最近一直用的比较多的就是 java.util.concurrent(J.U.C)，实际上这块一直也没有完全深入研究，这次准备花点时间研究下 Java 里面整个并发体系。初步的设想包括比较大的方便（包括硬件、软件、思想以及误区等等），因此可能会持续较长的时间。这块内容也是 Java 在多线程方面引以为豪的一部分，深入这一部分不仅对整个 Java 体系有更深的了解，也对工作、学习的态度有多帮助。

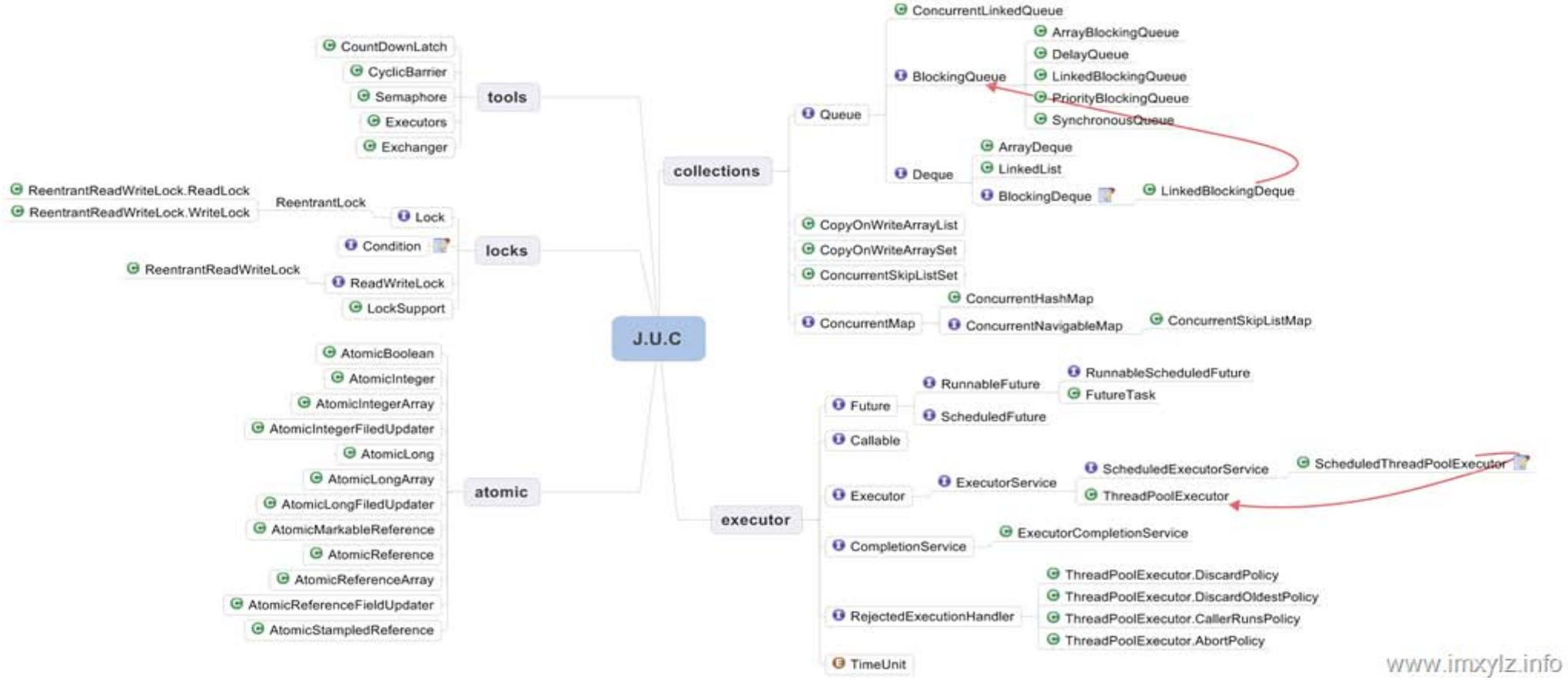
从深入浅出入手，大体内容包括一下几个方面：

（1）J.U.C 的 API：包括完整的类库结构和样例分析。

（2）J.U.C 的硬件原理以及软件思想：这部分也就将自己个人对硬件与程序语言的一些认识与大家分享，主要以总结前人的经验和所谓的理论来进行一些描述。

（3）J.U.C 的误区和常见陷阱：包括对 J.U.C 的一些设计思想和使用上的原则进行说明，同时对可能犯的错误进行一些总结。

下面的图是J.U.C完整的API。[完整的图片地址在这里]。



原子操作

深入浅出 Java Concurrency (2): 原子操作 part 1

从相对简单的 Atomic 入手（java.util.concurrent 是基于 Queue 的并发包，而 Queue，很多情况下使用到了 Atomic 操作，因此首先从这里开始）。很多情况下我们只是需要一个简单的、高效的、线程安全的递增递减方案。注意，这里有三个条件：简单，意味着程序员尽可能少的操作底层或者实现起来要比较容易；高效，意味着耗用资源要少，程序处理速度要快；线程安全也非常重要，这个在多线程下能保证数据的正确性。这三个条件看起来比较简单，但是实现起来却难以令人满意。

通常情况下，在 Java 里面，++i 或者--i 不是线程安全的，这里面有三个独立的操作：或者变量当前值，为该值+1/-1，然后写回新的值。在没有额外资源可以利用的情况下，只能使用加锁才能保证读-改-写这三个操作时“原子性”的。

Doug Lea在未将backport-util-concurrent合并到JSR 166里面来之前，是采用纯Java实现的，于是不可避免的采用了synchronized关键字。

```
public final synchronized void set(int newValue);
public final synchronized int getAndSet(int newValue);
public final synchronized int incrementAndGet();
```

同时在变量上使用了 volatile（后面会具体来讲 volatile 到底是个什么东东）来保证 get()的时候不用加锁。尽管 synchronized 的代价还是很高的，但是在没有 JNI 的手段下纯 Java 语言还是不能实现此操作的。

JSR 166 提上日程后，backport-util-concurrent 就合并到 JDK 5.0 里面了，在这里面重复使用了现代 CPU 的特性来降低锁的消耗。后本章的最后小结中会谈到这些原理和特性。在此之前先看看 API 的使用。

一切从 **java.util.concurrent.atomic.AtomicInteger** 开始。

int addAndGet(int delta)	以原子方式将给定值与当前值相加。 实际上就是等于线程安全版本的 i =i+delta 操作。
boolean compareAndSet(int expect, int update)	如果当前值 == 预期值，则以原子方式将该值设置为给定的更新值。 如果成功就返回 true，否则返回 false，并且不修改原值。
int decrementAndGet()	以原子方式将当前值减 1。 相当于线程安全版本的--i 操作。
int get()	获取当前值。
int getAndAdd(int delta)	以原子方式将给定值与当前值相加。 相当于线程安全版本的 t=i;i+=delta;return t;操作。
int getAndDecrement()	以原子方式将当前值减 1。 相当于线程安全版本的 i--操作。
int getAndIncrement()	以原子方式将当前值加 1。 相当于线程安全版本的 i++操作。
int getAndSet(int newValue)	以原子方式设置为给定值，并返回旧值。 相当于线程安全版本的 t=i;i=newValue;return t;操作。
int incrementAndGet()	以原子方式将当前值加 1。 相当于线程安全版本的++i 操作。
void lazySet(int newValue)	最后设置为给定值。 延时设置变量值，这个等价于 set()方法，但是由于字段是 volatile 类型的，因此次字段的修改会比普通字段（非 volatile 字段）有稍微的性能延时（尽管可以忽略），所以如果不是想立即读取设置的新值，允许在“后台”修改值，那么此方法就很有用。如果还是难以理解，这里就类似于启动一个后台线程如执行修改新值的任务，原线程就不等待修改结果立即返回（这种解释其实是不正确的，但是可以

	这么理解）。
void set(int newValue)	设置为给定值。 直接修改原始值，也就是 i=newValue 操作。
boolean weakCompareAndSet(int expect, int update)	如果当前值 == 预期值，则以原子方式将该设置为给定的更新值。JSR 规范中说：以原子方式读取和有条件地写入变量但不 创建任何 happen-before 排序，因此不提供与除 weakCompareAndSet 目标外任何变量以前或后续读取或写入操作有关的任何保证。大意就是说调用 weakCompareAndSet 时并不能保证不存在 happen-before 的发生（也就是可能存在指令重排序导致此操作失败）。但是从 Java 源码来看，其实此方法并没有实现 JSR 规范的要求，最后效果和 compareAndSet 是等效的，都调用了 unsafe.compareAndSwapInt()完成操作。

AtomicIntegerArray / AtomicLongArray / AtomicReferenceArray 的 API 类似，选择有代表性的 AtomicIntegerArray 来描述这些问题。

int get(int i) 获取位置 i 的当前值。很显然，由于这个是数组操作，就有索引越界的问题（IndexOutOfBoundsException 异常）。

对于下面的API起始和AtomicInteger是类似的，这种通过方法、参数的名称就能够得到函数意义的写法是非常值得称赞的。在《重构：改善既有代码的设计》和《代码整洁之道》中都非常推崇这种做法。

void set(int i, int newValue)

void lazySet(int i, int newValue)

int getAndSet(int i, int newValue)

boolean compareAndSet(int i, int expect, int update)

boolean weakCompareAndSet(int i, int expect, int update)

int getAndIncrement(int i)

int getAndDecrement(int i)

int getAndAdd(int i, int delta)

int incrementAndGet(int i)

int decrementAndGet(int i)

int addAndGet(int i, int delta)

整体来说，数组的原子操作在理解上还是相对比较容易的，这些 API 就是有多使用才能体会到它们的好处，而不仅仅是停留在理论阶段。

现在关注字段的原子更新。

AtomicIntegerFieldUpdater<T> / AtomicLongFieldUpdater<T> / AtomicReferenceFieldUpdater<T,V>是基于反射的原子更新字段的值。相应的 API 也是非常简单的，但是也是有一些约束的。

（1）字段必须是 volatile 类型的！在后面的章节中会详细说明为什么必须是 volatile，volatile 到底是个什么东西。

（2）字段的描述类型（修饰符 public/protected/default/private）是与调用者与操作对象字段的关系一致。也就是说调用者能够直接操作对象字段，那么就可以反射进行原子操作。但是对于父类的字段，子类是不能直接操作的，尽管子类可以访问父类的字段。

（3）只能是实例变量，不能是类变量，也就是说不能加 static 关键字。

（4）只能是可修改变量，不能使 final 变量，因为 final 的语义就是不可修改。实际上 final 的语义和 volatile 是有冲突的，这两个关键字不能同时存在。

（5）对于 **AtomicIntegerFieldUpdater** 和 **AtomicLongFieldUpdater** 只能修改 int/long 类型的字段，不能修改其包装类型（Integer/Long）。如果要修改包装类型就需要使用 **AtomicReferenceFieldUpdater**。

```
package xylz.study.concurrency.atomic;
import java.util.concurrent.atomic.AtomicIntegerFieldUpdater;
public class AtomicIntegerFieldUpdaterDemo {
    class DemoData{
        public volatile int value1 = 1;
        volatile int value2 = 2;
        protected volatile int value3 = 3;
        private volatile int value4 = 4;
    }
     AtomicIntegerFieldUpdater<DemoData> getUpdater(String fieldName) {
         return AtomicIntegerFieldUpdater.newUpdater(DemoData.class, fieldName);
     }
     void doit() {
         DemoData data = new DemoData();
         System.out.println("1 ==> "+getUpdater("value1").getAndSet(data, 10));
         System.out.println("3 ==> "+getUpdater("value2").incrementAndGet(data));
         System.out.println("2 ==> "+getUpdater("value3").decrementAndGet(data));
         System.out.println("true ==> "+getUpdater("value4").compareAndSet(data, 4, 5));
     }
     public static void main(String[] args) {
         AtomicIntegerFieldUpdaterDemo demo = new AtomicIntegerFieldUpdaterDemo();
         demo.doit();
     }
}
```

在上面的例子中 DemoData 的字段 value3/value4 对于 AtomicIntegerFieldUpdaterDemo 类是不可见的，因此通过反射是不能直接修改其值的。

AtomicMarkableReference 类描述的一个<Object,Boolean>的对，可以原子的修改 Object 或者 Boolean 的值，这种数据结构在一些缓存或者状态描述中比较有用。这种结构在单个或者同时修改 Object/Boolean 的时候能够有效的提高吞吐量。

AtomicStampedReference 类维护带有整数“标志”的对象引用，可以用原子方式对其进行更新。对比 **AtomicMarkableReference** 类的<Object,Boolean>，**AtomicStampedReference** 维护的是一种类似<Object,int>的数据结构，其实就是对对象（引用）的一个并发计数。但是与 **AtomicInteger** 不同的是，此数据结构可以携带一个对象引用（Object），并且能够对此对象和计数同时进行原子操作。

在后面的章节中会提到“ABA 问题”，而 **AtomicMarkableReference / AtomicStampedReference** 在解决“ABA 问题”上很有用。

原子操作的使用大概就是这么多，大体来说还算是比较清晰的，在下一个章节中，将对象原子操作进行总结，重点介绍下原子操作的原理和设计思想。

在这个小结里面重点讨论原子操作的原理和设计思想。

由于在下一个章节中会谈到锁机制，因此此小节中会适当引入锁的概念。

在Java Concurrency in Practice中是这样定义线程安全的：

当多个线程访问一个类时，如果不用考虑这些线程在运行时环境下的调度和交替运行，并且不需要额外的同步及在调用方代码不必做其他的协调，这个类的行为仍然是正确的，那么这个类就是线程安全的。

显然只有资源竞争时才会导致线程不安全，因此无状态对象永远是线程安全的。

原子操作的描述是： 多个线程执行一个操作时，其中任何一个线程要么完全执行完此操作，要么没有执行此操作的任何步骤，那么这个操作就是原子的。

枯燥的定义介绍完了，下面说更枯燥的理论知识。

指令重排序

Java语言规范规定了JVM线程内部维持顺序化语义，也就是说只要程序的最终结果等同于它在严格的顺序化环境下的结果，那么指令的执行顺序就可能与代码的顺序不一致。这个过程通过叫做指令的重排序。指令重排序存在的意义在于：**JVM能够根据处理器的特性（CPU的多级缓存系统、多核处理器等）适当的重新排序机器指令，使机器指令更符合CPU的执行特点，最大限度的发挥机器的性能。**

程序执行最简单的模型是按照指令出现的顺序执行，这样就与执行指令的 CPU 无关，最大限度的保证了指令的可移植性。这个模型的专业术语叫做顺序化一致性模型。但是现代计算机体系和处理器架构都不保证这一点（因为人为的指定并不能总是保证符合 CPU 处理的特性）。

我们来看最经典的一个案例。

```
package xylz.study.concurrency.atomic;
public class ReorderingDemo {
    static int x = 0, y = 0, a = 0, b = 0;
    public static void main(String[] args) throws Exception {
        for (int i = 0; i < 100; i++) {
            x=y=a=b=0;
            Thread one = new Thread() {
                public void run() {a = 1; x = b; }
            };
            Thread two = new Thread() {
                public void run() {b = 1; y = a; }
            };
            one.start();two.start();
            one.join();two.join();
            System.out.println(x + " " + y);
```

```
            }
        }
    }
```

在这个例子中 one/two 两个线程修改区 x,y,a,b 四个变量，在执行 100 次的情况下，可能得到(0 1)或者（1 0）或者（1 1）。事实上按照 JVM 的规范以及 CPU 的特性有很可能得到（0 0）。当然上面的代码大家不一定能得到（0 0），因为 run()里面的操作过于简单，可能比启动一个线程花费的时间还少，因此上面的例子难以出现（0,0）。但是在现代 CPU 和 JVM 上确实是存在的。由于 run()里面的动作对于结果是无关的，因此里面的指令可能发生指令重排序，即使是按照程序的顺序执行，数据变化刷新到主存也是需要时间的。假定是按照 a=1;x=b;b=1;y=a;执行的，x=0 是比较正常的，虽然 a=1 在 y=a 之前执行的，但是由于线程 one 执行 a=1 完成后还没有来得及将数据 1 写回主存（这时候数据是在线程 one 的堆栈里面的），线程 two 从主存中拿到的数据 a 可能仍然是 0（显然是一个过期数据，但是是有可能的），这样就发生了数据错误。

在两个线程交替执行的情况下数据的结果就不确定了，在机器压力大，多核 CPU 并发执行的情况下，数据的结果就更加不确定了。

Happens-before 法则

Java 存储模型有一个 happens-before 原则，就是如果动作 B 要看到动作 A 的执行结果（无论 A/B 是否在同一个线程里面执行），那么 A/B 就需要满足 happens-before 关系。

在介绍 happens-before 法则之前介绍一个概念：JMM 动作（Java Memeory Model Action），Java 存储模型动作。一个动作（Action）包括：变量的读写、监视器加锁和释放锁、线程的 start()和 join()。后面还会提到锁的的。

happens-before 完整规则：

（1）同一个线程中的每个 Action 都 happens-before 于出现在其后的任何一个 Action。

（2）对一个监视器的解锁 happens-before 于每一个后续对同一个监视器的加锁。

（3）对 volatile 字段的写入操作 happens-before 于每一个后续的同一个字段的读操作。

（4）Thread.start()的调用会 happens-before 于启动线程里面的动作。

（5）Thread 中的所有动作都 happens-before 于其他线程检查到此线程结束或者 Thread.join（）中返回或者 Thread.isAlive()==false。

（6）一个线程 A 调用另一个另一个线程 B 的 interrupt()都 happens-before 于线程 A 发现 B 被 A 中断（B 抛出异常或者 A 检测到 B 的 isInterrupted()或者 interrupted()）。

（7）一个对象构造函数的结束 happens-before 与该对象的 finalizer 的开始

（8）如果 A 动作 happens-before 于 B 动作，而 B 动作 happens-before 与 C 动作，那么 A 动作 happens-before 于 C 动作。

volatile 语义

到目前为止，我们多次提到 volatile，但是却仍然没有理解 volatile 的语义。

volatile 相当于 synchronized 的弱实现，也就是说 volatile 实现了类似 synchronized 的语义，却又没有锁机制。它确保对 volatile 字段的更新以可预见的方式告知其他的线程。

volatile 包含以下语义：

（1）Java 存储模型不会对 valatile 指令的操作进行重排序：这个保证对 volatile 变量的操作时按照指令的出现顺序执行的。

（2）volatile 变量不会被缓存在寄存器中（只有拥有线程可见）或者其他对 CPU 不可见的地方，每次总是从主存中读取 volatile 变量的结果。也就是说对于 volatile 变量的修改，其它线程总是可见的，并且不是使用自己线程栈内部的变量。也就是在 happens-before 法则中，对一个 valatile 变量的写操作后，其后的任何读操作理解可见此写操作的结果。

尽管 volatile 变量的特性不错，但是 volatile 并不能保证线程安全的，也就是说 volatile 字段的操作不是原子性的，volatile 变量只能保证可见性（一个线程修改后其它线程能够理解看到此变化后的结果），要想保证原子性，目前为止只能加锁！

volatile 通常在下面的场景：

```
volatile boolean done = false;
    while( ! done ){
        dosomething();
    }
```

应用 volatile 变量的三个原则：

（1）写入变量不依赖此变量的值，或者只有一个线程修改此变量

（2）变量的状态不需要与其它变量共同参与不变约束

（3）访问变量不需要加锁

这一节理论知识比较多，但是这是很面很多章节的基础，在后面的章节中会多次提到这些特性。

本小节中还是没有谈到原子操作的原理和思想，在下一节中将根据上面的一些知识来介绍原子操作。在 JDK 5 之前 Java 语言是靠 synchronized 关键字保证同步的，这会导致有锁（后面的章节还会谈到锁）。

锁机制存在以下问题：

（1）在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。

（2）一个线程持有锁会导致其它所有需要此锁的线程挂起。

（3）如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。

volatile 是不错的机制，但是 volatile 不能保证原子性。因此对于同步最终还是要回到锁机制上来。

独占锁是一种悲观锁，synchronized就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。

CAS 操作

上面的乐观锁用到的机制就是 CAS，Compare and Swap。

CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则什么都不做。

非阻塞算法 （nonblocking algorithms）

一个线程的失败或者挂起不应该影响其他线程的失败或挂起的算法。

现代的 CPU 提供了特殊的指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 compareAndSet() 就用这些代替了锁定。

拿出 AtomicInteger 来研究在没有锁的情况下是如何做到数据正确性的。

private volatile int value;

首先毫无以为，在没有锁的机制下可能需要借助 volatile 原语，保证线程间的数据是可见的（共享的）。

这样才获取变量的值的时候才能直接读取。

```
public final int get() {
        return value;
    }
```

然后来看看++i 是怎么做到的。

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next)) return next;
    }
}
```

在这里采用了 CAS 操作，每次从内存中读取数据然后将此数据和+1 后的结果进行 CAS 操作，如果成功就返回结果，否则重试直到成功为止。

而 compareAndSet 利用 JNI 来完成 CPU 指令的操作。

```
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
```

整体的过程就是这样子的，利用 CPU 的 CAS 指令，同时借助 JNI 来完成 Java 的非阻塞算法。其它原子操作都是利用类似的特性完成的。

而整个 J.U.C 都是建立在 CAS 之上的，因此对于 synchronized 阻塞算法，J.U.C 在性能上有了很大的提升。参考资料的文章中介绍了如果利用 CAS 构建非阻塞计数器、队列等数据结构。

CAS 看起来很爽，但是会导致“ABA 问题”。

CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。如果链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。因此前面提到的原子操作 AtomicStampedReference/AtomicMarkableReference 就很有用了。这允许一对变化的元素进行原子操作。

锁机制

上面的章节主要谈谈原子操作，至于与原子操作一些相关的问题或者说陷阱就放到最后的总结篇来整体说明。从这一章开始花少量的篇幅谈谈锁机制。

上一个章节中谈到了锁机制，并且针对于原子操作谈了一些相关的概念和设计思想。接下来的文章中，尽可能的深入研究锁机制，并且理解里面的原理和实际应用场合。

尽管 synchronized 在语法上已经足够简单了，在 JDK 5 之前只能借助此实现，但是由于是独占锁，性能却不高，因此 JDK 5 以后就开始借助于 JNI 来完成更高级的锁实现。

JDK 5 中的锁是接口 **java.util.concurrent.locks.Lock**。另外 **java.util.concurrent.locks.ReadWriteLock** 提供了一对可供读写并发的锁。根据前面的规则，我们从 **java.util.concurrent.locks.Lock** 的 API 开始。

void lock();	获取锁。 如果锁不可用，出于线程调度目的，将禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态。
void lockInterruptibly() throws InterruptedException;	如果当前线程未被中断，则获取锁。 如果锁可用，则获取锁，并立即返回。 如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下两种情况之一以前，该线程将一直处于休眠状态： •锁由当前线程获得；或者 •其他某个线程中断当前线程，并且支持对锁获取的中断。 如果当前线程： •在进入此方法时已经设置了该线程的中断状态；或者 •在获取锁时被中断，并且支持对锁获取的中断， 则将抛出 InterruptedException，并清除当前线程的已中断状态。
Condition newCondition();	返回绑定到此 Lock 实例的新 Condition 实例。下一小节中会重点谈 Condition，此处不做过多的介绍。
boolean tryLock();	仅在调用时锁为空闲状态才获取该锁。 如果锁可用，则获取锁，并立即返回值 true。如果锁不可用，则此方法将立即返回值 false。 通常对于那些不是必须获取锁的操作可能有用。
boolean tryLock(long time, TimeUnit unit) throws InterruptedException;	如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁。 如果锁可用，则此方法将立即返回值 true。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下三种情况之一前，该线程将一直处于休眠状态： •锁由当前线程获得；或者 •其他某个线程中断当前线程，并且支持对锁获取的中断；或者 •已超过指定的等待时间 如果获得了锁，则返回值 true。 如果当前线程： •在进入此方法时已经设置了该线程的中断状态；或者 •在获取锁时被中断，并且支持对锁获取的中断， 则将抛出 InterruptedException，并会清除当前线程的已中断状态。 如果超过了指定的等待时间，则将返回值 false。如果 time 小于等于 0，该方法将完全不等待。

void unlock();	释放锁。对应于 lock()、tryLock()、tryLock(xx)、lockInterruptibly()等操作，如果成功的话应该对应着一个 unlock()，这样可以避免死锁或者资源浪费。

相对于比较空洞的 API，来看一个实际的例子。下面的代码实现了一个类似于 AtomicInteger 的操作。

```
package xylz.study.concurrency.lock;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
    public class AtomicIntegerWithLock {
        private int value;
        private Lock lock = new ReentrantLock();
        public AtomicIntegerWithLock() {
                super();
            }
        public AtomicIntegerWithLock(int value) {
                this.value = value;
            }
        public final int get() {
           lock.lock();
           try {
              return value;
           } finally {
              lock.unlock();
           }
        }
        public final void set(int newValue) {
           lock.lock();
           try {
              value = newValue;
           } finally { lock.unlock();}
        }
        public final int getAndSet(int newValue) {
           lock.lock();
           try {
```

```
    public final boolean compareAndSet(int expect, int update) {
           lock.lock();
           try {
              if (value == expect) {
                 value = update;  return true;
              } return false;
           } finally { lock.unlock();}
        }
        public final int getAndIncrement() {
           lock.lock();
           try {
              return value++;
           } finally { lock.unlock();}
        }
        public final int getAndDecrement() {
           lock.lock();
           try {
              return value--;
           } finally {lock.unlock();}
        }
        public final int incrementAndGet() {
           lock.lock();
           try {
              return ++value;
           } finally {lock.unlock();}
        }
        public final int decrementAndGet() {
           lock.lock();
           try {
```

```
            int ret = value;
            value = newValue;
            return ret;
        } finally {
            lock.unlock();
        }
    }
```

```
            return --value;
        } finally { lock.unlock();}
    }
    public String toString() {
        return Integer.toString(get());
    }
}
```

类 **AtomicIntegerWithLock** 是线程安全的，此结构中大量使用了 Lock 对象的 lock/unlock 方法对。同样可以看到的是对于自增和自减操作使用了++/--。之所以能够保证线程安全，是因为 Lock 对象的 lock()方法保证了只有一个线程能够只有此锁。需要说明的是对于任何一个 lock()方法，都需要一个 unlock()方法与之对于，通常情况下为了保证 unlock 方法总是能够得到执行，unlock 方法被置于 finally 块中。另外这里使用了 **java.util.concurrent.locks.ReentrantLock.ReentrantLock** 对象，下一个小节中会具体描述此类作为 Lock 的唯一实现是如何设计和实现的。

尽管 synchronized 实现 Lock 的相同语义，并且在语法上比 Lock 要简单多，但是前者却比后者的开销要大得多。做一个简单的测试。

```
public static void main(String[] args) throws Exception{
    final int max = 10;
    final int loopCount = 100000;
    long costTime = 0;
    for (int m = 0; m < max; m++) {
        long start1 = System.nanoTime();
        final AtomicIntegerWithLock value1 = new AtomicIntegerWithLock(0);
        Thread[] ts = new Thread[max];
        for(int i=0;i<max;i++) {
            ts[i] = new Thread() {
                public void run() {
                    for (int i = 0; i < loopCount; i++) {
                        value1.incrementAndGet();
                    }
                }
            };
        }
        for(Thread t:ts) {
            t.start();
        }
        for(Thread t:ts) {
```

```
            t.join();
        }
        long end1 = System.nanoTime();
        costTime += (end1-start1);
    }
    System.out.println("cost1: " + (costTime));
        System.out.println();
    costTime = 0;
      final Object lock = new Object();
    for (int m = 0; m < max; m++) {
        staticValue=0;
        long start1 = System.nanoTime();
        Thread[] ts = new Thread[max];
        for(int i=0;i<max;i++) {
            ts[i] = new Thread() {
                public void run() {
                    for (int i = 0; i < loopCount; i++) {
                        synchronized(lock) {
                            ++staticValue;
                        }
                    }
                }
            };
        }
        for(Thread t:ts) {
            t.start();
        }
        for(Thread t:ts) {
            t.join();
        }
        long end1 = System.nanoTime();
        costTime += (end1-start1);
    }
```

```
        //
        System.out.println("cost2: " + (costTime));
    }
```

static int staticValue = 0;

在这个例子中每次启动 10 个线程，每个线程计算 100000 次自增操作，重复测试 10 次，下面是某此测试的结果：

cost1: 624071136

cost2: 2057847833

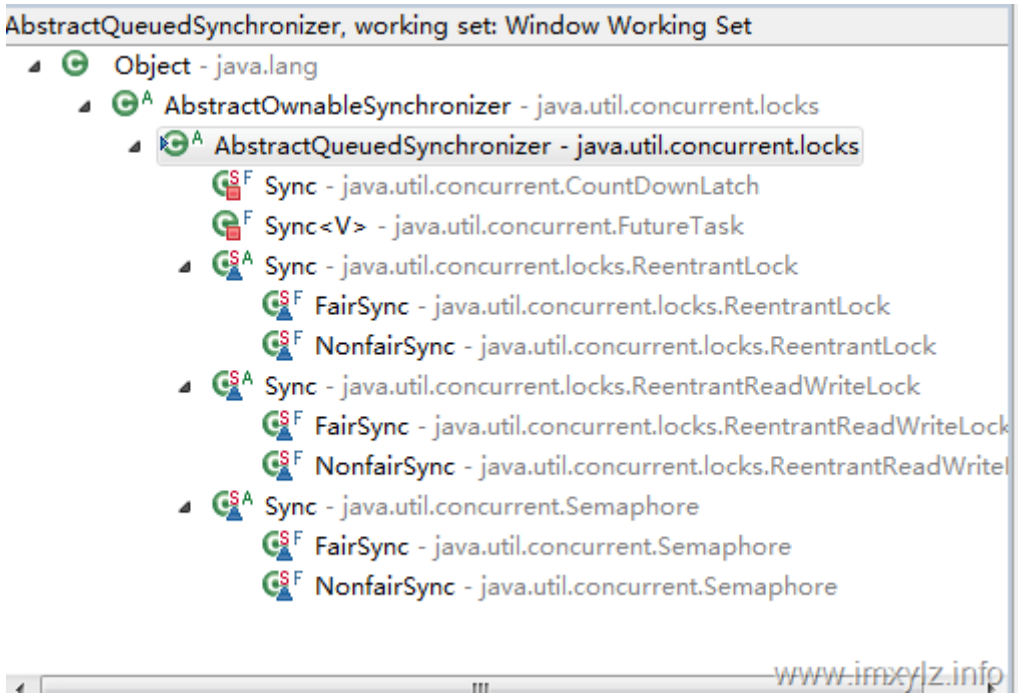
尽管上面的例子不是非常正式的测试案例，但上面的例子在于说明，Lock 的性能比 synchronized 的要好得多。如果可以的话总是使用 Lock 替代 synchronized 是一个明智的选择。

在理解 J.U.C 原理以及锁机制之前，我们来介绍 J.U.C 框架最核心也是最复杂的一个基础类：**java.util.concurrent.locks.AbstractQueuedSynchronizer**。

AQS

AbstractQueuedSynchronizer，简称 AQS，是 J.U.C 最复杂的一个类，导致绝大多数讲解并发原理或者实战的时候都不会提到此类。但是虚心的作者愿意借助自己有限的能力和精力来探讨一二（参考资源中也有一些作者做了部分的分析。）。

首先从理论知识开始，在了解了相关原理后会针对源码进行一些分析，最后加上一些实战来描述。



面的继承体系中，AbstractQueuedSynchronizer 是 CountDownLatch/FutureTask/ReentrantLock/RenntrantReadWriteLock/Semaphore 的基础，因此 AbstractQueuedSynchronizer 是 Lock/Executor 实现的前提。公平锁、不公平锁、Condition、CountDownLatch、Semaphore 等放到后面的篇幅中说明。

完整的设计原理可以参考 Doug Lea 的论文 The java.util.concurrent Synchronizer Framework ，这里做一些简要的分析。

基本的思想是表现为一个**同步器**，支持下面两个操作：

获取锁：首先判断当前状态是否允许获取锁，如果是就获取锁，否则就阻塞操作或者获取失败，也就是说如果是独占锁就可能阻塞，如果是共享锁就可能失败。另外如果是阻塞线程，那么线程就需要进入阻塞队列。当状态位允许获取锁时就修改状态，并且如果进了队列就从队列中移除。

```
while(synchronization state does not allow acquire){
    enqueue current thread if not already queued;
    possibly block current thread;
}
dequeue current thread if it was queued;
```

释放锁:这个过程就是修改状态位，如果有线程因为状态位阻塞的话就唤醒队列中的一个或者更多线程。

```
update synchronization state;
if(state may permit a blocked thread to acquire)
    unlock one or more queued threads;
```

要支持上面两个操作就必须有下面的条件：

- 原子性操作同步器的状态位
- 阻塞和唤醒线程
- 一个有序的队列

目标明确，要解决的问题也清晰了，那么剩下的就是解决上面三个问题。

状态位的原子操作

这里使用一个 32 位的整数来描述状态位，前面章节的原子操作的理论知识整好派上用场，在这里依然使用 CAS 操作来解决这个问题。事实上这里还有一个 64 位版本的同步器（AbstractQueuedLongSynchronizer），这里暂且不谈。

阻塞和唤醒线程

标准的 JAVA API 里面是无法挂起（阻塞）一个线程，然后在将来某个时刻再唤醒它的。JDK 1.0 的 API 里面有 Thread.suspend 和 Thread.resume，并且一直延续了下来。但是这些都是过时的 API，而且也是不推荐的做法。

在 JDK 5.0 以后利用 JNI 在 LockSupport 类中实现了此特性。

```
LockSupport.park()
LockSupport.park(Object)
LockSupport.parkNanos(Object, long)
LockSupport.parkNanos(long)
LockSupport.parkUntil(Object, long)
LockSupport.parkUntil(long)
```

LockSupport.unpark(Thread)

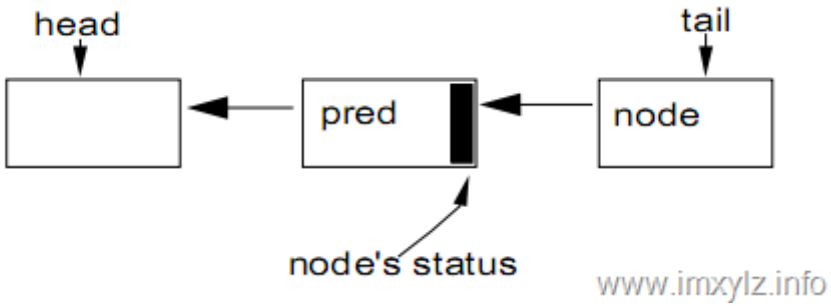
上面的 API 中 park()是在当前线程中调用，导致线程阻塞，带参数的 Object 是挂起的对象，这样监视的时候就能够知道此线程是因为什么资源而阻塞的。由于 park()立即返回，所以通常情况下需要在循环中去检测竞争资源来决定是否进行下一次阻塞。park()返回的原因有三：

- 其他某个线程调用将当前线程作为目标调用 unpark；
- 其他某个线程中断当前线程；
- 该调用不合逻辑地（即毫无理由地）返回。

其实第三条就决定了需要循环检测了，类似于通常写的 while(checkCondition()){Thread.sleep(time);}类似的功能。

有序队列

在 AQS 中采用 CHL 列表来解决有序的队列的问题。



AQS 采用的 CHL 模型采用下面的算法完成 FIFO 的入队列和出队列过程。

对于入队列(enqueue)：采用 CAS 操作，每次比较尾结点是否一致，然后插入的到尾结点中。

```
do {pred = tail;
}while ( !compareAndSet(pred,tail,node) );
```

对于出队列(dequeue):由于每一个节点也缓存了一个状态，决定是否出队列，因此当不满足条件时就需要自旋等待，一旦满足条件就将头结点设置为下一个节点。

```
while (pred.status != RELEASED) ;
head  = node;
```

实际上这里自旋等待也是使用 LockSupport.park()来实现的。

AQS 里面有三个核心字段：

```
private volatile int state;
private transient volatile Node head;
private transient volatile Node tail;
```

其中 state 描述的有多少个线程取得了锁，对于互斥锁来说 state<=1。head/tail 加上 CAS 操作就构成了一个 CHL 的 FIFO 队列。

下面是 Node 节点的属性：

volatile int waitStatus; 节点的等待状态，一个节点可能位于以下几种状态：

- CANCELLED = 1： 节点操作因为超时或者对应的线程被 interrupt。节点不应该不留在此状态，一旦达到此状态将从 CHL 队列中踢出。

- SIGNAL = -1： 节点的继任节点是（或者将要成为）BLOCKED 状态（例如通过 LockSupport.park()操作），因此一个节点一旦被释放（解锁）或者取消就需要唤醒（LockSupport.unpack()）它的继任节点。
- CONDITION = -2：表明节点对应的线程因为不满足一个条件（Condition）而被阻塞。
- 0： 正常状态，新生的非 CONDITION 节点都是此状态。
- 非负值标识节点不需要被通知（唤醒）。

volatile Node prev;此节点的前一个节点。节点的 waitStatus 依赖于前一个节点的状态。

volatile Node next;此节点的后一个节点。后一个节点是否被唤醒（uppark()）依赖于当前节点是否被释放。

volatile Thread thread;节点绑定的线程。

Node nextWaiter;下一个等待条件（Condition）的节点，由于 Condition 是独占模式，因此这里有一个简单的队列来描述 Condition 上的线程节点。

AQS 在 J.U.C 里面是一个非常核心的工具，而且也非常复杂，里面考虑到了非常多的逻辑实现，所以在后面的章节中总是不断的尝试介绍 AQS 的特性和实现。

这一个小节主要介绍了一些理论背景和相关的数据结构，在下一个小节中将根据以上知识来了解 Lock.lock/unlock 是如何实现的。

接上篇，这篇从 Lock.lock/unlock 开始。特别说明在没有特殊情况下所有程序、API、文档都是基于 JDK 6.0 的。

public void java.util.concurrent.locks.ReentrantLock.lock() 获取锁。

如果该锁没有被另一个线程保持，则获取该锁并立即返回，将锁的保持计数设置为 1。

如果当前线程已经保持该锁，则将保持计数加 1，并且该方法立即返回。

如果该锁被另一个线程保持，则出于线程调度的目的，禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态，此时锁保持计数被设置为 1。

从上面的文档可以看出 ReentrantLock 是**可重入锁**的实现。而内部是委托 java.util.concurrent.locks.ReentrantLock.Sync.lock()实现的。java.util.concurrent.locks.ReentrantLock.Sync 是抽象类，有 java.util.concurrent.locks.ReentrantLock.FairSync 和 java.util.concurrent.locks.ReentrantLock.NonfairSync 两个实现，也就是常说的公平锁和不公平锁。

公平锁和非公平锁

如果获取一个锁是按照请求的顺序得到的，那么就是公平锁，否则就是非公平锁。

在没有深入了解内部机制及实现之前，先了解下为什么会存在公平锁和非公平锁。公平锁保证一个阻塞的线程最终能够获得锁，因为是有序的，所以总是可以按照请求的顺序获得锁。不公平锁意味着后请求锁的线程可能在其前面排列的休眠线程恢复前拿到锁，这样就有可能提高并发的性能。这是因为通常情况下挂起的线程重新开始与它真正开始运行，二者之间会产生严重的延时。因此非公平锁就可以利用这段时间完成操作。这是非公平锁在某些时候比公平锁性能要好的原因之一。

二者在实现上的区别会在后面介绍，我们先从公平锁（FairSync）开始。

前面说过 **java.util.concurrent.locks.AbstractQueuedSynchronizer （AQS)**是 Lock 的基础，对于一个 FairSync 而言，lock()就直接调用 AQS 的 acquire(int arg)；

public final void acquire(int arg) 以**独占模式获取对象**，忽略中断。通过至少调用一次 tryAcquire(int) 来实现此方法，并在成功时返回。否则在成功之前，一直调用 tryAcquire(int) 将线程加入队列，线程可能重复被阻塞或不被阻塞。

在介绍实现之前先要补充上一节的知识，对于一个 AQS 的实现而言，通常情况下需要实现以下方法来描述如何锁定线程。

- tryAcquire(int) 试图在独占模式下获取对象状态。此方法应该查询是否允许它在独占模式下获取对象状态，如果允许，则获取它。

此方法总是由执行 acquire 的线程来调用。如果此方法报告失败，则 acquire 方法可以将线程加入队列（如果还没有将它加入队列），直到获得其他某个线程释放了该线程的信号。也就是说此方法是一种尝试性方法，如果成功获取锁那最好，如果没有成功也没有关系，直接返回 false。

- tryRelease(int) 试图设置状态来反映独占模式下的一个释放。 此方法总是由正在执行释放的线程调用。释放锁可能失败或者抛出异常，这个在后面会具体分析。
- tryAcquireShared(int) 试图在共享模式下获取对象状态。
- tryReleaseShared(int) 试图设置状态来反映共享模式下的一个释放。
- isHeldExclusively() 如果对于当前（正调用的）线程，同步是以独占方式进行的，则返回 true。

除了 tryAcquire(int)外，其它方法会在后面具体介绍。首先对于 ReentrantLock 而言，不管是公平锁还是非公平锁，都是独占锁，也就是说同时能够有一个线程持有锁。因此对于 acquire(int arg)而言，arg==1。在 AQS 中 acquire 的实现如下：

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

这个看起来比较复杂，我们分解以下 4 个步骤。

1. 如果 tryAcquire(arg)成功，那就没有问题，已经拿到锁，整个 lock()过程就结束了。如果失败进行操作 2。
2. 创建一个独占节点（Node）并且此节点加入 CHL 队列末尾。进行操作 3。
3. 自旋尝试获取锁，失败根据前一个节点来决定是否挂起（park()），直到成功获取到锁。进行操作 4。
4. 如果当前线程已经中断过，那么就中断当前线程（清除中断位）。

这是一个比较复杂的过程，我们按部就班一个一个分析。

tryAcquire(acquires)

对于公平锁而言，它的实现方式如下：

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (isFirst(current) &&compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
```

```
            }
            return false;
        }
    }
```

在这段代码中，前面说明对于 AQS 存在一个 state 来描述当前有多少线程持有锁。由于 AQS 支持共享锁（例如读写锁，后面会继续讲），所以这里 state>=0，但是由于 ReentrantLock 是独占锁，所以这里不妨理解为 0<=state，acquires=1。isFirst(current)是一个很复杂的逻辑，包括踢出无用的节点等复杂过程，这里暂且不提，大体上的意思是说判断 AQS 是否为空或者当前线程是否在队列头（为了区分公平与非公平锁）。

1. 如果当前锁有其它线程持有，c!=0，进行操作 2。否则，如果当前线程在 AQS 队列头部，则尝试将 AQS 状态 state 设为 acquires（等于 1），成功后将 AQS 独占线程设为当前线程返回 true，否则进行 2。这里可以看到 compareAndSetState 就是使用了 CAS 操作。
2. 判断当前线程与 AQS 的独占线程是否相同，如果相同，那么就将当前状态位加 1（这里+1 后结果为负数后面会讲，这里暂且不理它），修改状态位，返回 true，否则进行 3。这里之所以不是将当前状态位设置为 1，而是修改为旧值+1 呢？这是因为 ReentrantLock 是可重入锁，同一个线程每持有一次就+1。
3. 返回 false。

比较非公平锁的 tryAcquire 实现 java.util.concurrent.locks.ReentrantLock.Sync.nonfairTryAcquire(int)，公平锁多了一个判断当前节点是否在队列头，这个就保证了是否按照请求锁的顺序来决定获取锁的顺序（同一个线程的多次获取锁除外）。

现在再回头看公平锁和非公平锁的 lock()方法。公平锁只有一句 acquire(1)；而非公平锁的调用如下：

```
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```

很显然，非公平锁在第一次获取锁，或者其它线程释放锁后（可能等待），优先采用 compareAndSetState(0,1)然后设置 AQS 独占线程而持有锁，这样有时候比 acquire(1)顺序检查锁持有而要高效。即使在重入锁上，也就是 compareAndSetState(0,1)失败，但是是当前线程持有锁上，非公平锁也没有问题。

addWaiter(mode)

tryAcquire 失败就意味着入队列了。此时 AQS 的队列中节点 Node 就开始发挥作用了。一般情况下 AQS 支持独占锁和共享锁，而独占锁在 Node 中就意味着条件（Condition）队列为空（上一篇中介绍过相关概念）。在 java.util.concurrent.locks.AbstractQueuedSynchronizer.Node 中有两个常量，

```
static final Node EXCLUSIVE = null; //独占节点模式
static final Node SHARED = new Node(); //共享节点模式
```

addWaiter(mode)中的 mode 就是节点模式，也就是共享锁还是独占锁模式。

前面一再强调 **ReentrantLock 是独占锁模式**。

```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
```

```
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node; return node;
        }
    }
    enq(node);
    return node;
}
```

上面是节点如队列的一部分。当前仅当队列不为空并且将新节点插入尾部成功后直接返回新节点。否则进入 enq(Node)进行操作。

```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            Node h = new Node(); // Dummy header
            h.next = node;
            node.prev = h;
            if (compareAndSetHead(h)) {
                tail = node; return h;
            }
        }
        else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;  return t;
            }
        }
    }
}
```

enq(Node)去队列操作实现了 CHL 队列的算法，如果为空就创建头结点，然后同时比较节点尾部是否是改变来决定 CAS 操作是否成功，当且仅当成功后才将为不节点的下一个节点指向为新节点。可以看到这里仍然是 CAS 操作。

acquireQueued(node,arg)

自旋请求锁，如果可能的话挂起线程，直到得到锁，返回当前线程是否中断过（如果 park()过并且中断过的话有一个 interrupted 中断位）。

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

下面的分析就需要用到上节节点的状态描述了。acquireQueued 过程是这样的：

1. 如果当前节点是 AQS 队列的头结点（如果第一个节点是 DUMP 节点也就是傀儡节点，那么第二个节点实际上就是头结点了），就尝试在此获取锁 tryAcquire(arg)。如果成功就将头结点设置为当前节点（不管第一个结点是否是 DUMP 节点），返回中断位。否则进行 2。

2. 检测当前节点是否应该 park()，如果应该 park()就挂起当前线程并且返回当前线程中断位。进行操作 1。

一个节点是否该 park()是关键，这是由方法 java.util.concurrent.locks.AbstractQueuedSynchronizer.shouldParkAfterFailedAcquire(Node, Node)实现的。

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int s = pred.waitStatus;
    if (s < 0) return true;
    if (s > 0) {
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else compareAndSetWaitStatus(pred, 0, Node.SIGNAL);
```

```
    return false;
}
```

1. 如果前一个节点的等待状态 waitStatus<0，也就是前面的节点还没有获得到锁，那么返回 true，表示当前节点（线程）就应该 park()了。否则进行 2。

2. 如果前一个节点的等待状态 waitStatus>0，也就是前一个节点被 CANCELLED 了，那么就将前一个节点去掉，递归此操作直到所有前一个节点的 waitStatus<=0，进行 4。否则进行 3。

3. 前一个节点等待状态 waitStatus=0，修改前一个节点状态位为 SINGAL，表示后面有节点等待你处理，需要根据它的等待状态来决定是否该 park()。进行 4。

4. 返回 false，表示线程不应该 park()。

selfInterrupt()

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

如果线程曾经中断过（或者阻塞过）（比如手动 interrupt()或者超时等等，那么就再中断一次，中断两次的意思就是清除中断位）。

大体上整个 Lock.lock()就这样一个流程。除了 lock()方法外，还有 lockInterruptibly()/tryLock()/unlock()/newCondition()等，在接下来的章节中会一一介绍。

本小节介绍锁释放 Lock.unlock()。

Release / TryRelease

unlock 操作实际上就调用了 **AQS** 的 release 操作，释放持有的锁。

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0) unparkSuccessor(h);
        return true;
    }
    return false;
}
```

前面提到过 **tryRelease(arg)**操作，此操作里面总是尝试去释放锁，如果成功，说明锁确实被当前线程持有，那么就看 **AQS** 队列中的头结点是否为空并且能否被唤醒，如果可以的话就唤醒继任节点（下一个非 CANCELLED 节点，下面会具体分析）。

对于独占锁而言，java.util.concurrent.locks.ReentrantLock.Sync.tryRelease(int)展示了如何尝试释放锁(**tryRelease**)操作。

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread()) throw new IllegalMonitorStateException();
    boolean free = false;
```

```
    if (c == 0) {
        free = true; setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

整个 **tryRelease** 操作是这样的：

1. 判断持有锁的线程是否是当前线程，如果不是就抛出 IllegalMonitorStateExeception()，因为一个线程是不能释放另一个线程持有的锁（否则锁就失去了意义）。否则进行 2。

2. 将 AQS 状态位减少要释放的次数（对于独占锁而言总是 1），如果剩余的状态位 0（也就是没有线程持有锁），那么当前线程就是最后一个持有锁的线程，清空 AQS 持有锁的独占线程。进行 3。

3. 将剩余的状态位写回 AQS，如果没有线程持有锁就返回 true，否则就是 false。

参考上一节的分析就可以知道，这里 c==0 决定了是否完全释放了锁。由于 **ReentrantLock** 是可重入锁，因此同一个线程可能多重持有锁，那么当且仅当最后一个持有锁的线程释放锁是才能将 AQS 中持有锁的独占线程清空，这样接下来的操作才需要唤醒下一个需要锁的 **AQS** 节点（Node），否则就只是减少锁持有的计数器，并不能改变其他操作。

当 **tryRelease** 操作成功后（也就是完全释放了锁），release 操作才能检查是否需要唤醒下一个继任节点。这里的前提是 **AQS** 队列的头结点需要锁(waitStatus!=0)，如果头结点需要锁，就开始检测下一个继任节点是否需要锁操作。

在上一节中说道 **acquireQueued** 操作完成后（拿到了锁），会将当前持有锁的节点设为头结点，所以一旦头结点释放锁，那么就需要寻找头结点的下一个需要锁的继任节点，并唤醒它。

```
private void unparkSuccessor(Node node) {
    //此时 node 是需要是需要释放锁的头结点
    //清空头结点的 waitStatus，也就是不再需要锁了
    compareAndSetWaitStatus(node, Node.SIGNAL, 0)  //从头结点的下一个节点开始寻找继任节点，当且仅当继任节点的 waitStatus<=0 才是有效继任节点，否则将这些 waitStatus>0（也就是 CANCELLED 的节点）从 AQS 队列中剔除
    Node s = node.next;
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0) s = t;
    }
    //如果找到一个有效的继任节点，就唤醒此节点线程
    if (s != null) LockSupport.unpark(s.thread);
}
```

这里再一次把 **acquireQueued** 的过程找出来。对比 **unparkSuccessor**，一旦头节点的继任节点被唤醒，那么继任节点就会尝试去获取锁（在 **acquireQueued** 中 node 就是有效的继任节点，p 就是唤醒它的头结点），如果成功就会将头结点设置为自身，并且将头结点的前任节点清空，这样前任节点（已经过时了）就可以被 GC 释放了。

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

在 **setHead** 中，将头结点的前任节点清空并且将头结点的线程清空就是为了更好的 GC，防止内存泄露。

```
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```

对比 lock()操作，unlock()操作还是比较简单的，主要就是释放响应的资源，并且唤醒 **AQS** 队列中有效的继任节点。这样所就按照请求的顺序去尝试获取锁了。

整个 lock()/unlock()过程完成了，我们再回头看公平锁(FairSync)和非公平锁(NonfairSync)。

公平锁和非公平锁只是在获取锁的时候有差别，其它都是一样的。

```
final void lock() {
    if (compareAndSetState(0, 1))  setExclusiveOwnerThread(Thread.currentThread());
    else acquire(1);
}
```

在上面非公平锁的代码中总是优先尝试当前是否有线程持有锁，一旦没有任何线程持有锁，那么非公平锁就霸道的尝试将锁“占为己有”。如果在抢占锁的时候失败就和公平锁一样老老实实的去排队。

也即是说公平锁和非公平锁只是在入 **AQS** 的 **CLH** 队列之前有所差别，一旦进入了队列，所有线程都是按照队列中先来后到的顺序请求锁。

Condition

条件变量很大一个程度上是为了解决 Object.wait/notify/notifyAll 难以使用的问题。

条件（也称为条件队列 或条件变量）为线程提供了一个含义，以便在某个状态条件现在可能为 true 的另一个线程通知它之前，一直挂起该线程（即让其“等待”）。因为访问此共享状态信息发生在不同的线程中，所以它必须受保护，因此要将某种形式的锁与该条件相关联。等待提供一个条件的主要属性是：以原子方式 释放相关的锁，并挂起当前线程，就像 Object.wait 做的那样。

上述 API 说明表明条件变量需要与锁绑定，而且多个 Condition 需要绑定到同一锁上。前面的 **Lock** 中提到，获取一个条件变量的方法是 **Lock.newCondition()**。

```
void await() throws InterruptedException;
void awaitUninterruptibly();
long awaitNanos(long nanosTimeout) throws InterruptedException;
boolean await(long time, TimeUnit unit) throws InterruptedException;
boolean awaitUntil(Date deadline) throws InterruptedException;
void signal();
void signalAll();
```

以上是 **Condition** 接口定义的方法，await*对应于 Object.wait，signal 对应于 Object.notify，signalAll 对应于 Object.notifyAll。特别说明的是 **Condition** 的接口改变名称就是为了避免与 Object 中的 wait/notify/notifyAll 的语义和使用上混淆，因为 Condition 同样有 wait/notify/notifyAll 方法。

每一个 **Lock** 可以有任意数据的 **Condition** 对象，**Condition** 是与 **Lock** 绑定的，所以就有 **Lock** 的公平性特性：如果是公平锁，线程为按照 FIFO 的顺序从 Condition.await 中释放，如果是非公平锁，那么后续的锁竞争就不保证 FIFO 顺序了。

一个使用 Condition 实现生产者消费者的模型例子如下。

```
package xylz.study.concurrency.lock;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ProductQueue<T> {
    private final T[] items;
    private final Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();
```

```
//
private int head, tail, count;
public ProductQueue(int maxSize) {   items = (T[]) new Object[maxSize]; }
public ProductQueue() {
   this(10);
}
public void put(T t) throws InterruptedException {
   lock.lock();
   try {
      while (count == getCapacity()) { notFull.await(); }
      items[tail] = t;
      if (++tail == getCapacity()) {
         tail = 0;
      }
      ++count;
      notEmpty.signalAll();
   } finally {
      lock.unlock();
   }
}
public T take() throws InterruptedException {
   lock.lock();
   try {
      while (count == 0) {   notEmpty.await();    }
      T ret = items[head];
      items[head] = null;//GC
      //
      if (++head == getCapacity()) {
         head = 0;
      }
      --count;
      notFull.signalAll();
      return ret;
```

```
        } finally {
            lock.unlock();
        }
    }
    public int getCapacity() {
        return items.length;
    }
    public int size() {
        lock.lock();
        try {return count;
        } finally {lock.unlock();
        }
    }
}
```

在这个例子中消费 take()需要 队列不为空，如果为空就挂起（await()），直到收 到 notEmpty 的信号；生产 put()需要队列不满，如果满了就挂起（await()），直到收到 notFull 的信号。

可能有人会问题，如果一个线程 lock()对象后被挂起还没有 unlock，那么另外一个线程就拿不到锁了（lock()操作会挂起），那么就无法通知(notify)前一个线程，这样岂不是“死锁”了？

await* 操作

上一节中说过多次 ReentrantLock 是独占锁，一个线程拿到锁后如果不释放，那么另外一个线程肯定是拿不到锁，所以在 lock.lock()和 lock.unlock()之间可能有一次释放锁的操作（同样也必然还有一次获取锁的操作）。我们再回头看代码，不管 take()还是 put()，在进入 lock.lock()后唯一可能释放锁的操作就是 await()了。也就是说 await()操作实际上就是释放锁，然后挂起线程，一旦条件满足就被唤醒，再次获取锁！

```
public final void await() throws InterruptedException {
    if (Thread.interrupted()) throw new InterruptedException();
    Node node = addConditionWaiter();
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0) break;
    }
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)  interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) unlinkCancelledWaiters();
```

```
        if (interruptMode != 0)  reportInterruptAfterWait(interruptMode);
    }
```

上面是 await()的代码片段。上一节中说过，**AQS** 在获取锁的时候需要有一个 **CHL** 的 FIFO 队列，所以对于一个 Condition.await()而言，如果释放了锁，要想再一次获取锁那么就需要进入队列，等待被通知获取锁。完整的 await()操作是安装如下步骤进行的：

1. 将当前线程加入 Condition 锁队列。特别说明的是，这里不同于 **AQS** 的队列，这里进入的是 Condition 的 FIFO 队列。后面会具体谈到此结构。进行 2。
2. 释放锁。这里可以看到将锁释放了，否则别的线程就无法拿到锁而发生死锁。进行 3。
3. 自旋(while)挂起，直到被唤醒或者超时或者 CACELLED 等。进行 4。
4. 获取锁(acquireQueued)。并将自己从 Condition 的 FIFO 队列中释放，表明自己不再需要锁（我已经拿到锁了）。

这里再回头介绍 Condition 的数据结构。我们知道一个 Condition 可以在多个地方被 await*()，那么就需要一个 FIFO 的结构将这些 Condition 串联起来，然后根据需要唤醒一个或者多个（通常是所有）。所以在 Condition 内部就需要一个 FIFO 的队列。

```
    private transient Node firstWaiter;
    private transient Node lastWaiter;
```

上面的两个节点就是描述一个FIFO的队列。我们再结合前面提到的节点（Node）数据结构。我们就发现Node.nextWaiter就派上用场了！ nextWaiter就是将一系列的Condition.await*串联起来组成一个FIFO的队列。

signal/signalAll 操作

await*()清楚了，现在再来看 signal/signalAll 就容易多了。按照 signal/signalAll 的需求，就是要将 Condition.await*()中 FIFO 队列中第一个 **Node** 唤醒（或者全部 **Node**）唤醒。尽管所有 **Node** 可能都被唤醒，但是要知道的是仍然只有一个线程能够拿到锁，其它没有拿到锁的线程仍然需要自旋等待，就上上面提到的第 4 步(acquireQueued)。

```
    private void doSignal(Node first) {
        do {
            if ( (firstWaiter = first.nextWaiter) == null) lastWaiter = null;
            first.nextWaiter = null;
        } while (!transferForSignal(first) &&  (first = firstWaiter) != null);
    }
    private void doSignalAll(Node first) {
        lastWaiter = firstWaiter  = null;
        do {
            Node next = first.nextWaiter;
            first.nextWaiter = null;
            transferForSignal(first);
            first = next;
        } while (first != null);
    }
```

上面的代码很容易看出来，signal 就是唤醒**Condition**队列中的第一个非 CANCELLED 节点线程，而 signalAll 就是唤醒所有非 CANCELLED 节点线程。当然了遇到 CANCELLED 线程就需要将其从 FIFO 队列中剔除。

```
final boolean transferForSignal(Node node) {
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))   return false;
    Node p = enq(node);
    int c = p.waitStatus;
    if (c > 0 || !compareAndSetWaitStatus(p, c, Node.SIGNAL))   LockSupport.unpark(node.thread);
    return true;
}
```

上面就是唤醒一个 await*()线程的过程，根据前面的小节介绍的，如果要 unpark 线程，并使线程拿到锁，那么就需要线程节点进入 **AQS** 的队列。所以可以看到在 LockSupport.unpark 之前调用了 enq(node)操作，将当前节点加入到 **AQS** 队列。

整个锁机制的原理就介绍完了，从下一节开始就进入了锁机制的应用了。

此小节介绍几个与锁有关的有用工具。

闭锁（Latch）

闭锁（Latch）：一种同步方法，可以延迟线程的进度直到线程到达某个终点状态。通俗的讲就是，一个闭锁相当于一扇大门，在大门打开之前所有线程都被阻断，一旦大门打开所有线程都将通过，但是一旦大门打开，所有线程都通过了，那么这个闭锁的状态就失效了，门的状态也就不能变了，只能是打开状态。也就是说闭锁的状态是一次性的，它确保在闭锁打开之前所有特定的活动都需要在闭锁打开之后才能完成。

CountDownLatch 是 JDK 5+里面闭锁的一个实现，允许一个或者多个线程等待某个事件的发生。**CountDownLatch** 有一个正数计数器，countDown 方法对计数器做减操作，await 方法等待计数器达到 0。所有 await 的线程都会阻塞直到计数器为 0 或者等待线程中断或者超时。

CountDownLatch 的 API 如下。

- public void await() throws InterruptedException
- public boolean await(long timeout, TimeUnit unit) throws InterruptedException
- public void countDown()
- public long getCount()

其中 getCount()描述的是当前计数，通常用于调试目的。

下面的例子中描述了闭锁的两种常见的用法。

```
package xylz.study.concurrency.lock;
import java.util.concurrent.CountDownLatch;
public class PerformanceTestTool {
    public long timecost(final int times, final Runnable task) throws InterruptedException {
```

```
        if (times <= 0) throw new IllegalArgumentException();
        final CountDownLatch startLatch = new CountDownLatch(1);
        final CountDownLatch overLatch = new CountDownLatch(times);
        for (int i = 0; i < times; i++) {
            new Thread(new Runnable() {
                public void run() {
                    try {
                        startLatch.await();
                        task.run();
                    } catch (InterruptedException ex) {
                        Thread.currentThread().interrupt();
                    } finally {
                        overLatch.countDown();
                    }
                }
            }).start();
        }
        //
        long start = System.nanoTime();
        startLatch.countDown();
        overLatch.await();
        return System.nanoTime() - start;
    }
}
```

在上面的例子中使用了两个闭锁，第一个闭锁确保在所有线程开始执行任务前，所有准备工作都已经完成，一旦准备工作完成了就调用 startLatch.countDown()打开闭锁，所有线程开始执行。第二个闭锁在于确保所有任务执行完成后主线程才能继续进行，这样保证了主线程等待所有任务线程执行完成后才能得到需要的结果。在第二个闭锁当中，初始化了一个 N 次的计数器，每个任务执行完成后都会将计数器减一，所有任务完成后计数器就变为了 0，这样主线程闭锁 overLatch 拿到此信号后就可以继续往下执行了。

根据前面的happend-before法则可以知道闭锁有以下特性：

内存一致性效果：线程中调用 countDown() 之前的操作 happen-before 紧跟在从另一个线程中对应 await() 成功返回的操作。

在上面的例子中第二个闭锁相当于把一个任务拆分成 N 份，每一份独立完成任务，主线程等待所有任务完成后才能继续执行。这个特性在后面的线程池框架中会用到，其实 **FutureTask** 就可以看成一个闭锁。后面的章节还会具体分析 **FutureTask** 的。

同样基于探索精神，仍然需要“窥探”下 **CountDownLatch** 里面到底是如何实现 await*和 countDown 的。

首先，研究下 await()方法。内部直接调用了 **AQS** 的 acquireSharedInterruptibly(1)。

```
public final void acquireSharedInterruptibly(int arg) throws InterruptedException {
    if (Thread.interrupted()) throw new InterruptedException();
    if (tryAcquireShared(arg) < 0) doAcquireSharedInterruptibly(arg);
}
```

前面一直提到的都是独占锁（排它锁、互斥锁），现在就用到了另外一种锁，**共享锁**。

所谓共享锁是说**所有共享锁的线程共享同一个资源，一旦任意一个线程拿到共享资源，那么所有线程就都拥有的同一份资源。**也就是通常情况下共享锁只是一个标志，所有线程都等待这个标识是否满足，一旦满足所有线程都被激活（相当于所有线程都拿到锁一样）。这里的闭锁**CountDownLatch**就是基于共享锁的实现。

闭锁中关于 **AQS** 的 tryAcquireShared 的实现是如下代码（**java.util.concurrent.CountDownLatch.Sync.tryAcquireShared**）：

```
public int tryAcquireShared(int acquires) {
    return getState() == 0? 1 : -1;
}
```

在这份逻辑中，对于闭锁而言第一次 await 时 tryAcquireShared 应该总是-1，因为对于闭锁 **CountDownLatch** 而言 state 的值就是初始化的 count 值。这也就解释了为什么在 countDown 调用之前闭锁的 count 总是>0。

```
private void doAcquireSharedInterruptibly(int arg)  throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt()) break;
        }
    } catch (RuntimeException ex) {
        cancelAcquire(node);
        throw ex;
    }
```

```
        // Arrive here only if interrupted
        cancelAcquire(node);
        throw new InterruptedException();
    }
```

上面的逻辑展示了如何通过 await 将所有线程串联并挂起，直到被唤醒或者条件满足或者被中断。整个过程是这样的：

1. 将当前线程节点以共享模式加入**AQS**的**CLH**队列中（相关概念参考这里和这里）。进行 2。
2. 检查当前节点的前任节点，如果是头结点并且当前闭锁计数为 0 就将当前节点设置为头结点，唤醒继任节点，返回（结束线程阻塞）。否则进行 3。
3. 检查线程是否该阻塞，如果应该就阻塞(park)，直到被唤醒（unpark）。重复 2。
4. 如果 2、3 有异常就抛出异常（结束线程阻塞）。

这里有一点值得说明下，设置头结点并唤醒继任节点 setHeadAndPropagate。由于前面 tryAcquireShared 总是返回 1 或者-1，而进入 setHeadAndPropagate 时总是 propagate>=0，所以这里 propagate==1。后面唤醒继任节点操作就非常熟悉了。

```
    private void setHeadAndPropagate(Node node, int propagate) {
        setHead(node);
        if (propagate > 0 && node.waitStatus != 0) {
            Node s = node.next;
            if (s == null || s.isShared())
                unparkSuccessor(node);
        }
    }
```

从上面的所有逻辑可以看出 countDown 应该就是在条件满足（计数为 0）时唤醒头结点（时间最长的一个节点），然后头结点就会根据 FIFO 队列唤醒整个节点列表（如果有的话）。

从 **CountDownLatch** 的 countDown 代码中看到，直接调用的是 **AQS** 的 releaseShared(1)，参考前面的知识，这就印证了上面的说法。

tryReleaseShared 中正是采用 CAS 操作减少计数（每次减-1）。

```
    public boolean tryReleaseShared(int releases) {
        for (;;) {
            int c = getState();
            if (c == 0)  return false;
            int nextc = c-1;
            if (compareAndSetState(c, nextc)) return nextc == 0;
        }
    }
```

整个 **CountDownLatch** 就是这个样子的。其实有了前面原子操作和 **AQS** 的原理及实现，分析 **CountDownLatch** 还是比较容易的。

如果说CountDownLatch是一次性的，那么**CyclicBarrier**正好可以循环使用。它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。所谓屏障点就是一组任务执行完毕的时刻。

清单 1 一个使用 CyclicBarrier 的例子

```
package xylz.study.concurrency.lock;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierDemo {
    final CyclicBarrier barrier;
    final int MAX_TASK;
    public CyclicBarrierDemo(int cnt) {
        barrier = new CyclicBarrier(cnt + 1);
        MAX_TASK = cnt;
    }
    public void doWork(final Runnable work) {
        new Thread() {
            public void run() {
                work.run();
                try {
                    int index = barrier.await();
                    doWithIndex(index);
                } catch (InterruptedException e) {
                    return;
                } catch (BrokenBarrierException e) {
                    return;
                }
            }
        }.start();
    }
    private void doWithIndex(int index) {
        if (index == MAX_TASK / 3) { System.out.println("Left 30%.");
        } else if (index == MAX_TASK / 2) {   System.out.println("Left 50%");
        } else if (index == 0) {   System.out.println("run over");
        }
```

```
    }
    public void waitForNext() {
        try {
            doWithIndex(barrier.await());
        } catch (InterruptedException e) {
            return;
        } catch (BrokenBarrierException e) {
            return;
        }
    }
    public static void main(String[] args) {
        final int count = 10;
        CyclicBarrierDemo demo = new CyclicBarrierDemo(count);
        for (int i = 0; i < 100; i++) {
            demo.doWork(new Runnable() {
                public void run() {
                    //do something
                    try {
                        Thread.sleep(1000L);
                    } catch (Exception e) {
                        return;
                    }
                }
            });
            if ((i + 1) % count == 0) {
                demo.waitForNext();
            }
        }
    }
}
```

清单 1 描述的是一个周期性处理任务的例子，在这个例子中有一对的任务（100 个），希望每 10 个为一组进行处理，当前仅当上一组任务处理完成后才能进行下一组，另外在每一组任务中，当任务剩下 50%，30%以及所有任务执行完成时向观察者发出通知。

在这个例子中，CyclicBarrierDemo 构建了一个 count+1 的任务组（其中一个任务时为了外界方便挂起主线程）。每一个子任务里，人物本身执行完毕后都需要等待同组内其它任务执行完成后才能继续。同时在剩下任务 50%、30%已经 0 时执行特殊的其他任务（发通知）。

很显然 CyclicBarrier 有以下几个特点：

- await()方法将挂起线程，直到同组的其它线程执行完毕才能继续
- await()方法返回线程执行完毕的索引，注意，索引时从任务数-1 开始的，也就是第一个执行完成的任务索引为 parties-1,最后一个为 0，这个 parties 为总任务数，清单中是 cnt+1
- CyclicBarrier 是可循环的，显然名称说明了这点。在清单 1 中，每一组任务执行完毕就能够执行下一组任务。

另外除了 CyclicBarrier 除了以上特点外，还有以下几个特点：

- 如果屏障操作不依赖于挂起的线程，那么任何线程都可以执行屏障操作。在清单 1 中可以看到并没有指定那个线程执行 50%、30%、0%的操作，而是一组线程（cnt+1）个中任何一个线程只要到达了屏障点都可以执行相应的操作
- CyclicBarrier 的构造函数允许携带一个任务，这个任务将在 0%屏障点执行，它将在 await()==0 后执行。
- CyclicBarrier 如果在 await 时因为中断、失败、超时等原因提前离开了屏障点，那么任务组中的其他任务将立即被中断，以 InterruptedException 异常离开线程。
- 所有 await()之前的操作都将在屏障点之前运行，也就是 CyclicBarrier 的内存一致性效果

CyclicBarrier 的所有 API 如下：

- public CyclicBarrier(int parties) 创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。
- public CyclicBarrier(int parties, Runnable barrierAction) 创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动 barrier 时执行给定的屏障操作，该操作由最后一个进入 barrier 的线程执行。
- public int await() throws InterruptedException, BrokenBarrierException 在所有参与者都已经在此 barrier 上调用 await 方法之前，将一直等待。
- public int await(long timeout,TimeUnit unit) throws InterruptedException, BrokenBarrierException,TimeoutException 在所有参与者都已经在此屏障上调用 await 方法之前将一直等待,或者超出了指定的等待时间。
- public int getNumberWaiting() 返回当前在屏障处等待的参与者数目。此方法主要用于调试和断言。
- public int getParties() 返回要求启动此 barrier 的参与者数目。
- public boolean isBroken() 查询此屏障是否处于损坏状态。
- public void reset() 将屏障重置为其初始状态。

针对以上 API，下面来探讨下 CyclicBarrier 的实现原理，以及为什么有这样的 API。

清单 2 CyclicBarrier.await*()的实现片段

```
private int dowait(boolean timed, long nanos) throws InterruptedException, BrokenBarrierException, TimeoutException {
final ReentrantLock lock = this.lock;
lock.lock();
try {
    final Generation g = generation;
    if (g.broken) throw new BrokenBarrierException();
```

```
 if (Thread.interrupted()) {
     breakBarrier(); throw new InterruptedException();
 }
int index = --count;
if (index == 0) {   // tripped
    boolean ranAction = false;
    try {
        final Runnable command = barrierCommand;
        if (command != null)   command.run();
        ranAction = true;
        nextGeneration();
        return 0;
    } finally {
        if (!ranAction) breakBarrier();
    }
}
 // loop until tripped, broken, interrupted, or timed out
 for (;;) {
     try {
         if (!timed) trip.await();
         else if (nanos > 0L) nanos = trip.awaitNanos(nanos);
     } catch (InterruptedException ie) {
         if (g == generation && ! g.broken) {
             breakBarrier(); throw ie;
         } else {
             Thread.currentThread().interrupt();
         }
     }
     if (g.broken) throw new BrokenBarrierException();
     if (g != generation) return index;
     if (timed && nanos <= 0L) {
         breakBarrier(); throw new TimeoutException();
     }
```

```
            }
        } finally {
            lock.unlock();
        }
    }
```

清单 2 有点复杂，这里一点一点的剖析，并且还原到最原始的状态。

利用前面学到的知识，我们知道要想让线程等待其他线程执行完毕，那么已经执行完毕的线程（进入 await*()方法）就需要 park()，直到超时或者被中断，或者被其它线程唤醒。

前面说过 CyclicBarrier 的特点是要么大家都正常执行完毕，要么大家都异常被中断，不会其中有一个被中断而其它正常执行完毕的现象存在。这种特点叫 all-or-none。类似的概念是原子操作中的要么大家都执行完，要么一个操作都不执行完。当前这其实是两个概念了。要完成这样的特点就必须有一个状态来描述曾经是否有过线程被中断（broken)了，这样后面执行完的线程就该知道是否需要继续等待了。而在 CyclicBarrier 中 Generation 就是为了完成这件事情的。Generation 的定义非常简单，整个结构就只有一个变量 boolean broken = false;，定义是否发生了 broken 操作。

由于有竞争资源的存在（broken/index），所以毫无疑问需要一把锁 lock。拿到锁后整个过程是这样的：

1. 检查是否存在中断位(broken)，如果存在就立即以 BrokenBarrierException 异常返回。此异常描述的是线程进入屏障被破坏的等待状态。否则进行 2。
2. 检查当前线程是否被中断，如果是那么就设置中断位（使其它将要进入等待的线程知道），另外唤醒已经等待的线程，同时以 InterruptedException 异常返回，表示线程要处理中断。否则进行 3。
3. 将剩余任务数减 1，如果此时剩下的任务数为 0，也就是达到了公共屏障点，那么就执行屏障点任务（如果有的话），同时创建新的 Generation（在这个过程中会唤醒其它所有线程，因此当前线程是屏障点线程，那么其它线程就都应该在等待状态）。否则进行 4。
4. 到这里说明还没有到达屏障点，那么此时线程就应该 park()。很显然在下面的 for 循环中就是要 park 线程。这里 park 线程采用的是 Condition.await()方法。也就是 trip.await*()。为什么需要 Condition？因为所有的 await*()其实等待的都是一个条件，一旦条件满足就应该都被唤醒，所以 Condition 整好满足这个特点。所以到这里就会明白为什么在步骤 3 中到达屏障点时创建新的 Generation 的时候是一定要唤醒其它线程的原因了。

上面 4 个步骤其实只是描述主体结构，事实上整个过程中有非常多的逻辑来处理异常引发的问题，比如执行屏障点任务引发的异常，park 线程超时引发的中断异常和超时异常等等。所以对于 await()而言，异常的处理比业务逻辑的处理更复杂，这就解释了为什么 await()的时候可能引发 InterruptedException,BrokenBarrierException,TimeoutException 三种异常。

清单 3 生成下一个循环周期并唤醒其它线程

```
private void nextGeneration() {
    trip.signalAll();
    count = parties;
    generation = new Generation();
}
```

清单 3 描述了如何生成下一个循环周期的过程，在这个过程中当然需要使用 Condition.signalAll()唤醒所有已经执行完成并且正在等待的线程。另外这里 count 描述的是还有多少线程需要执行，是为了线程执行完毕索引计数。

isBroken() 方法描述的就是 generation.broken，也即线程组是否发生了异常。这里再一次解释下为什么要有这个状态的存在。

如果一个将要位于屏障点或者已经位于屏障点的而执行屏障点任务的线程发生了异常，那么即使唤醒了其它等待的线程，其它等待的线程也会因为循环等待而“死去”，因为再也没有一个线程来唤醒这些第二次进行 park 的线程了。还有一个意图是，如果屏障点都已经损坏了，那么其它将要等待屏障点的再线程挂起就没有意义了。

写到这里的时候非常不幸，用了 4 年多了台灯终于“寿终正寝了”。

其实 CyclicBarrier 还有一个 reset 方法，描述的是手动立即将所有线程中断，恢复屏障点，进行下一组任务的执行。也就是与重新创建一个新的屏障点相比，可能维护的代价要小一些（减少同步，减少上一个 CyclicBarrier 的管理等等）。

本来是想和 Semaphore 一起将的，最后发现铺开后就有点长了，而且也不利于理解和吸收，所以放到下一篇吧。

Semaphore

深入浅出 Java Concurrency (12): 锁机制 part 7 信号量（Semaphore）

Semaphore 是一个计数信号量。从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动。

说白了，Semaphore 是一个计数器，在计数器不为 0 的时候对线程就放行，一旦达到 0，那么所有请求资源的新线程都会被阻塞，包括增加请求到许可的线程，也就是说 Semaphore 不是可重入的。每一次请求一个许可都会导致计数器减少 1，同样每次释放一个许可都会导致计数器增加 1，一旦达到了 0，新的许可请求线程将被挂起。

缓存池整好使用此思想来实现的，比如链接池、对象池等。

清单 1 对象池

```
package xylz.study.concurrency.lock;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ObjectCache<T> {
    public interface ObjectFactory<T> {
        T makeObject();
    }
    class Node {
        T obj;
        Node next;
    }
    final int capacity;
    final ObjectFactory<T> factory;
    final Lock lock = new ReentrantLock();
```

```
final Semaphore semaphore;
private Node head;
private Node tail;
public ObjectCache(int capacity, ObjectFactory<T> factory) {
   this.capacity = capacity;
   this.factory = factory;
   this.semaphore = new Semaphore(this.capacity);
   this.head = null;
   this.tail = null;
}
public T getObject() throws InterruptedException {
   semaphore.acquire();
   return getNextObject();
}
private T getNextObject() {
   lock.lock();
   try {
      if (head == null) {
         return factory.makeObject();
      } else {
         Node ret = head;
         head = head.next;
         if (head == null) tail = null;
         ret.next = null;//help GC
         return ret.obj;
      }
   } finally {
      lock.unlock();
   }
}
private void returnObjectToPool(T t) {
   lock.lock();
   try {
```

```
            Node node = new Node();
            node.obj = t;
            if (tail == null) {
                head = tail = node;
            } else {
                tail.next = node;
                tail = node;
            }
        } finally {
            lock.unlock();
        }
    }
    public void returnObject(T t) {
        returnObjectToPool(t);
        semaphore.release();
    }
}
```

清单 1 描述了一个基于信号量 Semaphore 的对象池实现。此对象池最多支持 capacity 个对象，这在构造函数中传入。对象池有一个基于 FIFO 的队列，每次从对象池的头结点开始取对象，如果头结点为空就直接构造一个新的对象返回。否则将头结点对象取出，并且头结点往后移动。特别要说明的如果对象的个数用完了，那么新的线程将被阻塞，直到有对象被返回回来。返还对象时将对象加入 FIFO 的尾节点并且释放一个空闲的信号量，表示对象池中增加一个可用对象。

实际上对象池、线程池的原理大致上就是这样的，只不过真正的对象池、线程池要处理比较复杂的逻辑，所以实现起来还需要做很多的工作，例如超时机制，自动回收机制，对象的有效期等等问题。

这里特别说明的是信号量只是在信号不够的时候挂起线程，但是并不能保证信号量足够的时候获取对象和返还对象是线程安全的，所以在清单 1 中仍然需要锁 Lock 来保证并发的正确性。

将信号量初始化为 1，使得它在使用时最多只有一个可用的许可，从而可用作一个相互排斥的锁。这通常也称为二进制信号量，因为它只能有两种状态：一个可用的许可，或零个可用的许可。按此方式使用时，二进制信号量具有某种属性（与很多 Lock 实现不同），即可以由线程释放“锁”，而不是由所有者（因为信号量没有所有权的概念）。在某些专门的上下文（如死锁恢复）中这会很有用。

上面这段话的意思是说当某个线程 A 持有信号量数为 1 的信号量时，其它线程只能等待此线程释放资源才能继续，这时候持有信号量的线程 A 就相当于持有了“锁”，其它线程的继续就需要这把锁，于是线程 A 的释放才能决定其它线程的运行，相当于扮演了“锁”的角色。

另外同公平锁非公平锁一样，信号量也有公平性。如果一个信号量是公平的表示线程在获取信号量时按 FIFO 的顺序得到许可，也就是按照请求的顺序得到释放。这里特别说明的是：所谓请求的顺序是指在请求信号量而进入 FIFO 队列的顺序，有可能某个线程先请求信号而后进去请求队列，那么次线程获取信号量的顺序就会晚于其后请求但是先进入请求队列的线程。这个在公平锁和非公平锁中谈过很多。

除了 acquire 以外，Semaphore 还有几种类似的 acquire 方法，这些方法可以更好的处理中断和超时或者异步等特性，可以参考 JDK API。

按照同样的学习原则，下面对主要的实现进行分析。Semaphore 的 acquire 方法实际上访问的是 **AQS** 的 acquireSharedInterruptibly(arg)方法。这个可以参考 **CountDownLatch** 一节或者 **AQS** 一节。

所以 Semaphore 的 await 实现也是比较简单的。与 CountDownLatch 不同的是，Semaphore 区分公平信号和非公平信号。

清单 2 公平信号获取方法

```
protected int tryAcquireShared(int acquires) {
    Thread current = Thread.currentThread();
    for (;;) {
        Thread first = getFirstQueuedThread();
        if (first != null && first != current)
            return -1;
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

清单 3 非公平信号获取方法

```
protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}
final int nonfairTryAcquireShared(int acquires) {
    for (;;) {
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

对比清单 2 和清单 3 可以看到，公平信号和非公平信号在于第一次尝试能否获取信号时，公平信号量总是将当前线程进入 AQS 的 CLH 队列进行排队（因为第一次尝试时队列的头结点线程很有可能不是当前线程，当然不排除同一个线程第二次进入信号量），从而根据 AQS 的 CLH 队列的顺序 FIFO 依次获取信号量；而对于非公平信号量，第一次立即尝试能否拿到信号量，一旦信号量的剩余数 available 大于请求数（acquires 通常为 1），那么线程就立即得到了释放，而不需要进行 AQS 队列进行排队。只有 remaining<0 的时候（也就是信号量不够的时候）才会进入 AQS 队列。

所以非公平信号量的吞吐量总是要比公平信号量的吞吐量要大，但是需要强调的是非公平信号量和非公平锁一样存在“饥渴死”的现象，也就是说活跃线程可能总是拿到信号量，而非活跃线程可能难以拿到信号量。而对于公平信号量由于总是靠请求的线程的顺序来获取信号量，所以不存在此问题。

深入浅出 Java Concurrency (13): 锁机制 part 8 读写锁 (ReentrantReadWriteLock) (1)

从这一节开始介绍锁里面的最后一个工具：读写锁(ReadWriteLock)。

ReentrantLock 实现了标准的互斥操作，也就是一次只能有一个线程持有锁，也即所谓独占锁的概念。前面的章节中一直在强调这个特点。显然这个特点在一定程度上面减低了吞吐量，实际上独占锁是一种保守的锁策略，在这种情况下任何“读/读”，“写/读”，“写/写”操作都不能同时发生。但是同样需要强调的一个概念是，锁是有一定的开销的，当并发比较大的时候，锁的开销就比较客观了。所以如果可能的话就尽量少用锁，非要用锁的话就尝试看能否改造为读写锁。

ReadWriteLock 描述的是：一个资源能够被多个读线程访问，或者被一个写线程访问，但是不能同时存在读写线程。也就是说读写锁使用的场合是一个共享资源被大量读取操作，而只有少量的写操作（修改数据）。清单 1 描述了 ReadWriteLock 的 API。

清单 1 ReadWriteLock 接口

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

清单 1 描述的 ReadWriteLock 结构，这里需要说明的是 ReadWriteLock 并不是 Lock 的子接口，只不过 ReadWriteLock 借助 Lock 来实现读写两个视角。在 ReadWriteLock 中每次读取共享数据就需要读取锁，当需要修改共享数据时就需要写入锁。看起来好像是两个锁，但其实不尽然，在下一节中的分析中会解释这点奥秘。

在 JDK 6 里面 ReadWriteLock 的实现是 ReentrantReadWriteLock。

清单 2 SimpleConcurrentMap

```
package xylz.study.concurrency.lock;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
```

```
public class SimpleConcurrentMap<K, V> implements Map<K, V> {
    final ReadWriteLock lock = new ReentrantReadWriteLock();
    final Lock r = lock.readLock();
    final Lock w = lock.writeLock();
    final Map<K, V> map;
    public SimpleConcurrentMap(Map<K, V> map) {
        this.map = map;
        if (map == null) throw new NullPointerException();
    }
    public void clear() {
        w.lock();
        try {
            map.clear();
        } finally {
            w.unlock();
        }
    }
    public boolean containsKey(Object key) {
        r.lock();
        try {
            return map.containsKey(key);
        } finally {
            r.unlock();
        }
    }
    public boolean containsValue(Object value) {
        r.lock();
        try {
            return map.containsValue(value);
        } finally {
            r.unlock();
        }
    }
```

```
public Set<java.util.Map.Entry<K, V>> entrySet() {
    throw new UnsupportedOperationException();
}
public V get(Object key) {
    r.lock();
    try {
        return map.get(key);
    } finally {
        r.unlock();
    }
}
public boolean isEmpty() {
    r.lock();
    try {
        return map.isEmpty();
    } finally {
        r.unlock();
    }
}
public Set<K> keySet() {
    r.lock();
    try {
        return new HashSet<K>(map.keySet());
    } finally {
        r.unlock();
    }
}
public V put(K key, V value) {
    w.lock();
    try {
        return map.put(key, value);
    } finally {
        w.unlock();
```

```
    }
}
public void putAll(Map<? extends K, ? extends V> m) {
    w.lock();
    try {
        map.putAll(m);
    } finally {
        w.unlock();
    }
}
public V remove(Object key) {
    w.lock();
    try {
        return map.remove(key);
    } finally {
        w.unlock();
    }
}
public int size() {
    r.lock();
    try {
        return map.size();
    } finally {
        r.unlock();
    }
}
public Collection<V> values() {
    r.lock();
    try {
        return new ArrayList<V>(map.values());
    } finally {
        r.unlock();
    }
```

```
    }
}
```

清单 2 描述的是用读写锁实现的一个线程安全的 Map。其中需要特别说明的是并没有实现 entrySet()方法，这是因为实现这个方法比较复杂，在后面章节中讲到 ConcurrentHashMap 的时候会具体谈这些细节。另外这里 keySet()和 values()也没有直接返回 Map 的视图，而是一个映射原有元素的新视图，其实这个 entrySet()一样，是为了保护原始 Map 的数据逻辑，防止不正确的修改导致原始 Map 发生数据错误。特别说明的是在没有特别需求的情况下没有必要按照清单 2 写一个线程安全的 Map 实现，因为 ConcurrentHashMap 已经完成了此操作。

ReadWriteLock 需要严格区分读写操作，如果读操作使用了写入锁，那么降低读操作的吞吐量，如果写操作使用了读取锁，那么就可能发生数据错误。

另外 ReentrantReadWriteLock 还有以下几个特性：

- **公平性**
  - 非公平锁（默认） 这个和独占锁的非公平性一样，由于读线程之间没有锁竞争，所以读操作没有公平性和非公平性，写操作时，由于写操作可能立即获取到锁，所以会推迟一个或多个读操作或者写操作。因此非公平锁的吞吐量要高于公平锁。
  - 公平锁 利用 AQS 的 CLH 队列，释放当前保持的锁（读锁或者写锁）时，优先为等待时间最长的那个写线程分配写入锁，当前前提是写线程的等待时间要比所有读线程的等待时间要长。同样一个线程持有写入锁或者有一个写线程已经在等待了，那么试图获取公平锁的（非重入）所有线程（包括读写线程）都将被阻塞，直到最先的写线程释放锁。如果读线程的等待时间比写线程的等待时间还有长，那么一旦上一个写线程释放锁，这一组读线程将获取锁。
- **重入性**
  - 读写锁允许读线程和写线程按照请求锁的顺序重新获取读取锁或者写入锁。当然了只有写线程释放了锁，读线程才能获取重入锁。
  - 写线程获取写入锁后可以再次获取读取锁，但是读线程获取读取锁后却不能获取写入锁。
  - 另外读写锁最多支持 65535 个递归写入锁和 65535 个递归读取锁。
- **锁降级**
  - 写线程获取写入锁后可以获取读取锁，然后释放写入锁，这样就从写入锁变成了读取锁，从而实现锁降级的特性。
- **锁升级**
  - 读取锁是不能直接升级为写入锁的。因为获取一个写入锁需要释放所有读取锁，所以如果有两个读取锁视图获取写入锁而都不释放读取锁时就会发生死锁。
- **锁获取中断**
  - 读取锁和写入锁都支持获取锁期间被中断。这个和独占锁一致。
- **条件变量**
  - 写入锁提供了条件变量(Condition)的支持，这个和独占锁一致，但是读取锁却不允许获取条件变量，将得到一个 UnsupportedOperationException 异常。
- **重入数**
  - 读取锁和写入锁的数量最大分别只能是 65535（包括重入数）。这在下节中有介绍。

上面几个特性对读写锁的理解很有帮助，而且也是必要的，另外在下一节中讲 ReadWriteLock 的实现会用到这些知识的。

深入浅出 Java Concurrency (14): 锁机制 part 9 读写锁 (ReentrantReadWriteLock) (2)

这一节主要是谈谈读写锁的实现。

上一节中提到，ReadWriteLock 看起来有两个锁：readLock/writeLock。如果真的是两个锁的话，它们之间又是如何相互影响的呢？

事实上在 ReentrantReadWriteLock 里锁的实现是靠 java.util.concurrent.locks.ReentrantReadWriteLock.Sync 完成的。这个类看起来比较眼熟，实际上它是 AQS 的一个子类，这中类似的结构在 CountDownLatch、ReentrantLock、Semaphore 里面都存在。同样它也有两种实现：公平锁和非公平锁，也就是 java.util.concurrent.locks.ReentrantReadWriteLock.FairSync 和 java.util.concurrent.locks.ReentrantReadWriteLock.NonfairSync。这里暂且不提。

在 ReentrantReadWriteLock 里面的锁主体就是一个 Sync，也就是上面提到的 FairSync 或者 NonfairSync，所以说实际上只有一个锁，只是在获取读取锁和写入锁的方式上不一样，所以前面才有读写锁是独占锁的两个不同视图一说。

ReentrantReadWriteLock 里面有两个类：ReadLock/WriteLock，这两个类都是 Lock 的实现。

清单 1 ReadLock 片段

```
public static class ReadLock implements Lock, java.io.Serializable  {
    private final Sync sync;
    protected ReadLock(ReentrantReadWriteLock lock) {
        sync = lock.sync;
    }
    public void lock() {
        sync.acquireShared(1);
    }
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }
    public  boolean tryLock() {
        return sync.tryReadLock();
    }
    public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException {
        return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }
    public  void unlock() {
        sync.releaseShared(1);
    }
    public Condition newCondition() {
        throw new UnsupportedOperationException();
    }
}
```

清单 2 WriteLock 片段

```
public static class WriteLock implements Lock, java.io.Serializable  {
    private final Sync sync;
    protected WriteLock(ReentrantReadWriteLock lock) {
        sync = lock.sync;
    }
    public void lock() {
        sync.acquire(1);
    }
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1);
    }
    public boolean tryLock( ) {
        return sync.tryWriteLock();
    }
    public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(timeout));
    }
    public void unlock() {
        sync.release(1);
    }
    public Condition newCondition() {
        return sync.newCondition();
    }
    public boolean isHeldByCurrentThread() {
        return sync.isHeldExclusively();
    }
    public int getHoldCount() {
        return sync.getWriteHoldCount();
    }
}
```

清单 1 描述的是读锁的实现，清单 2 描述的是写锁的实现。显然 WriteLock 就是一个独占锁，这和 ReentrantLock 里面的实现几乎相同，都是使用了 AQS 的 acquire/release 操作。当然了在内部处理方式上与 ReentrantLock 还是有一点不同的。对比清单 1 和清单 2 可以看到，ReadLock 获取的是共享锁，WriteLock 获取的是独占锁。

在 AQS 章节中介绍到 AQS 中有一个 state 字段（int 类型，32 位）用来描述有多少线程获持有锁。在独占锁的时代这个值通常是 0 或者 1（如果是重入的就是重入的次数），在共享锁的时代就是持有锁的数量。在上一节中谈到，ReadWriteLock 的读、写锁是相关但是又不一致的，所以需要两个数来描述读锁（共享锁）和写锁（独占锁）的数量。显然现在一个 state 就不够用了。于是在 ReentrantReadWrilteLock 里面将这个字段一分为二，高位 16 位表示共享锁的数量，低位 16 位表示独占锁的数量（或者重入数量）。2^16-1=65536，这就是上节中提到的为什么共享锁和独占锁的数量最大只能是 65535 的原因了。

有了上面的知识后再来分析读写锁的获取和释放就容易多了。

清单 3 写入锁获取片段

```
protected final boolean tryAcquire(int acquires) {
    Thread current = Thread.currentThread();
    int c = getState();
    int w = exclusiveCount(c);
    if (c != 0) {
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
    }
    if ((w == 0 && writerShouldBlock(current)) ||
        !compareAndSetState(c, c + acquires))
        return false;
    setExclusiveOwnerThread(current);
    return true;
}
```

清单 3 是写入锁获取的逻辑片段，整个工作流程是这样的：

1. 持有锁线程数非 0（c=getState()不为 0），如果写线程数（w）为 0（那么读线程数就不为 0）或者独占锁线程（持有锁的线程）不是当前线程就返回失败，或者写入锁的数量（其实是重入数）大于 65535 就抛出一个 Error 异常。否则进行 2。
2. 如果当且写线程数位 0（那么读线程也应该为 0，因为步骤 1 已经处理 c!=0 的情况），并且当前线程需要阻塞那么就返回失败；如果增加写线程数失败也返回失败。否则进行 3。
3. 设置独占线程（写线程）为当前线程，返回 true。

清单 3 中 exclusiveCount(c)就是获取写线程数（包括重入数），也就是 state 的低 16 位值。另外这里有一段逻辑是当前写线程是否需要阻塞 writerShouldBlock(current)。清单 4 和清单 5 就是公平锁和非公平锁中是否需要阻塞的片段。很显然对于非公平锁而言总是不阻塞当前线程，而对于公平锁而言如果 AQS 队列不为空或者当前线程不是在 AQS 的队列头那么就阻塞线程，直到队列前面的线程处理完锁逻辑。

清单 4 公平读写锁写线程是否阻塞

```
final boolean writerShouldBlock(Thread current) {
    return !isFirst(current);
```

```
}
```

清单 5 非公平读写锁写线程是否阻塞

```
final boolean writerShouldBlock(Thread current) {
    return false;
}
```

写入锁的获取逻辑清楚后，释放锁就比较简单了。清单 6 描述的写入锁释放逻辑片段，其实就是检测下剩下的写入锁数量，如果是 0 就将独占锁线程清空（意味着没有线程获取锁），否则就是说当前是重入锁的一次释放，所以不能将独占锁线程清空。然后将剩余线程状态数写回 AQS。

清单 6 写入锁释放逻辑片段

```
protected final boolean tryRelease(int releases) {
    int nextc = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    if (exclusiveCount(nextc) == 0) {
        setExclusiveOwnerThread(null);
        setState(nextc);
        return true;
    } else {
        setState(nextc);
        return false;
    }
}
```

清单 3~6 描述的写入锁的获取释放过程。读取锁的获取和释放过程要稍微复杂些。 清单 7 描述的是读取锁的获取过程。

清单 7 读取锁获取过程片段

```
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
        return -1;
    if (sharedCount(c) == MAX_COUNT)
        throw new Error("Maximum lock count exceeded");
    if (!readerShouldBlock(current) &&
        compareAndSetState(c, c + SHARED_UNIT)) {
```

```
            HoldCounter rh = cachedHoldCounter;
            if (rh == null || rh.tid != current.getId())
                cachedHoldCounter = rh = readHolds.get();
            rh.count++;
            return 1;
        }
        return fullTryAcquireShared(current);
    }
    final int fullTryAcquireShared(Thread current) {
        HoldCounter rh = cachedHoldCounter;
        if (rh == null || rh.tid != current.getId())
            rh = readHolds.get();
        for (;;) {
            int c = getState();
            int w = exclusiveCount(c);
            if ((w != 0 && getExclusiveOwnerThread() != current) ||
                ((rh.count | w) == 0 && readerShouldBlock(current)))
                return -1;
            if (sharedCount(c) == MAX_COUNT)
                throw new Error("Maximum lock count exceeded");
            if (compareAndSetState(c, c + SHARED_UNIT)) {
                cachedHoldCounter = rh; // cache for release
                rh.count++;
                return 1;
            }
        }
    }
```

读取锁获取的过程是这样的：

1. 如果写线程持有锁（也就是独占锁数量不为 0），并且独占线程不是当前线程，那么就返回失败。因为允许写入线程获取锁的同时获取读取锁。否则进行 2。
2. 如果读线程请求锁数量达到了 65535（包括重入锁），那么就跑出一个错误 Error，否则进行 3。
3. 如果读线程不用等待（实际上是是否需要公平锁），并且增加读取锁状态数成功，那么就返回成功，否则进行 4。
4. 步骤 3 失败的原因是 CAS 操作修改状态数失败，那么就需要循环不断尝试去修改状态直到成功或者锁被写入线程占有。实际上是过程 3 的不断尝试直到 CAS 计数成功或者被写入线程占有锁。

在清单 7 中有一个对象 HoldCounter，这里暂且不提这是什么结构和为什么存在这样一个结构。

接下来根据清单 8 我们来看如何释放一个读取锁。同样先不理 HoldCounter，关键的在于 for 循环里面，其实就是一个不断尝试的 CAS 操作，直到修改状态成功。前面说过 state 的高 16 位描述的共享锁（读取锁）的数量，所以每次都需要减去 2^16，这样就相当于读取锁数量减 1。实际上 SHARED_UNIT=1<<16。

清单 8 读取锁释放过程

```
protected final boolean tryReleaseShared(int unused) {
    HoldCounter rh = cachedHoldCounter;
    Thread current = Thread.currentThread();
    if (rh == null || rh.tid != current.getId())
        rh = readHolds.get();
    if (rh.tryDecrement() <= 0)
        throw new IllegalMonitorStateException();
    for (;;) {
        int c = getState();
        int nextc = c - SHARED_UNIT;
        if (compareAndSetState(c, nextc))
            return nextc == 0;
    }
}
```

好了，现在回头看 HoldCounter 到底是一个什么东西。首先我们可以看到只有在获取共享锁（读取锁）的时候加 1，也只有在释放共享锁的时候减 1 有作用，并且在释放锁的时候抛出了一个 IllegalMonitorStateException 异常。而我们知道 IllegalMonitorStateException 通常描述的是一个线程操作一个不属于自己的监视器对象的引发的异常。也就是说这里的意思是一个线程释放了一个不属于自己或者不存在的共享锁。

前面的章节中一再强调，对于共享锁，其实并不是锁的概念，更像是计数器的概念。一个共享锁就相对于一次计数器操作，一次获取共享锁相当于计数器加 1，释放一个共享锁就相当于计数器减 1。显然只有线程持有了共享锁（也就是当前线程携带一个计数器，描述自己持有多少个共享锁或者多重共享锁），才能释放一个共享锁。否则一个没有获取共享锁的线程调用一次释放操作就会导致读写锁的 state（持有锁的线程数，包括重入数）错误。

明白了 HoldCounter 的作用后我们就可以猜到它的作用其实就是当前线程持有共享锁（读取锁）的数量，包括重入的数量。那么这个数量就必须和线程绑定在一起。

在 Java 里面将一个对象和线程绑定在一起，就只有 ThreadLocal 才能实现了。所以毫无疑问 HoldCounter 就应该是绑定到线程上的一个计数器。

清单 9 线程持有读取锁数量的计数器

```
static final class HoldCounter {
    int count;
    final long tid = Thread.currentThread().getId();
    int tryDecrement() {
        int c = count;
        if (c > 0) count = c - 1;
```

```
            return c;
        }
    }
    static final class ThreadLocalHoldCounter
        extends ThreadLocal<HoldCounter> {
        public HoldCounter initialValue() {
            return new HoldCounter();
        }
    }
```

清单 9 描述的是线程持有读取锁数量的计数器。可以看到这里使用 ThreadLocal 将 HoldCounter 绑定到当前线程上，同时 HoldCounter 也持有线程 Id，这样在释放锁的时候才能知道 ReadWriteLock 里面缓存的上一个读取线程（cachedHoldCounter）是否是当前线程。这样做的好处是可以减少 ThreadLocal.get()的次数，因为这也是一个耗时操作。需要说明的是这样 HoldCounter 绑定线程 id 而不绑定线程对象的原因是避免 HoldCounter 和 ThreadLocal 互相绑定而 GC 难以释放它们（尽管 GC 能够智能的发现这种引用而回收它们，但是这需要一定的代价），所以其实这样做只是为了帮助 GC 快速回收对象而已。

除了 readLock()和 writeLock()外，Lock 对象还允许 tryLock()，那么 ReadLock 和 WriteLock 的 tryLock()不一样。清单 10 和清单 11 分别描述了读取锁的 tryLock()和写入锁的 tryLock()。

读取锁 tryLock()也就是 tryReadLock()成功的条件是：没有写入锁或者写入锁是当前线程，并且读线程共享锁数量没有超过 65535 个。

写入锁 tryLock()也就是 tryWriteLock()成功的条件是：没有写入锁或者写入锁是当前线程，并且尝试一次修改 state 成功。

清单 10 读取锁的 tryLock()

```
final boolean tryReadLock() {
    Thread current = Thread.currentThread();
    for (;;) {
        int c = getState();
        if (exclusiveCount(c) != 0 && getExclusiveOwnerThread() != current) return false;
        if (sharedCount(c) == MAX_COUNT) throw new Error("Maximum lock count exceeded");
        if (compareAndSetState(c, c + SHARED_UNIT)) {
            HoldCounter rh = cachedHoldCounter;
            if (rh == null || rh.tid != current.getId()) cachedHoldCounter = rh = readHolds.get();
            rh.count++;
            return true;
        }
    }
}
```

清单 11 写入锁的 tryLock()

```
final boolean tryWriteLock() {
    Thread current = Thread.currentThread();
    int c = getState();
    if (c != 0) {
        int w = exclusiveCount(c);
        if (w == 0 ||current != getExclusiveOwnerThread()) return false;
        if (w == MAX_COUNT) throw new Error("Maximum lock count exceeded");
    }
    if (!compareAndSetState(c, c + 1))   return false;
    setExclusiveOwnerThread(current);
    return true;
}
```

整个读写锁的逻辑大概就这么多，其实真正研究起来也不是很复杂，真正复杂的东西都在 AQS 里面。

锁部分的原理和思想都介绍完了，下一节里面会对锁机进行小节，并对线程并发也会有一些简单的小节。

深入浅出 Java Concurrency (15): 锁机制 part 10 锁的一些其它问题

主要谈谈锁的性能以及其它一些理论知识，内容主要的出处是《Java Concurrency in Practice》，结合自己的理解和实际应用对锁机制进行一个小小的总结。

首先需要强调的一点是：所有锁（包括内置锁和高级锁）都是有性能消耗的，也就是说在高并发的情况下，由于锁机制带来的上下文切换、资源同步等消耗是非常可观的。在某些极端情况下，线程在锁上的消耗可能比线程本身的消耗还要多。所以如果可能的话，在任何情况下都尽量少用锁，如果不可避免那么采用非阻塞算法是一个不错的解决方案，但是却也不是绝对的。

内部锁

Java 语言通过 synchronized 关键字来保证原子性。这是因为每一个 Object 都有一个隐含的锁，这个也称作监视器对象。在进入 synchronized 之前自动获取此内部锁，而一旦离开此方式（不管通过和中方式离开此方法）都会自动释放锁。显然这是一个独占锁，每个锁请求之间是互斥的。相对于前面介绍的众多高级锁（Lock/ReadWriteLock 等），synchronized 的代价都比后者要高。但是 synchronized 的语法比较简单，而且也比较容易使用和理解，不容易写法上的错误。而我们知道 Lock 一旦调用了 lock()方法获取到锁而未正确释放的话很有可能就死锁了。所以 Lock 的释放操作总是跟在 finally 代码块里面，这在代码结构上也是一次调整和冗余。另外前面介绍中说过 Lock 的实现已经将硬件资源用到了极致，所以未来可优化的空间不大，除非硬件有了更高的性能。但是 synchronized 只是规范的一种实现，这在不同的平台不同的硬件还有很高的提升空间，未来 Java 在锁上的优化也会主要在这上面。

性能

由于锁总是带了性能影响，所以是否使用锁和使用锁的场合就变得尤为重要。如果在一个高并发的 Web 请求中使用了强制的独占锁，那么就可以发现 Web 的吞吐量将急剧下降。

为了利用并发来提高性能，出发点就是：更有效的利用现有的资源，同时让程序尽可能的开拓更多可用的资源。这意味着机器尽可能的处于忙碌的状态，通常意义是说 CPU 忙于计算，而不是等待。当然 CPU 要做有用的事情，而不是进行无谓的循环。当然在实践中通常会预留一些资源出来以便应急特殊情况，这在以后的线程池并发中可以看到很多例子。

线程阻塞

锁机制的实现通常需要操作系统提供支持，显然这会增加开销。当锁竞争的时候，失败的线程必然会发生阻塞。JVM 既能自旋等待（不断尝试，知道成功，很多 CAS 就是这样实现的），也能够在操作系统中挂起阻塞的线程，直到超时或者被唤醒。通常情况下这取决于上下文切换的开销以及与获取锁需要等待的时间二者之间的关系。自旋等待适合于比较短的等待，而挂起线程比较适合那些比较耗时的等待。

挂起一个线程可能是因为无法获取到锁，或者需要某个特定的条件，或者耗时的 I/O 操作。挂起一个线程需要两次额外的上下文切换以及操作系统、缓存等多资源的配合：如果线程被提前换出，那么一旦拿到锁或者条件满足，那么又需要将线程换回执行队列，这对线程而言，两次上下文切换可能比较耗时。

锁竞争

影响锁竞争性的条件有两个：锁被请求的频率和每次持有锁的时间。显然当而这二者都很小的时候，锁竞争不会成为主要的瓶颈。但是如果锁使用不当，导致二者都比较大，那么很有可能 CPU 不能有效的处理任务，任务被大量堆积。

所以减少锁竞争的方式有下面三种：

1. 减少锁持有的时间
2. 减少锁请求的频率
3. 采用共享锁取代独占锁

死锁

如果一个线程永远不释放另外一个线程需要的资源那么就会导致死锁。这有两种情况：一种情况是线程 A 永远不释放锁，结果 B 一直拿不到锁，所以线程 B 就“死掉”了；第二种情况下，线程 A 拥有线程 B 需要的锁 Y，同时线程 B 拥有线程 A 需要的锁 X，那么这时候线程 A/B 互相依赖对方释放锁，于是二者都“死掉”了。

还有一种情况为发生死锁，如果一个线程总是不能被调度，那么等待此线程结果的线程可能就死锁了。这种情况叫做线程饥饿死锁。比如说在前面介绍的非公平锁中，如果某些线程非常活跃，在高并发情况下这类线程可能总是拿到锁，那么那些活跃度低的线程可能就一直拿不到锁，这样就发生了“饥饿死”。

避免死锁的解决方案是：尽可能的按照锁的使用规范请求锁，另外锁的请求粒度要小（不要在不需要锁的地方占用锁，锁不用了尽快释放）；在高级锁里面总是使用 tryLock 或者定时机制（这个以后会讲，就是指定获取锁超时的时间，如果时间到了还没有获取到锁那么就放弃）。高级锁（Lock）里面的这两种方式可以有效的避免死锁。

活锁

活锁描述的是线程总是尝试某项操作却总是失败的情况。这种情况下尽管线程没有被阻塞，但是人物却总是不能被执行。比如在一个死循环里面总是尝试做某件事，结果却总是失败，现在线程将永远不能跳出这个循环。另外一种情况是在一个队列中每次从队列头取出一个任务来执行，每次都失败，然后将任务放入队列头，接下来再一次从队列头取出任务执行，仍然失败。

还有一种活锁方式发生在“碰撞协让”情况下：两个人过独木桥，如果在半路相撞，双方礼貌退出去然后再试一次。如果总是失败，那么这两个任务将一直无法得到执行。

总之解决锁问题的关键就是：从简单的开始，先保证正确，然后再开始优化。

并发容器

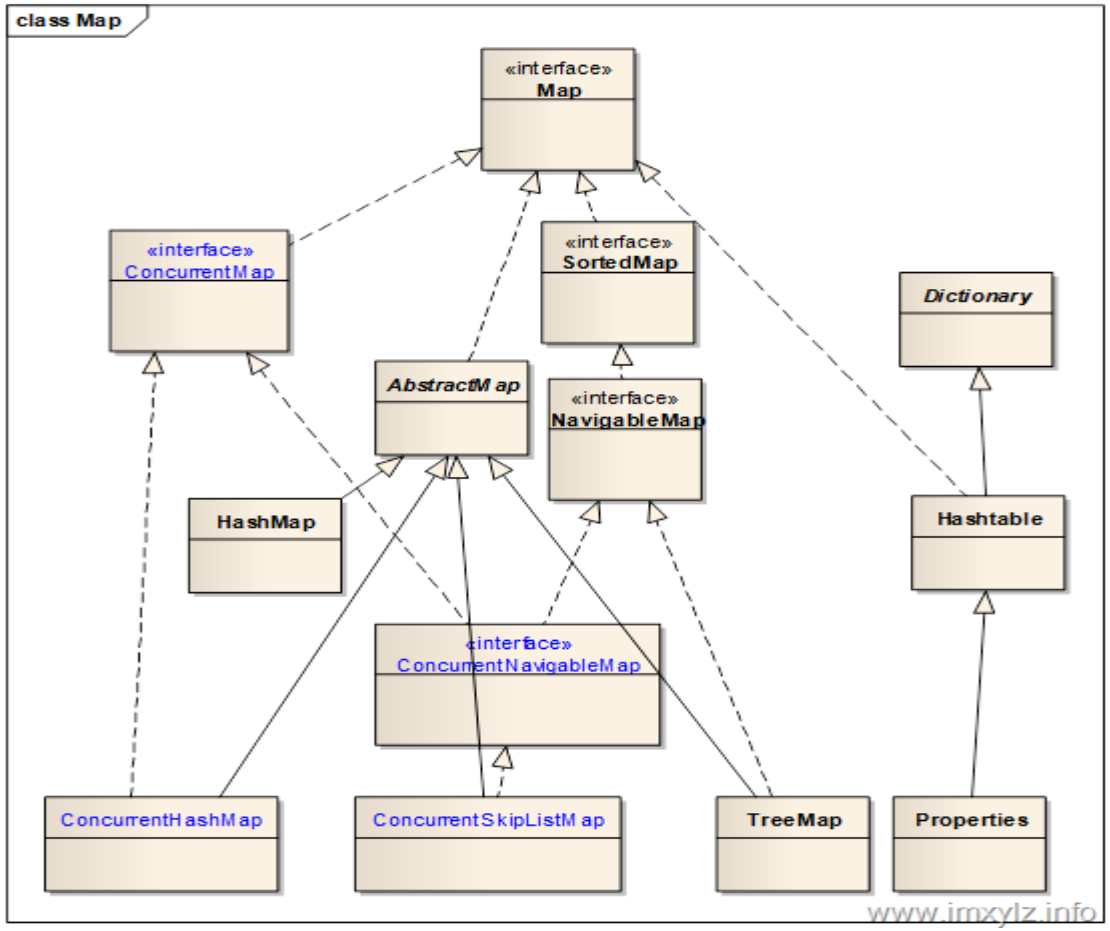
ConcurrentMap

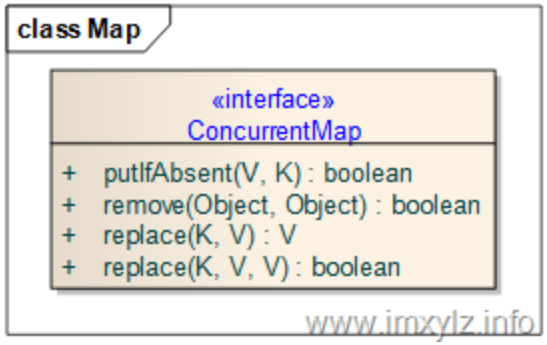
深入浅出 Java Concurrency (16): 并发容器 part 1 ConcurrentMap (1)

从这一节开始正式进入并发容器的部分，来看看 JDK 6 带来了哪些并发容器。

在 JDK 1.4 以下只有 Vector 和 Hashtable 是线程安全的集合（也称并发容器，Collections.synchronized*系列也可以看作是线程安全的实现）。从 JDK 5 开始增加了线程安全的 Map 接口 ConcurrentMap 和线程安全的队列 BlockingQueue（尽管 Queue 也是同时期引入的新的集合，但是规范并没有规定一定是线程安全的，事实上一些实现也不是线程安全的，比如 PriorityQueue、ArrayDeque、LinkedList 等，在 Queue 章节中会具体讨论这些队列的结构图和实现）。

在介绍 ConcurrencyMap 之前先来回顾下 Map 的体系结构。下图描述了 Map 的体系结构，其中蓝色字体的是 JDK 5 以后新增的并发容器。





针对上图有以下几点说明：

1. Hashtable 是 JDK 5 之前 Map 唯一线程安全的内置实现（Collections.synchronizedMap 不算）。特别说明的是 Hashtable 的 t 是小写的（不知道为啥），Hashtable 继承的是 Dictionary（Hashtable 是其唯一公开的子类），并**不继承 AbstractMap 或者 HashMap**。尽管 Hashtable 和 HashMap 的结构非常类似，但是他们之间并没有多大联系。
2. ConcurrentHashMap 是 HashMap 的线程安全版本，ConcurrentSkipListMap 是 TreeMap 的线程安全版本。
3. 最终可用的线程安全版本 Map 实现是 ConcurrentHashMap/ConcurrentSkipListMap/Hashtable/Properties 四个，但是 Hashtable 是过时的类库，因此如果可以的应该尽可能的使用 ConcurrentHashMap 和 ConcurrentSkipListMap。

回到正题来，这个小节主要介绍 ConcurrentHashMap 的 API 以及应用，下一节才开始将原理和分析。

除了实现 Map 接口里面对象的方法外，ConcurrentHashMap 还实现了 ConcurrentMap 里面的四个方法。

V putIfAbsent(K key,V value)

如果不存在 key 对应的值，则将 value 以 key 加入 Map，否则返回 key 对应的旧值。这个等价于清单 1 的操作：

清单 1 putIfAbsent 的等价操作

```
if (!map.containsKey(key))
   return map.put(key, value);
else
   return map.get(key);
```

在前面的章节中提到过，连续两个或多个原子操作的序列并不一定是原子操作。比如上面的操作即使在 Hashtable 中也不是原子操作。而 putIfAbsent 就是一个线程安全版本的操作的。

有些人喜欢用这种功能来实现单例模式，例如清单 2。

清单 2 一种单例模式的实现

```
package xylz.study.concurrency;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
public class ConcurrentDemo1 {
    private static final ConcurrentMap<String, ConcurrentDemo1> map = new ConcurrentHashMap<String, ConcurrentDemo1>();
    private static ConcurrentDemo1 instance;
    public static ConcurrentDemo1 getInstance() {
        if (instance == null) {
            map.putIfAbsent("INSTANCE", new ConcurrentDemo1());
            instance = map.get("INSTANCE");
```

```
        }
        return instance;
    }
    private ConcurrentDemo1() {
    }
}
```

当然这里只是一个操作的例子，实际上在单例模式文章中有很多的实现和比较。清单 2 在存在大量单例的情况下可能有用，实际情况下很少用于单例模式。但是这个方法避免了向 Map 中的同一个 Key 提交多个结果的可能，有时候在去掉重复记录上很有用（如果记录的格式比较固定的话）。

boolean remove(Object key,Object value)

只有目前将键的条目映射到给定值时，才移除该键的条目。这等价于清单 3 的操作。

清单 3 remove(Object,Object)的等价操作

```
if (map.containsKey(key) && map.get(key).equals(value)) {
   map.remove(key);
   return true;
}
return false;
```

由于集合类通常比较的 hashCode 和 equals 方法，而这两个方法是在 Object 对象里面，因此两个对象如果 hashCode 一致，并且覆盖了 equals 方法后也一致，那么这两个对象在集合类里面就是“相同”的，不管是否是同一个对象或者同一类型的对象。也就是说只要 key1.hashCode()==key2.hashCode() && key1.equals(key2)，那么 key1 和 key2 在集合类里面就认为是一致，哪怕他们的 Class 类型不一致也没关系，所以在很多集合类里面允许通过 Object 来类型来比较（或者定位）。比如说 Map 尽管添加的时候只能通过制定的类型<K,V>，但是删除的时候却允许通过一个 Object 来操作，而不必是 K 类型。

既然 Map 里面有一个 remove(Object)方法，为什么 ConcurrentMap 还需要 remove(Object,Object)方法呢？这是因为尽管 Map 里面的 key 没有变化，但是 value 可能已经被其他线程修改了，如果修改后的值是我们期望的，那么我们就不能拿一个 key 来删除此值，尽管我们的期望值是删除此 key 对于的旧值。

这种特性在原子操作章节的AtomicMarkableReference和AtomicStampedReference里面介绍过。

boolean replace(K key,V oldValue,V newValue)

只有目前将键的条目映射到给定值时，才替换该键的条目。这等价于清单 4 的操作。

清单 4 replace(K,V,V)的等价操作

```
if (map.containsKey(key) && map.get(key).equals(oldValue)) {
   map.put(key, newValue);
```

```
    return true;
}
return false;
```

V replace(K key,V value)

只有当前键存在的时候更新此键对于的值。这等价于清单 5 的操作。

清单 5 replace(K,V)的等价操作

```
if (map.containsKey(key)) {
    return map.put(key, value);
}
return null;
```

replace(K,V,V)相比 replace(K,V)而言，就是增加了匹配 oldValue 的操作。

其实这 4 个扩展方法，是 ConcurrentMap 附送的四个操作，其实我们更关心的是 Map 本身的操作。当然如果没有这 4 个方法，要完成类似的功能我们可能需要额外的锁，所以有总比没有要好。比如清单 6，如果没有 putIfAbsent 内置的方法，我们如果要完成此操作就需要完全锁住整个 Map，这样就大大降低了 ConcurrentMap 的并发性。这在下一节中有详细的分析和讨论。

清单 6 putIfAbsent 的外部实现

```
public V putIfAbsent(K key, V value) {
    synchronized (map) {
        if (!map.containsKey(key)) return map.put(key, value);
        return map.get(key);
    }
}
```

本来想比较全面和深入的谈谈 ConcurrentHashMap 的，发现网上有很多对 HashMap 和 ConcurrentHashMap 分析的文章，因此本小节尽可能的分析其中的细节，少一点理论的东西，多谈谈内部设计的原理和思想。

要谈 ConcurrentHashMap 的构造，就不得不谈 HashMap 的构造，因此先从 HashMap 开始简单介绍。

HashMap 原理

我们从头开始设想。要将对象存放在一起，如何设计这个容器。目前只有两条路可以走，一种是采用分格技术，每一个对象存放于一个格子中，这样通过对格子的编号就能取到或者遍历对象；另一种技术就是采用串联的方式，将各个对象串联起来，这需要各个对象至少带有下一个对象的索引（或者指针）。显然第一种就是数组的概念，第二种就是链表的概念。所有的容器的实现其实都是基于这两种方式的，不管是数组还是链表，或者二者俱有。HashMap 采用的就是数组的方式。

有了存取对象的容器后还需要以下两个条件才能完成 Map 所需要的条件。

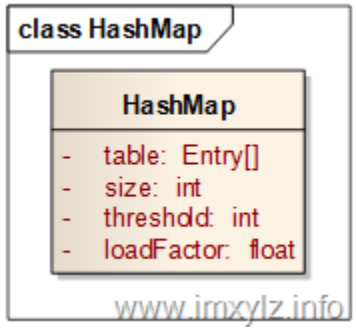
- 能够快速定位元素：Map 的需求就是能够根据一个查询条件快速得到需要的结果，所以这个过程需要的就是尽可能的快。
- 能够自动扩充容量：显然对于容器而然，不需要人工的去控制容器的容量是最好的，这样对于外部使用者来说越少知道底部细节越好，不仅使用方便，也越安全。

首先条件 1，快速定位元素。快速定位元素属于算法和数据结构的范畴，通常情况下哈希（Hash）算法是一种简单可行的算法。所谓**哈希算法**，是将任意长度的二进制值映射为固定长度的较小二进制值。常见的 MD2,MD4,MD5，SHA-1 等都属于 Hash 算法的范畴。具体的算法原理和介绍可以参考相应的算法和数据结构的书籍，但是这里特别提醒一句，由于将一个较大的集合映射到一个较小的集合上，所以必然就存在多个元素映射到同一个元素上的结果，这个叫“碰撞”，后面会用到此知识，暂且不表。

条件 2，如果满足了条件 1，一个元素映射到了某个位置，现在一旦扩充了容量，也就意味着元素映射的位置需要变化。因为对于 Hash 算法来说，调整了映射的小集合，那么原来映射的路径肯定就不复存在，那么就需要对现有重新计算映射路径，也就是所谓的 rehash 过程。

好了有了上面的理论知识后来看 HashMap 是如何实现的。

在 HashMap 中首先由一个对象数组 table 是不可避免的，修饰符 transient 只是表示序列号的时候不被存储而已。size 描述的是 Map 中元素的大小，threshold 描述的是达到指定元素个数后需要扩容，loadFactor 是扩容因子(loadFactor>0)，也就是计算 threshold 的。那么元素的容量就是 table.length，也就是数组的大小。换句话说，如果存取的元素大小达到了整个容量(table.length)的 loadFactor 倍（也就是 table.length*loadFactor 个），那么就需要扩充容量了。在 HashMap 中每次扩容就是将扩大数组的一倍，使数组大小为原来的两倍。



然后接下来看如何将一个元素映射到数组table中。显然要映射的key是一个无尽的超大集合，而table是一个较小的有限集合，那么一种方式就是将key编码后的hashCode值取模映射到table上，这样看起来不错。但是在Java中采用了一种更高效的办法。由于与(&)是比取模(%)更高效的操作，因此Java中采用hash值与数组大小-1 后取与来确定数组索引的。为什么这样做是更有效的？参考资料 7对这一块进行非常详细的分析，这篇文章的作者非常认真，也非常仔细的分析了里面包含的思想。

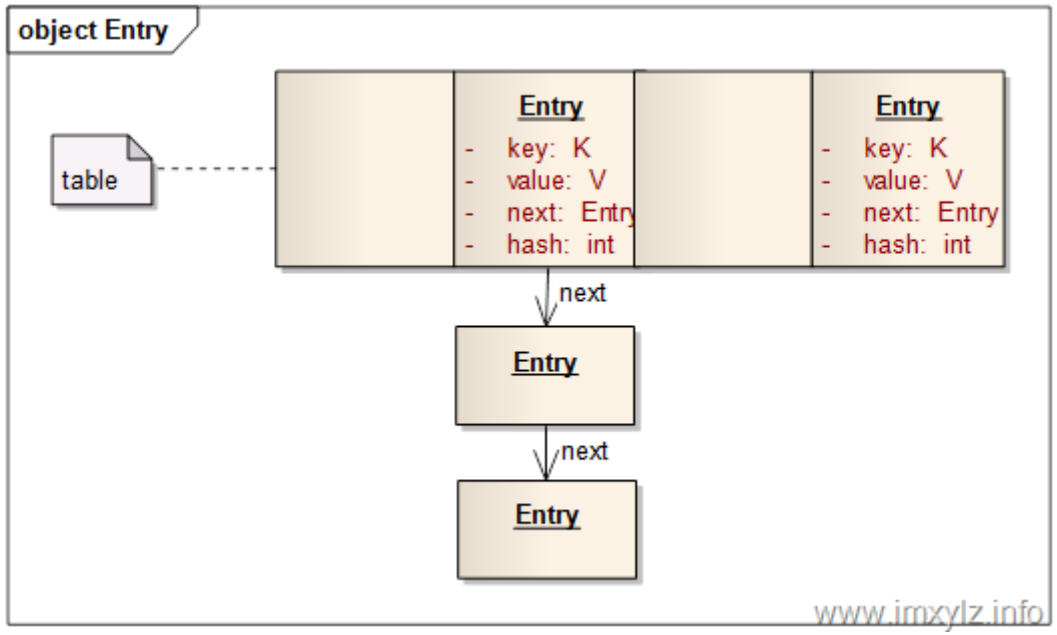
清单 1 indexFor 片段

```
static int indexFor(int h, int length) {
    return h & (length-1);
```

}

前面说明，既然是大集合映射到小集合上，那么就必然存在“碰撞”，也就是不同的 key 映射到了相同的元素上。那么 HashMap 是怎么解决这个问题的？在 HashMap 中采用了下面方式，解决了此问题。

1. 同一个索引的数组元素组成一个链表，查找允许时循环链表找到需要的元素。
2. 尽可能的将元素均匀的分布在数组上。



对于问题 1，HashMap 采用了上图的一种数据结构。table 中每一个元素是一个 Map.Entry，其中 Entry 包含了四个数据，key,value,hash,next。key 和 value 是存储的数据；hash 是元素 key 的 Hash 后的表现形式（最终要映射到数组上），这里链表上所有元素的 hash 经过清单 1 的 indexFor 后将得到相同的数组索引；next 是指向下一个元素的索引，同一个链表上的元素就是通过 next 串联起来的。

再来看问题 2 尽可能的将元素均匀的分布在数组上这个问题是怎么解决的。首先清单 2 是将 key 的 hashCode 经过一系列的变换，使之更符合小数据集合的散列模型。

清单 2 hashCode 的二次散列

```
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
```

```
}
```

至于清单 2 为什么这样散列我没有找到依据，也没有什么好的参考资料。参考资料 1 分析了此过程，认为是一种比较有效的方式，有兴趣的可以研究下。

第二点就是在清单 1 的描述中，尽可能的与数组的长度减 1 的数与操作，使之分布均匀。这在参考资料 7 中有介绍。

第三点就是构造数组时数组的长度是 2 的倍数。清单 3 反映了这个过程。为什么要是 2 的倍数？在参考资料 7 中分析说是使元素尽可能的分布均匀。

清单 3 HashMap 构造数组

```
// Find a power of 2 >= initialCapacity
int capacity = 1;
while (capacity < initialCapacity)
    capacity <<= 1;
this.loadFactor = loadFactor;
threshold = (int)(capacity * loadFactor);
table = new Entry[capacity];
```

另外 loadFactor 的默认值 0.75 和 capacity 的默认值 16 是经过大量的统计分析得出的，很久以前我见过相关的数据分析，现在找不到了，有兴趣的可以查询相关资料。这里不再叙述了。

有了上述原理后再来分析 HashMap 的各种方法就不是什么问题的。

清单 4 HashMap 的 get 操作

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
         e != null;
         e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
    return null;
}
```

清单 4 描述的是 HashMap 的 get 操作，在这个操作中首先判断 key 是否为空，因为为空的话总是映射到 table 的第 0 个元素上（可以看上面的清单 2 和清单 1）。然后就需要查找 table 的索引。一旦找到对应的 Map.Entry 元素后就开始遍历此链表。由于不同的 hash 可能映射到同一个 table[index]上，而相同的 key 却同时映射到相同的 hash 上，所以一个 key 和 Entry 对应的条件就是 hash(key)==e.hash 并且 key.equals(e.key)。从这里我们看到，Object.hashCode()

只是为了将相同的元素映射到相同的链表上（Map.Entry），而 Object.equals()才是比较两个元素是否相同的关键！这就是为什么总是成对覆盖 hashCode()和 equals()的原因。

清单 5 HashMap 的 put 操作

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
void addEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        if (size++ >= threshold)
            resize(2 * table.length);
}
```

清单 5 描述的是 HashMap 的 put 操作。对比 get 操作，可以发现，put 实际上是先查找，一旦找到 key 对应的 Entry 就直接修改 Entry 的 value 值，否则就增加一个元素。增加的元素是在链表的头部，也就是占据 table 中的元素，如果 table 中对应索引原来有元素的话就将整个链表添加到新增加的元素的后面。也就是说新增加的元素再次查找的话是优于在它之前添加的同一个链表上的元素。这里涉及到就是扩容，也就是一旦元素的个数达到了扩容因子规定的数量(threhold=table.length*loadFactor)，就将数组扩大一倍。

清单 6 HashMap 扩容过程

```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}
```

清单 6 描述的是 HashMap 扩容的过程。可以看到扩充过程会导致元素数据的所有元素进行重新 hash 计算，这个过程也叫 rehash。显然这是一个非常耗时的过程，否则扩容都会导致所有元素重新计算 hash。因此尽可能的选择合适的初始化大小是有效提高 HashMap 效率的关键。太大了会导致过多的浪费空间，太小了就可能会导致繁重的 rehash 过程。在这个过程中 loadFactor 也可以考虑。

举个例子来说，如果要存储 1000 个元素，采用默认扩容因子 0.75，那么 1024 显然是不够的，因为 1000>0.75*1024 了，所以选择 2048 是必须的，显然浪费了 1048 个空间。如果确定最多只有 1000 个元素，那么扩容因子为 1，那么 1024 是不错的选择。另外需要强调的一点是扩容因此越大，从统计学角度讲意味着链表的长度就也大，也就是在查找元素的时候就需要更多次的循环。所以凡事必然是一个平衡的过程。

这里可能有人要问题，一旦我将 Map 的容量扩大后（也就是数组的大小），这个容量还能减小么？比如说刚开始 Map 中可能有 10000 个元素，运行一旦时间以后 Map 的大小永远不会超过 10 个，那么 Map 的容量能减小到 10 个或者 16 个么？答案就是不能，这个 capacity 一旦扩大后就不能减小了，只能通过构造一个新的 Map 来控制 capacity 了。

HashMap 的几个内部迭代器也是非常重要的，这里限于篇幅就不再展开了，有兴趣的可以自己研究下。

Hashtable 的原理和 HashMap 的原理几乎一样，所以就不讨论了。另外 LinkedHashMap 是在 Map.Entry 的基础上增加了 before/after 两个双向索引，用来将所有 Map.Entry 串联起来，这样就可以遍历或者做 LRU Cache 等。这里也不再展开讨论了。

memcached 内部数据结构就是采用了HashMap类似的思想来实现的，有兴趣的可以参考资料 8,9，10。

为了不使这篇文章过长，因此将 ConcurrentHashMap 的原理放到下篇讲。需要说明的是，尽管 ConcurrentHashMap 与 HashMap 的名称有些渊源，而且实现原理有些相似，但是为了更好的支持并发，ConcurrentHashMap 在内部也有一些比较大的调整，这个在下篇会具体介绍。

ConcurrentHashMap

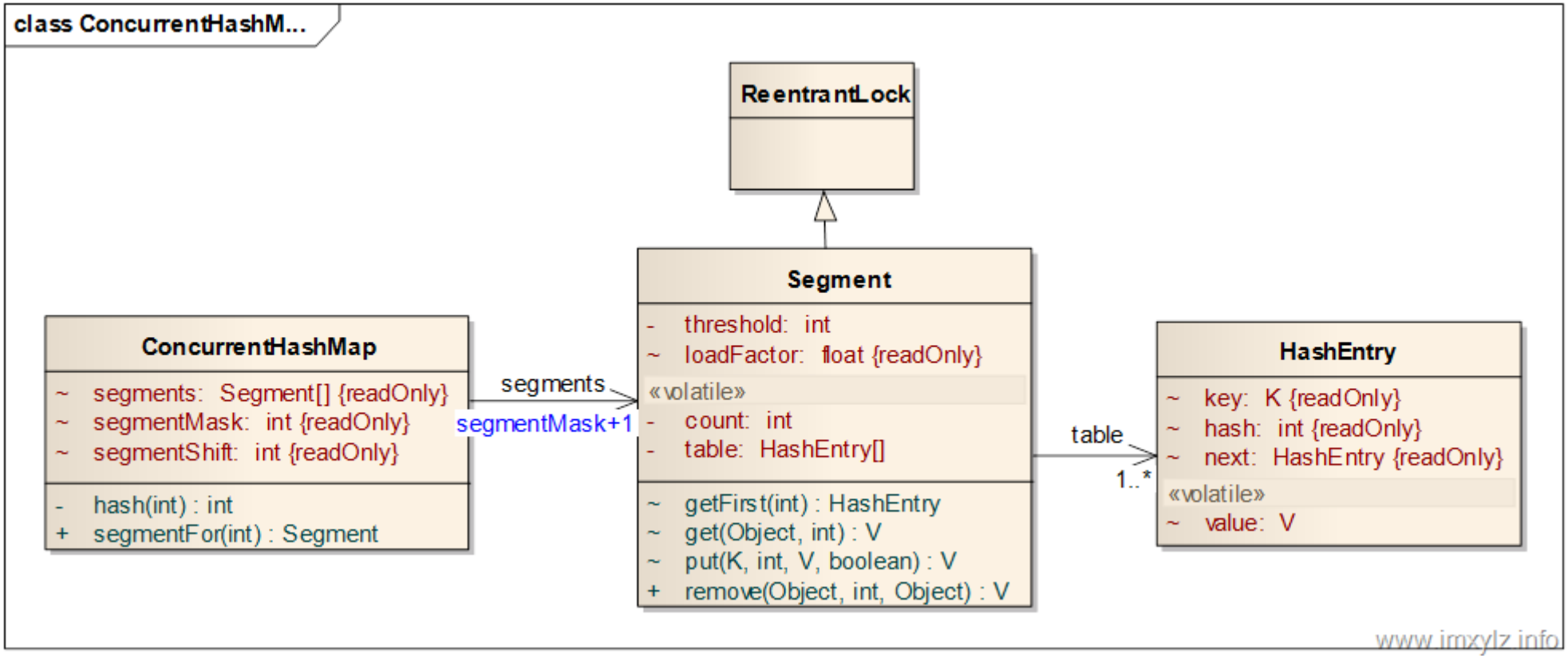
深入浅出 Java Concurrency (18): 并发容器 part 3 ConcurrentMap (3)

在上一篇中介绍了 HashMap 的原理，这一节是 ConcurrentMap 的最后一节，所以会完整的介绍 ConcurrentHashMap 的实现。

ConcurrentHashMap 原理

在读写锁章节部分介绍过一种是用读写锁实现Map的方法。此种方法看起来可以实现Map响应的功能，而且吞吐量也应该不错。但是通过前面对读写锁原理的分析后知道，读写锁的适合场景是读操作>>写操作，也就是读操作应该占据大部分操作，另外读写锁存在一个很严重的问题是读写操作不能同时发生。要想解决读写同时进行问题（至少不同元素的读写分离），那么就只能将锁拆分，不同的元素拥有不同的锁，这种技术就是“锁分离”技术。

默认情况下 ConcurrentHashMap 是用了 16 个类似 HashMap 的结构，其中每一个 HashMap 拥有一个独占锁。也就是说最终的效果就是通过某种 Hash 算法，将任何一个元素均匀的映射到某个 HashMap 的 Map.Entry 上面，而对某个一个元素的操作就集中在其分布的 HashMap 上，与其它 HashMap 无关。这样就支持最多 16 个并发的写操作。



上图就是 ConcurrentHashMap 的类图。参考上面的说明和 HashMap 的原理分析，可以看到 ConcurrentHashMap 将整个对象列表分为 segmentMask+1 个片段（Segment）。其中每一个片段是一个类似于 HashMap 的结构，它有一个 HashEntry 的数组，数组的每一项又是一个链表，通过 HashEntry 的 next 引用串联起来。

这个类图上面的数据结构的定义非常有学问，接下来会一个个有针对性的分析。

首先如何从 ConcurrentHashMap 定位到 HashEntry。在 HashMap 的原理分析部分说过，对于一个 Hash 的数据结构来说，为了减少浪费的空间和快速定位数据，那么就需要数据在 Hash 上的分布比较均匀。对于一次 Map 的查找来说，首先就需要定位到 Segment，然后从过 Segment 定位到 HashEntry 链表，最后才是通过遍历链表得到需要的元素。

在不讨论并发的前提下先来讨论如何定位到HashEntry的。在ConcurrentHashMap中是通过hash(key.hashCode())和segmentFor(hash)来得到Segment的。清单 1 描述了如何定位Segment的过程。其中hash(int)是将key的hashCode进行二次编码，使之能够在segmentMask+1 个Segment上均匀分布（默认是 16 个）。可以看到的是这里和HashMap还是有点不同的，这里采用的算法叫Wang/Jenkins hash，有兴趣的可以参考资料 1和参考资料 2。总之它的目的就是使元素能够均匀的分布在不同的Segment上，这样才能够支持最多segmentMask+1 个并发，这里segmentMask+1 是segments的大小。

清单 1 定位 Segment

```
private static int hash(int h) {
```

```
        // Spread bits to regularize both segment and index locations,
        // using variant of single-word Wang/Jenkins hash.
        h += (h <<  15) ^ 0xffffcd7d;
        h ^= (h >>> 10);
        h += (h <<   3);
        h ^= (h >>>  6);
        h += (h <<   2) + (h << 14);
        return h ^ (h >>> 16);
    }
    final Segment<K,V> segmentFor(int hash) {
        return segments[(hash >>> segmentShift) & segmentMask];
    }
```

显然在不能够对 Segment 扩容的情况下，segments 的大小就应该是固定的。所以在 ConcurrentHashMap 中 segments/segmentMask/segmentShift 都是常量，一旦初始化后就不能被再次修改，其中 segmentShift 是查找 Segment 的一个常量偏移量。

有了 Segment 以后再定位 HashEntry 就和 HashMap 中定位 HashEntry 一样了，先将 hash 值与 Segment 中 HashEntry 的大小减 1 进行与操作定位到 HashEntry 链表，然后遍历链表就可以完成相应的操作了。

能够定位元素以后ConcurrentHashMap就已经具有了HashMap的功能了，现在要解决的就是如何并发的问题。要解决并发问题，加锁是必不可免的。再回头看Segment的类图，可以看到Segment除了有一个volatile类型的元素大小count外，Segment还是集成自ReentrantLock的。另外在前面的原子操作和锁机制中介绍过，要想最大限度的支持并发，那么能够利用的思路就是尽量读操作不加锁，写操作不加锁。如果是读操作不加锁，写操作加锁，对于竞争资源来说就需要定义为volatile类型的。volatile类型能够保证happens-before法则，所以volatile能够近似保证正确性的情况下最大程度的降低加锁带来的影响，同时还与写操作的锁不产生冲突。

同时为了防止在遍历 HashEntry 的时候被破坏，那么对于 HashEntry 的数据结构来说，除了 value 之外其他属性就应该是常量，否则不可避免的会得到 ConcurrentModificationException。这就是为什么 HashEntry 数据结构中 key,hash,next 是常量的原因(final 类型）。

有了上面的分析和条件后再来看 Segment 的 get/put/remove 就容易多了。

get 操作

清单 2 Segment 定位元素

```
    V get(Object key, int hash) {
        if (count != 0) { // read-volatile
            HashEntry<K,V> e = getFirst(hash);
            while (e != null) {
                if (e.hash == hash && key.equals(e.key)) {
```

```
                V v = e.value;
                if (v != null)
                    return v;
                return readValueUnderLock(e); // recheck
            }
            e = e.next;
        }
    }
    return null;
}
HashEntry<K,V> getFirst(int hash) {
    HashEntry<K,V>[] tab = table;
    return tab[hash & (tab.length - 1)];
}
V readValueUnderLock(HashEntry<K,V> e) {
    lock();
    try {
        return e.value;
    } finally {
        unlock();
    }
}
```

清单 2 描述的是 Segment 如何定位元素。首先判断 Segment 的大小 count>0，Segment 的大小描述的是 HashEntry 不为空(key 不为空)的个数。如果 Segment 中存在元素那么就通过 getFirst 定位到指定的 HashEntry 链表的头节点上，然后遍历此节点，一旦找到 key 对应的元素后就返回其对应的值。但是在清单 2 中可以看到拿到 HashEntry 的 value 后还进行了一次判断操作，如果为空还需要加锁再读取一次（readValueUnderLock）。为什么会有这样的操作？尽管 ConcurrentHashMap 不允许将 value 为 null 的值加入，但现在仍然能够读到一个为空的 value 就意味着此值对当前线程还不可见（这是因为 HashEntry 还没有完全构造完成就赋值导致的，后面还会谈到此机制）。

put 操作

清单 3 描述的是 Segment 的 put 操作。首先就需要加锁了，修改一个竞争资源肯定是要加锁的，这个毫无疑问。需要说明的是 Segment 集成的是 ReentrantLock，所以这里加的锁也就是独占锁，也就是说同一个 Segment 在同一时刻只有能一个 put 操作。

接下来来就是检查是否需要扩容，这和 HashMap 一样，如果需要的话就扩大一倍，同时进行 rehash 操作。

查找元素就和 get 操作是一样的，得到元素就直接修改其值就好了。这里 onlyIfAbsent 只是为了实现 ConcurrentMap 的 putIfAbsent 操作而已。需要说明以下几点：

- 如果找到 key 对于的 HashEntry 后直接修改就好了，如果找不到那么就需要构造一个新的 HashEntry 出来加到 hash 对于的 HashEntry 的头部，同时就的头部就加到新的头部后面。这是因为 HashEntry 的 next 是 final 类型的，所以只能修改头节点才能加元素加入链表中。
- 如果增加了新的操作后，就需要将 count+1 写回去。前面说过 count 是 volatile 类型，而读取操作没有加锁，所以只能把元素真正写回 Segment 中的时候才能修改 count 值，这个要放到整个操作的最后。
- 在将新的 HashEntry 写入 table 中时是通过构造函数来设置 value 值的，这意味对 table 的赋值可能在设置 value 之前，也就是说得到了一个半构造完的 HashEntry。这就是重排序可能引起的问题。所以在读取操作中，一旦读到了一个 value 为空的 value 是就需要加锁重新读取一次。为什么要加锁？加锁意味着前一个写操作的锁释放，也就是前一个锁的数据已经完成写完了了，根据 happens-before 法则，前一个写操作的结果对当前读线程就可见了。当然在 JDK 6.0 以后不一定存在此问题。
- 在 Segment 中 table 变量是 volatile 类型，多次读取 volatile 类型的开销要不非 volatile 开销要大，而且编译器也无法优化，所以在 put 操作中首先建立一个临时变量 tab 指向 table，多次读写 tab 的效率要比 volatile 类型的 table 要高，JVM 也能够对此进行优化。

清单 3 Segment 的 put 操作

```
V put(K key, int hash, V value, boolean onlyIfAbsent) {
    lock();
    try {
        int c = count;
        if (c++ > threshold) // ensure capacity
            rehash();
        HashEntry<K,V>[] tab = table;
        int index = hash & (tab.length - 1);
        HashEntry<K,V> first = tab[index];
        HashEntry<K,V> e = first;
        while (e != null && (e.hash != hash || !key.equals(e.key)))
            e = e.next;
        V oldValue;
        if (e != null) {
            oldValue = e.value;
            if (!onlyIfAbsent)
                e.value = value;
        }
        else {
```

```
            oldValue = null;
            ++modCount;
            tab[index] = new HashEntry<K,V>(key, hash, first, value);
            count = c; // write-volatile
        }
        return oldValue;
    } finally {
        unlock();
    }
}
```

remove 操作

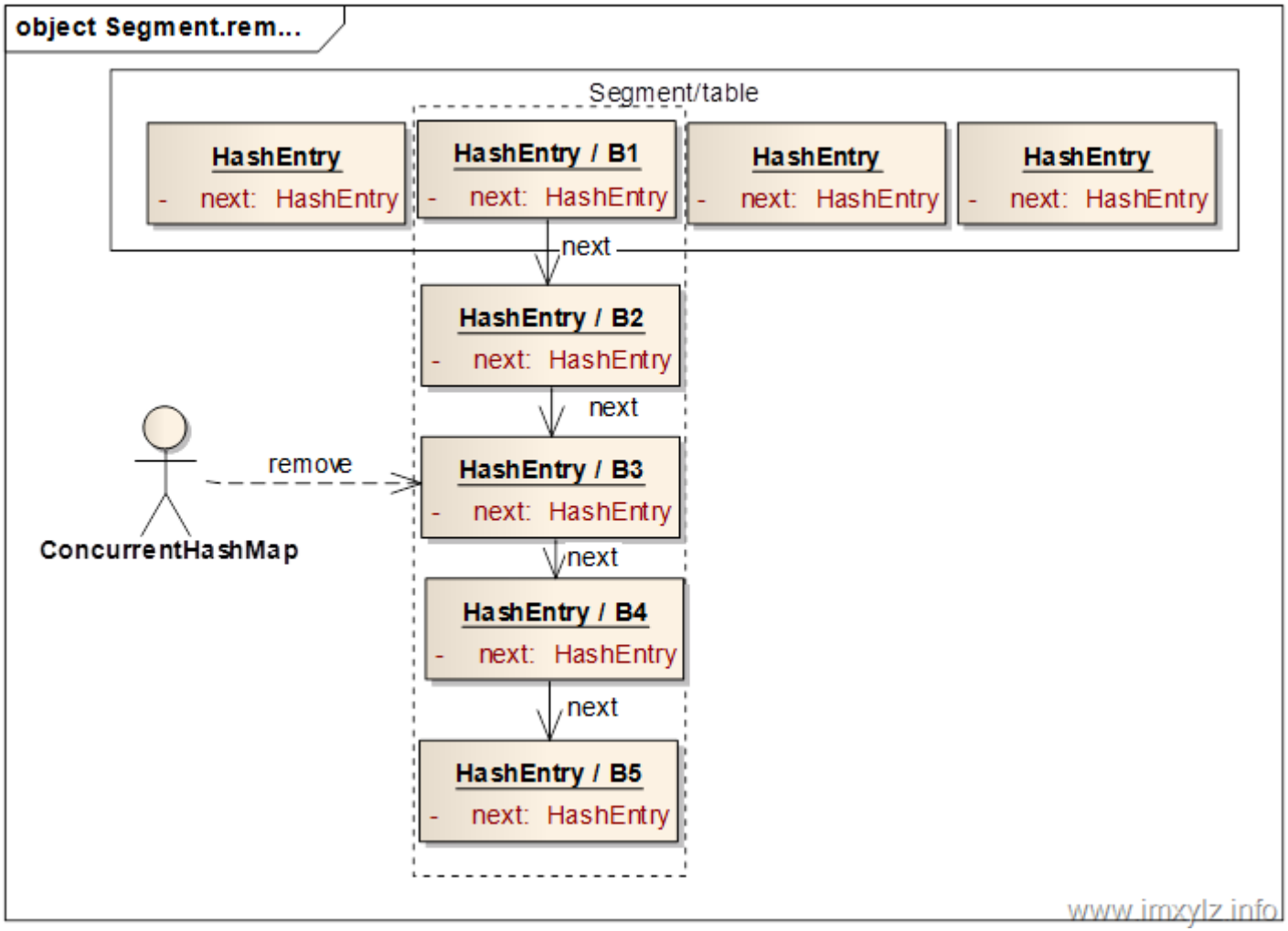
清单 4 描述了 Segment 删除一个元素的过程。同 put 一样，remove 也需要加锁，这是因为对 table 可能会有变更。由于 HashEntry 的 next 节点是 final 类型的，所以一旦删除链表中间一个元素，就需要将删除之前或者之后的元素重新加入新的链表。而 Segment 采用的是将删除元素之前的元素一个个重新加入删除之后的元素之前（也就是链表头结点）来完成新链表的构造。

清单 4 Segment 的 remove 操作

```
V remove(Object key, int hash, Object value) {
    lock();
    try {
        int c = count - 1;
        HashEntry<K,V>[] tab = table;
        int index = hash & (tab.length - 1);
        HashEntry<K,V> first = tab[index];
        HashEntry<K,V> e = first;
        while (e != null && (e.hash != hash || !key.equals(e.key)))
            e = e.next;
        V oldValue = null;
        if (e != null) {
            V v = e.value;
            if (value == null || value.equals(v)) {
                oldValue = v;
```

```
                // All entries following removed node can stay
                // in list, but all preceding ones need to be
                // cloned.
                ++modCount;
                HashEntry<K,V> newFirst = e.next;
                for (HashEntry<K,V> p = first; p != e; p = p.next)
                     newFirst = new HashEntry<K,V>(p.key, p.hash, newFirst, p.value);
                    tab[index] = newFirst;
                    count = c; // write-volatile
            }
        }
        return oldValue;
    } finally {
        unlock();
    }
}
```

下面的示意图描述了如何删除一个已经存在的元素的。假设我们要删除 B3 元素。首先定位到 B3 所在的 Segment，然后再定位到 Segment 的 table 中的 B1 元素，也就是 Bx 所在的链表。然后遍历链表找到 B3，找到之后就从头结点 B1 开始构建新的节点 B1（蓝色）加到 B4 的前面，继续 B1 后面的节点 B2 构造 B2（蓝色），加到由蓝色的 B1 和 B4 构成的新的链表。继续下去，直到遇到 B3 后终止，这样就构造出来一个新的链表 B2（蓝色）->B1（蓝色）->B4->B5，然后将此链表的头结点 B2（蓝色）设置到 Segment 的 table 中。这样就完成了元素 B3 的删除操作。需要说明的是，尽管就的链表仍然存在(B1->B2->B3->B4->B5)，但是由于没有引用指向此链表，所以此链表中无引用的（B1->B2->B3）最终会被 GC 回收掉。这样做的一个好处是，如果某个读操作在删除时已经定位到了旧的链表上，那么此操作仍然将能读到数据，只不过读取到的是旧数据而已，这在多线程里面是没有问题的。



除了对单个元素操作外，还有对全部的 Segment 的操作，比如 size()操作等。

size 操作

size 操作涉及到统计所有 Segment 的大小，这样就会遍历所有的 Segment，如果每次加锁就会导致整个 Map 都被锁住了，任何需要锁的操作都将无法进行。这里用到了一个比较巧妙的方案解决此问题。

在 Segment 中有一个变量 modCount，用来记录 Segment 结构变更的次数，结构变更包括增加元素和删除元素，每增加一个元素操作就+1，每进行一次删除操作+1，每进行一次清空操作(clear)就+1。也就是说每次涉及到元素个数变更的操作 modCount 都会+1，而且一直是增大的，不会减小。

遍历两次 ConcurrentHashMap 中的 segments，每次遍历是记录每一个 Segment 的 modCount，比较两次遍历的 modCount 值的和是否相同，如果相同就返回在遍历过程中获取的 Segment 的 count 的和，也就是所有元素的个数。如果不相同就重复再做一次。重复一次还不相同就将所有 Segment 锁住，一个一个的获取其大小(count)，最后将这些 count 加起来得到总的大小。当然了最后需要将锁一一释放。清单 5 描述了这个过程。

这里有一个比较高级的话题是为什么在读取 modCount 的时候总是先要读取 count 一下。为什么不是先读取 modCount 然后再读取 count 的呢？也就是说下面的两条语句能否交换下顺序？

```
sum += segments[i].count;
mcsum += mc[i] = segments[i].modCount;
```

答案是不能！为什么？这是因为 modCount 总是在加锁的情况下才发生变化，所以不会发生多线程同时修改的情况，也就是没必要时 volatile 类型。另外总是在 count 修改的情况下修改 modCount，而 count 是一个 volatile 变量。于是这里就充分利用了 volatile 的特性。

根据happens-before法则，第（3）条：对volatile字段的写入操作happens-before于每一个后续的同一个字段的读操作。也就是说一个操作C在volatile字段的写操作之后，那么volatile写操作之前的所有操作都对此操作C可见。所以修改modCount总是在修改count之前，也就是说如果读取到了一个count的值，那么在count变化之前的modCount也就能够读取到，换句话说就是如果看到了count值的变化，那么就一定看到了modCount值的变化。而如果上面两条语句交换下顺序就无法保证这个结果一定存在了。

在 ConcurrentHashMap.containsValue 中，可以看到每次遍历 segments 时都会执行 int c = segments[i].count;，但是接下来的语句中又不用此变量 c，尽管如此 JVM 仍然不能将此语句优化掉，因为这是一个 volatile 字段的读取操作，它保证了一些列操作的 happens-before 顺序，所以是至关重要的。在这里可以看到： ***ConcurrentHashMap 将 volatile 发挥到了极致！*** 另外 isEmpty 操作于 size 操作类似，不再累述。

清单 5 ConcurrentHashMap 的 size 操作

```
public int size() {
    final Segment<K,V>[] segments = this.segments;
    long sum = 0;
    long check = 0;
    int[] mc = new int[segments.length];
    // Try a few times to get accurate count. On failure due to
    // continuous async changes in table, resort to locking.
    for (int k = 0; k < RETRIES_BEFORE_LOCK; ++k) {
        check = 0;
        sum = 0;
        int mcsum = 0;
        for (int i = 0; i < segments.length; ++i) {
            sum += segments[i].count;
            mcsum += mc[i] = segments[i].modCount;
        }
```

```
            if (mcsum != 0) {
                for (int i = 0; i < segments.length; ++i) {
                    check += segments[i].count;
                    if (mc[i] != segments[i].modCount) {
                        check = -1; // force retry
                        break;
                    }
                }
            }
            if (check == sum)
                break;
        }
        if (check != sum) { // Resort to locking all segments
            sum = 0;
            for (int i = 0; i < segments.length; ++i) segments[i].lock();
            for (int i = 0; i < segments.length; ++i) sum += segments[i].count;
            for (int i = 0; i < segments.length; ++i) segments[i].unlock();
        }
        if (sum > Integer.MAX_VALUE)
            return Integer.MAX_VALUE;
        else
            return (int)sum;
    }
```

ConcurrentSkipListMap/Set

本来打算介绍下ConcurrentSkipListMap的，结果打开源码一看，彻底放弃了。那里面的数据结构和算法我估计研究一周也未必能够完全弄懂。很久以前我看TreeMap的时候就头大，想想那些复杂的“红黑二叉树”我头都大了。这些都归咎于从前没有好好学习《数据结构和算法》，现在再回头看这些复杂的算法感觉非常头疼，为了减少脑细胞的死亡，暂且还是不要惹这些“玩意儿”。有兴趣的可以看看参考资料 4 中对TreeMap的介绍。

Queue

深入浅出 Java Concurrency (19): 并发容器 part 4 并发队列与 Queue 简介

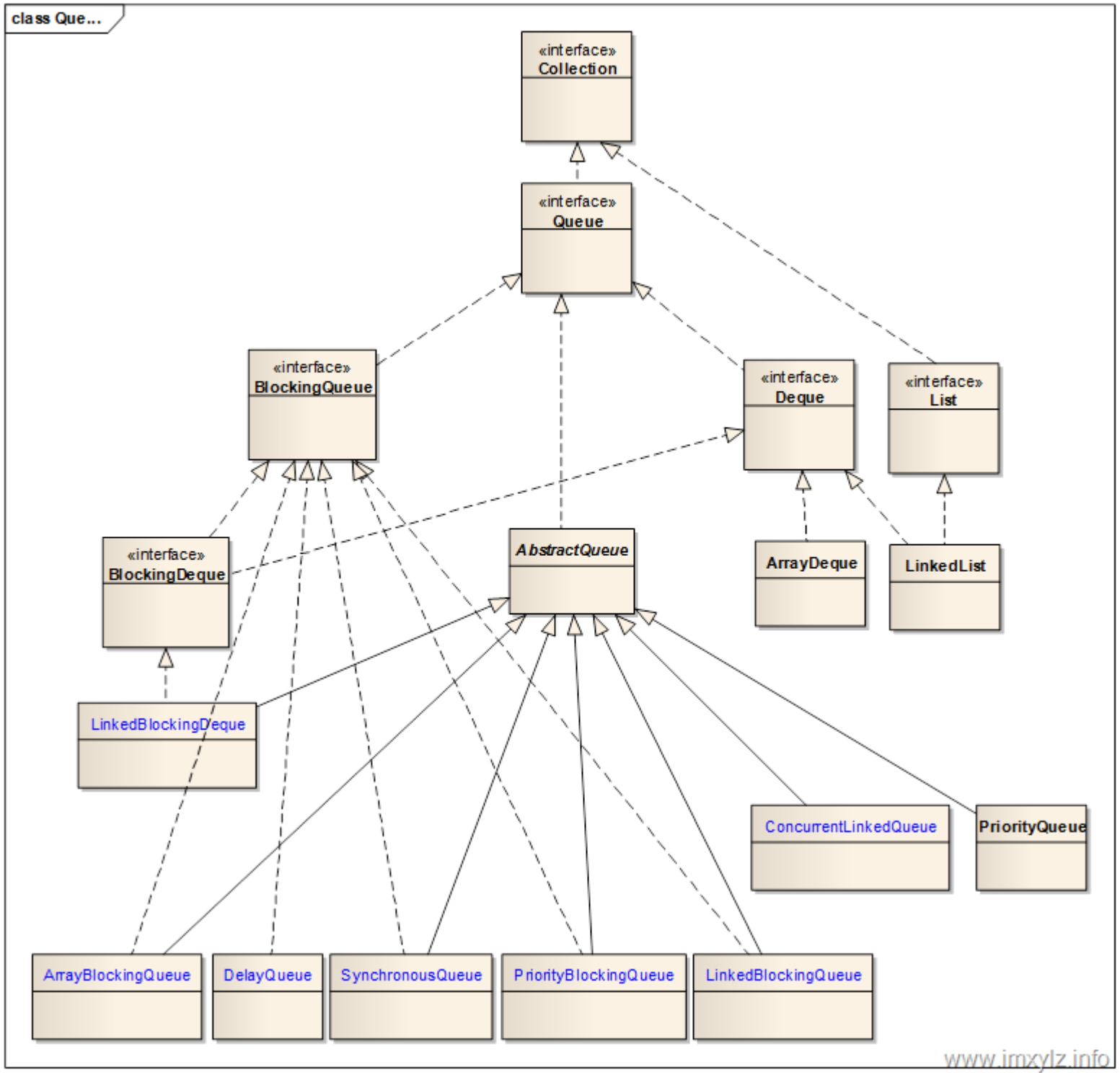
Queue 是 JDK 5 以后引入的新的集合类，它属于 Java Collections Framework 的成员，在 Collection 集合中和 List/Set 是同一级别的接口。通常来讲 Queue 描述的是一种 FIFO 的队列，当然不全都是，比如 PriorityQueue 是按照优先级的顺序（或者说是自然顺序，借助于 Comparator 接口）。

下图描述了 Java Collections Framework 中 Queue 的整个家族体系。

对于 Queue 而言是在 Collection 的基础上增加了 offer/remove/poll/element/peek 方法，另外重新定义了 add 方法。对于这六个方法，有不同的定义。

	抛出异常	返回特殊值	操作描述
插入	add(e)	offer(e)	将元素加入到队列尾部
移除	remove()	poll()	移除队列头部的元素
检查	element()	peek()	返回队列头部的元素而不移除此元素

特别说明的是对于 Queue 而言，规范并没有规定是线程安全的，为了解决这个问题，引入了可阻塞的队列 BlockingQueue。对于 BlockingQueue 而言所有操作的是线程安全的，并且队列的操作可以被阻塞，直到满足某种条件。Queue 的另一个子接口 Deque 描述的是一个双向的队列。与 Queue 不同的是，Deque 允许在队列的头部增加元素和在队列的尾部删除元素。也就是说 Deque 是一个双向队列。二者功能都有的队列就是 BlockingDeque，这种阻塞队列允许在队列的头和尾部分别操作元素，应该说是 Queue 中功能最强大的实现。



在 JDK 5 之前 LinkedList 就已经存在，而且本身实现都是一种双向队列。所以到了 JDK 5 以后就将 LinkedList 同时实现 Deque 接口，这样 LinkedList 就又属于 Queue 的一部分了。

通常情况下 Queue 都是靠链表结构实现的，但是链表意味着有一些额外的引用开销，如果是双向链表开销就更大了。所以为了节省内存，一种方式就是使用固定大小的数组来实现队列。在这种情况下队列的大小是固定，元素的遍历通过数组的索引进行，很显然这是一种双向链表的模型。ArrayDeque 就是这样一种实现。

另外 ArrayBlockingQueue 也是一种数组实现的队列，但是却没有改造成双向，仅仅实现了 BlockingQueue 的模型。理论上和 ArrayDeque 一样也应该容易改造成双向的实现。

PriorityQueue 和 PriorityBlockingQueue 实现了一种排序的队列模型。这很类似与 SortedSet，通过队列的 Comparator 接口或者 Comparable 元素来排序元素。这种情况下元素在队列中的出入就不是按照 FIFO 的形式，而是根据比较后的自然顺序来进行。

CocurrentLinkedQueue 是一种线程安全却非阻塞的 FIFO 队列，这种队列通常实现起来比较简单，但是却很有效。在接下来的章节会详细的描述它。

SynchronousQueue 是一种特别的 BlockingQueue，它只是把一个 add/offer 操作的元素直接移交给 remove/take 操作。也就是说它本身不会缓存任何元素，所以严格意义上说来讲并不是一种真正的队列。此队列维护一个线程列表，这些线程等待从队列中加入元素或者移除元素。简单的说，至少有一个 remove/take 操作时 add/offer 操作才能成功，同样至少有一个 add/offer 操作时 remove/take 操作才能成功。这是一种双向等待的队列模型，出队列等待加入等列，而入队列又等待出队列。这种队列的好处在于能够最大线程的保持吞吐量却又是线程安全的。所以对于一个需要快速处理的任务队列，SynchronousQueue 是一个不错的选择。

BlockingQueue 还有一种实现 DelayQueue，这种实现允许每一个元素(Delayed)带有一个延时时间，当调用 take/poll 的时候会检测队列头元素这个时间是否<=0，如果满足就是说已经超时了，那么此元素就可以被移除了，否则就会等待。特别说明的是这个头元素应该是最先被超时的元素（这个时间是绝对时间）。这个类设计很巧妙，被用于 ScheduledFutureTask 来进行定时操作。希望后面会开辟一个章节讲讲这里面的想法。实在不行在讲线程池部分肯定会提到这个。

ConcurrentLinkedQueue

深入浅出 Java Concurrency (20): 并发容器 part 5 ConcurrentLinkedQueue

ConcurrentLinkedQueue 是 Queue 的一个线程安全实现。先来看一段文档说明。

一个基于链接节点的无界线程安全队列。此队列按照 FIFO（先进先出）原则对元素进行排序。队列的头部 是队列中时间最长的元素。队列的尾部 是队列中时间最短的元素。新的元素插入到队列的尾部，队列获取操作从队列头部获得元素。当多个线程共享访问一个公共 collection 时，ConcurrentLinkedQueue 是一个恰当的选择。此队列不允许使用 null 元素。

由于 ConcurrentLinkedQueue 只是简单的实现了一个队列 Queue，因此从 API 的角度讲，没有多少值的介绍，使用起来也很简单，和前面遇到的所有 FIFO 队列都类似。出队列只能操作头节点，入队列只能操作尾节点，任意节点操作就需要遍历完整的队列。

重点放在解释 ConcurrentLinkedQueue 的原理和实现上。

在继续探讨之前，结合前面线程安全的相关知识，我来分析设计一个线程安全的队列哪几种方法。

第一种：使用 synchronized 同步队列，就像 Vector 或者 Collections.synchronizedList/Collection 那样。显然这不是一个好的并发队列，这会导致吞吐量急剧下降。

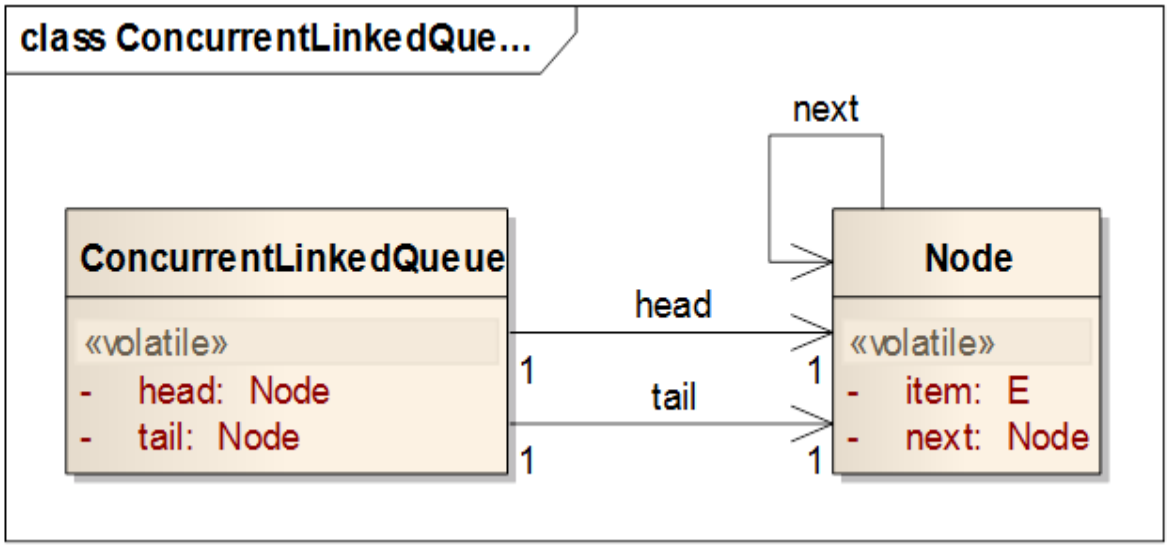
第二种：使用 Lock。一种好的实现方式是使用 ReentrantReadWriteLock 来代替 ReentrantLock 提高读取的吞吐量。但是显然 ReentrantReadWriteLock 的实现更为复杂，而且更容易导致出现问题，另外也不是一种通用的实现方式，因为 ReentrantReadWriteLock 适合哪种读取量远远大于写入量的场合。当然了 ReentrantLock 是一种很好的实现，结合 Condition 能够很方便的实现阻塞功能，这在后面介绍 BlockingQueue 的时候会具体分析。

第三种：使用 CAS 操作。尽管 Lock 的实现也用到了 CAS 操作，但是毕竟是间接操作，而且会导致线程挂起。一个好的并发队列就是采用某种非阻塞算法来取得最大的吞吐量。

ConcurrentLinkedQueue采用的就是第三种策略。它采用了参考资料 1 中的算法。

在锁机制中谈到过，要使用非阻塞算法来完成队列操作，那么就需要一种“循环尝试”的动作，就是循环操作队列，直到成功为止，失败就会再次尝试。这在前面的章节中多次介绍过。

针对各种功能深入分析。



在开始之前先介绍下 ConcurrentLinkedQueue 的数据结构。

上面的数据结构中，ConcurrentLinkedQueue 只有头结点、尾节点两个元素，而对于一个节点 Node 而言除了保存队列元素 item 外，还有一个指向下一个节点的引用 next。 看起来整个数据结构还是比较简单的。但是也有几点是需要说明：

1. 所有结构（head/tail/item/next）都是 volatile 类型。 这是因为 ConcurrentLinkedQueue 是非阻塞的，所以只有 volatile 才能使变量的写操作对后续读操作是可见的（这个是有 happens-before 法则保证的）。同样也不会导致指令的重排序。

2. 所有结构的操作都带有原子操作，这是由 AtomicReferenceFieldUpdater 保证的，这在原子操作中介绍过。它能保证需要的时候对变量的修改操作是原子的。

3. 由于队列中任何一个节点（Node）只有下一个节点的引用，所以这个队列是单向的，根据 FIFO 特性，也就是说出队列在头部(head)，入队列在尾部(tail)。头部保存有进入队列最长时间的元素，尾部是最近进入的元素。

4. 没有对队列长度进行计数，所以队列的长度是无限的，同时获取队列的长度的时间不是固定的，这需要遍历整个队列，并且这个计数也可能是不精确的。

5. 初始情况下队列头和队列尾都指向一个空节点，但是非 null，这是为了方便操作，不需要每次去判断 head/tail 是否为空。但是 head 却不作为存取元素的节点，tail 在不等于 head 情况下保存一个节点元素。也就是说 head.item 这个应该一直是空，但是 tail.item 却不一定是空（如果 head!=tail，那么 tail.item!=null）。

对于第 5 点，可以从 ConcurrentLinkedQueue 的初始化中看到。这种头结点也叫“伪节点”，也就是说它不是真正的节点，只是一标识，就像 c 中的字符数组后面的\0 以后，只是用来标识结束，并不是真正字符数组的一部分。

```
private transient volatile Node<E> head = new Node<E>(null, null);
private transient volatile Node<E> tail = head;
```

有了上述 5 点再来解释相关 API 操作就容易多了。

在上一节中列出了 add/offer/remove/poll/element/peek 等价方法的区别，所以这里就不再重复了。

清单 1 入队列操作

```
public boolean offer(E e) {
    if (e == null) throw new NullPointerException();
    Node<E> n = new Node<E>(e, null);
    for (;;) {
        Node<E> t = tail;
        Node<E> s = t.getNext();
        if (t == tail) {
            if (s == null) {
                if (t.casNext(s, n)) {
```

```
                casTail(t, n);
                return true;
            }
        } else {
            casTail(t, s);
        }
    }
  }
}
```

清单 1 描述的是入队列的过程。整个过程是这样的。

1. 获取尾节点 t，以及尾节点的下一个节点 s。如果尾节点没有被别人修改，也就是 t==tail，进行 2，否则进行 1。
2. 如果 s 不为空，也就是说此时尾节点后面还有元素，那么就需要把尾节点往后移，进行 1。否则进行 3。
3. 修改尾节点的下一个节点为新节点，如果成功就修改尾节点，返回 true。否则进行 1。

从操作 3 中可以看到是先修改尾节点的下一个节点，然后才修改尾节点位置的，所以这才有操作 2 中为什么获取到的尾节点的下一个节点不为空的原因。特别需要说明的是，对尾节点的 tail 的操作需要换成临时变量 t 和 s，一方面是为了去掉 volatile 变量的可变性，另一方面是为了减少 volatile 的性能影响。

清单 2 描述的出队列的过程，这个过程和入队列相似，有点意思。头结点是为了标识队列起始，也为了减少空指针的比较，所以头结点总是一个 item 为 null 的非 null 节点。也就是说 head!=null 并且 head.item==null 总是成立。所以实际上获取的是 head.next，一旦将头结点 head 设置为 head.next 成功就将新 head 的 item 设置为 null。至于以前就的头结点 h，h.item=null 并且 h.next 为新的 head，但是由于没有对 h 的引用，所以最终会被 GC 回收。这就是整个出队列的过程。

清单 2 出队列操作

```
public E poll() {
    for (;;) {
        Node<E> h = head;
        Node<E> t = tail;
        Node<E> first = h.getNext();
        if (h == head) {
            if (h == t) {
                if (first == null) return null;
                else casTail(t, first);
            } else if (casHead(h, first)) {
                E item = first.getItem();
```

```
                if (item != null) { first.setItem(null); return item; }
                // else skip over deleted item, continue loop,
            }
        }
    }
}
```

另外对于清单 3 描述的获取队列大小的过程，由于没有一个计数器来对队列大小计数，所以获取队列的大小只能通过从头到尾完整的遍历队列，显然这个代价是很大的。所以通常情况下 ConcurrentLinkedQueue 需要和一个 AtomicInteger 搭配才能获取队列大小。后面介绍的 BlockingQueue 正是使用了这种思想。

清单 3 遍历队列大小

```
public int size() {
    int count = 0;
    for (Node<E> p = first(); p != null; p = p.getNext()) {
        if (p.getItem() != null) {
            // Collections.size() spec says to max out
            if (++count == Integer.MAX_VALUE)
                break;
        }
    }
    return count;
}
```

BlockingQueue

深入浅出 Java Concurrency (21): 并发容器 part 6 可阻塞的 BlockingQueue (1)

在《并发容器 part 4 并发队列与Queue简介》节中的类图中可以看到，对于Queue来说，BlockingQueue是主要的线程安全版本。这是一个可阻塞的版本，也就是允许添加/删除元素被阻塞，直到成功为止。

BlockingQueue 相对于 Queue 而言增加了两个操作：put/take。下面是一张整理的表格。

	抛出异常	特殊值	阻塞	超时	描述
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)	将元素加入到队列尾部
移除	remove()	poll()	take()	poll(time,unit)	移除队列列头的元素
检查	element()	peek()			读取而不删除队列头部的元素

看似简单的 API，非常有用。这在控制队列的并发上非常有好处。既然加入队列和移除队列能够被阻塞，这在实现生产者-消费者模型上就简单多了。

清单 1 是生产者-消费者模型的一个例子。这个例子是一个真实的场景。服务端（ICE 服务）接受客户端的请求(accept)，请求计算此人的好友生日，然后将计算的结果存取缓存中（Memcache）中。在这个例子中采用了 ExecutorService 实现多线程的功能，尽可能的提高吞吐量，这个在后面线程池的部分会详细说明。目前就可以理解为 new Thread(r).start()就可以了。另外这里阻塞队列使用的是 LinkedBlockingQueue。

清单 1 一个生产者-消费者例子

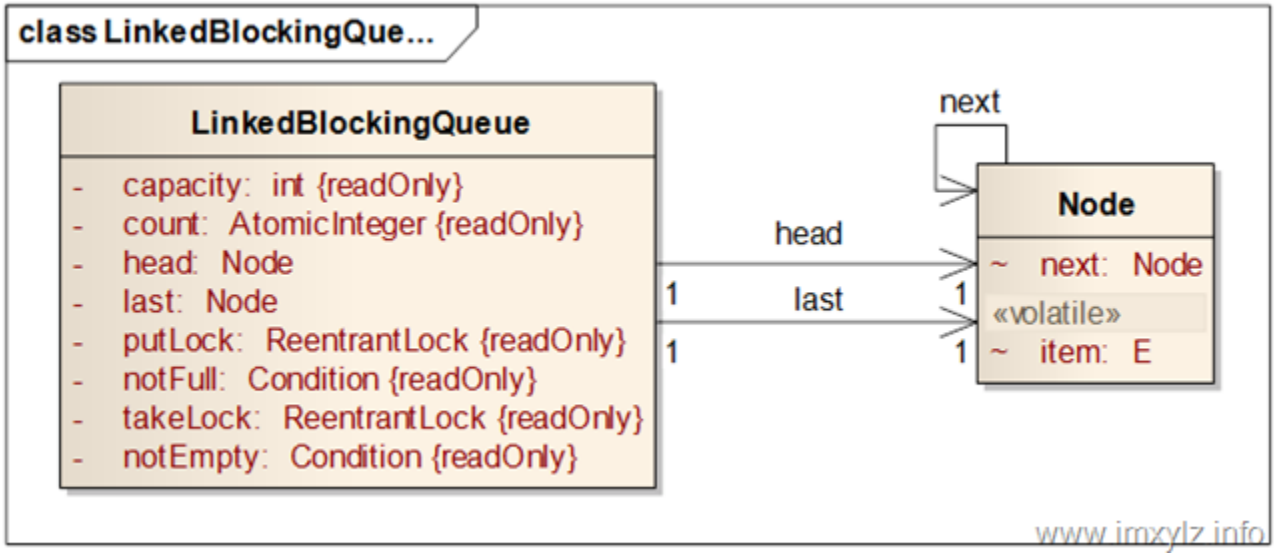
```
package xylz.study.concurrency;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingDeque;
public class BirthdayService {
    final int workerNumber;
    final Worker[] workers;
    final ExecutorService threadPool;
    static volatile boolean running = true;
    public BirthdayService(int workerNumber, int capacity) {
        if (workerNumber <= 0) throw new IllegalArgumentException();
        this.workerNumber = workerNumber;
        workers = new Worker[workerNumber];
        for (int i = 0; i < workerNumber; i++) {
            workers[i] = new Worker(capacity);
        }
        //
        boolean b = running;// kill the resorting
        threadPool = Executors.newFixedThreadPool(workerNumber);
        for (Worker w : workers) {
            threadPool.submit(w);
```

```
    }
}
Worker getWorker(int id) {
    return workers[id % workerNumber];
}
class Worker implements Runnable {
    final BlockingQueue<Integer> queue;
    public Worker(int capacity) {
        queue = new LinkedBlockingQueue<Integer>(capacity);
    }
    public void run() {
        while (true) {
            try {
                consume(queue.take());
            } catch (InterruptedException e) {
                return;
            }
        }
    }
    void put(int id) {
        try {
            queue.put(id);
        } catch (InterruptedException e) {
            return;
        }
    }
}
public void accept(int id) {
    //accept client request
    getWorker(id).put(id);
}
protected void consume(int id) {
```

```
            //do the work
            //get the list of friends and save the birthday to cache
        }
    }
```

在清单 1 中可以看到不管是 put()还是 get()，都抛出了一个 InterruptedException。我们就从这里开始，为什么会抛出这个异常。

上一节中提到实现一个并发队列有三种方式。显然只有第二种 Lock 才能实现阻塞队列。在锁机制中提到过，Lock结合Condition就可以实现线程的阻塞，这在锁机制部分的很多工具中都详细介绍过，而接下来要介绍的LinkedBlockingQueue就是采用这种方式。



对比ConcurrentLinkedQueue的结构图，LinkedBlockingQueue多了两个ReentrantLock和两个Condition以及用于计数的AtomicInteger，显然这会导致LinkedBlockingQueue的实现有点复杂。对照此结构，有以下几点说明：

1. 但是整体上讲，LinkedBlockingQueue 和 ConcurrentLinkedQueue 的结构类似，都是采用头尾节点，每个节点指向下一个节点的结构，这表示它们在操作上应该类似。
2. LinkedBlockingQueue 引入了原子计数器 count，这意味着获取队列大小 size()已经是常量时间了，不再需要遍历队列。每次队列长度有变更时只需要修改 count 即可。
3. 有了修改 Node 指向有了锁，所以不需要 volatile 特性了。既然有了锁 Node 的 item 为什么需要 volatile 在后面会详细分析，暂且不表。
4. 引入了两个锁，一个入队列锁，一个出队列锁。当然同时有一个队列不满的 Condition 和一个队列不空的 Condition。其实参照锁机制前面介绍过的生产者-消费者模型就知道，入队列就代表生产者，出队列就代表消费者。为什么需要两个锁？一个锁行不行？其实一个锁完全可以，但是一个锁意味着入队列和出队列同时只能有一个在进行，另一个必须等待其释放锁。而从 ConcurrentLinkedQueue 的实现原理来看，事实上 head 和 last (ConcurrentLinkedQueue 中是 tail)是分离的，互相独立的，这意味着入队列实际上是不会修改出队列的数据的，同时出队列也不会修改入队列，也就是说这两个操作是互不干扰的。更通俗的将，这个锁相当于两个写入锁，入队列是一种写操作，操作 head，出队列是一种写操作，操作 tail。可见它们是无关的。但是并非完全无关，后面详细分析。

在没有揭示入队列和出队列过程前，暂且猜测下实现原理。

根据前面学到的锁机制原理结合 ConcurrentLinkedQueue 的原理，入队列的阻塞过程大概是这样的：

1. 获取入队列的锁 putLock，检测队列大小，如果队列已满，那么就挂起线程，等待队列不满信号 notFull 的唤醒。
2. 将元素加入到队列尾部，同时修改队列尾部引用 last。
3. 队列大小加 1。

4.释放锁 putLock。

5.唤醒 notEmpty 线程（如果有挂起的出队列线程），告诉消费者，已经有了新的产品。

对比入队列，出队列的阻塞过程大概是这样的：

1.获取出队列的锁 takeLock，检测队列大小，如果队列为空，那么就挂起线程，等待队列不为空 notEmpty 的唤醒。

2.将元素从头部移除，同时修改队列头部引用 head。

3.队列大小减 1。

4.释放锁 takeLock。

5.唤醒 notFull 线程（如果有挂起的入队列线程），告诉生产者，现在还有空闲的空间。

下面来验证上面的过程。

入队列过程（put/offer）

清单 2 阻塞的入队列过程

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        try {
            while (count.get() == capacity)
                notFull.await();
        } catch (InterruptedException ie) {
            notFull.signal(); // propagate to a non-interrupted thread
            throw ie;
        }
        insert(e);
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
```

```
    if (c == 0)
        signalNotEmpty();
}
```

清单 2 描述的是入队列的阻塞过程。可以看到和上面描述的入队列的过程基本相同。但是也有以下几个问题：

1.如果在入队列的时候线程被中断，那么就需要发出一个 notFull 的信号，表示下一个入队列的线程能够被唤醒（如果阻塞的话）。

2.入队列成功后如果队列不满需要补一个 notFull 的信号。为什么？队列不满的时候其它入队列的阻塞线程难道不知道么？有可能。这是因为为了减少上下文切换的次数，每次唤醒一个线程（不管是入队列还是出队列）都是只随机唤醒一个(notify)，而不是唤醒所有的（notifyall()）。这会导致其它阻塞的入队列线程不能够即使处理队列不满的情况。

3.如果队列不为空并且可能有一个元素的话就唤醒一个出队列线程。这么做说明之前队列一定为空，因为在加入队列之后队列最多只能为 1，那么说明未加入之前是 0，那么就可能有被阻塞的出队列线程，所以就唤醒一个出队列线程。特别说明的是为什么使用一个临时变量 c，而不用 count。这是因为读取一个 count 的开销比读取一个临时一个变量大，而此处 c 又能够完成确认队列最多只有一个元素的判断。首先 c 默认为-1，如果加入队列后获取原子计数器的结果为 0，说明之前队列为空，不可能消费（出队列），也不可能入队列，因为此时锁还在当前线程上，那么加入一个后队列就不为空了，所以就可以安全的唤醒一个消费（出对立）线程。

4.入队列的过程允许被中断，所以总是抛出 InterruptedException 异常。

针对第 2 点，特别补充说明下。本来这属于锁机制中条件队列的范围，由于没有应用场景，所以当时没有提。

前面提高 notifyall 总是比 notify 更可靠，因为 notify 可能丢失通知，为什么不适用 notifyall 呢？

先解释下 notify 丢失通知的问题。

notify 丢失通知问题

假设线程 A 因为某种条件在条件队列中等待，同时线程 B 因为另外一种条件在同一个条件队列中等待，也就是说线程 A/B 都被同一个 Conditon.await()挂起，但是等待的条件不同。现在假设线程 B 的线程被满足，线程 C 执行一个 notify 操作，此时 JVM 从 Conditon.await()的多个线程（A/B）中随机挑选一个唤醒，不幸的是唤醒了 A。此时 A 的条件不满足，于是 A 继续挂起。而此时 B 仍然在傻傻的等待被唤醒的信号。也就是说本来给 B 的通知却被一个无关的线程持有了，真正需要通知的线程 B 却没有得到通知，而 B 仍然在等待一个已经发生过的通知。

如果使用 notifyall，则能够避免此问题。notifyall 会唤醒所有正在等待的线程，线程 C 发出的通知线程 A 同样能够收到，但是由于对于 A 没用，所以 A 继续挂起，而线程 B 也收到了此通知，于是线程 B 正常被唤醒。

既然 notifyall 能够解决单一 notify 丢失通知的问题，那么为什么不总是使用 notifyall 替换 notify 呢？

假设有 N 个线程在条件队列中等待，调用 notifyall 会唤醒所有线程，然后这 N 个线程竞争同一个锁，最多只有一个线程能够得到锁，于是其它线程又回到挂起状态。这意味每一次唤醒操作可能带来大量的上下文切换（如果 N 比较大的话），同时有大量的竞争锁的请求。这对于频繁的唤醒操作而言性能上可能是一种灾难。

如果说总是只有一个线程被唤醒后能够拿到锁，那么为什么不使用 notify 呢？所以某些情况下使用 notify 的性能是要高于 notifyall 的。

如果满足下面的条件，可以使用单一的 notify 取代 notifyall 操作：

相同的等待者，也就是说等待条件变量的线程操作相同，每一个从 wait 放回后执行相同的逻辑，同时一个条件变量的通知至多只能唤醒一个线程。

也就是说理论上讲在 put/take 中如果使用 sinallAll 唤醒的话，那么在清单 2 中的 notFull.singal 就是多余的。

出队列过程（poll / take）

再来看出队列过程。清单 3 描述了出队列的过程。可以看到这和入队列是对称的。从这里可以看到，出队列使用的是和入队列不同的锁，所以入队列、出队列这两个操作才能并行进行。

清单 3 阻塞的出队列过程

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        try {
            while (count.get() == 0)
                notEmpty.await();
        } catch (InterruptedException ie) {
            notEmpty.signal(); // propagate to a non-interrupted thread
            throw ie;
        }
        x = extract();
        c = count.getAndDecrement();
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

为什么有异常？

有了入队列、出队列的过程后再来回答前面的几个问题。

为什么总是抛出 InterruptedException 异常？ 这是很大一块内容，其实是 Java 对线程中断的处理问题，希望能够在系列文章的最后能够对此开辟单独的篇章来谈谈。

在锁机制里面也是总遇到，这是因为，Java 里面没有一种直接的方法中断一个挂起的线程，所以通常情况下等于一个处于 WAITING 状态的线程，允许设置一个中断位，一旦线程检测到这个中断位就会从 WAITING 状态退出，以一个 InterruptedException 的异常返回。所以只要是对一个线程挂起操作都会导致 InterruptedException 的可能，比如 Thread.sleep()、Thread.join()、Object.wait()。尽管 LockSupport.park()不会抛出一个 InterruptedException 异常，但是它会将当前线程的的 interrupted 状态位置上，而对于 Lock/Condition 而言，当捕捉到 interrupted 状态后就认为线程应该终止任务，所以就抛出了一个 InterruptedException 异常。

又见 volatile

还有一个不容易理解的问题。**为什么 Node.item 是 volatile 类型的？**

起初我不大明白，因为对于一个进入队列的 Node，它的 item 是不变，当且仅当出队列的时候会将头结点元素的 item 设置为 null。尽管在 remove(o)的时候也是设置为 null,但是那时候是加了 putLock/takeLock 两个锁的，所以肯定是没有问题的。那么问题出在哪？

我们知道，item 的值是在 put/offer 的时候加入的。这时候都是有 putLock 锁保证的，也就是说它保证使用 putLock 锁的读取肯定是没有问题的。那么问题就只可能出在一个不适用 putLock 却需要读取 Node.item 的地方。

peek 操作时获取头结点的元素而不移除它。显然他不会操作尾节点，所以它不需要 putLock 锁，也就是说它只有 takeLock 锁。清单 4 描述了这个过程。

清单 4 查询队列头元素过程

```
public E peek() {
    if (count.get() == 0)
        return null;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        Node<E> first = head.next;
        if (first == null)
            return null;
        else
            return first.item;
    } finally {
        takeLock.unlock();
```

```
    }
}
```

清单 4 描述了 peek 的过程，最后返回一个非 null 节点的结果是 Node.item。这里读取了 Node 的 item 值，但是整个过程却是使用了 takeLock 而非 putLock。换句话说 putLock 对 Node.item 的操作，peek()线程可能不可见！

清单 5 队列尾部加入元素

```
private void insert(E x) {
    last = last.next = new Node<E>(x);
}
```

清单 5 是入队列 offer/put 的一部分，这里关键在于 last=new Node<E>(x)可能发生重排序。Node 构造函数是这样的：Node(E x) { item = x; }。在这一步里面我们可能得到以下一种情况：

1. 构建一个 Node 对象 n；
2. 将 Node 的 n 赋给 last
3. 初始化 n，设置 item=x

在执行步骤 2 的时候一个 peek 线程可能拿到了新的 Node n，这时候它读取 item，得到了一个 null。显然这是不可靠的。

对 item 采用 volatile 之后，JMM 保证对 item=x 的赋值一定在 last=n 之前，也就是说 last 得到的一个是一个已经赋值了的新节点 n。这就不会导致读取空元素的问题的。

出对了 poll/take 和 peek 都是使用的 takeLock 锁，所以不会导致此问题。

删除操作和遍历操作由于同时获取了 takeLock 和 putLock，所以也不会导致此问题。

总结：当前仅当元素加入队列时读取此元素才可能导致不一致的问题。采用 volatile 正式避免此问题。

附加功能

BlockingQueue 有一个额外的功能，允许批量从队列中异常元素。这个 API 是：

int drainTo(Collection<? super E> c, int maxElements); *最多从此队列中移除给定数量的可用元素，并将这些元素添加到给定 collection 中。*

int drainTo(Collection<? super E> c); *移除此队列中所有可用的元素，并将它们添加到给定 collection 中。*

清单 6 描述的是最多移除指定数量元素的过程。由于批量操作只需要一次获取锁，所以效率会比每次获取锁要高。但是需要说明的，需要同时获取 takeLock/putLock 两把锁，因为当移除完所有元素后这会涉及到尾节点的修改（last 节点仍然指向一个已经移走的节点）。

由于迭代操作 **contains()/remove()/iterator()**也是获取了两个锁，所以迭代操作也是线程安全的。

清单 6 批量移除操作

```
public int drainTo(Collection<? super E> c, int maxElements) {
    if (c == null)
        throw new NullPointerException();
```

```
        if (c == this)
            throw new IllegalArgumentException();
        fullyLock();
        try {
            int n = 0;
            Node<E> p = head.next;
            while (p != null && n < maxElements) {
                c.add(p.item);
                p.item = null;
                p = p.next;
                ++n;
            }
            if (n != 0) {
                head.next = p;
                assert head.item == null;
                if (p == null)
                    last = head;
                if (count.getAndAdd(-n) == capacity)
                    notFull.signalAll();
            }
            return n;
        } finally {
            fullyUnlock();
        }
    }
```

深入浅出 Java Concurrency (22): 并发容器 part 7 可阻塞的 BlockingQueue (2)

在上一节中详细分析了**LinkedBlockingQueue** 的实现原理。实现一个可扩展的队列通常有两种方式：一种方式就像LinkedBlockingQueue一样使用链表，也就是每一个元素带有下一个元素的引用，这样的队列原生就是可扩展的；另外一种就是通过数组实现，一旦队列的大小达到数组的容量的时候就将数组扩充一倍（或者一定的系数倍），从而达到扩容的目的。常见的ArrayList就属于第二种。前面章节介绍过的HashMap确是综合使用了这两种方式。

对于一个 Queue 而言，同样可以使用数组实现。使用数组的好处在于各个元素之间原生就是通过数组的索引关联起来的，一次元素之间就是有序的，在通过索引操作数组就方便多了。当然也有它不利的一面，扩容起来比较麻烦，同时删除一个元素也比较低效。

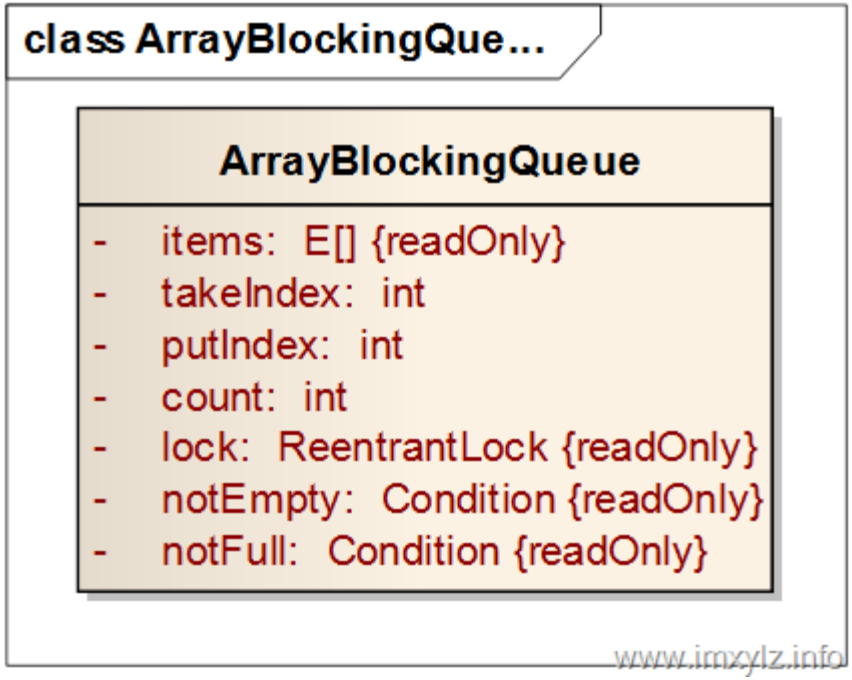
ArrayBlockingQueue 就是 Queue 的一种数组实现。

ArrayBlockingQueue 原理

在没有介绍 ArrayBlockingQueue 原理之前可以想象下，一个数组如何实现 Queue 的 FIFO 特性。首先，数组是固定大小的，这个是毫无疑问的，那么初始化就是所有元素都为 null。假设数组一段为头，另一端为尾。那么头和尾之间的元素就是 FIFO 队列。

1. 入队列就将尾索引往右移动一个，新元素加入尾索引的位置；
2. 出队列就将头索引往尾索引方向移动一个，同时将旧头索引元素设为 null，返回旧头索引的元素。
3. 一旦数组已满，那么就不允许添加新元素（除非扩充容量）
4. 如果尾索引移到了数组的最后（最大索引处），那么就从索引 0 开始，形成一个“闭合”的数组。
5. 由于头索引和尾索引之间的元素都不能为空（因为为空不知道 take 出来的元素为空还是队列为空），所以删除一个头索引和尾索引之间的元素的话，需要移动删除索引前面或者后面的所有元素，以便填充删除索引的位置。
6. 由于是阻塞队列，那么显然需要一个锁，另外由于只是一份数据（一个数组），所以只能有一个锁，也就是同时只能有一个线程操作队列。

有了上述几点分析，设计一个可阻塞的数组队列就比较容易了。



上图描述的 ArrayBlockingQueue 的数据结构。首先有一个数组 E[]，用来存储所有的元素。由于 ArrayBlockingQueue 最终设置为一个不可扩展大小的 Queue，所以这里 items 就是初始化就固定大小的数组（final 类型）；另外有两个索引，头索引 takeIndex，尾索引 putIndex；一个队列的大小 count；要支持阻塞就必须需要一个锁 lock 和两个条件（非空、非满），这三个元素都是不可变更类型的（final）。

由于只有一把锁，所以任何时刻对队列的操作都只有一个线程，这意味着对索引和大小的操作都是线程安全的，所以可以看到这个 takeIndex/putIndex/count 就不需要原子操作和 volatile 语义了。

清单 1 描述的是一个可阻塞的添加元素过程。这与前面介绍的消费者、生产者模型相同。如果队列已经满了就挂起等待，否则就插入元素，同时唤醒一个队列已空的线程。对比清单 2 可以看到是完全相反的两个过程。这在前面几种实现生产者-消费者模型的时候都介绍过了。

清单 1 可阻塞的添加元素

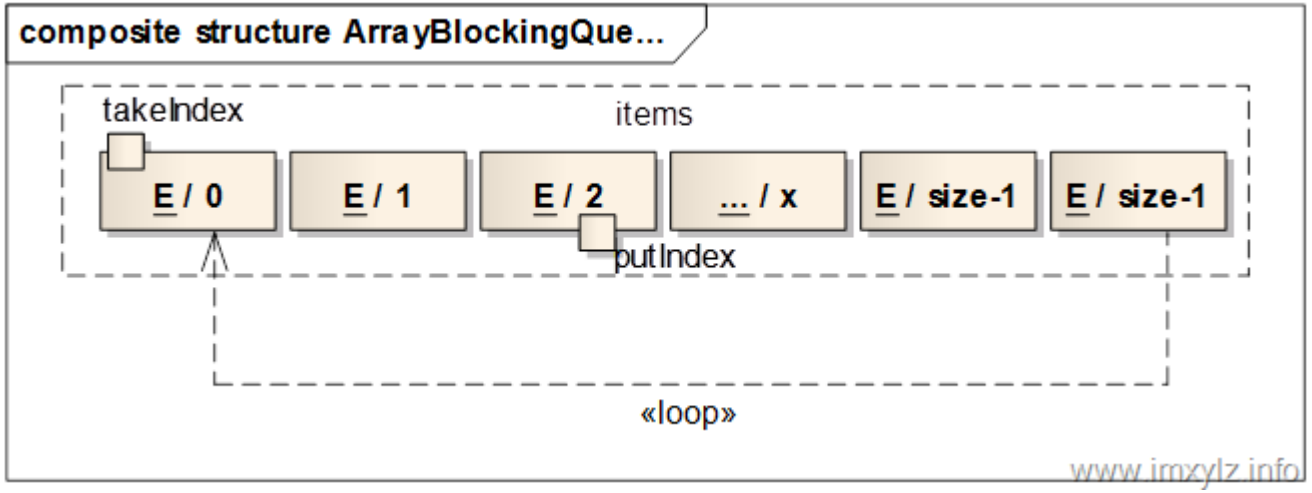
```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    final E[] items = this.items;
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        try {
            while (count == items.length)
                notFull.await();
        } catch (InterruptedException ie) {
            notFull.signal(); // propagate to non-interrupted thread
            throw ie;
        }
        insert(e);
    } finally {
        lock.unlock();
    }
}
```

清单 2 可阻塞的移除元素

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        try {
            while (count == 0)
```

```
                notEmpty.await();
            } catch (InterruptedException ie) {
                notEmpty.signal(); // propagate to non-interrupted thread
                throw ie;
            }
            E x = extract();
            return x;
        } finally {
            lock.unlock();
        }
    }
```

需要注意到的是，尽管每次加入、移除一个元素使用的都是 signal()通知，而不是 signalAll()通知。我们参考上一节中 notify 替换 notifyAll 的原则：每一个 await 醒来的动作相同，每次最多唤醒一个线程来操作。显然这里符合这两种条件，因此使用 signal 要比使用 signalAll 要高效，并且是可靠的。



上图描述了 take()/put()的索引位置示意图。

一开始 takeIndex/putIndex 都在 E/0 位置，然后每加入一个元素 offer/put，putIndex 都增加 1，也就是往后边移动一位；每移除一个元素 poll/take，takeIndex 都增加 1，也是往后边移动一位，显然 takeIndex 总是在 putIndex 的“后边”，因为当队列中没有元素的时候 takeIndex 和 putIndex 相等，同时当前位置也没有元素，takeIndex 也就是无法再往右边移动了；一旦 putIndex/takeIndex 移动到了最后面，也就是 size-1 的位置（这里 size 是指数组的长度），那么就移动到 0，继续循环。循环的前提是数组中元素的个数小于数组的长度。整个过程就是这样的。可见 putIndex 同时指向头元素的下一个位置（如果队列已经满了，那么就是尾元素位置，否则就是一个元素为 null 的位置）。

比较复杂的操作时删除任意一个元素。清单 3 描述的是删除任意一个元素的过程。显然删除任何一个元素需要遍历整个数组，也就是它的复杂度是 O(n)，这与根据索引从 ArrayList 中查找一个元素的复杂度 O(1)相比开销要大得多。参考声明的结构图，一旦删除的是 takeIndex 位置的元素，那么只需要将 takeIndex 往“右

边”移动一位即可；如果删除的是 takeIndex 和 putIndex 之间的元素怎么办？这时候就从删除的位置 i 开始，将 i 后面的所有元素位置都往“左”移动一位，直到 putIndex 为止。最终的结果是删除位置的所有元素都“后退”了一个位置，同时 putIndex 也后退了一个位置。

清单 3 删除任意一个元素

```
public boolean remove(Object o) {
    if (o == null) return false;
    final E[] items = this.items;
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        int i = takeIndex;
        int k = 0;
        for (;;) {
            if (k++ >= count) return false;
            if (o.equals(items[i])) {
                removeAt(i);   return true;
            }
            i = inc(i);
        }
    } finally {   lock.unlock(); }
}
void removeAt(int i) {
    final E[] items = this.items;
    // if removing front item, just advance
    if (i == takeIndex) {
        items[takeIndex] = null;
        takeIndex = inc(takeIndex);
    } else { // slide over all others up through putIndex.
         for (;;) {
            int nexti = inc(i);
            if (nexti != putIndex) {
                items[i] = items[nexti]; i = nexti;
```

```
            } else {
                items[i] = null; putIndex = i; break;
            }
        }
    }
    --count;
    notFull.signal();
}
```

对于其他的操作，由于都是带着 Lock 的操作，所以都比较简单就不再展开了。

下一篇中将介绍另外两个 BlockingQueue， PriorityBlockingQueue 和 SynchronousQueue 然后对这些常见的 Queue 进行一个小范围的对比。

在 Set 中有一个排序的集合 SortedSet，用来保存按照自然顺序排列的对象。Queue 中同样引入了一个支持排序的 FIFO 模型。

并发队列与Queue简介 中介绍了，PriorityQueue和PriorityBlockingQueue就是支持排序的Queue。显然一个支持阻塞的排序Queue要比一个非线程安全的Queue实现起来要复杂的多，因此下面只介绍PriorityBlockingQueue，至于PriorityQueue只需要去掉Blocking功能就基本相同了。

PriorityBlockingQueue

排序的 BlockingQueue — PriorityBlockingQueue

先简单介绍下 PriorityQueue，因为 PriorityBlockingQueue 内部就是通过 PriorityQueue 适配实现的，只不过通过锁进行同步和阻塞而已。

PriorityQueue 是一个数组实现的，是一个二叉树的实现，这个二叉树的任意一个节点都比其子节点要小，这样顶点就是最小的节点。每一个元素或者节点要么本身是可比较的（Comparable），或者队列本身带有一个比较器（Comparator<? super E>），所有元素就是靠比较自身的大小来确定顺序的。而数组中顶点就是数组的第 0 个元素，因此出队列的话总是取第 0 个元素。对于第 0 个元素，其子节点是第 1 个元素和第 2 个元素，对于第 1 个元素，其子元素又是第 3/4 个元素，以此类推，第 i 个元素的父节点就是(i-1)/2。这样任意一个元素加入队列就从其父节点(i-1)/2 开始比较，一旦新节点比父节点小就交换两个节点，然后继续比较新节点与其新的父节点。知道所有节点都是按照父节点一定比子节点小的顺序排列。这是一个有点复杂的算法，此处不再讨论更多的细节。不管是删除还是查找，我们只需要了解的顶点（索引为 0 的元素）总是最小的。

特别需要说明的是 PriorityQueue 是一个无界的队列，也就是说一旦元素的个数达到了数组的大小，那么就将数组扩大 50%，这样这个数组就是无穷大的。当然了如果达到了整数的最大值就会得到一个 OutOfMemoryError，这个是由逻辑保证的。

对于 PriorityBlockingQueue 而言，由于是无界的，因此就只有非空的信号，也就是说只有 take()才能阻塞，put 是永远不会阻塞（除非达到 Integer.MAX_VALUE 直到抛出一个 OutOfMemoryError 异常）。

只有 take()操作的时候才可能因为队列为空而挂起。同时其它需要操作队列变化和大小的只需要使用独占锁 ReentrantLock 就可以了，非常方便。需要说明的是 PriorityBlockingQueue 采用了一个公平的锁。

总的来说 PriorityBlockingQueue 不是一个 FIFO 的队列，而是一个有序的队列，这个队列总是取“自然顺序”最小的对象，同时又是一个只能出队列阻塞的 BlockingQueue，对于入队列却不是阻塞的。所有操作都是线程安全的。

SynchronousQueue

直接交换的 BlockingQueue — SynchronousQueue

这是一个很有意思的阻塞队列，其中每个插入操作必须等待另一个线程的移除操作，同样任何一个移除操作都等待另一个线程的插入操作。因此此队列内部其实没有任何一个元素，或者说容量是 0，严格说并不是一种容器。由于队列没有容量，因此不能调用 peek 操作，因为只有移除元素时才有元素。

一个没有容量的并发队列有什么用了？或者说存在的意义是什么？

SynchronousQueue 的实现非常复杂，当然了如果真要去分析还是能够得到一些经验的，但是前面分析了过多的结构后，发现越来越陷于数据结构与算法里面了。我的初衷是通过研究并发实现的原理来更好的利用并发来最大限度的利用可用资源。所以在后面的章节中尽可能的少研究数据结构和算法，但是为了弄清楚里面的原理，必不可免的会涉及到一些这方面的知识，希望后面能够适可而止。

再回到话题。SynchronousQueue 内部没有容量，但是由于一个插入操作总是对应一个移除操作，反过来同样需要满足。那么一个元素就不会再 SynchronousQueue 里面长时间停留，一旦有了插入线程和移除线程，元素很快就从插入线程移交给移除线程。也就是说这更像是一种信道（管道），资源从一个方向快速传递到另一方向。

需要特别说明的是，尽管元素在 SynchronousQueue 内部不会“停留”，但是并不意味之 SynchronousQueue 内部没有队列。实际上 SynchronousQueue 维护者线程队列，也就是插入线程或者移除线程在不同时存在的时候就会有线程队列。既然有队列，同样就有公平性和非公平性特性，公平性保证正在等待的插入线程或者移除线程以 FIFO 的顺序传递资源。

显然这是一种快速传递元素的方式，也就是说在这种情况下元素总是以最快的方式从插入着（生产者）传递给移除着（消费者），这在多任务队列中是最快处理任务的方式。在线程池的相关章节中还会更多的提到此特性。

事实上在《并发队列与Queue简介》中介绍了还有一种BlockingQueue的实现DelayQueue，它描述的是一种延时队列。这个队列的特性是，队列中的元素都要延迟时间（超时时间），只有一个元素达到了延时时间才能出队列，也就是说每次从队列中获取的元素总是最先到达延时的元素。这种队列的场景就是计划任务。比如以前要完成计划任务，很有可能是使用Timer/TimerTask，这是一种循环检测的方式，也就是在循环里面遍历所有元素总是检测元素是否满足条件，一旦满足条件就执行相关任务。显然这中方式浪费了很多的检测工作，因为大多数时间总是在进行无谓的检测。而DelayQueue 却能避免这种无谓的检测。在线程池的计划任务部分还有更加详细的讨论此队列实现。

下面就对常见的 BlockingQueue 进行小节下，这里不包括双向的队列，尽管 ConcurrentLinkedQueue 不是可阻塞的 Queue，但是这里还是将其放在一起进行对比。

队列	场景	优点	缺点
ConcurrentLinkedQueue	Queue 的最佳线程安全版本，在不适用阻塞功能下几乎是最有效的	使用原子操作，效率高，同时是一种无界的队列	不能阻塞线程，同时无法直接获取队列大小，也不能控制队列的容量
LinkedBlockingQueue	基于链表的 Queue 的可阻塞线程安全	可阻塞，入出队列锁分离，效率高，支	由于存在锁机制，同时链表需要遍历才

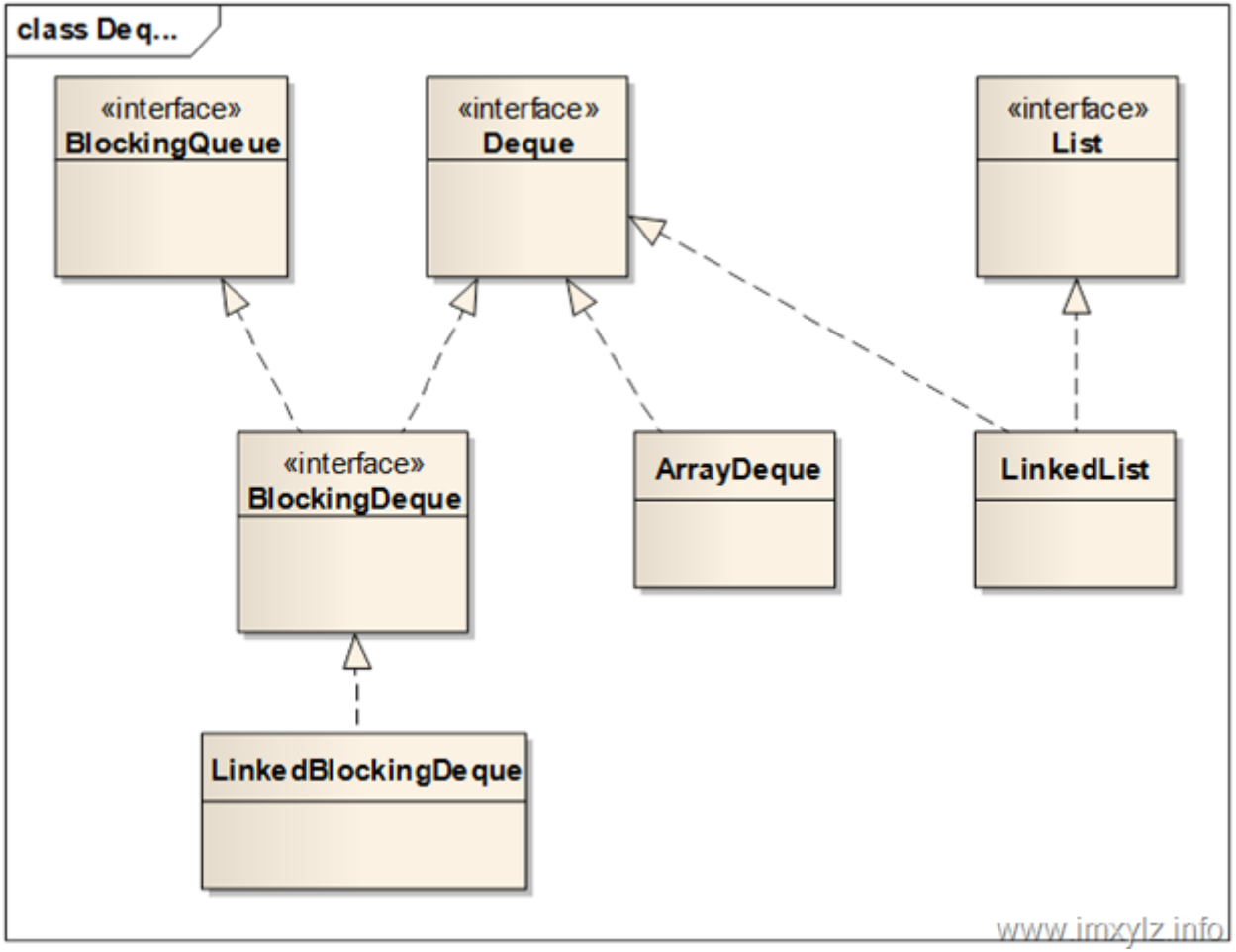
	版本，在需要阻塞且无界的环境下，这是一种不错的选择	持容量限制，可以作为无界的队列实现	能准确定位元素，因此效率有一定的影响
ArrayBlockingQueue	基于数组的 Queue 的可阻塞线程安全版本，在容量固定的环境下，这是一种不错的选择	可阻塞，入出队列效率更高，同时能省内存，遍历元素更快	容量固定，不能扩容，入出队列不能同时进行
PriorityBlockingQueue	按照自然排序实现的阻塞队列，在元素需要排序的情况下是唯一的选择	可阻塞，元素有序，能够自动扩容	出入队列比较慢，效率比较低，基于数组实现，每次扩容需要数组复制，同时容量不能减小，入队列不能够被阻塞
SynchronousQueue	一种直接交换元素的实现，在这快速处理任务队列是最有效的方式	可阻塞，快速交换队列	内部没有容量
DelayQueue	延时处理队列的是吸纳，队列的每个元素都有一个延时时间过期后才能出队列	可阻塞，可延时	基于排序的 Queue 实现，效率很低，入队列不能够被阻塞

如果不需要阻塞队列，优先选择 ConcurrentLinkedQueue；如果需要阻塞队列，队列大小固定优先选择 ArrayBlockingQueue，队列大小不固定优先选择 LinkedBlockingQueue；如果需要对队列进行排序，选择 PriorityBlockingQueue；如果需要一个快速交换的队列，选择 SynchronousQueue；如果需要对队列中的元素进行延时操作，则选择 DelayQueue。

深入浅出 Java Concurrency (24): 并发容器 part 9 双向队列集合 Deque

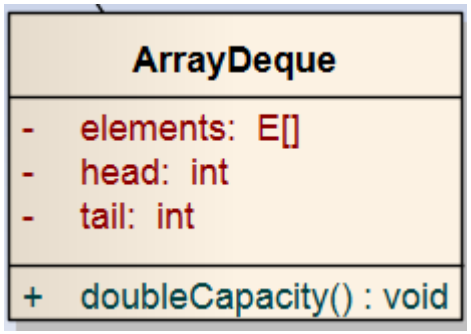
有一段时间没有更新了。接着上节继续吧。

Queue 除了前面介绍的实现外，还有一种双向的 Queue 实现 Deque。这种队列允许在队列头和尾部进行入队出队操作，因此在功能上比 Queue 显然要更复杂。下图描述的是 Deque 的完整体系图。需要说明的是 LinkedList 也已经加入了 Deque 的一部分（LinkedList 是从 jdk1.2 开始就存在数据结构）。



Deque 在 Queue 的基础上增加了更多的操作方法。

	第一个元素		最后一个元素		描述
	抛出异常	返回特殊值	抛出异常	返回特殊值	
插入	addFirst(e)	offerFist(e)	addLast(e)	offerLast(e)	将元素加入队列
	push(e)		add(e)	offer(e)	
移除	removeFirst()	pollFirst()	removeLast()	pollLast()	将元素移除队列
	remove()/pop()	poll()			
检查	getFirst()	peekFirst()	getLast()	peekLast()	查看队列头或尾元素
	element()	peek()			

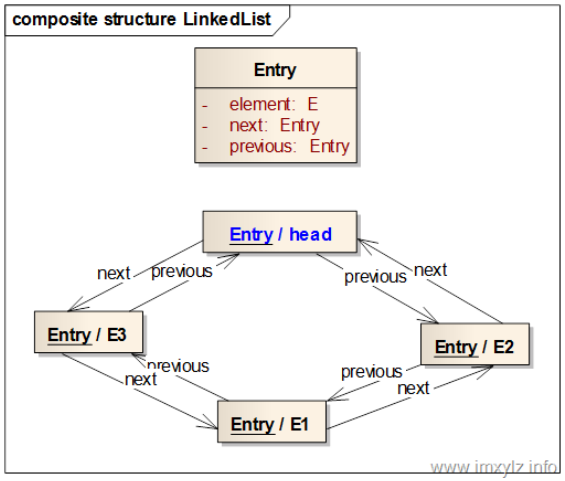


同时在 Deque 的体系结构图中可以看到，实现一个 Deque 可以使用数组（ArrayDeque），同时也可以使用链表（LinkedList），还可以同实现一个支持阻塞的线程安全版本队列 LinkedBlockingDeque。

对于数组实现的 Deque 来说，数据结构上比较简单，只需要一个存储数据的数组以及头尾两个索引即可。由于数组是固定长度的，所以很容易就得到数组的头和尾，那么对于数组的操作只需要移动头和尾的索引即可。

特别说明的是 ArrayDeque 并不是一个固定大小的队列，每次队列满了以后就将队列容量扩大一倍（doubleCapacity()），因此加入一个元素总是能成功，而且也不会抛出一个异常。也就是说 ArrayDeque 是一个没有容量限制的队列。

同样继续性能的考虑，使用 System.arraycopy 复制一个数组比循环设置要高效得多。



对于 LinkedList 本身而言，数据结构就更简单了，除了一个 size 用来记录大小外，只有 head 一个元素 Entry。对比 Map 和 Queue 的其它数据结构可以看到这里的 Entry 有两个引用，是双向的队列。

在示意图中，LinkedList 总是有一个“傀儡”节点，用来描述队列“头部”，但是并不表示头部元素，它是一个执行 null 的空节点。

队列一开始只有 head 一个空元素，然后从尾部加入 E1(add/addLast)，head 和 E1 之间建立双向链接。然后继续从尾部加入 E2，E2 就在 head 和 E1 之间建立双向链接。最后从队列的头部加入 E3(push/addFirst)，于是 E3 就在 E1 和 head 之间链接双向链接。

双向链表的数据结构比较简单，操作起来也比较容易，从事从“傀儡”节点开始，“傀儡”节点的下一个元素就是队列的头部，前一个元素是队列的尾部，换句话说，“傀儡”节点在头部和尾部之间建立了一个通道，是整个队列形成一个循环，这样就可以从任意一个节点的任意一个方向能遍历完整的队列。

同样 LinkedList 也是一个没有容量限制的队列，因此入队列（不管是从头部还是尾部）总能成功。

上面描述的 ArrayDeque 和 LinkedList 是两种不同方式的实现，通常在遍历和节省内存上 ArrayDeque 更高效（索引更快，另外不需要 Entry 对象），但是在队列扩容下 LinkedList 更灵活，因为不需要复制原始的队列，某些情况下可能更高效。

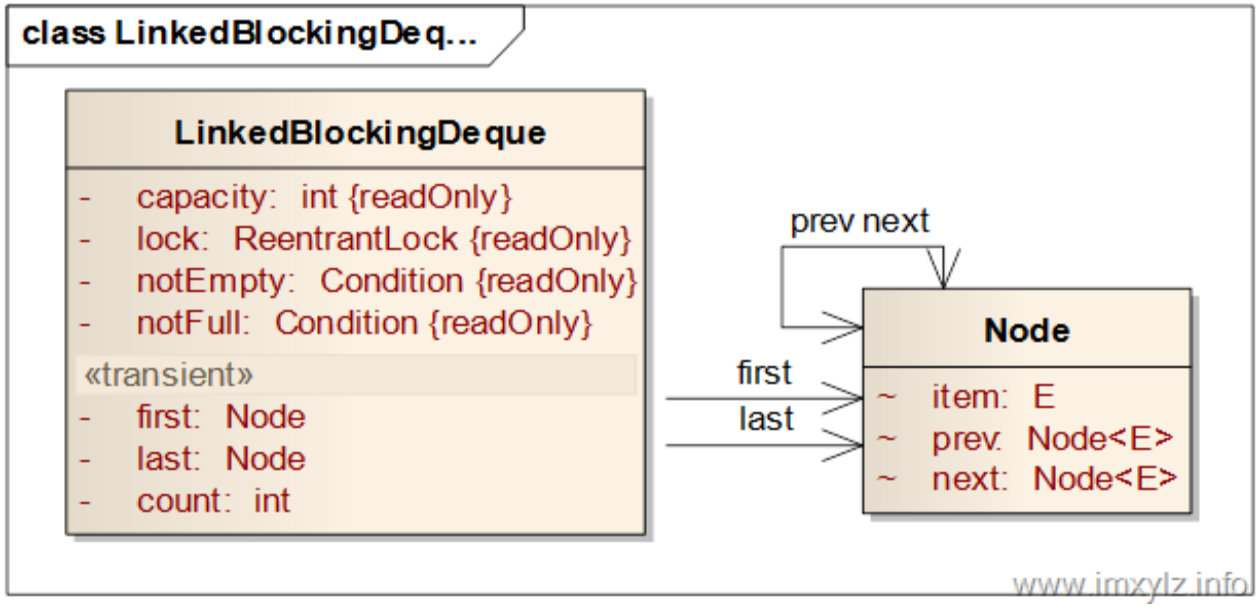
同样需要注意的上述两个实现都不是线程安全的，因此只适合在单线程环境下使用，下面章节要介绍的 LinkedBlockingDeque 就是线程安全的可阻塞的 Deque。事实上也应该是功能最强大的 Queue 实现，当然了实现起来也许会复杂一点。

BlockingDeque 双向并发阻塞队列

深入浅出 Java Concurrency (25): 并发容器 part 10 双向并发阻塞队列 BlockingDeque

这个小节介绍 Queue 的最后一个工具，也是最强大的一个工具。从名称上就可以看到此工具的特点：双向并发阻塞队列。所谓双向是指可以从队列的头和尾同时操作，并发只是线程安全的实现，阻塞允许在入队出队不满足条件时挂起线程，这里说的队列是指支持 FIFO/FILO 实现的链表。

首先看下 LinkedBlockingDeque 的数据结构。通常情况下从数据结构上就能看出这种实现的优缺点，这样就知道如何更好的使用工具了。



从数据结构和功能需求上可以得到以下结论：

1. 要想支持阻塞功能，队列的容量一定是固定的，否则无法在入队的时候挂起线程。也就是 capacity 是 final 类型的。
2. 既然是双向链表，每一个结点就需要前后两个引用，这样才能将所有元素串联起来，支持双向遍历。也即需要 prev/next 两个引用。
3. 双向链表需要头尾同时操作，所以需要 first/last 两个节点，当然可以参考 LinkedList 那样采用一个节点的双向来完成，那样实现起来就稍微麻烦点。
4. 既然要支持阻塞功能，就需要锁和条件变量来挂起线程。这里使用一个锁两个条件变量来完成此功能。

有了上面的结论再来研究 LinkedBlockingDeque 的优缺点。

优点当然是功能足够强大，同时由于采用一个独占锁，因此实现起来也比较简单。所有对队列的操作都加锁就可以完成。同时独占锁也能够很好的支持双向阻塞的特性。

凡事有利必有弊。缺点就是由于独占锁，所以不能同时进行两个操作，这样性能上就大打折扣。从性能的角度讲 LinkedBlockingDeque 要比 LinkedBlockingQueue 要低很多，比 CocurrentLinkedQueue 就低更多了，这在高并发情况下就比较明显了。

前面分析足够多的 Queue 实现后，LinkedBlockingDeque 的原理和实现就不值得一提了，无非是在独占锁下对一个链表的普通操作。

有趣的是此类支持序列化，但是 Node 并不支持序列化，因此 fist/last 就不能序列化，那么如何完成序列化/反序列化过程呢？

清单 1 LinkedBlockingDeque 的序列化、反序列化

```
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    lock.lock();
    try {
        // Write out capacity and any hidden stuff
        s.defaultWriteObject();
        // Write out all elements in the proper order.
        for (Node<E> p = first; p != null; p = p.next) s.writeObject(p.item);
        // Use trailing null as sentinel
        s.writeObject(null);
    } finally {
        lock.unlock();
    }
}
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    s.defaultReadObject();
    count = 0;
    first = null;
    last = null;
    // Read in all elements and place in queue
    for (;;) {
        E item = (E)s.readObject();
        if (item == null)
            break;
```

```
            add(item);
        }
    }
```

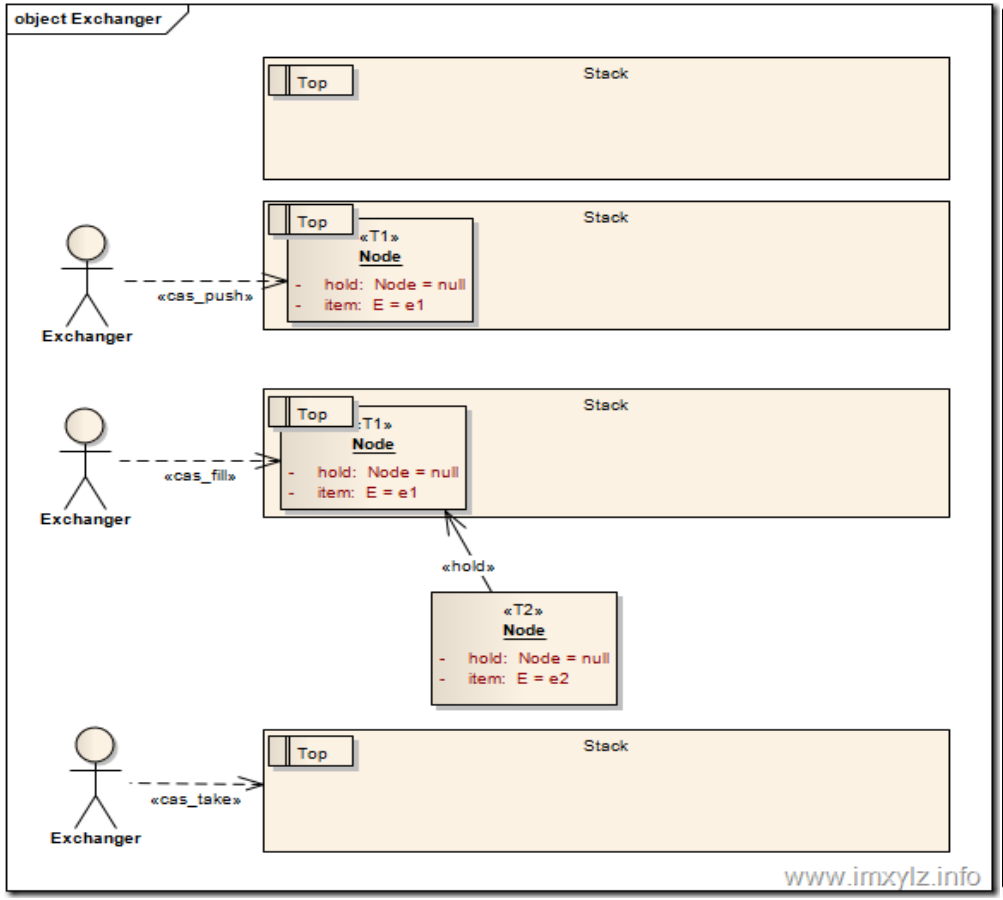
清单 1 描述的是 LinkedBlockingDeque 序列化/反序列化的过程。序列化时将真正的元素写入输出流，最后还写入了一个 null。读取的时候将所有对象列表读出来，如果读取到一个 null 就表示结束。这就是为什么写入的时候写入一个 null 的原因，因为没有将 count 写入流，所以就靠 null 来表示结束，省一个整数空间。

Exchanger

深入浅出 Java Concurrency (26): 并发容器 part 11 Exchanger

可以在对中对元素进行配对和交换的线程的同步点。每个线程将条目上的某个方法呈现给 exchange 方法，与伙伴线程进行匹配，并且在返回时接收其伙伴的对象。Exchanger 可能被视为 SynchronousQueue 的双向形式。

换句话说 Exchanger 提供的是一个交换服务，允许原子性的交换两个（多个）对象，但同时只有一对才会成功。先看一个简单的实例模型。




```
00  Object exchange(Object x, boolean timed,
01    long patience) throws TimeoutException {
02    boolean success = false;
03    long start = System.nanotime();
04    Node mine = new Node(x);
05    for (;;) {
06      Node top = stack.getTop();
07      if (top == null) {
08        if (stack.casTop(null, mine)) {
09          while (null == mine.hole) {
10            if (timedOut(start, timed, patience) {
11              if (mine.casHole(null, FAIL))
12                throw new TimeoutException();
13              break;
14            }
15            /* else spin */
16          }
17          return mine.hole.item;
18        }
19      } else {
20        success = top.casHole(null, mine);
21        stack.casTop(top, null);
22        if (success)
23          return top.item;
24      }
25    }
26  }
```


在上面的模型中，我们假定一个空的栈（Stack），栈顶（Top）当然是没有元素的。同时我们假定一个数据结构 Node，包含一个要交换的元素 E 和一个要填充的“洞”Node。这时线程 T1 携带节点 node1 进入栈（cas_push），当然这是 CAS 操作，这样栈顶就不为空了。线程 T2 携带节点 node2 进入栈，发现栈里面已经有元素了 node1，同时发现 node1 的 hold（Node）为空，于是将自己（node2）填充到 node1 的 hold 中（cas_fill）。然后将元素 node1 从栈中弹出（cas_take）。这样线程 T1 就得到了 node1.hold.item 也就是 node2 的元素 e2，线程 T2 就得到了 node1.item 也就是 e1，从而达到了交换的目的。

算法描述就是下图展示的内容。

JDK 5 就是采用类似的思想实现的 Exchanger。JDK 6 以后为了支持多线程多对象同时 Exchanger 了就进行了改造（为了支持更好的并发），采用 ConcurrentHashMap 的思想，将 Stack 分割成很多的片段（或者说插槽 Slot），线程 Id（Thread.getId()）hash 相同的落在同一个 Slot 上，这样在默认 32 个 Slot 上就有很好的吞吐量。当然会根据机器 CPU 内核的数量有一定的优化，有兴趣的可以去了解下 Exchanger 的源码。

至于 Exchanger 的使用，在 JDK 文档上有个例子，讲述的是两个线程交换数据缓冲区的例子（实际上仍然可以认为是生产者/消费者模型）。

```
class FillAndEmpty {
  Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
  DataBuffer initialEmptyBuffer = .... a made-up type
  DataBuffer initialFullBuffer = ....
  class FillingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialEmptyBuffer;
      try {
        while (currentBuffer != null) {
          addToBuffer(currentBuffer);
          if (currentBuffer.isFull()) currentBuffer = exchanger.exchange(currentBuffer);
        }
      } catch (InterruptedException ex) { ... handle.... }
    }
  }
  class EmptyingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialFullBuffer;
      try {
        while (currentBuffer != null) {
          takeFromBuffer(currentBuffer);
          if (currentBuffer.isEmpty()) currentBuffer = exchanger.exchange(currentBuffer);

        } } catch (InterruptedException ex) { ··· handle ···}

    }
  }
  void start() {
    new Thread(new FillingLoop()).start();   new Thread(new EmptyingLoop()).start();
  }
}
```

Exchanger 实现的是一种数据分片的思想，这在大数据情况下将数据分成一定的片段并且多线程执行的情况下有一定的使用价值。

最近一直推托工作忙，更新频度越来越低了，好在现在的工作还有点个人时间，以后争取多更新下吧，至少也要把这个专辑写完。

线程安全的 List/Set

深入浅出 Java Concurrency (27): 并发容器 part 12 线程安全的 List/Set

本小节是《并发容器》的最后一部分，这一个小节描述的是针对 List/Set 接口的一个线程版本。

在《并发队列与Queue简介》中介绍了并发容器的一个概括，主要描述的是Queue的实现。其中特别提到一点LinkedList是List/Queue的实现，但是LinkedList确实非线程安全的。不管BlockingQueue还是ConcurrentMap的实现，我们发现都是针对链表的实现，当然尽可能的使用CAS或者Lock的特性，同时都有通过锁部分容器来提供并发的特性。而对于List或者Set而言，增、删操作其实都是针对整个容器，因此每次操作都不可避免的需要锁定整个容器空间，性能肯定会大打折扣。要实现一个线程安全的List/Set，只需要在修改操作的时候进行同步即可，比如使用java.util.Collections.synchronizedList(List<T>)或者java.util.Collections.synchronizedSet(Set<T>)。当然也可以使用Lock来实现线程安全的List/Set。

通常情况下我们的高并发都发生在“多读少写”的情况，因此如果能够实现一种更优秀的算法这对生产环境还是很有好处的。ReadWriteLock 当然是一种实现。CopyOnWriteArrayList/CopyOnWriteArraySet 确实另外一种思路。

CopyOnWriteArrayList/CopyOnWriteArraySet 的基本思想是一旦对容器有修改，那么就“复制”一份新的集合，在新的集合上修改，然后将新集合复制给旧的引用。当然了这部分少不了要加锁。显然对于 CopyOnWriteArrayList/CopyOnWriteArraySet 来说最大的好处就是“读”操作不需要锁了。

我们来看看源码。

```
/** The array, accessed only via getArray/setArray. */
private volatile transient Object[] array;
public E get(int index) {
    return (E)(getArray()[index]);
}
private static int indexOf(Object o, Object[] elements, int index, int fence) {
    if (o == null) {
        for (int i = index; i < fence; i++)  if (elements[i] == null) return i;
    } else {
        for (int i = index; i < fence; i++) if (o.equals(elements[i])) return i;
    }
    return -1;
}
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}
    public void clear() {
        final ReentrantLock lock = this.lock;
```

```
        lock.lock();
        try {
         setArray(new Object[0]);
        } finally { lock.unlock();}
    }
```

对于上述代码，有几点说明：

1. List 仍然是基于数组的实现，因为只有数组是最快的。
2. 为了保证无锁的读操作能够看到写操作的变化，因此数组 array 是 volatile 类型的。
3. get/indexOf/iterator 等操作都是无锁的，同时也可以看到所操作的都是某一时刻 array 的镜像（这得益于数组是不可变化的）
4. add/set/remove/clear 等元素变化的都是需要加锁的，这里使用的是 ReentrantLock。

这里有一段有意思的代码片段。

```
public E set(int index, E element) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        Object oldValue = elements[index];
        if (oldValue != element) {
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len);
            newElements[index] = element;  setArray(newElements);
        } else { // Not quite a no-op; ensures volatile write semantics
            setArray(elements);
        }
        return (E)oldValue;
    } finally { lock.unlock(); }
    }
final void setArray(Object[] a) {
    array = a;
}
```

对于 set 操作，如果元素有变化，修改后 setArray(newElements);将新数组赋值还好理解。那么如果一个元素没有变化，也就是上述代码的 else 部分，为什么还需要进行一个无谓的 setArray 操作？毕竟 setArray 操作没有改变任何数据。

对于这个问题也是很有意思，有一封邮件讨论了此问题（1、2、3）。

大致的意思是，尽管没有改变任何数据，但是为了保持“volatile”的语义，任何一个读操作都应该是一个写操作的结果，也就是读操作看到的数据一定是某个写操作的结果（尽管写操作没有改变数据本身）。所以这里即使不设置也没有问题，仅仅是为了一个语义上的补充（个人理解）。

这里还有一个有意思的讨论，说什么 addIfAbsent 在元素没有变化的时候为什么没有 setArray 操作？这个要看怎么理解 addIfAbsent 的语义了。如果说 addIfAbsent 语义是”写“或者”不写“操作，而把”不写“操作当作一次”读“操作的话，那么”读“操作就不需要保持 volatile 语义了。

对于 CopyOnWriteArraySet 而言就简单多了，只是持有一个 CopyOnWriteArrayList，仅仅在 add/addAll 的时候检测元素是否存在，如果存在就不加入集合中。

```
    private final CopyOnWriteArrayList<E> al;
    /** Creates an empty set.*/
    public CopyOnWriteArraySet() {
        al = new CopyOnWriteArrayList<E>();
    }
    public boolean add(E e) {
       return al.addIfAbsent(e);
    }
```

在使用上 CopyOnWriteArrayList/CopyOnWriteArraySet 就简单多了，和 List/Set 基本相同，这里就不再介绍了。

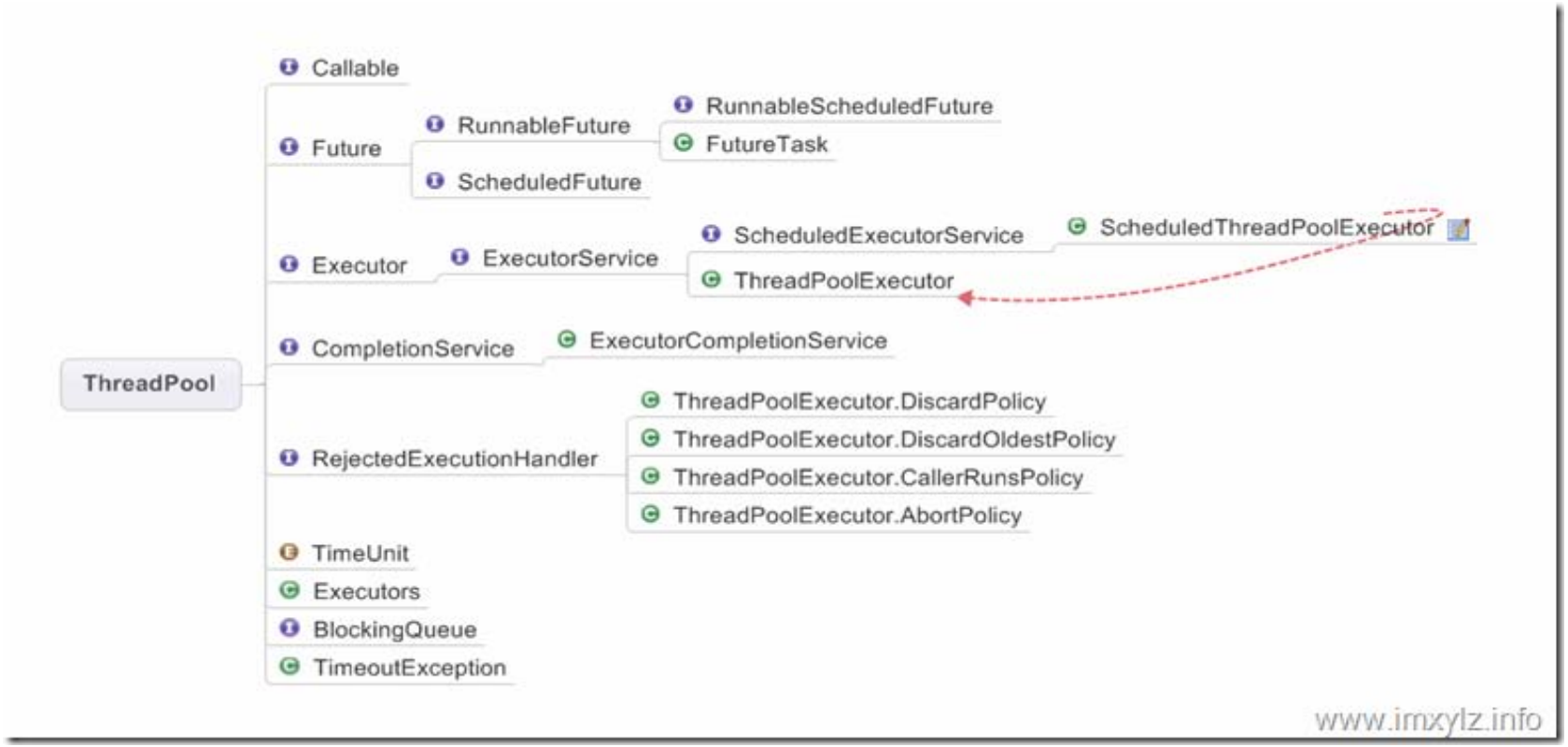
整个并发容器结束了，接下来好好规划下线程池部分，然后进入最后一部分的梳理。

线程池

深入浅出 Java Concurrency (28): 线程池 part 1 简介

从这一节开始正式进入线程池的部分。其实整个体系已经拖了很长的时间，因此后面的章节会加快速度，甚至只是一个半成品或者简单化，以后有时间的慢慢补充、完善。

其实线程池是并发包里面很重要的一部分，在实际情况中也是使用很多的一个重要组件。下图描述的是线程池 API 的一部分。广义上的完整线程池可能还包括 Thread/Runnable、Timer/TimerTask 等部分。这里只介绍主要的和高级的 API 以及架构和原理。



大多数并发应用程序是围绕执行任务（Task）进行管理的。所谓任务就是抽象、离散的工作单元（unit of work）。把一个应用程序的工作（work）分离到任务中，可以简化程序的管理；这种分离还在不同事物间划分了自然的分界线，可以方便程序在出现错误时进行恢复；同时这种分离还可以为并行工作提供一个自然的结构，有利于提高程序的并发性。[1]

并发执行任务的一个很重要前提是拆分任务。把一个大的过程或者任务拆分成很多小的工作单元，每一个工作单元可能相关、也可能无关，这些单元在一定程度上可以充分利用 CPU 的特性并发的执行，从而提高并发性（性能、响应时间、吞吐量等）。

所谓的任务拆分就是确定每一个执行任务（工作单元）的边界。理想情况下独立的工作单元有最大的吞吐量，这些工作单元不依赖于其它工作单元的状态、结果或者其他资源等。因此将任务尽可能的拆分成一个个独立的工作单元有利于提高程序的并发性。

对于有依赖关系以及资源竞争的工作单元就涉及到任务的调度和负载均衡。工作单元的状态、结果或者其他资源等有关联的工作单元就需要有一个总体的调度者来协调资源和执行顺序。同样在有限的资源情况下，大量的任务也需要一个协调各个工作单元的调度者。这就涉及到任务执行的策略问题。

任务的执行策略包括 4W3H 部分：

- 任务在什么（What）线程中执行
- 任务以什么（What）顺序执行（FIFO/LIFO/优先级等）
- 同时有多少个（How Many）任务并发执行

- 允许有多少个（How Many）个任务进入执行队列
- 系统过载时选择放弃哪一个（Which）任务，如何（How）通知应用程序这个动作
- 任务执行的开始、结束应该做什么（What）处理

在后面的章节中会详细分写这些策略是如何实现的。我们先来简单回答些如何满足上面的条件。

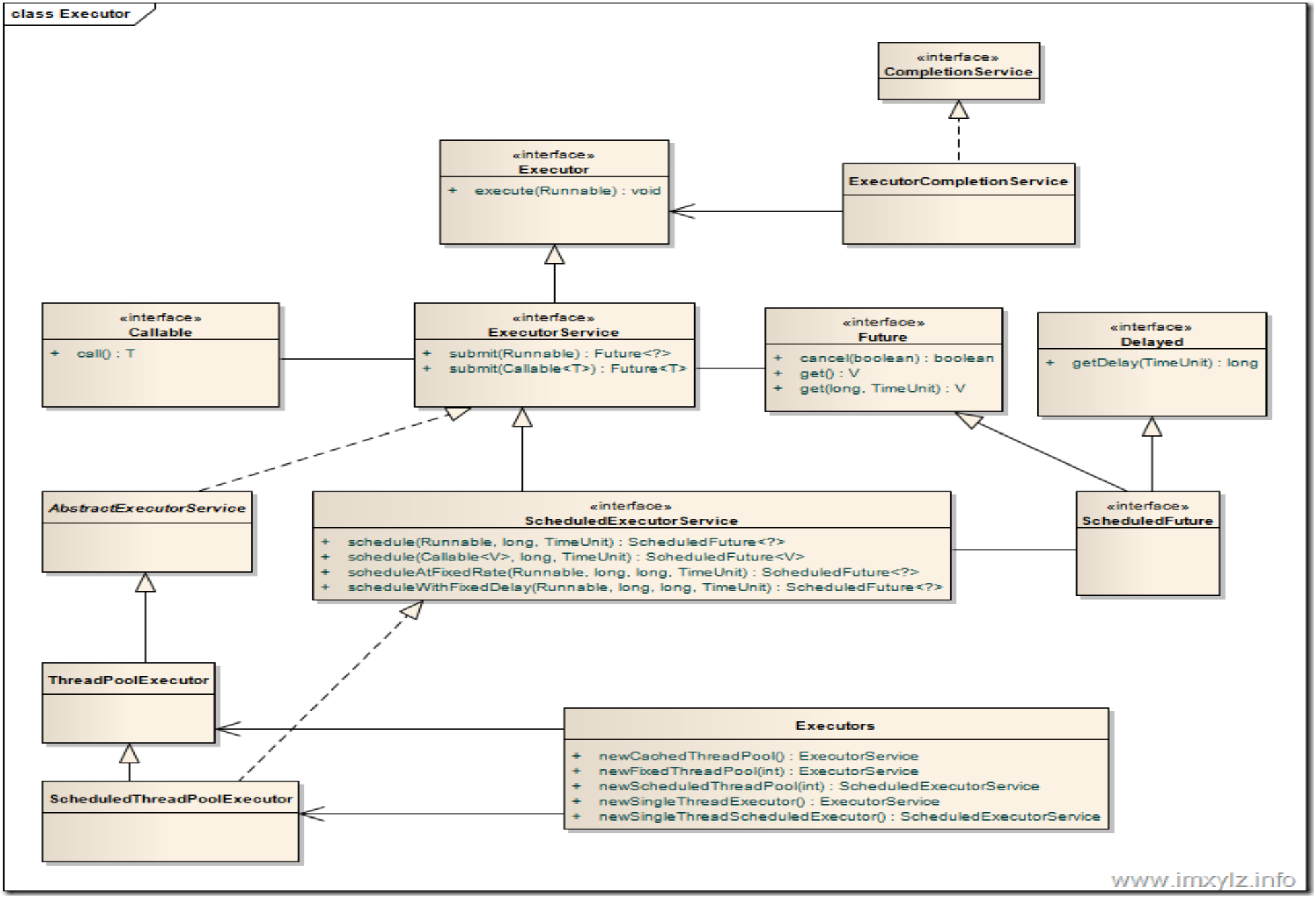
1. 首先明确一定是在 Java 里面可以供使用者调用的启动线程类是 Thread。因此 Runnable 或者 Timer/TimerTask 等都是要依赖 Thread 来启动的，因此在 ThreadPool 里面同样也是靠 Thread 来启动多线程的。
2. 默认情况下 Runnable 接口执行完毕后是不能拿到执行结果的，因此在 ThreadPool 里就定义了一个 Callable 接口来处理执行结果。
3. 为了异步阻塞的获取结果，Future 可以帮助调用线程获取执行结果。
4. Executor 解决了向线程池提交任务的入口问题，同时 ScheduledExecutorService 解决了如何进行重复调用任务的问题。
5. CompletionService 解决了如何按照执行完毕的顺序获取结果的问题，这在某些情况下可以提高任务执行的并发，调用线程不必在长时间任务上等待过多时间。
6. 显然线程的数量是有限的，而且也不宜过多，因此合适的任务队列是必不可少的，BlockingQueue 的容量正好可以解决此问题。
7. 固定任务容量就意味着在容量满了以后需要一定的策略来处理过多的任务（新任务），RejectedExecutionHandler 正好解决此问题。
8. 一定时间内阻塞就意味着有超时，因此 TimeoutException 就是为了描述这种现象。TimeUnit 是为了描述超时时间方便的一个时间单元枚举类。
9. 有上述问题就意味了配置一个合适的线程池是很复杂的，因此 Executors 默认的一些线程池配置可以减少这个操作。

线程池的基本策略大致就这些，从下一节开始就从线程池的基本原理和执行方法开始描述。

深入浅出 Java Concurrency (29): 线程池 part 2 Executor 以及 Executors

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。

下面这张图完整描述了线程池的类体系结构。



首先 Executor 的 execute 方法只是执行一个 Runnable 的任务，当然了从某种角度上将最后的实现类也是在线程中启动此任务的。根据线程池的执行策略最后这个任务可能在新的线程中执行，或者线程池中的某个线程，甚至是调用者线程中执行（相当于直接运行 Runnable 的 run 方法）。这点在后面会详细说明。

ExecutorService 在 Executor 的基础上增加了一些方法，其中有两个核心的方法：

- Future<?> submit(Runnable task)
- <T> Future<T> submit(Callable<T> task)

这两个方法都是向线程池中提交任务，它们的区别在于 Runnable 在执行完毕后没有结果，Callable 执行完毕后有一个结果。这在多个线程中传递状态和结果是非常有用的。另外他们的相同点在于都返回一个 Future 对象。Future 对象可以阻塞线程直到运行完毕（获取结果，如果有的话），也可以取消任务执行，当然也能够检测任务是否被取消或者是否执行完毕。

在没有 Future 之前我们检测一个线程是否执行完毕通常使用 Thread.join()或者用一个死循环加状态位来描述线程执行完毕。现在有了更好的方法能够阻塞线程，检测任务执行完毕甚至取消执行中或者未开始执行的任务。

ScheduledExecutorService 描述的功能和 Timer/TimerTask 类似，解决那些需要任务重复执行的问题。这包括延迟时间一次性执行、延迟时间周期性执行以及固定延迟时间周期性执行等。当然了继承 ExecutorService 的 ScheduledExecutorService 拥有 ExecutorService 的全部特性。

ThreadPoolExecutor 是 ExecutorService 的默认实现，其中的配置、策略也是比较复杂的，在后面的章节中会有详细的分析。

ScheduledThreadPoolExecutor 是继承 ThreadPoolExecutor 的 ScheduledExecutorService 接口实现，周期性任务调度的类实现，在后面的章节中会有详细的分析。

这里需要稍微提一下的是 CompletionService 接口，它是用于描述顺序获取执行结果的一个线程池包装器。它依赖一个具体的线程池调度，但是能够根据任务的执行先后顺序得到执行结果，这在某些情况下可能提高并发效率。

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在 Executors 类里面提供了一些静态工厂，生成一些常用的线程池。

- **newSingleThreadExecutor**：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
- **newFixedThreadPool**：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
- **newCachedThreadPool**：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。
- **newScheduledThreadPool**：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。
- **newSingleThreadExecutor**：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

在详细讲解 ThreadPoolExecutor 的时候会具体讨论上述参数配置后的意义和原理。

线程池是一个复杂的任务调度工具，因此它涉及到任务、线程池等的生命周期问题，在下一节中来探讨下这个问题。

深入浅出 Java Concurrency (30): 线程池 part 3 Executor 生命周期

我们知道线程是有多种执行状态的，同样管理线程的线程池也有多种状态。JVM 会在所有线程（非后台 daemon 线程）全部终止后才退出，为了节省资源和有效释放资源关闭一个线程池就显得很重要。有时候无法正确的关闭线程池，将会阻止 JVM 的结束。

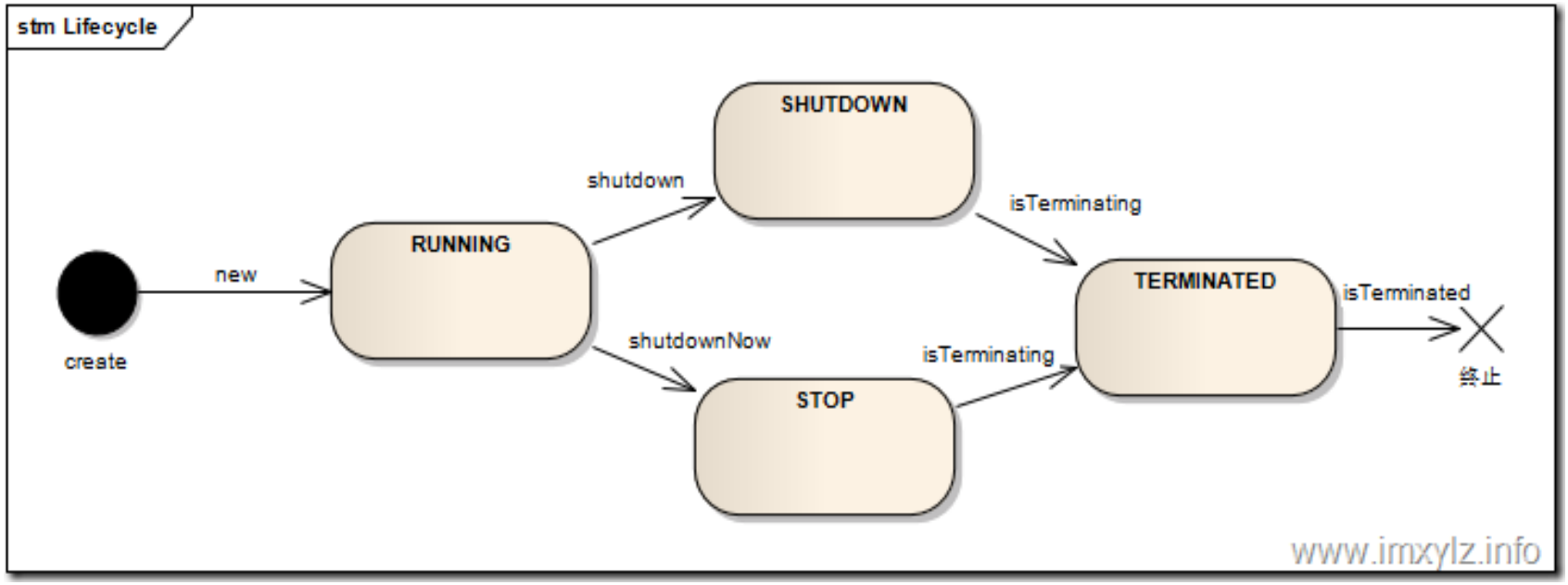
线程池 Executor 是异步的执行任务，因此任何时刻不能够直接获取提交的任务的状态。这些任务有可能已经完成，也有可能正在执行或者还在排队等待执行。因此关闭线程池可能出现一下几种情况：

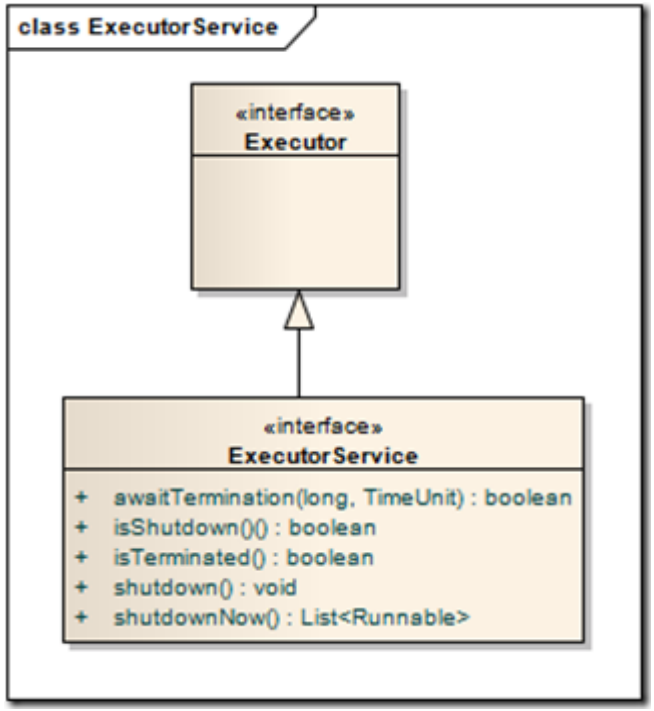
- 平缓关闭：已经启动的任务全部执行完毕，同时不再接受新的任务
- 立即关闭：取消所有正在执行和未执行的任务

另外关闭线程池后对于任务的状态应该有相应的反馈信息。

图 1 描述了线程池的 4 种状态。

- 线程池在构造前（new 操作）是初始状态，一旦构造完成线程池就进入了执行状态 RUNNING。严格意义上讲线程池构造完成后并没有线程被立即启动，只有进行“预启动”或者接收到任务的时候才会启动线程。这个会后面线程池的原理会详细分析。但是线程池是出于运行状态，随时准备接受任务来执行。
- 线程池运行中可以通过 shutdown()和 shutdownNow()来改变运行状态。shutdown()是一个平缓的关闭过程，线程池停止接受新的任务，同时等待已经提交的任务执行完毕，包括那些进入队列还没有开始的任务，这时候线程池处于 SHUTDOWN 状态；shutdownNow()是一个立即关闭过程，线程池停止接受新的任务，同时线程池取消所有执行的任务和已经进入队列但是还没有执行的任务，这时候线程池处于 STOP 状态。
- 一旦 shutdown()或者 shutdownNow()执行完毕，线程池就进入 TERMINATED 状态，此时线程池就结束了。
- isTerminating()描述的是 SHUTDOWN 和 STOP 两种状态。
- isShutdown()描述的是非 RUNNING 状态，也就是 SHUTDOWN/STOP/TERMINATED 三种状态。





线程池的 API 如下：

其中 shutdownNow()会返回那些已经进入了队列但是还没有执行的任务列表。awaitTermination 描述的是等待线程池关闭的时间，如果等待时间线程池还没有关闭将会抛出一个超时异常。

对于关闭线程池期间发生的任务提交情况就会触发一个拒绝执行的操作。这是 java.util.concurrent.RejectedExecutionHandler 描述的任务操作。下一个小结中将描述这些任务被拒绝后的操作。

总结下这个小节：

1. 线程池有运行、关闭、停止、结束四种状态，结束后就会释放所有资源
2. 平缓关闭线程池使用 shutdown()
3. 立即关闭线程池使用 shutdownNow()，同时得到未执行的任务列表
4. 检测线程池是否正处于关闭中，使用 isShutdown()
5. 检测线程池是否已经关闭使用 isTerminated()
6. 定时或者永久等待线程池关闭结束使用 awaitTermination()操作

深入浅出 Java Concurrency (31): 线程池 part 4 线程池任务拒绝策略

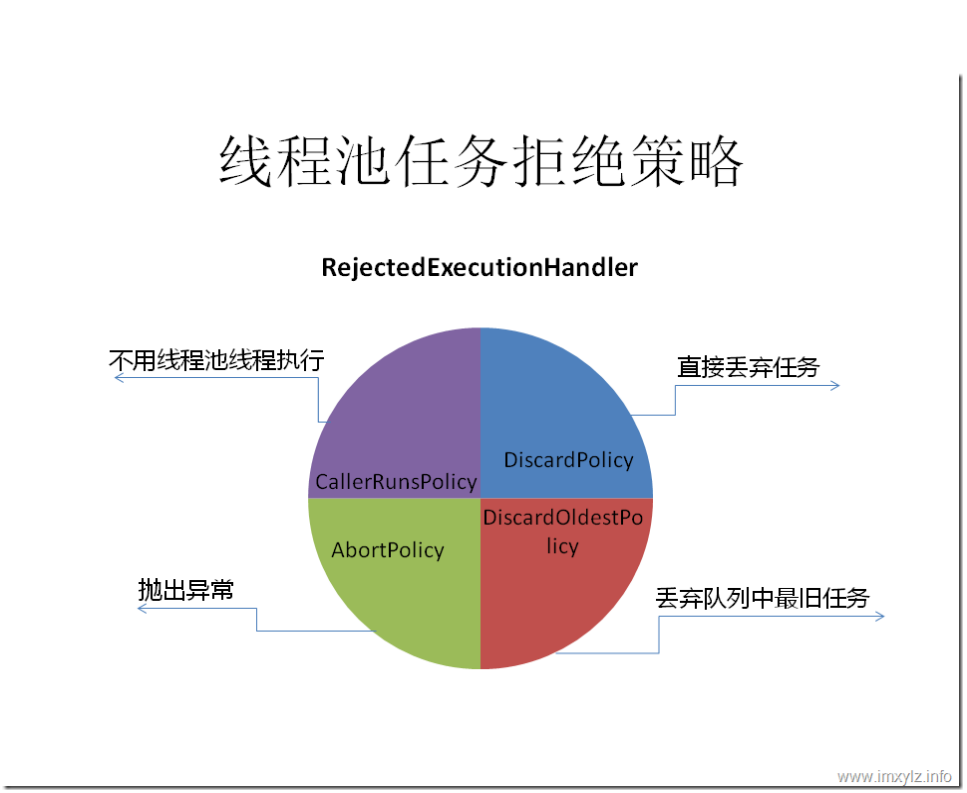
[本文PDF地址：http://www.blogjava.net/Files/xylz/Inside.Java.Concurrency_31.ThreadPool.part4_RejectedPolicy.pdf]

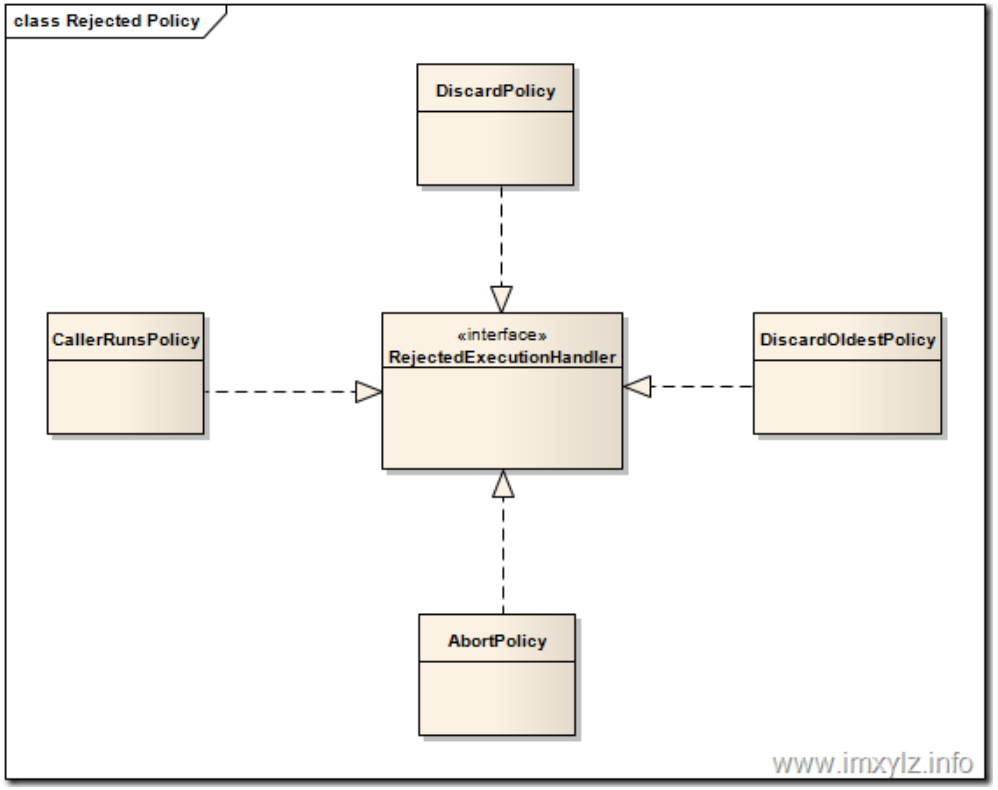
上一节中提到关闭线程池过程中需要对新提交的任务进行处理。这个是 java.util.concurrent.RejectedExecutionHandler 处理的逻辑。

在没有分析线程池原理之前先来分析下为什么有任务拒绝的情况发生。

这里先假设一个前提：线程池有一个任务队列，用于缓存所有待处理的任务，正在处理的任务将从任务队列中移除。因此在任务队列长度有限的情况下就会出现新任务的拒绝处理问题，需要有一种策略来处理应该加入任务队列却因为队列已满无法加入的情况。另外在线程池关闭的时候也需要对任务加入队列操作进行额外的协调处理。

RejectedExecutionHandler 提供了四种方式来处理任务拒绝策略。





这四种策略是独立无关的，是对任务拒绝处理的四中表现形式。最简单的方式就是直接丢弃任务。但是却有两种方式，到底是该丢弃哪一个任务，比如可以丢弃当前将要加入队列的任务本身（DiscardPolicy）或者丢弃任务队列中最旧任务（DiscardOldestPolicy）。丢弃最旧任务也不是简单的丢弃最旧的任务，而是有一些额外的处理。除了丢弃任务还可以直接抛出一个异常（RejectedExecutionException），这是比较简单的方式。抛出异常的方式（AbortPolicy）尽管实现方式比

较简单，但是由于抛出一个 RuntimeException，因此会中断调用者的处理过程。除了抛出异常以外还可以不进入线程池执行，在这种方式（CallerRunsPolicy）中任务将有调用者线程去执行。

上面是一些理论知识，下面结合一些例子进行分析讨论。

```
package xylz.study.concurrency;
import java.lang.reflect.Field;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy;
import java.util.concurrent.ThreadPoolExecutor.DiscardPolicy;

public class ExecutorServiceDemo {
    static void log(String msg) {
        System.out.println(System.currentTimeMillis() + " -> " + msg);
    }

    static int getThreadPoolRunState(ThreadPoolExecutor pool) throws Exception {
        Field f = ThreadPoolExecutor.class.getDeclaredField("runState");
        f.setAccessible(true);
        int v = f.getInt(pool);
        return v;
    }

    public static void main(String[] args) throws Exception {
        ThreadPoolExecutor pool = new ThreadPoolExecutor(1, 1, 0, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(1));
        pool.setRejectedExecutionHandler(new ThreadPoolExecutor.DiscardPolicy());
        for (int i = 0; i < 10; i++) {
            final int index = i;
            pool.submit(new Runnable() {
                public void run() {
                    log("run task:" + index + " -> " + Thread.currentThread().getName());
                    try {
                        Thread.sleep(1000L);
```

```
                } catch (Exception e) {
                    e.printStackTrace();
                }
                log("run over:" + index + " -> " + Thread.currentThread().getName());
            }
        });
    }
    log("before sleep");
    Thread.sleep(4000L);
    log("before shutdown()");
    pool.shutdown();
    log("after shutdown(),pool.isTerminated=" + pool.isTerminated());
    pool.awaitTermination(1000L, TimeUnit.SECONDS);
    log("now,pool.isTerminated=" + pool.isTerminated() + ", state="+ getThreadPoolRunState(pool));
  }

}
```

第一种方式直接丢弃（DiscardPolicy）的输出结果是：

```
1294494050696 -> run task:0
1294494050696 -> before sleep
1294494051697 -> run over:0 -> pool-1-thread-1
1294494051697 -> run task:1
1294494052697 -> run over:1 -> pool-1-thread-1
1294494054697 -> before shutdown()
1294494054697 -> after shutdown(),pool.isTerminated=false
1294494054698 -> now,pool.isTerminated=true, state=3
```

对于上面的结果需要补充几点。

1. 线程池设定线程大小为 1，因此输出的线程就只有一个”pool-1-thread-1”，至于为什么是这个名称，以后会分析。
2. 任务队列的大小为 1，因此可以输出一个任务执行结果。但是由于线程本身可以带有一个任务，因此实际上一共执行了两个任务(task0 和 task1)。
3. shutdown()一个线程并不能理解是线程运行状态位 terminated，可能需要稍微等待一点时间。尽管这里等待时间参数是 1000 秒，但是实际上从输出时间来看仅仅等了约 1ms。
4. 直接丢弃任务是丢弃将要进入线程池本身的任务，所以当运行 task0 是，task1 进入任务队列，task2~task9 都被直接丢弃了，没有运行。

如果把策略换成丢弃最旧任务（DiscardOldestPolicy），结果会稍有不同。

```
1294494484622 -> run task:0
1294494484622 -> before sleep
1294494485622 -> run over:0 -> pool-1-thread-1
1294494485622 -> run task:9
1294494486622 -> run over:9 -> pool-1-thread-1
1294494488622 -> before shutdown()
1294494488622 -> after shutdown(),pool.isTerminated=false
1294494488623 -> now,pool.isTerminated=true, state=3
```

这里依然只是执行两个任务，但是换成了任务 task0 和 task9。实际上 task1~task8 还是进入了任务队列，只不过被 task9 挤出去了。

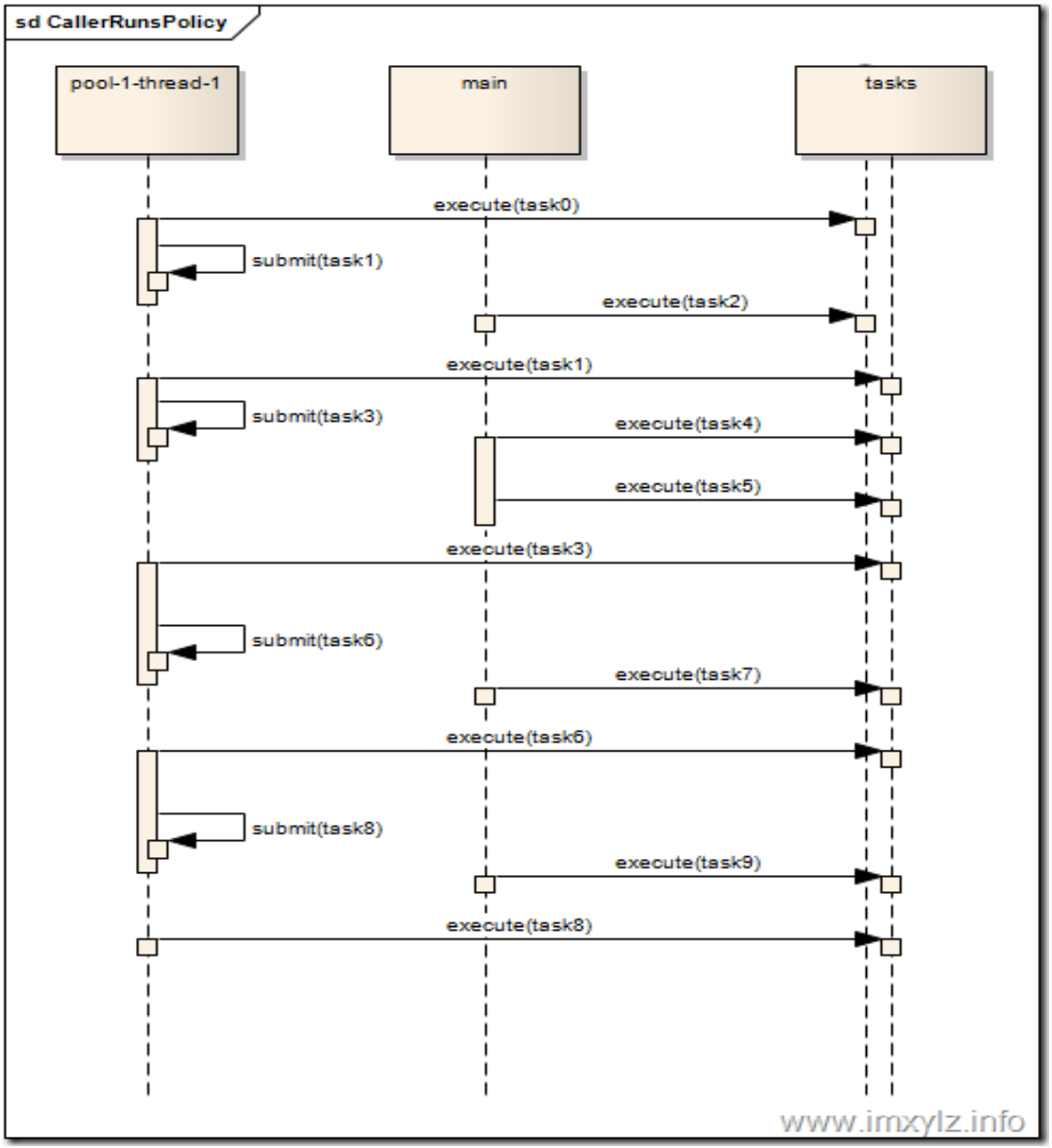
对于异常策略（AbortPolicy）就比较简单，这回调用线程的任务执行。

对于调用线程执行方式（CallerRunsPolicy），输出的结果就有意思了。

```
294496076266 -> run task:2 -> main
1294496076266 -> run task:0 -> pool-1-thread-1
1294496077266 -> run over:0 -> pool-1-thread-1
1294496077266 -> run task:1 -> pool-1-thread-1
1294496077266 -> run over:2 -> main
1294496077266 -> run task:4 -> main
1294496078267 -> run over:4 -> main
1294496078267 -> run task:5 -> main
1294496078267 -> run over:1 -> pool-1-thread-1
1294496078267 -> run task:3 -> pool-1-thread-1
1294496079267 -> run over:3 -> pool-1-thread-1
1294496079267 -> run over:5 -> main
1294496079267 -> run task:7 -> main
1294496079267 -> run task:6 -> pool-1-thread-1
1294496080267 -> run over:7 -> main
1294496080267 -> run task:9 -> main
1294496080267 -> run over:6 -> pool-1-thread-1
1294496080267 -> run task:8 -> pool-1-thread-1
1294496081268 -> run over:9 -> main
1294496081268 -> before sleep
1294496081268 -> run over:8 -> pool-1-thread-1
1294496085268 -> before shutdown()
```

```
1294496085268 -> after shutdown(),pool.isTerminated=false
1294496085269 -> now,pool.isTerminated=true, state=3
```

由于启动线程有稍微的延时，因此一种可能的执行顺序是这样的。



1. 首先 pool-1-thread-1 线程执行 task0,同时将 task1 加入任务队列（submit(task1)）。
2. 对于 task2，由于任务队列已经满了，因此有调用线程 main 执行（execute(task2)）。
3. 在 mian 等待 task2 任务执行完毕，对于任务 task3，由于此时任务队列已经空了，因此 task3 将进入任务队列。
4. 此时 main 线程是空闲的，因此对于 task4 将由 main 线程执行。此时 pool-1-thread-1 线程可能在执行任务 task1。任务队列中依然有任务 task3。
5. 因此 main 线程执行完毕 task4 后就立即执行 task5。
6. 很显然 task1 执行完毕，task3 被线程池执行，因此 task6 进入任务队列。此时 task7 被 main 线程执行。
7. task6 开始执行时，task8 进入任务队列。main 线程开始执行 task9。
8. 然后线程池执行线程 task8 结束。
9. 整个任务队列执行完毕，线程池完毕。

如果有兴趣可以看看 ThreadPoolExecutor 中四种 RejectedExecutionHandler 的源码，都非常简单。

深入浅出 Java Concurrency (32): 线程池 part 5 周期性任务调度

[本文地址：http://www.blogjava.net/Files/xylz/Inside.Java.Concurrency_32.ThreadPool.part5_ScheduledExecutorService.pdf]

周期性任务调度前世

在 JDK 5.0 之前，java.util.Timer/TimerTask 是唯一的内置任务调度方法，而且在很长一段时间里很热衷于使用这种方式进行周期性任务调度。

首先研究下 Timer/TimerTask 的特性（至于 javax.swing.Timer 就不再研究了）。

```
public void schedule(TimerTask task, long delay, long period) {
    if (delay < 0)  throw new IllegalArgumentException("Negative delay.");
    if (period <= 0) throw new IllegalArgumentException("Non-positive period.");
    sched (task, System.currentTimeMillis()+delay, -period);
}
public void scheduleAtFixedRate(TimerTask task, long delay, long period) {
    if (delay < 0)  throw new IllegalArgumentException("Negative delay.");
    if (period <= 0) throw new IllegalArgumentException("Non-positive period.");
    sched(task, System.currentTimeMillis()+delay, period);
}
```

```
public class Timer {
    private TaskQueue queue = new TaskQueue();
    /**The timer thread. */
    private TimerThread thread = new TimerThread(queue);
```

java.util.TimerThread.mainLoop()

```
private void mainLoop() {
    while (true) {
        try {
            TimerTask task;
            boolean taskFired;
            synchronized(queue) {
                // Wait for queue to become non-empty
                while (queue.isEmpty() && newTasksMayBeScheduled)
                    queue.wait();
                if (queue.isEmpty())
```

```
                    break; // Queue is empty and will forever remain; die
    。。。。。。

                if (!taskFired) // Task hasn't yet fired; wait
                    queue.wait(executionTime - currentTime);
            }
            if (taskFired)  // Task fired; run it, holding no locks
                task.run();
        } catch(InterruptedException e) {
        }
    }
}
```

上面三段代码反映了 Timer/TimerTask 的以下特性：

- Timer 对任务的调度是基于绝对时间的。
- 所有的 TimerTask 只有一个线程 TimerThread 来执行，因此同一时刻只有一个 TimerTask 在执行。
- 任何一个 TimerTask 的执行异常都会导致 Timer 终止所有任务。
- 由于基于绝对时间并且是单线程执行，因此在多个任务调度时，长时间执行的任务被执行后有可能导致短时间任务快速在短时间内被执行多次或者干脆丢弃多个任务。

由于 Timer/TimerTask 有这些特点（缺陷），因此这就导致了需要一个更加完善的任务调度框架来解决这些问题。

周期性任务调度今生

java.util.concurrent.ScheduledExecutorService 的出现正好弥补了 Timer/TimerTask 的缺陷。

```
public interface ScheduledExecutorService extends ExecutorService {
    public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
    public <V> ScheduledFuture<V> schedule(Callable<V> callable,  long delay, TimeUnit unit);
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);
    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);
}
```

首先 ScheduledExecutorService 基于 ExecutorService，是一个完整的线程池调度。另外在提供线程池的基础上增加了四个调度任务的 API。

- schedule(Runnable command,long delay, TimeUnit unit)：在指定的延迟时间一次性启动任务（Runnable），没有返回值。
- schedule(Callable<V> callable, long delay, TimeUnit unit)：在指定的延迟时间一次性启动任务（Callable），携带一个结果。
- **scheduleAtFixedRate**(Runnable command,long initialDelay,long period,TimeUnit unit)：建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期；也就是将在 initialDelay 后开始执行，然后在 initialDelay+period 后执行，接着在 initialDelay + 2 * period 后执行，依此类

推。如果任务的任何一个执行遇到异常，则后续执行都会被取消。否则，只能通过执行程序的取消或终止方法来终止该任务。如果此任务的任何一个执行要花费比其周期更长的时间，则将推迟后续执行，但不会同时执行。

- **scheduleWithFixedDelay**(Runnable command,long initialDelay,long delay,TimeUnit unit)：创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在每一次执行终止和下一次执行开始之间都存在给定的延迟。如果任务的任一执行遇到异常，就会取消后续执行。否则，只能通过执行程序的取消或终止方法来终止该任务。

上述 API 解决了以下几个问题：

- ScheduledExecutorService 任务调度是基于相对时间，不管是一次性任务还是周期性任务都是相对于任务加入线程池（任务队列）的时间偏移。
- 基于线程池的 ScheduledExecutorService 允许多个线程同时执行任务，这在添加多种不同调度类型的任务是非常有用的。
- 同样基于线程池的 ScheduledExecutorService 在其中一个任务发生异常时会退出执行线程，但同时会有新的线程补充进来进行执行。
- ScheduledExecutorService 可以做到不丢失任务。

下面的例子演示了一个任务周期性调度的例子。

```
package xylz.study.concurrency.executor;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorServiceDemo {
    public static void main(String[] args) throws Exception{
        ScheduledExecutorService execService =  Executors.newScheduledThreadPool(3);
        execService.scheduleAtFixedRate(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName()+" -> "+System.currentTimeMillis());
                try {
                    Thread.sleep(2000L);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, 1, 1, TimeUnit.SECONDS);
        //
        execService.scheduleWithFixedDelay(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName()+" -> "+System.currentTimeMillis());
```

```
                try {
                    Thread.sleep(2000L);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, 1, 1, TimeUnit.SECONDS);
        Thread.sleep(5000L);
        execService.shutdown();
    }
}
```

一次可能的输出如下：

```
pool-1-thread-1 -> 1294672392657
pool-1-thread-2 -> 1294672392659
pool-1-thread-1 -> 1294672394657
pool-1-thread-2 -> 1294672395659
pool-1-thread-1 -> 1294672396657
```

在这个例子中启动了默认三个线程的线程池，调度两个周期性任务。第一个任务是每隔 1 秒执行一次，第二个任务是以固定 1 秒的间隔执行，这两个任务每次执行的时间都是 2 秒。上面的输出有以下几点结论：

- 任务是在多线程中执行的，同一个任务应该是在同一个线程中执行。
- scheduleAtFixedRate 是每次相隔相同的时间执行任务，如果任务的执行时间比周期还长，那么下一个任务将立即执行。例如这里每次执行时间 2 秒，而周期时间只有 1 秒，那么每次任务开始执行的间隔时间就是 2 秒。
- scheduleWithFixedDelay 描述是下一个任务的开始时间与上一个任务的结束时间间隔相同。流入这里每次执行时间 2 秒，而周期时间是 1 秒，那么两个任务开始执行的间隔时间就是 2+1=3 秒。

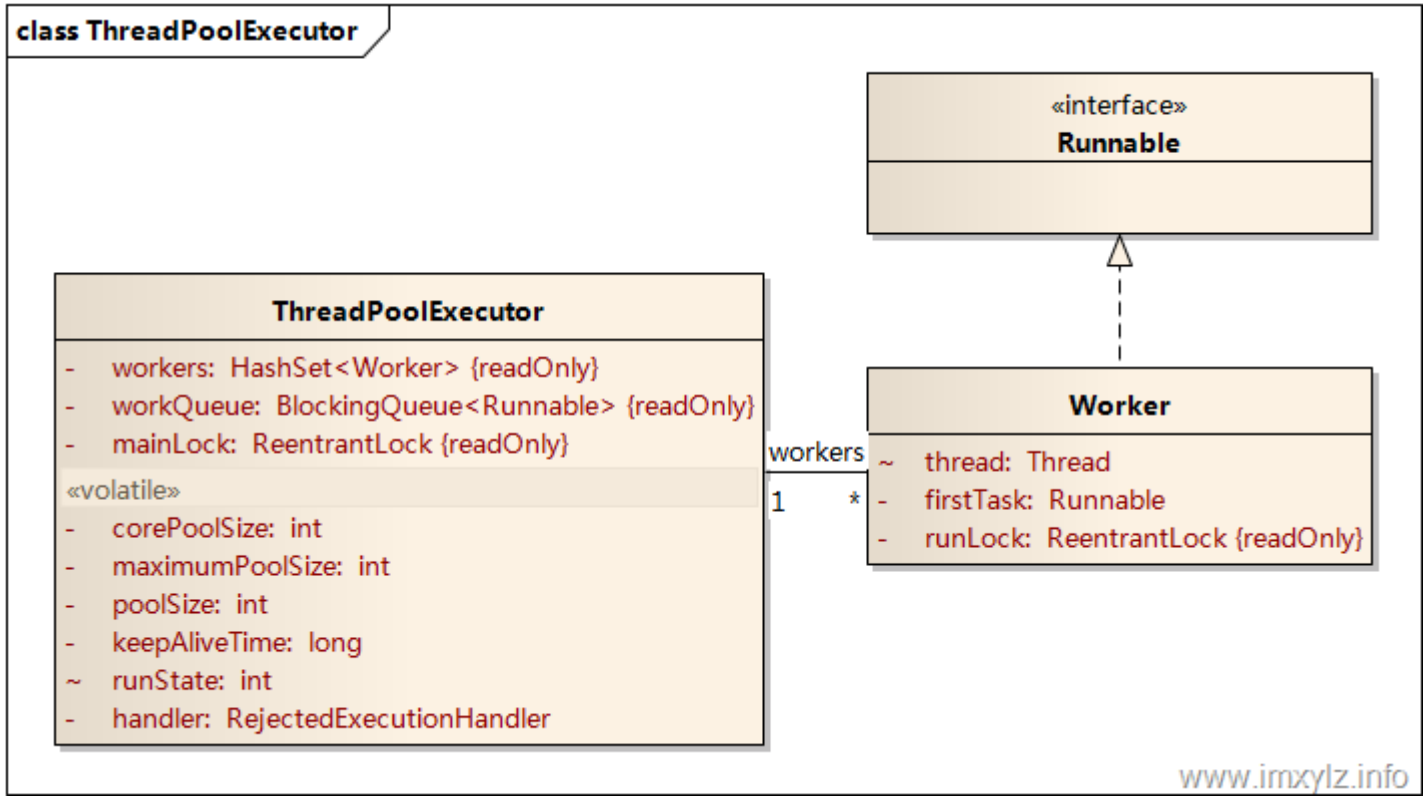
事实上 ScheduledExecutorService 的实现类 ScheduledThreadPoolExecutor 是继承线程池类 ThreadPoolExecutor 的，因此它拥有线程池的全部特性。但是同时又是一种特殊的线程池，这个线程池的线程数大小不限，任务队列是基于 DelayQueue 的无限任务队列。具体的结构和算法在以后的章节中分析。

由于 ScheduledExecutorService 拥有 Timer/TimerTask 的全部特性，并且使用更简单，支持并发，而且更安全，因此没有理由继续使用 Timer/TimerTask，完全可以全部替换。需要说明的一点事构造 ScheduledExecutorService 线程池的核心线程池大小要根据任务数来定，否则可能导致资源的浪费。

深入浅出 Java Concurrency (33): 线程池 part 6 线程池的实现及原理 (1)

线程池数据结构与线程构造方法

由于已经看到了 ThreadPoolExecutor 的源码，因此很容易就看到了 ThreadPoolExecutor 线程池的数据结构。图 1 描述了这种数据结构。



其实，即使没有上述图形描述 ThreadPoolExecutor 的数据结构，我们根据线程池的要求也很能够猜测出其数据结构出来。

- 线程池需要支持多个线程并发执行，因此有一个线程集合 Collection<Thread>来执行线程任务；
- 涉及任务的异步执行，因此需要有一个集合来缓存任务队列 Collection<Runnable>；
- 很显然在多个线程之间协调多个任务，那么就需要一个线程安全的任务集合，同时还需要支持阻塞、超时操作，那么 BlockingQueue 是必不可少的；
- 既然是线程池，出发点就是提高系统性能同时降低资源消耗，那么线程池的大小就有限制，因此需要有一个核心线程池大小（线程个数）和一个最大线程池大小（线程个数），有一个计数用来描述当前线程池大小；
- 如果是有限的线程池大小，那么长时间不使用的线程资源就应该销毁掉，这样就需要一个线程空闲时间的计数来描述线程何时被销毁；

- 前面描述过线程池也是有生命周期的，因此需要有一个状态来描述线程池当前的运行状态；
- 线程池的任务队列如果有边界，那么就需要有一个任务拒绝策略来处理过多的任务，同时在线程池的销毁阶段也需要有一个任务拒绝策略来处理新加入的任务；
- 上面种的线程池大小、线程空闲实际那、线程池运行状态等等状态改变都不是线程安全的，因此需要有一个全局的锁（mainLock）来协调这些竞争资源；
- 除了以上数据结构以外，ThreadPoolExecutor 还有一些状态用来描述线程池的运行计数，例如线程池运行的任务数、曾经达到的最大线程数，主要用于调试和性能分析。

对于 ThreadPoolExecutor 而言，一个线程就是一个 Worker 对象，它与一个线程绑定，当 Worker 执行完毕就是线程执行完毕，这个在后面详细讨论线程池中线程的运行方式。

既然是线程池，那么就首先研究下线程的构造方法。

```
public interface ThreadFactory {
   Thread newThread(Runnable r);
}
```

ThreadPoolExecutor 使用一个线程工厂来构造线程。线程池都是提交一个任务 Runnable，然后在某一个线程 Thread 中执行，ThreadFactory 负责如何创建一个新线程。

在 J.U.C 中有一个通用的线程工厂 java.util.concurrent.Executors.DefaultThreadFactory，它的构造方式如下：

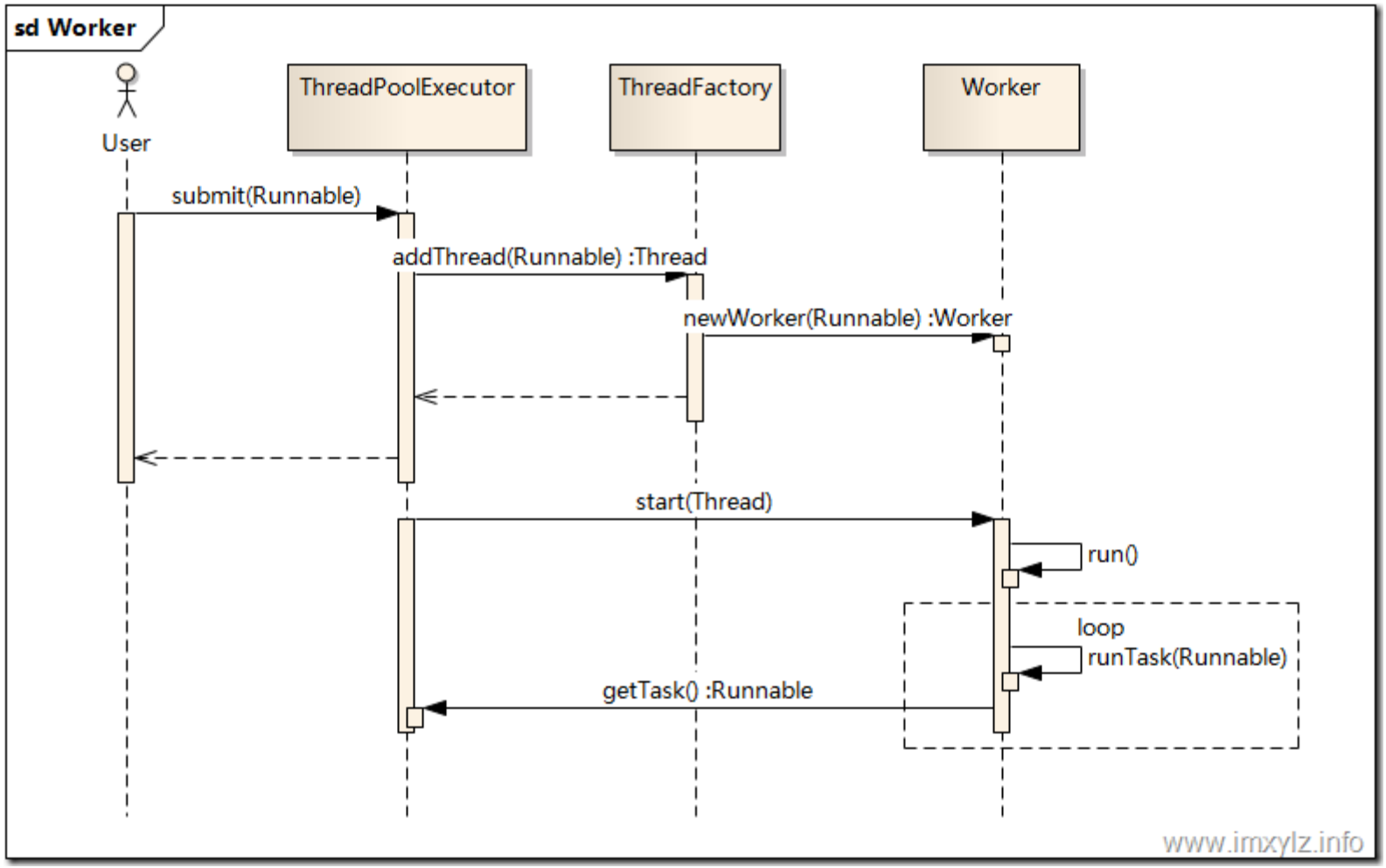
```
   static final AtomicInteger poolNumber = new AtomicInteger(1);
   final ThreadGroup group;
   final AtomicInteger threadNumber = new AtomicInteger(1);
   final String namePrefix;
   DefaultThreadFactory() {
      SecurityManager s = System.getSecurityManager();
      group = (s != null)? s.getThreadGroup() : Thread.currentThread().getThreadGroup();
      namePrefix = "pool-" +poolNumber.getAndIncrement() +"-thread-";
   }
   public Thread newThread(Runnable r) {
      Thread t = new Thread(group, r, namePrefix + threadNumber.getAndIncrement(),0);
      if (t.isDaemon())  t.setDaemon(false);
      if (t.getPriority() != Thread.NORM_PRIORITY) t.setPriority(Thread.NORM_PRIORITY);
      return t;
   }
}
```

在这个线程工厂中，同一个线程池的所有线程属于同一个线程组，也就是创建线程池的那个线程组，同时线程池的名称都是“pool-<poolNum>-thread-<threadNum>”，其中 poolNum 是线程池的数量序号，threadNum 是此线程池中的线程数量序号。这样如果使用 jstack 的话很容易就看到了系统中线程池的数量和线程池中线程的数量。另外对于线程池中的所有线程默认都转换为非后台线程，这样主线程退出时不会直接退出 JVM，而是等待线程池结束。还有一点就是默认将线程池中的所有线程都调为同一个级别，这样在操作系统角度来看所有系统都是公平的，不会导致竞争堆积。

线程池中线程生命周期

一个线程 Worker 被构造出来以后就开始处于运行状态。以下是一个线程执行的简版逻辑。

```
private final class Worker implements Runnable {
   private final ReentrantLock runLock = new ReentrantLock();
   private Runnable firstTask;
   Thread thread;
   Worker(Runnable firstTask) {
      this.firstTask = firstTask;
   }
   private void runTask(Runnable task) {
      final ReentrantLock runLock = this.runLock;
      runLock.lock();
      try {
        task.run();
      } finally {runLock.unlock();}
   }
   public void run() {
      try {
         Runnable task = firstTask;
         firstTask = null;
         while (task != null || (task = getTask()) != null) {
            runTask(task); task = null;
         }
      } finally {workerDone(this); }
   }
}
```



当提交一个任务时，如果需要创建一个线程（何时需要在下一节中探讨）时，就调用线程工厂创建一个线程，同时将线程绑定到 Worker 工作队列中。需要说明的是，Worker 队列构造的时候带着一个任务 Runnable，因此 Worker 创建时总是绑定着一个待执行任务。换句话说，创建线程的前提是有必要创建线程（任务数已经超出了线程或者强制创建新的线程，至于为何强制创建新的线程后面章节会具体分析），不会无缘无故创建一堆空闲线程等着任务。这是节省资源的一种方式。

一旦线程池启动线程后（调用线程 run()）方法，那么线程工作队列 Worker 就从第 1 个任务开始执行（这时候发现构造 Worker 时传递一个任务的好处了），一旦第 1 个任务执行完毕，就从线程池的任务队列中取出下一个任务进行执行。循环如此，直到线程池被关闭或者任务抛出了一个 RuntimeException。

由此可见，线程池的基本原理其实也很简单，无非预先启动一些线程，线程进入死循环状态，每次从任务队列中获取一个任务进行执行，直到线程池被关闭。如果某个线程因为执行某个任务发生异常而终止，那么重新创建一个新的线程而已。如此反复。

其实，线程池原理看起来简单，但是复杂的是各种策略，例如何时该启动一个线程，何时该终止、挂起、唤醒一个线程，任务队列的阻塞与超时，线程池的生命周期以及任务拒绝策略等等。下一节将研究这些策略问题。

深入浅出 Java Concurrency (34): 线程池 part 7 线程池的实现及原理 (2)

线程池任务执行流程

我们从一个 API 开始接触 Executor 是如何处理任务队列的。

java.util.concurrent.Executor.execute(Runnable)

Executes the given task sometime in the future. The task may execute in a new thread or in an existing pooled thread. If the task cannot be submitted for execution, either because this executor has been shutdown or because its capacity has been reached, the task is handled by the current RejectedExecutionHandler*.*

线程池中所有任务执行都依赖于此接口。这段话有以下几个意思：

1. 任务可能在将来某个时刻被执行，有可能不是立即执行。为什么这里有两个“可能”？继续往下面看。
2. 任务可能在一个新的线程中执行或者线程池中存在的一个线程中执行。
3. 任务无法被提交执行有以下两个原因：线程池已经关闭或者线程池已经达到了容量限制。
4. 所有失败的任务都将被“当前”的任务拒绝策略 RejectedExecutionHandler 处理。

回答上面两个“可能“。任务可能被执行，那不可能的情况就是上面说的情况 3；可能不是立即执行，是因为任务可能还在队列中排队，因此还在等待分配线程执行。了解完了字面上的问题，我们再来看具体的实现。

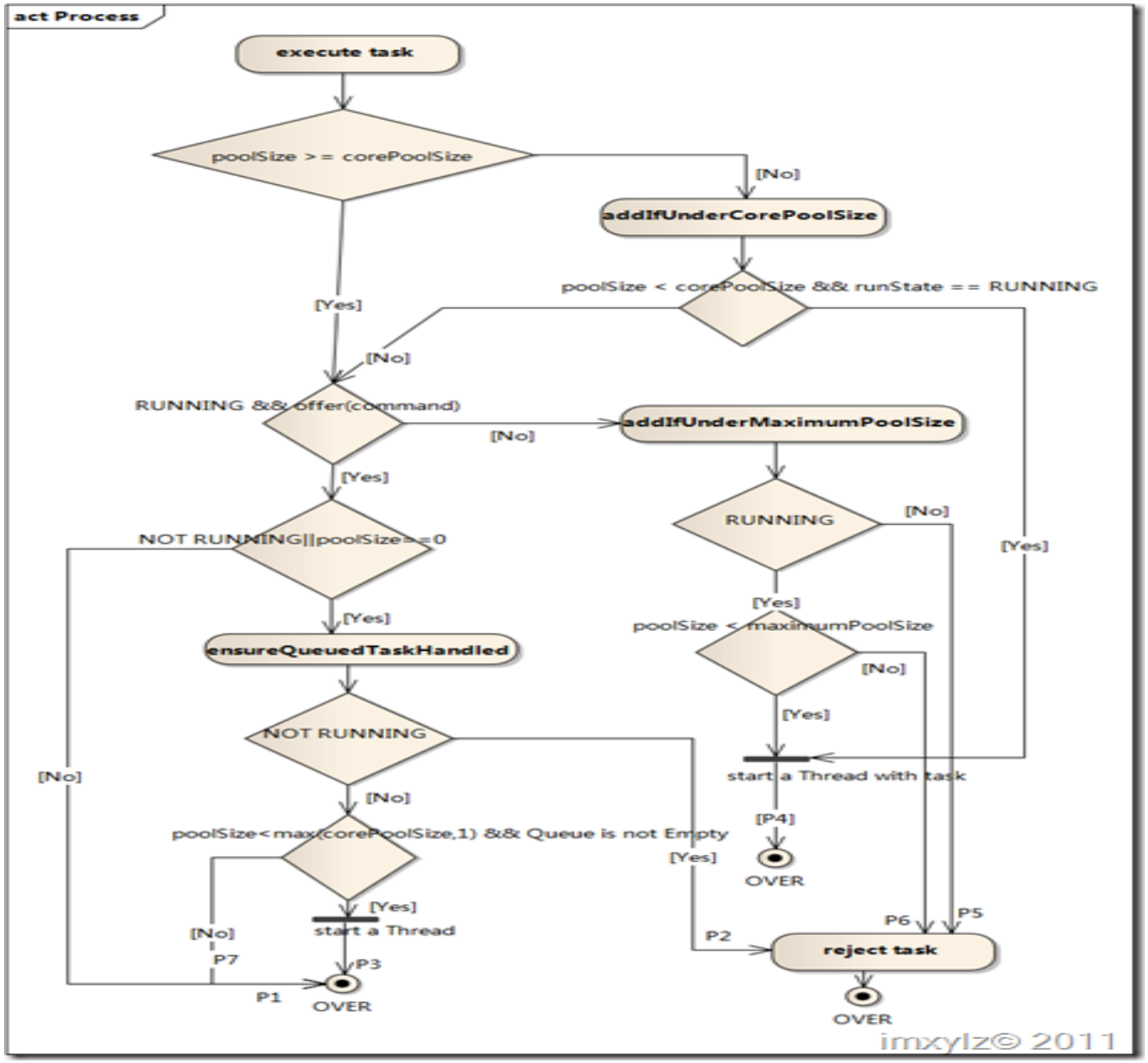
```
public void execute(Runnable command) {
   if (command == null)  throw new NullPointerException();
   if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
      if (runState == RUNNING && workQueue.offer(command)) {
         if (runState != RUNNING || poolSize == 0) ensureQueuedTaskHandled(command);
       }
      else if (!addIfUnderMaximumPoolSize(command)) reject(command); // is shutdown or saturated
   }
}
```

这一段代码看起来挺简单的，其实这就是线程池最重要的一部分，如果能够完全理解这一块，线程池还是挺容易的。整个执行流程是这样的：

1. 如果任务 command 为空，则抛出空指针异常，返回。否则进行 2。
2. 如果当前线程池大小 大于或等于 核心线程池大小，进行 4。否则进行 3。
3. 创建一个新工作队列（线程，参考上一节），成功直接返回，失败进行 4。
4. 如果线程池正在运行并且任务加入线程池队列成功，进行 5，否则进行 7。
5. 如果线程池已经关闭或者线程池大小为 0，进行 6，否则直接返回。
6. 如果线程池已经关闭则执行拒绝策略返回，否则启动一个新线程来进行执行任务，返回。

7．如果线程池大小 不大于 最大线程池数量，则启动新线程来进行执行，否则进行拒绝策略，结束。

文字描述步骤不够简单？下面图形详细表述了此过程。



老实说这个图比上面步骤更难以理解，那么从何入手呢。

流程的入口很简单，我们就是要执行一个任务（Runnable command），那么它的结束点在哪或者有哪几个？

根据左边这个图我们知道可能有以下几种出口：

（1）图中的 P1、P7，我们根据这条路径可以看到，仅仅是将任务加入任务队列（offer(command)）了；

（2）图中的 P3，这条路径不将任务加入任务队列，但是启动了一个新工作线程（Worker）进行扫尾操作，用户处理为空的任务队列；

（3）图中的 P4，这条路径没有将任务加入任务队列，但是启动了一个新工作线程（Worker），并且工作现场的第一个任务就是当前任务；

（4）图中的 P5、P6，这条路径没有将任务加入任务队列，也没有启动工作线程，仅仅是抛给了任务拒绝策略。P2 是任务加入了任务队列却因为线程池已经关闭于是又从任务队列中删除，并且抛给了拒绝策略。

如果上面的解释还不清楚，可以去研究下面两段代码：

```
java.util.concurrent.ThreadPoolExecutor.addIfUnderCorePoolSize(Runnable)
java.util.concurrent.ThreadPoolExecutor.addIfUnderMaximumPoolSize(Runnable)
java.util.concurrent.ThreadPoolExecutor.ensureQueuedTaskHandled(Runnable)
```

那么什么时候一个任务被立即执行呢？

在线程池运行状态下，如果线程池大小 小于 核心线程池大小或者线程池已满（任务队列已满）并且线程池大小 小于 最大线程池大小（此时线程池大小 大于 核心线程池大小的），用程序描述为：

```
runState == RUNNING && ( poolSize < corePoolSize || poolSize < maxnumPoolSize && workQueue.isFull())
```

上面的条件就是一个任务能够被立即执行的条件。有了 execute 的基础，我们看看 ExecutorService 中的几个 submit 方法的实现。

```
 public Future<?> submit(Runnable task) {
       if (task == null) throw new NullPointerException();
       RunnableFuture<Object> ftask = newTaskFor(task, null);
       execute(ftask);
       return ftask;
    }
    public <T> Future<T> submit(Runnable task, T result) {
       if (task == null) throw new NullPointerException();
       RunnableFuture<T> ftask = newTaskFor(task, result);
       execute(ftask);
       return ftask;
    }
    public <T> Future<T> submit(Callable<T> task) {
       if (task == null) throw new NullPointerException();
       RunnableFuture<T> ftask = newTaskFor(task);
```

```
        execute(ftask);
        return ftask;
    }
```

很简单，不是么？对于一个线程池来说复杂的地方也就在 execute 方法的执行流程。在下一节中我们来讨论下如何获取任务的执行结果，也就是 Future 类的使用和原理。

深入浅出 Java Concurrency (35): 线程池 part 8 线程池的实现及原理 (3)

线程池任务执行结果

这一节来探讨下线程池中任务执行的结果以及如何阻塞线程、取消任务等等。

```
package info.imxylz.study.concurrency.future;
public class SleepForResultDemo implements Runnable {
    static boolean result = false;
    static void sleepWhile(long ms) {
        try {
            Thread.sleep(ms);
        } catch (Exception e) {}
    }
    public void run() {
        //do work
        System.out.println("Hello, sleep a while.");
        sleepWhile(2000L);
        result = true;
    }
    public static void main(String[] args) {
        SleepForResultDemo demo = new SleepForResultDemo();
        Thread t = new Thread(demo);
        t.start();
        sleepWhile(3000L);
        System.out.println(result);
    }
```

```
}
```

在没有线程池的时代里面，使用 Thread.sleep(long)去获取线程执行完毕的场景很多。显然这种方式很笨拙，他需要你事先知道任务可能的执行时间，并且还会阻塞主线程，不管任务有没有执行完毕。

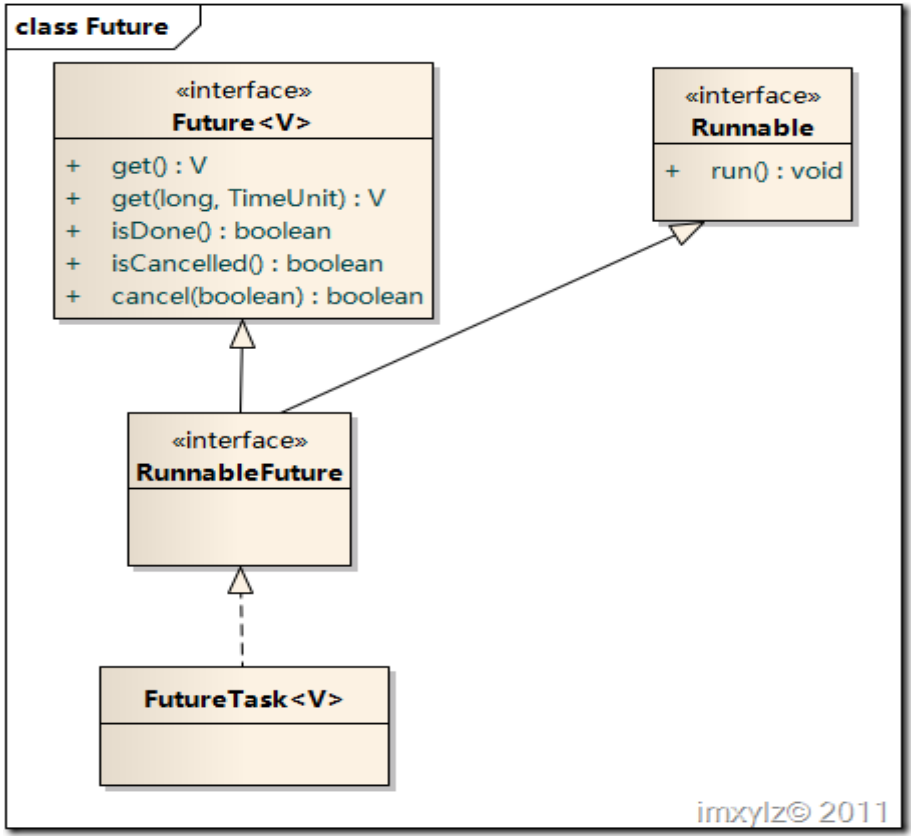
```
package info.imxylz.study.concurrency.future;
 public class SleepLoopForResultDemo implements Runnable {
     boolean result = false;
     volatile boolean finished = false;
     static void sleepWhile(long ms) {
        try {
            Thread.sleep(ms);
         } catch (Exception e) {}
    }
    public void run() {
        //do work
       try {
            System.out.println("Hello, sleep a while.");
            sleepWhile(2000L);
            result = true;
         } finally {
            finished = true;
         }
    }
     public static void main(String[] args) {
       SleepLoopForResultDemo demo = new SleepLoopForResultDemo();
        Thread t = new Thread(demo);
        t.start();
        while (!demo.finished) {
            sleepWhile(10L);
        }
        System.out.println(demo.result);
    }
 }
```

使用 volatile 与 while 死循环的好处就是等待的时间可以稍微小一点，但是依然有 CPU 负载高并且阻塞主线程的问题。最简单的降低 CPU 负载的方式就是使用 Thread.join().

```
SleepLoopForResultDemo demo = new SleepLoopForResultDemo();
        Thread t = new Thread(demo);
        t.start();
        t.join();
        System.out.println(demo.result);
```

显然这也是一种不错的方式，另外还有自己写锁使用 wait/notify 的方式。其实 join()从本质上讲就是利用 while 和 wait 来实现的。

上面的方式中都存在一个问题，那就是会阻塞主线程并且任务不能被取消。为了解决这个问题，线程池中提供了一个 Future 接口。



在 Future 接口中提供了 5 个方法。

- V get() throws InterruptedException, ExecutionException： 等待计算完成，然后获取其结果。

- V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException。最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。
- boolean cancel(boolean mayInterruptIfRunning)：试图取消对此任务的执行。
- boolean isCancelled()：如果在任务正常完成前将其取消，则返回 true。
- boolean isDone()：如果任务已完成，则返回 true。 可能由于正常终止、异常或取消而完成，在所有这些情况中，此方法都将返回 true。

API 看起来容易，来研究下异常吧。get()请求获取一个结果会阻塞当前进程，并且可能抛出以下三种异常：

- InterruptedException：执行任务的线程被中断则会抛出此异常，此时不能知道任务是否执行完毕，因此其结果是无用的，必须处理此异常。
- ExecutionException：任务执行过程中(Runnable#run()）方法可能抛出 RuntimeException，如果提交的是一个 java.util.concurrent.Callable<V>接口任务，那么 java.util.concurrent.Callable.call()方法有可能抛出任意异常。
- CancellationException：实际上 get()方法还可能抛出一个 CancellationException 的 RuntimeException，也就是任务被取消了但是依然去获取结果。

对于 get(long timeout, TimeUnit unit)而言，除了 get()方法的异常外，由于有超时机制，因此还可能得到一个 TimeoutException。

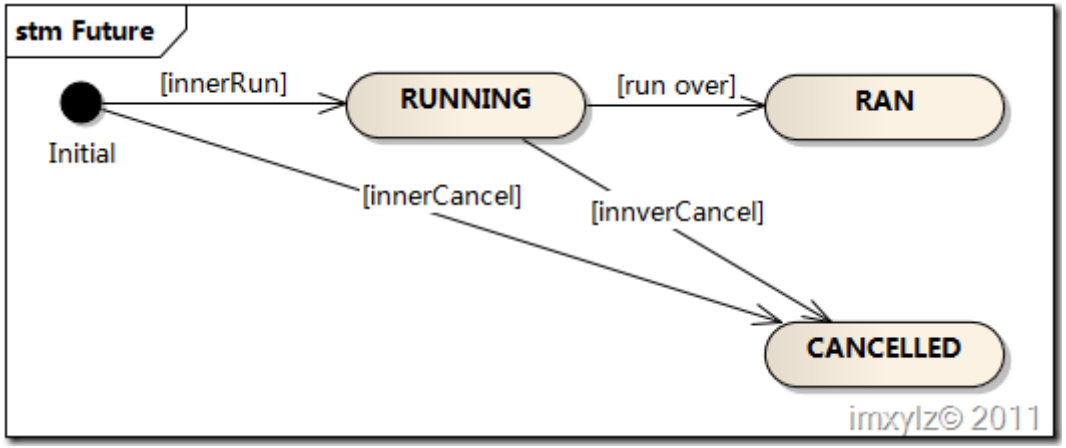
boolean cancel(boolean mayInterruptIfRunning)方法比较复杂，各种情况比较多：

1. 如果任务已经执行完毕，那么返回 false。
2. 如果任务已经取消，那么返回 false。
3. 循环直到设置任务为取消状态，对于未启动的任务将永远不再执行，对于正在运行的任务，将根据 mayInterruptIfRunning 是否中断其运行，如果不中断那么任务将继续运行直到结束。
4. 此方法返回后任务要么处于运行结束状态，要么处于取消状态。isDone()将永远返回 true，如果 cancel()方法返回 true，isCancelled()始终返回 true。

来看看 Future 接口的实现类 java.util.concurrent.FutureTask<V>具体是如何操作的。

在FutureTask中使用了一个AQS数据结构来完成各种状态以及加锁、阻塞的实现。

在此 AQS 类 java.util.concurrent.FutureTask.Sync 中一个任务用 4 中状态：



初始情况下任务状态 state=0，任务执行(innerRun)后状态变为运行状态 RUNNING(state=1)，执行完毕后变成运行结束状态 RAN(state=2)。任务在初始状态或者执行状态被取消后就变为状态 CANCELLED(state=4)。AQS 最擅长无锁情况下处理几种简单的状态变更的。

```
void innerRun() {
        if (!compareAndSetState(0, RUNNING))
            return;
        try {
            runner = Thread.currentThread();
            if (getState() == RUNNING)  // recheck after setting thread
                    innerSet(callable.call());
            else
                    releaseShared(0); // cancel
         } catch (Throwable ex) {
            innerSetException(ex);
         }
      }
```

执行一个任务有四步：设置运行状态、设置当前线程（AQS 需要）、执行任务(Runnable#run 或者 Callable#call）、设置执行结果。这里也可以看到，一个任务只能执行一次，因为执行完毕后它的状态不在为初始值 0，要么为 CANCELLED，要么为 RAN。

取消一个任务(cancel)又是怎样进行的呢？对比下前面取消任务的描述是不是很简单，这里无非利用 AQS 的状态来改变任务的执行状态，最终达到放弃未启动或者正在执行的任务的目的。

```
boolean innerCancel(boolean mayInterruptIfRunning) {
   for (;;) {
      int s = getState();
      if (ranOrCancelled(s))  return false;
      if (compareAndSetState(s, CANCELLED)) break;
   }
   if (mayInterruptIfRunning) {
      Thread r = runner;
      if (r != null) r.interrupt();
   }
   releaseShared(0);
   done();
   return true;
}
```

到目前为止我们依然没有说明到底是如何阻塞获取一个结果的。下面四段代码描述了这个过程。

```
V innerGet() throws InterruptedException, ExecutionException {
        acquireSharedInterruptibly(0);
        if (getState() == CANCELLED)  throw new CancellationException();
        if (exception != null)  throw new ExecutionException(exception);
        return result;
    }
    //AQS#acquireSharedInterruptibly
    public final void acquireSharedInterruptibly(int arg) throws InterruptedException {
        if (Thread.interrupted())  throw new InterruptedException();
        if (tryAcquireShared(arg) < 0) doAcquireSharedInterruptibly(arg); //park current Thread for result
    }
    protected int tryAcquireShared(int ignore) {
        return innerIsDone()? 1 : -1;
    }
    boolean innerIsDone() {
        return ranOrCancelled(getState()) && runner == null;
    }
```

当调用 Future#get()的时候尝试去获取一个共享变量。这就涉及到 AQS 的使用方式了。这里获取一个共享变量的状态是任务是否结束(innerIsDone())，也就是任务是否执行完毕或者被取消。如果不满足条件，那么在 AQS 中就会 doAcquireSharedInterruptibly(arg)挂起当前线程，直到满足条件。AQS 前面讲过，挂起线程使用的是 LockSupport 的 park 方式，因此性能消耗是很低的。

至于将 Runnable 接口转换成 Callable 接口，java.util.concurrent.Executors.callable(Runnable, T)也提供了一个简单实现。

```
static final class RunnableAdapter<T> implements Callable<T> {
      final Runnable task;
      final T result;
      RunnableAdapter(Runnable  task, T result) {
        this.task = task;
        this.result = result;
      }
      public T call() {
        task.run();
        return result;
      }
```

```
}
```

延迟、周期性任务调度的实现

java.util.concurrent.ScheduledThreadPoolExecutor 是默认的延迟、周期性任务调度的实现。

有了整个线程池的实现，再回头来看延迟、周期性任务调度的实现应该就很简单了，因为所谓的延迟、周期性任务调度，无非添加一系列有序的任务队列，然后按照执行顺序的先后来处理整个任务队列。如果是周期性任务，那么在执行完毕的时候加入下一个时间点的任务即可。

由此可见，ScheduledThreadPoolExecutor 和 ThreadPoolExecutor 的唯一区别在于任务是有序（按照执行时间顺序）的，并且需要到达时间点（临界点）才能执行，并不是任务队列中有任务就需要执行的。也就是说唯一不同的就是任务队列 BlockingQueue<Runnable> workQueue 不一样。ScheduledThreadPoolExecutor 的任务队列是 java.util.concurrent.ScheduledThreadPoolExecutor.DelayedWorkQueue，它是基于 java.util.concurrent.DelayQueue<RunnableScheduledFuture>队列的实现。

DelayQueue是基于有序队列PriorityQueue实现的。PriorityQueue 也叫优先级队列，按照自然顺序对元素进行排序，类似于TreeMap/Collections.sort一样。

同样是有序队列，DelayQueue和PriorityQueue区别在什么地方？

由于 DelayQueue 在获取元素时需要检测元素是否“可用”，也就是任务是否达到“临界点”（指定时间点），因此加入元素和移除元素会有一些额外的操作。

典型的，移除元素需要检测元素是否达到“临界点”，增加元素的时候如果有一个元素比“头元素”更早达到临界点，那么就需要通知任务队列。因此这需要一个条件变量 final Condition available 。

移除元素（出队列）的过程是这样的：

- 总是检测队列的头元素（顺序最小元素，也是最先达到临界点的元素）
- 检测头元素与当前时间的差，如果大于 0，表示还未到底临界点，因此等待响应时间（使用条件变量 available)
- 如果小于或者等于 0，说明已经到底临界点或者已经过了临界点，那么就移除头元素，并且唤醒其它等待任务队列的线程。

```
public E take() throws InterruptedException {
      final ReentrantLock lock = this.lock;
       lock.lockInterruptibly();
      try {
         for (;;) {
            E first = q.peek();
           if (first == null) { available.await();
            } else {
              long delay =  first.getDelay(TimeUnit.NANOSECONDS);
              if (delay > 0) {  long tl = available.awaitNanos(delay);
               } else {
                  E x = q.poll();
                  assert x != null;
```

```
                if (q.size() != 0) available.signalAll(); // wake up other takers
                return x;
            }
        }
    }
} finally {
    lock.unlock();
}
}
```

同样加入元素也会有相应的条件变量操作。当前仅当队列为空或者要加入的元素比队列中的头元素还小的时候才需要唤醒“等待线程”去检测元素。因为头元素都没有唤醒那么比头元素更延迟的元素就更加不会唤醒。

```
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        E first = q.peek();
        q.offer(e);
        if (first == null || e.compareTo(first) < 0) available.signalAll();
        return true;
    } finally {
        lock.unlock();
    }
}
```

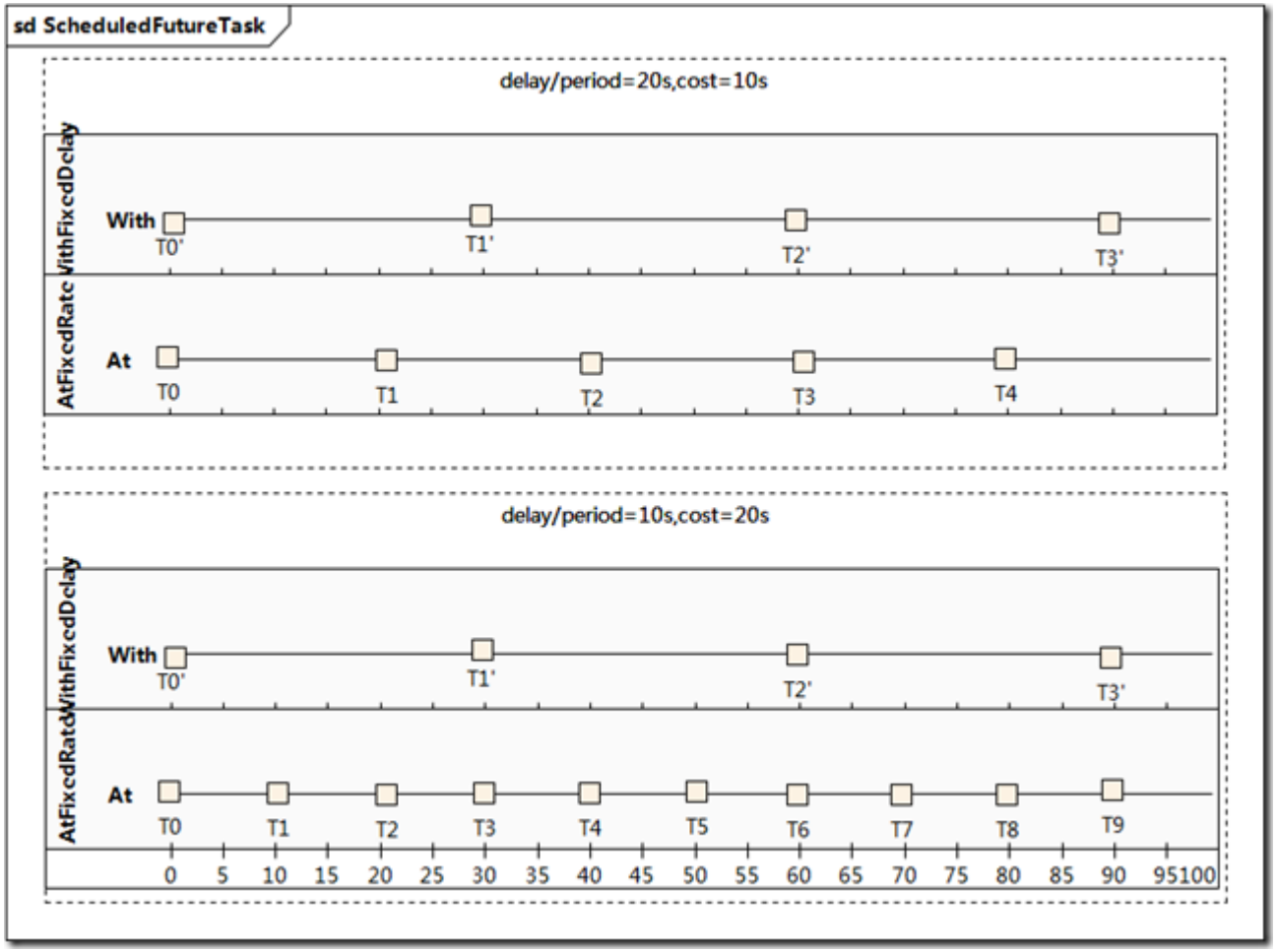
有了任务队列后再来看 Future 在 ScheduledThreadPoolExecutor 中是如何操作的。

java.util.concurrent.ScheduledThreadPoolExecutor.ScheduledFutureTask<V>是继承 java.util.concurrent.FutureTask<V>的，区别在于执行任务是否是周期性的。

```
private void runPeriodic() {
        boolean ok = ScheduledFutureTask.super.runAndReset();
        boolean down = isShutdown();
        // Reschedule if not cancelled and not shutdown or policy allows
        if (ok && (!down || (getContinueExistingPeriodicTasksAfterShutdownPolicy() &&  !isStopped()))) {
            long p = period;
```

```
            if (p > 0)   time += p;
            else time = now() - p;
             ScheduledThreadPoolExecutor.super.getQueue().add(this);
        }
        // This might have been the final executed delayed
        // task.  Wake up threads to check.
        else if (down) interruptIdleWorkers();
    }
    /**Overrides FutureTask version so as to reset/requeue if periodic. */
    public void run() {
        if (isPeriodic()) runPeriodic();
        else   ScheduledFutureTask.super.run();
    }
}
```

如果不是周期性任务调度，那么就和 java.util.concurrent.FutureTask.Sync 的调度方式是一样的。如果是周期性任务（isPeriodic()）那么就稍微有所不同的。



先从功能/结构上分析下。第一种情况假设提交的任务每次执行花费 10s，间隔（delay/period)为 20s，对于scheduleAtFixedRate而言，每次执行开始时间 20s，对于scheduleWithFixedDelay来说每次执行开始时间 30s。第二种情况假设提交的任务每次执行时间花费 20s，间隔（delay/period)为 10s，对于scheduleAtFixedRate而言，每次执行开始时间 10s，对于scheduleWithFixedDelay来说每次执行开始时间 30s。（具体分析可以参考这里）

也就是说 scheduleWithFixedDelay 的执行开始时间为(delay+cost)，而对于 scheduleAtFixedRate 来说执行开始时间为 max(period,cost)。

回头再来看上面源码 runPeriodic()就很容易了。但特别要提醒的，如果任务的任何一个执行遇到异常，则后续执行都会被取消，这从 runPeriodic()就能看出。要强调的第二点就是**同一个周期性任务不会被同时执行**。就比如说尽管上面第二种情况的 scheduleAtFixedRate 任务每隔 10s 执行到达一个时间点，但是由于每次执行时间花费为 20s，因此每次执行间隔为 20s，只不过执行的任务次数会多一点。但从本质上讲就是每隔 20s 执行一次，如果任务队列不取消的话。

为什么不会同时执行？

这是因为 ScheduledFutureTask 执行的时候会将任务从队列中移除来，执行完毕以后才会添加下一个同序列的任务，因此任务队列中其实最多只有同序列的任务的一份副本，所以永远不会同时执行（尽管要执行的时间在过去）。

ScheduledThreadPoolExecutor使用一个无界（容量无限，整数的最大值）的容器（DelayedWorkQueue队列），根据ThreadPoolExecutor的原理，只要当容器满的时候才会启动一个大于corePoolSize的线程数。因此实际上ScheduledThreadPoolExecutor是一个固定线程大小的线程池，固定大小为corePoolSize，构造函数里面的Integer.MAX_VALUE其实是不生效的（尽管PriorityQueue使用数组实现有PriorityQueue大小限制，如果你的任务数超过了 2147483647 就会导致OutOfMemoryError，这个参考PriorityQueue的grow方法）。

再回头看scheduleAtFixedRate等方法就容易多了。无非就是往任务队列中添加一个未来某一时刻的ScheduledFutureTask任务，如果是scheduleAtFixedRate那么period/delay就是正数，如果是scheduleWithFixedDelay那么period/delay就是一个负数，如果是 0 那么就是一次性任务。直接调用父类ThreadPoolExecutor的execute/submit等方法就相当于period/delay是 0，并且initialDelay也是 0。

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,  long period,  TimeUnit unit) {
      if (command == null || unit == null)  throw new NullPointerException();
      if (period <= 0) throw new IllegalArgumentException();
      if (initialDelay < 0) initialDelay = 0;
      long triggerTime = now() + unit.toNanos(initialDelay);
      RunnableScheduledFuture<?> t = decorateTask(command,
                                          new ScheduledFutureTask<Object>(command,null,  triggerTime, unit.toNanos(period)));
      delayedExecute(t);
      return t;
   }
```

另外需要补充说明的一点，前面说过 java.util.concurrent.FutureTask.Sync 任务只能执行一次，那么在 runPeriodic()里面怎么又将执行过的任务加入队列中呢？这是因为 java.util.concurrent.FutureTask.Sync 提供了一个 innerRunAndReset()方法，此方法不仅执行任务还将任务的状态还原成 0（初始状态）了，所以此任务就可以重复执行。这就是为什么 runPeriodic()里面调用 runAndRest()的缘故。

```
boolean innerRunAndReset() {
        if (!compareAndSetState(0, RUNNING))  return false;
        try {
            runner = Thread.currentThread();
            if (getState() == RUNNING) callable.call(); // don't set result
            runner = null;
            return compareAndSetState(RUNNING, 0);
        } catch (Throwable ex) {
            innerSetException(ex);
            return false;
```

```
        }
    }
```

后话

整个并发实践原理和实现（源码）上的东西都讲完了，后面几个小节是一些总结和扫尾的工作，包括超时机制、异常处理等一些细节问题。也就是说大部分只需要搬出一些理论和最佳实践知识出来就好了，不会有大量费脑筋的算法分析和原理、思想探讨之类的。后面的章节也会加快一些进度。

老实说从刚开始的好奇到中间的兴奋，再到现在的彻悟，收获还是很多，个人觉得这是最认真、最努力也是自我最满意的一次技术研究和探讨，同时在这个过程中将很多技术细节都串联起来了，慢慢就有了那种技术相通的感觉。原来有了理论以后再去实践、再去分析问题、解决问题和那种纯解决问题得到的经验完全不一样。整个专辑下来不仅仅是并发包这一点点知识，设计到硬件、软件、操作系统、网络、安全、性能、算法、理论等等，总的来说这也算是一次比较成功的研究切入点，这比Guice那次探讨要深入和持久的多。