**QuantBox行情数据存储方案**

**伍侃 2015.03.17**

1. **项目需求**

行情数据过去使用csv文本格式进行存储，占用空间大，即使使用7z等工具进行压缩，仍然较大，并且csv格式解析时较为耗时。因此，可以考虑使用其他存储格式。

行情数据中存在大量重复数据，如涨跌停、交易日每日内取值相同。因此，若发现当前数据的某个字段与上一笔相同，可以跳过，以节省空间。

1. **解决方案**

采用二进制存储结合差分算法保存数据。

* 1. **二进制存储——Google Protocol Buffer**

首先，在csv文本格式的基础上，考虑使用简化的csv格式，文件占用空间变小。如：

1,2,3,4,5,6,7

4,,,,,,

5,2,,,,

进一步的，可用二进制表示替代文本表示。使用二进制存储需解决两个问题——如何做字段的分隔，如何表示与上一条数据的某个字段相同。

根据以上需求，采用Google Protocol Buffer进行二进制存储。在存储行情数据方面，其优点如下：

1. Key-Value模式：无需使用分隔符，无需记录使用默认值的数据。例如，若默认值为0/null，则不需要记录。
2. Varint：int32通常占用4个字节，若采用Varint，很小的数字可以1字节表示。例如，小于128的数就可以1字节表示，每个byte的最高1 bit表示后一个byte是否是当前数字的一部分。
3. ZigZag编码：负数占用字节较多，若正负数交错表示并结合Varint，即可在占用较少字节的情况下表示负数。例如，0->0,-1->1,1->2,-2->3。
   1. **差分算法**
      1. **概述**

第一笔保存全量，后续数据仅保存相对于上一笔的变化量，没有变化则记为0/null。如此protobuf保存可跳过为0的部分，使数据量变小。

* + 1. **设计原则**
       1. **依据Google Protocol Buffer的特性**

依据Google Protocol Buffer的特性，设计算法时应遵循以下原则：

1. Key不仅表示了Field编号，也表示了Field类型。因此，1字节无法表示128种编号。故而出现频率高的编号尽量放在1-15，从16开始使用2字节。
2. 字符串、对象：按Key-Length-Value进行存储，当对象中的所有值都为0时，至少占用两个字节Key-Length，例如0x12 0x00。因此，当对象中的所有元素都为0时，令对象 = null。
3. double：double类型永远是8字节，因此，尽量转换为int或long类型。
   * + 1. **依据行情数据的密集程度**

本文认为，行情数据依据其密集程度可大致分为两种，针对两种数据分别采取不同的设计原则：

1. 数据本身密集，需通过差分算法转换为稀疏数据。
2. 数据本身稀疏，可直接保存。

来自交易所的行情数据实时变化，是密集数据，因此需对其使用差分算法。而对于稀疏数据的判断，则存在争议：

1. 除权除息数据：除权除息数据是稀疏的，因为间隔较长时间才有一条。
2. 静态数据：静态数据如涨跌停，每天仅更新一次，但CTP行情中每Tick都有，因此本文将其归为密集数据，事实上，对其差分后效果很好。
3. Bar数据：在Tick中用来存当日的最高价与最低价、收盘、开盘。本文将其作为密集数据，对其差分后效果很好。
4. 配置数据：本文将配置数据归为稀疏的。事实上，若将不同品种的行情存在一个文件中，将导致配置数据来回切换，这是不合适的。
   * 1. **操作要点**
        1. **double类型转换为int类型**

行情中主要是double类型的数据，因此首先需要转换为int类型。

容易想到，可利用TickSize进行转换，即存储的数据事实上是Price / TickSize。例如，100 / 0.2 = 500。使用此方法可以将大部分数值转为int类型。

最初计划使用此方法存储结算价，但发现IF的交易日结算价与交割日结算价存在精度差异，交易日结算价是1位小数0.2，交割日结算价是2位小数0.01。因此，结算价的转换方法是先 \*100，再 / TickSize。(100是SettlementPriceMultiplier)

平均价的转换方法也是 \* 100 / TickSize。(100是AveragePriceMultiplier)

成交额的变化一定是最小变动金额的整数倍，因此其转化方法是 / (tickSize \* ContractMultiplier)。例如，IF是0.2 \* 300 = 60。

TradingDay和ActionDay也保存为 int类型。例如，20141125。

时间的保存也是关键，尤其是分秒和毫秒的保存。首先考虑将时间分为两部分，HHmmss/fff。然而，采取这种方式将产生以下问题：第一部分差分后不明显，因为每秒都更新，变化值不大，但却至少占用2字节；而fff每秒都变，且每次变化500，占用3字节。这种保存方式几乎每笔占用5字节，而事实上fff后面的ff大多数情况下为00。为进一步简化，最后采取的解决方法是用三部分表示，HHmm/ssf/ff。HHmm每分钟变动一次，ssf每次都变动，但通常变化5，占用2字节，每分钟仅出现一次占用3字节的情况。由于ff通常为00，因此几乎不占用空间。另外，由于ssf在IF合约中大量情况下取值为5，所以得到ssf后再减预定义的5，最终得到 0，又可减少2字节。

到底是减5、减10，还是不减呢，商品一般减10，而对于股票的数据，有可能返回的几条的时间都是一样的，所以不减。实际需要搞一个规则文本。

* + - 1. **多档行情存储**

一档行情需占用4个编号，五档占用20个编号，再加上时间、成交量、交易日等信息，一共需要超过50个编号。因此，考虑使用对象，每次只增加Key-Length，通常占用2字节。在对象内部，需注意不能大于15。

每一个价位的数据有价格、成交量、笔数。N档就有N\*3个数据，但多档行情却是线性的，一次记录14个，再用一个对象记录下一排。

所以我们需要有一个方法把二维的多档行情转成一维的数字排列。当然也得提供一个方法转回来。

我们在做差分时，只对二维的列表（列表是一维的，但每个位置存了三个值）进行差分，这样处理起来简单方便。首先，这个列表是已经按价格排好序的。然后对前后两个列表做差分。

1. 先将两个列表的长度变成一样长，索引相同，价格一样。当前列表不存在的价格，对应的量就标记为0。
2. 对同长度的列表同位置的成交量与笔数进行差分，生成新的列表
3. 对新列表由第1到最后一个价格进行差分，并减1，这样大部分价格就可以变成0，需要跳过成交量为0的项

这样就差分完了，最后，将差分后的临时列表转换为记录多档行情的对象进行存储。

多档行情为了区分Bid和Ask，决定再加一个临界点的价格，比如说买一价或卖一价，目前我们使用的是卖一价。

当只有Bid没有Ask时，卖一价为买一价+1跳。反过来当只有Ask没有Bid，卖一价就是卖一价本身。

当Bid和Ask都没有时，我们会先读List的长度，这里AskPrice1用0还是用上一笔？

* + - 1. **静态数据存储**

本文将每天变动一次的数据定义为静态数据，例如涨跌停、结算价等。这些静态数据单独存放在一个对象中。

* + - 1. **Bar数据存储**

对于Bar数据，CTP中的数据有开高低收，这与K线数据对应，所以可以用此数据结构同时记录Bar数据。

对Bar数据进行差分时，考虑到第一笔的Close与第二笔的Open有很大的概率相等，这里就存在第二笔的Open与上一笔的Close做差分还是与上笔的Open做差分。本文采用与上笔的Open做差分的方法，因为存储CTP中Tick数据的机会更多，而Bar数据即使被记录，其数量也少于Tick数据。

* + - 1. **TickSize存储**

差分算法的核心是TickSize，一旦TickSize不正确，其他的操作都无法准确进行。为了数据的自解析，TickSize需要保存，但不能保存为double类型。因此，本文采用 \* 10000的方法转换为int类型，也就是保存第一个TickSize，后面差分数据的TickSize不保存。

* + - 1. **测试及修改**

进行到这一步，一个数据文件的存储结构已经大致设计完成，编写以下测试代码：

1. 源数据的Tick转为PbTick
2. 多个PbTick进行差分
3. 顺序保存

读取测试结果，发现读出的数据是大量数据中的最后一笔。原因是是保存的数据没有办法区分每个Tick的分隔，这样一直读到最后，然后按Key-Value，同样的Key被多次更新，最后更新到最后一次出现的Key。

为解决此问题，使用Length，即改成Length-PbTick，一个长度，一块Tick即可。

若此存储方案有效，按日、按合约存储数据，每天会生成上百个几KB到几MB的小文件，这不利于文件分享。因此考虑合并，此种文件格式可以直接用copy 1.bin /b + 2.bin /b 3.bin的方式来合并。如果按照这种方式合并，数据中会出现：快照，差分，差分……快照，差分，差分……。

为了定位其中的快照数据，需考虑前述几个关键参数，如TickSize, ContractMultiplier等。这些参数开始是写在了编解码器中，最后定一个参数配置对象，将这些核心参数放在对象中。因此，约定当这个对象不为空时表示是快照，需要将当前的PbTick直接返回。这种方法对于国外的“价格范围不同TickSize也不同”进行了支持，只要在这时记录下新TickSize的一次快照即可。

如果不使用成交金额数据，ContractMultiplier可以不设置，使用默认的1即可，这样成交金额每次变动为TickSize的整数倍，但这样数字太大，所以只要是在使用成交金额的情况下，一定要设置正确合约乘数，例如IF的合约乘数是300。

当然，这种方案也存在局限性。首先，它不能随机访问，必需文件加载完成后才能解析。这对于行情交易影响较小。用户想使用时，至少返回的整天数据，而不是说你只想要其中一小时的我就只返回一小时。过滤是由客户端来完成的。其次，它不能直接编辑，需要编辑时必须生成新文件。

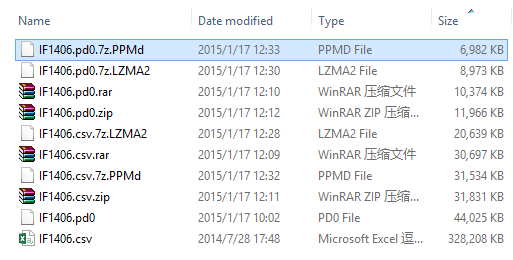
* 1. **专业压缩软件压缩**

生成文件后，可以对其再次使用专业压缩软件进行压缩，如7z，zip等。

由于这种行情格式中的数据有它自己的特点，再用不同的压缩算法达到的效果差距很大。下面测试哪种压缩算法更好（以下试验均采用最大压缩方式，注意，此表格中的数据使用的是V1版本的行情编码算法，它在处理一档行情时比较好，在处理多档行情时效率快速下降）。

将csv中的一档行情数据完全转成pd0格式，csv是IF1406从2013年10月21日到2014年6月20日，共5059003条数据，基本的一档行情。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | csv | Pd0 | csv | Pd0 | 压缩需内存 | 解压需内存 |
| 原始文件 | 320MB | 42.9MB |  |  |  |  |
| 7z/PPMd | 30.7MB | 6.81MB | 42s | 12s | 223MB | 194MB |
| 7z/LZMA2 | 20.1 MB | 8.76 MB | 103s | 31s | 2221MB | 66MB |
| Rar | 29.9MB | 10.1MB | 38s | 9s |  |  |
| Zip | 31.0MB | 11.6MB | 6s | 1s |  |  |



对于纯文本压缩，LZMA2编码明显压缩率高，但耗时太长。

对于这种特殊的二进制编码，PPMd明显好于其它几种的压缩效率，耗时也在可接受范围内。

因此，7z->极限压缩->PPMd->192MB为最优方案。

1. **参考文献**

[1] Google Protocol Buffer 的使用和原理

<http://www.ibm.com/developerworks/cn/linux/l-cn-gpb/>

[2] PPM压缩算法的分析与实现

<http://yeye55blog.blog.163.com/blog/static/197241021201110715759577/>