并发编程

JUC-线程池

FutureTask

1. 是一个Runnable， 它的run方法

```
public void run() {
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                     null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                result = c.call();   // 【1】执行了给入的任务
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex); // 【3】 异常时，设置Ex
            }
            if (ran)
                set(result);  // 【2】 设置结果
        }
    } finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
        runner = null;
        // state must be re-read after nulling runner to prevent
        // leaked interrupts
        int s = state;
        if (s >= INTERRUPTING)
            handlePossibleCancellationInterrupt(s);
    }
}
```

2. get方法的逻辑

```
/**
 * @throws CancellationException {@inheritDoc}
 */
public V get() throws InterruptedException, ExecutionException {
    int s = state;
    if (s <= COMPLETING)
        s = awaitDone(false, 0L); //等待
    return report(s);
```

```
}

    private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        WaitNode q = null;
        boolean queued = false;
        for (;;) {   //【1】 无限循环，来判断条件是否满足，以及中途被中断后、唤醒后，继续等待
            if (Thread.interrupted()) {
                removeWaiter(q);
                throw new InterruptedException();
            }

            int s = state;
            if (s > COMPLETING) {
                if (q != null)
                    q.thread = null;
                return s;
            }
            else if (s == COMPLETING) // cannot time out yet
                Thread.yield();
            else if (q == null)
                q = new WaitNode();
            else if (!queued)
                queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                     q.next = waiters, q);
            else if (timed) {
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {
                    removeWaiter(q);
                    return state;
                }
                LockSupport.parkNanos(this, nanos);
            }
            else
                LockSupport.park(this);   //让一个线程等待，【2】阻塞自己
        }
    }
```

3. cancel的逻辑

4.
```
/**
 * Attempts to cancel execution of this task.  This attempt will
 * fail if the task has already completed, has already been cancelled,
 * or could not be cancelled for some other reason. If successful,
 * and this task has not started when {@code cancel} is called,
 * this task should never run.  If the task has already started,
 * then the {@code mayInterruptIfRunning} parameter determines
 * whether the thread executing this task should be interrupted in
 * an attempt to stop the task.
 *
 * <p>After this method returns, subsequent calls to {@link #isDone} will
 * always return {@code true}.  Subsequent calls to {@link #isCancelled}
 * will always return {@code true} if this method returned {@code true}.
 */
```

```
    public boolean cancel(boolean mayInterruptIfRunning) {
        if (!(state == NEW &&
              UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
                  mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
            return false;
        try {    // in case call to interrupt throws exception
            if (mayInterruptIfRunning) {
                try {
                    Thread t = runner;
                    if (t != null)
                        t.interrupt();
                } finally { // final state
                    UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
                }
            }
        } finally {
            finishCompletion();
        }
        return true;
    }
```

ThreadPoolExecutor

submit

```
/**
 * @throws RejectedExecutionException {@inheritDoc}
 * @throws NullPointerException       {@inheritDoc}
 */
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    //【要点】封装成了RunnableFuture，再调用的execute()方法
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

execute(runnable);

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    /*
     * Proceed in 3 steps:
     *
     * 1. If fewer than corePoolSize threads are running, try to
     * start a new thread with the given command as its first
     * task.  The call to addWorker atomically checks runState and
     * workerCount, and so prevents false alarms that would add
     * threads when it shouldn't, by returning false.
     *
     * 2. If a task can be successfully queued, then we still need
     * to double-check whether we should have added a thread
     * (because existing ones died since last checking) or that
     * the pool shut down since entry into this method. So we
     * recheck state and if necessary roll back the enqueuing if
```

```
     * stopped, or start a new thread if there are none.
     *
     * 3. If we cannot queue task, then we try to add a new
     * thread.  If it fails, we know we are shut down or saturated
     * and so reject the task.
     */
    int c = ctl.get();
    //【1】当前工作线程数小于核心线程数，添加工作线程来执行该任务
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    //【2】池是否在工作，如果在工作，放入任务队列
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        //【2-1】放入队列后，再次检查池状态
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //【3】 任务队列满了，或者池不在运行状态，添加线程，如果已达到最大线程数
    else if (!addWorker(command, false))
        reject(command);
}
```

池状态和线程计数

```
/**
 * The main pool control state, ctl, is an atomic integer packing
 * two conceptual fields
 *   workerCount, indicating the effective number of threads
 *   runState,    indicating whether running, shutting down etc
 *【1】低29位用于线程计数，高3位用于状态表示
 * In order to pack them into one int, we limit workerCount to
 * (2^29)-1 (about 500 million) threads rather than (2^31)-1 (2
 * billion) otherwise representable. If this is ever an issue in
 * the future, the variable can be changed to be an AtomicLong,
 * and the shift/mask constants below adjusted. But until the need
 * arises, this code is a bit faster and simpler using an int.
 *
 * The workerCount is the number of workers that have been
 * permitted to start and not permitted to stop.  The value may be
 * transiently different from the actual number of live threads,
 * for example when a ThreadFactory fails to create a thread when
 * asked, and when exiting threads are still performing
 * bookkeeping before terminating. The user-visible pool size is
 * reported as the current size of the workers set.
 *
 * The runState provides the main lifecycle control, taking on values:
 *【2】池的5个状态
 *   RUNNING:  Accept new tasks and process queued tasks
 *   SHUTDOWN: Don't accept new tasks, but process queued tasks
 *   STOP:     Don't accept new tasks, don't process queued tasks,
 *             and interrupt in-progress tasks
 *   TIDYING:  All tasks have terminated, workerCount is zero,
```

```
 *              the thread transitioning to state TIDYING
 *              will run the terminated() hook method
 *   TERMINATED: terminated() has completed
 *
 * The numerical order among these values matters, to allow
 * ordered comparisons. The runState monotonically increases over
 * time, but need not hit each state. The transitions are:
 *【3】状态转换
 * RUNNING -> SHUTDOWN
 *    On invocation of shutdown(), perhaps implicitly in finalize()
 * (RUNNING or SHUTDOWN) -> STOP
 *    On invocation of shutdownNow()
 * SHUTDOWN -> TIDYING
 *    When both queue and pool are empty
 * STOP -> TIDYING
 *    When pool is empty
 * TIDYING -> TERMINATED
 *    When the terminated() hook method has completed
 *
 * Threads waiting in awaitTermination() will return when the
 * state reaches TERMINATED.
 *
 * Detecting the transition from SHUTDOWN to TIDYING is less
 * straightforward than you'd like because the queue may become
 * empty after non-empty and vice versa during SHUTDOWN state, but
 * we can only terminate if, after seeing that it is empty, we see
 * that workerCount is 0 (which sometimes entails a recheck -- see
 * below).
 */
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

扩展

ThreadFactory

```
/* 【用途】通过提供自己的ThreadFactory，可以控制线程的name, thread group, priority,
daemon status
<dd>New threads are created using a {@link ThreadFactory}.  If not
* otherwise specified, a {@link Executors#defaultThreadFactory} is
* used, that creates threads to all be in the same {@link
* ThreadGroup} and with the same {@code NORM_PRIORITY} priority and
* non-daemon status. By supplying a different ThreadFactory, you can
* alter the thread's name, thread group, priority, daemon status,
* etc. If a {@code ThreadFactory} fails to create a thread when asked
* by returning null from {@code newThread}, the executor will
* continue, but might not be able to execute any tasks. Threads
* should possess the "modifyThread" {@code RuntimePermission}. If
* worker threads or other threads using the pool do not possess this
* permission, service may be degraded: configuration changes may not
* take effect in a timely manner, and a shutdown pool may remain in a
* state in which termination is possible but not completed.</dd>
```

RejectedExecutorHandler

```
/* <dt>Rejected tasks</dt>
* 【目标】了解提供了哪些拒绝策略，可自定义拒绝策略
* <dd>New tasks submitted in method {@link #execute(Runnable)} will be
* <em>rejected</em> when the Executor has been shut down, and also when
* the Executor uses finite bounds for both maximum threads and work queue
* capacity, and is saturated.  In either case, the {@code execute} method
* invokes the {@link
* RejectedExecutionHandler#rejectedExecution(Runnable, ThreadPoolExecutor)}
* method of its {@link RejectedExecutionHandler}.  Four predefined handler
* policies are provided:
*
* <ol>
*
* <li> In the default {@link ThreadPoolExecutor.AbortPolicy}, the
* handler throws a runtime {@link RejectedExecutionException} upon
* rejection. </li>
*
* <li> In {@link ThreadPoolExecutor.CallerRunsPolicy}, the thread
* that invokes {@code execute} itself runs the task. This provides a
* simple feedback control mechanism that will slow down the rate that
* new tasks are submitted. </li>
*
* <li> In {@link ThreadPoolExecutor.DiscardPolicy}, a task that
* cannot be executed is simply dropped.  </li>
*
* <li>In {@link ThreadPoolExecutor.DiscardOldestPolicy}, if the
* executor is not shut down, the task at the head of the work queue
* is dropped, and then execution is retried (which can fail again,
* causing this to be repeated.) </li>
*
* </ol>
*
* It is possible to define and use other kinds of {@link
* RejectedExecutionHandler} classes. Doing so requires some care
* especially when policies are designed to work only under particular
* capacity or queuing policies. </dd>
```

ThreadPoolExecutor 本身的扩展

```
* <dt>Hook methods</dt>
*【要点】了解钩子方法  beforeExecute  afterExecute  terminated
* <dd>This class provides {@code protected} overridable
* {@link #beforeExecute(Thread, Runnable)} and
* {@link #afterExecute(Runnable, Throwable)} methods that are called
* before and after execution of each task.  These can be used to
* manipulate the execution environment; for example, reinitializing
* ThreadLocals, gathering statistics, or adding log entries.
* Additionally, method {@link #terminated} can be overridden to perform
* any special processing that needs to be done once the Executor has
* fully terminated.
*
* <p>If hook or callback methods throw exceptions, internal worker
* threads may in turn fail and abruptly terminate.</dd>
```

并发协同

线程安全

1 锁

2 原子类

3 AQS

ReentrantLock 和 Semaphore 的异同

相同：

1. 都支持公平、非公平模式，都可以阻塞线程
2. 都是抢到资源的线程可以执行，抢不到的排队 int 10

不同点：

ReentrantLock 对象 ，只可以被一个线程独占，Semaphore 对象 可以多个线程拿到信号量（可被多个线程共享）

```
ReentrantLock  lock = new ReentrantLock ()

线程1：
lock.lock();
线程2：
lock.lock();

Semaphore semap = nwe Semaphore(10);
线程1：
semap.acquire();
线程2：
semap.acquire();
```

1 是什么

抽象排队同步器（大家都称它为：抽象队列同步器），JUC包中提供的一个基于FIFO等待队列来实现阻塞锁和线程同步器（如semaphores）的框架。对于大多数依赖于**单个原子int值**来表示状态的同步器，这个类被设计成一个非常有用的基础。

2 怎么使用它

1 AQS中定义的方法分为两大类：

- public的final方法，对外的行为
- protected 的空方法，由子类重写来更新内部的int状态值来实现对**这个对象**的acquire、release的含义。给定这些，这个类中的其他方法执行所有排队和阻塞机制。

2 子类应该定义为非public的内部助手类，用于实现其封闭类的同步属性。具体的锁和相关的同步器可以根据需要调用AQS的方法来实现它们的公共方法。

3、此类支持默认独占模式和共享模式中的一个或两个。在独占模式下获取时，其他线程尝试的获取无法成功。多线程的共享模式获取可能（但不一定）成功。在不同模式下等待的线程共享相同的FIFO队列。通常，子类只需实现支持其中一种模式，但也可实现使用两种模式，如ReadWriteLock。只支持独占或共享模式的子类不需要定义支持未使用模式的方法。

4、AQS中定义了一个内嵌的ConditionObject类，可被子类用作Condition的实现。举例ReentrantLock的newCondition()方法。

5、此类对象的序列化只会保留表示状态的int值，所以反序列化回来，线程队列是空的。请了解这一点。

6、**子类实现重写如下protected 空方法的注意实现：**

```
tryAcquire
tryRelease
tryAcquireShared   int
tryReleaseShared
isHeldExclusively
```

1. 子类重写 protected 空方法时使用`getState`、`setState`和`compareAndSetState`方法操作表示同步状态的int值
2. 这些方法内部需保证是线程安全的，应是非阻塞的短操作
3. 注意tryAcquireShared方法的返回值是int（其他是boolean）：负数表示失败；0表示成功，不传播唤醒排队线程获取；大于0成功，并传播唤醒排队线程获取。

7、AQS虽是基于内部的FIFO队列来排队的，但**它并不自动强制FIFO。**如 独占模式的核心代码如下：

```
* Acquire:
 *     while (!tryAcquire(arg)) {
 *        <em>enqueue thread if it is not already queued</em>;
 *        <em>possibly block current thread</em>;
 *     }
 *
 * Release:
 *     if (tryRelease(arg))
 *        <em>unblock the first queued thread</em>;
```

如需公平性，需由子类在tryAcquire方法中来提供。

8、AQS继承了`AbstractOwnableSynchronizer`,它提供了跟踪独占同步器的持有线程的方法，可以使用它来监控和诊断工具中帮助用户确定持有锁的线程。

9、它里面还提供一些get方法，可以获取排队线程等的信息，自行了解它们

10、AQS提供了一个高效的和可扩展的同步基础，它的使用范围是：可以依赖int state、acquire和release参数以及内部FIFO等待队列的同步器。当这还不够时，您可以使用原子类、您自己的自定义java.util.Queue类和LockSupport阻塞支持从较低级别构建同步器。

3 如何支持带超时等待

tryAcquire(long time)

节点状态

```
// CANCELLED：由于超时或中断，此节点被取消。节点一旦被取消了就不会再改变状态。特别是，取消节点的线程不会再阻塞。
static final int CANCELLED =  1;
// SIGNAL:此节点后面的节点已（或即将）被阻止（通过park），因此当前节点在释放或取消时必须断开后面的节点
// 为了避免竞争，acquire方法时前面的节点必须是SIGNAL状态，然后重试原子acquire，然后在失败时阻塞。
static final int SIGNAL    = -1;
// 此节点当前在条件队列中。标记为CONDITION的节点会被移动到一个特殊的条件等待队列（此时状态将设置为0），直到条件时才会被重新移动到同步等待队列 。（此处使用此值与字段的其他用途无关，但简化了机制。）
static final int CONDITION = -2;
//传播：应将releaseShared传播到其他节点。这是在doReleaseShared中设置的（仅适用于头部节点），以确保传播继续，即使此后有其他操作介入。
static final int PROPAGATE = -3;

//0:以上数值均未按数字排列以简化使用。非负值表示节点不需要发出信号。所以，大多数代码不需要检查特定的值，只需要检查符号。
//对于正常同步节点，该字段初始化为0；对于条件节点，该字段初始化为条件。它是使用CAS修改的
```

4 Synchronized原理

1 了解synchronized对应的字节码

```
javap -verbose SynchroinzedDemo.class
```

说明：

1. 方法上加synchronized，是在方法修饰符上加 ACC_SYNCHRONIZED
2. synchronized(obj)，对应的字节码是 monitorenter monitorexit

```
         0: aload_0
         1: dup
         2: astore_2
         3: monitorenter
         4: getstatic     #2                  // Field c:I
         7: iload_1
         8: iadd
         9: putstatic     #2                  // Field c:I
        12: aload_2
        13: monitorexit
        14: goto          22
        17: astore_3
```

问题： 何为监视器锁?

利用监视器来实现的锁

那何为监视器?

2 Monitor说明

请思考：synchronized的效果和ReentrantLock是一样的，那么synchronized实现同步的原理是否应该是一样的?

1. 要有互斥量
2. 有重入计数
3. 有阻塞排队队列
4. 有wait等待队列
5. JVM在实现时可以使用什么做互斥量?

   jvm也是程序，它直接面对操作系统，它可以使用操作系统提供的线程同步原语：mutex 互斥量和 semaphore 信号量；

   也可以基于CAS操作来充当互斥量。
6. 而它使用的是monitor，monitor是什么?

   > 管程，英文是 Monitor，也常被翻译为“监视器”，monitor 不管是翻译为“管程”还是“监视器”，都是比较晦涩难懂。它到底是什么？
   > 在使用基本的 mutex 进行并发控制时，需要程序员非常小心地控制 mutex 的 down 和 up 操作，否则很容易引起死锁等问题。为了更容易地编写出正确的并发程序，所以在 mutex 和 semaphore 的基础上，提出了更高层次的同步原语 monitor，不过需要注意的是，操作系统本身并不支持 monitor 机制，实际上，monitor 是属于编程语言的范畴，当你想要使用 monitor 时，先了解一下语言本身是否支持 monitor 原语，例如 C 语言它就不支持 monitor，Java 语言支持 monitor。
   > 一般的 monitor 实现模式是编程语言在语法上提供语法糖，而如何实现 monitor 机制，则属于编译器的工作，Java 就是这么干的。

   每个对象都有一个monitor，jvm中是如何实现对象monitor的

   ```
   openjdk\hotspot\src\share\vm\runtime\objectMonitor.hpp
   openjdk\hotspot\src\share\vm\runtime\objectMonitor.cpp
   ```

   hpp类上的注释：

```
//
// The ObjectMonitor class is used to implement JavaMonitors which have
// transformed from the lightweight structure of the thread stack to a
// heavy weight lock due to contention
```
ObjectMonitor类用于实现javamonitor，javamonitor是由于线程争用而从线程堆栈的轻量级结构转换为的重量级锁
```
// It is also used as RawMonitor by the JVMTI


class ObjectMonitor {
```

看结构体，了解其构成

```
  //【1】从这里可看到semaphore
// initialize the monitor, exception the semaphore, all other fields
  // are simple integers or pointers
  ObjectMonitor() {
    _header       = NULL;
    _count        = 0;
    _waiters      = 0,
    _recursions   = 0;       //线程的重入次数
    _object       = NULL;
    _owner        = NULL;    //标识拥有该monitor的线程
    _WaitSet      = NULL;    //等待线程组成的双向循环链表，_WaitSet是第一个节点
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;    //多线程竞争锁进入时的单向链表
    FreeNext      = NULL ;
    _EntryList    = NULL ;    //_owner从该双向循环链表中唤醒线程结点，_EntryList是第一个节点
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
    _previous_owner_tid = 0;
  }
```

看下方法

```
  bool      try_enter (TRAPS) ;
  void      enter(TRAPS);
  void      exit(bool not_suspended, TRAPS);
  void      wait(jlong millis, bool interruptable, TRAPS);
  void      notify(TRAPS);
  void      notifyAll(TRAPS);
```

3 锁的优化

1 自旋锁与自适应自旋

排队、阻塞、唤醒是比较耗时的。如果我们的使用场景中都是简单的++，set操作需同步，这样的操作很快就可以完成，**同时又有多个处理器**，则抢不到锁的线程完全可以通过忙循环（**自旋**）的方式在那里"稍等一下"，这样程序的性能会更好。这成为**自旋优化（自旋锁）**。jdk1.4.2引入，默认关闭，jdk1.6改为默认开启，开关参数：

```
-XX:+UseSpinning
```

优缺点说明：

对处理器数量有要求，自旋不是阻塞，需占用处理器时间。如果锁被占用的时间很短，自旋等待的效果会非常好，反之，自旋的线程会白白耗费处理器资源，反而会带来性能上的浪费。因此自旋等待的时间必须要有一定的限度。**自旋次数的默认值是10次，可通过参数`-XX:PreBlockSpin`来更改。**

```
//抢不到锁
排队
阻塞
唤醒
//
//抢不到锁
int i = 0;
while(i++ < 10);
//尝试抢锁
//抢不到锁
排队
阻塞
唤醒
//
```

jdk1.6中引入了自适应的自旋，自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的持有者的状态来决定。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更长的时间，比如100个循环。另外，如果自旋很少成功获得锁，那在以后要去抢这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。有了自适应自旋，随着程序运行和性能监控信息的不断完善，虚拟机对锁的状况预测就会越来越准确。

2 锁消除

通过逃逸分析发现其实根本就没有别的线程产生竞争的可能（别的线程没有临界量的引用），而"自作多情"地给自己加上了锁。有可能虚拟机会直接去掉这个锁。

3 锁粗化

通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽可能短（同步块的作用范围限制得尽量小——只在共享数据的实际作用域中才进行同步。）。

大部分情况下，上面的原则都是正确的。但是如果一系列的连续操作都对同一个对象反复加锁和解锁，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带来不必要的性能损耗。所以JVM中会针对这样的情况进行**锁粗化优化。**

4 轻量级锁

jdk1.6中加入的新型锁机制，“轻量级”是相对于monitor使用操作系统互斥量实现而言的，因此monitor锁成为”重量级锁“，在ObjectMonitor的注释中我们也看到：轻量级转为重量级锁。也即轻量级不是用来替代重量级的，而是在没有多线程竞争的前提下，减少传统重量级锁使用操作系统互斥量产生的性能消耗。

出现同一时间争抢情况不严重。

它主要是优化没有锁争用情况下的性能。

思考：不使用操作系统互斥量，那怎么达成抢锁判断（互斥）？

答案：CAS。轻量级锁利用对象头中的mark word来做互斥判断。

Mark Word (64 bits)	锁状态
unused:25 | identity_hashcode:31 | unused:1 | age:4 | biased_lock:0 | lock:01	正常
thread:54 | epoch:2 | unused:1 | age:4 | biased_lock:1 | lock:01	偏向锁
ptr_to_lock_record:62 | lock:00	轻量级锁
ptr_to_heavyweight_monitor:62 | lock:10	重量级锁
| lock:11	GC标记

以上是Java对象处于5种不同状态时，Mark Word中64个位的表现形式，上面每一行代表对象处于某种状态时的样子。其中各部分的含义如下：

lock:2位的锁状态标记位，由于希望用尽可能少的二进制位表示尽可能多的信息，所以设置了lock标记。该标记的值不同，整个Mark Word表示的含义不同。biased_lock和lock一起，表达的锁状态含义如下：

biased_lock lock 状态

0 01 无锁

1 01 偏向锁

00	轻量级锁
10	重量级锁
11	GC标记

biased_lock：对象是否启用偏向锁标记，只占1个二进制位。为1时表示对象启用偏向锁，为0时表示对象没有偏向锁。lock和biased_lock共同表示对象处于什么锁状态。

age：4位的Java对象年龄。在GC中，如果对象在Survivor区复制一次，年龄增加1。当对象达到设定的阈值时，将会晋升到老年代。默认情况下，并行GC的年龄阈值为15，并发GC的年龄阈值为6。由于age只有4位，所以最大值为15，这就是-XX:MaxTenuringThreshold选项最大值为15的原因。

identity_hashcode：31位的对象标识hashCode，采用延迟加载技术。调用方法System.identityHashCode()计算，并会将结果写到该对象头中。当对象加锁后（偏向、轻量级、重量级），MarkWord的字节没有足够的空间保存hashCode，因此该值会移动到管程Monitor中。

thread：持有偏向锁的线程ID。

epoch：偏向锁的时间戳。

ptr_to_lock_record：轻量级锁状态下，指向栈中锁记录的指针。

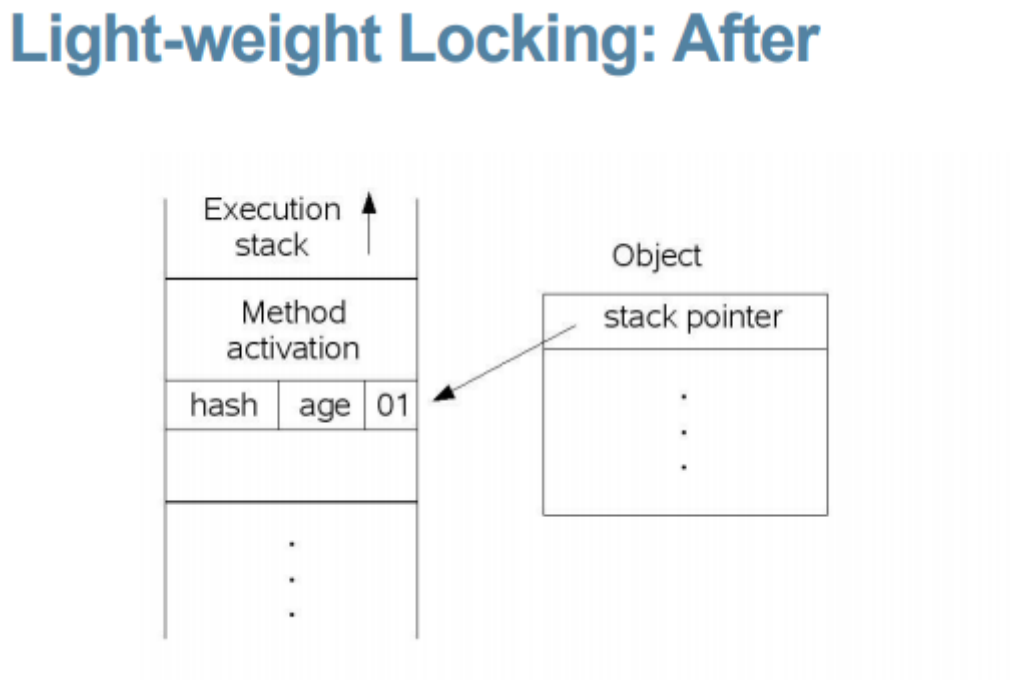
ptr_to_heavyweight_monitor：重量级锁状态下，指向对象监视器Monitor的指针。

轻量级锁的使用过程：

1. CAS修改mark word 的 lock标识为00，成功获得锁，失败则是有竞争，自旋，自旋获取不到，转为重量级锁。mark word 如何变化?

ptr_to_lock_record:62	lock:00	轻量级锁
ptr_to_heavyweight_monitor:62	lock:10	重量级锁

2. 抢到轻量级锁后将mark word 保存到执行栈上，释放时CAS还原到对象头上，能还原成功，意味着没线程争用，还原不成功，则表示有线程抢且阻塞等待了，唤醒等待线程，将mark word 复制给它。



轻量级锁能提升同步性能的依据是：”对于绝大部分的锁，在整个同步周期内都是不存在竞争的“，这是一个经验数据，如果满足，当然能带来性能提升。如果存在锁竞争，则可能会比重量级锁更慢。

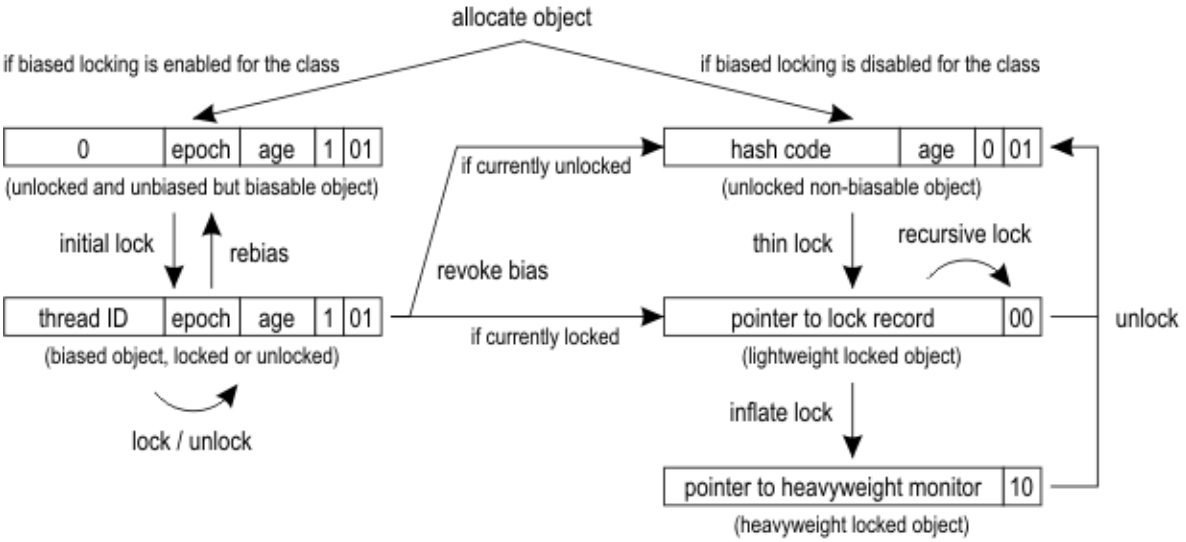
5 偏向锁

轻量级锁，毕竟还有抢锁、复制markword的过程，jdk1.6中更进一步引入了偏向锁来优化无争用时的性能。偏向即偏爱获得它的线程，无锁化执行。

偏向锁可以提高带有同步但无竞争的程序性能，它同样是带有效益权衡性质的优化。也就是说，它并不一定总是对程序运行有利，如果程序中大多数的锁总是被多个线程访问，那偏向模式就是多余的，此时可用如下参数禁用偏向锁优化，反而可以提升性能。

```
-XX:-UseBiasedLocking
```

6 锁的升级过程



5 ThreadLocal

是什么

```
/**
线程本地变量。   何为线程本地变量?
 * This class provides thread-local variables.  These variables differ from
 * their normal counterparts in that each thread that accesses one (via its
 * {@code get} or {@code set} method) has its own, independently initialized
 * copy of the variable.  {@code ThreadLocal} instances are typically private
 * static fields in classes that wish to associate state with a thread (e.g.,
 * a user ID or Transaction ID).
 *
```

变量的作用域?

- 局部变量 线程安全

```
class A {
    void doSome1(){
        int a = 10;
        int x = Thread.currentThread().get("x");

    }

    void doSome2() {
        int a = 11;
        int x = Thread.currentThread().getX();
    }

    void doSome3(){
    Thread.currentThread().set("x",10);
        doSome1();
        doSome2();
    }
}
```

- 全局变量 线程不安全的，需要加同步控制才能安全

```
class A {
    public static int Count = 1;
}
```

线程本地变量

线程的变量，在线程的执行过程中，随时可以去访问它

可以怎样去定义它?

```
class Thread {
    private Map<String,Object> datas;

    public Object get(String name){
        return datas.get(name);
    }

    public void set(String name,Object value){
        datas.put(name,value);
    }
}
```

```
static final ThreadLocal threadId  只有一个对象
    t1:
     threadid.get();
        t1.threadLocas   (t1的ThreadLocalMap)
            t1.threadLocas(threadId)
    t2:
     threadid.get();
        t2.threadLocas   (t2的ThreadLocalMap)
            t2.threadLocas(threadId)
```

线程本地变量是线程安全的吗?

当然

内存泄露问题:

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

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

//得到的信息:
Map 是以 ThreadLocal对象的 WeakReference 弱引用为key
```

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	当内存不足时	对象缓存	内存不足时终止
弱引用	正常垃圾回收时	对象缓存	垃圾回收后终止
虚引用	正常垃圾回收时	跟踪对象的垃圾回收	垃圾回收后终止

正确使用

- 使用 ThreadLocal 的时候，最好要声明为静态的；
- 使用完 ThreadLocal ， 一定手动调用 remove() 方法。否则可能导致：
    1. 内存被占用
    2. 内存泄露
    3. 线程是被复用的时（如线程池中的线程，web容器线程池中线程），可能会造成使用遗留的脏数据、影响业务逻辑。

  例如上面说到的 Session 的例子，如果不在拦截器或过滤器中处理，不仅可能出现内存泄漏问题，而且会影响业务逻辑；

正确的标准用法

```
private static final ThreadLocal<>  var1 = new ThreadLcoal<>();

try {
    threadLocal.set(a);
    //执行业务逻辑，逻辑中 get()值
}finally{
    //确保用完后，清除
    threadLocal.remove();
}
```

并发集合类

学习的要点：

1. 熟悉基本的API,满足使用
2. 明白他们是如何解决线程安全问题的
3. 性能是很好的，如何做到的

Queue

DelayQueue

延时队列

Map

HashMap 非线程安全

HashTable 线程安全