liuming_1992

Java Properties类源码分析

一、Properties类介绍

java.util.Properties继承自java.util.Hashtable，从jdk1.1版本开始，Properties的实现基本上就没有什么大的变动。从http://docs.oracle.com/javase/7/docs/api/的jdk7的官方api文档中我们可以看到对Properties类的介绍。Properties class是一个持久化的属性保存对象，可以将属性内容写出到stream中或者从stream中读取属性内容，在底层的Hashtable中，每一对属性的key和value都是按照string类型来保存的。 Properties可以将其他的Properties对象作为默认的值，Properties继承自Hashtable，所以Hashtable的所有方法Properties对象均可以访问。

Properties支持文本方式和xml方式的数据存储。在文本方式中，格式为key:value,其中分隔符可以是：冒号(:)、等号(=)、空格。其中空格可以作为key的结束，同时获取的值回将分割符号两端的空格去掉。

Properties只支持1对1模式的属性设置，而且不支持多层多级属性设置。

二、Properties类属性

protected Properties defaults：包含默认values的Properties对象，默认为null。我们在找不到对应key的情况下，就回递归的从这个默认列表中里面来找。

```
/**
 * A property list that contains default values for any keys not
 * found in this property list.
 *
 * @serial
 */
protected Properties defaults;
```

三、初始化方法

Properties提供两种方式来创建Properties对象，第一种是不指定默认values对象的创建方法，另外一种是指定默认values对象的创建方法。但是此时是没有加载属性值的，加载key/value属性必须通过专门的方法来加载。

```
/**
 * Creates an empty property list with no default values.
 */
public Properties() {
    this(null);
}

/**
 * Creates an empty property list with the specified defaults.
 *
 * @param   defaults   the defaults.
 */
public Properties(Properties defaults) {
    this.defaults = defaults;
}
```

四、常用方法

getProperty(String):根据指定的key获取对应的属性value值，如果在自身的存储集合中没有找到对应的key，那么就直接到默认的defaults属性指定的Properties中获取属性值。

```
/**
 * Searches for the property with the specified key in this property list.
 * If the key is not found in this property list, the default property list,
 * and its defaults, recursively, are then checked. The method returns
 * <code>null</code> if the property is not found.
```

昵称： liuming_1992
园龄： 6年8个月
粉丝： 38
关注： 0
+加关注

< 2021年9月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

常用链接

- 我的随笔
- 我的评论
- 我的参与
- 最新评论
- 我的标签

我的标签

- hadoop(16)
- Kafka(10)
- Data structure(6)
- Spark(4)
- MongoDB(4)
- 数据挖掘(4)
- java web(4)
- java多线程(3)
- solr(3)
- linux(3)
- 更多

随笔档案

- 2017年3月(2)
- 2017年2月(16)
- 2016年1月(3)
- 2015年12月(2)

```
     *
     * @param   key   the property key.
     * @return  the value in this property list with the specified key value.
     * @see     #setProperty
     * @see     #defaults
     */
    public String getProperty(String key) {
        Object oval = super.get(key);
        String sval = (oval instanceof String) ? (String)oval : null;
        return ((sval == null) && (defaults != null)) ? defaults.getProperty(key) : sval;
    }
```

getProperty(String, String):当getProperty(String)方法返回值为null的时候，返回给定的默认值，而不是返回null。

```
    /**
     * Searches for the property with the specified key in this property list.
     * If the key is not found in this property list, the default property list,
     * and its defaults, recursively, are then checked. The method returns the
     * default value argument if the property is not found.
     *
     * @param   key            the hashtable key.
     * @param   defaultValue   a default value.
     *
     * @return  the value in this property list with the specified key value.
     * @see     #setProperty
     * @see     #defaults
     */
    public String getProperty(String key, String defaultValue) {
        String val = getProperty(key);
        return (val == null) ? defaultValue : val;
    }
```

load(InputStream):从byte stream中加载key/value键值对，要求所有的key/value键值对是按行存储，同时是用ISO-8859-1编译的。

```
    /**
     * Reads a property list (key and element pairs) from the input
     * byte stream. The input stream is in a simple line-oriented
     * format as specified in
     * {@link #load(java.io.Reader) load(Reader)} and is assumed to use
     * the ISO 8859-1 character encoding; that is each byte is one Latin1
     * character. Characters not in Latin1, and certain special characters,
     * are represented in keys and elements using Unicode escapes as defined in
     * section 3.3 of
     * <cite>The Java&trade; Language Specification</cite>.
     * <p>
     * The specified stream remains open after this method returns.
     *
     * @param      inStream   the input stream.
     * @exception  IOException  if an error occurred when reading from the
     *             input stream.
     * @throws     IllegalArgumentException if the input stream contains a
     *             malformed Unicode escape sequence.
     * @since 1.2
     */
    public synchronized void load(InputStream inStream) throws IOException {
        load0(new LineReader(inStream));
    }
```

load(Reader):从字符流中加载key/value键值对，要求所有的键值对都是按照行来存储的。

```
    /**
     * Reads a property list (key and element pairs) from the input
     * character stream in a simple line-oriented format.
     * <p>
     * Properties are processed in terms of lines. There are two
     * kinds of line, <i>natural lines</i> and <i>logical lines</i>.
     * A natural line is defined as a line of
     * characters that is terminated either by a set of line terminator
     * characters (<code>\n</code> or <code>\r</code> or <code>\r\n</code>)
     * or by the end of the stream. A natural line may be either a blank line,
     * a comment line, or hold all or some of a key-element pair. A logical
```

2015年11月(2)
2015年10月(4)
2015年9月(6)
2015年8月(13)
2015年3月(1)
2015年1月(13)
2014年12月(2)

最新评论

1. Re:[Kafka] - Kafka Java Consumer实现(一)
优秀啊。依赖里面必须用0.8.2.1的版本。否则凉凉。
很好用，谢谢
--West_Jing

2. Re:[Kafka] - Kafka Java Producer代码实现
222
--诸葛烤鱼2019

3. Re:[Kafka] - Kafka Java Producer代码实现
111
--诸葛烤鱼2019

4. Re:[Linux] - xxx 不在 sudoers 文件中。此事将被报告。
能问一下，报告的内容在哪?
--MrWu08

5. Re:[Kafka] - Kafka Java Consumer实现(二)
代码在ConsumerKafkaStreamProcesser类中，while (iter.hasNext()) {}进不去，并且后面的打印语句也没有结果
--Jelly旺

阅读排行榜

1. [Kafka] - Kafka基本操作命令(44412)
2. [Netty] - Netty入门(最简单的Netty客户端/服务器程序)(34582)
3. [Kafka] - Kafka Java Consumer实现(一)(32326)
4. [Kafka] - Kafka Java Producer代码实现(23407)
5. [Linux] - xxx 不在 sudoers 文件中。此事将被报告。(22794)

评论排行榜

1. [Kafka] - Kafka Java Consumer实现(一)(4)
2. [Kafka] - Kafka Java Producer代码实现(2)
3. [Kafka] - Kafka Java Consumer实现(二)(2)
4. [Linux] - xxx 不在 sudoers 文件中。此事将被报告。(1)
5. [Netty] - Netty入门(最简单的Netty客户端/服务器程序)(1)

推荐排行榜

1. [Kafka] - Kafka Java Consumer实现(二)(3)
2. [Kafka] - Kafka Java Consumer实现(一)(3)
3. [Spark] - HashPartitioner & RangePartitioner区别(3)
4. [Kafka] - Kafka Java Producer代码实现(2)
5. [Kafka] - Kafka基本概念介绍(2)

```
     * line holds all the data of a key-element pair, which may be spread
     * out across several adjacent natural lines by escaping
     * the line terminator sequence with a backslash character
     * <code>\</code>.  Note that a comment line cannot be extended
     * in this manner; every natural line that is a comment must have
     * its own comment indicator, as described below. Lines are read from
     * input until the end of the stream is reached.
     *
     * <p>
     * A natural line that contains only white space characters is
     * considered blank and is ignored.  A comment line has an ASCII
     * <code>'#'</code> or <code>'!'</code> as its first non-white
     * space character; comment lines are also ignored and do not
     * encode key-element information.  In addition to line
     * terminators, this format considers the characters space
     * (<code>' '</code>, <code>'&#92;u0020'</code>), tab
     * (<code>'\t'</code>, <code>'&#92;u0009'</code>), and form feed
     * (<code>'\f'</code>, <code>'&#92;u000C'</code>) to be white
     * space.
     *
     * <p>
     * If a logical line is spread across several natural lines, the
     * backslash escaping the line terminator sequence, the line
     * terminator sequence, and any white space at the start of the
     * following line have no affect on the key or element values.
     * The remainder of the discussion of key and element parsing
     * (when loading) will assume all the characters constituting
     * the key and element appear on a single natural line after
     * line continuation characters have been removed.  Note that
     * it is <i>not</i> sufficient to only examine the character
     * preceding a line terminator sequence to decide if the line
     * terminator is escaped; there must be an odd number of
     * contiguous backslashes for the line terminator to be escaped.
     * Since the input is processed from left to right, a
     * non-zero even number of 2<i>n</i> contiguous backslashes
     * before a line terminator (or elsewhere) encodes <i>n</i>
     * backslashes after escape processing.
     *
     * <p>
     * The key contains all of the characters in the line starting
     * with the first non-white space character and up to, but not
     * including, the first unescaped <code>'='</code>,
     * <code>':'</code>, or white space character other than a line
     * terminator. All of these key termination characters may be
     * included in the key by escaping them with a preceding backslash
     * character; for example,<p>
     *
     * <code>\:\=</code><p>
     *
     * would be the two-character key <code>":="</code>.  Line
     * terminator characters can be included using <code>\r</code> and
     * <code>\n</code> escape sequences.  Any white space after the
     * key is skipped; if the first non-white space character after
     * the key is <code>'='</code> or <code>':'</code>, then it is
     * ignored and any white space characters after it are also
     * skipped.  All remaining characters on the line become part of
     * the associated element string; if there are no remaining
     * characters, the element is the empty string
     * <code>&quot;&quot;</code>.  Once the raw character sequences
     * constituting the key and element are identified, escape
     * processing is performed as described above.
     *
     * <p>
     * As an example, each of the following three lines specifies the key
     * <code>"Truth"</code> and the associated element value
     * <code>"Beauty"</code>:
     * <p>
     * <pre>
     * Truth = Beauty
     *  Truth:Beauty
     * Truth                    :Beauty
     * </pre>
     * As another example, the following three lines specify a single
     * property:
     * <p>
     * <pre>
     * fruits                           apple, banana, pear, \
     *                                  cantaloupe, watermelon, \
     *                                  kiwi, mango
     * </pre>
     * The key is <code>"fruits"</code> and the associated element is:
     * <p>
     * <pre>"apple, banana, pear, cantaloupe, watermelon, kiwi, mango"</pre>
     * Note that a space appears before each <code>\</code> so that a space
     * will appear after each comma in the final result; the <code>\</code>,
     * line terminator, and leading white space on the continuation line are
     * merely discarded and are <i>not</i> replaced by one or more other
     * characters.
     * <p>
     * As a third example, the line:
     * <p>
     * <pre>cheeses
     * </pre>
     * specifies that the key is <code>"cheeses"</code> and the associated
     * element is the empty string <code>""</code>.<p>
     * <p>
     *
     * <a name="unicodeescapes"></a>
     * Characters in keys and elements can be represented in escape
     * sequences similar to those used for character and string literals
     * (see sections 3.3 and 3.10.6 of
     * <cite>The Java&trade; Language Specification</cite>).
     *
     * The differences from the character escape sequences and Unicode
     * escapes used for characters and strings are:
     *
     * <ul>
     * <li> Octal escapes are not recognized.
     *
     * <li> The character sequence <code>\b</code> does <i>not</i>
     * represent a backspace character.
     *
     * <li> The method does not treat a backslash character,
     * <code>\</code>, before a non-valid escape character as an
     * error; the backslash is silently dropped.  For example, in a
     * Java string the sequence <code>"\z"</code> would cause a
     * compile time error.  In contrast, this method silently drops
     * the backslash.  Therefore, this method treats the two character
     * sequence <code>"\b"</code> as equivalent to the single
     * character <code>'b'</code>.
     *
     * <li> Escapes are not necessary for single and double quotes;
     * however, by the rule above, single and double quote characters
     * preceded by a backslash still yield single and double quote
     * characters, respectively.
     *
     * <li> Only a single 'u' character is allowed in a Uniocde escape
     * sequence.
     *
     * </ul>
     * <p>
     * The specified stream remains open after this method returns.
     *
     * @param   reader   the input character stream.
     * @throws  IOException  if an error occurred when reading from the
     *          input stream.
     * @throws  IllegalArgumentException if a malformed Unicode escape
     *          appears in the input.
     * @since   1.6
     */
    public synchronized void load(Reader reader) throws IOException {
        load0(new LineReader(reader));
    }
```

loadFromXML(InputStream):从xml文件中加载property，底层使用XMLUtils.load(Properties,InputStream)方法来加载。

```
    /**
     * Loads all of the properties represented by the XML document on the
     * specified input stream into this properties table.
     *
     * <p>The XML document must have the following DOCTYPE declaration:
     * <pre>
     * &lt;!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"&gt;
     * </pre>
     * Furthermore, the document must satisfy the properties DTD described
     * above.
     *
     * <p>The specified stream is closed after this method returns.
     *
     * @param in the input stream from which to read the XML document.
     * @throws IOException if reading from the specified input stream
     *         results in an <tt>IOException</tt>.
     * @throws InvalidPropertiesFormatException Data on input stream does not
     *         constitute a valid XML document with the mandated document type.
     * @throws NullPointerException if <code>in</code> is null.
     * @see    #storeToXML(OutputStream, String, String)
     * @since 1.5
     */
    public synchronized void loadFromXML(InputStream in)
        throws IOException, InvalidPropertiesFormatException
    {
        if (in == null)
            throw new NullPointerException();
        XMLUtils.load(this, in);
        in.close();
    }
```

store(OutputStream/Writer，comments)将所有的property(保存defaults的)都写出到流中，同时如果给定comments的话，那么要加一个注释。

```
    /**
     * Writes this property list (key and element pairs) in this
     * <code>Properties</code> table to the output character stream in a
     * format suitable for using the {@link #load(java.io.Reader) load(Reader)}
     * method.
     * <p>
     * Properties from the defaults table of this <code>Properties</code>
     * table (if any) are <i>not</i> written out by this method.
     * <p>
     * If the comments argument is not null, then an ASCII <code>#</code>
     * character, the comments string, and a line separator are first written
     * to the output stream. Thus, the <code>comments</code> can serve as an
     * identifying comment. Any one of a line feed ('\n'), a carriage
     * return ('\r'), or a carriage return followed immediately by a line feed
     * in comments is replaced by a line separator generated by the <code>Writer</code>
     * and if the next character in comments is not character <code>#</code> or
     * character <code>!</code> then an ASCII <code>#</code> is written out
     * after that line separator.
     * <p>
     * Next, a comment line is always written, consisting of an ASCII
     * <code>#</code> character, the current date and time (as if produced
     * by the <code>toString</code> method of <code>Date</code> for the
     * current time), and a line separator as generated by the <code>Writer</code>.
     * <p>
     * Then every entry in this <code>Properties</code> table is
     * written out, one per line. For each entry the key string is
     * written, then an ASCII <code>=</code>, then the associated
     * element string. For the key, all space characters are
     * written with a preceding <code>\</code> character.  For the
     * element, leading space characters, but not embedded or trailing
     * space characters, are written with a preceding <code>\</code>
     * character. The key and element characters <code>#</code>,
     * <code>!</code>, <code>=</code>, and <code>:</code> are written
     * with a preceding backslash to ensure that they are properly loaded.
     * <p>
     * After the entries have been written, the output stream is flushed.
     * The output stream remains open after this method returns.
     * <p>
     *
     * @param   writer      an output character stream writer.
     * @param   comments   a description of the property list.
     * @exception  IOException if writing this property list to the specified
     *             output stream throws an <tt>IOException</tt>.
     * @exception  ClassCastException  if this <code>Properties</code> object
     *             contains any keys or values that are not <code>Strings</code>.
     * @exception  NullPointerException  if <code>writer</code> is null.
     * @since 1.6
     */
    public void store(Writer writer, String comments)
        throws IOException
    {
        store0((writer instanceof BufferedWriter)?(BufferedWriter)writer
                                                 : new BufferedWriter(writer),
               comments,
               false);
    }

    /**
     * Writes this property list (key and element pairs) in this
     * <code>Properties</code> table to the output stream in a format suitable
     * for loading into a <code>Properties</code> table using the
     * {@link #load(InputStream) load(InputStream)} method.
     * <p>
     * Properties from the defaults table of this <code>Properties</code>
     * table (if any) are <i>not</i> written out by this method.
     * <p>
     * This method outputs the comments, properties keys and values in
     * the same format as specified in
     * {@link #store(java.io.Writer, java.lang.String) store(Writer)},
     * with the following differences:
     * <ul>
     * <li>The stream is written using the ISO 8859-1 character encoding.
     *
     * <li>Characters not in Latin-1 in the comments are written as
     * <code>&#92;u</code><i>xxxx</i> for their appropriate unicode
     * hexadecimal value <i>xxxx</i>.
     *
     * <li>Characters less than <code>&#92;u0020</code> and characters greater
     * than <code>&#92;u007E</code> in property keys or values are written
     * as <code>&#92;u</code><i>xxxx</i> for the appropriate hexadecimal
     * value <i>xxxx</i>.
     * </ul>
     * <p>
     * After the entries have been written, the output stream is flushed.
     * The output stream remains open after this method returns.
     * <p>
     * @param   out      an output stream.
     * @param   comments   a description of the property list.
     * @exception  IOException if writing this property list to the specified
     *             output stream throws an <tt>IOException</tt>.
     * @exception  ClassCastException  if this <code>Properties</code> object
     *             contains any keys or values that are not <code>Strings</code>.
     * @exception  NullPointerException  if <code>out</code> is null.
     * @since 1.2
     */
    public void store(OutputStream out, String comments)
        throws IOException
    {
        store0(new BufferedWriter(new OutputStreamWriter(out, "8859_1")),
               comments,
               true);
    }
```

storeToXML(OutputSteam, comment, encoding):写出到xml文件中。

```
    /**
     * Emits an XML document representing all of the properties contained
     * in this table, using the specified encoding.
     *
     * <p>The XML document will have the following DOCTYPE declaration:
     * <pre>
     * &lt;!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"&gt;
     * </pre>
     *
     *<p>If the specified comment is <code>null</code> then no comment
     * will be stored in the document.
     *
     * <p>The specified stream remains open after this method returns.
     *
     * @param os        the output stream on which to emit the XML document.
     * @param comment   a description of the property list, or <code>null</code>
     *                  if no comment is desired.
     * @param  encoding the name of a supported
     *                  <a href="../lang/package-summary.html#charenc">
     *                  character encoding</a>
     *
     * @throws IOException if writing to the specified output stream
     *         results in an <tt>IOException</tt>.
     * @throws NullPointerException if <code>os</code> is <code>null</code>,
     *         or if <code>encoding</code> is <code>null</code>.
     * @throws ClassCastException  if this <code>Properties</code> object
     *         contains any keys or values that are not
     *         <code>Strings</code>.
     * @see    #loadFromXML(InputStream)
     * @since 1.5
     */
    public void storeToXML(OutputStream os, String comment, String encoding)
        throws IOException
    {
        if (os == null)
            throw new NullPointerException();
        XMLUtils.save(this, os, comment, encoding);
    }
```

四、源码分析

主要针对加载属性方法(load/loadFromXML)和写出属性到磁盘文件方法来进行分析(store/storeToXML)。

1、load(Reader)和load(InputStream)

这两个方法是指定从文本文件中加载key/value属性值，底层都是将流封装成为LineReader对象，然后通过load0方法来加载属性键值对的，加载完属性后流对象是不会关闭的。这两个方法对应的properties文件格式如下：

```
# this is comment
key1:value1
key2=value2
key3   :   vlaue3
key4   :   value4
# the value is 'value4  ', because the Properties only trim the space of the split charset before and after.

#   key5  = value5
# this is error, the key not start with the space.

key6   value7
```

LineReader源码分析：

```
    class LineReader {
        /**
         * 根据字节流创建LineReader对象
         * 
         * @param inStream
         *            属性键值对对应的字节流对象
         */
        public LineReader(InputStream inStream) {
            this.inStream = inStream;
            inByteBuf = new byte[8192];
        }

        /**
         * 根据字符流创建LineReader对象
         * 
         * @param reader
         *            属性键值对对应的字符流对象
         */
        public LineReader(Reader reader) {
            this.reader = reader;
            inCharBuf = new char[8192];
        }

        // 字节流缓冲区，大小为8192个字节
        byte[] inByteBuf;
        // 字符流缓冲区，大小为8192个字符
        char[] inCharBuf;
        // 当前行信息的缓冲区，大小为1024个字符
        char[] lineBuf = new char[1024];
        // 读取一行数据时候的实际读取大小
        int inLimit = 0;
        // 读取的时候指向当前字符位置
        int inOff = 0;
        // 字节流对象
        InputStream inStream;
        // 字符流对象
        Reader reader;

        /**
         * 读取一行，将行信息保存到{@link lineBuf}对象中，并返回实际的字符个数
         * 
         * @return 实际读取的字符个数
         * @throws IOException
         */
        int readLine() throws IOException {
            // 总的字符长度
            int len = 0;
            // 当前字符
            char c = 0;
```

```java
boolean skipWhiteSpace = true;
boolean isCommentLine = false;
boolean isNewLine = true;
boolean appendedLineBegin = false;
boolean precedingBackslash = false;
boolean skipLF = false;

while (true) {
    if (inOff >= inLimit) {
        // 读取一行数据，并返回这一行的实际读取大小
        inLimit = (inStream == null) ? reader.read(inCharBuf) : inStream.read(inByteBuf);
        inOff = 0;
        // 如果没有读取到数据，那么就直接结束读取操作
        if (inLimit <= 0) {
            // 如果当前长度为0或者是改行是注释，那么就返回-1。否则返回len的值。
            if (len == 0 || isCommentLine) {
                return -1;
            }
            return len;
        }
    }

    // 判断是根据字符流还是字节流读取当前字符
    if (inStream != null) {
        // The line below is equivalent to calling a ISO8859-1 decoder.
        // 字节流是根据ISO8859-1进行编码的，所以在这里进行解码操作。
        c = (char) (0xff & inByteBuf[inOff++]);
    } else {
        c = inCharBuf[inOff++];
    }

    // 如果前一个字符是换行符号，那么判断当前字符是否也是换行符号
    if (skipLF) {
        skipLF = false;
        if (c == '\n') {
            continue;
        }
    }

    // 如果前一个字符是空格，那么判断当前字符是不是空格类字符
    if (skipWhiteSpace) {
        if (c == ' ' || c == '\t' || c == '\f') {
            continue;
        }
        if (!appendedLineBegin && (c == '\r' || c == '\n')) {
            continue;
        }
        skipWhiteSpace = false;
        appendedLineBegin = false;
    }

    // 如果当前新的一行，那么进入该if判断中
    if (isNewLine) {
        isNewLine = false;
        // 如果当前字符是#或者是!，那么表示该行是一个注释行
        if (c == '#' || c == '!') {
            isCommentLine = true;
            continue;
        }
    }

    // 根据当前字符是不是换行符号进行判断操作
    if (c != '\n' && c != '\r') {
        // 当前字符不是换行符号
        lineBuf[len++] = c;// 将当前字符写入到行信息缓冲区中，并将len自增加1.
        // 如果len的长度大于行信息缓冲区的大小，那么对lineBuf进行扩容，扩容大小为原来的两倍，最大为Integer.MAX_VALUE
        if (len == lineBuf.length) {
            int newLength = lineBuf.length * 2;
            if (newLength < 0) {
                newLength = Integer.MAX_VALUE;
            }
            char[] buf = new char[newLength];
            System.arraycopy(lineBuf, 0, buf, 0, lineBuf.length);
            lineBuf = buf;
        }
        // 是否是转义字符
        // flip the preceding backslash flag
        if (c == '\\') {
            precedingBackslash = !precedingBackslash;
        } else {
            precedingBackslash = false;
        }
```

```
            } else {
                // reached EOL
                if (isCommentLine || len == 0) {
                    // 如果这一行是注释行，或者是当前长度为0，那么进行clean操作。
                    isCommentLine = false;
                    isNewLine = true;
                    skipWhiteSpace = true;
                    len = 0;
                    continue;
                }
                // 如果已经没有数据了，就重新读取
                if (inOff >= inLimit) {
                    inLimit = (inStream == null) ? reader.read(inCharBuf) : inStream.read(inByteBuf);
                    inOff = 0;
                    if (inLimit <= 0) {
                        return len;
                    }
                }
                // 查看是否是转义字符
                if (precedingBackslash) {
                    // 如果是，那么表示是另起一行，进行属性的定义，len要自减少1.
                    len -= 1;
                    // skip the leading whitespace characters in following line
                    skipWhiteSpace = true;
                    appendedLineBegin = true;
                    precedingBackslash = false;
                    if (c == '\r') {
                        skipLF = true;
                    }
                } else {
                    return len;
                }
            }

        }
    }
}
```

根据这个源码，我们可以看出一些特征：readLine这个方法每次读取一行数据；如果我们想在多行写数据，那么可以使用'\'来进行转义，在该转义符号后面换行，是被允许的。

load0方法源码：

```
private void load0(LineReader lr) throws IOException {
    char[] convtBuf = new char[1024];
    // 读取的字符总数
    int limit;
    // 当前key所在位置
    int keyLen;
    // value的起始位置
    int valueStart;
    // 当前字符
    char c;
    //
    boolean hasSep;
    // 是否是转义字符
    boolean precedingBackslash;

    while ((limit = lr.readLine()) >= 0) {
        c = 0;
        // key的长度
        keyLen = 0;
        // value的起始位置默认为limit
        valueStart = limit;
        //
        hasSep = false;
        precedingBackslash = false;

        // 如果key的长度小于总的字符长度，那么就进入循环
        while (keyLen < limit) {
            // 获取当前字符
            c = lr.lineBuf[keyLen];
            // 如果当前字符是=或者是:，而且前一个字符不是转义字符，那么就表示key的描述已经结束
            if ((c == '=' || c == ':') && !precedingBackslash) {
                // 指定value的起始位置为当前keyLen的下一个位置
                valueStart = keyLen + 1;
                // 并且指定，去除空格
                hasSep = true;
                break;
            } else if ((c == ' ' || c == '\t' || c == '\f') && !precedingBackslash) {
```

```
                        // 如果当前字符是空格类字符，而且前一个字符不是转义字符，那么表示key的描述已经结束
                        // 指定value的起始位置为当前位置的下一个位置
                        valueStart = keyLen + 1;
                        break;
                    }
                    // 如果当前字符为'\'，那么跟新是否是转义号。
                    if (c == '\\') {
                        precedingBackslash = !precedingBackslash;
                    } else {
                        precedingBackslash = false;
                    }
                    keyLen++;
                }

                // 如果value的起始位置小于总的字符长度，那么就进入该循环
                while (valueStart < limit) {
                    // 获取当前字符
                    c = lr.lineBuf[valueStart];
                    // 判断当前字符是否是空格类字符，达到去空格的效果
                    if (c != ' ' && c != '\t' && c != '\f') {
                        // 当前字符不是空格类字符，而且当前字符为=或者是:，并在此之前没有出现过=或者:字符。
                        // 那么value的起始位置继续往后移动。
                        if (!hasSep && (c == '=' || c == ':')) {
                            hasSep = true;
                        } else {
                            // 当前字符不是=或者:，或者在此之前出现过=或者:字符。那么结束循环。
                            break;
                        }
                    }
                    valueStart++;
                }
                // 读取key
                String key = loadConvert(lr.lineBuf, 0, keyLen, convtBuf);
                // 读取value
                String value = loadConvert(lr.lineBuf, valueStart, limit - valueStart, convtBuf);
                // 包括key/value
                put(key, value);
            }
        }
```

我们可以看到，在这个过程中，会将分割符号两边的空格去掉，并且分割符号可以是=,:,空格等。而且=和:的级别比空格分隔符高，即当这两个都存在的情况下，是按照=/:分割的。可以看到在最后会调用一个loadConvert方法，该方法主要是做key/value的读取，并将十六进制的字符进行转换。

2、loadFromXML方法

该方法主要是提供一个从XML文件中读取key/value键值对的方法。底层是调用的XMLUtil的方法，加载完对象属性后，流会被显示的关闭。xml格式如下所示：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>comments</comment>
<entry key="key7">value7</entry>
<entry key="key6">value7</entry>
<entry key="key4">value4  </entry>
<entry key="key3">vlaue3</entry>
<entry key="key2">value2</entry>
<entry key="key1">value1</entry>
</properties>
```

底层调用的是XMLUtil.load方法，在该方法中是使用DOM方式来访问xml文件的，在这里不做详细的介绍。

3、store(InputStream/Reader,String)方法

该方法主要是将属性值写出到文本文件中，并写出一个comment的注释。底层调用的是store0方法。针对store(InputStream,String)方法，我们可以看到在调用store0方法的时候，进行字节流封装成字符流，并且指定字符集为8859-1。源码如下：

```
    private void store0(BufferedWriter bw, String comments, boolean escUnicode) throws IOException {
        if (comments != null) {
            // 写出注释，如果是中文注释，那么转化成为8859-1的字符
            writeComments(bw, comments);
        }
        // 写出时间注释
        bw.write("#" + new Date().toString());
        // 新起一行
        bw.newLine();
        // 进行线程间同步的并发控制
        synchronized (this) {
            for (Enumeration e = keys(); e.hasMoreElements();) {
                String key = (String) e.nextElement();
                String val = (String) get(key);
                // 针对空格进行转义，并根据是否需要进行8859-1编码
                key = saveConvert(key, true, escUnicode);
                /*
                 * No need to escape embedded and trailing spaces for value,
                 * hence pass false to flag.
                 */
                // value不对空格进行转义
                val = saveConvert(val, false, escUnicode);
                // 写出key/value键值对
                bw.write(key + "=" + val);
                bw.newLine();
            }
        }
        bw.flush();
    }
```

4、storeToXML方法

将属性写出到xml文件中，底层调用的是XMLUtil.store方法。不做详细的介绍。

五、实例

直接代码：

```
package com.gerry.bd.properties.jdk;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Properties;
import java.util.Set;

/**
 * 操作jdk自身操作属性配置文件的Properties类。<br/>
 * jdk1.7文档地址：http://docs.oracle.com/javase/7/docs/api/<br/>
 * java.util.Properties继承自HashTable，最主要的子类是Provider
 *
 * @author jsliuming
 *
 */
public class PropertiesApp {
    public static void main(String[] args) {
        InputStream input = null;
        // 第一种，使用ClassLoad的方法获取InputStram对象。
        input = PropertiesApp.class.getClassLoader().getResourceAsStream("propertiesApp.properties");
        // 第二种，直接使用Class的方法来获取InputStream对象。必须加'/'表示在classpath路径下，如果不加的话，那么获取的是PropertiesApp这个类所在package下的文件。
        input = PropertiesApp.class.getResourceAsStream("/propertiesApp.properties");
        OutputStream os = null;

        try {
            os = new FileOutputStream("storePropertiesApp.xml");
        } catch (FileNotFoundException e1) {
        }

        // 第一步：创建Properties对象
        Properties prop = new Properties();
        try {
            // 第二步：加载属性，不会自动关闭input输入流。
            prop.load(input);
            // 第三步：获取属性
            String value1 = prop.getProperty("key1");
            String value5 = prop.getProperty("key5");
            String value7 = prop.getProperty("key7", "defaultvalue");
            System.out.println("[key1:" + value1 + "],[key5:" + value5 + "],[key7:" + value7 + "]");
            Set<String> keys = prop.stringPropertyNames();
            System.out.println("全部的key/value属性：");
            for (String key : keys) {
                System.out.println("[" + key + "][" + prop.getProperty(key) + "]");
            }
            // 第四步：设置属性
            prop.setProperty("key7", "value7");
            // 第五步：保存成文件
            prop.storeToXML(os, "comments");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(input != null) {
                try {
                    input.close();
                } catch (IOException e) {
                    // ignore
                }
            }
            if (os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    // ignore
                }
            }
        }
    }

}
```

结果console的输出为：

```
[key1:value1],[key5:null],[key7:defaultvalue]
全部的key/value属性：
[key6][value7]
[key4][value4  ]
[key3][vlaue3]
[key2][value2]
[key1][value1]
```

标签：java源码分析

好文要顶 关注我 收藏该文

liuming_1992
关注 - 0
粉丝 - 38
+加关注

0 0

« 上一篇：数据结构-二叉树
» 下一篇：[solr] - solr5.2.1环境搭建 - 使用tomcat做为容器

posted @ 2015-03-24 09:19 liuming_1992 阅读(2503) 评论(1) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部

登录后才能查看或发表评论，立即 登录 或者 逛逛 博客园首页