# 线程

* 线程的状态：

创建、就绪、运行、阻塞、终止。

* Java提供两种创建线程的方法：继承类Thread、实现Runnable接口。

1、创建Thread类的子类，重写run方法，通过start方法启动线程。

2、实现Runnable接口，重写run方法，将该类的实例对象作为Thread的构造方法的参数，使用start方法启动。

* 比较：

使用Thread，由于Java单继承的特性，不能继承其他类；使用Runnable可以继承多个类。

* 线程的属性—线程标识符：

如果继承Thread，线程内通过this.getId()获得；如果实现Runnable，线程内通过Thread.currentThread().getId()获得。

* 线程的属性—线程名：

this.getName或Thread.currentThread.getName()。更改线程名通过this.setName()。

* 线程的属性—优先级：

setPriority()和getPriority()设置。最小值为1(Thread.MIN\_PRIORITY)，最大为10(Thread.MAX\_PRIORITY)，默认值为5(Thread.NORM\_PRIORITY)。调度策略：基于优先级的时间片轮转。

* 线程的属性—状态：

通过this.getState()获得。以下值通过Java.lang.Thread.State获取，取值列表如下：

|  |  |
| --- | --- |
| 状态值 | 含义 |
| NEW | 创建 |
| RUNNABLE | 正在运行或就绪 |
| BLOCKED | 阻塞，等待监视器锁(monitor lock) |
| WATING | 阻塞，调用了wait/join(with no timeout) |
| TIMED\_WAITING | 阻塞，调用了sleep/wait/join(with timeout) |
| TERMINATED | 终止 |

* 守护线程：

为用户线程（User Thread）提供服务。通过this.SetDaemon(true)设置为守护线程，该方法必须在start方法前调用。如果父线程为守护线程，子线程也是。通过this.isDaemon()判断是否是守护线程。如果jvm中都是守护进程，jvm将exit。守护进程的优先级较低。

应用：使用守护线程完成数据维护任务，当数据总数超过10，删除队列尾的两个数据。

* 线程管理—join方法：

当调用某个线程对象的join方法后，将会等待该线程执行结束。

应用：定义三个线程，线程A用来产生若干随机数，线程B用来计算这些数的和，线程C用来输出结果，具有先后顺序。

* 线程管理—sleep方法

使进程暂停运行一段固定时间。

* 线程管理—interrupt方法

使进程中断执行。线程本身无法调用该方法。如果一个线程由于调用了wait或join，则中断请求不会被响应。线程本身可以通过this.isInterrupted判断是否被中断。避免使用stop()方法。

* 线程分组

在同一分组的线程可以视作一个整体进行操作。通常的构造函数，一个是ThreadGroup（name），一个是ThreadGroup(parent,name)。名称是线程组的唯一标识。通过activeCount()方法获取当前线程组及其子线程组的活动的线程数。

* 带返回值的线程—Callable

带返回值的线程通过接口Callable定义,必须实现call()方法。定义的一般格式：

public class Worker implements Callable<Integer>{

public Integer call(){

//…

}

}

* 带返回值的线程—Future

接口Future允许在未来某个时间获取Callable接口定义的线程异步运行结果。一般使用FutureTask包装器，将Callable对象转化为Future对象。方法get()方法在计算完成时获取，否则会一直阻塞直到任务转入完成状态。

FutureTask<Integer> task=new FutureTask<Integer>(worker);

new Thread(task).start();

//…

Integer result=task.get();

# 线程同步

* 同步控制：

为了保证多线程环境下数据访问的正确性，即同一时刻只有一个线程对数据进行访问。

已有的同步控制机制：锁、原子块操作、软件事务性内存（STM）。

* 临界区（Critical Section）：线程中访问共享数据的那段代码。
* 监视器（Monitor）:

只有一个私有属性的类。

每个监视器类的对象都有一个相关联的锁。

Java中每个对象都有一个隐式的锁。

* 锁：分为加锁和解锁两个操作。Java提供同步锁、可重入锁、读写锁的锁机制。
* 同步锁：使用synchronized关键字，有两种形式：同步方法、同步块。
* 同步锁—同步方法：使用synchronized作为方法的修饰符，该方法同一时刻只有被一个线程访问。

public synchronized void method\_name(){

}

* 同步锁—同步块：使用synchronized作为某段代码的修饰符，需要明确指出监视器对象，常使用this。

同步块可以实现比同步方法更细粒度的同步控制。

* 可重入锁：无阻塞的互斥锁，与同步锁具有基本相同的行为和语义，但是有更多的功能：

|  |  |
| --- | --- |
| 方法 | 含义 |
| void lock() | 请求加锁 |
| void unlock() | 释放锁 |
| Boolean trylock() | 尝试获得锁，仅当在调用时刻没有其他线程持有该锁的情况下 |
| protected Thread getOwener() | 返回该锁的持有者 |
| protected Collection<Thread> getQueuedThreads() | 返回一当前正在试图获得该锁的线程集合 |
| int HoldCount() | 由当前线程持有该锁数 |
| int getQueueLength() | 获得当前正在试图获得该锁的线程集合大小 |

构造器ReentrantLock(Boolean fair)中参数fair指明了一个公平策略，保证等待了很长时间的线程获得该锁，即公平策略。

应用：实现线程安全的ArrayList的读写。

* 读写锁：

读锁可以由多个线程同时持有，写锁是排它锁，只能被一个线程持有。

与可重入锁类似，支持fair参数。常用方法与可重入锁类似。

构造器ReentrantReadWriteLock（），方法readLock（）、writeLock（）方法获得读锁、写锁。

应用：实现线程安全的ArrayList的读写。

* volatile变量:

为了降低多个线程访问同一个类的域变量的加锁开销，以volatile关键字修饰域变量，无需加锁，是线程安全的。

总而言之，下面三个情况，对一个域的访问是线程安全的：1、final修饰域变量。2、synchronized修饰域变量的访问方法。3、volatile修饰域变量。

volatile与synchronized不同的是：volatile是变量修饰符；synchronized是方法和代码块修饰符。

* 原子类：

原子类为单一变量提供了无锁的、线程安全的访问方式。本质是对volatile的拓展。例如AtomicInteger的常用方法：

|  |  |
| --- | --- |
| int addAndGet(int delta) | 增加delta，返回更新值 |
| int getAndAdd(int delta) | 增加delta，返回旧值 |
| int getAndSet(int newValue) | 设为新值，返回旧值 |
| void set(int newValue) | 设为新值 |
| int get() | 获取当前值 |

# 线程间通信

* wait方法：线程在对象上的等待。需要在synchronized中调用。方法调用后，线程进入阻塞状态并进入等待集合，释放对同步锁的控制权。
* notify方法：从该对象的等待集合中唤醒一个线程。一个线程阻塞建议使用该方法。
* notifyAll方法:将等待集合的所有线程唤醒。多个线程阻塞建议使用该方法。

应用：单缓冲区的生产者和消费者问题。