

复旦大学计算机科学技术学院



# 编程方法与技术

## A.1. 线程等的复习

周扬帆

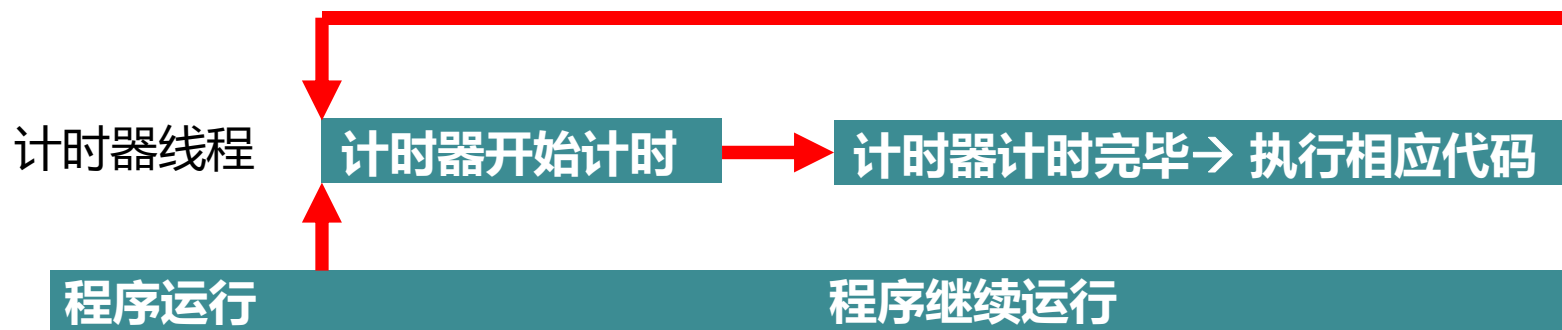
2021-2022第一学期



# 定时器：Timer

## □ 使用场景：程序的周期性工作

- 每一秒钟刷新界面的一条文本信息
- 每一分钟重连网络资源
- 每一小时检查电子邮件
- 每一天检查版本更新



# TimerTask

```
class MyTask extends TimerTask {  
    public void run() {  
        System.out.println("timer timeout: start run().");  
        ...  
    }  
}
```

- ❑ **TimerTask: 定义定时任务**
- ❑ **程序需继承TimerTask抽象类**
  - 实现其**run方法**
  - **run(): 实现定时器timeout之后的操作**
- ❑ **“填空”式编程**



# 定时器例子

```
import java.util.Timer;
import java.util.TimerTask;
...
Timer timer = new Timer();
timer.schedule( new TimerTask() {
    public void run() {
        System.out.println("timer timeout: start run().");
        ...
    }
}, 2000, 1000); // 设定指定的开始时间,此处为2000毫秒
                // 设定周期,此处为1000毫秒
```

优先级队列：二叉堆

## □ 构造Timer类对象

## □ 使用Timer类的schedule方法设置定时器

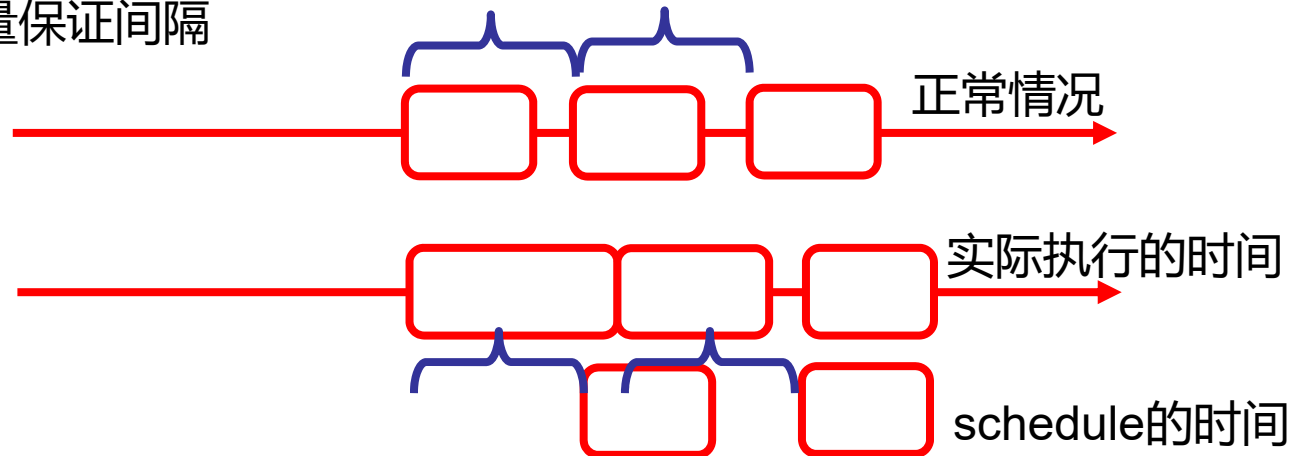
- 参数：一个**TimerTask**对象
- 其他参数：控制启动时间，周期性等



# Timer的schedule方法

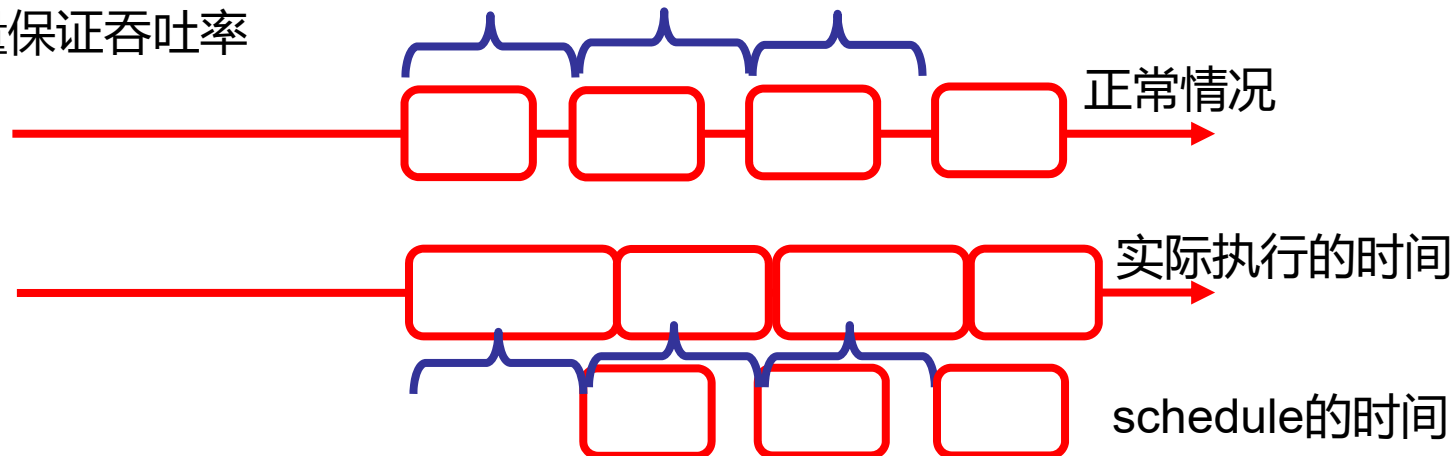
`void schedule(TimerTask task, long delay, long period)`

尽量保证间隔



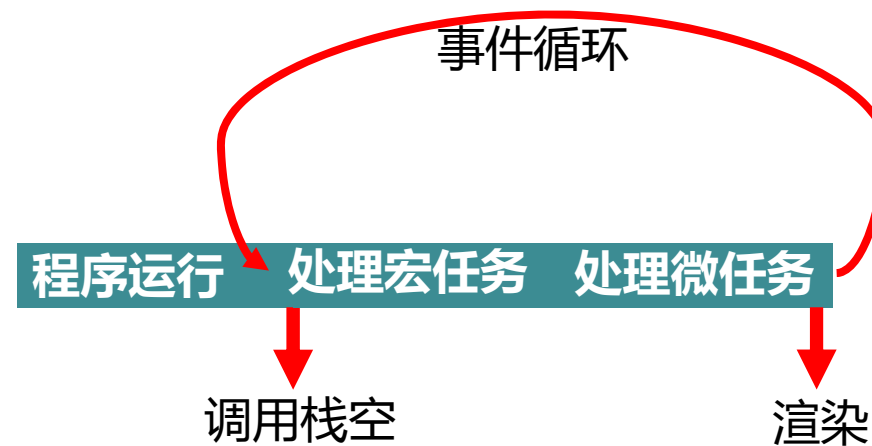
`void scheduleAtFixedRate(TimerTask task, long delay, long period)`

尽量保证吞吐率



# 定时器: Timer

## □ JavaScript



# 多线程编程方法1

## □ Thread类的定义

- Thread是抽象类，具体实现，需重载run方法

```
Class MyThread extends Thread{  
    @Override  
    public void run(){  
        System.out.println("Hello World!");  
    }  
}
```

## □ 线程的启动：Thread对象的调用

- 创建实例并调用start()方法

```
public static void main(String[] args) {  
    MyThread myThread = new MyThread();  
    myThread.start();  
}
```

“填空”式编程



# 多线程编程方法2

## □ 使用Runnable接口来实现多线程

```
class MyRunnable implements Runnable{  
    @Override  
    public void run() {  
        System.out.println("Hello World!");  
    }  
}  
  
public static void main(String[] args) {  
    Thread myThread = new Thread(new MyRunnable());  
    myThread.start();  
}
```

- 方法1直接继承Thread类限制了继承关系
  - 不能多重继承
- 方法2使用接口更为灵活
  - Thread构造时，传入Runnable对象
  - Runnable接口：实现run方法





# synchronized关键字

## □ 可修饰方法

```
public synchronized void myMethod(... ) {  
    ...  
}
```

- 对象这个方法在获得对象monitor（锁）才能运行
  - 否则在entry list上等着
- 任何时刻，只有一个线程可以获得一个对象的monitor
  - 保证永远只有一个线程在执行这个方法
  - return释放锁

## □ 可修饰块(block)

```
synchronized（共享对象的引用） {  
    // 获得该共享对象的monitor，才能运行  
    // 只能同时被一个线程执行  
}
```



# synchronized关键字

## □ 可修饰静态方法

```
public static synchronized void myMethod(... )  
{  
    ...  
}
```

- 类这个静态方法在获得类的monitor（锁）才能运行
  - 类的monitor → 类加载之后管理类数据的那个对象的monitor
  - 一个类只有一个



# 对象的monitor

3+1个monitor

```
class A {  
    ...  
}
```

```
A a1 = new A();  
A a2 = new A();  
A a3 = new A();
```

1. 加载A的时候, 有个object来存储A的基本信息, 它有一个monitor  
**static synchronized**方法需要有这个monitor才能运行

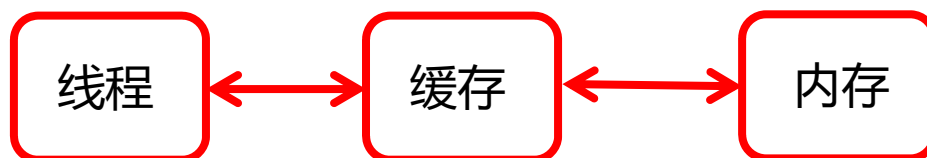
2. A的三个实例对象a1、a2、a3  
有三个monitor, 每个实例对应的实例化的**synchronized**方法  
需要有对应的monitor才能运行



# volatile关键字

## □ volatile关键字修饰变量

- 每次读写volatile变量，会强制缓存更新
- 因此volatile变量的值与此刻的内存是一致的



- 加入了**内存屏障**
  - 阻止指令重排乱序
  - 保证读写先后顺序



# 单例模式

```
class Log {  
    private static Log instance = null;  
    public static Log getInstance() {  
        if (instance == null) {  
            synchronized (作为锁的对象) {  
                if (instance == null) {  
                    Log instance = new Log ();  
                }  
            }  
        }  
        return instance;  
    }  
    private Log () {  
        ... //初始化, 如打开日志文件  
    }  
    public void logError(String log) {  
        ...  
    }  
}  
Log.getInstance().logError("This is a log record");
```

比如Log.class

- ❑ 双重检验+锁
- ❑ 为什么效率高?



# 单例模式

```
class Log {  
    private static class LogHolder {  
        private static final Log INSTANCE = new Log();  
    }  
    public static Log getInstance() {  
        return LogHolder.INSTANCE;  
    }  
    private Log () {  
        ... //初始化, 如打开日志文件  
    }  
    public void logError (String log) {  
        ...  
    }  
}  
Log.getInstance().logError("This is a log record");
```

- 不调用getInstance不会加载LogHolder
  - 不会new Log()



复旦大学计算机科学技术学院



# 编程方法与技术

## A.2. 并发与多线程：线程同步

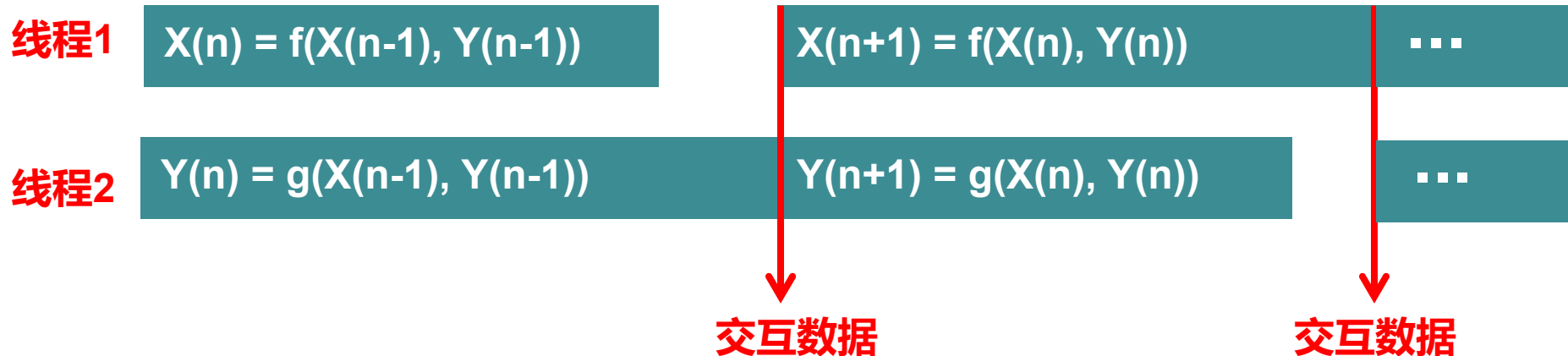
周扬帆

2021-2022第一学期



# 线程同步

- ❑ 线程之间有时需等待相互的结果以继续进行
- ❑ 例：多线程协同递归迭代运算 (?)





# 线程同步方法（错误方法）

## □ 根据变量同步

```
volatile boolean isDoneX = false;  
volatile boolean isDoneY = false;
```

### 线程1

```
isDoneX = false;  
//using X(n-1) and Y(n-1) to  
//update X(n)  
...  
isDoneX = true;  
for(;;) {  
    Thread.sleep(20);  
    if(isDoneY) {  
        break;  
    }  
}  
...
```

循环

### 线程2

```
isDoneY = false;  
//using X(n-1) and Y(n-1) to  
//update Y(n)  
...  
isDoneY = true;  
for(;;) {  
    Thread.sleep(20);  
    if(isDoneX) {  
        break;  
    }  
}  
...
```

循环



# 线程同步方法

## □ 加锁 (synchronized)

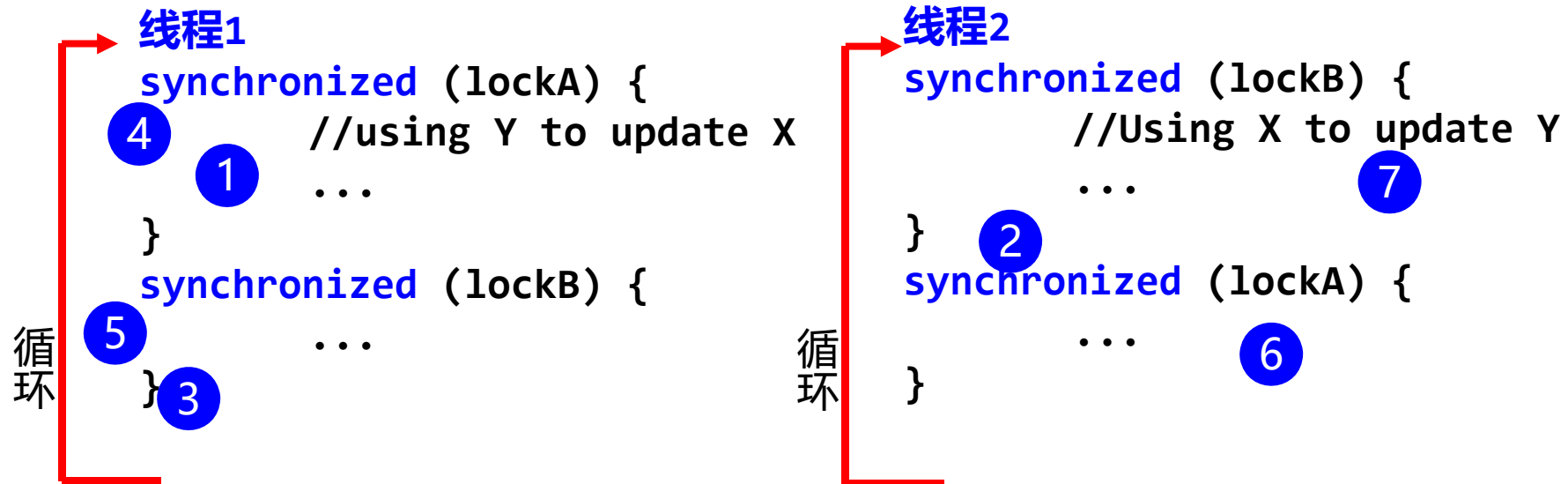
```
Object a = new Object();  
synchronized (a) {  
  
}
```

- 每个对象“天然”有一把锁（对象的monitor）
- 可以被synchronized加上锁
- synchronized块里的东西不能多线程重入(同时执行)



# 线程同步方法（错误方法）

## ❑ 加锁 (synchronized)



问题：抢锁



# wait/notify机制

```
Object lock = new Object();
```

线程1:

```
...  
try {  
    lock.wait(); //等待被notify, 然后继续  
} catch (InterruptedException e) {  
}  
}
```

线程2:

```
lock.notify(); //让线程1的lock.wait结束
```



# wait/notify机制

```
Object lock= new Object();
```

线程1:

```
...  
synchronized (lock) { //获得lock的锁  
    ...  
    try {  
        lock.wait(); //交出lock的锁, 等待被notify  
    } catch (InterruptedException e) {  
    }  
}
```

线程2:

```
...  
synchronized (lock) { //获得lock的锁  
    ...  
    lock.notify();  
}
```



# wait/notify机制例子

```
Thread t1 = new Thread() {
    public void run() {
        synchronized (object) {
            System.out.println("T1 start!");
            try {
                object.wait();
            } catch (InterruptedException e) {
            }
            System.out.println("T1 end!");
        }
    }
};

Thread t2 = new Thread() {
    public void run() {
        synchronized (object) {
            System.out.println("T2 start!");
            object.notify();
            System.out.println("T2 end!");
        }
    }
};

t1.start();
//Thread.sleep(3000);
t2.start();
```

能不能保证T1 T2都Start才End



# 线程同步方法（错误方法）

## ❑ wait-notify锁机制

```
volatile boolean isDoneX = false;  
volatile boolean isDoneY = false;
```

### 线程1

```
isDoneX = false;  
//use Y to update X  
...  
isDoneX = true;  
synchronized (lock) {  
    if(!isDoneY) {  
        try {  
            lock.wait();  
        } catch (InterruptedException e) {  
        }  
    }  
    lock.notify();  
}  
...  
}
```

循环

### 线程2

```
isDoneY = false;  
synchronized (lock) {  
    //use X to update Y  
    ...  
    isDoneY = true;  
    lock.notify();  
    lock.wait();  
}  
...  
}
```

循环



# 线程同步方法

## □ wait-notify锁机制

```
volatile boolean isDoneX = false;
volatile boolean isDoneY = false;
```

### 线程1

```
isDoneX = false;
//use Y to update X
```

```
...
```

```
isDoneX = true;
```

```
synchronized (lock) {
    if(!isDoneY) {
        try {
```

6

```
            lock.wait();
```

```
        } catch (InterruptedException e) {
        }
```

```
    }
```

```
    lock.notify();
```

```
    ...
}
```

### 线程2

```
isDoneY = false;
```

```
synchronized (lock) {
```

```
    //use X to update Y
```

```
    ...
```

```
    isDoneY = true;
```

```
    lock.notify();
```

```
    lock.wait();
```

7

8

```
    ...
}
```

循环

循环

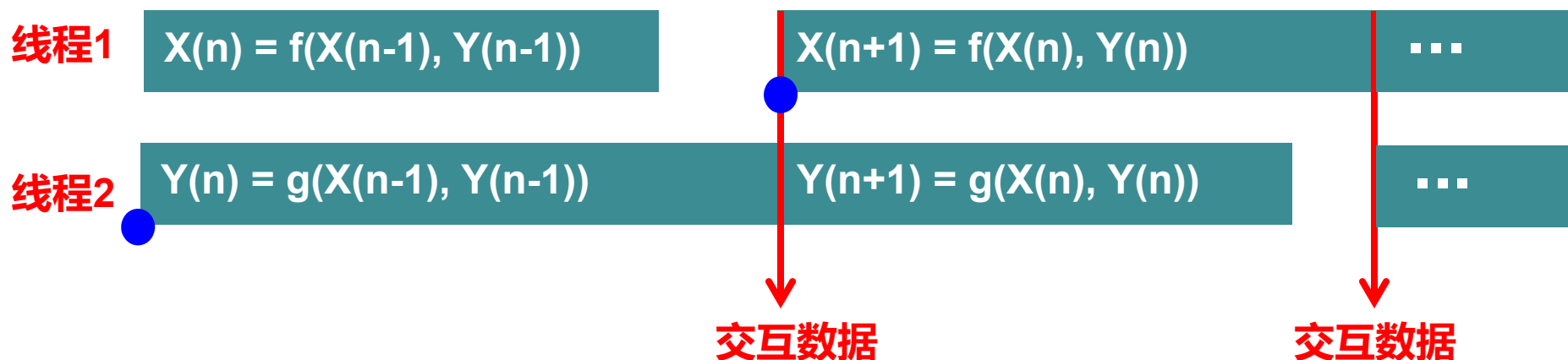
notify和wait之间有没有可能左边线程先notify了，右边线程在wait里还是不能往下执行？也就是9之后，是不是右边线程的8一定能往下执行？





# 线程同步

## □ 线程之间等待相互的结果以继续进行



上述方法都有问题？

解决：判断数据的版本号（循环index，如果一致才继续）



# wait/notify/notifyAll 机制

```
Object lock = new Object();
```

线程1:

```
...  
try {  
    lock.wait(); //等待被notify, 然后继续  
} catch (InterruptedException e) {  
}  
}
```

线程2:

```
...  
try {  
    lock.wait(); //等待被notify, 然后继续  
} catch (InterruptedException e) {  
}  
}
```

线程3:

```
lock.notifyAll(); //让线程1及线程2的lock.wait结束  
//lock.notify(); //让线程1或线程2的lock.wait结束
```



# wait/notify/notifyAll 机制

## □ wait()

- 必须持有monitor, 否则抛异常
- 交出去monitor
- 然后在wait set上等着被notify
- notify了之后去entry list等重新获得monitor

## □ notify()/notifyAll()

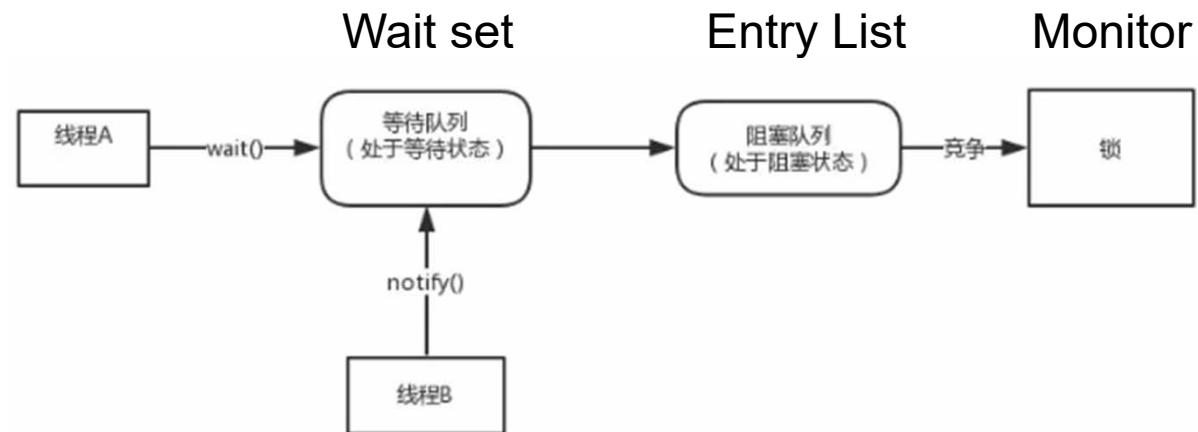
- 必须持有monitor, 否则抛异常
- wait set上的一个/所有线程被notify
- 不交出去monitor

```
lock.wait();  
...
```

```
synchronized (lock) {  
    ...  
}
```



# wait/notify/notifyAll 机制

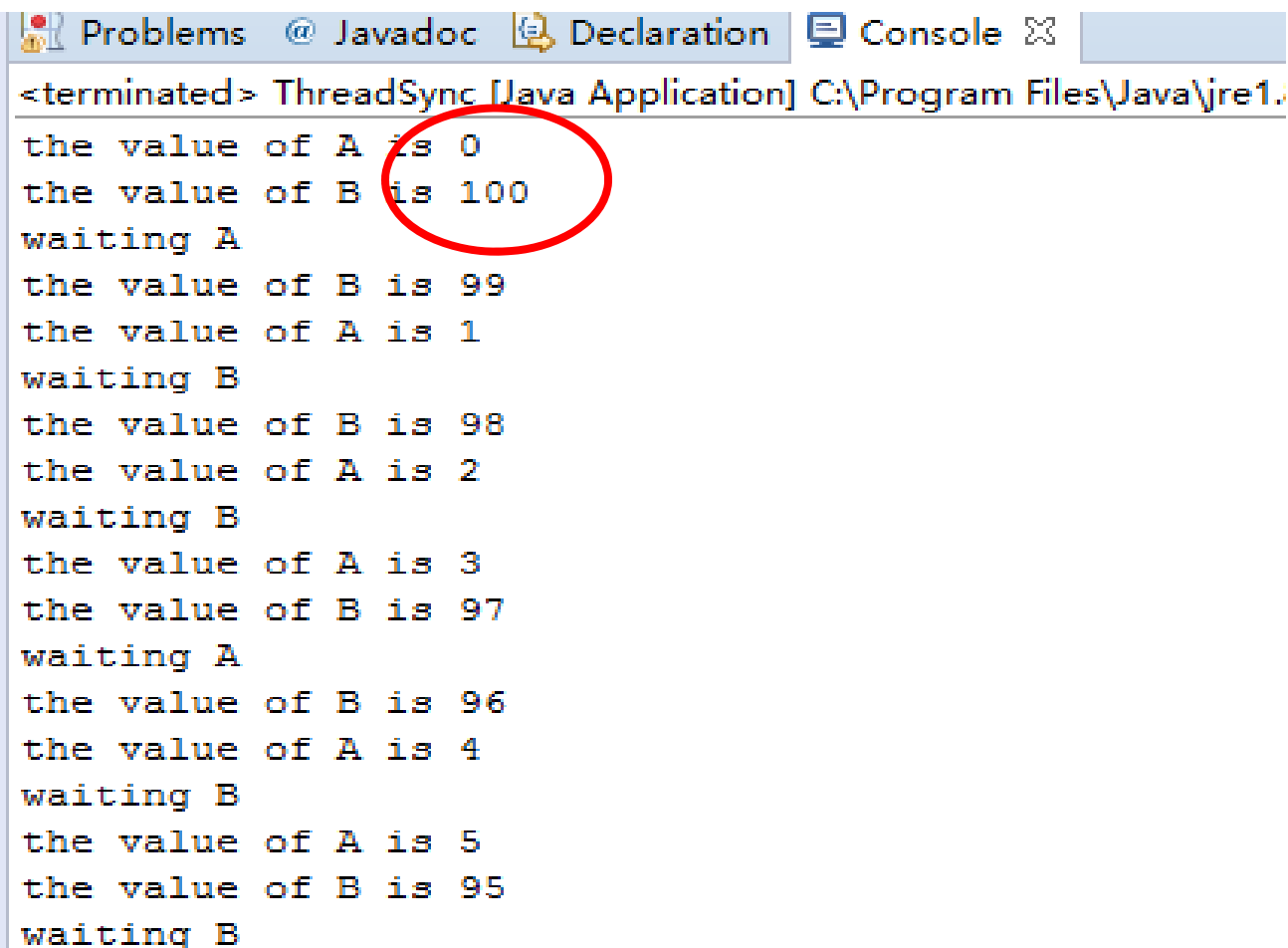


# 思考

- ❑ 理解同步的实现方法
- ❑ 理解notify()和notifyAll()
- ❑ 理解wait等唤醒和for循环等待某个变量的值为真，来等待其他线程完工，哪个好
  - 了解polling机制



# 作业



Problems @ Javadoc Declaration Console

<terminated> ThreadSync [Java Application] C:\Program Files\Java\jre1.6.0\_02\bin\java.exe

```
the value of A is 0
the value of B is 100
waiting A
the value of B is 99
the value of A is 1
waiting B
the value of B is 98
the value of A is 2
waiting B
the value of A is 3
the value of B is 97
waiting A
the value of B is 96
the value of A is 4
waiting B
the value of A is 5
the value of B is 95
waiting B
```



复旦大学计算机科学技术学院



# 编程方法与技术

## A.3. 并发与多线程：线程的控制

周扬帆

2021-2022第一学期



# 线程的执行过程

## □ 线程的启动: start → run

```
class MyRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("Hello World!");
    }
}

public static void main(String[] args) {
    Thread myThread = new Thread(new MyRunnable());
    myThread.start();
}
```

## □ 休眠

- 调用Thread.sleep(毫秒数)
- monitor不释放





# 线程的执行过程

## □ 休眠

### ■ 调用Thread.sleep(毫秒数)

```
class MyRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("Hello World!");
        try {
            for(;;) {
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("I am Interrupted");
        }
    }
}

Thread myThread = new Thread(new MyRunnable());
myThread.start();
```

```
...
myThread.interrupt();
```

什么时候执行?



# 线程的执行过程

## □ 交出执行权

### ■ 调用Thread.yield()

```
class MyRunnable implements Runnable{  
    @Override  
    public void run() {  
        for(;;) {  
            ... //busy doing something...  
            Thread.yield();  
        }  
    }  
}
```

→ 操作系统可以根据其他线程的优先级进行调度

→ 如果没有其他线程需要执行，则本线程继续执行

→ 优点？



# 线程的执行过程

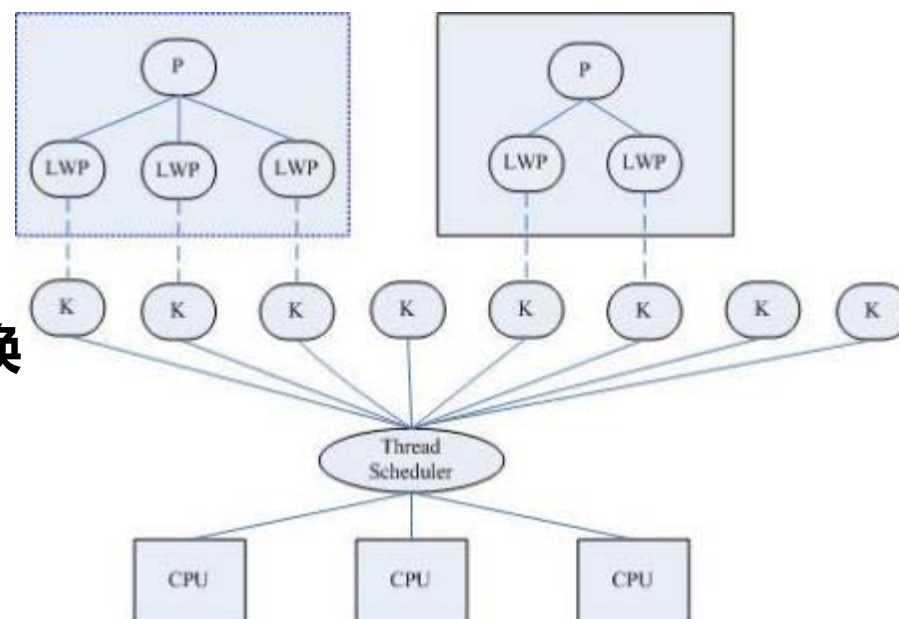
## □ 线程

### ■ 内核线程: KLT

- 操作系统内核管理
- 内核负责将KLT映射到处理器, 并进行调度

### ■ 轻量级进程: LWP

- 基于KLT 1:1实现
- 优点: 阻塞不影响别人
- 缺点?
  - 需要再用户态和内核态切换
  - 耗费内核资源



[Linux: NPTL](https://akkadia.org/drepper/nptl-design.pdf)

<https://akkadia.org/drepper/nptl-design.pdf>

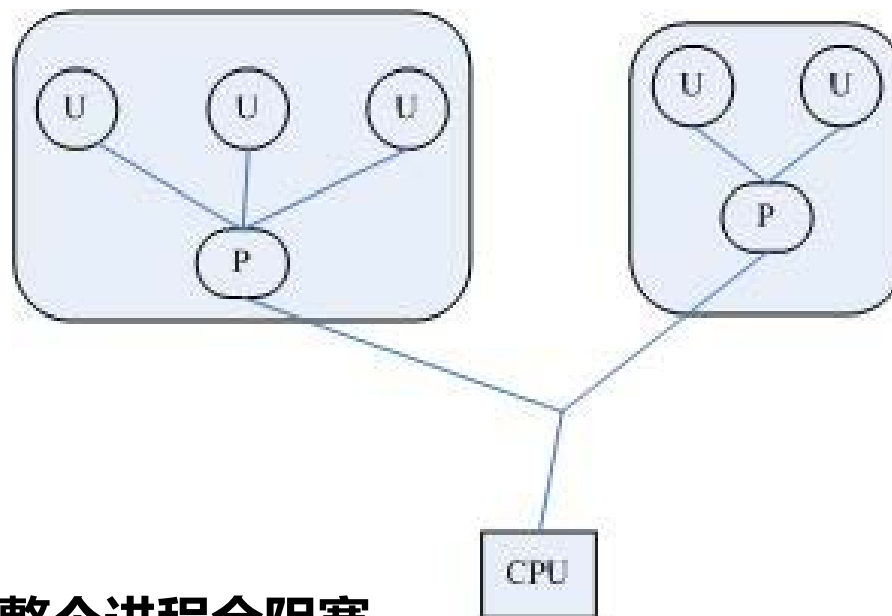


# 线程的执行过程

## □ 线程

### ■ 用户线程

- 实现为用户空间的线程库
- 管理在用户空间完成
- 优点: 快
- 缺点
  - 线程库实现比较复杂
  - 如果阻塞在系统调用中, 整个进程会阻塞

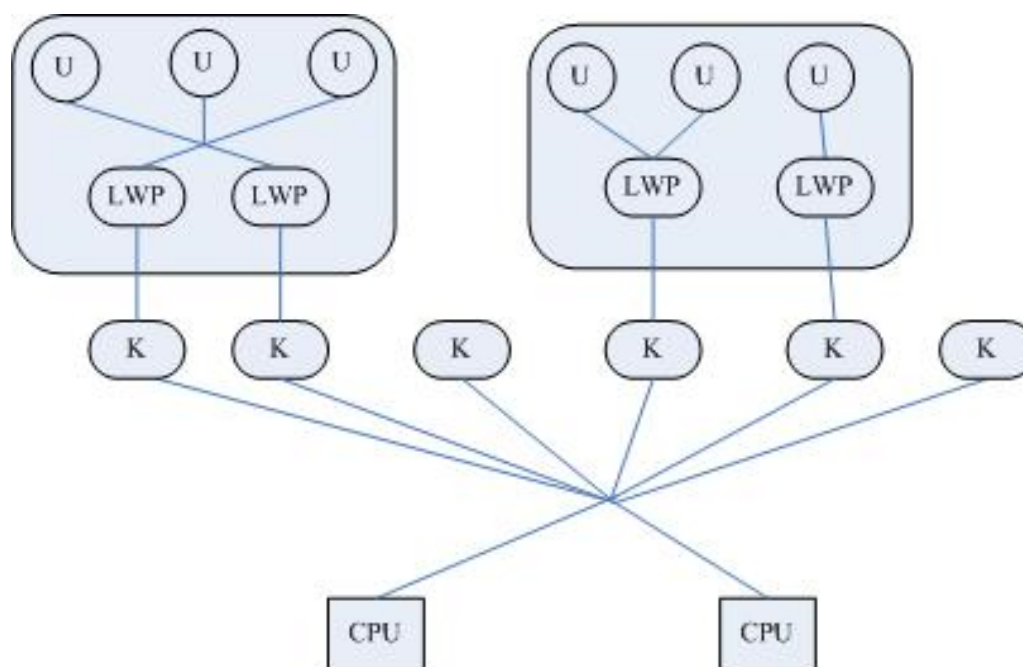


# 线程的执行过程

## □ 线程

### ■ 用户线程+LWP的实现

- 线程管理在用户空间
- 系统调用由LWP完成，降低进程被完全阻塞的风险



# 线程的执行过程

## □ 线程在Java的实现

- JDK 1.2之前

- Green Threads: 用用户线程机制实现

- 目前: 无规定

- Linux / Windows: LWP 1:1的机制实现

## □ 线程调度

- 协同式

- 抢占式

- 优先级

- 时间片



# 线程的执行过程

## □ 优先级

### ■ Thread类的setPriority/getPriority方法

```
class MyRunnable implements Runnable{  
    @Override  
    public void run() {  
        ... //in case that I need to do something critical...  
        setPriority(Thread.MAX_PRIORITY);  
    }  
}
```

### ■ Runnable类里怎么调用Thread的成员函数？

→ 静态: Thread.XXX

→ 非静态 ?

### ■ Thread.currentThread()获取当前的Thread的引用

Thread.currentThread().setPriority(Thread.MAX\_PRIORITY)

主线程也可以



# 线程的执行过程

## □ 优先级

- Java支持10种优先级
  - 最小: `Thread.MIN_PRIORITY`
  - 最大: `Thread.MAX_PRIORITY`
- Windows支持7种优先级
- Solaris支持 $2^{32}$ 种优先级
- 操作系统可能有对LWP的相应的优先级动态调整策略
- 优先级设置并不能严格保证的优先级次序





# 线程的执行过程

## □ 等待线程结束: 调用join方法

```
class MyRunnable implements Runnable {  
    public void run() {  
        try {  
            Thread.sleep(3000);  
            System.out.println("thread ends");  
        } catch (InterruptedException e) { }  
    }  
}  
  
public class ThreadDemo {  
    public static void main(String[] args) {  
        Thread myThread = new Thread(new MyRunnable());  
        myThread.start();  
        try {  
            myThread.join();  
        } catch (InterruptedException e) { }  
        System.out.println("main ends");  
    }  
}
```

本线程开始等待  
myThread线程结束



# 线程的执行过程：思考

## □ wait/notify/synchronized (复习)

```
Object lock = new Object();
```

线程1:

```
...
synchronized (lock) { //获得lock的锁
    ...
    try {
        lock.wait(); //交出lock的锁, 等待被notify
    } catch (InterruptedException e) {
    }
}
```

线程2:

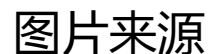
```
...
synchronized (lock) { //获得lock的锁
    ...
    lock.notify();
}
```

1. notify/notifyAll?
2. interrupt() ?
3. sleep/wait/yield区别



## 线程的状态

- ### 线程状态图



<https://my.oschina.net/mingdongcheng/blog/139263>



复旦大学计算机科学技术学院



# 编程方法与技术

## A.4. Java集合类

周扬帆

2021-2022第一学期



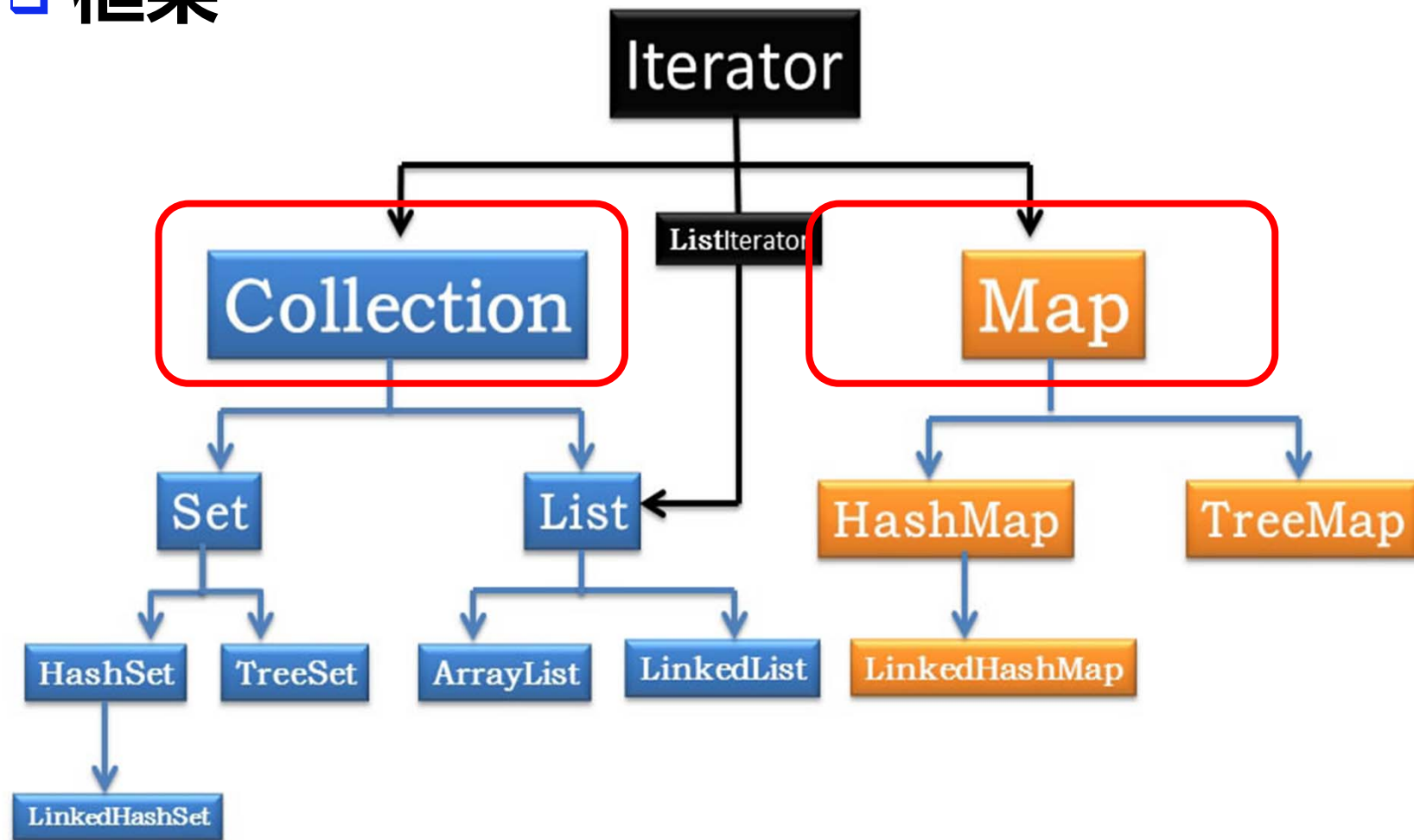
# Java容器(集合)类

- 如何存储类的一组对象(的引用)
  - 数组
  - ?
- Java容器类: 对一组对象(的引用)进行存储、管理
  - 实现Collection接口的容器类
  - 实现Map接口的容器类
    - 键值(Key-Value)对集合的管理



# Java容器(集合)类

## □ 框架



# Collection接口

- ❑ 如何方便支持不同类型的类的对象集
  - 如String对象集, Integer对象集
  - 泛型接口: **interface** Collection <E>
- ❑ 支持的操作?



# Collection接口

## □ 如何方便支持不同类型的类的对象集

- 如String对象集, Integer对象集
- 泛型接口: **interface** Collection <E>

## □ 支持的操作?

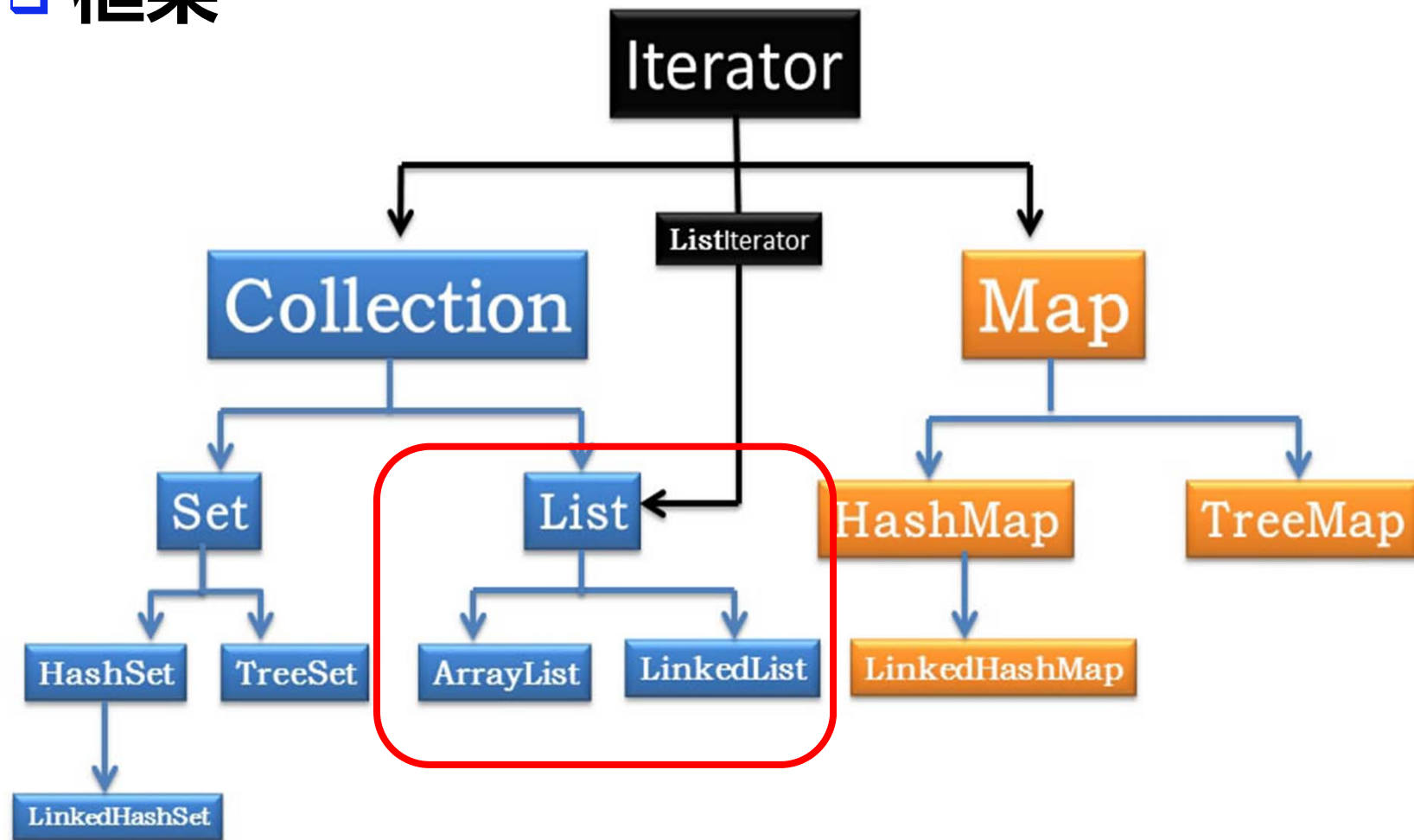
- add、remove、contains、isEmpty、size ...
- addAll: 把另一个集合所有数据加进来
- toArray: 转换为数组
- iterator: 返回一个Iterator对象, 用于遍历数据  
→ 后面会讲
- ...





# Java容器(集合)类

## □ 框架



# List接口

- ❑ 继承Collection接口
- ❑ 提供元素的基于index的次序管理
- ❑ 方法
  - void add(int index, Object element)
  - Object get(int index)
  - int indexOf(Object o): 第一个出现元素o的位置
  - int lastIndexOf(Object o)
  - Object remove(int index)
  - Object set(int index, Object element) .



# ArrayList/LinkedList

## □ ArrayList类

- 用数组的形式管理元素集合

## □ LinkedList类

- 用链表的形式管理元素集合

## □ 如何选择ArrayList/LinkedList

- 是否需要频繁删除，插入
- 是否需要频繁随机访问



# Iterator

## □ 用于对容器类里元素集合的遍历

```
List<String> words = new LinkedList<String>();  
words.add("hello");  
words.add("world");  
Iterator<String> iterator = words.iterator();  
while(iterator.hasNext()) {  
    String s = iterator.next();  
    System.out.println(s);  
}
```

得到这个collection的  
Iterator对象

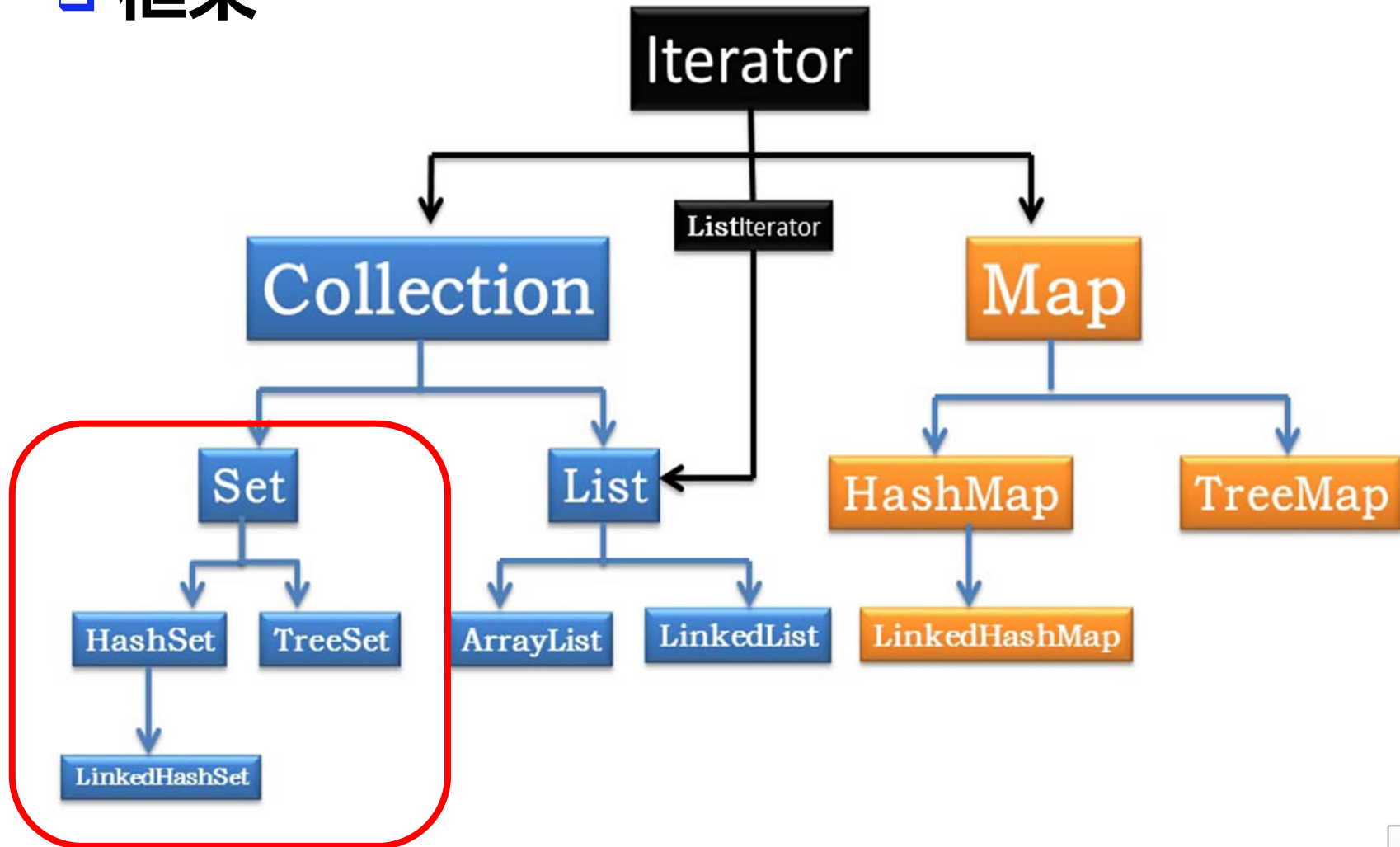
返回false: 遍历结束

用这个方法逐一得到  
元素



# Java容器(集合)类

## □ 框架



# Set接口

- ❑ 继承Collection接口
- ❑ Set 的每个元素必须**唯一**
- ❑ 不管理元素的次序

```
Set <String> words = new HashSet<String>();  
words.add("hello");  
words.add("hello");  
Iterator<String> iterator = words.iterator();  
while(iterator.hasNext()) {  
    String s = iterator.next();  
    System.out.println(s);  
}
```

**输出: hello**



# HashSet

- ❑ HashSet: 方便快速查找元素
- ❑ 基于HashMap实现
  - 后面会讲
- ❑ 元素唯一性的实现
  - 通过元素的两个方法`hashCode`和`equals`完成
  - 如果两个元素a和b的`hashCode()`返回值相同
    - 判断`a.equals(b)`是否返回`true`: 是, 认为是一样的元素
    - `==` ? 判断引用是否一样

```
String A = "hello";  
String B = "he".concat("llo");  
System.out.println(A.hashCode() == B.hashCode()); true  
System.out.println(A.equals(B)); true  
System.out.println(A == B); false
```



# HashSet

- ❑ HashSet: 方便快速查找元素
- ❑ 基于HashMap实现
  - 后面会讲
- ❑ 元素唯一性的实现
  - 通过元素的两个方法`hashCode`和`equals`完成
  - 如果两个元素a和b的`hashCode()`返回值相同
    - 判断`a.equals(b)`是否返回true: 是, 认为是一样的元素
    - `==` ? 判断引用是否一样
- ❑ 判断元素是否存在及删除等操作, 依赖元素的`hashCode`和`equals`方法
  - 可以编程覆盖





# HashSet

```
Set <Test> words = new HashSet<Test>();
words.add(new Test(1));
words.add(new Test(2));
Iterator<Test> iterator = words.iterator();
while(iterator.hasNext()) {
    Test s = iterator.next();
    System.out.print(s.value);
}
```

...

```
class Test {
    public int value;
    public Test(int value) {
        this.value = value;
    }
    public int hashCode() {
        return 0; //return value; 输出12
    }
    public boolean equals(Object t) {
        return true;
    } //本代码在业务系统中不合理，此处仅为展示重复的判断依据
}
```

输出: 1



# 关于hashCode和equals

## □ Object类的方法，可以被子类覆盖

- `int hashCode()`
- `boolean equals(Object o)`



## □ 为什么要用hashCode+equals方法判断重复

## □ 判断两个对象是否一样: equals方法

- 设想一个对象管理的数据量很大，“一样”定义为各变量的值相等
- equals方法需逐一比较各个域

## □ 要用hashCode+equals方法判断重复

- 逐一调用equals → 如果对象复杂，equals会比较耗时
- hashCode() → 可提取对象特征 → 大大加速查找



# 关于hashCode和equals

## □ 最简单的hashCode

- 根据对象的存储地址运算获得
- 根据对象的关键域的值经过运算获得  
→ 如String类的hashCode实现

```
public int hashCode() {  
    int h = hash;  
    if (h == 0 && value.length > 0) {  
        char val[] = value;  
  
        for (int i = 0; i < value.length; i++)  
            h = 31 * h + val[i];  
    }  
    hash = h;  
}  
return h;  
}
```

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$



# 关于hashCode和equals

- ❑ hashCode()返回值一样的对象，不一定是同一个对象
  - Hash的Nature: 多对一映射
  - "gdejicbegh"与字符串"hgebcijedg"具有相同的hashCode()返回值
  - 需再调用equals方法进一步判断



# 关于hashCode和equals

- ❑ 覆盖equals方法实现自己的比较方法，须记得覆盖hashCode方法
  - 否则？
  - equals返回true，但是hashCode返回不一样 → 会错误认为两个对象不“一样”
- ❑ 只要equals方法的比较操作作用到的数据没有被修改，那么hashCode方法必须返回同一个整数
  - 否则？



# 关于hashCode和equals

## □ equal属性

- 自反, 对称, 传递, 一致

## □ 实现 `boolean equals(Object o)`

- `==`符号
- `instanceof`
- Cast, 再比较关心的数据域



# TreeSet

## □ TreeSet: 有序的Set

- 不同于List通过index来表示顺序
- TreeSet用元素的**大小关系**来表示顺序
- 通过RB树存储元素

## □ 大小关系

- 通过元素的compareTo方法来获得
- 因此元素需要实现Comparable接口



# TreeSet

```
Set <Test> words = new TreeSet<Test>();
words.add(new Test(2));
words.add(new Test(1));
Iterator<Test> iterator = words.iterator();
while(iterator.hasNext()) {
    Test s = iterator.next();
    System.out.print(s.value);
}
...
class Test {
    public int value;
    public Test(int value) {
        this.value = value;
    }
    public int compareTo(Object o) {
        if(o.getClass() == Test.class) {
            return (value - ((Test)o).value);
        }
        return 0;
    }
}
```

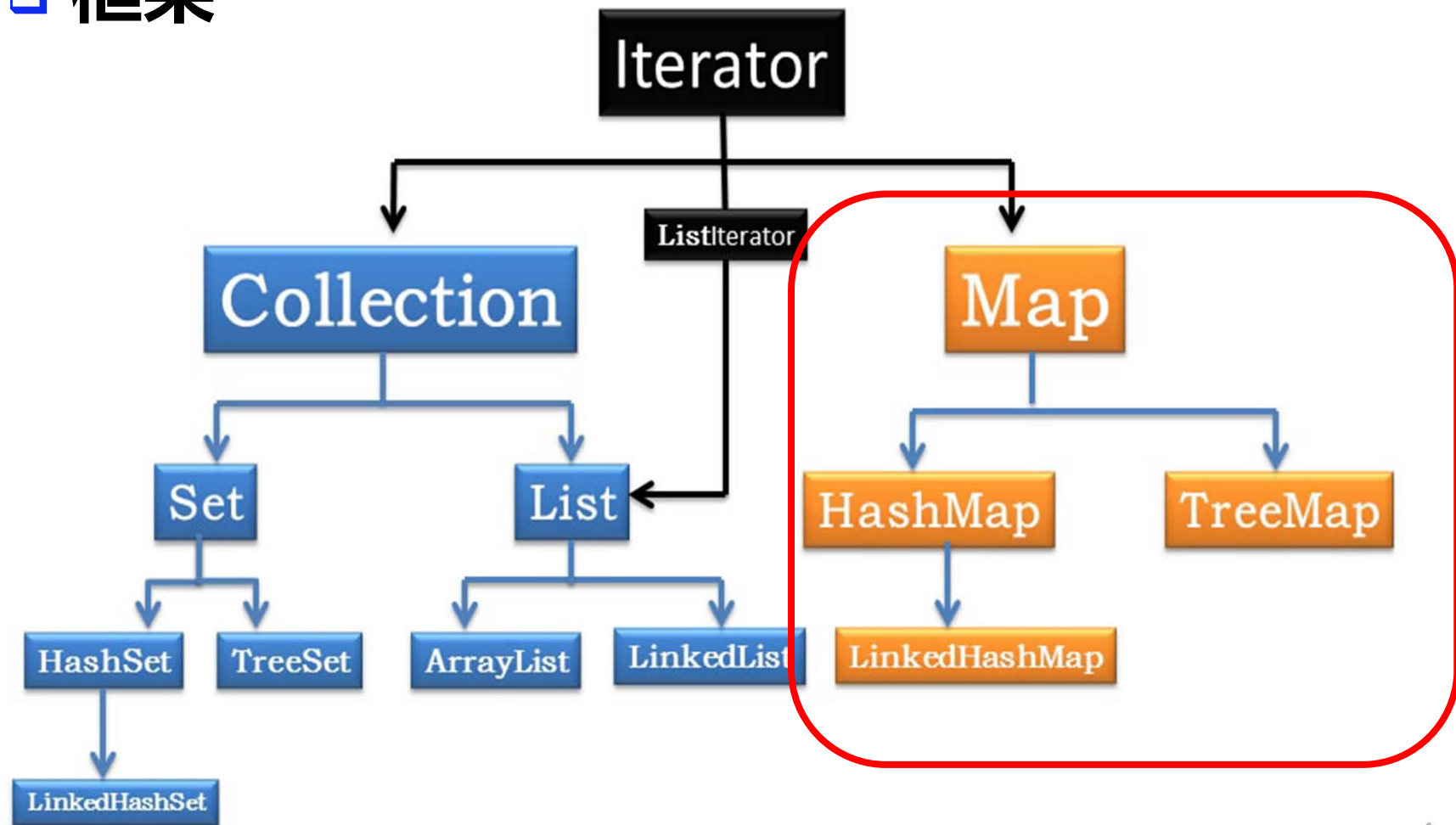
输出: 12





# Java容器(集合)类

## □ 框架



# Map接口

- ❑ 管理键值对 (key-value pair)
- ❑ 键值对
  - 每个值对应一个**全局唯一的键**
  - 以空间换时间：通过键查找，可以比通过值查找快
    - 如果值很复杂，键实现为比较短小简单的数据
    - 如：身份证号和个人户籍资料
- ❑ 同样为支持各种对象类型，Map为泛型接口
  - **interface** Map<K, V>



# Map接口

## □ 方法

- put、putAll
- remove
- get、values、keySet、entrySet
- size
- containsKey、containsValue



# Map类最简单的实现

## □ 两个链表

- 一个存Key
- 一个存Value
- 通过index一一对应

## □ 查找

- 在key链表中查给定的key，得到index
- 通过index在value链表中找对应的value

## □ 不足？



# HashMap

## □ 顾名思义

- Key → 取Hash → 方便通过Key查找Value

在HashMap中加入键值对

```
Map <String, String> map = new HashMap<String, String>();  
map.put("1", "Hello1");  
map.put("2", "Hello2");  
Iterator<String> iterator = map.keySet().iterator();  
while(iterator.hasNext()) {  
    String s = iterator.next();  
    System.out.print(s);  
}
```

返回一个所有key的Set

输出: 12



# HashMap

```
Map <String, String> map = new HashMap<String, String>();
map.put("1", "Hello1");
map.put("2", "Hello2");
Iterator<String> iterator = map.keySet().iterator();
while(iterator.hasNext()) {
    String k = iterator.next();
    String v = map.get(k);
    System.out.print(v);
}
```

得到这个key对应的value

输出: Hello1Hello2



# HashMap

```
Map <String, String> map = new HashMap<String, String>();
map.put("1", "Hello1");
map.put("2", "Hello2");
Iterator<Entry<String, String>> iterator = map.entrySet().iterator();
while(iterator.hasNext()) {
    Entry<String, String> v = iterator.next();
    System.out.print(v.getValue());
}
```

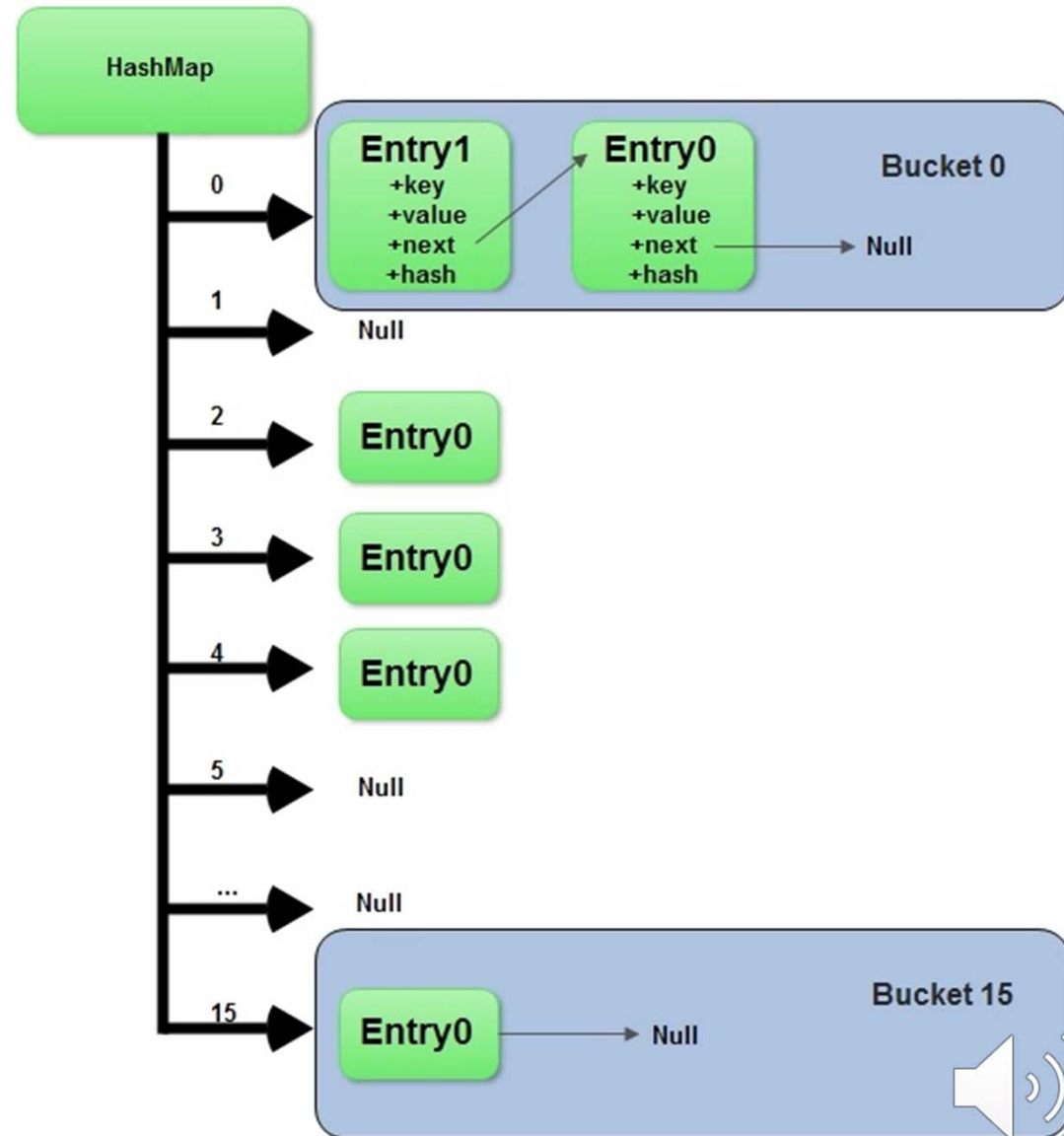
得到键值对的集合

输出: Hello1Hello2



# HashMap的存储

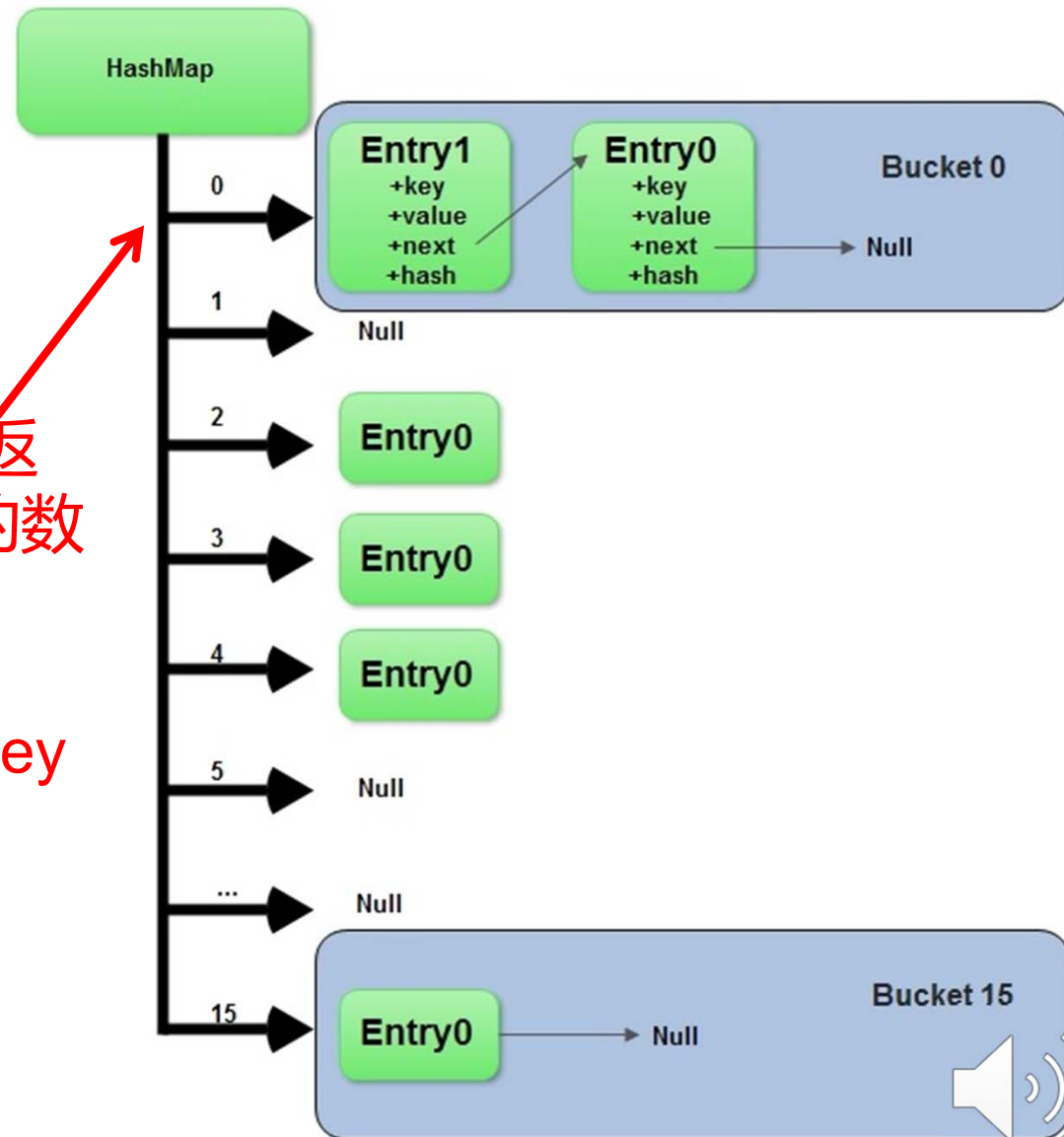
- ❑ 数组+链表
- ❑ 维护n个数组
- ❑ 每个数组对应一个链表





# HashMap的存储

1. 通过Key的hashCode返回  
值，运算得到对应的数  
组位置
  2. 在对应的链表中查找Key  
(可调用equals)
- 存在: 覆盖  
不存在: 新增链表元素



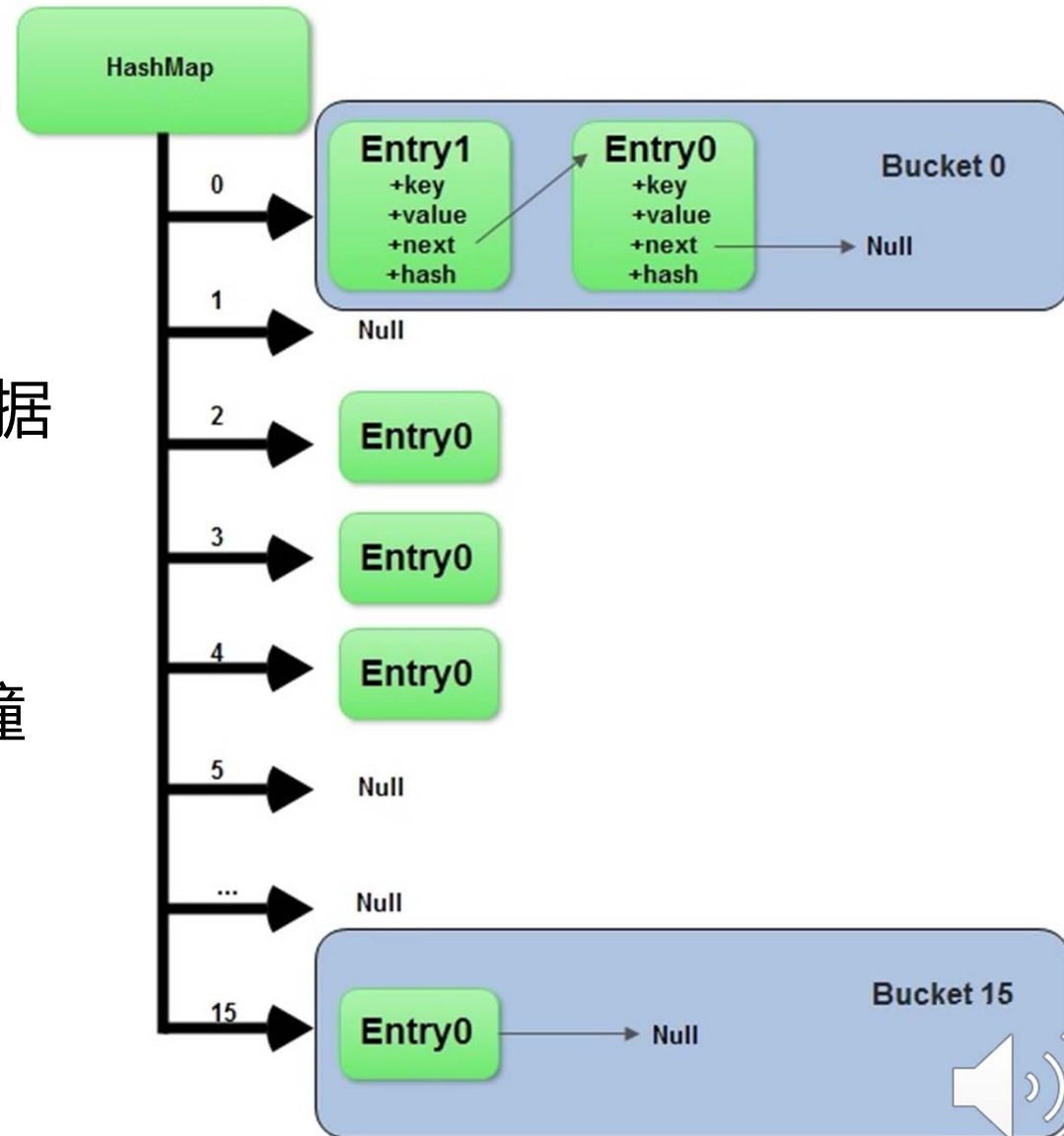
# HashMap的存储

## 1. 什么适合做Key

- a. 简单数据
- b. hashCode>equals根据内容实现

## 1. 如果链表过长?

- a. 对应链表非空→碰撞
- b. 查询会变慢
- c. 填充因子



# TreeMap

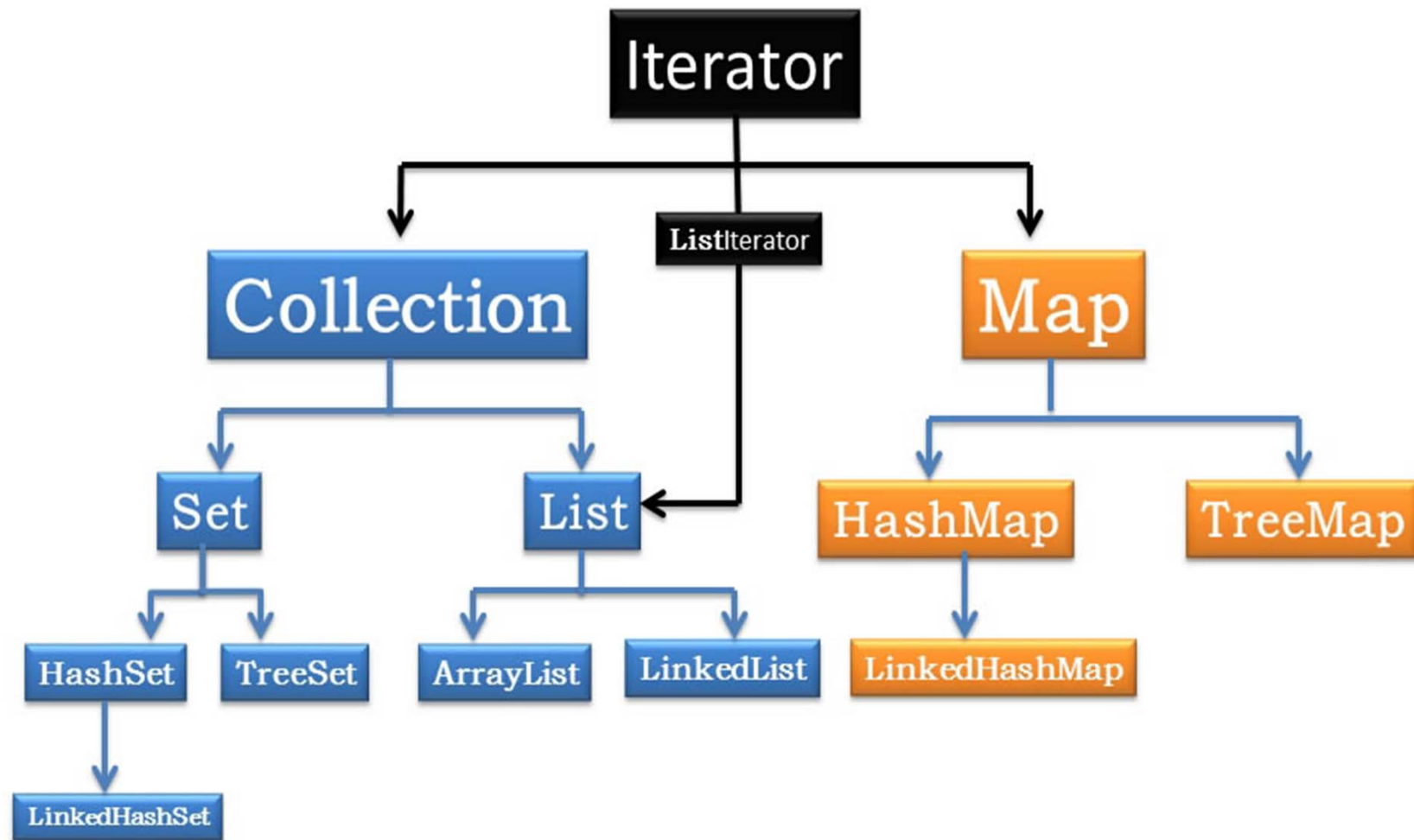
- 根据Key有序存储
  - 基于RB树
- TreeSet使用TreeMap实现

```
Map <Integer, String> map = new TreeMap<Integer, String>();
map.put(101, "Hello1");
map.put(100, "Hello2");
Iterator<Entry<Integer, String>> iterator = map.entrySet().iterator();
while(iterator.hasNext()) {
    Entry<Integer, String> v = iterator.next();
    System.out.print(v.getValue());
}
```

输出: Hello2Hello1



# Java容器(集合)类



# 思考

- ❑ 了解Enumeration，理解其和Iterator接口的区别
- ❑ 理解hashCode()和equals()方法的关系
- ❑ 理解什么类适合作为Map的Key
- ❑ 理解HashMap与TreeMap的使用场景
- ❑ 理解ArrayList和LinkedList的使用场景
- ❑ 了解哪些集合类是线程安全的



复旦大学计算机科学技术学院



# 编程方法与技术

A.5. Java集合类练习（不用交）

周扬帆

2021-2022第一学期



# Java集合类应用练习

get

- 输入：一个纯英文语言的Text文件
- 输出：出现的单词及次数
  - A: 10
  - B: 100
- 输入：两个纯英文语言的Text文件
- 输出：他们各自包含的单词的集合的Jaccard距离

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

$$J_{\delta}(A, B) = 1 - J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

