极客大学

极客大学 Java 进阶训练营
第 6 课
Java 并发编程（1）

KimmKing

Apache Dubbo/ShardingSphere PMC

个人介绍

极客大学

Apache Dubbo/ShardingSphere PMC

前某集团高级技术总监/阿里架构师/某银行北京研发中心负责人

阿里云 MVP、腾讯 TVP、TGO 会员

10 多年研发管理和架构经验

熟悉海量并发低延迟交易系统的设计实现

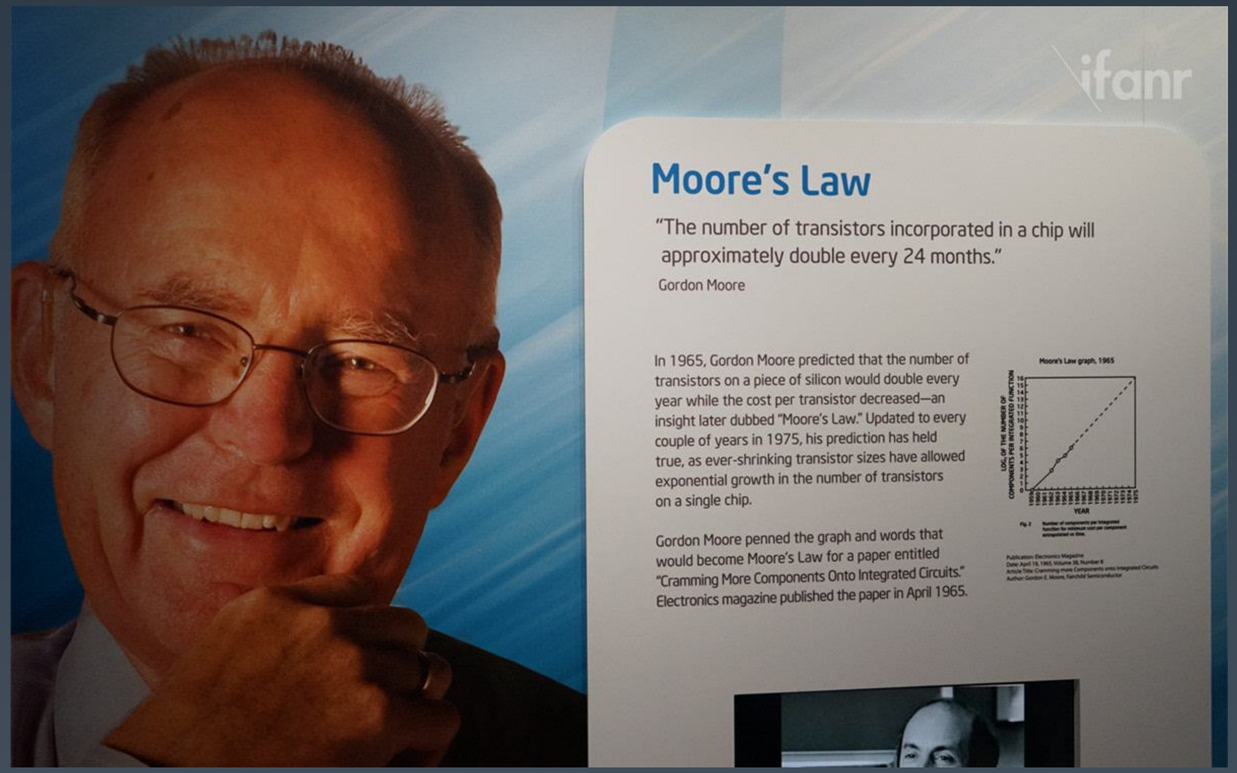
极客大学

目录

极客大学

1.多线程基础

为什么会有多线程

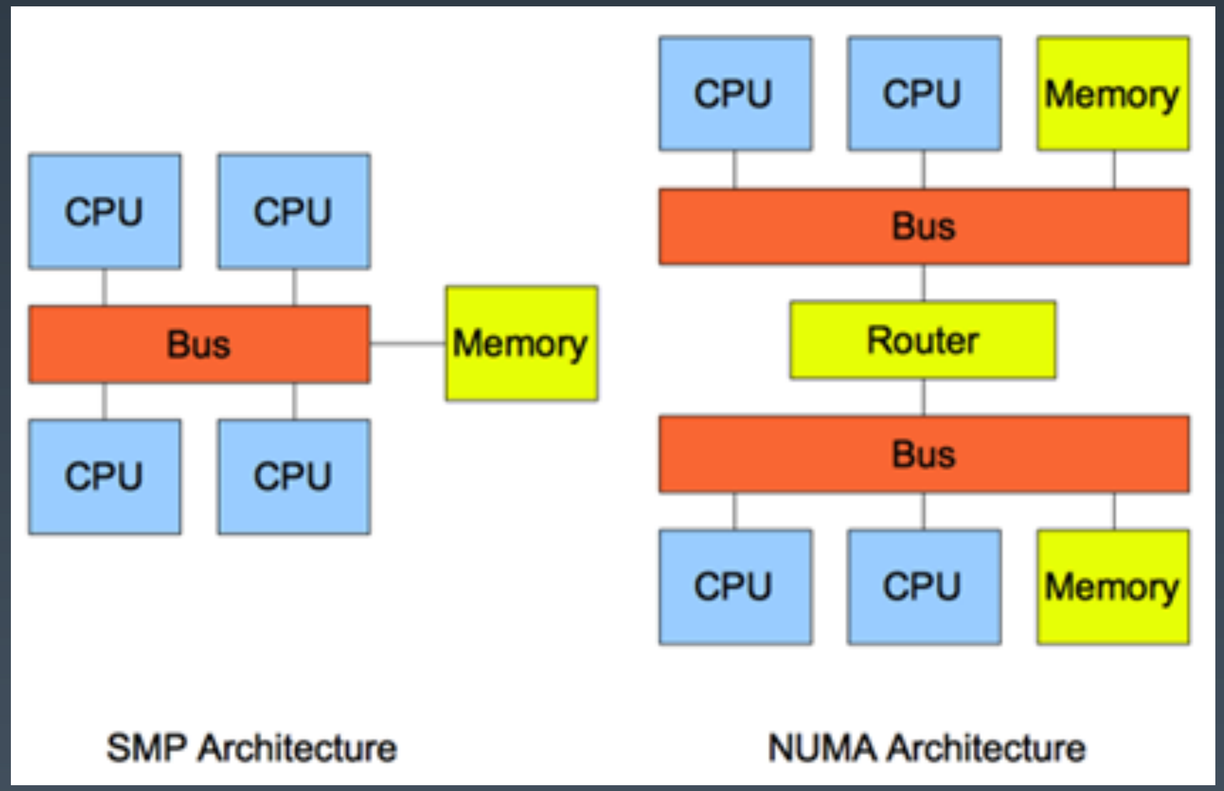


本质原因是摩尔定律失效 -> 多核+分布式时代的来临。

JVM、NIO 是不是都因为这个问题变复杂？

后面讲的分布式系统，也是这个原因。

为什么会有多线程



多 CPU 核心意味着同时操作系统有更多的并行计算资源可以使用。

操作系统以线程作为基本的调度单元。

单线程是最好处理不过的。

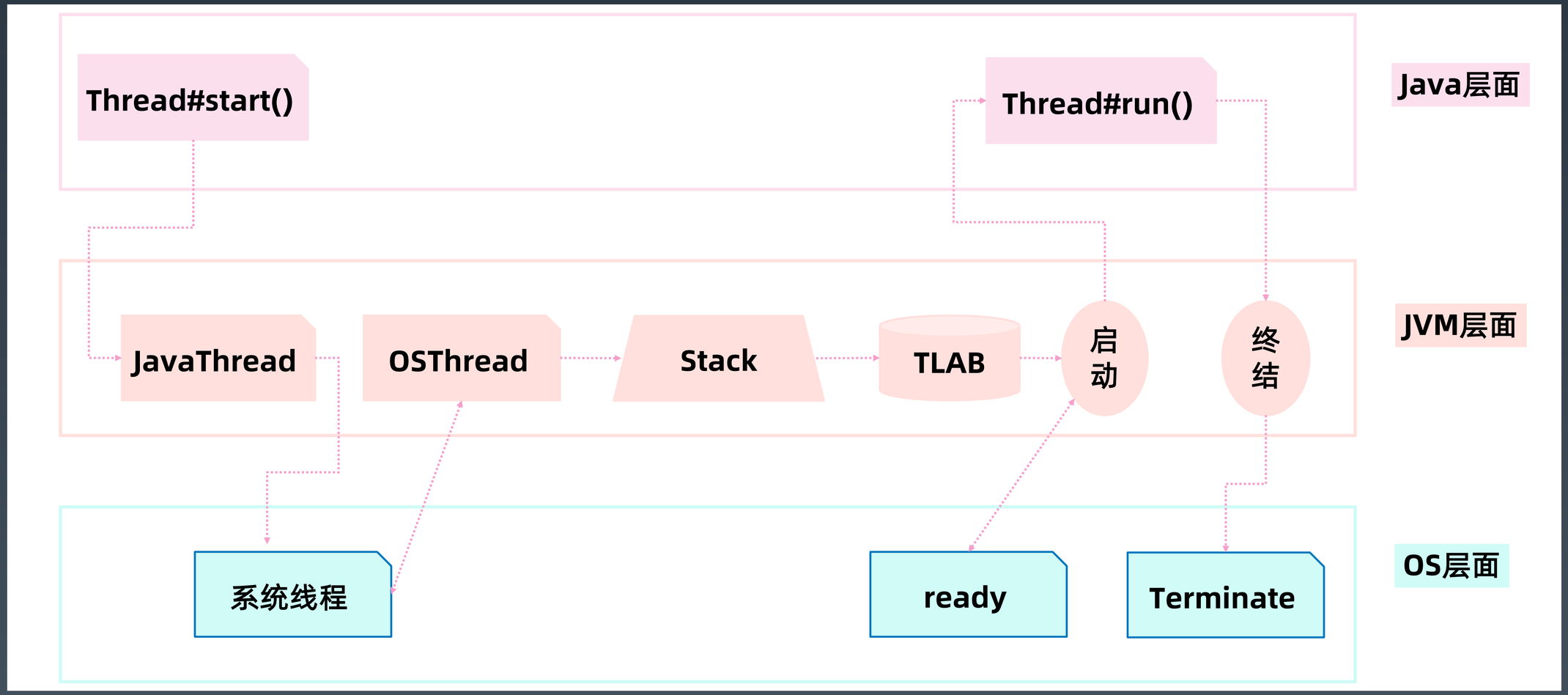
线程越多，管理复杂度越高。

跟我们程序员都喜欢自己单干一样。

《人月神话》里说加人可能干得更慢。

可见多核时代的编程更有挑战。

Java 线程的创建过程



线程与进程的区别是什么？

2. Java 多线程*

Thread 使用示例

```
public static void main(String[] args) {
  Runnable task = new Runnable() {
    @Override
    public void run() {
      try {
        Thread.sleep(5000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      Thread t = Thread.currentThread();
      System.out.println("当前线程:" + t.getName());
    }
  };
  Thread thread = new Thread(task);
  thread.setName("test-thread-1");
  thread.setDaemon(true);
  thread.start();
}
```

- 守护线程
- 启动方式

思考:

1. 输出结果是什么?
2. 为什么?
3. 有哪些方式可以修改?

基础接口 - Runnable

```
// 接口定义
public interface Runnable {
    public abstract void run();
}
```

```
// 重要实现
Thread implements Runnable ...
```

辨析:

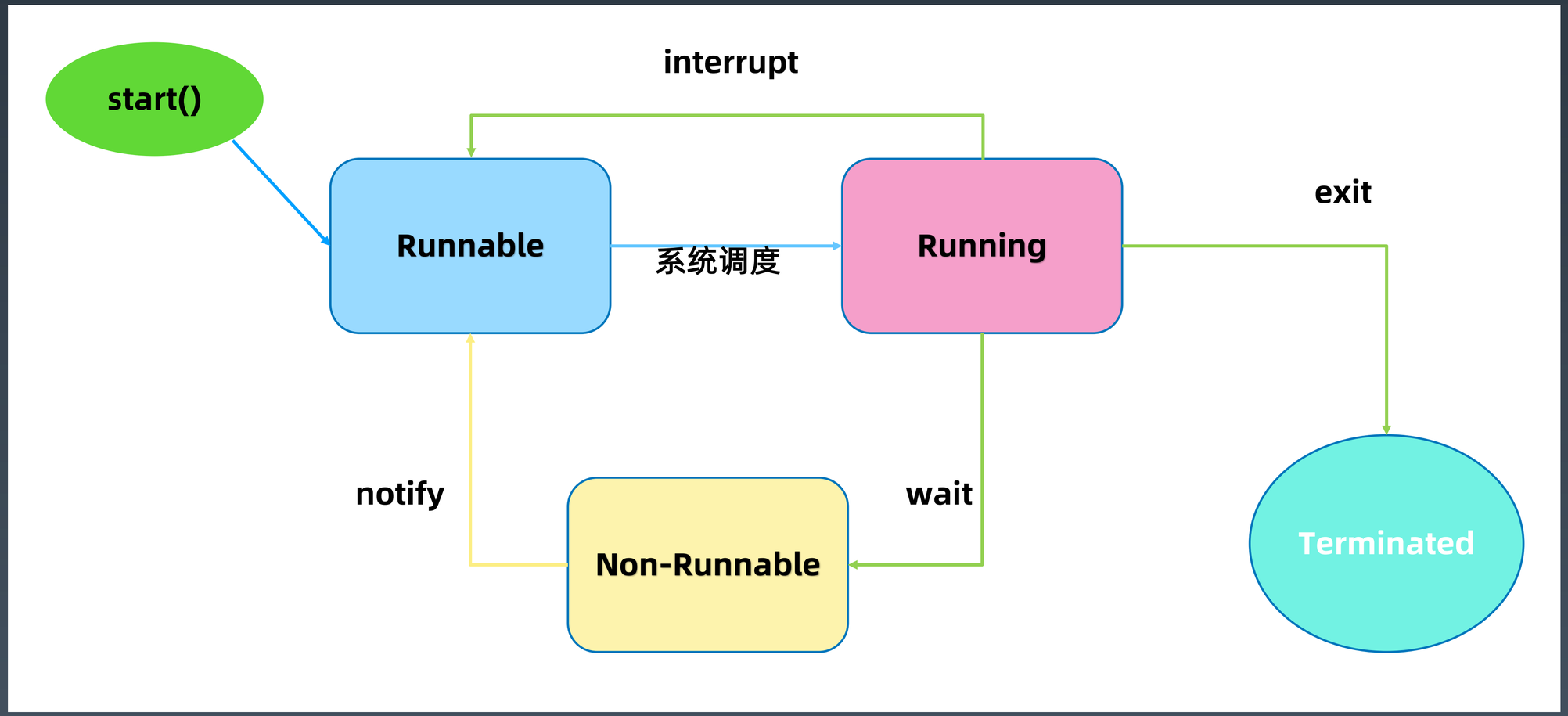
Thread#start():创建新线程

Thread#run()：本线程调用

```
// 示例1
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("业务逻辑...");
    }
};
```

```
// 示例2
public class XXXTask implements Runnable {
    @Override
    public void run() {
        System.out.println("执行逻辑...");
    }
}
```

线程状态


start()
interrupt
Runnable
系统调度
Running
exit
notify
Non-Runnable
wait
Terminated

Thread 类

重要属性/方法	说明
volatile String **name**;	线程名称 - 诊断分析使用
boolean daemon = **false**;	后台守护线程标志 - 决定JVM优雅关闭
Runnable **target**;	任务(只能通过构造函数传入)
synchronized void start()	【协作】启动新线程并自动执行
void join()	【协作】等待某个线程执行完毕（来汇合）
static native Thread currentThread();	静态方法: 获取当前线程信息
static native void sleep(**long** millis);	静态方法: 线程睡眠并让出CPU时间片

Thread: 线程

wait & notify

Object# 方法	说明
void wait()	放弃锁+等待0ms+尝试获取锁;
void wait(**long** timeout, **int** nanos)	放弃锁 + wait + 到时间自动唤醒/中途唤醒 (精度: nanos>0则 timeout++)
native void wait(**long** timeout);	放弃锁+ wait + 到时间自动唤醒/中途被唤醒 (唤醒之后需要自动获取锁)
native void notify();	发送信号通知1个等待线程
native void notifyAll();	发送信号通知所有等待线程

辨析:

- Thread.sleep: 释放 CPU
- Object#wait ：释放锁

Thread 的状态改变操作

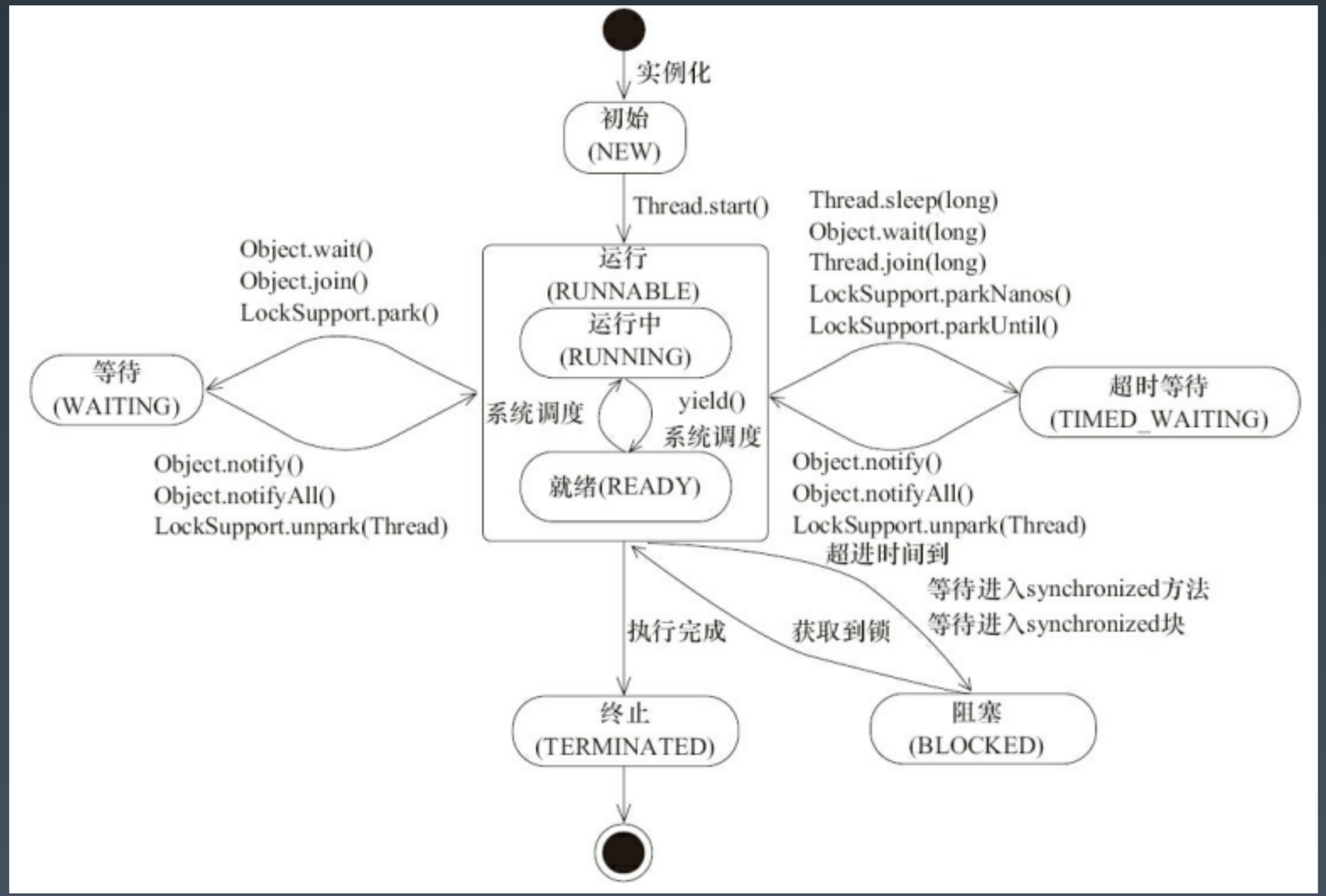
1. Thread.sleep(long millis)，一定是当前线程调用此方法，当前线程进入 TIMED_WAITING 状态，但不释放对象锁，millis 后线程自动苏醒进入就绪状态。作用：给其它线程执行机会的最佳方式。
2. Thread.yield()，一定是当前线程调用此方法，当前线程放弃获取的 CPU 时间片，但不释放锁资源，由运行状态变为就绪状态，让 OS 再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一定会轮流执行。实际中无法保证 yield() 达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。Thread.yield() 不会导致阻塞。该方法与 sleep() 类似，只是不能由用户指定暂停多长时间。
3. t.join()/t.join(long millis)，当前线程里调用其它线程 t 的 join 方法，当前线程进入WAITING/TIMED_WAITING 状态，当前线程不会释放已经持有的对象锁。线程t执行完毕或者 millis 时间到，当前线程进入就绪状态。
4. obj.wait()，当前线程调用对象的 wait() 方法，当前线程释放对象锁，进入等待队列。依靠 notify()/notifyAll() 唤醒或者 wait(long timeout) timeout 时间到自动唤醒。
5. obj.notify() 唤醒在此对象监视器上等待的单个线程，选择是任意性的。notifyAll() 唤醒在此对象监视器上等待的所有线程。

Thread 的中断与异常处理

1. 线程内部自己处理异常，不溢出到外层。
2. 如果线程被 Object.wait, Thread.join 和 Thread.sleep 三种方法之一阻塞，此时调用该线程的interrupt() 方法，那么该线程将抛出一个 InterruptedException 中断异常（该线程必须事先预备好处理此异常），从而提早地终结被阻塞状态。如果线程没有被阻塞，这时调用 interrupt() 将不起作用，直到执行到 wait(),sleep(),join() 时,才马上会抛出 InterruptedException。
3. 如果是计算密集型的操作怎么办？

   分段处理，每个片段检查一下状态，是不是要终止。

Thread 状态



1、本线程主动操作

2、被动：

- 遇到锁
- 被通知

极客大学

3.线程安全*

多线程执行会遇到什么问题?

竞态条件

临界区

共享与同步
(线程间通信)

多线程

多个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。

导致竞态条件发生的代码区称作临界区。

不进行恰当的控制，会导致线程安全问题

并发相关的性质

原子性：原子操作，注意跟事务 ACID 里原子性的区别与联系

对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

只有语句1是原子操作。

```
1   x = 10;          //语句1
2   y = x;           //语句2
3   x++;             //语句3
4   x = x + 1;       //语句4
```

多个线程并发问题

类似于

多个事务的并发问题

并发相关的性质

可见性：对于可见性，Java 提供了 volatile 关键字来保证可见性。

当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

另外，通过 synchronized 和 Lock 也能够保证可见性，synchronized 和 Lock 能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。

volatile 并不能保证原子性。

并发相关的性质

有序性：Java 允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。可以通过 volatile 关键字来保证一定的“有序性”（synchronized 和 Lock 也可以）。

happens-before 原则（先行发生原则）：

1. 程序次序规则：一个线程内，按照代码先后顺序
2. 锁定规则：一个 unLock 操作先行发生于后面对同一个锁的 lock 操作
3. Volatile 变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
4. 传递规则：如果操作 A 先行发生于操作 B，而操作 B 又先行发生于操作 C，则可以得出 A 先于 C
5. 线程启动规则：Thread 对象的 start() 方法先行发生于此线程的每个一个动作
6. 线程中断规则：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生
7. 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过 Thread.join() 方法结束、Thread.isAlive() 的返回值手段检测到线程已经终止执行
8. 对象终结规则：一个对象的初始化完成先行发生于他的 finalize() 方法的开始

一个简单的实际例子

最简单的例子

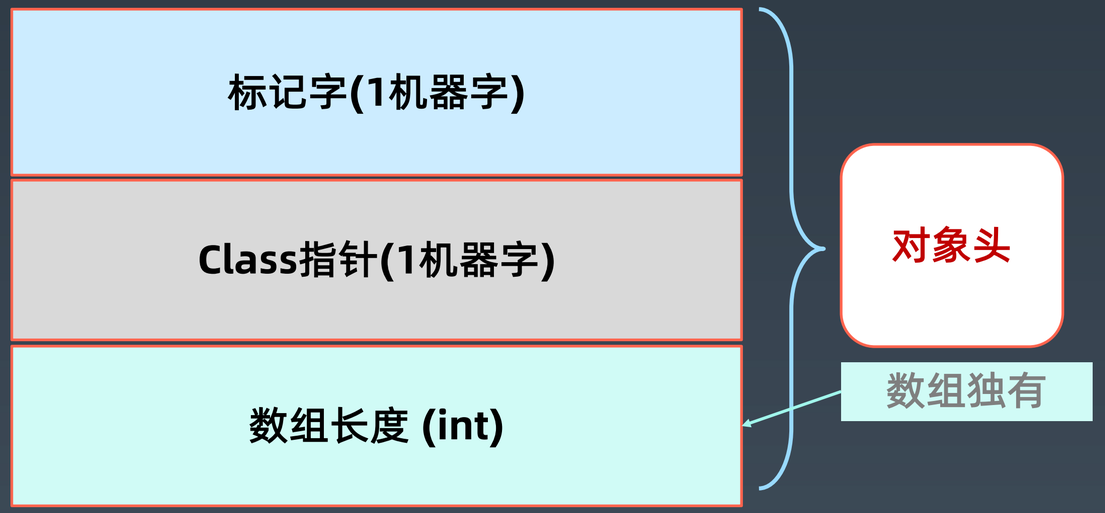
多线程计数

如何解决?

synchronized 的实现

1. 使用对象头标记字(Object monitor)
2. Synchronized 方法优化
3. 偏向锁: BiaseLock

锁状态	31bit(25bit unused)		4bit	1bit	2bit
	54bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向轻量级锁的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空				11



synchronized 使用示例

```
public class SyncCounter {
    private int sum = 0;
    public synchronized int incrAndGet() {
        return ++sum;
    }
    public int addAndGet() {
        synchronized (this) {
            return ++sum;
        }
    }
    public int getSum() {
        return sum;
    }
}
```

```
// 测试代码
public static void testSyncCounter1() {
    int loopNum = 100_0000;
    SyncCounter counter = new SyncCounter();
    IntStream.range(0, loopNum).parallel()
      .forEach(i -> counter.incrAndGet());
}
```

思考: 哪种方式性能更高?

- 同步块　: 粒度小
- 同步方法: 专有指令

volatile

1. 每次读取都强制从主内存刷数据
2. 适用场景： 单个线程写；多个线程读
3. 原则： 能不用就不用，不确定的时候也不用
4. 替代方案： Atomic 原子操作类

```
1 | x = 2;          //语句1
2 | y = 0;          //语句2
3 | flag = true;    //语句3
4 | x = 4;          //语句4
5 | y = -1;         //语句5
```

那么，语句1和2，不会被重排到3的后面，4和5也不会到前面。

同时可以保证1和2的结果是对3、4、5可见。

final

final 定义类型	说明
final class XXX	不允许继承
final 方法	不允许 Override
final 局部变量	不允许修改
final 实例属性	• 构造函数/初始化块/<init>之后不允许变更； • 只能赋值一次 • **安全发布**：构造函数结束返回时，final 域最新的值被保证对其他线程可见
final static 属性	<clinit>静态块执行后不允许变更； 只能赋值一次

思考：final 声明的引用类型与原生类型在处理时有什么区别?

Java 里的常量替换。写代码最大化用 final 是个好习惯。

极客大学

4.线程池原理与应用*

线程池

1. Excutor: 执行者 - 顶层接口
2. ExcutorService: 接口 API
3. ThreadFactory: 线程工厂
4. Excutors: 工具类

Executor - 执行者

重要方法	说明
void execute(Runnable command);	执行可运行的任务

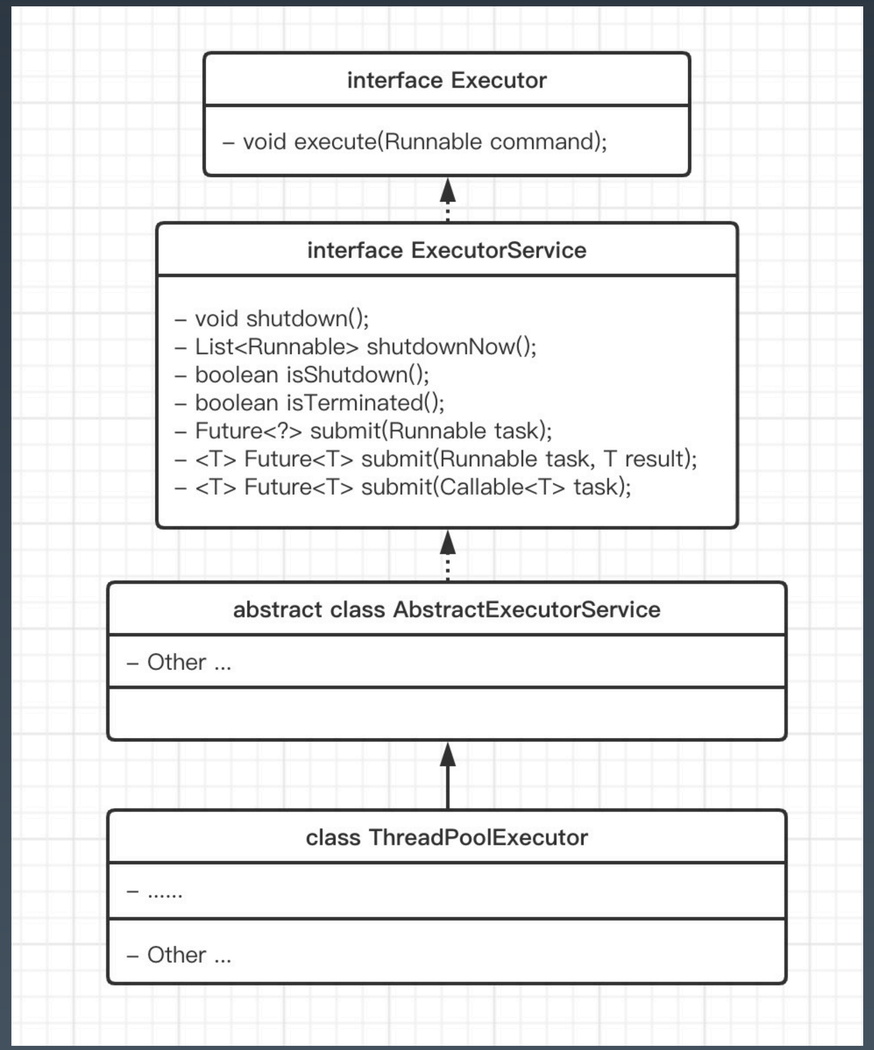
线程池从功能上看，就是一个任务执行器

submit 方法 -> 有返回值，用 Future 封装

execute 方法 -> 无返回值

submit 方法还异常可以在主线程中 catch 到。

execute 方法执行任务是捕捉不到异常的。



ExecutorService

重要方法	说明
void execute(Runnable command);	执行可运行的任务
void shutdown();	关闭线程池
List<Runnable> shutdownNow();	立即关闭
Future<?> submit(Runnable task);	提交任务；允许获取执行结果
<T> Future<T> submit(Runnable task, T result);	提交任务（指定结果）；控制|获取执行结果
<T> Future<T> submit(Callable<T> task);	提交任务；允许控制任务和获取执行结果

shutdown()：停止接收新任务，原来的任务继续执行

shutdownNow()：停止接收新任务，原来的任务停止执行

awaitTermination(long timeOut, TimeUnit unit)：当前线程阻塞

ThreadFactory

重要方法	说明
Thread newThread(Runnable r);	创建新线程

ThreadPoolExecutor 提交任务逻辑:

1. 判断 corePoolSize 【创建】
2. 加入 workQueue
3. 判断 maximumPoolSize 【创建】
4. 执行拒绝策略处理器

ThreadPoolExecutor.java

```
1342    public void execute(Runnable command) {
1343        if (command == null)
1344            throw new NullPointerException();
1345        /*...*/
1365        int c = ctl.get();
1366        if (workerCountOf(c) < corePoolSize) {
1367            if (addWorker(command, core: true))
1368                return;
1369            c = ctl.get();
1370        }
1371        if (isRunning(c) && workQueue.offer(command)) {
1372            int recheck = ctl.get();
1373            if (! isRunning(recheck) && remove(command))
1374                reject(command);
1375            else if (workerCountOf(recheck) == 0)
1376                addWorker( firstTask: null,  core: false);
1377        }
1378        else if (!addWorker(command,  core: false))
1379            reject(command);
1380    }
```

线程池参数

缓冲队列

BlockingQueue 是双缓冲队列。BlockingQueue 内部使用两条队列，允许两个线程同时向队列一个存储，一个取出操作。在保证并发安全的同时，提高了队列的存取效率。

1. ArrayBlockingQueue:规定大小的 BlockingQueue，其构造必须指定大小。其所含的对象是 FIFO 顺序排序的。
2. LinkedBlockingQueue:大小不固定的 BlockingQueue，若其构造时指定大小，生成的 BlockingQueue 有大小限制，不指定大小，其大小有 Integer.MAX_VALUE 来决定。其所含的对象是 FIFO 顺序排序的。
3. PriorityBlockingQueue:类似于 LinkedBlockingQueue，但是其所含对象的排序不是 FIFO，而是依据对象的自然顺序或者构造函数的 Comparator 决定。
4. SynchronizedQueue:特殊的 BlockingQueue，对其的操作必须是放和取交替完成。

线程池参数

拒绝策略

1. ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出 RejectedExecutionException 异常。
2. ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常。
3. ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新提交被拒绝的任务
4. ThreadPoolExecutor.CallerRunsPolicy：由调用线程（提交任务的线程）处理该任务

ThreadFactory 示例

极客大学

```
public class CustomThreadFactory implements ThreadFactory {
    private AtomicInteger serial = new AtomicInteger(0);

    @Override
    public Thread newThread(Runnable r) {
        Thread thread = new Thread(r);
        thread.setDaemon(true); // 根据需要，设置守护线程
        thread.setName("CustomeThread-" + serial.getAndIncrement());
        return thread;
    }
}
```

ThreadPoolExecutor

重要属性/方法	说明
int corePoolSize;	核心线程数
int maximumPoolSize;	最大线程数
ThreadFactory **threadFactory**;	线程创建工厂
BlockingQueue<Runnable> **workQueue**;	工作队列
RejectedExecutionHandler **handler**;	拒绝策略处理器
void execute(Runnable command)	执行
Future<?> submit(Runnable task)	提交任务
submit(Runnable task, T result)	提交任务
submit(Callable<T> task)	提交任务

ThreadPoolExecutor 示例

```
public static ThreadPoolExecutor initThreadPoolExecutor() {
    int coreSize = Runtime.getRuntime().availableProcessors();
    int maxSize = Runtime.getRuntime().availableProcessors() * 2;
    BlockingQueue<Runnable> workQueue = new
LinkedBlockingDeque<>(500);
    CustomThreadFactory threadFactory = new CustomThreadFactory();
    ThreadPoolExecutor executor = new ThreadPoolExecutor(coreSize,
maxSize,
            1, TimeUnit.MINUTES, workQueue, threadFactory);
    return executor;
}
```

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
```

创建线程池方法

极客大学

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2.newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

4.newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

Callable - 基础接口

重要方法	说明
V call() **throws** Exception;	调用执行

对比:

- Runnable#run()没有返回值
- Callable#call()方法有返回值

```
public class RandomSleepTask implements
Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        Integer sleep = new
Random().nextInt(10000);
        TimeUnit.MILLISECONDS.sleep(sleep);
        return sleep;
    }
}
```

Future - 基础接口

重要方法	说明
boolean cancel(**boolean** mayInterruptIfRunning);	取消任务 （执行时是否打断）
boolean isCancelled();	是否被取消
boolean isDone();	是否执行完毕
V get() **throws** InterruptedException, ExecutionException;	获取执行结果
V get(**long** timeout, TimeUnit unit) **throws** InterruptedException, ExecutionException, TimeoutException;	限时获取执行结果

```
public static void main(String[] args) throws Exception {
    Callable<Integer> task = new RandomSleepTask();
    ExecutorService executorService = 
initThreadPoolExecutor();
    Future<Integer> future1 = executorService.submit(task);
    Future<Integer> future2 = executorService.submit(task);
    // 等待执行结果
    Integer result1 = future1.get(1, TimeUnit.SECONDS);
    Integer result2 = future2.get(1, TimeUnit.SECONDS);
    System.out.println("result1=" + result1);
    System.out.println("result2=" + result2);
}
```

极客大学

5.总结回顾与作业实践

第六节课总结回顾

极客大学

多线程基础

Java 多线程

线程安全

线程池原理与应用

第六节课作业实践

1、（可选）跑一跑课上的各个例子，加深对多线程的理解

2、（可选）完善网关的例子，试着调整其中的线程池参数

THANKS!