# 3efd64016a7238741e17ffca24e84606

# 操作系统

分时系统OS

模拟程序

8208221223

计科2203班

张子洋

2024 年 6 月 17 日

1. **【实验目的】**

设计编写分时系统OS模拟程序，模拟OS进程、内存与文件管理功能的实现。加深操作系统进程控制原语主要任务和过程的理解，加深操作系统内存分配的基本策略，加深文件管理的策略理解，加深操作系统以进程为核心的完整架构的理解。

提高综合性实验的分析、设计及编程实现能力。

1. **【实验内容】**

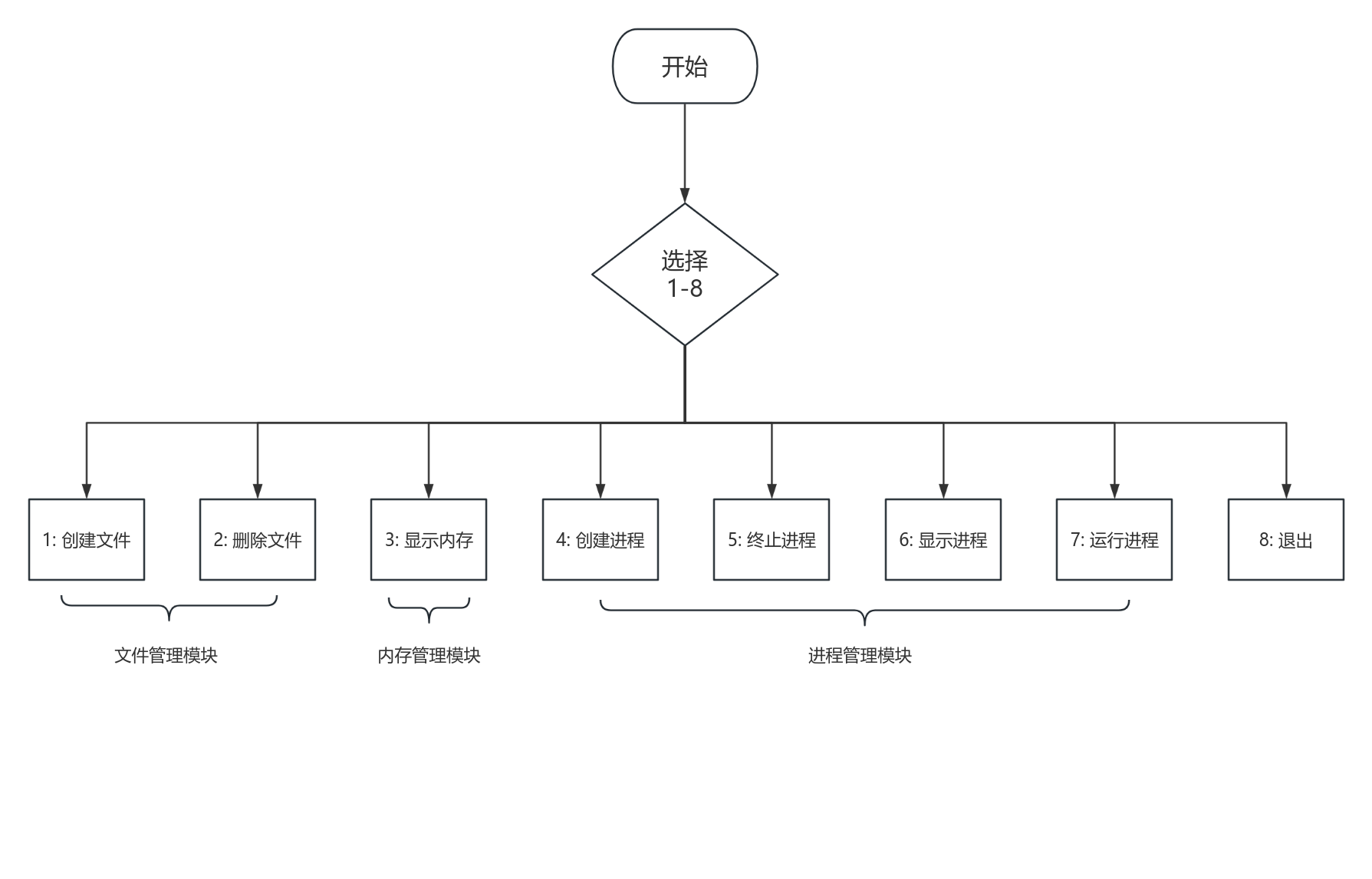
设计一个OS模拟程序，程序运行后提供一个交互界面或命令窗口，允许用户输入以下命令并可以对命令进行解释执行：

1. **creatproc命令：**提交作业命令，要求用户提供作业估计运行时长、内存大小。如果作业执行期间有I/O操作，需要提供该作业I/O操作的起始停止时间。如果该作业执行期间没有I/O操作，则该项参数为-1，-1。该命令的解释执行过程为对该作业创建对应的进程，完成PCB建立、存储空间分配等工作。（即执行OS的创建进程原语）
2. **killproc命令：**终止进程命令。该命令的解释执行过程为执行进程撤销原语。killproc命令格式：killproc [进程号]
3. **psproc命令：**显示所有进程状态命令。该命令的解释执行过程为显示出所有进程的状态信息，主要包括进程id，进程状态，存储空间地址。
4. **mem命令：**显示内存空间使用情况信息。该命令的解释执行过程为显示内存空间的占用和空闲情况。
5. **creatfile命令：**创建文件命令。该命令的解释执行过程为执行创建文件原语。creatfile命令格式：creatfile [文件名]
6. **deletefile命令**：删除文件命令。该命令的解释执行过程为执行删除文件原语。creatfile命令格式：creatfile [文件名]
7. **lsfile命令**：显示文件信息命令。该命令的解释执行过程为显示文件的FCB信息，主要显示文件id，文件名，文件物理位置（即盘块号）。

**设计思路提示：**

1. 内存空间和磁盘外存空间可用数组模拟。
2. 进程管理需要设计相应的函数模拟实现进程管理原语，包括创建进程，终止进程，阻塞进程，唤醒进程，进程调度等原语。
3. 进程调度算法选择RR。
4. 内存分配可选择可变分区策略或页式内存分配方案中任意一种。
5. 外存分配可选择连续分配、链式分配或索引分配任一种。
6. **【实验设计】**

流程图



**文件管理功能：**

FCB类用来管理文件的相关信息，采用变量保存信息，如文件名、起始块号、文件大小等。通过用户输入命令创建和删除文件，并显示文件的相关信息。

FileAllocationTable类中， boolean类型的数组变量allocationTable来模拟磁盘分配表，通过输入的磁盘大小与内存大小来分配，freeBlocks实现释放功能。

**进程管理**：

PCB类实现进程控制块功能，用变量存储信息，包含进程ID、内存起始地址和结束地址、状态、I/O操作开始时间、I/O操作结束时间等信息。

ProcessManager类使用进程表，内存管理器，就绪队列和阻塞队列进行模拟，通过这些数据结构，设计相应的函数，实现创建进程，终止进程，阻塞进程，唤醒进程，进程调度等原语。提供进程状态的显示功能，模拟操作系统的进程管理。

使用简单的轮转法（RR）进行进程调度。在RR调度中，每个进程分配一个固定的时间片，在这个时间片内运行，若在时间片内未完成，则将其放回到就绪队列的末尾，并调度下一个进程继续运行。

* 定义时间片： 在ProcessManager类中增加一个常量来定义时间片大小。
* runProcess方法： 在每个时间片内执行进程，并根据进程状态决定是将其重新放入队列还是终止。
* 处理I/O操作： 检查当前时间是否在I/O操作时间范围内，如果是，则将进程置于阻塞状态。

**内存管理**：

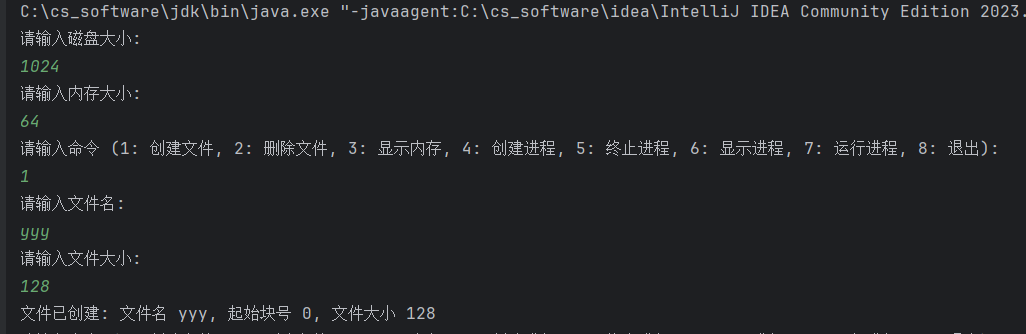
MemoryManager类负责模拟内存的分配和释放。我们采用可变分区来进行内存分配，内存总大小和空闲内存我们使用变量实现，进程ID与内存大小映射采用Map数据结构，allocateMemory，find，freeMemory，分别实现分配内存，查找可用内存块，释放内存功能。使用数组模拟内存的分配和释放，showMemoryStatus提供内存状态的显示功能。

**交互命令**：

* creatproc：创建进程
* killproc：终止进程
* psproc：显示所有进程状态
* mem：显示内存使用情况
* creatfile：创建文件
* deletefile：删除文件
* lsfile：显示文件信息

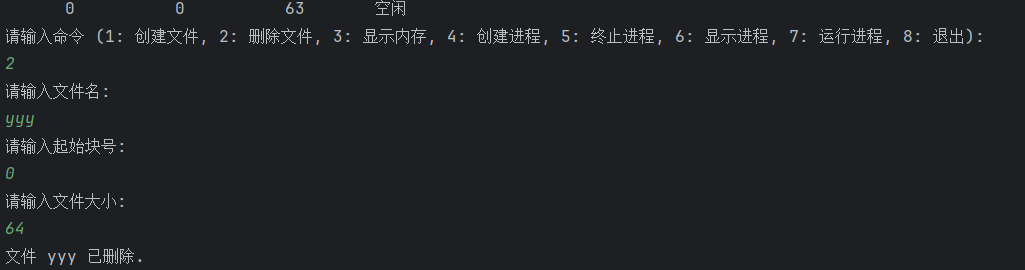
1. **【代码及运行结果】**
2. **创建文件**

public int getFileId() {  
 return fileId;  
}  
  
public void setFileId(int fileId) {  
 this.fileId = fileId;  
}  
  
public String getFileName() {  
 return fileName;  
}  
public void setFileName(String fileName) {  
 this.fileName = fileName;  
}  
public int getStartBlock() {  
 return startBlock;  
}  
public void setStartBlock(int startBlock) {  
 this.startBlock = startBlock;  
}  
public int getFileSize() {  
 return fileSize;  
}



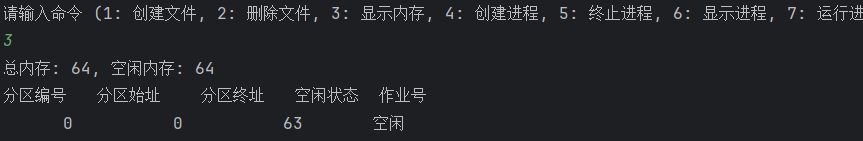
1. **删除文件**

// 分配盘块  
public int allocateBlocks(int numberOfBlocks) {  
 int startBlock = -1;  
 int count = 0;  
 for (int i = 0; i < diskSize + 1; i++) {  
 if (i < diskSize && !allocationTable[i]) {  
 if (count == 0) {  
 startBlock = i;  
 }  
 count++;  
 if (count == numberOfBlocks) {  
 break;  
 }  
 } else {  
 count = 0;  
 startBlock = -1;  
 }  
 }  
  
 if (startBlock != -1) {  
 for (int i = startBlock; i < startBlock + numberOfBlocks; i++) {  
 allocationTable[i] = true;  
 }  
 }  
  
 return startBlock;  
}  
  
// 释放盘块  
public void freeBlocks(int startBlock, int numberOfBlocks) {  
 for (int i = startBlock; i < startBlock + numberOfBlocks; i++) {  
 allocationTable[i] = false;  
 }  
}



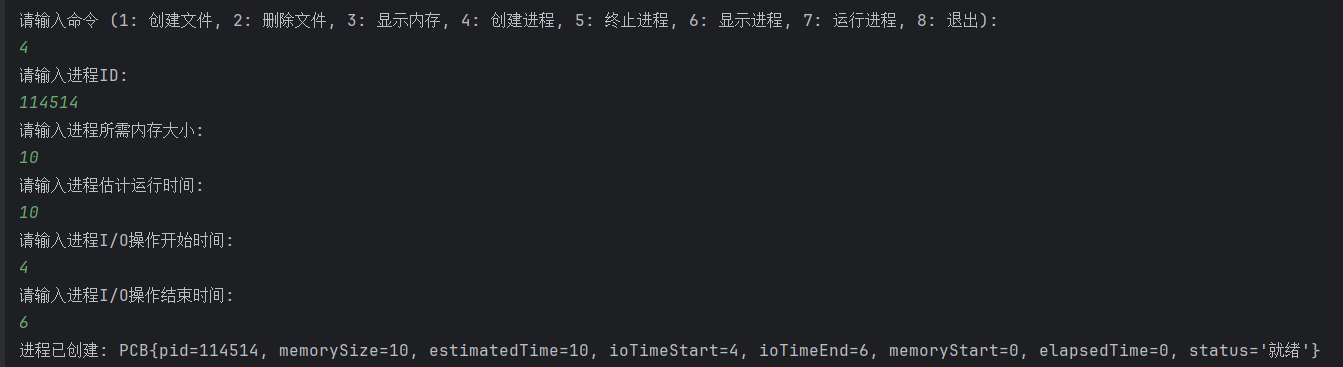
1. **显示内存**

// 分配内存  
boolean allocateMemory(int pid, int size, int index) {  
 if (index != -1) {  
 for (int i = index; i < index + size; i++) {  
 memory[i] = pid;  
 }  
 memoryMap.put(pid, size);  
 freeMemory -= size;  
 return true;  
 } else {  
 return false;  
 }  
}  
  
// 查找可用内存块  
int find(int size) {  
 int i, j, index = -1, minMemory = Integer.MAX\_VALUE;  
 for (i = 0; i < totalMemory; i++) {  
 j = i;  
 if (memory[i] == 0) {  
 while (j < totalMemory && memory[j] == 0) {  
 j++;  
 }  
 }  
 if (j - i < minMemory && j - i >= size) {  
 index = i;  
 }  
 i = j;  
 }  
 return index;  
}  
  
// 释放内存  
void freeMemory(int pid) {  
 if (memoryMap.containsKey(pid)) {  
 int size = memoryMap.get(pid);  
 for (int i = 0; i < totalMemory; i++) {  
 if (memory[i] == pid) {  
 for (int j = i; j < i + size; j++) {  
 memory[j] = 0;  
 }  
 break;  
 }  
 }  
 freeMemory += size;  
 memoryMap.remove(pid);  
 }  
}  
  
// 显示内存使用情况  
void showMemoryStatus() {  
 System.out.println("总内存: " + totalMemory + ", 空闲内存: " + freeMemory);  
 System.out.println("分区编号 分区始址 分区终址 空闲状态 作业号");  
 int i = 0, j = i + 1, t = 0;  
 while (i < totalMemory) {  
 while (j < totalMemory && memory[j] == memory[j - 1]) {  
 j++;  
 }  
  
 if (memory[i] == 0) {  
 System.out.printf(" %6d %6d %6d 空闲\n", t, i, (j - 1));  
 } else {  
 System.out.printf(" %6d %6d %6d 占用 %6d\n", t, i, (j - 1), memory[i]);  
 }  
 t++;  
 i = j;  
 j++;  
 }  
}



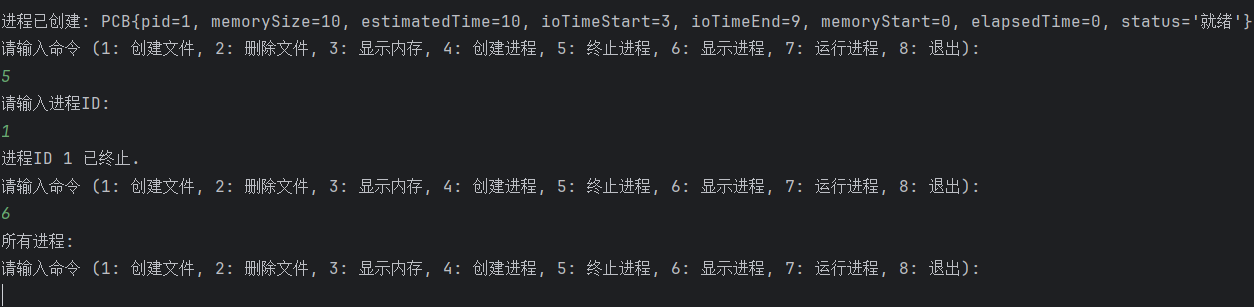
1. **创建进程**

// 创建进程  
void createProcess(int pid, int memorySize, int estimatedTime, int ioTimeStart, int ioTimeEnd) {  
 if (processes.containsKey(pid)) {  
 System.out.println("进程ID " + pid + " 已存在.");  
 return;  
 }  
 // 尝试分配内存  
 int memoryStart = memoryManager.find(memorySize);  
 if (memoryStart == -1) {  
 System.out.println("内存不足，无法创建进程ID " + pid);  
 return;  
 }  
 memoryManager.allocateMemory(pid, memorySize, memoryStart);  
 // 创建进程并添加到进程表和就绪队列  
 PCB newProcess = new PCB(pid, memorySize, estimatedTime, ioTimeStart, ioTimeEnd, memoryStart);  
 processes.put(pid, newProcess);  
 readyQueue.add(newProcess);  
  
 System.out.println("进程已创建: " + newProcess);  
}



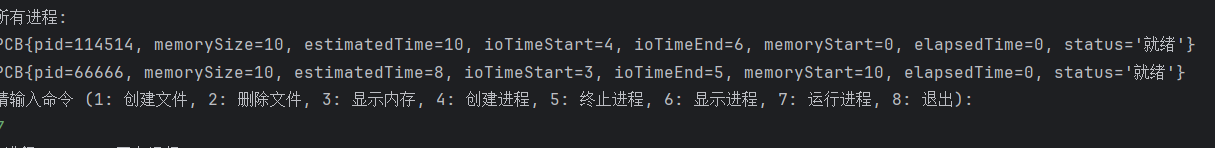
1. **终止进程**

// 终止进程  
void killProcess(int pid) {  
 if (!processes.containsKey(pid)) {  
 System.out.println("找不到进程ID " + pid);  
 return;  
 }  
 PCB process = processes.get(pid);  
 processes.remove(pid);  
 readyQueue.remove(process);  
 blockedQueue.remove(process);  
 memoryManager.freeMemory(pid);  
  
 System.out.println("进程ID " + pid + " 已终止.");  
}



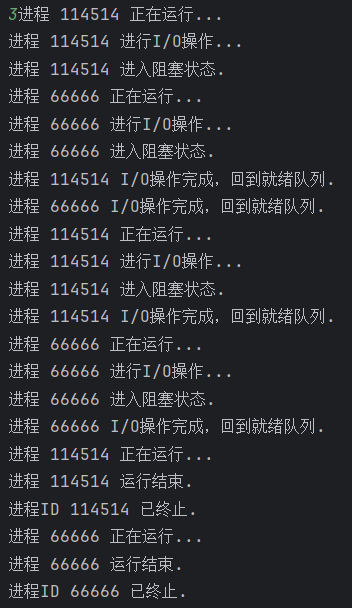
1. **显示进程**

void showAllProcesses() {  
 System.out.println("所有进程:");  
 for (PCB process : processes.values()) {  
 System.out.println(process); // 打印进程信息  
 }  
}



1. **运行进程**

void runProcess() {  
 int currentTime = 0;  
 while (!readyQueue.isEmpty() || !blockedQueue.isEmpty()) {  
 // 检查阻塞队列中的进程，是否完成I/O操作  
 Queue<PCB> tempQueue = new LinkedList<>();  
 while (!blockedQueue.isEmpty()) {  
 PCB blockedProcess = blockedQueue.poll();  
 if (currentTime >= blockedProcess.ioTimeEnd) {  
 blockedProcess.status = "就绪";  
 readyQueue.add(blockedProcess);  
 System.out.println("进程 " + blockedProcess.pid + " I/O操作完成，回到就绪队列.");  
 } else {  
 tempQueue.add(blockedProcess);  
 }  
 }  
 blockedQueue = tempQueue;  
  
 if (!readyQueue.isEmpty()) {  
 PCB currentProcess = readyQueue.poll(); // 获取就绪队列中的第一个进程  
 currentProcess.status = "运行"; // 设置其状态为运行  
 System.out.println("进程 " + currentProcess.pid + " 正在运行...");  
  
 int timeSlice = Math.min(TIME\_QUANTUM, currentProcess.estimatedTime - currentProcess.elapsedTime);  
  
 for (int i = 0; i < timeSlice; i++) {  
 currentTime++;  
 currentProcess.elapsedTime++;  
  
 if (currentProcess.elapsedTime >= currentProcess.ioTimeStart && currentProcess.elapsedTime < currentProcess.ioTimeEnd) {  
 System.out.println("进程 " + currentProcess.pid + " 进行I/O操作...");  
 blockProcess(currentProcess); // 阻塞进程进行I/O操作  
 break; // 结束当前时间片  
 }  
  
 if (currentProcess.elapsedTime >= currentProcess.estimatedTime) {  
 currentProcess.status = "结束";  
 System.out.println("进程 " + currentProcess.pid + " 运行结束.");  
 killProcess(currentProcess.pid);  
 break;  
 }  
 }  
  
 if (currentProcess.status.equals("运行")) {  
 currentProcess.status = "就绪";  
 readyQueue.add(currentProcess);  
 System.out.println("进程 " + currentProcess.pid + " 本次时间片用完，未完成.");  
 }  
 }  
 }  
}



1. **【总结与收获】**

通过本次实验，我对操作系统的核心功能有了更深的理解，尤其是进程管理、内存管理和文件管理。在模拟OS的过程中，我学会了如何通过编程实现进程的创建、终止、阻塞、唤醒以及调度等操作，同时也掌握了内存和文件的分配策略。此实验不仅提高了我对操作系统理论知识的理解，还增强了我解决实际问题的编程能力。