# 1.线程池ThreadPoolExecutor.

参考：<https://www.cnblogs.com/dolphin0520/p/3932921.html>

https://blog.csdn.net/l\_kanglin/article/details/57411851

## 继承关系：

ThreadPoolExecutor继承自抽象类AbstractExecutorService.

AbstractExcutorService实现了ExecutorService接口

ExecutorService接口继承了Executor接口

## 构造函数：

**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**.acc = System.*getSecurityManager*() == **null** ?

**null** :

AccessController.*getContext*();

**this**.corePoolSize = corePoolSize;

**this**.maximumPoolSize = maximumPoolSize;

**this**.workQueue = workQueue;

**this**.keepAliveTime = unit.toNanos(keepAliveTime);

**this**.threadFactory = threadFactory;

**this**.handler = handler;

}

共有四个构造函数，但是其它三个构造函数都是调用这个构造函数来完成对象的实例化的。

各参数的意义：

**corePoolSize：**表示线程池的大小，一旦创建，它就是这么大。

**maximumPoolSize:**表示线程池的最大的大小。当任务过多时，可能会创建新的线程来执行任务，但是总的线程池的大小不能超过这个maximumPoolSize，否则就会抛出异常，或者拒绝任务，这个根据拒绝策略来定。

**KeepAliveTime:** 当线程池中线程的数量大于corePoolSize的时候，如果某个线程空闲时间超过keepAliveTime，就会销毁这个线程。

**Unit：**keppAliveTime的时间单位。

**workQueue**:任务缓存队列，当线程数量大于等于corePoolSize的时候，新来的任务就会被缓存到workQueue中。

**ThreadFactory**:用来创建新的线程的线程工厂。

**RejectedExecutionHandler** :拒绝处理任务的策略。

## 线程池的五种状态

// runState is stored in the high-order bits

**private** **static** **final** **int** ***RUNNING*** = -1 << ***COUNT\_BITS***;

**private** **static** **final** **int** ***SHUTDOWN*** = 0 << ***COUNT\_BITS***;

**private** **static** **final** **int** ***STOP*** = 1 << ***COUNT\_BITS***;

**private** **static** **final** **int** ***TIDYING*** = 2 << ***COUNT\_BITS***;

**private** **static** **final** **int** ***TERMINATED*** = 3 << ***COUNT\_BITS***;

当线程池刚刚创建的时候处于***RUNNING***状态，当调用shutdown()方法的时候，线程池会处于***SHUTDOWN***状态，此时线程池不再接收新的任务，但是会继续把正在执行的任务和缓存队列中的任务都执行完了以后再销毁。当调用线程池的shutdownNow()方法的时候,线程池会进行***STOP***状态，此时线程池不再接收新的任务，并且会尝试去终止正在执行的任务和清空缓存对列。当线程池为空的时候，即线程池中没有执行的线程了的时候。就会进入***TIDYING*** 状态。此时线程池就会执行另一个钩子函数terminate(),执行完这个函数，线程池就会从***TIDYING*** 状态变为***TERMINATED***状态。

如下图：



## 线程池原理：

线程池创建之初，会设定**corePoolSize**和**maximumPoolSize**。并且不会创建线程，除非通过预创建的方法prestartAllCoreThreads()或者prestartCoreThread()。在没有预创建的情况下，就是来一个任务创建一个线程。当线程数到达**corePoolSize**的时候，就会把新来的任务放到缓存队列中。当缓存队列满了以后，就会创建新的线程。当线程数到达**maximumPoolSize**。再有新的任务过来，就会抛出异常，拒绝任务，并关闭线程池。但是已经在缓存队列中的线程和正在执行的线程会执行完。

当线程的数量大于**corePoolSize的时候，**如果某个线程的空闲时间超过**KeepAliveTime，**就会被销毁。

## 使用示例

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.ThreadPoolExecutor;

**import** java.util.concurrent.TimeUnit;

**public** **class** TreadPoolTest {

**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("线程池数目"+executor.getPoolSize()+",队列中正在等待执行的任务数目："+executor.getQueue().size()+

",已执行完别的任务数目："+executor.getCompletedTaskCount());

}

executor.shutdown();

}

}

**class** MyTask **implements** Runnable{

**private** **int** taskNum;

**public** MyTask(**int** i) {

**this**.taskNum=i;

}

@Override

**public** **void** run() {

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

**try** {

Thread.*currentThread*().*sleep*(4000);

} **catch** (Exception e) {

e.printStackTrace();

}

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

}

}

**输出结果：**

正在执行task0

线程池数目1,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目2,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task1

线程池数目3,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task2

线程池数目4,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task3

线程池数目5,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：1,已执行完别的任务数目：0

正在执行task4

线程池数目5,队列中正在等待执行的任务数目：2,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：3,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：4,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

线程池数目6,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task10

线程池数目7,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task11

线程池数目8,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task12

线程池数目9,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task13

线程池数目10,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task14

task0执行完毕

task1执行完毕

task2执行完毕

正在执行task6

正在执行task5

正在执行task7

task4执行完毕

task3执行完毕

正在执行task9

task10执行完毕

正在执行task8

task14执行完毕

task12执行完毕

task13执行完毕

task11执行完毕

从输出结果中可以看出，当缓冲队列满的时候，才会创建新的线程。如果把任务数改为20的时候，输出接口如下：

正在执行task0

线程池数目1,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目2,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目3,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task1

正在执行task2

线程池数目4,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task3

线程池数目5,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：1,已执行完别的任务数目：0

正在执行task4

线程池数目5,队列中正在等待执行的任务数目：2,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：3,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：4,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

线程池数目6,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task10

线程池数目7,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task11

线程池数目8,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task12

线程池数目9,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task13

线程池数目10,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task14

Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task MyTask@7d4991ad rejected from java.util.concurrent.ThreadPoolExecutor@28d93b30[Running, pool size = 10, active threads = 10, queued tasks = 5, completed tasks = 0]

at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2063)

at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:830)

at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1379)

at TreadPoolTest.main(TreadPoolTest.java:11)

task4执行完毕

task0执行完毕

task1执行完毕

正在执行task7

task2执行完毕

task11执行完毕

task3执行完毕

正在执行task9

task10执行完毕

正在执行task8

task12执行完毕

正在执行task6

正在执行task5

task13执行完毕

task14执行完毕

从上面可以看出，当缓冲队列满的时候，而且线程池中的线程已经为maximumPoolSize的时候，就会报出异常，但是报出异常之后并不会终止线程池中的线程，而是会把正在执行的线程和缓存队列中的线程都执行完了以后，再终止线程池。

# 2.Synchronized的使用方法

**public** **class** Test5 {

**class** Inner{

//synchronized规则：

//1.两个线程并发访问同一个object的同一同步代码块，当一个线程执行这个代码块时，另一个线程就会阻塞

//2.两个线程并发访问同一个对象，访问不同的同步代码块时，当一个线程执行一个同步代码块时，另一个线程执行另一个同步代码块的访问也会被阻塞。

//3.两个线程并发访问同一个对象，一个线程访问同步代码块时，另一个线程可以访问非同步代码块，不会被阻塞。

//synchronized的两个用法：1.修饰方法。2.synchronized(object) ,加对象锁

**private** **void** mt1() {

**int** i=5;

**while**(i-->0) {

System.***out***.println(Thread.*currentThread*().getName()+":Inner.mt1()="+i);

**try** {

Thread.*sleep*(500);

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

}

**private** **void** mt2() {

**int** i=5;

**while**(i-->0) {

System.***out***.println(Thread.*currentThread*().getName()+":Inner.mt2()="+i);

**try** {

Thread.*sleep*(500);

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

}

}

**private** **void** mt1(Inner inner) {

**synchronized** (inner){//使用对象锁

inner.mt1();

}

}

**private** **void** mt2(Inner inner) {

// synchronized (inner) {

inner.mt2();

// }

}

**public** **static** **void** main(String[] args) {

**final** Test5 myt3=**new** Test5();

**final** Inner inner=myt3.**new** Inner();

Thread t1=**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

myt3.mt1(inner);

}

},"t1");

Thread t2=**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

myt3.mt2(inner);

}

},"t2");

t1.start();

t2.start();

}

}

# 3.线程和原子性

## 进程和线程

进程是资源调度和分配的基本单位，线程是CPU分配的基本单位。多个线程共享进程的资源（堆栈，上下文，寄存器）。进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束

## 原子性

原子性，所谓原子性就是不可分割性，是指某组操作是一个不可分割的部分。

Java提供了一种内置锁来支持原子性，叫做同步代码块（Synchronized block）.

若父类中的某个方法，和子类重写的该方法都是Synchronized方法，那么调用这个两个方法时，都会加上父类的锁，所以如果内置锁不可重入，则可能会发生死锁。

## 可见性

其实就是执行读操作的线程能够看到其他线程写入的值。即多个线程对内存的写入都是可见性。可见性需要使用同步机制来实现。

## 有序性

就是让程序按照代码的顺序串行执行，不进行重新排序。

## volatail变量

Volatail变量是一种削弱的同步机制，用来确保变量的更新操作通知到其他线程，也就是说编译器会把volatail变量标记为共享变量，不会让对volatail变量的操作和其他内存操作一起重排序。Volatail变量总是可见的，不会被缓存到如寄存器之类的处理器不可见的地方，所以每次读都是读取最新的值。

Volatial变量的优势：

（1）volatail变量的访问不会加锁，所以它不会造成线程阻塞，这就是一种比Syncronized更加轻量级的同步机制。

(2) volatail变量提供了变量状态的可见性。

Volatail变量在什么情况下使用？

仅当volatail变量能够简化代码的实现以及对同步策略的验证时，使用它：

1. 确保他们自身状态的可见性。
2. 确保他们所引用的对象的状态的可见性。
3. 标识一些重要的程序生命周期事件的发生。

（注：当在验证正确性的时候需要进行复杂的可见性判断时，就不要使用volatail变量）

使用volatail变量的条件：

1. 对变量的写入操作不依赖变量的当前值（反例：i++），或者你能确保只有单线程来访问这个变量。
2. 该变量不会和其它变量一起纳入不变性条件种
3. 在访问变量时不需要枷锁

典型用法：判断时否推出循环的某个标记变量。

# 4.jdk1.8中的锁

截止JDK1.8，java中的锁包括：参考 <http://ifeve.com/java_lock_see/>

# 5.ThreadLocal源码分析

对于ThreadLocal源码的分析，我觉得要从两个方面来进行。

第一步是对于ThreadLocal的存，取和删除变量等的流程进行分析。

第二步是对于ThreadLocal的ThreadLocalMap内部类进行分析，这一步同样非常重要。

## ThreadLocal的存取变量原理分析

对于存取原理的分析，我们要从以下几个方法入手：

**public** **void** set(T value);

**public** T get();

**public** **void** remove();

**private** T setInitialValue();

首先我们来看set方法，ThreadLocal的本质还是一个容器，容器一开始是空的，需要我们往里面存值。set方法的源码如下：

**public** **void** set(T value) {

Thread t = Thread.*currentThread*();

ThreadLocalMap map = getMap(t);

**if** (map != **null**)

map.set(**this**, value);

**else**

createMap(t, value);

}

上面的代码第一步是获取当前线程对象，这一步很简单，跳过。

第二步是调用getMap方法获取一个ThreadLocalMap对象，这一步乍看过去会有两个问题， ThreadLocalMap是一个什么东西，getMap方法干了什么。

我们上面说过ThreadLocalMap是ThreadLocal的一个内部类，它的实现原理我们后面再讨论，现在我们只需要知道它是一个存储<key,value>形式的键值对的容器。然后在Thread类里面定义了这样一个实例变量threadLocals：

ThreadLocal.ThreadLocalMap threadLocals = **null**;

接下来我们看一看getMap方法干了什么：

ThreadLocalMap getMap(Thread t) {

**return** t.threadLocals;

}

可以看到，getMap方法就是返回当前线程的实例变量threadLocals。

到这里第二步就完了。

第三步就是判空，我们获取到的map是不是空的，如果不为空，就以当前的ThreadLocal对象为key值（其实是存的是ThreadLocal对象的弱引用，为什么这么存，我们后面再讨论），以我们传进来的要保存的变量为value值，存到这个map容器中。 如果为空，我们就要新建一个ThreadLocalMap对象，从源码中，我们可以看到它调用了createMap()方法，那么我们来看看createMap方法是如何创建对象的：

**void** createMap(Thread t, T firstValue) {

t.threadLocals = **new** ThreadLocalMap(**this**, firstValue);

}

我们可以看到createMap方法调用了一个具有两个参数的构造方法，把当前ThreadLocal对象和要存入的值作为参数，这样做的好处是，在创建这个map的同时把我们要存入的值存进去了，不用再调用map的set方法存值。

到这里，我们ThreadLocal的set方法就看完了，其实从这里我们能得到很重要的信息，那就是ThreadLocal到底是利用了什么方法来巧妙地实现线程封闭的。

我们在当前线程调用ThreadLocal对象来存值的时候，其实最后是创建了一个变量的副本，然后把它存到了当前线程对象里面，而不是真的存在ThreadLocal对象里面，这样我们获取的时候，也只需要获取本线程对象中存的变量就可以了，这样就实现了各线程的变量之间相互不干扰。而它把ThreadLocal对象作为存储的键值，也能够很巧妙地解决当存在多个ThreadLocal变量时，如何从map中的取得你想要的值的问题。

接下来，我们明白了它是如何存值了以后，我们来看看它是怎么取值的。ThreadLocal的get方法源码如下：

**public** T get() {

Thread t = Thread.*currentThread*();

ThreadLocalMap map = getMap(t);

**if** (map != **null**) {

ThreadLocalMap.Entry e = map.getEntry(**this**);

**if** (e != **null**) {

@SuppressWarnings("unchecked")

T result = (T)e.value;

**return** result;

}

}

**return** setInitialValue();

}

可以看到前两步和set方法的前两步是一样的，跳过。

第三步也是判空，类似于set方法，当map非空时，我们把map中的值取出来就好了，取值的过程时这样的，首先调用getEntry()方法获取一个Entry对象，Entry对象可以简单理解为就是一个键值对，然后我们再返回Entry对象的value值，get方法就结束了。

当map为空时，我们不能再像set方法一样直接调用createMap方法来生成一个map对象，因为我们没有要存的值，那么我们首先要生成一个初始值来存，再来生成map对象。ThreadLocal把这两步封装成了一个方法，这个方法就是setInitialValue方法。那么我们来看一看，这个方法如何完成任务的:

**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;

}

根据上面的代码，我们可以看到，它第一步调用了initialValue（）来获取一个初始值，我们看以下这个方法：

**protected** T initialValue () {

**return** **null**;

}

可以看到它返回的是个空值，也就是这个方法其实是让我们根据需要在创建对象的时候重写的。不重写当然可以，那么你就必须在set了以后再get，否则就会报空指针异常。

然后我们再往后面看，是不是很眼熟，因为后面基本和set方法是一样的了，唯一多了一步返回value值，用于get方法的返回值。

最后我们来看看remove方法。

**public** **void** remove() {

ThreadLocalMap m = getMap(Thread.*currentThread*());

**if** (m != **null**)

m.remove(**this**);

}

这个方法非常简单，只有两步，第一步获取ThreadLocalMap对象，第二步调用map的remove方法把这个键值对从map对象中移除就可以了。

到这里，我们的第一大步就完成了，我们了解了ThreadLocal是如何存取和移除变量的。

## ThreadLocalMap的原理分析

ThreadLocalMap的源码不多，我主要从以下这5个方面来分析：

内部类Entry

相关的重要成员变量介绍

构造方法

get方法获取值的过程

set方法设置值得过程

### 内部类Entry

我们直接看源码：

**static** **class** Entry **extends** WeakReference<ThreadLocal<?>> {

/\*\* The value associated with this ThreadLocal. \*/

Object value;

Entry(ThreadLocal<?> k, Object v) {

**super**(k);

value = v;

}

我们上面说过，可以把它看成一个键值对对象，一个Entry存储 一个键值对。然后我们从源码中看到它继承了WeakReference类，也就是说它保存的key值的本质是一个ThreadLocal对象的弱引用，它为什么要这样做呢？

那是因为我们知道只有弱引用指向的对象是活不过下一次GC的。假设我们存的都是强引用，在主程序中这个ThreadLocal对象我们已经使用完了，或者把它置为null了，那么 这个对象是不是应该被回收了。然而它可能并不会被回收，因为我们在各个线程对象中还存着它的强引用，那么它就可能是GC roots链可达的。那么它就不会被回收，这样可能就会造成内存泄漏。

### 重要成员变量介绍

主要成员变量如下：

**private** **static** **final** **int** ***INITIAL\_CAPACITY*** = 16;

初始容量

**private** Entry[] table;

Entry数组table.

**private** **int** size = 0;

size是table中已经存在的变量

**private** **int** threshold; // Default to 0

当size超过这个值时，要调用resize方法来重新设置容量

### 构造方法

它有两个构造方法。

第一个构造函数是通过初始key和value来构造一个map对象。源码如下：

ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {

table = **new** Entry[***INITIAL\_CAPACITY***];

**int** i = firstKey.threadLocalHashCode & (***INITIAL\_CAPACITY*** - 1);

table[i] = **new** Entry(firstKey, firstValue);

size = 1;

setThreshold(***INITIAL\_CAPACITY***);

}

第一步是构建实例化Entry数组table，大小为初始容量字段值。

第二步是获取当前的key值的hashCode，然后对容量取模得到它在table中对应的索引值。但是firstKey.threadLocalHashCode字段是如何定义的，是怎么得到的它的索引值的呢？我们看源码：

**private** **final** **int** threadLocalHashCode = *nextHashCode*();

可以看到它定义了一个私有整型常量threadLocalHashCode 作为当前ThreadLocal对象的hashCode。而这个常量的值是通过nextHashCode方法得到的。那么，我们来看看这个方法的源码：

/\*\*

\* Returns the next hash code.

\*/

**private** **static** **int** nextHashCode() {

**return** *nextHashCode*.getAndAdd(***HASH\_INCREMENT***);

}

看到这个方法还是有点懵，这个*nextHashCode* 是什么东西？***HASH\_INCREMENT***又是什么东西？不要急，我们一个一个来看。首先是*nextHashCode，*它的定义如下：

/\*\*

\* The next hash code to be given out. Updated atomically. Starts at

\* zero.

\*/

**private** **static** AtomicInteger *nextHashCode* =

**new** AtomicInteger();

可以看到这是个AntomicInteger变量，这个就是能够自动保持原子性的一个静态整型变量。通过getAndAdd它能够根据增量自动获取下一个值。从0开始，每次获取都是在原理值的基础上加上一个增量。在nextHashCode方法中，这个增量就是***HASH\_INCREMENT***也就是说，每次都是在上一个ThreadLocal对象的key值基础上加上***HASH\_INCREMENT***得到的（因为是静态的，所以所有的对象共用一个值）。我们来看一下***HASH\_INCREMENT*** 的定义：

/\*\*

\* The difference between successively generated hash codes - turns

\* implicit sequential thread-local IDs into near-optimally spread

\* multiplicative hash values for power-of-two-sized tables.

\*/

**private** **static** **final** **int** ***HASH\_INCREMENT*** = 0x61c88647;

这个值为什么是0x61c88647呢，从注释中我们看到，这个值对于容量是2的多少次幂的hash表，是可以实现把连续的hash值转换成最优的分布。简单点说，对于连续的要存入hash表中的对象，是可以实现均匀分布的。为什么这个值可以实现均匀分布呢。。。目前水平有限，所以不知道。。。。

构造函数后面的步骤就比较简单了，跳过。

接下来，我们来看第二个构造函数：

**private** ThreadLocalMap(ThreadLocalMap parentMap) {

Entry[] parentTable = parentMap.table;

**int** len = parentTable.length;

setThreshold(len);

table = **new** Entry[len];

**for** (**int** j = 0; j < len; j++) {

Entry e = parentTable[j];

**if** (e != **null**) {

@SuppressWarnings("unchecked")

ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();

**if** (key != **null**) {

Object value = key.childValue(e.value);

Entry c = **new** Entry(key, value);

**int** h = key.threadLocalHashCode & (len - 1);

**while** (table[h] != **null**)

h = *nextIndex*(h, len);

table[h] = c;

size++;

}

}

}

}

第二个构造函数，是通过一个map构建一个新的map。感觉用得不多，也不难理解，因此步多做描述。（简而言之，就是懒了。。。s）

### get方法获取值的过程

我觉得get方法和set方法是这个map得核心，首先来看get方法（因为get方法比较简单）：

**private** Entry getEntry(ThreadLocal<?> key) {

**int** i = key.threadLocalHashCode & (table.length - 1);

Entry e = table[i];

**if** (e != **null** && e.get() == key)

**return** e;

**else**

**return** getEntryAfterMiss(key, i, e);

}

这个map的get方法就是getEntry(ThreadLocal<?> key)，它的作用是通过key值获取存储这个key值的Entry对象。

第一步，根据key值获取对应的索引值。

第二步，根据索引值得到Entry对象。

第三步，判断这个Entry对象是否为空。如果不为空，并且通过e.get()方法获取到的key值于我们要取得key值相同，就直接返回这个Entry（这个get方法是Entry的父类WeakReference的父类Reference定义的一个方法，作用就是返回我们调用父类构造方法的时候传进去的值）。

否则我们就要通过getEntryAfterMiss(key, i, e)来进行线性探测来寻找key值。getEntryAfterMiss方法的代码如下：

/\*\*

\* Version of getEntry method for use when key is not found in

\* its direct hash slot.

\*

\* **@param** key the thread local object

\* **@param** i the table index for key's hash code

\* **@param** e the entry at table[i]

\* **@return** the entry associated with key, or null if no such

\*/

**private** Entry getEntryAfterMiss(ThreadLocal<?> key, **int** i, Entry e) {

Entry[] tab = table;

**int** len = tab.length;

**while** (e != **null**) {

ThreadLocal<?> k = e.get();

**if** (k == key)

**return** e;

**if** (k == **null**)

expungeStaleEntry(i);

**else**

i = *nextIndex*(i, len);

e = tab[i];

}

**return** **null**;

}

这个方法就是从当前的第一个值开始，也就是所谓的key值得hashCode取模后得到得的索引位置开始，往后遍历一个段。遍历的时候首先分两种情况，第一种情况就是当前Entry的key值等于我们想要的key值，那么直接返回当前Entry对象，遍历结束，这个方法也结束了。 第二种情况是当前的key值与我们想要的key值不相等，在这个情况下，我们又可以继续分两种情况考虑，一种是当前Entry的key值为null，也就是说这个Entry存储的ThreadLocal对象已经被回收了，这个索引

失效了。我们把这种情况称为无效Entry，当遇到无效Entry的时候，就从当前Entry开始进行一次段清理，这个段清理通过expungeStaleEntry方法实现。另一种是当前Entry的key值不为null，那么就继续遍历下一个Entry。 最后 如果遍历完了当前段，也没有找到对应的Entry，说明当前的map种没有存这个key值，直接返回null。这个方法的大体逻辑弄清楚了，那么实现段清理的方法expungeStaleEntry是怎样的呢？它的代码如下：

**private** **int** expungeStaleEntry(**int** staleSlot) {

Entry[] tab = table;

**int** len = tab.length;

// expunge entry at staleSlot

tab[staleSlot].value = **null**;

tab[staleSlot] = **null**;

size--;

// Rehash until we encounter null

Entry e;

**int** i;

**for** (i = *nextIndex*(staleSlot, len);

(e = tab[i]) != **null**;

i = *nextIndex*(i, len)) {

ThreadLocal<?> k = e.get();

**if** (k == **null**) {

e.value = **null**;

tab[i] = **null**;

size--;

} **else** {

**int** h = k.threadLocalHashCode & (len - 1);

**if** (h != i) {

tab[i] = **null**;

// Unlike Knuth 6.4 Algorithm R, we must scan until

// null because multiple entries could have been stale.

**while** (tab[h] != **null**)

h = *nextIndex*(h, len);

tab[h] = e;

}

}

}

**return** i;

}

这个方法的功能说白了就是把当前Entry所在段的后面的所有无效Entry都清理掉

因为我们是从检测到的第一个无效Entry开始，所以这个方法的第一步就是清理掉当前Entry，清理方法是把Entry的value值设置为null,再把Entry设置为null。然后从下一个Entry开始，依次遍历这个段的所有Entry。遍历的时候分两种情况，第一种是当前Entry是无效的Entry，那么就按照上述方法清理掉这个Entry。 第二种情况是当前Entry不是无效的，那么就要对这个Entry进行重定位，因为前面可能删除了Entry，所以为了使同一个索引值的key值之间不出现空的情况，那么就把这个Entry进行重定位，重定位的方法分为两步，第一步就是将当前索引的table的值置为空（table[i]=null），第二步是从这个key值的第一个索引值开始往后遍历，找到第一个空Entry，就把当前Entry赋值给那个空Entry。清理完成后返回第一个空值。

### set方法设置值的过程

直接来看set方法的源码：

**private** **void** set(ThreadLocal<?> key, Object value) {

// We don't use a fast path as with get() because it is at

// least as common to use set() to create new entries as

// it is to replace existing ones, in which case, a fast

// path would fail more often than not.

Entry[] tab = table;

**int** len = tab.length;

**int** i = key.threadLocalHashCode & (len-1);

**for** (Entry e = tab[i];

e != **null**;

e = tab[i = *nextIndex*(i, len)]) {

ThreadLocal<?> k = e.get();

**if** (k == key) {

e.value = value;

**return**;

}

**if** (k == **null**) {

replaceStaleEntry(key, value, i);

**return**;

}

}

tab[i] = **new** Entry(key, value);

**int** sz = ++size;

**if** (!cleanSomeSlots(i, sz) && sz >= threshold)

rehash();

}

# 6．ReentrantLock和synchronized

## ReentrantLock类

Java的java.util.concurrent.lock类是对锁定的抽象，它允许把锁定的实现作为java类，而不是语言的特性。这样就为Lock的实现留下了很多种可能性，lock的实现可能有不同的调度算法，锁定语义或性能特性。

ReentrantLock就是lock的一个是实现类。它拥有和synchronized相同的并发性和内存语义。但是增加了轮询锁、定时锁等候和可中断锁等候的一些特性。此外，它还在激烈争用情况下有更佳的性能。（即在多线程环境中，可以花更少的时间调度线程，更多的时间来执行线程）。

Reetrant意味着什么呢？简单来说，它有一个锁获取计数器，当拥有锁的线程再次得到锁，这个计数器就加1 ，意味着需要释放两次才能真正地释放锁。这模仿了synchronized的语义：当线程进入线程已经拥有监控器保护的synchronized代码块时，允许线程继续执行。但是当线程退出第二个synchronized代码块时，不释放锁。仅当线程退出它进入监控器保护的第一个代码块时，才释放锁。

## Reentrant和syncronized的区别

第一是使用方法的区别。Lock必须在finally中释放，否则，若受保护的代码块在执行过程中出现了异常，锁将永远无法被释放。这一点看起来没什么，但是相当于埋下了一颗定时炸弹，一旦爆炸，就会让你耗费很长的时间区解决。而synchronized就没有这个问题，JVM会自动释放锁。

第二是伸缩性的区别。Lock花在调度线程上的时间比synchronized要少很多，从而为提供吞吐率留下了空间。

## 这不公平

（我两，太不公平。。。哈哈哈，看到这个标题，就想到了朱克版本的离不开你，真的是很深入人心。哈哈，所以延用了这个标题。）

ReentrantLock的一个构造参数是boolean类型的，它允许我们选择想要一个公平锁还是不公平锁，公平锁按照线程请求锁的顺序分配锁；不公平锁允许线程直接获取，这种情况下，可能比先请求的线程更快获得锁。

为什么不把所有的锁设置为公平锁呢，公平是好事，不是吗？是的，公平保证了锁是非常健壮的锁，有很大的性能成本。保证公平需要更多性能去保证记账和同步，意味着公平锁的效率会比不公平锁低很多。Reentrantlock的参数默认是false的，默认是一个不公平锁，除非你的程序需要线程严格按照顺序去执行，否则，不要把它设置为公平锁。

## 为什么ReentrantLock不能取代synchronized

看起来，ReentrantLock拥有synchronized所有拥有的一起，还拥有synchronized不拥有的东西。ReentrantLock拥有和synchronized相同的并发性和内存语义，而且在高负荷的情况下拥有更好的性能。那么ReentrantLock是不是完全取代synchronized了？答案是不能。为什么呢？我一觉得用“好东西”完全取代“旧事物” 并非是可取的做法。

1. synchronized同步锁的释放由JVM来完成，而如果在使用lock的时候忘记在finally释放这个锁了，那么会造成严重的后果。
2. synchronized在线程转储时，能够存储锁的信息，这对调试非常有价值，它能 够追踪死锁或其它有异常的来源。而lock是一个类，你不知道哪些类拥有lock对象。

# 7.如何调试CPU占用率过高的线程