* 多线程基础

**1.1. 进程和线程**

**1.1.1. 什么是进程**

所谓进程（process）就是一块包含了某些资源的内存区域。操作系统利用进程把它的工作划分为一些功能单元。进程中所包含的一个或多个执行单元称为线程（thread）。进程还拥有一个私有的虚拟地址空间，该空间仅能被它所包含的线程访问。线程只能归属于一个进程并且它只能访问该进程所拥有的资源。当操作系统创建一个进程后，该进程会自动申请一个名为主线程或首要线程的线程。操作系统中有若干个线程在"同时"运行。通常，操作系统上运行的每一个应用程序都运行在一个进程中，例如:QQ,IE等等。

注:进程并不是真正意义上的同时运行，而是并发运行。后面我们会具体说明。

**1.1.2. 什么是线程**

一个线程是进程的一个顺序执行流。同类的多个线程共享一块内存空间和一组系统资源，线程本身有一个供程序执行时的堆栈。线程在切换时负荷小，因此，线程也被称为轻负荷进程。一个进程中可以包含多个线程。

注:切换——线程并发时的一种现象，后面讲解并发时会具体说明。

**1.1.3. 进程与线程的区别**

一个进程至少有一个线程。线程的划分尺度小于进程，使得多线程程序的并发性高。另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程在执行过程中与进程的区别在于每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用来实现进程的调度和管理以及资源分配。

**1.1.4. 线程使用的场合**

线程通常用于在一个程序中需要同时完成多个任务的情况。我们可以将每个任务定义为一个线程，使他们得以一同工作。

例如我们在玩某个游戏时，这个游戏由操作系统运行，所以其运行在一个独立的进程中，而在游戏中我们会听到某些背景音乐，某个角色在移动，出现某些绚丽的动画效果等，这些在游戏中都是同时发生的，但实际上，播放音乐是在一个线程中独立完成的，移动某个角色，播放某些特效也都是在独立的线程中完成的。这些事情我们无法在单一线程中完成。

也可以用于在单一线程中可以完成，但是使用多线程可以更快的情况。比如下载文件。

比如迅雷，我们尝尝会开到它会打开很多个节点来同时下载一个文件。

**1.1.5. 并发原理**

通过上面几节知识我们知道进程与线程都是并发运行的，那么什么是并发呢?

多个线程或进程”同时”运行只是我们感官上的一种表现。事实上进程和线程是并发运行的，OS的线程调度机制将时间划分为很多时间片段（时间片），尽可能均匀分配给正在运行的程序，获取CPU时间片的线程或进程得以被执行，其他则等待。而CPU则在这些进程或线程上来回切换运行。微观上所有进程和线程是走走停停的，宏观上都在运行，这种都运行的现象叫并发，但是不是绝对意义上的“同时发生。

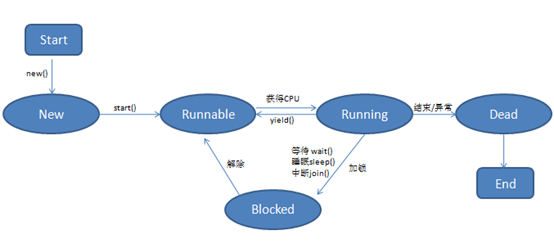
注1:之所以这样做是因为CPU只有一个，同一时间只能做一件事情。但随着计算机的发展，出现了多核心CPU，例如两核心的CPU可以实现真正意义上的2线程同时运行，但因为CPU的时间片段分配给那个进程或线程是由线程调度决定，所以不一定两个线程是属于同一个进程的，无论如何我们只需要知道线程或进程是并发运行即可。

注2:OS—Operating System我们称为:操作系统

注3:线程调度机制是OS提供的一个用于并发处理的程序。java虚拟机自身也提供了线程调度机制，用于减轻OS切换线程带来的更多负担。

**1.1.6. 线程状态**

对于程序而言，我们实际上关心的是线程而非进程。通过上面学习的只是，我们了解了什么是线程以及并发的相关知识。那么我们来看看线程在其生命周期中的各个状态:



图示:

New：当我们创建一个线程时，该线程并没有纳入线程调度，其处于一个new状态。

Runnable:当调用线程的start方法后，该线程纳入线程调度的控制，其处于一个可运行状态，等待分配时间片段以并发运行。

Running:当该线程被分配到了时间片段后其被CPU运行，这是该线程处于running状态。

Blocked:当线程在运行过程中可能会出现阻塞现象，比如等待用户输入信息等。但阻塞状态不是百分百出现的，具体要看代码中是否有相关需求。

Dead:当线程的任务全部运行完毕，或在运行过程中抛出了一个未捕获的异常，那么线程结束，等待GC回收

**1.2. 创建线程**

**1.2.1. 使用Thread创建线并启动线程**

java.lang.Thread类是线程类，其每一个实例表示一个可以并发运行的线程。我们可以通过继承该类并重写run方法来定义一个具体的线程。其中重写run方法的目的是定义该线程要执行的逻辑。启动线程时调用线程的start()方法而非直接调用run()方法。start()方法会将当前线程纳入线程调度，使当前线程可以开始并发运行。当线程获取时间片段后会自动开始执行run方法中的逻辑。

例如:

    public class TestThread extends Thread{

        public void run() {

            for(int i=0;i<100;i++){

                System.out.println("我是线程");

            }

        }

    }

创建和启动线程:

…

Thread thread = new TestThread();//实例化线程

thread.start();//启动线程

…

当调用完start()方法后，run方法会很快执行起来。

**1.2.2. 使用Runnable创建并启动线程**

实现Runnable接口并重写run方法来定义线程体，然后在创建线程的时候将Runnable的实例传入并启动线程。

这样做的好处在于可以将线程与线程要执行的任务分离开减少耦合，同时java是单继承的，定义一个类实现Runnable接口这样的做法可以更好的去实现其他父类或接口。因为接口是多继承关系。

例如:

public class TestRunnable implements Runnable{

public void run() {

for(int i=0;i<100;i++){

System.out.println("我是线程");

}

}

}

启动线程的方法:

…

Runnable runnable = new TestRunnable();

Thread thread = new Thread(runnable);//实例化线程并传入线程体

thread.start();//启动线程

…

**1.2.3. 使用内部类创建线程**

通常我们可以通过匿名内部类的方式创建线程，使用该方式可以简化编写代码的复杂度，当一个线程仅需要一个实例时我们通常使用这种方式来创建。

例如：

* 继承Thread方式:

Thread thread = new Thread(){ //匿名类方式创建线程

public void run(){

//线程体

}

};

thread.start();//启动线程

* Runnable方式:

    Runnable runnable = new Runnable(){    //匿名类方式创建线程

        public void run(){

        }

    };

    Thread thread = new Thread(runnable);

    thread.start();//启动线程

**1.3. 线程操作API**

**1.3.1. Thread.currentThread方法**

Thread的静态方法currentThread方法可以用于获取运行当前代码片段的线程。

Thread current = Thread.currentThread();

**1.3.2. 获取线程信息**

Thread提供了获取线程信息的相关方法:

long getId():返回该线程的标识符

String getName():返回该线程的名称

int getPriority():返回线程的优先级

Thread.state getState():获取线程的状态

boolean isAlive():测试线程是否处于活动状态

boolean isDaemon():测试线程是否为守护线程

boolean isInterrupted():测试线程是否已经中断

例如:

public class TestThread {

public static void main(String[] args) {

Thread current = Thread.currentThread();

long id = current.getId();

String name = current.getName();

int priority = current.getPriority();

Thread.State state = current.getState();

boolean isAlive = current.isAlive();

boolean isDaemon = current.isDaemon();

boolean isInterrupt = current.isInterrupted();

System.out.println("id:"+id);

System.out.println("name:"+name);

System.out.println("priority:"+priority);

System.out.println("state:"+state);

System.out.println("isAlive:"+isAlive);

System.out.println("isDaemon:"+isDaemon);

System.out.println("isInterrupt:"+isInterrupt);

}

}

**1.3.3. 线程优先级**

线程的切换是由线程调度控制的，我们无法通过代码来干涉，但是我们可以通过提高线程的优先级来最大程度的改善线程获取时间片的几率。

线程的优先级被划分为10级，值分别为1-10，其中1最低，10最高。线程提供了3个常量来表示最低，最高，以及默认优先级:

Thread.MIN\_PRIORITY,

Thread.MAX\_PRIORITY,

Thread.NORM\_PRIORITY

设置优先级的方法为:

void setPriority(int priority)

**1.3.4. 守护线程**

守护线程与普通线程在表现上没有什么区别，我们只需要通过Thread提供的方法来设定即可:

void setDaemon(boolean )

当参数为true时该线程为守护线程。

守护线程的特点是，当进程中只剩下守护线程时，所有守护线程强制终止。

GC就是运行在一个守护线程上的。

需要注意的是，设置线程为后台线程要在该线程启动前设置。

Thread daemonThread = new Thread();

daemonThread.setDaemon(true);

daemonThread.start();

**1.3.5. sleep方法**

Thread的静态方法sleep用于使当前线程进入阻塞状态:

static void sleep(long ms)

该方法会使当前线程进入阻塞状态指定毫秒，当指定毫秒阻塞后，当前线程会重新进入Runnable状态，等待分配时间片。

该方法声明抛出一个InterruptException。所以在使用该方法时需要捕获这个异常。

例如:电子钟程序

public static void main(String[] args) {

SimpleDateFormat sdf= new SimpleDateFormat("hh:mm:ss");

while(true){

System.out.println(sdf.format(new Date()));

try {

Thread.sleep(1000);//每输出一次时间后阻塞1秒钟

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

注:改程序可能会出现"跳秒"现象，因为阻塞一秒后线程并非是立刻回到running状态，而是出于runnable状态，等待获取时间片。那么这段等待时间就是"误差"。所以以上程序并非严格意义上的每隔一秒钟执行一次输出。

**1.3.6. yield方法**

Thread的静态方法yield:

static void yield()

该方法用于使当前线程主动让出当次CPU时间片回到Runnable状态，等待分配时间片。

**1.3.7. join方法**

Thread的方法join:

void join()

该方法用于等待当前线程结束。此方法是一个阻塞方法。

该方法声明抛出InterruptException。

例如:

public static void main(String[] args) {

final Thread t1 = new Thread(){

public void run(){

//一些耗时的操作

}

};

Thread t2 = new Thread(){

public void run(){

try {

t1.join();//这里t2线程会开始阻塞，直到t1线程的run方法执行完毕

} catch (InterruptedException e) {

e.printStackTrace();

}

//以下是当前线程的任务代码,只有t1线程运行完毕才会运行。

}

};

}

**1.4. 线程同步**

**1.4.1. synchronized关键字**

多个线程并发读写同一个临界资源时候会发生”线程并发安全问题”

常见的临界资源:

多线程共享实例变量

多线程共享静态公共变量

若想解决线程安全问题，需要将异步的操作变为同步操作。

所谓异步操作是指多线程并发的操作，相当于各干各的。

所谓同步操作是指有先后顺序的操作，相当于你干完我再干。

而java中有一个关键字名为:synchronized,该关键字是同步锁，用于将某段代码变为同步操作，从而解决线程并发安全问题。

**1.4.2. 锁机制**

Java提供了一种内置的锁机制来支持原子性：

同步代码块(synchronized 关键字 )，同步代码块包含两部分：一个作为锁的对象的引用，一个作为由这个锁保护的代码块。

synchronized (同步监视器—锁对象引用){

//代码块

}

若方法所有代码都需要同步也可以给方法直接加锁。

每个Java对象都可以用做一个实现同步的锁，线程进入同步代码块之前会自动获得锁，并且在退出同步代码块时释放锁，而且无论是通过正常路径退出锁还是通过抛异常退出都一样，获得内置锁的唯一途径就是进入由这个锁保护的同步代码块或方法。

**1.4.3. 选择合适的锁对象**

使用synchroinzed需要对一个锁对象上锁以保证线程同步。

那么这个锁对象应当注意:多个需要同步的线程在访问该同步块时，看到的应该是同一个锁对象引用。否则达不到同步效果。 通常我们会使用**this**来作为锁对象。

**1.4.4. 选择合适的锁范围**

在使用同步块时，应当尽量在允许的情况下减少同步范围，以提高并发的执行效率。

**1.4.5. 静态方法锁**

当我们对一个静态方法加锁，如:

public synchronized static void xxx(){

….

}

那么该方法锁的对象是类对象。每个类都有唯一的一个类对象。获取类对象的方式:类名.class

静态方法与非静态方法同时声明了synchronized，他们之间是非互斥关系的。原因在于，静态方法锁的是类对象而非静态方法锁的是当前方法所属对象。

**1.4.6. wait和notify**

多线程之间需要协调工作。

例如，浏览器的一个显示图片的 displayThread想要执行显示图片的任务，必须等待下载线程downloadThread将该图片下载完毕。如果图片还没有下载完，displayThread可以暂停，当downloadThread完成了任务后，再通知displayThread“图片准备完毕，可以显示了”，这时，displayThread继续执行。

以上逻辑简单的说就是：如果条件不满足，则等待。当条件满足时，等待该条件的线程将被唤醒。在Java中，这个机制的实现依赖于wait/notify。等待机制与锁机制是密切关联的。

**2.1. 线程同步**

**2.1.1. 线程安全API与非线程安全API**

之前学习的API中就有设计为线程安全与非线程安全的类:

StringBuffer 是同步的 synchronized append();

StringBuilder 不是同步的 append();

相对而言StringBuffer在处理上稍逊于StringBuilder,但是其是线程安全的。当不存在并发时首选应当使用StringBuilder。

同样的:

Vector 和 Hashtable 是线程安全的而ArrayList 和 HashMap则不是线程安全的。

对于集合而言，Collections提供了几个静态方法，可以将集合或Map转换为线程安全的:

例如:

Collections.synchronizedList() :获取线程安全的List集合

Collections.synchronizedMap():获取线程安全的Map

List<String> list = new ArrayList<String>();

list.add("A");

list.add("B");

list.add("C");

list = Collections.synchronizedList(list);//将ArrayList转换为线程安全的集合

System.out.println(list);//[A,B,C] 可以看出，原集合中的元素也得以保留

...

**2.1.2. 使用ExecutorService实现线程池**

当一个程序中若创建大量线程，并在任务结束后销毁，会给系统带来过度消耗资源，以及过度切换线程的危险，从而可能导致系统崩溃。为此我们应使用线程池来解决这个问题。

ExecutorService是java提供的用于管理线程池的类。

线程池有两个主要作用:控制线程数量/重用线程

线程池的概念：首先创建一些线程，它们的集合称为线程池，当服务器接受到一个客户请求后，就从线程池中取出一个空闲的线程为之服务，服务完后不关闭该线程，而是将该线程还回到线程池中。

在线程池的编程模式下，任务是提交给整个线程池，而不是直接交给某个线程，线程池在拿到任务后，它就在内部找有无空闲的线程，再把任务交给内部某个空闲的线程，任务是提交给整个线程池，一个线程同时只能执行一个任务，但可以同时向一个线程池提交多个任务

线程池有以下几种实现策略:

Executors.newCachedThreadPool()

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。

Executors.newFixedThreadPool(int nThreads)

创建一个可重用固定线程集合的线程池，以共享的无界队列方式来运行这些线程。

Executors.newScheduledThreadPool(int corePoolSize)

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

Executors.newSingleThreadExecutor()

创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

可以根据实际需求来使用某种线程池。例如，创建一个有固定线程数量的线程池：

ExecutorService threadPool = Executors.newFixedThreadPool(30);//创建具有30个线程的线程池

Runnable r1 = new Runable(){

public void run(){

//线程体

}

};

threadPool.execute(r1);//将任务交给线程池，其会分配空闲线程来运行这个任务。

...

**1.1.3. 使用BlockingQueue**

BlockingQueue是双缓冲队列。

在多线程并发时，若需要使用队列，我们可以使用Queue，但是要解决一个问题就是同步，但同步操作会降低并发对Queue操作的效率。

BlockingQueue内部使用两条队列，可允许两个线程同时向队列一个做存储，一个做取出操作。在保证并发安全的同时提高了队列的存取效率。

双缓冲队列有一下几种实现:

ArrayBlockingDeque:规定大小的BlockingDeque,其构造函数必须带一个int参数来指明其大小.其所含的对象是以FIFO(先入先出)顺序排序的。

LinkedBlockingDeque:大小不定的BlockingDeque,若其构造函数带一个规定大小的参数,生成的BlockingDeque有大小限制,若不带大小参数,所生成的BlockingDeque的大小由Integer.MAX\_VALUE来决定.其所含的对象是以FIFO(先入先出)顺序排序的。

PriorityBlockingDeque:类似于LinkedBlockDeque,但其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序。

SynchronousQueue:特殊的BlockingQueue,对其的操作必须是放和取交替完成的。