# Java并发编程实战

## 第二章 线程安全性

- 如果发现一个状态变量没有使用合适的同步，三种修复方法：
  - 不在线程之间共享该状态变量
  - 将状态变量修改为不可变的变量
  - 在访问状态变量是使用同步
- 设计线程安全的类：
  - 良好的面向对象技术
  - 不可修改性
  - 明细的不可变规范
- 什么是线程安全：

> 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。
- 竞态条件（Race Condition）

> 在并发编程中，由于不恰当的执行时序而出现不正确的结果是一种非常重要的情况，它有一个正式的名字：竞态条件（Race Condition）

- 在实际情况中，应尽可能地使用现有的线程安全对象（例如 AcomicLong）来管理累的状态。
- AtomicLong 是一种替代long类型整数的线程安全类，类似的，AtomicReference是一种替代对象引用的线程安全类。
- **静态** 的 Synchronized 方法以 Class 对象作为锁。

```
synchronized (lock){
  //访问或修改有所保护的共享状态
}
```

- 每个Java对象都可以用作一个实现同步的锁，这些锁被称为 **内置锁**（Intrinsic Lock）或 **监视锁**（Monitor Lock），内置锁相当于一种互斥体（或互斥锁）
- **重入**：当某个线程请求一个由其他线程持有的锁时，发出的请求就会阻塞。然而，由于内置所是可重入的，因为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的粒度是“线程”，而不是“调用”。这也避免了子类覆写父类的 synchronized doSomething() 方法，然后调用 super.doSomething() 出现死锁的情况。

- 锁能是其保护的代码路径以 **串行** 形式来访问，确保状态的一致性。串行访问意味着多个线程依次以独占的方式访问对象，而不是并发访问。
- 每个对象都有一个内置锁，只是为了避免显式地创建锁对象。（糟糕的设计，对象可能会非常大，加锁性能不佳）
- 每个共享的和可变的变量都应该 **只由一个锁** 来保护。
- 一种常见的枷锁约定：将所有的可变状态都封装在对象内部，并通过对象的内置锁对所有访问可变状态的代码路径进行同步。（添加新的方法时忘记使用同步会很容易破坏这种加锁协议）
- 只有被多个线程同时访问的可变数据才需要通过锁来保护。
- 对于每个包含多个变量的不变性条件，其中涉及的所有变量都需要 **同一个锁** 来保护。
- 如果持有锁的时间过长，会带来活跃性或者性能问题。
- 当执行时间较长的计算或者可能无法快速完成的操作时（例如网络 I/O 或者控制台 I/O），一定不要持有锁。
- 尽量将不影响共享状态且执行时间较长的操作从同步代码块中分离出去。
- 局部（位于栈上的）变量，不在在多个线程间共享，因为不需要同步。

# 第三章 对象的共享

- 内存可见性（Memory Visibility）
- **无法确保执行读操作的线程能适时地看到其他线程写入的值。**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。
- **重排序**：在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整。
- 在缺少同步的情况下，Java内存模型允许编译器对操作顺序进行重排序，并将数值缓存在寄存器中，它还允许CPU对操作顺序进行重排序，并将数值缓存在处理器特定的缓存中。这种设计能使JVM充分地利用现代多核处理器的强大性能。
- **最低安全性**（out of thin air safety）：当线程在没有同步的情况下读取变量时，可能会得到一个失效值，但至少这个值是由之前某个线程设置的值，而不是一个随机值。
- 非 volatile 类型的64位数值变量（double 和 long），不保证最低安全性。JVM允许64位的读操作和写操作分解为2个32为的操作。需要声明 volatile，或者用锁保护起来。

- 加锁的含义不仅仅局限于互斥行为，还包括 **内存可见性** 。为了确保所有线程都能看到共享变量的最新值，所有执行读操作或者写操作的线程都必须在 **同一个锁** 上同步。
- 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，因为在读取 volatile 类型的变量时，总会返回最新写入的值。
- 如果在验证正确性时需要对可见性进行 **复杂的判断**，那么就不要使用 volatile 变量。
- 变量的正确使用方法：
  - 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
  - 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）
- 读取 volatile 变量的开销只比读取非 volatile 变量的开销略高一点。
- volatile 的语义不足以确保递增操作（count++）的原子操作。
- **加锁机制既可以确保可见性又可以确保原子性，而 volatile 变量只能保证原子性。**
- 当且仅当满足以下所有条件是，才应该使用 volatile 变量：
  - 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
  - 该变量不会与其他状态变量一起纳入不变性条件中。
  - 在访问变量时不需要加锁。
- **发布**（Publish）一个对象的意思是指，是对象能够在当前作用域之外的代码中使用。
- **逸出**（Escape），当某个不应该发布的对象被发布时，这种情况就被称为逸出。
- 不要在构造过程中使 this 逸出。内部类会隐式包含外部类的 this 引用（闭包）。
- 在构造函数中创建线程并没有错误，但不要立即启动它，而是通过一个 start 或者 initialize 方法来启动它。
- 使用工厂方法来防止 this 引用在构造过程中逸出。
- 线程封闭（Thread Confinement）：如果仅在单线程内部访问数据，就不需要同步。当某个对象封闭在一个线程中时，这种用法将自动实现线程安全性，即使被封闭的对象本身不是线程安全的。（例如Swing和JDBC）
- Java语言及其核心库提供了一些机制来帮助维持线程封闭性，例如局部变量和ThreadLocal类。
- 栈封闭，是线程封闭的一个特例。在栈封闭中，只能通过局部变量才能访问对象。
- 局部变量的一个固有属性就是封闭在执行线程中。它们位于执行线程的栈中，其他线程无法访问这个栈。
- 在维持对象引用的栈封闭性时，需要确保被引用对象不会逸出。

- 维持线程封闭性的一种更规范方法是使用 ThreadLocal。为每个使用该变量的线程都存有一份独立的副本。
- **ThreadLocal 的实现不同于Map。这些特定于线程的值保存在Thread对象中**，当线程终止后，这些值会作为垃圾回收。
- 满足同步需求的另一种方法是使用不可变对象（Immutable Object）。**不可变对象一定是线程安全的。**
- 不可变对象需要满足以下条件：
  - 对象创建以后其状态不能修改。
  - 对象的所有域都是 final 类型。
  - 对象是正确创建的（在对象的创建期间，this 引用没有逸出）。
- 在不可变对象的内部仍然可以使用可变对象来管理它们的状态。
- **final**，在Java内存模型中，final域有着**特殊的语义。**final 域能确保初始化过程的安全性，从而可以不受限制的访问不可变对象，并在共享这些对象时无须同步。
- 不可变对象的内存分配开销，相对于加锁或者垃圾回收的影响，要有性能优势。
- 除非需要更高的可见域，应将所有的域都声明为私有域。
- **除非需要某个域是可变的，否则应将其声明为 final 域**（就像 Scala 中要求不可变变量尽量声明为 val，而不是 var）。
- 任何线程都可以在不需要额外同步的情况下安全地访问不可变对象，即使在发布这些对象时没有使用同步。
- 如果 final 类型的域所指向的是可变对象，那么在访问这些域所指向的对象的状态时仍需要同步。