创建线程的两个方法

继承Thread 和 实现Runnable，如果调用了线程的run()方法，那就是一个简单的函数调用，如果调用了start()方法，那么操作系统会分配一条单独的执行流，也就是会启动一个线程，只有调用了start()才会开启线程。

线程的基本属性和方法:

id，name，状态(State)，优先级，是否是守护线程(isDaemon)。守护线程可以理解为一个"监视者"，如果一个进程只有守护线程，那么进程就会退出。比如垃圾回收线程，就是守护线程，可以通过thread.setDaemon(true)设置为守护线程。

sleep()方法

public static native void sleep(long millis) throws InterruptedException; 让当前线程让出cpu，并且休眠millis毫秒，在休眠期间不会再去争夺cpu，注意是可中断的，可以使用thread.interrupt()来中断正在休眠的线程。

yield()方法

public static native void yield(); 让出cpu重新竞争，可以理解为: A目前抢到了篮球(cpu)，然后又抛出去和大家一起抢。

join()方法

public final void join() throws InterruptedException; "可以让当前所在线程等待join()的线程跑完再执行"，同样需要处理中断异常，举个例子:

public class T extends Thread {

@Override

public void run() {

Thread t = new Thread();

t.join();

//t跑完了这里才继续执行

System.out.println("after t join");

}

}

线程的生命周期

public enum State {

NEW,

RUNNABLE,

BLOCKED,

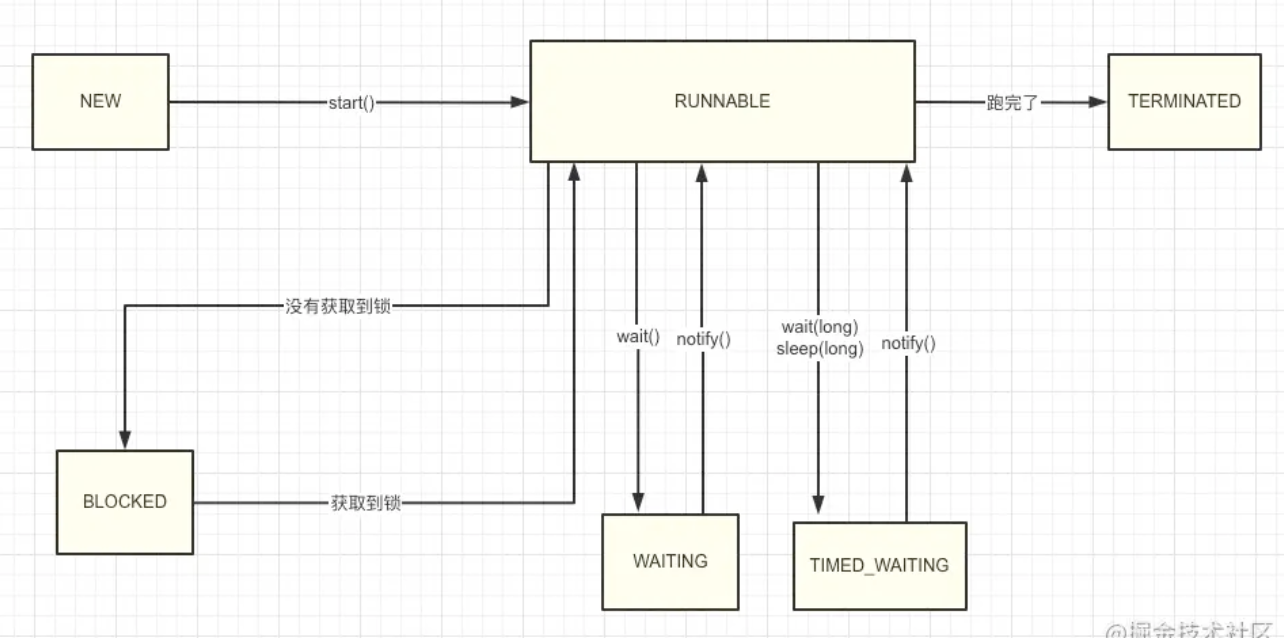
WAITING,

TIMED\_WAITING,

TERMINATED

}

创建一个线程，那么就是NEW状态，调用start()跑起来，就是RUNNABLE状态，跑完了就是TERMINATED状态，如果遇上synchronized等同步锁，且获取不到锁的时候，就是BLOCKED状态，遇到wait()就进入WAITTING状态，遇到wait(long)/sleep(long)则会进入TIMED\_WAITTING状态。



**同步与JVM内存模型**

public class T {

private int a = 1;

int size = 100000;

Thread[] threads = new Thread[size];

for(int i = 0; i < size; i++) {

threads[i] = new Thread(){

run(){

a++;

}

}

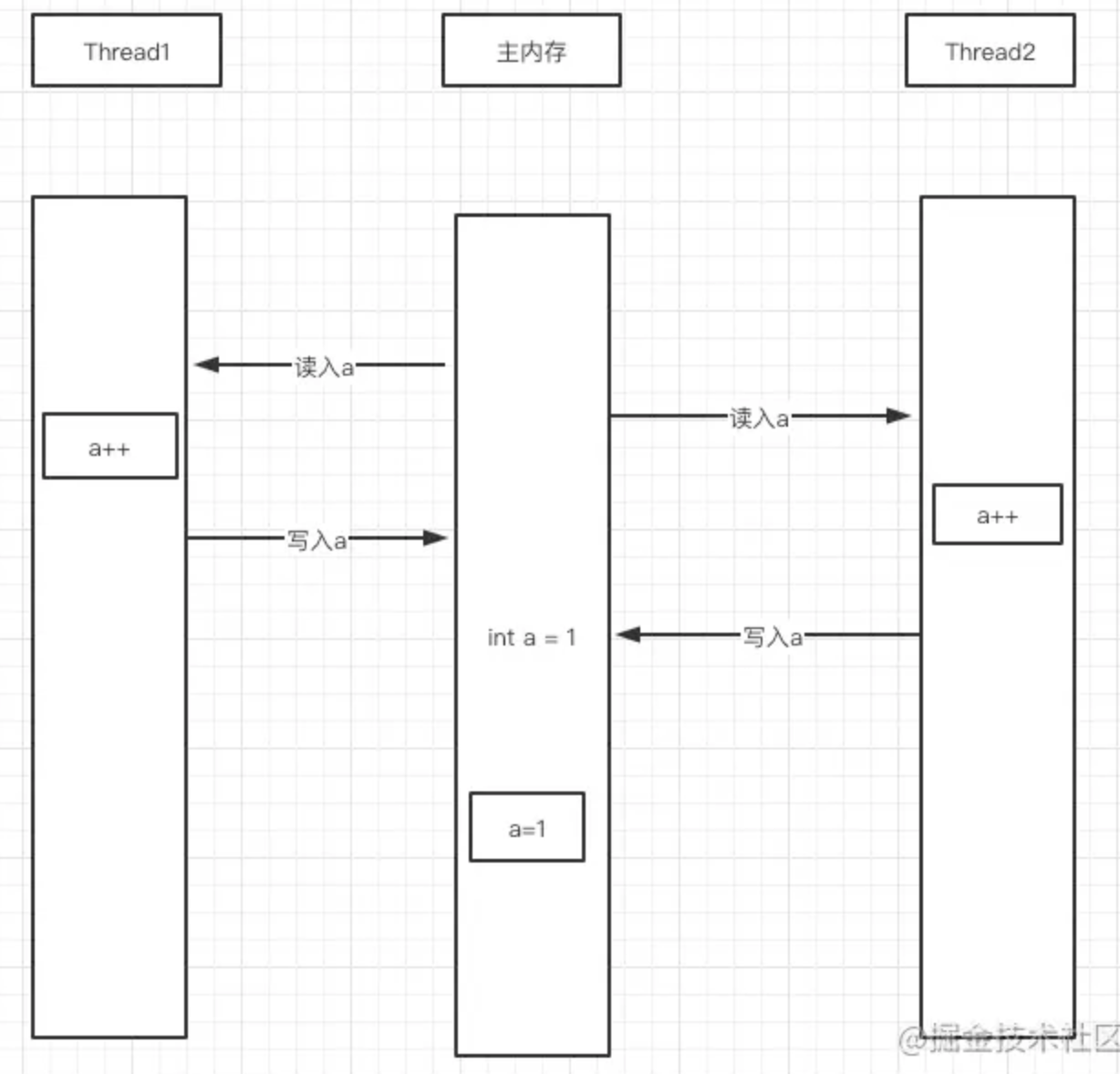
}

for(Thread t : threads) t.start();

System.out.println(a);//a是多少?

}

上述运行结果大多数情况是对的(因为cpu跑的快)，但是也有不对的时候，不对的原因就是因为并发情况下，程序不再顺序执行，我们看下并发情况下的程序流程:



可以见到，线程1从主内存读数据的时候，在自己的工作内存进行计算，这时候线程2也从主内存读数据，然后也自己计算，此时线程1计算完毕写入内存，紧接着线程2也计算完毕写入内存，从而导致两个运算做了相同的操作，这就是同步问题。怎么解决呢？很简单，我们可以设置线程1在进行计算的时候，让其他线程排队等待，等到线程1计算完毕，写回主内存的时候，再让其他线程执行。synchronized就可以解决这个问题

public class T {

//声明为volatile，保证每次都从内存同步

private volatile int a = 1;

//声明锁对象

private Object lock = new Object();

int size = 100000;

Thread[] threads = new Thread[size];

for(int i = 0; i < size; i++) {

threads[i] = new Thread(){

run(){

//加锁

synchronized(lock){

a++;

}

}

}

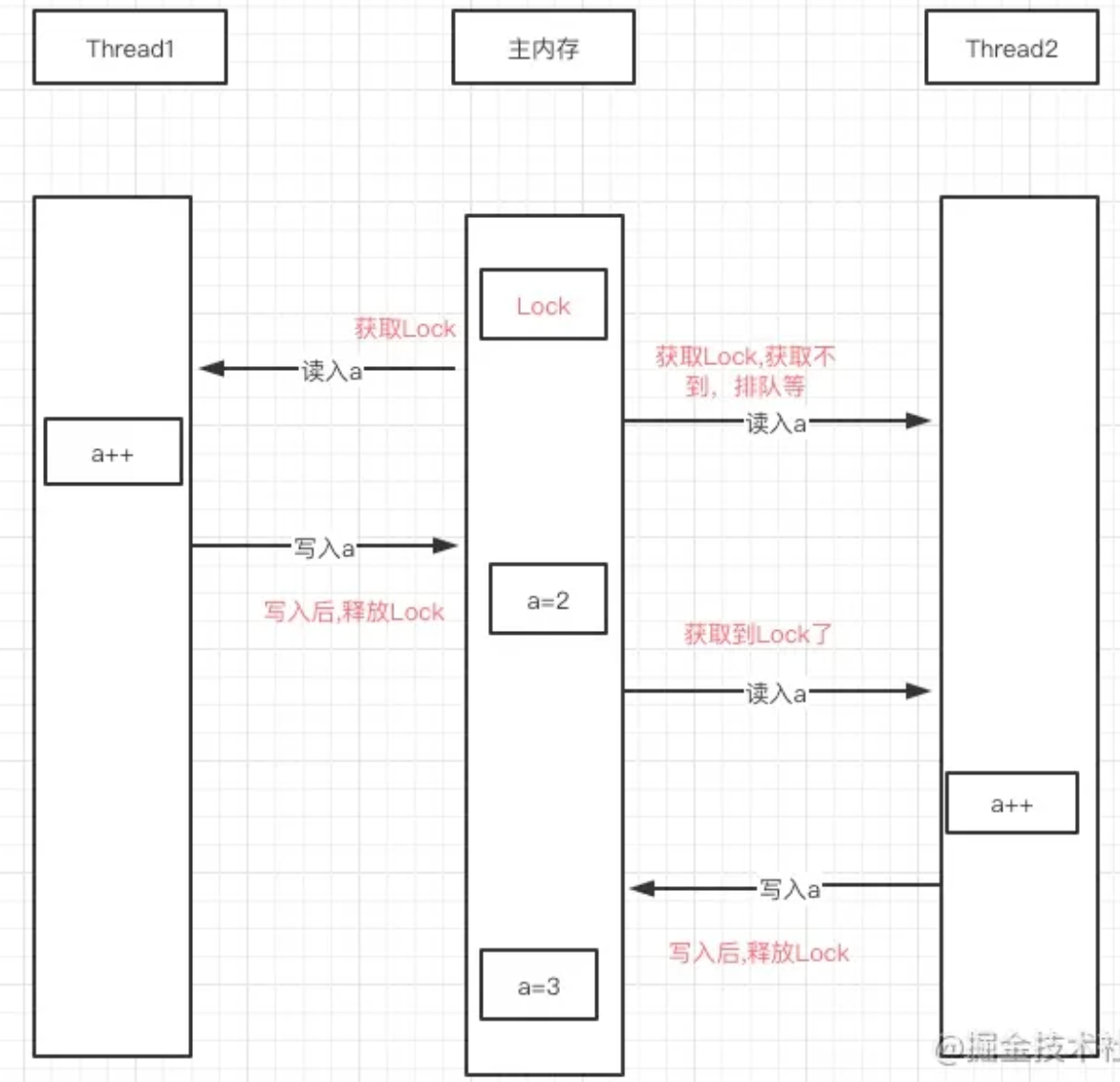
}

for(Thread t : threads) t.start();

System.out.println(a);//a是多少?

}

synchronized同步块每次都会请求获取锁，如果获取到了，则向下执行，执行完毕会自动释放锁并且随机唤醒一个正在等待的线程；如果获取不到，则会排队等待，等待到了才会继续执行。可以把synchronized块内的代码理解为一个"原子操作"。



我们也需要用volatile修饰变量a，volatile的作用是: 每次都从内存中同步，也就是保证每次都从内存中读数据，并且操作完后会及时写回内存。

### 线程的协作

每一个对象都有一个"锁等待队列"和一个"条件等待队列"，其中锁等待队列用在synchronized(lock)等场景中，比如当前线程获取不到lock锁，就会添加到lock的锁等待队列中去，如果在一个线程中调用了object.wait(); 那么这个线程就会阻塞并添加到object的条件等待队列中去。锁等待队列中的线程，在申请到锁的时候，就会被移出；条件等待队列中的线程，在调用了object.notify()/notifyAll()后会被移出，notify()会随机选择一个线程来唤醒并移出等待队列，notifyAll()会全部唤醒并移出等待队列。

wait()/notify()需要在synchronized代码块内被调用，如果不在synchronized代码块内，则会抛出java.lang.IllegalMonitorStateExcception；并且synchronized同步的锁对象也必须是wait()/notify()的接收者，否则也会抛出同样的异常。wait()/notify()是响应中断的。伪代码如下:

public void test(){

//需要用synchronized保护

synchronized(lock){

try {

//wait()的调用者需要和synchronized同步的锁对象相同

while(!condition) lock.wait();

} catch (InterruptedException e) {

//需要捕获中断异常

e.printStackTrace();

}

}

}

为什么wait()一定要运行在synchronized块内呢？因为竞态条件，比如上面的condition，如果不加synchronized，那么在while(!condition)这个条件满足后，在将要运行wait()前，其他线程更改了condition的值，就会导致condition为true的情况下也进行了wait()，这显然是不对的，所以我们加synchronized是为了保证"判断条件和wait()"这两步操作是"原子"的，同理notify()也是这样的。

为什么wait()一定要运行在synchronized块内呢？因为竞态条件，比如上面的condition，如果不加synchronized，那么在while(!condition)这个条件满足后，在将要运行wait()前，其他线程更改了condition的值，就会导致condition为true的情况下也进行了wait()，这显然是不对的，所以我们加synchronized是为了保证"判断条件和wait()"这两步操作是"原子"的，同理notify()也是这样的。

### 线程的中断

中断就是终止正在进行的动作，比如线程正在运行就停止运行，正在等待就停止等待，正在休眠就停止休眠

public void test(){

Thread thread = new Thread() {

@Override

public void run() {

//检测是否被中断

while (!isInterrupted()) {

System.out.println("running");

}

}

};

thread.start();

//中断

thread.interrupt();

}

这里简单解释下相关方法:

* 1. public boolean isInterrupted(); //检测当前线程是否被中断
  2. public void interrupt(); //中断当前线程

此外，还有个静态方法:

* 1. public static boolean interrupted(); //返回当前线程是否被中断，并且"清空中断标志位"，也就是说，返回后就会清空中断标志位，如果连续调用，第二次肯定为false，因为第一次调用之后，中断标记为就会被清空。

不同状态下线程对中断的反应不同:

* 如果线程处于RUNNABLE状态下，则会直接结束；
* 如果处于WAITTING/TIMED\_WAITTING则会抛出InerruptedException，并且中断标记位被清空；
* 如果处于BLOCKED状态，只会设置线程的中断标志位，仍然处于BLOKCED状态，因为线程要从BLOCKED出来需要获取锁，而线程中断并不会使得线程获得锁；synchronized是不响应中断的，这是它的缺点；

如果处于NEW/TERMINATED状态，则无任何作用

所以如果我们要正确的取消线程，不应该不分青红皂白的使用interrupt()，而是应该封装一套逻辑来调用，比如如下伪代码:

public class T extends Thread{

private boolean cancel = false;

run(){

if(!cancel) {

}else {

//做一些逻辑判断处理

}

}

public void cancel(){

this.cancel = true;

interrupt();

}

}

具体代码可以参看java.util.concurrent.FutureTask的cancel()实现。