

Java 并发编程核心

一、基本概念

1.1、什么是进程？

假如有两个程序 A 和 B，程序 A 执行到一半的过程中，需要读取大量的数据输入(I/O 操作)，而此时 CPU 只能静静地等待读取完数据才能继续执行，这样就白白浪费了 CPU 资源。是不是在程序 A 读取的过程中，让程序 B 去执行，当程序 A 读取完数据之后，让程序 B 暂停，然后让程序 A 继续执行？

当然没问题，但这里有一个关键词：切换(I/O 切换或时间轮询切换)。

既然是切换，那么就涉及到了状态的保存，状态的恢复，加上程序 A 与程序 B 所需要的系统资源(内存、硬盘、键盘等等)是不一样的，自然而然的就需要有一个东西去记得程序 A 和程序 B 分别需要什么资源，怎么去识别两个程序，所以就有了一个叫进程的抽象。

即使可以利用的 cpu 只有一个（早期的计算机确实如此），也能保证支持（伪）并发的能力。将一个单独的 cpu 变成多个虚拟的 cpu（多道技术：时间多路复用和空间多路复用+硬件上支持隔离），没有进程的抽象，现代计算机将不复存在。

举个例子（单核+多道，实现多个进程的并发执行）：

张三在一个时间段内有很多任务要做：写代码，看书，交女朋友，王者荣耀上分...

但同一时刻只能做一个任务（cpu 同一时间只能干一个活），如何才能玩出多个任务并发执行的效果？

写一会代码，再去跟某一个女孩聊聊天，再去打一会王者荣耀，英雄死了再看看书....这就保证了每个任务都在进行中。

1.2、进程与程序的区别

程序仅仅只是一堆代码而已，而进程指的是程序的运行过程。(抽象的概念)

tips：程序是程序，进程是进程，程序不运行，永远不是进程。

进程定义：

进程即程序(软件)在一个数据集上的一次动态执行过程。进程是对正在运行程序的一个抽象。进程一般由程序、数据集、进程控制块三部分组成。一个进程是一份独立的内存空间。多个进程之间用户程序无法互相操作。

程序：我们编写的程序用来描述进程要完成哪些功能以及如何完成；

数据集：是程序在执行过程中所需要使用的资源；

进程控制块：用来记录进程的外部特征，描述进程的执行变化过程，系统可以利用它来控制和管理进程，它是系统感知进程存在的唯一标志。

举个栗子：

想象一位厨师正在为他的女儿烘制生日蛋糕

他有做生日蛋糕的食谱，厨房里有所需的原料:面粉、鸡蛋、韭菜、蒜泥等

在这个比喻中：

做蛋糕的食谱就是程序(即用适当形式描述的算法)

厨师就是处理器(cpu)

而做蛋糕的各种原料就是输入数据

进程就是厨师阅读食谱、取来各种原料以及烘制蛋糕等一系列动作的总和

现在假设厨师的儿子哭着跑了进来，说：头被蜜蜂蛰了。

厨师想了想，处理儿子蛰伤的任务比给女儿做蛋糕的任务更重要，于是厨师就记录下他照着食谱做到哪儿了(保存进程的当前状态)，然后拿出一本急救手册，按照其中的指示处理蛰伤。这里，我们看到处理器(cpu)从一个进程(做蛋糕)切换到另一个高优先级的进程(实施医疗救治)，每个进程拥有各自的程序(食谱和急救手册)。当蜜蜂蛰伤处理完之后，这位厨师又回来做蛋糕，从他离开时的那一步继续做下去。

需要强调的是：同一个程序执行两次，那也是两个进程，比如打开 PotPlayer，虽然都是同一个软件，但是一个可以播放 Java，一个可以播放 Python。

1.3、什么是线程？

进程只是用来把资源集中到一起（进程只是一个资源单位，或者说资源集合），而线程才是 CPU 上的执行单位。在操作系统中，每个进程有一个地址空间，而且默认就有一个控制线程（主线程）。多线程（即多个控制线程）的概念是，在一个进程中存在多个控制线程，多个控制线程共享该进程的地址空间。线程不能够独立执行，必须依存在进程中。一个线程可以创建和撤销另一个线程（子线程），同一个进程中的多个线程之间可以并发执行。

举个栗子：一条流水线工作的过程

一条流水线（线程）必须属于一个车间，一个车间的工作过程是一个进程，车间负责把资源整合到一起，是一个资源单位，而一个车间内至少有一个流水线，流水线的工作需要电源，电源就相当于 CPU，多线程相当于一个车间内有多条流水线，都共用一个车间的资源。

再比如：

开启一个文本处理软件进程，该进程肯定需要办不止一件事情，比如监听键盘输入，处理文字显示，定时自动将文字保存到硬盘，这三个任务操作的都是同一块数据，因而不能用多进程。只能在一个进程里并发地开启三个线程，如果是单线程，那就只能是，键盘输入时，不能处理文字和自动保存，自动保存时又不能输入和处理文字。

1.4、并发和并行

无论是并行还是并发，在用户看来都是‘同时’运行的，不管是进程还是线程，都只是一个任务而已，真是干活的是 cpu，cpu 来做这些任务，而一个 cpu 同一时刻只能执行一个任务。

4.1、并发（串行）

是伪并行，即看起来是同时运行。单个 cpu+多道技术就可以实现并发。

渣男例：

你是一个 cpu，你同时谈了三个女朋友，每一个都可以是一个恋爱任务，你被这三个任务共享，要玩出发恋爱效果，应该是你先跟女友 1 去看电影，看了一会：不好，我要拉肚子，然后跑去跟第二个女友吃饭，吃了一会：那啥，我去趟洗手间，然后跑去跟女友 3 喝下午茶。

4.2、并行

同时运行，只有具备多个 cpu 才能实现并行

单核下，可以利用多道技术，多个核，每个核也都可以利用多道技术（多道技术是针对单核而言的）

有四个核，六个任务，这样同一时间有四个任务被执行，假设分别被分配给了 cpu1，cpu2，cpu3，cpu4，可能任务 1 遇到 I/O 就被迫中断执行，此时任务 5 就拿到 cpu1 的

时间片去执行，这就是单核下的多道技术，而一旦任务 1 的 I/O 结束了，操作系统会重新调用它(进程的调度、分配给哪个 cpu 运行，由操作系统说了算)，可能被分配给四个 cpu 中的任意一个去执行。

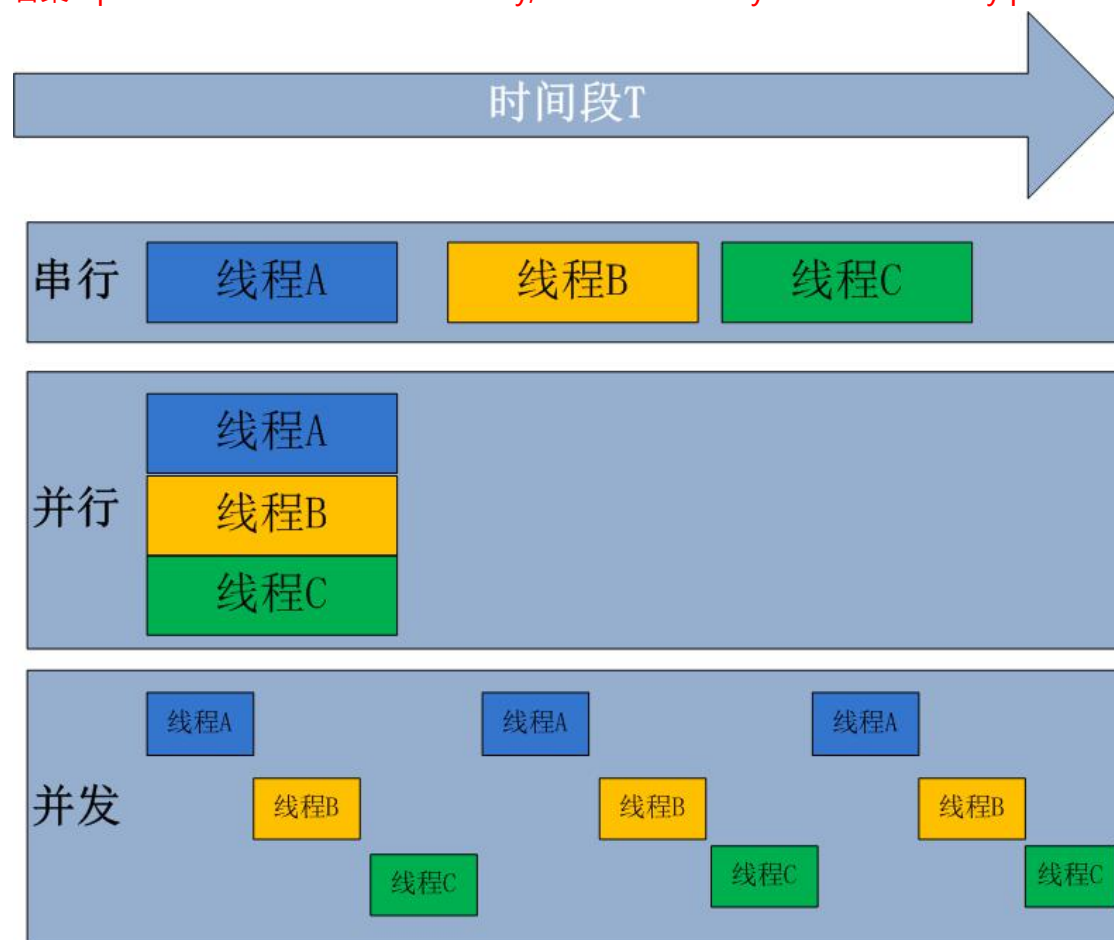
4.3、总结

并发：系统具有处理多个任务的能力

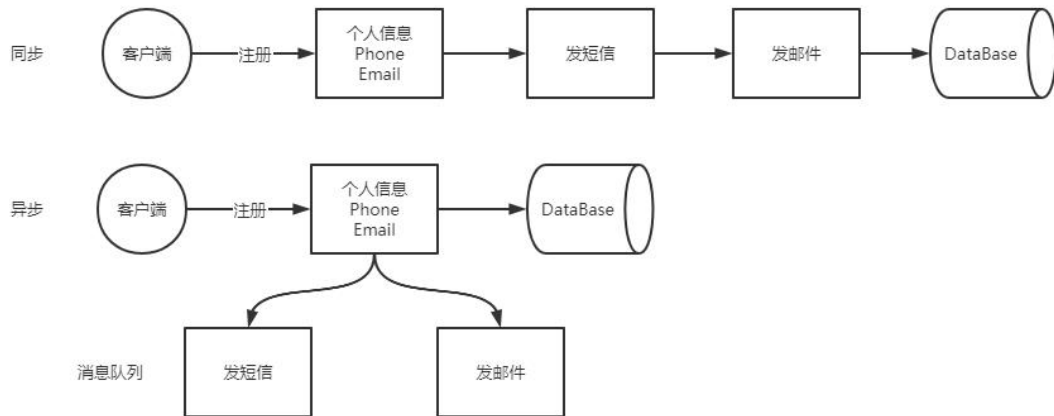
并行：系统具有同时处理多个任务的能力

思考：并发是不是并行的一个子集？并行是不是并发的一个子集？

答案：parallelism must be concurrency, and concurrency is not necessarily parallel



1.5、同步和异步



1.5.1、同步

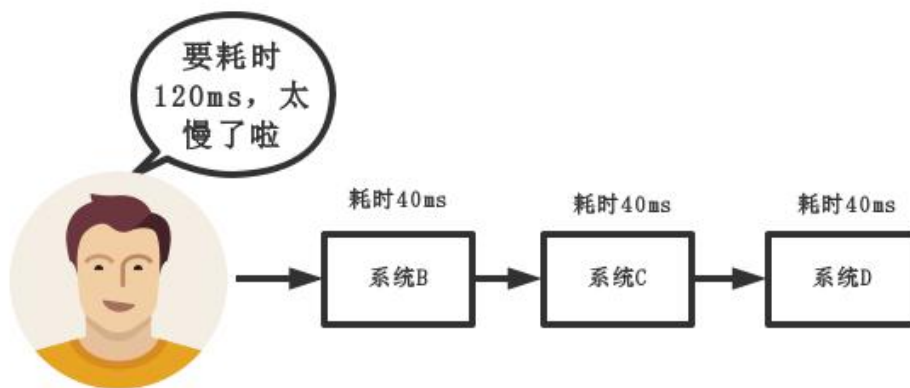
同步的思想是：所有的操作都做完，才返回给用户。这样用户在线等待的时间太长，给用户一种卡死了的感觉（就是系统迁移中，点击了迁移，界面就不动了，但是程序还在执行，卡死了的感觉）。这种情况下，用户不能关闭界面，如果关闭了，即迁移程序就中断了。

1.5.2、异步

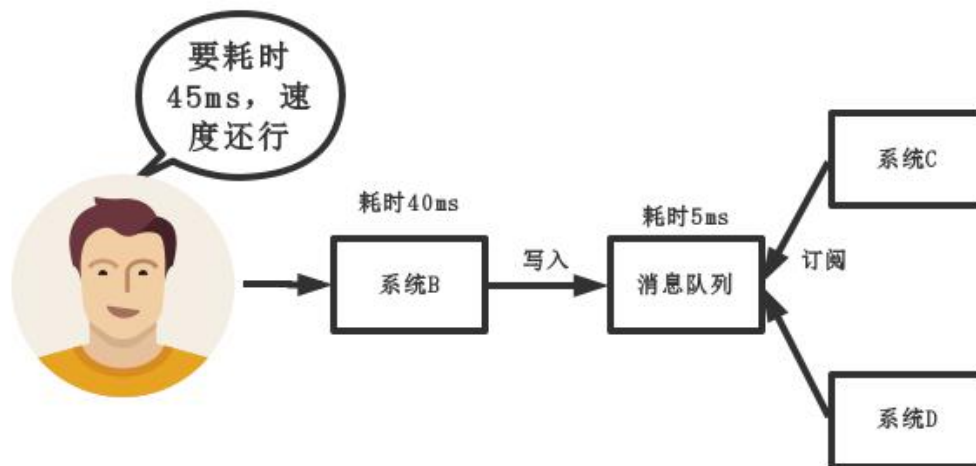
将用户请求放入消息队列，并反馈给用户友好提示，系统迁移程序启动，你可以关闭浏览器了。然后程序再慢慢地去写入数据库去。这就是异步。用户没有卡死的感觉，会告诉你，你的请求系统已经响应了。你可以关闭界面了。

1.5.3、总结

同步，是所有的操作都做完，才返回给用户结果。即写完数据库之后，在响应用户，用户体验不好。



异步，不用等所有操作等做完，就响应用户请求。即先响应用户请求，然后慢慢去写数据库，用户体验较好。



异步操作例子：

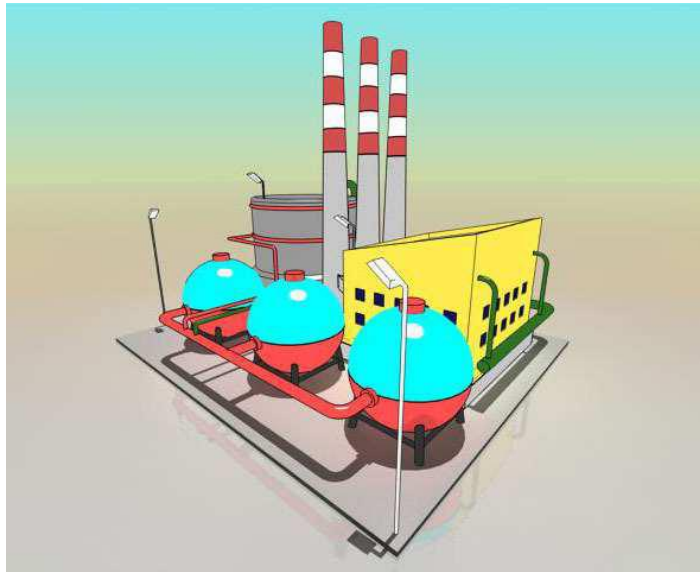
为了避免短时间大量的数据库操作，可以使用缓冲机制，也就是消息队列。先将数据放入消息队列，然后再慢慢写入数据库。

引入消息队列机制，虽然可以保证用户请求的快速响应，但是并没有使得我数据迁移的时间变短（即 80 万条数据写入 mysql 需要 1 个小时，用了缓冲机制之后，还是需要 1 个小时，只是保证用户的请求的快速响应。用户输入完 http url 请求之后，就可以把浏览器关闭了，干别的去了。如果不用缓冲机制，浏览器不能关闭）。

同步就没有任何价值了吗？

有。比如银行的转账功能。打电话等。

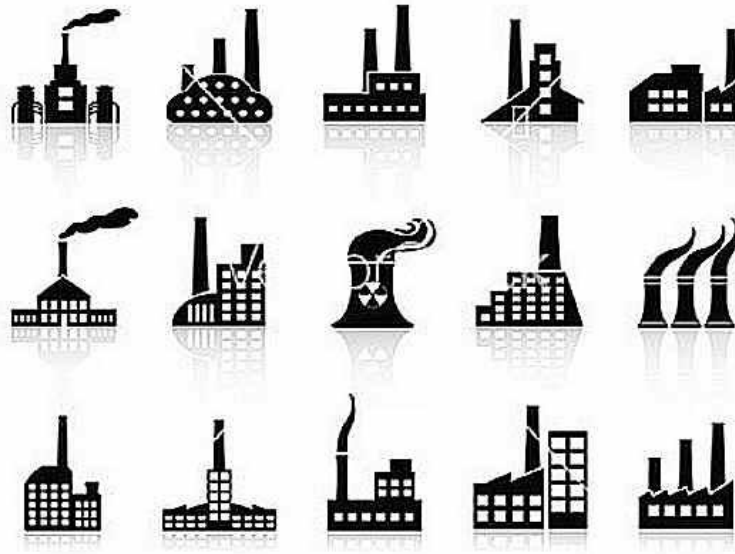
1.6、面试题：有了进程为什么还需要线程？



计算机的核心是 CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。



假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个 CPU 一次只能运行一个任务。



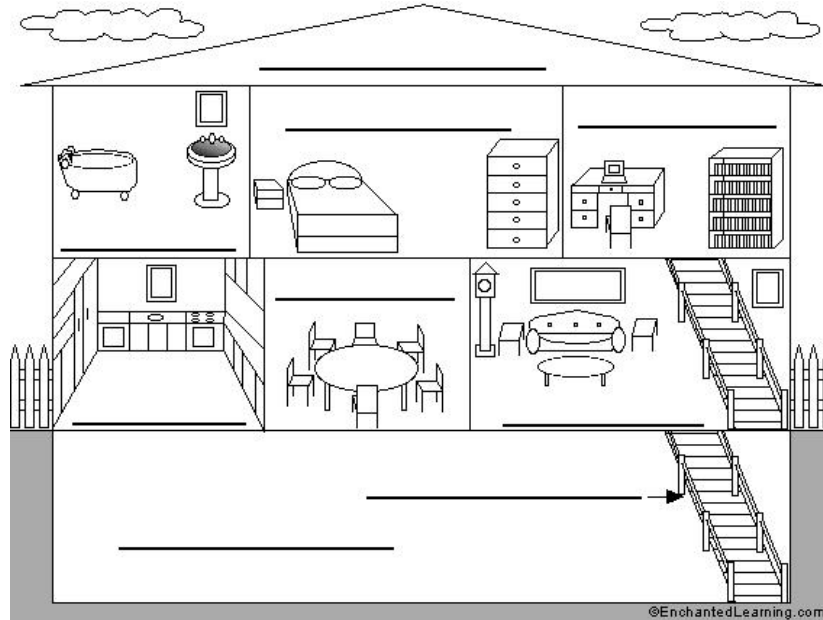
进程就好比工厂的车间,它代表 CPU 所能处理的单个任务。任一时刻,CPU 总是运行一个进程,其他进程处于非运行状态。



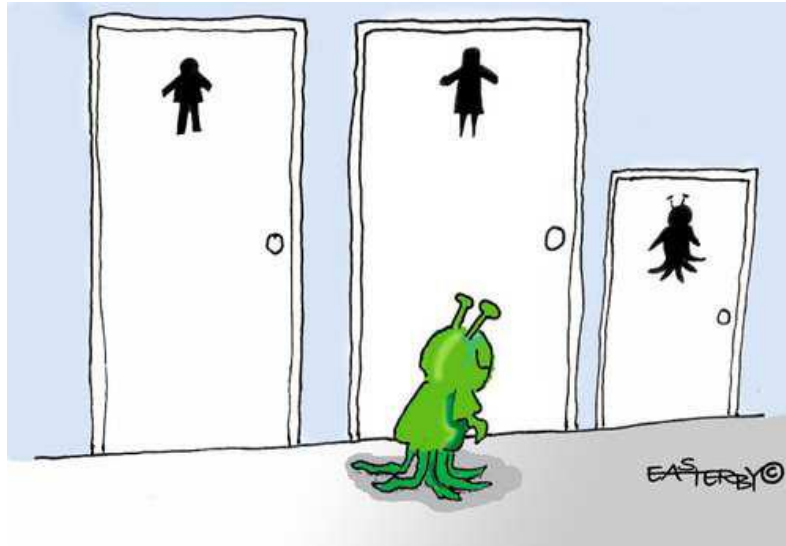
一个车间里,可以有很多工人。他们协同完成一个任务。



线程就好比车间里的工人。一个进程可以包括多个线程。



车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。



可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人时，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。



一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫“互斥锁”（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。



还有些房间，可以同时容纳 n 个人，比如厨房。也就是说，如果人数大于 n ，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。



这时的解决方法，就是在门口挂 n 把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做“信号量”（Semaphore），用来保证多个线程不会互相冲突。

不难看出，mutex 是 semaphore 的一种特殊情况（ $n=1$ 时）。也就是说，完全可以用后者替代前者。但是，因为 mutex 较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。

总结：

1. 从资源上来讲：线程是一种非常“节俭”的多任务操作方式。而进程的创建需要更多的资源。
2. 从切换效率上来讲：运行于一个进程中的多个线程，它们之间使用相同的

地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的 30 倍左右。

3. 从通信机制上来讲：对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。

二、多线程基础

在计算机的世界里，当我们探讨并行的时候，实际上是指一系列的任务在计算机中同时运行，比如在浏览网页的时候还能打开音乐播放器，在撰写邮件的时候，收件箱还能接收新的邮件。在单 CPU 的计算机中，其实并没有真正的并行，它只不过是 CPU 时间片轮转机制带给你的错觉，而这种错觉让你产生了它们真的在同一时刻同时运行。当然如果是多核 CPU，那么并行运行还是真实存在的。

2.1、创建并启动线程

案例：假设你想在浏览网页看新闻的同时还想听听音乐。

2.1.1、尝试并行运行

```
package com.bjsxt.chapter01;

public class TryConcurrency01 {

    public static void main(String[] args) {
        browseNews();
        enjoyMusic();
    }

    // 浏览新闻
    private static void browseNews() {
        while (true) {
            System.out.println("Uh huh. The good news.");
        }
    }
}
```

```
        try {  
            Thread.sleep(1000L);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}  
  
// 欣赏音乐  
private static void enjoyMusic() {  
    while (true) {  
        System.out.println("Uh huh. The nice music.");  
        try {  
            Thread.sleep(1000L);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

以上的程序输出永远都是在看新闻,而听音乐的任务永远都得不到执行。输出结果如下:

```
Uh huh. The good news.  
Uh huh. The good news.  
Uh huh. The good news.
```

2.1.2、并行运行

如果能让听音乐和看新闻两个事件并发执行,必须借助 Java 提供的 Thread 这个类(后面我们会详细讲解 Thread),只需将 main 方法中的任意一个方法交给 Thread 即可。

```
package com.bjsxt.chapter01;  
  
public class TryConcurrency02 {  
  
    public static void main(String[] args) {
```



```
// JDK 1.8 之前
/*
new Thread() {
    @Override
    public void run() {
        browseNews();
    }
}.start();
*/

// JDK 1.8 Lambda 编码方式
new Thread(TryConcurrency02::browseNews).start();

enjoyMusic();
}

// 浏览新闻
private static void browseNews() {
    while (true) {
        System.out.println("Uh huh. The good news.");
        try {
            Thread.sleep(1000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// 欣赏音乐
private static void enjoyMusic() {
    while (true) {
        System.out.println("Uh huh. The nice music.");
        try {
            Thread.sleep(1000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

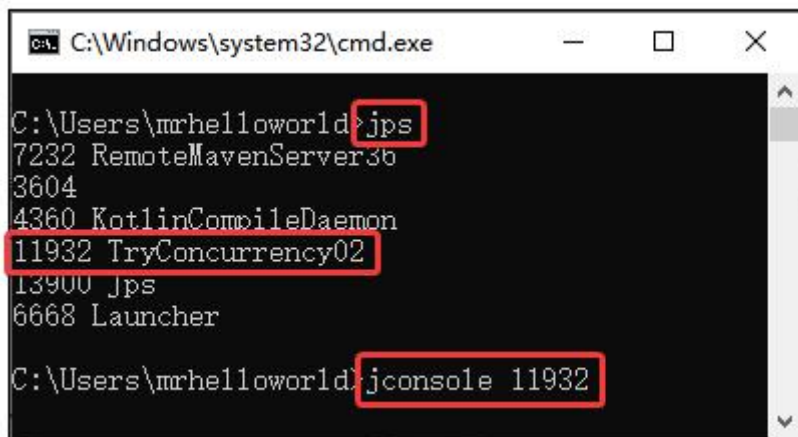
```
}
}
}
```

输出结果如下：

```
Uh huh. The nice music.
Uh huh. The good news.
Uh huh. The nice music.
Uh huh. The good news.
Uh huh. The good news.
Uh huh. The nice music.
```

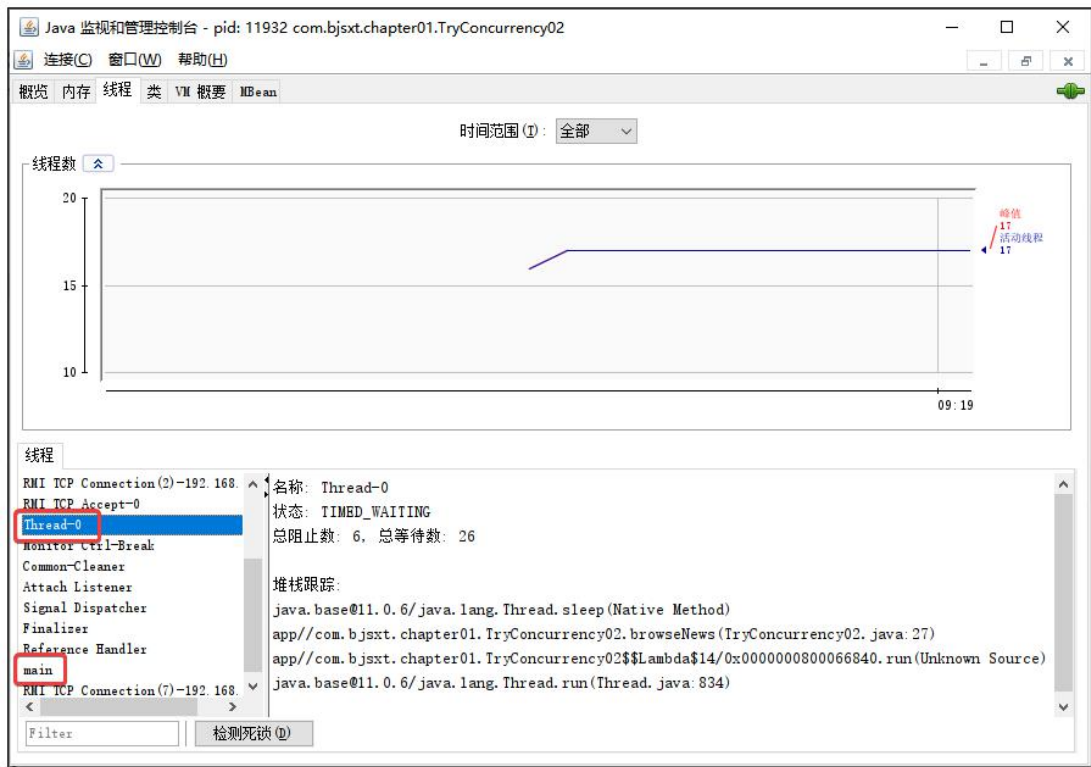
2.2、使用 Jconsole 观察线程

2.1.2 中的代码创建了一个 Thread 并启动，那么此时 JVM 中有多少个线程呢？我们可以借助 Jconsole 或者 Jstack 命令来查看，这两个 JVM 工具都是由 JDK 自身提供，如下图所示：



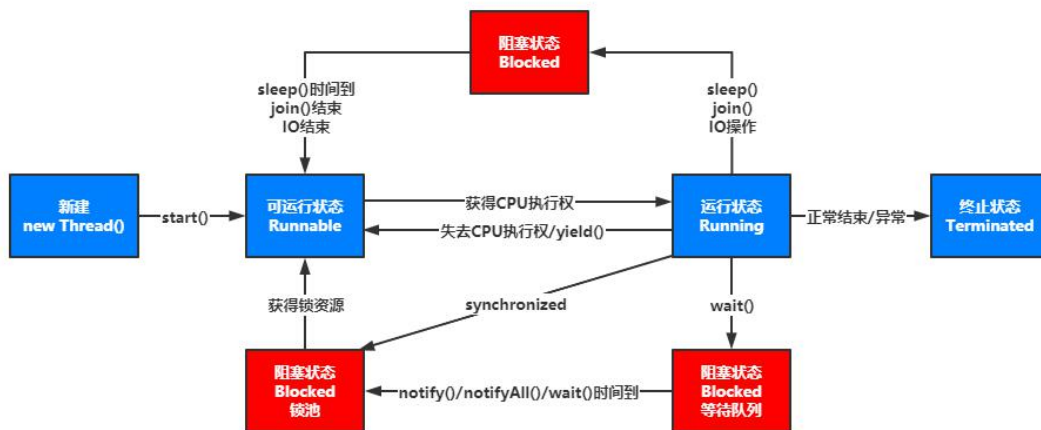
```
C:\Windows\system32\cmd.exe
C:\Users\mrhelloworld>jps
7232 RemoteMavenServer3b
3604
4360 KotlinCompileDaemon
11932 TryConcurrency02
13900 Jps
6668 Launcher
C:\Users\mrhelloworld>jconsole 11932
```

如下图所示，操作系统启动一个 Java 虚拟机（JVM）的时候，其实启动了一个进程，而在该进程里面启动了一个以上的线程，其中 Thread-0 这个线程就是 2.1.2 中创建的，main 线程是由 JVM 启动的时候创建的，当然还有一些其他的守护线程，比如垃圾回收线程、RMI 线程等等。



2.3、线程生命周期

在开始解释线程的生命周期之前，请大家思考一个问题：执行了 Thread 的 start 方法就代表该线程已经开始执行了吗？



2.3.1、NEW

当我们用关键字 new 创建一个 Thread 对象时，此时它并不处于执行状态，因为没有

调用 start 方法启动该线程，此时线程的状态为 new 状态，它只是 Thread 对象的状态，因为在没有 start 之前，该线程根本不存在，这与你用关键字 new 创建一个普通的 Java 对象没有任何区别。

new 状态时通过 start 方法进入 Runnable 状态。

2.3.2、RUNNABLE

线程对象进入 Runnable 状态必须调用 start 方法，此时才真正地在 JVM 进程中创建了一个线程。线程一经启动就会立即得到执行吗？答案是否定的，线程的运行与否和进程一样都要听令于 CPU 的调度，所以我们把这个中间状态称为可执行状态(Runnable)，也就是说它具备执行的资格，但是并没有真正的执行起来而是在等待 CPU 的调度。

由于存在 Running 状态，所以不会直接进入 Blocked 状态和 Terminated 状态，即使是在线程的执行逻辑中调用 wait、sleep 或者其他的 block 的 IO 操作等，也必须先获得 CPU 的调度执行权才可以，严格来讲，Runnable 的线程只能意外终止或者进入 Running 状态。

2.3.3、RUNNING

一旦 CPU 通过轮询或其他方式从任务可执行队列中选中了线程，那么此时它才能真正地执行 run 方法里的逻辑代码。在该状态中，线程的状态可以发生如下的状态转换。

- a. 直接进入 Terminated 状态，比如调用 JDK 已经不推荐使用的 stop 方法或者意外死亡；
- b. 进入 Blocked 状态，比如调用 sleep 或者 wait 方法而加入了 waitSet 中；
- c. 进行某个阻塞的 IO 操作，比如因网络数据的读写而进入了 Blocked；
- d. 获取某个锁资源，从而加入到该锁的阻塞队列中而进入了 Blocked；
- e. 由于 CPU 的调度器轮询使该线程放弃执行，进入 Runnable 状态；
- f. 线程主动调用 yield 方法，放弃 CPU 执行权，进入 Runnable；

2.3.4、BLOCKED

上面已经介绍了线程进入 Blocked 状态的原因，这里不在赘述。线程在 Blocked 状态中可以切换至如下几个状态。

- a. 直接进入 Terminated 状态，比如调用 JDK 已经不推荐使用的 stop 方法或者意外死亡；
- b. 线程阻塞的操作结束，比如读取了想要的数据字节进入到 Runnable；

- c. 线程完成了指定时间的休眠，进入到了 Runnable；
- d. wait 中的线程被其他线程 notify/notifyall 唤醒，进入 Runnable；
- e. 线程获取到了某个锁资源，进入 Runnable；
- f. 线程在阻塞过程中被打断，比如其他线程调用了 interrupt 方法，进入 Runnable。

2.3.5、TERMINATED

Terminated 是一个线程的最终状态，在该状态中线程将不会切换到其他任何状态，线程进入 Terminated，意味着该线程的生命周期都结束了，下面这些情况会使线程进入 Terminated 状态。

- a. 线程运行正常结束，结束生命周期；
- b. 线程运行出错，意外结束；
- c. JVM Crash,导致所有的线程都结束。

2.4、线程 start 方法源码剖析

```
/**
 * Causes this thread to begin execution; the Java Virtual Machine
 * calls the {@code run} method of this thread.
 * 使此线程开始执行；Java 虚拟机调用这个线程的 run 方法。
 * <p>
 * The result is that two threads are running concurrently: the
 * current thread (which returns from the call to the
 * {@code start} method) and the other thread (which executes its
 * {@code run} method).
 * <p>
 * It is never legal to start a thread more than once.
 * 多次调用 start 方法启动一个线程是非法的。
 * In particular, a thread may not be restarted once it has completed
 * execution.
 * 特别是，线程在执行完成后不能重新启动。
 *
 * @throws  IllegalThreadStateException  if the thread was already started.
 * 已经启动的线程再次 start 会抛出线程非法状态异常。
 * @see      #run()
```



```

* @see      #stop()
*/
public synchronized void start() {
    /**
     * This method is not invoked for the main method thread or "system"
     * group threads created/set up by the VM. Any new functionality added
     * to this method in the future may have to also be added to the VM.
     *
     * A zero status value corresponds to state "NEW".
     */
    if (threadStatus != 0) // 状态校验 0 表示 NEW 新建状态
        throw new IllegalStateException();

    /* Notify the group that this thread is about to be started
     * so that it can be added to the group's list of threads
     * and the group's unstarted count can be decremented. */
    group.add(this); // 添加进线程组

    boolean started = false;
    try {
        start0(); // 调用 native 方法执行线程 run 方法
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this); // 启动失败,从线程组中移除当前线程
            }
        } catch (Throwable ignore) {
            /* do nothing. If start0 threw a Throwable then
             it will be passed up the call stack */
        }
    }
}

private native void start0(); // native 方法, C++ 程序执行

```

总结：

start 方法的源码特别简单，其实最核心的部分就是 start0 这个本地方法，也就是 JNI

方法：`private native void start0()`;

也就是说在 `start` 方法中会调用 `start0` 方法，那么重写的那个 `run` 方法何时被调用了呢？JDK 官方文档是这样说的：Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

上面这句话的意思是：在开始执行这个线程时；Java 虚拟机将会调用这个线程的 `run` 方法。换言之，`run` 方法就是被 JNI 方法 `start0` 调用的。仔细阅读源码会总结出以下几点：

- a. Thread 被构造后的 NEW 状态，`threadStatus` 这个内部属性为 0；
- b. 不能两次启动 Thread，否则就会出现 `IllegalThreadStateException` 异常；
- c. 线程启动后将会被加入到一个 `ThreadGroup` 中，后文中我们会详细介绍它；
- d. 一个线程生命周期结束，也就是到了 `Terminated` 状态，再次调用 `start` 方法是非法的，也就是说 `Terminated` 状态是没有办法回到 `Runnable/Running` 状态的。

2.5、模板方法模式

通过阅读源码发现，线程真正的执行逻辑是在 `run` 方法中，而启动线程的却是 `start` 方法，为什么这么设计呢？这就是我们接下来要讲的模板方法设计模式。

2.5.1、概念

它定义了一个操作算法的框架，将一些步骤延迟到子类中执行。使得子类可以不改变一个算法的结构即可重定义该算法的某些步骤。还可以通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制。

比如启动线程 `start` 的行为由 JVM 虚拟机来决定，但是该线程做什么事情就不由它来决定了。对外提供了 `run` 方法，内部就是你自己线程的执行逻辑。下面我们写一个 Demo 方便大家理解。

2.5.3、案例

我们拿做饭的例子来说明，同样的步骤其实不同的人做出来的饭是不一样的。就拿自己和五星级大厨来比较吧。比如就做个西红柿鸡蛋，我们可以简单地定义一下步骤：

- 第一步：放油
- 第二步：放鸡蛋
- 第三步：放西红柿

2.5.3.1、分析

如果按照模板方法的思路去构建，我们需要剥离出两个角色：

1. 模板方法：父类，定义一系列方法，提供骨架；
2. 具体类：实现模板方法类提供的骨架。根据自己的个性化需求重写模板方法。

2.5.3.2、实现

a. 定义模板方法类（Cook 骨架）

```
package com.bjsxt.chapter02;

/**
 * 模板方法类
 */
public abstract class Cook {

    abstract void oil(); // 食用油

    abstract void egg(); // 鸡蛋

    abstract void tomato(); // 西红柿

    // 封装具体的行为：做饭
    public final void cook() {
        this.oil();
        this.egg();
        this.tomato();
    }
}
```

b. 定义具体类（我和大厨）

我做饭：

```
package com.bjsxt.chapter02;
```

```
/**
 * 具体类
 */
public class Me extends Cook {

    @Override
    void oil() {
        System.out.println("自己：油放多了！");
    }

    @Override
    void egg() {
        System.out.println("自己：鸡蛋炒糊了！");
    }

    @Override
    void tomato() {
        System.out.println("自己：西红柿放多了！");
    }

}
```

大厨做饭：

```
package com.bjsxt.chapter02;

/**
 * 具体类
 */
public class Chef extends Cook {

    @Override
    void oil() {
        System.out.println("厨师：油适量！");
    }

    @Override
    void egg() {
```

```
        System.out.println("厨师：鸡蛋适量！");
    }

    @Override
    void tomato() {
        System.out.println("厨师：西红柿适量！");
    }
}
```

c. 测试

```
package com.bjsxt.chapter02;

public class TestCook {

    public static void main(String[] args) {
        new Me().cook();
        new Chef().cook();
    }
}
```

```
自己：油放多了！
自己：鸡蛋炒糊了！
自己：西红柿放多了！
厨师：油适量！
厨师：鸡蛋适量！
厨师：西红柿适量！
```

看结果我们就能知道，炒西红柿鸡蛋的过程是一样的，但是实现起来却不一样，这就是模板方法模式。

2.5.4、钩子函数

前面我们讲到，模板方法还可以通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制。这个功能我们可以通过钩子函数来实现。

钩子就是给子类一个授权，让子类来决定模板方法的逻辑执行。比如在炒西红柿鸡蛋的时候，由子类去决定是否要加调料。我们去实现一下：

a. 修改模板类

```
package com.bjsxt.chapter03;

/**
 * 模板方法类
 */
public abstract class Cook {

    abstract void oil(); // 食用油

    abstract void egg(); // 鸡蛋

    abstract void tomato(); // 西红柿

    // 钩子函数：子类可以覆写，让子类决定是否添加油，默认加油
    boolean isAddOil() {
        return true;
    }

    // 封装具体的行为：做饭
    public final void cook() {
        // 如果子类决定添加，则执行添加油的方法
        if (this.isAddOil()) {
            this.oil();
        }
        this.egg();
        this.tomato();
    }
}
```

b. 修改具体类

我：

```
package com.bjsxt.chapter03;
```

```

/**
 * 具体类
 */
public class Me extends Cook {

    private boolean isAddOilFlag = true;

    public boolean isAddOilFlag() {
        return isAddOilFlag;
    }

    public void setAddOilFlag(boolean addOilFlag) {
        isAddOilFlag = addOilFlag;
    }

    @Override
    boolean isAddOil() {
        return this.isAddOilFlag;
    }

    @Override
    void oil() {
        System.out.println("自己：油放多了！");
    }

    @Override
    void egg() {
        System.out.println("自己：鸡蛋炒糊了！");
    }

    @Override
    void tomato() {
        System.out.println("自己：西红柿放多了！");
    }
}

```

大厨：

```
package com.bjsxt.chapter03;

/**
 * 具体类
 */
public class Chef extends Cook {

    private boolean isAddOilFlag = true;

    public boolean isAddOilFlag() {
        return isAddOilFlag;
    }

    public void setAddOilFlag(boolean addOilFlag) {
        isAddOilFlag = addOilFlag;
    }

    @Override
    boolean isAddOil() {
        return this.isAddOilFlag;
    }

    @Override
    void oil() {
        System.out.println("厨师：油适量！");
    }

    @Override
    void egg() {
        System.out.println("厨师：鸡蛋适量！");
    }

    @Override
    void tomato() {
        System.out.println("厨师：西红柿适量！");
    }
}
```

```
}
```

c. 测试

```
package com.bjsxt.chapter03;

public class TestCook {

    public static void main(String[] args) {
        Me me = new Me();
        me.setAddOil(false);
        me.cook();
        new Chef().cook();
    }
}
```

自己：鸡蛋炒糊了！

自己：西红柿放多了！

厨师：油适量！

厨师：鸡蛋适量！

厨师：西红柿适量！

2.6、Thread 模拟营业大厅叫号机程序

相信很多人都去过银行、公积金中心等，在这些机构的营业大厅都有排队等号的机制，这种机制的主要作用就是限流，减轻业务受理人员的压力。当你走进营业大厅后，需要先领取一张流水号纸票，然后拿着纸票坐在休息区等待你的号码显示在业务办理的窗口显示器上。



假设大厅共有四个受理业务的柜台窗口，这就意味着有四个线程在工作，下面我们用程序模拟一下叫号的过程，约定当天最多受理 50 笔业务，也就是说号码最多可以出到 50，代码如下：

```
package com.bjsxt.chapter04;

/**
 * 柜台窗口
 */
public class CounterWindow extends Thread {

    // 窗口名称
    private final String windowName;

    // 最多受理 50 笔业务
    private static final int MAX = 50;

    // 起始号码
    private int index = 1;

    public CounterWindow(String windowName) {
        this.windowName = windowName;
    }

    @Override
    public void run() {
        while (index <= MAX) {
            System.out.format("请[%d]号到[%s]办理业务\n", index++,

```



```

windowName);
    }
}

public static void main(String[] args) {
    new CounterWindow("一号窗口").start();
    new CounterWindow("二号窗口").start();
    new CounterWindow("三号窗口").start();
    new CounterWindow("四号窗口").start();
}
}

```

上面的程序运行以后，结果似乎令人大失所望，为何每一个 CounterWindow 所出的号码都是从 1 到 50？

```

请[1]号到[三号窗口]办理业务
请[1]号到[四号窗口]办理业务
请[1]号到[一号窗口]办理业务
请[1]号到[二号窗口]办理业务
请[2]号到[三号窗口]办理业务
请[2]号到[四号窗口]办理业务
请[2]号到[二号窗口]办理业务
请[2]号到[一号窗口]办理业务
请[3]号到[四号窗口]办理业务
请[3]号到[三号窗口]办理业务
请[3]号到[一号窗口]办理业务
请[3]号到[二号窗口]办理业务
...

```

之所以出现这个问题，根本原因是因为每一个线程的逻辑执行单元都不一样，我们新建了四个 CounterWindow 线程，它们的票号都是从 1 开始到 50 结束，四个线程并没有获取一个唯一的递增的号码，那么应该如何改进呢？

其实无论 CounterWindow 被实例化多少次，只需要保证 index 是唯一的即可，我们会立即想到使用 static 去修饰 index 以达到目的，改进后的代码如下：

```

package com.bjsxt.chapter04;

/**
 * 柜台窗口
 */

```

```
public class CounterWindow02 extends Thread {

    // 窗口名称
    private final String windowName;

    // 最多受理 50 笔业务
    private static final int MAX = 50;

    // 起始号码
    private static int index = 1;

    public CounterWindow02(String windowName) {
        this.windowName = windowName;
    }

    @Override
    public void run() {
        while (index <= MAX) {
            System.out.format("请[%d]号到[%s]办理业务\n", index++,
windowName);
        }
    }

    public static void main(String[] args) {
        new CounterWindow02("一号窗口").start();
        new CounterWindow02("二号窗口").start();
        new CounterWindow02("三号窗口").start();
        new CounterWindow02("四号窗口").start();
    }
}
```

再次运行上面的 main 函数,会发现情况似乎有些改善,四个窗口交替输出不同的号码,内如如下:

```
请[1]号到[一号窗口]办理业务
请[4]号到[四号窗口]办理业务
请[3]号到[三号窗口]办理业务
请[2]号到[二号窗口]办理业务
```

请[5]号到[四号窗口]办理业务
 请[6]号到[一号窗口]办理业务
 请[7]号到[二号窗口]办理业务
 请[8]号到[三号窗口]办理业务
 请[9]号到[一号窗口]办理业务
 请[10]号到[四号窗口]办理业务
 ...

通过对 index 进行 static 修饰，做到了多线程下共享资源的唯一性，看起来似乎满足了我们的需求（事实上，如果将线程阻塞时间缩短或者将最大号码调整到 500、1000 等稍微大一些的数字就会出现线程安全问题，关于这点会在后面的章节中进行详细讲解）。

目前只有一个 index 共享资源，如果共享资源很多呢？且共享资源要经过一些比较复杂的计算呢？不可能都使用 static 修饰，而且 static 修饰的变量生命周期很长，所以 Java 提供了一个 Runnable 接口专门用于解决该问题，将线程的控制和业务逻辑的运行彻底分离开来。

2.7、Runnable 模拟营业大厅叫号机程序

2.7.1、Runnable 的职责

Runnable 接口非常简单，只定义了一个无参数无返回值的 run 方法，在 JDK8 的概念中属于一个函数式接口，源码如下：

```
package java.lang;

@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

通过官方 API 可以发现创建线程的方式有两种，第一种是通过继承 Thread 类，重写 run 方法；第二种是通过实现 Runnable 接口，创建线程时传入该接口的子类。这里需要说明一点：很多人认为创建线程的第二种方式是通过 Runnable 接口，这种说法是错误的，最起码是不严谨的，Java 中代表线程的就只有 Thread 这个类，根据我们在 start 方法源码中的分析，线程的执行单元就是 run 方法，你可以通过继承 Thread 然后重写 run 方法实现自己的业务逻辑，也可以实现 Runnable 接口实现自己的业务逻辑，源码如下：

```
@Override
public void run() {
```

```
// 如果构造 Thread 时传递了 Runnable , 则会执行 Runnable 的 run 方法
if (target != null) {
    target.run();
}
// 否则需要重写 Thread 类的 run 方法
}
```

通过源码发现, 创建线程只有一种方式那就是构造 Thread 类, 而实现线程的执行单元则有两种方式, 第一种是重写 Thread 类的 run 方法, 第二种是实现 Runnable 接口的 run 方法, 并且将 Runnable 实例用作构造 Thread 的参数。

官方为什么这么设计? Runnable 接口存在的必要是什么? 接下来我们通过案例带大家详细了解一下。

2.7.2、策略模式

无论是 Runnable 的 run 方法, 还是 Thread 类本身的 run 方法(事实上 Thread 类也是实现了 Runnable 接口)都是想将线程的控制和业务逻辑的运行分离开来, 达到职责分明、功能单一的原则, 这一点与 GoF 设计模式中的策略模式很相似, 在本节中, 我们一起学习一下什么是策略模式, 然后再来对比 Thread 和 Runnable 两者之间的区别。

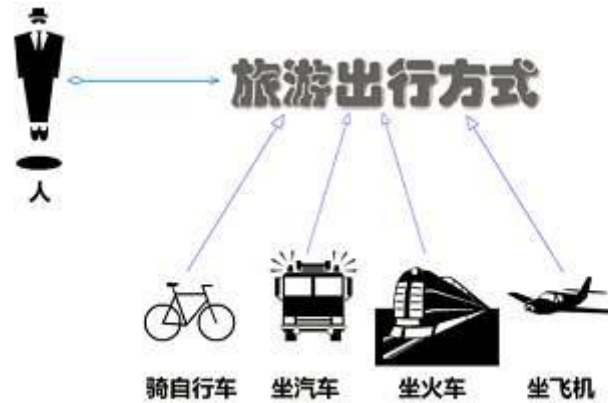
2.7.2.1、概念

在策略模式 (Strategy Pattern) 中, 一个类的行为或其算法可以在运行时更改, 我们创建表示各种策略的对象和运算规则随着策略对象的改变而改变。策略模式把对象本身和运算规则进行了分离。

这里所指的对象本身就是 Thread 线程对象, 而运算规则可以理解为 run 方法中的业务逻辑。

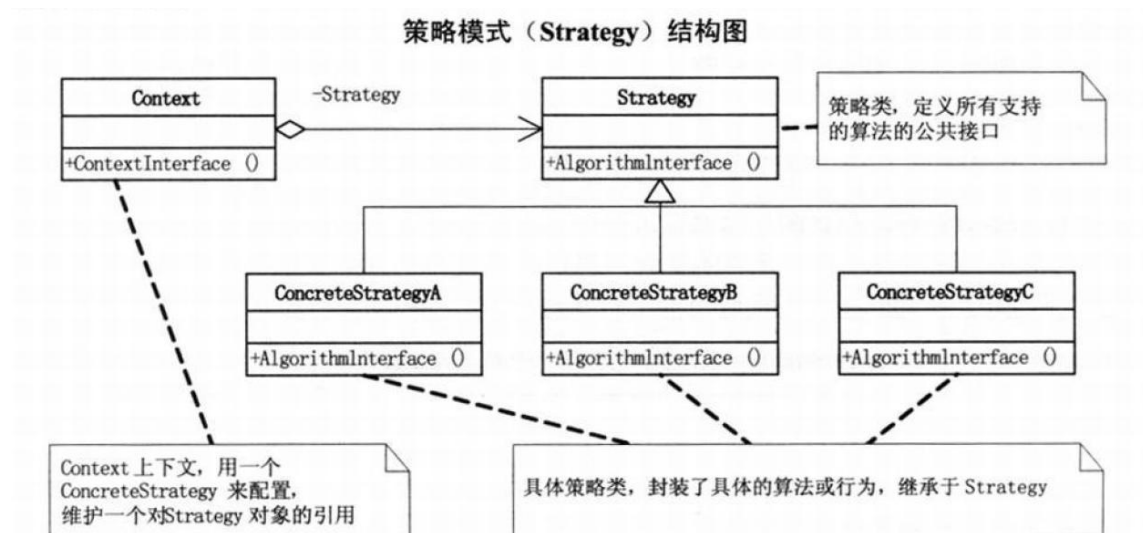
2.7.2.2、案例

为了更好的理解这个模式, 我们举例说明一下, 我们出去旅游的时候可能有很多种出行方式, 比如说我们可以坐汽车、坐火车、坐飞机等等。不管我们使用哪一种出行方式, 最终的目的地都是一样的。也就是选择不同的方式产生的结果都是一样的。



有了这个例子，我相信你应该对其思想有了一个基本的认识，下面看一下其正式的概念介绍：**定义一系列的算法，把每一个算法封装起来，并且使它们可相互替换。**

2.7.2.2.1、分析



策略模式把对象本身和运算规则进行了分离，因此我们整个模式被分为三个部分：

1. 抽象策略类(Strategy)：策略的抽象，出行方式的抽象；
2. 具体策略类(ConcreteStrategy)：具体的策略实现，每一种出行方式的具体实现；
3. 环境类(Context)：用来操作策略的上下文环境，也就是我们游客。

下面我们通过代码去实现一遍就能很清楚的理解了。

2.7.2.2.2、实现

a. 定义抽象策略接口

```
package com.bjsxt.chapter05;
```

```
/**
 * 旅行策略接口
 */
public interface travelStrategy {

    // 旅行算法
    void travelAlgorithm();

}
```

b. 具体策略类

```
package com.bjsxt.chapter05;

/**
 * 具体策略类 汽车
 */
public class CarStrategy implements travelStrategy {

    /**
     * 算法具体实现
     */
    @Override
    public void travelAlgorithm() {
        System.out.println("坐汽车...");
    }

}
```

```
package com.bjsxt.chapter05;

/**
 * 具体策略类 高铁
 */
public class HighTrainStrategy implements travelStrategy {

    /**
     * 算法具体实现
     */
}
```



```

        */
        @Override
        public void travelAlgorithm() {
            System.out.println("坐高铁...");
        }
    }

package com.bjsxt.chapter05;

/**
 * 具体策略类 飞机
 */
public class PlaneStrategy implements travelStrategy {

    /**
     * 算法具体实现
     */
    @Override
    public void travelAlgorithm() {
        System.out.println("坐飞机...");
    }
}

```

c. 环境类实现

```

package com.bjsxt.chapter05;

/**
 * 环境类 旅行者
 */
public class Traveler {

    // 维护策略接口对象的一个引用
    private travelStrategy travelStrategy;

    // 通过 set 方法设置旅行策略
    public void setTravelStrategy(com.bjsxt.chapter05.travelStrategy

```

```
travelStrategy) {
    this.travelStrategy = travelStrategy;
}

// 调用对应策略的实现
public void travelStyle() {
    travelStrategy.travelAlgorithm();
}

public static void main(String[] args) {
    Traveler traveler = new Traveler();
    // 设置旅行策略
    traveler.setTravelStrategy(new CarStrategy());
    // traveler.setTravelStrategy(new HighTrainStrategy());
    // traveler.setTravelStrategy(new PlaneStrategy());
    traveler.travelStyle();
}
}
```

通过以上案例我们可以清晰感受到策略模式带来的好处，下面我们来总结一下：

优点：

- 我们之前在选择出行方式的时候，往往会使用 if-else 语句，也就是用户不选择 A 那么就选择 B 这样的一种情况。这种情况耦合性太高了，而且代码臃肿，有了策略模式我们就可以避免这种现象；
- 策略模式遵循开闭原则，实现代码的解耦合。扩展新的方法时也比较方便，只需要继承策略接口就好了。

缺点：

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类；
- 策略模式会出现很多的策略类；
- 客户端在使用这些策略类的时候，这些策略类由于继承了策略接口，所以有些数据可能用不到，但是依然初始化了。

2.7.2.3、拓展

重写 Thread 类的 run 方法和实现 Runnable 接口的 run 方法还有一个很重要的不同那就是 Thread 类的 run 方法是不能和共享的，也就是说 A 线程不能把 B 线程的 run 方法当作自己的执行单元，而使用 Runnable 接口则很容易就能实现这一点，使用同一个

Runnable 的实例构造不同的 Thread 实例。

2.7.3、模拟营业大厅叫号机程序

既然我们说使用 static 修饰 index 这个共享资源不是一种好的方式，那么我们在本节中使用 Runnable 接口来实现逻辑执行单元重构之前营业大厅叫号机程序。

首先我们将 Thread 的 run 方法抽取成一个 Runnable 接口的实现，代码如下：

```
package com.bjsxt.chapter06;

/**
 * 具体策略类 叫号机
 */
public class CounterWindowRunnable implements Runnable {

    // 最多受理 50 笔业务
    private static final int MAX = 50;

    // 起始号码，不做 static 修饰
    private int index = 1;

    @Override
    public void run() {
        while (index <= MAX) {
            System.out.format("请[%d]号到[%s]办理业务\n", index++,
Thread.currentThread().getName());
        }
    }

    public static void main(String[] args) {
        final CounterWindowRunnable task = new CounterWindowRunnable();
        new Thread(task, "一号窗口").start();
        new Thread(task, "二号窗口").start();
        new Thread(task, "三号窗口").start();
        new Thread(task, "四号窗口").start();
    }
}
```

可以看到上面的代码中并没有对 index 进行 static 的修饰，并且我们也将 Thread 中 run 的代码逻辑抽取到了 Runnable 的一个实现中。

程序的输出结果虽然和之前是一样的，但是这次是使用了同一个 Runnable 接口，这样他们的资源就是共享的，不会再出现每一个叫号机都从 1 打印到 50 这样的情况。

注意：不管是之前 static 修饰 index 的方式，还是用实现 Runnable 接口的方式，这两个程序多运行几次或者将线程阻塞时间缩短或者 MAX 的值从 50 增加到 500、1000 等稍微大一些的数字就会出现一个号码出现两次的情况，也会出现某个号码根本不会出现的情况，更会出现超过最大值的情况，这是因为共享资源 index 存在线程安全的问题，我们后面学习数据同步的时候会详细介绍。

2.8、总结

本章我们学习了什么是线程，以及初步掌握了如何创建一个线程，并且通过重写 Thread 的 run 方法和实现 Runnable 接口的 run 方法进而实现线程的执行单元。

通过两个版本的叫号机程序，让大家对多线程的程序有了一个初步的认识，模板设计模式以及策略设计模式和 Thread 以及 Runnable 的结合使得大家能够更加清晰地掌握多线程的 API 是如何实现线程控制和业务执行解耦分离的。

本章中最为重要的内容就是线程的生命周期，在使用多线程的过程中，线程的生命周期将会贯穿始终，只有清晰的掌握生命周期各个阶段的切换，才能更好的理解线程的阻塞以及唤醒机制，同时也为掌握同步锁等概念打下一个良好的基础。

三、深入理解 Thread 构造器

在 Java 中的 Thread 为我们提供了比较丰富的构造函数，在本章中，我们将会逐一介绍每一个构造函数，以及分析其中一些可能并未引起你关注的细节。Thread 构造函数如下图：

Constructors	
Constructor	Description
<code>Thread()</code>	Allocates a new Thread object.
<code>Thread(Runnable target)</code>	Allocates a new Thread object.
<code>Thread(Runnable target, String name)</code>	Allocates a new Thread object.
<code>Thread(String name)</code>	Allocates a new Thread object.
<code>Thread(ThreadGroup group, Runnable target)</code>	Allocates a new Thread object.
<code>Thread(ThreadGroup group, Runnable target, String name)</code>	Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
<code>Thread(ThreadGroup group, Runnable target, String name, long stackSize)</code>	Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size.
<code>Thread(ThreadGroup group, Runnable target, String name, long stackSize, boolean inheritThreadLocals)</code>	Allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, has the specified stackSize, and inherits initial values for inheritable thread-local variables if inheritThreadLocals is true.
<code>Thread(ThreadGroup group, String name)</code>	Allocates a new Thread object.

3.1、线程的命名

我们构造线程的时候可以为线程起一个有特殊意义的名字，这也是比较好的一种做法，尤其在线程比较多的程序中，为线程赋予一个包含特殊意义的名字有助于问题的排查和线程的跟踪，根据这个特性，我们可以将构造器分别两种，一种是构建时可以指定线程名称的，一种是使用线程默认名称的。

3.1.1、线程的默认命名

下面的几个构造函数，并没有提供为线程命名的参数，那么此时线程会有一个怎样的命名呢？

```
Thread()  
Thread (Runnable target)  
Thread (ThreadGroup group, Runnable target)
```

打开 JDK 的源码会看到下面的代码：

```
/**  
 * Allocates a new {@code Thread} object. This constructor has the same  
 * effect as {@link Plain #Thread(ThreadGroup,Runnable,String) Thread}  
 * {@code (null, null, gname)}, where {@code gname} is a newly generated  
 * name. Automatically generated names are of the form  
 * {@code "Thread-"}+<i>n</i>, where <i>n</i> is an integer.  
 */  
public Thread() {  
    this(null, null, "Thread-" + nextThreadNum(), 0);  
}  
  
/* For autonumbering anonymous threads. */  
private static int threadInitNumber;  
private static synchronized int nextThreadNum() {  
    return threadInitNumber++;  
}
```

如果没有为线程显示的指定一个名字，那么线程将会以“Thread-”作为前缀与一个自增数字进行组合，这个自增数字在整个 JVM 进程中将会不断自增，且加了 synchronized 不会出现重复的情况：

```
public static void main(String[] args) {  
    for (int i = 0; i < 5; i++) {  
        new Thread(() ->  
            System.out.println(Thread.currentThread().getName()).start();  
    }  
}
```

执行上面的代码，我们通过无参构造函数创建了 5 个线程，并且分别输出了各自的名字，发现输出结果与我们对源码的分析时一致的，结果如下：

```
Thread-0
```

```
Thread-3
Thread-2
Thread-1
Thread-4
```

3.1.2、命名线程

在线程比较多的程序中,为线程赋予一个包含特殊意义的名字有助于问题的排查和线程的跟踪, Thread 同样提供了对应的构造函数, 具体如下:

```
Thread (String name)
Thread (Runnable target, String name)
Thread (ThreadGroup group, Runnable target, String name)
Thread (ThreadGroup group, Runnable target, String name, long stackSize)
Thread (ThreadGroup group, Runnable target, String name, long stackSize,
boolean inheritThreadLocals)
Thread (ThreadGroup group, String name)
```

实例代码如下:

```
public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        new Thread(() ->
System.out.println(Thread.currentThread().getName()),
                "MyThread-" + i).start();
    }
}
```

上面的代码中,我们定义了一个新的前缀 "MyThread-" ,然后用 0~4 之间的数字作为后缀对线程进行了命名,代码执行输出的结果如下:

```
MyThread-0
MyThread-3
MyThread-1
MyThread-2
MyThread-4
```

3.1.3、修改线程的名字

修改线程的名称可以借助 setName 方法进行修改,代码如下:

```
new Thread(() -> {
    int index = 0;
    while (index < 1) {
        System.out.println(Thread.currentThread().getName());
        Thread.currentThread().setName("AfterThread");
        System.out.println(Thread.currentThread().getName());
        index++;
    }
})
```



```
}, "BeforeThread").start();
```

运行结果如下：

```
BeforeThread
AfterThread
```

3.2、线程的父子关系

Thread 的所有构造函数，最终都会去调一个私有构造器，我们截取片段代码对其进行分析，不难发现新创建的任何一个线程的都会有一个父线程：

```
private Thread(ThreadGroup g, Runnable target, String name,
               long stackSize, AccessControlContext acc,
               boolean inheritThreadLocals) {
    if (name == null) {
        throw new NullPointerException("name cannot be null");
    }

    this.name = name;

    Thread parent = currentThread(); // 获取当前线程作为父线程
```

上面代码中的 `currentThread()` 是获取当前线程，在线程生命周期中，我们说过线程的最初状态为 NEW，没有执行 start 方法之前，它只能算是一个 Thread 的实例，并不意味着一个新的线程被创建，因此 `currentThread()` 代表的将会是创建它的那个线程，因此我们可以得出以下结论：

- 一个线程的创建肯定是由另一个线程完成的；
- 被创建的线程属于创建它的线程的子线程。

我们都知道 main 函数所在的线程是由 JVM 创建的，也就是 main 线程，那就意味着我们前面创建的所有线程，其父线程都是 main 线程。

3.3、Thread 与 ThreadGroup

在 Thread 的构造函数中，可以显示的指定线程的 Group，也就是 ThreadGroup（关于 ThreadGroup 会在后面的章节中重点介绍）。

继续阅读私有构造器的源码：

```
SecurityManager security = System.getSecurityManager();
if (g == null) {
    /* Determine if it's an applet or not */

    /* If there is a security manager, ask the security manager
       what to do. */
    if (security != null) {
        g = security.getThreadGroup();
    }
}
```

```

/* If the security manager doesn't have a strong opinion
   on the matter, use the parent thread group. */
if (g == null) {
    g = parent.getThreadGroup();
}
}

```

通过对源码进行分析，我们可以看出，如果在构造 Thread 的时候没有显示的指定一个 ThreadGroup，那么子线程将会被加入父线程所在的线程组，下面写一个简单的代码来测试一下。

```

package com.bjsxt.chapter03.demo02;

public class ThreadConstructors02 {

    public static void main(String[] args) {
        // 创建线程对象 t1
        Thread t1 = new Thread("t1");
        // 创建线程组对象
        ThreadGroup testGroup = new ThreadGroup("TestGroup");
        // 创建线程对象 t2，并且将它加入线程组对象 testGroup 中
        Thread t2 = new Thread(testGroup, "t2");
        // 当前运行线程 main 线程的线程组
        ThreadGroup mainThreadGroup =
            Thread.currentThread().getThreadGroup();

        // main 线程所属线程组的名称
        System.out.println("main thread belong thread group: " +
            mainThreadGroup.getName());
        // t1 线程和 main 线程是否属于相同的线程组
        System.out.println("do t1 and main belong the same thread group:
            "
            + (t1.getThreadGroup() == mainThreadGroup));
        // t2 线程是否属于 mainThreadGroup
        System.out.println("do t2 belong the mainThreadGroup: "
            + (t2.getThreadGroup() == mainThreadGroup));
        // t2 线程是否属于 testGroup
        System.out.println("do t2 belong the testGroup: "
            + (t2.getThreadGroup() == testGroup));
    }
}

```

通过对 Thread 源码的分析和我们自己的测试可以得出以下结论：

- main 线程所在的 ThreadGroup 名称为 main
- 构造一个线程的时候如果没有显示的指定 ThreadGroup，那么它将会和父线程同属于一个 ThreadGroup。

在默认设置中，当然除了子线程会和父线程同属于一个 Group 之外，它还会和父线程拥有同样的优先级，同样的 daemon，关于这点我们在后文中将会详细讲解。

3.4、Thread 与 JVM 虚拟机栈

在 Thread 的构造函数中,有一个特殊的参数 `stackSize`。这个参数的作用是什么呢?它的值对线程有什么影响呢?下面我们就来一起探讨这个问题。

3.4.1、Thread 与 StackSize

打开 JDK 官方文档,将会发现 Thread 中对 `stackSize` 构造函数的文字说明,具体如下:

```
public Thread (ThreadGroup group,  
               Runnable target,  
               String name,  
               long stackSize)
```

Allocates a new *Thread* object so that it has *target* as its run object, has the specified *name* as its name, and belongs to the thread group referred to by *group*, and has the specified *stack size*.

This constructor is identical to `Thread(ThreadGroup,Runnable,String)` with the exception of the fact that it allows the thread stack size to be specified. The stack size is the approximate number of bytes of address space that the virtual machine is to allocate for this thread's stack. **The effect of the `stackSize` parameter, if any, is highly platform dependent.** (`stackSize` 参数的效果(如果有的话)与平台高度相关。)

On some platforms, specifying a higher value for the `stackSize` parameter may allow a thread to achieve greater recursion depth before throwing a `StackOverflowError`. Similarly, specifying a lower value may allow a greater number of threads to exist concurrently without throwing an `OutOfMemoryError` (or other internal error). The details of the relationship between the value of the `stackSize` parameter and the maximum recursion depth and concurrency level are platform-dependent. **On some platforms, the value of the `stackSize` parameter may have no effect whatsoever.** (在某些平台上, `stackSize` 参数的值可能没有任何影响。)

一般情况下,创建线程的时候不会手动指定栈内存的地址空间字节数组,统一通过 `xss` 参数进行设置即可,通过上面这段官网文档的描述,我们不难发现 `stackSize` 越大则代表着正在线程内方法调用递归的深度就越深, `stackSize` 越小则代表着创建的线程数量越多,当然了,这个参数对平台的依赖性比较高,比如不同的操作系统、不同的硬件。

在有些平台下,越高的 `stack` 设定,可以允许的递归深度越深;反之,越少的 `stack` 设定,则递归深度越浅。当然在某些平台下,该参数压根不会起到任何作用,如果将该参数

设置为 0，也不会起到任何的作用。

下面我们在 Windows 电脑上进行一波测试，看看 stackSize 给线程带来的影响。

```
package com.bjsxt.chapter03.demo03;

public class ThreadConstructors05 {

    private static int counter = 0;

    public static void main(String[] args) {
        new Thread(null, new Runnable() {
            @Override
            public void run() {
                try {
                    add(0);
                } catch (Error e) {
                    e.printStackTrace();
                    System.out.println(counter);
                }
            }

            private void add(int i) {
                counter++;
                add(i + 1);
            }
        }, "Test", 10000000L).start();
    }
}
```

通过以上代码，我们在线程中设定了一个简单的递归，就是不断调用自己，然后输出 int 值。

由于不断的进行压栈弹栈操作，整个栈内存肯定会被压爆，也就是说最后都会抛出 java.lang.StackOverflowError 异常，在创建 Thread 时传入的 stackSize 对递归深度的影响具体如下：

stackSize	1	10	100	1000	10000	100000	1000000	100000000
Windows	18866	19068	26157	31537	164	786	26957	581045

Windows 的测试数据有些不是特别合理，尤其 10000 和 100000 让人摸不着头脑，不过大方向还是正确的，随着 stackSize 数量级的不断增加，递归的深度也變得越来越大，该参数一般情况下不会主动设置，采用系统默认值就可以了，默认情况下会设置为 0。

3.4.2、JVM 内存结构

3.4.2.1、JVM 运行时数据区域概览



这里介绍的是 JDK1.8 JVM 运行时内存数据区域划分。1.8 同 1.7 比，最大的差别就是：元数据区取代了永久代。元空间的本质和永久代类似，都是对 JVM 规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元数据空间并不在虚拟机中，而是使用本地内存。

3.4.2.2、各区域介绍

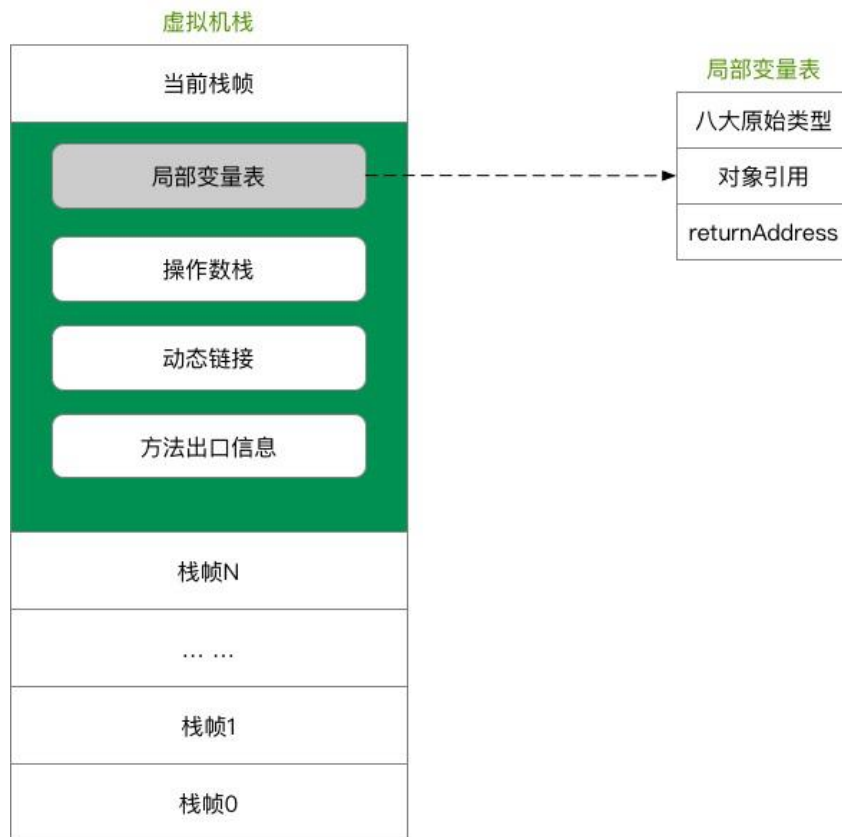
程序计数器

线程私有 指向当前线程正在执行的字节码代码的行号。如果当前线程执行的是 native 方法，则其值为 null。

Java 虚拟机栈 stack

线程私有，每个线程对应一个 Java 虚拟机栈，其生命周期与线程同进同退。每个 Java

方法在被调用的时候都会创建一个栈帧，并入栈。一旦完成调用，则出栈。所有的的栈帧都出栈后，线程也就完成了使命。



本地方法栈

功能与 Java 虚拟机栈十分相同。区别在于，本地方法栈为虚拟机使用到的 native 方法服务。

堆 heap

堆是 JVM 内存占用最大，管理最复杂的一个区域。其唯一的用途就是 存放对象实例：几乎所有的对象实例及数组都在堆上进行分配。JDK1.7 及其以后，字符串常量池从永久代中剥离出来，存放在堆中。堆有自己进一步的内存分块划分，按照 GC 分代收集角度的划分请参见上图。

字符串常量池

JDK1.7 就开始“去永久代”的工作了。 1.7 把字符串常量池从永久代中剥离出来，存放在堆空间中。

元数据区

元数据区取代了 1.7 版本 及以前的永久代。元数据区和永久代本质上都是方法区的实现。方法区存放虚拟机加载的类信息，静态变量，常量等数据。

直接内存

jdk1.4 引入了 NIO，它可以使用 Native 函数库 直接分配堆外内存。

3.5、守护线程

守护线程是一种比较特殊的线程，一般用于处理一些后台的工作，比如 JDK 的垃圾回收线程，什么是守护线程？为什么要有守护线程，以及何时需要守护线程？本节我们就来一起探讨一下。

要回答关于守护线程的问题，就必须先搞清除另外一个特别重要的问题：JVM 程序在什么情况下会退出？

The Java Virtual Machine exits when the only threads running are all daemon threads.

通过上面这句话的描述可以得知，如果 JVM 中有非守护线程存在，则 JVM 的进程不会退出。

3.5.1、什么是守护线程

我们先通过一个简单的程序，来认识一下守护线程的特点：

```
package com.bjsxt.chapter03.demo04;

public class DaemonThread {

    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        // 创建子线程
        Thread t1 = new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(1000L);
                    System.out.println("子线程...");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        // t1.setDaemon(true); // 是否设置为守护线程
    }
}
```

```
t1.start(); // 启动子线程
Thread.sleep(2000L);
// main 父线程结束
System.out.println("main thread finished lifecycle.");
}
}
```

上面的代码存在两个线程，一个是由 JVM 启动的 main 线程，另外一个则是我们自己创建的线程 Thread，运行上面的这段代码，你会发现 JVM 进程永远不会退出，即使 main 线程正常结束了自己的生命周期，原因就是因为在 JVM 进程中还存在一个非守护线程在运行。

如果打开是否设置为守护线程的注释，将自己创建的 Thread 设置为了守护线程，那么 main 进程结束生命周期后，JVM 也会随之退出运行，当然 Thread 线程也会结束。

3.5.2、守护线程的作用

在了解了什么是守护线程，以及如何创建守护线程之后，我们来讨论一下为什么要有守护线程以及何时使用守护线程。

通过上面的分析，如果一个 JVM 进程中没有一个非守护线程，那么 JVM 会退出，也就是说守护线程具备自动结束生命周期的特性，而非守护线程则不具备这个特点，试想一下如果 JVM 进程的垃圾回收线程是非守护线程，如果 main 线程完成了工作，则 JVM 无法退出，因为垃圾回收线程还在正常的工作。再比如有一个简单的游戏程序，其中有一个线程正在与服务器不断交互以获取玩家最新的信息，若希望在退出游戏客户端的时候，这些数据同步的工作也能够立即结束，等等。

守护线程经常用作与执行一些后台任务，因此有时它也被称为后台线程，当你希望关闭某些线程的时候，或者退出 JVM 进程的时候，一些线程能够自动关闭，此时就可以考虑用守护线程为你完成这样的工作。

3.6、总结

在本章节中，我们非常详细的讲解了 Thread 的构造函数，并且挖掘了其中很多细节，尤其是 stackSize 对 Thread 的影响。

除此之外，本章也介绍了线程的父子关系，默认情况下线程从父线程那里是否继承了守护线程、优先级、ThreadGroup 等特性。

最后，本章还分析了什么是守护线程，以及守护线程的特性以及其应该使用在何种场景之下。

四、ThreadAPI 的详细介绍

第 2 章和第 3 章主要是从概念上了解 Thread，本章中，我们将细致的学习 Thread 所有 API 的作用以及用法。

4.1、线程 sleep

sleep 是一个静态方法，其有两个重载方法，其中一个需要传入毫秒数，另外一个既需要毫秒数也需要纳秒数。

4.1.1、sleep 方法介绍

```
public static void sleep (long millis) throws InterruptedException
```

```
public static void sleep (long millis, int nanos) throws InterruptedException
```

sleep 方法会使当前线程进入指定毫秒数的休眠，暂停执行，虽然给定了一个休眠的时间，但是最终要以系统的定时器和调度器的精度为准，休眠有一个非常重要的特性，那就是其不会放弃 monitor 锁的所有权（线程同步和锁的时候会重点介绍 monitor），下面我们来看一个简单的例子。

```
package com.bjsxt.chapter04.demo01;

public class ThreadSleep {

    public static void main(String[] args) {
        // 子线程
        new Thread(() -> {
            long startTime = System.currentTimeMillis();
            sleep(2000L);
            long endTime = System.currentTimeMillis();
            System.out.printf("Total spend %d ms\n", endTime - startTime);
        }).start();

        // main 线程
        long startTime = System.currentTimeMillis();
        sleep(3000L);
        long endTime = System.currentTimeMillis();
        System.out.printf("Main thread spend %d ms", endTime - startTime);
    }

    private static void sleep(long ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
        }
    }
}
```

```
e.printStackTrace();
    }
}
```

在上面的例子中，我们分别在自定义的线程和主线程中进行了休眠，每个线程的休眠互不影响，从结果看，Thread.sleep 只会导致当前线程进入指定时间的休眠。

4.1.2、使用 TimeUnit 替代 Thread.sleep

在 JDK1.5 以后，JDK 引入了一个枚举 TimeUnit，其对 sleep 方法提供了很好的封装，使用它可以省去时间单位的换算步骤，比如线程想休眠 3 小时 24 分 17 秒 88 毫秒，使用 TimeUnit 来实现就非常简便优雅了：

```
Thread.sleep(12257088L);

TimeUnit.HOURS.sleep(3);
TimeUnit.MINUTES.sleep(24);
TimeUnit.SECONDS.sleep(17);
TimeUnit.MILLISECONDS.sleep(88);
```

同样的时间表达，TimeUnit 显然清晰很多，强烈建议在使用 Thread.sleep 的地方，完全使用 TimeUnit 来代替，因为 sleep 能做的事情，TimeUnit 全部都能完成，并且可以做的更好，后面内容中，我将全部采用 TimeUnit 替代 sleep。

4.1.3、Thread.sleep(0)

Thread.sleep(0) 表示挂起 0 毫秒，你可能觉得没作用。其实 Thread.sleep(0) 并非是真的要线程挂起 0 毫秒，意义在于这次调用 Thread.sleep(0) 的当前线程确实的被冻结了一下，让其他线程有机会优先执行。Thread.sleep(0) 是你的线程暂时放弃 cpu，也就是释放一些未用的时间片给其他线程或进程使用，就相当于一个让位动作。

在线程中，调用 sleep(0) 可以释放 cpu 时间，让线程马上重新回到就绪队列而非等待队列，sleep(0) 释放当前线程所剩余的时间片（如果有剩余的话），这样可以让操作系统切换其他线程来执行，提升效率。

4.2、线程 yield

4.2.1、yield 方法介绍

yield 方法属于一种启发式的方法，其会提醒调度器我愿意放弃当前的 CPU 资源，如果 CPU 的资源不紧张，则会忽略这种提醒。

调用 yield 方法会使当前线程从 Running 状态切换到 Runnable 状态，一般这个方法不太常用，接下来我们看一个案例：

```
package com.bjsxt.chapter04.demo02;

import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;

public class ThreadYield {

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 100; i++) {
            IntStream.range(0,
2).mapToObj(ThreadYield::create).forEach(Thread::start);
            System.out.println("-----华丽分割线-----");
            TimeUnit.MILLISECONDS.sleep(500L);
        }
    }

    private static Thread create(int index) {
        return new Thread(() -> {
            if (index == 0)
                Thread.yield();
            System.out.println(index);
        });
    }
}
```

上面的程序运行多次，你会发现输出结果不一致，有时候是 0 最先打印出来，有时候是 1 最先打印出来。根据代码运行结果分析如下：

第一个线程如果最先获得了 CPU 资源，它会比较谦虚，主动告诉 CPU 调度器试放原本属于自己的资源，但是 yield 只是一个提示（hint），CPU 调度器并不会担保每次都能满足 yield 提示。

4.2.2、yield 和 sleep

看过前面的内容之后，会发现 yield 和 sleep 有一些混淆的地方，在 JDK1.5 以前的

版本中 `yield` 的方法事实上是调用了 `sleep(0)`，但是他们之间存在着本质的区别，具体如下：

- `sleep` 会导致当前线程暂停指定的时间，没有 CPU 时间片的消耗；
- `yield` 只是对 CPU 调度器的一个提示，如果 CPU 调度器没有忽略这个提示，它会导致线程上下文的切换；
- `sleep` 会使线程短暂 block，会在给定的时间内试放 CPU 资源；
- `yield` 会使 Running 状态的 Thread 进入 Runnable 状态（如果 CPU 调度器没有忽略这个提示的话）；
- `sleep` 几乎百分之百的完成了给定时间的休眠，而 `yield` 的提示并不能一定保证。

4.3、设置线程优先级

```
public final void setPriority (int newPriority)
public final int getPriority()
```

4.3.1、线程优先级介绍

进程有进程的优先级，线程同样也有优先级，理论上是优先级比较高的线程会获取优先被 CPU 调度的机会，但是事实上往往并不会如你所愿，设置线程的优先级同样也是一个 hint 操作，具体如下。

- 对于 root 用户，他会 hint 操作系统你想要设置的优先级别，否则它会被忽略。
- 如果 CPU 比较忙，设置优先级可能会获得更多的 CPU 时间片，但是闲时优先级的高低几乎不会有任何作用。

所以，不要在程序设计当中企图使用线程优先级绑定某些特定的业务，或者让业务严重依赖于线程优先级，这可能会让你大失所望。

举个简单的例子，可能不同情况下的与运行效果不会完全一样，但是我们只是想让优先级比较高的线程获得更多的信息输出机会，代码如下：

```
package com.bjsxt.chapter04.demo03;

public class ThreadPriority {

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            while (true) {
                System.out.println("t1");
            }
        });
        t1.setPriority(3);

        Thread t2 = new Thread(() -> {
```



```

        while (true) {
            System.out.println("t2");
        }
    });
    t2.setPriority(10);

    t1.start();
    t2.start();
}
}

```

运行上面的程序，会发现 t2 出现的频率很明显要高一些，当然这也和笔者当前 CPU 的资源情况有关系，不同情况下的运行会有不一样的结果。

4.3.2、线程优先级源码分析

设置线程的优先级，只需要调用 setPriority 方法即可，下面我们打开 Thread 源码，一起来分析一下：

```

public final void setPriority(int newPriority) {
    ThreadGroup g;
    checkAccess();
    if (newPriority > MAX_PRIORITY || newPriority < MIN_PRIORITY) {
        throw new IllegalArgumentException();
    }
    if ((g = getThreadGroup()) != null) {
        if (newPriority > g.getMaxPriority()) {
            newPriority = g.getMaxPriority();
        }
        setPriority0(priority = newPriority);
    }
}

```

通过上面源码的分析，我们可以看出，线程的优先级不能小于 1 也不能大于 10，如果指定的线程优先级大于线程所在 group 的优先级，那么指定的优先级将会失败，取而代之的是 group 的最大优先级，下面我们通过一个例子来证明一下：

```

package com.bjsxt.chapter04.demo03;

public class ThreadPriority02 {

    public static void main(String[] args) {
        // 定义一个线程组
        ThreadGroup testGroup = new ThreadGroup("testGroup");
        // 将线程组的优先级指定为 7
        testGroup.setMaxPriority(7);
        // 定义一个线程，将该线程加入到 group 中
        Thread testThread = new Thread(testGroup, "testThread");
        // 企图将线程的优先级设定为 10
        testThread.setPriority(10);
        // 企图未遂
    }
}

```

```
        System.out.println(testThread.getPriority());
    }
}
```

上面的结果输出为 7，而不是 10。因为它超过了所在线程组的优先级。

4.3.3、关于优先级的一些总结

一般情况下，不会对线程设定优先级别，更不会对某些业务严重的依赖线程的优先级别，比如权重，借助优先级设定某个任务的权重，这种方式是不可取的，一般定义线程的时候使用默认的优先级就好了，那么线程默认的优先级是多少呢？

线程默认的优先级和它的父类保持一致，一般情况下都是 5，因为 main 线程的优先级就是 5，所以它派生出来的线程都是 5，代码如下：

```
package com.bjsxt.chapter04.demo03;

public class ThreadPriority03 {

    public static void main(String[] args) {
        Thread t1 = new Thread();
        System.out.println("t1 priority " + t1.getPriority()); // 5

        Thread t2 = new Thread() -> {
            Thread t3 = new Thread();
            System.out.println("t3 priority " + t3.getPriority()); // 6
        };

        t2.setPriority(6);
        t2.start();
        System.out.println("t2 priority " + t2.getPriority()); // 6
    }
}
```

上面的程序的输出结果是 t1 的优先级为 5，因为 main 线程的优先级是 5；t2 的优先级是 6，因为显示的将其指定为 6；t3 的优先级为 6，没有显示值当，因此其父线程保持一致。

4.4、获取线程 ID

`public long getId()` 获取线程的唯一 ID，线程的 ID 在整个 JVM 进程中都是唯一的，并且是从 0 开始逐次递增。如果你在 main 线程（main 函数）中创建了一个唯一的线程，并且调用 `getId()` 后发现其并不等于 0，也许你会纳闷，不应该是从 0 开始的吗？之前已经说过了在一个 JVM 进程启动的时候，实际上是开辟了很多个线程，自增序列已经有了一定的消耗，因此我们自己创建的线程绝非第 0 号线程。

4.5、获取当前线程

`public static Thread currentThread()` 用于返回当前执行线程的引用，这个方法虽然很简单，但是使用非常广泛，我们在后面的内容中会大量的使用该方法，来看一段示例代码：

```
package com.bjsxt.chapter04.demo05;

public class CurrentThread {

    public static void main(String[] args) {
        Thread t1 = new Thread(){
            @Override
            public void run() {
                System.out.println(Thread.currentThread() == this);
            }
        };
        t1.start();

        String name = Thread.currentThread().getName();
        System.out.println("main".equals(name));
    }
}
```

上面程序运行输出的两个结果都是 true。

4.6、线程 interrupt

线程 interrupt，是一个非常重要的 API，也是经常使用的方法，与线程中断相关，相关的 API 有以下几个，在本节中我们也将 Thread 深入源码对其进行详细的剖析。

- `public void interrupt()`
- `public static boolean interrupted()`
- `public boolean isInterrupted()`

4.6.1、interrupt

以下方法的调用会使得当前线程进入阻塞状态，而调用当前线程的 interrupt 方法，就可以打断阻塞。

- Object 的 wait 方法；
- Object 的 wait(long)方法；

- Object 的 wait(long, int)方法；
- Object 的 sleep(long)方法；
- Thread 的 sleep(long)方法；
- Thread 的 join 方法；
- Thread 的 join(long)方法；
- Thread 的 join(long, int)方法；
- InterruptibleChannel 的 io 操作；
- Selector 的 wakeup 方法。

上述若干方法都会使得当前线程进入阻塞状态，若另外的一个线程调用被阻塞线程的 interrupt 方法，则会打断这种阻塞，因此这种方法有时会被称为可中断方法，记住，打断一个线程并不等于该线程的生命周期结束，仅仅是打断了当前线程的阻塞状态。

一旦线程在阻塞的情况下被打断，都会抛出一个称为 InterruptedException 的异常，这个异常就像一个 signal(信号)一样通知当前线程被打断了，下面我们来看一个例子：

```
package com.bjsxt.chapter04.demo06;

import java.util.concurrent.TimeUnit;

public class ThreadInterrupt {

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            try {
                TimeUnit.MINUTES.sleep(1);
            } catch (InterruptedException e) {
                System.out.println("阻塞状态被中断");
                e.printStackTrace();
            }
        });
        t1.start();

        // 短暂的阻塞是为了保证 t1 线程已启动
        TimeUnit.MILLISECONDS.sleep(100);
        // 中断 t1 线程的阻塞状态
        t1.interrupt();
    }
}
```

上面的代码创建了一个线程，并且企图休眠 1 分钟的时长，不过很可惜，大约在 100 毫秒之后就被主线调用 interrupt 方法打断，程序的执行结果就是“阻塞状态被中断”。

interrupt 这个方法到底做了什么样的事情呢？在一个线程内部存在着名为 interrupt flag 的标识，如果一个线程被 interrupt，那么它的 flag 将被设置，但是如果当前线程正在执行可中断方法被阻塞时，调用 interrupt 方法将其中断，反而会导致 flag 被清除，关于这点我们在后面还会做详细的介绍。另外有一点需要注意的是，如果一个线程已经是死亡状态，那么尝试对其的 interrupt 会直接被忽略。

4.6.2、isInterrupted

isInterrupted 是 Thread 的一个成员方法，它主要判断当前线程是否被中断，该方法仅仅是对 interrupt 标识的一个判断，并不会影响标识发生任何改变，这个与我们即将学习到的 interrupted 是存在差别的，下面我们看一个简单的程序：

```
package com.bjsxt.chapter04.demo06;

import java.util.concurrent.TimeUnit;

public class ThreadInterrupt02 {

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread() {
            @Override
            public void run() {
                while (true) {
                }
            }
        };
        t1.setDaemon(true);
        t1.start();

        TimeUnit.MILLISECONDS.sleep(100);
        System.out.printf("Thread is interrupted ? %s\n",
t1.isInterrupted());
        t1.interrupt();
        System.out.printf("Thread is interrupted ? %s\n",
t1.isInterrupted());
    }
}
```

上面的代码中定义了一个线程，并且在线程的执行单元中（run 方法）写了一个空的死循环，为什么不写 sleep 呢？因为 sleep 是可中断方法，会捕获到中断信号，从而干扰我们程序的结果。下面是程序运行的结果，记得手动结束上面的程序运行，或者你也可以将上面定义的线程指定为守护线程，这样就会随着主线程的结束导致 JVM 中没有非守护线程而自动退出。运行结果如下：

```
Thread is interrupted ? false
Thread is interrupted ? true
```

可中断方法捕获到了中断信号（signal）之后，也就是捕获了 InterruptedException 异常之后会擦除掉 interrupt 的标识，对上面的程序稍作修改，你会发现程序的结果又会出现很大的不同，示例代码如下：

```
package com.bjsxt.chapter04.demo06;

import java.util.concurrent.TimeUnit;

public class ThreadInterrupt03 {
```

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread() {
        @Override
        public void run() {
            while (true) {
                try {
                    TimeUnit.MINUTES.sleep(1);
                } catch (InterruptedException e) {
                    System.out.printf("I am be interrupted? %s\n",
isInterrupted());
                    e.printStackTrace();
                }
            }
        }
    };
    t1.setDaemon(true);
    t1.start();

    TimeUnit.MILLISECONDS.sleep(100);
    System.out.printf("Thread is interrupted? %s\n",
t1.isInterrupted());
    t1.interrupt();
    TimeUnit.MILLISECONDS.sleep(100);
    System.out.printf("Thread is interrupted? %s\n",
t1.isInterrupted());
}
}
```

由于在 run 方法中使用了 sleep 这个可中断方法，它会捕获到中断信号，并且会擦除 interrupt 标识，因此程序的执行结果都会是 false，程序输出如下：

```
Thread is interrupted? false
I am be interrupted? false
Thread is interrupted? false
```

其实这也不难理解，可中断方法捕获到了中断信号之后，为了不影响线程中其他方法的执行，将线程的 interrupt 标识复位是一种很合理的设计。

4.6.3、interrupted

interrupted 是一个静态方法，虽然其也用于判断当前线程是否被中断，但是它和成员方法 isInterrupted 还是有很大的区别，调用该方法会直接擦除掉线程的 interrupt 标识，需要注意的是，如果当前线程被打断了，那么第一次调用 interrupted 方法会返回 true，并且立即擦除了 interrupt 标识；第二次包括以后的调用永远都会返回 false，除非在此期间线程又一次被打断，下面设计了一个简单的例子，来验证我们的说法：

```
package com.bjsxt.chapter04.demo06;

import java.util.concurrent.TimeUnit;

public class ThreadInterrupt04 {
```



```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread() {
        @Override
        public void run() {
            while (true) {
                System.out.println(Thread.interrupted());
            }
        }
    };
    t1.setDaemon(true);
    t1.start();

    // 短暂的阻塞是为了保证 t1 线程已启动
    TimeUnit.MILLISECONDS.sleep(2);
    // 中断 t1 线程的阻塞状态
    t1.interrupt();
}
}
```

运行结果如下：

```
...
false
false
true
false
False
...
```

在很多的 false 包围中发现了一个 true，也就是说 interrupted 方法判断到了其被中断，立即擦除了中断标识，并且只有这一次返回 true，后面的都将会是 false。

4.6.4、interrupt 注意事项

打开 Thread 的源码，不难发现，interrupted 方法和 interrupted 方法都调用了同一个本地方法：

```
private native boolean isInterrupted(boolean ClearInterrupted);
```

其中参数 ClearInterrupted 主要用来控制是否擦除线程 interrupt 的标识。

isInterrupted 方法的源码中该参数为 false，表示不想擦除：

```
public boolean isInterrupted() {
    return isInterrupted(false);
}
```

而 interrupted 静态方法中该参数则为 true，表示想要擦除：

```
public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}
```

在比较详细的学习了 interrupt 方法之后，大家思考一个问题，如果一个线程在没有执行可中断方法之前就被打断，那么其接下来将执行可中断方法，比如 sleep 会发生什么样的情况呢？下面我们通过一个简单的实验来回答这个疑问：

```
package com.bjsxt.chapter04.demo06;

import java.util.concurrent.TimeUnit;

public class ThreadInterrupt05 {

    public static void main(String[] args) {
        // 判断当前线程是否被中断
        System.out.printf("Main thread is interrupted? %s\n",
            Thread.interrupted());

        // 中断当前线程
        Thread.currentThread().interrupt();

        // 判断当前线程是否已经被中断
        System.out.printf("Main thread is interrupted? %s\n",
            Thread.interrupted());

        try {
            // 当前线程执行可中断方法
            TimeUnit.MINUTES.sleep(1);
        } catch (InterruptedException e) {
            // 捕获中断信号
            System.out.println("I will be interrupted still.");
            e.printStackTrace();
        }
    }
}
```

通过运行上面的程序，你会发现，如果一个线程设置了 interrupt 标识，那么接下来的可中断方法会立即中断，因此最后一步的捕获中断信号部分代码会被执行。

4.7、线程 join

Thread 的 join 方法同样是一个非常重要的方法，使用它的特性可以实现很多比较强大的功能，Thread 的 API 为我们提供了三个不同的 join 方法，具体如下。

- `public final void join() throws InterruptedException`
- `public final void join (long millis) throws InterruptedException`
- `public final void join (long millis, int nanos)`
`throws InterruptedException`

在本节中，将会详细介绍 join 方法以及如何在实际应用中使用 join 方法。

4.7.1、线程 join 方法详解

join 某个线程 A，会使当前线程 B 进入等待，直到线程 A 结束生命周期，或者到达给定的时间，那么在此期间 B 线程是处于 Blocked 的，而不是 A 线程，下面就来通过一个简单的实例解释一下 join 方法的基本用法：

```
package com.bjsxt.chapter04.demo07;

public class ThreadJoin {

    public static void main(String[] args) throws InterruptedException {
        // 1. 定义两个线程
        Thread t1 = new Thread(() -> printNum());
        Thread t2 = new Thread(() -> printNum());

        // 2. 启动这两个线程
        t1.start();
        t2.start();

        // 3. 执行这两个线程的 join 方法
        t1.join();
        t2.join();

        // 4. main 线程循环输出
        printNum();
    }

    private static void printNum() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + "#" + i);
        }
    }
}
```

上面的代码结合 Java8 的语法，创建了两个线程，分别启动，并且调用了每个线程的 join 方法（注意：join 方法是被主线程调用的，因此在第一个线程还没结束生命周期的时候，第二个线程的 join 不会得到执行，但是此时，第二个线程也已经启动了），运行上面的程序，你会发现线程一和线程二会交替的输出直到他们结束生命周期，main 线程的循环才会开始运行，程序输出如下：

```
...
Thread-0#8
Thread-0#9
Thread-1#6
Thread-1#7
Thread-1#8
Thread-1#9
main#0
main#1
main#2
```

```
main#3
main#4
main#5
...
```

如果你将第三步下面的 join 全部注释掉，那么三个线程将会交替的输出，程序输出如下：

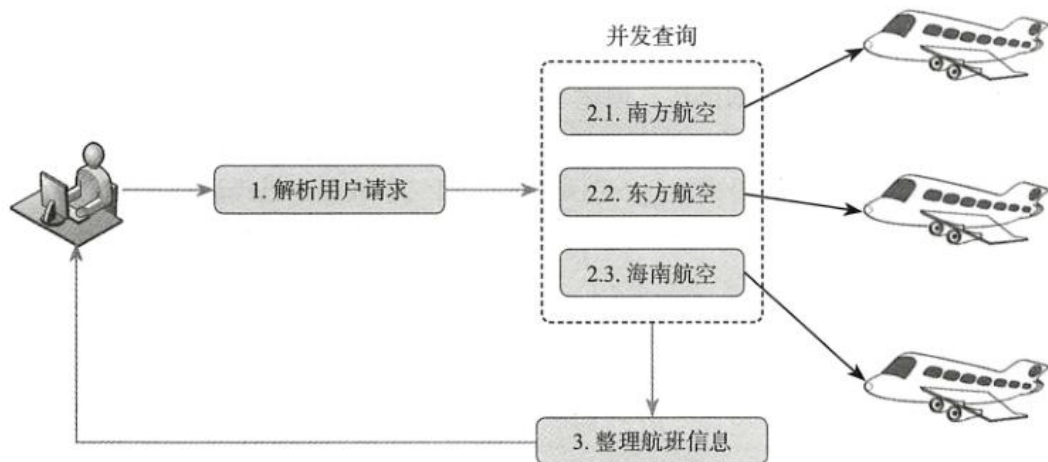
```
main#0
Thread-0#0
Thread-1#0
Thread-0#1
main#1
Thread-0#2
Thread-1#1
Thread-0#3
main#2
...
```

join 方法会使当前线程永远的等待下去，直到期间被另外的线程中断，或者 join 的线程执行结束，当然你也可以使用 join 的另外两个重载方法，指定毫秒数，在指定的时间到达之后，当前线程也会退出阻塞。

同样思考一个问题，如果一个线程已经结束了生命周期，那么调用它的 join 方法的当前线程会被阻塞吗？

4.7.2、join 方法结合实战

本节我们将结合一个实际的案例，来看一下 join 方法的应用场景，假设你有一个 APP，主要用于查询航班信息，你的 APP 是没有这些实时数据的，当用户发起查询请求时，你需要到各大航空公司的接口获取信息，最后统一整理加工返回到 APP 客户端，如图所示，当然 JDK 自带了很多高级工具，比如 CountDownLatch 和 CyclicBarrier 等都可以完成类似的功能，但是仅就我们目前所学的知识，使用 join 方法即可完成下面的功能。



该例子是典型的串行任务局部并行化处理，用户在 APP 客户端输入出发地“北京”和目

的地“上海”，服务器接收到这个请求之后，先来验证用户的信息，然后到各大航空公司的接口查询信息，最后经过整理加工返回给客户端，每一个航空公司的接口不会都一样，获取的数据格式也不一样，查询的速度也存在着差异，如果再跟航空公司进行串行化交互（逐个地查询），很明显客户端需要等待很长的时间，这样的话，用户体验就会非常差。如果我们将每一个航空公司的查询都交给一个线程去工作，然后在它们结束工作之后统一对数据进行整理，这样就可以极大地节约时间，从而提高用户体验效果。

```
package com.bjsxt.chapter04.demo08;

import java.util.List;

/**
 * 航班查询
 */
public interface FlightQuery {

    List<String> get();

}
```

以上代码中，FlightQuery 提供了一个返回方法，学到这里大家应该注意到了，不管是 Thread 的 run 方法，还是 Runnable 接口，都是 void 返回类型，如果你想通过某个线程的运行得到结果，就需要自己定义一个返回接口。

查询 Flight 的 task，其实就是一个线程的子类，主要用于到各大航空公司获取数据，示例代码如下：

```
package com.bjsxt.chapter04.demo08;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

public class FlightQueryTask extends Thread implements FlightQuery {

    private final String origin; // 出发地
    private final String destination; // 目的地
    private final List<String> flightList = new ArrayList<>(); // 航班信息集合

    public FlightQueryTask(String airline, String origin, String destination) {
        super("[ " + airline + " ]"); // 使用航空公司名称为线程命名
        this.origin = origin;
        this.destination = destination;
    }

    @Override
    public void run() {
        // 航空公司查询从出发地到目的地
        System.out.printf("%s-query from %s to %s \n", this.getName(), origin, destination);
    }
}
```

```
// 随机生成随机数模拟查询时长
int randomVal = ThreadLocalRandom.current().nextInt(10);
try {
    TimeUnit.SECONDS.sleep(randomVal);
    // 将航空公司和查询时长添加至航班信息集合中
    this.flightList.add(this.getName() + "-" + randomVal);
    System.out.printf("The Flight:%s list query successful\n",
this.getName());
} catch (InterruptedException e) {
    e.printStackTrace();
}
}

@Override
public List<String> get() {
    return this.flightList;
}
}
```

接口定义好了，查询航班数据的线程也有了，下面就来实现一下从 SH(上海)到 BJ(北京)的航班查询吧。示例代码如下：

```
package com.bjsxt.chapter04.demo08;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class FlightQueryExample {

    // APP 合作的航空公司
    private static List<String> airlineList = Arrays.asList("南方航空", "
    东方航空", "海南航空");

    public static void main(String[] args) {
        List<String> results = search("SH", "BJ");
    }

    private static List<String> search(String origin, String destination)
    {
        final List<String> result = new ArrayList<>();
        // 创建查询航班信息的线程列表
        List<FlightQueryTask> tasks = airlineList.stream()
            .map(f -> new FlightQueryTask(f, origin, destination))
            .collect(Collectors.toList());
        // 启动线程
        tasks.forEach(Thread::start);
        // 调用每一个线程的 join 方法，阻塞当前线程
        tasks.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
}
```



```
    }  
    });  
    // 当前线程阻塞，获取每一个查询线程的结果，并且加入 result 中  
    tasks.stream().map(FlightQuery::get).forEach(result::addAll);  
    return result;  
}  
}
```

上面的代码，关键的地方已通过注释解释得非常清楚，主线程收到了 search 请求之后，交给了若干个查询线程分别进行工作，最后将每一个线程获取的航班数据进行统一的汇总。由于每个航空公司的查询时间可能不一样，所以用了一个随机值来反应不同的查询速度，返回给客户端（打印到控制台），程序的执行结果输出如下：

```
[南方航空]-query from SH to BJ  
[海南航空]-query from SH to BJ  
[东方航空]-query from SH to BJ  
The Flight:[海南航空] list query successful  
The Flight:[东方航空] list query successful  
The Flight:[南方航空] list query successful  
-----result-----  
[南方航空]-7  
[东方航空]-2  
[海南航空]-1
```

4.8、如何关闭一个线程

JDK 有一个 Deprecated 方法 stop，但是该方法存在一个问题，JDK 官方早已经不推荐使用，其在后面的版本中有可能被移除，根据官网的描述，该方法在关闭线程时可能不会释放掉 monitor 的锁，所以强烈建议不要使用该方法结束线程，本节将主要介绍几种关闭线程的方法。

4.8.1、正常关闭

A. 线程结束生命周期正常结束

线程运行结束，完成了自己的使命之后，就会正常退出，如果线程中的任务耗时比较短，或者时间可控，那么放任它正常结束就好了。

B. 捕获中断信号关闭线程

我们通过 new Thread 的方式创建线程，这种方式看似很简单，其实它的派生成本是比较高的，因此在一个线程中往往会循环地执行某个任务，比如心跳检查，不断地接收网络消息报文等，系统决定退出的时候，可以借助中断线程的方式使其退出，示例代码如下：

```
package com.bjsxt.chapter04.demo09;

import java.util.concurrent.TimeUnit;

public class InterruptThreadExit {

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread() {
            @Override
            public void run() {
                System.out.println("I will start work.");
                while (!isInterrupted()) {
                    // working.
                }
                System.out.println("I will be exiting.");
            }
        };
        t.start();
        TimeUnit.SECONDS.sleep(5);
        System.out.println("System will be shutdown.");
        t.interrupt();
    }
}
```

上面的代码是通过检查线程 interrupt 的标识来决定是否退出的，如果在线程中执行某个可中断方法，则可以通过捕获中断信号来决定是否退出。

C. 使用 volatile 开关控制

由于线程的 interrupt 标识很有可能被擦除，或者逻辑单元中不会调用任何可中断方法，所以使用 volatile 修饰的开关 flag 关闭线程也是一种常用的做法，具体如下：

```
package com.bjsxt.chapter04.demo09;

import java.util.concurrent.TimeUnit;

public class FlagThreadExit {

    public static void main(String[] args) throws InterruptedException {
        MyTask t = new MyTask();
        t.start();
        TimeUnit.SECONDS.sleep(5);
        System.out.println("System will be shutdown.");
    }
}
```

```
t.close();  
}  
}  
  
class MyTask extends Thread {  
  
    private volatile boolean closed = false;  
  
    @Override  
    public void run() {  
        System.out.println("I will start work.");  
        while (!closed && !isInterrupted()) {  
            // working.  
        }  
        System.out.println("I will be exiting.");  
    }  
  
    public void close() {  
        this.closed = true;  
        this.interrupt();  
    }  
}
```

上面的例子中定义了一个 `closed` 开关变量，并且是使用 `volatile` 修饰（关于 `volatile` 关键字会在后面进行非常细致地讲解，`volatile` 关键字在 Java 中是一个革命性的关键字，非常重要，它是 Java 原子变量以及并发包的基础），运行上面的程序同样也可以关闭线程。

4.8.2、异常退出

在一个线程的执行单元中，是不允许抛出 `checked` 异常的，不论 `Thread` 中的 `run` 方法，还是 `Runnable` 中的 `run` 方法，如果线程在运行过程中需要捕获 `checked` 异常并且判断是否还有运行下去的必要，那么此时可以将 `checked` 异常封装成 `unchecked` 异常（`RuntimeException`）抛出进而结束线程的生命周期。

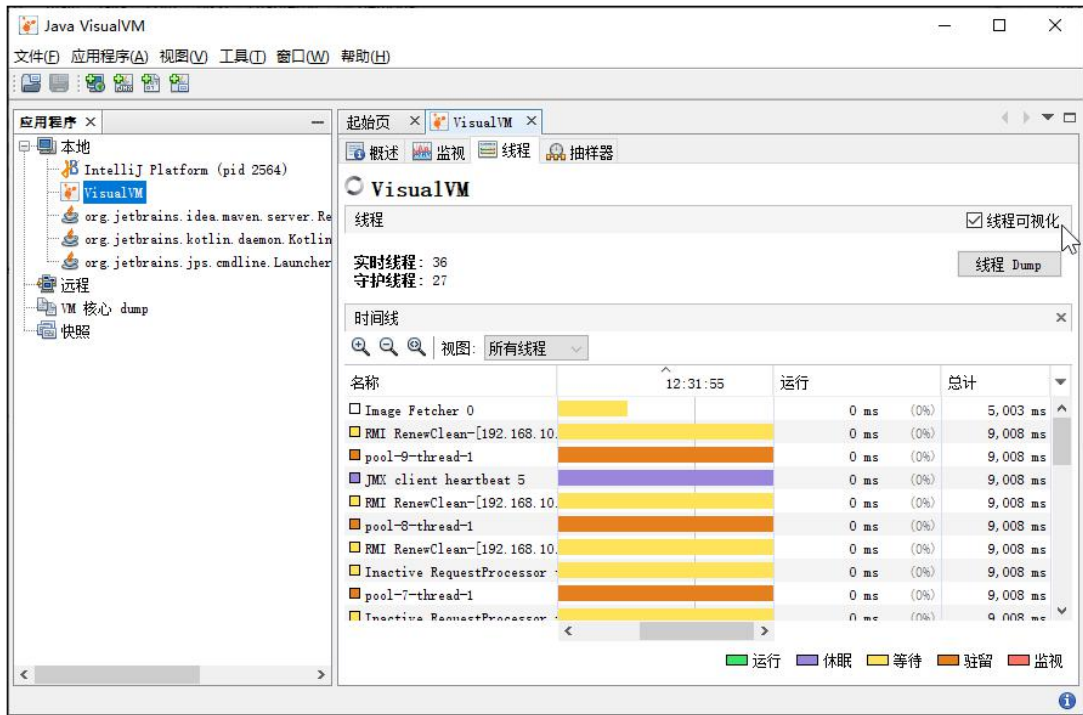
4.8.3、进程假死

相信很多程序员都会遇到进程假死的情况，所谓假死就是进程虽然存在，但没有日志输出，程序不进行任何的作业，看起来就像死了一样，但事实上它是没有死的，程序之所以出现这样的情况，绝大部分的原因就是某个线程阻塞了，或者线程出现了死锁的情况。

我们需要借助一些工具来帮助诊断，比如 `jstack`、`jconsole`、`jvisualvm` 等工具，在本节中，这一节简单介绍一下 `jvisualvm` 这个可视化工具，后面我们还会接触这些工具

进行死锁的判断等操作。

IntelliJ IDEA 其实也是一个 Java 进程，打开 jvisualvm，选择 IntelliJ IDEA 进程，如下图所示，将右侧的 Tab 切换到【线程】。



如果进程无法退出，则会出现假死的情况，可以打开 jvisualvm 查看有哪些活跃线程，它们的状态是什么，该线程在调用哪个方法而进入了阻塞。

4.9、总结

在本章中，我们比较详细地学习了 Thread 的大多数 API，其中有获取线程信息的方法，如 `getId()`、`getName()`、`getPriority()`、`currentThread()`，也有阻塞方法 `sleep()`、`join()` 等方法，并且结合若干个实战例子帮助大家更好地理解相关的 API，Thread 的 API 是掌握高并发编程的基础，因此非常有必要熟练掌握。

五、线程安全与数据同步

本章我们将学习多线程中最复杂也是最重要的内容之一，那就是数据同步、线程安全、锁等概念，在串行化的任务执行过程中，由于不存在资源的共享，线程安全的问题几乎不用考虑，但是串行化的程序，运行效率低下，不能最大化地利用 CPU 的计算能力，随着 CPU 核数的增加和计算速度的提升，串行化的任务执行显然是对资源的极大浪费，比如 B 客户提交了一个业务请求，只有等到 A 客户处理结束才能开始，这样的体验显然是用户无法忍受的。

无论是互联网系统，还是企业级系统，在追求稳定计算的同时也在追求更高的系统吞吐量，这也对系统的开发者提出了更高的要求，如何开发高效率的程序成了每个程序员必须掌握的技能，并发或者并行的程序并不意味着可以满足越多的 Thread，Thread 的多少对系统的性能来讲是一个抛物线，同时多线程的引入也带来了共享资源安全的隐患。在本章中，我们主要来探讨如何在安全的前提下高效地共享数据。

什么是共享资源？共享资源指的是多个线程同时对同一份资源进行访问（读写操作），被多个线程访问的资源就称为共享资源，如何保证多个线程访问到的数据是一致的，则被称为数据同步或者资源同步。

5.1、数据同步

5.1.1、数据不一致问题的引入

在第 2 章中，我们写了一个简单的营业大厅叫号机程序，当时我们设定的最大号码是 50（可能有些人已经测试出了问题），现在我们对该程序稍加修改，就会出现数据不一致的情况，具体如下：

```
package com.bjsxt.chapter05.demo01;

import java.util.concurrent.TimeUnit;

/**
 * 具体策略类 叫号机
 */
public class CounterWindowRunnable implements Runnable {

    // 最多受理 50 笔业务
    private static final int MAX = 50;

    // 起始号码，不做 static 修饰
    private int index = 1;

    @Override
    public void run() {
        while (index <= MAX) {
            try {
                TimeUnit.MILLISECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.format("请【%d】号到【%s】办理业务\n", index++,
                Thread.currentThread().getName());
        }
    }
}
```

```
public static void main(String[] args) {
    final CounterWindowRunnable task = new CounterWindowRunnable();
    new Thread(task, "一号窗口").start();
    new Thread(task, "二号窗口").start();
    new Thread(task, "三号窗口").start();
    new Thread(task, "四号窗口").start();
}
}
```

多次运行上述程序，每次都会有不一样的发现，但是总结起来主要有三个问题，具体如下。

- 第一，某个号码被略过没有出现。
- 第二，某个号码被多次显示。
- 第三，号码超过了最大值 50。

多次运行上面的程序，找出了数据不一致的几种情况，如下图所示。

请【498】号到【三号窗口】办理业务
请【497】号到【一号窗口】办理业务
请【501】号到【四号窗口】办理业务
请【503】号到【一号窗口】办理业务
请【502】号到【三号窗口】办理业务
请【500】号到【二号窗口】办理业务

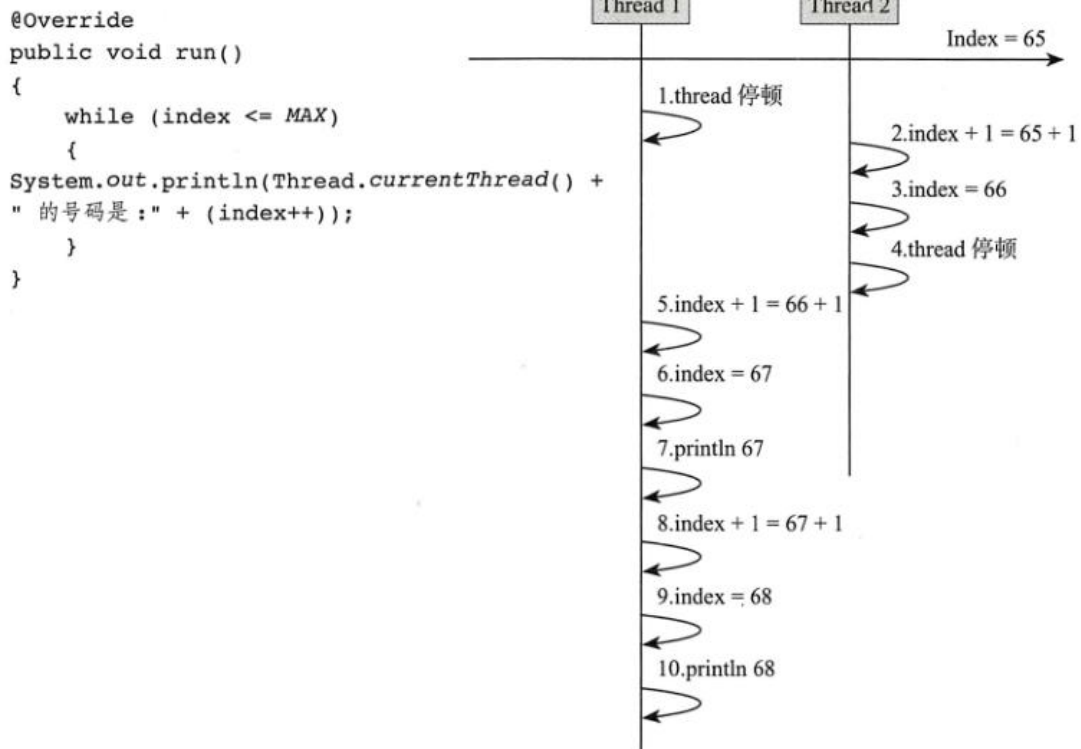
请【497】号到【三号窗口】办理业务
请【497】号到【四号窗口】办理业务
请【500】号到【一号窗口】办理业务
请【499】号到【三号窗口】办理业务
请【499】号到【四号窗口】办理业务
请【499】号到【二号窗口】办理业务

5.1.2、数据不一致问题原因分析

本节将针对上面出现的几种数据不一致问题分别进行分析。

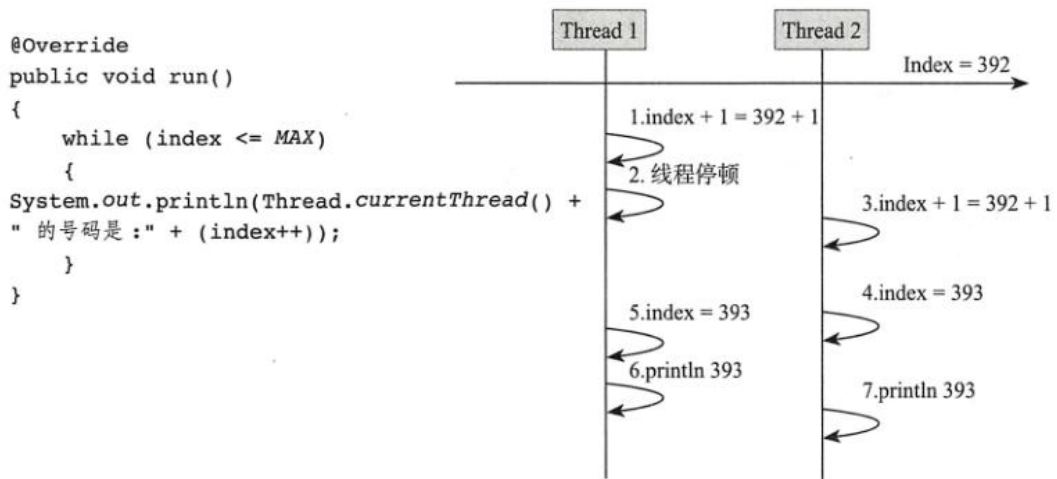
A. 号码被略过

如图所示，线程的执行是由 CPU 时间片轮询调度的，假设此时线程 1 和 2 都执行到了 index=65 的位置，其中线程 2 将 index 修改为 66 之后未输出之前，CPU 调度器将执行权利交给了线程 1，线程 1 直接将其累加到了 67，那么 66 就被忽略了。



B. 号码重复出现

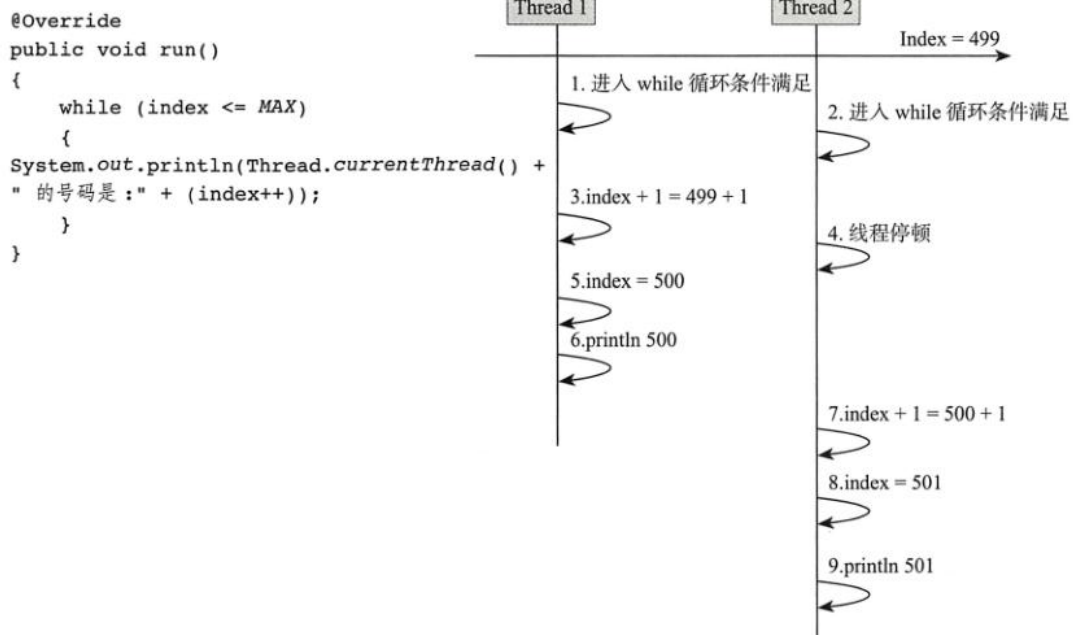
线程 1 执行 `index+1`, 然后 CPU 执行权落入线程 2 手里, 由于线程 1 并没有给 `index` 赋予计算后的结果 393, 因此线程 2 执行 `index+1` 的结果仍然是 393, 所以会出现重复号码的情况。



C. 号码超过了最大值

下面来分析一下号码超过最大值的情况，当 `index=499` 的时候，线程 1 和线程 2 都看到条件满足，线程 2 短暂停顿，线程 1 将 `index` 增加到了 500，线程 2 恢复运行后又将 500 增加到了 501，此时就出现了超过最大值的情况。

我们虽然使用了时序图的方式对数据同步问题进行了分析，但是这样的解释还是不够严谨，后面我们会讲解 Java 的内存模型以及 CPU 缓存等知识，到时候会更加清晰和深入的讲解数据不一致的问题。

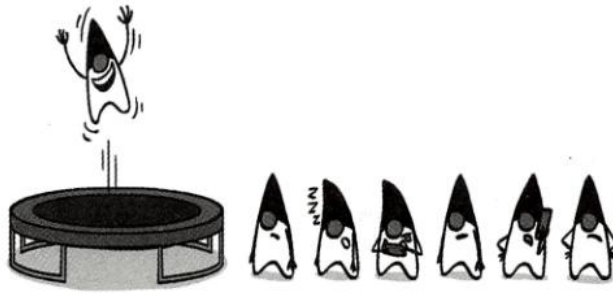


5.2、初识 synchronized 关键字

5.1.1 节出现的几个问题，究其原因就是因为多个线程对 `index` 变量（共享变量/资源）同时操作引起的，在 JDK1.5 版本以前，要解决这个问题需要使用 `synchronized` 关键字，`synchronized` 提供了一种排他机制，也就是在同一时间只能有一个线程执行某些操作，在本章中，我们就来详细地探讨一下 `synchronized` 关键字的本质和用法。

5.2.1、什么是 synchronized？

下面是一段来自于 JDK 官网对 `synchronized` 关键字比较权威的解释，如图所示。



Synchronized keyword enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.

上述解释的意思是：synchronized 关键字可以实现一个简单的策略来防止线程干扰和内存一致性错误，如果一个对象对多个线程是可见的，那么对该对象的所有读或者写都将通过同步的方式来进行，具体表现如下。

- synchronized 关键字提供了一种锁的机制，能够确保共享变量的互斥访问，从而防止数据不一致问题的出现。
- synchronized 关键字包括 monitor enter 和 monitor exit 两个 JVM 指令，它能够保证在任何时候任何线程执行到 monitor enter 成功之前都必须从主内存中获取数据，而不是从缓存中，在 monitor exit 运行成功之后，共享变量被更新后的值必须刷入主内存（后面会重点介绍）
- synchronized 的指令严格遵守 java happens-before 规则，一个 monitor exit 指令之前必定要有一个 monitor enter（后面会详细介绍）

5.3、synchronized 关键字的用法

synchronized 可以用于对代码块或方法进行修饰，而不能用于对 class 以及变量进行修饰。

5.3.1、同步方法

同步方法的语法非常简单即：

```
[default|public|private|protected] synchronized [static] type method()
```

示例代码如下：

```
public synchronized void sync() {
    ...
}
public synchronized static void staticSync() {
    ...
}
```

5.3.2、同步代码块

同步代码块的语法示例如下：

```
private final Object MUTEX = new Object();
public void sync() {
    synchronized (MUTEX) {
        ...
    }
}
```

介绍了什么是 `synchronized` 关键字以及它的基本用法之后，我们再次改写一下叫号程序：

```
package com.bjsxt.chapter05.demo01;

import java.util.concurrent.TimeUnit;

/**
 * 具体策略类 叫号机
 */
public class CounterWindowRunnable implements Runnable {

    // 最多受理 500 笔业务
    private static final int MAX = 500;

    // 起始号码，不做 static 修饰
    private int index = 1;

    private static final Object NUTEX = new Object();

    @Override
    public void run() {
        synchronized (NUTEX) {
            while (index <= MAX) {
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.format("请【%d】号到【%s】办理业务\n", index++,
Thread.currentThread().getName());
            }
        }
    }

    public static void main(String[] args) {
        final CounterWindowRunnable task = new CounterWindowRunnable();
        new Thread(task, "一号窗口").start();
        new Thread(task, "二号窗口").start();
        new Thread(task, "三号窗口").start();
        new Thread(task, "四号窗口").start();
    }
}
```

```
}
```

上面的程序无论运行多少次，都不会出现数据不一致的问题。

5.4、深入 synchronized 关键字

5.4.1、线程堆栈分析

synchronized 关键字提供了一种互斥机制，也就是说在同一时刻，只能有一个线程访问同步资源，很多资料、书籍将 synchronized(mutex) 称为锁，其实这种说法是不严谨的，准确地讲应该是某线程获取了与 mutex 关联的 monitor 锁（当然写程序的时候知道它想要表达的语义即可），下面我们来看一个简单的例子对其进行说明：

```
package com.bjsxt.chapter05.demo03;

import java.util.concurrent.TimeUnit;

public class Mutex {

    public static final Object MUTEX = new Object();

    public void accessResource() {
        synchronized (MUTEX) {
            try {
                TimeUnit.MINUTES.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

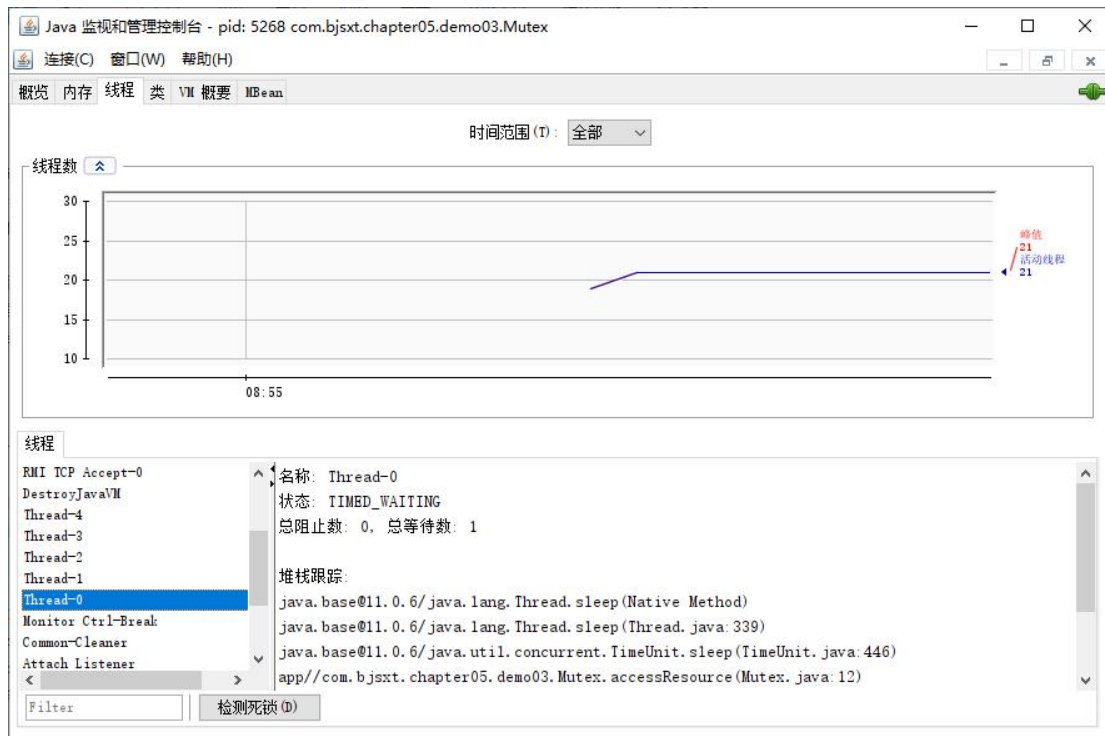
    public static void main(String[] args) {
        final Mutex mutex = new Mutex();
        for (int i = 0; i < 5; i++) {
            new Thread(mutex::accessResource).start();
        }
    }
}
```

上面的代码中定义了一个方法 accessResource，并且使用同步代码块的方式对 accessResource 进行了同步，同时定义了 5 个线程调用 accessResource 方法，由于同步代码块的互斥性，只能有一个线程获取了 mutex monitor 的锁，其他线程只能进入阻塞状态，等待获取 mutex monitor 锁的线程对其进行释放，运行上面的程序然后打开 JConsole 工具监控，如下图所示。



选中要建立连接的本地进程，然后点击【连接】按钮进入 JConsole 控制台，将 tab 切换至【线程】，如图所示。

随便选中程序中创建的某个线程，会发现只有个线程在 IMED WAITING(sleeping) 状态，其他线程都进入了 BLOCKED 状态，如图所示。



使用 jstack 命令打印进程的线程堆栈信息，选取其中几处关键的地方对其进行分析。Thread-0 持有 monitor <0x000000071118f160> 的锁并且处于休眠状态中，那么其他线程将会无法进入 accessResource 方法，如图所示。

```

C:\Windows\System32\cmd.exe

Microsoft Windows [版本 10.0.18363.959]
(c) 2019 Microsoft Corporation. 保留所有权利。

C:\Windows\system32>jps
5268 Mutex
7716
11864 Jps
14568 KotlinCompileDaemon
14872 RemoteMavenServer36
13724 JConsole
14508 Launcher

C:\Windows\system32>jstack 5268

"Thread-0" #14 prio=5 os_prio=0 cpu=0.00ms elapsed=152.12s tid=0x00000134c57b7000 nid=0x2528 waiting on condition [0x0000002c049fe000]
  java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:339)
    at java.util.concurrent.TimeUnit.sleep(Thread.java:446)
    at com.bjsxt.chapter05.demo03.Mutex.accessResource(Mutex.java:12)
    - locked <0x000000071118f160> (a java.lang.Object)
    at com.bjsxt.chapter05.demo03.Mutex$$Lambda$14/0x00000000800066c40.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:834)
    
```

Thread-1 线程进入 BLOCKED 状态并且等待着获取 monitor <0x000000071118f160> 的锁，其他的几个线程同样也是 BLOCKED 状态，如下图所示。

```

"Thread-1" #15 prio=5 os_prio=0 cpu=0.00ms elapsed=152.12s tid=0x00000134c57b8000 nid=0x2938 waiting for monitor entry
[0x00000002c0d4aff000]
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.bjsxt.chapter05.demo03.Mutex.accessResource(Mutex.java:12)
    - waiting to lock <0x0000000071118f160> (a java.lang.Object)
    at com.bjsxt.chapter05.demo03.Mutex$$Lambda$14/0x00000000800066c40.run(Unknown Source)
    at java.lang.Thread.run(java.base@11.0.6/Thread.java:834)
    
```

5.4.2、使用 synchronized 需要注意的问题

在详细了解了 synchronized 关键字的用法和本质之后，这里罗列了几个初学者容易出现的错误，以供参考。

A. 与 monitor 关联的对象不能为空

```

private static final Object MUTEX = null;

private void accessResource() {
    synchronized (MUTEX) {
        try {
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
    
```

Mutex 为 null，很多人还是会犯这么简单的错误，每一个对象和一个 monitor 关联，对象都为 null 了，monitor 肯定无从谈起。

B. synchronized 作用域太大

由于 synchronized 关键字存在排他性，也就是说所有的线程必须串行地经过 synchronized 保护的共享区域，如果 synchronized 作用域越大，则代表着其效率越低，甚至还会丧失并发的优势，示例代码如下：

```

@Override
public synchronized void run() {
    //
}
    
```

上面的代码对整个线程的执行逻辑单元都进行了 synchronized 同步，从而丧失了并发的能力，synchronized 关键字应该尽可能地只作用于共享资源（数据）的读写作用域。

C.不同的 monitor 企图锁相同的方法

```
package com.bjsxt.chapter05.demo04;

public class Task implements Runnable {

    private final Object MUTEX = new Object();

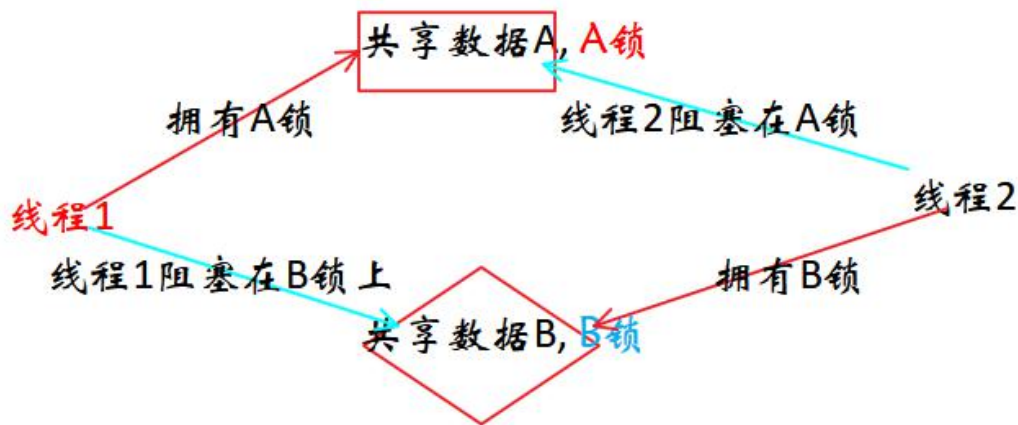
    @Override
    public void run() {
        synchronized (MUTEX) {

        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            new Thread(Task::new).start();
        }
    }
}
```

上面的代码构造了五个线程，同时也构造了五个 Runnable 实例，Runnable 作为线程逻辑执行单元传递给 Thread，然后你将发现，synchronized 根本互斥不了与之对应的作用域，线程之间进行 monitor lock 的争抢只能发生在与 monitor 关联的同一个引用上，上面的代码每一个线程争抢的 monitor 关联引用都是彼此独立的，因此不可能起到互斥的作用。

D. 多个锁的交叉导致死锁



线程1对共享资源A加锁成功 - A锁

线程2对共享资源b加锁成功 - B锁

线程1访问共享资源B，对b锁加锁 - 线程1阻塞在B锁上

线程2访问共享资源A，对A锁加锁 - 线程2阻塞在A锁上

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

5.5、This Monitor 和 Class Monitor 的详细介绍

通过 5.2 节和 5.3 节的学习，想必大家一定非常清楚，多个线程争抢同一个 monitor 的 lock 会陷入阻塞进而达到数据同步、资源同步的目的，在本章中我们将通过实例认识两个比较特别的 monitor。

5.5.1、this monitor

在下面的代码 ThisMonitor 中，两个方法 method1 和 method2 都被 synchronized 关键字修饰，启动了两个线程分别访问 method1 和 method2，在开始运行之前请读者思考一个问题：synchronized 关键字修饰了同一个实例对象的两个不同方法，那么与之对应的 monitor 是什么？两个 monitor 是否一致呢？

```
package com.bjsxt.chapter05.demo05;

import java.util.concurrent.TimeUnit;

public class ThisMonitor {

    public synchronized void method1() {
        System.out.println(Thread.currentThread().getName() + " enter to method1");
        try {
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public synchronized void method2() {
        System.out.println(Thread.currentThread().getName() + " enter to method2");
        try {
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ThisMonitor thisMonitor = new ThisMonitor();
        new Thread(thisMonitor::method1, "T1").start();
        new Thread(thisMonitor::method2, "T2").start();
    }
}
```

带着上面的疑问，运行程序将会发现只有一个方法被调用，另外一个方法根本没有被调用，分析线程的堆栈信息，执行 jdk 自带的 stack pid 命令，如图所示。

```
"T1" #14 prio=5 os_prio=0 cpu=0.00ms elapsed=13.43s tid=0x0000012cbc3e1800 nid=0x388c waiting on condition [0x0000002a9bf000]
java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(java.base@11.0.6/Native Method)
    at java.lang.Thread.sleep(java.base@11.0.6/Thread.java:339)
    at java.util.concurrent.TimeUnit.sleep(java.base@11.0.6/TimeUnit.java:446)
    at com.bjsxt.chapter05.demo05.ThisMonitor.method1(ThisMonitor.java:10)
    - locked <0x000000071118f478> (a com.bjsxt.chapter05.demo05.ThisMonitor)
    at com.bjsxt.chapter05.demo05.ThisMonitor$$Lambda$14/0x00000000800066c40.run(Unknown Source)
    at java.lang.Thread.run(java.base@11.0.6/Thread.java:834)

"T2" #15 prio=5 os_prio=0 cpu=0.00ms elapsed=13.43s tid=0x0000012cbc3e2800 nid=0x2860 waiting for monitor entry [0x00000002a9cfff000]
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.bjsxt.chapter05.demo05.ThisMonitor.method2(ThisMonitor.java:17)
    - waiting to lock <0x000000071118f478> (a com.bjsxt.chapter05.demo05.ThisMonitor)
    at com.bjsxt.chapter05.demo05.ThisMonitor$$Lambda$15/0x00000000800066040.run(Unknown Source)
    at java.lang.Thread.run(java.base@11.0.6/Thread.java:834)
```

笔者将重点的地方用红色的框标识了出来，T1 线程获取了<0x000000071118f478> monitor 的 lock 并且处于休眠状态，而 T2 线程企图获取<0x000000071118f478> monitor 的 lock 时陷入了 BLOCKED 状态，可见使用 synchronized 关键字同步类的不同实例方法，争抢的是同一个 monitor 的 lock，而与之关联的引用则是 ThisMonitor 的实例引用，为了证实我们的推论，将上面的代码稍作修改，如下所示：

```
package com.bjsxt.chapter05.demo05;

import java.util.concurrent.TimeUnit;

public class ThisMonitor02 {

    public synchronized void method1() {
        System.out.println(Thread.currentThread().getName() + " enter to method1");
    }
}
```



```

        try {
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void method2() {
        synchronized (this) {
            System.out.println(Thread.currentThread().getName() + " enter to method2");
            try {
                TimeUnit.MINUTES.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        ThisMonitor02 thisMonitor = new ThisMonitor02();
        new Thread(thisMonitor::method1, "T1").start();
        new Thread(thisMonitor::method2, "T2").start();
    }
}

```

其中，method1 保持方法同步的方式，method2 则采用了同步代码块的方式，并且使用的是 this 的 monitor，运行修改后的代码将会发现效果完全一样，在 JDK 官方文档中也有这样的描述。

<https://docs.oracle.com/javase/tutorial/essential/concurrency/locksynchron.html>

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

当线程调用一个同步方法时，它会自动获取该方法对象的内部锁，并在该方法返回时释放锁。即使返回是由未捕获的异常引起的，也会发生锁释放。

5.5.2、class monitor

同样的方式，来看下面的例子，有两个类方法（静态方法）分别使用 synchronized。对其进行同步：

```

package com.bjsxt.chapter05.demo05;

import java.util.concurrent.TimeUnit;

public class ClassMonitor {

    public synchronized static void method1() {
        System.out.println(Thread.currentThread().getName() + " enter to method1");
        try {
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```



```

    }

    public synchronized static void method2() {
        System.out.println(Thread.currentThread().getName() + " enter to method2");
        try {
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new Thread(ClassMonitor::method1, "T1").start();
        new Thread(ClassMonitor::method2, "T2").start();
    }
}

```

运行上面的例子，在同一时刻只能有一个线程访问 ClassMonitor 的静态方法，我们仍旧使用 jstack 命令分析其线程堆栈信息，如图所示。

```

"T1" #14 prio=5 os_prio=0 cpu=0.00ms elapsed=12.14s tid=0x000001a169d77800 nid=0x4070 waiting on condition [0x00000032d85fe000]
java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(java.base@11.0.6/Native Method)
    at java.lang.Thread.sleep(java.base@11.0.6/Thread.java:339)
    at java.util.concurrent.TimeUnit.sleep(java.base@11.0.6/TimeUnit.java:446)
    at com.bjsxt.chapter05.demo05.ClassMonitor.method1(ClassMonitor.java:10)
    - locked <0x0000000071118ed60> (a java.lang.Class for com.bjsxt.chapter05.demo05.ClassMonitor)
    at com.bjsxt.chapter05.demo05.ClassMonitor$$Lambda$14/0x00000000800066c40.run(Unknown Source)
    at java.lang.Thread.run(java.base@11.0.6/Thread.java:834)

"T2" #15 prio=5 os_prio=0 cpu=0.00ms elapsed=12.13s tid=0x000001a169d7c800 nid=0xf7c waiting for monitor entry [0x00000032d86ff000]
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.bjsxt.chapter05.demo05.ClassMonitor.method2(ClassMonitor.java:17)
    - waiting to lock <0x0000000071118ed60> (a java.lang.Class for com.bjsxt.chapter05.demo05.ClassMonitor)
    at com.bjsxt.chapter05.demo05.ClassMonitor$$Lambda$15/0x00000000800066c40.run(Unknown Source)
    at java.lang.Thread.run(java.base@11.0.6/Thread.java:834)

```

同样，将关键的地方用红色方框标识了出来，T1 线程持有<0x0000000071118ed60> monitor 的锁在正在休眠，而 T2 线程在试图获取<0x0000000071118ed60> monitor 锁的时候陷入了 BLOCKED 状态，因此我们可以得出用 synchronized 同步某个类的不同静态方法争抢的也是同一个 monitor 的 lock，再仔细对比堆栈信息会发现与 5.5.1 节中关于 monitor 信息不一样的地方在于 (a java.lang.Class for com.bjsxt.chapter05.demo05.ClassMonitor)，由此可以推断与该 monitor 关联的引用是 ClassMonitor.class 实例。

对上面的代码稍作修改，然后运行会发现具有同样的效果，示例代码如下：

```

package com.bjsxt.chapter05.demo05;

import java.util.concurrent.TimeUnit;

public class ClassMonitor02 {

    public synchronized static void method1() {
        System.out.println(Thread.currentThread().getName() + " enter to method1");
        try {
            TimeUnit.MINUTES.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void method2() {
        synchronized (ClassMonitor02.class) {
            System.out.println(Thread.currentThread().getName() + " enter to method2");
            try {

```

```
        TimeUnit.MINUTES.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    new Thread(ClassMonitor02::method1, "T1").start();
    new Thread(ClassMonitor02::method2, "T2").start();
}
```

其中静态方法 `method1` 继续保持同步方法的方式，而 `method2` 则修改为同步代码块的方式，使用 `ClassMonitor.class` 的实例引用作为 `monitor`，同样在 JDK 官方文档中对 `ClassMonitor` 也有比较权威的说明：

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

详见：

<https://docs.oracle.com/javase/tutorial/essential/concurrency/locksyn.c.html>

5.6、总结

本章继续使用营业厅叫号程序引入数据不一致的问题，并且重点分析了在多线程下出现数据不一致情况的各种可能性。

`synchronized` 关键字在 Java 中提供了同步语义，它可以保证在同一时间只允许一个线程访问共享数据资源，长期以来 `synchronized` 关键字的性能一直被诟病，但是随着 JDK 的不断发展，`synchronized` 关键字在同步过程中的性能也是逐步提升，使用得当性能不输 `LockSupport`，本章不仅详细介绍了 `synchronized` 关键字的作用和用法，而且更进一步深入地分析了 `synchronized` 关键字的内在原理。

在多线程访问共享资源的情况下，对线程驾驭不得当很容易引起死锁的情况发生，当然，如果程序出现死锁，那就必须要掌握如何对其进行诊断，在本章中我们也介绍了 `stack`、`jvisualvm` 等工具。

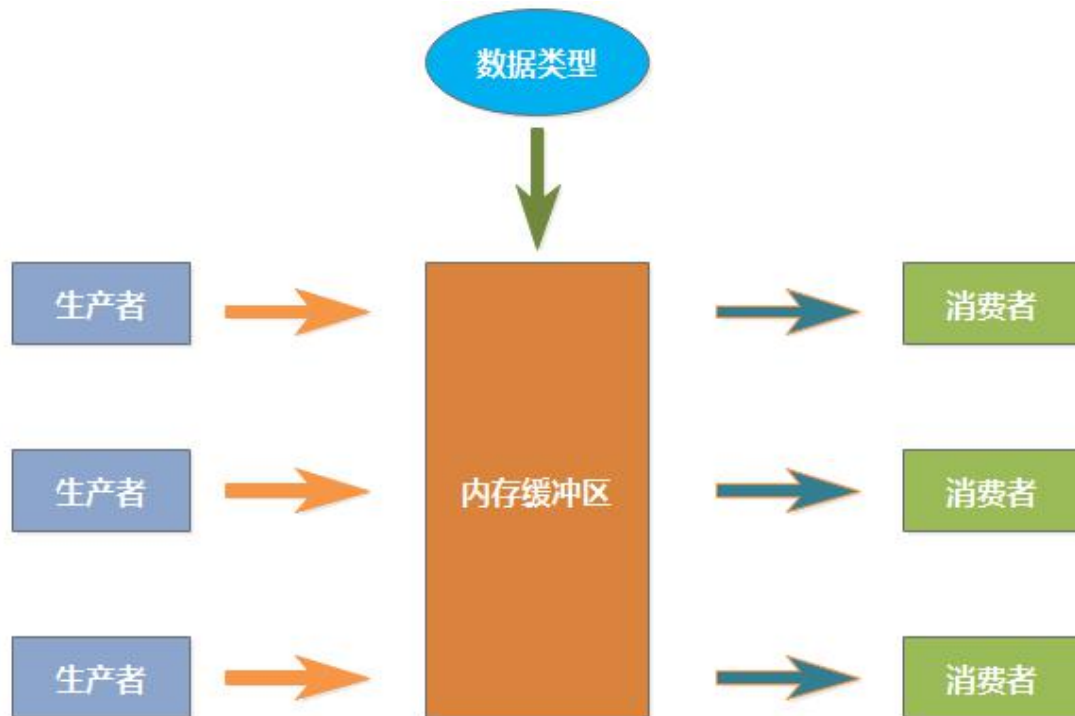
六、线程间通信

与网络通信等进程间通信方式不一样，线程间通信又称为进程内通信，多个线程实现互斥访问共享资源时会互相发送信号或等待信号，比如线程等待数据到来的通知，线程收到变量改变的信号等。本章将通过对一些案例的分析来学习 Java 提供的原生通信 API，以

及这些通信机制背后的内幕。

6.1、多线程间通信

6.1.1、初识 wait 和 notify



假设我们现在需要两个线程，一个负责生产数据，一个负责消费数据，理想状态下我们希望一边生产一边消费，对于初学者而言可能首先想到的代码是这样子的：

```
package com.bjsxt.chapter06.demo01;

public class ProduceConsumerVersion {

    private final Object LOCK = new Object();
    private int i = 0;

    public void produce() {
        synchronized (LOCK) {
            System.out.println("P -> " + (i++));
        }
    }

    public void consumer() {
        synchronized (LOCK) {
            System.out.println("C -> " + i);
        }
    }

    public static void main(String[] args) {
        ProduceConsumerVersion pc = new ProduceConsumerVersion();
    }
}
```

```

new Thread(() -> {
    while (true)
        pc.produce();
}).start();

new Thread(() -> {
    while (true)
        pc.consumer();
}).start();
}
}

```

上面的代码运行以后，结果和我们想的并不是一样的，在生产者产生了很多数据以后，消费者直接获取到最大的那个数据然后一直消费这个最大值，这个设计显然是失败的。

```

...
P -> 50173
P -> 50174
P -> 50175
C -> 50176
C -> 50176
C -> 50176
...

```

怎么修改这个代码呢？下面我们使用 wait 和 notify 重构代码，如下：

```

package com.bjsxt.chapter06.demo01;

public class ProduceConsumerVersion2 {

    private final Object LOCK = new Object();
    private int i = 0;
    private boolean isProduce = false;

    public void produce() {
        synchronized (LOCK) {
            // 如果生产了数据则等待消费者消费，否则生产数据
            if (isProduce) {
                try {
                    LOCK.wait(); // 等待消费者消费
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {
                i++;
                System.out.println("P -> " + i);
                isProduce = true;
                LOCK.notify(); // 生产者已生产，通知消费者消费
            }
        }
    }

    public void consumer() {
        synchronized (LOCK) {
            // 如果生产者生产了数据则消费，否则等待生产者生产数据
            if (isProduce) {
                System.out.println("C -> " + i);
                isProduce = false;
                LOCK.notify(); // 消费者已消费，通知生产者生产
            } else {
                try {

```

```

        LOCK.wait();// 等待生产者生产
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
}

public static void main(String[] args) {
    ProduceConsumerVersion2 pc = new ProduceConsumerVersion2();

    new Thread(() -> {
        while (true)
            pc.produce();
    }).start();

    new Thread(() -> {
        while (true)
            pc.consumer();
    }).start();
}
}

```

上面的代码运行结果如下：

```

...
P -> 1515
C ->1515
P -> 1516
C ->1516
P -> 1517
C ->1517
...

```

生产者生产数据以后会通知消费者消费，并进入阻塞等待状态；消费者消费以后会唤醒生产者生产数据，并进入阻塞等待状态，如此循环往复。从结果上看貌似已经实现了我们的需求但是在多个消费者和生产者的情况下可能效果又会不一样了，我们接着说。

6.2、多线程通信

将刚才的代码修改为以下代码，创建多个线程进行通信：

```

package com.bjsxt.chapter06.demo02;

import java.util.stream.Stream;

public class ProduceConsumerVersion3 {

    private final Object LOCK = new Object();
    private int i;
    private boolean isProduced;

    // 生产者
    public void produce() {
        synchronized (LOCK) {
            // 如果已生产者数据等待消费者消费，否则生产数据
            if (isProduced) {
                try {

```

```

        LOCK.wait();// 等待消费者消费
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
} else {
    i++; // 生产数据
    System.out.println("P -> " + i);
    isProduced = true;
    LOCK.notify();// 唤醒消费者进行消费
}
}
}

// 消费者
public void consumer() {
    synchronized (LOCK) {
        // 如果已生产者数据就消费数据，否则等待生产者生产数据
        if (isProduced) {
            System.out.println("C -> " + i);
            isProduced = false;
            LOCK.notify();// 唤醒生产者生产数据
        } else {
            try {
                LOCK.wait();// 等待生产者生产数据
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[] args) {
    ProduceConsumerVersion3 pc = new ProduceConsumerVersion3();

    Stream.of("P1", "P2").forEach(p -> {
        new Thread(() -> {
            while (true)
                pc.produce();
        }).start();
    });

    Stream.of("C1", "C2").forEach(p -> {
        new Thread(() -> {
            while (true)
                pc.consumer();
        }).start();
    });
}
}

```

运行以上代码以后会得到以下结果，这是为什么呢？我们分析一波：

```

P -> 1
C -> 1
P -> 2
C -> 2

```




怎么解决这个问题呢？继续往下看。

6.2.1、notifyAll

通过 notifyAll 可以解决以上问题，我们只需要唤醒所有其他线程即可，然后等待 CPU 调度获得锁资源继续执行。

```
package com.bjsxt.chapter06.demo02;

import java.util.stream.Stream;

public class ProduceConsumerVersion4 {

    private final Object LOCK = new Object();
    private int i;
    private boolean isProduced;

    // 生产者
    public void produce() {
        synchronized (LOCK) {
            // 如果已生产者数据等待消费者消费，否则生产数据
            if (isProduced) {
                try {
                    LOCK.wait(); // 等待消费者消费
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {
                i++; // 生产数据
                System.out.println(Thread.currentThread().getName() + " -> " + i);
                isProduced = true;
            }
        }
    }
}
```

```

        LOCK.notifyAll(); // 唤醒消费者进行消费
    }
}

// 消费者
public void consumer() {
    synchronized (LOCK) {
        // 如果已生产者数据就消费数据，否则等待生产者生产数据
        if (isProduced) {
            System.out.println(Thread.currentThread().getName() + " -> " + i);
            isProduced = false;
            LOCK.notifyAll(); // 唤醒生产者生产数据
        } else {
            try {
                LOCK.wait(); // 等待生产者生产数据
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[] args) {
    ProduceConsumerVersion4 pc = new ProduceConsumerVersion4();

    Stream.of("P1", "P2", "P3").forEach(p -> {
        new Thread(p) {
            @Override
            public void run() {
                while (true)
                    pc.produce();
            }
        }.start();
    });

    Stream.of("C1", "C2", "C3", "C4", "C5").forEach(c -> {
        new Thread(c) {
            @Override
            public void run() {
                while (true)
                    pc.consumer();
            }
        }.start();
    });
}
}

```

运行以上代码以后，从结果可以看到之前的问题已经得到解决。

6.3、wait 和 sleep 的区别

1. sleep 是属于 Thread 的方法，wait 属于 Object；
2. sleep 方法不需要被唤醒，wait 需要。
3. sleep 方法不需要 synchronized，wait 需要；
4. sleep 不会释放锁，wait 会释放锁并将线程加入 wait 队列；

七、脏读

对于对象的同步和异步的方法，我们在设计自己的程序的时候，一定要考虑问题的整体，不然就会出现数据不一致的错误，很经典的错误就是脏读（dirtyread）示例代码如下：

```
package com.bjsxt.chapter07;

import java.util.concurrent.TimeUnit;

/**
 * 业务整体需要使用完整的 synchronized，保持业务的原子性。
 */
public class DirtyRead {

    private String username = "bjsxt";
    private String password = "123";

    public synchronized void setValue(String username, String password) {
        this.username = username;

        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        this.password = password;

        System.out.println("setValue 最终结果: username = " + username + " , password = " + password);
    }

    public void getValue() {
        System.out.println("getValue 方法得到: username = " + this.username + " , password = " + this.password);
    }

    public static void main(String[] args) {
        final DirtyRead dr = new DirtyRead();

        Thread t1 = new Thread(() -> dr.setValue("z3", "456"));
        t1.start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        dr.getValue();
    }
}
```

示例总结：

在我们对一个对象的方法加锁的时候，需要考虑业务的整体性，即为

setValue/getValue 方法同时加锁 synchronized 同步关键字，保证业务（service）的原子性，不然会出现业务错误（也从侧面保证业务的一致性）。

八、volatile 关键字

8.1、volatile 关键字的概念

自 Java1.5 版本起，volatile 关键字所扮演的角色越来越重要，该关键字也成为并发包的基础，所有的原子数据类型都以此作为修饰。相比 synchronized 关键字，volatile 被称为“轻量级锁”，能实现部分 synchronized 关键字的语义。

volatile 概念：volatile 关键字的主要作用是使变量在多个线程间可见。下面我们通过一个案例来了解一下 volatile：

```
package com.bjsxt.chapter08;

import java.util.concurrent.TimeUnit;

public class VolatileDemo extends Thread {

    private boolean isRunning = true;

    private void setRunning(boolean isRunning) {
        this.isRunning = isRunning;
    }

    public void run() {
        System.out.println("进入 run 方法..");
        int i = 0;
        while (isRunning) {
            //..
        }
        System.out.println("线程停止");
    }

    public static void main(String[] args) throws InterruptedException {
        VolatileDemo rt = new VolatileDemo();
        rt.start();
        TimeUnit.SECONDS.sleep(2);
        rt.setRunning(false);
        System.out.println("isRunning 的值已经被设置了 false");
    }
}
```

示例总结：

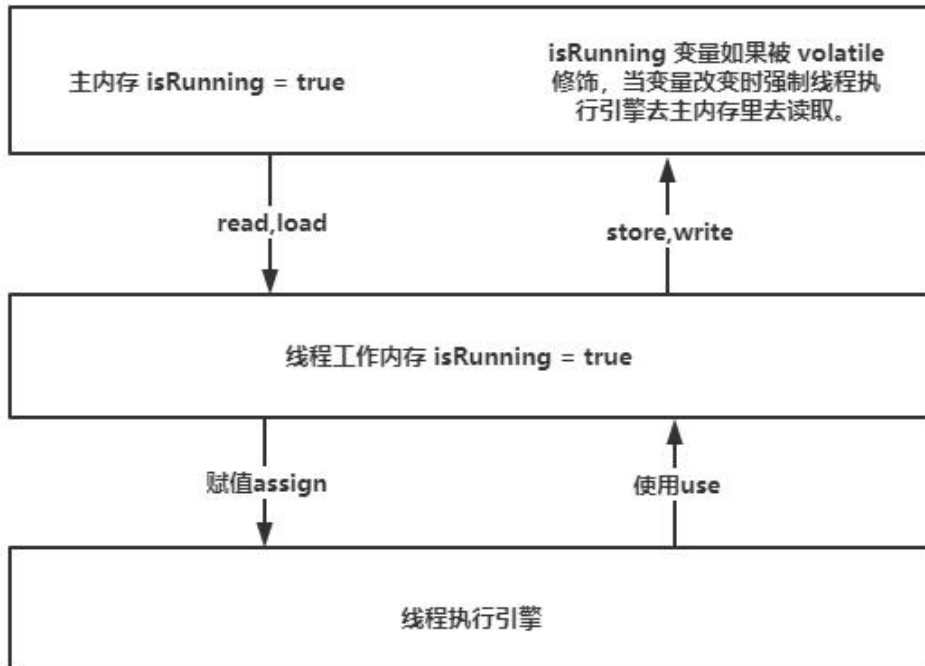
在 Java 中，每一个线程都会有一块工作内存区，其中存放着所有线程共享的主内存中的变量值的拷贝。当线程执行时，他在自己的工作内存区中操作这些变量。为了存取一个共享的变量，线程通常先获取锁定并去清除它的内存工作区，把这些共享变量从所有线程的共

享内存区中正确的装入到他自己的工作内存区中,当线程解锁时保证该工作内存区中变量的值写回到共享内存中。

一个线程可以执行的操作有使用(use)、赋值(assign)、装载(load)、存储(store)、锁定(lock)、解锁(unlock)。

而主内存可以执行的操作有读(read)、写(write)、锁定(lock)、解锁(unlock) 每个操作都是原子的。

volatile 的作用就是强制线程到主内存（共享内存）里去读取变量，而不去线程工作内存区里去读取，从而实现了多个线程间的变量可见。也就是满足线程安全的可见性。



8.2、volatile 关键字的非原子性

volatile-关键字虽然拥有多个线程之间的可见性,但是却不具备同步性(也就是原子性),可以算上是一个轻量级的 synchronized,性能要比 synchronized 强很多,不会造成阻塞(在很多开源的架构里,比如 netty 的底层代码就大量使用 volatile,可见 netty 性能一定是非常不错的。)这里需要注意:一般 volatile 用于只针对于多个线程可见的变量操作,并不能代替 synchronized 的同步功能,示例代码如下:

```
package com.bjsxt.chapter08;

import java.util.concurrent.atomic.AtomicInteger;

/**
 * volatile 关键字不具备 synchronized 关键字的原子性（同步）
 */
public class VolatileNoAtomic extends Thread {
    // private static volatile int count;
```

```
private static AtomicInteger count = new AtomicInteger(0);

private static void addCount() {
    for (int i = 0; i < 1000; i++) {
        // count++ ;
        count.incrementAndGet();
    }
    System.out.println(count);
}

public void run() {
    addCount();
}

public static void main(String[] args) {
    VolatileNoAtomic[] arr = new VolatileNoAtomic[10];
    for (int i = 0; i < 10; i++) {
        arr[i] = new VolatileNoAtomic();
    }

    for (int i = 0; i < 10; i++) {
        arr[i].start();
    }
}
```

示例总结：

volatile 关键字只具有可见性，没有原子性。要实现原子性建议使用 atomic 类的系列对象，支持原子性操作（注意 Atomica 类只保证本身方法原子性，并不保证多次操作的原子性）示例如下：

```
package com.bjsxt.chapter08;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicUse {

    private static AtomicInteger count = new AtomicInteger(0);

    /*
     * 多个 addAndGet 在一个方法内是非原子性的，
     * 需要加 synchronized 进行修饰，保证 4 个 addAndGet 整体原子性
     */
    public synchronized int multiAdd() {
        try {
            TimeUnit.MILLISECONDS.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        count.addAndGet(1);
        count.addAndGet(2);
        count.addAndGet(3);
        count.addAndGet(4); // +10
        return count.get();
    }

    public static void main(String[] args) throws InterruptedException {
        final AtomicUse au = new AtomicUse();
    }
}
```



```
List<Thread> ts = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    ts.add(new Thread(() -> System.out.println(au.multiAdd())));
}

for (Thread t : ts) {
    t.start();
}

TimeUnit.SECONDS.sleep(2);
System.out.println("count = " + count); // 1000
}
}
```

十、ThreadLocal

ThreadLocal 概念：线程局部变量，是一种多线程间并发访问变量的解决方案。与其 synchronized 等加锁的方式不同， ThreadLocal 完全不提供锁，而使用以空间换时间的手段，为每个线程提供变量的独立副本，以保障线程安全。

从性能上说， ThreadLocal 不具有绝对的优势，在并发不是很高的时候，加锁的性能会更好，但作为一套与锁完全无关的线程安全解决方案，在高并发量或者竞争激烈的场景，使用 ThreadLocal 可以在一定程度上减少锁竞争。

```
package com.bjsxt.chapter10;

import java.util.concurrent.TimeUnit;

public class ConnThreadLocal {

    public static ThreadLocal<String> th = new ThreadLocal<>();

    public void setTh(String value) {
        th.set(value);
    }

    public void getTh() {
        System.out.println(Thread.currentThread().getName() + ":" + this.th.get());
    }

    public static void main(String[] args) throws InterruptedException {

        final ConnThreadLocal ct = new ConnThreadLocal();
        Thread t1 = new Thread(() -> {
            ct.setTh("张三");
            ct.getTh();
        }, "t1");

        Thread t2 = new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
                ct.getTh();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "t2");
```

```
t1.start();  
t2.start();  
}  
}
```

十一、同步类容器

同步类容器都是线程安全的，同步容器类包括 Vector 和 Hashtable，二者都是早期 JDK 的一部分，此外还包括在 JDK1.2 当中添加的一些功能相似的类，这些同步的封装类是由 Collections.synchronizedXxx 等工厂方法创建的。

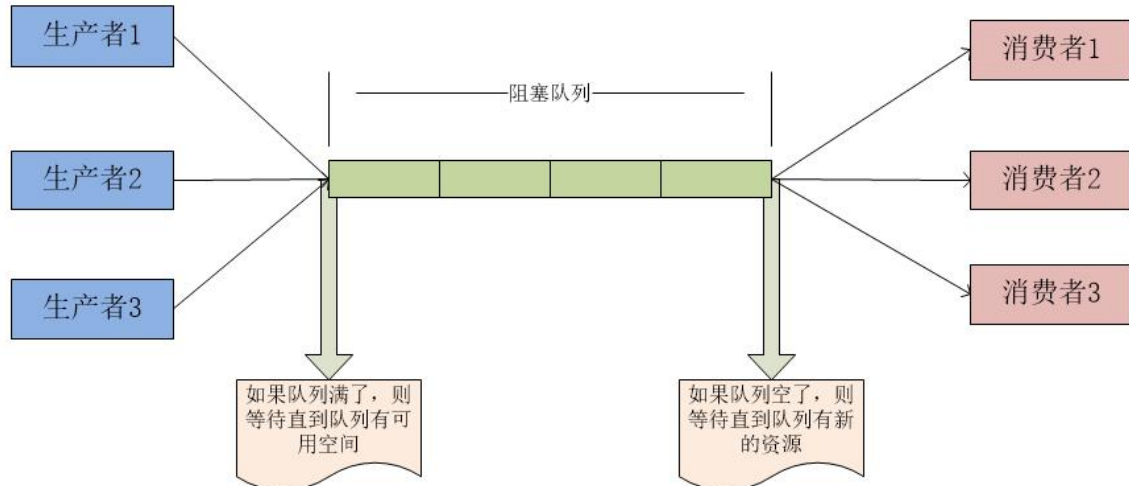
但在某些场景下可能需要加锁来保护复合操作。复合类操作如：迭代（反复访问元素，遍历完容器中所有的元素）、跳转（根据指定的顺序找到当前元素的下一个元素）、以及条件运算。这些复合操作在多线程并发地修改容器时，可能会表现出意外的行为，最经典的便是 ConcurrentModificationException，原因是当容器迭代的过程中，被并发的修改了内容，这是由于早期迭代器设计的时候并没有考虑并发修改的问题，示例代码如下：

```
package com.bjsxt.chapter11;  
  
import java.util.*;  
  
public class ConcurrentModificationExceptionDemo {  
  
    public static void main(String[] args) {  
        Vector<String> list = new Vector<>();  
        list.add("111");  
  
        Thread t1 = new Thread(() -> {  
            System.out.println(Thread.currentThread().getName() + "-----");  
            for (Iterator<String> iterator = list.iterator(); iterator.hasNext(); ) {  
                String element = iterator.next();  
                System.out.println("element = " + element);  
            }  
        });  
  
        Thread t2 = new Thread(() -> {  
            list.remove(0);  
        });  
  
        t1.start();  
        t2.start();  
    }  
}
```

通过阅读源码发现，同步类容器其底层的机制无非就是用传统的 synchronized 关键字对每个公用的方法都进行同步，使得每次只能有一个线程访问容器的状态。这很明显不满足我们今天互联网时代高并发的需求，在保证线程安全的同时，也必须要有足够好的性能。

九、模拟阻塞 Queue

BlockingQueue 即阻塞队列，它是基于 ReentrantLock，依据它的基本原理，我们可以实现 Web 中的长连接聊天功能，当然其最常用的还是用于实现生产者与消费者模式，大致如下图所示：



在 Java 中，BlockingQueue 是一个接口，它的实现类有 ArrayBlockingQueue、DelayQueue、LinkedBlockingDeque、LinkedBlockingQueue、PriorityBlockingQueue、SynchronousQueue 等，它们的区别主要体现在存储结构上或对元素操作上的不同，但是对于 take 与 put 操作的原理，却是类似的。

put(an Object)：把 an Object 加到 BlockingQueue 里，如果 BlockingQueue 没有空间，则调用此方法的线程被阻断，直到 BlockingQueue 里面有空间再继续。

take：取走 BlockingQueue 里排在首位的对象，若 BlockingQueue 为空，阻断进入等待状态直到 BlockingQueue 有新的数据被加入。

下面我们通过 LinkedList 模拟一个阻塞 Queue，代码如下：

```
package com.bjsxt.chapter09;

import java.util.LinkedList;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public class MyQueue {

    // 1 需要一个承装元素的集合
    private LinkedList<Object> list = new LinkedList<>();

    // 2 需要一个计数器
    private AtomicInteger count = new AtomicInteger(0);

    // 3 需要制定上限和下限
    private final int minSize = 0;

    private final int maxSize;
```

```
// 4 构造方法
public MyQueue(int size) {
    this.maxSize = size;
}

// 5 初始化一个对象 用于加锁
private final Object LOCK = new Object();

// put(an Object): 把 an Object 加到BlockingQueue 里, 如果BlockingQueue 没有空间, 则调用此
// 方法的线程被阻断, 直到BlockingQueue 里面有空间再继续.
public void put(Object obj) {
    synchronized (LOCK) {
        // 空间满了, 线程阻塞
        while (count.get() == this.maxSize) {
            try {
                LOCK.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 1 加入元素
        list.add(obj);
        // 2 计数器累加
        count.incrementAndGet();
        // 3 通知另外一个线程 (唤醒)
        LOCK.notify();
        System.out.println("新加入的元素为:" + obj);
    }
}

// take: 取走BlockingQueue 里排在首位的对象, 若BlockingQueue 为空, 阻断进入等待状态直到
// BlockingQueue 有新的数据被加入.
public Object take() {
    Object ret = null;
    synchronized (LOCK) {
        while (count.get() == this.minSize) {
            try {
                LOCK.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 1 做移除元素操作
        ret = list.removeFirst();
        // 2 计数器递减
        count.decrementAndGet();
        // 3 唤醒另外一个线程
        LOCK.notify();
    }
    return ret;
}

public int getSize() {
    return this.count.get();
}

public static void main(String[] args) {

    final MyQueue mq = new MyQueue(5);
    mq.put("a");
    mq.put("b");
    mq.put("c");
}
```

```
mq.put("d");
mq.put("e");

System.out.println("当前容器的长度:" + mq.getSize()); // 5

Thread t1 = new Thread(() -> {
    mq.put("f");
    mq.put("g");
}, "t1");
t1.start();

Thread t2 = new Thread(() -> {
    Object o1 = mq.take();
    System.out.println("移除的元素为:" + o1);
    Object o2 = mq.take();
    System.out.println("移除的元素为:" + o2);
}, "t2");

try {
    TimeUnit.SECONDS.sleep(2);
} catch (InterruptedException e) {
    e.printStackTrace();
}

t2.start();
}
```

十二、并发类容器

JDK 5.0 以后提供了多种并发类容器来替代同步类容器从而改善性能。同步类容器的状态都是串行化的。他们虽然实现了线程安全，但是严重降低了并发性，在多线程环境时，严重降低了应用程序的吞吐量。

并发类容器是专门针对并发设计的，使用 `ConcurrentHashMap` 来代替给予散列的传统的 `HashTable`，而且在 `ConcurrentHashMap` 中，添加了一些常见复合操作的支持。以及使用了 `CopyOnWriteArrayList` 代替 `Vocctor`，并发的 `CopyOnWriteArraySet`，以及并发的 `Queue`，`ConcurrentLinkedQueue` 和 `LinkedBlockingQueue`，前者是高性能的队列，后者是以阻塞形式的队列，具体实现 `Queue` 还有很多，例如 `ArrayBlockingQueue`、`PriorityBlockingQueue`、`SynchronousQueue` 等。

12.1、CurrentHashMap

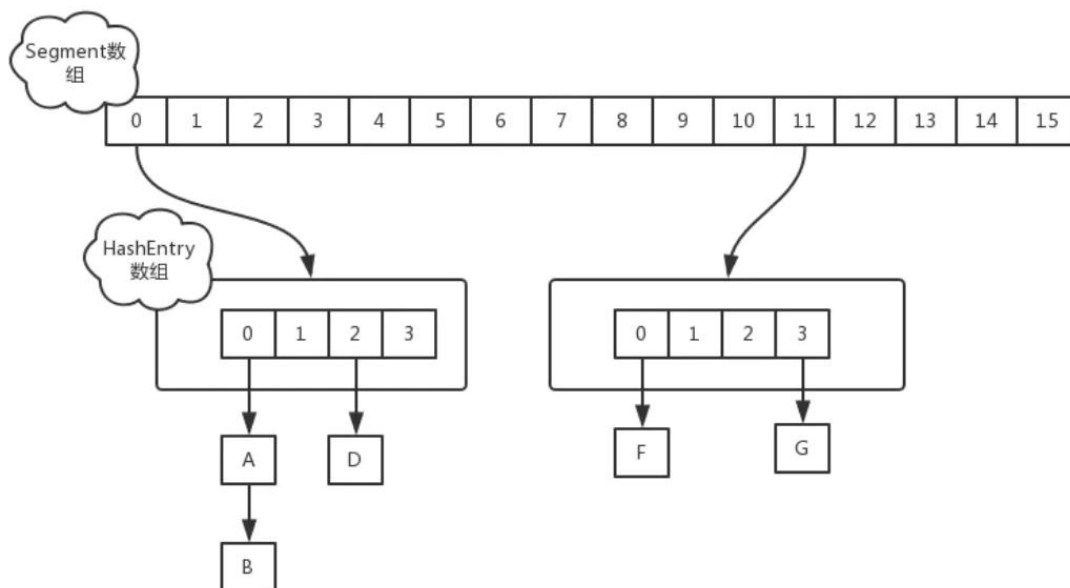
由于 `HashMap` 是线程不同步的，虽然处理数据的效率高，但是在多线程的情况下存在着安全问题，因此设计了 `CurrentHashMap` 来解决多线程安全问题。

`HashMap` 在 `put` 的时候，插入的元素超过了容量（由负载因子决定）的范围就会触发

扩容操作，就是 rehash，这个会重新将原数组的内容重新 hash 到新的扩容数组中，在多线程的环境下，存在同时其他的元素也在进行 put 操作，如果 hash 值相同，可能出现同时同一数组下用链表表示，造成闭环，导致在 get 时会出现死循环，所以 HashMap 是线程不安全的。

JDK7 下的 ConcurrentHashMap

在 JDK1.7 版本中，ConcurrentHashMap 的数据结构是由一个 Segment 数组和多个 HashEntry 组成，主要实现原理是实现了锁分离的思路解决了多线程的安全问题，如下图所示：



Segment 数组的意义就是将一个大的 table 分割成多个小的 table 来进行加锁，也就是上面的提到的锁分离技术，而每一个 Segment 元素存储的是 HashEntry 数组+链表，这个和 HashMap 的数据存储结构一样。

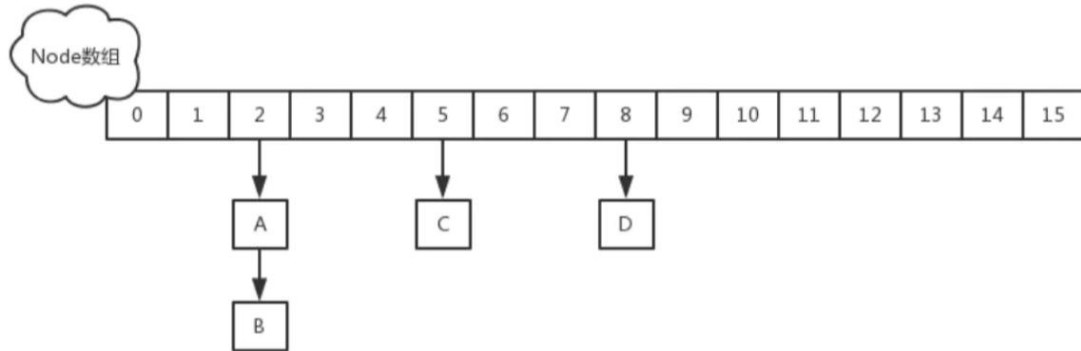
ConcurrentHashMap 内部使用段 (Segment) 来表示这些不同的部分，每个段其实就是个小的 HashTable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。把一个整体分成了 16 个段 (Segment)。也就是最高支持 16 个线程的并发修改操作。这也是在多线程场景时减小锁的粒度从而降低锁竞争的一种方案。并且代码中大多共享变量使用 volatile 关键字声明，目的是第一时间获取修改的内容，性能非常好。

JDK8 的 ConcurrentHashMap

JDK1.8 的实现已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 Synchronized 和 CAS 来操作，整个看起来就像是优化过且

线程安全的 HashMap，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本。

Node 是 ConcurrentHashMap 存储结构的基本单元，继承于 HashMap 中的 Entry，用于存储数据，Node 数据结构很简单，就是一个链表，但是只允许对数据进行查找，不允许进行修改。



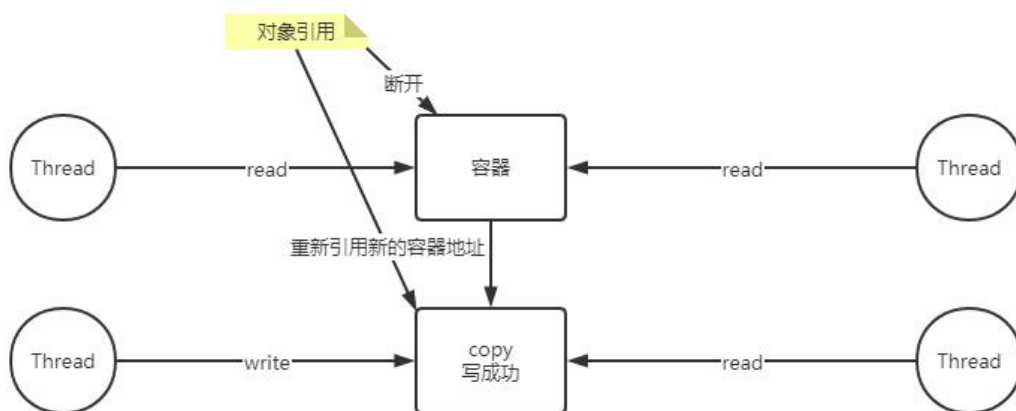
12.2、CopyOnWrite 容器

Copy-On-Write 简称 COW，是一种用于程序设计中的优化策略。

JDK 里的 COW 容器有两种：CopyOnWriteArrayList 和 CopyOnWriteArraySet，COW 容器非常有用，可以在非常多的并发场景中使用到。

什么是 CopyOnWrite 容器？

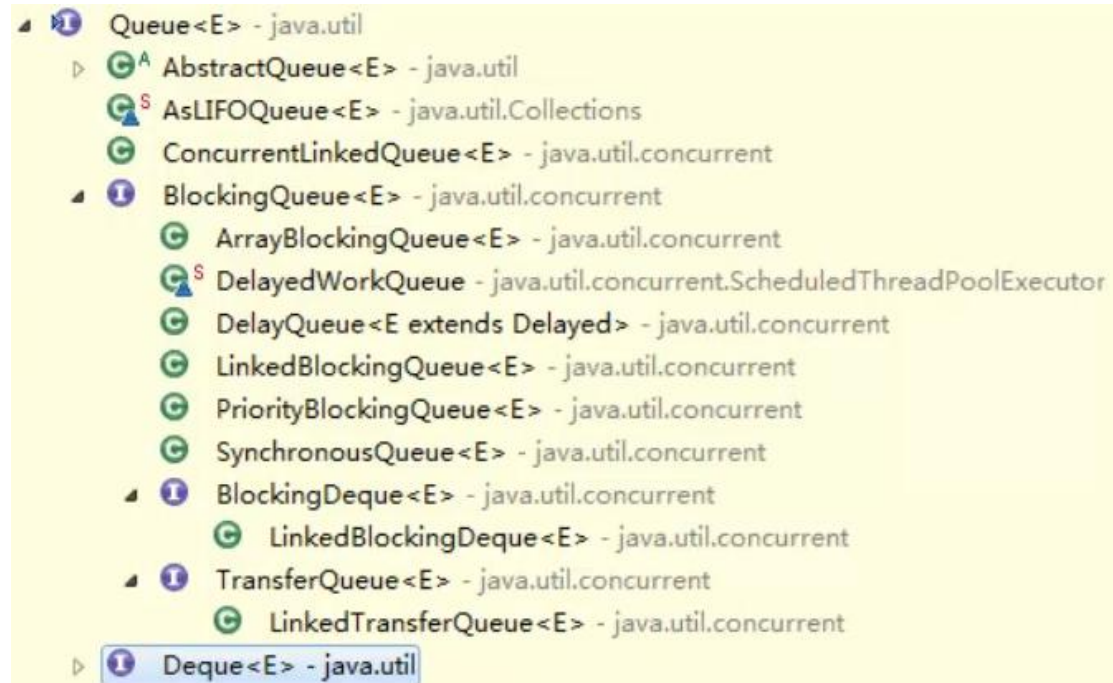
CopyOnWrite 容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对 CopyOnWrite 容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以 CopyOnWrite 容器也是一种读写分离的思想，读和写不同的容器。



比较适合读多写少的场景

十三、并发 Queue

在并发队列上 JDK 提供了两套实现，一个是以 ConcurrentLinkedQueue 为代表的高性能队列，一个是以 BlockingQueue 接口为代表的阻塞队列，无论哪种都继承自 Queue。



ConcurrentLinkedQueue

是一个适用于高并发场景下的队列，通过无锁的方式，实现了高并发状态下的高性能，通常 ConcurrentLinkedQueue 性能好于 BlockingQueue。它是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先加入的，尾是最近加入的，该队列不允许 null 元素

ConcurrentLinkedQueue 重要方法：

add()和 offer()都是加入元素的方法（在 ConcurrentLinkedQueue 中，这两个方法没有任何区别）

poll()和 peek()都是取头元素节点，区别在于前者会删除元素，后者不会。

```
// 高性能无阻塞无界队列: ConcurrentLinkedQueue
ConcurrentLinkedQueue<String> clq = new ConcurrentLinkedQueue<>();
clq.offer("a");
clq.offer("b");
clq.offer("c");
clq.offer("d");
clq.add("e");

System.out.println(clq.poll()); // a 从头部取出元素，并从队列里删除
System.out.println(clq.size()); // 4
System.out.println(clq.peek()); // b
System.out.println(clq.size()); // 4
```

BlockingQueue 接口

ArrayBlockingQueue：基于数组的阻塞队列实现，在 `ArrayBlockingQueue` 内部，维护了一个定长数组，以便缓存队列中的数据对象，其内部没实现读写分离，也就意味着生产和消费不能完全并行，长度是需要定义的，可以指定先进先出或者先进后出，也叫有界队列，在很多场合非常适合使用。

LinkedBlockingQueue：基于链表的阻塞队列，同 `ArrayBlockingQueue` 类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），`LinkedBlockingQueue` 之所以能够高效的处理并发数据，是因为其内部实现采用分离锁（读写分离两个锁），从而实现生产者和消费者操作的完全并行运行。他是一个无界队列。

SynchronousQueue：一种没有缓冲的队列，生产者产生的数据直接会被消费者获取并消费。

程序根据不同的时间段采用不同的手段处理业务
 6:00~18:00 任务高发期
 18:00 ~ 24:00 任务低发期
 24:00 ~ 6:00 任务非常少

队列
 6:00~18:00 任务高发期 —— `ArrayBlockingQueue`
 18:00 ~ 24:00 任务低发期 —— `LinkedBlockingQueue`
 24:00 ~ 6:00 任务非常少 `SynchronousQueue`

```
package com.bjsxt.chapter12;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class UseQueue {

    public static void main(String[] args) throws Exception {

        // 高性能无阻塞无界队列: ConcurrentLinkedQueue
        ConcurrentLinkedQueue<String> clq = new ConcurrentLinkedQueue<>();
        clq.offer("a");
        clq.offer("b");
        clq.offer("c");
    }
}
```

```

clq.offer("d");
clq.add("e");

System.out.println(clq.size());    // 5
System.out.println(clq.poll());    // a 从头部取出元素，并从队列里删除
System.out.println(clq.size());    // 4
System.out.println(clq.peek());    // b
System.out.println(clq.size());    // 4

System.out.println("-----");

// 阻塞有界队列
ArrayBlockingQueue<String> array = new ArrayBlockingQueue<>(5);
array.put("a");
array.put("b");
array.add("c");
array.add("d");
array.add("e");
// array.add("f");
System.out.println(array.offer("a", 3, TimeUnit.SECONDS));

System.out.println("-----");

// 阻塞无界队列，可声明长度
LinkedBlockingQueue<String> lbq = new LinkedBlockingQueue<>();
lbq.offer("a");
lbq.offer("b");
lbq.offer("c");
lbq.offer("d");
lbq.offer("e");
lbq.add("f");
System.out.println(lbq.size());
lbq.forEach(System.out::println);

System.out.println("-----");

List<String> list = new ArrayList<>();
System.out.println(lbq.drainTo(list, 3));
System.out.println(list.size());
list.forEach(System.out::println);

System.out.println("-----");

```

```
// 无缓冲队列
final SynchronousQueue<String> q = new SynchronousQueue<>();

Thread t1 = new Thread(() -> {
    try {
        System.out.println(q.take());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});
t1.start();

Thread t2 = new Thread(() -> q.add("asdadsd"));
t2.start();
}
```

PriorityBlockingQueue：基于优先级的阻塞队列（优先级的判断通过构造函数传入的 `Compator` 对象来决定，也就是说传入队列的对象必须实现 `Comparable` 接口，在实现 `PriorityBlockingQueue` 时，内部控制线程同步的锁采用的是公平锁，他也是一个无界的队列。

DelayQueue：带有延迟时间的 `Queue`，其中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。`DelayQueue` 中的元素必须实现 `Delayed` 接口，`DelayQueue` 是一个没有大小限制的队列，应用场景很多，比如对缓存超时的数据进行移除、任务超时处理、空闲连接的关闭等等。