Multi Threading

Reference

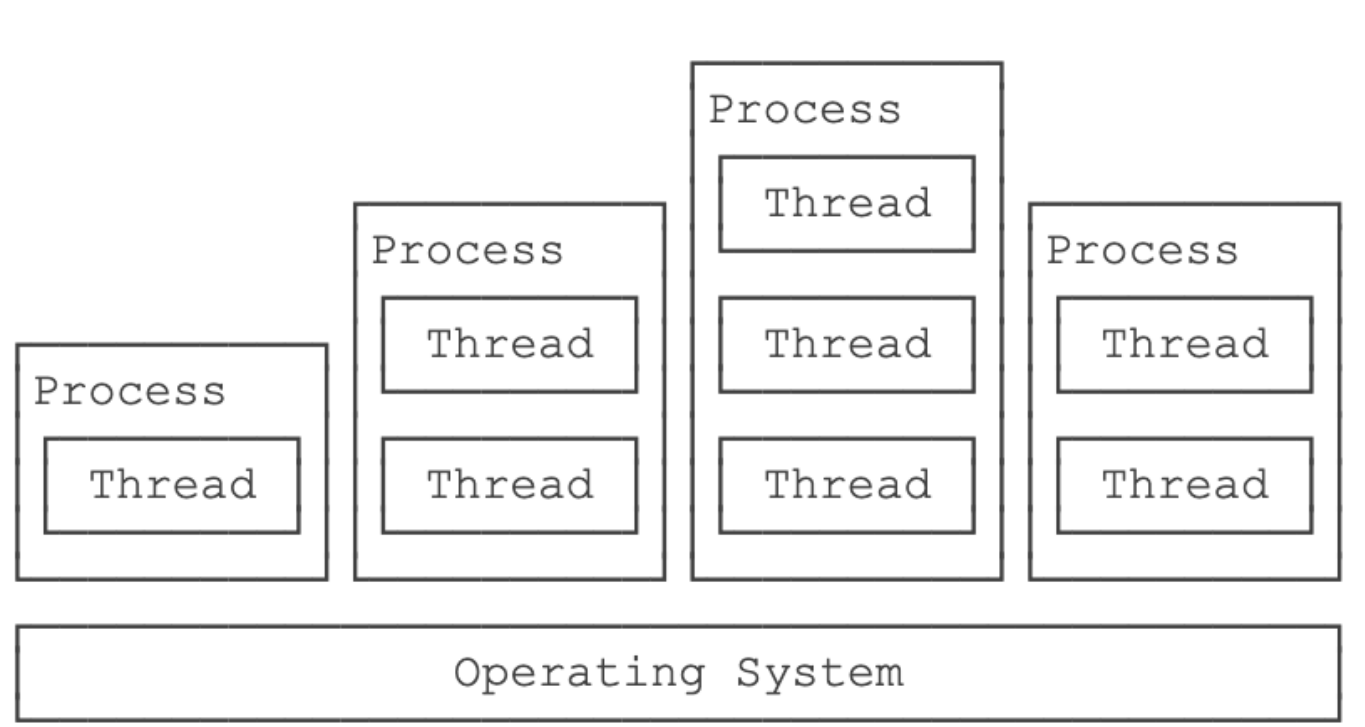
Tutorial: https://www.liaoxuefeng.com/wiki/1252599548343744/1304521607217185

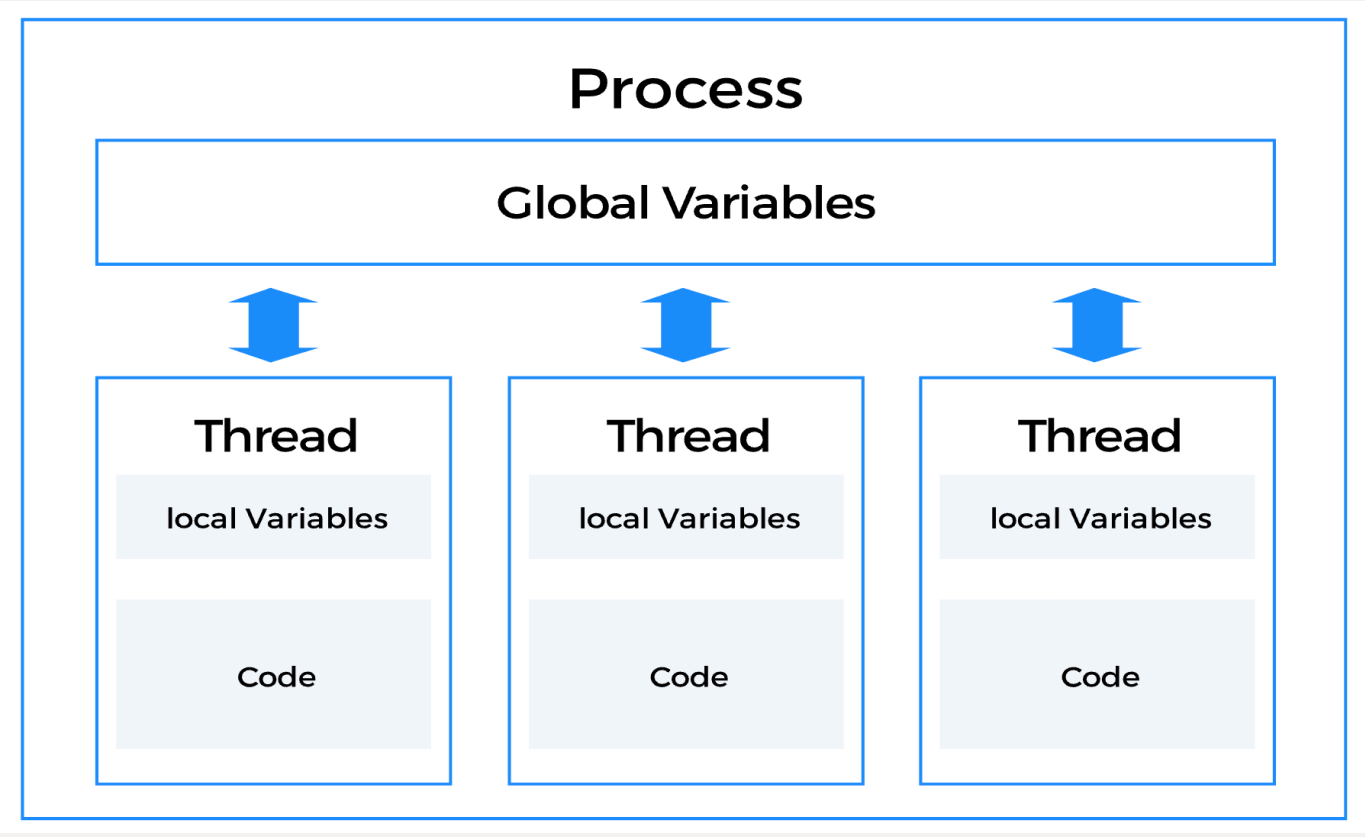
Repo: https://github.com/TAIsRich/chuwa-eij-tutorial.git

in pakcage: com.chuwa.tutorial.t08_multithreading

Questions: https://www.interviewbit.com/multithreading-interview-questions/#benefits-of-multithreading

What is Thread


Process
Thread
Process
Thread
Thread
Process
Thread
Thread
Thread
Process
Thread
Thread
Operating System


Process
Global Variables
Thread
local Variables
Code
Thread
local Variables
Code
Thread
local Variables
Code

Creation (Easy, 重要)

```
com.chuwa.tutorial.t08_multithreading.c01_creat
```

Extends Thread Class

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("start new thread using extends thread");
    }
}

Thread t = new MyThread();  // JVM没有创建thread
t.start(); // 此时JVM才创建新的thread
```

Implements Runnable

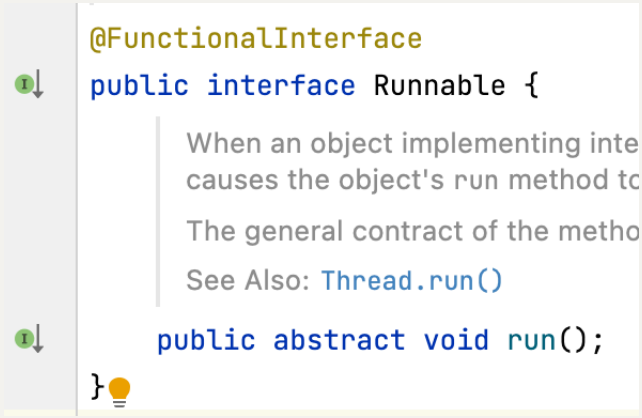
```
public class MyRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("Start new thread using Runnable");
    }
}

Thread t2 = new Thread(new MyRunnable());
```

Implements Callable

```java
public class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        Thread.sleep(5000);
        return "Start new thread using Callable";
    }
}
```

Questions

- What is the difference between t.start() and t.run()?
    - t.start starts a new thread to excute the task（run()）
    - t.run() excute the task in the current thread.
- Wht is the differecence between Callable and Runnbale?
    - runnable has no return;
    - callable has return;
- Can we use new Thread(lambda)? is it equal to implement Runnable? Why?
    - yes, if there is only one interface with @FunctionalInterface, we can use lambda;
    - in Runnable, only have one interface run();
    -

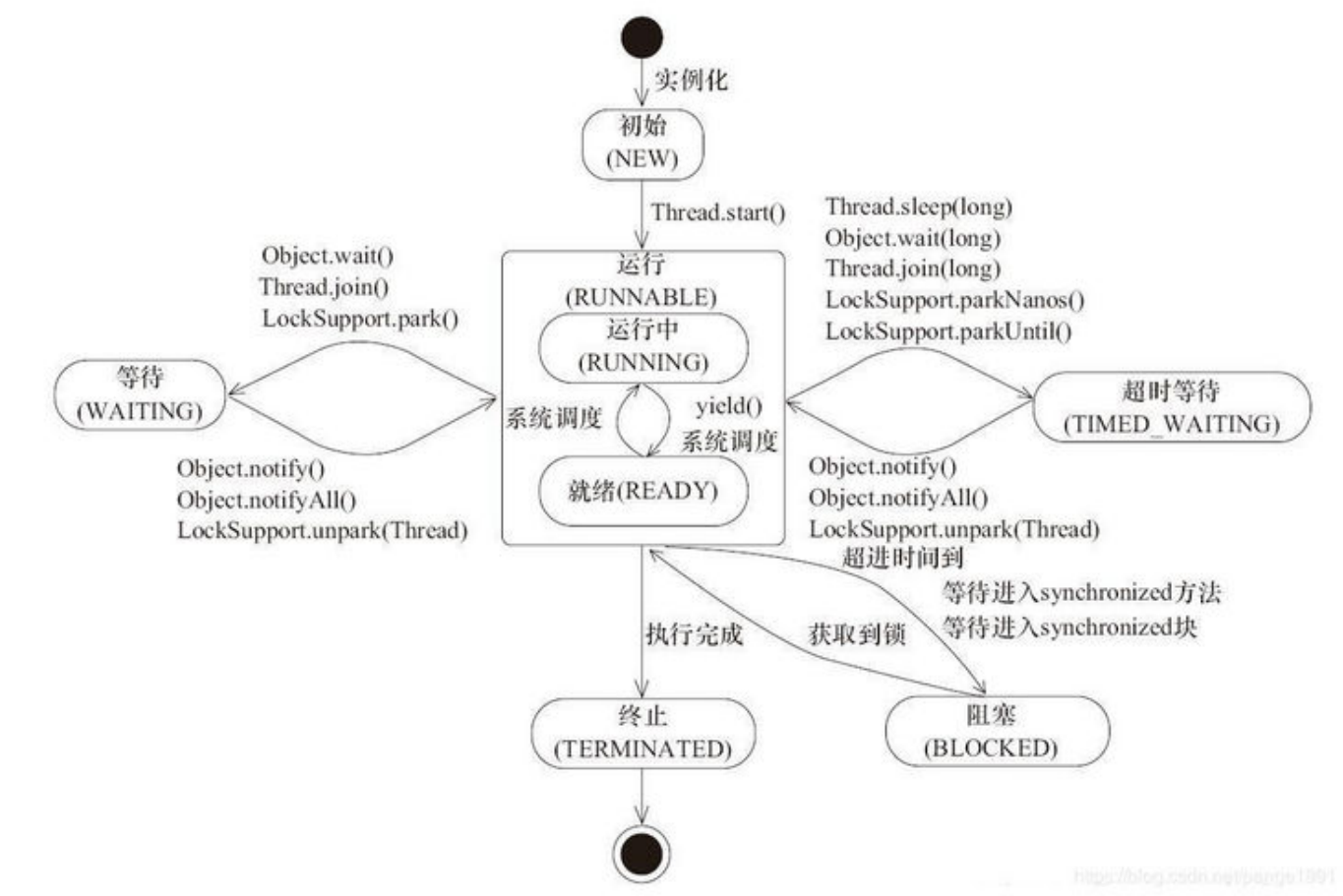


- When the thread terminated?
    - if there is no infinite loop, it will terminated when task is done.

- Which one is popular between Runnable and Callable? Why?

Thread Status （Hard, 知道概念）

- **初始(NEW)**：新创建了一个线程对象，但还没有调用start()方法。
- **运行(RUNNABLE)**：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态（running）。
- **阻塞(BLOCKED)**：表示线程阻塞于锁。
- **等待(WAITING)**：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
- **超时等待(TIMED_WAITING)**：该状态不同于WAITING，它可以在指定的时间后自行返回。
- **终止(TERMINATED)**：表示该线程已经执行完毕。



https://segmentfault.com/a/1190000038392244#:~:text=Thread%E7%BA%BF%E7%A8%8B%E7%8A%B6%E6%80%81%E7%9A%84%E5%88%92%E5%88%86%3A&text=%E5%88%9D%E5%A7%8B(NEW)%EF%BC%9A%E6%96%B0%E5%88%9B%E5%BB%BA,%E5%AF%B9%E8%B1%A1%E7%9A%84start()%E6%96%B9%E6%B3%95%E3%80%82

Interrupt, Daemon, Join

- `t.join()`

```java
public class JoinTest {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            ...
              Thread.sleep();
        });

        System.out.println("Main start");
```

```
 9          t.start();
10          Thread.sleep();
11          try {
12              System.out.println("Main thread is
    stopped and waiting for t thread end");
13              t.join();   // main thread wait for t
    thread end. main thread's status is Timed Waiting.
14          } catch (InterruptedException e) {
15              e.printStackTrace();
16          }
17          System.out.println("Main stop");
18      }
19  }
20
```

- Main thread call t.join()
- Main thread will stop and wait for thread t complets its task
- Main thread will continue after t finished its task

- `t.setDaemon(true)`

```
1  Thread t = new MyThread();
2  t.setDaemon(true);
3  t.start();
```

- A daemon thread runs in the background but **as soon as the main application thread exits, all daemon threads are killed by the JVM**.
- Note that in case a spawned thread isn't marked as a daemon then **even if the main thread finishes execution, JVM will wait for the spawned thread to finish before tearing down the process**.

- `Thread.sleep(1000)`
  - the thread which call Thread.sleep(1000) would sleep
- `t.interrupt()`

- 中断thread t
- can interrupt sleep()
- can not interrupt while(true)
- can interrupt !isInterrupted()

Synchronized (重要，easy)

> *解决多线程竞争的问题 - 线程不安全*

锁一个Java 对象

任何对象都可以当做lock

```
synchronized(lock) {
    n = n + 1;
}
```

In singleton, we need to `syncronized(this)`

```
public class Singleton {

    /**
     * volatile 使得修改值立即更新到主程序
     * <p>
     * 双重检测锁模式看上去完美无缺，其实是存在问题，在多线程的情况下，可能会出现空指针问题，出现问
     * 题的原因是JVM在实例化对象的时候会进行优化和指令重排序操作。
     * 要解决双重检查锁模式带来空指针异常的问题，只需要使用 volatile 关键字，volatile 关
     * 键字可以保证可见性和有序性。
     */

    // 1, static volatile variable
    private static volatile Singleton instance;
```

```
14
15      // 2, make constructor be private
16      private Singleton() {
17      }
18
19      // 3. static synchronized getInstance method
20      public static Singleton getInstance() {
21
22          // 4, make sure thread safe
23          if (instance == null) { // performance
24            // t2, t1, t3,
25              synchronized (this) {
26                  if (instance == null) {
27                      instance = new Singleton();
28                  }
29              }
30          }
31
32          return instance;
33      }
34  }
```

Question

- If we need two locks, what can we do?
    - use 2 different objects, each object is a lock.

```
1  class Counter {
2      public static final Object LOCK_STUDENT = new
   User();
3      public static final Object LOCK_TEACHER = new
   Object();
4  }
```

- What is the difference between synchronized(this) and synchronized(Singleton.class)?

Synchronized method

```
public void add(int n) {
    synchronized(this) { // 锁住this
        count += n;
    } // 解锁
}
等价于
public synchronized void add(int n) { // 锁住this
    count += n;
} // 解锁
```

```
public synchronized static void test(int n) {
    ...
}
等价于
public static void test(int n) {
        synchronized(Counter.class) {
            ...
        }
}
```

DeadLock （重要，理解概念就行）

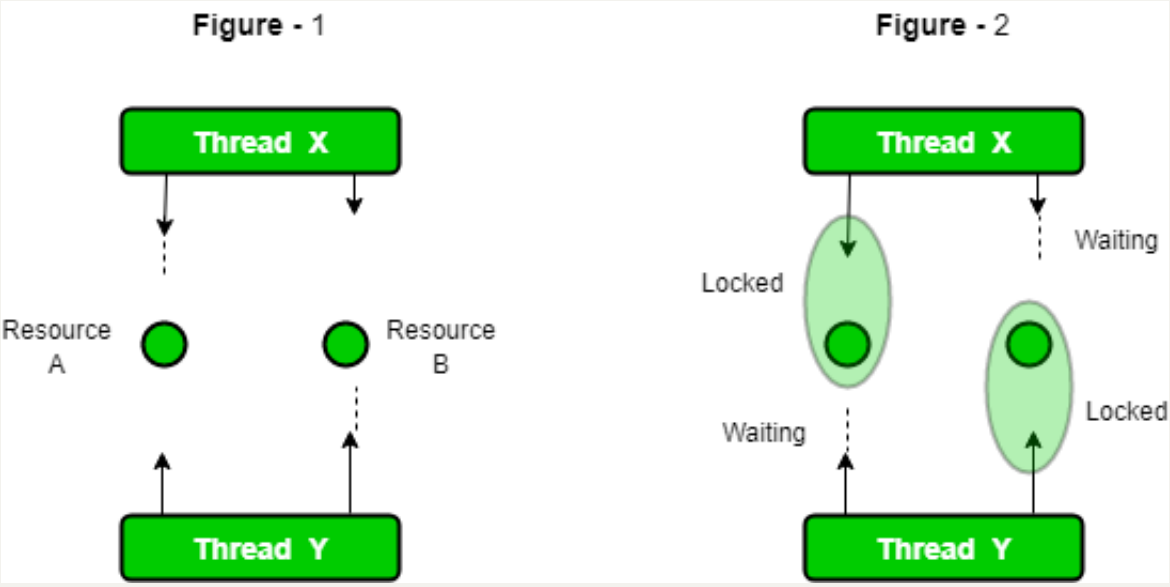
一个人吃饭需要同时用刀和叉。

刀被大熊拿走了，叉被小象拿走了。

大熊拿着刀等叉子

小象拿着叉等刀子

死锁了。



Wait & Notify （重要）

> *synchronized 解决了多线程竞争的问题*
> *但是`synchronized`并没有解决多线程协调的问题*
>
> *Wait Notify解决线程之间的协调问题*

线程1打印完数字后，把线程2 唤醒，然后线程1 wait()

线程2被唤醒后，打印数字，然后唤醒线程1，然后线程2 wait()

线程1和线程2之间则产生了沟通和协调

```
1  public class OddEventPrinter {
2      private static final Object monitor = new Object();
3      private static int value = 1; // 公共资源,
4
5      public static void main(String[] args) {
6          PrintRunnable runnable = new PrintRunnable();
7          new Thread(runnable).start(); // t0
8          new Thread(runnable).start(); // t1
```

```
    }

    static class PrintRunnable implements Runnable {
        @Override
        public void run() {
          // synchronized : 门
          // 门里有资源
          // 买一把锁 monitor
          // t0, t1
            synchronized (monitor) {
                while (value <= 10) {

 System.out.println(Thread.currentThread().getName() + ": " +
value++);
                    monitor.notifyAll(); // t0: monitor.notify()
-> 持有同一把锁的
                    try {
                        monitor.wait(); // 解锁 且进入Waiting
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
Thread-0: 1
Thread-1: 2
Thread-0: 3
Thread-1: 4
Thread-0: 5
Thread-1: 6
Thread-0: 7
Thread-1: 8
Thread-0: 9
Thread-1: 10
```

Locks （知道存在和各个锁的特性即可）

我们知道Java语言直接提供了`synchronized`关键字用于加锁，但这种锁一是很重，二是获取时必须一直等待，没有额外的尝试机制。

`java.util.concurrent.locks`包提供的`ReentrantLock`用于替代`synchronized`加锁

ReentrantLock

- `private final Lock lock = new ReentrantLock();`
- `lock.lock();`

```
try {
    count += n;
} finally {
    lock.unlock();
}
```

- 因为`synchronized`是Java语言层面提供的语法，所以我们不需要考虑异常，而`ReentrantLock`是Java代码实现的锁，我们就必须先获取锁，然后在`finally`中正确释放锁。

  顾名思义，`ReentrantLock`是可重入锁，它和`synchronized`一样，一个线程可以多次获取同一个锁

```
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int count;

    public void add(int n) {
        lock.lock();
        try {
            count += n;
        } finally {
            lock.unlock();
        }
```

```
    }
}
example 2:


static class PrintRunnable implements Runnable {
  private final Lock lock = new ReentrantLock();
  private final Condition condition = lock.newCondition();
        @Override
        public void run() {
          // synchronized : 门
          // 门里有资源
          // 买一把锁 monitor
          lock.lock();

         try   {
                while (value <= 10) {

 System.out.println(Thread.currentThread().getName() + ": " +
value++);
                    condition.signal();
                    try {
                        condition.await(); // 解锁
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            } finally {
              lock.unlock()
            }
         }
        }
    }


```

Questions

- What is the main difference between synchronized and ReentrantLock?

```
if (lock.tryLock(1, TimeUnit.SECONDS)) {
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

Condition

线程之间通信和协调：

syncronized: wait & notify & notifyAll

ReentrantLock: condition.await() & condition.signal() & condition.signalAll()

```
class TaskQueue {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private Queue<String> queue = new LinkedList<>();

    public void addTask(String s) {
        lock.lock();
        try {
            queue.add(s);
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public String getTask() {
```

```
        lock.lock();
        try {
            while (queue.isEmpty()) {
                condition.await();
            }
            return queue.remove();
        } finally {
            lock.unlock();
        }
    }
}
```

ReadWriteLock

- 只允许一个线程写入（其他线程既不能写入也不能读取）；
- 没有写入时，多个线程允许同时读（提高性能）。
- `ReadWriteLock`适合读多写少的场景。
- 悲观锁，因为读的过程中是不允许写的。

读锁和写锁是分开的。

```
public class Counter {
    private final ReadWriteLock rwlock = new
ReentrantReadWriteLock();
    private final Lock rlock = rwlock.readLock();
    private final Lock wlock = rwlock.writeLock();
    private int[] counts = new int[10];

    public void inc(int index) {
        wlock.lock(); // 加写锁
        try {
            counts[index] += 1;
        } finally {
            wlock.unlock(); // 释放写锁
```

```
        }
    }

    public int[] get() {
        rlock.lock(); // 加读锁
        try {
            return Arrays.copyOf(counts, counts.length);
        } finally {
            rlock.unlock(); // 释放读锁
        }
    }
}
```

StampedLock

- 乐观锁，读的过程中允许写。
    - 但可能造成数据不一致的问题，所以需要写代码handle。

获得一个乐观读锁 --> 检查乐观读锁后是否有其他写锁发生 --> 获取一个悲观读锁 --> 释放悲观读锁

1. `long stamp = stampedLock.tryOptimisticRead();``
2. `` `!stampedLock.validate(stamp) ``
3. stamp = stampedLock.readLock();
4. `stampedLock.unlockRead(stamp);`

```
public class Point {
    private final StampedLock stampedLock = new StampedLock();

    private double x;
    private double y;

    public void move(double deltaX, double deltaY) {
        long stamp = stampedLock.writeLock(); // 获取写锁
```

```
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            stampedLock.unlockWrite(stamp); // 释放写锁
        }
    }

    public double distanceFromOrigin() {
        long stamp = stampedLock.tryOptimisticRead(); // 获得一个乐观读锁
        // 注意下面两行代码不是原子操作
        // 假设x,y = (100,200)
        double currentX = x;
        // 此处已读取到x=100，但x,y可能被写线程修改为(300,400)
        double currentY = y;
        // 此处已读取到y，如果没有写入，读取是正确的(100,200)
        // 如果有写入，读取是错误的(100,400)
        if (!stampedLock.validate(stamp)) { // 检查乐观读锁后是否有其他写锁发生
            stamp = stampedLock.readLock(); // 获取一个悲观读锁
            try {
                currentX = x;
                currentY = y;
            } finally {
                stampedLock.unlockRead(stamp); // 释放悲观读锁
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }
}
```

Concurrent Collections (easy, 重要)

针对各种集合，我们日常接触的都是non-thread-safe的集合。

Java提供了对应的thread-safe的集合，方便我们直接使用。

INTERFACE	NON-THREAD-SAFE	THREAD-SAFE
List	ArrayList	CopyOnWriteArrayList
Map	HashMap	ConcurrentHashMap
Set	HashSet / TreeSet	CopyOnWriteArraySet
Queue	ArrayDeque / LinkedList	ArrayBlockingQueue / LinkedBlockingQueue
Deque	ArrayDeque / LinkedList	LinkedBlockingDeque

How to create a thread-safe collection

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

2.

```
1  Map unsafeMap = new HashMap();
2  Map threadSafeMap = Collections.synchronizedMap(unsafeMap);
```

Atomic （easy, 重要）

- 原子操作实现了无锁的线程安全；
- 适用于计数器，累加器等。
- CAS

https://tobebetterjavaer.com/thread/atomic.html#%E9%A2%84%E5%A4%87%E7%9F%A5%E8%AF%86-cas%E6%93%8D%E4%BD%9C

计数器example

Unthread safe 计数器

```
public class AtomicDemo {
    private static int counter = 1;

    public static void main(String[] args) {
        System.out.println(counter++/++counter);
        System.out.println(counter);
    }
}
输出结果:
1
2
多线程情况下，就不安全了。
```

Thread Safe 计数器

```
public class AtomicDemo {
    private static AtomicInteger atomicInteger = new
AtomicInteger(1);

    public static void main(String[] args) {
        System.out.println(atomicInteger.getAndIncrement());
//count++ vs ++count
        System.out.println(atomicInteger.get());//count
    }
}

输出结果:
1
2
```

Conclusion

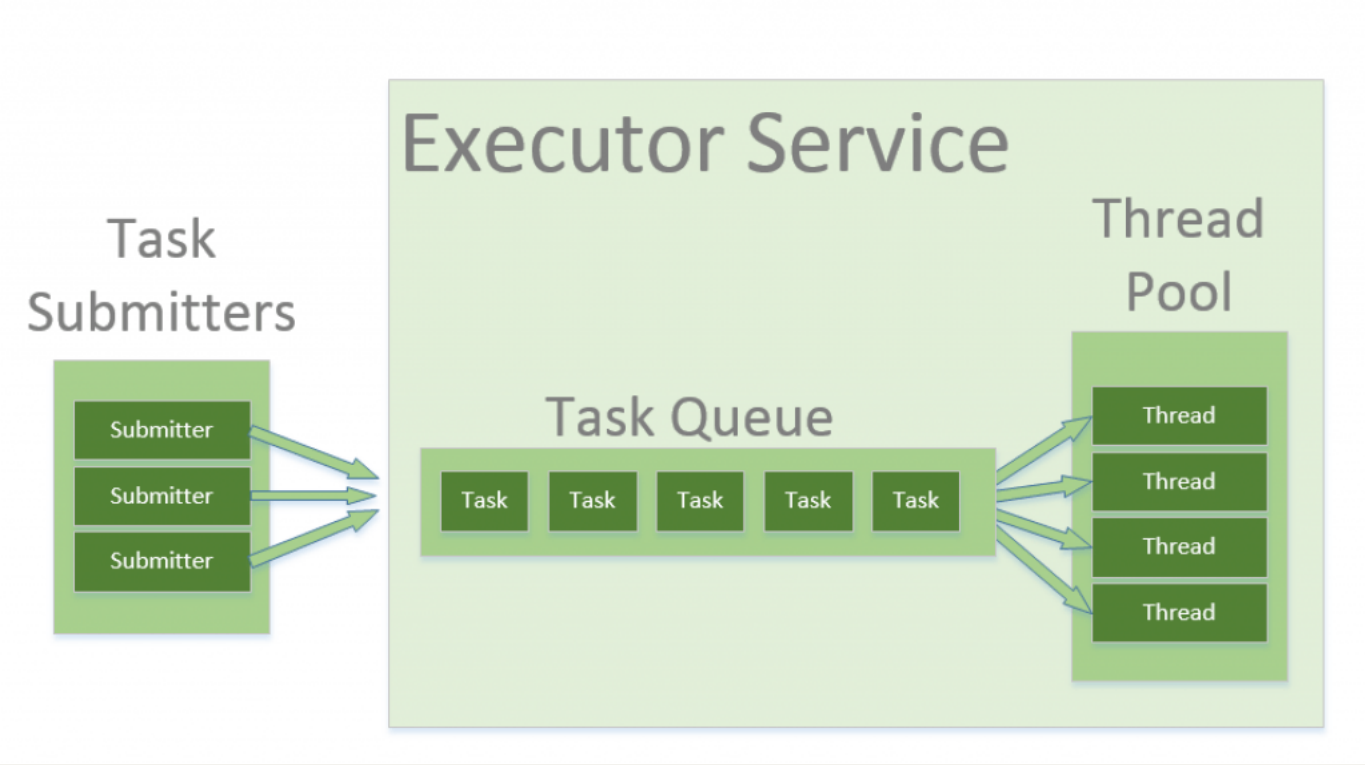
atomic包提高原子更新基本类型的工具类，主要有这些：

1. AtomicBoolean：以原子更新的方式更新boolean；
2. AtomicInteger：以原子更新的方式更新Integer;
3. AtomicLong：以原子更新的方式更新Long；

这几个类的用法基本一致，这里以AtomicInteger为例总结常用的方法

1. addAndGet(int delta)：以原子方式将输入的数值与实例中原本的值相加，并返回最后的结果；count = count + delta
2. incrementAndGet()：以原子的方式将实例中的原值进行加1操作，并返回最终相加后的结果；++counter
3. getAndSet(int newValue)：将实例中的值更新为新值，并返回旧值；
4. getAndIncrement()：以原子的方式将实例中的原值加1，返回的是自增前的旧值；counter++

Thread Pool (重要，easy)




```java
// 创建固定大小的线程池：
ExecutorService executor = Executors.newFixedThreadPool(4);
// 提交任务：
executor.submit(task1);
executor.submit(task2);
executor.submit(task3);
executor.submit(task4);
executor.submit(task5);
```

```java
Thread t = new MyThread();
  <- vs ->  ExecutorService executor =
Executors.newFixedThreadPool(4);

t.start();

```

特性	单个线程	线程池
创建	`new Thread(runnable)`	`Executors.newFixedThreadPool(n)` 或 `new ThreadPoolExecutor(...)`
执行	`thread.start()`	`executorService.submit(runnable)` 或 `execute(runnable)`
资源消耗	每个任务创建一个新线程，占用更多资源	固定数量或可配置数量的线程，共享线程资源
性能	可能因为频繁创建和销毁线程导致性能低下	复用线程，减少线程创建和销毁的开销，提高性能
线程生命周期管理	需要手动管理线程的生命周期	线程池自动管理线程的生命周期
并发控制	难以控制并发任务的数量	通过配置线程池大小来控制并发任务的数量
任务排队与执行策略	无法排队等待执行的任务	可以对等待执行的任务进行排队
结果获取（可选）	不直接返回结果，需要手动同步	`submit(runnable)` 返回 `Future` 对象，可用于获取结果
错误处理（可选）	需要在任务内部处理错误	可以提供一个 `RejectedExecutionHandler` 来处理错误
关闭线程/线程池	无法直接关闭线程，需要实现中断逻辑	使用 `executorService.shutdown()` 或 `shutdownNow()` 关闭线程池

使用线程池通常比直接创建单个线程具有更好的性能和资源管理。线程池可以控制并发任务的数量，减少线程创建和销毁的开销，提高性能。此外，线程池还可以对等待执行的任务进行排队，自动管理线程的生命周期，并提供更灵活的错误处理机制。然而，在某些简单的场景中，使用单个线程可能会更简单。

FEATURE	SINGLE THREAD	THREAD POOL
Creation	`new Thread(runnable)`	`Executors.newFixedThreadPool(n)` or `new ThreadPoolExecutor(...)`
Execution	`thread.start()`	`executorService.submit(runnable)` or `execute(runnable)`
Resource Consumption	New thread created for each task, consuming more resources	Fixed or configurable number of threads, sharing thread resources
Performance	Potential performance issues due to frequent thread creation and destruction	Thread reuse, reducing overhead of thread creation and destruction, improving performance
Thread Lifecycle Management	Manual thread lifecycle management	Automatic thread lifecycle management by thread pool
Concurrency Control	Difficult to control the number of concurrent tasks	Control the number of concurrent tasks by configuring the pool size
Task Queuing & Execution Strategy	No task queuing for waiting tasks	Allows queuing of tasks waiting for execution
Result Retrieval (optional)	No direct result return, manual synchronization required	`submit(runnable)` returns a `Future` object for result retrieval
Error Handling (optional)	Error handling within the task	Provide a `RejectedExecutionHandler` for handling errors
Thread/Pool Shutdown	No direct way to close thread, interrupt logic implementation required	Close thread pool using `executorService.shutdown()` or `shutdownNow()`

Using thread pools generally offers better performance and resource management compared to creating single threads directly. Thread pools control the number of concurrent tasks, reduce the overhead of thread creation and destruction, and improve performance. Moreover, thread pools allow task queuing for pending execution,

automatically manage thread lifecycles, and provide more flexible error handling mechanisms. However, in some simple scenarios, using a single thread might be more straightforward.

因为`ExecutorService`只是接口，Java标准库提供的几个常用实现类有：

- FixedThreadPool：线程数固定的线程池；
- CachedThreadPool：线程数根据任务动态调整的线程池；
- SingleThreadExecutor：仅单线程执行的线程池。

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        // 创建一个固定大小的线程池：
        ExecutorService es = Executors.newFixedThreadPool(4);
        for (int i = 0; i < 6; i++) {
            es.submit(new Task("" + i));
          // 不用thread pool的话，怎么创建和执行一个thread.
          // Thread t0 = new Thread(new Task(" " + i));
          // t.start();
        }
        // 关闭线程池：
        es.shutdown();
    }
}

class Task implements Runnable {
    private final String name;

    public Task(String name) {
        this.name = name;
    }

    @Override
```

```
    public void run() {
        System.out.println("start task " + name);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        System.out.println("end task " + name);
    }
}
```

Question

1. How many ways we can create a thread for task?
    a. `new Thread(new Task("name"))`
    b. `es.submit(new Task("name"))`

Future （超级大重点）

runnable 没有返回值，callable有返回值。并且**`Callable`**接口是一个泛型接口，可以返回指定类型的结果。

现在的问题是，如何获得异步执行的结果?

如果仔细看`ExecutorService.submit()`方法，可以看到，它返回了一个`Future`类型，一个`Future`类型的实例代表一个未来能获取结果的对象：

```
class Task implements Callable<String> {
    public String call() throws Exception {
        return longTimeCalculation();
    }
}

ExecutorService executor = Executors.newFixedThreadPool(4);
// 定义任务：
```

```
Callable<String> task = new Task();
// 提交任务并获得Future:
Future<String> future1 = executor.submit(task);
....
  ....
  ....
System.out.println(future.get());
int res = add(2, 3);
Systemout.println(res);

main            thread
  submit()       run task
  sdsfsdfss        1
    dsdsd              2
    sdss             res
sdsds
  dsdsd
  dsds
  future.get()
                    re s
...
// 从Future获取异步执行返回的结果:
String result = future.get(); // 可能阻塞

...
if (future.isDone()) {
  future.get();
} else {
  future.cancel()
}
```

当我们提交一个`Callable`任务后，我们会同时获得一个`Future`对象，然后，我们在主线程某个时刻调用`Future`对象的`get()`方法，就可以获得异步执行的结果。在调用`get()`时，如果异步任务已经完成，我们就直接获得结果。如果异步任务还没有完成，那么`get()`会阻塞，直到任务完成后才返回结果。

一个`Future<V>`接口表示一个未来可能会返回的结果，它定义的方法有：

- `get()`：获取结果（可能会等待）
- `get(long timeout, TimeUnit unit)`：获取结果，但只等待指定的时间；
- `cancel(boolean mayInterruptIfRunning)`：取消当前任务；
- `isDone()`：判断任务是否已完成。

对线程池提交一个`Callable`任务，可以获得一个`Future`对象；

可以用`Future`在将来某个时刻获取结果。

假设我们要出一个商家的过去一个月的销售订单金额。并且该步骤消耗时间很长。

传统方法（顺序执行）：

1. 获得所有订单的信息（阻塞，等待结果，数据量大，耗时间很长）
2. 获得商家信息
3. 从daily file中获得数据
4. 在数据库中创建一行含有基本信息的记录。
5. 并将订单数据处理并populate到数据库中

多线程（并发执行）：

1. 获得所有订单的信息, Future future = executor.submit(task); (不阻塞)
2. 获得商家信息
3. 从daily file中获得数据
4. 在数据库中创建一行含有基本信息的记录。
5. future.get()获取订单数据，并将订单数据处理并populate到数据库中。
    a. 可能阻塞，如果订单数据还没获取完

b. 可能不阻塞，已经有了结果。

CompletableFuture (超级大重点)

Introduction to CompletableFuture

- An **asynchronous programming tool** introduced in **Java 8**
- Implements the Future interface, providing powerful asynchronous capabilities
- Supports **chaining operations** for easy combination and management of multiple asynchronous tasks
- **Non-blocking** asynchronous operations

使用`Future`获得异步执行结果时，要么调用阻塞方法`get()`，要么轮询看`isDone()`是否为`true`，这两种方法都不是很好，因为主线程也会被迫等待。

从Java 8开始引入了`CompletableFuture`，它针对`Future`做了改进，可以传入回调对象，当异步任务完成或者发生异常时，自动调用回调对象的回调方法。

```
public class Main {
    public static void main(String[] args) throws Exception {
        // 创建异步执行任务:
        CompletableFuture<Double> cf =
CompletableFuture.supplyAsync(Main::fetchPrice);
        // 如果执行成功:
        cf.thenAccept((result) -> {
            System.out.println("price: " + result);
        });
        // 如果执行异常:
        cf.exceptionally((e) -> {
            e.printStackTrace();
            return null;
        });
```

```
14          // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭：
15          Thread.sleep(200);
16      }
17
18      static Double fetchPrice() {
19          try {
20              Thread.sleep(100);
21          } catch (InterruptedException e) {
22          }
23          if (Math.random() < 0.3) {
24              throw new RuntimeException("fetch price failed!");
25          }
26          return 5 + Math.random() * 20;
27      }
28  }
```

`CompletableFuture`可以指定异步处理流程：

- `thenAccept()`处理正常结果；
- `exceptional()`处理异常结果；
- `thenApplyAsync()`用于串行化另一个`CompletableFuture`；
- `anyOf()`和`allOf()`用于并行化多个`CompletableFuture`。

Core Concepts and Methods of CompletableFuture

Creating a CompletableFuture

- `CompletableFuture.supplyAsync(Supplier<U> supplier)`: Executes the given task asynchronously and returns the result

- `CompletableFuture.runAsync(Runnable runnable)`: Executes the given task asynchronously, without returning a value

```
supplyAsync(fetch price). -> cf.thenAccept((result) -> {
            System.out.println("price: " + result);
        });

supplyAsync(fetch price). -> cf.thenApply((result) -> {
            System.out.println("price: " + result);
        }).thenAccept();
```

Result Processing and Transformation

- `CompletableFuture.thenApply(Function<T, U> fn)`: Processes the return value and transforms it into another type
- `CompletableFuture.thenAccept(Consumer<T> action)`: Processes the return value without returning a new CompletableFuture
- `CompletableFuture.thenRun(Runnable action)`: Ignores the return value and runs a runnable

Combination and Linking

- `CompletableFuture.thenCompose(Function<T, CompletionStage<U>> fn)`: Links another CompletableFuture and passes the result to the next task
- `CompletableFuture.thenCombine(CompletionStage<U> other, BiFunction<T, U, V> fn)`: Combines the results of two CompletableFutures and returns a new CompletableFuture

Exception Handling

- `CompletableFuture.exceptionally(Function<Throwable, T> fn)`: Handles exceptions and returns an alternative value
- `CompletableFuture.handle(BiFunction<T, Throwable, U> fn)`: Handles exceptions while allowing access to the result value (if any)

Practical Application Scenarios

> *Scenario: Retrieve data from multiple APIs, then merge and process the results*

```
CompletableFuture<String> api1 = CompletableFuture.supplyAsync(()
-> {
    // Simulate API call
    return "API 1 result";
});
CompletableFuture<String> api2 = CompletableFuture.supplyAsync(()
-> {
    // Simulate API call
    return "API 2 result";
});

// Merge and process the results
// api1的结果和api2的结果合并。result 1 from api1, result 2 from api
2.
CompletableFuture<String> combinedResult = api1.thenCombine(api2,
(result1, result2) -> {
    return result1 + ", " + result2; //new
CompletableFuture(result1 + ", " + result2);
});
CF.thenCombine(api1, api2)
  api1.thenCombine(ap2, (api1Result, api2Result) -> {})
// Get the final result
String result = combinedResult.get();
System.out.println(result); // Output: API 1 result, API 2 result
```

More Information: https://www.liaoxuefeng.com/wiki/1252599548343744/1306581182447650

ForkJoin

类似于MapReduce. parallel operating system 也是用的这个原理，利用GPU的计算也是这个原理

Fork/Join是一种基于“分治”的算法：通过分解任务，并行执行，最后合并结果得到最终结果。

`ForkJoinPool`线程池可以把一个大任务分拆成小任务并行执行，任务类必须继承自`RecursiveTask`或`RecursiveAction`。

使用Fork/Join模式可以进行并行计算以提高效率。

More Information: https://www.liaoxuefeng.com/wiki/1252599548343744/1306581226487842

ThreadLocal （重要）

More Information: https://www.liaoxuefeng.com/wiki/1252599548343744/1306581251653666

问题：**如何在一个线程内传递状态?**

```java
public void process(User user) {
    checkPermission(user);
    doWork(user);
    saveStatus(user);
    sendResponse(user);
}

void doWork(User user) {
    queryStatus(user);
    checkStatus();
    setNewStatus(user);
    log();
}
```

这种在一个线程中，横跨若干方法调用，需要传递的对象，我们通常称之为上下文（**Context**），它是一种状态，可以是用户身份、任务信息等。

给每个方法增加一个context参数非常麻烦，而且有些时候，如果调用链有无法修改源码的第三方库，`User`对象就传不进去了。

`ThreadLocal`实例通常总是以静态字段初始化如下：

在父方法中threadLocalUser.set(user); 子方法们threadLocalUser.get();便可得到数据

```java
static ThreadLocal<User> threadLocalUser = new ThreadLocal<>();
void processUser(user) {
    try {
        threadLocalUser.set(user);
        step1();
        step2();
    } finally {
        threadLocalUser.remove();
    }
}

```

```
void step1() {
    User u = threadLocalUser.get();
    log();
    printUser();
}

void log() {
    User u = threadLocalUser.get();
    println(u.name);
}

void step2() {
    User u = threadLocalUser.get();
    checkUser(u.id);
}
```

Conclusion

- `ThreadLocal`表示线程的“局部变量”，它确保每个线程的`ThreadLocal`变量都是各自独立的；
- `ThreadLocal`适合在一个线程的处理流程中保持上下文（避免了同一参数在所有方法中传递）；
- 使用`ThreadLocal`要用`try ... finally`结构，并在`finally`中清除。