线程池

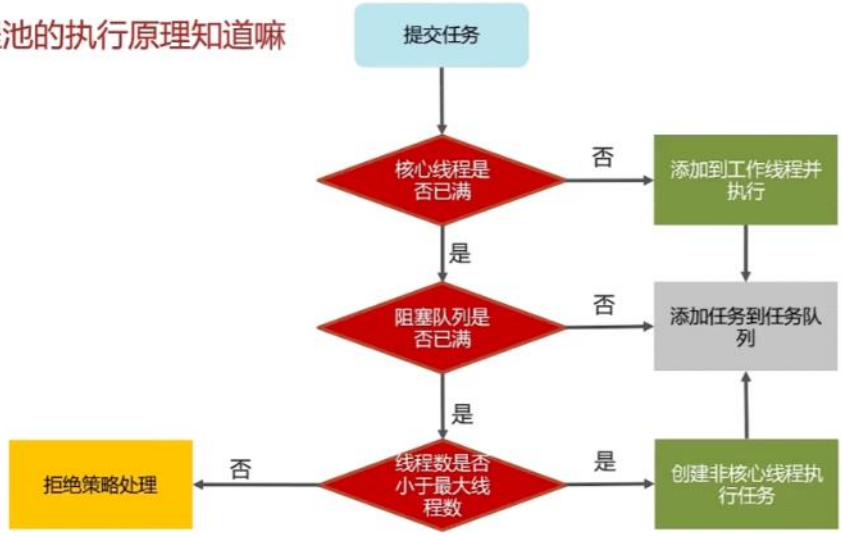
2024年4月3日 19:51

说一下线程池的核心参数

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

- corePoolSize 核心线程数目
- maximumPoolSize 最大线程数目 = (核心线程+救急线程的最大数目)
- keepAliveTime 生存时间 - 救急线程的生存时间，生存时间内没有新任务，此线程资源会释放
- unit 时间单位 - 救急线程的生存时间单位，如秒、毫秒等
- workQueue - 当没有空闲核心线程时，新来任务会加入到此队列排队，队列满会创建救急线程执行任务
- threadFactory 线程工厂 - 可以定制线程对象的创建，例如设置线程名字、是否是守护线程等
- handler 拒绝策略 - 当所有线程都在繁忙，workQueue 也放满时，会触发拒绝策略

线程池的执行原理知道嘛



如果核心或临时线程执行完成任务后会检查阻塞队列中是否有需要执行的线程，如果有，则使用非核心线程执行任务

1.AbortPolicy：直接抛出异常，默认策略；

2.CallerRunsPolicy：用调用者所在的线程来执行任务；

3.DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

4.DiscardPolicy：直接丢弃任务；

线程池中有哪些常见的阻塞队列

workQueue - 当没有空闲核心线程时，新来任务会加入到此队列排队，队列满会创建救急线程执行任务

1.ArrayBlockingQueue：基于数组结构的有界阻塞队列，FIFO。

2.LinkedBlockingQueue：基于链表结构的有界阻塞队列，FIFO。

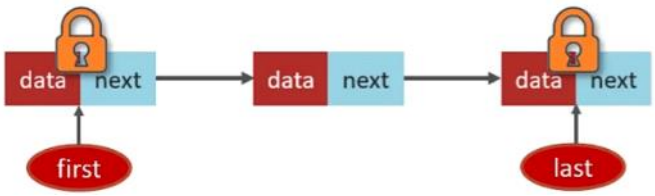
3.DelayedWorkQueue：是一个优先级队列，它可以保证每次出队的任务都是当前队列中执行时间最靠前的

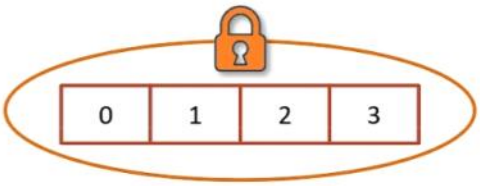
4.SynchronousQueue：不存储元素的阻塞队列，每个插入操作都必须等待一个移出操作。

线程池中有哪些常见的阻塞队列

ArrayBlockingQueue的LinkedBlockingQueue区别

LinkedBlockingQueue	ArrayBlockingQueue
默认无界，支持有界	强制有界
底层是链表	底层是数组
是懒惰的，创建节点的时候添加数据	提前初始化 Node 数组
入队会生成新 Node	Node需要是提前创建好的
两把锁（头尾）	一把锁





如何确定核心线程数

- IO密集型任务

一般来说：文件读写、DB读写、网络请求等　　核心线程数大小设置为2N+1

- CPU密集型任务

一般来说：计算型代码、Bitmap转换、Gson转换等　　核心线程数大小设置为N+1

```
public static void main(String[] args) {
    //查看机器的CPU核数
    System.out.println(Runtime.getRuntime().availableProcessors());
}
```

查看机器的CPU核数

如何确定核心线程数

参考回答：

① 高并发、任务执行时间短 → （ CPU核数+1 ），减少线程上下文的切换

② 并发不高、任务执行时间长

- IO密集型的任务 → (CPU核数 * 2 + 1)
- 计算密集型任务 → （ CPU核数+1 ）

③ 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）

线程池的种类有哪些

在java.util.concurrent.Executors类中提供了大量创建连接池的静态方法，常见就有四种

1. 创建使用固定线程数的线程池

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

- 核心线程数与最大线程数一样，没有救急线程
- 阻塞队列是LinkedBlockingQueue，最大容量为Integer.MAX_VALUE

适用于任务量已知，相对耗时的任务

线程池的种类有哪些

2. 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)执行

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

- 核心线程数和最大线程数都是1
- 阻塞队列是LinkedBlockingQueue，最大容量为Integer.MAX_VALUE

适用于按照顺序执行的任务

线程池的种类有哪些

3. 可缓存线程池

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

- 核心线程数为0
- 最大线程数是Integer.MAX_VALUE
- 阻塞队列为SynchronousQueue:不存储元素的阻塞队列，每个插入操作都必须等待一个移出操作。

适合任务数比较密集，但每个任务执行时间较短的情况

线程池的种类有哪些

4. 提供了“延迟”和“周期执行”功能的ThreadPoolExecutor。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,new DelayedWorkQueue());
}
public ScheduledThreadPoolExecutor(int corePoolSize,
                                   ThreadFactory threadFactory) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), threadFactory);
}
public ScheduledThreadPoolExecutor(int corePoolSize,
                                   RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), handler);
}
public ScheduledThreadPoolExecutor(int corePoolSize,
                                   ThreadFactory threadFactory,
                                   RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), threadFactory, handler);
}
```

线程池的种类有哪些

① newFixedThreadPool：创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待

② newSingleThreadExecutor：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)执行

③ newCachedThreadPool：创建一个可缓存线程池，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

④ newScheduledThreadPool：可以执行延迟任务的线程池，支持定时及周期性任务执行

为什么不建议用Executors创建线程池

参考阿里开发手册《Java开发手册-嵩山版》

【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

1） FixedThreadPool 和 SingleThreadPool：

允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。

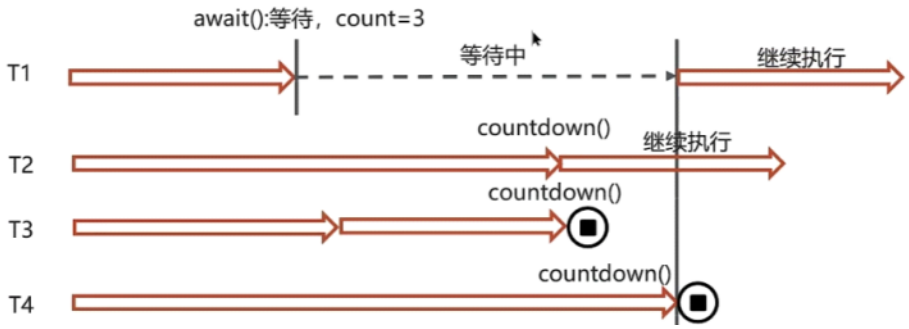
2） CachedThreadPool：

允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

CountDownLatch

CountDownLatch（闭锁/倒计时锁）用来进行线程同步协作，等待所有线程完成倒计时（一个或者多个线程，等待其他多个线程完成某件事情之后才能执行）

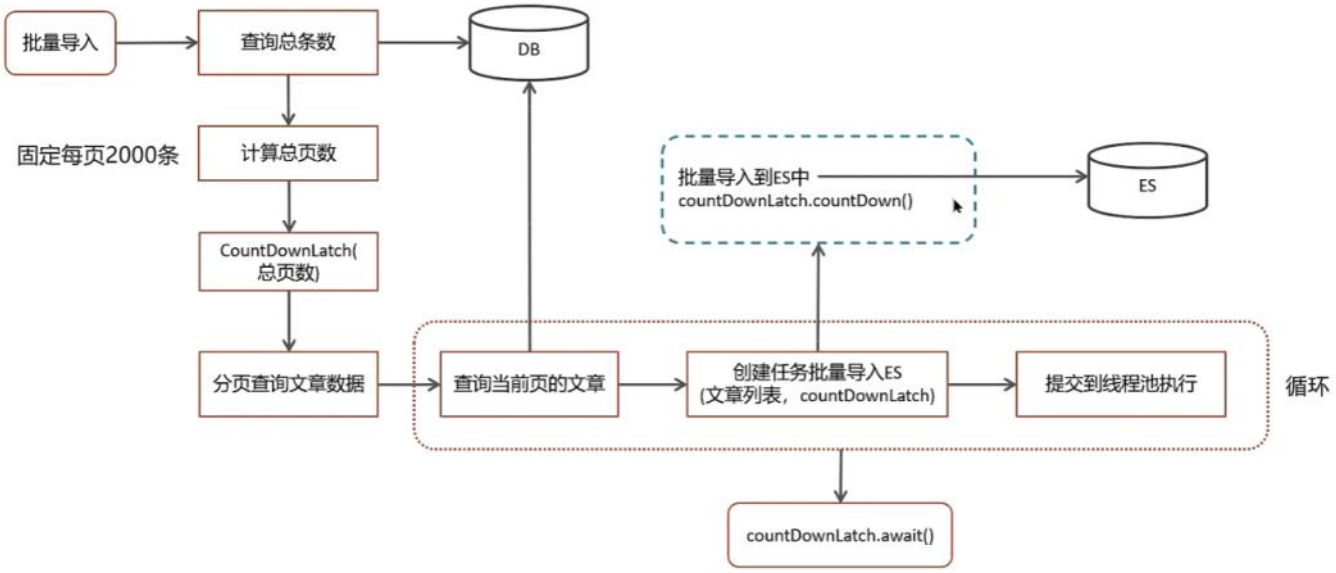
- 其中构造参数用来初始化等待计数值
- await() 用来等待计数归零
- countDown() 用来让计数减一



多线程使用场景一（es数据批量导入）

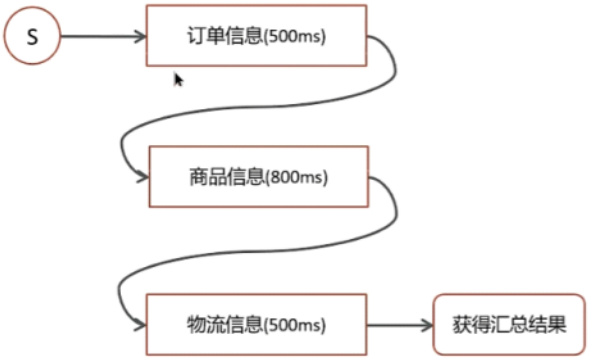
在我们项目上线之前，我们需要把数据库中的数据一次性的同步到es索引库中，但是当时的数据好像是1000万左右，一次性读取数据肯定不行（oom异常），当时我就想到可以使用线程池的方式导入，利用CountDownLatch来控制，就能避免一次性加载过多，防止内存溢出

多线程使用场景一（es数据批量导入）

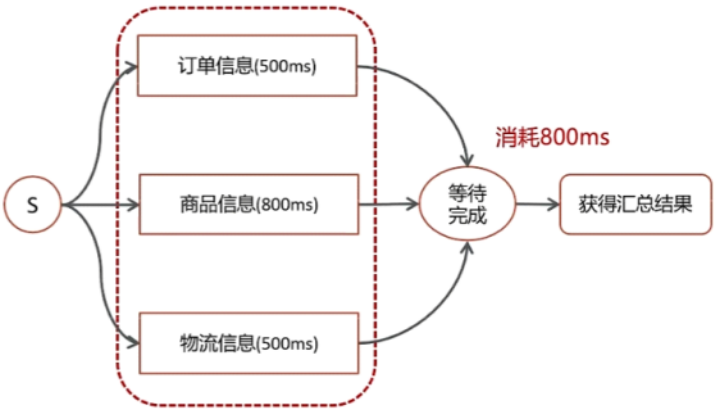


多线程使用场景二（数据汇总）

在一个电商网站中，用户下单之后，需要查询数据，数据包含了三部分：订单信息、包含的商品、物流信息；这三块信息都在不同的微服务中进行实现的，我们如何完成这个业务呢？



共消耗：1800ms



线程池

如何控制某个方法允许并发访问线程的数量

Semaphore [ˈsɛməˌfor] 信号量，是JUC包下的一个工具类，底层是AQS，我们可以通过其限制执行的线程数量

使用场景：

通常用于那些资源有明确访问数量限制的场景，常用于限流 。

停车场共3个车位

如何控制某个方法允许并发访问线程的数量

Semaphore使用步骤

- 创建Semaphore对象，可以给一个容量
- semaphore.acquire()： 请求一个信号量，这时候的信号量个数-1（一旦没有可使用的信号量，也即信号量个数变为负数时，再次请求的时候就会阻塞，直到其他线程释放了信号量）
- semaphore.release()：释放一个信号量，此时信号量个数+1

```
// 1. 创建 semaphore 对象
Semaphore semaphore = new Semaphore(3);
// 2. 10个线程同时运行
for (int i = 0; i < 10; i++) {
    new Thread(() -> {
        try {
            // 3. 获取许可
            semaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            System.out.println("running...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("end...");
        } finally {
            // 4. 释放许可
            semaphore.release();
        }
    }).start();
}
```

如何控制某个方法允许并发访问线程的数量

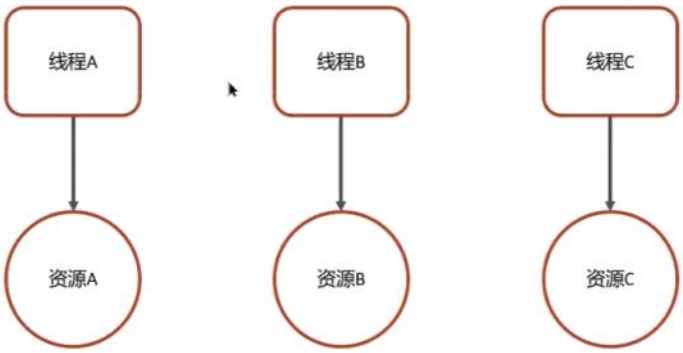
在多线程中提供了一个工具类Semaphore，信号量。在并发的情况下，可以控制方法的访问量

1. 创建Semaphore对象，可以给一个容量
2. acquire()可以请求一个信号量，这时候的信号量个数-1
3. release()释放一个信号量，此时信号量个数+1

ThreadLocal概述

ThreadLocal是多线程中对于解决线程安全的一个操作类，它会为每个线程都分配一个独立的线程副本从而解决了变量并发访问冲突的问题。ThreadLocal 同时实现了线程内的资源共享

案例：使用JDBC操作数据库时，会将每一个线程的Connection放入各自的ThreadLocal中，从而保证每个线程都在各自的 Connection 上进行数据库的操作，避免A线程关闭了B线程的连接。



ThreadLocal基本使用

- set(value) 设置值
- get() 获取值
- remove() 清除值

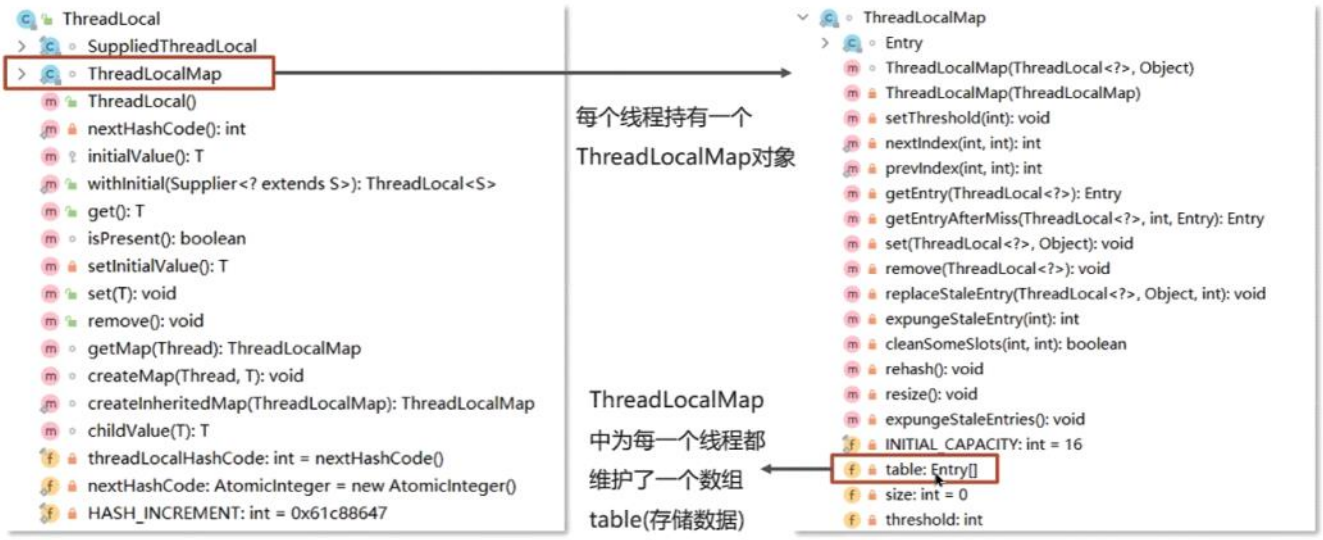
```
static ThreadLocal<String> threadLocal = new ThreadLocal<>();

public static void main(String[] args) {
    new Thread(() -> {
        String name = Thread.currentThread().getName();
        threadLocal.set("itcast");
        print(name);
        System.out.println(name + "-after remove : " + threadLocal.get());
    }, "t1").start();
    new Thread(() -> {
        String name = Thread.currentThread().getName();
        threadLocal.set("itheima");
        print(name);
        System.out.println(name + "-after remove : " + threadLocal.get());
    }, "t2").start();
}

static void print(String str) {
    //打印当前线程中本地内存中本地变量的值
        System.out.println(str + " :" + threadLocal.get());
    //清除本地内存中的本地变量
        threadLocal.remove();
}
```

ThreadLocal的实现原理&源码解析

ThreadLocal本质来说就是一个线程内部存储类，从而让多个线程只操作自己内部的值，从而实现线程数据隔离



ThreadLocal的实现原理&源码解析

set方法

```
public void set(T value) {
  //获取当前线程对象
  Thread t = Thread.currentThread();
  //根据当前线程对象，获取ThreadLocal中的ThreadLocalMap
  ThreadLocalMap map = getMap(t);
  //如果map存在
  if (map != null)
    //执行map中的set方法，进行数据存储
    map.set(this, value);
  else
    //否则创建ThreadLocalMap，并存值
    createMap(t, value);
}
```

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

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

    //内部成员数组，INITIAL_CAPACITY值为16的常量
    table = new Entry[INITIAL_CAPACITY];

    //位运算，结果与取模相同，计算出需要存放的位置
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
    table[i] = new Entry(firstKey, firstValue);
    size = 1;
    setThreshold(INITIAL_CAPACITY);
}
```

ThreadLocal的实现原理&源码解析

get方法/remove方法

```
public T get() {
    Thread t = Thread.currentThread();
    //根据线程对象，获取对应的ThreadLocalMap
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        //获取ThreadLocalMap中对应的Entry对象
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            //获取Entry中的value
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

```
private Entry getEntry(ThreadLocal<?> key) {
    //确定数组下标位置
    int i = key.threadLocalHashCode & (table.length - 1);
    //得到该位置上的Entry
    Entry e = table[i];
    if (e != null && e.get() == key)
        return e;
    else
        return getEntryAfterMiss(key, i, e);
}
```

面试官：你对ThreadLocal理解的挺深的，你知道ThreadLocal的内存泄露问题吗?

ThreadLocal-内存泄露问题

Java对象中的四种引用类型：强引用、软引用、弱引用、虚引用

- 强引用：最为普通的引用方式，表示一个对象处于有用且必须的状态，如果一个对象具有强引用，则GC并不会回收它。即便堆中内存不足了，宁可出现OOM，也不会对其进行回收

```
User user = new User();
```

- 弱引用：表示一个对象处于可能有用且非必须的状态。在GC线程扫描内存区域时，一旦发现弱引用，就会回收到弱引用相关联的对象。对于弱引用的回收，无关内存区域是否足够，一旦发现则会被回收

```
User user = new User();
WeakReference weakReference = new WeakReference(user);
```

ThreadLocal-内存泄露问题

每一个Thread维护一个ThreadLocalMap，在ThreadLocalMap中的Entry对象继承了WeakReference。其中key为使用弱引用的ThreadLocal实例，value为线程变量的副本

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v
    }
}
```

弱引用，内存不太够的时候，优先回收

强引用，不会被回收

内存泄漏

防止内存泄漏：务必remove

谈谈你对ThreadLocal的理解

1. ThreadLocal 可以实现【资源对象】的线程隔离，让每个线程各用各的【资源对象】，避免争用引发的线程安全问题

2. ThreadLocal 同时实现了线程内的资源共享

3. 每个线程内有一个 ThreadLocalMap 类型的成员变量，用来存储资源对象

   a)调用 set 方法，就是以 ThreadLocal 自己作为 key，资源对象作为 value，放入当前线程的 ThreadLocalMap 集合中

   b)调用 get 方法，就是以 ThreadLocal 自己作为 key，到当前线程中查找关联的资源值

   c)调用 remove 方法，就是以 ThreadLocal 自己作为 key，移除当前线程关联的资源值

4. ThreadLocal内存泄漏问题

ThreadLocalMap 中的 key 是弱引用，值为强引用； key 会被GC 释放内存，关联 value 的内存并不会释放。建议主动 remove 释放 key，value