第9章
并发控制

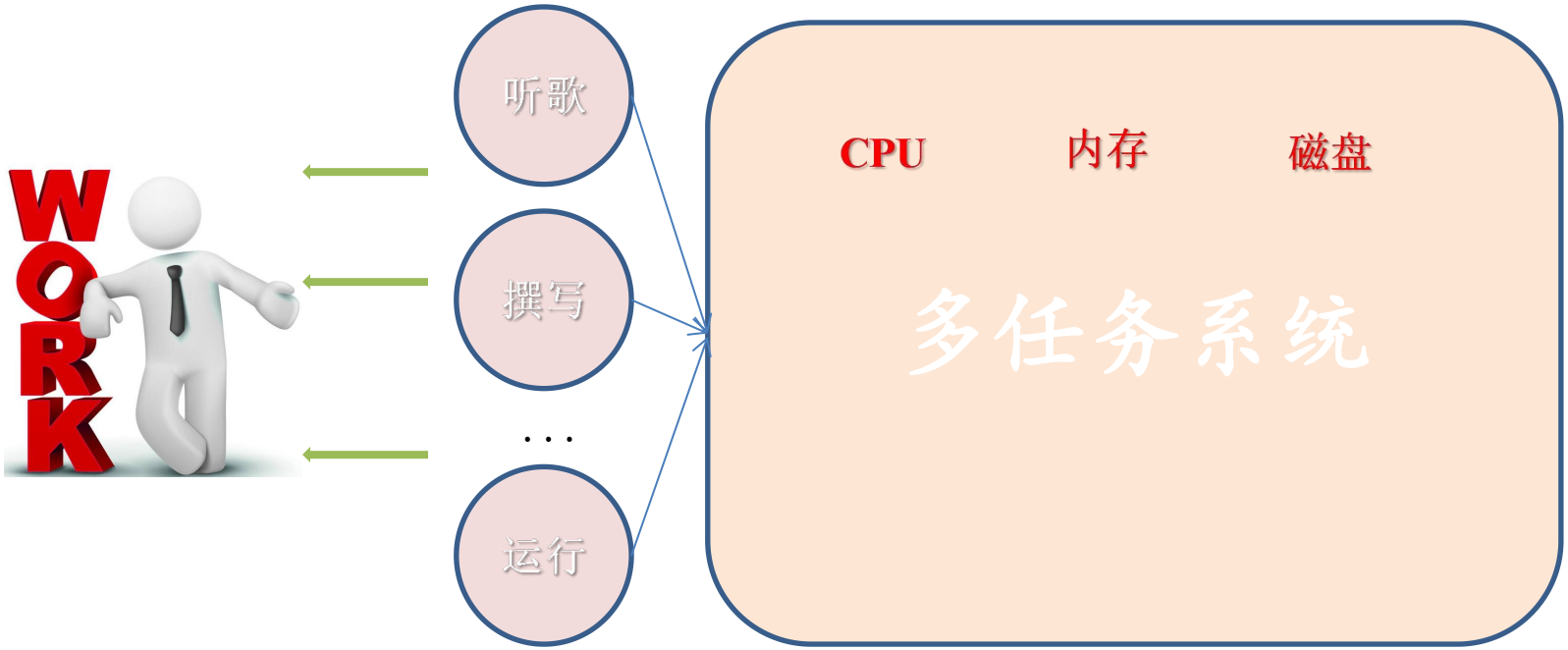
目录

- 基础知识
- 线程创建
- 线程之间的协作
- 扩展锁机制

基础知识

- 多任务系统的发展
- 多任务系统解决的问题
- 进程和线程
- 线程之间共享
- Java线程的特性

基础知识

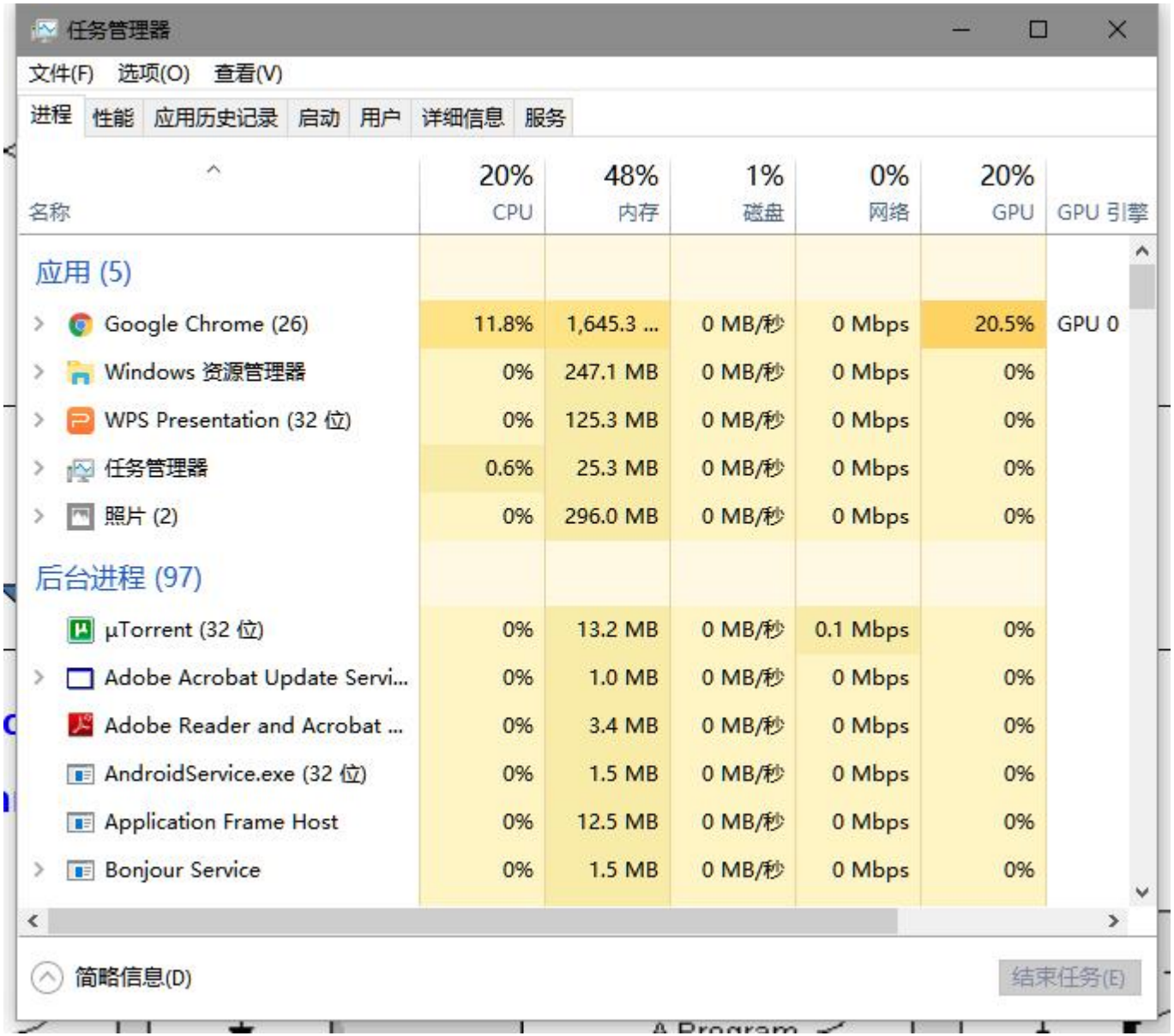


多任务操作系统

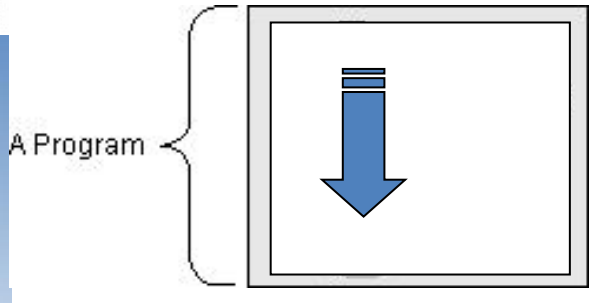
- 如果一个用户在同一时间只能运行一个应用程序，则对应的操作系统称为**单任务操作系统。**
- 如果用户在同一时间可以运行多个应用程序（每个应用程序被称作一个任务），则这样的操作系统被称为**多任务操作系统。**

操作系统（Operating System，简称OS）是管理和控制计算机硬件与软件资源的计算机程序，是直接运行在“裸机”上的最基本的系统软件，任何其他软件都必须在操作系统的支持下才能运行。

多任务操作系统

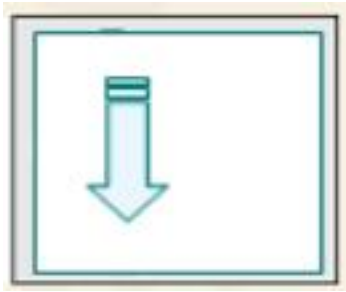

任务管理器
文件(F) 选项(O) 查看(V)
进程 性能 应用历史记录 启动 用户 详细信息 服务
名称 20% CPU 48% 内存 1% 磁盘 0% 网络 20% GPU GPU 引擎
应用 (5)
Google Chrome (26) 11.8% 1,645.3 ... 0 MB/秒 0 Mbps 20.5% GPU 0
Windows 资源管理器 0% 247.1 MB 0 MB/秒 0 Mbps 0%
WPS Presentation (32 位) 0% 125.3 MB 0 MB/秒 0 Mbps 0%
任务管理器 0.6% 25.3 MB 0 MB/秒 0 Mbps 0%
照片 (2) 0% 296.0 MB 0 MB/秒 0 Mbps 0%
后台进程 (97)
µTorrent (32 位) 0% 13.2 MB 0 MB/秒 0.1 Mbps 0%
Adobe Acrobat Update Servi... 0% 1.0 MB 0 MB/秒 0 Mbps 0%
Adobe Reader and Acrobat ... 0% 3.4 MB 0 MB/秒 0 Mbps 0%
AndroidService.exe (32 位) 0% 1.5 MB 0 MB/秒 0 Mbps 0%
Application Frame Host 0% 12.5 MB 0 MB/秒 0 Mbps 0%
Bonjour Service 0% 1.5 MB 0 MB/秒 0 Mbps 0%
简略信息(D)
结束任务(E)

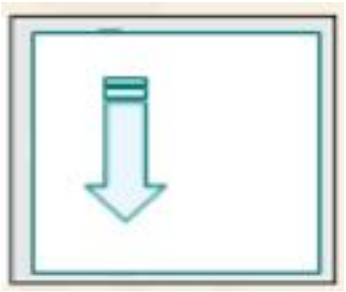
进程与线程

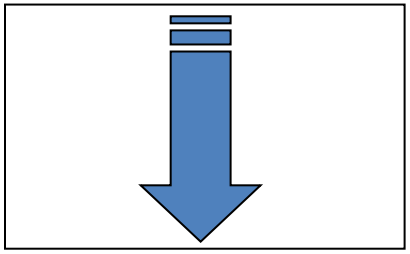


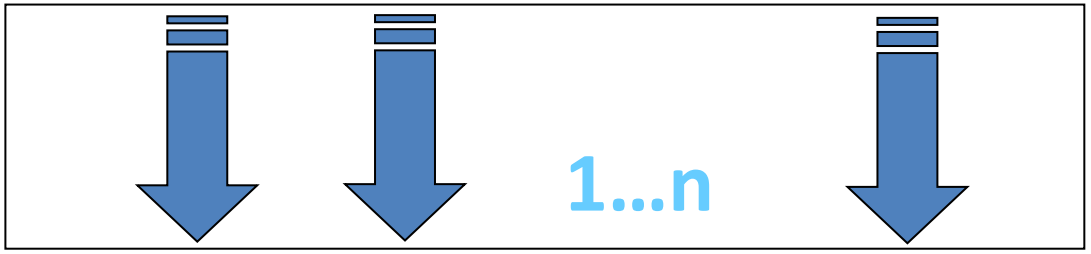
0 … n

processes







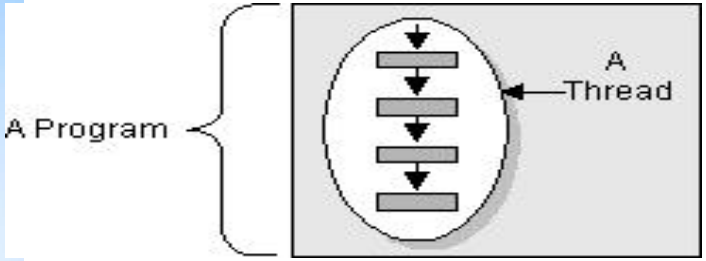


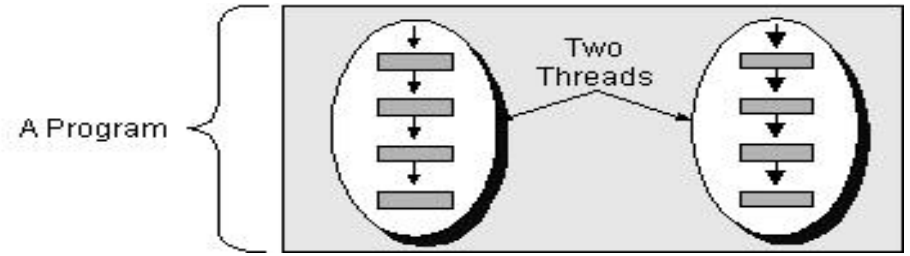
One process

One thread

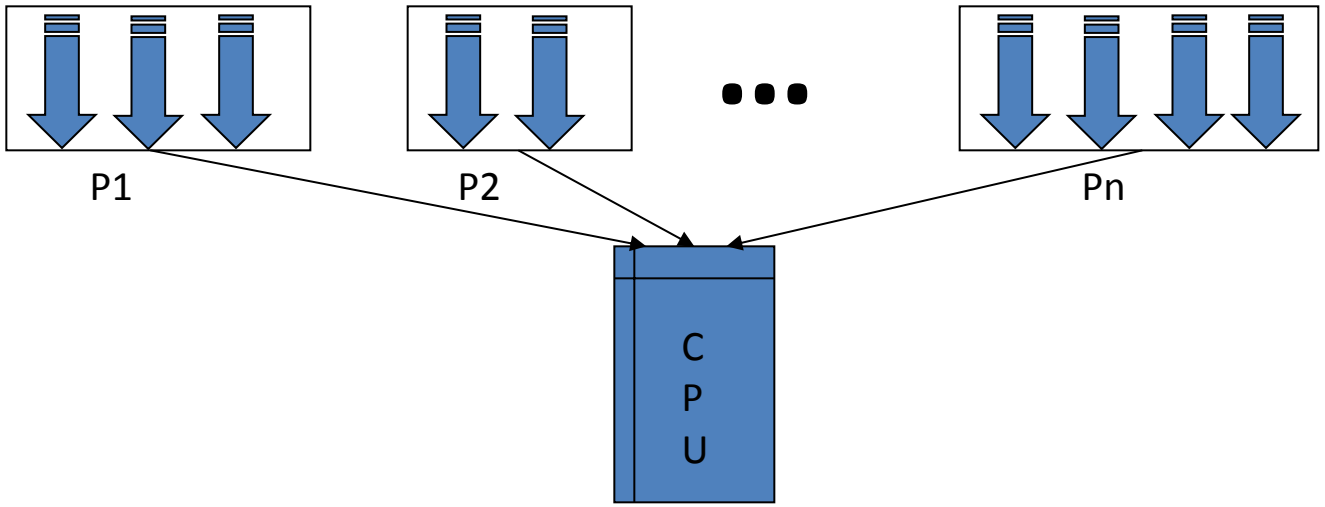
One process

Multiple threads





进程与线程



基础知识

cpu与核心

物理核

- 物理核数量=cpu数(机子上装的cpu的数量)*每个cpu的核心数

虚拟核

- 所谓的4核8线程，4核指的是物理核心。通过超线程技术，用一个物理核模拟两个虚拟核，每个核两个线程，总数为8线程。
- 在操作系统看来是8个核，但是实际上是4个物理核。
- 通过超线程技术可以实现单个物理核实现线程级别的并行计算，但是比不上性能两个物理核。

单核cpu和多核cpu

- 都是一个cpu，不同的是每个cpu上的核心数
- 多核cpu是多个单核cpu的替代方案，多核cpu减小了体积，同时也减少了功耗
- 一个核心只能同时执行一个线程

基础知识

串行，并发与并行

串行

- 多个任务，执行时一个执行完再执行另一个。
- 比喻：吃完饭再看球赛。

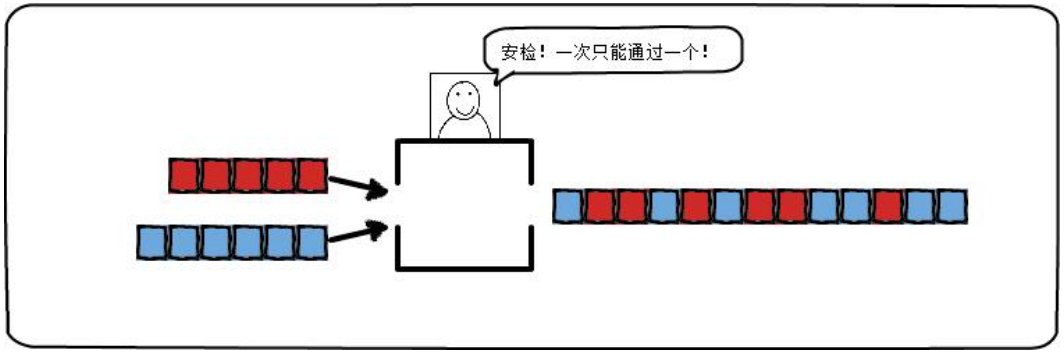
并发

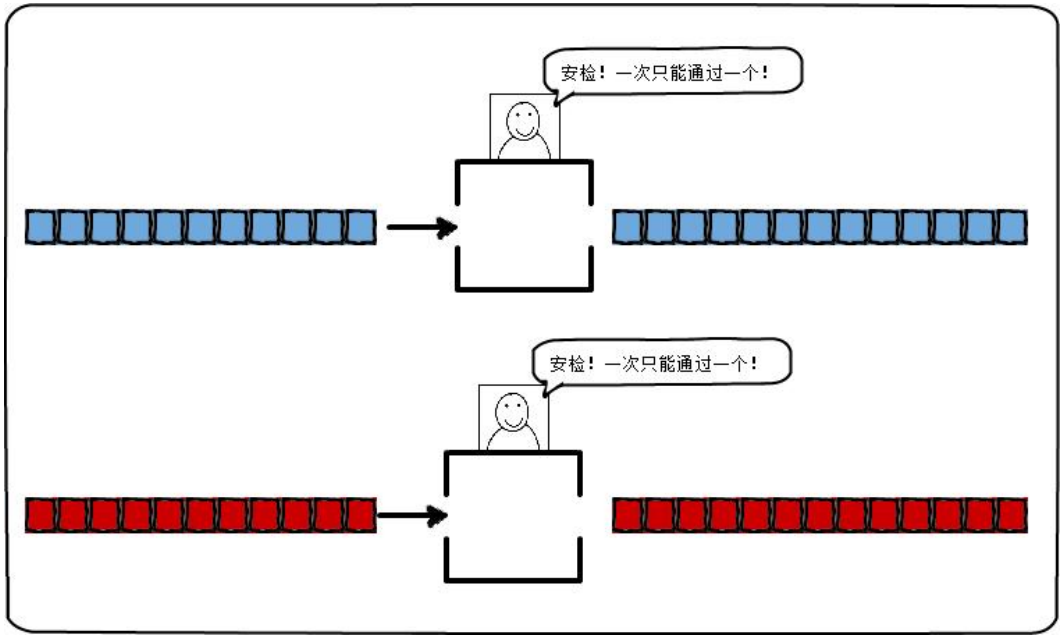
- 多个线程在单个核心运行，同一时间一个线程运行，系统不停切换线程，看起来像同时运行，实际上是线程不停切换。
- 比喻： 一会跑去食厅吃饭，一会跑去客厅看球赛。

并行

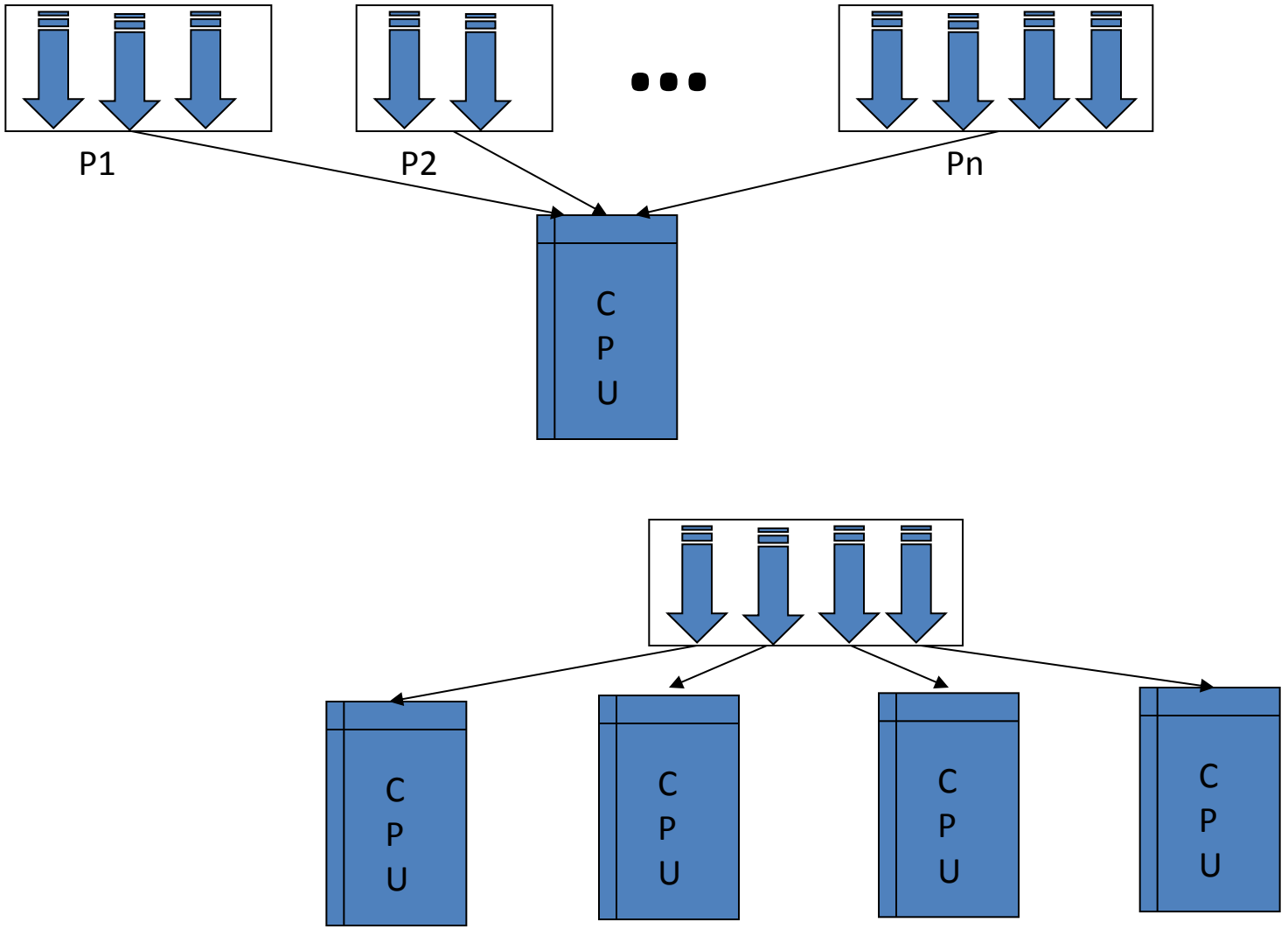
- 每个线程分配给独立的核心，线程同时运行。
- 比喻：一边吃饭一边看球赛。

基础知识





进程与线程

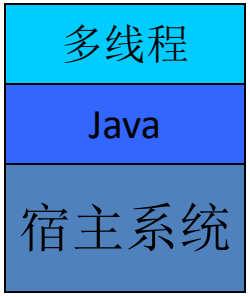


多线程的作用

- 提高UI响应速度
- 提高硬件资源的利用效率
- 隔离高速硬件和低速硬件
- 提供程序上的抽象

JAVA多线程

- Java要求宿主系统支持多线程
- Java在各个宿主系统上使用同样的多线程编程模型
- 具体的实现委派给了宿主系统的多任务管理模块




WORK
昵图网 www.nipic.com
Byrnavyxia No.20181222144252560000


昵图网 www.nipic.com
By:bangaini123 No.20140217180649369305

目录

- 基础知识
- 线程创建
- 线程之间的协作
- 扩展锁机制

线程的创建

Thread类

继承于java.lang.Object

构造方法

public Thread()

public Thread(Runnable target)

常用方法

sleep(long millis)

start()

yeild()

interrupt()

setPriority(in newPriority)

线程的创建

方法一：继承Thread类

```
class PrimeThread extends Thread {
        long minPrime;
        PrimeThread(long minPrime) {
            this.minPrime = minPrime;
        }

        public void run() {
            // compute primes larger than minPrime
             . . .
        }
    }
```

线程的创建

```
1
2  public class TestThread {
3      public static void main(String[] args) {
4          // TODO Auto-generated method stub
5          MyThread t1 = new MyThread("t1");
6          MyThread t2 = new MyThread("t2");
7          MyThread t3 = new MyThread("t3");
8          System.out.println("t1 start");
9          t1.start();
10         System.out.println("t2 start");
11         t2.start();
12         System.out.println("t3 start");
13         t3.start();
14     }
15 }
16 class MyThread extends Thread{
17     String threadName;
18     public MyThread(String threadName){
19         this.threadName = threadName;
20     }
21     public void run(){
22         for(int i = 1;i<10;i++)
23             System.out.println(threadName+"---"+i);
24     }
25 }
```


```
t1  start
t2  start
t1---1
t1---2
t1---3
t2---1
t2---2
t2---3
t2---4
t3  start
t2---5
t1---4
t1---5
t1---6
t1---7
t1---8
t1---9
t2---6
t3---1
t2---7
t2---8
t2---9
t3---2
t3---3
t3---4
t3---5
t3---6
t3---7
t3---8
t3---9
```

```
<terminated> TestThread [.
t1  start
t2  start
t3  start
t1---1
t1---2
t2---1
t2---2
t2---3
t2---4
t1---3
t1---4
t1---5
t1---6
t3---1
t2---5
t2---6
t2---7
t2---8
t2---9
t3---2
t3---3
t1---7
t1---8
t1---9
t3---4
t3---5
t3---6
t3---7
t3---8
t3---9
```

线程的创建

```
1
2  public class TestThread {
3      public static void main(String[] args) {
4          // TODO Auto-generated method stub
5          MyThread t1 = new MyThread("t1");
6          MyThread t2 = new MyThread("t2");
7          MyThread t3 = new MyThread("t3");
8          System.out.println("t1 start");
9          t1.run();//t1.start();
10         System.out.println("t2 start");
11         t2.run();//t2.start();
12         System.out.println("t3 start");
13         t3.run();//t3.start();
14     }
15 }
16 class MyThread extends Thread{
17     String threadName;
18     public MyThread(String threadName){
19         this.threadName = threadName;
20     }
21     public void run(){
22         for(int i = 1;i<10;i++)
23             System.out.println(threadName+"---"+i);
24     }
25 }
```

的创建

```
<terminated> TestThread
t1  start
t1---1
t1---2
t1---3
t1---4
t1---5
t1---6
t1---7
t1---8
t1---9
t2  start
t2---1
t2---2
t2---3
t2---4
t2---5
t2---6
t2---7
t2---8
t2---9
t3  start
t3---1
t3---2
t3---3
t3---4
t3---5
t3---6
t3---7
t3---8
t3---9
```

线程的创建

方法二：使用Runnable接口

```
class PrimeRun implements Runnable {
        long minPrime;
        PrimeRun(long minPrime) {
            this.minPrime = minPrime;
        }

        public void run() {
            // compute primes larger than minPrime
             . . .
        }
    }
```

线程的创建

```
1
2  public class TestRunnable implements Runnable{
3
4      private String threadName;
5      public TestRunnable(String threadName){
6          this.threadName = threadName;
7      }
8      @Override
9      public void run() {
10         // TODO Auto-generated method stub
11         for(int i=0;i<10;i++)
12             System.out.println(threadName+"---"+i);
13     }
14     public static void main(String[] args) {
15         // TODO Auto-generated method stub
16         Runnable t1 = new TestRunnable("t1");
17         Runnable t2 = new TestRunnable("t2");
18         Runnable t3 = new TestRunnable("t3");
19
20         System.out.println("t1 start");
21         new Thread(t1).start();
22         System.out.println("t2 start");
23         new Thread(t2).start();
24         System.out.println("t3 start");
25         new Thread(t3).start();
26     }
27 }
```

线程的创建

```
t1 start
t2 start
t1---0
t1---1
t3 start
t2---0
t1---2
t2---1
t2---2
t2---3
t2---4
t2---5
t1---3
t2---6
t2---7
t3---0
t2---8
t2---9
t1---4
t3---1
t3---2
t3---3
t3---4
t3---5
t3---6
t3---7
t3---8
t3---9
t1---5
t1---6
t1---7
t1---8
t1---9
```

线程的创建

- 目标是获取到一个Thread的实例
- 调用start方法开始线程并分配资源
- 从run方法中返回是退出线程唯一正确的方法。

线程的创建

- Thread类和Runnable接口两种方式
- run（）和start（）的区别
- 线程只能被启动一次
- 方式一与方式二有什么区别？

Thread和Runnable的区别

```
1
2  public class MyThreadTest {
3      public static void main(String[] args) {
4          System.out.println("thread");
5          for(int i = 0;i<10;i++){
6              Thread t = new MyThread();
7              t.start();
8          }
9          System.out.println("runnable");
10         MyRunnable r = new MyRunnable();
11         for(int i = 0;i<10;i++){
12             Thread t = new Thread(r);
13             t.start();
14         }
15     }
16 }
17 class MyThread extends Thread{
18     public int x = 0;
19     public void run(){
20         System.out.println(++x);
21     }
22 }
23 class MyRunnable implements Runnable{
24     public int x = 0;
25     public void run() {
26         System.out.println(++x);
27     }
28 }
```

```
thread
1
1
1
1
1
1
1
1
1
runnable
1
1
2
3
4
5
6
7
8
9
10
```

Thread和Runnable的区别

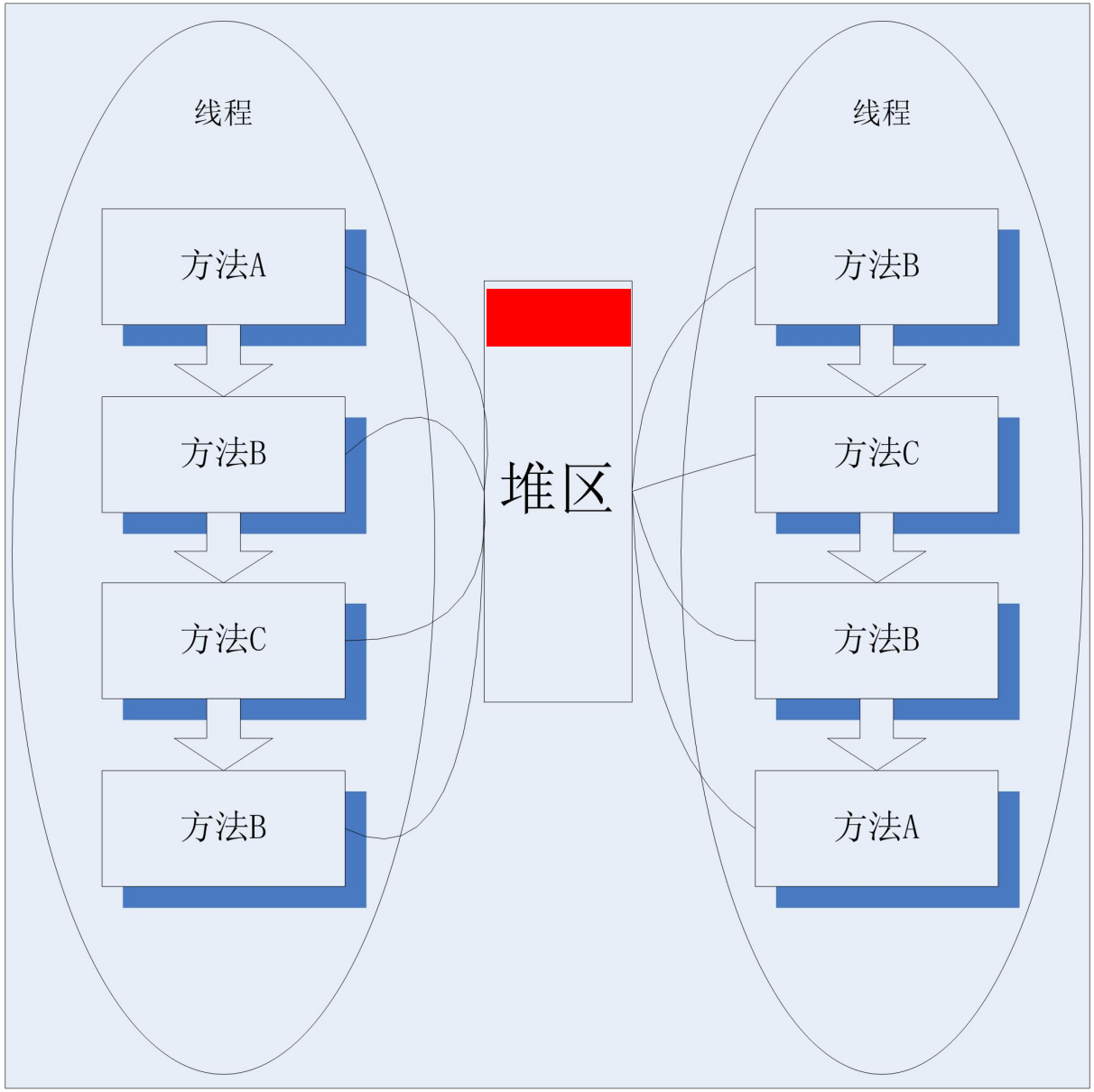
上面10个线程对象产生的10个线程运行时打印了10次1。下面10个线程对象产生的10个线程运行时打印了1到10。我们把下面的10个线程称为同一实例(Runnable实例)的多个线程。

Thread和Runnable的区别

```
2  public class MyThreadTest {
3      public static void main(String[] args) {
4          System.out.println("thread");
5          Thread t1 = new MyThread();
6          Thread t2 = new MyThread();
7          t1.start();
8          t2.start();
9          System.out.println("runnable");
10         MyRunnable r = new MyRunnable();
11         Thread t3 = new Thread(r);
12         Thread t4 = new Thread(r);
13         t3.start();
14         t4.start();
15     }
16 }
17 class MyThread extends Thread{
18     public int x = 0;
19     public void run(){
20         for(int i = 0; i< 10;i++)
21             System.out.println(++x);
22     }
23 }
24 class MyRunnable implements Runnable{
25     public int x = 0;
26     public void run() {
27         for(int i = 0; i< 10;i++)
28             System.out.println(++x);
29     }
30 }
```

```
<terminated> MyThread
thread
1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
10
runnable
1
2
3
4
5
7
8
9
10
11
6
12
13
14
15
16
17
18
19
```

多线程内存模型



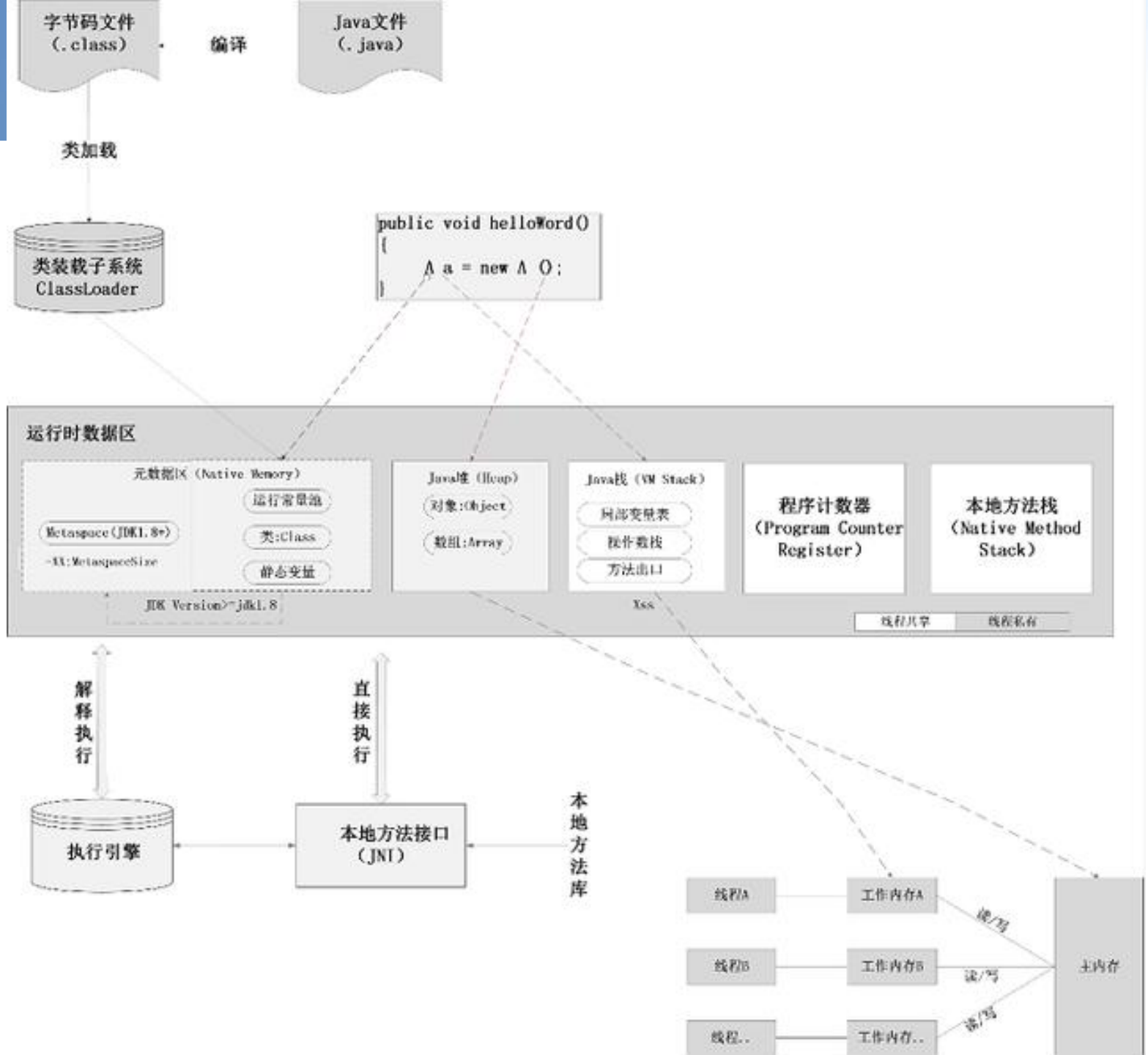

CPU
编号：564509 (图片114网 www.tupian114.com) 永久免费素材网

线程之间的协作

- 共享信息：消息或共享内存
  - 部分更新问题

    一个实例有多个属性，多个线程同时试图更新该实例
  - 读写不一致问题

```
//线程1
int step=1
shareInt=shareInt+step
//线程2
int step=2
shareInt=shareInt+step
```


字节码文件
(.class)
编译
Java文件
(.java)
类加载
类装载子系统
ClassLoader
public void helloWord()
{
A a = new A ();
}
运行时数据区
元数据区 (Native Memory)
运行常量池
Metaspace(JDK1.8+)
类:Class
-XX:MetaspaceSize
静态变量
JDK Version>=jdk1.8
Java堆 (Heap)
对象:Object
数组:Array
Java栈 (VM Stack)
局部变量表
操作数栈
方法出口
Xss
程序计数器
(Program Counter
Register)
本地方法栈
(Native Method
Stack)
线程共享
线程私有
解释执行
直接执行
执行引擎
本地方法接口
(JNI)
本地方法库
线程A
工作内存A
线程B
工作内存B
线程..
工作内存..
读/写
读/写
读/写
主内存

同步区域

- 使用了锁（monitor）对象
- 同一时间只能有一个线程拥有监控器对象
- 编程者需要根据需要设置监控器的设置。

同步区域

◆方法一

```
synchronized ReturnType
        methodName(parameterList){
            method statement;
}
```

◆ 方法二

```
synchronized (object){
        statement;
}
```

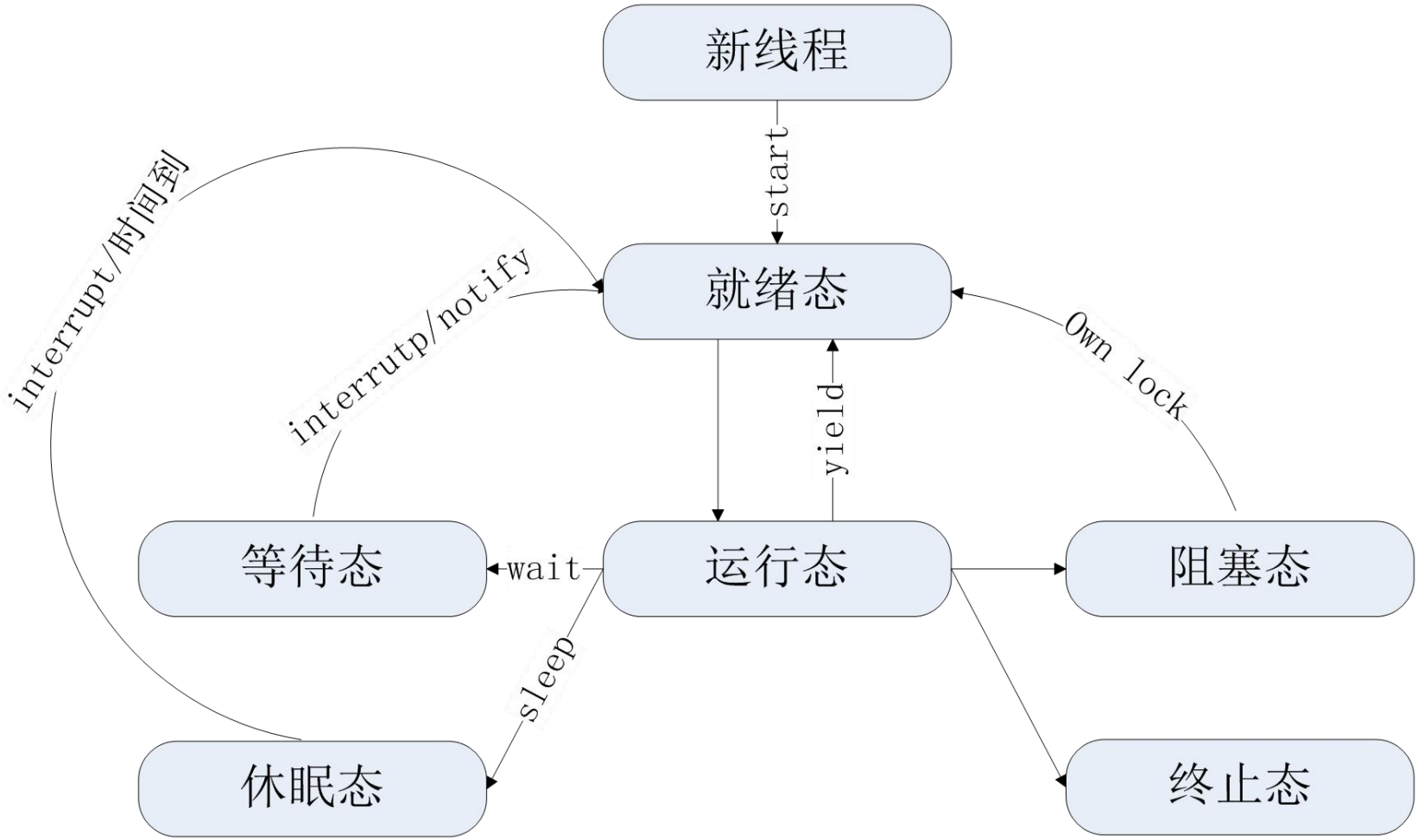
同步区域

```
//线程1
synchronized (lock){
        shareInt = shareInt+1；
}
//线程2
synchronized (lock){
        shareInt = shareInt+1；
}
```

线程的状态

- Born state / New Thread（出生）
- Ready state/Runnable Thread（就绪）
- Running state（运行）
- Sleeping state（休眠）
- Waiting state（等待）
- Blocked state（阻塞）
- Dead state（死亡）

线程的生命周期


新线程
start
就绪态
interrupt/时间到
interrutp/notify
yield
Own lock
等待态
wait
运行态
阻塞态
sleep
休眠态
终止态

线程的生命周期

```
public class LifeCycleThread extends Thread{
    private Object lock = new Object();
    public void run(){
        synchronized(lock){
            try {
                System.out.println("11");
                lock.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        LifeCycleThread thread = new LifeCycleThread();
        thread.start();
    }
```

线程优先级

- Java线程优先级由操作系统实现
- 优先级分为10级

setPriority(int)

- 线程组调度依赖于宿主系统的调度算法,优先级设置的复杂性
- Daemon属性

Daemon线程

线程有一个特殊的属性，属性名为daemon。如果此值为真，当其他非Daemon线程都处于终止态时，整个进程结束。也就是说，daemon线程是后台监控线程，它有可能在任何运行点被终止。

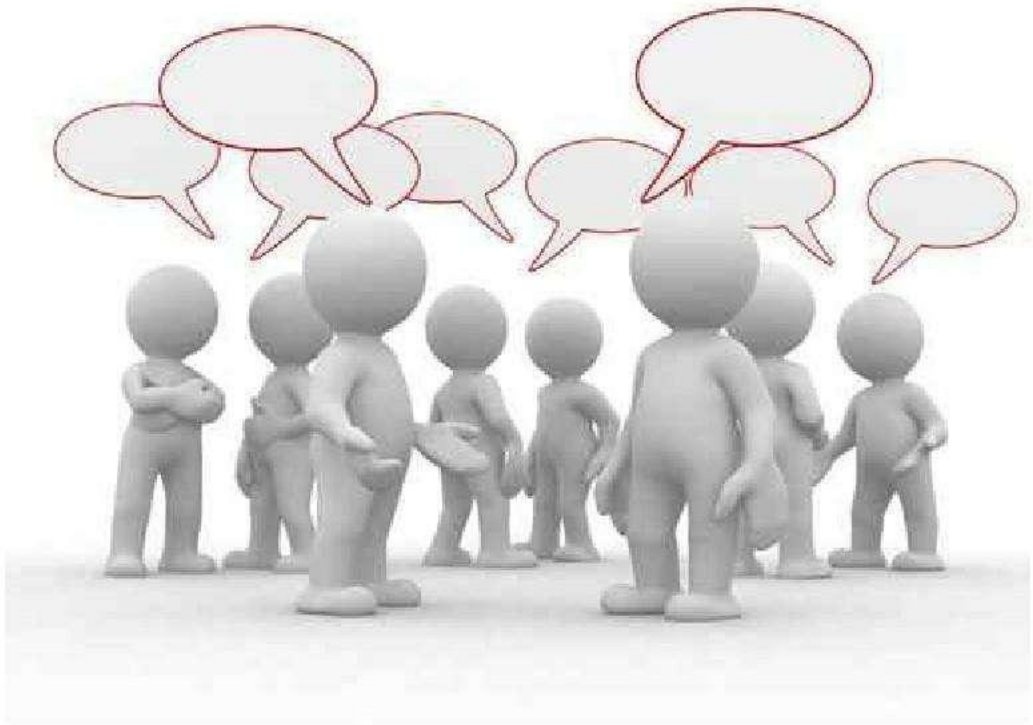
两个相关的方法：

setDaemon

isDaemon

目录

- 基础知识
- 线程创建
- 线程之间的协作
- 扩展锁机制



线程之间的协作

- 共享信息：消息或共享内存
  - 部分更新问题

    一个实例有多个属性，多个线程同时试图更新该实例
  - 读写不一致问题

```
//线程1
int step=1
shareInt=shareInt+step
//线程2
int step=2
shareInt=shareInt+step
```

同步区域

- 使用了锁（monitor）对象
- 同一时间只能有一个线程拥有监控器对象
- 编程者需要根据需要设置监控器的设置。

协作机制

- 多线程程序如何协作完成任务？
  - 在共享内存中设置标记
- Java提供了一种通知机制
  - wait方法簇
    - ◇在一个引用对象上等待接收通知
  - notify方法簇
    - ◇通知等待的线程事件已经发生，选择一个等待线程

3.4 Java标准类库

◆Object类

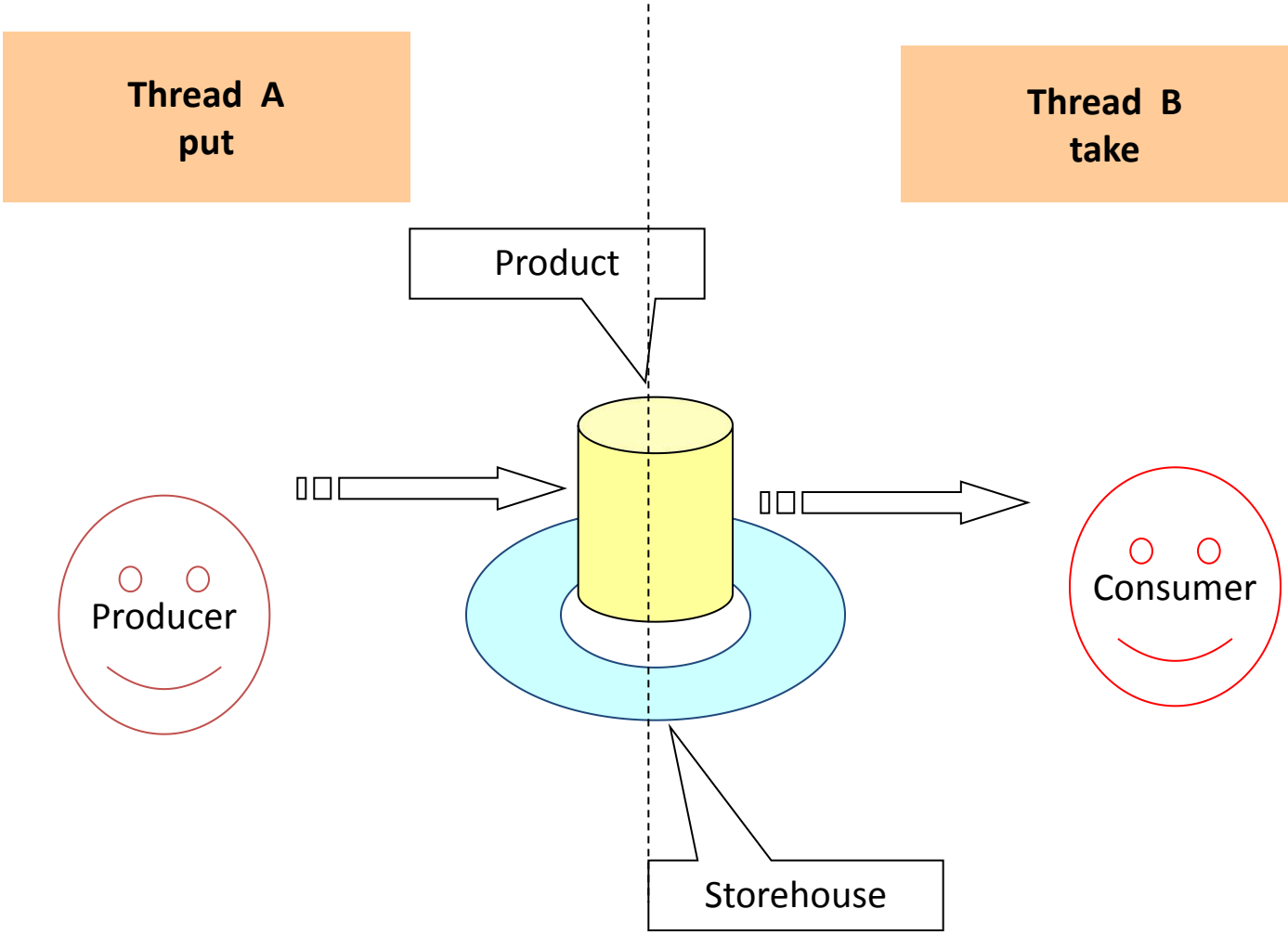
```
public class java.lang.Object{
   public Object();                                    //构造方法
   protected Object clone();                           //建立当前对象的拷贝
   public boolean equals(Object obj);                  //比较对象
   protected void finalize();                          //释放资源
   public final Class getClass();                      //求对象对应的类
   public int hashCode();                              //求hash码值
   public final void notify();                         //唤醒当前线程
   public final void notifyAll();                      //唤醒所有线程
   public String toString();                           //返回当前对象的字符串
   public final void wait();                           //使线程等待
   public final void wait(long timeout);
   public final void wait(long timeout, int nanos);
// 其中timeout为最长等待时间，单位为毫秒；nanos为附加时间，单位为纳秒，
// 取值范围为0～999999。
}
```

协作机制

- 调用wait方法后进入等待态（获得锁后可调用）
- 释放监控器的锁，暂停运行
- 条件满足后，恢复运行
- 申请监控器的锁，如不能，进入阻塞态。
- 获取锁，运行wait方法后续代码

协作机制

Producers and consumers



同步方法

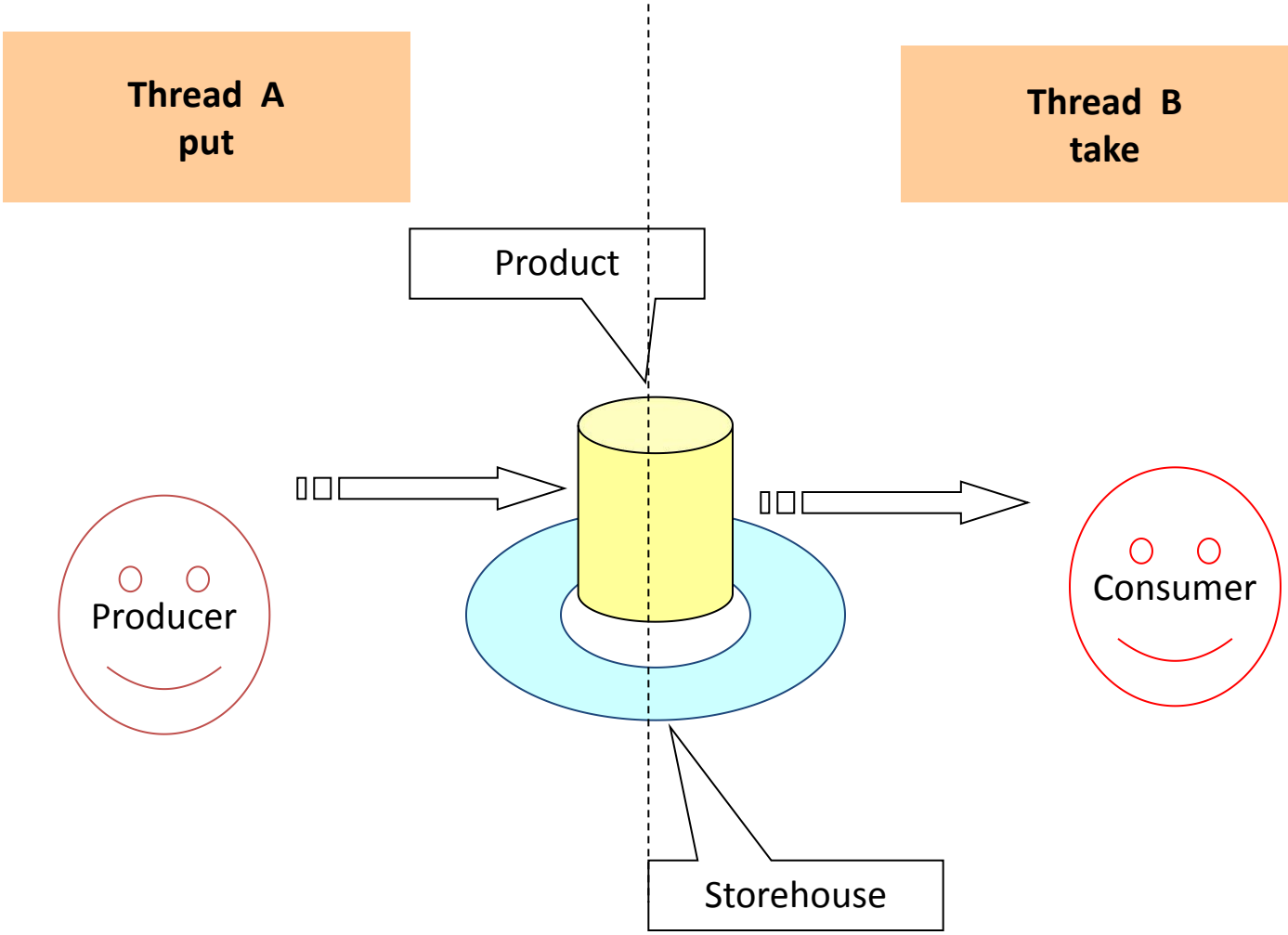
```
Object lock = new Object();
Object emptyLock = new Object();
public void put(Object x) throws InterruptedException {
    synchronized (lock) {
        while (count == items.length)
            lock.wait();    //等待take线程取走数据
        items[putptr] = x;
        if (++putptr == items.length)
                putptr = 0;
        ++count;
        synchronized (emptyLock) {
                emptyLock.notify(); //通知take线程
        }}}
```

同步方法

```
public Object take() throws InterruptedException {
   synchronized (lock) {
      while (count == 0){
         synchronized (emptyLock) {  //如果为空，等待
            emptyLock.wait();
         }
      }
      Object x = items[takeptr];
      if (++takeptr == items.length)
           takeptr = 0;
      --count;
      lock.notify();  //通知put线程
      return x;          }
}
```

协作机制

Producers and consumers



目录

- 介绍
- 基本概念
- 线程之间的协作
- 扩展锁机制

现有机制的缺陷

- 这个锁队列中获取到锁的顺序是不确定的。
- 同时苏醒算法不是公平的
- 查询不到当前的锁的状态。
- 锁的申请和释放有严格的顺序。
- 每个锁只能有一个通知事件。

Lock&Condition

- Lock对应synchronized
  - java.util.concurrent.locks.Lock
  - ReentrantLock
- Condition上可以调用类wait/notify方法
  - java.util.concurrent.locks.Condition
  - AbstractQueuedSynchronizer.ConditionObject

ReentrantLock

```
class SimpleLock {
   private final ReentrantLock lock =
new ReentrantLock();                    //①
   public void doSth() {
     lock.lock();                       //②
     try {
       // statement
     } finally {                        //③
       lock.unlock()                    //④
     }  }
 }
```

如何同步

```
class ExtendBoundedBuffer
    final Lock lock =new ReentrantLock();
    final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;
```

如何同步

```
public void put(Object x)  {
    lock.lock();
    try {
        while (count == items.length)
            notFull.await();
        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
            ++count;
        notEmpty.signal();
    } finally {
        lock.unlock();
}}
```

如何同步

```
public Object take() {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length)
                takeptr = 0;
        --count;
        notFull.signal();
        return x; } finally {
            lock.unlock();} }
```

线程同步注意事项

- 有几个问题是在编写线程同步程序时应该注意的：

(1)当对象的锁被释放时，阻塞线程调用一个用“synchronized”关键字说明的方法并不保证一定就能立刻成为下一个获得锁的线程。

(2)调用了监控器的wait()方法成为等待态的线程经由其它线程调用notify()方法之后并不保证一定会脱离等待态。

(3)在同步方法中建议用notifyAll()方法唤醒所有等待态线程，包括该线程自身，而将同步线程的控制选择权交由标记变量控制。

线程同步注意事项

⑷使用

while(!条件)……wait()……notify()

结构来完成同步方法的定义，这样做要比使用

if(!条件)……wait()……notify()

结构来完成定义要安全。

⑸不要在线程同步的程序中调用sleep()方法，这样做通常是错误的。

⑹wait()方法通常要抛出中断异常InterruptedException，所以在wait()方法外部要进行捕获和处理异常的操作。

线程同步注意事项

举例说明

while(x < 100){ wait(); } //而不是用if,为什么呢?

go();

在多个线程同时执行时,if(x <100)是不安全的(只检查了一次).因为如果线程A和线程B都在休息室中等待,这时另一个线程使x==100了,并调用notifyAll方法,线程A继续 执行下面的go().而它执行完成后,x有可能又小于100,比如下面的程序中调用了--x,这时切换到线程B,线程B没有继续判断,直接执行go(); 就产生一个错误的条件,只有while才能保证线程B又继续检查一次.

死锁

- 死锁的概念：after you after you...
- 避免死锁的几个设计原则
  - 逻辑隔离：每个逻辑上独立的同步区域都使用各自的锁对象。
  - 尽快释放：使用完毕后尽快释放锁对向。
  - 按顺序申请：对需要使用多个锁对象的区域，所有申请都按固定顺序。

哲学家进餐问题

死锁：如何解决？

四个条件同时发生时会死锁：
1）互斥条件（在一段时间内某资源仅为一任务所占用）
2）必须有一个任务他持有一个资源，并且等待一个被其他任务持有的资源
3）资源不能被任务抢占
4）必须有循环等待

破坏任意一个条件，即可解开死锁。

小结

- 多任务系统的历史
- 进程和线程
- 堆区和栈区的性质
- 多线程之间的访问冲突
- 线程如何协作
- Lock和Condition

```
//: concurrency/Chopstick.java
// Chopsticks for dining philosophers.

public class Chopstick {
  private boolean taken = false;
  public synchronized
  void take() throws InterruptedException {
    while(taken)
      wait();
    taken = true;
  }
  public synchronized void drop() {
    taken = false;
    notifyAll();
  }
} ///:~
```

```
//: concurrency/Philosopher.java
// A dining philosopher
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Philosopher implements Runnable {
  private Chopstick left;
  private Chopstick right;
  private final int id;
  private final int ponderFactor;
  private Random rand = new Random(47);
  private void pause() throws InterruptedException {
    if(ponderFactor == 0) return;
    TimeUnit.MILLISECONDS.sleep(
      rand.nextInt(ponderFactor * 250));
  }
```

```
  public Philosopher(Chopstick left, Chopstick right,
    int ident, int ponder) {
    this.left = left;
    this.right = right;
    id = ident;
    ponderFactor = ponder;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        print(this + " " + "thinking");
        pause();
        // Philosopher becomes hungry
        print(this + " " + "grabbing right");
        right.take();
        print(this + " " + "grabbing left");
        left.take();
        print(this + " " + "eating");
        pause();
        right.drop();
        left.drop();
      }
    } catch(InterruptedException e) {
      print(this + " " + "exiting via interrupt");
    }
  }
  public String toString() { return "Philosopher " + id; }
} ///:~
```

```
//: concurrency/DeadlockingDiningPhilosophers.java
// Demonstrates how deadlock can be hidden in a program.
// {Args: 0 5 timeout}
import java.util.concurrent.*;

public class DeadlockingDiningPhilosophers {
  public static void main(String[] args) throws Exception {
    int ponder = 5;
    if(args.length > 0)
      ponder = Integer.parseInt(args[0]);
    int size = 5;
    if(args.length > 1)
      size = Integer.parseInt(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Chopstick[] sticks = new Chopstick[size];
    for(int i = 0; i < size; i++)
      sticks[i] = new Chopstick();
    for(int i = 0; i < size; i++)
      exec.execute(new Philosopher(
        sticks[i], sticks[(i+1) % size], i, ponder));
    if(args.length == 3 && args[2].equals("timeout"))
      TimeUnit.SECONDS.sleep(5);
    else {
      System.out.println("Press 'Enter' to quit");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```

```
      size = Integer.parseInt(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Chopstick[] sticks = new Chopstick[size];
    for(int i = 0; i < size; i++)
      sticks[i] = new Chopstick();
    for(int i = 0; i < size; i++)
      if(i < (size-1))
        exec.execute(new Philosopher(
          sticks[i], sticks[i+1], i, ponder));
      else
        exec.execute(new Philosopher(
          sticks[0], sticks[i], i, ponder));
    if(args.length == 3 && args[2].equals("timeout"))
      TimeUnit.SECONDS.sleep(5);
    else {
      System.out.println("Press 'Enter' to quit");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```