第 33 卷 第 1 期 计 算 机 工 程 2007 年 1 月
Vol.33 No.1 Computer Engineering January 2007
·软件技术与数据库· 文章编号：1000—3428(2007)01—0098—03 文献标识码：A 中图分类号：TP311

GML 空间数据流压缩算法研究

杜成龙[1,2]，关佶红[1]，王 治[1,3]
(1. 武汉大学计算机学院，武汉 430072；2. 湖北国土资源职业学院，荆州 434002；3. 九江学院，九江 332005)

摘 要：提出了一种新的 GML 数据流压缩算法。研究了 GML 结构与数据动态分离并动态压缩，采用多线程技术进行并行传输，再在接收端动态解压并动态合并的过程。实验结果表明，该方法对大幅提高 GML 数据传输效率具有实际意义。
关键词：GML；数据流；压缩；多线程

Algorithm Study of Spatial Dataflow Compression Based on GML

DU Chenglong[1,2], GUAN Jihong[1], WANG Zhi[1,3]
(1. School of Computer, Wuhan University, Wuhan 430072; 2. Hubei Professional College of Land Resource, Jingzhou 434002;
3. Jiujiang University, Jiujiang 332005)

【Abstract】This paper proposes a kind of new GML dataflow and compressed algorithm, studies GML structure and data separate dynamically and compress dynamically, multi-thread technology is adopted in order to parallel transmit, and then receive the data and solve the course that depresses dynamically and amalgamates dynamically. The experimental result indicates that this method has actual meanings in improving GML data transmission efficiency by a wide margin.
【Key words】GML; Dataflow; Compression; Multi-thread

随着 GIS 的发展，尤其是 WebGIS 的迅猛发展，基于个人的地理信息将越来越多，数据格式的不同使得不同的 WebGIS 应用系统之间的数据共享和交换十分困难，无法适应新的应用需求。GML(Geography Markup Language)作为一种基于 XML 的用于地理信息表示的标记语言，能够表示地理空间对象的空间数据和非空间数据，本课题提出以 GML 为中间件正是适应这种发展要求的。共享和交换数据必然存在数据传输问题，这导致网络数据传输成为迫切需要解决的非常关键的问题。

如何提高 XML 数据流处理效率，满足数据流应用系统的要求，成为了数据流研究领域的热点问题。由于数据流具有动态性，XML 数据流中的数据随着时间推移而增加，系统内存中无法保留已经处理的全部数据，最多只能进行部分数据缓存；数据流处理算法无法对数据进行整体 2 次扫描；并且数据流以网络速度进行传输，因此要求处理算法能够实时处理数据流。但静态算法主要目的是减少文档的存储空间代价，很显然静态方法在数据流中是不适用的，已有的 XML 流算法又没有考虑局部相似性的特点，所以应用在 GML 数据流中并不理想。

基于这一背景，本文提出基于 GML 的空间数据流压缩算法，这也是基于常用压缩算法进行的 2 次开发，实现在海量数据流的情况下高速传输数据的可能。

1 GML 数据流压缩实现

1.1 技术线路概述

首先，给定 GML 数据流，数据源利用 SAX 解析器将数据流解析成为 GML 元素事件输出事件流，然后通过输出事件流分别获取数据信息和结构信息，并分发到两个数据流管道中，然后分别对两条数据流进行压缩，并进行网络传输(见图 1)。

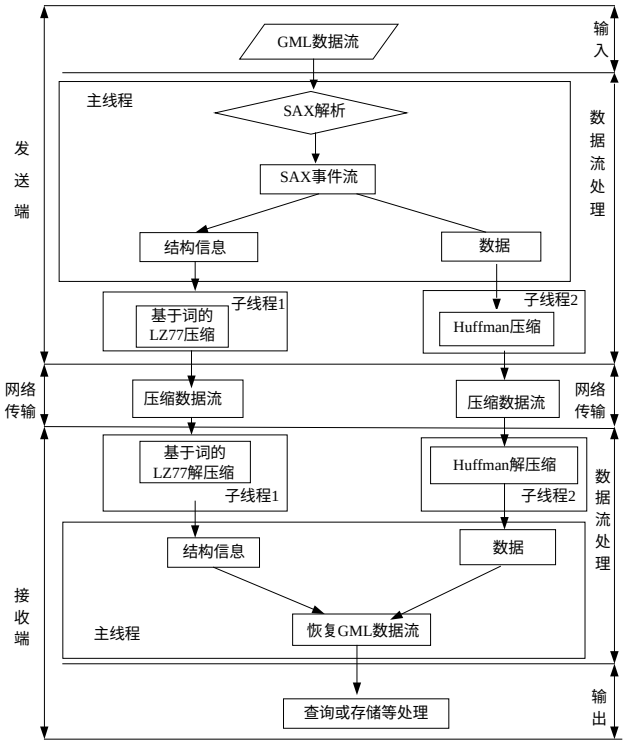

图 1 GML 数据流压缩实现整体框架

基金项目：国家自然科学基金资助项目(60373019)；国家“863”计划基金资助项目(2002AA135340)；软件工程国家重点实验室开放基金资助项目(SKL(4)003)；测绘遥感国家重点实验室开放基金资助项目(WKL(01)0303)和 IBM 软件奖研金项目资助
作者简介：杜成龙(1973－)，男，硕士生、讲师，主研方向：空间数据库；关佶红，博士、教授、博导；王 治，硕士生、助教
收稿日期：2006-01-28 **E-mail：**dchengl@tom.com

针对数据采用增量压缩算法比较合理，本处选择了 Huffman 压缩算法；针对结构信息则采用字典压缩法比较合理，本处则选择了基于词的 LZ77 压缩算法。由于本课题研究的内容有别于静态的压缩数据，数据是流动的并且总长度及概率等都是未知的，因此本处研究的压缩算法都是自适应模式的。当然，比以上压缩效果更高的压缩算法还有很多，但是本课程研究的最终目的不是达到最高压缩比，而是达到最大传输效率，所以综合考虑了压缩比和压缩时间以后才选择了以上算法的。以上压缩算法还有一个显著特点就是数据类型越少，编码会越短，复杂度也越低。因此从理论上讲，分离后分别采用合适的压缩算法进行压缩，产生的编码总长度要比单纯的对整体文件压缩的总长度要短。可想而知，本处分离压缩是可行的。

从处理时间和布局上考虑，使用 SAX 解析进行分离，此时产生了 2 条数据流，此处考虑可以采用两个线程对两条数据流分别实现压缩，并让所有线程同步，这样会提高压缩效率。另外此处进行数据传输使用了多线程并行传输技术，从整体上看，很明显会大幅提高传输效率。整体压缩传输流程图如图 1 所示。下面就 GML 数据流压缩算法进行研究。

1.2 GML 数据及结构分离与合并设计

空间数据的表达结构大多以图层为表达基本单位，即由一组具有相同空间类型的简单空间对象构成一个图层。GML 文档的内容中，含有大量重复的元素标签、以及高精度的浮点数和少量字符数据。对于少量的字符数据由于对大量浮点数压缩影响比较小，因此本处没有考虑将其分开。下面开始对几个重要技术环节进行设计。

1.2.1 基于 SAX 的 GML 文档解析

GML 的数据解析分为两种主要方法，一种采用 DOM 树的方法，即解析器在内存中构造 GML 树，树的节点就是 GML 的空间数据元素；另一种方法是 SAX 的方法，它的输出不是一个文档树，而是前序扫描 GML 文档树时所触发的事件。在数据流环境下，输入的数据量一般远远大于系统所能够利用的内存，所以，考虑到系统的资源限制，DOM 解析方法不再适用。只能采用 SAX 的解析方法，获取 GML 文档树中元素起始事件和终止事件流。本课题研究 GML 数据流的压缩，实际上是研究 SAX 解析后的输出事件流的压缩。

1.2.2 结构信息与数据的分离

作者已经知道以上事件序列分别对应的处理方法，要分离结构和数据就好办，只需在方法中选择不同的数据流输出管道，就可以解决问题了。现在以 FeatureMember 空间对象作为识别块，设计如下：

假定有两个 PipedOutputStream 类对象 out1，out2，其中 out1 输出结构信息，out2 输出数据，另外还有一个 StringBuffer 类的 elemValue 作为缓存变量。如果要对结构信息和数据进行操作，就必须重载 DefaultHandler 接口中的 startElement()、endElement()和 characters()方法。下面看看如何实现结构和数据的分离，设计伪代码如下：

```
int   flag=0;   //1 表示 out1 输出，2 表示 out2 输出
public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException
{   Synchronized(this){
            flag=1;     //表示 out1 输出
            elemValue.setLength(0);
            elemValue.append(qName);
        }
    }
public void endElement(String uri, String localName, String qName) throws SAXException
{     Synchronized(this){
            flag=1;   //表示 out1 输出
            if(qName==”FeatureMember”)
        { count=true; //此处  count 用于 Huffman 开始统计的标志
        }
            elemValue.setLength(0);
            elemValue.append(qName);
        }
}
public void characters(char[] ch, int start, int length) throws SAXException
{     Synchronized(this){
            flag=2;   //表示 out2 输出
            elemValue.setLength(0);
            elemValue.append(ch,start,length);
        }
  }
```

下面是压缩端子线程的伪代码：

```
public run(){
 Synchronized(this){
        String str=new String(elemValue.toString());
        flag1=flag;
      }
if(flag1==1)   //使用基于词 LZ77 压缩标签并输出到 out1
     LZ77WtoStream(str,out1);
if(flag1==2) //使用 Heffman 压缩浮点数并输出到 out2
    HeffmantoStream(str,out2);
}
```

特别注意的是，以上算法前 3 个方法运行在主线程中，而 run()方法运行在子线程 1、子线程 2 中，有两个副本在运行。显然以上代码能够完成结构信息和数据的分离。

1.2.3 结构信息与数据的合并

前面已经设计出了分离方法，对于合并方法，只要有个对应关系就可以解决问题，实际上，在接受端也是采用了多线程技术，合并线程是主线程，它主要控制经过解压后的结构信息和数据信息的正确合并，通过一定的策略，让两个数据源同步进行即可，下面来设计一下合并的方法：

同样的设有两个解压后的字符串 str1、str2，其中 str1 为结构信息，str2 为数据，out 为输出流，设计伪代码如下：

```
public void Union_GML(){
while(str1!=-1){
    while(flag1==true){
      if(str1.charAt(1)==’/’)     //检测第 2 个字符是否为’/’
       cstr=’<’+str1.substring(2); //获取结束元素的子串
    if(cstr!=pstr) //如果相等表示两元素间要插入数据。
       { str+=str1; pstr=str1;flag1=false;}
      else
{
do{
if(flag2==true){str+=str2;str+=str1; }
} while(flag2==false);   //do…while 在此起同步作用
```

```
flag2=false;     }
if(pstr=”</FeatureMember>”) //相当于分段输出的标志
{    out(str);    str=””;    }
}    }     }
```

下面是解压缩端子线程的伪代码：

```
public run(){
      Synchronized(this){
            String str=new String(elemValue.toString());
          }
if(flag==1)    //使用基于词 LZ77 解压缩标签
       ULZ77WtoStream(str1);
If(flag==2)    //使用 Heffman 解压缩浮点数
       UHeffmantoStream(str3);    }
```

对于解压缩端应该注意的是：通过设置 flag1、flag2 两个同步控制变量为 false，完成主线程与子线程间的数据准确传输；两个同步控制变量在解压缩代码中被修改设为 true，以便提示有新的值传递；另外通过循环控制完成某种数据类型的读取。由此看出以上算法是容易实现的。

2 实验对比测试

为了验证本算法的有效性，本人进行了测试，得到多组数据并进行了比较，由于篇幅的原因，本文不再列出试验数据。根据试验所得到的数据，结构部分总是数据部分的 1.5~3 倍，由此看出真正占文档容量的主要是结构信息，但是数据部分也是不可忽略的部分。

(1)比较单线程组和多线程组，如图 2。由图中可以看出，没有多大差距，但是采用多线程技术是在并行传输时才发挥作用，由于本试验数据是在本地机器上运行的，因此完全没有考虑网络传输的瓶颈问题。其实网络传输速度才是真正限制速度的瓶颈，很显然，采用多线程技术，并行传输，效率将会呈多倍的增长。

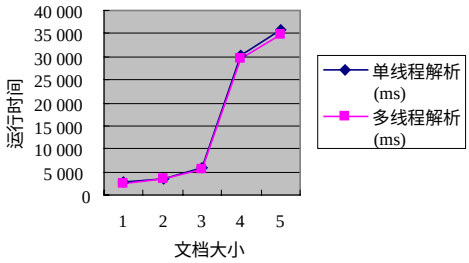

图 2 运行效率比较

(2)对结构信息，采用 Huffman 和 LZ77 压缩。如图 3。可以看出，LZ77 压缩的效果真让人惊奇！达到了 Huffman 编码无法比较的地步。再看图 4，压缩时间差不多，但是前面已经看到，LZ77 压缩的效果要出奇的好，由此看出对结构信息压缩使用 LZ77 算法是合理的。

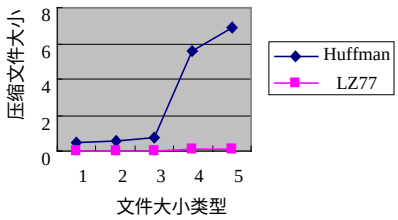

图 3 结构信息压缩效果比较

(3)对数据信息，见图 5。LZ77 编码和 Huffman 编码压缩的效果相差很小。但是对比数据压缩时间，从图 6 可以看出，Huffman 比 LZ77 压缩的时间要快 10 倍左右。要注意的是，本文研究的是数据流压缩算法，最重要的是传输效率，而不是压缩效果，所以在相同的压缩效果下，要优先考虑较少的压缩时间。因此采用 Huffman 算法是较合理的。

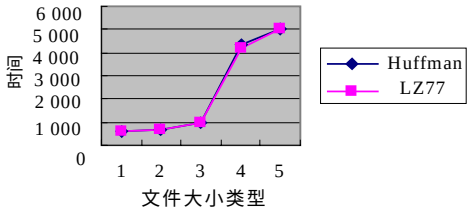

图 4 结构信息压缩时间比较

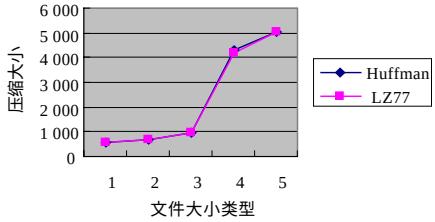

图 5 数据压缩效果比较

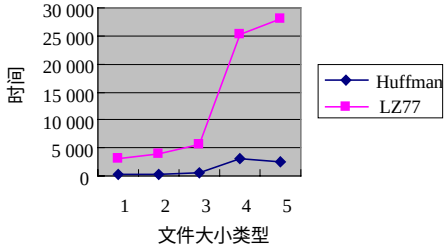

图 6 数据压缩时间比较

(4)同类算法比较。高军等研究的 XSC 系统采用单线程技术，并且只对数据部分压缩，而本算法采用多线程技术，并对数据和结构分别进行压缩。由此看来，本算法要比 XSC 系统传输效率要高。

3 结论

从试验的结果可以看出，本课题研究的数据流压缩技术是比较合理的，对提高 GML 数据传输效率具有实际意义。

本课题虽然只是“海量 GML 空间数据库管理系统关键技术研究”这一项目的一部分，但是它占有非常重要的位置。对该项目的成败起着关键性的作用。

参考文献

1 Vitter, Jeffrey S. Design and Analysis of Dynamic Huffman Codes[J]. Journal of the ACM, 1987, 34(4): 825-845.

2 Ziv J, Lempel A. A Universal Algorithm for Sequential Data Compression[J]. IEEE Transactions on Information Theory, 1997, IT-23(3): 337-343.

3 高　军, 杨冬青, 唐世渭, 等. 基于树自动机的 XPath 在 XML 数据流上的高效执行[J]. 软件学报, 2005, 16(2): 123-232.

4 Salomon D. 数据压缩原理与应用[M]. 吴乐南, 译. 北京: 电子工业出版社, 2003.