**花1K内存实现高效I/O的RandomAccessFile类**

**通过扩展RandomAccessFile类使之具备Buffer改善I/O性能**

JAVA的文件随机存取类（RandomAccessFile）的I/O效率较低。通过分析其中原因，提出解决方案。逐步展示如何创建具备缓存读写能力的文件随机存取类，并进行了优化。通过与其它文件访问类的性能对比，证明了其实用价值。

1http://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](http://www.ibm.com/developerworks/cn/java/l-javaio/#icomments)

[崔志翔](http://www.ibm.com/developerworks/cn/java/l-javaio/#authorN10019) ([bladeinco@citiz.net](mailto:bladeinco@citiz.net?subject=%E8%8A%B11K%E5%86%85%E5%AD%98%E5%AE%9E%E7%8E%B0%E9%AB%98%E6%95%88I/O%E7%9A%84RandomAccessFile%E7%B1%BB&cc=bladeinco@citiz.net)), 上海颐东网络信息有限公司产品部经理

2002 年 11 月 18 日

* [expand](http://www.ibm.com/developerworks/cn/java/l-javaio/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-l-javaio-_-sidebar)

**主体：**

目前最流行的J2SDK版本是1.3系列。使用该版本的开发人员需文件随机存取，就得使用RandomAccessFile类。其I/O性能较之其它常用开发语言的同类性能差距甚远，严重影响程序的运行效率。

开发人员迫切需要提高效率，下面分析RandomAccessFile等文件类的源代码，找出其中的症结所在，并加以改进优化，创建一个"性/价比"俱佳的随机文件访问类BufferedRandomAccessFile。

**在改进之前先做一个基本测试：逐字节COPY一个12兆的文件（这里牵涉到读和写）。**

| **读** | **写** | **耗用时间（秒）** |
| --- | --- | --- |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |

我们可以看到两者差距约32倍，RandomAccessFile也太慢了。先看看两者关键部分的源代码，对比分析，找出原因。

**1．1．[RandomAccessFile]**

public class RandomAccessFile implements DataOutput, DataInput {

public final byte readByte() throws IOException {

int ch = this.read();

if (ch < 0)

throw new EOFException();

return (byte)(ch);

}

public native int read() throws IOException;

public final void writeByte(int v) throws IOException {

write(v);

}

public native void write(int b) throws IOException;

}

可见，RandomAccessFile每读/写一个字节就需对磁盘进行一次I/O操作。

**1．2．[BufferedInputStream]**

public class BufferedInputStream extends FilterInputStream {

private static int defaultBufferSize = 2048;

protected byte buf[]; // 建立读缓存区

public BufferedInputStream(InputStream in, int size) {

super(in);

if (size <= 0) {

throw new IllegalArgumentException("Buffer size <= 0");

}

buf = new byte[size];

}

public synchronized int read() throws IOException {

ensureOpen();

if (pos >= count) {

fill();

if (pos >= count)

return -1;

}

return buf[pos++] & 0xff; // 直接从BUF[]中读取

}

private void fill() throws IOException {

if (markpos < 0)

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

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

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

int sz = pos - markpos;

System.arraycopy(buf, markpos, buf, 0, sz);

pos = sz;

markpos = 0;

} else if (buf.length >= marklimit) {

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

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

} else { /\* grow buffer \*/

int nsz = pos \* 2;

if (nsz > marklimit)

nsz = marklimit;

byte nbuf[] = new byte[nsz];

System.arraycopy(buf, 0, nbuf, 0, pos);

buf = nbuf;

}

count = pos;

int n = in.read(buf, pos, buf.length - pos);

if (n > 0)

count = n + pos;

}

}

**1．3．[BufferedOutputStream]**

public class BufferedOutputStream extends FilterOutputStream {

protected byte buf[]; // 建立写缓存区

public BufferedOutputStream(OutputStream out, int size) {

super(out);

if (size <= 0) {

throw new IllegalArgumentException("Buffer size <= 0");

}

buf = new byte[size];

}

public synchronized void write(int b) throws IOException {

if (count >= buf.length) {

flushBuffer();

}

buf[count++] = (byte)b; // 直接从BUF[]中读取

}

private void flushBuffer() throws IOException {

if (count > 0) {

out.write(buf, 0, count);

count = 0;

}

}

}

可见，Buffered I/O putStream每读/写一个字节，若要操作的数据在BUF中，就直接对内存的buf[]进行读/写操作；否则从磁盘相应位置填充buf[]，再直接对内存的buf[]进行读/写操作，绝大部分的读/写操作是对内存buf[]的操作。

**1．3．小结**

内存存取时间单位是纳秒级（10E-9），磁盘存取时间单位是毫秒级（10E-3）， 同样操作一次的开销，内存比磁盘快了百万倍。理论上可以预见，即使对内存操作上万次，花费的时间也远少对于磁盘一次I/O的开销。 显然后者是通过增加位于内存的BUF存取，减少磁盘I/O的开销，提高存取效率的，当然这样也增加了BUF控制部分的开销。从实际应用来看，存取效率提高了32倍。

[**回页首**](http://www.ibm.com/developerworks/cn/java/l-javaio/#ibm-pcon)

**根据1.3得出的结论，现试着对RandomAccessFile类也加上缓冲读写机制。**

随机访问类与顺序类不同，前者是通过实现DataInput/DataOutput接口创建的，而后者是扩展FilterInputStream/FilterOutputStream创建的，不能直接照搬。

**2．1．开辟缓冲区BUF[默认：1024字节]，用作读/写的共用缓冲区。**

**2．2．先实现读缓冲。**

读缓冲逻辑的基本原理：

A 欲读文件POS位置的一个字节。

B 查BUF中是否存在？若有，直接从BUF中读取，并返回该字符BYTE。

C 若没有，则BUF重新定位到该POS所在的位置并把该位置附近的BUFSIZE的字节的文件内容填充BUFFER，返回B。

以下给出关键部分代码及其说明：

public class BufferedRandomAccessFile extends RandomAccessFile {

// byte read(long pos)：读取当前文件POS位置所在的字节

// bufstartpos、bufendpos代表BUF映射在当前文件的首/尾偏移地址。

// curpos指当前类文件指针的偏移地址。

public byte read(long pos) throws IOException {

if (pos < this.bufstartpos || pos > this.bufendpos ) {

this.flushbuf();

this.seek(pos);

if ((pos < this.bufstartpos) || (pos > this.bufendpos))

throw new IOException();

}

this.curpos = pos;

return this.buf[(int)(pos - this.bufstartpos)];

}

// void flushbuf()：bufdirty为真，把buf[]中尚未写入磁盘的数据，写入磁盘。

private void flushbuf() throws IOException {

if (this.bufdirty == true) {

if (super.getFilePointer() != this.bufstartpos) {

super.seek(this.bufstartpos);

}

super.write(this.buf, 0, this.bufusedsize);

this.bufdirty = false;

}

}

// void seek(long pos)：移动文件指针到pos位置，并把buf[]映射填充至POS

所在的文件块。

public void seek(long pos) throws IOException {

if ((pos < this.bufstartpos) || (pos > this.bufendpos)) { // seek pos not in buf

this.flushbuf();

if ((pos >= 0) && (pos <= this.fileendpos) && (this.fileendpos != 0))

{ // seek pos in file (file length > 0)

this.bufstartpos = pos \* bufbitlen / bufbitlen;

this.bufusedsize = this.fillbuf();

} else if (((pos == 0) && (this.fileendpos == 0))

|| (pos == this.fileendpos + 1))

{ // seek pos is append pos

this.bufstartpos = pos;

this.bufusedsize = 0;

}

this.bufendpos = this.bufstartpos + this.bufsize - 1;

}

this.curpos = pos;

}

// int fillbuf()：根据bufstartpos，填充buf[]。

private int fillbuf() throws IOException {

super.seek(this.bufstartpos);

this.bufdirty = false;

return super.read(this.buf);

}

}

至此缓冲读基本实现，逐字节COPY一个12兆的文件（这里牵涉到读和写，用BufferedRandomAccessFile试一下读的速度）：

| **读** | **写** | **耗用时间（秒）** |
| --- | --- | --- |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |

可见速度显著提高，与BufferedInputStream+DataInputStream不相上下。

**2．3．实现写缓冲。**

写缓冲逻辑的基本原理：

A欲写文件POS位置的一个字节。

B 查BUF中是否有该映射？若有，直接向BUF中写入，并返回true。

C若没有，则BUF重新定位到该POS所在的位置，并把该位置附近的 BUFSIZE字节的文件内容填充BUFFER，返回B。

下面给出关键部分代码及其说明：

// boolean write(byte bw, long pos)：向当前文件POS位置写入字节BW。

// 根据POS的不同及BUF的位置：存在修改、追加、BUF中、BUF外等情

况。在逻辑判断时，把最可能出现的情况，最先判断，这样可提高速度。

// fileendpos：指示当前文件的尾偏移地址，主要考虑到追加因素

public boolean write(byte bw, long pos) throws IOException {

if ((pos >= this.bufstartpos) && (pos <= this.bufendpos)) {

// write pos in buf

this.buf[(int)(pos - this.bufstartpos)] = bw;

this.bufdirty = true;

if (pos == this.fileendpos + 1) { // write pos is append pos

this.fileendpos++;

this.bufusedsize++;

}

} else { // write pos not in buf

this.seek(pos);

if ((pos >= 0) && (pos <= this.fileendpos) && (this.fileendpos != 0))

{ // write pos is modify file

this.buf[(int)(pos - this.bufstartpos)] = bw;

} else if (((pos == 0) && (this.fileendpos == 0))

|| (pos == this.fileendpos + 1)) { // write pos is append pos

this.buf[0] = bw;

this.fileendpos++;

this.bufusedsize = 1;

} else {

throw new IndexOutOfBoundsException();

}

this.bufdirty = true;

}

this.curpos = pos;

return true;

}

至此缓冲写基本实现，逐字节COPY一个12兆的文件，（这里牵涉到读和写，结合缓冲读，用BufferedRandomAccessFile试一下读/写的速度）：

| **读** | **写** | **耗用时间（秒）** |
| --- | --- | --- |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |

可见综合读/写速度已超越BufferedInput/OutputStream+DataInput/OutputStream。

[**回页首**](http://www.ibm.com/developerworks/cn/java/l-javaio/#ibm-pcon)

**优化BufferedRandomAccessFile。**

优化原则：

* 调用频繁的语句最需要优化，且优化的效果最明显。
* 多重嵌套逻辑判断时，最可能出现的判断，应放在最外层。
* 减少不必要的NEW。

这里举一典型的例子：

public void seek(long pos) throws IOException {

...

this.bufstartpos = pos \* bufbitlen / bufbitlen;

// bufbitlen指buf[]的位长，例：若bufsize=1024，则bufbitlen=10。

...

}

seek函数使用在各函数中，调用非常频繁，上面加重的这行语句根据pos和bufsize确定buf[]对应当前文件的映射位置，用"\*"、"/"确定，显然不是一个好方法。

优化一：this.bufstartpos = (pos << bufbitlen) >> bufbitlen;

优化二：this.bufstartpos = pos & bufmask; // this.bufmask = ~((long)this.bufsize - 1);

两者效率都比原来好，但后者显然更好，因为前者需要两次移位运算、后者只需一次逻辑与运算（bufmask可以预先得出）。

至此优化基本实现，逐字节COPY一个12兆的文件，（这里牵涉到读和写，结合缓冲读，用优化后BufferedRandomAccessFile试一下读/写的速度）：

| **读** | **写** | **耗用时间（秒）** |
| --- | --- | --- |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |
| BufferedRandomAccessFile优 | BufferedRandomAccessFile优 | 2.197 |

可见优化尽管不明显，还是比未优化前快了一些，也许这种效果在老式机上会更明显。

以上比较的是顺序存取，即使是随机存取，在绝大多数情况下也不止一个BYTE，所以缓冲机制依然有效。而一般的顺序存取类要实现随机存取就不怎么容易了。

[**回页首**](http://www.ibm.com/developerworks/cn/java/l-javaio/#ibm-pcon)

**需要完善的地方**

提供文件追加功能：

public boolean append(byte bw) throws IOException {

return this.write(bw, this.fileendpos + 1);

}

提供文件当前位置修改功能：

public boolean write(byte bw) throws IOException {

return this.write(bw, this.curpos);

}

返回文件长度（由于BUF读写的原因，与原来的RandomAccessFile类有所不同）：

public long length() throws IOException {

return this.max(this.fileendpos + 1, this.initfilelen);

}

返回文件当前指针（由于是通过BUF读写的原因，与原来的RandomAccessFile类有所不同）：

public long getFilePointer() throws IOException {

return this.curpos;

}

提供对当前位置的多个字节的缓冲写功能：

public void write(byte b[], int off, int len) throws IOException {

long writeendpos = this.curpos + len - 1;

if (writeendpos <= this.bufendpos) { // b[] in cur buf

System.arraycopy(b, off, this.buf, (int)(this.curpos - this.bufstartpos),

len);

this.bufdirty = true;

this.bufusedsize = (int)(writeendpos - this.bufstartpos + 1);

} else { // b[] not in cur buf

super.seek(this.curpos);

super.write(b, off, len);

}

if (writeendpos > this.fileendpos)

this.fileendpos = writeendpos;

this.seek(writeendpos+1);

}

public void write(byte b[]) throws IOException {

this.write(b, 0, b.length);

}

提供对当前位置的多个字节的缓冲读功能：

public int read(byte b[], int off, int len) throws IOException {

long readendpos = this.curpos + len - 1;

if (readendpos <= this.bufendpos && readendpos <= this.fileendpos ) {

// read in buf

System.arraycopy(this.buf, (int)(this.curpos - this.bufstartpos),

b, off, len);

} else { // read b[] size > buf[]

if (readendpos > this.fileendpos) { // read b[] part in file

len = (int)(this.length() - this.curpos + 1);

}

super.seek(this.curpos);

len = super.read(b, off, len);

readendpos = this.curpos + len - 1;

}

this.seek(readendpos + 1);

return len;

}

public int read(byte b[]) throws IOException {

return this.read(b, 0, b.length);

}

public void setLength(long newLength) throws IOException {

if (newLength > 0) {

this.fileendpos = newLength - 1;

} else {

this.fileendpos = 0;

}

super.setLength(newLength);

}

public void close() throws IOException {

this.flushbuf();

super.close();

}

至此完善工作基本完成，试一下新增的多字节读/写功能，通过同时读/写1024个字节，来COPY一个12兆的文件，（这里牵涉到读和写，用完善后BufferedRandomAccessFile试一下读/写的速度）：

| **读** | **写** | **耗用时间（秒）** |
| --- | --- | --- |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |
| BufferedRandomAccessFile优 | BufferedRandomAccessFile优 | 2.197 |
| BufferedRandomAccessFile完 | BufferedRandomAccessFile完 | 0.401 |

[**回页首**](http://www.ibm.com/developerworks/cn/java/l-javaio/#ibm-pcon)

**与JDK1.4新类MappedByteBuffer+RandomAccessFile的对比？**

JDK1.4提供了NIO类 ，其中MappedByteBuffer类用于映射缓冲，也可以映射随机文件访问，可见JAVA设计者也看到了RandomAccessFile的问题，并加以改进。怎么通过MappedByteBuffer+RandomAccessFile拷贝文件呢？下面就是测试程序的主要部分：

RandomAccessFile rafi = new RandomAccessFile(SrcFile, "r");

RandomAccessFile rafo = new RandomAccessFile(DesFile, "rw");

FileChannel fci = rafi.getChannel();

FileChannel fco = rafo.getChannel();

long size = fci.size();

MappedByteBuffer mbbi = fci.map(FileChannel.MapMode.READ\_ONLY, 0, size);

MappedByteBuffer mbbo = fco.map(FileChannel.MapMode.READ\_WRITE, 0, size);

long start = System.currentTimeMillis();

for (int i = 0; i < size; i++) {

byte b = mbbi.get(i);

mbbo.put(i, b);

}

fcin.close();

fcout.close();

rafi.close();

rafo.close();

System.out.println("Spend: "+(double)(System.currentTimeMillis()-start) / 1000 + "s");

试一下JDK1.4的映射缓冲读/写功能，逐字节COPY一个12兆的文件，（这里牵涉到读和写）：

| **读** | **写** | **耗用时间（秒）** |
| --- | --- | --- |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |
| BufferedRandomAccessFile优 | BufferedRandomAccessFile优 | 2.197 |
| BufferedRandomAccessFile完 | BufferedRandomAccessFile完 | 0.401 |
| MappedByteBuffer+ RandomAccessFile | MappedByteBuffer+ RandomAccessFile | 1.209 |

确实不错，看来JDK1.4比1.3有了极大的进步。如果以后采用1.4版本开发软件时，需要对文件进行随机访问，建议采用MappedByteBuffer+RandomAccessFile的方式。但鉴于目前采用JDK1.3及以前的版本开发的程序占绝大多数的实际情况，如果您开发的JAVA程序使用了RandomAccessFile类来随机访问文件，并因其性能不佳，而担心遭用户诟病，请试用本文所提供的BufferedRandomAccessFile类，不必推翻重写，只需IMPORT 本类，把所有的RandomAccessFile改为BufferedRandomAccessFile，您的程序的性能将得到极大的提升，您所要做的就这么简单。

[**回页首**](http://www.ibm.com/developerworks/cn/java/l-javaio/#ibm-pcon)

**未来的考虑**

读者可在此基础上建立多页缓存及缓存淘汰机制，以应付对随机访问强度大的应用。

**参考资料**

SUN JDK1.3/1.4 SRC。

* [*源代码*](http://www.ibm.com/developerworks/cn/java/l-javaio/src.zip)

**条评论**

请 [登录](http://www.ibm.com/developerworks/cn/java/l-javaio/) 或 [注册](http://www.ibm.com/developerworks/dwwi/jsp/ssoregister.jsp?d=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Fjava%2Fl-javaio%2F%23icomments&lang=cn&a=dwmav) 后发表评论。

**添加评论:**

注意：评论中不支持 HTML 语法

窗体顶端



有新评论时提醒我剩余 1000 字符

窗体底端

窗体顶端

**共有评论 (1)**

窗体底端

你好，请问你那seek(long pos)方法，为什么那么定位，有点看不出来什么意思，还有就是，你这个方法要表现出高效的话，应该是在特定场景下吧？比如是在反复读取某个位置附近的字节，这样才能体现出你的代码意义，不知道我说的对不，希望能解答一下