# day18

## 多线程

### 多线程的第一种实现方式

1. 继承Thread类 : Thread类表示一个线程, 就是一个线程类



1. 多线程实现步骤:
2. 自定义出一个类, 成为Thread类的子类, 于是自定义类属于线程类
3. 在自定义线程类中, 重写从父类Thread中继承到的run方法功能, 将需要独立运行的代码逻辑写到run方法中
4. 创建出一个自定义的线程类对象, 表示准备有一个独立代码执行通道
5. 使用线程类对象, 调用start方法, 开启一个线程 :



代码

|  |
| --- |
| package com.ujiuye.thread;  // 1. 自定义出一个类, 称为Thread类的子类, 于是自定义类属于线程类  public class MyThread extends Thread {  // 2) 在自定义线程类中, 重写从父类Thread中继承到来的run方法功能,  // 将需要独立运行的代码逻辑写到run方法中  @Override  public void run() {  for(int i = 1; i <= 10; i++) {  System.out.println("run----" + i);  }  }  } |

|  |
| --- |
| package com.ujiuye.thread;  public class TestThread {  public static void main(String[] args) {  // 1. main方法本身就是一个线程  // 2. 自定出一个额外线程通道, 执行独立代码  // 3)创建出一个自定义的线程类对象, 表示准备有一个独立代码执行通道  MyThread my = new MyThread();  my.start();    for(int i = 1; i <= 10; i++) {  System.out.println("main---" + i);  }  }  } |

### 多线程第二种实现方式

1. 实现Runnable接口 : Runnable接口是就是一个线程接口, 实现类实现Runnable接口, 那么这个实现类就是一个线程类
2. 实现步骤:
3. 自定义出一个类, 实现Runnable接口
4. 重写Runnable中的run方法功能, 将需要独立运行的代码写在run中
5. 创建出一个自定义线程类对象
6. 因为Runnable接口中没有开启线程的方法, 需要依靠Thread类中的开启线程方式:

Thread(Runnable target) : 将需要运行的线程类对象封装在一个Thread类型中, 以后线程运行的就是参数target中的线程内容

1. 调用Thread类中的start方法开启线程, 线程运行的就是构造参数中target线程对象中的run方法功能

代码

|  |
| --- |
| package com.ujiuye.thread;  // 1. 自定义出一个类, 实现Runnable接口  public class MyThread implements Runnable {  // 2. 重写Runnable中的run方法功能, 将需要独立运行的代码写在run中  @Override  public void run() {  for(int i = 1; i <= 10; i++) {  System.out.println("runnable---" + i);  }  }  } |

|  |
| --- |
| package com.ujiuye.thread;  public class TestThread {  public static void main(String[] args) {  // main方法本身就是一个线程    // 再额外创建出另外一个线程  //3)创建出一个自定义线程类对象  MyThread my = new MyThread();    // 4) Thread(Runnable target) : 将需要运行的线程类对象封装在一个Thread类型中, 以后线程运行的就是参数target中的线程内容  Thread th = new Thread(my);    // 5) 调用th中start方法运行my线程中内容  th.start();    for(int i= 1; i <= 10; i++) {  System.out.println("main---" + i);  }  }  } |

### Thread类中的常用方法功能

#### 获取线程名称

1. getName() : 获取到当前线程的名称, 返回String类型线程名;

如果线程没有手动命名,默认会有名字 : Thread-0, Thread-1.....

#### 设置线程名称

1. setName(String name) : 修改线程名称为参数name
2. Thread类构造方法可以在封装一个线程对象同时, 给线程设置名称

[Thread](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "Thread(java.lang.String))([String](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) name)

[Thread](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "Thread(java.lang.Runnable, java.lang.String))([Runnable](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Runnable.html" \o "java.lang 中的接口) target, [String](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) name)

代码

|  |
| --- |
| package com.ujiuye.thread;  public class Demo01\_ThreadNameMethod {  public static void main(String[] args) {  // 匿名内部类形式, 创建出线程  Thread t0 = new Thread() {  @Override  public void run() {  for(int i = 1; i <= 50; i++) {  // 1. getName() : 获取到当前线程的名称  System.out.println(getName() + "---" + i);  }  }  };    t0.start();  // 2.setName(String name) : 修改线程名称为参数name  t0.setName("线程1");    // 3. 通过构造给线程设置名称  Thread t1 = new Thread("小强") {  @Override  public void run() {  for(int i = 1; i <= 50; i++) {  System.out.println(getName() + "---" + i);  }  }  };  t1.start();  // t1.setName("线程2");    Runnable able = new Runnable() {  @Override  public void run() {  for(int i = 1; i <= 10; i++) {  System.out.println("runnable---" + i);  }  }  };    // 4. 通过构造给线程设置名称  Thread t = new Thread(able,"小黑");  t.start();  //t.setName("线程3");  System.out.println(t.getName());  }  } |

#### 获取当前线程对象

1. Runnable接口实现类中没有任何操作线程的相关方法, 但如果就是想要在Runnable实现类中使用线程相关功能, 于是currentThread()方法, 获取到当前正在运行的线程对象可以做到让Runnable实现类中也能使用Thread类中的功能
2. 如果某段代码正在运行, 证明一定有一个线程通道执行代码, 就可以在代码中获取出正在执行代码的线程对象
3. static currentThread() : 获取到当前正在运行的线程对象, 返回Thread线程对象

代码

|  |
| --- |
| package com.ujiuye.thread;  public class Demo01\_ThreadNameMethod {  public static void main(String[] args) {  Runnable able = new Runnable() {  @Override  public void run() {  for(int i = 1; i <= 10; i++) {  // 5.static currentThread() : 获取到当前正在运行的线程对象, 返回Thread线程对象  System.out.println(Thread.currentThread().getName() + "---" + i);  }  }  };    // 4. 通过构造给线程设置名称  Thread t = new Thread(able,"小黑");  t.start();  }  } |

#### 线程休眠

1. static sleep(long time) : 表示让当前正在执行的线程休眠参数time毫秒值时间(线程休眠期间, 暂时停止执行, 失去了竞争CPU资源的资格), 待线程休眠结束, 再继续进行资源争夺
2. sleep方法调用时 : 可能会发生 打断异常InterruptedException
3. 方法重写 : 子类重写从父类(父接口)中继承来的方法功能

a: 子类重写方法 : 返回值类型, 方法名, 参数列表与父类方法一致

b : 重写后方法的权限修饰符 使用范围大于等于父类权限修饰

c : 子类重写方法, 在异常上小于等于父类

1. 如果父类方法声明上, 有异常, 子类异常只能小于等于父类的异常
2. 如果父类方法声明上, 没有异常, 那么子类重写方法在声明上就不能出现任何异常

1. 实际开发中sleep休眠使用场景 :

读取文件 : 程序员所开发的系统每天都需要对账(将今天发生的所有的资金交易的流水进行比对, 比对实际划账与记录的账务是否有问题); 于是每天晚上 24:00需要进行对账, 第三方划账系统会在24:00之前将今天的对账数据以txt文本的形式准备好

读取文件之前 : 需要做必要判断

// 最多反复读取3次文件

for(int i = 1; i <= 3 ; i++){

if(f.exists() && f.length() > 0){

// 才进行文件读取...

break;

}else{

// 休眠一点时间,等待文件上传

Thread.sleep(1000);

}

}

代码

|  |
| --- |
| package com.ujiuye.thread;  public class Demo02\_ThreadOtherMethod {  public static void main(String[] args) {  /\*for(int i = 1; i <= 5; i++) {  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + "---" + i);  }\*/    // 匿名内部类形式, 创建出线程  Thread t0 = new Thread() {  @Override  public void run() {  for(int i = 1; i <= 50; i++) {  try {  Thread.sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(getName() + "---" + i);  }  }  };    t0.start();  }  } |

#### 守护线程

1. 守护线程对比理解:

玩象棋 :

a : 炮, 马, 兵, 车..., 目的就是为了守护 将/帅 而存在, 炮, 马, 兵, 车称为守护线程

b : 将/帅 , 理解为非守护线程, 保证自己正常运行就可以了, 不需要保护其他线程

如果将/帅阵亡, 那么剩下的炮, 马, 兵, 车都没有意义了, 也跟着结束

1. setDaemon(boolean boo) : 将参数设置为true, 那么这个线程就是一个守护线程; 守护线程的执行时机 : 如果所有非守护线程都运行完毕, 那么守护线程就直接结束

守护线程通常称为用户线程或者后台线程, 很多守护线程的存在都是为了维护一个稳定的运行环境

代码

|  |
| --- |
| package com.ujiuye.thread;  public class Demo03\_ThreadDaemon {  public static void main(String[] args) {  // 匿名内部类形式, 创建出线程  Thread t0 = new Thread("守护") {  @Override  public void run() {  for(int i = 1; true ; i++) {  System.out.println(getName() + "---" + i);  }  }  };  t0.setDaemon(true);  t0.start();      // 匿名内部类形式, 创建出线程  Thread t1 = new Thread("非守护") {  @Override  public void run() {  for(int i = 1; i <= 5; i++) {  System.out.println(getName() + "---" + i);  }  }  };    t1.start();  }  } |

#### 线程优先级

1. [setPriority](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "setPriority(int))(int newPriority) : 用于设置线程的优先级; 参数数值越高证明线程的优先级别就越高;

线程的优先级直接使用Thread类中三个静态常量进行表示



1. 高优先级的线程, 会在前期执行相对比较多; 但是多线程程序, 都是并发运行, 竞争CPU资源, 因此即使设定了线程优先级别, 线程的运行仍然具有很大随机性

代码

|  |
| --- |
| package com.ujiuye.thread;  public class Demo04\_ThreadPriority {  public static void main(String[] args) {  Thread t0 = new Thread("低优先级") {  @Override  public void run() {  for(int i = 1; i <= 10; i++) {  System.out.println(getName() + "---" + i);  }  }  };  // 设置t0线程为最小优先级别  t0.setPriority(Thread.MIN\_PRIORITY);  t0.start();    Thread t1 = new Thread("普通优先级") {  @Override  public void run() {  for(int i = 1; i <= 10; i++) {  System.out.println(getName() + "---" + i);  }  }  };  // 设置t1线程为普通优先级别  t1.setPriority(Thread.NORM\_PRIORITY);  t1.start();    Thread t2 = new Thread("最高优先级") {  @Override  public void run() {  for(int i = 1; i <= 10; i++) {  System.out.println(getName() + "---" + i);  }  }  };  // 设置t2线程为最高优先级别  t2.setPriority(Thread.MAX\_PRIORITY);  t2.start();  }  } |

### 线程安全问题的发生

1. 需求 : 模拟影院销售电影票过程; 某影院要上映一场电影<葫芦娃救爷爷>, 目前共有影票100张, 可以通过美团, 猫眼, 影院三个渠道进行售票, 三个渠道共同销售这100张票, 直到票全部销售结束为止, 输出每一张票都是在什么渠道销售的; 请使用线程模拟售票过程

例如 : 影院销售第 100张票

美团销售第99张票

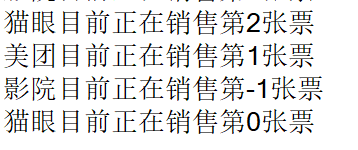
...

猫眼销售第1张票

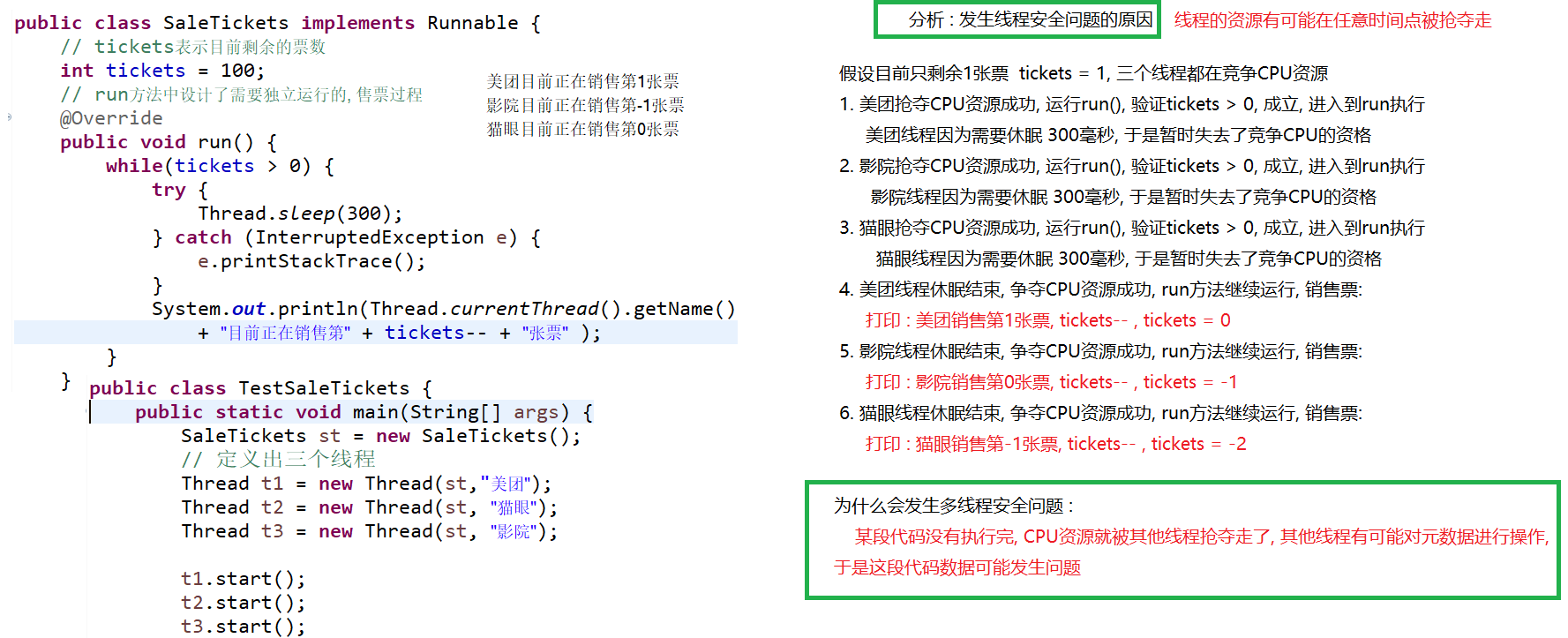
1. 分析 :

a : 美团可以单独售票, 猫眼可以单独售票, 影院可以单独售票, 于是想要把美团, 猫眼, 影院设置成三个线程, 三个线程中运行的是相同售票过程

1. 代码实现: 发现多线程同时进行售票, 售票结果与预期不符,发生线程安全问题



1. 发生线程安全问题的原因 :



### 同步代码块解决线程安全问题

保证代码可以完整执行, 执行完毕, 其他线程再争夺资源; 因此可以同步代码代码块语法结构保证代码可以完整执行

synchronized(锁对象){

// 需要保证完整执行代码

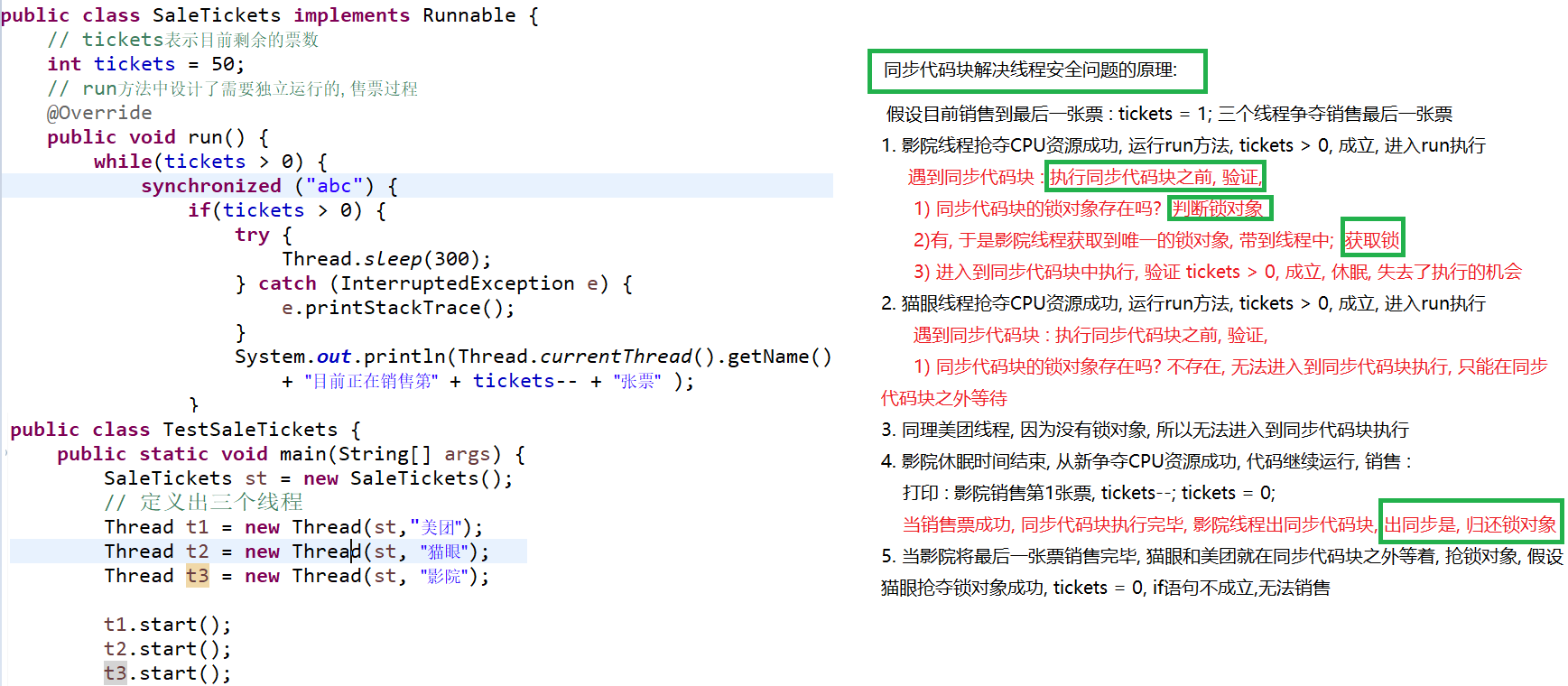
}

解释 :

1. synchronized关键字表示同步, syn alt + / 自动补全
2. 小括号中的锁对象 : 可以是任意引用数据类型对象, 但是要保证对象是唯一的

同步代码块运行机制 :

1. 进入同步代码块之前, 判断是否有锁对象; 如果有锁的对象, 获取锁对象, 进入同步代码块执行; 如果没有锁对象, 只能在同步代码块之外等待
2. 当执行同步代码块结束, 归还锁对象



代码

|  |
| --- |
| package com.ujiuye.thread;  // SaleTickets类表示售票过程线程类  public class SaleTickets implements Runnable {  // tickets表示目前剩余的票数  int tickets = 50;  // run方法中设计了需要独立运行的, 售票过程  @Override  public void run() {  while(tickets > 0) {  synchronized ("abc") {  if(tickets > 0) {  try {  Thread.sleep(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName()  + "目前正在销售第" + tickets-- + "张票" );  }  }  }  }  } |

|  |
| --- |
| package com.ujiuye.thread;  public class TestSaleTickets {  public static void main(String[] args) {  SaleTickets st = new SaleTickets();  // 定义出三个线程  Thread t1 = new Thread(st,"美团");  Thread t2 = new Thread(st, "猫眼");  Thread t3 = new Thread(st, "影院");    t1.start();  t2.start();  t3.start();  }  } |

### 同步方法解决线程安全问题

同步方法与同步代码块一样, 都可以解决线程安全问题, 只不过写法上略有不同

1. 同步方法语法结构 :

修饰符 synchronized 返回值类型 方法名(参数列表){

// 需要保证完整执行代码逻辑

}

1. 同步方法也有锁对象, 只是隐藏起来, 没有看到

a : 如果非静态方法, 同步锁对象就是 this 关键字

b : 如果静态方法, 同步锁对象就是 当前类.class字节码文件对象, 保证唯一

代码

|  |
| --- |
| package com.ujiuye.thread;  // SaleTickets类表示售票过程线程类  public class SaleTicketsMethodSave implements Runnable {  // tickets表示目前剩余的票数  static int tickets = 10;  // run方法中设计了需要独立运行的, 售票过程  @Override  public void run() {  while(tickets > 0) {  sales2();  }  }    // 静态同步方法  public static synchronized void sales2() {  // synchronized (SaleTicketsMethodSave.class) {  if(tickets > 0) {  try {  Thread.sleep(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName()  + "目前正在销售第" + tickets-- + "张票" );  }  //}  }    // 非静态同步方法  public synchronized void sales() {  //synchronized (this) {  if(tickets > 0) {  try {  Thread.sleep(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName()  + "目前正在销售第" + tickets-- + "张票" );  }  //}  }  } |

### StringBuffer和StringBuilder之间比较

StringBuffer线程安全, 运行效率相对比较低 : StringBuffer中的每一个方法都是线程安全的方法, 都添加了同步代码块保证数据安全性, 因此每一个方法都会多做几个动作 :

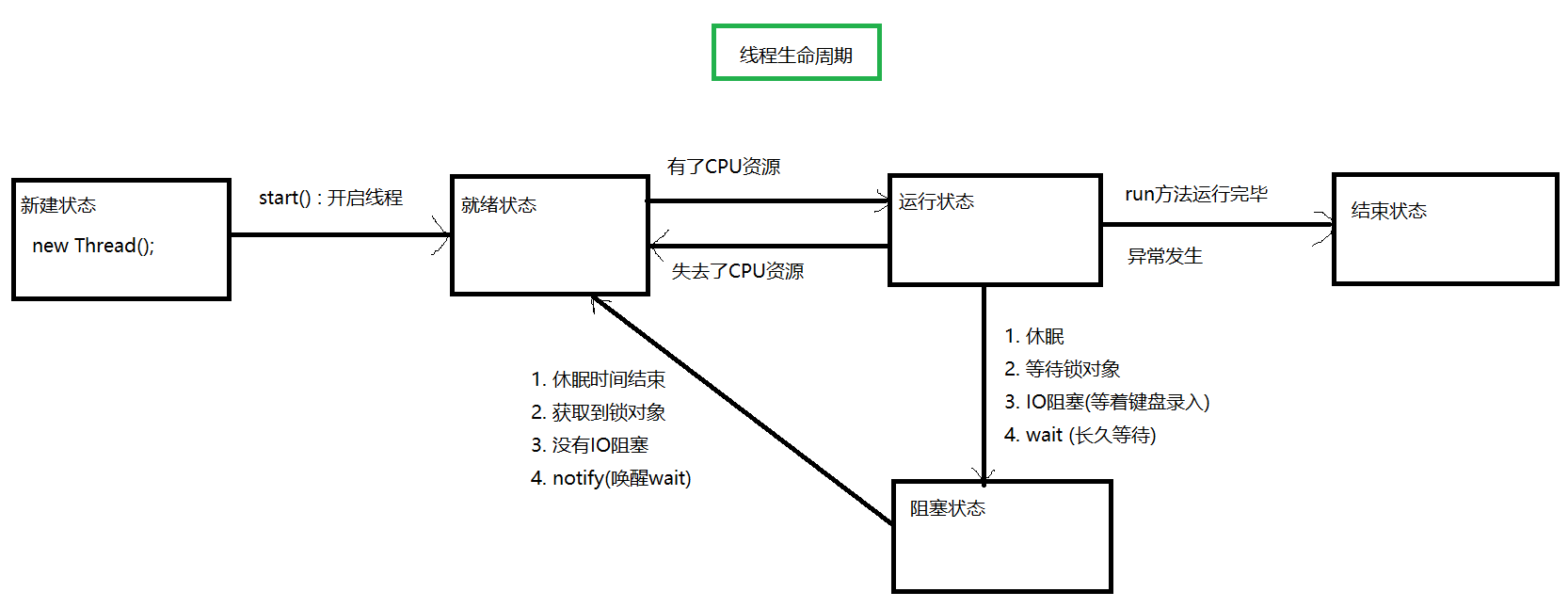
1. 判断锁
2. 获取锁
3. 归还锁

因此执行效率低

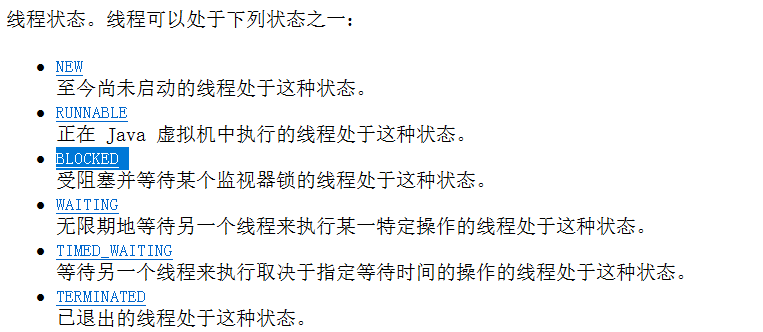
StringBuilder线程不安全, 运行效率相对比较高

### 线程状态

1. 从一个线程开启, 直到这个线程结束, 过程中一个线程最多会经历5种状态, 5种状态彼此连接组成了一个线程的生命周期
2. 线程状态:
3. 新建状态 : 创建出一个线程类对象
4. 就绪状态 : 使用start()方法开启一个线程, 只需要CPU资源
5. 运行状态 : 线程具有CPU资源, 正在运行
6. 阻塞状态 : 被动等待, 例如 : 休眠, 等待锁对象, IO阻塞(键盘录入:等待通过键盘录入数据)
7. 结束状态 : 线程中代码执行完毕, run方法执行完毕; 发生异常



1. JDK中Thread类有内部的枚举类型, State表示线程的状态



1. 枚举类型(扩展)

java中数据类型主要就是2大类:

a : 基本数据类型, 4类8种

b : 引用数据类型,主要有类, 接口, 数组, 枚举

1. 枚举类型的定义 : 使用关键字enum (枚举类型与类定义和组成非常相似)

权限修饰符 enum 枚举类型名{

}

1. 枚举使用场景 : 如果一个类型可以定义的对象的个数是固定的, 那么可以使用枚举类型表示, 举例 : 星期类型, 星期一---星期日 7个; 月份类型 : 1月----12月, 12个

代码

|  |
| --- |
| package com.ujiuye.thread;  public class WeekdayClass {  // 思考: 只能创建出7个WeekdayClass对象  // 1) 不能让其他类可以随意new对象, 构造方法私有化  private WeekdayClass() {}    // 2)在本类中创建出7个对象  public static final WeekdayClass MON = new WeekdayClass();  public static final WeekdayClass TES = new WeekdayClass();  public static final WeekdayClass WEN = new WeekdayClass();  } |

枚举类型代码:

|  |
| --- |
| package com.ujiuye.thread;  public enum Weekday {  // 只主要将需要创建出的固定个数的对象的名字罗列出来即可  // 每一个对象名称称为 : 枚举项, 多个枚举项之间使用逗号进行分隔, 最后一个枚举项使用分号作为结尾  // 枚举项必须写在第一行  // 枚举类型中的构造方法默认私有化  /\*  \* public static final WeekdayClass MON = new WeekdayClass();  public static final WeekdayClass TES = new WeekdayClass();  public static final WeekdayClass WEN = new WeekdayClass();  \*/  MON,TES,WEN;  } |

|  |
| --- |
| package com.ujiuye.thread;  public class TestEnum {  public static void main(String[] args) {  WeekdayClass mon = WeekdayClass.MON;  WeekdayClass mon1 = WeekdayClass.MON;  System.out.println(mon == mon1);    // 测试枚举类型  Weekday monEnum = Weekday.MON;  Weekday monEnum1 = Weekday.MON;  System.out.println(monEnum == monEnum1);// true  }  } |

### 线程池

1. 没有线程池 : 如果代码中有很多线程任务需要执行, 每一个线程任务中代码逻辑不是很多, 每次线程任务执行都需要新建一个线程通道, 需要开辟空间; 当线程任务执行完毕, 线程占有内存空间需要回收, 对于线程任务很多的场景, 频繁去开辟线程空间, 回收线程空间, 很浪费时间和空间问题, 引入线程池解决
2. 有线程池 : 创建出一个线程池, 在线程池中, 提前准备好几个线程, 如果有线程任务需要执行, 从线程池中获取出一个线程执行任务; 当任务执行完毕, 线程重新回到线程池中, 继续反复使用, 再接着执行其他任务; 不需要频繁的开辟和销毁新城池空间
3. 线程池使用步骤:
4. Executors : 类, 来自于java.uitl.concurrent包中, 表示线程池的工厂类, 用于创建出具有指定线程个数线程池出来

static newFixedThreadPool(int count) : 表示创建出一个具有count个线程的线程池

方法的返回值类型ExecutorService接口,证明方法实际返回是接口的实现类对象

1. ExecutorService : 接口, 表示操作线程池的类型

submit(Runnable target) : 将需要执行的线程任务提交到线程池中, 运行, 如果有空闲线程直接执行, 如果没有空闲线程, 等待

当线程池使用完毕, 需要关闭线程池, 否则代码不停, 线程池持续等待为你进行线程任务服务

[shutdown](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/../../../java/util/concurrent/ExecutorService.html" \l "shutdown())(): 关闭线程池, 所有已经提交任务,保证完成

[shutdownNow](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/../../../java/util/concurrent/ExecutorService.html" \l "shutdownNow())() : 关闭线程池,立马关闭销毁, 目前正在执行线程任务保证完成, 等待任务不在不执行

代码

|  |
| --- |
| package com.ujiuye.thread;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  public class Demo01\_ThreadPool {  public static void main(String[] args) {  Runnable run1 = new Runnable() {  @Override  public void run() {  for(int i = 1; i <= 5; i++) {  System.out.println(Thread.currentThread().getName() +"--"+i);  }  }  };    Runnable run2 = new Runnable() {  @Override  public void run() {  for(int i = 1; i <= 5; i++) {  System.out.println(Thread.currentThread().getName() +"--"+i);  }  }  };    Runnable run3 = new Runnable() {  @Override  public void run() {  for(int i = 1; i <= 5; i++) {  System.out.println(Thread.currentThread().getName() +"--"+i);  }  }  };    Runnable run4 = new Runnable() {  @Override  public void run() {  for(int i = 1; i <= 5; i++) {  System.out.println(Thread.currentThread().getName() +"--"+i);  }  }  };    // 1. 创建出一个线程池, 具有两个线程任务  ExecutorService es = Executors.newFixedThreadPool(2);  // 2. submit方法提交线程任务  es.submit(run1);  es.submit(run2);  es.submit(run3);  es.submit(run4);    // 3. 结束线程池  // es.shutdown();  es.shutdownNow();  }  } |