- 线程
  - 1、创建线程的几种方式
  - 2、线程的几种状态
  - 3、sleep() VS wait()
- 线程池
  - 使用线程池的优点
  - 线程池工作原理
  - 线程池用过吗都有什么参数？
  - 如何设置核心线程数大小
  - 各种线程池的使用场景
- 并发关键字
  - synchronized
    - 实现原理
    - 锁的力度
    - synchronized特点
  - volatile
    - 实现原理
    - 特点
    - volatile和synchronized的区别
    - 什么场景volatile下可替换synchronized
  - Lock锁
    - AQS
    - Lock接口特点
    - Lock接口实现类
      - ReentrantLock
      - ReentrantLock特点
      - synchronized 和 ReentrantLock 异同
      - ReadWriteLock
      - Condition
- JMM(内存模型)
  - 什么是JMM
  - 两个线程之间是如何通讯的
  - 重排序
  - happens-before

- 并发容器
  - ThreadLocal
    - ThreadLocal 是什么底层如何实现？
      - ThreadLocalMap如何处理Hash冲突的
    - ThreadLocal可能导致内存泄漏，为什么？如何避免？
  - ConcurrentHashMap
  - CopyOnWriteArrayList
    - 实现方式
    - 使用场景
    - 存在问题
  - 阻塞队列
    - ArrayBlockingQueue
    - LinkedBlockingQueue
    - PriorityBlockingQueue
    - DelayQueue
    - SynchronousQueue
    - LinkedTransferQueue
    - LinkedBlockingDeque
- 原子操作
  - 什么是原子操作
  - CAS

线程

1、创建线程的几种方式

- 继承Thread类，重新run方法
- 实现Runable接口
- 实现callable接口

2、线程的几种状态

- NEW 状态是指线程刚创建, 尚未启动
- RUNNABLE 状态是线程正在正常运行中, 当然可能会有某种耗时计算/IO等待的操作/CPU时间片切换等, 这个状态下发生的等待一般是其他系统资源, 而不是锁, Sleep等
- BLOCKED 这个状态下, 是在多个线程有同步操作的场景, 比如正在等待另一个线程的synchronized 块的执行释放, 或者可重入的 synchronized块里别人调用 wait() 方法, 也就是这里是线程在等待进入临界区
- WAITING 这个状态下是指线程拥有了某个锁之后, 调用了他的wait方法, 等待其他线程/锁拥有者调用 notify / notifyAll 一遍该线程可以继续下一步操作, 这里要区分 BLOCKED 和 WATING 的区别, 一个是在临界点外面等待进入, 一个是在理解点里面wait等待别人notify, 线程调用了join方法 join了另外的线程的时候, 也会进入WAITING状态, 等待被他join的线程执行结束
- TIMED_WAITING 这个状态就是有限的(时间限制)的WAITING, 一般出现在调用 wait(long), join(long)等情况下, 另外一个线程sleep后, 也会进入 TIMED_WAITING状态
- TERMINATED 这个状态下表示 该线程的run方法已经执行完毕了, 基本上就等于死亡了(当时如果线程被持久持有, 可能不会被回收)

3、sleep() VS wait()

- sleep()方法是Thread的静态方法，而wait是Object实例方法
- wait()方法必须要在同步方法或者同步块中调用，也就是必须已经获得对象锁。而sleep()方法没有这个限制可以在任何地方种使用。另外，wait()方法会释放占有的对象锁，使得该线程进入等待池中，等待下一次获取资源。而sleep()方法只是会让出CPU并不会释放掉对象锁；
- sleep()方法在休眠时间达到后如果再次获得CPU时间片就会继续执行，而wait()方法必须等待Object.notift/Object.notifyAll通知后，才会离开等待池，并且再次获得CPU时间片才会继续执行。

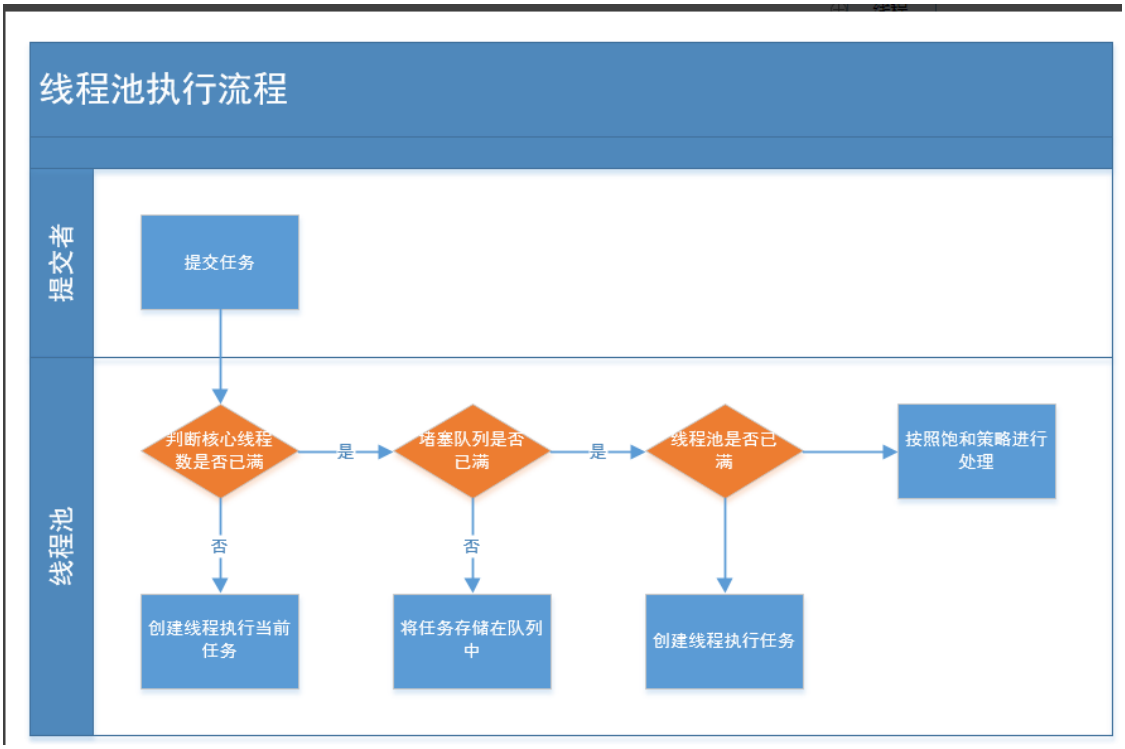
线程池

使用线程池的优点

在实际使用中，线程是很占用系统资源的，如果对线程管理不善很容易导致系统问题。因此，在大多数并发框架中都会使用线程池来管理线程，使用线程池管理线程主要有如下好处：

- ①降低资源消耗。通过复用已存在的线程和降低线程关闭的次数来尽可能降低系统性能损耗；
- ②提升系统响应速度。通过复用线程，省去创建线程的过程，因此整体上提升了系统的响应速度；
- ③提升系统响应速度。通过复用线程，省去创建线程的过程，因此整体上提升了系统的响应速度；

线程池工作原理



线程池用过吗都有什么参数？

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

- corePoolSize：核心线程数大小，当线程数 < 核心线程数，创建执行任务线程
- maximumPoolSize：最大线程数，当线程数>=corePoolSize，会把任务放到maximumPoolSize中
  - keepAliveTime：保持存活时间，当线程数>corePoolSize的空闲线程能保持的最大时间
  - unit：时间单位
- threadFactory 创建线程的工厂
- workQueue：保存任务的阻塞队列
  - handler：超过阻塞队列大小，使用什么样的拒绝策略
    - ThreadPoolExecutor.AbortPolicy() ，直接抛出异常 RejectedExecutionException
    - ThreadPoolExecutor.CallerRunsPolicy() ，直接调用 run 方法并且阻塞执行。
    - ThreadPoolExecutor.DiscardPolicy() ，直接丢弃后来的任务。
    - hreadPoolExecutor.DiscardOldestPolicy() ，丢弃在队列中队首的任务。

如何设置核心线程数大小

- CPU密集型应用，线程池大小设置为N+1（因为cpu密集型任务使的cpu使用率很高，若开过多的线程数，只能增加上线文频繁切换次数，会带来过多的额外开销）
- 如果IO密集型应用，线程池大小设置为2N+1（IO密集型的任务CPU使用率不高，因此可以让CPU等待IO的时候，去处理任务，提高CPU利用率）
- 如果是混合型的应用，则分别设置核心线程数

各种线程池的使用场景

- newCacheThreadPool:执行很短期异步的小程序或者负载较轻的服务器
- newFixedThreadPool:执行长期的任务，性能好很多
- newSingleThreadPool:一个任务一个任务的去执行
- newScheduleThreadPool:周期性的执行任务

并发关键字

synchronized

实现原理

- synchronized是通过Java对象投标记和Monitor对象实现的同步
- 同步代码块是使用 monitorenter 和 monitorexit 指令实现的 （监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块）

锁的力度

- 普通同步方法：锁的是当前实例对象
- 静态同步方法：多的是当前Class对象
- 同步代码块：括号里面的对象

synchronized特点

- 采用独占式的加锁方式
- 可重入锁
- 一个线程获取之后，其他线程处于等待状态，如果获取锁的线程处于休眠状态，那么其他线程会一直阻塞
- 不需要手动获取和释放锁，使用简单
- 不具有继承性

volatile

实现原理

特点

- 保证内存可见性
- 禁止指令重排
- 不能保证所有操作原子性

volatile和synchronized的区别

- Volatile本质是告诉JVM当前变量在寄存器中的值是不确定的，需要从内存中读取。synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程是阻塞的
- Volatile只能修饰变量，不能保证原子性；synchronized可以修饰变量（保证变量可见性和原子性）、方法、代码块
- Volatile不会造成线程阻塞；synchronized会造成线程阻塞
- Volatile编辑的变量不会被编译器优化；synchronized标记的变量可以被编译器优化

什么场景volatile下可替换synchronized

- 只保证资源可见性的时候volatile可以替换synchronized
- 1写 N读
- 不与其他变脸构成不变条件时候使用volatile

Lock锁

AQS

- 是一个用于构建锁和同步容器的同步器
- 使用一个FIFO的队列标识排队等待的线程
- 提供给同步组件实现者，为其屏蔽了同步状态管理、线程队列等底层操作，实现者只需要通过重新AQS提供的模板方法实现同步组件语义即可

Lock接口特点

- Lock提供基于API的可操作性，提供可相应中断方式获取锁，超时时间获取锁以及非阻塞式获取锁
- synchronized的执行完同步快以及遇到异常或自动释放锁，而Lock需要调用unlock手动释放锁

Lock接口实现类

ReentrantLock

ReentrantLock特点

- 通过参数控制是否是公平锁，默认非公平锁
- 可重入锁
- 他的实现是一种自旋锁，通过循环调用CAS操作来实现加锁

synchronized 和 ReentrantLock 异同

- 相同点
  - 都是可重入锁
  - 都实现了内存可见性和多线程同步功能
- 不同点
  - 同步机制实现方式不同
    - synchronized是通过Java对象投标记和Monitor对象实现的同步
    - ReentrantLock通过CAS、AQS和LockSupport来实现的
  - 可见性实现方式不同
    - synchronized依赖jvm内存模型保证共享变量的多线程内存可见
    - ReentrantLock是通过AQS的volatile state保证共享变量多线程内存可见性
  - 使用方式不同
    - synchronized可以修改实例方法（锁住当前对象）、静态方法（锁住类对象）、代码块（制定锁对象）
    - ReentrantLock显示调用tryLock和lock方法，手动释放
  - 功能丰富程度不同
    - synchronized不可设置等待时间，不可中断
    - ReentrantLock提供等候锁、可中断所等丰富功能
  - 锁类型不同
    - synchronized只支持非公平锁
    - ReentrantLock支持费公平锁和公平锁，默认非公平锁

ReadWriteLock

- 它是一个读写锁，用来提升并发程序性能的锁分离技术
- 同样通过Lock接口中的模板方法实现的
- 可用于读多写上的场景
- 如果写已经被其他线程持有，那么任何读取者都不能访问获取，直到写锁释放

Condition

//TODO

JMM(内存模型)

什么是JMM

- Java 虚拟机规范中试图定义一种 Java 内存模型（Java Memory Model，JMM）来屏蔽掉各层硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。
- Java 内存模型规定了所有的变量都存储在主内存（Main Memory）中。每条线程还有自己的工作内存（Working Memory），线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在主内存中进行，而不能直接读写主内存中的变量

两个线程之间是如何通讯的

- 共享内存
- 消息传递

重排序

在程序执行过程中，为了提高性能，处理器和编译器常常会对指令进行重排序

满足条件

- 单线程环境下不能改变程序运行结果（会影响多线程执行结果）
- 存在数据依赖关系的不能重排

happens-before

《Java 内存模型之 happens-before》

```
i = 1; // 线程 A 执行
j = i; //线程 B 执行
```

j 是否等于 1 呢？假定线程 A 的操作（i = 1）happens-before 线程 B 的操作（j = i），那么可以确定，线程 B 执行后 j = 1 一定成立。如果他们不存在 happens-before 原则，那么 j = 1 不一定成立。这就是happens-before原则的威力。

并发容器

ThreadLocal

ThreadLocal 是什么底层如何实现？

ThreadLocal内部还有一个静态内部类ThreadLocalMap，该内部类才是实现线程隔离机制的关键，get()、set()、remove()都是基于该内部类操作。ThreadLocalMap提供了一种用键值对方式存储每一个线程的变量副本的方法，key为当前ThreadLocal对象，value则是对应线程的变量副本。

ThreadLocalMap如何处理Hash冲突的

由于ThreadLocalMap使用线性探测法来解决散列冲突，所以实际上Entry[]数组在程序逻辑上是作为一个环形存在的。

至此，我们已经可以大致勾勒出ThreadLocalMap的内部存储结构。下面是我绘制的示意图。虚线表示弱引用，实线表示强引用。

ThreadLocal可能导致内存泄漏，为什么？如何避免？

Java中的四种引用类型（强、软、弱、虚）

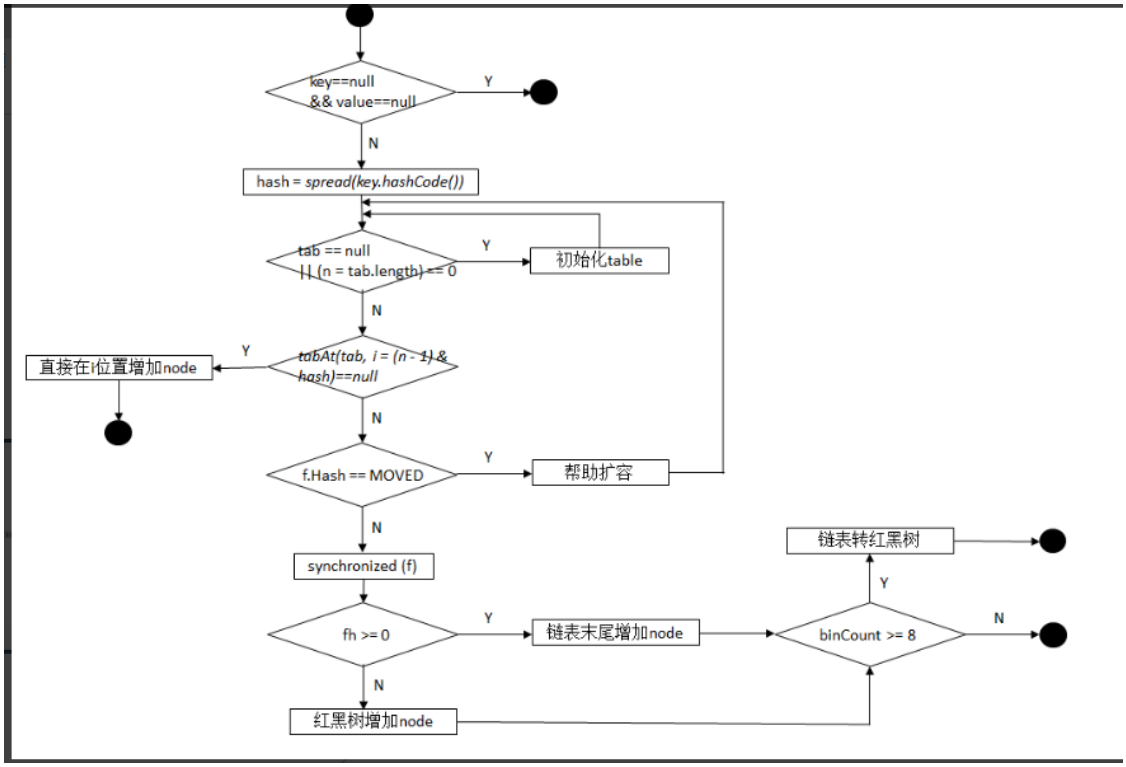
ThreadLocal在ThreadLocalMap中是以一个弱引用身份被Entry中的Key引用的，因此如果ThreadLocal没有外部强引用来引用它，那么ThreadLocal会在下次JVM垃圾收集时被回收。这个时候就会出现Entry中Key已经被回收，出现一个null Key的情况，外部读取ThreadLocalMap中的元素是无法通过null Key来找到Value的。因此如果当前线程的生命周期很长，一直存在，那么其内部的ThreadLocalMap对象也一直生存下来，这些null key就存在一条强引用链的关系一直存在：Thread --> ThreadLocalMap-->Entry-->Value，这条强引用链会导致Entry不会回收，Value也不会回收，但Entry中的Key却已经被回收的情况，造成内存泄漏。

每次使用完ThreadLocal，都调用它的remove()方法，清除数据。

ConcurrentHashMap

- JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作
- JDK8中的实现也是锁分离思想，只是锁住的是一个node，而不是JDK7中的Segment；锁住Node之前的操作是基于在volatile和CAS之上无锁并且线程安全的。

put操作流程：



CopyOnWriteArrayList

实现方式

- Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略
- 实际上CopyOnWriteArrayList内部维护的就是一个数组,并且该数组引用是被volatile修饰

使用场景

适用于读操作远远多于写操作的情况

存在问题

- CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性
- 所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

阻塞队列

ArrayBlockingQueue

- 数组结构组成的有界阻塞队列
- 此队列先进先出（FIFO）的原则对元素进行排序，默认情况下不保证线程公平访问队列

LinkedBlockingQueue

- 一个由链表组成的误解队列
- 按照先出先进的原则对元素进行排序

PriorityBlockingQueue

- 一个支持优先级排序的无界阻塞队列

DelayQueue

- 支持延时获取元素的无界阻塞队列，即可以执行多久才能从队列中获取元素

SynchronousQueue

- 存储元素的阻塞队列
- 每一个put必须等待一个take操作，否则不能继续添加元素，支持公平访问队列

LinkedTransferQueue

- 链表组成的无界阻塞队列

LinkedBlockingDeque

- 链表组成的双向阻塞队列

原子操作

什么是原子操作

- 不可中断的一个或一系列操作
- 原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段

CAS

- ABA问题
  - 比如说一个线程 one 从内存位置 V 中取出 A ，这时候另一个线程 two 也从内存中取出 A ，并且 two 进行了一些操作变成了 B ，然后 two 又将 V 位置的数据变成 A ，这时候线程 one 进行 CAS 操作发现内存中仍然是 A ，然后 one 操作成功。尽管线程one的CAS操作成功，但可能存在潜藏的问题。
- 循环时间长开销大
  - 于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized 。
- 只能保证一个共享变量的原子操作
  - 当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。