多线程

1. java关键词volatile

java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致的更新，线程应该确保通过排他锁单独获得这个变量。

**Java语言提供了volatile**，在某些情况下比锁更加方便。如果一个字段被声明成volatile，java线程内存模型确保所有线程看到这个变量的值是一致的。

* 1. 实现原理
     1. 代码解析

**java ： instance = new Singleton();//instance是volatile变量**

**汇编语言：**

**0x01a3de1d: movb $0x0,0x1104800(%esi);**

**0x01a3de24: lock addl $0x0,(%esp);**

**有volatile变量修饰的共享变量进行写操作的时候会多第二行汇编代码，通过查IA-32架构软件开发者手册可知，lock前缀的指令在多核处理器下会引发了两件事情。**

**1、将当前处理器缓存行的数据会写回到系统内存。**

**2、这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效**

* + 1. 原理

处理器为什么使用缓存？

处理器为了提高处理速度，不直接和内存进行通讯，而是先将系统内存的数据读到内部缓存（L1,L2或其他）后再进行操作；

为什么使用volatile？

但操作完之后不知道何时会写到内存，如果对声明了Volatile变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。

如何解决多个CPU之间缓存数据不同步？

但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

这两件事情在IA-32软件开发者架构手册的第三册的多处理器管理章节（第八章）中有详细阐述。

Lock前缀指令会引起处理器缓存回写到内存。Lock前缀指令导致在执行指令期间，声言处理器的 LOCK# 信号。在多处理器环境中，LOCK# 信号确保在声言该信号期间，处理器可以独占使用任何共享内存。（因为它会锁住总线，导致其他CPU不能访问总线，不能访问总线就意味着不能访问系统内存），但是在最近的处理器里，LOCK＃信号一般不锁总线，而是锁缓存，毕竟锁总线开销比较大。在8.1.4章节有详细说明锁定操作对处理器缓存的影响，对于Intel486和Pentium处理器，在锁操作时，总是在总线上声言LOCK#信号。但在P6和最近的处理器中，如果访问的内存区域已经缓存在处理器内部，则不会声言LOCK#信号。相反地，它会锁定这块内存区域的缓存并回写到内存，并使用缓存一致性机制来确保修改的原子性，此操作被称为“缓存锁定”，缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域数据。

一个处理器的缓存回写到内存会导致其他处理器的缓存无效。IA-32处理器和Intel 64处理器使用MESI（修改，独占，共享，无效）控制协议去维护内部缓存和其他处理器缓存的一致性。在多核处理器系统中进行操作的时候，IA-32 和Intel 64处理器能嗅探其他处理器访问系统内存和它们的内部缓存。它们使用嗅探技术保证它的内部缓存，系统内存和其他处理器的缓存的数据在总线上保持一致。例如在Pentium和P6 family处理器中，如果通过嗅探一个处理器来检测其他处理器打算写内存地址，而这个地址当前处理共享状态，那么正在嗅探的处理器将无效它的缓存行，在下次访问相同内存地址时，强制执行缓存行填充。

* 1. 基础概念
     1. 共享变量

在多个线程之间能够被共享的变量被称为共享变量。

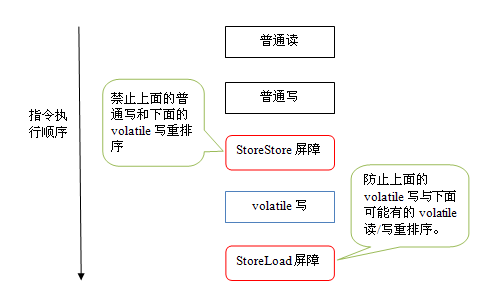
共享变量包括**所有的实例变量**，**静态变量**和**数组元素**。

他们都被存放在**堆**内存中，Volatile只作用于共享变量

* + 1. 内存屏障

是一组**处理器指令**，用于实现**对内存操作的顺序限制；**

**禁止处理器对命令进行重排序；**



* + 1. 缓存行

缓存中可以分配的**最小存储单位**。

处理器填写缓存线时会加载整个缓存线，需要使用多个主内存读周期。

* + 1. 原子操作

不可中断的一个或一系列操作

* + 1. 缓存行填充

当处理器识别到从**内存中读取操作数是可缓存的**，

处理器读取整个缓存行到适当的缓存（L1，L2，L3的或所有）

* + 1. 缓存命中

如果进行高速缓存行填充操作的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存（更快）。

* + 1. 写命中

当处理器将操作数**写回**到一个**内存缓存**的区域时，

它首先会检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存，这个操作被称为写命中。

* + 1. 写缺失

一个有效的缓存行被写入到不存在的内存区域

* 1. happens before关系

class VolatileExample {

int a = 0;

volatile boolean flag = false;

public void writer() {

a = 1; //1

flag = true; //2

}

public void reader() {

if (flag) { //3

int i = a; //4

……

}

}

}

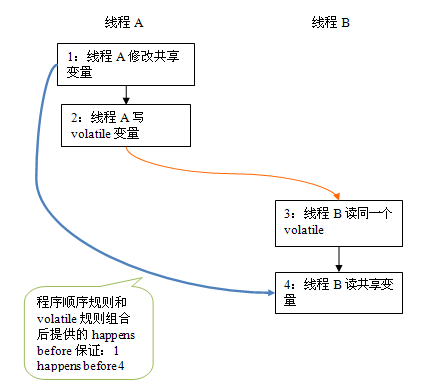
**假设**线程A执行writer()方法之后，线程B执行reader()方法。根据happens before规则，这个过程建立的happens before 关系可以分为两类：

根据程序次序（线程内）规则，1 happens before 2; 3 happens before 4。

根据volatile（线程之间）规则，2 happens before 3。

根据happens before 的传递性规则，1 happens before 4。

上述happens before 关系的图形化表现形式如下：



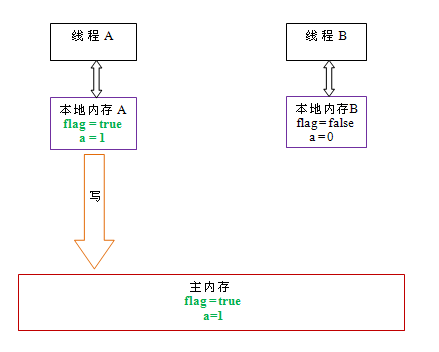
* 1. 读写内存语义
     1. 写内存语义

当写一个volatile变量时，JMM会把该线程对应的**本地内存中的共享变量刷新到主内存**。

以happens-before中的代码为例：

假设线程A首先执行writer()方法，随后线程B执行reader()方法，初始时两个线程的本地内存中的flag和a都是初始状态。

下图是线程A执行volatile写后，共享变量的状态示意图：



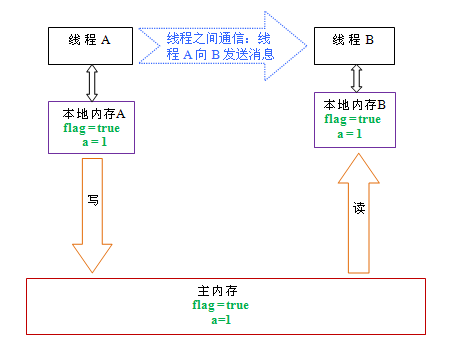
线程A在写flag变量后，本地内存A中被线程A更新过的两个共享变量的值被刷新到主内存中。

此时，本地内存A和主内存中的共享变量的值是一致的。

* + 1. 读内存语义

当读一个volatile变量时，JMM会把该线程对应的**本地内存（实现原理中说的缓存）**置为无效。

线程接下来将从主内存中读取共享变量。



* 1. volatile内存语义的实现

**重排序**分为**编译器重排序**和**处理器重排序**。

为了实现volatile内存语义，JMM会分别限制这两种类型的重排序类型。下面是JMM针对编译器制定的volatile重排序规则表：

|  |  |  |  |
| --- | --- | --- | --- |
| 是否重排序 | 第二个操作 | | |
| 第一个操作 | 普通读/写 | volatile读 | volatile写 |
| 普通读/写 |  |  | NO |
| volatile读 | NO | NO | NO |
| volatile写 |  | NO | NO |

举例来说，第三行最后一个单元格的意思是：在程序顺序中，当第一个操作为普通变量的读或写时，如果第二个操作为volatile写，则**编译器**不能重排序这两个操作。

从上表我们可以看出：

当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序。这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。

当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前。

当第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，JMM采取保守策略。下面是基于保守策略的JMM内存屏障插入策略：

在每个volatile写操作的前面插入一个StoreStore屏障。

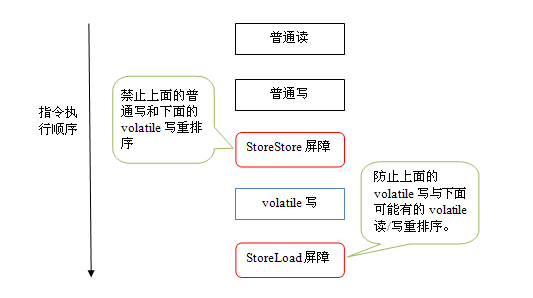
在每个volatile写操作的后面插入一个StoreLoad屏障。

在每个volatile读操作的后面插入一个LoadLoad屏障。

在每个volatile读操作的后面插入一个LoadStore屏障。

上述内存屏障插入策略非常保守，但它可以保证在任意处理器平台，任意的程序中都能得到正确的volatile内存语义。

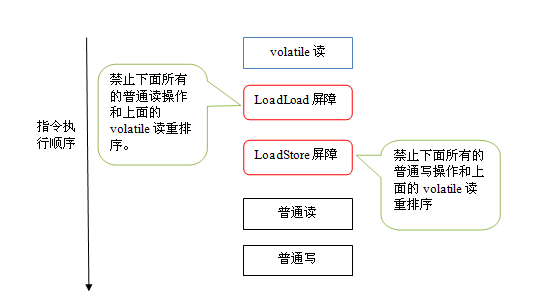
下面是保守策略下，volatile写插入内存屏障后生成的指令序列示意图：



上图中的StoreStore屏障可以保证在volatile写之前，其前面的所有普通写操作已经对任意处理器可见了。这是因为**StoreStore屏障将保障上面所有的普通写在volatile写之前刷新到主内存。**

这里比较有意思的是volatile写后面的StoreLoad屏障。这个屏障的作用是避免volatile写与后面可能有的volatile读/写操作重排序。因为编译器常常无法准确判断在一个volatile写的后面，是否需要插入一个StoreLoad屏障（比如，一个volatile写之后方法立即return）。为了保证能正确实现volatile的内存语义，JMM在这里采取了保守策略：在每个volatile写的后面或在每个volatile读的前面插入一个StoreLoad屏障。从整体执行效率的角度考虑，JMM选择了在每个volatile写的后面插入一个StoreLoad屏障。因为volatile写-读内存语义的常见使用模式是：一个写线程写volatile变量，多个读线程读同一个volatile变量。当读线程的数量大大超过写线程时，选择在volatile写之后插入StoreLoad屏障将带来可观的执行效率的提升。从这里我们可以看到JMM在实现上的一个特点：首先确保正确性，然后再去追求执行效率。

下面是在保守策略下，**volatile读**插入内存屏障后生成的指令序列示意图：



上图中的LoadLoad屏障用来禁止处理器把上面的volatile读与下面的普通读重排序。LoadStore屏障用来禁止处理器把上面的volatile读与下面的普通写重排序。

上述volatile写和volatile读的内存屏障插入策略非常保守。在实际执行时，只要不改变volatile写-读的内存语义，编译器可以根据具体情况省略不必要的屏障。下面我们通过具体的示例代码来说明：

class VolatileBarrierExample {

int a;

volatile int v1 = 1;

volatile int v2 = 2;

void readAndWrite() {

int i = v1; //第一个volatile读

int j = v2; // 第二个volatile读

a = i + j; //普通写

v1 = i + 1; // 第一个volatile写

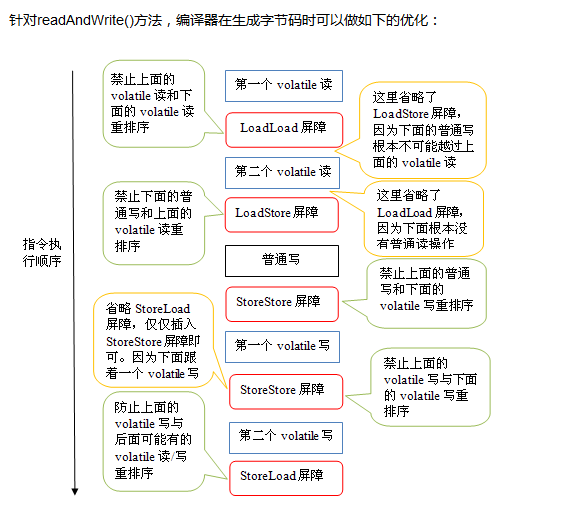
v2 = j \* 2; //第二个 volatile写

}

… //其他方法

}

针对readAndWrite()方法，编译器在生成字节码时可以做如下的优化：



注意，最后的StoreLoad屏障不能省略。因为第二个volatile写之后，方法立即return。此时编译器可能无法准确断定后面是否会有volatile读或写，为了安全起见，编译器常常会在这里插入一个StoreLoad屏障。

1. 静态变量、实例变量、局部变量与线程安全

**成员变量和类变量的区别：**

1、两个变量的生命周期不同

成员变量随着对象的创建而存在，随着对象的回收而释放。

静态变量随着类的加载而存在，随着类的消失而消失，且优先于对象存在。

2、调用方式不同

成员变量只能被对象调用。

静态变量可以被对象调用，还可以被类名调用。

3、数据存储位置不同

成员变量存储在堆内存的对象中，是对象的特有数据。

静态变量数据存储在方法区（共享数据区）的静态区

static还可以修饰方法，静态方法只能访问静态变量，不可以访问成员变量，因为静态方法加载时，优先于对象存在，所以没有办法访问对象中的成员。同时静态方法中不能使用this，super关键字，因为this代表当前对象，而静态方法在时，有可能没有对象，所以this无法使用。

* 1. 静态变量：线程不安全

使用static修饰的变量是类变量也叫静态变量，在整个类的生命周期中只有一个，且是所有的该类的实例所共有的。

静态变量在类加载的时候会进行初始化（可参加，jvm第7章类加载）。位于方法区，共享一份内存，一旦静态变量被修改，其他对象均会受到影响，故线程非安全。

* 1. 实例变量

成员变量（实例变量）：

1、成员变量定义在类中，即类中的普通变量，在整个类中都可以被类中方法所访问（如过和局部变量重名，需用this关键字）。

2、成员变量随着对象的建立而建立，随着对象的消失而消失，存在于对象所在的堆内存中。

3、成员变量有默认初始化值

* + 1. 单例：线程非安全

实例变量为对象实例私有，在虚拟机的堆中分配，若在系统中只存在一个此对象的实例，在多线程环境下，被某个线程修改后，其他线程对修改均可见，故线程非安全；

* + 1. 非单例：线程安全

如果每个线程执行都是在不同的对象中(非单例模式)，那对象与对象之间的实例变 量的修改将互不影响，故线程安全。

* 1. 局部变量：线程安全

**局部变量：线程安全：**

1、局部变量只定义在局部范围内，如：函数内，for循环语句内等，只在所属的区域有效。

2、局部变量存在于**栈内存（更准确的说是存在于虚拟机栈的栈帧中）**中（虚拟机栈是线程私有的），作用的范围结束，变量空间会自动释放。

3、局部变量没有默认初始化值

4、在使用变量时需要遵循的原则为：就近原则，先找局部变量，再找实例变量（如果同名，实例变量需要用this关键字引用）

5、局部变量不能逐级重名，比如函数内定义过一个变量，就不能在for循环内定义相同的变量（两层for循环一个用i一个用j也是这个道理）

**由于每个线程执行时将会把局部变量放在各自栈帧的工作内存中，线程间不共享，故不存在线程安全问题。**

1. 线程池
   1. 为什么使用线程池

**问题提出：**

我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一些问题：

* 如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。
* 线程需要占用内存，同时也会占用大量的GC；

**提出假设：**

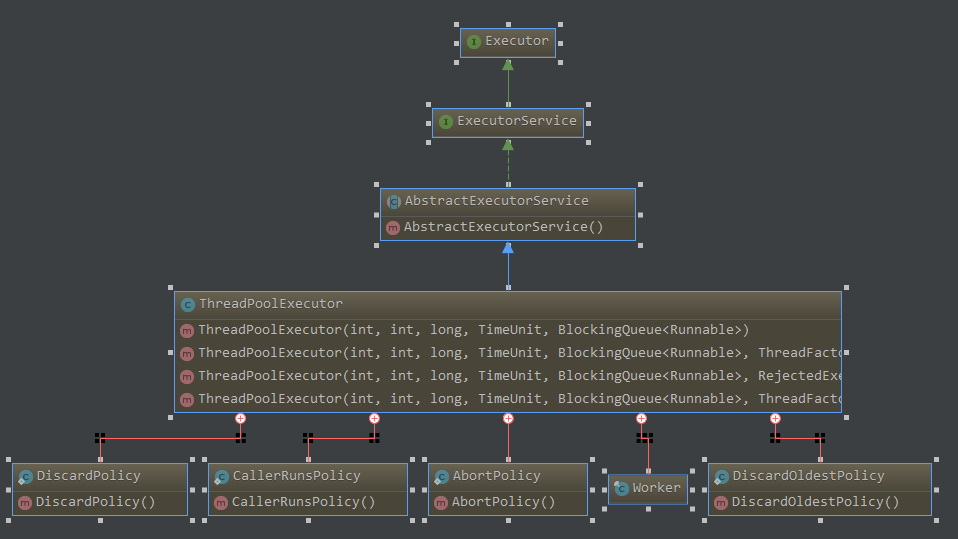
那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务？

**问题解决：**

在Java中可以通过**线程池**来达到这样的效果。

* 1. Java中的ThreadPoolExecutor类

* + 1. 类结构



* + - 1. AbstractExecutorService



public **abstract** class **AbstractExecutorService** implements **ExecutorService** {

protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) { };

protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) { };

public Future<?> submit(Runnable task) {};

public <T> Future<T> submit(Runnable task, T result) { };

public <T> Future<T> submit(Callable<T> task) { };

private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,

boolean timed, long nanos)

throws InterruptedException, ExecutionException, TimeoutException {

};

public <T> T invokeAny(Collection<? extends Callable<T>> tasks)

throws InterruptedException, ExecutionException {

};

public <T> T invokeAny(Collection<? extends Callable<T>> tasks,

long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException {

};

public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

throws InterruptedException {

};

public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,

long timeout, TimeUnit unit)

throws InterruptedException {

};

}

* + - * 1. ExecutorService



public **interface ExecutorService** extends **Executor** {

void shutdown();

boolean isShutdown();

boolean isTerminated();

boolean awaitTermination(long timeout, TimeUnit unit)throws InterruptedException;

<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException;

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)

throws InterruptedException;

<T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException;

<T> T invokeAny(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;

}

Executor



public interface Executor {  
  
 */\*\*  
 \* Executes the given command at some time in the future. The command  
 \* may execute in a new thread, in a pooled thread, or in the calling  
 \* thread, at the discretion of the <tt>Executor</tt> implementation.  
 \*  
 \** ***@param*** *command the runnable task  
 \** ***@throws*** *RejectedExecutionException if this task cannot be  
 \* accepted for execution.  
 \** ***@throws*** *NullPointerException if command is null  
 \*/* void execute(Runnable command);  
}

* + 1. 构造函数



构造器源码：从源码中可以看出，前边的三个构造器都调用了第四个构造器；

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,  
 long keepAliveTime,TimeUnit unit,  
 BlockingQueue<Runnable> workQueue) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 Executors.*defaultThreadFactory*(), *defaultHandler*);  
}  
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,  
 long keepAliveTime,TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 threadFactory, *defaultHandler*);  
}  
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,  
 long keepAliveTime,TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 RejectedExecutionHandler handler) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 Executors.*defaultThreadFactory*(), handler);  
}  
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,  
 long keepAliveTime,TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {  
 if (corePoolSize < 0 || maximumPoolSize <= 0 || maximumPoolSize < corePoolSize || keepAliveTime < 0)

throw new IllegalArgumentException();  
 if (workQueue == null || threadFactory == null || handler == null)  
 throw new NullPointerException();  
 this.corePoolSize = corePoolSize;  
 this.maximumPoolSize = maximumPoolSize;  
 this.workQueue = workQueue;  
 this.keepAliveTime = unit.toNanos(keepAliveTime);  
 this.threadFactory = threadFactory;  
 this.handler = handler;  
}

* + - 1. corePoolSize



**核心池的大小**，这个参数跟线程池的实现原理有非常大的关系。

1、在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。

2、默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到**达的任务**放到缓存队列当中；

实例：

* + - 1. maximumPoolSize



线程池最大线程数，它表示在线程池中最多能创建多少个线程;

如果队列中任务已满，并且当前线程个数小于maximumPoolSize，那么会创建新的线程来执行任务。这里值得一提的是largestPoolSize，该变量记录了线程池在整个生命周期中曾经出现的最大线程个数。为什么说是曾经呢？因为线程池创建之后，可以调用setMaximumPoolSize()改变运行的最大线程的数目。

* + - 1. keepAliveTime



表示线程没有任务执行时最多保持多久时间会终止。

1、默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize:

即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

* + - 1. unit



参数keepAliveTime的**时间单位**，有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

* + - 1. workQueue



一个**阻塞**队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的**排队策略**与BlockingQueue有关。

* + - 1. threadFactory



线程工厂，主要用来创建线程

* + - 1. handler



表示当拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

具体参数的配置与线程池的关系将在下一节讲述。

* + 1. 重要的方法



* + - 1. execute()



execute()方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

* + - 1. submit()



submit()方法是在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同：

它能够返回任务执行的结果，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果:

*/\*\*  
 \** ***@throws*** *RejectedExecutionException {****@inheritDoc****}  
 \** ***@throws*** *NullPointerException {****@inheritDoc****}  
 \*/*public <T> Future<T> submit(Callable<T> task) {  
 if (task == null)

throw new NullPointerException();  
 RunnableFuture<T> ftask = newTaskFor(task);  
 execute(ftask);  
 return ftask;  
}

*/\*\*  
 \* Returns a <tt>RunnableFuture</tt> for the given callable task.  
 \*  
 \** ***@param*** *callable the callable task being wrapped  
 \** ***@return*** *a <tt>RunnableFuture</tt> which when run will call the  
 \* underlying callable and which, as a <tt>Future</tt>, will yield  
 \* the callable's result as its result and provide for  
 \* cancellation of the underlying task.  
 \** ***@since*** *1.6  
 \*/*protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {  
 return new FutureTask<T>(callable);  
}

* + - 1. shutdown()/shutdownNow()



shutdown()和shutdownNow()是用来关闭线程池的：

*/\*\*  
 \* Initiates an orderly shutdown in which previously submitted  
 \* tasks are executed, but no new tasks will be accepted.  
 \* Invocation has no additional effect if already shut down.  
 \*  
 \* <p>This method does not wait for previously submitted tasks to  
 \* complete execution. Use {****@link*** *#awaitTermination awaitTermination}  
 \* to do that.  
 \*  
 \** ***@throws*** *SecurityException {****@inheritDoc****}  
 \*/*public void shutdown() {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();  
 advanceRunState(*SHUTDOWN*);  
 interruptIdleWorkers();  
 onShutdown(); // hook for ScheduledThreadPoolExecutor  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();  
}

* + 1. 重要的成员变量



private final BlockingQueue<Runnable> workQueue; //任务缓存队列，用来存放等待执行的任务

private final ReentrantLock mainLock = new ReentrantLock(); //线程池的主要状态锁，对线程池状态（比如线程池大小

//、runState等）的改变都要使用这个锁

**private final** HashSet<Worker> workers = new HashSet<Worker>(); //用来存放工作集

private volatile long keepAliveTime; //线程存货时间

private volatile boolean allowCoreThreadTimeOut; //是否允许为核心线程设置存活时间

**private volatile** int corePoolSize; //核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）

private volatile int maximumPoolSize; //线程池最大能容忍的线程数

private volatile int poolSize; //线程池中当前的线程数

private volatile RejectedExecutionHandler handler; //任务拒绝策略

private volatile ThreadFactory threadFactory; //线程工厂，用来创建线程

private int largestPoolSize; //用来记录线程池中曾经出现过的最大线程数

private long completedTaskCount; //用来记录已经执行完毕的任务个数

**举个简单的例子：**

假如有一个工厂，工厂里面有10个工人，每个工人同时只能做一件任务。

因此只要当10个工人中有工人是空闲的，来了任务就分配给空闲的工人做；

当10个工人都有任务在做时，如果还来了任务，就把任务进行排队等待；

如果说新任务数目增长的速度远远大于工人做任务的速度，那么此时**工厂主管**可能会想补救措施，比如重新招4个临时工人进来；

然后就将任务也分配给这4个临时工人做；

如果说着14个工人做任务的速度还是不够，此时工厂主管可能就要考虑不再接收新的任务或者抛弃前面的一些任务了。

当这14个工人当中有人空闲时，而新任务largestPoolSize只是一个用来起记录作用的变量，用来记录线程池中曾经有过的最大线程数目，跟线程池的容量没有任何关系。增长的速度又比较缓慢，工厂主管可能就考虑辞掉4个临时工了，只保持原来的10个工人，毕竟请额外的工人是要花钱的。

**这个例子中的corePoolSize就是10，而maximumPoolSize就是14（10+4）。**

也就是说corePoolSize就是线程池大小，maximumPoolSize在我看来是线程池的一种补救措施，即任务量突然过大时的一种补救措施。

不过为了方便理解，在本文后面还是将corePoolSize翻译成核心池大小。

* 1. 深入剖析线程池实现原理
     1. 线程池状态



在ThreadPoolExecutor中定义了一个**volatile**变量，另外定义了几个**static final**变量表示线程池的各个状态：

volatile int runState;

static final int RUNNING = 0;

static final int SHUTDOWN = 1;

static final int STOP = 2;

static final int TERMINATED = 3;

1、runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

2、下面的几个static final变量表示runState可能的几个取值。

当创建线程池后，初始时，线程池处于RUNNING状态；

如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

* + 1. 任务的执行



在ThreadPoolExecutor类中，最核心的任务提交方法是execute()方法，虽然通过submit也可以提交任务，但是实际上submit方法里面最终调用的还是execute()方法，所以我们只需要研究execute()方法的实现原理即可：

*/\*\*  
 \* 在将来的某个时间来执行给定的任务. 这个任务可能会被一个新的线程执行，也有可能被一个线程池中的线程执行.如果这个任务由于执行器已经停止，或者执行器的容量已经达到了上线而没有能够被提交处理，那么这个任务将由当前的RejectedExecutionHandler来处理  
 \** ***@param*** *command 要被执行的任务  
 \** ***@throws*** *RejectedExecutionException at discretion of  
 \* {****@code*** *RejectedExecutionHandler}, if the task  
 \* cannot be accepted for execution  
 \** ***@throws*** *NullPointerException if {****@code*** *command} is null  
 \*/*public void execute(Runnable command) {  
 if (command == null)  
 throw new NullPointerException();  
 /\*  
 \* Proceed in 3 steps:  
 \*  
 \* 1. If fewer than corePoolSize threads are running, try to  
 \* start a new thread with the given command as its first  
 \* task. The call to addWorker atomically checks runState and  
 \* workerCount, and so prevents false alarms that would add  
 \* threads when it shouldn't, by returning false.  
 \*  
 \* 2. If a task can be successfully queued, then we still need  
 \* to double-check whether we should have added a thread  
 \* (because existing ones died since last checking) or that  
 \* the pool shut down since entry into this method. So we  
 \* recheck state and if necessary roll back the enqueuing if  
 \* stopped, or start a new thread if there are none.  
 \*  
 \* 3. If we cannot queue task, then we try to add a new  
 \* thread. If it fails, we know we are shut down or saturated  
 \* and so reject the task.  
 \*/  
 int c = ctl.get();  
 if (*workerCountOf*(c) < corePoolSize) {  
 if (addWorker(command, true))  
 return;  
 c = ctl.get();  
 }  
 if (*isRunning*(c) && workQueue.offer(command)) {  
 int recheck = ctl.get();  
 if (! *isRunning*(recheck) && remove(command))  
 reject(command);  
 else if (*workerCountOf*(recheck) == 0)  
 addWorker(null, false);  
 }  
 else if (!addWorker(command, false))  
 reject(command);  
}

* + - 1. execute()



* + - * 1. step1：判断任务是否为空



首先，判断提交的任务command是否为null，若是null，则抛出空指针异常；

* + - * 1. step2：判断当前线程数是否达到corePoolSize



int c = ctl.get();  
 if (*workerCountOf*(c) < corePoolSize) {  
 if (addWorker(command, true))  
 return;  
 c = ctl.get();  
 }

如果当前运行的线程数少于corePoolSize，尝试启动一个新的线程，并且以当前的任务作为这个线程的第一个任务，对addWorker 的调用会自动的检查线程池的运行状态以及当线程数量。从而避免当线程池停止的情况下添加新的线程而引起的警告，如果添加新的线程成功则直接返回（即当前线程池中的线程数还没有达到corePoolSize）

*/\*\*  
 \* Checks if a new worker can be added with respect to current  
 \* pool state and the given bound (either core or maximum). If so,  
 \* the worker count is adjusted accordingly, and, if possible, a  
 \* new worker is created and started, running firstTask as its  
 \* first task. This method returns false if the pool is stopped or  
 \* eligible to shut down. It also returns false if the thread  
 \* factory fails to create a thread when asked. If the thread  
 \* creation fails, either due to the thread factory returning  
 \* null, or due to an exception (typically OutOfMemoryError in  
 \* Thread#start), we roll back cleanly.  
 \*  
 \** ***@param*** *firstTask the task the new thread should run first (or  
 \* null if none). Workers are created with an initial first task  
 \* (in method execute()) to bypass queuing when there are fewer  
 \* than corePoolSize threads (in which case we always start one),  
 \* or when the queue is full (in which case we must bypass queue).  
 \* Initially idle threads are usually created via  
 \* prestartCoreThread or to replace other dying workers.  
 \*  
 \** ***@param*** *core if true use corePoolSize as bound, else  
 \* maximumPoolSize. (A boolean indicator is used here rather than a  
 \* value to ensure reads of fresh values after checking other pool  
 \* state).  
 \** ***@return*** *true if successful  
 \*/*private boolean addWorker(Runnable firstTask, boolean core) {  
 retry:  
 for (;;) {  
 int c = ctl.get();  
 int rs = *runStateOf*(c);  
  
 // Check if queue empty only if necessary.  
 if (rs >= *SHUTDOWN* &&  
 ! (rs == *SHUTDOWN* &&  
 firstTask == null &&  
 ! workQueue.isEmpty()))  
 return false;  
  
 for (;;) {  
 int wc = *workerCountOf*(c);  
 if (wc >= *CAPACITY* ||  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 return false;  
 if (compareAndIncrementWorkerCount(c))  
 break retry;  
 c = ctl.get(); // Re-read ctl  
 if (*runStateOf*(c) != rs)  
 continue retry;  
 // else CAS failed due to workerCount change; retry inner loop  
 }  
 }

* + - * 1. step3：判断是否可以向队列中添加任务



if (*isRunning*(c) && workQueue.offer(command)) {  
 int recheck = ctl.get();  
 if (! *isRunning*(recheck) && remove(command))  
 reject(command);  
 else if (*workerCountOf*(recheck) == 0)  
 addWorker(null, false);  
 }

如果当前线程池正在运行，并且线程池的任务队列能够放得下新的任务，接着判断是不是需要添加新的线程（此时线程池中的线程数已经达到corePoolSize），这句判断是为了防止在将此任务添加进任务缓存队列的同时其他线程突然调用shutdown或者shutdownNow方法关闭了线程池的一种应急措施。如果是这样就执行

* + - * 1. step4：不能向队列中添加新的任务



else if (!addWorker(command, false))  
 reject(command);

如果不能向任务队列中添加新的任务，我们将尝试添加一个新的线程（直到达到maximumPollSize）。如果添加成功，新的线程将会执行这个任务，如果添加失败（1、线城市关闭；2、线程池达到最大），则拒绝任务

* + - 1. Worker类的实现



* + - 1. 线程池处理任务策略



1、如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

2、如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中：

若添加成功，则该任务会等待空闲线程将其取出去执行；

若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

3、如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

4、如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

* + 1. 线程池中的线程初始化



默认情况下，创建线程池之后，线程池中是没有线程的，**需要提交任务之后才会创建线程**。

1、在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

prestartCoreThread()：初始化一个核心线程；

prestartAllCoreThreads()：初始化所有核心线程

下面是这2个方法的实现：

*/\*\*  
 \* Starts a core thread, causing it to idly wait for work. This  
 \* overrides the default policy of starting core threads only when  
 \* new tasks are executed. This method will return {****@code*** *false}  
 \* if all core threads have already been started.  
 \*  
 \** ***@return*** *{****@code*** *true} if a thread was started  
 \*/*public boolean prestartCoreThread() {  
 return *workerCountOf*(ctl.get()) < corePoolSize &&  
 addWorker(null, true);  
}

*/\*\*  
 \* Starts all core threads, causing them to idly wait for work. This  
 \* overrides the default policy of starting core threads only when  
 \* new tasks are executed.  
 \*  
 \** ***@return*** *the number of threads started  
 \*/*public int prestartAllCoreThreads() {  
 int n = 0;  
 while (addWorker(null, true))  
 ++n;  
 return n;  
}

注意上面传进去的参数是null，如果传进去的参数为null，则最后执行线程会阻塞在getTask方法中的

r = workQueue.take();

即等待任务队列中有任务:

*/\*\*  
 \** ***Performs blocking or timed wait for a task,*** *depending on  
 \* current configuration settings, or returns null if this worker  
 \* must exit because of any of:  
 \* 1. There are more than maximumPoolSize workers (due to  
 \* a call to setMaximumPoolSize).  
 \* 2. The pool is stopped.  
 \* 3. The pool is shutdown and the queue is empty.  
 \* 4. This worker timed out waiting for a task, and timed-out  
 \* workers are subject to termination (that is,  
 \* {****@code*** *allowCoreThreadTimeOut || workerCount > corePoolSize})  
 \* both before and after the timed wait.  
 \*  
 \** ***@return*** *task, or null if the worker must exit, in which case  
 \* workerCount is decremented  
 \*/*private Runnable getTask() {  
 boolean timedOut = false; // Did the last poll() time out?  
  
 retry:  
 for (;;) {  
 int c = ctl.get();  
 int rs = *runStateOf*(c);  
  
 // Check if queue empty only if necessary.  
 if (rs >= *SHUTDOWN* && (rs >= *STOP* || workQueue.isEmpty())) {  
 decrementWorkerCount();  
 return null;  
 }  
  
 boolean timed; // Are workers subject to culling?  
  
 for (;;) {  
 int wc = *workerCountOf*(c);  
 timed = allowCoreThreadTimeOut || wc > corePoolSize;  
  
 if (wc <= maximumPoolSize && ! (timedOut && timed))  
 break;  
 if (compareAndDecrementWorkerCount(c))  
 return null;  
 c = ctl.get(); // Re-read ctl  
 if (*runStateOf*(c) != rs)  
 continue retry;  
 // else CAS failed due to workerCount change; retry inner loop  
 }  
  
 try {  
 Runnable r = timed ?  
 workQueue.poll(keepAliveTime, TimeUnit.*NANOSECONDS*) :  
 workQueue.take();  
 if (r != null)  
 return r;  
 timedOut = true;  
 } catch (InterruptedException retry) {  
 timedOut = false;  
 }  
 }  
}

* + 1. 任务缓存队列及排队策略



在前面我们多次提到了**任务缓存队列**，即workQueue，它用来存放等待执行的任务。

workQueue的类型为BlockingQueue<Runnable>，通常可以取下面三种类型：

1）ArrayBlockingQueue：**基于数组**的先进先出队列，此队列创建时必须指定大小；

2）LinkedBlockingQueue：**基于链表**的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

3）synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

*/\*\*  
 \* The queue used for holding tasks and handing off to worker  
 \* threads. We do not require that workQueue.poll() returning  
 \* null necessarily means that workQueue.isEmpty(), so rely  
 \* solely on isEmpty to see if the queue is empty (which we must  
 \* do for example when deciding whether to transition from  
 \* SHUTDOWN to TIDYING). This accommodates special-purpose  
 \* queues such as DelayQueues for which poll() is allowed to  
 \* return null even if it may later return non-null when delays  
 \* expire.  
 \*/*private final BlockingQueue<Runnable> workQueue;

* + 1. 任务拒绝策略



当线程池的**任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize**，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

*/\*\*  
 \* A handler for rejected tasks that throws a  
 \* {****@code*** *RejectedExecutionException}.  
 \*/*public static class AbortPolicy implements RejectedExecutionHandler {  
 */\*\*  
 \* Creates an {****@code*** *AbortPolicy}.  
 \*/* public AbortPolicy() { }  
  
 */\*\*  
 \* Always throws RejectedExecutionException.  
 \*  
 \** ***@param*** *r the runnable task requested to be executed  
 \** ***@param*** *e the executor attempting to execute this task  
 \** ***@throws*** *RejectedExecutionException always.  
 \*/* public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 throw new RejectedExecutionException("Task " + r.toString() +  
 " rejected from " +  
 e.toString());  
 }  
}  
  
*/\*\*  
 \* A handler for rejected tasks that silently discards the  
 \* rejected task.  
 \*/*public static class DiscardPolicy implements RejectedExecutionHandler {  
 */\*\*  
 \* Creates a {****@code*** *DiscardPolicy}.  
 \*/* public DiscardPolicy() { }  
  
 */\*\*  
 \* Does nothing, which has the effect of discarding task r.  
 \*  
 \** ***@param*** *r the runnable task requested to be executed  
 \** ***@param*** *e the executor attempting to execute this task  
 \*/* public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 }  
}  
  
*/\*\*  
 \* A handler for rejected tasks that discards the oldest unhandled  
 \* request and then retries {****@code*** *execute}, unless the executor  
 \* is shut down, in which case the task is discarded.  
 \*/*public static class DiscardOldestPolicy implements RejectedExecutionHandler {  
 */\*\*  
 \* Creates a {****@code*** *DiscardOldestPolicy} for the given executor.  
 \*/* public DiscardOldestPolicy() { }  
  
 */\*\*  
 \* Obtains and ignores the next task that the executor  
 \* would otherwise execute, if one is immediately available,  
 \* and then retries execution of task r, unless the executor  
 \* is shut down, in which case task r is instead discarded.  
 \*  
 \** ***@param*** *r the runnable task requested to be executed  
 \** ***@param*** *e the executor attempting to execute this task  
 \*/* public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 if (!e.isShutdown()) {  
 e.getQueue().poll();  
 e.execute(r);  
 }  
 }  
}

* + 1. 线程池的关闭



ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

**shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务:**

*/\*\*  
 \* Initiates an orderly shutdown in which previously submitted  
 \* tasks are executed, but no new tasks will be accepted.  
 \* Invocation has no additional effect if already shut down.  
 \*  
 \* <p>This method does not wait for previously submitted tasks to  
 \* complete execution. Use {****@link*** *#awaitTermination awaitTermination}  
 \* to do that.  
 \*  
 \** ***@throws*** *SecurityException {****@inheritDoc****}  
 \*/*public void shutdown() {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();  
 advanceRunState(*SHUTDOWN*);  
 interruptIdleWorkers();  
 onShutdown(); // hook for ScheduledThreadPoolExecutor  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();  
}

**shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务：**

*/\*\*  
 \* Attempts to stop all actively executing tasks, halts the  
 \* processing of waiting tasks, and returns a list of the tasks  
 \* that were awaiting execution. These tasks are drained (removed)  
 \* from the task queue upon return from this method.  
 \*  
 \* <p>This method does not wait for actively executing tasks to  
 \* terminate. Use {****@link*** *#awaitTermination awaitTermination} to  
 \* do that.  
 \*  
 \* <p>There are no guarantees beyond best-effort attempts to stop  
 \* processing actively executing tasks. This implementation  
 \* cancels tasks via {****@link*** *Thread#interrupt}, so any task that  
 \* fails to respond to interrupts may never terminate.  
 \*  
 \** ***@throws*** *SecurityException {****@inheritDoc****}  
 \*/*public List<Runnable> shutdownNow() {  
 List<Runnable> tasks;  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();  
 advanceRunState(*STOP*);  
 interruptWorkers();  
 tasks = drainQueue();  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();  
 return tasks;  
}

* + 1. 线程池容量的动态调整



ThreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()：

**setCorePoolSize：设置核心池大小：**

*/\*\*  
 \* Sets the core number of threads. This overrides any value set  
 \* in the constructor. If the new value is smaller than the  
 \* current value, excess existing threads will be terminated when  
 \* they next become idle. If larger, new threads will, if needed,  
 \* be started to execute any queued tasks.  
 \*  
 \** ***@param*** *corePoolSize the new core size  
 \** ***@throws*** *IllegalArgumentException if {****@code*** *corePoolSize < 0}  
 \** ***@see*** *#getCorePoolSize  
 \*/*public void setCorePoolSize(int corePoolSize) {  
 if (corePoolSize < 0)  
 throw new IllegalArgumentException();  
 int delta = corePoolSize - this.corePoolSize;  
 this.corePoolSize = corePoolSize;  
 if (*workerCountOf*(ctl.get()) > corePoolSize)  
 interruptIdleWorkers();  
 else if (delta > 0) {  
 // We don't really know how many new threads are "needed".  
 // As a heuristic, prestart enough new workers (up to new  
 // core size) to handle the current number of tasks in  
 // queue, but stop if queue becomes empty while doing so.  
 int k = Math.*min*(delta, workQueue.size());  
 while (k-- > 0 && addWorker(null, true)) {  
 if (workQueue.isEmpty())  
 break;  
 }  
 }  
}

**setMaximumPoolSize：设置线程池最大能创建的线程数目大小：**

*/\*\*  
 \* Sets the maximum allowed number of threads. This overrides any  
 \* value set in the constructor. If the new value is smaller than  
 \* the current value, excess existing threads will be  
 \* terminated when they next become idle.  
 \*  
 \** ***@param*** *maximumPoolSize the new maximum  
 \** ***@throws*** *IllegalArgumentException if the new maximum is  
 \* less than or equal to zero, or  
 \* less than the {****@linkplain*** *#getCorePoolSize core pool size}  
 \** ***@see*** *#getMaximumPoolSize  
 \*/*public void setMaximumPoolSize(int maximumPoolSize) {  
 if (maximumPoolSize <= 0 || maximumPoolSize < corePoolSize)  
 throw new IllegalArgumentException();  
 this.maximumPoolSize = maximumPoolSize;  
 if (*workerCountOf*(ctl.get()) > maximumPoolSize)  
 interruptIdleWorkers();  
}

当上述参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。

* 1. 使用示例

源码：

public class Test {

public static void main(String[] args) {

ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 200, TimeUnit.MILLISECONDS, new ArrayBlockingQueue<Runnable>(5));

for(int i=0;i<15;i++){

MyTask myTask = new MyTask(i);

executor.execute(myTask);

System.out.println("线程池中线程数目："+executor.getPoolSize()+"，队列中等待执行的任务数目："+

executor.getQueue().size()+"，已执行玩的任务数目："+executor.getCompletedTaskCount());

}

executor.shutdown();

}

}

class MyTask implements Runnable {

private int taskNum;

public MyTask(int num) {

this.taskNum = num;

}

@Override

public void run() {

System.out.println("正在执行task "+taskNum);

try {

Thread.currentThread().sleep(4000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("task "+taskNum+"执行完毕");

}

}

结果分析：

结果：

正在执行task 0

线程池中线程数目：1，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

线程池中线程数目：2，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 1

线程池中线程数目：3，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 2

线程池中线程数目：4，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 3

线程池中线程数目：5，队列中等待执行的任务数目：0，已执行玩别的任务数目：0

正在执行task 4

线程池中线程数目：5，队列中等待执行的任务数目：1，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：2，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：3，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：4，已执行玩别的任务数目：0

线程池中线程数目：5，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

线程池中线程数目：6，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 10

线程池中线程数目：7，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 11

线程池中线程数目：8，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 12

线程池中线程数目：9，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 13

线程池中线程数目：10，队列中等待执行的任务数目：5，已执行玩别的任务数目：0

正在执行task 14

task 3执行完毕

task 0执行完毕

task 2执行完毕

task 1执行完毕

正在执行task 8

正在执行task 7

正在执行task 6

正在执行task 5

task 4执行完毕

task 10执行完毕

task 11执行完毕

task 13执行完毕

task 12执行完毕

正在执行task 9

task 14执行完毕

task 8执行完毕

task 5执行完毕

task 7执行完毕

task 6执行完毕

task 9执行完毕

* 1. 如何合理配置线程池的大小

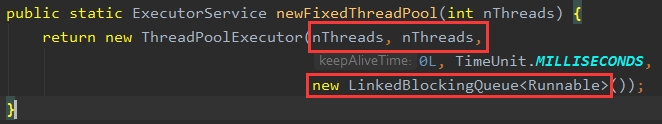
本节来讨论一个比较重要的话题：如何合理配置线程池大小：

一般需要根据任务的类型来配置线程池大小：

* 如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 NCPU+1
* 如果是IO密集型任务，参考值可以设置为2\*NCPU

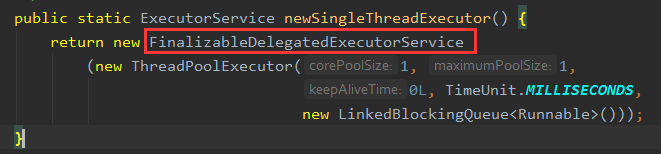
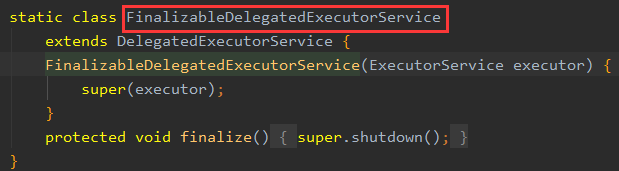
当然，这只是一个参考值，具体的设置还需要根据实际情况进行调整，比如可以先将线程池大小设置为参考值，再观察任务运行情况和系统负载、资源利用率来进行适当调整。

* 1. 几种线程池
     1. FixedThreadPool



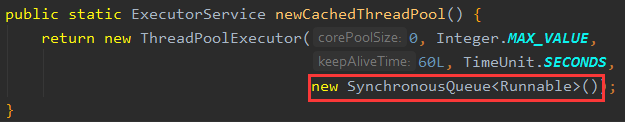
创建**线程数量固定的线程池**，线程池的corePoolSize和maximumPoolSize大小一样，并且keepAliveTime为0，传入的队列LinkedBlockingQueue为无界队列

* + 1. SingleThreadExecutor



corePoolSize和maximumPoolSize都是1，keepAliveTime是0L, 传入的队列是无界队列。**线程池中永远只要一个线程在工作。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**

* + 1. newCachedThreadPool



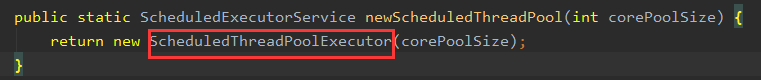
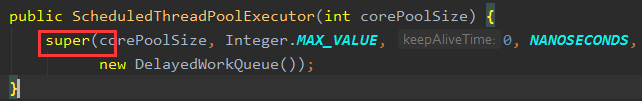
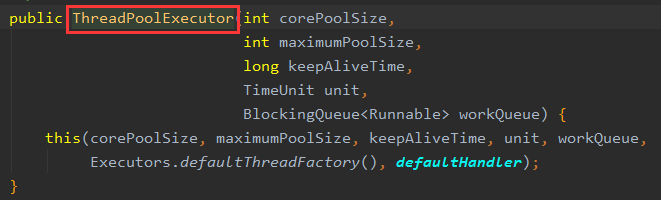
在newCachedThreadPool中如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程

初看该构造函数时我有这样的疑惑：核心线程池为0，那按照前面所讲的线程池策略新任务来临时无法进入核心线程池，只能进入 SynchronousQueue中进行等待，而SynchronousQueue的大小为1，那岂不是第一个任务到达时只能等待在队列中，直到第二个任务到达发现无法进入队列才能创建第一个线程？

这个问题的答案在SynchronousQueue已经给出了，要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接收这个元素。因此即便SynchronousQueue一开始为空且大小为1，第一个任务也无法放入其中，因为没有线程在等待从SynchronousQueue中取走元素。因此第一个任务到达时便会创建一个新线程执行该任务。

这里引申出一个小技巧：有时我们可能希望线程池在没有任务的情况下销毁所有的线程，既设置线程池核心大小为0，但又不想使用SynchronousQueue而是想使用有界的等待队列。显然，不进行任何特殊设置的话这样的用法会发生奇怪的行为：直到等待队列被填满才会有新线程被创建，任务才开始执行。这并不是我们希望看到的，此时可通过allowCoreThreadTimeOut使等待队列中的元素出队被调用执行，详细原理和使用将会在后续博客中阐述

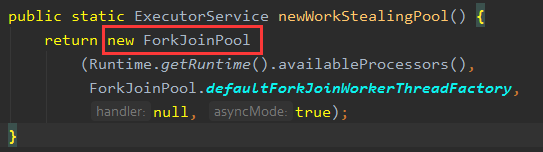
* + 1. newScheduledThreadPool



创建一个定长线程池，支持定时及周期性任务执行。

* + 1. newWorkStealingPool

创建一个拥有多个任务队列（以便减少连接数）的线程池。



* + - 1. ForkJoinPool

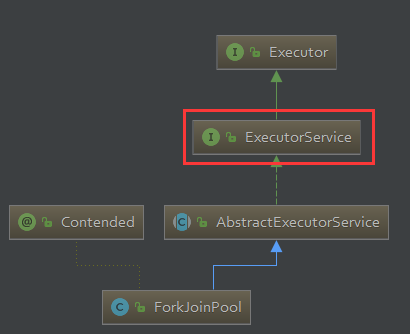


背景：ForkJoinPool的优势在于：

可以充分利用多cpu，多核cpu的优势，把一个任务拆分成多个“小任务”，把多个“小任务”放到多个处理器核心上并行执行；

当多个“小任务”执行完成之后，再将这些执行结果合并起来即可。

**ForkJoinPool是ExecutorService的实现类**，因此是一种特殊的线程池:

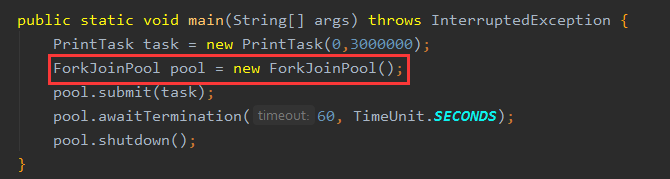


* + - * 1. 使用

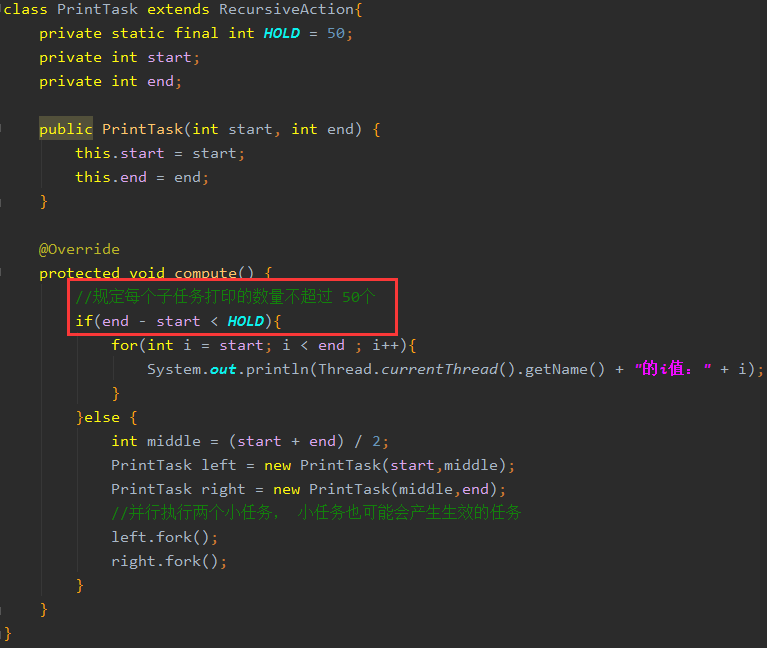
创建了ForkJoinPool实例之后，就可以调用ForkJoinPool的**submit**(ForkJoinTask<T> task) 或**invoke**(ForkJoinTask<T> task)方法来执行指定任务了。

其中ForkJoinTask代表一个可以并行、合并的任务。ForkJoinTask是一个抽象类，它还有两个抽象子类：RecusiveAction和RecusiveTask。其中**RecusiveTask**代表有返回值的任务，而**RecusiveAction**代表没有返回值的任务。

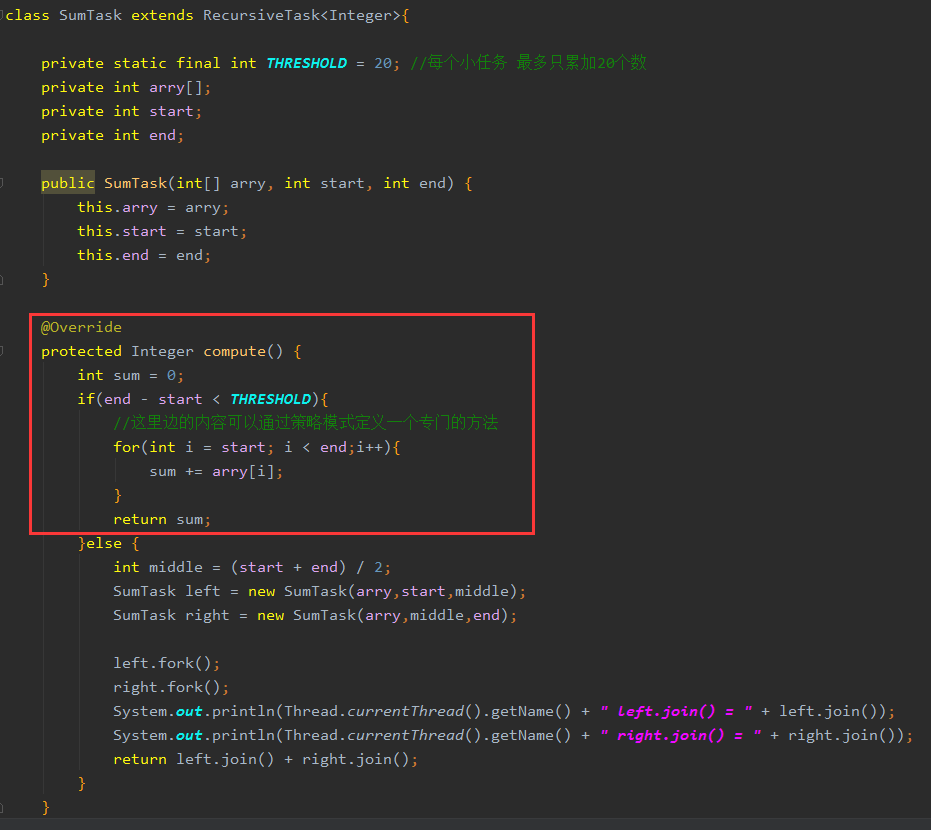
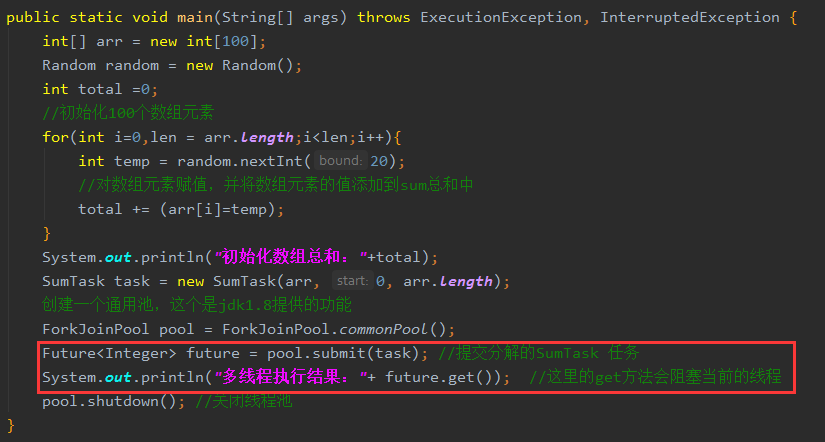
举例（没有返回值的）



任务：



举例（有返回值的）



* + - * 1. 分析

它同ThreadPoolExecutor一样，也实现了Executor和ExecutorService接口。它使用了一个**无限队列**来保存需要执行的任务，而线程的数量则是通过构造函数传入，如果没有向构造函数中传入希望的线程数量，那么**当前计算机可用的CPU数量会被设置为线程数量的默认值**。

ForkJoinPool主要用来使用分治法(Divide-and-Conquer Algorithm)来解决问题。典型的应用比如快速排序算法。

这里的要点在于，ForkJoinPool需要使用相对少的线程来处理大量的任务。

比如要对1000万个数据进行排序，那么会将这个任务分割成两个500万的排序任务和一个针对这两组500万数据的合并任务。以此类推，对于500万的数据也会做出同样的分割处理，到最后会设置一个阈值来规定当数据规模到多少时，停止这样的分割处理。比如，当元素的数量小于10时，会停止分割，转而使用插入排序对它们进行排序。

那么到最后，所有的任务加起来会有大概2000000+个。问题的关键在于，对于一个任务而言，只有当它所有的子任务完成之后，它才能够被执行。

所以当使用ThreadPoolExecutor时，使用分治法会存在问题，因为ThreadPoolExecutor中的线程无法像任务队列中再添加一个任务并且在等待该任务完成之后再继续执行。而使用ForkJoinPool时，就能够让其中的线程创建新的任务，并挂起当前的任务，此时线程就能够从队列中选择子任务执行。

以上程序的关键是fork()和join()方法。在ForkJoinPool使用的线程中，会使用一个内部队列来对需要执行的任务以及子任务进行操作来保证它们的执行顺序。

那么使用ThreadPoolExecutor或者ForkJoinPool，会有什么性能的差异呢？

首先，**使用ForkJoinPool能够使用数量有限的线程来完成非常多的具有父子关系的任务**，比如使用4个线程来完成超过200万个任务。但是，使用ThreadPoolExecutor时，是不可能完成的，因为ThreadPoolExecutor中的Thread无法选择优先执行子任务，需要完成200万个具有父子关系的任务时，也需要200万个线程，显然这是不可行的。

1. 多线程中的单例实现
   1. 饱汉模式

1、饱汉，即已经吃饱，不着急再吃，饿的时候再吃。所以他就先不初始化单例，等第一次使用的时候再初始化，即“**懒加载**”

2、饱汉模式的核心就是懒加载。

好处是:更启动速度快、节省资源，一直到实例被第一次访问，才需要初始化单例；

小坏处是写起来麻烦;

大坏处是线程不安全，if语句存在竞态条件。

* + 1. 基础版

public class Singleton1 {

private static Singleton1 singleton = null;

private Singleton1() {

}

public static Singleton1 getInstance() {

if (singleton == null) {

singleton = new Singleton1();

}

return singleton;

}

}

单线程环境下，基础饱汉是没有问题的。但多线程环境下，基础饱汉就彻底不可用了，if语句存在竞态；

**存在的问题：多线程下不安全**

* + 1. 饱汉 – 变种 1

针对基础版的问题：

最粗暴的办法是用**synchronized**关键字修饰getInstance()方法，这样能达到**绝对**的线程安全。

// 饱汉

// ThreadSafe

public class Singleton1\_1 {

private static Singleton1\_1 singleton = null;

private Singleton1\_1() {

}

public **synchronized** static Singleton1\_1 getInstance() {

if (singleton == null) {

singleton = new Singleton1\_1();

}

return singleton;

}

}

好处是：写起来简单，且绝对线程安全；

坏处是：并发性能极差，事实上完全退化到了串行。单例只需要初始化一次，但就算初始化以后，synchronized的锁也无法避开，从而getInstance()完全变成了串行操作。

* + 1. 饱汉 – 变种 2

针对变种1中单例初始化后**锁仍然无法避开**的问题：

变种2在变种1的外层又套了一层check，加上synchronized内层的check，即所谓“**双重检查锁**”（Double Check Lock，简称DCL）。

// 饱汉

// UnThreadSafe

public class Singleton1\_2 {

private static Singleton1\_2 singleton = null;

private Singleton1\_2() {

}

public static Singleton1\_2 getInstance() {

// may get half object

**if (singleton == null) {**

**synchronized (Singleton1\_2.class) {**

**if (singleton == null) {**

**singleton = new Singleton1\_2();**

**}**

**}**

**}**

return singleton;

}

}

变种2的核心是DCL，看起来变种2似乎已经达到了理想的效果：懒加载+线程安全。可惜的是，正如注释中所说，DCL仍然是线程不安全的，由于**指令重排序**，你可能会得到“半个对象”

* + 1. 饱汉 – 变种 3

针对变种3的**“半个对象”**问题，变种3在instance上增加了**volatile**关键字

// 饱汉

// ThreadSafe

public class Singleton1\_3 {

private static **volatile** Singleton1\_3 singleton = null;

private Singleton1\_3() {

}

public static Singleton1\_3 getInstance() {

if (singleton == null) {

synchronized (Singleton1\_3.class) {

// must be a complete instance

if (singleton == null) {

singleton = new Singleton1\_3();

}

}

}

return singleton;

}

}

多线程环境下，变种3更适用于性能敏感的场景。

* 1. 饿汉模式

饿汉的好处是：**天生的线程安全**（得益于类加载机制），写起来超级简单，使用时没有延迟；

坏处是：有可能造成资源浪费（如果类加载后就一直不使用单例的话）

// 饿汉

// ThreadSafe

public class Singleton2 {

private static final Singleton2 singleton = new Singleton2();

private Singleton2() {

}

public static Singleton2 getInstance() {

return singleton;

}

}

* 1. Holder模式

饿汉模式中静态变量的方便和线程安全 + 懒加载规避资源浪费：

Holder模式满足了这两点要求：

核心仍然是静态变量：足够方便和线程安全；

通过静态的Holder类持有真正实例，间接实现了懒加载。

// Holder模式

// ThreadSafe

public class Singleton3 {

private **static class** SingletonHolder {

private static final Singleton3 singleton = new Singleton3();

private SingletonHolder() {

}

}

private Singleton3() {

}

public synchronized static Singleton3 getInstance() {

return SingletonHolder.singleton;

}

}

1. 线程的ThreadLocal
   1. ThreadLocal的理解

ThreadLocal，很多地方叫做**线程本地变量**，也有些地方叫做**线程本地存储**，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

* + 1. 示例

**class ConnectionManager {**

**private static Connection connect = null;**

**public static Connection openConnection() {**

**if(connect == null){**

**connect = DriverManager.getConnection();**

**}**

**return connect;**

**}**

**public static void closeConnection() {**

**if(connect!=null)**

**connect.close();**

**}**

**}**

**懒汉单例模式**

* + 1. 示例解析

有这样一个数据库链接管理类，这段代码在单线程中使用是没有任何问题的，但是如果在多线程中使用呢？

很显然，在多线程中使用会存在线程安全问题：

第一，这里面的2个方法都没有进行同步，很可能在openConnection方法中会多次创建connect；

第二，由于connect是共享变量，那么必然在调用connect的地方需要使用到同步来保障线程安全，因为很可能一个线程在使用connect进行数据库操作，而另外一个线程调用closeConnection关闭链接。

所以出于线程安全的考虑，必须将这段代码的两个方法进行同步处理，并且在调用connect的地方需要进行同步处理。

这样将会大大影响程序执行效率，因为一个线程在使用connect进行数据库操作的时候，其他线程只有等待。

**数据库连接需不需要共享？**

* + 1. 数据库连接不需要共享

**数据库连接需不需要共享？**

事实上，是不需要的。假如每个线程中都有一个connect变量，各个线程之间对connect变量的访问实际上是没有依赖关系的，即一个线程不需要关心其他线程是否对这个connect进行了修改的。

既然不需要在线程之间共享这个变量，可以直接这样处理：在每个需要使用数据库连接的方法中具体使用时才创建数据库链接，然后在方法调用完毕再释放这个连接。

比如下面这样：

class ConnectionManager {

private Connection connect = null;

public Connection **openConnection**() {

if(connect == null){

connect = DriverManager.getConnection();

}

return connect;

}

public void closeConnection() {

if(connect!=null)

connect.close();

}

}

class Dao{

public void insert() {

ConnectionManager connectionManager = new ConnectionManager();

Connection connection = connectionManager.openConnection();

//使用connection进行操作

connectionManager.closeConnection();

}

}

这样处理确实也没有任何问题，由于每次都是在方法内部创建的连接，那么线程之间自然不存在线程安全问题。**但是这样会有一个致命的影响**：导致服务器压力非常大，并且严重影响程序执行性能。由于在方法中需要频繁地开启和关闭数据库连接，这样不仅严重影响程序执行效率，还可能导致服务器压力巨大。

那么这种情况下使用ThreadLocal是再适合不过的了，**因为ThreadLocal在每个线程中对该变量会创建一个副本**，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以**要考虑它对资源的消耗**，比如内存的占用会比不使用ThreadLocal要大。

* 1. ThreadLocal类
     1. 几个方法

**get()方法**是用来获取ThreadLocal在当前线程中保存的变量副本；

**set()**用来设置当前线程中变量的副本;

**remove()**用来移除当前线程中变量的副本；

**initialValue()**是一个protected方法，一般是用来在使用时进行重写的，它是一个延迟加载方法;

* + - 1. get()

*/\*\*  
 \* Returns the value in the current thread's copy of this  
 \* thread-local variable. If the variable has no value for the  
 \* current thread, it is first initialized to the value returned  
 \* by an invocation of the {****@link*** *#initialValue} method.  
 \*  
 \** ***@return*** *the current thread's value of this thread-local  
 \*/*public T get() {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 if (map != null) {  
 ThreadLocalMap.Entry e = map.getEntry(this);  
 if (e != null)  
 return (T)e.value;  
 }  
 return setInitialValue();  
}

第一句是取得当前线程;

然后通过getMap(t)方法获取到一个map，map的类型为ThreadLocalMap。

然后获取到<key,value>键值对，注意这里获取键值对传进去的是 this**(map以ThreadLocal作为键**)，而不是当前线程t。

如果获取成功，则返回value值。

如果map为空，则调用setInitialValue方法返回value

* + - * 1. getMap(t)

*/\*\*  
 \* Get the map associated with a ThreadLocal. Overridden in  
 \* InheritableThreadLocal.  
 \*  
 \** ***@param*** *t the current thread  
 \** ***@return*** *the map  
 \*/*ThreadLocalMap getMap(Thread t) {  
 return t.threadLocals;  
}

在getMap中，是调用当期线程t，返回当前线程t中的一个成员变量threadLocals。

t.threadLocals

t.threadLocals：

ThreadLocal.ThreadLocalMap threadLocals = null;

成员变量threadLocals实际上就是一个ThreadLocalMap，这个类型是ThreadLocal类的一个内部类.

static class ThreadLocalMap



ThreadLocalMap，这个类型是ThreadLocal类的一个内部类.

static class ThreadLocalMap {  
  
 */\*\*  
 \* The entries in this hash map extend WeakReference, using  
 \* its main ref field as the key (which is always a  
 \* ThreadLocal object). Note that null keys (i.e. entry.get()  
 \* == null) mean that the key is no longer referenced, so the  
 \* entry can be expunged from table. Such entries are referred to  
 \* as "stale entries" in the code that follows.  
 \*/* static class Entry extends WeakReference<ThreadLocal> {  
 */\*\* The value associated with this ThreadLocal. \*/* Object value;  
  
 Entry(ThreadLocal k, Object v) {  
 super(k);  
 value = v;  
 }  
 }

ThreadLocalMap的**Entry**继承了WeakReference，并且使用**ThreadLocal**作为键

* + - * 1. T setInitialValue()

*/\*\*  
 \* Variant of set() to establish initialValue. Used instead  
 \* of set() in case user has overridden the set() method.  
 \*  
 \** ***@return*** *the initial value  
 \*/*private T setInitialValue() {  
 T value = initialValue();  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 if (map != null)  
 map.set(this, value);  
 else  
 createMap(t, value);  
 return value;  
}

**很容易了解，就是如果map不为空，就设置键值对，为空，再创建Map，**

createMap(t, value);



*/\*\*  
 \* Create the map associated with a ThreadLocal. Overridden in  
 \* InheritableThreadLocal.  
 \*  
 \** ***@param*** *t the current thread  
 \** ***@param*** *firstValue value for the initial entry of the map  
 \** ***@param*** *map the map to store.  
 \*/*void createMap(Thread t, T firstValue) {  
 t.threadLocals = new ThreadLocalMap(this, firstValue);  
}

**ThreadLocal为每个线程创建变量的副本的步骤：**

**首先，**在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键为当前ThreadLocal变量，value为变量副本（即T类型的变量）。

**初始时，**在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键，以ThreadLocal要保存的副本变量为value，存到threadLocals。

**然后在**当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找。

* + - 1. set(T value)
      2. remove()
      3. T initialValue()
    1. 实例

public class Test {

ThreadLocal<Long> longLocal = new ThreadLocal<Long>();

ThreadLocal<String> stringLocal = new ThreadLocal<String>();

public void set() {

longLocal.set(Thread.currentThread().getId());

stringLocal.set(Thread.currentThread().getName());

}

public long getLong() {

return longLocal.get();

}

public String getString() {

return stringLocal.get();

}

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

final Test test = new Test();

test.set(); //将当前线程（**main线程**）的id和线程的名字存放到线程的ThreadLocal中

System.out.println(test.getLong());

System.out.println(test.getString());

Thread thread1 = new Thread(){

public void run() {

test.set(); **//每一个线程创建自己的副本**

System.out.println(test.getLong());

System.out.println(test.getString());

};

};

thread1.start();

thread1.join();

System.out.println(test.getLong());

System.out.println(test.getString());

}

}

结果：

1

main

10

Thread-0

1

main

1）实际的通过ThreadLocal创建的副本是存储在每个线程自己的threadLocals中的；

2）为何threadLocals的类型ThreadLocalMap的键值为ThreadLocal对象，因为每个线程中可有多个threadLocal变量，就像上面代码中的longLocal和stringLocal；

**3）在进行get之前，必须先set，否则会报空指针异常；**

如果想在get之前不需要调用set就能正常访问的话，必须重写initialValue()方法。

* 1. ThreadLocal的应用场景

最常见的ThreadLocal使用场ïº 用来解决 数据库连接、Session管理等：

1、数据库连接：

private static ThreadLocal<Connection> connectionHolder = new ThreadLocal<Connection>() {

public Connection initialValue() { **//重写initialValue方法，防止空指针异常**

return DriverManager.getConnection(DB\_URL);

}

};

public static Connection getConnection() {

return connectionHolder.get();

}

2、session管理

private static final ThreadLocal threadSession = new ThreadLocal();

public static Session getSession() throws InfrastructureException {

Session s = (Session) threadSession.get();

try {

if (s == null) {

s = getSessionFactory().openSession();

threadSession.set(s);

}

} catch (HibernateException ex) {

throw new InfrastructureException(ex);

}

return s;

}

**3、需要注意的是，虽然ThreadLocal也是用来解决线程安全问题的，但是它跟同步在使用场景上室友一定区别的；使用gThreadLocal是为了在多个线程中操作共享变量的时候线程之间互不影响。但是有些时候我们希望是互相影响的：**

**例如一个对象的使用次数，此时只能使用同步的方式来达到线程安全的目的。**

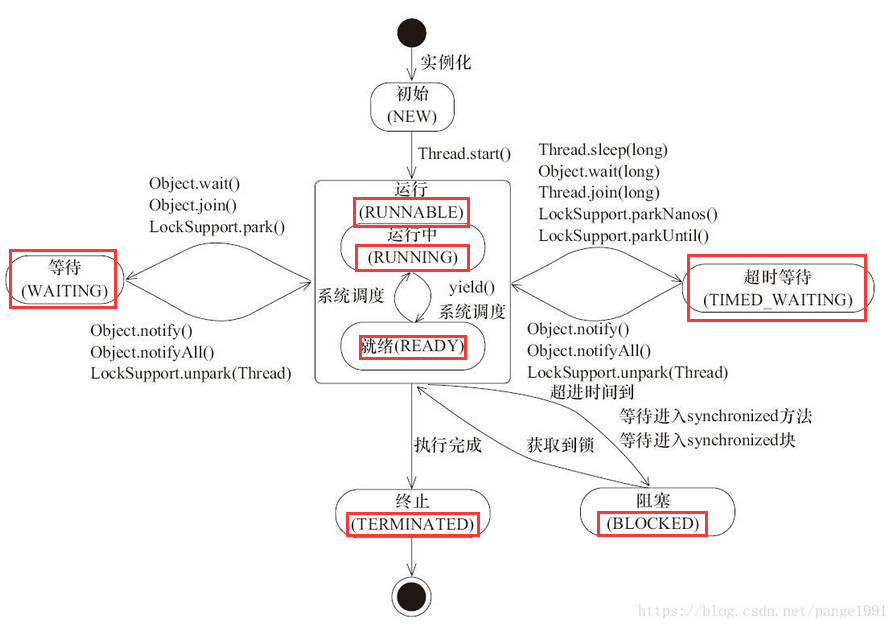
**如果我们希望线程之间对共享变量的操作不互相影响，例如数据库连接，A线程正在操作数据库就不希望B线程对连接的关闭对A线程造成影响；**

1. 多线程的锁原理



* 1. Synchroized
  2. ReentrantLock
  3. 读写锁

1. 线程基础
   1. 线程的各种状态



1. **初始(NEW)**：新创建了一个线程对象，但还没有调用start()方法。

2. **运行(RUNNABLE)**：Java线程中将就绪（ready）和运行中（running）两种状态笼统的成为“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）。

3.**阻塞(BLOCKED)**：表线程阻塞于锁。

4.**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）。

5.**超时等待(TIME\_WAITING)**：该状态不同于WAITING，它可以在指定的时间内自行返回。

6. **终止(TERMINATED)**：表示该线程已经执行完毕。

* 1. 线程的基本操作
     1. 创建线程

Java使用Thread类代表线程，所有的线程对象都必须是**Thread类或其子类的实例**。

Java可以用三种方式来创建线程，如下所示：

1）继承Thread类创建线程

2）实现Runnable接口创建线程

3）使用Callable和Future创建线程

几种方式的区别：

1、以实现类还是实现接口来区别：

* 线程只是实现Runnable或实现Callable接口，还可以继承其他类。
* 这种方式下，多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。
* 但是编程稍微复杂，如果需要访问当前线程，必须调用Thread.currentThread()方法。
* 继承Thread类的线程类不能再继承其他父类（Java单继承决定）。

2、rannable 与 callable：

不过是后者执行call()方法有返回值，后者线程执行体run()方法无返回值；

* + - 1. 继承Thread类创建线程

通过继承Thread类来创建并启动多线程的一般步骤如下

1】定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的任务，run()方法也称为线程执行体。

2】创建Thread子类的实例，也就是创建了线程对象

3】启动线程，即调用线程的start()方法

代码实例

public class MyThread extends Thread{//继承Thread类

　　public void run(){

　　//重写run方法

　　}

}

public class Main {

　　public static void main(String[] args){

　　　　new MyThread().**start()**;//创建并启动线程，此时线程处于Rannable的ready状态下，等待cpu分配；

　　}

}

* + - 1. 实现Runnable接口创建线程

通过实现Runnable接口创建并启动线程一般步骤如下：

1】定义Runnable接口的实现类，一样要重写run()方法，这个run（）方法和Thread中的run()方法一样是线程的执行体

2】创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象；

3】第三部依然是通过调用线程对象的start()方法来启动线程

代码实例：

public class MyThread2 implements Runnable {//实现Runnable接口

　　public void run(){

　　//重写run方法

　　}

}

public class Main {

　　public static void main(String[] args){

　　　　//创建并启动线程

　　　　MyThread2 myThread=new MyThread2();

　　　　Thread thread=new Thread(myThread);

　　　　thread().start();

　　　　//或者 new Thread(new MyThread2()).start();

　　}

}

* + - 1. 使用Callable和Future创建线程

Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()方法功能要强大。

**》call()方法可以有返回值**

**》call()方法可以声明抛出异常**

Future接口来代表Callable接口里call()方法的返回值，并且为Future接口提供了一个实现类FutureTask，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Thread类的target。在Future接口里定义了几个公共方法来控制它关联的Callable任务。

>boolean cancel(boolean mayInterruptIfRunning)：视图取消该Future里面关联的Callable任务

>V get()：**返回Callable里call（）方法的返回值，调用这个方法会导致程序阻塞，必须等到子线程结束后才会得到返回值**

>V get(long timeout,TimeUnit unit)：**返回Callable里call（）方法的返回值，最多阻塞timeout时间**，经过指定时间没有返回抛出TimeoutException

>boolean isDone()：**若Callable任务完成，返回True**

>boolean isCancelled()：如果在Callable任务正常完成前被取消，返回True

**创建并启动有返回值的线程的步骤如下：**

1】创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例;

2】使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值;

3】使用FutureTask对象作为Thread对象的target创建并启动线程（因为FutureTask实现了Runnable接口）

4】调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

代码实例：

public class UseCallable {  
  
 public static void main(String[] args) {  
 FutureTask<Integer> task = new FutureTask<>(new Callable<Integer>() {  
 @Override  
 public Integer call() throws Exception {  
 return 5;  
 }  
 });  
  
 new Thread(task).start();  
  
 try {  
 System.***out***.println(task.get());  
 } catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
}

* + 1. 终止线程

使用stop方法可以强行终止**正在运行或挂起的线程**。我们可以使用如下的代码来终止线程：

thread.stop();

虽然使用上面的代码可以终止线程，但使用stop方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果(数据不一致)，因此，并不推荐使用stop方法来终止线程。

* + 1. 线程中断

线程中断并不会使线程立即退出，而是给线程发送一个通知，告诉被中断的线程有人希望你退出了（通过中断标志位置位的方式）；

关于中断的几个方法：

public void Thread.interrupt() : 中断线程

public boolean Thread.isInterrupted() : 通过查询中断标志位，判断线程是否被中断

public static boolean Thread.interrupted() : 判断是否被中断， **并且清除当前的中断标志位**

这段程序当thread进入while循环之后，即使调用了thread线程的中断方法也不会使这个线程退出。

这里体现了interrupt方法和sleep方法的区别；

public static void main(String[] args) throws InterruptedException {  
 Thread thread = new Thread(){  
 public void run(){  
 while (true){  
 Thread.*yield*();  
 }  
 }  
 };  
 thread.start();  
 Thread.*sleep*(2000);  
 thread.interrupt();  
}

如果想要上边的代码在调用中断后退出：

public static void main(String[] args) throws InterruptedException {  
 Thread thread = new Thread(){  
 public void run(){  
 while (true){  
 if(Thread.*currentThread*().isInterrupted()){  
 System.***out***.println(***"线程被中断了，我要跳出while循环了"***);  
 break;  
 }  
 Thread.*yield*();  
 }  
 }  
 };  
  
 thread.start();  
 Thread.*sleep*(2000);  
 thread.interrupt();  
}

* + - 1. Thread.sleep()方法

如果线程在sleep的时候被调用了interrupt方法。 就会抛出InterruptedException

异常， 这个异常会将本来已经置位的中断标志位清零；

一次下面的代码是不能成功的退出while循环的：

public static void main(String[] args) throws InterruptedException {  
 Thread thread = new Thread(){  
 public void run(){  
 while (true){  
 if(Thread.*currentThread*().isInterrupted()){  
 System.***out***.println(***"我要跳出线程咯"***);  
 break;  
 }  
  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 System.***out***.println(***"我在sleep，中断我干嘛"***);  
 }  
 Thread.*yield*();  
 }  
 }  
 };  
 thread.start();  
 Thread.*sleep*(2000);  
 thread.interrupt();  
}

上述代码不能退出while循环是因为没有进入if子句， 但是明明thread被main线程中断了，为啥不能退出呢，原因是 sleep引起了中断异常， 而中断异常会将thread线程的中断标志位清零， 等进入下一个while循环的时候自然不会退出。

如果先要thread线程退出，可以在线程抛出中断异常的时候再次调用被线程的中断方法；

* + 1. 等待（wait）和通知（notify）

为了支持多线程之间的协作，jdk提供了以下的方法：

1）wait()、notify()和notifyAll()方法是本地方法，并且为final方法，无法被重写。

2）调用某个对象的wait()方法:

* 首先这个线程必须拿到这个对象的锁；
* 能让当前线程阻塞；
* 当前线程释放拿到的对象锁，**这里与sleep不同，sleep不会释放任何资源**；
* 当前线程会被放入所对象的等待队列中，等待被其他的线程调用所对象的notify方法；

3）调用某个对象（作为锁）的notify()方法：

* 能够唤醒一个正在等待这个对象的monitor的线程；
* 如果有多个线程都在等待这个对象的monitor，则只能唤醒其中一个线程；

4）调用notifyAll()方法能够唤醒所有正在等待这个对象的monitor的线程；

private static final Object ***CLOCK*** = new Object();  
  
public static class T1 extends Thread{  
 public void run(){  
 synchronized (***CLOCK***){  
 System.***out***.println(System.*currentTimeMillis*() + ***": T1 start !"***);  
 try {  
 System.***out***.println(System.*currentTimeMillis*() + ***": T1 wait for object"***);  
 //执行这个方法之后当前的线程将会进入CLOCK对象的等待队列中，等待别的线程调用CLOCK对象的notify方法  
 ***CLOCK***.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(System.*currentTimeMillis*() + ***": T1 end!"***);  
 }  
 }  
}  
  
public static class T2 extends Thread{  
 public void run(){  
 synchronized (***CLOCK***){  
 System.***out***.println(System.*currentTimeMillis*() + ***": T2 start notify one thread!"***);  
 ***CLOCK***.notify();  
 System.***out***.println(System.*currentTimeMillis*() + ***": T2 end"***);  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}  
  
public static void main(String[] args) {  
 Thread t1 = new T1();  
 Thread t2 = new T2();  
  
 t1.start();  
 t2.start();  
}

* + 1. 挂起（suspend）和继续执行（resume）
    2. 等待线程结束（join）和谦让（yield）
       1. join线程

**Thread提供了让一个线程等待另一个线程完成的方法-join方法。**

当某个程序执行流中调用其他线程的join()方法时，**调用线程**将被阻塞，直到被join()方法加入的join线程执行完毕，当前线程才会重新回到就绪状态。

join()方法有如下三种重载形式：

join()：等待被join的线程执行完毕

join(long millis)：等待被join的线程的时间最长为millis毫秒。如果在millis毫秒内被join()的线程还没有执行结束，则不再等待。

join(long millis,int nanos)：等待被join的线程最长为millis豪秒，nanos豪微秒

public class JoinThreadTest {  
  
 public static void main(String[] args) throws InterruptedException {  
 JoinThread joinThread = new JoinThread(***"被join的线程"***);  
 joinThread.start();  
 joinThread.join();  
  
 System.***out***.println(***"主线程等到了 被join的线程，两个线程一块"***);  
 }  
}

* + - 1. yield

yield()方法是一个和sleep()方法有点相似的方法，它也是Thread类提供的一个静态方法。

它也可以让当前正在执行的线程**暂停**，但它不会阻塞该线程，只是将该线程转入**就绪状态**。

yeild()只是让当前线程暂停一下，让**系统的线程调度器重新调度一次，**完全可能的情况是：当某个线程调用了yield()线程暂停之后，线程调度器又将其调度出来重新执行。

当某个线程调用了yield()方法暂停之后，**只有优先级与当前线程相同，或者优先级比当前线程更高的处于就绪状态的线程才会获得执行机会**。