Java并发编程分享

同步和异步

同步和异步通常用来形容一次方法调用。同步方法调用一旦开始，调用者必须等到方法调用返回后才能进行后续的行为。大部分的接口方法都是同步方法。

异步方法调用立即返回，不必等待方法执行完。在nap-api中充斥着异步思想，比如使用rabbitmq进行防火墙设备的配置同步，我们去调用接口，手动同步配置，在方法中的核心逻辑就是发送消息给rabbitmq队列通知同步配置，接口就会直接返回，当队列的监听器监听到时才会进行同步操作。

在springboot中也可将同步方法变成异步方法：`@EnableAsync`,`@Async`

将`@EnableAsync`标注在主类上，将`@Async`标注在方法上

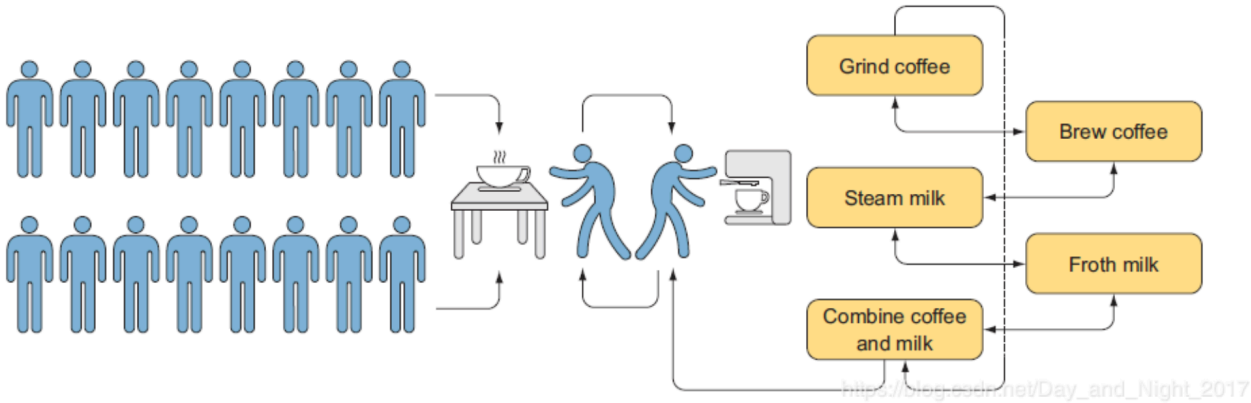
被注解的方法被调用的时候，会在新线程中执行，而调用它的方法会在原来的线程中执行。这样可以避免阻塞、以及保证任务的实时性。

并发和并行

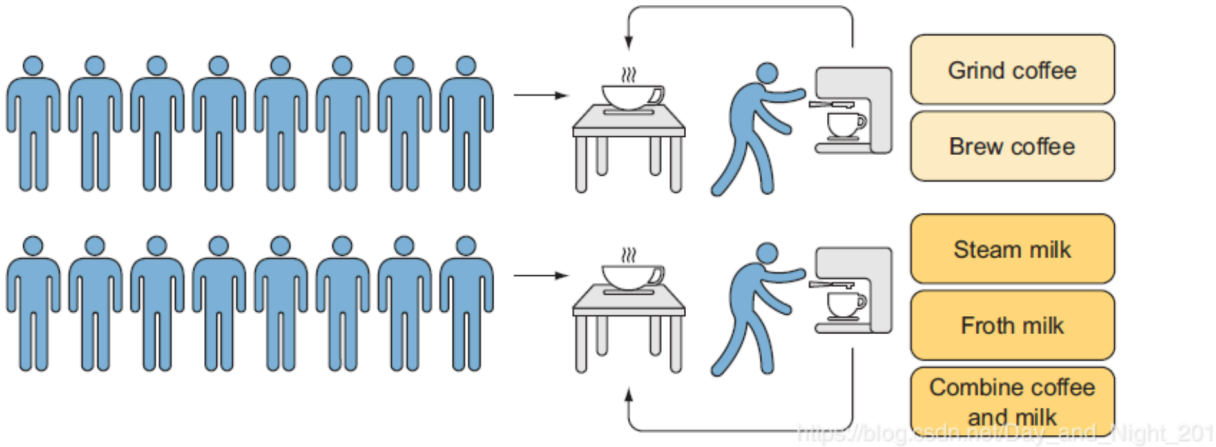
什么是并发？**并发(Concurrency)和并行(Parallelism)**是非常容易混淆的概念，他们都可以表示两个或多个任务一起执行，并发偏重于多个任务交替执行，期间也可能串行。而并行是真正意义上的“同时执行”，并行的核心是CPU数量，当CPU数量大于1个时才有并行的可能，如果只有一个处理器，没有并行的概念。

如图：

并发



并行



临界区

临界区用来表示一种公共资源或共享数据，可以被多个线程使用，但每一次只能有一个线程使用它，如果临界区被一个线程所占用，其他线程必须等待。

三大特性

在java的并发编程中，如果要保证程序的线程安全，就要保证代码的**原子性，可见性，有序性**。

原子性

原子性指一个操作是不可中断的，即使是在多个线程一起执行，一个操作一旦开始，就不会被其他线程干扰。

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。
如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。
这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。
所以这2个操作必须要具备原子性才能保证不出现一些意外的问题

可见性

可见性是指一个线程修改了某个共享变量的值时，其他线程是否能够立即知道这个修改。显然在串行程序中，不存在这种问题，因为在串行程序中，后面执行的步骤读到的变量值永远是更新后的值。

对于可见性，Java提供了volatile关键字来保证可见性。
当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主内存，以及每次使用前立即从主内存刷新。

有序性

对于一个线程的执行代码而言，我们习惯性的认为它是从前往后依次执行的，对于一个线程内自然是这样，不然应用根本无法工作，但在并发时可能会出现乱序，因为程序在执行时可能进行指令重排序。

java也提供了一些关键字或实现类保证有序性 例如：**synchronized**

进程和线程

进程

进程是系统进行资源分配和调度的基本单位，而线程则是程序执行的最小单位，线程生活在进程中。

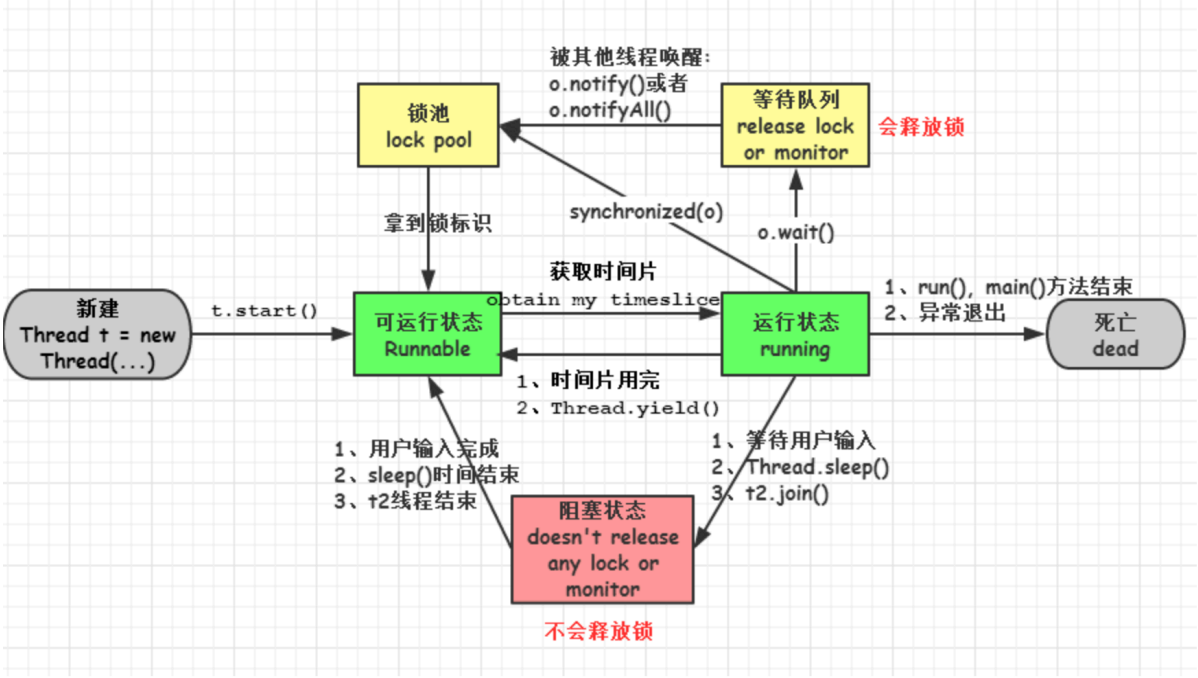
在java中java虚拟机**java.exe**作为一个进程存在，而编译后的class文件只能被加载到虚拟机中作为一个线程运行。

线程的生命周期

线程的所有状态：

```
public enum State {

        NEW,

        RUNNABLE,

        BLOCKED,

        WAITING,

        TIMED_WAITING,

        TERMINATED;
    }
```

NEW表示刚刚创建的线程,这种线程还没开始执行，等到线程的start()方法被调用时，才表示线程开始执行，执行时处于**RUNNABLE**状态，如果线程在执行中遇到了synchronized同步块，就会进入**BLOCKED**状态，线程会暂停执行，直到获取到请求的锁。**WAITING**和**TIMED_WAITING**都表示等待状态，他们的区别是WAITING会进入一个无时间限制的等待，TIMED_WAITING会进入一个有时限的等待，一般等待的是一些特殊的事件，比如，通过wait()方法等待的线程在等待notify()方法，而通过join()方法等待的线程则等待目标线程的终止，一旦等到了期望的事件，线程就会再次执行，进入**RUNNABLE**状态，当线程执行完毕后，则进入**TERMINATED**状态，表达结束。



线程

创建线程这个问题就不必多加描述了，继承Thread类或者实现Runnable接口(推荐)，也可使用线程池获取线程。

如何关闭一个线程呢?

一般来说线程执行完毕后就会结束，无需手动关闭。但还是有关闭线程的API

Thread类提供了~~stop()~~, jdk不推荐使用此方法，强行将执行一半的线程终止，可能引起数据不统一的问题。

那如何正确的关闭一个线程呢,我们可以在线程里面加上while循环和标识符，若达到了条件就break跳出循环

```
public class MyRunnable implements Runnable {

  //定义退出标志，true会一直执行，false会退出循环
  public volatile boolean flag = true;

  public void run() {
    System.out.println("第" + Thread.currentThread().getName() + "个线程创建");

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

    //退出标志生效位置
```

```
    while (flag) {
    }
    System.out.println("第" + Thread.currentThread().getName() + "个线程终止");
  }
```

线程中断

正因为stop()过于暴力,所以jdk又提供了线程中断机制。

严格的来讲，线程中断并不会使线程立即退出，而是给线程发送一个通知，告诉线程，有人希望你退出。至于线程接到通知如何处理，完全由线程自己决定。

```
    public void Thread.interrupt()             //中断线程
public boolean Thread.isInterrupt()       //判断是否被中断
public static boolean Thread.interrupt() //判断是否被中断，并清除当前中断状态
public static native void sleep(long millis) throws InterruptedException //线程休眠若干时间
```

```
public static void main(string [] args) throws InterruptedException{
    Thread t1 = new Thread(){
        @Override
        public void run(){
            while(true){
                //do someting
            }
        }
    };
    t1.start();
    Thread.sleep(2000);
    t1.interrupt();
}
```

在这里，虽然对t1进行了中断，但在ti中并未有处理中断的逻辑，因此这个中断也不会起作用。

类似我们自己使用标识符退出，我们也要给出中断处理：

```
public static void main(string [] args) throw InterruptedException{
    Thread t1 = new Thread(){
        @Override
        public void run(){
            while(true){

                if(Thread.currentThread().isIntrrupted()){
                  //do someting
                  break;
```

```
                }

                //do someting
            }
        }
    };
    t1.start();
    Thread.sleep(2000);
    t1.interrupt();
}
```

sleep()方法会让线程休眠若干时间，他会抛出一个InterruptedException中断异常，InterruptedException 不是运行时异常，也就是说程序必须捕获他，当线程在sleep休眠时，如果被中断，这个异常就会产生。

注：

《实战Java高并发程序设计》：

> Thread.sleep()方法由于中断而抛出异常，此时，他会清除中断标记，如果不在加处理，那么在下次循环开始时，就无法捕捉这个中断，故在异常处理中，再次设置中断标记位。

```
public static void main(string [] args) throw InterruptedException{
    Thread t1 = new Thread(){
        @Override
        public void run(){
            while(true){

                if(Thread.currentThread().isIntrrupted()){
                  //do someting
                  break;
                }
                //do someting
                try{
                    Thread.sleep(2000)
                } catch(InterruptedException e){
                    Thread.currentThread().interrupt();//再次设置中断状态
                }
            }
        }
    };
    t1.start();
    Thread.sleep(2000);
    t1.interrupt();
}
```

等待(wait)和通知(notify)

为了支持多线程之间的协作，JDK提供了两个重要的方法，等待**wait()**方法，通知**notify()**方法，这是两个Object类中的方法。意味着任何对象都有这两个方法。

方法签名如下：

```
public final void wait() throws InterruptedException
public final native void notifyAll();
```

线程A在一个对象实例obj上调用了wait()后，线程A就会在obj这个对象上等待，一直等到其他线程调用了obj.notify(),所以这个对象就成了线程间的交互手段。

假如有多个线程在wait，当某个线程调用了notify方法时，会唤醒具体的哪一个呢?

不会根据wait的先后顺序唤醒，而是完全随机的。

notifyAll()此方法会唤醒所有等待的线程。

等待线程结束（join）和谦让(yeild)

方法签名：

```
public final void join() throws InterruptedException
public final synchronized void join(long millis) throws InterruptedException
```

第一个join方法表示无限等待，它会一直阻塞当前线程，直到目标线程执行完毕。第二个会给出一个最大等待时间，如果超过时间，目标线程还未执行完毕，则当前线程不在阻塞，继续执行下去。

列子：

```
public class JoinMain{
    public volatile static int i = 0;
    public static class AddThread extends Thread{
        @Override
        public void run(){
            for(i=0 ; i< 10000000 ; i++);
        }
    }

    public static void main(String[] args) throws InterruptedException{
        AddThread t = new AddThread();
        t.start();
        t.join();
        System.out.println(i);
    }

}
```

在main方法中，如果不使用join方法等待线程t,那么得到的i很可能是0或者是一个很小的数字，因为t还没执行或者还没执行完，主线程就结束了，使用了join方法，表示主线程愿意等待线程t，才能得到正确结果。

```
public static native void yield();
```

如果一个线程调用了yield()方法，他会使当前的线程让出cpu，但不代表他不执行了，他让出cpu后，还会进行cpu资源的竞争。

守护线程(Daemon)

守护线程是一种特殊的线程，它是系统的守护者，在后台完成一些系统性的服务，比如垃圾回收线程，与之相对的是用户线程，如果用户线程全部结束，则意味着守护线程要守护的对象已经不存在了，当整个应用程序退出时，守护线程才会结束。

```
public class DaemonDemo{
    public static class DaemonT extends Thread{
        @Override
        public void run(){
            while(true){
                System.out.Println("I am alive")
                  try{
                      Thread.sleep(1000);
                  } catch (InterruptedExcption e){
                      e.printStackTrace();
                  }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException{
        DaemonT t = new DaemonT();
        t.setDaemon(true);  //将当前线程设置成守护线程
        t.start();
        Thread.sleep(2000);
    }
}
```

设置守护线程一定要在调用start方法之前设置，否则会抛出异常，设置守护线程失败，但线程依旧正常执行，只是被当做用户线程了。

在上面的例子中，将线程t设置为守护线程，系统中只有main是用户线程，所以当main线程休眠2秒后退出时，整个程序也就结束了，t线程也会停止执行，如果不把t设为守护线程，那在main线程停止后，t会不停打印，永远不会结束。

线程优先级

线程可以有自己的优先级，优先级高的线程在竞争资源时会更有优势，但这只是概率问题，优先级高的拥有更高的概率，如果运气不好，高优先级也可能抢占资源失败。

Thread类自带了三个指标：

```
/**
     * The minimum priority that a thread can have.
     */
```

```
    public final static int MIN_PRIORITY = 1;

   /**
     * The default priority that is assigned to a thread.
     */
    public final static int NORM_PRIORITY = 5;

    /**
     * The maximum priority that a thread can have.
     */
    public final static int MAX_PRIORITY = 10;
```

有效值在**1-10**之间 可在在调用start方法之前调用**setPriority**方法设置优先级。

线程安全

并发程序的一大关注重点就是线程安全，在线程安全的前提下使用并发编程，才是我们需要的。不然并行程序也就没什么意义。

synchronized

java 提供了**synchronized**关键字，它的作用就是实现线程间的同步。对同步的代码加锁，使得每一次只有一个线程进入同步块，从而保证安全性。

用法：

- 指定加锁对象，给定对象加锁，进入同步代码前要获得给定对象的锁。

```
public class Demo implements Runnable{
    static Demo instance = new Demo();
    static int i = 0;
    @Override
    public void run(){
        for(int j = 0 ; j<100000;j++){
            synchronized(instance){
                i++;
            }
        }
    }
    //main 略
}
```

- 直接作用于实例方法：相当于对当前实例加锁，进入同步方法前需要获得当前实例对象的锁。

```
public class Demo implements Runnable{
    static Demo instance = new Demo();
    static int i = 0;
    public synchronized void increase(){
```

```
        i++;
    }
    @Override
    public void run(){
        for(int j = 0 ; j<100000;j++){
             increase();
        }
    }

    public static void main(String[] args) throws InterruptedException{
        Demo t1 = new Thread(instance); //此处instance 一定要同一对象
        Demo t2 = new Thread(instance);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.Println(i)
    }

}
```

- 直接作用于静态方法：相当于对当前类加锁。进入同步代码时需要获得当前类的锁。

```
public class Demo implements Runnable{
    static Demo instance = new Demo();
    static int i = 0;
    public static synchronized void increase(){
        i++;
    }
    @Override
    public void run(){
        for(int j = 0 ; j<100000;j++){
             increase();
        }
    }

    public static void main(String[] args) throws InterruptedException{
        Demo t1 = new Thread(new Demo());
        Demo t2 = new Thread(new Demo());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.Println(i)
    }

}
```

java.util.concurrent

JDK5.0 java引入java.util.concurrent并发包，其中提供了许多并发编程很有用的工具。

重入锁

什么是重入锁?

使用了重入锁表示当前线程可以反复加锁却不导致死锁，但也需要释放同样加锁次数的锁，即重入了多少次，就要释放多少次，不然也会导致死锁。

synchronized也是重入锁

在juc并发包中也提供了重入锁，**java.util.concurrent.locks.ReentrantLock**

它完全可以替代 synchronized 关键字来实现它的所有功能，而且 ReentrantLock 锁的灵活度要远远大于 synchronized 关键字。

在JDK5.0时效率远高于synchronized，但在JDK6.0开始,JDK在synchronized 关键字上做出了大量优化，使得两者性能差距不大。

```
public class ReentrantLockTest {

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

        ReentrantLock lock = new ReentrantLock();

        for (int i = 1; i <= 3; i++) {
            lock.lock();
        }

        for(int i=1;i<=3;i++){
            try {

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

上面的代码通过`lock()`方法先获取锁三次，然后通过`unlock()`方法释放锁3次，程序可以正常退出。从上面的例子可以看出,ReentrantLock是可以重入的锁,当一个线程获取锁时,还可以接着重复获取多次。

跟synchronized 不同的是 ReentrantLock是需要手动释放锁的。ReentrantLock还提供了些高级功能。

- **lockInterruptibly()**:获得锁，但优先响应中断。

lockInterruptibly()方法能够中断等待获取锁的线程。当两个线程同时通过lock.lockInterruptibly()获取某个锁时，假若此时线程A获取到了锁，而线程B只有等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

```
public class LockTest {

    private Lock lock = new ReentrantLock();

    public void doBussiness() {
        String name = Thread.currentThread().getName();

        try {
            System.out.println(name + " 开始获取锁");
            lock.lockInterruptibly();
            System.out.println(name + " 得到锁");
            System.out.println(name + " 开工干活");
            for (int i=0; i<5; i++) {
                Thread.sleep(1000);
                System.out.println(name + " : " + i);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " 被中断");
            System.out.println(name + " 做些别的事情");
        } finally {
            try {
                lock.unlock();
                System.out.println(name + " 释放锁");
            } catch (Exception e) {
                System.out.println(name + " : 没有得到锁的线程运行结束");
            }
        }
    }

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

        LockTest lockTest = new LockTest();

        Thread t0 = new Thread(
                new Runnable() {
                    public void run() {
                        lockTest.doBussiness();
                    }
                }
                );

        Thread t1 = new Thread(
                new Runnable() {
                    public void run() {
                        lockTest.doBussiness();
                    }
                }
                );
```

```
        // 启动线程t1
        t0.start();
        Thread.sleep(10);
        // 启动线程t2
        t1.start();
        Thread.sleep(100);
        // 线程t1没有得到锁，中断t1的等待
        t1.interrupt();
    }
}
```

- **tryLock()**:尝试获得锁，如果成功，则返回true,否则返回false，该方法不等待，立即返回。

```
public class TryLockTest {
  //实例化Lock对象
  Lock lock = new ReentrantLock();

  /**
   * @param args
   */
  public static void main(String[] args) {
    //实例化本类对象，目的是调用runThread方法
    TryLock tl = new TryLockTest();
    //匿名对象创建线程1，并重写run方法，启动线程
    new Thread(){
      public void run(){

        tl.runThread(Thread.currentThread());
      }
    }.start();
    //匿名对象创建线程2，并重写run方法，启动线程
    new Thread(){
      public void run(){

        tl.runThread(Thread.currentThread());
      }
    }.start();


  }
  //线程共同调用方法
  public void runThread(Thread t){
    //lock对象调用trylock()方法尝试获取锁
    if(lock.tryLock()){
      //获锁成功代码段
      System.out.println("线程"+t.getName()+"获取锁成功");
      try {
```

```
            //执行的代码
            Thread.sleep(5000);
        } catch (Exception e) {
            //异常处理内容，比如中断异常，需要恢复等
        } finally {
            //获取锁成功之后，一定记住加finally并unlock()方法,释放锁
            System.out.println("线程"+t.getName()+"释放锁");
            lock.unlock();
        }
    }else{
        //获锁失败代码段
        //具体获取锁失败的回复响应
        System.out.println("线程"+t.getName()+"获取锁失败");
    }
  }
}
```

- **tryLock(long time, TimeUnit unit)**:在给定时间内尝试获得锁。

读写锁（ReadWriteLock）

ReadWriteLock管理一组锁，一个是只读的锁，一个是写锁。
Java并发库中ReetrantReadWriteLock实现了ReadWriteLock接口并添加了可重入的特性。

读写锁允许多个线程同时读，使得线程之间真正的并行。但是考虑到数据完整性，写写操作和读写操作之间还是需要互相等待和持有锁的。如图：

ReentrantReadWriteLock 是对读写锁的具体实现类，从他的名字可看出，也具有重入锁的性质。

在读操作远大于写操作时，使用读写锁的性能会有明显的提升。

```
/** 读锁 */
    private final ReentrantReadWriteLock.ReadLock readerLock;

    /** 写锁 */
    private final ReentrantReadWriteLock.WriteLock writerLock;

    final Sync sync;

    /** 使用默认（非公平）的排序属性创建一个新的 ReentrantReadWriteLock */
    public ReentrantReadWriteLock() {
        this(false);
    }

    /** 使用给定的公平策略创建一个新的 ReentrantReadWriteLock */
    public ReentrantReadWriteLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
```

```
        readerLock = new ReadLock(this);
        writerLock = new WriteLock(this);
    }

    /** 返回用于写入操作的锁 */
    public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }

    /** 返回用于读取操作的锁 */
    public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }


    abstract static class Sync extends AbstractQueuedSynchronizer {}

    static final class NonfairSync extends Sync {}

    static final class FairSync extends Sync {}

    public static class ReadLock implements Lock, java.io.Serializable {}

    public static class WriteLock implements Lock, java.io.Serializable {}
```

CountDownLatch

CountDownLatch是通过一个计数器来实现的，当我们在new 一个CountDownLatch对象的时候需要带入该计数器值，该值就表示了线程的数量。每当一个线程完成自己的任务后，计数器的值就会减1。当计数器的值变为0时，就表示所有的线程均已经完成了任务，然后就可以恢复等待的线程继续执行了。

CountDownLatch所描述的是”在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待“。在API中是这样描述的：

用给定的计数 初始化 CountDownLatch。由于调用了**countDown()** 方法，所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用**CyclicBarrier**。

```
public class CountDownLatchTest {
  private static CountDownLatch countDownLatch = new CountDownLatch(5);

  static class BossThread extends Thread {
    @Override
    public void run() {
      System.out.println("Boss在会议室等待，总共有" + countDownLatch.getCount() +
"个人开会...");
      try {
        // Boss等待
        countDownLatch.await();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
```

```
      System.out.println("所有人都已经到齐了，开会吧...");
    }

    // 员工到达会议室
    static class EmpleoyeeThread extends Thread {
      @Override
      public void run() {
        try {
          System.out.println(Thread.currentThread().getName() + "，到达会议
室....");
        }finally {
          // 员工到达会议室 count - 1
          countDownLatch.countDown();
        }
      }
    }

    public static void main(String[] args) {
      // Boss线程启动
      new BossThread().start();

      long cnt = countDownLatch.getCount();

      for (long i = 0; i < cnt; i++) {
        new EmpleoyeeThread().start();
      }
    }
  }
}
```

此外,jcu并发包还提供了许多有用的原子类,和许多高性能的并发集合

如:**ConcurrentHashMap,CopyOnWriteArrayList** 等等。

ThreadLocal

从名字中可看出,这是一个线程的局部变量,也就是说,只有当前线程可以访问,自然线程安全的。

```
public class ThreadLocalTest  {
    /*定义一个全局变量 来存放线程需要的变量*/
    public static ThreadLocal<Integer> ti = new ThreadLocal<Integer>();
    public static void main(String[] args) {
        /*创建两个线程*/
        for(int i=0; i<2;i++){
            new Thread(new Runnable() {
                @Override
                public void run() {
                    Double d = Math.random()*10;
```

```
                    /*存入当前线程独有的值*/
                    ti.set(d.intValue());
                    new A().get();
                    new B().get();
                }
            }).start();
        }
    }
    static class A{
        public void get(){
            /*取得当前线程所需要的值*/
            System.out.println(ti.get());
        }
    }
    static class B{
        public void get(){
            /*取得当前线程所需要的值*/
            System.out.println(ti.get()+" "+Thread.currentThread().getName());
        }
    }
}
```

上面的例子中，这两个线程里面的A和B类打印的值都是相同的。

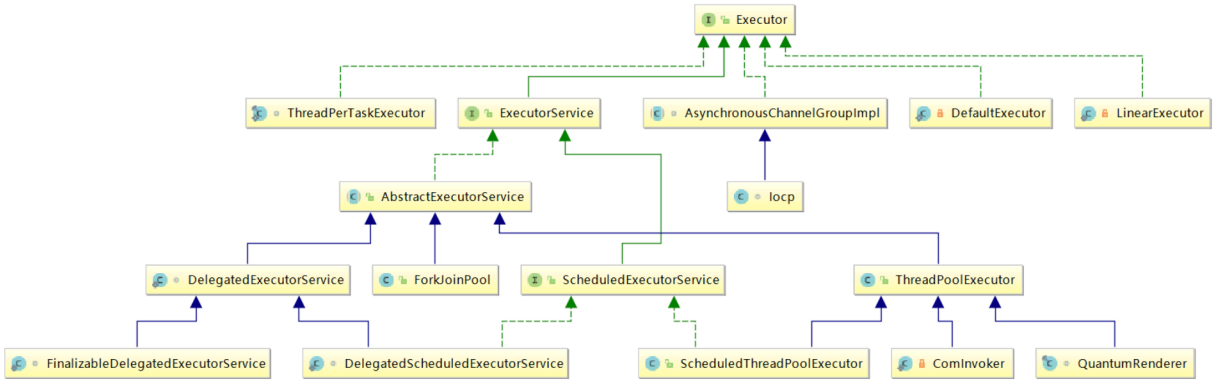
线程池

为什么需要线程池?

资源非常宝贵，减少创建和销毁线程的次数，让每个线程可以多次使用,可根据系统情况**调整执行**的线程数量，防止消耗过多内存，所以需要使用线程池。

ThreadPoolExecutor

关系图如下:



jdk1.5提供了四种线程池，通过**Executors**类来获取：

1. newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

2. newFixedThreadPool

创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

3. newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。

> 更新于 2020.6.25，通过b3log的开源项目源码中学习下ScheduledThreadPoolExecutor

在solo中创建了一个用于定时任务的线程池

```
    /**
     * Cron thread pool.
     */
    private static final ScheduledExecutorService SCHEDULED_EXECUTOR_SERVICE =
  Executors.newScheduledThreadPool(1);
```

然后通过这个线程去执行定时任务

```
/**
     * Start all cron tasks.
     */
    public void start() {
        long delay = 10000;

        SCHEDULED_EXECUTOR_SERVICE.scheduleAtFixedRate(() -> {
            try {
                StatisticMgmtService.removeExpiredOnlineVisitor();
            } catch (final Exception e) {
                LOGGER.log(Level.ERROR, "Executes cron failed", e);
            } finally {
```

```
                Stopwatchs.release();
            }
        }, delay, 1000 * 60 * 10, TimeUnit.MILLISECONDS);
        delay += 2000;

        SCHEDULED_EXECUTOR_SERVICE.scheduleAtFixedRate(() -> {
            try {
                Solos.reloadBlacklistIPs();
            } catch (final Exception e) {
                LOGGER.log(Level.ERROR, "Executes cron failed", e);
            } finally {
                Stopwatchs.release();
            }
        }, delay, 1000 * 60 * 30, TimeUnit.MILLISECONDS);
        delay += 2000;

        SCHEDULED_EXECUTOR_SERVICE.scheduleAtFixedRate(() -> {
            try {
                articleMgmtService.refreshGitHub();
                userMgmtService.refreshUSite();
                exportService.exportHacPai();
                exportService.exportGitHub();
            } catch (final Exception e) {
                LOGGER.log(Level.ERROR, "Executes cron failed", e);
            } finally {
                Stopwatchs.release();
            }
        }, delay, 1000 * 60 * 60 * 24, TimeUnit.MILLISECONDS);
        delay += 2000;
    }
```

4. SingleThreadPool

   创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务。

《阿里巴巴开发手册》：

> Executors 返回的线程池对象 的弊端 如下
>
> 1 FixedThreadPool 和 SingleThread Pool
> 允许的请求队列长度为 Integer.MAX_VALUE，可 能会堆积大量的请求，从而导致 OOM。
> 2 CachedThreadPool 和 ScheduledThreadPool
> 允许的创建线程数量为 Integer.MAX_VALUE 可能会创建大量的线程，从而导致 OOM。

所以更推荐使用**ThreadPoolExecutor**来自定义线程池，查看Executor源码可知上面4种线程池也是用ThreadPoolExecutor创建的。

其中的一个构造方法

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

主要参数

- int corePoolSize, 核心线程数量量
- int maximumPoolSize, 最大线程数量
- long keepAliveTime, 空闲线程在多长时间内会被销毁
- TimeUnit unit, keepAliveTime空闲时间单位
- BlockingQueue workQueue, 阻塞队列
- ThreadFactory threadFactory，线程工厂
- RejectedExecutionHandler handler 拒绝策略;当线程和队列满时处理新任务方案

大部分参数都简单易懂，只有workQueue需要详细说明，因为根据使用的队列不同，性能上也大不同，上述四种有缺陷的系统自带线程池也是由各自队列的属性所导致

- 直接提交的队列，**SynchronousQueue**，这个队列没有容量，每一个插入操作都要等待一个相应的删除操作，反之亦然，如果使用它，提交的任务不会被真实的保存，而总是将新任务交给线程执行，如果没有空闲的线程，则会尝试创建新的线程，如果线程数已达最大，就会执行拒绝策略，由于CachedThreadPool 的最大线程数为**Integer.MAX_VALUE**，所以可能会不断创建线程，导致OOM
- 有界队列，有界队列可以用**ArrayBlockingQueue**类来实现，此类的构造方法必须带一个容量参数，表示该队列的最大容量，当使用有界队列时，若有新任务需要执行时，如果线程池的线程数小于corePoolSize,则会优先创建新的线程，若大于corePoolSize,将新任务加入等待队列。若队列已满，无法加入，则在总线程数不大于maximumPoolSize的前提下创建新的线程执行任务，若大于maximumPoolSize，则执行拒绝策略。
- 无界队列，**LinkedBlockingQueue**类实现。与有界队列相比，除非系统资源用尽，否则一直能够任务入队，如果线程池的线程数小于corePoolSize,则会优先创建新的线程，若达到了corePoolSize,就不会继续增加了，后续任务进来，直接进入队列等待，直到系统内存耗尽，FixedThreadPool 和 SingleThreadPool 就是这种情况。

jdk1.8，Executors又提供了一个线程池**newWorkStealingPool**,顾名思义这个线程池是一个具有**抢占**式操作的线程池

newWorkStealingPool适合使用在很耗时的操作，但是newWorkStealingPool不是ThreadPoolExecutor的扩展，它是新的线程池类ForkJoinPool的扩展，但是都是在统一的一个Executors类中实现，由于能够合理的使用CPU进行对任务操作（并行操作），所以适合使用在很耗时的任务中

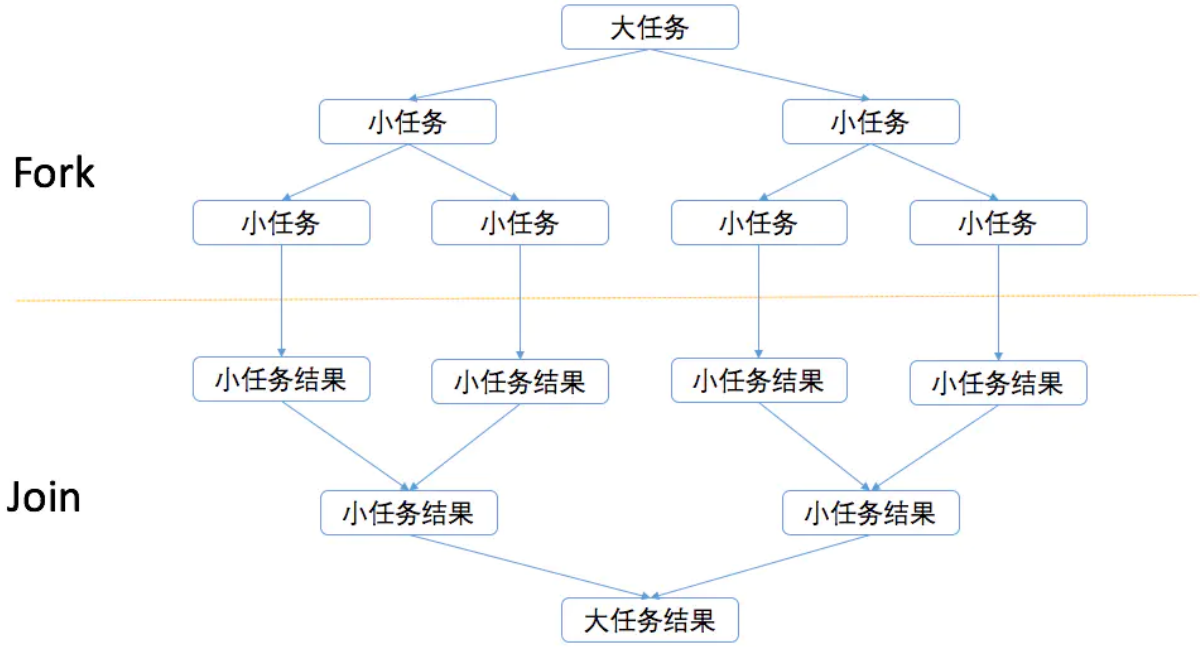
```
/**
     * Creates a thread pool that maintains enough threads to support
     * the given parallelism level, and may use multiple queues to
```

```
     * reduce contention. The parallelism level corresponds to the
     * maximum number of threads actively engaged in, or available to
     * engage in, task processing. The actual number of threads may
     * grow and shrink dynamically. A work-stealing pool makes no
     * guarantees about the order in which submitted tasks are
     * executed.
     *
     * @param parallelism the targeted parallelism level
     * @return the newly created thread pool
     * @throws IllegalArgumentException if {@code parallelism <= 0}
     * @since 1.8
     */
    public static ExecutorService newWorkStealingPool(int parallelism) {
        return new ForkJoinPool
            (parallelism,
             ForkJoinPool.defaultForkJoinWorkerThreadFactory,
             null, true);
    }
```

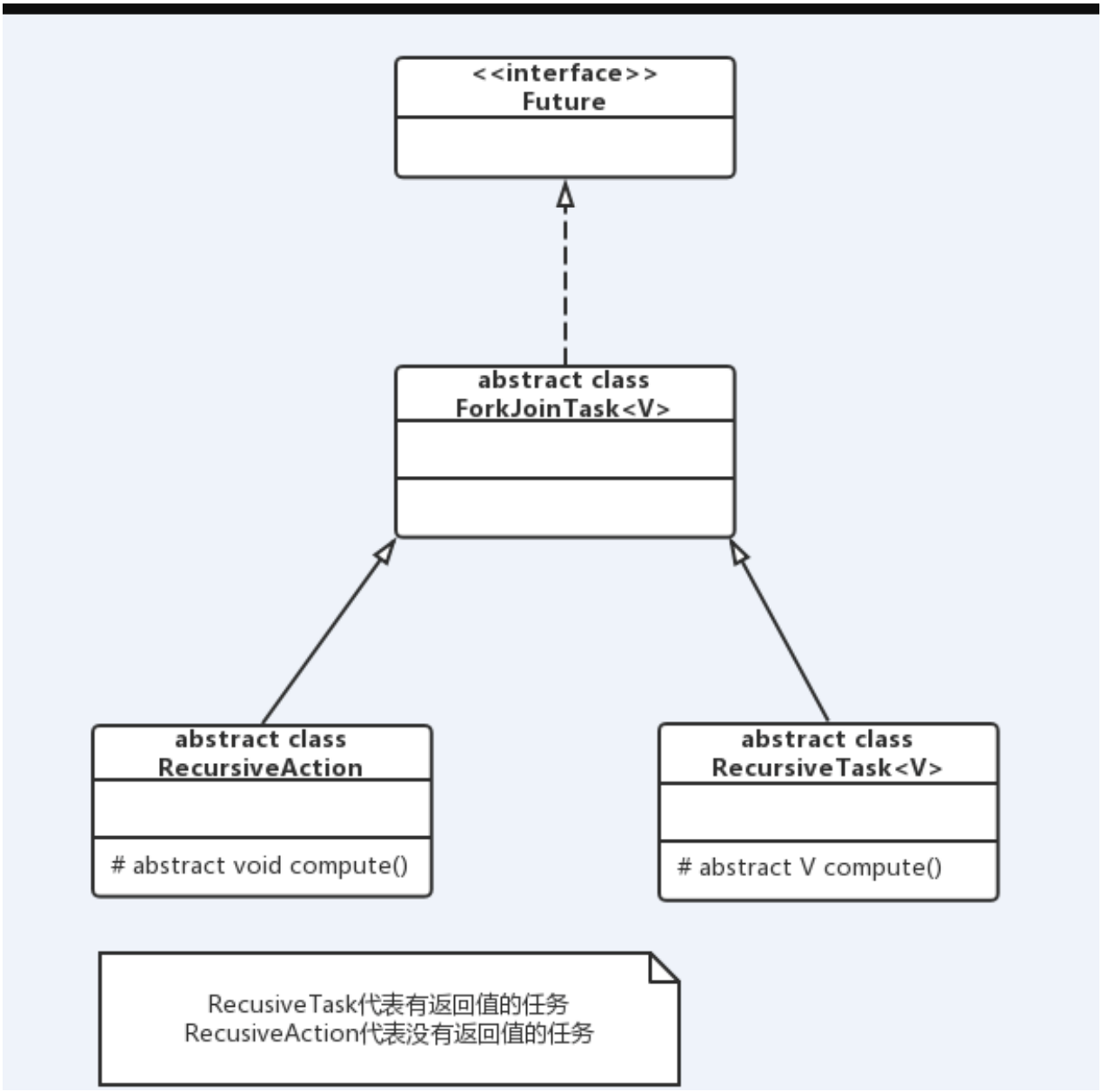
ForkJoinPool

JDK7.0提供了ForkJoinPool，ForkJoinPool也是ExecutorService的实现类，因此是一种特殊的线程池。

顾名思义，ForkJoinPool运用了Fork/Join原理，使用“分而治之”的思想，将大任务分拆成小任务分配给多个线程执行，最后合并得到最终结果，加快运算。



创建了ForkJoinPool实例之后，就可以调用ForkJoinPool的**submit(ForkJoinTask task)**或**invoke(ForkJoinTask task)**方法来执行指定任务了。




```
public class ForkJoinPoolTest {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        int count  = 5000000;
        SumTask sumTask = new SumTask(0, count);
        ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
        ForkJoinTask<Integer> submit = forkJoinPool.submit(sumTask);
        System.out.println(submit.get());
        int s = 0;
        for (int i = 0;i<=count;i++) {
            s +=i;
        }
        System.out.println(s);
    }


    public static class SumTask extends RecursiveTask<Integer> {
```

```
        private final static int threshold = 5000;

        private int start;

        private int end;

        public SumTask() {
        }

        public SumTask(int start, int end) {
            this.start = start;
            this.end = end;
        }

        @Override
        protected Integer compute() {
            int sum = 0;
            if (end - start < threshold){
                for (int i = start; i< end; i++){
                    sum +=i;
                }
            }else {
                int middle = (start + end) / 2;
                SumTask sumTask = new SumTask(start, middle);
                SumTask sumTask1 = new SumTask(middle, end);
                //sumTask.fork();
                //sumTask1.fork();
                SumTask.invokeAll(sumTask, sumTask1);
                sum = sumTask.join() + sumTask1.join();
            }
            return sum;
        }
    }
}
```

自定义 ForkJoinPool

在 java8 中 添加了流Stream，可以让你以一种声明的方式处理数据。使用起来非常简单优雅。ParallelStream 则是一个并行执行的流，内部ForkJoinPool 并行执行任务，提高执行速度。

Parallel Stream，默认采用的是一个 ForkJoinPool.commonPool 的线程池，这样我们就算使用了 Parallel Stream，整个 jvm 共用一个 common pool 线程池，一不小心就任务堆积了，所以我们也可以根据自己机器的CPU核心数自定义ForkJoinPool。

ForkJoinPool forkJoinPool = new ForkJoinPool(8); 此参数为你CPU的核心数

public ForkJoinTask<?> submit(Runnable task) 也可直接接受一个Runnable

```
List<String> list = new ArrayList<>();
ForkJoinPool forkJoinPool = new ForkJoinPool(8);
forkJoinPool.submit(() -> list.parallelStream()
    .collect(Collectors.toList())
).get();
```

Timer

在开发中，我们经常需要一些周期性的操作，例如每隔几分钟就进行某一项操作。这时候我们就要去设置个定时器，Java中最方便、最高效的实现方式是用java.util.Timer工具类，再通过调度java.util.TimerTask任务。

Timer是一种工具，线程用其安排以后在后台线程中执行的任务。可安排任务执行一次，或者定期重复执行。实际上是个线程，定时调度所拥有的TimerTasks。

TimerTask是一个抽象类，它的子类由 Timer 安排为一次执行或重复执行的任务。实际上就是一个拥有run方法的类，需要定时执行的代码放到run方法体内。

```
Timer timer = Timer(true);  //设置为true 则是守护线程

TimerTask task = new TimerTask() {
    public void run() {
        ... //每次需要执行的代码放到这里面。
    }
};

//以下是几种常用调度task的方法：

timer.schedule(task, time);
// time为Date类型：在指定时间执行一次。

timer.schedule(task, firstTime, period);
// firstTime为Date类型,period为long
// 从firstTime时刻开始，每隔period毫秒执行一次。

timer.schedule(task, delay)
// delay 为long类型：从现在起过delay毫秒执行一次

timer.schedule(task, delay, period)
// delay为long,period为long：从现在起过delay毫秒以后，每隔period
// 毫秒执行一次。
```

通过b3log的开源项目**symphony**来学习下Timer的使用

```
// 向 Rhy 发送统计数据，仅发送站点名称、URL。用于 Sym 使用统计，如果不想发送请移除该代码
        new Timer(true).schedule(new TimerTask() {
            @Override
            public void run() {
                final String symURL = Latkes.getServePath();
                if (Strings.isIPv4(symURL)) {
                    return;
                }

                HttpURLConnection httpConn = null;
                try {
                    final BeanManager beanManager = BeanManager.getInstance();
                    final OptionQueryService optionQueryService =
beanManager.getReference(OptionQueryService.class);

                    final JSONObject statistic =
optionQueryService.getStatistic();
                    final int articleCount =
statistic.optInt(Option.ID_C_STATISTIC_ARTICLE_COUNT);
                    if (articleCount < 66) {
                        return;
                    }

                    final LangPropsService langPropsService =
beanManager.getReference(LangPropsService.class);

                    httpConn = (HttpURLConnection) new
URL("https://rhythm.b3log.org/sym").openConnection();
                    httpConn.setConnectTimeout(10000);
                    httpConn.setReadTimeout(10000);
                    httpConn.setDoOutput(true);
                    httpConn.setRequestMethod("POST");
                    httpConn.setRequestProperty(Common.USER_AGENT,
USER_AGENT_BOT);

                    httpConn.connect();

                    try (final OutputStream outputStream =
httpConn.getOutputStream()) {
                        final JSONObject sym = new JSONObject();
                        sym.put("symURL", symURL);
                        sym.put("symTitle",
langPropsService.get("symphonyLabel", Latkes.getLocale()));

                        IOUtils.write(sym.toString(), outputStream, "UTF-8");
                        outputStream.flush();
                    }

                    httpConn.getResponseCode();
```

```
                } catch (final Exception e) {
                    // ignore
                } finally {
                    if (null != httpConn) {
                        try {
                            httpConn.disconnect();
                        } catch (final Exception e) {
                            // ignore
                        }
                    }
                }
            }
        }, 1000 * 60 * 60 * 2, 1000 * 60 * 60 * 12);
```

Springboot Async 异步方法

本质是使用多线程来达到异步的效果，当主线程执行到当前异步函数，则会由线程池(可以自己指定所用的线程池)分配一个新的线程去执行，主线程直接返回

```
@SpringBootApplication
@EnableAsync  //添加上此注解 开启异步方法支持
@EnableNeo4jRepositories
@EnableSkyCloudGlobalResponse
public class CmdbApiApplication
```

```
@Configuration
@EnableAsync
public class AsyncConfiguration
{
    @Bean(name = "asyncExecutor") //自定义线程池
    public Executor asyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(20);
        executor.setMaxPoolSize(50);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("DomAsyncThread-");
        executor.initialize();
        return executor;
    }
}
```

```
@Async("asyncExecutor") //将此方法标注为异步方法，开启线程去执行
    public void uploadExcelAsync(MultipartFile file, String username) throws
Exception {
        LOGGER.info("async import device from excel and send websocket ...");
        JSONArray array = uploadExcel(file, username);
        JSONObject[] arr = array.toArray(new JSONObject[1]);
        for (int pos = 0; pos < arr.length; pos += 2000) {
            int length = pos + 2000 <= arr.length ? 2000 : arr.length - pos;
            JSONObject[] brr = new JSONObject[length];
            System.arraycopy(arr, pos, brr, 0, length);
            String msg = String.format("用户 [%s] 通过excel导入 %s 个设备",
username, length);
            sendEvent(username, msg, brr);
        }
    }
```