R5：辅助函数和类型

`<event2/util.h>` 中定义了很多函数，您可能会发现这些函数对使用 Libevent 实现可移植程序很有帮助。Libevent 内部也使用了这些类型和函数。

基本类型

evutil_socket_t

除 Windows 外的绝大多数地方，套接字都是一个整型变量，操作系统按照数值次序进行分配。然而，在使用 Windows 套接字 API 时，套接字的类型是 `SOCKET`，是一个类似于指针的操作系统句柄，并且接收的次序是未定义的。在 Windows 中，我们将 `evutil_socket_t` 定义为整型指针，能够在避免指针截断风险的同时处理 `socket()` 和 `accept()` 的输出。

定义

```
#ifdef WIN32
#define evutil_socket_t intptr_t
#else
#define evutil_socket_t int
#endif
```

该类型在 Libevent 2.0.1-alpha 中引入。

标准整数类型

您可能经常发现自己使用的 C 语言系统落后于 21 实际，因此没有实现 C99 标准的 `stdint.h` 头文件。出于这个原因，Libevent 定义了来自 `stdint.h` 的指定比特宽度的整数：

类型	宽度	符号	最大值	最小值
ev_uint64_t	64	No	EV_UINT64_MAX	0
ev_int64_t	64	Yes	EV_INT64_MAX	EV_INT64_MIN
ev_uint32_t	32	No	EV_UINT32_MAX	0
ev_int32_t	32	Yes	EV_INT32_MAX	EV_INT32_MIN
ev_uint16_t	16	No	EV_UINT16_MAX	0
ev_int16_t	16	Yes	EV_INT16_MAX	EV_INT16_MIN
ev_uint8_t	8	No	EV_UINT8_MAX	0
ev_int8_t	8	Yes	EV_INT8_MAx	EV_INT8_MIN

和 C99 标准一样，这些类型都有明确的比特位宽。

这些类型在 Libevent 1.4.0-beta 中引入，`MAX/MIN` 常量在 Libevent 2.0.4-alpha 中首次出现。

其他兼容类型

在定义了 `ssize_t`（有符号 `size_t`）的平台上，`ev_ssize_t` 被定义成 `ssize_t`，否则定义成一个合理的默认类型。`ev_ssize_t` 的最大值是 `EV_SSIZE_MAX`，最小值是 `EV_SSIZE_MIN`。（如果您的平台没有定义 `SIZE_MAX`，则 `size_t` 的最大值是 `EV_SIZE_MAX`。

`ev_off_t` 类型用来表示文件或内存块的偏移。在有 `off_t` 合理定义的平台上被定义成 `off_t`，在 Windows 上被定义成 `ev_int64_t`。

有些套接字 API 提供了长度类型 `socklen_t`，有些则没有。在有这个类型定义的平台中，`ev_socklen_t` 被定义成这个类型；在没有这个定义的平台中，则定义成合理的默认类型。

`ev_intptr_t` 类型是一个有符号整数，足够容纳指针类型且不会产生截断。`ev_uintptr_t` 类型是一个无符号整数，足够容纳指针类型且不会产生截断。

`ev_ssize_t` 类型由 Libevent 2.0.2-alpha 加入，`ev_socklen_t` 类型由 Libevent 2.0.3-alpha 加入，`ev_intptr_t` 和 `ev_uintptr` 类型，`EV_SSIZE_MAX/MIN` 宏由 Libevent 2.0.4-alpha 加入，`ev_off_t` 类型在 Libevent 2.0.9-rc 中首次出现。

定时器可移植函数

并不是所有的平台都定义了标准 `timeval` 操作函数，因此我们提供了自己的实现。

接口

```
#define evutil_timeradd(tvp, uvp, vvp) /* ... */
#define evutil_timersub(tvp, uvp, vvp) /* ... */
```

这些宏对前两个参数做加法或减法（分别地），并把结果存在第三个参数中。

接口

```
#define evutil_timerclear(tvp) /* ... */
#define evutil_timerisset(tvp) /* ... */
```

将 `timeval` 置 0。检查 `timeval` 是否设置了值，非 0 返回真，否则返回假。

接口

```
#define evutil_timercmp(tvp, uvp, cmp)
```

`evutil_timercmp` 宏比较两个 `timeval`，如果满足 `cmp` 关系则返回真。例如 `evutil_timercmp(t1, t2, <=)` 的意思是 "`t1` 是否小于等于 `t2`？"。注意不同于某些操作系统，Libevent 的 `timercmp` 支持所有的 C 关系运算符（即 <，>，==，!=，<= 和 >=）。

接口

```
int evutil_gettimeofday(struct timeval *tv, struct timezone *tz);
```

`evutil_gettimeofday` 函数将 `tv` 设置成当前时间。`tz` 参数没有使用。

示例

```
struct timeval tv1, tv2, tv3;

/* 设置 tv1 为 5.5s */
tv1.tv_sec = 5; tv1.tv_usec = 500*1000;

/* 设置 tv2 为当前时间 */
evutil_gettimeofday(&tv2, NULL);

/* 设置 tv3 为 5.5s 后的时间 */
evutil_timeradd(&tv1, &tv2, &tv3);

/* 3 次都是真 */
if (evutil_timercmp(&tv1, &tv1, ==))  /* == "If tv1 == tv1" */
   puts("5.5 sec == 5.5 sec");
if (evutil_timercmp(&tv3, &tv2, >=))  /* == "If tv3 >= tv2" */
   puts("The future is after the present.");
if (evutil_timercmp(&tv1, &tv2, <))   /* == "If tv1 < tv2" */
   puts("It is no longer the past.");
```

`evutil_gettimeofday()` 函数于 Libevent 2.0 引入，其余函数在 Libevent 1.4.0-beta 中引入。

> 注意：Libevent 1.4.4 之前在 `timercmp` 中使用 <= 或 >= 是不安全的。

套接字 API 兼容性

本小节因历史原因而存在，Windows 从来没有真正以良好的（以及优雅的）兼容性实现伯克利套接字（也称为 BSD 套接字） API。以下是一些您可以用来假装它有（伯克利套接字 API）的函数。

接口

```
int evutil_closesocket(evutil_socket_t s);

#define EVUTIL_CLOSESOCKET(s) evutil_closesocket(s)
```

该函数用来关闭套接字。在 Unix 系统中，它是 `close()` 的别名；在 Windows 中，它会调用 `closesocket()` 。（您无法在 Windows 上对套接字调用 `close()` ，也没有其他系统定义了 `closesocket()` 。）

`evutil_closesocket` 函数自 Libevent 2.0.5-alpha 引入。在此之前，需要调用 `EVUTIL_CLOSESOCKET` 宏。

接口

```
#define EVUTIL_SOCKET_ERROR()
#define EVUTIL_SET_SOCKET_ERROR(errcode)
#define evutil_socket_geterror(sock)
#define evutil_socket_error_to_string(errcode)
```

这些宏用来访问和处理套接字错误码。 `EVUTIL_SOCKET_ERROR()` 返回该线程最后一次处理套接字的全局错误码， `evutil_socket_geterror()` 对指定套接字做相同操作。（两者都是类 Unix 系统中的 `errno` 类型） `EVUTIL_SET_SOCKET_ERROR()` 修改当前套接字的错误码（类似于在 Unix 系统上设置 `errno` ）， `evutil_socket_error_to_string()` 返回一个代表给定套接字错误码的字符串（类似于 Unix 系统中的 `strerror()` ）。

（我们需要这些函数是因为 Windows 的套接字函数并不使用 `errno` 表示错误，而是使用 `WSAGetLastError()` 。）

注意 Windows 的套接字错误与您在 `errno` 中看到的标准 C 错误并不一致，请当心。

接口

```
int evutil_make_socket_nonblocking(evutil_socket_t sock);
```

甚至在套接字上执行的非阻塞 IO 调用也是不能移植到 Windows 的。`evutil_make_socket_nonblocking()` 函数接收一个套接字（来自 `socket()` 或 `accept()`）并将其转换成非阻塞套接字。（该函数在 Unix 中设置 `O_NONBLOCK`，在 Windows 中设置 `FIONBIO`。）

接口

```
int evutil_make_listen_socket_reuseable(evutil_socket_t sock);
```

该函数确保一个监听套接字使用的地址在关闭后可以立刻被其他套接字使用。（在 Unix 中设置 `SO_REUSEADDR`，在 Windows 中什么都不做。Windows 中 `SO_REUSEADDR` 有其他含义。）

接口

```
int evutil_make_socket_closeonexec(evutil_socket_t sock);
```

该函数告知操作系统如果调用了 `exec()` 则应当关闭这个套接字。函数在 Unix 系统中设置 `FD_CLOEXEC` 标志位，在 Windows 中什么都不做。

接口

```
int evutil_socketpair(int family, int type, int protocol,
        evutil_socket_t sv[2]);
```

这个函数行为与 Unix 中 `socketpair()` 类似：创建两个互相连接的套接字，可对其使用普通套接字 IO 调用。该函数将两个套接字分别存储在 `sv[0]` 和 `sv[1]` 中，成功返回 0，失败返回 -1。

在 Windows 上，该函数只支持 `AF_INET` 地址族，`SOCK_STREAM` 类型，以及协议 0。（`AF_INET` 指 IPv4 地址族，`SOCK_STREAM` 可靠的、基于连接的字节流传输类型，0 指系统自动选择协议。）注意该函数在一些 Windows 主机上可能会失败，因为防火墙阻止 127.0.0.1，以防止主机与自己通信。

这些函数自 Libevent 1.4.0-beta 引入，除了 `evutil_make_socket_closeonexec()`，该函数是 Libevent 2.0.4-alpha 引入的。

可移植字符串处理函数

接口

```
ev_int64_t evutil_strtoll(const char *s, char **endptr, int base);
```

该函数与 `strtol` 行为类似，只是用于 64 位整数。在某些平台上，仅支持十进制。

接口

```
int evutil_snprintf(char *buf, size_t buflen, const char *format, ...);
int evutil_vsnprintf(char *buf, size_t buflen, const char *format, va_list ap);
```

这些 `snprintf` 的替代函数的行为与标准 `snprintf` 和 `vsnprintf` 接口一致。它们返回在缓冲区足够长时应该写入缓冲区的字节数，不包括结尾的 `NULL`。（这个行为遵循 C99 `snprintf()` 标准，但是与 Windows 的 `_snprintf()` 相反，如果字符串无法存入缓冲区，`_snprintf()` 会返回负数。）

`evutil_strtoll()` 函数自 Libevent 1.4.2-rc 出现，其他函数在 1.4.5 版本中首次出现。

独立于语言环境的字符串处理函数

在实现基于 ASCII 的协议时，您希望根据 ASCII 的字符类型处理字符串，而不管当前的语言环境（此处语言并非指编程语言，而是自然语言）。Libevent 提供了一些函数来帮助解决这个问题。

接口

```
int evutil_ascii_strcasecmp(const char *str1, const char *str2);
int evutil_ascii_strncasecmp(const char *str1, const char *str2, size_t n);
```

这些函数的行为与 `strcasecmp()` 和 `strncasecmp()` 类似，不同的是它们只使用 ASCII 字符集进行比较，而不管当前的语言环境。`evutil_ascii_str[n]casecmp()` 函数在 Libevent 2.0.3-alpha 中首次出现。

IPv6 辅助及可移植函数

接口

```
const char *evutil_inet_ntop(int af, const void *src, char *dst, size_t len);
int evutil_inet_pton(int af, const char *src, void *dst);
```

这些函数与标准 `inet_ntop()` 和 `inet_pton()` 函数行为一致，根据 RFC3493 标准解析以及格式化 IPv4 与 IPv6 地址。要格式化 IPv4 地址，调用 `evutil_inet_ntop()` 并将 `af` 设置成 `AF_INET`，`src` 指向 `in_addr` 结构体，`dst` 指向长度为 `len` 的字符缓冲区。对于 IPv6 地址，`af` 和 `src` 分别是 `AF_INET6` 和 `in6_addr` 结构体。要解析 ~~IPv4~~（此处应当是原文疏忽）地址，调用 `evutil_inet_pton()` 并将 `af` 设置成 `AF_INET` 或 `AF_INET6`，要解析的字符串 `src`，以及指向 `in_addr` 或 `in_addr6` 结构体的指针 `dst`。

`evutil_inet_ntop()` 失败时返回 `NULL`，成功时返回 `dst`。`evutil_inet_pton()` 成功返回 0，失败返回 -1。

接口

```
int evutil_parse_sockaddr_port(const char *str, struct sockaddr *out,
    int *outlen);
```

该函数从 `str` 解析地址并将结果写进 `out`。`outlen` 必须指向一个 `out` 可用字节数的整数，并且会被修改为实际使用的字节数。该函数成功时返回 0，失败时返回 -1。该函数可以识别以下地址格式：

- [ipv6]:port（形如 "[ffff::]:80"）
- ipv6（形如 "ffff::"）
- [ipv6]（形如 "[ffff::]"）
- ipv4:port（形如 "1.2.3.4:80"）
- ipv4（形如 "1.2.3.4"）

如果给有给定端口，则 `out` 中的端口置为 0。

接口

```
int evutil_sockaddr_cmp(const struct sockaddr *sa1,
    const struct sockaddr *sa2, int include_port);
```

`evutil_sockaddr_cmp()` 函数用于比较两个地址，如果 `sa1` 在 `sa2` 前面返回负数，相等返回 0，否则返回正数。该函数可用于 `AF_INET` 和 `AF_INET6` 地址，对于其他地址，返回值是未定义的。该函数保证给出这些地址的次序，但是在不同版本的 Libevent 中，排序方法可能会改变。

如果 `include _port` 参数是 `false`，则两个只在端口上有区别的 `sockaddr` 被认为是相等的。否则（`include _port` 参数是 `true`），端口不同的 `sockaddr` 被认为是不相等的。

这些函数在 Libevent 2.0.1-alpha 中引入，`evutil_sockaddr_cmp()` 除外，该函数在 2.0.3-alpha 中引入。

结构体宏可移植函数

接口

```
#define evutil_offsetof(type, field) /* ... */
```

与标准 `offsetof` 宏类似，该宏返回从 `type` 类型开始处到 `field` 字段的字节数。

该宏于 Libevent 2.0.1-alpha 引入，但是在 Libevent 2.0.3-alpha 前的每个版本都有 bug。

安全随机数生成器

出于安全考虑，很多应用（包括 `evdns`）需要一个难以预测的随机数源。

接口

```
void evutil_secure_rng_get_bytes(void *buf, size_t n);
```

该函数使用随机数据填充一个 `n` 比特长的缓冲区。

如果您的平台提供 `arc4random()` 函数，Libevent 会使用这个函数。否则会使用 Libevent 自己实现的 `arc4random()`，种子来自于操作系统的熵池（entropy pool，Windows 为 `CryptGenRandom`，其他操作系统为 `/dev/urandom`）。

接口

```
int evutil_secure_rng_init(void);
void evutil_secure_rng_add_bytes(const char *dat, size_t datlen);
```

您无需手动初始化安全随机数生成器（random number generator，RNG），不过如果想确保生成器成功初始化，您可以调用 `evutil_secure_rng_init()`。该函数播种 RNG（如果还没有播种）并在成功时返回 0。如果返回 -1，意味着 Libevent 在操作系统上无法找到一个好的熵源，如果不自己初始化 RNG，就无法安全使用 RNG。

如果能正在运行的环境中，程序可能失去特权（例如，执行 `chroot()`），那么应该在执行之前调用 `evutil_secure_rng_init()`。

可以通过调用 `evutil_secure_rng_add_bytes` 向熵池中添加更多随机字节，通常来说不需要这么做。

这些函数是 Libevent 2.0.4-alpha 新引入的。