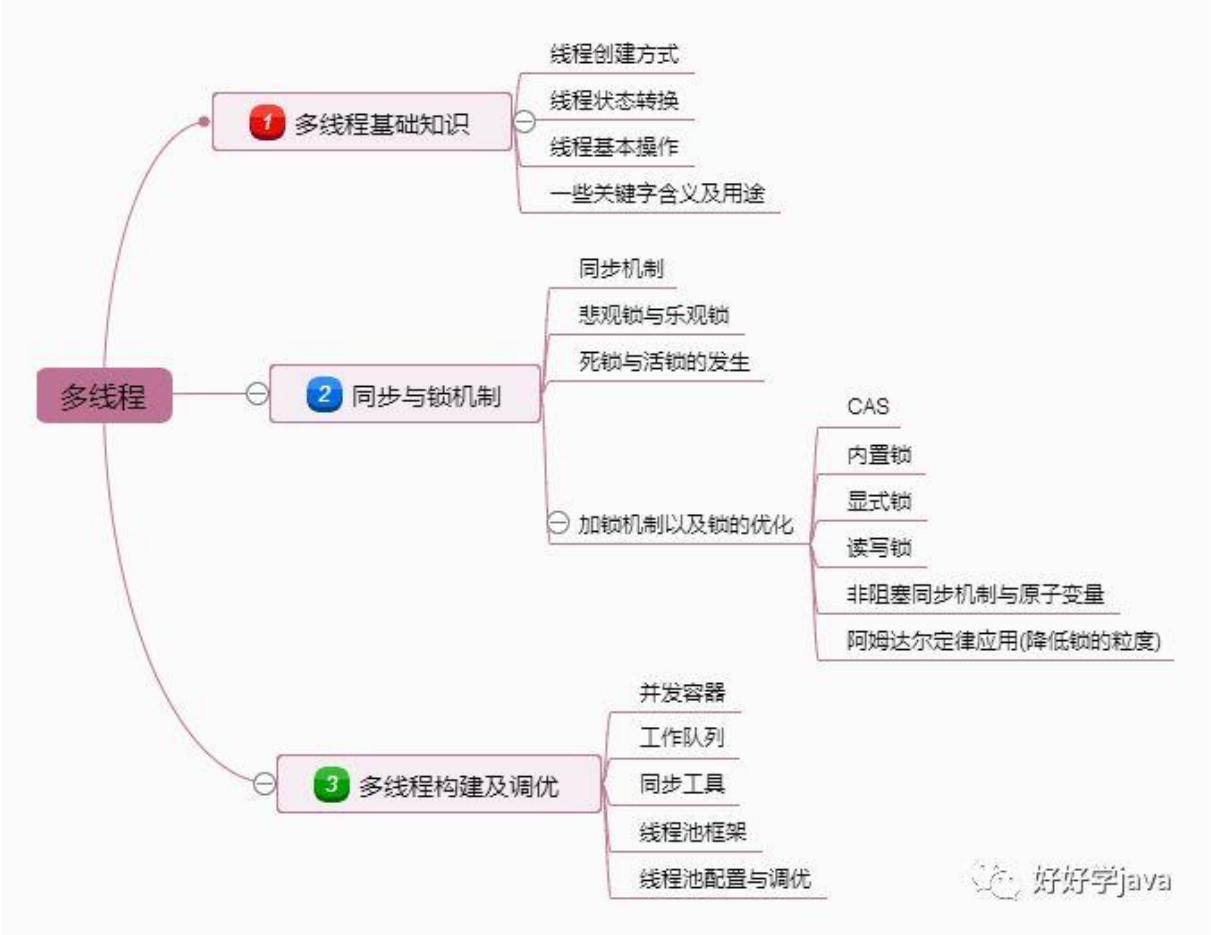
并发基础（1）：线程介绍

一、线程的简介



线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程 ID，当前指令指针（有一个程序计数器，它的作用是存放下一条指令所在单元的地址的地方），寄存器集合（寄存器是中央处理器内的组成部分。寄存器是有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和地址。在中央处理器的控制部件中，包含的寄存器有指令寄存器(IR)和程序计数器(PC)。在中央处理器的算术及逻辑部件中，存器有累加器(ACC)。）和堆栈组成。

另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约（共用资源造成的），致使线程在运行中呈现出间断性。线程也有就绪、阻塞和运行三种基本状态。就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；运行状态是指线程占有处理机正在运行；阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执行。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。

线程是程序中一个单一的顺序控制流程。进程内一个相对独立的、可调度的执行单元，是系统独立调度和分派 CPU 的基本单位指运行中的程序的调度单位。在单个程序中同时运行多个线程完成不

同的工作，称为多线程。

特点

在多线程 OS 中，通常是在一个进程中包括多个线程，每个线程都是作为利用 CPU 的基本单位，是花费最小开销的实体。线程具有以下属性。

1）轻型实体

线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证独立运行的资源。线程的实体包括程序、数据和 TCB。线程是动态概念，它的动态特性由线程控制块 TCB（Thread Control Block）描述。TCB 包括以下信息：

（1）线程状态。

（2）当线程不运行时，被保存的现场资源。

（3）一组执行堆栈。

（4）存放每个线程的局部变量主存区。

（5）访问同一个进程中的主存和其它资源。

用于指示被执行指令序列的程序计数器、保留局部变量、少数状态参数和返回地址等的一组寄存器和堆栈。

2）独立调度和分派的基本单位

在多线程 OS 中，线程是能独立运行的基本单位，因而也是独立调度和分派的基本单位。由于线程很“轻”，故线程的切换非常迅速且开销小（在同一进程中的）。

3）可并发执行

在一个进程中的多个线程之间，可以并发执行，甚至允许在一个进程中所有线程都能并发执行；同样，不同进程中的线程也能并发执行，充分利用和发挥了处理机与外围设备并行工作的能力。

4）共享进程资源。

在同一进程中的各个线程，都可以共享该进程所拥有的资源，这首先表现在：所有线程都具有相同的地址空间（进程的地址空间），这意味着，线程可以访问该地址空间的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。由于同一个进程内的线程共享内存和文件，所以线程之间互相通信不必调用内核。

与进程比较(容易混淆)

进程是资源分配的基本单位。所有与该进程有关的资源，都被记录在进程控制块 PCB 中。以表示该进程拥有这些资源或正在使用它们。

另外，进程也是抢占处理机的调度单位，(进程就像一个表演团队一样，而处理机可以形容为舞台，线程可以形容为表演团队的个人）它拥有一个完整的虚拟地址空间（进程可用的虚拟地址范围称为该进程的虚拟地址空间（当处理器读或写入内存位置时，它会使用虚拟地址。作为读或写操作的一部分，处理器将虚拟地址转换为物理地址。））。当进程发生调度时，不同的进程拥有不同的虚拟地址空间，而同一进程内的不同线程共享同一地址空间。

与进程相对应，线程与资源分配无关，它属于某一个进程，并与进程内的其他线程一起共享进程

的资源。

线程只由相关堆栈（系统栈或用户栈）寄存器和线程控制表 TCB 组成。寄存器可被用来存储线程内的局部变量，但不能存储其他线程的相关变量。

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位。由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度，从而显著提高系统资源的利用率和吞吐量。因而近年来推出的通用操作系统都引入了线程，以便进一步提高系统的并发性，并把它视为现代操作系统的一个重要指标。

线程与进程的区别可以归纳为以下 4 点：

1）地址空间和其它资源（如打开文件）：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。

2）通信：进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

3）调度和切换：线程上下文切换比进程上下文切换要快得多。

4）在多线程 OS 中，进程不是一个可执行的实体。

线程基础知识点可以总结为几点：

- 线程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。
- 每个线程都有一个优先级，高优先级线程的执行优先于低级优先级线程。但不应该通过设置线程优先级的方式来安排线程的执行顺序，后续将会细说。
- 每个线程都可以或者不标志为一个守护线程。即在 java 中，线程分为两类：用户线程 和 守护线程。
- 当 java 虚拟机启动时，都会有一个非守护线程（即用户线程）启动运行。这个线程通常是调用指定类的 main 方法。简单来说，当你执行一个类的 main 方法时，其实是作为一个线程在运行，即为 main 线程。
- 线程在操作系统中是不拥有资源， 线程是共享进程的资源。即属于同一个进程的多个线程之间是对资源可能要进行互斥访问。这就要涉及到锁的概念。同时在 JVM 中，除了用锁来解决并发的问题外，还可以让每个线程拥有私有资源---线程副本（ThreadLocal）,这样，线程就不需要竞争资源。

二、用户线程与守护线程

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

用个比较通俗的比如，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。

值得一提的是，守护线程并非只有虚拟机内部提供，用户在编写程序时也可以自己设置守护线程。下面的方法就是用来设置守护线程的。

```
Thread daemonTread = 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
    daemonThread.isDaemon();
```

这里有几点需要注意：

(1) thread.setDaemon(true) 必须在 thread.start() 之前设置，否则会跑出一个 IllegalThreadStateException 异常。你不能把正在运行的常规线程设置为守护线程。

(2) 在 Daemon 线程中产生的新线程也是 Daemon 的。

(3) 不要认为所有的应用都可以分配给 Daemon 来进行服务，比如读写操作或者计算逻辑。

因为你不可能知道在所有的 User 完成之前，Daemon 是否已经完成了预期的服务任务。一旦 User 退出了，可能大量数据还没有来得及读入或写出，计算任务也可能多次运行结果不一样。这对程序是毁灭性的。造成这个结果理由已经说过了：一旦所有 User Thread 离开了，虚拟机也就退出运行 //完成文件输出的守护线程任务.

```
import java.io.*;

class TestRunnable implements Runnable {
   public void run() {
       try {
           Thread.sleep(1000);// 守护线程阻塞 1 秒后运行
           File f = new File("daemon.txt");
           FileOutputStream os = new FileOutputStream(f, true);
           os.write("daemon".getBytes());
       } catch (IOException e1) {
           e1.printStackTrace();
       } catch (InterruptedException e2) {
           e2.printStackTrace();
       }
   }
}

public class TestDemo2 {
   public static void main(String[] args) throws InterruptedException {
       Runnable tr = new TestRunnable();
```

```
        Thread thread = new Thread(tr);
        thread.setDaemon(true); // 设置守护线程
        thread.start(); // 开始执行分进程
    }
}
//运行结果：文件 daemon.txt 中没有"daemon"字符串。
```

看到了吧，把输入输出逻辑包装进守护线程多么的可怕，字符串并没有写入指定文件。原因也很简单，直到主线程完成，守护线程仍处于 1 秒的阻塞状态。这个时候主线程很快就运行完了，虚拟机退出，Daemon 停止服务，输出操作自然失败了。

```
class MyCommon extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("线程 1 第" + i + "次执行！");
            try {
                Thread.sleep(7);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class MyDaemon implements Runnable {
    public void run() {
        for (long i = 0; i < 9999999L; i++) {
            System.out.println("后台线程第" + i + "次执行！");
            try {
                Thread.sleep(7);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Test {
    public static void main(String[] args){
        Thread t1 = new MyCommon();
        Thread t2 = new Thread(new MyDaemon());
```

```
        t2.setDaemon(true); //设置为守护线程
        t2.start();
        t1.start();
    }
}
```

输出结果：

```
后台线程第 0 次执行！
线程 1 第 0 次执行！
线程 1 第 1 次执行！
后台线程第 1 次执行！
后台线程第 2 次执行！
线程 1 第 2 次执行！
线程 1 第 3 次执行！
后台线程第 3 次执行！
线程 1 第 4 次执行！
后台线程第 4 次执行！
后台线程第 5 次执行！
后台线程第 6 次执行！
后台线程第 7 次执行！
Process finished with exit code 0
```

从上面的执行结果可以看出：

前台线程是保证执行完毕的，后台线程还没有执行完毕就退出了。

实际上： 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，而不管后台线程的状态，因此，在使用后台县城时候一定要注意这个问题。

补充说明：

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。

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。

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。

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。

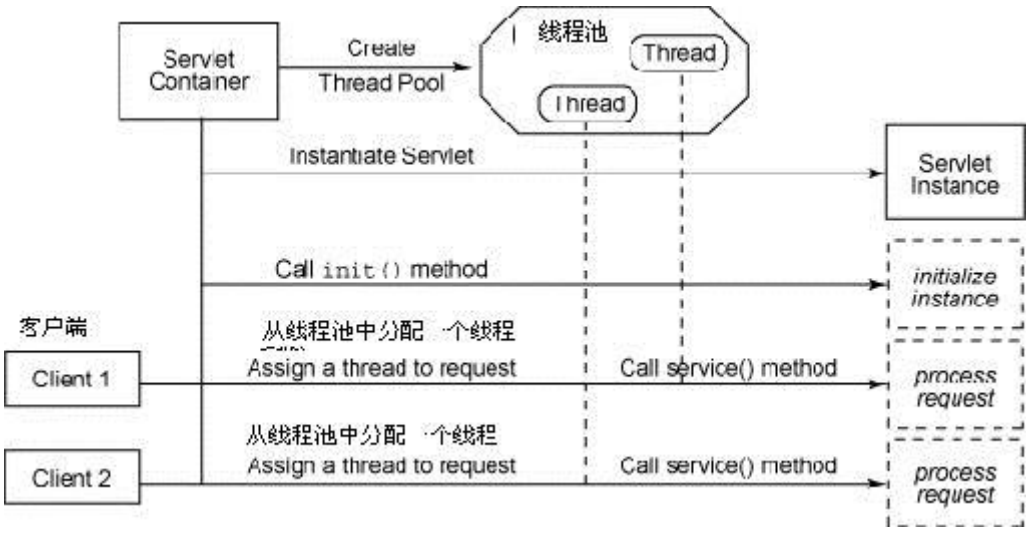
生命周期：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统“同生共死”。那 Java 的守护线程是什么样子的呢。当 JVM 中所有的线程都是守护线程的时候，JVM 就可以退出了；如果还有一个或以上的非守护线程则 JVM 不会退出。

三、实际应用例子

在使用长连接的 comet 服务端推送技术中，消息推送线程设置为守护线程，服务于 ChatServlet 的 servlet 用户线程，在 servlet 的 init 启动消息线程，servlet 一旦初始化后，一直存在服务器，servlet 摧毁后,消息线程自动退出

容器收到一个 Servlet 请求，调度线程从线程池中选出一个工作者线程,将请求传递给该工作者线程，然后由该线程来执行 Servlet 的 service 方法。当这个线程正在执行的时候,容器收到另外一个请求,调度线程同样从线程池中选出另一个工作者线程来服务新的请求,容器并不关心这个请求是否访问的是同一个 Servlet.当容器同时收到对同一个 Servlet 的多个请求的时候，那么这个 Servlet 的 service()方法将在多线程中并发执行。

Servlet 容器默认采用单实例多线程的方式来处理请求，这样减少产生 Servlet 实例的开销，提升了对请求的响应时间，对于 Tomcat 可以在 server.xml 中通过元素设置线程池中线程的数目。
如图：



我们知道静态变量是 ClassLoader 级别的，如果 Web 应用程序停止，这些静态变量也会从 JVM 中清除。但是线程则是 JVM 级别的，如果你在 Web 应用中启动一个线程，这个线程的生命周期并不会和 Web 应用程序保持同步。也就是说，即使你停止了 Web 应用，这个线程依旧是活跃的。正是因为这个很隐晦的问题，所以很多有经验的开发者不太赞成在 Web 应用中私自启动线程。

如果我们手工使用 JDK Timer（Quartz 的 Scheduler），在 Web 容器启动时启动 Timer，当 Web 容器关闭时，除非你手工关闭这个 Timer，否则 Timer 中的任务还会继续运行！

下面通过一个小例子来演示这个“诡异”的现象，我们通过 ServletContextListener 在 Web 容器启动时创建一个 Timer 并周期性地运行一个任务：

```
//代码清单 StartCycleRunTask：容器监听器
package com.baobaotao.web;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
```

```
public class StartCycleRunTask implements ServletContextListener ...{
    private Timer timer;
    public void contextDestroyed(ServletContextEvent arg0) ...{
        // ②该方法在 Web 容器关闭时执行
        System.out.println("Web 应用程序启动关闭...");
    }
    public void contextInitialized(ServletContextEvent arg0) ...{
         //②在 Web 容器启动时自动执行该方法
        System.out.println("Web 应用程序启动...");
        timer = new Timer();//②-1:创建一个 Timer，Timer 内部自动创建一个背景线程
        TimerTask task = new SimpleTimerTask();
        timer.schedule(task, 1000L, 5000L); //②-2:注册一个 5 秒钟运行一次的任务
    }
}
class SimpleTimerTask extends TimerTask ...{//③任务
    private int count;
    public void run() ...{
        System.out.println((++count)+"execute task..."+(new Date()));
    }
}
```

在 web.xml 中声明这个 Web 容器监听器：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<listener>
<listener-class>com.baobaotao.web.StartCycleRunTask</listener-class>
</listener>
</web-app>
```

在 Tomcat 中部署这个 Web 应用并启动后，你将看到任务每隔 5 秒钟执行一次。
运行一段时间后，登录 Tomcat 管理后台，将对应的 Web 应用（chapter13）关闭。

转到 Tomcat 控制台，你将看到虽然 Web 应用已经关闭，但 Timer 任务还在我行我素地执行如故——舞台已经拆除，戏子继续表演：

我们可以通过改变清单 StartCycleRunTask 的代码，在 contextDestroyed(ServletContextEvent arg0)中添加 timer.cancel()代码，在 Web 容器关闭后手工停止 Timer 来结束任务。

Spring 为 JDK Timer 和 Quartz Scheduler 所提供的 TimerFactoryBean 和 SchedulerFactoryBean 能够和 Spring 容器的生命周期关联，在 Spring 容器启动时启动调度器，而在 Spring 容器关闭时，停止调度器。所以在 Spring 中通过这两个 FactoryBean 配置调度器，再从 Spring IoC 中获取调度器引用进行任务调度将不会出现这种 Web 容器关闭而任务依然运行的问题。而如果你在程序中直接使用 Timer 或 Scheduler，如不进行额外的处理，将会出现这一问题。