BufferedInputstream源码简析

**1.保证并发时内部buf的原子性**

/\*\*

\* The internal buffer array where the data is stored. When necessary,

\* it may be replaced by another array of

\* a different size.

\*/

**protected** **volatile** **byte** buf[];

/\*\*

\* Atomic updater to provide compareAndSet for buf. This is

\* necessary because closes can be asynchronous. We use nullness

\* of buf[] as primary indicator that this stream is closed. (The

\* "in" field is also nulled out on close.)

\*/

**private** **static** **final**

AtomicReferenceFieldUpdater<BufferedInputStream, **byte**[]> ***bufUpdater*** =

AtomicReferenceFieldUpdater.*newUpdater*

(BufferedInputStream.**class**, **byte**[].**class**, "buf");

**预备知识：**

可以看到，BufferedInputStream内部维护了一个byte[] 字节数组，流 进来后，先被放到buf中缓存，read的时候从buf中read数据。

这时候注意区分read(byte[] buff)这个方法，它从buf中读出数据，输送到buff中

我们知道并发访问时，如果有多个线程同时读取buf，那么原子性无法得到保证，因为首先每个线程都有自己的线程栈存放自己修改的数据，而且每个线程同时修改后无限制地各自提交，那么会有数据丢失的问题。

基于这两个问题，设计者

先用**protected**使子类无法继承修改该buf，保证定义的唯一性；

然后用**volatile**禁用了线程栈，保证每个线程读取时都是最新的值；

最后使用**AtomicReferenceFieldUpdater**这个类对**指定类**的volatile成员进行原子更新，基于反射原理（本例中这个成员（数组）**是**引用类型，另外还有专门针对基本类型的，比如AtomICLongFieldUpdater）

这里AtomicReferenceFieldUpdater主要提供了compareAndSet方法供使用，用来替代setter进行set操作，返回boolean表示成败

**byte** nbuf[] = **new** **byte**[nsz];

System.*arraycopy*(buffer, 0, nbuf, 0, pos);

**if** (!***bufUpdater***.compareAndSet(**this**, buffer, nbuf)) {

// Can't replace buf if there was an async close.

// Note: This would need to be changed if fill()

// is ever made accessible to multiple threads.

// But for now, the only way CAS can fail is via close.

// assert buf == null;

**throw** **new** IOException("Stream closed");

}

buffer = nbuf;

还有close方法，

**public** **void** close() **throws** IOException {

**byte**[] buffer;

**while** ( (buffer = buf) != **null**) {

**if** (***bufUpdater***.compareAndSet(**this**, buffer, **null**)) {

InputStream input = in;

in = **null**;

**if** (input != **null**)

input.close();

**return**;

}

// Else retry in case a new buf was CASed in fill()

}

}

**2.私有方法只能在类内访问**！！，在别的地方创建实例访问是不行的，这相当于出了类了。典型的类内访问就是main方法调用（当然，非静态的方法一样要创建实例）

像BufferedInputstream中的 private byte[] getBufIfOpen()方法，只能在类内调用，用以辅助其他方法获得缓存。

**3.BufferedInputstream大部分供用户使用的方法都套上了synchronized关键字**，除了close和markSupported方法

这表明**BufferedInputstream是线程安全**的。

**public** **synchronized** **int** read() **throws** IOException {

**if** (pos >= count) {

fill();

**if** (pos >= count)

**return** -1;

}

**return** getBufIfOpen()[pos++] & 0xff;

}

**public** **synchronized** **int** read(**byte** b[], **int** off, **int** len)

**throws** IOException

{

getBufIfOpen(); // Check for closed stream

**if** ((off | len | (off + len) | (b.length - (off + len))) < 0) {

**throw** **new** IndexOutOfBoundsException();

} **else** **if** (len == 0) {

**return** 0;

}

**int** n = 0;

**for** (;;) {

**int** nread = read1(b, off + n, len - n);

**if** (nread <= 0)

**return** (n == 0) ? nread : n;

n += nread;

**if** (n >= len)

**return** n;

// if not closed but no bytes available, return

InputStream input = in;

**if** (input != **null** && input.available() <= 0)

**return** n;

}

}

**public** **synchronized** **long** skip(**long** n) **throws** IOException {

getBufIfOpen(); // Check for closed stream

**if** (n <= 0) {

**return** 0;

}

**long** avail = count - pos;

**if** (avail <= 0) {

// If no mark position set then don't keep in buffer

**if** (markpos <0)

**return** getInIfOpen().skip(n);

// Fill in buffer to save bytes for reset

fill();

avail = count - pos;

**if** (avail <= 0)

**return** 0;

}

**long** skipped = (avail < n) ? avail : n;

pos += skipped;

**return** skipped;

}

**public** **synchronized** **int** available() **throws** IOException {

**int** n = count - pos;

**int** avail = getInIfOpen().available();

**return** n > (Integer.***MAX\_VALUE*** - avail)

? Integer.***MAX\_VALUE***

: n + avail;

}

**public** **synchronized** **void** mark(**int** readlimit) {

marklimit = readlimit;

markpos = pos;

}

**public** **synchronized** **void** reset() **throws** IOException {

getBufIfOpen(); // Cause exception if closed

**if** (markpos < 0)

**throw** **new** IOException("Resetting to invalid mark");

pos = markpos;

}

SE源码看起来真是舒服

**4.BufferedInputstream使用fill方法将流填充到buf中**

**private** **void** fill() **throws** IOException {

**byte**[] buffer = getBufIfOpen();

**if** (markpos < 0)

pos = 0; /\* no mark: throw away the buffer \*/

**else** **if** (pos >= buffer.length) /\* no room left in buffer \*/

**if** (markpos > 0) { /\* can throw away early part of the buffer \*/

**int** sz = pos - markpos;

System.*arraycopy*(buffer, markpos, buffer, 0, sz);

pos = sz;

markpos = 0;

} **else** **if** (buffer.length >= marklimit) {

markpos = -1; /\* buffer got too big, invalidate mark \*/

pos = 0; /\* drop buffer contents \*/

} **else** **if** (buffer.length >= *MAX\_BUFFER\_SIZE*) {

**throw** **new** OutOfMemoryError("Required array size too large");

} **else** { /\* grow buffer \*/

**int** nsz = (pos <= *MAX\_BUFFER\_SIZE* - pos) ?

pos \* 2 : *MAX\_BUFFER\_SIZE*;

**if** (nsz > marklimit)

nsz = marklimit;

**byte** nbuf[] = **new** **byte**[nsz];

System.*arraycopy*(buffer, 0, nbuf, 0, pos);

**if** (!***bufUpdater***.compareAndSet(**this**, buffer, nbuf)) {

// Can't replace buf if there was an async close.

// Note: This would need to be changed if fill()

// is ever made accessible to multiple threads.

// But for now, the only way CAS can fail is via close.

// assert buf == null;

**throw** **new** IOException("Stream closed");

}

buffer = nbuf;

}

count = pos;

**int** n = getInIfOpen().read(buffer, pos, buffer.length - pos);

**if** (n > 0)

count = n + pos;

}

提取出关键的：

（1）**byte**[] buffer = getBufIfOpen();

取到默认buf的引用的值，传给buffer，得到的是buf的一个镜像

（2）System.*arraycopy*(buffer, 0, nbuf, 0, pos);

经过一番判断，先将buffer的值赋值给nbuf

（

[**arraycopy**](mk:@MSITStore:C:\Users\Administrator\Desktop\API_JDK_1_6.CHM::/java/lang/System.html#arraycopy(java.lang.Object, int, java.lang.Object, int, int))([Object](mk:@MSITStore:C:\Users\Administrator\Desktop\API_JDK_1_6.CHM::/java/lang/Object.html) src, int srcPos, [Object](mk:@MSITStore:C:\Users\Administrator\Desktop\API_JDK_1_6.CHM::/java/lang/Object.html) dest, int destPos, int length)   
          从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。

）

（3）**if** (!***bufUpdater***.compareAndSet(**this**, buffer, nbuf))

this是BufferedInputstream的buf，compareAndSet将nbuf赋给buf，如果buf赋值后值是buffer，那么原子更新成功

（4）buffer = nbuf;

最后来这一句，在已经提交数据给buf后，nbuf的任务完成，将数据返还给buffer，buffer就是buf，这里主要是要和临时缓冲区长度同步**byte** nbuf[] = **new** **byte**[nsz];

因为读到最后，可能长度就没这么长了，避免了读空值；

**（5）此时buffer的长度就是经过筛选的了，不会读到空值**

当然，在no room left in buffer的情况下，最后还要流中的数据督导buffer中，

**int** n = getInIfOpen().read(buffer, pos, buffer.length - pos);

**这一步才是实际地从流中读出数据到缓存**

上面的都是在判断符不符合读流的条件，目标缓存的规格是怎样的。