

## 16 | 标准库：日期、时间与实用函数

2022-01-19 于航

《深入C语言和程序运行原理》

课程介绍 >



讲述：于航

时长 15:17 大小 14.00M



你好，我是于航。

在前面的几讲中，我都以较大的篇幅介绍了 C 标准库中的一些重要概念，和相关接口的使用方式。除此之外，标准库中还有一些功能十分明确，使用方式也十分简单的常用接口，这些接口也为日常的 C 应用开发提供了重要支持。因此，在接下来的两讲中，我将围绕这部分内容展开介绍。

今天，我们先来看看标准库中与日期、时间以及实用函数有关的内容。其中，日期与时间的相关接口由**头文件 time.h** 提供；而实用函数的功能则可被进一步细分为字符串与数值转换、随机数生成、动态内存管理，以及进程控制等不同的几类，这些功能对应的编程接口均由**头文件 stdlib.h** 提供。



下面，我们就来分别看看这两类接口的使用方式，以及它们背后的一些基本原理。

# 日期与时间

首先来看由头文件 `time.h` 提供的日期与时间相关接口。那么，在 C 语言中，日期与时间的概念是怎样体现的？又应该如何对它们进行操作和转换呢？

在构建应用程序时，我们经常会用到日期与时间这两种概念。比如，在记录日志时，通常需要保存每个事件的确切发生日期和时间；在进行优化时，则需要通过测量代码的运行时间来寻找性能痛点；而在生成随机数时，甚至需要使用当前时间，作为不同的随机种子。

看到这里，你可能已经发现了：这一讲中我们提到的“时间”，有两种不同的含义。一种是指时间上的跨度，而另一种则指以小时、分钟、秒组成的确切时间点。至于具体是哪一种含义，需要你结合上下文来理解。当然，在可能会引起误解的地方，我也会特别说明下。

## 日历时间

在 C 语言中，时间可以被分为“日历时间（Calendar Time）”与“处理器时间（Processor Time）”。其中，前者是指从世界协调时间（UTC）1970 年 1 月 1 日 00 时 00 分 00 秒，到当前 UTC 时间所经历的秒数，其中不包括闰秒。在 C 标准库中，该值由自定义的类型关键字 `time_t` 表示。该类型通常对应于一个整数类型，在某些老版本的标准库中，可能被实现为 32 位有符号整型。

同时，`time.h` 头文件也提供了一个非常直观的，名称为 `time` 的方法，可用于获取这个值。来看下面这个例子：


```
1 #include <stdio.h>
2 #include <time.h>
3 int main(void) {
4     time_t currTime = time(NULL);
5     if(currTime != (time_t)(-1))
6         printf("The current timestamp is: %ld(s)", currTime);
7     return 0;
8 }
```

复制代码

领资料

这里，我们将 `time` 方法调用后的返回值保存在了变量 `currTime` 中。而当方法调用成功后（即返回值不为 -1），该值通过 `printf` 函数被打印了出来。

不仅如此，在获取到这个整型时间值后，借由标准库提供的其他时间与日期处理函数，我们还可以对它做进一步处理。比如，将它格式化为本地时间，并以特定的格式输出。继续来看下面这个例子：

 复制代码

```
1 #include <stdio.h>
2 #include <time.h>
3 int main(void) {
4     time_t currTime = time(NULL);
5     if(currTime != (time_t)(-1)) {
6         char buff[64];
7         struct tm* tm = localtime(&currTime);
8         if (strftime(buff, sizeof buff, "%A %c", tm))
9             printf("The current local time is: %s", buff); // "The current local tir
10    }
11    return 0;
12 }
```

在这段代码中，我们用与之前类似的方式获取了当前的日历时间，该值被存放到变量 **currTime** 中。在代码的第 7 行，通过使用名为 **localtime** 的方法，我们可以将该日历时间转换成与本地时间相关的多种信息。这些信息将以不同字段的形式被存放在名为 **tm** 的结构对象中。

接着，通过调用 **strftime** 方法，我们可以继续对这个时间对象进行格式化。该方法调用后，会将生成的结果字符串存放到由变量 **buff** 对应的字符数组中。这里，传入的第三个参数为一个包含有格式控制占位符的字符串。其中，**%A** 用于显示完整的周工作日名称，**%c** 用于显示标准日期和时间字符串。**strftime** 方法将根据占位符字符串的具体组成格式，来输出相应的结果字符串。

## 处理器时间

接着，我们再来看处理器时间（**CPU Time**）。顾名思义，处理器时间，即 **CPU** 资源被调度以支持程序在某段时间内正常运作所花费的时间。需要注意的是，在默认情况下，这个时间应该是应用运行所涉及的所有独立 **CPU** 所消耗时间的总和。**C** 标准库也为我们提供了一个直观的，名称为 **clock** 的方法，可用于返回这个值。来看下面这个例子：

 复制代码

```
1 #include <time.h>
2 #include <stdio.h>
```

领资料

```

3  int main(void) {
4      clock_t startTime = clock();
5      for(int i = 0; i < 100000000; i++) {}
6      clock_t endTime = clock();
7      printf("Consumed CPU time is: %fs\n",
8             (double)(endTime - startTime) / CLOCKS_PER_SEC);
9      return 0;
10 }

```

位于代码第 4 行的 `clock` 方法在调用后会返回类型为 `clock_t` 的值。该类型由标准库的具体实现定义，因此，其值可能为整数，也可能为浮点数。不同于日历时间的是，为了更精确地计算 CPU 耗时，处理器时间并不直接以“秒”为单位，而是以“clock tick”为单位。为了将这个时间换算为秒，你需要将它除以标准库中提供的宏常量 `CLOCKS_PER_SEC`。该常量表明了，在当前系统上每 1 秒钟对应的 clock tick 次数。

需要注意的是，这里我们提到的 clock tick，与程序运行所在计算机的实际物理 CPU 频率没有直接关系。应用程序可以通过读取计算机上的硬件定时器，来获得对应进程的 CPU 使用时间。

对于程序运行来说，一段时间内花费的处理器时间与墙上时钟时间（Wall-clock Time）可能并不一致。前者依赖于程序使用的线程数量、所在平台的物理 CPU 核数，以及操作系统调度 CPU 的具体策略等。而后者则是现实世界的时间流逝，也就是一个恒定递增的值。因此，调用一次 `clock` 方法所返回的处理器时间一般没有太多意义。通常，我们会按照上面例子中的方式来使用这个时间，即在一段代码的前后，分两次获取处理器时间，并通过计算两者之间的差值，来了解 CPU 执行这段代码所花费的时间。

当然，为了方便你更好地理解处理器时间与墙上时钟时间的区别，你可以在具有多核 CPU 的计算机上编译和运行下面这段代码，并在评论区告诉我你的运行结果。关于代码的具体实现细节，你可以尝试参考注释进行理解，如果有问题，也可以在评论区随时跟我讨论。

```

1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4  #include <threads.h>
5  typedef struct timespec ts_t;
6  int run(void* data) { // 模拟的耗时任务;
7      volatile double d = 0;
8      for (int n = 0; n < 10000; ++n)
9          for (int m = 0; m < 10000; ++m)

```

复制代码

领资料

```

10     d += d * n * m;
11     return 0;
12 }
13 int main(void) {
14     // 首次记录日历时间与处理器时间;
15     ts_t ts1;
16     timespec_get(&ts1, TIME_UTC);
17     clock_t t1 = clock();
18     // 创建两个线程, 做一些耗时任务;
19     thrd_t thr1, thr2;
20     thrd_create(&thr1, run, NULL);
21     thrd_create(&thr2, run, NULL);
22     thrd_join(thr1, NULL);
23     thrd_join(thr2, NULL);
24     // 再次记录日历时间与处理器时间;
25     ts_t ts2;
26     timespec_get(&ts2, TIME_UTC);
27     clock_t t2 = clock();
28     // 分别计算和打印处理器时间与墙上时钟时间耗时;
29     printf("CPU time used (per clock()): %.2f ms\n", 1000.0 * (t2 - t1) / CLOCKS_
30     printf("Wall time passed: %.2f ms\n",
31         1000.0 * ts2.tv_sec + 1e-6 * ts2.tv_nsec - (1000.0 * ts1.tv_sec + 1e-6 * ts
32     return 0;
33 }

```

## 其他相关处理函数

除了我在上面介绍过的一些常见日期与时间处理函数外，C 标准库还提供了另外一些相关函数，我将它们整理在了下面的表格中，供你参考。当然，这里并没有包含那些已经被标记为“废弃”的接口。你可以点击 [这个链接](#)，来查看更多信息。

方法名	功能简介
difftime	计算两个以 time_t 类型表示的日历时间之间的差值
timespec_get	获取基于给定时间基准（如 TIME_UTC）的日历时间
timespec_getres	获取基于给定时间基准（如 TIME_UTC）的时间分辨率信息
wcsftime	将给定的 tm 结构体对象转换为对应的宽字符表示
gmtime	将给定的 time_t 值转换为对应的 tm 结构体对象
mktime	将给定的 tm 结构体对象转换为对应的 time_t 值

领资料



到这里，我讲完了如何在 C 代码中使用日期和时间操作的相关函数，接下来，我想和你讨论一个可能由 `time_t` 类型引发的问题。

我在讲日历时间的时候提到过，某些旧版本的 C 标准库在实现用于存放日历时间的 `time_t` 类型时，可能会采用 32 位有符号整数。而在这种情况下，`time_t` 所能够表示的时间跨度便会大大缩小，并会在不久之后的 UTC 时间 2038 年 1 月 19 日 03 时 14 分 08 秒发生上溢出。当该类型变量溢出后，其表示的具体日期和时间，将会从 1901 年开始“重新计时”。你可以通过下图（图片来自 [Wikipedia](#)）来观察这个问题的发生过程。


```
Binary   : 01111111 11111111 11111111 11110000
Decimal  : 2147483632
Date     : 2038-01-19 03:13:52 (UTC)
Date     : 2038-01-19 03:13:52 (UTC)
```

可以说，这是一个全球性问题，严重性甚至可以与 Y2K 等问题比肩。由于 C 语言被广泛应用于各类软硬件系统中，因此，从常见的交通设施、通信设备，到某些早期的计算机操作系统，它们都可能会在那时受到 Y2038 问题的影响。

看完了与头文件 `time.h` 相关的内容，接着，我们再来看看 `stdlib.h` 头文件提供的众多实用函数。由于这些函数的功能十分混杂，我将它们分为了几个不同的类别，来分别为你介绍。首先来看数值与字符串转换的相关接口。

## 字符串到数值的转换

这类接口的使用方式都十分简单，直接来看下面这段代码：

 复制代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 int main(void) {
5     // 一次性字符串到数值转换；
6     const char* strA = "1.0";
7     printf("%f\n", atof(strA));
8     // 带溢出检查的转换函数，执行后会保存不能被转换部分的地址；
9     const char* strB = "20000000000000000000000000000000.0";
10    char* end;
11    double num = strtol(strB, &end, 10);
12    if(errno == ERANGE) { // 判断转换结果是否发生溢出；
```

领资料



```
13     printf("Range error, got: ");
14     errno = 0;
15 }
16 printf("%f\n", num);
17 return 0;
18 }
```

`stdlib.h` 头文件提供了众多函数，可用于将一个字符串转换为特定类型的数字值。在上面代码的第 7 行，我们使用名为 `atof` 的函数，将字符串 `strA` 转换为一个双精度浮点数。实际上，以字母“a”开头的这类函数，只能对字符串进行一次性转换。

相对地，在代码的第 11 行，名为 `strtol` 的函数将字符串 `strB` 转换为了对应的长整型数值。而这一类以“str”开头的函数，会在每次执行时判断转换结果是否发生溢出，同时保存不能被转换部分的地址。在这种情况下，通过再次调用这类函数，我们便能够对剩余部分的字符串继续进行转换，直至将整个字符串处理完毕。关于这些函数的更多信息，你可以点击 [这个链接](#) 进行查看。

## 生成随机数

作为实用函数的一部分，“随机数生成”是一个不可或缺的重要功能。同样地，我们也可以通过配合使用 `stdlib.h` 提供的 `rand` 与 `srand` 方法，来生成随机数。它们的基本用法如下所示：

 复制代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 int main (void) {
5     srand(time(NULL)); // 初始化随机数种子;
6     while (getchar() == '\n')
7         printf("%d", rand() % 10); // 生成并打印 0-9 的随机数;
8     return 0;
9 }
```

领资料

可以看到，我们首先在代码的第 5 行，使用 `srand` 方法设定了程序每次运行时需要使用的随机数种子。在代码第 7 行，`rand` 函数的调用会产生范围为 `[0, RAND_MAX]` 的随机数。通过对它进行求余处理，可以将结果限定到一个指定的范围。

对于大多数 C 标准库实现来说，rand 函数在内部会采用“线性同余发生器（Linear Congruential Generator）”等伪随机算法，来计算函数每次调用时需要产生的随机数。这也就意味着，该函数产生的随机数，本质上并不是随机的。如果我们没有使用 srand 函数设置新的随机数种子，那么，当每次程序重新运行时，通过 rand 函数产生的随机数序列都将会是相同的。而这个种子便会作为 rand 函数在计算下一个随机数时，所采用算法的输入参数。

事实上，计算机无法生成“真正的随机数”。正如 MIT 教授 Steve Ward 说的那样：“传统计算机系统最不擅长的一件事就是抛硬币”。计算机软件的执行会按照既定的算法展开，因此，当输入和算法不变时，输出结果就变得有迹可循。即使我们可以用更复杂的算法，来让输出的变化模式变得难以琢磨，但无论如何，这都并非真正的随机。

而伪随机数算法之所以可被用来生成随机数，则是因为从统计学角度来讲，其生成的数字符合随机数在均匀性、独立性等特征上的要求。并且，伪随机数的生成不需要特殊的硬件支持。同时，在大多数场景中，伪随机数也可以满足基本的使用需求。

## 动态内存管理

动态内存管理，本质上就是堆内存管理。我曾在 [🔗08 讲](#) 中介绍过 VAS 中堆的概念，以及如何使用 malloc 与 free 函数，来在堆上分配和释放一段内存空间。但实际上，除这两个函数外，C 标准库还为我们提供了另外一些函数，可用于在分配堆内存时进行更加精确的控制。你可以通过下面的表格，来了解这几个函数的基本功能。由于它们的使用方式较为简单，这里我就不详细介绍了。

方法名	功能简介
aligned_alloc	从符合对齐要求的地址分配指定大小的堆内存
calloc	分配指定大小的堆内存，并将其清零
realloc	扩大或缩小之前已经分配的堆内存



## 进程控制

接下来，我们来看看实用函数中与进程控制相关的内容。虽然 C 标准库为我们提供了进程控制的相关能力，但这个能力实际上却十分有限。借助标准库提供的接口，我们可以控制程序的



退出形式（正常终止、异常终止、快速终止等），获取当前系统的环境变量，或是与宿主机上的命令处理器进行交互。但除此之外，我们无法再控制进程的其他行为，比如创建进程，或使用进程间通信。对于其中几个函数的使用方式，你可以参考下面这个例子：

复制代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void exitHandler() {
4     printf("%s\n", getenv("PATH"));
5 }
6 int main(void) {
7     if (!atexit(exitHandler)) {
8         exit(EXIT_SUCCESS);
9     }
10    return 0;
11 }
```

这里，在代码的第 7 行，我们使用函数 `atexit` 为程序注册了一个回调函数。这个函数会在程序显式调用 `exit` 函数时，或从 `main` 函数内正常退出时被触发。在对应的回调函数 `exitHandler` 中，我们使用 `getenv` 函数，获取并打印了当前宿主机上环境变量 `PATH` 的值。当回调函数注册成功后（返回整型数值 0），通过显式调用函数 `exit`，我们正常退出了当前程序。此时，回调函数被调用，环境变量 `PATH` 的值被打印了出来。

还有一些我没提到的函数，它们的使用方式也很简单。同样地，我将它们整理在了下面的表格中，供你参考。

方法名	功能简介
abort	导致一个异常程序终止，不做资源清理
quick_exit	导致一个快速程序终止，仅做不完全的资源清理
_Exit	导致一个正常程序终止，不做资源清理
at_quick_exit	设置回调函数，在 quick_exit 调用时被调用
system	与宿主机上的命令行通信

领资料

这里需要你注意，`exit`、`quick_exit`、`_Exit`，以及 `abort` 这四个可用于终止程序运行的函数，它们实际上对应着不同的使用场景。

其中，`exit` 函数在退出程序时，会进行一系列资源清理工作，比如冲刷并关闭各种 IO 流、移除由函数 `tmpfile` 创建的临时文件等。除此之外，它还会在中途触发用户注册的回调函数，以进行自定义的收尾工作。但相对地，`quick_exit` 函数在终止程序时，并不会进行上述资源清理工作。它仅会通过回调函数，来执行用户自定义的收尾工作。`_Exit` 函数则更加彻底，它会直接终止程序的执行，而不做任何处理。

不同于这三类函数，`abort` 函数在调用时，会向当前程序发送信号 `SIGABRT`，并根据情况，选择终止程序或执行相应的信号处理程序。

## 其他接口

除了上面提到的这几类重要接口外，`stdlib.h` 中还包含有一些与搜索排序、整数算数、宽字符（串）转换相关的接口。它们的使用方式也十分简单，这里我就不一一介绍了。你可以参考下表来了解这些最常用接口的名称与功能，也可以点击 [这个链接](#) 来了解有关它们的更多信息。

方法名	功能简介
<code>qsort</code>	对指定数组中的元素进行 in-place 排序
<code>bsearch</code>	从指定数组中查找给定的元素，返回指向该元素的指针或 <code>NULL</code>
<code>abs</code>	计算给定值的绝对值
<code>div</code>	计算整数除法的商和余数
<code>mblen</code>	返回下一个多字节字符的字节数
<code>mbtowc</code>	将下一个多字节字符转换为宽字符

 极客时间

 领资料

## 总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我主要介绍了 C 标准库中与时间（日期）处理、字符串和数值转换、随机数生成、动态内存管理、进程控制，以及搜索排序等功能相关的接口。其中，第一部分功能的接口由

time.h 提供，其余部分由 stdlib.h 提供。

C 语言中的时间可以被分为日历时间与处理器时间。通过名为 `time` 与 `clock` 的两个接口，我们可以分别获得与它们对应的两个值。进一步，借助 `localtime` 与 `strftime` 等接口，日历时间可以被转换为本地时间，并按照指定的形式进行格式化。而处理器时间由于在默认情况下以 `clock tick` 为单位，因此，需要将它除以宏常量 `CLOCKS_PER_SEC`，从而得到以秒为单位的值。

通过使用 `atof` 与 `strtol` 等接口，我们可以实现字符串到数字值的转换。其中，前一类以 “a” 开头的接口，仅能对字符串进行一次转换；而后一类以 “str” 开头的接口，则可在对字符串进行数值转换的基础上，同时检查转换结果是否发生溢出，并保存不能被转换部分的地址。

通过配合使用 `rand` 与 `srand` 接口，我们可以在 C 程序中生成伪随机数。在大多数标准库中，`rand` 函数会使用伪随机算法来实现。因此，为了使每次调用 `rand` 函数生成的随机数序列都不尽相同，在调用该接口前，可以配合使用 `srand` 与 `time` 函数，来为其设置随时间变化的不同随机数种子。

`malloc`、`calloc`，以及 `free` 等接口的实现，方便了我们在程序中动态操作堆内存。而通过调用 `exit`、`abort`、`quick_exit` 等接口，我们可以精确地控制程序在退出时的具体行为。除此之外，`abs`、`qsort`、`bsearch`、`mblen` 等接口的提供，也使得标准库在搜索排序、整数算数、宽字符（串）转换等方面，为 C 编程提供了一定帮助。

## 思考题

最后，让我们来讨论一个有意思的话题：既然计算机无法产生真正的随机数，那么有哪些方法可以用来产生真正的随机数呢？它们的原理是什么呢？欢迎在评论区告诉我你的发现。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

领资料

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

生成海报并分享

上一篇 15 | 标准库：信号与操作系统软中断有什么关系？

下一篇 17 | 标准库：断言、错误处理与对齐

## 更多课程推荐

# 操作系统实战 45 讲

## 从 0 到 1, 实现自己的操作系统

彭东

网名 LMOS

Intel 傲腾项目关键开发者



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

### 精选留言 (2)

💬 写留言



liu\_liu

2022-01-19

真随机数需结合物理现象来做到真正的随机，比如大气噪音，硬件噪音等。

这里有个网站 <https://www.random.org/>，可以生成真随机数。

作者回复: 这个网站很有意思诶！



👍 4

📁 领资料

ppm

2022-01-19

随机 可以点1000根火柴，图像唯一

