

ST_Curve 升级报告

至 2008 年 7 月 24 日，控件自诞生以来第二次大的升级基本完成（第一次是将所有数据的内存管理交给了 STL），这次升级的主要内容是增加了曲线的绘制速度，以及减少了 CPU 资源的占用量，当然内存占用会更多一些。ST_Curve 自编写第一行代码之前，我就知道，这个控件的优势将是随心所欲的拖动曲线，缩放曲线，以及绘制速度。在曲线的拖动过程中，曲线不得不一次又一次的重绘，以前的做法是，每次重绘的时候，都去计算屏幕坐标，这样当然效率就会差一些，于是我不得不从其它一些地方来弥补（这些弥补，不一定眼睛可以看到，但至少可以减少占用的 CPU 资源，比如 CPU 从原来的占用%2 到占用 1%，用眼睛是感觉不到的，但提高效率是每一位程序员的责任），比如我将每条曲线在画布中的第一个可见点的位置保存了下来，这样重绘的时候，就从这一点开始即可；我又将曲线分为 1 次曲线和 2 次曲线，如果是 1 次曲线，则绘制到第一个画布之外的点的时候，马上退出绘制（2 次则要绘制到曲线结束），于是更新每一条曲线在画布中的第一个可见点变得非常麻烦，而且更新非常的频繁（曲线经常被拖动，那么第一个可见点经常要改变）。为了解决这一系列问题，在这一次升级中，我将曲线的屏幕坐标也保存了下（这就带来了每一点 8 个字节的额外开销），这样曲线在重新绘制的时候，不需要计算屏幕坐标，而曲线的移动过程中，可以直接对保存下来的屏幕坐标做一个偏移，这些都是整数运算，相对于以前的 double 型加减乘除，速度自然提高许多，具体请往下看。

ST_Curve 计算屏幕坐标的内部原理详解

一：

首先找到一个基点（double, float），（参看 SetBenchmark 函数，控件自动将第一条曲线的第一个点的坐标当成基点，基点最好不要修改，但有些时候又必须修改，请往后看），然后对于当前坐标原点（double, float），根据每个刻度所代表的间隔（参看 SetTimeSpan 和 SetValueStep 函数），以及每个刻度所代表的屏幕像素个数（21 个，不能修改），就可以得到原点到基点的屏幕距离（long, long）；

二：

对于所有曲线的所有点，按上面的方法，得到每个点到基点的屏幕距离（long, long），将得到的距离（long, long）当成坐标来看待，于是可以映射到画布坐标系（结合原点到基点的屏幕距离，映射过程只有对整数的加减），得到每个点在画布中的坐标（long, long），并将所有这些坐标保存下来，在绘制曲线的时候，直接取出来使用，不需要做任何处理；

三：

当移动曲线的时候（参看 DragCurve 函数），对当前坐标原点，以及所有曲线的点的屏幕坐标，直接偏移一个距离即可（均为整数的加减）。

四：

当改变当前坐标原点的时候（参看 SetBeginTime、SetBeginTime2、SetBeginValue），那么对原点做第一步操作，并记录原点到基点的距离（long, long）在原点改变后，相对于改变之前的偏移量（均为整数），有了这个偏移量之后，对所有曲线的所有点，直接应用这个偏移（均为整数的加减），就得到了所有曲线的新的绘制位置（屏幕坐标）；

五：

当画布（画曲线的地方，不包括坐标区域）大小发生改变时（比如控件窗口大小改变，纵坐标显示精度改变，纵坐标显示范围改变等都有可能引起画布大小的改变），记录画布改变后相对于改变前的偏移量，对所有曲线的所有点，直接应用这个偏移（均为整数的加减），就得到了曲线的新的绘制位置（此时原点到基点的距离不用重新计算）；

六：

当坐标的一个刻度所代表的间隔发生改变时（参看 `SetTimeSpan` 和 `SetValueStep` 函数，`SetZoom` 也会产生相同的效果），必须重新做第一、二步；

七：

大家可以看到，对曲线的缩放（调用 `SetTimeSpan` 和 `SetValueStep` 函数也是缩放的一种特例），要远远慢于对曲线的移动，知道了这些特性，有助于大家更好的使用本控件；

八：

溢出处理，从上面的坐标计算可以看出，溢出是可能的，但要想让 `long` 溢出，是很难的，前提是基点要找好，一个理想的基点是容纳所有曲线的那个最小的矩形的中心点。实际使用时并没有必要去找这个理想的基点，但至少要在曲线附近。溢出只是理论上的，现实中可以不用关注这个问题。如果基点选择没有问题，而仍然发生了溢出，那么曲线的长度已经超过了本控件的允许范围，就无能为力了！

九：

关于 `SetBenchmark` 函数，提供它只是让控件尽量的完整，其实基本上不使用，控件会自动将第一条曲线的第一个点的坐标当成基点，这个函数的唯一用处就是，如果曲线长度超过控件的允许范围，但还没有超过 2 倍范围的时候，将基点设置成“理想基点”（参看上面第八条），可以消除溢出。但当长度超过曲线允许范围的 2 倍时，也就再也无能为力了。本函数会重复第一、二步，没事可千万别调用。

十：

曲线的允许长度（在屏幕上的长度，而不是指有多少个点）可以由下面的公式大概计算：
$$\text{maxwidth} = (\text{maxtime} - \text{Benchmarktime}) / \text{timespan} / (\text{zoom} > 0 ? (\text{zoom} * .25f + 1) : (-\text{zoom} * .25f + 1)) * 21;$$
$$\text{maxheight} = (\text{maxvalue} - \text{Benchmarkvalue}) / \text{valuestep} / (\text{zoom} > 0 ? (\text{zoom} * .25f + 1) : (-\text{zoom} * .25f + 1)) * 21;$$
只要上面的 `maxwidth` 和 `maxheight` 不超过 `long` 数据类型的限制，就可行。

十一：

上面八提到的 `long` 数据溢出可解决吗？表面上看，只要记下发生溢出的点（发生了溢出的点，一定不在画布之内），等显示到它的时候（通过翻页或者拖动曲线），再计算屏幕坐标不就行了？其实并不是这么简单，比如：如何知道溢出的点已经处于画布之中了呢？计算得到的屏幕坐标，因为已经溢出了，显然是不能用的，那么只能通过原始坐标来判断，此时又退化成了 `double` 数据的加减乘除了！另外还有一个技术难题，这涉及到控件的内部实现原理，控件保存了每一条曲线的最小坐标和最大坐标（参看 `GetOneFirstPos`、`GetOneTimeRange` 和 `GetTimeRange` 函数），以及他们的屏幕坐标，这两个屏幕坐标可以迅速

的判断曲线在画布中有没有显示，如果没有显示，绘制的时候直接跳过，以加快绘制速度。控件还保存了所有曲线最小坐标和最大坐标，以及他们的屏幕坐标，这两个屏幕坐标可以防止曲线被完全拖出画布。现在回到说溢出的问题，如果有溢出发生，则上面提到的那些最小坐标和最大坐标会最先溢出，他们一旦溢出，所有的判断将会出现问题，曲线的拖动也会因为判断错误而出问题。

此次升级，控件增加的功能概述

一：

填充方面有所增强，可向四个方向填充，而且同时可向多个方向填充。对于封闭的 2 次曲线（封闭的曲线一定是 2 次的），可对其内部填充。

二：

去掉了所有关于曲线次数的接口，但控件内部仍然是区分 1 次曲线和 2 次曲线的。0 次和 1 次曲线都归为 1 次曲线，2 次及其以上的曲线归为 2 次曲线。区分曲线次数是为了绘制速度，比如一条曲线有 100 年长，而显示在屏幕上的总是那么几个点，对于 1 次曲线，在绘制过程中，当出现第一个画布之外的点时，就退出绘制；而 2 次曲线必须要绘制到曲线结尾，因为曲线会折回，这样显然效率不如 1 次曲线。

现在去掉了关于幂次的接口，不但方便了二次开发者，同时也避免了出错的可能，因为开发者有可能误将 1 次曲线设置成了 2 次曲线，那样将影响绘制速度，或者误将 2 次曲线设置成了 1 次曲线，那样显示将不正常。

三：

去掉了 AddMinDataOver 和 AllAddDataOver 接口，表面上看只是减少了两个接口而已，其实远没有这么简单，这要涉及到控件内部的一些东西。上面的《ST_Curve 计算屏幕坐标的内部原理详解》中第十一条，大家还记得吧，控件内部保存了每一条曲线占的矩形区域，以及所有曲线占的矩形区域，AddMinDataOver 和 AllAddDataOver 接口正是用于计算这些区域的，如果让开发者来调用它们，那么开发者如果忘记了呢？这将带来严重的问题，比如看见曲线在那里，就是拖不动，翻不了页等。为此，此次升级，控件将在内部自动调用这两个函数（去掉了它们的接口属性，现在只是两个内部函数），那么控件内部是如何做到的呢？

控件内部用了一个链表来保存需要调用 AddMinDataOver 函数的曲线的地址（那么哪些曲线需要调用 AddMinDataOver 函数呢？答案是，按 VisibleState 为 0 调用过 AddMainData 和 AddMainData2 函数的曲线，调用过 InsertMainData 和 InsertMainData2 函数的曲线），在曲线需要被绘制时，对链表里面的所有曲线调用 AddMinDataOver 函数，完了后再调用一次 AllAddDataOver 函数，当然这样做以后，会清空链表，不用担心效率问题，只会执行一次。

我之所以透漏这么多控件内部的实现原理，是要大家明白一点，在曲线没有得到首次刷新之前，调用一些范围函数会出现问题，比如：GetOneTimeRange、GetOneValueRange、GetTimeRange、GetValueRange、FirstPage、GotoPage、ReportPageInfo 等等函数，大家可以看看 demo 里面的演示“方案五”的代码，在代码后面，特别添加了 Refresh 函数调用，就是为控件创造机会来调用 AddMinDataOver 和 AllAddDataOver 函数，否则最后面的 FirstPage 将达不到想要的结果，那么“方案一”至“方案四”为什么不需要 Refresh 呢？因为“方案五”里面的曲线是 2 次的，最小时间不在曲线头，调用 Refresh 就是要找到最小时间（当然还包括最大时间），以便翻到首页。而“方案一”至“方案四”由于都是 1 次曲线（至少第一条曲线是 1 次的，也是屏幕位置最靠前的，这就足够了），最小时间刚好是曲

线头，所以歪打正着，也就刚好达到要求，如果要翻页到最后一页，则“方案一”至“方案五”都将达不到要求（不调用 Refresh 函数，而曲线又长于一页的话），大家不防一试。

当然，去掉了 AddMinDataOver 和 AllAddDataOver 接口，也可以防止开发者忘记对他们的调用，对开发者也方便了，现在问题顶多是在必要的时候，要调用一下 Refresh 函数而已，其实要让控件刷新，有很多方法，比如将控件覆盖一下也可，因为大家都是开发者，这个问题我就不多说了。

四：

导出图元文件时，对于文本格式，里面的时间的表达方式增加了标准日期时间格式，以前只能按浮点数格式导出。

关于一些网友的建议

当然首先我要感谢那些给我提出意见建议，发现 BUG 的网友们。下面我将对于这些意见建议，不管我实现与否，都做一个解答：

一：

只让坐标显示一定范围之内的值，这个建议已经实现，开始我不大愿意做这个功能，因为我觉得控件的优势之一就是拖动，现在却要固定住坐标，有点不情愿，但后来想想，这在某些时候是还是有用的。

当然，在使用过程中，由于全部是浮点数比较，最好将范围设置得比想要的范围稍微大一点点（往两边都拓展一点点），只要不大到下一刻度的位置不小到上一刻度的位置，效果将是一样。

二：

让标题、脚注、正文、图例的字体与颜色独立，这一点我没有做，非常的抱歉。因为管理如此多的字体句柄和颜色的确是太麻烦，特别是多个字体的时候，字体的变化会影响到所有不以 (0, 0) 坐标为参照物的文字的绘制，比如横坐标，它是以画布底部为参照物的，目前我的做法是，只设置一个字体，对于标题，适当的增大字体，并用粗体，颜色不变。对于脚注，字体不变，颜色为前景色的四分之三，对于帮助信息，字体不变，颜色为前景色的二分之一。这样在灵活性与复杂度之间，我认为是找到了一个平衡。也许，正是由于控件在字体以及颜色方面的粗糙，说明它不是一个文字编辑器，而是一个绘制曲线的工具！

三：

控件在非 XP 样式下弹不出来坐标提示框，这个问题我至今还没有解决，为此我使用了另外一种方法，由我自己来实现一个类似 tooltip 的东西，目前看来效果还不错。

young wolf
2008 年 7 月 26 日

控件内部 GDI 部分技术文档（部分）

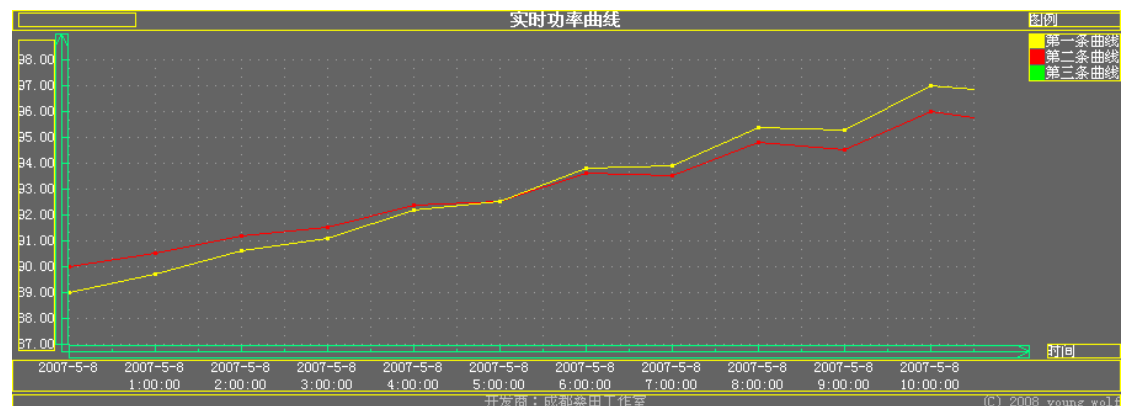
一：

采用双缓冲，两个内存兼容 DC，一个前景，一个背景。每当有刷新的时候，会将刷新应用到前景 DC（结合背景 DC），那么在窗口需要刷新的时候，只需要一条 BitBlt 函数，下面是控件的 OnDraw 函数（就一行代码）：

```
BitBlt(pdc->m_hDC, rcInvalid.left, rcInvalid.top,  
rcInvalid.Width(), rcInvalid.Height(),  
hFrceDC, rcInvalid.left, rcInvalid.top, SRCCOPY);
```

二：

区域刷新技术，下图为分的区域，可见已经很详细了，区域刷新是为了节省 CPU，对防止闪烁帮助不大。



关于区域刷新，请看我在 CSDN 中的技术文章：

<http://blog.csdn.net/yang79tao/archive/2008/07/31/2743184.aspx>

三：

尽量使用能够连续绘制的 GDI 函数，这也是为了节省 CPU 资源，具体为：

在绘制坐标值时，采用了 PolyTextOut 函数；在绘制坐标刻度时，采用了 PolyPolyline 函数；在绘制曲线的时候，采用了 Polyline 函数，在填充的时候，使用 Polygon 函数以及 ExtTextOut 函数（后者用于填充节点、填充图例前面的颜色示意块）。这些函数大多都能一次调用，执行一批绘制命令，所以节省了 CPU 资源，比如本控件中，一次调用 PolyTextOut 函数，就可以将横坐标或者纵坐标的刻度值全部显示出来，一次调用 PolyPolyline 函数，就可以将横坐标或者纵坐标的刻度全部显示出来。

当然，一个永远的事实是，要提高效率，要么提高运行速度，要么以空间换取时间，我们不可能去提高 CPU 的运算速度，所以只能以空间换取时间，所有这些一次调用执行多条 GDI 指令的函数（姑且这样说吧），是要付出更多的内存空间的。