# 线程和进程

1、如何在Java中实现线程(4种)？

1.继承Thread类，重写run方法（其实Thread类本身也实现了Runnable接口）

2.实现Runnable接口，重写run方法

3.实现Callable接口，重写call方法（有返回值）

4.使用线程池（有返回值）

https://www.cnblogs.com/duanjiapingjy/p/9434244.htmlhttps://www.cnblogs.com/duanjiapingjy/p/9434244.html

2、在具体多线程编程实践中，如何选用Runnable还是Thread？

  Java中实现多线程有两种方法：继承Thread类、实现Runnable接口，在程序开发中只要是多线程，肯定永远以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类有如下优势：

    1、可以避免由于Java的单继承特性而带来的局限；

    2、增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；

适合多个相同程序代码的线程区处理同一资源的情况。

3、Thread类中的start()和run()方法有什么区别？

 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而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。然后通过此Thread类调用方法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，而CPU再运行其它线程。

  run（）方法当作普通方法的方式调用，程序还是要顺序执行，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： 而如果直接用run方法，这只是调用一个方法而已，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，这样就没有达到多线程的目的。

4、Java中Runnable和Callable有什么不同

相同点：

1. 两者都是接口；（废话）

2. 两者都可用来编写多线程程序；

3. 两者都需要调用Thread.start()启动线程；

不同点：

1. 两者最大的不同点是：实现Callable接口的任务线程能返回执行结果；而实现Runnable接口的任务线程不能返回结果；

2. Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛；

注意点：

• Callable接口支持返回执行结果，此时需要调用FutureTask.get()方法实现，此方法会阻塞主线程直到获取‘将来’结果；当不调用此方法时，主线程不会阻塞！

5、如何避免死锁？

1. 加锁顺序

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁(并对这些锁做适当的排序)，但总有些时候是无法预知的。

2. 加锁时限

另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。

3.死锁检测

死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。例如，线程A请求锁7，但是锁7这个时候被线程B持有，这时线程A就可以检查一下线程B是否已经请求了线程A当前所持有的锁。如果线程B确实有这样的请求，那么就是发生了死锁（线程A拥有锁1，请求锁7；线程B拥有锁7，请求锁1）。

当然，死锁一般要比两个线程互相持有对方的锁这种情况要复杂的多。线程A等待线程B，线程B等待线程C，线程C等待线程D，线程D又在等待线程A。线程A为了检测死锁，它需要递进地检测所有被B请求的锁。从线程B所请求的锁开始，线程A找到了线程C，然后又找到了线程D，发现线程D请求的锁被线程A自己持有着。这是它就知道发生了死锁。

6、Java多线程中调用wait() 和 sleep()方法有什么不同？

**共同点：**

1. 他们都是在多线程的环境下，都可以在程序的调用处阻塞指定的毫秒数，并返回。

2. wait()和sleep()都可以通过interrupt()方法 **打断线程的暂停状态** ，从而使线程立刻抛出InterruptedException。

   如果线程A希望立即结束线程B，则可以对线程B对应的Thread实例调用interrupt方法。如果此刻线程B正在wait/sleep /join，则线程B会立刻抛出InterruptedException，在catch() {} 中直接return即可安全地结束线程。

   需要注意的是，InterruptedException是线程自己从内部抛出的，并不是interrupt()方法抛出的。**对某一线程调用 interrupt()时，如果该线程正在执行普通的代码，那么该线程根本就不会抛出InterruptedException。但是，一旦该线程进入到 wait()/sleep()/join()后，就会立刻抛出InterruptedException** 。

**不同点：**

1. Thread类的方法：sleep(),yield()等

   Object的方法：wait()和notify()等

2. **每个对象都有一个锁来控制同步访问**。Synchronized关键字可以和对象的锁交互，来实现线程的同步。

   sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3.**wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用**

4. sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

**7、什么是Executor框架**

我们知道线程池就是线程的集合，线程池集中管理线程，以实现线程的重用，降低资源消耗，提高响应速度等。线程用于执行异步任务，单个的线程既是工作单元也是执行机制，从JDK1.5开始，为了把工作单元与执行机制分离开，Executor框架诞生了，他是一个用于统一创建与运行的接口。Executor框架实现的就是线程池的功能。

**8、在Java中Executor和Executors的区别**

Executor是**http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "https://blog.csdn.net/qq991029781/article/details/\_blankJava**线程池的顶级**接口；**

Executors是一个**类，** Executors类提供了若干个静态方法，用于生成不同类型的线程池：

**9、什么是多线程中的上下文切换？**

即使是单核CPU也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）。

CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态，**从任务保存到再加载的过程就是一次上下文切换**。

这就像我们同时读两本书，当我们在读一本英文的技术书籍时，发现某

**10、什么是线程安全**

线程安全的代码是多个线程同时执行也能工作的代码

如果一段代码可以保证多个线程访问的时候正确操作共享数据，那么它是线程安全的

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，

就是线程安全的。

或者说:一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。

**11、如何检测死锁？怎么预防死锁？**

利用Java自带工具 https://www.baidu.com/s?wd=JConsole&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "https://blog.csdn.net/wolfcode\_cn/article/details/\_blankJConsole.界面化查看死锁。

**利用jstack 命令检测死锁 jpg jstack -l pid >wenjian.txt**

1. 破坏“不可剥夺”条件：一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到 系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动，执行。

2. 破坏”请求与保持条件“：第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源。第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源。

3. 破坏“循环等待”条件：采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程。

**12、Java中用到的线程调度算法是什么**

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行

(时间片轮转法、优先级调度法、多级反馈队列调度法等

**13、Java中如何获取到线程dump文件**

jmap -dump:format=b,file=F:/heamdump.out 16540”命令即可以生成

**jpg jstack -l pid >wenjian.txt**

**14、池技术有什么作用，常见的池技术有哪些**

**起到对象复用**

**15、用线程池有什么好处，请谈谈线程池的使用场景**

1、避免重复创建线程，减少在创建和 销毁线程时所花时间，及系统的整体开销

2、避免系统创建大量线程而消耗系统资源

3、用户提交的数据能够及时得到处理，响应速度快

4、能够更好的监控和管理线程

常量池 线程池 数据库连接池

**16、线程池的技术原理是什么**

预先启动一些线程，线程无限循环从任务队列中获取一个任务进行执行，直到线程池被关闭。如果某个线程因为执行某个任务发生异常而终止，那么重新创建一个新的线程而已。如此反复。

**17、线程池有哪些种类，各自的使用场景是什么？**

newCachedThreadPool：

• 底层：返回ThreadPoolExecutor实例，corePoolSize为0；maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为60L；unit为TimeUnit.SECONDS；workQueue为SynchronousQueue(同步队列)

• 通俗：当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。

• 适用：执行很多短期异步的小程序或者负载较轻的服务器

newFixedThreadPool：

• 底层：返回ThreadPoolExecutor实例，接收参数为所设定线程数量nThread，corePoolSize为nThread，maximumPoolSize为nThread；keepAliveTime为0L(不限时)；unit为：TimeUnit.MILLISECONDS；WorkQueue为：new LinkedBlockingQueue<Runnable>() 无解阻塞队列

• 通俗：创建可容纳固定数量线程的池子，每隔线程的存活时间是无限的，当池子满了就不在添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)

• 适用：执行长期的任务，性能好很多

newSingleThreadExecutor:

• 底层：FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，corePoolSize为1；maximumPoolSize为1；keepAliveTime为0L；unit为：TimeUnit.MILLISECONDS；workQueue为：new LinkedBlockingQueue<Runnable>() 无解阻塞队列

• 通俗：创建只有一个线程的线程池，且线程的存活时间是无限的；当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)

• 适用：一个任务一个任务执行的场景

NewScheduledThreadPool:

• 底层：创建ScheduledThreadPoolExecutor实例，corePoolSize为传递来的参数，maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为0；unit为：TimeUnit.NANOSECONDS；workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列

• 通俗：创建一个固定大小的线程池，线程池内线程存活时间无限制，线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构

• 适用：周期性执行任务的场景

**18、线程池有哪些重要的参数？**

a.核心线程数

b 最大线程数

c 线程空闲时间

c 时间单位

d 阻塞队列大小：queueCapacity

e 任务拒绝处理器 ：rejectedExceptionHandler

**19、你们在具体的设计开发过程中是如何设置这些重要参数的？**

**根据任务的特性具体方案 具体定制 参见17**

**20、单例的使用场景是什么，如何实现单例**

系统中只存在一个实力，一种是枚举，还有一种私有静态内部类

单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为

**21、如何在Java中创建线程安全的Singleton**

public class StaticSingleton {

02 private StaticSingleton(){

03 System.out.println("StaticSingleton is create");

04 }

05 private static class SingletonHolder {

06 private static StaticSingleton instance = new StaticSingleton();

07 }

08 public static StaticSingleton getInstance() {

09 return SingletonHolder.instance;

10 }

**22、11 }**

**23、synchronzied关键词的使用**

synchronized包裹代码块：

I . synchronized(对象){}

II . synchronized(类名.class){}

III. synchronized(this){}

synchronized修饰方法：

I .public synchronized void memberMethod(){};

II.public static synchronized void staticMethod(){};

**24、ReentrantLock和synchronized使用的场景是什么，机制有何不同**

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化

synchronize，另外可读性非常好，不管用没用过5.0多线程包的程序员都能理解。

ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似轮询锁、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳

的性能。

其实ReentrantLock是一个可重入的互斥锁，重入锁是一种递归无阻塞的同步机制。ReentrantLock由最近成功获取锁，还没有释放的线程所拥有，当锁被另一个线程拥有时，调用lock的线程可以成功获取锁。如果锁已经被当前线程拥有，当前线程会立即返回

ReentrantLock可以等同于synchronized使用，但是比synchronized有更强的功能、可以提供更灵活的锁机制、同时减少死锁的发生概率。在确实需要一些 synchronized 所没有的特性的时候，比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者轮询锁。 ReentrantLock 还具有可伸缩性的好处，应当在高度争用的情况下使用它，但是请记住，大多数 synchronized 块几乎从来没有出现过争用，所以可以把高度争用放在一边。我建议用 synchronized 开发，直到确实证明 synchronized 不合适，而不要仅仅是假设如果使用 ReentrantLock “性能会更好”。请记住，这些是供高级用户使用的高级工具。（而且，真正的高级用户喜欢选择能够找到的最简单工具，直到他们认为

**25、什么是ThreadLocal变量**

ThreadLocal，Thread：线程，这个毫无疑问。那Local呢？本地的，局部的。也就是说，ThreadLocal是线程本地的变量，只要是本线程内都可以使用，线程结束了，那么相应的线程本地变量也就跟随着线程消失了。

**26、ThreadLocal技术原理是什么，它在架构中常常用来做什么？**

**27、java多线程有哪些常见的锁，各自用法是什么？**

synchronized (同步)synchronized关键字修饰的代码相当于数据库上的互斥锁。确保多个线程在同一时刻只能由一个线程处于方法或同步块中，确保线程对变量访问的可见和排它，获得锁的对象在代码结束后，会对锁进行释放。

       synchronzied使用方法有两个：①加在方法上面锁定方法，②定义synchronized块。

condition (配合lock使用 类似 object.wait)Condition.await()方法相当于Object.wait()方法,而Condition.signal()方法相当于Object.notify()方法。当然它也有对应的Condition.signalAll()方法。同样的在调用Condition.await()之后,线程占用的锁会被释放。这样在Condition.signal()方法调用的时候才获取到锁。

需要注意的是Condition.signal()方法调用之后,被唤醒的线程因为需要重新获取锁。所以需要等到调用Condition.signal()的线程释放了锁(调用ReentrantLock.unlock())之后才能继续执行。

**lockInterruptibly()方法**比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。由于lockInterruptibly()的声明中抛出了异常，所以lock.lockInterruptibly()必须放在try块中或者在调用lockInterruptibly()的方法外声明抛出InterruptedException。

ReentrantLock，意思是“可重入锁”，ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法。

分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，这是由jvm自己控制的，我们只要上好相应的锁即可。如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁；如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁！读写锁接口：**ReadWriteLock**，它的具体实现类为：**ReentrantReadWriteLock**。

**28、CountDownLatch用于多线程的什么场景**

1. ountDownLatch类。这个类是一个同步辅助类。用于一个线程等待多个操作完成之后再执行，也就是这个当前线程会一直阻塞，直到它所等待的多个操作已经完成。await方法，需要等到其他操作先完成的那个线程调用的，先将线程休眠，直到其他操作完成，计数器减为0，才会唤醒因此休眠的线程

2. countDown方法，每个被等待的事件在完成之后调用，会将计数器减一

**29、volatile适用于高并发的什么场景**

volatile最适用一个线程写，多个线程读的场合。

   如果有多个线程并发写操作，仍然需要使用锁或者线程安全的容器或者原子变量来代替。(摘自Netty权威指南)

您只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

• 对变量的写操作不依赖于当前值。

• 该变量没有包含在具有其他变量的不变式中。

**30、多线程join方法用于什么场景？**

，主线程创建并启动了子线程，如果子线程中需要进行大量的耗时运算，主线程往往将早于子线程结束之前结束，如果主线程想等待子线程执行完毕后，获得子线程中的处理完的某个数据，就要用到join方法了，方法join（）的作用是等待线程对象呗销毁；

**join底层是wait方法，所以它是会释放https://www.baidu.com/s?wd=%E5%AF%B9%E8%B1%A1%E9%94%81&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "https://blog.csdn.net/liuyifeng1920/article/details/\_blank对象锁的，而sleep在同步的方法中是不释放对象锁的，只有同步方法执行完毕，其他线程才可以执行。**

**31、java多线程中让所有子线程执行完毕的方法有哪几种？**

1、用sleep方法，让主线程睡眠一段时间，当然这个睡眠时间是主观的时间，是我们自己定的，这个方法不推荐，但是在这里还是写一下，毕竟是解决方法

2、使用Thread的join()等待所有的子线程执行完毕，主线程在执行，thread.join()把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程。比如在线程B中调用了线程A的Join()方法，直到线程A执行完毕后，才会继续执行线程B。

3、countDownLatch不可能重新初始化或者修改CountDownLatch对象内部计数器的值，一个线程调用countdown方法happen-before另外一个线程调用await方法

4、同步屏障CyclicBarrier方法可以使用reset()方法重置，所以CyclicBarrier方法可以能处理更为复杂的业务场景。

**32、高并发环境下的计数器如何实现？**

**33、HashTable、HashMap、ConcurrentHashMap各自的技术原理和使用场景是什么？**

HashMap

实现了Map接口，实现了将唯一键隐射到特定值上。允许一个NULL键和多个NULL值。非线程安全。

HashTable

类似于HashMap，但是不允许NULL键和NULL值，比HashMap慢，因为它是同步的。HashTable是一个线程安全的类，它使用synchronized来锁住整张Hash表来实现线程安全，即每次锁住整张表让线程独占。

ConcurrentHashMap

ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对hash表的不同部分进行的修改。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的Hashtable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

**34、LinkedBlockingQueue、ConcurrentLinkedQueue各自技术原理和使用场景是什么?**

 LinkedBlockingQueue是一个线程安全的阻塞队列，基于链表实现，一般用于生产者与消费者模型的开发中。采用锁机制来实现多线程同步，提供了一个构造方法用来指定队列的大小，如果不指定大小，队列采用默认大小（Integer.MAX\_VALUE，即整型最大值）。

ConcurrentLinkedQueue是一个线程安全的非阻塞队列，基于链表实现。java并没有提供构造方法来指定队列的大小，因此它是无界的。为了提高并发量，它通过使用更细的锁机制，使得在多线程环境中只对部分数据进行锁定，从而提高运行效率

**35、Java中如何停止一个线程?**

  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方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果）。

1.     3.  使用interrupt方法中断线程。

**36、Java中Semaphore是什么?**

Semaphore是用来保护一个或者多个共享资源的访问，Semaphore内部维护了一个计数器，其值为可以访问的共享资源的个数。一个线程要访问共享资源，先获得信号量，如果信号量的计数器值大于1，意味着有共享资源可以访问，则使其计数器值减去1，再访问共享资源。

如果计数器值为0,线程进入休眠。当某个线程使用完共享资源后，释放信号量，并将信号量内部的计数器加1，之前进入休眠的线程将被唤醒并再次试图获得信号量。

Semaphore除了控制资源的多个副本的并发访问控制，也可以使用二进制信号量来实现类似synchronized关键字和Lock锁的并发访问控制功能。

**37、java多线程中有哪些并发流量控制工具类？**

别为CountDownLatch、CyclicBarrier、Semaphore和Exchanger，;

1、CountDownLatch，它是一种计数器的方式保证线程同步；它不去控制多个子线程之间的前后关系，只保证某一线程能够在这些子线程执行完成后再执行。

2、CyclicBarrier，通过设置屏障的方式使得多线程同步，能够控制多个线程在屏障处等等其他线程也执行到屏障点，可以实现CountDownLatch具有的功能，但是比CountDownLatch功能强大；

3、Semaphore，信号量，用于控制访问某一公共资源的并发线程数；

4、Exchanger，用于两个线程之间的数据交换。

**38、如何理解动态代理?**

   代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。简单的说就是，我们在访问实际对象时，是通过代理对象来访问的，代理模式就是在访问实际对象时引入一定程度的间接性，因为这种间接性，可以附加多种用途;

**39、什么是线程安全？**

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是https://www.baidu.com/s?wd=%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8&tn=SE\_PcZhidaonwhc\_ngpagmjz&rsv\_dl=gh\_pc\_zhidao" \t "https://zhidao.baidu.com/question/\_blank线程安全的。

**40、能举几个不是线程安全的数据结构么？**

Hashmap 、ArrayList 、LinkedList 、TreeMap、

**41、常见的多线程数据结构有哪些，你用过其中的哪些多线程数据结构？**

Vector、CopyOnWriteArrayList、CopyOnWriteArraySet、、HashTable(子类LinkedHashMap)ConcurrentHashMap

ConcurrentLinkedQueue采用的是无锁的方式，所以其性能在高并发中很好。

　　　　BlockingQueue采用的是生产者消费者模式的方式进行加锁。Blocking的具体实现有ArrayBlockingQueue和LinkedBlockingQueue

**42、多线程的常见设计模式，你用过其中的哪些设计模式**

Future模式，Master-Worker模式，生产者-消费者模型

**43、什么是Master-Worker模式？如何实现Master-Worker模式？**

Master-Worker模式是常用的并行计算模式。他的核心思想是系统由两类进程协作工作：Master进程和Worker进程.Maseter负责接收和分配任务, Worker负责处理子任务。当各个Worker子进行处理完成后，会将结果返回给Master，由Msster做归纳总结，好处是能将一个大任务分解成若干个小任务，并行执行，从而提高系统的吞吐量

**44、什么是Producer-Consumer模式？如何实现Producer-Consumer模式；**

在生产-消费模式中:通常由两类线程，即若干个生产者和若干个消费者的线程。生产者负责提交用户数据，消费者负责具体处理生产者提交的任务，在生产者和消费者之间通过共享内存缓存区进行通信。

**45、什么是Future模式？如何实现Future模式**

Future模式类似于异步请求

**46、多线程使用场景是什么？**

1、数据库的数据分析(待分析的数据太多)，数据迁移。

2、servlet多线程。

3、FTP下载，多线程操作文件。

4、数据库用到的多线程。

5、分布式计算。

6、tomcat，tomcat内部采用多线程，上百个客户端访问同一个WEB应用，tomcat接入后就是把后续的处理扔给一个新的线程来处理，这个新的线程最后调用我们的servlet程序，比如doGet或者dpPost方法。

7、后台任务：如定时向大量(100W以上)的用户发送邮件；定期更新配置文件、任务调度(如quartz)，一些监控用于定期信息采集。

8、自动作业处理：比如定期备份日志、定期备份数据库。

9、异步处理：如发微博、记录日志。

10、页面异步处理：比如大批量数据的核对工作(有10万个手机号码，核对哪些是已有用户)。

**47、多线程有优缺点？**

何时使用多线程技术,何时避免用它,是我们需要掌握的重要课题。多线程技术是一把双刃剑,在使用时需要充分考虑它的优缺点。

多线程处理可以同时运行多个线程。由于多线程应用程序将程序划分成多个独立的任务,因此可以在以下方面显著提高性能:

(1)多线程技术使程序的响应速度更快 ,因为用户界面可以在进行其它工作的同时一直处于活动状态;

(2)当前没有进行处理的任务时可以将处理器时间让给其它任务;

(3)占用大量处理时间的任务可以定期将处理器时间让给其它任务;

(4)可以随时停止任务;

(5)可以分别设置各个任务的优先级以优化性能。

 是否需要创建多个线程取决于各种因素。在以下情况下,最适合采用多线程处理:

(1)耗时或大量占用处理器的任务阻塞用户界面操作;

(2)各个任务必须等待外部资源 (如远程文件或 Internet连接)。

同样的 ,多线程也存在许多缺点 ,在考虑多线程时需要进行充分的考虑。多线程的主要缺点包括:

(1)等候使用共享资源时造成程序的运行速度变慢。这些共享资源主要是独占性的资源 ,如打印机等。

(2)对线程进行管理要求额外的 CPU开销。线程的使用会给系统带来上下文切换的额外负担。当这种负担超过一定程度时,多线程的特点主要表现在其缺点上,比如用独立的线程来更新数组内每个元素。

(3)线程的死锁。即较长时间的等待或资源竞争以及死锁等多线程症状。

(4)对公有变量的同时读或写。当多个线程需要对公有变量进行写操作时,后一个线程往往会修改掉前一个线程存放的数据,从而使前一个线程的参数被修改;另外 ,当公用变量的读写操作是非原子性时,在不同的机器上,中断时间的不确定性,会导致数据在一个线程内的操作产生错误,从而产生莫名其妙的错误,而这种错误是程序员无法预知的。

**48、假设某系统的某个接口的峰值TPS为2w/s(其它接口的并发峰值至多为200每秒)，且该接口会保存数据至数据库，如何提升该接口的性能？**

利用多线程 将并发数改成200；

创建一个任务队列里面存入要存放任务书2.1W;创建线程2个100或1个200处理请求。如果实时，处理不来可以设置超时返回错误。

**49、是否熟悉java concurrent包的内容，请讲讲concurrent包有哪些重要的内容？**

locks部分：显式锁(互斥锁和速写锁)相关；

atomic部分：原子变量类相关，是构建非阻塞算法的基础；

executor部分：线程池相关；

collections部分：并发容器相关；

tools部分：同步工具相关，如信号量、闭锁、栅栏等功能；

**50、请讲讲并发编程的CAS理论**

CAS 操作包含三个操作数 -- 内存位置、预期数值和新值。CAS 的实现逻辑是将内存位置处的数值与预期数值想比较，若相等，则将内存位置处的值替换为新值。若不相等，则不做任何操作。

**51、请讲讲并发队列和阻塞队列**

ConcurrentLinkedQueue : 是一个适用于高并发场景下的队列，通过无锁的方式，实现

了高并发状态下的高性能，通常ConcurrentLinkedQueue性能好于BlockingQueue.它

是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先

加入的，尾是最近加入的，该队列不允许null元素。

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

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。

BlockingQueue即阻塞队列，从阻塞这个词可以看出，在某些情况下对阻塞队列的访问可能会造成阻塞。被阻塞的情况主要有如下两种：1. 当队列满了的时候进行入队列操作2. 当队列空了的时候进行出队列操作

因此，当一个线程试图对一个已经满了的队列进行入队列操作时，它将会被阻塞，除非有另一个线程做了出队列操作；同样，当一个线程试图对一个空队列进行出队列操作时，

它将会被阻塞，除非有另一个线程进行了入队列操作。

在Java中，BlockingQueue的接口位于java.util.concurrent 包中(在Java5版本开始提供)，由上面介绍的阻塞队列的特性可知，阻塞队列是线程安全的。

在新增的Concurrent包中，BlockingQueue很好的解决了多线程中，如何高效安全“传输”数据的问题。通过这些高效并且线程安全的队列类。

**52、多线程yield方法使用于什么场景？**

Thread.yield()方法作用是：暂停当前正在执行的线程对象（及放弃当前拥有的cup资源），并执行其他线程。

yield()做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的

目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被

线程调度程序再次选中。

**53、请讲讲线程异步处理的原理及关键组件？**

在http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "https://www.cnblogs.com/tuojunjie/p/\_blankJava平台,实现异步调用的角色有如下三个角色：调用者、 提货单 、真实数据，一个调用者在调用耗时操作,不能立即返回数据时,先返回一个提货单

.然后在过一断时间后凭提货单来获取真正的数据.去蛋糕店买蛋糕，不需要等蛋糕做出来(假设现做要很长时间)，只需要领个提货单就可以了(去干别的

事情)，等到蛋糕做好了，再拿提货单取蛋糕就可以了。

**54、在实际项目（产品）研发过程中，你是否有使用过多线程，和线程池，如果有，请举例说明（要用STAR模型）；**

**55、什么是多线程的原子操作？Java 中有哪些原子操作？**

即不能被线程调度机制中断的操作。原子操作不需要进行同步控制。

原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而只执行其中的一部分，将整个操作视作一个整体是原子性的核心特征；

原子更新基本类型

1）除long和double之外的基本类型的赋值操作

2）所有引用reference的赋值操作

3）java.concurrent.Atomic.\* 包中所有类的一切操作

AtomicBoolean ：原子更新布尔类型

AtomicInteger： 原子更新整型

AtomicLong: 原子更新长整型

原子更新数组

AtomicIntegerArray ：原子更新整型数组里的元素

AtomicLongArray :原子更新长整型数组里的元素

AtomicReferenceArray : 原子更新引用类型数组的元素

AtomicBooleanArray ：原子更新布尔类型数组的元素

原子更新引用类型

AtomicReference ：原子更新引用类型

AtomicReferenceFieldUpdater ：原子更新引用类型里的字段

AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和应用类型

原子更新字段类

AtomicIntegerFieldUpdater:原子更新整型的字段的更新器

AtomicLongFieldUpdater：原子更新长整型字段的更新器

AtomicStampedReference:原子更新带有版本号的引用类型。该类将整型数值与引用关联起来，可用于原子的更新数据和数据的版本号，可以解决使用CAS进行原子更新时可能出现的ABA问题。

**56、多线程的原子操作类的使用场景是什么，你在项目的实际研发过程中是否有使用过原子操作类？**

原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而只执行其中的一部分，将整个操作视作一个整体是原子性的核心特征；

原子更新基本类型

计数器 可以用原子操作

**57、如何在多个线程间共享数据？**

如果每个线程执行的代码相同，可以使用同一个Runnable对象，这个Runnable对象中有那个共享数据，例如，卖票系统就可以这么做。

将共享数据封装成另外一个对象，然后将这个对象逐一传递给各个Runnable对象，每个线程对共享数据的操作方法也分配到那个对象身上完成，这样容易实现针对数据进行各个操作的互斥和通信

将Runnable对象作为一个类的内部类，共享数据作为这个类的成员变量，每个线程对共享数据的操作方法也封装在外部类，以便实现对数据的各个操作的同步和互斥，作为内部类的各个Runnable对象调用外部类的这些方法。

**58、线程的状态有哪些，线程状态的使用场景是什么？**

　1、新状态：线程对象已经创建，还没有在其上调用start()方法。

　　2、可运行状态：当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态。

　　3、运行状态：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。

　　4、等待/阻塞/睡眠状态：这是线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的，但是当前没有条件运行。换句话说，它是可运行的，但是如果某件事件出现，他可能返回到可运行状态。

5、死亡态：当线程的run()方法完成时就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

**59、有多个线程T1，T2，T3，怎么确保它们按顺序执行？**

1）可以在线程里面加入join方法 一次等待前面的线程执行完 在执行后面的

2）用Executors.newSingleThreadExecutor();分别依次提交 开启，这样就执行有序了。

**60、volatile变量和atomic变量有什么不同？**

Volatile是让变量在所有线程中变得可见。操作时不一定保证原子性，线程安全。

Atomic是原子性，线程安全的。他的修改是sysnizied。

**61、wait/notify/notifyAll一般使用于什么场景？**

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。

   当有线程调用了对象的notifyAll（）方法（唤醒所有wait线程）或notify（）方法（只随机唤醒一个wait线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。

   优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用wait（）方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了synchronized代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

**62、什么是JVM ?**

JVM是Java Virtual Machine（Javahttps://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "https://baike.baidu.com/item/JVM/\_blank虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

https://baike.baidu.com/item/Java%E8%AF%AD%E8%A8%80" \t "https://baike.baidu.com/item/JVM/\_blankJava语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的https://baike.baidu.com/item/%E7%9B%AE%E6%A0%87%E4%BB%A3%E7%A0%81/9407934" \t "https://baike.baidu.com/item/JVM/\_blank目标代码。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E7%A8%8B%E5%BA%8F/8290180" \t "https://baike.baidu.com/item/JVM/\_blank编译程序只需生成在Java虚拟机上运行的目标代码（https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81/9953683" \t "https://baike.baidu.com/item/JVM/\_blank字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E6%8C%87%E4%BB%A4/8553126" \t "https://baike.baidu.com/item/JVM/\_blank机器指令执行。这就是Java的能够“一次编译，到处运行”的原因。

**63、Java中堆和栈有什么区别？**

各司其职：

最主要的区别就是栈内存用来存储局部变量和方法调用。

而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

独有还是共享：

栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。

而堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

异常错误：

如果栈内存没有可用的空间存储方法调用和局部变量，JVM会抛出java.lang.StackOverFlowError。

而如果是堆内存没有可用的空间存储生成的对象，JVM会抛出java.lang.OutOfMemoryError。

空间大小：

栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满。如果递归没有及时跳出，很可能发生StackOverFlowError问题。

你可以通过-Xss选项设置栈内存的大小。-Xms选项可以设置堆的开始时的大小，-Xmx选项可以设置堆的最大值。

这就是Java中堆和栈的区别。理解好这个问题的话，可以对你解决开发中的问题，分析堆内存和栈内存使用，甚至性能调优都有帮助。

**64、请说说jvm的基本结构？**

它包括：类加载器子系统、运行时数据区、执行引擎和本地方法接口。

运行时数据区是JVM从操作系统申请来的堆空间和操作系统给JVM分配的栈空间的总称。JVM为了运行Java程序，又进一步对运行时数据区进行了划分，划分为Java方法区、Java堆、Java栈、PC寄存器、本地方法栈等，这里JVM从操作系统申请来的堆空间被划分为方法区和Java堆，操作系统给JVM分配的栈空间构成Java栈。

**65、堆空间的结构**

运行时数据区中Java的方法区和Java堆（图中显示的是：永久、新生、老年，这是分代垃圾回收时的术语，实际上永久代和Java方法区对应，https://www.baidu.com/s?wd=%E6%96%B0%E7%94%9F%E4%BB%A3&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "https://blog.csdn.net/zhangzl4321/article/details/\_blank新生代和老年代和Java堆对应），也就说Java方法区和Java堆其实都是JVM堆的一部分。JVM的栈区构成了Java的线程栈。

**66、为何新生代要设置两个survivor区，jvm的设计上有何目的？**

1、Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。

2、设置两个Survivor区最大的好处就是解决了碎片化，永远有一个survivor space是空的，另一个非空的survivor space无碎片。

设计目的：

应该建立两块Survivor区，刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到第一块survivor space S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制送入第二块survivor space S1（这个过程非常重要，因为这种复制算法保证了S1中来自S0和Eden两部分的存活对象占用连续的内存空间，避免了碎片化的发生）。S0和Eden被清空，然后下一轮S0与S1交换角色，如此循环往复。如果对象的复制次数达到16次，该对象就会被送到老年代中。下图中每部分的意义和上一张图一样，就不加注释了。

**67、垃圾回收中的复制算法适用于在什么场景下使用？**

将内存分为（大小相等）两部分，每次只使用其中一块进行内存分配，当内存使用完后，就出发GC，将存活的对象直接复制到另一块空闲的内存中，然后对当前使用的内存块一次性清除所有，然后转到另一块内存进行使用。

优点：简单，高效。

缺点：浪费内存，因为每次都有另一块内存空闲着。

Eden survivor 垃圾回收

**68、老年代的垃圾回收一般用什么算法？**

标记-压缩-清理算法进行垃圾回收，将标记对象移动到堆的另一端，同时更新对象的引用地址

1、mark\_sweep\_phase1： 标记活跃对象

2、mark\_sweep\_phase2： 计算活跃对象在压缩完成之后的新地址

**69、怎么获取 Java 程序使用的内存？堆使用的百分比？**

jhat：内存分析工具：

主要是对java应用程序的资源和性能进行实时的命令行监控，包括了对heap size和垃圾回收状况的监控。

 jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

option：我们经常使用的选项有gc、gcutil

vmid：java进程id

interval：间隔时间，单位为毫秒

count：打印次数

jmap -heap pid:查看堆使用情况

jmap -histo pid：查看堆中对象数量和大小

jmap -dump:format=b,file=heapdump pid：将内存使用的详细情况输出到文件

序列号、Class实例的数量、内存的占用、类限定名

如果是内部类，类名的开头会加上\*，如果加上live子参数的话，如jmap -histo：live pid，这个命名会触发一次FUll GC，只统计存活对象

**70、GC回收机制？**

**71、jmap命令是有什么用途？jstat命令是有什么用途？**

https://www.baidu.com/s?wd=Jmap&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "\_blankmap是一个可以输出所有内存中对象的工具，甚至可以将VM 中的heap，以二进制输出成文本。打印出某个java进程（使用pid）内存内的，

Jstat是JDK自带的一个轻量级小工具。全称“Java Virtual Machine statistics monitoring tool”，它位于java的bin目录下，主要利用JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，包括了对Heap size和垃圾回收状况的监控。可见，Jstat是轻量级的、专门针对JVM的工具，非常适用。

**72、有哪些常见的jvm命令，说说各自的用途是什么？**

**Jps:JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程**

Jstat: 是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

Jmap:(JVM Memory Map)命令用于生成heap dump文件，如果不使用这个命令，还阔以使用-XX:+HeapDumpOnOutOfMemoryError参数来让虚拟机出现OOM的时候·自动生成dump文件。

jmap不仅能生成dump文件，还阔以查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等。

jhat(JVM Heap Analysis Tool)命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看。在此要注意，一般不会直接在服务器上进行分析，因为jhat是一个耗时并且耗费硬件资源的过程，一般把服务器生成的dump文件复制到本地或其他机器上进行分析。

jstack用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。 如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。另外，jstack工具还可以附属到正在运行的java程序中，看到当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。

info(JVM Configuration info)这个命令作用是实时查看和调整虚拟机运行参数。

之前的jps -v口令只能查看到显示指定的参数，如果想要查看未被显示指定的参数的值就要使用jinfo口令

**73、、GC有哪些算法（\*\*\*\*\*）**

引用计数,没有被Java采用

标记-清除

标记-压缩 标记-整理算法

复制算法 新生代

**74、、什么是线程中断。tips: stop the world，简称STW，参考billy1.GC算法与种类**

Java中一种全局暂停的现象

全局停顿，所有Java代码停止，native代码可以执行，但不能和JVM交互

**75、MGC、FGC分别是什么意思，它们在什么情况下会发生**

，YG用来放新产生的对象，经过几次回收还没回收掉的对象往OG中移动，对YG进行垃圾回收又叫做MinorGC，对 OG垃圾回收又叫MajorGC，.

1、当eden满了，触发young GC；

2.young GC做2件事：一，去掉一部分没用的object；二，把老的还被引用的object发到survior里面，等下几次GC以后，survivor再放到old里面。

3.当old满了，触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

**76、、请讲讲jvm的分代，为什么要分代，jvm分代有什么好处？**

虚拟机中的共划分为三个代：年轻代（Young Generation）、年老点（Old Generation）和持久代（Permanent Generation）。其中持久代主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

利用对象存活的生命不同。利用的算法不同。

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个Eden区，两个Survivor区(一般而言)。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当这个Survivor去也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来 对象，和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor去过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

**77、、你知道哪些jvm调优工具么？**

uptime 系统时间 运行时间 连接数 1,5,15分钟内的系统平均负载

top

vmstat可以统计系统的CPU，内存，swap，io等情况

pidstat细致观察进程

任务管理器

Perfmon

Process Explorer

pslist

**78、、在jvm中，年轻代如何向老年代转变的？年轻代向老年代转换的重要参数是什么?**

当eden满了，触发young GC；

2.young GC做2件事：一，去掉一部分没用的object；二，把老的还被引用的object发到survior里面，等下几次GC以后，survivor再放到old里面。

3.当old满了，触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

-Xms 和 -Xmx (-XX:InitialHeapSize 和 -XX:MaxHeapSize)：指定JVM初始占用的堆内存和最大堆内存。JVM也是一个软件，也必须要获取本机的物理内

-XX:MaxTenuringThreshold= 设置熬过年轻代多少次收集后移入老人区，CMS中默认为0，熬过第一次GC就转入，可以用-XX:+PrintTenuringDistribution 查看

调用16次

**79、、直接内存使用场景是什么，使用直接内存可能会存在什么问题？tips**

80、、堆内存有哪些重要参数？

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制，可以由 -XX:MinHeapFreeRatio=指定。

默认空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制，可以由 -XX:MaxHeapFreeRatio=指定。

服务器一般设置-Xms、-Xmx相等以避免在每次GC后调整堆的大小，所以上面的两个参数没啥用。

**oung(Nursery)：年轻代**

研究表明大部分对象都是朝生暮死，随生随灭的。所以对于年轻代在GC时都采取复制收集算法，具体算法参考下面的描述；

Young的默认值为4M，随堆内存增大，约为1/15，JVM会根据情况动态管理其大小变化。

Young里面又分为3 个区域，一个Eden，所有新建对象都会存在于该区，两个Survivor区，用来实施复制算法。

-XX:NewRatio= 参数可以设置Young与Old的大小比例，-server时默认为1:2，但实际上young启动时远低于这个比率？如果信不过JVM，也可以用 -Xmn硬性规定其大小，有文档推荐设为Heap总大小的1/4。

-XX:SurvivorRatio= 参数可以设置Eden与Survivor的比例，默认为32。Survivio大了会浪费，小了的话，会使一些年轻对象潜逃到老人区，引起老人区的不安，但这个参数对性能并不太重要。

**Old(Tenured)：年老代**

年轻代的对象如果能够挺过数次收集，就会进入老人区。老人区使用标记整理算法。因为老人区的对象都没那么容易死的，采用复制算法就要反复的复制对象，很不合算，只好采用标记清理算法，但标记清理算法其实也不轻松，每次都要遍历区域内所有对象，所以还是没有免费的午餐啊。

-XX:MaxTenuringThreshold= 设置熬过年轻代多少次收集后移入老人区，CMS中默认为0，熬过第一次GC就转入，可以用-XX:+PrintTenuringDistribution 查看。

**Permanent：持久代**

装载Class信息等基础数据，默认64M，如果是类很多很多的服务程序，需要加大其设置 -XX:MaxPermSize=，否则它满了之后会引起fullgc()或Out of Memory。 注意Spring，Hibernate这类喜欢AOP动态生成类的框架需要更多的持久代内存。一般情况下，持久代是不会进行GC的，除非通过 -XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled进行强制设置。

81、如何设置堆大小，是否有一些经验值？

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统 下，一般限制在1.5G~2G；64为操作系统对内存无限制。我在Windows Server 2003 系统，3.5G物理内存，JDK5.0下测试，最大可设置为1478m。

**典型JVM参数设置：**

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-Xmx3550m：设置JVM最大可用内存为3550M。

-Xms3550m：设置JVM促使内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn2g：设置年轻代大小为2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-Xss128k：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具应用的线程所需内存大小进行 调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5

-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个 Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

-XX:MaxPermSize=16m:设置持久代大小为16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。 对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活 时间，增加在年轻代即被回收的概论。

82、如何打印JVM日志？

-XX:+PrintGCDetails -Xloggc:../logs/gc.log -XX:+PrintGCTimeStamps

83、请介绍常见的jvm参数

-XX:+PrintGCTimeStamps:

打印此次垃圾回收距离jvm开始运行的所耗时间

-XX:+PrintGCDeatils

打印垃圾回收的细节信息

-Xloggc:<filename>

将垃圾回收信息输出到指定文件

-XX:+PrintGCDateStamps

需要打印日历形式的时间戳选项

-XX:+PrintGCApplicationStoppedTime

-XX:+PrintGCApplicationConcurrentTime

打印应用程序由于执行VM安全点操作而阻塞的时间以及两个安全点操作之间应用程序的运行时间

-XX:+PrintSafepointStatistics

可以将垃圾回收的安全点与其他的安全点区分开

ν -XX:+UseSerialGC：在新生代和老年代使用串行收集器

ν -XX:SurvivorRatio：设置eden区大小和survivior区大小的比例

ν -XX:NewRatio:新生代和老年代的比

ν -XX:+UseParNewGC：在新生代使用并行收集器

ν -XX:+UseParallelGC ：新生代使用并行回收收集器

ν -XX:+UseParallelOldGC：老年代使用并行回收收集器

ν -XX:ParallelGCThreads：设置用于垃圾回收的线程数

ν -XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用CMS+串行收集器

ν -XX:ParallelCMSThreads：设定CMS的线程数量

ν -XX:CMSInitiatingOccupancyFraction：设置CMS收集器在老年代空间被使用多少后触发

ν -XX:+UseCMSCompactAtFullCollection：设置CMS收集器在完成垃圾收集后是否要进行一次内存碎片的整理

ν -XX:CMSFullGCsBeforeCompaction：设定进行多少次CMS垃圾回收后，进行一次内存压缩

ν -XX:+CMSClassUnloadingEnabled：允许对类元数据进行回收

ν -XX:CMSInitiatingPermOccupancyFraction：当永久区占用率达到这一百分比时，启动CMS回收

ν -XX:UseCMSInitiatingOccupancyOnly：表示只在到达阀值的时候，才进行CMS回收

84、CMS收集器有什么特点？

尽可能降低停顿

会影响系统整体吞吐量和性能

比如，在用户线程运行过程中，分一半CPU去做GC，系统性能在GC阶段，反应速度就下降一半

清理不彻底

因为在清理阶段，用户线程还在运行，会产生新的垃圾，无法清理

因为和用户线程一起运行，不能在空间快满时再清理

85、G1收集器有什么特点？

并行于并发：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。

2、分代收集：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。它能够采用不同的方式去处理新创建的对象和已经存活了一段时间，熬过多次GC的旧对象以获取更好的收集效果。

3、空间整合：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。

4、可预测的停顿：这是G1相对于CMS的另一个大优势，降低停顿时间是G1和ＣＭＳ共同的关注点，但Ｇ１除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确

86、垃圾回收器有哪些？

串行垃圾回收器（Serial Garbage Collector）

并行垃圾回收器（Parallel Garbage Collector）

并发标记扫描垃圾回收器（CMS Garbage Collector）

G1垃圾回收器（G1 Garbage Collector）

87、java内存模型

1、主内存和工作内存

    （1）所有变量均存储在主内存（虚拟机内存的一部分）

    （2）每个线程都对应着一个工作线程，主内存中的变量都会复制一份到每个线程的自己的工作空间，线程对变量的操作都在自己的工作内存中，操作完成后再将变量更新至主内存；

    （3）其他线程再通过主内存来获取更新后的变量信息，即线程之间的交流通过主内存来传递

Note：JMM的空间划分和JVM的内存划分不一样，非要对应的话，关系如下：

    （1）JMM的主内存对应JVM中的堆内存对象实例数据部分

    （2）JMM的工作内存对应JVM中栈中部分区域

88、什么是类加载器，类加载器有哪些，类加载器的加载顺序是什么？

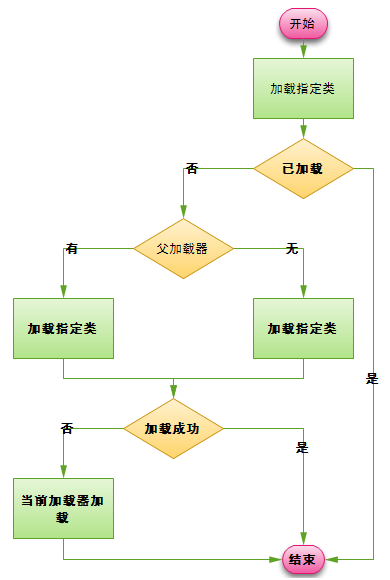
类加载器是一个用来加载类文件的类。Java源代码通过javac编译器编译成类文件。然后JVM来执行类文件中的字节码来执行程序。类加载器负责加载文件系统、网络或其他来源的类文件。有三种默认使用的类加载器：Bootstrap类加载器、Extension类加载器和System类加载器（或者叫作Application类加载器）。

1）Bootstrap类加载器 – JRE/lib/rt.jar

2) Extension类加载器 – JRE/lib/ext或者java.ext.dirs指向的目录

3) Application类加载器 – CLASSPATH环境变量, 由-classpath或-cp选项定义,或者是JAR中的Manifest的classpath属性定义.

VM并不是把所有的类一次性全部加载到JVM中的，也不是每次用到一个类的时候都去查找，对于JVM级别的类加载器在启动时就会把默认的JAVA\_HOME/lib里的class文件加载到JVM中，因为这些是系统常用的类，对于其他的第三方类，则采用用到时就去找，找到了就缓存起来的，下次再用到这个类的时候就可以直接用缓存起来的类对象了，ClassLoader之间也是有父子关系的，没个ClassLoader都有一个父ClassLoader,在加载类时ClassLoader与其父ClassLoader的查找



89、简述java内存分配与回收策略

1、 当eden满了，触发young GC；

2.young GC做2件事：一，去掉一部分没用的object；二，把老的还被引用的object发到survior里面，等下几次GC以后，survivor再放到old里面。

3.当old满了，触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

•Tips：eden、Sruvivor、老年代、永久代（元空间）

90、JDK1.8之后Perm Space有哪些变动? MetaSpace大小默认是无限的么? 还是你们会通过什么方式来指定大小?

1、 JDK 1.8后用元空间替代了 Perm Space；字符串常量存放到堆内存中。

2、 MetaSpace大小默认没有限制，一般根据系统内存的大小。JVM会动态改变此值。

3、 -XX:MetaspaceSize：分配给类元数据空间（以字节计）的初始大小（Oracle逻辑存储上的初始高水位，the initial high-water-mark）。此值为估计值，MetaspaceSize的值设置的过大会延长垃圾回收时间。垃圾回收过后，引起下一次垃圾回收的类元数据空间的大小可能会变大。

4、 -XX:MaxMetaspaceSize：分配给类元数据空间的最大值，超过此值就会触发Full GC，此值默认没有限制，但应取决于系统内存的大小。JVM会动态地改变此值。

91、Perm Space中保存什么数据？会引起OutOfMemory吗？

加载class文件。

会引起，出现异常可以设置 -XX:PermSize 的大小。JDK 1.8后，字符串常量不存放在永久带，而是在堆内存中，JDK8以后没有永久代概念，而是用元空间替代，元空间不存在虚拟机中，二是使用本地内存。

详细查看Java8内存模型—永久代(PermGen)和元空间(Metaspace)

92、java类加载全过程，从架构角度理解，类加载和反射、动态代理有什么关系？

93、简述java类加载机制？tips：看ClassLoader源码讲解类加载机制，理解记忆

**加载----验证----准备----解析-----初始化----使用-----卸载**

94、GC收集器有哪些？CMS收集器与G1收集器的特点

串行垃圾回收器（Serial Garbage Collector）

并行垃圾回收器（Parallel Garbage Collector）

并发标记扫描垃圾回收器（CMS Garbage Collector）

G1垃圾回收器（G1 Garbage Collector）

95、类加载器双亲委派模型机制，“双亲委派”中的双亲是什么意思？tips：演示ClassLoaderTest2，讲解双亲委派流程图

如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。

96、什么情况下会出现永久代内存溢出，如何解决此类问题？

生成大量的类，增大Perm区 允许Class回收

97、什么情况下会出现堆内存溢出，如何解决此类问题？

占用大量堆空间，直接溢出

增大堆空间，及时释放内存

98、什么情况下会出现直接内存溢出，如何解决此类问题？

ByteBuffer.allocateDirect()无法从操作系统获得足够的空间

解决方法：**减少**堆内存 有意触发GC

99、什么情况下会出现过多线程导致内存溢出的问题，如何解决此类问题？

多线程 没有释放内存

– 1、OOM由于保存多线程过多引起，可以考虑增加堆大小

– 2. 如果应用允许，缩短多线程的过期时间，使得session可以及时过期，并回收

100、什么情况下会出现CPU使用率过高的问题，如何解决此类问题？

多线程竞争资源，多线程上下文切换太频繁

合理设置线程最大开启数量，并发数量

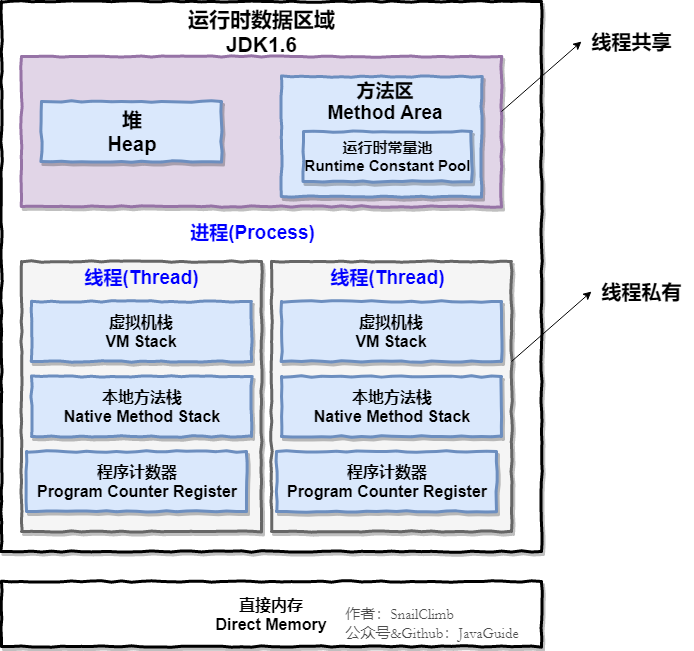
个人整理的各大博主写的资料供自己日后复习

**1、什么是线程和进程?**

**进程**：在操作系统中能够独立运行，并且作为资源分配的基本单位。它表示运行中的程序。系统运行一个程序就是一个进程从创建、运行到消亡的过程。

**线程**：是一个比进程更小的执行单位，能够完成进程中的一个功能，也被称为轻量级进程。一个进程在其执行的过程中可以产生多个线程。

【注】线程与进程不同的是：同类的多个线程**共享进程的堆和方法区资源**，但**每个线程有自己的程序计数器、虚拟机栈和本地方法栈**，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多。



**为什么程序计数器、虚拟机栈和本地方法栈是线程私有的呢？为什么堆和方法区是线程共享的呢？**

* **程序计数器为什么是私有的?**

程序计数器主要有下面两个作用：

* 1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
  2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

（需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。）

所以，程序计数器私有主要是为了**线程切换后能恢复到正确的执行位置**。

* **虚拟机栈和本地方法栈为什么是私有的?**
* **虚拟机栈**： **每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程**。
* **本地方法栈**： 和虚拟机栈所发挥的作用非常相似，区别是： 虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。 在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

所以，为了**保证线程中的局部变量不被别的线程访问到**，虚拟机栈和本地方法栈是线程私有的。

**堆和方法区是所有线程共享的资源**，其中：

* **堆是进程中最大的一块内存，主要用于存放新创建的对象 (所有对象都在这里分配内存)**。
* **方法区主要用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据**。

**2、什么是上下文切换?**

　　即使单核处理器也支持多线程执行代码，**CPU通过给每个线程分配CPU时间片来实现这个机制**。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程是同时执行的。（时间片一般是几十毫秒）

　　CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以**任务从保存到加载的过程就是一次上下文切换**。**上下文切换会影响多线程的执行速度**。

**3、并发与并行？**

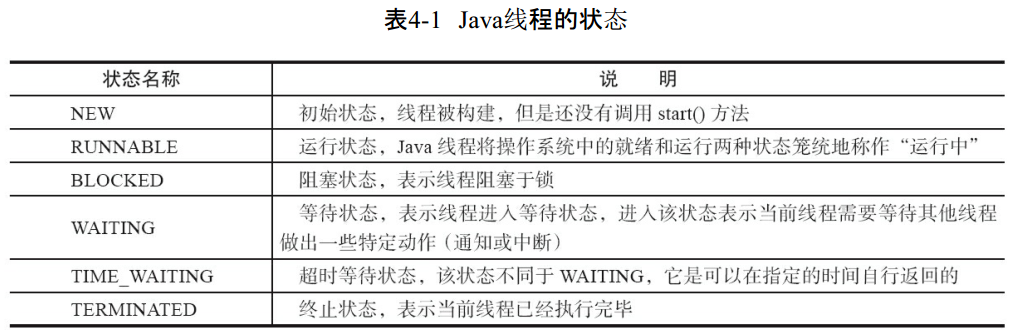
**并发指的是多个任务交替进行，并行则是指真正意义上的“同时进行”**。

　　实际上，如果系统内只有一个CPU，使用多线程时，在真实系统环境下不能并行，只能通过切换时间片的方式交替进行，从而并发执行任务。真正的并行只能出现在拥有多个CPU的系统中。

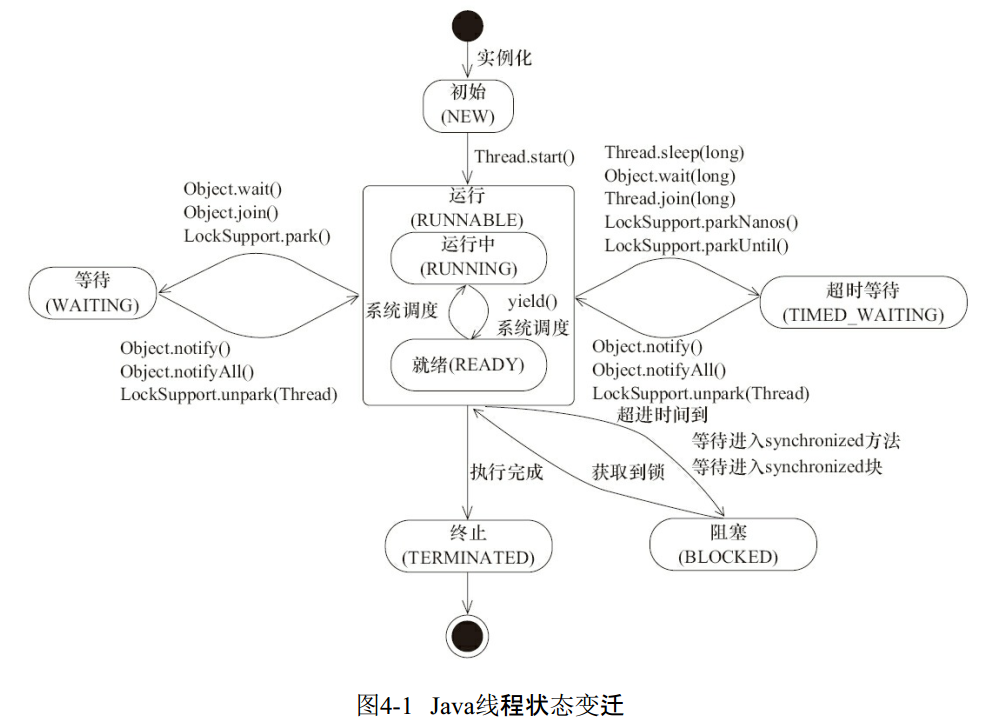
**4、线程的生命周期和状态?（重要！）**

 Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态：

**初始状态、运行状态、阻塞状态、等待状态、超时等待状态、终止状态**



**线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换**：



* 线程创建之后它将处于 **初始状态**(NEW)，**调用 start() 方法后开始运行，线程这时候处于 可运行状态**(READY)。
* **可运行状态的线程获得了 CPU 时间片后就处于 运行状态**(RUNNING)。
* 当**线程执行 wait()方法之后，线程进入 等待状态**(WAITING)，**进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态**【notify()】。 而 **超时等待状态**(TIME\_WAITING)**相当于在等待状态的基础上增加了超时限制，【sleep（long millis）/wait（long millis）】，当超时时间到达后 Java 线程将会返回到运行状态**。
* **当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到阻塞状态**(BLOCKED)。
* **线程在执行 Runnable 的run()方法之后将会进入到** **终止状态**(TERMINATED)。

**5、什么是线程死锁?如何避免死锁?**

 　　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。

　　假如线程 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。

**避免死锁的几个常见方法：**

* **避免一个线程同时获取多个锁**
* **避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。**
* **尝试使用定时锁，使用 lock.tryLock(timeout) 来代替使用内部锁机制。**
* **对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。**

**6、sleep() 方法和 wait() 方法区别和共同点?（重要！）**

**相同点：**

　　两者都可以暂停线程的执行，都会让线程进入等待状态。

**不同点：**

* **sleep()方法没有释放锁，而 wait()方法释放了锁。**
* **sleep()方法属于Thread类的静态方法，作用于当前线程；而wait()方法是Object类的实例方法，作用于对象本身。**
* **执行sleep()方法后，可以通过超时或者调用interrupt()方法唤醒休眠中的线程；执行wait()方法后，通过调用notify()或notifyAll()方法唤醒等待线程。**

**7、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？**

**new 一个 Thread，线程进入初始状态；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**。

**8、多线程开发带来的问题与解决方法？（重要）**

使用多线程主要会带来以下几个问题：

**（一）线程安全问题**

　　线程安全问题**指的是在某一线程从开始访问到结束访问某一数据期间，该数据被其他的线程所修改，那么对于当前线程而言，该线程就发生了线程安全问题，表现形式为数据的缺失，数据不一致等**。

　　线程安全问题发生的条件：

　　　　1）多线程环境下，即存在包括自己在内存在有多个线程。

　　　　2）多线程环境下存在共享资源，且多线程操作该共享资源。

　　　　3）多个线程必须对该共享资源有非原子性操作。

　　线程安全问题的解决思路：

**1）尽量不使用共享变量，将不必要的共享变量变成局部变量来使用**。

**2）使用synchronized关键字同步代码块，或者使用jdk包中提供的Lock为操作进行加锁**。

**3）使用ThreadLocal为每一个线程建立一个变量的副本，各个线程间独立操作，互不影响**。

**（二）性能问题**

　　线程的生命周期开销是非常大的，**一个线程的创建到销毁都会占用大量的内存**。同时如果不合理的创建了多个线程，cup的处理器数量小于了线程数量，那么将会有很多的线程被闲置，闲置的线程将会占用大量的内存，为垃圾回收带来很大压力，同时cup在分配线程时还会消耗其性能。

　　解决思路：

**利用线程池**，模拟一个池，预先创建有限合理个数的线程放入池中，当需要执行任务时从池中取出空闲的先去执行任务，执行完成后将线程归还到池中，这样就**减少了线程的频繁创建和销毁，节省内存开销**和减小了垃圾回收的压力。同时因为任务到来时本身线程已经存在，减少了创建线程时间，提高了执行效率，而且合理的创建线程池数量还会使各个线程都处于忙碌状态，提高任务执行效率，线程池还提供了拒绝策略，当任务数量到达某一临界区时，线程池将拒绝任务的进入，保持现有任务的顺利执行，减少池的压力。

**（三）活跃性问题**

　　1）**死锁**，**假如线程 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**。多个线程环形占用资源也是一样的会产生死锁问题。

　　解决方法：

* **避免一个线程同时获取多个锁**
* **避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。**
* **尝试使用定时锁，使用 lock.tryLock(timeout) 来代替使用内部锁机制。**

　　想要避免死锁，可以**使用无锁函数**（cas）**或者使用重入锁（ReentrantLock），通过重入锁使线程中断或限时等待可以有效的规避死锁问题**。

　　2）**饥饿**，饥饿**指的是某一线程或多个线程因为某些原因一直获取不到资源，导致程序一直无法执行**。如某一线程优先级太低导致一直分配不到资源，或者是某一线程一直占着某种资源不放，导致该线程无法执行等。

　　解决方法：

　　与死锁相比，饥饿现象还是有可能在一段时间之后恢复执行的。**可以设置合适的线程优先级来尽量避免饥饿的产生**。

　　3）**活锁**，活锁体现了一种谦让的美德，每个线程都想把资源让给对方，但是由于机器“智商”不够，可能会产生一直将资源让来让去，导致资源在两个线程间跳动而无法使某一线程真正的到资源并执行，这就是活锁的问题。

**（四）阻塞**

　　阻塞是用来形容多线程的问题，**几个线程之间共享临界区资源，那么当一个线程占用了临界区资源后，所有需要使用该资源的线程都需要进入该临界区等待，等待会导致线程挂起，一直不能工作，这种情况就是阻塞**，**如果某一线程一直都不释放资源，将会导致其他所有等待在这个临界区的线程都不能工作**。当我们使用synchronized或重入锁时，我们得到的就是阻塞线程，如论是synchronized或者重入锁，都会在试图执行代码前，得到临界区的锁，如果得不到锁，线程将会被挂起等待，知道其他线程执行完成并释放锁且拿到锁为止。

　　解决方法：

**可以通过减少锁持有时间，读写锁分离，减小锁的粒度，锁分离，锁粗化等方式来优化锁的性能**。

临界区：

　　临界区是用来表示一种公共的资源（共享数据），它可以被多个线程使用，但是在每次只能有一个线程能够使用它，当临界区资源正在被一个线程使用时，其他的线程就只能等待当前线程执行完之后才能使用该临界区资源。

　　比如办公室办公室里有一支笔，它一次只能被一个人使用，假如它正在被甲使用时，其他想要使用这支笔的人只能等甲使用完这支笔之后才能允许另一个人去使用。这就是临界区的概念。

参考 <https://www.cnblogs.com/Eternally-dream/p/9678314.html>

**9、 synchronized 关键字**

synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

synchronized关键字最主要的三种使用方式：**修饰实例方法:、修饰静态方法、修饰代码块。**

* **对于普通同步方法，锁是当前实例对象。**
* **对于静态同步方法，锁是当前类的Class对象。**
* **对于同步代码块，锁是synchronized括号里配置的对象。**

　　当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。

**synchronized在JVM里是怎么实现的？**

**synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置**。 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因)

**synchronized用的锁是存在哪里的？**

　　synchronized用到的锁是存在**Java对象头**里的。

**10、说说 JDK1.6 之后的synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗**

　　JDK1.6 对锁的实现引入了大量的优化，如**偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化**等技术来减少锁操作的开销。

**锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态**，他们会随着竞争的激烈而逐渐升级。注意**锁可以升级不可降级**，这种策略是为了提高获得锁和释放锁的效率。

　　关于这几种优化的详细信息可以查看这篇文章：<https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/Multithread/synchronized.md>

**11、synchronized和 Lock 的区别？（重要）**

1）**Lock是一个接口，而synchronized是Java中的关键字**，synchronized是内置的语言实现；

2）**synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生**；而**Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁**；

3）**Lock可以让等待锁的线程响应中断**，而**synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断**；

4）**通过Lock可以知道有没有成功获取锁（tryLock()方法：如果获取锁成功，则返回true）**，而**synchronized却无法办到**。

5）**Lock可以提高多个线程进行读操作的效率**。

　　在性能上来说，如果**竞争资源不激烈，两者的性能是差不多的**，而当**竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized**。所以说，在具体使用时要根据适当情况选择。

参考<https://blog.csdn.net/qq_38200548/article/details/82943222>

**12、synchronized和ReentrantLock（重入锁） 的区别？**

* 两者都是**可重进入锁**，就是**能够支持一个线程对资源的重复加锁。sychnronized关键字隐式的支持重进入**，比如一个sychnronized修饰的递归方法，在方法执行时，执行线程在获取了锁之后仍能连续多次地获取该锁。**ReentrantLock**虽然没能像sychnronized关键字一样隐式的重进入，但是**在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不被阻塞**。
  + 线程重复n次获取了锁，随后在第n次释放该锁后，其他线程能够获取到该锁。锁的最终释放要求**锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于0时表示锁已经成功被释放**。
* **synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API**。ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成）
* **ReentrantLock 比 synchronized 增加了一些高级功能，主要有3点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）**
  + **ReentrantLock提供了一种能够中断等待锁的线程的机制**，也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。通过lock.lockInterruptibly()来实现这个机制。
  + **ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。**（公平锁就是先等待的线程先获得锁）
  + synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制。ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition() 方法。用ReentrantLock类结合Condition实例可以实现“选择性通知” 。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程

**13、volatile关键字**

**保证共享变量的“可见性”**。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。

　　把变量声明为volatile，这就指示 JVM每次使用它都到主存中进行读取。

**14、synchronized 关键字和 volatile 关键字的区别**

* volatile关键字是线程同步的轻量级实现，所以**volatile性能比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**。
* **多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞**。、
* **volatile**关键字主要用于**解决变量在多个线程之间的可见性**，而 **synchronized**关键字解决的是**多个线程之间访问资源的同步性**。
* volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。

**15、使用线程池的好处？**

1. **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**。
2. **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**。
3. **提高线程的可管理性**。**线程是稀缺资源，如果无限制地创建，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分配、调优和监控**。

**16、说一说几种常见的线程池及适用场景？（重要）**

　　可以创建（**Executors.newXXX**）3种类型的ThreadPoolExecutor：**FixedThreadPool**、**SingleThreadExecutor**、**CachedThreadPool**。

* **FixedThreadPool**：**可重用固定线程数的线程池**。（适用于负载比较重的服务器）
  + **FixedThreadPool使用无界队列LinkedBlockingQueue作为线程池的工作队列**
  + 该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。
* **SingleThreadExecutor**：**只会创建一个线程执行任务。**（适用于需要保证顺序执行各个任务；并且在任意时间点，没有多线程活动的场景。）
  + **SingleThreadExecutorl也使用无界队列LinkedBlockingQueue作为工作队列**
  + 若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
* **CachedThreadPool：是一个会根据需要调整线程数量的线程池。**（大小无界，适用于执行很多的短期异步任务的小程序，或负载较轻的服务器）
  + **CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。**
  + 线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。
* **ScheduledThreadPool**：继承自ThreadPoolExecutor。它主要用来**在给定的延迟之后运行任务，或者定期执行任务**。使用DelayQueue作为任务队列。

**16、线程池都有哪几种工作队列？（重要）**

* **ArrayBlockingQueue**：是一个**基于数组结构的有界阻塞队列**，此队列按FIFO（先进先出）原则对元素进行排序。
* **LinkedBlockingQueue**：是一个**基于链表结构的阻塞队列**，此队列按FIFO排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。
* **SynchronousQueue**：是一个**不存储元素的阻塞队列**。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于Linked-BlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
* **PriorityBlockingQueue**：一个**具有优先级的无限阻塞队列**。

**17、创建线程的几种方式？（重要）**

有4种方式：**继承Thread类**、**实现Runnable接口**、**实现Callable接口**、**使用Executor框架来创建线程池**。

**（1）通过继承Thread类创建线程**

public class MyThread **extends Thread**{//继承Thread类

**//重写run方法**

　　public void run(){

　　}

}

----------------------------------------------------------------------------------

public class Main {

　　public static void main(String[] args){

　　　　new MyThread()**.start()**; //创建并启动线程

　　}

}

**（2）通过实现Runnable接口来创建线程**

public class MyThread2 **implements Runnable** {//实现Runnable接口

**//重写run方法**

　　public void run(){

　　}

}

------------------------------------------------------------------------------------------

public class Main {

　　public static void main(String[] args){

　　　　//创建并启动线程

　　　　MyThread2 myThread=new MyThread2();

　　　　Thread thread=new Thread(myThread);

　　　　thread()**.start()**;

　　　　//或者    new Thread(new MyThread2()).start();

　　}

}

不管是继承Thread还是实现Runnable接口，多线程代码都是通过运行Thread的start()方法来运行的。

**（3）实现Callable接口来创建线程**

与实现Runnable接口类似，和Runnable接口不同的是，Callable接口提供了一个call() 方法作为线程执行体，call()方法比run()方法功能要强大：call()方法可以有返回值、call()方法可以声明抛出异常。

public class Main {

　　public static void main(String[] args){

　　　MyThread3 th=new MyThread3();

　　　//使用Lambda表达式创建Callable对象

　　   //使用FutureTask类来包装Callable对象

**Future**Task<Integer> future=new FutureTask<Integer>(

　　　　(Callable<Integer>)()->{

　　　　　　return 5;

　　　　}

　　  );

　　　new Thread(task,"有返回值的线程")**.start()**;//实质上还是以Callable对象来创建并启动线程

　　  try{

　　　　System.out.println("子线程的返回值："+future.get());//get()方法会阻塞，直到子线程执行结束才返回

 　　 }catch(Exception e){

　　　　ex.printStackTrace();

　　　}

　　}

}

**（4）使用Executor框架来创建线程池**

**Executors.newXXXX： newFixedThreadPool(int )、newSingleThreadExecutor、newCachedThreadPool、newScheduledThreadPool(int)**

通过Executors的以上四个静态工厂方法获得**ExecutorService**实例**，**而后可以执行Runnable任务或Callable任务。

* **Executor执行Runnable任务**：

　　通过Executors的以上四个静态工厂方法获得 ExecutorService实例，而后调用该实例的execute（Runnable command）方法即可。一旦Runnable任务传递到execute（）方法，该方法便会自动在一个线程上。

[复制代码](javascript:void(0);)

public class TestCachedThreadPool{

public static void main(String[] args){

ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 5; i++){

executorService.execute(new TestRunnable());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* a" + i + " \*\*\*\*\*\*\*\*\*\*\*\*\*");

}

executorService.shutdown();

}

}

class TestRunnable implements Runnable{   
 //重写run方法

public void run(){

System.out.println(Thread.currentThread().getName() + "线程被调用了。");

}

[复制代码](javascript:void(0);)

* **Executor执行Callable任务：**

　　当将一个Callable的对象传递给ExecutorService的submit方法，则该call方法自动在一个线程上执行，并且**会返回执行结果Future对象**。同样，将Runnable的对象传递给ExecutorService的submit方法，则该run方法自动在一个线程上执行，并且会返回执行结果Future对象，但是在该Future对象上调用get方法，将返回null。

[复制代码](javascript:void(0);)

public class CallableDemo{

public static void main(String[] args){

ExecutorService executorService = Executors.newCachedThreadPool();

List<Future<String>> resultList = new ArrayList<Future<String>>();

//创建10个任务并执行

for (int i = 0; i < 10; i++){

//使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中

Future<String> future = executorService.submit(new TaskWithResult(i));

//将任务执行结果存储到List中

resultList.add(future);

}

//遍历任务的结果

for (Future<String> fs : resultList){

try{

while(!fs.isDone);//Future返回如果没有完成，则一直循环等待，直到Future返回完成

System.out.println(fs.get()); //打印各个线程（任务）执行的结果

}catch(InterruptedException e){

e.printStackTrace();

}catch(ExecutionException e){

e.printStackTrace();

}finally{

//启动一次顺序关闭，执行以前提交的任务，但不接受新任务

executorService.shutdown();

}

}

}

}

class TaskWithResult implements Callable<String>{

private int id;

public TaskWithResult(int id){

this.id = id;

}

// 重写call()方法

public String call() throws Exception {

System.out.println("call()方法被自动调用！！！ " + Thread.currentThread().getName());

//该返回结果将被Future的get方法得到

return "call()方法被自动调用，任务返回的结果是：" + id + " " + Thread.currentThread().getName();

}

}

[复制代码](javascript:void(0);)

实现Runnable接口和Callable接口的区别？

 Runnable接口或Callable接口实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。两者的区别在于 **Runnable 接口不会返回结果但是 Callable 接口可以返回结果**。

**执行execute()方法和submit()方法的区别是什么呢？**

1)**execute() 方法用于提交不需要返回值的任务**，所以无法判断任务是否被线程池执行成功与否；

2) **submit() 方法用于提交需要返回值的任务**。线程池会返回一个**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()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

**18.线程池参数？**

①**corePoolSize：线程池的基本大小**，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。说白了就是，即便是线程池里没有任何任务，也会有corePoolSize个线程在候着等任务。

②**maximumPoolSize:最大线程数**，不管你提交多少任务，线程池里最多工作线程数就是maximumPoolSize。

③**keepAliveTime:线程的存活时间**。当线程池里的线程数大于corePoolSize时，如果等了keepAliveTime时长还没有任务可执行，则线程退出。

⑤**unit**：这个用来指定keepAliveTime的单位，比如秒:TimeUnit.SECONDS。

⑥**workQueu**e：用于保存等待执行任务的**阻塞队列**，提交的任务将会被放到这个队列里。

⑦**threadFactory：线程工厂，用来创建线程**。主要是为了给线程起名字，默认工厂的线程名字：pool-1-thread-3。

⑧**handler：拒绝策略**，即当线程和队列都已经满了的时候，应该采取什么样的策略来处理新提交的任务。默认策略是AbortPolicy（抛出异常），其他的策略还有：CallerRunsPolicy(只用调用者所在线程来运行任务)、DiscardOldestPolicy(丢弃队列里最近的一个任务，并执行当前任务)、DiscardPolicy(不处理，丢弃掉)

**19.线程池执行流程?**

　　任务被提交到线程池，会先判断当前线程数量是否小于corePoolSize，如果小于则创建线程来执行提交的任务，否则将任务放入workQueue队列，如果workQueue满了，则判断当前线程数量是否小于maximumPoolSize,如果小于则创建线程执行任务，否则就会调用handler，以表示线程池拒绝接收任务。