Java进阶1 第14天

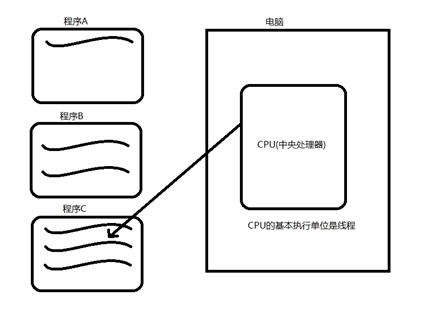
**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】多线程概述**
2. 【理解】能够理解线程与进程的概念
3. 【理解】能够理解多线程提高运行效率原因
4. **【应用】多线程实现**
5. 【理解】能够阐述Java是否支持多线程以及如何在多个线程上执行
6. 【应用】能够独立写出线程的两种创建方式
7. 【理解】能够理解用匿名内部类创建线程的代码
8. 【应用】能够独立写出多线程卖票的案例
9. **【应用】线程池与线程生命周期**
10. 【理解】 能够阐述线程四个基本状态
11. 【应用】 能够独立完成利用Runnable接口向线程池提交任务的代码
12. 【应用】 能够独立完成利用Callable接口向线程池提交任务的代码

# 多线程概述

## 线程与进程概念



A：进程

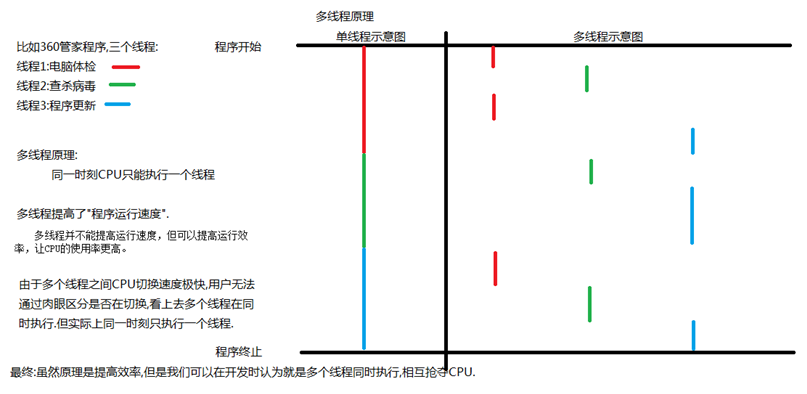
正在运行的程序。确切的来说，当一个程序进入内存运行，即是一个或多个进程在运行，具有一定独立功能。

B：线程

线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程。

## 多线程提高运行效率



多线程并不能提高运行速度，但可以提高运行效率，让CPU的使用率更高。

并且，如果多线程有安全问题的处理时，运算速度反而更低。

# 多线程实现

## Java中的多线程

A：java支持多线程。

B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。

可以在main方法执行时，开启多个线程A、B、C。

多个线程main，A，B，C同时执行，相互抢夺CPU。

C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。

#### 案例代码一:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.Date;  /\*\*  \* **@ClassName**: ThreadDemo  \* **@Description**: Java中的多线程  \* **@date** 2018年2月4日 上午10:38:55  \* Company www.igeekhome.com  \*  \* A：java支持多线程。  \*  \* B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。  \* 可以在main方法执行时，开启多个线程A、B、C。  \* 多个线程main，A，B，C同时执行，相互抢夺CPU。  \*  \* C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。  \*/  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  Date today = **new** Date();  //......  }  } |

## 创建线程

该如何创建线程呢？通过API中搜索，查到Thread类。通过阅读Thread类中的描述。Thread是程序中的执行线程。Java 虚拟机允许应用程序并发地运行多个执行线程。

我们需要用到Thread类的一些方法：

public final String getName() 获取线程名称

public final void setName(String name) 指定线程名称

public static Thread currentThread() 获取当前线程对象

### 继承Thread类创建线程

将类声明为Thread的子类。该子类重写Thread类的run方法。创建对象，开启线程。

开启线程的步骤：

1、指定线程执行目标：定义一个Thread类的子类，重写run方法，将相关逻辑实现

public void run() 线程要执行的业务逻辑方法，相当于该线程的"main方法"

2、创建自定义的线程子类对象

3、开启线程动作

public void start() 使该线程开始执行

#### 案例代码二:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: ThreadDemo  \* **@Description**: Java中的多线程  \* **@date** 2018年2月4日 上午10:42:57  \* Company www.igeekhome.com  \*  \* A：java支持多线程。  \*  \* B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。  \* 可以在main方法执行时，开启多个线程A、B、C。  \* 多个线程main，A，B，C同时执行，相互抢夺CPU。  \*  \* C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。  \*  \* 第一种方式创建线程：  \* 开启线程的步骤：  \* 1、指定线程执行目标：定义一个Thread类的子类，重写run方法，将相关逻辑实现  \* public void run() 线程要执行的业务逻辑方法，相当于该线程的"main方法"  \* 2、创建自定义的线程子类对象  \* 3、开启线程动作  \* public void start() 使该线程开始执行  \*  \* Thread类的一些方法：  \* public final String getName() 获取线程名称  \* public final void setName(String name) 指定线程名称  \* public static Thread currentThread() 获取当前线程对象  \*/  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {    //开启线程  //创建线程对象  MyThread thread = **new** MyThread();  MyThread thread2 = **new** MyThread();    //开启线程动作  thread.start();  thread2.start();    //返回main线程  Thread mainThread = Thread.*currentThread*();    **for**(**int** i=0;i<50;i++){  //System.out.println("main:"+i);  System.***out***.println(mainThread.getName()+":"+i);  }    }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: MyThread  \* **@Description**: 自定义线程类  \* **@date** 2018年2月4日 上午11:34:45  \* Company www.igeekhome.com  \*  \* 定义了一个Thread类的子类线程  \*/  **public** **class** MyThread **extends** Thread{  /\*\*  \* **@Title**: run  \* **@Description**: 重写run方法，将相关逻辑实现  \* **@see** java.lang.Thread#run()  \*/  @Override  **public** **void** run() {  **for**(**int** i=0;i<50;i++){  //System.out.println("MyThread:"+i);  System.***out***.println(**this**.getName()+":"+i);  }  }    } |

### 实现Runnable接口创建线程

创建线程的另一种方法是声明实现 Runnable 接口的类，该类然后实现 run 方法。然后创建Runnable的实现类对象，传入到某个线程的构造方法中，开启线程。

为何要实现Runnable接口，Runnable是啥玩意呢？继续API搜索。

查看Runnable接口说明文档：Runnable接口用来指定每个线程要执行的任务。包含了一个 run 的无参数抽象方法，需要由接口实现类重写该方法。

开启线程的步骤：

1、指定线程执行目标：定义Runnable线程执行目标实现类，重写run方法，指定目标逻辑

2、通过指定线程执行目标的构造方法创建线程对象

public Thread(Runnable target)

a)创建线程执行目标对象

b)通过线程执行目标创建线程对象

3、开启线程动作 (start开启线程)

#### 案例代码三:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: ThreadDemo2  \* **@Description**: Java中的多线程  \* **@date** 2018年2月4日 上午10:42:57  \* Company www.igeekhome.com  \*  \* A：java支持多线程。  \*  \* B：当java程序执行main方法的时候，就是在执行一个名字叫做main的线程。  \* 可以在main方法执行时，开启多个线程A、B、C。  \* 多个线程main，A，B，C同时执行，相互抢夺CPU。  \*  \* C：Thread类是java.lang包下的一个常用类，每一个Thread类的对象，就代表一个处于某种状态的线程。  \*  \* 第一种方式创建线程：  \* 开启线程的步骤：  \* 1、指定线程执行目标：定义一个Thread类的子类，重写run方法，将相关逻辑实现  \* public void run() 线程要执行的业务逻辑方法，相当于该线程的"main方法"  \* 2、创建自定义的线程子类对象  \* 3、开启线程动作  \* public void start() 使该线程开始执行  \*  \* Thread类的一些方法：  \* public final String getName() 获取线程名称  \* public final void setName(String name) 指定线程名称  \* public static Thread currentThread() 获取当前线程对象  \*  \* 第二种方式创建线程：  \* 开启线程的步骤：  \* 1、指定线程执行目标：定义Runnable线程执行目标实现类，重写run方法，指定目标逻辑  \* 2、通过指定线程执行目标的构造方法创建线程对象  \* public Thread(Runnable target)  \* a)创建线程执行目标对象  \* b)通过线程执行目标创建线程对象  \* 3、开启线程动作 (start开启线程)  \*/  **public** **class** ThreadDemo2 {  **public** **static** **void** main(String[] args) {    //开启线程  //创建线程执行目标  MyRunnable mr = **new** MyRunnable();  //通过指定线程执行目标的构造方法创建线程对象  Thread thread = **new** Thread(mr);  thread.setName("Jack");  Thread thread2 = **new** Thread(mr);  thread2.setName("Rose");    //开启线程动作  thread.start();  thread2.start();    //返回main线程  Thread mainThread = Thread.*currentThread*();    **for**(**int** i=0;i<50;i++){  //System.out.println("main:"+i);  System.***out***.println(mainThread.getName()+":"+i);  }    }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: MyRunnable  \* **@Description**: Runnable的实现类  \* **@date** 2018年2月4日 下午12:36:55  \* Company www.igeekhome.com  \*  \* 定义了线程执行目标实现类  \*/  **public** **class** MyRunnable **implements** Runnable{  /\*\*  \* **@Title**: run  \* **@Description**: 指定线程执行目标逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {    //返回当前线程  Thread thisThread = Thread.*currentThread*();    **for**(**int** i=0;i<50;i++){  //System.out.println("线程名称:"+i);  System.***out***.println(thisThread.getName()+":"+i);  }  }  } |

### 两种实现方式对比

继承Thread类，线程对象和线程任务耦合在一起。一旦创建Thread类的子类对象，既是线程对象，有又有线程任务。

实现Runnable接口，将线程任务单独分离出来封装成对象，类型就是Runnable接口类型。

Runnable接口对线程对象和线程任务进行解耦。

A：第二种方式实现Runnable接口避免了单继承的局限性，所以较为常用。

B：实现Runnable接口的方式，更加的符合面向对象，线程分为两部分，一部分线程对象，一部分线程任务。

### 利用匿名内部类创建线程

匿名内部类格式：

new 父类/父接口(){

//重写父类或父接口的方法

};

//不但定义了该父类/父接口的子类而且创建该子类的对象

#### 案例代码四:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: AnonymousThreadDemo  \* **@Description**: 利用匿名内部类创建线程  \* **@date** 2018年2月4日 下午1:13:08  \* Company www.igeekhome.com  \*  \* 使用匿名内部类开启线程  \*  \*/  **public** **class** AnonymousThreadDemo {  **public** **static** **void** main(String[] args) {  //方式一：  //使用匿名内部类创建线程的子类对象  Thread thread = **new** Thread(){  /\*\*  \* **@Title**: run  \* **@Description**: **TODO**(这里用一句话描述这个方法的作用)  \* **@see** java.lang.Thread#run()  \*/  @Override  **public** **void** run() {  System.***out***.println("我的线程执行了");  }    };  //开启线程  thread.start();    //使用匿名内部类创建线程的子类匿名对象  **new** Thread(){  /\*\*  \* **@Title**: run  \* **@Description**: **TODO**(这里用一句话描述这个方法的作用)  \* **@see** java.lang.Thread#run()  \*/  @Override  **public** **void** run() {  System.***out***.println("我的线程执行了2");  }    }.start();    //方式二：  //使用匿名内部类的方式，创建线程执行目标类的对象  Runnable runnable = **new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println("我的线程执行目标，执行了");  }  };    //通过目标创建线程对象  Thread thread2 = **new** Thread(runnable);  //开启线程  thread2.start();    //使用匿名内部类的方式，创建线程执行目标类的匿名对象  //通过目标创建线程对象  Thread thread3 = **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println("我的线程执行目标，执行了2");    }  });  //开启线程  thread3.start();    //使用匿名内部类的方式，创建线程执行目标类的匿名对象  //通过目标创建线程对象  //开启线程  **new** Thread(**new** Runnable() {    @Override  **public** **void** run() {  System.***out***.println("我的线程执行目标，执行了3");  }  }).start();  }  } |

## 多线程卖票案例

火车站有多个售票窗口，这些售票窗口一块卖100张票，即100张票被多个窗口共享。

用线程模拟售票窗口；用输出语句模拟卖出的票。

### 案例代码五:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: TicketDemo  \* **@Description**: 多线程模拟火车站卖票  \* **@date** 2018年2月4日 下午1:59:57  \* Company www.igeekhome.com  \*  \* 使用第二种创建并开启线程的方式，方便数据共享  \*  \* 1、定义卖票的线程执行目标  \* 在成员变量位置将票定义为数字100，卖一张票，该数字减1，一直到小于0为止。  \* 重写run方法，完成卖票逻辑  \* while(true){  \* if(有票){  \* //卖票  \* 打印，xxx线程正在卖第几张票  \* 每卖完一张票，要将票数-1  \* }else{  \* //没有票，跳出循环，结束程序  \* }  \* }  \* 2、创建卖票的线程执行目标对象  \* 3、使用该卖票线程执行目标对象创建多个线程  \* 4、开启多个线程  \*/  **public** **class** TicketDemo {  **public** **static** **void** main(String[] args) {  //创建卖票的线程执行目标对象  Ticket ticket = **new** Ticket();    //使用该卖票线程执行目标对象创建多个线程  Thread thread = **new** Thread(ticket,"Jack");  Thread thread2 = **new** Thread(ticket,"Rose");  Thread thread3 = **new** Thread(ticket,"Trump");    //开启多个线程  thread.start();  thread2.start();  thread3.start();  }  } |

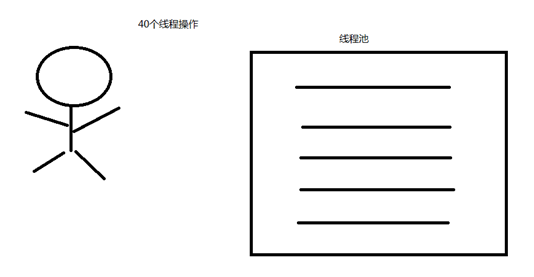
|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //有票就买票  **if**(number>0){  System.***out***.println(Thread.*currentThread*().getName()+"正在销售第"+(number--)+"张票");  }**else**{  //没有票，就跳出循环，不再卖票  **break**;  }  }  }  } |

# 线程池与线程生命周期

## 线程生命周期



## 线程池



A：线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。

B：我们详细的解释一下为什么要使用线程池？

在java中，如果每个请求到达就创建一个新线程，开销是相当大的。在实际使用中，创建和销毁线程花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，需要采取一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务。

线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重复使用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使用应用程序响应更快。另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。

### 利用Runnable接口向线程池提交任务

通常，线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。

A：Executors：线程池创建工厂类

public static ExecutorService newFixedThreadPool(int nThreads)： 返回线程池对象

B：ExecutorService：线程池类

Future<?> **submit**(Runnable task)：

获取线程池中的某一个线程对象，并执行

Future接口：

用来记录线程任务执行完毕后产生的结果。

C：使用线程池中线程对象的步骤（线程池创建与使用）：

创建线程池对象

创建Runnable接口实现类对象

提交Runnable接口实现类对象

关闭线程池

#### 案例代码六:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  /\*\*  \* **@ClassName**: ThreadPoolDemo  \* **@Description**: 线程池的使用  \* **@date** 2018年2月4日 下午3:48:52  \* Company www.igeekhome.com  \*  \* 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。  \*  \* Executors：线程池创建工厂类  \* 返回线程池方法：  \* public static ExecutorService newFixedThreadPool(int nThreads)：  \* 返回固定线程个数的线程池  \*  \* ExecutorService：线程池类  \* 线程池赋值线程的生命周期，我们只需要向线程提交执行目标，线程池会自动分配线程，执行对应的操作  \*  \* 1、定义线程执行目标  \* 这里使用卖票的线程执行目标  \* 2、向线程池提交线程  \* Future<?> submit(Runnable task):  \* 接收一个Runnable，执行该线程执行目标  \*  \*/  **public** **class** ThreadPoolDemo {  **public** **static** **void** main(String[] args) {  //返回一个线程池  ExecutorService threadPool = Executors.*newFixedThreadPool*(3);    //创建线程执行目标  Ticket ticket = **new** Ticket();    //向线程池提交任务  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);    //可以在适当的时候，关闭线程池  //threadPool.shutdown();  }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: Ticket  \* **@Description**: 卖票类  \* **@date** 2018年2月4日 下午2:14:50  \* Company www.igeekhome.com  \*  \* 定义卖票的线程执行目标类  \*/  **public** **class** Ticket **implements** Runnable{  //在成员变量位置将票定义为数字100  /\*\*  \* **@Fields** number :票数  \*/  **private** **int** number = 100;    /\*\*  \* **@Title**: run  \* **@Description**: 完成卖票的线程逻辑  \* **@see** java.lang.Runnable#run()  \*/  @Override  **public** **void** run() {  //车站不停地在卖票  **while**(**true**){  //有票就买票  **if**(number>0){  System.***out***.println(Thread.*currentThread*().getName()+"正在销售第"+(number--)+"张票");  }**else**{  //没有票，就跳出循环，不再卖票  **break**;  }  }  }  } |

### 利用Callable接口向线程池提交任务

A：Callable接口：

与Runnable接口功能相似，用来指定线程的任务。其中的call()方法，用来返回线程任务执行完毕后的结果，call方法可抛出异常。

B：ExecutorService：线程池类

<T> Future<T> **submit**(Callable<T> task)：

获取线程池中的某一个线程对象，并执行线程中的call()方法

Future接口：

用来记录线程任务执行完毕后产生的结果。

C：使用线程池中线程对象的步骤（线程池创建与使用）：

创建线程池对象

创建Callable接口实现类对象

提交Callable接口实现类对象

关闭线程池

#### 案例代码六:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.concurrent.ExecutionException;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.Future;  /\*\*  \* **@ClassName**: ThreadPoolDemo2  \* **@Description**: 线程池的使用  \* **@date** 2018年2月4日 下午3:48:52  \* Company www.igeekhome.com  \*  \* 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。  \*  \* Executors：线程池创建工厂类  \* 返回线程池方法：  \* public static ExecutorService newFixedThreadPool(int nThreads)：  \* 返回固定线程个数的线程池  \*  \* ExecutorService：线程池类  \* 线程池赋值线程的生命周期，我们只需要向线程提交执行目标，线程池会自动分配线程，执行对应的操作  \*  \* 1、定义线程执行目标  \* 这里使用卖票的线程执行目标  \* 2、向线程池提交线程  \* a、Future<?> submit(Runnable task):  \* 接收一个Runnable，执行该线程执行目标  \* b、<T> Future<T> submit(Callable<T> task)：  \* 接收一个Callable，执行该线程执行目标  \* Callable<T>:相当于有返回值的Runnable  \* V call() throws Exception  \* Future：任务的结果  \* V get() throws InterruptedException, ExecutionException  \* 返回call方法的结果  \*  \*/  **public** **class** ThreadPoolDemo2 {  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  //返回一个线程池  ExecutorService threadPool = Executors.*newFixedThreadPool*(3);    //创建线程执行目标  Ticket ticket = **new** Ticket();    //向线程池提交任务  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);  threadPool.submit(ticket);    //对应后边Callable的返回值Future，没有返回值的run方法，返回值是null，没有意义，不会这样使用  Future<?> ticketFuture = threadPool.submit(ticket);  System.***out***.println(ticketFuture.get());    //创建带返回值的线程执行目标  MyCallable callable = **new** MyCallable();  //向线程池提交任务，并返回线程执行目标的结果  Future<String> future = threadPool.submit(callable);  //从执行结果中返回call的具体返回值  String result = future.get();  System.***out***.println(result);    //可以在适当的时候，关闭线程池  //threadPool.shutdown();  }  } |

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.concurrent.Callable;  /\*\*  \* **@ClassName**: MyCallable  \* **@Description**: 我的Callable实现类  \* **@date** 2018年2月4日 下午4:11:39  \* Company www.igeekhome.com  \*  \* 定义线程执行目标  \*/  **public** **class** MyCallable **implements** Callable<String>{  /\*\*  \* **@Title**: call  \* **@Description**: 定义线程执行目标逻辑  \* **@return**  \* **@throws** Exception  \* **@see** java.util.concurrent.Callable#call()  \*/  @Override  **public** String call() **throws** Exception {  **return** "我是线程任务的返回值结果";  }  } |

重点和总结

1、多线程的基本概念

2、多线程的实现方式

3、线程的生命周期

4、线程池