**POCO C++库学习和分析 -- 日期与时间**

        在Poco库中，与时间和日期相关的一些类，其内部实现是非常简单的。看相关文档时，比较有意思的倒是历史上的不同时间表示法。

**1. 系统时间函数**

        在编程时，时间函数不可避免的会被使用。linux系统下相关时间的数据结构有time\_t，timeval，timespec，tm，clock\_t; windows下time\_t，tm，SYSTEMTIME，clock\_t。其中clock\_t、timeval、timespec用于表示时间跨度，time\_t、tm、SYSTEMTIME用于表示绝对时间。不同的数据结构之间，多少也有些差异。

       首先 这些时间结构体的精度不同，Second(time\_t/tm), microsecond(timeval/SYSTEMTIME),  nanoSeconds(timespec)。

        起始时间不同，time\_t起始于1970年1月1日0时0分0秒，tm表示起始于1900年，SYSTEMTIME起始于1601年，clock起始于机器开机。

        同这些数据结构相关联，C语言为tm，time\_t提供了一组函数用于时间运算和数据结构转换：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. // 日历时间（一个用time\_t表示的整数）
3. // 比较日历时间
4. **double** difftime(**time\_t** time1, **time\_t** time0);
5. // 获取日历时间
6. **time\_t** time(**time\_t** \* timer);
7. // 转换日历时间为字符串
8. **char** \* ctime(**const** **time\_t** \*timer);
9. // 转换日历时间为我们平时看到的把年月日时分秒分开显示的时间格式tm(GMT timezone)
10. **struct** **tm** \* gmtime(**const** **time\_t** \*timer);
11. // 转换日历时间为我们平时看到的把年月日时分秒分开显示的时间格式tm(本地 timezone)
12. **struct** **tm** \* localtime(**const** **time\_t** \* timer);
13. // 关于本地时间的计算公式：
14. localtime = utctime[Gmt time] + utcOffset()[时区偏移] + dst()[夏令时偏移]

17. // 把tm转换为字符串
18. **char** \* asctime(**const** **struct** **tm** \* timeptr);
19. // 把tm转换为日历时间
20. **time\_t** mktime(**struct** **tm** \* timeptr);

23. // 获取开机以来的微秒数
24. **clock\_t** clock (**void**);

        我们回想一下程序中的时间数据结构和函数的用法，可以发现主要是2个目的：  
        1. 获取绝对时间  
        2. 获取两个时间点的相对时间

**2. Timestamp类**

        同C语言中函数类似，Poco中定义了自己的时间类。Timestamp类似于time\_t，用于获取比较日历时间。Timestamp定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. **class** Foundation\_API Timestamp
2. {
3. **public**:
4. **typedef** Int64 TimeVal;    /// monotonic UTC time value in microsecond resolution
5. **typedef** Int64 UtcTimeVal; /// monotonic UTC time value in 100 nanosecond resolution
6. **typedef** Int64 TimeDiff;   /// difference between two timestamps in microseconds
8. Timestamp();
9. /// Creates a timestamp with the current time.
11. Timestamp(TimeVal tv);
12. /// Creates a timestamp from the given time value.
14. Timestamp(**const** Timestamp& other);
15. /// Copy constructor.
17. ~Timestamp();
18. /// Destroys the timestamp
20. Timestamp& operator = (**const** Timestamp& other);
21. Timestamp& operator = (TimeVal tv);
23. **void** swap(Timestamp& timestamp);
24. /// Swaps the Timestamp with another one.
26. **void** update();
27. /// Updates the Timestamp with the current time.

30. **bool** operator == (**const** Timestamp& ts) **const**;
31. **bool** operator != (**const** Timestamp& ts) **const**;
32. **bool** operator >  (**const** Timestamp& ts) **const**;
33. **bool** operator >= (**const** Timestamp& ts) **const**;
34. **bool** operator <  (**const** Timestamp& ts) **const**;
35. **bool** operator <= (**const** Timestamp& ts) **const**;
37. Timestamp  operator +  (TimeDiff d) **const**;
38. Timestamp  operator -  (TimeDiff d) **const**;
39. TimeDiff   operator -  (**const** Timestamp& ts) **const**;
40. Timestamp& operator += (TimeDiff d);
41. Timestamp& operator -= (TimeDiff d);
43. std::**time\_t** epochTime() **const**;
44. /// Returns the timestamp expressed in time\_t.
45. /// time\_t base time is midnight, January 1, 1970.
46. /// Resolution is one second.
48. UtcTimeVal utcTime() **const**;
49. /// Returns the timestamp expressed in UTC-based
50. /// time. UTC base time is midnight, October 15, 1582.
51. /// Resolution is 100 nanoseconds.
53. TimeVal epochMicroseconds() **const**;
54. /// Returns the timestamp expressed in microseconds
55. /// since the Unix epoch, midnight, January 1, 1970.
57. TimeDiff elapsed() **const**;
58. /// Returns the time elapsed since the time denoted by
59. /// the timestamp. Equivalent to Timestamp() - \*this.
61. **bool** isElapsed(TimeDiff interval) **const**;
62. /// Returns true iff the given interval has passed
63. /// since the time denoted by the timestamp.
65. **static** Timestamp fromEpochTime(std::**time\_t** t);
66. /// Creates a timestamp from a std::time\_t.
68. **static** Timestamp fromUtcTime(UtcTimeVal val);
69. /// Creates a timestamp from a UTC time value.
71. **static** TimeVal resolution();
72. /// Returns the resolution in units per second.
73. /// Since the timestamp has microsecond resolution,
74. /// the returned value is always 1000000.
76. **private**:
77. TimeVal \_ts;
78. };

        Timestamp内部定义了一个Int64的变量\_ts。存储了一个基于utc时间的64位int值，理论上可以提供微秒级的精度(实际精度依赖于操作系统)。由于Poco::Timestamp是基于UTC(世界标准时间或世界協調時間)的，所以它是独立于时区设置的。Poco::Timestamp实现了值语义，比较和简单的算术操作。  
        1. UTC (Coordinated Universal Time)是从1582年10月15日深夜开始计时的. Poco库中精度为100纳秒。  
        2. epoch time指是从1970年1月1日深夜开始计时的(指unix诞生元年)。Poco库中精度为1秒。  
  
**数据类型：**  
        Poco::Timestamp内部定义了下列数据类型：  
        1. TimeVal  
          一个64位的int整数值，保存utc时间，精度微秒  
        2. UtcTimeVal  
         一个64位的int整数值，保存utc时间，精度100纳秒(真实精度仍然是微秒)  
        3. TimeDiff  
        一个64位的int整数值，保存两个Timestamp的差值，精度微秒  
  
  
**构造函数：**  
        1. 默认构造函数会以当前时间初始化一个Timestamp值，基于UTC时间(从1582年10月15日开始计时，精度为100纳秒)  
        2. 提供了两个静态函数用于创建Timestamp对象，  
                   a) Timestamp fromEpochTime(time\_t time)。这个函数从time\_t构建，内部会把EpochTime(从1970年1月1日深夜开始计时的，精度为1秒)的时间转换成为UTC时间。  
                   b) Timestamp fromUtcTime(UtcTimeVal val)。这个函数从一个UtcTimeVal构建。

**Timestamp的成员函数：**  
        1. time\_t epochTime() const  
        返回一个以epoch time计算的日历时间(精度秒)。(函数内部会把基于UTC时间的值转为基于epoch time的值)  
        2. UtcTimeVal utcTime() const  
        返回一个以UTC时间计算的日历时间(精度100纳秒)。  
        3. TimeVal epochMicroseconds() const  
        返回一个以epoch time计算的日历时间(精度微秒)  
        4. void update()  
        取当前的时间更新  
        5. TimeDiff elapsed() const  
        返回当前时间与Timestamp内部时间\_ts的一个时间差值(精度微秒)  
        6. bool isElapsed(TimeDiff interval) const  
        如果当前时间与Timestamp内部时间\_ts的一个时间差值大于interval时间，返回true。(精度微秒)

**Timestamp算术计算：**  
        1. Timestamp operator + (TimeDiff diff) const  
        增加一个时间偏移，并返回值。(精度微秒)  
        2. Timestamp operator - (TimeDiff diff) const  
        减掉一个时间偏移，并返回值。(精度微秒)  
        3. TimeDiff operator - (const Timestamp& ts) const  
        返回两个Timestamp对象的时间偏移。(精度微秒)  
        4. Timestamp& operator += (TimeDiff d)  
            Timestamp& operator -= (TimeDiff d)  
        增加或减小一个时间偏移值  
  
        下面来看一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/Timestamp.h"
2. #include <ctime>
3. **using** Poco::Timestamp;
4. **int** main(**int** argc, **char**\*\* argv)
5. {
6. Timestamp now; // the current date and time
7. std::**time\_t** t1 = now.epochTime(); // convert to time\_t ...
8. Timestamp ts1(Timestamp::fromEpochTime(t1)); // ... and back again
9. **for** (**int** i = 0; i < 100000; ++i) ; // wait a bit
10. Timestamp::TimeDiff diff = now.elapsed(); // how long did it take?
11. Timestamp start(now); // save start time
12. now.update(); // update with current
13. time diff = now - start; // again, how long?
14. **return** 0;
15. }

**3. DateTime类**

        Poco中提供了DateTime类，作用和tm类似。下面是它的定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. **class** Foundation\_API DateTime
2. {
3. **public**:
4. **enum** Months
5. /// Symbolic names for month numbers (1 to 12).
6. {
7. JANUARY = 1,
8. FEBRUARY,
9. MARCH,
10. APRIL,
11. MAY,
12. JUNE,
13. JULY,
14. AUGUST,
15. SEPTEMBER,
16. OCTOBER,
17. NOVEMBER,
18. DECEMBER
19. };
21. **enum** DaysOfWeek
22. /// Symbolic names for week day numbers (0 to 6).
23. {
24. SUNDAY = 0,
25. MONDAY,
26. TUESDAY,
27. WEDNESDAY,
28. THURSDAY,
29. FRIDAY,
30. SATURDAY
31. };
33. DateTime();
34. /// Creates a DateTime for the current date and time.

37. DateTime(**const** Timestamp& timestamp);
38. /// Creates a DateTime for the date and time given in
39. /// a Timestamp.
41. DateTime(**int** year, **int** month, **int** day, **int** hour = 0, **int** minute = 0, **int**

44. second = 0, **int** millisecond = 0, **int** microsecond = 0);
45. /// Creates a DateTime for the given Gregorian date and time.
46. ///   \* year is from 0 to 9999.
47. ///   \* month is from 1 to 12.
48. ///   \* day is from 1 to 31.
49. ///   \* hour is from 0 to 23.
50. ///   \* minute is from 0 to 59.
51. ///   \* second is from 0 to 59.
52. ///   \* millisecond is from 0 to 999.
53. ///   \* microsecond is from 0 to 999.

56. DateTime(**double** julianDay);
57. /// Creates a DateTime for the given Julian day.

60. DateTime(Timestamp::UtcTimeVal utcTime, Timestamp::TimeDiff diff);
61. /// Creates a DateTime from an UtcTimeVal and a TimeDiff.
62. ///
63. /// Mainly used internally by DateTime and friends.

66. DateTime(**const** DateTime& dateTime);
67. /// Copy constructor. Creates the DateTime from another one.

70. ~DateTime();
71. /// Destroys the DateTime.

74. DateTime& operator = (**const** DateTime& dateTime);
75. /// Assigns another DateTime.
77. DateTime& operator = (**const** Timestamp& timestamp);
78. /// Assigns a Timestamp.

81. DateTime& operator = (**double** julianDay);
82. /// Assigns a Julian day.

85. DateTime& assign(**int** year, **int** month, **int** day, **int** hour = 0, **int** minute = 0,

88. **int** second = 0, **int** millisecond = 0, **int** microseconds = 0);
89. /// Assigns a Gregorian date and time.
90. ///   \* year is from 0 to 9999.
91. ///   \* month is from 1 to 12.
92. ///   \* day is from 1 to 31.
93. ///   \* hour is from 0 to 23.
94. ///   \* minute is from 0 to 59.
95. ///   \* second is from 0 to 59.
96. ///   \* millisecond is from 0 to 999.
97. ///   \* microsecond is from 0 to 999.

100. **void** swap(DateTime& dateTime);
101. /// Swaps the DateTime with another one.

104. **int** year() **const**;
105. /// Returns the year.
107. **int** month() **const**;
108. /// Returns the month (1 to 12).
110. **int** week(**int** firstDayOfWeek = MONDAY) **const**;
111. /// Returns the week number within the year.
112. /// FirstDayOfWeek should be either SUNDAY (0) or MONDAY (1).
113. /// The returned week number will be from 0 to 53. Week number 1 is

116. the week
117. /// containing January 4. This is in accordance to ISO 8601.
118. ///
119. /// The following example assumes that firstDayOfWeek is MONDAY. For 2005, which started
120. /// on a Saturday, week 1 will be the week starting on Monday, January 3.
121. /// January 1 and 2 will fall within week 0 (or the last week of the previous year).
122. ///
123. /// For 2007, which starts on a Monday, week 1 will be the week

126. startung on Monday, January 1.
127. /// There will be no week 0 in 2007.
129. **int** day() **const**;
130. /// Returns the day witin the month (1 to 31).
132. **int** dayOfWeek() **const**;
133. /// Returns the weekday (0 to 6, where
134. /// 0 = Sunday, 1 = Monday, ..., 6 = Saturday).
136. **int** dayOfYear() **const**;
137. /// Returns the number of the day in the year.
138. /// January 1 is 1, February 1 is 32, etc.
140. **int** hour() **const**;
141. /// Returns the hour (0 to 23).
143. **int** hourAMPM() **const**;
144. /// Returns the hour (0 to 12).
146. **bool** isAM() **const**;
147. /// Returns true if hour < 12;

150. **bool** isPM() **const**;
151. /// Returns true if hour >= 12.
153. **int** minute() **const**;
154. /// Returns the minute (0 to 59).
156. **int** second() **const**;
157. /// Returns the second (0 to 59).
159. **int** millisecond() **const**;
160. /// Returns the millisecond (0 to 999)
162. **int** microsecond() **const**;
163. /// Returns the microsecond (0 to 999)
165. **double** julianDay() **const**;
166. /// Returns the julian day for the date and time.
168. Timestamp timestamp() **const**;
169. /// Returns the date and time expressed as a Timestamp.

172. Timestamp::UtcTimeVal utcTime() **const**;
173. /// Returns the date and time expressed in UTC-based
174. /// time. UTC base time is midnight, October 15, 1582.
175. /// Resolution is 100 nanoseconds.
177. **bool** operator == (**const** DateTime& dateTime) **const**;
178. **bool** operator != (**const** DateTime& dateTime) **const**;
179. **bool** operator <  (**const** DateTime& dateTime) **const**;
180. **bool** operator <= (**const** DateTime& dateTime) **const**;
181. **bool** operator >  (**const** DateTime& dateTime) **const**;
182. **bool** operator >= (**const** DateTime& dateTime) **const**;

185. DateTime  operator +  (**const** Timespan& span) **const**;
186. DateTime  operator -  (**const** Timespan& span) **const**;
187. Timespan  operator -  (**const** DateTime& dateTime) **const**;
188. DateTime& operator += (**const** Timespan& span);
189. DateTime& operator -= (**const** Timespan& span);
191. **void** makeUTC(**int** tzd);
192. /// Converts a local time into UTC, by applying the given time zone

195. differential.
197. **void** makeLocal(**int** tzd);
198. /// Converts a UTC time into a local time, by applying the given time

201. zone differential.
203. **static** **bool** isLeapYear(**int** year);
204. /// Returns true if the given year is a leap year;
205. /// false otherwise.
207. **static** **int** daysOfMonth(**int** year, **int** month);
208. /// Returns the number of days in the given month
209. /// and year. Month is from 1 to 12.
211. **static** **bool** isValid(**int** year, **int** month, **int** day, **int** hour = 0, **int** minute =

214. 0, **int** second = 0, **int** millisecond = 0, **int** microsecond = 0);
215. /// Checks if the given date and time is valid
216. /// (all arguments are within a proper range).
217. ///
218. /// Returns true if all arguments are valid, false otherwise.
220. **protected**:
221. // ...
223. **private**:
224. // ...
226. Timestamp::UtcTimeVal \_utcTime;
227. **short**  \_year;
228. **short**  \_month;
229. **short**  \_day;
230. **short**  \_hour;
231. **short**  \_minute;
232. **short**  \_second;
233. **short**  \_millisecond;
234. **short**  \_microsecond;
235. };

        Poco::DateTime是基于格里高利历([Gregorian calenda](http://baike.baidu.com/view/22993.htm?subLemmaId=6418416&fromId=533255)r)(就是公历啦)设计的。它除了可以用来保存日历时间外，还可以被用于日期计算。如果只是为了日历时间的存储，Timestamp类更加适合。在DateTime的内部，DateTime类用两种格式维护了日期和时间。第一种是UTC日历时间。第二种是用年、月、日、时、分、秒、微秒、毫秒。为了进行内部时间日期之间的换算，DateTime使用了儒略日([Julian day](http://zh.wikipedia.org/wiki/%E5%84%92%E7%95%A5%E6%97%A5))历法。  
  
**格里高利历(Gregorian calendar)**  
        格里高利历就是我们通常讲的公历。它以耶稣的诞生为初始年份，也就是公元0001年。格里高利历以日为基本单位，1年有365或366天，分成12个月，每个月时长不等。由于不同国家采用格里高利历时间不同(德国1582, 英国1752)，所以格里高利历日期和旧式的日期有差别，即使是用来表示历史上相同的一件事。一个耶稣像，^\_^。  
      \_      xxxx      \_  
     /\_;-.\_\_ / \_\  \_.-;\_\  
        `-.\_`'`\_/'`.-'  
            `\   /`  
             |  /  
            /-.(  
            \\_.\_\  
             \ \`;  
              > |/  
             / //  
             |//  
             \(\  
          
  
**儒略日和儒略日日期**儒略日的起点订在公元前4713年（天文学上记为 -4712年）1月1日格林威治时间平午（世界时12:00），即JD 0指定为UT时间B.C.4713年1月1日12:00到UC时间B.C.4713年1月2日12:00的24小时。注意这一天是礼拜一。每一天赋予了一个唯一的数字，顺数而下，如：1996年1月1日12:00:00的儒略日是2450084。这个日期是考虑了太阳、月亮的轨道运行周期，以及当时收税的间隔而订出来的。Joseph Scliger定义儒略周期为7980年，是因28、19、15的最小公倍数为28×19×15=7980。  
  
**日期的注意事项：**1. 0是一个合法数字(根据ISO 8691和天文年编号)  
2. 0年是一个闰年。  
3. 负数是不支持的。比如说公元前1年。  
4. 格里高利历同历史上的日期可能不同，由于它们采用的日历方式不同。  
5. 最好只是用DateTime用来计算当前的时间。对于历史上的或者天文日历的时间计算，还是使用其他的特定软件。  
  
**构造DateTime：**1. 可以从一个已有的DateTime构造  
2. 当前的时间和日期  
3. 一个Timestamp对象  
4. 用年、月、日、时、分、秒、微秒、毫秒构造  
5. 使用一个儒略日日期构造(用double形式保存)  
  
  
**成员函数：**1. int year() const  
   返回年  
2. int month() const  
   返回月(1-12)  
3. int week(int firstDayOfWeek = DateTime::MONDAY) const  
   返回所在的周，根据ISO 8601标准(第一周是1月4日所在的周)，一周的第一天是礼拜一或者礼拜天。  
4. int day() const  
   返回月中的所在天(1 - 31)  
5. int dayOfWeek() const  
   返回周中的所在天 0为周日，1为周一，.....  
6. int dayOfYear() const  
   返回年中的所在天(1 - 366)  
7. int hour() const  
   返回天中所在小时(0 - 23)  
8. int hourAMPM() const  
   返回上下午所在小时(0 - 12)  
9. bool isAM() const  
   如果上午返回真  
10. bool isPM() const  
   如果下午返回真  
11. int minute() const  
   返回分钟数(0 - 59)  
12. int second() const  
   返回秒数(0 - 59)  
13. int millisecond() const  
   返回毫秒数(0 - 999)  
14. int microsecond() const  
   返回微秒数(0 - 999)  
15. Timestamp timestamp() const  
   返回用Timestamp保存的日历时间(精度微秒)  
16. Timestamp::UtcTimeVal utcTime() const  
   返回用Timestamp保存的日历时间(精度100纳秒)  
17. DateTime支持关系运算符(==, !=, >, >=, <, <=).  
18. DateTime支持算术操作(+, -, +=, -=)  
  
**静态函数：**1. bool isLeapYear(int year)  
   所给年是否闰年  
2. int daysOfMonth(int year, int month)  
   所给年和月的天数  
3. bool isValid(int year, int month, int day, int hour, int minute, int second, int millisecond, int microsecond)  
   判断所给年月日是否合法  
  
下面是DateTime的一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/DateTime.h"
2. **using** Poco::DateTime;
3. **int** main(**int** argc, **char**\*\* argv)
4. {
5. DateTime now; // the current date and time in UTC
6. **int** year = now.year();
7. **int** month = now.month();
8. **int** day = now.day();
9. **int** dow = now.dayOfWeek();
10. **int** doy = now.dayOfYear();
11. **int** hour = now.hour();
12. **int** hour12 = now.hourAMPM();
13. **int** min = now.minute();
14. **int** sec = now.second();
15. **int** ms = now.millisecond();
16. **int** us = now.microsecond();
17. **double** jd = now.julianDay();
18. Poco::Timestamp ts = now.timestamp();
19. DateTime xmas(2006, 12, 25); // 2006-12-25 00:00:00
20. Poco::Timespan timeToXmas = xmas - now;

23. DateTime dt(1973, 9, 12, 2, 30, 45); // 1973-09-12 02:30:45
24. dt.assign(2006, 10, 13, 13, 45, 12, 345); // 2006-10-13 12:45:12.345
25. **bool** isAM = dt.isAM(); // false
26. **bool** isPM = dt.isPM(); // true
27. **bool** isLeap = DateTime::isLeapYear(2006); // false
28. **int** days = DateTime::daysOfMonth(2006, 2); // 28
29. **bool** isValid = DateTime::isValid(2006, 02, 29); // false
30. dt.assign(2006, DateTime::OCTOBER, 22); // 2006-10-22 00:00:00
31. **if** (dt.dayOfWeek() == DateTime::SUNDAY)
32. {
33. // ...
34. }
35. **return** 0;
36. }

**4. LocalDateTime类**

Poco::LocalDateTime同Poco::DateTime类似，不同的是Poco::LocalDateTime存储一个本地时间。关于本地时间和UTC时间有如下计算公式：  
(UTC时间 = 本地时间 - 时区差).  
  
**构造函数：**  
1. 通过当前时间构造  
2. 通过Timestamp对象  
3. 通过年、月、日、时、分、秒、微秒、毫秒构造  
4. 通过儒略日时间构造(儒略日时间用double存储)  
5. 作为可选项。时区可作为构造时第一个参数被指定。(如果没有指定的话，会使用系统当前的时区)  
  
**成员函数：**  
1. LocalDateTime支持所有的DateTime的函数。  
2. 在进行比较之前，所有的关系操作符函数会把时间都换算为UTC时间  
3. int tzd() const  
   返回时区差  
4. DateTime utc() const  
   转换本地时间到utc时间  
  
下面是一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/LocalDateTime.h"
2. **using** Poco::LocalDateTime;
3. **int** main(**int** argc, **char**\*\* argv)
4. {
5. LocalDateTime now; // the current date and local time
6. **int** year = now.year();
7. **int** month = now.month();
8. **int** day = now.day();
9. **int** dow = now.dayOfWeek();
10. **int** doy = now.dayOfYear();
11. **int** hour = now.hour();
12. **int** hour12 = now.hourAMPM();
13. **int** min = now.minute();
14. **int** sec = now.second();
15. **int** ms = now.millisecond();
16. **int** us = now.microsecond();
17. **int** tzd = now.tzd();
18. **double** jd = now.julianDay();
19. Poco::Timestamp ts = now.timestamp();

22. LocalDateTime dt1(1973, 9, 12, 2, 30, 45); // 1973-09-12 02:30:45
23. dt1.assign(2006, 10, 13, 13, 45, 12, 345); // 2006-10-13 12:45:12.345
24. LocalDateTime dt2(3600, 1973, 9, 12, 2, 30, 45, 0, 0); // UTC +1 hour
25. dt2.assign(3600, 2006, 10, 13, 13, 45, 12, 345, 0);
26. Poco::Timestamp nowTS;
27. LocalDateTime dt3(3600, nowTS); // construct from Timestamp
28. **return** 0;
29. }

**5. Timespan类**

Poco::Timespan能够提供一个微秒精度的时间间隔，也可以用天、小时、分钟、秒、微秒、毫秒来表示。在其内部这个时间间隔用一个64-bit整形来表示。  
  
**构造函数：**  
1. 一个TimeStamp::TimeDiff对象(微秒精度)  
2. 秒+微秒  
   主要用于从timeval结构体构建  
3. 通过日、时、分、秒、微秒构造  
  
**操作符：**  
1. Poco::Timespan支持所有的关系操作符  
(==, !=, <, <=, >, >=)  
2. Poco::Timespan支持加法和减法操作  
(+, -, +=, -=)  
  
**成员函数：**  
1. int days() const  
   返回时间跨度的天  
2. int hours() const  
   返回时间跨度的小时(0 - 23)  
3. int totalHours() const  
   返回时间跨度总的小时数  
4. int minutes() const  
   返回时间跨度的分钟(0 - 59)  
5. int totalMinutes() const  
   返回时间跨度总的分钟数  
6. int seconds() const  
   返回时间跨度的秒(0 - 60)  
7. int totalSeconds() const  
   返回时间跨度总的秒数  
8. int milliseconds() const  
   返回时间跨度的毫秒((0 - 999)  
9. int totalMilliseconds() const  
   返回时间跨度总的毫秒数  
10. int microseconds() const  
   返回时间跨度的微秒( (0 - 999)  
11. int totalMicroseconds() const  
   返回时间跨度总的微秒数  
  
下面是一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/Timespan.h"
2. **using** Poco::Timespan;
3. **int** main(**int** argc, **char**\*\* argv)
4. {
5. Timespan ts1(1, 11, 45, 22, 123433); // 1d 11h 45m 22.123433s
6. Timespan ts2(33\*Timespan::SECONDS); // 33s
7. Timespan ts3(2\*Timespan::DAYS + 33\*Timespan::HOURS); // 3d 33h
8. **int** days = ts1.days(); // 1
9. **int** hours = ts1.hours(); // 11
10. **int** totalHours = ts1.totalHours(); // 35
11. **int** minutes = ts1.minutes(); // 45
12. **int** totalMins = ts1.totalMinutes(); // 2145
13. **int** seconds = ts1.seconds(); // 22
14. **int** totalSecs = ts1.totalSeconds(); // 128722
15. **return** 0;
16. }

下面来看一个DateTime, LocalDateTime和Timespan的混合例子:

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/DateTime.h"
2. #include "Poco/Timespan.h"
3. **using** Poco::DateTime;
4. **using** Poco::Timespan;
5. **int** main(**int** argc, **char**\*\* argv)
6. {
7. // what is my age?
8. DateTime birthdate(1973, 9, 12, 2, 30); // 1973-09-12 02:30:00
9. DateTime now;
10. Timespan age = now - birthdate;
11. **int** days = age.days(); // in days
12. **int** hours = age.totalHours(); // in hours
13. **int** secs = age.totalSeconds(); // in seconds
14. // when was I 10000 days old?
15. Timespan span(10000\*Timespan::DAYS);
16. DateTime dt = birthdate + span;
17. **return** 0;
18. }

**6. Timezone类**

Poco::Timezone提供静态函数用于获取时区信息和夏令时信息。  
1. 时区差  
2. 是否采用夏时制(daylight saving time (DST))  
3. 时区名  
  
**成员函数：**  
1. int utcOffset()  
   返回本地时间相对于UTC时间的差值(精度秒)。不包括夏令时偏移：  
   (local time = UTC + utcOffset())  
2. int dst()  
   返回夏令时偏移。通常是固定值3600秒。0的话表示无夏令时。  
3. bool isDst(const Timestamp& timestamp)  
   对于给定的timestamp时间测试，是否使用夏令时  
4. int tzd()  
   返回本地时间相对于UTC时间的差值(精度秒)。包括夏令时偏移：  
   (tzd = utcOffset() + dst())  
5. std::string name()  
   返回当前的时区名  
6. std::string standardName()  
   返回当前的标准时区名(如果不采用夏令时)  
7. std::string dstName()  
   返回当前的时区名(如果采用夏令时)  
8. 时区名的返回依赖于操作系统，并且名称不具有移植性，仅作显示用。  
  
        下面是一个使用的例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/Timezone.h"
2. #include "Poco/Timestamp.h"
3. **using** Poco::Timezone;
4. **using** Poco::Timestamp;
5. **int** main(**int** argc, **char**\*\* argv)
6. {
7. **int** utcOffset = Timezone::utcOffset();
8. **int** dst = Timezone::dst();
9. **bool** isDst = Timezone::isDst(Timestamp());
10. **int** tzd = Timezone::tzd();
11. std::string name = Timezone::name();
12. std::string stdName = Timezone::standardName();
13. std::string dstName = Timezone::dstName();
14. **return** 0;
15. }

**6. Poco::DateTimeFormatter类**

Poco::DateTimeFormatter用来定义当Timestamp, DateTime, LocalDateTime and Timespan转换为字符串时所需的日期和事件格式。Poco::DateTimeFormatter的作用和strftime()是类似的。Poco::DateTimeFormat内部定义了一些约定的格式。  
1. ISO8601\_FORMAT (2005-01-01T12:00:00+01:00)  
2. RFC1123\_FORMAT (Sat, 1 Jan 2005 12:00:00 +0100)  
3. SORTABLE\_FORMAT (2005-01-01 12:00:00)  
4. For more information, please see the reference documentation.  
  
**成员函数：**  
所有的DateTimeFormatter函数都是静态的。  
  
        下面是一个使用的例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/Timestamp.h"
2. #include "Poco/Timespan.h"
3. #include "Poco/DateTimeFormatter.h"
4. #include "Poco/DateTimeFormat.h"
5. **using** Poco::DateTimeFormatter;
6. **using** Poco::DateTimeFormat;
7. **int** main(**int** argc, **char**\*\* argv)
8. {
9. Poco::DateTime dt(2006, 10, 22, 15, 22, 34);
10. std::string s(DateTimeFormatter::format(dt, "%e %b %Y %H:%M"));
11. // "22 Oct 2006 15:22"
12. Poco::Timestamp now;
13. s = DateTimeFormatter::format(now, DateTimeFormat::SORTABLE\_FORMAT);
14. // "2006-10-30 09:27:44"
15. Poco::Timespan span(5, 11, 33, 0, 0);
16. s = DateTimeFormatter::format(span, "%d days, %H hours, %M minutes");
17. // "5 days, 11 hours, 33 minutes"
18. **return** 0;
19. }

**7. Poco::DateTimeParser类**

        Poco::DateTimeParser用来从字符串中解析时间和日期。下面是其一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. #include "Poco/DateTimeParser.h"
2. #include "Poco/DateTime.h"
3. #include "Poco/DateTimeFormat.h"
4. #include "Poco/LocalDateTime.h"
5. #include "Poco/Timestamp.h"
6. **using** Poco::DateTimeParser;
7. **using** Poco::DateTimeFormat;
8. **using** Poco::DateTime;
9. **int** main(**int** argc, **char**\*\* argv)
10. {
11. std::string s("Sat, 1 Jan 2005 12:00:00 GMT");
12. **int** tzd;
13. DateTime dt;
14. DateTimeParser::parse(DateTimeFormat::RFC1123\_FORMAT, s, dt, tzd);
15. Poco::Timestamp ts = dt.timestamp();
16. Poco::LocalDateTime ldt(tzd, dt);
17. **bool** ok = DateTimeParser::tryParse("2006-10-22", dt, tzd);
18. ok = DateTimeParser::tryParse("%e.%n.%Y", "22.10.2006", dt, tzd);
19. **return** 0;
20. }

**8. Stopwatch类**

        Stopwatch用来测量时间差值，精度为微秒.下面是其定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698425)

1. **class** Foundation\_API Stopwatch
2. /// A simple facility to measure time intervals
3. /// with microsecond resolution.
4. ///
5. /// Note that Stopwatch is based on the Timestamp
6. /// class. Therefore, if during a Stopwatch run,
7. /// the system time is changed, the measured time
8. /// will not be correct.
9. {
10. **public**:
11. Stopwatch();
12. ~Stopwatch();

15. **void** start();
16. /// Starts (or restarts) the stopwatch.
18. **void** stop();
19. /// Stops or pauses the stopwatch.
21. **void** reset();
22. /// Resets the stopwatch.
24. **void** restart();
25. /// Resets and starts the stopwatch.
27. Timestamp::TimeDiff elapsed() **const**;
28. /// Returns the elapsed time in microseconds
29. /// since the stopwatch started.
31. **int** elapsedSeconds() **const**;
32. /// Returns the number of seconds elapsed
33. /// since the stopwatch started.

36. **static** Timestamp::TimeVal resolution();
37. /// Returns the resolution of the stopwatch.

40. **private**:
41. Stopwatch(**const** Stopwatch&);
42. Stopwatch& operator = (**const** Stopwatch&);

45. Timestamp           \_start;
46. Timestamp::TimeDiff \_elapsed;
47. **bool**                \_running;
48. };

**附录：**

1.  The Gregorian Calendar： <http://en.wikipedia.org/wiki/Gregorian_calendar>  
2. The Julian Calendar：<http://en.wikipedia.org/wiki/Julian_calendar>  
3. The Julian Day：<http://en.wikipedia.org/wiki/Julian_day>  
4. Coordinated Universal Time (UTC)：<http://en.wikipedia.org/wiki/UTC>  
5.  ISO 8601：<http://en.wikipedia.org/wiki/ISO_8601>

（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8698425>）