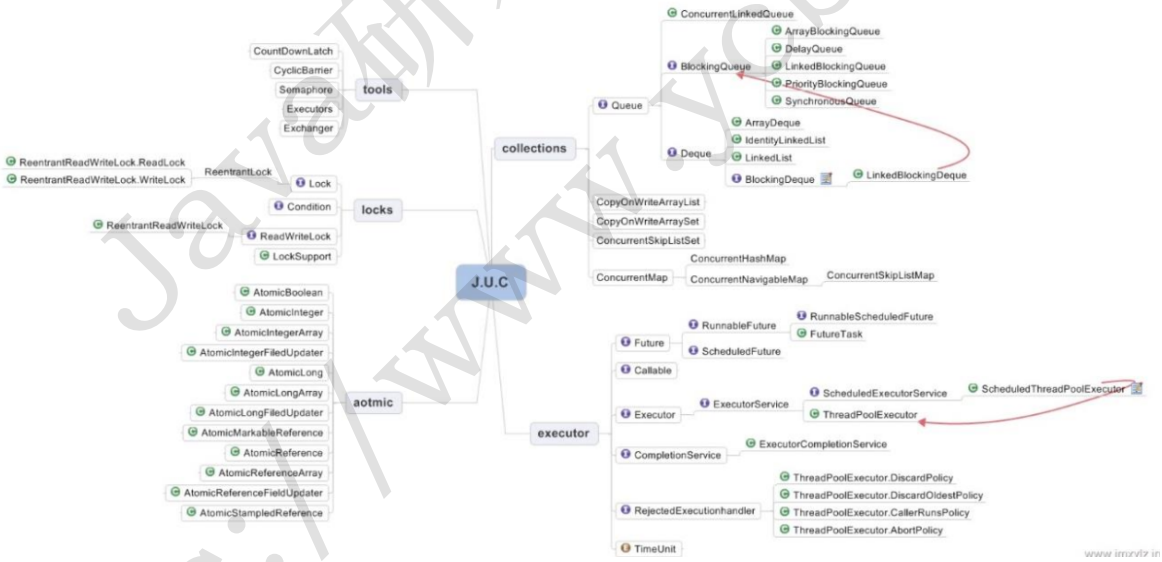
多线程&并发面试题

JAVA 并发知识库



1、Java中实现多线程有几种方法

继承Thread类；
实现Runnable接口；
实现Callable接口通过FutureTask包装器来创建Thread线程；
使用ExecutorService、Callable、Future实现有返回结果的多线程（也就是使用了ExecutorService来管理前面的三种方式）。

2、继承 Thread 类

Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。 启动线程的唯一方法就是通过 Thread 类的 start()实例方法。 start()方法是一个 native 方法，它将启动一个新线程，并执行 run()方法。

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("MyThread.run()");
    }
}
MyThread myThread1 = new MyThread();
myThread1.start();
```

3、实现 Runnable 接口。

如果自己的类已经 extends 另一个类，就无法直接 extends Thread，此时，可以实现一个 Runnable 接口。

```
public class MyThread extends OtherClass implements Runnable {
    public void run() {
        System.out.println("MyThread.run()");
    }
}
//启动 MyThread，需要首先实例化一个 Thread，并传入自己的 MyThread 实例：
MyThread myThread = new MyThread();
Thread thread = new Thread(myThread);
thread.start();
//事实上，当传入一个 Runnable target 参数给 Thread 后， Thread 的 run()方法就会调用
target.run()
public void run() {
    if (target != null) {
        target.run();
    }
}
```

4、ExecutorService、 Callable、 Future 有返回值线程

有返回值的任务必须实现 Callable 接口，类似的，无返回值的任务必须 Runnable 接口。执行 Callable 任务后，可以获取一个 Future 的对象，在该对象上调用 get 就可以获取到 Callable 任务返回的 Object 了，再结合线程池接口 ExecutorService 就可以实现传说中有返回结果的多线程了。

```
//创建一个线程池
ExecutorService pool = Executors.newFixedThreadPool(taskSize);
// 创建多个有返回值的任务
List<Future> list = new ArrayList<Future>();
for (int i = 0; i < taskSize; i++) {
    Callable c = new MyCallable(i + " ");
// 执行任务并获取 Future 对象
    Future f = pool.submit(c);
    list.add(f);
}
// 关闭线程池
pool.shutdown();
// 获取所有并发任务的运行结果
for (Future f : list) {
// 从 Future 对象上获取任务的返回值，并输出到控制台
    System.out.println("res：  " + f.get().toString());
```

```
}
```

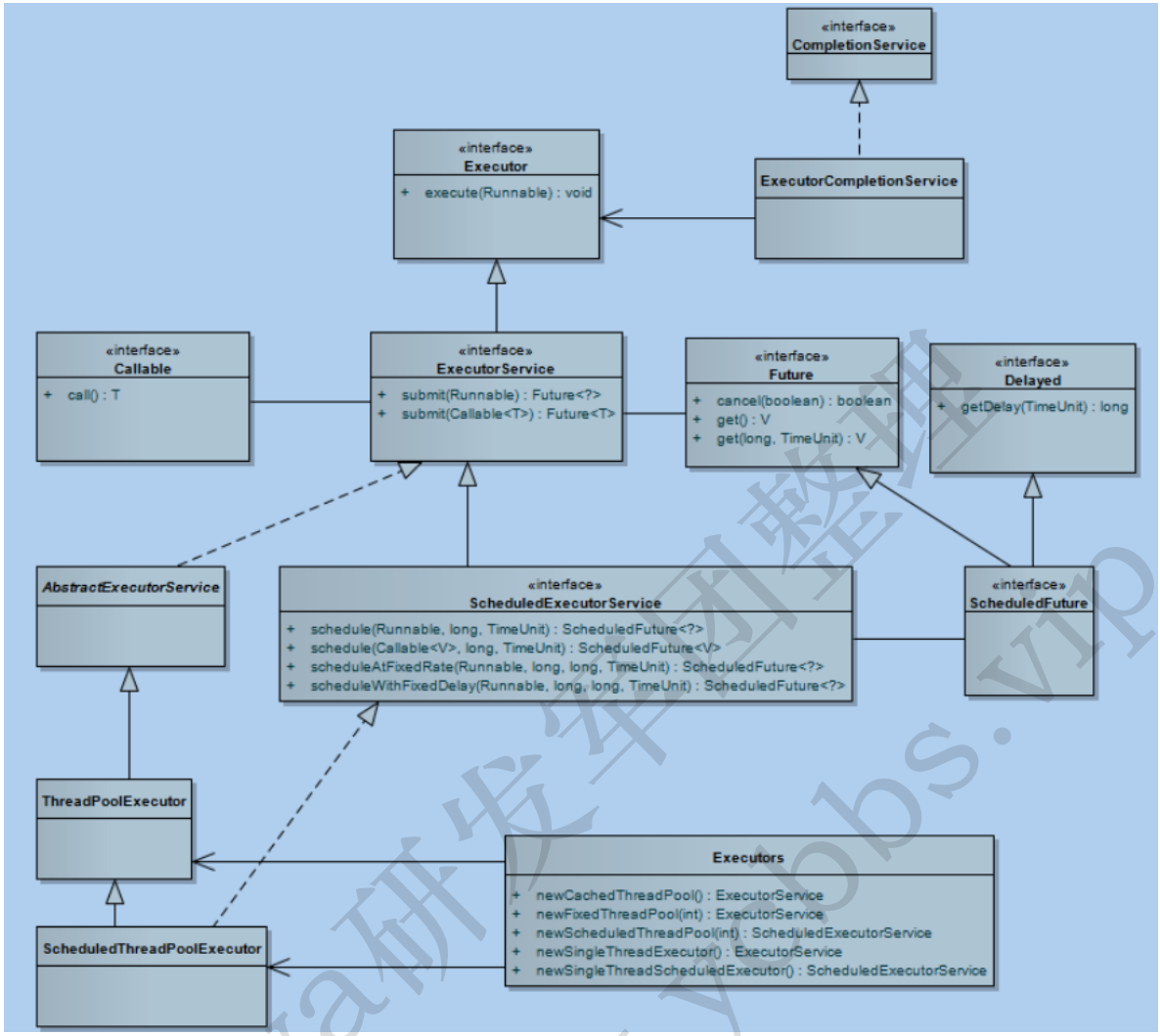
5、基于线程池的方式

线程和数据库连接这些资源都是非常宝贵的资源。那么每次需要的时候创建，不需要的时候销毁，是非常浪费资源的。那么我们就可以使用缓存的策略，也就是使用线程池。

```
// 创建线程池
ExecutorService threadPool = Executors.newFixedThreadPool(10);
while(true) {
    threadPool.execute(new Runnable() { // 提交多个线程任务，并执行
                        @Override
                        public void run() {
                            System.out.println(Thread.currentThread().getName()
+ " is running ..");
                            try {
                                Thread.sleep(3000);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                    });
    }
}
```

6、4 种线程池

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。



newCachedThreadPool

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。 调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。 因此，长时间保持空闲的线程池不会使用任何资源。

newFixedThreadPool

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

newScheduledThreadPool

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

```
ScheduledExecutorService scheduledThreadPool=
Executors.newScheduledThreadPool(3);
scheduledThreadPool.schedule(newRunnable(){
    @Override
    public void run() {
        System.out.println("延迟三秒");
    }
}, 3, TimeUnit.SECONDS);
scheduledThreadPool.scheduleAtFixedRate(newRunnable(){
    @Override
    public void run() {
        System.out.println("延迟 1 秒后每三秒执行一次");
    }
},1,3,TimeUnit.SECONDS);
```

newSingleThreadExecutor

Executors.newSingleThreadExecutor()返回一个线程池（这个线程池只有一个线程），这个线程池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去！

7、如何停止一个正在运行的线程

1、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，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
3、使用interrupt方法中断线程。

```
class MyThread extends Thread {
    volatile boolean stop = false;
        public void run() {
            while (!stop) {
                System.out.println(getName() + " is running");
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("week up from blcok...");
                    stop = true; // 在异常处理代码中修改共享变量的状态
                }
        }
    System.out.println(getName() + " is exiting...");
    }
}

class InterruptThreadDemo3 {
    public static void main(String[] args) throws InterruptedException {
        MyThread m1 = new MyThread();
        System.out.println("Starting thread...");
        m1.start();
        Thread.sleep(3000);
        System.out.println("Interrupt thread...: " + m1.getName());
        m1.stop = true; // 设置共享变量为true
        m1.interrupt(); // 阻塞时退出阻塞状态
        Thread.sleep(3000); // 主线程休眠3秒以便观察线程m1的中断情况
        System.out.println("Stopping application...");
    }
}
```

8、notify()和notifyAll()有什么区别？

notify可能会导致死锁，而notifyAll则不会

任何时候只有一个线程可以获得锁，也就是说只有一个线程可以运行synchronized 中的代码
使用notifyall,可以唤醒

所有处于wait状态的线程，使其重新进入锁的争夺队列中，而notify只能唤醒一个。

wait() 应配合while循环使用，不应使用if，务必在wait()调用前后都检查条件，如果不满足，必须调用notify()唤醒另外的线程来处理，自己继续wait()直至条件满足再往下执行。

notify() 是对notifyAll()的一个优化，但它有很精确的应用场景，并且要求正确使用。不然可能导致死锁。正确的场景应该是 WaitSet中等待的是相同的条件，唤醒任一个都能正确处理接下来的事项，如果唤醒的线程无法正确处理，务必确保继续notify()下一个线程，并且自身需要重新回到WaitSet中.

9、sleep()和wait() 有什么区别？

1. 对于 sleep()方法，我们首先要知道该方法是属于 Thread 类中的。而 wait()方法，则是属于 Object 类中的。
2. sleep()方法导致了程序暂停执行指定的时间，让出 cpu 该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态
3. 在调用 sleep()方法的过程中， 线程不会释放对象锁。
4. 而当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用 notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

10、volatile 是什么?可以保证有序性吗?

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的,volatile关键字会强制将修改的值立即写入主存。

2）禁止进行指令重排序。
volatile 不是原子性操作
什么叫保证部分有序性?

当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

```
x = 2;       //语句1
y = 0;       //语句2
flag = true; //语句3
x = 4;       //语句4
y = -1;      //语句5
```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。
使用 Volatile 一般用于 状态标记量 和 单例模式的双检锁

11、Thread 类中的start() 和 run() 方法有什么区别？

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程 。

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

明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象 。

13、为什么wait和notify方法要在同步块中调用？

1. 只有在调用线程拥有某个对象的独占锁时，才能够调用该对象的wait(),notify()和notifyAll()方法。
2. 如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。
3. 还有一个原因是为了避免wait和notify之间产生竞态条件。

wait()方法强制当前线程释放对象锁。这意味着在调用某对象的wait()方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的wait()方法。

在调用对象的notify()和notifyAll()方法之前，调用线程必须已经得到该对象的锁。因此，必须在某个对象的同步方法或同步代码块中才能调用该对象的notify()或notifyAll()方法。

调用wait()方法的原因通常是，调用线程希望某个特殊的状态(或变量)被设置之后再继续执行。调用notify()或notifyAll()方法的原因通常是，调用线程希望告诉其他等待中的线程:"特殊状态已经被设置"。这个状态作为线程间通信的通道，它必须是一个可变的共享状态(或变量)。

14、Java中interrupted 和 isInterruptedd方法的区别？

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有可能被其它线程调用中断来改变 。

15、Java中synchronized 和 ReentrantLock 有什么不同？

相似点：
这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的.

区别：
这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而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。

Synchronized进过编译，会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器就减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止 。

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下3项：
1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。

2.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。

3.锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象 。

16、有三个线程T1,T2,T3,如何保证顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

实际上先启动三个线程中哪一个都行，
因为在每个线程的run方法中用join方法限定了三个线程的执行顺序

```
public class JoinTest2 {

 // 1.现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行

    public static void main(String[] args) {
        final Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("t1");
            }
        });
        final Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {

// 引用t1线程，等待t1线程执行完
                    t1.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } S
                ystem.out.println("t2");
            }
        });
        Thread t3 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
// 引用t2线程，等待t2线程执行完
                    t2.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } S
                ystem.out.println("t3");
            }
        });
        t3.start();//这里三个线程的启动顺序可以任意，大家可以试下！
```

```
        t2.start();
        t1.start();
    }
}
```

17、SynchronizedMap和ConcurrentHashMap有什么区别？

SynchronizedMap()和Hashtable一样，实现上在调用map所有方法时，都对整个map进行同步。而ConcurrentHashMap的实现却更加精细，它对map中的所有桶加了锁。所以，只要有一个线程访问map，其他线程就无法进入map，而如果一个线程在访问ConcurrentHashMap某个桶时，其他线程，仍然可以对map执行某些操作。

所以，ConcurrentHashMap在性能以及安全性方面，明显比Collections.synchronizedMap()更加有优势。同时，同步操作精确控制到桶，这样，即使在遍历map时，如果其他线程试图对map进行数据修改，也不会抛出ConcurrentModificationException 。

18、什么是线程安全

线程安全就是说多线程访问同一代码，不会产生不确定的结果。

在多线程环境中，当各线程不共享数据的时候，即都是私有（private）成员，那么一定是线程安全的。但这种情况并不多见，在多数情况下需要共享数据，这时就需要进行适当的同步控制了。

线程安全一般都涉及到synchronized， 就是一段代码同时只能有一个线程来操作 不然中间过程可能会产生不可预制的结果。

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。

19、Thread类中的yield方法有什么作用？

Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。

20、Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

21、说一说自己对于 synchronized 关键字的了解

synchronized关键字解决的是多个线程之间访问资源的同步性，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。

另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。

如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized 效率低的原因。

庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销 。

22、说说自己是怎么使用 synchronized 关键字，在项目中用到了吗synchronized关键字最主要的三种使用方式

修饰实例方法: 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁
修饰静态方法: 也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态synchronized 方法占用的锁是当前实例对象锁。**
修饰代码块: 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。
总结： synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓存功能

23、什么是线程安全？Vector是一个线程安全类吗？

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。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分 成两组，线程安全和非线程安全的。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。

24、volatile关键字的作用？

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：
保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

禁止进行指令重排序。

1. volatile本质是在告诉jvm当前变量在寄存器（工作内存）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。
2. volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的。
3. volatile仅能实现变量的修改可见性，并不能保证原子性；synchronized则可以保证变量的修改可见性和原子性。
4. volatile不会造成线程的阻塞；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；synchronized标记的变量可以被编译器优化

25、简述一下你对线程池的理解

如果问到了这样的问题，可以展开的说一下线程池如何用、线程池的好处、线程池的启动策略）合理利用线程池能够带来三个好处。

第一：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。
第二：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。
第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，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的分配，调优和监控

26、线程生命周期(状态)

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换

27、新建状态（NEW）

当程序使用 new 关键字创建了一个线程之后，该线程就处于新建状态，此时仅由 JVM 为其分配内存，并初始化其成员变量的值

28、就绪状态（RUNNABLE）

当线程对象调用了 start()方法之后，该线程处于就绪状态。Java 虚拟机会为其创建方法调用栈和程序计数器，等待调度运行。

29、运行状态（RUNNING）

如果处于就绪状态的线程获得了 CPU，开始执行 run()方法的线程执行体，则该线程处于运行状态。

30、阻塞状态（BLOCKED）

阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得 cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

等待阻塞（o.wait->等待对列）：
运行(running)的线程执行 o.wait()方法， JVM 会把该线程放入等待队列(waitting queue)中。

同步阻塞(lock->锁池)
运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池(lock pool)中。

其他阻塞(sleep/join)
运行(running)的线程执行 Thread.sleep(long ms)或 t.join()方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、 join()等待线程终止或者超时、或者 I/O处理完毕时，线程重新转入可运行(runnable)状态。

31、线程死亡（DEAD）

线程会以下面三种方式结束，结束后就是死亡状态。
正常结束

1. run()或 call()方法执行完成，线程正常结束。异常结束
2. 线程抛出一个未捕获的 Exception 或 Error。调用 stop
3. 直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁，不推荐使用。

32、终止线程 4 种方式

正常运行结束

程序运行结束，线程自动结束。

使用退出标志退出线程

一般 run()方法执行完，线程就会正常结束，然而，常常有些线程是伺服线程。它们需要长时间的运行，只有在外部某些条件满足的情况下，才能关闭这些线程。使用一个变量来控制循环，例如：最直接的方法就是设一个 boolean 类型的标志，并通过设置这个标志为 true 或 false 来控制 while循环是否退出，代码示例 ：

```
public class ThreadSafe extends Thread {
    public volatile boolean exit = false;
    public void run() {
        while (!exit){
//do something
        }
    }
}
```

定义了一个退出标志 exit，当 exit 为 true 时， while 循环退出， exit 的默认值为 false.在定义 exit 时，使用了一个 Java 关键字 volatile，这个关键字的目的是使 exit 同步，也就是说在同一时刻只能由一个线程来修改 exit 的值。

Interrupt 方法结束线程

使用 interrupt()方法来中断线程有两种情况：

1.线程处于阻塞状态： 如使用了 sleep,同步锁的 wait,socket 中的 receiver,accept 等方法时，会使线程处于阻塞状态。当调用线程的 interrupt()方法时，会抛出 InterruptException 异常。阻塞中的那个方法抛出这个异常，通过代码捕获该异常，然后 break 跳出循环状态，从而让我们有机会结束这个线程的执行。 通常很多人认为只要调用 interrupt 方法线程就会结束，实际上是错的， 一定要先捕获 InterruptedException 异常之后通过 break 来跳出循环，才能正常结束 run 方法。

2.线程未处于阻塞状态： 使用 isInterrupted()判断线程的中断标志来退出循环。当使用interrupt()方法时，中断标志就会置 true，和使用自定义的标志来控制循环是一样的道理。

```
public class ThreadSafe extends Thread {
    public void run() {
        while (!isInterrupted()){ //非阻塞过程中通过判断中断标志来退出
            try{
                Thread.sleep(5*1000);//阻塞过程捕获中断异常来退出
            }catch(InterruptedException e){
                e.printStackTrace();
                break;//捕获到异常之后，执行 break 跳出循环
            }
        }
    }
}
```

stop 方法终止线程（线程不安全）

程序中可以直接使用 thread.stop()来强行终止线程，但是 stop 方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，不安全主要是：thread.stop()调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子线程所持有的所有锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。因
此，并不推荐使用 stop 方法来终止线程。

33、start 与 run 区别

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。
2. 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。
3. 方法 run()称为线程体，它包含了要执行的这个线程的内容，线程就进入了运行状态，开始运行 run 函数当中的代码。 Run 方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

34、JAVA 后台线程

1. 定义：守护线程--也称"服务线程"， 他是后台线程， 它有一个特性，即为用户线程 提供 公共服务， 在没有用户线程可服务时会自动离开。
2. 优先级：守护线程的优先级比较低，用于为系统中的其它对象和线程提供服务。
3. 设置：通过 setDaemon(true)来设置线程为"守护线程"；将一个用户线程设置为守护线程的方式是在 线程对象创建 之前 用线程对象的 setDaemon 方法。
4. 在 Daemon 线程中产生的新线程也是 Daemon 的。
5. 线程则是 JVM 级别的，以 Tomcat 为例，如果你在 Web 应用中启动一个线程，这个线程的生命周期并不会和 Web 应用程序保持同步。也就是说，即使你停止了 Web 应用，这个线程依旧是活跃的。
6. example: 垃圾回收线程就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread, 程序就不会再产生垃圾，垃圾回收器也就无事可做， 所以当垃圾回收线程是 JVM 上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。
7. 生命周期：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统"同生共死"。当 JVM 中所有的线程都是守护线程的时候， JVM 就可以退出了；如果还有一个或以上的非守护线程则 JVM 不会退出

35、什么是乐观锁

乐观锁是一种乐观思想，即认为读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读-比较-写的操作。

java 中的乐观锁基本都是通过 CAS 操作实现的， CAS 是一种更新的原子操作， 比较当前值跟传入值是否一样，一样则更新，否则失败。

36、什么是悲观锁

悲观锁是就是悲观思想，即认为写多，遇到并发写的可能性高，每次去拿数据的时候都认为别人会修改，所以每次在读写数据的时候都会上锁，这样别人想读写这个数据就会 block 直到拿到锁。java中的悲观锁就是Synchronized,AQS框架下的锁则是先尝试cas乐观锁去获取锁，获取不到，才会转换为悲观锁，如 RetreenLock。

37、什么是自旋锁

自旋锁原理非常简单， 如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。

线程自旋是需要消耗 cup 的，说白了就是让 cup 在做无用功，如果一直获取不到锁，那线程也不能一直占用 cup 自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

自旋锁的优缺点
自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用 cpu 做无用功，占着 XX 不 XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要 cup 的线程又不能获取到 cpu，造成 cpu 的浪费。所以这种情况下我们要关闭自旋锁；

自旋锁时间阈值（1.6 引入了适应性自旋锁）
自旋锁的目的是为了占着 CPU 的资源不释放，等到获取到锁立即进行处理。但是如何去选择自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用 CPU 资源，进而会影响整体系统的性能。因此自旋的周期选的额外重要！

JVM 对于自旋周期的选择， jdk1.5 这个限度是一定的写死的， 在 1.6 引入了适应性自旋锁，适应性自旋锁意味着自旋的时间不在是固定的了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定，基本认为一个线程上下文切换的时间是最佳的一个时间，同时 JVM 还针对当前 CPU 的负荷情况做了较多的优化， 如果平均负载小于 CPUs 则一直自旋， 如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞， 如果正在自旋的线程发现 Owner 发生了变化则延迟自旋时间（自旋计数）或进入

阻塞， 如果 CPU 处于节电模式则停止自旋， 自旋时间的最坏情况是 CPU的存储延迟（CPU A 存储了一个数据，到 CPU B 得知这个数据直接的时间差） ， 自旋时会适当放弃线程优先级之间的差异。

自旋锁的开启

JDK1.6 中-XX:+UseSpinning 开启；
-XX:PreBlockSpin=10 为自旋次数；
JDK1.7 后，去掉此参数，由 jvm 控制；

38、Synchronized 同步锁

synchronized 它可以把任意一个非 NULL 的对象当作锁。 他属于独占式的悲观锁，同时属于可重入锁。 Synchronized 作用范围**

1. 作用于方法时，锁住的是对象的实例(this)；
2. 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带 PermGen（jdk1.8 则是 metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；
3. synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。 它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

Synchronized 核心组件

1) Wait Set：哪些调用 wait 方法被阻塞的线程被放置在这里；
2) Contention List： 竞争队列，所有请求锁的线程首先被放在这个竞争队列中；
3) Entry List： Contention List 中那些有资格成为候选资源的线程被移动到 Entry List 中；
4) OnDeck：任意时刻， 最多只有一个线程正在竞争锁资源，该线程被成为 OnDeck；
5) Owner：当前已经获取到所资源的线程被称为 Owner；
6) !Owner：当前释放锁的线程。

Synchronized 实现

1. JVM 每次从队列的尾部取出一个数据用于锁竞争候选者（OnDeck），但是并发情况下，ContentionList 会被大量的并发线程进行 CAS 访问，为了降低对尾部元素的竞争， JVM 会将一部分线程移动到 EntryList 中作为候选竞争线程。
2. Owner 线程会在 unlock 时，将 ContentionList 中的部分线程迁移到 EntryList 中，并指定 EntryList 中的某个线程为 OnDeck 线程（一般是最先进去的那个线程）。
3. Owner 线程并不直接把锁传递给 OnDeck 线程，而是把锁竞争的权利交给 OnDeck，OnDeck 需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在 JVM 中，也把这种选择行为称之为“竞争切换”。
4. OnDeck 线程获取到锁资源后会变为 Owner 线程，而没有得到锁资源的仍然停留在 EntryList 中。如果 Owner 线程被 wait 方法阻塞，则转移到 WaitSet 队列中，直到某个时刻通过 notify 或者 notifyAll 唤醒，会重新进去 EntryList 中。
5. 处于 ContentionList、 EntryList、 WaitSet 中的线程都处于阻塞状态，该阻塞是由操作系统来完成的（Linux 内核下采用 pthread_mutex_lock 内核函数实现的）。
6. Synchronized 是非公平锁。 Synchronized 在线程进入 ContentionList 时， 等待的线程会先尝试自旋获取锁，如果获取不到就进入 ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占 OnDeck 线程的锁资源。
   参考： https://blog.csdn.net/zqz_zqz/article/details/70233767
7. 每个对象都有个 monitor 对象， 加锁就是在竞争 monitor 对象，代码块加锁是在前后分别加上 monitorenter 和 monitorexit 指令来实现的，方法加锁是通过一个标记位来判断的
8. synchronized 是一个重量级操作，需要调用操作系统相关接口，性能是低效的，有可能给线程加锁消耗的时间比有用操作消耗的时间更多。

9. Java1.6， synchronized 进行了很多的优化， 有适应自旋、锁消除、锁粗化、轻量级锁及偏向锁等，效率有了本质上的提高。在之后推出的 Java1.7 与 1.8 中，均对该关键字的实现机理做了优化。引入了偏向锁和轻量级锁。都是在对象头中有标记位，不需要经过操作系统加锁。
10. 锁可以从偏向锁升级到轻量级锁，再升级到重量级锁。这种升级过程叫做锁膨胀；
11. JDK 1.6 中默认是开启偏向锁和轻量级锁，可以通过-XX:-UseBiasedLocking 来禁用偏向锁。

39、ReentrantLock

ReentantLock 继承接口 Lock 并实现了接口中定义的方法， 他是一种可重入锁， 除了能完成 synchronized 所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。
Lock 接口的主要方法

void lock(): 执行此方法时, 如果锁处于空闲状态, 当前线程将获取到锁. 相反, 如果锁已经被其他线程持有, 将禁用当前线程, 直到当前线程获取到锁.

boolean tryLock()： 如果锁可用, 则获取锁, 并立即返回 true, 否则返回 false. 该方法和 lock()的区别在于, tryLock()只是"试图"获取锁, 如果锁不可用, 不会导致当前线程被禁用, 当前线程仍然继续往下执行代码. 而 lock()方法则是一定要获取到锁, 如果锁不可用, 就一直等待, 在未获得锁之前,当前线程并不继续向下执行.

void unlock()：执行此方法时, 当前线程将释放持有的锁. 锁只能由持有者释放, 如果线程并不持有锁, 却执行该方法, 可能导致异常的发生.

Condition newCondition()： 条件对象，获取等待通知组件。该组件和当前的锁绑定，当前线程只有获取了锁，才能调用该组件的 await()方法，而调用后，当前线程将缩放锁。

getHoldCount() ： 查询当前线程保持此锁的次数，也就是执行此线程执行 lock 方法的次数。

getQueueLength（ ） ： 返回正等待获取此锁的线程估计数，比如启动 10 个线程， 1 个线程获得锁，此时返回的是 9

getWaitQueueLength： （Condition condition）返回等待与此锁相关的给定条件的线程估计数。比如 10 个线程，用同一个 condition 对象，并且此时这 10 个线程都执行了 condition 对象的 await 方法，那么此时执行此方法返回 10

hasWaiters(Condition condition)： 查询是否有线程等待与此锁有关的给定条件(condition)，对于指定 contidion 对象，有多少线程执行了 condition.await 方法

hasQueuedThread(Thread thread)： 查询给定线程是否等待获取此锁

hasQueuedThreads()： 是否有线程等待此锁

isFair()： 该锁是否公平锁

isHeldByCurrentThread()： 当前线程是否保持锁锁定，线程的执行 lock 方法的前后分别是 false 和 true

isLock()： 此锁是否有任意线程占用

lockInterruptibly（ ） ： 如果当前线程未被中断，获取锁

tryLock（ ） ： 尝试获得锁，仅在调用时锁未被线程占用，获得锁

tryLock(long timeout TimeUnit unit)： 如果锁在给定等待时间内没有被另一个线程保持，则获取该锁。

非公平锁
JVM 按随机、就近原则分配锁的机制则称为不公平锁， ReentrantLock 在构造函数中提供了是否公平锁的初始化方式，默认为非公平锁。 非公平锁实际执行的效率要远远超出公平锁，除非程序有特殊需要，否则最常用非公平锁的分配机制。

公平锁
公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁，ReentrantLock 在构造函数中提供了是否公平锁的初始化方式来定义公平锁。

40、Condition 类和 Object 类锁方法区别区别

1. Condition 类的 awiat 方法和 Object 类的 wait 方法等效
2. Condition 类的 signal 方法和 Object 类的 notify 方法等效
3. Condition 类的 signalAll 方法和 Object 类的 notifyAll 方法等效
4. ReentrantLock 类可以唤醒指定条件的线程，而 object 的唤醒是随机的

41、tryLock 和 lock 和 lockInterruptibly 的区别

1. tryLock 能获得锁就返回 true，不能就立即返回 false， tryLock(long timeout,TimeUnit unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回 false
2. lock 能获得锁就返回 true，不能的话一直等待获得锁
3. lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程，lock 不会抛出异常，而 lockInterruptibly 会抛出异常。

42、Semaphore 信号量

Semaphore 是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做完自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。 Semaphore 可以用来构建一些对象池，资源池之类的， 比如数据库连接池

实现互斥锁（计数器为 1）
我们也可以创建计数为 1 的 Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态。
代码实现

```
// 创建一个计数阈值为 5 的信号量对象
// 只能 5 个线程同时访问
Semaphore semp = new Semaphore(5);
try { // 申请许可
        semp.acquire();
        try {
    // 业务逻辑
        } catch (Exception e) {
        } finally {
    // 释放许可
            semp.release();
        }
```

```
    } catch (InterruptedException e) {
    }
```

43、Semaphore 与 ReentrantLock 区别

Semaphore 基本能完成 ReentrantLock 的所有工作，使用方法也与之类似，通过 acquire()与 release()方法来获得和释放临界资源。经实测， Semaphone.acquire()方法默认为可响应中断锁，与 ReentrantLock.lockInterruptibly()作用效果一致，也就是说在等待临界资源的过程中可以被 Thread.interrupt()方法中断。

此外， Semaphore 也实现了可轮询的锁请求与定时锁的功能，除了方法名 tryAcquire 与 tryLock 不同，其使用方法与 ReentrantLock 几乎一致。 Semaphore 也提供了公平与非公平锁的机制，也可在构造函数中进行设定。

Semaphore 的锁释放操作也由手动进行，因此与 ReentrantLock 一样，为避免线程因抛出异常而无法正常释放锁的情况发生，释放锁的操作也必须在 finally 代码块中完成。

44、可重入锁（递归锁）

本文里面讲的是广义上的可重入锁，而不是单指 JAVA 下的 ReentrantLock。 可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。在 JAVA 环境下 ReentrantLock 和 synchronized 都是 可重入锁。

45、公平锁与非公平锁

公平锁（Fair）
加锁前检查是否有排队等待的线程，优先排队等待的线程，先来先得
非公平锁（Nonfair）
加锁时不考虑排队等待问题，直接尝试获取锁，获取不到自动到队尾等待

1. 非公平锁性能比公平锁高 5~10 倍，因为公平锁需要在多核的情况下维护一个队列
2. Java 中的 synchronized 是非公平锁， ReentrantLock 默认的 lock()方法采用的是非公平锁。

46、ReadWriteLock 读写锁

为了提高性能， Java 提供了读写锁，在读的地方使用读锁，在写的地方使用写锁，灵活控制，如果没有写锁的情况下，读是无阻塞的,在一定程度上提高了程序的执行效率。 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。

读锁
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

写锁
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！

Java 中读写锁有个接口 java.util.concurrent.locks.ReadWriteLock ，也有具体的实现 ReentrantReadWriteLock。

47、共享锁和独占锁

java 并发包提供的加锁模式分为独占锁和共享锁。

独占锁
独占锁模式下，每次只能有一个线程能持有锁， ReentrantLock 就是以独占方式实现的互斥锁。
独占锁是一种悲观保守的加锁策略，它避免了读/读冲突，如果某个只读线程获取锁，则其他读线程都只能等待，这种情况下就限制了不必要的并发性，因为读操作并不会影响数据的一致性。

共享锁
共享锁则允许多个线程同时获取锁，并发访问 共享资源，如： ReadWriteLock。 共享锁则是一种乐观锁，它放宽了加锁策略，允许多个执行读操作的线程同时访问共享资源。

1. AQS 的内部类 Node 定义了两个常量 SHARED 和 EXCLUSIVE，他们分别标识 AQS 队列中等待线程的锁获取模式。
2. java 的并发包中提供了 ReadWriteLock，读-写锁。它允许一个资源可以被多个读操作访问，或者被一个 写操作访问，但两者不能同时进行。

48、重量级锁（Mutex Lock）

Synchronized 是通过对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的 Mutex Lock 来实现的。

而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么Synchronized 效率低的原因。

因此， 这种依赖于操作系统 Mutex Lock 所实现的锁我们称之为“重量级锁”。 JDK 中对 Synchronized 做的种种优化，其核心都是为了减少这种重量级锁的使用。

JDK1.6 以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和“偏向锁”。

49、轻量级锁

锁的状态总共有四种：无锁状态、偏向锁、轻量级锁和重量级锁。
锁升级
随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁（但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级）。

“轻量级” 是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。

在解释轻量级锁的执行过程之前， 先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁

50、偏向锁

Hotspot 的作者经过以往的研究发现大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得。 偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏护。

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次 CAS 原子指令， 而偏向锁只需要在置换ThreadID 的时候依赖一次 CAS 原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的 CAS 原子指令的性能消耗）。

上面说过， 轻量级锁是为了在线程交替执行同步块时提高性能， 而偏向锁则是在只有一个线程执行同步块时进一步提高性能

51、分段锁

分段锁也并非一种实际的锁，而是一种思想 ConcurrentHashMap 是学习分段锁的最好实践

52、锁优化

减少锁持有时间
只用在有线程安全要求的程序上加锁

减小锁粒度
将大对象（这个对象可能会被很多线程访问），拆成小对象，大大增加并行度，降低锁竞争。降低了锁的竞争，偏向锁，轻量级锁成功率才会提高。最最典型的减小锁粒度的案例就是 ConcurrentHashMap。

锁分离
最常见的锁分离就是读写锁 ReadWriteLock，根据功能进行分离成读锁和写锁，这样读读不互斥，读写互斥，写写互斥，即保证了线程安全，又提高了性能，具体也请查看[高并发 Java 五] JDK 并发包 1。读写分离思想可以延伸，只要操作互不影响，锁就可以分离。比如 LinkedBlockingQueue 从头部取出，从尾部放数据

锁粗化
通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完公共资源后，应该立即释放锁。但是，凡事都有一个度， 如果对同一个锁不停的进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化 。

锁消除
锁消除是在编译器级别的事情。 在即时编译器时，如果发现不可能被共享的对象，则可以消除这些对象的锁操作，多数是因为程序员编码不规范引起。
参考： https://www.jianshu.com/p/39628e1180a9

53、线程基本方法

线程相关的基本方法有 wait， notify， notifyAll， sleep， join， yield 等。

54、线程等待（wait）

调用该方法的线程进入 WAITING 状态，只有等待另外线程的通知或被中断才会返回，需要注意的是调用 wait()方法后， 会释放对象的锁。因此， wait 方法一般用在同步方法或同步代码块中。

55、线程睡眠（sleep）

sleep 导致当前线程休眠，与 wait 方法不同的是 sleep 不会释放当前占有的锁,sleep(long)会导致线程进入 TIMED-WATING 状态，而 wait()方法会导致当前线程进入 WATING 状态

56、线程让步（yield）

yield 会使当前线程让出 CPU 执行时间片，与其他线程一起重新竞争 CPU 时间片。一般情况下，优先级高的线程有更大的可能性成功竞争得到 CPU 时间片， 但这又不是绝对的，有的操作系统对线程优先级并不敏感。

57、线程中断（interrupt）

中断一个线程，其本意是给这个线程一个通知信号，会影响这个线程内部的一个中断标识位。 这个线程本身并不会因此而改变状态(如阻塞，终止等)。

1. 调用 interrupt()方法并不会中断一个正在运行的线程。也就是说处于 Running 状态的线程并不会因为被中断而被终止，仅仅改变了内部维护的中断标识位而已。
2. 若调用 sleep()而使线程处于 TIMED-WATING 状态，这时调用 interrupt()方法，会抛出 InterruptedException,从而使线程提前结束 TIMED-WATING 状态。
3. 许多声明抛出 InterruptedException 的方法(如 Thread.sleep(long mills 方法))，抛出异常前，都会清除中断标识位，所以抛出异常后，调用 isInterrupted()方法将会返回 false。
4. 中断状态是线程固有的一个标识位，可以通过此标识位安全的终止线程。比如,你想终止一个线程 thread 的时候，可以调用 thread.interrupt()方法，在线程的 run 方法内部可以根据 thread.isInterrupted()的值来优雅的终止线程。

58、Join 等待其他线程终止

join() 方法，等待其他线程终止，在当前线程中调用一个线程的 join() 方法，则当前线程转为阻塞状态，回到另一个线程结束，当前线程再由阻塞状态变为就绪状态，等待 cpu 的宠幸。

59、为什么要用 join()方法？

很多情况下，主线程生成并启动了子线程，需要用到子线程返回的结果，也就是需要主线程需要在子线程结束后再结束，这时候就要用到 join() 方法 。

```
System.out.println(Thread.currentThread().getName() + "线程运行开始!");
Thread6 thread1 = new Thread6();
thread1.setName("线程 B");
thread1.join();
System.out.println("这时 thread1 执行完毕之后才能执行主线程");
```

60、线程唤醒（notify）

Object 类中的 notify() 方法， 唤醒在此对象监视器上等待的单个线程，如果所有线程都在此对象上等待，则会选择唤醒其中一个线程，选择是任意的，并在对实现做出决定时发生，线程通过调用其中一个 wait() 方法，在对象的监视器上等待， 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程，被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争。类似的方法还有 notifyAll() ，唤醒再次监视器上等待的所有线程。

61、线程其他方法

1. sleep()：强迫一个线程睡眠N毫秒。
2. isAlive()： 判断一个线程是否存活。
3. join()： 等待线程终止。
4. activeCount()： 程序中活跃的线程数。
5. enumerate()： 枚举程序中的线程。
6. currentThread()： 得到当前线程。
7. isDaemon()： 一个线程是否为守护线程。
8. setDaemon()： 设置一个线程为守护线程。 (用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束)
9. setName()： 为线程设置一个名称。
10. wait()： 强迫一个线程等待。
    11.notify()： 通知一个线程继续运行。
11. setPriority()： 设置一个线程的优先级。
12. getPriority(): ：获得一个线程的优先级。

62、进程

（有时候也称做任务）是指一个程序运行的实例。在 Linux 系统中，线程就是能并行运行并且与他们的父进程（创建他们的进程）共享同一地址空间（一段内存区域）和其他资源的轻量级的进程。

63、上下文

是指某一时间点 CPU 寄存器和程序计数器的内容。

64、寄存器

是 CPU 内部的数量较少但是速度很快的内存（与之对应的是 CPU 外部相对较慢的 RAM 主内存）。寄存器通过对常用值（通常是运算的中间值）的快速访问来提高计算机程序运行的速度。

65、程序计数器

是一个专用的寄存器， 用于表明指令序列中 CPU 正在执行的位置，存的值为正在执行的指令的位置或者下一个将要被执行的指令的位置，具体依赖于特定的系统。

66、PCB-“切换桢”

上下文切换可以认为是内核（操作系统的核心）在 CPU 上对于进程（包括线程）进行切换，上下文切换过程中的信息是保存在进程控制块（PCB, process control block）中的。 PCB 还经常被称作“切换桢”（switchframe）。 信息会一直保存到 CPU 的内存中，直到他们被再次使用。

67、上下文切换的活动

1.挂起一个进程，将这个进程在 CPU 中的状态（上下文）存储于内存中的某处。

2.在内存中检索下一个进程的上下文并将其在 CPU 的寄存器中恢复。

3.跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程在程序中。

68、引起线程上下文切换的原因

1. 当前执行任务的时间片用完之后，系统 CPU 正常调度下一个任务；
2. 当前执行任务碰到 IO 阻塞，调度器将此任务挂起，继续下一任务；
3. 多个任务抢占锁资源，当前任务没有抢到锁资源，被调度器挂起，继续下一任务；
4. 用户代码挂起当前任务，让出 CPU 时间；
5. 硬件中断；

69、同步锁

当多个线程同时访问同一个数据时，很容易出现问题。为了避免这种情况出现，我们要保证线程同步互斥，就是指并发执行的多个线程，在同一时间内只允许一个线程访问共享数据。 Java 中可以使用 synchronized 关键字来取得一个对象的同步锁。

70、死锁

何为死锁，就是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。

71、线程池原理

线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量超出数量的线程排队等候，等其它线程执行完毕，再从队列中取出任务来执行。 他的主要特点为： 线程复用； 控制最大并发数； 管理线程。

72、线程复

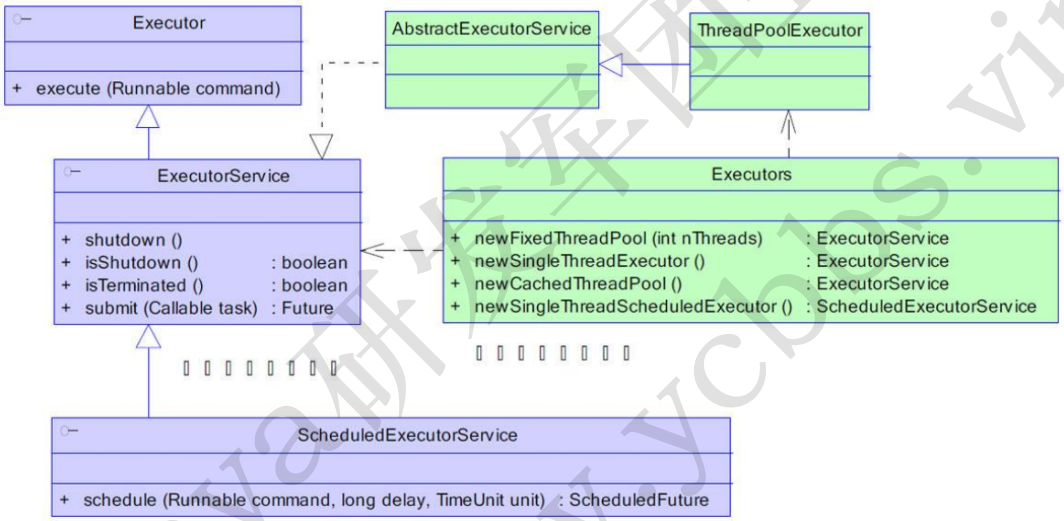
每一个 Thread 的类都有一个 start 方法。 当调用 start 启动线程时 Java 虚拟机会调用该类的 run 方法。 那么该类的 run() 方法中就是调用了 Runnable 对象的 run() 方法。 我们可以继承重写 Thread 类，在其 start 方法中添加不断循环调用传递过来的 Runnable 对象。 这就是线程池的实现原理。 循环方法中不断获取 Runnable 是用 Queue 实现的，在获取下一个 Runnable 之前可以是阻塞的 。

73、线程池的组成

一般的线程池主要分为以下 4 个组成部分：

1. 线程池管理器：用于创建并管理线程池
2. 工作线程：线程池中的线程
3. 任务接口：每个任务必须实现的接口，用于工作线程调度其运行
4. 任务队列：用于存放待处理的任务，提供一种缓冲机制

> Java 中的线程池是通过 Executor 框架实现的，该框架中用到了 Executor， Executors，ExecutorService， ThreadPoolExecutor ， Callable 和 Future、 FutureTask 这几个类。



ThreadPoolExecutor 的构造方法如下：

```
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize, long
keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue) {

    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
    Executors.defaultThreadFactory(), defaultHandler);
}
```

1. corePoolSize：指定了线程池中的线程数量。
2. maximumPoolSize：指定了线程池中的最大线程数量。
3. keepAliveTime：当前线程池数量超过 corePoolSize 时，多余的空闲线程的存活时间，即多次时间内会被销毁。
4. unit： keepAliveTime 的单位。
5. workQueue：任务队列，被提交但尚未被执行的任务。
6. threadFactory：线程工厂，用于创建线程，一般用默认的即可。
7. handler：拒绝策略，当任务太多来不及处理，如何拒绝任务。

74、拒绝策略

线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列也已经排满了，再也塞不下新任务了。这时候我们就需要拒绝策略机制合理的处理这个问题。
JDK 内置的拒绝策略如下：

1. AbortPolicy ： 直接抛出异常，阻止系统正常运行。
2. CallerRunsPolicy ： 只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。
3. DiscardOldestPolicy ： 丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。
4. DiscardPolicy ： 该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢失，这是最好的一种方案。
   以上内置拒绝策略均实现了 RejectedExecutionHandler 接口，若以上策略仍无法满足实际需要，完全可以自己扩展 RejectedExecutionHandler 接口。

75、Java 线程池工作过程

1. 线程池刚创建时，里面没有一个线程。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：
   a) 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；
   b) 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；
   c) 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；
   d) 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。

76、JAVA 阻塞队列原理

阻塞队列，关键字是阻塞，先理解阻塞的含义，在阻塞队列中，线程阻塞有这样的两种情况：

1. 当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。
2. 当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。

阻塞队列的主要方法 ：

方法类型	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除	remove()	poll()	take()	poll(time,unit)
检查	element()	peek()	不可用	不可用

抛出异常：抛出一个异常；
特殊值：返回一个特殊值（null 或 false,视情况而定）
则塞：在成功操作之前，一直阻塞线程
超时：放弃前只在最大的时间内阻塞

插入操作

1： public abstract boolean add(E paramE)： 将指定元素插入此队列中（如果立即可行且不会违反容量限制），成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。如果该元素是 NULL，则会抛出 NullPointerException 异常。

2： public abstract boolean offer(E paramE)： 将指定元素插入此队列中（如果立即可行且不会违反容量限制），成功时返回 true，如果当前没有可用的空间，则返回 false。

3： public abstract void put(E paramE) throws InterruptedException： 将指定元素插入此队列中，将等待可用的空间（如果有必要）

```
public void put(E paramE) throws InterruptedException {
    checkNotNull(paramE);
    ReentrantLock localReentrantLock = this.lock;
    localReentrantLock.lockInterruptibly();
    try {
            while (this.count == this.items.length)
            this.notFull.await();//如果队列满了，则线程阻塞等待
            enqueue(paramE);
            localReentrantLock.unlock();
        } finally {
            localReentrantLock.unlock();
        }
}
```

4： offer(E o, long timeout, TimeUnit unit)： 可以设定等待的时间， 如果在指定的时间内， 还不能往队列中加入 BlockingQueue， 则返回失败。

获取数据操作：

1： poll(time):取走 BlockingQueue 里排在首位的对象,若不能立即取出,则可以等 time 参数规定的时间,取不到时返回 null;

2： poll(long timeout, TimeUnit unit)： 从 BlockingQueue 取出一个队首的对象， 如果在指定时间内， 队列一旦有数据可取， 则立即返回队列中的数据。否则直到时间超时还没有数据可取，返回失败。

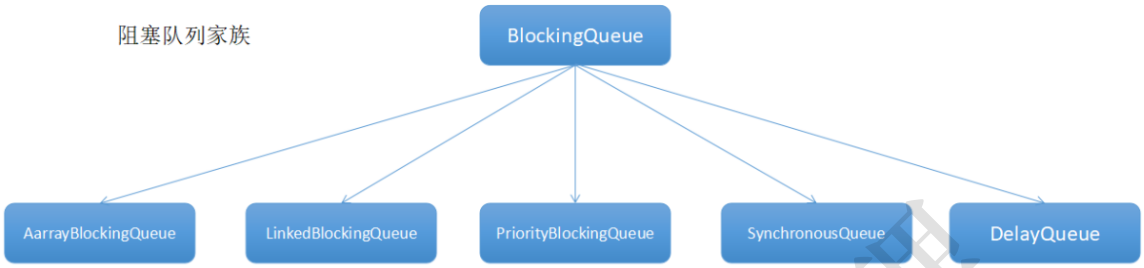
3： take():取走 BlockingQueue 里排在首位的对象,若 BlockingQueue 为空,阻断进入等待状态直到 BlockingQueue 有新的数据被加入。

4.drainTo():一次性从 BlockingQueue 获取所有可用的数据对象（还可以指定获取数据的个数），通过该方法，可以提升获取数据效率；不需要多次分批加锁或释放锁。

77、Java 中的阻塞队列

1. ArrayBlockingQueue ：由数组结构组成的有界阻塞队列。
2. LinkedBlockingQueue ：由链表结构组成的有界阻塞队列。
3. PriorityBlockingQueue ：支持优先级排序的无界阻塞队列。
4. DelayQueue：使用优先级队列实现的无界阻塞队列。
5. SynchronousQueue：不存储元素的阻塞队列。

6. LinkedTransferQueue：由链表结构组成的无界阻塞队列。
7. LinkedBlockingDeque：由链表结构组成的双向阻塞队列



78、ArrayBlockingQueue（公平、非公平）

用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。 默认情况下不保证访问者公平的访问队列，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列

```
ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(1000,true);
```

79、LinkedBlockingQueue（两个独立锁提高并发）

基于链表的阻塞队列，同 ArrayListBlockingQueue 类似，此队列按照先进先出（FIFO）的原则对元素进行排序。而 LinkedBlockingQueue 之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。LinkedBlockingQueue 会默认一个类似无限大小的容量（Integer.MAX_VALUE）

80、PriorityBlockingQueue（compareTo 排序实现优先）

是一个支持优先级的无界队列。默认情况下元素采取自然顺序升序排列。 可以自定义实现 compareTo()方法来指定元素进行排序规则，或者初始化 PriorityBlockingQueue 时，指定构造参数 Comparator 来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

81、DelayQueue（缓存失效、定时任务 ）

是一个支持延时获取元素的无界阻塞队列。队列使用 PriorityQueue 来实现。队列中的元素必须实现 Delayed 接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。我们可以将 DelayQueue 运用在以下应用场景：

1. 缓存系统的设计：可以用 DelayQueue 保存缓存元素的有效期，使用一个线程循环查询 DelayQueue，一旦能从 DelayQueue 中获取元素时，表示缓存有效期到了。
2. 定时任务调度：使用 DelayQueue 保存当天将会执行的任务和执行时间，一旦从 DelayQueue 中获取到任务就开始执行，从比如 TimerQueue 就是使用 DelayQueue 实现的

82、SynchronousQueue（不存储数据、可用于传递数据）

是一个不存储元素的阻塞队列。每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。SynchronousQueue 可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另外一个线程使用，SynchronousQueue 的吞吐量高于 LinkedBlockingQueue 和 ArrayBlockingQueue。

83、LinkedTransferQueue

是一个由链表结构组成的无界阻塞 TransferQueue 队列。相对于其他阻塞队列，LinkedTransferQueue 多了 tryTransfer 和 transfer 方法。

1. transfer 方法： 如果当前有消费者正在等待接收元素（消费者使用 take()方法或带时间限制的 poll()方法时），transfer 方法可以把生产者传入的元素立刻 transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer 方法会将元素存放在队列的 tail 节点，并等到该元素被消费者消费了才返回。
2. tryTransfer 方法。则是用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回 false。和 transfer 方法的区别是 tryTransfer 方法无论消费者是否接收，方法立即返回。而 transfer 方法是必须等到消费者消费了才返回。
   对于带有时间限制的 tryTransfer(E e, long timeout, TimeUnit unit)方法，则是试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回 false，如果在超时时间内消费了元素，则返回 true。

84、LinkedBlockingDeque

是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque 多了 addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast 等方法，以 First 单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以 Last 单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法 add 等同于 addLast，移除方法 remove 等效于 removeFirst。但是 take 方法却等同于 takeFirst，不知道是不是 Jdk 的 bug，使用时还是用带有 First 和 Last 后缀的方法更清楚。
在初始化 LinkedBlockingDeque 时可以设置容量防止其过渡膨胀。另外双向阻塞队列可以运用在“工作窃取”模式中。

85、在 java 中守护线程和本地线程区别

java 中的线程分为两种：守护线程（Daemon）和用户线程（User）。

任何线程都可以设置为守护线程和用户线程，通过方法 Thread.setDaemon(boolean on)；true 则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon()必须在 Thread.start()之前调用，否则运行时会抛出异常。

两者的区别：
唯一的区别是判断虚拟机(JVM)何时离开，Daemon 是为其他线程提供服务，如果全部的 User Thread 已经撤离，Daemon 没有可服务的线程，JVM 撤离。也可以理解为守护线程是 JVM 自动创建的线程（但不一定），用户线程是程序创建的线程；比如 JVM 的垃圾回收线程是一个守护线程，当所有线程已经撤离，不再产

生垃圾，守护线程自然就没事可干了，当垃圾回收线程是 Java 虚拟机上仅剩的线程时，Java 虚拟机会自动离开。

扩展：

Thread Dump 打印出来的线程信息，含有 daemon 字样的线程即为守护进程，可能会有：服务守护进程、编译守护进程、windows 下的监听 Ctrl+break 的守护进程、Finalizer 守护进程、引用处理守护进程、GC 守护进程。

86、线程与进程的区别？

进程是操作系统分配资源的最小单元，线程是操作系统调度的最小单元。
一个程序至少有一个进程,一个进程至少有一个线程。

87、什么是多线程中的上下文切换？

多线程会共同使用一组计算机上的 CPU，而线程数大于给程序分配的 CPU 数量时，为了让各个线程都有执行的机会，就需要轮转使用 CPU。不同的线程切换使用 CPU 发生的切换数据等就是上下文切换。

88、死锁与活锁的区别，死锁与饥饿的区别？

死锁：是指两个或两个以上的进程（或线程）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。

产生死锁的必要条件：
1、互斥条件：所谓互斥就是进程在某一时间内独占资源。
2、请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3、不剥夺条件:进程已获得资源，在末使用完之前，不能强行剥夺。
4、循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。
活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java 中导致饥饿的原因：
1、高优先级线程吞噬所有的低优先级线程的 CPU 时间。
2、线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
3、线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)，因为其他线程总是被持续地获得唤醒。

89、Java 中用到的线程调度算法是什么？

采用时间片轮转的方式。可以设置线程的优先级，会映射到下层的系统上面的优先级上，如非特别需要，尽量不要用，防止线程饥饿。

90、什么是线程组，为什么在 Java 中不推荐使用？

ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，
也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。
为什么不推荐使用？因为使用有很多的安全隐患吧，没有具体追究，如果需要使
用，推荐使用线程池。

91、为什么使用 Executor 框架？

每次执行任务创建线程 new Thread()比较消耗性能，创建一个线程是比较耗时、
耗资源的。

调用 new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限制的创建，
线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪，还有线程之间的
频繁交替也会消耗很多系统资源。

接使用 new Thread() 启动的线程不利于扩展，比如定时执行、定期执行、定时
定期执行、线程中断等都不便实现。

92、在 Java 中 Executor 和 Executors 的区别？

Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。
Executor 接口对象能执行我们的线程任务。
ExecutorService 接口继承了 Executor 接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。
使用 ThreadPoolExecutor 可以创建自定义线程池。

Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的
完成，并可以使用 get()方法获取计算的结果。

93、如何在 Windows 和 Linux 上查找哪个线程使用的 CPU 时间最长？

参考：
http://daiguahub.com/2016/07/31/使用 jstack 找出消耗 CPU 最多的线程代码/

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

原子操作（atomic operation）意为”不可被中断的一个或一系列操作” 。处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。 CAS 操作——
Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS的原子操作。

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。

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。

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 类型的原子包装类，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。

java.util.concurrent 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。

原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference
原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray
原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater
解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个 boolean来反映中间有没有变过），AtomicStampedReference（通过引入一个 int 来累加来反映中间有没有变过）

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

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。

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

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

96、什么是 Executors 框架？

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。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。

97、什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。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。
JDK7 提供了 7 个阻塞队列。分别是：

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
DelayQueue：一个使用优先级队列实现的无界阻塞队列。
SynchronousQueue：一个不存储元素的阻塞队列。
LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

Java 5 之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好，wait ,notify,notifyAll,sychronized 这些关键字。而在 java 5 之后，可以使用阻
塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。

BlockingQueue 接口是 Queue 的子接口，它的主要用途并不是作为容器，而是作为线程同步的的工具，因此他具有一个很明显的特性，当生产者线程试图向BlockingQueue 放入元素时，如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向 BlockingQueue 中放入元素，取出元素，它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景就是 socket 客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

98、什么是 Callable 和 Future?

Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值，这个返回值可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。可以认为是带有回调的 Runnable。

Future 接口表示异步任务，是还没有完成的任务给出的未来结果。所以说 Callable 用于产生结果，Future 用于获取结果。

99、什么是 FutureTask?使用 ExecutorService 启动任务。

在 Java 并发程序中 FutureTask 表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是调用了 Runnable接口所以它可以提交给 Executor 来执行。

100、什么是并发容器的实现？

何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如 Vector，Hashtable，以及 Collections.synchronizedSet，synchronizedList 等方法返回的容器。可以通过查看 Vector，Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 synchronized。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性，例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制，可以称为分段锁，在这种锁机制下，允许任意数量的读线程并发地访问 map，并且执行读操作的线程和写操作的线程也可以并发的访问 map，同时允许一定数量的写操作线程并发地修改 map，所以它可以在并发环境下实现更高的吞吐量。

101、多线程同步和互斥有几种实现方法，都是什么？

线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

102、什么是竞争条件？你怎样发现和解决竞争？

当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件（race condition）。

103、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？

当你调用 start()方法时你将创建新的线程，并且执行在 run()方法里的代码。但是如果你直接调用 run()方法，它不会创建新的线程也不会执行调用线程的代码，只会把 run 方法当作普通方法去执行。

104、Java 中你怎样唤醒一个阻塞的线程？

在 Java 发展史上曾经使用 suspend()、resume()方法对于线程进行阻塞唤醒，但随之出现很多问题，比较典型的还是死锁问题。

解决方案可以使用以对象为目标的阻塞，即利用 Object 类的 wait()和 notify()方法实现线程阻塞。

首先，wait、notify 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取改对象的锁，直到获取成功才能往下执行；其次，wait、notify 方法必须在 synchronized 块或方法中被调用，并且要保证同步块或方法的锁对象与调用 wait、notify 方法的对象是同一个，如此一来在调用 wait 之前当前线程就已经成功获取某对象的锁，执行 wait 阻塞后当前线程就将之前获取的对象锁释放。

105、在 Java 中 CycliBarriar 和 CountdownLatch 有什么区别？

CyclicBarrier 可以重复使用，而 CountdownLatch 不能重复使用。 Java 的 concurrent 包里面的 CountDownLatch 其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。你可以向 CountDownLatch 对象设置一个初始的数字作为计数值，任何调用这个对象上的 await()方法都会阻塞，直到这个计数器的计数值被其他的线程减为 0 为止。

所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。

CountDownLatch 的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个 CountDownLatch 对象的 await()方法，其他的任务执行完自己的任务后调用同一个 CountDownLatch 对象上的 countDown()方法，这个调用 await()方法的任务将一直阻塞等待，直到这个 CountDownLatch 对象的计数值减到 0 为止。

CyclicBarrier 一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrie在释放等待线程后可以重用，所以称它为循环 的 barrier。

106、什么是不可变对象，它对写并发应用有什么帮助

不可变对象(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象(Mutable Objects)。不可变对象的类即为不可变类(Immutable Class)。Java 平台类库中包含许多不可变类，如 String、基本类型的包装类、BigInteger 和 BigDecimal 等。不可变对象天生是线程安全的。它们的常量（域）是在构造函数中创建的。既然它们的状态无法修改，这些常量永远不会变。

不可变对象永远是线程安全的。
只有满足如下状态，一个对象才是不可变的；
它的状态不能在创建后再被修改；
所有域都是 final 类型；并且，
它被正确创建（创建期间没有发生 this 引用的逸出）。

107、Java 中用到的线程调度算法是什么？

计算机通常只有一个 CPU,在任意时刻只能执行一条机器指令,每个线程只有获得CPU 的使用权才能执行指令.所谓多线程的并发运行,其实是指从宏观上看,各个线程轮流获得 CPU 的使用权,分别执行各自的任务.在运行池中,会有多个处于就绪状态的线程在等待 CPU,JAVA 虚拟机的一项任务就是负责线程的调度,线程调度是指按照特定机制为多个线程分配 CPU 的使用权.

有两种调度模型：分时调度模型和抢占式调度模型。分时调度模型是指让所有的线程轮流获得 cpu 的使用权,并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。 java 虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

108、什么是线程组，为什么在 Java 中不推荐使用？

线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。