<https://blog.csdn.net/luotuo44/article/details/38780157>

**转载请注明出处：**[**http://blog.csdn.net/luotuo44/article/details/38780157**](http://blog.csdn.net/luotuo44/article/details/38780157)

        Libevent定义了一系列的可移植的兼容类型和函数。这使得在各个系统上都有一致的效果，Libevent一般都会在兼容通用类型和函数的前面加上ev或evutil前缀。

        在实现上，Libevent都是使用条件编译+宏定义的方式。使用这种方式，同一个宏名字，可以使得在不同的系统上， 编译时得到不同的值。这种方式在跨平台编程中，经常使用到。此外，对于Libevent的兼容类型，如果所在系统已经有对应功能的类型，那么Libevent将直接将ev\_XXX宏的值定义为该类型。如果所在系统没有对应的类型，那么就会选择一个比较合理的类型作为宏值。

# ****兼容类型：****

## 定长位宽类型：

        因为，C/C++中，整型int的位宽(多少bit)是没有限定的，而在有的时候却需要一个确定的长度。在C99标准中有一个stdint.h头文件定义了一些确定位宽的整型，如int64\_t、int32\_t。但Libevent考虑到可能有的环境并没有支持这个头文件，所以自己就定义了自己的一套确切位宽的整型。

        在util.h文件的一开始，就定义了一些通用的位宽确定的整数类型。如下:

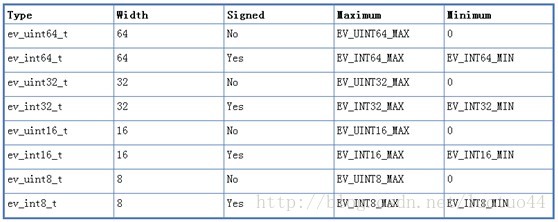
**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #ifdef \_EVENT\_HAVE\_UINT64\_T
2. #define ev\_uint64\_t uint64\_t
3. #define ev\_int64\_t int64\_t
4. #elif defined(WIN32)
5. #define ev\_uint64\_t unsigned \_\_int64
6. #define ev\_int64\_t signed \_\_int64
7. #elif \_EVENT\_SIZEOF\_LONG\_LONG == 8
8. #define ev\_uint64\_t unsigned long long
9. #define ev\_int64\_t long long
10. #elif \_EVENT\_SIZEOF\_LONG == 8
11. #define ev\_uint64\_t unsigned long
12. #define ev\_int64\_t long
13. #elif defined(\_EVENT\_IN\_DOXYGEN)
14. #define ev\_uint64\_t ...
15. #define ev\_int64\_t ...
16. #else
17. #error "No way to define ev\_uint64\_t"
18. #endif

        从代码中可以看到，它最先考虑当前环境是否已经定义了64位宽的整型，如果有的话，就直接使用。然后再考虑是否在Windows系统、是否定义了\_EVENT\_SIZEOF\_LONG\_LONG，并且值为8 ……

        正如《[event-config.h指明所在系统的环境](http://blog.csdn.net/luotuo44/article/details/38300965)》博文所说的，像\_EVENT\_HAVE\_UINT64\_T、\_EVENT\_SIZEOF\_LONG\_LONG这些宏定义都是在Libevent检测系统环境时定义的。

        Libevent定义了一系列位宽的整型，如下图：



        表来自<http://www.wangafu.net/~nickm/libevent-book/Ref5_evutil.html>

        其最值是直接计算出来的，如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #define EV\_UINT64\_MAX ((((ev\_uint64\_t)0xffffffffUL)<< 32) | 0xffffffffUL)
2. #define EV\_INT64\_MAX  ((((ev\_int64\_t) 0x7fffffffL) << 32) |0xffffffffL)
3. #define EV\_INT64\_MIN  ((-EV\_INT64\_MAX) - 1)
4. #define EV\_UINT32\_MAX((ev\_uint32\_t)0xffffffffUL)
5. #define EV\_INT32\_MAX  ((ev\_int32\_t) 0x7fffffffL)
6. #define EV\_INT32\_MIN  ((-EV\_INT32\_MAX) - 1)
7. #define EV\_UINT16\_MAX((ev\_uint16\_t)0xffffUL)
8. #define EV\_INT16\_MAX  ((ev\_int16\_t) 0x7fffL)
9. #define EV\_INT16\_MIN  ((-EV\_INT16\_MAX) - 1)
10. #define EV\_UINT8\_MAX  255
11. #define EV\_INT8\_MAX   127
12. #define EV\_INT8\_MIN   ((-EV\_INT8\_MAX) - 1)

        EV\_UINT64\_MAX是需要使用位操作才能得到的。因为对于UL（unsigned long）类型说，是可移植的最大值了。因为对于32位的OS来说，long类型位宽是32位的，64位的OS，long是64位的。对于0xffffffffUL这个只有32位的字面值来说保证了可移植性。接着把其强制转换成ev\_uint64\_t类型，此时就有了64位宽，无论是在32位的系统还是64位的系统。然后再利用位操作达到目的。

## 有符号类型size\_t：

        Libevent定义了ev\_ssize\_t作为有符号size\_t的兼容类型。因为在遵循POSIX标准的系统中，将有符号的size\_t定义为ssize\_t，而Windows系统则定义为SSIZE\_T。其的具体实现是，在util.h文件中如下定义：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #ifdef \_EVENT\_ssize\_t
2. #define ev\_ssize\_t \_EVENT\_ssize\_t
3. #else
4. #define ev\_ssize\_t ssize\_t
5. #endif

        然后在Windows系统的event-config.h文件中，则有下面的定义

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #define \_EVENT\_ssize\_t SSIZE\_T

        同样，Libevent也给ev\_size\_t和ev\_ssize\_t定义了范围：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #if \_EVENT\_SIZEOF\_SIZE\_T == 8
2. #define EV\_SIZE\_MAX EV\_UINT64\_MAX
3. #define EV\_SSIZE\_MAX EV\_INT64\_MAX
4. #elif \_EVENT\_SIZEOF\_SIZE\_T == 4
5. #define EV\_SIZE\_MAX EV\_UINT32\_MAX
6. #define EV\_SSIZE\_MAX EV\_INT32\_MAX
7. #elif defined(\_EVENT\_IN\_DOXYGEN)
8. #define EV\_SIZE\_MAX ...
9. #define EV\_SSIZE\_MAX ...
10. #else
11. #error "No way to defineSIZE\_MAX"
12. #endif
13. #define EV\_SSIZE\_MIN((-EV\_SSIZE\_MAX) - 1)

## 偏移类型：

        Libevent定义了ev\_off\_t作为兼容的偏移类型。其实现也很简单。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #ifdef WIN32
2. #define ev\_off\_t ev\_int64\_t
3. #else
4. #define ev\_off\_t off\_t
5. #endif

## socket类型：

        按照Libevent的说法，除了Windows系统外，其他OS的套接字类型大多数都是int类型的。而在Windows系统中，为SOCKET类型，实际为intptr\_t类型。所以Libevent的实现也很简单：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #ifdef WIN32
2. #define evutil\_socket\_t intptr\_t
3. #else
4. #define evutil\_socket\_t int
5. #endif

## socklen\_t类型：

        在Berkeley套接字中，有一些函数的参数类型是socklen\_t类型，你不能传一个int或者size\_t过去。但在Windows系统中，又没有这样的一个类型。比如bind函数。在使用Berkeley套接字的系统上，该函数的第三个参数为socklen\_t，而在Windows系统上，该参数的类型只是简单的int。为此，Libevent定义了一个兼容的ev\_socklen\_t类型。其实现为：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #ifdef WIN32
2. #define ev\_socklen\_t int
3. #elif defined(\_EVENT\_socklen\_t)
4. #define ev\_socklen\_t \_EVENT\_socklen\_t
5. #else
6. #define ev\_socklen\_t socklen\_t
7. #endif

        通用可以在event-config.h文件中找到\_EVENT\_socklen\_t的定义，要在没有定义socklen\_t系统的event-config.h文件中才能找到该定义。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. /\* Define to unsigned int if you donthave it \*/
2. #define \_EVENT\_socklen\_t unsigned int

## 指针类型：

        intptr\_t是一个很重要的类型，特别是在64位系统中。如果你要对两个指针进行运算，最好是先将这两个指针转换成intptr\_t类型，然后才进行运算。因为在一些64位系统中，int还是32位，而指针类型为64位，所以两个指针相减，其结果对于32位的int来说，可能会溢出。

        为了兼容，Libevent定义了兼容的intptr\_t类型。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #ifdef \_EVENT\_HAVE\_UINTPTR\_T
2. #define ev\_uintptr\_t uintptr\_t
3. #define ev\_intptr\_t intptr\_t
4. #elif \_EVENT\_SIZEOF\_VOID\_P <= 4
5. #define ev\_uintptr\_t ev\_uint32\_t
6. #define ev\_intptr\_t ev\_int32\_t
7. #elif \_EVENT\_SIZEOF\_VOID\_P <= 8
8. #define ev\_uintptr\_t ev\_uint64\_t
9. #define ev\_intptr\_t ev\_int64\_t
10. #elif defined(\_EVENT\_IN\_DOXYGEN)
11. #define ev\_uintptr\_t ...
12. #define ev\_intptr\_t ...
13. #else
14. #error "No way to defineev\_uintptr\_t"
15. #endif

        从代码中可以看到，如果系统本身有intptr\_t类型的话，那么Libevent直接使用之，如果没有，那么就选择一个完全能放得下指针的类型。

        在event-config.h中，\_EVENT\_SIZEOF\_VOID\_P被定义成sizeof(void\* )，即一个指针类型的字节数。一般来说，在32位系统中，为4字节； 在64位系统中，为8字节。在Windows版本的event-config.h文件中，定义如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. /\* The size of `void \*', as computed by sizeof. \*/
2. #ifdef \_WIN64
3. #define \_EVENT\_SIZEOF\_VOID\_P 8
4. #else
5. #define \_EVENT\_SIZEOF\_VOID\_P 4
6. #endif

        在我的Linux（32位），直接定义为：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. /\* The size of `void \*', as computed bysizeof. \*/
2. #define \_EVENT\_SIZEOF\_VOID\_P 4

        读者可以谷歌一下"intptr\_t"和" LP32 ILP32 LP64 LLP64 ILP64"。

# 兼容函数：

## 时间函数：

        在《[Libevent时间管理](http://blog.csdn.net/luotuo44/article/details/38661787)》一文中，列出了一些基本的时间操作函数。这里就不重复了。在Libevent还定义了一个evutil\_gettimeofday函数，那篇文章并没有展开讲。

        该函数作为一个兼容函数可以在各个平台上获取系统时间。在非Windows平台上，可以直接使用函数gettimeofday。Windows则通过\_ftime函数获取系统时间，然后转换。实现如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. //util.h文件
2. #ifdef \_EVENT\_HAVE\_GETTIMEOFDAY
3. #define evutil\_gettimeofday(tv, tz) gettimeofday((tv), (tz))
4. #else
5. **struct** timezone;
6. **int** evutil\_gettimeofday(**struct** timeval \*tv, **struct** timezone \*tz);
7. #endif
9. //evutil.c文件
10. #ifndef \_EVENT\_HAVE\_GETTIMEOFDAY
11. /\* No gettimeofday; this muse be windows. \*/
12. **int**
13. evutil\_gettimeofday(**struct** timeval \*tv, **struct** timezone \*tz)
14. {
15. **struct** **\_timeb** tb;
17. **if** (tv == NULL)
18. **return** -1;
20. \_ftime(&tb);
21. tv->tv\_sec = (**long**)tb.time;
22. tv->tv\_usec = ((**int**)tb.millitm) \* 1000;
23. **return** 0;
24. }
25. #endif

## socket API函数：

        由于遵循POSIX标准的OS有“一切皆文件”的思想，所以在处理socket 的时候比较简单。而在Windows平台，处理socket就没这么简单。而且Windows也没有很好地兼容Berkeley socket API。为了统一兼容，Libevent定义了一些通用的函数。

        Libevent定义了通用函数有下面这些：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. **int** evutil\_make\_socket\_nonblocking(evutil\_socket\_t sock);
2. **int** evutil\_make\_listen\_socket\_reuseable(evutil\_socket\_t sock);
3. **int** evutil\_make\_socket\_closeonexec(evutil\_socket\_t sock);
4. **int** evutil\_closesocket(evutil\_socket\_t sock);
5. **int** evutil\_socketpair(**int** d, **int** type, **int** protocol, evutil\_socket\_t sv[2]);
6. EVUTIL\_SOCKET\_ERROR();//宏定义
7. EVUTIL\_SET\_SOCKET\_ERROR(errcode);//宏定义

        上面的函数中，除了evutil\_socketpair其他的都没有什么好讲的。因为都是一些Linux和Windows系统编程的基础内容。

        在遵循POSIX的系统已经定义了socketpair，所以直接使用即可。但在Windows中并没有定义socketpair。Libevent的处理方法是：使用普通的socket，在函数内部创建一个服务器socket和客户端socket，并让客户端连接上服务器。其中，IP地址使用环路地址(ipv4中就是那个127.0.0.1，ipv6是::1)，端口号则由内核自动选定。实现起来还是有点麻烦的。因为具体的实现就是一个简单的C/S模式代码，这里就不贴代码了。

         Libevent还定义了另外三个socket相关的通用操作函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. **const** **char** \*evutil\_inet\_ntop(**int** af, **const** **void** \*src, **char** \*dst, **size\_t** len);
2. **int** evutil\_inet\_pton(**int** af, **const** **char** \*src, **void** \*dst);
3. **int** evutil\_parse\_sockaddr\_port(**const** **char** \*str, **struct** sockaddr \*out, **int** \*outlen);

        inet\_ntop主要的一个功能是，它可以用于ipv6。对于ipv4，有inet\_aton和inet\_ntoa。这两个函数在POSIX和Windows中都是有的，不需要Libevent做什么工作。但对于inet\_ntop和inet\_pton，Windows中并没有提供(POSIX提供了)。          Libevent也是可以用于ipv6的。所以Libevent就定义了两个通用的函数。在实现上，Libevent也是自己写代码将之转换。这里也不贴代码了。

        evutil\_parse\_sockaddr\_port函数是用来解析一个字符串的。字符串的格式是,IP:port。比如，8.8.8.8:53。这个函数的作用就是将字符串所表示的ip和端口进行解析，并存放到参数out所指向的结构体上。之前，我们需要手动将一个ip和端口赋值给sockaddr\_in 结构体上。现在有这个函数，这工作不用我们做了。第三个参数是一个 值-结果 参数。即调用函数时，它的值是第二个参数out指向空间的大小。函数返回后，它的值指明该函数写了多少字节在out上。具体的实现这里也不说了。

## 结构体偏移量：

        这个函数的功能主要是求结构体成员在结构体中的偏移量。定义如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. #ifdef offsetof
2. #define evutil\_offsetof(type, field) offsetof(type, field)
3. #else
4. #define evutil\_offsetof(type, field) ((off\_t)(&((type\*)0)->field))
5. #endif

        其中，type表示结构体名称，field表示成员名称。可以看到，Libevent还是优先使用所在系统本身提供的offsetof函数。Libevent自己实现的版本也是很巧妙的。它用(type\*)0来让编译器认为有个结构体，它的起始地址为0。这样，编译器给field所在的地址就是编译器给field安排的偏移量。

        这个求偏移量的功能是Libevent是很有用的。不过，Libevent不是直接使用这个宏evutil\_offsetof。而是使用宏EVUTIL\_UPCAST。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. //util-internal.h文件
2. #define EVUTIL\_UPCAST(ptr, type, field)                            \
3. ((type \*)(((**char**\*)(ptr))- evutil\_offsetof(type, field)))

        这个宏EVUTIL\_UPCAST的作用是通过成员变量的地址获取其所在的结构体变量地址。比如有下面的结构体

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. **struct** Parent
2. {
3. **struct** Children ch;
4. **struct** event ev;
5. };

        假如变量child是struct Children类型指针，并且它是struct Parent结构体的成员。而且知道了child的地址，现在想获取child所在结构体的struct Parent的地址。此时就可以用EVUTIL\_UPCAST宏了。如下使用就能转换了。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38780157) [copy](https://blog.csdn.net/luotuo44/article/details/38780157)

1. **struct** Parent \*par = EVUTIL\_UPCAST(child, **struct** Parent, ch);
2. //展开宏后，如下
3. **struct** Parent \*par = ((**struct** Parent \*)(((**char**\*)(child))  - evutil\_offsetof(**struct** Parent, ch)));

        其中，并不需要ch为struct Parent的第一个成员变量。

        EVUTIL\_UPCAST宏的工作原理也是挺简单的，成员变量的地址减去其本身相对于所在结构体的偏移量就是所在结构体的起始地址了，再将这个地址强制转换成即可。

参考：

<http://www.wangafu.net/~nickm/libevent-book/Ref5_evutil.html>

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/luotuo44/article/details/38780157