## 一 Java 局部内部类使用局部final变量

1. 内部类是外部类的一个成员，就像外部类的成员方法一样，所以内部类有权限访问外部类的所有成员，包括private的。
2. 内部类不能访问外部类方法中的局部变量，除非变量是final的(一般发生在方法中定义的内部类)。这是因为局部变量的生命周期原因。
3. 所谓“局部内部类”就是在对象的方法成员内部定义的类。而方法中的类，访问同一个方法中的局部变量，却必须要加上一个final。

原因是编译程序实现上的困难：

* 内部类对象的生命周期会超过局部变量的生命期。局部变量的生命期：当该方法被调用时，该方法中的局部变量在栈中被创建，当方法调用结束时，退栈，这些局部变量全部死亡。而内部类对象生命期，与其它类一样，当创建一个局部内部类对象后，只有当没有其它人再引用它时，它才能死亡。所以完全可能一个方法已调用结束（局部变量已死亡），但该局部类的对象仍然活着。即：局部类的对象生命期会超过局部变量。
* 局部内部类的对象访问同一个方法中的局部变量，那么这就要求只要局部内部类对象还活着，那么栈中的那些它要访问的局部变量就不能“死亡”（否则：它都死了，还访问个什么呢？）。这就是说：**局部变量的生命期至少等于或大于局部内部类对象的生命期。**
* 解决方法：局部内部类的对象可以访问同一个方法中被定义为final的局部变量。定义为final后，编译程序的实现方法：将所有的局部内部类对象要访问的final型局部变量，都**拷贝**成为该内部类对象中的一个数据成员。这样，即使栈中局部变量（含final）已死亡，但由于它是final,其值永不变，因而局部内部类对象在变量死亡后，照样可以访问final型局部变量
* 归纳总结：局部内部类对象中包含有要访问的final型局部变量的一个**拷贝**，成为它的数据成员。因此，正是在这个意义上，final型局部变量的生命期，超过其方法的一次调用。**严格来说，方法调用结束，所有的局部变量（含final）全死亡了。但：局部内部类对象中有final型局部变量的拷贝。**

## 二 java 24 \* 60 \* 60 \* 1000（一天）

Date d=new Date();

SimpleDateFormat df=new SimpleDateFormat("yyyy-MM-dd");

System.out.println("today:"+df.format(d));

System.out.println("the day **befor yestoday**:" + df.format(new Date(**d.getTime() - (long)2 \* 24 \* 60 \* 60 \* 1000)**));

System.out.println("the day **after tomorrow**:" + df.format(new Date(**d.getTime() + (long)3 \* 24 \* 60 \* 60 \* 1000)**));

## 三 Java中子类和父类的初始化顺序-静态优先，父类优先

类的初始化顺序是：

1、初始化父类中的静态成员变量和静态代码块。

2、初始化子类中的静态成员变量和静态代码块。

3、初始化父类中的普通成员变量和代码块，在执行父类中的构造方法。

4、初始化子类中的普通成员变量和代码块，在执行子类中的构造方法。

## 四 Java transient关键字

java语言的关键字，变量修饰符，如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。换句话来说就是，用transient关键字标记的成员变量不参与序列化过程。

## 五 Java >> n & << n

>> n （右移 除2的n次方）

<< n （左移 乘2的n次方）

## 六 Java Collection接口的toArry方法注意

1. public Object[] toArray()方法：方法中会重新构造一个Object[]数组，然后将集合中每个元素赋给数组，由于new的是Object数组，所以无法向上转型回之前的数组。
2. public <T> T[] toArray(T[] a)方法：返回传入的数组a指定的类型，a的length小于集合size时会利用反射重新构建一个指定类型数组，并把集合中所有元素赋给数组。

## 七 编程习惯

### 异常处理

<http://www.importnew.com/26858.html>

不要随便加null判断（并不“健壮”，反而掩盖错误）：如果写了，就必须测试为空和不为空的两种情况

不捕获异常，将异常交给调用者处理：比起从日志中寻找错误，应在编码阶段处理，尽早让错误抛出。

### 接口定义

<https://zhuanlan.zhihu.com/p/28708259>

### Controller规范

## 八 Java 8

<http://www.importnew.com/16436.html>

<http://www.importnew.com/11908.html>

1. Lambda表达式（闭包）；
2. 函数式编程支持： java.util.function.\* 下的众多接口；
3. 流API；
4. 接口的默认方法和静态方法；
5. 新的Date和Time API。

### 1 Optional 接口

<http://www.importnew.com/6675.html>

对对象的null值检测进行包装，如果值存在则isPresent返回true，否则false。

Optional.of：创建一个Optional，不能为null，赋值抛出NullPointerException。

Optional.ofNullable：可以创建一个null的Optional。

isPresent：值存在返回true，否则false。

ifPresent：值存在就调用Consumer对象的accept方法，否则无任何动作。

get：返回值，空Optional抛NoSuchElementException异常。

orElse：值存在返回，否则返回指定值。

orElseGet：值存在返回，否则调用Supplier的get返回。

orElseThrow：存在返回，否则抛出指定异常。

map、flatMap、filter

### 2 Lambda

Lambda表达式允许将函数作为一个方法参数传递，或许将代码看成数据。

用 () -> {} 代替整个匿名类。

new Thread( **() -> {**int i = 9 ; System.out.println(“somting”+i) ; }).start();

() 中参数可省略类型，没有参数时 () 仍然要保留。

{} 只有一条语句时 {} 可以省略，语句尾的 ; 也可以省略。

Arrays.asList(“a”, “b”, “c”, “d”).forEach(va -> System.out.println(va));

**FunctionalInterface：函数式接口，如果一个接口只有一个方法，那么编译器会将该接口视为函数式接口，如果在接口上加注解FunctionalInterface注解，那么接口就会被强制要求符合函数式接口的规范（只有一个抽象方法，默认，静态方法不包括在内）。**

### 3 Predicate接口

适用于过滤。

该接口有一个test方法，返回boolean，表示检查结果。提供了类似于逻辑操作符AND和OR的方法and()、or()、xor()。可以将多个条件进行组合。

Predicate<String> testStart = (str) -> str.startsWith("a");  
Predicate<String> testContain = (str) -> str.contains("b");  
boolean result = testStart.and(testContain).test("acvvb"); // true

### 4 Map的Reduce

map方法（Function接口）遍历并将每一个数乘以2，reduce（BiFunction接口）方法将结果汇总于sum。

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
Optional<Integer> reduce = list.stream().map(va -> va >> 1).reduce((sum, cur) -> sum += cur);  
*o*.accept(reduce.orElse(-1));

### 5 Function<T,R>接口

用于产生对象，函数式接口，接受一个参数，构造并返回目标对象。

R apply(T t);

### 6 BiFunction<T,R,U>接口

用于产生对象，接受两个参数，生成一个结果。

R apply(T t,U u);

### 7过滤集合元素

Filter方法（Predicate接口）遍历元素并检查，返回true保留，collect方法（Collector接口）将结果组合为新的集合。

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
List<Integer> collect = list.stream().filter(va -> va % 2 == 0).collect(Collectors.*toList*());  
*o*.accept(collect);

### 8 Supplier接口

用来产生对象

T get();

### 9 计算集合元素的最大，最小，总和以及平均值

IntStream、LongStream和DoubleStream中有个summaryStatics方法，可以返回IntSummaryStatics、LongSummaryStatics或DoubleSummaryStatics，描述流中的各种摘要数据。

Stream的mapToInt/Long/Double方法可将指定对象作为参数，产生对应的基本类型，mapXXX中接受的接口为：ToInt/Long/DoubleFunction，方法为(以int为例)：int applyToInt(T value);

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
IntSummaryStatistics statistics = list.stream().mapToInt(i -> i).summaryStatistics();  
statistics.getMax();  
statistics.getMin();  
statistics.getAverage();  
statistics.getSum();  
statistics.getCount();

### 10 Lambda VS 匿名内部类

this关键字：匿名类的this指向匿名类，而Lambda的this指向外部类。

编译方式：Lambda被编译为外部类的私有成员方法，而匿名类被编译为单独的类。

同：两种方式在引用外部的局部变量时局部变量要被final修饰。

### 11 方法引用

List<Car> cars = new ArrayList<>();

1. 方法构造器（无参可访问的构造器）引用：Class::new ，可以便捷的创建一个对象，通常配合Supplier接口使用。
2. 引用静态方法（一参可访问）：Class::static\_method，通常配合Consumer接口使用。
3. 引用特定类的方法（无参可访问）：Class::method
4. 引用特定对象的方法：instance::method

### 12 新的Date/Time API

在java.time.\*包下：

新的API涵盖了所有处理日期、时间、时区、过程与时钟的操作。

1. Clock类：

UTC（Universal Time Coordinated）为世界协调时间，与格林尼治时间一样，与英国伦敦时间也一样。北京是东八区，领先于UTC8个小时。

Clock clock = Clock.*systemUTC*();  
 Instant instant = clock.instant();  
 Instant plus = instant.plus(8, ChronoUnit.*HOURS*);// 加 8 小时  
  
 Instant minus = instant  
// .plus(24 \* 60 \* 60 \* 1000, ChronoUnit.MILLIS) // 加一天  
 .minus(24 \* 60 \* 60 \* 1000, ChronoUnit.*MILLIS*);// 减一天  
 *o*.accept(instant); // UTC 时间：2017-10-24T07:11:12.576Z  
 *o*.accept(plus);// 北京时间：2017-10-24T15:11:12.576Z  
 *o*.accept(minus);// UTC 时间减一天：2017-10-23T07:11:12.576Z

1. LocalDate、LocalTime和LocalDateTime

三者只持有ISO-8601格式具有时区信息的日期与时间。前两者都可以从Clock中获得。

Clock clock = Clock.*systemUTC*();  
LocalDate date = LocalDate.*now*();  
LocalTime time = LocalTime.*now*(clock);  
LocalDateTime dateTime = LocalDateTime.*now*();  
*o*.accept(date); //2017-10-24  
*o*.accept(time); // 07:22:07.781 系统时间：15:22:37.050  
*o*.accept(time.getNano()); // 319000000  
*o*.accept(dateTime); // 2017-10-24T15:23:54.109

1. Duration，在秒与纳秒级别上的一段时间。简化了计算两个时间差的不同。

Duration duration = Duration.*between*(  
 LocalDateTime.*of*(2014, Month.*JULY*, 13, 21, 0, 0), // 2014-07-13 21:00:00  
 LocalDateTime.*of*(2014, Month.*JULY*, 14, 22, 13, 16)); // 2014-07-14 22:13:16  
*o*.accept(duration.toDays()); // 1  
*o*.accept(duration.toHours()); // 25  
*o*.accept(duration.toMinutes()); // 1513  
*o*.accept(duration.toNanos()); // 90796000000000

## 九 java值传递

Java使用对象的引用进行计算，但是java在向方法传递对象时传递的不是引用，是值。

常见的swap函数：

public void swap(int a,int b) { // 错误的交换方法

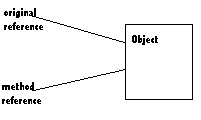
int temp = a;

a = b;

b = temp;

}

但swap方法返回时，a，b并没有交换。Java通过传值来传递引用：这意味着，**传向函数的引用实际上是原始引用的副本**。进行交换操作时，交换的是副本引用，真正的引用并未交换。（我们应该交换原始的引用，而不是它们的副本）。



当函数被调用后，一个对象至少存在两个引用。

## 十 jvm类加载机制

<http://shulianghan.iteye.com/blog/1699341>

### 类加载器深入剖析，Java虚拟机与程序的生命周期

当我们执行一个java程序的时候,会启动一个JVM进程,当程序执行完之后,JVM进程就消亡了;

在如下情况下JVM将结束生命周期:

1. **System.exit(int)方法:**

当执行这个方法的时候,虚拟机会退出;这个方法传入一个整形参数,这个参数是状态码，如果这个整形是*0的话,就是正常退出,如果不是0的话,就是异常退出;*

1. **程序执行过程中,遇到了异常或错误,而异常终止:**

如果我们的程序中出现了异常,而不去处理,会将异常一直抛给main函数,main函数会将异常抛给JVM,JVM如果处理不了异常,JVM就会异常退出;

1. **由于操作系统出现错误导致JVM进程终止:**

JVM所依赖的平台出现错误,导致JVM终止;

### 类的加载,连接和初始化加载

1. 加载

JVM查找并加载类的二进制数据,将class字节码文件加载到内存中;

1. **连接： 验证 – 准备 – 解析**

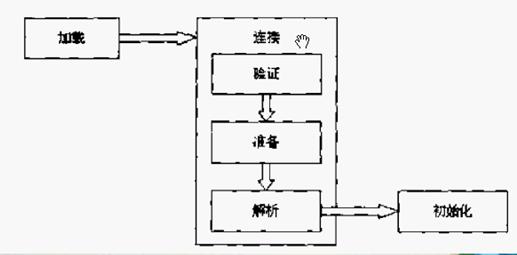
**验证：**确保被加载的类的正确性,使用javac编译工具生成的字节码文件能通过验证,如果不是由javac编译生成的字节码文件,或者生成的字节码文件不符合JVM虚拟机对字节码文件的要求,就会出现验证通不过的情况;比如说随便拿一个文件,将后缀名直接修改为.class,这样的字节码文件肯定不合法;

**准备:**为类的静态变量分配内存,并将其初始化为默认值;

**解析:**把类中的符号引用转为直接引用;

1. **初始化**

为类的静态变量赋予正确的初始值(正确的值指的是用户赋的值);（好像这个与连接阶段的准备有些重复,在连接的准备阶段只是**赋予初始变量**,如果用户给这个变量赋了初始值,那么这个变量在连接的准备阶段仍然会赋予初始值;在这个阶段,才会真正的将初始值赋给静态变量;）



### Java程序对类的使用方式有主动使用和被动使用

所有的JVM实现,必须在每个类或者接口被java程序“首次主动使用”时才初始化他们;

1. 主动使用的6种方式:
2. 创建类的实例;
3. 访问某个类或接口的静态变量,或者对静态变量赋值;
4. 调用类的静态方法;
5. 反射:Class.forName(“类名”);
6. 初始化一个类的子类,看做是对父类的主动使用;
7. java虚拟机启动的时候,被标明启动类的类,即包含main方法的类,程序的入口;

除了上面6种主动使用之外,其它的情况均为被动使用,其它情况都不会执行第三步初始化;

### 类的加载

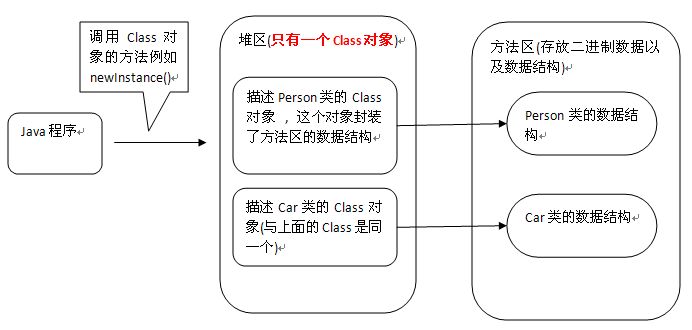
1. 概念

**类的加载：**将类的.class文件中的二进制数据**读入到内存**中,将其放在运行时**数据区的方法区**内,然后再在堆区创建一个java.lang.Class对象,用来封装类在方法区内的数据结构;

**反射：**反射就是根据堆区的字节码文件,获取方法区的数据结构;

**解析：**Class对象是由JVM自己创建的,所有的对象都是经过Class对象创建,这个Class对象是反射的入口,通过Class对象,可以关联到目标class字节码文件的内部结构;

**所有的类对应的Class对象都是唯一的**,只有一个。这个类对象是由JVM进行创建的,并且只有JVM才会创建Class对象;



类加载的最终产品是位于堆区中的Class对象,Class对象封装了类在方法区内的数据结构,并且向Java程序员提供了访问方法区内的数据结构的接口(反射用的接口);

1. 加载.class文件的方式

* 从**本地系统**中直接加载：编译好的.class字节码文件直接从硬盘中加载;
* 通过**网络下载**.class文件：将class字节码文件放在网络空间中,使用URLClassLoader来加载在网络上的.class字节码文件,使用默认的父亲委托机制加载字节码文件;
* 从**zip,jar等压缩文件**中加载字节码文件：在开发的时候,导入jar包,就是这种方式;
* 从**专有的数据库**中提取字节码文件;
* 将**java源文件动态编译**为字节码文件;

### 类加载器

1. **Java虚拟机自带的类加载器：**根类加载器(Bootstrap)；
2. 是C++写的,程序员无法在java代码中获取这个类,如果使用getClassLoader()方法获取，获取到的是一个null；（String类是由根类加载器进行加载的）
3. **用户自定义的类加载器：**自定义的类加载器都是java.lang.ClassLoader子类，用户可以定制类的加载方式。
4. 类加载器并不一定要在某个类被“首次主动使用”时再加载它

**预加载机制**：JVM规范允许类加载器在预料某个类将要被使用的时就**预先加载**它;

**报错时机**：如果在预加载的过程中遇到了字节码文件缺失或者存在错误的情况,类加载器会在程序首次**主动使用**(上面提到的六种情况)该类的时候**报错**(LinkageError错误);

**不报错时机**：如果这个错误的字节码文件所对应的**类一直没有被使用**,那么类加载器就不会报告错误,即便有错误也不会报错;

LinkageError:这个错误是Error的子类,程序不能处理这些错误,这些错误都是由虚拟机来处理,这个错误表示出错的是子类,在一定程序上依赖于另一个类,在编译了前面一个类的时候,与后面所依赖的类出现了不兼容的情况;例如:我们使用了jdk1.6在编译一个程序,但是运行环境是jre1.5的,就会出现LinkageError错误;

1. 类的链接

类被加载之后,就进入链接阶段;链接阶段会将已读入内存的二进制数据合并到虚拟机的运行时环境中去;

链接顾名思义就是讲类与类之间进行关联,例如**我们在类A中调用了类B,在链接过程中,就将A与B进行链接,**将面向对象语言转化为面向过程语言;

1. 类的验证：

类文件的结构检查

语义检查

字节码验证

二进制兼容性的验证

1. 类的解析

在解析阶段,JVM会把类的二进制数据中的符号引用替换为直接引用

1. 类的初始化

在初始化阶段,Java虚拟机执行类的初始化操作,为类的静态变量赋予初始值

# 十一 Thread的yield、sleep、join、interrupt、start

<http://blog.csdn.net/u014290221/article/details/51436710>

## join - 等待线程结束

线程A调用其它线程B的join方法，则线程A执行到该语句时将暂停，等到B线程执行完毕时调用线程才继续往下执行。

Thread th = new Thread(() -> {  
 Thread thread = Thread.*currentThread*();  
 *o*.accept(thread.getName() + " start");  
 try {  
 Thread.*sleep*(3000);  
 *o*.accept(thread.getName() + " finished");  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}, "th#1");  
  
th.start();  
*o*.accept(Thread.*currentThread*().getName() + " start join");  
th.join();  
*o*.accept(Thread.*currentThread*().getName() + " finished join");

输出：

main start join

th#1 start

th#1 finished

main finished join

## sleep wait – 线程睡眠/等待

Thread.sleep：

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，**将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态**）

Object.wait：

wait()是Object类的方法，调用对象的wait()方法导致**当前线程放弃对象的锁**（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lockpool），如果线程重新获得对象的锁就可以进入就绪状态。

## yield – 线程让步

yield方法和sleep方法有点相似，也是Thread.yield静态方法。yield方法会强制线程从**运行状态转入就绪状态**。

它也可以让当前正在执行的线程暂停，但它不会阻塞该线程，它只是将该线程转入到就绪状态。即让当前线程暂停一下，让系统的线程调度器重新调度一次，完全可能的情况是：当某个线程调用了yield()方法暂停之后，线程调度器又将其调度出来重新执行。**（当线程调用了yield方法后，只有优先级与当前线程相同或者比当前线程高的处于就绪状态的线程才会获得执行机会（抢占cpu执行权））。**

## Interrupt – 中断线程

interrupt()的作用是中断线程

本线程中断自己是被允许的；其它线程调用本线程的interrupt()方法时，会通过checkAccess()检查权限。这有可能抛出SecurityException异常。

如果本线程是处于阻塞状态*（调用线程的wait(), wait(long)或wait(long, int)会让它进入等待(阻塞)状态，或者调用线程的join(), join(long), join(long, int), sleep(long), sleep(long, int)也会让它进入阻塞状态），*此时调用了它的interrupt()方法，那么**它的“中断状态”会被清除并且线程会收到一个InterruptedException异常**。

例如，线程通过wait()进入阻塞状态，此时调用interrupt()中断该线程；调用interrupt()会将线程的中断标记置为“true”，线程会收到一个InterruptedException异常，同时“中断标记”会立即被清除为“false”。（收到InterruptedException异常后，线程就会恢复到运行状态）

中断一个“已终止的线程”不会产生任何操作。

## start – 使线程进入就绪状态

start方法用来启动线程，使其进入就绪状态，接受CPU调度，最终运行。

区别于直接调用run方法，run方法称为线程体，如果直接调用run方法，那么run方法将被当成普通方法执行，不会开启新的线程执行。

## 线程的优先级

线程的优先级用数字表示，范围为1~10，默认为5.

每个线程的默认优先级与创建它的父线程相同。数字越大，优先级越高。

Thread类有三个常量：

MAX\_PRIORITY (10)

MIN\_PRIORITY (1)

NORM\_PRIORITY (5)

Thread thread = new Thread();  
thread.setPriority(Thread.*MAX\_PRIORITY*);

## 线程的终止

Thread.stop方法已经废弃，终止线程的根本方式是**run方法返回**。

Runable的run方法并未抛出异常，所以终止线程的方法就是**run方法返回。**

1. 终止处于“阻塞状态”的线程

通常，我们通过“中断”方式终止处于“阻塞状态”的线程（即调用线程的interrupt方法）。线程由于被调用了sleep(), wait(), join()等方法而进入阻塞状态（这些方法都会抛出InterruptException）；若此时调用线程的interrupt()将线程的中断标记设为true。则线程将产生一个InterruptedException异常（阻塞方法将收到此异常），同时中断标记会被清除（置为false）。

那么只需在收到InterruptException异常时终止run方法执行即可。（run方法返回线程就会结束）。

1. 终止处于“运行状态”的线程

通常通过使用“中断标记”，或添加“额外的结束**标记**”的方式判断线程是否需要结束。

如：调用isInterrupt()判断中断标记是否为true，在需要结束的地方调用线程的interrupt方法。

Thread thread = new Thread(() -> {  
 Thread t = Thread.*currentThread*();  
 while (!t.isInterrupted()) {  
 *o*.accept("a");  
 }  
});  
  
thread.start();  
Thread.*sleep*(3000);  
thread.interrupt();

# 十二 守护（后台）线程

<http://blog.csdn.net/shimiso/article/details/8964414>

在java中有两类线程：用户线程（User Thread），守护线程（Daemon Thread）

任何一个守护线程都是整个JVM中所有非守护线程的保姆。

只要当前JVM实例中尚存在**任何一个非守护线程**没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。

Daemon的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC (垃圾回收器)，它就是一个很称职的守护者。

User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全部退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。 因为没有了被守护者，Daemon也就没有工作可做了，也就没有继续运行程序的必要了。

Thread daemonThread = new Thread();  
Thread thread = new Thread();  
  
// 设定 daemonThread 为 守护线程，default false(非守护线程)  
daemonThread.setDaemon(true);  
  
// 验证当前线程是否为守护线程，返回 true 则为守护线程  
*o*.accept(daemonThread.isDaemon()); // true  
*o*.accept(thread.isDaemon()); // false

注意点：

1. 手动创建守护线程，setDaemon(true)需在start之前设置（否则会抛IllegalThreadStateException）。
2. 在Daemon中启动的线程默认也是daemon的

JRE判断程序是否执行结束的标准是所有的前台执线程行完毕了。

补充说明：

1. **定义**：守护线程--也称“服务线程”，在没有用户线程可服务时会自动离开。
2. **优先级**：守护线程的优先级比较低，用于为系统中的其它对象和线程提供服务。
3. **设置**：通过setDaemon(true)来设置线程为“守护线程”；将一个用户线程设置为守护线程的方式是在 线程对象创建 之前 用线程对象的setDaemon方法。
4. **example**: 垃圾回收线程就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread,程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。
5. **生命周期**：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统“同生共死”。

那Java的守护线程是什么样子的呢。当JVM中所有的线程都是守护线程的时候，JVM就可以退出了；如果还有一个或以上的非守护线程则JVM不会退出。

实际应用：消息推送线程。