

实战体会 Java 多线程编程精要

2008-08-19 作者: 来源: CSDN 阅读 130 次

[摘要]本文说明了在 Java 程序中如何使用线程。像是否应该使用线程这样的更重要的问题在很大程度上取决于手头的应用程序。

[关键字]Java 多线程编程

在 Java 程序中使用多线程要比在 C 或 C++ 中容易得多，这是因为 Java 编程语言提供了语言级的支持。本文通过简单的编程示例来说明 Java 程序中的多线程是多么直观。读完本文以后，用户应该能够编写简单的多线程程序。

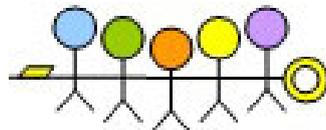
为什么会排队等待？

下面的这个简单的 Java 程序完成四项不相关的任务。这样的程序有单个控制线程，控制在这四个任务之间线性地移动。此外，因为所需的资源（打印机、磁盘、数据库和显示屏）-- 由于硬件和软件的限制都有内在的潜伏时间，所以每项任务都包含明显的等待时间。因此，程序在访问数据库之前必须等待打印机完成打印文件的任务，等等。如果您正在等待程序的完成，则这是对计算资源和您的时间的一种拙劣使用。改进此程序的一种方法是使它成为多线程的。

四项不相关的任务

```
class myclass {  
  
    static public void main(String args[]) {  
  
        print_a_file();  
  
        manipulate_another_file();  
  
        access_database();  
  
        draw_picture_on_screen();  
  
    }  
  
}
```

在本例中，每项任务在开始之前必须等待前一项任务完成，即使所涉及的任务毫不相关也是这样。但是，在现实生活中，我们经常使用多线程模型。我们在



处理某些任务的同时也可以让孩子、配偶和父母完成别的任务。例如，我在写信的同时可能打发我的儿子去邮局买邮票。用软件术语来说，这称为多个控制（或执行）线程。

可以用两种不同的方法来获得多个控制线程：

多个进程

在大多数操作系统中都可以创建多个进程。当一个程序启动时，它可以为即将开始的每项任务创建一个进程，并允许它们同时运行。当一个程序因等待网络访问或用户输入而被阻塞时，另一个程序还可以运行，这样就增加了资源利用率。但是，按照这种方式创建每个进程要付出一定的代价：设置一个进程要占用相当一部分处理器时间和内存资源。而且，大多数操作系统不允许进程访问其他进程的内存空间。因此，进程间的通信很不方便，并且也不会将它自己提供给容易的编程模型。

线程

线程也称为轻型进程 (LWP)。因为线程只能在单个进程的作用域内活动，所以创建线程比创建进程要廉价得多。这样，因为线程允许协作和数据交换，并且在计算资源方面非常廉价，所以线程比进程更可取。线程需要操作系统的支持，因此不是所有的机器都提供线程。Java 编程语言，作为相当新的一种语言，已将线程支持与语言本身合为一体，这样就对线程提供了强健的支持。

使用 Java 编程语言实现线程

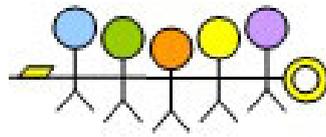
Java 编程语言使多线程如此简单有效，以致于某些程序员说它实际上是自然的。尽管在 Java 中使用线程比在其他语言中要容易得多，仍然有一些概念需要掌握。要记住的一件重要的事情是 main() 函数也是一个线程，并可用来做有用的工作。程序员只有在需要多个线程时才需要创建新的线程。

Thread 类

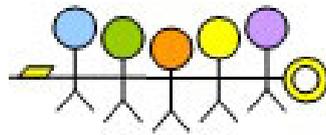
Thread 类是一个具体的类，即不是抽象类，该类封装了线程的行为。要创建一个线程，程序员必须创建一个从 Thread 类导出的新类。程序员必须覆盖 Thread 的 run() 函数来完成有用的工作。用户并不直接调用此函数；而是必须调用 Thread 的 start() 函数，该函数再调用 run()。下面的代码说明了它的用法：

创建两个新线程

```
import java.util.*;
```



```
class TimePrinter extends Thread {  
  
    int pauseTime;  
  
    String name;  
  
    public TimePrinter(int x, String n) {  
  
        pauseTime = x;  
  
        name = n;  
  
    }  
  
    public void run() {  
  
        while(true) {  
  
            try {  
  
                System.out.println(name + ":" + new  
                Date(System.currentTimeMillis()));  
  
                Thread.sleep(pauseTime);  
  
            } catch(Exception e) {  
  
                System.out.println(e);  
  
            }  
  
        }  
  
    }  
  
    static public void main(String args[]) {  
  
        TimePrinter tp1 = new TimePrinter(1000, "Fast Guy");  
  
        tp1.start();  
  
        TimePrinter tp2 = new TimePrinter(3000, "Slow Guy");
```



```
tp2.start();  
  
}  
  
}
```

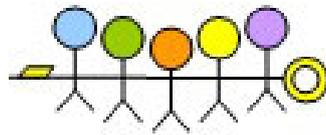
在本例中，我们可以看到一个简单的程序，它按两个不同的时间间隔（1 秒和 3 秒）在屏幕上显示当前时间。这是通过创建两个新线程来完成的，包括 main() 共三个线程。但是，因为有时要作为线程运行的类可能已经是某个类层次的一部分，所以就不能再按这种机制创建线程。虽然在同一个类中可以实现任意数量的接口，但 Java 编程语言只允许一个类有一个父类。同时，某些程序员避免从 Thread 类导出，因为它强加了类层次。对于这种情况，就要 runnable 接口。

Runnable 接口

此接口只有一个函数，run()，此函数必须由实现了此接口的类实现。但是，就运行这个类而论，其语义与前一个示例稍有不同。我们可以用 runnable 接口改写前一个示例。（不同的部分用黑体表示。）

创建两个新线程而不强加类层次

```
import java.util.*;  
  
class TimePrinter implements Runnable {  
  
    int pauseTime;  
  
    String name;  
  
    public TimePrinter(int x, String n) {  
  
        pauseTime = x;  
  
        name = n;  
  
    }  
  
    public void run() {  
  
        while(true) {
```

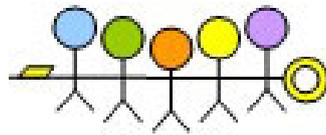


```
try {  
  
    System.out.println(name + ":" + new  
  
    Date(System.currentTimeMillis()));  
  
    Thread.sleep(pauseTime);  
  
    } catch(Exception e) {  
  
    System.out.println(e);  
  
    }  
  
    }  
  
    }  
  
    }  
  
    static public void main(String args[]) {  
  
    Thread t1 = new Thread(new TimePrinter(1000, "Fast Guy"));  
  
    t1.start();  
  
    Thread t2 = new Thread(new TimePrinter(3000, "Slow Guy"));  
  
    t2.start();  
  
    }  
  
    }
```

请注意，当使用 `Runnable` 接口时，您不能直接创建所需类的对象并运行它；必须从 `Thread` 类的一个实例内部运行它。许多程序员更喜欢 `Runnable` 接口，因为从 `Thread` 类继承会强加类层次。

`synchronized` 关键字

到目前为止，我们看到的示例都只是以非常简单的方式来利用线程。只有最小的数据流，而且不会出现两个线程访问同一个对象的情况。但是，在大多数有用的程序中，线程之间通常有信息流。试考虑一个金融应用程序，它有一个 `Account` 对象，如下例中所示：



一个银行中的多项活动

```
public class Account {  
  
    String holderName;  
  
    float amount;  
  
    public Account(String name, float amt) {  
  
        holderName = name;  
  
        amount = amt;  
  
    }  
  
    public void deposit(float amt) {  
  
        amount += amt;  
  
    }  
  
    public void withdraw(float amt) {  
  
        amount -= amt;  
  
    }  
  
    public float checkBalance() {  
  
        return amount;  
  
    }  
  
}
```

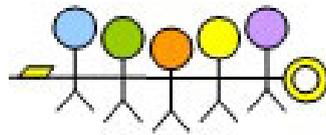
在此代码样例中潜伏着一个错误。如果此类用于单线程应用程序，不会有任何问题。但是，在多线程应用程序的情况下，不同的线程就有可能同时访问同一个 Account 对象，比如说一个联合帐户的所有者在不同的 ATM 上同时进行访问。在这种情况下，存入和支出就可能以这样的方式发生：一个事务被另一个事务覆盖。这种情况将是灾难性的。但是，Java 编程语言提供了一种简单的机制来防止发生这种覆盖。每个对象在运行时都有一个关联的锁。这个锁可通过为方法添

加关键字 `synchronized` 来获得。这样，修订过的 `Account` 对象（如下所示）将不会遭受像数据损坏这样的错误：

对一个银行中的多项活动进行同步处理

```
public class Account {  
  
    String holderName;  
  
    float amount;  
  
    public Account(String name, float amt) {  
  
        holderName = name;  
  
        amount = amt;  
  
    }  
  
    public synchronized void deposit(float amt) {  
  
        amount += amt;  
  
    }  
  
    public synchronized void withdraw(float amt) {  
  
        amount -= amt;  
  
    }  
  
    public float checkBalance() {  
  
        return amount;  
  
    }  
  
}
```

`deposit()` 和 `withdraw()` 函数都需要这个锁来进行操作，所以当在一个函数运行时，另一个函数就被阻塞。请注意，`checkBalance()` 未作更改，它严格是一个读函数。因为 `checkBalance()` 未作同步处理，所以任何其他方法都不会阻塞它，它也不会阻塞任何其他方法，不管那些方法是否进行了同步处理。



Java 编程语言中的高级多线程支持

线程组

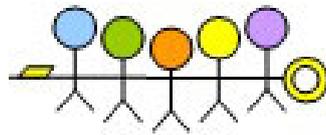
线程是被个别创建的，但可以将它们归类到线程组中，以便于调试和监视。只能在创建线程的同时将它与一个线程组相关联。在使用大量线程的程序中，使用线程组组织线程可能很有帮助。可以将它们看作是计算机上的目录和文件结构。

线程间发信

当线程在继续执行前需要等待一个条件时，仅有 `synchronized` 关键字是不够的。虽然 `synchronized` 关键字阻止并发更新一个对象，但它没有实现线程间发信。`Object` 类为此提供了三个函数：`wait()`、`notify()` 和 `notifyAll()`。以全球气候预测程序为例。这些程序通过将地球分为许多单元，在每个循环中，每个单元的计算都是隔离进行的，直到这些值趋于稳定，然后相邻单元之间就会交换一些数据。所以，从本质上讲，在每个循环中各个线程都必须等待所有线程完成各自的任务以后才能进入下一个循环。这个模型称为 屏蔽同步，下例说明了这个模型：

屏蔽同步

```
public class BSync {  
  
    int totalThreads;  
  
    int currentThreads;  
  
    public BSync(int x) {  
  
        totalThreads = x;  
  
        currentThreads = 0;  
  
    }  
  
    public synchronized void waitForAll() {  
  
        currentThreads++;  
  
        if(currentThreads < totalThreads) {  
  
            try {
```



```
wait();  
  
} catch (Exception e) {}  
  
}  
  
else {  
  
currentThreads = 0;  
  
notifyAll();  
  
}  
  
}  
  
}
```

当对一个线程调用 `wait()` 时，该线程就被有效阻塞，直到另一个线程对同一个对象调用 `notify()` 或 `notifyAll()` 为止。因此，在前一个示例中，不同的线程在完成它们的工作以后将调用 `waitForAll()` 函数，最后一个线程将触发 `notifyAll()` 函数，该函数将释放所有的线程。第三个函数 `notify()` 只通知一个正在等待的线程，当对每次只能由一个线程使用的资源进行访问限制时，这个函数很有用。但是，不可能预知哪个线程会获得这个通知，因为这取决于 Java 虚拟机 (JVM) 调度算法。

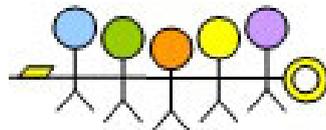
将 CPU 让给另一个线程

当线程放弃某个稀有的资源（如数据库连接或网络端口）时，它可能调用 `yield()` 函数临时降低自己的优先级，以便某个其他线程能够运行。

守护线程

有两类线程：用户线程和守护线程。用户线程是那些完成有用工作的线程。守护线程是那些仅提供辅助功能的线程。Thread 类提供了 `setDaemon()` 函数。Java 程序将运行到所有用户线程终止，然后它将破坏所有的守护线程。在 Java 虚拟机 (JVM) 中，即使在 `main` 结束以后，如果另一个用户线程仍在运行，则程序仍然可以继续运行。

避免不提倡使用的方法



不提倡使用的方法是为支持向后兼容性而保留的那些方法，它们在以后的版本中可能出现，也可能不出现。Java 多线程支持在版本 1.1 和版本 1.2 中做了重大修订，`stop()`、`suspend()` 和 `resume()` 函数已不提倡使用。这些函数在 JVM 中可能引入微妙的错误。虽然函数名可能听起来很诱人，但请抵制诱惑不要使用它们。

调试线程化的程序

在线程化的程序中，可能发生的某些常见而讨厌的情况是死锁、活锁、内存损坏和资源耗尽。

死锁

死锁可能是多线程程序最常见的问题。当一个线程需要一个资源而另一个线程持有该资源的锁时，就会发生死锁。这种情况通常很难检测。但是，解决方案却相当好：在所有的线程中按相同的次序获取所有资源锁。例如，如果有四个资源 A、B、C 和 D，并且一个线程可能要获取四个资源中任何一个资源的锁，则请确保在获取对 B 的锁之前首先获取对 A 的锁，依此类推。如果“线程 1”希望获取对 B 和 C 的锁，而“线程 2”获取了 A、C 和 D 的锁，则这一技术可能导致阻塞，但它永远不会在这四个锁上造成死锁。

活锁

当一个线程忙于接受新任务以致它永远没有机会完成任何任务时，就会发生活锁。这个线程最终将超出缓冲区并导致程序崩溃。试想一个秘书需要录入一封信，但她一直在忙于接电话，所以这封信永远不会被录入。

内存损坏

如果明智地使用 `synchronized` 关键字，则完全可以避免内存错误这种气死人的问题。

资源耗尽

某些系统资源是有限的，如文件描述符。多线程程序可能耗尽资源，因为每个线程都可能希望有一个这样的资源。如果线程数相当大，或者某个资源的候选线程数远远超过了可用的资源数，则最好使用资源池。一个最好的示例是数据库连接池。只要线程需要使用一个数据库连接，它就从池中取出一个，使用以后再将它返回池中。资源池也称为资源库。

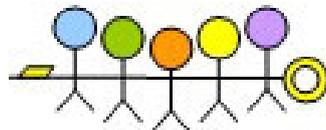
调试大量的线程

有时一个程序因为有大量的线程在运行而极难调试。在这种情况下，下面的这个类可能会派上用场：

```
public class Probe extends Thread {  
  
    public Probe() {}  
  
    public void run() {  
  
        while(true) {  
  
            Thread[] x = new Thread[100];  
  
            Thread.enumerate(x);  
  
            for(int i=0; i<100; i++) {  
  
                Thread t = x[i];  
  
                if(t == null)  
  
                    break;  
  
                else  
  
                    System.out.println(t.getName() + "\t" + t.getPriority()  
  
                    + "\t" + t.isAlive() + "\t" + t.isDaemon());  
  
                }  
  
            }  
  
        }  
  
    }  
  
}
```

限制线程优先级和调度

Java 线程模型涉及可以动态更改的线程优先级。本质上，线程的优先级是从 1 到 10 之间的一个数字，数字越大表明任务越紧急。JVM 标准首先调用优先级较高的线程，然后才调用优先级较低的线程。但是，该标准对具有相同优先级的线程的处理是随机的。如何处理这些线程取决于基层的操作系统策略。在



某些情况下，优先级相同的线程分时运行；在另一些情况下，线程将一直运行到结束。请记住，Java 支持 10 个优先级，基层操作系统支持的优先级可能要少得多，这样会造成一些混乱。因此，只能将优先级作为一种很粗略的工具使用。最后的控制可以通过明智地使用 `yield()` 函数来完成。通常情况下，请不要依靠线程优先级来控制线程的状态。

小结

本文说明了在 Java 程序中如何使用线程。像是否应该使用线程这样的更重要的问题在很大程度上取决于手头的应用程序。决定是否在应用程序中使用多线程的一种方法是，估计可以并行运行的代码量。并记住以下几点：

使用多线程不会增加 CPU 的能力。但是如果使用 JVM 的本地线程实现，则不同的线程可以在不同的处理器上同时运行（在多 CPU 的机器中），从而使多 CPU 机器得到充分利用。

如果应用程序是计算密集型的，并受 CPU 功能的制约，则只有多 CPU 机器能够从更多的线程中受益。

当应用程序必须等待缓慢的资源（如网络连接或数据库连接）时，或者当应用程序是非交互式的时，多线程通常是有利的。

基于 Internet 的软件有必要是多线程的；否则，用户将感觉应用程序反映迟钝。例如，当开发要支持大量客户机的服务器时，多线程可以使编程较为容易。在这种情况下，每个线程可以为不同的客户或客户组服务，从而缩短了响应时间。

某些程序员可能在 C 和其他语言中使用过线程，在那些语言中对线程没有语言支持。这些程序员可能通常都被搞得对线程失去了信心。