**JAVA基础**

# 进程

* 程序与进程

程序:开发出来的软件，还没有被CPU执行的软件

进程:当软件被加载到内存由CPU进行执行，即变成一个进程(系统按进程分配资源和进行调度)

进程特征：

*1.独立性:进程是系统中独立存在的实体，它可以拥有自己独立的资源，并有自己的有私有地址空间。*

*2.动态性:程序是一个静态的指令集合,进程是一个正在系统中活动的指令集合*

*3.并发性:多个进程可以在单个处理器上并发执行，多个进程之间不会互相影响*

# 线程

线程是进程的组成部分,一个进程可以拥有多个线程，一个线程必须有一个父进程。

线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不拥有系统资源（间接拿到系统资源）

父进程的其他线程共享进程所拥有的全部资源

线程是独立运行的，它并不知道进程中是否还有其他线程存在。

线程执行是抢占式的，当前运行的线程在任何时候都可能被挂起，以便另外一个线程可以运行

一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

总结:一个程序运行后至少有一个进程，一个进程里可以包含一个或多个线程

## 实现方式

* 继承Thread类创建线程类

通过继承Thread类来创建并启动多线程的步骤如下:

*1.定义Thread类的子类，并重写该类的run()方法,该run()方法的方法体就代表了线程需要完成的任务，因此把run()方法称为线程执行体。*

*2.创建Thread子类的实例，创建了线程对象*

*3.调用线程对象的start()方法启动该线程*

4、start()方法的调用后并不是立即执行多线程代码，而是使得该线程变为可运行态（Runnable），什么时候运行是由操作系统决定的。

=====

* 实现Runnable接口创建线程类

实现Runnable接口来创建并启动多线程的步骤如下.

*1.定义Runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体*

*2.创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象*

*3.调用线程对象的start()方法来启动该线程*

4、从程序运行的结果可以发现，多线程程序是乱序执行。因此，只有乱序执行的代码才有必要设计为多线程。

* 两种实现方式的区别与联系

在程序开发中只要是多线程肯定永远以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类有如下好处：

*1.避免继承的局限，一个类可以实现多个接口。*

*2.适合于资源的共享*

|  |
| --- |
| **public** **class** ThreadDemo **extends** Thread {  // 线程执行体，里面写上我们想让线程干的事情  @Override  **public** **void** run() {    **try** {  Thread.*sleep*(1500);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    System.***out***.println(getName());  System.***out***.println("123");    }  } |
| **public** **class** RunnableDemo **implements** Runnable {  **int** i = 0;  // 可以认为是线程执行体  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());    i++;  System.***out***.println(i);  }  } |
| **public** **class** Main {  // main我们一般认为他是主线程中的方法  // 线程 独立运行，互不影响  // 主线程与ThreadDemo线程并列执行  // 线程执行完之后自动结束  **public** **static** **void** main(String[] args) {      System.***out***.println(Thread.*currentThread*().getName() + "===");    ThreadDemo td = **new** ThreadDemo();  System.***out***.println(td.isAlive());// false  // 启动线程，会调用线程中重写的run方法  td.start();  System.***out***.println(td.isAlive());// true    System.***out***.println("main");    // 资源共享，多个线程只要操作同一个实例，那么实例成员便是多个线程共享的  RunnableDemo rd = **new** RunnableDemo();  Thread t = **new** Thread(rd,"run1");  t.start();  Thread t1 = **new** Thread(rd,"run2");  t1.start();    // callable的实现，能够更好的处理并发 since jdk1.5  CallableDemo callDemo = **new** CallableDemo();  FutureTask<String> ft = **new** FutureTask<>(callDemo);  Thread th = **new** Thread(ft);  // 当线程启动时，回去调用CallableDemo中的call方法  th.start();    **try** {  // 获取call方法的返回值,get是一种阻塞型的方法  // 也就是说必须等待get方法返回之后，才能继续向下执行  System.***out***.println(ft.get());  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (ExecutionException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    System.***out***.println("main end");  // 线程死亡后不要在调用start方法，否则会抛出  // java.lang.IllegalThreadStateException异常  // t.start();  // 线程是否已经死亡  System.***out***.println(t.isAlive());      }  } |

线程生命周期5种状态

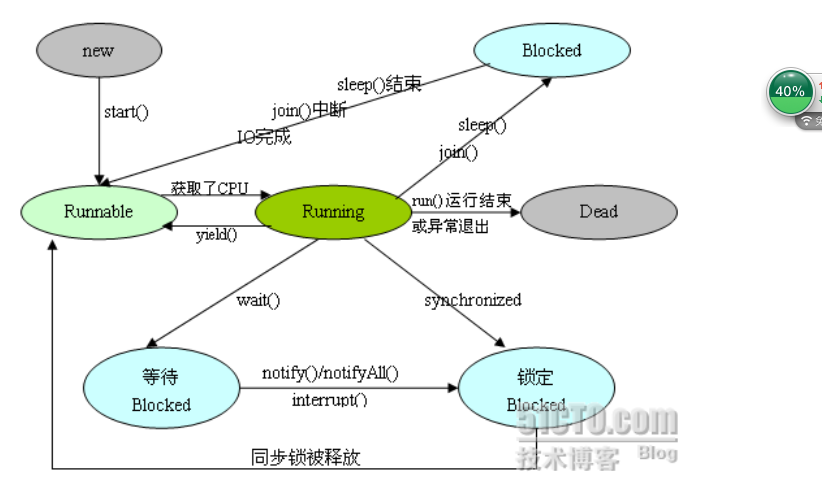
1.新建(New)

2.就绪(Runnable)

3.运行(Running)

4.阻塞(Blocked)

5.死亡(Dead)



## Join

|  |
| --- |
| **public** **class** JoinDemo {  **public** **static** **void** main(String[] args) {  OneRunnable or = **new** OneRunnable();  Thread t1 = **new** Thread(or);  t1.start();  }  }  **class** OneRunnable **implements** Runnable {  @Override  **public** **void** run() {  TwoRunnable tr = **new** TwoRunnable();  Thread t2 = **new** Thread(tr);  System.***out***.println("=======");  **try** {    t2.start();  // 新加入的这个线程会阻塞当前线程，待执行完毕后，在执行当前线程  // t2.join();  // 新加入的这个线程会阻塞当前线程，待等待2秒或新加入的线程执行完毕，  // 再执行当前线程  t2.join(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println("one");    }    }  **class** TwoRunnable **implements** Runnable {  @Override  **public** **void** run() {    **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println("two");      }    } |

## 优先级

MAX\_PRIORITY:值是10

MIN\_PRIORITY:值是1

NORM\_PRIORITY:值是5(这是默认值)

|  |
| --- |
| **public** **class** PriorityDemo {  **public** **static** **void** main(String[] args) {  PriorityRunnable pr = **new** PriorityRunnable();  Thread t1 = **new** Thread(pr, "t1");  // 优先级并不能保证高的一定比低的先执行，优先可以理解一个概率  t1.setPriority(Thread.***MAX\_PRIORITY***);    PriorityRunnable pr1 = **new** PriorityRunnable();  Thread t2 = **new** Thread(pr1, "t2");  t2.setPriority(Thread.***MIN\_PRIORITY***);  t2.start();  t1.start();  }  }  **class** PriorityRunnable **implements** Runnable {  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 1000; i++) {    System.***out***.println(Thread.*currentThread*().getName() + "--" + i);  Thread.*yield*();  }  }  } |

## 后台线程

|  |
| --- |
| **public** **class** DeamonDemo {  **private** String name;    **public** **static** **void** main(String[] args) {  DeamonDemo dd = **new** DeamonDemo();  dd.name = "";    System.***out***.println("===========");  DeamonRunnable dea = dd.**new** DeamonRunnable();  Thread thread = **new** Thread(dea);  // 设置为true表示当前线程为后台线程，否则为前台线程，默认是前台线程  // 如果所有前台线程死亡，后台线程会自动死亡  thread.setDaemon(**true**);  thread.start();    DeamonRunnable dea1 = dd.**new** DeamonRunnable();  Thread thread1 = **new** Thread(dea1);  thread1.start();  }    **class** DeamonRunnable **implements** Runnable {  @Override  **public** **void** run() {  **int** i = 0;  **while** (i < 20) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(i);  i ++;  }  }    }  } |

## 多线程并发问题

|  |
| --- |
| **public** **class** Account {  **private** String accountId;  **private** **double** balance;        **public** Account() {  **super**();  // **TODO** Auto-generated constructor stub  }  **public** Account(String accountId, **double** balance) {  **super**();  **this**.accountId = accountId;  **this**.balance = balance;  }  **public** String getAccountId() {  **return** accountId;  }  **public** **void** setAccountId(String accountId) {  **this**.accountId = accountId;  }  **public** **double** getBalance() {  **return** balance;  }  **public** **void** setBalance(**double** balance) {  **this**.balance = balance;  }  @Override  **public** String toString() {  **return** "Account [accountId=" + accountId + ", balance=" + balance + "]";  }  } |
| **public** **class** SyncProm {  // synchronized解决线程并发资源共享信息不一致的问题  **public** **static** **void** main(String[] args) {  Account account = **new** Account("123456", 20000);  DrawRunnable dr = **new** DrawRunnable(account);  Thread t1 = **new** Thread(dr);  Thread t2 = **new** Thread(dr);  // 线程并发，所带来的数据信息不一致的问题  t1.start();  t2.start();  }  }  **class** DrawRunnable **implements** Runnable {  **private** Account account;  **public** DrawRunnable(Account account) {  **this**.account = account;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **double** bala = account.getBalance() - 100;  account.setBalance(bala);  System.***out***.println(bala);  }  }  } |