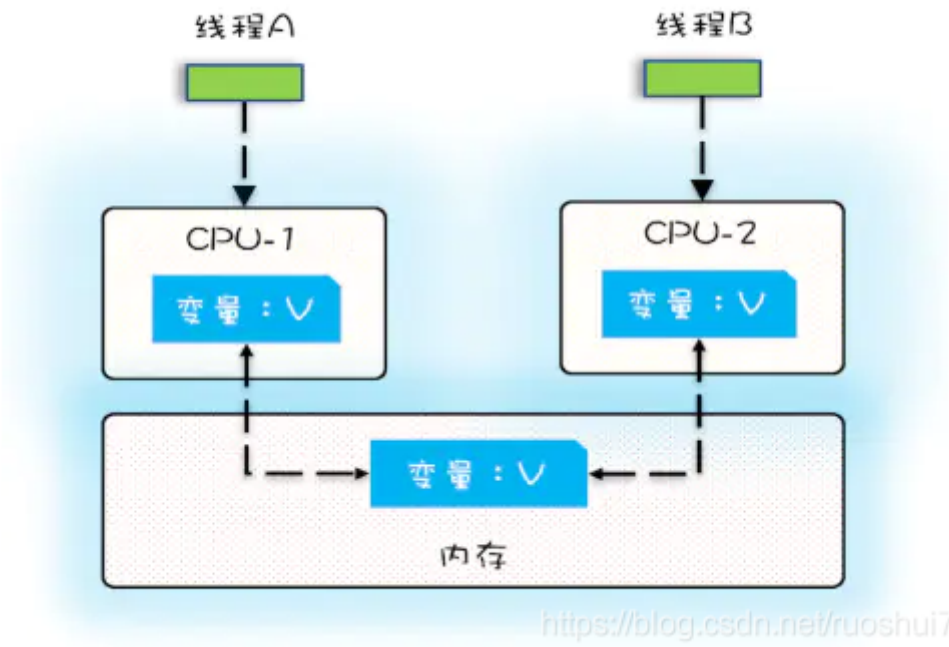
1.背景

互联网的快速发展，Java开发的过程或多或少会需要进行并发编程，也会遇到一些并发编程带来的各种bug。下面从并发编程的理论、并发工具类、并发设计模式、并发模型案例，记录一下自己的学习历程。

2.并发编程理论

2.1可见性、原子性、有序性



并发编程的来源于缓存导致的可见性问题，线程切换带来的原子性问题，编译优化带来的有序性问题，也就是并发编程需要遵循的三个原则。
可见性：一个线程对共享变量的修改，另外一个线程能够立刻看到
原子性：一个或者多个操作在 CPU 执行的过程中不被中断的特性
有序性：程序按照代码的先后顺序执行

2.2 Java内存模型

Java语言规范引入了Java内存模型，通过定义多项规则对编译器和处理器进行限制，主要是针对可见性和有序性。主要是通过volatile、synchronized 和 final 三个关键字，以及Happens-Before 规则。
（1）锁，锁操作是具备happens-before关系的，解锁操作happens-before之后对同一把锁的加锁操作。实际上，在解锁的时候，JVM需要强制刷新缓存，使得当前线程所修改的内存对其他线程可见。
（2）volatile字段，volatile字段可以看成是一种不保证原子性的同步但保证可见性的特性，其性能往往是优于锁操作的。但是，频繁地访问 volatile字段也会出现因为不断地强制刷新缓存而影响程序的性能的问题。
（3）final修饰符，final修饰的实例字段则是涉及到新建对象的发布问题。当一个对象包含final修饰的实例字段时，其他线程能够看到已经初始化的final实例字段，这是安全的。
Happpens-Before规则：
（1）程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
（2）管程锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作。这里必须强调的是同一个锁，而"后面"是指时间上的先后顺序。

（3）volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作，这里的"后面"同样是指时间上的先后顺序。
（4）线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。
（5）线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过Thread.join（）方法结束、Thread.isAlive（）的返回值等手段检测到线程已经终止执行。
（6）线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。
（7）对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的finalize()方法的开始。

2.3 互斥锁

互斥：同一个时刻只有一个线程在运行。锁是一种通用的技术方案，Java 语言提供的 synchronized 关键字，就是锁的一种实现。synchronized 关键字可以用来修饰方法，也可以用来修饰代码块。

```
class X {
  // 修饰非静态方法
  synchronized void foo() {
    // 临界区
  }

  // 修饰静态方法
  synchronized static void bar() {
    // 临界区
  }

  // 修饰代码块
  Object obj = new Object();
  void baz() {
    synchronized(obj) {
      // 临界区
    }
  }
}
```

当修饰静态方法的时候，锁定的是当前类的 Class 对象；当修饰非静态方法的时候，锁定的是当前实例对象 this。锁的本质是在锁定对象的头部字段写入锁定状态和线程信息，所以锁定的对象需要是一个不变的对象。
当用一把锁锁住多个资源，性能太差，会造成几个操作都是串行。所以可以用多把锁分别锁不同的资源，不同的操作可以并行操作。用不同的锁对受保护资源进行精细化管理，能够提升性能。这种锁还有个名字，叫细粒度锁。当然这样又会造成死锁的情况出现。要避免死锁就需要分析死锁发生的条件，有个叫 Coffman 的牛人早就总结过了，只有以下这四个条件都发生时才会出现死锁：
（1）互斥，共享资源 X 和 Y 只能被一个线程占用；
（2）占有且等待，线程 T1 已经取得共享资源 X，在等待共享资源 Y 的时候，不释放共享资源 X；
（3）不可抢占，其他线程不能强行抢占线程 T1 占有的资源；
（4）循环等待，线程 T1 等待线程 T2 占有的资源，线程 T2 等待线程 T1 占有的资源，就是循环等待。

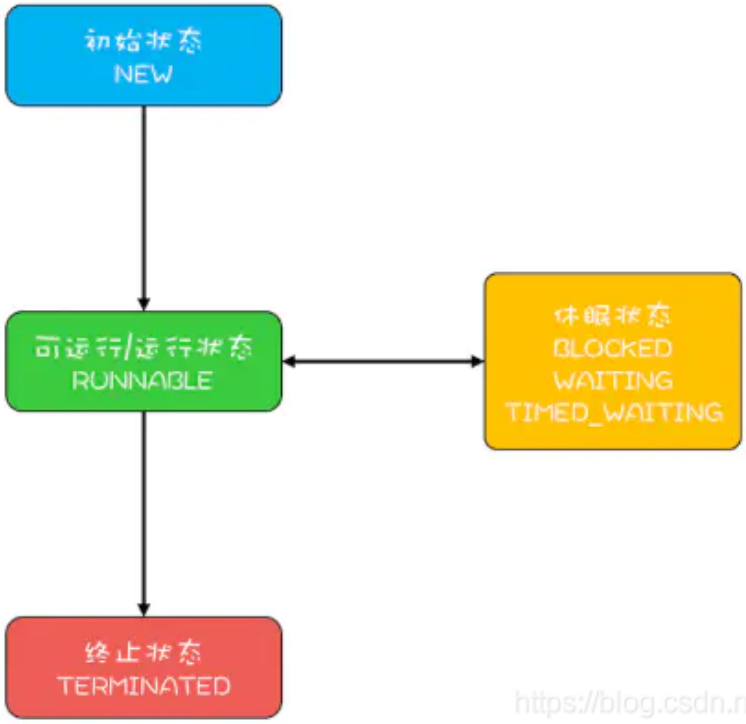
2.4 用“等待-通知”机制优化循环等待

在 Java 语言里，等待 - 通知机制可以有多种实现方式，比如 Java 语言内置的 synchronized 配合 wait()、notify()、notifyAll() 这三个方法就能轻松实现。如果 synchronized 锁定的是 this，那么对应的一定是 this.wait()、this.notify()、this.notifyAll()；如果 synchronized 锁定的是 target，那么对应的一定是 target.wait()、target.notify()、target.notifyAll() 。notify() 是会随机地通知等待队列中的一个线程，而 notifyAll() 会通知等待队列中的所有线程。有个阿姆达尔（Amdahl）定律，代表了处理器并行运算之后效率提升的能力，它正好可以解决这个问题，具体公式如下：

$$S = \frac{1}{(1-p)+\frac{p}{n}}$$

公式里的 n 可以理解为 CPU 的核数，p 可以理解为并行百分比。

Java线程



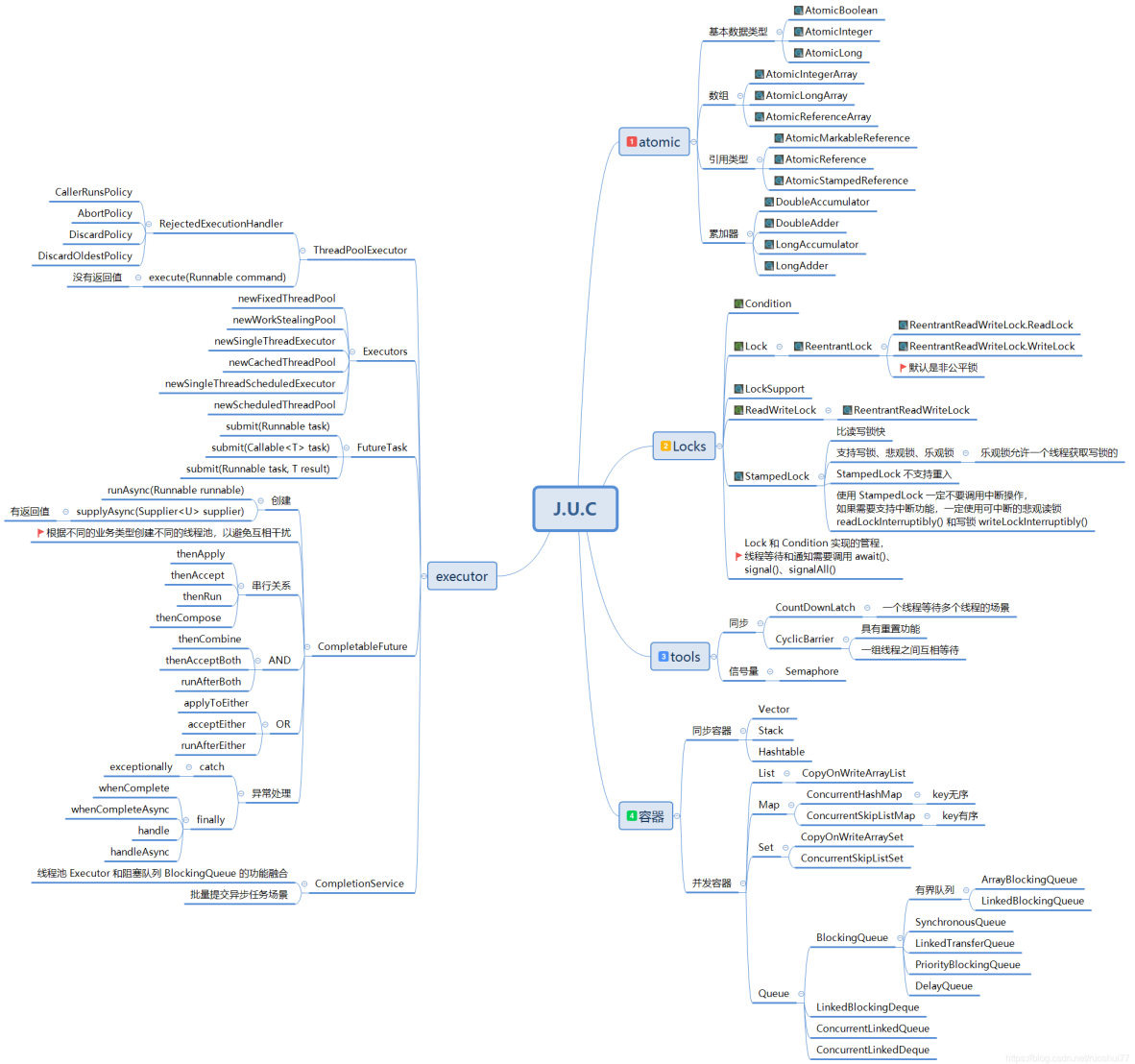
线程在sleep期间被打断了，抛出一个InterruptedException异常，try catch捕捉此异常，应该重置一下中断标示，因为抛出异常后，中断标示会自动清除掉！

```
Thread th = Thread.currentThread();
while(true) {
  if(th.isInterrupted()) {
    break;
  }

  // 省略业务代码无数
  try {
    Thread.sleep(100);
  }catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
}
```

CPU 密集型计算:理论上“线程的数量 =CPU 核数”就是最合适的。不过在工程上，线程的数量一般会设置为“CPU 核数 +1”，这样的话，当线程因为偶尔的内存页失效或其他原因导致阻塞时，这个额外的线程可以顶上，从而保证 CPU 的利用率。
I/O 密集型:最佳线程数 =CPU 核数 * [1 +（I/O 耗时 / CPU 耗时）]
Java不支持尾递归，尽量不要使用递归。

2.5 并发工具类


J.U.C
executor
ThreadPoolExecutor
RejectedExecutionHandler
CallerRunsPolicy
AbortPolicy
DiscardPolicy
DiscardOldestPolicy
没有返回值
execute(Runnable command)
Executors
newFixedThreadPool
newWorkStealingPool
newSingleThreadExecutor
newCachedThreadPool
newSingleThreadScheduledExecutor
newScheduledThreadPool
FutureTask
submit(Runnable task)
submit(Callable<T> task)
submit(Runnable task, T result)
CompletableFuture
创建
runAsync(Runnable runnable)
有返回值
supplyAsync(Supplier<U> supplier)
根据不同的业务类型创建不同的线程池，以避免互相干扰
串行关系
thenApply
thenAccept
thenRun
thenCompose
AND
thenCombine
thenAcceptBoth
runAfterBoth
OR
applyToEither
acceptEither
runAfterEither
异常处理
exceptionally
catch
finally
whenComplete
whenCompleteAsync
handle
handleAsync
CompletionService
线程池 Executor 和阻塞队列 BlockingQueue 的功能融合
批量提交异步任务场景
1 atomic
基本数据类型
AtomicBoolean
AtomicInteger
AtomicLong
数组
AtomicIntegerArray
AtomicLongArray
AtomicReferenceArray
引用类型
AtomicMarkableReference
AtomicReference
AtomicStampedReference
累加器
DoubleAccumulator
DoubleAdder
LongAccumulator
LongAdder
2 Locks
Condition
Lock
ReentrantLock
ReentrantReadWriteLock.ReadLock
ReentrantReadWriteLock.WriteLock
默认是非公平锁
LockSupport
ReadWriteLock
ReentrantReadWriteLock
StampedLock
比读写锁快
支持写锁、悲观锁、乐观锁
乐观锁允许一个线程获取写锁的
StampedLock 不支持重入
使用 StampedLock 一定不要调用中断操作，
如果需要支持中断功能，一定使用可中断的悲观读锁
readLockInterruptibly() 和写锁 writeLockInterruptibly()
Lock 和 Condition 实现的管程，
线程等待和通知需要调用 await()、
signal()、signalAll()
3 tools
同步
CountDownLatch
一个线程等待多个线程的场景
CyclicBarrier
具有重置功能
一组线程之间互相等待
信号量
Semaphore
4 容器
同步容器
Vector
Stack
Hashtable
并发容器
List
CopyOnWriteArrayList
Map
ConcurrentHashMap
key无序
ConcurrentSkipListMap
key有序
Set
CopyOnWriteArraySet
ConcurrentSkipListSet
Queue
BlockingQueue
有界队列
ArrayBlockingQueue
LinkedBlockingQueue
SynchronousQueue
LinkedTransferQueue
PriorityBlockingQueue
DelayQueue
LinkedBlockingDeque
ConcurrentLinkedQueue
ConcurrentLinkedDeque