Java多线程性能调优

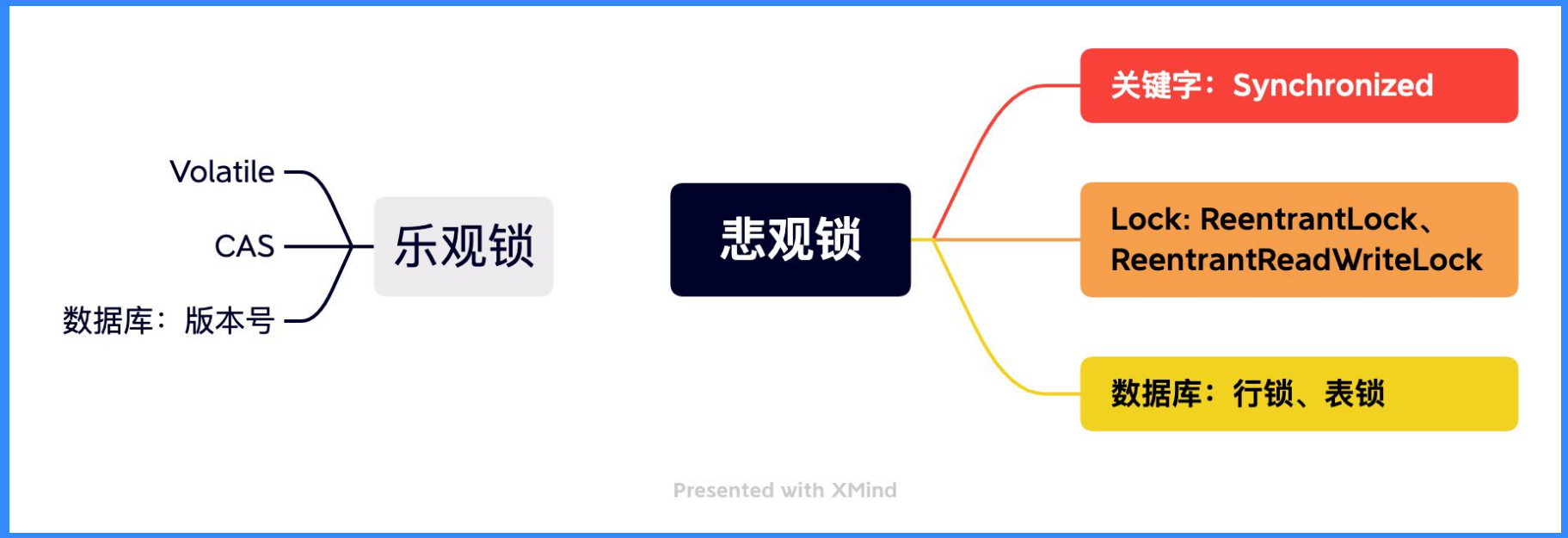
OpenTelemetry中国发起人 前阿里、美团架构师 蒋志伟

什么是乐观锁、悲观锁

ri ‘entrent

悲观锁：即很悲观，每次拿数据的时候都觉得数据会被人更改，所以拿数据的时候就把这条记录锁掉，这样别人就没法改这条数据了，一直到锁释放。

乐观锁：即很乐观，查询数据的时候总觉得不会有人更改数据，等到更新的时候再判断这个数据有没有被人更改，有人更改了则本次更新失败。



悲观锁：Synchronized 关键字

Synchronized 是基于底层操作系统的 Mutex Lock 实现的，每次获取和释放锁操作都会带来用户态和内核态的切换，在锁竞争激烈的情况下，Synchronized 同步锁在性能上就表现得非常糟糕，它也常被大家称为重量级锁。

```
// 关键字在代码块上，锁为括号里面的对象
public void methodB() {
    Object obj = new Object();
    synchronized (obj) {
        // code
    }
}
```

编译结果

```
public void methodB();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=4, args_size=1
       0: new           #2
       3: dup
       4: invokespecial #1
       7: astore_1
       8: aload_1
       9: dup
      10: astore_2
      11: monitorenter
      12: aload_2
      13: monitorexit
      14: goto          22
```

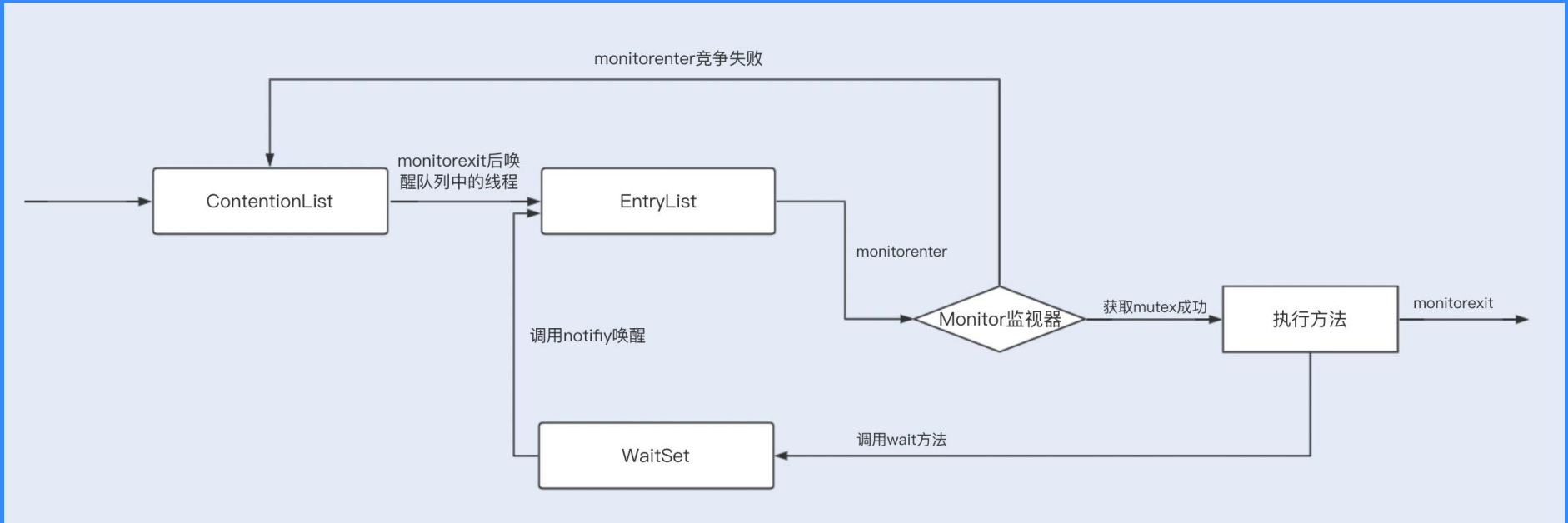
悲观锁：Synchronized 关键字

当 Synchronized 修饰同步方法时，并没有发现 monitorenter 和 monitorexit 指令。而是一个 ACC_SYNCHRONIZED 标志，放在JVM的常量池中。
这是因为 JVM 使用了 ACC_SYNCHRONIZED 访问标志来区分一个方法是否是同步方法。

```
// 关键字在方法上，锁为当前实例
  public synchronized void methodA() {
      // code
  }
```

```
public synchronized void methodA();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=0, locals=1, args_size=1
       0: return
    LineNumberTable:
      line 8: 0
```

Synchronized 关键字修饰方法如何加锁

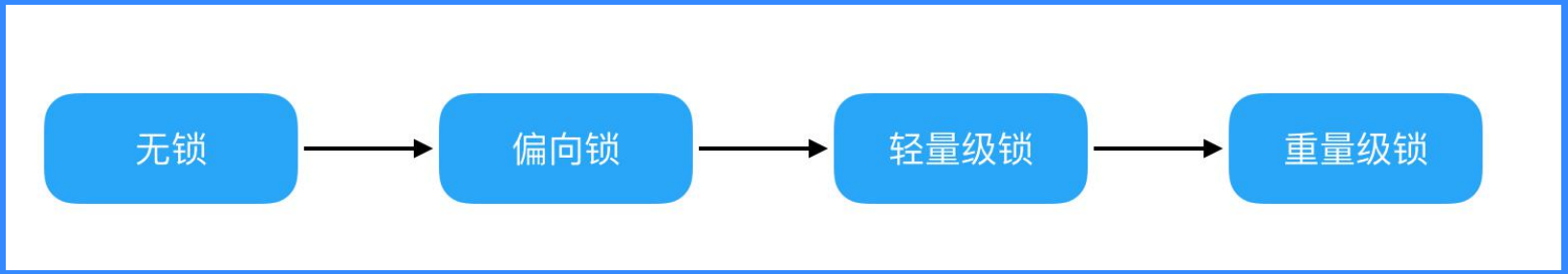


当多个线程同时访问一段同步代码时，多个线程会先被存放在 ContentionList 和 _EntryList 集合中，处于 block 状态的线程，都会被加入到该列表。

如果线程调用 wait() 方法，就会释放当前持有的 Mutex 互斥锁，并且该线程会进入 WaitSet 集合中，等待下一次被唤醒。如果当前线程顺利执行完方法，也将释放 Mutex。

JDK1.6 引入了偏向锁、轻量级锁、重量级锁概念

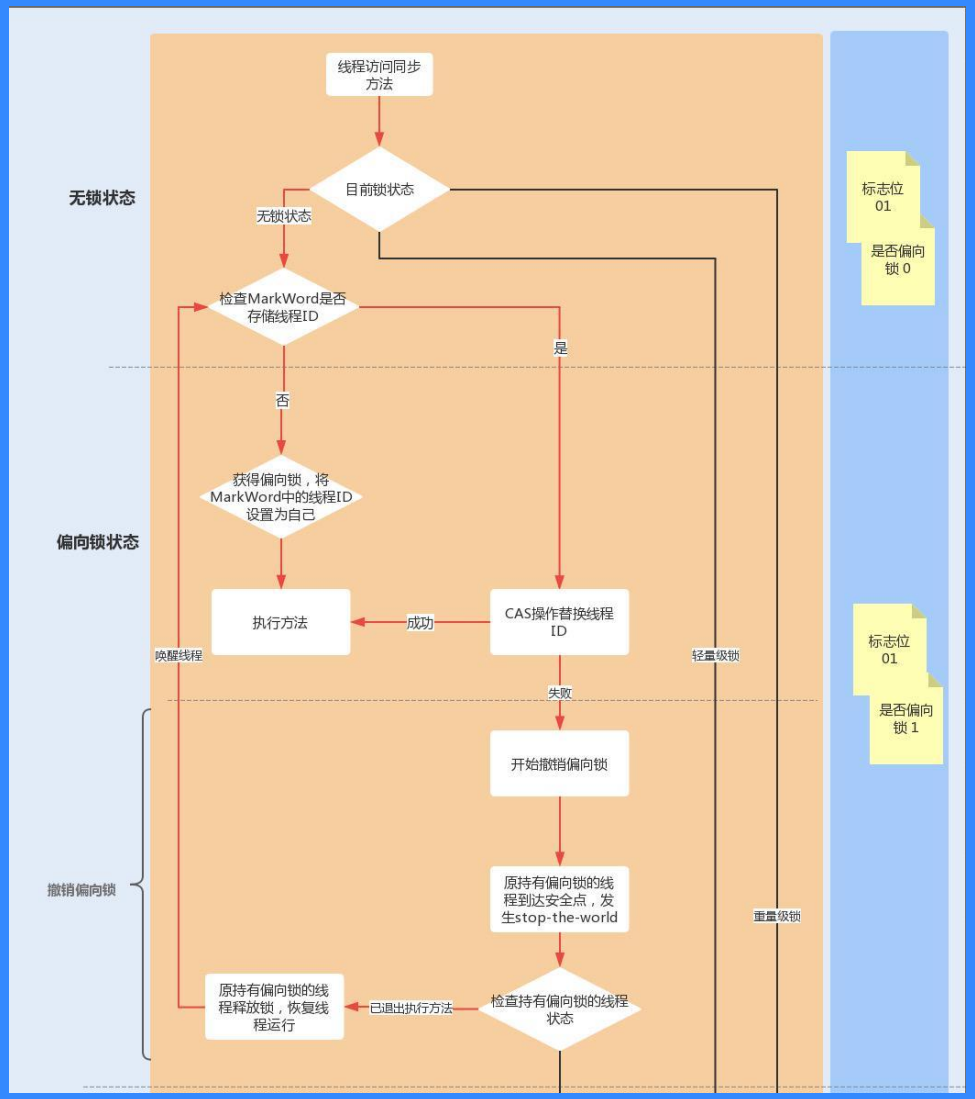
Synchronized 锁就是从偏向锁开始的，随着竞争越来越激烈，偏向锁升级到轻量级锁，最终升级到重量级锁。

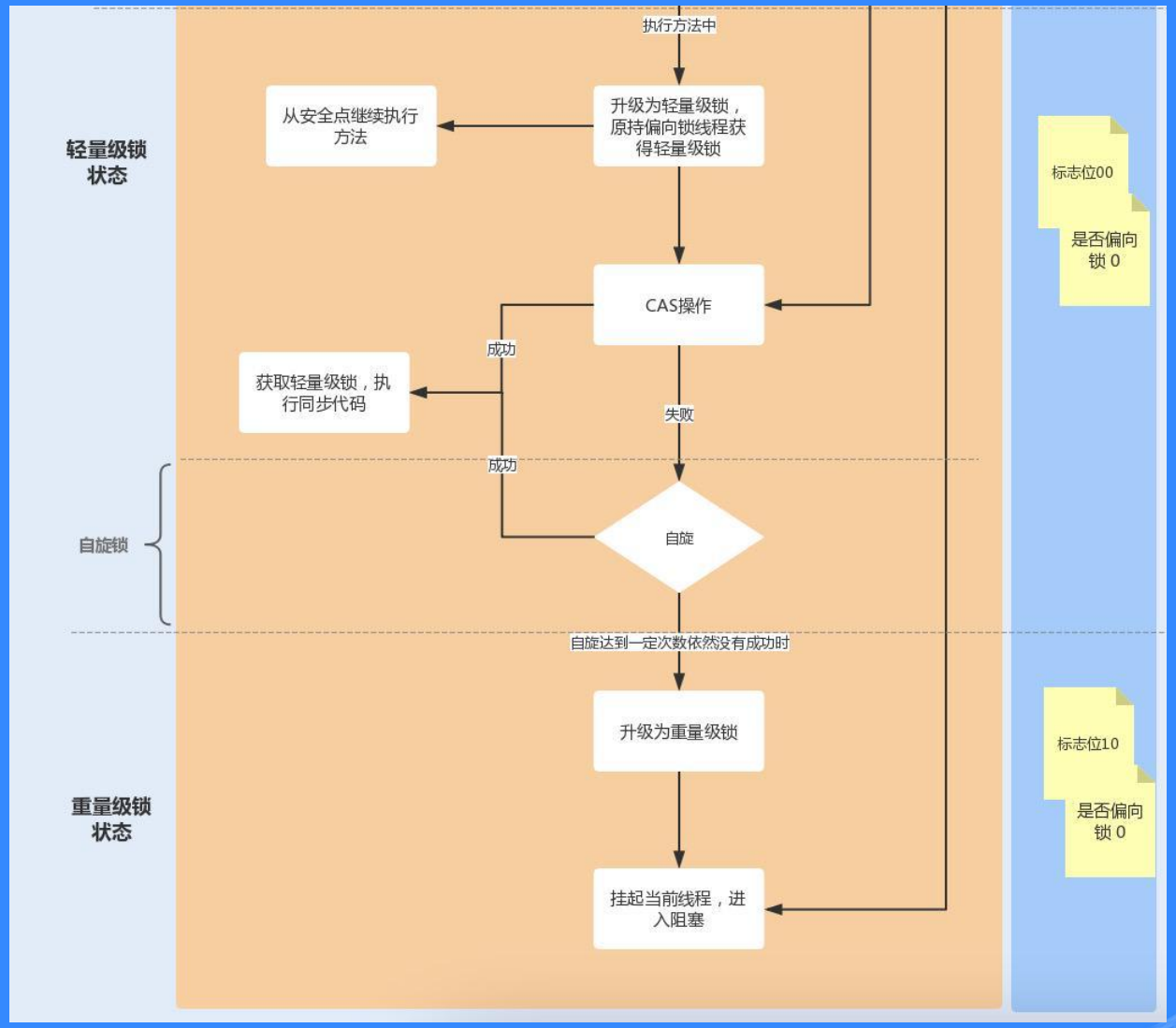


偏向锁：主要用来优化同一线程多次申请同一个锁的竞争。在某些情况下，大部分时间是同一个线程竞争锁资源，例如，单线程操作一个线程安全集合时，同一线程每次都需要获取和释放锁，每次操作都会发生用户态与内核态的切换。

如果抢锁失败,升级轻量级锁状态

Java 锁粒度升级机制：偏向锁、轻量级锁、自旋锁、重量级锁





Lock 并发同步锁

JDK1.5 后，Java 提供了Lock 同步锁：java.util.concurrent.locks包下常用的类与接口
那么它有什么优势呢？相对于需要 JVM 隐式获取和释放锁的 Synchronized 同步锁，Lock 同步锁显示获取和释放锁，这就为主动获取和释放锁提供了更多的灵活性。

Synchronzed VS Lock

维度	Synchronized	Lock
实现	Java关键字、JVM层实现	Java底层
锁的获取	隐式获取	Lock.lock() 获取锁，如锁定一直等待 Lock.trylock(timeout) 尝试获取锁，未获取不等待锁，避免死锁
锁的释放	隐式释放 A、获取锁的线程执行完同步代码，释放锁 B、线程异常，JVM主动释放锁	通过 Lock.unlock()主动释放 一定要释放，要不死锁
类型	非公平锁、可重入	可重入、非公平、公平
性能	在并发量不高、竞争不激烈的情况下，Synchronized 同步锁由于具有分级锁的优势，性能上与 Lock 锁差不多	但在高负载、高并发的情况下，Synchronized 同步锁由于竞争激烈会升级到重量级锁，性能比起 Lock 差很多

Lock 并发同步锁

独占锁：ReentrantLock 是一个独占锁，同一时间只允许一个线程访问。
场景：并发写

读写锁：ReadWriteLock 简称RRW 允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同时访问。读写锁内部维护了两个锁，一个是用于读操作的 ReadLock，一个是用于写操作的 WriteLock
场景：A、并发读 B、并发读写，读远大于写

要进一步提升并发执行效率，Java 8引入了新的读写锁：StampedLock。
StampedLock和ReadWriteLock相比，改进之处在于：读的过程中也允许获取写锁后写入！这样一来，我们读的数据就可能不一致，所以，需要一点额外的代码来判断读的过程中是否有写入，这种读锁是一种乐观锁：乐观地估计读的过程中大概率不会有写入。

读写锁使用案例：缓存对象的使用场景

缓存对象在使用时，一般并发读的场景远远大于并发写的场景，封装缓存对象是非常适合适用

```
class CachedData {
   //被缓存的具体对象
   Object data;
   //当前对象是否可用，使用volatile来保证可见性
   volatile boolean cacheValid;
   //今天的主角，ReentrantReadWriteLock
   final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
   void processCachedData() {
     //要读取数据时，先加读锁，如果加成功，说明此时没有人在并发写
     rwl.readLock().lock();
     //拿到读锁后，判断当前对象是否有效
     if (!cacheValid) {
       // Must release read lock before acquiring write lock
       //这里的处理非常经典，当你持有读锁之后，不能直接获取写锁，因为写锁是独占锁，如果直接获取写锁，那代码就在这里死锁了，所以必须要先释放读锁，然后手动获取写锁
       rwl.readLock().unlock();
       rwl.writeLock().lock();
       try {
         //经典处理之二，在独占锁内部要处理数据时，一定要做二次校验，因为可能同时有多个线程全都在获取写锁，
         //当时线程1释放写锁之后，线程2马上获取到写锁，此时如果不做二次校验那可能就导致某些操作做了多次
         if (!cacheValid) {
           data = ...
           //当缓存对象更新成功后，重置标记为true
           cacheValid = true; }
         //这里有一个非常神奇的锁降级操作，所谓降级是说当你持有写锁后，可以再次获取读锁，这里之所以要获取一次读锁是为了防止当前线程释放写锁之后，其他线程马上获取到写锁，改变缓存对象
         //因为读写互斥，所以有了这个读锁之后，在读锁释放之前，别的线程是无法修改缓存对象的
         rwl.readLock().lock();
       } finally { rwl.writeLock().unlock(); // Unlock write, still hold read
       }}
     try {  use(data); //读数据前，防止写入
     } finally {rwl.readLock().unlock(); } } }
```

同步锁性能对比概览

Synchronized、ReenTrantLock、ReentrantReadWriteLock

java version "1.6.0_29"
Java(TM) SE Runtime Environment (build 1.6.0_29-b11)
Java HotSpot(TM) 64-Bit Server VM (build 20.4-b02, mixed mode)

单纯的加载获取
读+ 写 ： 200

	读等于写 100读 + 100写	读多于写 180读+20写	读小于写 20读+180写
synchronized	read 20099 write:11639	read :12053 write:11556	read :16268 write :15291
ReentrantLock	read 14270 write :15381	read :30616 write 20547	read 10979 write :14298
ReentrantReadWriteLock	read:48137 write:16932	read :21545 write:19466	read :40149 write:13118

对比上面两组数据可以看出，在读写比相等的情况下，sync无论读写都比采用ReentrantReadWriteLock更有优势。当然我们这里前提条件是线程执行时间很短，因为synchronized的contentionlist采用的是自旋锁，故完全发挥了自旋锁的优点，并且也没有多CPU，SMP架构下cache 一致性流量的问题。占据绝对的优势，若是我们的实际使用是这种情况的话，那么sync是最佳选择！

读写方法有一定的睡眠时间，即在调用读和写方法后，在方法中睡眠一段时间(测试的是睡眠10毫秒)
读+写 ： 200

	读等于写 100读 + 100写	读多于写 180读+20写	读小于写 20读+180写
synchronized	read :1050043530 write:1066386015	read 1072334901 write:1065810347	read 1946124283 write:985446749
ReentrantLock	read: 526311265 write :533492549	read :974188775 write :104943616	read :115537464 write :989321366
ReentrantReadWriteLock	read :10289762 write :541641594	read :13995793 write :107608746	read :18237165 write :986742164

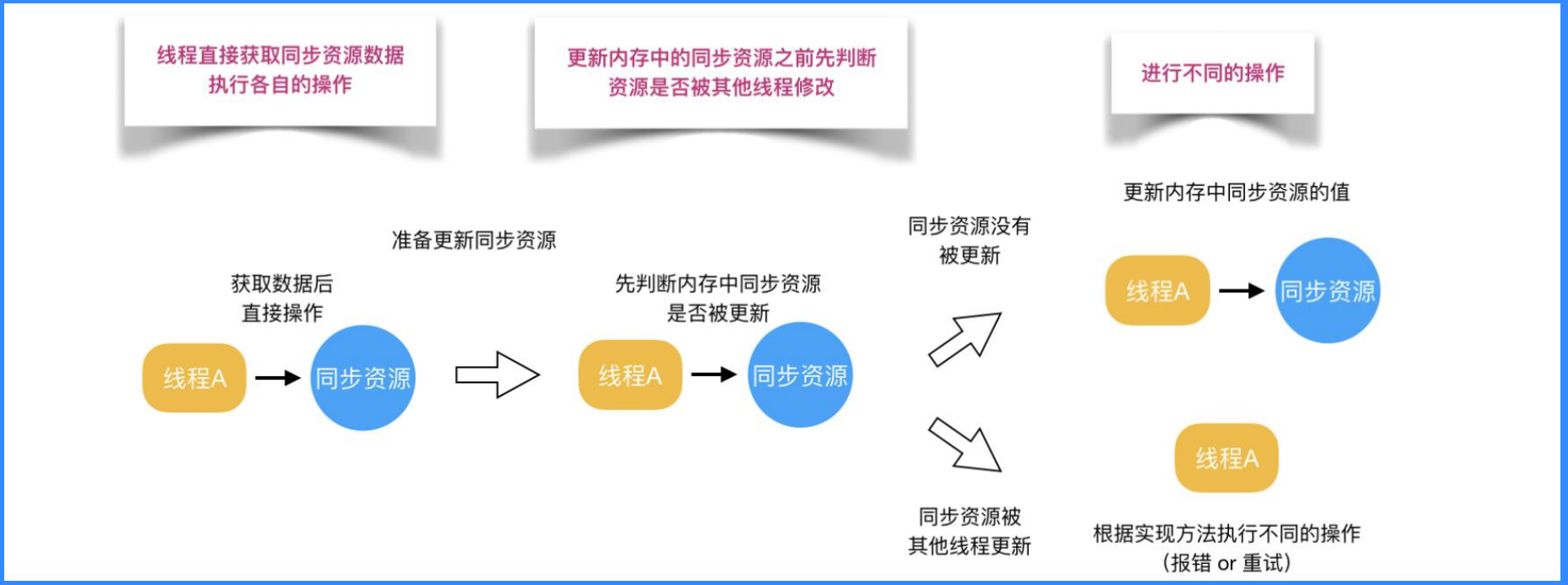
对比上面的数据，我们可以清晰的看出ReentrantReadWriteLock在有一定业务开销的情况下，性能有明显的优势，特别是对于一些我们实际的业务场景，有网络IO，磁盘IO等，那么采用读写锁应该是比较好的选择！

乐观锁：一种思想

乐观锁，一种思想。

相比悲观锁来说，不会带来死锁、饥饿等活性故障问题，线程间的相互影响也远远比悲观锁要小。更为重要的是，乐观锁没有因竞争造成的系统开销，所以在性能上也是更胜一筹。

CAS：Compare-and-swap ： 乐观锁常见原子操作的实现，有名的无锁（lock-free）算法
乐观锁另一典型实现：版本号，比如数据库应对并发读写



CAS无锁应用：原子操作Atomic

```
//volatile变量value
private volatile int value;
```

```
//方法相当于原子性的 ++i
public final int getAndIncrement() {
    //三个参数，1、当前的实例 2、value实例变量的偏移量 3、递增的值。
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
//方法相当于原子性的 --i
public final int getAndDecrement() {
    //三个参数，1、当前的实例 2、value实例变量的偏移量 3、递减的值。
    return unsafe.getAndAddInt(this, valueOffset, -1);
}
```

```
//native硬件级别的原子操作
//类似的有compareAndSwapInt,compareAndSwapLong,compareAndSwapBoolean,compareAndSwapCh
public final native boolean compareAndSwapInt(Object o, long offset,int expected,int

//内部使用自旋的方式进行CAS更新（while循环进行CAS更新，如果更新失败，则循环再次重试）
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        //获取对象内存地址偏移量上的数值v
        v = getIntVolatile(o, offset);
        //如果现在还是v,设置为 v + delta,否则返回false,继续循环再次重试.
    } while (!compareAndSwapInt(o, offset, v, v + delta));
    return v;
}
```

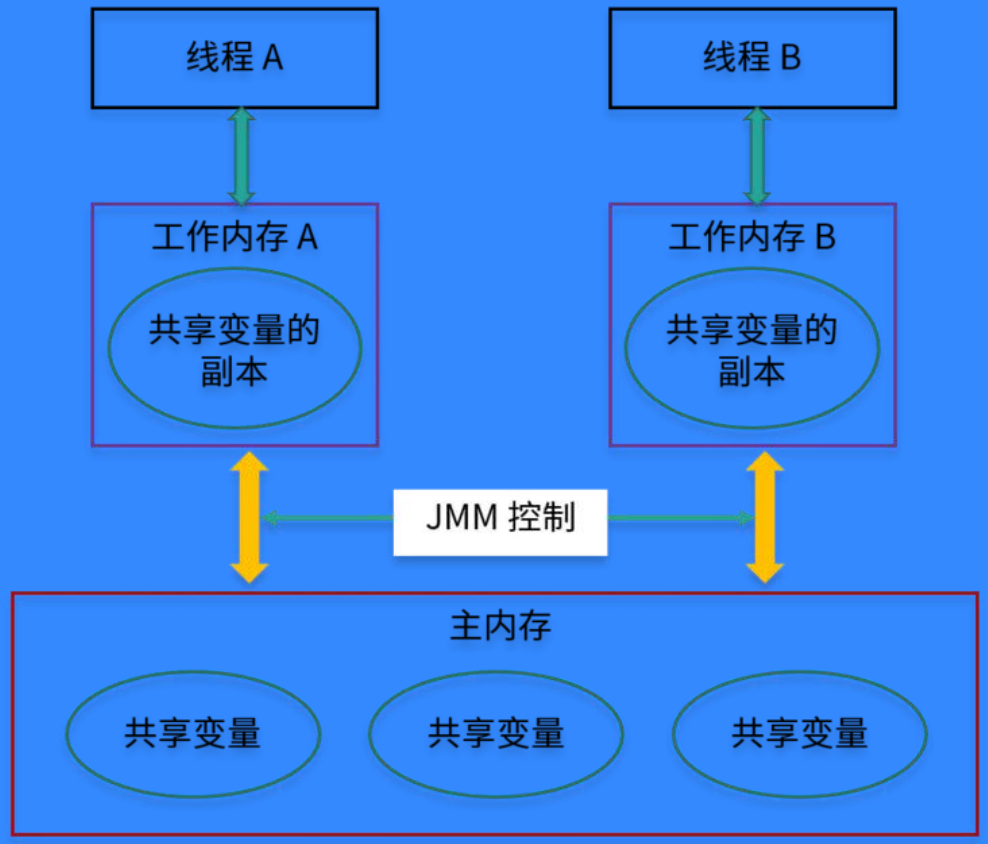
getAndAddInt()循环获取给定对象o中的偏移量处的值v，然后判断内存值是否等于v。如果相等则将内存值设置为 v + delta

整个“比较+更新”操作封装在compareAndSwapInt()中，在JNI里是借助于一个CPU指令完成的，属于原子操作，可以保证多个线程都能够看到同一个变量的修改值。

CAS 是调用处理器底层指令来实现原子操作
那么处理器底层又是如何实现原子操作的呢？

缓存锁定机制

Java 内存模型和volatile



Java 逻辑内存模型

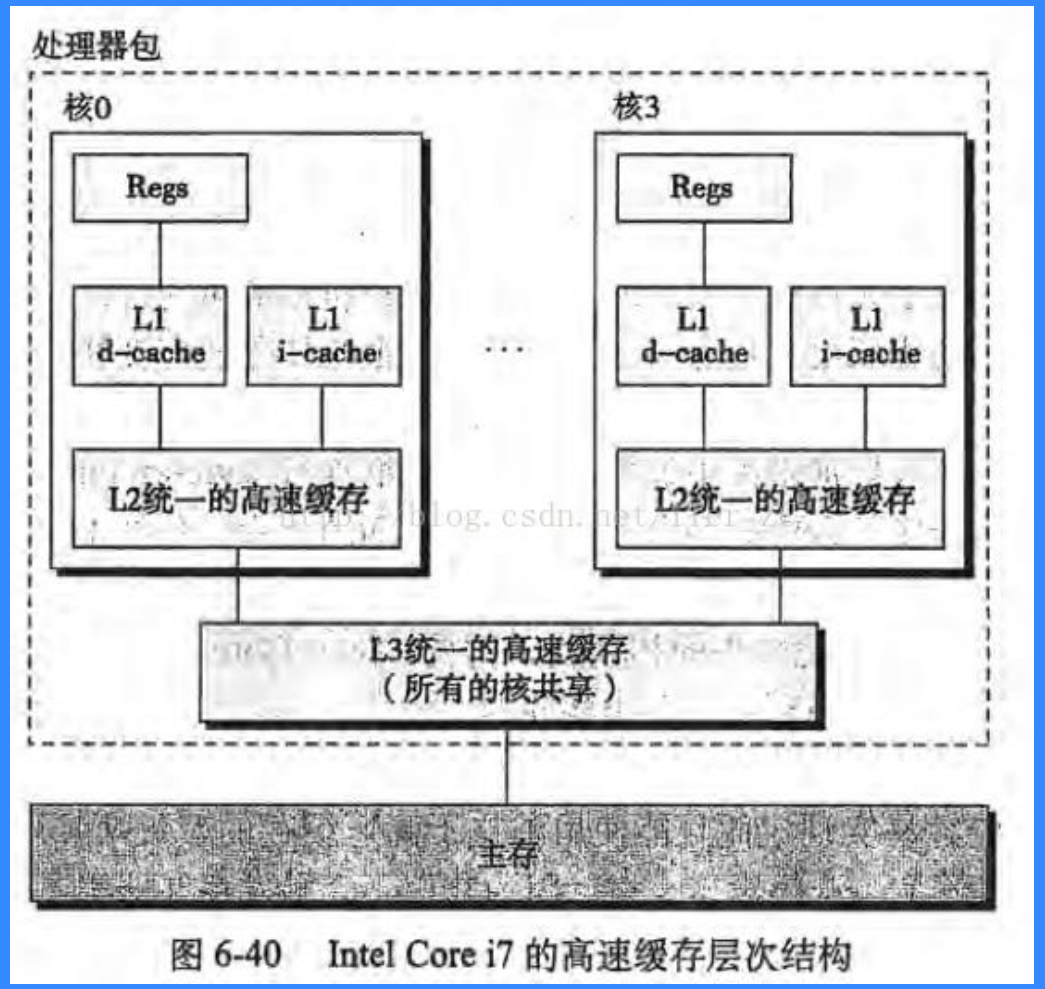


图 6-40 Intel Core i7 的高速缓存层次结构

Java Volatile 的底层原理

内存屏障（memory barrier）是一个CPU指令。它是这样一条指令： 确保一些特定操作执行的顺序和控制一些数据的可见性
编译器和CPU可以在保证输出结果一样的情况下对指令重排序，使性能得到优化。

被volatile关键字修饰的变量，在每个写操作之后，都会加入一条store内存屏障命令，此命令强制将此变量的最新值从工作内存同步至主内存；


```
汇编
lock addl $0x0,(%rsp)
IA32 中对 lock 的说明是
The LOCK # signal is asserted during execution of the instruction following the lock prefix. This signal can be used in a multiprocessor system to ensure exclusive use of shared memory while LOCK # is asserted
LOCK 用于在多处理器中执行指令时对共享内存的独占使用。它的作用是能够将当前处理器对应缓存的内容刷新到内存，并使其他处理器对应的缓存失效。另外还提供了有序的指令无法越过这个内存屏障的作用。
正是 lock 实现了 volatile 的「防止指令重排」「内存可见」的特性
```

乐观锁 CAS

CAS 乐观锁在高并发读写足够好的性能表现，高并发读大于写

那么写大于读的场景下，CAS还是会导致线程失败不断重试CAS操作，比如上面提到原子操作

大部分线程的原子操作会失败，失败后的线程将会不断重试 CAS 原子操作，这样就会导致大量线程长时间地占用 CPU 资源，给系统带来很大的性能开销。
LongAdder，它使用了空间换时间的方法，解决了上述问题。

LongAdder 的原理就是降低操作共享变量的并发数，也就是将对单一共享变量的操作压力分散到多个变量值上，将竞争的每个写线程的 value 值分散到一个数组中，不同线程会命中到数组的不同槽中，各个线程只对自己槽中的 value 值进行 CAS 操作，最后在读取值的时候会将原子操作的共享变量与各个分散在数组的 value 值相加，返回一个近似准确的数值。

写大于读高并发还有其他性能优化策略：比如常见读写分离技术

线程池：为什么要有线程池

线程本身创建和销毁有一定性能开销

在JVM 的线程模型中，Java 线程被一对一映射为内核线程。Java 在使用线程执行程序时，需要创建一个内核线程；当该 Java 线程被终止时，这个内核线程也会被回收。因此 Java 线程的创建与销毁将会消耗一定的计算机资源，从而增加系统的性能开销。
大量创建线程同样会给系统带来性能问题，对于频繁创建线程的业务场景，线程池可以创建固定的线程数量，并且在操作系统底层，轻量级进程将会把这些线程映射到内核

Java线程很强大、灵活。但是带来维护难度和风险

线程自维护技术门槛很高，因为工程师水平和认知差异，线程创建和销毁，如果稍有不慎，很容易产生内存泄露，CPU负荷问题。特别高并发情况，线程对性能影响非常大，优化线程的管理非常必要

避免重复造轮子，专业的线程统一管理，大大节约开发、维护成本。不仅Java提供基本线程池，开源线程池管理框架也很多，功能、性能也更好。

定位：大部分人、大部分场景都是使用线程池，一般能不自己维护线程生命周期，就交给线程池管理，从大多数场景来说，熟悉和精通线程池参数调优就很重要

Java自带的常用线程池

Java 最开始提供了 ThreadPool 实现了线程池，为了更好地实现用户级的线程调度，更有效地帮助开发人员进行多线程开发，Java 提供了一套 Executor 框架
这个框架中包括了 ScheduledThreadPoolExecutor 和 ThreadPoolExecutor 两个核心线程池。前者是用来定时执行任务，后者是用来执行被提交的任务

类型	特征	场景
newFixedThreadPool	固定大小的线程池，有新任务提交时，线程池如果有空闲线程，立即执行。否则新的任务会被缓存在一个任务队列中，等待线程池释放空闲线程	控制最大线程来使服务器最大的使用率，同事保证及时流量突然增大也不会超过服务器最大的负载 比如文件上传服务，带宽，CPU能力有限
newCachedThreadPool	如果线程池中的线程数量过大，它可以有效的回收多余的线程，如果线程数不足，那么它可以创建新的线程。	不足：根据业务场景自动的扩展线程数来处理业务，线程数无法控制的； 优点：第一个任务已经执行结束，第二个任务会复用第一个任务创建的线程，并不会重新创建新的线程，提高了线程的复用率； 频繁的连接数据库，然而创建连接是一个很消耗性能的事情，所有数据库连接池就出现了
newSingleThreadExecutor	单线程池，至始至终都由一个线程来执行	优点：如果线程失败异常退出，会自动创建新线程重试
newScheduledThreadPool	定时线程池，支持定时、周期任务执行	推荐更成熟、强大的定时框架， 灵活动态配置：XX-job、Spring Quartz

线程池：自定义一套线程池配置

透过现象看本质

避免线程池不正确用法

阿里狠的做法

4. 【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

1）FixedThreadPool 和 SingleThreadPool：

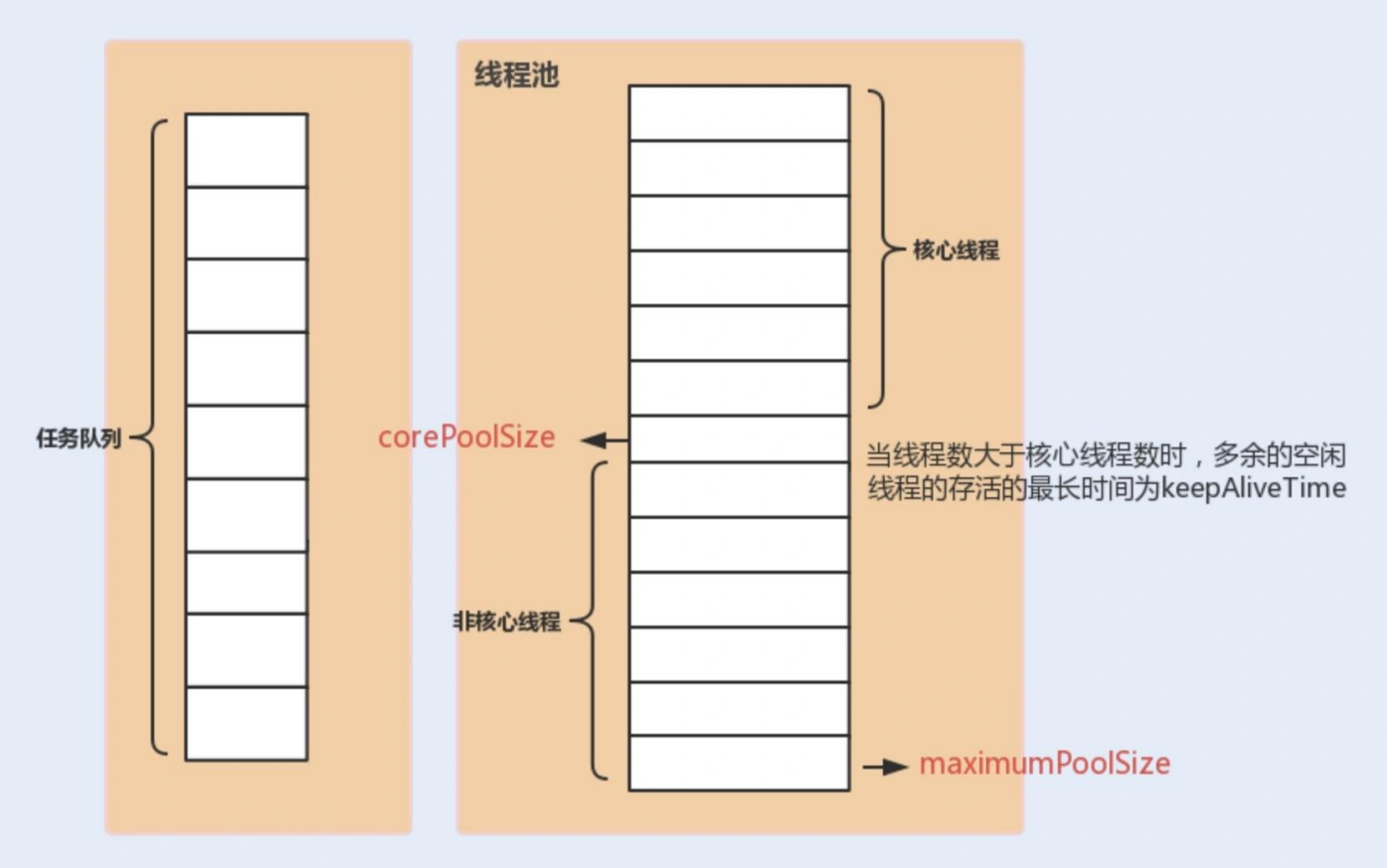
允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。

2）CachedThreadPool：

允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

```
public ThreadPoolExecutor(int corePoolSize,//线程池的核心线程数量
                          int maximumPoolSize,//线程池的最大线程数
                          long keepAliveTime,//当线程数大于核心线程数时，多余的空闲线程存活的最长时间
                          TimeUnit unit,//时间单位
                          BlockingQueue<Runnable> workQueue,//任务队列，用来储存等待执行任务的队列
                          ThreadFactory threadFactory,//线程工厂，用来创建线程，一般默认即可
                          RejectedExecutionHandler handler)
                          //拒绝策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务
```

线程池核心参数：



线程池都有哪几种工作队列

1、ArrayBlockingQueue是一个基于数组结构的有界阻塞队列
2、LinkedBlockingQueue一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素。Executors.newFixedThreadPool()使用了这个队列
3、SynchronousQueue Executors.newCachedThreadPool使用了这个队列。

RejectedExecutionHandler

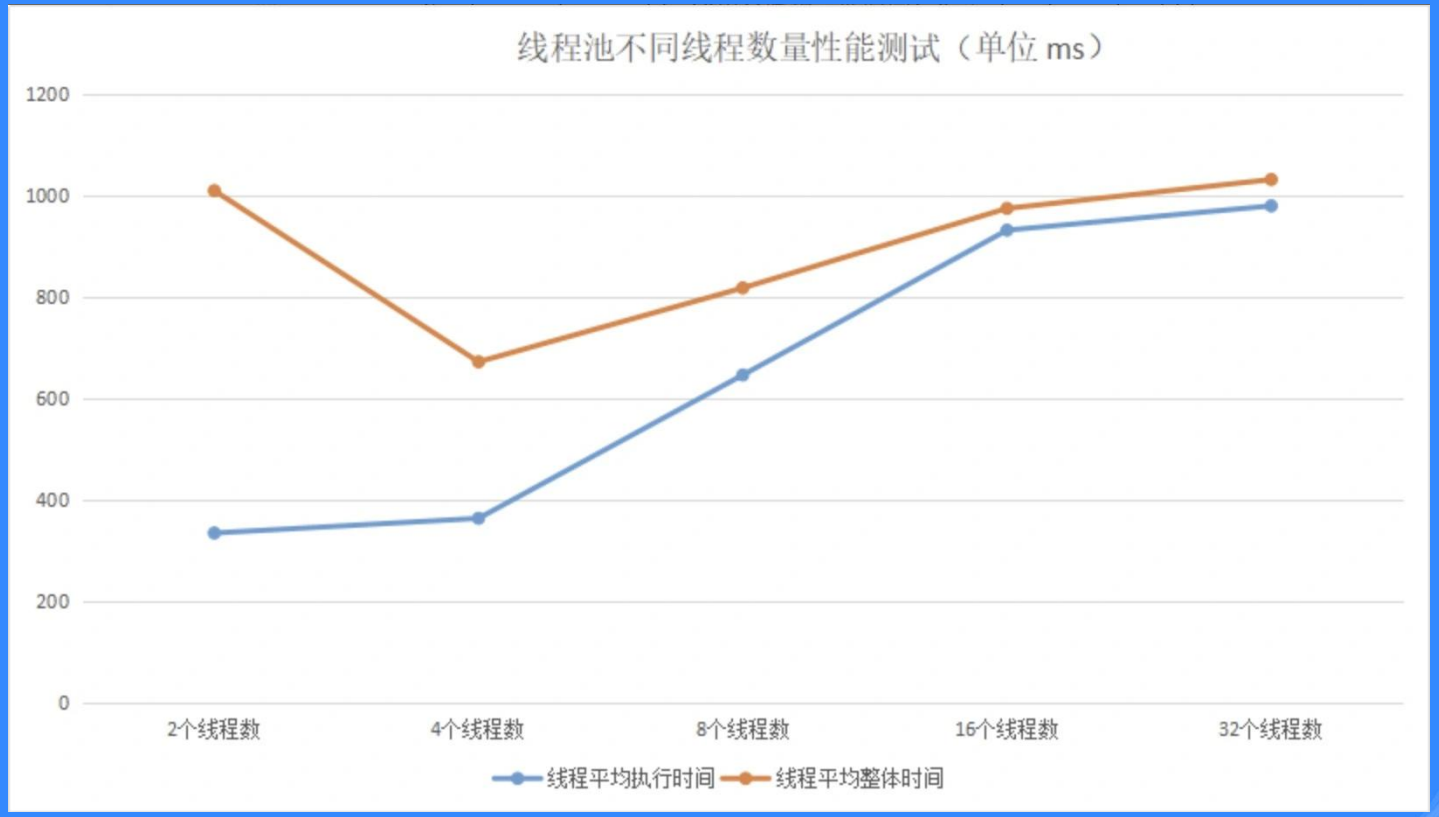
线程池对处理不过来的任务的拒绝执行策略
AbortPolicy
DiscardOldestPolicy

线程池核心参数：线程数设置

CPU 密集型任务 这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N（CPU 核心数）+ 1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

I/O 密集型任务 系统会用大部分的时间来处理 I/O 交互，大致初始从 2N调整开始，不断区间微调

IO 通信 并发连接数是根本瓶颈。线程数也受影响。比如数据库连接池限制，默认线程数也要考虑进来



CPU密集运算的程序在 4 核 intel i5 CPU 运行时间变化

Java 定时线程池和开源定时框架适用场景

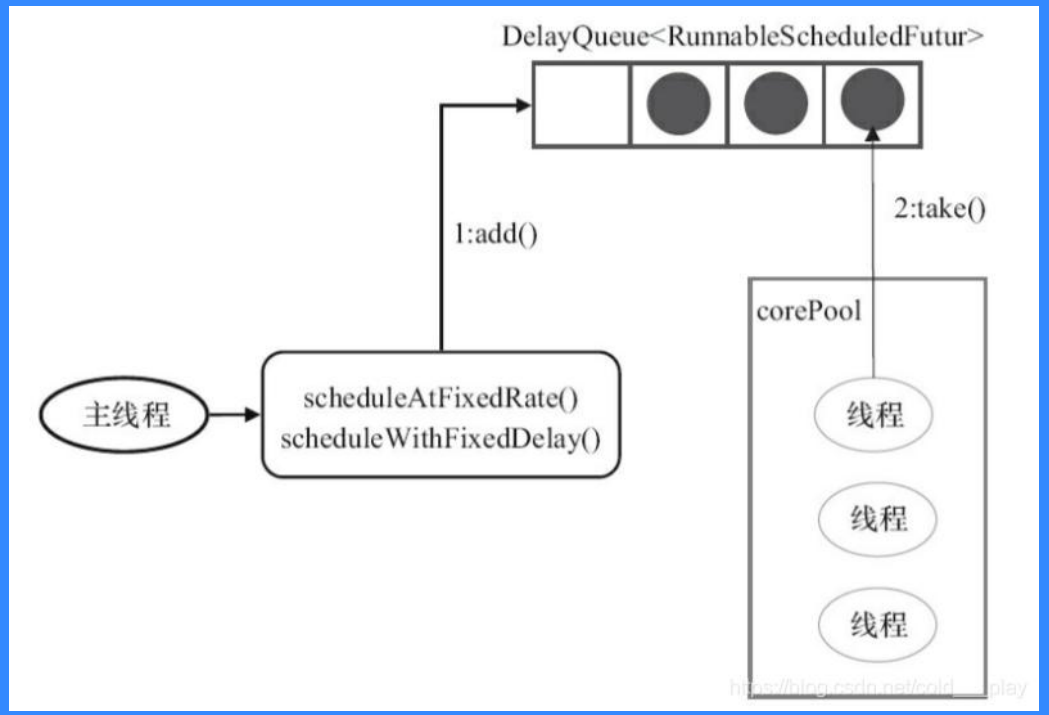
自JDK1.5开始，JDK提供了ScheduledThreadPoolExecutor类来支持周期性任务的调度。在这之前的实现需要依靠Timer和TimerTask或者其它第三方工具来完成

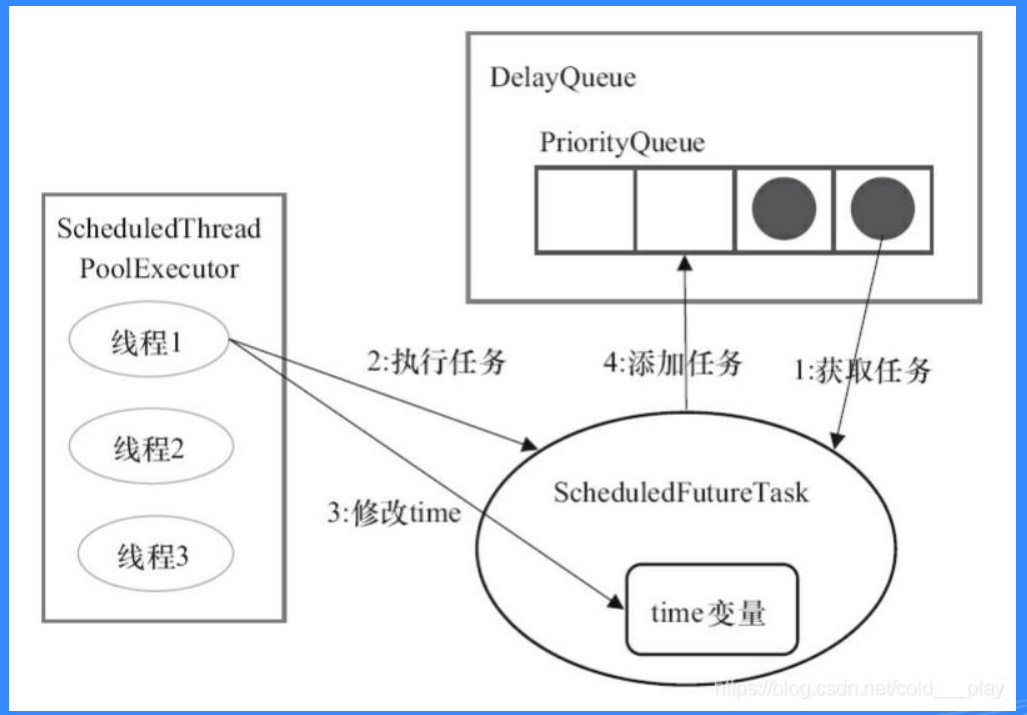
Timer	ScheduledThreadPoolExecutor
单线程	多线程
单个任务执行时间影响其他任务调度	多线程，不会影响
基于绝对时间	基于相对时间
一旦执行任务出现异常不会捕获，其他任务得不到执行	多线程，单个任务的执行不会影响其他线程

JDK1.5之后，应该没什么理由继续使用Timer进行任务调度了

ScheduledThreadPoolExecutor 定时程序运行原理

DelayQueue是一个无界队列
1、将任务添加队列中，它是异步的
2、线程池中的线程从DelayQueue队列中获取任务，然后执行异步任务
定时跑任务
1、线程从队列中获取已到期的ScheduledFutureTask，到期任务是指它的time大于等于当前时间
2、线程执行这个ScheduledFutureTask，完成后修改它的time变量为下次被执行的时间
3、线程把这个修改time之后的ScheduledFutureTask放回DelayQueue中





Java 定时线程池和开源定时框架适用场景

项目使用中，考虑我们的需求、痛点，Java原生Schedule 定时框架远远不够

1、定时程序复杂多变，希望能动态配置，支持热部署

2、希望管理简单，学习门槛低。同时支持高可用、故障告警、监控运维等功能

特性	quartz	elastic-job	xxl-job
高可用	多节点部署，通过竞争数据库锁来保证只有一个节点执行任务	通过zookeeper的注册与发现，可以动态的添加服务器	基于竞争数据库锁保证只有一个节点执行任务，支持水平扩容。可以手动增加定时任务，启动和暂停任务，有监控
任务分片	×	√	√
管理界面	×	√	√
难易程度	简单	简单	简单
高级功能	-	弹性扩容，故障转移，运行状态收集，幂等性，动态控制定时任务	弹性扩容，故障转移，Rolling实时日志，动态控制定时任务，任务进度监控，邮件报警，运行报表等

开源定时框架提供

XX-Job 效果

动态配置
自定义控制
故障迁移

进度监控
运行报表
报警支持

上下文原理

进程和线程

程序由指令和数据组成，但这些指令要运行，数据要读写，就必须将指令加载至 CPU，数据加载至内存。在指令运行过程中还需要用到磁盘、网络等设备。进程就是用来加载指令、管理内存、管理 IO 的

进程就可以视为程序的一个实例。实例进程：浏览器，记事本。操作系统会以进程为单位，分配系统资源（CPU时间片、内存等资源），进程是资源分配的最小单位

线程是进程中的实体，一个进程可以拥有多个线程，一个线程必须有一个父进程。 一个线程就是一个指令流，将指令流中的一条条指令以一定的顺序交给 CPU 执行 。 线程，有时被称为轻量级进程(Lightweight Process，LWP），是操作系统调度（CPU调度）执行的最小单位

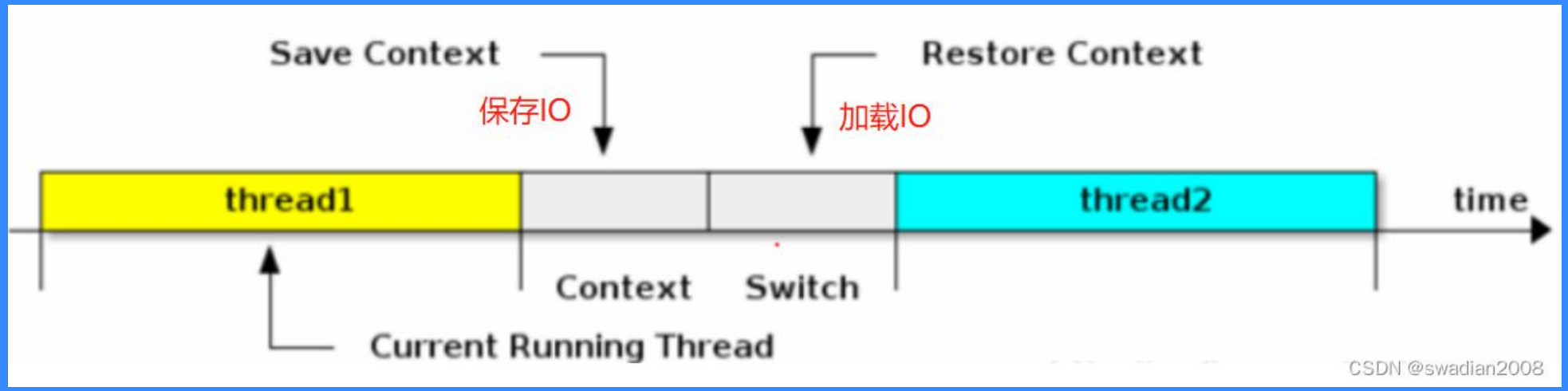
上下文切换的类型还可以分为进程间的上下文切换和线程间的上下文切换

重点说线程上下文切换

上下文原理

处理器给每个线程分配 CPU 时间片，线程在分配获得的时间片内执行任务
CPU 时间片一般几十毫秒
当一个线程的时间片用完了，或者因自身原因被迫暂停运行了，这个时候，另外一个线程（相同进程或其它进程的线程）就会被操作系统选中，来占用处理器

这种一个线程被暂停使用权，另一线程被选中开始或者继续运行的过程就叫做上下文切换（Context Switch）
切出：一个线程被剥夺处理器的使用权而被暂停运行
切入：一个线程被选中占用处理器开始或者继续运行
在这种切出切入的过程中，操作系统需要保存和恢复相应的进度信息，这个信息就是“上下文”

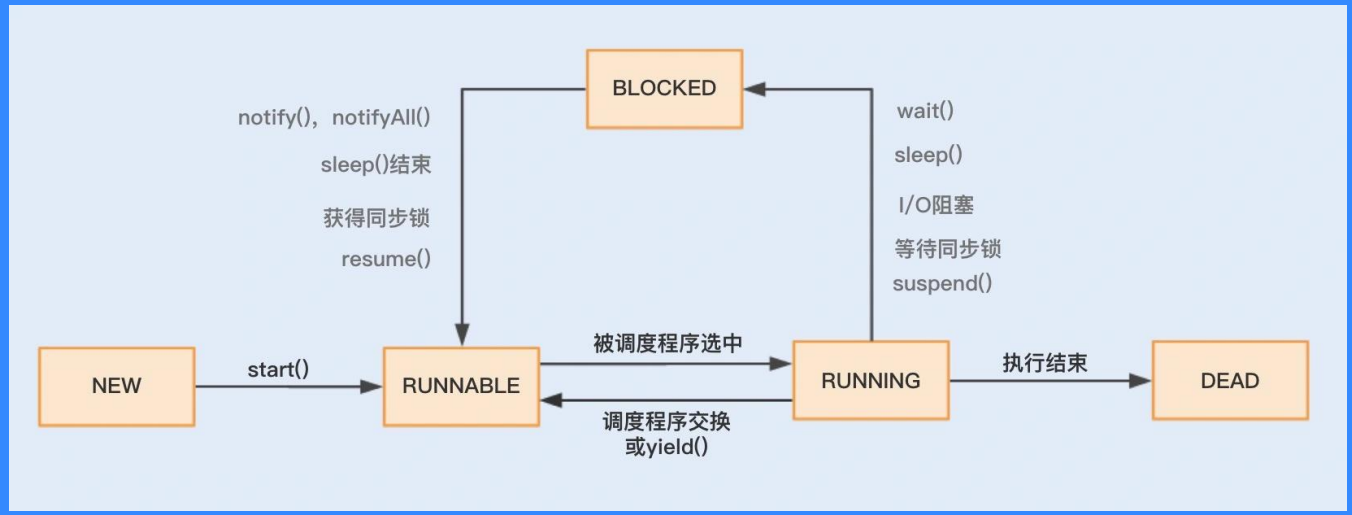


Java 多线程上下文切换场景

Java 线程上下文切换表现

线程由 RUNNABLE 转为非 RUNNABLE 的过程就是做上下文切换

多线程的上下文切换实际上就是由多线程两个运行状态的互相切换导致的



有两种情况会产生线程上下文切换
1、一种是程序本身触发的切换，称为自发性上下文切换：在多线程编程中，执行sleep() 方法或 synchronized 关键字，就会引发自发性上下文切换
2、由系统或者虚拟机诱发的非自发性上下文切换
常见的有：线程被分配的时间片用完，虚拟机垃圾回收导致或者执行优先级的问题

优化Java 线程上下文切换

多线程对锁资源的竞争会引起上下文切换，锁竞争导致的线程阻塞越多，上下文切换就越频繁，系统的性能开销也就越大，如何减少上下文切换？

减少锁的持有时间 Synchronized 尽量同步最少代码块和方法

降低锁的粒度 同步锁可以保证对象的原子性，我们可以考虑将锁粒度拆分得更小一些，以此避免所有线程对一个锁资源的竞争过于激烈。具体方式有以下两种
1、锁分离：与传统锁不同的是，读写锁实现了锁分离
2、锁分段我们在使用锁来保证集合或者大对象原子性时，可以考虑将锁对象进一步分解

合理地设置线程池大小 避免创建过多线程线程池的线程数量设置不宜过大，因为一旦线程池的工作线程总数超过系统所拥有的处理器数量，就会导致过多的上下文切换

非阻塞乐观锁替代竞争锁
Volatile 关键字的作用是保障可见性及有序性，Volatile 的读写操作不会导致上下文切换，因此开销比较小。
CAS保证原子性，避免用锁

上下文优化

多线程上下文切换案例

```
System.out.println(" 并发线程数: "+threadCount+"  开始启动咯！");
// 创建多个线程
for (int i = 0; i < threadCount; i++) {
  threads[i] = new Thread(myRunnable1);
  threads[i].start();
}
for (int i = 0; i < threadCount; i++) {
    // 等待一起运行完
    threads[i].join();
}

// 创建一个单线程
static class SerialTester extends ThreadContextSwitchTester {
  public void Start() {
    for (long i = 0; i < count; i++) {
      increaseCounter(); }}}

public volatile int counter = 0;
public void increaseCounter() {
  this.counter += 1; }
```

逻辑内核CPU为4

```
[root@community-demo project]# cat /proc/cpuinfo | grep "processor" | wc -l
4
[root@community-demo project]#
```

```
----------------------- 上下文切换开始，默认等待30s ---------
 并发线程数: 2  开始启动咯！
multi thread ,the thread count is  2 | exce time: 9843s
counter: 100000000

----------------------- 默认等待30s ---------
 并发线程数: 4  开始启动咯！
multi thread ,the thread count is  4 | exce time: 4045s
counter: 100000000
----------------------- 默认等待30s ---------
 并发线程数: 8  开始启动咯！
multi thread ,the thread count is  8 | exce time: 4289s
counter: 100000000

----------------------- 默认等待30s ---------
 并发线程数: 32  开始启动咯！
multi thread ,the thread count is  32 | exce time: 4502s
counter: 100000000
serial exec time: 1101s
counter: 100000000
```

线程数接近逻辑CPU核数性能最好

没有上下文串行时间最快

切换线程数调整，上下文实际切换频率

上下文切换统计，VMstat 看

```
[[root@community-demo ~]# vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 6410524      0 789536    0    0     1     1    2   18  0  0 100  0  0
 0  0      0 6410500      0 789536    0    0     0     0  185  298  0  0 99  0  0
 0  0      0 6410500      0 789536    0    0     0     0  199  339  0  0 100  0  0
 0  0      0 6410500      0 789536    0    0     0     0  218  358  1  0 99  0  0
 0  0      0 6410500      0 789536    0    0     0     0  176  295  0  0 100  0  0
 0  0      0 6410500      0 789536    0    0     0     4  190  308  0  0 99  0  0
 2  0      0 6408048      0 789536    0    0     0     0 2229 9949 73 14 13  0  0
 2  0      0 6408048      0 789536    0    0     0     0 2645 14489 79 19  2  0  0
 2  0      0 6408048      0 789536    0    0     0    16 2578 12555 81 18  2  0  0
```

线程数2

```
0  0      0 6408576      0 789536    0    0     0     0  184  300  0  0 99  0  0
2  0      0 6408452      0 789536    0    0     0     0  407  771  5  0 94  0  0
2  0      0 6408388      0 789540    0    0     0     0 2672 4292 75  1 24  0  0
2  0      0 6408388      0 789540    0    0     0     0 2914 5881 70  1 29  0  0
2  0      0 6408388      0 789540    0    0     0     0 3094 7498 60  1 39  0  0
4  0      0 6408388      0 789540    0    0     0     0 2850 4262 80  1 19  0  0
0  0      0 6408388      0 789540    0    0     0     0 2096 2671 71  0 29  0  0
0  0      0 6408452      0 789540    0    0     0     0  215  364  1  0 99  0  0
0  0      0 6408452      0 789540    0    0     0     0  184  307  0  0 100  0  0
0  0      0 6408436      0 789556    0    0     0    15  194  316  0  1 99  0  0
0  0      0 6408436      0 789556    0    0     0     0  170  299  0  0 100  0  0
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
```

线程为4

问答时间

关注公众号 **OpenTelemetry**

Thanks