java并发

2020年2月27日 21:27

```
Java 并发{
    锁{
        可重入锁{
            读写锁{
            }
        }
        乐观锁{
            CAS 机制
        }
        悲观锁{
        }
        JVM 对锁的优化{
            自旋锁{
                解释
                自适应自旋
            }
            锁分离{
                读写锁{
                }
            }
        }
    }
    ThreadLocal{
        使用
        内存溢出问题{
            原因
            解决
        }
    }
    线程{
        状态
        FutureTask{
```

```
        }
        fork/join{
        }
    }
    Atomic 包{

    }
    并发工具类{

    }

}
```

线程

2020年3月2日 0:26

synchronized

2020年2月29日 20:04

使用

- 修饰代码块：同步代码块

  注意锁的范围　是否完全锁住了并发的范围　　是否是临界竞争资源

```
synchronized(this) {

}
```

- 修饰普通方法

```
public synchronized void method(){
    //todo
};
```

- 修饰静态方法

```
public static synchronized void method(){
    //todo
};
```

- 　修饰类 （ 作用的对象是这个类的所有对象，只要是这个类型的class不管有几个对象都会起作用 ）

```
class ClassName {
    public void method() {
       synchronized(ClassName.class) {
          // todo
       }
    }
 }
```

内部原理

Synchronized 的同步是基于进入和退出监视器对象（Monitor）来实现的。

同步代码块	编译器会在自动在代码块前后加 monitorenter 和 monitorexit 指令;
同步方法	编译器会自动在方法标识上加上 ACC_SYNCHRONIZED 来表示这个方法为同步方法。

在JVM中，对象在内存中的布局为三块区域：

- 对齐填充
  - HotSpot虚拟机要求对象的起始地址必须是8字节的整数倍，也就是对象的大小必须是8字节的整数倍。
  - 对象头部分正好是8字节的倍数（1倍或者2倍），当对象实例数据部分没有对齐的时候，就需要通过对齐填充来补全。
- 实例数据
  - 是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。
  - 父类中继承下来的，子类中定义的，都会记录下来。
  - HotSpot虚拟机默认的分配策略为longs/doubles、ints、shorts/chars、bytes/booleans、oop，相同宽度的字段分配到一起。
- 对象头：
  - 运行时数据
    -

存储内容	标志位	状态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀（重量级锁定）
空，不需要记录信息	11	GC标记
偏向线程ID、偏向时间戳、对象分代年龄	01	可偏向

  - 类型指针
    - 对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。
    - 如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据
      - 虚拟机可以通过普通 Java 对象的元数据信息确定 Java 对象的大小，但是从数组的元数据中无法确定数组的大小

在重量级锁情况下，锁状态为重量级锁，锁标志位10，还有一个指向Monitor对象的指针。每个对象都存在着一个monitor与之关联，当一个monitor被某个线程持有后，它便处于锁定的状态。monitor是ObjectMonitor的具体实现。

```
ObjectMonitor() {
    _header       = NULL;
    _count        = 0; //记录个数
    _waiters      = 0,
    _recursions   = 0;
```

```
    _object       = NULL;
    _owner        = NULL;
    _WaitSet      = NULL; //处于wait状态的线程，会被加入到_WaitSet
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;
    FreeNext      = NULL ;
    _EntryList    = NULL ; //处于等待锁block状态的线程，会被加入到该列表
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
}
```

ObjectMonitor中有两个队列

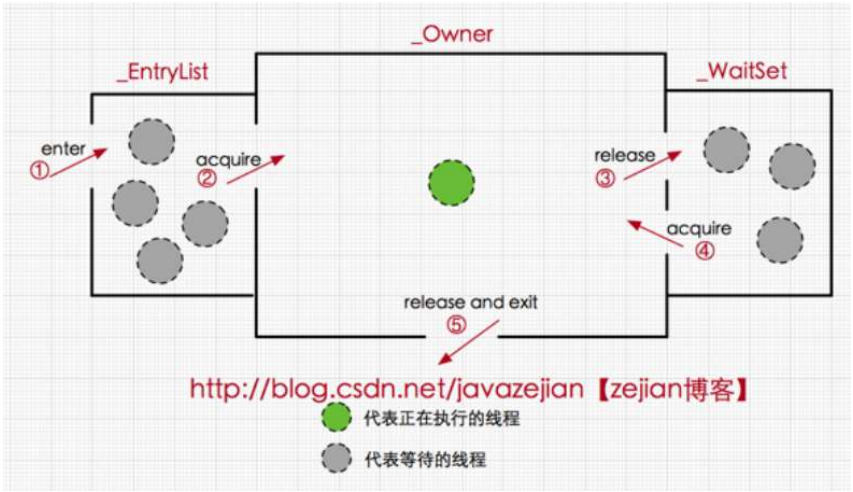
_WaitSet 和 _EntryList，用来保存ObjectWaiter对象列表(每个等待锁的线程都会被封装成ObjectWaiter对象)

_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，首先会进入 _EntryList 集合，

当线程获取到对象的monitor后进入_Owner区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，

若线程调用wait()方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSet集合中等待被唤醒。

若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)



synchronized 的锁优化

锁可以升级不可以降级，但是偏向锁状态可以被重置为无锁状态。

1. 偏向锁

偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程在访问，不存在多线程争用的情况，则线程时不需要触发同步的，这种情况下，就会给同步对象加一个偏向锁。

它通过消除资源无竞争情况下的同步原语，进一步提供了程序的运行性能。

理论基础： 研究发现，大多数情况下不仅不存在锁竞争，而且总是由同一个线程多次重入，为了让同一个线程多次重入获取锁的代价更低，就引入了偏向锁的概念。（更好的支持可重入的设计）

偏向锁获取过程：

1. 访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01，确认是否为可偏向状态。
2. 如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤5，否则进入步骤3。
3. 如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行5；如果竞争失败，执行4。
4. 如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。（撤销偏向锁的时候会导致当前获得偏向锁的线程被暂停）
5. 执行同步代码。

偏向锁的释放：

偏向锁只有当遇到其他线程也尝试获取偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。（也就是对于获取偏向锁的线程 只有lock的动

作，没有unlock的动作，这是因偏向的需要，即使可能这个线程已经死亡。）偏向锁的撤销步骤如下：

等到全局安全点（在这个时间点上没有正在执行的字节码）。
暂停持有偏向锁的线程，然后检查持有偏向锁的线程是否还活着，如果线程不处于活动状态，则将对象头设置为无锁状态。争抢锁的线程会再走生成偏向锁的过程，然后成为偏向锁的拥有者。
如果持有偏向锁的线程还处于活动状态，则将锁升级为轻量级锁。

2．轻量级锁

加锁过程：

线程在执行同步块之前，如果此同步对象没有被锁定（锁状态标志位为“01”状态），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象的目前的Mark Word的拷贝（官方称为：Displaced Mark Word）。

然后虚拟机将使用CAS操作尝试将对象头中的Mark Word更新为指向Lock Record的指针。如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位（Mark Word的最后两个Bits）将转变为“00”，即表示此对象处于轻量级锁定的状态。

如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，可以直接进入同步块继续执行，否则说明这个锁对象已经被其他线程抢占了。如果有两条以上的线程争抢同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。

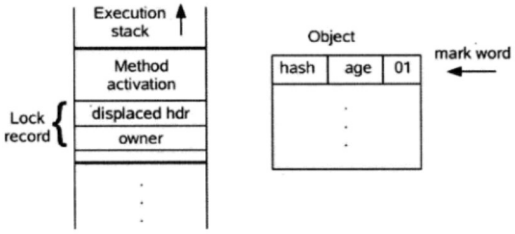


图 13-3　轻量级锁 CAS 操作之前堆栈与对象的状态[1]

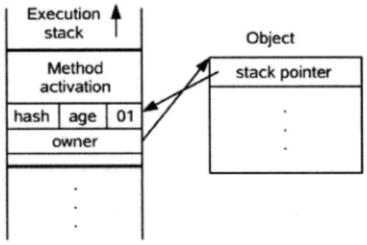


图 13-4　轻量级锁 CAS 操作之后堆栈与对象的状态

释放过程：

轻量级解锁也是通过CAS操作来进行的，如果对象的Mark Word仍然指向这线程的锁记录，那就用CAS操作把对象当前的Mark Word和线程中复制的 Displaced Mark Word 替换回来，如果替换成功，整个同步过程就完成了。如果替换失败，说明有其他线程尝试过获取该锁，那就要在释放锁的同时，唤醒被挂起的线程。

3．自旋锁

线程挂起和恢复都需要从用户态转为内核态，这些操作给系统的并发性能带来了很大的压力。为了应对持有锁的线程在很短时间内释放资源的场景，避免用户线程和内核的切换消耗。

4．自适应自旋锁

自适应自旋锁的自旋时间会根据前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如100个循环。另一个方面，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。

智能化的自旋

5．锁消除

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。

锁消除主要判定依据来源于逃逸分析的数据支持，如果判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁自然就无须进行。

6．锁粗化

锁粗化，如果虚拟机探测到同一个线程有一连串的对同一个锁进行加锁和解锁操作，将会把加锁同步的范围扩展到整个操作序列的外部，这样就只需要加锁一次。

锁粗化最常见的场景：如果有一系列的连续操作（在同一个线程中）都对同一个对象反复进行加锁和解锁，甚至加锁操作是出现在循环体中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。

由一个问题出发 ： 为什么synchronized无法禁止指令重排，却能保证有序性

- 最好的解决有序性问题的办法，就是禁止处理器优化和指令重排，就像 volatile 中使用内存屏障一样。
- 在Java中，不管怎么排序，都不能影响单线程程序的执行结果。这就是as-if-serial语义，所有硬件优化的前提都是必须遵守 as-if-serial 语义。
- synchronized 是 Java 提供的锁，可以通过他对Java中的对象加锁，并且他是一种排他的、可重入的锁。
- 当某个线程执行到一段被 synchronized 修饰的代码之前，会先进行加锁，执行完之后再进行解锁。在加锁之后，解锁之前，其他线程是无法再次获得锁的，只有这条加锁线程可以重复获得该锁。
- synchronized 通过排他锁的方式就保证了同一时间内，被 synchronized 修饰的代码是单线程执行的。
- 这就满足了 as-if-serial 语义的一个关键前提，那就是单线程，因为有 as-if-serial 语义保证，单线程的有序性就天然存在了。

synchronized与Lock锁的比较

锁的获取与释放

synchronized 是 Java 语言内置的实现，使用简单，无须关注锁的释放与获取，都是JVM自动管理。

Lock相关的锁，则需要手动获取与释放，稍有不慎，忘记释放锁或者程序处理异常导致锁没有释放，则会造成死锁。所以，通常unLock()操作都要在finally语句块中进行释放锁操作。但是，由于锁的释放与获取都由程序把控，能够实现更灵活的锁获取与释放。

Lock锁可以非阻塞、响应中断、响应超时

1. 使用Lock接口可以非阻塞地获取锁，具体API是tryLock()和tryLock(long time，TimeUnit unit)。
2. 使用Lock接口可以响应中断和超时：synchronized一旦尝试获取锁，就会一直等待下去，直到获取锁；但是Lock接口可以响应中断或者超时。具体API是lockInterruptibly()和tryLock(long time，TimeUnit unit)。

Lock锁提供了更丰富的锁

Lock接口提供了更丰富的锁语义：Lock接口及其实现类，实现了读写锁、公平锁与非公平锁等，读写锁：ReadWriteLock；公平锁与非公平锁：以ReentrantLock为例，只要构造函数传入true，就可以使用公平锁，默认是非公平锁。synchronized是支持重入的非公平锁。

实现原理

Lock接口都是基于 AQS 实现的，锁的标志是 AQS 中的 state 标志位，当获取锁失败后，Lock 会进入自旋 + CAS 的形式实现的，以等待锁的获取。

synchronized 则是通过争抢对象关联的Monitor实现的，当线程争抢Monitor失败后，则会进入阻塞状态；由于Java的线程时映射到操作系统原生的线程之上的，如果阻塞或唤醒一个线程就需要操作系统帮忙，这时就要从用户态转到核心态，因此线程状态转换需要花费很多的处理器时间，对于简单的同步块（比如被synchronized修饰的get、set方法），往往状态转换消耗的时间比用户代码执行的时间还要长，不过随着JDK1.6后，偏向锁、轻量级锁、锁消除、自旋锁、自适应自旋锁、锁粗化的出现，synchronized 的性能得到了很大的改善。

性能比较

ReadWriteLock 接口的实现类，可以实现读锁和写锁分离，极大提高了读多写少情况下系统性能。

从常规性能上来说，如果资源竞争不激烈，两者的性能是差不多的，但当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。(据说)

synchronized的横切面详解

1. synchronized原理
2. 升级过程
3. 汇编实现
4. vs reentrantLock的区别

java源码层级

synchronized(o)

字节码层级

monitorenter moniterexit

JVM层级（Hotspot）

Hotspot的实现

锁状态	25位	31位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
无锁态（new）	unused	hashCode（如果有调用）	unused	分代年龄	0	0	1

锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
偏向锁	当前线程指针 JavaThread*	Epoch	unused	分代年龄	1	0	1

锁状态	62位	2bit 锁标志位	
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针	0	0
重量级锁	指向互斥量（重量级锁）的指针	1	0
GC标记信息	CMS过程用到的标记信息	1	1

字节头的 markword

工具：JOL = Java Object Layout

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.openjdk.jol/jol-core -->
    <dependency>
        <groupId>org.openjdk.jol</groupId>
        <artifactId>jol-core</artifactId>
        <version>0.9</version>
    </dependency>
</dependencies>
```

jdk8u：mark0op.hpp

```
// Bit-format of an object header (most significant first, big endian layout below):
//
//  32 bits:
//  --------
//             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
//             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
//             size:32 ------------------------------------------>| (CMS free block)
//             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
//
//  64 bits:
//  --------
//  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
//  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
//  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
//  size:64 ----------------------------------------------------->| (CMS free block)
//
//  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
//  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
//  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted object)
//  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)
```

JVM层级（Hotspot）

原始类

```
import org.openjdk.jol.info.ClassLayout;
public class T01_Sync1 {

    public static void main(String[] args) {
        Object o = new Object();
        System.out.println(ClassLayout.parseInstance(o).toPrintable());
    }
}
```

查看字节头

```
com.insidesync.T01_Sync1$Lock object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)  05 00 00 00 (00000101 00000000 00000000 00000000) (5)
      4     4        (object header)  00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4        (object header)  49 ce 00 20 (01001001 11001110 00000000 00100000) (536923721)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
```

```
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

```
com.insidesync.T02_Sync2$Lock object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4   (object header)  05 90 2e 1e (00000101 10010000 00101110 00011110) (506368005)
      4     4   (object header)  1b 02 00 00 (00011011 00000010 00000000 00000000) (539)
      8     4   (object header)  49 ce 00 20 (01001001 11001110 00000000 00100000) (536923721)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes tota
```

InterpreterRuntime:: monitorenter方法

```
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thread, BasicObjectLock* elem))
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
  if (PrintBiasedLockingStatistics) {
    Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
  }
  Handle h_obj(thread, elem->obj());
  assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
         "must be NULL or an object");
  if (UseBiasedLocking) {
    // Retry fast entry if bias is revoked to avoid unnecessary inflation
    ObjectSynchronizer::fast_enter(h_obj, elem->lock(), true, CHECK);
  } else {
    ObjectSynchronizer::slow_enter(h_obj, elem->lock(), CHECK);
  }
  assert(Universe::heap()->is_in_reserved_or_null(elem->obj()),
         "must be NULL or an object");
#ifdef ASSERT
  thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END
```

synchronizer.cpp
revoke_and_rebias

```
void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock, bool attempt_rebias, TRAPS) {
 if (UseBiasedLocking) {
    if (!SafepointSynchronize::is_at_safepoint()) {
      BiasedLocking::Condition cond = BiasedLocking::revoke_and_rebias(obj, attempt_rebias, THREAD);
      if (cond == BiasedLocking::BIAS_REVOKED_AND_REBIASED) {
        return;
      }
    } else {
      assert(!attempt_rebias, "can not rebias toward VM thread");
      BiasedLocking::revoke_at_safepoint(obj);
    }
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
 }
 slow_enter (obj, lock, THREAD) ;
}
```

```
void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
  markOop mark = obj->mark();
  assert(!mark->has_bias_pattern(), "should not see bias pattern here");
  if (mark->is_neutral()) {
    // Anticipate successful CAS -- the ST of the displaced mark must
    // be visible <= the ST performed by the CAS.
    lock->set_displaced_header(mark);
    if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark)) {
      TEVENT (slow_enter: release stacklock) ;
      return ;
    }
    // Fall through to inflate() ...
  } else
  if (mark->has_locker() && THREAD->is_lock_owned((address)mark->locker())) {
    assert(lock != mark->locker(), "must not re-lock the same lock");
    assert(lock != (BasicLock*)obj->mark(), "don't relock with same BasicLock");
    lock->set_displaced_header(NULL);
    return;
  }
```

```
#if 0
  // The following optimization isn't particularly useful.
  if (mark->has_monitor() && mark->monitor()->is_entered(THREAD)) {
    lock->set_displaced_header (NULL) ;
    return ;
  }
#endif
  // The object header will never be displaced to this lock,
  // so it does not matter what the value is, except that it
  // must be non-zero to avoid looking like a re-entrant lock,
  // and must not look locked either.
  lock->set_displaced_header(markOopDesc::unused_mark());
  ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
}
```

inflate方法：膨胀为重量级锁

锁升级过程

JDK8 markword实现表：

无锁 - 偏向锁 - 轻量级锁 （自旋锁，自适应自旋） - 重量级锁

synchronized优化的过程和markword息息相关

用markword中最低的三位代表锁状态 其中1位是偏向锁位 两位是普通锁位

1. Object o = new Object()
   锁 = 0 01 无锁态
2. o.hashCode()
   001 + hashcode
   ```java
   00000001 10101101 00110100 00110110
   01011001 00000000 00000000 00000000
   ```
   little endian big endian
   00000000 00000000 00000000 01011001 00110110 00110100 10101101 00000000
3. 默认synchronized(o)
   00 -> 轻量级锁
   默认情况 偏向锁有个时延，默认是4秒
   why? 因为JVM虚拟机自己有一些默认启动的线程，里面有好多sync代码，这些sync代码启动时就知道肯定会有竞争，如果使用偏向锁，就会造成偏向锁不断的进行锁撤销和锁升级的操作，效率较低。

```
-XX:BiasedLockingStartupDelay=0
```

4. 如果设定上述参数
   new Object () - > 101 偏向锁 ->线程ID为0 -> Anonymous BiasedLock
   打开偏向锁，new出来的对象，默认就是一个可偏向匿名对象101
5. 如果有线程上锁
   上偏向锁，指的就是，把markword的线程ID改为自己线程ID的过程
   偏向锁不可重偏向 批量偏向 批量撤销
6. 如果有线程竞争

撤销偏向锁，升级轻量级锁

线程在自己的线程栈生成LockRecord ，用CAS操作将markword设置为指向自己这个线程的LR的指针，设置成功者得到锁

7. 如果竞争加剧

竞争加剧：有线程超过10次自旋， -XX:PreBlockSpin， 或者自旋线程数超过CPU核数的一半， 1.6之后，加入自适应自旋 Adapative Self Spinning ， JVM自己控制

升级重量级锁：-> 向操作系统申请资源，linux mutex ， CPU从3级-0级系统调用，线程挂起，进入等待队列，等待操作系统的调度，然后再映射回用户空间

(以上实验环境是JDK11，打开就是偏向锁，而JDK8默认对象头是无锁)

偏向锁默认是打开的，但是有一个时延，如果要观察到偏向锁，应该设定参数

加锁，指的是锁定对象

锁升级的过程

JDK较早的版本 OS的资源 互斥量 用户态 -> 内核态的转换 重量级 效率比较低

现代版本进行了优化

无锁 - 偏向锁 -轻量级锁（自旋锁）-重量级锁

偏向锁 - markword 上记录当前线程指针，下次同一个线程加锁的时候，不需要争用，只需要判断线程指针是否同一个，所以，偏向锁，偏向加锁的第一个线程 。hashCode备份在线程栈上 线程销毁，锁降级为无锁

有争用 - 锁升级为轻量级锁 - 每个线程有自己的LockRecord在自己的线程栈上，用CAS去争用markword的LR的指针，指针指向哪个线程的LR，哪个线程就拥有锁

自旋超过10次，升级为重量级锁 - 如果太多线程自旋 CPU消耗过大，不如升级为重量级锁，进入等待队列（不消耗CPU）-XX:PreBlockSpin

自旋锁在 JDK1.4.2 中引入，使用 -XX:+UseSpinning 来开启。JDK 6 中变为默认开启，并且引入了自适应的自旋锁（适应性自旋锁）。

自适应自旋锁意味着自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。

偏向锁由于有锁撤销的过程revoke，会消耗系统资源，所以，在锁争用特别激烈的时候，用偏向锁未必效率高。还不如直接使用轻量级锁。

synchronized最底层实现

```
public class T {
    static volatile int i = 0;

    public static void n() { i++; }

    public static synchronized void m() {}

    publics static void main(String[] args) {
        for(int j=0; j<1000_000; j++) {
            m();
            n();
        }
    }
}
```

java -XX:+UnlockDiagonositicVMOptions -XX:+PrintAssembly T

C1 Compile Level 1（一级优化）

C2 Compile Level 2（二级优化）

找到m() n()方法的汇编码，会看到 lock comxchg .....指令

synchronized vs Lock (CAS)

在高争用 高耗时的环境下synchronized效率更高

在低争用 低耗时的环境下CAS效率更高

synchronized到重量级之后是等待队列（不消耗CPU） CAS（等待期间消耗CPU）

一切以实测为准

锁消除 lock eliminate

```
public void add(String str1,String str2){
         StringBuffer sb = new StringBuffer();
         sb.append(str1).append(str2);
}
```

我们都知道 StringBuffer 是线程安全的，因为它的关键方法都是被 synchronized 修饰过的，但我们看上面这段代码，我们会发现，sb 这个引用只会在 add 方

法中使用，不可能被其它线程引用（因为是局部变量，栈私有），因此 sb 是不可能共享的资源，JVM 会自动消除 StringBuffer 对象内部的锁。

锁粗化 lock coarsening

```
public String test(String str){

        int i = 0;
        StringBuffer sb = new StringBuffer();
        while(i < 100){
            sb.append(str);
            i++;
        }
        return sb.toString();
}
```

JVM 会检测到这样一连串的操作都对同一个对象加锁（while 循环内 100 次执行 append，没有锁粗化的就要进行 100 次加锁/解锁），此时 JVM 就会将加锁的范围粗化到这一连串的操作的外部（比如 while 虚幻体外），使得这一连串操作只需要加一次锁即可。

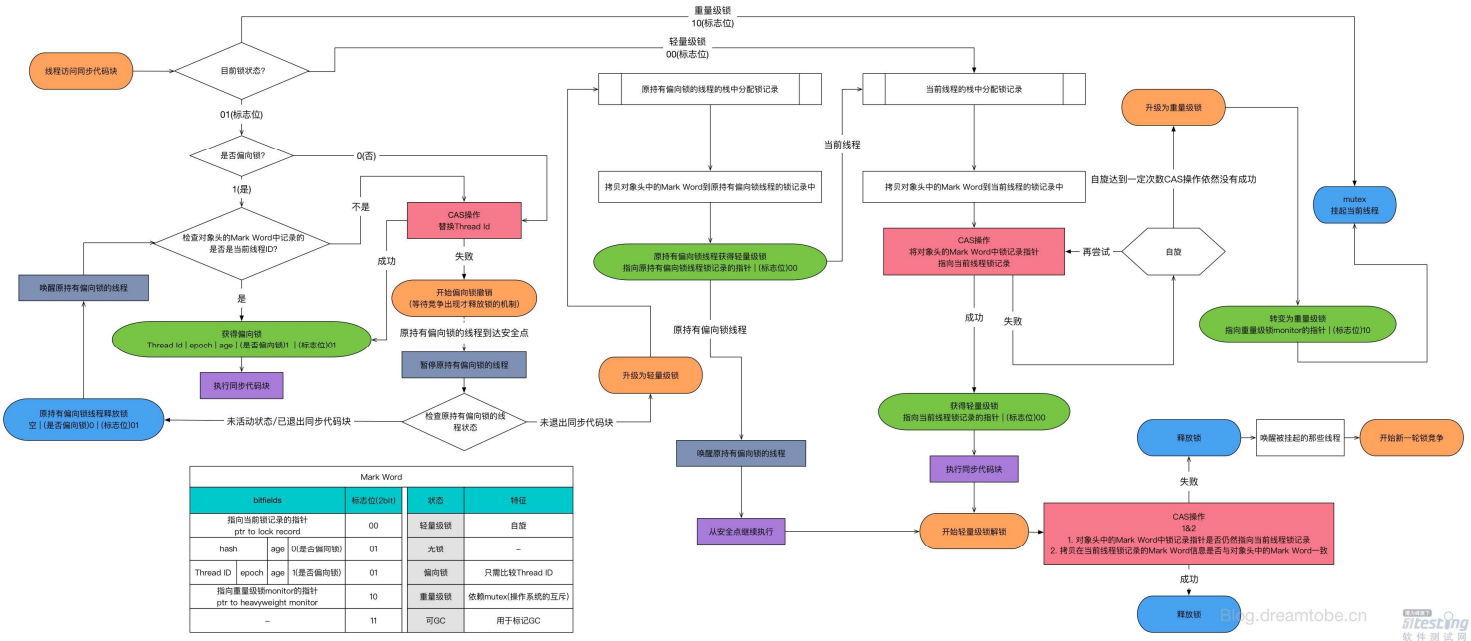
锁降级（不重要）
https://www.zhihu.com/question/63859501
只被VMThread访问，降级也就没啥意义了。所以可以简单认为锁降级不存在!

超线程
一个ALU + 两组Registers + PC

参考资料
http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html

<table>
<tr><th colspan="5">Mark Word</th></tr>
<tr><th colspan="4">bitfields</th><th>标志位(2bit)</th><th>状态</th><th>特征</th></tr>
<tr><td colspan="4">指向当前锁记录的指针
ptr to lock record</td><td>00</td><td>轻量级锁</td><td>自旋</td></tr>
<tr><td colspan="2">hash</td><td>age</td><td>0(是否偏向锁)</td><td>01</td><td>无锁</td><td>-</td></tr>
<tr><td>Thread ID</td><td>epoch</td><td>age</td><td>1(是否偏向锁)</td><td>01</td><td>偏向锁</td><td>只需比较Thread ID</td></tr>
<tr><td colspan="4">指向重量级锁monitor的指针
ptr to heavyweight monitor</td><td>10</td><td>重量级锁</td><td>依赖mutex(操作系统的互斥)</td></tr>
<tr><td colspan="4">-</td><td>11</td><td>可GC</td><td>用于标记GC</td></tr>
</table>

Java Synchronized原理



Mark Word

bitfields			标志位(2bit)
指向当前锁记录的指针 ptr to lock record			00
hash		age	0(是否偏向锁) 01
Thread ID	epoch	age	1(是否偏向锁) 01
指向重量级锁monitor的指针 ptr to heavyweight monitor			10
–			11

状态	特征
轻量级锁	自旋
无锁	–
偏向锁	只需比较Thread ID
重量级锁	依赖mutex(操作系统的互斥)
可GC	用于标记GC

volatile

2020年3月1日 1:01

被volatile修饰的共享变量，就具有了以下两点特性：

保证了不同线程对该变量操作的内存可见性；

禁止指令重排序；

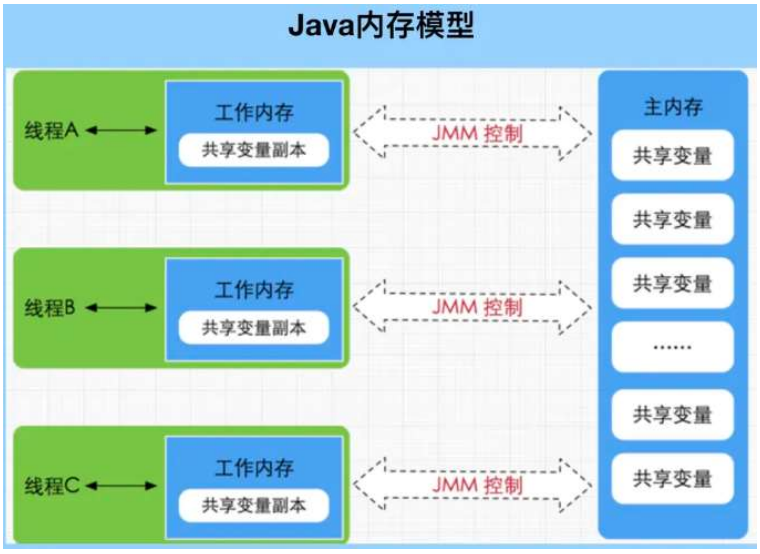
可见性

Java内存模型（JMM），来屏蔽掉各种硬件和操作系统的内存访问差异，让Java程序在各种平台上都能达到一致的内存访问效果

Java内存模型是通过变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值，将主内存作为传递媒介

可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。

volatile变量保证新值能够立马同步到主内存，使用时也立即从主内存刷新，保证了多线程操作时变量的可见性



volatile不能保证原子性，它只是对单个volatile变量的读/写具有原子性

原子性即一个操作或一系列是不可中断的。即使是在多个线程的情况下，操作一旦开始，就不会被其他线程干扰。

Java内存模型中有序性可归纳为这样一句话：如果在本线程内观察，所有操作都是有序的，如果在一个线程中观察另一个线程，所有操作都是无序的。

有序性是指对于单线程的执行代码，执行是按顺序依次进行的。

但在多线程环境中，则可能出现乱序现象，因为在编译过程会出现“指令重排”，重排后的指令与原指令的顺序未必一致。

happens-before 原则

- 程序次序规则：
  - 一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
- 锁定规则：
  - 一个unLock操作先行发生于后面对同一个锁额lock操作
- volatile变量规则：
  - 对一个变量的写操作先行发生于后面对这个变量的读操作
- 传递规则：
  - 如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C
- 线程启动规则：
  - Thread对象的start()方法先行发生于此线程的每个一个动作
- 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
- 线程终结规则：
  - 线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行
- 对象终结规则：
  - 一个对象的初始化完成先行发生于他的finalize()方法的开始

伪共享 False sharing

多CPU同时访问同一块内存区域就是“共享”，就会产生冲突，需要控制协议来协调访问。

会引起“共享”的最小内存区域大小就是一个cache line。当两个以上CPU都要访问同一个cache line大小的内存区域时，就会引起冲突，这种情况就叫“共享”。

但是，这种情况里面又包含了“其实不是共享”的“伪共享”情况。

比如，两个处理器各要访问一个word，这两个word却存在于同一个cache line大小的区域里，

应用逻辑层面说，这两个处理器并没有共享内存，因为他们访问的是不同的内容（不同的word）

但是因为cache line的存在和限制，这两个CPU要访问这两个不同的word时，却一定要访问同一个cache line块，产生了事实上的“共享”

美团的 Disruptor 队列

内存屏障

内存屏障分为两种：Load Barrier(读屏障) 和 Store Barrier(写屏障)

内存屏障有两个作用：

1.阻止屏障两侧的指令重排序；
2.强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。

对于Load Barrier来说，在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据；
对于Store Barrier来说，在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。

java的内存屏障通常所谓的四种

- LoadLoad
- StoreStore
- LoadStore
- StoreLoad

实际上也是上述两种的组合，完成一系列的屏障和数据同步功能。

LoadLoad屏障：对于这样的语句Load1；LoadLoad；Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
StoreStore屏障：对于这样的语句Store1；StoreStore；Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
LoadStore屏障：对于这样的语句Load1；LoadStore；Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
StoreLoad屏障：对于这样的语句Store1；StoreLoad；Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能

volatile的内存屏障策略非常严格保守，非常悲观且毫无安全感的心态：

在每个volatile写操作前插入StoreStore屏障，在写操作后插入StoreLoad屏障；
在每个volatile读操作前插入LoadLoad屏障，在读操作后插入LoadStore屏障；

由于内存屏障的作用，避免了volatile变量和其它指令重排序、线程之间实现了通信，使得volatile表现出了锁的特性。

volatile的用途
1.线程可见性

```
public class T01_ThreadVisibility {
    private static volatile boolean flag = true;
    public static void main(String[] args) throws InterruptedException {
        new Thread(()-> {
            while (flag) {
                //do sth
            }
            System.out.println("end");
        }, "server").start();

        Thread.sleep(1000);
        flag = false;
    }
}
```

2.防止指令重排序

问题：DCL单例需不需要加volatile?

CPU的基础知识

* 缓存行对齐

  缓存行64个字节是CPU同步的基本单位，缓存行隔离会比伪共享效率要高

  Disruptor

```
public class T02_CacheLinePadding {
      private static class Padding {
          public volatile long p1, p2, p3, p4, p5, p6, p7; //
      }

      private static class T extends Padding {
          public volatile long x = 0L;
      }

      public static T[] arr = new T[2];

      static {
          arr[0] = new T();
          arr[1] = new T();
      }

      public static void main(String[] args) throws Exception {
          Thread t1 = new Thread(()->{
              for (long i = 0; i < 1000_0000L; i++) {
                  arr[0].x = i;
              }
          });

          Thread t2 = new Thread(()->{
              for (long i = 0; i < 1000_0000L; i++) {
                  arr[1].x = i;
              }
          });

          final long start = System.nanoTime();
          t1.start();
          t2.start();
          t1.join();
          t2.join();
          System.out.println((System.nanoTime() - start)/100_0000);
      }
  }
```

  MESI

* 伪共享

* 合并写

  CPU内部的4个字节的Buffer

```
public final class WriteCombining {

private static final int ITERATIONS = Integer.MAX_VALUE;
private static final int ITEMS = 1 << 24;
private static final int MASK = ITEMS - 1;
private static final byte[] arrayA = new byte[ITEMS];
private static final byte[] arrayB = new byte[ITEMS];
private static final byte[] arrayC = new byte[ITEMS];
private static final byte[] arrayD = new byte[ITEMS];
private static final byte[] arrayE = new byte[ITEMS];
private static final byte[] arrayF = new byte[ITEMS];

public static void main(final String[] args) {
    for (int i = 1; i <= 3; i++) {
        System.out.println(i + " SingleLoop duration (ns) = " + runCaseOne());
        System.out.println(i + " SplitLoop  duration (ns) = " + runCaseTwo());
    }
}
```

```
public static long runCaseOne() {
    long start = System.nanoTime();
    int i = ITERATIONS;
    while (--i != 0) {
        int slot = i & MASK;
        byte b = (byte) i;
        arrayA[slot] = b;
        arrayB[slot] = b;
        arrayC[slot] = b;
        arrayD[slot] = b;
        arrayE[slot] = b;
        arrayF[slot] = b;
    }
    return System.nanoTime() - start;
}
public static long runCaseTwo() {
    long start = System.nanoTime();
    int i = ITERATIONS;
    while (--i != 0) {
        int slot = i & MASK;
        byte b = (byte) i;
        arrayA[slot] = b;
        arrayB[slot] = b;
        arrayC[slot] = b;
    }
    i = ITERATIONS;
    while (--i != 0) {
        int slot = i & MASK;
        byte b = (byte) i;
        arrayD[slot] = b;
        arrayE[slot] = b;
        arrayF[slot] = b;
    }
    return System.nanoTime() - start;
}
```

* 指令重排序

```
public class T04_Disorder {
    private static int x = 0, y = 0;
    private static int a = 0, b =0;

    public static void main(String[] args) throws InterruptedException {
        int i = 0;
        for(;;) {
            i++;
            x = 0; y = 0;
            a = 0; b = 0;
            Thread one = new Thread(new Runnable() {
                public void run() {
                    //由于线程one先启动，下面这句话让它等一等线程two. 读着可根据自己电脑的实际性能适当调整等待时间.
                    //shortWait(100000);
                    a = 1;
                    x = b;
                }
            });

            Thread other = new Thread(new Runnable() {
                public void run() {
                    b = 1;
                    y = a;
                }
            });
            one.start();other.start();
            one.join();other.join();
            String result = "第" + i + "次 (" + x + "," + y + ") ";
            if(x == 0 && y == 0) {
                System.err.println(result);
                break;
            } else {
                //System.out.println(result);
            }
        }
    }
```

```
    public static void shortWait(long interval){
        long start = System.nanoTime();
        long end;
        do{
            end = System.nanoTime();
        }while(start + interval >= end);
    }
}
```

volatile如何解决指令重排序

1: volatile i
2: ACC_VOLATILE
3: JVM的内存屏障
4: hotspot实现

bytecodeinterpreter.cpp

```
int field_offset = cache->f2_as_index();
    if (cache->is_volatile()) {
        if (support_IRIW_for_not_multiple_copy_atomic_cpu) {
            OrderAccess::fence();
        }
    }
```

orderaccess_linux_x86.inline.hpp

```
inline void OrderAccess::fence() {
  if (os::is_MP()) {
    // always use locked addl since mfence is sometimes expensive
#ifdef AMD64
    __asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
#else
    __asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
#endif
  }
}
```

CAS

2020年3月7日 13:08

Compare And Swap (Compare And Exchange) / 自旋 / 自旋锁 / 无锁
因为经常配合循环操作，直到完成为止，所以泛指一类操作
cas(v, a, b) ，变量v，期待值a，修改值b
ABA问题，你的女朋友在离开你的这段儿时间经历了别的人，自旋就是你空转等待，一直等到她接纳你为止

Unsafe
AtomicInteger:

```
public final int incrementAndGet() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }
public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
```

Unsafe:

```
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
```

运用:

```
import sun.misc.Unsafe;
import java.lang.reflect.Field;
public class T02_TestUnsafe {
    int i = 0;
    private static T02_TestUnsafe t = new T02_TestUnsafe();
    public static void main(String[] args) throws Exception {
        //Unsafe unsafe = Unsafe.getUnsafe();
        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe) unsafeField.get(null);
        Field f = T02_TestUnsafe.class.getDeclaredField("i");
        long offset = unsafe.objectFieldOffset(f);
        System.out.println(offset);
        boolean success = unsafe.compareAndSwapInt(t, offset, 0, 1);
        System.out.println(success);
        System.out.println(t.i);
        //unsafe.compareAndSwapInt()
    }
}
```

jdk8u: unsafe.cpp:
cmpxchg = compare and exchange

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

jdk8u: atomic_linux_x86.inline.hpp

is_MP = Multi Processor

```
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint     compare_value) {
  int mp = os::is_MP();
  __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
                    : "cc", "memory");
  return exchange_value;
}
```

jdk8u: os.hpp is_MP()

```
  static inline bool is_MP() {
    // During bootstrap if _processor_count is not yet initialized
    // we claim to be MP as that is safest. If any platform has a
    // stub generator that might be triggered in this phase and for
    // which being declared MP when in fact not, is a problem - then
    // the bootstrap routine for the stub generator needs to check
    // the processor count directly and leave the bootstrap routine
    // in place until called after initialization has ocurred.
    return (_processor_count != 1) || AssumeMP;
  }
```

jdk8u: atomic_linux_x86.inline.hpp

```
#define LOCK_IF_MP(mp) "cmp $0, " #mp "; je 1f; lock; 1: "
```

最终实现:

cmpxchg = cas修改变量值
lock cmpxchg 指令

硬件:

lock 指令在执行后面指令的时候锁定一个北桥信号
(不采用锁总线的方式)

并发容器与同步容器

2020年3月1日 16:20

同步容器（在并发下进行迭代的读和写时并不是线程安全的）

同步容器类在单个方法被使用时可以保证线程安全。复合操作则需要额外的客户端加锁来保护

Vector、Stack、HashTable

Collections 类的静态工厂方法创建的类（ 如 Collections.synchronizedList ）

以简单地理解为通过 synchronized 来实现同步的容器（并发环境下性能很差）

可以通过查看 Vector，Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 synchronized。

这样做的代价是削弱了并发性，当多个线程共同竞争容器级的锁时，吞吐量就会降低。

Java ConcurrentModificationException 异常(fail-fast 快速失败机制)

在遍历一个集合时，当集合结构被修改，会抛出Concurrent Modification Exception

fail-fast会在以下两种情况下抛出ConcurrentModificationException

（1）单线程环境

集合被创建后，在遍历它的过程中修改了结构。

注意 迭代器的 remove()方法会让 expectModcount 和 modcount 相等，所以是不会抛出这个异常。

（2）多线程环境

当一个线程在遍历这个集合，而另一个线程对这个集合的结构进行了修改。

JDK 1.5 新增的并发容器（线程安全）

采用了CAS算法和部分代码使用 synchronized 锁保证线程安全。

1. ConcurrentHashMap

对应的非并发容器：HashMap

目标：代替Hashtable、synchronizedMap，支持复合操作

原理：JDK6中采用一种更加细粒度的加锁机制Segment“分段锁”，JDK8中采用CAS无锁算法。

2. CopyOnWriteArrayList

对应的非并发容器：ArrayList

目标：代替Vector、synchronizedList

原理：利用高并发往往是读多写少的特性，对读操作不加锁，对写操作，先复制一份新的集合，在新的集合上面修改，然后将新集合赋值给旧的引用，并通过 volatile 保证其可见性，当然写操作的锁是必不可少的了。

3. CopyOnWriteArraySet

对应的非并发容器：HashSet

目标：代替synchronizedSet

原理：基于CopyOnWriteArrayList实现，其唯一的不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法，其遍历当前Object数组，如Object数组中已有了当前元素，则直接返回，如果没有则放入Object数组的尾部，并返回。

4. ConcurrentSkipListMap

对应的非并发容器：TreeMap

目标：代替synchronizedSortedMap(TreeMap)

原理：Skip list（跳表）是一种可以代替平衡树的数据结构，默认是按照Key值升序的。

5. ConcurrentSkipListSet

对应的非并发容器：TreeSet

目标：代替synchronizedSortedSet

原理：内部基于ConcurrentSkipListMap实现

6. ConcurrentLinkedQueue

不会阻塞的队列

对应的非并发容器：Queue

原理：基于链表实现的FIFO队列（LinkedList的并发版本）

7. LinkedBlockingQueue、ArrayBlockingQueue、PriorityBlockingQueue

对应的非并发容器：BlockingQueue

特点：拓展了Queue，增加了可阻塞的插入和获取等操作

原理：通过ReentrantLock实现线程安全，通过Condition实现阻塞和唤醒

实现类：

LinkedBlockingQueue：基于链表实现的可阻塞的FIFO队列
ArrayBlockingQueue：基于数组实现的可阻塞的FIFO队列
PriorityBlockingQueue：按优先级排序的队列

fail-safe 安全失败机制

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

- 原理：
  - 由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发Concurrent Modification Exception。
- 缺点：
  - 基于拷贝内容的优点是避免了 Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的，不能保证遍历的是最新内容。

HashMap & HashTable & Concurrent HashMap

2020年2月5日 14:50

HashMap 的底层结构

- JDK 7
  - 数组 + 链表
  - 插入新元素 —— 头插法
    - 多线程环境下会形成环路
- JDK 8
  - 数组 + 链表（红黑树）
    - 泊松分布
  - 插入新元素 —— 尾插法
- 扩容机制
  - Capacity + LoadFactor 决定什么时候扩容
  - 两步扩容
    - 创建新数组
    - ReHash 原数组中的元素 重新计算 Hash 值之后插入到新数组中
- Hash 算法
  - index = HashCode (Key) & (Length - 1)
- 多线程下的问题
  - JDK 8虽然解决了头插法带来的死循环问题
  - 但是在多线程环境下 Put/Get 方法都没有加锁处理。线程安全仍然无法保证
- 默认大小是多少 为什么要设定为这个值
  - 默认大小 16
  - 最好是 2 的幂，为了位运算的方便
  - 选择 16 是为了实现均匀分布
- 常见面试题
  - HashMap的底层数据结构?
  - HashMap的存取原理?
  - Java7和Java8的区别?
  - 为啥会线程不安全?
  - 有什么线程安全的类代替么?
  - 默认初始化大小是多少？为啥是这么多？为啥大小都是2的幂?
  - HashMap的扩容方式？负载因子是多少？为什是这么多?
  - HashMap的主要参数都有哪些?
  - HashMap是怎么处理hash碰撞的?
  - hash的计算规则?

HashMap 在多线程上有问题 那么在多线程环境下如何处理?

- Collections.synchronizedMap(Map) 创建线程安全的 Map 集合
  - SynchronziedMap 在内部维护了一个普通对象 Map ,还有一个互斥锁 mutex
  - 之后对 Map 进行操作的时候就会对方法上锁
- **Hashtable**
  - Hashtable 跟 HashMap 相比是线程安全的 但是在效率上不怎么乐观
  - 安全失败(fail-safe)机制
  - 与 HashMap 有什么区别?
    - 实现方式不同：Hashtable 继承了 Dictionary类，而 HashMap 继承的是 AbstractMap 类。
    - 初始化容量不同：HashMap 的初始容量为：16，Hashtable 初始容量为：11，两者的负载因子默认都是：0.75。
    - 扩容机制不同：当现有容量大于总容量 * 负载因子时，HashMap 扩容规则为当前容量翻倍，Hashtable 扩容规则为当前容量翻倍 + 1。
    - 迭代器不同：HashMap 中的 Iterator 迭代器是 fail-fast 的，而 Hashtable 的 Enumerator 不是 fail-fast 的。
- Concurrent HashMap

快速失败（fail-fast）与安全失败（fail-safe）

- 快速失败（fail—fast）
  - 在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出 Concurrent Modification Exception。
  - 原理
    - 迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。
    - 集合在被遍历期间如果内容发生变化，就会改变modCount的值。
    - 每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount变量是否为 expectedmodCount 值，是的话就返回遍历；否则抛出异常，终止遍历。
- 安全失败（fail—safe）
  - 采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。
  - 原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发Concurrent Modification Exception。
  - 缺点：基于拷贝内容的优点是避免了Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，

        - 即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。
    - 场景：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

ConcurrentHashMap 的底层结构

- JDK7
    - Segement 数组 + HashEntry 链表
    - HashEntry 使用 volatile 修饰数据和下一个节点
    - Segement继承于 ReentrantLock 使用分段锁技术
    - put 操作
        - 首先定位到 Segement 再进行 put 操作
        - Put 首先尝试获取锁，失败则自旋
        - 自旋次数过多进入阻塞状态等待唤醒再次进入争夺
    - Get 操作
        - Hash之后定位到具体的 Segement 再通过一次 Hash 定位到具体的元素
        - 因为数据是 volatile 修饰 所以每次取值都是最新值
        - Get 相对高效 因为不需加锁
    - 不足
        - Segement 的数量即为并发量 在某些场合并发量不够
        - 查询时还要遍历链表，效率较低
- JDK8
    - 使用 CAS + synchronized 来保证并发安全性
    - 将 JDK 7中的HashEntry 改成了 Node 但作用不变 值和 next 仍是 volatile 修饰
    - 引入红黑树 在链表长度大于 8 （小于 6）的时候会自动转换
    - Put 操作
        - 根据 key 计算 HashCode
        - 判断是否要进行初始化
        - 定位 Node 如果为空表示当前位置可以写入数据，利用 CAS 进行写入，失败则自旋保证成功
        - 如果当前位置 的 hashcode == MOVED == -1 则需要进行扩容
        - 如果都不满足 利用 synchronized 进行数据写入
        - 如果数量大于 TREEIFY_THERESHOLD 则转换为红黑树
        - 

    - Get 操作
        - 根据计算出来的 hashcode 寻址 在桶上直接返回值
        - 在红黑树上按照树的方式获取值
        - 在链表上按照链表方式获取值
- 常见面试问题
    - 谈谈你理解的 Hashtable，讲讲其中的 get put 过程。ConcurrentHashMap同问。
    - 1.8 做了什么优化?
    - 线程安全怎么做的?
    - 不安全会导致哪些问题?
    - 如何解决？有没有线程安全的并发容器?
    - ConcurrentHashMap 是如何实现的?
    - ConcurrentHashMap并发度为啥好这么多?
    - 1.7、1.8 实现有何不同？为什么这么做?
    - CAS是啥?
    - ABA是啥？场景有哪些，怎么解决?
    - synchronized底层原理是啥?
    - synchronized锁升级策略
    - 快速失败（fail—fast）是啥，应用场景有哪些？安全失败（fail—safe）同问。

Threadlocal

2020年3月2日 0:26

ThreadLocal用于给线程提供私有变量，每个线程通过set()和get()来对这个局部变量进行操作，但不会和其他线程的局部变量进行冲突，实现了线程的**数据隔离。**

用法解析

创建

```
ThreadLocal<String> mStringThreadLocal = new ThreadLocal<>();
```

set方法

```
mStringThreadLocal.set("droidyue.com");
```

get方法

```
mStringThreadLocal.get();
```

完整的使用示例

```
new Thread(() -> {
        ThreadLocal<String> stringThreadLocal = new ThreadLocal<>();
        stringThreadLocal.set("dsying");
        System.out.println(stringThreadLocal.get());
    }).start();
```

- 你也可以在创建ThreadLocal时初始化值

```
new Thread(() -> {
        ThreadLocal<String> stringThreadLocal = new ThreadLocal<String>() {
            @Override
            protected String initialValue() {
                return "dsying";
            }
        };
        System.out.println(stringThreadLocal.get());
}).start();
```

- 再进一步，你可以使用Java8的lambda表达式配合ThreadLocal的静态工厂方法

```
    new Thread(() -> {
        ThreadLocal<String> stringThreadLocal = ThreadLocal.withInitial(() -> "dsying");
        System.out.println(stringThreadLocal.get());
    }).start();
```

ThreadLocal内部实现

为了更好的掌握ThreadLocal，我认为了解其内部实现是很有必要的，这里我们以set方法从起始看一看ThreadLocal的实现原理。

存值

```
    public void set(T value) {
     // 获取当前线程
        Thread t = Thread.currentThread();
     // 获取当前线程的 threadLocals变量，该变量类型为ThreadLocalMap
        ThreadLocalMap map = getMap(t);
        if (map != null)
         // 如果当前线程的ThreadLocalMap不为空，则设置值
            map.set(this, value);
        else
         // 否则为当前线程创建ThreadLocalMap,并设置值
            createMap(t, value);
    }
    // 获取当前线程的threadLocals属性
    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }
    // 创建ThreadLocalMap，并赋值给当前线程的threadLocals属性
    void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }
```

上面的的getMap()方法其实获取的是Thread对象的threadLocals变量

```
    class Thread implements Runnable {
        /* ThreadLocal values pertaining to this thread. This map is maintained
         * by the ThreadLocal class. */
        ThreadLocal.ThreadLocalMap threadLocals = null;
    }
```

总结：实际上ThreadLocal的值是放入了当前线程的一个ThreadLocalMap实例中，所以只能在本线程中访问，其他线程无法访问。

取值

```
 public T get() {
     // 获取当前线程
        Thread t = Thread.currentThread();
     // 获取当前线程的 threadLocals变量，该变量类型为ThreadLocalMap
        ThreadLocalMap map = getMap(t);
        if (map != null) {
```

```
        // 如果 threadLocals不为空，以this，也就是当前ThreadLocal实例作为key去取值
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                // 如果 取到的value不为空，则返回
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
    // 如果当前线程的 threadLocals变量为空，则去初始化
        return setInitialValue();
    }

    private T setInitialValue() {
        T value = initialValue();
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
        return value;
    }
```

如果你在创建ThreadLocal时重写了initValue方法，则createMap时一并初始化，例如：

```
    new Thread(() -> {
        ThreadLocal<String> stringThreadLocal = ThreadLocal.withInitial(() -> "dsying");
        System.out.println(stringThreadLocal.get());
    }).start();
```

用什么存

通过上面的讲解我们已经知道，线程是通过各自的threadLocals属性去存储**私有变量**的

```
    class Thread implements Runnable {
        /* ThreadLocal values pertaining to this thread. This map is maintained
         * by the ThreadLocal class. */
        ThreadLocal.ThreadLocalMap threadLocals = null;
    }
```

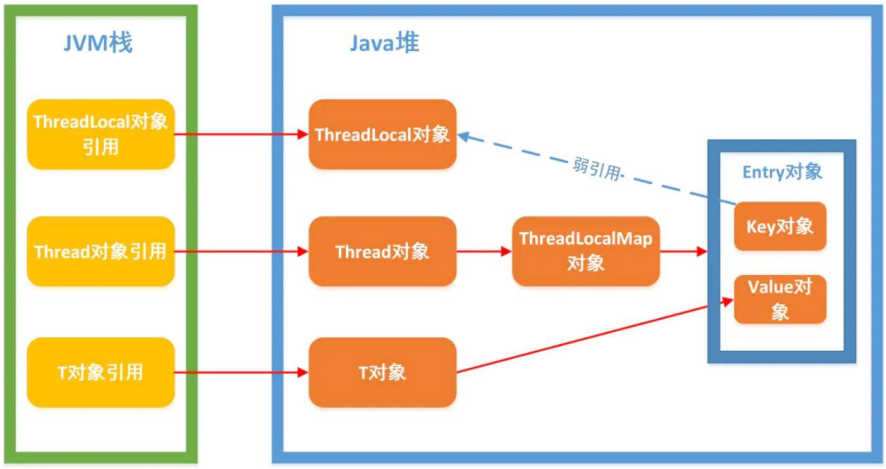
通过源码可以看到，threadLocals是一个ThreadLocalMap 类的对象

```
    static class ThreadLocalMap {
        /**
         * The entries in this hash map extend WeakReference, using
         * its main ref field as the key (which is always a
         * ThreadLocal object).  Note that null keys (i.e. entry.get()
         * == null) mean that the key is no longer referenced, so the
         * entry can be expunged from table.  Such entries are referred to
         * as "stale entries" in the code that follows.
         */
        static class Entry extends WeakReference<ThreadLocal<?>> {
            /** The value associated with this ThreadLocal. */
            Object value;

            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }

         // ...
    }
```

通过上面我们可以发现的是ThreadLocalMap是ThreadLocal的一个**静态内部类。**用Entry类来进行存储，而Entry的key永远是一个ThreadLocal对象。所以ThreadLocal的set和get方法里面存取值都是以当前ThreadLocal实例，也就是this作为key

```
    // 存
    map.set(this, value);
    // 取
    ThreadLocalMap.Entry e = map.getEntry(this);
```

ThreadLocal和Thread的关系



1. 每个Thread对象维护着一个ThreadLocalMap的引用
2. ThreadLocalMap是ThreadLocal的内部类，用Entry来进行存储
3. 调用ThreadLocal的set()方法时，实际上就是往ThreadLocalMap设置值，key是ThreadLocal对象，值是传递进来的对象
4. 调用ThreadLocal的get()方法时，实际上就是往ThreadLocalMap获取值，key是ThreadLocal对象
5. **ThreadLocal本身并不存储值**，它只是作为一个key来让线程从ThreadLocalMap获取value，正因为这个原理，所以ThreadLocal能够实现“**数据隔离**”，获取当前线程的局部变量值，不受其他线程影响~

是否会导致内存泄漏

有网上讨论说ThreadLocal会导致**内存泄露**，原因如下

- 首先ThreadLocal实例被线程的ThreadLocalMap实例持有，也可以看成被线程持有。
- 如果应用使用了线程池，那么之前的线程实例处理完之后出于复用的目的依然存活
- 所以，ThreadLocal设定的值被持有，导致**内存泄露**。

上面的逻辑是清晰的，可是ThreadLocal并不会产生内存泄露，因为ThreadLocalMap在选择key的时候，并不是直接选择ThreadLocal实例，而是ThreadLocal实例的**弱引用**

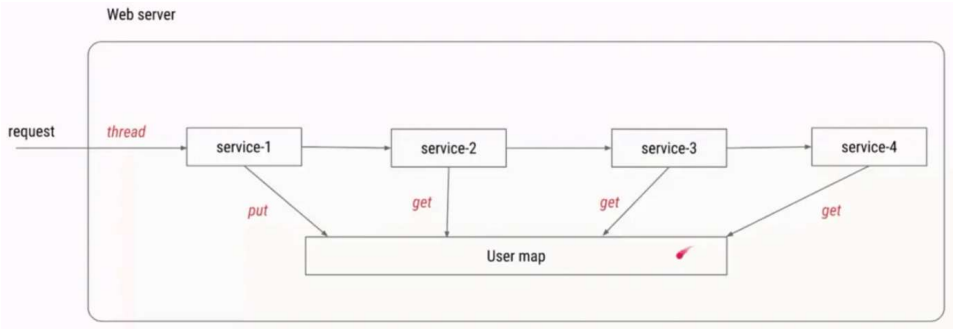
```
    static class ThreadLocalMap {
        static class Entry extends WeakReference<ThreadLocal<?>> {
            Object value;
            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }

         // ...
    }
```

我们可以看到class Entry extends WeakReference ，所以实际上从ThreadLocal设计角度来说是不会导致**内存泄露**的。**应用场景**

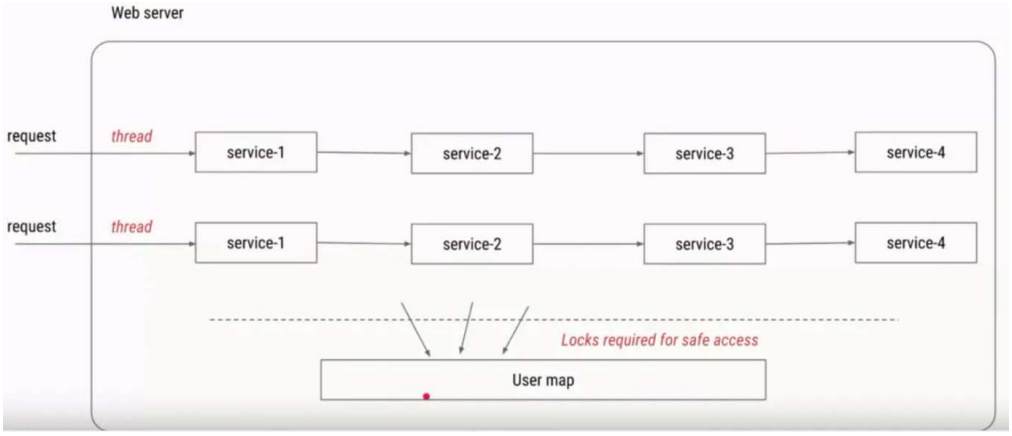
上面吧啦吧啦说了一大堆，那ThreadLocal在什么场景下有用呢？我们可以通过下面一个例子来看一下ThreadLocal具体的使用场景。

一个普通的http请求，会经过4层service,每层service都需要拿到当前请求的用户信息User

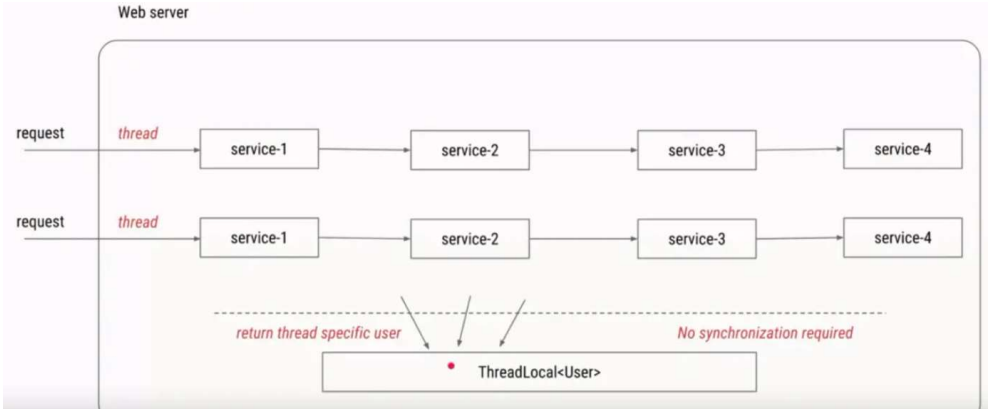
如果每个方法都需要绑定User参数的话就太麻烦了，所以我们可以把User放到User map中，在service-1中存储，2，3，4直接取值就可以了。



但是上面这种方法在多线程下就行不通了，我们知道多个线程对同一对象操作时会引发线程安全



当然，你可以使用ConcurrentHashMap 但我们可以有更好的方法， 通过ThreadLocal保存每个线程独有的User对象，只要在线程中，所有service读取到的肯定是当前线程独有的User



让我们通过代码来实现以下

```
public class UserContextHolder {
    public static ThreadLocal<User> holder = new ThreadLocal();
}

class Service1 {
     public void process() {
        User user = getUser();
        UserContextHolder.holder.set(user);
    }
}

class Service2 {
    public void process() {
        User user = UserContextHolder.holder.get();
        // process user
```

Set it for this thread

Get user for this thread

我们只需在最后一处service删除User信息即可

```
public class UserContextHolder {
    public static ThreadLocal<User> holder = new ThreadLocal();
}

class Service1 {
    // put user
}

class Service2 {
    // get user
}

class Service3 {
    // get user
}

class Service4 {
    public void process() {
        // get user
        // cleanup
        UserContextHolder.holder.remove();
    }
}
```

Last service, user no longer required

Atomic

2020年3月2日 0:26

Atomic包中的类可以分成4组：

AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference

AtomicIntegerArray，AtomicLongArray

AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

AtomicMarkableReference，AtomicStampedReference，AtomicReferenceArray

CAS虽然很高效的实现了原子操作，但是它依然存在三个问题。

1、ABA问题。CAS在操作值的时候检查值是否已经变化，没有变化的情况下才会进行更新。但是如果一个值原来是A，变成B，又变成A，那么CAS进行检查时会认为这个值没有变化，但是实际上却变化了。ABA问题的解决方法是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就变成1A-2B－3A。从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。

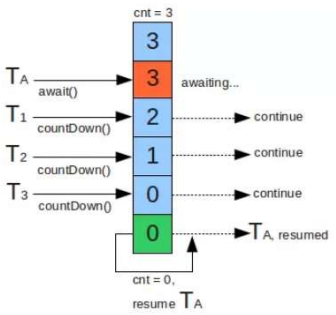
2、并发越高，失败的次数会越多，CAS如果长时间不成功，会极大的增加CPU的开销。因此CAS不适合竞争十分频繁的场景。

3、只能保证一个共享变量的原子操作。当对多个共享变量操作时，CAS就无法保证操作的原子性，这时就可以用锁，或者把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

并发工具类

2020年3月2日 0:26

CountDownLatch 等待多线程完成



CountDownLatch允许一个或多个线程等待其他线程完成操作。

来一个线程阻塞一次，直到达到指定的数量后，全部唤醒

实现原理也基本和显式锁类似，不同点依然在于对 state 的控制，CountDownLatch 只判断 state 是否等于零，不等于零就说明时机未到，阻塞当前线程。

而每一次的 countDown 方法调用都会减少一次倒计时资源，直至为零才唤醒阻塞的线程。

譬如：解析一个 excel，一个线程解析一个 sheet 页，当所有线程解析完成之后，提示解析完成。可以使用 join 来实现，也可以用 CountDownLatch。

使用 join

join 让当前执行线程等待 join 线程执行结束。

```
public class JoinCountDownLatchTest {
    public static void main(String[] args) {
        Thread parser1 = new Thread(new Runnable(){
            @Override
            public void run() {

            }
        });
        Thread parser2 = new Thread(new Runnable(){
            @Override
            public void run() {
                try {
                    Thread.sleep(1000*2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("parser2 finished!");
            }
        });
        parser1.start();
        parser2.start();
        try {
            parser1.join();
            parser2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("All parser finished!");
    }
}
```

使用 CountDownLatch

CountDownLatch 传入一个N当做计数器，每次执行countDown的时候N就会减1，CountDownLatch的await方法就会阻塞当前线程，直到N变成零。countDown可以是一个线程中的N个步骤或者是N个线程。

一个线程调用countDown方法，一个线程调用await方法。

```
public class CountDownLatchTest {
    public static CountDownLatch cdl = new CountDownLatch(2);

    public static void main(String[] args) {
        new Thread(new Runnable(){
            @Override
            public void run() {
                System.out.println(1);
                cdl.countDown();
                System.out.println(2);
                cdl.countDown();
            }
        }).start();
        try {
```

```
            cdl.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(3);
    }
}
```

CyclicBarrier 同步屏障

让一组线程到达一个屏障（或者是同步点）的时候被阻塞，直到最后一个线程到达屏障，屏障才会打开，所有的线程继续往下执行。

```
public class CyclicBarrierTest {
    static CyclicBarrier c = new CyclicBarrier(2);

    public static void main(String[] args) throws Exception {
        new Thread(new Runnable(){
            @Override
            public void run() {
                try {
                    Thread.sleep(3000);
                    System.out.println(2);
                    c.await();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        System.out.println(1);
        c.await();
        System.out.println(3);
    }
}
```

CyclicBarrier升级版

高级的构造方法 CyclicBarrier(int parties, Runnable barrierAction)：当所有线程到达同步点之后，优先执行barrierAction，等待该线程执行完之后，再继续执行await后面的方法。

```
public class CyclicBarrierTest {
    static CyclicBarrier c = new CyclicBarrier(2, new A());

    public static void main(String[] args) throws Exception {
        new Thread(new Runnable(){
            @Override
            public void run() {
                try {
                    Thread.sleep(3000);
                    System.out.println(2);
                    c.await();
                    System.out.println(2.1);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        System.out.println(1);
        c.await();
        System.out.println(1.1);
    }

    static class A implements Runnable{
        @Override
        public void run() {
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(3);
        }
    }
}
```

CyclicBarrier和CountDownLatch的区别

1. CountDownBatch 的计数器只能使用一次，而 CyclicBarrier 的计数器可以使用 reset() 方法重置。因此 CyclicBarrier 可以实现更加复杂的功能。例如：处理计算错误，可以重置计数器，让线程重新执行一次。
2. CountDownLatch 是计数器，线程来一个就记一个，此期间不阻塞线程，当达到指定数量之后才会去唤醒外部等待的线程。CyclicBarrier 像一个栅栏，来一个线程阻塞一个，直到阻塞了指定数量的线程后，一次性全部激活。

CyclicBarrier的其他方法：

- getNumberWaiting：获取阻塞的线程数量。
- isBroken()用来了解阻塞的线程是否被中断。

```
public class CyclicBarrierTest2 {
    static CyclicBarrier c = new CyclicBarrier(2);

    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable(){
            @Override
            public void run() {
```

```
                try {
                    c.await();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
        t1.start();
        t1.interrupt();
        try {
            c.await();
        } catch (Exception e) {//这里为什么会抛出异常呢？？？不明白
            System.out.println(c.isBroken());
            e.printStackTrace();
        }
    }
}
```

控制并发线程数的 Semaphore

Semaphore（信号量）用来控制同时访问特定资源的线程数量，通过协调各个线程，以保证合理的使用公共资源。
有一种场景是，临界资源允许多个线程同时访问，超过限定数量的外的线程得阻塞等待。

- int availablePermits：返回此信号量中当前可用的许可证数。
- int getQueueLength()：返回正在等待获取许可证的线程数。
- boolean hasQueueThreads：是否有线程正在等待获取许可证
- void reducePermits(int reduction)：减少reduction个许可证。
- Collection getQueuedThreads()：返回所有等待获取许可证的线程集合

semaphore 的内部原理和 ReentrantLock 的实现极其类似，包括公平与非公平策略的支持，只不过，AQS 里面的 state 在前者的实现中，一般小于等于一（除非重入锁），而后者的 state 则小于等于十，记录的是剩余可用临界资源数量。

所以，semaphore 天生就存在一个问题，如果某个线程重入了临界区，可用临界资源的数量是否需要减少？

停车场一共十个停车位，一辆车进去并占有了一个停车位，过了一段时间，这个向管理员报告，我还要占用一个停车位，先不管他占两个干啥，此时的管理员会同意吗？

实际上，在 Java 这个管理员看来，已经进入临界区的线程是「老爷」，提出的要求都会优先满足，即便他自身占有的资源并没有释放。

所以，在 Semaphore 机制里，一个线程进入临界区之后占用掉所有的临界资源都是可能的。

下面同时开启了30个线程，都进入了run方法内，但是同时运行在s.acquire();***s.release();之间的只能有10个线程。

```
public class SemaphoreTest {
    private static final int THREAD_COUNT = 30;
    private static ExecutorService threadPool = Executors.newFixedThreadPool(THREAD_COUNT);
    private static Semaphore s = new Semaphore(10);

    public static void main(String[] args) {
        for(int i=0; i<THREAD_COUNT; i++){
            threadPool.execute(new MyThread(i, s));
        }
        threadPool.shutdown();
    }

}
class MyThread implements Runnable{
    int c = 0;
    Semaphore s;
    public MyThread(int c, Semaphore s) {
        this.c = c;
        this.s = s;
    }
    @Override
    public void run() {
        try {
            System.out.println(c + " begin:");
            s.acquire();
            System.out.println("saveDate=" + c);
            Thread.sleep(3000);
            s.release();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

线程之间交换数据 Exchanger

Exchanger是个用于线程间协作的工具类，用于线程之间的数据交换。
它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。第一个线程先执行 exchange() 方法，第二个线程也执行 exchange() 方法，当两个线程同时到达同步点，这两个线程就可以交换数据。
如果一个线程一直没有执行 exchange() 方法，那么会一直等下去，如果担心特殊情况，可以使用 exchange(V v,longtimeout, TimeUnit unit) 设置最大等待时间。

```
public class ExchangerTest {
    private static final Exchanger<String> exgr = new Exchanger<String>();
    private static ExecutorService threadPool = Executors.newFixedThreadPool(2);
```

```
    public static void main(String[] args) {
        threadPool.execute(new Runnable(){
            @Override
            public void run() {
                String a = "银行流水A";
                try {
                    String b = exgr.exchange(a);
                    System.out.println("a中数据交换完毕.a=" + a+";b="+b);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        threadPool.execute(new Runnable(){
            @Override
            public void run() {
                String b = "银行流水B";
                try {
                    Thread.sleep(3000);
                    String a = exgr.exchange(b);//传递b数据并获得a的数据
                    System.out.println("b中数据交换完毕.a=" + a+";b="+b);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

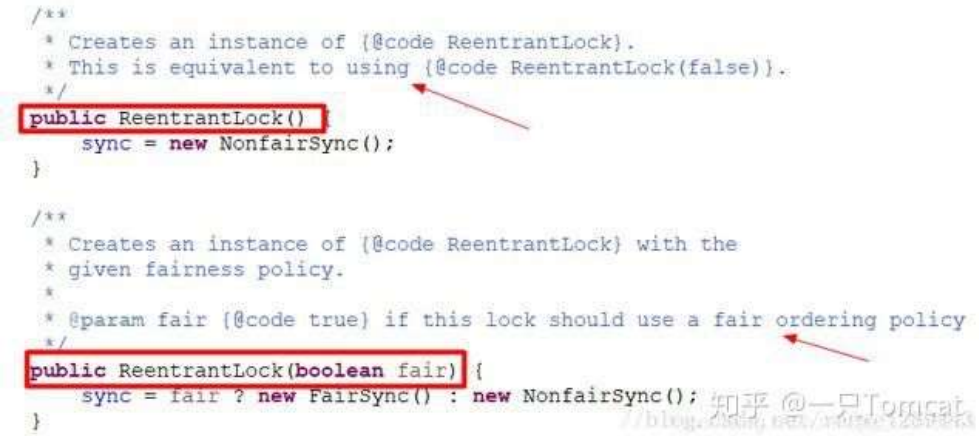
锁

2020年3月2日 0:26

锁{
    可重入锁{
        读写锁{
        }
    }
    乐观锁{
        CAS 机制
    }
    悲观锁{
    }
    JVM 对锁的优化{
        自旋锁{
            解释
            自适应自旋
        }
        锁分离{
            读写锁{
            }
        }
    }
}

01 由 ReentrantLock 和 synchronized 实现的一系列锁

jdk1.5 的 java.util.concurrent 并发包中的 Lock 接口和 1.5 之前的 synchronized 或许是我们最常用的同步方式，这两种同步方式特别是 Lock 的 ReentrantLock 实现，经常拿来进行比较，其实他们有很多相似之处，其实它们在实现同步的思想上大致相同，只不过在一些细节的策略上（诸如抛出异常是否自动释放锁）有所不同。前边说过了，本文着重讲锁的实现思想和不同锁的概念与分类，不对实现原理的细节深究，因此我在下面介绍第一类锁的时候经常讲他们放在一起来说。我们先来说一下 Lock 接口的实现之一 ReentrantLock。当我们想要创建 ReentrantLock 实例的时候，jdk 为我们提供两种重载的构造函数，如图：



fair 是什么意思？公平的意思，没错，这就是我们要说的第一种锁。

1.1 从其它等待中的线程是否按顺序获取锁的角度划分 - 公平锁与非公平锁

- 公平锁：是指多个线程在等待同一个锁时，必须按照申请锁的先后顺序来一次获得锁。所用公平锁就好像在餐厅的门口安装了一个排队的护栏，谁先来的谁就站的靠前，无法进行插队，当餐厅中的人用餐结束后会把钥匙交给排在最前边的那个人，以此类推。公平锁的好处是，可以保证每个排队的人都有饭吃，先到先吃后到后吃。但是弊端是，要额外安装排队装置。
- 非公平锁：理解了公平锁，非公平锁就很好理解了，它无非就是不用排队，当餐厅里的人出来后将钥匙往地上一扔，谁抢到算谁的。但是这样就造成了一个问题，那些身强体壮的人可能总是会先抢到钥匙，而那些身体瘦小的人可能一直抢不到，这就有可能将一直抢不到钥匙，最后导致需要很长时间才能拿到钥匙甚至一直拿不到直至饿死。

公平锁与非公平所的总结：

（1） 公平锁的好处是等待锁的线程不会饿死，但是整体效率相对低一些；非公平锁的好处是整体效率相对高一些，但是有些线程可能会饿死或者说很早就在等待锁，但要等很久才会获得锁。其中的原因是公平锁是严格按照请求所的顺序来排队获得锁的，而非公平锁时可以抢占的，即如果在某个时刻有线程需要获取锁，而这个时候刚好锁可用，那么这个线程会直接抢占，而这时阻塞在等待队列的线程则不会被唤醒。

（2） 在 java 中，公平锁可以通过 new ReentrantLock (true) 来实现；非公平锁可以通过 new ReentrantLock (false) 或者默认构造函数 new ReentrantLock () 实现。

（3）synchronized 是非公平锁，并且它无法实现公平锁。

1.2 从能否有多个线程持有同一把锁的角度划分 - 互斥锁

互斥锁的概念非常简单，也就是我们常说的同步，即一次最多只能有一个线程持有的锁，当一个线程持有该锁的时候其它线程无法进入上锁的区域。在 Java 中 synchronized 就是互斥锁，从宏观概念来讲，互斥锁就是通过悲观锁的理念引出来的，而非互斥锁则是通过乐观锁的概念引申的。

1.3 从一个线程能否递归获取自己的锁的角度划分 - 重入锁（递归锁）

我们知道，一条线程若想进入一个被上锁的区域，首先要判断这个区域的锁是否已经被某条线程所持有。如果锁正在被持有那么线程将等待锁的释放，但是这就引发了一个问题，我们来看这样一段简单的代码：

```
public class ReentrantDemo {
    private Lock mLock;

    public ReentrantDemo(Lock mLock) {
        this.mLock = mLock;
    }

    public void outer() {
        mLock.lock();
        inner();
        mLock.unlock();
    }

    public void inner() {
```

```
        mLock.lock();
        // do something
        mLock.unlock();
    }
}
```

当线程 A 调用 outer () 方法的时候，会进入使用传进来 mlock 实例来进行 mlock.lock () 加锁，此时 outer () 方法中的这片区域的锁 mlock 就被线程 A 持有了，当线程 B 想要调用 outer () 方法时会先判断，发现这个 mlock 这把锁被其它线程持有了，因此进入等待状态。

我们现在不考虑线程 B，单说线程 A，线程 A 进入 outer () 方法后，它还要调用 inner () 方法，并且 inner () 方法中使用的也是 mlock () 这把锁，于是接下来有趣的事情就来了。

按正常步骤来说，线程 A 先判断 mlock 这把锁是否已经被持有了，判断后发现这把锁确实被持有了，但是可笑的是，是 A 自己持有的。那你说 A 能否在加了 mlock 锁的 outer () 方法中调用加了 mlock 锁的 inner 方法呢？答案是如果我们使用的是可重入锁，那么递归调用自己持有的那把锁的时候，是允许进入的。

- 可重入锁：可以再次进入方法 A，就是说在释放锁前此线程可以再次进入方法 A（方法 A 递归）。
- 不可重入锁（自旋锁）：不可以再次进入方法 A，也就是说获得锁进入方法 A 是此线程在释放锁前唯一的一次进入方法 A。

下面这段代码演示了不可重入锁：

```
public class Lock{
    private boolean isLocked = false;
    public synchronized void lock()
        throws InterruptedException{
        while(isLocked){
            wait();
        }
        isLocked = true;
    }

    public synchronized void unlock(){
        isLocked = false;
        notify();
    }
}
```

可以看到，当 isLocked 被设置为 true 后，在线程调用 unlock () 解锁之前不管线程是否已经获得锁，都只能 wait ()。

1.4 从编译器优化的角度划分 – 锁消除和锁粗化

锁消除和锁粗化，是编译器在编译代码阶段，对一些没有必要的、不会引起安全问题的同步代码取消同步（锁消除）或者对那些多次执行同步的代码且它们可以合并到一次同步的代码（锁粗化）进行的优化手段，从而提高程序的执行效率。

（1）锁消除

对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判断依据是来源于逃逸分析的数据支持，如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而能被其他线程访问到，那就可以把他们当做栈上数据对待，认为他们是线程私有的，同步加锁自然就无需进行。

来看这样一个方法：

```
    public String concatString(String s1, String s2, String s3)
    {
        StringBuffer sb = new StringBuffer();
        sb.append(s1);
        sb.append(s2);
        sb.append(s3);
        return sb.toString();
    }
```

源码中 StringBuffer 的 append 方法定义如下：

```
    public synchronized StringBuffer append(StringBuffer sb) {
        super.append(sb);
        return this;
    }
```

可见 append 的方法使用 synchronized 进行同步，我们知道对象的实例总是存在于堆中被多线程共享，即使在局部方法中创建的实例依然存在于堆中，但是对该实例的引用是线程私有的，对其他线程不可见。即上边代码中虽然 StringBuffer 的实例是共享数据，但是对该实例的引用确是每条线程内部私有的。不同的线程引用的是堆中存在的不同的 StringBuffer 实例，它们互不影响互不可见。也就是说在 concatString () 方法中涉及了同步操作。但是可以观察到 sb 对象它的作用域被限制在方法的内部，也就是 sb 对象不会 “逃逸” 出去，其他线程无法访问。因此，虽然这里有锁，但是可以被安全的消除，在即时编译之后，这段代码就会忽略掉所有的同步而直接执行了。

（2）锁粗化

原则上，我们在编写代码的时候，总是要将同步块的作用范围限制的尽量小 —— 只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁禁止，那等待的线程也能尽快拿到锁。大部分情况下，这些都是正确的。但是，如果一系列的联系操作都是同一个对象反复加上和解锁，甚至加锁操作是出现在循环体中的，那么即使没有线程竞争，频繁地进行互斥同步操作也导致不必要的性能损耗。

举个案例，类似上面锁消除的 concatString () 方法。如果 StringBuffer sb = new StringBuffer (); 定义在方法体之外，那么就会有线程竞争，但是每个 append () 操作都对同一个对象反复加锁解锁，那么虚拟机探测到有这样的情况的话，会把加锁同步的范围扩展到整个操作序列的外部，即扩展到第一个 append () 操作之前和最后一个 append () 操作之后，这样的一个锁范围扩展的操作就称之为锁粗化。

1.5 在不同的位置使用 synchronized-- 类锁和对象锁

这是最常见的锁了，synchronized 作为锁来使用的时候，无非就只能出现在两个地方（其实还能修饰变量，但作用是保证可见性，这里讨论锁，故不阐述）：代码块、方法（一般方法、静态方法）。由于可以使用不同的类型来作为锁，因此分成了类锁和对象锁。

- 类锁：使用字节码文件（即.class）作为锁。如静态同步函数（使用本类的.class），同步代码块中使用.class。
- 对象锁：使用对象作为锁。如同步函数（使用本类实例，即 this），同步代码块中是用引用的对象。

下面代码涵盖了所有 synchronized 的使用方式：

```
public class Demo {
    public Object obj = new Object();
     // 静态同步函数,使用本类字节码做类锁（即Demo.class）
    public static synchronized void method1() {
    }

    public void method2() {
        // 同步代码块，使用字节码做类锁
        synchronized (Demo.class) {
        }
    }

    // 同步函数，使用本类对象实例即this做对象锁
    public synchronized void method3() {
    }

    // 同步代码块，使用本类对象实例即this做对象锁
    public void method4() {
        synchronized (this) {
        }
    }

    public void method5() {
```

```
        //同步代码块，使用共享数据obj实例做对象锁。
        synchronized (obj) {
        }
    }
}
```

02 从锁的设计理念来分类 - 悲观锁、乐观锁

如果将锁在宏观上进行大的分类，那么所只有两类，即悲观锁和乐观锁。

悲观锁

- 悲观锁是就是悲观思想，即认为读少写多，遇到并发写的可能性高，每次去拿数据的时候都认为别人会修改，所以每次在读写数据的时候都会上锁，这样别人想读写这个数据就会 block 直到拿到锁。java 中的悲观锁就是 Synchronized,AQS 框架下的锁则是先尝试 cas 乐观锁去获取锁，获取不到，才会转换为悲观锁，如 RetreenLock。

乐观锁

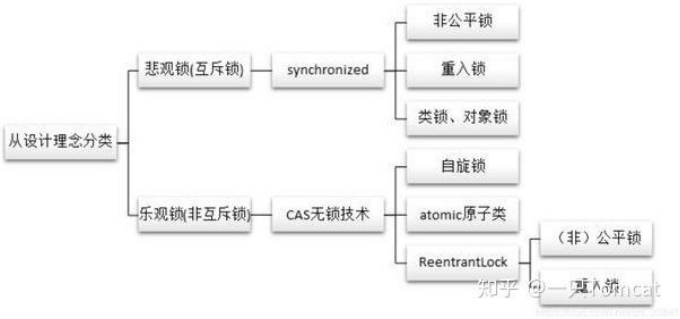
- 乐观锁是一种乐观思想，即认为读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读 - 比较 - 写的操作。
  乐观锁的实现思想 --CAS（Compare and Swap）无锁
  CAS 并不是一种实际的锁，它仅仅是实现乐观锁的一种思想，java 中的乐观锁（如自旋锁）基本都是通过 CAS 操作实现的，CAS 是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。
  乐观锁常见的两种实现方式
  乐观锁一般会使用版本号机制或 CAS 算法实现

1. 版本号机制
   一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时，version 值会加一。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。
2. CAS 算法
   即 compare and swap（比较与交换），是一种有名的无锁算法。
   无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）

CAS 算法涉及到三个操作数

- 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（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。



03 数据库中常用到的锁 - 共享锁、排它锁

共享锁和排它锁多用于数据库中的事物操作，主要针对读和写的操作。

而在 Java 中，对这组概念通过 ReentrantReadWriteLock 进行了实现，它的理念和数据库中共享锁与排它锁的理念几乎一致

一条线程进行读的时候，允许其他线程进入上锁的区域中进行读操作；当一条线程进行写操作的时候，不允许其他线程进入进行任何操作。

读 + 读可以存在，读 + 写、写 + 写均不允许存在

- 共享锁：也称读锁或 S 锁。如果事务 T 对数据 A 加上共享锁后，则其他事务只能对 A 再加共享锁，不能加排它锁。获准共享锁的事务只能读数据，不能修改数据。
- 排它锁：也称独占锁、写锁或 X 锁。如果事务 T 对数据 A 加上排它锁后，则其他事务不能再对 A 加任何类型的锁。获得排它锁的事务即能读数据又能修改数据。

04 由于并发问题产生的锁 - 死锁、活锁

4.1 死锁

（1）什么是死锁
所谓死锁是指多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。
（2）死锁形成的必要条件
产生死锁必须同时满足以下四个条件，只要其中任一条件不成立，死锁就不会发生：

- 互斥条件：进程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。
- 不剥夺条件：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。
- 请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。
- 循环等待条件：存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被 链中下一个进程所请求。
  - 即存在一个处于等待状态的进程集合 {Pl, P2, …, pn}，其中 Pi 等 待的资源被 P (i+1) 占有（i=0, 1, …, n-1)，Pn 等待的资源被 P0 占有

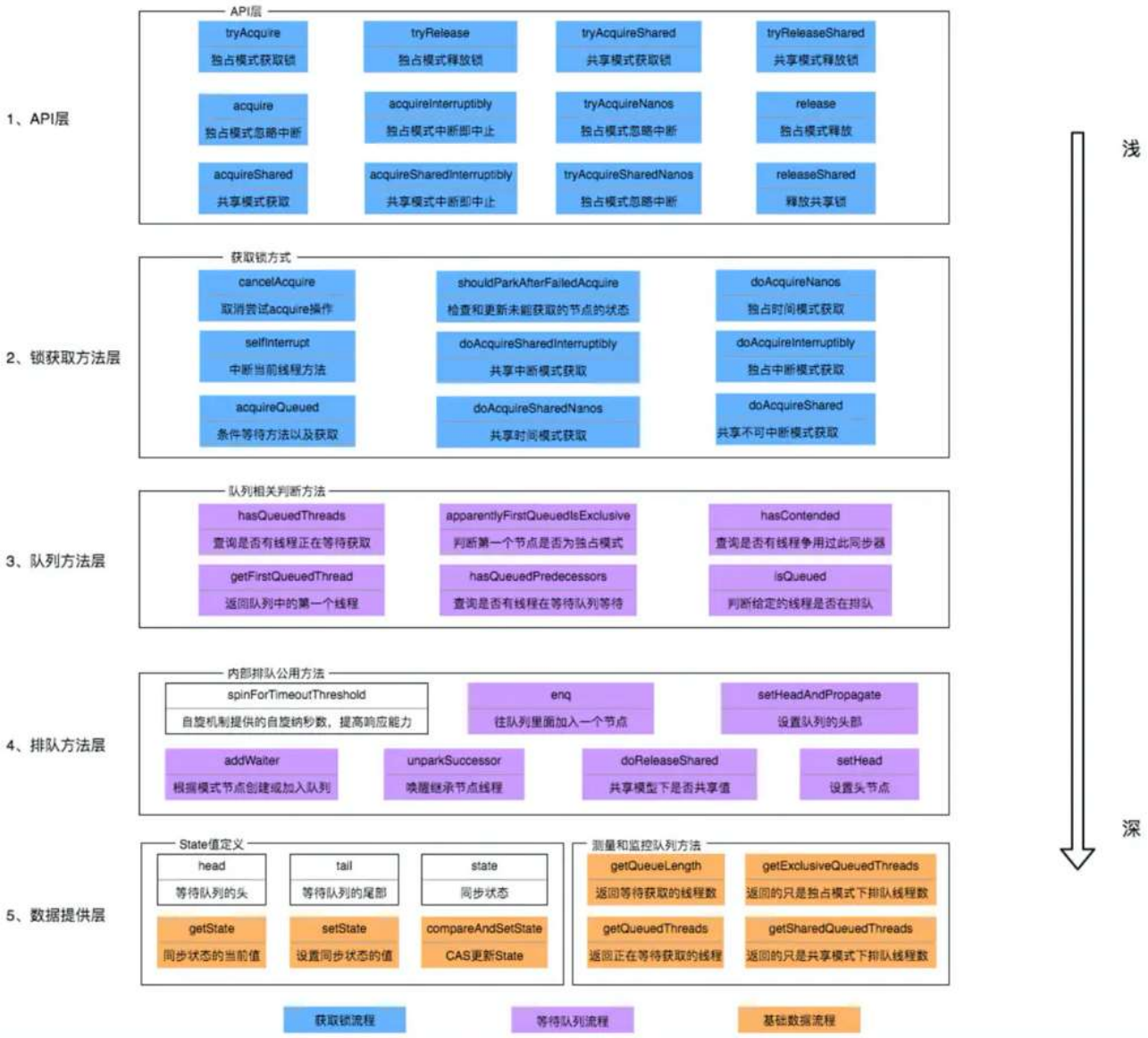
4.2 活锁

活锁和死锁在表现上是一样的两个线程都没有任何进展，区别在于：

死锁，两个线程都处于阻塞状态，说白了就是它不会再做任何动作，我们通过查看线程状态是可以分辨出来的。

活锁，并不会阻塞，而是一直尝试去获取需要的锁，不断的 try，这种情况下线程并没有阻塞所以是活的状态，我们查看线程的状态也会发现线程是正常的，但重要的是整个程序却不能继续执行了，一直在做无用功。

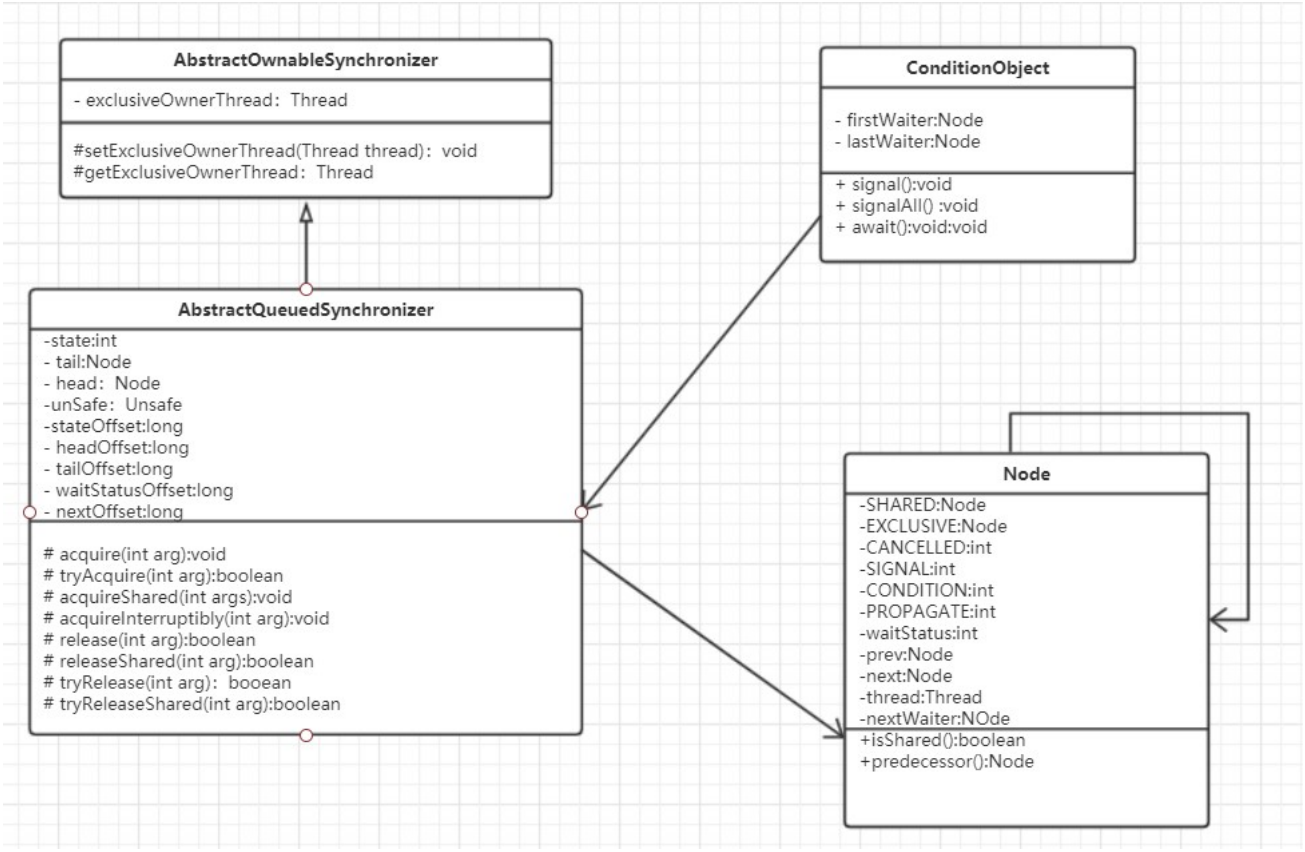
举个生动的例子的话，两个人都没有停下来等对方让路，而是都有很有礼貌的给对方让路，但是两个人都在不断朝路的同一个方向移动，这样只是在做无用功，还是不能让对方通过。



- 图中有颜色的为 Method，无颜色的为 Attribution。
- 当有自定义同步器接入时，只需重写第一层所需要的部分方法即可，不需要关 注底层具体的实现流程。当自定义同步器进行加锁或者解锁操作时，先经过第 一层的 API 进入 AQS 内部方法，然后经过第二层进行锁的获取，接着对于获 取锁失败的流程，进入第三层和第四层的等待队列处理，而这些处理方式均依 赖于第五层的基础数据提供层。

AQS

2020年2月7日 16:15



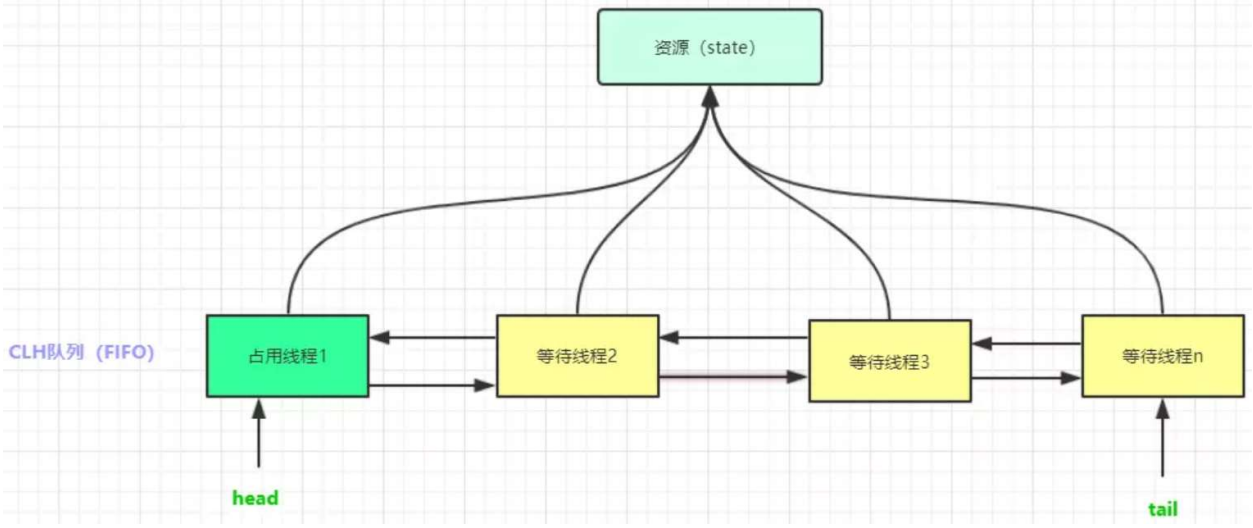
- State 状态的维护:

  设置 volatile 状态保证可见性

  使用 CAS 方法进行写入修改

  使用 final 修饰 get()\set() 方法禁止子类修改

CLH 队列



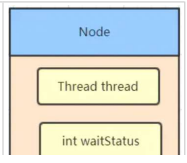
- CLH 队列 简明解释:
  - FIFO 的双向队列
  - 使用两个节点 head / tail 来记录队首和对尾的状态
  - 队列内元素类型为 Node
  - Node 类型元素简明解释:

保存:

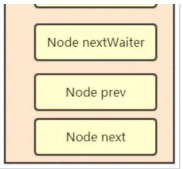
线程的引用 (thread)

状态 (waitStatus)

前驱节点 (prev)



后继节点（next）
Condition 队列的后续节点（nextWaiter）



- waitStatus 的几种状态：

SIGNAL	表示当前节点的后续节点中的线程通过 park 被阻塞了 当前节点在释放或取消时要通过 unpark 来解除他的阻塞
CANCELLED	表示当前节点的线程因为超时或中断被取消了
CONDITION	表示当前节点在 Condition 队列中
PROPAGATE	共享模式的头节点可能在此状态，表示无条件往下传播 引入此状态是为了优化锁竞争，使队列中线程有序地一个一个唤醒
0	一般是节点初始状态

- CLH 队列一般工作流程：
  - 当前线程获取同步补状态失败
    - AQS 将当前线程已经等待状态等信息构造成一个节点 （Node）
    - 将 Node 加入到 CLH 同步队列
    - 阻塞当前线程
  - 当同步状态释放
    - 使用公平锁将首节点唤醒
    - 首节点再次尝试获取同步状态
- CLH 入列
  - tail 指向新的节点 --> 新节点的 prev 指向当前最后的节点--> 当前最后一个结点的 next 指向当前节点、
  - addWaiter() 代码：

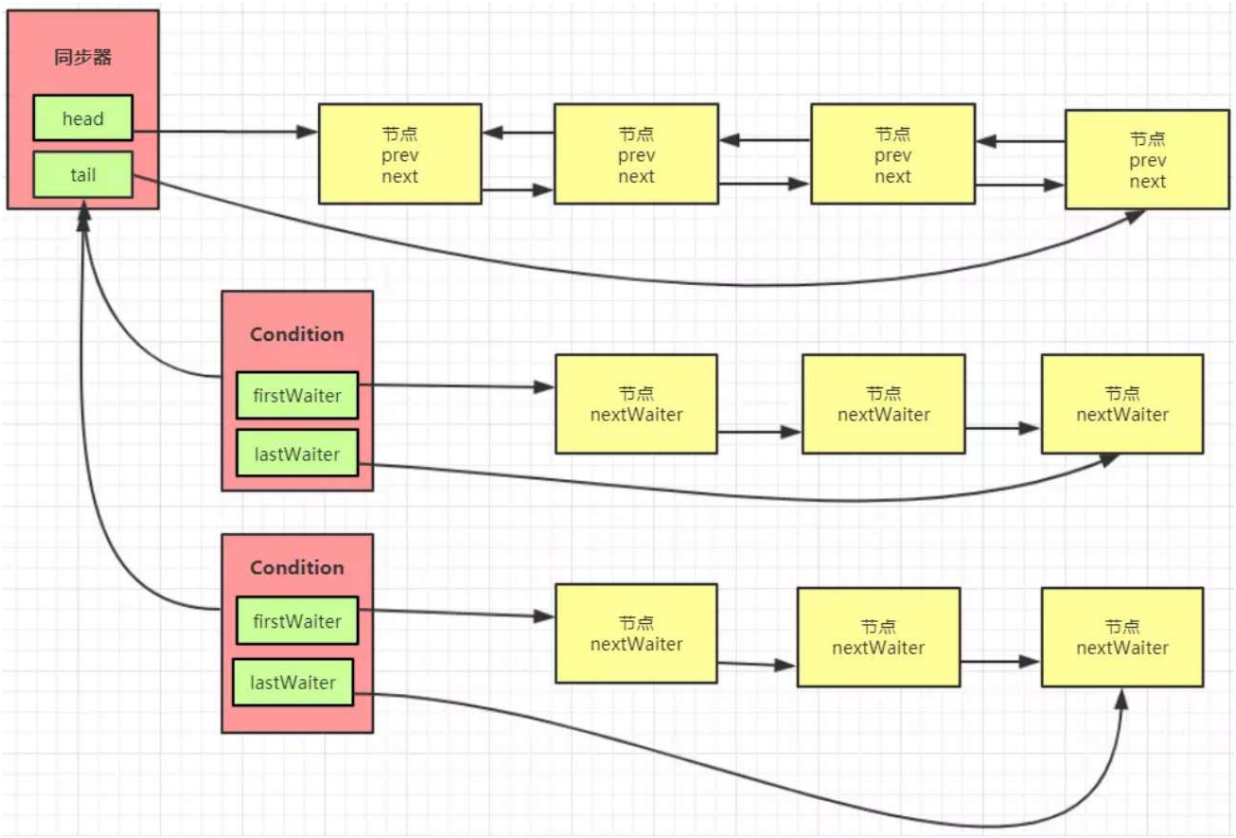
```
//构造Node
private Node addWaiter(Node mode) {
  Node node = new Node(Thread.currentThread(), mode);
        // 快速尝试添加尾节点
        Node pred = tail;
        if (pred != null) {
            node.prev = pred;
            //CAS设置尾节点
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        //多次尝试
        enq(node);
        return node;
        }
```

    - addWaiter设置尾节点失败的话，调用enq(Node node)方法设置尾节点

```
 private Node enq(final Node node) {
        //死循环尝试，知道成功为止
        for (;;) {
            Node t = tail;
            //tail 不存在，设置为尾节点
            if (t == null) { // Must initialize
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else {
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
```

- CLH 出列
  - 首节点的线程释放同步状态后，将会唤醒它的后继节点（next），后继节点将会在获取同步状态成功时将自己设置为首节点
- ConditionObject
  - 为 Lock 提供 await( ) / signal( ) / signalAll( ) 等方法，实现等待 / 通知机制

- ConditionObject 实现了 Condition 接口，为 AQS 提供条件变量支持
- ConditionObject 队列与 CLH 队列的关系



  - 调用了 await( ) 方法的线程。会被加入到 conditionObject 等待队列中，并且唤醒 CLH 队列中 head 节点的下一个节点
  - 线程在某个 ConditionObject 对象上调用了signal( ) 方法后，等待队列中的 firstWaiter 会被加入到 AQS 的 CLH 队列中，等待唤醒
  - 当前成调用 unLock( ) 方法释放锁时， CLH 队列中的 head 节点的下一个节点会被唤醒 (图中是firstWaiter)
  - ConditionObject 对象都维护了一个单独的等待队列，AQS 维护的 CLH 队列 是同步队列，他们节点类型相同，都是 Node

独占模式 与 共享模式

独占模式:

同一时刻仅有一个线程持有同步状态　可分为公平锁和非公平锁

公平锁：按照线程在队列中的排队顺序，先到者先拿到锁

非公平锁：当线程要获取锁时　五十队列顺序，直接去抢锁

在独占模式下，方法 acquire(long arg) 用于独占式获取同步状态

```
public final void acquire(long arg) {
      if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
          selfInterrupt();
   }
```

其中:

acquireQueued(final Node node, long arg) 方法详情如下:

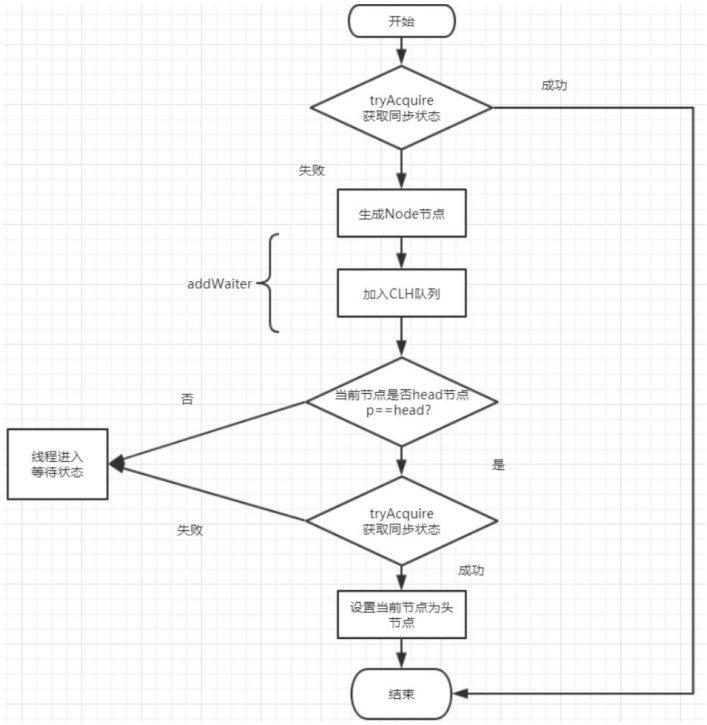
```
final boolean acquireQueued(final Node node, long arg) {
       boolean failed = true;
       try {
           boolean interrupted = false;
           for (;;) {
               final Node p = node.predecessor();
               if (p == head && tryAcquire(arg)) {
                   setHead(node);
                   p.next = null; // help GC
                   failed = false;
                   return interrupted;
               }
               if ( shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt() )
                   interrupted = true;
           }
       } finally {
           if (failed)
               cancelAcquire(node);
```

```
        }
    }
```

selfInterrupt( ) 方法详情如下

```
static void selfInterrupt() {
    Thread.currentThread().interrupt();
}
```

由此，可得出 acquire(long arg) 的流程图



共享模式：

多个线程可任意共同执行， Semaphore / CountDownLatch 都是共享式的

acquireShared(long arg) 是共享式获取同步状态的方法

```
  public final void acquireShared(long arg) {
        if (tryAcquireShared(arg) < 0)
            doAcquireShared(arg);
    }
```

由上可得，先调用tryAcquireShared(int arg)方法尝试获取同步状态

```
protected final boolean tryAcquire(int acquires) {
    //当前线程
    Thread current = Thread.currentThread();
    //获取状态
    int c = getState();
    //写线程数量（即获取独占锁的重入数）
    int w = exclusiveCount(c);

    //当前同步状态state != 0，说明已经有其他线程获取了读锁或写锁
    if (c != 0) {
        // 当前state不为0，此时：如果写锁状态为0说明读锁此时被占用返回false;
        // 如果写锁状态不为0且写锁没有被当前线程持有返回false
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;

        //判断同一线程获取写锁是否超过最大次数（65535），支持可重入
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        //更新状态
        //此时当前线程已持有写锁，现在是重入，所以只需要修改锁的数量即可。
        setState(c + acquires);
        return true;
    }
```

```
    //到这里说明此时c=0,读锁和写锁都没有被获取
    //writerShouldBlock表示是否阻塞
    if (writerShouldBlock() ||
        !compareAndSetState(c, c + acquires))
        return false;

    //设置锁为当前线程所有
    setExclusiveOwnerThread(current);
    return true;
}
```

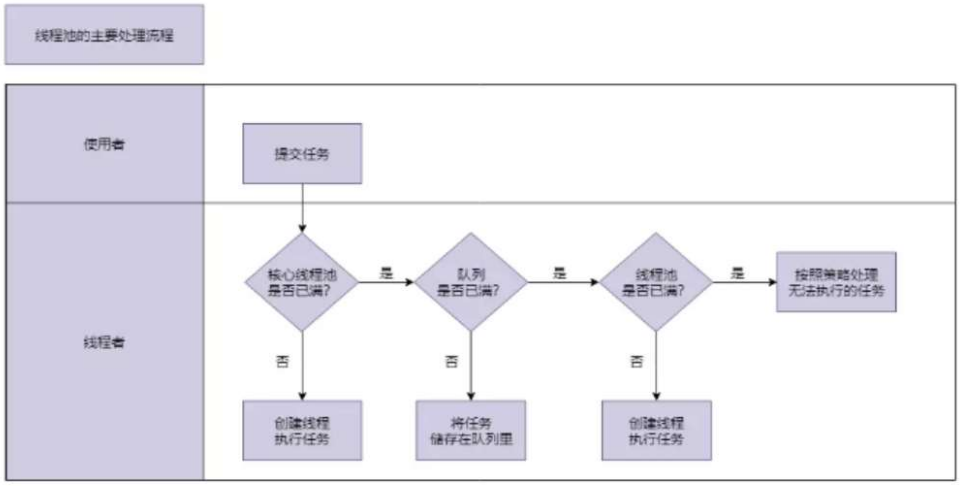
如果获取失败，调用 doAcquireShared(long arg) 自旋方式获取同步状态

```
private void doAcquireShared(long arg) {
        final Node node = addWaiter(Node.SHARED);
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                final Node p = node.predecessor();
                if (p == head) {
                    long r = tryAcquireShared(arg);
                    if (r >= 0) {
                        setHeadAndPropagate(node,r);
                        p.next = null; // help GC
                        if (interrupted)
                            selfInterrupt();
                        failed = false;
                        return;
                    }
                }
                if (shouldParkAfterFailedAcquire(p,node) && parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

线程池实现原理

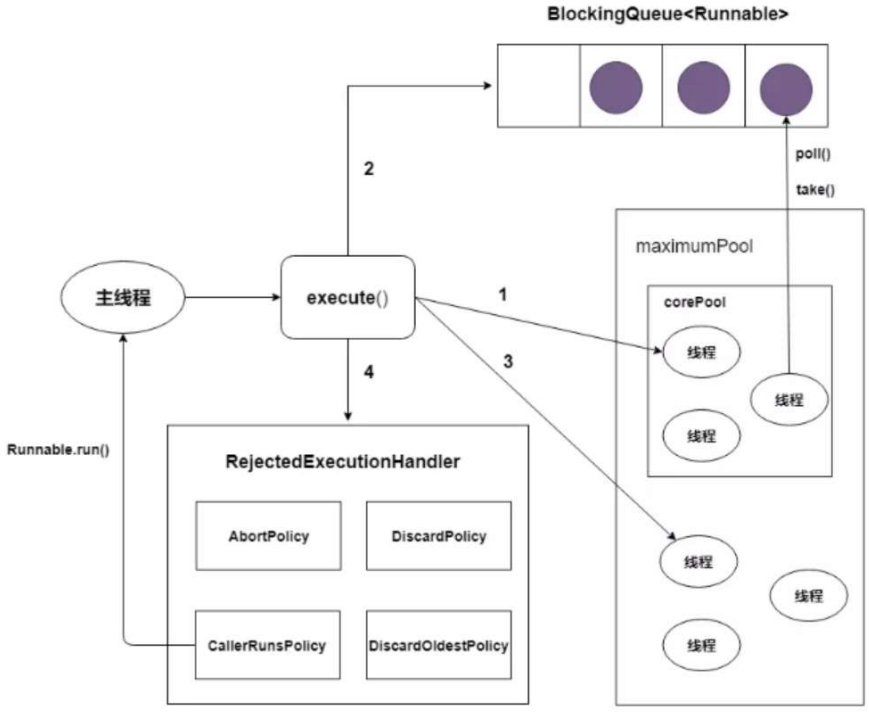
2020年2月5日 21:36

首先看线程池主要的处理流程



线程池的主要处理流程

ThreadPoolExecutor 执行 execute() 方法的示意图如下：

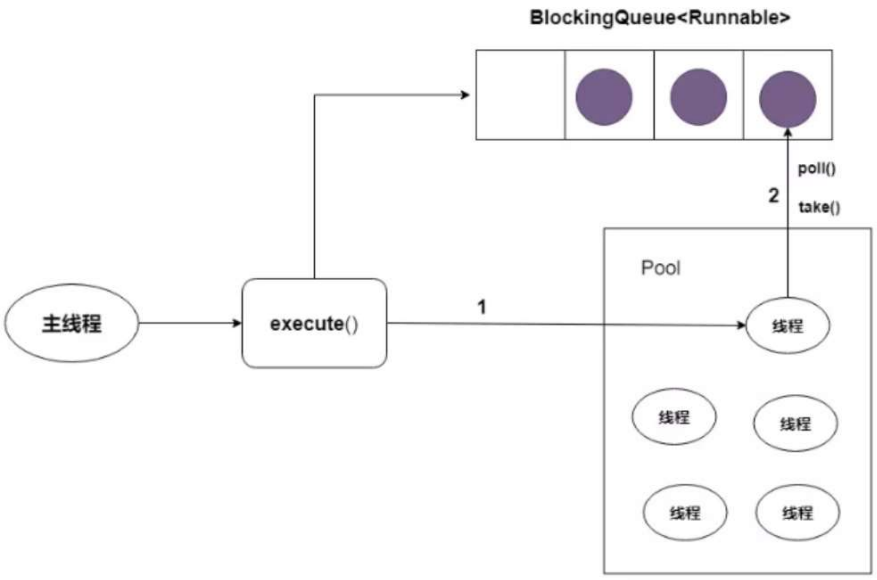


ThreadPoolExecutor执行示意图

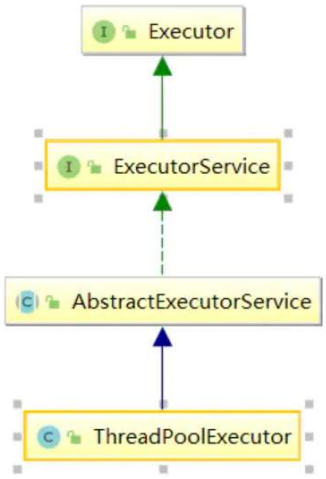
分别是四种情况，依次说明：

- 如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。上图1
- 如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。上图2
- 如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。上图3
- 如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。上图4

ThreadPoolExecutor中线程执行任务的示意图如下



线程池源码分析



- Executor接 口只有一个方法 execute ,传入线程任务参数
- ExecutorService 接口继承Executor接口，并增加了submit、shutdown、invokeAll等等一系列方法。
- AbstractExecutorService 抽象类实现 ExecutorService 接口
  - 提供了一些方法的默认实现，例如submit方法、invokeAny方法、invokeAll方法。
  - 像 execute 方法、线程池的关闭方法（shutdown、shutdownNow 等等）就没有提供默认的实现。
- ThreadPoolExecutor 最下面的底层实现

由继承结构，主要分析 ThreadPoolExecutor

线程池的五种状态：

- RUNNING - 接受新任务并且继续处理阻塞队列中的任务
- SHUTDOWN - 不接受新任务但是会继续处理阻塞队列中的任务
- STOP - 不接受新任务，不在执行阻塞队列中的任务，中断正在执行的任务
- TIDYING - 所有任务都已经完成，线程数都被回收，线程会转到TIDYING状态会继续执行钩子方法
- TERMINATED - 钩子方法执行完毕

线程之间的转换：

- RUNNING -> SHUTDOWN
  - 显式调用shutdown()方法，或者隐式调用了finalize()方法
- (RUNNING or SHUTDOWN) -> STOP
  - 显式调用shutdownNow()方法
- SHUTDOWN -> TIDYING
  - 当线程池和任务队列都为空的时候
- STOP -> TIDYING
  - 当线程池为空的时候
- TIDYING -> TERMINATED
  - 当 terminated() hook 方法执行完成时候

构造函数分析：

底层构造函数：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
}
```

参数	类型	含义
corePoolSize	int	核心线程数
maximumPoolSize	int	最大线程数
keepAliveTime	long	存活时间
unit	TimeUnit	时间单位
workQueue	BlockingQueue	存放线程的队列
threadFactory	ThreadFactory	创建线程的工厂
handler	RejectedExecutionHandler	多余的的线程处理器（拒绝策略）

- corePoolSize：核心线程数，即最小的keep alive线程数，如果allowCoreThreadTimeOut设置为true，则该参数无效，即为0；
- maximumPoolSize：最大线程数，即定义了线程池的最大线程数（实际最大值不能超过CAPACITY）；
- keepAliveTime：空闲时间，即线程的最大空闲时间，默认情况是当线程池中的线程数超过corePoolSize时，线程的最大空闲时间，及线程数小于corePoolSize时不生效，除非allowCoreThreadTimeOut设置为true；
- workQueue：任务队列，为阻塞队列，保存需要执行的任务。
- threadFactory： 创建线程的工厂类。
- handler： 当queue满了和线程数达到最大限制，对于继续到达的任务采取的策略。默认采取AbortPolicy ， 也就是拒绝策略。

拒绝策略分析：

1. AbortPolicy

```
 /**
 *  默认的拒绝策略,当继续有任务到来时直接抛出异常
 */
public static class AbortPolicy implements RejectedExecutionHandler {

    public AbortPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        throw new RejectedExecutionException("Task " + r.toString() +
                                             " rejected from " +
                                             e.toString());
    }
}
```

2. DiscardPolicy：rejectedexecution是个空方法，意味着直接抛弃该任务，不处理。

```
public static class DiscardPolicy implements RejectedExecutionHandler {

    public DiscardPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    }
```

3. DiscardOldestPolicy：抛弃queue中的第一个任务，再次执行该任务。

```
public static class DiscardOldestPolicy implements RejectedExecutionHandler {

    public DiscardOldestPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            e.getQueue().poll();
            e.execute(r);
        }
    }
}
```

4. CallerRunsPolicy： 直接由执行该方法的线程继续执行该任务，除非调用了shutdown方法，这个任务才会被丢弃，否则继续执行该任务会发生阻塞。

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {

    public CallerRunsPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
}
```

workQueue

用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列：

- rrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按FIFO（先进先出）原则对元素进行排序。
- LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂- 方法Executors.newFixedThreadPool()使用了这个队列。
- SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态- ，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
- PriorityBlockingQueue：一个具有优先级的无限阻塞队列。

execute方法

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
```

```
        /*
         *  3步操作
         *
         * 1. 如果当前运行的线程数<核心线程数,创建一个新的线程执行任务,调用addWorker方法原子性地检查
         *    运行状态和线程数,通过返回false防止不需要的时候添加线程
         * 2. 如果一个任务能够成功的入队,仍然需要双重检查,因为我们添加了一个线程(有可能这个线程在上次检查后就已经死亡了)
         *    或者进入此方法的时候调用了shutdown,所以需要重新检查线程池的状态,如果必要的话,当停止的时候要回滚入队操作,
         *    或者当线程池为空的话创建一个新的线程
         * 3. 如果不能入队,尝试着开启一个新的线程,如果开启失败,说明线程池已经是shutdown状态或饱和了,所以拒绝执行该任务
         */
        int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        else if (!addWorker(command, false))
            reject(command);
    }
```

1. 检查当前线程池中的线程数是否<核心线程数，如果小于核心线程数，就调用addWorker方法创建一个新的线程执行任务，addworker中的第二个参数传入true，表示当前创建的是核心线程。如果当前线程数>=核心线程数或者创建线程失败的话，直接进入第二种情况。
2. 通过调用isRunning方法判断线程池是否还在运行，如果线程池状态不是running，那就直接退出execute方法，没有执行的必要了；如果线程池的状态是running，尝试着把任务加入到queue中，再次检查线程池的状态， 如果当前不是running，可能在入队后调用了shutdown方法，所以要在queue中移除该任务，默认采用拒绝策略直接抛出异常。如果当前线程数为0，可能把allowCoreThreadTimeOut设为了true，正好核心线程全部被回收，所以必须要创建一个空的线程，让它自己去queue中去取任务执行。
3. 如果当前线程数>核心线程数，并且入队失败，调用addWorker方法创建一个新的线程去执行任务，第二个参数是false，表示当前创建的线程不是核心线程。这种情况表示核心线程已满并且queue已满，如果当前线程数小于最大线程数，创建线程执行任务。如果当前线程数>=最大线程数，默认直接采取拒绝策略。

addWorker方法

看一下addWorker是怎么具体执行的。代码有点长，慢慢看。

```
 private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);
            // Check if queue empty only if necessary.
            if (rs >= SHUTDOWN &&
                ! (rs == SHUTDOWN &&
                   firstTask == null &&
                   ! workQueue.isEmpty()))
                return false;
            for (;;) {
                int wc = workerCountOf(c);
                if (wc >= CAPACITY ||
                    wc >= (core ? corePoolSize : maximumPoolSize))
                    return false;
                if (compareAndIncrementWorkerCount(c))
                    break retry;
                c = ctl.get();  // Re-read ctl
                if (runStateOf(c) != rs)
                    continue retry;
                // else CAS failed due to workerCount change; retry inner loop
            }
        }
        boolean workerStarted = false;
        boolean workerAdded = false;
        Worker w = null;
        try {
            w = new Worker(firstTask);
            final Thread t = w.thread;
            if (t != null) {
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                    // Recheck while holding lock.
                    // Back out on ThreadFactory failure or if
                    // shut down before lock acquired.
                    int rs = runStateOf(ctl.get());
                    if (rs < SHUTDOWN ||
                        (rs == SHUTDOWN && firstTask == null)) {
                        if (t.isAlive()) // precheck that t is startable
                            throw new IllegalThreadStateException();
                        workers.add(w);
                        int s = workers.size();
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                        workerAdded = true;
                    }
                } finally {
                    mainLock.unlock();
                }
                if (workerAdded) {
                    t.start();
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                addWorkerFailed(w);
        }
```

```
        return workerStarted;
    }
```
首先判断线程池的runstate，如果runstate为shutdown，那么并不能满足第二个条件，runstate != shutdown，所以这是针对runstate不是running，shutdown的情况，当runstate>shutdown时，队列为空，此时仍然有任务的话，直接返回false，线程池已关闭，并不能在继续执行任务了。

第二个自旋操作就的目的就是对线程数量自增，由于涉及到高并发，所以采用了cas来控制，判断线程的workcount>=CAPACITY，那么直接返回false，或者通过判断是否核心线程，如果是true，判断workcount>=核心线程数，如果是false，判断workcount>=最大线程数，直接返回false。如果不满足上个条件，直接使用cas把线程数自增，退出自旋操作。

worker对象。

```
  Worker(Runnable firstTask) {
            setState(-1); // inhibit interrupts until runWorker
            this.firstTask = firstTask;
            this.thread = getThreadFactory().newThread(this);
        }
```
在当前线程不为空的情况下，加一把可重入锁reentrantlock，在加锁后，再次检查线程池的状态runstate，防止在获取到锁之前线程池已经关闭了，线程池的状态为running或者状态为shutdown并且任务为空的情况下，才能继续往下执行任务，这是充分必要条件。如果当前线程已经开启了，直接抛出异常，这是绝不允许的。

```
private final HashSet workers = new HashSet();
```

addWorkerFailed方法

如果添加worker失败或者开启线程失败就要调用addWorkerFailed方法移除失败的worker。

```
   private void addWorkerFailed(Worker w) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            if (w != null)
                workers.remove(w);
            decrementWorkerCount();
            tryTerminate();
        } finally {
            mainLock.unlock();
        }
    }
```
首先还是获取全局锁mainlock，接着对workers集合中移除worker，workers的数量自减。

runWorker方法

这个方法是运行task的方法

```
 final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            while (task != null || (task = getTask()) != null) {
                w.lock();

                if ((runStateAtLeast(ctl.get(), STOP) ||
                     (Thread.interrupted() &&
                      runStateAtLeast(ctl.get(), STOP))) &&
                    !wt.isInterrupted())
                    wt.interrupt();
                try {
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
                        task.run();
                    } catch (RuntimeException x) {
                        thrown = x; throw x;
                    } catch (Error x) {
                        thrown = x; throw x;
                    } catch (Throwable x) {
                        thrown = x; throw new Error(x);
                    } finally {
                        afterExecute(task, thrown);
                    }
                } finally {
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            processWorkerExit(w, completedAbruptly);
        }
    }
```
首先还是获取当前线程，获取当前worker对象中的任务task，把当前线程的状态由-1设为0，表示可以获取锁执行任务，接下来就是一个while循环，当task不为空或者从gettask方法取出的任务不为空的时候，加锁，底层还是使用了AQS，保证了只有一个线程执行完毕其他线程才能执行。在执行任务之前，必须进行判断，线程池的状态如果>=STOP，必须中断当前线程，如果是running或者shutdown，当前线程不能被中断，防止线程池调用了shutdownnow方法必须中断所有的线程。

在处理任务之前，会执行beforeExecute方法， 在处理任务之后，执行afterExecute方法，这两个都是钩子方法，继承了ThreadPoolExecutor可以重写此方法，嵌入自定义的逻辑。一旦在任务运行的过程中，出现异常会直接抛出，所以在实际的业务中，应该使用try..catch，把这些日常加入到日志中。

任务执行完，就把task设为空，累加当前线程完成的任务数，unlock，继续从queue中取任务执行。

getTask方法。

```
  private Runnable getTask() {
        boolean timedOut = false; // Did the last poll() time out?
for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);
// Check if queue empty only if necessary.
            if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                decrementWorkerCount();
                return null;
            }
int wc = workerCountOf(c);
// Are workers subject to culling?
```

```
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }
```

try {
Runnable r = timed ?
workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
workQueue.take();
if (r != null)
return r;
timedOut = true;
} catch (InterruptedExceptio
	n retry) {
	timedOut = false;
	}
	}
	}

目的就是获得一个要运行的任务，判断比较多

shutdown方法

```
public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            advanceRunState(SHUTDOWN);
            interruptIdleWorkers();
            onShutdown(); // hook for ScheduledThreadPoolExecutor
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
    }
```

调用了shutdown，意味着不能在继续往queue中添加任务，也不能在接受新的任务。利用cas把当前线程池的状态设为shutdown，中断所有的空闲线程，onShutdown是一个钩子方法，是专门给ScheduledThreadPoolExecutor来实现的，再次调用tryTerminate方法来尝试中止线程池，直到queue中的任务全部处理完毕才能正常关闭。

阿里手册上为什么不推荐使用Excutor直接创建线程池?

2020年1月2日 17:56

总结

阿里巴巴P3C手册上明确不推荐使用 Executors，而是推荐使用 ThreadPoolExecutor？
四个原因　两个重点　两个非重点
Executors 可创建四种线程池：

1. Executors.newFixdThreadPool() 创建固定大小的线程池

   源码如下：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

   参数解读：

   corePoolSize ： 线程池中核心线程数的最大值。此处为 nThreads个。

   maximumPoolSize ： 线程池中能拥有最多线程数 。此处为 nThreads 个。

   LinkedBlockingQueue 用于缓存任务的阻塞队列 。 此处没有设置容量大小，默认是 Integer.MAX_VALUE，可以认为是无界的。

   使用 LinkedBlockingQueue来存储来不及处理的多余线程 ---> 无界队列-----> 如果同一时间并发量特别大，无界队列一直加入线程可能导致OOM

2. Executors.newSingleThreadPool() 创建一个单线程的线程池，保证线程的顺序执行 ；

   源码如下：

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
                            (new ThreadPoolExecutor(1, 1,
                                                    0L, TimeUnit.MILLISECONDS,
                                                     new LinkedBlockingQueue<Runnable>()));
}
```

   可能会出现同样OOM问题

问题一：

   newFixedThreadPool()、newSingleThreadExecutor() 底层代码 中 **LinkedBlockingQueue 没有设置容量大小**，默认是 Integer.MAX_VALUE， 可以认为是无界的。线程池中 多余的线程会被缓存到 LinkedBlockingQueue中，最终内存撑爆。

3. Executors.newCachedThreadPool() 缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量

   源码解读：

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

   参数解读：

   corePoolSize ： 线程池中核心线程数的最大值。此处为 0 个。

   maximumPoolSize ： 线程池中能拥有最多线程数 。此处为 Integer.MAX_VALUE 。可以认为是无限大 。

   优点： 很灵活，弹性的线程池线程管理，用多少线程给多大的线程池，不用后及时回收，用则新建；

   缺点： 从源码中可以看出，SynchronousQueue() 只能存一个队列，可以认为所有 放到 newCachedThreadPool() 中的线程，不会缓存到队列中，而是直接运行的， 由于最大线程数是 Integer.MAX_VALUE ，这个数量级可以认为是无限大了， 随着执行线程数量的增多 和 线程没有及时结束，最终会将内存撑爆。

4. Executors.newScheduledThreadPool()创建固定大小的线程，可以延迟或定时的执行任务

   源码解读：

```
public static ScheduledExecutorService newScheduledThreadPool(
        int corePoolSize, ThreadFactory threadFactory) {
    return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
}
// ScheduledThreadPoolExecutor 类的源码：
public ScheduledThreadPoolExecutor(int corePoolSize,
                                   ThreadFactory threadFactory) {
    super(corePoolSize, Integer.MAX_VALUE, 0, TimeUnit.NANOSECONDS,
          new DelayedWorkQueue(), threadFactory);
}
```

   优点： 创建一个固定大小线程池，可以定时或周期性的执行任务；

缺点：与 newCachedThreadPool() 相同。

问题二：

newCachedThreadPool()、newScheduledThreadPool() 的底层代码中的**最大线程数（maximumPoolSize）是 Integer.MAX_VALUE，可以认为是无限大**，如果线程池中，执行中的线程没有及时结束，并且不断地有线程加入并执行，最终会将内存撑爆。

问题三：拒绝策略不能自定义（这个不是重点）

它们统一缺点：不支持自定义拒绝策略

Executors 底层其实是使用的 ThreadPoolExecutor 的方式创建的，但是使用的是 ThreadPoolExecutor 的默认策略，即 AbortPolicy。

源码解读：

```
//默认策略
 private static final RejectedExecutionHandler defaultHandler =
        new AbortPolicy();
//构造函数
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
   this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
            Executors.defaultThreadFactory(), defaultHandler);
}
```

问题四：创建线程或线程池时请指定有意义的线程名称，方便出错时回溯（这个不是重点）