参考：<http://blog.csdn.net/u012734441/article/details/51619751>

[AtomicInteger类的理解与使用](http://blog.csdn.net/u012734441/article/details/51619751)

AtomicInteger类的理解与使用

首先看两段代码，一段是**Integer**的，一段是**AtomicInteger**的，为以下：

public class Sample1 {

private static Integer count = 0;

synchronized public static void increment() {

count++;

}

}

以下是**AtomicInteger**的：

public class Sample2 {

private static AtomicInteger count = new AtomicInteger(0);

public static void increment() {

count.getAndIncrement();

}

}

以上两段代码，在使用**Integer**的时候，必须加上**synchronized**保证不会出现并发线程同时访问的情况，而在**AtomicInteger**中却不用加上**synchronized**，在这里**AtomicInteger**是提供原子操作的，下面就对这进行相应的介绍。

AtomicInteger介绍

**AtomicInteger**是一个提供原子操作的**Integer**类，通过线程安全的方式操作加减。

AtomicInteger使用场景

**AtomicInteger**提供原子操作来进行**Integer**的使用，因此十分适合高并发情况下的使用。

AtomicInteger源码部分讲解

public class AtomicInteger extends Number implements java.io.Serializable {

private static final long serialVersionUID = 6214790243416807050L;

// setup to use Unsafe.compareAndSwapInt for updates

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long valueOffset;

static {

try {

valueOffset = unsafe.objectFieldOffset

(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

private volatile int value;

以上为**AtomicInteger**中的部分源码，在这里说下其中的value，这里value使用了volatile关键字，volatile在这里可以做到的作用是使得多个线程可以共享变量并从主存读和写数据，但是问题在于使用volatile将使得JVM优化失去作用，导致效率较低，所以要在必要的时候使用，因此**AtomicInteger**类不要随意使用，要在使用场景下使用。

AtomicInteger实例使用

以下就是在多线程情况下，使用**AtomicInteger**的一个实例，这段代码是借用IT宅中的一段代码。

public class AtomicTest {

static long randomTime() {

return (long) (Math.random() \* 1000);

}

public static void main(String[] args) {

// 阻塞队列，能容纳100个文件

final BlockingQueue<File> queue = new LinkedBlockingQueue<File>(100);

// 线程池

final ExecutorService exec = Executors.newFixedThreadPool(5);

final File root = new File("D:\\ISO");

// 完成标志

final File exitFile = new File("");

// 原子整型，读个数

// AtomicInteger可以在并发情况下达到原子化更新，避免使用synchronized，而且性能非常高。

final AtomicInteger rc = new AtomicInteger();

// 原子整型，写个数

final AtomicInteger wc = new AtomicInteger();

// 读线程

Runnable read = new Runnable() {

public void run() {

scanFile(root);

scanFile(exitFile);

}

public void scanFile(File file) {

if (file.isDirectory()) {

File[] files = file.listFiles(new FileFilter() {

public boolean accept(File pathname) {

return pathname.isDirectory() || pathname.getPath().endsWith(".iso");

}

});

for (File one : files)

scanFile(one);

} else {

try {

// 原子整型的incrementAndGet方法，以原子方式将当前值加 1，返回更新的值

int index = rc.incrementAndGet();

System.out.println("Read0: " + index + " " + file.getPath());

// 添加到阻塞队列中

queue.put(file);

} catch (InterruptedException e) {

}

}

}

};

// submit方法提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。

exec.submit(read);

// 四个写线程

for (int index = 0; index < 4; index++) {

// write thread

final int num = index;

Runnable write = new Runnable() {

String threadName = "Write" + num;

public void run() {

while (true) {

try {

Thread.sleep(randomTime());

// 原子整型的incrementAndGet方法，以原子方式将当前值加 1，返回更新的值

int index = wc.incrementAndGet();

// 获取并移除此队列的头部，在元素变得可用之前一直等待（如果有必要）。

File file = queue.take();

// 队列已经无对象

if (file == exitFile) {

// 再次添加"标志"，以让其他线程正常退出

queue.put(exitFile);

break;

}

System.out.println(threadName + ": " + index + " " + file.getPath());

} catch (InterruptedException e) {

}

}

}

};

exec.submit(write);

}

exec.shutdown();

}

}

AtomicInteger使用总结

**AtomicInteger**是在使用非阻塞算法实现并发控制，在一些高并发程序中非常适合，但并不能每一种场景都适合，不同场景要使用不同的数值类。踩