线程池的正确创建方式

在阿里巴巴开发手册中不推荐使用Excutors创建线程池，因为有可能出现OOM问题

原因：其中阻塞队列的默认实现可能是无界队列，默认最大容量为Integer.MAX_VALUE，当任务添加速度大于删除速度时是有可能造成OOM问题的

正确使用方法：

1）使用ThreadPoolExecutor构造创建线程池

2）使用开源类库，如apache，guava等，例如：guava提供的ThreadFactoryBuilder创建线程池

线程池的底层实现原理

allowCoreThreadTimeOut，可以使线程池也能回收超时的核心线程，默认为false，设置为true即可

线程池的五种状态：https://zhuanlan.zhihu.com/p/93041206

1）运行：RUNNING　　-1 << COUNT_BITS　　COUNT_BITS = Integer.SIZE - 3=29

2）关机：SHUTDOWN　　0

3）停止：STOP　　1

4）整理：TIDYING　　2

5）终止：TERMINATED　　3

1. 运行->关机。调用了shutdown()之后，或者执行了finalize()
2. (运行或者关机)->停止。调用了shutdownNow()之后会转换这个状态
3. 关机->清理。当线程池和队列都为空的时候
4. 停止整理。当线程池为空的时候
5. IDYING->终止。执行完terminated()回调之后会转换为这个状态

ctl分析：https://www.cnblogs.com/moonfair/p/13477974.html

作用：记录线程池当前的生命周期以及当前的工作的线程数，将一个整形变量按二进制分为两部分，分别表示两个信息，是AtomicInteger类型（原子整形），对其操作具有原子性，ThreadPoolExecutor使用ctlOf方法来将runState、workCount两个变量打包成一个ctl变量。

两个工具变量：COUNT_BITS=29，CAPACITY=（1<<COUNT_BITS）-1

部分源码：

```
1 //拆包函数
2 private static int runStateOf(int c)     { return c & ~CAPACITY; }
3 private static int workerCountOf(int c)  { return c & CAPACITY; }
4 //打包函数
5 private static int ctlOf(int rs, int wc) { return rs | wc; }
```

好文就要多看：

https://github.com/aCoder2013/blog/issues/28

http://www.likecs.com/default/index/show?id=66368

volatile的作用

1）保持内存可见性

volatile如何保持内存可见性

volatile的特殊规则就是：

- read、load、use动作必须**连续出现**。
- assign、store、write动作必须**连续出现**。

所以，使用volatile变量能够保证：

- 每次 `读取前` 必须先从主内存刷新最新的值。
- 每次 `写入后` 必须立即同步回主内存当中。

2）防止指令重排

volatile关键字通过 `"内存屏障"` 来防止指令被重排序。

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。然而，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，Java内存模型采取保守策略。

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。

ThreadLocal：（ThreadLocalMap是ThreadLocal的一个静态内部类）

从名字我们就可以看到ThreadLocal叫做线程变量，意思是ThreadLocal中填充的变量属于当前线程，该变量对其他线程而言是隔离的。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。

使用场景：

1、在进行对象跨层传递的时候，使用ThreadLocal可以避免多次传递，打破层次间的约束。

2、线程间数据隔离

3、进行事务操作，用于存储线程事务信息。

4、数据库连接，Session会话管理。

（1）每个Thread维护着一个ThreadLocalMap的引用

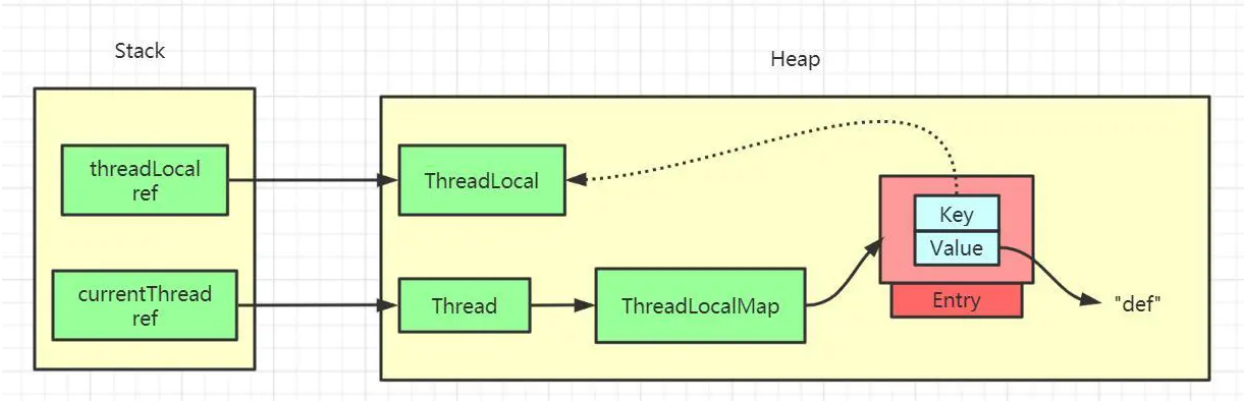
（2）ThreadLocalMap是ThreadLocal的内部类，用Entry来进行存储

（3）ThreadLocal创建的副本是存储在自己的threadLocals中的，也就是自己的ThreadLocalMap。

（4）ThreadLocalMap的键值为ThreadLocal对象，而且可以有多个threadLocal变量，因此保存在map中

（5）在进行get之前，必须先set，否则会报空指针异常，当然也可以初始化一个，但是必须重写initialValue()方法。

（6）ThreadLocal本身并不存储值，它只是作为一个key来让线程从ThreadLocalMap获取value。



内存泄漏问题：

1、Thread中有一个map，就是ThreadLocalMap

2、ThreadLocalMap的key是ThreadLocal，值是我们自己设定的。

3、ThreadLocal是一个弱引用，当为null时，会被当成垃圾回收

4、重点来了，突然我们ThreadLocal是null了，也就是要被垃圾回收器回收了，但是此时我们的ThreadLocalMap生命周期和Thread的一样，它不会回收，这时候就出现了一个现象。那就是ThreadLocalMap的key没了，但是value还在，这就造成了内存泄漏。

解决办法：使用完ThreadLocal后，执行remove操作，避免出现内存溢出情况。

解决办法：

不过不用担心，ThreadLocal提供了这个问题的解决方案。

每次操作set、get、remove操作时，会相应调用 ThreadLocalMap 的三个方法，ThreadLocalMap的三个方法在每次被调用时 都会直接或间接调用一个 expungeStaleEntry() 方法，这个方法会将key为null的 Entry 删除，从而避免内存泄漏。

```
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 stal
                while (tab[h] != null)
                    h = nextIndex(h, len);
                tab[h] = e;
            }
        }
    }
```

如果键为空，则把这个 Entry 键值对删除

那么问题又来了，如果一个线程运行周期较长，而且将一个大对象放入LocalThreadMap后便不再调用set、get、remove方法仍然有可能key的弱引用被回收后，值引用没有被回收，此时该仍然可能会导致内存泄漏。

这个问题确实存在，没办法通过ThreadLocal解决，而是需要程序员在完成ThreadLocal的使用后要养成手动调用remove的习惯，从而避免内存泄漏。

既然弱引用会导致内存泄漏，那ThreadLocalMap为什么对ThreadLocal的引用要设置成弱引用?

为了尽快回收这个线程变量，因为这个线程变量可能使用场景不是特别多，所以希望使用完后能尽快被释放掉。因为线程拥有的资源越多，就越臃肿，线程切换的开销就越大，所以希望尽量降低线程拥有的资源量。

进程之间进行通信的几种方式

IPC的方式通常有管道（包括无名管道和命名管道）（pipe）、消息队列、信号量、共享存储（Shared Memory）、Socket、Streams等。其中 Socket和Streams支持不同主机上的两个进程

IPC。

1.管道：速度慢，容量有限，只有父子进程能通讯　　（UNIX 系统IPC最古老的形式）

特点：

1）是半双工的（数据只能在一个方向上流动），具有固定的写端与读端

2）只能用于具有亲缘关系的进程之间的通行（父子进程或者兄弟进程）

3）不属于任何文件系统，只存在于内存中

2.FIFO：任何进程间都能通讯，但速度慢

3.消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

4.信号量：不能传递复杂消息，只能用来同步

特点：

1）用于进程间同步，若要在进程间传递数据需要结合共享内存

2）信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作

5.共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

特点：

1）共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

2）因为多个进程可以同时操作，所以需要进行同步。

3）信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

多线程之间的通信

什么是线程间的通信：多个线程协同处理同一资源，线程的任务不相同

1）使用volatile关键字：能够保证所有线程对变量访问的可见性

2）使用synchronized关键字，不仅能保证线程对变量访问的可见性，还能保证排他性

3）使用Thread.join（）

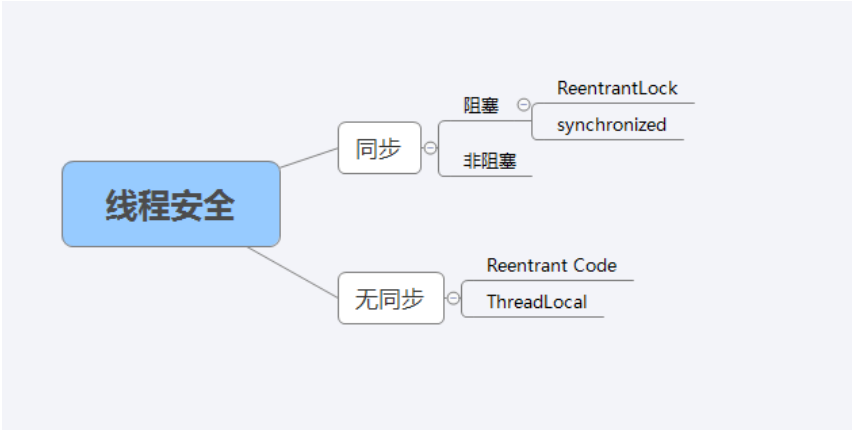
4）使用ThreadLocal线程变量

5）使用等待-通知机制，传统的object.wait（），object.notify（），他们都是属于Object类

6）使用阻塞队列（BlockingQueue）控制线程通信：一般用于生产者-消费者模型

7）使用管道输入/输出流（PipedReader，PipedInputStream）

线程安全如何实现



同步概念：是指在多线程并发访问共享数据时，保证共享数据在同一时刻只被一个线程使用

1：同步方案

1）阻塞同步

加锁 利用Synchronized或者ReenTrantLock来对不安全对象进行加锁，来实现线程执行的串行化，从而保证多线程同时操作对象的安全性

2）非阻塞同步

(最常见的措施就是不断地重试，直到成功为止)。这种方法需要硬件的支持，因为我们需要操作和冲突检测这两个步骤具备原子性。通常这种指令包括CAS SC,FAI TAS等。

2：无需同步方案

1）利用Threadlocal来为每一个线程创造一个共享变量的副本来（副本之间是无关的）避免几个线程同时操作一个对象时发生线程安全问题。

2）使用volatile，通过缓存一致性协议，保证数据在各个线程之中的可见性（正式开发环境下少用）