二

78 一份独家的 Java 并发工具图谱

本课时将提纲挈领的对本专栏的重点进行提炼，对前面 77 个课时的内容进行了整理和梳理，方便你复习前面的内容。如果你正准备面试，没有时间看前面的内容，可以通过本课时把 Java 并发知识体系快速建立起来，发现哪一块知识有薄弱的话，可以有针对性的去回顾那一课时的具体内容。

本专栏总共分为 3 个大模块，分别是模块一：夯实并发基础，模块二：玩转 JUC 并发工具，模块三：深入浅出底层原理，知其所以然。我们就从模块一：夯实并发基础部分开始讲起。

模块一：夯实并发基础

线程基础升华

首先对线程基础进行讲解和升华，在实现多线程上，讲解了为何本质只有 1 种**实现线程**的方法，并对于传统的 2 种或 3 种的说法进行了辨析；同时讲解了应该如何正确的**停止线程**，用 volatile 标记位的停止方法是不够全面的。

然后介绍了线程的 **6 种状态**，即 NEW、RUNNABLE、BLOCKED、WAITING、TIMED_WAITING、TERMINATED，还介绍了转换路径。之后就把目光聚焦到了 **wait、notify/notifyAll、sleep** 相关的方法上，这也是面试中常考的内容，我们讲解了它们的注意事项，包括：

- 为什么 wait 方法必须在 synchronized 保护的同步代码中使用?
- 为什么 wait / notify / notifyAll 被定义在 Object 类中，而 sleep 定义在 Thread 类中?

我们还把 wait / notify 和 sleep 进行了比较，并分析它们的异同。之后我们用三种方式实现了**生产者和消费者模式**，分别是 wait / notify、Condition、BlockingQueue 的方式，并对它们进行了对比。

线程安全

在线程安全的相关课时中，首先讲解了**什么是线程安全**，线程**不安全的场景**包括运行结果错误、发布或初始化错误以及活跃性问题，而活跃性问题又包括死锁、活锁和饥饿。

然后总结了 4 种特别需要**注意线程安全的情况**，分别是：

- 有操作共享资源或变量的时候；
- 依赖时序的操作；
- 不同数据之间存在绑定关系；
- 使用的类没有声明自己是线程安全的。

之后，讲解了多线程所带来的**性能问题**，包括线程调度所产生的上下文切换和缓存失效，以及线程协作带来的开销。

模块二：玩转 JUC 并发工具

线程池

下面进入模块二：玩转 JUC 并发工具的部分，在线程池部分中我们首先给出了 3 点使用**线程池**的原因，也就是说，使用线程池比手动创建线程好的地方在于：

- 可以解决线程生命周期的系统开销问题，同时还可以加快响应速度；
- 可以统筹内存和 CPU 的使用，避免资源使用不当；
- 可以统一管理资源。

在了解了线程池的好处之后，就需要掌握线程池的**各个参数**的含义，即 corePoolSize、maxPoolSize、keepAliveTime、workQueue、ThreadFactory、Handler，并且这也是**面试中非常常见的考点**，我们需要知道每个参数代表什么含义。

而线程池也可能会**拒绝**我们提交的任务，我们讲解了 2 种拒绝的时机以及 4 种拒绝的策略，分别是 AbortPolicy、DiscardPolicy、DiscardOldestPolicy、CallerRunsPolicy，我们可以根据自己的业务需求去选择合适的拒绝策略。

之后介绍了 **6 种常见的线程池**，即 FixedThreadPool、CachedThreadPool、ScheduledThreadPool、SingleThreadExecutor、SingleThreadScheduledExecutor 和 ForkJoinPool，这 6 种线程池各有各的特点，它们所采用的的参数也各不相同。

接下来介绍了**阻塞队列**，在线程池中比较常用的是 3 种阻塞队列，即 LinkedBlockingQueue、SynchronousQueue、DelayedWorkQueue。然后讲解了为什么不应该自动创建线程池，主要原因是考虑到自动创建的线程池可能会发生 OOM 等风险，我们

手动创建线程池，就可以更加明确其运行规则，也可以在必要的时候拒绝新的任务提交，所以是更加安全的。

既然说到要手动去创建线程，那怎么设置线程池的参数呢？这里就需要考虑到**合适的线程数量**是多少，我们给出了一个通用的建议：

- 线程的平均工作时间所占比例越高，则需要越少的线程；
- 线程的平均等待时间所占比例越高，则需要越多的线程；
- 针对不同的程序，进行对应的压力测试就可以得到最合适的线程数。

最后讲解了如何**关闭线程池**，讲解了和关闭线程池相关的 5 个方法，即 shutdown()、isShutdown()、isTerminated()、awaitTermination()、shutdownNow() 。其中的重点是 **shutdown() 和 shutdownNow()** 这两个方法的区别，前一个是优雅关闭，后一个则是立刻关闭。接着还对线程池实现“线程复用”的原理进行了讲解，同时分析了 **execute 方法的源码，这是线程池中一个非常重要的方法。**

各种各样的“锁”

在 Java 中，锁有很多种类，比如**悲观锁和乐观锁、共享锁和独占锁、公平锁和非公平锁、可重入锁和非可重入锁、可中断锁和不可中断锁、自旋锁和非自旋锁、偏斜锁/轻量级锁/重量级锁**等。关于悲观锁和乐观锁，我们分析了它们各自的使用场景，还对 synchronized 这种悲观锁分析了原理，看到了其背后的 “monitor ” 锁，然后对 synchronized 和 Lock 进行了比较，并且给出了选择建议：

如果可以，最好既不使用 Lock 也不使用 synchronized，而是优先使用 JUC 包中其他的成熟工具，因为它们通常会帮我们自动处理所有的加锁和解锁操作；如果必须使用锁，则优先使用 synchronized，因为它可以减少代码编写的数量以及降低出错的概率，因为一旦使用 Lock，就必须在 finally 中写上 unlock，不然代码可能会出很大的问题，而使用 synchronized 就不必考虑这些问题，因为它会自动解锁。当然如果 synchronized 不能满足我们的需求，就得考虑使用 Lock。

所以接下来就是 Lock 相关的内容，它有很多强大的功能，比如尝试获取锁、有超时的获取等。我们介绍了 lock() 、tryLock()、tryLock(long time, TimeUnit unit)、lockInterruptibly()、unlock() 这几个常用的方法，并且讲解了它们的作用。然后讲解了**公平锁和非公平锁**，其中公平锁会按照线程申请锁的顺序来依次获取锁，而非公平锁存在插队的情况，这在一定情况下可以提高整体的效率，通常默认也是非公平的。

接着是读写锁内容。ReadWriteLock 适用于读多写少的情况，合理使用可以进一步提高并发效率，它的规则是：**要么是一个或多个线程同时持有读锁，要么是一个线程持有写锁**，但两者不会同时出现。也可以总结为读读共享、其他都互斥（包括写写互斥、读写互斥、写读

互斥）。之后还讲解了读写锁的升降级和插队策略。

对于自旋锁而言，首先介绍了什么是自旋锁，然后对比了自旋和非自旋锁的获取锁的过程，讲解了自旋锁的好处，然后自己实现了一个可重入的自旋锁，最后还分析了自旋锁的缺点和适用场景。

在锁的内容中，最后还讲解了 **JVM 对锁进行的优化**点，包括自适应的自旋锁、锁消除、锁粗化、偏向锁、轻量级锁、重量级锁等。有了这些优化点之后，synchronized 的性能并不比其他的锁差，所以我们使用 synchronized 来满足业务条件在性能方面是完全 OK 的。

并发容器面面观

并发容器是一个重点。在并发容器的章节中，首先讲解了 HashMap 为什么是线程不安全的，然后对比了 **ConcurrentHashMap** 在 Java 7 和 8 中的区别，包括数据结构、并发度、保证并发安全的原理、遇到 Hash 碰撞、查询时间复杂度方面的区别。然后还分析了在 Map 桶中为什么超过 8 个才转为红黑树？ 这是一种时间和空间上的平衡，以及对比了 ConcurrentHashMap 和 Hashtable，虽然它们都是线程安全的，但在出现版本上、实现线程安全的方式上、性能上、迭代时修改上都是不同的。

接着介绍了 CopyOnWriteArrayList，它的适用场景是读操作可以尽可能的快，而写即使慢一些也没关系，以及读多写少的场景。CopyOnWriteArrayList 的读写规则是读取完全不用加锁，而写入也不会阻塞读取操作，也就是可以在写入的同时进行读取，只有写入和写入之间需要进行同步，不允许多个写入同时发生。之后还介绍了它的允许迭代时修改集合内容的特点以及 3 个缺点，分别是内存占用问题、在元素较多或者复杂的情况下，复制开销大的问题以及数据一致性问题，最后我们还对它的源码进行了分析。

阻塞队列

在并发容器里还有一个重点，那就是**阻塞队列**，首先介绍了什么是阻塞队列以及对于阻塞队列中的 3 组方法进行了辨析，同时还给出了代码演示。然后分别介绍了常见的 5 种阻塞队列，以及它们的特点，分别是 ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue 和 DelayQueue。

之后对比了阻塞和非阻塞队列的并发安全原理，其中阻塞队列主要利用了 ReentrantLock 以及它的 Condition 来实现的，而非阻塞队列则是利用了 CAS 保证线程安全。

最后，我们讲解了如何选择适合自己的阻塞队列，需要从功能、容量、能否扩容、内存结构及性能这些方面去考虑综合选择适合自己的阻塞队列。

原子类

原子类是 JUC 包中的一个重量级的人物。首先介绍了 6 种原子类型，即基本类型原子类、数组类型原子类、引用类型原子类、升级类型原子类、Adder 和 Accumulator。

接下来分析了 **AtomicInteger** 在高并发下性能不好以及如何解决的问题。性能不好的主要原因是在高并发下碰撞和冲突会比较多，我们可以使用 LongAdder 来解决这个问题；同时分析了 **LongAdder** 内部的原理。然后对比了原子类和 volatile ，如果只是有可见性问题的话，那么可以使用 volatile 来解决，但如果需要保证原子性的话，就需要使用原子类或其他工具来解决，而不应使用 volatile。

之后，我们把原子类和 synchronized 进行了对比，它们在功能上相似，但是在原理上、适用范围上、粒度上、性能上都有区别。最后还介绍了 Java 8 加入的 Accumulator，它是一个更通用版本的 Adder。

ThreadLocal

首先讲解了两种场景是适合于 ThreadLocal 的：

- 第一种是用作每个线程保存独享的对象，比如日期工具类；
- 第二种是 ThreadLocal 给每个线程去保存场景、上下文信息，以便后续的方法更方便的获取其信息，避免了传参。

当然 ThreadLocal 并不是用来解决共享资源的多线程访问的问题的，因为它设计的本意是，资源并不是共享的，只是在每个线程内有个资源的副本而已，而每个副本都是各线程独享的。

接下来还分析了 ThreadLocal 的内部结构，需要掌握 **Thread、ThreadLocal 及 ThreadLocalMap 三者之间的关系**，同时还介绍了使用 ThreadLocal 之后要使用 remove 方法来防止内存泄漏。

Future

接下来是 Future 相关的内容。首先对比了 Callable 和 Runnable 的不同，它们在方法名、返回值、抛出异常上，以及和 Future 类的关系上都有所不同。然后介绍了 Future 类的主要功能，即把运算的过程放到子线程去执行，再通过 Future 去控制执行过程，最后获取到计算结果。这样一来就可以把整个程序的运行效率提高，是一种**异步**的思想。

我们还对 Future 的 get、get(long timeout, TimeUnit unit)、isDone()、cancel()、isCancelled() 这 5 种方法进行了详细讲解。在使用 Future 的时候要注意，比如我们用 for 循环批量获取 Future 的结果时容易阻塞，应该使用超时限制，并且 Future 的生命周期不能后退，而且 Future 本身并不能产生新的线程，它需要借助 Thread 类或者线程池才能用子

线程执行任务。

之后讲解了一个“旅游平台”的问题，它希望高效获取各航空公司的机票信息，我们对代码进行了演进：从最开始的串行，到并行，然后到有超时的并行，最后我们发现，而且如果航空公司的响应速度都很快的话，也不需要一直等到超时的时间到了，而是可以提前结束等待的。我们就这样进行了一步一步的迭代，升级了我们的代码，该“旅游平台”问题也是平时工作中经常会遇到的问题，因为我们经常需要并行获取和处理数据。

线程协作

在线程配合相关的类中，我们讲解了 **Semaphore 信号量、CountDownLatch、CyclicBarriar 和 Condition。**

在信号量的课程中，首先介绍了它的使用场景、用法及注意点，其中注意点包括获取和释放的许可证数量尽量保持一致，在初始化的时候可以设置公平性以及信号量是支持跨线程、跨线程池的。

对于 CountDownLatch 而言，我们在创建类的时候，需要在构造函数中传入“倒数”次数，然后由需要等待的线程去调用 await 方法来等待，而每一次其他线程调用了 countDown 方法之后，计数便会减 1，直到减为 0 时，之前等待的线程便会继续运行。

接下来介绍了 CyclicBarria，它和 CountDownLatch 在用法上是有些相似的，即都能阻塞一个或一组线程，直到某个预设的条件达成发生，再统一出发，但它们也有很多的不同点，它们的作用对象不同、可重用性不同及执行动作的能力不同。

最后介绍了 Condition 和 wait / notify / notifyAll 的关系。如果说 Lock 是用来代替 synchronized 的，那么 Condition 就是用来代替相对应的 Object 的 wait / notify / notifyAll 的，所以它们在用法和性质上都是非常相似的。

模块三：深入浅出底层原理，知其所以然

Java 内存模型

然后就进入到了我们第 3 个模块：深入浅出底层原理，知其所以然。第一个重点是 **Java 内存模型。**首先介绍了为什么需要 Java 内存模型，然后介绍了什么是 Java 内存模型，重点包括重排序、原子性、可见性。

接着首先介绍了**重排序**的相关内容，其好处是可以提高处理速度。

接着介绍了**原子性**，包括什么是原子性、Java 中的原子操作有哪些、long 和 double 原子

性的特殊性以及简单地把原子操作组合在一起，并不能保证整体依然具备原子性。

之后讲解了**可见性**，我们需要知道主内存和工作内存之间的关系，还需要知道 **happens-before** 关系：如果第一个操作 happens-before 第二个操作（也可以描述为第一个操作和第二个操作之间满足 happens-before 关系），那么我们就说第一个操作对于第二个操作一定是可见的，也就是第二个操作在执行时就一定能保证看见第一个操作执行的结果。**这个关系非常重要，也是可见性内容的一个重点。**

最后介绍了 volatile 的两个作用，分别是保证可见性以及一定程度上禁止重排序，还分析了在单例模式的双重检查锁模式为什么必须加 volatile？主要是为了保证线程安全。

CAS 原理

在 CAS （Compare-And-Swap）相关课时中，首先介绍了 CAS 的核心思想，是通过将内存中的值与指定数据进行比较，当这两个数值一样时，才将内存中的数据替换为新的值，整个过程具备原子性。

然后介绍了 CAS 的应用，包括在并发容器、数据库以及原子类中都有很多对 CAS 的应用；之后介绍了 CAS 的三个缺点，即 ABA 问题、自旋时间过长问题，以及线程安全的范围不能灵活控制问题。

死锁问题

在死锁的相关课时中，首先介绍了什么是死锁：两个或多个线程（或进程）被无限期地阻塞，相互等待对方手中资源的状态就是死锁。我们写了必然死锁的例子，介绍了发生死锁必须满足的互斥条件、请求与保持条件、不剥夺条件和循环等待条件这 4 个必要条件，还分别用命令行和代码定位死锁并且给出了 3 种解决死锁问题的策略，分别是避免策略、检测与恢复策略、鸵鸟策略。最后还分析了经典的哲学家就餐问题。

final 关键字和“不变性”

首先介绍了 final 分别作用在**变量上、方法上和类上**的不同作用，以及分析了为什么加了 final 却依然无法拥有“不变性”？主要原因是 final 修饰的对象，内容依然可以变。然后分析了为什么 String 被设计为是不可变的？主要分析了这样设计的好处分别是可以利用字符串常量池、用作 HashMap 的 key、缓存 HashCode 以及保证线程安全。

AQS 框架

最后是 AQS 的内容，我们介绍了为什么需要 AQS 以及它内部的原理；还对 AQS 在 CountDownLatch 类中的应用进行了源码分析。

总结

以上就是本专栏的重点内容了，也涵盖到了 Java 并发编程的大部分重点知识。我也非常高兴能和你一起来学习和探讨关于 Java 并发的知识，在写作的过程中难免会有遗漏的知识点，可通过留言，或联系拉勾客服人员加入本课程的读者群一起讨论。

上一页