- 基础知识
  - 并发编程的优缺点
    - 为什么要使用并发编程（并发编程的优点）
    - 并发编程有什么缺点
    - 并发编程三要素是什么？在 Java 程序中怎么保证多线程的运行安全？
    - 并行和并发有什么区别？
    - 什么是多线程，多线程的优劣？
  - 线程和进程区别
    - 什么是线程和进程？
    - 进程与线程的区别
    - 什么是上下文切换？
    - 守护线程和用户线程有什么区别呢？
    - 如何在 Windows 和 Linux 上查找哪个线程cpu利用率最高？
    - 什么是线程死锁

- 形成死锁的四个必要条件是什么
- 如何避免线程死锁

- 创建线程的四种方式
  - 创建线程有哪几种方式?
  - 说一下 runnable 和 callable 有什么区别?
  - 线程的 run()和 start()有什么区别?
  - 为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法?
  - 什么是 Callable 和 Future?
  - 什么是 FutureTask
- 线程的状态和基本操作
  - 说说线程的生命周期及五种基本状态?
  - Java 中用到的线程调度算法是什么?
  - 线程的调度策略

- 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)?
- 请说出与线程同步以及线程调度相关的方法。
- sleep() 和 wait() 有什么区别?
- 你是如何调用 wait() 方法的? 使用 if 块还是循环? 为什么?
- 为什么线程通信的方法 wait(), notify()和 notifyAll()被定义在 Object 类里?
- 为什么 wait(), notify()和 notifyAll()必须在同步方法或者同步块中被调用?
- Thread 类中的 yield 方法有什么作用?
- 为什么 Thread 类的 sleep()和 yield ()方法是静态的?

- 线程的 sleep()方法和 yield()方法有什么区别?
- 如何停止一个正在运行的线程?
- Java 中 interrupted 和 isInterrupted 方法的区别?
- 什么是阻塞式方法?
- Java 中你怎样唤醒一个阻塞的线程?
- notify() 和 notifyAll() 有什么区别?
- 如何在两个线程间共享数据?
- Java 如何实现多线程之间的通讯和协作?
- 同步方法和同步块，哪个是更好的选择?
- 什么是线程同步和线程互斥，有哪几种实现方式?
- 在监视器(Monitor)内部，是如何做线程同步的？程序应该做哪

种级别的同步?

- 如果你提交任务时，线程池队列已满，这时会发生什么
- 什么叫线程安全？servlet 是线程安全吗?
- 在 Java 程序中怎么保证多线程的运行安全?
- 你对线程优先级的理解是什么?
- 线程类的构造方法、静态块是被哪个线程调用的
- Java 中怎么获取一份线程 dump 文件？你如何在 Java 中获取线程堆栈?
- 一个线程运行时发生异常会怎样?
- Java 线程数过多会造成什么异常?

  - 并发理论
    - Java内存模型

- Java中垃圾回收有什么目的？什么时候进行垃圾回收？
- 如果对象的引用被置为null，垃圾收集器是否会立即释放对象占用的内存？
- finalize()方法什么时候被调用？析构函数(finalization)的目的是什么？

- 重排序与数据依赖性
  - 为什么代码会重排序？
- as-if-serial规则和happens-before规则的区别

- 并发关键字
  - synchronized
    - synchronized 的作用？
    - 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗
    - 说一下 synchronized 底层实现原理？
    - 什么是自旋

- 多线程中 synchronized 锁升级的原理是什么?
- 线程 B 怎么知道线程 A 修改了变量
- 当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B?
- synchronized、volatile、CAS 比较
- synchronized 和 Lock 有什么区别?
- synchronized 和 ReentrantLock 区别是什么?

- volatile
  - volatile 关键字的作用
  - Java 中能创建 volatile 数组吗?
  - volatile 变量和 atomic 变量有什么不同?

- volatile 能使得一个非原子操作变成原子操作吗?
- volatile 修饰符的有过什么实践?
- synchronized 和 volatile 的区别是什么?

- final
  - 什么是不可变对象，它对写并发应用有什么帮助?

- Lock体系
  - Lock简介与初识AQS
    - Java Concurrency API 中的 Lock 接口(Lock interface)是什么？对比同步它有什么优势?
    - 乐观锁和悲观锁的理解及如何实现，有哪些实现方式?
    - 什么是 CAS
    - CAS 的会产生什么问题?
    - 什么是死锁?
    - 产生死锁的条件是什么？怎么防止死锁?

- 死锁与活锁的区别，死锁与饥饿的区别?
- 多线程锁的升级原理是什么?

- AQS(AbstractQueuedSynchronizer)详解与源码分析
  - AQS 介绍
  - AQS 原理分析
- ReentrantLock(重入锁)实现原理与公平锁非公平锁区别
  - 什么是可重入锁（ReentrantLock）？
- 读写锁ReentrantReadWriteLock源码分析
  - ReadWriteLock 是什么
- Condition源码分析与等待通知机制
- LockSupport详解

- 并发容器
  - 并发容器之ConcurrentHashMap详解(JDK1.8版本)与源码分析
    - 什么是ConcurrentHashMap?

- ThreadLocal内存泄漏解决方案?
- 并发容器之BlockingQueue详解
  - 什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型?
- 并发容器之ConcurrentLinkedQueue详解与源码分析
- 并发容器之ArrayBlockingQueue与LinkedBlockingQueue详解

- 线程池
  - Executors类创建四种常见线程池
    - 什么是线程池？有哪几种创建方式?
    - 线程池有什么优点?
    - 线程池都有哪些状态?
    - 什么是 Executor 框架？为什么使用 Executor 框架?
    - 在 Java 中 Executor 和 Executors 的区别?

        - 线程池中 submit() 和 execute() 方法有什么区别?
        - 什么是线程组，为什么在 Java 中不推荐使用?
    - 线程池之ThreadPoolExecutor详解
        - Executors和ThreaPoolExecutor创建线程池的区别
        - 你知道怎么创建线程池吗?
        - ThreadPoolExecutor构造函数重要参数分析
        - ThreadPoolExecutor饱和策略
        - 一个简单的线程池Demo:`Runnable`+`ThreadPoolExecutor`
    - 线程池之ScheduledThreadPoolExecutor详解
    - FutureTask详解
- 原子操作类

基础知识

并发编程的优缺点

为什么要使用并发编程（并发编程的优点）

- 充分利用多核CPU的计算能力：通过并发编程的形式可以将多核CPU的计算能力发挥到极致，性能得到提升
- 方便进行业务拆分，提升系统并发能力和性能：在特殊的业务场景下，先天的就适合于并发编程。现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线

程机制可以大大提高系统整体的并发能力以及性能。面对复杂业务模型，并行程序会比串行程序更适应业务需求，而并发编程更能吻合这种业务拆分 。

并发编程有什么缺点

并发编程的目的就是为了能提高程序的执行效率，提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如**：内存泄漏、上下文切换、线程安全、死锁**等问题。

并发编程三要素是什么？在 Java 程序中怎么保证多线程的运行安全?

并发编程三要素（线程的安全性问题体现在）：

原子性：原子，即一个不可再被分割的颗粒。原子性指的是一个或多个操作要么全部执行成功要么全部执行失败。

可见性：一个线程对共享变量的修改,另一个线程能够立刻看到。（synchronized,volatile）

有序性：程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排序）

出现线程安全问题的原因：

- 线程切换带来的原子性问题
- 缓存导致的可见性问题
- 编译优化带来的有序性问题

解决办法：

- JDK Atomic开头的原子类、synchronized、LOCK，可以解决原子性问题
- synchronized、volatile、LOCK，可以解决可见性问题
- Happens-Before 规则可以解决有序性问题

并行和并发有什么区别?

- 并发：多个任务在同一个 CPU 核上，按细分的时间片轮流(交替)执行，从逻辑上来看那些任务是同时执行。
- 并行：单位时间内，多个处理器或多核处理器同时处理多个任务，是真正意义上的“同时进行”。
- 串行：有n个任务，由一个线程按顺序执行。由于任务、方法都在一个线程执行所以不存在线程不安全情况，也就不存在临界区的问题。

做一个形象的比喻：

并发 = 两个队列和一台咖啡机。

并行 = 两个队列和两台咖啡机。

串行 = 一个队列和一台咖啡机。

什么是多线程，多线程的优劣?

多线程：多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。

多线程的好处：

可以提高 CPU 的利用率。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其它的线程而不是等待，这样就大大提高了程序的效率。也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。

多线程的劣势：

- 线程也是程序，所以线程需要占用内存，线程越多占用内存也越多；
- 多线程需要协调和管理，所以需要 CPU 时间跟踪线程；
- 线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题。

线程和进程区别

什么是线程和进程?

进程

一个在内存中运行的应用程序。每个进程都有自己独立的一块内存空间，一个进程可以有多个线程，比如在Windows系统中，一个运行的xx.exe就是一个进程。

线程

进程中的一个执行任务（控制单元），负责当前进程中程序的执行。一个进程至少有一个线程，一个进程可以运行多个线程，多个线程可共享数据。

进程与线程的区别

线程具有许多传统进程所具有的特征，故又称为轻型进程(Light—Weight Process)或进程元；而把传统的进程称为重型进程(Heavy—Weight Process)，它相当于只有一个线程的任务。在引入了线程的操作系统中，通常一个进程都有若干个线程，至少包含一个线程。

根本区别：进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位

资源开销：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

包含关系：如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

内存分配：同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的

影响关系：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

执行过程：每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

什么是上下文切换?

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到再加载的过程就是一次上下文切换**。

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

守护线程和用户线程有什么区别呢?

守护线程和用户线程

- **用户 (User) 线程**：运行在前台，执行具体的任务，如程序的主线程、连接网络的子线程等都是用户线程
- **守护 (Daemon) 线程**：运行在后台，为其他前台线程服务。也可以说守护线程是 JVM 中非守护线程的 **“佣人”**。一旦所有用户线程都结束运行，守护线程会随 JVM 一起结束工作

main 函数所在的线程就是一个用户线程啊，main 函数启动的同时在 JVM 内部同时还启动了好多守护线程，比如垃圾回收线程。

比较明显的区别之一是用户线程结束，JVM 退出，不管这个时候有没有守护线程运行。而守护线程不会影响 JVM 的退出。

注意事项:

1. setDaemon(true)必须在start()方法前执行，否则会抛出 IllegalThreadStateException 异常
2. 在守护线程中产生的新线程也是守护线程
3. 不是所有的任务都可以分配给守护线程来执行，比如读写操作或者计算逻辑
4. 守护 (Daemon) 线程中不能依靠 finally 块的内容来确保执行关闭或清理资源的逻辑。因为我们上面也说过了一旦所有用户线程都结束运行，守护线程会随 JVM 一起结束工作，所以守护 (Daemon) 线程中的 finally 语句块可能无法被执行。

如何在 Windows 和 Linux 上查找哪个线程cpu利用率最高?

windows上面用任务管理器看，linux下可以用 top 这个工具看。

1. 找出cpu耗用厉害的进程pid， 终端执行top命令，然后按下shift+p 查找出cpu利用最厉害的pid号
2. 根据上面第一步拿到的pid号，top -H -p pid 。然后按下shift+p，查找出cpu利用率最厉害的线程号，比如top -H -p 1328
3. 将获取到的线程号转换成16进制，去百度转换一下就行
4. 使用jstack工具将进程信息打印输出，jstack pid号 > /tmp/t.dat，比如 jstack 31365 > /tmp/t.dat

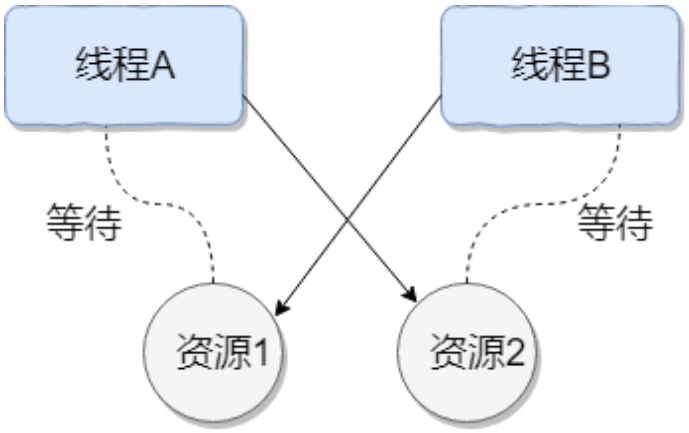
5. 编辑/tmp/t.dat文件，查找线程号对应的信息

什么是线程死锁

百度百科：死锁是指两个或两个以上的进程（线程）在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程（线程）。

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



线程死锁

下面通过一个例子来说明线程死锁，代码模拟了上图的死锁的情况 (代码来源于《并发编程之美》)：

```
publicclassDeadLockDemo{privatestatic Object resource1 =newObject();//资源
1privatestatic Object resource2 =newObject();//资源 2publicstaticvoidmain(String[]
args){newThread(()->{synchronized(resource1){
System.out.println(Thread.currentThread()+"get resource1");try{
Thread.sleep(1000);}catch(InterruptedException e){                e.printStackTrace();}
System.out.println(Thread.currentThread()+"waiting get
resource2");synchronized(resource2){
System.out.println(Thread.currentThread()+"get resource2");}}},"线程
1").start();newThread(()->{synchronized(resource2){
System.out.println(Thread.currentThread()+"get resource2");try{
Thread.sleep(1000);}catch(InterruptedException e){                e.printStackTrace();}
System.out.println(Thread.currentThread()+"waiting get
```

```
resource1");synchronized(resource1){
System.out.println(Thread.currentThread()+"get resource1");}}},"线程 2").start();}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33

- 34
- 35
- 36

输出结果

Thread[线程 1,5,main]get resource1Thread[线程 2,5,main]get resource2Thread[线程 1,5,main]waiting get resource2Thread[线程 2,5,main]waiting get resource1

- 1
- 2
- 3
- 4

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000)；让线程 A 休眠 1s 为的是让线程 B 得到CPU执行权，然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。上面的例子符合产生死锁的四个必要条件。

形成死锁的四个必要条件是什么

1. 互斥条件：线程(进程)对于所分配到的资源具有排它性，即一个资源只能被一个线程(进程)占用，直到被该线程(进程)释放
2. 请求与保持条件：一个线程(进程)因请求被占用资源而发生阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：线程(进程)已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
4. 循环等待条件：当发生死锁时，所等待的线程(进程)必定会形成一个环路（类似于死循环），造成永久阻塞

如何避免线程死锁

我们只要破坏产生死锁的四个条件中的其中一个就可以了。

破坏互斥条件

这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。

破坏请求与保持条件

一次性申请所有的资源。

破坏不剥夺条件

占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。

破坏循环等待条件

靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

我们对线程 2 的代码修改成下面这样就不会产生死锁了。

```
newThread(()->{synchronized(resource1){
System.out.println(Thread.currentThread()+"get resource1");try{
Thread.sleep(1000);}catch(InterruptedException e){          e.printStackTrace();}
System.out.println(Thread.currentThread()+"waiting get
resource2");synchronized(resource2){
System.out.println(Thread.currentThread()+"get resource2");}}},"线程 2").start();
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

输出结果

```
Thread[线程 1,5,main]get resource1Thread[线程 1,5,main]waiting get
resource2Thread[线程 1,5,main]get resource2Thread[线程 2,5,main]get
resource1Thread[线程 2,5,main]waiting get resource2Thread[线程 2,5,main]get
resource2
```

- 1
- 2

- 3
- 4
- 5
- 6

我们分析一下上面的代码为什么避免了死锁的发生?

线程 1 首先获得到 resource1 的监视器锁，这时候线程 2 就获取不到了。然后线程 1 再去获取 resource2 的监视器锁，可以获取到。然后线程 1 释放了对 resource1、resource2 的监视器锁的占用，线程 2 获取到就可以执行了。这样就破坏了破坏循环等待条件，因此避免了死锁。

创建线程的四种方式

创建线程有哪几种方式?

创建线程有四种方式：

- 继承 Thread 类；
- 实现 Runnable 接口；
- 实现 Callable 接口；
- 使用 Executors 工具类创建线程池

继承 Thread 类

步骤

1. 定义一个Thread类的子类，重写run方法，将相关逻辑实现，run()方法就是线程要执行的业务逻辑方法
2. 创建自定义的线程子类对象
3. 调用子类实例的star()方法来启动线程

```
publicclassMyThreadextendsThread{@Overridepublicvoidrun(){
System.out.println(Thread.currentThread().getName()+" run()方法正在执行...");}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

```
publicclassTheadTest{publicstaticvoidmain(String[] args){ MyThread myThread =newMyThread(); myThread.start(); System.out.println(Thread.currentThread().getName()+" main()方法执行结束");}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

运行结果

```
main main()方法执行结束Thread-0run()方法正在执行...
```

- 1
- 2

实现 Runnable 接口

步骤

1. 定义Runnable接口实现类MyRunnable，并重写run()方法
2. 创建MyRunnable实例myRunnable，以myRunnable作为target创建Thead对象，**该Thread对象才是真正的线程对象**
3. 调用线程对象的start()方法

```
publicclassMyRunnableimplementsRunnable{@Overridepublicvoidrun(){ System.out.println(Thread.currentThread().getName()+" run()方法执行中...");}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

```
publicclassRunnableTest{publicstaticvoidmain(String[] args){        MyRunnable myRunnable =newMyRunnable();        Thread thread =newThread(myRunnable);        thread.start();        System.out.println(Thread.currentThread().getName()+" main()方法执行完成");}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

执行结果

```
main main()方法执行完成Thread-0run()方法执行中...
```

- 1
- 2

实现 Callable 接口

步骤

1. 创建实现Callable接口的类myCallable
2. 以myCallable为参数创建FutureTask对象
3. 将FutureTask作为参数创建Thread对象
4. 调用线程对象的start()方法

```
publicclassMyCallableimplementsCallable<Integer>{@Overridepublic Integer call(){System.out.println(Thread.currentThread().getName()+" call()方法执行中...");return1;}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

- 9

```
publicclassCallableTest{publicstaticvoidmain(String[] args){ FutureTask<Integer> futureTask =newFutureTask<Integer>(newMyCallable()); Thread thread =newThread(futureTask); thread.start();try{ Thread.sleep(1000); System.out.println("返回结果 "+ futureTask.get());}catch(InterruptedException e){ e.printStackTrace();}catch(ExecutionException e){ e.printStackTrace();} System.out.println(Thread.currentThread().getName()+" main()方法执行完成");}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

执行结果

```
Thread-0call()方法执行中...返回结果 1main main()方法执行完成
```

- 1
- 2
- 3

使用 Executors 工具类创建线程池

Executors提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。

主要有newFixedThreadPool，newCachedThreadPool，newSingleThreadExecutor，newScheduledThreadPool，后续详细介绍这四种线程池

```
publicclassMyRunnableimplementsRunnable{@Overridepublicvoidrun(){
System.out.println(Thread.currentThread().getName()+" run()方法执行中...");}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

```
publicclassSingleThreadExecutorTest{publicstaticvoidmain(String[] args){
ExecutorService executorService = Executors.newSingleThreadExecutor();
MyRunnable runnableTest =newMyRunnable();for(int i =0; i <5; i++){
executorService.execute(runnableTest);}        System.out.println("线程任务开始执行");
executorService.shutdown();}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

执行结果

```
线程任务开始执行pool-1-thread-1 is running...pool-1-thread-1 is running...pool-1-
thread-1 is running...pool-1-thread-1 is running...pool-1-thread-1 is running...
```

- 1
- 2
- 3
- 4
- 5
- 6

说一下 runnable 和 callable 有什么区别?

相同点

- 都是接口
- 都可以编写多线程程序
- 都采用Thread.start()启动线程

主要区别

- Runnable 接口 run 方法无返回值；Callable 接口 call 方法有返回值，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果
- Runnable 接口 run 方法只能抛出运行时异常，且无法捕获处理；Callable 接口 call 方法允许抛出异常，可以获取异常信息

注：Callalbe接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

线程的 run()和 start()有什么区别?

每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，run()方法称为线程体。通过调用Thread类的start()方法来启动一个线程。

start() 方法用于启动线程，run() 方法用于执行线程的运行时代码。run() 可以重复调用，而 start() 只能调用一次。

start()方法来启动一个线程，真正实现了多线程运行。调用start()方法无需等待run方法体代码执行完毕，可以直接继续执行其他的代码； 此时线程是处于就绪状态，并没有运行。 然后通过此Thread类调用方法run()来完成其运行状态，run()方法运行结束， 此线程终止。然后CPU再调度其它线程。

run()方法是在本线程里的，只是线程里的一个函数，而不是多线程的。 如果直接调用run()，其实就相当于是调用了一个普通函数而已，直接待用run()方法必须等待run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是run()方法。

为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法?

这是另一个非常经典的 java 多线程面试问题，而且在面试中会经常被问到。很简单，但是很多人都会答不上来!

new 一个 Thread，线程进入了新建状态。调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作。 而直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结： 调用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

什么是 Callable 和 Future?

Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值，这个返回值可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。

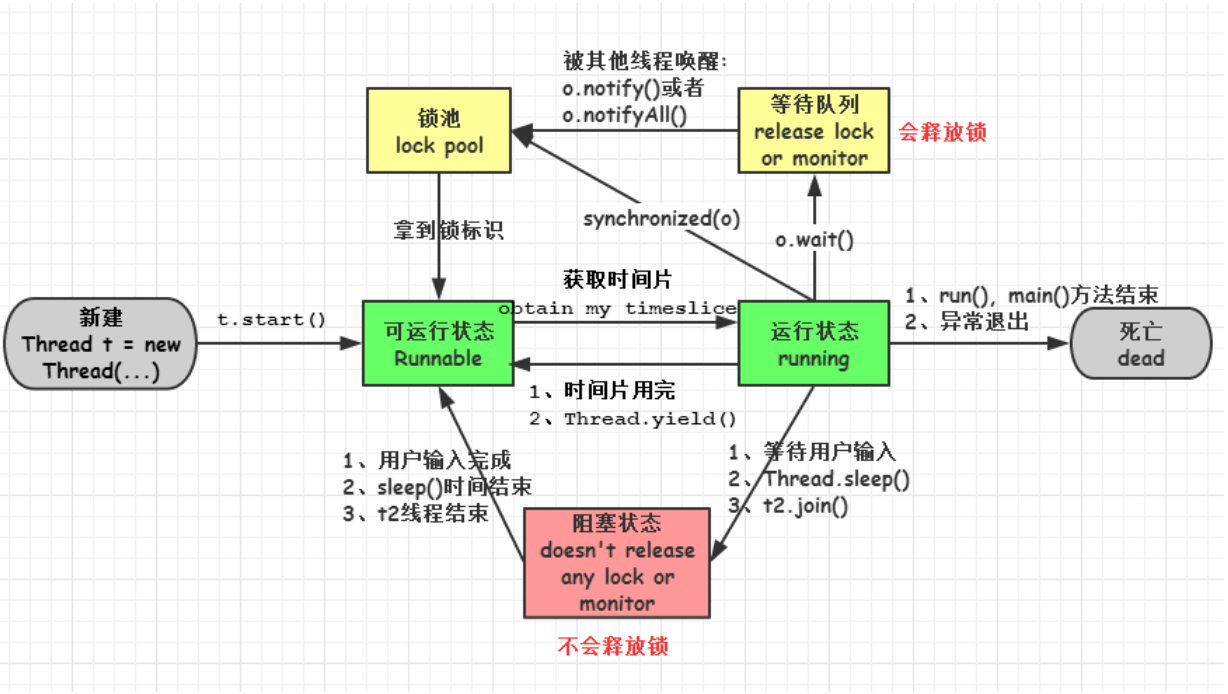
Future 接口表示异步任务，是一个可能还没有完成的异步任务的结果。所以说 Callable用于产生结果，Future 用于获取结果。

什么是 FutureTask

FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一个 Callable 的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

线程的状态和基本操作

说说线程的生命周期及五种基本状态?



线程的基本状态

1. 新建(new)：新创建了一个线程对象。

2. 可运行(runnable)：线程对象创建后，当调用线程对象的 start()方法，该线程处于就绪状态，等待被线程调度选中，获取cpu的使用权。

3. 运行(running)：可运行状态(runnable)的线程获得了cpu时间片（timeslice），执行程序代码。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

4. 阻塞(block)：处于运行状态中的线程由于某种原因，暂时放弃对 CPU 的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被 CPU 调用以进入到运行状态。

阻塞的情况分三种：

(一). 等待阻塞：运行状态中的线程执行 wait()方法，JVM会把该线程放入等待队列(waitting queue)中，使本线程进入到等待阻塞状态；

(二). 同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，，则JVM会把该线程放入锁池(lock pool)中，线程会进入同步阻塞状态；

(三). 其他阻塞: 通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。

5. 死亡(dead)：线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

Java 中用到的线程调度算法是什么?

计算机通常只有一个 CPU，在任意时刻只能执行一条机器指令，每个线程只有获得CPU 的使用权才能执行指令。所谓多线程的并发运行，其实是指从宏观上看，各个线程轮流获得 CPU 的使用权，分别执行各自的任务。在运行池中，会有多个处于就绪状态的线程在等待 CPU，JAVA 虚拟机的一项任务就是负责线程的调度，线程调度是指按照特定机制为多个线程分配 CPU 的使用权。

有两种调度模型：分时调度模型和抢占式调度模型。

分时调度模型是指让所有的线程轮流获得 cpu 的使用权，并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。

Java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用 CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用 CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

线程的调度策略

线程调度器选择优先级最高的线程运行，但是，如果发生以下情况，就会终止线程的运行：

（1）线程体中调用了 yield 方法让出了对 cpu 的占用权利

（2）线程体中调用了 sleep 方法使线程进入睡眠状态

（3）线程由于 IO 操作受到阻塞

（4）另外一个更高优先级线程出现

（5）在支持时间片的系统中，该线程的时间片用完

什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)?

线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。

时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。

线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

请说出与线程同步以及线程调度相关的方法。

（1） 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；

（2）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；

（3）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；

（4）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；

sleep() 和 wait() 有什么区别?

两者都可以暂停线程的执行

- 类的不同：sleep() 是 Thread线程类的静态方法，wait() 是 Object类的方法。
- 是否释放锁：sleep() 不释放锁；wait() 释放锁。
- 用途不同：Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。
- 用法不同：wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用wait(long timeout)超时后线程会自动苏醒。

你是如何调用 wait() 方法的？使用 if 块还是循环？为什么?

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

```
synchronized(monitor){// 判断条件谓词是否得到满足while(!locked){// 等待唤醒
monitor.wait();}// 处理其他的业务逻辑}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

- 8

为什么线程通信的方法 wait(), notify()和 notifyAll()被定义在 Object 类里?

Java中，任何对象都可以作为锁，并且 wait()，notify()等方法用于等待对象的锁或者唤醒线程，在 Java 的线程中并没有可供任何对象使用的锁，所以任意对象调用方法一定定义在Object类中。

wait(), notify()和 notifyAll()这些方法在同步代码块中调用

有的人会说，既然是线程放弃对象锁，那也可以把wait()定义在Thread类里面啊，新定义的线程继承于Thread类，也不需要重新定义wait()方法的实现。然而，这样做有一个非常大的问题，一个线程完全可以持有很多锁，你一个线程放弃锁的时候，到底要放弃哪个锁？当然了，这种设计并不是不能实现，只是管理起来更加复杂。

综上所述，wait()、notify()和notifyAll()方法要定义在Object类中。

为什么 wait(), notify()和 notifyAll()必须在同步方法或者同步块中被调用?

当一个线程需要调用对象的 wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的 notify()方法。同样的，当一个线程需要调用对象的 notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

Thread 类中的 yield 方法有什么作用?

使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。

当前线程到了就绪状态，那么接下来哪个线程会从就绪状态变成执行状态呢？可能是当前线程，也可能是其他线程，看系统的分配了。

为什么 Thread 类的 sleep()和 yield ()方法是静态的?

Thread 类的 sleep()和 yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静

态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

线程的 sleep()方法和 yield()方法有什么区别?

（1） sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

（2） 线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；

（3）sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；

（4）sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性，通常不建议使用yield()方法来控制并发线程的执行。

如何停止一个正在运行的线程?

在java中有以下3种方法可以终止正在运行的线程：

1. 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
2. 使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
3. 使用interrupt方法中断线程。

Java 中 interrupted 和 isInterrupted 方法的区别?

interrupt：用于中断线程。调用该方法的线程的状态为将被置为”中断”状态。
注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（也就是线程中断后会抛出 interruptedException 的方法）就是在监视线程的中断状态，一旦线程的中断状态被置为“中断状态”，就会抛出中断异常。

interrupted：是静态方法，查看当前中断信号是true还是false并且清除中断信号。如果一个线程被中断了，第一次调用 interrupted 则返回 true，第二次和后面的就返回 false 了。

isInterrupted：查看当前中断信号是true还是false

什么是阻塞式方法?

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，ServerSocket的accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

Java 中你怎样唤醒一个阻塞的线程?

首先 ，wait()、notify() 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取该对象的锁，直到获取成功才能往下执行；

其次，wait、notify 方法必须在 synchronized 块或方法中被调用，并且要保证同步块或方法的锁对象与调用 wait、notify 方法的对象是同一个，如此一来在调用 wait 之前当前线程就已经成功获取某对象的锁，执行 wait 阻塞后当前线程就将之前获取的对象锁释放。

notify() 和 notifyAll() 有什么区别?

如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

notifyAll() 会唤醒所有的线程，notify() 只会唤醒一个线程。

notifyAll() 调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争。而notify()只会唤醒一个线程，具体唤醒哪一个线程由虚拟机控制。

如何在两个线程间共享数据?

在两个线程间共享变量即可实现共享。

一般来说，共享变量要求变量本身是线程安全的，然后在线程内使用的时候，如果有对共享变量的复合操作，那么也得保证复合操作的线程安全性。

Java 如何实现多线程之间的通讯和协作?

可以通过中断 和 共享变量的方式实现线程间的通讯和协作

比如说最经典的生产者-消费者模型：当队列满时，生产者需要等待队列有空间才能继续往里面放入商品，而**在等待的期间内，生产者必须释放对临界资源（即队列）的占用权。**因为生产者如果不释放对临界资源的占用权，那么**消费者就无法消费队列中的商品**，就不会让队列有空间，那么生产者就会一直无限等待下去。

因此，一般情况下，当队列满时，会让生产者交出对临界资源的占用权，并进入挂起状态。然后等待消费者消费了商品，然后消费者通知生产者队列有空间了。同样地，当队列空时，消费者也必须等待，等待生产者通知它队列中有商品了。这种互相通信的过程就是线程间的协作。

Java中线程通信协作的最常见的两种方式：

一.syncrhoized加锁的线程的**Object类**的wait()/notify()/notifyAll()

二.ReentrantLock类加锁的线程的**Condition类的**await()/signal()/signalAll()

线程间直接的数据交换：

三.通过管道进行线程间通信：1）字节流；2）字符流

同步方法和同步块，哪个是更好的选择?

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

请知道一条原则：同步的范围越小越好。

什么是线程同步和线程互斥，有哪几种实现方式?

当一个线程对共享的数据进行操作时，应使之成为一个”原子操作“，即在没有完成相关操作之前，不允许其他线程打断它，否则，就会破坏数据的完整性，必然会得到错误的处理结果，这就是线程的同步。

在多线程应用中，考虑不同线程之间的数据同步和防止死锁。当两个或多个线程之间同时等待对方释放资源的时候就会形成线程之间的死锁。为了防止死锁的发生，需要通过同步来实现线程安全。

线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

实现线程同步的方法

- 同步代码方法：sychronized 关键字修饰的方法
- 同步代码块：sychronized 关键字修饰的代码块
- 使用特殊变量域volatile实现线程同步：volatile关键字为域变量的访问提供了一种免锁机制
- 使用重入锁实现线程同步：reentrantlock类是可冲入、互斥、实现了lock接口的锁他与sychronized方法具有相同的基本行为和语义

在监视器(Monitor)内部，是如何做线程同步的？程序应该做哪种级别的同步?

在 java 虚拟机中，每个对象(Object 和 class)通过某种逻辑关联监视器,每个监视器和一个对象引用相关联，为了实现监视器的互斥功能，每个对象都关联着一把锁。

一旦方法或者代码块被 **synchronized** 修饰，那么这个部分就放入了监视器的监视区域，**确保一次只能有一个线程执行该部分的代码**，线程在获取锁之前不允许执行该部分的代码

另外 java 还提供了显式监视器(Lock)和隐式监视器(synchronized)两种锁方案

如果你提交任务时，线程池队列已满，这时会发生什么

这里区分一下：

（1）如果使用的是无界队列 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务

（2）如果使用的是有界队列比如 ArrayBlockingQueue，任务首先会被添加到 ArrayBlockingQueue 中，ArrayBlockingQueue 满了，会根据 maximumPoolSize 的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue 继续满，那么则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy

什么叫线程安全？servlet 是线程安全吗?

线程安全是编程中的术语，指某个方法在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。
Servlet 不是线程安全的，servlet 是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。
Struts2 的 action 是多实例多线程的，是线程安全的，每个请求过来都会 new 一个新的 action 分配给这个请求，请求完成后销毁。
SpringMVC 的 Controller 是线程安全的吗？不是的，和 Servlet 类似的处理流程。
Struts2 好处是不用考虑线程安全问题；Servlet 和 SpringMVC 需要考虑线程安全问题，但是性能可以提升不用处理太多的 gc，可以使用 ThreadLocal 来处理多线程的问题。

在 Java 程序中怎么保证多线程的运行安全？

- 方法一：使用安全类，比如 java.util.concurrent 下的类，使用原子类 AtomicInteger
- 方法二：使用自动锁 synchronized。
- 方法三：使用手动锁 Lock。

手动锁 Java 示例代码如下：

```
Lock lock = new ReentrantLock();lock. lock();try {     System. out. println("获得锁");} catch (Exception e) {     // TODO: handle exception} finally {     System. out. println("释放锁");     lock. unlock();}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个 int 变量(从 1-10)，1 代表最低优先级，10 代表最高优先级。

Java 的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

线程类的构造方法、静态块是被哪个线程调用的

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了 Thread1，main 函数中 new 了 Thread2，那么：

（1）Thread2 的构造方法、静态块是 main 线程调用的，Thread2 的 run()方法是Thread2 自己调用的

（2）Thread1 的构造方法、静态块是 Thread2 调用的，Thread1 的 run()方法是Thread1 自己调用的

Java 中怎么获取一份线程 dump 文件？你如何在 Java 中获取线程堆栈?

Dump文件是进程的内存镜像。可以把程序的执行状态通过调试器保存到dump文件中。

在 Linux 下，你可以通过命令 kill -3 PID （Java 进程的进程 ID）来获取 Java应用的 dump 文件。

在 Windows 下，你可以按下 Ctrl + Break 来获取。这样 JVM 就会将线程的 dump 文件打印到标准输出或错误文件中，它可能打印在控制台或者日志文件中，具体位置依赖应用的配置。

一个线程运行时发生异常会怎样?

如果异常没有被捕获该线程将会停止执行。

Thread.UncaughtExceptionHandler是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候，JVM 会使用

Thread.getUncaughtExceptionHandler()来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给 handler 的 uncaughtException()方法进行处理。

Java 线程数过多会造成什么异常?

- 线程的生命周期开销非常高
- 消耗过多的 CPU

资源如果可运行的线程数量多于可用处理器的数量，那么有线程将会被闲置。大量空闲的线程会占用许多内存，给垃圾回收器带来压力，而且大量的线程在竞争CPU资源时还将产生其他性能的开销。

- 降低稳定性JVM

在可创建线程的数量上存在一个限制，这个限制值将随着平台的不同而不同，并且承受着多个因素制约，包括 JVM 的启动参数、Thread 构造函数中请求栈的大小，以及底层操作系统对线程的限制等。如果破坏了这些限制，那么可能抛出 OutOfMemoryError 异常。

并发理论

Java内存模型

Java中垃圾回收有什么目的？什么时候进行垃圾回收?

垃圾回收是在内存中存在没有引用的对象或超过作用域的对象时进行的。
垃圾回收的目的是识别并且丢弃应用不再使用的对象来释放和重用资源。

如果对象的引用被置为null，垃圾收集器是否会立即释放对象占用的内存?

不会，在下一个垃圾回调周期中，这个对象将是被可回收的。
也就是说并不会立即被垃圾收集器立刻回收，而是在下一次垃圾回收时才会释放其占用的内存。

finalize()方法什么时候被调用？析构函数(finalization)的目的是什么?

1）垃圾回收器（garbage colector）决定回收某对象时，就会运行该对象的 finalize()方法；

finalize是Object类的一个方法，该方法在Object类中的声明protected void finalize() throws Throwable { }
在垃圾回收器执行时会调用被回收对象的finalize()方法，可以覆盖此方法来实现对其资源的回收。注意：一旦垃圾回收器准备释放对象占用的内存，将首先调用该对象的finalize()方法，并且下一次垃圾回收动作发生时，才真正回收对象占用的内存空间
2）GC本来就是内存回收了，应用还需要在finalization做什么呢？ 答案是大部分时候，什么都不用做(也就是不需要重载)。只有在某些很特殊的情况下，比如你调用了一些native的方法(一般是C写的)，可以要在finaliztion里去调用C的释放函数。

重排序与数据依赖性

为什么代码会重排序?

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

- 在单线程环境下不能改变程序运行的结果；
- 存在数据依赖关系的不允许重排序

需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

as-if-serial规则和happens-before规则的区别

- as-if-serial语义保证单线程内程序的执行结果不被改变，happens-before关系保证正确同步的多线程程序的执行结果不被改变。
- as-if-serial语义给编写单线程程序的程序员创造了一个幻境：单线程程序是按程序的顺序来执行的。happens-before关系给编写正确同步的多线程程序的程序员创造了一个幻境：正确同步的多线程程序是按happens-before指定的顺序来执行的。
- as-if-serial语义和happens-before这么做的目的，都是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

并发关键字

synchronized

synchronized 的作用?

在 Java 中，synchronized 关键字是用来控制线程同步的，就是在多线程的环境下，控制 synchronized 代码段不被多个线程同时执行。synchronized 可以修饰类、方法、变量。

另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

说说自己是怎么使用 synchronized 关键字，在项目中用到了吗

synchronized关键字最主要的三种使用方式：

- **修饰实例方法:** 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁
- **修饰静态方法:** 也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。**
- **修饰代码块:** 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

总结： synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓存功能！

下面我以一个常见的面试题为例讲解一下 synchronized 关键字的具体使用。

面试中面试官经常会说：“单例模式了解吗？来给我手写一下！给我解释一下双重检验锁方式实现单例模式的原理呗！”

双重校验锁实现对象单例（线程安全）

```
publicclassSingleton{privatevolatilestatic Singleton uniqueInstance;privateSingleton(){}publicstatic Singleton getUniqueInstance(){//先判断对象是否已经实例过，没有实例化过才进入加锁代码if(uniqueInstance == null){//类对象加锁synchronized(Singleton.class){if(uniqueInstance == null){ uniqueInstance =newSingleton();}}}return uniqueInstance;}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance

3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

说一下 synchronized 底层实现原理?

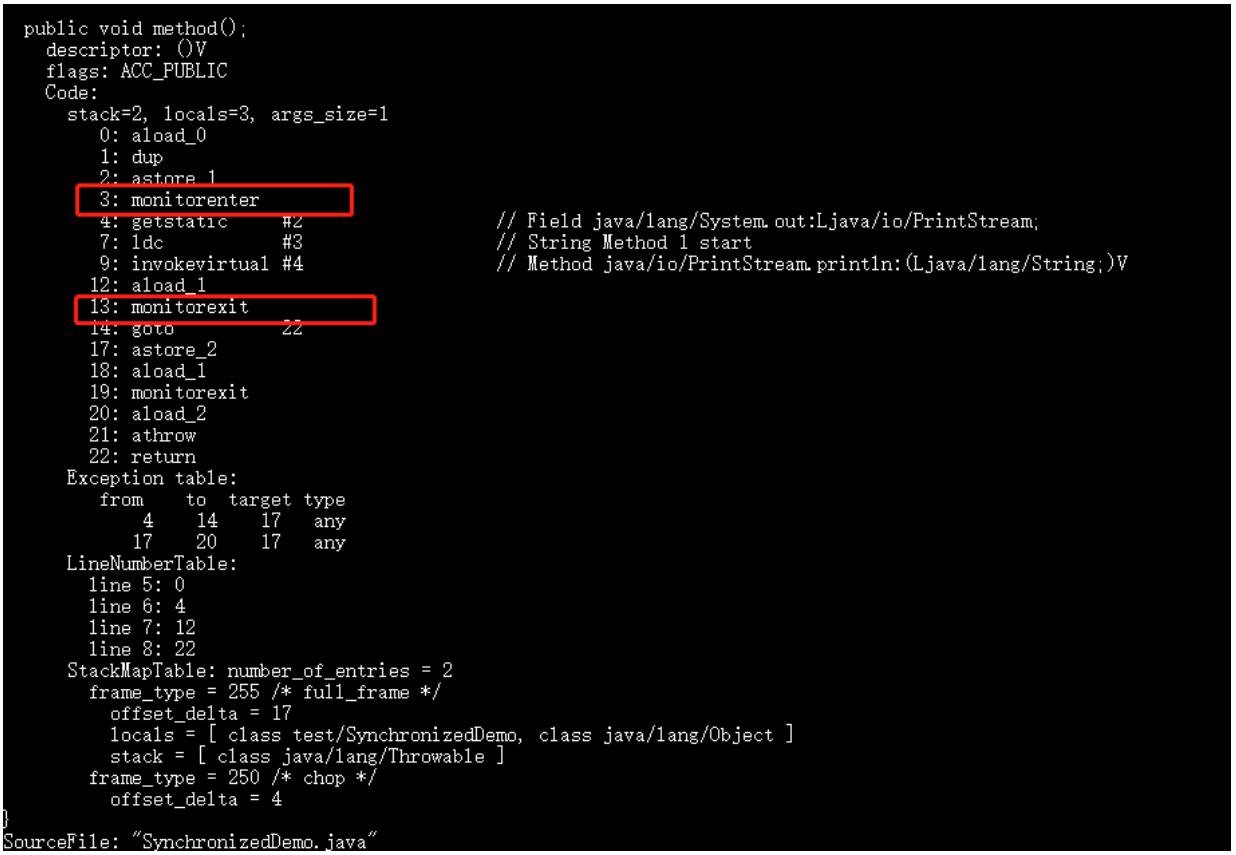
synchronized是Java中的一个关键字，在使用的过程中并没有看到显示的加锁和解锁过程。因此有必要通过javap命令，查看相应的字节码文件。

synchronized 同步语句块的情况

```
publicclassSynchronizedDemo{publicvoidmethod(){synchronized(this){
System.out.println("synchronized 代码块");}}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

通过JDK 反汇编指令 javap -c -v SynchronizedDemo



synchronized关键字原理

可以看出在执行同步代码块之前之后都有一个monitor字样，其中前面的是monitorenter，后面的是离开monitorexit，不难想象一个线程也执行同步代码块，首先要获取锁，而获取锁的过程就是monitorenter ，在执行完代码块之后，要释放锁，释放锁就是执行monitorexit指令。

为什么会有两个monitorexit呢?

这个主要是防止在同步代码块中线程因异常退出，而锁没有得到释放，这必然会造成死锁（等待的线程永远获取不到锁）。因此最后一个monitorexit是保证在异常情况下，锁也可以得到释放，避免死锁。

仅有ACC_SYNCHRONIZED这么一个标志，该标记表明线程进入该方法时，需要monitorenter，退出该方法时需要monitorexit。

synchronized可重入的原理

重入锁是指一个线程获取到该锁之后，该线程可以继续获得该锁。底层原理维护一个计数器，当线程获取该锁时，计数器加一，再次获得该锁时继续加一，释放锁时，计数器减一，当计数器值为0时，表明该锁未被任何线程所持有，其它线程可以竞争获取锁。

什么是自旋

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

多线程中 synchronized 锁升级的原理是什么?

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

线程 B 怎么知道线程 A 修改了变量

（1）volatile 修饰变量

（2）synchronized 修饰修改变量的方法

（3）wait/notify

（4）while 轮询

当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B?

不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的 synchronized 修饰符要求执行方法时要获得对象的锁，如果已经进入A 方法说明对象锁已经被取走，那么试图进入 B 方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

synchronized、volatile、CAS 比较

（1）synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。

（2）volatile 提供多线程共享变量可见性和禁止指令重排序优化。

（3）CAS 是基于冲突检测的乐观锁（非阻塞）

synchronized 和 Lock 有什么区别?

- 首先synchronized是Java内置关键字，在JVM层面，Lock是个Java类；
- synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。
- synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。
- 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

synchronized 和 ReentrantLock 区别是什么?

synchronized 是和 if、else、for、while 一样的关键字，ReentrantLock 是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比 synchronized 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量

synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大，但是在 Java 6 中对 synchronized 进行了非常多的改进。

相同点：两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

主要区别如下：

- ReentrantLock 使用起来比较灵活，但是必须有释放锁的配合动作；
- ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动释放和开启锁；
- ReentrantLock 只适用于代码块锁，而 synchronized 可以修饰类、方法、变量等。
- 二者的锁机制其实也是不一样的。ReentrantLock 底层调用的是 Unsafe 的park 方法加锁，synchronized 操作的应该是对象头中 mark word

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象

- 同步方法块，锁是括号里面的对象

volatile

volatile 关键字的作用

对于可见性，Java 提供了 volatile 关键字来保证可见性和禁止指令重排。
volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。
从实践角度而言，volatile 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。
volatile 常用于多线程环境下的单次操作(单次读或者单次写)。

Java 中能创建 volatile 数组吗?

能，Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。意思是，如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

volatile 变量和 atomic 变量有什么不同?

volatile 变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用 volatile 修饰 count 变量，那么 count++ 操作就不是原子性的。
而 AtomicInteger 类提供的 atomic 方法可以让这种操作具有原子性如 getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

volatile 能使得一个非原子操作变成原子操作吗?

关键字volatile的主要作用是使变量在多个线程间可见，但无法保证原子性，对于多个线程访问同一个实例变量需要加锁进行同步。
虽然volatile只能保证可见性不能保证原子性，但用volatile修饰long和double可以保证其操作原子性。
所以从Oracle Java Spec里面可以看到：

- 对于64位的long和double，如果没有被volatile修饰，那么对其操作可以不是原子的。在操作的时候，可以分成两步，每次对32位操作。

- 如果使用volatile修饰long和double，那么其读写都是原子操作
- 对于64位的引用地址的读写，都是原子操作
- 在实现JVM时，可以自由选择是否把读写long和double作为原子操作
- 推荐JVM实现为原子操作

volatile 修饰符的有过什么实践?

单例模式

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：对于Double-Check这种可能出现的问题（当然这种概率已经非常小了，但毕竟还是有的嘛~），解决方案是：只需要给instance的声明加上volatile关键字即可volatile关键字的一个作用是禁止指令重排，把instance声明为volatile之后，对它的写操作就会有一个内存屏障（什么是内存屏障？），这样，在它的赋值完成之前，就不用会调用读操作。注意：volatile阻止的不是singleton = newSingleton()这句话内部[1-2-3]的指令重排，而是保证了在一个写操作（[1-2-3]）完成之前，不会调用读操作（if (instance == null)）。

```
publicclassSingleton7{privatestaticvolatile Singleton7 instance = null;privateSingleton7(){}publicstatic Singleton7 getInstance(){if(instance == null){synchronized(Singleton7.class){if(instance == null){                instance =newSingleton7();}}}return instance;}}
```

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13

- 14
- 15
- 16
- 17
- 18
- 19

synchronized 和 volatile 的区别是什么?

synchronized 表示只有一个线程可以获取作用对象的锁，执行代码，阻塞其他线程。

volatile 表示变量在 CPU 的寄存器中是不确定的，必须从主存中读取。保证多线程环境下变量的可见性；禁止指令重排序。

区别

- volatile 是变量修饰符；synchronized 可以修饰类、方法、变量。
- volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
- 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；synchronized标记的变量可以被编译器优化。
- **volatile关键字**是线程同步的**轻量级实现**，所以**volatile性能肯定比synchronized关键字要好。**但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。**synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些。**

final

什么是不可变对象，它对写并发应用有什么帮助?

不可变对象(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象(Mutable Objects)。

不可变对象的类即为不可变类(Immutable Class)。Java 平台类库中包含许多不可变类，如 String、基本类型的包装类、BigInteger 和 BigDecimal 等。

只有满足如下状态，一个对象才是不可变的；

- 它的状态不能在创建后再被修改；
- 所有域都是 final 类型；并且，它被正确创建（创建期间没有发生 this 引用的逸出）。

不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

Lock体系

Lock简介与初识AQS

Java Concurrency API 中的 Lock 接口(Lock interface)是什么？对比同步它有什么优势?

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

（1）可以使锁更公平

（2）可以使线程在等待锁的时候响应中断

（3）可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

（4）可以在不同的范围，以不同的顺序获取和释放锁

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

乐观锁和悲观锁的理解及如何实现，有哪些实现方式?

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。在Java中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

乐观锁的实现方式：

1、使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

2、java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。 CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置 V 的值与预期原值 A 相匹配，那么处理器会自动将该位置值更新为新值 B。否则处理器不做任何操作。

什么是 CAS

CAS 是 compare and swap 的缩写，即我们所说的比较交换。

cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的(AtomicInteger,AtomicBoolean,AtomicLong)。

CAS 的会产生什么问题?

1、ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

2、循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

3、只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

什么是死锁?

当线程 A 持有独占锁a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

产生死锁的条件是什么？怎么防止死锁?

产生死锁的必要条件：

1、互斥条件：所谓互斥就是进程在某一时间内独占资源。

2、请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3、不剥夺条件：进程已获得资源，在未使用完之前，不能强行剥夺。

4、循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之 一不满足，就不会发生死锁。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和 解除死锁。

防止死锁可以采用以下的方法：

- 尽量使用 tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。

- 尽量使用 Java. util. concurrent 并发类代替自己手写锁。
- 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。
- 尽量减少同步的代码块。

死锁与活锁的区别，死锁与饥饿的区别?

死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，这就是所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java 中导致饥饿的原因：

1、高优先级线程吞噬所有的低优先级线程的 CPU 时间。

2、线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。

3、线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)，因为其他线程总是被持续地获得唤醒。

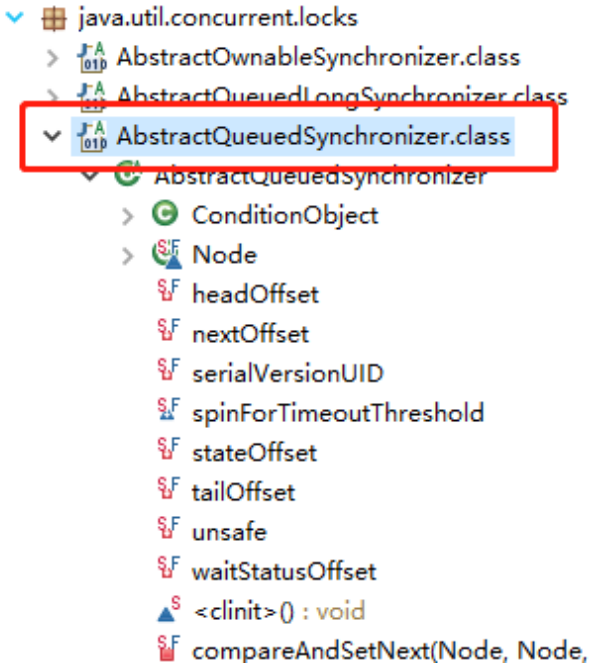
多线程锁的升级原理是什么?

在Java中，锁共有4种状态，级别从低到高依次为：无状态锁，偏向锁，轻量级锁和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级。

AQS(AbstractQueuedSynchronizer)详解与源码分析

AQS 介绍

AQS的全称为（AbstractQueuedSynchronizer），这个类在java.util.concurrent.locks包下面。



AQS类

AQS是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是基于AQS的。当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

AQS 原理分析

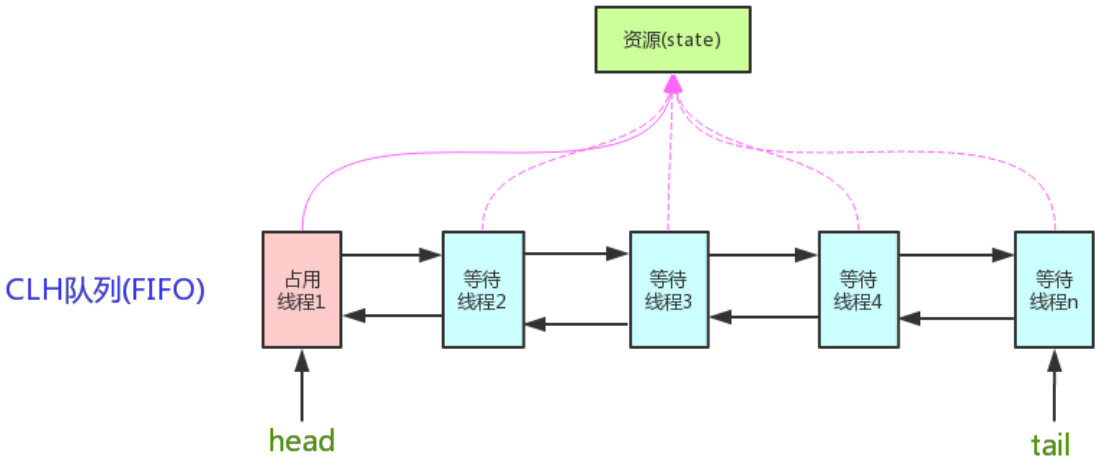
下面大部分内容其实在AQS类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

AQS 原理概览

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

> CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁的分配。

看个AQS(AbstractQueuedSynchronizer)原理图：



AQS原理图

AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
```

- 1

状态信息通过protected类型的getState，setState，compareAndSetState进行操作

```
//返回同步状态的当前值protectedfinalintgetState(){return state;}// 设置同步状态的值
protectedfinalvoidsetState(int newState){        state = newState;}//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）
protectedfinalbooleancompareAndSetState(int expect,int update){return
unsafe.compareAndSwapInt(this, stateOffset, expect, update);}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

- 12

AQS 对资源的共享方式

AQS定义两种资源共享方式

- Exclusive（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：
  - 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
  - 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
- **Share**（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。Semaphore、CountDownLatch、CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式，因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。

AQS底层使用了模板方法模式

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）
2. 将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法：

isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。
tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。
tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。
tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩

余可用资源；正数表示成功，且有剩余资源。tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。

- 1
- 2
- 3
- 4
- 5
- 6

默认情况下，每个方法都抛出 UnsupportedOperationException。 这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS类中的其他方法都是final ，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS(Compare and Swap)减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

ReentrantLock(重入锁)实现原理与公平锁非公平锁区别

什么是可重入锁（ReentrantLock）？

ReentrantLock重入锁，是实现Lock接口的一个类，也是在实际编程中使用频率很高的一个锁，支持重入性，表示能够对共享资源能够重复加锁，即当前线程获取该锁再次获取不会被阻塞。

在java关键字synchronized隐式支持重入性，synchronized通过获取自增，释放自减的方式实现重入。与此同时，ReentrantLock还支持公平锁和非公平锁两种方式。那么，要想完完全全的弄懂ReentrantLock的话，主要也就是ReentrantLock同步语义的学习：1. 重入性的实现原理；2. 公平锁和非公平锁。

重入性的实现原理

要想支持重入性，就要解决两个问题：**1. 在线程获取锁的时候，如果已经获取锁的线程是当前线程的话则直接再次获取成功；2. 由于锁会被获取n次，那么只有锁在被释放同样的n次之后，该锁才算是完全释放成功。**

ReentrantLock支持两种锁：**公平锁**和**非公平锁。何谓公平性，是针对获取锁而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求上的绝对时间顺序，满足FIFO。**

读写锁ReentrantReadWriteLock源码分析

ReadWriteLock 是什么

首先明确一下，不是说 ReentrantLock 不好，只是 ReentrantLock 某些时候有局限。如果使用 ReentrantLock，可能本身是为了防止线程 A 在写数据、线程 B 在读数据造成的数据不一致，但这样，如果线程 C 在读数据、线程 D 也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。因为这个，才诞生了读写锁 ReadWriteLock。

ReadWriteLock 是一个读写锁接口，读写锁是用来提升并发程序性能的锁分离技术，ReentrantReadWriteLock 是 ReadWriteLock 接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

而读写锁有以下三个重要的特性：

（1）公平选择性：支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平。

（2）重进入：读锁和写锁都支持线程重进入。

（3）锁降级：遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级成为读锁。

Condition源码分析与等待通知机制

LockSupport详解

并发容器

并发容器之ConcurrentHashMap详解(JDK1.8版本)与源码分析

什么是ConcurrentHashMap?

ConcurrentHashMap是Java中的一个**线程安全且高效的HashMap实现。**平时涉及高并发如果要用map结构，那第一时间想到的就是它。相对于hashmap来说，ConcurrentHashMap就是线程安全的map，其中利用了锁分段的思想提高了并发度。

那么它到底是如何实现线程安全的?

JDK 1.6版本关键要素:

- segment继承了ReentrantLock充当锁的角色，为每一个segment提供了线程安全的保障;
- segment维护了哈希散列表的若干个桶，每个桶由HashEntry构成的链表。

JDK1.8后，ConcurrentHashMap抛弃了原有的**Segment 分段锁，而采用了CAS + synchronized 来保证并发安全性。**

Java 中 ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是 ConcurrentHashMap 类构造函数的一个可选参数，默认值为 16，这样在多线程情况下就能避免争用。

在 JDK8 后，它摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现,利用 CAS 算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

什么是并发容器的实现?

何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如 Vector，Hashtable，以及 Collections.synchronizedSet，synchronizedList 等方法返

回的容器。可以通过查看 Vector，Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 synchronized。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性，例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制，可以称为分段锁，在这种锁机制下，允许任意数量的读线程并发地访问 map，并且执行读操作的线程和写操作的线程也可以并发的访问 map，同时允许一定数量的写操作线程并发地修改 map，所以它可以在并发环境下实现更高的吞吐量。

Java 中的同步集合与并发集合有什么区别?

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在 Java1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5 介绍了并发集合像 ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

SynchronizedMap 和 ConcurrentHashMap 有什么区别?

SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map。

ConcurrentHashMap 使用分段锁来保证在多线程下的性能。

ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get，put，remove 等常用操作只锁当前需要用到的桶。

这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。

另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，当iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在改变时 new 新的数据从而不影响原有的数据，iterator 完成后再将头指针替换为新的数据 ，这样 iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。

并发容器之CopyOnWriteArrayList详解

CopyOnWriteArrayList 是什么，可以用于什么应用场景？有哪些优缺点?

CopyOnWriteArrayList 是一个并发容器。有很多人称它是线程安全的，我认为这句话不严谨，缺少一个前提条件，那就是非复合场景下操作它是线程安全的。CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException。在 CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

CopyOnWriteArrayList 的使用场景

通过源码分析，我们看出它的优缺点比较明显，所以使用场景也就比较明显。就是合适读多写少的场景。

CopyOnWriteArrayList 的缺点

1. 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 young gc 或者 full gc。
2. 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的，虽然CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求。
3. 由于实际使用中可能没法保证 CopyOnWriteArrayList 到底要放置多少数据，万一数据稍微有点多，每次 add/set 都要重新复制数组，这个代价实在太高昂了。在高性能的互联网应用中，这种操作分分钟引起故障。

CopyOnWriteArrayList 的设计思想

1. 读写分离，读和写分开
2. 最终一致性
3. 使用另外开辟空间的思路，来解决并发冲突

并发容器之ThreadLocal详解

ThreadLocal 是什么？有哪些使用场景?

ThreadLocal 是一个本地线程副本变量工具类，在每个线程中都创建了一个 ThreadLocalMap 对象，简单说 ThreadLocal 就是一种以空间换时间的做法，每个线程可以访问自己内部 ThreadLocalMap 对象内的 value。通过这种方式，避免资源在多线程间共享。

原理：线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

经典的使用场景是为每个线程分配一个 JDBC 连接 Connection。这样就可以保证每个线程的都在各自的 Connection 上进行数据库的操作，不会出现 A 线程关了 B线程正在使用的 Connection； 还有 Session 管理 等问题。

ThreadLocal 使用例子：

```
publicclassTestThreadLocal{//线程本地存储变量privatestaticfinal
ThreadLocal<Integer> THREAD_LOCAL_NUM =newThreadLocal<Integer>()
{@Overrideprotected Integer initialValue(){return0;}};publicstaticvoidmain(String[]
args){for(int i =0; i <3; i++){//启动三个线程 Thread t =newThread()
{@Overridepublicvoidrun(){add10ByThreadLocal();}}; t.start();}}/** * 线程本地
存储变量加 5 */privatestaticvoidadd10ByThreadLocal(){for(int i =0; i <5; i++){
Integer n = THREAD_LOCAL_NUM.get(); n +=1;
THREAD_LOCAL_NUM.set(n);
System.out.println(Thread.currentThread().getName()+" : ThreadLocal num="+ n);}}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36

打印结果：启动了 3 个线程，每个线程最后都打印到 “ThreadLocal num=5”，而不是 num 一直在累加直到值等于 15

Thread-0: ThreadLocal num=1Thread-1: ThreadLocal num=1Thread-0: ThreadLocal num=2Thread-0: ThreadLocal num=3Thread-1: ThreadLocal num=2Thread-2: ThreadLocal num=1Thread-0: ThreadLocal num=4Thread-2: ThreadLocal num=2Thread-1: ThreadLocal num=3Thread-1: ThreadLocal num=4Thread-2: ThreadLocal num=3Thread-0: ThreadLocal num=5Thread-2: ThreadLocal num=4Thread-2: ThreadLocal num=5Thread-1: ThreadLocal num=5

- 1
- 2
- 3
- 4
- 5
- 6

- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

什么是线程局部变量?

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java 提供 ThreadLocal 类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

ThreadLocal内存泄漏分析与解决方案

ThreadLocal造成内存泄漏的原因?

ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用,而 value 是强引用。所以，如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。这样一来，ThreadLocalMap 中就会出现key为null的Entry。假如我们不做任何措施的话，value 永远无法被GC 回收，这个时候就可能会产生内存泄露。ThreadLocalMap实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。使用完ThreadLocal方法后 最好手动调用remove()方法

ThreadLocal内存泄漏解决方案?

- 每次使用完ThreadLocal，都调用它的remove()方法，清除数据。
- 在使用线程池的情况下，没有及时清理ThreadLocal，不仅是内存泄漏的问题，更严重的是可能导致业务逻辑出现问题。所以，使用ThreadLocal就跟加锁完要解锁一样，用完就清理。

并发容器之BlockingQueue详解

什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。
这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。
阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。
JDK7 提供了 7 个阻塞队列。分别是：
ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列。
LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列。
PriorityBlockingQueue ： 一个支持优先级排序的无界阻塞队列。
DelayQueue：一个使用优先级队列实现的无界阻塞队列。
SynchronousQueue：一个不存储元素的阻塞队列。
LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。
Java 5 之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好，wait,notify,notifyAll,sychronized 这些关键字。而在 java 5 之后，可以使用阻塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。
BlockingQueue 接口是 Queue 的子接口，它的主要用途并不是作为容器，而是作为线程同步的的工具，因此他具有一个很明显的特性，当生产者线程试图向 BlockingQueue 放入元素时，如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向 BlockingQueue 中放入元素，取出元素，它可以很好的控制线程之间的通信。
阻塞队列使用最经典的场景就是 socket 客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

并发容器之ConcurrentLinkedQueue详解与源码分析

并发容器之ArrayBlockingQueue与LinkedBlockingQueue详解

线程池

Executors类创建四种常见线程池

什么是线程池？有哪几种创建方式?

> 池化技术相比大家已经屡见不鲜了，线程池、数据库连接池、Http 连接池等等都是对这个思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在 Java 中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。

线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。Java 5+中的 Executor 接口定义一个执行线程的工具。它的子类型即线程池接口是 ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类 Executors 面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

（1）newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。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。

（2）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。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。如果希望在服务器上使用线程池，建议使用 newFixedThreadPool方法来创建线程池，这样能获得更好的性能。

（3） newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。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。

（4）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。

线程池有什么优点?

- 降低资源消耗：重用存在的线程，减少对象创建销毁的开销。
- 提高响应速度。可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- 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。
- 附加功能：提供定时执行、定期执行、单线程、并发数控制等功能。

综上所述使用线程池框架 Executor 能更好的管理线程、提供系统资源使用率。

线程池都有哪些状态?

- RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
- SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
- STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
- TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。
- TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

什么是 Executor 框架？为什么使用 Executor 框架?

Executor 框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

每次执行任务创建线程 new Thread()比较消耗性能，创建一个线程是比较耗时、耗资源的，而且无限制的创建线程会引起应用程序内存溢出。

所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors 框架可以非常方便的创建一个线程池。

在 Java 中 Executor 和 Executors 的区别?

- Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。
- Executor 接口对象能执行我们的线程任务。
- ExecutorService 接口继承了 Executor 接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。
- 使用 ThreadPoolExecutor 可以创建自定义线程池。
- Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成，并可以使用 get()方法获取计算的结果。

线程池中 submit() 和 execute() 方法有什么区别?

接收参数：execute()只能执行 Runnable 类型的任务。submit()可以执行 Runnable 和 Callable 类型的任务。

返回值：submit()方法可以返回持有计算结果的 Future 对象，而execute()没有

异常处理：submit()方便Exception处理

什么是线程组，为什么在 Java 中不推荐使用?

ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。

线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。

为什么不推荐使用线程组？因为使用有很多的安全隐患吧，没有具体追究，如果需要使用，推荐使用线程池。

线程池之ThreadPoolExecutor详解

Executors和ThreaPoolExecutor创建线程池的区别

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的

运行规则，规避资源耗尽的风险

Executors 各个方法的弊端：

- newFixedThreadPool 和 newSingleThreadExecutor:

主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至 OOM。

- newCachedThreadPool 和 newScheduledThreadPool:

主要问题是线程数最大数是 Integer.MAX_VALUE，可能会创建数量非常多的线程，甚至 OOM。

ThreaPoolExecutor创建线程池方式只有一种，就是走它的构造函数，参数自己指定

你知道怎么创建线程池吗?

创建线程池的方式有多种，这里你只需要答 ThreadPoolExecutor 即可。

ThreadPoolExecutor() 是最原始的线程池创建，也是阿里巴巴 Java 开发手册中明确规范的创建线程池的方式。

ThreadPoolExecutor构造函数重要参数分析

ThreadPoolExecutor3 个最重要的参数:

- **corePoolSize** ： 核心线程数，线程数定义了最小可以同时运行的线程数量。
- **maximumPoolSize** ： 线程池中允许存在的工作线程的最大数量
- **workQueue**：当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，任务就会被存放在队列中。

ThreadPoolExecutor其他常见参数:

1. **keepAliveTime**：线程池中的线程数量大于 corePoolSize 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 keepAliveTime才会被回收销毁；
2. **unit** ： keepAliveTime 参数的时间单位。
3. **threadFactory**：为线程池提供创建新线程的线程工厂
4. **handler** ： 线程池任务队列超过 maxinumPoolSize 之后的拒绝策略

ThreadPoolExecutor饱和策略

ThreadPoolExecutor饱和策略定义:

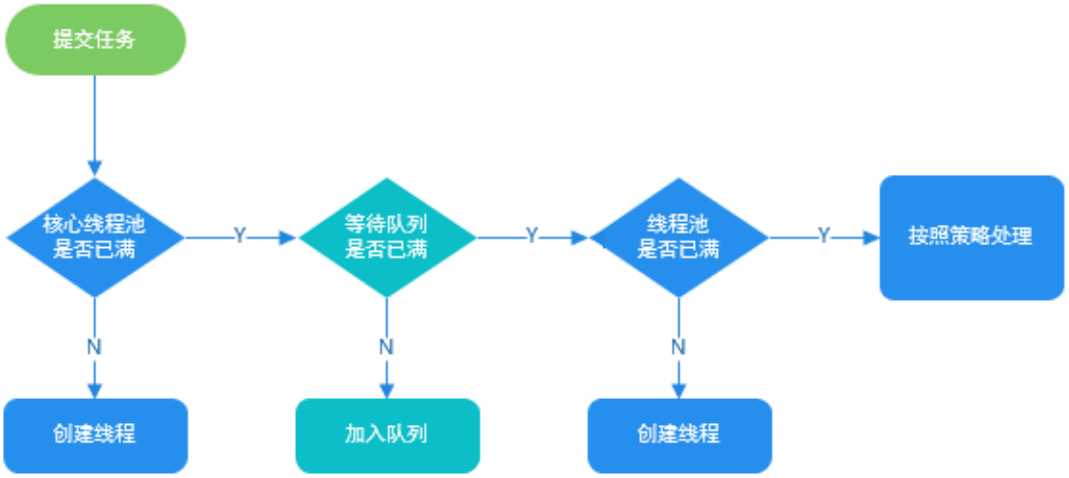
如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，ThreadPoolTaskExecutor 定义一些策略:

- **ThreadPoolExecutor.AbortPolicy**：抛出 RejectedExecutionException来拒绝新任务的处理。
- **ThreadPoolExecutor.CallerRunsPolicy**：调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。
- **ThreadPoolExecutor.DiscardPolicy**：不处理新任务，直接丢弃掉。
- **ThreadPoolExecutor.DiscardOldestPolicy**： 此策略将丢弃最早的未处理的任务请求。

举个例子： Spring 通过 ThreadPoolTaskExecutor 或者我们直接通过 ThreadPoolExecutor 的构造函数创建线程池的时候，当我们不指定 RejectedExecutionHandler 饱和策略的话来配置线程池的时候默认使用的是 ThreadPoolExecutor.AbortPolicy。在默认情况下，ThreadPoolExecutor 将抛出 RejectedExecutionException 来拒绝新来的任务 ，这代表你将丢失对这个任务的处理。 对于可伸缩的应用程序，建议使用 ThreadPoolExecutor.CallerRunsPolicy。当最大池被填满时，此策略为我们提供可伸缩队列。（这个直接查看 ThreadPoolExecutor 的构造函数源码就可以看出，比较简单的原因，这里就不贴代码了）

一个简单的线程池Demo:Runnable+ThreadPoolExecutor

线程池实现原理



图解线程池实现原理

为了让大家更清楚上面的面试题中的一些概念，我写了一个简单的线程池 Demo。

首先创建一个 Runnable 接口的实现类（当然也可以是 Callable 接口，我们上面也说了两者的区别。）

```
import java.util.Date;/** * 这是一个简单的Runnable类，需要大约5秒钟来执行其任务。 */publicclassMyRunnableimplementsRunnable{private String command;publicMyRunnable(String s){this.command = s;}@Overridepublicvoidrun(){ System.out.println(Thread.currentThread().getName()+" Start. Time = "+newDate());processCommand(); System.out.println(Thread.currentThread().getName()+" End. Time = "+newDate());}privatevoidprocessCommand(){try{ Thread.sleep(5000);}catch(InterruptedException e){ e.printStackTrace();}}@Overridepublic String toString(){returnthis.command;}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24

- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33

编写测试程序，我们这里以阿里巴巴推荐的使用 ThreadPoolExecutor 构造函数自定义参数的方式来创建线程池。

```
import java.util.concurrent.ArrayBlockingQueue;import
java.util.concurrent.ThreadPoolExecutor;import
java.util.concurrent.TimeUnit;publicclassThreadPoolExecutorDemo{privatestaticfinalint
 CORE_POOL_SIZE =5;privatestaticfinalint MAX_POOL_SIZE =10;privatestaticfinalint
QUEUE_CAPACITY =100;privatestaticfinal Long KEEP_ALIVE_TIME
=1L;publicstaticvoidmain(String[] args){//使用阿里巴巴推荐的创建线程池的方式//通过
ThreadPoolExecutor构造函数自定义参数创建        ThreadPoolExecutor executor
=newThreadPoolExecutor(              CORE_POOL_SIZE,              MAX_POOL_SIZE,
KEEP_ALIVE_TIME,              TimeUnit.SECONDS,newArrayBlockingQueue<>
(QUEUE_CAPACITY),newThreadPoolExecutor.CallerRunsPolicy());for(int i =0; i <10;
i++){//创建WorkerThread对象（WorkerThread类实现了Runnable 接口）
Runnable worker =newMyRunnable(""+ i);//执行Runnable
executor.execute(worker);}//终止线程池
executor.shutdown();while(!executor.isTerminated()){}
System.out.println("Finished all threads");}}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35

可以看到我们上面的代码指定了：

1. corePoolSize: 核心线程数为 5。
2. maximumPoolSize ： 最大线程数 10
3. keepAliveTime : 等待时间为 1L。
4. unit: 等待时间的单位为 TimeUnit.SECONDS。
5. workQueue：任务队列为 ArrayBlockingQueue，并且容量为 100;
6. handler:饱和策略为 CallerRunsPolicy。

Output：

pool-1-thread-2 Start. Time = Tue Nov 1220:59:44 CST 2019pool-1-thread-5 Start. Time = Tue Nov 1220:59:44 CST 2019pool-1-thread-4 Start. Time = Tue Nov 1220:59:44 CST 2019pool-1-thread-1 Start. Time = Tue Nov 1220:59:44 CST 2019pool-1-thread-3 Start. Time = Tue Nov 1220:59:44 CST 2019pool-1-thread-5 End. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-3 End. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-2 End. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-4 End. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-1 End. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-2 Start. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-1 Start. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-4 Start. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-3 Start. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-5 Start. Time = Tue Nov 1220:59:49 CST 2019pool-1-thread-2 End. Time = Tue Nov 1220:59:54 CST 2019pool-1-thread-3 End. Time = Tue Nov 1220:59:54 CST 2019pool-1-thread-4 End. Time = Tue Nov 1220:59:54 CST 2019pool-1-thread-5 End. Time = Tue Nov 1220:59:54 CST 2019pool-1-thread-1 End. Time = Tue Nov 1220:59:54 CST 2019

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

线程池之ScheduledThreadPoolExecutor详解

FutureTask详解

原子操作类

什么是原子操作？在 Java Concurrency API 中有哪些原子类 (atomic classes)?

原子操作（atomic operation）意为”不可被中断的一个或一系列操作” 。
处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。
在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。 CAS 操作——Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作。
原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。
int++并不是一个原子操作，所以当一个线程读取它的值并加 1 时，另外一个线程有可能会读到之前的值，这就会引发错误。
为了解决这个问题，必须保证增加操作是原子的，在 JDK1.5 之前我们可以使用同步技术来做到这一点。到 JDK1.5，java.util.concurrent.atomic 包提供了 int 和long 类型的原子包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。
java.util.concurrent 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择另一个线程进入，这只是一种逻辑上的理解。
原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference
原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray
原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个boolean来反映中间有没有变过），AtomicStampedReference（通过引入一个int 来累加来反映中间有没有变过）

说一下 atomic 的原理?

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

AtomicInteger 类的部分源码：

```
// setup to use Unsafe.compareAndSwapInt for updates（更新操作时提供“比较并替换”的作用）privatestaticfinal Unsafe unsafe = Unsafe.getUnsafe();privatestaticfinallong valueOffset;static{try{ valueOffset = unsafe.objectFieldOffset (AtomicInteger.class.getDeclaredField("value"));}catch(Exception ex) {thrownewError(ex);}}privatevolatileint value;
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。

UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile

变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

并发工具

并发工具之CountDownLatch与CyclicBarrier

在 Java 中 CycliBarriar 和 CountdownLatch 有什么区别?

CountDownLatch与CyclicBarrier都是用于控制并发的工具类，都可以理解成维护的就是一个计数器，但是这两者还是各有不同侧重点的：

- CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；CountDownLatch强调一个线程等多个线程完成某件事情。CyclicBarrier是多个线程互等，等大家都完成，再携手共进。
- 调用CountDownLatch的countDown方法后，当前线程并不会阻塞，会继续往下执行；而调用CyclicBarrier的await方法，会阻塞当前线程，直到CyclicBarrier指定的线程全部都到达了指定点的时候，才能继续往下执行；
- CountDownLatch方法比较少，操作比较简单，而CyclicBarrier提供的方法更多，比如能够通过getNumberWaiting()，isBroken()这些方法获取当前多个线程的状态，并且CyclicBarrier的构造方法可以传入barrierAction，指定当所有线程都到达时执行的业务功能；
- CountDownLatch是不能复用的，而CyclicLatch是可以复用的。

并发工具之Semaphore与Exchanger

Semaphore 有什么作用

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。

Semaphore(信号量)-允许多个线程同时访问： synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

什么是线程间交换数据的工具Exchanger

Exchanger是一个用于线程间协作的工具类，用于两个线程间交换数据。它提供了一个交换的同步点，在这个同步点两个线程能够交换数据。交换数据是通过exchange方法来实现的，如果一个线程先执行exchange方法，那么它会同步等待另一个线程也执行exchange方法，这个时候两个线程就都达到了同步点，两个线程就可以交换数据。

常用的并发工具类有哪些?

- **Semaphore(信号量)-允许多个线程同时访问：** synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。
- **CountDownLatch(倒计时器)：** CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。
- **CyclicBarrier(循环栅栏)：** CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await()方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

并发实践