1. 新建一个线程的方式

for (int j = 0; j < 100; j++) {  
 int z = j;  
 Thread t = new Thread(() -> {  
 while (true) {  
 System.*out*.println(z);  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
 t.start();  
}

1. 不要调用Thread或者Runnable对象的run方法。直接调用run方法只会执行同一个线程中的任务，不会启动新线程。应该通过thread.start()方法。这个方法将创建一个执行run方法的新线程。
2. Thread.interrupt方法，当一个被阻塞的线程（调用sleep或wait）上调用interrupt方法时，阻塞调用将被Interrupted Exception异常中断。

Interrupt设置中断状态为true。

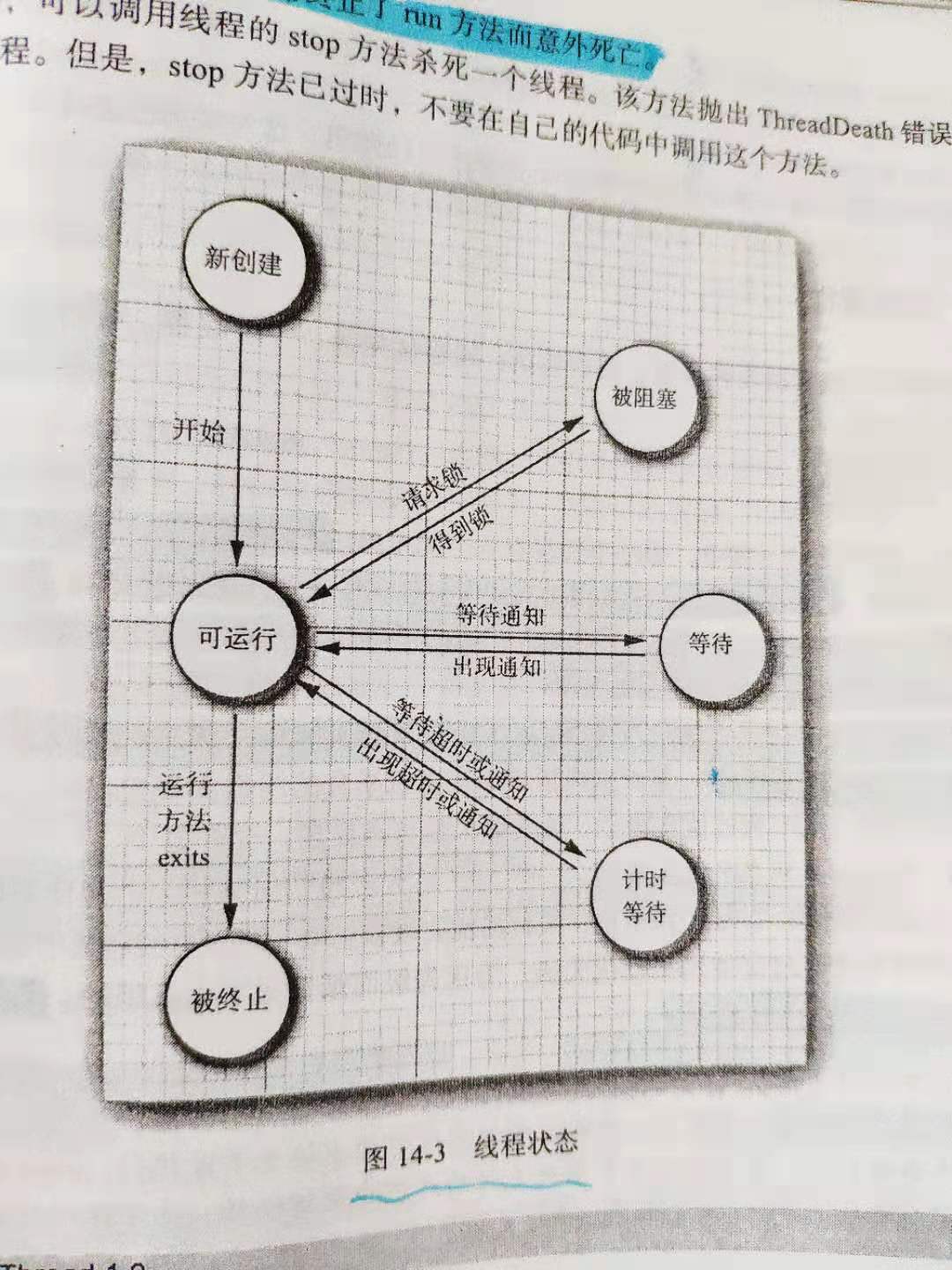
Thread.isInterrupted方法检测线程的中断状态。

Thread t = new Thread(() -> {  
 while (true) {  
 System.*out*.println("1");  
 try {  
 System.*out*.println("2");  
 //休眠  
 Thread.*currentThread*().*sleep*(5000);  
 //设置中断状态，也就是打断阻塞状态  
 Thread.*currentThread*().interrupt();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
});  
t.start();

1. Static Thread.currentThread()返回当前执行的Thread对象。
2. 获取当前线程状态

Thread.currentThread().getState()

有New、Runnable、Blocked、Waiting、Timed waiting（计时等待）、Terminated（被终止）



1. 当线程处于被阻塞或等待状态时，它暂时不活动。不运行任何代码且消耗最少的资源。直到线程调度器重新激活它。

•当一个线程试图获取一个内部的对象锁，而该锁被其他线程持有，则该线程进入阻塞状态。当所有其他线程释放该锁，并且线程调度器允许本线程持有它的时候，该线程将变成非阻塞状态。

•当线程等待另一个线程通知调度器一个条件时，它自己进入等待状态。在调用Object.wait或Thread.join，或等待concurrent库中的Lock或者Condition时，就会出现这种情况。

•有几个方法有超时参数。调用它们导致线程进入计时等待(Timed waiting)状态。这一状态将一直保持到超时期满或者接收到适当的通知。带有超时参数的方法有Thread.sleep和Object.wait、Thread.join\Lock.tryLock及Condition.await的计时版。

1. 线程被终止

•因为run方法正常退出而自然死亡。

•因为一个没有捕获的异常终止了run方法而意外死亡。调用stop的方法就是因为抛出一个ThreadDeath错误对象，由此杀死线程（不要在自己的方法中调用这个方法）。

1. Join 或join(long millis)等待终止指定的线程。
2. stop（）停止该线程。

suspend()暂停这一线程的执行。

Resume()恢复线程。

以上3个方法都已过时。

1. 线程优先级，在java中，一个线程默认继承它父线程的优先级。

优先级越高会被线程调度器优先选择。

不要将程序构建为功能的正确性依赖于优先级。

Thread.*MIN\_PRIORITY*; 1  
Thread.*NORM\_PRIORITY*; 5  
t.setPriority(Thread.*MAX\_PRIORITY*); 10

1. Statis yield()导致当前执行线程处于让步状态。如果有其他的可运行线程具有与此线程同样高的优先级，那么这些线程接下来会被调度。
2. 守护线程

t.setDaemon(true)将线程转换成守护线程，这一方法必须在线程启动前调用。

守护线程的唯一用途是为其他线程提供服务。

如计时线程将定时的发送信号给其他线程或清空过时的缓存线程。

守护线程应该永远不去访问固有资源，如文件、数据库，因为它会在任何时候甚至一个操作中间发生中断。

1. 未捕获异常处理器。

处理器必须属于一个实现Thread.UncaughtExceptionHandler接口的类。这个接口只有一个方法。

Void uncaughtException(Thread tm Throwable e)

可以调用setUncaughtExceptionHandle为所有线程安装一个默认的处理器。替换处理器可以使用日志API发送未捕获异常的报告到日志文件。

如果不安装默认处理器，默认的处理器为空。但是如果不为独立的线程安装处理器，此时的处理器就是该线程的ThreadGroup对象。

线程组ThreadGroup类实现了Thread.UncaughtExceptionHandler接口。它的uncaughtException方法如下：

1. 如果该线程组有父线程组，那么父线程组的uncaughtException方法被调用。
2. 否则，如果Thread.getDefaultExceptionHandler方法返回一个非空的处理器，则调用该处理器。
3. 否则，如果Throwable是ThreadDeath实例，什么都不做。
4. 否则，线程的名字以及Throwable的栈轨迹被输出到System.err上。

有如下方法。

//设置和获取默认处理器  
Thread.setDefaultUncaughtExceptionHandler(Thread.*getDefaultUncaughtExceptionHandler*());  
//设置和获取未捕获异常处理器。如果没有安装处理器，则将线程组对象作为处理器。  
t.setUncaughtExceptionHandler(t.getUncaughtExceptionHandler());

1. 同步Lock

示例：

//新建一个锁对象  
private Lock bankLock = new ReentrantLock();  
  
public void transfer(int from, int to, double amount) {  
 bankLock.lock();//锁住  
 try {  
 accounts[from] = accounts[from] - amount;  
 accounts[to] = accounts[to] + amount;  
 System.*out*.println(getTotal());  
 } finally {  
 //解锁，要把解锁操作括在finally子句之内，如果临界区的代码抛出异常，锁必须被释放，否则，其他线程将永远阻塞。  
 bankLock.unlock();  
 }  
}

如果两个线程同时访问同一个Bank对象（带有锁的对象），那么锁以串行的方式提供服务。

getTotal()也拥有bankLock锁对象，且在getTotal()和transfer方法完成前，bankLock的对象持有计数未2，当两个方法都完成时，持有计数变为0，线程释放锁。

ReentrantLock()

ReentrantLock(Boolean fair)带有公平策略的锁。一个公平锁偏爱等待时间最长的线程。这一公平的保证将大大降低性能。默认情况下，锁没有被强制为公平的。

1. 临界区

临界区就是在同一时刻只能有一个任务访问的代码区。

1. 条件对象

示例:

public Bank(){  
 this.accounts = new double[10000];  
 Arrays.*fill*(accounts, 100);  
 //新建这个锁的条件对象  
 sufficientFunds = bankLock.newCondition();  
}  
  
//新建一个锁对象  
private Lock bankLock = new ReentrantLock();  
//条件  
private Condition sufficientFunds;  
  
public void transfer(int from, int to, double amount) {  
 bankLock.lock();//锁住  
 try {  
 //设置一个等待条件  
 while (accounts[from] < amount) {  
 sufficientFunds.await();  
 }  
 accounts[from] = accounts[from] - amount;  
 accounts[to] = accounts[to] + amount;  
  
 //当完成转账时，通知其他因为这个条件而等待的线程，其他线程会重新检测是否满足条件  
 sufficientFunds.signalAll();  
  
 System.*out*.println(getTotal());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 //解锁，要把解锁操作括在finally子句之内，如果临界区的代码抛出异常，锁必须被释放，否则，其他线程将永远阻塞。  
 bankLock.unlock();  
 }  
}

condition.signal() 随机解除等待集中的一个线程的阻塞状态。

1. 锁和条件的作用：

•锁用来保护代码片段，任何时刻这只能有一个线程执行被保护的代码。

•锁可以管理试图进入被保护代码段的线程。

•锁可以拥有一个或多个相关的条件对象。

•每个条件对象管理那些已经进入被保护的代码但还不能运行的线程。

1. synchronized关键字

等价于方法使用lock和unlock

wait方法等价于await如果当前线程不是对象锁的持有者，该方法将抛出一个IllegalMonitorStateException

wait(long millis, int nanos)方法等价于await如果当前线程不是对象锁的持有者，该方法将抛出一个IllegalMonitorStateException

notifyAll方法等价于signalAll如果当前线程不是对象锁的持有者，该方法将抛出一个IllegalMonitorStateException

notify方法等价于signal如果当前线程不是对象锁的持有者，该方法将抛出一个IllegalMonitorStateException

示例：

public synchronized void transfer(int from, int to, double amount) {  
 try {  
 //设置一个等待条件  
 while (accounts[from] < amount) {  
 this.wait();  
 }  
 accounts[from] = accounts[from] - amount;  
 accounts[to] = accounts[to] + amount;  
  
 //当完成转账时，通知其他因为这个条件而等待的线程，其他线程会重新检测是否满足条件  
 this.notifyAll();  
  
 System.*out*.println(getTotal());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

1. synchronized(obj)同步阻塞块

示例1：

//为了用java对象持有的锁。  
private Object lock = new Object();  
  
public void transfer( int from, int to, double amount) throws InterruptedException {  
 //使用别的对象的锁  
 synchronized (lock) {  
 accounts[from] = accounts[from] - amount;  
 accounts[to] = accounts[to] + amount;  
 System.*out*.println(getTotal());  
 }  
}

示例2：

public void transfer(Vector<Double> accounts, int from, int to, double amount) throws InterruptedException {  
 //使用别的对象的锁,会把这个对象的方法锁死，使在使用这个对象的某个方法时，里面的其他方法不可以被调用。  
 //但是这个需要依赖于这个被获取锁的对象里面的方法都实现了内部锁，条件比较苛刻。  
 synchronized (accounts) {  
 accounts.set(from, accounts.get(from) - amount);  
 accounts.set(to, accounts.get(to) - amount);  
 System.*out*.println(getTotal());  
 }  
}

1. 监视器

•监视器至少包含私有域的类。

•使用该锁对所有方法进行加锁。如调用obj.method()时，自动给method加上锁，在方法返回时自动释放该锁。因为所有域是私有的，这样的安排可以确保一个线程正在对对象操作时，没有其他线程能够访问该域。

•该锁可以有任意多个相关条件。

Synchronized是一个不是很精确的监视器，但是java有三个问题，使得不是很满足监视器的安全性：

1. 域要求必须是private。
2. 方法不要求必须是synchronized。
3. 内部锁对客户是可用的。
4. Final修饰符，会在域初始化之后才看到这个变量。不然可能其他线程会看到null。但是这个参数内部也并不是线程安全的。