多线程并发

https://github.com/LLN825

0、多进程与多线程出现的原因是什么?

如果计算机自始至终只执行单个程序，并且它能访问计算机中的所有资源显然是一种浪费。操作系统管理cpu的方式就是让多个进程同时执行，主要有以下原因：

- 资源利用率：当某进程执行IO-外部操作时，CPU应立即切换到别的进程，而不是空转着等待IO完毕。
- 公平性：不同用户和程序对计算机上资源有同等的使用权，应让它们能轮流使用接近共享，而不是A独占后B再独占。
- 便利性：有些业务都涉及多个任务，完成业务需要这些任务共同推进，并在必要时相互通信。

1、多线程有哪些优势、又有哪些问题呢?

- 优势：多线程可以充分利用多核处理器的强大性能、多工作流的建模方案、创造更灵敏的用户体验更好的软件。
- 问题：1、安全性：放任多线程随意执行，很可能会出现运算结果的错误。
  2、活跃性：可能带来无限等待，如死锁、饥饿等问题。
  3、性能：只要是切换就会带来开销，丢失程序的局部性，使高速缓存的数据失效。

2、什么是线程安全性？如果不使用同步策略，什么对象会/不会出现线程安全问题?

线程安全性指的是无论运行时环境采用何种调度方式，这些线程将如何交替执行，执行的操作是什么，最终这个对象的行为都符合其规范或后验条件中对状态的约束，即运行的结果是正确的，是和串行执行的结果相同的。

有可变状态的实例对象和静态对象(共享数据)会有线程安全问题，无状态或无可变状态的对象、线程栈中未发布的局部变量没有线程安全问题。

3、什么是原子性？非原子操作有哪些问题？如何避免?

它与事务应用程序的原子性具有相同的含义：一组语句作为一个不可分割的单元被执行。

常见的复合操作类似于读取 - 修改 - 写入、先检查后执行、延迟的初始化、分别修改两个相互关联的变量，单线程执行起来完全没有问题。但是多线成并发时，会轮流使用cpu，指令可能交错执行，就会导致结果混乱，即陷入一种**"计算的正确性取决于多个线程的交替执行顺序"**的境地。因为在观察到采取行动之间的这段时间内，其他线程的操作可能导致我们之前观察的结果失效。

将复合操作原子化常见的方法：

1、使用JUC包下的原子变量类，能够保证访问操作都是原子的，如AtomicLong、AtomicReference

2、java的内置锁机制，即synchronized修饰的同步代码是以原子方式执行的。

4、什么是可见性？不可见是如何发生的？如何避免?

可见性是指：当某个线程修改了对象的状态后，其他线程能看到该变化。

java内存模型允许编译器通过调整指令顺序，在不改变程序语义的前提下，尽可能减少寄存器的读取、存储次数，从而反复使用寄存器中的值而不是从内存中获取新鲜的值，这就会破坏有序性，并导致不可见性问题。

保证可见性的措施：

1、使用volatile关键字避免指令重排（JIT编译优化）不仅能保证有序性，而且强迫每次读写直接和主存打交道，而不是利用自己工作内存的高速缓存保证了可见性。

2、JMM通过happens-before法则保证顺序执行语义，满足happens-before原则的程序就能保证可见性。

3、使用synchronized关键字，在加锁的时候从主存中读取最新数据，释放的时候将修改写回主存。

4、加System.out.println("XXX")也能保证可见性，因为该函数底层有synchonized，强制当前线程从主存获取值。

5、什么是发布，什么是逸出？有什么样的注意事项？

发布是指使对象能在当前作用域之外的代码中使用，当本不该发布的对象被发布了就称为逸出。

注意事项：

- 把对象传递给外部方法或公开的静态变量时就相当于发布了这个对象，发布后**我们需要认为**任何类和线程都能访问它。
- 发布一个对象，在该对象的**非私有域**中引用的所有对象和所有方法所能触及的对象同样会被发布。
- 在构造方法中隐含地使this引用逸出会**提前发布一个不成熟的对象**。我们不能指望一个尚未成熟的对象拥有线程安全性。

6、实现线程安全性最简单的方式是什么？有哪些实现方式？

线程封闭是实现线程安全性的最简单方式。因为把对象封闭在一个线程中必然会实现线程安全性，即使被封闭的对象本身并没有安全机制

实现方式：

- **线程栈封闭** java语言保证了基本类型的局部变量始终封闭在栈中，引用类型的局部变量，只要确保该引用不被发布即可。
- **ThreadLocal**每个线程都能凭自己的身份证“threadlocal map ref”，获取线程私有的柜子ThreadLocalMap，在属于自己的柜子中 get(),set(),remove()，这个柜子的key是ThreadLocal对象，value是要在本线程中封闭的值，不会影响到其他线程中的对象。

ThreadLocal确保同一线程之间参数传递的方便。也常用于将从连接池获取到的连接分配给某个线程（放到它的Map中）。

7、String为什么是线程安全的？内涵？StringBuffer和StringBuilder呢？

String的底层是一个final修饰的byte数组，由于数组一旦声明其长度就不变，且final修饰的变量一旦初始化后也保持不变。因此String的状态一旦声明即是不可变的，不可变对象一定是线程安全的。

不可变的内涵是对象自身的状态都是不可变的，而不是对象引用不可变。如：String对象是不可变的，但是引用是可变的，因此引用是可以修改的，即通过将一个保存新状态的实例“替换”原有的不可变对象，这也可以看作是一种弱原子性。

StringBuffer和StringBuilder底层的byte数组并没用final修饰，因此其引用的指向(sb对象的状态)可以改变。而StringBuffer中的方法有synchronized关键字修饰，即它使用了自身的内置锁来保证自身是线程安全的。StringBuilder线程不安全，单线程下性能会好一点。

8、面向对象在线程安全方面有什么优势？有哪些实现方式？分层技术有什么优势？

面向对象这种技术不仅有助于编写出结构优雅，可维护性高的类，还有助于编写出线程安全的类。程序状态的封装性越好，就越容易实现程序的线程安全性。

实现方式：

- **装饰器模式：**通过将非线程安全的对象封装在一个装饰器对象中，装饰器中所有方法都是同步的。包装之后，对底层对象的所有访问都必须通过装饰器来进行，安全性由表层的装饰器方法实现。
- **线程安全性的委托：**外部类的安全性是把自身的状态委托给了底层的线程安全类。例如String的线程安全类委托给了底层的byte数组，我们也可以自定义一个含有某些高级方法的MyMap，安全性委托给底层的CurrentHashMap。
- **克隆对象**：将共享数据克隆到某个线程栈中，该线程在栈中可以随意地对副本进行读写，常常用在并发环境下的迭代器迭代操作中，安全性由克隆（已经不是同一个对象了）实现。

面向对象后更方便将程序分层，层内高耦合完成复杂业务，层间低耦合清晰地传递信息，结构非常优雅更方便扩展

单一的应用程序虽然避免了在不同层次之间传递任务时存在的网络时延和将计算过程分解到不同的抽象层次带来的开销，但是这种系统到达能力极限后，想要提升是非常困难的。三层模型中的表现层、业务逻辑层、持久化层是彼此独立的，可以由不同的系统来处理，使得不同层次中并发线程数量可以不同，这提高了可伸缩性更利于充分利用计算资源。

9、同步容器与并发容器的辨析？

同步容器：包括Vector和HashTable，这类实现线程安全的方式是：将状态封装起来，并对每个公有方法都进行同步，使得每次只有一个线程能访问容器的状态。

并发容器：常见的有ConcurrentHashMap、CopyOnWriteArrayList/Set、BlockingQueue、Dequeue它们是专门针对多线程并发设计的，使用它们通常可以支持任意数量的并发读线程，从而并发访问下实现更高的吞吐量。或者有助于实现某些用于多线的设计模式，从而使多线程更合理地工作。

10、什么是同步工具类？有哪些同步工具类？

顾名思义，同步工具是用于显式地完成线程间同步的工具，利用这些工具的特性我们可以自构建各式各样的多线程应用。同步工具封装了一些状态也提供了一些对自身状态进行操作的方法。这些状态将决定执行同步工具的线程是继续执行还是等待。

- **闭锁CountDownLatch：**在闭锁满足条件到达结束状态之前，将所有线程都阻塞在门口，当它到达结束状态时会让所有的线程通过。闭锁的状态包括一个计数器，它的值表示需要等待的事件数量，countDown()递减计数器表示一个事件发生了，await()会一直阻塞当前进程直至计数器达到零即所有需要等待的事都已经发生。可以用于确保某个服务在其依赖的所有服务都已经启动之后才启动；当所有玩家都准备就绪后游戏才正式开始。
- **FutureTask：**用Callable实现的，相当于一种可生成结果的Runnable，可以处于：等待运行、正在运行、运行完成三种个状态。直接继承Thread或者实现Runnable接口都可以创建线程，但是这两种方法的最大局限是：不能返回一个值或者抛出一个受检异常。Callable是一种更好的抽象，它可以做到这两件事。

即在起初主线程和子线程一同开启。等到主线程需要子线程的结果的时候再去获取子线程的结果。此时子线程的任务如果已经完成，get会立即返回；如果没有完成，get会阻塞并直到任务完成；如果抛出了异常，那么get会将该异常封装为ExecutionException并重新抛出；如果被取消，那么个体会抛出CancellationExeception异常。

- **信号量Semaphore**：与操作系统中的功能类似。计数信号量用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。Semaphore是一组虚拟的许可，执行操作前得获取许可，执行操作后释放许可，如果没有许可那就保持阻塞直到有许可。

11、任务执行接口Executor是干嘛的？它与传统方式相比有什么优势？

并发应用程序都是围绕任务执行构建的，如Web，Mysql中的一次客户请求就是一个任务（多么清晰的任务边界）。独立的任务也更有助于实现并发。而Executor接口就为**独立任务并发执行**制定了一套规范，它基于生产者-消费者模式，通过底层灵活的任务队列和线程池实现。主线程将任务提交到的任务调度队列中，线程池中的线程从中取出任务并加以执行。

传统方式为每一个请求分配一个线程，这种方法总是在不停地新建线程、销毁线程、且没有限制可创建线程的数量

而线程池是管理一组同构工作线程的资源池,我们将为每个任务分配一个线程策略变成了基于线程池的策略后

1、当任务来临时，工作线程通常已经存在，通过重用线程池中现有的线程而不是创建新线程，可以避免为每个任务创建线程和销毁线程而产生的巨大开销，提高了**响应性**。

2、适当调整线程池的大小，可以创建足够多的线程让CPU保持忙碌状态，同时还能够防止线程过多相互竞争资源使性能急剧下降进而保证系统的**稳定性**。

3、Executor实现了任务提交和任务执行之间的解耦合，并且可以实现管理、监视、记录日志等功能。其子类ExecutorService还增加了对线程生命周期的管理即用shutdown方法关闭，更贴近真实的开发场景具有**灵活性**。

12、由于人的意愿或者时间限制我们往往不得不将任务取消，任务取消需要定义什么？都有什么将任务取消的策略呢？

一个可取消的任务要定义，其他代码如何请求取消该任务，任务在何时检查是否已经请求取消，响应取消请求时应该执行什么操作。

取消策略：

- **线程中断**：java并没有提供任何抢占式的机制来取消操作或者终止线程。相反，它提供了一种协作式的中断机制来实现取消操作。线程的中断方法interrupt()的含义并不是立即停止目标线程正在进行的工作，而只是传递了请求中断的消息。然后由目标线程在下一个合适的时刻选择中断自己。
- **停止基于线程的服务**：如在含有多个线程的ExecutorService任务执行服务例程中，其底层的线程池是其工作线程的所有者，即任务的执行者。**关闭ExecutorService**以及它所拥有的线程，即可取消执行的任务。它提供了两个生命周期方法，它们的差别在于**安全性和响应性**。
- shutdown():正常关闭，会等到队列中所有任务都执行完毕后才关闭，缓慢但安全。
- shutdownNow():强行关闭，放弃任务队列中的任务，速度快但风险大。
- **异常终止**：前两者是主动取消某个任务，这一种我们被动看到的，任务由于自身执行过程中的错误或异常而失败了。

13、线程的异常终止服务会就此结束吗？如何处理这种异常终止？

基于单线程的服务会因为异常而终止，而拥有多个线程的服务可能不会(**也可能会！**)由于某个线程的异常而导致全盘崩溃。这却决于崩溃的是什么类型的线程，某个负责记录日志的线程崩溃并不会导致整体服务显著地失败，而若是负责任务派发的线程崩溃了，整体服务就会寸步难行。我们可以通过主动被动捕获异常的方式解决线程泄露的方式

主动捕获：工作线程能够胜任我们交给它的任务，我们有权利在主线程中保持怀疑，即在主线程中主动地捕获工作线程中的异常。

被动捕获：所有线程的未主动捕获的异常指定同一个异常处理器*UncaughExceptionHandler*，当线程由于为捕获异常而退出时，JVM会把这个事件报告给异常处理器。

线程泄露：因为未合理捕捉某个线程的异常终止，导致总服务戛然而止，然而其他未异常的线程并未关闭且仍占用着内存、cpu等资源。

服务的行为也取决于我们如何处理： 是捕获之后继续执行？还是就此结束？这一个问题。*“To be or not to be ？it's a question。”*

最常见的至少要将错误以及栈追踪信息写入应用程序日志，也有更直接的尝试重启一个工作线程继续执行，或者直接关闭应用程序服务。

14、如何创建一个基于线程池的服务?

Executor有四种固定的线程池，但！存在着致命的缺陷。

- **FixedThreadPool** ： 该方法返回一个固定线程数量的线程池。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**： 方法返回一个只有一个线程的线程池。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。

FixedThreadPool和newSingleThreadExecutor默认的任务队列可以无限制地增加到 Integer.MAX_VALUE ，大量堆积从而导致 OOM

- **CachedThreadPool**： 该方法返回一个可根据实际情况调整线程数量的线程池。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**：固定长度线程池，以时延或定时的方式执行任务，它的任务调度队列是DelayQueue，其中的每个Delayed元素都是一个延迟任务，都有一个响应的延迟时间，只有当某个元素逾期之后，才能将任务take出来放在一个线程上执行，不同的任务放在不同的线程上执行。

CachedThreadPool 和 ScheduledThreadPool 默认允许创建的线程数量为 MAX_VALUE ，可能会创建大量线程，从而导致 OOM。

默认的执行策略不能满足特定任务的需求，那么可以通过ThreadPoolExecutor构造方法来自己定制

ThreadPoolExecutor（core，max，kat，queue，）

corePoolSize：线程池的基本大小，即在没有任务执行时线程**池子**的大小，只有在工作队列满了的情况下才会创建超出这个数量的线程

MaximumPoolSize：线程池的最大大小表示可同时活动的线程数量的上限。

keepAliveTime：如果某个线程的空闲时间超过了存活时间，那么将被设为可回收的，并且在当前大小超过基本大小时终止自身

线程池的基本大小、最大大小以及存活时间共同负责线程的创建与销毁，进而控制资源的分配与回收。

workQueue：工作队列是一个用来保存等待执行的任务的阻塞队列，有三种基本类型：无界队列、有界队列、同步移交。

handler：饱和策略会在队列被填满后发挥作用。有 终止策略、抛弃策略、调用者运行策略。

threadFactory：线程池创建的线程都是通过线程工厂创建的，可以通过自定义的线程工厂定制线程的行为、权限、甚至是名字等。

ThreadPoolExecutor是可继承的，它提供了几个可以在子类中重写的方法。

beforeExecute,afterExecute、terminated 这些方法可以添加日志、计时、监视、释放资源或统计信息收集等功能。

15、多线程导致的活跃性问题及其应对措施?

线程饥饿死锁：在单线程的Execcutor中，一个任务将另一个任务提交到同一个Executor并等待这个被提交任务的结果会引发死锁。

多线程，当一个正在执行的任务一直因为等待其他仍处于工作队列的任务而阻塞，除非线程池足够大，否则会发生线程饥饿死锁。

锁顺序死锁：多线程在获取多个锁时没有采用一致的顺序而导致的死锁

两阶段策略：找出可能获得多个锁的位置 ，这个集合应该尽量小；全局分析，确保它们在整个程序中获取锁的顺序都保持一致

线程转储：JVM提供的帮助我们分析线程(死锁)的一种机制，包括了各个运行中的线程的站追踪信息和加锁信息。当诊断死锁时，它可以告诉我们那些锁导致了这个问题，涉及哪些线程，它们持有那些其他的锁等等

支持定时的锁：使用Lock类中的定时tryLock功能代替内置锁机制。在内置锁机制中如果没有获得锁就会永远等待下去，而显示锁可以指定一个时限，超过该事件后就返回一个失败信息。

饥饿：线程长时间无法使用到它所需要的资源，无法继续执行。引发饥饿的最常见资源是CPU时钟周期

线程优先级的使用：CPU密集型的后台线程会与前台线程共同竞争CPU的时钟周期，此时应发挥优先级的作用，将后台线程优先级降低，从而提高前台程序的响应。

活锁：该问题尽管不会阻塞线程，但也无法继续执行下去。因为线程不断重复执行相同的操作，而且总是失败。这种形式是由于错误恢复机制产生的，它总将不可修复的错误作为可修复的错误恢复，执着且刻板。

要想解决，须在重试机制中引入随机性。等待随机的时间和回退可以有效地避免活锁的发生！

丢失的信号：线程必须等待一个为真的条件，但是在开始等待时并没有检查条件谓词，即条件谓词早已为真，通知也已经发过了，但线程一意孤行地进入了等待状态，等待一个已经发生过的事件，因此会一直等待下去。

所以一定要在等待和通知之前检测条件谓词！

16、多线程可能会导致的性能问题有什么?

提升性能意味着用有限的资源做更多的事情。引入多线程的目标本是为了提升性能，然而与单线程相比，多线程总会引入一些额外的开销。如果过度使用，这些开销甚至会超过充分利用了计算能力所带来的性能提升。因此对于为了提升性能而引入的多线程来说，**并发带来的性能提升必须要超过并发导致的开销！**

阻塞

JVM在实现阻塞行为时，可以采用自旋等待不断地尝试获取锁直到成功，适合于等待时间较短的情况。也可以通过操作系统挂起被阻塞的线程，适合于等待时间较长的情况。如果简单地将线程挂起，就一定会去执行额外的两次上下文切换。

上下文切换

如果可运行的线程数量大于CPU 的数量，就会导致上下文切换，在这个过程中将保存当前运行线程的执行上下文，并且将新调度进来的线程的执行上下文设置为当前上下文。上下文切换需要操作系统的介入，并且将导致缓存的缺失。线程越多、阻塞情况越多，就会发生越多的上下文切换,从而增加调度开销降低吞吐量。

内存同步

synchronizized和volatile通过一些特殊指令保证可见性，即Memory Barrier，内存栅栏可以使缓存无效。它们还抑制了一些编译器优化操作，例如操作的重排序。因此同样会对性能带来间接影响

线程的创建与销毁

线程的创建和销毁并不是没有开销的，如果需要执行的任务本身非常简短，那么为每个任务分配一根线程可能会产生创建线程和销毁线程的时间甚至超过了任务执行的时间。此时倒不如使用单线程。

17、Amdhal定理的思想是什么？我们如何主动对锁进行优化性能?

Amdahl定律告诉我们，当计算资源增多时，系统的加速比严重受限于必须被串行执行的代码比例。因为java程序中必须串行执行的操作的主要来源是独占方式的资源锁，因此我们要尽量降低锁的竞争程度。

1、减少锁的持有时间

将一些与锁无关的代码移出同步代码块，尤其是非常耗时的操作如IO、大数据计算等。我们只在必须持有锁的时候才持有锁，极可能地减少在持有锁使需要执行的指令数量。

2、降低锁的粒度

对于一个锁需要保护多个相互独立的状态变量时，可以将这个一锁分解为多个独立的锁，每个锁保护一个变量这让很多变量可以并发修改，将对一个锁的请求分摊到多个锁上也降低了每个锁被请求的频率。在锁分解的基础上进一步扩展为对容器（一组独立对象）上的锁进行分解称为锁分段技术，类似。

3、放弃使用独占锁使用无锁机制

原子变量类

原子变量比锁的粒度更细更，量级更轻。更新原子变量的路径速度比获取锁的路径更快，因为它不会挂起或者调度线程。因此使用原子变量而非锁的算法中，线程在执行时更不易出现延迟。它又比volatile 更强一些因为它同时实现了可见性和原子性，因为它内部的值是volatile声明的，它其中的方法都直接利用了硬件对并发的支持即CAS。（CAS+volatile实现无锁并发）

无锁队列算法

18、内置锁与显式锁的区别，即lock和synchronized相比有何不同，二者如何选取?

简单的来说synchronized会在进入代码块时自动获取一个内置锁，在退出代码块时自动释放它，这简化了编码工作。而且在代码执行出现异常时，JVM会自动释放锁。在大多数情况下，内置锁都能很好地完成工作，但是功能上存在一些局限性。

Lock接口是API层面的锁，它提供了一系列**可轮询、可定时的、可中断、非块结构**的锁获取操作，但在进入和离开保护的代码块时没有了自动的操作。因此必须手动进行锁的获取，并且需要我们显式地捕获异常，锁的释放必须要在finally块中显式进行。

在java6以前，内置锁在多线程环境中的性能远远落后于显式锁。然而自java6以来，因为它是内置属性，因此在实现时JVM做出了一系列的优化包括如锁消除、锁粗化、轻量级锁+膨胀、偏向锁、自旋等，因此其性能已经不再是短板了，良好的可伸缩性使之在竞争激烈的情况下也可与ReenrantLock相媲美。并且它结构紧凑，用起来更加安全方便也更被人们熟知。因此除了在一些内置锁无法满足的情况下，如需要更灵活的高级功能：定时、轮询、可中断、非块结构的锁时才使用显式锁，否则还是应该优先使用内置锁。

19、AQS?

ReentrantLock、CountDownLatch、Semaphore、FutureTask这两些同步工具或者接口之间存在许多共同点，它们都可以作为一个阀门，每当线程到达阀门时，可以通过、可以等待、可以取消，而且它们还可以支持中断相关、限时获取操作、公平或者非公平的操作。事实上它们底层实现都使用了一个共同的基类**AbstractQueuedSynchronizer**。

如果一个类想成为状态依赖的类即同步器，它必须拥有一些状态。**AQS负责管理同步器类中的状态**，它管理一个整数状态信息State，可以通过getState、setState、conpareAndSetState进行操作，这些方法并没有直接对外提供，而是作为基本操作封装在tryAcquire,tryRealeasedShared等方法内对外提供。ReentrantLock用它来表示所有者线程重复获得锁的次数、Semaphore用它来表示剩余的许可数量，FutureTask用它来表示任务的状态，CountDownLatch用它来表示等待发生的事件个数。

AQS 在内部维护了一个 叫做CLH的双向队列，队列中的元素都是在这个同步器上等待着的线程。当线程在竞争锁失败之后，线程和线程的有关信息会封装成 Node 加入到 AQS 队列中；获取锁的线程释放锁之后，会从队列中唤醒一个阻塞的 Node。

总之，AQS作为一个非常流行的同步器内核，它提供的条件队列和同步状态变量的存在让实现一个同步器变得简单高效。

20、CAS?

java乐观锁与悲观锁

乐观锁的思想：最乐观的思想，认为即使不加锁，也未必会出现冲突错误。这种方法基于冲突检查机制来判断在更新过程中是否存在其他线程的干扰，如果存在则失败(重试)，不存在则更新成功

悲观锁的思想：最悲观的估计，认为只要不加锁，就一定会发生冲突错误，因此要时刻提防着其他线程来修改共享变量，只有等到释放了，其他线程才有机会访问。

硬件对并发的支持

synchronized独占锁是悲观技术的代表，对于细粒度的操作还有一种乐观的方法即CAS。现在几乎所有的处理器都提供了这种特殊指令，包含了某种形式的原子读-改-写，这比基于代码/软件的同步机制粒度细的多得多，速度快的多得多。操作系统和JVM使用这些指令可以进一步实现同步器或互斥体，通过这些又可以进一步实现更复杂的并发对象。这就是软硬件协同设计的强大之处。

CompareAndSwap

CAS原子指令包含3个操作数（V，A，B），V是需要读写的内存位置，A是进行比较的值，B是是拟写入的新值。当且仅当V的值是A时，CAS才会用B来更新V的值。多个线程尝试使用CAS更新同一个变量时，只有其中一个线程能成功，其他的线程并不会挂起而是被告知在这次竞争中失败，并可以快速地再次尝试。

21Happen-Before原则是什么？具体有哪些原则

是对于成员变量与静态变量（共享数据）读写的可见性规则，并且具有传递性。

线程解锁m之前对变量的写，对于该变量对接下来对m加锁的其他线程的读是可见的

线程对于volatile变量的写，对于该变量对于其他线程的读也都是可见的。

线程start前对变量的写，对于该变量在线程运行之后的读也是可见的。

线程结束前对变量的写，对于该变量对于其他线程得知该线程结束后的读是可见的。

线程中断前对变量的写，对于该变量对于其他线程得知该线程中断后的读是可见的。

对变量默认值（0，false，null）的写，对于其他线程的读可见