**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?

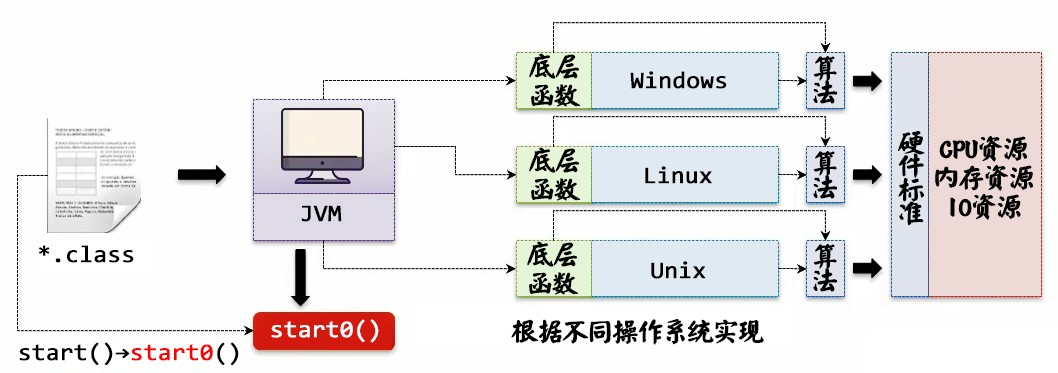
start()方法源代码：

|  |
| --- |
| **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(); |

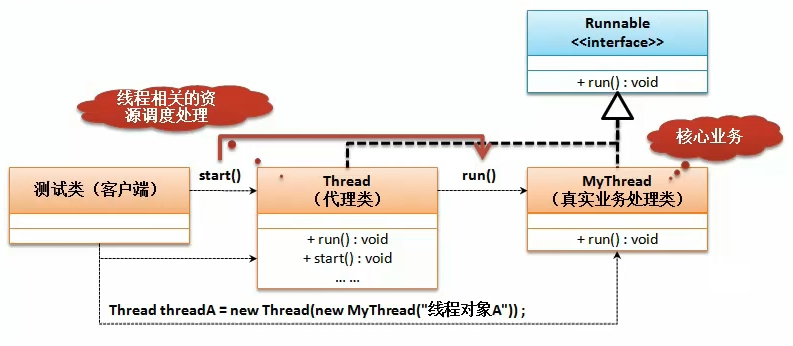
JDK 1.8开始Runnable使用了函数式接口定义，可以使用Lambda表达式定义多线程类。

|  |
| --- |
| **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

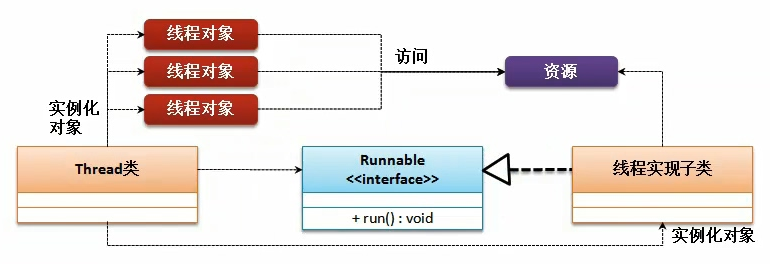
Thread类是Runnable接口的子类



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  }  // ...  } |

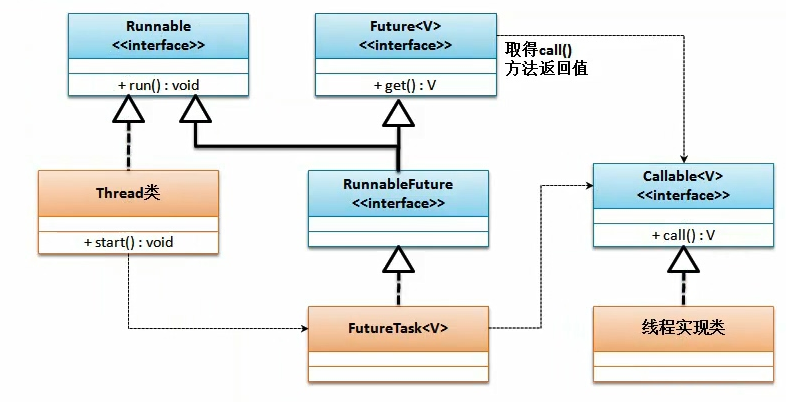
RunnableFuture接口继承于Runnable接口和Future接口。

|  |
| --- |
| **public** **interface** RunnableFuture<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

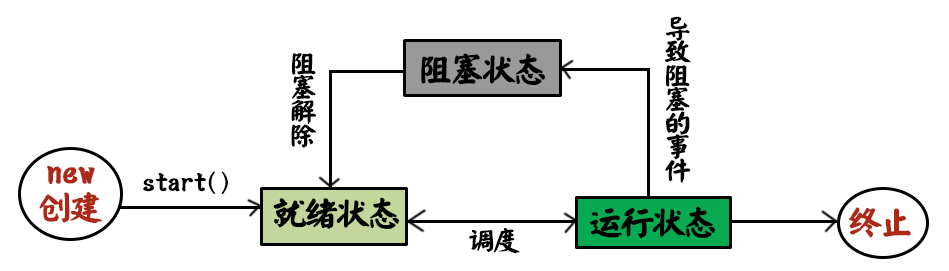
|  |
| --- |
| **class** Race **implements** Callable<Integer> {  **private** String name;  **private** **long** time; // 多少毫秒走一步  **private** **int** step; // 步数  **private** **boolean** flag = **true**; // 设为false结束线程  **public** Race(String name, **long** time) {  **this**.name = name;  **this**.time = 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。