

# JUC 并发编程 + 底层原理

注意，一定要是JDK1.8、IDE 一定要设置

## 1、什么是JUC（重要）



`java.util.concurrent;`

## 2、进程和线程回顾

进程 / 线程是什么？

技术：不能使用一句话说出来，你不会！

进程：QQ.exe 、 Music.exe . 程序

线程：一个进程中可能包含多个线程，至少包含一个。JAVA

main 、 GC 线程

并发 / 并行是什么？

并发编程？ 并发。并行；

并发：多线程、多个线程操作一个资源类，快速交替过程。

并行：多核多CPU；

你吃饭，吃到一半，电话来了，3种情况

1、吃完再去接电话（单线程）

2、先接电话再吃（交替、并发）

3、边吃边接电话（并行）

一个CPU 的电脑，能不能并行执行任务？

==所以说，并发编程的主要目的，充分利用CPU的资源，提高性能；==

## 线程的状态

线程的状态 6 种 （学习的方式：看源码+官方文档）

```
public enum State {  
  
    NEW,  
  
    RUNNABLE,  
  
    BLOCKED,  
  
    WAITING, // 等待  
  
    TIMED_WAITING, // 延时等待  
  
    TERMINATED; //  
}
```

## wait / sleep 的区别

### 1、类

wait Object

sleep Thread , 谁调用的谁睡觉!

A 调用了 B 的sleep方法, 实际上是谁睡觉?

### 2、是否释放锁

sleep抱着锁睡觉。

wait 会释放锁!

### 3、使用范围不同

wait、notify、notifyAll 只能用在同步方法中或者同步代码块中;

Sleep 可以再任意地方使用;

### 4、异常

sleep, 必须捕获异常!

wait , 不需要捕获异常!

## 3、Lock锁

### 传统的 synchronized

```
package com.coding.demo01;  
  
import java.util.TimerTask;  
  
/**  
 * 卖票    自己会写    3个售票员卖出30张票
```

```

* 企业中禁止这样写，Coding：企业级开发！
*
* 多线程编程的固定套路：
*     1、高内聚，低耦合    （前提）
*     2、线程    操作（调用对外暴露的方法）    资源类    （要点）
*/
public class SaleTicketTest1 {
    public static void main(String[] args) {

        // 资源类
        final SaleTicket saleTicket = new SaleTicket();

        new Thread(new Runnable() {
            public void run() {
                for (int i = 1; i < 40; i++) {
                    saleTicket.saleTicket();
                }
            }
        }, "A").start();

        new Thread(new Runnable() {
            public void run() {
                for (int i = 1; i < 40; i++) {
                    saleTicket.saleTicket();
                }
            }
        }, "B").start();

        new Thread(new Runnable() {
            public void run() {
                for (int i = 1; i < 40; i++) {
                    saleTicket.saleTicket();
                }
            }
        }, "C").start();

    }
}

// 属性，和方法    高内聚
class SaleTicket{ //资源类
    private int number = 30;

    // 卖票方法
    public synchronized void saleTicket(){
        if (number>0){
            System.out.println(Thread.currentThread().getName()+"卖出第"+
(number--) +"还剩下: "+number+"张票");
        }
    }
}

```

java.text.spi  
java.time  
java.time.chrono  
java.time.format  
java.time.temporal  
java.time.zone  
java.util  
java.util.concurrent  
java.util.concurrent.atomic  
**java.util.concurrent.locks**  
java.util.function  
java.util.jar  
java.util.logging

java.util.concurrent.locks

Interfaces  
Condition  
**Lock**  
ReadWriteLock  
Classes  
AbstractOwnableSynchronizer  
AbstractQueuedLongSynchronizer  
AbstractQueuedSynchronizer  
LockSupport  
ReentrantLock  
ReentrantReadWriteLock  
ReentrantReadWriteLock.ReadLock  
ReentrantReadWriteLock.WriteLock  
StampedLock

概述 软件包 类 使用 树 已过时的 索引 帮助

上一个 下一个 框架 无框架

概要: 嵌套 | 字段 | 构造方法 | 方法 详细信息: 字段 | 构造方法 | 方法

compact1, compact2, compact3  
java.util.concurrent.locks

**Interface Lock**

所有已知实现类:  
ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock

public interface Lock

**Lock实现提供比使用synchronized方法和语句可以获得的更广泛的锁定操作。它们允许更灵活的结构化,可能具有完全不同的属性,并且可以支持多个相关联的对象Condition。**

锁是用于通过多个线程控制对共享资源的访问的工具。通常,锁提供对共享资源的独占访问,一次只能有一个线程可以获取锁,并且对共享资源的所有访问都要求首先获取锁。但是,一些锁可能允许并发访问共享资源,如ReadWriteLock的读锁。

使用synchronized方法或语句提供对与每个对象相关联的隐式监视器锁的访问,但是强制所有锁获取和释放以块结构的方式发生;当获取多个锁时,它们必须以相反的顺序被释放,并且所有的锁都必须被释放在与它们相同的词汇范围内。

虽然synchronized方法和语句的范围机制使得使用监视器锁更容易编程,并且有助于避免涉及锁的许多常见编程错误,但是有时您需要以更灵活的方式处理锁。例如,用于遍历并发访问的数据结构的一些算法需要使用“手动”或“链锁定”;您获取节点A的锁,然后获取节点B,然后释放A并获取C,然后释放B并获取D等。所述的实施方式中Lock接口通过允许获得并在不同的范围释放的锁,并允许获得并以任何顺序释放多个锁使得能够使用这样的技术。

随着这种增加的灵活性,额外的责任。没有块结构化锁定会删除使用synchronized方法和语句发生的锁的自动释放。在大多数情况下,应使用以下惯用语:

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

网站地址1 网站地址  
我要纠错... 修正翻译内容  
QQ群: 86472519  
安卓帮助文档

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

```
package com.coding.demo01;  
  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
/**  
 * 卖票 自己会写 3个售票员卖出30张票  
 * 企业中禁止这样写, Coding: 企业级开发!  
 * <p>  
 * 多线程编程的固定套路:  
 * 1、高内聚, 低耦合 (前提)  
 * 2、线程 操作(调用对外暴露的方法) 资源类 (要点)  
 */  
public class SaleTicketTest2 {  
    public static void main(String[] args) {  
  
        // 并发: 多线程操作同一个资源类  
        // 资源类  
        SaleTicket2 saleTicket = new SaleTicket2();  
  
        // lambda表达式、链式编程、流式计算!  
        // lambda表达式, () -> {} 自动推断类型  
  
        // IDEA 一定要设置 JDK 版本为 1.8 版本  
        new Thread(() -> {  
            for (int i = 1; i < 40; i++) saleTicket.saleTicket();  
        }, "A").start();  
  
        new Thread(() -> {  
            for (int i = 1; i < 40; i++) saleTicket.saleTicket();  
        }, "B").start();  
    }  
}
```

```

        new Thread() -> {
            for (int i = 1; i < 40; i++) saleTicket.saleTicket();
        }, "C").start();
    }
}

// 属性, 和方法 高内聚
class SaleTicket2 { //资源类
    private int number = 30;

    // 锁LOCK
    private Lock lock = new ReentrantLock(); // 可重入

    // 卖票方法
    public void saleTicket() {
        lock.lock(); // 加锁
        try {
            // 业务代码
            if (number > 0) {
                System.out.println(Thread.currentThread().getName() + "卖出第" +
                    (number--) + "还剩下: " + number + "张票");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock(); // 解锁
        }
    }
}

```

synchronized 和 lock 区别 (自动挡 手动挡)

- 1、synchronized 关键字, java内置的。lock 是一个Java 类
- 2、synchronized 无法判断是否获取锁、lock 可以判断是否获得锁
- 3、synchronized 锁会自动释放! lock 需要手动在 finally 释放锁, 如果不释放锁, 就会死锁
- 4、synchronized 线程1阻塞 线程2永久等待下去。 lock可以 lock.tryLock(); // 尝试获取锁, 如果尝试获取不到锁, 可以结束等待
- 5、synchronized 可重入, 不可中断, 非公平的, Lock锁, 可重入、可以判断、可以公平!

## 4、生产者和消费者 (高频)

线程间的通信、无法通信, 调度线程

生产者和消费者 synchronized 版

```
package com.coding.demo02;
```

```
/**
```

```
 * 题目：现在两个线程，操作一个初始值为0的变量
 *       一个线程 + 1， 一个线程 -1。判断什么时候+1，什么时候-1
 *       交替10 次
 *
 * 方法论：
 *
 * 多线程编程的固定套路：
 * 1、高内聚，低耦合 （前提）
 * 2、线程 操作(调用对外暴露的方法) 资源类 （要点）
 *
 * 生产者消费者模型： 判断、干活、通知
 */
```

```
public class A {
```

```
    public static void main(String[] args) {
        Data data = new Data();
```

```
        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "A").start();
```

```
        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "B").start();
```

```
    }
```

```
}
```

```
// 资源类 属性，方法
```

```
class Data{
```

```
    private int num = 0;
```

```
    // +1
```

```
    public synchronized void increment() throws Exception{
```

```
        //判断
```

```
        if (num!=0){
```

```
            this.wait();
```

```
        }
```

```
        // 干活
```

```
        num++;
```

```
        System.out.println(Thread.currentThread().getName()+"\t"+num);
```

```

        // 通知
        this.notifyAll();
    }

    // -1
    public synchronized void decrement() throws Exception{
        // 判断
        if (num==0){
            this.wait();
        }
        // 干活
        num--;
        System.out.println(Thread.currentThread().getName()+"\t"+num);
        // 通知
        this.notifyAll();
    }
}

```

## 问题升级：防止虚假唤醒，4个线程，两个加，两个减

```

public final void wait()
    throws InterruptedException

```

导致当前线程等待，直到另一个线程调用该对象的notify()方法或notifyAll()方法。换句话说，这个方法的行为就好像简单地执行wait(0)。

当前的线程必须拥有该对象的显示器。该线程释放此监视器的所有权，并等待另一个线程通知等待该对象监视器的线程通过调用notify方法或notifyAll方法notifyAll。然后线程等待，直到它可以重新获得监视器的所有权并恢复执行。

像在一个参数版本中，中断和虚假唤醒是可能的，并且该方法应该始终在循环中使用：

```

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

```

该方法只能由作为该对象的监视器的所有者的线程调用。有关线程可以成为监视器所有者的方式的说明，请参阅notify方法。

### 异常

IllegalMonitorStateException - 如果当前线程不是对象监视器的所有者。

InterruptedException - 如果任何线程在当前线程等待通知之前或当前线程中断当前线程。当抛出此异常时，当前线程的 *中断状态* 将被清除。

另请参见：

notify() , notifyAll()

```

package com.coding.demo02;

```

```

/**
 * 题目：现在两个线程，操作一个初始值为0的变量
 *       一个线程 + 1， 一个线程 -1。判断什么时候+1，什么时候-1
 *       交替10 次
 *
 * 方法论：
 *
 * 多线程编程的固定套路：
 * 1、高内聚，低耦合 （前提）
 * 2、线程 操作（调用对外暴露的方法） 资源类 （要点）
 *
 * 生产者消费者模型： 判断、干活、通知
 */
public class A {

    public static void main(String[] args) {
        Data data = new Data();
    }
}

```

```

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "A").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "B").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "C").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "D").start();
    }
}

// 资源类 属性, 方法
class Data{
    private int num = 0;

    // +1
    public synchronized void increment() throws Exception{
        //判断 if 只判断了一次, 0 1 0 1
        while (num!=0){
            this.wait();
        }
        // 干活
        num++;
        System.out.println(Thread.currentThread().getName()+"\t"+num);
    }
}

```



```

        // 通知
        this.notifyAll();
    }

    // -1
    public synchronized void decrement() throws Exception{
        // 判断
        while (num==0){
            this.wait(); //0
        }
        // 干活
        num--;
        System.out.println(Thread.currentThread().getName()+"\t"+num);
        // 通知
        this.notifyAll();
    }
}

```

传统的, JUC!

## 新版生产者和消费者写法

AbstractQueuedLongSynchronizer.ConditionObject , AbstractQueuedSynchronizer.ConditionObject

```

public interface Condition

```

Condition因素出Object监视器方法（wait, notify和notifyAll）成不同的对象，以得到具有多个等待集的每个对象，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（也称为条件或条件变量）为一个线程暂停执行（“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。因为访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。等待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。

一个Condition实例本质上绑定到一个锁。要获得特定Condition实例的Condition实例，请使用其newCondition()方法。

例如，假设我们有一个有限的缓冲区，它支持put和take方法。如果在一个空的缓冲区尝试一个take，则线程将阻塞直到一个项目可用；如果put试图在一个完整的缓冲区，那么线程将阻塞，直到空间变得可用。我们希望在单独的等待集中等待put线程和takes线程，以便我们可以在缓冲区中的项目或空间可用时使用仅通知单个线程的优化。这可以使用两个Condition实例来实现。

```

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;

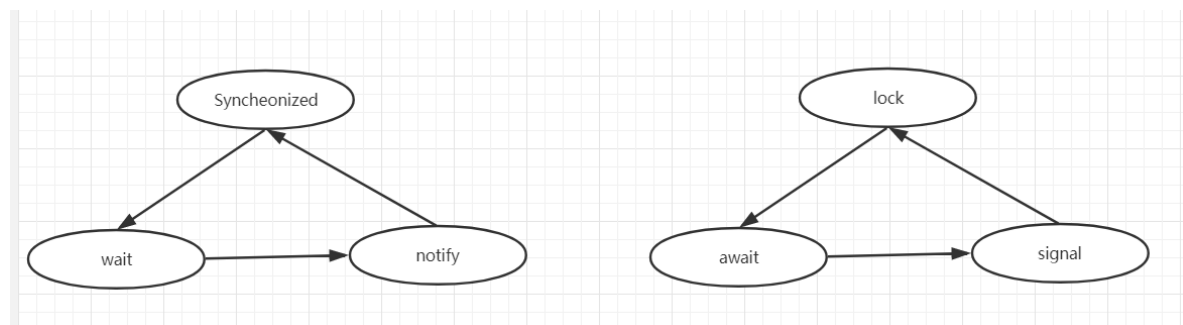
    public void put(Object x) throws InterruptedException {
        lock.lock(); try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
     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

    public Object take() throws InterruptedException {

```

jdk  
中  
英  
对  
照

任何一个新技术的出现，一定不仅仅是换了个马甲



手写生产者消费者问题：100 加分项目！

```

package com.coding.demo02;

```

```
import sun.awt.SunHints;

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

// lock版生产者消费者
public class B {

    public static void main(String[] args) {
        // 新版
        Data2 data = new Data2();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "A").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "B").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.increment();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "C").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                try {
                    data.decrement();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, "D").start();

    }

}
```

// 资源类 属性, 方法

```
class Data2{
    private int num = 0;
    // 定义锁
    Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    // +1
    public void increment() throws Exception{

        // 加锁
        lock.lock();

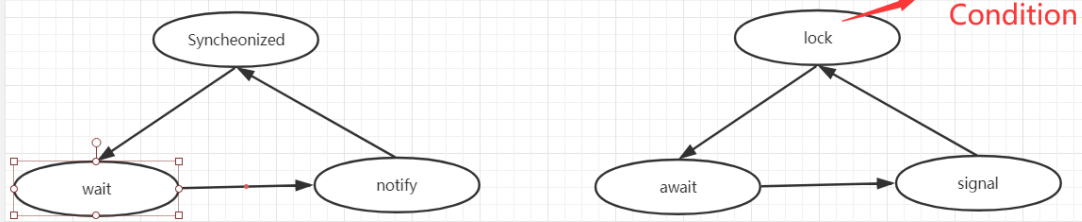
        try {
            //判断
            while (num!=0){
                condition.await(); //等待
            }
            // 干活
            num++;
            System.out.println(Thread.currentThread().getName()+"\t"+num);
            // 通知
            condition.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 解锁
            lock.unlock();
        }
    }

    // -1
    public void decrement() throws Exception{

        // 加锁
        lock.lock();
        try {
            // 判断
            while (num==0){
                condition.await(); //等待
            }
            // 干活
            num--;
            System.out.println(Thread.currentThread().getName()+"\t"+num);
            // 通知
            condition.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 解锁
            lock.unlock();
        }
    }
}
```

## 如何精确通知访问!

### 精确通知顺序访问 Condition



```
package com.coding.demo02;

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

/**
 * 多个线程启动 A -- B -- C
 * 三个线程依次打印
 * A 5次
 * B 10次
 * C 15次
 * 依次循环
 * 精确通知线程消费
 */
public class C {

    public static void main(String[] args) {
        Data3 data = new Data3();

        // 线程操作资源类
        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                data.print5();
            }
        }, "A").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                data.print10();
            }
        }, "B").start();

        new Thread()->{
            for (int i = 1; i <= 10; i++) {
                data.print15();
            }
        }, "C").start();
    }
}
```

```

// 资源类 属性, 方法
class Data3{
    private int num = 1; // A1 B2 C3
    // 定义锁
    Lock lock = new ReentrantLock();
    private Condition condition1 = lock.newCondition(); //3个判断, 交替执行 A--B--
C--A
    private Condition condition2 = lock.newCondition(); //3个判断, 交替执行 A--B--
C--A
    private Condition condition3 = lock.newCondition(); //3个判断, 交替执行 A--B--
C--A

    // 3个方法、作业, 合3为1
    // +1
    public void print5(){
        // 加锁
        lock.lock();
        try {
            //判断
            while (num!=1){
                condition1.await(); //等待
            }
            // 干活
            for (int i = 1; i <=5 ; i++) {
                System.out.println(Thread.currentThread().getName()+"\t"+i);
            }

            // 第一个线程通知第二个线程, 第二个线程通知第三个.... 计数器
            num=2;
            // 通知第二个线程干活, 指定谁干活
            condition2.signal();

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 解锁
            lock.unlock();
        }
    }

    public void print10() {
        // 加锁
        lock.lock();
        try {
            //判断
            while (num!=2){
                condition2.await(); //等待
            }
            // 干活
            for (int i = 1; i <=10 ; i++) {
                System.out.println(Thread.currentThread().getName()+"\t"+i);
            }

            // 第一个线程通知第二个线程, 第二个线程通知第三个.... 计数器
            num=3;
            // 通知第二个线程干活, 指定谁干活
            condition3.signal();
        }
    }
}

```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        // 解锁
        lock.unlock();
    }
}

public void print15() {
    // 加锁
    lock.lock();
    try {
        //判断
        while (num!=3){
            condition3.await(); //等待
        }
        // 干活 = 业务代码
        for (int i = 1; i <=15 ; i++) {
            System.out.println(Thread.currentThread().getName()+"\t"+i);
        }

        num=1;
        condition1.signal();

    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        // 解锁
        lock.unlock();
    }
}
}

```

大家懂锁的吗？锁的谁！

## 5、8锁的现象（深入理解锁）

### 1、标准访问，请问先打印邮件还是短信？

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/**
 * **1、标准访问，请问先打印邮件1还是短信2？ **
 *
 *
 */
public class Test1 {
    public static void main(String[] args) {

```

```

        Phone phone = new Phone();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
        new Thread(() -> {
            phone.sendEmail();
        }, "A").start();

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

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

// 手机，发短信，发邮件
class Phone {
    // 被 synchronized 修饰的方法、锁的对象是方法的调用者、
    public synchronized void sendEmail() {
        System.out.println("sendEmail");
    }

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

```

## 2、邮件方法暂停4秒钟，请问先打印邮件还是短信？

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/**
 * **2、邮件方法暂停4秒钟，请问先打印邮件还是短信？**
 */
public class Test2 {
    public static void main(String[] args) {

        Phone2 phone = new Phone2();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
        new Thread(() -> {
            phone.sendEmail();
        }, "A").start();

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

```

```

        e.printStackTrace();
    }

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

// 手机，发短信，发邮件
class Phone2 {
    // 被 synchronized 修饰的方法、锁的对象是方法的调用者、
    public synchronized void sendEmail() {
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }

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

```

### 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello?

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/**
 * 3、新增一个普通方法hello()没有同步,请问先打印邮件还是hello?
 */
public class Test3 {

    // 回家 卧室(锁) 厕所
    public static void main(String[] args) {

        Phone3 phone = new Phone3();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
        new Thread(() -> { // 一开始就执行了
            phone.sendEmail();
        }, "A").start();

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

        new Thread(() -> { // 一秒后执行

```



```

        phone.hello();
    }, "B").start();

}

}

// 锁：竞争机制

// 手机，发短信，发邮件
class Phone3 {
    // 被 synchronized 修饰的方法、锁的对象是方法的调用者、
    public synchronized void sendEmail() {
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }

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

    // 新增的方法没有被 synchronized 修饰，不是同步方法，所以不需要等待，其他线程用了一个把
    // 锁
    public void hello() {
        System.out.println("hello");
    }

}

```

#### 4、两部手机、请问先打印邮件还是短信？

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/**
 * **4、两部手机、请问先打印邮件还是短信？**
 */
public class Test4 {

    // 回家 卧室(锁) 厕所
    public static void main(String[] args) {
        // 两个对象，互不干预
        Phone4 phone1 = new Phone4();
        Phone4 phone2 = new Phone4();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
        new Thread(() -> { // 一开始就执行了
            phone1.sendEmail();
        }, "A").start();

        // 干扰
        try {
            TimeUnit.SECONDS.sleep(1);

```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    new Thread(() -> { // 一秒后执行
        phone2.sendMS();
    }, "B").start();

}

}

// 手机，发短信，发邮件
class Phone4 {
    // 被 synchronized 修饰的方法、锁的对象是方法的调用者、调用者不同，没有关系，量个方法用得不是同一个锁！
    public synchronized void sendEmail() {
        // 善意的延迟
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }

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

}

```

## 5、两个静态同步方法，同一部手机，请问先打印邮件还是短信？

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/*
**5、两个静态同步方法，同一部手机，请问先打印邮件还是短信？**
*/
public class Test5 {
    // 回家 卧室(锁) 厕所
    public static void main(String[] args) {
        // 两个对象，互不干预
        Phone5 phone = new Phone5();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
        new Thread(() -> { // 一开始就执行了
            phone.sendEmail();
        }, "A").start();

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

```

```

    }

    new Thread(() -> { // 一秒后执行
        phone.sendMS();
    }, "B").start();

}

}

// 手机，发短信，发邮件
class Phone5 {

    // 对象    类模板可以new 多个对象!
    // Class    类模版，只有一个

    // 被 synchronized 修饰 和 static 修饰的方法，锁的对象是类的 class 对象！唯一的
    // 同一把锁

    public static synchronized void sendEmail() {
        // 善意的延迟
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }

    public static synchronized void sendMS() {
        System.out.println("sendMS");
    }

}

```

## 6、两个静态同步方法，2部手机，请问先打印邮件还是短信？

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/** 第一次听可能不会，第二次也可能不会，但是不要放弃，你可以！
**6、两个静态同步方法，2部手机，请问先打印邮件还是短信？**
*/
public class Test6 {

    // 回家 卧室(锁) 厕所
    public static void main(String[] args) {
        // 两个对象，互不干预
        Phone5 phone = new Phone5();
        Phone5 phone2 = new Phone5();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
        new Thread(() -> { // 一开始就执行了
            phone.sendEmail();
        }, "A").start();
    }
}

```

```

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

        new Thread(() -> { // 一秒后执行
            phone2.sendMS();
        }, "B").start();
    }
}

// 手机，发短信，发邮件
class Phone6 {

    // 对象    类模板可以new 多个对象！
    // Class    类模版，只有一个

    // 被 synchronized 修饰 和 static 修饰的方法，锁的对象是类的 class 对象！唯一的
    // 同一把锁

    public static synchronized void sendEmail() {
        // 善意的延迟
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }

    public static synchronized void sendMS() {
        System.out.println("sendMS");
    }
}

```

## 7、一个普通同步方法，一个静态同步方法，同一部手机，请问先打印邮件还是短信？

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/**
 * 7、一个普通同步方法，一个静态同步方法，同一部手机，请问先打印邮件还是短信？ **
 */
public class Test7 {
    // 回家 卧室(锁) 厕所
    public static void main(String[] args) {
        // 两个对象，互不干预
        Phone7 phone = new Phone7();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
    }
}

```

```

        new Thread(() -> { // 一开始就执行了
            phone.sendEmail();
        }, "A").start();

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

        new Thread(() -> { // 一秒后执行
            phone.sendMS();
        }, "B").start();
    }
}

class Phone7{
    // CLASS
    public static synchronized void sendEmail() {
        // 善意的延迟
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }

    // 对象
    // 普通同步方法
    public synchronized void sendMS() {
        System.out.println("sendMS");
    }
}

```

#### 8、一个普通同步方法，一个静态同步方法，2部手机，请问先打印邮件还是短信？

```

package com.coding.lock8;

import java.util.concurrent.TimeUnit;

/**
 * **8、一个普通同步方法，一个静态同步方法，2部手机，请问先打印邮件还是短信？**
 */
public class Test8 {

    public static void main(String[] args) {
        // 两个对象，互不干预
        Phone8 phone = new Phone8();
        Phone8 phone2 = new Phone8();

        // 我们这里两个线程使用的是同一个对象。两个线程是一把锁！先调用的先执行！
        new Thread(() -> { // 一开始就执行了

```

```

        phone.sendEmail();
    }, "A").start();

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

    new Thread(() -> { // 一秒后执行
        phone2.sendMS();
    }, "B").start();

}

}

class Phone8{

    // CLASS
    public static synchronized void sendEmail() {
        // 善意的延迟
        try {
            TimeUnit.SECONDS.sleep(4);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("sendEmail");
    }

    // 对象
    // 普通同步方法
    public synchronized void sendMS() {
        System.out.println("sendMS");
    }

}

```

coding: 魔鬼教练, 难受

## 小结

**new this** 本身的这个对象, 调用者

**static class** 类模板, 保证唯一!

一个对象中有多个 synchronized 方法, 某个时刻内只要有一个线程去访问 synchronized 方法了就会被加锁, 独立公共厕所! 其他线程就会阻塞!

加了一个普通方法后, 两个对象, 无关先后, 一个有锁, 一个没锁! 情况会变化!

换成静态同步方法, 情况会变化! CLASS, 所有静态同步方法的锁唯一 对象实例class 本身!

## 6、集合类不安全

list 不安全

单线程：安全

```
package com.coding.collunsafe;

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

public class UnsafeList1 {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("a", "b", "c");
        list.forEach(System.out::println);
    }
}
```

多线程：不安全

```
package com.coding.collunsafe;

import java.util.ArrayList;
import java.util.Random;
import java.util.UUID;

public class UnsafeList2 {
    public static void main(String[] args) {
        // 代码实现
        ArrayList<Object> list = new ArrayList<>();

        // 测试多线程下是否安全List，3条线程都不安全了
        // 多线程下记住一个异常，并发修改异常
        java.util.ConcurrentModificationException

        // Exception ConcurrentModificationException
        for (int i = 1; i <= 30; i++) {
            new Thread()->{
                // 3个结果
                list.add(UUID.randomUUID().toString().substring(0,5));
                System.out.println(list);
            }.start();
        }
    }
}
```

不安全怎么解决：

```
package com.coding.collunsafe;
```

```

import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * 善于总结：
 * 1、 故障现象： ConcurrentModificationException
 * 2、 导致原因： 多线程操作集合类不安全
 * 3、 解决方案：
 *     List<String> list = new Vector<>(); // Vector 是一个线程安全的类,效率低下
50
 *     List<String> list = Collections.synchronizedList(new ArrayList<>()); //
60
 *     List<String> list = new CopyOnWriteArrayList<>(); // JUC 100 推荐使用
 */
public class UnsafeList2 {
    public static void main(String[] args) {
        // 代码实现
        // ArrayList<Object> list = new ArrayList<>(); // 效率高, 不支持并发!
        // List<String> list = new Vector<>(); // Vector 是一个线程安全的类,效率低下
50
        // List<String> list = Collections.synchronizedList(new ArrayList<>());
// 60

        // 多线程高并发程序中, 一致性最为重要

        // 写入时复制; COW 思想, 计算机设计领域。优化策略
        // 思想: 多个调用者, 想调用相同的资源; 指针
        // 只是去读, 就不会产生锁!
        // 假如你是去写, 就需要拷贝一份都自己哪里, 修改完毕后, 在替换指针!

        List<String> list = new CopyOnWriteArrayList<>(); // JUC 100

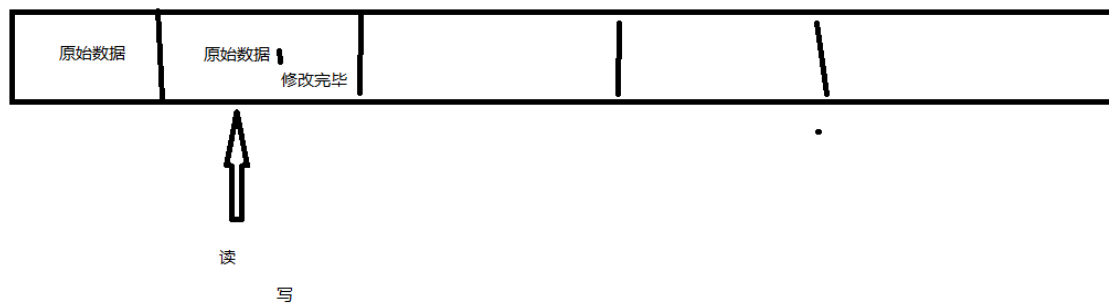
        // 测试多线程下是否安全List, 3条线程都不安全了
        // 多线程下记住一个异常, 并发修改异常
        java.util.ConcurrentModificationException

        // Exception ConcurrentModificationException
        for (int i = 1; i <= 30; i++) {
            new Thread()->{
                // 3个结果
                list.add(UUID.randomUUID().toString().substring(0,5));
                System.out.println(list);
            },String.valueOf(i)).start();
        }
    }
}

```

思想原理：指针，复制指向的问题





## set 不安全

```
package com.coding.collunsafe;

import java.util.Collections;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;
import java.util.concurrent.CopyOnWriteArraySet;

/**
 * 善于总结：
 * 1、故障现象： ConcurrentModificationException 并发修改异常！
 * 2、导致原因： 并发下 HashSet 存在安全的问题
 * 3、解决方案：
 *    Set<String> set = Collections.synchronizedSet(new HashSet<>()); 60
 *    Set<String> set =new CopyOnWriteArraySet<>(); // 100
 */
public class UnsafeSet1 {
    public static void main(String[] args) {
        // Set<String> set = new HashSet<>(); // 底层是什么
        Set<String> set =new CopyOnWriteArraySet<>();

        for (int i = 1; i <=30 ; i++) {
            new Thread()->{
                set.add(UUID.randomUUID().toString().substring(0,5));
                System.out.println(set);
            },String.valueOf(i)).start();
        }
    }
}
```

## map 不安全

java.time.chrono  
java.time.format  
java.time.temporal  
java.time.zone  
java.util  
java.util.concurrent  
java.util.concurrent.atomic  
java.util.concurrent.locks  
java.util.function  
java.util.jar  
java.util.logging  
java.util.prefs  
java.util.regex  
java.util.spi  
TransferQueue

Classes

AbstractExecutorService  
ArrayBlockingQueue  
CompletableFuture  
ConcurrentHashMap  
ConcurrentHashMap.KeySetView  
ConcurrentLinkedDeque  
ConcurrentLinkedQueue  
ConcurrentSkipListMap  
ConcurrentSkipListSet  
CopyOnWriteArrayList  
CopyOnWriteArraySet  
CountDownLatch  
CountedCompleter  
CyclicBarrier  
DelayQueue  
Exchanger  
ExecutorCompletionService  
Executors  
ForkJoinPool  
ForkJoinTask  
ForkJoinWorkerThread  
FutureTask  
LinkedBlockingDeque  
LinkedBlockingQueue  
LinkedTransferQueue  
Phaser  
PriorityBlockingQueue

概述 软件包 类 使用 树 已过时的 索引 帮助

上一个 下一个 框架 无框架

概要: NESTED | 字段 | 构造方法 | 方法 详细信息: 字段 | 构造方法 | 方法

compact1, compact2, compact3

java.util.concurrent

**Class ConcurrentHashMap<K,V>**

java.lang.Object  
java.util.AbstractMap<K,V>  
java.util.concurrent.ConcurrentHashMap<K,V>

参数类型

K - 由该地图维护的键的类型

V - 映射值的类型

All Implemented Interfaces:

Serializable, ConcurrentMap<K, V>, Map<K, V>

public class ConcurrentHashMap<K,V>  
extends AbstractMap<K,V>  
implements ConcurrentMap<K,V>, Serializable

支持检索的完全并发性和更新的高预期并发性的哈希表。这个类服从相同功能规范如Hashtable，并且包括对应于每个方法的方法版本Hashtable。不过，尽管所有操作都是线程安全的，检索操作并不意味着锁定，并避免为防止所有访问的方式锁定整个表的任何支持。这个类可以在依赖于线程安全性的程序中与Hashtable完全互Hashtable，但不依赖于其同步细节。

检索操作（包括get）通常不阻止，因此可能与更新操作重叠（包括put和remove）。检索反映了最近完成的更新操作的结果。（正式地，对于给定密钥的更新操作发生之前发生与任何（非空关系）检索该键报告经更新的值。）对于聚合操作，比如putAll和clear，并发检索可能反映插入或删除只有一些条目。类似地，迭代器，分割器和枚举返回在反映迭代器/枚举创建过程中或之后反映哈希表状态的元素。他们不抛出ConcurrentModificationException。然而，迭代器被设计为一次只能由一个线程使用。请记住，骨科状态方法的结果，包括size，isEmpty和containsValue通常是有用的，只有当一个地图没有发生在其他线程并发更新。否则，这些方法的结果反映了可能足以用于监视或估计目的的瞬态状态，但不适用于程序控制。

当存在太多的冲突（即，具有不同的哈希码但是以表的大小为模数落入相同的时隙的密钥）时，该表被动态扩展，并且每个映射保持大致两个bin的预期平均效果（对应于0.75负载因素阈值调整大小）。由于映射被添加和删除，这个平均值可能会有很大差异，但是总的来说，这为哈希表保留了普遍接受的时间/空间权衡。然而，调整这个或任何其他类型的散列表可能是相对较慢的操作。如果可能，最好提供一个尺寸估计作为可选的InitialCapacity构造函数参数。附加的可选的loadFactor构造函数参数提供了另外的手段，通过指定在计算给定数量的元素时要分配的空间量时使用的表密度来定制初始表容量。此外，为了与此

```
package com.coding.collunsafe;
```

```
import java.util.Collections;  
import java.util.HashMap;  
import java.util.Map;  
import java.util.UUID;  
import java.util.concurrent.ConcurrentHashMap;
```

```
// 任何存在想修改JDK源码都是不可取的
```

```
// ConcurrentModificationException
```

```
// 并发下 HashMap 不安全
```

```
// 解决方案: Map<String, String> map = new ConcurrentHashMap<>();
```

```
public class UnsafeMap {
```

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

```
        // 在开发中会这样使用 HashMap 吗? 不会 一开始就知道 100大小的容量
```

```
        // 根据实际的业务设置初始值
```

```
        // 人生如程序，不是选择，就是循环，学习和总结十分重要！
```

```
        //Map<String, String> map = new HashMap<>();
```

```
        Map<String, String> map = new ConcurrentHashMap<>();
```

```
        // 加载因子，初始值
```

```
        // Map<String, String> map = new HashMap<>(100, 0.75);
```

```
        for (int i = 1; i <= 30; i++) {  
            new Thread()->{
```

```
                map.put(Thread.currentThread().getName(), UUID.randomUUID().toString().substring(  
0, 5));
```

```
                System.out.println(map);  
            },String.valueOf(i)).start();
```

```
        }
```

```
}  
}
```

## 7、Callable

线程创建方式 Thread、Runnable、Callable区别：

- 是否有返回值
- 是否跑出异常
- 方法不同，run () ， call ()

### 基础入门

#### 构造方法

##### Constructor and Description

**Thread()**

分配一个新的 Thread对象。

**Thread(Runnable target)**

分配一个新的 Thread对象。

**Thread(Runnable target, String name)**

分配一个新的 Thread对象。

**Thread(String name)**

分配一个新的 Thread对象。

**Thread(ThreadGroup group, Runnable target)**

分配一个新的 Thread对象。

**Thread(ThreadGroup group, Runnable target, String name)**

分配一个新的 Thread对象，使其具有 target作为其运行对象，具有指定的 name作为其名称，属于 group引用的线程组。

**Thread(ThreadGroup group, Runnable target, String name, long stackSize)**

分配一个新的 Thread对象，以便它具有 target作为其运行对象，将指定的 name正如其名，以及属于该线程组由称作 group，并具有指定的 栈大小。

**Thread(ThreadGroup group, String name)**

分配一个新的 Thread对象。

Thread 只认识==> Runnable

Callable

男生 Callable

女朋友 Runnable 搭桥，桥接模式

闺蜜 Thread

```
java.util.concurrent
```

### Interface RunnableFuture<V>

参数类型

V - 未来的 get方法返回的结果类型

All Superinterfaces:

Future <V>, Runnable

All Known Subinterfaces:

RunnableScheduledFuture <V>

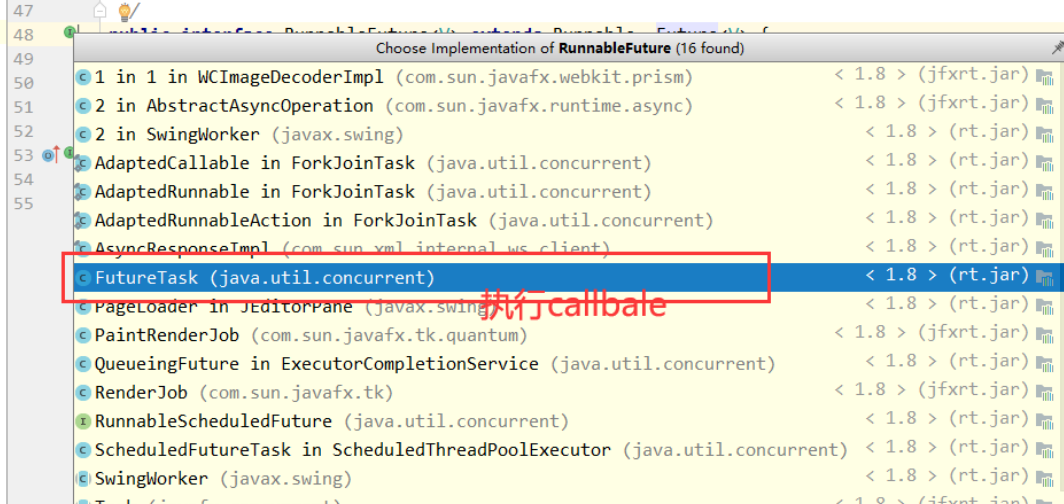
所有已知实现类:

FutureTask , SwingWorker

```

45 * @author Doug Lea
46 * @param <V> The result type returned by this Future's {@code get} method
47

```



#### 构造方法

##### Constructor and Description

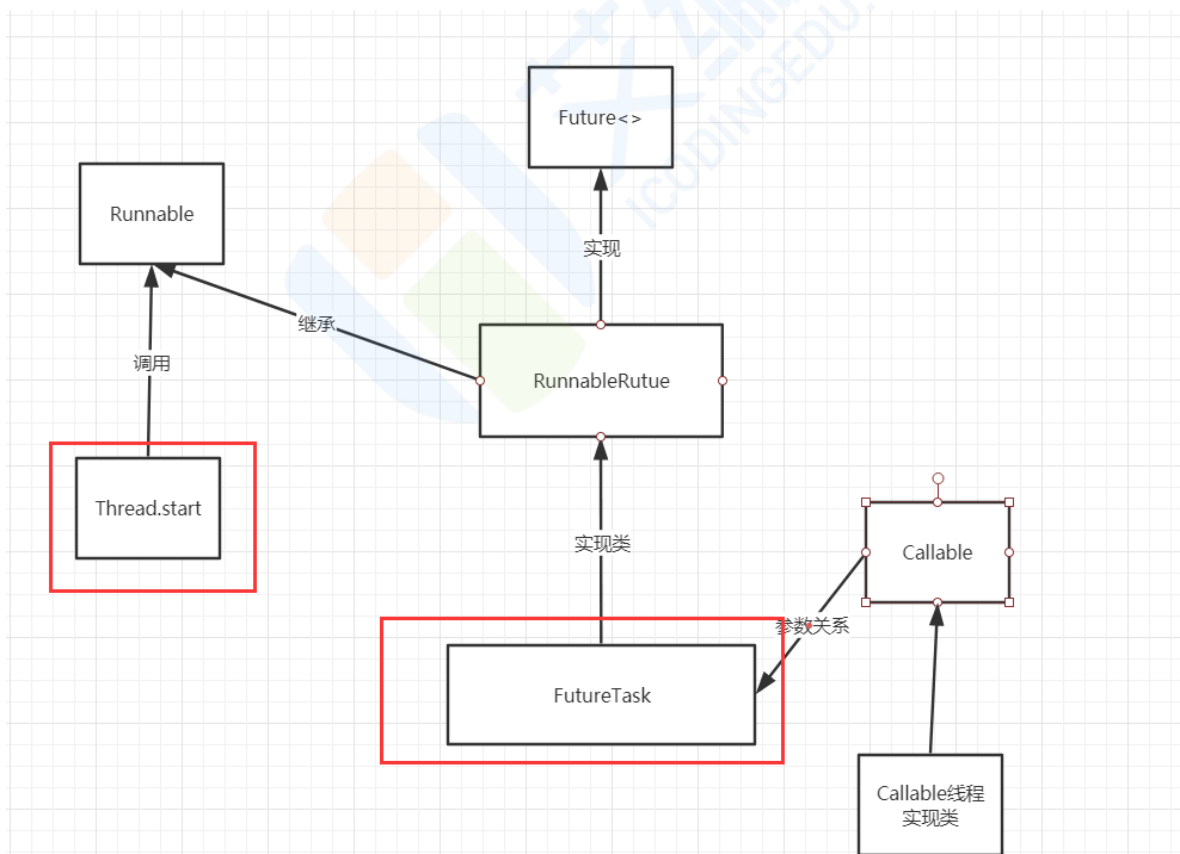
**FutureTask(Callable<V> callable)**

创建一个 FutureTask，它将在运行时执行给定的 Callable。

万能的桥

**FutureTask(Runnable runnable, V result)**

创建一个 FutureTask，将在运行时执行给定的 Runnable，并安排 get 将在成功完成后返回给定的结果。



```

package com.coding.callabledemo;

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

```

```

// 练武不练功，到老一场空
// API 工程师，只会用，不会分析~
public class Test1 {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        // Thread (Runnable)
        // Thread (RunnableFuture)
        // Thread (FutureTask)

        MyThread myThread = new MyThread();
        FutureTask task = new FutureTask(myThread); // 适配类

        new Thread(task, "A").start(); // 执行线程

        System.out.println(task.get()); // 获取返回值， get()

    }
}

class MyThread implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        System.out.println("end");
        return 1024;
    }

}

```

## Callable 细节

缓存

结果获取会阻塞

```

package com.coding.callabledemo;

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

// 练武不练功，到老一场空
// API 工程师，只会用，不会分析~
public class Test1 {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        // Thread (Runnable)
        // Thread (RunnableFuture)
        // Thread (FutureTask)

```

```

MyThread myThread = new MyThread();
FutureTask task = new FutureTask(myThread); // 适配类

// 会打印几次 end
new Thread(task, "A").start(); // 执行线程
new Thread(task, "B").start(); // 执行线程。细节1: 结果缓存! 效率提高N倍

System.out.println(task.get()); // 获取返回值, get()

// 细节2: task.get() 获取值的方法一般放到最后, 保证程序平稳运行的效率, 因为他会阻塞等待结果产生!
// 线程是一个耗时的线程, 不重要!

}
}

class MyThread implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        System.out.println("end");

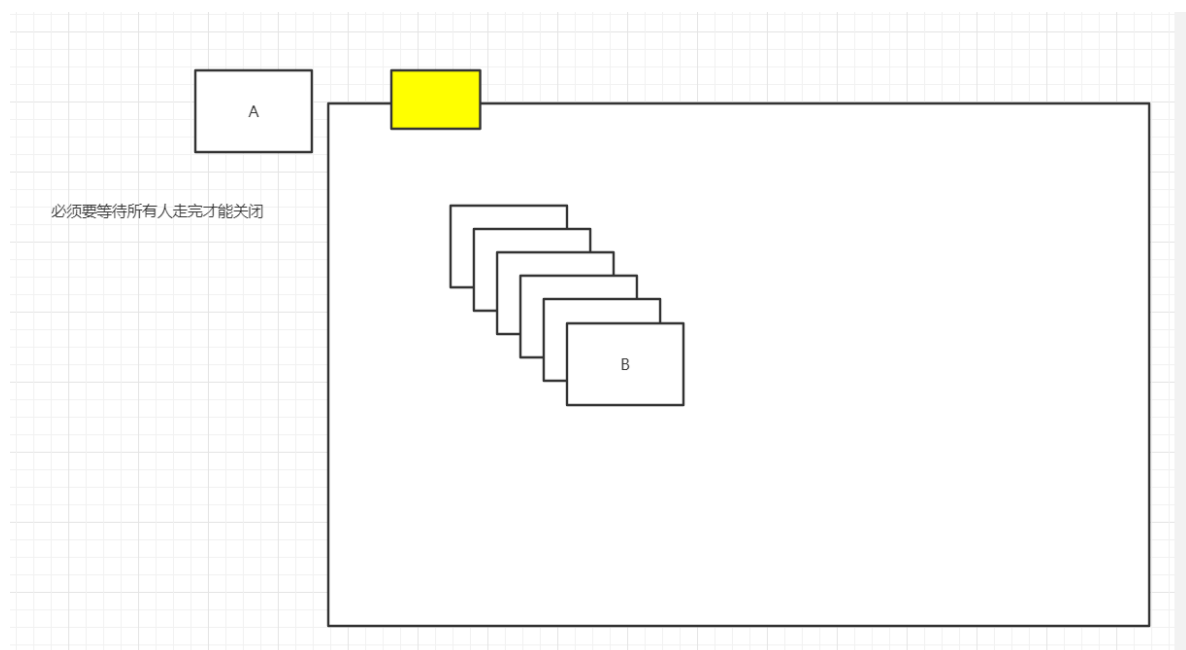
        TimeUnit.SECONDS.sleep(3);

        return 1024;
    }
}

```

## 8、常用辅助类

### 8.1、CountDownLatch



```

package com.coding.demo03;

import java.util.concurrent.CountDownLatch;

// 程序如果不加以生活的理解再加上代码的测试，你就算不会
public class CountDownLatchDemo {

    // 有些任务是不得不阻塞的 减法计数器
    public static void main(String[] args) throws InterruptedException {

        CountDownLatch countDownLatch = new CountDownLatch(6); // 初始值

        for (int i = 1; i <= 6 ; i++) {
            new Thread()->{
                System.out.println(Thread.currentThread().getName()+"Start");
                // 出去一个人计数器就 -1
                countDownLatch.countDown();
            }.start();
        }

        countDownLatch.await(); // 阻塞等待计数器归零
        // 阻塞的操作：计数器 num++
        System.out.println(Thread.currentThread().getName()+"==END");

    }

    // 结果诡异的吗，达不到预期的 Main end 在最后一个
    public static void test1(){
        for (int i = 1; i <= 6 ; i++) {
            new Thread()->{
                System.out.println(Thread.currentThread().getName()+"Start");
            }.start();
        }
        System.out.println(Thread.currentThread().getName()+"End");
    }

}

```

CountDownLatch countDownLatch = new CountDownLatch(6);

countDownLatch.countDown(); 出去一个人计数器就 -1

countDownLatch.await(); 阻塞等待计数器归零

## 8.2、CyclicBarrier

```

package com.coding.demo03;

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

// CyclicBarrier 栅栏 加法计数器
public class CyclicBarrierDemo {
    public static void main(String[] args) {

```

```

// 集齐7个龙珠召唤神龙 ++ 1

// public CyclicBarrier(int parties, Runnable barrierAction)
// 等待cyclicBarrier计数器满, 就执行后面的Runnable, 不满就阻塞
CyclicBarrier cyclicBarrier = new CyclicBarrier(8, new Runnable() {
    @Override
    public void run() {
        System.out.println("神龙召唤成功! ");
    }
});

for (int i = 1; i <= 7; i++) {
    final int temp = i;
    new Thread()->{
        System.out.println(Thread.currentThread().getName()+"收集了
第"+temp+"颗龙珠");

        try {
            cyclicBarrier.await(); // 等待 阻塞
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }

        }, String.valueOf(i)).start();
}

}
}

```

## 8.3、Semaphore

Semaphore: 信号灯。抢位置

```

package com.coding.demo03;

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

// 抢车位
public class SemaphoreDemo {
    public static void main(String[] args) {
        // 模拟6个车, 只有3个车位
        Semaphore semaphore = new Semaphore(3); // 3个位置

        for (int i = 1; i <= 6; i++) {
            new Thread()->{
                // 得到车位
                try {
                    semaphore.acquire(); // 得到

```



```

        System.out.println(Thread.currentThread().getName()+"抢到了车
位");

        TimeUnit.SECONDS.sleep(3);
        System.out.println(Thread.currentThread().getName()+"离开了车
位");

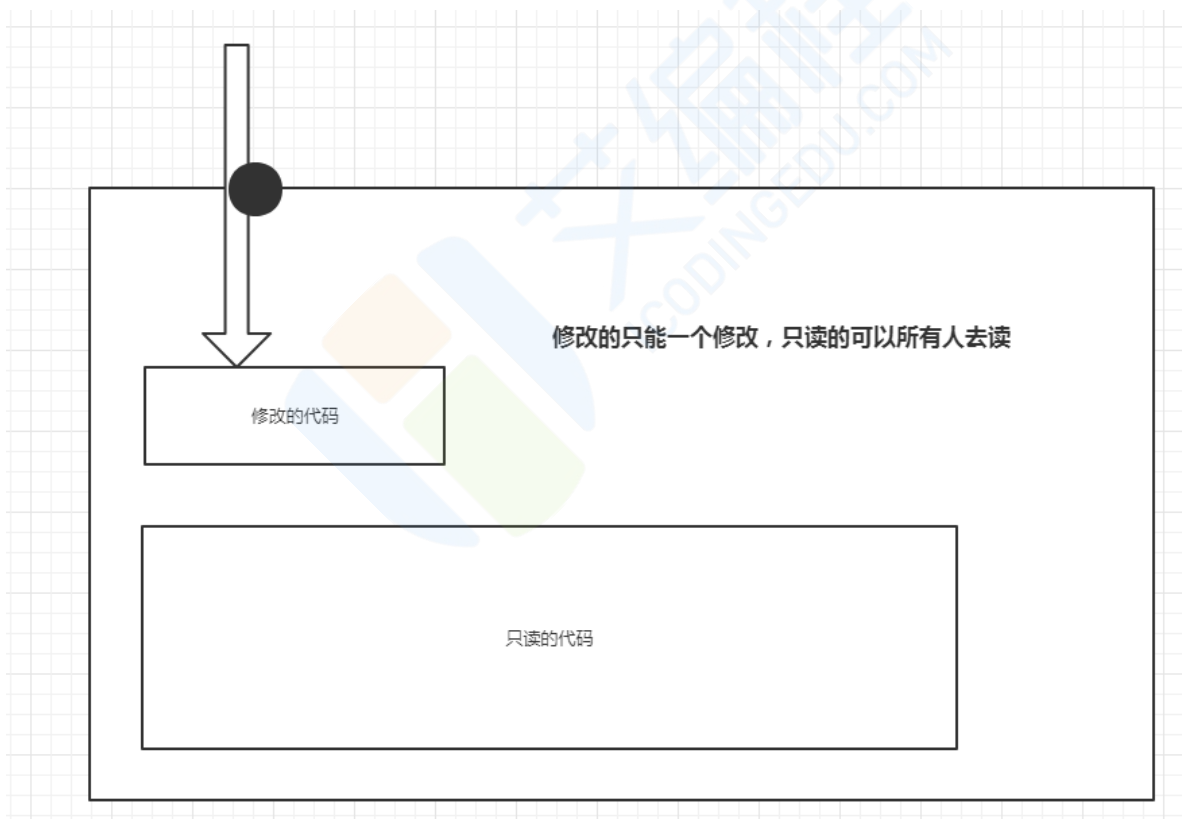
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release(); // 释放位置
        }

        },String.valueOf(i)).start();
    }

}
}

```

## 9、读写锁



java.security.spec  
java.sql  
java.text  
java.text.spi  
java.time  
java.time.chrono  
java.time.format  
java.time.temporal  
java.time.zone  
java.util  
java.util.concurrent  
java.util.concurrent.atomic  
java.util.concurrent.locks  
java.util.function  
java.util.jar  
java.util.logging  
java.util.regex  
java.util.zip

java.util.concurrent.locks

Interfaces  
Condition  
Lock  
ReadWriteLock  
Classes  
AbstractOwnableSynchronizer  
AbstractQueuedLongSynchronizer  
AbstractQueuedSynchronizer  
LockSupport  
ReentrantLock  
ReentrantReadWriteLock  
ReentrantReadWriteLock.ReadLock  
ReentrantReadWriteLock.WriteLock  
StampedLock

概述 软件包 类 使用 例 已过时的 索引 帮助

上一个 下一个 框架 无框架

概要: 嵌套 | 字段 | 构造方法 | 方法 详细信息: 字段 | 构造方法 | 方法

compact1, compact2, compact3  
java.util.concurrent.locks

Interface ReadWriteLock

所有已知实现类:  
ReentrantReadWriteLock

public interface ReadWriteLock

A ReadWriteLock维护一对关联的locks, 一个用于只读操作, 一个用于写入。 read lock可以由多个阅读器线程同时进行, 只要没有作者。 write lock是独家的。

所有ReadWriteLock实现必须保证的存储设备同步应WriteLock操作(如在指定Lock接口)也保持相对于所述相关联的readLock。 也就是说, 一个线程成功获取读锁将会看到在之前发布的写锁所做的所有更新。

读写锁允许访问共享数据时的并发性高于互斥锁所允许的并发性。 它利用了这样一个事实: 一次只有一个线程( 写入线程) 可以修改共享数据, 在许多情况下, 任何数量的线程都可以同时读取数据( 因此, 读取线程)。 从理论上讲, 通过使用读写锁允许的并发性增加将导致性能改进超过使用互斥锁。 实际上, 并发性增加只能在多处理器上完全实现, 然后只有在共享数据的访问模式是合适的时才可以。

读写锁是否会提高使用互斥锁的性能取决于数据被读取的频率与被修改的频率相比, 读取和写入操作的持续时间以及数据的争用 - 即是, 将尝试同时读取或写入数据的线程数。 例如, 最初填充数据的集合, 然后经常被修改的频繁搜索( 例如某种目录) 是使用读写锁的理想候选。 然而, 如果更新变得频繁, 那么数据的大部分时间将被专门锁定, 并且并发性增加很少。 此外, 如果读取操作太短, 则读写锁实现( 其本身比互斥锁更复杂) 的开销可以支配执行成本, 特别是因为许多读写锁实现仍将序列化所有线程通过小部分代码。 最终, 只有剖析和测量将确定使用读写锁是否适合您的应用程序。

虽然读写锁的基本操作是直接的, 但是执行必须做出许多策略决策, 这可能会影响给定应用程序中读写锁定的有效性。 这些政策的例子包括:

- 在写入器放入写入锁定时, 确定在读取器和写入器都在等待时是否授予读取锁定或写入锁定。 作家偏好是常见的, 因为写作预计会很短, 很少见。 读者喜好不常见, 因为如果读者经常和长期的预期, 写作可能导致增长的延迟。 公平的或"按顺序"的实现也是可能的。
- 确定在读者处于活动状态并且写入器正在等待时请求读取锁定的读取器是否被授予读取锁定。 读者的偏好可以无限期地拖延作者, 而对作者的偏好可以减少并发的潜力。
- 确定锁是否可重入: 一个具有写锁的线程是否可以重新获取? 持有写锁可以获取读锁吗? 读锁本身是否可重入?
- 写入锁可以降级到读锁, 而不允许插入写入者? 读锁可以升级到写锁, 优先于其他等待读者或作者吗?

在评估应用程序的给定实现的适用性时, 应考虑所有这些方面。

网站地址: 网站地址  
我要纠错: 修正翻译内容  
QQ群: 864725  
安卓帮助文档

jdk  
中

读锁（共享锁）：这个锁可以被多个线程持有！

写锁（独占锁）：这个锁一次只能被一个线程占用！

## 代码

```
package com.coding.rwdemo;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

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

        MyCache2 myCache = new MyCache2();

        // 多个线程同时进行读写
        // 五个线程在写 线程是CPU调度的
        for (int i = 1; i < 5; i++) {
            final int temp = i;
            new Thread()->{
                myCache.put(temp+ "", temp+ "");
            },String.valueOf(i)).start();
        }

        // 五个线程在读
        for (int i = 1; i < 5; i++) {
            final int temp = i;
            new Thread()->{
                myCache.get(temp+ "");
            },String.valueOf(i)).start();
        }

    }
}
```

```

}

// 线程操作资源类，存在问题的
class MyCache{

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

    // 没有加读写锁的时候，第一个线程还没有写入完成，可能会存在其他写入~

    // 写。独占
    public void put(String key,String value){
        System.out.println(Thread.currentThread().getName()+"写入"+key);
        map.put(key,value);
        // 存在别的线程插队
        System.out.println(Thread.currentThread().getName()+"写入完成");
    }

    // 读
    public void get(String key){
        System.out.println(Thread.currentThread().getName()+"读取"+key);
        Object result = map.get(key);
        System.out.println(Thread.currentThread().getName()+"读取结果: "+result);
    }
}

// 线程操作资源类，存在问题的
class MyCache2{

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

    // ReadWriteLock --> ReentrantReadWriteLock    lock不能区分读和写
    // ReentrantReadWriteLock 可以区分读和写，实现更加精确的控制
    // 读写锁
    private ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();

    // 写。独占
    public void put(String key,String value){
        // lock.lock 加锁
        readWriteLock.writeLock().lock();
        try {
            System.out.println(Thread.currentThread().getName()+"写入"+key);
            map.put(key,value);
            // 存在别的线程插队
            System.out.println(Thread.currentThread().getName()+"写入完成");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            readWriteLock.writeLock().unlock(); // lock.unlock();
        }
    }

    // 多线程下尽量加锁！

    // 读
    public void get(String key){
        readWriteLock.readLock().lock();
    }
}

```

```

        try {
            System.out.println(Thread.currentThread().getName()+"读取"+key);
            Object result = map.get(key);
            System.out.println(Thread.currentThread().getName()+"读取结果: "+result);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            readwriteLock.readLock().unlock();
        }
    }
}

```

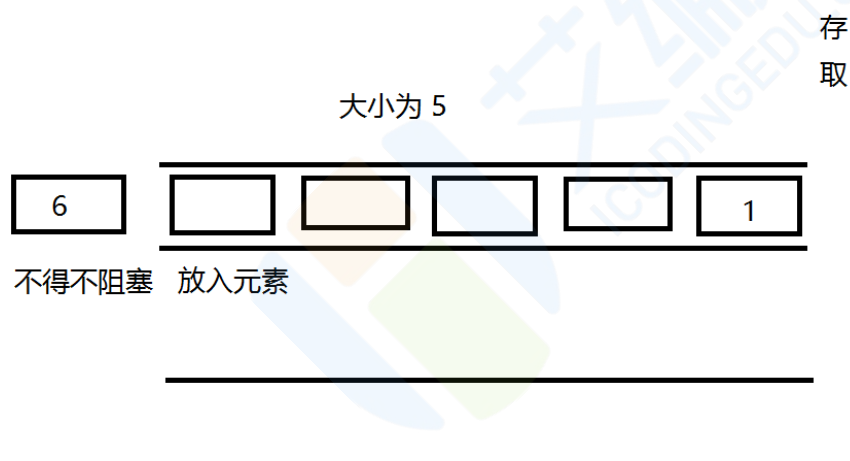
读写分离，提高效率~判断业务中那些代码是只读的业务，不要去锁这些业务~

## 10、阻塞队列

阻塞队列

队列：排队 FIFO

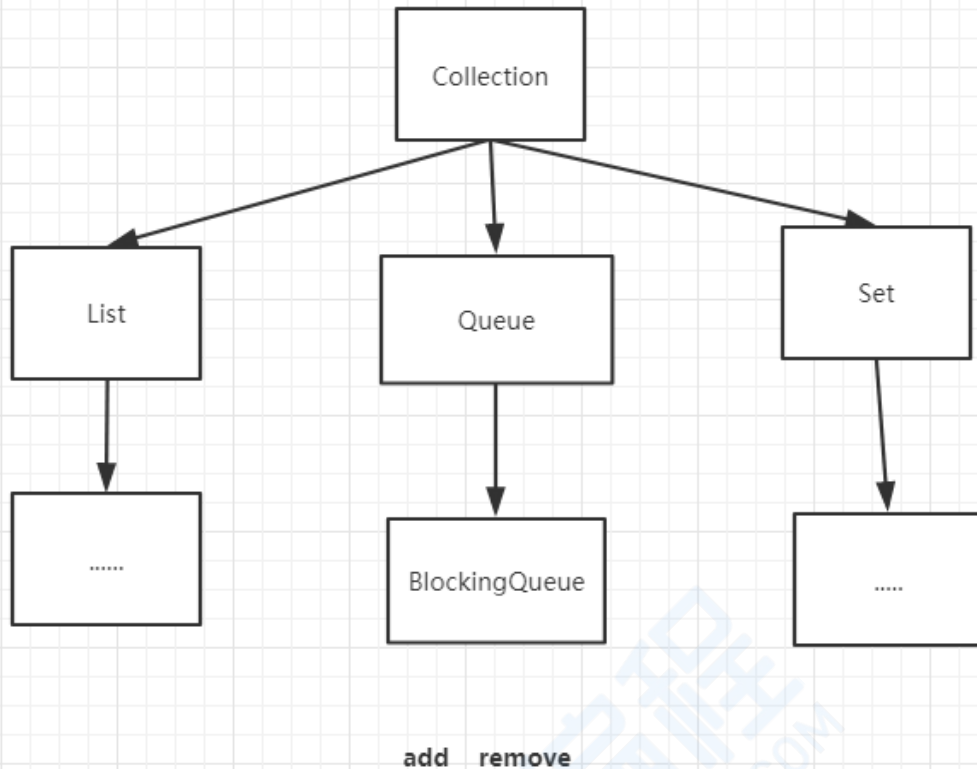
阻塞：必须要阻塞、不得不阻塞~



取出、加入队列为空  
也会阻塞等待队列中  
产生新的值

接口架构图

## 学习方法：新的东西，一定要和已经学过的结合起来理解



java.swing.JOptionPane

java.sql

java.text

java.text.spi

java.time

java.time.chrono

java.time.format

java.time.temporal

java.time.zone

java.util

java.util.concurrent

java.util.concurrent.atomic

java.util.concurrent.locks

java.util.function

java.util.jar

java.util.logging

java.util.regex

java.util.zip

java.util.concurrent

Interfaces

BlockingDeque

BlockingQueue

Callable

CompletableFuture

CompletableFuture.Async

CompletionService

CompletionStage

ConcurrentHashMap

ConcurrentNavigableMap

Delayed

Executor

ExecutorService

ForkJoinPool

ForkJoinPool.ForkJoinWorkerThreadFactory

ForkJoinPool.ManagedBlocker

Future

RejectedExecutionHandler

Runnable

RunnableScheduledFuture

ScheduledExecutorService

ScheduledFuture

ThreadFactory

TransferQueue

Classes

AbstractExecutorService

ArrayBlockingQueue

CompletableFuture

ConcurrentHashMap

ConcurrentHashMap.KeySetView

ConcurrentLinkedDeque

ConcurrentLinkedQueue

概述 软件包 类 使用 例 已过时的 索引 帮助

上一个 下一个 框架 无框架

概要 | 字段 | 构造方法 | 方法 | 详细信息 | 字段 | 构造方法 | 方法

compact1, compact2, compact3

java.util.concurrent

阻塞队列

Interface BlockingQueue<E>

参数类型

E - 此集合中保存的元素类型

All Superinterfaces:

Collection<E>, Iterable<E>, Queue<E>

All Known Subinterfaces:

BlockingDeque<E>, TransferQueue<E>

所有已知实现类:

ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue

同步队列

public interface BlockingQueue<E>

extends Queue<E>

A Queue另外支持在检索元素时等待队列为非空的操作，并且在存储元素时等待队列中的空间变得可用。

BlockingQueue方法有四种形式，具有不同的操作方式。不能立即满足，但可能在将来的某个时间点满足：一个抛出异常，第二个返回一个特殊值（null或false，具体取决于操作），第三个将无限期地阻止当前线程，直到操作成功为止，而第四个程序块在放弃之前只有给定的最大时限。 这些方法总结在下表中：

Summary of BlockingQueue methods Throws exception Special value Blocks Times out Insert add(e) offer(e) put(e) offer(e, time, unit) Remove remove() poll()

A BlockingQueue不接受null元素。实现抛出NullPointerException上尝试add, put或offer一个null。A null用作哨兵值以指示poll操作失败。

A BlockingQueue可能是容量有限的。 在任何给定的时间它可能有一个remainingCapacity超过其中没有额外的元素可以put没有阻止。 没有任何内在容量限制的A BlockingQueue总是报告剩余容量为Integer.MAX\_VALUE。

BlockingQueue实现被设计为主要用于生产者-消费者队列，但另外支持Collection接口。 因此，例如，可以使用remove(x)从队列中删除任意元素。 然而，这样的操作通常不经常非常有效地执行，并且仅用于偶尔使用，例如当排队消息被取消时。

BlockingQueue实现是线程安全的。 所有排队方法使用内部锁或其他形式的并发控制在原子上实现其效果。 然而，大量的Collection操作addAll, containsAll, retainAll和removeAll不一定原子上除非在实现中另有规定执行。 因此有可能，例如，为addAll(c)到只增加一些元素在后失败（抛出异常）c。

jdk 8 中 对 阻塞 版

网站地址1 网站地址2  
我要纠错——修正翻译内容。  
QQ群：86472519  
安卓帮助文档。

### API 的使用

四组API：任何一个方法存在，就一定有对应的业务场景，应为都是在生活中或者编码中遇到了困难，才写的

方法	抛出异常	返回特殊值	一直阻塞	超时退出
插入 存	add	offer	put ()	offer(e,time)
移除 取	remove	poll	take()	poll(e,time)
检查队首	element	peek	-	-

### 第一组：抛出异常

```

package com.coding.blocking;

import java.util.ArrayList;
import java.util.concurrent.ArrayBlockingQueue;

public class Test1 {
    public static void main(String[] args) {
        // 队列的大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3); // 阻塞队
        列

        // add返回布尔值
        System.out.println(blockingQueue.add("a"));
        System.out.println(blockingQueue.add("b"));
        System.out.println(blockingQueue.add("c"));
        // System.out.println(blockingQueue.add("d")); //
        java.lang.IllegalStateException: Queue full

        System.out.println(blockingQueue.element());

        System.out.println(blockingQueue.remove()); // a

        System.out.println(blockingQueue.element());

        blockingQueue.remove();

        System.out.println(blockingQueue.element());

        System.out.println(blockingQueue.remove()); // b
        System.out.println(blockingQueue.remove()); // c
        System.out.println(blockingQueue.remove()); //
        java.util.NoSuchElementException

    }
}

```

### 第二组：没有异常

```

package com.coding.blocking;

import java.util.concurrent.ArrayBlockingQueue;

```

```
// 通常!
public class Test2 {

    public static void main(String[] args) {

        // 队列的大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3); // 阻塞队
        列

        System.out.println(blockingQueue.offer("a"));
        System.out.println(blockingQueue.offer("b"));
        System.out.println(blockingQueue.offer("c"));
        // 等待（一直等待，超时就不等你）
        // System.out.println(blockingQueue.offer("d")); // false 我们通常不希望代
        码报错！这时候就使用offer

        System.out.println(blockingQueue.peek()); // 查看队首

        System.out.println(blockingQueue.poll());
        System.out.println(blockingQueue.poll());
        System.out.println(blockingQueue.poll());
        System.out.println(blockingQueue.poll()); // null

        // System.out.println(blockingQueue.peek()); // 查看队首 null

    }

}
```

### 第三组：一直阻塞

```
package com.coding.blocking;

import java.util.concurrent.ArrayBlockingQueue;

public class Test3 {

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

        // 队列的大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3); // 阻塞队
        列

        // 一直阻塞。超过3秒我就不等了，业务必须要做！
        blockingQueue.put("a");
        blockingQueue.put("b");
        blockingQueue.put("c");
        // blockingQueue.put("d");

        System.out.println(blockingQueue.take());
        System.out.println(blockingQueue.take());
        System.out.println(blockingQueue.take());
        System.out.println(blockingQueue.take()); // 阻塞等待拿出元素

    }

}
```

超时就退出:

```
package com.coding.blocking;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.TimeUnit;

public class Test4 {
    public static void main(String[] args) throws InterruptedException {
        // 队列的大小
        ArrayBlockingQueue blockingQueue = new ArrayBlockingQueue<>(3); // 阻塞队
        列

        // 设置超时的时间
        blockingQueue.offer("a");
        blockingQueue.offer("b");
        blockingQueue.offer("c");
        // 超过3秒就不等待了
        blockingQueue.offer("d", 3L, TimeUnit.SECONDS); // 不让他一直等待, 然后也不想返
        回false, 设置等待时间

        System.out.println(blockingQueue.poll()); // a
        System.out.println(blockingQueue.poll()); // b
        System.out.println(blockingQueue.poll()); // c
        System.out.println(blockingQueue.poll(3L, TimeUnit.SECONDS)); // 阻塞
    }
}
```

## SynchronousQueue 同步队列

不存储元素、队列空的

每一个 put 操作。必须等待一个take。否则无法继续添加元素!

```
package com.coding.blocking;

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

// 同步队列
// 每一个 put 操作。必须等待一个take。否则无法继续添加元素!
public class Test5 {
    public static void main(String[] args) {
        // 不用写参数!
        SynchronousQueue<String> queue = new SynchronousQueue<>();

        new Thread(()->{
            try {
                System.out.println(Thread.currentThread().getName()+"put 1");
                queue.put("1");
                System.out.println(Thread.currentThread().getName()+"put 2");
                queue.put("2");
                System.out.println(Thread.currentThread().getName()+"put 3");
                queue.put("3");
            } catch (InterruptedException e) {
            }
        }
    }
}
```



```

        e.printStackTrace();
    }
}, "A").start();

new Thread()->{
    try {
        TimeUnit.SECONDS.sleep(3);

        System.out.println(Thread.currentThread().getName()+queue.take());
        TimeUnit.SECONDS.sleep(3);

        System.out.println(Thread.currentThread().getName()+queue.take());
        TimeUnit.SECONDS.sleep(3);

        System.out.println(Thread.currentThread().getName()+queue.take());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}, "B").start();

    }
}

```

## 11、线程池(重点)

三大方法、7大参数、拒绝策略、优化配置

### 池化技术

程序运行的本质：占用系统资源，CPU/磁盘网络进行使用！我们希望可以高效的使用！池化技术就是演进出来的。

白话：提前准备一些资源、以供使用！

线程池、连接池、内存池、对象池.....

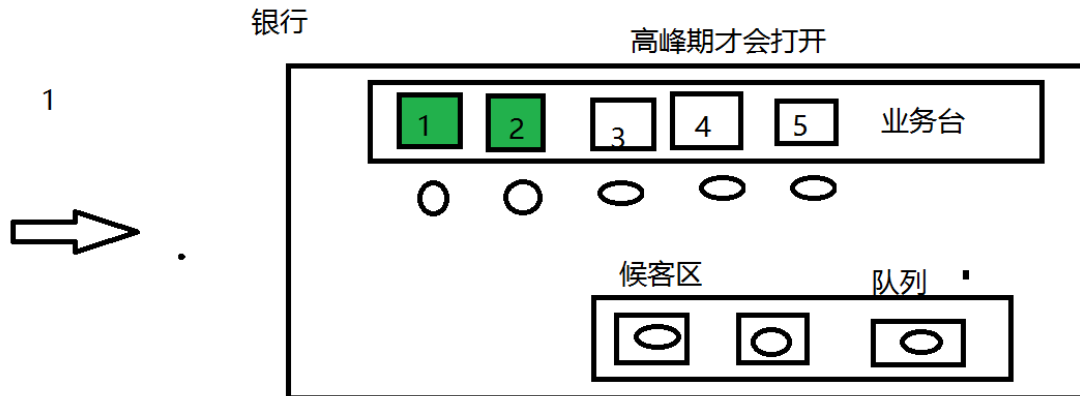
线程的创建和销毁，数据库的连接和断开都十分浪费资源。

-

minSize

maxSize

弹性访问！保证系统运行的效率



core 核心大小 2  
queue 队列 3  
maxSize 5  
最多可以存在的人：maxSize + queue

### 为什么使用线程池

10年单核电脑，假的多线程！交替速度块，现在是多核多CPU，各自的线程跑在独立的CPU上，不用切换，效率会高~

CPU密集型，IO 密集型；

### 线程池的优势：

控制运行的线程的数量，处理的时候可以把一些任务放入队列；

特点：实现线程的复用！控制最大并发数！

### 线程池的三大方法

```
compact1, compact2, compact3  
java.util.concurrent
```

#### Interface Executor

All Known Subinterfaces:

ExecutorService , ScheduledExecutorService

所有已知实现类：

AbstractExecutorService , ForkJoinPool , ScheduledThreadPoolExecutor , ThreadPoolExecutor

public interface Executor

### 使用、调度

执行提交的对象Runnable任务：该界面提供了一种将任务提交从每个任务的运行机制分解的方式，包括线程使用，调度等的Executor。通常使用Executor而不是显式创建线程。例如，不是一组任务调用new Thread(new RunnableTask()).start()，您可以使用：

```
Executor executor = anExecutor;  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());  
...
```

但是，Executor接口并不严格要求执行是异步的。在最简单的情况下，执行程序可以立即在调用者的线程中运行提交的任务：

```
class DirectExecutor implements Executor { public void execute(Runnable r) { r.run(); } }
```

更典型的是，除了调用者的线程之外，任务在一些线程中执行。下面的执行器为每个任务生成一个新的线程。

```
class ThreadPerTaskExecutor implements Executor { public void execute(Runnable r) { new Thread(r).start(); } }
```

许多Executor实现如何和何时安排任务施加某种限制。下面的执行器将任务的提交序列化到第二个执行器，说明复合执行器。

```
class SerialExecutor implements Executor { final Queue<Runnable> tasks = new ArrayDeque<Runnable>(); final Executor executor; Runnable
```

该包中提供的Executor实现了ExecutorService，这是一个更广泛的界面。ThreadPoolExecutor类提供了一个可扩展的线程池实现。Executors类为这些执行人员提供了方便的工厂方法。

内存一致性效果：操作在一个线程提交之前Runnable对象到Executor happen-before其执行开始，也许在另一个线程。

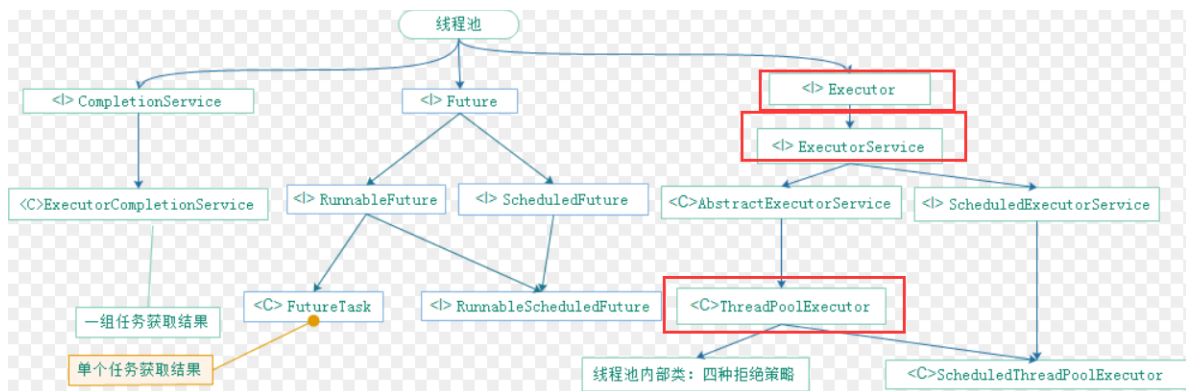
从以下版本开始。

我要纠错——修正翻译

QQ群：86472

安卓助手

jdk  
中  
英  
~



```
package com.coding.pool;

import java.util.Arrays;
import java.util.Collections;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * Executors.
 * ExecutorService.execute
 */
public class Test1 {
    public static void main(String[] args) {
        // 平时我们创建一些类使用工具类操作 s
        // 总数可以管理

        // 线程池 Executors原生三大方法
        // ExecutorService threadpool1 = Executors.newFixedThreadPool(5); // 固定大小
        // ExecutorService threadpool2 = Executors.newCachedThreadPool(); // 可以弹性伸缩的线程池，遇强则强
        ExecutorService threadpool3 = Executors.newSingleThreadExecutor(); // 只有一个

        try {
            // 10个线程，会显示几个线程~
            for (int i = 1; i <= 100; i++) {
                // 线程池，执行线程
                threadpool3.execute(() -> {
                    System.out.println(Thread.currentThread().getName() + "
running...");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 线程池关闭
            threadpool3.shutdown();
        }
    }
}
```

分析三个方法的源码

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(5, 5,
                                   0L, TimeUnit.MILLISECONDS,
                                   new LinkedBlockingQueue<Runnable>());
}

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

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

ThreadPoolExecutor核心方法:

```
public ThreadPoolExecutor(int corePoolSize, // 核心池子的大小
                          int maximumPoolSize, // 池子的最大大小
                          long keepAliveTime, // 空闲线程的保留时间
                          TimeUnit unit, // 时间单位
                          BlockingQueue<Runnable> workQueue, // 队列
                          ThreadFactory threadFactory, // 线程工厂, 不修改! 用来创建
线程
                          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;
}
```

思考: 工作中怎么使用线程池? 只能够自己根据业务情况去自定义线程池的大小策略, 禁止使用 Executors。

4. 【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：

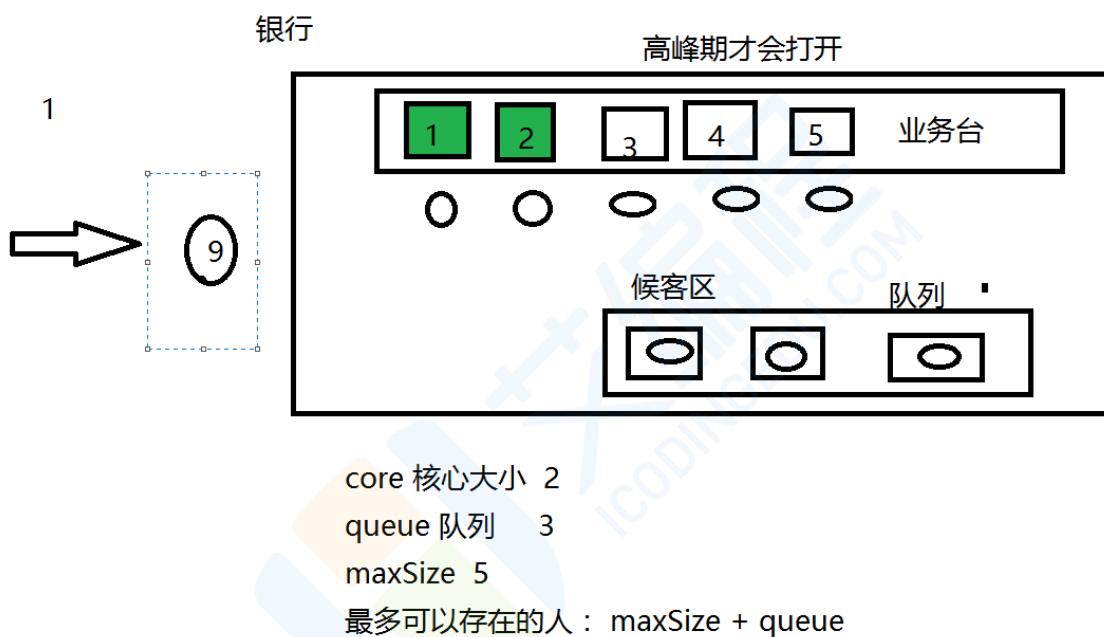
1) `FixedThreadPool` 和 `SingleThreadPool`:

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。

## ThreadPoolExecutor 底层工作原理



```
package com.coding.pool;

import java.util.concurrent.*;

public class Test2 {
    public static void main(String[] args) {
        ExecutorService threadPool = new ThreadPoolExecutor(
            2,
            5, // 线程池最大大小5
            2L,
            TimeUnit.SECONDS, // 超时回收空闲的线程
            new LinkedBlockingDeque<>(3), // 根据业务设置队列大小，队列大小一定要
            Executors.defaultThreadFactory(), // 不用变
            new ThreadPoolExecutor.CallerRunsPolicy() // 拒绝策略
        );

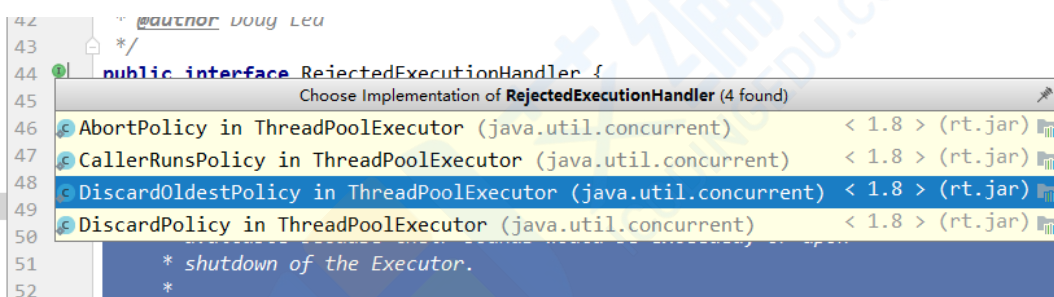
        // 拒绝策略说明：
        // 1. AbortPolicy (默认的：队列满了，就丢弃任务抛出异常！)
        // 2. CallerRunsPolicy (哪来的回哪去？谁叫你来的，你就去哪里处理)
        // 3. DiscardOldestPolicy (尝试将最早进入对立与的人任务删除，尝试加入队列)
```

```
// 4. DiscardPolicy (队列满了任务也会丢弃,不抛出异常)

try {
    // 队列 RejectedExecutionException 拒绝策略
    for (int i = 1; i <= 10; i++) {
        // 默认在处理
        threadPool.execute()->{
            system.out.println(Thread.currentThread().getName()+"
running....");
        };
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    threadPool.shutdown();
}

}
```

四种拒绝策略:



流程图:



线程池用哪个? 生产中如何设置合理参数

在工作中, 我们不会使用Executors, 自定义根据业务来定义线程池!

原因:

- 1、OOM. 默认大小 integer最大值
- 2、阿里巴巴开发手册说的

==**注意点：最大参数该如何设置？讲究？**==

CPU 密集型：CPU设置，每一次都要去写吗？

IO 密集型：磁盘读写、 一个线程在IO操作的时候、另外一个线程在CPU中跑，造成CPU空闲。

最大线程数应该设置为 IO任务数！ 大文件读写耗时！单独的线程让他慢慢跑。

```
package com.coding.pool;

import java.util.concurrent.*;

public class Test2 {
    public static void main(String[] args) {

        // 代码级别的
        System.out.println(Runtime.getRuntime().availableProcessors());

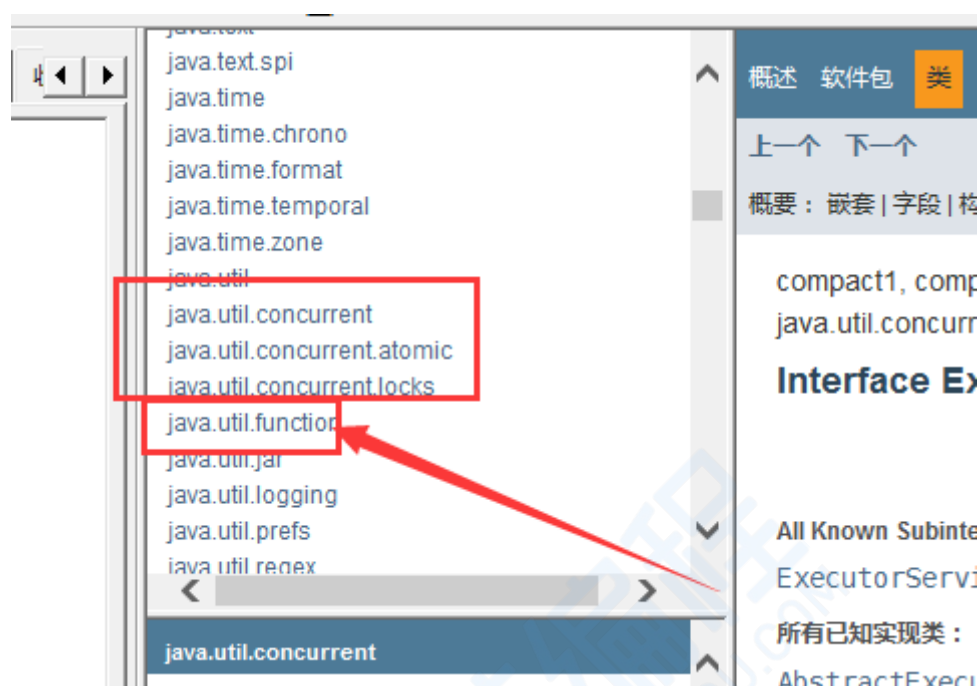
        ExecutorService threadPool = new ThreadPoolExecutor(
            2,
            Runtime.getRuntime().availableProcessors(), // 线程池最大大小5
            2L,
            TimeUnit.SECONDS, // 超时回收空闲的线程，假设超过了指定的时间，这个最大
            // 的线程就不被
            // 设置
            new LinkedBlockingDeque<>(3), // 根据业务设置队列大小，队列大小一定要
            Executors.defaultThreadFactory(), // 不用变
            new ThreadPoolExecutor.CallerRunsPolicy() //拒绝策略
        );

        // 拒绝策略说明：
        // 1. AbortPolicy （默认的：队列满了，就丢弃任务抛出异常！）
        // 2. CallerRunsPolicy （哪来的回哪去？ 谁叫你来的，你就去哪里处理）
        // 3. DiscardOldestPolicy （尝试将最早进入对立与的人任务删除,尝试加入队列）
        // 4. DiscardPolicy （队列满了任务也会丢弃,不抛出异常）

        try {
            // 队列 RejectedExecutionException 拒绝策略
            for (int i = 1; i <= 10; i++) {
                // 默认在处理
                threadPool.execute()->{
                    System.out.println(Thread.currentThread().getName()+"
running....");
                };
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPool.shutdown();
        }

    }
}
```

## 12、四大函数式接口







函数型接口，有一个输入，有一个输出

```
    * @param <R> the type of the result of the function
    *
    * @since 1.8
    */
    @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);
    }
```

传入一个参数T，返回一个结果R

```
package com.coding.function4;

import java.util.function.Function;

public class Demo01 {
    public static void main(String[] args) {
```

```

        // new Runnable(); ()-> {}

//
//      Function<String,Integer> function = new Function<String,Integer>() {
//      @Override // 传入一个参数，返回一个结果
//      public Integer apply(String o) {
//      System.out.println("into");
//      return 1024;
//      }
//      };

// 链式编程、流式计算、lambda表达式
Function<String,Integer> function = s->{return s.length();};
System.out.println(function.apply("abc"));

}
}

```

断定型接口，有一个输入参数，返回只有布尔值。

```

*
* <p>This is a <a href="package-summary.html">functional interface</a>
* whose functional method is {@link #test(Object)}.
*
* @param <T> the type of the input to the predicate
*
* @since 1.8
*
* @FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     *         otherwise {@code false}
     */
    boolean test(T t);

    /**
     * Returns a composed predicate that represents a short-circuiting logical

```

有且只有一个参数，返回值是boolean型

```

package com.coding.function4;

import java.util.function.Predicate;

public class Demo02 {
    public static void main(String[] args) {
        //      Predicate<String> predicate = new Predicate<String>(){
        //      @Override
        //      public boolean test(String o) {
        //      if (o.equals("abc")){
        //      return true;
        //      }
        //      return false;
        //      }
        //      };

        Predicate<String> predicate = s->{return s.isEmpty();};
        System.out.println(predicate.test("abced"));

    }
}

```

消费型接口，有一个输入参数，没有返回值

```
38 *
39 * @since 1.8
40 */
41 @FunctionalInterface
42 public interface Consumer<T> {
43
44     /**
45      * Performs this operation on the given argument.
46      *
47      * @param t the input argument
48      */
49     void accept(T t);
50
51     /**
52      * Returns a composed {@code Consumer} that performs, in sequence, this
53      * operation followed by the {@code after} operation. If performing either
```

```
package com.coding.function4;

import java.util.function.Consumer;

public class Demo03 {
    public static void main(String[] args) {
        // 没有返回值，只能传递参数 消费者
        // Consumer<String> consumer = new Consumer<String> () {
        //     @Override
        //     public void accept(String o) {
        //         System.out.println(o);
        //     }
        // };

        Consumer<String> consumer = s->{System.out.println(s)};
        consumer.accept("123");

        // 供给型接口 只有返回值，没有参数 生产者
    }
}
```

供给型接口，没有输入参数，只有返回参数

```
38 * @since 1.8
39 */
40 @FunctionalInterface
41 public interface Supplier<T> {
42
43     /**
44      * Gets a result.
45      *
46      * @return a result
47      */
48     T get();
49 }
50
```

只有返回值，没有参数

```
package com.coding.function4;

import java.util.function.Supplier;

public class Demo04 {
    public static void main(String[] args) {
        // Supplier<String> supplier = new Supplier<String>() {
        //     @Override
        //     public String get() {
```

```
//          return "aaa";
//      }
//  };

Supplier<String> supplier = ()->{return "aaa";};
System.out.println(supplier.get());
}
}
```

## 13、Stream流式计算

效率!

流 (Stream) 到底是什么呢?

数据库: 保存数据 + SQL操作

数据就是数据, 计算交给Stream!

**==集合就是数据, Stream管理计算==**

代码验证

```
package com.coding.stream;

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

/**
 * 一下数据, 进行操作筛选用户: 要求: 一行代码做出此题, 时长1分钟!
 * 1、全部满足偶数ID
 * 2、年龄都大于24
 * 3、用户名转为大写
 * 4、用户名字母倒排序
 * 5、只能输出一个名字
 */
public class StreamDemo {
    public static void main(String[] args) {
        User u1 = new User(11, "a", 23);
        User u2 = new User(12, "b", 24);
        User u3 = new User(13, "c", 22);
        User u4 = new User(14, "d", 28);
        User u5 = new User(16, "e", 26);

        // 集管理数据
        List<User> list = Arrays.asList(u1, u2, u3, u4, u5);
        // 计算交给Stream
        // 过滤 filter
        // 映射 map
        // 排序, sort
        // 分页 limit
        list.stream()
            .filter(u->{return u.getId()%2==0;})
            .filter(u->{return u.getAge()>24;})
            .map(u->{return u.getUsername().toUpperCase();})
            .sorted((o1,o2)->{return o2.compareTo(o1);})
    }
}
```

```

        .limit(1)
        .forEach(System.out::println);

// 泛型、注解、反射
// 链式编程 + 流式计算 + lambda表达式

    }
}

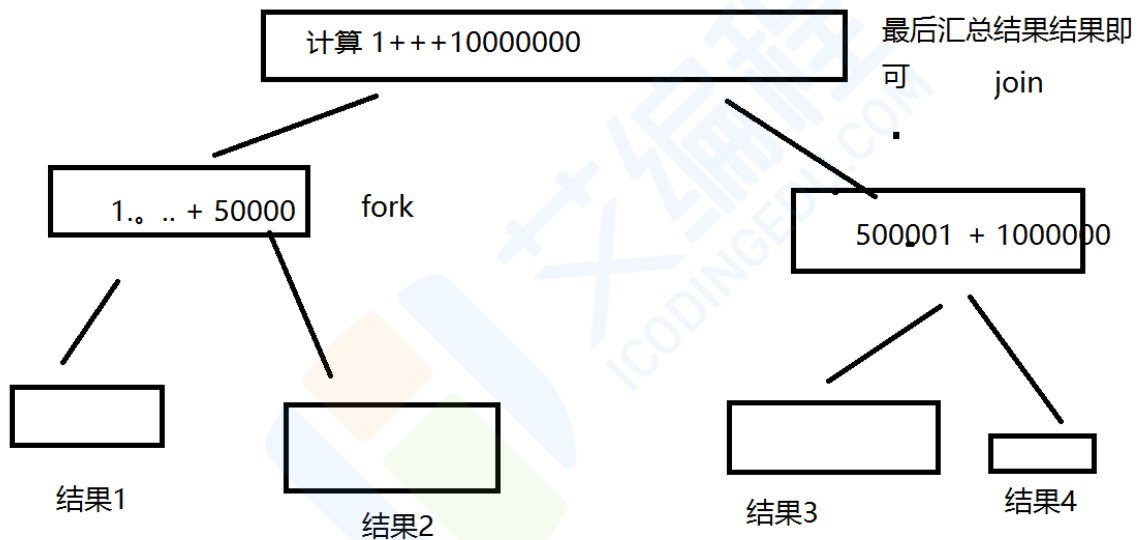
```

## 14、分支合并

什么是ForkJoin

大数据：mapreduce

任务切分，结果合并



工作窃取

A 任务1 --> 任务2 --> 任务3 --> 任务4 A领先执行完成，帮B执行任务！

B 任务1 --> 任务2 任务3 任务4

工作开始从头，窃取从尾，有效提高速度，双端队列

A 任务1 --> 任务2 --> 任务3 --> 任务4 A领先执行完成，帮B执行任务！

B 任务1 --> 任务2 → 任务3 任务4

核心类

ForkJoinPool 对列来执行，找到实现接口的类

WorkQueue是ForkJoinPool的一个内部类。

每一个线程都有一个 WorkQueue !

**ForkJoinTask** 代表正在 ForkJoinPool 中运行的 任务

```
m reportException(int): void
m fork(): ForkJoinTask<V>
m join(): V
m invoke(): V
```

fork: 安排任务异步执行, 白话: 创建一个子任务

join: 当任务完成后互殴去返回的计算结果!

invoke: 开始执行! 如果计算没有完毕, 就会等待!

**子类:** 递归 RecursiveTask

```
compact1, compact2, compact3
java.util.concurrent
Class ForkJoinTask<V>
java.lang.Object
  java.util.concurrent.ForkJoinTask<V>

All Implemented Interfaces:
Serializable, Future<V>
已知直接子类:
CountedCompleter, RecursiveAction, RecursiveTask

public abstract class ForkJoinTask<V>
extends Object
implements Future<V>, Serializable

在ForkJoinPool内运行的任务的抽象基类。A ForkJoinTask是一个线程实体,其重量比普通线程轻得多。大量任务和子任务可能由ForkJoinPool中的少量实际线程托管
```

没有返回值

有返回值

代码验证

```
* @return the result of the computation
*/
protected abstract V compute();
```

计算方法

```
package com.coding.stream;

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

// 裁员就是机会! 你要加工资的机会来了
public class ForkJoinTest {
    public static void main(String[] args) {
        // test1(); // 10582 ms 60
        // test2(); // 9965 ms 90
        // test3(); // 158 ms 101
    }

    // 正常测试
    public static void test1(){
        long start = System.currentTimeMillis();

        Long sum = 0L;
```

```

        for (Long i = 0L; i <= 10_0000_0000 ; i++) {
            sum +=i;
        }
        long end = System.currentTimeMillis();
        System.out.println("time:"+(end-start)+" sum:"+sum);
    }

    // ForkJoin测试
    public static void test2(){
        long start = System.currentTimeMillis();

        ForkJoinPool forkJoinPool = new ForkJoinPool();
        ForkJoinDemo forkJoinDemo = new ForkJoinDemo(0L,10_0000_0000L);
        Long sum = forkJoinPool.invoke(forkJoinDemo);

        long end = System.currentTimeMillis();
        System.out.println("time:"+(end-start)+" sum:"+sum);
    }

    // Stream并行流测试
    public static void test3(){
        long start = System.currentTimeMillis();

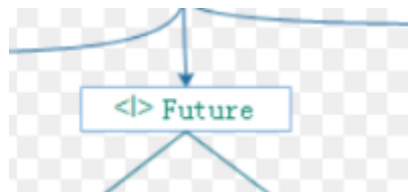
        long sum = LongStream.rangeClosed(0, 10_0000_0000).parallel().reduce(0,
        Long::sum);

        long end = System.currentTimeMillis();
        System.out.println("time:"+(end-start)+" sum:"+sum);
    }
}

```

## 15、异步回调

### 概述

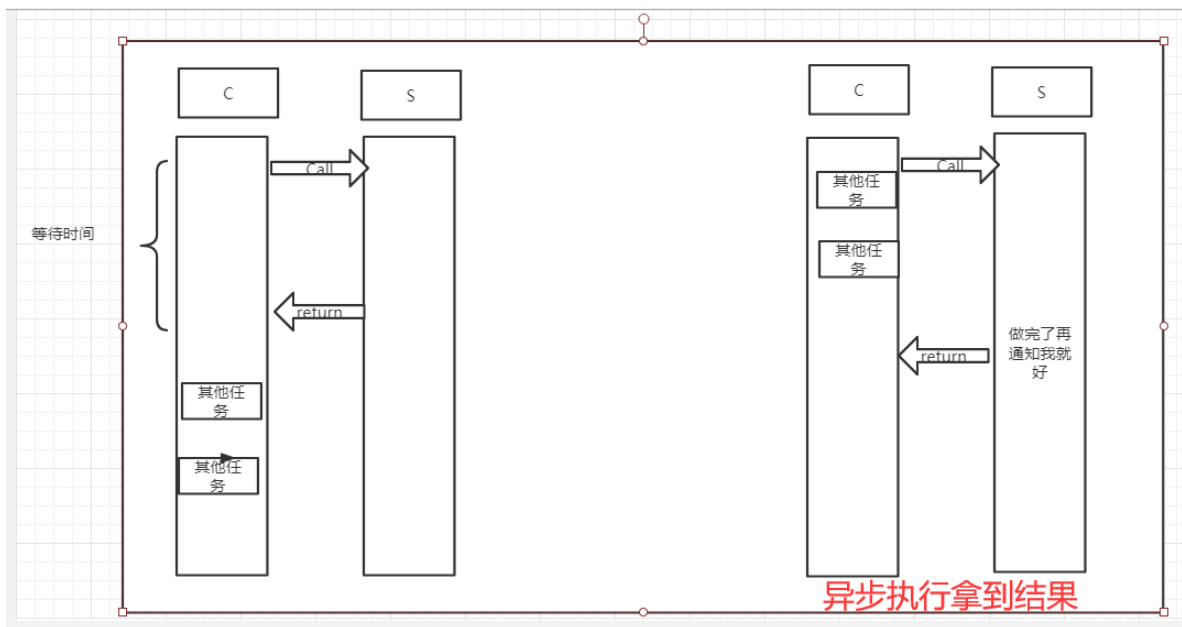


Future 设计初衷：对将来会发生的结果进行建模~

程序的性能要高，要异步处理！同步并阻塞！

A 线程做完了返回一个结果告诉main我做完了！

Future! ajax



分布式思想！

Java™ Platform Standard Ed. 8

Class: **CompletableFuture<T>**

java.lang.Object  
java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:  
CompletionStage<T>, Future<T>

public class **CompletableFuture<T>**  
extends Object  
implements Future<T>, CompletionStage<T>

甲Future可以明确地完成（设定其值和状态），并且可以被用作CompletionStage，支持有关的功能和它的完成时触发动作。

当两个或多个线程试图complete，completeExceptionally，或cancel一个CompletableFuture，只有一个成功。

除了直接操作状态和结果的这些和相关方法外，CompletableFuture还实现接口CompletionStage，具有以下策略：

- 为异步方法的操作可以由完成当前CompletableFuture的线程或完成方法的任何其他调用者执行。
- 所有不使用Executor参数的异步方法都使用ForkJoinPool.commonPool()执行（除非它不支持至少两个并行级别，在这种情况下，使用新的线程）。为了简化监视、调试和跟踪，所有生成的异步任务都是标记接口CompletionStage.AsynchronousCompletionTask的实例。
- 所有CompletionStage方法都是独立于其他公共方法实现的，因此一个方法的行为不会受到子类中其他方法的覆盖的影响。

CompletableFuture还实现Future，具有以下政策：

- 由于（不同于FutureTask），这个类不能直接控制导致其完成的计算，所以取消被视为另一种形式的异常完成。方法cancel具有相同的效果completeExceptionally(new CancellationException())。方法isCompletedExceptionally()可用于确定CompletableFuture是否以任何特殊方式完成。
- 如果使用CompletionException异常完成，方法get()和get(long, TimeUnit)将抛出与对应的CompletionException中保持的相同原因的ExecutionException。为了简化大多数情况下的使用，此类还定义了方法join()和getNow(T)，而是在这些情况下直接抛出CompletionException。

从以下版本开始：  
1.8

jdk  
中  
英  
对

## 实例

```
package com.coding.stream;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

// CompletableFuture 异步回调，对将来的结果进行结果，ajax就是一种异步回调！
public class CompletableFutureDemo {
    public static void main(String[] args) throws Exception {
        // 多线程也可以异步回调

        //
        //      // 没有返回结果,任务执行完了就完毕了！ 新增~
        //      CompletableFuture<Void> voidCompletableFuture =
        //      CompletableFuture.runAsync(() -> {
        //          // 插入数据，修改数据
```



```

//          System.out.println(Thread.currentThread().getName() + " 没有返回
值! ");
//      });
//
//          System.out.println(voidCompletableFuture.get());

// 有返回结果 ajax。 成功或者失败!
CompletableFuture<Integer> uCompletableFuture =
CompletableFuture.supplyAsync(() -> {
    System.out.println(Thread.currentThread().getName() + " 有返回值!");
    // int i = 10/0;
    return 1024;
});

// 有一些任务不紧急, 但是可以给时间做! 占用主线程! 假设这个任务需要返回结果!

System.out.println(uCompletableFuture.whenComplete((t, u) -> { // 正常编译
    完成!

        System.out.println("=t==" + t); // 正常结果
        System.out.println("=u==" + u); // 信息错误!
    }).exceptionally(e -> { // 异常!
        System.out.println("getMessage=>" + e.getMessage());
        return 555; // 异常返回结果
    }).get());
    }
}

```

## 16、JMM

问题: 请你谈谈你对volatile的理解

volatile 是轻量级的同步机制:

- 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 是一个抽象的概念! 并不真实存在! 一组规范!

Java内存模型:

- 1、线程解锁前, 必须要把共享的变量值刷新会主内存;
- 2、线程加锁前, 必须读取主内存的最新值到自己的工作内存;
- 3、必须是一把锁!

在你的电脑上运行完好!

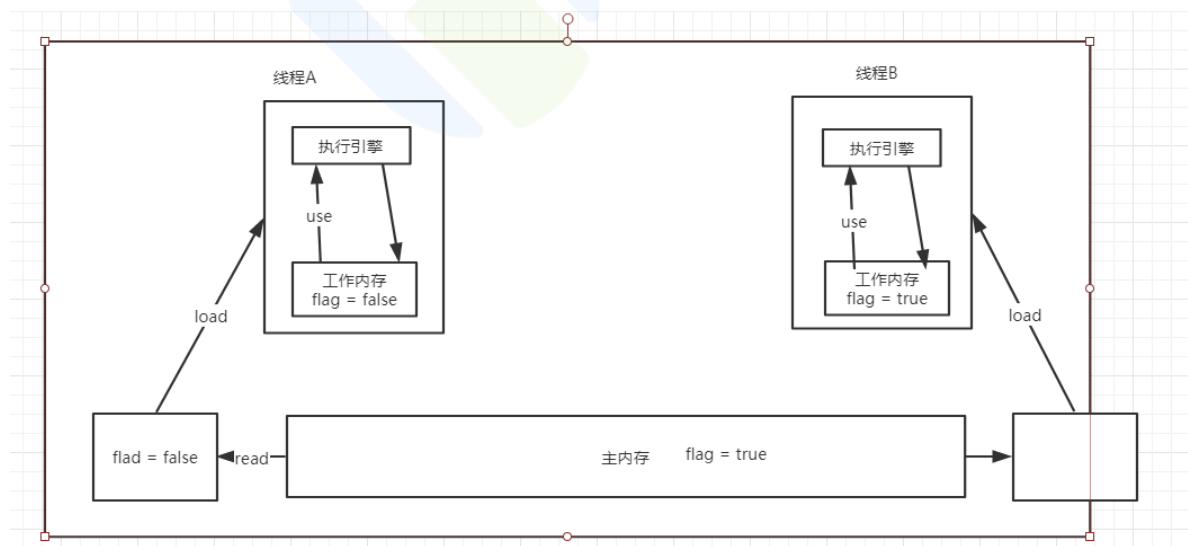
Java 内存模型对主内存与工作内存之间的具体交互协议定义了八种操作，具体如下：

- lock（锁定）：作用于主内存变量，把一个变量标识为一条线程独占状态。
- unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- read（读取）：作用于主内存变量，把一个变量从主内存传输到线程的工作内存中，以便随后的load动作使用。
- load（载入）：作用于工作内存变量，把read操作从主内存中得到的变量值放入工作内存的变量副本中。
- use（使用）：作用于工作内存变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量值的字节码指令时执行此操作。
- assign（赋值）：作用于工作内存变量，把一个从执行引擎接收的值赋值给工作内存的变量，每当虚拟机遇到一个需要给变量进行赋值的字节码指令时执行此操作。
- store（存储）：作用于工作内存变量，把工作内存中一个变量的值传递到主内存中，以便后续write操作。
- write（写入）：作用于主内存变量，把store操作从工作内存中得到的值放入主内存变量中。

## 17、volatile

volatile是不错的机制，但是也不能保证原子性。

代码验证可见性



```
package com.coding.jmm;

import java.util.concurrent.TimeUnit;

public class Test1 {
    // volatile 读取的时候去主内存中读取在最新值！
    private volatile static int num = 0;
```

```

    public static void main(String[] args) throws InterruptedException { // Main
线程
        new Thread()->{ // 线程A 一秒后会停止!    0
            while (num==0){

            }
        }).start();

        TimeUnit.SECONDS.sleep(1);

        num = 1;
        System.out.println(num);

    }

}

```

验证 volatile 不保证原子性

原子性: ACID 不可分割! 完整, 要么同时失败, 要么同时成功!

```

package com.coding.jmm;

public class Test2 {

    private volatile static int num = 0;

    public static void add(){
        num++;
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 20; i++) {
            new Thread()->{
                for (int j = 1; j <= 1000; j++) {
                    add(); // 20 * 1000 = 20000
                }
            },String.valueOf(i)).start();
        }

        // main线程等待上面执行完成, 判断线程存活数    2
        while (Thread.activeCount()>2){ // main gc
            Thread.yield();
        }

        System.out.println(Thread.currentThread().getName()+" "+num);

    }

}

```

```

1 public class Test2 {
2
3     private volatile static int num = 0;
4
5     public static void add(){
6         num++;
7     }
8
9     public static void main(String[] args) {
10         for (int i = 1; i <= 20; i++)
11             new Thread(()->{
12                 for (int j = 1; j <= 10; j++)
13                     add(); // 20 * 10 = 200
14             },String.valueOf(i)).start();
15     }
16 }

```

```

Compiled from Test2.java
public class com.coding.jmm.Test2 {
    public com.coding.jmm.Test2() {
        Code:
            0: aload_0
            1: invokespecial #1         // Method java/lang/Object.<init>:()V
            4: return

    public static void add0();
        Code:
            0: getstatic     #2         // Field num:I
            3: iconst_1
            4: iadd
            5: putstatic     #2         // Field num:I
            8: return

    public static void main(java.lang.String[]);
        Code:
            0: iconst_1
            1: istore_1
            2: iload_1
            3: bipush       20
            5: if_icmplt    33
            8: new          #3         // class java/lang/Thread

```

The screenshot shows the IntelliJ IDEA IDE interface. On the left, the 'Classes' pane displays the package hierarchy for `CompletableFuture`, including `java.time.format`, `java.time.temporal`, `java.time.zone`, `java.util`, `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`, `java.util.function`, and `java.util.jar`. A red arrow points to the `java.util.concurrent.atomic` package, with the Chinese text '原子性' (Atomicity) next to it. The main editor area shows the `CompletableFuture` class, its implemented interfaces `CompletionStage` and `Future`, and the start of its class definition: `public class CompletableFuture<T>`.

```
package com.coding.jmm;

import java.util.concurrent.atomic.AtomicInteger;

// 遇到问题不要着急，要思考如何去做！
public class Test2 {

    private volatile static AtomicInteger num = new AtomicInteger();

    public static void add(){
        num.getAndIncrement(); // 等价于 num++
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 20; i++) {
            new Thread()->{
                for (int j = 1; j <= 1000; j++) {
                    add(); // 20 * 1000 = 20000
                }
            },String.valueOf(i)).start();
        }

        // main线程等待上面执行完成，判断线程存活数 2
        while (Thread.activeCount()>2){ // main gc
            Thread.yield();
        }

        System.out.println(Thread.currentThread().getName()+" "+num);

    }
}
```

## 指令重排讲解

计算及在执行程序的之后，为了提高性能，编译器和处理器会进行指令重排！

处理在指令重排的时候必须要考虑==数据之间的依赖性！==

**指令重排：程序最终执行的代码，不一定是按照你写的顺序来的！**

```
int x = 11; // 语句1
int y = 12; // 语句2
x = y + 5;  // 语句3
y = x*x;    // 语句4

//怎么执行
1234
2134
1324

// 请问语句4 能先执行吗？
```

加深：int x,y,a,b = 0;

线程1	线程2
x = a;	y = b;
b = 1;	a = 2;
x = 0, y = 0	

假设编译器进行了指令重排！

线程1	线程2
b = 1;	a = 2;
x = a;	y = b;
x = 2, y = 1	

```
package com.coding.jmm;

// 两个线程交替执行的！
public class Test3 {

    int a = 0;
    boolean flag = false;

    public void m1(){ // A
        flag = true; // 语句2
        a = 1;       // 语句1
    }
}
```

```

    }

    public void m2(){ // B
        if (flag){
            a = a + 5; // 语句3
            System.out.println("m2=>" + a);
        }
    }
}

```

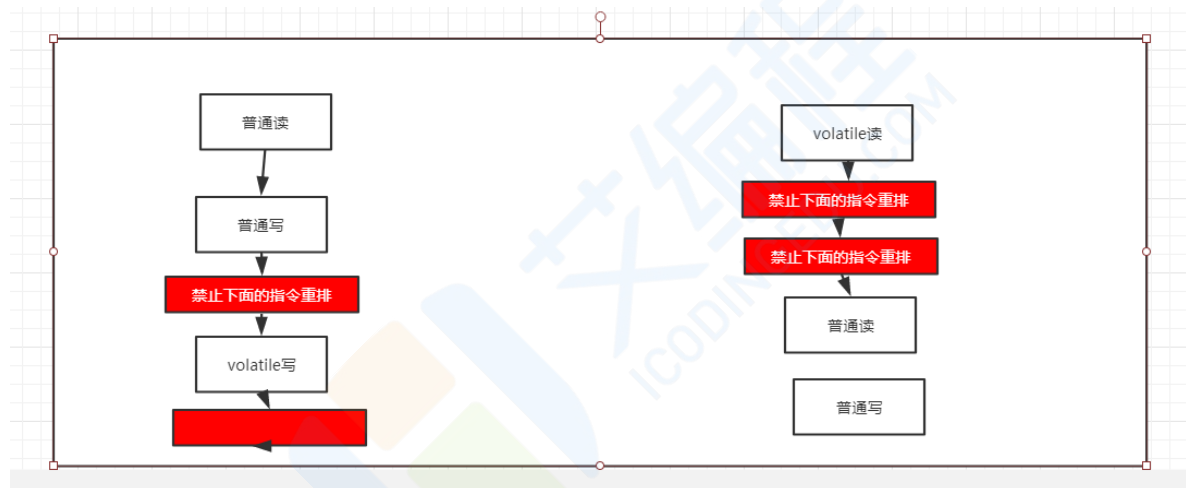
由于有指令重排的问题！

volatile：实现禁止指令重排！

刨根问底：

**内存屏障**：CPU的指令，作用两个：

- 1、保证特定的操作执行顺序
- 2、保证某些变量的内存可见性



**线程安全可以获得保证性**

可能存在 指令重排，有序性的问题的时候 volatile ！

volatile ！

## 18、深入单例模式

1、饿汉式

```

package com.coding.single;

public class Hungry {

    private byte[] data1 = new byte[10240];
    private byte[] data2 = new byte[10240];
    private byte[] data3 = new byte[10240];
    private byte[] data4 = new byte[10240];
}

```

```
// 单例模式核心思想，构造器私有！
private Hungry(){

}

private final static Hungry HUNGRY = new Hungry();

public static Hungry getInstance(){
    return HUNGRY;
}
}
```

## 2、懒汉式 DCL 双重检测锁

```
package com.coding.single;

public class LazyMan {
    private LazyMan(){
        System.out.println(Thread.currentThread().getName()+"Start");
    }

    private volatile static LazyMan lazyMan;

    public static LazyMan getInstance(){
        if (lazyMan==null){
            synchronized (LazyMan.class){
                if (lazyMan==null){
                    lazyMan = new LazyMan(); // 可能存在指令重排！
                    /*
                    A: 1    3    2
                    B: lazyMan = null ;
                    1. 分配对象的内存空间
                    2. 执行构造方法初始化对象
                    3. 设置实例对象指向刚分配的内存的地址， instance = 0xffffffff;
                    */
                }
            }
        }
        return lazyMan;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread()->{
                LazyMan.getInstance();
            }.start();
        }
    }
}
```

## 3、静态内部类

```
package com.coding.single;

import java.util.concurrent.RecursiveTask;
```

```

public class Holder {
    private Holder(){

    }

    public static Holder getInstance(){
        return InnerClass.HOLDER;
    }

    private static class InnerClass {
        private static final Holder HOLDER = new Holder();
    }
}

```

反射:

```

package com.coding.single;

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

public class LazyMan {

    private static boolean flag = false;

    private LazyMan(){
        synchronized (LazyMan.class){
            if (flag==false){
                flag = true;
            }else {
                throw new RuntimeException("不要试图使用反射破坏单例模式");
            }
        }
    }

    private volatile static LazyMan lazyMan;

    public static LazyMan getInstance(){
        if (lazyMan==null){
            synchronized (LazyMan.class){
                if (lazyMan==null){
                    lazyMan = new LazyMan(); // 可能存在指令重排!
                    /*
                    A: 1    3    2
                    B:  lazyMan = null ;
                    1. 分配对象的内存空间
                    2. 执行构造方法初始化对象
                    3. 设置实例对象指向刚分配的内存的地址,  instance = 0xffffffff;
                    */
                }
            }
        }
        return lazyMan;
    }
}

```



```

    public static void main(String[] args) throws NoSuchMethodException,
        IllegalAccessException, InvocationTargetException, InstantiationException,
        NoSuchFieldException {

        //LazyMan instance1 = LazyMan.getInstance();
        Constructor<LazyMan> declaredConstructors =
        LazyMan.class.getDeclaredConstructor(null);
        declaredConstructors.setAccessible(true); // 无视 private 关键字

        Field flag = LazyMan.class.getDeclaredField("flag");
        flag.setAccessible(true);

        LazyMan instance1 = declaredConstructors.newInstance();

        flag.set(instance1, false);

        LazyMan instance2 = declaredConstructors.newInstance();

        System.out.println(instance1);
        System.out.println(instance2);

    }
}

```

#### 4、枚举（最安全的）

```

package com.coding.single;

import java.lang.reflect.Constructor;

// 枚举是一个类! EnumSingle.class
public enum EnumSingle {

    INSTANCE;

    public EnumSingle getInstance(){
        return INSTANCE;
    }

    public static void main(String[] args) throws Exception {
        EnumSingle enumSingle2 = EnumSingle.INSTANCE;

        Constructor<EnumSingle> declaredConstructor =
        EnumSingle.class.getDeclaredConstructor(null);
        declaredConstructor.setAccessible(true);
        // 期望的异常 throw new IllegalArgumentException("Cannot reflectively
        create enum objects");

        // java.lang.NoSuchMethodException: com.coding.single.EnumSingle.<init>
        ()
        declaredConstructor.newInstance();
    }
}

```

```
}
```

```
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation. 保留所有权利。

C:\Users\Administrator\Desktop\JUC并发专题\juc\target\classes\com\coding\single>javap -p EnumSingle.class
Compiled from "EnumSingle.java"
public final class com.coding.single.EnumSingle extends java.lang.Enum<com.coding.single.EnumSingle> {
    public static final com.coding.single.EnumSingle INSTANCE;
    private static final com.coding.single.EnumSingle[] $VALUES;
    public static com.coding.single.EnumSingle[] values();
    public static com.coding.single.EnumSingle valueOf(java.lang.String);
    private com.coding.single.EnumSingle();
    public com.coding.single.EnumSingle getInstance();
    public static void main(java.lang.String[]) throws java.lang.Exception;
    static {};
}

C:\Users\Administrator\Desktop\JUC并发专题\juc\target\classes\com\coding\single>
```

jad 反编译工具!

找到万恶之源

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   EnumSingle.java

package com.coding.single;

import java.lang.reflect.Constructor;

public final class EnumSingle extends Enum
{

    public static EnumSingle[] values()
    {
        return (EnumSingle[])$VALUES.clone();
    }

    public static EnumSingle valueOf(String name)
    {
        return (EnumSingle)Enum.valueOf(com/coding/single/EnumSingle, name);
    }

    private EnumSingle(String s, int i)
    {
        super(s, i);
    }

    public EnumSingle getInstance()
    {
        return INSTANCE;
    }

    public static void main(String args[])
        throws Exception
    {
        EnumSingle enumSingle2 = INSTANCE;
        Constructor declaredConstructor =
com/coding/single/EnumSingle.getDeclaredConstructor(null);
        declaredConstructor.setAccessible(true);
    }
}
```

```

        declaredConstructor.newInstance(null);
    }

    public static final EnumSingle INSTANCE;
    private static final EnumSingle $VALUES[];

    static
    {
        INSTANCE = new EnumSingle("INSTANCE", 0);
        $VALUES = (new EnumSingle[] {
            INSTANCE
        });
    }
}

```

再次测试

```

package com.coding.single;

import java.lang.reflect.Constructor;

// 枚举是一个类! EnumSingle.class
public enum EnumSingle {

    INSTANCE;

    public EnumSingle getInstance(){
        return INSTANCE;
    }

    public static void main(String[] args) throws Exception {
        EnumSingle enumSingle2 = EnumSingle.INSTANCE;

        Constructor<EnumSingle> declaredConstructor =
            EnumSingle.class.getDeclaredConstructor(String.class, int.class);
        declaredConstructor.setAccessible(true);
        // 期望的异常 throw new IllegalArgumentException("Cannot reflectively
        create enum objects");
        // Exception in thread "main" java.lang.IllegalArgumentException: Cannot
        reflectively create enum objects
        // java.lang.NoSuchMethodException: com.coding.single.EnumSingle.<init>()
        declaredConstructor.newInstance();
    }
}

```

## 19、深入理解CAS

在互联网缩招的情下，初级程序员大量过剩，高级程序员重金难求！

CAS：比较并交换

```

package com.coding.cas;

```

```
import java.util.concurrent.atomic.AtomicInteger;

public class CASDemo {
    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(5);

        // compareAndSet 简称 CAS 比较并交换!
        // compareAndSet(int expect, int update) 我期望原来的值是什么, 如果是, 就更新

        System.out.println(atomicInteger.compareAndSet(5,
2020)+"=>" +atomicInteger.get());

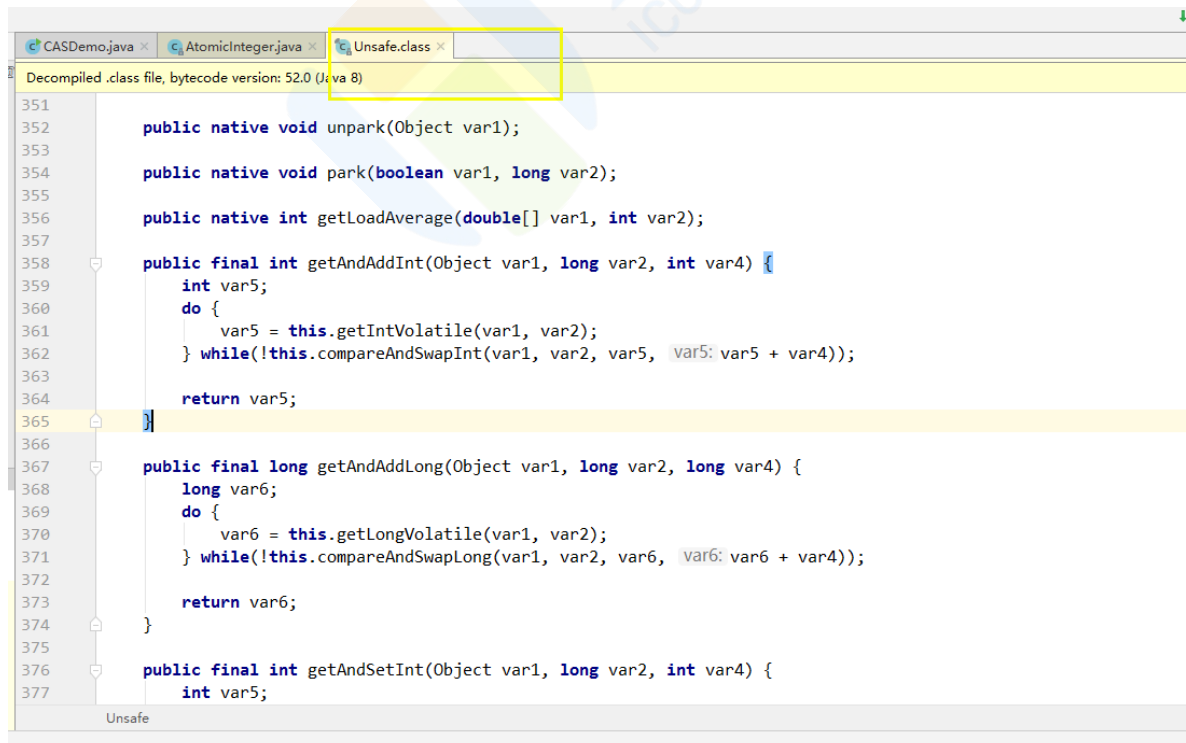
        // 2020
        System.out.println(atomicInteger.compareAndSet(2020,
1024)+"=>" +atomicInteger.get());
    }
}
```

CAS 底层原理? 如果知道, 谈谈你对Unsafe的理解?

getAndIncrement 分析这个+1是怎么实现的

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

java出生就自带的



```
Decompiled .class file, bytecode version: 52.0 (Java 8)

351
352 public native void unpark(Object var1);
353
354 public native void park(boolean var1, long var2);
355
356 public native int getLoadAverage(double[] var1, int var2);
357
358 public final int getAndAddInt(Object var1, long var2, int var4) {
359     int var5;
360     do {
361         var5 = this.getIntVolatile(var1, var2);
362     } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));
363
364     return var5;
365 }
366
367 public final long getAndAddLong(Object var1, long var2, long var4) {
368     long var6;
369     do {
370         var6 = this.getLongVolatile(var1, var2);
371     } while(!this.compareAndSwapLong(var1, var2, var6, var6: var6 + var4));
372
373     return var6;
374 }
375
376 public final int getAndSetInt(Object var1, long var2, int var4) {
377     int var5;
```

```

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; // 内存地址偏移

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;

    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;
    }
}

```

Unsafe 就是CAS 的核心类！

JAVA 无法直接操作系统的底层！ native

Unsafe 后门！操作特定内存中的数据！ 里面的所有的所有方法，都可以像C的指针一样直接操作内存！

native

最后解释CAS 是什么

CAS 就是 他是一个 CPU的 并发原语！

它的功能就是判断内存中的某个位置的值，是否是预期值，如果是更新为自己指定的新值，原子性的！内存级别的，连续的

**==本身就不存在了数据不一致的问题！根治！==**

```

public native void park(boolean var1, long var2);

public native int getLoadAverage(double[] var1, int var2);

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;
}

```

通过这个对象获得它的内存地址 0xffff

内存地址+1

通过内存地址直接拿到值！

汇编层面理解

Unsafe 类中的 compareAndSwapint，是一个本地方法，该方法的实现位于 unsafe.cpp 中；

image-20200206193735516

CAS: 比较当前工作内存的中值和主内存的中值, 如果相同, 则执行操作, 否则就一直比较知道值一致为止!

CAS:

内存值A: 旧的预期值 B , 想修改为 V!

**CAS的缺点:**

- 1、循环时间长, 开销大!
- 2、只能保证一个共享变量的原子操作!
- 3、ABA 问题! ? 狸猫换太子!

## 20、原子引用

原子类 AtomicInteger 的ABA问题谈谈? 原子更新引用知道吗?

```
package com.coding.cas;

import com.coding.demo02.A;

import java.util.concurrent.atomic.AtomicInteger;

public class CASDemo {
    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(5);

        // compareAndSet 简称 CAS 比较并交换!
        // compareAndSet(int expect, int update) 我期望原来的值是什么, 如果是, 就更新

        // a
        System.out.println(atomicInteger.compareAndSet(5,
2020)+"=>" + atomicInteger.get());

        // c 偷偷的改动
        System.out.println(atomicInteger.compareAndSet(2020,
2021)+"=>" + atomicInteger.get());
        System.out.println(atomicInteger.compareAndSet(2021,
5)+"=>" + atomicInteger.get());

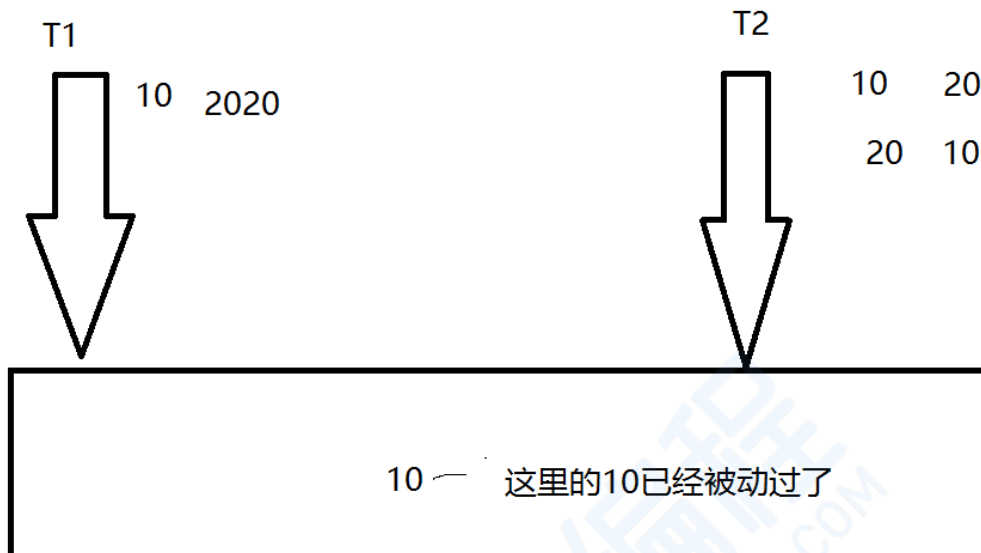
        // b
        System.out.println(atomicInteger.compareAndSet(5,
1024)+"=>" + atomicInteger.get());
    }
}
```

```
}
```

CAS 会导致 ABA 的问题!

CAS 算法的前提是：取出内存中某个时刻的数据，并且比较并交换！在这个时间差内有可能数据被修改！

**== 尽管 CAS 操作成功！但是不代表这个过程就是没有问题的！ ==**



乐观锁!

原子引用 AtomicReference

版本号，时间戳!

The screenshot shows the Java API documentation for the `AtomicReference` class. The left sidebar lists the package hierarchy: `java.util.concurrent.atomic`. The main content area shows the class `AtomicReference<V>`, which extends `Object` and implements `Serializable`. It includes the constructor `AtomicReference()` and the method `get()`.

版本号原子引用，类似乐观锁

```
package com.coding.cas;

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

/**
 * AtomicReference 原子引用
 * AtomicStampedReference 加了时间戳 类似于乐观锁！ 通过版本号
 */
public class CASDemo2 {
    static AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(100,1);

    public static void main(String[] args) {
        new Thread()->{
            //1 、 获得版本号
            int stamp = atomicStampedReference.getStamp();
            System.out.println("T1 stamp 01=>" + stamp);

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

            atomicStampedReference.compareAndSet(100,101,

atomicStampedReference.getStamp(),atomicStampedReference.getStamp()+1);

            System.out.println("T1 stamp
02=>" + atomicStampedReference.getStamp());

            atomicStampedReference.compareAndSet(101,100,

atomicStampedReference.getStamp(),atomicStampedReference.getStamp()+1);

            System.out.println("T1 stamp
03=>" + atomicStampedReference.getStamp());

        }, "T1").start();

        new Thread()->{
            // GIT 看到数据被动过了！

            //1 、 获得版本号
            int stamp = atomicStampedReference.getStamp();
            System.out.println("T1 stamp 01=>" + stamp);

            // 保证上面的线程先执行完毕！
```



```

        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        boolean b = atomicStampedReference.compareAndSet(100, 2019,
            stamp, stamp + 1);
        System.out.println("T2 是否修改成功: "+b);
        System.out.println("T2 最新的stamp: "+stamp);
        System.out.println("T2 当前的最新
值: "+atomicStampedReference.getReference());
    }, "T2").start();

}

}

```

解决ABA问题：AtomicStampedReference

## 21、Java锁

### 1、公平锁非公平锁

是什么

公平锁：就是非常公平，先来后到

非公平锁：就是非常不公平，可以插队！但是有时候插队可以提高效率 3个小时

```

public ReentrantLock() {
    sync = new NonfairSync(); // 默认是非公平锁，随机
}

public ReentrantLock(boolean fair) { // 公平锁
    sync = fair ? new FairSync() : new NonfairSync();
}

```

两者区别

公平锁：并发环境下，每个线程在获取到锁的时候都要先看一下这个锁的等待队列！如果为空，那就可以占有锁！否则就要等待！

非公平锁：上来就直接尝试占有该锁！如果失败就会采用类似公平锁的方式！

synchronized：默认就是非公平锁，改不了

ReentrantLock：默认就是非公平锁，可以通过参数修改！

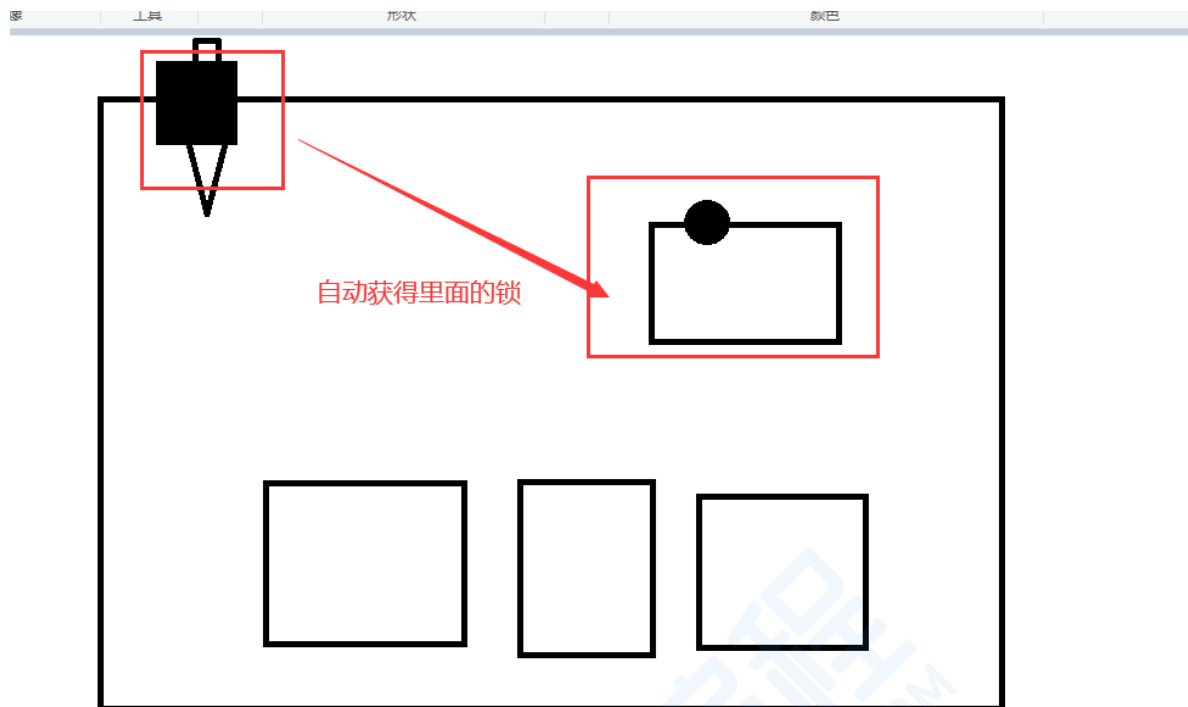
### 2、可重入锁

递归锁就是可重入锁！

大门 卧室A 卧室B 厕所

白话：线程可以进入任何一个他已经拥有锁的，锁所同步的代码块！

==最大的好处：就是避免死锁！==



练武不练功，到老一场空！

```
package com.coding.lock;

import java.util.function.Predicate;

public class RTLock {

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

        new Thread()->{
            phone.sendSMS();
        },"T1").start();

        new Thread()->{
            phone.sendMail();
        },"T2").start();
    }
}

class Phone {
    public synchronized void sendSMS(){ // 外面的锁
        System.out.println(Thread.currentThread().getName()+" sendSMS");
        sendMail(); // 这个方法本来也是被锁的，但是由于获得了外面的锁，所以这个锁也获得了！
    }

    public synchronized void sendMail(){
        System.out.println(Thread.currentThread().getName()+" sendMail");
    }
}
```

```
}  
}
```

## ReentrantLock

```
package com.coding.lock;  
  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class RTLock2 {  
    public static void main(String[] args) {  
        Phone2 phone2 = new Phone2();  
  
        // T1 线程在获得外面锁的时候，也会拿到里面的锁！  
        new Thread(phone2, "T1").start();  
  
        new Thread(phone2, "T2").start();  
    }  
}  
  
class Phone2 implements Runnable{  
  
    Lock lock = new ReentrantLock();  
  
    public void get(){  
        lock.lock(); // A // lock 锁必须匹配！  
        // lock.lock(); // A // lock 锁必须匹配！  
        try {  
            System.out.println(Thread.currentThread().getName()+"=>get");  
            set();  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            lock.unlock(); // A  
        }  
    }  
  
    public void set(){  
        lock.lock(); // B  
        try {  
            System.out.println(Thread.currentThread().getName()+"=>set");  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            lock.unlock(); // B  
        }  
    }  
  
    @Override  
    public void run() {  
        get();  
    }  
}
```

### 3、自旋锁

自旋锁 spinlock

尝试获取锁的线程不会立即阻塞，采用循环的方式尝试获取锁！减少上下文的切换！缺点会消耗CPU

```
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;
}
```

我们手动编写一个锁来测试！

```
package com.coding.lock;

import com.sun.org.apache.bcel.internal.generic.NEW;

import java.util.concurrent.atomic.AtomicReference;

// coding自己定义的艾编程的锁！
public class MyLock {

    // 原子引用 CAS
    AtomicReference<Thread> atomicReference = new AtomicReference<>();

    // 加锁
    public void myLock(){
        Thread thread = Thread.currentThread();
        System.out.println(Thread.currentThread().getName()+"==mylock");

        // 期望是空的没有加锁， thread // 自旋，（循环！）
        while (atomicReference.compareAndSet(null,thread)){// cas

        }

    }

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

}
```

```

package com.coding.lock;

import java.util.concurrent.TimeUnit;

public class SpinLockDemo {

    public static void main(String[] args) {
        MyLock myLock = new MyLock();

        new Thread()->{
            myLock.myLock();

            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            myLock.myUnlock();

        }, "T1").start();

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

        new Thread()->{
            myLock.myLock();

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

            myLock.myUnlock();

        }, "T2").start();

    }

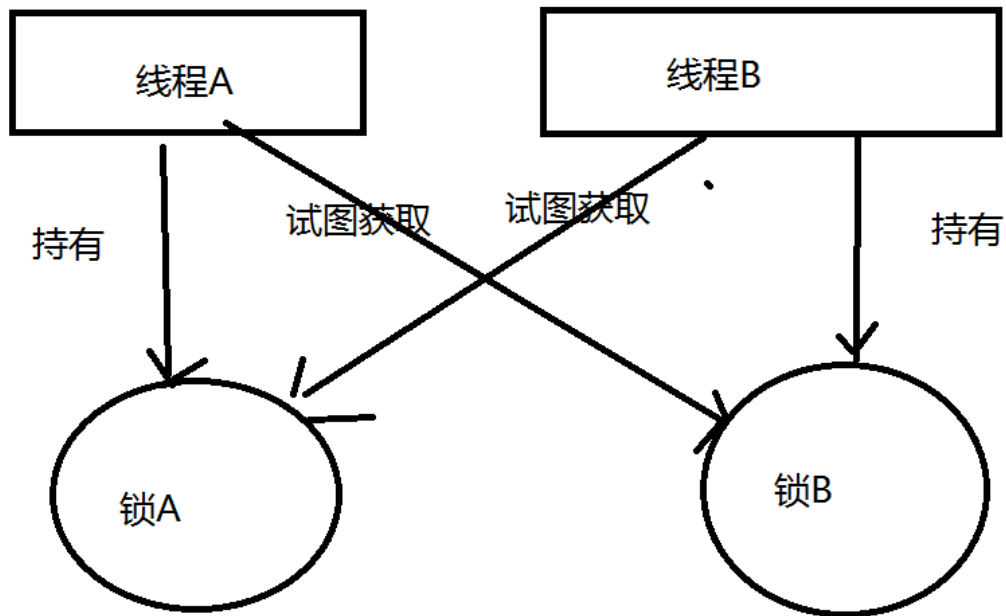
}

```

## 4、死锁

可重入锁：可以避免死锁！

什么是死锁！



产生死锁的原因：

- 1、系统资源不足
- 2、进程运行顺序不当！
- 3、资源分配不当！

```
package com.coding.lock;

import java.util.concurrent.TimeUnit;

public class DeadLock {
    public static void main(String[] args) {
        String lockA = "lockA";
        String lockB = "lockB";

        new Thread(new HoldLockThread(lockA, lockB), "T1").start();
        new Thread(new HoldLockThread(lockB, lockA), "T2").start();
    }
}

class HoldLockThread implements Runnable{

    private String lockA;
    private String lockB;

    public HoldLockThread(String lockA, String lockB) {
        this.lockA = lockA;
        this.lockB = lockB;
    }

    @Override
    public void run() {
        // A 想要拿B
        synchronized (lockA){
```

```

System.out.println(Thread.currentThread().getName()+"lock:"+lockA+"=>get" +
lockB);

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

    // B想要拿到A
    synchronized (lockB){

        System.out.println(Thread.currentThread().getName()+"lock:"+lockB+"=>get" +
lockA);

    }

}

}
}

```

jmc.ini	2020/1/27 星期...	配置设置	1 KB
jps.exe	2020/1/27 星期...	应用程序	17 KB
jrunscript.exe	2020/1/27 星期...	应用程序	17 KB
isadebugd.exe	2020/1/27 星期...	应用程序	17 KB
jstack.exe	2020/1/27 星期...	应用程序	17 KB
jstat.exe	2020/1/27 星期...	应用程序	17 KB
jstatd.exe	2020/1/27 星期...	应用程序	17 KB
jvisualvm.exe	2020/1/27 星期...	应用程序	193 KB

解决:

jps -l

✖ (c) 2016 Microsoft Corporation。保留所有权利。

```

C:\Users\Administrator\Desktop\JUC并发专题\juc>jps -l
14400 com.coding.lock.DeadLock → 14400
11588 sun.tools.jps.Jps
15588 org.jetbrains.jps.cmdline.Launcher
7960 org.jetbrains.idea.maven.server.RemoteMavenServer
6732

C:\Users\Administrator\Desktop\JUC并发专题\juc>

```

查看死锁现象: jstack 进程号

```
Found one Java-level deadlock:
=====
"T2":
  waiting to lock monitor 0x00000000184b07c8 (object 0x00000000d5b84fe8, a java.lang.String),
  which is held by "T1"
"T1":
  waiting to lock monitor 0x00000000184b2ef8 (object 0x00000000d5b85020, a java.lang.String),
  which is held by "T2"

Java stack information for the threads listed above:
=====
"T2":
  at com.coding.lock.HoldLockThread.run(DeadLock.java:39)
  - waiting to lock <0x00000000d5b84fe8> (a java.lang.String)
  - locked <0x00000000d5b85020> (a java.lang.String)
  at java.lang.Thread.run(Thread.java:748)
"T1":
  at com.coding.lock.HoldLockThread.run(DeadLock.java:39)
  - waiting to lock <0x00000000d5b85020> (a java.lang.String)
  - locked <0x00000000d5b84fe8> (a java.lang.String)
  at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

死锁：看日志 9 10 我分析堆栈！

==每一个理论，背后一定要去实践！不仅仅是看一遍，还要自己过一遍==