了一个相对复杂的多层路径环境，进行文件的查找，构造过程和结果如下图所示：



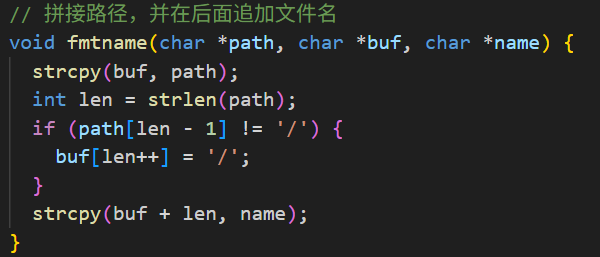
**分析：**很好地找到了所有子路径下符合要求的文件（文件名完全一致，不是部分匹配），并输出路径

测试也通过了：



1. 实验中遇到的问题和解决办法

·路径拼接，需要将当前路径拼接为完整路径，要注意缓冲区足够。代码如下图所示（LINE 6）



·防止无线递归，必须跳过上层目录标识“.”和“..”否则就会进入死循环。遍历到目录为上层目录标识直接跳过

·由于de.name不以’\0’结尾，需要手动复制到新的字符串中

1. 实验心得

通过本实验，进一步掌握了xv6下文件系统的遍历方法以及如何手动处理目录项与递归。处理字符数组、字符拼接与递归调用等程序设计基本技能得到了加强。这些基本功能在实现过程中往往对细节有着很高的要求，比如尾零的添加。

## 6.xargs ([moderate](https://pdos.csail.mit.edu/6.828/2024/labs/guidance.html))

1）实验目的

实现一个简化版的xargs程序，该程序的功能如下：

从标准输入中按行读取内容，每读完一行就将其作为参数加到命令行末尾，执行一次命令。

xargs的应用十分广泛，可以用于指令的拼接从而实现复杂的操作，扩展了指令的功能。例如可以用上一个实验编写的find功能通过xrags拼接一个grep（抓取）指令，可以直接定位到某个文件的某个内容。

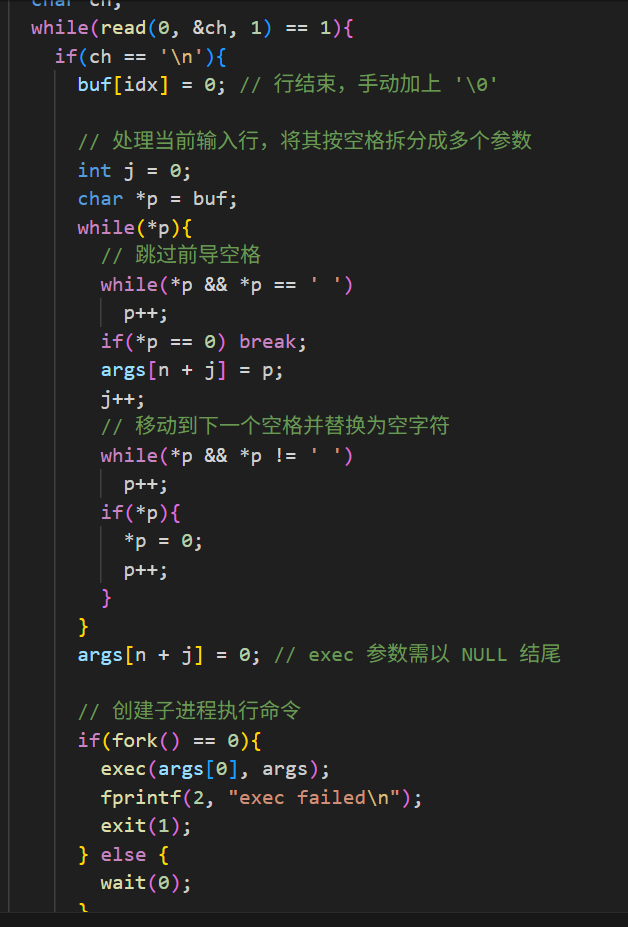
1. 实验步骤

·程序设计思路分析

xrags主要的工作流程可以划分为以下几个部分：解析命令行参数，读取标准输入的一行内容，拼接为新的参数表数组（argv），执行新命令、等待子进程结束。重复执行直到所有输入都被读完为止。

·具体代码实现

根据程序的设计思路编写代码，核心迭代部分如下（LINE 21）：

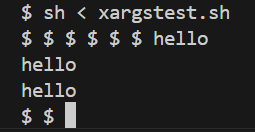


将文件保存为user/xrags.c并修改Makefile。程序实现中的难点与细节在3）中进一步展开。

·程序测试与结果分析

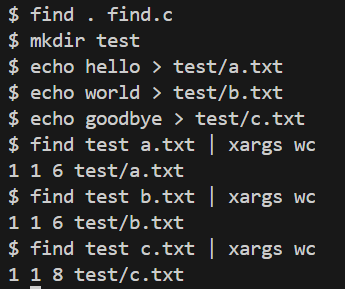
首先使用xv6自带脚本xargstext.sh在shell状态向下进行测试：

sh < xargstest.h



**分析：**输出结果与预期完全一致，算法成功。

再尝试用xrags算法拼接指令，构造一个更加实用的功能。利用了上一部分编写好的find指令，再使用xargs拼接一个wc（文件统计）功能，效果如下：



**分析：**可以看到命令找到了每个文件的路径并执行统计功能，左边先输出的是统计（行数 单词数 字节数）结果，再输出文件路径。这样就把两个功能整合到一起，具有一定的实用功能。

1. 实验中遇到的问题和解决办法

·在xv6中读取一整行是首先遇到的困难。由于xv6没有getline()，在实际实现的过程中使用read()逐个字节进行读取，读到换行符”\n”停止，并在结尾加上尾零。

·指令中需要进行参数分割，使用空格来分割参数，需要自行处理字符串的截断。

·每次收到前面一个标准输入给出的参数，xrags都会创建一个子进程，在这个子进程里执行exec()，实现拼接之后的指令效果；而父进程xrags等待子进程的执行完成。

1. 实验心得

通过本次实验，进一步加深了对exec()、fork()、wait()这三类系统调用的工作机制和使用场景。掌握了xrags方式的工作原理，并能够灵活运用从而实现命令行的组合与功能扩展。

学会了如何调试进程管理类程序，能够比较熟练地定位问题来源于子进程/父进程还是参数构造过程。