并发编程精通篇

尹洪亮 | Kevin.Yin

互联网架构师 / 自由讲师

每天都要让自己比别人多努力一分钟

Kevin、让你每天进步一点点

我的微信 liang19871023liang

加微信，获取**项目源码、高清课件**

加微信，所有**新课七折**优惠，职业规划，互动答疑，免费资料

加微信，受邀进入**KEVIN社区**，与大咖和同龄人会面

关注公众号，每周推送**Kevin原创文章**，不定期优惠活动

课程内容介绍

- Concurrent同步工具类
  - CountDownLatch
  - CyclicBarrier
  - Semaphore
  - Exchanger
  - ReentrantLock
  - ReentrantReadWriteLock
- 四种线程池与自定义线程池、底层代码
- 设计模式：单例、Future、Master-Worker、Producer- Consumer模式原理和实现

CountDownLatch

- CountDownLatch 是一个辅助工具类，它允许一个或多个线程等待一系列指定操作的完成。CountDownLatch 以一个给定的数量初始化。countDown() 每被调用一次，这一数量就减一。通过调用 await() 方法之一，线程可以阻塞等待这一数量到达零。

- 示例：CountDownLatchTest1、CountDownLatchTest2

CyclicBarrier

- CyclicBarrier一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点(common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。

- 需要所有的子任务都完成时，才执行主任务，这个时候就可以选择使用CyclicBarrier。

- 示例：CyclicBarrierTest1、CyclicBarrierTest2

Semaphore

- Semaphore一个计数信号量。信号量维护了一个许可集合; 通过acquire()和release()来获取和释放访问许可证。只有通过acquire获取了许可证的线程才能执行,否则阻塞。通过release释放许可证其他线程才能进行获取。

- 公平性：没有办法保证线程能够公平地可从信号量中获得许可。也就是说，无法担保掉第一个调用 acquire() 的线程会是第一个获得一个许可的线程。如果第一个线程在等待一个许可时发生阻塞，而第二个线程前来索要一个许可的时候刚好有一个许可被释放出来，那么它就可能会在第一个线程之前获得许可。如果你想要强制公平，Semaphore 类有一个具有一个布尔类型的参数的构造子，通过这个参数以告知 Semaphore 是否要强制公平。强制公平会影响到并发性能，所以除非你确实需要它否则不要启用它。

- 示例： SemaphoreTest1

Exchanger

- Exchanger Exchanger 类表示一种两个线程可以进行互相交换对象的会和点。
- 只能用于两个线程之间，并且两个线程必须都到达汇合点才会进行数据交换

- 示例： ExchangerTest1

ReentrantLock

- ReentrantLock可以用来替代Synchronized，在需要同步的代码块加上锁，最后一定要释放锁，否则其他线程永远进不来。
- 示例：com.mimaxueyuan.demo.high.lock1.ReentrantLockTest1

- 可以使用Condition来替换wait和notify来进行线程间的通讯，Condition只针对某一把锁。
- 示例：com.mimaxueyuan.demo.high.lock1.ConditionTest1

- 一个Lock可以创建多个Condition，更加灵活
- 示例：com.mimaxueyuan.demo.high.lock1.ConditionTest2

ReentrantLock

- ReentrantLock的构造函数可以如传入一个boolean参数，用来指定公平/非公平模式，默认是false非公平的。非公平的效率更高。

- Lock的其他方法：
  - tryLock()：尝试获得锁，返回true/false
  - tryLock(timeout, unit)：在给定的时间内尝试获得锁
  - isFair()：是否为公平锁
  - isLocked()：当前线程是否持有锁
  - lock.getHoldCount()：持有锁的数量，只能在当前调用线程内部使用，不能再其他线程中使用
  - 示例：com.mimaxueyuan.demo.high.lock1.HoldCountTest

ReentrantReadWriteLock

- ReentrantReadWriteLock读写所，采用读写分离机制，高并发下读多写少时性能优于ReentrantLock。
- 读读共享，写写互斥，读写互斥

- 示例：com.mimaxueyuan.demo.high.lock2.ReadWriteLockTest1

四种线程池

- newCachedThreadPool 具有缓存性质的线程池,线程最大空闲时间60s,线程可重复利用(缓存特性),没有最大线程数限制。任务耗时端，数量大。
- newFixedThreadPool 具有固定数量的线程池,核心线程数等于最大线程数,线程最大空闲时间为0,执行完毕即销毁,超出最大线程数进行等待。高并发下控制性能。
- newScheduledThreadPool 具有时间调度特性的线程池,必须初始化核心线程数,底层使用DelayedWorkQueue实现延迟特性。
- newSingleThreadExecutor 核心线程数与最大线程数均为1,用于不需要并发顺序执行。

- 示例： CachedThreadPoolTest、 FixedThreadPoolTest
- ScheduledThreadPoolTest、 SingleThreadExecutorTest

ThreadPoolExecutor

- 四种线程池都是通过Executors类创建的, 底层创建的都是ThreadPoolExecutor类, 可以构建自己需要的线程类。

- 示例：ThreadPoolExecutorTest

设计模式-单例模式

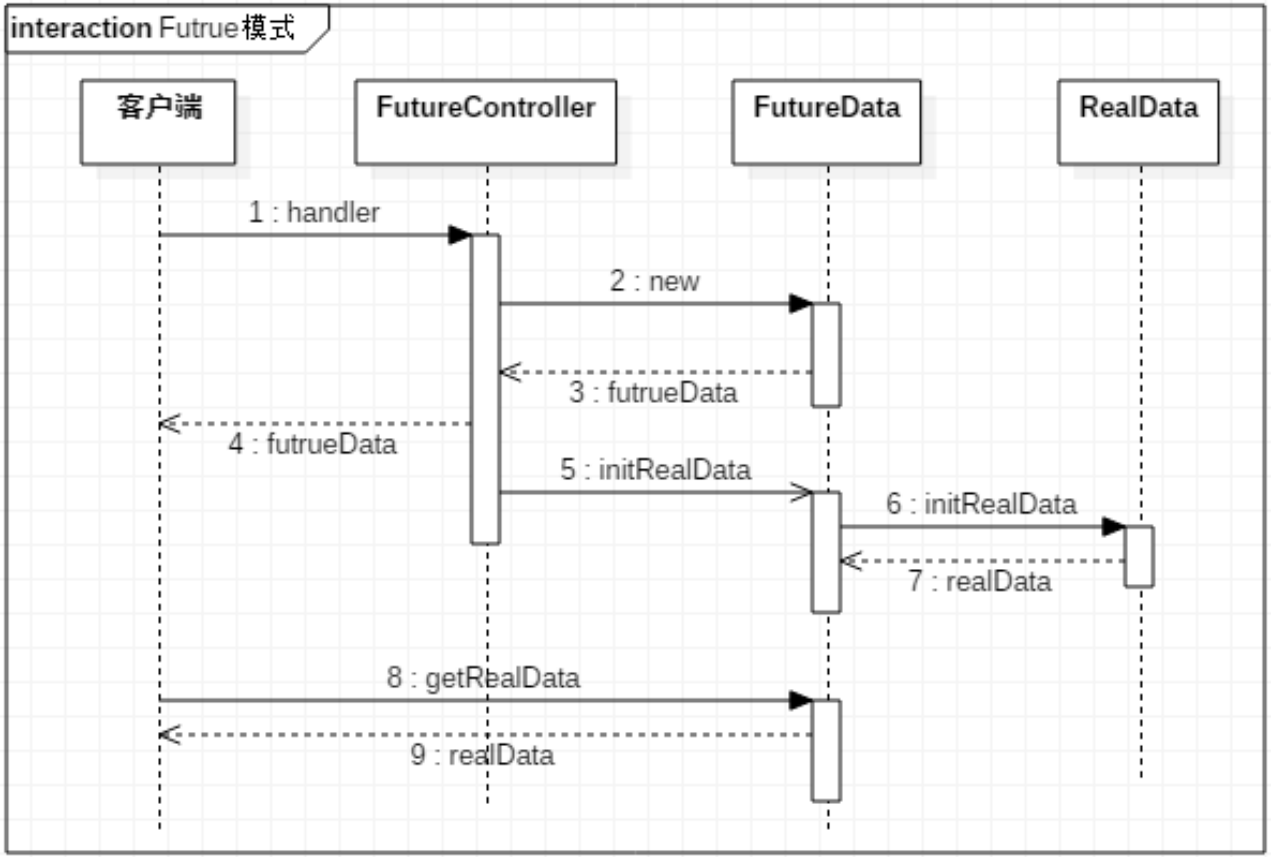
- 饿汉模式：类加载的时候，就进行对象的创建，系统开销较大，但是不存在线程安全问题
- 懒汉模式：多数采用饿汉模式，在使用时才真正的创建单例对象，但是存在线程安全问题
- 静态内部类单例：兼具懒汉模式和饿汉模式的优点
- 饿汉示例： DemoThread22
- 懒汉示例： DemoThread23 （线程安全问题和解决方案）
- 懒汉模式： DemoThread24 （线程安全的性能优化）
- 静态内部类单例： DemoThread25

设计模式-Future

- 简单来说,客户端请求之后，先返回一个应答结果，然后异步的去准备数据，客户端可以先去处理其他事情，当需要最终结果的时候再来获取, 如果此时数据已经准备好，则将真实数据返回；如果此时数据还没有准备好，则阻塞等待。

- 示例： com.mimaxueyuan.demo.high.futrue.Main

设计模式-Future



设计模式-Future

- JDK的Concurrent包提供了Futrue模式的实现，可以直接使用。
- 使用Futrue模式需要实现Callable接口，并使用FutureTask进行封装，使用线程池进行提交。
- 示例： com.mimaxueyuan.demo.high.futrue.JdkFuture

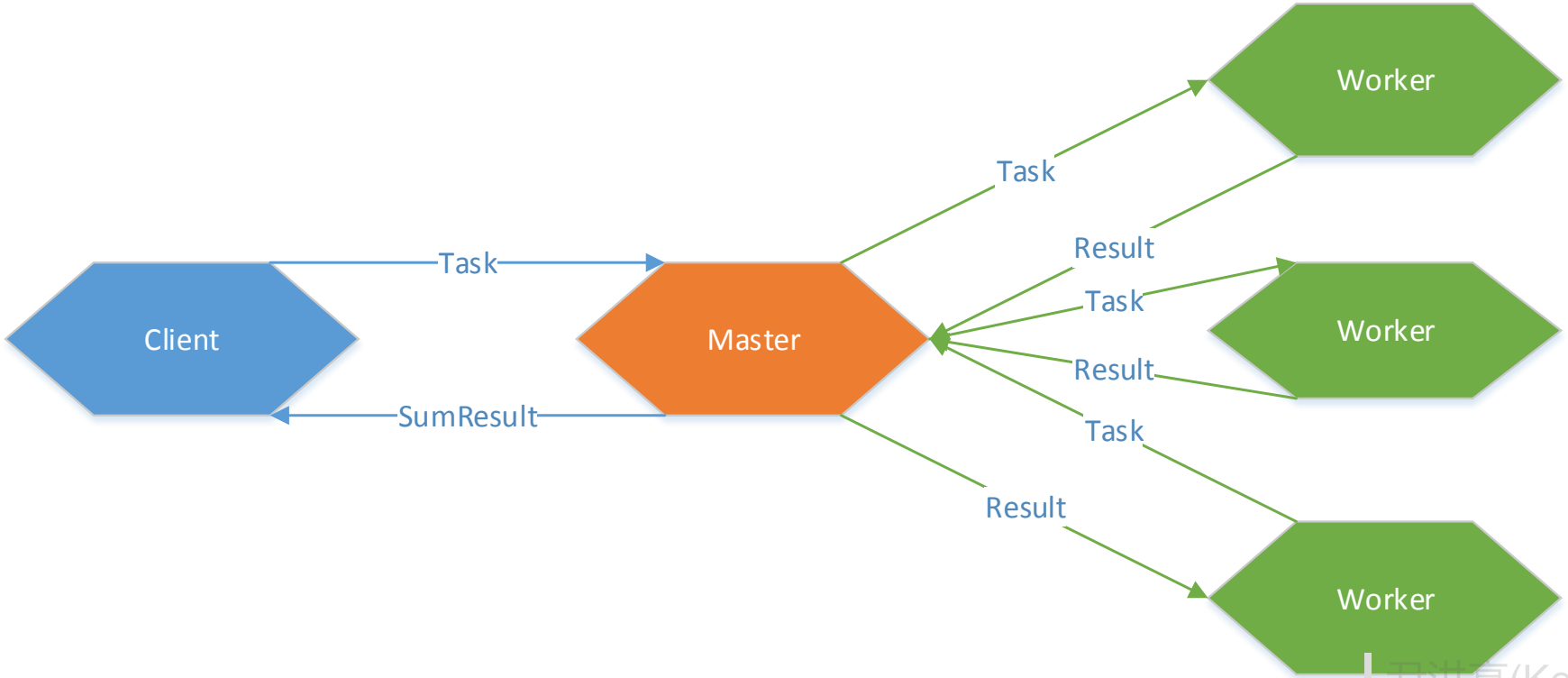
设计模式-Producer-Consumer

- Producer-Consumer称为生产者消费者模式，是消息队列中间件的核心实现模式，ActiveMQ、RocketMQ、Kafka、RabbitMQ。
- 示例： com.mimaxueyuan.demo.high.mq.Main

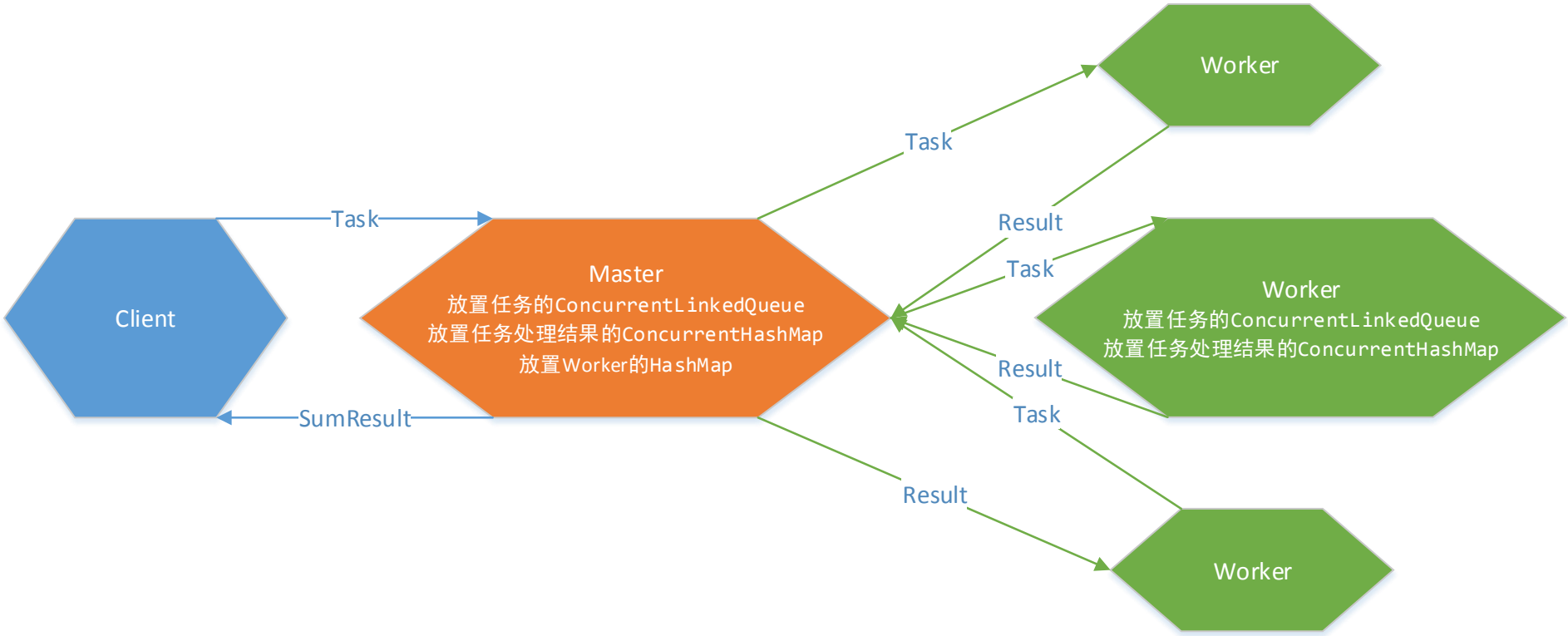
设计模式-Master-Worker

- Master-Worker模式是一种将串行任务并行化的方案，被分解的子任务在系统中可以被并行处理，同时，如果有需要，Master进程不需要等待所有子任务都完成计算，就可以根据已有的部分结果集计算最终结果集。
- 客户端将所有任务提交给Master，Master分配Worker去并发处理任务，并将每一个任务的处理结果返回给Master，所有的任务处理完毕后,由Master进行结果汇总再返回给Client
- 示例： com.mimaxueyuan.demo.high.masterworker.Main

设计模式-Master-Worker



设计模式-Master-Worker



高性能随机数

Random 与 ThreadLocalRandom在高并发场景下的性能差异和原理

高并发随机数ThreadLocalRandom与Random分析

知识点

Random存在性能缺陷，主要原因是要不断的计算新的种子更新原种子，使用CAS方法。高并发的情况下会造成大量的线程自旋，而只有一个线程会更新成功。

ThreadLocalRandom采用ThreadLocal的机制，每一个线程都是用自己的种子去进行计算下一个种子，规避CAS在并发下的问题。

源码阅读

java.util.Random

java.util.concurrent.ThreadLocalRandom 继承Random

代码演示

com.mkevin.demo4.RandomDemo0
com.mkevin.demo4.RandomDemo1 两者性能对比

高性能累加器

LongAdder、DoubleAdder、
LongAccumulator、DoubleAccumulator

高性能累加器LongAddr

知识点

AtomicLong存在性能瓶颈，由于使用CAS方法。高并发的情况下会造成大量的线程自旋，而只有一个线程会更新成功，浪费CPU资源。

LongAdder的思想是将单一的原子变量拆分为多个变量，从而降低高并发下的资源争抢。

源码阅读

java.util.concurrent.atomic.AtomicLong

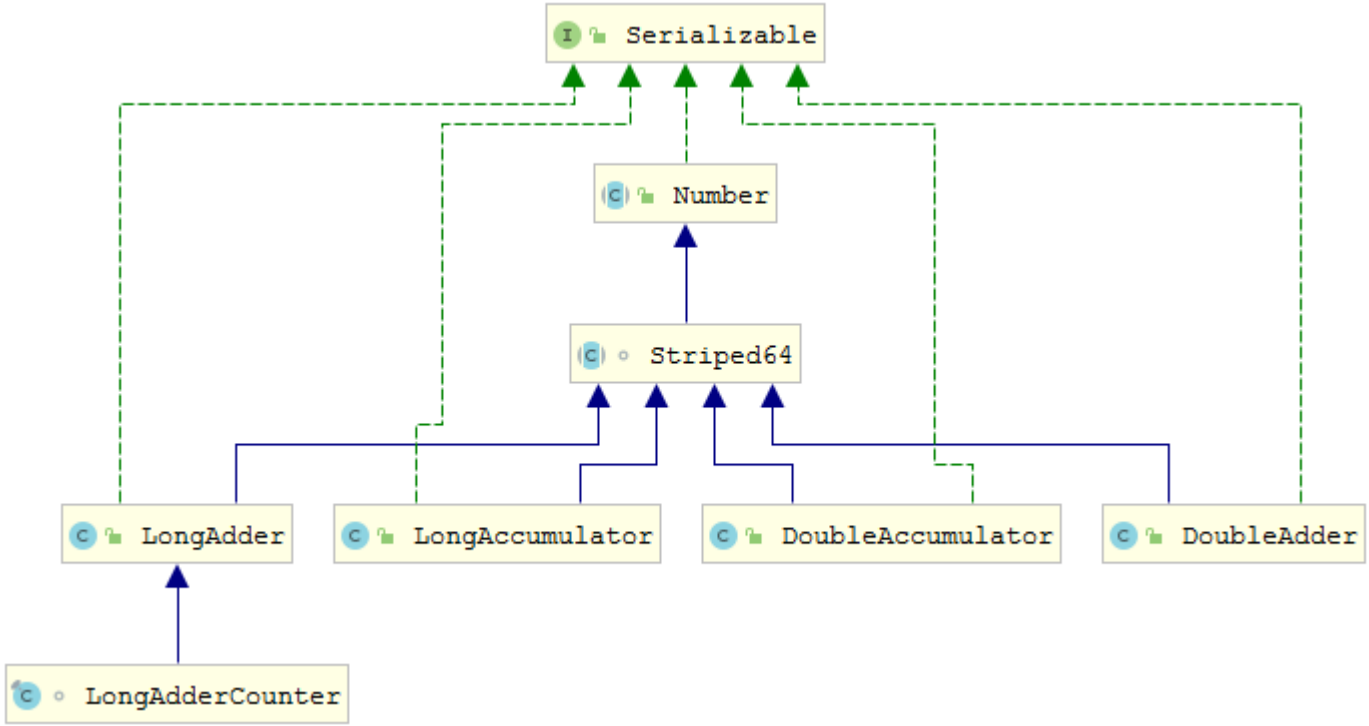
java.util.concurrent.atomic.LongAdder 继承自java.util.concurrent.atomic.Striped64

代码演示

com.mkevin.demo5.LongAdderDemo0
com.mkevin.demo5.LongAdderDemo1 两者性能对比

累加器

知识点



累加器

知识点

没有incrementAndGet、decrementAndGet这种方法，只有单独的increment、longValue这种方法，如果组合使用则需要自己做同步控制，否则无法保证原子性。

LongAddr本质上是一种空间换时间的策略，累加器家族还有以下3种

java.util.concurrent.atomic.DoubleAdder

java.util.concurrent.atomic.LongAccumulator

java.util.concurrent.atomic.DoubleAccumulator

LongAdder是LongAccumulator的特例，DoubleAdder是DoubleAccumulator的特列
Accumulator的特点是可以设置初始值、自定义累加算法

COW迭代器的弱一致性

java.util.concurrent.CopyOnWriteArrayList

java.util.concurrent.CopyOnWriteArraySet

COWIterator的弱一致性

知识点

使用COW容器的iterator方法实际返回的是COWIterator实例，遍历的数据为快照数据，其他线程对于容器元素增加、删除、修改不对快照产生影响。
对java.util.concurrent.CopyOnWriteArrayList、java.util.concurrent.CopyOnWriteArraySet均适用。

源码阅读

java.util.concurrent.CopyOnWriteArrayList
java.util.concurrent.CopyOnWriteArraySet

代码演示

com.mkevin.demo7.COWDemo0
com.mkevin.demo7.COWDemo1

LockSupport

java.util.concurrent.locks.LockSupport

LockSupport

知识点

1、LockSupport的底层采用Unsafe类来实现，他是其他同步类的阻塞与唤醒的基础。
2、park与unpark需要成对适用，parkUntil与parkNanos可以单独适用
3、先调用unpark再调用park会导致park失效
4、线程中断interrupte会导致park失效并且不抛异常
5、例如blocker可以对堆栈进行追踪，官方推荐，例如结合jstack进行使用

源码阅读

java.util.concurrent.locks.LockSupport　#知识点1

代码演示

com.mkevin.demo6.LockSupportDemo0　#知识点2
com.mkevin.demo6.LockSupportDemo1　#知识点4
com.mkevin.demo6.LockSupportDemo2　#知识点3
com.mkevin.demo6.LockSupportDemo3　#知识点5

AQS

java.util.concurrent.locks.AbstractQueuedSynchronizer

简称AQS

AbstractQueuedSynchronizer

知识点 *java.util.concurrent.locks.AbstractQueuedSynchronizer*

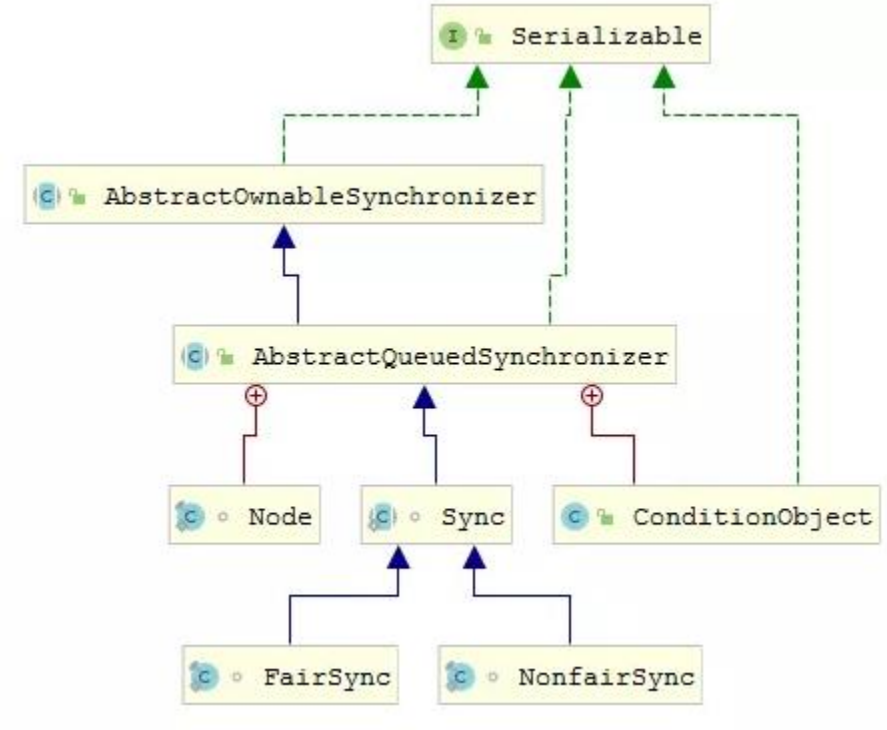
1、抽象队列同步器简称AQS，它同步器的基础组件，JUC种锁的底层实现均依赖于AQS，开发不需要使用。。

2、采用FIFO的双向队列实现，队列元素为Node（静态内部类），Node内的thread变量用于存储进入队列的线程。

3、Node节点内部的SHARED用来标记该线程是获取共享资源时被阻塞挂起后放入AQS队列的，EXCLUSIVE用来标记线程是获取独占资源时被挂起后放入AQS队列的。waitStatus记录当前线程等待状态，可以为CANCELLED（线程被取消了）、SIGNAL（线程需要被唤醒）、CONDITION（线程在条件队列里面等待）、PROPAGATE（释放共享资源时需要通知其他节点）；prev记录当前节点的前驱节点，next记录当前节点的后继节点。

4、在AQS中维持了一个单一的状态信息state，可以通过getState、setState、compareAndSetState函数修改其值。对于ReentrantLock的实现来说，state可以用来表示当前线程获取锁的可重入次数；对于读写锁ReentrantReadWriteLock来说，state的高16位表示读状态，也就是获取该读锁的次数，低16位表示获取到写锁的线程的可重入次数；对于semaphore来说，state用来表示当前可用信号的个数；对于CountDownlatch来说，state用来表示计数器当前的值。

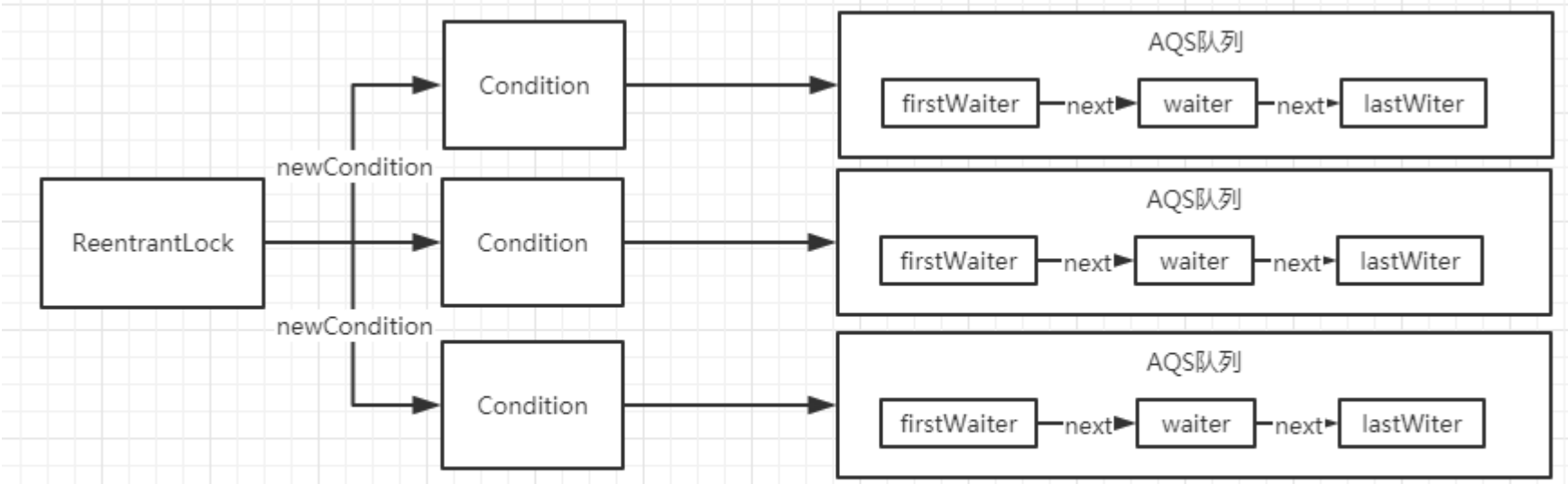
类图



Demo: com.mkevin.demo8.AQSDemo0

Demo: com.mkevin.demo8.AQSDemo1

AQS.ConditionObject



ReentrantLock.newCondition()创建的每一个Condition对象，实质上都是AQS.ConditionObject对象，而这个对象也是一个FIFO的队列

Phaser

移相器/阶段器

Phaser 图解 源码

Phaser，中文为移相器，是电子专业中使用的术语。此类在JDK7中加入的并发工具类
全路径为java.util.concurrent.Phaser

1、party
通过phaser同步的线程被称为party *（参与者）*. **所有需要同步的party必须持有同一个phaser对象.**
party需要向phaser注册,执行phaser.register()方法注册,该方法仅仅是增加phaser中的线程计数. *（不常用方式）*
也可以通过构造器注册,比如new Phaser(3)就会在创建phaser对象时注册3个party. (常用方式)
这3个party只要持有该phaser对象并调用该对象的api就能实现同步.
2、unarrived party到达一个phaser *（阶段）*之前处于unarrived状态
3、arrived 到达时处于arrived状态.一个arrived的party也被称为arrival
4、deregister
一个线程可以在arrive某个phase后退出(deregister),与参赛者中途退赛相同，可以使用
arriveAndDeregister()方法来实现. *（到达并注销）*

Phaser

5、phase计数

Phaser类有一个phase计数,初始阶段为0.当一个阶段的所有线程arrive时,会将phase计数加1,这个动作被称为advance. 当这个计数达到Integer.MAX_VALUE时,会被重置为0,开始下一轮循环

advace这个词出现在Phaser类的很多api里，比如arriveAndAwaitAdvance()、awaitAdvance(int phase)等.

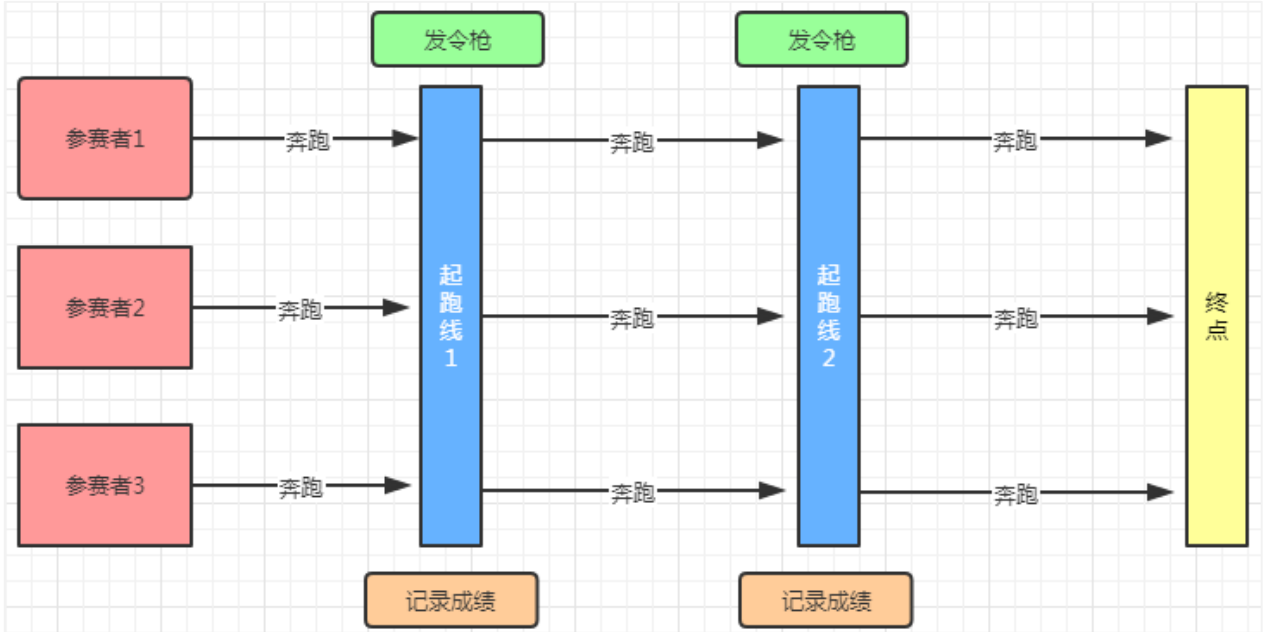
在advance时,会触发onAdvance(int phase, int registeredParties)方法的执行.

6、onAdvance(int phase, int registeredParties)

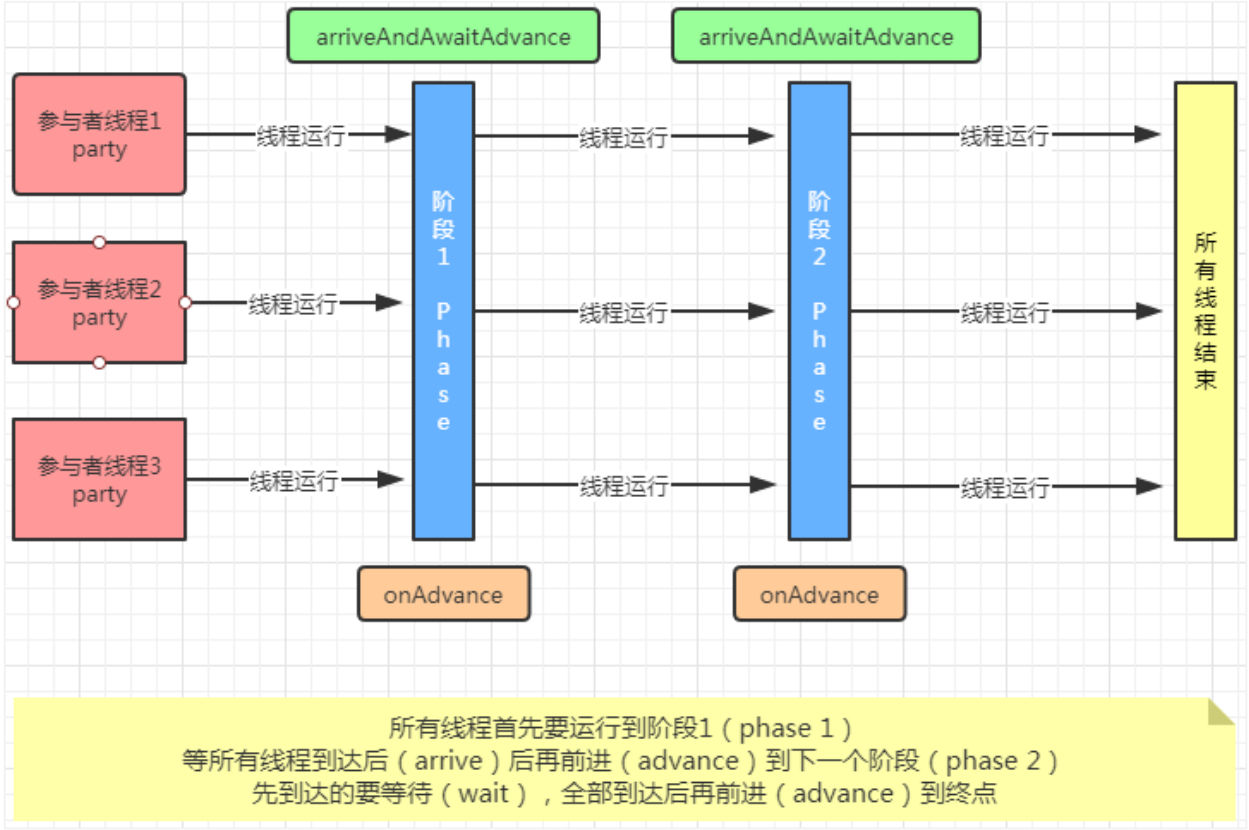
可以在这个方法中定义advance过程中需要执行何种操作。

如果需要进入下一阶段(phase)执行,返回false.如果返回true,会导致phaser结束

因此该方法也是终止phaser的关键所在



所有的参赛者必须先奔跑到第一个起跑线
等所有人到达后，发令枪立刻给出信号，所有参赛者一同出发奔向第二个起跑线
所有参赛者到达第二个起跑线后，发令枪立刻给出信号，所有参赛者一同出发奔向终点
最后结束



所有线程首先要运行到阶段1（phase 1）
等所有线程到达后（arrive）后再前进（advance）到下一个阶段（phase 2）
先到达的要等待（wait），全部到达后再前进（advance）到终点

源码

代码演示

com.mkevin.demo10.PhaserDemo1

com.mkevin.demo10.PhaserDemo2

com.mkevin.demo10.PhaserDemo3

StampedLock

写锁、悲观读锁、乐观读锁

StampedLock简介&写锁writeLock

写写互斥、读写互斥、读读共享

StampedLock类，在JDK8中加入全路径为java.util.concurrent.locks.StampedLock。
功能与RRW（ReentrantReadWriteLock）功能类似提供三种读写锁。

StampedLock中引入了一个stamp（邮戳）的概念。它代表线程获取到锁的版本，每一把锁都有一个唯一的stamp。

写锁writeLock，是排它锁、也叫独占锁，相同时间只能有一个线程获取锁，其他线程请求读锁和写锁都会被阻塞。

功能类似于ReetrantReadWriteLock.writeLock。
区别是**StampedLock的写锁是不可重入锁**。当前没有线程持有读锁或写锁的时候才可以获得获取到该锁。

写锁writeLock

Demo：com.mkevin.demo9.StampedLockDemo1

writeLock与unlockWrite必须成对儿使用，解锁时必须需要传入相对应的stamp才可以释放锁。每次获得锁之后都会得到一个新stamp值。

同一个线程获取锁后，再次尝试获取锁而无法获取，则证明其为非重入锁。

对于ReentrantLock，同一个线程获取锁后，再次尝试获取锁可以获取，则证明其为重入锁。

对于ReentrantReadWriteLock.WriteLock，同一个线程获取写锁后，再次尝试获取锁依然可获取锁，则证明其为重入锁。

悲观读readLock

Demo: com.mkevin.demo9.StampedLockDemo2

悲观读锁是一个共享锁，没有线程占用写锁的情况下，多个线程可以同时获取读锁。如果其他线程已经获得了写锁，则阻塞当前线程。

读锁可以多次获取（没有写锁占用的情况下），写锁必须在读锁全部释放之后才能获取写锁。

只要还有任意的锁没有释放（无论是写锁还是读锁），这时候来尝试获取写锁都会失败，因为读写互斥，写写互斥。写锁本身就是排它锁。

在多个线程之间依然存在写写互斥、读写互斥、读读共享的关系

为什么叫悲观读锁？悲观锁认为数据是极有可能被修改的，所以在使用数据之前都需要先加锁，锁未释放之前如果有其他线程想要修改数据（加写锁）就必须阻塞它。
悲观读锁并算不上绝对的悲观，排他锁才是真正的悲观锁，由于读锁具有读读共享的特性，所以对于读多写少的场景十分适用，可以大大提高并发性能。

乐观读readLock

tryOptimisticRead通过名字来记忆很简单，try代表尝试，说明**它是无阻塞的。Optimistic乐观的**，Read代表读锁。
乐观锁认为数据不会轻易的被修改，因此在操作数据前并没有加锁（使用cas方式更新锁的状态），而是采用试探的方式，只要当前没有写锁就可以获得一个非0的stamp，如果已经存在写锁则返回一个为0的stamp。
又有**没有使用cas方法，也没有真正的加锁，所以并发性能要比readLock还要高。**

但是由于没有使用真正的锁，如果数据中途被修改，就会造成数据不一致问题。
但是**StampLock使用快照的方式**，需要复制一份要操作的变量到方法栈，操作的数据只是一个快照，从而**保证了数据的最终一致性。但是读线程可能使用的数据不是最新的。**
特别适用于读多写少的高并发场景

乐观读readLock

Demo：com.mkevin.demo9.StampedLockDemo3

tryOptimisticRead与validate一定要紧紧挨着使用，否则在获取和验证之间很可能数据被修改。
如果这期间锁发生变化则validate返回false，否则返回true。

原理很简单，就是我先尝试获取，这时候没有写锁我就拿到了一个锁，在我真正要使用的时候我再验证一下是否发生了改变，如果没有发生改变就可以安心使用。

如果某个线程已经获取了写锁，这时候再尝试获取乐观锁也是可以获取的，只是得到的stamp为0，无法通过validate验证。

乐观读锁本质上并未加锁，而是提供了获取和检测的方法，由程序人员来控制该做些什么。
虽然性能大大提升，但是却增加了开发人员的复杂度，如果不是特别高的并发场景，对性能不要求极致，可以不考虑使用。

锁的分类

各种锁

锁的分类

锁

五类锁

五类锁是并发编程的基础、线程安全的重要保障

乐观锁/悲观锁

是否在修改之前给记录增加排它锁

公平锁/非公平锁

请求锁的时间顺序是否与获得锁的时间顺序一致。一致为公平锁，不一致为非公平锁

独占锁/共享锁

是否可以被多个线程共同持有，可以则为共享锁、不可以则为独占锁

可重入锁

一个线程再次获取它自己已经获取的锁时是否会被阻塞

自旋锁

无法获取锁时是否立刻阻塞，还是继续尝试获取指定次数

乐观锁与悲观锁

1. 乐观锁和悲观锁的概念来自于数据库
2. 悲观锁对数据被修改持悲观态度，**认为数据很容易就会被其他线程修改**，所以在**处理数据之前先加锁**，处理完毕释放锁。
3. 乐观锁对数据被修改持乐观态度，**认为数据一般情况下不会被其他线程修改**，所以在**处理数据之前不会加锁**，而是在数据进行更新时进行冲突检测。

4. 对于数据库的悲观锁就是排它锁，在处理数据之前，先尝试给记录加排它锁，如果成功则继续处理，如果失败则挂起或抛出异常，直到数据处理完毕释放锁。
5. 对于数据库的乐观锁所典型的就是CAS方式更新, 例如:update name='kevin' where id=1 and name='kevin0'， 在更新数据的时候校验这个值是否发生了变化，类似于CAS的操作。

公平锁与非公平锁

1. 据线程获取锁的抢占机制，锁可以分为公平锁和非公平锁，**最早请求锁的线程将最早获取到锁。**而非公平锁则先请求不一定先得。JUC中的ReentrantLock提供了公平和非公平锁特性。·
2. 公平锁：ReentrantLock pairLock = new ReentrantLock(true)
3. 非公平锁：ReentrantLock pairLock = new ReentrantLock(false)，如果构造函数不传递参数，则默认是非公平锁。
4. 非必要情况下使用非公平锁，**公平锁存在性能开销**

独占锁与共享锁

1. **只能被单个线程所持有**的锁是独占锁，**可以被多个线程持有**的锁是共享锁。
2. ReentrantLock就是以独占方式实现的，属于悲观锁
3. ReadWriteLock读写锁是以共享锁方式实现的，属于乐观锁
4. StampedLock的写锁，属于悲观锁。
5. StampedLock的乐观读锁，悲观读锁、属于乐观锁。

可重入锁

1. 当一个线程想要获取**本线程已经持有的锁**时，**不会被阻塞**，而是能够再次获得这个锁，这就是重入锁。
2. Synchornized是一种可重入锁，内部维护一个线程标志(谁持有锁），以及一个计数器。
3. ReentrantLock也是一种可重入锁
4. ReadWriteLock、StampedLock的读锁也是可重入锁

自旋锁

1. 当获取锁的时候如果发现锁已经被其他线程占有，则不阻塞自己，也不释放CPU使用权，而是尝试多次获取，如果尝试了指定次数之后仍然没有获得锁，再阻塞线程。
2. 自旋锁认为锁不会被长时间持有，**使用CPU时间来换取线程上下文切换的开销**，从而提高性能。但是可能会浪费CPU资源。
3. -XX:PreBlockSpin=n可以设置自旋次数（已经成为了历史），在Jdk7u40时被删除了，其实在jkd6的时候就已经无效了，**现在HotSpotVM采用的是adaptive spinning（自适应自旋），虚拟机会根据情况来对每个线程使用不同的自旋次数。**

ThreadFactory

改变线程池中，线程创建的行为

ThreadFactory

知识点

我们通常使用线程池的submit方法将任务提交到线程池内执行。
如果此时线程池内有空闲的线程，则会立即执行该任务，如果没有则需要根据线程池的类型选择等待，或者新建线程。
所以线程池内的线程并不是线程池对象初始化（new）的时候就创建好的。而是当有任务被提交进来之后才创建的，而创建线程的过程是无法干预的。
如果我们想在每个线程创建时记录一些日志，或者推送一些消息那怎么做？

使用ThreadFactory

第一步：编写ThreadFactory接口的实现类
第二步：创建线程池时传入ThreadFactory对象

Demo

com.mkevin.demo13.ThreadFactoryDemo0

线程池内异常的优雅处理

优雅的处理线程池内未捕获异常

线程池状态

- 线程池内运行的线程如果发生异常，一定要捕获，要养成习惯。
- 可以采用更优雅的方式处理所有线程的异常，例如记录日志，发送预警消息等。
- 结合ThreadFactory以及线程的setUncaughtExceptionHandler方法来处理最为优雅
- **对execute提交的任务有效，对submit提交的任务无效，巨坑！**

Demo

- com.mkevin.demo15.ThreadExceptionDemo1

关闭线程池

shutdown和shutdownNow的作用和区别

shutdown 与 shutdownNow

知识点

- shutdown让线程池内的任务继续执行完毕，但是不允许新的任务提交
- shutdown方法不阻塞, 等所有线程执行完毕后，销毁线程
- shutdown之后提交的任务会抛出RejectedExecutionException异常，代表拒绝接收

- shutdownNow之后提交的任务会抛出RejectedExecutionException异常，代表拒绝接收
- shutdownNow之后会引发sleep、join、wait方法的InterruptedException异常
- 如果任务中没有触发InterruptedException的条件，则任务会继续运行直到结束

Demo

com.mkevin.demo14.ThreadPoolDemo1

线程池的结束状态

线程池内线程运行结束的标志

线程池状态

- isShutdown用来判断线程池是否已经关闭
- isTerminated任务全部执行完毕，并且线程池已经关闭，才会返回true
- awaitTermination 阻塞，直到所有任务在关闭请求后完成执行，或发生超时，或当前线程中断（以先发生者为准）。

Demo

- com.mkevin.demo14.ThreadPoolDemo2

允许核心线程超时策略

核心线程也允许超时销毁

允许核心线程超时策略

- 核心线程也允许销毁， allowsCoreThreadTimeOut就用来做这个事
- 设置控制核心线程是否可能超时的策略，如果在保持活动时间内没有任务到达，则该策略将在新任务到达时根据需要被替换。
- 如果为false，则不会由于缺少传入任务而终止核心线程。
- 如果为true，则应用于非核心线程的相同保持活动策略也适用于核心线程。
- 为避免连续更换线程，设置为true时保持活动时间必须大于零。
- 通常应该在池被激活之前调用此方法。

Demo

- com.mkevin.demo16.allowCoreThreadTimeOutDemo

核心线程预启动策略

核心线程预启动

核心线程预启动策略

- 默认情况下，核心线程只有在任务提交的时候才会创建
- 而预启动策略，可以让核心线程提前启动，从而增强最初提交的线程运行性能
- prestartCoreThread启动1个核心线程，覆盖仅在执行新任务时启动核心线程的默认策略。如果所有核心线程都已启动，则此方法将返回false。
- prestartAllCoreThreads启动所有核心线程。覆盖仅在执行新任务时启动核心线程的默认策略。如果核心线程全部启动后再次调用，则会返回0

Demo

- com.mkevin.demo17.prestartAllCoreThreadsDemo

线程及线程池切面

切面

线程及线程池切面

- 在线程执行前、执行后增加切面，在线程池关闭时执行某段程序。
- 需要实现自己的线程池类，并覆写beforeExecute、afterExecute、terminated方法

Demo

- com.mkevin.demo19.BeforAfterTerminatedDemo

移除线程池中的任务

怎样删除线程池中的任务

移除线程池中的任务

- 使用remove方法
- 已经正在运行中的任务不可以删除
- execute方法提交的，未运行的任务可以删除
- **submit方法提交的，未运行任务就不可以删除，小心采坑！**

Demo

- com.mkevin.demo20.TaskRemoveDemo

获取各种线程池状态数据

大量的get方法怎么玩?

获取各种线程池状态数据

- **可以获取线程池的各种动态和静态数据，用于程序控制。**
- 返回核心线程数getCorePoolSize
- 返回当前线程池中的线程数getPoolSize
- 返回最大允许的线程数getMaximumPoolSize
- 返回池中同时存在的最大线程数getLargestPoolSize
- 返回预定执行的任务总和getTaskCount
- 返回当前线程池已经完成的任务数getCompletedTaskCount
- 返回正在执行任务的线程的大致数目getActiveCount
- 返回线程池空闲时间getKeepAliveTime

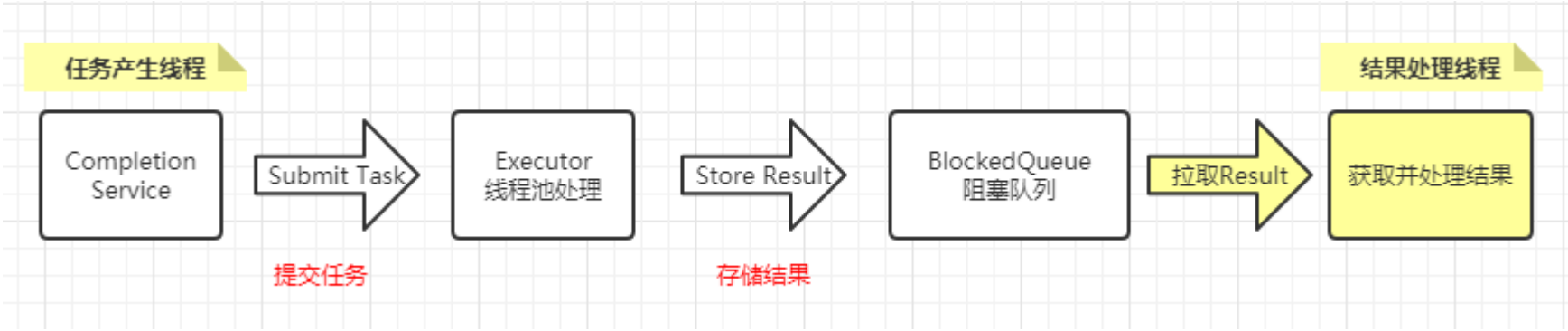
Demo

- com.mkevin.demo18.ThreadPoolGetDemo

CompletionService

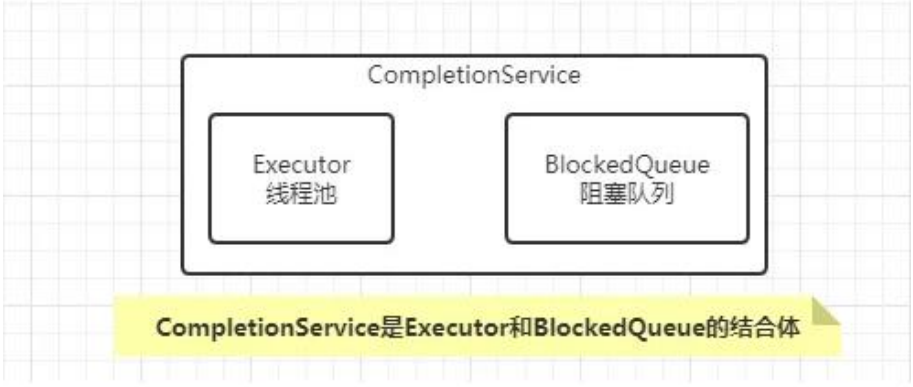
轻松搞定Master Worker模式

CompletionService



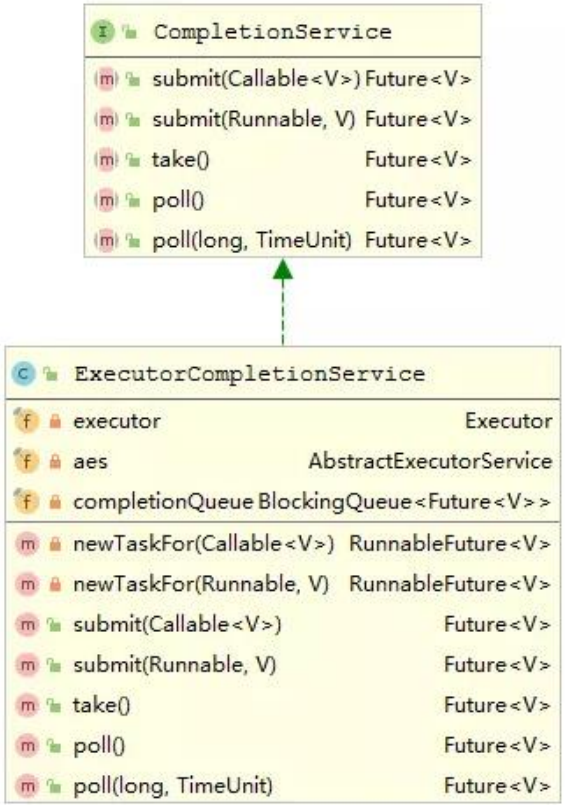
这张图说明了CompletionService的使用模式
你可用它不断的提交任务（线程）给Executor处理
处理后的结果都会自动放入BlockedQueue
另外一个线程不断的从队列里取得处理结果
好处是，哪个任务先处理完就能先得到哪个结果
最后做汇总处理
从而轻松完成MasterWorker模式相同的功能

CompletionService



这张图说明了CompletionService的本质
就是线程池Executor加上阻塞队列BlockedQueue
线程池Executor用来处理任务
BlockedQueue用来获取每个线程的运行结果

CompletionService



- submit用于提交任务
- take用于获取处理结果（阻塞式）
- poll也用于获取处理结果（非阻塞式）

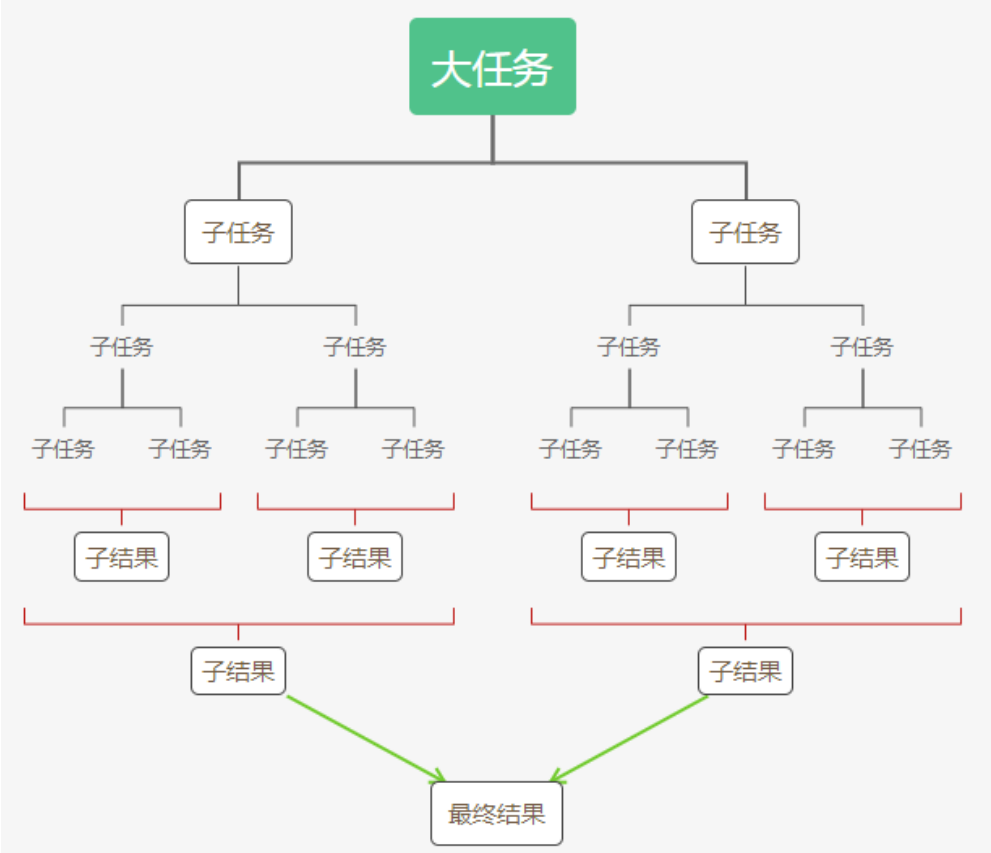
Demo

com.mkevin.demo11.CompletionServiceDemo0
com.mkevin.demo11.CompletionServiceDemo1
com.mkevin.demo11.CompletionServiceDemo2
com.mkevin.demo11.CompletionServiceDemo3

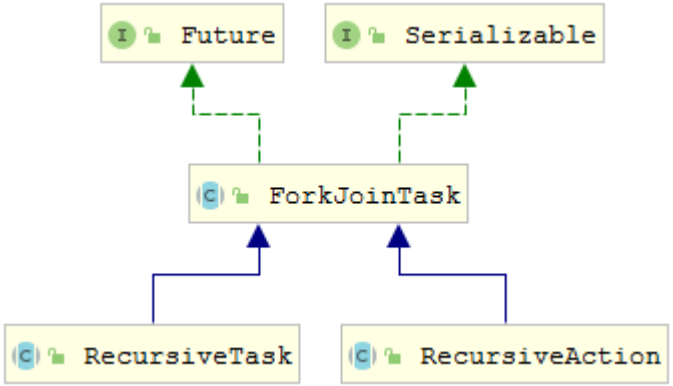
ForkJoin

ForkJoin模式的使用

ForkJoin思想



ForkJoin使用



- 两个重要的实现类，又同时是抽象类
- *java.util.concurrent.RecursiveTask* *递归任务*
- *java.util.concurrent.RecursiveAction* *递归活动*

Demo

- com.mkevin.demo12.ForkJoinDemo0 至 ForkJoinDemo7

精品教程

JAVA并发编程系列

一次性搞定数据库事务

Memcached系列

一次性精通JVM

Disruptor高并发框架

SpringCloud微服务架构

RateLimiter访问限流

程序员转型项目经理

合作平台

扫描二维码学习更多教程

腾讯课堂 学习成就梦想

网易云课堂 我的职业课堂

51CTO.com 技术成就梦想