一、线程的实现/创建

1. 继承 Thread 类

public class MyThread extends Thread {

public void run() {

...

}

}

MyThread myThread1 = new MyThread();

myThread1.start();

2. 实现 Runnable 接口（如果类已经继承了另一个类，则无法直接继承Thread类）

public class MyThread extends OtherClass implements Runnable {

public void run() {

...

}

}

MyThread myThread1 = new MyThread();

Thread thread = new Thread(myThread1);

thread.start();

3. 通过Callable和Future创建线程

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实现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，再结合线程池接口ExecutorService就可以实现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 + ""); // 实现Callable接口

// 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对象中获取任务的返回值，并输出

System.out.println("res: " + f.get().toString());

}

4. 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。

那么可以使用缓存的策略，也就是使用线程池。

// 创建一个线程池

ExecutorService pool = Executors.newFixedThreadPool(taskSize);

while(true) {

pool.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();

}

}

})

}

二、四种线程池

1. newCachedThreadPool

对于执行很多短期异步任务的程序而言，这个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。

2. 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。

如果在所有线程处于活动状态时提交新的任务，则在有可用线程之前，会在队列中等待。

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）。

在某个线程被显示地关闭之前，池中的线程会一直存在。

3. newScheduledThreadPool

创建一个线程池，可以在给定延迟后运行任务或定期执行。

4. newSingleThreadExecutor

Executors.newSingleThreadExecutor()返回一个线程池（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。

三、线程的生命周期

新建（New）、就绪（Runnable）、运行（Running）、阻塞（Blocked）、死亡（Dead）

1. 新建（New）：使用new关键字创建了一个线程后，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。

2. 就绪（Runnable）：线程对象调用start方法后，该线程就处于就绪状态，JVM会为其创建方法调用栈和程序计数器，等待调度运行。

3. 运行（Running）：处于就绪状态的线程获得CPU，开始执行run方法的线程执行体，则该线程处于运行状态

4. 阻塞（Blocked）：线程因某种原因放弃了CPU的使用权，即让出了cpu timeslice，暂时停止运行。直到线程进入可运行状态，才有机会再次获得CPU使用权，转到运行状态。

死亡（Dead）：线程结束。

四、线程的三种阻塞状态

1. 等待阻塞：运行中的线程执行t.wait()方法，JVM会把该线程放入等待队列中。

2. 同步阻塞：运行中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中。

3. 其他阻塞：运行中的线程执行Thread.sleep或者t.join方法，或者发出了I/O请求时，JVM会把该线程设置为阻塞状态。

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，该线程重新转入可运行状态。

五、线程的三种死亡方式

1. 正常结束：run()或call()方法执行完成，线程正常结束。

2. 异常结束：线程抛出一个未捕获的Exception或者Error。

3. JVM崩溃导致所有线程结束

六、终止线程的四种方式

1. 正常运行结束

2. 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，但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true或false来控制while循环是否退出：

public class ThreadSafe extends Thread {

public volatile boolean exit = false;

public void run () {

while (!exit) {

// do something

}

}

}

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，也就是说同一时刻只能由一个线程来修改exit的值。

3.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，从而有机会结束该线程的执行。

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InterruptException异常后通过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

// 使用interrupt方法

try {

Thread.sleep(5\*1000); // 阻塞过程捕获中断异常来退出

} catch(InterruptedException e) {

e.printStackTrace();

break; // 捕获异常后，执行break跳出循环

}

}

}

}

4. stop方法终止线程（线程不安全）

程序中可以直接使用thread.stop()来强制终止线程，但是stop方法是很危险的，就像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()调用后，创建子线程的线程就会抛出ThreadDeath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，有可能导致一些奇怪的应用程序错误。因此，并不推荐使用该方法来终止线程。

七、sleep与wait区别

1. 对于sleep方法，属于Thread类的。而wait方法是属于Object类的。

2. sleep方法导致了程序暂停执行指定的时间，让出CPU给其他线程，但是它的监控状态依然保持着，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。

八、start和run区别

1. 调用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，这时该线程是处于就绪状态，并没有运行。

3. run方法称为线程体，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，该线程终止，然后CPU再调度其它线程。

九、JAVA后台线程

1. 定义：守护线程--也称为“服务线程”，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，用于为系统中的其他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. 线程是JVM级别的，以Tomcat为例，如果在Web应用中启动一个线程，该线程的生命周期并不会和Web应用程序保持同步。

也就是说，即使停止了Web应用，该线程依旧是活跃的。

6. 举例：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，用户实时监控和管理系统中的可回收资源。

7. 生命周期：守护线程是运行在后台的一种特殊线程。它独立于控制终端并且周期性地执行某中任务或者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。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。

class SimpleDaemons implements Runnable{

@Override

public void run() {

try {

while (true) {

// do something

}

} catch (Exception e) {

System.out.println("sleep() interrupted");

}

}

}

public class Main{

public static void main(String[] args) throws InterruptedException {

for (int i = 0; i < 10; i++) {

Thread daemon=new Thread(new SimpleDaemons());

daemon.setDaemon(true); // 设置为守护线程

daemon.start();

}

}

}

1. 并发编程的三要素
   * 1. 原子性：一个或多个操作，要么全部执行并且在执行的过程中不被其他操作打断，要么全部都不执行。
     2. 可见性：多个线程操作一个共享变量时，其中一个线程对变量进行修改后，其他线程可以立即看到修改的结果。
     3. 有序性：程序的执行顺序按照代码的先后顺序来执行。
2. 实现可见性的方法有哪些

Synchronized关键字或者Lock接口：保证同一时刻只有一个线程获取锁执行代码，锁释放之前把最新的值刷新到主内存，实现可见性。

1. 多线程的优势
   * 1. 发挥多核CPU的优势
     2. 防止阻塞
     3. 便于建模
2. 创建线程的三种方式对比
   * 1. 采用Runnable、Callable：线程类还可以继承其他类。在这种方式下，多个线程可以共享同一个目标对象，所以适合多个相同线程来处理同一份资源的情况，从而可以江CUP、代码和数据分开，形成清晰的模型。劣势就是，如果要访问当前线程，必须使用Thread.currentThread()方法。
     2. 使用继承Thread类：线程类无法再继承其他类，但是可以直接使用this获取当前线程
     3. Runnable、Callable的区别又是：前者重写的是run方法，而后者重写的是call方法。后者的任务执行后可返回值，前者没有返回值。后者的call方法可以抛出异常，run方法不可以。后者的结果可以拿到Future对象，通过这个对象可以了解任务执行的情况。
3. 线程池的优点
   * 1. 重用存在的线程，减少对象创建销毁的开销
     2. 可有效的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
     3. 提供定时、定期执行，单线程执行等功能
4. 常用的并发工具类

CountDownLatch、CyclicBarrier、Semaphore、Exchanger

1. CyclicBarrier和CountDownLatch的区别

①后者只能用一次，而前者可以使用reset重置循环使用

②后者主要实现1个或多个线程需要等待其他线程完成某项操作才能继续执行，而前者主要实现多个线程之间相互等待，直到所有线程都满足了条件才继续执行后续操作，能处理更复杂的场景。

1. synchronized的作用

用来控制线程同步，就是在多线程的环境下，控制synchronized代码段不被多个线程同时执行，既可以加在一段代码上，也可以加在方法上。

Synchronized锁的是什么：

①普通同步方法：锁的是当前实例对象

②静态同步方法：锁的是当前类的class对象

③同步代码块：锁的是synchronized括号里配置的对象

1. volatile的作用

用volatile来保证可见性。当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当其他线程需要读取时，它会去内存中读取新值。

1. 乐观锁和悲观锁

悲观锁是将资源锁住，等一个之前所得锁的线程释放锁后，下一个线程才可以访问。而乐观锁采取了一个宽泛的态度，通过某个方式不加锁来处理资源，比如通过给记录加version来获取数据。

1. CAS的问题
   * 1. CAS容易造成ABA问题，可以通过AtomicStampedReference解决。
     2. 不能保证代码块的原子性。CAS只能保证一个变量的原子性，代码块还是需要synchronized。
     3. CAS造成CPU利用率增加
2. Synchronized和ReentrantLock的区别
   * 1. 前者是关键字，后者是类
     2. 两者都是独占锁，但是前者的加锁解锁都是自动的，易于操作，但是不灵活，而后者需要手动进行加锁解锁，不易操作，但是灵活。
     3. 两者都可重入，但是前者不必担心最后是否释放锁，而后者必须手动释放后，才能使其他线程获得锁
     4. 前者不可响应中断，后者可以
     5. 后者可以对锁的等待时间进行设置，这样就避免了死锁
3. 如何唤醒一个阻塞的线程

可以中断线程（interrupt），并且通过抛出InterruptedException来唤醒。如果是IO阻塞，则没有办法，因为IO是操作系统层面的。

1. 线程类的构造方法、静态块是被哪个线程调用的

记住：构造方法、静态块是被new这个线程所在的线程调用的，而run方法是被线程自身调用的

例：Thread2中new了Thread1，main函数中new了Thread2

则Thread2的构造函数、静态块是被man线程的调用的，Thread1的则是被Thread2调用的；而Thread2的run是Thread2自身调用的，Thread1的run是被自身调用的

二十四、wait和notify

作用是使当前执行代码的线程进行等待或者通知线程继续，只能是在同步方法或者同步代码块中执行。

二十五、向线程池提交任务：pool.execute()/pool.submit()

关闭线程池：shutdown()/shutdownNow()

二十六、并发容器（J.U.C（java.util.concurrent））

ArrayList -> CopyOnWriteArrayList

HashSet -> CopyOnWriteArraySet（底层实现是CopyOnWriteArrayList）

TreeSet -> ConcurrentSkipListSet（和TreeSet一样支持自然排序）

HashMap -> ConcurrentHashMap

TreeMap -> ConcurrentSkipListMap

二十七、J.U.C之AQS

CountDownLatch、CyclicBarrier、Semaphore、ReentrantLock、Condition、FutureTask