new Thread的弊端如下：

a. 每次new Thread新建对象性能差。

b. 线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom。

c. 缺乏更多功能，如定时执行、定期执行、线程中断。

相比new Thread，Java提供的四种线程池的好处在于：

a. 重用存在的线程，减少对象创建、消亡的开销，性能佳。

b. 可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。

c. 提供定时执行、定期执行、单线程、并发数控制等功能

## Java创建线程之后，直接调用start()方法和run()的区别

答：

1.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，可以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。然后CPU再调度其它线程。

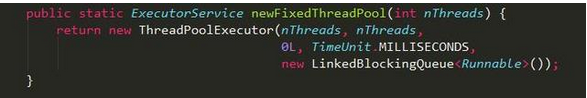
2.run（）方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到写线程的目的。

记住：多线程就是分时利用CPU，宏观上让所有线程一起执行 ，也叫并发

## 1、常用的线程池模式以及不同线程池的使用场景

答：

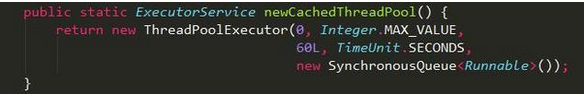
1. 初始化一个指定线程数的线程池，其中corePoolSize == maximumPoolSize，使用LinkedBlockingQuene作为阻塞队列，不过当线程池没有可执行任务时，也不会释放线程



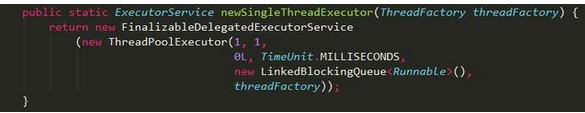
2. 初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到Integer.MAX\_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；

和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；

所以，使用该线程池时，一定要注意控制并发的任务数，否则创建大量的线程可能导致严重的性能问题。



1. 初始化的线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行，内部使用LinkedBlockingQueue作为阻塞队列。



1. 初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据



## 3、newFixedThreadPool此种线程池如果线程数达到最大值后会怎么办，底层原理。

http://blog.csdn.net/zqz\_zqz/article/details/69488570?locationNum=12&fps=1

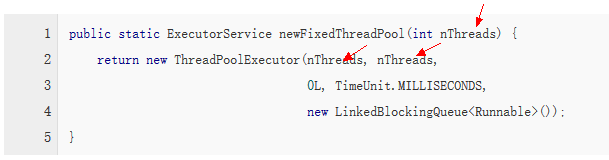
答：FixedThreadPool 用于负载比较重的服务器，为了资源的合理利用，需要限制当前线程数量。

这个线程池执行任务的流程如下：

1.线程数少于核心线程数，也就是设置的线程数时，新建线程执行任务

2.线程数等于核心线程数后，将任务加入阻塞队列，由于队列容量非常大，可以一直加加加

3.执行完任务的线程反复去队列中取任务执行



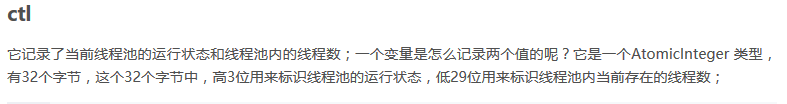
根据这个构造方法可知，newFixedThreadPool的核心线程数和最大线程数是一样的，也就说，当线程池中的线程数超过核心线程数后，任务都会被放到阻塞队列中。

此外 keepAliveTime 为 0，也就是多余的空余线程会被立即终止（由于这里没有多余线程，这个参数也没什么意义了）。

而这里选用的阻塞队列是 LinkedBlockingQueue，使用的是默认容量 Integer.MAX\_VALUE，相当于没有上限。

当线程数达到最大值时，分析源码：





AtomicInteger介绍

AtomicInteger是一个提供原子操作的Integer类，通过线程安全的方式操作加减。

## 4、多线程之间通信的同步问题，synchronized锁的是对象，衍伸出和synchronized相关很多的具体问题，例如同一个类不同方法都有synchronized锁，一个对象是否可以同时访问。（互斥的）或者一个类的static构造方法加上synchronized之后的锁的影响。

答：http://blog.csdn.net/luoweifu/article/details/46613015

一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。

三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。

四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。

五、以上规则对其它对象锁同样适用.



## 5、了解可重入锁的含义，以及ReentrantLock和synchronized的区别

答：<https://my.oschina.net/u/2250599/blog/410585>

一.什么是sychronized

sychronized是java中最基本同步互斥的手段,可以修饰代码块,方法,类.

在修饰代码块的时候需要一个reference对象作为锁的对象.

在修饰方法的时候默认是当前对象作为锁的对象.

在修饰类时候默认是当前类的Class对象作为锁的对象.

synchronized会在进入同步块的前后分别形成monitorenter和monitorexit字节码指令.在执行monitorenter指令时会尝试获取对象的锁,如果此没对象没有被锁,或者此对象已经被当前线程锁住,那么锁的计数器加一,每当monitorexit被锁的对象的计数器减一.直到为0就释放该对象的锁.由此synchronized是可重入的,不会出现自己把自己锁死.

二.什么ReentrantLock

以对象的方式来操作对象锁.相对于sychronized需要在finally中去释放锁

三.synchronized和ReentrantLock的区别

除了synchronized的功能,多了三个高级功能.

等待可中断,公平锁,绑定多个Condition.

1.等待可中断

在持有锁的线程长时间不释放锁的时候,等待的线程可以选择放弃等待. tryLock(long timeout, TimeUnit unit)

2.公平锁

按照申请锁的顺序来一次获得锁称为公平锁.synchronized的是非公平锁,ReentrantLock可以通过构造函数实现公平锁. new RenentrantLock(boolean fair)

3.绑定多个Condition

通过多次newCondition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal();

## 6、同步的数据结构，例如concurrentHashMap的源码理解以及内部实现原理，为什么他是同步的且效率高

答：<http://blog.csdn.net/liuzhengkang/article/details/2916620>

ConcurrentHashMap采用了非常精妙的"分段锁"策略，ConcurrentHashMap的主干是个Segment数组。

final Segment<K,V>[] segments;

　Segment继承了ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在ConcurrentHashMap，一个Segment就是一个子哈希表，Segment里维护了一个HashEntry数组，并发环境下，对于不同Segment的数据进行操作是不用考虑锁竞争的。（就按默认的ConcurrentLeve为16来讲，理论上就允许16个线程并发执行，有木有很酷）

　　所以，对于同一个Segment的操作才需考虑线程同步，不同的Segment则无需考虑。

Segment类似于HashMap，一个Segment维护着一个HashEntry数组

transient volatile HashEntry<K,V>[] table;

HashEntry是目前我们提到的最小的逻辑处理单元了。一个ConcurrentHashMap维护一个Segment数组，一个Segment维护一个HashEntry数组。

## 7、atomicinteger和volatile等线程安全操作的关键字的理解和使用

答: http://www.blogjava.net/hello-yun/archive/2012/12/01/392334.html

volatile本质是在告诉jvm当前变量在寄存器中的值是不确定的,需要从主存中读取,synchronized则是锁定当前变量,只有当前线程可以访问该变量,其他线程被阻塞住.

volatile仅能使用在变量级别,synchronized则可以使用在变量,方法.

volatile仅能实现变量的修改可见性,但不具备原子特性,而synchronized则可以保证变量的修改可见性和原子性.

volatile不会造成线程的阻塞,而synchronized可能会造成线程的阻塞.

volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化.

## 8、线程间通信，wait和notify

答：

调用yield和sleep并没有释放锁，而调用wait则释放锁。可以通过notify()/notifyAll()，或者时间到期，从wait()中恢复执行。

使用wait、notify方法实现线程间的通信(注意这两个方法都是object的类的final方法，换句话说java为所有的对象都提供了这两个方法)

1.wait和notify必须配合synchronized关键字使用（用于协调多个线程对共享数据的存取，所以必须在Synchronized语句块内使用这三个方法）

2.wait方法释放锁，notify方法不释放锁

3．join()方法使调用该方法的线程在此之前执行完毕，也就是等待调用该方法的线程执行完毕后再往下继续执行。注意该方法和sleep方法都要捕获异常。

4．yield()方法是让当前线程直接由运行状态进入就需状态，然后让jvm重新调度一次，但是这次调度只会让处于就绪队列中比当前线程优先级高或者相等的线程运行，很可能某个线程在调用了yield方法后，又被jvm调度进来运行。

## 9、定时线程的使用

答：主要涉及到两个类，一个是java.util.Timer，一个是java.util.TimerTask。Timer:是一种工具，用Timer操作线程，可以在后台执行线程中的任务，可以控制这个任务执行一次，或者定期重复执行。TimerTask就是Timer需要安排的任务。

## 10、场景：在一个主线程中，要求有大量(很多很多)子线程执行完之后，主线程才执行完成。多种方式，考虑效率。

答：参考：http://www.jb51.net/article/96842.htm，方法一：设置一个集合，将所有的线程放到list集合中，判断集合是否还有元素，方法二：设置计数器进行- - 操作；方法三：使用java.util.concurrent.CountDownLatch代替MyCountDown,用await()方法代替while(true){...}

## 11、进程和线程的区别

答：线程是指进程内的一个执行单元,也是进程内的可调度实体.与进程的区别:

(1)地址空间:进程内的一个执行单元;进程至少有一个线程;它们共享进程的地址空间;而进程有自己独立的地址空间;

(2)资源拥有:进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源

(3)线程是处理器调度的基本单位,但进程不是.

## 13、线程的几种状态

答：新建，就绪，运行，阻塞，死亡；

新建：当使用new操作创建一个线程

就绪：调用了start（），线程分配cpu以外的全部资源，等待获得cpu调度

运行：线程获得cpu，开始真正执行run()方法

阻塞：正在运行的线程没有运行结束，暂时让出CPU，这时其他处于就绪状态的线程就可以获得CPU时间，进入运行状态。有以下的几种原因：

1、线程调用sleep()方法

2、线程调用一个在I/O上被阻塞的操作，即该操作在输入输出操作完成之前不会返回到它的调用者；

3、线程试图得到一个锁，而该锁正被其他线程持有；

4、线程在等待某个触发条件；

死亡：正常退出或者一个异常结束stop（）、interrupt或者设置标志位

## 15、HashMap 是否线程安全，为何不安全。ConcurrentHashMap，线程安全，为何安全。底层实现是怎么样的。

答：不是线程安全地。当多个线程同时操作同一个数组位置的时候，也都会先取得现在状态下该位置存储的头结点，然后各自去进行计算操作，之后再把结果写会到该数组位置去，其实写回的时候可能其他的线程已经就把这个位置给修改过了，就会覆盖其他线程的修改。

## 16、J.U.C下的常见类的使用。 ThreadPool的深入考察； BlockingQueue的使用。（take，poll的区别，put，offer的区别）；原子类的实现。

