**20180606**

**22 多线程**

* 22.1 进程与线程

进程(Process)是资源(CPU、内存等)分配的基本单位，是程序执行时的一个实例。程序运行时系统会创建一个进程，为它分配资源，把该进程放入进程就绪队列，进程调度器选中它时就会为它分配CPU时间，程序开始真正运行。

单进程的特定的同一时间段只允许一个程序运行。多进程一个时间段可以运行多个程序，这些程序进行资源的轮流抢占(单核CPU)，同一个时间点只有一个进程运行。

线程(Thread)是程序执行流的最小单位，一个进程可以由多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。

线程的启动速度比进程快许多，多线程进行并发处理性能高于多进程。

* 22.2 继承Thread类实现多线程

一个类继承了java.lang.Thread表示此类是线程的主体类，还需要覆写run()方法，run()方法属于线程的主方法。多线程要执行的内容都在run()方法内定义。

run()方法不能直接调用，因为牵扯到操作系统资源调度问题，使用start()方法启动多线程。

|  |
| --- |
| **class** MyThread **extends** Thread {  **private** String name;  **private** **int** x;  **public** MyThread(String name) {  **this**.name = name;  }  @Override  **public** **void** run() {  **while** (**this**.x < 5) {  System.***out***.println(**this**.name + ": " + **this**.x);  **this**.x++;  }  }  }  **new** MyThread("maki").start();  **new** MyThread("rin").start(); |

结果：

|  |
| --- |
| maki: 0  rin: 0  maki: 1  rin: 1  maki: 2  rin: 2  rin: 3  rin: 4  maki: 3  maki: 4 |

实例化对象调用start()方法，但是执行的是run()方法内容，所有线程交替执行，执行顺序不可控，打印结果随机。

为什么要使用start()方法启动多线程呢?

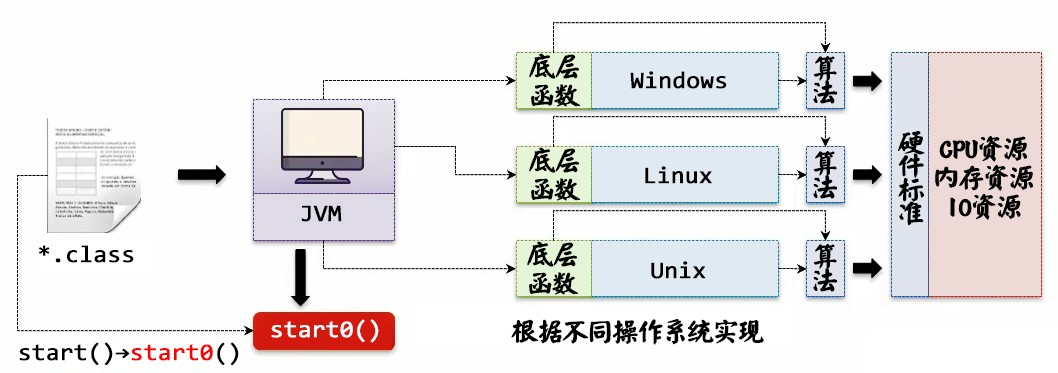
start()方法源代码：

|  |
| --- |
| public synchronized *void* start() {  if (threadStatus != 0) // 线程的状态0表示线程未开始  throw new IllegalThreadStateException();  group.add(this);  *boolean* started = false;  try {  start0();  started = true;  } finally {  try {  if (!started) {  group.threadStartFailed(this);  }  } catch (*Throwable* *ignore*) {}  }  }  private native *void* start0(); // 该方法没有方法体,没有实现 |

一个线程只能被启动一次，如果重复启动抛出IllegalThreadStateException异常。但没有throws声明或try-catch 处理，说明该异常是RuntimeException的子类

start()方法中又调用了start0()方法，此方法使用**native**关键字修饰。

Java支持本地操作系统函数调用，称为JNI (Java Native Interface)技术，但是Java开发中不推荐这样使用，利用JNI可以使用操作系统提供的底层函数。Thread类的start0()表示此方法依赖于不同的操作系统实现。



* 22.3 基于Runnable接口实现多线程

Java继承存在单继承的局限，实现java.lang.Runnable接口也可以实现多线程。

Runnable接口的定义：

|  |
| --- |
| *@FunctionalInterface* // 函数式接口  public interface Runnable {  public abstract *void* run();  } |

将MyThread改为实现Runnable接口：

|  |
| --- |
| **class** MyThread **implements** Runnable {  // 与之前一模一样...  } |

但是此时MyThread没有继承Thread，不能使用start()方法。

Thread类有一个构造方法可以接收Runnable对象作为参数：

|  |
| --- |
| public Thread(*Runnable* target) {  init(null, target, "Thread-" + nextThreadNum(), 0);  } |

启动多线程：

|  |
| --- |
| **new** Thread(**new** MyThread("maki")).start();  **new** Thread(**new** MyThread("rin")).start(); |

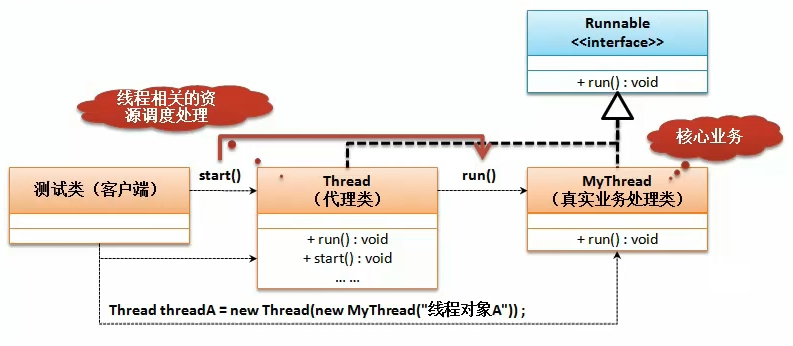
JDK 1.8开始Runnable使用了函数式接口定义，可以使用Lambda表达式定义多线程类。

|  |
| --- |
| **for** (**int** i = 0; i < 3; i++) { // 3个线程  String name = "线程-" + i;  Runnable run = () -> { // Runnable对象  **for** (**int** j = 0; j < 5; j++) {  System.***out***.println(name + ": " + j);  }  };  // 始终使用Thread对象start()方法启动多线程  **new** Thread(run).start();  } |

也可以不定义run变量，直接将其右边传入Thread的构造方法。

* 22.4 Thread类和Runnable接口关系

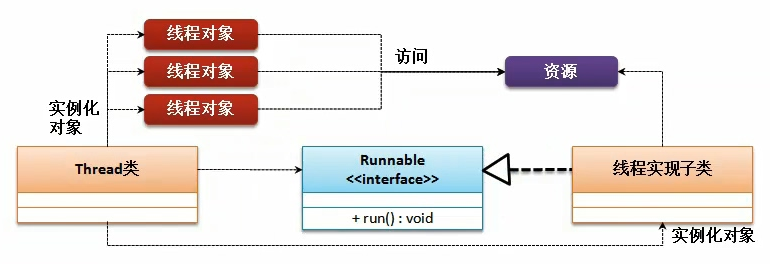
Thread类是Runnable接口的子类



多线程设计使用了代理设计模式的结构，用户自定义的线程主体只是负责核心业务的实现，所有的辅助实现都由Thread类处理。

通过Thread类的构造方法传递Runnable对象时，该对象被Thread的target属性保存。Thread启动多线程调用start()方法，start()调用run()方法，此Thread类的run()方法又去调用Runnable对象的run()方法。

多线程开发本质的多个线程可以进行同一资源的抢占。Thread主要描述线程，Runnable主要描述资源，因为*n*个Thread对象的target属性都指向了同一个Runnable对象。



* 22.5 Callable实现多线程

Runnable接口的缺点是当线程执行完毕无法获取返回值。JDK 1.5提出新的线程实现接口java.util.concurrent.Callable：

|  |
| --- |
| *@FunctionalInterface*  public interface Callable<*V*> {  *V* call() throws Exception;  } |

Callable对象可以作为FutureTask构造方法的参数保存为callable属性。

FutureTask是RunnableFuture接口的子类。

|  |
| --- |
| public class FutureTask<*V*> implements *RunnableFuture*<*V*> {  // ...  public FutureTask(*Callable*<*V*> *callable*) {  if (callable == null)  throw new NullPointerException();  this.callable = callable;  this.state = NEW; // ensure visibility of callable  }  // ...  } |

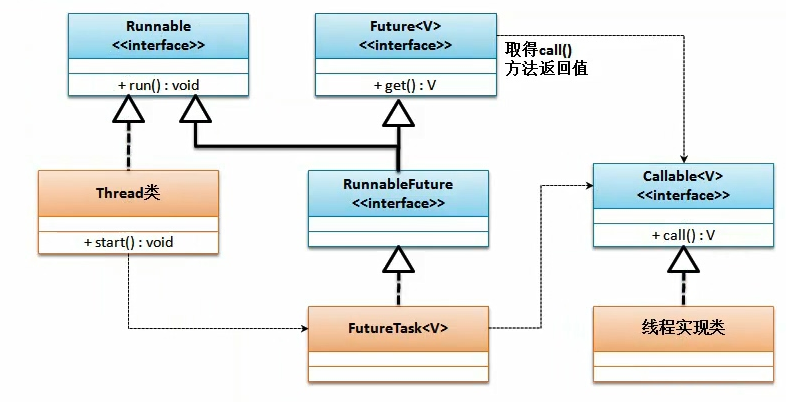
RunnableFuture接口继承于Runnable接口和Future接口。

|  |
| --- |
| public interface Runnable Future<*V*> extends *Runnable*, *Future*<*V*> {  *void* run();  } |

FutureTask类覆写了Future接口的get()方法，可以获取callable属性调用call()方法的返回值。

FutureTask类也是Runnable接口的子类，可以作为Thread构造方法的参数。

关系有点复杂：



示例：用Callable实现龟兔赛跑

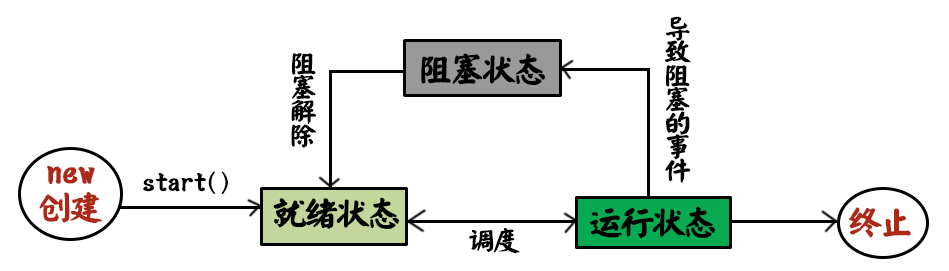
|  |
| --- |
| **class** Race **implements** Callable<Integer> {  **private** String name;  **private** **long** time; // 多少毫秒走一步  **private** **int** step; // 步数  **private** **boolean** flag = **true**; // 设为false结束线程  **public** Race(String name, **long** time) {  **this**.name = name;  **this**.time = time;  }  **public** **void** setFlag(**boolean** flag) {  **this**.flag = flag;  }  @Override  **public** Integer call() **throws** Exception {  **while** (flag) {  Thread.*sleep*(**this**.time);  **this**.step++;  System.***out***.println(Thread.*currentThread*().getName() + " " + **this**.name + ": " + **this**.step);  }  **return** **this**.step;  }  }  // 客户端代码太多了...  Race tortoise = **new** Race("乌龟", 2000);  Race rabbit = **new** Race("兔子", 500);  FutureTask<Integer> task1 = **new** FutureTask<>(tortoise);  FutureTask<Integer> task2 = **new** FutureTask<>(rabbit);  **new** Thread(task1).start();  **new** Thread(task2).start();  Thread.*sleep*(10000); // 跑10秒  tortoise.setFlag(**false**);  rabbit.setFlag(**false**);  System.***out***.println("10秒后, 乌龟:" + task1.get());  System.***out***.println("10秒后, 兔子:" + task2.get()); |

结果：

|  |
| --- |
| 10秒后, 乌龟:5  10秒后, 兔子:20 |

* 22.6 多线程运行状态

定义线程主体类，通过Thread类start()方法启动线程，但并不是调用start()方法线程就开始运行。



1) 任何一个线程对象需要使用Thread类封装，线程启动使用start()方法；但是启动时所有线程进入就绪状态，并没有执行；

2) 等待资源调度，某个线程调度成功则进入运行状态(run()方法)；但是一个线程不可能一直执行下去，执行一段时间之后就会让出资源进入阻塞状态，随后重新回到就绪状态；

3) run()方法执行完毕，线程任务结束，此时进入停止状态。

**20180607**

* 22.7 Thread类常用方法

1. 线程的命名与取得

多线程运行状态不确定，所有线程的名字是个重要的属性。

|  |
| --- |
| public Thread(*Runnable* target, *String* name) // 构造方法可以自定义线程名  public final *void* setName(*String* name) // 设置线程名字  public final *String* getName() // 获取线程名字 |

线程对象的获取不可能只靠this完成，因为线程状态不可控，但是所有线程一定会执行run()方法，则可以考虑获取当前线程。

|  |
| --- |
| public static *Thread* currentThread() // 返回当前正在执行线程的引用 |

示例：自定义线程名字和获取当前线程名

|  |
| --- |
| **class** MyThread **implements** Runnable{  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  }  }  MyThread mt=**new** MyThread();  **new** Thread(mt,"线程1").start(); // 自定义线程名字  **new** Thread(mt).start();  **new** Thread(mt).start();  **new** Thread(mt,"线程2").start(); // 自定义线程名字  mt.run(); // main |

结果：

|  |
| --- |
| 线程1  Thread-0  线程2  Thread-1  main |

如果没有设置线程名字，会自动生成一个不重复的名字。

|  |
| --- |
| // 匿名线程使用类静态属性自动编号  **private** **static** **int** *threadInitNumber*;  **private** **static** **synchronized** **int** nextThreadNum() {  **return** *threadInitNumber*++;  } |

直接执行mt.run()就是在主方法中调用线程对象的run()方法，获得线程名字为main，所以主方法也是一个线程。

每当使用java命令执行程序时就启动了一个JVM的进程，一台电脑可以同时启动多个JVM进程，一个JVM进程都有各自的线程。

主线程可以创建若干子线程，主要将一些复杂逻辑或耗时操作交给子线程处理。

|  |
| --- |
| System.***out***.println("吃饭");  **new** Thread(() -> {  // 模拟耗时操作,耗时操作交给子线程完成  **double** pi = 0;  **double** flag = 1;  **for** (**int** i = 1; i < 1e9; i += 2) {  pi += flag / i;  flag = -flag;  }  pi \*= 4;  System.***out***.println("pi=" + pi);  }).start();  System.***out***.println("睡觉"); |

主线程复杂整体流程，子线程负责处理耗时操作。

2. 线程的休眠 (sleep)

|  |
| --- |
| public static *void* sleep​(*long* millis) throws *InterruptedException*  public static *void* sleep​(*long* millis, *int* nanos) throws *InterruptedException* |

休眠时可能产生中断异常InterruptedException，是Exception的子类，说明该异常必须被处理。

多线程休眠有先后顺序，一个线程休眠会释放执行权，其他线程抢占资源。

|  |
| --- |
| Runnable run = () -> {  **for** (**int** i = 0; i < 10; i++) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ", i=" + i);  }  };  // 5个子线程几乎同时休眠同时唤醒, 因为执行太快了; 差别是每轮打印顺序都不一样  **for** (**int** n = 0; n < 5; n++) {  **new** Thread(run, "hikari - " + n).start();  } |

3. 线程中断 (interrupt)

线程休眠可能会产生中断异常，也就是线程休眠可能被其他线程打断。

|  |
| --- |
| public *boolean* isInterrupted() // 判断线程是否被中断  public *void* interrupt() // 中断该线程 |

示例：main线程中止子线程

|  |
| --- |
| Thread t = **new** Thread(() -> {  System.***out***.println("该睡觉了...");  **try** {  Thread.*sleep*(10000); // 预计休眠10s  System.***out***.println("醒来...");  } **catch** (InterruptedException e) {  System.***out***.println("草泥马, 不要打扰劳资睡觉!");  }  });  t.start();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {}  **if** (!t.isInterrupted()) {  // 如果线程没被中止, 则中止它; main线程中止子线程  System.***out***.println("小伙子, 该醒了!");  t.interrupt();  } |

4. 线程强制执行 (join)

当满足某些条件后，某个线程对象一直独占资源，直到该线程执行结束。

|  |
| --- |
| public final *void* join() throws *InterruptedException* |

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  // sleep()和join()都会抛出异常, 直接在方法上声明吧...  Thread t = **new** Thread(() -> {  **for** (**int** x = 0; x < 100; x++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ", x=" + x);  }  }, "子线程");  t.start();  **for** (**int** x = 0; x < 100; x++) {  **if** (x == 10) {  t.join(); // 主线程等待主线程t执行完再执行  }  Thread.*sleep*(100);  System.***out***.println("【main线程】, x=" + x);  }  } |

5. 线程的礼让 (yield)

线程的礼让是将资源让给其他线程先执行。

|  |
| --- |
| public static *void* yield() |

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  Thread t = **new** Thread(() -> {  **for** (**int** x = 0; x < 100; x++) {  **if** (x%5==0) {  // 每次x为5的倍数时,子线程让出执行权  Thread.*yield*(); // 静态方法  System.***out***.println("\*\*\*\*\*子线程礼让\*\*\*\*\*");  }  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ", x=" + x);  }  }, "子线程");  t.start();  **for** (**int** x = 0; x < 100; x++) {  Thread.*sleep*(100);  System.***out***.println("【main线程】, x=" + x);  }  } |

礼让执行每次调用yield()方法只会礼让一次。该方法很少使用。

6. 线程优先级

理论上线程的优先级越高越可以先执行(抢占到资源)。

|  |
| --- |
| public final *void* setPriority(*int* newPriority)  public final *int* getPriority() |

优先级定义使用int，Thread类定义三个与优先级的int常量：

|  |
| --- |
| public static final *int* MAX\_PRIORITY = 10; // 最高优先级  public static final *int* NORM\_PRIORITY = 5; // 中等优先级  public static final *int* MIN\_PRIORITY = 1; // 最低优先级 |

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  System.***out***.println("主线程优先级: " + Thread.*currentThread*().getPriority());  Runnable run = () -> {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + " " + i);  }  };  Thread t1 = **new** Thread(run, "子线程01");  Thread t2 = **new** Thread(run, "子线程02");  Thread t3 = **new** Thread(run, "子线程03");  System.***out***.println("新创建的线程对象的优先级: " + t3.getPriority());  t1.setPriority(Thread.***MAX\_PRIORITY***); // 将子线程01优先级设为最高  t1.start();  t2.start();  t3.start();  }  } |

子线程和默认创建的子线程优先级都是5。