《Java 并发编程实战》笔记

-- Dawei Min

1 简介

之所以在计算机中加入操作系统来实现多个程序的同时执行，主要基于以下原因：资源利用率、公平性、便利性。

2 线程安全性

修复线程不安全的三种方式：

（1）不在线程之间共享该状态变量。

（2）将状态变量修改为不可变的变量。

（3）在访问转态变量时使用同步。

2.1 什么是线程安全性

当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在主调代码中不需要任何额外的同步或协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

无状态对象一定是线程安全的。

2.2 原子性

将“先检查后执行”以及“读取-修改-写入”等操作统称为复合操作：包含了一组必须以原子方式执行的操作以确保线程安全性。

在实际情况中，应尽可能地使用现有的线程安全对象（例如 AtomicLong、AtomicReference）来管理类的状态。

2.3 加锁机制

Java 提供了一种内置的锁机制来支持原子性：同步代码块（Synchronized Block）。

Java 的内置锁相当于一种互斥体（或互斥锁），这意味着最多只有一个线程能持有这种锁。

由于内置锁是可重入的，因此如果某个线程试图获得一个已经由它自己持有的锁，那么这个请求就会成功。“重入”意味着获取锁的操作粒度是“线程”，而不是“调用”。

当实现某个同步策略时，一定不要盲目地为了性能而牺牲简单性（这个可能破坏安全性）。

3 对象的共享

3.1 可见性

为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

在没有同步的情况下，编译器、处理器以及运行时都可能对操作的执行顺序进行一些意向不到的调整。在缺乏足够同步的多线程程序中，要想对内存操作的执行顺序进行判断，几乎无法得出正确的结论。

当读取一个非 volatile 类型的 long 变量时，如果对该变量的读操作和写操作在不同的线程中执行，那么很可能会读取到某个值的高 32 位和另一个值的低 32 位。

加锁的含义不仅仅局限于互斥行为，还包括内存可见性。

volatile 变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取 volatile 类型的变量时总会返回最新写入的值。

volatile 变量是一种比 sychronized 关键字更轻量级的同步机制。但不建议过度依赖 volatile 变量提供的可见性。

volatile 变量的正确使用方式包括：确保它们自身状态的可见性，确保它们所引用对象的状态的可见性，以及标识一些重要的程序生命周期事件的发生（例如，初始化或关闭）。

对于服务器应用程序，无论在开发阶段还是在测试阶段，当启动 JVM 时一定都要指定-server 命令行选项。server 模式的 JVM 将比 client 模式进行更多优化，因此，在开发环境（client 模式的 JVM）中能正确运行的代码可能会在部署环境（server 模式的 JVM）中运行失败。

加锁机制既可以确保可见性又可以确保原子性，而 volatile 变量只能确保可见性。

3.2 发布与溢出

“发布（Publish）”一个对象的意思是指，使对象能够在当前作用域之外的代码中使用。

当某个不该发布的对象被发布时，这种情况就被称为逸出（Escape）。

3.3 线程封闭

如果仅在单线程内访问数据，就不需要同步。这种技术被称为线程封闭（Thread Confinement）。

Ad-hoc 线程封闭是指，维护线程封闭性的职责完全由程序实现来承担。

栈封闭是线程封闭的一种特例，在栈封闭中，只能通过局部变量才能访问对象。

ThreadLocal 提供了 get 与 set 等访问接口或方法，这些方法为每个使用该变量的线程都存有一份独立的副本，因此 get 总是返回由当前执行线程在调用 set 时设置的最新值。

3.4 不变性

不可变对象（Immutable Object）一定是线程安全的。

即使对象中所有的域都是 final 类型的，这个对象也仍然是可变的，因为在 final 类型的域中可以保存对可变对象的引用。

当满足以下条件时，对象才是不可变的：

（1）对象创建以后其状态就不能修改。

（2）对象的所有域都是 final 类型（技术上不一定都要 final,如 String）。

（3）对象是正确创建的（在对象创建期间，this 引用没有溢出）。

在 Java 内存中，final 域还有着特殊的语义。Final 可以确保初始化过程的安全性。

3.5 安全发布

要正确的发布一个对象，对象的引用以及对象的状态必须同时对其他线程可见。一个正确构造的对象可以通过以下方式发布：

（1）在静态初始化函数中初始化一个对象引用。

（2）将对象的引用保存到 volatile 类型的域或者 AtomicReferance 对象中。

（3）将对象的引用保存到某个正确构造对象的 final 类型中。

（4）将对象的引用保存到一个由锁保护的域中。

如果对象从技术上来看是可变的，但其状态在发布后不会再改变，那么把这种对象称为“事实不可变对象（Effectively Immutable Object）”。

在并发程序中使用和共享对象时，可以使用的一些实用的策略，包括：

（1）线程封闭。

（2）只读共享。

（3）线程安全共享（在对象内部实现同步，外部可以安全访问）。

（4）保护对象（通过特定的锁来访问）。

4 对象的组合

4.1 设计线程安全的类

包含多个变量的不变性条件将带来原子性需求：这些相关的变量必须在单个原子操作中进行读取或更新。

4.2 实例封闭

一些基本的容器并非线程安全的，例如 ArrayList 和 HashMap，但类库提供了包装器工厂方法（例如 Collections.synchronizedList 及其类似方法），使得这

些线程安全的类在多线程环境中安全地使用。

封闭机制更易于构造线程安全的类，因为当封闭类的状态时，在分析线程安全性时就无须检查整个程序。

4.3 线程安全性的委托

略

4.4 在现有的线程安全类中添加功能

重用能降低开发工作量、开发风险（因为现有的类都已经通过测试）以及维护成本。

客户端加锁机制与扩展类机制有许多共同点，二者都是将派生类的行为与基类的合在一起，这会破坏封装性。

当为现有的类添加一个原子操作时，有一种更好的方式：组合（Composition）。

4.5 将同步策略文档化

在维护线程安全的时，文档是最强大的（同时也是最未被重发利用的）工具之一。

在文档中说明客户代码需要了解的线程安全性保证，以及代码维护人员需要了解的同步策略。

5 基础构件模块

5.1 同步容器类

如果不想在迭代期间对容器加锁，那么一种替代方法就是“克隆”容器，并在副本上进行迭代。

容器的 hashCode 和 equals 等方法也会间接地执行迭代操作，当容器作为另一个容器的元素或键值时，就会出现这种情况。所有这些间接地迭代操作都可能抛出 ConcurrentModificationException。

5.2 并发容器

通过并发容器来代替同步容器，可以极大地提高伸缩性并降低风险。

虽然可以用 List 来模拟 Queue 的行为，但还需要一个 Queue 的类，因为它能去掉 List 的随机访问需求，从而实现了更高效的并发。

ConcurrentHashMap 使用了一种粒度更细的加锁机制来实现更大程度的共享，这种机制称为分段锁（Lock Striping）。

ConcurrentHashMap 中，由于 size 返回的结果在计算时已经过期了吗，所以 size 返回的值可能是一个近似值而不是精确值。

“写入时复制（Copy-On-Write）”容器的线程安全性在于，只要正确地发布一个事实不可变对象，那么在访问该对象时就不再需要进一步的同步。

仅当迭代操作远远多于修改操作时，才应该使用“写入时复制”容器。这个准则很适合事件通知系统（注册监听器）。

5.3 阻塞队列和生产者 - 消费者模式

阻塞队列提供了可阻塞的 put 和 take 方法，以及支持定时的 offer 和 poll 方法。无界队列永远都不会充满，所以无界队列的 put 方法也永远不会阻塞。

在构建高可靠的应用程序时，有界队列是一种强大的资源管理工具：它们能抑制并防止产生过多的工作项，使应用程序在负荷过载的情况下变得更加健壮。

应该尽早地通过阻塞队列在设计中构建资源管理机制——这件事情做得越早，就越容易。

BlockingQueue 的多种实现：

（1）LinkedBlockingQueue 和 ArrayBlockingQueue 是 FIFO 队列。

（2）PriorityBlockingQueue 是一种按优先级排序的队列。

（3）SynchronousQueue 维护一组线程，可以直接交付工作。

双端队列适用于工作密取（Work Stealing）模式。

5.4 阻塞方法与中断方法

线程可能会阻塞或暂停执行，原因有多种：

（1）等待 I/O 操作结束；

（2）等待获得一个锁；

（3）等待重 Thread.sleep 方法中醒来；

（4）等待另一个线程的计算结果。

5.5 同步工具类

闭锁是一种同步工具类，可以延迟线程的进度直到其到达终止状态。闭锁可以确保某些活动直到其他活动都完成后才继续执行。

CountDownLatch 是一种灵活的闭锁实现，它可以使一个或多个线程等待一组事件发生。

FutureTask 也可以用做闭锁。（FutureTask 实现了 Future 语义，表示一种抽象的可生成结果的计算）。

计数信号量（Counting Semaphore）用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。计数信号量还可以用来实现某种资源池，或者对容器施加边界。

栅栏（Barrier）类似于闭锁，它能阻塞一组线程直到某个事件的发生。栅栏与闭锁的关键区别在于，所有线程必须同时到达栅栏位置，才能继续执行。

CyclicBarrier 可以使一定数量的参与方反复地在栅栏位置汇集，它在并行迭代算法中非常有用：这种算法通常将一个问题拆分成一系列相互独立的子问题。

Exchanger 是一种两方（Two-Party）栅栏，各方在栅栏位置上交换数据。

5.6 构建高效且可伸缩的结果缓存

略

第一部分小结

（1）可变状态时至关重要的（It’s the mutable state,stupid）。

（2）尽量将域声明为 final 类型，除非需要它们是可变的。

（3）不可变对象一定是线程安全的。

（4）封装有助于管理复杂性。

（5）用来保护每个可变变量。

（6）当保护同一个不变性条件中的所有变量时，要使用同一个锁。

（7）在执行复合操作期间，要持有锁。

（8）如果从多个线程中访问同一个可变变量时没有同步机制，那么程序会出问题。

（9）不要故作聪明地推断出不需要使用同步的地方。

（10）在设计过程中考虑线程安全，或者在文档中明确地指出它不是线程安全的。

（11）将通办不成策略文档化。

6 任务执行