Java JUC

讲师：李贺飞

主要内容

1. Java JUC 简介
2. volatile 关键字-内存可见性
3. 原子变量-CAS算法
4. ConcurrentHashMap 锁分段机制
5. CountDownLatch 闭锁
6. 实现 Callable 接口
7. Lock 同步锁
8. Condition 控制线程通信
9. 线程按序交替
10. ReadWriteLock 读写锁
11. 线程八锁
12. 线程池
13. 线程调度
14. ForkJoinPool 分支/合并框架 工作窃取

Java JUC 简介

- 在 Java 5.0 提供了 **java.util.concurrent** （简称 JUC ）包，在此包中增加了在并发编程中很常用的实用工具类，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。提供可调的、灵活的线程池。还提供了设计用于多线程上下文中的 Collection 实现等。

1-volatile 关键字 内存可见性

内存可见性

- 内存可见性（Memory Visibility）是指当某个线程正在使用对象状态而另一个线程在同时修改该状态，需要确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。
- 可见性错误是指当读操作与写操作在不同的线程中执行时，我们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，有时甚至是根本不可能的事情。
- 我们可以通过同步来保证对象被安全地发布。除此之外我们也可以使用一种更加轻量级的 volatile 变量。

volatile 关键字

- Java 提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他线程。

  可以将 volatile 看做一个轻量级的锁，但是又与锁有些不同：

  - 对于多线程，不是一种互斥关系
  - 不能保证变量状态的“原子性操作”

2-原子变量 CAS算法

CAS 算法

- CAS (Compare-And-Swap) 是一种硬件对并发的支持，针对多处理器操作而设计的处理器中的一种特殊指令，用于管理对共享数据的并发访问。
- CAS 是一种无锁的非阻塞算法的实现。
- CAS 包含了 3 个操作数：
  - 需要读写的内存值 V
  - 进行比较的值 A
  - 拟写入的新值 B
- 当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作。

原子变量

- 类的小工具包，支持在单个变量上解除锁的线程安全编程。事实上，此包中的类可将 volatile 值、字段和数组元素的概念扩展到那些也提供原子条件更新操作的类。
- 类 AtomicBoolean、AtomicInteger、AtomicLong 和 AtomicReference 的实例各自提供对相应类型单个变量的访问和更新。每个类也为该类型提供适当的实用工具方法。
- AtomicIntegerArray、AtomicLongArray 和 AtomicReferenceArray 类进一步扩展了原子操作，对这些类型的数组提供了支持。这些类在为其数组元素提供 volatile 访问语义方面也引人注目，这对于普通数组来说是不受支持的。
- **核心方法：boolean compareAndSet(expectedValue, updateValue)**
- java.util.concurrent.atomic 包下提供了一些原子操作的常用类:
  - AtomicBoolean 、AtomicInteger 、AtomicLong 、 AtomicReference
  - AtomicIntegerArray 、AtomicLongArray
  - AtomicMarkableReference
  - AtomicReferenceArray
  - AtomicStampedReference

3-ConcurrentHashMap 锁分段机制

ConcurrentHashMap

- Java 5.0 在 java.util.concurrent 包中提供了多种并发容器类来改进同步容器的性能。
- ConcurrentHashMap 同步容器类是Java 5 增加的一个线程安全的哈希表。对与多线程的操作，介于 HashMap 与 Hashtable 之间。内部采用“锁分段”机制替代 Hashtable 的独占锁。进而提高性能。
- 此包还提供了设计用于多线程上下文中的 Collection 实现：ConcurrentHashMap、ConcurrentSkipListMap、ConcurrentSkipListSet、CopyOnWriteArrayList 和 CopyOnWriteArraySet。当期望许多线程访问一个给定 collection 时，ConcurrentHashMap 通常优于同步的 HashMap，ConcurrentSkipListMap 通常优于同步的 TreeMap。当期望的读数和遍历远远大于列表的更新数时，CopyOnWriteArrayList 优于同步的 ArrayList。

4-CountDownLatch 闭锁

CountDownLatch

- Java 5.0 在 java.util.concurrent 包中提供了多种并发容器类来改进同步容器的性能。
- CountDownLatch 一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。
- 闭锁可以延迟线程的进度直到其到达终止状态，闭锁可以用来确保某些活动直到其他活动都完成才继续执行：
  - 确保某个计算在其需要的所有资源都被初始化之后才继续执行;
  - 确保某个服务在其依赖的所有其他服务都已经启动之后才启动;
  - 等待直到某个操作所有参与者都准备就绪再继续执行。

5-实现 Callable 接口

Callable 接口

- Java 5.0 在 java.util.concurrent 提供了一个新的创建执行线程的方式：Callable 接口
- Callable 接口类似于 Runnable，两者都是为那些其实例可能被另一个线程执行的类设计的。但是 Runnable 不会返回结果，并且无法抛出经过检查的异常。
- Callable 需要依赖FutureTask，FutureTask 也可以用作闭锁。

6-Lock 同步锁

显示锁 Lock

- 在 Java 5.0 之前，协调共享对象的访问时可以使用的机制只有 synchronized 和 volatile 。Java 5.0 后增加了一些新的机制，但并不是一种替代内置锁的方法，而是当内置锁不适用时，作为一种可选择的高级功能。
- ReentrantLock 实现了 Lock 接口，并提供了与 synchronized 相同的互斥性和内存可见性。但相较于 synchronized 提供了更高的处理锁的灵活性。

7-Condition 控制线程通信

Condition

- Condition 接口描述了可能会与锁有关联的条件变量。这些变量在用法上与使用 Object.wait 访问的隐式监视器类似，但提供了更强大的功能。需要特别指出的是，单个 Lock 可能与多个 Condition 对象关联。为了避免兼容性问题，Condition 方法的名称与对应的 Object 版本中的不同。
- 在 Condition 对象中，与 wait、notify 和 notifyAll 方法对应的分别是 await、signal 和 signalAll。
- Condition 实例实质上被绑定到一个锁上。要为特定 Lock 实例获得 Condition 实例，请使用其 newCondition() 方法。

8-线程按序交替

线程按序交替

- 编写一个程序，开启 3 个线程，这三个线程的 ID 分别为 A、B、C，每个线程将自己的 ID 在屏幕上打印 10 遍，要求输出的结果必须按顺序显示。

  如：ABCABCABC…… 依次递归

9-ReadWriteLock 读写锁

读-写锁 ReadWriteLock

- ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。。
- ReadWriteLock 读取操作通常不会改变共享资源，但执行写入操作时，必须独占方式来获取锁。对于读取操作占多数的数据结构。 ReadWriteLock 能提供比独占锁更高的并发性。而对于只读的数据结构，其中包含的不变性可以完全不需要考虑加锁操作。

10-线程八锁

线程八锁

- 一个对象里面如果有多个synchronized方法，某一个时刻内，只要一个线程去调用其中的一个synchronized方法了，其它的线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些synchronized方法
- 锁的是当前对象this，被锁定后，其它的线程都不能进入到当前对象的其它的synchronized方法
- 加个普通方法后发现和同步锁无关
- 换成两个对象后，不是同一把锁了，情况立刻变化。
- 都换成静态同步方法后，情况又变化
- 所有的非静态同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。
- 所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

11-线程池

线程池

- 第四种获取线程的方法：线程池，一个 ExecutorService，它使用可能的几个池线程之一执行每个提交的任务，通常使用 Executors 工厂方法配置。
- 线程池可以解决两个不同问题：由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源（包括执行任务集时使用的线程）的方法。每个 ThreadPoolExecutor 还维护着一些基本的统计数据，如完成的任务数。
- 为了便于跨大量上下文使用，此类提供了很多可调整的参数和扩展钩子 (hook)。但是，强烈建议程序员使用较为方便的 Executors 工厂方法：
  - Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）
  - Executors.newFixedThreadPool(int)（固定大小线程池）
  - Executors.newSingleThreadExecutor()（单个后台线程）

  它们均为大多数使用场景预定义了设置。

12-线程调度

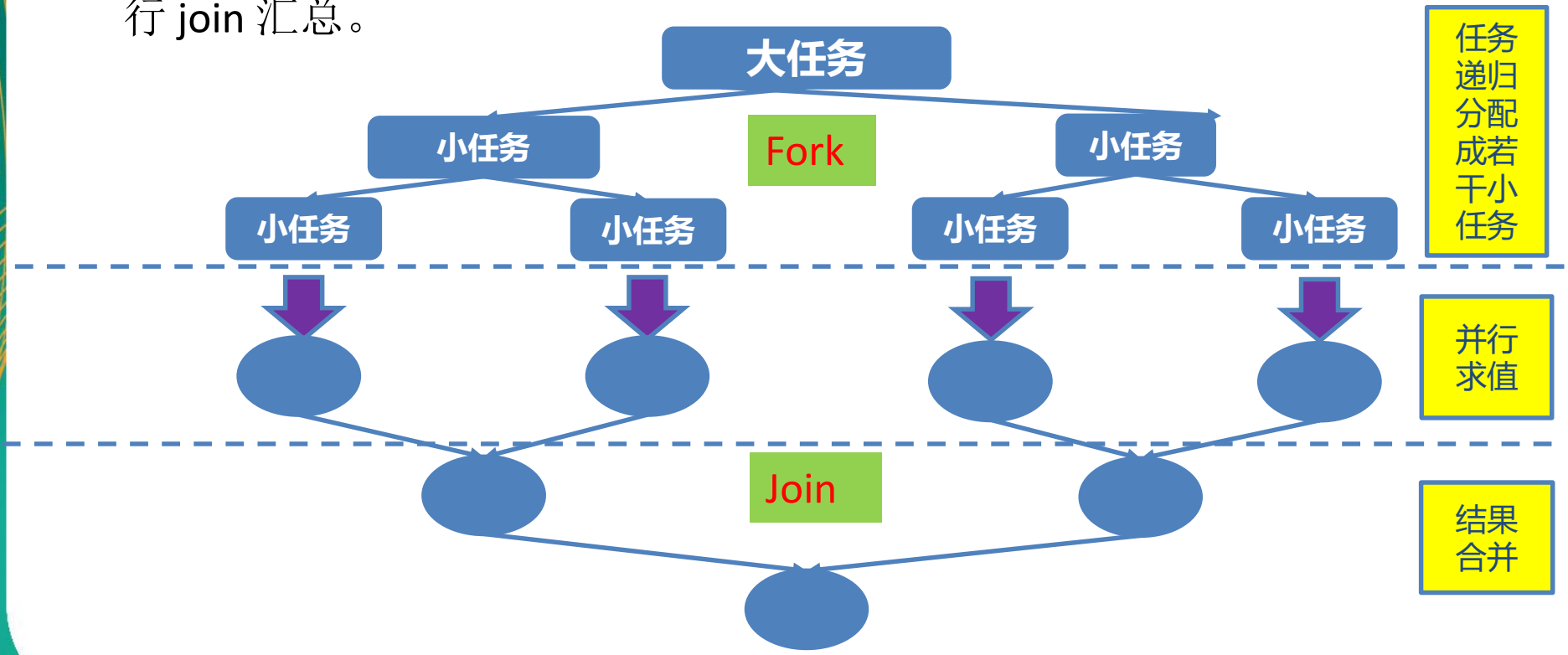
ScheduledExecutorService

- 一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

13-ForkJoinPool 分支/合并框架 工作窃取

Fork/Join 框架

- Fork/Join 框架：就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行 join 汇总。



Fork/Join 框架与线程池的区别

- 采用 “工作窃取”模式（work-stealing）：

当执行新的任务时它可以将其拆分分成更小的任务执行，并将小任务加到线程队列中，然后再从一个随机线程的队列中偷一个并把它放在自己的队列中。

- 相对于一般的线程池实现，fork/join框架的优势体现在对其中包含的任务的处理方式上.在一般的线程池中，如果一个线程正在执行的任务由于某些原因无法继续运行，那么该线程会处于等待状态。而在fork/join框架实现中，如果某个子问题由于等待另外一个子问题的完成而无法继续运行。那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行.这种方式减少了线程的等待时间，提高了性能。


尚硅谷
www.atguigu.com