@[toc]

【JavaSE】多线程与并发编程（总结）

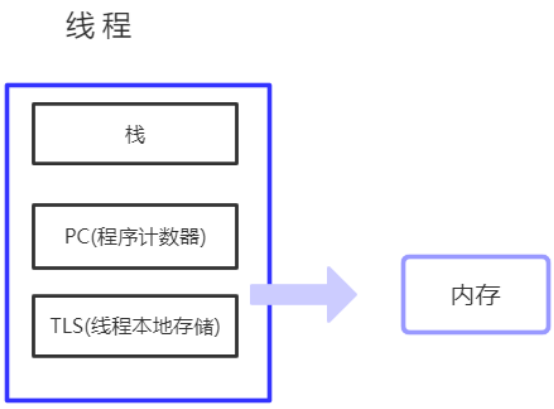
一、多线程概述

参考：https://www.cnblogs.com/zsql/p/11144688.html

进程与线程

进程与线程的区别

1. 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位
2. 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线
3. 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间 (包括代码段，数据集，堆等) 及一些进程级的资源(如打开文件和信号等)，某进程内的线程在其他进程不可见
4. 调度和切换：线程上下文切换比进程上下文切换要快得多



注：我们通常所理解的内存是我们所见到的(2G/4G/8G/16G)物理内存,它为什么会在进程之中呢?
实际上，这里的内存是逻辑内存。指的是内存的寻址空间。每个进程的内存是相互独立的。
否则的话会出现一个问题：我们把指针的值改一改就指向其他进程的内存了

线程(栈+PC+TLS)

- 栈:用于存储该线程的局部变量，这些局部变量是该线程私有的，除此之外还用来存放线程的调用栈祯。
  我们通常都是说调用堆栈，其实这里的堆是没有含义的，调用堆栈就是调用栈的意思。
  那么我们的栈里面有什么呢?
  我们从主线程的入口main函数，会不断的进行函数调用，
  每次调用的时候，会把所有的参数和返回地址压入到栈中。

  > 栈中存放的是栈帧。每调用一个方法就会压入一个新的栈帧。

- PC：是一块内存区域，用来记录线程当前要执行的指令地址 。
  Program Counter 程序计数器，操作系统真正运行的是一个个的线程，
  而我们的进程只是它的一个容器。PC就是指向当前的指令，而这个指令是放在内存中。
  每个线程都有一串自己的指针，去指向自己当前所在内存的指针。
  计算机绝大部分是存储程序性的，说的就是我们的数据和程序是存储在同一片内存里的
  这个内存中既有我们的数据变量又有我们的程序。所以我们的PC指针就是指向我们的内存的。

  > PC计数器存放该线程下一条要执行的指令的地址。用来在执行引擎进行线程切换时，我下一步要从哪里继续执行。

- 缓冲区溢出
  例如我们经常听到一个漏洞：缓冲区溢出
  这是什么意思呢?
  例如：我们有个地方要输入用户名，本来是用来存数据的地方。
  然后黑客把数据输入的特别长。这个长度超出了我们给数据存储的内存区，这时候跑到了
  我们给程序分配的一部分内存中。黑客就可以通过这种办法将他所要运行的代码
  写入到用户名框中，来植入进来。我们的解决方法就是，用用户名的长度来限制不要超过
  用户名的缓冲区的大小来解决。

- TLS:
  全称：thread local storage
  之前我们看到每个进程都有自己独立的内存，这时候我们想，我们的线程有没有一块独立的内存呢?答案是有的，就是TLS。
  可以用来存储我们线程所独有的数据。
  可以看到：线程才是我们操作系统所真正去运行的，而进程呢，则是像容器一样他把需要的一些东西放在了一起，而把不需要的东西做了一层隔离，进行隔离开来。

并行与并发

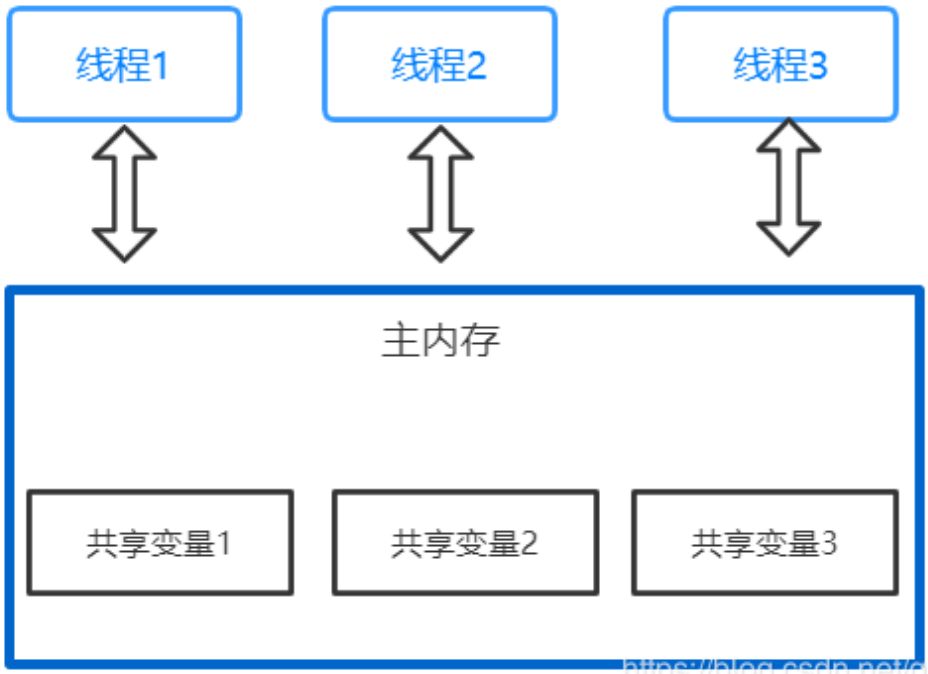
并发：是指`同一个时间段`内多个任务同时都在执行，并且都没有执行结束。并发任务强调在一个时间段内同时执行，而一个时间段由多个单位时间累积而成，所以说并发的多个任务在单位时间内不一定同时在执行 。

并行：`同一时刻`上多个任务同时在执行 。

在多线程编程实践中，线程的个数往往多于CPU的个数，所以一般都称多线程并发编程而不是多线程并行编程。

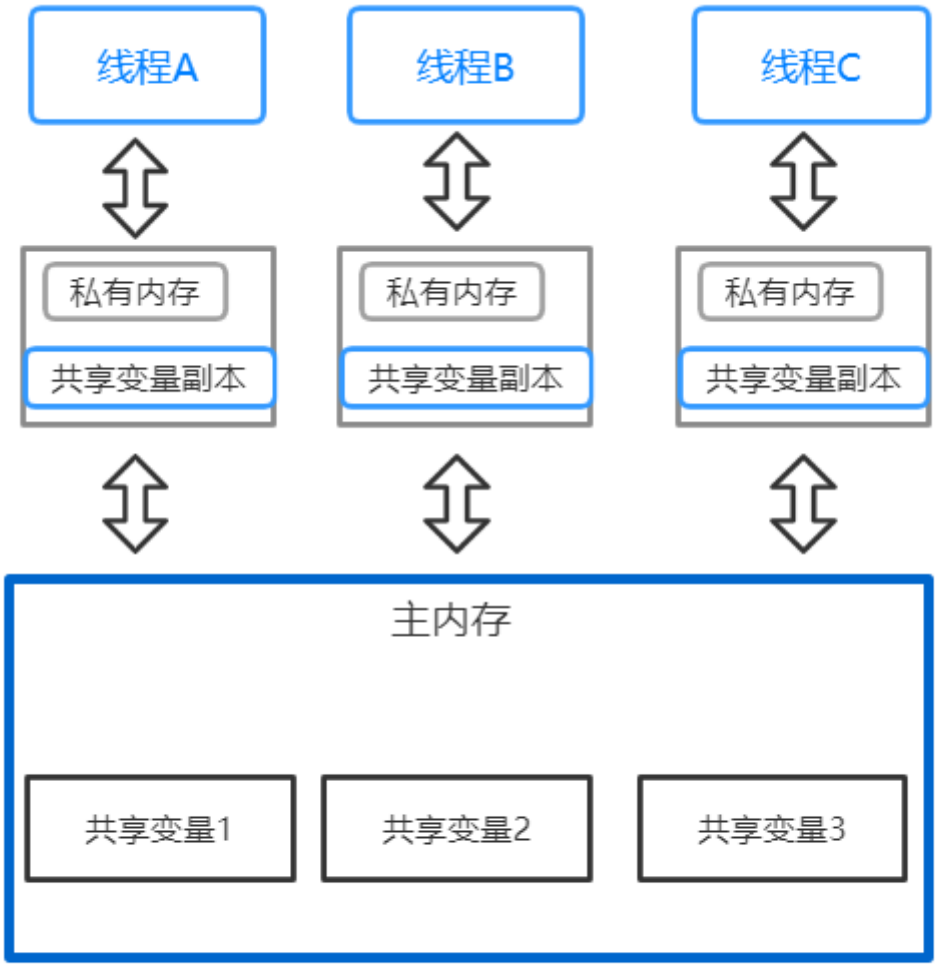
线程安全问题

多个线程同时操作`共享变量1`时，会出现线程1更新共享变量1的值，但是其他线程获取到的是共享变量1没有被更新之前的值。就会导致数据不准确问题。（读到脏数据）

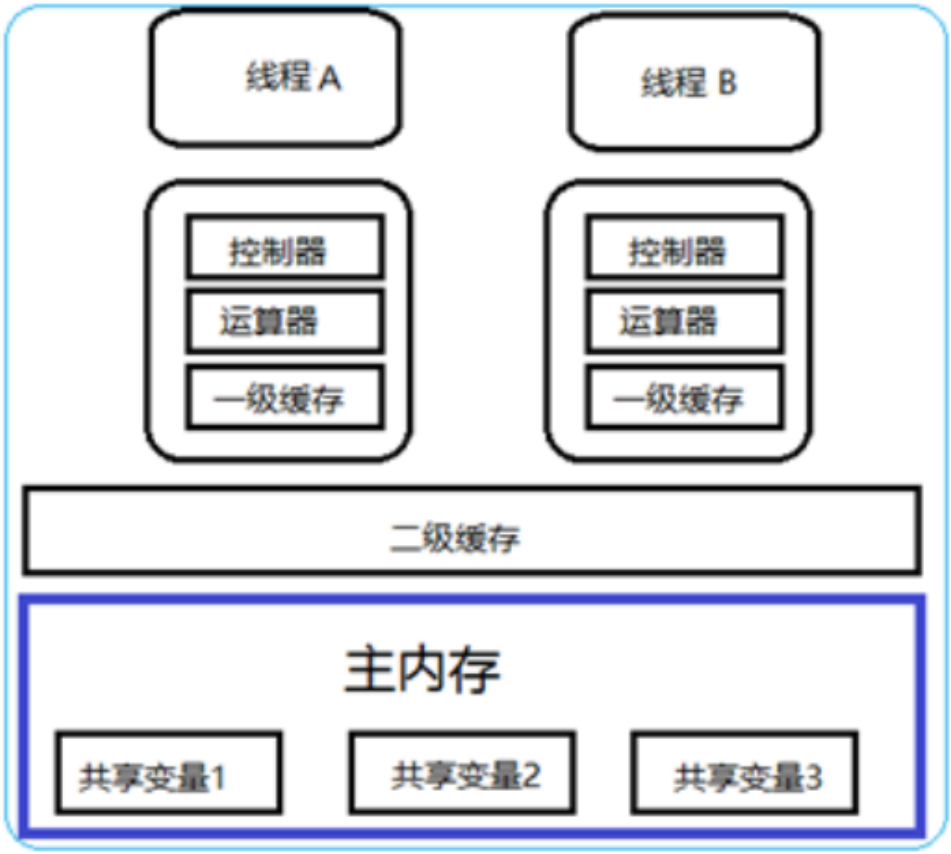


共享内存不可见性问题

1. Java 内存模型规定，将所有的变量都存放在主内存中，当线程使用变量时，会把主内存里面的变量复制到自己的工作空间或者叫作工作内存，线程读写变量时操作的是自己工作内存中的变量 。（如下图所示）



Java内存模型（处理共享变量）



(实际工作的java内存模型)

上图中所示是一个双核 CPU 系统架构，每个核有自己的控制器和运算器，其中控制器包含一组寄存器和操作控制器，运算器执行算术逻辅运算。CPU的每个核都有自己的一级缓存，在有些架构里面还有一个所有CPU都共享的二级缓存。 那么Java内存模型里面的工作内存，就对应这里的 LI或者 L2 缓存或者 CPU 的寄存器

1. 线程A首先获取共享变量X的值，由于两级Cache都没有命中，所以加载主内存中X的值，假如为0。然后把X=0的值缓存到两级缓存，线程A修改X的值为1,然后将其写入两级Cache，并且刷新到主内存。线程A操作完毕后，线程A所在的CPU的两级Cache内和主内存里面的X的值都是1。
2. 线程B获取X的值，首先一级缓存没有命中，然后看二级缓存，二级缓存命中了，所以返回X=1；到这里一切都是正常的，因为这时候主内存中也是X=I。然后线程B修改X的值为2，并将其存放到线程2所在的一级Cache和共享二级Cache中，最后更新主内存中X的值为2，到这里一切都是好的。
3. 线程A这次又需要修改X的值，获取时一级缓存命中，并且X=I这里问题就出现了，明明线程B已经把X的值修改为2，为何线程A获取的还是I呢？这就是共享变量的内存不可见问题，也就是线程B写入的值对线程A不可见。

二、实现多线程

方式1：继承Thread类

需求：我们要实现多线程的程序。 如何实现呢?

> 由于线程是依赖进程而存在的，所以我们应该先创建一个进程出来。
> 而进程是由系统创建的，所以我们应该去**调用系统功能创建一个进程。**

> Java是不能直接调用系统功能的，所以，我们没有办法直接实现多线程程序。 但是呢？ Java可以去调用C / C++写好的程序来实现多线程程序。
>
> 由C/C++去调用系统功能创建进程，然后由Java去调用这样的API， 然后提供一些类供我们使用。我们就可以实现多线程程序了。

> 那么Java提供的类是什么呢?

- **Thread类**
  步骤
- A : 自定义类`MyThread`继承`Thread`
- B : 重写run () 方法
  - 为什么是run ()方法呢?
- C :创建对象
- D : 启动线程

> 注意：单独调用 run() 方法其实和调用一个类的普通方法是没有区别的，想要开启线程，应该让JVM帮我们去进行系统调用
>
> 要想启动线程实际上应该调用的是 start() 方法，这是为什么呢?

---

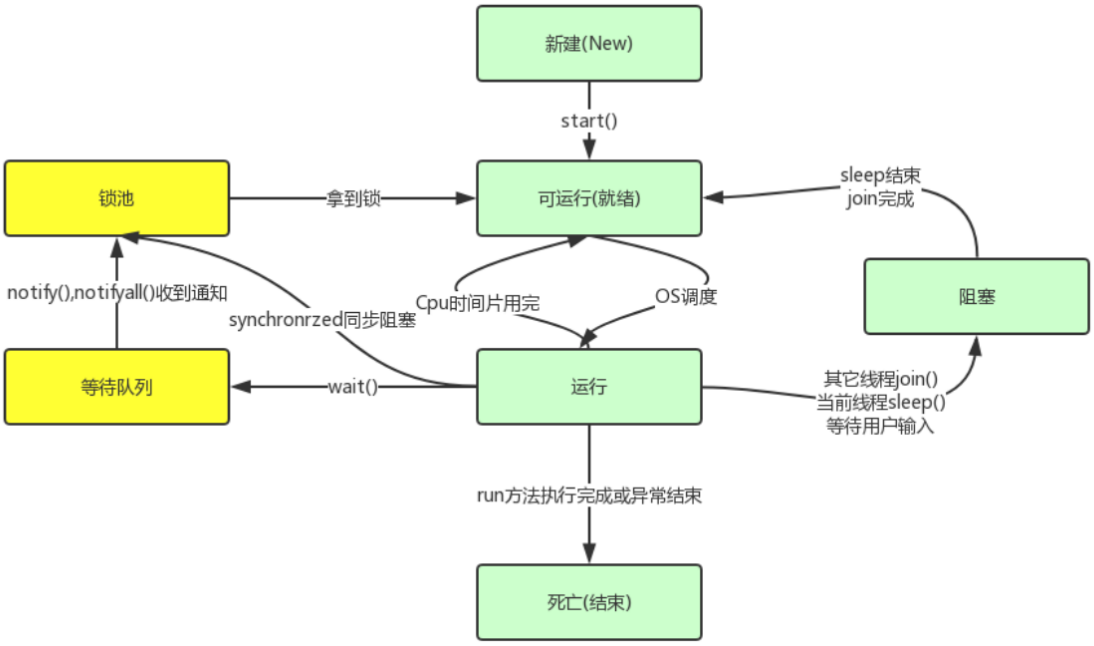
start() 和 run() 的区别?

- **start()** :
  使该线程开始执行；Java 虚拟机调用该线程的 run() 方法。
  用start方法来启动线程，真正实现了多线程运行，这时无需等待run方法体中的代码执行完毕而直接继续执行后续的代码。通过调用Thread类的 start()方法来启动一个线程，这时此线程处于就绪（可运行）状态，并没有运行，一旦得到cpu时间片，就开始执行run()方法，这里的run()方法 称为线程体，它包含了要执行的这个线程的内容，Run方法运行结束，此线程随即终止。

- **run()** :
  如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，`该方法不执行任何操作并返回`。
  Thread 的子类应该重写该方法。
  run()方法只是类的一个普通方法而已，如果直接调用Run方法，程序中依然只有主线程这一个线程，其程序执行路径还是只有一条，还是要`顺序执行`，还是要等待run方法体执行完毕后才可继续执行下面的代码，这样就没有达到写线程的目的。

总结：

void run()	在线程运行启动后，此方法将被调用执行
void start()	使该线程开始执行；Java 虚拟机调用该线程的 run 方法。



两个小问题

- **为什么要重写 run()方法?**
  - 因为run()是用来封装被线程执行的代码
- **run() 方法和 start() 方法的区别?**
  - run()：封装线程执行的代码，直接调用，相当于普通方法的调用
  - start()：启动线程；然后由`JVM调用`此线程的run()方法

创建自定义线程代码：

```
//定义一个线程类
public class MyThread extends Thread {
  @Override
  public void run() {
    for(int i=0; i<100; i++) {
      System.out.println(i);
   }
 }
}

//调用方法执行线程
public class MyThreadDemo {
  public static void main(String[] args) {
    MyThread my1 = new MyThread();
    MyThread my2 = new MyThread();
//    my1.run();
//    my2.run();
    //void start() 导致此线程开始执行; Java虚拟机调用此线程的run方法
    my1.start();

```

匿名内部类的方式创建一个线程

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("多线程");
    }
}).start();
```

获取和设置线程对象名称

void setName(String name)	将此线程的名称更改为等于参数name
String getName()	返回此线程的名称
Thread currentThread()	返回对当前正在执行的线程对象的引用


```
//void setName(String name)：将此线程的名称更改为等于参数 name
    my1.setName("高铁");
    my2.setName("飞机");



    //Thread(String name)
    MyThread my1 = new MyThread("高铁");
    MyThread my2 = new MyThread("飞机");


    System.out.println(Thread.currentThread().getName());


```

线程优先级

Java 使用的是抢占式调度模型

随机性

final int getPriority()	返回此线程的优先级
final void setPriority(int newPriority)	更改此线程的优先级 线程默认优先级是5；线程优先级的范围 是：1-10

线程控制

static void sleep(long millis)	使当前正在执行的线程停留（暂停执行）指定的毫秒数
void join()	等待这个线程死亡
void setDaemon(boolean on)	将此线程标记为**守护线程**，当运行的线程都是守护线程时，Java虚拟机 将退出

方法名	说明
void wait()	**阻塞** 导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法
void notify()	**唤醒阻塞队列的单个线程** **唤醒**正在等待对象监视器的单个线程
void notifyAll()	**唤醒阻塞队列的所有线程** 唤醒正在等待对象监视器的所有线程

jion（）等待线程死亡

```
public class ThreadTest {
    public static void main(String[] args)  {
        ThreadDemo th1=new ThreadDemo();
        ThreadDemo th2=new ThreadDemo();
        ThreadDemo th3=new ThreadDemo();

        th1.setName("康熙");
        th2.setName("四阿哥");
        th3.setName("八阿哥");

        th1.start();
        try {
            th1.join();//等待th1线程的死亡
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //下面的两个线程会等待康熙线程死亡以后才开始执行
        th2.start();
        th2.start();
    }
}
```

守护线程

```
th1.setName("刘备");
th2.setName("张飞");
th3.setName("关羽");

th2.setDaemon(true);
th3.setDaemon(true);

th2.start();
th3.start();
th1.start();
//th2、th3 在th1死亡时，也会跟着死亡
```

礼让线程

```
/*

```

```
 * public static void yield():暂停当前正在执行的线程对象，并执行其他线程。

 * 让多个线程的执行更和谐，但是不能靠它保证一人一次。

 */
public class ThreadYield extends Thread{

    @Override
    public void run() {
        for (int i = 0; i < 100; i++){
            System.out.println(getName() + ":" + i);

            Thread.yield();//暂停当前正在执行的线程对象
        }
    }
}
```

```
public class TreadDemo {
    public static void main(String[] args) {
        ThreadYield th1 = new ThreadYield();
        ThreadYield th2 = new ThreadYield();
        th1.setName("JOJO");
        th2.setName("林青霞");

        th1.start();
        th2.start();
    }
}
```

```
JOJO:0
林青霞:0
JOJO:1
林青霞:1
JOJO:2
林青霞:2
JOJO:3
林青霞:3
JOJO:4
林青霞:4
JOJO:5
林青霞:5
JOJO:6
林青霞:6
JOJO:7
林青霞:7
林青霞:8
JOJO:8
林青霞:9
JOJO:9
林青霞:10
JOJO:10
林青霞:11
JOJO:11
```

```
林青霞:12
JOJO:12
林青霞:13//
林青霞:14//
JOJO:13
林青霞:15
JOJO:14//
JOJO:15//
林青霞:16
JOJO:16
林青霞:17
JOJO:17
JOJO:18
林青霞:18
JOJO:19
林青霞:19

Process finished with exit code 0

```

中断线程

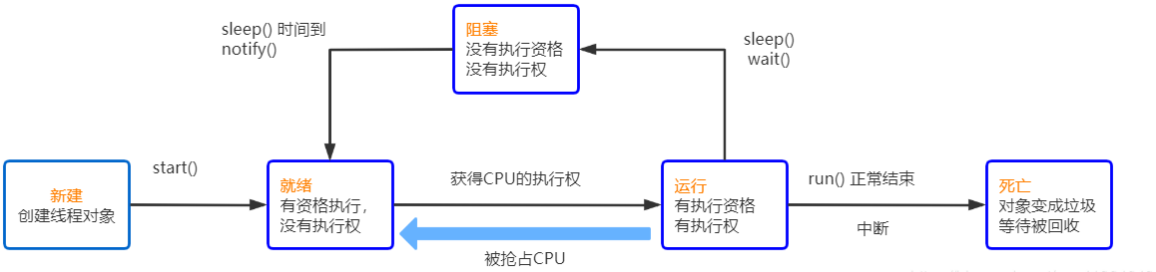
stop() 中断线程很暴力，后面的程序都不走了

```
public boolean isInterrupted()

public void interrupt()

public static boolean interrupted()
```

ThreadStop.java | ThreadStopDemo.java

```
package cn.itcast_04;

/*
 * public final void stop():让线程停止，过时了，但是还可以使用。
 * public void interrupt():中断线程。把线程的状态终止，并抛出一个InterruptedException。
 */
public class ThreadStopDemo {
    public static void main(String[] args) {
        ThreadStop ts = new ThreadStop();
        ts.start();

        // 你超过三秒不醒过来，我就干死你
        try {
            Thread.sleep(3000);
            // ts.stop();
            ts.interrupt();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

线程的生命周期



方式2：实现Runnable接口

多线程的实现方案有两种

- 继承 Thread类
- 实现 Runnable接口
  相比继承 Thread类，实现Runnable接口的好处
- 避免了 Java单继承的局限性
- 适合多个相同程序的代码去处理同一个资源的情况，把线程和程序的代码、数据有效分离，较好的体现了面向对象的设计思想

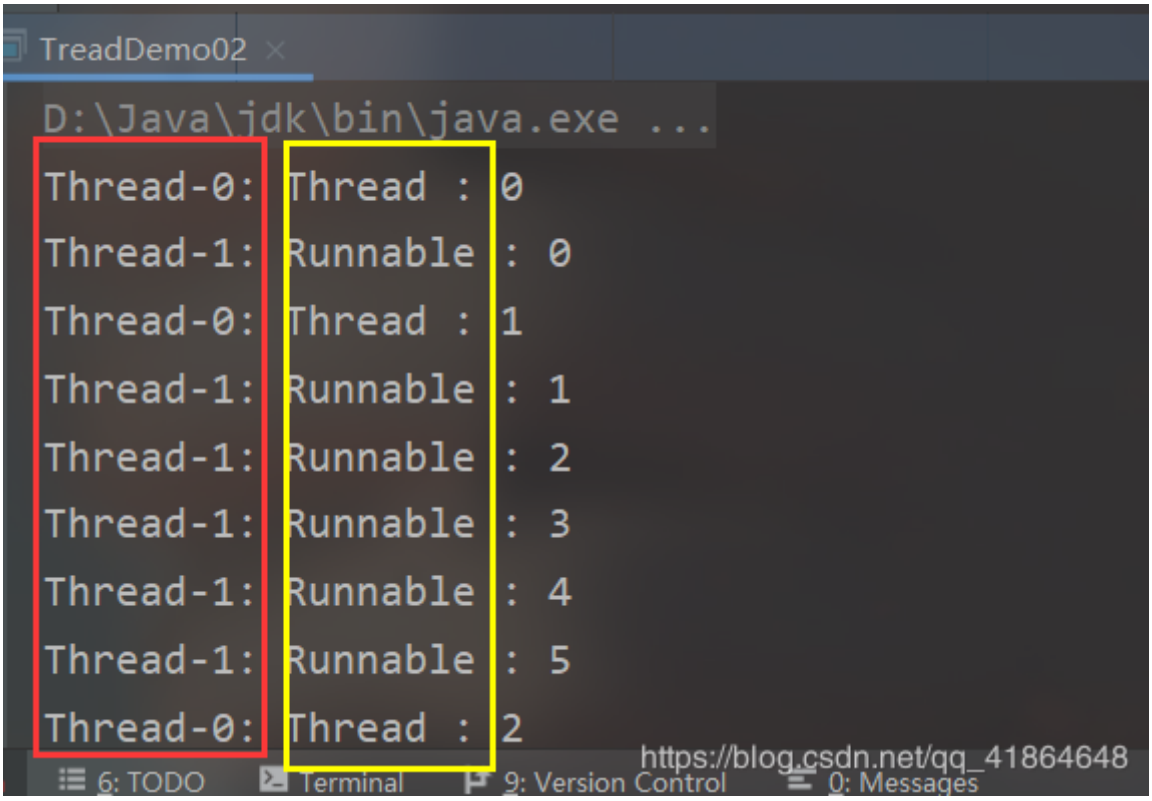
```
MyRunnable runnable = new MyRunnable();
        Thread th1 = new Thread(runnable,"线程一");
        Thread th2 = new Thread(runnable,"线程二");

        th1.start();
        th2.start();
```

```
/**
 * 自定义runnable接口，
 */
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("自定义runnable接口下的run()方法执行了...");
    }
}
```

练习：两种匿名内部类方式创建线程

```
public class TreadDemo02 {
    public static void main(String[] args) {
        // 继承Thread类实现多线程
        new Thread(){
            @Override
            public void run() {
                for (int i = 0; i<20;i++){
                    System.out.println(Thread.currentThread().getName() +
": Thread : " + i);
                }
            }
        }.start();

```

```
13          // 实现Runnable接口实现多线程
14          new Thread(new Runnable() {
15              @Override
16              public void run() {
17                  for (int i = 0; i<20;i++){
18                      System.out.println(Thread.currentThread().getName() +
    ": Runnable : " + i);
19                  }
20              }
21          }){
22              //thread子类
23          }.start();
24      }
25  }
26
```



###

方式3：实现Callable接口，线程池

参考文章：https://www.cnblogs.com/dolphin0520/p/3949310.html
https://blog.csdn.net/zsj777/article/details/85089993
https://blog.csdn.net/meng19910117/article/details/81043988

创建线程的2种方式，一种是直接继承Thread，另外一种就是实现Runnable接口。

这2种方式都有一个缺陷就是：在执行完任务之后无法获取执行结果。
如果需要获取执行结果，就必须通过共享变量或者使用线程通信的方式来达到效果，这样使用起来就比较麻烦。

我的理解：

- **Callable接口**
  Callable类似于Runnable，只不过Callable中要实现的方法是有返回值结果的，也就线程执行完任务后是会有一个结果返回给用户看的噢。

```
public interface Runnable {
    public abstract void run();
}
```

```
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

- **ExecutorService**
  ExecutorService，是用来把线程跑起来的服务，下面是该接口中声明的一些方法，可见它支持很多重载方式去将一个可以执行的任务跑起来，而且返回值都是 Future 类型，那么Future是干啥的呢?

```
<T> Future<T> submit(Callable<T> task);
<T> Future<T> submit(Runnable task, T result);
Future<?> submit(Runnable task);
```

- **Future**
  Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过`get方法`获取线程执行结果，该方法会阻塞直到任务返回结果。
  简言之，Future可以控制和查看任务的执行进度、还可以获取任务的结果，这不就是我们实现Callable想要的嘛~我们就是想获得结果呀!

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

- **FutureTask实现类**
  因为`Future`只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的实现类`FutureTask`。一个`FutureTask`可以用来包装一个`Callable`或是一个`runnable`对象。因为`FurtureTask`实现了`Runnable`方法，所以一个`FutureTask`可以提交(submit)给一个`Excutor`执行(excution).

  如上提供了两个构造函数，一个以Callable为参数，另外一个以Runnable为参数。这些类之间的关联对于任务建模的办法非常灵活，允许你基于FutureTask的Runnable特性（因为它实现了Runnable接口），把任务写成Callable，然后封装进一个由执行者调度并在必要时可以取消的FutureTask。

FutureTask可以由执行者调度，这一点很关键 。它对外提供的方法基本上就是Future和Runnable接口的组合：get()、cancel、isDone()、isCancelled()和run()，而run()方法通常都是由执行者调用，我们基本上不需要直接调用它。

总结：

- Callable和Runnable差不多，但是任务执行时有返回值有最后的结果。
- ExecutorService 是执行者，会帮我们去启动任务
- Future 是接口，声明了操控线程任务的方法
- FutureTask是Future的实现类，可以实例化，实例化时可以以Callable、Runnable为参数，还可以在执行者带动下跑起任务来，还可以跑起来后观察这个线程任务、还可以从中获得些结果。

```
public FutureTask(Callable<V> callable) {
        if (callable == null)
            throw new NullPointerException();
        sync = new Sync(callable);
    }
    public FutureTask(Runnable runnable, V result) {
        sync = new Sync(Executors.callable(runnable, result));
    }
```

一个Callable、Future、ExecutorService联合的例子
实现Callable接口

```
package com.ps.learn.socketio.service;

import java.util.concurrent.Callable;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 自定义一个任务类，实现Callable接口
 */
public  class MyCallableClass implements Callable<String> {
    // 标志位
    private AtomicInteger flag = new AtomicInteger(0);



    public AtomicInteger getFlag() {
        return flag;
    }

    public void setFlag(AtomicInteger flag) {
        this.flag = flag;
    }



    public String call() throws Exception {
        if (this.flag.get() == 0) {
            // 如果flag的值为0，则立即返回
            return "flag = 0";
        }
```

```
        if (this.flag.get() == 1) {
            // 如果flag的值为1，做一个无限循环
            try {
                while (true) {
                    System.out.println("looping......");
                    //sleep 中断当前线程
                    Thread.sleep(2000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "false";
        } else {
            // falg不为0或者1，则抛出异常
            throw new Exception("Bad flag value!");
        }
    }
}
```

程序启动类

```
package com.ps.learn.socketio.service;

/**
 * Author:ZhuShangJin
 * Date:2018/12/19
 */
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Callable 和 Future接口
 * Callable是类似于Runnable的接口，实现Callable接口的类和实现Runnable的类都是可被其它线程执行的任务。
 * Callable和Runnable有几点不同:
 * （1）Callable规定的方法是call()，而Runnable规定的方法是run().
 * （2）Callable的任务执行后可返回值，而Runnable的任务是不能返回值的。
 * （3）call()方法可抛出异常，而run()方法是不能抛出异常的。
 * （4）运行Callable任务可拿到一个Future对象， Future表示异步计算的结果。
 * 它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。
 * 通过Future对象可了解任务执行情况，可取消任务的执行，还可获取任务执行的结果。
 */
public class CallableAndFuture {


    public static void main(String[] args) {
        // 定义3个Callable类型的任务
        MyCallableClass task1 = new MyCallableClass();
        task1.setFlag(new AtomicInteger(0));
        MyCallableClass task2 = new MyCallableClass();
        task2.setFlag(new AtomicInteger(1));
        MyCallableClass task3 = new MyCallableClass();
        task3.setFlag(new AtomicInteger(2));

```

```
        // 创建一个执行任务的服务
        ExecutorService es = Executors.newFixedThreadPool(3);//创建一个线程池，大小为3
        try {
            // 提交并执行任务，任务启动时返回了一个Future对象，
            // 如果想得到任务执行的结果或者是异常可对这个Future对象进行操作
            Future future1 = es.submit(task1);
            // 获得第一个任务的结果，如果调用get方法，当前线程会等待任务执行完毕后才往下执行
            System.out.println("task1: " + future1.get());
//
            Future future2 = es.submit(task2);
            // 等待5秒后，再停止第二个任务。因为第二个任务进行的是无限循环
            Thread.sleep(5000);
            System.out.println("task2 cancel: " + future2.cancel(true));

            // 获取第三个任务的输出，因为执行第三个任务会引起异常
            // 所以下面的语句将引起异常的抛出
            Future future3 = es.submit(task3);
            System.out.println("task3: " + future3.get());
        } catch (Exception e) {
            //捕获异常
            e.printStackTrace();
        }
        // 停止任务执行服务
        es.shutdownNow();
    }
}
```

FutureTask执行多任务计算的使用场景

利用FutureTask和ExecutorService，可以用多线程的方式提交计算任务，主线程继续执行其他任务，当主线程需要子线程的计算结果时，在异步获取子线程的执行结果。

```
public class FutureTest1 {

    public static void main(String[] args) {
        Task task = new Task();// 新建异步任务
        FutureTask<Integer> future = new FutureTask<Integer>(task) {
            // 异步任务执行完成，回调
            @Override
            protected void done() {
                try {
                    System.out.println("future.done():" + get());//通过get方法获取执行结果
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (ExecutionException e) {
                    e.printStackTrace();
                }
            }
        };
        // 创建线程池（使用了预定义的配置）
        ExecutorService executor = Executors.newCachedThreadPool();//线程池
        executor.execute(future);//使用执行者与future搭配方式启动线程

        try {
            Thread.sleep(1000);
```

```
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
        // 可以取消异步任务
        // future.cancel(true);

        try {
            // 阻塞，等待异步任务执行完毕-获取异步任务的返回值
            System.out.println("future.get():" + future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

    // 异步任务
    static class Task implements Callable<Integer> {
        // 返回异步任务的执行结果
        @Override
        public Integer call() throws Exception {
            int i = 0;
            for (; i < 10; i++) {
                try {
                    System.out.println(Thread.currentThread().getName() +
"_" + i);
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            return i;
        }
    }

}
```

三、线程安全问题

安全问题出现的条件

- 是多线程环境
- 有共享数据
- 有多条语句操作共享数据

同步的好处和弊端

- 好处：解决了多线程的数据安全问题
- 弊端：当线程很多时，因为每个线程都会去判断同步上的锁，这是很耗费资源的，无形中会降低程序的运行效率

怎么实现呢？

- 把多条语句操作共享数据的代码给锁起来，让任意时刻只能有一个线程执行即可
- Java 提供了同步代码块的方式来解决

Volatile关键字

该关键字可以确保对一个变量的更新对其他线程马上可见。当一个变量被声明为volatile时，线程在写入变量时不会把值缓存在寄存器或者其他地方，而是会把值刷新回主内存。当其他线程读取该共享变量时 - ，会从主内存重新获取最新值，而不是使用当前线程的工作内存中的值。

volatile的内存语义和synchronized有相似之处，具体来说就是，

当线程写入了`volatile`变量值时就等价于线程退出`synchronized`同步块（把写入工作内存的变量值同步到主内存），

读取`volatile`变量值时就相当于进入同步块（先清空本地内存变量值，再从主内存获取最新值）。

volatilet特性

- 保证可见性
- 不能保证原子性
- 禁止指令重排

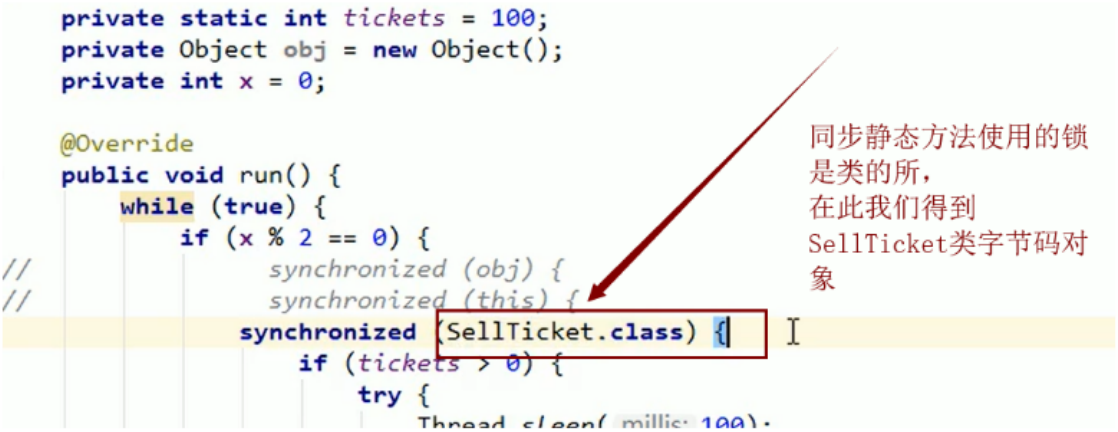
synchronized 关键字

synchronized 的内存语义：

- 这个内存语义就可以解决共享变量内存可见性问题。
- `进入synchronized块的内存语义`是把在synchronized块内使用到的变量从线程的工作内存中清除，这样在synchronized块内使用到该变量时就不会从线程的工作内存中获取，而是直接从主内存中获取。
- `退出synchronized块的内存语义`是把在synchronized块内对共享变量的修改刷新到主内存。会造成上下文切换的开销，独占锁，降低并发性
- synchronized保证了在同一时刻，只能有一个线程执行同步代码块，所以执行同步代码块的时候相当于是单线程操作了，那么线程的可见性、原子性、有序性（线程之间的执行顺序）它都能保证了。

方式一：synchronized 同步代码块

```
Object obj=new Object();

synchronized (obj) {
多条语句操作共享数据的代码
}
```



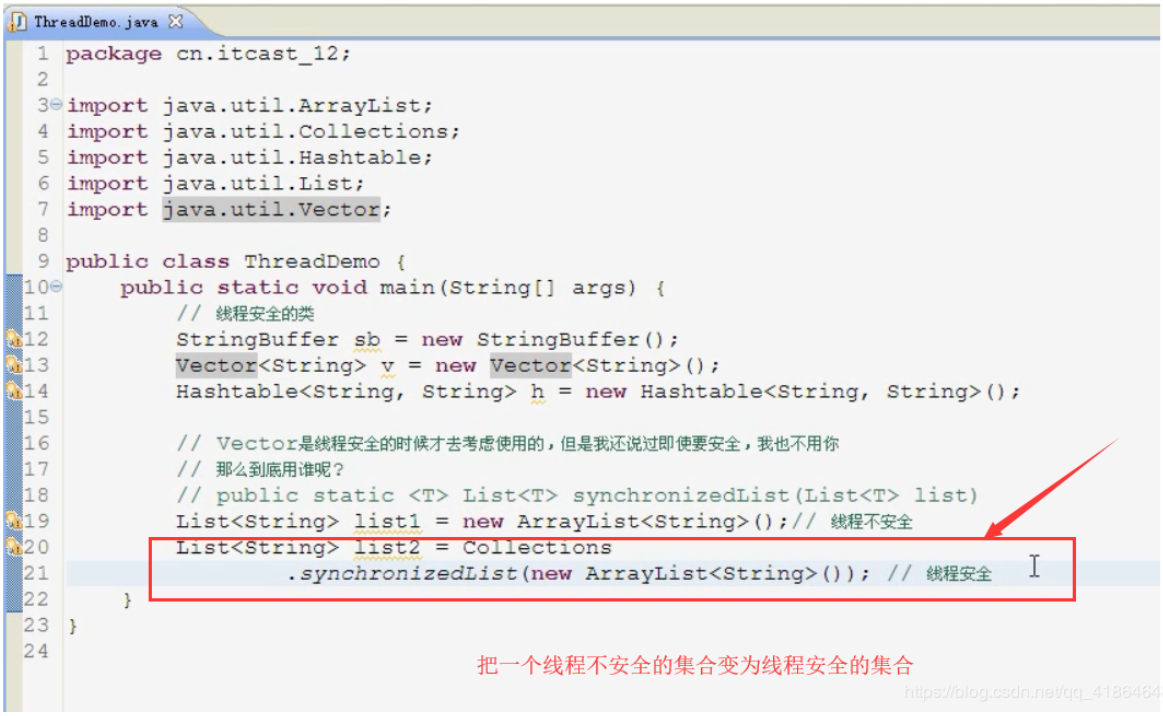
Lock接口

- Lock是接口不能直接实例化，这里采用它的实现类`ReentrantLock`来实例化（重入锁）
- 使用`try....finally`代码块来包裹

```
private int ticket=100;
private Lock lock=new ReentrantLock();//可重入锁


@Override
public void run() {

    while (true) {

        try {
            lock.lock();
            if (ticket > 0) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + "卖票"
+ ticket);
                ticket--;
            }
        }finally {
            lock.unlock();
        }

```

线程安全的集合

回顾：StringBuffer、Vector、Hashtable集合、CopyOnWriteArrayList



#

四、多线程的面试题

1. 基础题

1. 线程的实现方式

> 1.继承Thread类
>
> 2.实现Runnable接口
>
> 3.使用Callable和Future

2. start() 和 run() 区别

1.start（）方法来启动线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码；

通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。

然后通过JVM自动调用 run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

2.run（）方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码；程序中只有主线程------这一个线程， 其程序执行路径还是只有一条， 这样就没有达到写线程的目的。

3. 线程的blocked状态

线程正在等待获取锁。

- 进入BLOCKED状态，比如调用了sleep,或者wait方法
- 进行某个阻塞的io操作，比如因网络数据的读写进入BLOCKED状态
- 获取某个锁资源，从而加入到该锁的阻塞队列中而进入BLOCKED状态

4. sleep( ) 和 wait( ) 区别

方法sleep()的作用是在指定的毫秒数内让当前的"正在执行的线程"休眠（暂停执行）。sleep和wait的5个区别，推荐大家看下。

5. wait()

方法wait()的作用是使当前执行代码的线程进行等待，wait()是Object类通用的方法，该方法用来将当前线程置入"预执行队列"中，并在 wait()所在的代码处停止执行，直到接到通知或中断为止。

在调用wait之前线程需要获得该对象的对象级别的锁。代码体现上，即只能是同步方法或同步代码块内。调用wait()后当前线程释放锁。

6. notify()

notify()也是Object类的通用方法，也要在同步方法或同步代码块内调用，该方法用来通知哪些可能灯光该对象的对象锁的其他线程，如果有多个线程等待，则随机挑选出其中一个呈wait状态的线程，对其发出 通知 notify，并让它等待获取该对象的对象锁。

> **notify/notifyAll**

> notify等于说将等待队列中的一个线程移动到同步队列中，而notifyAll是将等待队列中的所有线程全部移动到同步队列中。

等待

```
synchronized(obj) {
        while(条件不满足) {
                obj.wait();
        }
        执行对应逻辑
}
```

通知

```
synchronized(obj) {
      改变条件
        obj.notifyAll();
}
```

下面再来几个很简单的面试题：

1:多线程有几种实现方案，分别是哪几种?
两种。

```
继承Thread类
实现Runnable接口

扩展一种：实现Callable接口。这个得和线程池结合。
```

2:同步有几种方式，分别是什么?
两种。

```
同步代码块
同步方法
```

3:启动一个线程是run()还是start()?它们的区别?
start();

```
run():封装了被线程执行的代码,直接调用仅仅是普通方法的调用
start():启动线程，并由JVM自动调用run()方法
```

4:sleep()和wait()方法的区别

```
    sleep():必须指时间;不释放锁。
    wait():可以不指定时间，也可以指定时间;释放锁。
```

5:为什么wait(),notify(),notifyAll()等方法都定义在Object类中

```
因为这些方法的调用是依赖于锁对象的，而同步代码块的锁对象是任意锁。
而Object代码任意的对象，所以，定义在这里面。
```

7. 如何优雅的设置睡眠时间

jdk1.5 后，引入了一个枚举 TimeUnit,对 sleep方法提供了很好的封装。

比如要表达2小时22分55秒899毫秒。

```
Thread.sleep(8575899L);
TimeUnit.HOURS.sleep(3);
TimeUnit.MINUTES.sleep(22);
TimeUnit.SECONDS.sleep(55);
TimeUnit.MILLISECONDS.sleep(899);
```

可以看到表达的含义更清晰，更优雅。线程休眠只会用 Thread.sleep？来，教你新姿势！推荐看下。

8. 停止线程

interrupted 和 isInterrupted

interrupted : 判断当前线程是否已经中断,会清除状态。

isInterrupted： 判断线程是否已经中断，不会清除状态。

> run方法执行完成，自然终止。
>
> stop()方法，suspend()以及resume()都是过期作废方法，使用它们结果不可预期。
>
> 大多数停止一个线程的操作使用Thread.interrupt()等于说给线程打一个停止的标记, 此方法不回去终止一个正在运行的线程，需要加入一个判断才能可以完成线程的停止。

9. yield() 礼让线程

放弃当前cpu资源，将它让给其他的任务占用cpu执行时间。但放弃的时间不确定，有可能刚刚放弃，马上又获得cpu时间片。多线程 Thread.yield 方法到底有什么用? 推荐看下。

10. join()

join是指把指定的线程加入到当前线程，比如join某个线程a,会让当前线程b进入等待,直到a的生命周期结束，此期间b线程是处于blocked状态。

11. 线程种类: 用户线程、守护线程

Java线程有两种，一种是用户线程，一种是守护线程。

> **守护线程的特点**
>
> 守护线程是一个比较特殊的线程，主要被用做程序中后台调度以及支持性工作。当Java虚拟机中不存在非守护线程时，守护线程才会随着JVM一同结束工作。

> **Java中典型的守护线程**
>
> GC（垃圾回收器）
>
> **如何设置守护线程**
>
> Thread.setDaemon(true)
>
> PS:Daemon属性需要再启动线程之前设置，不能再启动后设置。

Java虚拟机退出时Daemon线程中的finally块一定会执行?

Java虚拟机退出时Daemon线程中的finally块并不一定会执行。

代码示例:

```
public class XKDaemon {
    public static void main(String\[\] args) {
        Thread thread = new Thread(new DaemonRunner(),"xkDaemonRunner");
        thread.setDaemon(true);
        thread.start();

```

```
    }

    static class DaemonRunner implements Runnable {

        @Override
        public void run() {
            try {
                SleepUtils.sleep(10);
            } finally {
                System.out.println("Java技术栈公众号 daemonThread finally run ...");
            }

        }
    }
}
```

结果：

没有任何的输出，说明没有执行finally。

♥ 2. 锁的分类与区别

1. synchronized关键字

底层使用指令码让JVM帮我们来控制锁：`monitor enter` 获得锁和 `monitor exit` 释放锁。

synchronized关键字用法?

可以用于对代码块或方法的修饰，Synchronized 有几种用法? 推荐看下。

synchronized锁的是什么?

普通同步方法 ----------> 锁的是当前实力对象。

静态同步方法----------> 锁的是当前类的Class对象。

同步方法快 ----------> 锁的是synchonized括号里配置的对象。

synchronized特性：

- 可重入锁
- 不公平锁
- 非中断锁。Synchronized是非中断锁，必须等待线程执行完成或异常自动释放锁。（这样的设定一点好处时不会在同步代码块发生异常时而出现的死锁现象）
- 保证可见性、原子性
- 会造成线程阻塞

2. volatile关键字

volatile 是轻量级的synchronized,它在多处理器开发中保证了共享变量的"可见性"。详细看下这篇文章：volatile关键字解析。

volatile 特性：

> 一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：
>
> - 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
> - 不保证原子性
> - 禁止进行指令重排序。保证了有序性

其实volatile关键字的作用就是保证了可见性和有序性（不保证原子性），如果一个共享变量被volatile关键字修饰，那么如果一个线程修改了这个共享变量后，其他线程是立马可知的。为什么是这样的呢？比如，线程A修改了自己的共享变量副本，这时如果该共享变量没有被volatile修饰，那么本次修改不一定会马上将修改结果刷新到主存中，如果此时B去主存中读取共享变量的值，那么这个值就是没有被A修改之前的值。如果该共享变量被volatile修饰了，那么本次修改结果会强制立刻刷新到主存中，如果此时B去主存中读取共享变量的值，那么这个值就是被A修改之后的值了。

什么是禁止指令重排：volatile能禁止指令重新排序，在指令重排序优化时，在volatile变量之前的指令不能在volatile之后执行，在volatile之后的指令也不能在volatile之前执行，所以它保证了有序性。

> 原文链接：https://blog.csdn.net/songzi1228/article/details/99975018

3. Lock接口

锁可以防止多个线程同时共享资源。Java5前程序是靠synchronized实现锁功能。Java5之后，并发包新增Lock接口来实现锁功能。

Lock接口提供 synchronized不具备的主要特性

特　　性	描　　述
尝试非阻塞地获取锁	当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁
能被中断地获取锁	与 synchronized 不同，获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放
超时获取锁	在指定的截止时间之前获取锁，如果截止时间到了仍旧无法获取[illegible]，周派技术栈

Lock锁的特性

- 中断锁。允许线程在长时间等待锁而不得时，中断获取等待，而去做别的事情。
- 调用 unlock() 方法手动 在 finally 块中释放锁。

♥ 4. CAS机制原理和ABA问题

> 什么是CAS机制：https://mp.weixin.qq.com/s/f9PYMnpAgS1gAQYPDuCq-w
>
> CAS是英文单词**Compare And Swap**的缩写，翻译过来就是比较并替换。
>
> CAS机制当中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。
>
> 更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。

> CAS机制底层原理 + ABA问题： https://mp.weixin.qq.com/s/nRnQKhiSUrDKu3mz3vltWg
>
> 总结：
>
> **1. Java语言CAS底层如何实现?**
>
> 利用unsafe提供了原子性操作方法。
>
> **2. 什么是ABA问题? 怎么解决?**
>
> 当一个值从A更新成B，又更新会A，普通CAS机制会误判通过检测。
>
> 利用版本号比较可以有效解决ABA问题。

5. synchronized和volatile区别

volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

- volatile仅能使用在变量，范围较小；synchronized则可以使用在方法、类、代码块上 。
- volatile仅能实现变量的修改可见性，并不能保证原子性，禁止指令重排保证有序性；synchronized则三点都保证。
- volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。
- volatile标记的变量不会被编译器优化（禁止指令重排）；synchronized标记的变量可以被编译器优化。

记忆技巧： ① 关键字范围 ② 三点式 ③ 阻塞

6. synchronized和Lock的区别

两种锁的底层实现

> - Synchronized：悲观锁。底层使用 JVM指令码方式 来控制锁的，映射成字节码指令就是增加来两个指令：`monitorenter和monitorexit`。当线程执行遇到monitorenter指令时会尝试获取内置锁，如果获取锁则锁计数器+1，如果没有获取锁则阻塞；当遇到monitorexit指令时锁计数器-1，如果计数器为0则释放锁。
> - Lock：底层是CAS乐观锁，依赖AbstractQueuedSynchronizer类，把所有的请求线程构成一个CLH队列。而对该队列的操作均通过Lock-Free（CAS）操作。

Synchronized和Lock比较

> - Synchronized是关键字，可以修饰方法和代码块，内置指令语言实现，Lock是接口。
> - Synchronized在线程发生异常时会自动释放锁，因此不会发生异常死锁。Lock异常时不会自动释放锁，所以需要在finally中实现释放锁。
> - Lock是可以中断锁，Synchronized是非中断锁，必须等待线程执行完成或异常自动释放锁。
> - Lock可以使用读锁提高多线程读效率。

记忆技巧：两个角度考虑：① 关键字OR接口 ② 是否是中断锁

> 中断锁的含义：从等待线程角度考虑
>
> 线程A和B都要获取对象O的锁定，假设A获取了对象O锁，B将等待A释放对O的锁定，
>
> 如果使用 synchronized ，如果A不释放，B将一直等下去，不能被中断
>
> 如果 使用ReentrantLock，如果A不释放，可以使B在等待了足够长的时间以后，中断等待，而干别的事情

7. ReentrantLock 重入锁

重进入是什么意思?

支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。除此之外，该锁的还支持获取锁时的公平和非公平性选择。

详细阅读：到底什么是重入锁，拜托，一次搞清楚！

> 重进入是指任意线程在获取到锁之后能够再次获锁而不被自己锁住。
>
> 该特性主要解决以下两个问题：
>
> - 一、锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是则再次成功获取。
> - 二、所得最终释放。线程重复n次是获取了锁，随后在第n次释放该锁后，其他线程能够获取到该锁。

ReentrantLock获取锁定与三种方式：

- lock(), 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁
- tryLock(), 如果获取了锁立即返回true，如果别的线程正持有锁，立即返回false；
- **tryLock**(long timeout,TimeUnit unit)， 如果获取了锁定立即返回true，如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回true，如果等待超时，返回false；
- lockInterruptibly: 如果获取了锁定立即返回，如果没有获取锁定，当前线程处于休眠状态，直到或者锁定，或者当前线程被别的线程中断

ReenTrantLock独有的功能：

1.ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。

2.ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

3.ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。

7. Synchronized 与 ReentrantLock 的区别

原文：推荐看下：Synchronized 与 ReentrantLock 的区别！

- 都是可重入锁
- ReentrantLock 可设置公平锁。Synchronized 只是非公平锁。
- Synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的，底层使用CAS机制，尽量避免了内核态阻塞，需要手动加锁、释放锁。
- 性能差别：在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了。
- 锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized
- ReentrantLock 独有功能

记忆技巧： ① 重入 ② 公平 ③性能 ④ 锁粒度

8. Synchronized 底层原理

第一个问题：锁存在在哪个位置？为什么对象都可以成为锁?

锁存在于每个对象的 `markOop 对象头` 中.对于为什么每个对象都可以成为锁呢？ 因为每个 Java Object 在 JVM 内部都有一个 native 的 C++ 对象 oop/oopDesc 与之对应，而对应的 oop/oopDesc 都会存在一个`markOop 对象头`，而这个`对象头是存储锁的位置`，里面还有对象监视器，即`ObjectMonitor`，所以这也是为什么每个对象都能成为锁的原因之一。

Java对象头：

在Hotspot虚拟机中，对象在内存中的布局分为三块区域：`对象头、实例数据和对齐填充`；Java对象头是实现synchronized的锁对象的基础，一般而言，synchronized使用的锁对象是存储在Java对象头里。它是轻量级锁和偏向锁的关键

Mark Word 对象头：

Mark Word用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等。Java对象头一般占有两个机器码（在32位虚拟机中，1个机器码等于4字节，也就是32bit），下面就是对象头的一些信息：

***

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

***

第二个问题：那么 synchronized是如何实现锁的呢?

synchronized的锁是进行过优化的，引入了偏向锁、轻量级锁；锁的级别从低到高逐步升级， 无锁->偏向锁->轻量级锁->重量级锁.锁的类型：锁从宏观上分类，分为悲观锁与乐观锁。

详细请看synchronized是如何实现锁：https://www.cnblogs.com/wuzhenzhao/p/10250801.html

```
void ATTR ObjectMonitor::enter(TRAPS) {//获取重量级锁的过程
  // The following code is ordered to check the most common cases first
  // and to reduce RTS->RTO cache line upgrades on SPARC and IA32
processors.
  Thread * const Self = THREAD ;
  void * cur ;

  cur = Atomic::cmpxchg_ptr (Self, &_owner, NULL) ;//进行CAS自旋操作
  if (cur == NULL) {
     // Either ASSERT _recursions == 0 or explicitly set _recursions = 0.
     assert (_recursions == 0   , "invariant") ;
     assert (_owner      == Self, "invariant") ;
     // CONSIDER: set or assert OwnerIsThread == 1
     return ;
  }
  //自旋结果相等，则重入（重入的原理）
  if (cur == Self) {
     // TODO-FIXME: check for integer overflow!  BUGID 6557169.
     _recursions ++ ;
     return ;
  }//接下去就是有并发的情况下竞争的过程了
....
```

16. Java并发容器，你知道几个?

ConcurrentHashMap、CopyOnWriteArrayList 、CopyOnWriteArraySet 、ConcurrentLinkedQueue、

ConcurrentLinkedDeque、ConcurrentSkipListMap、ConcurrentSkipListSet、ArrayBlockingQueue、

LinkedBlockingQueue、LinkedBlockingDeque、PriorityBlockingQueue、SynchronousQueue、

LinkedTransferQueue、DelayQueue

> ### ConcurrentHashMap
>
> 并发安全版HashMap,java7中采用分段锁技术来提高并发效率，默认分16段。Java8放弃了分段锁，采用CAS，同时当哈希冲突时，当链表的长度到8时，会转化成红黑树。（如需了解细节，见jdk中代码）
>
> 推荐阅读：HashMap, ConcurrentHashMap 一次性讲清楚！
>
> ### ConcurrentLinkedQueue
>
> 基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用cas算法来实现。（如需了解细节，见jdk中代码）

17. 什么是线程池，如何使用?

线程池就是事先将多个线程对象放到一个容器中，当使用的时候就不用new线程而是直接去池中拿线程即可，节省了开辟子线程的时间，提高的代码执行效率。在JDK的java.util.concurrent.Executors中提供了生成多种线程池的静态方法。

```
ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(4);
ScheduledExecutorService newScheduledThreadPool =
Executors.newScheduledThreadPool(4);
ExecutorService newSingleThreadExecutor =
Executors.newSingleThreadExecutor();
```

然后调用他们的 execute 方法即可。

17. 请叙述一下您对线程池的理解?

（如果问到了这样的问题，可以展开的说一下线程池如何用、线程池的好处、线程池的启动策略）合理利用线程池能够带来三个好处。

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

17. 常用的线程池有哪些?

- newSingleThreadExecutor：创建一个单线程的线程池，此线程池保证所有任务的执行顺序按照任务的提交顺序执行
- newFixedThreadPool：创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。
- newCachedThreadPool：创建一个可缓存的线程池，此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
- newScheduledThreadPool：创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。
- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

17. 创建线程池参数有哪些，作用?

```
public ThreadPoolExecutor(   int corePoolSize,
                             int maximumPoolSize,
                             long keepAliveTime,
                             TimeUnit unit,
                             BlockingQueue<Runnable> workQueue,
                             ThreadFactory threadFactory,
                             RejectedExecutionHandler handler)
```

1.corePoolSize:核心线程池大小，当提交一个任务时，线程池会创建一个线程来执行任务，即使其他空闲的核心线程能够执行新任务也会创建，等待需要执行的任务数大于线程核心大小就不会继续创建。

2.maximumPoolSize:线程池最大数，允许创建的最大线程数，如果队列满了，并且已经创建的线程数小于最大线程数，则会创建新的线程执行任务。如果是无界队列，这个参数基本没用。

3.keepAliveTime: 线程保持活动时间，线程池工作线程空闲后，保持存活的时间，所以如果任务很多，并且每个任务执行时间较短，可以调大时间，提高线程利用率。

4.unit: 线程保持活动时间单位，天（DAYS)、小时(HOURS)、分钟(MINUTES、毫秒MILLISECONDS)、微秒(MICROSECONDS)、纳秒(NANOSECONDS)

5.workQueue: 任务队列，保存等待执行的任务的阻塞队列。

一般来说可以选择如下阻塞队列：

ArrayBlockingQueue:基于数组的有界阻塞队列。

LinkedBlockingQueue:基于链表的阻塞队列。

SynchronizedQueue:一个不存储元素的阻塞队列。

PriorityBlockingQueue:一个具有优先级的阻塞队列。

6.threadFactory：设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

7.handler: 饱和策略也叫拒绝策略。当队列和线程池都满了，即达到饱和状态。所以需要采取策略来处理新的任务。默认策略是AbortPolicy。

```
AbortPolicy:直接抛出异常。

CallerRunsPolicy: 调用者所在的线程来运行任务。

DiscardOldestPolicy:丢弃队列里最近的一个任务，并执行当前任务。

DiscardPolicy:不处理，直接丢掉。

当然可以根据自己的应用场景，实现RejectedExecutionHandler接口自定义策略。
```

18. 向线程池提交任务

可以使用execute()和submit() 两种方式提交任务。

execute():无返回值，所以无法判断任务是否被执行成功。

submit():用于提交需要有返回值的任务。线程池返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功，并且可以通过future的get()来获取返回值，get()方法会阻塞当前线程知道任务完成。get(long timeout,TimeUnit unit)可以设置超市时间。

19. 关闭线程池

可以通过shutdown()或shutdownNow()来关闭线程池。它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt来中断线程，所以无法响应终端的任务可以能永远无法停止。

shutdownNow首先将线程池状态设置成STOP,然后尝试停止所有的正在执行或者暂停的线程，并返回等待执行任务的列表。

shutdown只是将线程池的状态设置成shutdown状态，然后中断所有没有正在执行任务的线程。

只要调用两者之一，isShutdown就会返回true,当所有任务都已关闭，isTerminaed就会返回true。

一般来说调用shutdown方法来关闭线程池，如果任务不一定要执行完，可以直接调用shutdownNow方法。

20. Executor

从JDK5开始，把工作单元和执行机制分开。工作单元包括Runnable和Callable,而执行机制由Executor框架提供。

Executor框架的主要成员

ThreadPoolExecutor :可以通过工厂类Executors来创建。

可以创建3种类型的

ThreadPoolExecutor：SingleThreadExecutor、FixedThreadPool、CachedThreadPool。

ScheduledThreadPoolExecutor ： 可以通过工厂类Executors来创建。

可以创建2中类型的ScheduledThreadPoolExecutor：ScheduledThreadPoolExecutor、SingleThreadScheduledExecutor

> - Future接口:Future和实现Future接口的FutureTask类来表示异步计算的结果。
> - Runnable和Callable:它们的接口实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。Runnable不能返回结果，Callable可以返回结果。

21. FixedThreadPool

可重用固定线程数的线程池。

查看源码：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

corePoolSize 和maxPoolSize都被设置成我们设置的nThreads。

当线程池中的线程数大于corePoolSize ,keepAliveTime为多余的空闲线程等待新任务的最长时间，超过这个时间后多余的线程将被终止，如果设为0，表示多余的空闲线程会立即终止。

工作流程：

1.当前线程少于corePoolSize,创建新线程执行任务。

2.当前运行线程等于corePoolSize,将任务加入LinkedBlockingQueue。

3.线程执行完1中的任务，会循环反复从LinkedBlockingQueue获取任务来执行。

LinkedBlockingQueue作为线程池工作队列（默认容量Integer.MAX_VALUE)。因此可能会造成如下赢下。

1.当线程数等于corePoolSize时，新任务将在队列中等待，因为线程池中的线程不会超过corePoolSize。

2.maxnumPoolSize等于说是一个无效参数。

3.keepAliveTime等于说也是一个无效参数。

4.运行中的FixedThreadPool(未执行shundown或shundownNow))则不会调用拒绝策略。

5.由于任务可以不停的加到队列，当任务越来越多时很容易造成OOM。

22. CachedThreadPool

根据需要创建新线程的线程池。

查看源码：

```
public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
}
```

corePoolSize设置为0，maxmumPoolSize为Integer.MAX_VALUE。keepAliveTime为60秒。

工作流程：

1.首先执行SynchronousQueue.offer (Runnable task)。如果当前maximumPool 中有空闲线程正在执行S ynchronousQueue.poll(keepAliveTIme,TimeUnit.NANOSECONDS)，那么主线程执行offer操作与空闲线程执行的poll操作配对成功，主线程把任务交给空闲线程执行,execute方 法执行完成;否则执行下面的步骤2。

2.当初始maximumPool为空或者maximumPool中当前没有空闲线程时，将没有线程执行SynchronousQueue.poll (keepAliveTime，TimeUnit.NANOSECONDS)。这种情况下，步骤 1将失败。此时CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成。

3.在步骤2中新创建的线程将任务执行完后，会执行SynchronousQueue.poll (keepAliveTime，TimeUnit.NANOSECONDS)。这个poll操作会让空闲线程最多在SynchronousQueue中等待60秒钟。如果60秒钟内主线程提交了一个新任务(主线程执行步骤1)，那么这个空闲线程将执行主线程提交的新任务;否则，这个空闲线程将终止。由于空闲60秒的空闲线程会被终止,因此长时间保持空闲的CachedThreadPool不会使用任何资源。

一般来说它适合处理时间短、大量的任务。

23. 什么情况下导致线程死锁，遇到线程死锁该怎么解决?

● 死锁的定义：所谓死锁是指多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。

● 死锁产生的必要条件：

> - 互斥条件：线程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某资源仅为一个线程所占有。此时若有其他线程请求该资源，则请求线程只能等待。
> - 不剥夺条件：线程所获得的资源在未使用完毕之前，不能被其他线程强行夺走，即只能由获得该资源的线程自己来释放（只能是主动释放)。
> - 请求和保持条件：线程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他线程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。
> - 循环等待条件：存在一种线程资源的循环等待链，链中每一个线程已获得的资源同时被链中下一个线程所请求。即存在一个处于等待状态的线程集合{Pl，P2，…，pn}，其中Pi等待的资源被P(i+1)占有（i=0，1，…，n-1)，Pn等待的资源被P0占有，如图2-15所示。

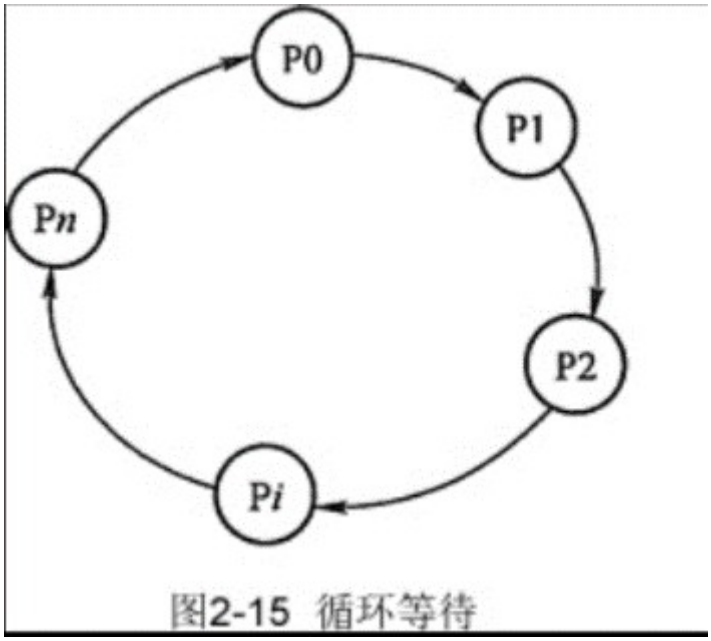


图2-15 循环等待

24. 请说出同步线程及线程调度相关的方法?

> wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
>
> sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
>
> notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
>
> notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；
>
> 注意：java 5通过Lock接口提供了显示的锁机制，Lock接口中定义了加锁（lock()方法）和解锁（unLock()方法），增强了多线程编程的灵活性及对线程的协调。