链滴

并发编程进阶，深入浅出理解 JUC

作者：iceTang

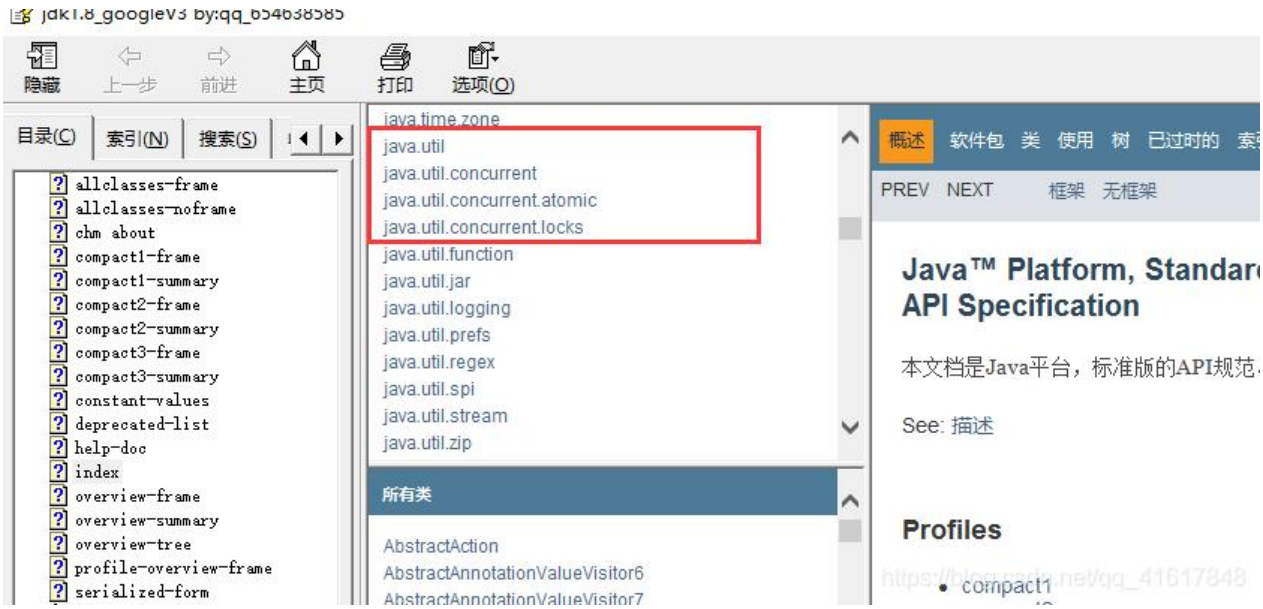
原文链接：https://ld246.com/article/1620724674275

来源网站：链滴

JUC多线程进阶

1.什么是JUC

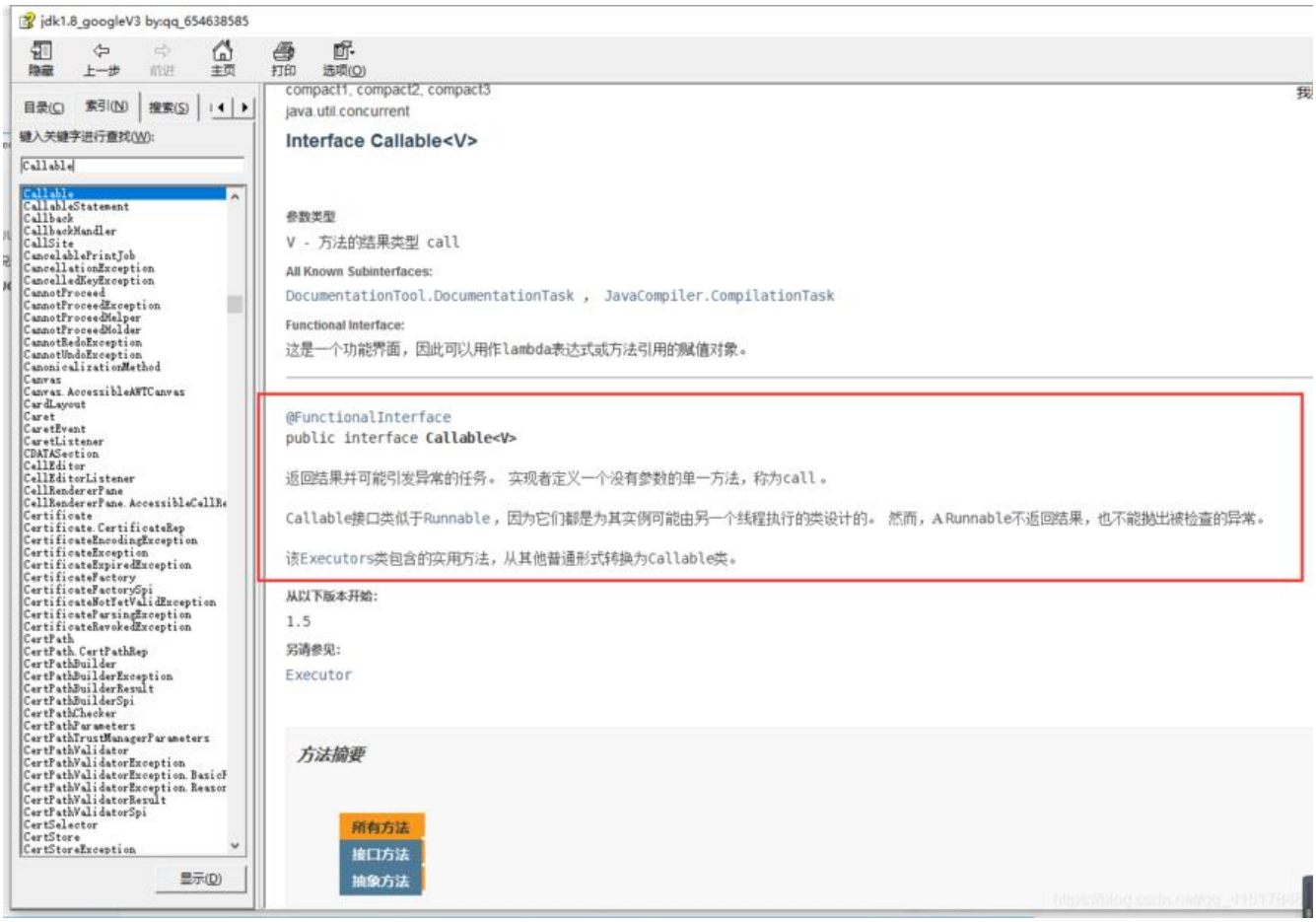
源码 + 官方文档



JUC是 java util concurrent

业务：普通的线程代码 Thread

Runnable： ** 没有返回值、效率相比于**Callable** 相对较低！**



2.线程和进程

线程与进程的关系

进程是一个程序的执行；一个进程可以包含多个线程，线程是程序执行的最小单位（资源调度的最小位）

java默认有几个线程?

2个，main与GC

线程：开了一个进程Typore，写字，自动保存（线程负责的，输入）

对于Java而言：Thread、Runnable、Callable

Java可以开启线程吗?

答：不能，调用Thread.start()方法实质上是调用的本地方法(native修饰)，底层是C++来实现的，为Java无法操作硬件。

```
public synchronized void start() {
    /**
     * This method is not invoked for the main method thread or "system"
     * group threads created/set up by the VM. Any new functionality added
     * to this method in the future may have to also be added to the VM.
     *
     * A zero status value corresponds to state "NEW".
```

```
         */
        if (threadStatus != 0)
            throw new IllegalThreadStateException();

        /* Notify the group that this thread is about to be started
         * so that it can be added to the group's list of threads
         * and the group's unstarted count can be decremented. */
        group.add(this);

        boolean started = false;
        try {
            start0();
            started = true;
        } finally {
            try {
                if (!started) {
                    group.threadStartFailed(this);
                }
            } catch (Throwable ignore) {
                /* do nothing. If start0 threw a Throwable then
                  it will be passed up the call stack */
            }
        }
    }
    //本地方法，底层C++，Java无法直接操作硬件
    private native void start0();
```

并发、并行

并发（多线程操作同一个资源）

- **CPU一核，模拟出来多条线程。快速交替**

并行（多个人一起干事）

- **CPU多核，多个线程可以同时执行。**

```
package cn.fyyice.juc;

public class Test1 {
    public static void main(String[] args) {
        //获取CPU核数
        System.out.println(Runtime.getRuntime().availableProcessors());
    }
}
```

并发编程的本质：__充分利用CPU的资源__

线程有几个状态

```
    public enum State {
        //  新生
        NEW,
```

```
        // 运行
        RUNNABLE,

        // 阻塞
        BLOCKED,

        // 等待
        WAITING,

        // 超时等待
        TIMED_WAITING,

        // 终止
        TERMINATED;
    }
```

wait与sleep的区别

1、来自不同的类

wait => Object

sleep => Thread

2、关于锁的释放

wait：会释放锁

sleep：睡觉了，抱着锁睡觉，不会释放！

3、使用的范围是不同的

wait：必须在同步代码块中

sleep：可以在任何地方睡

4、是否需要捕获异常

wait：不需要捕获异常

sleep：必须要捕获异常（超市等待）

在线程sleep过后，会让出CPU资源。目的是不让当前线程独自霸占该进程所获的CPU资源，以留一时间给其他线程执行的机会。所以不会占用cpu。时间到了会正常返回，线程处于就绪状态，然后参与pu调度，获取到cpu资源之后就可以运行。

3.Lock锁

在真正的多线程开发中（公司），线程就是一个单独的资源类，没有任何附属的操作。

这里以卖票的demo来进行讲解

传统synchronized解决

本质：排队

```
package cn.fyyice.juc;

public class SaleTicket {
    public static void main(String[] args) {
        //并发：多线程操作一个资源类
        Ticket t = new Ticket();

        //@FunctionalInterface函数式接口，jdk 1.8 lombda表达式 (参数)->{代码块}
        new Thread(()->{ for (int i = 0; i < 50; i++) t.sale();},"甲").start();
        new Thread(()->{ for (int i = 0; i < 50; i++) t.sale();},"乙").start();
        new Thread(()->{ for (int i = 0; i < 50; i++) t.sale();},"丙").start();
    }
}

// OOP资源类
class Ticket{
    private Integer ticket = 20;

    public synchronized void sale(){
        if (ticket > 0){
            System.out.println(Thread.currentThread().getName()+"卖出了编号:"+ticket--+"的票,
余:"+ticket);
        }
    }
}
```

Lock接口

```
Lock l = ...; l.lock(); try { // access the resource protected by this lock } finally { l.unlock(); }
```

当在不同范围内发生锁定和解锁时，必须注意确保在锁定时执行的所有代码由try-finally或try-catc保护，以确保在必要时释放锁定。

Lock**实现提供了使用**synchronized**方法和语句的附加功能，通过提供非阻塞尝试来获取锁（tryLock()），尝试获取可被中断的锁（lockInterruptibly()），以及尝试获取可以超时（tryLock(long, TieUnit)）。**

LOCK中声明的所有方法：

Modifier and Type	Method	D scription
void	lock()	获得锁。
void	lockInterruptibly()	获取 定，除非当前线程是 interrupted 。
Condition	newCondition()	返 一个新Condition绑定到该实例Lock实例。
boolean	tryLock()	只有在调用 才可以获得锁。
boolean	tryLock(long time, TimeUnit unit)	果在给定的等待时间内是空闲的，并且当前的线程尚未得到 interrupted，则获取该锁。

void unlock() 释放锁。

实现类：**

所有已知实现类：

1. **ReentrantLock** （可重入锁---常用）**
2. **ReentrantReadWriteLock.ReadLock** （读锁）**
3. **ReentrantReadWriteLock.WriteLock（写锁）**

ReentrantLock源码：

```
/**
     * Creates an instance of {@code ReentrantLock}.
     * This is equivalent to using {@code ReentrantLock(false)}.
     */
    public ReentrantLock() {
        sync = new NonfairSync();
    }

    /**
     * Creates an instance of {@code ReentrantLock} with the
     * given fairness policy.
     *
     * @param fair {@code true} if this lock should use a fair ordering policy
     */
    public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
    }
```

NonfairSync：非公平锁，可以插队

FairSync：公平锁，先来后到

```
package cn.fyyice.juc;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class SaleTicket2 {
    public static void main(String[] args) {
        //并发：多线程操作一个资源类
        Ticket2 t = new Ticket2();

        //@FunctionalInterface函数式接口，jdk 1.8 lombda表达式 (参数)->{代码块}
        new Thread(()->{ for (int i = 0; i < 30; i++) t.sale();},"A").start();
        new Thread(()->{ for (int i = 0; i < 30; i++) t.sale();},"B").start();
        new Thread(()->{ for (int i = 0; i < 30; i++) t.sale();},"C").start();
    }
}

// OOP资源类
class Ticket2{
```

```
    private Integer ticket = 20;
    Lock lock = new ReentrantLock();
    public void sale(){
        //加锁
        lock.lock();
        try {
            if (ticket > 0){
                System.out.println(Thread.currentThread().getName()+"卖出了编号:"+ticket--+"的票
剩余:"+ticket);
            }
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //解锁
            lock.unlock();
        }
    }
}
```

synchronized 与 Lock 区别

1. **Synchronized 是内置的Java关键字， Lock 是一个Java类**
2. **Synchronized 无法判断获取锁的状态，Lock 可以判断是否获取到了锁**
3. **Synchronized 会自动释放锁， Lock 必须要手动释放锁！如果不释放锁，** **死锁**
4. **Synchronized 线程1（获得锁，但发生了阻塞），线程2会一直等待；Lock锁就不一定会等待下（tryLock方法，尝试获取锁）**
5. **Synchronized 可重入锁，不可以中断的，非公平；Lock 可重入锁。可以判断锁，自己设定公平非公平**
6. **Synchronized 适合锁少量的代码同步问题；Lock 适合锁大量的同步代码**

锁是什么，如何判断锁的是谁

4.生产者和消费者问题

```
线程之间的通信问题：生产者和消费者问题。    等待换新，通知唤醒
线程交替执行  A   B 操作统一变量    num = 0
A   num+1
B   num-1
```

面试的：

单例模式、排序算法、生产者和消费者问题、死锁

```
package cn.fyyice.juc.pc;

public class A {
    public static void main(String[] args) {
        Data data = new Data();
        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
```

```
                data.increment();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    },"A").start();
    new Thread(()->{
        for (int i = 0; i < 10; i++) {
            try {
                data.decrement();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    },"B").start();
  }
}

//判断等待、业务、通知
class Data{
  private int number = 0;

  public synchronized void increment() throws InterruptedException {
    if (number != 0){
      //等待
      this.wait();
    }
    number ++;
    System.out.println(Thread.currentThread().getName()+"-------->"+number);
    //通知 业务完成
    this.notifyAll();
  }

  public synchronized void decrement() throws InterruptedException {
    if (number == 0){
      //等待
      this.wait();
    }
    number --;
    System.out.println(Thread.currentThread().getName()+"-------->"+number);
    //通知 业务完成
    this.notifyAll();
  }
}
```

问题存在，A B C D四个线程。虚假唤醒！

新加入线程后，是否安全？

** **不安全。用if判断的话，唤醒后线程会从wait之后的代码开始运行，但是不会重新判断if条件直接继续运行if代码块之后的代码，而如果使用while的话，也会从wait之后的代码运行，但是唤醒会重新判断循环条件，如果不成立再执行while代码块之后的代码块，成立的话继续wait。**

官方文档解释：

线程也可以唤醒，而不会被通知，中断或超时，即所谓的**=**虚假唤醒**=** 。 虽然这在实践中很少发生，但应用程序必须通过测试应该使线程被唤醒的条件来防范，并且如果条件不满足则继续等待。句话说，等待应该总是出现在循环中，就像这样：**

```
  synchronized (obj) {
        while (<condition does not hold>)
            obj.wait(timeout);
        ... // Perform action appropriate to condition
    }
```

解决方法：

****if 改为 white！****防止虚假唤醒**

```
package cn.fyyice.juc.pc;

public class A {
    public static void main(String[] args) {
        Data data = new Data();
        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"A").start();
        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"B").start();
        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"C").start();
        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
```

```
        },"D").start();

    }
}

//判断等待、业务、通知
class Data{
    private int number = 0;

    public synchronized void increment() throws InterruptedException {
        while (number != 0){
            //等待
            this.wait();
        }
        number ++;
        System.out.println(Thread.currentThread().getName()+"-------->"+number);
        //通知 业务完成
        this.notifyAll();
    }

    public synchronized void decrement() throws InterruptedException {
        while (number == 0){
            //等待
            this.wait();
        }
        number --;
        System.out.println(Thread.currentThread().getName()+"-------->"+number);
        //通知 业务完成
        this.notifyAll();
    }
}
```

JUC 版本的生产者和消费者问题

传统的：Synchronized　wait　notifyAll/notify

新版：Lock　await　signal

官方文档：

public interface Condition

Condition**因素出**Object**监视器方法（ wait ， notify和notifyAll ）成不同的对象，以得到具有多等待集的每个对象，通过将它们与使用任意的组合的效果**Lock**个**实现。 **Lock**替换**synchronized*方法和语句的使用， **Condition**取代了对象监视器方法的使用。**

条件（也称为条件队列或条件变量 **）**
一个线程暂停执行（“等待”）提供了一种方法，直到另一个线程通知某些状态现在可能为真。 因为问此共享状态信息发生在不同的线程中，所以它必须被保护，因此某种形式的锁与该条件相关联。 等条件的关键属性是它**原子地**释放相关的锁并挂起当前线程，就像**Object.wait** 。**

```
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(); }
  }
}
```

代码实现：

```
package cn.fyyice.juc.pc;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class B {
    public static void main(String[] args) {
        Data2 data = new Data2();
        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"A1").start();
        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
            }
        }
    },"B1").start();
    new Thread(()->{
        for (int i = 0; i < 10; i++) {
            try {
                data.increment();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    },"C1").start();
    new Thread(()->{
        for (int i = 0; i < 10; i++) {
            try {
                data.decrement();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    },"D1").start();

  }
}

//判断等待、业务、通知
class Data2{
    private int number = 0;

    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();

    public void increment() throws InterruptedException {
        lock.lock();
        try {
            while (number != 0){
                //等待
                condition.await();
            }
            number ++;
            System.out.println(Thread.currentThread().getName()+"-------->"+number);
            //通知 业务完成
            condition.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    public void decrement() throws InterruptedException {
        lock.lock();
        try {
            while (number == 0){
                //等待
```

```
                condition.await();
            }
            number --;
            System.out.println(Thread.currentThread().getName()+"-------->"+number);
            //通知 业务完成
            condition.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

任何一个新的技术绝对不仅仅只是覆盖了原来的技术

Condition的优势

精准的通知和操作线程

1. **设定标志位**

```
Condition condition1 = lock.newCondition();
Condition condition2 = lock.newCondition();
Condition condition3 = lock.newCondition();

//    通过对监控的Condition进行signal方法进行唤醒
```

2. **设定多个监视器Condition**

通过while循环对标志位进行判断。满足条件，则对指定的监视器执行singal操作，从而达到精确唤醒

5.8锁现象

8锁，就是关于锁的八个问题

```
package cn.fyyice.juc.lock8;

import java.util.concurrent.TimeUnit;

public class Test {
    public static void main(String[] args) {
        Phone phone = new Phone();

        new Thread(()->{
            phone.send();

        },"A").start();

        try {
            System.out.println("我要等待3s");
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
        new Thread(()->{
            phone.call();
        },"B").start();
    }
}

class Phone {
    // synchronized锁的对象是方法的调用者!
    // 这里两个方法用的是同一个锁，谁先拿到谁先执行!
    public synchronized void send(){
        try {
            System.out.println("我是phone中的等待");
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("发信息");
    }
    public synchronized void call(){
        System.out.println("打电话");
    }
}

package cn.fyyice.juc.lock8;

import java.util.concurrent.TimeUnit;

public class Test2 {
    public static void main(String[] args) {
        Phone2 phone = new Phone2();
        Phone2 phone2 = new Phone2();

        new Thread(()->{
            phone.send();

        },"A2").start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        new Thread(()->{
            phone2.call();
        },"B2").start();
    }
}

class Phone2 {
    // synchronized锁的对象是方法的调用者!
    // 这里两个方法用的是同一个锁，谁先拿到谁先执行!
    public synchronized void send(){
        try {
```

```
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+"-->发信息");
    }
    public synchronized void call(){
        System.out.println(Thread.currentThread().getName()+"-->打电话");
    }

    public void hello(){
        System.out.println("hello");
    }
}

package cn.fyyice.juc.lock8;

import java.util.concurrent.TimeUnit;

public class Test3 {
    public static void main(String[] args) {
        Phone3 phone = new Phone3();

        new Thread(() -> {
            phone.send();

        }, "A").start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        new Thread(() -> {
            phone.call();
        },"B").start();

    }
}

class Phone3 {
    // synchronized锁的对象是方法的调用者!
    // 这里两个方法用的是同一个锁，谁先拿到谁先执行!
    public static synchronized void send(){
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+"-->发信息");
    }
    public static synchronized void call(){
        System.out.println(Thread.currentThread().getName()+"-->打电话");
```

```
    }
}

package cn.fyyice.juc.lock8;

import java.util.concurrent.TimeUnit;

public class Test4 {
    public static void main(String[] args) {
        Phone4 phone = new Phone4();

        new Thread(() -> {
            phone.send();

        }, "A").start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        new Thread(() -> {
            phone.call();
        },"B").start();

    }
}

class Phone4 {
    // synchronized锁的对象是方法的调用者!
    // 这里两个方法用的是同一个锁，谁先拿到谁先执行!
    public static synchronized void send(){
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+"-->发信息");
    }
    public synchronized void call(){
        System.out.println(Thread.currentThread().getName()+"-->打电话");
    }
}
```

顺序：发信息在前，打电话在后

1. **标准情况下，两个线程先打印 发信息还是打电话?**
2. **send()方法延迟4s，两个线程先打印发信息 还是打电话?**
3. **在加入普通方法hello后，两个线程先打印发信息 还是hello?**
4. **两个对象，2个同步方法， 发信息还是 打电话?**
5. **增加两个静态的同步方法，只有一个对象，先打印 发短信还是打电话?**

6. **两个对象，增加两个静态的同步方法，只有一个对象，先打印 发短信还是打电话?**

7. **static方法的send，同步方法call，只有一个对象，先打印 发短信还是打电话?**

8. **static方法的send，同步方法call，只有两个对象，先打印 发短信还是打电话?**

回复:

问题1,2： 按照顺序执行，原因是因为这两个方法都是加了锁的，而synchronized是修饰在方法上面，所以锁的是这个方法的调用对象，在这里两个方法都是同一个调用对象，所以谁先拿到锁谁先执行

问题3： 不是同步的方法，不考虑锁的因素。这里其实还是应该先执行发信息，只不过因为sleep了而hello不需要去等待这个锁，所以先执行的hello。
问题4:
问题5： 先打印发短信，再打印打电话。因为在加入static修饰后，变成了静态方法，在类初始化的候就会执行，相当于类一加载就有了！锁的是Class对象（模板）
问题6： 先打印发短信，再打印打电话。因为锁的是Class对象，类模板是同一个，所以谁先拿到锁先执行，这里就是顺序执行。
问题7： 先打印打电话，再打印发信息。因为锁的对象不一样，发信息方法锁的对象是Class对象，打电话方法锁的是类调用对象。锁不一样，不需要去等待锁的释放。
问题8： 同上。

小结:

- **new this 具体的一个类对象**
- **static Class 唯一的一个模板**
- **在主线程与子线程中的代码执行顺序问题**

1. **当主线程代码段在(一个或多个)子线程中间时，首先执行的还是主线程代码段。因为刚开始时，有主线程在使用CPU的执行权，因为其他两个线程还没有被创建，这时主线程的代码就自上而下的去行。在线程都创建完成后，此时就存在多个线程了，而线程的执行需要抢到CPU资源去执行，所以在续就是谁先抢到CPU资源谁先执行了。**

6.集合类不安全

List不安全

```
package cn.fyyice.juc.unfairy;

import java.sql.Connection;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.UUID;
import java.util.concurrent.CopyOnWriteArrayList;

//java.util.ConcurrentModificationException 并发修改异常
public class ListTest1 {
    public static void main(String[] args) {
        /**
         * 并发下，list是不安全的
         * 解决方案:
         * 1.List<String> list = Vector()<>集合类
```

```
         *  2.List<String> list = Collections.synchronizedList(new ArrayList<>())
         *  3.List<String> list = new CopyOnWriteArrayList<>();
         *      CopyOnWrite 写入时复制    COW 优化策略
         *      多个线程调用的时候，list读取是固定的，但是在写入的时候可能会被后面的线程给覆盖掉
         *      在写入的时候避免覆盖，造成数据问题
         *      读写分离
         */
        List<String> list = new CopyOnWriteArrayList<>();

        for (int i = 1; i <= 10; i++) {
            new Thread(()->{
                list.add(UUID.randomUUID().toString().substring(0,6));
                System.out.println(list);
            },String.valueOf(i)).start();

        }
    }
}
```

Set不安全

```
package cn.fyyice.juc.unfairy;

import java.util.HashSet;
import java.util.Set;
import java.util.UUID;

public class SetList {
    public static void main(String[] args) {
        //不安全
        Set<Object> set = new HashSet<>();
        /**
         * 解决方案:
         *  Collections.synchronizedSet(new HashSet<>());
         *  new CopyOnWriteArraySet<>()
         */
        for (int i = 0; i < 30; i++) {
            new Thread(()->{
                set.add(UUID.randomUUID().toString().substring(0,6));
            },String.valueOf(i)).start();
        }
    }
}
```

hashSet底层是什么?

```
public HashSet() {
    map = new HashMap<>();
}
// add  set 本质就是 map key 是无法重复的!
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

```
private static final Object PRESENT = new Object();   //不变的值
```

Map 不安全

```
<pre class="md-fences mock-cm md-end-block md-fences-with-lineno" spellcheck="false" l
ng="java" cid="n217" mdtype="fences"><br/>    /**<br/>     * The default initial capacity -
UST be a power of two.<br/>     */<br/>    static final int DEFAULT_INITIAL_CAPACITY = 1 <<
4; // aka 16 默认容量<br/><br/>    /**<br/>     * The maximum capacity, used if a higher value
is implicitly specified<br/>     * by either of the constructors with arguments.<br/>     * MUST
be a power of two <= 1<<30.<br/>     */<br/>    static final int MAXIMUM_CAPACITY = 1 <<
30;<br/><br/>    /**<br/>     * The load factor used when none specified in constructor.<br/>
    */<br/>    static final float DEFAULT_LOAD_FACTOR = 0.75f;    //加载因子</pre>
```

```
package cn.fyyice.juc.unfairy;

import java.util.HashMap;
import java.util.Map;
import java.util.UUID;

public class MapTest {
    public static void main(String[] args) {
        //map是这样用的？默认等价于什么?
        Map<String, String> map = new HashMap<>();
        //加载因子，初始化容量
        /**
         *  new ConcurrentHashMap<>();
         *  Collections.synchronizedMap(new HashMap<>());
         */
        for (int i = 1; i < 20; i++) {
            new Thread(()->{
                map.put(Thread.currentThread().getName(), UUID.randomUUID().toString().substrin
(0,5));
                System.out.println(map);
            },String.valueOf(i)).start();
        }
    }
}
```

7.Callable

```
@FunctionalInterface
public interface Callable<V>
```

返回结果并可能引发异常的任务。 实现者定义一个没有参数的单一方法，称为call** 。 **

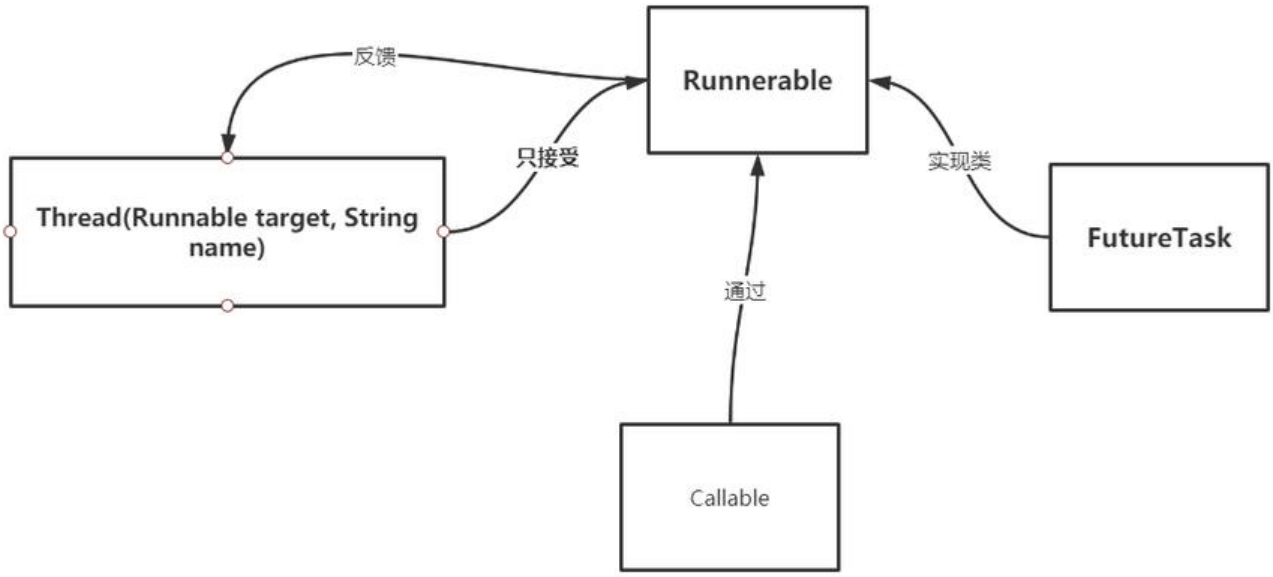
Callable**接口类似于**Runnable** ，因为它们都是为其实例可能由另一个线程执行的类设计的。 然而A **Runnable**不返回结果，也不能抛出被检查的异常。**

该Executors**类包含的实用方法，从其他普通形式转换为**Callable**类。**

1. **有返回值**
2. **可以跑出异常**
3. **方法不同，run()/call()**

4. **通过FutureTask来实现**

- **一个FutureTask可以用来包装Callable或Runnable对象。因为FutureTask实现Runnable，一Futur Task可以提交执行Executor**。**** **
-




```
/**
 * Creates a {@code FutureTask} that will, upon running, execute the
 * given {@code Callable}.
 *
 * @param  callable the callable task
 * @throws NullPointerException if the callable is null
 */
public FutureTask(Callable<V> callable) {
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;       // ensure visibility of callable
}

/**
 * Creates a {@code FutureTask} that will, upon running, execute the
 * given {@code Runnable}, and arrange that {@code get} will return the
 * given result on successful completion.
 *
 * @param runnable the runnable task
 * @param result the result to return on successful completion. If
 * you don't need a particular result, consider using
 * constructions of the form:
 * {@code Future<?> f = new FutureTask<Void>(runnable, null)}
 * @throws NullPointerException if the runnable is null
 */
public FutureTask(Runnable runnable, V result) {
    this.callable = Executors.callable(runnable, result);
    this.state = NEW;       // ensure visibility of callable
```

```
    }
```

代码实现

```
package cn.fyyice.juc.callable;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class CallableTest {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        // new Thread(new Runnable() {}).start();
        // new Thread(new FutureTask<V>()).start());

        MyThread thread = new MyThread();
        //适配类
        FutureTask<String> task = new FutureTask<>(thread);
        new Thread(task,"A").start();
        new Thread(task,"C").start();
        new Thread(task,"D").start();
        //这个get方法可能会出现阻塞！把他放到最后、或者使用异步通信来处理
        System.out.println(task.get());
    }
}

class MyThread implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println("成功进入Callable方法");
        return Thread.currentThread().getName()+"--->hello";
    }
}
```

输出结果：

```
成功进入Callable方法
A--->hello

Process finished with exit code 0
```

注：

- **有缓存**
- **结果可能需要等待，会阻塞。**

8.常用辅助类

-

CountDownLatch(减法计数器)

****官方解释：****CountDownLatch是一个同步的辅助类，允许一个或多个线程，等待其他一组线完成操作，再继续执行。**

可以理解为倒计时锁　　**。就比如玩LOL匹配/排位。一场对局的开始必须等待所有人加载10%才能开始，所以就算你是外星人，**是小霸王，你也得等着。**

demo 01

```
package cn.fyyice.juc.utils;

import java.util.concurrent.CountDownLatch;

//计数器
public class CountDownLatchDemo {
    public static void main(String[] args) throws InterruptedException {
        //总数是10，必须要执行任务的时候才使用
        CountDownLatch count = new CountDownLatch(10);
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                System.out.println(Thread.currentThread().getName()+"溜了");
                count.countDown();
            },String.valueOf(i)).start();
        }
        //等待计数器归0，再向下执行
        count.await();
        System.out.println("遛完了，关门");
    }
}
```

demo 02

```
package cn.fyyice;

import java.util.Random;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;

public class CountDown {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(5);
```

```
        for (int i = 0; i < 5; i++) {
            new Thread(()->{
                try{
                    int time  = new Random().nextInt(4)+1;
                    System.out.println(Thread.currentThread().getName()+"--->start loading,need "+t
me+" s");
                    TimeUnit.SECONDS.sleep(time);
                    System.out.println(Thread.currentThread().getName()+"--->loading finally!");
                }catch (Exception e){
                    e.printStackTrace();
                }finally {
                    countDownLatch.countDown();
                }
            },"Player " + i).start();
        }

        try {
            System.out.println("主线程开始执行...");
            countDownLatch.await();
            System.out.println("主线程执行结束...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

原理：

=**countDown();**=** **** **** 数量减1**

=**await()**=** ****等待计数器归零，然后再向下执行**

当计数器归零时，=**await()**=**方法就会被唤醒，继续执行。**

-

CyclicBarrier（加法计数器）

****官方解释：****CyclicBarrier是一个同步的辅助类，允许一组线程相互之间等待，达到一个共同，再继续执行。**

可以把CyclicBarrier看成是个障碍，所有的线程必须到齐后才能一起通过这个障碍。常用于多线程分计算。比如一个大型的任务，常常需要分配好多子任务去执行，只有当所有子任务都执行完成时候，能执行主任务；通俗点可以理解为一个公会副本奖励，需要公会所有人成员都完成至少在线一小时才发布奖励。

```
// 计数，并执行一个线程
public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}
```

```
package cn.fyyice.juc.utils;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {
    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(7,()->{
            System.out.println("太下饭了，饱了");
        });

        for (int i = 1; i <= 7; i++) {
            //因为i是临时变量
            final int temp = i;
            new Thread(()->{
                System.out.println(Thread.currentThread().getName()+"-->第"+temp+"次吃饭");
                try {
                    //  每执行一次await() 内部会执行一次+1
                    cyclicBarrier.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            },String.valueOf(i)).start();

        }
    }
}
```

-

Semaphore

```
 public Semaphore(int permits) {
        sync = new NonfairSync(permits);
}
```

计数信号量。从概念上讲，一个信号量维护一组允许。每个 **acquire()**块如果必要的许可证前，可用的，然后把它。每个 **release()**添加许可，潜在收购方释放阻塞。然而，不使用实际允许的象； **Semaphore**只是计数的数量和相应的行为。 **

信号量通常是用来限制线程的数量比可以访问一些（物理或逻辑）资源。例如，这里是一个类使用一信号量来控制访问的项目池：

```
package cn.fyyice.juc.utils;

import java.util.Random;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class SemaphoreDemo {
    public static void main(String[] args) {
        //限流
```

```
        Semaphore semaphore = new Semaphore(4);
        for (int i = 0; i < 10; i++) {
            new Thread(()->{
                try {
                    //得到
                    semaphore.acquire();
                    int time  = new Random().nextInt(4)+1;
                    System.out.println(Thread.currentThread().getName()+"得到了车位,并且要停"+tim
+"s");
                    TimeUnit.SECONDS.sleep(time);
                    System.out.println(Thread.currentThread().getName()+"开走了，空出一个车位");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    // 释放
                    semaphore.release();
                }
            },String.valueOf(i)).start();
        }

    }
}
```

原理：

=**semaphore.acquire();**=**　****获得，假设已经满了，那么就会等待，直到被释放位置！**

=**semaphore.release();**=**　****释放，会将当前的信号量释放+1，然后唤醒等待的线程！**

作用：多个共享资源互斥作用！并发限流，控制最大的线程数！

9.读写锁

ReadWriteLock

所有已知实现类：

ReentrantReadWriteLock

---

public interface ReadWriteLock

一个 **ReadWriteLock保持一对一联系 **locks**，只读操作和书写。的 **read lock**可能被多个程同时举行的读者，只要没有作家。 **write lock**是独家的。 **

读的时候可以被多线程同时读，写的时候只能由一个线程去写。

● 读写锁允许访问共享数据比用互斥锁允许更多的并发级别。它利用的事实，而只有一个单一的线程一个时间（一个作家线程）可以修改共享数据，在许多情况下，任何数量的线程可以同时读取数据（此读者线程）。在理论上，并发的读写锁的使用允许的增加将导致在一个互斥锁的使用性能的改进。实践中，这种增加的并发性只会完全实现一个多处理器，然后，只有当共享数据的访问模式是合适的。

package cn.fyyice.juc.raw;

```
import java.util.HashMap;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * 又称为   独占锁（写锁）    一次只能被一个线程占有
 *          共享锁（读锁）    多个线程可以同时占有
 * ReadWriteLock
 * 读-读  可以共存
 * 读-写  不能共存
 * 写-写  不能共存
 */
public class ReadWriteLockDemo {
    public static void main(String[] args) {
//        MyCache myCache = new MyCache();
        MyLockCache myCache = new MyLockCache();
        for (int i = 1; i <= 5; i++) {
            final int temp = i;
            new Thread(()->{
                myCache.put(temp+"", UUID.randomUUID().toString().substring(0,5));
            },String.valueOf(i)).start();
        }

        for (int i = 1; i <= 5; i++) {
            final int temp = i;
            new Thread(()->{
                myCache.get(temp+"");
            },String.valueOf(i)).start();
        }
    }
}

class MyLockCache{

    private volatile Map<String,Object> map = new HashMap<>();
    //读写锁：更加细粒度的控制
    private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

    //在写操作的时候希望同一时间，只有一个线程进行写操作，避免覆盖
    public void put(String key,Object value){
        readWriteLock.writeLock().lock();
        try {
            System.out.println(Thread.currentThread().getName()+"写入："+value);
            map.put(key,value);
            System.out.println(Thread.currentThread().getName()+"写入完成");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            readWriteLock.writeLock().unlock();
        }
    }
```

```
    //读的时候加锁是为了防止  在读的时候发生写操作
    public void get(String key){
        readWriteLock.readLock().lock();
        try {
            System.out.println(Thread.currentThread().getName()+"读取：  "+key);
            Object obj = map.get(key);
            System.out.println("----------"+Thread.currentThread().getName()+"读取完成---------");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            readWriteLock.readLock().unlock();
        }
    }

}

/**
 * 自定义缓存
 */
class MyCache{

    private volatile Map<String,Object> map = new HashMap<>();

    public void put(String key,Object value){
        System.out.println(Thread.currentThread().getName()+"写入：  "+value);
        map.put(key,value);
        System.out.println("----------"+Thread.currentThread().getName()+"写入完成---------");
    }

    public void get(String key){
        System.out.println(Thread.currentThread().getName()+"读取：  "+key);
        Object obj = map.get(key);
        System.out.println("----------"+Thread.currentThread().getName()+"读取完成---------");
    }
}
```

10.阻塞队列

- **阻塞队列（FIFO）**
  - **写入：如果队列满了，就必须阻塞等待（取出）**
  - **取：如果队列是空的，必须阻塞等待生产（写入）**

InInterface BlockingQueue <E> ****

-
  - **参数类型 ****

E -元素举行此集合中的类型 **

- **All Superinterfaces: ****

Collection， Iterable， Queue **

- All Known Subinterfaces: **

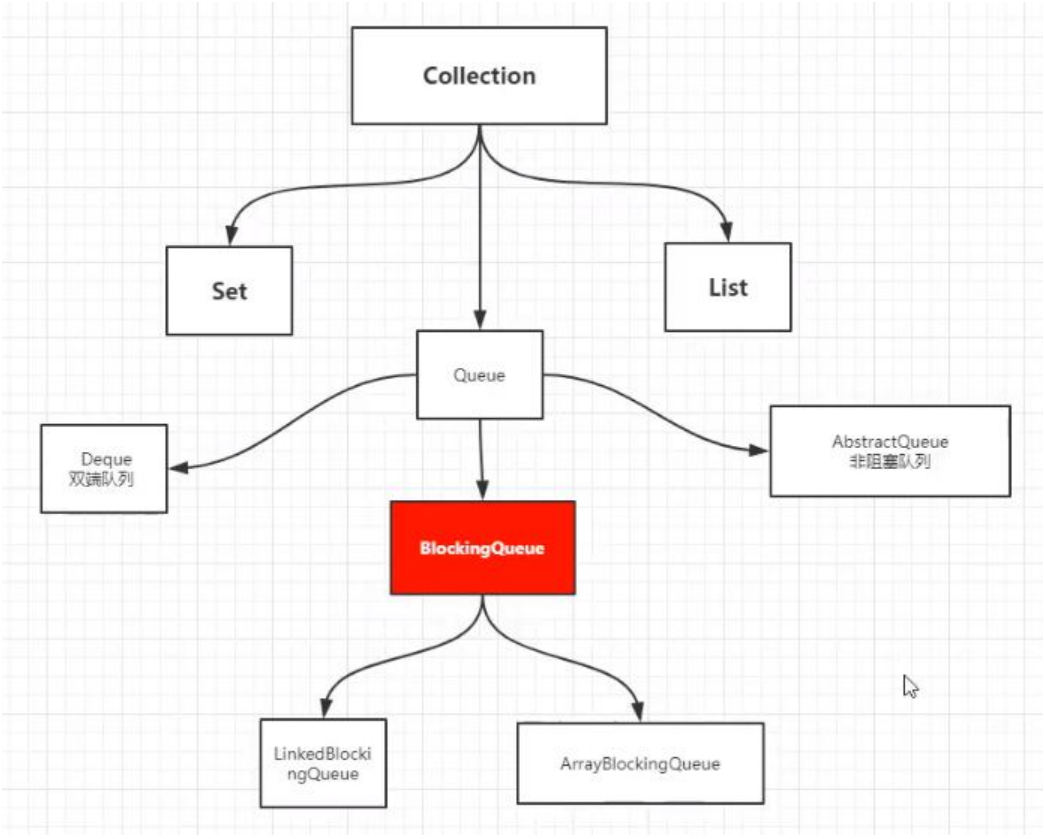
BlockingDeque， TransferQueue **

- 所有已知实现类： **

ArrayBlockingQueue， DelayQueue， LinkedBlockingDeque， LinkedBlockingQueue， LnkedTransferQueue， PriorityBlockingQueue， SynchronousQueue **

什么情况下会使用阻塞队列：

- **多线程并发处理**
- **线程池**



四组API

```
BlockingQueue<Object> blockingQueue = new ArrayBlockingQueue<>(3);
```

方式	抛出异常	有返回值	阻塞等待	时等待
添加	add()	offer()	put()	ffer(,,)
移除	remove()	poll()	take()	oll(,)
检测队首元素	element()	peek()	-	

原文链接：并发编程进阶，深入浅出理解 JUC

SynchronousQueue 同步队列

没有容量

进去一个元素，必须等待元素取出来过后，才能put进去

```
package cn.fyyice.juc.bq;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.TimeUnit;

public class SynchronousQueueDemo {
    public static void main(String[] args) {
        // 同步队列     和其他BlockingQueue不一样，synchronousQueue不存储元素
        BlockingQueue<String> synchronousQueue = new SynchronousQueue<>();

        new Thread(()->{
            try {
                System.out.println(Thread.currentThread().getName()+"--->put1");
                synchronousQueue.put("1");
                System.out.println(Thread.currentThread().getName()+"--->put2");
                synchronousQueue.put("2");
                System.out.println(Thread.currentThread().getName()+"--->put3");
                synchronousQueue.put("3");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"A").start();

        new Thread(()->{
            try {
                TimeUnit.SECONDS.sleep(2);
                System.out.println(Thread.currentThread().getName()+"--->get1");
                synchronousQueue.take();
                TimeUnit.SECONDS.sleep(2);
                System.out.println(Thread.currentThread().getName()+"--->get2");
                synchronousQueue.take();
                TimeUnit.SECONDS.sleep(2);
                System.out.println(Thread.currentThread().getName()+"--->get3");
                synchronousQueue.take();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"B").start();
    }
}
```

11.线程池

线程池必会：三大方法、七大参数、四种拒绝策略

池化技术

程序的运行，本质：占用系统资源。我们需要做的是优化资源的使用！

常见：线程池、连接池、对象池

池化技术：事先准备好一些资源，有人要用就直接来这里拿，用完过后回收。**

线程池的好处：

1. **降低资源的消耗**
2. **提高相应的速度**
3. **方便管理**

=线程复用、可以控制最大并发数、管理线程=

【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

1）FixedThreadPool 和 SingleThreadPool:

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。

三大方法

1. **Executors.newSingleThreadExecutor();**
2. **Executors.newFixedThreadPool(线程数量);**
3. **Executors.newCachedThreadPool();**

```
package cn.fyyice.juc.pool;

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

//Executor  工具类，3大方法
public class Demo1 {
    public static void main(String[] args) {
        //创建单个线程池
        ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
        //创建一个固定大小的线程池
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(6);
        //创建一个可伸缩的线程池
        ExecutorService cachedThreadPool = Executors.newCachedThreadPool();


        try {
            for (int i = 0; i < 10; i++) {
                //使用了线程池之后，使用线程池来创建线程
```

```
            //始终只有一个线程执行
//              singleThreadExecutor.execute(()->{
//                  System.out.println(Thread.currentThread().getName()+"----> start");
//              });

            //固定线程数量执行
//              fixedThreadPool.execute(()->{
//                  System.out.println(Thread.currentThread().getName()+"----> start");
//              });

            //根据cpu支持线程数量执行
            cachedThreadPool.execute(()->{
                System.out.println(Thread.currentThread().getName()+"----> start");
            });

        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        //线程池用完，程序结束，关闭线程池
        //singleThreadExecutor.shutdown();
        //fixedThreadPool.shutdown();
        cachedThreadPool.shutdown();
    }
  }
}
```

七大参数

源码分析

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

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

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

//本质：开启线程都是调用ThreadPoolExecutor
public ThreadPoolExecutor(int corePoolSize,    // 核心线程池大小
                          int maximumPoolSize,    // 最大核心线程池大小
```

```
                          long keepAliveTime,    // 超时了，没有人调用就会有释放
                          TimeUnit unit,         //超时单位
                          BlockingQueue<Runnable> workQueue,    // 阻塞队列
                          ThreadFactory threadFactory,           // 线程工厂 创建线程的 ， 一般不用动
                          RejectedExecutionHandler handler) {    // 拒绝策略
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
        null :
    AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

手动创建线程池

```
package cn.fyyice.juc.pool;

import java.util.concurrent.*;

public class Demo2 {
    public static void main(String[] args) {
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2, 6, 3, TimeUnit.SE
ONDS,
                new LinkedBlockingQueue<>(3),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());
        System.out.println("当前CPU核数:"+Runtime.getRuntime().availableProcessors());
        try {
            //最大承载 maxImumPoolSize + capacity
            for (int i = 1; i <= 8; i++) {
                threadPoolExecutor.execute(()->{
                    System.out.println(Thread.currentThread().getName()+"---> deal");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPoolExecutor.shutdown();
        }
    }
}
```

四种拒绝策略

1.

new ThreadPoolExecutor.AbortPolicy()

如果超出了最大处理量（maxImumPoolSize + capacity），则不进行处理，抛出异常。

2.

new ThreadPoolExecutor.CallerRunsPolicy()

哪来的去哪里。

3.

new ThreadPoolExecutor.DiscardOldestPolicy(由调用线程处理该任务)

（放弃最旧的策略）队列满了，尝试去和最早的竞争。也不会抛出异常

4.

new ThreadPoolExecutor.DiscardPolicy()

队列满了，直接丢掉任务，不会抛出异常。（解决不了问题，就解决提出问题的人）

小结

最大线程倒地如何定义？（调优）

1. **CPU密集型**
   - **看电脑配置，几核就是定义几，可以保持CPU型效率最高**
2. **IO 密集**
   - **判断你的程序中，十分耗IO的线程，然后设置大于这个数的线程**

12.四大函数式接口

作用：简化底层编程模型

- **lambda表达式**
- **链式编程**
- **函数式接口**
- **Stream流式计算**

函数式接口

只有只一个方法的接口

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
//简化编程模型，在新版本的框架底层中大量应用
//foreach(消费者类型的函数式接口)
```

四大函数式接口

- **Consumer**
- **Function**
- **Predicate**
- **Supplier**

代码测试

1.Function

```
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);

}
```

```
package cn.fyyice.juc.function;

import java.util.function.Function;

public class FunctionDemo {
    public static void main(String[] args) {
//          可用作工具类
//        Function<String, String> f = new Function<String, String>() {
//            @Override
//            public String apply(String str) {
//                return str;
//            }
//        };
        Function<String, String> f = (str)-> {return str;};
        System.out.println(f.apply("hello world"));
    }
}
```

2.Predicate

```
package cn.fyyice.juc.function;

import java.util.function.Predicate;

/**
 * 断定性接口，返回值只能是boolean
 */
public class PredicateDemo {
    public static void main(String[] args) {
//        Predicate<String> predicate = new Predicate<String>() {
//            @Override
```

```
//            public boolean test(String str) {
//                return str.isEmpty();
//            }
//        };
        Predicate<String> p = str->{return str.isEmpty();};
        System.out.println(p.test("123"));

    }
}
```

3.Consumer

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
```

```
package cn.fyyice.juc.function;

import java.util.function.Consumer;

/**
 * 消费性接口：只有输入，没有返回值
 */
public class ConsumerDemo {
    public static void main(String[] args) {

        Consumer<String> consumer = str->{
            System.out.println(str);
        };
        consumer.accept("hello");
    }
}
```

4.Supplier

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

```
package cn.fyyice.juc.function;

import java.util.function.Supplier;
```

```
/**
 * 供给型接口：没有参数，只有返回值
 */
public class SupplierDemo {
    public static void main(String[] args) {
        Supplier supplier = ()->{return 1024;};
        System.out.println(supplier.get());
    }
}
```

13.Stream流式计算

什么是Stream流式计算

大数据：存储 + 计算

集合、MySQL本质就是存储东西的；计算都是交给流来操作的!

Interface Stream<T>****

- **参数类型 ****

T -流中的元素的类型 **

**All Superinterfaces: **

AutoCloseable， **BaseStream**<t， **Stream**< T > > **

```
package cn.fyyice.juc.stream;

import java.util.Arrays;
import java.util.List;

public class StreamTest {
    public static void main(String[] args) {
        /**
         * 从以下用户数据中筛选
         * 1.id为偶数
         * 2.年龄大于等于20
         * 3.姓名转为大写
         * 4.输出第一条用户记录
         */
        User user1 = new User(1,18,"a");
        User user2 = new User(2,29,"b");
        User user3 = new User(3,20,"c");
        User user4 = new User(4,19,"d");
        User user5 = new User(5,23,"e");

        List<User> userList = Arrays.asList(user1,user2,user3,user4,user5);
        userList.stream()
                .filter(user -> {return user.getId() %2==0;})
                .filter(user -> {return user.getAge() >= 20;})
                .map((user ->{return user.getUsername().toUpperCase();}))
                .limit(1)
```

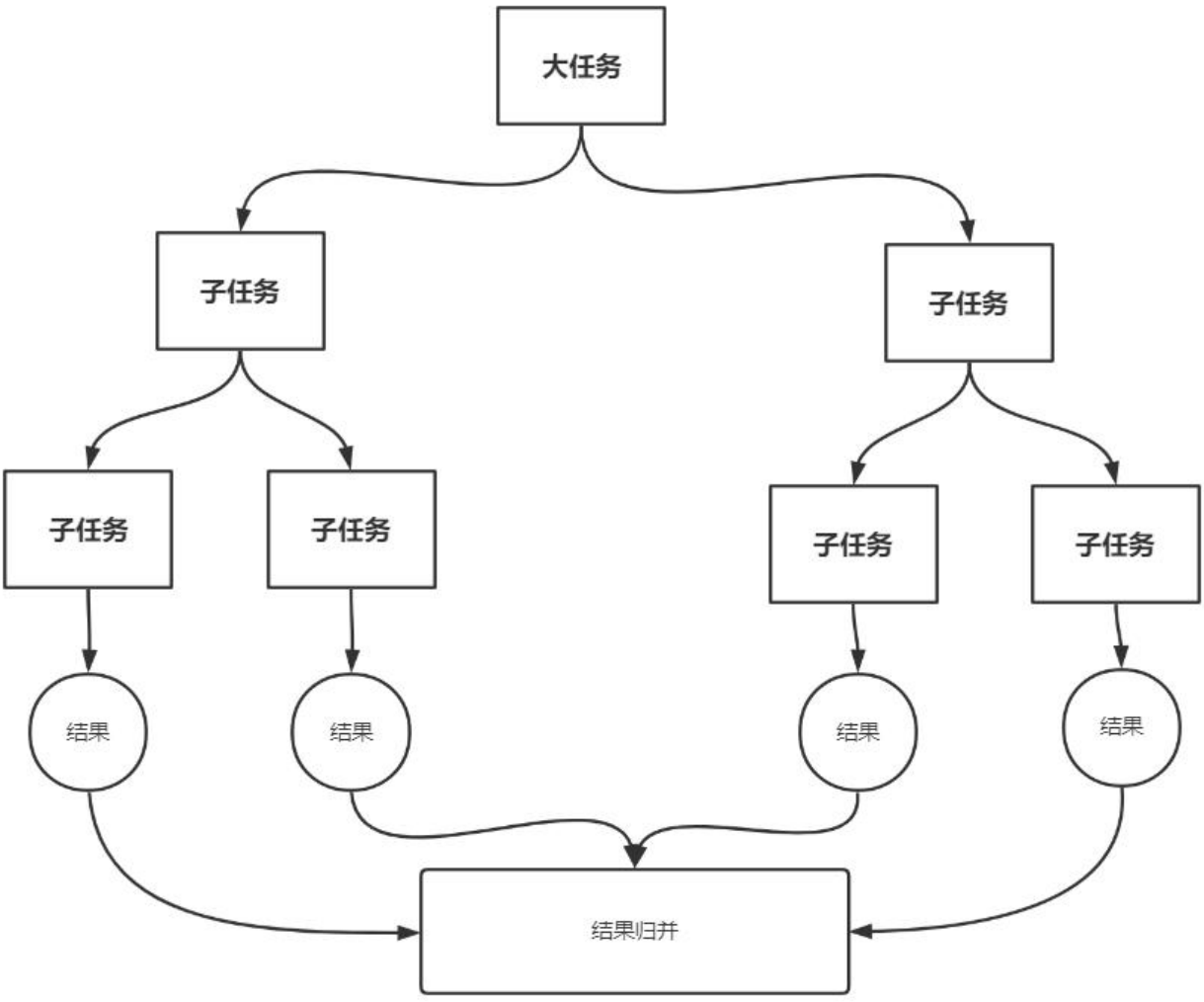
```
                .forEach(System.out::println);
    }
}
```

14.ForkJoin

什么是ForkJoin

ForkJoin在JDK1.7，并行执行任务！提高效率，大数据量

大数据：Map Reduce（把大任务拆分成小任务）



FockJoin特点：工作窃取

这个里面维护的都是双端队列

假设 2个线程A和B。

** **A线程有4个任务，当前执行到第2个任务，B线程也有4个任务，当前已执行完。这时候B线程会一直等待A线程执行完，而是会转过去执行A线程未执行的任务，这就是工作窃取。**

FockJoin 测试

执行

execute(ForkJoinTask<?> task)

Class ForkJoinTask<V>**

- **java.lang.Object**** **
- **java.util.concurrent.ForkJoinTask**<V>**** **
- **All Implemented Interfaces: **

Serializable， **Future** **

**已知直接子类： **

CountedCompleter， **RecursiveAction**， **RecursiveTask** **

RecursiveAction：递归事件（没有返回值）

RecursiveTask：递归任务（有返回值）

ForkJoinDemo

```
package cn.fyyice.juc.fockjoin;

import java.util.concurrent.RecursiveTask;

/**
 * 求和计算
 * FockJoin
 * 1. 通过ForkJoinPool来执行
 * 2. 计算任务:execute(ForkJoinTask<?> task)
 * 3. 计算类集成类 RecursiveTask<V>
 */
public class ForkJoinDemo extends RecursiveTask<Long> {

    private Long start = 1L;
    private Long end = 10_0000_0000L;
    private Long temp = 10000L;

    public ForkJoinDemo(Long start, Long end) {
        this.start = start;
        this.end = end;
    }
    @Override
    protected Long compute() {
        if ((end-start) < temp){
            Long sum = 0L;
            for (Long i = start; i <= end; i++) {
                sum += i;
            }
            return sum;
        }else {
            long middle = (start+end)/2;
            ForkJoinDemo task1 = new ForkJoinDemo(start,middle);
            //拆分任务，把任务压入线程队列
            task1.fork();
            ForkJoinDemo task2 = new ForkJoinDemo(middle+1,end);
```

```
            //拆分任务，把任务压入线程队列
            task2.fork();

            return task1.join()+task2.join();
        }
    }
}
```

计算求和效率对比

```
package cn.fyyice.juc.fockjoin;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.stream.LongStream;

public class Test {

    private final static Long start = 1L;
    private final static Long end = 10_0000_0000L;

    public static void main(String[] args) {
        test1();
        System.out.println("----------------------------------------------------");
        test2();
        System.out.println("----------------------------------------------------");
        test3();
    }

    public static void test1(){
        long startTime = System.currentTimeMillis();
        Long sum = 0L;
        for (Long i = start; i <= end; i++) {
            sum += i;
        }
        long endTime = System.currentTimeMillis();
        System.out.println("普通累加求和结果："+sum+"\n求和时间："+(endTime-startTime));
    }

    public static void test2(){
        try {
            long startTime = System.currentTimeMillis();
            ForkJoinPool forkJoinPool = new ForkJoinPool();
//          forkJoinPool.execute(new ForkJoinDemo(start,end));        执行任务没有结果，所以这里
用sumbit
            ForkJoinTask<Long> result = forkJoinPool.submit(new ForkJoinDemo(start, end));
            Long sum = result.get();
            long endTime = System.currentTimeMillis();
            System.out.println("ForkJoin求和结果："+sum+"\n求和时间："+(endTime-startTime));
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
```

```
        }
    }

    //Stream 并行流
    public static void test3(){
        long startTime = System.currentTimeMillis();
        //range() rangeClosed()
        long result = LongStream.rangeClosed(start, end).parallel().reduce(0, Long::sum);
        long endTime = System.currentTimeMillis();
        System.out.println("Stream 并行流求和结果："+result+"\n求和时间："+(endTime-startTim
));
    }
}
```

运行结果

```
普通累加求和结果：500000000500000000
求和时间：6083
-------------------------------------------------
ForkJoin求和结果：500000000500000000
求和时间：3876
-------------------------------------------------
Stream 并行流求和结果：500000000500000000
求和时间：243

Process finished with exit code 0
```

15.异步回调

Future设计初衷

对将来的某个时间的结果进行建模

这一节不是很懂，等我理解了再来写吧。

16.JMM

谈谈你对Volatile的理解

Volatile是Java虚拟机提供的轻量级的同步机制（和Synchronized差多，但是没有他那么强）

1. **保证可见性**
2. **不保证原子性**
3. **禁止指令重排**

什么是JMM

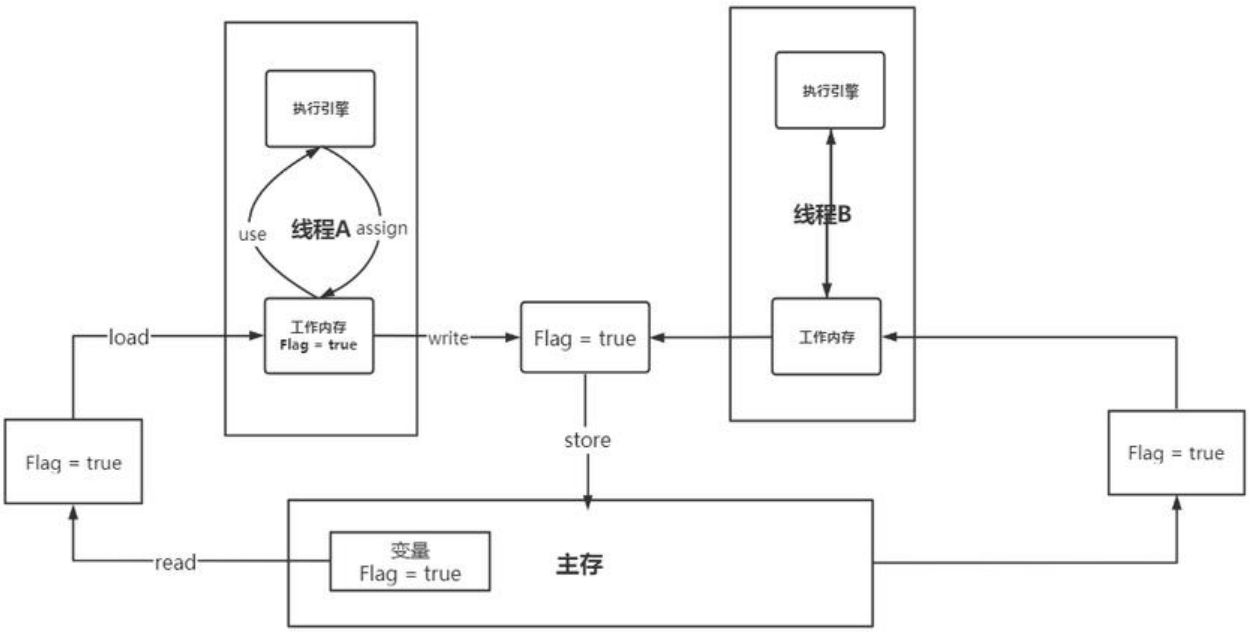
JMM： Java内存模型，不存在的东西。是一个概念、约定

关于JMM的一些同步的约定：

1. **线程解锁前，必须把共享变量=立刻**=**刷回主存。**

2. **线程加锁前，必须读取主存中的最新值到自己的工作内存中。**

3. **加锁和解锁是同一把锁。**



线程工作内存、主内存

内存交互操作

** 内存交互操作有**=**8种**=**，虚拟机实现必须保证每一个操作都是原子的，不可在分的（对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外）**

-

  - **lock （锁定）：作用于主内存的变量，把一个变量标识为线程独占状态**
  - **unlock （解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变才可以被其他线程锁定**
  - **read （读取）：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以随后的load动作使用**
  - **load （载入）：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中**
  - **use （使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟遇到一个需要使用到变量的值，就会使用到这个指令**
  - **assign （赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存变量副本中**
  - **store （存储）：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存，以便后续的write使用**
  - **write （写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入内存的变量中**

** JMM对这**=**八种指令**=**的使用，制定了如下**=**规则**=**：**

-

● 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必write

● 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存

● 不允许一个线程将没有assign的数据从工作内存同步回主内存

● 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是怼量实施use、store操作之前，必须经过assign和load操作

● 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才解锁

● 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量，必须重新load或assign操作初始化变量的值

● 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的量

● 对一个变量进行unlock操作之前，必须把此变量同步回主内存

** JMM对这八种操作规则和对**volatile的一些特殊规则**就能确定哪里操作是线程安全，哪些操是线程不安全的了。但是这些规则实在复杂，很难在实践中直接分析。所以一般我们也不会通过上述则进行分析。更多的时候，使用java的happen-before规则来进行分析。**

问题：程序不知道主内存的值已经被修改过了

17.Volatile

保证可见性

```
package cn.fyyice.juc.jmm;

import java.util.concurrent.TimeUnit;

public class JmmDemo {

    // 不加volatile 程序会出现死循环，加了  volatile 可以保证可见性
    private static int number = 0;

    public static void main(String[] args) {

        new Thread(()->{
            while (number == 0) {

            }
        }).start();

        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        number = 1;
        System.out.println(number);
    }
```

```
}
```

不保证原子性

原子性：不可分割

线程A在执行任务的时候，不能被打扰的，也不能被分割。要么同时成功，要么同时失败。

```
package cn.fyyice.juc.jmm;

//volatile不保证原子性
public class VDemo {

    private static volatile int number = 0;
    public static void main(String[] args) {

        for (int i = 1; i <= 20; i++) {
            new Thread(()->{
                for (int j = 0; j < 1000; j++) {
                    add();
                }
            },String.valueOf(i)).start();
        }

        while (Thread.activeCount() > 2){
            Thread.yield();
        }

        System.out.println(Thread.currentThread().getName()+":"+number);
    }

    public static void add (){
        //不是一个原子性操作
        /**
         * temp = number
         * number = number +1;
         * return temp
         */
            number++;
    }
}

private static volatile AtomicInteger number = new AtomicInteger();

public static void add (){
        // number++;
        //AtomicInteger +1操作
        number.getAndIncrement();
    }
```

问题：不使用synchronized、Lock锁，如何解决原子性操作?

答：使用原子类操作

原子类为何这么牛逼

** **这些类的底层都是直接和操作系统内存挂钩** ****。在内存中修改值！Unsafe类一个很特殊的存在。**

指令重排

概念：你写的程序，计算机并不是按照你写的那样执行的。**

源代码 → 编译器优化的重拍 → 指令并行也可能重排 → 内存系统也会重排 → 执行

=系统处理器在指令重排的时候，会考虑数据之间的依赖性=

volatile可以避免指令重排

在加入volatile后，系统会自动生成一个=内存屏障=。CPU指令。作用：**

1. **保证特定的操作的执行顺序。**
2. **可以保证某些变量的内存可见性（利用这些特性，volatile实现了可见性）。**

18.彻底玩转单例模式

最重要的一点：构造器私有化

饿汉模式

这种方法可能会浪费大量空间

```
package cn.fyyice.juc.single;

public class Hungry {
    /**
     * 可能会浪费空间，在类加载的时候就把所有东西给创建出来了
     */
    private Hungry(){}

    private final static Hungry HUNGRY = new Hungry();

    public static Hungry getInstance(){
        return HUNGRY;
    }

}
```

DCL懒汉模式

需要考虑到多线程并发下的安全问题，还有指令重排的问题

```
package cn.fyyice.juc.single;

//懒汉单例
public class Lazy {
    private Lazy(){
        System.out.println(Thread.currentThread().getName()+"-> Hello");
    }
```

```
    private static volatile Lazy lazy;

    public static Lazy getInstance() {
        //由于在多线程并发下不安全，加入双重检测锁模式  DCL懒汉式
        if (lazy == null){
            synchronized (Lazy.class){
                if (lazy == null){
                    /**
                     * 不是一个原子性操作
                     *  1.分配内存空间
                     *  2.执行构造方法
                     *  3.把这个对象指向这个空间
                     *
                     *  123
                     *  132  A
                     *       B  //此时Lazy还没有完成构造，导致 B 线程在判断的时候，lazy ！ = null，直接
return，空间是一片虚无，所以需要加入volatile
                     *
                     */
                    lazy = new Lazy();
                }
            }
        }
        return lazy;
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            new Thread(()->{
                Lazy.getInstance();
            }).start();
        }
    }
}
```

静态内部类

```
package cn.fyyice.juc.single;

/**
 * 静态内部类
 */
public class Holder {

    private Holder(){}

    public static Holder getInstance(){
        return innerClass.holder;
    }

    public static class innerClass{
        private static final Holder holder = new Holder();
    }
}
```

单例不安全，反射

枚举类登场

```
package cn.fyyice.juc.single;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

/**
 * enum : 本身也是一个class类，jkd1.5
 *
 */
public enum EnumSingle {

   INSTANCE;

   public static EnumSingle getInstance(){
      return INSTANCE;
   }

    public static void main(String[] args) throws NoSuchMethodException, IllegalAccessExcepti
n, InvocationTargetException, InstantiationException {
        EnumSingle single = EnumSingle.getInstance();
        //在这里idea和源码欺骗了我们，说的是无参构造，但事实上通过jad反编译过后，发现构造函
是一个String,int。狂神牛逼
        Constructor<EnumSingle> constructor = EnumSingle.class.getDeclaredConstructor(Strin
.class,int.class);
        constructor.setAccessible(true);
        EnumSingle enumSingle = constructor.newInstance();
        System.out.println(single == enumSingle);
    }
}
```

反射尝试暴力破解

```
Exception in thread "main" java.lang.IllegalArgumentException: Cannot reflectively create en
m objects
    at java.lang.reflect.Constructor.newInstance(Constructor.java:417)
    at cn.fyyice.juc.single.EnumSingle.main(EnumSingle.java:22)

Process finished with exit code 1
```

枚举类不能被破解，实属牛批

19.深入理解CAS

什么是CAS

```
//compareAndSet   比较并交换
//如果我期望的值达到了，那么就更新，否则不更新
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```

```
package cn.fyyice.juc.cas;

import java.util.concurrent.atomic.AtomicInteger;

public class CASDemo {

    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(2020);
        /**
         * public final boolean compareAndSet(int expect, int update) {
         *         return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
         * }
         *  如果我期望的值达到了，那么就更新，否则不更新
         */
        atomicInteger.compareAndSet(2020,2021);
        atomicInteger.compareAndSet(2020,2022);
        System.out.println("result:"+atomicInteger.get());
    }
}
```

unsafe类

****　**Java无法操作内存，但是 Java可以调用C++(native)操作内存。这个Unsafe相当于Java的门，通过这个类来操作内存。**

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            //获取地址偏移量
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }
    private volatile int value;
```

内存操作，效率很高

```
// var1 -->getAndIncrement   var2 -->valueOffset    var4 --> 1
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    //自旋锁
    do {
        //获取内存地址中的值
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

CAS：比较当前工作内存中的值，如果这个值是期望的，那么就执行。如果不是，就一直循环

缺点：

1. **由于底层是自旋锁，循环会耗时**
2. **一次性只能保证一个共享变量的原子性**
3. **存在ABA问题**

CAS：ABA问题（狸猫换太子）

****业务场景：****库存业务**

** **两个并发同时读取数据库查询库存。此时库存count = 10**

** **A线程、B线程同时读取到库存都是为10，A线程购买了2个商品，B线程购买了3个商品。由于用了CAS，假设A线程先完成，则库存count 为 10→ 8 → 7，但理想结果应该为5，这里就出现了数不一致。（最后一次设置库存会覆盖和掩盖前一次并发操作）**

20.原子引用

带版本号的操作（类似乐观锁操作）

```
AtomicStampedReference<Integer> atomicStampedReference = new AtomicStampedRefere
ce<>(1,1);
```

** **Integer使用了对象缓存机制，默认范围-128~127，推荐侍弄静态工厂方法valueOf获取对实例，而不是new，因为valueOf使用缓存，而new一定会创建新的对象分配新的内存空间。**

```
package cn.fyyice.juc.cas;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicStampedReference;

public class CASDemo {

    //如果泛型是一个包装类，要注意泛型的引用问题
    //在正常的业务操作中，这里比较的都是一个个的对象
    static AtomicStampedReference<Integer> atomicStampedReference = new AtomicStampe
Reference<>(1,1);

    public static void main(String[] args) {

        new Thread(()->{
            int stamp = atomicStampedReference.getStamp();
            System.out.println("Thread 1-1 ----"+stamp);

            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(atomicStampedReference.compareAndSet(1, 2, stamp, stamp + 1));
```

```
            System.out.println("Thread 1-2 ----"+atomicStampedReference.getStamp());

            System.out.println(atomicStampedReference.compareAndSet(2, 1, atomicStampedRef
rence.getStamp(), atomicStampedReference.getStamp() + 1));
            System.out.println("Thread 1-3 ----"+atomicStampedReference.getStamp());
        },"Thread 1").start();


        new Thread(()->{
            int stamp = atomicStampedReference.getStamp();
            System.out.println("Thread 2-1 ----"+stamp);

            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread 2 修改："+atomicStampedReference.compareAndSet(1, 6, s
amp, stamp + 1));
            System.out.println("Thread 2-2 ----"+atomicStampedReference.getStamp());
        },"Thread 2").start();
    }
}
```

21.各种锁的理解

公平锁、非公平锁

公平锁：非常公平，线程顺序执行，先来后到，不能插队

非公平锁：非常不公平，可以插队（默认都是非公平锁）

通过构造器来创建

```
/**
* Creates an instance of {@code ReentrantLock}.
* This is equivalent to using {@code ReentrantLock(false)}.
*/
public ReentrantLock() {
    sync = new NonfairSync();
}

/**
* Creates an instance of {@code ReentrantLock} with the
* given fairness policy.
*
* @param fair {@code true} if this lock should use a fair ordering policy
*/
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

可重入锁

又称为递归锁。

拿到了外面的锁之后，就可以拿到里面的锁（自动获得的）

Synnchronized锁

```
package cn.fyyice.juc.lock;

public class Demo1 {
    public static void main(String[] args) {
        Store store = new Store();

        new Thread(()->{
            store.pay();
            store.sale();
        },"A").start();

        new Thread(()->{
            store.pay();
            store.sale();
        },"B").start();

    }
}
// synchronized对普通方法加锁，锁的是这个方法的调用者，本质上是同一把锁
class Store{
    public synchronized void pay(){
        System.out.println(Thread.currentThread().getName()+"-->Pay");
    }
    public synchronized void sale(){
        System.out.println(Thread.currentThread().getName()+"-->Sale");
    }
}
```

Lock锁

```
package cn.fyyice.juc.lock;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Demo2 {
    public static void main(String[] args) {
        Store2 store = new Store2();

        new Thread(()->{
            store.pay();
            store.sale();
        },"A").start();

        new Thread(()->{
            store.pay();
            store.sale();
        },"B").start();
```

```
    }
}

//lock锁。与synchronized体现的效果是一样的，但是本质上是有区别的。
//在这里lock锁有两把锁，在使用的时候必须配对，否则会出现死锁的情况
class Store2{
    private Lock lock = new ReentrantLock();
    public void pay(){
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName()+"-->Pay");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public void sale(){
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName()+"-->Sale");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

自旋锁

SpinLock（核心CAS）

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

自己写一个自旋锁

```
package cn.fyyice.juc.lock;

import java.util.concurrent.atomic.AtomicReference;

public class SpinLock {
    AtomicReference<Thread> atomicReference = new AtomicReference<>();

    //加锁
    public void lock(){
        Thread thread = Thread.currentThread();
```

```
        System.out.println(thread.getName()+"--->myLock");
        do{

        }while (!atomicReference.compareAndSet(null,thread));
    }

    //解锁
    public void unlock(){
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName()+"--->myUnLock");
        atomicReference.compareAndSet(thread,null);
    }
}
```

测试

A加锁过后，B加锁，但是A先拿到了，进入try模块中睡觉，此时此刻，B线程在不断的自旋；当A线睡完过后执行解锁，释放了锁后B线程拿到锁，解锁。

```
package cn.fyyice.juc.lock;

import java.util.concurrent.TimeUnit;

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

        SpinLock lock = new SpinLock();

        new Thread(()->{
            lock.lock();
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        },"A").start();

        new Thread(()->{
            lock.lock();
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        },"B").start();
    }
}
```

结果

```
A--->myLock
```

```
B--->myLock
A--->myUnLock
B--->myUnLock

Process finished with exit code 0
```

死锁

多个线程相互争抢资源。

```
package cn.fyyice.juc.lock;

import java.util.concurrent.TimeUnit;

public class DeadLock {
    public static void main(String[] args) {
        new Thread(new MyLock("lockA","lockB"),"T1").start();
        new Thread(new MyLock("lockB","lockA"),"T2").start();
    }
}

class MyLock implements Runnable{
    private String lockA;
    private String lockB;
    MyLock(String lockA,String lockB){
        this.lockA = lockA;
        this.lockB = lockB;
    }

    @Override
    public void run() {
        synchronized (lockA){
            System.out.println(Thread.currentThread().getName() + "-->lock:" + lockA + "===get-
>"+lockB);

            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (lockB){
                System.out.println(Thread.currentThread().getName() + "-->lock:" + lockB + "===g
et-->"+lockA);
            }
        }
    }
}
```

解决问题

排查死锁解决问题：

1. **使用**jps-l**定位进程号**

```
Microsoft Windows [版本 10.0.19042.867]
(c) 2020 Microsoft Corporation. 保留所有权利。

D:\project\juc>jps -l
11428 cn.fyyice.juc.lock.DeadLock
1208 cn.fyyice.juc.lock.Demo2
7596
7980 sun.tools.jps.Jps
9260 org.jetbrains.jps.cmdline.Launcher
```

2. **使用**jstack 进程号**找到死锁问题（看堆栈信息）**

```
Found one Java-level deadlock:
=============================
"T2":
  waiting to lock monitor 0x0000017fd1a26d68 (object 0x000000076b6a6440, a java.lang.Stri
g),
  which is held by "T1"
"T1":
  waiting to lock monitor 0x0000017fd1a296a8 (object 0x000000076b6a6478, a java.lang.Strin
),
  which is held by "T2"

Java stack information for the threads listed above:
===================================================
"T2":
        at cn.fyyice.juc.lock.MyLock.run(DeadLock.java:31)
        - waiting to lock <0x000000076b6a6440> (a java.lang.String)
        - locked <0x000000076b6a6478> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:748)
"T1":
        at cn.fyyice.juc.lock.MyLock.run(DeadLock.java:31)
        - waiting to lock <0x000000076b6a6478> (a java.lang.String)
        - locked <0x000000076b6a6440> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

总结

**　**视频学习大概花了5天时间，有些地方仍然有些问题，还需要多多理解呀！**