【博客地址永久迁移到】：http://zhengxiaoyong.me

新博客将不在此处发表！

并发编程之ThreadLocal、Volatile、synchronized、Atomic关键字扫盲

标签： java 并发 多线程

2016-01-21 10:25 1591人阅读 评论(1) 收藏 举报

分类： Android开发之路（68）

目录(?) [+]

前言

对于ThreadLocal、Volatile、synchronized、Atomic这四个关键字，我想一提及到大家肯定都想到的是解决在多线程并发环境下资源的共享问题，但是要细

每一个的特点、区别、应用场景、内部实现等，却可能模糊不清，说不出个所以然来，所以，本文就对这几个关键字做一些作用、特点、实现上的讲解。

1、Atomic

作用

对于原子操作类，**Java**的concurrent并发包中主要为我们提供了这么几个常用的：AtomicInteger、AtomicLong、AtomicBoolean、AtomicReference<T>。

对于原子操作类，最大的特点是在多线程并发操作同一个资源的情况下，使用Lock-Free**算法**来替代锁，这样开销小、速度快，对于原子操作类是采用原

操作指令实现的，从而可以保证操作的原子性。什么是原子性？比如一个操作i++；实际上这是三个原子操作，先把i的值读取、然后修改(+1)、最后写入

i。所以使用Atomic原子类操作数，比如：i++；那么它会在这步操作都完成情况下才允许其它线程再对它进行操作，而这个实现则是通过Lock-Free+原子

作指令来确定的

如：

AtomicInteger类中：

```
    public final int incrementAndGet() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }
```

而关于Lock-Free算法，则是一种新的策略替代锁来保证资源在并发时的完整性的，Lock-Free的实现有三步：

1、循环（for(;;)、while）
2、CAS（CompareAndSet）
3、回退（return、break）

用法

比如在多个线程操作一个count变量的情况下，则可以把count定义为AtomicInteger，如下：

```
public class Counter {
    private AtomicInteger count = new AtomicInteger();

    public int getCount() {
        return count.get();
    }

    public void increment() {
        count.incrementAndGet();
    }
}
```

在每个线程中通过increment()来对count进行计数增加的操作，或者其它一些操作。这样每个线程访问到的将是安全、完整的count。

内部实现

采用Lock-Free算法替代锁+原子操作指令实现并发情况下资源的安全、完整、一致性

2、Volatile

个人资料

Sunzxyong

关注 发私信

访问： 191609次
积分： 3589
等级： BLOG 5
排名： 第5866名

原创： 131篇 转载： 10篇
译文： 0篇 评论： 248条

博客专栏

Android性能优化
文章：16篇
阅读：60575

集：性能优化与代码简洁之道
文章：15篇
阅读：58915

Android数据存储之GreenDAO的使用
文章：5篇
阅读：4693

符合Material Design的设计小栈
文章：15篇
阅读：35143

关于我

作用

Volatile可以看做是一个轻量级的synchronized，它可以在多线程并发的情况下保证变量的“可见性”，什么是可见性？就是在一个线程的工作内存中修改了变量的值，该变量的值立即能回显到主内存中，从而保证所有的线程看到这个变量的值是一致的。所以在处理同步问题上它大显作用，而且它的开销比synchronized小、使用成本更低。

举个栗子：在写单例模式中，除了用静态内部类外，还有一种写法也非常受欢迎，就是Volatile+DCL：

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

这样单例不管在哪个线程中创建的，所有线程都是共享这个单例的。

虽说这个Volatile关键字可以解决多线程环境下的同步问题，不过这也是相对的，因为它不具有操作的原子性，也就是它不适合在对该变量的写操作依赖于变量本身自己。举个最简单的栗子：在进行计数操作时count++，实际是count=count+1;，count最终的值依赖于它本身的值。所以使用volatile修饰的变量在进行这么一系列的操作的时候，就有并发的问题

举个栗子：因为它不具有操作的原子性，有可能1号线程在即将进行写操作时count值为4；而2号线程就恰好获取了写操作之前的值4，所以1号线程在完成的写操作后count值就为5了，而在2号线程中count的值还为4，即使2号线程已经完成了写操作count还是为5，而我们期望的是count最终为6，所以这样就有并发的问题。而如果count换 =num+1；假设num是同步的，那么这样count就没有并发的问题的，只要最终的值不依赖自己本身。

用法

因为volatile不具有操作的原子性，所以如果用volatile修饰的变量在进行依赖于它自身的操作时，就有并发问题，如：count，像下面这样写在并发环境中达不到任何效果的：

```
public class Counter {
    private volatile int count;

    public int getCount(){
        return count;
    }
    public void increment(){
        count++;
    }
}
```

而要想count能在并发环境中保持数据的一致性，则可以在increment()中加synchronized同步锁修饰，改进后的为：

```
public class Counter {
    private volatile int count;

    public int getCount(){
        return count;
    }
    public synchronized void increment(){
        count++;
    }
}
```

内部实现

汇编指令实现

可以看这篇详细了解：Volatile实现原理

3、synchronized

作用

synchronized叫做同步锁，是Lock的一个简化版本，由于是简化版本，那么性能肯定是不如Lock的，不过它操作起来方便，只需要在一个方法或把需要同的代码块包装在它内部，那么这段代码就是同步的了，所有线程对这块区域的代码访问必须先持有锁才能进入，否则则拦截在外面等待正在持有锁的线程处理完毕再获取锁进入，正因为它基于这种阻塞的策略，所以它的性能不太好，但是由于操作上的优势，只需要简单的声明一下即可，而且被它声明的代码块也是具有操作的原子性。

用法

```
    public synchronized void increment(){
```

```
            count++;
    }

    public void increment(){
        synchronized (Counte.class){
            count++;
        }
    }
```

内部实现

重入锁ReentrantLock+一个Condition，所以说是Lock的简化版本，因为一个Lock往往可以对应多个Condition

4、ThreadLocal

作用

关于ThreadLocal，这个类的出现并不是用来解决在多线程并发环境下资源的共享问题的，它和其它三个关键字不一样，其它三个关键字都是从线程外来保证变量的一致性，这样使得多个线程访问的变量具有一致性，可以更好的体现出资源的共享。

而ThreadLocal的设计，**并不是解决资源共享的问题**，而是用来提供线程内的局部变量，这样每个线程都自己管理自己的局部变量，别的线程操作的数据不会对我产生影响，互不影响，所以不存在解决资源共享这么一说，如果是解决资源共享，那么其它线程操作的结果必然我需要获取到，而ThreadLocal则是自己管理自己的，相当于封装在Thread内部了，供线程自己管理。

用法

一般使用ThreadLocal，官方建议我们定义为private static ，至于为什么要定义成静态的，这和内存泄露有关，后面再讲。

它有三个暴露的方法，set、get、remove。

```
public class ThreadLocalDemo {
    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>(){
        @Override
        protected String initialValue() {
            return "hello";
        }
    };
    static class MyRunnable implements Runnable{
        private int num;
        public MyRunnable(int num){
            this.num = num;
        }
        @Override
        public void run() {
            threadLocal.set(String.valueOf(num));
            System.out.println("threadLocalValue:"+threadLocal.get());
        }
    }

    public static void main(String[] args){
        new Thread(new MyRunnable(1));
        new Thread(new MyRunnable(2));
        new Thread(new MyRunnable(3));
    }
}
```

运行结果如下，这些ThreadLocal变量属于线程内部管理的，互不影响：

```
threadLocalValue:2
threadLocalValue:3
threadLocalValue:4
```

对于get方法，在ThreadLocal没有set值得情况下，默认返回null，所有如果要有一个初始值我们可以重写initialValue()方法，在没有set值得情况下调用get则回初始值。

值得注意的一点：ThreadLocal在线程使用完毕后，我们应该手动调用remove方法，移除它内部的值，这样可以防止内存泄露，当然还有设为static。

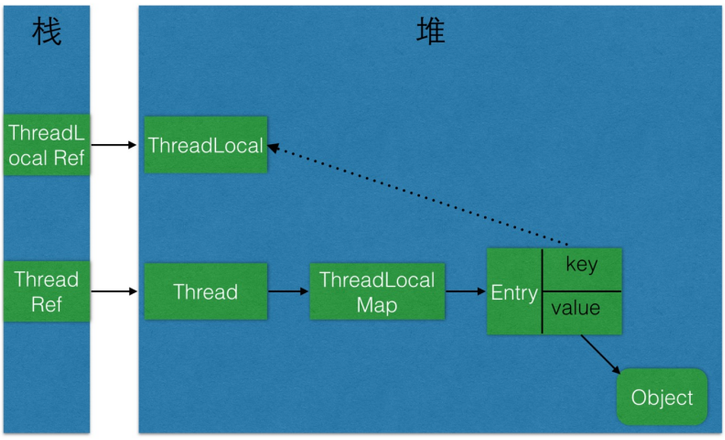
内部实现

ThreadLocal内部有一个静态类ThreadLocalMap，使用到ThreadLocal的线程会与ThreadLocalMap绑定，维护着这个Map对象，而这个ThreadLocalMap的作用是映射当前ThreadLocal对应的值，它key为当前ThreadLocal的弱引用：WeakReference

内存泄露问题

对于ThreadLocal，一直涉及到内存的泄露问题，即当该线程不需要再操作某个ThreadLocal内的值时，应该手动的remove掉，为什么呢？我们来看看ThreadLocal与Thread的联系图：

此图来自网络：



其中虚线表示弱引用，从该图可以看出，一个Thread维持着一个ThreadLocalMap对象，而该Map对象的key又由提供该value的ThreadLocal对象弱引用提供

所以这就有这种情况：

如果ThreadLocal不设为static的，由于Thread的生命周期不可预知，这就导致了当系统gc时将会回收它，而ThreadLocal对象被回收了，此时它对应key必定

null，这就导致了该key对应得value拿不出来了，而value之前被Thread所引用，所以就存在key为null、value存在强引用导致这个Entry回收不了，从而导致

存泄露。

所以避免内存泄露的方法，是对于ThreadLocal要设为static静态的，除了这个，还必须在线程不使用它的值是手动remove掉该ThreadLocal的值，这样Entry

能够在系统gc的时候正常回收，而关于ThreadLocalMap的回收，会在当前Thread销毁之后进行回收。

总结

> 关于Volatile关键字具有可见性，但不具有操作的原子性，而synchronized比volatile对资源的消耗稍微大点，但可以保证变量操作的原子性，保证变量的一致性，最佳实践则是二者结合一起使用。

1、对于synchronized的出现，是解决多线程资源共享的问题，同步机制采用了“以时间换空间”的方式：访问串行化，对象共享化。同步机制是提供一份变量，让所有线程都可以访问。

2、对于Atomic的出现，是通过原子操作指令+Lock-Free完成，从而实现非阻塞式的并发问题。

3、对于Volatile，为多线程资源共享问题解决了部分需求，在非依赖自身的操作的情况下，对变量的改变将对任何线程可见。

4、对于ThreadLocal的出现，并不是解决多线程资源共享的问题，而是用来提供线程内的局部变量，省去参数传递这个不必要的麻烦，ThreadLocal采用了“以空间换时间”的方式：访问并行化，对象独享化。ThreadLocal是为每一个线程都提供了一份独有的变量，各个线程互不影响。

- 上一篇 Android性能优化之Splash页应该这样设计
- 下一篇 关于生产者/消费者/订阅者模式的那些事

我的同类文章

Android开发之路（68）

- 我的第二个独立开发的邮箱类App—“简… 2015-11-09 阅读 1590
- Android中让多个线程顺序执行探究 2015-09-14 阅读 490
- Android开发你不知道的TIPS 2015-10-04 阅读 643
- RecyclerView添加Header和Footer 2015-08-26 阅读 2727
- Android实现RecyclerView侧滑删除和长… 2015-08-24 阅读 2821
- 版本控制—使用Gradle自动管理应用程… 2015-11-07 阅读 758
- Android Studio中创建Kotlin For Android… 2015-09-08 阅读 1117
- Android百分比布局支持库（android-perc… 2015-09-02 阅读 573
- Android下拉上滑显示与隐藏Toolbar另一… 2015-08-26 阅读 1416
- Android事件分发机制 2015-09-17 阅读 868

更多文章

参考知识库

猜你在找

CCIE专家讲解思科认证网络认证…
Java 并发编程中使用 ReentrantLo…

Java之路

Struts实战-使用SSH框架技术开发...

数据结构和算法

软件测试基础

JAVA并发编程3_线程同步之sync...

Java并发编程volatile关键字解析

JAVA并发编程4_线程同步之volat...

转Java并发编程volatile关键字解析

查看评论

任晓天

1楼 2016-01-23 12:27发表

总结的非常全面，

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap