1.1 线程

1.1.1进程和线程

进程

进程：进程指正在运行的程序，进程拥有一个完整的、私有的基本运行资源集合。通常，每个进程都有自己的内存空间。

进程往往被看作是程序或应用的代名词，然而，用户看到的一个单独的应用程序实际上可能是一组相互协作的进程集合。

为了便于进程之间的通信，大多数操作系统都支持进程间通信（IPC），如pipes 和sockets。IPC不仅支持同一系统上的通信，也支持不同的系统。IPC通信方式包括管道（包括无名管道和命名管道）、消息队列、信号量、共享存储、Socket、Streams等方式，其中 Socket和Streams支持不同主机上的两个进程IPC。

线程

线程有时也被称为轻量级的进程。进程和线程都提供了一个执行环境，但创建一个新的线程比创建一个新的进程需要的资源要少。

线程是在进程中存在的 — 每个进程最少有一个线程。线程共享进程的资源，包括内存和打开的文件。这样提高了效率，但潜在的问题就是线程间的通信。

多线程的执行是Java平台的一个基本特征。每个应用都至少有一个线程 – 或几个，如果算上“系统”线程的话，比如内存管理和信号处理等。但是从程序员的角度来看，启动的只有一个线程，叫主线程。

简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程。

1.1.2 线程实践

1.1.2.1 线程的创建

两种方式：

```
1  public class HelloThread extends Thread {
2
3      public void run() {
4          System.out.println("Hello from a thread!");
5      }
6
7      public static void main(String args[]) {
8          (new HelloThread()).start();
9      }
10
11 }
```

方式2：

```java
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

1.1.2.2 线程启动和停止

启动线程

调用start方法

停止线程

线程自带的stop方法，一方面已经过时，另一方面，不会对停止的线程做状态保存，使得线程中涉及的对象处于未知状态，如果这些状态，其他线程也会使用，将会使得其他线程出现无法预料的异常，所以，停止程序的功能，需要自己实现。

```java

public class ThreadTest {
    public static void main(String[] args) throws InterruptedException {
        StopThread thread = new StopThread();
        thread.start();
        Thread.sleep(1000L);
        thread.stop();
        while (thread.isAlive()) { }
        thread.print();
    }

    private static class StopThread extends Thread {

        private int x = 0; private int y = 0;

        @Override
        public void run() {
            synchronized (this) {
                ++x;
                try {
                    Thread.sleep(3000L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                ++y;
            }
        }

        public void print() {
            System.out.println("x=" + x + " y=" + y);
        }
    }
}
```

上述代码中，run方法里是一个同步的原子操作，x和y必须要共同增加，然而这里如果调用thread.stop()方法强制中断线程，输出如下：

```
x=1 y=0
```

没有异常，也破坏了我们的预期。如果这种问题出现在我们的程序中，会引发难以预期的异常。因此这种不安全的方式很早就被废弃了。

```
public class MyRunnable implements Runnable {

    private boolean doStop = false;

    public synchronized void doStop() {
        this.doStop = true;
    }

    private synchronized boolean keepRunning() {
        return this.doStop == false;
    }

    @Override
    public void run() {
        while(keepRunning()) {
            // keep doing what this thread should do.
            System.out.println("Running");

            try {
                Thread.sleep(3L * 1000L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    }
}
```

调用

```
public class MyRunnableMain {

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();

        Thread thread = new Thread(myRunnable);

        thread.start();

        try {
            Thread.sleep(10L * 1000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        myRunnable.doStop();
    }
}
```

1.1.2.3 线程暂停和中断

暂停

Java中线程的暂停是调用`java.lang.Thread`类的`sleep`方法。该方法会使当前正在执行的线程暂停指定的一段时间，如果线程持有锁，`sleep`方法结束前并不会释放该锁。

中断

java.lang.Thread类有一个`interrupt`方法，该方法直接对线程调用。当被interrupt的线程正在sleep或wait时，会抛出`InterruptedException`异常。
事实上，`interrupt`方法只是改变目标线程的中断状态（interrupt status），而那些会抛出`InterruptedException`异常的方法，如wait、sleep、join等，都是在方法内部不断地检查中断状态的值。

- interrupt方法

Thread实例方法：必须由其它线程获取被调用线程的实例后，进行调用。实际上，只是改变了被调用线程的内部中断状态；

- Thread.interrupted方法

Thread类方法：必须在当前执行线程内调用，该方法返回当前线程的内部中断状态，然后清除中断状态（置为false）；

- isInterrupted方法

Thread实例方法：用来检查指定线程的中断状态。当线程为中断状态时，会返回true；否则返回false。

```
public class ThreadTest {
    public static void main(String[] args) throws InterruptedException {
        StopThread thread = new StopThread();
        thread.start();
        Thread.sleep(1000L);
        thread.interrupt();
        while (thread.isAlive()) { }
        thread.print();
    }

    private static class StopThread extends Thread {

        private int x = 0; private int y = 0;

        @Override
        public void run() {
            synchronized (this) {
                ++x;
                try {
                    Thread.sleep(3000L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                ++y;
            }
        }

        public void print() {
```

```
            System.out.println("x=" + x + " y=" + y);
        }
    }
}
```

输出结果如下：

```
x=1 y=1
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at ThreadTest$StopThread.run(ThreadTest.java:28)
```

x=1,y=1 这个结果是符合我们的预期，同时还抛出了个异常。

底层源码实现：

```
    // 核心 interrupt 方法
    public void interrupt() {
        if (this != Thread.currentThread()) // 非本线程，需要检查权限
            checkAccess();

        synchronized (blockerLock) {
            Interruptible b = blocker;
            if (b != null) {
                interrupt0();           // 仅仅设置interrupt标志位
                b.interrupt(this);    // 调用如 I/O 操作定义的中断方法
                return;
            }
        }
        interrupt0();
    }
    // 静态方法，这个方法有点坑，调用该方法调用后会清除中断状态。
    public static boolean interrupted() {
        return currentThread().isInterrupted(true);
    }
    // 这个方法不会清除中断状态
    public boolean isInterrupted() {
        return isInterrupted(false);
    }
    // 上面两个方法会调用这个本地方法，参数代表是否清除中断状态
    private native boolean isInterrupted(boolean ClearInterrupted);
```

interrupt() ：

- interrupt 中断操作时，非自身打断需要先检测是否有中断权限，这由jvm的安全机制配置；
- 如果线程处于sleep, wait, join 等状态，那么线程将立即退出被阻塞状态，并抛出一个InterruptedException异常；
- 如果线程处于I/O阻塞状态，将会抛出ClosedByInterruptException（IOException的子类）异常；
- 如果线程在Selector上被阻塞，select方法将立即返回；
- 如果非以上情况，将直接标记 interrupt 状态；

注意：interrupt 操作不会打断所有阻塞，只有上述阻塞情况才在jvm的打断范围内，如处于锁阻塞的线程，不会受 interrupt 中断；

阻塞情况下中断，抛出异常后线程恢复非中断状态，即 interrupted = false

```
public class ThreadTest {

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Task("mytask"));
        t.start();
        t.interrupt();
    }

    static class Task implements Runnable{
        String name;

        public Task(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("thread has been interrupt!");
            }
            System.out.println("isInterrupted: " +
Thread.currentThread().isInterrupted());
            System.out.println("task " + name + " is over");
        }
    }
}
```

输出：

```
thread has been interrupt!
isInterrupted: false
task 1 is over
```

调用Thread.interrupted() 方法后线程恢复非中断状态

```
public class ThreadTest {

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Task("mytask"));
        t.start();
        t.interrupt();
    }

    static class Task implements Runnable{
        String name;

        public Task(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            System.out.println("first :" + Thread.interrupted());
            System.out.println("second:" + Thread.interrupted());
```

```
            System.out.println("task " + name + " is over");
        }
    }
}
```

输出结果：

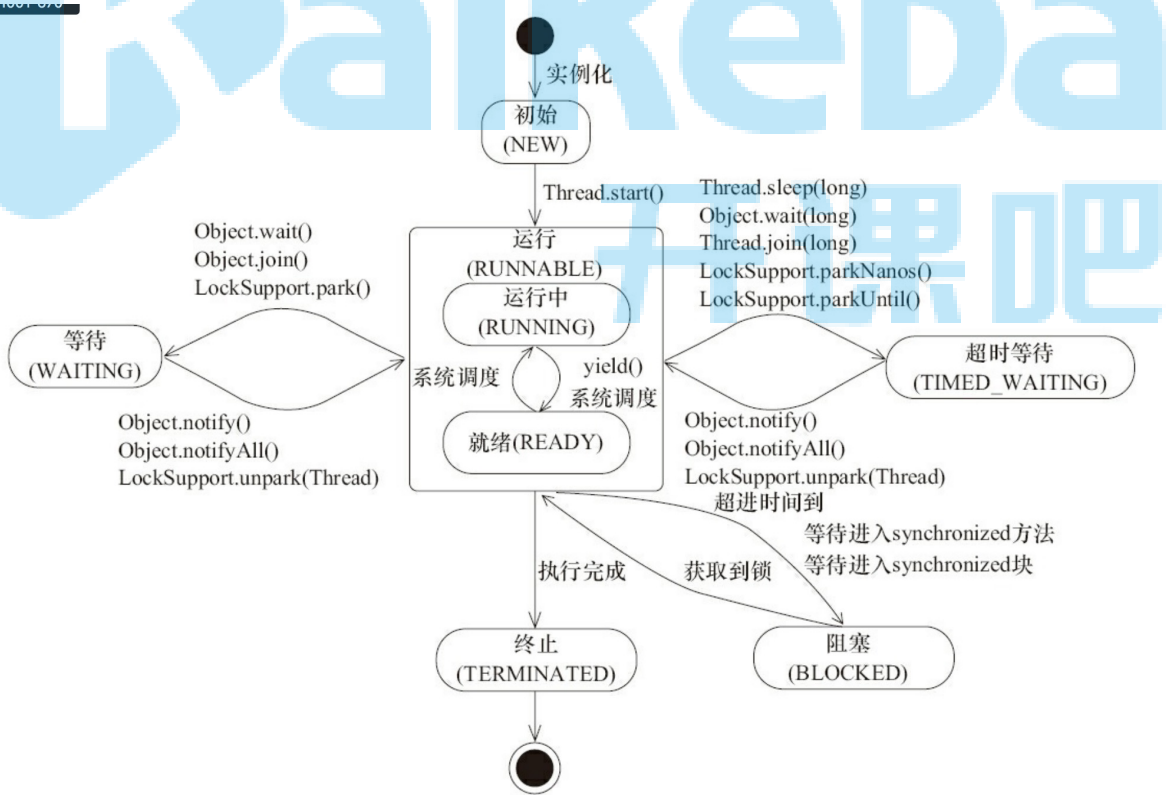
```
first :true
second:false
task 1 is o
```

1.1.2.4 线程的状态

Java线程可能的状态：

状态名称	说　明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

线程的状态变迁



1.2 多线程

线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

1.2.1 并发和并行

- 并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。

- 并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。
- 在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

1.2.2 多线程好处

提高cpu的利用率

单线程：

```
  5 seconds reading file A
  2 seconds processing file A
  5 seconds reading file B
  2 seconds processing file B
----------------------
 14 seconds total
```

多线程

```
  5 seconds reading file A
  5 seconds reading file B + 2 seconds processing file A
  2 seconds processing file B
----------------------
 12 seconds total
```

一般来说，在等待磁盘IO，网络IO或者等待用户输入时，CPU可以同时去处理其他任务。

更高效的响应

多线程技术使程序的响应速度更快 ，因为用户界面可以在进行其它工作的同时一直处于活动状态，不会造成无法响应的现象。

公平使用CPU资源

当前没有进行处理的任务，可以将处理器时间让给其它任务;占用大量处理时间的任务，也可以定期将处理器时间让给其它任务;通过对CPU时间的划分，使得CPU时间片可以在多个线程之间切换，避免需要长时间处理的线程独占CPU，导致其它线程长时间等待。

1.2.3 多线程的代价

更复杂的设计

共享数据的读取，数据的安全性，线程之间的交互，线程的同步等；

上下文环境切换

线程切换，cpu需要保存本地数据、程序指针等内容；

更多的资源消耗

每个线程都需要内存维护自己的本地栈信息，操作系统也需要资源对线程进行管理维护；

二、并发编程基础

2.1 临界资源

临界资源是一次仅允许一个进程使用的共享资源。*各进程采取互斥的方式，实现共享的资源称作临界资源。*属于临界资源的硬件有，打印机，磁带机等；软件有消息队列，变量，数组，缓冲区等。诸进程间采取互斥方式，实现对这种资源的共享。

```
public class Counter {

    protected long count = 0;

    public void add(long value){
        this.count = this.count + value;
    }

```

2.2 线程安全

2.2.1 基本概念

何谓竞态条件

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。

导致竞态条件发生的代码区称作临界区。

在临界区中使用适当的同步就可以避免竞态条件，如使用synchronized或者加锁机制。

何谓线程安全

允许被多个线程同时执行的代码称作线程安全的代码。线程安全的代码不包含竞态条件。

2.2.2 对象的安全

局部基本类型变量

局部变量存储在线程自己的栈中。也就是说，局部变量永远也不会被多个线程共享。所以，基础类型的局部变量是线程安全的。下面是基础类型的局部变量的一个例子：

```
package kaikeba.com;

public class ThreadTest {

    public static void main(String[]args){
        MyThread share = new MyThread();
        for (int i=0;i<50;i++){
            new Thread(share,"线程"+i).start();
        }
    }
}

 class MyThread implements Runnable{

    public void run() {
        int a =0;
        ++a;
        System.out.println(Thread.currentThread().getName()+":"+a);
    }
}

```

无论多少个线程对run()方法中的基本类型a执行++a操作，只是更新当前线程栈的值，不会影响其他线程，也就是不共享数据；

局部的对象引用

对象的局部引用和基础类型的局部变量不太一样，尽管引用本身没有被共享，但引用所指的对象并没有存储在线程的栈内。所有的对象都存在共享堆中。
如果在某个方法中创建的对象不会逃逸出（即该对象不会被其它方法获得，也不会被非局部变量引用到）该方法，那么它就是线程安全的。
实际上，哪怕将这个对象作为参数传给其它方法，只要别的线程获取不到这个对象，那它仍是线程安全的。

```
public void method1(){
  LocalObject localObject = new LocalObject();
  localObject.callMethod();
  method2(localObject);
}
public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

对象成员(成员变量)

对象成员存储在堆上。如果两个线程同时更新同一个对象的同一个成员，那这个代码就不是线程安全的。

```
package kaikeba.com;

public class ThreadTest {

    public static void main(String[]args){
        NotThreadSafe sharedInstance = new NotThreadSafe();
        new Thread(new MyRunnable(sharedInstance)).start();
        new Thread(new MyRunnable(sharedInstance)).start();

    }
}

 class MyRunnable implements Runnable{
    NotThreadSafe instance = null;
    public MyRunnable(NotThreadSafe instance){
        this.instance = instance;
    }
    public void run(){
        this.instance.add(" "+Thread.currentThread().getName());
        System.out.println(this.instance.builder.toString());
    }
}

 class NotThreadSafe{
    StringBuilder builder = new StringBuilder();

    public void add(String text){
         this.builder.append(text);
    }
}
```

如果两个线程同时调用同一个NotThreadSafe实例上的add()方法，就会有竞态条件问题。

2.2.3 不可变性

通过创建不可变的共享对象来保证对象在线程间共享时不会被修改，从而实现线程安全。如下示例：

```
public class ImmutableValue{
    private int value = 0;

    public ImmutableValue(int value){
        this.value = value;
    }

    public int getValue(){
        return this.value;
    }
}
```

请注意ImmutableValue类的成员变量 `value` 是通过构造函数赋值的，并且在类中没有set方法。这意味着一旦ImmutableValue实例被创建， `value` 变量就不能再被修改，这就是不可变性。但你可以通过getValue()方法读取这个变量的值。

2.3 Java内存模型

Java内存模型即Java Memory Model，简称JMM。JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。

线程之间的通信

线程的通信是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种共享内存和消息传递。

在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。

在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是wait()和notify()。

线程之间的同步

同步是指程序用于控制不同线程之间操作发生相对顺序的机制。

在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。

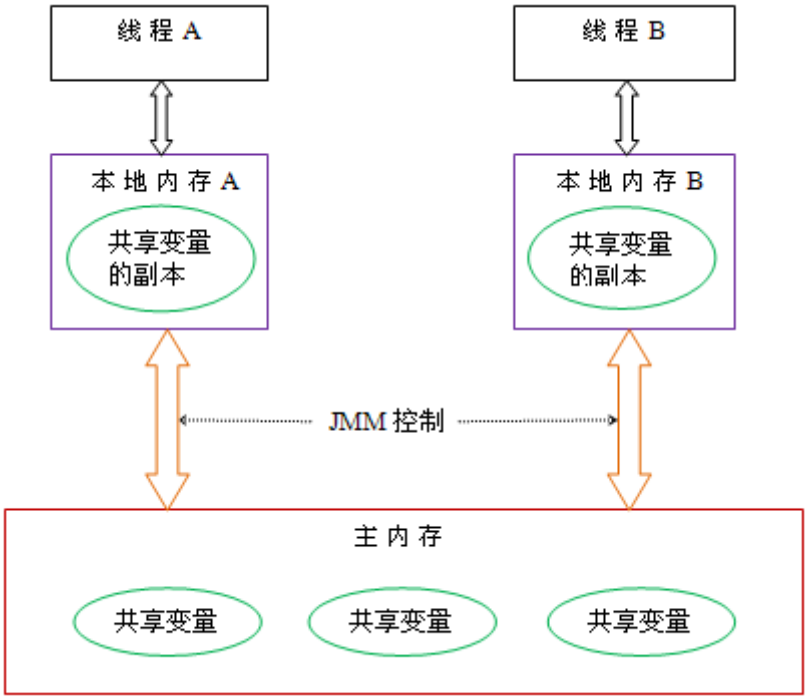
在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

Java的并发采用的是共享内存模型

Java线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。如果编写多线程程序的Java程序员不理解隐式进行的线程之间通信的工作机制，很可能会遇到各种奇怪的内存可见性问题。

Java内存模型结构

Java内存模型(简称JMM)，JMM决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

2.4 CAS乐观锁

乐观锁：不加锁，假设没有冲突去完成某项操作，如果因为冲突失败就重试，直到成功为止。其实现方式有一种比较典型的就是Compare and Swap(CAS)。

CAS机制当中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。

更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。

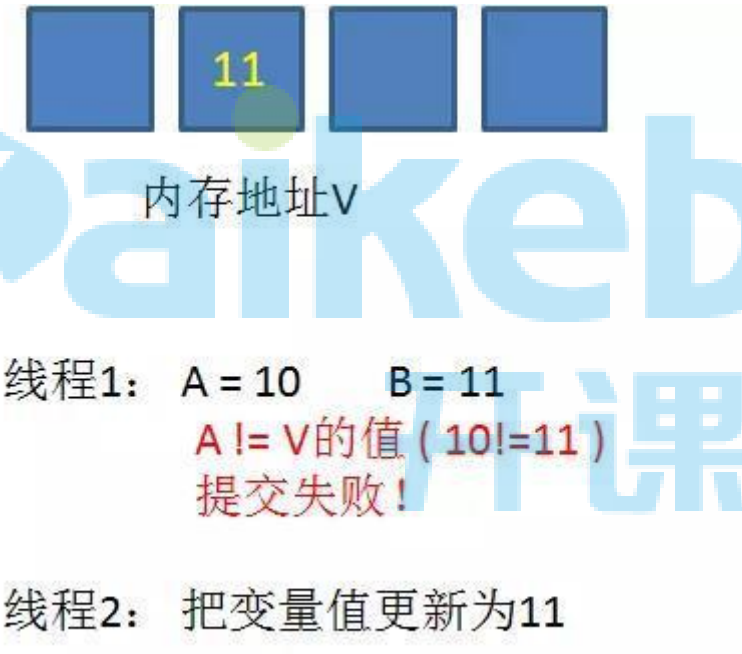
这样说或许有些抽象，我们来看一个例子：

1.在内存地址V当中，存储着值为10的变量。

2.此时线程1想要把变量的值增加1。对线程1来说，旧的预期值A=10，要修改的新值B=11。

3.在线程1要提交更新之前，另一个线程2抢先一步，把内存地址V中的变量值率先更新成了11。

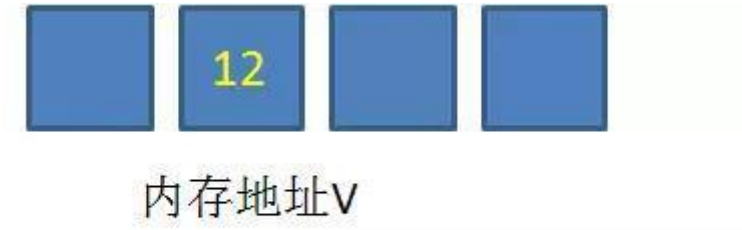
4.线程1开始提交更新，首先进行A和地址V的实际值比较（Compare），发现A不等于V的实际值，提交失败。



5.线程1重新获取内存地址V的当前值，并重新计算想要修改的新值。此时对线程1来说，A=11，B=12。这个重新尝试的过程被称为自旋。

6.这一次比较幸运，没有其他线程改变地址V的值。线程1进行Compare，发现A和地址V的实际值是相等的。

7.线程1进行SWAP，把地址V的值替换为B，也就是12。



从思想上来说，Synchronized属于悲观锁，悲观地认为程序中的并发情况严重，所以严防死守。CAS属于乐观锁，乐观地认为程序中的并发情况不那么严重，所以让线程不断去尝试更新。

CAS的缺点：

1.CPU开销较大
在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性
CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

2.5 Synchronized块

Java中的同步块用synchronized标记。同步块在Java中是同步在某个对象上。所有同步在一个对象上的同步块在同时只能被一个线程进入并执行操作。所有其他等待进入该同步块的线程将被阻塞，直到执行该同步块中的线程退出。

有四种不同的同步块：

1. 实例方法
2. 静态方法
3. 实例方法中的同步块
4. 静态方法中的同步块

上述同步块都同步在不同对象上。实际需要那种同步块视具体情况而定。

实例方法同步

下面是一个同步的实例方法：

```
 public synchronized void add(int value){
this.count += value;
 }
```

注意在方法声明中同步（ synchronized ）关键字。

Java实例方法同步是同步在拥有该方法的对象上。这样，每个实例其方法同步都同步在不同的对象上，即该方法所属的实例。只有一个线程能够在实例方法同步块中运行。如果有多个实例存在，那么一个线程一次可以在一个实例同步块中执行操作。

静态方法同步

静态方法同步和实例方法同步方法一样，也使用synchronized 关键字。Java静态方法同步如下示例：

```
public static synchronized void add(int value){
 count += value;
 }
```

同样，这里synchronized 关键字告诉Java这个方法是同步的。

静态方法的同步是指同步在该方法所在的类对象上。因为在Java虚拟机中一个类只能对应一个类对象，所以同时只允许一个线程执行同一个类中的静态同步方法。

对于不同类中的静态同步方法，一个线程可以执行每个类中的静态同步方法而无需等待。不管类中的那个静态同步方法被调用，一个类只能由一个线程同时执行。

实例方法中的同步块

有时你不需要同步整个方法，而是同步方法中的一部分。Java可以对方法的一部分进行同步。

在非同步的Java方法中的同步块的例子如下所示：

```
public void add(int value){

    synchronized(this){
       this.count += value;
    }
  }
```

示例使用Java同步块构造器来标记一块代码是同步的。该代码在执行时和同步方法一样。

注意Java同步块构造器用括号将对象括起来。在上例中，使用了“this”，即为调用add方法的实例本身。在同步构造器中用括号括起来的对象叫做监视器对象。上述代码使用监视器对象同步，同步实例方法使用调用方法本身的实例作为监视器对象。

一次只有一个线程能够在同步于同一个监视器对象的Java方法内执行。

下面两个例子都同步他们所调用的实例对象上，因此他们在同步的执行效果上是等效的。

```
 public class MyClass {

    public synchronized void log1(String msg1, String msg2){
       log.writeln(msg1);
       log.writeln(msg2);
    }

    public void log2(String msg1, String msg2){
       synchronized(this){
          log.writeln(msg1);
          log.writeln(msg2);
       }
    }
  }
```

在上例中，每次只有一个线程能够在两个同步块中任意一个方法内执行。

如果第二个同步块不是同步在this实例对象上，那么两个方法可以被线程同时执行。

静态方法中的同步块

和上面类似，下面是两个静态方法同步的例子。这些方法同步在该方法所属的类对象上。

```
public class MyClass {
    public static synchronized void log1(String msg1, String msg2){
       log.writeln(msg1);
       log.writeln(msg2);
    }

    public static void log2(String msg1, String msg2){
       synchronized(MyClass.class){
          log.writeln(msg1);
          log.writeln(msg2);
       }
    }
  }
```

这两个方法不允许同时被线程访问。

如果第二个同步块不是同步在MyClass.class这个对象上。那么这两个方法可以同时被线程访问。

Synchronized锁的存储

synchronized用的锁存储在Java对象头，如果对象是数组类型，则虚拟机用3个字宽存储对象头，如果对象是非数组类型，则用2字宽存储对象头，32位虚拟机，1字宽等于4字节，即32位.

Java对象头的长度

长　　度	内　　容	说　　明
32/64bit	Mark Word	存储对象的 hashCode 或锁信息等
32/64bit	Class Metadata Address	存储到对象类型数据的指针
32/32bit	Array length	数组的长度（如果当前对象是数组）

Mark Word的存储结构

锁状态	25bit	4bit	1bit 是否是偏向锁	2bit 锁标志位
无锁状态	对象的 hashCode	对象分代年龄	0	01

Mark Word可能的存储结果：

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

偏向锁

偏向锁的获取流程：

（1）查看Mark Word中偏向锁的标识以及锁标志位，若是否偏向锁为1且锁标志位为01，则该锁为可偏向状态。

（2）若为可偏向状态，则测试Mark Word中的线程ID是否与当前线程相同，若相同，表示线程已经获得了锁，如果不同，则进入（3）

（3）测试Mark Word的偏向锁的标识是否设置为1，如果没有设置，则使用CAS操作竞争锁，如何设置了，则尝试使用CAS尝试将Mark Word中线程ID设置为当前线程ID，如果尝试失败，则执行（4）

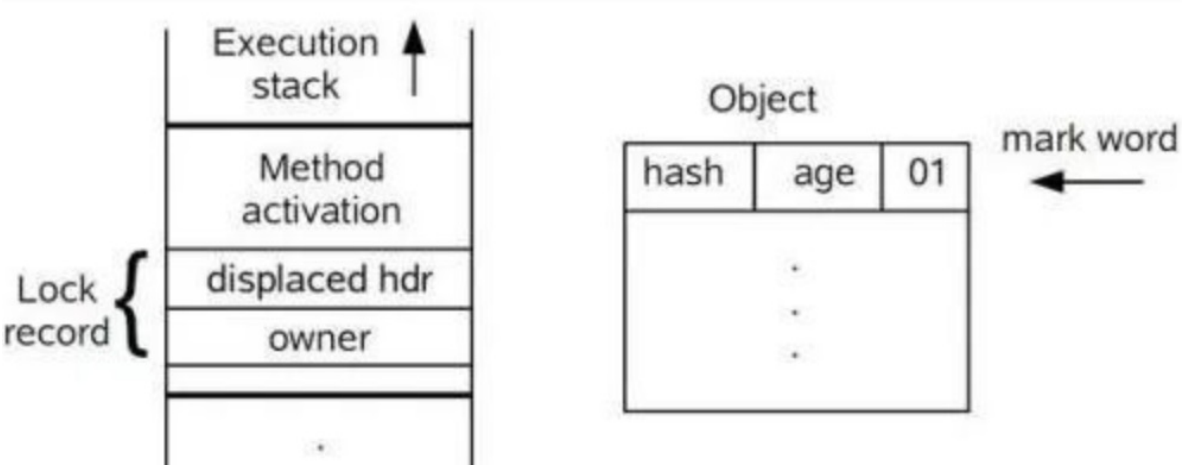
（4）当前线程通过CAS竞争锁失败的情况下，说明有竞争。当到达全局安全点（在这个时间点，没有正在执行的代码）时之前获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。

轻量级锁

轻量级锁不是用来替代传统的重量级锁的，而是在没有多线程竞争的情况下，使用轻量级锁能够减少性能消耗，但是当多个线程同时竞争锁时，轻量级锁会膨胀为重量级锁。

轻量级锁的加锁过程：

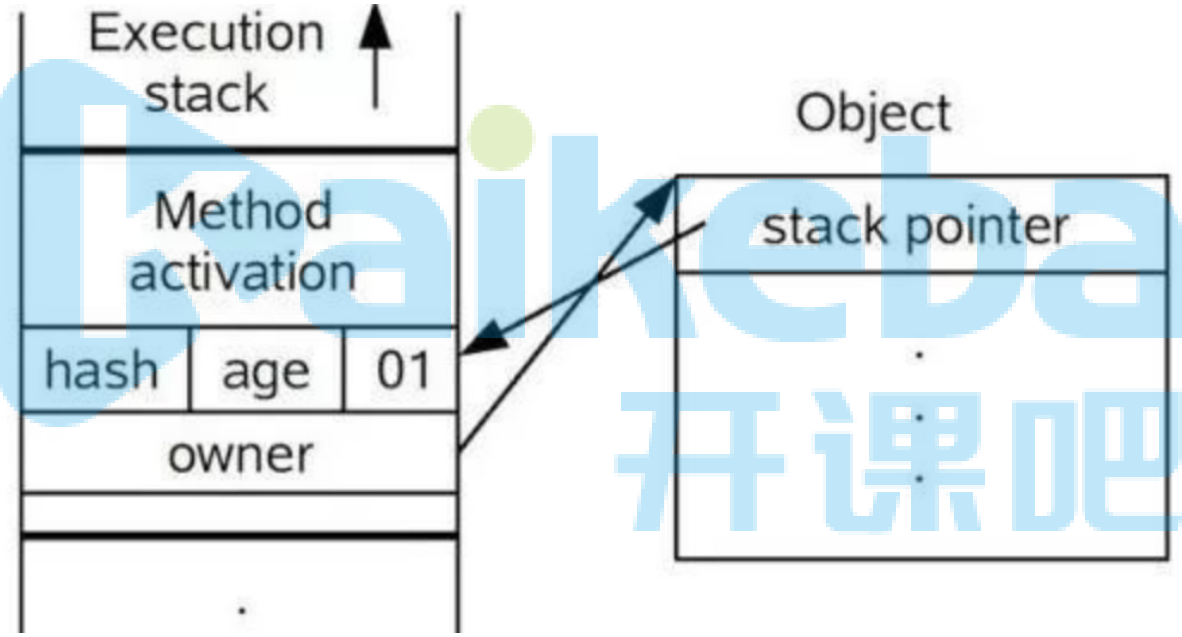
（1）当线程执行代码进入同步块时，若Mark Word为无锁状态，虚拟机先在当前线程的栈帧中建立一个名为Lock Record的空间，用于存储当前对象的Mark Word的拷贝，官方称之为“Dispalced Mark Word”，此时状态如下图：



（2）复制对象头中的Mark Word到锁记录中。

（3）复制成功后，虚拟机将用CAS操作将对象的Mark Word更新为执行Lock Record的指针，并将Lock Record里的owner指针指向对象的Mark Word。如果更新成功，则执行4，否则执行5。；

（4）如果更新成功，则这个线程拥有了这个锁，并将锁标志设为00，表示处于轻量级锁状态，此时状态图：



（5）如果更新失败，则说明有其他线程竞争锁，当前线程便通过自旋来获取锁。轻量级锁就会膨胀为重量级锁，Mark Word中存储重量级锁（互斥锁）的指针，后面等待锁的线程也要进入阻塞状态。

重量级锁

即当有其他线程占用锁时，当前线程会进入阻塞状态。

2.6 关键字Volatile

Volatile是轻量级的synchronized,在多处理器环境下，可以保证共享变量的可见性。它不会引起线程上下文的切换和调度，正确的使用Volatile,比synchronized的使用和执行成本更低。

可见性：

可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改一个共享变量时，另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。

volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作 a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就是这个操作同样存在线程安全问题。

在 Java 中 volatile、synchronized 和 final 实现可见性。

原子性：

子是世界上的最小单位，具有不可分割性。比如 a=0；（a非long和double类型） 这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++； 这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。非原子操作都会存在线程安全问题，需要我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

有序性：

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

Java语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。

在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制。

当一个变量定义为 volatile 之后，将具备两种特性：

- 保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存来完成。
- 禁止指令重排序优化。有volatile修饰的变量，赋值后多执行了一个“load addl $0x0, (%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障；（什么是指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理。

2.7 本地线程

Java中的ThreadLocal类允许我们创建只能被同一个线程读写的变量。因此，如果一段代码含有一个ThreadLocal变量的引用，即使两个线程同时执行这段代码，它们也无法访问到对方的ThreadLocal变量。

如何创建ThreadLocal变量

以下代码展示了如何创建一个ThreadLocal变量：

```
1 private ThreadLocal myThreadLocal = new ThreadLocal();
```

我们可以看到，通过这段代码实例化了一个ThreadLocal对象。我们只需要实例化对象一次，并且也不需要知道它是被哪个线程实例化。虽然所有的线程都能访问到这个ThreadLocal实例，但是每个线程却只能访问到自己通过调用ThreadLocal的set()方法设置的值。即使是两个不同的线程在同一个ThreadLocal对象上设置了不同的值，他们仍然无法访问到对方的值。

如何访问ThreadLocal变量

一旦创建了一个ThreadLocal变量，你可以通过如下代码设置某个需要保存的值：

```
myThreadLocal.set("A thread local value”);
```

可以通过下面方法读取保存在ThreadLocal变量中的值：

```
String threadLocalValue = (String) myThreadLocal.get();
```

ThreadLocal例子：

```
public class ThreadLocalExample {

    public static class MyRunnable implements Runnable {

        private ThreadLocal threadLocal = new ThreadLocal();

        @Override
        public void run() {
            threadLocal.set((int) (Math.random() * 100D));
            try {
            Thread.sleep(2000);
            } catch (InterruptedException e) {

            }
            System.out.println(threadLocal.get());
        }
    }

    public static void main(String[] args) {
         MyRunnable sharedRunnableInstance = new MyRunnable();
         Thread thread1 = new Thread(sharedRunnableInstance);
         Thread thread2 = new Thread(sharedRunnableInstance);
         thread1.start();
         thread2.start();
    }

}
```

上面的例子创建了一个MyRunnable实例，并将该实例作为参数传递给两个线程。两个线程分别执行run()方法，并且都在ThreadLocal实例上保存了不同的值。如果它们访问的不是ThreadLocal对象并且调用的set()方法被同步了，则第二个线程会覆盖掉第一个线程设置的值。但是，由于它们访问的是一个ThreadLocal对象，因此这两个线程都无法看到对方保存的值。也就是说，它们存取的是两个不同的值。

关于InheritableThreadLocal

InheritableThreadLocal类是ThreadLocal类的子类。ThreadLocal中每个线程拥有它自己的值，与ThreadLocal不同的是，InheritableThreadLocal允许一个线程以及该线程创建的所有子线程都可以访问它保存的值。

2.8、多线程问题

2.8.1 死锁

死锁的产生

死锁是两个或更多线程阻塞着等待其它处于死锁状态的线程所持有的锁。死锁通常发生在多个线程同时但以不同的顺序请求同一组锁的时候。

例如，如果线程1锁住了A，然后尝试对B进行加锁，同时线程2已经锁住了B，接着尝试对A进行加锁，这时死锁就发生了。线程1永远得不到B，线程2也永远得不到A，并且它们永远也不会知道发生了这样的事情。为了得到彼此的对象（A和B），它们将永远阻塞下去。这种情况就是一个死锁。

该情况如下：

```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for
```

更复杂的死锁

死锁可能不止包含2个线程，这让检测死锁变得更加困难。下面是4个线程发生死锁的例子：

```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for C
Thread 3  locks C, waits for D
Thread 4  locks D, waits for A
```

线程1等待线程2，线程2等待线程3，线程3等待线程4，线程4等待线程1。

数据库的死锁

更加复杂的死锁场景发生在数据库事务中。一个数据库事务可能由多条SQL更新请求组成。当在一个事务中更新一条记录，这条记录就会被锁住避免其他事务的更新请求，直到第一个事务结束。同一个事务中每一个更新请求都可能会锁住一些记录。

当多个事务同时需要对一些相同的记录做更新操作时，就很有可能发生死锁，例如：

```
Transaction 1, request 1, locks record 1 for update
Transaction 2, request 1, locks record 2 for update
Transaction 1, request 2, tries to lock record 2 for update.
Transaction 2, request 2, tries to lock record 1 for update.
```

因为锁发生在不同的请求中，并且对于一个事务来说不可能提前知道所有它需要的锁，因此很难检测和避免数据库事务中的死锁。

死锁的避免

加锁顺序

当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。

如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。看下面这个例子：

```
Thread 1:
  lock A
  lock B

Thread 2:
   wait for A
   lock C (when A locked)

Thread 3:
   wait for A
   wait for B
   wait for C
```

如果一个线程（比如线程3）需要一些锁，那么它必须按照确定的顺序获取锁。它只有获得了从顺序上排在前面的锁之后，才能获取后面的锁。

例如，线程2和线程3只有在获取了锁A之后才能尝试获取锁C(*译者注：获取锁A是获取锁C的必要条件*)。因为线程1已经拥有了锁A，所以线程2和3需要一直等到锁A被释放。然后在它们尝试对B或C加锁之前，必须成功地对A加了锁。

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁(*译者注：并对这些锁做适当的排序*)，但总有些时候是无法预知的。

加锁时限

另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行(*译者注：加锁超时后可以先继续运行干点其它事情，再回头来重复之前加锁的逻辑*)。

以下是一个例子，展示了两个线程以不同的顺序尝试获取相同的两个锁，在发生超时后回退并重试的场景：

```
Thread 1 locks A
Thread 2 locks B

Thread 1 attempts to lock B but is blocked
Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out
Thread 1 backs up and releases A as well
Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out
Thread 2 backs up and releases B as well
Thread 2 waits randomly (e.g. 43 millis) before retrying.
```

在上面的例子中，线程2比线程1早200毫秒进行重试加锁，因此它可以先成功地获取到两个锁。这时，线程1尝试获取锁A并且处于等待状态。当线程2结束时，线程1也可以顺利的获得这两个锁（除非线程2或者其它线程在线程1成功获得两个锁之前又获得其中的一些锁）。

需要注意的是，由于存在锁的超时，所以我们不能认为这种场景就一定是出现了死锁。也可能是因为获得了锁的线程（导致其它线程超时）需要很长的时间去完成它的任务。

此外，如果有非常多的线程同一时间去竞争同一批资源，就算有超时和回退机制，还是可能会导致这些线程重复地尝试但却始终得不到锁。如果只有两个线程，并且重试的超时时间设定为0到500毫秒之间，这种现象可能不会发生，但是如果是10个或20个线程情况就不同了。因为这些线程等待相等的重试时间的概率就高的多（或者非常接近以至于会出现问题）。

这种机制存在一个问题，在Java中不能对synchronized同步块设置超时时间。你需要创建一个自定义锁，或使用Java5中java.util.concurrent包下的工具。写一个自定义锁类不复杂，但超出了本文的内容。后续的Java并发系列会涵盖自定义锁的内容。

死锁检测

死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。例如，线程A请求锁7，但是锁7这个时候被线程B持有，这时线程A就可以检查一下线程B是否已经请求了线程A当前所持有的锁。如果线程B确实有这样的请求，那么就是发生了死锁（线程A拥有锁1，请求锁7；线程B拥有锁7，请求锁1）。

当然，死锁一般要比两个线程互相持有对方的锁这种情况要复杂的多。线程A等待线程B，线程B等待线程C，线程C等待线程D，线程D又在等待线程A。线程A为了检测死锁，它需要递进地检测所有被B请求的锁。从线程B所请求的锁开始，线程A找到了线程C，然后又找到了线程D，发现线程D请求的锁被线程A自己持有着。这是它就知道发生了死锁。

那么当检测出死锁时，这些线程该做些什么呢？

一个可行的做法是释放所有锁，回退，并且等待一段随机的时间后重试。这个和简单的加锁超时类似，不一样的是只有死锁已经发生了才回退，而不会是因为加锁的请求超时了。虽然有回退和等待，但是如果有大量的线程竞争同一批锁，它们还是会重复地死锁（*编者注：原因同超时类似，不能从根本上减轻竞争*）。

一个更好的方案是给这些线程设置优先级，让一个（或几个）线程回退，剩下的线程就像没发生死锁一样继续保持着它们需要的锁。如果赋予这些线程的优先级是固定不变的，同一批线程总是会拥有更高的优先级。为避免这个问题，可以在死锁发生的时候设置随机的优先级。

2.8.2 饥饿和公平

如果一个线程因为CPU时间全部被其他线程抢走而得不到CPU运行时间，这种状态被称之为“饥饿”。而该线程被“饥饿致死”正是因为它得不到CPU运行时间的机会。解决饥饿的方案被称之为“公平性” – 即所有线程均能公平地获得运行机会。

Java中导致饥饿的原因

在Java中，下面三个常见的原因会导致线程饥饿：

- 高优先级线程吞噬所有的低优先级线程的CPU时间

  你能为每个线程设置独自的线程优先级，优先级越高的线程获得的CPU时间越多，线程优先级值设置在1到10之间，而这些优先级值所表示行为的准确解释则依赖于你的应用运行平台。对大多数应用来说，你最好是不要改变其优先级值。

- 线程被永久堵塞在一个等待进入同步块的状态

  Java的同步代码区也是一个导致饥饿的因素。Java的同步代码区对哪个线程允许进入的次序没有任何保障。这就意味着理论上存在一个试图进入该同步区的线程处于被永久堵塞的风险，因为其他线程总是能持续地先于它获得访问，这即是“饥饿”问题，而一个线程被“饥饿致死”正是因为它得不到CPU运行时间的机会。

- 线程在等待一个本身(在其上调用wait())也处于永久等待完成的对象

  如果多个线程处在wait()方法执行上，而对其调用notify()不会保证哪一个线程会获得唤醒，任何线程都有可能处于继续等待的状态。因此存在这样一个风险：一个等待线程从来得不到唤醒，因为其他等待线程总是能被获得唤醒。

三、JUC:java.util.concurrent

3.1 集合

3.1.1 BlockingQueue

什么是阻塞队列？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

阻塞队列提供了四种处理方法：

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

- 异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出IllegalStateException("Queue full")异常。当队列为空时，从队列里获取元素时会抛出NoSuchElementException异常 。
- 返回特殊值：插入方法会返回是否成功，成功则返回true。移除方法，则是从队列里拿出一个元素，如果没有则返回null
- 一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里take元素，队列也会阻塞消费者线程，直到队列可用。
- 超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

阻塞队列接口：

```
BlockingQueue的核心方法：

public interface BlockingQueue<E> extends Queue<E> {

    //插入元素e到队列中，成功返回true，否则抛出异常。如果向限定了容量的队列中插入值，推荐使用offer()方法。
    boolean add(E e);

    //插入元素e到队列中，如果设置成功返回true，否则返回false. e的值不能为空，否则抛出空指针异常。
    boolean offer(E e);

    //插入元素e到队列中，，如果队列中没有多余的空间，该方法会一直阻塞，直到队列中有多余的空间。
    void put(E e) throws InterruptedException;

```

```
    //在给定的时间插入元素e到队列中，如果设置成功返回true，否则返回false.
    boolean offer(E e, long timeout, TimeUnit unit)
        throws InterruptedException;

    //检索并从队列的头部删除元素，如果队列中没有值，线程会一直阻塞，直到队列中有值，并且该方法取得了该值。
    E take() throws InterruptedException;

    //在给定的时间范围内，检索并从队列的头部删除元素，从队列中获取值，如果没有取到会抛出异常。
    E poll(long timeout, TimeUnit unit)
        throws InterruptedException;

    //获取队列中剩余的空间。
    int remainingCapacity();

    //从队列中移除指定的值。
    boolean remove(Object o);

    //判断队列中是否包含该值。
    public boolean contains(Object o);

    //将队列中值，全部移除，并追加到给定的集合中。
    int drainTo(Collection<? super E> c);

    //指定最多数量限制将队列中值，全部移除，并追加到给定的集合中。
    int drainTo(Collection<? super E> c, int maxElements);
}
```

继承关系

- 子接口：

BlockingDeque

Summary of BlockingDeque methods

First Element (Head)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	addFirst(e)	offerFirst(e)	putFirst(e)	offerFirst(e, time, unit)
Remove	removeFirst()	pollFirst()	takeFirst()	pollFirst(time, unit)
Examine	getFirst()	peekFirst()	*not applicable*	*not applicable*
Last Element (Tail)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	addLast(e)	offerLast(e)	putLast(e)	offerLast(e, time, unit)
Remove	removeLast()	pollLast()	takeLast()	pollLast(time, unit)
Examine	getLast()	peekLast()	*not applicable*	*not applicable*

TransferQueue

TransferQueue继承了BlockingQueue,并扩展了一些新方法。

BlockingQueue是指这样的一个队列：当生产者向队列添加元素但队列已满时，生产者会被阻塞；当消费者从队列移除元素但队列为空时，消费者会被阻塞。

TransferQueue则更进一步，生产者会一直阻塞直到所添加到队列的元素被某一个消费者所消费（不仅仅是添加到队列里就完事）。新添加的transfer方法用来实现这种约束。顾名思义，阻塞就是发生在元素从一个线程transfer到另一个线程的过程中，它有效地实现了元素在线程之间的传递（以建立Java内存模型中的happens-before关系的方式）。

TransferQueue还包括了其他的一些方法：两个tryTransfer方法，一个是非阻塞的，另一个带有timeout参数设置超时时间的。还有两个辅助方法hasWaitingConsumer()和getWaitingConsumerCount()。

- 实现类

  ArrayBlockingQueue

  DelayQueue

  LinkedBlockingDeque

  LinkedBlockingQueue

  LinkedTransferQueue

  PriorityBlockingQueue

  SynchronousQueue

3.1.2 ArrayBlockingQueue

`ArrayBlockingQueue` 是一个线程安全的、基于数组、有界的、阻塞的、FIFO 队列。试图向已满队列中放入元素会导致操作受阻塞；试图从空队列中提取元素将导致类似阻塞。

此类基于 `java.util.concurrent.locks.ReentrantLock` 来实现线程安全，所以提供了 `ReentrantLock` 所能支持的公平性选择。

源码:略

使用：

```
1  package kaikeba.com;
2
3  import java.util.concurrent.ArrayBlockingQueue;
4  import java.util.concurrent.BlockingQueue;
5
6  public class ArrayBlockingQueueDemo {
7      public static void main(String[] args) {
8
9          BlockingQueue<Integer> blockingQueue = new
   ArrayBlockingQueue<Integer>(3,true);
10         Producer producer = new Producer(blockingQueue);
11         Consumer consumer = new Consumer(blockingQueue);
12
13         new Thread(producer).start();
14
15         new Thread(consumer).start();
16
17
18     }
19
20 }
21
22 class Producer implements Runnable {
23
24     private  BlockingQueue<Integer> blockingQueue;
```

```
    private static int element = 0;

    public Producer(BlockingQueue<Integer> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }


    public void run() {
        try {
            while(element < 20) {
                System.out.println("生产元素："+element);
                blockingQueue.put(element++);
            }
        } catch (Exception e) {
            System.out.println("生产者在等待空闲空间的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("生产者终止了生产过程！");
    }
}
 class Consumer implements Runnable {

    private  BlockingQueue<Integer> blockingQueue;

    public Consumer(BlockingQueue<Integer> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }


    public void run() {
        try {
            while(true) {
                System.out.println("消费元素："+blockingQueue.take());
            }
        } catch (Exception e) {
            System.out.println("消费者在等待新产品的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("消费者终止了消费过程！");
    }
}
```

3.1.3 PriorityBlockingQueue

PriorityBlockingQueue是带优先级的无界阻塞队列，每次出队都返回优先级最高的元素，是二叉树最小堆的实现。

源码：略。

使用：

```
package kaikeba.com;

import java.util.Random;
import java.util.concurrent.PriorityBlockingQueue;

public class PriorityBlockingQueueTest {
    public static void main(String[] args) throws InterruptedException {
```

```
        PriorityBlockingQueue<PriorityElement> queue = new
PriorityBlockingQueue<>();
        for (int i = 0; i < 5; i++) {
            Random random=new Random();
            PriorityElement ele = new PriorityElement(random.nextInt(10));
            queue.put(ele);
        }
        while(!queue.isEmpty()){
            System.out.println(queue.take());
        }
    }
}
 class PriorityElement implements Comparable<PriorityElement> {
    private int priority;//定义优先级
    PriorityElement(int priority) {
        //初始化优先级
        this.priority = priority;
    }
    @Override
    public int compareTo(PriorityElement o) {
        //按照优先级大小进行排序
        return priority >= o.getPriority() ? 1 : -1;
    }
    public int getPriority() {
        return priority;
    }
    public void setPriority(int priority) {
        this.priority = priority;
    }
    @Override
    public String toString() {
        return "PriorityElement [priority=" + priority + "]";
    }
}
```

3.1.4 DelayQueue

DelayQueue队列中每个元素都有个过期时间，并且队列是个优先级队列，当从队列获取元素时候，只有过期元素才会出队列。

源码:略

使用

```
package kaikeba.com;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class DelayQueueTest  {

    public static void main(String[] args) throws InterruptedException {
        Item item1 = new Item("item1", 5, TimeUnit.SECONDS);
        Item item2 = new Item("item2",10, TimeUnit.SECONDS);
        Item item3 = new Item("item3",15, TimeUnit.SECONDS);
```

```
        DelayQueue<Item> queue = new DelayQueue<>();
        queue.put(item1);
        queue.put(item2);
        queue.put(item3);
        System.out.println("begin time:" +
LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
        for (int i = 0; i < 3; i++) {
            Item take = queue.take();
            System.out.format("name:{%s}, time:{%s}\n",take.name,
LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
        }
    }

}

class Item implements Delayed {
    /* 触发时间*/
    private long time;
    String name;

    public Item(String name, long time, TimeUnit unit) {
        this.name = name;
        this.time = System.currentTimeMillis() + (time > 0?
unit.toMillis(time): 0);
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return time - System.currentTimeMillis();
    }

    @Override
    public int compareTo(Delayed o) {
        Item item = (Item) o;
        long diff = this.time - item.time;
        if (diff <= 0) {
            return -1;
        }else {
            return 1;
        }
    }

    @Override
    public String toString() {
        return "Item{" +
                "time=" + time +
                ", name='" + name + '\'' +
                '}';
    }
}
```

3.1.5 LinkedBlockingQueue

LinkedBlockingQueue是一个基于单向链表的、范围任意的（其实是有界的）、FIFO 阻塞队列。访问与移除操作是在队头进行，添加操作是在队尾进行，并分别使用不同的锁进行保护，只有在可能涉及多个节点的操作才同时对两个锁进行加锁。

队列是否为空、是否已满仍然是通过元素数量的计数器（count）进行判断的，由于可以同时在队头、队尾并发地进行访问、添加操作，所以这个计数器必须是线程安全的，这里使用了一个原子类 `AtomicInteger`，这就决定了它的容量范围是： 1 — Integer.MAX_VALUE。

由于同时使用了两把锁，在需要同时使用两把锁时，加锁顺序与释放顺序是非常重要的：必须以固定的顺序进行加锁，再以与加锁顺序的相反的顺序释放锁。

头结点和尾结点一开始总是指向一个哨兵的结点，它不持有实际数据，当队列中有数据时，头结点仍然指向这个哨兵，尾结点指向有效数据的最后一个结点。这样做的好处在于，与计数器 count 结合后，对队头、队尾的访问可以独立进行，而不需要判断头结点与尾结点的关系。

源码：略。

使用：

```
package kaikeba.com;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.LinkedBlockingQueue;

public class ArrayBlockingQueueDemo {
    public static void main(String[] args) {

        BlockingQueue<Integer> blockingQueue = new LinkedBlockingQueue<>();
        Producer producer = new Producer(blockingQueue);
        Consumer consumer = new Consumer(blockingQueue);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
}

class Producer implements Runnable {

    private  BlockingQueue<Integer> blockingQueue;
    private static int element = 0;

    public Producer(BlockingQueue<Integer> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }


    public void run() {
        try {
            while(element < 20) {
                System.out.println("生产元素："+element);
                blockingQueue.put(element++);
            }
        } catch (Exception e) {
            System.out.println("生产者在等待空闲空间的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("生产者终止了生产过程！");
    }
}
 class Consumer implements Runnable {
    private  BlockingQueue<Integer> blockingQueue;
```

```
    public Consumer(BlockingQueue<Integer> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }


    public void run() {
        try {
            while(true) {
                System.out.println("消费元素："+blockingQueue.take());
            }
        } catch (Exception e) {
            System.out.println("消费者在等待新产品的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("消费者终止了消费过程！");
    }
}
```

3.1.6 LinkedBlockingDeque

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque多了addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast等方法，以First单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以Last单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法add等同于addLast，移除方法remove等效于removeFirst。在初始化LinkedBlockingDeque时可以初始化队列的容量，用来防止其再扩容时过渡膨胀。

源码:略

```
package kaikeba.com;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingDeque;

public class LinkedBlockingQueueDemo {
    public static void main(String[] args) {

        BlockingQueue<Integer> blockingQueue = new
LinkedBlockingDeque<Integer>();
        Producer producer = new Producer(blockingQueue);
        Consumer consumer = new Consumer(blockingQueue);
        new Thread(producer).start();
        new Thread(consumer).start();
    }

}

class Producer implements Runnable {

    private  BlockingQueue<Integer> blockingQueue;
    private static int element = 0;

    public Producer(BlockingQueue<Integer> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }
```

```
27
28
29      public void run() {
30          try {
31              while(element < 20) {
32                  System.out.println("生产元素："+element);
33                  blockingQueue.put(element++);
34              }
35          } catch (Exception e) {
36              System.out.println("生产者在等待空闲空间的时候发生异常！");
37              e.printStackTrace();
38          }
39          System.out.println("生产者终止了生产过程！");
40      }
41  }
42   class Consumer implements Runnable {
43      private  BlockingQueue<Integer> blockingQueue;
44      public Consumer(BlockingQueue<Integer> blockingQueue) {
45          this.blockingQueue = blockingQueue;
46      }
47
48
49      public void run() {
50          try {
51              while(true) {
52                  System.out.println("消费元素："+blockingQueue.take());
53              }
54          } catch (Exception e) {
55              System.out.println("消费者在等待新产品的时候发生异常！");
56              e.printStackTrace();
57          }
58          System.out.println("消费者终止了消费过程！");
59      }
60  }
```

3.1.7 SynchronousQueue

SynchronousQueue是一个没有数据缓冲的BlockingQueue，生产者线程对其的插入操作put必须等待消费者的移除操作take，反过来也一样。

SynchronousQueue内部并没有数据缓存空间，你不能调用peek()方法来看队列中是否有数据元素，因为数据元素只有当你试着取走的时候才可能存在，不取走而只想偷窥一下是不行的，当然遍历这个队列的操作也是不允许的。

数据是在配对的生产者和消费者线程之间直接传递的，并不会将数据缓冲到队列中。

SynchronousQueue支持公平访问队列，默认情况下，线程采用非公平策略，如果使用公平策略，等待的线程采用先进先出的顺序访问队列。

SynchronousQueue适合传递性场景，一个使用场景是在线程池里。
Executors.newCachedThreadPool()就使用了SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了60秒后会被回收。

源码略

使用

```java
package kaikeba.com;

import java.util.concurrent.*;

public class SynchronousQueueDemo {
    public static void main(String[] args) {

        BlockingQueue<Integer> blockingQueue = new SynchronousQueue<>();
        Producer producer = new Producer(blockingQueue);
        Consumer consumer = new Consumer(blockingQueue);

        new Thread(producer).start();
        new Thread(consumer).start();

    }

}

class Producer implements Runnable {

    private  BlockingQueue<Integer> blockingQueue;
    private static int element = 0;

    public Producer(BlockingQueue<Integer> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }

    public void run() {
        try {
            while(element < 20) {
                System.out.println("生产元素："+element);
                blockingQueue.put(element++);
            }
        } catch (Exception e) {
            System.out.println("生产者在等待空闲空间的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("生产者终止了生产过程！");
    }
}
 class Consumer implements Runnable {

    private  BlockingQueue<Integer> blockingQueue;

    public Consumer(BlockingQueue<Integer> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }


    public void run() {
        try {

            while(true) {
                Thread.sleep(1000l);
                System.out.println("消费元素："+blockingQueue.take());
            }
        } catch (Exception e) {
```

```
            System.out.println("消费者在等待新产品的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("消费者终止了消费过程！");
    }
}
```

3.1.8 LinkedTransferQueue

LinkedTransferQueue是一个由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列LinkedTransferQueue多了tryTransfer和transfer方法。

transfer方法。如果当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可以把生产者传入的元素立刻transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回。transfer方法的关键代码如下：

第一行代码是试图把存放当前元素的s节点作为tail节点。第二行代码是让CPU自旋等待消费者消费元素。因为自旋会消耗CPU，所以自旋一定的次数后使用Thread.yield()方法来暂停当前正在执行的线程，并执行其他线程。

tryTransfer方法。则是用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回false。和transfer方法的区别是tryTransfer方法无论消费者是否接收，方法立即返回。而transfer方法是必须等到消费者消费了才返回。

对于带有时间限制的tryTransfer(E e, long timeout, TimeUnit unit)方法，则是试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回false，如果在超时时间内消费了元素，则返回true。

源码：略

使用：

```
package kaikeba.com;

import java.util.concurrent.*;

public class LinkedTransferQueueTest {
    public static void main(String[] args) {

        LinkedTransferQueue<Integer> blockingQueue = new
LinkedTransferQueue<Integer>();
        Producer producer = new Producer(blockingQueue);
        Consumer consumer = new Consumer(blockingQueue);

        new Thread(producer).start();
        new Thread(consumer).start();
    }

}

class Producer implements Runnable {

    private  LinkedTransferQueue<Integer> linkedTransferQueue;
    private static int element = 0;

    public Producer(LinkedTransferQueue<Integer> linkedTransferQueue) {
        this.linkedTransferQueue = linkedTransferQueue;
```

```
    }


    public void run() {
        try {
            while(element < 20) {
                System.out.println("生产元素："+element);
                linkedTransferQueue.put(element++);
            }
        } catch (Exception e) {
            System.out.println("生产者在等待空闲空间的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("生产者终止了生产过程！");
    }
}
 class Consumer implements Runnable {

    private  LinkedTransferQueue<Integer> linkedTransferQueue;

    public Consumer(LinkedTransferQueue<Integer> linkedTransferQueue) {
        this.linkedTransferQueue = linkedTransferQueue;
    }


    public void run() {
        try {

            while(true) {
                Thread.sleep(1000l);
                System.out.println("消费元素："+linkedTransferQueue.take());
            }
        } catch (Exception e) {
            System.out.println("消费者在等待新产品的时候发生异常！");
            e.printStackTrace();
        }
        System.out.println("消费者终止了消费过程！");
    }

```

3.1.9 ConcurrentHashMap

HashMap容量

```
/**
 * Constructs an empty <tt>HashMap</tt> with the specified initial
 * capacity and load factor.
 *
 * @param  initialCapacity the initial capacity
 * @param  loadFactor      the load factor
 * @throws IllegalArgumentException if the initial capacity is negative
 *         or the load factor is nonpositive
 */
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
```

```
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

给定的默认容量为 16，负载因子为 0.75。Map 在使用过程中不断的往里面存放数据，当数量达到了 `16 * 0.75 = 12` 就需要将当前 16 的容量进行扩容，而扩容这个过程涉及到 rehash、复制数据等操作，所以非常消耗性能。

因此通常建议能提前预估 HashMap 的大小最好，尽量的减少扩容带来的性能损耗。

线程不安全的 HashMap

因为多线程环境下，使用 HashMap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，所以在并发情况下不能使用 HashMap，如以下代码

```
final HashMap<String, String> map = new HashMap<String, String>(2);
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    map.put(UUID.randomUUID().toString(), "");
                }
            }, "kaikeba" + i).start();
        }
    }
}, "kaikeba");
t.start();
t.join();
```

效率低下的 HashTable 容器

HashTable 容器使用 syncronized来保证线程安全，但在线程竞争激烈的情况下 HashTable 的效率非常低下。因为当一个线程访问 HashTable 的同步方法时，其他线程访问 HashTable 的同步方法时，可能会进入阻塞或轮询状态。如线程 1 使用 put 进行添加元素，线程 2 不但不能使用 put 方法添加元素，并且也不能使用 get 方法来获取元素，所以竞争越激烈效率越低。

源码：略。

栗子：

```
package kaikeba.com;

import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapTest {

    public static void main(String[] args) {
```

```
        Map<String, String> map = new ConcurrentHashMap<String, String>();
        map.put("key1", "1");
        map.put("key2", "2");
        map.put("key3", "3");
        map.put("key4", "4");
        Iterator<String> it = map.keySet().iterator();

        while (it.hasNext()) {
            String key = it.next();
            System.out.println(key + ","+ map.get(key));

        }
    }

}

```

3.1.10 ConcurrentSkipListMap

JDK1.6时，为了对高并发环境下的有序Map提供更好的支持，J.U.C新增了一个**ConcurrentNavigableMap**接口，ConcurrentNavigableMap很简单，它同时实现了NavigableMap和ConcurrentMap接口。

ConcurrentNavigableMap接口提供的功能也和NavigableMap几乎完全一致，很多方法仅仅是返回的类型不同。

NavigableMap接口，进一步扩展了SortedMap的功能，提供了根据指定Key返回最接近项、按升序/降序返回所有键的视图等功能。

J.U.C提供了基于ConcurrentNavigableMap接口的一个实现——`ConcurrentSkipListMap`。ConcurrentSkipListMap可以看成是并发版本的TreeMap，但是和TreeMap不同是，ConcurrentSkipListMap并不是基于红黑树实现的，其底层是一种类似跳表（Skip List）的结构。

栗子：

```
package kaikeba.com;

import java.util.Map;
import java.util.concurrent.ConcurrentNavigableMap;
import java.util.concurrent.ConcurrentSkipListMap;

public class ConcurrentSkipListMapTest {
        public static void main(String[] args) {
            ConcurrentSkipListMap<String, Contact> map = new
ConcurrentSkipListMap<>();
            Thread threads[]=new Thread[25];
            int counter=0;
            //创建和启动25个任务，对于每个任务指定一个大写字母作为ID
            for (char i='A'; i<'Z'; i++) {
                Task0 task=new Task0(map, String.valueOf(i));
                threads[counter]=new Thread(task);
                threads[counter].start();
                counter++;
            }
            //使用join()方法等待线程的结束
            for (int i=0; i<25; i++) {
```

```
                try {
                    threads[i].join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.printf("Size of the map: %d\n",map.size());
            Map.Entry<String, Contact> element;
            Contact contact;
           // 使用firstEntry()方法获取map的第一个实体，并输出。
            element=map.firstEntry();
            contact=element.getValue();
            System.out.printf("First Entry: %s: %s\n",contact.
                    getName(),contact.getPhone());
            //使用lastEntry()方法获取map的最后一个实体，并输出。
            element=map.lastEntry();
            contact=element.getValue();
            System.out.printf("Last Entry: %s: %s\n",contact.
                    getName(),contact.getPhone());
            //使用subMap()方法获取map的子map，并输出。

            System.out.printf("Submap from A1996 to B1002: \n");

            ConcurrentNavigableMap<String, Contact> submap=map.

            subMap("A1996", "B1001");

            do {
                element=submap.pollFirstEntry();
                if (element!=null) {
                    contact=element.getValue();
                    System.out.printf("%s: %s\n",contact.getName(),contact.
                            getPhone());
                }
            } while (element!=null);
        }

        }

class Contact {
    private String name;
    private String phone;

    public Contact(String name, String phone) {
        this.name = name;
        this.phone = phone;

    }

    public String getName() {
        return name;
    }

    public String getPhone() {
        return phone;
    }
}

```

```
class Task0 implements Runnable {

    private ConcurrentSkipListMap<String, Contact> map;
    private String id;

    public Task0(ConcurrentSkipListMap<String, Contact> map, String id) {
        this.id = id;
        this.map = map;
    }

    @Override
    public void run() {

        for (int i = 0; i < 1000; i++) {
            Contact contact = new Contact(id, String.valueOf(i + 1000));
            map.put(id + contact.getPhone(), contact);
        }
    }
}
```

3.1.11 ConcurrentSkipListSet

ConcurrentSkipListSet，是JDK1.6时J.U.C新增的一个集合工具类，它是一种有序的SET类型。

ConcurrentSkipListSet实现了NavigableSet接口，ConcurrentSkipListMap实现了NavigableMap接口，以提供和排序相关的功能，维持元素的有序性，所以ConcurrentSkipListSet就是一种为并发环境设计的有序SET工具类。

ConcurrentSkipListSet底层实现引用了ConcurrentSkipListMap。

栗子：

```
package kaikeba.com;

import java.util.concurrent.ConcurrentSkipListSet;

public class ConcurrentSkipListSetTest {
    public static void main(String[] args) {
        ConcurrentSkipListSet<Contact1> set = new ConcurrentSkipListSet<>
();
        Thread threads[]=new Thread[25];
        int counter=0;
        //创建和启动25个任务，对于每个任务指定一个大写字母作为ID
        for (char i='A'; i<'Z'; i++) {
            Task1 task=new Task1(set, String.valueOf(i));
            threads[counter]=new Thread(task);
            threads[counter].start();
            counter++;
        }
        //使用join()方法等待线程的结束
        for (int i=0; i<25; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
        System.out.printf("Size of the set: %d\n",set.size());
        Contact1 contact;
        // 使用first方法获取set的第一个实体，并输出。
        contact=set.first();
        System.out.printf("First Entry: %s: %s\n",contact.
                getName(),contact.getPhone());
        //使用last方法获取set的最后一个实体，并输出。
        contact=set.last();
        System.out.printf("Last Entry: %s: %s\n",contact.
                getName(),contact.getPhone());

    }

}

class Contact1 implements Comparable<Contact1> {
    private String name;
    private String phone;

    public Contact1(String name, String phone) {
        this.name = name;
        this.phone = phone;

    }

    public String getName() {
        return name;
    }

    public String getPhone() {
        return phone;
    }

    @Override
    public int compareTo(Contact1 o) {
        return name.compareTo(o.name);
    }
}

class Task1 implements Runnable {

    private ConcurrentSkipListSet<Contact1> set;
    private String id;

    public Task1(ConcurrentSkipListSet<Contact1> set, String id) {
        this.id = id;
        this.set = set;
    }

    @Override
    public void run() {

        for (int i = 0; i < 100; i++) {
            Contact1 contact = new Contact1(id, String.valueOf(i + 100));
            set.add(contact);
        }
    }
```

```
82 }
```

3.1.12 CopyOnWriteArrayList

Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，可以在非常多的并发场景中使用到。

什么是CopyOnWrite容器

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要注意一下。

内存占用问题。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。

数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

源码：略。

使用：

```java
package kaikeba.com;

import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class ReadThread implements Runnable {
    private List<Integer> list;

    public ReadThread(List<Integer> list) {
        this.list = list;
    }

    @Override
    public void run() {
        System.out.print("size:="+list.size()+",::");
        for (Integer ele : list) {
```

```
            System.out.print(ele + ",");
        }
        System.out.println();
    }
}

class WriteThread implements Runnable {
    private List<Integer> list;

    public WriteThread(List<Integer> list) {
        this.list = list;
    }

    @Override
    public void run() {
        this.list.add(9);
    }
}


public class TestCopyOnWriteArrayListTest {

    private void test() {
        //1、初始化CopyOnWriteArrayList
        List<Integer> tempList = Arrays.asList(new Integer [] {1,2});
        CopyOnWriteArrayList<Integer> copyList = new CopyOnWriteArrayList<>
(tempList);


        //2、模拟多线程对list进行读和写
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        executorService.execute(new ReadThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new ReadThread(copyList));
        executorService.execute(new WriteThread(copyList));
        executorService.execute(new ReadThread(copyList));
        executorService.execute(new WriteThread(copyList));
        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("copyList size:"+copyList.size());
    }


    public static void main(String[] args) {
        new TestCopyOnWriteArrayList().test();
    }
}
```

3.1.13 CopyOnWriteArraySet

CopyOnWriteArraySet相对CopyOnWriteArrayList用来存储不重复的对象，是线程安全的。虽然继承了AbstractSet类，但CopyOnWriteArraySet与HashMap 完全不同，内部是用CopyOnWriteArrayList实现的，实现不重复的特性也是直接调用CopyOnWriteArrayList的方法实现的，感觉加的最有用的函数就是eq函数判断对象是否相同

####

3.2 原子操作类

在并发编程中很容易出现并发安全的问题，有一个很简单的例子就是多线程更新变量i=1,比如多个线程执行i++操作，就有可能获取不到正确的值，而这个问题，最常用的方法是通过Synchronized进行控制来达到线程安全的目的。但是由于synchronized是采用的是悲观锁策略，并不是特别高效的一种解决方案。实际上，在J.U.C下的atomic包提供了一系列的操作简单，性能高效，并能保证线程安全的类去更新基本类型变量，数组元素，引用类型以及更新对象中的字段类型。atomic包下的这些类都是采用的是乐观锁策略去原子更新数据，在java中则是使用CAS操作具体实现。

3.2.1 原子基本数据类型

原子更新基本类型

atomic包提高原子更新基本类型的工具类，主要有这些：

- AtomicBoolean：以原子更新的方式更新boolean；
- AtomicInteger：以原子更新的方式更新Integer;
- AtomicLong：以原子更新的方式更新Long；

AtomicInteger常用的方法:

- addAndGet(int delta) ：以原子方式将输入的数值与实例中原本的值相加，并返回最后的结果；
- incrementAndGet() ：以原子的方式将实例中的原值进行加1操作，并返回最终相加后的结果；
- getAndSet(int newValue)：将实例中的值更新为新值，并返回旧值；
- getAndIncrement()：以原子的方式将实例中的原值加1，返回的是自增前的旧值；

```
1 public final int getAndIncrement() {
2     return unsafe.getAndAddInt(this, valueOffset, 1);
3 }
```

可以看出，该方法实际上是调用了unsafe实例的getAndAddInt方法，unsafe实例的获取时通过UnSafe类的静态方法getUnsafe获取：

```
1 private static final Unsafe unsafe = Unsafe.getUnsafe();
```

Unsafe类在sun.misc包下，Unsafer类提供了一些底层操作，atomic包下的原子操作类的也主要是通过Unsafe类提供的compareAndSwapInt，compareAndSwapLong等一系列提供CAS操作的方法来进行实现。

atomicInteger借助了UnSafe提供的CAS操作能够保证数据更新的时候是线程安全的，并且由于CAS是采用乐观锁策略，因此，这种数据更新的方法也具有高效性。

AtomicLong的实现原理和AtomicInteger一致，只不过一个针对的是long变量，一个针对的是int变量。而boolean变量的更新类AtomicBoolean类是怎样实现更新的呢?核心方法是`compareAndSet`t方法，其源码如下：

```
public final boolean compareAndSet(boolean expect, boolean update) {
    int e = expect ? 1 : 0;
    int u = update ? 1 : 0;
    return unsafe.compareAndSwapInt(this, valueOffset, e, u);
}
```

可以看出，compareAndSet方法的实际上也是先转换成0,1的整型变量，然后是通过针对int型变量的原子更新方法compareAndSwapInt来实现的。可以看出atomic包中只提供了对boolean,int ,long这三种基本类型的原子更新的方法，参考对boolean更新的方式，原子更新char,double,float也可以采用类似的思路进行实现。

使用：

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        AtomicInteger ai = new AtomicInteger();

        List<Thread> list = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new Accumlator(ai), "thread-" + i);
            list.add(t);
            t.start();
        }

        for (Thread t : list) {
            t.join();
        }

        System.out.println(ai.get());
    }

    static class Accumlator implements Runnable {
        private AtomicInteger ai;

        Accumlator(AtomicInteger ai) {
            this.ai = ai;
        }

        @Override
        public void run() {
            for (int i = 0, len = 1000; i < len; i++) {
                ai.incrementAndGet();
            }
        }
    }
}
```

3.2.2 原子数组

atomic包下提供能原子更新数组中元素的类有：

- AtomicIntegerArray：原子更新整型数组中的元素；
- AtomicLongArray：原子更新长整型数组中的元素；
- AtomicReferenceArray：原子更新引用类型数组中的元素

这几个类的用法一致，就以AtomicIntegerArray来总结下常用的方法：

- addAndGet(int i, int delta)：以原子更新的方式将数组中索引为i的元素与输入值相加；
- getAndIncrement(int i)：以原子更新的方式将数组中索引为i的元素自增加1；
- compareAndSet(int i, int expect, int update)：将数组中索引为i的位置的元素进行更新

可以看出，AtomicIntegerArray与AtomicInteger的方法基本一致，只不过在AtomicIntegerArray的方法中会多一个指定数组索引位i。

```
package kaikeba.com;

import java.util.concurrent.atomic.AtomicIntegerArray;

public class AtomicIntegerArrayTest {

    public static void main(String[] args) {
        //创建给定长度的AtomicIntegerArray。
        AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(10);
        //将位置 i 的元素设置为给定值,默认值为0
        atomicIntegerArray.set(9,10);
        System.out.println("Value: " + atomicIntegerArray.get(9) + "默认值：" + atomicIntegerArray.get(0));
        //返回该数组的长度
        System.out.println("数组长度：" + atomicIntegerArray.length());
        //以原子方式先对给定下标加上特定的值，再获取相加后的值
        atomicIntegerArray.set(0,10);
        System.out.println("Value: " + atomicIntegerArray.get(0));
        System.out.println("Value: " + atomicIntegerArray.addAndGet(5,10));
        //如果当前值 == 预期值，将位置 i 的元素设置为给定的更新值。
        Boolean bool = atomicIntegerArray.compareAndSet(5,10,30);
        System.out.println("结果值:  " + atomicIntegerArray.get(5) + " Result: " + bool);
        //以原子方式先将当前下标的值减1，再获取减1后的结果
        System.out.println("下标为5的值为：" + atomicIntegerArray.decrementAndGet(5));
        System.out.println("下标为5的值为：" + atomicIntegerArray.get(5));
        //以原子方式先获取当前下标的值，再将当前下标的值加上给定的值
        Integer result2 = atomicIntegerArray.getAndAdd(5,5);
        System.out.println("下标为5的值为：" + result2);
        System.out.println("下标为5的值为：" + atomicIntegerArray.get(5));
        //以原子方式先获取当前下标的值，再对当前下标的值减1
        System.out.println("下标为1的值为：" + atomicIntegerArray.getAndDecrement(1));
        System.out.println("下标为1的值为：" + atomicIntegerArray.get(1));
        // 以原子方式先获取当前下标的值，再对当前下标的值加1
        System.out.println("下标为2的值为：" + atomicIntegerArray.getAndIncrement(2));
        System.out.println("下标为2的值为：" + atomicIntegerArray.get(2));
        //将位置 i 的元素以原子方式设置为给定值，并返回旧值。
        System.out.println("下标为3的值为：" + atomicIntegerArray.getAndSet(3,50));
        System.out.println("下标为3的值为：" + atomicIntegerArray.get(3));
        //以原子方式先对下标加1再获取值
        System.out.println("下标为4的值为：" + atomicIntegerArray.incrementAndGet(4));
        System.out.println("下标为4的值为：" + atomicIntegerArray.get(4));

    }
```

```
}
```

并发测试：

```
package kaikeba.com;

import java.util.concurrent.atomic.AtomicIntegerArray;

public class AtomicIntegerArrayTest2 {

    static AtomicIntegerArray arr = new AtomicIntegerArray(10);
    public static class AddThread implements Runnable{
        public void run () {
        for (int k = 0; k < 10; k++)
            arr.getAndIncrement(k % arr.length());
    }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] ts = new Thread[10];
        for (int k = 0; k < 10; k++) {
            ts[k] = new Thread(new AddThread());
        }
        for (int k = 0; k < 10; k++) {
            ts[k].start();
        }
        for (int k = 0; k < 10; k++) {
            ts[k].join();
        }
        System.out.println(arr);
    }
}

```

3.2.3 原子引用类型

如果需要原子更新引用类型变量的话，为了保证线程安全，atomic也提供了相关的类：

- AtomicReference
- AtomicStampedReference
- AtomicMarkableReference

AtomicReference的引入是为了可以用一种类似乐观锁的方式操作共享资源，在某些情景下以提升性能。

```
public class AtomicReferenceTest {
    public static void main(String[] args) throws InterruptedException {
        AtomicReference<Integer> ref = new AtomicReference<>(new
Integer(1000));

        List<Thread> list = new ArrayList<>();
        for (int i = 0; i < 1000; i++) {
            Thread t = new Thread(new Task(ref), "Thread-" + i);
            list.add(t);
            t.start();
```

```
        }

        for (Thread t : list) {
            t.join();
        }

        System.out.println(ref.get());
    }

}

class Task implements Runnable {
    private AtomicReference<Integer> ref;

    Task(AtomicReference<Integer> ref) {
        this.ref = ref;
    }

    @Override
    public void run() {
        for (; ; ) {    //自旋操作
            Integer oldV = ref.get();
            if (ref.compareAndSet(oldV, oldV + 1))  // CAS操作
                break;
        }
    }
}
```

该案例并没有使用锁，是使用自旋+CAS的无锁操作保证共享变量的线程安全。

CAS操作可能存在ABA的问题：
假如一个值原来是A，变成了B，又变成了A，那么CAS检查时会发现它的值没有发生变化，但是实际上却变化了。

一般来讲这并不是什么问题，比如数值运算，线程其实根本不关心变量中途如何变化，只要最终的状态和预期值一样即可。

但是，有些操作会依赖于对象的变化过程，此时的解决思路一般就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A - B - A 就会变成1A - 2B - 3A。

AtomicStampedReference就是上面所说的加了版本号的AtomicReference。

```
 private static class Pair<T> {
        final T reference;
        final int stamp;
        private Pair(T reference, int stamp) {
            this.reference = reference;
            this.stamp = stamp;
        }
        static <T> Pair<T> of(T reference, int stamp) {
            return new Pair<T>(reference, stamp);
        }
    }

    private volatile Pair<V> pair;

    /**
     * Creates a new {@code AtomicStampedReference} with the given
```

```
     * initial values.
     *
     * @param initialRef the initial reference
     * @param initialStamp the initial stamp
     */
    public AtomicStampedReference(V initialRef, int initialStamp) {
        pair = Pair.of(initialRef, initialStamp);
    }
}
```

解决ABA问题，引入了AtomicStampedReference。

AtomicStampedReference可以给引用加上版本号，追踪引用的整个变化过程，如：A -> B -> C -> D - > A，通过AtomicStampedReference，可以知道，引用变量中途被更改了3次。

但是，有时候，我们并不关心引用变量更改了几次，只关心是否更改过，就有了AtomicMarkableReference：

AtomicMarkableReference和AtomicStampedReference的唯一区别就是不再用int标识引用，而是使用boolean变量——表示引用变量是否被更改过。

AtomicMarkableReference对于那些不关心引用变化过程，只关心引用变量是否变化过的应用会更加友好。

```
public class AtomicMarkableReference<V> {

    private static class Pair<T> {
        final T reference;
        final boolean mark;
        private Pair(T reference, boolean mark) {
            this.reference = reference;
            this.mark = mark;
        }
        static <T> Pair<T> of(T reference, boolean mark) {
            return new Pair<T>(reference, mark);
        }
    }

    private volatile Pair<V> pair;

    /**
     * Creates a new {@code AtomicMarkableReference} with the given
     * initial values.
     *
     * @param initialRef the initial reference
     * @param initialMark the initial mark
     */
    public AtomicMarkableReference(V initialRef, boolean initialMark) {
        pair = Pair.of(initialRef, initialMark);
    }
}
```

3.2.4 原子更新字段类型

如果需要更新对象的某个字段，并在多线程的情况下，能够保证线程安全，atomic同样也提供了相应的原子操作类：

- AtomicIntegeFieldUpdater：原子更新整型字段类；
- AtomicLongFieldUpdater：原子更新长整型字段类；
- AtomicReferenceFieldUpdater：原子更新引用字段类型；

要想使用原子更新字段需要两步操作：

- 原子更新字段类都是抽象类，只能通过静态方法`newUpdater`来创建一个更新器，并且需要设置想要更新的类和属性；
- 更新类的属性必须使用`public volatile`进行修饰；
- 字段必须是volatile类型的，在线程之间共享变量时保证立即可见
- 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。
- 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。
- 只能是实例变量，不能是类变量，也就是说不能加static关键字。
- 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。
- 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。

这几个类提供的方法基本一致，以AtomicIntegerFieldUpdater为例来看看具体的使用：

```
1  package kaikeba.com;
2  import java.util.concurrent.atomic.AtomicIntegerFieldUpdater;
3
4  public class AtomicIntegerFieldUpdaterTest{
5
6      private static AtomicIntegerFieldUpdater<User> a =
   AtomicIntegerFieldUpdater.newUpdater(User.class, "age");
7
8      public static void main(String[] args) {
9          User user = new User("conan", 10);
10         System.out.println(a.getAndIncrement(user));
11         System.out.println(a.get(user));
12     }
13
14     public static class User {
15         private String name;
16         public volatile int age;
17         public User(String name, int age) {
18             this.name = name;
19             this.age = age;
20         }
21         public String getName() {
22             return name;
23         }
24         public int getAge() {
25             return age;
26         }
27     }
28 }
29
```

从示例中可以看出，创建`AtomicIntegerFieldUpdater`是通过它提供的静态方法进行创建，`getAndAdd`方法会将指定的字段加上输入的值，并且返回相加之前的值。user对象中age字段原值为1，加5之后，可以看出user对象中的age字段的值已经变成了6。

3.3 锁：Lock

`java.util.concurrent.locks`包，该包提供了一系列基础的锁工具，用以对synchronizd、wait、notify等进行补充、增强。
juc-locks锁框架中一共就三个接口：Lock、Condition、ReadWriteLock。

Hierarchy For Package java.util.concurrent.locks

Package Hierarchies:
All Packages

Class Hierarchy

- java.lang.**Object**
  - java.util.concurrent.locks.**AbstractOwnableSynchronizer** (implements java.io.Serializable)
    - java.util.concurrent.locks.**AbstractQueuedLongSynchronizer** (implements java.io.Serializable)
    - java.util.concurrent.locks.**AbstractQueuedSynchronizer** (implements java.io.Serializable)
  - java.util.concurrent.locks.**AbstractQueuedLongSynchronizer.ConditionObject** (implements java.util.concurrent.locks.Condition, java.io.Serializable)
  - java.util.concurrent.locks.**AbstractQueuedSynchronizer.ConditionObject** (implements java.util.concurrent.locks.Condition, java.io.Serializable)
  - java.util.concurrent.locks.**LockSupport**
  - java.util.concurrent.locks.**ReentrantLock** (implements java.util.concurrent.locks.Lock, java.io.Serializable)
  - java.util.concurrent.locks.**ReentrantReadWriteLock** (implements java.util.concurrent.locks.ReadWriteLock, java.io.Serializable)
  - java.util.concurrent.locks.**ReentrantReadWriteLock.ReadLock** (implements java.util.concurrent.locks.Lock, java.io.Serializable)
  - java.util.concurrent.locks.**ReentrantReadWriteLock.WriteLock** (implements java.util.concurrent.locks.Lock, java.io.Serializable)
  - java.util.concurrent.locks.**StampedLock** (implements java.io.Serializable)

Interface Hierarchy

- java.util.concurrent.locks.**Condition**
- java.util.concurrent.locks.**Lock**
- java.util.concurrent.locks.**ReadWriteLock**

3.3.1 ReentrantLock

ReentrantLock叫做可重入锁，指的是线程可以重复获取同一把锁，或者说该锁支持一个线程对资源的重复加锁。同时该锁还支持获取锁的公平性和非公平性选择，锁的公平性是指，在绝对时间上，先对锁获取的请求一定先被满足，也就是等待时间最长的那个线程优先获得，可以说，锁的获取是顺序的，即符合FIFO规则。

ReentrantLock也是互斥锁，因此也可以保证原子性。

ReentrantLock 重入锁的基本原理是判断上次获取锁的线程是否为当前线程，如果是则可再次进入临界区，如果不是，则阻塞。

由于ReentrantLock是基于AQS实现的，底层通过操作同步状态来获取锁.，下面看一下非公平锁的实现逻辑：

```
final boolean nonfairTryAcquire(int acquires) {
            //获取当前线程
            final Thread current = Thread.currentThread();
            //通过AQS获取同步状态
            int c = getState();
            //同步状态为0，说明临界区处于无锁状态，
            if (c == 0) {
                //修改同步状态，即加锁
                if (compareAndSetState(0, acquires)) {
                    //将当前线程设置为锁的owner
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            //如果临界区处于锁定状态，且上次获取锁的线程为当前线程
            else if (current == getExclusiveOwnerThread()) {
                 //则递增同步状态
                int nextc = c + acquires;
                if (nextc < 0) // overflow
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
```

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

成功获取锁的线程再次获取锁，只是增加了同步状态值，在释放同步转态时，相应的减少同步状态值，实现如下：

```
        protected final boolean tryRelease(int releases) {
            int c = getState() - releases;
            if (Thread.currentThread() != getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();
            boolean free = false;
          //在同步状态完全释放了，设置true
            if (c == 0) {
                free = true;
                setExclusiveOwnerThread(null);
            }
            setState(c);
            return free;
        }
```

公平锁和非公平锁的测试：

```
package kaikeba.com;

import org.junit.Test;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;


public class ReentrantLockTest {
    private static Lock fairLock = new ReentrantLockMine(true);
    private static Lock unfairLock = new ReentrantLockMine(false);

    @Test
    public void unfair() throws InterruptedException {
        testLock("unfair lock", unfairLock);
    }

    @Test
    public void fair() throws InterruptedException {
        testLock("fair lock", fairLock);
    }

    private void testLock(String type, Lock lock) throws
InterruptedException {
        System.out.println(type);
        for (int i = 0; i < 5; i++) {
            Thread thread = new Thread(new Job(lock)){
                public String toString() {
                    return getName();
```

```
                }
            };
            thread.setName("" + i);
            thread.start();
        }
        Thread.sleep(11000);
    }

    private static class Job implements Runnable{
        private Lock lock;
        public Job(Lock lock) {
            this.lock = lock;
        }

        public void run() {
            for (int i = 0; i < 2; i++) {
                lock.lock();
                try {
                    Thread.sleep(1000);
                    System.out.println("获取锁的当前线程[" +
Thread.currentThread().getName() + "], 同步队列中的线程" +
((ReentrantLockMine)lock).getQueuedThreads() + "");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        }
    }

    private static class ReentrantLockMine extends ReentrantLock {  //重新实
现ReentrantLock类是为了重写getQueuedThreads方法，便于我们试验的观察
        public ReentrantLockMine(boolean fair) {
            super(fair);
        }

        @Override
        protected Collection<Thread> getQueuedThreads() {   //获取同步队列中的
线程
            List<Thread> arrayList = new ArrayList<Thread>
(super.getQueuedThreads());
            Collections.reverse(arrayList);
            return arrayList;
        }
    }

```

非公平锁的获取，只要获取了同步状态就可以获取锁，有可能导致饥饿现象，但是非公平锁，线程的切换比较少，更高效。

ReentrantLock与synchronized的区别

- 重入
  synchronized可重入，因为加锁和解锁自动进行，不必担心最后是否释放锁；ReentrantLock也可重入，但加锁和解锁需要手动进行，且次数需一样，否则其他线程无法获得锁。

- 实现
  synchronized是JVM实现的、而ReentrantLock是JDK实现的。说白了就是，是操作系统来实现，还是用户自己敲代码实现。
- 性能
  在 Java 的 1.5 版本中，synchronized 性能不如 SDK 里面的 Lock，但 1.6 版本之后，synchronized 做了很多优化，将性能追了上来。
- 功能
  ReentrantLock锁的细粒度和灵活度，优于synchronized。

  ReentrantLock不同点一：可在构造函数中指定是公平锁还是非公平锁，而synchronized只能是非公平锁。

```
private static final ReentrantLock reentrantLock = new ReentrantLock(true);
```

ReentrantLock不同点二：可以避免死锁问题，因为它可以非阻塞地获取锁。如果尝试获取锁失败，并不进入阻塞状态，而是直接返回false，这时候线程不用阻塞等待，可以先去做其他事情。所以不会造成死锁。

```
// 支持非阻塞获取锁的 API
boolean tryLock();
```

tryLock还支持超时。调用tryLock时没有获取到锁，会等待一段时间，如果线程在一段时间之内还是没有获取到锁，不是进入阻塞状态，而是throws InterruptedException，那这个线程也有机会释放曾经持有的锁，这样也能破坏死锁不可抢占条件。

```
boolean tryLock(long time, TimeUnit unit)
```

ReentrantLock不同点三：提供能够中断等待锁机制。

synchronized 的问题是，持有锁 A 后，如果尝试获取锁 B 失败，那么线程就进入阻塞状态，一旦发生死锁，就没有任何机会来唤醒阻塞的线程。

但如果阻塞状态的线程能够响应中断信号，也就是说当我们给阻塞的线程发送中断信号的时候，能够唤醒它，那它就有机会释放曾经持有的锁 A。ReentrantLock可以用lockInterruptibly方法来实现。

ReentrantLock不同点四：可以用J.U.C包中的Condition实现分组唤醒需要等待的线程。而synchronized只能notify或者notifyAll。

3.3.2 LockSupport

LockSupport类，是JUC包中的一个工具类，定义了一组静态方法，提供最基本的线程阻塞和唤醒功能，是构建同步组件的基础工具，用来创建锁和其他同步类的基本线程阻塞原语。

LockSupport类的核心方法其实就两个：park() 和 unpark()，其中 park() 方法用来阻塞线程，unpark()方法用于唤醒指定线程。

和Object类的wait() 和 signal() 方法有些类似，但是LockSupport的这两种方法从语意上讲比Object类的方法更清晰，而且可以针对指定线程进行阻塞和唤醒。

LockSupport类使用了一种名为Permit（许可）的概念来做到阻塞和唤醒线程的功能，可以把许可看成是一种（0，1）信号量（Semaphore），但与Semaphore不同的是，许可的量加上限1。

初始时，permit为0，当调用 unpark() 方法时，线程的permit加1，当调用 park()方法时，如果permit为0，则调用线程进入阻塞状态。

假设现在需要实现一种FIFO类型的独占锁，可以把这种锁看成是ReentrantLock的公平锁简单版本，且是不可重入的，就是说当一个线程获得锁后，其他等待线程以FIFO的调度方式等待获取锁。

```
package kaikeba.com;

import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.locks.LockSupport;

 class FIFOMutex  {
    private final AtomicBoolean locked = new AtomicBoolean(false);
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<>();

    public void lock(){
        Thread current = Thread.currentThread();
        waiters.add(current);
        // 如果当前线程不在队首，或锁已被占用，则当前线程阻塞
        // 这个判断的内在意图：锁必须由队首元素拿到
        while (waiters.peek() != current ||
!locked.compareAndSet(false,true)){
            LockSupport.park();
        }
        waiters.remove();// 删除队首元素
    }


    public void unlock(){
        locked.set(false);
        LockSupport.unpark(waiters.peek());
    }
}
public class TestFIFOMutex {
    public static void main(String[] args) throws InterruptedException {
        FIFOMutex mutex = new FIFOMutex();
        MyThread a1 = new MyThread("a", mutex);
        MyThread a2 = new MyThread("b", mutex);
        MyThread a3 = new MyThread("c", mutex);
        a1.start();
        a2.start();
        a3.start();
        a1.join();
        a2.join();
        a3.join();
        System.out.println("Finished");
    }
}

class MyThread extends Thread{
    private String name;
    private FIFOMutex mutex;
    private static int count;
    public MyThread(String name, FIFOMutex mutex) {
        this.name = name;
        this.mutex = mutex;

    }
```

```
54
55      @Override
56      public void run() {
57          for (int i = 0; i < 20; i++) {
58              mutex.lock();
59              count++;
60              System.out.println("thread:"+Thread.currentThread().getName()+"
    name:" + name + " count:" + count);
61              mutex.unlock();
62          }
63      }
64  }
```

上述FIFOMutex类的实现中，当判断锁已被占用时，会调用 LockSupport.park(this) 方法，将当前调用线程阻塞；当使用完锁时，会调用 LockSupport.unpark(waiters.peek()) 方法将等待队列中的队首线程唤醒。

通过LockSupport的这两个方法，可以很方便的阻塞和唤醒线程。

- park 方法是会响应中断的，但是不会抛出异常。（也就是说如果当前调用线程被中断，则会立即返回但不会抛出中断异常）
- park 的重载方法 park(Object blocker)，会传入一个blocker对象，所谓Blocker对象，其实就是当前线程调用时所在调用对象（如上述示例中的FIFOMutex对象）。该对象一般供监视、诊断工具确定线程受阻塞的原因时使用。

3.3.3 Condition

在没有Lock之前，我们使用synchronized来控制同步，配合Object的wait()、wait(long timeout)、notify()、以及notifyAll 等方法可以实现等待/通知模式。

Condition接口也提供了类似于Object的监听器方法、与Lock接口配合可以实现等待/通知模式，但是两者还是有很大区别的，下图是两者的对比：

对比项	Object Monitor Methods	Condition
前置条件	获取对象的锁	调用 Lock.lock() 获取锁 调用 Lock.newCondition() 获取 Condition 对象
调用方式	直接调用 如：object.wait()	直接调用 如：condition.await()
等待队列个数	一个	多个
当前线程释放锁并进入等待状态	支持	支持
当前线程释放锁并进入等待状态，在等待状态中不响应中断	不支持	支持
当前线程释放锁并进入超时等待状态	支持	支持
当前线程释放锁并进入等待状态到将来的某个时间	不支持	支持
唤醒等待队列中的一个线程	支持	支持
唤醒等待队列中的全部线程	支持	支持

Condition 将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set（wait-set）。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。

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。

Condition 实例实质上被绑定到一个锁上。要为特定 Lock 实例获得 Condition 实例，请使用其 newCondition() 方法。

核心方法

Condition提供了一系列的方法来对阻塞和唤醒线程：

- **await()**：造成当前线程在接到信号或被中断之前一直处于等待状态。
- **await(long time, TimeUnit unit)** ：造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。
- **awaitNanos(long nanosTimeout)** ：造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。返回值表示剩余时间，如果在nanosTimesout之前唤醒，那么返回值 = nanosTimeout - 消耗时间，如果返回值 <= 0 ,则可以认定它已经超时了。
- **awaitUninterruptibly()** ：造成当前线程在接到信号之前一直处于等待状态。【注意：该方法对中断不敏感】。
- **awaitUntil(Date deadline)** ：造成当前线程在接到信号、被中断或到达指定最后期限之前一直处于等待状态。如果没有到指定时间就被通知，则返回true，否则表示到了指定时间，返回返回false。
- **signal()** ：唤醒一个等待线程。该线程从等待方法返回前必须获得与Condition相关的锁。
- **signalAll()** ：唤醒所有等待线程。能够从等待方法返回的线程必须获得与Condition相关的锁。

Condition是一种广义上的条件队列。他为线程提供了一种更为灵活的等待/通知模式，线程在调用await方法后执行挂起操作，直到线程等待的某个条件为真时才会被唤醒。Condition必须要配合锁一起使用，因为对共享状态变量的访问发生在多线程环境下。一个Condition的实例必须与一个Lock绑定，因此Condition一般都是作为Lock的内部实现。

源码探索

获取一个Condition必须要通过Lock的newCondition()方法。该方法定义在接口Lock下，返回的结果是绑定到此 Lock 实例的新 Condition 实例。

Condition为一个接口，其下仅有一个实现类ConditionObject，由于Condition的操作需要获取相关的锁，而AQS则是同步锁的实现基础，所以ConditionObject则定义为AQS的内部类。

```
public class ConditionObject implements Condition, java.io.Serializable {
    // 省略方法
}
```

等待队列

每个Condition对象都包含着一个FIFO队列，该队列是Condition对象通知/等待功能的关键。

在队列中每一个节点都包含着一个线程引用，该线程就是在该Condition对象上等待的线程。

Condition的定义：

```
public class ConditionObject implements Condition, java.io.Serializable {
        private static final long serialVersionUID = 1173984872572414699L;

        /** First node of condition queue. */
        private transient Node firstWaiter;

        /** Last node of condition queue. */
        private transient Node lastWaiter;

        /**
         * Creates a new {@code ConditionObject} instance.
```

```
         */
        public ConditionObject() { }

        // Internal methods
        // 省略方法
}
```

从上面代码可以看出Condition拥有首节点（firstWaiter），尾节点（lastWaiter）。

当前线程调用await()方法，将会以当前线程构造成一个节点（Node），并将节点加入到该队列的尾部。

Node里面包含了当前线程的引用。Node定义与AQS的CLH同步队列的节点使用的都是同一个类。

Condition的队列结构比CLH同步队列的结构简单些，新增过程较为简单只需要将原尾节点的nextWaiter指向新增节点，然后更新lastWaiter即可。

等待

调用Condition的await()方法会使当前线程进入等待状态，同时会加入到Condition等待队列并释放锁。

当从await()方法返回时，当前线程一定是获取了Condition相的锁。

```
public final void await() throws InterruptedException {
            // 当前线程中断、直接异常
            if (Thread.interrupted())
                throw new InterruptedException();

            //加入等待队列
            Node node = addConditionWaiter();

            //释放锁
            int savedState = fullyRelease(node);

            int interruptMode = 0;

            //检测当前节点是否在同步队列上、如果不在则说明该节点没有资格竞争锁，继续等待。
            while (!isOnSyncQueue(node)) {

                // 挂起线程
                LockSupport.park(this);

                // 线程释是否被中断，中断直接退出
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
            }

            // 获取同步状态
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;

            // 清理条件队列
            if (node.nextWaiter != null) // clean up if cancelled
                unlinkCancelledWaiters();
```

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

此段代码的逻辑是：
首先将当前线程新建一个节点同时加入到等待队列中，然后释放当前线程持有的同步状态。

然后则是不断检测该节点代表的线程是否出现在CLH同步队列中（收到signal信号之后就会在AQS队列中检测到），如果不存在则一直挂起，否则参与竞争同步状态。

加入条件队列（addConditionWaiter()）源码如下

```
private Node addConditionWaiter() {

            //队列的尾节点
            Node t = lastWaiter;
            // If lastWaiter is cancelled, clean out.

            // 如果该节点的状态的不是CONDITION，则说明该节点不在等待队列上，需要清除
            if (t != null && t.waitStatus != Node.CONDITION) {

                // 清除等待队列中状态不为CONDITION的节点
                unlinkCancelledWaiters();

                //清除后重新获取尾节点
                t = lastWaiter;
            }

            // 将当前线程构造成等待节点
            Node node = new Node(Thread.currentThread(), Node.CONDITION);

            // 将node节点添加到等待队列的尾部

            if (t == null)
                firstWaiter = node;
            else
                t.nextWaiter = node;
            lastWaiter = node;
            return node;
        }
```

该方法主要是将当前线程加入到Condition条件队列中。当然在加入到尾节点之前会清除所有状态不为Condition的节点。

fullyRelease(Node node)，负责释放该线程持有的锁

```
final int fullyRelease(Node node) {
        boolean failed = true;
        try {

            // 获取节点持有锁的数量
            int savedState = getState();

            // 释放锁也就是释放共享状态
            if (release(savedState)) {
                failed = false;
                return savedState;
```

```
            } else {
                throw new IllegalMonitorStateException();
            }
        } finally {
            if (failed)
                node.waitStatus = Node.CANCELLED;
        }
    }
```

isOnSyncQueue(Node node)：如果一个节点刚开始在条件队列上，现在在同步队列上获取锁则返回true。

```
final boolean isOnSyncQueue(Node node) {

        // 状态为CONDITION 、前驱节点为空，返回false
        if (node.waitStatus == Node.CONDITION || node.prev == null)
            return false;

         // 如果后继节点不为空，则说明节点肯定在同步队列中
        if (node.next != null) // If has successor, it must be on queue
            return true;
        /*
         * node.prev can be non-null, but not yet on queue because
         * the CAS to place it on queue can fail. So we have to
         * traverse from tail to make sure it actually made it.  It
         * will always be near the tail in calls to this method, and
         * unless the CAS failed (which is unlikely), it will be
         * there, so we hardly ever traverse much.
         */
        return findNodeFromTail(node);
    }
```

unlinkCancelledWaiters()：负责将条件队列中状态不为Condition的节点删除。

```
private void unlinkCancelledWaiters() {

            // 首节点
            Node t = firstWaiter;

            Node trail = null;
            // 从头开始清除状态不为CONDITION的节点
            while (t != null) {
                Node next = t.nextWaiter;

                if (t.waitStatus != Node.CONDITION) {
                    t.nextWaiter = null;
                    if (trail == null)
                        firstWaiter = next;
                    else
                        trail.nextWaiter = next;
                    if (next == null)
                        lastWaiter = trail;
                }
                else
                    trail = t;
                t = next;
```

```
            }
        }
```

通知

调用Condition的signal()方法，将会唤醒在等待队列中等待最长时间的节点（条件队列里的首节点），在唤醒节点前，会将节点移到同步队列中。

```
public final void signal() {

            // 如果同步是以独占方式进行的，则返回 true；其他情况则返回 false
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();

            // 唤醒首节点
            Node first = firstWaiter;

            if (first != null)
                doSignal(first);
        }

```

该方法首先会判断当前线程是否已经获得了锁，这是前置条件。然后唤醒条件队列中的头节点。

doSignal(Node first)：唤醒头节点

```
private void doSignal(Node first) {
            do {
                // 修改头节点、方便移除
                if ( (firstWaiter = first.nextWaiter) == null)
                    lastWaiter = null;
                first.nextWaiter = null;
                // 将该节点移到同步队列
            } while (!transferForSignal(first) &&
                     (first = firstWaiter) != null);
        }

```

doSignal(Node first)主要是做两件事：

- 修改头节点；
- 调用transferForSignal(Node first) 方法将节点移动到CLH同步队列中。

transferForSignal(Node first)源码如下

```
final boolean transferForSignal(Node node) {
        /*
         * If cannot change waitStatus, the node has been cancelled.
         */
         // 将该节点的状态改为初始状态0
        if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
            return false;

        /*
         * Splice onto queue and try to set waitStatus of predecessor to
         * indicate that thread is (probably) waiting. If cancelled or
         * attempt to set waitStatus fails, wake up to resync (in which
```

```
         * case the waitStatus can be transiently and harmlessly wrong).
         */

        // 将该节点添加到同步队列的尾部、返回的是旧的尾部节点，也就是 node.prev节点
        Node p = enq(node);

        //如果结点p的状态为cancel 或者修改waitStatus失败，则直接唤醒
        int ws = p.waitStatus;
        if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
            LockSupport.unpark(node.thread);
        return true;
    }
```

整个通知的流程如下：

- 判断当前线程是否已经获取了锁，如果没有获取则直接抛出异常，因为获取锁为通知的前置条件。
- 如果线程已经获取了锁，则将唤醒条件队列的首节点。
- 唤醒首节点是先将条件队列中的头节点移出，然后调用AQS的enq(Node node)方法将其安全地移到CLH同步队列中 。
- 最后判断如果该节点的同步状态是否为Cancel，或者修改状态为Signal失败时，则直接调用LockSupport唤醒该节点的线程。

一个线程获取锁后，通过调用Condition的await()方法，会将当前线程先加入到条件队列中，然后释放锁，最后通过isOnSyncQueue(Node node)方法不断自检看节点是否已经在CLH同步队列了，如果是则尝试获取锁，否则一直挂起。

当线程调用signal()方法后，程序首先检查当前线程是否获取了锁，然后通过doSignal(Node first)方法唤醒CLH同步队列的首节点。被唤醒的线程，将从await()方法中的while循环中退出来，然后调用acquireQueued()方法竞争同步状态。

栗子：

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }

   public Object take() throws InterruptedException {
     lock.lock();
     try {
```

```
        while (count == 0)
          notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return x;
      } finally {
        lock.unlock();
      }
    }
  }
```

3.4 Synchronizers

3.4.1 Semaphore

Semaphore（信号量）是用来控制同时访问特定资源的线程数量，通过协调各个线程，保证合理的使用公共资源。

Semaphore维护了一个许可集，其实就是一定数量的“许可证”。
当有线程想要访问共享资源时，需要先获取(acquire)的许可；如果许可不够了，线程需要一直等待，直到许可可用。当线程使用完共享资源后，可以归还(release)许可，以供其它需要的线程使用。

和ReentrantLock类似，Semaphore支持公平/非公平策略。

源码:略。

使用

```
package kaikeba.com;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

//创建一个会实现print queue的类名为 PrintQueue。
class PrintQueue {

    // 声明一个对象为Semaphore，称它为semaphore。
    private final Semaphore semaphore;
    // 实现类的构造函数并初始能保护print quere的访问的semaphore对象的值。
    public PrintQueue() {
        semaphore = new Semaphore(1);
    }

    //实现Implement the printJob()方法，此方法可以模拟打印文档，并接收document对象作为参数。
    public void printJob(Object document) {
//在这方法内，首先，你必须调用acquire()方法获得demaphore。这个方法会抛出InterruptedException异常，使用必须包含处理这个异常的代码。
        try {
            semaphore.acquire();

//然后，实现能随机等待一段时间的模拟打印文档的行。
            long duration = (long) (Math.random() * 10);

```

```
            System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n", Thread.currentThread().getName(), duration);

            Thread.sleep(duration);

//最后，释放semaphore通过调用semaphore的relaser()方法。
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release();
        }
    }

}

//创建一个名为Job的类并一定实现Runnable 接口。这个类实现把文档传送到打印机的任务。
class Job implements Runnable {
    //声明一个对象为PrintQueue，名为printQueue。
    private PrintQueue printQueue;
    //实现类的构造函数，初始化这个类里的PrintQueue对象。
    public Job(PrintQueue printQueue) {
        this.printQueue = printQueue;
    }

    //实现方法run()。
    @Override
    public void run() {
                //首先， 此方法写信息到操控台表明任务已经开始执行了。
        System.out.printf("%s: Going to print a job\n", Thread.currentThread().getName());
                // 然后，调用PrintQueue 对象的printJob()方法。
        printQueue.printJob(new Object());
                //最后， 此方法写信息到操控台表明它已经结束运行了。
        System.out.printf("%s: The document has been printed\n", Thread.currentThread().getName());

    }
}

public class SemaphoreTest {

    public static void main(String args[]) {

                // 创建PrintQueue对象名为printQueue。
        PrintQueue printQueue = new PrintQueue();
            //创建10个threads。每个线程会执行一个发送文档到print queue的Job对象。
        Thread thread[] = new Thread[10];

        for (int i = 0; i < 10; i++) {
            thread[i] = new Thread(new Job(printQueue), "Thread" + i);
        }

        for (int i = 0; i < 10; i++) {
            thread[i].start();
        }

    }
}
```

3.4.2 CountDownLatch

在多线程协作完成业务功能时，有时候需要等待其他多个线程完成任务之后，主线程才能继续往下执行业务功能，在这种的业务场景下，通常可以使用Thread类的join方法，让主线程等待被join的线程执行完之后，主线程才能继续往下执行。当然，使用线程间消息通信机制也可以完成。其实，java并发工具类中为我们提供了类似“倒计时”这样的工具类，可以十分方便的完成所说的这种业务场景。

CountDownLatch允许一个或多个线程等待其他线程完成工作。

CountDownLatch相关方法：

- public CountDownLatch(int count) 构造方法会传入一个整型数N，之后调用CountDownLatch的`countDown`方法会对N减一，知道N减到0的时候，当前调用`await`方法的线程继续执行。
- await() throws InterruptedException：调用该方法的线程等到构造方法传入的N减到0的时候，才能继续往下执行；
- await(long timeout, TimeUnit unit)：与上面的await方法功能一致，只不过这里有了时间限制，调用该方法的线程等到指定的timeout时间后，不管N是否减至为0，都会继续往下执行；
- countDown()：使CountDownLatch初始值N减1；
- long getCount()：获取当前CountDownLatch维护的值

源码：略。

栗子：运动员进行跑步比赛时，假设有6个运动员参与比赛，裁判员在终点会为这6个运动员分别计时，可以想象没当一个运动员到达终点的时候，对于裁判员来说就少了一个计时任务。直到所有运动员都到达终点了，裁判员的任务也才完成。这6个运动员可以类比成6个线程，当线程调用CountDownLatch.countDown方法时就会对计数器的值减一，直到计数器的值为0的时候，裁判员（调用await方法的线程）才能继续往下执行。

```
public class CountDownLatchTest {
private static CountDownLatch startSignal = new CountDownLatch(1);
//用来表示裁判员需要维护的是6个运动员
private static CountDownLatch endSignal = new CountDownLatch(6);

public static void main(String[] args) throws InterruptedException {
    ExecutorService executorService = Executors.newFixedThreadPool(6);
    for (int i = 0; i < 6; i++) {
        executorService.execute(() -> {
            try {
                System.out.println(Thread.currentThread().getName() + " 运动员等待裁判员响哨！！！");
                startSignal.await();
                System.out.println(Thread.currentThread().getName() + "正在全力冲刺");
                endSignal.countDown();
                System.out.println(Thread.currentThread().getName() + "  到达终点");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
    System.out.println("裁判员响哨开始啦！！！");
    startSignal.countDown();
    endSignal.await();
    System.out.println("所有运动员到达终点，比赛结束！");
```

```
25      executorService.shutdown();
26  }
27  }
```

该示例代码中设置了两个CountDownLatch，第一个endSignal用于控制让main线程（裁判员）必须等到其他线程（运动员）让CountDownLatch维护的数值N减到0为止，相当于一个完成信号；另一个startSignal用于让main线程对其他线程进行“发号施令”，相当于一个入口或者开关。

startSignal引用的CountDownLatch初始值为1，而其他线程执行的run方法中都会先通过startSignal.await()让这些线程都被阻塞，直到main线程通过调用startSignal.countDown();，将值N减1，CountDownLatch维护的数值N为0后，其他线程才能往下执行，并且，每个线程执行的run方法中都会通过endSignal.countDown();对endSignal维护的数值进行减一，由于往线程池提交了6个任务，会被减6次，所以endSignal维护的值最终会变为0，因此main线程在latch.await();阻塞结束，才能继续往下执行。

注意：当调用CountDownLatch的countDown方法时，当前线程是不会被阻塞，会继续往下执行。

3.4.3 CyclicBarrier

CountDownLatch是一个倒数计数器，在计数器不为0时，所有调用await的线程都会等待，当计数器降为0，线程才会继续执行，且计数器一旦变为0，就不能再重置了。

CyclicBarrier可以认为是一个栅栏，栅栏的作用是什么？就是阻挡前行。

CyclicBarrier是一个可以循环使用的栅栏，它做的事情就是：让线程到达栅栏时被阻塞(调用await方法)，直到到达栅栏的线程数满足指定数量要求时，栅栏才会打开放行，被栅栏拦截的线程才可以执行。

当多个线程都达到了指定点后，才能继续往下继续执行。这就有点像报数的感觉，假设6个线程就相当于6个运动员，到赛道起点时会报数进行统计，如果刚好是6的话，这一波就凑齐了，才能往下执行。这里的6个线程，也就是计数器的初始值6，是通过CyclicBarrier的构造方法传入的。

CyclicBarrier的主要方法：

- await() throws InterruptedException, BrokenBarrierException 等到所有的线程都到达指定的临界点；
- await(long timeout, TimeUnit unit) throws InterruptedException, BrokenBarrierException, TimeoutException 与上面的await方法功能基本一致，只不过这里有超时限制，阻塞等待直至到达超时时间为止；
- int getNumberWaiting()获取当前有多少个线程阻塞等待在临界点上；
- boolean isBroken()用于查询阻塞等待的线程是否被中断
- void reset()将屏障重置为初始状态。如果当前有线程正在临界点等待的话，将抛出BrokenBarrierException。

另外需要注意的是，CyclicBarrier提供了这样的构造方法：

```
1  public CyclicBarrier(int parties, Runnable barrierAction)
```

可以用来，当指定的线程都到达了指定的临界点的时，接下来执行的操作可以由barrierAction传入即可。

源码：略。

栗子：
6个运动员准备跑步比赛，运动员在赛跑需要在起点做好准备，当裁判发现所有运动员准备完毕后，就举起发令枪，比赛开始。这里的起跑线就是屏障，是临界点，而这6个运动员就类比成线程的话，就是这6个线程都必须到达指定点了，意味着凑齐了一波，然后才能继续执行，否则每个线程都得阻塞等

待，直至凑齐一波即可。

```
public class CyclicBarrierTest {
    public static void main(String[] args) {

        int N = 6;  // 运动员数
        CyclicBarrier cb = new CyclicBarrier(N, new Runnable() {
            @Override
            public void run() {
                System.out.println("所有运动员已准备完毕，发令枪：跑！");
            }
        });

        for (int i = 0; i < N; i++) {
            Thread t = new Thread(new PrepareWork(cb), "运动员[" + i + "]");
            t.start();
        }
    }


    private static class PrepareWork implements Runnable {
        private CyclicBarrier cb;

        PrepareWork(CyclicBarrier cb) {
            this.cb = cb;
        }

        @Override
        public void run() {

            try {
                Thread.sleep(500);
                System.out.println(Thread.currentThread().getName() + ": 准备完成");
                cb.await();          // 在栅栏等待
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
}
```

从输出结果可以看出，当6个运动员（线程）都到达了指定的临界点（barrier）时候，才能继续往下执行，否则，则会阻塞等待在调用`await()`处。

CyclicBarrier对异常的处理

线程在阻塞过程中，可能被中断，那么既然CyclicBarrier放行的条件是等待的线程数达到指定数目，万一线程被中断导致最终的等待线程数达不到栅栏的要求怎么办？

```
public int await() throws InterruptedException, BrokenBarrierException {
    //...
}
```

可以看到，这个方法除了抛出InterruptedException异常外，还会抛出`BrokenBarrierException`。

BrokenBarrierException表示当前的CyclicBarrier已经损坏了，等不到所有线程都到达栅栏了，所以已经在等待的线程也没必要再等了，可以散伙了。

出现以下几种情况之一时，当前等待线程会抛出BrokenBarrierException异常：

- 其它某个正在await等待的线程被中断了；
- 其它某个正在await等待的线程超时了；
- 某个线程重置了CyclicBarrier；

另外，只要正在Barrier上等待的任一线程抛出了异常，那么Barrier就会认为肯定是凑不齐所有线程了，就会将栅栏置为损坏（Broken）状态，并传播BrokenBarrierException给其它所有正在等待（await）的线程。

异常情况模拟：

```
public class CyclicBarrierTest {
    public static void main(String[] args) throws InterruptedException {

        int N = 6;  // 运动员数
        CyclicBarrier cb = new CyclicBarrier(N, new Runnable() {
            @Override
            public void run() {
                System.out.println("所有运动员已准备完毕，发令枪：跑！");
            }
        });

        List<Thread> list = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            Thread t = new Thread(new PrepareWork(cb), "运动员[" + i + "]");
            list.add(t);
            t.start();
            if (i == 3) {
                t.interrupt();  // 运动员[3]置中断标志位
            }
        }

        Thread.sleep(2000);
        System.out.println("Barrier是否损坏：" + cb.isBroken());
    }


    private static class PrepareWork implements Runnable {
        private CyclicBarrier cb;

        PrepareWork(CyclicBarrier cb) {
            this.cb = cb;
        }

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + ": 准备完成");
                cb.await();
            } catch (InterruptedException e) {
```

```
                System.out.println(Thread.currentThread().getName() + ": 被中断");
            } catch (BrokenBarrierException e) {
                System.out.println(Thread.currentThread().getName() + ": 抛出BrokenBarrierException");
            }
        }
    }
}
```

CountDownLatch与CyclicBarrier的比较

CountDownLatch与CyclicBarrier都是用于控制并发的工具类，都可以理解成维护的就是一个计数器，但是这两者还是各有不同侧重点的：

- CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；CountDownLatch强调一个线程等多个线程完成某件事情。CyclicBarrier是多个线程互等，等大家都完成，再携手共进。
- 调用CountDownLatch的countDown方法后，当前线程并不会阻塞，会继续往下执行；而调用CyclicBarrier的await方法，会阻塞当前线程，直到CyclicBarrier指定的线程全部都到达了指定点的时候，才能继续往下执行；
- CountDownLatch方法比较少，操作比较简单，而CyclicBarrier提供的方法更多，比如能够通过getNumberWaiting()，isBroken()这些方法获取当前多个线程的状态，并且CyclicBarrier的构造方法可以传入barrierAction，指定当所有线程都到达时执行的业务功能；
- CountDownLatch是不能复用的，而CyclicLatch是可以复用的。

3.4.4 Exchanger

Exchanger可以用来在两个线程之间交换持有的对象。当Exchanger在一个线程中调用exchange方法之后，会等待另外的线程调用同样的exchange方法，两个线程都调用exchange方法之后，传入的参数就会交换。

两个主要方法

```
public V exchange(V x) throws InterruptedException
```

当这个方法被调用的时候，当前线程将会等待直到其他的线程调用同样的方法。当其他的线程调用exchange之后，当前线程将会继续执行。

在等待过程中，如果有其他的线程interrupt当前线程，则会抛出InterruptedException。

```
public V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException, TimeoutException
```

多了一个timeout时间。如果在timeout时间之内没有其他线程调用exchange方法，抛出TimeoutException。

源码：略。

栗子：

我们先定义一个带交换的类：

然后定义两个Runnable，在run方法中调用exchange方法：

```
public class ExchangerTest {

```

```
    public static void main(String[] args) {
        Exchanger<CustBook> exchanger = new Exchanger<>();
        // Starting two threads
        new Thread(new ExchangerOne(exchanger)).start();
        new Thread(new ExchangerTwo(exchanger)).start();
    }
}
public class CustBook {

    private String name;
}
public class ExchangerOne implements Runnable{

    Exchanger<CustBook> ex;

    ExchangerOne(Exchanger<CustBook> ex){
      this.ex=ex;
    }

    @Override
    public void run() {
    CustBook custBook= new CustBook();
        custBook.setName("book one");

        try {
            CustBook exhangeCustBook=ex.exchange(custBook);
            log.info(exhangeCustBook.getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class ExchangerTwo implements Runnable{

    Exchanger<CustBook> ex;

    ExchangerTwo(Exchanger<CustBook> ex){
      this.ex=ex;
    }

    @Override
    public void run() {
    CustBook custBook= new CustBook();
        custBook.setName("book two");

        try {
            CustBook exhangeCustBook=ex.exchange(custBook);
            log.info(exhangeCustBook.getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3.4.5 Phaser

Phaser是一个同步工具类，适用于一些需要分阶段的任务的处理。它的功能与 CyclicBarrier和CountDownLatch类似，类似于一个多阶段的栅栏，并且功能更强大，我们来比较下这三者的功能：

CountDownLatch	倒数计数器，初始时设定计数器值，线程可以在计数器上等待，当计数器值归0后，所有等待的线程继续执行
CyclicBarrier	循环栅栏，初始时设定参与线程数，当线程到达栅栏后，会等待其它线程的到达，当到达栅栏的总数满足指定数后，所有等待的线程继续执行
Phaser	多阶段栅栏，可以在初始时设定参与线程数，也可以中途注册/注销参与者，当到达的参与者数量满足栅栏设定的数量后，会进行阶段升级（advance）

相关概念：

phase(阶段)

Phaser也有栅栏，在Phaser中，栅栏的名称叫做phase(阶段)，在任意时间点，Phaser只处于某一个phase(阶段)，初始阶段为0，最大达到Integerr.MAX_VALUE，然后再次归零。当所有parties参与者都到达后，phase值会递增。

parties(参与者)

Phaser既可以在初始构造时指定参与者的数量，也可以中途通过register、bulkRegister、arriveAndDeregister等方法注册/注销参与者。

arrive(到达) / advance(进阶)

Phaser注册完parties（参与者）之后，参与者的初始状态是unarrived的，当参与者到达（arrive）当前阶段（phase）后，状态就会变成arrived。当阶段的到达参与者数满足条件后（注册的数量等于到达的数量），阶段就会发生进阶（advance）——也就是phase值+1。

Termination（终止）

代表当前Phaser对象达到终止状态。

Tiering（分层）

Phaser支持分层（Tiering） —— 一种树形结构，通过构造函数可以指定当前待构造的Phaser对象的父结点。之所以引入Tiering，是因为当一个Phaser有大量参与者（parties）的时候，内部的同步操作会使性能急剧下降，而分层可以降低竞争，从而减小因同步导致的额外开销。

在一个分层Phasers的树结构中，注册和撤销子Phaser或父Phaser是自动被管理的。当一个Phaser参与者（parties）数量变成0时，如果有该Phaser有父结点，就会将它从父结点中溢移除。

核心方法：

- arriveAndDeregister() 该方法立即返回下一阶段的序号，并且其它线程需要等待的个数减一，
  取消自己的注册、把当前线程从之后需要等待的成员中移除。
  如果该Phaser是另外一个Phaser的子Phaser（层次化Phaser），
  并且该操作导致当前Phaser的成员数为0，则该操作也会将当前Phaser从其父Phaser中移除。
- arrive() 某个参与者完成任务后调用，该方法不作任何等待，直接返回下一阶段的序号。
- awaitAdvance(int phase) 该方法等待某一阶段执行完毕。
  如果当前阶段不等于指定的阶段或者该Phaser已经被终止，则立即返回。
  该阶段数一般由arrive()方法或者arriveAndDeregister()方法返回。

返回下一阶段的序号，或者返回参数指定的值（如果该参数为负数），或者直接返回当前阶段序号（如果当前Phaser已经被终止）。

- awaitAdvanceInterruptibly(int phase) 效果与awaitAdvance(int phase)相当，
  唯一的不同在于若该线程在该方法等待时被中断，则该方法抛出InterruptedException。
- awaitAdvanceInterruptibly(int phase, long timeout, TimeUnit unit)
  效果与awaitAdvanceInterruptibly(int phase)相当，
  区别在于如果超时则抛出TimeoutException。
- bulkRegister(int parties) 动态调整注册任务parties的数量。如果当前phaser已经被终止，则该方法无效，并返回负数。
  如果调用该方法时，onAdvance方法正在执行，则该方法等待其执行完毕。
  如果该Phaser有父Phaser则指定的party数大于0，且之前该Phaser的party数为0，那么该Phaser会被注册到其父Phaser中。
- forceTermination() 强制让该Phaser进入终止状态。
  已经注册的party数不受影响。如果该Phaser有子Phaser，则其所有的子Phaser均进入终止状态。
  如果该Phaser已经处于终止状态，该方法调用不造成任何影响。

代码示例

栗子：3个线程，4个阶段，每个阶段都并发处理

```
package kaikeba.com;

import java.util.concurrent.Phaser;

public class PhaserTest {
    public static void main(String[] args) {
        int parties = 3;
        int phases = 4;
        final Phaser phaser = new Phaser(parties) {
            @Override
            //每个阶段结束时
            protected boolean onAdvance(int phase, int registeredParties) {
                System.out.println("====== Phase : " + phase + "  end ======");
                return registeredParties == 0;
            }
        };
        for (int i = 0; i < parties; i++) {
            int threadId = i;
            Thread thread = new Thread(() -> {
                for (int phase = 0; phase < phases; phase++) {
                    if (phase == 0) {
                        System.out.println(String.format("第一阶段操作 Thread %s, phase %s", threadId, phase));
                    }
                    if (phase == 1) {
                        System.out.println(String.format("第二阶段操作 Thread %s, phase %s", threadId, phase));
                    }
                    if (phase == 2) {
                        System.out.println(String.format("第三阶段操作 Thread %s, phase %s", threadId, phase));
                    }
                    if (phase == 3) {
                        System.out.println(String.format("第四阶段操作 Thread %s, phase %s", threadId, phase));
```

```
32                      }
33              /**
34               * arriveAndAwaitAdvance() 当前线程当前阶段执行完毕，等待其它线程完成当前阶段。
35               * 如果当前线程是该阶段最后一个未到达的，则该方法直接返回下一个阶段的序号（阶段序号从0开始），
36               * 同时其它线程的该方法也返回下一个阶段的序号。
37               **/
38                      int nextPhaser = phaser.arriveAndAwaitAdvance();
39
40                  }
41              });
42              thread.start();
43          }
44      }
45  }
```

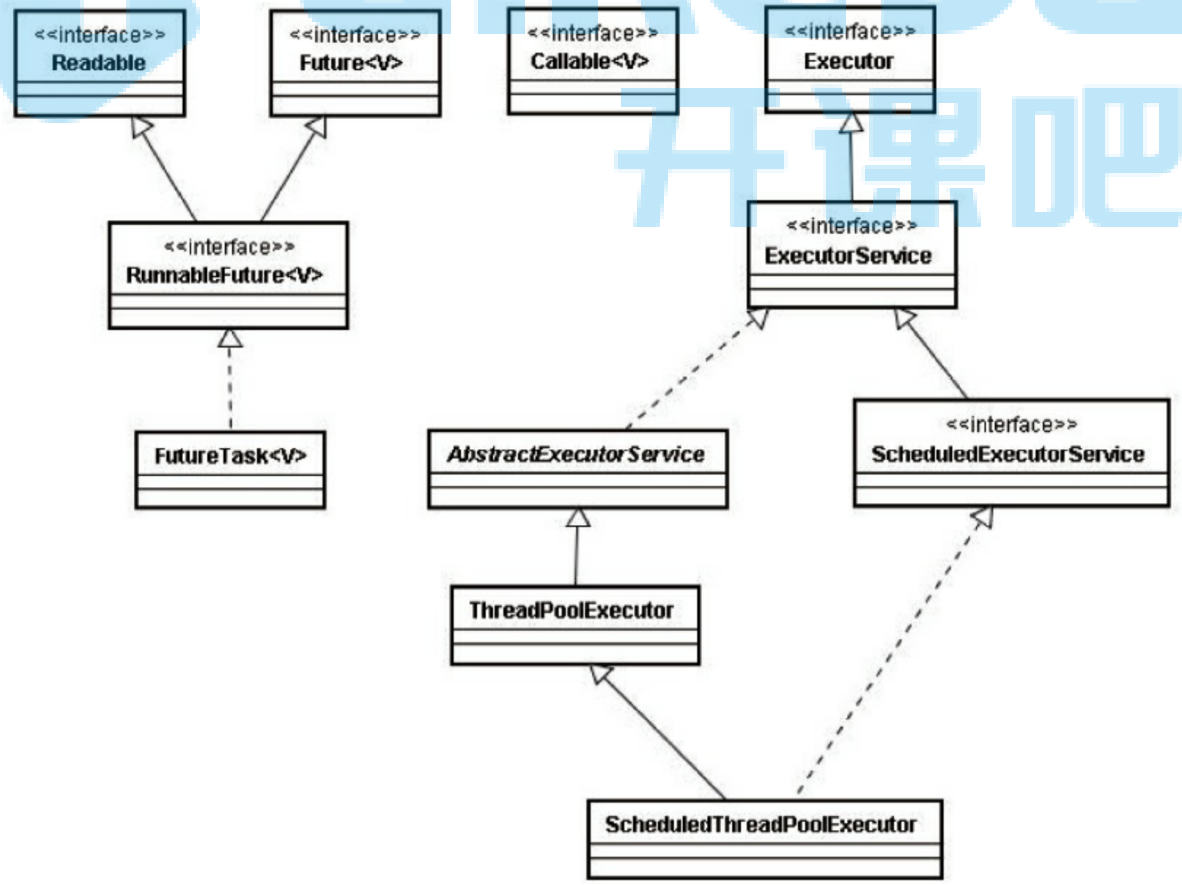
3.5 Executors

3.5.1 Executor框架

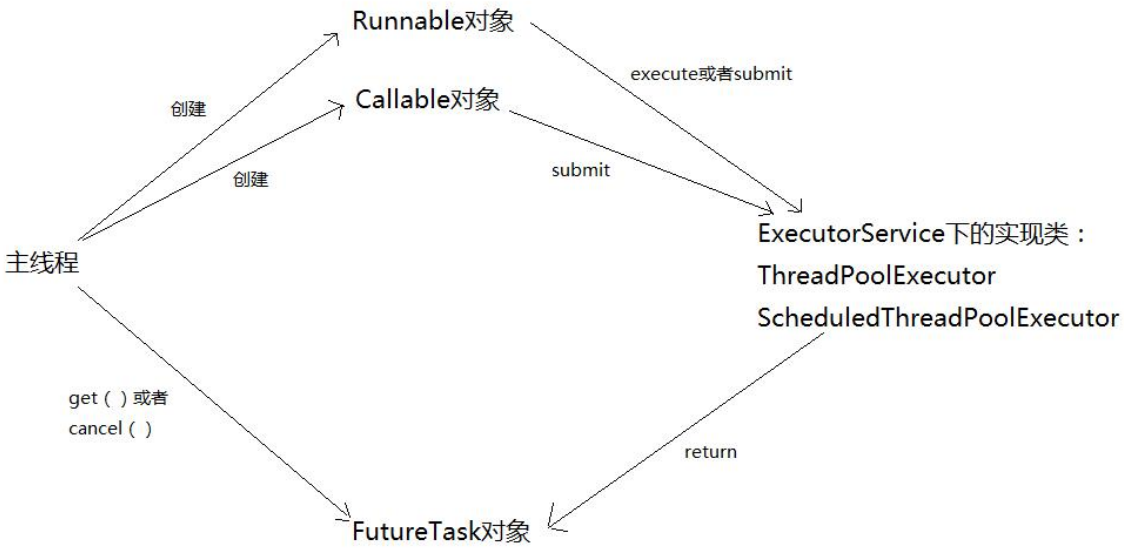
Executor框架包括3大部分：

- 任务。也就是工作单元，包括被执行任务需要实现的接口：Runnable接口或者Callable接口；
- 任务的执行。也就是把任务分派给多个线程的执行机制，包括Executor接口及继承自Executor接口的ExecutorService接口。
- 异步计算的结果。包括Future接口及实现了Future接口的FutureTask类。

Executor框架的成员及其关系可以用一下的关系图表示：



Executor框架的使用示意图：



使用步骤：

- 创建Runnable并重写run ( ) 方法或者Callable对象并重写call ( ) 方法，得到一个任务对象

```
class callableImp implements Callable<String >{
        @Override
        public String call() {
            try{
                String a = "return String";
                return a;
            }
            catch(Exception e){
                e.printStackTrace();
                return "exception";
            }
        }
    }
```

- 创建ExecutorService接口的实现类ThreadPoolExecutor类或者ScheduledThreadPoolExecutor类的对象，然后调用其execute()方法或者submit()方法，提交任务对象执行。
- 主线程调用Future对象的get()方法获取返回值，或者调用Future对象的cancel()方法取消当前线程的执行。

Executor框架成员：ThreadPoolExecutor实现类、ScheduledThreadPoolExecutor实现类、Future接口、Runnable和Callable接口、Executors工厂类

Executor:执行器接口，也是最顶层的抽象核心接口， 分离了任务和任务的执行。ExecutorService在Executor的基础上提供了执行器生命周期管理，任务异步执行等功能。

Executors：生产具体的执行器的静态工厂。

ThreadPoolExecutor:线程池Executor，也是最常用的Executor，通常使用Executors来创建，可以创建三种类型的ThreadPoolExecutor：SingleThreadPoolExecutor，FixedThreadPool和CachedThreadPool，以线程池的方式管理线程。

ScheduledThreadPoolExecutor：在ThreadPoolExecutor基础上，增加了对周期任务调度的支持。

Runnable和Callable接口：Runnable和Callable接口的实现类，可以被ThreadPoolExecutor和ScheduledThreadPoolExecutor执行，区别是，亲着没有返回结果，候着可以返回结果。

3.5.2 ThreadPoolExecutor

ThreadPoolExecutor一共提供了4种构造器，但其它三种内部其实都调用了下面的构造器。

```
/**
 * 使用给定的参数创建ThreadPoolExecutor.
 *
 * @param corePoolSize     核心线程池中的最大线程数
 * @param maximumPoolSize 总线程池中的最大线程数
 * @param keepAliveTime    空闲线程的存活时间
 * @param unit             keepAliveTime的单位
 * @param workQueue        任务队列，保存已经提交但尚未被执行的线程
 * @param threadFactory    线程创建工厂
 * @param handler          拒绝策略 (当任务太多导致工作队列满时的处理策略)
 */
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

线程池状态定义；

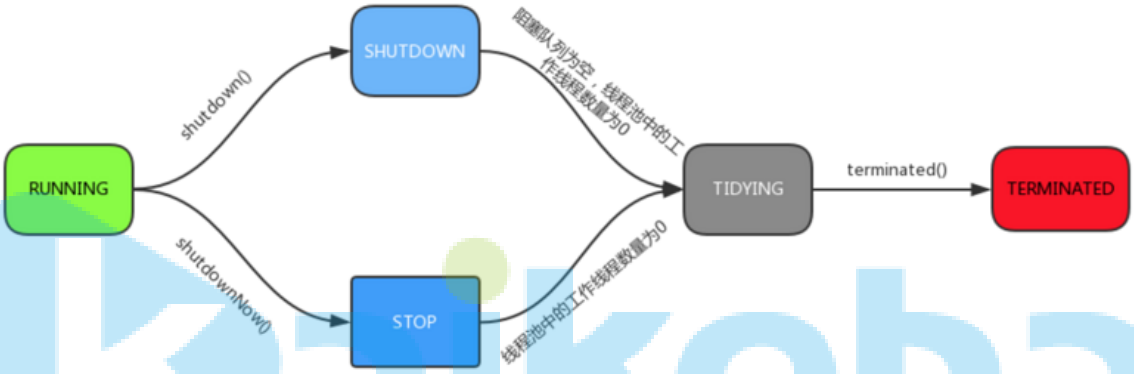
```
 /*
     * RUNNING -> SHUTDOWN
     *    On invocation of shutdown(), perhaps implicitly in finalize()
     * (RUNNING or SHUTDOWN) -> STOP
     *    On invocation of shutdownNow()
     * SHUTDOWN -> TIDYING
     *    When both queue and pool are empty
     * STOP -> TIDYING
     *    When pool is empty
     * TIDYING -> TERMINATED
     *    When the terminated() hook method has completed
     *
*/
    private static final int RUNNING    = -1 << COUNT_BITS;
    private static final int SHUTDOWN   =  0 << COUNT_BITS;
```

```
    private static final int STOP       =  1 << COUNT_BITS;
    private static final int TIDYING    =  2 << COUNT_BITS;
    private static final int TERMINATED =  3 << COUNT_BITS;
```

ThreadPoolExecutor一共定义了5种线程池状态：

- RUNNING ： 接受新任务，且处理已经进入阻塞队列的任务
- SHUTDOWN ： 不接受新任务，但处理已经进入阻塞队列的任务
- STOP ： 不接受新任务，且不处理已经进入阻塞队列的任务，同时中断正在运行的任务
- TIDYING ： 所有任务都已终止，工作线程数为0，线程转化为TIDYING状态并准备调用terminated方法
- TERMINATED ： terminated方法已经执行完成

各个状态之间的流转图：



执行execute

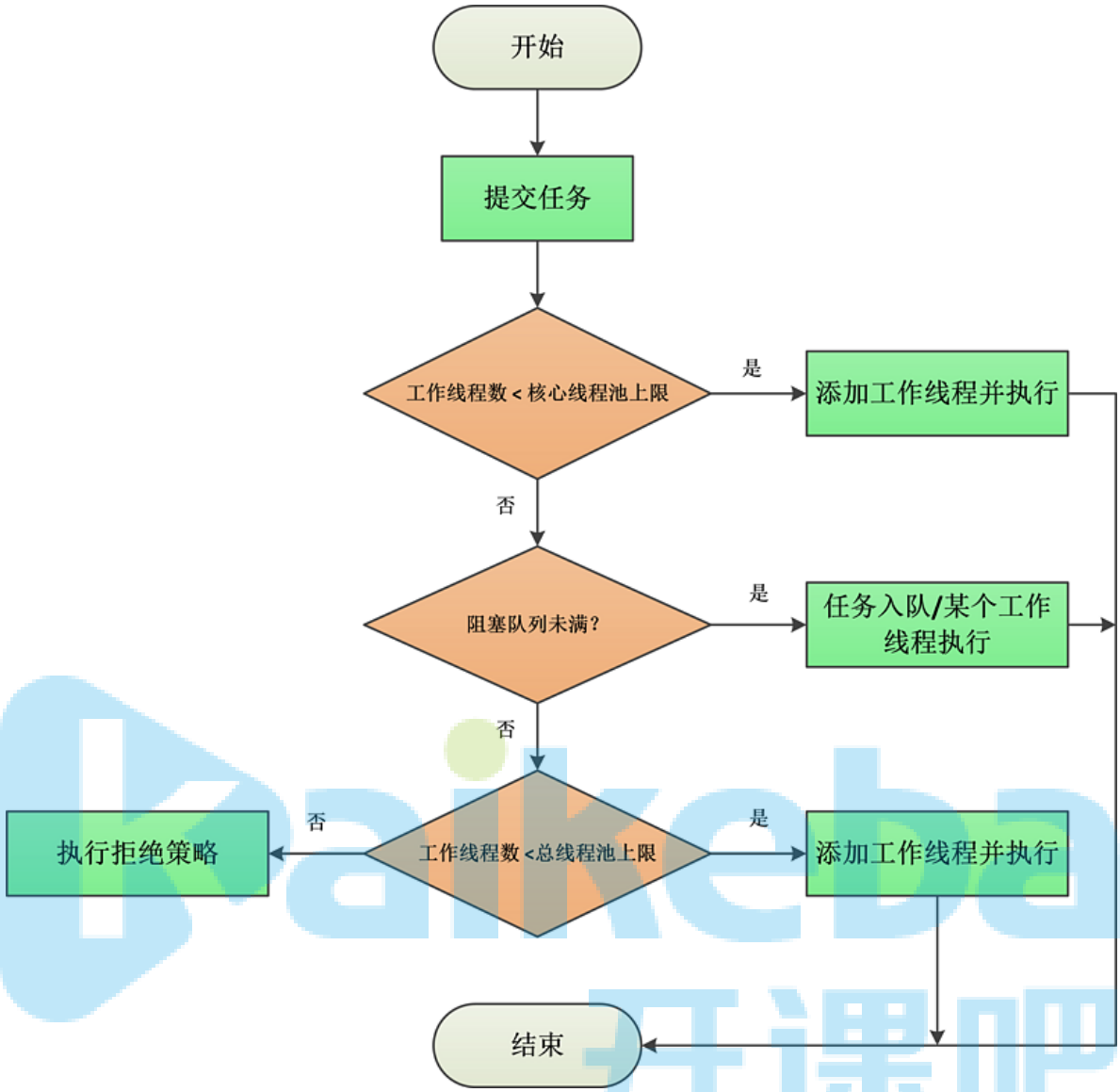
```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {        // CASE1: 工作线程数 < 核心线程池上限
        if (addWorker(command, true))             // 添加工作线程并执行
            return;
        c = ctl.get();
    }

    // 执行到此处，说明工作线程创建失败 或 工作线程数≥核心线程池上限
    if (isRunning(c) && workQueue.offer(command)) {     // CASE2: 插入任务至队列

        // 再次检查线程池状态
        int recheck = ctl.get();
        if (!isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    } else if (!addWorker(command, false))        // CASE3: 插入队列失败，判断工作线程数 < 总线程池上限
        reject(command);    // 执行拒绝策略
}
```

上述execute的执行流程可以用下图描述：



execute的整个执行流程关键是下面两点：

- 如果工作线程数小于核心线程池上限（CorePoolSize），则直接新建一个工作线程并执行任务；
- 如果工作线程数大于等于CorePoolSize，则尝试将任务加入到队列等待以后执行。如果加入队列失败了（比如队列已满的情况），则在总线程池未满的情况下（`CorePoolSize ≤ 工作线程数 < maximumPoolSize`）新建一个工作线程立即执行任务，否则执行拒绝策略。

通过Executor框架的工具类Executors，可以创建三种类型的ThreadPoolExecutor：

FixedThreadPool：可重用固定线程数的线程池：

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

newFixedThreadPool创建一个固定长度的线程池，每次提交一个任务的时候就会创建一个新的线程，直到达到线程池的最大数量限制。

- 定长，可以控制线程最大并发数， corePoolSize 和 maximumPoolSize 的数值都是nThreads。

- 超出线程数的任务会在队列中等待。
- 工作队列为LinkedBlockingQueue。

创建方法

```
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(int nThreads);
```

SingleThreadExecutor:使用单个线程的Executor

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

newSingleThreadExecutor，只创建一个工作线程执行任务，若这个唯一的线程异常故障了，会新建另一个线程来替代，newSingleThreadExecutor可以保证任务依照在工作队列的排队顺序来串行执行。

- 有且仅有一个工作线程执行任务；
- 所有任务按照工作队列的排队顺序执行，先进先出的顺序。
- 工作队列LinkedBlockingQueue。

创建方法

```
ExecutorService singleThreadPool = Executors.newSingleThreadPool();
```

CachedThreadPool:会根据需要创建新线程的线程池

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

newCachedThreadPool将创建一个可缓存的线程池，如果当前线程数超过处理任务时，回收空闲线程；当需求增加时，可以添加新线程去处理任务。

特点：

- 线程数无限制，corePoolSize数值为0， maximumPoolSize 的数值都是为Integer.MAX_VALUE。
- 若线程未回收，任务到达时，会复用空闲线程；若无空闲线程，则新建线程执行任务。
- 因为复用性，一定程序减少频繁创建/销毁线程，减少系统开销。
- 工作队列选用SynchronousQueue。

创建方法

```
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
```

栗子：

```
package kaikeba.com;

import java.io.IOException;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

public class ThreadPoolExecutorTest {

    public static void main(String[] args) throws InterruptedException, IOException {
        int corePoolSize = 2;
        int maximumPoolSize = 4;
        long keepAliveTime = 10;
        TimeUnit unit = TimeUnit.SECONDS;
        BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(2);
        RejectedExecutionHandler handler = new RejectedExecutionPolicy();
        ThreadPoolExecutor executor = new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, unit,
                workQueue, handler);
        executor.prestartAllCoreThreads(); // 预启动所有核心线程

        for (int i = 1; i <= 10; i++) {
            ThreadTask task = new ThreadTask(String.valueOf(i));
            executor.execute(task);
        }

        System.in.read(); //阻塞主线程
    }


    public static class RejectedExecutionPolicy implements RejectedExecutionHandler {

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

        private void doLog(Runnable r, ThreadPoolExecutor e) {
            System.err.println( r.toString() + " rejected");
        }
    }

    static class ThreadTask implements Runnable {
        private String name;

        public ThreadTask(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            try {
```

```
                System.out.println(this.toString() + " is running!");
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        public String getName() {
            return name;
        }

        @Override
        public String toString() {
            return "ThreadTask [name=" + name + "]";
        }
    }
}
```

3.5.3 ScheduledExecutorService

构造线程池

Executors使用`newScheduledThreadPool`工厂方法创建ScheduledThreadPoolExecutor：

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory) {
    return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
}
```

ScheduledThreadPoolExecutor的构造器，内部其实都是调用了父类ThreadPoolExecutor的构造器，这里比较特别的是任务队列的选择——DelayedWorkQueue。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue());
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), threadFactory);
}

public ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), handler);
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory, RejectedExecutionHandler handler) {
```

```
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new
DelayedWorkQueue(), threadFactory, handler);
}
```

线程池的调度

该线程池的核心调度方法，是schedule、scheduleAtFixedRate、scheduleWithFixedDelay，通过schedule方法来看下整个调度流程：

```
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit
unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    RunnableScheduledFuture<?> t = decorateTask(command, new
ScheduledFutureTask<Void>(command, null,
            triggerTime(delay, unit)));
    delayedExecute(t);
    return t;
}
```

上述的decorateTask方法把Runnable任务包装成ScheduledFutureTask，用户可以根据自己的需要覆写该方法：

```
protected <V> RunnableScheduledFuture<V> decorateTask(Runnable runnable,
RunnableScheduledFuture<V> task) {
    return task;
}
```

ScheduledFutureTask是RunnableScheduledFuture接口的实现类，任务通过period字段来表示任务类型

```
private class ScheduledFutureTask<V> extends FutureTask<V> implements
RunnableScheduledFuture<V> {

    //任务序号，自增唯一
     private final long sequenceNumber;

     // 首次执行的时间点
     private long time;

     // 0: 非周期任务; >0: fixed-rate任务;<0: fixed-delay任务
     private final long period;

     //在堆中的索引
     int heapIndex;

     ScheduledFutureTask(Runnable r, V result, long ns) {
         super(r, result);
         this.time = ns;
         this.period = 0;
         this.sequenceNumber = sequencer.getAndIncrement();
     }

     // ...
}
```

ScheduledThreadPoolExecutor中的任务队列——DelayedWorkQueue，保存的元素就是ScheduledFutureTask。DelayedWorkQueue是一种**堆结构**，time最小的任务会排在堆顶（表示最早过期），每次出队都是取堆顶元素，这样最快到期的任务就会被先执行。如果两个ScheduledFutureTask的time相同，就比较它们的序号——sequenceNumber，序号小的代表先被提交，所以就会先执行。

---

schedule的核心是其中的delayedExecute方法：

```
private void delayedExecute(RunnableScheduledFuture<?> task) {
    if (isShutdown())   // 线程池已关闭
        reject(task);   // 任务拒绝策略
    else {
        super.getQueue().add(task);                 // 将任务入队

        // 如果线程池已关闭且该任务是非周期任务，则将其从队列移除
        if (isShutdown() && !canRunInCurrentRunState(task.isPeriodic()) &&
remove(task))
            task.cancel(false);  // 取消任务
        else
            ensurePrestart();   // 添加一个工作线程
    }
}
```

处理过程：

- 任务被提交到线程池后，会判断线程池的状态，如果不是RUNNING状态会执行拒绝策略；
- 然后，将任务添加到阻塞队列中，由于DelayedWorkQueue是无界队列，所以一定会add成功；
- 然后，会创建一个工作线程，加入到核心线程池或者非核心线程池；

```
void ensurePrestart() {
    int wc = workerCountOf(ctl.get());
    if (wc < corePoolSize)//如果核心线程池未满，则新建的工作线程会被放到核心线程
池中。
        addWorker(null, true);
    else if (wc == 0)   //当通过setCorePoolSize方法设置核心线程池大小为0时，这
里必须要保证任务能够被执行，会创建一个工作线程，放到非核心线程池中。
        addWorker(null, false);
  //如果核心线程池已经满了,不会再去创建工作线程,直接返回。
}
```

- 最后，线程池中的工作线程会去任务队列获取任务并执行，当任务被执行完成后，如果该任务是周期任务，则会重置time字段，并重新插入队列中，等待下次执行。
- 从队列中获取元素的方法：

  对于核心线程池中的工作线程来说，如果没有超时设置（`allowCoreThreadTimeOut == false`），则会使用阻塞方法take获取任务（因为没有超时限制，所以会一直等待直到队列中有任务）；如果设置了超时，则会使用poll方法（方法入参需要超时时间），超时还没拿到任务的话，该工作线程就会被回收。

  对于非工作线程来说，都是调用poll获取队列元素，超时取不到任务就会被回收。

栗子：

```
package kaikeba.com;
```

```
2
3  import java.util.concurrent.*;
4
5  public class ScheduledThreadPoolExecutorTest {
6      public static void main(String[] args) throws ExecutionException,
   InterruptedException {
7          ScheduledThreadPoolExecutorTest.scheduleWithFixedDelay();
8          ScheduledThreadPoolExecutorTest.scheduleAtFixedRate();
9          ScheduledThreadPoolExecutorTest.scheduleCaller();
10         ScheduledThreadPoolExecutorTest.scheduleRunable();
11     }
12
13     // 任务以固定时间间隔执行，延迟5s后开始执行任务，任务执行完毕后间隔5s再次执行，依次往
   复
14     static void scheduleWithFixedDelay() throws InterruptedException,
   ExecutionException {
15         ScheduledExecutorService executorService = new
   ScheduledThreadPoolExecutor(10);
16
17         ScheduledFuture<?> result =
   executorService.scheduleWithFixedDelay(new Runnable() {
18             public void run() {
19
20                 System.out.println(System.currentTimeMillis());
21
22             }
23         }, 5000, 5000, TimeUnit.MILLISECONDS);
24
25         // 由于是定时任务，一直不会返回
26         result.get();
27         System.out.println("over");
28
29     }
30     // 相对开始加入任务的时间点固定频率执行：从加入任务开始算2s后开始执行任务，2+5s开始
   执行，2+2*5s执行，2+n*5s开始执行；
31     // 但是如果执行任务时间大于5s,则不会并发执行，后续任务将会延迟。
32
33     static void scheduleAtFixedRate() throws InterruptedException,
   ExecutionException {
34         ScheduledExecutorService executorService = new
   ScheduledThreadPoolExecutor(10);
35
36         ScheduledFuture<?> result = executorService.scheduleAtFixedRate(new
   Runnable() {
37             public void run() {
38
39                 System.out.println(System.currentTimeMillis());
40
41             }
42         }, 2000, 5000, TimeUnit.MILLISECONDS);
43
44         // 由于是定时任务，一直不会返回
45         result.get();
46         System.out.println("over");
47     }
48
49     // 延迟2s后开始执行，只执行一次，没有返回值
```

```
    static void scheduleRunable() throws InterruptedException, ExecutionException {
        ScheduledExecutorService executorService = new ScheduledThreadPoolExecutor(10);

        ScheduledFuture<?> result = executorService.schedule(new Runnable() {

            @Override
            public void run() {
                System.out.println("gh");
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

            }
        }, 2000, TimeUnit.MILLISECONDS);

        System.out.println(result.get());

    }

    // 延迟2s后开始执行，只执行一次，有返回值
    static void scheduleCaller() throws InterruptedException, ExecutionException {
        ScheduledExecutorService executorService = new ScheduledThreadPoolExecutor(10);

        ScheduledFuture<String> result = executorService.schedule(new Callable<String>() {

            @Override
            public String call() throws Exception {

                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                return "gh";
            }

        }, 2000, TimeUnit.MILLISECONDS);

        // 阻塞，直到任务执行完成
        System.out.print(result.get());

    }
}
```

kaikeba
开课吧