进程

程序由指令和数据组成，但这些指令要运行，数据要读写，就必须将指令加载到CPU，数据加载到内存，在指令运行过程中还需要用到磁盘，网络等设备，进程就是用来加载指令，管理内存，管理IO的

当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程

进程可以视为程序的一个实例，大部分程序可以同时运行多个实例进程

线程

一个进程之内可以分为一到多个线程

一个线程就是一个指令流，将指令流中的一条条指令以一定的顺序交给CPU执行

java中，线程作为最小调度单位，进程作为资源分配最小单位，在windows中进程是不活动的，只是作为线程的容器

对比

进程基本上相互独立，而线程存在于进程内，是进程的一个子集

进程拥有共享的资源，如内存空间等，使其内部的线程共享

进程间通信较为复杂

同一台计算机的进程通信称为IPC

不同计算机之间的进程通信，需要通过网络，并遵守共同的协议，例如http

线程通信相对简单，因为它们共享进程内的内存，一个例子是多个线程可以访问同一个共享变量

线程更轻量，线程上下文切换成本一般上要比进程上下文切换低

并行和串行:串行是做完一件事再做另一件事，并行是多件事同时做

并发多件事轮着做，操作系统中有一个组件叫任务调度器，将CPU的是片分给不同线程使用，由于cpu在线程时间片很短的切换，人类感觉是同时运行的，总结为微观串行，宏观上并行。一般将这种线程轮流使用CPU的做法叫做并发

同步和异步

从方法调用的角度来讲，如果

需要等待结果返回，才能继续运行就是同步

不需要等待结果返回，就能继续运行就是异步

注意：同步在多线程还有另外一层意思，是让多个线程步调一致

Thread和Runnable

一个是类一个是接口

Thread是线程和任务合并一起，

Runnable是任务

推荐用Runnable，更容易与线程池等高级API配合，让任务类脱离了Thread继承体系，更灵活

```
    public static void main1(String[] args) {
        Runnable myThread = new MyThread ();
        Thread thread = new Thread(myThread);
        thread.start();
    }

    public static void main2(String[] args) {
       Thread thread = new Thread(new Runnable() {
           @Override
           public void run() {
               System.out.println("Runnable 匿名内部类创建线程");
           }
       });
       thread.start();
   }

   /**
         *切记不是new 接口！！！
         *
         * 匿名内部类(Anonymous Inner Class)，在创建实例的同时给出类的定义，所有这些在一个表达式中完成。
         */
        Runnable runnable=new Runnable() {
            @Override
            public void run() {
                System.out.println("");
            }
        };
```

栈和栈帧

JVM是由堆，栈，方法区所组成，其中栈内存是给线程用的，每一个线程启动后，虚拟机就会为其分配一块栈内存

每个栈由多个栈帧组成，对应着每次方法调用时所占用的内存

每个线程只能有一个活动栈帧，对应着当前正在执行的那个方法

该知识点看视频理解：

https://www.bilibili.com/video/BV16J411h7Rd/?is_story_h5=false&p=21&share_from=ugc&share_medium=iphone&share_plat=ios&share_session_id=5E3B8AEE-4D2F-462F-B255-8EBF50554DB5&share_source=WEIXIN&share_tag=s_i&timestamp=1666249919&unique_k=xgdcTHb&vd_source=9a8eaac8fc4342dbc67d1925a4cce27c

线程上下文切换

因为以下一些原因导致cpu不再执行当前的线程，转而执行另一个线程的代码：线程的cpu时间片用完，垃圾回收，有更高优先级的线程需要运行，线程自己调用了sleep等方法

当context switch发生时候，contextswitch频繁发生会影响性能，需要由操作系统保存当前线程的状态，并恢复另一个线程的状态，java中对应的概念就是程序计数器，它的作用是记住下一条jvm指令的执行地址，是线程私有的

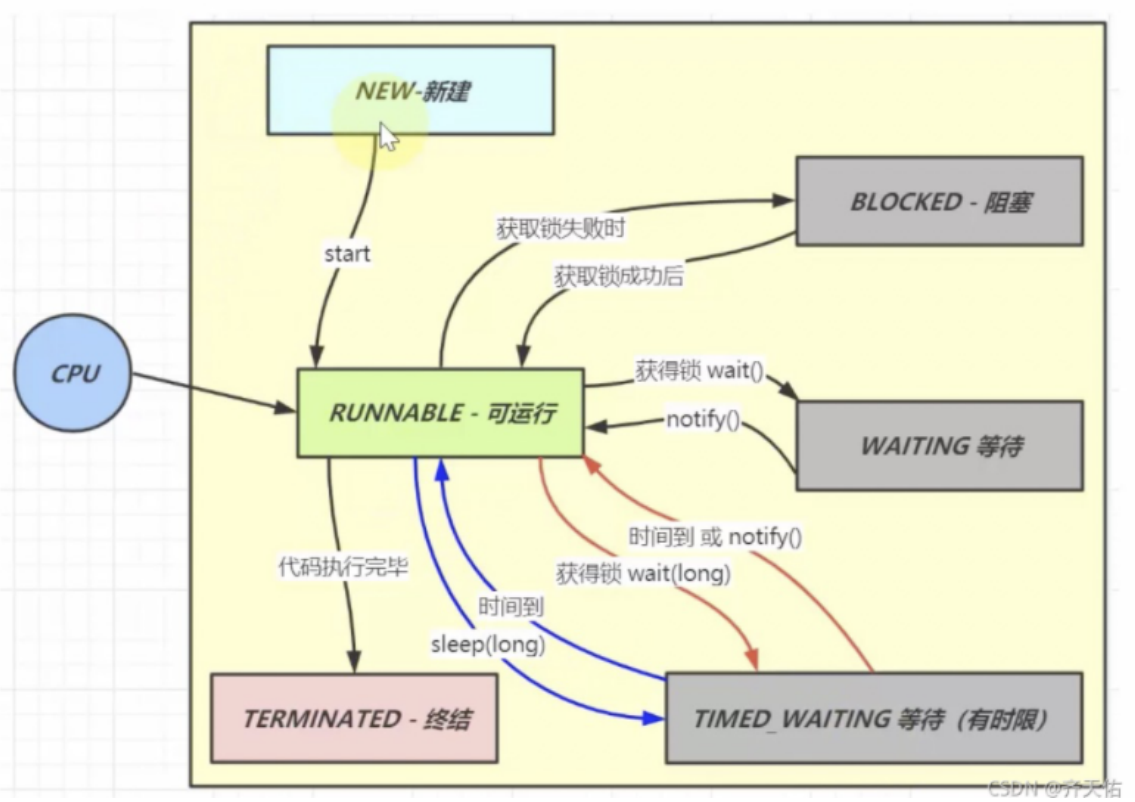
状态包括程序计数器，虚拟机栈中每一个栈帧的信息，如局部变量，操作数栈，返回地址等

方法大全

调用start方法方可启动线程，而run方法只是thread的一个普通方法调用，还是在主线程里执行。

方法名	static	功能说明	注意
start		启动一个新线程，在新的线程运行run方法中的代码	start方法只是让线程进入就绪，里面代码不一定立刻运行，每一个线程对象的start方法只能调用一次，如果调用多次会出现异常illegalthread
run		新线程启动后调用的方法	如果在thread对象时传递了runnable参数，则线程启动会调用runnable中的run方法，否则默认不执行任何操作，但可以创建thread子类对象，覆盖默认位置
join		等待线程运行结束	在a线程中b.join（），则a会等b执行结束
join（long n）		等待线程运行结束最多等待n毫秒	
getId		获取线程长整型的id	id唯一
getName		获取线程名	
setName（String）		修改线程名	
getPriority		获取线程优先级	
setPriority（int）		修改线程优先级	java中规定线程优先级是1-10的整数，较大的优先级能提高线程被cpu调度的概率
getState		获取线程状态	Thread.State，java中线程状态是用6个enum表示NEW,RUNNABLE,BLOCKED,WAITING,TIMED_WAITING,TERMINATED
isInterrupted		判断是否被打断	打断sleep，wait，join，则消除以上“等待”，但状态会被清空，只会是false，若是打断正常运行的线程，相当于线程被发了一个中断信号，不会被清空打断状态，则是true
interrupt		打断线程	例如在a线程中b.interrupt，则会中断一下b的线程/
isAlive		线程是否存货	

线程状态



1. 新建状态：

> 使用new关键字创建一个thread对象，刚刚创建出的这个线程就处于新建状态。在这个状态的线程没有与操作系真正的线程产生关联，仅仅是一个java对象。

2. 可运行：

> 正在进行运行的线程，只有处于可运行状态的线程才会得到cpu资源。

3. 阻塞：

> 在可运行阶段争抢锁失败的线程就会从可运行--->阻塞

4. 等待：

> 可运行状态争抢锁成功，但是资源不满足，主动放弃锁（调用wait()方法）。条件满足后再恢复可运行状态（调用notiy()方法）。

5. 有时限等待：

> 类似于等待，不过区别在于有一个等待的时间，到达等待时间后或者调用notiy()，都能恢复为可运行状态。
>
> 有两种方式可以进入有时限等待：wait(Long)和sleep(Long)

6. 终结：

> 代码全部执行完毕后，会进入到终结状态，释放所有的资源。

状态	含义	说明
NEW	初始	线程对象已经创建，但尚未启动，既还没有调用start()方法
RUNNABLE	运行	已经调用了线程的start()方法，又分为就绪（Ready）和运行中（Running）两种状态就绪：该线程等待获取CPU的使用权，出属于就绪状态的线程，随时可能被CPU调度执行；运行中：获得CPU时间片后变为运行中状态（Running），线程只能从就绪状态进入到运行中状态
BLOCKED	阻塞	表示线程阻塞于锁，等待获取锁进入同步块/方法
WAITING	等待	处于这种状态的线程不会被分配CPU执行时间，它们要等待被显式地唤醒，否则会处于无限期等待的状态。
TIMED_WAITING	超时等待	该状态不同于WAITING，它可以在达到一定时间后自动唤醒
TERMINATED	终止	表示该线程已经执行完毕，既run()方法执行结束

runnable：java将运行状态，可运行状态以及阻塞去获取东西时候状态都将runnable

sleep

1调用 sleep 会让当前线程从 Rumning 进入 Timed Traiting 状态
2.其它线程可以使用 interrupt 方法打断正在睡眠的线程，这时 sleep 方法会抛出 InterruptedException
3.睡眠结束后的线程未必会立刻得到执行

4·建议用 TimeUnit 的sleep 代替 Thread 的sleep 来获得更好的可读性

yield

1调用 yield 会让当前线程从 Running 进入Runnabie 状态，然后调度执行其它同优先级的线程。如果这时没
有同优先级的线程，那么不能保证让当前线程暂停的效果
2.具体的实现依赖于操作系统的任务调度器
线程优先级

interrupt

打断sleep，wait，join，则消除以上“等待”，但状态会被清空，只会是false，若是打断正常运行的线程，相当于线程被发了一个中断信号，不会被清空打断状态，则是true

LockSupport.park()

让当前线程睡眠

LockSupport.unpark(Thread)

唤醒线程

不推荐的方法

stop　停止线程运行

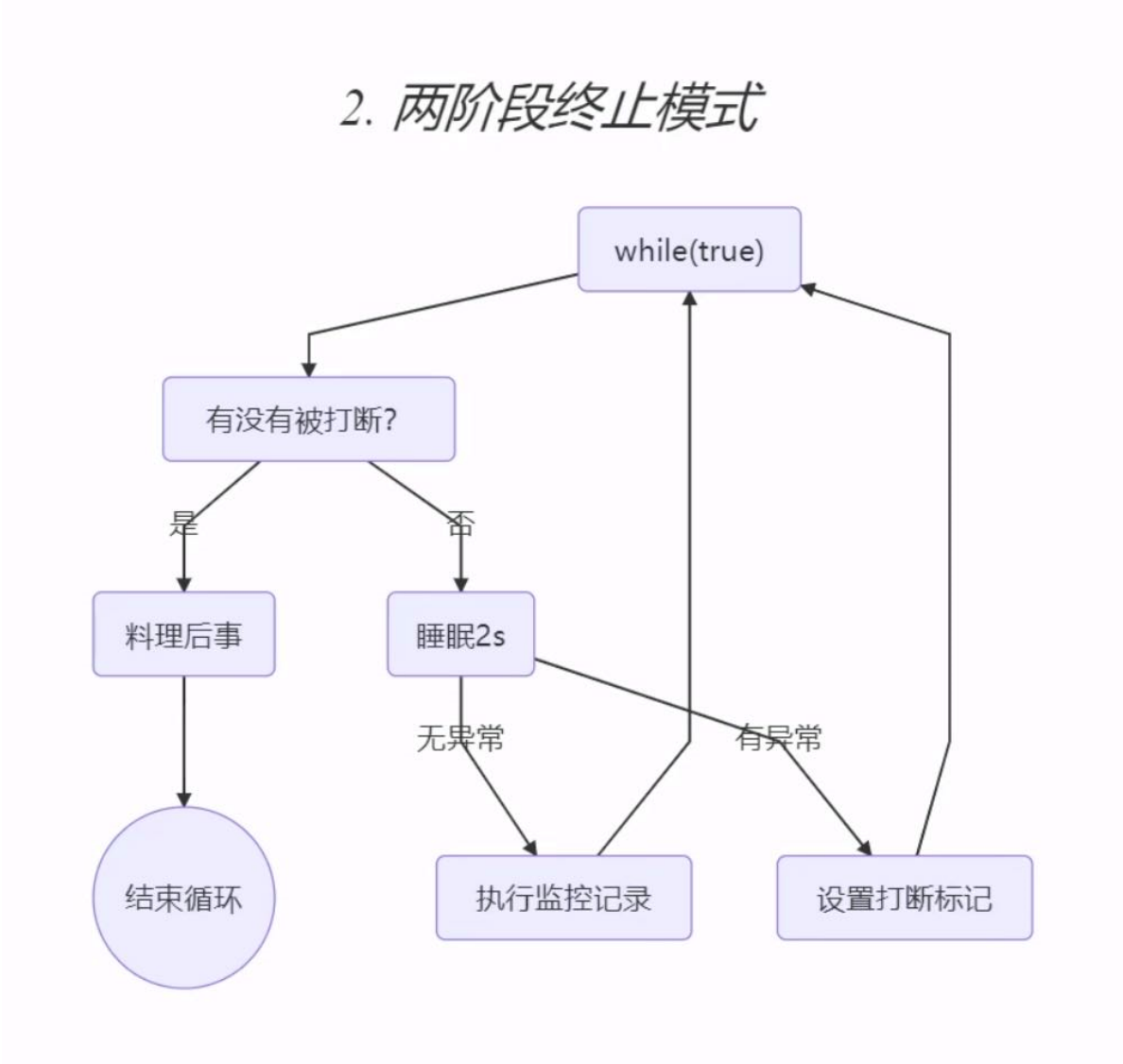
suspend 挂起线程

resume　恢复线程

终止方法

不推荐用stop方法杀死线程，因为stop方法会真正杀死线程，如果这时候线程锁住了共享资源，那么当它被杀死后就再也没有机会释放锁，其他线程将无法获取锁

两阶段终止模式



线程优先级

•线程优先级会提示 （hint）调度器优先调度该线程，但它仅仅是
一个提示，调度器可以忽略它
•如果cpu 比较忙，那么优先级高的线程会获得更多的时间片，但cpu 闲时，优先级几乎没作用

守护线程

setDaemon(boolean);

默认情况下，java进程需要等待所有线程都运行结束才会结束，有一种特殊的线程叫做守护线程，只要其他非守护线程运行结束，它会被强制结束

线程共享的问题

读操作则线程安全!!!!

由于分时系统造成的线程安全问题，共享的数据！！！！即运行结果未来得及更新数据导致数据错误，即对共享资源读写操作时发生指令交错

一段代码块内对共享资源的多线程读写操作，称这段代码块为临界区，

```
synchronized修饰代码块，可用对象或者class作为锁
synchronized (this)
{
    System.out.println("");
}

synchronized修饰普通方法中的同步锁就是引用了这个方法的对象本身，即"this"。
synchronized作用于静态方法时，其锁就是当前类的class对象锁
```

通过锁来执行，即写前获取锁，写后释放锁

synchronized实际时用对象锁保证了临界区内代码的原子性，临界区内的代码对外不可分割，不会被线程切换所打断！！！！

注意代码的区域，除了都要加锁（不加锁可用理解为翻窗户进去，打破规则的人）外还要锁对象一致

非共享数据，即局部变量不是共享的，不用考虑共享问题

线程安全

即多个线程调用同一个实例某个方法是线程安全，但如果多个方法就不保证原子性

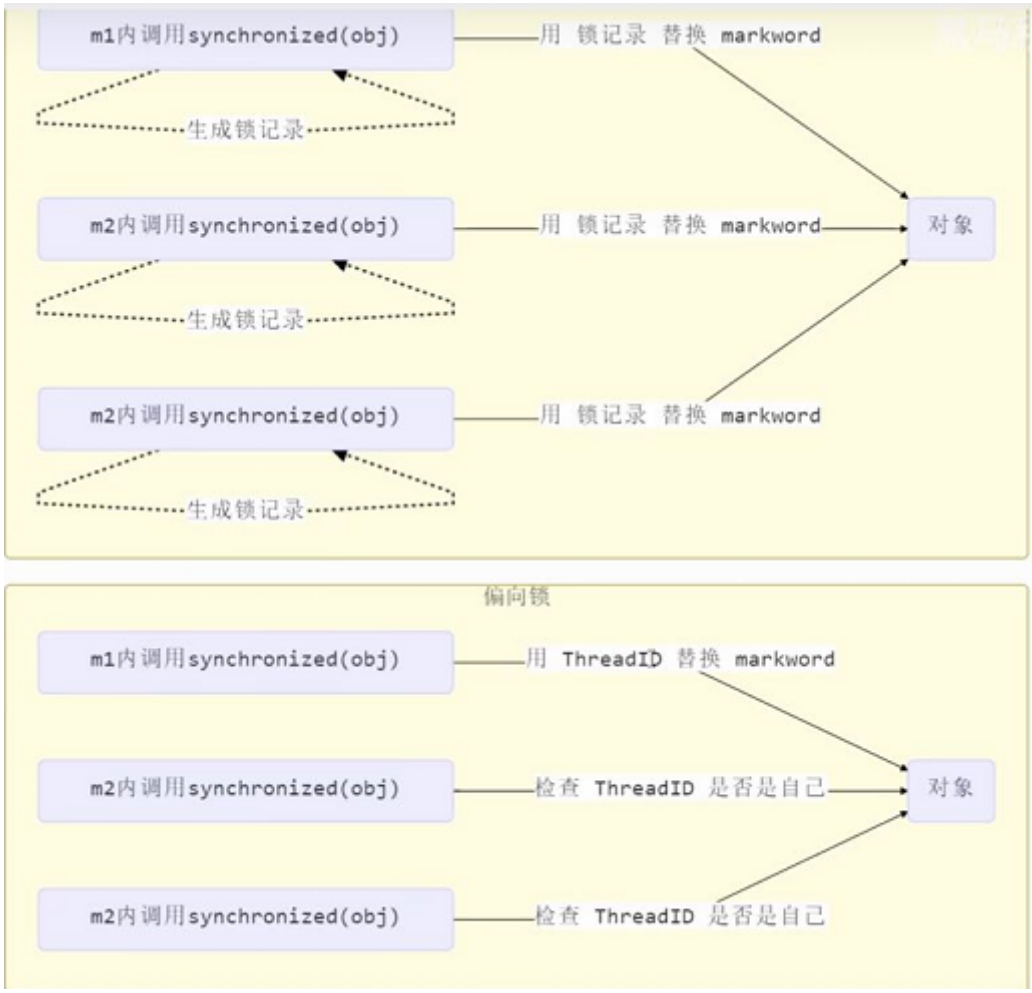
字符串属于线程安全，因为写操作，是去引用另一个字符串，而不是修改。string类为什么设置成final，因为不然的话子类可以破坏其线程安全

获取锁

获取不到锁，先重试，不行再阻塞

如果是自己执行了synchronized锁重入，则会再添加一条锁记录作为重入的计数

偏向锁



原来锁记录要替换markword，如果是自己线程，也还是要锁记录替换。偏向锁则第二次用锁是检查而不是替换。

wait和notify

wait和notify是线程间进行协作的手段，wait和notify使用都先必须获得对象的锁，不获取锁直接用Object对象调用wait和notify,会报错java.lang.IllegalMonitorStateException。

```
//创建对象
Object lock=new Object();
//.....
//线程A
synchronized(lock){
   while(条件不成立){
     lock.wait();
  }
  //说明条件成立，继续执行该线程拿到资源后的工作
}
//...
synchronized(lock){
//做一下处理...完成输送资源的准备
     lock.notifyAll();

}
```

应用场景是在获取锁后发现没有资源，然后锁.wait，然后另一个线程获取锁之后来唤醒它，锁.notify是唤醒一个，notifyAll是唤醒全部

wait（long timeout ）

最多等多久

防止虚假唤醒

while执行被唤醒后是不是真的可以醒，不可以就接着睡觉

sleep和wait的区别

sleep是Thread方法，而wait是object方法

sleep不需要synchronized，wait需要

sleep睡眠时候不会释放锁，wait会

状态都是timed——waiting

wait和park的区别

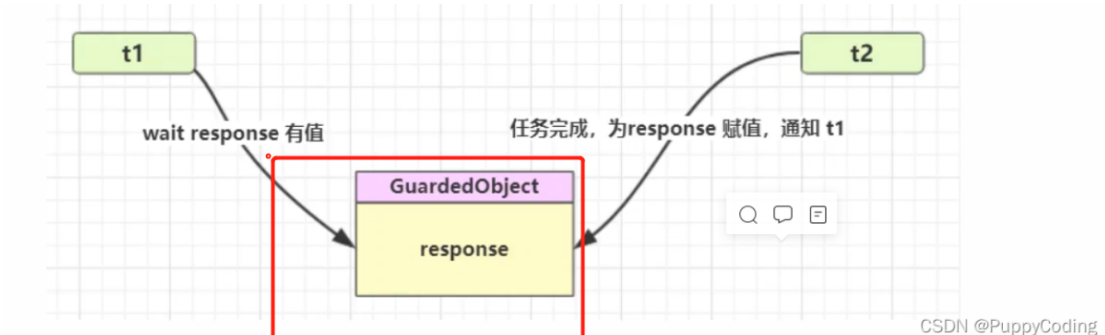
park不需要synchronized，是以线程为单位来阻塞和唤醒线程

如果没有先park而先unpark一次或者多次，则下次park会无效

同步模式之保护性暂停

guarded：监视

- 有一个结果需要从一个线程传递到另一个线程，让他们关联同一个 GuardedObject
- 如果有结果不断从一个线程到另外一个线程那么可以使用消息队列（生产者/消费者）
- JDK中，join的实现，Future的实现，采用的就是此模式
- 因为要等待另一方的结果，因此归类到同步模式



保护性暂停作为一种 设计模式，其关键点在于GuardedObject，它有一个response的属性，t1想要获取它，需要**等待**，直到t2得到它时，将response返回给GuardedObject，再由GuardedObject返回给t1。

在这里需要用到线程 的wait()和notify()方法来完成代码编写。

```
// timeout 表示最大的等待时间（即超时时间）
    public Object get(long timeout) {
        long startTime = System.currentTimeMillis();
        synchronized (this) {
            // 经历的时间
            long passtime = 0;
            //为了防止虚假唤醒 使用while条件判断 直到response不会空时才打破循环
            while (response == null) {
                // 经历的时间大于等于超时时间 则跳出循环
                if (passtime >= timeout) {
                    break;
```

```
                }
                try {
                    this.wait(timeout);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 开始时间 - 当前时间 = 经历的时间
                passtime = startTime - System.currentTimeMillis();
            }
            return response;
        }
    }
```

异步模式之生产者/消费者

之所以异步，是因为消息被消费可能被推迟，而不是像同步模式之保护性暂停一一对应去被消费。

多把锁

两个synchronized的锁不一样，则互不相干。

锁粒度细分好处是增强并发度。可以理解为大家可以进入同一个大门，进入不同的房间。

坏处就是会导致死锁

活锁

没有阻塞，互相改变对方状态导致对方无法停止。

顺序加锁

所有线程都按同一个顺序获取锁，就不会死锁。坏处就是即使不用的资源，也先拿过来了

reentrantLock可重入锁

共同点就是支持可重入：可重入是指同一个线程成为锁的拥有者，有权利再次获取这把锁

相比于synchronized

它可中断，可设置超时时间，可设置为公平锁，支持多个条件变量

```
        ReentrantLock reentrantLock = new ReentrantLock();
        reentrantLock.lock();   //获取锁，获取不到就阻塞
        reentrantLock.tryLock();    //能获取锁就true,不能就false，可用该方法判断是否执
行接下来操作
        try {
            reentrantLock.tryLock(1, TimeUnit.MINUTES);//尝试获取锁多久，可打断
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            reentrantLock.lockInterruptibly();  //获取可打断的锁，其他线程对该线
程.interrupt()
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
class  a extends ReentrantLock{
    //可把对象当成锁
}
```

用reentrantLock可解决顺序加锁中的死锁问题，即获取a锁后，b锁一直获取不到，就把a锁释放掉

相对synchronized防止虚假唤醒

```
ReentrantLock reentrantLock = new ReentrantLock();
//创建阻塞对了
Condition condition1 = reentrantLock.newCondition();
//创建阻塞对了
Condition condition2 = reentrantLock.newCondition();
int a=0;
new Thread(()->{
    //获取锁
   reentrantLock.lock();
   while (a!=0)
   {
       try {
           //进入哪一个阻塞队列等待
           condition1.await();
       } catch (InterruptedException e) {
           e.printStackTrace();
       }
   }
   reentrantLock.unlock();//释放锁
});

//唤醒一个线程
condition1.signal();
//唤醒全部线程
condition1.signalAll();
```

同步模式顺序模式

想先线程二再线程一，思路可用wait-notify，也可以用await和signal，也可以用park和unpark

互斥和同步

互斥:可以利用synchronized和lock达到共享资源互斥效果

同步:也可以用wait/notify或者lock的条件变量来达到线程间通信效果

可见性问题

volatile 易变关键字，可以修饰成员变量和局部变量，避免线程从自己的工作缓存中查找变量的值（会出现主存已经修改了但工作缓存没修改的优化问题），该关键字要求线程必须到主存获取它的值，线程操作volatile遍历都是直接操作主存

synchronized和volatile的区别

或者不加volatile，在symchronized代码区域操作共享变量。但symchronized属于重量级操作，好处就是多了保证代码块的原子性。

同步模式值balking

即balking犹豫模式用在一个线程发现另一个线程或本线程已经在做了某一件相同的事，那么本线程就无需再做，直接结束返回

```
synchronizde(this)
{
    if（start）
        return;
    start=true;
}
```

有序性问题

读屏障和写屏障:防止指令重排序可以给变量加volatile关键字，防止它以及它前面的代码重排序去读或者写。

注意synchronized不能完全阻止有序性

前提是变量全由synchronized管理

CAS

思路

compare and set比较并设置

也有compare and swap比较并交换

即获取初始值后修改，再重新获取最新的初始值判断初始值有没有改变，有就更新失败重新操作。

CAS中存在ABA问题

真正要做到严谨的CAS机制，我们在compare阶段不仅要比较期望值A和地址V中的实际值，还要比较变量的版本号是否一致。

核心知识（CAS原理）

https://blog.csdn.net/qq_32998153/article/details/79529704?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216665288581678241702448 5%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=166652885816782417024485&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-79529704-null-null.142^v59^pc_rank_34_1,201^v3^control_1&utm_term=cas&spm=1018.2226.3001.4187

CAS和synchronized

为什么无锁效率高
•无锁情况下，即使重试失败，线程始终在高速运行，没有停歇，而 synchronized 会让线程在没有获得锁的时候，发生上下文切换，进入阻塞。打个比喻
•线程就好像高速跑道上的赛车，高速运行时，速度超快，
一旦发生上下文切换，就好比赛车要减速、熄火等被唤醒又得重新打火、启动、加速...恢复到高速运行，代价比较大
•但无锁情况下，因为线程要保持运行，需要额外 CPU 的支持，CPU 在这里就好比高速跑道，没有额外的跑道，线程想高速运行也无从谈起，虽然不会进入阻塞，但由于没有分到时间片，仍然会进入可运行状
态，还是会导致上下文切换。

场景使用不同

synchronized适用于并发比较高的情况，cas常用在并发比较低的情况下（因为自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。）

原子性范围不同

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

锁的不同

CAS是基于乐观锁实现

synchronized是基于悲观锁实现

CAS原子操作类

符合原子性，可见性，有序性

原子类： AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference。

原子数组： AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray。

原子属性更新器：

AtomicLongFieldUpdater，AtomicIntegerFieldUpdater， AtomicReferenceFieldUpdater

```
//原子操作
AtomicInteger atomicInteger=new AtomicInteger();
//自增操作++i
atomicInteger.incrementAndGet();
```

解决 ABA 问题的原子类：

AtomicMarkableReference，通过引入一个 boolean 类型值 来反映中间有没有变过。

AtomicStampedReference，通过引入一个 int 类型值来累加来反映中间有没有变过。

享元模式

解决重复对象的内存浪费的问题

享元模式可以解决当系统中有大量相似对象需要缓冲池时候，不需要总是创建新对象，可以从缓冲池里拿，这样可以降低内存，提高效率

```
class Pool {
    // 1. 连接池大小
    private final int poolSize;
    // 2. 连接对象数组
    private Connection[] connections;
    // 3. 连接状态数组 0 表示空闲， 1 表示繁忙
    private AtomicIntegerArray states;  //共享要读写的变量
    // 4. 构造方法初始化
    public Pool(int poolSize) {
        this.poolSize = poolSize;
        this.connections = new Connection[poolSize];
        this.states = new AtomicIntegerArray(new int[poolSize]);
        for (int i = 0; i < poolSize; i++) {
            connections[i] = new MockConnection("连接" + (i+1));
        }
    }
    // 5. 借连接是考虑并发
    public Connection borrow() {
        while(true) {
            for (int i = 0; i < poolSize; i++) {
// 获取空闲连接
                if(states.get(i) == 0) {
                    if (states.compareAndSet(i, 0, 1)) {
                        log.debug("borrow {}", connections[i]);
                        return connections[i];
                    }
                }
            }
            // 如果没有空闲连接，当前线程进入等待
            synchronized (this) {
                try {
                    log.debug("wait...");
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    // 6. 归还连接
    public void free(Connection conn) {
        for (int i = 0; i < poolSize; i++) {
            if (connections[i] == conn) {
                states.set(i, 0);
                synchronized (this) {
                    log.debug("free {}", conn);
                    this.notifyAll();
                }
```

```
                break;
            }
        }
    }
}

class MockConnection implements Connection {
// 实现略
}
/**
*没有考虑连接动态增缩
*没有考虑可用性检测
*没有考虑超时处理
/
```

final类

在初始化完final字段之后，会插入一个写屏障（StoreStore）。

- 保证写屏障之前的代码不会指令重排序到写屏障之后。
- 保证写屏障之前的变量修改之后，都会同步到主存中。

自定义线程池

作用

1，降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗
2，提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行
3，提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控

代码细看

```
import lombok.extern.slf4j.Slf4j;

import java.util.ArrayDeque;
import java.util.Deque;
import java.util.HashSet;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

@Slf4j
class ThreadPool
{
    //任务型阻塞队列
    private BlockQueue<Runnable> taskQueue;

    //线程集合
    private HashSet<Worker> workers = new HashSet<>();

    //核心线程数
    private int coreSize;

    //设置任务的超时时间
```

```
    private long timeout;
    private TimeUnit timeUnit;

    //拒绝策略
    private RejectPolicy rejectPolicy;

    public ThreadPool(int coreSize, long timeout, TimeUnit timeUnit,int
queueCapcity,RejectPolicy<Runnable> rejectPolicy) {
        this.coreSize = coreSize;
        this.timeout = timeout;
        this.timeUnit = timeUnit;
        this.taskQueue = new BlockQueue<>(queueCapcity);
        this.rejectPolicy = rejectPolicy;
    }
    //执行任务
    public void  execute(Runnable task)
    {
        //多线程执行任务
        synchronized (workers)
        {
            //当任务数没有超过coreSize时候，直接交给worker对象执行
            //如果任务数超过coreSize,放入任务阻塞队列暂存
            if (workers.size() < coreSize)
            {
                //创建work添加到线程集合
                Worker worker = new Worker(task);
                log.info("新增worker{}，{}",worker,task);
                workers.add(worker);

                //执行work
                worker.start();
            }else
            {
                //加入阻塞队列
                log.info("加入任务队列{}",task);
                taskQueue.put(task);
                //1） 死等
                //2） 带超时等待
                //3） 让调用者放弃任务执行
                //4） 让调用者抛出异常
                //5） 让调用者自己执行任务
                taskQueue.tryPut(rejectPolicy,task);
            }
        }
    }

    class Worker extends Thread{
        private Runnable task;
        public Worker(Runnable task) {
            this.task = task;
        }


        @Override
        public void run() {
            //执行任务

            //1）当task不为空,执行任务
```

```
            //2）当task执行完毕，              //阻塞队列获取的到就继续执行,死
等,taskQueue.poll()这个是超时获取
            while (task!=null || (task =taskQueue.take()) !=null)
            {
                try {
                    log.info("正在执行....{}",task);
                    task.run();
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    task = null;
                }
            }

            //锁住集合
            synchronized (workers)
            {
                log.info("worker被移除{}",this);
                //工作结束移除线程
                workers.remove(this);
            }
        }

    }
}

@FunctionalInterface//拒绝策略
interface RejectPolicy<T>
{
    void  reject(BlockQueue<T> queue,T task);

}
//阻塞队列
@Slf4j
class BlockQueue<T>{
    public BlockQueue(int capcity) {
        this.capcity = capcity;
    }

    //任务队列，先进先出
    private Deque<T> queue = new ArrayDeque<>();

    //锁
    private ReentrantLock lock = new ReentrantLock();

    //生产者条件变量
    private Condition fullWaitSet = lock.newCondition();

    //消费者条件变量
    private  Condition emptyWaitSet = lock.newCondition();

    //容量
    private int capcity;

    //带超时的阻塞获取
    public T poll(long timeout, TimeUnit unit)
    {
        lock.lock();
```

```
        try {
            //转换成纳秒
            long nanos = unit.toNanos(timeout);

            while (queue.isEmpty())
            {
                //剩余时间小于等于0表示等很久了
                if (nanos<=0)
                {
                    //返回null
                    return null;
                }
                try {
                    //获取不到阻塞元素就消费者条件变量睡觉，带有一定时间
                    //返回的是剩余时间
                    nanos = emptyWaitSet.awaitNanos(nanos);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            //获取排最前面的阻塞队列元素
            T t = queue.removeFirst();
            //阻塞队列空出来了,唤醒生产者条件变量睡觉
            fullWaitSet.signal();
            return t;
        }finally {
            lock.unlock();
        }
    }

    //阻塞获取
    public T take()
    {
        lock.lock();
        try {
            while (queue.isEmpty())
            {
                try {
                    //获取不到阻塞元素就消费者条件变量睡觉
                    emptyWaitSet.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            //获取排最前面的阻塞队列元素
            T t = queue.removeFirst();
            //阻塞队列空出来了,唤醒生产者条件变量睡觉
            fullWaitSet.signal();
            return t;
        }finally {
            lock.unlock();
        }
    }

    //阻塞添加
    public void  put(T element)
    {
        lock.lock();
```

```
        try {
            //容量满了
            while (queue.size()==capcity)
            {
                try {
                    log.info("等待加入任务队列{}...",element);
                    //容器满了就生产者条件变量睡觉
                    fullWaitSet.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            log.info("加入任务队列{}",element);
            //容量不满了，添加到阻塞队列元素
            queue.addLast(element);
            //唤醒消费者条件变量
            emptyWaitSet.signal();
        }finally {
            lock.unlock();
        }

    }

    //带超时时间阻塞添加
    public boolean  offer(T element,long timeout,TimeUnit timeUnit)
    {
        lock.lock();
        try {
            long nanos = timeUnit.toNanos(timeout);
            //容量满了
            while (queue.size()==capcity)
            {
                try {
                   log.info("等待加入任务队列{}...",element);
                   if (nanos <= 0)
                       return false;

                   nanos = fullWaitSet.awaitNanos(nanos);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            log.info("加入任务队列{}",element);
            //容量不满了，添加到阻塞队列元素
            queue.addLast(element);
            //唤醒消费者条件变量
            emptyWaitSet.signal();
            return true;
        }finally {
            lock.unlock();
        }
    }

    public void tryPut(RejectPolicy rejectPolicy, T task) {
        lock.lock();
        try {
            //容量满了
            if (queue.size() == capcity)
```

```
            {
                //抽象拒绝策略
                rejectPolicy.reject(this,task);
            }else
            {
                //有空闲
                log.info("加入任务队列{}",task);
                queue.addLast(task);
                emptyWaitSet.signal();
            }
        }finally {
            lock.unlock();
        }
    }

    //获取大小
    public int size()
    {
        lock.lock();
        try {
            return queue.size();
        } finally {
            lock.unlock();
        }
    }

}
@Slf4j
public class test {
    public static void main(String[] args){
        //创建线程池
        ThreadPool threadPool = new ThreadPool(2, 1000, TimeUnit.MILLISECONDS,
5,((queue, task) ->{
            //自己写拒绝方法
            //死等
            //queue.put(task);
            //带超时等待
            queue.offer(task,500,TimeUnit.MILLISECONDS);
            //直接放弃
            log.info("放弃{}",task);
            //直接抛出异常
            throw new RuntimeException("任务执行失败");
            //自己执行任务

        } ));
        for (int i=0;i<10;i++)
        {
            int j=i;
            threadPool.execute(() ->{
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                log.info("{}",j);
            });
        }
    }
```

```
}
```

threadPoolExecutor

状态和线程数量存储

1) 线程池状态

ThreadPoolExecutor 使用 int 的高 3 位来表示线程池状态，低 29 位表示线程数量

状态名	高 3 位	接收新任务	处理阻塞队列任务	说明
RUNNING	111	Y	Y	
SHUTDOWN	000	N	Y	不会接收新任务，但会处理阻塞队列剩余任务
STOP	001	N	N	会中断正在执行的任务，并抛弃阻塞队列任务
TIDYING	010	-	-	任务全执行完毕，活动线程为 0 即将进入终结
TERMINATED	011	-	-	终结状态

从数字上比较，TERMINATED > TIDYING > STOP > SHUTDOWN > RUNNING

这些信息存储在一个原子变量 ctl 中，目的是将线程池状态与线程个数合二为一，这样就可以用一次 cas 原子操作进行赋值

```
// c 为旧值， ctlOf 返回结果为新值
ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c)));

// rs 为高 3 位代表线程池状态， wc 为低 29 位代表线程个数，ctl 是合并它们
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

核心参数

```
public ThreadPoolExecutor(int corePoolSize,//核心线程数
                          int maximumPoolSize,//最大线程数
                          long keepAliveTime,//线程空闲时间
                          TimeUnit unit,//时间单位
                          BlockingQueue<Runnable> workQueue,//任务队列
                          ThreadFactory threadFactory,//线程工厂
                          RejectedExecutionHandler handler//拒绝策略)
{
    ...
}
```

线程池的执行顺序

线程池按以下行为执行任务

- 当线程数小于核心线程数时，创建线程。
- 当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。
- 当线程数大于等于核心线程数，且任务队列已满，若线程数小于最大线程数，创建线程。
- 若线程数等于最大线程数，则执行拒绝策略

threadFactory

线程工厂，用来创建线程。

handler

拒绝策略，默认是AbortPolicy，会抛出异常。
当线程数已经达到maxPoolSize，且队列已满，会拒绝新任务。
当线程池被调用shutdown()后，会等待线程池里的任务执行完毕再shutdown。如果在调用shutdown()和线程池真正shutdown之间提交任务，会拒绝新任务。
AbortPolicy 丢弃任务，抛运行时异常。
CallerRunsPolicy 由当前调用的任务线程执行任务。
DiscardPolicy 忽视，什么都不会发生。
DiscardOldestPolicy 从队列中踢出最先进入队列（最后一个执行）的任务。

方法

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
        10,
        100,
        10,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<>(100),
        new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
                return null;
            }
        },
        new ThreadPoolExecutor.DiscardOldestPolicy()
);

pool.execute(() ->{
    //要执行的方法
});
//任务带有返回值
Callable<a> task=new Callable<a>() {
    @Override
    public a call() throws Exception {
        return null;
    }
};
//执行带有返回值的方法
Future<a> submit = pool.submit(task);
try {
    //获取返回值
    a a = submit.get();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
// 提交 tasks 中所有任务
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws
InterruptedException;

// 提交 tasks 中所有任务，带超时时间
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long
timeout, TimeUnit unit) throws InterruptedException;
```

饥饿

即没有多余的线程去运行。例如只有a工作型线程，想要b工作型线程去接收a的，但是根本就创建不了。所以需要不同的工作线程时候就用不同的线程池

创建多少线程池合适

CPU 密集型运算
通常采用cpu 校数 + 1能够实现最优的 CPU 利用率， +1是保证当线程由于页缺失故障（操作系统）或其它
原因导致暂停时，额外的这个线程就能顶上去，保证CPU 时钟周期不被浪费
IO 密集型运算
CPU 不总是处于繁忙状态，例如，当你执行业务计算时，这时候会使用 CPU 资源，但当你执行 10 操作时、远程 RPC 调用时，包括进行数据库操作时，这时候 CPU 就闲下来了，你可以利用多线程提高它的利用
率.
经验公式如下
线裡数二校数，m型 CpU 利用常 * 总时间(CPU计-算时间中等待时间〉/CPU 计集时间
例如4核CPU 计算时间是 50%，其它等待时间是 50%，期望cpu被100%利用，套用公式
4 * 100% * 100% / 50% = 8
例如4核CPU 计算时间是 10%，其它等待时间是 90%，期望cpu 被100%利用，套用公式
4 * 100% * 100% / 10% = 40

结构

Runnable和Callable

1、Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；而Runnable接口实现类中run()方法的异常必须在内部处理掉

2、Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。Callable接口实现类中run()方法允许将异常向上抛出

Future

*Future*就是对于具体的*Runnable*或者*Callable*任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过*get*方法获取执行结果，该方法会阻塞直到任务返回结果。
Future类位于java.util.concurrent包下，它是一个接口：

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

cancel()方法用来取消任务，如果取消任务成功则返回true，如果取消任务失败则返回false。参数mayInterruptIfRunning表示是否允许取消正在执行却没有执行完毕的任务，如果设置true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论mayInterruptIfRunning为true还是false，此方法肯定返回false，即如果取消已经完成的任务会返回false；如果任务正在执行，若

mayInterruptIfRunning设置为true，则返回true，若mayInterruptIfRunning设置为false，则返回false；如果任务还没有执行，则无论mayInterruptIfRunning为true还是false，肯定返回true。

isCancelled方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。

isDone方法表示任务是否已经完成，若任务完成，则返回true；

get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；

get(long timeout, TimeUnit unit)用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null。

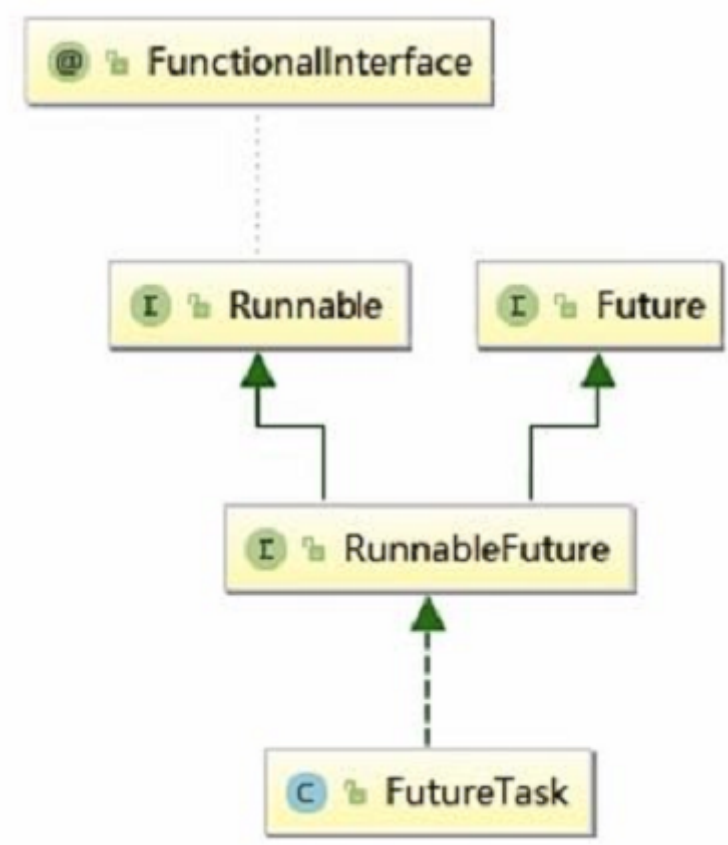
缺点

get容易导致阻塞，调用就一定要获取结果

isDone浪费无谓的CPU资源

没有灵活的线程配合，即线程一个接着一个运行或者选一个最快的线程。

FutureTask



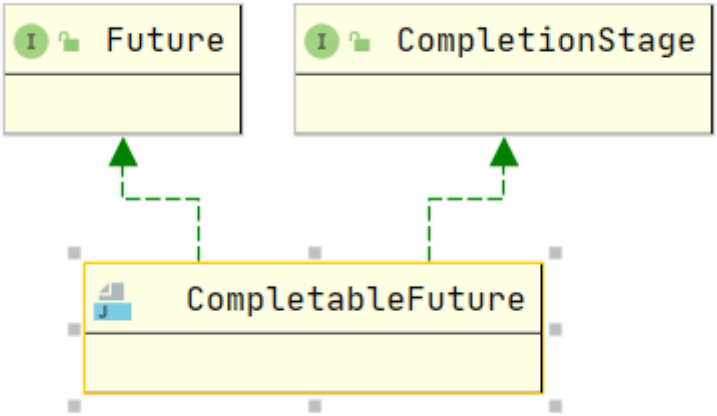

```
public FutureTask(Callable<V> callable) {
    if (callable == null)
      throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;    // ensure visibility of callable

}

public FutureTask(Runnable runnable, V result) {
        this.callable = Executors.callable(runnable, result);
        this.state = NEW;       // ensure visibility of callable
```

```
}
```

FutureTask构造函数传入Runnable实现类，使用FutureTask.get()得到传入参数result的值，该值可以用来判断线程是否正常执行等，无法通过它来得到线程执行想要返回的结果。

既然传入的Runnable，为什么还能得到result值喃，因为在Executors.callable(runnable, result)里面使用RunnableAdapter将Runnable封装成了Callable,result值传给了RunnableAdapter.result，然后线程执行call()方法后，原封不动的返回RunnableAdapter.result。

Futrue有缺陷推出CompleteFuture




```
public class CompletableFuture<T> implements Future<T>, CompletionStage<T>
```

不推荐new空参

优点

异步任务结束时，会自动回调某个对象的方法

异步任务出错时，会自动回调某个对象的方法

方法

runAsync 无返回值

supplyAsync 有返回值

```
runAsync(Runnable runnable,    Executor executor)
//不指明线程池则用自带的ForkJoinPool
```

减少阻塞和轮询

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -
> {
    //int i=1/0;
    return "hello";
}).whenComplete((v,e)->{
    //v是上一步的结果，e是上一步的异常,没有异常就为null
    if (e==null)
        System.out.println(v);
}).exceptionally(e ->{
    //处理异常
    System.out.println("异常情况"+e.getMessage());
    return null;
});
```

串行化执行

```
CompletableFuture<Void> completableFuture = CompletableFuture.supplyAsync(() ->
{
    return "hello";
}).thenApply(r ->{
    //获取上一步的值下一步处理
    return r;
}).thenAccept(r ->{
    //处理完没有返回值了
    System.out.println(r);
});

completableFuture.thenRun(()->{
           //执行完上一步执行下一步，并且不需要上一步结果
        });
```

thenRun和thenRunAsync

thenRun是跟随前一个线程池，thenRunAsync是自定义线程池，没有线程池就用默认

谁快用谁

```
CompletableFuture<String> a = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "a";
});
CompletableFuture<String> b = CompletableFuture.supplyAsync(() -> {
    return "b";
});
CompletableFuture<String> completableFuture = a.applyToEither(b, r -> {
    return r + " is winner";
});
System.out.println(completableFuture.join());
```

结果合并

```
CompletableFuture<String> a = CompletableFuture.supplyAsync(() -> {
    System.out.println(Thread.currentThread().getName());
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "a";
});
CompletableFuture<String> b = CompletableFuture.supplyAsync(() -> {
    System.out.println(Thread.currentThread().getName());
    return "b";
});
//线程合并
CompletableFuture<String> result = a.thenCombine(b, (x, y) -> {
    System.out.println(Thread.currentThread().getName());
    return x + y;
});
System.out.println(result.join());


ForkJoinPool.commonPool-worker-9
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-9
ab
```

获取值

```
//没有获取到值有一个备胎
System.out.println(completableFuture.getNow("xxx"));
//还在运行中会打断后给值
System.out.println(completableFuture.complete("interrupt")+"*
*"+completableFuture.join());

结果为
xxx
true* *interrupt
```

函数式接口名称

函数式接口名称	方法名称	参数	返回值
Runnable	run	无参数	无返回值
Function	apply	1	有
Consumer	accept	1	无
Supplier	get	无	有
BigConsumer	accept	2	无

```
@Accessors(chain = true)
则可以set方法进行函数式编程
```

join和get

是一样的，一个编码检查异常一个不检查

函数式编程

面向对象思想关注用什么对象完成什么事情，但是函数式编程思想类似于我们数学中的函数，他主要关心的是对数据进行了什么操作。

优点

代码简洁，开发快速

接近自然语言，易于理解

易于并发编程

匿名内部类

如果实现类Interface01Impl全程只使用一次，那么为了这一次的使用去创建一个类，未免太过麻烦。我们需要一个方式来帮助我们摆脱这个困境。匿名内部类则可以很好的解决这个问题。

```
interface a
{
    int test(int i);
}
public class ThreadPoolTest
{
    public static void main(String[] args)
    {
        new a() {
            @Override
            public int test(int i) {
                return 0;
            }
        };
    }
}
```

https://blog.csdn.net/a850661962/article/details/109642780?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216667662381678239056023 5%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=166676623816782390560235&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-109642780-null-null.142^v59^pc_rank_34_1,201^v3^control_1,213^v1^t3_control1&utm_term=%E5%8C%BF%E5%90%8D%E5%86%85%E9%83%A8%E7%B1%BB&spm=1018.2226.3001.4187

lambda

ambda 表达式的形式为 () -> {}; () 里面是函数式接口中抽象方法的形参， {} 中是抽象方法的实现。

可以理解为自己用的时候再去写一段代码

省略

（匿名内部类）实现类中只传入一个参数所以小括号可以取消，方法体只有一条语句，大括号可以取消

由于在接口的抽象方法中，已经定义了参数的数量类型 所以在Lambda表达式中参数的类型可以省略
备注：如果需要省略类型，则每一个参数的类型都要省略，千万不要一个省略一个不省略

如果方法体中唯一的一条语句是一个返回语句，则省略大括号的同时 也必须省略return

```
LambdaSingleReturnNoneParmeter lambda4=()->{
    return 10;
};
//可以精简为:
LambdaSingleReturnNoneParmeter lambda4=()->10;
```

例子

```
//先写接口
interface print
{
    void test(String val);
}
class lambda
{
    print print;
    lambda(print print){
        this.print=print;
    }
    //方法,是抽象的操作，让你自定义对参数的操作
    public void printSomething(String val)
    {
        //方法传入一段代码
        print.test(val);
    }
}

public class ThreadPoolTest
{
    public static void main(String[] args)
    {
        //初始化时候要抽象的操作具体化
        lambda lambda = new lambda((s)->{
            System.out.println(s+s);
        });
        lambda.printSomething("1");
    }
}
```

方法引用(普通方法与静态方法)

语法:

方法引用: 可以快速的将一个Lambda表达式的实现指向一个已经实现的方法

方法的隶属者如果是静态方法，隶属的就是一个类，其他的话就是隶属对象

语法：方法的隶属者::方法名

注意:

引用的方法中，参数数量和类型一定要和接口中定义的方法一致
返回值的类型也一定要和接口中的方法一致

```
//类Syntax3自定义的实现方法
    private static int change(int a){
        return a*2;
    }
//简化
LambdaSingleReturnSingleParmeter lambda4=a->change(a);
//方法引用       LambdaSingleReturnSingleParmeter lambda5=Syntax3::change;
```

方法引用(构造方法)

```
public class Person {
    public String name;
    public int age;

    public Person() {
        System.out.println("Person的无参构造方法执行");
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Person的有参构造方法执行");
    }
}



interface PersonCreater{
    //通过Person的无参构造实现
    Person getPerson();
}

interface PersonCreater2{
    //通过Person的有参构造实现
    Person getPerson(String name,int age);
}


PersonCreater creater=()->new Person();
        //引用的是Person的无参构造
         //PersonCreater接口的方法指向的是Person的方法
```

```
        PersonCreater creater1=Person::new; //等价于上面的()->new Person()
        //实际调用的是Person的无参构造 相当于把接口里的getPerson()重写成new Person()。
        Person a=creater1.getPerson();
        //引用的是Person的有参构造
        PersonCreater2 creater2=Person::new;
        Person b=creater2.getPerson("张三",18);
```

接口实现类

用接口方式创建为

- Person p1=new student();

注： 用接口方式创建的类的对象，只能用**对象名.方法名** 访问接口中定义的方法，不能访问类中自己定义的属性和方法以及接口中的属性。

stream

非关系型数据库获取数据处理数据

关系型数据库可用sql语句

stream和collection（都是接口）

collection关注数据的存储，与内存打交道，而stream是有关数据计算的，与cpu打交道

注意

stream

不会存储元素

不会改变源对象，返回的是一个新stream

操作是有延迟的即一旦执行终止操作，才会执行中间操作链

```
        //过滤操作
        temp.filter(i ->{
            return i.getAge()>12;
        }).forEach(System.out::println);
        //不可以再次操作
        temp.limit(1);

Exception in thread "main" java.lang.IllegalStateException: stream has already
been operated upon or closed
```

获取stream

```
//通过集合获取stream
ArrayList<String> strings = new ArrayList<>();
//通过数组获取stream
int a[] = {1,2,23,3,3,};
IntStream stream1 = Arrays.stream(a);
//通过stream的of获取
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

stream的类型

```
ArrayList<String> strings = new ArrayList<>();
//顺序流
Stream<String> stream = strings.stream();
//并行流
Stream<String> stringStream = strings.parallelStream();
```

过滤

```
person person0 = new person(11,"11");
person person1 = new person(12,"12");
person person2 = new person(13,"13");
person person3 = new person(14,"14");

Stream<person> temp = Stream.of(person0, person1, person2, person3);
//过滤操作
temp.filter(i ->{
    return i.getAge()>12;
}).forEach(System.out::println);
```

限制

```
//要前i个
temp.limit(i);
```

跳过

```
//跳过i个
temp.skip(i).forEach(System.out::println);
```

筛选

```
根据hashcode和equals去重复
//筛选重复        temp.distinct().forEach(System.out::println);
```

映射

```
temp.map(i ->{
    //对数据进行处理
    i.setAge(i.getAge()+1);
    return i;
}).forEach(System.out::println);
```

```
//接口Function的R apply(T t)获取的是person类的年龄代码
temp.map(person::getAge).forEach(System.out::println);

11
12
13
14
```

```
//如果是一个集合里面套着另一个集合,flatMap会去拆大集合里的元素，如果元素是集合，会接着拆
temp.flatMap();
```

排序

```
//自然排序，对数字排序，不是数字不用Comparable会报错
temp.sorted().forEach(System.out::println);
//Exception in thread "main" java.lang.ClassCastException: person cannot be cast
to java.lang.Comparable
```

lambda实现 Comparator

```
temp.sorted((o1, o2) -> o1.getAge()-o2.getAge()).forEach(System.out::println);

 //o1的age大于o2则为正数，即为升序
 o1.getAge()-o2.getAge();
 两个对象比较的结果有三种：大于，等于，小于。
如果要按照升序排序，
则o1 小于o2，返回（负数），相等返回0，01大于02返回（正数）
如果要按照降序排序
则o1 小于o2，返回（正数），相等返回0，01大于02返回（负数）
```

也可以实现如果年龄相等就再比较其他

```
(o1, o2) -> {
    int i=o1.getAge()-o2.getAge();
    if (i==0)
    {
        //接着比较
        return 0;
    }else
        return i;
}
```

终止操作！！！

操作是有延迟的即一旦执行终止操作，才会执行中间操作链，所以一定要终止操作

forEach

```
temp.forEach(i->{
    System.out.println(i);
});
```

allMatch

```
//终止判断所有，返回boolean
boolean b = temp.limit(3).allMatch(e -> {
    return e.getAge() > 18;
});
```

anyMatch

//anyMatch，只要有符合条件的就返回*true*,默认为*false*

noneMatch

//noneMatch，必须全部不匹配才返回*true*，默认为*true*

findFirst

*//findFirst*查找*list*第一个元素,相当于*list.get(0)* Employee employeeFindFirst = list.stream().findFirst().get();

findAny

Employee employeeFindAny = list.stream().findAny().get();

count

*//*计算*list*的*count*，相当于*list.size()* long count = list.stream().count();

max

*//*计算*list*中相关值的最大值 Employee max = list.stream().max((a, b) -> a.getAge() - b.getAge()).get();

min

*//*计算*list*中相关值的最小值 Employee min = list.stream().min((a, b) -> a.getAge() - b.getAge()).get();

reduce

求总和

```
//映射年龄,求总和的方式
Integer reduce = temp.limit(3).map(person::getAge).reduce((e1, e2) -> e1 + e2);
```

```
//映射年龄,求总和的方式,有初始值
Integer reduce = temp.limit(3).map(person::getAge).reduce(10, (e1, e2) -> e1 +
e2);
```

collect

将流转换成其他形式

```
List<person> reduce = temp.limit(3).collect(Collectors.toList());

Set<person> collect = temp.limit(3).collect(Collectors.toSet());

还有其他Collectors.to方法
```

optional

是一个容器类，保存类型T的值，防止空指针异常