架构师干货 www.it520.xyz

1. 什么是线程?

1、线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位，可以使用多线程对进行运算提速。

> 比如，如果一个线程完成一个任务要100毫秒，那么用十个线程完成改任务只需10毫秒

2. 什么是线程安全和线程不安全?

1、线程安全

线程安全: 就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问，直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

Vector 是用同步方法来实现线程安全的, 而和它相似的ArrayList不是线程安全的。

2、线程不安全

线程不安全：就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

线程安全问题都是由全局变量及静态变量引起的。

若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

3. 什么是自旋锁?

自旋锁是SMP架构中的一种low-level的同步机制。

1、当线程A想要获取一把自旋锁而该锁又被其它线程锁持有时，线程A会在一个循环中自旋以检测锁是不是已经可用了。

2、自选锁需要注意：

- 由于自旋时不释放CPU，因而持有自旋锁的线程应该尽快释放自旋锁，否则等待该自旋锁的线程会一直在那里自旋，这就会浪费CPU时间。
- 持有自旋锁的线程在sleep之前应该释放自旋锁以便其它线程可以获得自旋锁。

3、目前的JVM实现自旋会消耗CPU，如果长时间不调用doNotify方法，doWait方法会一直自旋，CPU会消耗太大

4、自旋锁比较适用于锁使用者保持锁时间比较短的情况，这种情况自旋锁的效率比较高。

5、自旋锁是一种对多处理器相当有效的机制，而在单处理器非抢占式的系统中基本上没有作用。

4. 什么是CAS?

1、CAS（compare and swap）的缩写，中文翻译成比较并交换。

2、CAS 不通过JVM,直接利用java本地方 JNI（Java Native Interface为JAVA本地调用）,直接调用CPU 的cmpxchg（是汇编指令）指令。

3、利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法,实现原子操作。其它原子操作都是利用类似的特性完成的。

4、整个java.util.concurrent都是建立在CAS之上的，因此对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。

5、CAS是项乐观锁技术，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。

> 1、使用CAS在线程冲突严重时，会大幅降低程序性能；CAS只适合于线程冲突较少的情况使用。
>
> 2、synchronized在jdk1.6之后，已经改进优化。synchronized的底层实现主要依靠Lock-Free的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

5. 什么是乐观锁和悲观锁?

1、悲观锁

Java在JDK1.5之前都是靠synchronized关键字保证同步的，这种通过使用一致的锁定协议来协调对共享状态的访问，可以确保无论哪个线程持有共享变量的锁，都采用独占的方式来访问这些变量。独占锁其实就是一种悲观锁，所以可以说synchronized是悲观锁。

2、乐观锁

乐观锁（ Optimistic Locking）其实是一种思想。相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。

> memcached使用了cas乐观锁技术保证数据一致性。

6. 什么是AQS?

1、AbstractQueuedSynchronizer简称AQS，是一个用于构建锁和同步容器的框架。事实上concurrent包内许多类都是基于AQS构建，例如ReentrantLock，Semaphore，CountDownLatch，ReentrantReadWriteLock，FutureTask等。AQS解决了在实现同步容器时设计的大量细节问题。

2、AQS使用一个FIFO的队列表示排队等待锁的线程，队列头节点称作“哨兵节点”或者“哑节点”，它不与任何线程关联。其他的节点与等待线程关联，每个节点维护一个等待状态waitStatus。

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

1、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。

2、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。

3、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。

8. 什么是Executors框架?

Java通过Executors提供四种线程池，分别为：

1、**newCachedThreadPool**创建一个**可缓存线程池**，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

2、**newFixedThreadPool** 创建一个**定长线程池**，可控制线程最大并发数，超出的线程会在队列中等待。

3、**newScheduledThreadPool** 创建一个**定长线程池**，支持定时及周期性任务执行。

4、**newSingleThreadExecutor** 创建一个**单线程化的线程池**，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

9. 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型?

1、JDK7提供了7个阻塞队列。（也属于并发容器）

i. ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列。

ii. LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列。

iii. PriorityBlockingQueue ： 一个支持优先级排序的无界阻塞队列。

iv. DelayQueue：一个使用优先级队列实现的无界阻塞队列。

v. SynchronousQueue：一个不存储元素的阻塞队列。

vi. LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

vii. LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

2、概念：阻塞队列是一个在队列基础上又支持了两个附加操作的队列。

3、2个附加操作：

支持阻塞的插入方法：队列满时，队列会阻塞插入元素的线程，直到队列不满。

支持阻塞的移除方法：队列空时，获取元素的线程会等待队列变为非空。

10. 什么是Callable和Future?

1、Callable 和 Future 是比较有趣的一对组合。当我们需要获取线程的执行结果时，就需要用到它们。Callable用于产生结果，Future用于获取结果。

2、Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，必须等待它返回的结果。java.util.concurrent.Future对象解决了这个问题。

3、在线程池提交Callable任务后返回了一个Future对象，使用它可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法，等待Callable结束并获取它的执行结果。

11. 什么是FutureTask?

1、FutureTask可用于异步获取执行结果或取消执行任务的场景。通过传入Runnable或者Callable的任务给FutureTask，直接调用其run方法或者放入线程池执行，之后可以在外部通过FutureTask的get方法异步获取执行结果，因此，FutureTask非常适合用于耗时的计算，主线程可以在完成自己的任务后，再去获取结果。另外，FutureTask还可以确保即使调用了多次run方法，它都只会执行一次Runnable或者Callable任务，或者通过cancel取消FutureTask的执行等。

2、futuretask可用于执行多任务、以及避免高并发情况下多次创建数据机锁的出现。

12. 什么是同步容器和并发容器的实现?

1、同步容器

1、主要代表有Vector和Hashtable，以及Collections.synchronizedXxx等。

2、锁的粒度为当前对象整体。

3、迭代器是及时失败的，即在迭代的过程中发现被修改，就会抛出ConcurrentModificationException。

2、并发容器

1、主要代表有ConcurrentHashMap、CopyOnWriteArrayList、ConcurrentSkipListMap、ConcurrentSkipListSet。

2、锁的粒度是分散的、细粒度的，即读和写是使用不同的锁。

4、迭代器具有弱一致性，即可以容忍并发修改，不会抛出ConcurrentModificationException。

> ConcurrentHashMap
> 采用分段锁技术，同步容器中，是一个容器一个锁，但在ConcurrentHashMap中，会将hash表的数组部分分成若干段，每段维护一个锁，以达到高效的并发访问；

13. 什么是多线程的上下文切换?

1、多线程：是指从软件或者硬件上实现多个线程的并发技术。

2、多线程的好处：

i. 使用多线程可以把程序中占据时间长的任务放到后台去处理，如图片、视屏的下载

ii. 发挥多核处理器的优势，并发执行让系统运行的更快、更流畅，用户体验更好

3、多线程的缺点：

1. 大量的线程降低代码的可读性；
2. 更多的线程需要更多的内存空间
3. 当多个线程对同一个资源出现争夺时候要注意线程安全的问题。

4、多线程的上下文切换：

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

14. ThreadLocal的设计理念与作用?

1、Java中的ThreadLocal类允许我们创建只能被同一个线程读写的变量。因此，如果一段代码含有一个ThreadLocal变量的引用，即使两个线程同时执行这段代码，它们也无法访问到对方的ThreadLocal变量。

1、概念：线程局部变量。在并发编程的时候，成员变量如果不做任何处理其实是线程不安全的，各个线程都在操作同一个变量，显然是不行的，并且我们也知道volatile这个关键字也是不能保证线程安全的。那么在有一种情况之下，我们需要满足这样一个条件：变量是同一个，但是每个线程都使用同一个初始值，也就是使用同一个变量的一个新的副本。这种情况之下ThreadLocal就非常适用，比如说DAO的数据库连接，我们知道DAO是单例的，那么他的属性Connection就不是一个线程安全的变量。而我们每个线程都需要使用他，并且各自使用各自的。这种情况，ThreadLocal就比较好的解决了这个问题。

2、原理：从本质来讲，就是每个线程都维护了一个map，而这个map的key就是threadLocal，而值就是我们set的那个值，每次线程在get的时候，都从自己的变量中取值，既然从自己的变量中取值，那肯定就不存在线程安全问题，总体来讲，ThreadLocal这个变量的状态根本没有发生变化，他仅仅是充当一个key的角色，另外提供给每一个线程一个初始值。

3、实现机制：每个Thread对象内部都维护了一个ThreadLocalMap这样一个ThreadLocal的Map，可以存放若干个ThreadLocal。

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;
```

4、应用场景：当很多线程需要多次使用同一个对象，并且需要该对象具有相同初始化值的时候最适合使用ThreadLocal。

15. ThreadPool（线程池）用法与优势?

1、ThreadPool 优点

1、减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务

2、可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为因为消耗过多的内存，而把服务器累趴下（每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机）

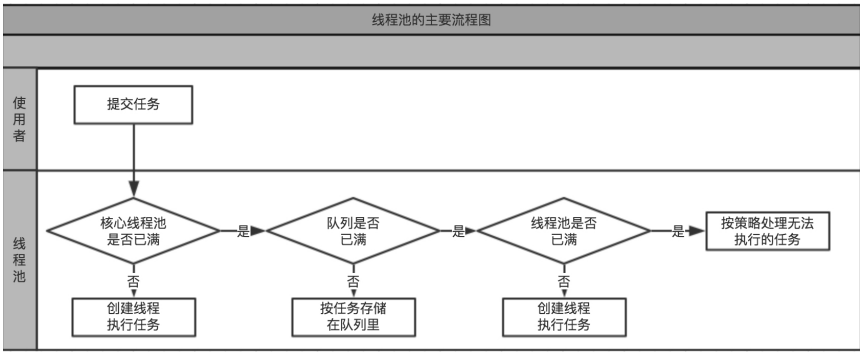
- 减少在创建和销毁线程上所花的时间以及系统资源的开销
- 如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存

2、比较重要的几个类：

类	描述
ExecutorService	真正的线程池接口。
ScheduledExecutorService	能和Timer/TimerTask类似，解决那些需要任务重复执行的问题。
ThreadPoolExecutor	ExecutorService的默认实现。
ScheduledThreadPoolExecutor	继承ThreadPoolExecutor的ScheduledExecutorService接口实现，周期性任务调度的类实现。

> Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。

3、任务执行顺序：



i. 当线程数小于corePoolSize时，创建线程执行任务。

ii. 当线程数大于等于corePoolSize并且workQueue没有满时，放入workQueue中

iii. 线程数大于等于corePoolSize并且当workQueue满时，新任务新建线程运行，线程总数要小于maximumPoolSize

iv. 当线程总数等于maximumPoolSize并且workQueue满了的时候执行handler的rejectedExecution。也就是拒绝策略。

16. Concurrent包里的其他东西：ArrayBlockingQueue、CountDownLatch等等。

1、ArrayBlockingQueue 数组结构组成的有界阻塞队列：

2、**CountDownLatch 允许一个或多个线程等待其他线程完成操作**；

> join用于让当前执行线程等待join线程执行结束。其实现原理是不停检查join线程是否存活，如果join线程存活则让当前线程永远wait

17. synchronized和ReentrantLock的区别?

1、基础知识

- 可重入锁。可重入锁是指同一个线程可以多次获取同一把锁。ReentrantLock和synchronized都是可重入锁。
- 可中断锁。可中断锁是指线程尝试获取锁的过程中，是否可以响应中断。synchronized是不可中断锁，而ReentrantLock则提供了中断功能。
- 公平锁与非公平锁。公平锁是指多个线程同时尝试获取同一把锁时，获取锁的顺序按照线程达到的顺序，而非公平锁则允许线程“插队”。synchronized是非公平锁，而ReentrantLock的默认实现是非公平锁，但是也可以设置为公平锁。
- CAS操作(CompareAndSwap)。CAS操作简单的说就是比较并交换。CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

2、Synchronized

i. synchronized是java内置的关键字，它提供了一种独占的加锁方式。synchronized的获取和释放锁由JVM实现，用户不需要显示的释放锁，非常方便。然而synchronized也有一定的局限性：

1. 当线程尝试获取锁的时候，如果获取不到锁会一直阻塞。
2. 如果获取锁的线程进入休眠或者阻塞，除非当前线程异常，否则其他线程尝试获取锁必须一直等待。

3、ReentrantLock

i. ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成。

ii. 等待可中断避免，出现死锁的情况（如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回true，如果等待超时，返回false）

iii. 公平锁与非公平锁多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

18. ~~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

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

1、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。

2、它的优势有：

- 可以使锁更公平

- 可以使线程在等待锁的时候响应中断
- 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
- 可以在不同的范围，以不同的顺序获取和释放锁

20. Hashtable的size()方法中明明只有一条语句"return count"，为什么还要做同步?

1、同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程A在执行Hashtable的put方法添加数据，线程B则可以正常调用size()方法读取Hashtable中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对size++，线程B就已经读取size了，那么对于线程B来说读取到的size一定是不准确的。

2、而给size()方法加了同步之后，意味着线程B调用size()方法只有在线程A调用put方法完毕之后才可以调用，这样就保证了线程安全性。

21. ConcurrentHashMap的并发度是什么?

1、工作机制（分片思想）：它引入了一个**"分段锁"**的概念，具体可以理解为把一个大的Map拆分成N个小的segment，根据key.hashCode()来决定把key放到哪个HashTable中。可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。

2、应用：当读>写时使用，适合做缓存，在程序启动时初始化，之后可以被多个线程访问；

3、hash冲突：

1、简介：HashMap中调用hashCode()方法来计算hashCode。由于在Java中两个不同的对象可能有一样的hashCode,所以不同的键可能有一样hashCode，从而导致冲突的产生。

2、hash冲突解决：使用平衡树来代替链表，当同一hash中的元素数量超过特定的值便会由链表切换到平衡树

4、无锁读：ConcurrentHashMap之所以有较好的并发性是因为ConcurrentHashMap是无锁读和加锁写，并且利用了分段锁（不是在所有的entry上加锁，而是在一部分entry上加锁）；

```
/* Specialized implementations of map methods */

V get(Object key, int hash) {
    if (count != 0) { // read-volatile
        HashEntry<K,V> e = getFirst(hash);
        while (e != null) {
            if (e.hash == hash && key.equals(e.key)) {
                V v = e.value;
                if (v != null)
                    return v;
                return readValueUnderLock(e); // recheck
            }
            e = e.next;
        }
    }
    return null;
}
```

读之前会先判断count(jdk1.6)，其中的count是被volatile修饰的(当变量被volatile修饰后，每次更改该变量的时候会将更改结果写到系统主内存中，利用多处理器的缓存一致性，其他处理器会发现自己的缓存行对应的内存地址被修改，就会将自己处理器的缓存行设置为失效，并强制从系统主内存获取最新的数据。)，故可以实现无锁读。

5、ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势。

22. ReentrantReadWriteLock读写锁的使用?

1、读写锁：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，你只要上好相应的锁即可。

2、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；

3、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!

23. CyclicBarrier和CountDownLatch的用法及区别?

CyclicBarrier和CountDownLatch 都位于java.util.concurrent 这个包下

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为0时释放所有等待的线程	计数达到指定值时释放所有等待线程
计数为0时，无法重置	计数达到指定值时，计数置为0重新开始
调用countDown()方法计数减一，调用await()方法只进行阻塞，对计数没任何影响	调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

24. ~~LockSupport~~工具?

LockSupport是JDK中比较底层的类，用来创建锁和其他同步工具类的基本线程阻塞。java锁和同步器框架的核心 AQS: AbstractQueuedSynchronizer，就是通过调用 LockSupport .park()和 LockSupport .unpark()实现线程的阻塞和唤醒 的。

25. Condition接口及其实现原理?

1. 在java.util.concurrent包中，有两个很特殊的工具类，Condition和ReentrantLock，使用过的人都知道，ReentrantLock（重入锁）是jdk的concurrent包提供的一种独占锁的实现
2. 我们知道在线程的同步时可以使一个线程阻塞而等待一个信号，同时放弃锁使其他线程可以能竞争到锁
3. 在synchronized中我们可以使用Object的wait()和notify方法实现这种等待和唤醒
4. 但是在Lock中怎么实现这种wait和notify呢？答案是Condition，学习Condition主要是为了方便以后学习blockqueue和concurrenthashmap的源码，同时也进一步理解ReentrantLock。

26. Fork/Join框架的理解?

1、Fork就是把一个大任务切分为若干子任务并行的执行。

2、Join就是合并这些子任务的执行结果，最后得到这个大任务的结果。

27. wait()和sleep()的区别?

1、sleep()

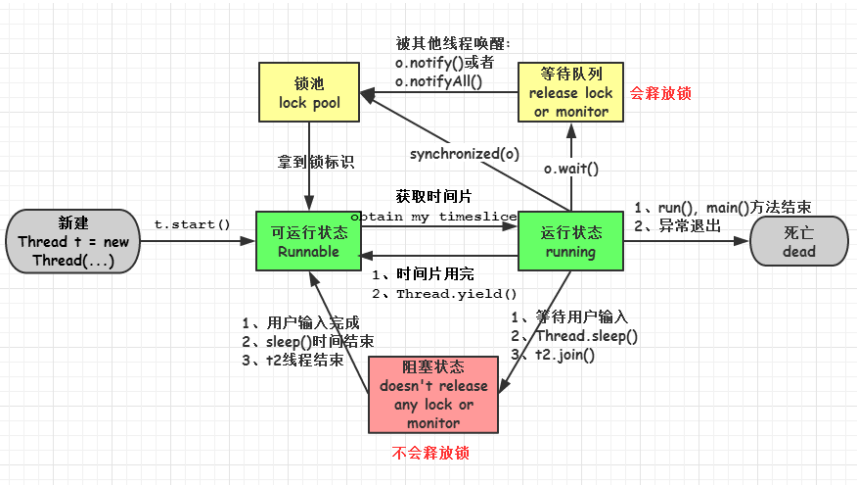
方法是线程类（Thread）的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。

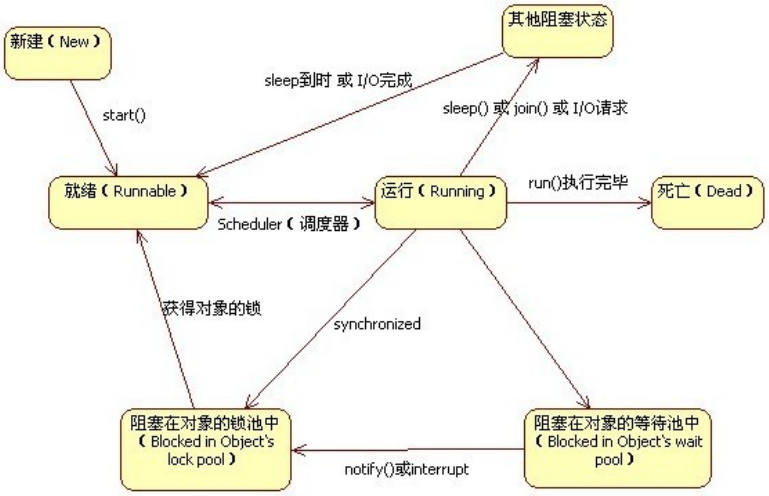
因为sleep() 是static静态的方法，他不能改变对象的机锁，当一个synchronized块中调用了sleep() 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

2、wait()

wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程

28. 线程的五个状态（五种状态，创建、就绪、运行、阻塞和死亡）?





线程通常都有五种状态，创建、就绪、运行、阻塞和死亡。

i. 第一是创建状态。在生成线程对象，并没有调用该对象的start方法，这是线程处于创建状态。

ii. 第二是就绪状态。当调用了线程对象的start方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。

iii. 第三是运行状态。线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行run函数当中的代码。

iv. 第四是阻塞状态。线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。sleep,suspend，wait等方法都可以导致线程阻塞。

v. 第五是死亡状态。如果一个线程的run方法执行结束或者调用stop方法后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪。

29. start()方法和run()方法的区别?

1、start()方法来启动一个线程，真正实现了多线程运行。

2、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，直接调用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。

30. Runnable接口和Callable接口的区别?

1. Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；

2. Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

31. volatile关键字的作用?

1. 多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据。

2. 代码底层执行不像我们看到的高级语言——Java程序这么简单，它的执行是Java代码–>字节码–>根据字节码执行对应的C/C++代码–>C/C++代码被编译成汇编语言–>和硬件电路交互，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率。

32. Java中如何获取到线程dump文件?

死循环、死锁、阻塞、页面打开慢等问题，查看线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

1、获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java

2、打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

3、另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取到的是具体某个线程当前运行的堆栈。

33. 线程和进程有什么区别?

1. 进程是系统进行资源分配的基本单位，有独立的内存地址空间

2. 线程是CPU独立运行和独立调度的基本单位，没有单独地址空间，有独立的栈，局部变量，寄存器，程序计数器等。

3. 创建进程的开销大，包括创建虚拟地址空间等需要大量系统资源

4. 创建线程开销小，基本上只有一个内核对象和一个堆栈。
5. 一个进程无法直接访问另一个进程的资源；同一进程内的多个线程共享进程的资源。
6. 进程切换开销大，线程切换开销小；进程间通信开销大，线程间通信开销小。
7. 线程属于进程，不能独立执行。每个进程至少要有一个线程，成为主线程

34. 线程实现的方式有几种（四种）？

1. 继承Thread类，重写run方法
2. 实现Runnable接口，重写run方法，实现Runnable接口的实现类的实例对象作为Thread构造函数的target
3. 实现Callable接口通过FutureTask包装器来创建Thread线程
4. 通过线程池创建线程

```
public class ThreadDemo03 {
    public static void main(String[] args) {
        Callable<Object> oneCallable = new Tickets<Object>();
        FutureTask<Object> oneTask = new FutureTask<Object>(oneCallable);
        Thread t = new Thread(oneTask);
        System.out.println(Thread.currentThread().getName());
        t.start();
    }
}

class Tickets<Object> implements Callable<Object>{
    //重写call方法
    @Override
    public Object call() throws Exception {
        // TODO Auto-generated method stub
        System.out.println(Thread.currentThread().getName()+"-->我是通过实现Callable接口通过Fut
        return null;
    }
}
```

35. 高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

1. 高并发、任务执行时间短的业务：线程池线程数可以设置为CPU核数+1，减少线程上下文的切换。
2. 并发不高、任务执行时间长的业务要区分开看：
   a. 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
   b. 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换
3. 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

36. 如果你提交任务时，线程池队列已满，这时会发生什么？

1、如果你使用的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、如果你使用的是有界队列比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

37. 锁的等级：方法锁、对象锁、类锁？

1. 方法锁（synchronized修饰方法时）
   a. 通过在方法声明中加入 synchronized关键字来声明 synchronized 方法。
   b. synchronized 方法控制对类成员变量的访问：
   c. 每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处

于可执行状态，从而有效避免了类成员变量的访问冲突。

2. 对象锁（synchronized修饰方法或代码块）

a. 当一个对象中有synchronized method或synchronized block的时候调用此对象的同步方法或进入其同步区域时，就必须先获得对象锁。如果此对象的对象锁已被其他调用者占用，则需要等待此锁被释放。（方法锁也是对象锁）

b. java的所有对象都含有1个互斥锁，这个锁由JVM自动获取和释放。线程进入synchronized方法的时候获取该对象的锁，当然如果已经有线程获取了这个对象的锁，那么当前线程会等待；**synchronized方法正常返回或者抛异常而终止，JVM会自动释放对象锁。**这里也体现了用synchronized来加锁的1个好处，方法抛异常的时候，锁仍然可以由JVM来自动释放。

3. 类锁(synchronized 修饰静态的方法或代码块)

a. 由于一个class不论被实例化多少次，其中的静态方法和静态变量在内存中都只有一份。所以，一旦一个静态的方法被申明为synchronized。此类所有的实例化对象在**调用此方法**，共用同一把锁，我们称之为类锁。

4. 对象锁是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步

38. 如果同步块内的线程抛出异常会发生什么?

synchronized方法正常返回或者抛异常而终止，JVM会自动释放对象锁

39. 并发编程（concurrency）并行编程（parallellism）有什么区别?

1. 解释一：并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。

2. 解释二：并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。

3. 解释三：在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

> 所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

40. 如何保证多线程下 i++ 结果正确?

1. volatile只能保证你数据的可见性，获取到的是最新的数据，不能保证原子性；

2. 用AtomicInteger保证原子性。

3. synchronized既能保证共享变量可见性，也可以保证锁内操作的原子性。

41. 一个线程如果出现了运行时异常会怎么样?

1. 如果这个异常没有被捕获的话，这个线程就停止执行了。

2. 另外重要的一点是：如果这个线程持有某个对象的监视器，那么这个对象监视器会被立即释放.

42. 如何在两个线程之间共享数据?

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的。

1. 卖票系统：

```
package 多线程共享数据;
public class Ticket implements Runnable{
    private int ticket = 10;
    public void run() {
        while(ticket>0){
            ticket--;
            System.out.println("当前票数为: "+ticket);
        }
    }
}

package 多线程共享数据;
public class SellTicket {
    public static void main(String[] args) {
        Ticket t = new Ticket();
        new Thread(t).start();
        new Thread(t).start();
    }
}
```

2. 银行存取款：

```java
public class MyData {
    private int j=0;
    public  synchronized void add(){
        j++;
        System.out.println("线程"+Thread.currentThread().getName()+"j为: "+j);
    }
    public  synchronized void dec(){
        j--;
        System.out.println("线程"+Thread.currentThread().getName()+"j为: "+j);
     }
     public int getData(){
         return j;
     }
}

public class AddRunnable implements Runnable{
    MyData data;
    public AddRunnable(MyData data){
        this.data= data;
    }
    public void run() {
        data.add();
    }
}

public class DecRunnable implements Runnable {
    MyData data;
    public DecRunnable(MyData data){
        this.data = data;
    }
    public void run() {
        data.dec();
    }
}

public class TestOne {
    public static void main(String[] args) {
        MyData data = new MyData();
        Runnable add = new AddRunnable(data);
        Runnable dec = new DecRunnable(data);
        for(int i=0;i<2;i++){
            new Thread(add).start();
            new Thread(dec).start();
        }
    }
```

43. 生产者消费者模型的作用是什么?

1. 通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用。
2. 解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要受到相互的制约。

44. 怎么唤醒一个阻塞的线程?

1. 如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞；

1、suspend与resume

Java废弃 suspend() 去挂起线程的原因，是因为 suspend() 在导致线程暂停的同时，并不会去释放任何锁资源。其他线程都无法访问被它占用的锁。直到对应的线程执行 resume() 方法后，被挂起的线程才能继续，从而其它被阻塞在这个锁的线程才可以继续执行。

但是，如果 resume() 操作出现在 suspend() 之前执行，那么线程将一直处于挂起状态，同时一直占用锁，这就产生了死锁。而且，对于被挂起的线程，它的线程状态居然还是 Runnable。

2、wait与notify

wait与notify必须配合synchronized使用，因为调用之前必须持有锁，wait会立即释放锁，notify则是同步块执行完了才释放

3、await与singal

Condition类提供，而Condition对象由new ReentLock().newCondition()获得，与wait和notify相同，因为使用Lock锁后无法使用wait方法

4、park与unpark

LockSupport是一个非常方便实用的线程阻塞工具，它可以在线程任意位置让线程阻塞。和Thread.suspenf()相比，它弥补了由于resume()在前发生，导致线程无法继续执行的情况。和Object.wait()相比，它不需要先获得某个对象的锁，也不会抛出IException异常。可以唤醒指定线程。

2. 如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统。

45. Java中用到的线程调度算法是什么

1. 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。

46. 单例模式的线程安全性?

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法，我总结一下：

（1）饿汉式单例模式的写法：线程安全

（2）懒汉式单例模式的写法：非线程安全

（3）双检锁单例模式的写法：线程安全

47. 线程类的构造方法、静态块是被哪个线程调用的?

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。

48. 同步方法和同步块，哪个是更好的选择?

1. 同步块是更好的选择，因为它不会锁住整个对象（当然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。

synchronized(this)以及非static的synchronized方法（至于static synchronized方法请往下看），只能防止多个线程同时执行同一个对象的同步代码段。

如果要锁住多个对象方法，可以锁住一个固定的对象，或者锁住这个类的Class对象。

synchronized锁住的是括号里的对象，而不是代码。对于非static的synchronized方法，锁的就是对象本身也就是this。

2. 例如：

```
public class SynObj{

    public synchronized void showA(){
        System.out.println("showA..");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void showB(){
        synchronized (this) {
            System.out.println("showB..");
        }
    }
```

```
}
```

49. 如何检测死锁？怎么预防死锁?

1. 概念：

是指两个或两个以上的进程在执行过程中，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。此时称系统处于死锁；

2. 死锁的四个必要条件：

i. 互斥条件：进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源

ii. 请求和保持条件：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此时请求阻塞，但又对自己获得的资源保持不放

iii. 不可剥夺条件：是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放

iv. 环路等待条件：是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

2. 死锁产生的原因：

1.因竞争资源发生死锁 现象：系统中供多个进程共享的资源的数目不足以满足全部进程的需要时，就会引起对诸资源的竞争而发生死锁现象

2.进程推进顺序不当发生死锁

3. 检查死锁

i. 有两个容器，一个用于保存线程正在请求的锁，一个用于保存线程已经持有的锁。每次加锁之前都会做如下检测:

ii. 检测当前正在请求的锁是否已经被其它线程持有,如果有，则把那些线程找出来

iii. 遍历第一步中返回的线程，检查自己持有的锁是否正被其中任何一个线程请求，如果第二步返回真,表示出现了死锁

4. 死锁的解除与预防：控制不要让四个必要条件成立。

50. HashMap在多线程环境下使用需要注意什么?

要注意死循环的问题，HashMap的put操作引发扩容，这个动作在多线程并发下会发生线程死循环的问题。

1、HashMap不是线程安全的；Hashtable线程安全，但效率低，因为是Hashtable是使用synchronized的，所有线程竞争同一把锁；而ConcurrentHashMap不仅线程安全而且效率高，因为它包含一个segment数组，将数据分段存储，给每一段数据配一把锁，也就是所谓的锁分段技术。

2、HashMap为何线程不安全：

1、put时key相同导致其中一个线程的value被覆盖；

2、多个线程同时扩容，造成数据丢失；

3、多线程扩容时导致Node链表形成环形结构造成.next()死循环，导致CPU利用率接近100%；

3、ConcurrentHashMap最高效；

51. 什么是守护线程？有什么用?

守护线程（即daemon thread），是个服务线程，准确地来说就是服务其他的线程，这是它的作用——而其他的线程只有一种，那就是用户线程。所以java里线程分2种，

1、守护线程，比如垃圾回收线程，就是最典型的守护线程。

2、用户线程，就是应用程序里的自定义线程。

52. 如何实现线程串行执行?

a. 为了控制线程执行的顺序，如ThreadA->ThreadB->ThreadC->ThreadA循环执行三个线程，我们需要确定唤醒、等待的顺序。这时我们可以同时使用 Obj.wait()、Obj.notify()与synchronized(Obj)来实现这个目标。

线程中持有上一个线程类的对象锁以及自己的锁，由于这种依赖关系，该线程执行需要等待上个对象释放锁，从而保证类线程执行的顺序。

b. 通常情况下，wait是线程在获取对象锁后，主动释放对象锁，同时本线程休眠，直到有其它线程调用对象的notify()唤醒该线程，才能继续获取对象锁，并继续执行。而notify()则是对等待对象锁的线程的唤醒操作。但值得注意的是notify()调用后，并不是马上就释放对象锁，而是在相应的synchronized(){}语句块执行结束。释放对象锁后，JVM会在执行wait()等待对象锁的线程中随机选取一线程，赋予其对象锁，唤醒线程，继续执行。

```
public class ThreadSerialize {

    public static void main(String[] args){
        ThreadA threadA = new ThreadA();
        ThreadB threadB = new ThreadB();
        ThreadC threadC = new ThreadC();

```

```
        threadA.setThreadC(threadC);
        threadB.setThreadA(threadA);
         threadC.setThreadB(threadB);

         threadA.start();
         threadB.start();
         threadC.start();

         while (true){
             try {
                 Thread.currentThread().sleep(1000);
             } catch (InterruptedException e) {
                 e.printStackTrace();
             }
         }
    }
}

  class ThreadA extends Thread{
    private ThreadC threadC;
    @Override
    public void run() {
        while (true){
            synchronized (threadC){
                synchronized (this){
                    System.out.println("I am ThreadA。。。");
                    this.notify();
                }
                try {
                    threadC.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

    }

    public void setThreadC(ThreadC threadC) {
        this.threadC = threadC;
    }
}
  class ThreadB extends Thread{
    private ThreadA threadA;
    @Override
    public void run() {
        while (true){
            synchronized (threadA){
                synchronized (this){
                    System.out.println("I am ThreadB。。。");
                    this.notify();
                }
```

```
                try {
                    threadA.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

    }

    public void setThreadA(ThreadA threadA) {
        this.threadA = threadA;
    }
}
  class ThreadC extends Thread{
    private ThreadB threadB;
    @Override
    public void run() {
        while (true){
            synchronized (threadB){
                synchronized (this){
                    System.out.println("I am ThreadC。。。");
                    this.notify();
                }
                try {
                    threadB.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

    }

    public void setThreadB(ThreadB threadB) {
        this.threadB = threadB;
    }
}


```

53. 可以运行时kill掉一个线程吗?

a. 不可以，线程有5种状态，新建（new）、可运行（runnable）、运行中（running）、阻塞（block）、死亡（dead）。

b. 只有当线程run方法或者主线程main方法结束，又或者抛出异常时，线程才会结束生命周期。

54. 关于`synchronized`:

1. 在某个对象的所有`synchronized`方法中,在某个时刻只能有一个唯一的一个线程去访问这些`synchronized`方法

2. 如果一个方法是`synchronized`方法,那么该`synchronized`关键字表示给当前对象上锁(即`this`)相当于`synchronized(this){}`

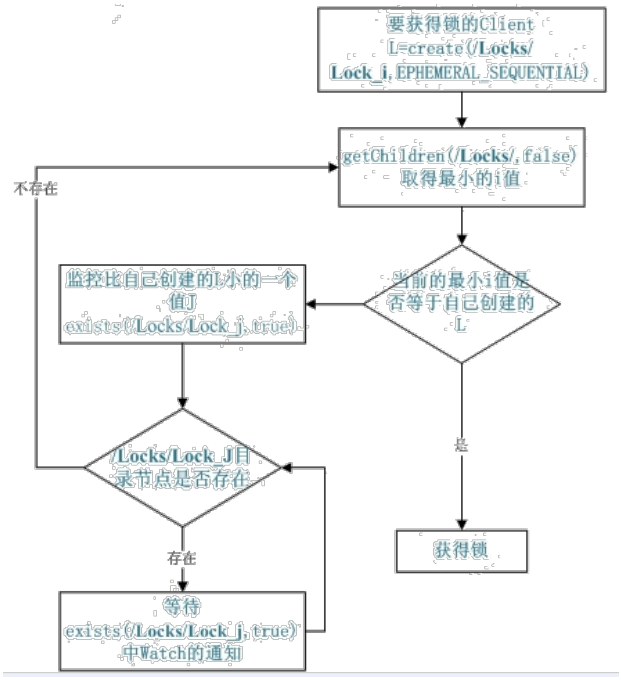
3. 如果一个`synchronized`方法是`static`的,那么该`synchronized`表示给当前对象所对应的`class`对象上锁(每个类不管生成多少对象,其对应的`class`对象只有一个)

55. 分步式锁,程序数据库中死锁机制及解决方案

基本原理：用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

1、三种分布式锁：

1、Zookeeper：基于zookeeper瞬时有序节点实现的分布式锁，其主要逻辑如下。大致思想即为：每个客户端对某个功能加锁时，在zookeeper上的与该功能对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。



2、优点

锁安全性高，zk可持久化，且能实时监听获取锁的客户端状态。一旦客户端宕机，则瞬时节点随之消失，zk因而能第一时间释放锁。这也省去了用分布式缓存实现锁的过程中需要加入超时时间判断的这一逻辑。

3、缺点

性能开销比较高。因为其需要动态产生、销毁瞬时节点来实现锁功能。所以不太适合直接提供给高并发的场景使用。

4、实现

可以直接采用zookeeper第三方库curator即可方便地实现分布式锁。

5、适用场景

对可靠性要求非常高，且并发程度不高的场景下使用。如核心数据的定时全量/增量同步等。

2、memcached：memcached带有add函数，利用add函数的特性即可实现分布式锁。add和set的区别在于：如果多线程并发set，则每个set都会成功，但最后存储的值以最后的set的线程为准。而add的话则相反，add会添加第一个到达的值，并返回true，后续的添加则都会返回false。利用该点即可很轻松地实现分布式锁。

2、优点

并发高效

3、缺点

memcached采用列入LRU置换策略，所以如果内存不够，可能导致缓存中的锁信息丢失。

memcached无法持久化，一旦重启，将导致信息丢失。

4、使用场景

高并发场景。需要 1)加上超时时间避免死锁； 2)提供足够支撑锁服务的内存空间； 3)稳定的集群化管理。

3、redis：redis分布式锁即可以结合zk分布式锁锁高度安全和memcached并发场景下效率很好的优点，其实现方式和memcached类似，采用setnx即可实现。需要注意的是，这里的redis也需要设置超时时间，以避免死锁。可以利用jedis客户端实现。

```
ICacheKey cacheKey = new ConcurrentCacheKey(key, type);
return RedisDao.setnx(cacheKey, "1");
```

2、数据库死锁机制和解决方案：

1、死锁：死锁是指两个或者两个以上的事务在执行过程中，因争夺锁资源而造成的一种互相等待的现象。

2、处理机制：解决死锁最有用最简单的方法是不要有等待，将任何等待都转化为回滚，并且事务重新开始。但是有可能影响并发性能。

1、超时回滚，innodb_lock_wait_time设置超时时间；

2、wait-for-graph方法：跟超时回滚比起来，这是一种更加主动的死锁检测方式。InnoDB引擎也采用这种方式。

56. spring单例为什么没有安全问题(ThreadLocal)

1、ThreadLocal：spring使用ThreadLocal解决线程安全问题；ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。概括起来说，对于多线程资源共享的问题，同步机制采用了"以时间换空间"的方式，而ThreadLocal采用了"以空间换时间"的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

2、单例：无状态的Bean(无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象，不能保存数据，是不变类，是线程安全的。)适合用不变模式，技术就是单例模式，这样可以共享实例，提高性能。

57. 线程池原理：

1、使用场景：假设一个服务器完成一项任务所需时间为：T1-创建线程时间，T2-在线程中执行任务的时间，T3-销毁线程时间。如果T1+T3远大于T2，则可以使用线程池，以提高服务器性能；

2、组成：

1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；

2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；

4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

2、原理：线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。

3、工作流程：

1、线程池刚创建时，里面没有一个线程(也可以设置参数prestartAllCoreThreads启动预期数量主线程)。任务队列是作为参数传进来的。不过，就算队列里面有任务，线程池也不会马上执行它们。

2、当调用 execute() 方法添加一个任务时，线程池会做如下判断：

1. 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；
2. 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；
3. 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；
4. 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常RejectExecutionException。

3、当一个线程完成任务时，它会从队列中取下一个任务来执行。

4、当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

58. java锁多个对象：

例如： 在银行系统转账时，需要锁定两个账户，这个时候，顺序使用两个synchronized可能存在死锁的情况，在网上搜索到下面的例子：

```
public class Bank {
    final static Object obj_lock = new Object();

    // Deadlock crisis 死锁
    public void transferMoney(Account from, Account to, int number) {
        synchronized (from) {
            synchronized (to) {
                from.debit();
                to.credit();
            }
        }
    }

    // Thread safe
```

```
    public void transferMoney2(final Account from, final Account to, int number) {
        class Help {
            void transferMoney2() {
                from.debit();
                to.credit();
            }
        }

        //通过hashCode大小调整加锁顺序
        int fromHash = from.hashCode();
        int toHash = to.hashCode();

        if (fromHash < toHash) {
            synchronized (from) {
                synchronized (to) {
                    new Help().transferMoney2();
                }
            }
        } else if (toHash < fromHash) {
            synchronized (to) {
                synchronized (from) {
                    new Help().transferMoney2();
                }
            }
        } else {
            synchronized (obj_lock) {
                synchronized (to) {
                    synchronized (from) {
                        new Help().transferMoney2();
                    }
                }
            }
        }
    }
}
```

若操作账户A，B：

1. A的hashCode小于B，先锁A再锁B
2. B的hashCode小于A，先锁B再锁A
3. 产生的hashCode相等，先锁住一个全局静态变量，在锁A，B

这样就避免了两个线程分别操作账户A,B和B,A而产生死锁的情况。

需要为Account对象写一个好的hashCode算法，使得不同账户间产生的hashCode尽量不同。

59. java线程如何启动：

1、继承Thread类；

2、实现Runnable接口；

3、直接在函数体内：

4、比较：

1、实现Runnable接口优势：

1）适合多个相同的程序代码的线程去处理同一个资源

2）可以避免java中的单继承的限制

3）增加程序的健壮性，代码可以被多个线程共享，代码和数据独立。

2、继承Thread类优势：

1）可以将线程类抽象出来，当需要使用抽象工厂模式设计时。

2）多线程同步

3、在函数体使用优势

1）无需继承thread或者实现Runnable，缩小作用域。

60．java中加锁的方式有哪些，如何实现怎么个写法．

1、java中有两种锁：一种是方法锁或者对象锁（在非静态方法或者代码块上加锁），第二种是类锁（在静态方法或者class上加锁）；

2、注意：其他线程可以访问未加锁的方法和代码；synchronized同时修饰静态方法和实例方法，但是运行结果是交替进行的，这证明了类锁和对象锁是两个不一样的锁，控制着不同的区域，它们是互不干扰的。

3、示例代码：

1、方法锁和同步代码块：

```
public class TestSynchronized
{
    public void test1()
    {
        synchronized(this)
        {
            int i = 5;
            while( i-- > 0)
            {
                System.out.println(Thread.currentThread().getName() + " : " + i);
                try
                {
                    Thread.sleep(500);
                }
                catch (InterruptedException ie)
                {
                }
            }
        }
    }

    public synchronized void test2()
    {
        int i = 5;
        while( i-- > 0)
        {
            System.out.println(Thread.currentThread().getName() + " : " + i);
            try
            {
                Thread.sleep(500);
            }
            catch (InterruptedException ie)
            {
            }
        }
    }

    public static void main(String[] args)
    {
        final TestSynchronized myt2 = new TestSynchronized();
        Thread test1 = new Thread(  new Runnable() {  public void run() {  myt2.test1();  }
        Thread test2 = new Thread(  new Runnable() {  public void run() { myt2.test2();   }
        test1.start();;
        test2.start();
```

```
//          TestRunnable tr=new TestRunnable();
//          Thread test3=new Thread(tr);
//          test3.start();
    }
}
```

2、类锁:

```
public class TestSynchronized
{
    public void test1()
    {
        synchronized(TestSynchronized.class)
        {
            int i = 5;
            while( i-- > 0)
            {
                System.out.println(Thread.currentThread().getName() + " : " + i);
                try
                {
                    Thread.sleep(500);
                }
                catch (InterruptedException ie)
                {
                }
            }
        }
    }

    public static synchronized void test2()
    {
        int i = 5;
        while( i-- > 0)
        {
            System.out.println(Thread.currentThread().getName() + " : " + i);
            try
            {
                Thread.sleep(500);
            }
            catch (InterruptedException ie)
            {
            }
        }
    }

    public static void main(String[] args)
    {
        final TestSynchronized myt2 = new TestSynchronized();
        Thread test1 = new Thread(  new Runnable() {  public void run() {  myt2.test1();  }
        Thread test2 = new Thread(  new Runnable() {  public void run() { TestSynchronized.
        test1.start();
        test2.start();
//          TestRunnable tr=new TestRunnable();
```

```
//          Thread test3=new Thread(tr);
//          test3.start();
    }

}
```

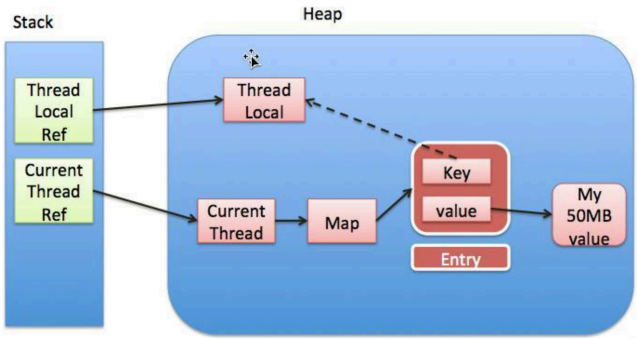
62、如何保证数据不丢失：

1、使用消息队列，消息持久化；

2、添加标志位：未处理 0，处理中 1，已处理 2。定时处理。

63、ThreadLocal为什么会发生内存泄漏?

1、threadlocal原理图：



2、OOM实现：

1、ThreadLocal的实现是这样的：每个Thread 维护一个 ThreadLocalMap 映射表，这个映射表的 key 是 ThreadLocal实例本身，value 是真正需要存储的 Object。

2、也就是说 ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。值得注意的是图中的虚线，表示 ThreadLocalMap 是使用 ThreadLocal 的弱引用作为 Key 的，弱引用的对象在 GC 时会被回收。

3、ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无法回收，造成内存泄漏。

3、预防办法：在ThreadLocal的get(),set(),remove()的时候都会清除线程ThreadLocalMap里所有key为null的value。

但是这些被动的预防措施并不能保证不会内存泄漏：

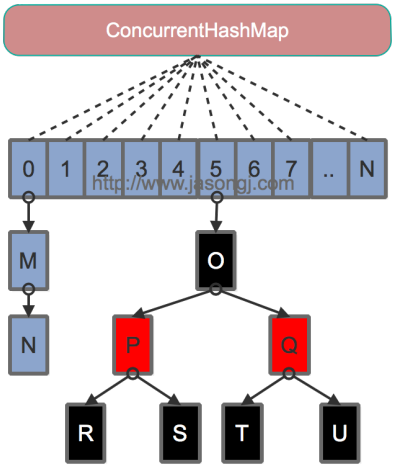
（1）使用static的ThreadLocal，延长了ThreadLocal的生命周期，可能导致内存泄漏。

（2）分配使用了ThreadLocal又不再调用get(),set(),remove()方法，那么就会导致内存泄漏，因为这块内存一直存在。

64、jdk8中对ConcurrentHashmap的改进

1. Java 7为实现并行访问，引入了Segment这一结构，实现了分段锁，理论上最大并发度与Segment个数相等。

2. Java 8为进一步提高并发性，摒弃了分段锁的方案，而是直接使用一个大的数组。同时为了提高哈希碰撞下的寻址性能，Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(long(N))）。其数据结构如下图所示



3、源码：

```
public V put(K key, V value) {
    return putVal(key, value, false);
}

    /** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    //ConcurrentHashMap 不允许插入null键，HashMap允许插入一个null键
    if (key == null || value == null) throw new NullPointerException();
    //计算key的hash值
     int hash = spread(key.hashCode());
     int binCount = 0;
     //for循环的作用：因为更新元素是使用CAS机制更新，需要不断的失败重试，直到成功为止。
     for (Node<K,V>[] tab = table;;) {
         // f：链表或红黑二叉树头结点，向链表中添加元素时，需要synchronized获取f的锁。
         Node<K,V> f; int n, i, fh;
         //判断Node[]数组是否初始化，没有则进行初始化操作
         if (tab == null || (n = tab.length) == 0)
             tab = initTable();
         //通过hash定位Node[]数组的索引坐标，是否有Node节点，如果没有则使用CAS进行添加（链表的头结点），添
         else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
             if (casTabAt(tab, i, null,
                          new Node<K,V>(hash, key, value, null)))
                 break;                   // no lock when adding to empty bin
         }
         //检查到内部正在移动元素（Node[] 数组扩容）
         else if ((fh = f.hash) == MOVED)
             //帮助它扩容
             tab = helpTransfer(tab, f);
         else {
             V oldVal = null;
             //锁住链表或红黑二叉树的头结点
             synchronized (f) {
                 //判断f是否是链表的头结点
                 if (tabAt(tab, i) == f) {
                     //如果fh>=0 是链表节点
                     if (fh >= 0) {
```

```
                        binCount = 1;
                        //遍历链表所有节点
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            //如果节点存在，则更新value
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            //不存在则在链表尾部添加新节点。
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
                        }
                    }
                    //TreeBin是红黑二叉树节点
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        //添加树节点
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }

            if (binCount != 0) {
                //如果链表长度已经达到临界值8 就需要把链表转换为树结构
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    //将当前ConcurrentHashMap的size数量+1
    addCount(1L, binCount);
    return null;
}
```

65、concurrent包下有哪些类?

ConcurrentHashMap、Future、FutureTask、AtomicInteger...

66、线程a,b,c,d运行任务，怎么保证当a,b,c线程执行完再执行d线程?

1、CountDownLatch类

一个同步辅助类，常用于某个条件发生后才能执行后续进程。给定计数初始化CountDownLatch，调用countDown()方法，在计数到达零之前，await方法一直受阻塞。

重要方法为countdown()与await()；

2、join方法

将线程B加入到线程A的尾部，当A执行完后B才执行。

```
public static void main(String[] args) throws Exception {
    Th t = new Th("t1");
    Th t2 = new Th("t2");
    t.start();
    t.join();
    t2.start();
}
```

3、notify、wait方法，Java中的唤醒与等待方法，关键为synchronized代码块，参数线程间应相同，也常用Object作为参数。

67、高并发系统如何做性能优化？如何防止库存超卖?

1、高并发系统性能优化：

优化程序，优化服务配置，优化系统配置

1.尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与数据库的交互，提高性能。

2.用jprofiler等工具找出性能瓶颈，减少额外的开销。

3.优化数据库查询语句，减少直接使用hibernate等工具的直接生成语句（仅耗时较长的查询做优化）。

4.优化数据库结构，多做索引，提高查询效率。

5.统计的功能尽量做缓存，或按每天一统计或定时统计相关报表，避免需要时进行统计的功能。

6.能使用静态页面的地方尽量使用，减少容器的解析（尽量将动态内容生成静态html来显示）。

7.解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

2、防止库存超卖：

1、悲观锁：在更新库存期间加锁，不允许其它线程修改；

1、数据库锁：select xxx for update；

2、分布式锁；

2、乐观锁：使用带版本号的更新。每个线程都可以并发修改，但在并发时，只有一个线程会修改成功，其它会返回失败。

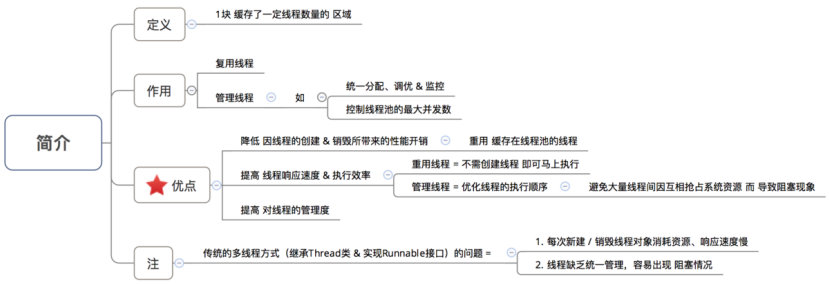
1、redis watch：监视键值对，作用时如果事务提交exec时发现监视的监视对发生变化，事务将被取消。

3、消息队列：通过 FIFO 队列，使修改库存的操作串行化。

4、总结：总的来说，不能把压力放在数据库上，所以使用 "select xxx for update" 的方式在高并发的场景下是不可行的。FIFO 同步队列的方式，可以结合库存限制队列长，但是在库存较多的场景下，又不太适用。所以相对来说，我会倾向于选择：乐观锁 / 缓存锁 / 分布式锁的方式。

68、线程池的参数配置，为什么java官方提供工厂方法给线程池?

1、线程池简介：



2、核心参数：

参数	意义	说明
corePoolSize	核心线程数	默认情况下，核心线程会一直存活 （包括 空闲状态）
maximumPoolSize	线程池所能容纳的最大线程数	当活动线程数到达该数值后，后续的新任务将会阻塞
keepAliveTime	非核心线程 闲置超时时长	超过该时长，非核心线程就会被回收 （当将 allowCoreThreadTimeout 设置为true时，keepAliveTime同样作用于核心线程）
unit	指定keepAliveTime参数的时间单位	常用：(毫秒) TimeUnit.MILLISECONDS、(秒) TimeUnit.SECONDS、(分) TimeUnit.MINUTES
workQueue	任务队列	通过线程池的execute（）方法提交的Runnable对象 将存储在该参数中
threadFactory	线程工厂 （是1个接口）	• 作用 = 为线程池创建新线程 • 具体使用 = 只有1个方法：Thread newThread(Runnable r)

3、工厂方法作用：ThreadPoolExecutor类就是Executor的实现类，但ThreadPoolExecutor在使用上并不是那么方便，在实例化时需要传入很多歌参数，还要考虑线程的并发数等与线程池运行效率有关的参数，所以官方建议使用Executors工程类来创建线程池对象。

69、说说java同步机制，java有哪些锁，每个锁的特性?

70、说说volatile如何保证可见性，从cpu层面分析?