C++ 之日期与时间

吴咏炜

博览首席咨询师

A long time ago in a continent far, far away. . . .

EPISODE I

THE DAWN OF AN EPOCH

Master Ken Thompson invented UNIX, with joined force from Master Dennis Ritchie, who created C. Their partnership was as powerful as the Force itself, and changed the shape of the computing universe for ever.

THE UNIX EPOCH

```
time_t t = 0;
```

THE UNIX EPOCH

```
time_t t = 0;
cout << ctime(&t);
```

→ Thu Jan 1 08:00:00 1970

THE UNIX EPOCH

```
time_t t = 0;
auto tm_ptr = gmtime(&t);
cout << asctime(tm_ptr);
```

→ Thu Jan 1 00:00:00 1970

THE UNIX EPOCH

```
auto tm_ptr = gmtime(&t);
Time time_to_print{};
memcpy(&time_to_print, tm_ptr,
       sizeof(Time));
mozi::println(time_to_print);
```

→

```
{
    tm_sec: 0,
    tm_min: 0,
    tm_hour: 0,
    tm_mday: 1,
    tm_mon: 0,
    tm_year: 70,
    tm_wday: 4,
    tm_yday: 0,
    tm_isdst: 0,
    tm_gmtoff: 0,
    tm_zone: "UTC"
}
```

THE UNIX EPOCH

```
auto tm_ptr = localtime(&t);
Time time_to_print{};
memcpy(&time_to_print, tm_ptr,
       sizeof(Time));
mozi::println(time_to_print);
cout << asctime(tm_ptr);
```

→

```
{
    tm_sec: 0,
    tm_min: 0,
    tm_hour: 8,
    tm_mday: 1,
    tm_mon: 0,
    tm_year: 70,
    tm_wday: 4,
    tm_yday: 0,
    tm_isdst: 0,
    tm_gmtoff: 28800,
    tm_zone: "CST"
}
Thu Jan  1 08:00:00 1970
```

THE UNIX EPOCH

此时区写法仅适用于 Unix 系统

```
setenv("TZ", ":US/Pacific", 1);
tzset();
auto tm_ptr = localtime(&t);
Time time_to_print{};
memcpy(&time_to_print, tm_ptr,
       sizeof(Time));
mozi::println(time_to_print);
cout << asctime(tm_ptr);
```

→

```
{
    tm_sec: 0,
    tm_min: 0,
    tm_hour: 16,
    tm_mday: 31,
    tm_mon: 11,
    tm_year: 69,
    tm_wday: 3,
    tm_yday: 364,
    tm_isdst: 0,
    tm_gmtoff: -28800,
    tm_zone: "PST"
}
Wed Dec 31 16:00:00 1969
```

可重入问题

以下函数都会修改全局状态，因而不适合多线程环境

- `asctime`, `ctime`, `gmtime`, `localtime`, `tzset`

替代品

- `asctime`, `ctime` → `strftime`（标准 C/C++），`asctime_r`, `ctime_r`（POSIX）
- `gmtime`, `localtime` → `gmtime_r`, `localtime_r`（POSIX）
- `tzset` →（无，且只属于 POSIX 而不属于标准 C/C++）

格式化时间 – strftime

```
setenv("TZ", ":UTC", 1);
tzset();
tm tm_data{};
localtime_r(&t, &tm_data);
char buffer[40];
strftime(buffer, sizeof buffer, "%F %T %Z", &tm_data);
puts(buffer);
```

→ 1970-01-01 00:00:00 UTC

从本地日历时间（tm）生成时间 – mktime（C89）

```
tm tm_data{};
tm_data.tm_year = 2023 - 1900;
tm_data.tm_mon = 12 - 1;
tm_data.tm_mday = 17;
tm_data.tm_hour = 14;
time_t t = mktime(&tm_data);
cout << t << '\n';
```

→ 1702792800

注意：结果受当前时区设置的影响

获取当前时间 – time (C89)

Unix 时间本质上是协调世界时（UTC）

```
time_t t;
time(&t);
localtime_r(&t, &tm_data);
char buffer[40];
strftime(buffer, sizeof buffer, "%F %T %Z", &tm_data);
puts(buffer);
```

→ 2023-12-03 16:21:38 CST

获取当前时间 – gettimeofday (POSIX)

```
timeval tv;
gettimeofday(&tv, nullptr);
localtime_r(&tv.tv_sec, &tm_data);
char buffer[40];
strftime(buffer, sizeof buffer, "%F %T", &tm_data);
printf("%s.%06d\n", buffer, static_cast<int>(tv.tv_usec));
```

→ 2023-12-03 16:42:20.152768

获取当前时间 – clock_gettime (POSIX)

```
timespec ts;
clock_gettime(CLOCK_REALTIME, &ts);
localtime_r(&ts.tv_sec, &tm_data);
char buffer[40];
strftime(buffer, sizeof buffer, "%F %T", &tm_data);
printf("%s.%09ld\n", buffer, ts.tv_nsec);
```

→ 2023-12-03 16:48:48.059124566

获取当前时间 – clock_gettime (POSIX)

```
timespec ts;
clock_gettime(CLOCK_REALTIME, &ts);
localtime_r(&ts.tv_sec, &tm_data);
char buffer[40];
strftime(buffer, sizeof buffer, "%F %T", &tm_data);
printf("%s.%09ld\n", buffer, ts.tv_nsec);
```

→ 2023-12-03 16:48:48.059124566

获取当前时间 – clock_gettime (POSIX)

```
timespec ts;
clock_gettime(CLOCK_MONOTONIC, &ts);
printf("%ld.%09ld\n",
       static_cast<long>(ts.tv_sec), ts.tv_nsec);
```

→ 10458.487359516

废柴？– clock（C89）

函数	精度	耗时（时钟周期）	说明
clock (Windows)	1 ms	~160	墙钟时间
clock (Linux)	1 μs	~1800	进程时间
QueryPerformanceCounter (Windows)	0.1 μs	~61	性能计数器
gettimeofday (Linux)	1 μs	~69	墙钟时间

目前看到的问题

问题	解决方案
没有跨平台的高精度时间类型	?
没有跨平台的稳定时钟接口（性能测试常用）	?
`time_t` 是弱类型，对强类型语言仍不够“好”	?
没有表示秒之外的时长的方便方法	?
表示日期较为麻烦	?
不支持方便的日期运算	?
语言层面缺乏标准化的时区支持	?
获得和打印时间的代码常常较为冗长	?

EPISODE II

RETURN OF A LANGUAGE

C++11 is making a triumphant return, armed with innovative capabilities. The Chrono library emerges as a new frontier, brimming with user-friendly tools and facilities as if crafted by skilled droid engineers.

C++11 的时长类型

```
template <
    class Rep,
    class Period = std::ratio<1>
> class duration;
```

底层类型，可以是整型或浮点型

比例类型，kilo、milli 等已预定义

标准时长类型

类型	定义
std::chrono::**nanoseconds**	std::chrono::duration</* *int64* */, std::nano>
std::chrono::**microseconds**	std::chrono::duration</* *int55* */, std::micro>
std::chrono::**milliseconds**	std::chrono::duration</* *int45* */, std::milli>
std::chrono::**seconds**	std::chrono::duration</* *int35* */>
std::chrono::**minutes**	std::chrono::duration</* *int29* */, std::ratio<60>>
std::chrono::**hours**	std::chrono::duration</* *int23* */, std::ratio<3600>>

至少可以涵盖 ±292 年!

时长类型间的转换

通过数值构造时长

秒可以安全无损地转换为毫秒

反向转换可能有损失，不允许隐式

但可以显式要求转换

```
using namespace std::chrono;
seconds s{1};
milliseconds ms = s;
s = ms;
s = duration_cast<seconds>(ms);
duration<double> fs = ms;
ms = fs;
cout << fs.count() << "s\n";
```

毫秒也可以隐式转换成浮点形式的秒

同样不允许隐式自动转换

成员函数可取出底层数值

标准时长字面量后缀（C++14）

类型	字面量后缀
std::chrono::nanoseconds	ns
std::chrono::microseconds	us
std::chrono::milliseconds	ms
std::chrono::seconds	s
std::chrono::minutes	min
std::chrono::hours	h

C++11 的时间点类型

```
template <
    class Clock,
    class Duration = typename Clock::duration
> class time_point;
```

关联的时钟类型

关联的时长类型

时间点和时长的操作

- 时长 ± 时长 → 时长
- 时间点 ± 时长 → 时间点
- 时间点 – 时间点 → 时长
- 时长 * 标量 → 时长
- 时长 / (或 %) 标量 → 时长
- 时长 / (或 %) 时长 → 标量

```
time_point<system_clock, seconds> now{1702792800s};
now -= 24h;
now += 15h + 25min - 14h;
time_t t = now.time_since_epoch().count();
cout << put_time(localtime(&t), "%F %T") << '\n';    →    2023-12-16 15:25:00
static_assert(1h * 10 == 10h);
static_assert(1h / 1min == 60);
```

时钟的基本接口

duration	time_point	is_steady	now()
• 该时钟的时长类型	• 该时钟的时间点类型	• 该时钟是否稳定的属性：单调增长且增长速度恒定	• 获得当前时间点的函数

三种不同的时钟

- `system_clock`
  - 主流平台满足 `!system_clock::is_steady`
- `steady_clock`
  - 标准要求满足 `steady_clock::is_steady`
- `high_resolution_clock`
  - 一般为别名，定义为 `system_clock` 或 `steady_clock` 两者之一
  - 一般不推荐使用

性能测试的可能做法

```
auto start = steady_clock::now();
some_calculation();
auto end = steady_clock::now();
cout << "Some calculation took "
     << (end - start) / 1us << "us\n";
```

重温问题

问题	解决方案
没有跨平台的高精度时间类型	C++11 的 time_point 模板
没有跨平台的稳定时钟接口（性能测试常用）	C++11 的 steady_clock 时钟
time_t 是弱类型，对强类型语言仍不够“好”	C++11 区分 time_point 和 time_duration，并支持常用操作
没有表示秒之外的时长的方便方法	C++14 的时长字面量（如 500ms）
表示日期较为麻烦	?
不支持方便的日期运算	?
语言层面缺乏标准化的时区支持	?
获得和打印时间的代码常常较为冗长	?

EPISODE III

LONG LIVE THE KING

C++20 ascends with the power of the new Chrono. It rules with civil calendar support, precise time points, and respects the leap seconds. Its court includes versatile formatting and parsing functions. Vive le roi.

时长和系统时间点的输出支持

```
auto tp = system_clock::now();
cout << tp << '\n';
this_thread::sleep_for(100ms);
cout << duration_cast<milliseconds>(system_clock::now() - tp)
     << '\n';
```

→

```
2023-12-05 06:23:49.510617
102ms
```

需要新版的编译器：GCC 13、Clang 17 等

日期和日期字面量

```
cout << 2023y/12/17 << '\n';
cout << 12/17d/2023 << '\n';
cout << 17d/12/2023 << '\n';
cout << 17d/December/2023 << '\n';
cout << December/17/2023 << '\n';
cout << 2023y/December/17 << '\n';
static_assert(is_same_v<decltype(2023y/12/17), year_month_day>);
static_assert(is_same_v<decltype(December/17), month_day>);
```

2023-12-17

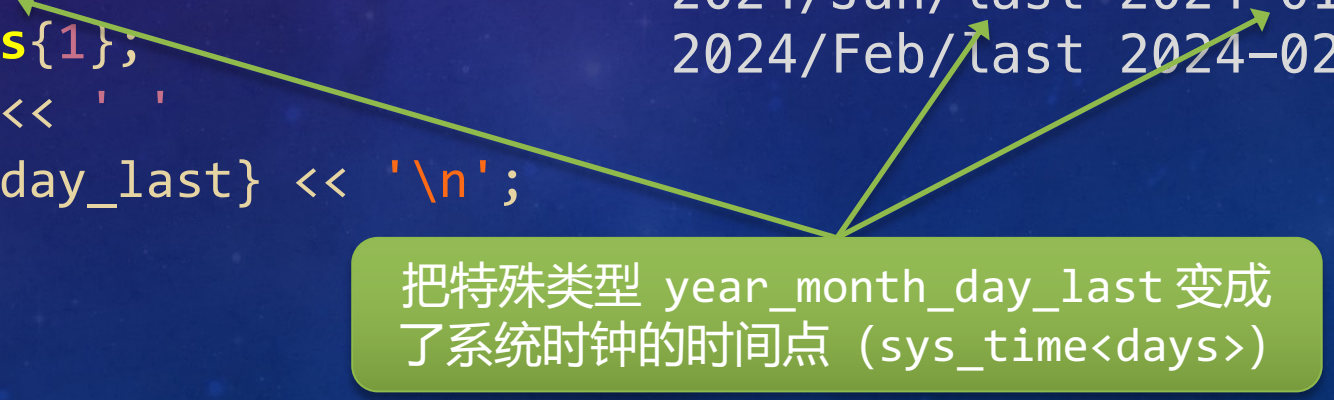
日期的加减法

```
auto day31 = 2024y/1/31;
cout << day31 + months{1} << '\n';
cout << day31 + months{2} << '\n';

auto day_last = 2024y/1/last;
cout << day_last << ' '
     << sys_days{day_last} << '\n';
day_last += months{1};
cout << day_last << ' '
     << sys_days{day_last} << '\n';
```

```
2024-02-31 is not a valid date
2024-03-31

2024/Jan/last 2024-01-31
2024/Feb/last 2024-02-29
```



时间点的加减法

```
auto tp = sys_seconds{sys_days{day_last}};
tp += months{1};
cout << tp << '\n';
```

→ 2024-03-30 10:29:06

C++20 新增的时长类型

类型	定义
`std::chrono::days`	`std::chrono::duration</* int25 */, std::ratio<86400>>`
`std::chrono::weeks`	`std::chrono::duration</* int22 */, std::ratio<604800>>`
`std::chrono::months`	`std::chrono::duration</* int20 */, std::ratio<2629746>>`
`std::chrono::years`	`std::chrono::duration</* int17 */, std::ratio<31556952>>`

至少可以涵盖 ±40,000 年!

日期和时间的组合

```
auto sys_tp = sys_days{2023y/12/17} + 14h + 25min;
static_assert(is_same_v<decltype(sys_tp),
                        sys_time<minutes>>);
cout << sys_tp << '\n';
```

→ 2023-12-17 14:25:00

格式化输出

```
format("{}", sys_tp)                    2023-12-17 14:25:00
format("{:%F %T}", sys_tp)               2023-12-17 14:25:00
format("{:%d-%b-%Y %H:%M}", sys_tp)      17-Dec-2023 14:25
format("{:%F %T %Z}", sys_tp)            2023-12-17 14:25:00 UTC
```

本地时间和时区

得到本地（伪）时钟里的天数

```
auto local_tp = local_days{2023y/12/17} + 14h + 25min;
cout << local_tp << '\n';                                  2023-12-17 14:25:00

auto* local_tz = get_tzdb().current_zone();
auto zoned_tp = zoned_time{local_tz, local_tp};
cout << zoned_tp << '\n';                                  2023-12-17 14:25:00 CST

cout << sizeof local_tp << '\n';                           8
cout << sizeof zoned_tp << '\n';                           16
```

把时间点绑定当前时区上

指定时区

```
auto* pst_tz = get_tzdb().locate_zone("US/Pacific");
auto pst_tp = zoned_time{pst_tz, zoned_tp};
cout << pst_tp << '\n';
```

→ 2023-12-16 22:25:00 PST

今天是几号?

和 duration_cast 相似，
但转换的是时间点而非时长

```
auto now = system_clock::now();
cout << "UTC today:   "
     << floor<days>(now) << '\n';
auto local_now = local_tz->to_local(now);
cout << "Local today: "
     << floor<days>(local_now) << '\n';
auto pst_now = pst_tz->to_local(now);
cout << "PST today:   "
     << floor<days>(pst_now) << '\n';
```

```
UTC today:   2023-12-17

Local today: 2023-12-17

PST today:   2023-12-16
```

母亲节是几号?

```
for (auto yr = 2021y; yr != 2025y; ++yr) {
    auto mothers_day = yr / May / Sunday[2];
    cout << local_days{mothers_day} << '\n';
}
```

```
2021-05-09
2022-05-08
2023-05-14
2024-05-12
```

夏令时的麻烦 I

```cpp
auto dst_start_day = 2023y / March / Sunday[2];
auto local_tp = local_days{dst_start_day} + 1h + 59min + 59s;
auto* pst_tz = get_tzdb().locate_zone("US/Pacific");
sys_time<milliseconds> sys_tp = pst_tz->to_sys(local_tp);
for (int i = 0; i < 5; ++i) {
    auto zoned_tp = zoned_time{pst_tz, sys_tp};
    cout << zoned_tp << '\n';
    sys_tp += 500ms;
}
```

```
2023-03-12 01:59:59.000 PST
2023-03-12 01:59:59.500 PST
2023-03-12 03:00:00.000 PDT
2023-03-12 03:00:00.500 PDT
2023-03-12 03:00:01.000 PDT
```

夏令时的麻烦 II

```cpp
local_tp = local_days{dst_start_day} + 2h + 30min;
try {
    cout << zoned_time(pst_tz, local_tp) << '\n';
}
catch (nonexistent_local_time& e) {
    cout << "*** Error:\n" << e.what() << '\n';
}
```

```
*** Error:
2023-03-12 02:30:00 is in a gap between
2023-03-12 02:00:00 PST and
2023-03-12 03:00:00 PDT which are both equivalent to
2023-03-12 10:00:00 UTC
```

中国什么时候用过夏令时?

```
auto* tz_ptr = get_tzdb().locate_zone("Asia/Shanghai");
for (auto yr = 1980y; yr != 2000y; ++yr) {
    auto local_tp = local_days{yr / June / 1};
    auto sys_tp = sys_days{yr / June / 1};
    auto diff = sys_tp - tz_ptr->to_sys(local_tp);
    if (diff != 8h) {
        cout << yr << '\n';
    }
}
```

当前只有 GCC 13 的 libstdc++ 可以工作……

问题和解决方案小结

问题	解决方案
没有跨平台的高精度时间类型	C++11 的 time_point 模板
没有跨平台的稳定时钟接口（性能测试常用）	C++11 的 steady_clock 时钟
time_t 是弱类型，对强类型语言仍不够“好”	C++11 区分 time_point 和 time_duration，并支持常用操作
没有表示秒之外的时长的方便方法	C++14 的时长字面量（如 500ms）
表示日期较为麻烦	C++20 的日期字面量（如 2023y/December/17）
不支持方便的日期运算	C++20 的日期支持
语言层面缺乏标准化的时区支持	C++20 的时区支持
获得和打印时间的代码常常较为冗长	C++20 的格式化支持

C++20 的其他时钟

- 真正的 UTC 时钟（`utc_clock`）
  - 有闰秒，某些分钟有 61 秒（可能出现 23:59:60）；一年的总秒数可能不是 86400 的整数倍
- GPS 时钟（`gps_clock`）
  - 起始点不同；由于闰秒，一年可能比实际的一年要短；目前比 UTC 时钟快 18 秒
- 国际原子钟（`tai_clock`）
  - 和 GPS 时钟相似，但起始点不同；恒定比 GPS 时钟快 19 秒
- 文件时钟（`file_clock`）
  - 用于文件系统，细节由实现定义

```
Sys clock: 2023-12-07 03:56:06.596 UTC
UTC clock: 2023-12-07 03:56:06.596 UTC
GPS clock: 2023-12-07 03:56:24.596 GPS
TAI clock: 2023-12-07 03:56:43.596 TAI
```

其他……

- 对 12/24 小时制的支持
- 对“当天时刻” 的支持
- 不同时钟之间的转换
- 对时间字符串流的解析
- ……

示例代码

- https://github.com/adah1972/cpp_summit_2023

Special thanks to GPT-4 for the episode scripts. . . .

谢谢！