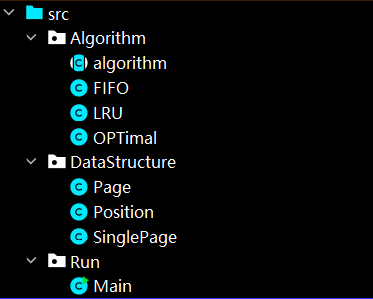
## 一.程序结构



## 二.Java程序代码

### Main.java

package Run;  
  
import Algorithm.FIFO;  
import Algorithm.LRU;  
import Algorithm.OPTimal;  
import DataStructure.Page;  
import DataStructure.Position;  
  
import java.util.ArrayList;  
import java.util.Random;  
  
*/\*\*  
 \* @author JasonGu  
 \* @date 2021/12/9 22:52  
 \*/*public class Main *{  
 /\*\*  
 \* n 地址序列数  
 \*/* final public static int n=400;  
 public Main*()  
 {  
 // 生成随机数据* Random random = new Random*()*;  
 *// 地址流数组* ArrayList*<*Position*>* positionArrayList = new ArrayList*<>()*;  
 *// 生成n=400条地址序列,前半部空间和后半部空间呈均匀分布* int m=0;  
  
 for*(*int i=0;i<*n*;i+=4*)  
 {  
 // [0,199]随机选一数m,记录到地址流数组中(这是非是顺序执行)* m = random.nextInt*(*200*)*;  
 positionArrayList.add*(*new Position*(*m*))*;  
 *// 顺序执行一条指令，将m+1记录进数组* positionArrayList.add*(*new Position*(*m+1*))*;  
 *// [200,399]随机选一数m\*,记录到地址流数组中(这是非是顺序执行)* m = random.nextInt*(*200*)*+200;  
 positionArrayList.add*(*new Position*(*m*))*;  
 *// 顺序执行一条指令，将m\*+1记录进数组* positionArrayList.add*(*new Position*(*m+1*))*;  
 *}* Page pages = new Page*()*;  
 pages.setPages*(*positionArrayList*)*;  
*// pages.PrintPage1();  
// pages.PrintPage2();* System.*out*.println*(*"页框数 FIFO缺页率 LRU 缺页率 OPT 缺页率"*)*;  
 *//循环运行，使用户内存容量从 4 页框到 40页框* for*(*int i=4;i<=40;i++*)  
 {* FIFO fifo = new FIFO*(*i,pages.pages*)*;  
 LRU lru = new LRU*(*i,pages.pages*)*;  
 OPTimal opt = new OPTimal*(*i,pages.pages*)*;  
 double p1 = fifo.calculate*(n)*;  
 double p2 = lru.calculate*(n)*;  
 double p3 = opt.calculate*(n)*;  
 System.*out*.println*(*"["+String.*format(*"%2d",i*)*+"]"  
 +" FIFO缺页率："+String.*format(*"%5.4f",p1*)* +" "+"LRU 缺页率："+String.*format(*"%5.4f",p2*)* +" "+"OPT缺页率："+String.*format(*"%5.4f",p3*))*;  
  
 *}* System.*out*.println*(*"\n页框数 FIFO命中率 LRU 命中率 OPT 命中率"*)*;  
 *//循环运行，使用户内存容量从 4 页框到 40页框* for*(*int i=4;i<=40;i++*)  
 {* FIFO fifo = new FIFO*(*i,pages.pages*)*;  
 LRU lru = new LRU*(*i,pages.pages*)*;  
 OPTimal opt = new OPTimal*(*i,pages.pages*)*;  
 double q1 = fifo.calculate2*(n)*;  
 double q2 = lru.calculate2*(n)*;  
 double q3 = opt.calculate2*(n)*;  
 System.*out*.println*(*"["+String.*format(*"%2d",i*)*+"]"  
 +" FIFO命中率："+String.*format(*"%5.4f",q1*)* +" "+"LRU 命中率："+String.*format(*"%5.4f",q2*)* +" "+"OPT命中率："+String.*format(*"%5.4f",q3*))*;  
  
 *}  
  
 }* public static void main*(*String*[]* args*) {* new Main*()*;  
 *}  
}*

### Position.java

package DataStructure;  
  
*/\*\*  
 \* @author JasonGu  
 \* @date 2021/12/9 22:30  
 \*/  
  
/\*\*  
 \* 指令地址  
 \*/*public class Position implements Cloneable ,Comparable*<*Position*>{* private int location;  
 private boolean visited = false;  
  
 public Position*(*int location*){* this.location = location;  
 *}* public int getLocation*(){* return this.location;  
 *}* public void setLocation*(*int location*){* this.location = location;  
 *}* public boolean isVisited*(){* return visited;  
 *}* public void toVisited*(){* this.visited = true;  
 *}* @Override  
 public String toString*() {  
// return "Postion{" +  
// "location=" + location +  
// ", visited=" + visited +  
// '}';* return String.*valueOf(*location*)*;  
 *}* @Override  
 public int compareTo*(*Position o*) {* return Integer.*compare(*this.getLocation*()*, o.getLocation*())*;  
 *}  
}*

### SinglePage.java

package DataStructure;

import java.util.ArrayList;

/\*\*

\* @author JasonGu

\* @date 2021/12/10 0:13

\*/

/\*\*

\* 页

\* 1个页存放10条指令（地址）

\*/

public class SinglePage implements Cloneable ,Comparable<SinglePage>{

private ArrayList<Position> locations;

private boolean visited = false;

/\*\*

\* i 该页在页地址流中的序列

\*/

private int i;

public SinglePage(){

locations = new ArrayList<>();

}

public void addLocation(Position location){

boolean x = false;

for(int i=0;i<locations.size();i++)

{

if(location.compareTo(this.locations.get(i))==0){

x = true;

break;

}

}

if(!x) this.locations.add(location);

}

public int getLocations() {

return i;

}

/\*\*

\* 设置在页地址流中的序列

\* @param i 序列

\*/

public void setLocations(int i){

this.i = i;

}

public boolean isVisited(){

return visited;

}

public void toVisited(){

this.visited = true;

}

public void PrintLocation() {

// return "Postion{" +

// "location=" + location +

// ", visited=" + visited +

// '}';

System.out.print("第"+String.format("%-2d",this.i)+"页 -- ");

for(Position location:locations){

System.out.print(String.format("%-4d",location.getLocation())+" ");

}

System.out.println();

}

@Override

public int compareTo(SinglePage o) {

return Integer.compare(this.i, o.i);

}

}

### Page.java

package DataStructure;  
  
*/\*\*  
 \* @author JasonGu  
 \* @date 2021/12/9 23:49  
 \*/*import java.util.ArrayList;  
  
*/\*\*  
 \* 页地址（页号）  
 \*/  
  
/\*\*  
 \* 用户虚存中，每 K 存放10 条指令，所以那 400 条指令访问地址所对应的页地址（页号）流为：指令访问地址为[0 ，9] 的地址为第 0 页；指令访问地址为[10，19]的地址为24  
 \* 第1 页；……。按这种方式，把 400 条指令组织进“40页”，并将“要访问的页号序  
 \* 列”记录到页地址流数组中。  
 \*/*public class Page implements Cloneable*{* public ArrayList*<*SinglePage*>* pages;  
  
 public Page*(){* pages = new ArrayList*<>()*;  
 *}* public void setPages*(*ArrayList*<*Position*>* positionArrayList*) {* int n=0;  
 SinglePage*[]*page = new SinglePage*[*41*]*;  
 for*(*int i=0;i<41;i++*){* page*[*i*]* = new SinglePage*()*;  
 page*[*i*]*.setLocations*(*i*)*;  
 *}* while*(*n<positionArrayList.size*()) {* int location = positionArrayList.get*(*n*)*.getLocation*()*;  
 page*[*location/10*]*.addLocation*(*positionArrayList.get*(*n*))*;  
 this.pages.add*(*page*[*location/10*])*;  
 n++;  
 *}  
 }* public void PrintPage1*() {* for*(*SinglePage page:pages*) {* page.PrintLocation*()*;  
 *}  
 }* public void PrintPage2*(){* int i=0;  
 System.*out*.println*(*"页地址流："*)*;  
 for*(*SinglePage page:pages*){* System.*out*.print*(*String.*format(*"%2d",page.getLocations*())*+" "*)*;  
 i++;  
 if*(*i==10*) {*System.*out*.println*()*;i=0;*}  
 }  
 }  
}*

### algorithm.java

package Algorithm;  
  
import DataStructure.SinglePage;  
  
import java.util.ArrayList;  
  
*/\*\*  
 \* @author JasonGu  
 \* @date 2021/12/10 9:08  
 \*/*public abstract class algorithm *{  
 /\*\*  
 \* 页框填满之前和之后的总缺页次数  
 \*/* public int p=0;  
 */\*\*  
 \* 页框填满之后的总缺页次数  
 \*/* public int q=0;  
 */\*\*  
 \* 页框数  
 \*/* public int num=0;  
 public ArrayList*<*SinglePage*>* pages;  
 */\*\*  
 \*  
 \* p 页框填满之前和之后的总缺页次数  
 \* @param i 页地址流大小  
 \* @return 缺页率  
 \*/* public double calculate*(*int i*)  
 {* return p/Double.*valueOf(*i*)*;  
 *}  
  
 /\*\*  
 \*  
 \* @param i 页地址流大小  
 \* @return 命中率  
 \*/* public double calculate2*(*int i*)  
 {* return 1-*(*double*)*q/i;  
 *}* public boolean compare*(*SinglePage page1, SinglePage page2*){* if *(*page1.compareTo*(*page2*)*==0*) {* return true;  
 *}* else return false;  
 *}  
}*

### FIFO.java

package Algorithm;  
  
*/\*\*  
 \* @author JasonGu  
 \* @date 2021/12/9 22:17  
 \* 先进先出算法  
 \* FIFO策略把分配给进程的页框视为一个缓冲区，并按循环方式移动页  
 \* 隐含逻辑：置换驻留在内存时间最长的页  
 \* 写另外两个算法只用这两个方法够用了  
 \* fIFOlist.set(hand,pages.get(x));将页地址流的第x位的页号set进指针所指的位  
 \* compare(pages.get(x), fIFOlist.get(j));对比页地址流第x位的页号和内存中第j页的页号是否相同  
 \*/*import DataStructure.SinglePage;  
  
import java.util.ArrayList;  
  
public class FIFO extends algorithm *{* public ArrayList*<*SinglePage*>* fIFOlist;  
 */\*\*  
 \* 指针  
 \*/* private int hand=0;  
  
 */\*\*  
 \*  
 \* @param i 内存页框数  
 \* @param pages 页地址流，存着SinglePage  
 \*/* public FIFO*(*int i,ArrayList*<*SinglePage*>* pages*) {* this.num = i;  
 fIFOlist = new ArrayList*<>(*i*)*;  
 *// 内存列表* for *(*int x = 0; x < pages.size*()*; x++*)  
 // 访问页地址流里的第x位存的页号  
 {  
 // 如果此时内存中的页框没用完* int a = 0;  
 boolean A = true,B = false;  
 while *(*x>=1&&a < fIFOlist.size*()) {  
 // compare(pages.get(x), fIFOlist.get(j)) 对比页地址流第x位的页号和内存中第j页的页号是否相同* if *(*compare*(*pages.get*(*x*)*, fIFOlist.get*(*a*))) {  
 // 如果相同，不需要置换* A = false;  
 break;  
 *}  
 // 否则，看内存的下一个页框* else *{*a++;*}  
 }* if *(*fIFOlist.size*()*<i&&A*) {  
 // 向内存中添加页* fIFOlist.add*(*pages.get*(*x*))*;  
 p++;  
 *}* if*(*fIFOlist.size*()*<=i-1*)  
 {* B = true;  
 *}*else B = false;  
 *//页框用完后* int j = 0;  
 *//j 内存中第j个页框  
  
 //页框用完后，要再存入新页的话，首先要遍历内存里是否存在此页* while *(*j < fIFOlist.size*()*&& !B*) {  
 // compare(pages.get(x), fIFOlist.get(j)) 对比页地址流第x位的页号和内存中第j页的页号是否相同* if *(*compare*(*pages.get*(*x*)*, fIFOlist.get*(*j*))) {  
 // 如果相同，不需要置换* break;  
 *}  
 // 否则，看内存的下一个页框* else *{* j++;  
 *}  
 }  
 // 遍历完后，发现没有相同的，需要置换* if *(*j == fIFOlist.size*()) {  
 // 将页地址流的第x位的页号set进指针所指的位* fIFOlist.set*(*hand, pages.get*(*x*))*;  
 *// 指针移位* hand = *(*hand + 1*)* % fIFOlist.size*()*;  
 p++;q++;  
 *}  
 }  
 }  
 }*

### LRU.java

package Algorithm;  
  
*/\*\*  
 \* @author JasonGu  
 \* @date 2021/12/9 22:17  
 \*/*import DataStructure.SinglePage;  
  
import java.util.ArrayList;  
import java.util.Vector;  
  
*/\*\*  
 \* 最近最少使用算法  
 \*Least Recently Used策略：置换内存中最长时间未被引用的页  
 \*/*public class LRU extends algorithm {  
 public ArrayList<SinglePage> LRUlist;  
 private Vector<Integer> vectotList;  
 */\*\*  
 \* 指针  
 \*/* private int hand=0;  
  
 */\*\*  
 \* @param i 内存页框数  
 \* @param pages 页地址流，存着SinglePage  
 \*/* public LRU*(*int i, ArrayList*<*SinglePage*>* pages*) {* this.num = i;  
 LRUlist = new ArrayList*<>(*i*)*;  
 vectotList = new Vector*<>(*400*)*;  
 *// 内存列表* for *(*int x = 0; x < pages.size*()*; x++*)  
 // 访问页地址流里的第x位存的页号  
 {  
 // 如果此时内存中的页框没用完* int a = 0,b=0;  
 boolean A = true,B = false;  
 while *(*x>=1&&a < LRUlist.size*()) {  
 // compare(pages.get(x), fIFOlist.get(j)) 对比页地址流第x位的页号和内存中第j页的页号是否相同* if *(*compare*(*pages.get*(*x), LRUlist.get(a))) {  
 *// 如果相同，不需要置换* VisitOperate(pages.get(x).getLocations());  
 A = false;  
 break;  
 }  
 *// 否则，看内存的下一个页框* else {a++;}  
 }  
 if (LRUlist.size()<i&&A) {  
 *// 向内存中添加页* VisitOperate(pages.get(x).getLocations());  
 LRUlist.add(pages.get(x));  
 p++;  
 }  
 if(LRUlist.size()<=i-1)  
 {  
 B = true;  
 }  
 else B = false;  
 *//页框用完后* int j = 0;  
 *//j 内存中第j个页框  
  
 //页框用完后，要再存入新页的话，首先要遍历内存里是否存在此页* while (j < LRUlist.size()&& !B) {  
 *// compare(pages.get(x), fIFOlist.get(j)) 对比页地址流第x位的页号和内存中第j页的页号是否相同* if (compare(pages.get(x), LRUlist.get(j))) {  
 *// 如果相同，不需要置换* VisitOperate(pages.get(x).getLocations());  
 break;  
 }  
 *// 否则，看内存的下一个页框* else {  
 j++;  
 }  
 }  
 *// 遍历完后，发现没有相同的，需要置换* if (j == LRUlist.size()) {  
 *// 将页地址流的第x位的页号set进指针所指的页框* int removedPagenum;  
 try  
 {  
 removedPagenum = vectotList.remove(0);  
 }catch (ArrayIndexOutOfBoundsException e)  
 {  
 removedPagenum = -1;  
 }  
  
  
 for(SinglePage page:LRUlist){  
 if(removedPagenum==-1||removedPagenum==page.getLocations()){  
 break;  
 }  
 hand++;  
 }  
 LRUlist.set(hand,pages.get(x));  
 hand = 0; *// hand重新初始化* p++;q++;  
 }  
 }  
 }  
  
 */\*\*  
 \* 建一个栈，每次访问某一页时将其页号记录进栈顶，若栈中存在同样的页号，则将此页号删除再进行前面所说的操作  
 \* @param pagenum 页号  
 \*/* private void VisitOperate(int pagenum)  
 {  
 if(vectotList.size()==0){  
 vectotList.add(pagenum);  
 }  
 else {  
 vectotList.remove(Integer.valueOf(pagenum));  
 vectotList.addElement(Integer.valueOf(pagenum));  
 }  
 }  
}

### OPTimal.java

package Algorithm;  
  
*/\*\*  
 \* @author JasonGu  
 \* @date 2021/12/9 22:16  
 \* OPT 最佳置换法  
 \* OPT策略选择置换下次访问距当前时间最长的那些页，这种算法导致缺页中断最少  
 \* 它要求操作系统必须知道将来的事件，因此不可能实现，现在模拟实现  
 \*/*import DataStructure.SinglePage;  
import java.util.ArrayList;  
  
public class OPTimal extends algorithm *{* ArrayList*<*SinglePage*>* OPTlist;  
  
 public OPTimal*(*int i, ArrayList*<*SinglePage*>* pages*) {* this.num = i;  
 OPTlist = new ArrayList*<>(*i*)*;  
  
 *// 内存列表* for *(*int x = 0; x < pages.size*()*; x++*)  
 // 访问页地址流里的第x位存的页号  
 {  
 // 如果此时内存中的页框没用完* int a = 0;  
 boolean A = true, B = false;  
 while *(*x >= 1 && a < OPTlist.size*()) {  
 // compare(pages.get(x), fIFOlist.get(j)) 对比页地址流第x位的页号和内存中第j页的页号是否相同* if *(*compare*(*pages.get*(*x*)*, OPTlist.get*(*a*))) {  
 // 如果相同，不需要置换* A = false;  
 break;  
 *}  
 // 否则，看内存的下一个页框* else *{*a++;*}  
 }* if *(*OPTlist.size*()* < i && A*) {* OPTlist.add*(*pages.get*(*x*))*;  
 p++;  
 *}* if *(*OPTlist.size*()* <= i - 1) {  
 B = true;  
 }else B = false;  
 *//页框用完后* int j = 0;  
 *//j 内存中第j个页框  
  
 //页框用完后，要再存入新页的话，首先要遍历内存里是否存在此页* while (j < OPTlist.size() && !B) {  
 *// compare(pages.get(x), fIFOlist.get(j)) 对比页地址流第x位的页号和内存中第j页的页号是否相同* if (compare(pages.get(x), OPTlist.get(j))) {  
 *// 如果相同，不需要置换* break;  
 }  
 *// 否则，看内存的下一个页框* else {  
 j++;  
 }  
 }  
 *// 遍历完后，发现没有相同的，需要置换* if (j == OPTlist.size()) {  
 int hand = Find(OPTlist,pages,x+1);  
 if(hand>=0){OPTlist.set(hand, pages.get(x));  
 p++;q++;}  
 }  
 }  
 }  
  
 */\*\*  
 \* 寻找下次访问距当前时间最长的那些页  
 \* @param pages 页地址流  
 \* @param OPTlist 内存列表  
 \* @param X 开始寻找的位置指针  
 \* @return hand 替换的页框的指针  
 \*/* private int Find(ArrayList<SinglePage> OPTlist,ArrayList<SinglePage> pages,int X) {  
 int hand=-1,max=0;  
 for(int a=0;a<OPTlist.size();a++){  
 for(int x=X;x<pages.size();x++)  
 {  
 if(compare(OPTlist.get(a),pages.get(x))){  
 int m = x-X-1;  
 if(m>=max){  
 max = m;  
 hand = a;  
 break;  
 }  
 }  
 }  
 }  
 return hand;  
 }  
  
}

### 三．JavaScript程序代码

/\*\*\*\*\*\*

\* addressArr[] : 地址流数组，用于存放随机生成的 400 个范围在 0-399 的指令地址

\* pageArr[] : 页号流数组，用于存放 400 个由地址流转换过来的范围在 0-39 的页号

\* fifoArr[] : FIFO 算法的页框数组，用于存放 FIFO 算法置换过程中的页框

\* lruArr[] : LRU 算法的页框数组，用于存放 LRU 算法置换过程中的页框

\* optArr[] : OPT 算法的页框数组，用于存放 OPT 算法置换过程中的页框

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* fifoQueue : 页号在页框中的模拟队列，用于模拟 FIFO 算法的页号在页框中的置换队列

\* lruQueue : 页号在页框中的模拟队列，用于模拟 LRU 算法的页号在页框中的置换队列

\* optQueue : 页号在页框中的模拟队列，用于模拟 OPT 算法的页号在页框中的置换队列

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* function random( ) { } : 用于生成 400 个范围在 0-399 的地址流，并将其转换位 400 个范围在 0-39 的页号流

\* function FIFO( ) { } : 实现 FIFO 页面置换算法

\* function LRU( ) { } : 实现 LRU 页面置换算法

\* function OPT( ) { } : 实现 OPT 页面置换算法

\* function replace( ) { } : simulation.js 程序的主函数，调用 FIFO()函数、LRU()函数以及 OPT()函数，实现从 4-40 页框的三大页面置换算法——FIFO 算法、LRU 算法以及 OPT 算法的页面置换，与此同时将 4-40 个页框对应的 FIFO 算法、LRU 算法以及 OPT 算法的命中率以表格的形式显示在图形用户界面上。

\*/

var addressArr = [],

pageArr = [];

var fifoArr = [],

lruArr = [],

optArr = []; //页框数组

var num2 = 0,

num3 = 0,

num4 = 0; //缺页次数

// function random( ) { }函数伪代码如下：

function random() {

addressArr = [];

var randomNum;

for (let i = 0; i < 100; i++) {

//将数组按要求填满400个数

randomNum = Math.floor(Math.random() \* 200); //随机生成一个[0,200)之间的整数

addressArr.push(randomNum); //将m推进队列里

if (randomNum == 199) {

//如果是199，则加1就溢出了

addressArr.push(0); //将0推进队列里

} else {

addressArr.push(randomNum + 1); //将m+1推进队列里

}

randomNum = Math.floor(Math.random() \* 200 + 200); //随机生成一个[200,400)之间的整数

addressArr.push(randomNum); //将m’推进队列里

if (randomNum == 399) {

//如果是399，则加1就溢出了

addressArr.push(200); //将200推进队列里

} else {

addressArr.push(randomNum + 1); //将m’+1推进队列里

}

}

pageArr = addressArr.map(function (x) {

return Math.floor(x / 10);

});

}

// function FIFO( ) { } : 实现 FIFO 页面置换算法

/\*\*\*\*\*\*fifo\*\*\*\*\*\*/

function FIFO(k, fifoArr, pageArr) {

for (let i = 0; i < pageArr.length; i++) {

if (fifoArr.length < k) {

//页框初始化

if (fifoArr.indexOf(pageArr[i]) == -1) {

//若页框数组里没有该页号了，则加入

fifoArr.push(pageArr[i]); //进队列

} else {

//若页框数组里有该页号了，则不用再加入了，循环页地址流中的下一个页号

continue;

}

} else {

//页框初始化后

if (fifoArr.indexOf(pageArr[i]) != -1) {

//若页框数组里有该页号了，则循环页地址流中的下一个页号

continue;

} else {

//若页框数组里没有该页号，则删掉第一个元素（因为它最早进来），再从后面插入一个元素

fifoArr.shift(); //出队列

fifoArr.push(pageArr[i]); //进队列

num2++; //同时缺页数加1

}

}

}

}

// function LRU( ) { } : 实现 LRU 页面置换算法

/\*\*\*\*\*lru\*\*\*\*\*\*/

function LRU(k, lruArr, pageArr) {

for (let i = 0; i < pageArr.length; i++) {

//循环页地址流数组

if (lruArr.length < k) {

//页框初始化时

if (lruArr.indexOf(pageArr[i]) == -1) {

//页框队列里的元素都不与页地址流的页号不重复时

lruArr.push(pageArr[i]); //新的页号直接插入页框队列尾部

} else {

//页框队列里有元素与页地址流的页号重复时，将该页号调出插到队列尾

lruArr.splice(lruArr.indexOf(pageArr[i]), 1); //删除该元素

lruArr.push(pageArr[i]); //插到队列尾

}

} else {

//页框初始化后

if (lruArr.indexOf(pageArr[i]) == -1) {

//不重复时,删除队头页号，新的页号插入队列尾

lruArr.shift(); //删除队头页号

lruArr.push(pageArr[i]); //新来的页号插到队列尾

num3++; //缺页数加1

} else {

//重复时，与初始化时的有重复时的处理一样,将该重复的页号调到队列尾

lruArr.splice(lruArr.indexOf(pageArr[i]), 1); //删除

lruArr.push(pageArr[i]); //重新插到队列尾

}

}

}

}

// function OPT( ) { } : 实现 OPT 页面置换算法

/\*\*\*\*\*\*\*\*opt算法\*\*\*\*\*\*\*/

function OPT(k, optArr, pageArr) {

// var pageArr = [2,3,2,1,5,2,4,5,3,2,5,2]; //页地址流数组

let num5; //当缺页时判断是否可以置换了的标志

for (let i = 0; i < pageArr.length; i++) {

if (optArr.length < k) {

//页框初始化前

if (optArr.indexOf(pageArr[i]) != -1) {

//有重复，则跳出该循环

continue;

} else {

//无重复，则新页号直接插入到队列尾

optArr.push(pageArr[i]);

}

} else {

//页框初始化后

if (optArr.indexOf(pageArr[i]) != -1) {

//有重复，则也是直接跳出该循环

continue;

} else {

//无重复，

num4++; //缺页数加1

num5 = 0;

//从i开始往后循环数组1,两种情况下跳出循环

for (let j = i + 1; num5 < k - 1 && j < pageArr.length; j++) {

//注意这里的循环跳出条件，贼坑，看我下面注释掉的代码，应该是要取反的，因为是满足才进入，不满足跳出

if (optArr.indexOf(pageArr[j]) != -1) {

//pageArr后面有跟pageArr数组重复时，num2++，并将该元素调到队列尾

num5++;

optArr.splice(optArr.indexOf(pageArr[j]), 1);

optArr.push(pageArr[j]);

} else {

continue;

}

}

//跳出循环后，删除队头元素，将新元素加到队列尾

optArr.shift();

optArr.push(pageArr[i]);

}

}

}

}

function replace() {

// for (let i = 0; i < 40; i++) {

// console.log(pageArr[i]);

// }

for (let k = 4; k <= 40; k++) {

/\*\*\*\*\*\*fifo\*\*\*\*\*\*/

FIFO(k, fifoArr, pageArr);

/\*\*\*\*\*lru\*\*\*\*\*\*/

LRU(k, fifoArr, pageArr);

/\*\*\*\*\*\*\*\*opt算法\*\*\*\*\*\*\*/

OPT(k, fifoArr, pageArr);

/\*\*\*动态添加命中率数据到表格中\*\*\*/

num2 = 1 - num2 / 400;

num3 = 1 - num3 / 400;

num4 = 1 - num4 / 400;

num2 = num2.toFixed(4);

num3 = num3.toFixed(4);

num4 = num4.toFixed(4);

console.log(

"页框数为" + k + "时，" + "FIFO:" + num2 + " LRU:" + num3 + " OPT:" + num4

);

}

}

random();

replace();