第10章 并发

2018年7月20日

14:14

**第66条：同步访问共享的可变数据**

为了提高性能而在读或写共享原子数据的时候不使用同步是非常危险和错误的。

为了在线程之间进行可靠的通信，也为了互斥访问，同步是必须的。

不要使用Thread.stop。

当多个线程共享可变数据的时候，每个读或者写数据的线程都必须执行同步。

未同步共享可变数据会造成程序的活性失败和安全性失败。

如果线程间交互通信不需要互斥，volatile修饰符是一种可以接受的同步形式，但是注意它不保证原子性。

**第67条：避免过度同步**

为了避免活性失败和安全性失败，在一个被同步的方法或者代码块中，永远不要放弃对客户端的控制。

如果在内部同步了类，就可以使用不同的方法来实现高并发，例如分拆锁，分离锁和非阻塞锁并发控制。

为了避免死锁和数据破坏，千万不要从同步区域内部调用外来方法。

要限制同步区域内部的工作量。

设计一个可变类的时候，要考虑一下它们是否应该自己完成同步操作。

**第68条：executor和task优先于线程**

|  |  |
| --- | --- |
| 业务场景 | 合适的线程池 |
| 小程序或者轻载的服务器 | Executors.newCachedThreadPool |
| 大负载的服务器 | Executors.newFixedThreadPool |
| 最大限制控制线程 | ThreadPoolExecutor类 |

不仅应该尽量不要编写自己的工作队列，而且还应该尽量不直接使用线程，要使用Executor Famework。

现在关键的抽象不再是Thread,它以前既充当工作单元 ，又是指向机制。

现在的关键抽象是工作但永远，称作任务，分为Runnable及其近亲Callable，执行任务的通用机制是executor service。

ScheduledThreadPoolExecutor可以代替Timer,定时更加准确。

**第69条：并发工具优先于wait和notify**

既然正确地使用wait和notify比较困难，就应该用更高级的并发工具代替。

java.util.concurrent中的高级工具分成三类：

* 并发集合
* 同步器
* 执行器框架

除非不得已，否则应该优先使用ConcurrentHashMap,而不是使用conllections.synchronizedMap或者Hashtable。

对于间歇性的定时，始终应该优先使用System.nanoTime，而不是使用System.currentTimeMills。

没有理由在新代码中使用wait和notity,即使有，也是极少的。

**第70条：线程安全性的文档化**

* 不可变
* 无条件的线程安全
* 有条件的线程安全
* 非线程安全
* 线程对立的

有条件的线程安全类必须在文档中指明“哪个方法调用需要外部同步 ，以及在执行这些序列的时候要获得哪把锁”。

如果编写的是无条件的线程安全类，就应该考虑使用私有锁对象来代替同步的方法。

**第71条：慎用使用延迟初始化**

在大多数情况下，正常的初始化要优先于延迟初始化。

如果出于性能的考虑而需要对静态域使用延迟初始化，就使用lazy initialzation holder class 模式。

对于实例域，就使用双重检查模式。

对于静态域，则使用 lazy initialzation holder class idiom 。

对于可以接受重复初始化的实例域，也可以考虑使用单重检查模式。

**第72条：不要依赖于线程调度器**

让每个线程做些有意义的工作，然后等待更多有意义的工作。如果线程没有在做有意义的工作，就不应该运行。

线程不应该一直处于忙-等的状态，即反复地检查一个共享对象，以等待某些事情的发生。

线程优先级是Java平台上最不可移植的特征。

对于大多数程序员来说，Thread.yield的唯一用途是在测试期间人为地增加程序的并发性。

不要让应用程序的正确性依赖于线程调度器。

不要依赖于Thread.yield或者线程优先级。

**第73条：避免使用线程组**

线程组的并没有提供太多有用的功能。

忽略线程组，就当它们根本不存在一样。