# 一、Java多线程基础

## # 1、多线程概述

1. 进程（正在运行的程序）
2. 多进程（计算机同时开几个程序）
3. 多线程（一个进程内同时进行多个任务）

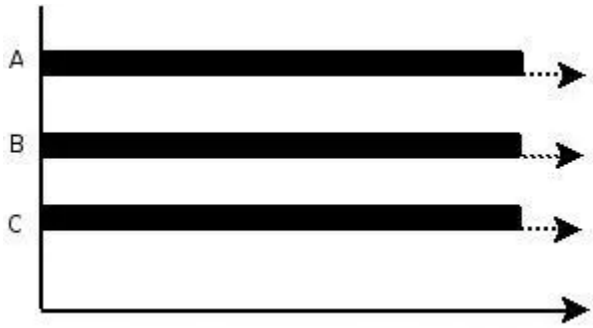
   线程：是程序的执行单元、路径。是程序使用CPU的最基本单位。

4. 意义

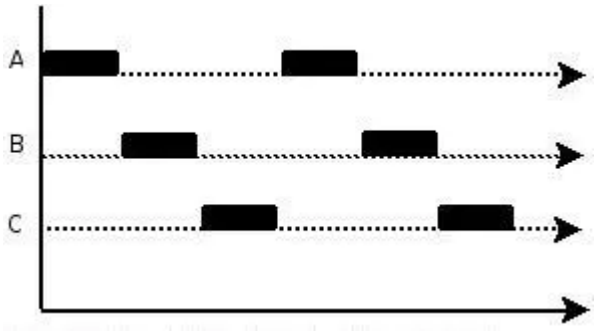
   - 多线程的存在，不提高程序速度。其实是为了提高应用程序的使用率。
   - 程序的执行其实都是在抢CPU的资源，CPU的执行权。
   - 多个进程是在抢这个资源，而其中的某一个进程如果执行路径比较多，就会有更高的几率抢到CPU的执行权。

5. 并发与并行

   并行：指在同一时刻，有多条指令在多个处理器上同时执行。所以无论从微观还是从宏观来看，二者都是一起执行的。




   并发：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。



# 2、线程控制方法

## （1）休眠 静态 用在run中

下面都要抛异常，后者是继承了Thread类

```
Thread.sleep(1000);      等价于 sleep(1000);
```

## （2）加入线程

要我先执行完，才能继续执行我之外的其他句子（抛出异常）

放在start之后！

```
mt1.join();
```


## （3）礼让线程 静态 用在run中

能在一定程度让线程执行更规律AB AB AB

```
Thread.yield();
```

## （4）守护线程

主线程（一般为main）结束后其他的也关掉

// 当主线程结束，这两个也关掉， 位置在start之前

```
    mt1.setDaemon(true);

    mt2.setDaemon(true);
```

## （5）线程中断（停）动态 抛异常

直接停止线程，已过时

```
mt1.stop();
```

停止线程，后面的语句照样执行

```
mt2.interrupt();
```

## （6）等待和唤醒 用在run中

这三个方法必须要用在加锁的线程中

利用到机制（在任意对象中，因为锁）

wait() 等待（被唤醒了，就在等待的位置醒来）

notify() 唤醒（唤醒了并不代表立即 执行，还得抢CPU）

notifyAll()唤醒全部线程

```
public class test3 {
    public static void main(String[] args) throws InterruptedException {
        Student s = new Student();
        SetThread1 st = new SetThread1(s);
        Thread t1 = new Thread(st);

        GetThread1 gt = new GetThread1(s);
        Thread t2 = new Thread(gt);

        t1.setDaemon(true);
        t2.setDaemon(true);
        t1.start();
        t2.start();
        Thread.sleep(100);
    }
}

class SetThread1 implements Runnable {
    private Student s;
    private int x = 0;
    public SetThread1(Student s) {this.s = s;}
    @SneakyThrows
    @Override
    public void run() {
        while (true) {
            if (x % 2 == 0) {
              s.set("林青霞");
```

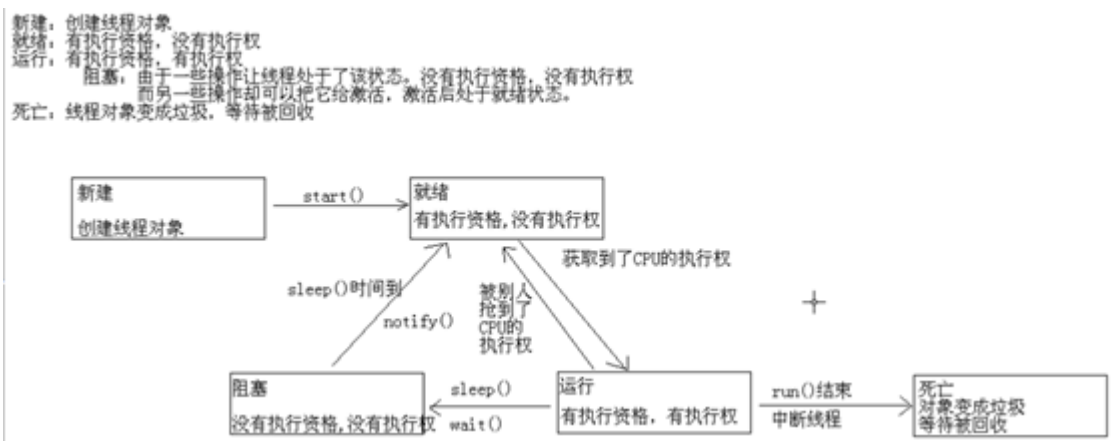
```
            } else{ s.set("刘意");}
            x++;
        }
    }
}

class GetThread1 implements Runnable {
    private Student s;
    public GetThread1(Student s) {this.s = s;}
    @SneakyThrows
    @Override
    public void run() {
        while (true) {s.get();}
    }
}

class Student {
    private String name;
    private boolean flag; // 默认情况是没有数据 ,如果是true,说明有数据

    public synchronized void set(String name) throws InterruptedException {
        if (this.flag) {// 如果有数据,就等待
            this.wait();
        }
        this.name = name;
        this.flag = true;// 修改标记
        this.notify();
    }
    public synchronized void get() throws InterruptedException {
        if (!this.flag) {// 如果没有数据,就等待
            this.wait();
        }
        System.out.println(this.name);// 获取数据
        this.flag = false;// 修改标记
        this.notify();
    }
}
```

# 3、线程生命周期



# 4、实现多线程

## （1）继承Thread类

### ①内部方法

1、获取名称

```
mt1.getName()

getName(); //自定义类中

Thread.currentThread().getName();//获取当前线程名称 run中
```

3、获取线程对象 优先级（默认5，范围1~10）

```
public final int getPriority() : //返回线程对象的优先级

mt1.getPriority();
```

4、设置线程忧先级

public final void setPriority(int newPriority) //更改线程的忧先级。

```
mt1.setPriority(1);
mt2.setPriority(10);
```

### ②注意

- run ()方法是单线程。因为 run() 方法直接调用只执行一次、这也是它和 start() 的区别
- 被多线程执行的代码很耗时
- run外面的语句只执行一次！

### ③代码实例

```
public static void main(String[] args) throws InterruptedException {
        MyThread2 mt1 = new MyThread2();
        mt1.setName("第一个线程: ");// 设置线程名称
        // mt1.getPriority();// 获取优先级
        mt1.setPriority(10);// 修改优先级
```

```
        mt1.setDaemon(true);// 设置收获线程,当主线程结束，这两个也关掉
        mt1.start();//开始
        // mt1.join();

        MyThread2 mt2 = new MyThread2();
        mt2.setName("第二个线程: ");
        mt2.setPriority(1);
        mt2.setDaemon(true);
        mt2.start();

        Thread.currentThread().setName("刘备");
    }

class MyThread2 extends Thread {
    @Override
    public void run() {
        // 写需要多线程执行的代码
        for (int i = 0; i < 10; i++) {
            System.out.println(getName() + "好好学习天天向上" + i);
            Thread.yield();
            Thread.sleep(500);
        }
    }
}
```

## （2）实现Runnable接口

### ①实现接口的好处

- 避免Java 单继承带的局限性 。
- 适合多个程序处理同一线程

### ②解决线程安全问题的标准

- 是否是多线程环境
- 是否有共享数据
- 是否有多条语句操作共享数据（同步）

### ③代码实例

```
public static void main(String[] args) {
            Runnable mr = new MyRunnable();
            Thread t1 = new Thread(mr,"第一个线程");
            Thread t2 = new Thread(mr,"第er个线程");

            t1.start();
            t2.start();
```

```
}



class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

## （3）Callable泛型接口

### ①实现过程

第一步：自定义类实现泛型接口

```
class MyCall implements Callable<Integer> {
    private int number;
    public MyCall(int number) {
        this.number = number;
    }
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i < number; i++) {
            sum += i;
        }
        return sum;
    }
}
```

第二步：用线程池调用刚刚的类

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
    ExecutorService pool = Executors.newFixedThreadPool(2);
    //实现线程类，并加入线程池
    Future<Integer> f1 = pool.submit(new MyCall(10));
    Future<Integer> f2 = pool.submit(new MyCall(5));
    //输出返回值
    System.out.println(f1.get());
    System.out.println(f2.get());
    //结束
    pool.shutdown();
}
```

## ②内部类实现

```
public static void main(String[] args) {
    ExecutorService es = Executors.newSingleThreadExecutor();
    Future<Integer> ft = es.submit(new Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
            int sum = 0;
            for (int i = 0; i < number; i++) {
                sum += i;
            }
            return sum;
        }
    });
    System.out.println(ft.get());
}
```

## ③注意

1、实现的Callable接口要加`泛型<Object>`

2、Call方法实现的包装类型，不能使用基本类型做返回值

3、它貌似不能跟前两个一样直接运行，要借助线程池（我没看到start方法）

## ④不用地址池实现Callable线程

### 1、FutureTask接口

（1）概念

实现了Runnable接口和Future接口，它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

（2）格式

① `FutureTask<Integer> fu = new FutureTask<Integer>([Callable线程名]);`

②使用Thread实现线程 `new Thread(fu).start();`

（3）实例

```
Callable<Integer> ca = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        for (int i = 0; i < 100 ; i++) {
            System.out.println(i);
        }
        return 0;
    }
}; //这前面用的内部类，就相当于一条语句
FutureTask<Integer> fu = new FutureTask<Integer>(ca);
new Thread(fu).start();
```

# 5、匿名内部内实现线程

## （1）Thread方式

```
//使用Thread实现线程
new Thread() {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }
}.start();
```

## （2）Runnable方式

```
new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(" " + i);
        }
    }
}) {
    //如果这里也放个run方法，那么线程只执行此处的run
}.start();
```

示例：

```
new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    System.out.println(" " + i);
                }
            }
        }) {
            //如果这里也放个run方法，那么线程只执行此处的run
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    System.out.println(Thread.currentThread().getName() +"  "+ i);
                }
            }
        }.start();
```

```
Thread-0  0
Thread-0  1
Thread-0  2
Thread-0  3
Thread-0  4
Thread-0  5
Thread-0  6
Thread-0  7
Thread-0  8
Thread-0  9

Process finished with exit code 0
```

# 6、线类 程组ThreadGroup

## （1）概念

ThreadGroup表示线程组，它对一批线程进行分类管理，程序直接控制线程组。

默认下所有线程（包括**main**）都属于**main**线程组

### ②代码实例

```
ThreadGroup tg = new ThreadGroup("新的组");
MyRunnable mr = new MyRunnable();

Thread t1 = new Thread(tg, mr, "爸爸");
Thread t2 = new Thread(tg, mr, "傻儿子");

System.out.println(t1.getThreadGroup().getName());
```

# 7、线程池Execu

### ①线程池概念

线程池的好处：线程池里的每一个线程代码结束后，并不会死亡,而是再次回到线程池中成为空闲状态，等待下一个对象调用。

### ②实现线程

1、创建方式

1. `newCachedThreadPool` 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
2. `newFixedThreadPool` 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
3. `newScheduledThreadPool` 创建一个定长线程池，支持定时及周期性任务执行。
4. `newSingleThreadExecutor` 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

```
ExecutorService pool = Executors.newFixedThreadPool(2);
```

2、执行线程

submit(Runnable) 和 execute(Runnable) 区别是前者可以返回一个**Future**对象，通过返回的Future对象，我们可以检查提交的任务是否执行完毕，请看下面执行的例子：

```
Future future = pool.submit(new MyRunnable());
future.get();  //如果任务正确完成，则返回null。

Future<Integer> f1 = pool.submit(new MyCall(10));

pool.execute(new MyRunnable());
```

3、结束线程池

```
pool.shutdown();
```

## ③代码实例

```
public class xiancheng {
    public static void main(String[] args) throws InterruptedException {
        //创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待
        ExecutorService pool = Executors.newFixedThreadPool(2);
        //开始
        pool.submit(new MyRunn());
        pool.submit(new MyRunn());
        //结束线程池
        pool.shutdown();
    }
}


class MyRunn implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() +"  "+ i);
        }
    }
}
```

```
pool-1-thread-1  0
pool-1-thread-2  0
pool-1-thread-1  1
pool-1-thread-1  2
pool-1-thread-2  1
pool-1-thread-2  2
pool-1-thread-2  3
pool-1-thread-2  4
pool-1-thread-2  5
```

# 8、加锁

## 1、synchronize同步代码块

synchronize(任意对象)

使用synchronize把要同步的语句包起来，不过需要用对象锁

```
synchronize(Obj对象) {
    //需要同步的代码
}
```

```
Object obj = new Object();
// 定义5张票
int tickets = 5;
@Override
public void run() {
// 模拟一直有票
while (true) {
    synchronized (obj) {
        if (tickets > 0) {
            System.out.println(getName() + "正在售票，还剩" + (--tickets) + "张票");
            Thread.sleep(100);
        }
    }
}
```

## 2、synchronize同步方法

synchronize

```
public synchronize way(){
    //需要同步的代码
}
```

```
public synchronized void way() {
    if (tickets > 0) {
        System.out.println(getName() + "正在售票，还剩" + (--tickets) + "张票");
        Thread.sleep(100);
    }
}
```

### 3、使用Lock锁

synchronize(字节码文件对象)

```
Lock l = new ReentrantLock();//创建锁
try {
    l.lock();
    if (tickets > 0) {
        System.out.println(getName() + "正在售票，还剩" + (--tickets) + "张票");
        Thread.sleep(100);
    }
} finally {
    // 释放锁
    l.unlock();
}
```

## # 9、线程安全问题

### （1）线程安全的类

```
StringBuffer sb=new StringBuffer( );
Vector<String> v=new Vector<String>( );
Hashtable<String,String> h=new Hashtable<String,String>( );
```

### （2）将List集合设为线程安全

```
List<String> list= Collections.synchronizedList( new ArrayList<String>() );
```

# 10、计时器Timer+TimerTask

## （1）Timer：定时

`Timer t = new Timer();`

构造方法：

① `public void schedule ([定义的TimerTask对象]，[毫秒])`

```
t.schedule(new MyTask(), 3000);//不结束
```

② `public void schedule ([定义的Ti...对象], [第一次run时间],[每隔n秒执行一次run])`

```
t.schedule(new MyTask(t), 3000, 1000)
```

## （2）TimerTask：任务

```
class MyTask extends TimerTask{
    @Override
    public void run() {
        System.out.println("和嘎嘎嘎");
    }
}
```

```
public class A9计时器 {
    public static void main(String[] args) {
        //造定时器对象
        Timer t = new Timer();
        //3秒后执行任务
        t.schedule(new MyTask(t), 3000);
    }
}

class MyTask extends TimerTask{
    private Timer t;
    public MyTask() {
    }
    public MyTask(Timer t) {
        this.t = t;
    }
    @Override
    public void run() {
        System.out.println("和嘎嘎嘎");
        t.cancel();//停止
```

```
    }
}
```

# 11、面试题

- 1、同步有几种方式，分别是什么?

  同步代码块，同步方法

- 2、启动一个线程是run ()还是start () ?它们的区别?

  run() :封装了被线程执行的代码,直接调用仅仅是普通方法的调用

  start() :启动线程，并由JVM自动调用run()方法，

- 3、sleep ()和wait ()方法的区别

  sleep() :必须指时间；不释放锁。

  wait() :可以不指定时间，也可以指定时间；释放锁

- 4、为什么wait() ,notify() ,notifyAll ()等方法都定义在0bject类中

  因为这些方法的调用是依赖于锁对象的，而同步代码块的锁对象是任意锁。

  而0bject代码任意的对象，所以，定义在这里面。

- 5、线程的生僉周期图

  新建--就緒--运行--死亡

  新建--就绪--运行--阻塞--就绪--运行---死亡

# 一、MySQL锁

# 1、行锁InnoDB

## 1、乐观锁：

读取的时候不加锁数据，写的时候判断有没有被修改过，如果被修改过就不更新

实现：在表中加一个字段，每次修改此行时值加1，在修改前先查询出此行的值，然后在修改时判断值是否相等，不相等就不更新。

适合大量的读取操作

## 2、悲观锁：

悲观锁分为共享锁和排它锁 适合大量写入的操作

只要获取事务就要修改，其他对这个事务操作的线程全部等待

### (1)共享锁(S锁)读锁

若事务我对数据对象A加上S锁，则就连我自己都只能读A， 不能修改A；其他事务只能再对他加S锁，而不能加X锁，直到我释放A上的S锁。

实现：在执行语句后面加上 lock in share mode 就代表对某些资源加上共享锁了。

### （2）排他锁(X锁)写锁

又称为写锁、独占锁

若【事务我】对数据对象A加上X锁，则只允许我读取和修改A，其他任何事务都不能再对A加任何类型的锁， 不能读写 ，直到我释放A上的锁。

实现：在需要执行的语句后面加上for update

### 3、二者区别

1、共享锁只用于表级，排他锁用于表级和行级。

2、加了共享锁的对象，可以继续加共享锁，不能再加排他锁。加了排他锁后，不能再加任何锁。

4、当执行DDL操作时，就需要在全表上加排他锁。

# 2、表锁(非事务引擎)

- 使用表级锁定的主要有MyISAM，MEMORY，CSV等一些 非事务性存储引擎。

分为表共享锁和表独占锁

select时之前自动加读锁

# 二、分布式锁

## # 1、基于mysql的分布式锁

优点：直接借助数据库容易理解

缺点： 在使用关系型数据库实现的过程中会出现各种问题，例如数据库单点问题和可重入问题，并且在解决过程中会使得整个方案越来越复杂

## # 2、基于Redis的分布式锁

### 优点：

性能好，实现起来较为方便

### 缺点：

- key的过期时间设置难以确定，如何设置的失效时间太短，方法没等执行完，锁就自动释放了，那么就会产生并发问题。如果设置的时间太长，其他获取锁的线程就可能要平白的多等一段时间。
- Redis的集群部署虽然能解决单点问题，但是并不是强一致性的，锁的不够健壮

| 方案 | 复杂度 | 性能 | 可靠性 | 学习成本 |
| --- | --- | --- | --- | --- |
| 基于关系型数据库 | 低 | 低 | 低 | 低 |
| 基于Redis | 中 | 高 | 中 | 中 |
| 基于zookeeper | 高 | 中 | 高 | 高 |

### 实现

Redis实现分布式锁的主要命令：`setnx`，该命令的作用是**当key不存在时设置key的值**（加锁），当Key存在时，什么都不做。

## Redis如何实现分布式锁

可以直接通过 `set [key] [unique_value] px nx px [秒]` 命令实现加锁， 通过Lua脚本实现解锁。

```
/**
 * random_value 是客户端生成的唯一的字符串。
 * NX 代表只在键不存在时，才对键进行设置操作。
 * PX 5000 设置键的过期时间为5000毫秒。
 */

SET key unique_value NX PX 5000
```

```
/**
     * 创建分布式锁执行事务后删除锁
     */
    public void lockRedis(String email, int code) {


        try {
            String key = "lock_redis";//这个key是专门用来放置分布式锁的
            redisTemplate.watch(key);//key加乐观锁
            Boolean lock = redisTemplate.opsForValue().setIfAbsent(key, redisID, 5,
TimeUnit.SECONDS);//获取锁

            String lockValue = redisTemplate.opsForValue().get(key);

            if (lock) {

                //此处是实际调用代码块
                redisTemplate.opsForValue().set("email_code_" + email, code + "", 3,
TimeUnit.MINUTES);//1分钟过期

                //正要删除锁时，锁已过期，别人已设置新值。那么我们删除的是别人的锁
                //解决：删除锁必须保证原子性。使用redis+Lua脚本完成
                //if redis.call("get",keys[1] == argv[1]
                //      then return redis.call("get",keys[1])
                //      else return 0
                //      end
                String script = "if redis.call(\"get\",KEYS[1]) == ARGV[1] then\n" +
                        "    return redis.call(\"del\",KEYS[1])\n" +
                        "else\n" +
                        "    return 0\n" +
                        "end";
```

```
                redisTemplate.execute(new DefaultRedisScript<Long>(script, Long.class),
Arrays.asList(key), lockValue);

            } else {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                lockRedis(email, code);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            redisTemplate.unwatch(); //清除连接中的所有被监视的key
        }
    }
```

# 三、Java锁

| 类别 | **synchronized** | **Lock** |
| --- | --- | --- |
| 存在层次 | Java的关键字、jvm层面 | 是一个类 |
| 锁释放 | 自动释放；若出现异常jvm也会让它自动释放 | 需手动释放( finally() 中写上 unLock() )，否则死锁 |
| 锁获取 | A线程拿到锁,此时B只能一直等 | A线程拿到锁,此时B可以响应中断去干别的 |
| 锁状态 | 无法判断有没有获取成功 | 可以判断 |
| 锁类型 | 可重入、不可中断、非公平 | 可重入、可中断、可公平 |
| 性能 | 少量同步 | 大量同步（大量线程同时竞争） |

# 四、Java中锁的种类

## # 1、可重入锁

就是说我刚刚获取了这个锁之后进去了，然后我再进去，是不需要再获取锁的

锁具有可重入性，`synchronized`和`Lock`接口的`ReentrantLock`都是可重入锁，从一个线程到另一个线程不需要再申请而是可以直接进去.

```
new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    lock.lock();
                    System.out.println("第1次获取锁，这个锁是：" + lock);

                    int index = 1;
                    while (true) {
                        try {
                            lock.lock();
                            System.out.println("第" + (++index) + "次获取锁，这个锁是：" + lock);

                            Thread.sleep(new Random().nextInt(200));
                            if (index == 10) {
                                break;
                            }
                        } finally {
                            lock.unlock();
                        }
                    }
                } finally {
                    lock.unlock();
                }
            }
        }).start();
```

## # 2、可中断锁

在Java中，`synchronized`就不是可中断锁，而`Lock`是可中断锁。

如果线程A正在执行锁中的代码，线程B不想等待了，想先处理其他事情，我们可以让B中断自己或者在C线程中中断B，这种就是可中断锁。

# 3、公平锁

多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（先来先服务）获得该锁

- synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。
- ReentrantLock和ReadWriteLock接口的ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。
- 我们可以在创建ReentrantLock对象时，通过以下方式来设置锁的公平性：

```
ReentrantLock lock =  new  ReentrantLock( true );
```

另外在ReentrantLock类中定义了很多方法，比如：

isFair() //判断锁是否是公平锁

isLocked() //判断锁是否被任何线程获取了

isHeldByCurrentThread() //判断锁是否被当前线程获取了

hasQueuedThreads() //判断是否有线程在等待该锁

在ReentrantReadWriteLock中也有类似的方法，同样也可以设置为公平锁和非公平锁。不过要记住，ReentrantReadWriteLock并未实现Lock接口，它实现的是ReadWriteLock接口。

# 4、ReentrantReadWriteLock

ReentrantReadWriteLock里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

下面这个是synchronize

这段程序的输出结果会是，直到**thread1**执行完读操作之后，才会打印**thread2**执行读操作的信息。

```
public  class  Demo {
    public  static  void  main(String[] args)  {
        final Demo test =  new Demo();

        new  Thread(){
            public  void  run() {
                test.get(Thread.currentThread());
            };
        }.start();

        new  Thread(){
```

```
            public  void  run() {
                test.get(Thread.currentThread());
            };
        }.start();

    }

    public  synchronized  void  get(Thread thread) {
        long  start = System.currentTimeMillis();
        while (System.currentTimeMillis() - start <=  1 ) {
            System.out.println(thread.getName()+ "正在进行读操作" );
        }
        System.out.println(thread.getName()+ "读操作完毕" );
    }
}
```

下面这个是ReentrantReadWriteLock

**thread1**和**thread2**在同时进行读操作。

- 如果有 线程A 已经占用了 读锁 ， 则此时 线程B 如果要申请 写锁 ， 则 线程B 会一直等待A释放
- 如果有 线程A 已经占用了 写锁 ， 则此时 线程B 读写都要一直等待 线程A 释放。

```
public class Test2 {
    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

    public static void main(String[] args) {
        final Test2 test = new Test2();
        new Thread() {
            public void run() {
                test.get(Thread.currentThread());
            }
        }.start();

        new Thread() {
            public void run() {
                test.get(Thread.currentThread());
            }
        }.start();
    }

    public void get(Thread thread) {
        rwl.readLock().lock();
        try {
            long start = System.currentTimeMillis();

            while (System.currentTimeMillis() - start <= 1) {
                System.out.println(thread.getName() + "正在进行读操作");
            }
            System.out.println(thread.getName() + "读操作完毕");
        } finally {
            rwl.readLock().unlock();
```

```
        }
    }
}
```

# 五、线程间通讯

## # 1、使用 volatile 关键字

```
//定义一个共享变量来实现通信，它需要是volatile修饰，否则线程不能及时感知
static volatile boolean notice = false;
```

多个线程同时监听一个变量，当这个变量发生变化的时候 ，线程能够感知并执行相应的业务。这也是最简单的一种实现方式

## # 2、Object类的 wait()，notify()

Object类提供了线程间通信的方法：wait() 、 notify() 、 notifyaAl()

> 注意： **wait**和 **notify**必须配合**synchronized**使用，**wait**方法释放锁，**notify**方法不释放锁
>
> 程A发出notify()唤醒通知之后，依然是走完了自己线程的业务之后，线程B才开始执行，这也正好说明了，notify()方法不释放锁，而wait()方法释放锁。

```
public class TestSync {
    public static void main(String[] args) throws InterruptedException {
        // 定义一个锁对象
        Object lock = new Object();
        List<String> list = new ArrayList<>();
        // 实现线程A
        Thread threadA = new Thread(() -> {
            synchronized (lock) {
                for (int i = 1; i <= 10; i++) {
                    list.add("abc");
                    System.out.println("线程A向list中添加一个元素，此时list中的元素个数为：" +
list.size());
```

```
                Thread.sleep(500);
                if (list.size() == 5)  lock.notify();// 唤醒B线程
            }
        }
    });
    // 实现线程B
    Thread threadB = new Thread(() -> {
        while (true) {
            synchronized (lock) {
                if (list.size() != 5)   lock.wait();
                System.out.println("线程B收到通知，开始执行自己的业务...");
            }
        }
    });
    //  需要先启动线程B
    threadB.start();
    Thread.sleep(1000);
    // 再启动线程A
    threadA.start();
  }
}
```

# 3、JUC工具类 CountDownLatch

jdk1.5之后在java.util.concurrent包下提供了很多并发编程相关的工具类，简化了我们的并发编程代码的书写，CountDownLatch基于AQS框架，相当于也是维护了一个线程间共享变量state

```
public class TestSync4 {
    public static void main(String[] args) {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        List<String> list = new ArrayList<>();
        // 实现线程A
        Thread threadA = new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                list.add("abc");
                System.out.println("线程A向list中添加一个元素，此时list中的元素个数为：" + list.size());
                if (list.size() == 5)
                    countDownLatch.countDown();//用于使计数器减一，其一般是执行任务的线程调用
            }
        });
        // 实现线程B
        Thread threadB = new Thread(() -> {
            if (list.size() != 5)       countDownLatch.await(); //主线程唤醒
            System.out.println("线程B收到通知，开始执行自己的业务...");
        });
        //  需要先启动线程B
        threadB.start();
        Thread.sleep(1000);

        // 再启动线程A
```

```
        threadA.start();
    }
}
```

# 4、 ReentrantLock 结合 Condition

这种方式并不是很好，代码编写复杂，A在唤醒操作之后，并不释放锁。这种方法跟 **Object** 的 **wait()** 和 **notify()** 一样。

```
public class TestSync3 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

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

        // 实现线程A
        Thread threadA = new Thread(() -> {
            lock.lock();  //加锁
            for (int i = 1; i <= 10; i++) {
                list.add("abc");
                System.out.println("线程A向list中添加一个元素，此时list中的元素个数为: " + list.size());
                Thread.sleep(500);
                if (list.size() == 5)
                    condition.signal();  //唤醒等待的线程，但是等待的还是要等自己先执行完释放
            }
            lock.unlock();
        });
        // 实现线程B
        Thread threadB = new Thread(() -> {
            lock.lock();
            System.out.println("先执行B");
            if (list.size() != 5) {
                 condition.await(); //让线程B进入等待,执行别的线程
            }
            System.out.println("线程B收到通知，开始执行自己的业务...");
            lock.unlock(); //释放锁
        });

        threadB.start();
        Thread.sleep(1000);
        threadA.start();
    }
}
```

# 5、LockSupport实现线程间阻塞唤醒

**LockSupport** 是一种非常灵活的实现线程间阻塞和唤醒的工具，使用它不用关注是等待线程先进行还是唤醒线程先运行，但是得知道线程的名字。

```
public class TestSync2 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        // 实现线程B
        final Thread threadB = new Thread(() -> {
            if (list.size() != 5) {
                LockSupport.park(); //挂起当前线程
            }
            System.out.println("线程B收到通知，开始执行自己的业务...");
        });
        // 实现线程A
        Thread threadA = new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                list.add("abc");
                System.out.println("线程A向list中添加一个元素，此时list中的元素个数为：" + list.size());
                Thread.sleep(500);
                if (list.size() == 5)
                    LockSupport.unpark(threadB);//恢复threadB线程
            }
        });
        threadA.start();
        threadB.start();
    }
}
```

# 七、如何处理线程的返回值

## 1、主线程一直等待

等到主线程返回值给他，再执行

```
while (cw.value == null){
    Thread.sleep(100);
}
```

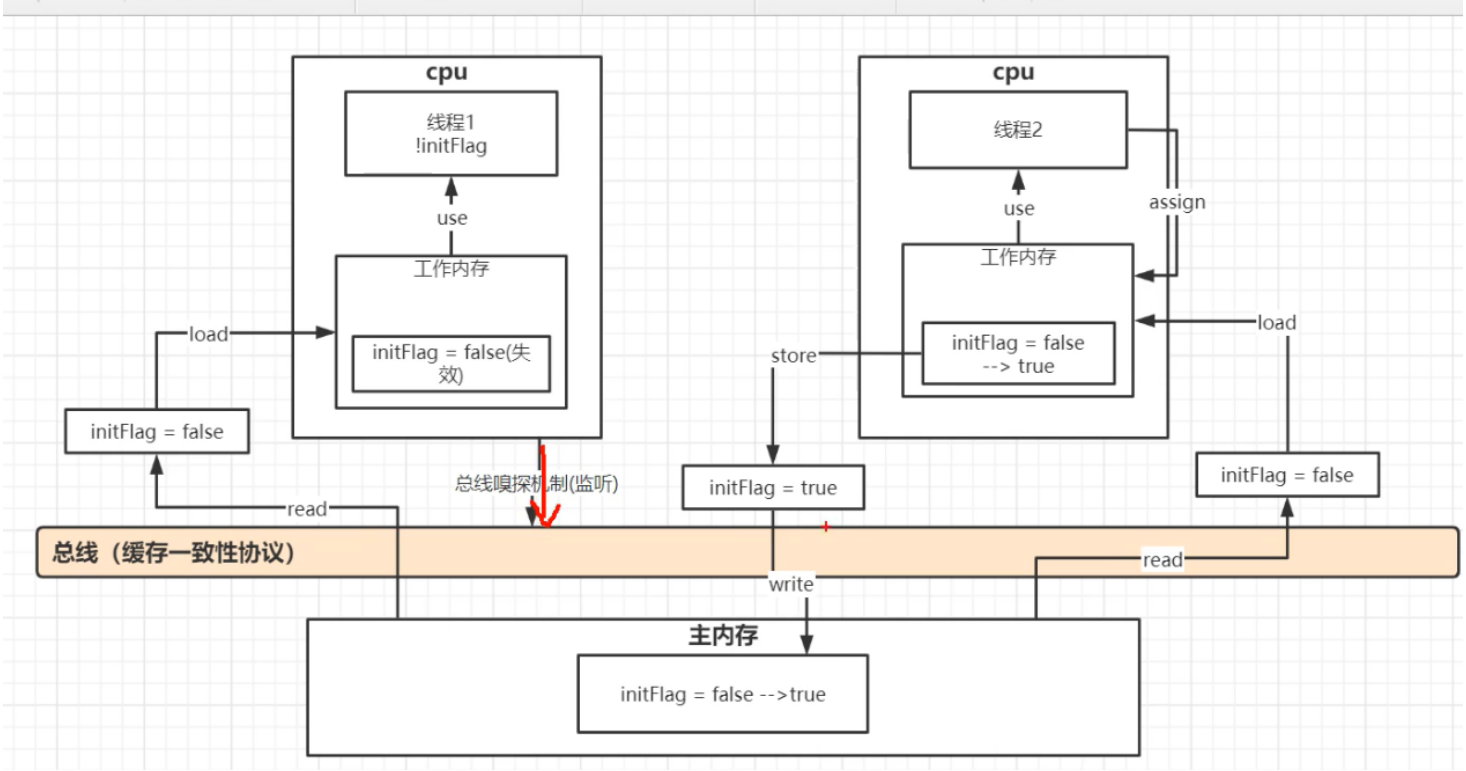
## 2、使用Thread类的join()方法阻塞当前线程以等待子线程处理完毕

```
thread.join();
```

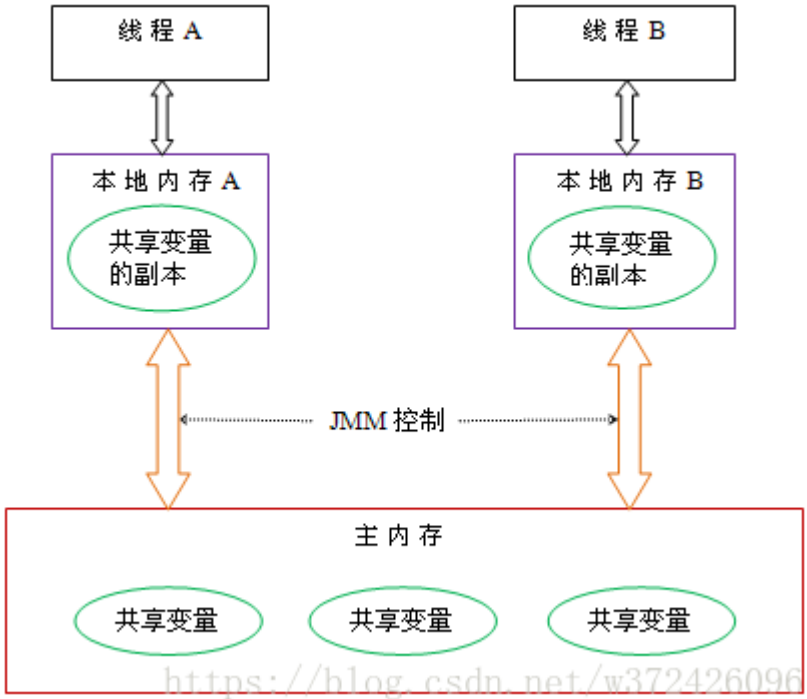
### 3、通过Callable接口实现：通过FutureTask Or 线程池获取

### 4、线程池

# 八、JMM内存模型




工作内存是个副本，不共享的若要共享：volatile





# 九、线程池执行过程

参数：最大核心数，工作队列，最大线程数，非核心线程生命时间，拒绝策略