Java 并发编程

1. 并发编程的挑战

1. 上下文切换指什么？多线程一定快吗？为什么？
   a) CPU 通过时间分配算法来循环执行任务，当前线程执行一个时间片后会切换到下一个线程，但在切换前会保留上一个任务的状态，以便下次切换回这个状态时，能够再加载这个任务的状态，**所以任务从保存到再加载就是一次上下文切换。**
   b) 因为线程有创建和上下文切换的开销。在并行执行操作不多的时候，多线程不一定好；当计算机资源限制比较严格的情况下，并行执行都会变为串行执行，这种情况下也不适合。
2. 用什么工具可以度量上下文切换次数和时长？如何减少上下文切换？
   a) Lmbench3 测切换时长
   b) Vmstat 测切换次数
   c) **减少上下文切换的办法有以下几种：**
      i. **无锁并发编程->如将数据的 ID 按照 Hash 算法取模分段，不同的线程处理不同段的数据()**
      ii. **CAS 算法。Java 的 Atomic 包使用 CAS 算法来更新数据，不需要加锁**
      iii. **使用最少线程，任务很少，线程很多，大多数线程都会处于等待状态。上下文切换也会增多**
      iv. **使用协程：单线程内任务调度，维持多个任务之间的切换。**
   d) 减少上下文切换实例：
      i. **通过 jstack 命令查看 dump 线程信息，**
      ii. 统计所有线程都处于上么状态，如果多数线程处于 WAITING 状态
      iii. 打开 dump 文件查看处于 WAITING 的线程在做什么，如果是接到的任务太少导致线程空闲。
      iv. 减少该类线程的数量。
3. **死锁出现的场景？产生死锁的原因？**
   a) **系统资源的竞争(只有对不可剥夺资源的竞争才可能产生死锁)**
   b) **进程推进顺序非法(请求和释放资源的顺序不对)**
4. **产生死锁的必要条件？**
   a) **互斥条件：在一段时间内，某资源只被一个进程占有。**
   b) **不剥夺条件：进程获得的资源在未使用完毕之前，不能被其他进程强行夺走。**
   c) **请求和保持条件：进程已经保持了至少一个资源，但又提出新的资源请求，而资源被其他线程占有，则请求进程被阻塞，而自己对已获得的资源保持不放。**
   d) **循环等待条件：存在一种进程资源的循环等待链，链中每个进程已获得的资源同时被链中下一个进程锁请求。**
5. **死锁的处理策略？**
   a) **预防死锁：破坏产生思索四个必要条件中的一个或几个，以防止发生死锁。**
   b) **避免死锁：在资源的动态分配过程中，用某种方法防止系统进入不安全的状态，从而避免死锁。**

c) **死锁检测：允许进程在运行过程中发生思索，通过系统检测机构及时地检测出死锁的发生，然后采取某种措施解除死锁。**

表 2-14 死锁处理策略的比较

	资源分配策略	各种可能模式	主要优点	主要缺点
死锁预防	保守，宁可资源闲置	一次请求所有资源，资源剥夺，资源按序分配	适用于做突发式处理的进程，不必进行剥夺	效率低，进程初始化时间延长；剥夺次数过多；不便灵活申请新资源
死锁避免	是“预防”和“检测”的折中（在运行时判断是否可能死锁）	寻找可能的安全允许顺序	不必进行剥夺	必须知道将来的资源需求；进程不能被长时间阻塞
死锁检测	宽松，只要允许就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间，允许对死锁进行现场处理	通过剥夺解除死锁，造成损失

6. 如何避免死锁的几种方法？

a) 避免一个线程同时获得多个锁
b) 避免一个线程在锁内占用多个资源，尽量保证每个锁只占用一个资源
c) 尝试使用定时锁，使用 lock.tryLock(timeout)来替代使用内部锁机制
d) 对于数据库锁，加锁和解锁必须在数据库连接里，否则会出现解锁失败的现象

7. 什么是资源限制？资源限制引发的问题？如何解决资源限制问题？如何在资源限制的条件下进行并发编程？

a) 在进行并发编程时，程序的执行速度受限于计算机硬件资源和软件资源
b) 在并发编程中，将代码执行速度加快的原因就是将串行变成了并发执行，资源受限导致仍串行执行+上下文切换和资源调度的时间
c) 硬件->**集群**并行执行程序 || 软件->使用**资源池**将资源复用
d) 根据不同的资源调整程序并发数

2. 并发机制的底层实现原理

1. 关于 volatile 的理解？它的实现原理是什么？如何进行优化？

a) 定义：**java 编程语言允许线程访问共享变量，为了确保这个共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获取这个变量**
b) 实现原理：volatile 变量修饰的共享变量在进行写操作时，调用 Lock 前缀指令：
   i. 将当前处理器缓存行的数据写回系统内存
   ii. 这个写回内存的操作会使其他 CPU 中缓存该内存地址的数据无效。
c) 优化：追加字节优化性能。什么时候不能通过这种方式优化？
   i. 队列的入队和出队时需要不停地改变头节点和尾节点，追加字节能够避免头尾节点加到同一个缓存行，使头尾节点修改时不会相互锁定。
d) Volatile 是一种较弱的同步机制，可以解决共享变量的可见性问题，如果变量被声明为 volatile，那么编译器和 JVM 会把变量当作共享变量并禁止对该变量的一些重排序操作，修改变量后立刻对所有线程可见。但 volatile 无法解决原子性问题和一致性问题。

2. Synchronized 实现同步的基础，有哪几种表现形式？Synchronized 在 JVM 中的实现原理？

a) JVM 进入和退出 monitor 对象来实现同步和代码块的同步，代码块同步和方法同步有所不同。
b) Monitorenter 指令是在编译后插入到同步代码的开始位置，monitorexit 是插入到方

法的结束和异常处，它们可以实现 sychronized 的可重入性。

3. 锁的状态有几种？偏向锁的目的和实现原理？各种锁的使用场景和优缺点？

a) 偏向锁：

i. 大多数情况下，锁不仅不存在着竞争，而是总是由同一线程多次获得，为了使线程获得锁的代价更低，而引入偏向锁，当一个线程访问同步代码块获得锁时，会在对象头和栈帧中的锁记录里记录锁偏向的线程 ID,以后**该锁在进入和退出同步代码块时不需要进行 CAS 操作来加锁和解锁。在出现锁竞争后，偏向锁释放，转变为轻量锁。**

ii. 加锁和解锁不需要额外的消耗，如果线程间存在锁竞争，会带来额外的锁撤销的消耗，适用于一个线程访问同步代码块的情形

b) 轻量级锁：

i. 竞争的线程不会阻塞，提高程序的响应速度，如果始终得不到锁竞争的线程，使用自旋会消耗 CPU，追求响应时间，同步块执行速度快

c) 重量级锁：

i. 线程竞争不使用自旋，不会消耗 CPU 线程阻塞，响应时间缓慢，追求吞吐量，同步快执行速度较长

4. 原子操作的实现原理？

a) 第一个机制是使用**总线锁**保证原子性

b) 使用**缓存锁**保证原子性

5. Java 如何实现原子操作？

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。

通过锁或循环 CAS 来实现原子操作。

问题：

(1) aba 问题->使用 AtomicStampedReference 来解决

(2) 循环开销时间大

(3) 只能保证一个共享变量的原子操作，使用 AtomicReference 类保证引用对象之间的原子性。

除了偏向锁，当一个线程想要进入同步代码块时使用循环 CAS 来获取锁，当它想退出同步代码块的时候使用循环 CAS 释放锁。

3. Java 内存模型

1. 并发编程中两大关键问题，线程之间如何通信？线程之间如何同步？

a) 在命令式编程中，线程之间通信机制有**共享内存**和**消息传递**。

b) 共享内存模型中，同步是显式进行的，而消息传递模型中，同步是隐式进行的。

2. 从线程 A 到线程 B 通信的步骤：

a) 线程 A 把本地内存 A 更新过的共享变量刷新到主内存中。

b) 线程 B 通过主内存把线程 A 更新后的共享变量读如线程 B 的本地内存中。

3. Java 代码的指令重排序：编译器优化重排序、指令级并行重排序、内存系统重排序。通过内存屏障来保证指令序列的一定程度上有序。

4. 指令重排序，在单线程看来，并没有什么影响，但在多线程程序中，可能产生不一样的结果。

5. 顺序一致性：如果程序是正确同步的，程序的执行将具有顺序一致性，即程序的执行结果

与该程序在顺序一致性内存模型中的执行结果相同。两大特征：

a) 一个线程中的所有操作必须按照程序的顺序来执行。

b) 不管程序是否同步，所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见。

但是 JMM 模型并没与保证顺序一致性，未同步的程序在 JMM 中不但整体的执行顺序是无序的，而且所有线程看到的操作执行顺序也可能不一致。

未同步的程序在 JMM 中执行时，整体上是无序的，其执行结果无法预知。

6. Volatile 内存语义？见 JVM62 条。

a) 当写一个 volatile 变量时，JMM 会把该线程对应的本地内存中的共享变量值刷新到主内存

b) 当读一个 volatile 变量时，JMM 会把该线程对应的本地内存置为无效。线程接下来从主内存中读取共享变量。

c) Volatile 的禁止重排序，实质上是利用在 volatile 的读和写的前后插入内存屏障来实现的。

7. 锁的内存语义？

a) 当线程释放锁，JMM 会把线程对应的本地内存中的共享变量刷新到主内存中。

b) 当线程获得锁，JMM 会把该线程对应的本地内存置为无效，从而保证临界区代码必须从主内存中读取共享变量。

这里可以看出，锁的获取和释放和 volatile 的读和写具有相同语义。

8. Final 的内存语义？

a) 在构造函数内对一个 final 域写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作不能重排序。(写 final 域一定在 final 域初始化之后)

b) 初次读一个包含 final 域的对象的引用，与随后初次读这个 final 域，这两个操作之间不能重排序。(读 final 域之前，一定先读包含 final 域的对象的引用)

Final 语义必须保证 final 引用不能从构造函数中逃逸，否则可能读到未初始化的值。

9. Happens-before 是 JMM 最核心的概念，出发点是保证程序员足够强的内存可见性保证。

a) JMM 遵循的基本原则：只要不改变程序的执行结果，编译器和处理器怎么优化都行。

b) As-if-serial 语义和 happens-before 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。

c) Happens-before 的具体规则见 JVM64 条。

10. 请写出线程安全的单例模式？为什么需要双重检查？为什么实例需要增加 volatile？你还知道其他的线程安全的方案吗？这两种实现的区别？

11. Java 内存模型综述

a) 顺序一致性内存模型是一个理论参考模型，JMM 和处理器内存模型在设计时通常会以顺序一致性内存模型为参考。

b) 由于常见的处理器内存模型比 JMM 要弱，Java 编译器在生成字节码时，会在执行命令序列的适当位置插入内存屏障来限制处理器的重排序。

c) JMM 屏蔽了不同处理器内存模型的差异，它在不同的处理器平台上为 Java 程序员呈现了一个一致的内存模型。

d) JMM 内存可见性保证：

i. 单线程程序，不会出现内存可见性问题，编译器、runtime 和处理器共同确保单线程程序的执行结果与该程序在顺序一致性模型中的执行结果相同。

ii. 正确同步的多线程程序，将具有顺序一致性，执行结果与该程序在顺序一致性

内存模型中的执行结果相同。

iii. 未同步/未正确同步的多线程程序，JMM 为它们提供了最小安全性保障：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值(0、null、false)。

e) JSR-133 对旧内存模型的修补：

i. 增强 volatile 的内存语义：旧模型允许 volatile 变量与普通变量重排序。JSR-133 严格限制 volatile 变量与普通变量的重排序，使 volatile 的写-读和锁的释放-获取具有相同的语义。

ii. 增强 final 的内存语义。在旧内存模型中，多次读取同一个 final 变量的值可能会不相同，为此 jsr-133 为 final 增加了两个重排序规则，在保证 final 引用不会从构造函数内溢出的情况下，final 具有初始化安全性。

4. Java 并发编程基础

1. 什么是线程？见 JVM65 条

现代操作系统调度的最小单元是线程，也叫轻量级进程。

为什么要使用多线程？

更多的处理器核心；更快的响应速度；更好的编程模型；

线程的优先级？

通过一个整数变量 priority 来控制优先级，优先级的范围是 1~10。

2. 线程的状态：见 JVM68 条

a) NEW 初始状态，线程被构建，但是还没有调用 start()方法；

b) RUNNABLE 运行状态=就绪 or 运行

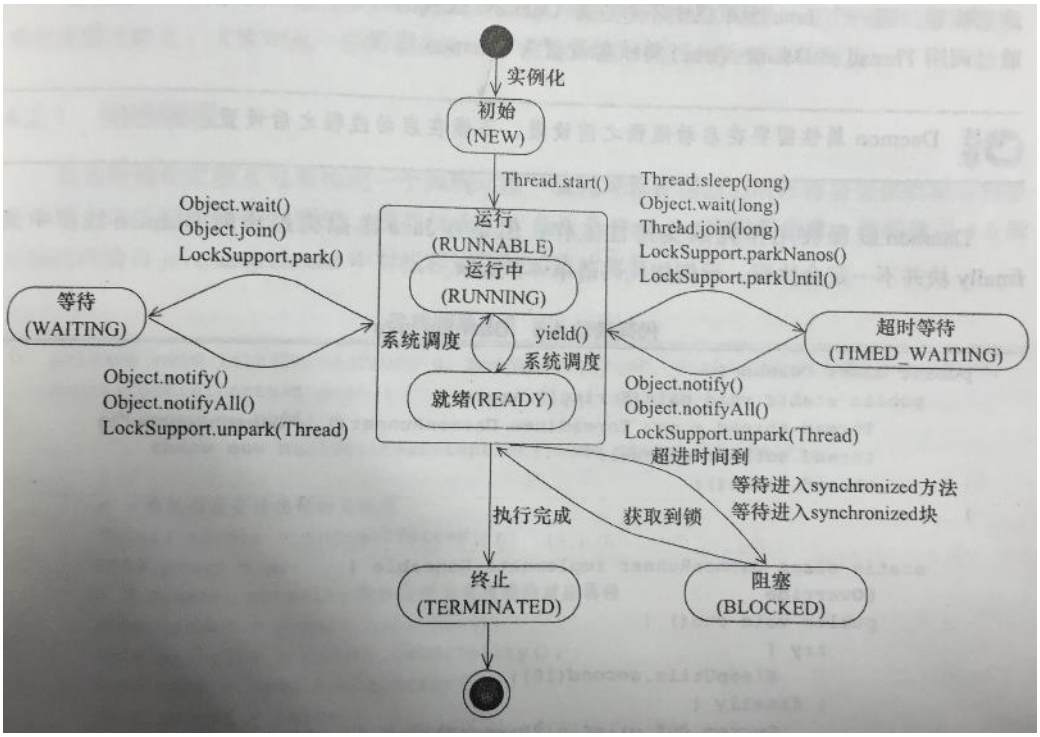
c) BLOCKED 阻塞状态

d) WAITING 等待状态，表示当前线程需要等待其他线程做出一些特定的动作

e) TIME_WAITING 指定时间自行返回

f) TERMINATED 终止状态，表示该线程已经结束，执行完 run()方法之后。

线程之间转换的关系：



3. Daemon 线程是什么？
   a) 程序中后台调度和支持性工作，守护线程。
   b) Java 虚拟机退出 Daemon 线程中的 finally 块不一定执行。

4. 启动和终止线程的详细介绍：
   a) 构造线程：一个新构造的线程对象是由其 parent 线程来进行空间分配，而 **child 线程继承了 parent 是否是 Daemon，优先级和加载资源的 contextClassLoader 以及可继承 ThreadLocal，同时分配一个唯一的 ID 来表示这个 child 线程。**
   b) 启动线程
   c) 理解中断->线程的一个标识位属性，其他线程调用该线程的 interupt()方法对其进行中断操作。
   d) 过期的 suspend() resume() stop()：原因：调用暂停后，线程不会释放占有的资源，而是占有着资源进入睡眠状态，这样容易发生死锁。
   e) 安全地终止线程。->通过标志位和中断操作结束线程，有机会去清理资源。

5. 线程间的通信
   a) Volatile 和 synchronized 关键字 ->确保所有线程对变量的可见性||确保所有线程对变量的可见性和排他性。
   b) 任意对象对 Object 的访问(由 synchronized 保护)，首先要获得 Object 的监视器。如果获取失败，线程进入**同步队列**，线程状态变为 **BLOCKED**，当访问 Object 的前驱释放了锁，则该释放锁的线程唤醒阻塞在同步队列中的线程。使其尝试重新对监视器的获取。
   c) 等待/通知机制(对应 Lock 的 Condition)
      i. 使用 wait(),notify(),notifyAll()事需要先对调用对象加锁；
      ii. 调用 wait()方法后，线程状态由 RUNNING 变为 WAITING，并将当前线程放置到等待队列中；
      iii. Notify(),notifyAll()方法调用后，等待线程仍然不会从 wait()方法返回，需等调用前面方法的线程释放锁之后，等待线程才有机会从 wait()返回；

iv. Notify()，notifyAll 的区别，进行方法后由 WAITING 状态变为 BLOCKED 状态；

v. 从 wait()方法返回的前提是获得了调用对象的锁；

d) 等待/通知的经典范式

等待方：

1) 获取对象的锁

2) 如果条件不满足，那么调用 wait()方法，被通知后仍要检查条件

3) 条件满足，执行对应的逻辑

通知方：

1) 获取对象的锁

2) 改变条件

3) 通知所有等待在对象上的线程

e) 管道输入/输出流 PipedOutputStream、PipedInputStream、PipedReader、PipedWriter

f) Thread.join()的使用

如果一个线程 a 执行了线程 b.join()=当前 a 等待 b 线程终止后才能从 b.join 返回；

g) ThreadLocal 的使用->线程变量，以 ThreadLocal 对象为键，任意对象为值的存储结构。一个线程可以根据一个 ThreadLocal 对象查询到绑定在该线程上的值。

6. 线程应用实例

a) 等待超时模式--数据库连接池实例

b) 线程池技术及其实例--预先创建若干数量的线程，并且不能由用户对线程的创建进行控制，在这个前提下，重复使用固定或较为固定数目的线程来完成任务的执行->优点

i. 消除频繁地创建和消亡线程的系统资源的开销，

ii. 面对过量的任务提交能够平缓地劣化

线程池的本质就是使用一个线程安全的工作队列连接工作者线程和客户端线程，客户端线程将任务放到工作队列后便返回，而工作者线程从队列上取出工作并执行。

c) 一个基于线程池技术的简单 web 服务器

7. Java 中如何停止一个线程

a) Java 提供了很丰富的 API 但没有为停止线程提供 API。JDK 1.0 本来有一些像 stop(), suspend() 和 resume()的控制方法但是由于潜在的死锁威胁因此在后续的 JDK 版本中他们被弃用了，之后 Java API 的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。当 run() 或者 call() 方法执行完的时候线程会自动结束,如果要手动结束一个线程，你可以用 volatile 布尔变量来退出 run()方法的循环或者是取消任务来中断线程。

8. 一个线程运行时发生异常会怎样？

a) 简单的说，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。

9. 如何在两个线程间共享数据？

a) 你可以通过**共享对象**来实现这个目的，或者使用**像阻塞队列这样并发的数据结构**。

10. 为什么 wait, notify 和 notifyAll 这些方法不在 thread 类里面？

a) 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。

11. Java 中 interrupted 和 isInterrupted 方法的区别？

a) interrupted() 和 isInterrupted()的主要区别是前者会将中断状态清除而后者不会。Java 多线程的中断机制是用内部标识来实现的，调用 Thread.interrupt()来中断一个线程就会设置中断标识为 true。当中断线程调用静态方法 Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法 isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出 InterruptedException 异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有可能被其它线程调用中断来改变。

12. 为什么 wait 和 notify 方法要在同步块中调用？

a) 主要是因为 Java API 强制要求这样做，如果你不这么做，你的代码会抛出 IllegalMonitorStateException 异常。还有一个原因是为了避免 wait 和 notify 之间产生竞态条件。只有当线程拥有对象的独占锁时，才能调用这些方法，，这就意味调用某对象的 wait()方法时，当前线程就已经获取该对象的锁了。

13. 为什么你应该在循环中检查等待条件?

a) 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。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在 notify()方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用 wait()方法效果更好的原因

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

a) 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。

15. 什么是死锁(Deadlock)？如何分析和避免死锁？

a) 死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上的资源。

b) 分析死锁 jstack，我们需要查看 Java 应用程序的线程转储。我们需要找出那些状态为 BLOCKED 的线程和他们等待的资源。每个资源都有一个唯一的 id，用这个 id 我们可以找出哪些线程已经拥有了它的对象锁。

c) 避免嵌套锁，只在需要的地方使用锁和避免无限期等待是避免死锁的通常办法

5. Java 中的锁

1. Lock 接口的出现-->

a) Lock lock = new ReentrantLock();
Lock.lock();
Try{
} finally{
Lock.unlock();
}---->不要将锁获取写在 try 中！

b) Lock 具备 synchronized 所不具备的特性：
i. 尝试非阻塞地获取锁
ii. 能被中断地获取锁

iii. 超时获取锁

2. 队列同步器

a) 定义：可以用来构建锁或者其他同步组件的基础框架，它使用一个 int 成员变量表示**同步状态**，通过内置的 FIFO 队列来完成资源获取线程的排队。

b) 同步器的主要工作方式是**继承**，子类通过继承同步器并实现它的抽象方法来管理同步状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的三个方法。**子类被推荐定义为自定义同步的**静态内部类。

锁是面向使用者的，它定义了使用者与锁交互的接口(比如可以允许两个线程并行访问)，隐藏了实现细节；同步器面向的是锁的实现者。

c) 队列同步器的接口与示例

i. getState() 获取当前同步状态

ii. setState(int newState) 设置当前同步状态

iii. compareAndSetState（int expect, int update）使用 CAS 设置当前状态，该方法能够保证状态设置的原子性

iv. 同步器提供的模板方法基本上有三类：**独占式获取与释放同步状态，共享式获取和释放同步状态，查询同步队列中的等待情况**

3. 队列同步器的实现分析

a) 同步队列

i. 同步队列中的节点用来保存**获取同步状态失败的线程引用，等待状态**以及**前驱和后继节点**。节点是构成同步队列的基础，同步器拥有首节点和尾节点，没有成功获取同步状态的线程将会成为节点加入该队列的**尾部**。**同步器中包含两个首尾节点的引用**，首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，**而后继节点将会在获取同步状态成功时将自己设置为首节点。**

b) 独占式同步状态获取与释放

i. 通过同步器的 acquire(int arg)方法可以获取同步状态，该方法对中断不敏感，也就是线程获取同步状态失败后进入同步队列中，后续线程进行中断操作时，线程不会从同步队列中移出。**Enq(final Node node)方法将并发添加节点的请求通过 CAS 变得“串行化”了，该方法为节点的构造和加入同步队列中调用的方法。**

ii. 只有前驱节点是头节点才能获取同步状态。

1. 头节点是成功获取到同步状态的节点，而头节点的线程释放了同步状态之后，将会唤醒其后继节点，后继节点的线程被唤醒后需要检查自己的前驱结点是否是头节点

2. 维护同步队列的 FIFO 原则。该方法中，节点自旋获取同步状态

Node.prev = head && tryAcquire(args)

iii. 前驱结点为头节点且能够获取同步状态的判断条件和线程进入等待状态是获取同步状态的自旋过程。

iv. 总结：在获取同步状态时，同步器维护了一个同步队列，获取状态失败的线程都会被加到队列中并在队列中进行自旋；移出队列或停止自旋的条件是前驱节点为头节点且成功获取了同步状态。在释放同步状态时，同步器调用 tryRelaease()方法释放同步状态，然后唤醒后继节点。

c) 共享式同步状态的获取与释放

i. 共享式获取与独占是获取最主要的区别是同一时刻能否有多个线程获取到同步状态。

ii. 共享式同步状态获取同步状态和释放同步状态的过程。

d) 独占式超时获取同步状态

i. 通过调用同步器的 doAcquireNanos(int arg, long nanosTimeout)方法可以超时获取同步状态。

ii. 获取同步状态的流程 p133

e) 自定义同步组件

i. TwinsLock 能够在同一时刻支持多个线程的访问，这显然是共享式访问。同时必须重写 tryAcquireShared(int args) 和 tryReleaseShared(int args)方法。

ii. 定义资源数，设置初始状态 status. 在同步状态变更时，需要使用 compareAndSet(int expect, int update)方法做原子保证，一般来说自定义同步器会被定义为自定义同步组件的内部类。

```
Public class TwinsLock implements Lock{
    Private final Sync sync = new Sync(2);
    Private static final class Sync extends AbstractQueuedSynchronized{
        Sync(int count){
            If (count <= 0){
                Throw new IllegalArgumentException("XXXX");
            }
            setState(count);
        }

        Public int tryAcquireShared(int reduceCount){
            For(;;){
                Int current = getState();
                Int newCount = current-reduceCount;
                If(newCount <0 || compareAndSetState(current, newCount)){
                    Return newCount;
                }
            }
        }

        Public boolean tryReleaseShared(int returnCount){
            For(;;){
                Int current = getState();
                Int newCount = current+returnCount;
                If(compareAndSetState(current, newCount)){
                    Return true;
                }
            }
        }
    }

    Public void lock(){
        sync.acquireShared(1);
```

```
        }

        Public void unlock(){
            sync.releaseShared(1);
        }
    }
```

4. 重入锁

a) 该锁能够支持一个线程对资源的重复加锁，除此之外，该锁还支持获取锁时的公平或不公平性选择

i. 如果在绝对的时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的。

ii. ReentrantLock 提供了一个构造函数能够控制锁是否公平

b) 实现重进入需要解决的问题：

i. 线程再次获得锁，锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次获取

ii. 锁的最终释放。通过锁的计数器的自增和自减操作来实现锁的最终释放

iii. 如果该锁被获取了 n 次，那么前 n-1 次 tryRelease()方法必须返回 false；而只有同步状态完全释放了，才能返回 true。

c) 公平锁和非公平锁的区别

i. TryAcquire(int acquires)和 nonfairTryAcquire(int acquires)比较，唯一的不同就是判断条件多了 hasQueuedProcessors()方法，即加入了同步队列中当前节点是否有前驱结点的判断，如果方法返回 true 表明有线程比当前节点更早地请求了获取锁，因此需要等待前驱线程获取锁并释放锁后才能继续获取锁。

ii. 公平锁保证了锁的获取按照 FIFO 原则，而代价是进行大量的线程切换，而非公平锁虽然可能造成线程“饥饿”，但极少的线程切换，保证更大的吞吐量。

5. 读写锁

a) 维护了一对锁，通过分离读锁和写锁，使得并发性相对一般的排他锁有了很大的提升，更好的并发性和吞吐量。

b) 读写锁的接口和示例->ReadWriteLock 仅定义了读锁和写锁的两个方法，即 readLock 方法和 writeLock 方法，而其实现 ReentrantReadWriteLock 还有一些方法：

i. GetReadCount() 返回当前读锁获取的次数，

ii. getReadHoldCount() 返回当前线程获取读锁的次数，使用 ThreadLocal 保存当前线程获取次数

iii. IsWriteLocked() 判断写锁是否已被获取

iv. GetWriteHoldCount 返回当前写锁被获取的次数

c) 读写锁的实现分析

i. 读写状态的设计->读写锁的自定义同步器需要在同步状态上维护多个读线程和一个写线程的状态，如果在一个整型变量上维护多种状态，就一定需要”按位切割使用”这个变量，读写锁将变量分成两个部分，高 16 位标识读，低 16 位表示写。假设当前同步状态为 S，根据状态的划分能得出一个结论：S 不等于 0 时，当写状态(S&0X0000FFFF)不为零，那么读状态(S>>>16)大于 0，即读锁已被获取。

ii. 写锁的获取与释放->写锁是一个支持重进入的排他锁，如果当前线程已经获取了写锁，则增加写状态。**如果当前线程在获取写锁时，读锁已经被获取(读状态不为 0)或者该线程不是已经获取写锁的线程，则当前线程进入等待状态**。而写锁一

旦被获取，则其他读写线程的后续访问均被阻塞。**读写锁确保写操作对读操作可见，如果允许读操作在已被获取的情况下对写操作获取，那么正在运行的其他读线程就无法感知到当前写写线程的操作。**

iii. 读锁的获取和释放->支持重进入的共享锁，如果当前线程在获取读锁时，写锁已经被其他线程获取，则进入等待状态。每个线程各自获取读锁的次数只能选择保存在 ThreadLocal 中，由线程自身维护。。。

iv. 锁降级->写锁降级成为读锁。把持住(当前拥有的)写锁，再获取到读锁，随后释放()写锁的过程。如果当前线程不释放读锁，而是直接释放写锁，假设此刻另外一个线程 T 获取了写锁并修改了数据，那么当前线程无法感知 T 的数据更新。

6. LockSupport 工具
   a) 作为构建同步组件的基础工具
   b) 一组以 park 开头的方法来阻塞当前线程，以及 unpark(Thread thread)方法来唤醒一个被阻塞的线程
   c) Park(Object blocker)提供阻塞对象的信息，通过线程转储快照
7. Condition 接口
   a) Object 的监视器方法和 Condition 接口的对比
   b) Condition 是依赖 Lock 对象的，一般将 Condition 对象作为成员变量，当调用 await()方法后，当前线程会释放锁并在此等待，而其他线程调用 Condition 对象的 signal()方法，通知当前线程后，当前线程才能从 await()方法返回，并且在返回前已经获取了锁
   c) 有界队列的实现->当队列为空时，队列的获取操作将会阻塞获取线程，知道队列中有新增元素，当队列已满，队列的插入操作将会阻塞插入线程，知道队列出现空位。
   d) Condition 的实现分析
      i. 等待队列
         1. FIFO，如果一个线程调用了 Condition.await()方法，那么该线程就会释放锁，构造成节点加入等待队列并进入等待状态。
         2. 一个 condition 包含一个等待队列 condition 拥有首节点和尾节点，**当前线程调用 Condition.await()方法时，将会以当前线程构造节点，并将节点从尾部加入等待队列。**该过程并没有使用 CAS 保证，是因为调用 await 操作时必定已经获得了对象的锁，**因此这里是用所保证线程安全的。**
         3. Lock 拥有一个同步队列和多个等待队列。
      ii. 等待
         1. 当调用 await()方法时，相当于同步队列的首节点移动到 Condition 的等待队列中。
      iii. 通知
         1. 通过调用 Condition 的 signal()方法，将会唤醒等待在队列中等待时间最长的节点，在唤醒节点之前，会将节点移动到同步队列中。
         2. 通过调用同步器的 enq(Node node)方法，等待队列中的头节点线程安全的移动到同步队列中，完成后，当前线程再使用 LockSupport 唤醒该节点的线程。
         3. 被唤醒的线程，将从 await()方法中 while()循环中退出，进而调用同步器的 acquireQueued()方法加入获取同步状态的竞争中
         4. Condition 的 signalAll 方法，相当于对等待队列中的每个节点执行了一次 sinal()方法，过程同上。
8. 怎么检测一个线程是否拥有锁？

a) 我一直不知道我们竟然可以检测一个线程是否拥有锁，直到我参加了一次电话面试。在 java.lang.Thread 中有一个方法叫 holdsLock()，它返回 true 如果当且仅当当前线程拥有某个具体对象的锁。

9. 如果同步块内的线程抛出异常会发生什么？
   a) 这个问题坑了很多 Java 程序员，若你能想到锁是否释放这条线索来回答还有点希望答对。无论你的同步块是正常还是异常退出的，里面的线程都会释放锁，所以对比锁接口我更喜欢同步块，因为它不用我花费精力去释放锁，该功能可以在 finally block 里释放锁实现。

10. Java Concurrenct API 中的 Lock 接口(Lock interface)是什么？对比同步它有什么优势？
    a) 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。
    b) 它的优势有：
       i. 可以使锁更公平
       ii. 可以使线程在等待锁的时候响应中断
       iii. 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
       iv. 可以在不同的范围，以不同的顺序获取和释放锁

6. Java 并发容器和框架

1. ConcurrentHashMap 的实现原理和使用
   a) 为什么要实现 ConCurrrentHashMap
      i. 线程不安全的 HashMap

         在多线程的情况下，使用 HashMap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，所以在并发的情况下不能使用 HASHMAP，是因为多线程会导致 HashMap 的 Entry 链表形成**环形数据结构**，一旦形成环形数据结构，Entry 的 next 节点永远不为空，就会产生死循环获取 Entry。

      ii. 效率低下的 HashTable
         1. 通过使用 synchronized 来保证线程安全，竞争激烈的时候效率低下，
      iii. HashTable 和 HashMap 的区别

```java
public class Hashtable
    extends Dictionary
    implements Map, Cloneable, java.io.Serializable
```

```java
public class HashMap
    extends AbstractMap
    implements Map, Cloneable, Serializable
```

可见Hashtable 继承自 Dictiionary 而 HashMap继承自AbstractMap

         1. HashTable 的 put 方法如下：
            a) 方法是同步的
            b) 不允许 value==null
            c) 调用了 key 的 hashCode 方法，如果 key==null,则会抛出空指针异常

```java
public synchronized V put(K key, V value) {  //###### 注意这里1
  // Make sure the value is not null
  if (value == null) { //###### 注意这里 2
    throw new NullPointerException();
  }
  // Makes sure the key is not already in the hashtable.
  Entry tab[] = table;
  int hash = key.hashCode(); //###### 注意这里 3
  int index = (hash & 0x7FFFFFFF) % tab.length;
  for (Entry e = tab[index]; e != null; e = e.next) {
    if ((e.hash == hash) && e.key.equals(key)) {
      V old = e.value;
      e.value = value;
      return old;
    }
  }
  modCount++;
  if (count >= threshold) {
    // Rehash the table if the threshold is exceeded
    rehash();
    tab = table;
    index = (hash & 0x7FFFFFFF) % tab.length;
  }
  // Creates the new entry.
  Entry e = tab[index];
  tab[index] = new Entry(hash, key, value, e);
  count++;
  return null;
}
```

2. HashMap 的 put 方法如下
   a) 方法是非同步的
   b) 方法允许 key == null
   c) 方法并没有对 value 进行任何调用，所以允许为 null

```java
public V put(K key, V value) { //###### 注意这里 1
  if (key == null)  //###### 注意这里 2
    return putForNullKey(value);
  int hash = hash(key.hashCode());
  int i = indexFor(hash, table.length);
  for (Entry e = table[i]; e != null; e = e.next) {
    Object k;
    if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
      V oldValue = e.value;
      e.value = value;
      e.recordAccess(this);
      return oldValue;
    }
  }
  modCount++;
  addEntry(hash, key, value, i);  //###### 注意这里
  return null;
}
```

iv. ConcurrentHashMap 的锁分段技术能有效提升并发访问效率

1. 加入容器中有多把锁，每一把锁用于锁容器其中一部分的数据，那么当多线程访问容器中不同数据段的数据时，线程间就不会存在锁竞争，从而有效地提高并发访问效率，这就是 ConcurrentHashmap 所使用的锁分段技术。首先将数据分成一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个数据段时，其他段的数据也能被其他线程访问。

b) ConcurrentHashMap 的结构

i. ConcurrentHashMap->segment[]

Segment->HashEntry[]

segment 是一种**可重入锁**，是一个数组和链表的结构；HashEntry 是一个链表结构的元素，每个 segment 守护一个 HashEntry 数组里面的元素，每当 HashEntry 里面的数据要修改时，需要首先获得它对应的 segment.

c) ConcurrentHashMap 的初始化

i. 初始化 segment 数组，必须保证 segments 数组的长度是 2 的 N 次方，所以必须计算出一个大于或等于 concurrencyLevel 的最小 2 的 N 次方的值来作为 segment 数组的长度->主要是为了能够按位与和散列算法来定位 segment 数组的索引

ii. 初始化 segmentShift 和 segmentMask->segment 用于定位参与散列运算的位数，segmentShift 等于 32-sshift，所以=28，segmentmask 是散列运算的掩码，等于 ssize-1 即为 15

iii. 初始化每个 segment->输入参数为 initialCapacity 和 loadfactor，默认分别为 16 和 0.75

d) 定位 segment

i. 既然使用分段锁 segment 来保护不同段的数据，那么在插入和获取元素的时候，必须通过散列算法定位到 segment.在进行散列计算前必须再进行散列，以减少

散列冲突。

e) ConcurrentHashMap 的操作

i. Get 操作->先经过一次散列，然后再使用这个散列值通过散列运算定位到 segment，再通过散列运算定位到元素

```
Public V get(Object key){
    Int hash = hash(key.hashCode());
    Return segmentFor(hash).get(key,hash);
}
```

高效在于 get 方法不用加锁，将所有的共享变量都定义为 volatile 类型，根据 happen before 原则，对 volatile 字段的写入操作先于读操作

```
Hash >>> segmentShift) & segmentMask
Int index = hash & (tab.length-1)
```

i. Put 操作->由于 put 方法需要对共享变量进行写操作，为了线程安全，在操作共享变量时，必须加锁。Put 方法首先定位到 Segment，然后再 segment 里进行插入操作，插入操作需要经历两个步骤：

1. 是否要对 segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置。然后将其放在 HashEntry 数组里
2. 如何扩容，首先会创建一个容器是原来容量两倍的数组，然后将原数组里的元素进行散列后插入新数组中，只对某个 segment 扩容。

2. Java 中的阻塞队列

a) 什么是阻塞队列？

i. 支持阻塞的插入方法：意思是当队列满时，队列会阻塞插入元素的线程，直到队列不满

ii. 支持队列的移出方法：队列为空时，获取元素的线程会等待队列变成非空。

b) Java 里的阻塞队列

i. Jdk7 提供了 7 个阻塞队列，p168

ii. 阻塞队列的实现原理

1. 使用通知模式实现。所谓通知模式，就是当生产者往满的队列中添加元素时，会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。使用 ArrayBlockingQueue+Condition 来实现

3. Fork/Join 框架

a) 分割任务->需要一个类来将大任务分割成子任务，

b) 执行任务并合并结果，分割的子任务分别放在双端队列中，然后几个启动线程分别中双端队列中获取任务执行。子任务执行完的结果都统一放在一个队列中，启动一个线程从队列中拿数据，然后合并这些数据。

c) ForkJoinTask 与一般任务的区别在于它需要实现 compute 方法，在这个方法中，首先要判断任务是否足够小，如果足够小就执行任务，如果不够小，就必须分割成来成两个任务，每个子任务在调用 fork 方法时会再次进入 compute 方法，看看当前子任务是否需要继续分割成子任务，如果不需要就就执行子任务并返回结果，使用 join 方法等待子任务完成并得到他的结果。

7. Java 中的 13 个原子操作类

a) 原子更新基本类型
   i. AtomicBoolean 原子更新布尔类型
   ii. AtomicInteger 原子更新整型
      1. AddAndGet(int delta) 方法：以原子方式将输入的数值与实例(AtomicInteger 里的 value)中的数值相加，并返回结果。
      2. CompareAndSet(int expect, int update)：如果输入的数值等于预期值(AtomicInteger 里的 value)，则以原子方式将该值设置为输入的值
      3. GetAndIncreament()：以原子的方式+1 注意这里返回的是自增前的值
      4. LazySet(int newValue)：最终会设置成 newValue
      5. GetAndSet 以原子的方式设置 newValue 的值，并返回旧值
      6. 那么 getAndIncrement 是如何实现原子操作的呢？
         a) For 循环体中第一步先取得 AtomicInteger 里存储的数值，接下来对当前数值进行+1，关键步骤在于第三步调用 compareAndSet 方法进行原子更新操作，检查当前数值是否等于 current，等于意味着其他线程没有修改，则 AtomicInteger 的当前数值更新成 next，如果不等 compareAndSet 方法会返回 false，程序会进入循环重新开始进行 compareAndSet 操作。
   iii. AtomicLong 原子更新长整型
b) 原子更新数组
   i. AtomicIntegerArray：原子更新整型数组里的元素
   ii. AtomicLongArray：原子更新长整型数组里的元素
      1. AddAndSet(int i，int delta)以原子方式将输入值与数组中索引 i 的元素相加
      2. CompareAndSet。
      3. 数组以构造方法传递进去，然后 AtomicIntegerArray 会将当前数组复制一份，所以当前数组 AtomicIntegerArray 对内部数组元素进行修改时，不会影响传入数组。
   iii. AtomicReferenceArray：原子更新引用类型数组里的元素
c) 原子更新引用类型
   i. AtomicReference：原子更新引用类型
   ii. AtomicReferenceFieldUpdate：原子更新引用类型里的字段
   iii. AtomicMarkableReference：原子更新带标记位的引用类型，

d) 原子更新字段类
   i. AtomicIntegerFieldUpdater：原子更新字段的更新器
   ii. AtomicLongFieldUpdater：原子更新长整型字段的更新器
   iii. AtomicStampedReference：原子更新带有版本号的引用类型解决 ABA 问题

8. java 中并发工具类

a) 等待多线程完成的 CountDownLatch
   i. CountDownlatch 允许一个或多个线程等待其他线程完成操作
   ii. 主线程需要等待其他线程完成后操作，可以使用 Thread.join 方法；join 用于让

当前执行线程等待 join 线程执行结束，其实现原理就是不停地检查 join 线程是否存活；

```
While(isAlive()){
    Wait(0);
}
```

直到 join 线程终止后，线程的 this.notifyAll()方法会被调用，此方法在 JVM 中实现

iii. CountDownLatch 也可以实现 join 的功能，构造函数接受一个 int 类型的参数作为计数器，如果你想等待 N 个点完成，这里就传入 N,使用 CountDownLatch 的 countDown 方法，N 就会减 1，await 方法会阻塞当前线程，知道 N 变为 0；这个 N 可以是 N 个线程，也可以是一个线程的 N 个执行步骤，也可以使用 await 方法来定时。

b) 同步屏障 CyclicBarrier

i. 让一组线程到达一个屏障(也可以叫同步点)时被阻塞，知道最后一个线程到达屏障，屏障才会开门，所有屏障拦截的线程才会继续执行

ii. 默认构造函数是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，**每个线程通过 await 方法告诉 CyclicBarrier 该线程已经到达屏障，然后该线程被阻塞**

iii. CylicBarrier 的应用场景->**多线程计算数据，最后合并计算结果的场景**。

iv. CyclicBarrier 和 CountDownLatch 的区别

1. Cd 的计数器只能使用一次，**cb 的计数器可以使用 reset()方法重置**：例如计算错误，可以重置计数器，让线程重新执行一次。
2. Cb 还提供了很多有用的方法->**getNumberWaiting 方法可以获取 cb 阻塞的线程数量，isBoken 方法用来了解阻塞线程是否被中断。**

c) 控制并发线程数的 Semaphore

1. Semaphore(信号量)是**用来控制同时访问特定资源的线程数量**。
2. 用于做流量控制->公共资源有限，数据库的连接
3. 构造函数 Semaphore(int permits)接受一个整型的数字，表示可用的线程数**使用 acquire 方法获取一个许可证，使用完调用 release 方法归还许可证。**

d) 线程间交换数据的 Exchanger

i. 线程间进行协作的工具类，**提供一个同步点，在这个同步点，两个线程可以彼此交换数据，两个线程通过 exchange 方法进行交换数据，**

ii. 使用场景：可以用于遗传算法，选出两个人作为交配对象，交换两个人的数据，并使用交叉规则得出两个交配结果，**还可以进行两个人工作的校对工作。**

9. Java 中的线程池

a) 线程池的实现原理

i. 线程池判断核心线程里的线程是否都在执行任务，如果不是，则创建一个新的工作线程来执行任务，都在执行任务，进入下流程

ii. 线程池判断工作队列是否已满。如果工作队列没有满，则将新提交的任务存储在这个工作队列中，否则进下步

iii. 判断线程池的线程是否都处于工作状态，如果没有创建线程执行任务，满了，交给饱和策略。

iv. ThreadPoolExecute 执行 execute 分以下四种情况：

1. 如果当前运行的程序少于 corePoolSize，则创建新线程来执行任务(需获取

全局锁)

2. 如果运行的线程等于或多于 corePoolSize，则将任务加入 BlockingQueue。
3. 如果无法加入 BlockinQueue，创建新的线程来处理任务
4. 如果创建新线程使当前运行的线程超出 maximumPoolSize，任务将被拒绝，并调用 RejectedExecutionHandler.rejectedExecution()方法
5. 工作线程：线程池创建线程时，会将线程封装成工作线程 Worker，Worker 在执行完任务之后，还会循环去工作队列中取任务来执行；
6. 总结，在 execute()方法中执行任务，首先在 execute 方法中创建一个线程，会让这个线程执行当前任务，这个线程执行完上述任务后，会反复从 BlockingQueue，获取任务来执行

b) 线程池的使用

i. 通过 ThreadPoolExecutor 来创建一个线程池

1. New ThreadPoolExecutor(corePoolSize,maximumPoolSize,keepAliveTime,milliseconds,runnabelTaskQueue,handler)
2. corePoolSize(线程池的基本大小) :当提交一个任务到线程池时，不管是否有空闲线程在，都创建新线程，等到需要执行的任务数大于线程池的基本大小时就不再创建。如果调用了线程池的 prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。
3. RunnableTaskQueue(任务队列):用于保存等待执行的任务的阻塞队列，可以选择以下几个阻塞队列：
   a) P204
4. maximumPoolSize(线程池最大数量)：使用无界的任务队列这个参数就没什么效果
5. ThreadFactory：用于设置创建线程的工厂，给线程设置更有意义的名字
6. RejectedExecutionHandler(饱和策略)：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采用一种策略处理新任务，四种策略
   a) AbortPolicy:直接抛出异常
   b) CallerRunsPolicy:只用调用者所在的线程来运行任务
   c) DiscardOlddestPolicy:丢弃队列里最近的一个任务，并执行当前任务
   d) DiscardPolicy:不处理
   e) 实现接口自定义，记录日志
   f) KeepAliveTime(线程活动保持时间)：线程的工作线程空闲后，保持存活的时间，如果任务很多，并且每个任务的执行时间比较短，可以调大时间，保持线程的利用率
   g) TimeUnit(线程活动保持时间的单位)；

c) 向线程池提交任务

i. Execute 方法用于提交**不需要返回值**的任务无法判断线程是否执行成功；
ii. 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，通过 future 的 get()方法取得返回值，get()方法会阻塞当前线程知道任务的完成，而是用 get(long timeout,TimeUnit unit)则会阻塞当前线程一段时间后立即返回，任务可能还没执行完成。

d) 关闭线程池->遍历线程池中的工作线程，然后逐个调用线程的 interrupt 方法来中断

线程

i. Shutdown->只是将线程池的状态设置成 SHUTDOWN 状态，然后中断所有没有在执行的任务的线程
ii. ShutdownNow->将线程池的状态设置成 STOP，然后尝试停止所有正在执行或暂停任务的线程。

e) 合理配置线程池
   i. 首先分析任务的特性：其中包括以下几个角度
      1. Cpu 密集任务，IO 密集任务，混合任务
      2. 任务优先级：高中低
      3. 任务执行时间：长中短
      4. 任务依赖性：是否依赖其他资源
   ii. Cpu 密集型的应该配置尽可能小的线程，Ncpu+1 个线程的线程池。IO 密集型的并不是一直在执行，所以尽可能配置多的线程，2*Ncpu。优先级不同的采用优先队列 PriorityBlockingQueue 来处理。
   iii. 依赖数据库连接池的任务，因为线程提交 SQL 后需要等待数据库返回结果，等待的时间越长，则 Cpu 空闲时间就越长，那么线程数应该设置得越大，更好地利用 CPU。
   iv. 建议使用有界队列：
f) 线程池的监控
   i. 方便在出现问题的时候，可以根据线程池的使用状况快速地定位问题，可以通过参数进行监控，属性如下：
      1. taskCount：线程需要执行的任务数量
      2. completedTaskCount：线程在运行过程中已完成的任务数量
      3. largestPoolSize：线程池里曾经创建过的最大线程数
      4. getPoolSize：线程池的线程数量
      5. getActiveCount:获取活动的线程数
   ii. 自定义线程池继承线程池重写代码-实现监控

10. 什么是 Executors 框架？

a) Executor 框架同 java.util.concurrent.Executor 接口在 Java 5 中被引入。**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。**

b) 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