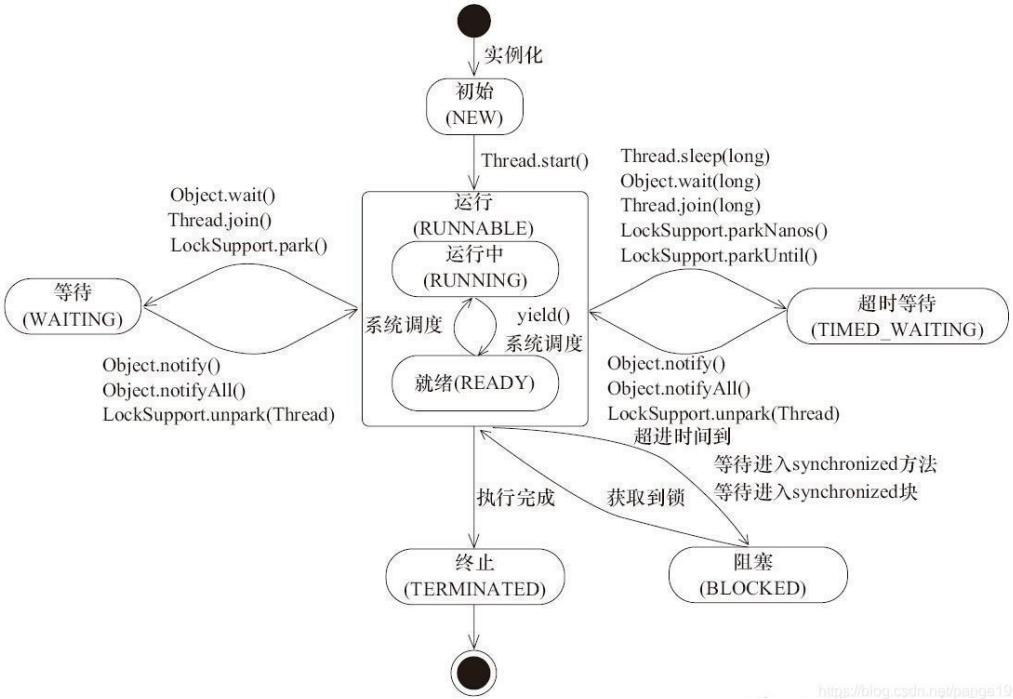
Multithreading Interview by Java - 1

线程的状态



6种状态

- 初始状态:新创建了一个线程对象，当时没有调用 start 方法
- 运行状态:Java 线程中将**就绪**和**运行**中两种状态统称为运行状态。线程对象创建完成以后，例如其他线程调用了该线程对象的 start 方法，那么该线程将会处于可运行线程池中，等待被线程调度选中，获取 CPU 的使用权，此时处于就绪状态，就绪状态的线程在获得 CPU 时间篇以后将会变为运行中状态。
- 阻塞状态:表示线程将会处于停止运行状态。
- 等待状态:进入该状态的线程将会等待其他线程做出例如通知，或者中断信号。
- 超时等待状态:进入等待状态已经超时。
- 终止状态:线程已经执行完毕。

线程池

线程池是一种线程使用模式，线程过多会带来调度开销，进而影响缓存和整体性能，如果使用线程池，那么线程池将会维护着多个线程，等待着管理者可以分配并并发执行相关的任务，这能够避免在短时间内出现创建和销毁线程的情况出息，这样还可以保证计算机的相关资源能够充分利用，并且不会造成额外的开销。

使用 ThreadPoolExecutor 类来创建相关的线程池。

实现多线程的原理

- 使用内核线程实现 - CPU scheduler
- 使用用户线程实现 - user space thread scheduler
- 二者混合

锁

- Synchronized
- java.utils.concurrent.locks

```
public class Counter{
    private int count = 0;

    public int inc(){
        synchronized(this){
            return ++count;
        }
    }
}
```

```
public class Counter{
    private Lock lock = new Lock();
    private int count = 0;

    public int inc(){
        lock.lock();
        int newCount = ++count;
        lock.unlock();
        return newCount;
    }
}
```

锁

类别	synchronized	Lock
存在层次	Java的关键字，在jvm层面上	是一个类
锁的释放	1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放锁	在finally中必须释放锁，不然容易造成线程死锁
锁的获取	假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待	分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待
锁状态	无法判断	可以判断
锁类型	可重入 不可中断 非公平	可重入 可判断 可公平（两者皆可）
性能	少量同步	大量同步

锁的可重入性 – Reentrancy

Java 中 synchronized 同步快是可重入的，即在 Java 线程进入代码中的 synchronized 同步块的时候，并且获得该线程上的任意一个锁的时候，调用任何该段代码上的任意一段加锁代码都是能够进入的，都可以正常的运行

```
public class Reentrant{
    public synchronized outer(){
        inner();
    }

    public synchronized inner(){
        //do something
    }
}
```

```
public class Reentrant2{
    Lock lock = new Lock();

    public outer(){
        lock.lock();
        inner();
        lock.unlock();
    }

    public synchronized inner(){
        lock.lock();
        //do something
        lock.unlock();
    }
}
```

锁的公平性 - Fairness by `java.util.concurrent.locks.ReentrantLock`

Java 中的 synchronized 块并不能保证进入线程的顺序，因此，如果有多个线程同时竞争访问同一块 synchronized 同步块，那么就会有一种风险，有一个线程永远都获得不了锁，这种情况称之为线程饥饿，为了避免这种情况发生，锁需要有公平性。

```java
public class MyFairLock extends Thread{

    private ReentrantLock lock = new ReentrantLock(true);
    public void fairLock(){
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()  + "正在持有锁");
        }finally {
            System.out.println(Thread.currentThread().getName()  + "释放了锁");
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        MyFairLock myFairLock = new MyFairLock();
        Runnable runnable = () -> {
            System.out.println(Thread.currentThread().getName() + "启动");
            myFairLock.fairLock();
        };
        Thread[] thread = new Thread[10];
        for(int i = 0;i < 10;i++){
            thread[i] = new Thread(runnable);
        }
        for(int i = 0;i < 10;i++){
            thread[i].start();
        }
    }
}
```

finally 语句中调用 unlock()

如果用 lock 来保护临界区，如果临界区中抛出异常，那么如果 finaly 语句中没有解锁，那么将会造成阻塞长时间的出现，导致死锁的发生。

```
lock.lock();
try{
    //do critical section code,
    //which may throw exception
} finally {
    lock.unlock();
}
```

线程的调度

抢占式调度

抢占式调度是指每条线程执行的时候，线程的切换都由系统控制，系统控制是说在一些运行的方式下，每条线程有可能得不到时间片，有可能得到时间片，有可能得到时间片时间长，有可能得到时间片时间短。这样不会一个线程的阻塞造成其他线程也发生阻塞。

协同式调度

协同式调度是说某一个线程执行完成以后会自己通知系统切换到另一个线程上执行。线程的执行时间由线程本身控制，线程可以进行预知执行的时间，不会有多线程同步的问题，当时如果有一个线程出现长时间的阻塞，那么整个系统将会发生阻塞。

JVM 线程调度实现

Java 使用的是抢占式线程调度，按照优先级分配 CPU 的时间片，并且优先级最高的执行的级别越高，但是优先级并不代表时间片的长短，有可能是反过来的。

多线程的调试