> 作者：JavaGieGie
>
> 微信公众号：Java开发零到壹

前言

上一期的蹲坑系列我们介绍了多线程的基础知识，是不是和你平时的了解有些出入呢。

《蹲坑也能进大厂》多线程这几道基础面试题，80%小伙伴第一题就答错

今天继续讲解多线程相对基础的理论知识点，如果你是新手或者对多线程了解不多，千万不要想着上去就肝实战课，没用的，随便出一个bug你都看不出来啥原因，花GIe强烈建议跟着本系列走完（手动狗头护体）。

> 我：哟呼，狗剩子今天怎么不在家修养，也不陪你女朋友，又来公司写bug啊
>
> 狗剩子：嘿嘿，怎么可能不陪女朋友，我们每天形影不离
>
> 我：……
>
> 狗剩子：趁着今天没人，走，坑里见，我要和你坦诚相对。

正文

> **我：狗剩子，请听第一题，守护线程和用户线程有什么区别？**

我们应该知道，Java有两种线程：【守护线程Daemon】与【用户线程User】，两者的唯一的区别就是：虚拟机离开时，如果JVM中所有线程都是守护线程时，JVM就会自动退出；但是如果还有一个或以上的非守护线程则不会退出。

> **我：昨天问过你notify，那你知道Java中notify 和 notifyAll有什么区别？或者说我们怎么选择使用哪一种？**

昨天的事居然还记得，你这记性还可以呀，我都忘记了。

是这样的，`notify`不能指定唤醒某一个具体的线程（这是网上说的，俺就跟着说，至于为啥后面告诉你），可能会导致信号丢失这样的问题，只有在一个线程等待的时候才是它的主场，而`notifyAll`会唤醒所有等待线程，并允许他们争夺锁，虽然效率不高，但是可以保证至少有一个线程继续执行。如果想要使用`notify`，必须确保满足以下两种情况。

- 一次通知仅需要唤醒最多一条线程。
- 所有等待唤醒的线程，自身处理逻辑相同。举个栗子大家就会明白，比如使用Runnable接口实例创建的不同线程，或者同一个Thread子类new出来的多个实例。

> **我：不要得意，这都是开胃菜，再问你一个，wait为什么只能在代码块中使用？**

啥？（心里捣鼓）`wait`只能在代码块中使用吗，我咋不知道。那是....可能`wait`有洁癖，喜欢一个人自嗨吧。

> **我：你踏马....**

哦...我想起来了，我们可以反过来想，如果wait不要求在同步块中，那可能会发生以下的错误。

先看一处用wait、notify实现的线程安全队列的代码：

```
class BlockingQueue {
    Queue<String> buffer = new LinkedList<String>();

    //
    public void give(String data) {
        buffer.add(data);
        notify();                    // Since someone may be waiting in take!
    }

    public String take() throws InterruptedException {
        while (buffer.isEmpty())     // don't use "if" due to spurious wakeups.
            wait();
        return buffer.remove();
    }
}
```

1. 消费者A调用 `take()` ， 此时`buffer.isEmpty()`为true；
2. 消费者A进入while，在调用`wait()` 方法之前, 生产者B调用了一个完整的 `give()` （即buffer.add(data)和notify()）；
3. 之后消费者A调用了 `wait()` ，但是错过了生产者B调用的`notify()`；
4. 如果之后没有别的生产者调用give()方法，消费者A所在线程则会一直等待。

> **我：这波解释我都忍不住给你点个赞，你知道Java中锁是什么吗？**

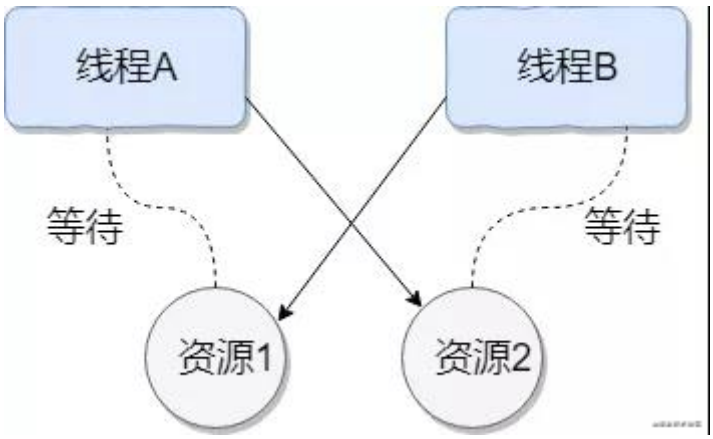
`锁的概念小伙伴们也可以，在学习完花Gie的volatile之后再看。`

`锁`这个东西说起来很抽象, 你可以就把它想象成现实中的锁, 至于他的防盗锁、金锁、还是指纹锁并不重要，哪怕它就是一根草绳，一个自行车、甚至一坨那啥，都可以当做锁。 `锁`是什么外在形象并不重要，重要的是它代表的含义: 谁持有它, 谁就有独立访问临界资源的权利。

> **我：你有了解过什么是线程死锁吗？**

`死锁`就是说两个或两个以上线程在执行过程中，由于竞争资源，它们都在等待某个资源被释放，因此线程被无限期地阻塞，此时的系统处于死锁状态。

如图，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



> **我：那你知道形成死锁的四个必要条件吗？**

咱家有啥不知道的，这就给你列举出来。

1. **互斥条件**：线程（进程）对分配到的资源具有排他性，也就是说该资源任意一个时刻只能有一个进程占用。

2. **请求与保持条件**：一个线程（进程）因请求资源而阻塞时，对已获得的资源保持不放。
3. **不剥夺条件**：线程（进程）已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
4. **循环等待条件**：当发生死锁时，所等待的线程(进程)必定形成一个环路，死循环造成永久堵塞。

> **我：既然你知道形成死锁的条件，那你肯定知道如何避免咯？**

正所谓对症下药，想要`避免死锁`，那就需要破坏者四个必要条件的任意一个即可。

1. **破坏互斥条件**：此路不通，因为我们用锁本来就是想让他们互斥的。
2. **破坏请求与保持条件**：一次性申请所有的资源。
3. **破坏不剥夺条件**：占有资源线程可以尝试申请其它资源，如果申请不到，可以主动释放它占有的资源。
4. **破坏循环等待条件**：按照某一顺序申请资源，释放资源时则反序释放。

> **我：这波回答的不错，昨晚是不是偷偷准备了。狗剩子请继续听题，上下文切换晓得吗？**

麻烦尊重俺一下，以后请叫狗爷。

说到`上下文切换`，那我们得先知道什么是`上下文`，直白说`上下文`就是某个时间点CPU寄存器和程序计数器的内容。

> **拓展**：每个线程都有一个程序计数器（记录要执行的下一条指令），一组寄存器（保存当前线程的工作变量），堆栈（记录执行历史，其中每一帧保存了一个已经调用但未返回的过程）。
>
> **寄存器**：寄存器就是CPU内部内存，负责存储已经、正在和将要执行的任务，数量较少但是速度很快，与之相对应的是CPU外部相对较慢的RAM主内存。
>
> **程序计数器**：程序计数器是一个专用的寄存器，用于表明指令序列CPU当前执行的位置，存储的内容是正在执行指令的位置或下一次将要执行指令的位置。

大致了解了`上下文`，那`上下文切换`也就简单了，它是指当前任务执行完，CPU时间片切换到下一个任务之前会先保存自己的状态，以便下次再切换回这个任务时可以继续执行下去，任务从保存到再加载执行就是一次上下文切换。

如果还不是十分清楚，可以分下面三个步骤理解。

1. 挂起一个进程，将这个进程在CPU中的状态（上下文）存储在内存中；
2. 在内存中检索下一个进程的上下文，并且将该进程在CPU的寄存器中回复；
3. 跳转到程序计数器所指向的位置，也就是该进程被中断时，代码当时执行到的位置，从而恢复该进程。

> **我：讲的好详细，突然好心动，那上下文切换会带来什么问题呢？**

看完上面介绍我们应该有一个感觉，那就是如果高并发情况下，频繁切换上下文会导致系统串行执行，运行速率大大降低。

- **直接消耗**：包括CPU寄存器需要保存和加载, 系统调度器的代码需要执行。
- **间接消耗**：CPU为了加快执行速度，会把常用的数据缓存起来，但是当上下文切换后（即CPU执行不同线程的不同代码），那原本所缓存的内容很大程度没有利用价值了，因此CPU就会重新进行缓存，这也导致线程被调度运行后，一开始启动速度会比较慢。

> **拓展**：线程调度器为了避免频繁切换上下文带来的开销，会给每个被调度到的线程设置一个最小执行时间，从而减少上下文切换的次数，从而提高性能，但是缺点也显而易见，就是会降低响应速度。

> **我：那你跟我讲一下volatile是啥呗？**

不了不了，今天快累死咱家了，老衲需要休息休息，明天我们再战。

总结

多线程知识点非常庞大，涉及到很多方面，特别是刚刚接触多线程的小伙伴，对于`锁`、`上下文`这种概念理解起来非常困难，想要真正全部掌握需要深究每一个问题所涉及到的知识面，比如`怎么用wait/notify实现生产者消费者模式`、`线程的调度过程`、`Java代码如何一步步转化被CPU执行`，还有非常重要的，就是上面这些知识点的原理是什么，`Thread如何启动、中断线程的`，`线程间进行通信的原理是什么`。

这些花GieGie后面都会逐步带大家掌握，有些大的知识点会拿出来进行单篇的讲解，**希望大家持续关注，假日不打样，继续肝。**

点关注，防走丢

以上就是本期全部内容，**如有纰漏之处，请留言指教，非常感谢。**我是花GieGie ，有问题大家随时留言讨论 ，我们下期见👋。

> 文章持续更新，可以微信搜一搜「 **Java开发零到壹** 」第一时间阅读，后续会持续更新Java面试和各类知识点，有兴趣的小伙伴欢迎关注，一起学习，一起哈🐯。

原创不易，你怎忍心白嫖，如果你觉得这篇文章对你有点用的话，感谢老铁为本文**点个赞、评论或转发一下**，因为这将是我输出更多优质文章的动力，感谢！