

The libcstl Library Design



The libcstl Library Design

for libcstl 2.1

Wangbo

2010-11-08

This file documents the libcstl library.

This is edition 2.1, last updated 2010-11-08, of *The libcstl Library Design* for libcstl 2.1.

Copyright (C) 2008, 2009, 2010, 2011 Wangbo <activesys.wb@gmail.com>

目录

第一章概述.....	5
第一节关于libcstl.....	5
第二章 libcstl库的基本结构.....	6
第一节 libcstl 的组成部分.....	6
第二节容器与迭代器.....	6
第三节内存管理.....	7
第四节算法和函数.....	8
第五节类型机制.....	9
第三章 libcstl 内存管理.....	11
第一节小内存管理机制.....	11
第二节内存管理代码结构.....	16
第三节内部接口.....	18
第四章 libcstl 类型机制.....	20
第一节类型机制.....	20
第二节类型机制的代码结构.....	21
第三节外部接口.....	22
第四节内部接口.....	23
第五节类型识别.....	26
第五章 libcstl 迭代器.....	33
第一节迭代器的机制.....	33
第二节迭代器的代码结构.....	33
第三节对外接口.....	34
第四节内部接口.....	40

第一章 概述

第一节 关于 **libcstl**

在使用 C 语言编程时没有像 C++ 的中 STL 那样常用而且易于使用的数据结构库，因此为了给 C 语言编程提供方便，以 SGI STL 为蓝本使用 C 语言编写一个为 C 编程提供方便的数据结构库。

STL 之所以方便是因为它提供的接口使用起来很方便，并且 STL 的容器中可以保存任何的数据类型，通过迭代器可以将通用的算法作用于任何容器之上，使用 C 语言编写的这个库要将 STL 这些便利之处都模仿出来，我们将这个库命名为 **libcstl**。

STL 是 C++ 语言的标准库，支持标准 C++ 的平台都支持 STL（虽然实现不同，但是接口是标准的），所以 **libcstl** 库也要具有可移植性，支持多平台。

第二章 libcstl 库的基本结构

第一节 libcstl 的组成部分

STL 中容器用来保存数据，算法用来操作容器中的数据，它们之间通过迭代器做到了相分离：算法可以通过迭代器来作用与适当的容器，容器的结构以及数据的细节都是隐藏与迭代器之下的，算法不需要知道。同时算法是可以配置的，通过适当的函数对象可以改变算法的行为。libcstl 也具有类似的结构：

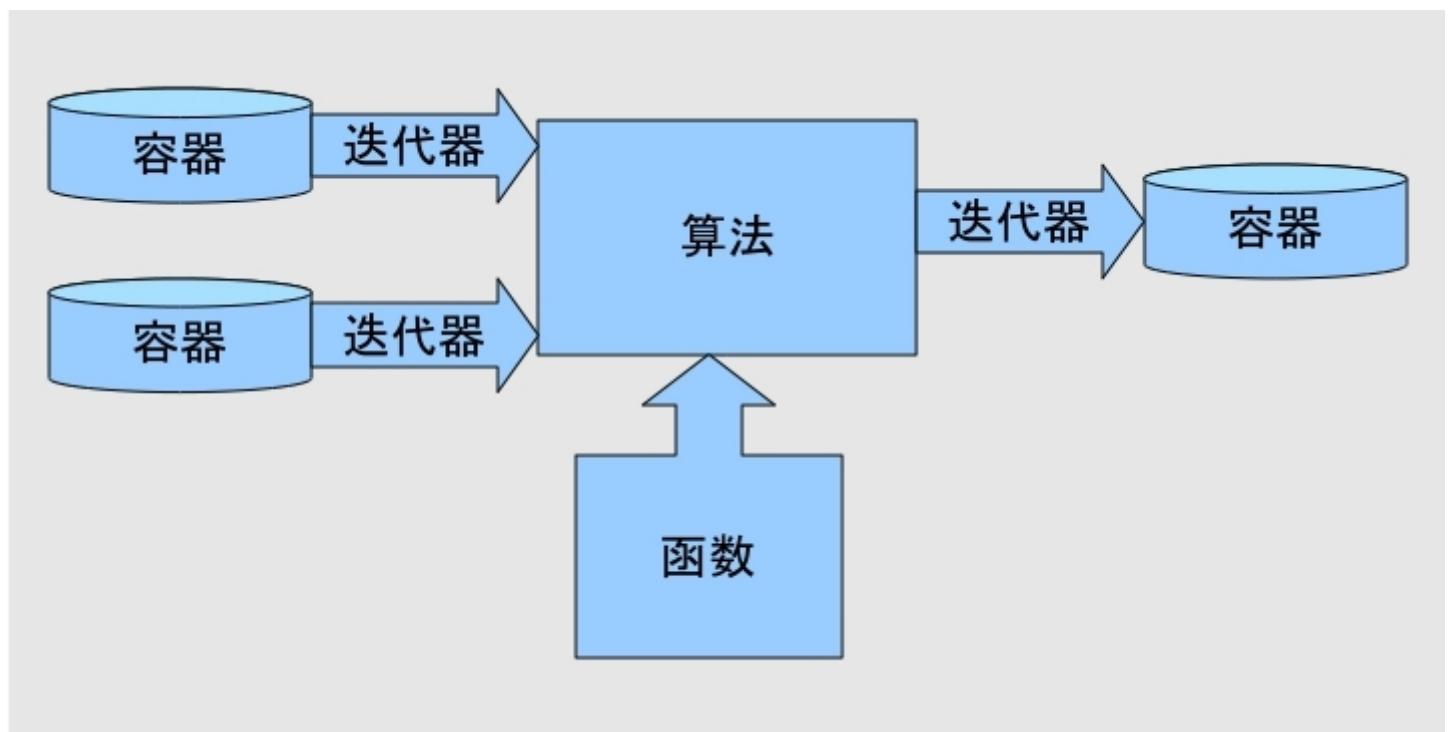


图 2.1 libcstl 组件之间的关系

第二节 容器与迭代器

当算法作用于某个容器的时候，算法只要知道容器的迭代器就可以对容器中的数据进行操作，库中的多种容器提供的是统一的迭代器接口，这样迭代器就将容器的细节封装在迭代器接口之下了：

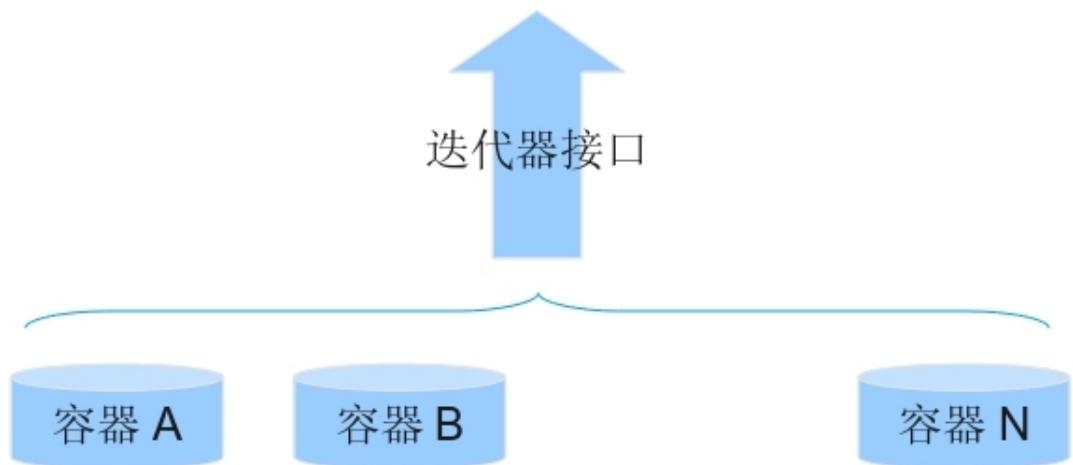


图 2.2 容器的迭代器接口

第三节 内存管理

容器中保存的数据都是保存在内存中的，因为容器中的数据是随时变化的所以容器免不了要经常分配和释放内存，libcstl 采用一个内存管理组建来管理内存，减少程序与系统之间频繁的内存交互。每一个容器都有自己的内存管理组建，各个容器之间的内存是互不干扰的：

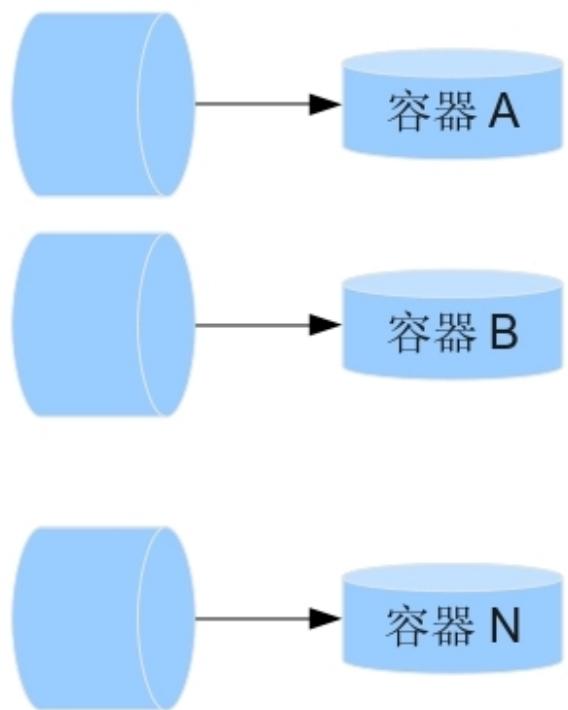


图 2.3 容器中的内存管理组件

第四节 算法和函数

算法是通过迭代器与容器中的数据进行交互的，算法还要靠函数来实际执行一些动作，没有函数的参与，算法不能够完成任务的。通过算法的接口可以改变算法使用的默认的函数，这样就改变了算法的行为：

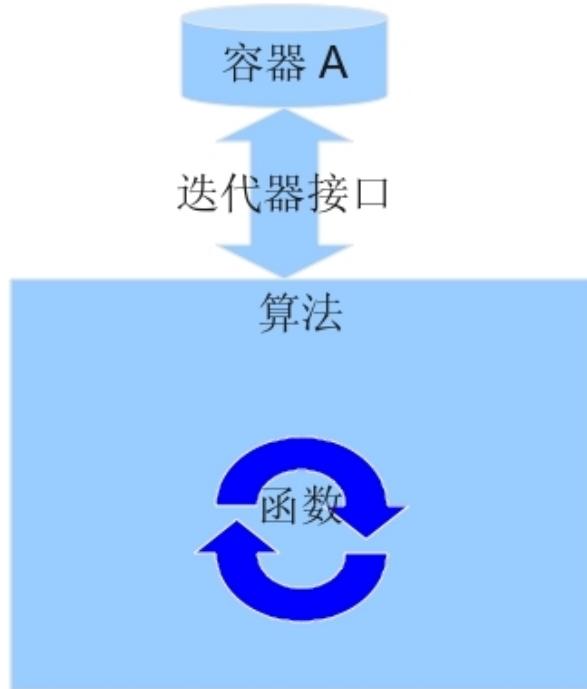


图 2.4 算法和函数

第五节 类型机制

容器中要保存任何类型的数据，这其中包括 C 语言内建的类型还有用户自定义的类型。对于各种类型容器都要知道它们相关的信息，最基本的，容器中经常要将数据排序，所以要知道关于某种类型数据的比较关系。此外可能有些数据需要在创建或者销毁的时候做一些额外的事情。这些类型信息都要保存在一个全局可访问的地方，并且这些类型不大可能经常变化：

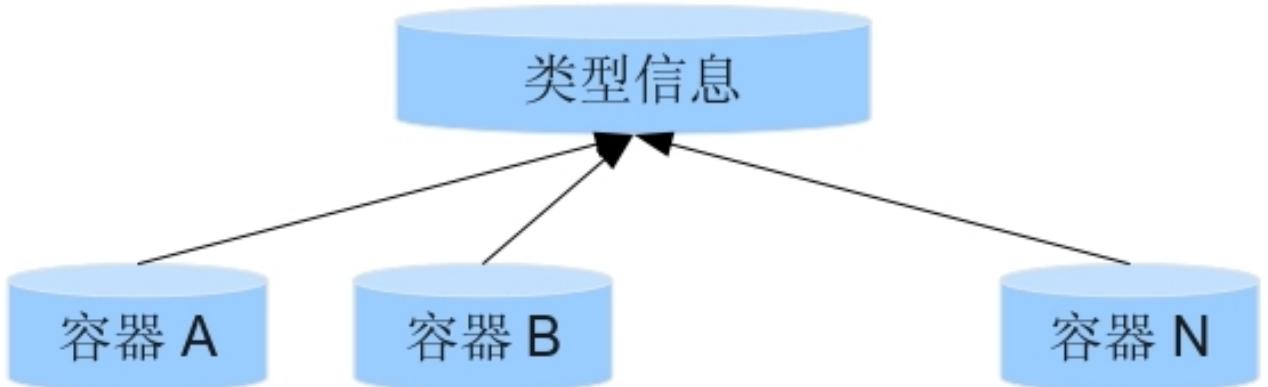


图 2.5 类型机制

第三章 libcstl 内存管理

libcstl 中的数据都是保存在容器中的，容器中用来保存数据的内存不是固定不变的，是动态分配的。所以，libcstl 要为容器提供一个统一的内存管理机制。这个机制不能够简单的使用 malloc() 和 free()，因为这样如果每次分配的内存较小，在多次分配后，容易造成内存碎片。所以在保存的单个数据足够小的情况下，要对于内存进行管理。

除了对于小内存进行管理外还要运行用户通过编译配置选择是否使用小内存管理，以及自动的内存管理切换。

第一节 小内存管理机制

我们认为当保存数据需要的内存小于 128 个字节的时候，分配内存的时候就容易造成内存碎片，所以在分配小于 128 个字节的内存的时候要启动小内存管理。同时在分配内存的时候都是按照字节边界分配的，也就是说例如你保存一个数据需要 30 个字节，在分配内存的时候程序自动的将这个数值提升到 8 的倍数，也就是实际回分配 32 个字节。这样小于等于 128 个字节并且的是 8 的倍数的内存块一共有 16 种：8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128。这样我们可以使用一个数组表示这 16 中内存块：

8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128
---	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----

图 3.1 小内存管理数组

这个数组中每一个元素都指向一个相应大小的内存块链表。我们采用下面的方式组成内存块的链表结构：

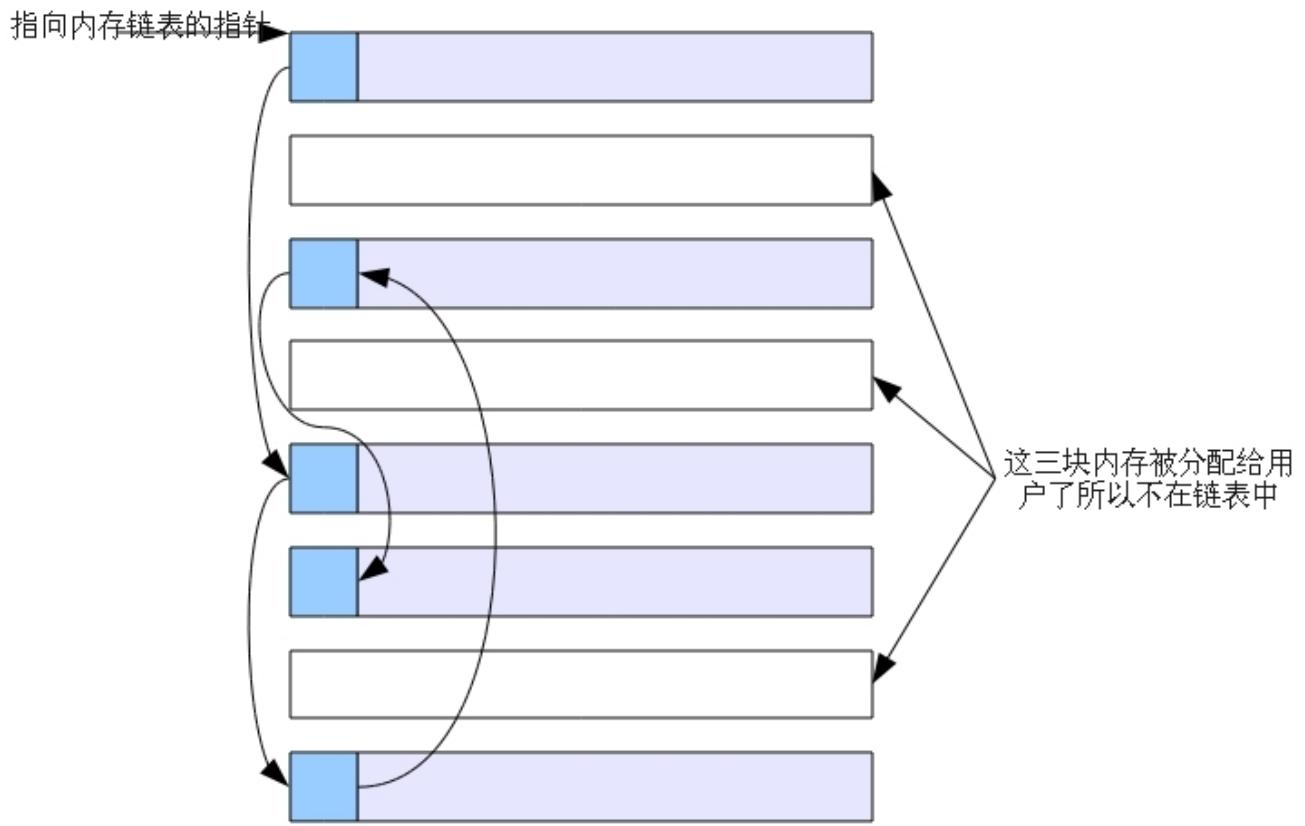


图 3.2 小内存块链表

这样的结构的一个好处是不会浪费存储空间，当内存块在链表中时，使用内存块的一部分作为指针来链接链表的各个项，当分配给用户之后，全部内存块都可以用来保存数据。在分配内存的时候，这样我们就可以在内存块链表中取出一个内存块分配给用户，然后将这个内存块从链表中去掉。在释放内存的时候，我们可以将用户使用后的内存块再次添加到相应的链表中等待下次再使用。

首先看一看分配内存时是怎么操作的：

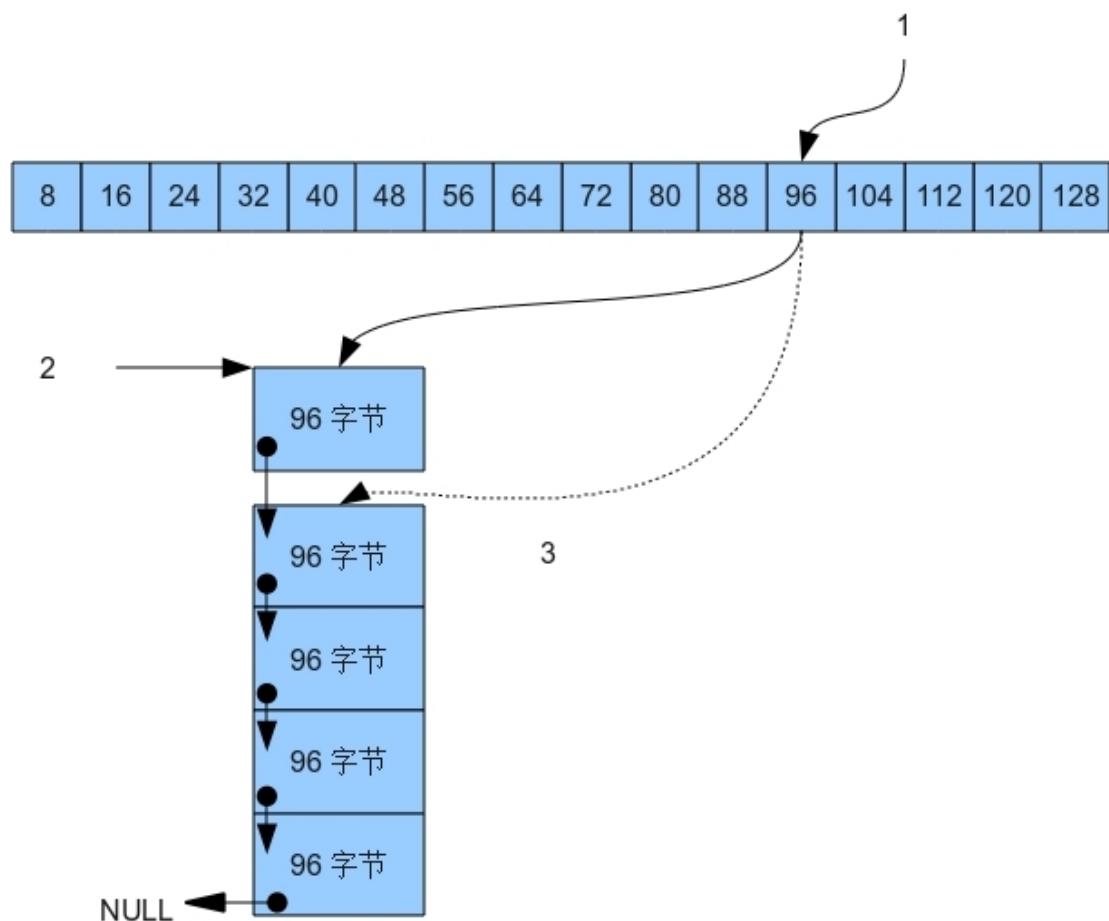


图 3.3 分配内存

在分配内存的时候，第一步通过待分配的内存块的大小在内存列表中找到相应的内存链表，第二步获得链表中的第一块内存，然后将内存列表中相应的内存链表指针指向下一块内存。

内存回收的时候是与内存分配相反的过程：

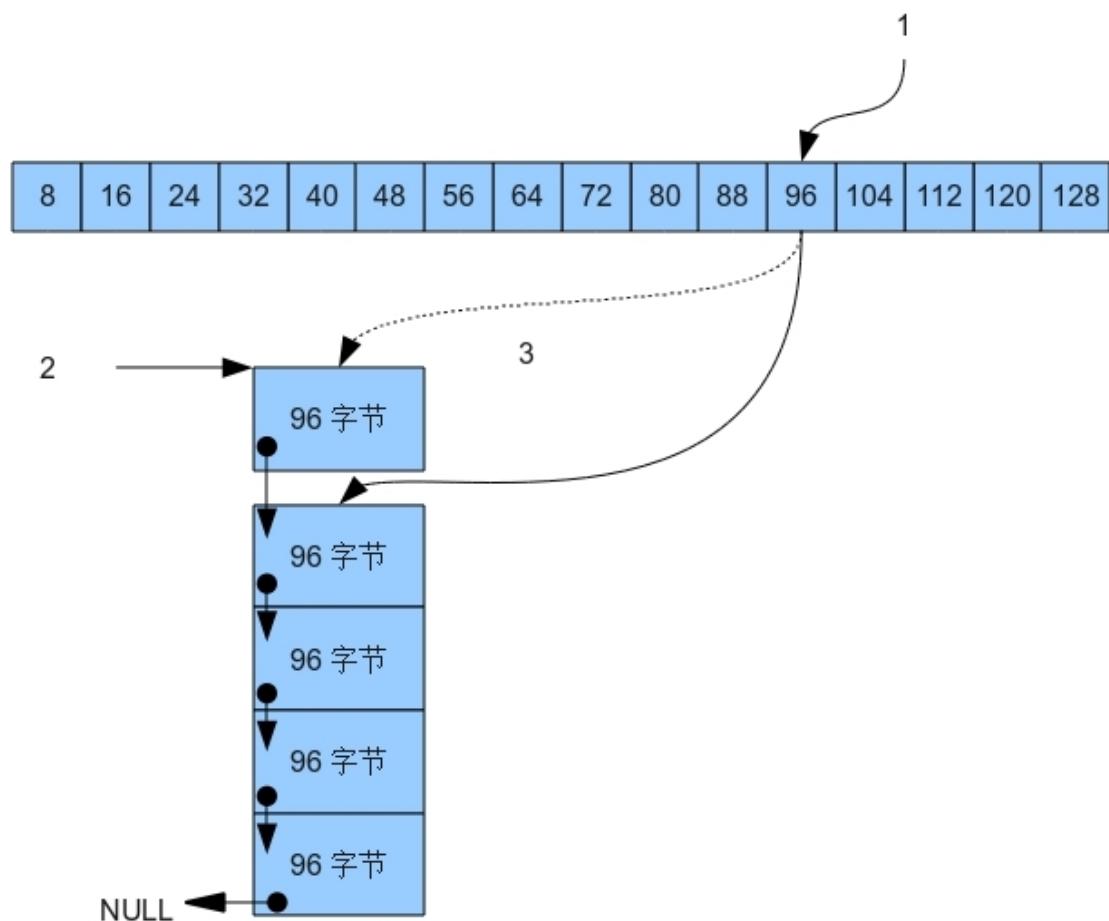


图 3.4 内存回收

内存回收是指将分配给用户的内存块重新链接到内存链表中，以后还可以使用。第一步同样是通过内存块的大小来计算相应的内存列表的位置从而得到内存链表的指针，第二步获得内存链表中的第一块内存的地址，将用户使用的内存块作为第一块内存块，将它格式化成内存块格式并指向当前内存链表的第一块内存。第三步将内存列表中的指针指向新的内存链表的第一块内存，也就是用户需要释放的内存。

上面的内存块全部来自于内存池，内存池是向系统申请的一大块内存，然后我们将这一大块内存格式化成很多小块内存并链接在内存列表的相应的位置上：

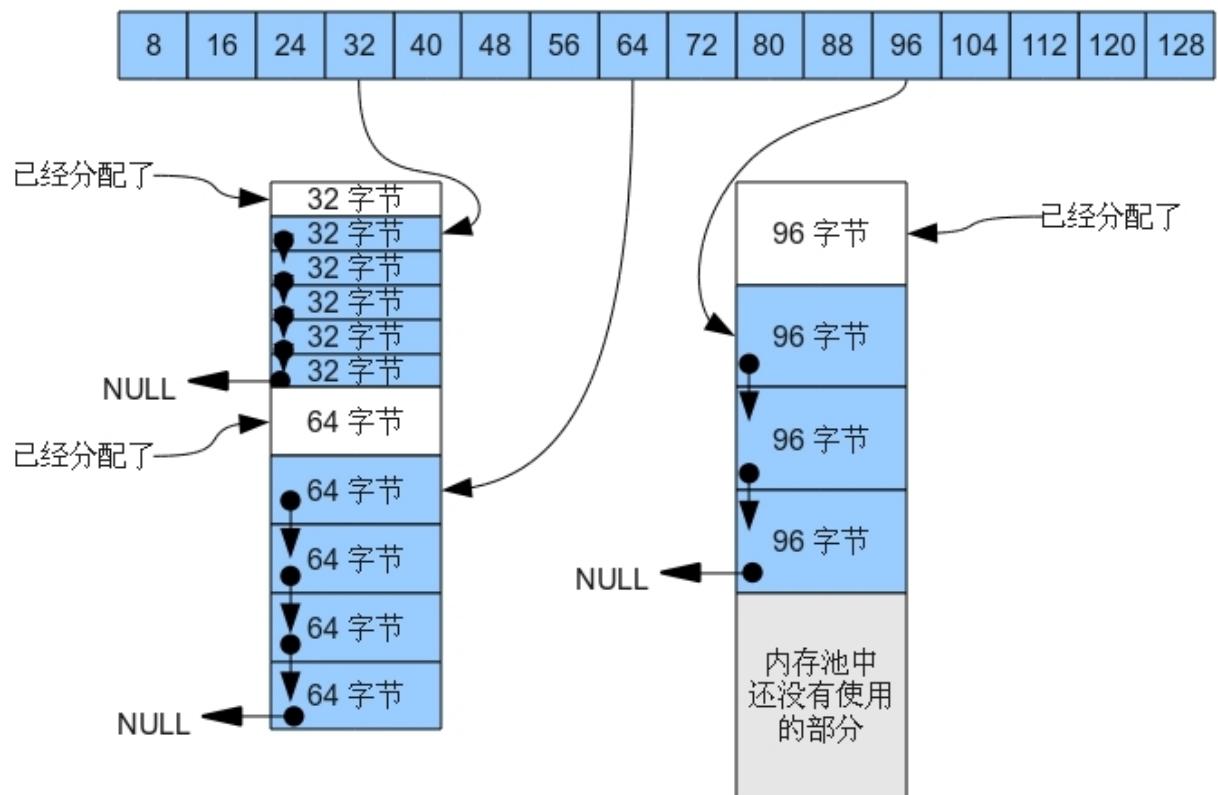


图 3.5 内存池的使用

库中使用的内存池不是一个，当一个内存池用完了之后还要向系统申请其他的内存池，多个内存池是使用内存池列表来管理的：

内存池管理列表，每一个都指向一个内存池

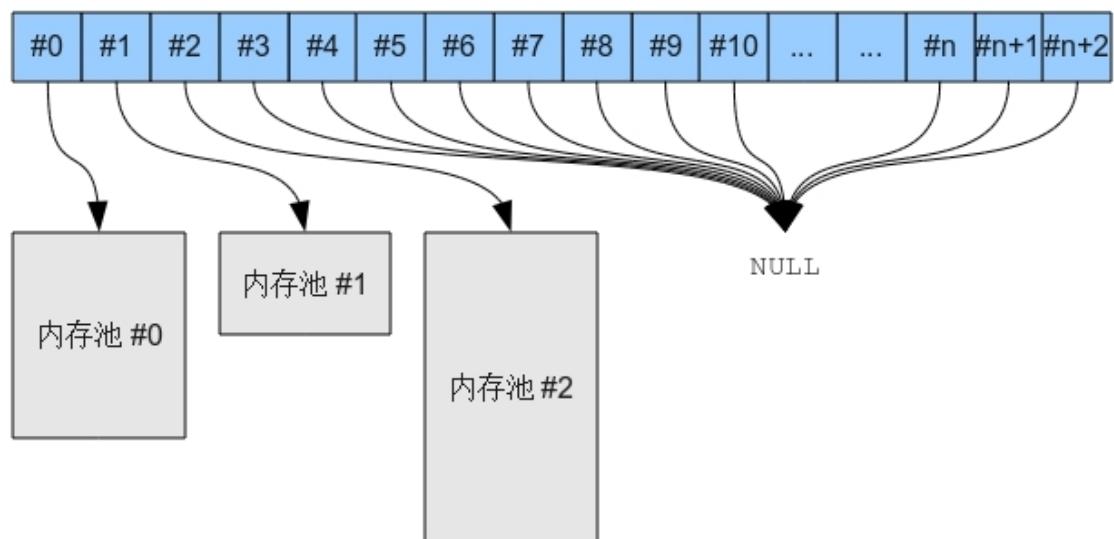


图 3.6 内存池管理

内存池管理列表是动态分配的列表，它的每一个元素都指向一个内存池，列表中的元素从头开始使用，没有使用的都指向 NULL。当内存池管理列表不够用的时候，重新为内存池管理列表分配空间来保存更多的内存池。

当用户申请的内存超过 128 个字节或者用户关闭了小内存管理机制，那么申请的内存就不属于内存管理机制的范围了，这样的内存申请使用传统的 malloc 和 free 来管理。

第二节 内存管理代码结构

libcstl 定义了一个 alloc_t 类型来表示内存管理，这个类型包含在每一个容器类型中，为容器提供内存。此外还有一个 memlink_t 类型表示小内存块的结构：

```
typedef union _tagmemlink
{
    union _tagmemlink* _pui_nextmem; /* 指向下一个内存块 */
    _byte_t             _pby_mem[1];   /* 内存块描述 */
} _memlink_t;
```

使用联合的特点是，当内存块在链表中时使用 _pui_nextmem 来表示链接，当分配给用户的时候，整个内存块都可以用来保存数据。这样就不会浪费空间来保存链表指针了。同时每一个内存块至少是 8 个字节，这在 32 位和 64 位系统上都是足以保存指针类型的。

下面是 alloc_t 类型，它表示内存管理结构：

```
typedef struct _tagalloc
{
    _memlink_t* _apt_memlink[_MEM_LINK_COUNT]; /* 内存块管理列表 */
    _byte_t**   _ppby_mempoolcontainer;        /* 内存池管理列表 */
    _byte_t*    _pby_mempool;                   /* 未使用的内存池的开始位置 */
    size_t      _t_mempoolszie;                /* 内存池的大小 */
    size_t      _t_mempoolindex;               /* 当前正在使用的内存池块的索引 */
    size_t      _t_mempoolcount;                /* 内存池块的个数 */
} _alloc_t;
```

这里定义了一个常量:_MEM_POOL_DEFAULT_COUNT 这个量是用来定义默认的内存池管理列表的大小的，2.0 中将这个值定义为了 4096 这显然过大。这样就造成了 libcstl 占用内存过大。2.1 要将这个值定义为更小更合理的值。内存管理结构如下：

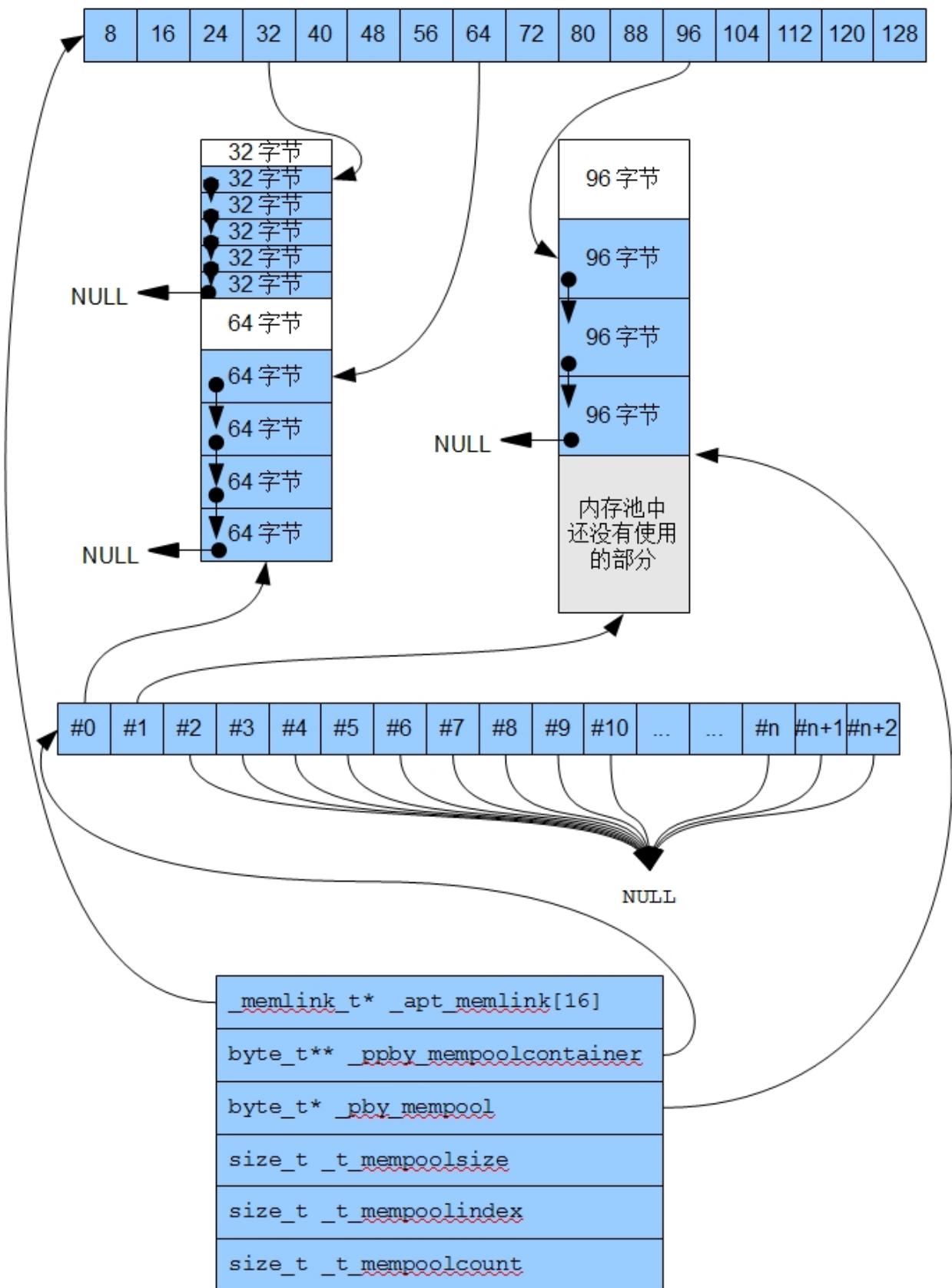


图 3.7 内存管理结构

第三节 内部接口

无论用户是否关闭了小内存管理，内存管理对外应该提供统一的接口，将处理的细节隐藏在接口的下面。

_alloc_init	初始化内存管理对象。
_alloc_destroy	销毁内存管理对象。
_alloc_allocate	分配内存。
_alloc_deallocate	回收内存。
_alloc_is_initied	测试_alloc_t 类型是否被初始化。

接口函数原型：

```
void _alloc_init(_alloc_t* pt_allocator);
```

描述：

初始化_alloc_t 类型的数据。

参数：

pt_allocator 指向_alloc_t 类型数据的指针。

返回值：

无

注意：

pt_allocator == NULL 函数的行为未定义。

```
void _alloc_destroy(_alloc_t* pt_allocator);
```

描述：

销毁_alloc_t 类型的数据。

参数：

pt_allocator 指向_alloc_t 类型数据的指针。

返回值：

无

注意：

pt_allocator == NULL 函数的行为未定义。

pt_allocator 指向的_alloc_t 类型的数据没有初始化那么函数的行为也是未定义的。

```
void* _alloc_allocate(_alloc_t* pt_allocator, size_t t_size, size_t t_count);
```

描述：

为用户分配指定大小的内存。

参数：

pt_allocator 指向_alloc_t 类型数据的指针。

t_size 数据的大小。

`t_count` 数据的个数。

返回值:

指向为用户分配的内存。

注意:

`pt_allocator == NULL` 函数的行为未定义。

`pt_allocator` 指向的`_alloc_t`类型的数据没有初始化那么函数的行为也是未定义的。

返回给用户的是指向`t_size * t_count`大小的内存块的指针，不会返回`NULL`。

但系统中没有足够的内存可以使用则回导致程序非正常退出并且标准出错中打印错误消息。

```
void _alloc_deallocate(  
    _alloc_t* pt_allocator, void* pv_allocmem, size_t t_size, size_t t_count);
```

描述:

回收用户指定大小的内存。

参数:

`pt_allocator` 指向`_alloc_t`类型数据的指针。

`pv_allocmem` 指向待回收的内存的指针。

`t_size` 数据的大小。

`t_count` 数据的个数。

返回值:

无。

注意:

`pt_allocator == NULL` 函数的行为未定义。

`pt_allocator` 指向的`_alloc_t`类型的数据没有初始化那么函数的行为也是未定义的。

`pv_allocmem` 是指向`t_size * t_count`大小的内存块的指针，并且是由`pt_allocator`分配的内存，否则函数的行为是未定义的。

```
bool_t _alloc_is_inited(const _alloc_t* cpt_allocator);
```

描述:

测试`_alloc_t`类型是否被初始化。

参数:

`cpt_allocator` 指向`_alloc_t`类型数据的指针。

返回值:

如果`_alloc_t`类型被初始化则返回`true`，否则返回`false`。

注意:

`cpt_allocator == NULL` 函数的行为未定义。

第四章 libcstl 类型机制

libcstl 在保存数据的时候需要类型信息，这些信息对于 libcstl 的各个容器来说是共用的，所以这些信息要保存在一个全局的位置，这样整个 libcstl 库都可以访问到。

第一节 类型机制

对于类型这样的信息主要是查询，添加操作只有在注册新的类型的时候才能够用到，至于删除则不会用到，因为一个类型一旦注册了就是全局可用的信息了，没有任何容器可以删除这个信息，没有提供删除操作也是防止将其他组件正在使用的信息删掉，并且一个类型信息不会占用太大的空间，而且注册的类型相对较少。由于查找的操作占的比例很大，所以保存类型信息的结构就要适应快速的查找，使用哈希表结构可以提供接近常数时间复杂度的查找操作。使用类型名为键值来构建哈希函数。

对于一个类型，首先关心的是名字，这唯一的标识了一个类型，同时它也决定了类型在类型哈希表中的存储位置。其次最关心的是类型占用的内存的大小，因为这决定向内存申请多少空间。此外还要有关于类型的一些操作，类型的初始化，类型的复制，比较以及销毁，这些操作要在容器的操作中用到。

对于类型还有一个特殊的环节，例如 C 语言的类型 int 和 signed int 是同样的类型，此外用户可以使用 `typedef` 来为自定义类型起别名如 `typedef struct _tagfoo {...} foo_t;` 这样 `struct _tagfoo` 和 `foo_t` 是相同的类型。我们要同时保存这两个类型的名字但是又要将它们识别为同一个类型。对于这样的情况我们可以使用两部来解决这个问题，第一步先注册一个名字，第二步使用另一个名字将先前注册的类型复制。这两个步骤不应该对于类型的名字有什么要求。

使用两种结构来表示类型的信息，一种结构是单纯的保存类型的信息，另一种结构是表示类型在类型注册表中的位置。对于同一个类型的两个名字，只有一个类型信息，但是在注册表中有两个位置，这两个位置节点都指向同一个信息节点。类型信息结构：

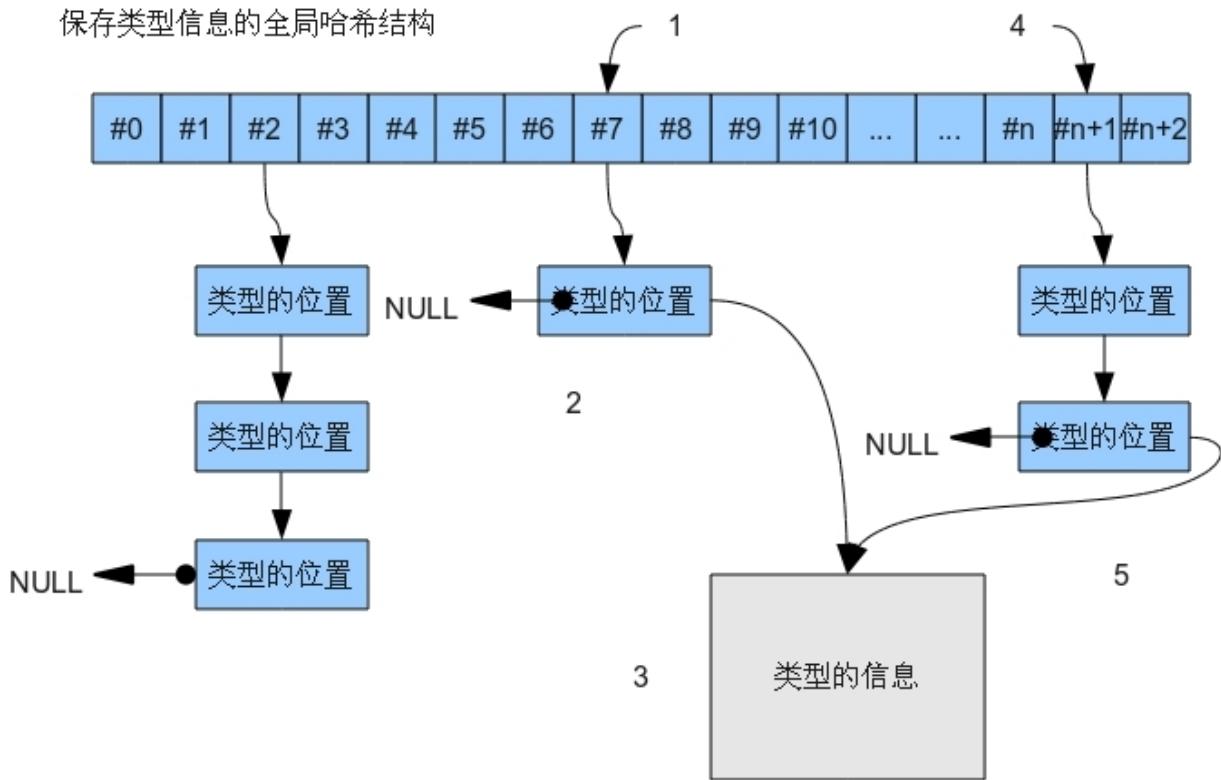


图 4.1 类型的注册以及复制

在注册一个类型的时候，首先根据类型的名字通过哈希函数计算得到这个类型在哈希表中的位置，然后查看是否这个类型的名字已经注册了，如果没有就生成一个类型位置的节点链接到哈希表相应的位置，然后根据用户提供的类型的名字，大小，操作函数等生成类型的信息节点，并将位置节点链接到这个信息节点上。

在复制一个类型的名字的时候，用户要提供一个已经注册的名字以及要复制的类型的名字，使用要复制的类型的名字通过哈希函数计算得到这个名字在哈希表中的位置，然后查看是否有同样的名字已经存在了，如果没有就生成一个类型位置节点并链接到哈希表的制定位置，然后根据已经注册的类型名字找到类型信息节点，将复制的类型位置节点指向类型信息节点。

对于用户来说，要提供一种机制让用户能够保存一种新的类型，以及复制现有的类型信息。对于 libcstl 内部要能够获得已经注册的类型信息。这样在用户注册了一个类型后 libcstl 应该能够自动的识别这个类型并得到相关的信息，同样，在复制了一个类型的信息之后，libcstl 能够自动的将多种相关的信息识别为同一种类型。

第二节 类型机制的代码结构

libcstl 定义结构 _type_t 来保存类型信息：

```

typedef struct _tagtype
{
    size_t          _t_typesize;
                                /* 类型的大小 */
  
```

```

char _sz_typename[_TYPE_NAME_SIZE+1]; /* 类型的名字 */
binary_function_t _t_typecopy; /* 类型的拷贝函数 */
binary_function_t _t_typeless; /* 类型的比较函数 */
unary_function_t _t_typeinit; /* 类型的初始化函数 */
unary_function_t _t_typedestroy; /* 类型的销毁函数 */

}_type_t;

```

使用_type_t 来表示类型的位置信息:

```

typedef struct _tagtypenode
{
    char _sz_typename[_TYPE_NAME_SIZE+1]; /* 类型的名字 */
    struct _tagtypenode* _pt_next; /* 下一个位置节点 */
    _type_t* _pt_type; /* 类型的注册信息 */
}_typenode_t;

```

使用_typeregister_t 表示类型注册表:

```

typedef struct _tagtyperegister
{
    bool_t _t_isinit; /* 初始化标记 */
    _typenode_t* _apt_bucket[_TYPE_REGISTER_BUCKET_COUNT]; /* 哈希表结构 */
    _alloc_t _t_allocator;
}_typeregister_t;

```

使用_typeinfo_t 表示在 libcst 内部使用信息:

```

typedef struct _tagtypeinfo
{
    char _sz_typename[_TYPE_NAME_SIZE+1]; /* 类型的名字 */
    _type_t* _pt_type; /* 指向类型注册表中的类型信息 */
    _typestyle_t _t_style;
}_typeinfo_t;

```

第三节 外部接口

类型机制提供了两个对外的接口，这个两个接口都是为用户提供的。

type_register	类型注册。
type_duplicate	类型复制。

接口函数原型:

```

bool_t type_register(
    typename,
    unary_function_t t_typeinit,
    binary_function_t t_typecopy,

```

```
binary_function_t t_typeless,
unary_function_t t_typedestroy);
```

描述:

注册名字为 typename 的数据类型。

参数:

typename 注册的数据类型的描述。
t_typeinit 数据类型的初始化函数。
t_typecopy 数据类型的拷贝函数。
t_typeless 数据类型的比较函数。
t_typedestroy 数据类型的销毁函数。

返回值:

注册成功返回 true, 否则返回 false。

注意:

typename 是被注册的类型的名字, 这个函数主要被用来注册用户自定义的类型。如果注册其他类型如 vector_t<int>这样的写法是未定义的。如果 typename 不符合类型描述的语法或者注册的类型已经被注册了或者 typename 超过 255 个字符则注册失败。使用 NULL 来表示为 typename 类型使用默认的操作函数。一个类型注册之后它的相关信息是不可以修改的。

```
bool_t type_duplicate( typename1, typename2 );
```

描述:

复制类型信息。

参数:

typename1 类型信息 1。
typename2 类型信息 2。

返回值:

复制成功返回 true, 否则返回 false。

注意:

这个函数主要被用来复制用户自定义的类型。如果使用其他类型如 vector_t<int>这样的写法是未定义的。typename1 和 typename2 其中必须有一个是已经注册的类型, 另一个是没有被注册的类型, 否则这个函数将失败。如果 typename 不符合类型描述的语法或者 typename 超过 255 个字符则注册失败。一个类型复制之后它两个类型名字被认为是相同的类型, 它们使用同样的类型信息, 在相同类型容器中分别使用这两个类型的名字被认为是容器中的数据也是相同的, 可以互操作的。可以多次使用这个函数来复制类型信息, 这样同一个类型就可以有很多别名了。

第四节 内部接口

同时还要对于 libcstl 内部实现提供接口以便其他部分使用类型信息。

_type_get_type	根据类型名字获得类型信息。
_type_get_type_pair	根据一对类型的名字获得两个类型信息。
_type_is_same	测试两个类型是否相同。

<code>_type_get_varg_value</code>	获得可变参数中的类型的值。
<code>_type_get_elem_typename</code>	获得容器类型中数据的类型的名字。
<code>_type_debug</code>	调试类型注册信息。

```
void _type_get_type(_typeinfo_t* pt_typeinfo, const char* s_typename);
```

描述:

根据类型的名字来获取获得类型信息并填充 pt_typeinfo 结构。

参数:

`pt_typeinfo` 类型信息。

`s_typename` 类型名字。

返回值:

无。

注意:

`pt_typeinfo` 或者 `s_typename` 为 NULL 则函数的行为是未定义的。如果 `s_typename` 不是已经注册的类型, `pt_typeinfo->_t_style == _TYPE_INVALID`, `pt_typeinfo->_pt_type == NULL`。

```
void _type_get_type_pair(
    _typeinfo_t* pt_typeinfofirst,
    _typeinfo_t* pt_typeinfosecond,
    const char* s_typename);
```

描述:

根据类型的名字来获取获得两个类型信息并填充 `pt_typeinfofirst` 和 `pt_typeinfosecond` 结构。

参数:

`pt_typeinfofirst` 第一个类型信息。

`pt_typeinfosecond` 第二个类型信息。

`s_typename` 类型名字。

返回值:

无。

注意:

这个函数是为 pair_t 类型提供的, pair_t 的函数名字字符串的形式是 “ typename1, typename2”这样的形式, typename1 和 typename2 都是类型的名字。pt_typeinfofirst 或者 pt_typeinfosecond 或者 s_typename 为 NULL, 这个函数的行为是未定义的。如果 s_typename 不是 “typename1,typename2”这样的形式或者 typename1 或者 typename2 不是注册类型那么 `pt_typeinfofirst->_t_style == _TYPE_INVALID`, `pt_typeinfofirst->_pt_type == NULL`, `pt_typeinfosecond->_t_style == _TYPE_INVALID`, `pt_typeinfosecond->_pt_type == NULL`。

```
bool_t _type_is_same(const char* s_typename1, const char* s_typename2);
```

描述:

测试两个已经注册的名字是否是同种类型。

参数:

`s_typename1` 第一个类型的名字。

`s_typename2` 第二个类型的名字。

返回值:

同种类型返回 true, 否则返回 false。

注意:

s_typename1 或者 s_typename2 为 NULL 则函数的行为是未定义的。如果 s_typename1 或者 s_typename2 其中有非注册类型那么返回 false。

```
void _type_get_varg_value(  
    _typeinfo_t* pt_typeinfo, va_list val_elemlist, void* pv_output);
```

描述:

根据类型信息从可变参数列表中取出数据并填充到 pv_output 中。

参数:

pt_typeinfo 类型信息。

val_elemlist 可变参数列表。

pv_output 获得数据的缓冲区。

返回值:

无。

注意:

pt_typeinfo 或者 pv_output 为 NULL, 则函数是未定义的。如果 pt_typeinfo->_t_style == _TYPE_INVALID 或者 pt_typeinfo->_pt_type == NULL 则函数的行为是未定义的。如果 pv_output 指向的缓冲区要满足制定类型的大小。

```
void _type_get_elem_typename(const char* s_typename, char* s_elemtypename);
```

描述:

根据类型名字获取类型中保存的数据的类型名字。

参数:

s_typename 类型名字。

s_elemtypename 数据的类型名字。

返回值:

无。

注意:

这个函数是从如 vector_t<map_t<int, long>>这样的容器类型名字中提取出容器中保存的数据的类型名字也就是 map_t<int, long>。如果 s_typename 不是容器类型的名字那么函数的行为是未定义的。如果 s_typename 和 s_elemtypename 是 NULL 那么函数的行为是未定义的。

```
void _type_debug(void);
```

描述:

调试类型注册表信息, 以树状的形式打印出类型注册表。

参数:

无。

返回值:

无。

注意:

这个函数是为开发者提供的调试函数。在类型注册表初始化之前调用这个函数行为是未定义的。

第五节 类型识别

当类型信息都保存在类型注册表中了，我们是通过类型的名字来获取类型信息的。但是识别合法的类型名字不是一件简单的事情，例如 C 内建类型有 int, unsigned long int 还有 char* 等等这样的都是合法的类型名字，用户自定义的类型名字可能是 struct abc 或者 enum def 等等，libcstl 容器的名字就更复杂了 vector_t<int>, map_t<int, char*>甚至是 hash_map_t<list_t<double>, pair_t<vector_t<unsigned long>, stack_t<signed short int>>>这样的名字都应该是合法的。那么怎么样来识别这些名字呢？

为这些名字定义一套语法，通过语法和此法解析来获得类型的名字。下面是词法和语法定义：

词法描述：

使用正则表达式的方式描述词法：

```
letter    : [a~zA~Z]
digit     : [0~9]
identifier : (letter | _)(letter | digit | \*)*
sign      : <|,|>
space     : ''|`t'|`v'|`f'|`r'|`n'
```

letter 是小写字符和大写字符的合计， digit 是字符 0 到 9， identifier 是以字符和下划线开头后面跟着数字或者下划线或者星号， sign 是大于号小于号和逗号， space 是空格 tab 回车换行等。

下面列出的都是描述中使用的保留字，它们与普通的 identifier 不同，我们使用蓝色表示：

```
char int short long float double signed unsigned char* bool_t struct enum union vector_t list_t
slist_t deque_t stack_t queue_t priority_queue_t set_t map_t multiset_t multimap_t hash_set_t hash_map_t
hash_multiset_t hash_multimap_t pair_t string_t iterator_t vector_iterator_t list_iterator_t slist_iterator_t
deque_iterator_t set_iterator_t map_iterator_t multiset_iterator_t multimap_iterator_t hash_set_iterator_t
hash_map_iterator_t hash_multiset_iterator_t hash_multimap_iterator_t string_iterator_t input_iterator_t
output_iterator_t forward_iterator_t bidirectional_iterator_t random_access_iterator_t
```

语法描述：

下面使用 BNF 描述整个语法，其中粗体的大写字符表示产生式如 **TYPE_DESCRIPTOR**，使用小写字符表示终结符如 identifier，使用蓝色的小写字符表示保留字如 **vector_t**，使用=> 表示推出符号，使用| 表示可选产生式，符号就使用本身表示如 < 。

TYPE_DESCRIPTOR => **C_BUILTIN** | **USER_DEFINE** | **CSTL_BUILTIN**

C_BUILTIN => **char** | **signed char** | **unsigned char** | **short** | **signed short** | **short int** | **signed short int** | **unsigned short** |
unsigned short int | **int** | **signed** | **signed int** | **unsigned** | **unsigned int** | **long** | **signed long** | **long int** |
signed long int | **unsigned long** | **unsigned long int** | **float** | **double** | **long double** | **char*** | **bool_t**

USER_DEFINE => **USER_DEFINE_TYPE** identifier | identifier

USER_DEFINE_TYPE => **struct** | **enum** | **union**

CSTL_BUILTIN => **SEQUENCE** | **RELATION** | **string_t** | **ITERATOR**

ITERATOR => **iterator_t** | **vector_iterator_t** | **list_iterator_t** | **slist_iterator_t** | **deque_iterator_t** | **set_iterator_t** | **map_iterator_t** |
multiset_iterator_t | **multimap_iterator_t** | **hash_set_iterator_t** | **hash_map_iterator_t** | **hash_multiset_iterator_t** |
hash_multimap_iterator_t | **string_iterator_t** | **input_iterator_t** | **output_iterator_t** | **forward_iterator_t** |
bidirectional_iterator_t | **random_access_iterator_t**

SEQUENCE => SEQUENCE_NAME < TYPE_DESCRIPTOR >

SEQUENCE_NAME => vector_t | list_t | slist_t | deque_t | queue_t | stack_t | priority_queue_t | set_t | multiset_t | hash_set_t | hash_multiset_t

RELATION => RELATION_NAME < TYPE_DESCRIPTOR , TYPE_DESCRIPTOR >

RELATION_NAME => map_t | multimap_t | hash_map_t | hash_multimap_t | pair_t

首先要对此法进行处理，将此法转换成有限的状态机：

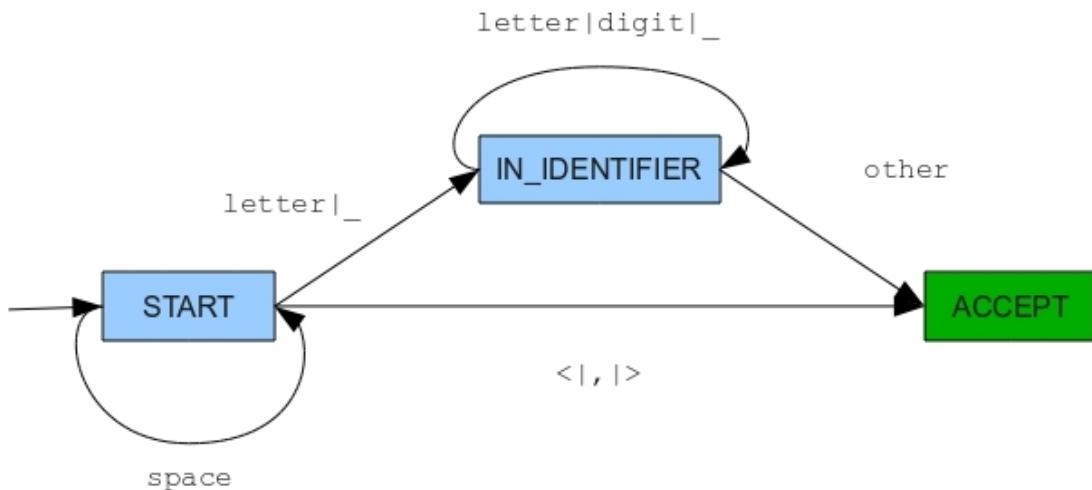


图 4.1 类型词法状态机

将 BNF 整理，并且求出三种集合：

TYPE_DESCRIPTOR -> C_BUILTIN | USER_DEFINE | CSTL_BUILTIN

C_BUILTIN -> char | signed char |

 unsigned char |

 short | signed short | short int | signed short int

 unsigned short | unsigned short int |

 int | signed | signed int |

 unsigned | unsigned int |

 long | signed long | long int | signed long int |

 unsigned long | unsigned long int |

 float |

 double |

 long double |

 char* |

 bool_t

USER_DEFINE -> USER_DEFINE_TYPE identifier | identifier

USER_DEFINE_TYPE -> struct | enum | union

CSTL_BUILTIN -> SEQUENCE | RELATION | string_t | ITERATOR

ITERATOR -> iterator_t | vector_iterator_t | list_iterator_t | slist_iterator_t | deque_iterator_t |

 set_iterator_t | map_iterator_t | multiset_iterator_t | multimap_iterator_t |

 hash_set_iterator_t | hash_map_iterator_t | hash_multiset_iterator_t | hash_multimap_iterator_t |

```

string_iterator_t |
    input_iterator_t | output_iterator_t | forward_iterator_t | bidirectional_iterator_t | random_access_iterator_t
SEQUENCE -> SEQUENCE_NAME < TYPE_DESCRIPTOR >
SEQUENCE_NAME -> vector_t | list_t | slist_t | deque_t | queue_t | stack_t | priority_queue_t |
    set_t | multiset_t | hash_set_t | hash_multiset_t
RELATION -> RELATION_NAME < TYPE_DESCRIPTOR , TYPE_DESCRIPTOR >
RELATION_NAME -> map_t | multimap_t | hash_map_t | hash_multimap_t | pair_t
=====
```

C_BUILTIN -> SIMPLE_BUILTIN | SIGNED_BUILTIN | UNSIGNED_BUILTIN

SIMPLE_BUILTIN -> char | short | short int | int | long | long int |

float | double | long double | char*

SIGNED_BUILTIN -> signed char |

signed short | signed short int |

signed | signed int |

signed long | signed long int |

UNSIGNED_BUILTIN -> unsigned char |

unsigned short | unsigned short int |

unsigned | unsigned int |

unsigned long | unaligned long int

TYPE_DESCRIPTOR -> C_BUILTIN | USER_DEFINE | CSTL_BUILTIN

C_BUILTIN -> SIMPLE_BUILTIN | SIGNED_BUILTIN | UNSIGNED_BUILTIN

SIMPLE_BUILTIN -> char | short COMMON_SUFFIX | int | long SIMPLE_LONG_SUFFIX | float | double | char* | bool_t

SIGNED_BUILTIN -> signed COMPLEX_SUFFIX

UNSIGNED_BUILTIN -> unsigned COMPLEX_SUFFIX

COMPLEX_SUFFIX -> {+''}char | {+''}short COMMON_SUFFIX | {+''}int | {+''}long SIMPLE_LONG_SUFFIX | \$

COMMON_SUFFIX -> {+''}int | \$

SIMPLE_LONG_SUFFIX -> {+''}double | COMMON_SUFFIX

USER_DEFINE -> USER_DEFINE_TYPE {+''}identifier | identifier

USER_DEFINE_TYPE -> struct | enum | union

CSTL_BUILTIN -> SEQUENCE | RELATION | string_t | ITERATOR

ITERATOR -> iterator_t | vector_iterator_t | list_iterator_t | slist_iterator_t | deque_iterator_t |

set_iterator_t | map_iterator_t | multiset_iterator_t | multimap_iterator_t |

hash_set_iterator_t | hash_map_iterator_t | hash_multiset_iterator_t | hash_multimap_iterator_t |

string_iterator_t |

input_iterator_t | output_iterator_t | forward_iterator_t | bidirectional_iterator_t | random_access_iterator_t

SEQUENCE -> SEQUENCE_NAME < TYPE_DESCRIPTOR >

SEQUENCE_NAME -> vector_t | list_t | slist_t | deque_t | queue_t | stack_t | priority_queue_t |

set_t | multiset_t | hash_set_t | hash_multiset_t

RELATION -> RELATION_NAME < TYPE_DESCRIPTOR , TYPE_DESCRIPTOR >

RELATION_NAME -> map_t | multimap_t | hash_map_t | hash_multimap_t | pair_t

follow(TYPE_DESCRIPTOR) = {#, >, ,}

follow(C_BUILTIN) = {#, >, ,}

```

follow(USER_DEFINE) = {#, >, ,}
follow(CSTL_BUILTIN) = {#, >, ,}
follow(SIMPLE_BUILTIN) = {#, >, ,}
follow(SIGNED_BUILTIN) = {#, >, ,}
follow(UNSIGNED_BUILTIN) = {#, >, ,}
follow(COMPLEX_SUFFIX) = {#, >, ,}
follow(COMMON_SUFFIX) = {#, >, ,}
follow(SIMPLE_LONG_SUFFIX) = {#, >, ,}
follow(USER_DEFINE_TYPE) = {identifier}
follow(SEQUENCE) = {#, >, ,}
follow(SEQUENCE_NAME) = {<}
follow(RELATION) = {#, >, ,}
follow(RELATION_NAME) = {<}
follow(ITERATOR) = {#, >, ,}

```

```

first(TYPE_DESCRIPTOR) = {char, short, int, long, float, double, char*, bool_t, signed, unsigned,
                         struct, enum, union, identifier,
                         vector_t, list_t, slist_t, deque_t, queue_t, stack_t, priority_queue_t,
                         set_t, map_t, multiset_t, multimap_t, hash_set_t, hash_map_t, hash_multiset_t,
                         hash_multimap_t, pair_t, string_t,
                         iterator_t, vector_iterator_t, list_iterator_t, slist_iterator_t, deque_iterator_t,
                         set_iterator_t, map_iterator_t, multiset_iterator_t, multimap_iterator_t,
                         hash_set_iterator_t, hash_map_iterator_t, hash_multiset_iterator_t, hash_multimap_iterator_t,
                         string_iterator_t,
                         input_iterator_t, output_iterator_t, forward_iterator_t, bidirectional_iterator_t,
                         random_access_iterator_t}

first(C_builtin) = {char, short, int, long, float, double, char*, bool_t, signed, unsigned}
first(SIMPLE_BUILTIN) = {char, short, int, long, float, double, char*, bool_t}
first(SIGNED_BUILTIN) = {signed}
first(UNSIGNED_BUILTIN) = {unsigned}
first(COMPLEX_SUFFIX) = {char, short, int, long, $}
first(COMMON_SUFFIX) = {int, $}
first(SIMPLE_LONG_SUFFIX) = {double, int, $}
first(USER_DEFINE) = {struct, enum, union, identifier}
first(USER_DEFINE_TYPE) = {struct, enum, union}
first(CSTL_BUILTIN) = {vector_t, list_t, slist_t, deque_t, queue_t, stack_t, priority_queue_t,
                      set_t, map_t, multiset_t, multimap_t, hash_set_t, hash_map_t, hash_multiset_t,
                      hash_multimap_t, pair_t, string_t,
                      iterator_t, vector_iterator_t, list_iterator_t, slist_iterator_t, deque_iterator_t,
                      set_iterator_t, map_iterator_t, multiset_iterator_t, multimap_iterator_t,
                      hash_set_iterator_t, hash_map_iterator_t, hash_multiset_iterator_t, hash_multimap_iterator_t,
                      string_iterator_t,
                      input_iterator_t, output_iterator_t, forward_iterator_t, bidirectional_iterator_t,
                      random_access_iterator_t}

```

```

first(SEQUENCE) = {vector_t, list_t, slist_t, deque_t, queue_t, stack_t, priority_queue_t,
                  set_t, multiset_t, hash_set_t, hash_multiset_t}
first(SEQUENCE_NAME) = {vector_t, list_t, slist_t, deque_t, queue_t, stack_t, priority_queue_t,
                       set_t, multiset_t, hash_set_t, hash_multiset_t}
first(RELATION) = {map_t, multimap_t, hash_map_t, hash_multimap_t, pair_t}
first(RELATION_NAME) = {map_t, multimap_t, hash_map_t, hash_multimap_t, pair_t}
first(ITERATOR) = {iterator_t, vector_iterator_t, list_iterator_t, slist_iterator_t, deque_iterator_t,
                   set_iterator_t, map_iterator_t, multiset_iterator_t, multimap_iterator_t,
                   hash_set_iterator_t, hash_map_iterator_t, hash_multiset_iterator_t, hash_multimap_iterator_t,
                   string_iterator_t,
                   input_iterator_t, output_iterator_t, forward_iterator_t, bidirectional_iterator_t,
                   random_access_iterator_t}

```

```

select(TYPE_DESCRIPTOR -> C_BUILTIN) = {char, short, int, long, float, double, char*, bool_t, signed, unsigned}
select(TYPE_DESCRIPTOR -> USER_DEFINE) = {struct, enum, union, identifier}
select(TYPE_DESCRIPTOR -> CSTL_BUILTIN) = {vector_t, list_t, slist_t, deque_t, queue_t, stack_t, priority_queue_t,
                                           set_t, map_t, multiset_t, multimap_t, hash_set_t, hash_map_t,
                                           hash_multiset_t, hash_multimap_t, pair_t, string_t,
                                           iterator_t, vector_iterator_t, list_iterator_t, slist_iterator_t, deque_iterator_t,
                                           set_iterator_t, map_iterator_t, multiset_iterator_t, multimap_iterator_t,
                                           hash_set_iterator_t, hash_map_iterator_t, hash_multiset_iterator_t, hash_multimap_iterator_t,
                                           string_iterator_t,
                                           input_iterator_t, output_iterator_t, forward_iterator_t, bidirectional_iterator_t,
                                           random_access_iterator_t}
select(C_BUILTIN -> SIMPLE_BUILTIN) = {char, short, int, long, float, double, char*, bool_t}
select(C_BUILTIN -> SIGNED_BUILTIN) = {signed}
select(C_BUILTIN -> UNSIGNED_BUILTIN) = {unsigned}
select(SIMPLE_BUILTIN -> char) = {char}
select(SIMPLE_BUILTIN -> short COMMON_SUFFIX) = {short}
select(SIMPLE_BUILTIN -> int) = {int}
select(SIMPLE_BUILTIN -> long SIMPLE_LONG_SUFFIX) = {long}
select(SIMPLE_BUILTIN -> float) = {float}
select(SIMPLE_BUILTIN -> double) = {double}
select(SIMPLE_BUILTIN -> char*) = {char*}
select(SIMPLE_BUILTIN -> bool_t) = {bool_t}
select(SIGNED_BUILTIN -> signed COMPLEX_SUFFIX) = {signed}
select(UNSIGNED_BUILTIN -> unsigned COMPLEX_SUFFIX) = {unsigned}
select(COMPLEX_SUFFIX -> char) = {char}
select(COMPLEX_SUFFIX -> short COMMON_SUFFIX) = {short}
select(COMPLEX_SUFFIX -> int) = {int}
select(COMPLEX_SUFFIX -> long COMMON_SUFFIX) = {long}
select(COMPLEX_SUFFIX -> $) = {#, >, ,}
select(COMMON_SUFFIX -> int) = {int}
select(COMMON_SUFFIX -> $) = {#, >, ,}

```

```

select(SIMPLE_LONG_SUFFIX -> double) = {double}
select(SIMPLE_LONG_SUFFIX -> COMMON_SUFFIX) = {int, #, >, ,}
select(USER_DEFINE -> USER_DEFINE_TYPE identifier) = {struct, enum, union}
select(USER_DEFINE -> identifier) = {identifier}
select(USER_DEFINE_TYPE -> struct) = {struct}
select(USER_DEFINE_TYPE -> enum) = {enum}
select(USER_DEFINE_TYPE -> union) = {union}
select(CSTL_BUILTIN -> SEQUENCE) = {vector_t, list_t, slist_t, deque_t, queue_t, stack_t, priority_queue_t,
                                         set_t, multiset_t, hash_set_t, hash_multiset_t}
select(CSTL_BUILTIN -> RELATION) = {map_t, multimap_t, hash_map_t, hash_multimap_t, pair_t}
select(CSTL_BUILTIN -> ITERATOR) = {iterator_t, vector_iterator_t, list_iterator_t, slist_iterator_t, deque_iterator_t,
                                         set_iterator_t, map_iterator_t, multiset_iterator_t, multimap_iterator_t,
                                         hash_set_iterator_t, hash_map_iterator_t, hash_multiset_iterator_t, hash_multimap_iterator_t,
                                         string_iterator_t,
                                         input_iterator_t, output_iterator_t, forward_iterator_t, bidirectional_iterator_t,
                                         random_access_iterator_t}
select(CSTL_BUILTIN -> string_t) = {string_t}
select(SEQUENCE -> SEQUENCE_NAME < TYPE_DESCRIPTOR >) = {vector_t, list_t, slist_t, deque_t, queue_t, stack_t,
                                         priority_queue_t, set_t, multiset_t, hash_set_t, hash_multiset_t}
select(SEQUENCE_NAME -> vector_t) = {vector_t}
select(SEQUENCE_NAME -> list_t) = {list_t}
select(SEQUENCE_NAME -> slist_t) = {slist_t}
select(SEQUENCE_NAME -> deque_t) = {deque_t}
select(SEQUENCE_NAME -> queue_t) = {queue_t}
select(SEQUENCE_NAME -> stack_t) = {stack_t}
select(SEQUENCE_NAME -> priority_queue_t) = {priority_queue_t}
select(SEQUENCE_NAME -> set_t) = {set_t}
select(SEQUENCE_NAME -> multiset_t) = {multiset_t}
select(SEQUENCE_NAME -> hash_set_t) = {hash_set_t}
select(SEQUENCE_NAME -> hash_multiset_t) = {hash_multiset_t}
select(RELATION -> RELATION_NAME < TYPE_DESCRIPTOR , TYPE_DESCRIPTOR >) = {map_t, multimap_t, hash_map_t,
                                         hash_multimap_t, pair_t}
select(RELATION_NAME -> map_t) = {map_t}
select(RELATION_NAME -> multimap_t) = {multimap_t}
select(RELATION_NAME -> hash_map_t) = {hash_map_t}
select(RELATION_NAME -> hash_multimap_t) = {hash_multimap_t}
select(RELATION_NAME -> pair_t) = {pair_t}
select(ITERATOR -> iterator_t) = {iterator_t}
select(ITERATOR -> vector_iterator_t) = {vector_iterator_t}
select(ITERATOR -> list_iterator_t) = {list_iterator_t}
select(ITERATOR -> slist_iterator_t) = {slist_iterator_t}
select(ITERATOR -> deque_iterator_t) = {deque_iterator_t}
select(ITERATOR -> set_iterator_t) = {set_iterator_t}
select(ITERATOR -> map_iterator_t) = {map_iterator_t}

```

```
select(ITERATOR -> multiset_iterator_t) = {multiset_iterator_t}
select(ITERATOR -> multimap_iterator_t) = {multimap_iterator_t}
select(ITERATOR -> hash_set_iterator_t) = {hash_set_iterator_t}
select(ITERATOR -> hash_map_iterator_t) = {hash_map_iterator_t}
select(ITERATOR -> hash_multiset_iterator_t) = {hash_multiset_iterator_t}
select(ITERATOR -> hash_multimap_iterator_t) = {hash_multimap_iterator_t}
select(ITERATOR -> string_iterator_t) = {string_iterator_t}
select(ITERATOR -> input_iterator_t) = {input_iterator_t}
select(ITERATOR -> output_iterator_t) = {output_iterator_t}
select(ITERATOR -> forward_iterator_t) = {forward_iterator_t}
select(ITERATOR -> bidirectional_iterator_t) = {bidirectional_iterator_t}
select(ITERATOR -> random_access_iterator_t) = {random_access_iterator_t}
```

对于类型的语法采用递归下降的方法进行分析，其中还添加了一些语义的操作。

第六节 类型的分类于以及使用

将符合类型描述语法的类型描述分为三类：C 语言内建类型，libcstl 内建类型，用户自定义类型。

C 语言内建类型就是 C 语言本身的类型如 int, short, double 等等，上面的类型描述语法中产生式 **C_BUILTIN** 表示的就是 C 语言的内建类型(其中 bool_t 不是 C 语言本身的类型，在这里也将它归类为 C 内建类型)。

libcstl 内建类型就是 libcstl 内部定义的类型如 vector_t, iterator_t 等等，上面的类型描述语法中的产生式 **CSTL_BUILTIN** 表示的就是 libcstl 内建类型(其中 bool_t 是 libcstl 定义的类型，但是它的行为和用法与 C 内建类型是一致的所以将它归为 C 内建类型)。

用户自定义类型，就是用户在开发过程中定义的结构体，联合，枚举等。其中有一个特殊的情况就是对类型的重定义如：typedef int myint;这样的重定义如果使用 type_duplicate 进行类型复制，那么这个新类型 myint 与原来的类型 int 就是同一种类型，但是如果将 myint 使用 type_register 注册，那么 myint 就是一个新的用户自定义类型，虽然它与 int 的含义和操作都完全一致。

各种类型之间是有区别的，对于 C 内建类型和 libcstl 内建类型，它们是不需要用户注册的，因为它们是在 libcstl 库内部注册的，用户只管使用就可以了，但是用户自定义的类型用户必须负责注册，否则 libcstl 不知道这个类型。libcstl 内建类型是一种特殊的用户自定义类型(libcstl 定义的)，只不过它不需要用户注册。

在使用的时候 C 内建类型与另外两种类型之间有区别，尤其是在直接使用数据的接口中，如 vector_push_back 这样的函数，它要求用户直接输入数据或者是保存数据的变量。例如：

现在有三个 vector 容器，它们分别保存三种不同种类的数据类型：

```
vector_t* pvec_int = create_vector(int);
vector_t* pvec_list = create_vector(list<double>);
vector_t* pvec_foo = create_vector(foo_t); /* typedef _tagfoo{ ... } foo_t; */
```

现在对这三个容器进行 vector_push_back 操作：

```
vector_push_back(pvec_int, 10); /* 或者 int n = 10; vector_push_back(pvec_int, n); */
vector_push_back(pvec_list, plist_sample); /* plist_sample = create_list(int); */
vector_push_back(pvec_foo, &foo_sample); /* foo_t foo_sample; */
```

对于这样直接要求使用类型数据的接口函数，如果容器中保存的是 C 内建类型，那么就直接使用数据常量或者是

变量。但是对于保存的是 libcstl 内建类型和用户自定义类型的容器，要求使用指向类型数据的指针。

此外对于在保存 C 内建类型的容器上使用直接使用数据的接口函数的时候还要注意的就是类型自动转换的问题，libcstl 的这类函数不支持类型的自动转换。例如：

```
vector_t* pvec_double = create_vector(double);
vector_push_back(pvec_double, 10);
```

这样的操作是错误的，这个函数不会将 10 自动的转换成 double 类型。但是如下几种方法是可行的：

```
vector_push_back(pvec_double, n); /* int n = 1; */
vector_push_back(pvec_double, 10.0);
vector_push_back(pvec_double, 1e1);
vector_push_back(pvec_double, (double)10);
```

这样是可以的。

第五章 libcstl 迭代器

迭代器将对于容器的操作封装起来，对外界提供一个统一的操作接口。使用这样一套同意的对外接口就可以完成对于任何类型的容器的操作，而不必关心各个容器的不同点。

第一节 迭代器的机制

迭代器最基本的语义类似与指针，指向容器中的数据。通过从容器的开头到末尾的迭代，遍历容器中每一个数据，通过迭代器来引用容器中相应的数据，对数据进行操作。迭代器定义了一个便利的范围，对于一个容器中的全部数据，迭代的起点是从容器中的第一个数据开始的，终点是容器中最后一个数据的下一个位置。这样的好处是方便遍历，当当前的迭代器位置等于终点的时候，遍历就借宿了，对于空容器起点和终点相等，也样也不用做特殊的处理：



图 5.1 迭代器

迭代器定义了一组统一的操作接口，但是并不是所有的容器都适用全部操作，因为各个容器的不同属性，适用的迭代器操作也是不同的。对于不同的容器它们的迭代器分别对应于不同类型的迭代器，使用迭代器的类型就可以区分迭代器适用的操作。容器可以分为随机访问迭代器(如向量的迭代器)，双向迭代器(如链表的迭代器)，前向迭代器(如单向链表迭代器)以及输入和输出迭代器。上述迭代器类型的能力依次递减，它们适用的操作也依次减少。

第二节 迭代器的代码结构

迭代器的结构是一个复杂的结构，它不仅要包含数据的信息还有包含容器本身的信息，迭代器的类型以及和容器内部结构相关的信息：

```
typedef struct _tagiterator
```

```

{
    /* 容器内部结构的信息 */
    union
    {
        char*      _pc_corepos;      /* 向量等容器中数据的位置 */
        struct
        {
            /* 与双端队列相关的结构 */
            char*      _pc_corepos;      /* 数据位置 */
            char*      _pc_first;       /* 第一个数据的位置 */
            char*      _pc_afterlast;   /* 最后一个数据的下一个位置 */
            char**     _ppc_mappos;     /* 映射表的位置 */
        }_t_dequepos;
        struct
        {
            /* 与 avl 树和 rb 树相关的结构 */
            char*      _pc_corepos;      /* 数据的位置 */
            void*     _pt_tree;         /* avl 树或者 rb 树的位置 */
        }_t_treepos;
        struct           /* 与哈希表相关的结构 */
        {
            char*      _pc_corepos;      /* 数据的位置 */
            char*      _pc_bucketpos;   /* 桶的位置 */
            void*     _pt_hashtable;    /* 哈希表的位置 */
        }_t_hashpos;
    }_t_pos;
    void*      _pt_container;      /* 容器的位置 */
    containertype_t _t_containertype; /* 容器的类型 */
    iteratortype_t _t_iteratortype;  /* 迭代器的类型 */
}iterator_t;

```

通过迭代器中的容器类型以及指向容器的指针可以得到相应的容器，然后利用与容器结构以及指向数据的指针，容器本身的迭代器操作函数可以作出具体的操作。所以迭代器提供的对外的接口都是对于不同容器本身操作的封装，因为只有容器本身最了解自己的数据结构以及操作过程。

第三节 对外接口

迭代器提供对外的接口，这些接口都是为用户提供的，但是并不是所有类型的容器都适用这些接口。

iterator_get_value	获得迭代器指向的数据的内容。
iterator_set_value	设置迭代器指向的数据的内容。
iterator_get_pointer	获得指向迭代器引用的数据的指针。
iterator_next	获得引用当前迭代器的下一个位置的数据的迭代器。

iterator_prev	获得引用当前迭代器的前一个位置的数据的迭代器。
iterator_next_n	获得引用当前迭代器的下 n 个位置的数据的迭代器。
iterator_prev_n	获得引用当前迭代器的前 n 个位置的数据的迭代器。
iterator_equal	测试两个迭代器是否相等。
iterator_not_equal	测试两个迭代器是否不等。
iterator_less	测试第一个迭代器是否小于第二个迭代器。
iterator_less_equal	测试第一个迭代器是否小于等于第二个迭代器。
iterator_greater	测试第一个迭代器是否大于第二个迭代器。
iterator_greater_equal	测试第一个迭代器是否大于等于第二个迭代器。
iterator_at	使用下标来随机访问迭代器引用的容器中的数据。
iterator_minus	计算两个迭代器的差。
iterator_advance	一次迭代器多步。
iterator_distance	计算两个迭代器之间的距离。

接口的原型:

```
void iterator_get_value(iterator_t t_iter, void* pv_value);
```

描述:

获得迭代器指向的数据的内容。

参数:

t_iter 输入迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意:

这个函数只能应用于输入迭代器。t_iter 为无效的迭代器或者是容器的末尾，函数的行为是未定义的。pv_value 为 NULL 或者是缓冲区的大小不能够保存迭代器引用的数据的时候，函数是未定义的。

```
void iterator_set_value(iterator_t t_iter, const void* cpv_value);
```

描述:

设置迭代器指向的数据的内容。

参数:

t_iter 输出迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意:

这个函数只能应用于输出迭代器。t_iter 为无效的迭代器或者是容器的末尾，函数的行为是未定义的。pv_value 为 NULL，函数是未定义的。此外这个函数不能够用户设置关联容器的数据，如果迭代器是关联容器的迭代器，那么这个

函数的行为是未定义的。

```
const void* iterator_get_pointer(iterator_t t_iter);
```

描述:

获得指向迭代器引用的数据的指针。

参数:

t_iter 输入迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

这个函数只能应用于输入迭代器。t_iter 为无效的迭代器或者容器的末尾，函数的行为是未定义的。这个函数的主要用途是通过指针直接引用容器中的数据，减少了 iterator_get_value 这样的函数中的数据复制的过程。不建议用户通过指针修改数据的内容，强烈建议不要通过这个指针修改关联容器中的数据。

```
iterator_t iterator_next(iterator_t t_iter);
```

描述:

获得引用当前迭代器的下一个位置的数据的迭代器。

参数:

t_iter 前向迭代器。

返回值:

引用当前迭代器的下一个位置的数据的迭代器。

注意:

这个函数只能应用于前向迭代器。t_iter 为无效的迭代器或者是容器的末尾，函数的行为是未定义的。

```
iterator_t iterator_prev(iterator_t t_iter);
```

描述:

获得引用当前迭代器的前一个位置的数据的迭代器。

参数:

t_iter 双向迭代器。

返回值:

引用当前迭代器的前一个位置的数据的迭代器。

注意:

这个函数只能应用于双向迭代器。t_iter 为无效的迭代器或者是容器的第一个位置，函数的行为是未定义的。

```
iterator_t iterator_next_n(iterator_t t_iter, int n_step);
```

描述:

获得引用当前迭代器的下 n 个位置的数据的迭代器。

参数:

t_iter 随机访问迭代器。

n_step 步数。

返回值:

引用当前迭代器的下 n 个位置的数据的迭代器。

注意：

这个函数只能应用于随机访问迭代器。`t_iter` 为无效的迭代器，函数的行为是未定义的。当前的迭代器与步数计算之后的结果超过了容器的有效范围的情况下，函数的行为是未定义的。当 `n_step > 0` 的时候，迭代器向容器的末尾迭代，当 `n_step < 0` 的时候，迭代器向容器的开头迭代。当 `n_step == 0` 的时候，迭代器不动。

```
iterator_t iterator_prev_n(iterator_t t_iter, int n_step);
```

描述：

获得引用当前迭代器的前 `n` 个位置的数据的迭代器。

参数：

`t_iter` 随机访问迭代器。

`n_step` 步数。

返回值：

引用当前迭代器的前 `n` 个位置的数据的迭代器。

注意：

这个函数只能应用于随机访问迭代器。`t_iter` 为无效的迭代器，函数的行为是未定义的。当前的迭代器与步数计算之后的结果超过了容器的有效范围的情况下，函数的行为是未定义的。当 `n_step > 0` 的时候，迭代器向容器的开头迭代，当 `n_step < 0` 的时候，迭代器向容器的末尾迭代。当 `n_step == 0` 的时候，迭代器不动。

```
bool_t iterator_equal(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述：

测试两个迭代器是否相等。

参数：

`t_iterfirst` 输入迭代器。

`t_itersecond` 输入迭代器。

返回值：

两个迭代器相等返回 `true`，否则返回 `false`。

注意：

这个函数只能应用于输入迭代器。当两个迭代器中存在无效迭代器的时候，函数的行为是未定义的。两个迭代器相等是指两个迭代器在引用同一个容器中的相同的位置，如果两个迭代器的容器类型不同，则返回 `false`。

```
bool_t iterator_not_equal(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述：

测试两个迭代器是否不等。

参数：

`t_iterfirst` 输入迭代器。

`t_itersecond` 输入迭代器。

返回值：

两个迭代器不等返回 `true`，否则返回 `false`。

注意：

这个函数只能应用于输入迭代器。当两个迭代器中存在无效迭代器的时候，函数的行为是未定义的。两个迭代器相等是指两个迭代器在引用同一个容器中的相同的位置，如果两个迭代器的容器类型不同，则返回 `true`。

```
bool_t iterator_less(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述:

测试第一个迭代器是否小于第二个迭代器。

参数:

t_iterfirst 随机访问迭代器。

t_itersecond 随机访问迭代器。

返回值:

第一个迭代器小于第二个迭代器返回 true, 否则返回 false。

注意:

这个函数只能应用于随机访问迭代器。当两个迭代器中存在无效迭代器的时候, 函数的行为是未定义的。如果两个迭代器了容器类型不同或者不属于同一个容器则函数的行为是未定义的。

```
bool_t iterator_less_equal(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述:

测试第一个迭代器是否小于等于第二个迭代器。

参数:

t_iterfirst 随机访问迭代器。

t_itersecond 随机访问迭代器。

返回值:

第一个迭代器小于等于第二个迭代器返回 true, 否则返回 false。

注意:

这个函数只能应用于随机访问迭代器。当两个迭代器中存在无效迭代器的时候, 函数的行为是未定义的。如果两个迭代器了容器类型不同或者不属于同一个容器则函数的行为是未定义的。

```
bool_t iterator_greater(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述:

测试第一个迭代器是否大于第二个迭代器。

参数:

t_iterfirst 随机访问迭代器。

t_itersecond 随机访问迭代器。

返回值:

第一个迭代器大于第二个迭代器返回 true, 否则返回 false。

注意:

这个函数只能应用于随机访问迭代器。当两个迭代器中存在无效迭代器的时候, 函数的行为是未定义的。如果两个迭代器了容器类型不同或者不属于同一个容器则函数的行为是未定义的。

```
bool_t iterator_greater_equal(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述:

测试第一个迭代器是否大于等于第二个迭代器。

参数:

t_iterfirst 随机访问迭代器。

t_itersecond 随机访问迭代器。

返回值:

第一个迭代器大于等于第二个迭代器返回 true, 否则返回 false。

注意:

这个函数只能应用于随机访问迭代器。当两个迭代器中存在无效迭代器的时候, 函数的行为是未定义的。如果两个迭代器了容器类型不同或者不属于同一个容器则函数的行为是未定义的。

```
void* iterator_at(iterator_t t_iter, int n_index);
```

描述:

使用下标来随机访问迭代器引用的容器中的数据。

参数:

t_iter 随机访问迭代器。
n_index 下标。

返回值:

指向被引用的容器中的数据。

注意:

这个函数只能应用于随机访问迭代器。当迭代器是无效迭代器的时候, 函数的行为是未定义的。迭代器与下标 n_index 计算的结果超过容器的范围的时候, 函数的行为是未定义的。当 n_index>0 的时候返回的是从当前迭代器的向容器末尾方向的数据。当 n_index<0 的时候返回的时候当前迭代器向容器开始方向的数据。当 n_index==0 的时候返回的是当前迭代器指的数据。

```
int iterator_minus(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述:

计算两个迭代器的差。

参数:

t_iterfirst 随机访问迭代器。
t_itersecond 随机访问迭代器。

返回值:

两个迭代器的差。

注意:

这个函数只能应用于随机访问迭代器。当两个迭代器中存在无效迭代器的时候, 函数的行为是未定义的。如果两个迭代器了容器类型不同或者不属于同一个容器则函数的行为是未定义的。

接下来的两个函数是迭代器的辅助函数, 这两个函数可以适用于任何迭代器类型。

```
iterator_t iterator_advance(iterator_t t_iter, int n_step);
```

描述:

获得引用当前迭代器的下 n 个位置的数据的迭代器。

参数:

t_iter 任何类型的迭代器。
n_step 步数。

返回值:

引用当前迭代器的下 n 个位置的数据的迭代器。

注意:

这个函数能应用于任何类型的迭代器。`t_iter` 为无效的迭代器，函数的行为是未定义的。当前的迭代器与步数计算之后的结果超过了容器的有效范围的情况下，函数的行为是未定义的。当 `n_step>0` 的时候，迭代器向容器的末尾迭代，当 `n_step<0` 的时候，迭代器向容器的开头迭代。当 `n_step==0` 的时候，迭代器不动。

```
int iterator_distance(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述：

计算两个迭代器之间的距离。

参数：

`t_iterfirst` 任何类型的迭代器。

`t_itersecond` 任何类型的迭代器。

返回值：

两个迭代器之间的距离。

注意：

这个函数能应用于任何类型的迭代器。当两个迭代器中存在无效迭代器的时候，函数的行为是未定义的。如果两个迭代器了容器类型不同或者不属于同一个容器则函数的行为是未定义的。

第四节 内部接口

迭代器提供给 libcstl 内部使用的接口，这些接口适用于所有类型的迭代器：

<code>_iterator_same_type</code>	测试两个迭代器的类型是否相同。
<code>_iterator_before</code>	测试第一个迭代器是否位于第二个迭代器之前。
<code>_iterator_limit_type</code>	测试迭代器是否符合类型限制规则。
<code>_iterator_valid_range</code>	测试两个迭代器是否能够组成一个可用的范围。
<code>_iterator_same_elem_type</code>	测试两个迭代器引用的数据的类型是否相同。
<code>_iterator_get_typestyle</code>	获得迭代器引用的数据的类型模式。
<code>_iterator_get_typebasename</code>	获得迭代器引用的数据的基本类型名字。
<code>_iterator_get_typeinfo</code>	获得指向迭代器引用的数据类型的信息结构。
<code>_iterator_get_typename</code>	获得迭代器所引用的数据的类型名字。
<code>_iterator_get_typecopy</code>	获得迭代器引用类型的拷贝函数。
<code>_iterator_get_typesize</code>	获得迭代器应用的类型的大小。
<code>_iterator_allocate_destroy_elem</code>	申请一个与迭代器引用的数据相同类型的元素并初始化它。
<code>_iterator_deallocate_destroy_elem</code>	销毁一个与迭代器引用的数据相同类型的元素并释放它占用的内存。

```
bool_t _iterator_same_type(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述:

测试两个迭代器的类型是否相同。

参数:

t_iterfirst 任何类型的迭代器。

t_itersecond 任何类型的迭代器。

返回值:

如果两个迭代器的类型以及两个迭代器引用的容器类型相同, 返回 true, 否则返回 false。

注意:

这个函数能应用于任何类型的迭代器。

```
bool_t _iterator_before(iterator_t t_iterfirst, iterator_t t_itersecond);
```

描述:

测试第一个迭代器是否位于第二个迭代器之前。

参数:

t_iterfirst 任何类型的迭代器。

t_itersecond 任何类型的迭代器。

返回值:

如果第一个迭代器引用的数据容器中的位置比第二个迭代器引用的数据在迭代器中的位置更靠近开头, 那么返回 true, 否则返回 false。两个迭代器相等返回 false。

注意:

这个函数能应用于任何类型的迭代器。两个迭代器必须是有效的迭代器, 两个迭代器的类型必须相同并且必须引用的是同一个容器中的数据, 否则函数的行为是未定义的。

```
bool_t _iterator_limit_type(iterator_t t_iter, iteratortype_t t_limittype);
```

描述:

测试一个迭代器是否符合类型限制规则。

参数:

t_iter 任何类型的迭代器。

t_limittype 限制类型。

返回值:

如果迭代器的能力满足限制类型的要求那么返回 true, 否则返回 false。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效的迭代器, 限制类型也必须是有效的迭代器类型, 否则函数的行为是未定义的。

```
bool_t _iterator_valid_range(
    iterator_t t_first, iterator_t t_last, iteratortype_t t_type);
```

描述:

测试一个迭代器对是否能够组成满足指定迭代器类型的有效区间。

参数:

t_first 任何类型的迭代器。

t_last 任何类型的迭代器。

`t_type` 限制类型。

返回值:

如果两个迭代器的能力满足限制类型的要求并且, `t_first` 在 `t_last` 之前那么返回 `true`, 否则返回 `false`。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效的迭代器并且属于同一容器, 限制类型也必须是有效的迭代器类型, 否则函数的行为是未定义的。

```
bool_t _iterator_same_elem_type(iterator_t t_first, iterator_t t_last);
```

描述:

测试两个迭代器引用的数据的类型是否相同。

参数:

`t_first` 任何类型的迭代器。

`t_last` 任何类型的迭代器。

返回值:

两个迭代器引用的数据的类型相同返回 `true`, 否则返回 `false`。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效, 否则函数的行为是未定义的。

```
_typestyle_t _iterator_get_typestyle(iterator_t t_iter);
```

描述:

获得迭代器引用的数据的类型模式。

参数:

`t_iter` 任何类型的迭代器。

返回值:

返回迭代器引用的数据的类型模式。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效, 否则函数的行为是未定义的。

```
const char* _iterator_get_typebasename(iterator_t t_iter);
```

描述:

获得迭代器引用的数据的基本类型名字。

参数:

`t_iter` 任何类型的迭代器。

返回值:

返回迭代器引用的数据的基本类型名字。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效, 否则函数的行为是未定义的。

```
_typeinfo_t* _iterator_get_typeinfo(iterator_t t_iter);
```

描述:

获得迭代器引用的数据的类型信息。

参数:

`t_iter` 任何类型的迭代器。

返回值:

返回指向迭代器引用的数据的类型信息结构的指针。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效，否则函数的行为是未定义的。

```
const char* _iterator_get_typename(iterator_t t_iter);
```

描述:

获得迭代器引用的数据的类型名字。

参数:

`t_iter` 任何类型的迭代器。

返回值:

返回迭代器引用的数据的类型名字。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效，否则函数的行为是未定义的。

```
binary_function_t _iterator_get_typecopy(iterator_t t_iter);
```

描述:

获得迭代器引用的数据的类型的拷贝函数。

参数:

`t_iter` 任何类型的迭代器。

返回值:

返回迭代器引用的数据的类型的拷贝函数。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效，否则函数的行为是未定义的。

```
size_t _iterator_get_typesize(iterator_t t_iter);
```

描述:

获得迭代器引用的数据的类型的大小。

参数:

`t_iter` 任何类型的迭代器。

返回值:

返回迭代器引用的数据的类型的大小。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效，否则函数的行为是未定义的。

```
void* _iterator_allocate_init_elem(iterator_t t_iter);
```

描述:

根据迭代器所引用的数据的类型再分配一个数据并且初始化这个数据。

参数:

`t_iter` 任何类型的迭代器。

返回值:

返回指向分配并初始化后的数据的指针。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效，否则函数的行为是未定义的。

```
void _iterator_deallocate_destroy_elem(iterator_t t_iter, void* pv_value);
```

描述:

销毁一个根据迭代器所引用的数据的类型再分配并且初始化的数据。

参数:

t_iter 任何类型的迭代器。

pv_value 指向数据的指针。

返回值:

无。

注意:

这个函数能应用于任何类型的迭代器。迭代器必须是有效，pv_value 不能为 NULL，否则函数的行为是未定义的。

pv_value 不是这个迭代器应用的数据的类型，那么函数行为是未定义的。

第六章 vector 容器

vector 容器是一种序列容器，它支持数据的随机访问，在容器末尾插入或者删除数据花费常数时间，但是在容器的开始或者中间插入或者删除数据要花费线性时间。容器的存储随着保存数据的增加而自动增长，并且容器的空间是连续的。

第一节 vector 容器的机制

vector 容器中的数据都保存在一个连续的内存中，为了保证在末尾对数据的操作是常数时间的所以数据都从连续地址空间的一头开始一次保存数据。申请的内存要比实际保存数据所需要的内存要多一些，这样就不必在每一次在末尾添加一个数据就要为容器重新分配一次内存。



图 6.1 vector 容器的内存

当 vector 容器中剩余的内存保存不了要求插入的数据的时候就引起了容器的内存的重新分配，分配的算法是：

1) $\text{new_size} = \text{old_size} + 2 * \text{insert_size}$

这个算法对于每一次插入大量的数据来说很有效但是每次插入的数据量很少的时候效率就会很低，如每次插入一个数据那么，一次内存的重新分配仅仅增加了两个数据大小的内存，这样很快就会用完。(libcstl 现在采用的是这样的算法)。

2) $\text{new_size} = \text{old_size} != 0 ? 2 * \text{old_size} : 1$

这种算法就是每次分配的量是原有内存的倍数，但是如果原有的内存量很小，突然插入大量的数据，这样一次重新分配可能还是没有满足要求还要再次进行重新分配。

3) $\text{new_size} = \text{old_size} + \max(\text{old_size}, \text{insert_size})$

这个算法当 $\text{insert_size} > \text{old_size}$ 的时候，重新分配的内存刚刚好容纳下插入之后的数据，这样马上内存就饱和了，再次插入还是会引起内存的重新分配。

4)

```
grow_size = (old_size + insert_size)/2;
```

```
grow_size = grow_size > threshold_size ? grow_size : threshold_size
```

```
new_size = old_size + insert_size + grow_size
```

threshold_size 是一个阀值，它是每次增加的最小数量。 grow_size 是插入数据后应该预留的内存的数量，这个值由

`insert_size` 和 `old_size` 决定。这个算法避免了突然的猛烈增长。

5)

```
grow_size = β * old_size + (1- β) * insert_size;  
grow_size = grow_size > threshold_size ? grow_size : threshold_size  
new_size = old_size + insert_size + grow_size
```

这个算法与 4) 类似，但是在 `grow_size` 受那个大小影响依赖于 β ，它是一个百分比，掌握两个 size 影响 `grow_size` 的程度。

(libcstl 采用的是第一种算法，在将来的版本可能改成第 4 个)。

对于 `vector` 中末尾的数据和开头以及中间的数据的操作完全不同。在容器的末尾插入数据很方便和快捷：

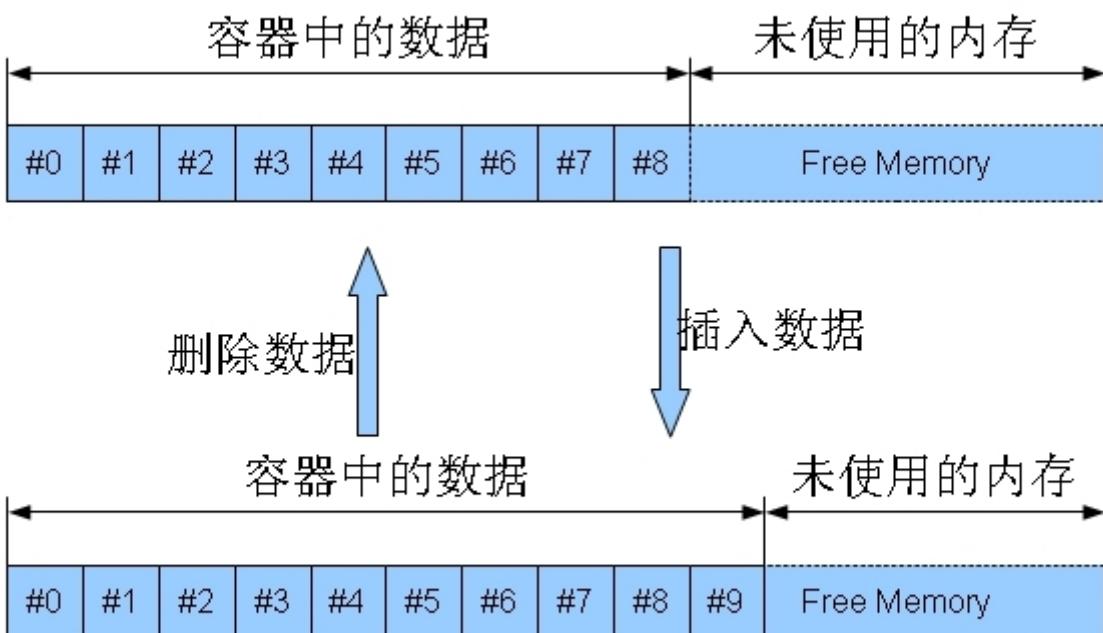


图 6.2 在 `vector` 的末尾插入和删除数据。

在容器的开头和中间插入或者删除数据就不是很方便了，而且效率也不高：

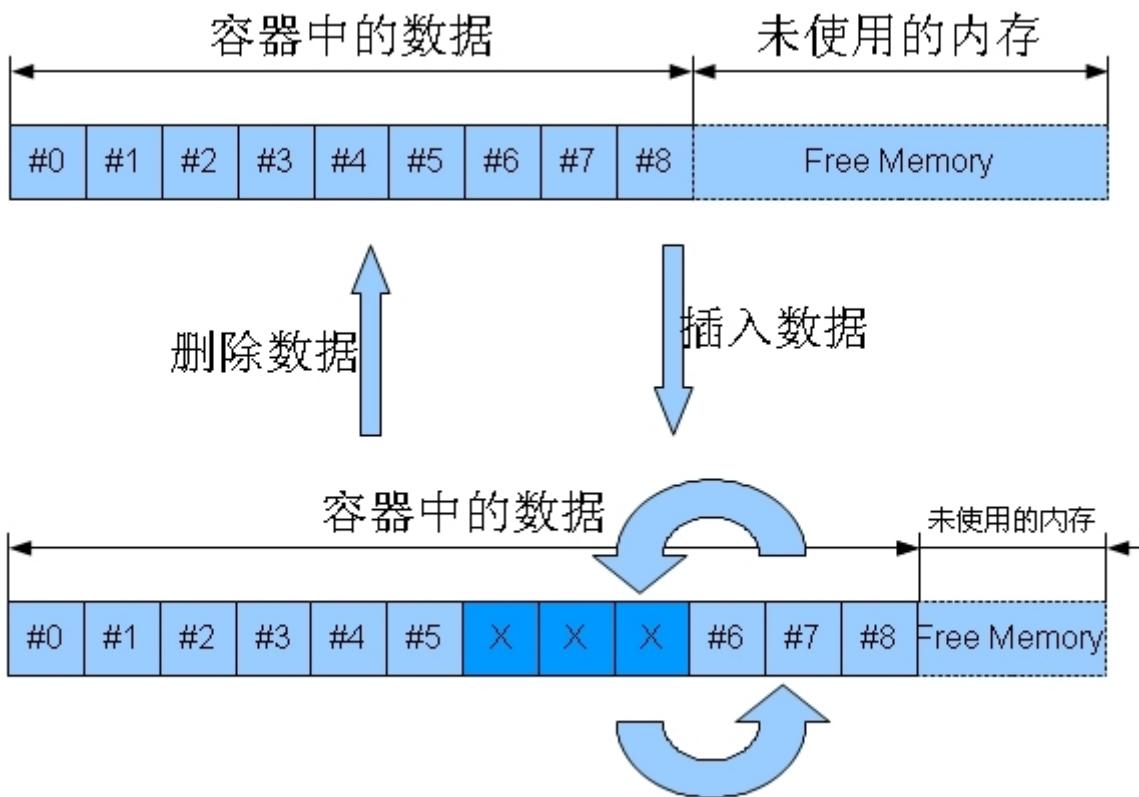


图 6.3 在中间插入和删除数据

第二节 vector 迭代器

vector 的结构比较简单，只涉及到每一个元素的位置，与普通的指针没有什么区别。向前移动和向后移动仅仅是将指针指向下一个元素或者指向前一个元素。同时 vector 迭代器还是随机访问迭代器，可以一次跨越多个元素移动。begin 和 end 也十分好确定：

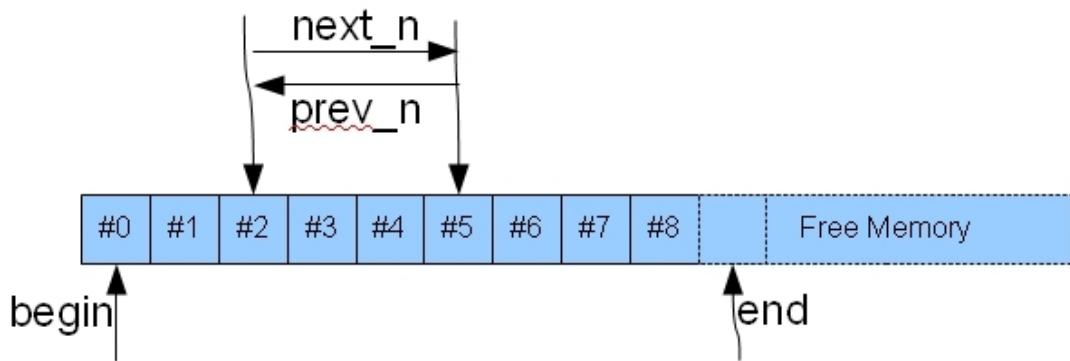


图 6.4 vector 迭代器

所以对于 vector 迭代器来说，迭代器的结构就变成了：

```
typedef struct _tagiterator
{
    /* 容器内部结构的信息 */
    union
    {
        char*      _pc_corepos;      /* 向量等容器中数据的位置 */
        _t_pos;
        void*       _pt_container;   /* 容器的位置 */
        containertype_t _t_containertype; /* 容器的类型 */
        iteratortype_t _t_iteratortype; /* 迭代器的类型 */
    }iterator_t;
```

第三节 vector 的代码结构

vector 的结构很简单，其中除了保存数据类型与内存管理的结构之外主要的就是指向容器内存的三个指针，一个表示容器的开头，一个表示容器中数据的结尾，一个表示容器内存的结束位置：

```
typedef struct _tagvector
{
    _typeinfo_t _t_typeinfo;          /* 数据类型信息 */
    _alloc_t    _t_allocator;         /* 内存管理 */

    _byte_t*    _pby_start;           /* 容器的开始位置 */
    _byte_t*    _pby_finish;          /* 数据的末尾 */
    _byte_t*    _pby_endofstorage;    /* 容器内存的结束位置 */
}vector_t;
```

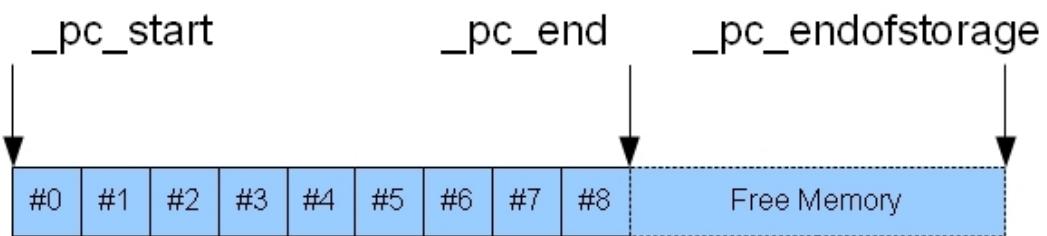


图 6.5 vector 结构

第四节 外部接口

vector 要提供一系列的接口给用户使用，下面这些接口是给用户使用的：

<code>create_vector</code>	创建 vector 容器。
<code>vector_init</code>	初始化一个空的 vector 容器。
<code>vector_init_n</code>	初始化一个包含 n 个默认数据的 vector 容器。
<code>vector_init_elem</code>	初始化一个包含 n 个指定数据的 vector 容器。
<code>vector_init_copy</code>	通过拷贝既存的 vector 容器中的数据来初始化 vector 容器。
<code>vector_init_copy_range</code>	通过拷贝一个范围内的数据来初始化 vector 容器。
<code>vector_destroy</code>	销毁 vector 容器。
<code>vector_size</code>	返回 vector 容器中数据的个数。
<code>vector_empty</code>	测试 vector 容器是否为空。
<code>vector_max_size</code>	返回 vector 容器能够保存数据的最大数量。
<code>vector_capacity</code>	返回 vector 容器的容量。
<code>vector_reserve</code>	设置 vector 容器的容量。
<code>vector_equal</code>	测试两个 vector 容器是否相等。
<code>vector_not_equal</code>	测试两个 vector 容器是否不等。
<code>vector_less</code>	测试第一个 vector 是否小于第二个 vector 容器。
<code>vector_less_equal</code>	测试第一个 vector 是否小于等于第二个 vector 容器。
<code>vector_greater</code>	测试第一个 vector 是否大于第二个 vector 容器。
<code>vector_greater_equal</code>	测试第一个 vector 是否大于等于第二个 vector 容器。
<code>vector_assign</code>	使用一个既存的 vector 给 vector 容器赋值。
<code>vector_assign_elem</code>	使用 n 个指定的数据为 vector 容器赋值。

vector_assign_range	使用指定范围的数据给 vector 容器赋值。
vector_swap	交换两个 vector 容器中的数据。
vector_at	通过下标随机访问 vector 容器中的数据。
vector_front	访问 vector 容器中的第一个数据。
vector_back	访问 vector 容器中的最后一个数据。
vector_begin	返回指向 vector 容器开始的迭代器。
vector_end	返回指向 vector 容器末尾的迭代器。
vector_insert	在指定的位置插入数据。
vector_insert_n	在指定的位置插入 n 个数据。
vector_insert_range	在指定的位置插入指定范围内的数据。
vector_push_back	在 vector 容器的末尾添加一个指定的数据。
vector_pop_back	删除 vector 容器末尾的一个数据。
vector_erase	删除 vector 容器中指定位置的数据。
vector_erase_range	删除 vector 容器中指定范围的数据。
vector_clear	删除 vector 容器中的所有数据。
vector_resize	重新设置 vector 容器中数据的个数。超过原有数据的部分使用默认值填充。
vector_resize_elem	重新设置 vector 容器中数据的个数。超过原有数据的部分使用指定数据填充。

接口原型:

```
vector_t* create_vector(typename);
```

描述:

创建一个保存指定类型的 vector 容器。

参数:

typename 保存的数据类型。

返回值:

指向 vector_t 容器的指针。

注意:

这个函数的参数比较特殊，它是描述容器元素的数据类型的表达式，关于类型机制以及类型描述语法请参考第四章。如果创建失败返回 NULL。

```
void vector_init(vector_t* pvec_vector);
```

描述:

初始化一个空的 vector 容器。

参数:

pvec_vector 指向被初始化的 vector 容器的指针。

返回值:

无。

注意:

被初始化的容器一定是使用 `create_vector` 创建的容器, 否则函数的行为是未定义的。`pvec_vector == NULL`, 函数的行为是未定义的。使用 `vector_init` 初始化之后的 `vector` 容器, `vector_size() == 0`, `vector_capacity() == 0`。

```
void vector_init_n(vector_t* pvec_vector, size_t t_count);
```

描述:

初始化一个包含 `n` 个数据的 `vector` 容器。

参数:

`pvec_vector` 指向被初始化的 `vector` 容器的指针。
`t_count` 初始化后包含的数据的个数。

返回值:

无。

注意:

被初始化的容器一定是使用 `create_vector` 创建的容器, 否则函数的行为是未定义的。`pvec_vector == NULL`, 函数的行为是未定义的。使用 `vector_init_n` 初始化之后的 `vector` 容器, `vector_size() == t_count`, 容量符合前面描述的算法, 并且 `t_count` 都是容器中保存的数据类型的默认数据。

```
void vector_init_elem(vector_t* pvec_vector, size_t t_count, elem);
```

描述:

使用用户指定的数据初始化 `vector` 容器。

参数:

`pvec_vector` 指向被初始化的 `vector` 容器的指针。
`t_count` 初始化后包含的数据的个数。
`elem` 指定的数据。

返回值:

无。

注意:

如果 `pvec_vector == NULL` 则函数的行为是未定义的。`vector` 容器必须是使用 `create_vector` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。使用 `vector_init_n` 初始化之后的 `vector` 容器, `vector_size() == t_count`, 容量符合前面描述的算法, 并且 `t_count` 是指定的数据 `elem`, `elem` 的限制请参考第四章。

```
void vector_init_copy(vector_t* pvec_dest, const vector_t* cpvec_src);
```

描述:

使用一个既存的 `vector` 容器来初始化 `vector` 容器。

参数:

`pvec_dest` 指向被初始化的容器的指针。
`cpvec_src` 指向既存容器的指针。

返回值:

无。

注意:

被初始化的容器一定是使用 `create_vector` 创建的容器, 否则函数的行为是未定义的。`cpvec_src` 指向的是已经初始

化的 vector 容器，否则函数的行为是未定义的。pvec_dest == NULL 或者 cpvec_src == NULL，函数的行为是未定义的。两个容器保存的数据类型必须是一致的否则函数的行为是未定义的。初始化之后 vector_size(pvec_dest) == vector_size(cpvec_src); 容量符合前面描述的算法。

```
void vector_init_copy_range(
    vector_t* pvec_dest, vector_iterator_t it_begin, vector_iterator_t it_end);
```

描述:

使用一个数据区间中的数据来初始化 vector 容器。

参数:

pvec_dest	指向被初始化的容器的指针。
it_begin	数据区间的开始。
it_end	数据区间的末尾。

返回值:

无。

注意:

被初始化的容器一定是使用 create_vector 创建的容器，否则函数的行为是未定义的。[it_begin, it_end)属于一个已经初始化的 vector 容器的有效数据区间，否则函数的行为是未定义的。pvec_dest == NULL 函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。初始化之后 vector_size(pvec_dest) == iterator_distance(it_begin, it_end); 容量符合前面描述的算法。

```
void vector_destroy(vector_t* pvec_vector);
```

描述:

销毁创建的 vector 容器。

参数:

pvec_vector	指向被销毁的容器的指针。
-------------	--------------

返回值:

无。

注意:

被销毁的容器一定已经初始化或者是使用 create_vector 创建的容器，否则函数的行为是未定义的。销毁后的容器不能够在用于其他的接口，否则接口的函数行为是未定义的。创建之后没有经过初始化的容器也要进行销毁。

```
size_t vector_size(const vector_t* cpvec_vector);
```

描述:

返回 vector 容器中数据的个数。

参数:

cpvec_vector	指向 vector 容器的指针。
--------------	------------------

返回值:

容器中数据的个数。

注意:

cpvec_vector == NULL 则函数的行为是未定义的，cpvec_vector 指向的容器是经过初始化以后的 vector 容器，否则函数的行为是未定义的。

```
bool_t vector_empty(const vector_t* cpvec_vector);
```

描述:

测试 vector 容器是否为空。

参数:

cpvec_vector 指向 vector 容器的指针。

返回值:

vector 容器为空, 返回 true 否则返回 false。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。

```
size_t vector_max_size(const vector_t* cpvec_vector);
```

描述:

返回 vector 容器中能够保存数据的最大数量的可能值。

参数:

cpvec_vector 指向 vector 容器的指针。

返回值:

返回 vector 容器中能够保存数据的最大数量的可能值。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。这个最大数量是与系统相关的, 不是一个固定的值。

```
size_t vector_capacity(const vector_t* cpvec_vector);
```

描述:

返回 vector 容器的容量。

参数:

cpvec_vector 指向 vector 容器的指针。

返回值:

返回 vector 容器的容量。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。容量符合前面描述的算法。

```
void vector_reserve(vector_t* pvec_vector, size_t t_reserveSize);
```

描述:

指定 vector 容器的容量。

参数:

pvec_vector 指向 vector 容器的指针。

t_reserveSize 修改后的容量。

返回值:

无。

注意:

`pvec_vector == NULL` 则函数的行为是未定义的，`pvec_vector` 指向的容器是经过初始化以后的 `vector` 容器，否则函数的行为是未定义的。如果 `t_reserveSize > vector_capacity()` 则函数执行后，`vector` 容器的容量扩充到 `t_reserveSize`。否则函数执行后 `vector` 的容量不变。`vector` 的容量只能增大不能够减小。

```
bool_t vector_equal(
    const vector_t* cpvec_first, const vector_t* cpvec_second);
```

描述：

测试两个容器是否相等。

参数：

`cpvec_first` 指向 `vector` 容器的指针。

`cpvec_second` 指向 `vector` 容器的指针。

返回值：

如果两个容器相等，返回 `true`，否则返回 `false`。

注意：

`cpvec_first == NULL` 或者 `cpvec_second == NULL` 则函数的行为是未定义的，`cpvec_first` 和 `cpvec_second` 指向的容器是经过初始化以后的 `vector` 容器，否则函数的行为是未定义的。如果两个 `vector` 容器保存的数据类型不同，那么这两个容器不等。如果两个容器中的数据个数相等并且对应数据相等则容器相等，如果 `cpvec_first == cpvec_second` 则返回 `true`。

```
bool_t vector_not_equal(
    const vector_t* cpvec_first, const vector_t* cpvec_second);
```

描述：

测试两个容器是否不相等。

参数：

`cpvec_first` 指向 `vector` 容器的指针。

`cpvec_second` 指向 `vector` 容器的指针。

返回值：

如果两个容器不相等，返回 `true`，否则返回 `false`。

注意：

`cpvec_first == NULL` 或者 `cpvec_second == NULL` 则函数的行为是未定义的，`cpvec_first` 和 `cpvec_second` 指向的容器是经过初始化以后的 `vector` 容器，否则函数的行为是未定义的。如果两个 `vector` 容器保存的数据类型不同，那么这两个容器不等。如果两个容器中的数据个数不等或者对应数据不等则容器不等，如果 `cpvec_first == cpvec_second` 则返回 `false`。

```
bool_t vector_less(
    const vector_t* cpvec_first, const vector_t* cpvec_second);
```

描述：

测试第一个 `vector` 容器是否小于第二个 `vector` 容器。

参数：

`cpvec_first` 指向 `vector` 容器的指针。

`cpvec_second` 指向 `vector` 容器的指针。

返回值：

如果第一个 vector 容器小于第二个 vector 容器，返回 true，否则返回 false。

注意：

cpvec_first == NULL 或者 cpvec_second == NULL 则函数的行为是未定义的，cpvec_first 和 cpvec_second 指向的容器是经过初始化以后的 vector 容器，否则函数的行为是未定义的。如果两个 vector 容器保存的数据类型不同，函数的行为是未定义的。如果第一个容器中的数据小于第二个容器中的对应数据则返回 true，如果大于则返回 false，如果对应数据都相等则，如果第一个容器中的数据个数小于第二个容器中的数据个数则返回 true 否则返回 false，如果 cpvec_first == cpvec_second 则返回 false。

```
bool_t vector_less_equal(
    const vector_t* cpvec_first, const vector_t* cpvec_second);
```

描述：

测试第一个 vector 容器是否小于等于第二个 vector 容器。

参数：

cpvec_first 指向 vector 容器的指针。
cpvec_second 指向 vector 容器的指针。

返回值：

如果第一个 vector 容器小于等于第二个 vector 容器，返回 true，否则返回 false。

注意：

cpvec_first == NULL 或者 cpvec_second == NULL 则函数的行为是未定义的，cpvec_first 和 cpvec_second 指向的容器是经过初始化以后的 vector 容器，否则函数的行为是未定义的。如果两个 vector 容器保存的数据类型不同，函数的行为是未定义的。如果第一个容器中的数据小于第二个容器中的对应数据则返回 true，如果大于则返回 false，如果对应数据都相等则，如果第一个容器中的数据个数小于等于第二个容器中的数据个数则返回 true 否则返回 false，如果 cpvec_first == cpvec_second 则返回 true。

```
bool_t vector_greater(
    const vector_t* cpvec_first, const vector_t* cpvec_second);
```

描述：

测试第一个 vector 容器是否大于第二个 vector 容器。

参数：

cpvec_first 指向 vector 容器的指针。
cpvec_second 指向 vector 容器的指针。

返回值：

如果第一个 vector 容器大于第二个 vector 容器，返回 true，否则返回 false。

注意：

cpvec_first == NULL 或者 cpvec_second == NULL 则函数的行为是未定义的，cpvec_first 和 cpvec_second 指向的容器是经过初始化以后的 vector 容器，否则函数的行为是未定义的。如果两个 vector 容器保存的数据类型不同，函数的行为是未定义的。如果第一个容器中的数据大于第二个容器中的对应数据则返回 true，如果小于则返回 false，如果对应数据都相等则，如果第一个容器中的数据个数大于第二个容器中的数据个数则返回 true 否则返回 false，如果 cpvec_first == cpvec_second 则返回 false。

```
bool_t vector_greater_equal(
    const vector_t* cpvec_first, const vector_t* cpvec_second);
```

描述:

测试第一个 vector 容器是否大于等于第二个 vector 容器。

参数:

cpvec_first 指向 vector 容器的指针。

cpvec_second 指向 vector 容器的指针。

返回值:

如果第一个 vector 容器大于等于第二个 vector 容器, 返回 true, 否则返回 false。

注意:

cpvec_first == NULL 或者 cpvec_second == NULL 则函数的行为是未定义的, cpvec_first 和 cpvec_second 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果两个 vector 容器保存的数据类型不同, 函数的行为是未定义的。如果第一个容器中的数据大于第二个容器中的对应数据则返回 true, 如果小于则返回 false, 如果对应数据都相等则, 如果第一个容器中的数据个数大于等于第二个容器中的数据个数则返回 true 否则返回 false, 如果 cpvec_first == cpvec_second 则返回 true。

```
void vector_assign(vector_t* pvec_dest, const vector_t* cpvec_src);
```

描述:

使用既存 vector 容器为 vector 容器赋值。

参数:

pvec_dest 指向目的 vector 容器的指针。

cpvec_src 指向源 vector 容器的指针。

返回值:

无。

注意:

pvec_dest == NULL 或者 cpvec_src == NULL 则函数的行为是未定义的, pvec_dest 和 cpvec_src 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果两个 vector 容器保存的数据类型不同, 函数的行为是未定义的。赋值以后两个容器的 vector_size() 相等但是 vector_capacity() 不一定相等。如果 vector_equal(pvec_dest, pvec_src) 函数不执行任何操作。

```
void vector_assign_elem(vector_t* pvec_vector, size_t t_count, elem)
```

描述:

使用指定的数据为 vector 容器赋值。

参数:

pvec_vector 指向 vector 容器的指针。

t_count 赋值的数据的个数。

elem 赋值的数据。

返回值:

无。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。elem 是与 vector 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
void vector_assign_range(
```

```
vector_t* pvec_vector, vector_iterator_t it_begin, vector_iterator_t it_end)
```

描述:

使用指定范围的数据为 vector 容器赋值。

参数:

pvec_vector 指向 vector 容器的指针。
it_begin 指定范围的开始。
it_end 指定范围的末尾。

返回值:

无。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围并且保存的数据类型与 vector 容器中保存的数据类型相同, 否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 pvec_vector 则函数的行为是未定义的。

```
void vector_swap(vector_t* pvec_first, vector_t* pvec_second);
```

描述:

交换两个 vector 容器中的数据。

参数:

pvec_first 指向 vector 容器的指针。
pvec_second 指向 vector 容器的指针。

返回值:

无。

注意:

pvec_first == NULL 或者 pvec_second == NULL 则函数的行为是未定义的, pvec_first 和 pvec_second 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果两个 vector 容器保存的数据类型不同, 函数的行为是未定义的。如果 vector_equal(pvec_first, pvec_second) 这函数不执行任何动作。

```
void* vector_at(const vector_t* cpvec_vector, size_t t_pos);
```

描述:

通过下标随机访问 vector 容器中的数据。

参数:

cpvec_vector 指向 vector 容器的指针。
t_pos 要访问的数据在容器中的索引。

返回值:

指向被访问的数据的指针。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果 t_pos 超过了 vector 容器中数据索引的范围, 则函数的行为是未定义的。

```
void* vector_front(const vector_t* cpvec_vector);
```

描述:

访问 vector 中第一个数据。

参数:

cpvec_vector 指向 vector 容器的指针。

返回值:

指向被访问的数据的指针。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果 vector 容器为空则函数的行为是未定义的。

```
void* vector_back(const vector_t* cpvec_vector);
```

描述:

访问 vector 中最后一个数据。

参数:

cpvec_vector 指向 vector 容器的指针。

返回值:

指向被访问的数据的指针。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果 vector 容器为空则函数的行为是未定义的。

```
vector_iterator_t vector_begin(const vector_t* cpvec_vector);
```

描述:

获得指向 vector 容器开头的迭代器。

参数:

cpvec_vector 指向 vector 容器的指针。

返回值:

返回指向 vector 容器开头的迭代器。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果 vector 为空, 则返回值与 vector_end(cpvec_vector)相等。

```
vector_iterator_t vector_end(const vector_t* cpvec_vector);
```

描述:

获得指向 vector 容器末尾的迭代器。

参数:

cpvec_vector 指向 vector 容器的指针。

返回值:

返回指向 vector 容器末尾的迭代器。

注意:

cpvec_vector == NULL 则函数的行为是未定义的, cpvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。

```
vector_iterator_t vector_insert(
```

```
vector_t* pvec_vector, vector_iterator_t it_pos, elem)
```

描述:

向 vector 容器的指定位置插入数据。

参数:

pvec_vector	指向 vector 容器的指针。
it_pos	要插入数据的位置。
elem	要插入的数据。

返回值:

返回指向插入后的数据的迭代器。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。it_pos 是属于 pvec_vector 并且是有效的迭代器, 否则函数的行为是未定义的。elem 是与 vector 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
vector_iterator_t vector_insert_n(  
    vector_t* pvec_vector, vector_iterator_t it_pos, size_t t_count, elem)
```

描述:

向 vector 容器的指定位置插入多个数据。

参数:

pvec_vector	指向 vector 容器的指针。
it_pos	要插入数据的位置。
t_count	插入的数据个数。
elem	要插入的数据。

返回值:

返回指向插入后的数据的迭代器。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。it_pos 是属于 pvec_vector 并且是有效的迭代器, 否则函数的行为是未定义的。elem 是与 vector 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
void vector_insert_range(  
    vector_t* pvec_vector, vector_iterator_t it_pos, vector_iterator_t it_begin,  
    vector_iterator_t it_end);
```

描述:

向 vector 容器的指定位置插入一个数据区间。

参数:

pvec_vector	指向 vector 容器的指针。
it_pos	要插入数据的位置。
it_begin	插入的数据区间的开头。
it_end	插入的数据区间的末尾。

返回值:

无。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。it_pos 是属于 pvec_vector 并且是有效的迭代器, 否则函数的行为是未定义的。[it_begin, it_end) 保存的是与 vector 容器中保存的数据类型兼容的数据, 并且[it_begin, it_end) 必须是有效的数据区间, 否则函数的行为是未定义的。[it_begin, it_end) 属于 vector 时函数的行为是未定义的。

```
void vector_push_back(vector_t* pvec_vector, elem)
```

描述:

向 vector 容器的末尾添加一个数据。

参数:

pvec_vector 指向 vector 容器的指针。

elem 要添加的数据。

返回值:

无。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。elem 是与 vector 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
void vector_pop_back(vector_t* pvec_vector);
```

描述:

删除 vector 容器末尾添的一个数据。

参数:

pvec_vector 指向 vector 容器的指针。

返回值:

无。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。如果 vector 为空, 则函数的行为是未定义的。

```
vector_iterator_t vector_erase(vector_t* pvec_vector, vector_iterator_t it_pos);
```

描述:

删除 vector 容器中指定位置的数据。

参数:

pvec_vector 指向 vector 容器的指针。

it_pos 要删除数据的位置。

返回值:

返回指向被删除数据后面的数据的迭代器。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。it_pos 是属于 vector 的有效迭代器, 否则函数的行为是未定义的。

```
vector_iterator_t vector_erase_range(
```

```
vector_t* pvec_vector, vector_iterator_t it_begin, vector_iterator_t it_end);
```

描述:

删除 vector 容器中指定数据区间的数据。

参数:

pvec_vector 指向 vector 容器的指针。

it_begin 删除的数据区间的开头。

it_end 删除的数据区间的末尾。

返回值:

返回指向被删除数据后面的数据的迭代器。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。[it_begin, it_end)是属于 vector 的有效数据区间, 否则函数的行为是未定义的。

```
void vector_clear(vector_t* pvec_vector);
```

描述:

删除 vector 容器中所有的数据。

参数:

pvec_vector 指向 vector 容器的指针。

返回值:

无。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。

```
void vector_resize(vector_t* pvec_vector, size_t t_resize);
```

描述:

重新设置 vector 容器中数据的个数。

参数:

pvec_vector 指向 vector 容器的指针。

t_resize vector 容器中数据的新个数。

返回值:

无。

注意:

pvec_vector == NULL 则函数的行为是未定义的, pvec_vector 指向的容器是经过初始化以后的 vector 容器, 否则函数的行为是未定义的。当 t_resize > vector_size() 的时候, t_resize - vector_size() 的部分使用容器中保存的数据类型的默认数据填充。

```
void vector_resize_elem(vector_t* pvec_vector, size_t t_resize, elem)
```

描述:

重新设置 vector 容器中数据的个数, 使用 elem 填充。

参数:

pvec_vector 指向 vector 容器的指针。

`t_resize` vector 容器中数据的新的个数。

`elem` 填充的数据。

返回值:

无。

注意:

`pvec_vector == NULL` 则函数的行为是未定义的, `pvec_vector` 指向的容器是经过初始化以后的 `vector` 容器, 否则函数的行为是未定义的。当 `t_resize > vector_size()` 的时候, `t_resize - vector_size()` 的部分使用 `elem` 填充, `elem` 是与 `vector` 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

第五节 迭代器接口

迭代器接口主要是为实现迭代器操作提供的, 这些接口有迭代器的外部接口使用, 用户不应该直接使用这些接口。这些接口是与容器相关的, `vector` 的迭代器接口只能够处理 `vector` 容器的迭代器, 对于其他容器的迭代器无能为力。

<code>_create_vector_iterator</code>	创建 <code>vector</code> 容器的迭代器。
<code>_vector_iterator_equal</code>	测试两个 <code>vector</code> 迭代器是否相等。
<code>_vector_iterator_less</code>	测试第一个 <code>vector</code> 迭代器是否小于第二个 <code>vector</code> 迭代器。
<code>_vector_iterator_before</code>	测试第一个 <code>vector</code> 迭代器是否在第二个 <code>vector</code> 迭代器前面。
<code>_vector_iterator_get_value</code>	获得迭代器引用的数据。
<code>_vector_iterator_set_value</code>	设置迭代器引用的数据。
<code>_vector_iterator_get_pointer</code>	获得指向迭代器引用的数据的指针。
<code>_vector_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_vector_iterator_prev</code>	获得引用上一个数据的迭代器。
<code>_vector_iterator_next_n</code>	获得引用向下第 n 个数据的迭代器。
<code>_vector_iterator_prev_n</code>	获得引用向上第 n 个数据的迭代器。
<code>_vector_iterator_at</code>	通过下标来访问迭代器引用的数据。
<code>_vector_iterator_minus</code>	获得两个迭代器之差。

```
vector_iterator_t _create_vector_iterator(void);
```

描述:

创建 `vector` 迭代器。

参数:

无。

返回值:

`vector` 迭代器。

注意:

获得的 `vector` 迭代器不是有效的迭代器, 因为它并没有与任何 `vector` 容器关联。这个函数创建的 `vector` 迭代器是供

给以后的 vector 函数使用的。

```
bool_t _vector_iterator_equal(
    vector_iterator_t it_first, vector_iterator_t it_second);
```

描述:

比较两个迭代器是否相等。

参数:

it_first 第一个 vector 迭代器。
it_second 第二个 vector 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个迭代器必须属于同一个 vector 容器, 否则函数的行为是未定义的。两个迭代器相等表示的是两个迭代器引用的是同一个容器中的同一个数据。

```
bool_t _vector_iterator_less(
    vector_iterator_t it_first, vector_iterator_t it_second);
```

描述:

测试第一个迭代器是否小于第二个迭代器。

参数:

it_first 第一个 vector 迭代器。
it_second 第二个 vector 迭代器。

返回值:

如果第一个迭代器小于第二个迭代器则返回 true, 否则返回 false。

注意:

两个迭代器必须属于同一个 vector 容器, 否则函数的行为是未定义的。第一个迭代器小于第二个迭代器表示的是第一个迭代器引用的数据比第二个迭代器引用的数据更靠近容器中的第一个数据。

```
bool_t _vector_iterator_before(
    vector_iterator_t it_first, vector_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器前面。

参数:

it_first 第一个 vector 迭代器。
it_second 第二个 vector 迭代器。

返回值:

如果第一个迭代器在第二个迭代器前面则返回 true, 否则返回 false。

注意:

两个迭代器必须属于同一个 vector 容器, 否则函数的行为是未定义的。第一个迭代器在第二个迭代器前面表示的是第一个迭代器引用的数据比第二个迭代器引用的数据更靠近容器中的第一个数据。

```
void _vector_iterator_get_value(vector_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter vector 迭代器。

pv_value 保存数据的缓存区。

返回值:

无。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的, pv_value 的缓冲区要能够保存 it_iter 引用的数据, 否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 中。

```
void _vector_iterator_set_value(vector_iterator_t it_iter, const void* cpv_value);
```

描述:

设置迭代器引用的数据。

参数:

it_iter vector 迭代器。

cpv_value 保存数据的缓存区。

返回值:

无。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。cpv_value == NULL 则函数的行为是未定义的, cpv_value 的缓冲区中保存的数据要与 it_iter 引用的数据类型相同, 否则函数的行为是未定义的。函数执行后 cpv_value 中保存的数据被拷贝到 it_iter 引用的数据中。

```
const void* _vector_iterator_get_pointer(vector_iterator_t it_iter);
```

描述:

返回被迭代器应用的数据的指针。

参数:

it_iter vector 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。

```
vector_iterator_t _vector_iterator_next(vector_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter vector 迭代器。

返回值:

引用下一个数据的迭代器。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是属于 vector 的有效的迭代器, 否则函数行为是未定义的。

```
vector_iterator_t _vector_iterator_prev(vector_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter vector 迭代器。

返回值:

引用上一个数据的迭代器。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是属于 vector 的有效的迭代器, 否则函数行为是未定义的。

```
vector_iterator_t _vector_iterator_next_n(vector_iterator_t it_iter, int n_step);
```

描述:

获得引用向下第 n 个数据的迭代器。

参数:

it_iter vector 迭代器。

n_step 前进的步数。

返回值:

引用向下第 n 个数据的迭代器。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。引用向下第 n 个数据的迭代器 也必须是属于 vector 的有效的迭代器, 否则函数行为是未定义的。n_step > 0 表示迭代器向容器末尾移动 n 步, n_step < 0 表示迭代器向容器开头移动 n 步, n_step == 0 表示不移动。

```
vector_iterator_t _vector_iterator_prev_n(vector_iterator_t it_iter, int n_step);
```

描述:

获得引用向上第 n 个数据的迭代器。

参数:

it_iter vector 迭代器。

n_step 前进的步数。

返回值:

引用向上第 n 个数据的迭代器。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。引用向上第 n 个数据的迭代器 也必须是属于 vector 的有效的迭代器, 否则函数行为是未定义的。n_step > 0 表示迭代器向容器开头移动 n 步, n_step < 0 表示迭代器向容器末尾移动 n 步, n_step == 0 表示不移动。

```
void* _vector_iterator_at(vector_iterator_t it_iter, int n_index);
```

描述:

使用下标随机访问迭代器引用的数据。

参数:

it_iter vector 迭代器。
n_index 访问的下标。

返回值:

获得随机访问的迭代器应用的数据的指针。

注意:

it_iter 必须是属于 vector 的有效迭代器, 否则函数的行为是未定义的。随机访问的数据也必须在 vector 范围内, 否则函数行为是未定义的。n_index > 0 表示访问从当前迭代器引用的数据起向容器末尾的第 n 个数据, n_index < 0 表示从当前迭代器引用的数据起向容器开头的第 n 个数据, n_index == 0 表示当前迭代器引用的数据。

```
int _vector_iterator_minus(  
    vector_iterator_t it_first, vector_iterator_t it_second);
```

描述:

求两个迭代器之间的差。

参数:

it_first 第一个 vector 迭代器。
it_second 第二个 vector 迭代器。

返回值:

返回两个迭代器之间相差的距离。

注意:

it_first 和 it_second 必须是属于同一个 vector 的有效迭代器, 否则函数的行为是未定义的。当 it_first < it_econd 的时候返回值小于 0, it_first == it_second 时返回值等于 0, it_first > it_second 时返回值大于 0。

第六节 内部和辅助接口

vector 内部接口是为了实现 vector 容器以及外部接口提供的, 用户不应该直接使用这些接口。

_create_vector	创建一个 vector 容器。
_create_vector_auxiliary	创建 vector 容器的辅助函数。
_vector_init_elem	使用用户指定的数据初始化 vector 容器。
_vector_init_elem_varg	使用可变参数列表中的数据初始化 vector 容器。
_vector_destroy_auxiliary	销毁 vector 容器的辅助函数。
_vector_assign_elem	为 vector 容器赋值。
_vector_assign_elem_varg	为 vector 容器赋值, 数据来自于可变参数列表。
_vector_push_back	向 vector 容器末尾添加一个数据。
_vector_push_back_varg	向 vector 容器末尾添加一个数据, 数据来自于可变参数列表。
_vector_resize_elem	重新设置 vector 中数据的个数, 使用指定的数据填充。

<code>_vector_resize_elem_varg</code>	重新设置 vector 中数据的个数，使用可变参数列表中的数据填充。
<code>_vector_insert_n</code>	向 vector 容器的指定位置插入多个数据。
<code>_vector_insert_n_varg</code>	向 vector 容器的指定位置插入多个来自于可变参数的数据。
<code>_vector_init_elem_auxiliary</code>	初始化数据的辅助函数。

```
vector_t* _create_vector(const char* s_typename);
```

描述:

创建一个 vector 容器。

参数:

`s_typename` vector 容器中保存的数据类型。

返回值:

创建成功返回指向容器的指针，否则返回 NULL。

注意:

如果 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型，libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
bool_t _create_vector_auxiliary(vector_t* pvec_vector, const char* s_typename);
```

描述:

创建一个 vector 容器的辅助函数。

参数:

`pvec_vector` 没有创建的 vector 容器。

`s_typename` vector 容器中保存的数据类型。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pvec_vector == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型，libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
void _vector_init_elem(vector_t* pvec_vector, size_t t_count, ...);
```

描述:

使用用户指定的数据初始化 vector 容器。

参数:

`pvec_vector` 未初始化的 vector 容器。

`t_count` 数据个数。

`...` 用户指定的数据。

返回值:

无。

注意:

如果 `pvec_vector == NULL` 则函数的行为是未定义的。vector 容器必须是使用 `create_vector` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _vector_init_elem_varg(
    vector_t* pvec_vector, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据初始化 vector 容器。

参数:

pvec_vector 未初始化的 vector 容器。
t_count 数据个数。
val_elemlist 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 pvec_vector == NULL 则函数的行为是未定义的。vector 容器必须是使用 create_vector 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _vector_destroy_auxiliary(vector_t* pvec_vector);
```

描述:

销毁 vector 容器的辅助函数。

参数:

pvec_vector vector 容器。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 vector 不是使用 create_vector 生成的则函数的行为是未定义的。

```
void _vector_assign_elem(vector_t* pvec_vector, size_t t_count, ...);
```

描述:

使用用户指定的数据为 vector 容器赋值。

参数:

pvec_vector vector 容器。
t_count 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _vector_assign_elem_varg(
    vector_t* pvec_vector, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据为 vector 容器赋值。

参数:

pvec_vector 的 vector 容器。
t_count 数据个数。
val_elemlist 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _vector_push_back(vector_t* pvec_vector, ...);
```

描述:

向 vector 容器末尾添加一个数据。

参数:

pvec_vector vector 容器。
... 用户指定的数据。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _vector_push_back_varg(vector_t* pvec_vector, va_list val_elemlist);
```

描述:

向 vector 容器末尾添加一个数据。

参数:

pvec_vector vector 容器。
val_elemlist 可变参数列表。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _vector_resize_elem(vector_t* pvec_vector, size_t t_resize, ...);
```

描述:

重新设置 vector 容器中的数据个数，不足的部分使用指定数据填充。

参数:

pvec_vector vector 容器。
t_resize 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用指定的数据填充。

```
void _vector_resize_elem_varg(  
    vector_t* pvec_vector, size_t t_resize, va_list val_elemlist);
```

描述:

重新设置 vector 容器中的数据个数，不足的部分使用可变参数列表中的数据填充。

参数:

pvec_vector 的 vector 容器。
t_resize 数据个数。
val_elemlist 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用可变参数列表中的数据填充。

```
vector_iterator_t _vector_insert_n(  
    vector_t* pvec_vector, vector_iterator_t it_pos, size_t t_count, ...);
```

描述:

向 vector 容器的指定位置插入多个数据。

参数:

pvec_vector vector 容器。
it_pos 插入数据的位置。
t_count 数据个数。
... 用户指定的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。it_pos 必须是属于 vector 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
vector_iterator_t _vector_insert_n_varg(  
    vector_t* pvec_vector, vector_iterator_t it_pos, size_t t_count,  
    va_list val_elemlist);
```

描述:

向 vector 容器的指定位置插入多个数据。

参数:

pvec_vector	vector 容器。
it_pos	插入数据的位置。
t_count	数据个数。
val_elemlist	用户指定的数据可变参数列表。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 pvec_vector == NULL 或者 vector 是未初始化的 vector 则函数的行为是未定义的。it_pos 必须是属于 vector 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _vector_init_elem_auxiliary(vector_t* pvec_vector, void* pv_value);
```

描述:

使用 vector 的数据类型对数据进行初始化。

参数:

pvec_vector	vector 容器。
pv_value	数据。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 pv_value == NULL 则函数的行为是未定义的。vector 必须是已经初始化或者是使用 create_vector 创建的 vector 容器，否则函数的行为是未定义的。

vector 辅助接口是为了实现 vector 容器以及外部接口提供的，用户不应该直接使用这些接口。

_vector_iterator_belong_to_vector	测试一个迭代器是否属于一个容器的范围。
_vector_same_type	测试两个 vector 是否保存同样类型的数据。
_vector_same_vector_iterator_type	测试 vector 中保存的数据类型是否和制定的迭代器引用的数据类型相同。
_vector_get_varg_value_auxiliary	从可变参数列表中获得与 vector 中的数据类型相同的数据。
_vector_destroy_varg_value_auxiliary	销毁与 vector 中的数据类型相同的数据。
_vector_init_elem_range_auxiliary	将制定范围内的数据初始化为 vector 中的数据类型。
_vector_is_created	测试 vector 是否被创建。
_vector_is_inited	测试 vector 是否被初始化。

`_vector_calculate_new_capacity`

计算 vector 容器的新容量。

```
bool_t _vector_iterator_belong_to_vector(
    const vector_t* cpvec_vector, vector_iterator_t it_iter);
```

描述:

判断一个迭代器引用的数据是否属于制定的容器。

参数:

cpvec_vector vector 容器。
it_iter vector 迭代器。

返回值:

如果迭代器引用的数据属于 vector 容器的有效范围内，返回 true，否则返回 false。

注意:

如果 cpvec_vector == NULL，那么函数的行为是未定义的，如果 it_iter 不是 vector 迭代器，那么函数的行为是未定义的，如果 it_iter 不属于容器 vector，那么函数的行为是未定义的。

```
bool_t _vector_same_type(const vector_t* cpvec_first, const vector_t* cpvec_second);
```

描述:

判断两个 vector 容器中保存的数据类型是否相同。

参数:

cpvec_first 第一个 vector 容器。
cpvec_second 第二个 vector 容器。

返回值:

如果两个 vector 中保存的数据类型相同返回 true，否则返回 false。

注意:

如果 cpvec_first == NULL 或者 cpvec_second == NULL，那么函数的行为是未定义的。两个 vector 容器必须是已经初始化或者使用 create_vector 创建的 vector 容器，否则函数的行为是未定义的。如果 cpvec_first == cpvec_second 则返回 true。

```
bool_t _vector_same_vector_iterator_type(
    const vector_t* cpvec_vector, vector_iterator_t it_iter);
```

描述:

判断 vector 容器中保存的数据类型与迭代器引用的数据类型是否相同。

参数:

cpvec_vector vector 容器。
it_iter vector 迭代器。

返回值:

如果 vector 中保存的数据类型与 it_iter 引用的数据类型相同返回 true，否则返回 false。

注意:

如果 cpvec_vector == NULL 或者 it_iter 不是 vector_iterator_t 类型，那么函数的行为是未定义的。

```
void _vector_get_varg_value_auxiliary(
    vector_t* pvec_vector, va_list val_elemlist, void* pv_varg);
```

描述:

从可变参数列表中获得与 vector 中的数据类型相同的数据。

参数:

pvec_vector	vector 容器。
val_elemlist	可变参数列表。
pv_varg	保存数据缓的缓冲区。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 pv_varg == NULL, 那么函数的行为是未定义的。pvec_vector 必须是已经初始化或者是由 create_vector() 创建的 vector 容器, 否则函数的行为是未定义的。

```
void _vector_destroy_varg_value_auxiliary(
    vector_t* pvec_vector, void* pv_varg);
```

描述:

销毁与 vector 中的数据类型相同的数据。

参数:

pvec_vector	vector 容器。
pv_varg	保存数据的缓冲区。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 pv_varg == NULL, 那么函数的行为是未定义的。pvec_vector 必须是已经初始化或者是由 create_vector() 创建的 vector 容器, 否则函数的行为是未定义的。

```
void _vector_init_elem_range_auxiliary(
    vector_t* pvec_vector, _byte_t* pby_start, _byte_* pby_finish);
```

描述:

将制定范围内的数据初始化为 vector 中的数据类型。

参数:

pvec_vector	vector 容器。
pby_start	初始化范围的开始。
pby_finish	初始化范围的末尾。

返回值:

无。

注意:

如果 pvec_vector == NULL 或者 pby_start == NULL 或者 pby_finish == NULL, 那么函数的行为是未定义的。如果 pby_finish < pby_start 则函数的行为未定义。pvec_vector 必须是初始化的或者是使用 create_vector 创建的, 否则函数的行为是未定义的。

```
bool_t _vector_is_created(const vector_t* cpvec_vector);
```

描述:

测试 vector 是否是由 create_vector 创建的。

参数:

cpvec_vector vector 容器。

返回值:

如果 vector 是由 create_vector 创建的则返回 true, 否则返回 false。

注意:

如果 cpvec_vector == NULL 那么函数的行为是未定义的。

```
bool_t _vector_is_inited(const vector_t* cpvec_vector);
```

描述:

测试 vector 是否被初始化。

参数:

cpvec_vector vector 容器。

返回值:

如果 vector 是被初始化了则返回 true, 否则返回 false。

注意:

如果 cpvec_vector == NULL 那么函数的行为是未定义的。

```
size_t _vector_calculate_new_capacity(size_t t_oldsize, size_t t_insertsize);
```

描述:

计算 vector 容器的新容量。

参数:

t_oldsize vector 容器包含数据的个数。

t_insertsize 插入数据的个数。

返回值:

返回新的 vector 容量。

第七章 list 容器

list 容器是一个链表容器，并且是双向链表。在 list 的中间和末尾插入和删除数据花费常数时间，但是访问数据要花费线性时间。list 的每一个数据都保存在单独的节点当中。

第一节 list 容器的机制

list 容器是一个双向的链表，每一个数据都保存在单独的节点中，并且这个链表是一个循环的链表：

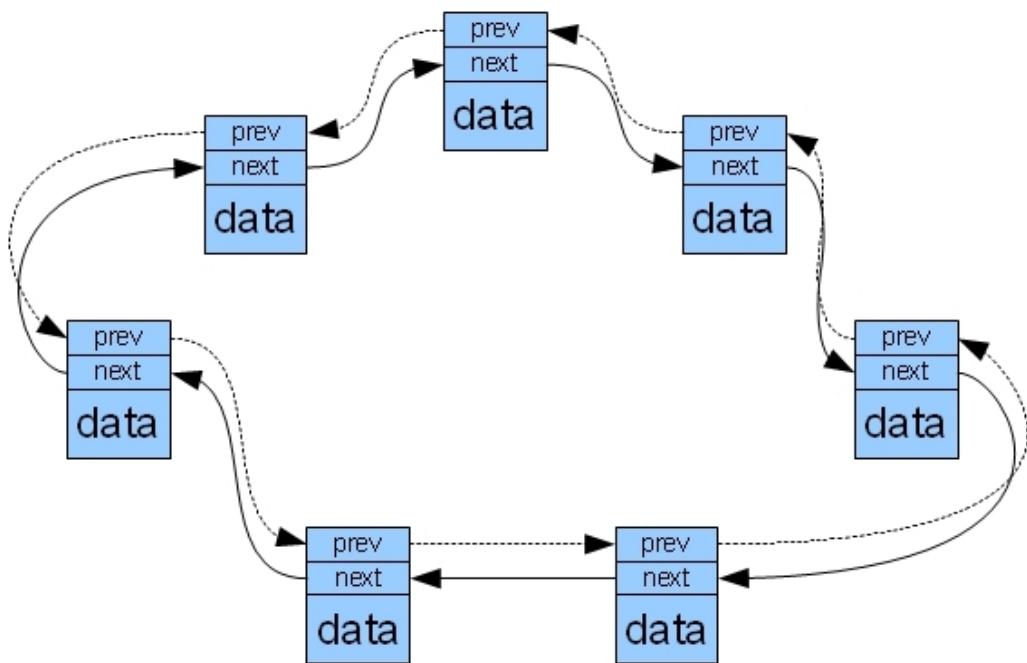


图 7.1 list 容器的结构

这样的结构使得在 list 容器中插入和删除数据花费常数的时间，但是要查找一个数据就要花费线性时间。但是现在的结构并没有区分出 list 的头和尾，为了区分 list 的头和尾以及处理空链表的方式我们在链表中引用了一个空白的节点这个空白节点不保存任何数据，它只表示的是链表末尾的下一个位置以及链表头的前一个位置，这样 list 的结构就变成了：

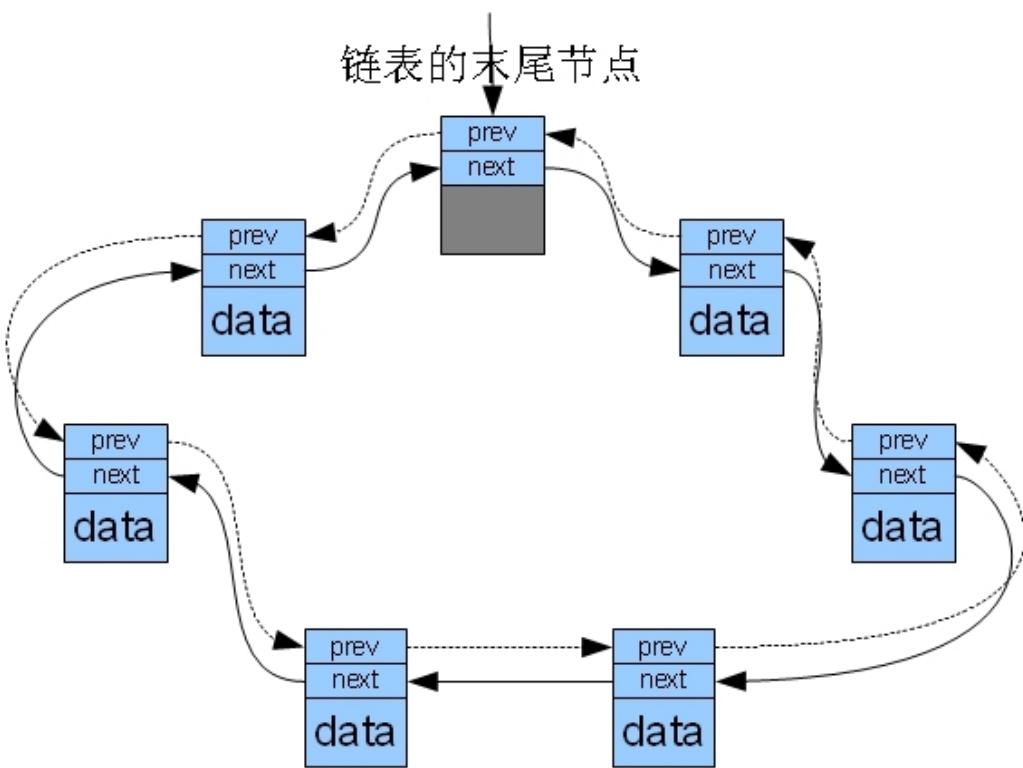


图 7.2 list 容器的结构

当 list 容器为空的时候就只有一个空白节点:

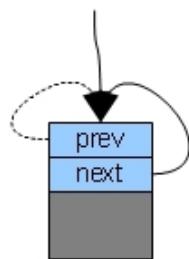
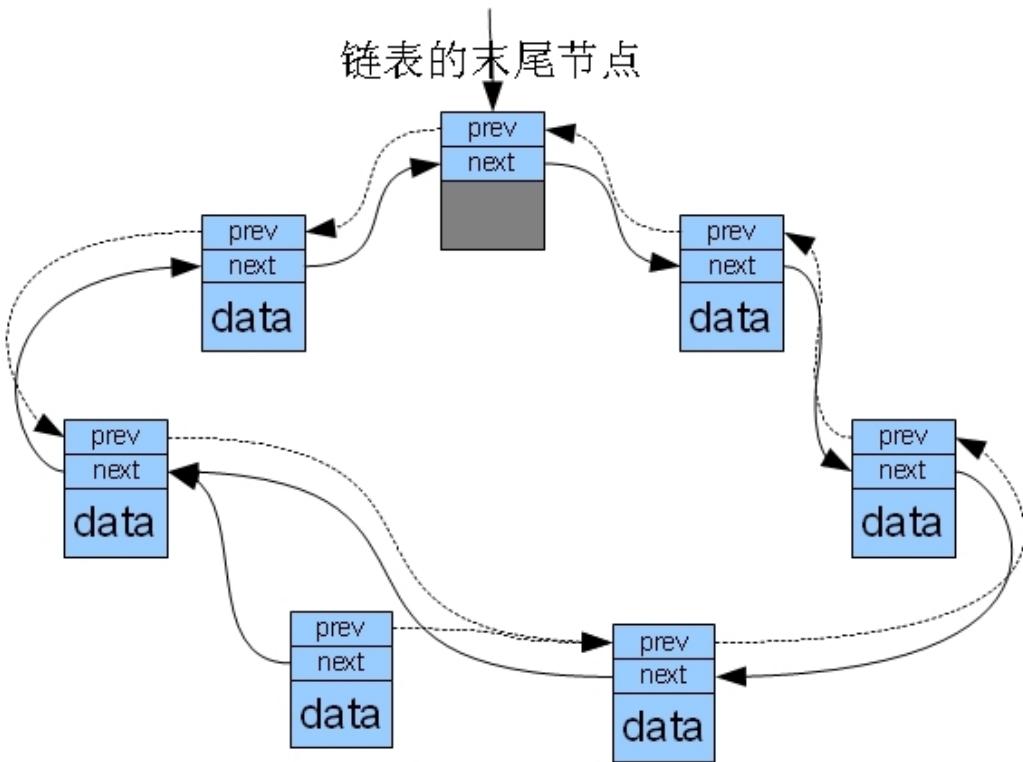


图 7.3 空 list 是的情况

向链表中任何位置插入数据对于链表的修改都是相同的，插入数据时只需要修改指针就可以了。同样删除数据的时候也需要修改指针就可以了：



插入数据 删除数据

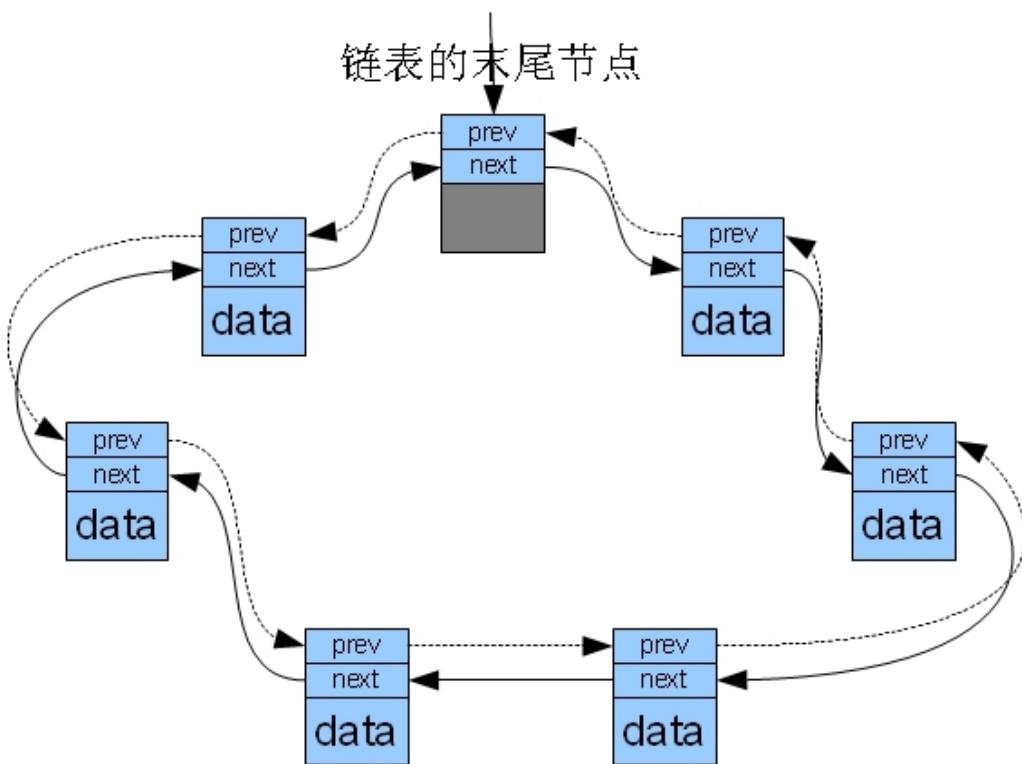


图 7.4 向 list 中插入数据

list 容器除了插入和删除操作意外还有排序操作，对于 list 的排序使用的是快速排序算法。快速排序的关键点在于枢组的选择，list 快速排序选择最后一个数据为枢组。

第二节 list 迭代器

由于引入了空白节点，所以 list 的迭代器也是很好实现的。同时由于 list 的迭代器是双向迭代器所以它只能每一次向前或者向后移动一个位置：

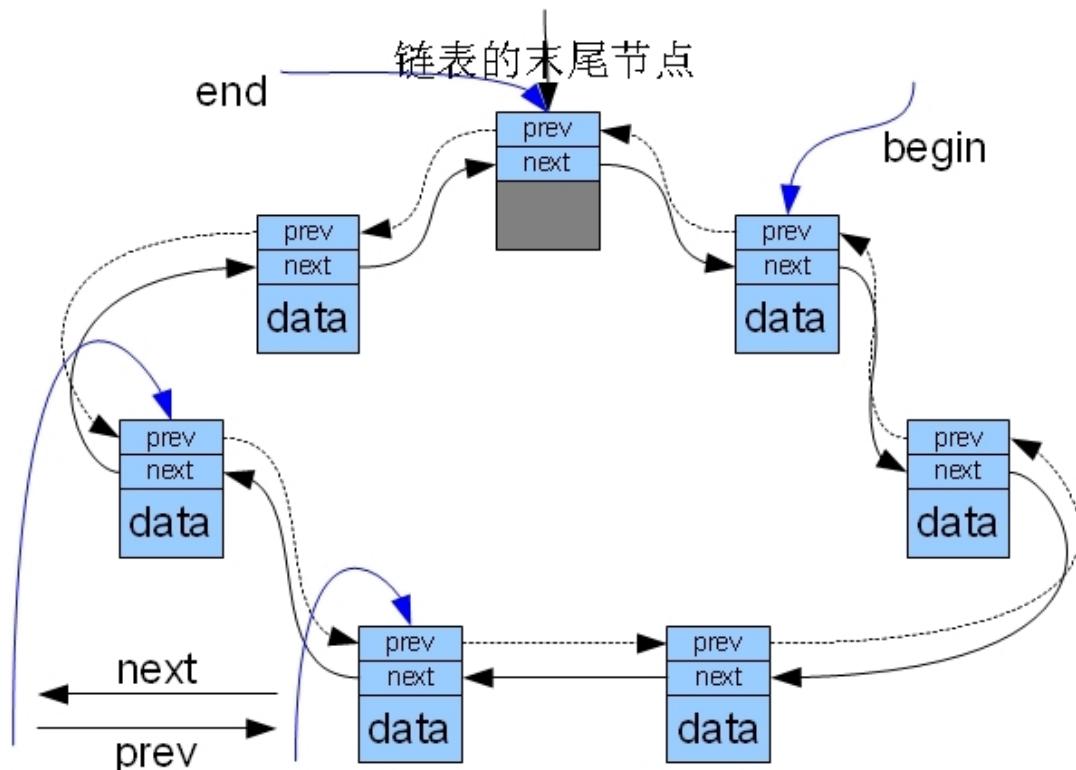


图 7.5 list 的迭代器操作

这样在迭代器中也不需要太多的结构来表示，所以 list 的迭代器结构是：

```
typedef struct _tagiterator
{
    /* 容器内部结构的信息 */
    union
    {
        char*      _pc_corepos;      /* 链表容器中数据的位置 */
        }_t_pos;
        void*      _pt_container;    /* 容器的位置 */
        containertype_t _t_containertype; /* 容器的类型 */
        iteratortype_t _t_iteratortype; /* 迭代器的类型 */
```

```
}iterator_t;
```

第三节 list 的代码结构

list 的代码结构分为两部分，第一部分是 list 节点的结构：

```
typedef struct _taglistnode
{
    struct _taglistnode* _pt_prev;      /* 前一个节点 */
    struct _taglistnode* _pt_next;      /* 后一个节点 */
    _byte_t             _pby_data[1];   /* 保存的数据 */
} listnode_t;
```

这个结构中有一个_pby_data[1]，这是因为保存在节点中的类型是可变的，我们不知道它具体啊的大小是多少，首先定义一个占位符然后，分配节点的时候根据保存的数据类型的大小决定节点的大小。

第二部分是 list 容器的结构：

```
typedef struct _taglist
{
    _typeinfo_t  _t_typeinfo;
    _alloc_t     _t_allocator;
    _listnode_t* _pt_node;        /* 指向空白节点 */
} list_t;
```

实际的代码结构是这样的：

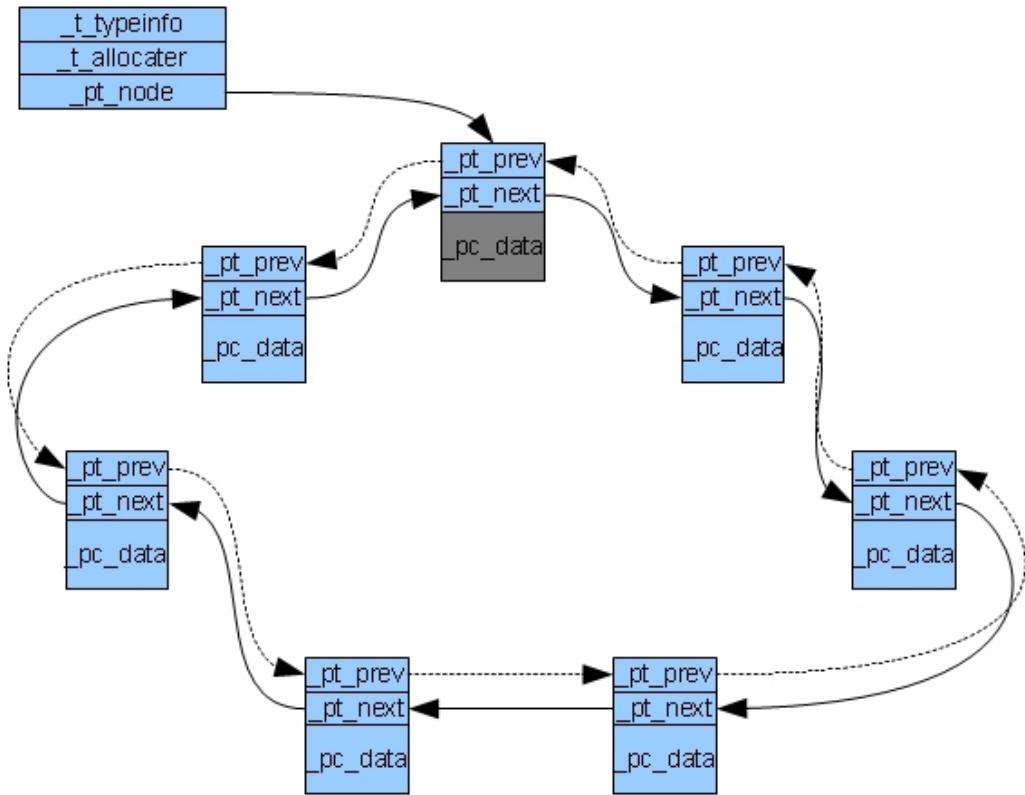


图 7.6 list 容器实际的代码结构

第四节 外部接口

list 容器要提供给用户外部接口：

<code>create_list</code>	创建 list 容器。
<code>list_init</code>	初始化一个空的 list 容器。
<code>list_init_n</code>	使用 n 个默认数据初始化 list 容器。
<code>list_init_elem</code>	使用 n 个指定数据初始化 list 容器。
<code>list_init_copy</code>	使用一个既存的 list 容器初始化另一个 list 容器。
<code>list_init_copy_range</code>	使用指定范围内的数据初始化一个 list 容器。
<code>list_destroy</code>	销毁创建的 list 容器。
<code>list_size</code>	获得 list 容器中数据的数量。
<code>list_empty</code>	测试 list 容器是否为空。
<code>list_max_size</code>	返回 list 容器中保存数据的最大数量的可能值。
<code>list_equal</code>	测试两个 list 容器是否相等。

list_not_equal	测试两个 list 容器是否不等。
list_less	测试第一个 list 容器是否小于第二个 list 容器。
list_less_equal	测试第一个 list 容器是否小于等于第二个 list 容器。
list_greater	测试第一个 list 容器是否大于第二个 list 容器。
list_greater_equal	测试第一个 list 容器是否大于等于第二个 list 容器。
list_assign	使用一个既存的 list 容器为另一个 list 容器赋值。
list_assign_elem	使用指定的数据为 list 容器赋值。
list_assign_range	使用指定范围内的数据为 list 容器赋值。
list_swap	交换两个 list 容器。
list_front	访问 list 容器中的第一个数据。
list_back	访问 list 容器中的最后一个数据。
list_begin	返回引用 list 容器的第一个数据的迭代器。
list_end	返回引用 list 容器末尾的迭代器。
list_insert	向 list 容器中插入数据。
list_insert_n	向 list 容器中插入 n 个数据。
list_insert_range	向 list 容器中插入指定范围的数据。
list_push_back	向 list 容器的末尾添加一个数据。
list_push_front	向 list 容器的开头添加一个数据。
list_pop_back	删除 list 容器末尾的一个数据。
list_pop_front	删除 list 容器开头的一个数据。
list_erase	删除 list 容器中指定位置的一个数据。
list_erase_range	删除 list 容器中指定范围内的数据。
list_remove	删除 list 容器中指定的数据。
list_remove_if	删除 list 容器中符合条件的数据。
list_resize	重新设置 list 容器中数据的数量。
list_resize_elem	重新设置 list 容器中数据的数量，多出的数据使用指定数据填充。
list_clear	删除 list 容器中所有的数据。
list_unique	删除 list 容器中相邻且相等的数据。
list_unique_if	删除 list 容器中相邻且符合条件的数据。
list_splice	将既存 list 容器中的数据转移到另一个 list 容器的指定位置。
list_splice_pos	将既存的 list 容器中指定位置的数据转移到另一个 list 容器的指定位置。
list_splice_range	将既存 list 容器中指定范围的数据转移到另一个 list 容器中的指定位置。

list_sort	将 list 容器中的数据排序。
list_sort_if	将 list 容器中的数据安装指定的规则排序。
list_merge	将两个 list 容器合并。
list_merge_if	将两个 list 容器安装指定规则合并。
list_reverse	将 list 容器逆序。

函数原型

```
list_t* create_list(typename);
```

描述:

创建 list 容器。

参数:

typename 类型描述。

返回值:

返回创建的 list 容器的指针。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void list_init(list_t* plist_list);
```

描述:

初始化一个空的 list 容器。

参数:

plist_list 指向 list 容器的指针。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的，list 必须是通过 create_list 创建的否则函数的行为是未定义的。

```
void list_init_n(list_t* plist_list, size_t t_count);
```

描述:

使用多个数据默认初始化 list 容器。

参数:

plist_list 指向 list 容器的指针。

t_count list 容器中包含的数据个数。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的，list 必须是通过 create_list 创建的否则函数的行为是未定义的。初始化后容器中包含的数据都是 list 容器中保存的数据类型的默认值。

```
void list_init_elem(list_t* plist_list, size_t t_count, elem);
```

描述:

使用多个指定的数据初始化 list 容器。

参数:

- | | |
|------------|------------------|
| plist_list | 指向 list 容器的指针。 |
| t_count | list 容器中包含的数据个数。 |
| elem | 初始化容器的数据。 |

返回值:

无。

注意:

如果 `plist_list == NULL` 则函数的行为是未定义的, `list` 必须是通过 `create_list` 创建的否则函数的行为是未定义的。`elem` 是与 `list` 容器中保存的数据类型相同的数据, 否则函数的行为是未定义的。

```
void list_init_copy(list_t* plist_dest, const list_t* cplist_src);
```

描述:

使用既存的 `list` 容器初始化另一个 `list` 容器。

参数:

- | | |
|------------|---------------------------------|
| plist_dest | 指向被初始化 <code>list</code> 容器的指针。 |
| cplist_src | 指向既存的 <code>list</code> 容器的指针。 |

返回值:

无。

注意:

如果 `plist_list == NULL` 则函数的行为是未定义的, `list` 必须是通过 `create_list` 创建的否则函数的行为是未定义的。既存的 `list` 是有效的 `list` 否则函数的行为是未定义的。既存的 `list` 要与当前的 `list` 保存的数据类型相同, 否则函数的行为是未定义的。初始化之后两个容器中的数据相等。

```
void list_init_copy_range(  
    list_t* plist_dest, list_iterator_t it_begin, list_iterator_t it_end);
```

描述:

使用指定的数据区间初始化另一个 `list` 容器。

参数:

- | | |
|-----------|---------------------------------|
| plis_dest | 指向被初始化 <code>list</code> 容器的指针。 |
| it_begin | 指定的数据区间的开始。 |
| it_end | 指定的数据区间的末尾。 |

返回值:

无。

注意:

如果 `plist_dest == NULL` 则函数的行为是未定义的, `list` 必须是通过 `create_list` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 是有效的数据区间, 否则函数的行为是未定义的。数据区间中保存的数据是与 `list` 中保存的数据类型相同, 否则函数的行为是未定义的。

```
void list_destroy(list_t* plist_list);
```

描述:

销毁一个被创建出来的 list 容器。

参数:

plist_list 指向 list 容器的指针。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 必须是通过 create_list 创建的否则函数的行为是未定义的。不管 list 容器是否被初始化都需要调用这个函数销毁。

```
size_t list_size(const list_t* cplist_list);
```

描述:

返回 list 容器中保存的数据的个数。

参数:

cplist_list 指向 list 容器的指针。

返回值:

list 容器中保存的数据的个数。

注意:

如果 cplist_list == NULL 则函数的行为是未定义的, list 必须已经初始化的, 否则函数的行为是未定义的。

```
bool_t list_empty(const list_t* cplist_list);
```

描述:

测试 list 容器是否为空。

参数:

cplist_list 指向 list 容器的指针。

返回值:

如果 list 容器为空返回 true, 否则返回 false。

注意:

如果 cplist_list == NULL 则函数的行为是未定义的, list 必须已经初始化的, 否则函数的行为是未定义的。

```
size_t list_max_size(const list_t* cplist_list);
```

描述:

返回 list 容器中能够保存的数据最大数量的可能值。

参数:

cplist_list 指向 list 容器的指针。

返回值:

返回 list 容器中能够保存的数据最大数量的可能值。

注意:

如果 cplist_list == NULL 则函数的行为是未定义的, list 必须已经初始化的, 否则函数的行为是未定义的。这个返回值不是一个固定的值。

```
bool_t list_equal(const list_t* cplist_first, const list_t* cplist_second);
```

描述:

测试两个 list 容器是否相等。

参数:

- cplist_first 指向 list 容器的指针。
- cplist_second 指向 list 容器的指针。

返回值:

如果两个 list 容器相等则返回 true, 否则返回 false。

注意:

如果 cplist_first == NULL 或者 cplist_second == NULL 则函数的行为是未定义的, 两个 list 必须已经初始化的, 否则函数的行为是未定义的。两个 list 容器中保存的数据类型不同认为这两个容器不等。两个 list 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cplist_first == cplist_second 那么返回 true。

```
bool_t list_not_equal(const list_t* cplist_first, const list_t* cplist_second);
```

描述:

测试两个 list 容器是否不等。

参数:

- cplist_first 指向 list 容器的指针。
- cplist_second 指向 list 容器的指针。

返回值:

如果两个 list 容器不等则返回 true, 否则返回 false。

注意:

如果 cplist_first == NULL 或者 cplist_second == NULL 则函数的行为是未定义的, 两个 list 必须已经初始化的, 否则函数的行为是未定义的。两个 list 容器中保存的数据类型不同认为两个 list 容器不等。两个 list 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cplist_first == cplist_second 那么返回 false。

```
bool_t list_less(const list_t* cplist_first, const list_t* cplist_second);
```

描述:

测试第一个 list 是否小于第二个 list。

参数:

- cplist_first 指向 list 容器的指针。
- cplist_second 指向 list 容器的指针。

返回值:

如果第一个 list 容器小于第二个 list 容器则返回 true, 否则返回 false。

注意:

如果 cplist_first == NULL 或者 cplist_second == NULL 则函数的行为是未定义的, 两个 list 必须已经初始化的, 否则函数的行为是未定义的。两个 list 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 list 中的数据小于第二个 list 中对应的数据则返回 true, 如果大于则返回 false, 当两个 list 中的对应数据相等, 第一个 list 中的数据数量小于第二个 list 中数据的数量返回 true, 否则返回 false。如果 cplist_first == cplist_second 那么返回 false。

```
bool_t list_less_equal(const list_t* cplist_first, const list_t* cplist_second);
```

描述:

测试第一个 list 是否小于等于第二个 list。

参数:

- cplist_first 指向 list 容器的指针。
- cplist_second 指向 list 容器的指针。

返回值:

如果第一个 list 容器小于等于第二个 list 容器则返回 true, 否则返回 false。

注意:

如果 cplist_first == NULL 或者 cplist_second == NULL 则函数的行为是未定义的, 两个 list 必须已经初始化的, 否则函数的行为是未定义的。两个 list 容器中保存的数据类型不同函数的行为是未定义的。如果 cplist_first == cplist_second 那么返回 true。

```
bool_t list_greater(const list_t* cplist_first, const list_t* cplist_second);
```

描述:

测试第一个 list 是否大于第二个 list。

参数:

- cplist_first 指向 list 容器的指针。
- cplist_second 指向 list 容器的指针。

返回值:

如果第一个 list 容器大于第二个 list 容器则返回 true, 否则返回 false。

注意:

如果 cplist_first == NULL 或者 cplist_second == NULL 则函数的行为是未定义的, 两个 list 必须已经初始化的, 否则函数的行为是未定义的。两个 list 容器中保存的数据类型不同函数的行为是未定义的。如果第一 list 中的数据大于第二个 list 中的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 list 中数据的数量大于第二个 list 中数据的数量的时候返回 true 否则返回 false。如果 cplistt_first == cplist_second 那么返回 false。

```
bool_t list_greater_equal(  
    const list_t* cplist_first, const list_t* cplist_second);
```

描述:

测试第一个 list 是否大于等于第二个 list。

参数:

- cplist_first 指向 list 容器的指针。
- cplist_second 指向 list 容器的指针。

返回值:

如果第一个 list 容器大于等于第二个 list 容器则返回 true, 否则返回 false。

注意:

如果 cplist_first == NULL 或者 cpt_listsecond == NULL 则函数的行为是未定义的, 两个 list 必须已经初始化的, 否则函数的行为是未定义的。两个 list 容器中保存的数据类型不同函数的行为是未定义的。如果 cplist_first == cplist_second 那么返回 true。

```
void list_assign(list_t* plist_dest, const list_t* cplist_src);
```

描述:

使用一个 list 为另一个 list 赋值。

参数:

plist_dest 指向被赋值的 list 容器的指针。

cplist_src 指向赋值的 list 容器的指针。

返回值:

无。

注意:

如果 plist_dest == NULL 或者 cplist_src == NULL 则函数的行为是未定义的，两个 list 必须已经初始化的，否则函数的行为是未定义的。两个 list 容器中保存的数据类型不同函数的行为是未定义的。如果 list_equal(plist_dest, cplist_src) 那么函数不做任何动作。

```
void list_assign_elem(list_t* plist_list, size_t t_count, elem);
```

描述:

使用指定的数据为 list 赋值。

参数:

plist_list 指向被赋值的 list 容器的指针。

t_count 赋值的数据的个数。

elem 赋值的数据。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。elem 是与 list 中保存的数据类型相同的数据，否则函数的行为是未定义的。

```
void list_assign_range(
    list_t* plist_list, list_iterator_t it_begin, list_iterator_t it_end);
```

描述:

使用指定的数据区间为 list 赋值。

参数:

plist_list 指向被赋值的 list 容器的指针。

it_begin 指定的数据区间的开始。

it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end) 是有效的数据区间否则函数的行为是未定义的。[it_begin, it_end) 中保存的数据是与 list 中保存的数据类型相同的数据，否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 plist_list 则函数的行为是未定义的。

```
void list_swap(list_t* plist_first, list_t* plist_second);
```

描述:

交换两个 list 容器。

参数:

plist_first 指向 list 容器的指针。

plist_second 指向 list 容器的指针。

返回值:

无。

注意:

如果 plist_first == NULL 或者 plist_second == NULL 则函数的行为是未定义的，两个 list 容器必须已经初始化的，否则函数的行为是未定义的。两个 list 容器中保存的数据类型必须是一致的，否则函数的行为是未定义的。如果 list_equal(plist_firt, plist_second)则函数不执行任何动作。

```
void* list_front(const list_t* cplist_list);
```

描述:

访问 list 容器中的第一个数据

参数:

cplist_list 指向 list 容器的指针。

返回值:

指向 list 容器中第一个数据的指针。

注意:

如果 cplist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。如果 list 为空则函数的行为是未定义的。

```
void* list_back(const list_t* cplist_list);
```

描述:

访问 list 容器中的最后一个数据

参数:

cplist_list 指向 list 容器的指针。

返回值:

指向 list 容器中最后一个数据的指针。

注意:

如果 cplist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。如果 list 为空则函数的行为是未定义的。

```
list_iterator_t list_begin(const list_t* cplist_list);
```

描述:

获得引用 list 容器中第一数据的迭代器。

参数:

cplist_list 指向 list 容器的指针。

返回值:

返回引用 list 容器中第一个数据的迭代器。

注意:

如果 cplist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。如果 list 容器为空，则返回值与 list_end(cplist_list)相等。

```
list_iterator_t list_end(const list_t* cplist_list);
```

描述:

获得引用 list 容器末尾的迭代器。

参数:

cplist_list 指向 list 容器的指针。

返回值:

返回引用 list 容器末尾的迭代器。

注意:

如果 cplist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。

```
list_iterator_t list_insert(list_t* plist_list, list_iterator_t it_pos, elem);
```

描述:

在 list 容器的指定位置插入数据。

参数:

plist_list 指向 list 容器的指针。

it_pos 插入数据的位置。

elem 被插入的数据。

返回值:

返回引用被插入的数据的迭代器。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 必须是属于 plist_list 容器的是有效迭代器, 否则函数的行为是未定义的。elem 是与 list 容器中保存的数据类型一致的数据, 否则函数的行为是未定义的。

```
list_iterator_t list_insert_n(
    list_t* plist_list, list_iterator_t it_pos, size_t t_count, elem);
```

描述:

在 list 容器的指定位置插入多个数据。

参数:

plist_list 指向 list 容器的指针。

it_pos 插入数据的位置。

t_count 插入的数据的个数。

elem 被插入的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 必须是属于 pt_list 容器的是有效迭代器, 否则函数的行为是未定义的。elem 是与 list 容器中保存的数据类型一致的数据, 否则函数的行为是未定义的。

```
void list_insert_range(
    list_t* plist_list, list_iterator_t it_pos,
    list_iterator_t it_begin, list_iterator_t it_end);
```

描述:

在 list 容器的指定位置插入一个数据区间。

参数:

plist_list 指向 list 容器的指针。
it_pos 插入数据的位置。
it_begin 插入的数据区间的开头。
it_end 插入的数据区间的末尾。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 必须是属于 plist_list 容器的是有效迭代器, 否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间, 否则函数的行为是未定义的, [it_begin, it_end)中保存的数据类型与 list 中保存的数据类型一致, 否则函数的行为是未定义的。如果[it_begin, it_end)属于 plist_list 则函数的行为是未定义的。

```
void list_push_back(list_t* plist_list, elem);
```

描述:

向 list 末尾添加数据。

参数:

plist_list 指向 list 容器的指针。
elem 添加的数据。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。elem 是与 list 容器中保存的数据类型一致的数据, 否则函数的行为是未定义的。

```
void list_push_front(list_t* plist_list, elem);
```

描述:

向 list 开头添加数据。

参数:

plist_list 指向 list 容器的指针。
elem 添加的数据。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。elem 是与 list 容器中保存的数据类型一致的数据, 否则函数的行为是未定义的。

```
void list_pop_back(list_t* plist_list);
```

描述:

删除 list 容器中的最后一个数据

参数:

plist_list 指向 list 容器的指针。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。如果 list 容器为空则函数的行为是未定义的。

```
void list_pop_front(list_t* plist_list);
```

描述:

删除 list 容器中的第一个数据

参数:

plist_list 指向 list 容器的指针。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的。如果 list 容器为空则函数的行为是未定义的。

```
list_iterator_t list_erase(list_t* plist_list, list_iterator_t it_pos);
```

描述:

删除 list 容器中指定位置的数据。

参数:

plist_list 指向 list 容器的指针。

it_pos 被删除的数据的位置。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 plist_list == NULL 则函数的行为是未定义的, list 容器必须已经初始化的, 否则函数的行为是未定义的 。
it_pos 是属于 plist_list 容器的有效的迭代器, 否则函数的行为是未定义的。

```
list_iterator_t list_erase_range(  
    list_t* plist_list, list_iterator_t it_begin, list_iterator_t it_end);
```

描述:

删除 list 容器中指定数据区间的数据。

参数:

plist_list 指向 list 容器的指针。

it_begin 指定的数据区间的开头。

it_end 指定的数据区间的末尾。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 `plist_list == NULL` 则函数的行为是未定义的，`list` 容器必须已经初始化的，否则函数的行为是未定义的。`[it_begin, it_end)` 是属于 `plist_list` 的有效数据区间，否则函数的行为是未定义的。

```
void list_remove(list_t* plist_list, elem);
```

描述：

删除 `list` 容器中指定的数据。

参数：

`plist_list` 指向 `list` 容器的指针。
`elem` 要删除的数据。

返回值：

无。

注意：

如果 `plist_list == NULL` 则函数的行为是未定义的，`list` 容器必须已经初始化的，否则函数的行为是未定义的。`elem` 是与 `list` 中保存的数据类型一致的数据，否则函数的行为是未定义的。如果 `list` 容器中包含 `elem` 则删除，如果不包含 `elem`，这个函数不做任何操作。

```
void list_remove_if(list_t* plist_list, unary_function_t ufun_op);
```

描述：

删除 `list` 容器中符合条件的数据。

参数：

`plist_list` 指向 `list` 容器的指针。
`ufun_op` 删除数据的条件。

返回值：

无。

注意：

如果 `plist_list == NULL` 则函数的行为是未定义的，`list` 容器必须已经初始化的，否则函数的行为是未定义的。`ufun_op` 是一个一元谓词，如果 `ufun_op == NULL` 则使用默认的一元谓词。

```
void list_resize(list_t* plist_list, size_t t_resize);
```

描述：

重新设置 `list` 中数据的数量。

参数：

`plist_list` 指向 `list` 容器的指针。
`t_resize` `list` 中数据的新数量。

返回值：

无。

注意：

如果 `plist_list == NULL` 则函数的行为是未定义的，`list` 容器必须已经初始化的，否则函数的行为是未定义的。如果重新设置的数据的数量超过了原有的数据的数量，那么添加的部分使用 `list` 容器中保存的数据类型的默认值来填充。

```
void list_resize_elem(list_t* plist_list, size_t t_resize, elem);
```

描述：

重新设置 list 中数据的数量，超出的部分使用 elem 来填充。

参数:

- plist_list 指向 list 容器的指针。
- t_resize list 中数据的新的数量。
- elem 填充数据。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。如果重新设置的数据的数量超过了原有的数据的数量，那么添加的部分 elem 来填充。elem 必须是与 list 容器中保存的数据的类型一致的数据，否则函数的行为是未定义的。

```
void list_clear(list_t* plist_list);
```

描述:

删除 list 中全部的数据。

参数:

- plist_list 指向 list 容器的指针。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。

```
void list_unique(list_t* plist_list);
```

描述:

使 list 容器中的数据唯一。

参数:

- plist_list 指向 list 容器的指针。

返回值:

无。

注意:

如果 plist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。这个函数删除掉相邻且相等的数据的后面一个，但是不相邻的数据则不删除。

```
void list_unique_if(list_t* plist_list, binary_function_t bfun_op);
```

描述:

使 list 容器中的符合条件的数据唯一。

参数:

- plist_list 指向 list 容器的指针。
- bfun_op 使数据唯一的条件。

返回值:

无。

注意:

如果 `plist_list == NULL` 则函数的行为是未定义的，`list` 容器必须已经初始化的，否则函数的行为是未定义的。这个函数删除掉相邻且符合条件的数据的后面一个，但是不相邻的数据则不删除。`bfun_op` 是二元谓词，如果 `bfun_op == NULL` 那么使用默认的二元谓词。

```
void list_splice(list_t* plist_list, list_iterator_t it_pos, list_t* plist_src);
```

描述：

将既存 `list` 中的全部数据转移到另一个 `list` 的指定位置。

参数：

`plist_list` 指向 `list` 容器的指针。

`it_pos` 插入数据的位置。

`plist_src` 转移数据的 `list` 容器。

返回值：

无。

注意：

如果 `plist_list == NULL` 或者 `plist_src == NULL` 则函数的行为是未定义的，两个 `list` 容器必须已经初始化的，否则函数的行为是未定义的。`it_pos` 是属于 `plist_list` 的有效的迭代器，否则函数的行为是未定义的。`plist_src` 中保存的数据是与 `plist_list` 中保存的数据类型一致，否则函数的行为是未定义的。当 `plist_list == plist_src` 时，函数不执行任何动作。

```
void list_splice_pos(
    list_t* plist_list, list_iterator_t it_pos,
    list_t* plist_src, list_iterator_t it_src);
```

描述：

将既存 `list` 中的指定位置的数据转移到另一个 `list` 的指定位置。

参数：

`plist_list` 指向 `list` 容器的指针。

`it_pos` 插入数据的位置。

`plist_src` 转移数据的 `list` 容器。

`it_src` 转移数据的位置。

返回值：

无。

注意：

如果 `plist_list == NULL` 或者 `plist_src == NULL` 则函数的行为是未定义的，两个 `list` 容器必须已经初始化的，否则函数的行为是未定义的。`it_pos` 是属于 `plist_list` 的有效的迭代器，且 `it_src` 是属于 `plist_src` 的有效迭代器，否则函数的行为是未定义的。`plist_src` 中保存的数据是与 `plist_list` 中保存的数据类型一致，否则函数的行为是未定义的。当 `plist_list == plist_src` 时，并且 `it_pos == it_src` 或者 `it_pos == iterator_next(it_src)`，函数不进行任何操作。

```
void list_splice_range(
    list_t* plist_list, list_iterator_t it_pos,
    list_t* plist_src, list_iterator_t it_begin, list_iterator_t it_end);
```

描述：

将既存 `list` 中的指定的数据区间转移到另一个 `list` 的指定位置。

参数：

plist_list 指向 list 容器的指针。
it_pos 插入数据的位置。
plist_src 转移数据的 list 容器。
it_begin 转移的数据区间的开头。
it_end 转移的数据区间的末尾。

返回值：

无。

注意：

如果 plist_list == NULL 或者 plist_src == NULL 则函数的行为是未定义的，两个 list 容器必须已经初始化的，否则函数的行为是未定义的。it_pos 是属于 plist_list 的有效的迭代器，且 [it_begin, it_end) 是属于 plist_src 的有效数据区间，否则函数的行为是未定义的。plist_src 中保存的数据是与 plist_list 中保存的数据类型一致，否则函数的行为是未定义的。it_pos 不能属于 [it_begin, it_end)，否则函数的行为是未定义的。当 plist_list == plist_src 时，并且 it_pos == it_begin 或者 it_pos == it_end 函数不进行任何操作。

```
void list_sort(list_t* plist_list);
```

描述：

将 list 中的数据按照由小到大的顺序排序。

参数：

plist_list 指向 list 容器的指针。

返回值：

无。

注意：

如果 plist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。

```
void list_sort_if(list_t* pt_list, binary_function_t bfun_op);
```

描述：

将 list 中的数据按照指定的规则排序。

参数：

plist_list 指向 list 容器的指针。

bfun_op 排序规则。

返回值：

无。

注意：

如果 plist_list == NULL 则函数的行为是未定义的，list 容器必须已经初始化的，否则函数的行为是未定义的。bfun_op 是一个表示比较的二元谓词。如果 bfun_op == NULL 使用 list 中保存的数据类型默认的比较函数。

```
void list_merge(list_t* plist_dest, list_t* plist_src);
```

描述：

将两个有序的 list 合并。

参数：

plist_dest 指向 list 容器的指针。

plist_src 指向 list 容器的指针。

返回值:

无。

注意:

如果 `plist_dest == NULL` 或者 `plist_src == NULL` 则函数的行为是未定义的，两个 `list` 容器必须已经初始化的，否则函数的行为是未定义的。两个 `list` 容器中保存的数据类型必须是一致的，否则函数的行为是未定义的。两个 `list` 容器必须都是按照默认比较规则排序的，否则函数的行为是未定义的。如果 `plist_dest == plist_src` 则函数不执行任何操作。

```
void list_merge_if(  
    list_t* plist_dest, list_t* plist_src, binary_function_t bfun_op);
```

描述:

将两个按照指定规则排序的 `list` 合并。

参数:

<code>plist_dest</code>	指向 <code>list</code> 容器的指针。
<code>plist_src</code>	指向 <code>list</code> 容器的指针。
<code>bfun_op</code>	指定的排序规则。

返回值:

无。

注意:

如果 `plist_dest == NULL` 或者 `plist_src == NULL` 则函数的行为是未定义的，两个 `list` 容器必须已经初始化的，否则函数的行为是未定义的。两个 `list` 容器中保存的数据类型必须是一致的，否则函数的行为是未定义的。两个 `list` 容器必须都是按照默认比较规则排序的，否则函数的行为是未定义的。如果 `plist_dest == plist_src` 则函数不执行任何操作。`bfun_op` 是表示排序规则的二元谓词，如果 `bfun_op == NULL` 则使用 `list` 容器中保存的数据类型的默认比较函数。

```
void list_reverse(list_t* plist_list);
```

描述:

将 `list` 中的数据逆序。

参数:

<code>plist_list</code>	指向 <code>list</code> 容器的指针。
-------------------------	-----------------------------

返回值:

无。

注意:

如果 `plist_list == NULL` 则函数的行为是未定义的，`list` 容器必须已经初始化的，否则函数的行为是未定义的。

第五节 迭代器接口

`list` 的迭代器是双向的迭代器，所以它没有随机访问迭代器那么强大的功能，向前和向后移动只能单步运行。这些接口只能由迭代器接口使用，用户不应该直接使用这些接口函数。

<code>_create_list_iterator</code>	创建 <code>list</code> 迭代器。
------------------------------------	---------------------------

<code>_list_iterator_get_value</code>	获得 <code>list</code> 迭代器引用的数据。
---------------------------------------	--------------------------------

<code>_list_iterator_set_value</code>	设置 list 迭代器引用的数据。
<code>_list_iterator_get_pointer</code>	获得 list 迭代器引用的数据的指针。
<code>_list_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_list_iterator_prev</code>	获得引用前一个数据的迭代器。
<code>_list_iterator_equal</code>	测试两个迭代器是否相等。
<code>_list_iterator_distance</code>	计算两个迭代器之间的距离。
<code>_list_iterator_before</code>	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
list_iterator_t _create_list_iterator(void);
```

描述:

创建一个 list 迭代器。

参数:

无。

返回值:

list 迭代器。

注意:

返回的 list 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _list_iterator_get_value(list_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

`it_iter` list 迭代器。

`pv_value` 保存数据的缓冲区。

返回值:

无。

注意:

`it_iter` 是有效的 list 迭代器，否则函数的行为是未定义的。`pv_value == NULL` 则函数的行为是未定义的，`pv_value` 是能够保存下 `it_iter` 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 `it_iter` 引用的数据被拷贝到 `pv_value` 指向的缓存区中。

```
void _list_iterator_set_value(list_iterator_t it_iter, const void* cpv_value);
```

描述:

设置迭代器引用的数据。

参数:

`it_iter` list 迭代器。

`cpv_value` 保存数据的缓冲区。

返回值:

无。

注意:

it_iter 是有效的 list 迭代器，否则函数的行为是未定义的。cpv_value == NULL 则函数的行为是未定义的，cpv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 cpv_value 中保存的数据被拷贝到迭代器引用的数据中。

```
const void* _list_iterator_get_pointer(list_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter list 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 list 迭代器，否则函数的行为是未定义的。

```
list_iterator_t _list_iterator_next(list_iterator_t t_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter list 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 list 迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 list 有效的迭代器，否则函数的行为是未定义的。

```
list_iterator_t _list_iterator_prev(list_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter list 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 list 迭代器，否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 list 有效的迭代器，否则函数的行为是未定义的。

```
bool_t _list_iterator_equal(
    list_iterator_t it_first, list_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first list 迭代器。
it_second list 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 list 容器的有效的 list 迭代器, 否则函数的行为是未定义的。两个 list 迭代器相等是指两个迭代器引用相同的数据。

```
int _list_iterator_distance(  
    list_iterator_t it_first, list_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_iterfirst list 迭代器。
it_itersecond list 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 list 容器的有效的 list 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为 0。

```
bool_t _list_iterator_before(  
    list_iterator_t it_first, list_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first list 迭代器。
it_second list 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 list 容器的有效的 list 迭代器, 否则函数的行为是未定义的。

第六节 内部和辅助接口

内部接口是提供给实现外部接口使用的, 用户不能够直接使用内部接口。

_create_list	创建一个 list 容器。
_create_list_auxiliary	创建 list 容器的辅助函数。

<code>_list_init_elem</code>	使用指定的数据初始化 list 容器。
<code>_list_init_elem_varg</code>	使用可变参数列表中的指定数据初始化 list 容器。
<code>_list_destroy_auxiliary</code>	list 容器销毁辅助函数。
<code>_list_assign_elem</code>	使用指定的数据为 list 赋值。
<code>_list_assign_elem_varg</code>	使用可变参数列表中的数据为 list 赋值。
<code>_list_push_back</code>	向 list 末尾添加一个指定数据。
<code>_list_push_back_varg</code>	向 list 末尾添加一个可变参数列表中指定的数据。
<code>_list_push_front</code>	向 list 开头添加一个指定数据。
<code>_list_push_front_varg</code>	向 list 开头添加一个可变参数列表中指定的数据。
<code>_list_resize_elem</code>	重新设置 list 中数据的个数，使用指定的数据填充。
<code>_list_resize_elem_varg</code>	重新设置 list 中数据的个数，使用可变参数列表中指定的数据。
<code>_list_remove</code>	从 list 中删除指定数据。
<code>_list_remove_varg</code>	从 list 中删除指定数据。
<code>_list_insert_n</code>	向 list 的指定位置插入指定的数据。
<code>_list_insert_n_varg</code>	向 list 的指定位置插入可变参数列表指定的数据。
<code>_list_init_elem_auxiliary</code>	初始化数据节点的辅助函数。

```
list_t* _create_list(const char* s_typename);
```

描述:

创建一个 list 容器。

参数:

`s_typename` list 容器中保存的数据类型。

返回值:

创建成功返回指向容器的指针，否则返回 NULL。

注意:

如果 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型，libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
bool_t _create_list_auxiliary(list_t* plist_list, const char* s_typename);
```

描述:

创建一个 list 容器的辅助函数。

参数:

`plist_list` list 容器

`s_typename` list 容器中保存的数据类型。

返回值:

创建成功返回 true，否则返回 false。

注意:

如果 `plist_list == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型, `libcstl` 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _list_init_elem(list_t* plist_list, size_t t_count, ...);
```

描述:

使用用户指定的数据初始化 list 容器。

参数:

`plist_list` 未初始化的 list 容器。
`t_count` 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 `plist_list == NULL` 则函数的行为是未定义的。list 容器必须是使用 `create_list` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
void _list_init_elem_varg(  
    list_t* plist_list, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据初始化 list 容器。

参数:

`plist_list` 未初始化的 list 容器。
`t_count` 数据个数。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 `plist_list == NULL` 则函数的行为是未定义的。list 容器必须是使用 `create_list` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
void _list_destroy_auxiliary(list_t* plist_list);
```

描述:

销毁 list 容器的辅助函数。

参数:

`plist_list` list 容器。

返回值:

无。

注意:

如果 `plist_list == NULL` 或者 list 不是使用 `create_list` 生成的则函数的行为是未定义的。

```
void _list_assign_elem(list_t* plist_list, size_t t_count, ...);
```

描述:

使用用户指定的数据为 list 容器赋值。

参数:

plist_list	list 容器。
t_count	数据个数。
...	用户指定的数据。

返回值:

无。

注意:

如果 `plist_list == NULL` 或者 `list` 是未初始化的 `list` 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _list_assign_elem_varg(  
    list_t* plist_list, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据为 list 容器赋值。

参数:

plist_list	list 容器。
t_count	数据个数。
val_elemlist	用户指定的数据的参数列表。

返回值:

无。

注意:

如果 `plist_list == NULL` 或者 `list` 是未初始化的 `list` 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _list_push_back(list_t* plist_list, ...);
```

描述:

向 list 容器末尾添加一个数据。

参数:

plist_list	list 容器。
...	用户指定的数据。

返回值:

无。

注意:

如果 `plist_list == NULL` 或者 `list` 是未初始化的 `list` 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _list_push_back_varg(list_t* plist_list, va_list val_elemlist);
```

描述:

向 list 容器末尾添加一个数据。

参数:

plist_list	list 容器。
val_elemlist	可变参数列表。

返回值:

无。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _list_push_front(list_t* plist_list, ...);
```

描述:

向 list 容器开头添加一个数据。

参数:

plist_list	list 容器。
...	用户指定的数据。

返回值:

无。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _list_push_front_varg(list_t* plist_list, va_list val_elemlist);
```

描述:

向 list 容器开头添加一个数据。

参数:

plist_list	list 容器。
val_elemlist	可变参数列表。

返回值:

无。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _list_resize_elem(list_t* plist_list, size_t t_resize, ...);
```

描述:

重新设置 list 容器中的数据个数，不足的部分使用指定数据填充。

参数:

plist_list list 容器。
t_resize 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用指定的数据填充。

```
void _list_resize_elem_varg(  
    list_t* plist_list, size_t t_resize, va_list val_elemlist);
```

描述:

重新设置 list 容器中的数据个数，不足的部分使用可变参数列表中的数据填充。

参数:

plist_list list 容器。
t_resize 数据个数。
val_elemlist 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用可变参数列表中的数据填充。

```
void _list_remove(list_t* plist_list, ...);
```

描述:

从 list 容器中删除指定的数据。

参数:

plist_list list 容器。
... 用户指定的数据。

返回值:

无。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。容器中所有等于指定数据的数据将会被删除，如果没有与指定数据相等的数据则不删除。

```
void _list_remove_varg(list_t* plist_list, va_list val_elemlist);
```

描述:

从 list 容器中删除指定的数据。

参数:

plist_list	list 容器。
val_elemlist	可变参数列表。

返回值:

无。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。容器中所有等于指定数据的数据将会被删除，如果没有与指定数据相等的数据则不删除。

```
list_iterator_t _list_insert_n(
    list_t* plist_list, list_iterator_t it_pos, size_t t_count, ...);
```

描述:

向 list 容器的指定位置插入多个数据。

参数:

plist_list	list 容器。
it_pos	插入数据的位置。
t_count	数据个数。
...	用户指定的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。it_pos 必须是属于 list 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
list_iterator_t _list_insert_n_varg(
    list_t* plist_list, list_iterator_t it_pos, size_t t_count,
    va_list val_elemlist);
```

描述:

向 list 容器的指定位置插入多个数据。

参数:

plist_list	list 容器。
it_pos	插入数据的位置。
t_count	数据个数。
val_elemlist	用户指定的数据可变参数列表。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 plist_list == NULL 或者 list 是未初始化的 list 则函数的行为是未定义的。it_pos 必须是属于 list 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个

函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _list_init_elem_auxiliary(list_t* plist_list, void* pv_elem);
```

描述:

使用 list 的数据类型对数据进行初始化。

参数:

plist_list	list 容器。
pv_elem	数据。

返回值:

无。

注意:

如果 plist_list == NULL 或者 pv_elem == NULL 则函数的行为是未定义的。list 必须是已经初始化或者是使用 create_list 创建的 list 容器，否则函数的行为是未定义的。

辅助接口是提供给实现迭代器接口，内部接口以及外部接口使用的，用户不能够直接使用辅助接口。

_list_is_created	测试 list 容器是否是由 create_list 函数创建的。
_list_is_inited	测试 list 容器是否已经被初始化。
_list_iterator_belong_to_list	测试一个迭代器是否属于指定的 list 容器。
_list_same_type	测试两个 list 保存的数据类型是否相同。
_list_same_list_iterator_type	测试 list 容器中保存的数据类型和迭代器引用的数据类型是否相等。
_list_get_varg_value_auxiliary	根据 list 容器中数据的类型获得可变参数列表中的数据。
_list_destroy_varg_value_auxiliary	销毁从可变参数列表中获得的数据。
_list_init_node_auxiliary	根据 list 容器中的数据类型初始化 list 节点。
_list_transfer	将数据区间内的数据转移到指定位置。
_list_swap_node	交换 list 中的两个节点。
_list_quick_sort	对 list 列表进行快速排序。

函数原型

```
bool_t _list_is_created(const list_t* cplist_list);
```

描述:

测试一个 list 是否是使用 create_list 创建的。

参数:

cplist_list	list 容器。
-------------	----------

返回值:

如果 list 是使用 create_list 创建的则返回 true，否则返回 false。

注意:

如果 cplist_list == NULL，函数的行为是未定义的。

```
bool_t _list_is_inited(const list_t* cplist_list);
```

描述:

测试一个 list 是否已经初始化。

参数:

cplist_list list 容器。

返回值:

如果 list 已经初始化了, 返回 true, 否则返回 false。

注意:

如果 cplist_list == NULL, 函数的行为是未定义的。

```
bool_t _list_iterator_belong_to_list(
    const list_t* cplist_list, list_iterator_t it_iter);
```

描述:

判断一个迭代器引用的数据是否属于制定的容器。

参数:

cplist_list list 容器。

it_iter list 迭代器。

返回值:

如果迭代器引用的数据属于 list 容器的有效范围内, 返回 true, 否则返回 false。

注意:

如果 cplist_list == NULL, 那么函数的行为是未定义的, 如果 it_iter 不是 list 迭代器, 那么函数的行为是未定义的, 如果 it_iter 不属于容器 list, 那么函数的行为是未定义的。

```
bool_t _list_same_type(const list_t* cplist_first, const list_t* cplist_second);
```

描述:

判断两个 list 容器中保存的数据类型是否相同。

参数:

cplist_first 第一个 list 容器。

cplist_second 第二个 list 容器。

返回值:

如果两个 list 中保存的数据类型相同返回 true, 否则返回 false。

注意:

如果 cplist_first == NULL 或者 cplist_second == NULL, 那么函数的行为是未定义的。两个 list 容器必须是已经初始化或者使用 create_list 创建的 list 容器, 否则函数的行为是未定义的。如果 cplist_first == cplist_second 则返回 true。

```
bool_t _list_same_list_iterator_type(
    const list_t* cplist_list, list_iterator_t it_iter);
```

描述:

判断 list 容器中保存的数据类型与迭代器引用的数据类型是否相同。

参数:

cplist_list list 容器。

it_iter list 迭代器。

返回值:

如果 list 中保存的数据类型与 it_iter 引用的数据类型相同返回 true, 否则返回 false。

注意:

如果 plist_list == NULL 或者 it_iter 不是 list_iterator_t 类型, 那么函数的行为是未定义的。

```
void _list_get_varg_value_auxiliary(  
    list_t* plist_list, va_list val_elemlist, _listnode_t* pt_node);
```

描述:

从可变参数列表中获得与 list 中的数据类型相同的数据。

参数:

plist_list	list 容器。
val_elemlist	可变参数列表。
pt_node	保存数据缓的冲区。

返回值:

无。

注意:

如果 plist_list == NULL 或者 pt_node == NULL, 那么函数的行为是未定义的。plist_list 必须是已经初始化或者是由 create_list() 创建的 list 容器, 否则函数的行为是未定义的。

```
void _list_destroy_varg_value_auxiliary(list_t* plist_list, _listnode_t* pt_node);
```

描述:

销毁与 list 中的数据类型相同的数据。

参数:

plist_list	list 容器。
pt_node	保存数据的缓冲区。

返回值:

无。

注意:

如果 plist_list == NULL 或者 pt_node == NULL, 那么函数的行为是未定义的。plist_list 必须是已经初始化或者是由 create_list() 创建的 list 容器, 否则函数的行为是未定义的。

```
void _list_init_node_auxiliary(list_t* plist_list, _listnode_t* pt_node);
```

描述:

根据 list 中的数据类型初始化 list 节点。

参数:

plist_list	list 容器。
pt_node	保存数据的缓冲区。

返回值:

无。

注意:

如果 plist_list == NULL 或者 pt_node == NULL, 那么函数的行为是未定义的。plist_list 必须是已经初始化或者是由 create_list() 创建的 list 容器, 否则函数的行为是未定义的。

```
void _list_transfer(
    list_iterator_t it_pos, list_iterator_t it_begin, list_iterator_t it_end);
```

描述:

将数据区间[it_begin, it_end)内的数据转移到 it_pos。

参数:

it_pos 指定的插入数据的位置。
it_begin 转移的数据区间的开始位置。
it_end 转移的数据区间的末尾位置。

返回值:

无。

注意:

it_pos, it_begin, it_end 必须是有效的 list 迭代器, 否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间, it_pos 与[it_begin, it_end)可以属于同一个 list 容器, 但是 it_pos 属于(it_begin, it_end)的时候函数的行为是未定义的。it_pos == it_begin 或者 it_pos == it_end 或者 it_begin == it_end 的时候, 函数不做任何操作。

```
void _list_swap_node(listnode_t** ppt_first, listnode_t** ppt_second);
```

描述:

交换 list 中的两个节点。

参数:

ppt_first 第一个 list 节点。
ppt_second 第二个 list 节点。

返回值:

无。

注意:

ppt_first == NULL 或者*ppt_first == NULL 或者 ppt_second == NULL 或者*ppt_second == NULL 则函数的行为是未定义的。如果*ppt_first == *ppt_second 那么函数不执行任何动作。

```
void _list_quick_sort(
    list_t* plist_list, listnode_t* pt_first, listnode_t* pt_last,
    binary_function_t bfun_less);
```

描述:

对 list 列表中的区间[pt_first, pt_last]进行快速排序。

参数:

plist_list list 容器。
pt_first 第一个 list 节点。
pt_last 最后一个 list 节点。
bfun_less 排序规则。

返回值:

无。

注意:

plist_list == NULL 或者 pt_first == NULL 或者 pt_last == NULL 则函数的行为是未定义的。plist_list 必须是已经初始

化的 list，否则函数的行为是未定义的。[pt_first, pt_last]必须是属于 plist_list 的有效的数据区间，否则函数的行为是未定义的。如果 bfun_less == NULL 使用 plist_list 中保存的数据类型的默认比较规则。

第八章 deque 容器

deque 容器是双端队列，它提供随机访问迭代器。在 deque 的开头和末尾插入和删除数据花费常数时间，deque 不支持容量的概念，但是 deque 可以自动生长，在删除数据的时候，deque 还可以自动的收缩。

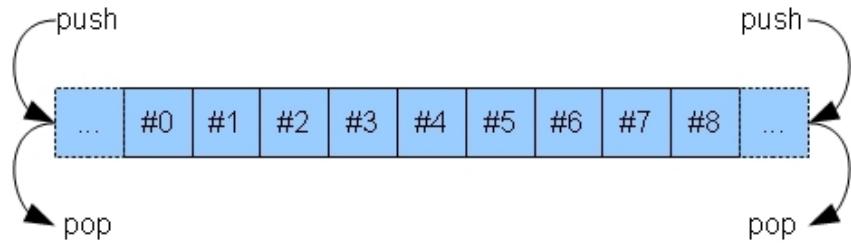


图 8.1 deque 容器

第一节 deque 容器的机制

为了能够让 deque 容器可以在两端进行常数级别的插入和删除操作，同时还要满足容器的自动生长和自动收缩的需要，采用分段存储数据的方式。这样 deque 容器可以很容易的向两端生长，同时当一个存储数据的段用完之后可以生长的方向上再分配一个，当一个存储数据的段中的数据都被删除以后，那么这个段就可以被销毁了并释放掉它占用的资源。此外还需要一个管理这些段的结构。

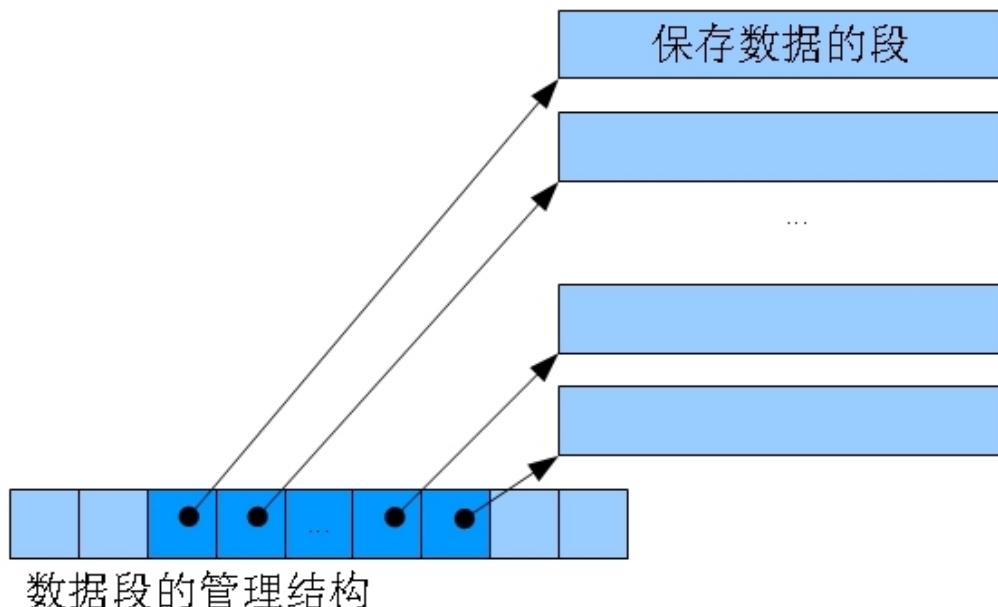


图 8.2 deque 的内部结构

除了这些之外我们还需要知道容器中保存的数据的位置，一个数据段可以保存多个数据，我们需要知道哪些已经保存了数据，哪些还没有使用。

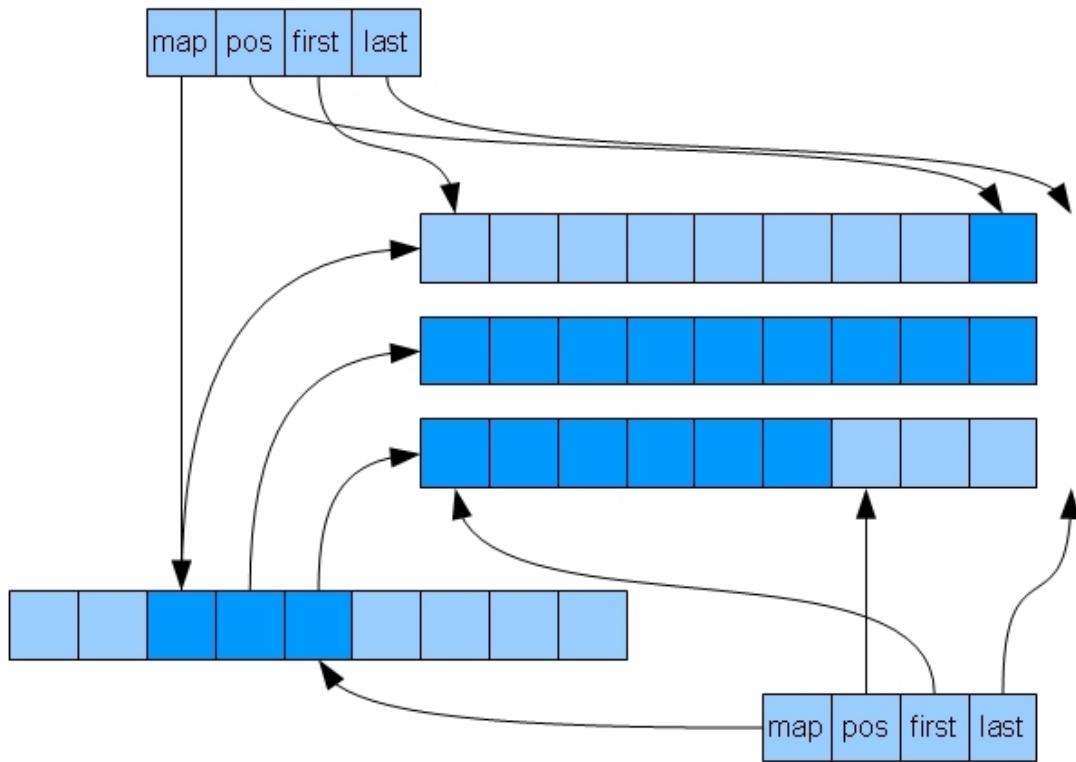


图 8.3 deque 的内部结构

通过 map 可以找到当前引用的数据块，通过 pos 可以找当前保存数据的位置，first 表示本块的第一个数据的位置，last 表示本块的最后一个数据的下一个位置。在第一个和最后一个保存数据的数据块中间的块都是满块，就是它们都已经满了。当在 deque 的开头插入数据的时候，指向第一个数据块的 pos 向前移动一个位置，然后将数据插入到 pos 指向的内存中，删除 deque 开头数据的动作正好相反，将指向第一个数据块的 pos 位置的数据删除，然后 pos 向后移动一个位置。同样在 deque 的末尾插入数据时，先向指向最后一个数据块的 pos 位置插入数据，然后将 pos 后移一个位置，删除 deque 末尾数据是，先将指向最后一个数据块的 pos 位置向前移动一个位置，然后删除 pos 指向的数据。

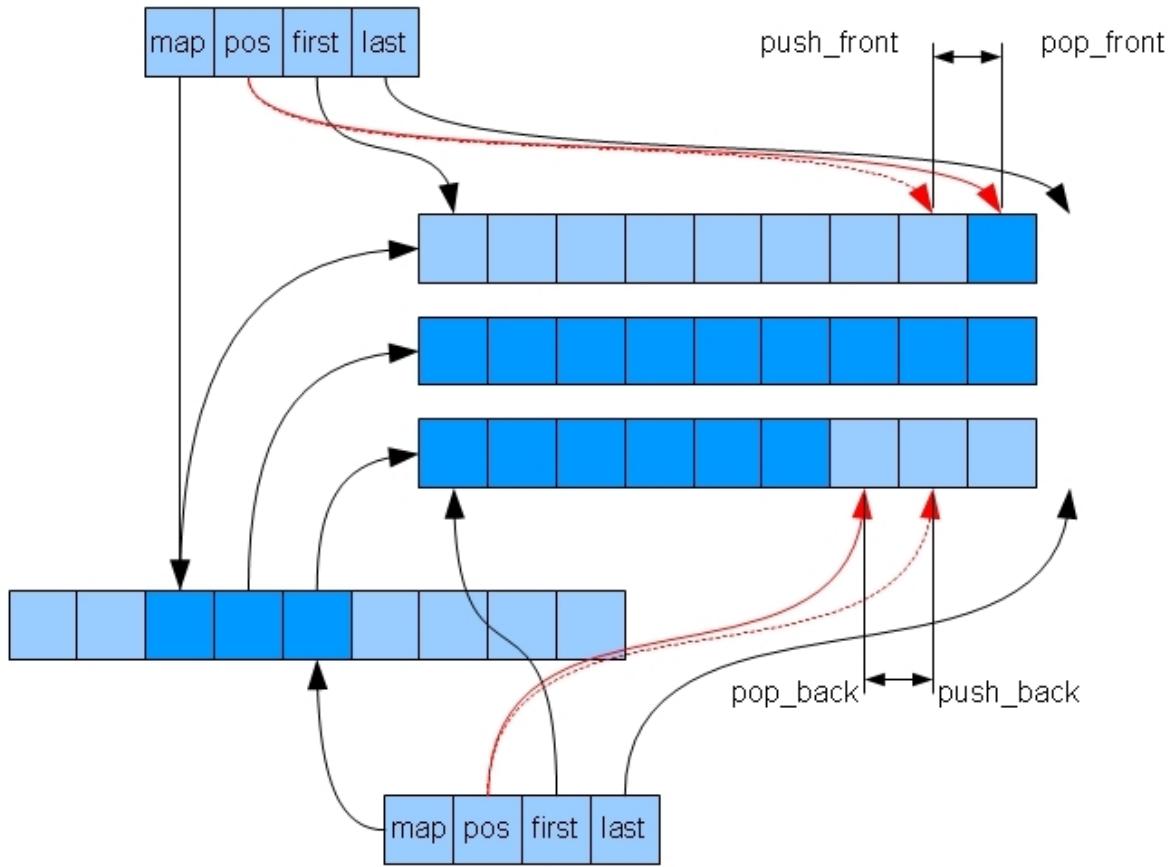


图 8.4 在 deque 开头和末尾插入或者删除数据

当一个数据块中的空间都被填满的时候，再向 `deque` 中添加数据就必须在分配数据块来保存新插入的数据。这种情况在开头和末尾都有可能发生。同样的当删除一个数据后，一个数据块中的数据都被删除了，这个时候这个数据块就会被释放，这样的情况在开头和末尾也都会发生。如下图，图中的实线部分表示操作之前的情况，虚线表示操作之后的情况，红色的线表示插入数据之后导致了分配新的数据块，绿色的线表示删除数据之后导致数据块被释放。

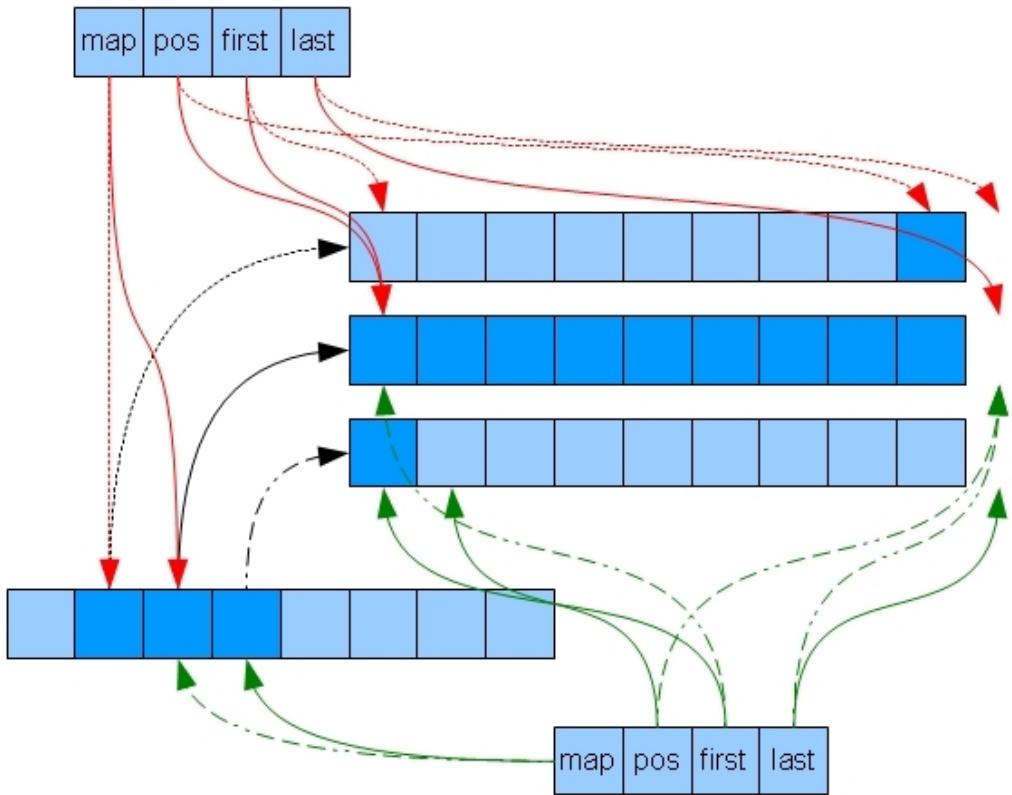


图 8.5 数据块的重新分配

在 deque 中插入数据是不明智的，这样的操作花费太多的时间，甚至比在 vector 中间插入数据的效率还要慢，因为 deque 是分段的，在移动数据的时候还要考虑跨越各个数据块的情况。

第二节 deque 迭代器

deque 提供了随机访问迭代器，但是由于 deque 的结构，这个迭代器的实现要比 vector 的迭代器复杂的多。首先迭代器必须保存当前迭代器引用的数据，以及数据所在的数据块，同时在迭代器移动的时候还要知道移动后迭代器是否超出了当前数据块的范围，如果超出了范围就要将整个迭代器移动到上一个或者下一个数据块中。图 8.5 中的 map, pos, first, last 结构就满足迭代器的这些要求。

当迭代器移动的时候，如果移动的范围在一个数据块以内，这样的情况很好处理，只是将迭代器中的 pos 的位置移动相应的距离就行了。当移动的范围跨越一个甚至是多个数据块的时候，这样就必须修改迭代器中全部 map, pos, first, last 变量，使它指向新的位置。图 8.5 中能够说明这种情况情况，以上面的迭代器为例，红色的虚线表示移动之前的迭代器的位置，红色的实线表示向后移动一个数据后的迭代器位置。下面的迭代器也说明了向前移动导致跨越数据块的情况，绿色的实线表示移动之前迭代器的位置，绿色的虚线表示移动之后迭代器的位置。

这种分块的结构给迭代器之间的比较也造成了麻烦，当相互比较的迭代器引用的数据位于同一个数据块的时候，这是最简单的情况，直接比较引用的数据的位置就可以了。还有一种情况也是相对来说比较简单的就是两个相互比较的迭代器位于不相邻的两个数据块中，这样比较数据块的位置就可以比较两个迭代器的关系了。最复杂的就是两个迭代器位于相邻的数据块的连接处的时候，这样的情况下，当一个迭代器引用前一个数据块的最后一个数据的下一个位置，另一个迭代器引用的是后一个数据块的第一个数据，虽然这两个迭代器引用的数据在不同的数据块但是这两个迭代器是相等的。图 8.6 中红色的迭代器和绿色的迭代器是相等的。不过这种情况只能出现在容器的开头，因为在容器被赋值的情况下不可能出现 pos 指向无效数据的情况，在迭代器向下移动的时候，如果结果是 pos 指向无效的数据，那

么移动函数会自动的将迭代器调整为指向正确位置的迭代器。

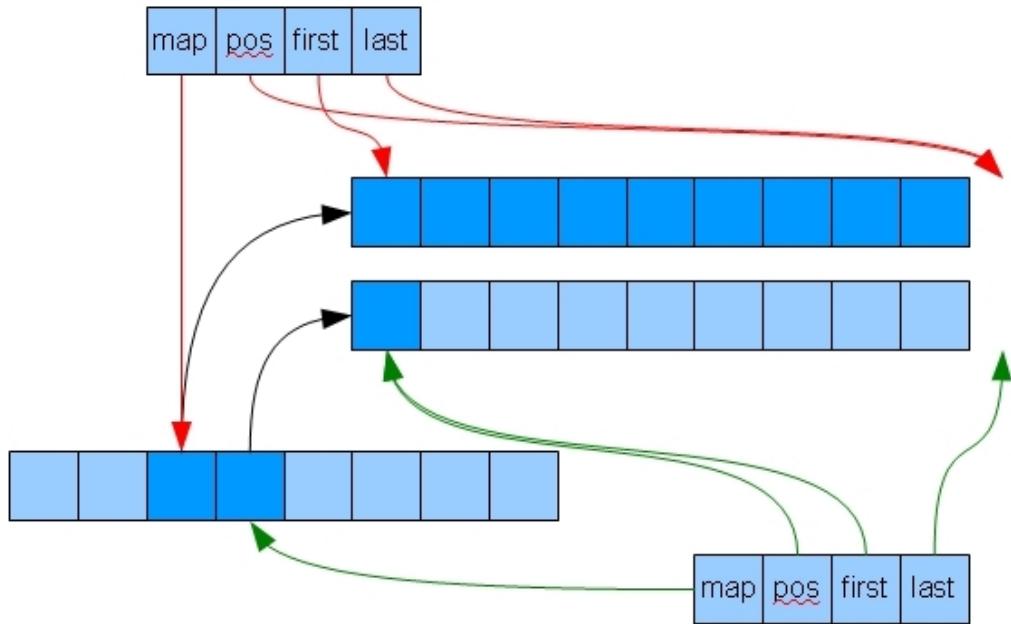


图 8.6 相等的迭代器

deque 迭代器的结构复杂，所以代码也复杂：

```
typedef struct _tagiterator
{
    /* 容器内部结构的信息 */
    union
    {
        struct
        {
            _byte_t*     _pby_corepos;           /* 引用的数据 */
            _byte_t*     _pby_first;             /* 数据块的第一个数据 */
            _byte_t*     _pby_afterlast;         /* 数据块的最后一个数据的下一个位置 */
            _byte_t**    _ppby_mappos;           /* 当前的数据块 */
            }_t_dequepos;
        }_t_pos;
    void*          _pt_container;           /* 容器的位置 */
    containertype_t _t_containertype;       /* 容器的类型 */
    iteratortype_t _t_iteratortype;         /* 迭代器的类型 */
}iterator_t;
```

第三节 deque 的代码结构

根据上面的结果可以很容易的得到 deque 的代码结构：

```

typedef _byte_t** _mappointer_t;
typedef struct _tagdeque
{
    _typeinfo_t      _t_typeinfo;
    _alloc_t         _t_allocator;
    _mappointer_t    _ppby_map;        /* 数据块管理表 */
    size_t           _t_mapsize;       /* 数据块管理表的大小 */
    deque_iterator_t _t_start;        /* 指向 deque 开头的迭代器 */
    deque_iterator_t _t_finish;       /* 指向 deque 末尾的迭代器 */
}deque_t;

```

deque 的实际代码结构就是这样的：

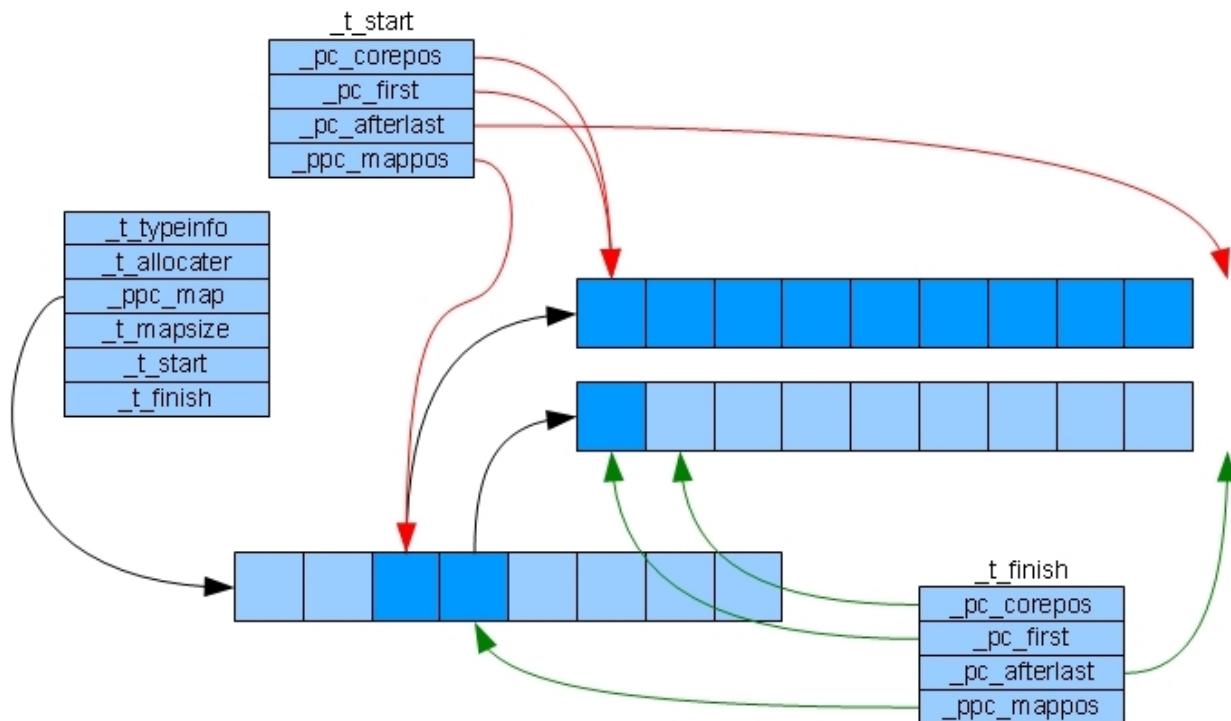


图 8.7 deque 的代码结构

第四节 外部接口

deque 提供给用户的外部接口：

create_deque	创建 deque 容器。
deque_init	初始化一个空的 deque 容器。
deque_init_n	初始化一个包含多个默认数据的 deque 容器。

deque_init_elem	初始化一个包含多个指定数据的 deque 容器。
deque_init_copy	使用一个已经存在的 deque 来初始化 deque 容器。
deque_init_copy_range	使用一个指定的数据区间来初始化 deque 容器。
deque_destroy	销毁一个 deque 容器。
deque_empty	测试 deque 容器是否为空。
deque_size	返回 deque 容器中数据的个数。
deque_max_size	返回 deque 容器中能够保存数据的最大数量。
deque_begin	返回引用 deque 容器第一个数据的迭代器。
deque_end	返回引用 deque 容器末尾的迭代器。
deque_assign	使用一个已经存在的 deque 容器为 deque 容器赋值。
deque_assign_elem	使用指定数据为 deque 容器赋值。
deque_assign_range	使用指定的数据区间为 deque 容器赋值。
deque_equal	测试两个 deque 容器是否相等。
deque_not_equal	测试两个 deque 容器是否不等。
deque_less	测试第一个 deque 容器是否小于第二个 deque 容器。
deque_less_equal	测试第一个 deque 容器是否小于等于第二个 deque 容器。
deque_greater	测试第一个 deque 容器是否大于第二个 deque 容器。
deque_greater_equal	测试第一个 deque 容器是否大于等于第二个 deque 容器。
deque_at	通过下标对 deque 容器中的数据进行随机访问。
deque_front	访问 deque 容器中的第一个数据。
deque_back	访问 deque 容器中的最后一个数据。
deque_swap	交换两个 deque 容器。
deque_push_back	向 deque 容器的末尾添加数据。
deque_push_front	向 deque 容器的开头添加数据。
deque_pop_back	删除 deque 容器的最后一个数据。
deque_pop_front	删除 deque 容器的第一个数据。
deque_insert	向 deque 容器中插入一个数据。
deque_insert_n	向 deque 容器中插入多个数据。
deque_insert_range	向 deque 容器中插入一个数据区间。
deque_erase	删除 deque 容器中指定位置的数据。
deque_erase_range	删除 deque 容器中指定数据区间的数据。
deque_clear	删除 deque 容器中的全部数据。

deque_resize	重新设置 deque 容器中数据的数量，使用默认数据填充。
deque_resize_elem	重新设置 deque 容器中数据的数量，使用指定数据填充。

函数原型

```
deque_t* create_deque(typename);
```

描述:

创建 deque 容器。

参数:

typename 类型描述。

返回值:

返回创建的 deque 容器的指针。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void deque_init(deque_t* pdeq_deque);
```

描述:

初始化一个空的 deque 容器。

参数:

pdeq_deque 指向 deque 容器的指针。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的，deque 必须是通过 create_deque 创建的否则函数的行为是未定义的。

```
void deque_init_n(deque_t* pdeq_deque, size_t t_count);
```

描述:

使用多个数据默认初始化 deque 容器。

参数:

pdeq_deque 指向 deque 容器的指针。

t_count deque 容器中包含的数据个数。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的，deque 必须是通过 create_deque 创建的否则函数的行为是未定义的。初始化后容器中包含的数据都是 deque 容器中保存的数据类型的默认值。

```
void deque_init_elem(deque_t* pdeq_deque, size_t t_count, elem);
```

描述:

使用多个指定的数据初始化 deque 容器。

参数:

pdeq_deque 指向 deque 容器的指针。
t_count deque 容器中包含的数据个数。
elem 初始化容器的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 必须是通过 create_deque 创建的否则函数的行为是未定义的。elem 是与 deque 容器中保存的数据类型相同的数据, 否则函数的行为是未定义的。

```
void deque_init_copy(deque_t* pdeq_dest, const deque_t* cpdeq_src);
```

描述:

使用既存的 deque 容器初始化另一个 deque 容器。

参数:

pdeq_dest 指向被初始化 deque 容器的指针。
cpdeq_src 指向既存的 deque 容器的指针。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 必须是通过 create_deque 创建的否则函数的行为是未定义的。既存的 deque 是有效的 deque 否则函数的行为是未定义的。既存的 deque 要与当前的 deque 保存的数据类型相同, 否则函数的行为是未定义的。初始化之后两个容器中的数据相等。

```
void deque_init_copy_range(  
    deque_t* pdeq_dest, deque_iterator_t it_begin, deque_iterator_t it_end);
```

描述:

使用指定的数据区间初始化另一个 deque 容器。

参数:

plis_dest 指向被初始化 deque 容器的指针。
it_begin 指定的数据区间的开始。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pdequeue_dest == NULL 则函数的行为是未定义的, deque 必须是通过 create_deque 创建的否则函数的行为是未定义的。[it_begin, it_end) 是有效的数据区间, 否则函数的行为是未定义的。数据区间中保存的数据是与 deque 中保存的数据类型相同, 否则函数的行为是未定义的。

```
void deque_destroy(deque_t* pdeq_deque);
```

描述:

销毁一个被创建出来的 deque 容器。

参数:

pdeq_deque 指向 deque 容器的指针。

返回值:

无。

注意:

如果 `pdeq_deque == NULL` 则函数的行为是未定义的, `deque` 必须是通过 `create_deque` 创建的否则函数的行为是未定义的。不管 `deque` 容器是否被初始化都需要调用这个函数销毁。

```
size_t deque_size(const deque_t* cpdeq_deque);
```

描述:

返回 `deque` 容器中保存的数据的个数。

参数:

`cpdeq_deque` 指向 `deque` 容器的指针。

返回值:

`deque` 容器中保存的数据的个数。

注意:

如果 `cpdeq_deque == NULL` 则函数的行为是未定义的, `deque` 必须已经初始化的, 否则函数的行为是未定义的。

```
bool_t deque_empty(const deque_t* cpdeq_deque);
```

描述:

测试 `deque` 容器是否为空。

参数:

`cpdeq_deque` 指向 `deque` 容器的指针。

返回值:

如果 `deque` 容器为空返回 `true`, 否则返回 `false`。

注意:

如果 `cpdeq_deque == NULL` 则函数的行为是未定义的, `deque` 必须已经初始化的, 否则函数的行为是未定义的。

```
size_t deque_max_size(const deque_t* cpdeq_deque);
```

描述:

返回 `deque` 容器中能够保存的数据最大数量的可能值。

参数:

`cpdeq_deque` 指向 `deque` 容器的指针。

返回值:

返回 `deque` 容器中能够保存的数据最大数量的可能值。

注意:

如果 `cpdeq_deque == NULL` 则函数的行为是未定义的, `deque` 必须已经初始化的, 否则函数的行为是未定义的。这个返回值不是一个固定的值。

```
deque_iterator_t deque_begin(const deque_t* cpdeq_deque);
```

描述:

获得引用 `deque` 容器中第一数据的迭代器。

参数:

`cpdeq_deque` 指向 `deque` 容器的指针。

返回值:

返回引用 deque 容器中第一个数据的迭代器。

注意:

如果 cpdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。如果 deque 容器为空, 则返回值与 deque_end(cpdeq_deque)相等。

```
deque_iterator_t deque_end(const deque_t* cpdeq_deque);
```

描述:

获得引用 deque 容器末尾的迭代器。

参数:

cpdeq_deque 指向 deque 容器的指针。

返回值:

返回引用 deque 容器末尾的迭代器。

注意:

如果 cpdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。

```
void deque_assign(deque_t* pdeq_dest, const deque_t* cpdeq_src);
```

描述:

使用一个 deque 为另一个 deque 赋值。

参数:

pdeq_dest 指向被赋值的 deque 容器的指针。

cpdeq_src 指向赋值的 deque 容器的指针。

返回值:

无。

注意:

如果 pdeq_dest == NULL 或者 cpdeq_src == NULL 则函数的行为是未定义的, 两个 deque 必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型不同函数的行为是未定义的。如果 deque_equal(pdeq_dest, cpdeq_src)那么函数不做任何动作。

```
void deque_assign_elem(deque_t* pdeq_deque, size_t t_count, elem);
```

描述:

使用指定的数据为 deque 赋值。

参数:

pdeq_deque 指向被赋值的 deque 容器的指针。

t_count 赋值的数据的个数。

elem 赋值的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。elem 是与 deque 中保存的数据类型相同的数据, 否则函数的行为是未定义的。

```
void deque_assign_range(  
    deque_t* pdeq_deque, deque_iterator_t it_begin, deque_iterator_t it_end);
```

描述:

使用指定的数据区间为 deque 赋值。

参数:

pdeq_deque 指向被赋值的 deque 容器的指针。
it_begin 指定的数据区间的开始。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的数据区间否则函数的行为是未定义的。[it_begin, it_end) 中保存的数据是与 deque 中保存的数据类型相同的数据, 否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 pdeq_deque 则函数的行为是未定义的。

```
bool_t deque_equal(const deque_t* cpdeq_first, const deque_t* cpdeq_second);
```

描述:

测试两个 deque 容器是否相等。

参数:

cpdeq_first 指向 deque 容器的指针。
cpdeq_second 指向 deque 容器的指针。

返回值:

如果两个 deque 容器相等则返回 true, 否则返回 false。

注意:

如果 cpdeq_first == NULL 或者 cpdeq_second == NULL 则函数的行为是未定义的, 两个 deque 必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型不同认为这两个容器不等。两个 deque 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cpdeq_first == cpdeq_second 那么返回 true。

```
bool_t deque_not_equal(const deque_t* cpdeq_first, const deque_t* cpdeq_second);
```

描述:

测试两个 deque 容器是否不等。

参数:

cpdeq_first 指向 deque 容器的指针。
cpdeq_second 指向 deque 容器的指针。

返回值:

如果两个 deque 容器不等则返回 true, 否则返回 false。

注意:

如果 cpdeq_first == NULL 或者 cpdeq_second == NULL 则函数的行为是未定义的, 两个 deque 必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型不同认为两个 deque 容器不等。两个 deque 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cpdeq_first == cpdeq_second 那么返回 false。

```
bool_t deque_less(const deque_t* cpdeq_first, const deque_t* cpdeq_second);
```

描述:

测试第一个 deque 是否小于第二个 deque。

参数:

cpdeq_first 指向 deque 容器的指针。
cpdeq_second 指向 deque 容器的指针。

返回值:

如果第一个 deque 容器小于第二个 deque 容器则返回 true, 否则返回 false。

注意:

如果 cpdeq_first == NULL 或者 cpdeq_second == NULL 则函数的行为是未定义的, 两个 deque 必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 deque 中的数据小于第二个 deque 中对应的数据则返回 true, 如果大于则返回 false, 当两个 deque 中的对应数据相等, 第一个 deque 中的数据数量小于第二个 deque 中数据的数量返回 true, 否则返回 false。如果 cpdeq_first == cpdeq_second 那么返回 false。

```
bool_t deque_less_equal(const deque_t* cpdeq_first, const deque_t* cpdeq_second);
```

描述:

测试第一个 deque 是否小于等于第二个 deque。

参数:

cpdeq_first 指向 deque 容器的指针。
cpdeq_second 指向 deque 容器的指针。

返回值:

如果第一个 deque 容器小于等于第二个 deque 容器则返回 true, 否则返回 false。

注意:

如果 cpdeq_first == NULL 或者 cpdeq_second == NULL 则函数的行为是未定义的, 两个 deque 必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型不同函数的行为是未定义的。如果 cpdeq_first == cpdeq_second 那么返回 true。

```
bool_t deque_greater(const deque_t* cpdeq_first, const deque_t* cpdeq_second);
```

描述:

测试第一个 deque 是否大于第二个 deque。

参数:

cpdeq_first 指向 deque 容器的指针。
cpdeq_second 指向 deque 容器的指针。

返回值:

如果第一个 deque 容器大于第二个 deque 容器则返回 true, 否则返回 false。

注意:

如果 cpdeq_first == NULL 或者 cpdeq_second == NULL 则函数的行为是未定义的, 两个 deque 必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 deque 中的数据大于第二个 deque 中对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 deque 中数据的数量大于第二个 deque 中数据的数量的时候返回 true 否则返回 false。如果 cpdeq_first == cpdeq_second 那么返回

false。

```
bool_t deque_greater_equal(
    const deque_t* cpdeq_first, const deque_t* cpdeq_second);
```

描述:

测试第一个 deque 是否大于等于第二个 deque。

参数:

cpdeq_first 指向 deque 容器的指针。
cpdeq_second 指向 deque 容器的指针。

返回值:

如果第一个 deque 容器大于等于第二个 deque 容器则返回 true, 否则返回 false。

注意:

如果 cpdeq_first == NULL 或者 cpdeq_second == NULL 则函数的行为是未定义的, 两个 deque 必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型不同函数的行为是未定义的。如果 cpdeq_first == cpdeq_second 那么返回 true。

```
void* deque_at(const deque_t* cpdeq_deque, size_t t_pos);
```

描述:

通过下标随机访问 deque 容器中的数据。

参数:

cpdeq_deque 指向 deque 容器的指针。
t_pos 要访问的数据在容器中的索引。

返回值:

指向被访问的数据的指针。

注意:

cpdeq_deque == NULL 则函数的行为是未定义的, cpdeq_deque 指向的容器是经过初始化以后的 deque 容器, 否则函数的行为是未定义的。如果 t_pos 超过了 deque 容器中数据索引的范围, 则函数的行为是未定义的。

```
void* deque_front(const deque_t* cpdeq_deque);
```

描述:

访问 deque 容器中的第一个数据

参数:

cpdeq_deque 指向 deque 容器的指针。

返回值:

指向 deque 容器中第一个数据的指针。

注意:

如果 cpdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。如果 deque 为空则函数的行为是未定义的。

```
void* deque_back(const deque_t* cpdeq_deque);
```

描述:

访问 deque 容器中的最后一个数据

参数:

cpdeq_deque 指向 deque 容器的指针。

返回值:

指向 deque 容器中最后一个数据的指针。

注意:

如果 cpdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。如果 deque 为空则函数的行为是未定义的。

```
void deque_swap(deque_t* pdeq_first, deque_t* pdeq_second);
```

描述:

交换两个 deque 容器。

参数:

pdeq_first 指向 deque 容器的指针。

pdeq_second 指向 deque 容器的指针。

返回值:

无。

注意:

如果 pdeq_first == NULL 或者 pdeq_second == NULL 则函数的行为是未定义的, 两个 deque 容器必须已经初始化的, 否则函数的行为是未定义的。两个 deque 容器中保存的数据类型必须是一致的, 否则函数的行为是未定义的。如果 deque_equal(pdeq_firt, pdeq_second)则函数不执行任何动作。

```
void deque_push_back(deque_t* pdeq_deque, elem);
```

描述:

向 deque 末尾添加数据。

参数:

pdeq_deque 指向 deque 容器的指针。

elem 添加的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。 elem 是与 deque 容器中保存的数据类型一致的数据, 否则函数的行为是未定义的。

```
void deque_push_front(deque_t* pdeq_deque, elem);
```

描述:

向 deque 开头添加数据。

参数:

pdeq_deque 指向 deque 容器的指针。

elem 添加的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的，deque 容器必须已经初始化的，否则函数的行为是未定义的。elem 是与 deque 容器中保存的数据类型一致的数据，否则函数的行为是未定义的。

```
void deque_pop_back(deque_t* pdeq_deque);
```

描述：

删除 deque 容器中的最后一个数据

参数：

pdeq_deque 指向 deque 容器的指针。

返回值：

无。

注意：

如果 pdeq_deque == NULL 则函数的行为是未定义的，deque 容器必须已经初始化的，否则函数的行为是未定义的。如果 deque 容器为空则函数的行为是未定义的。

```
void deque_pop_front(deque_t* pdeq_deque);
```

描述：

删除 deque 容器中的第一个数据

参数：

pdeq_deque 指向 deque 容器的指针。

返回值：

无。

注意：

如果 pdeq_deque == NULL 则函数的行为是未定义的，deque 容器必须已经初始化的，否则函数的行为是未定义的。如果 deque 容器为空则函数的行为是未定义的。

```
deque_iterator_t deque_insert(deque_t* pdeq_deque, deque_iterator_t it_pos, elem);
```

描述：

在 deque 容器的指定位置插入数据。

参数：

pdeq_deque 指向 deque 容器的指针。

it_pos 插入数据的位置。

elem 被插入的数据。

返回值：

返回引用被插入的数据的迭代器。

注意：

如果 pdeq_deque == NULL 则函数的行为是未定义的，deque 容器必须已经初始化的，否则函数的行为是未定义的。it_pos 必须是属于 pdeq_deque 容器的是有效迭代器，否则函数的行为是未定义的。elem 是与 deque 容器中保存的数据类型一致的数据，否则函数的行为是未定义的。

```
deque_iterator_t deque_insert_n(
    deque_t* pdeq_deque, deque_iterator_t it_pos, size_t t_count, elem);
```

描述：

在 deque 容器的指定位置插入多个数据。

参数:

pdeq_deque 指向 deque 容器的指针。
it_pos 插入数据的位置。
t_count 插入的数据的个数。
elem 被插入的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 必须是属于 pdeq_deque 容器的是有效迭代器, 否则函数的行为是未定义的。elem 是与 deque 容器中保存的数据类型一致的数据, 否则函数的行为是未定义的。

```
void deque_insert_range(  
    deque_t* pdeq_deque, deque_iterator_t it_pos,  
    deque_iterator_t it_begin, deque_iterator_t it_end);
```

描述:

在 deque 容器的指定位置插入一个数据区间。

参数:

pdeq_deque 指向 deque 容器的指针。
it_pos 插入数据的位置。
it_begin 插入的数据区间的开头。
it_end 插入的数据区间的末尾。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 必须是属于 pdeq_deque 容器的是有效迭代器, 否则函数的行为是未定义的。[it_begin, it_end) 必须是有效的数据区间, 否则函数的行为是未定义的, [it_begin, it_end) 中保存的数据类型与 deque 中保存的数据类型一致, 否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 pdeq_deque 则函数的行为是未定义的。

```
deque_iterator_t deque_erase(deque_t* pdeq_deque, deque_iterator_t it_pos);
```

描述:

删除 deque 容器中指定位置的数据。

参数:

pdeq_deque 指向 deque 容器的指针。
it_pos 被删除的数据的位置。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 是属于 pdeq_deque 容器的有效的迭代器, 否则函数的行为是未定义的。

```
deque_iterator_t deque_erase_range(  
    deque_t* pdeq_deque, deque_iterator_t it_begin, deque_iterator_t it_end);
```

描述:

删除 deque 容器中指定数据区间的数据。

参数:

pdeq_deque 指向 deque 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。
[it_begin, it_end) 是属于 pdeq_deque 的有效数据区间, 否则函数的行为是未定义的。

```
void deque_clear(deque_t* pdeq_deque);
```

描述:

删除 deque 中全部的数据。

参数:

pdeq_deque 指向 deque 容器的指针。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。

```
void deque_resize(deque_t* pdeq_deque, size_t t_resize);
```

描述:

重新设置 deque 中数据的数量。

参数:

pdeq_deque 指向 deque 容器的指针。
t_resize deque 中数据的新数量。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的, deque 容器必须已经初始化的, 否则函数的行为是未定义的。
如果重新设置的数据的数量超过了原有的数据的数量, 那么添加的部分使用 deque 容器中保存的数据类型的默认值来填充。

```
void deque_resize_elem(deque_t* pdeq_deque, size_t t_resize, elem);
```

描述:

重新设置 deque 中数据的数量, 超出的部分使用 elem 来填充。

参数:

pdeq_deque 指向 deque 容器的指针。

`t_resize` `deque` 中数据的新的数量。

`elem` 填充数据。

返回值:

无。

注意:

如果 `pdeq_deque == NULL` 则函数的行为是未定义的, `deque` 容器必须已经初始化的, 否则函数的行为是未定义的。如果重新设置的数据的数量超过了原有的数据的数量, 那么添加的部分 `elem` 来填充。`elem` 必须是与 `deque` 容器中保存的数据的类型一致的数据, 否则函数的行为是未定义的。

第五节 迭代器接口

`deque` 的迭代器接口主要用户实现迭代器, 用户不应该使用这些接口。

<code>_create_deque_iterator</code>	创建 <code>deque</code> 容器的迭代器。
<code>_deque_iterator_equal</code>	测试两个 <code>deque</code> 迭代器是否相等。
<code>_deque_iterator_less</code>	测试第一个 <code>deque</code> 迭代器是否小于第二个 <code>deque</code> 迭代器。
<code>_deque_iterator_before</code>	测试第一个 <code>deque</code> 迭代器是否在第二个 <code>deque</code> 迭代器前面。
<code>_deque_iterator_get_value</code>	获得迭代器引用的数据。
<code>_deque_iterator_set_value</code>	设置迭代器引用的数据。
<code>_deque_iterator_get_pointer</code>	获得指向迭代器引用的数据的指针。
<code>_deque_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_deque_iterator_prev</code>	获得引用上一个数据的迭代器。
<code>_deque_iterator_next_n</code>	获得引用向下第 n 个数据的迭代器。
<code>_deque_iterator_prev_n</code>	获得引用向上第 n 个数据的迭代器。
<code>_deque_iterator_at</code>	通过下标来访问迭代器引用的数据。
<code>_deque_iterator_minus</code>	获得两个迭代器之差。

```
deque_iterator_t _create_deque_iterator(void);
```

描述:

创建 `deque` 迭代器。

参数:

无。

返回值:

`deque` 迭代器。

注意:

获得的 `deque` 迭代器不是有效的迭代器, 因为它并没有与任何 `deque` 容器关联。这个函数创建的 `deque` 迭代器是供给以后的 `deque` 函数使用的。

```
bool_t _deque_iterator_equal(
    deque_iterator_t it_first, deque_iterator_t it_second);
```

描述:

比较两个迭代器是否相等。

参数:

it_first 第一个 deque 迭代器。
it_second 第二个 deque 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个迭代器必须属于同一个 deque 容器, 否则函数的行为是未定义的。两个迭代器相等表示的是两个迭代器引用的是同一个容器中的同一个数据。

```
bool_t _deque_iterator_less(
    deque_iterator_t it_first, deque_iterator_t it_second);
```

描述:

测试第一个迭代器是否小于第二个迭代器。

参数:

it_first 第一个 deque 迭代器。
it_second 第二个 deque 迭代器。

返回值:

如果第一个迭代器小于第二个迭代器则返回 true, 否则返回 false。

注意:

两个迭代器必须属于同一个 deque 容器, 否则函数的行为是未定义的。第一个迭代器小于第二个迭代器表示的是第一个迭代器引用的数据比第二个迭代器引用的数据更靠近容器中的第一个数据。

```
bool_t _deque_iterator_before(
    deque_iterator_t it_first, deque_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个第二个迭代器前面。

参数:

it_first 第一个 deque 迭代器。
it_second 第二个 deque 迭代器。

返回值:

如果第一个迭代器在第二个迭代器前面则返回 true, 否则返回 false。

注意:

两个迭代器必须属于同一个 deque 容器, 否则函数的行为是未定义的。第一个迭代器在第二个迭代器前面表示的是第一个迭代器引用的数据比第二个迭代器引用的数据更靠近容器中的第一个数据。

```
void _deque_iterator_get_value(deque_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter deque 迭代器。
pv_value 保存数据的缓存区。

返回值:

无。

注意:

it_iter 必须是属于 deque 的有效迭代器, 否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的, pv_value 的缓冲区要能够保存 it_iter 引用的数据, 否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 中。

```
void _deque_iterator_set_value(deque_iterator_t it_iter, const void* cpv_value);
```

描述:

设置迭代器引用的数据。

参数:

it_iter deque 迭代器。
cpv_value 保存数据的缓存区。

返回值:

无。

注意:

it_iter 必须是属于 deque 的有效迭代器, 否则函数的行为是未定义的。cpv_value == NULL 则函数的行为是未定义的, cpv_value 的缓冲区中保存的数据要与 it_iter 引用的数据类型相同, 否则函数的行为是未定义的。函数执行后 cpv_value 中保存的数据被拷贝到 it_iter 引用的数据中。

```
const void* _deque_iterator_get_pointer(deque_iterator_t it_iter);
```

描述:

返回被迭代器应用的数据的指针。

参数:

it_iter deque 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 必须是属于 deque 的有效迭代器, 否则函数的行为是未定义的。

```
deque_iterator_t _deque_iterator_next(deque_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter deque 迭代器。

返回值:

引用下一个数据的迭代器。

注意:

it_iter 必须是属于 deque 的有效迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是属于 deque 的有效的迭代器，否则函数行为是未定义的。

```
deque_iterator_t _deque_iterator_prev(deque_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter deque 迭代器。

返回值:

引用上一个数据的迭代器。

注意:

it_iter 必须是属于 deque 的有效迭代器，否则函数的行为是未定义的。引用上一个数据的迭代器也必须是属于 deque 的有效的迭代器，否则函数行为是未定义的。

```
deque_iterator_t _deque_iterator_next_n(deque_iterator_t it_iter, int n_step);
```

描述:

获得引用向下第 n 个数据的迭代器。

参数:

it_iter deque 迭代器。

n_step 前进的步数。

返回值:

引用向下第 n 个数据的迭代器。

注意:

it_iter 必须是属于 deque 的有效迭代器，否则函数的行为是未定义的。引用向下第 n 个数据的迭代器 也必须是属于 deque 的有效的迭代器，否则函数行为是未定义的。n_step > 0 表示迭代器向容器末尾移动 n 步，n_step < 0 表示迭代器向容器开头移动 n 步，n_step == 0 表示不移动。

```
deque_iterator_t _deque_iterator_prev_n(deque_iterator_t it_iter, int n_step);
```

描述:

获得引用向上第 n 个数据的迭代器。

参数:

it_iter deque 迭代器。

n_step 前进的步数。

返回值:

引用向上第 n 个数据的迭代器。

注意:

it_iter 必须是属于 deque 的有效迭代器，否则函数的行为是未定义的。引用向上第 n 个数据的迭代器 也必须是属于 deque 的有效的迭代器，否则函数行为是未定义的。n_step > 0 表示迭代器向容器开头移动 n 步，n_step < 0 表示迭代器向容器末尾移动 n 步，n_step == 0 表示不移动。

```
void* _deque_iterator_at(deque_iterator_t it_iter, int n_index);
```

描述:

使用下标随机访问迭代器引用的数据。

参数:

it_iter deque 迭代器。
n_index 访问的下标。

返回值:

获得随机访问的迭代器应用的数据的指针。

注意:

it_iter 必须是属于 deque 的有效迭代器, 否则函数的行为是未定义的。随机访问的数据也必须在 deque 范围内, 否则函数行为是未定义的。n_index > 0 表示访问从当前迭代器引用的数据起向容器末尾的第 n 个数据, n_index < 0 表示从当前迭代器引用的数据起向容器开头的第 n 个数据, n_index == 0 表示当前迭代器引用的数据。

```
int _deque_iterator_minus(  
    deque_iterator_t it_first, deque_iterator_t it_second);
```

描述:

求两个迭代器之间的差。

参数:

it_first 第一个 deque 迭代器。
it_second 第二个 deque 迭代器。

返回值:

返回两个迭代器之间相差的距离。

注意:

it_first 和 it_second 必须是属于同一个 deque 的有效迭代器, 否则函数的行为是未定义的。当 it_first < it_second 的时候返回值小于 0, it_first == it_second 时返回值等于 0, it_first > it_second 时返回值大于 0。

第六节 内部和辅助接口

deque 内部接口是用来实现外部接口使用的, 用户不应该直接使用这些接口。

_create_deque	创建一个 deque 容器。
_create_deque_auxiliary	创建 deque 容器的辅助函数。
_deque_init_elem	使用用户指定的数据初始化 deque 容器。
_deque_init_elem_varg	使用可变参数列表中的数据初始化 deque 容器。
_deque_destroy_auxiliary	销毁 deque 容器的辅助函数。
_deque_assign_elem	为 deque 容器赋值。
_deque_assign_elem_varg	为 deque 容器赋值, 值来自于可变参数列表。
_deque_push_back	向 deque 容器末尾添加一个数据。
_deque_push_back_varg	向 deque 容器末尾添加一个数据, 数据来自于可变参数列表。
_deque_push_front	向 deque 容器开头添加一个数据。

<code>_deque_push_front_varg</code>	向 deque 容器开头添加一个数据，数据来自于可变参数列表。
<code>_deque_resize_elem</code>	重新设置 deque 中数据的个数，使用指定的数据填充。
<code>_deque_resize_elem_varg</code>	重新设置 deque 中数据的个数，使用可变参数列表中的数据填充。
<code>_deque_insert_n</code>	向 deque 容器的指定位置插入多个数据。
<code>_deque_insert_n_varg</code>	向 deque 容器的指定位置插入多个来自于可变参数的数据。
<code>_deque_init_elem_auxiliary</code>	初始化数据的辅助函数。

```
deque_t* _create_deque(const char* s_typename);
```

描述:

创建一个 deque 容器。

参数:

`s_typename` deque 容器中保存的数据类型。

返回值:

创建成功返回指向容器的指针，否则返回 NULL。

注意:

如果 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型，libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
bool_t _create_deque_auxiliary(deque_t* pdeq_deque, const char* s_typename);
```

描述:

创建一个 deque 容器的辅助函数。

参数:

`pdeq_deque` 没有创建的 deque 容器。

`s_typename` deque 容器中保存的数据类型。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pdeq_deque == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型，libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
void _deque_init_elem(deque_t* pdeq_deque, size_t t_count, ...);
```

描述:

使用用户指定的数据初始化 deque 容器。

参数:

`pdeq_deque` 未初始化的 deque 容器。

`t_count` 数据个数。

`...` 用户指定的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的。deque 容器必须是使用 create_deque 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_init_elem_varg(
    deque_t* pdeq_deque, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据初始化 deque 容器。

参数:

pdeq_deque	未初始化的 deque 容器。
t_count	数据个数。
val_elemlist	用户指定的数据的参数列表。

返回值:

无。

注意:

如果 pdeq_deque == NULL 则函数的行为是未定义的。deque 容器必须是使用 create_deque 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_destroy_auxiliary(deque_t* pdeq_deque);
```

描述:

销毁 deque 容器的辅助函数。

参数:

pdeq_deque	deque 容器。
------------	-----------

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 不是使用 create_deque 生成的则函数的行为是未定义的。

```
void _deque_assign_elem(deque_t* pdeq_deque, size_t t_count, ...);
```

描述:

使用用户指定的数据为 deque 容器赋值。

参数:

pdeq_deque	deque 容器。
t_count	数据个数。
...	用户指定的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_assign_elem_varg(
    deque_t* pdeq_deque, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据为 deque 容器赋值。

参数:

pdeq_deque	deque 容器。
t_count	数据个数。
val_elemlist	用户指定的数据的参数列表。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_push_back(deque_t* pdeq_deque, ...);
```

描述:

向 deque 容器末尾添加一个数据。

参数:

pdeq_deque	deque 容器。
...	用户指定的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_push_back_varg(deque_t* pdeq_deque, va_list val_elemlist);
```

描述:

向 deque 容器末尾添加一个数据。

参数:

pdeq_deque	deque 容器。
val_elemlist	可变参数列表。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_push_front(deque_t* pdeq_deque, ...);
```

描述:

向 deque 容器开头添加一个数据。

参数:

pdeq_deque deque 容器。
... 用户指定的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_push_front_varg(deque_t* pdeq_deque, va_list val_elemlist);
```

描述:

向 deque 容器开头添加一个数据。

参数:

pdeq_deque deque 容器。
val_elemlist 可变参数列表。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _deque_resize_elem(deque_t* pdeq_deque, size_t t_resize, ...);
```

描述:

重新设置 deque 容器中的数据个数，不足的部分使用指定数据填充。

参数:

pdeq_deque deque 容器。
t_resize 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用指定的数据填充。

```
void _deque_resize_elem_varg(
```

```
deque_t* pdeq_deque, size_t t_resize, va_list val_elemlist);
```

描述:

重新设置 deque 容器中的数据个数, 不足的部分使用可变参数列表中的数据填充。

参数:

pdeq_deque	deque 容器。
t_resize	数据个数。
val_elemlist	用户指定的数据的参数列表。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量, 这是将末尾的数据删除, 当 t_resize 大于原来数据的数量的时候, 使用可变参数列表中的数据填充。

```
deque_iterator_t _deque_insert_n(  
    deque_t* pdeq_deque, deque_iterator_t it_pos, size_t t_count, ...);
```

描述:

向 deque 容器的指定位置插入多个数据。

参数:

pdeq_deque	deque 容器。
it_pos	插入数据的位置。
t_count	数据个数。
...	用户指定的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。it_pos 必须是属于 deque 容器的有效位置, 否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
deque_iterator_t _deque_insert_n_varg(  
    deque_t* pdeq_deque, deque_iterator_t it_pos, size_t t_count,  
    va_list val_elemlist);
```

描述:

向 deque 容器的指定位置插入多个数据。

参数:

pdeq_deque	deque 容器。
it_pos	插入数据的位置。
t_count	数据个数。
val_elemlist	用户指定的数据可变参数列表。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 pdeq_deque == NULL 或者 deque 是未初始化的 deque 则函数的行为是未定义的。it_pos 必须是属于 deque 容器的有效位置, 否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
void _deque_init_elem_auxiliary(deque_t* pdeq_deque, void* pv_elem);
```

描述:

使用 deque 的数据类型对数据进行初始化。

参数:

pdeq_deque	deque 容器。
pv_elem	数据。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 pv_elem == NULL 则函数的行为是未定义的。deque 必须是已经初始化或者是使用 create_deque 创建的 deque 容器, 否则函数的行为是未定义的。

辅助接口是提供给实现迭代器接口, 内部接口以及外部接口使用的, 用户不能够直接使用辅助接口。

_deque_is_created	测试 deque 容器是否是由 create_deque 函数创建的。
_deque_is_inited	测试 deque 容器是否已经被初始化。
_deque_iterator_belong_to_deque	测试一个迭代器是否属于指定的 deque 容器。
_deque_same_type	测试两个 deque 保存的数据类型是否相同。
_deque_same_deque_iterator_type	测试 deque 容器中保存的数据类型和迭代器引用的数据类型是否相等。
_deque_get_varg_value_auxiliary	根据 deque 容器中数据的类型获得可变参数列表中的数据。
_deque_destroy_varg_value_auxiliary	销毁从可变参数列表中获得的数据。
_deque_init_elem_range_auxiliary	根据 deque 容器中的数据类型初始化 deque 节点。
_deque_expand_at_end	在 deque 的末尾进行扩展。
_deque_expand_at_begin	在 deque 的开头进行扩展。
_deque_shirk_at_end	在 deque 的末尾删减数据。
_deque_shirk_at_begin	在 deque 的开头删减数据。
_deque_move_elem_to_end	将指定数据区间向末尾移动。
_deque_move_elem_to_begin	将指定数据区间向开头移动。

```
bool_t _deque_is_created(const deque_t* cpdeq_deque);
```

描述:

测试一个 deque 是否是使用 create_deque 创建的。

参数:

cpdeq_deque deque 容器。

返回值:

如果 deque 是使用 create_deque 创建的则返回 true, 否则返回 false。

注意:

如果 cpdeq_deque == NULL, 函数的行为是未定义的。

```
bool_t _deque_is_inited(const deque_t* cpdeq_deque);
```

描述:

测试一个 deque 是否已经初始化。

参数:

cpdeq_deque deque 容器。

返回值:

如果 deque 已经初始化了, 返回 true, 否则返回 false。

注意:

如果 cpdeq_deque == NULL, 函数的行为是未定义的。

```
bool_t _deque_iterator_belong_to_deque(
    const deque_t* cpdeq_deque, deque_iterator_t it_deque);
```

描述:

判断一个迭代器引用的数据是否属于制定的容器。

参数:

cpdeq_deque deque 容器。

it_iter deque 迭代器。

返回值:

如果迭代器引用的数据属于 deque 容器的有效范围内, 返回 true, 否则返回 false。

注意:

如果 cpdeq_deque == NULL, 那么函数的行为是未定义的, 如果 it_iter 不是 deque 迭代器, 那么函数的行为是未定义的, 如果 it_iter 不属于容器 deque, 那么函数的行为是未定义的。

```
bool_t _deque_same_type(const deque_t* cpdeq_first, const deque_t* cpdeq_second);
```

描述:

判断两个 deque 容器中保存的数据类型是否相同。

参数:

cpdeq_first 第一个 deque 容器。

cpdeq_second 第二个 deque 容器。

返回值:

如果两个 deque 中保存的数据类型相同返回 true, 否则返回 false。

注意:

如果 cpdeq_first == NULL 或者 cpdeq_second == NULL, 那么函数的行为是未定义的。两个 deque 容器必须是已经初始化或者使用 create_deque 创建的 deque 容器, 否则函数的行为是未定义的。如果 cpdeq_first == cpdeq_second 则返回

true。

```
bool_t _deque_same_deque_iterator_type(
    const deque_t* cpdeq_deque, deque_iterator_t it_iter);
```

描述:

判断 deque 容器中保存的数据类型与迭代器引用的数据类型是否相同。

参数:

cpdeq_deque	deque 容器。
it_iter	deque 迭代器。

返回值:

如果 deque 中保存的数据类型与 it_iter 引用的数据类型相同返回 true, 否则返回 false。

注意:

如果 cpdeq_deque == NULL 或者 it_iter 不是 deque_iterator_t 类型, 那么函数的行为是未定义的。

```
void _deque_get_varg_value_auxiliary(
    deque_t* pdeq_deque, va_list val_elemlist, void* pv_varg);
```

描述:

从可变参数列表中获得与 deque 中的数据类型相同的数据。

参数:

pdeq_deque	deque 容器。
val_elemlist	可变参数列表。
pv_varg	保存数据缓的冲区。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 pv_varg == NULL, 那么函数的行为是未定义的。pdeq_deque 必须是已经初始化或者是由 create_deque() 创建的 deque 容器, 否则函数的行为是未定义的。

```
void _deque_destroy_varg_value_auxiliary(deque_t* pdeq_deque, void* pv_varg);
```

描述:

销毁与 deque 中的数据类型相同的数据。

参数:

pdeq_deque	deque 容器。
pv_varg	保存数据的缓冲区。

返回值:

无。

注意:

如果 pdeq_deque == NULL 或者 pv_varg == NULL, 那么函数的行为是未定义的。pdeq_deque 必须是已经初始化或者是由 create_deque() 创建的 deque 容器, 否则函数的行为是未定义的。

```
void _deque_init_elem_range_auxiliary(
    deque_t* pdeq_deque, deque_iterator_t it_begin, deque_iterator_t it_end);
```

描述:

将制定范围内的数据初始化为 deque 中的数据类型。

参数:

pdeq_deque	deque 容器。
it_begin	初始化范围的开始。
it_end	初始化范围的末尾。

返回值:

无。

注意:

如果 pdeq_deque == NULL 那么函数的行为是未定义的。pdeq_deque 必须是初始化的或者是使用 create_deque() 创建的，否则函数的行为是未定义的。[it_begin, it_end) 必须是有效的数据区间，pdeq_deque 与 [it_begin, it_end) 保存的数据类型相同，否则函数的行为是未定义的。

```
void* _deque_iterator_get_pointer_auxiliary(deque_iterator_t it_iter);
```

描述:

返回被迭代器应用的数据的指针。

参数:

it_iter	deque 迭代器。
---------	------------

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 必须是属于 deque 的有效迭代器，否则函数的行为是未定义的。

```
deque_iterator_t _deque_expand_at_end(
    deque_t* pdeq_deque, size_t t_expandsize, deque_iterator_t* pit_pos);
```

描述:

在 deque 的末尾进行扩展。

参数:

pdeq_deque	deque 容器。
t_expandsize	扩展的数据的数量。
pit_pos	容器中的某个迭代器的指针。

返回值:

返回扩展前的指向 deque 容器末尾的迭代器。

注意:

如果 pdeq_deque == NULL，那么函数是未定义的。deque 容器必须是已经初始化的，否则函数的行为是未定义的。pit_pos 是扩展之前 deque 容器的某个迭代器的指针，函数执行后这个迭代器也变成扩展之后引用相应数据的迭代器。
*pit_pos 必须是属于 deque 的有效的迭代器，否则函数行为是未定义的。pit_pos 可以等于 NULL。

```
deque_iterator_t _deque_expand_at_begin(
    deque_t* pdeq_deque, size_t t_expandsize, deque_iterator_t* pit_pos);
```

描述:

在 deque 的开头进行扩展。

参数:

pdeq_deque deque 容器。
t_expandsize 扩展的数据的数量。
pit_pos 容器中的某个迭代器的指针。

返回值:

返回扩展前的指向 deque 容器开头的迭代器。

注意:

如果 pdeq_deque == NULL, 那么函数是未定义的。deque 容器必须是已经初始化的, 否则函数的行为是未定义的。 pit_pos 是扩展之前 deque 容器的某个迭代器的指针, 函数执行后这个迭代器也变成扩展之后引用相应数据的迭代器。 *pit_pos 必须是属于 deque 的有效的迭代器, 否则函数行为是未定义的。pit_pos 可以等于 NULL。

```
void _deque_shrink_at_end(deque_t* pdeq_deque, size_t t_shrinkszie);
```

描述:

在 deque 的末尾删减数据。

参数:

pdeq_deque deque 容器。
t_shrinkszie 删减的数据的数量。

返回值:

无。

注意:

如果 pdeq_deque == NULL, 那么函数是未定义的。deque 容器必须是已经初始化的, 否则函数的行为是未定义的。当一个块中的数据全部被删减之后, 这个块将被释放。当 t_shrinkszie > deque_size() 的时候, 将 deque 中的全部数据删除。

```
void _deque_shrink_at_begin(deque_t* pdeq_deque, size_t t_shrinkszie);
```

描述:

在 deque 的开头删减数据。

参数:

pdeq_deque deque 容器。
t_shrinkszie 删减的数据的数量。

返回值:

无。

注意:

如果 pdeq_deque == NULL, 那么函数是未定义的。deque 容器必须是已经初始化的, 否则函数的行为是未定义的。当一个块中的数据全部被删减之后, 这个块将被释放。当 t_shrinkszie > deque_size() 的时候, 将 deque 中的全部数据删除。

```
deque_iterator_t _deque_move_elem_to_end(
    deque_t* pdeq_deque, deque_iterator_t it_begin,
    deque_iterator_t it_end, size_t t_step);
```

描述:

将数据区间[it_begin, it_end)向 deque 容器末尾移动。

参数:

pdeq_deque deque 容器。
it_begin 移动的数据区间的开头。
it_end 移动的数据区间的末尾。
t_step 移动的距离。

返回值:

移动之后空隙数据区间的开头。

注意:

如果 pdeq_deque == NULL, 那么函数是未定义的。deque 容器必须是已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end)必须是属于 pdeq_deque 的有效的数据区间, 否则函数的行为是未定义的。[it_begin, it_end)移动之后也必须是属于 pdeq_deque 的有效的数据区间, 否则函数的行为是未定义的。

```
deque_iterator_t _deque_move_elem_to_begin(
    deque_t* pdeq_deque, deque_iterator_t it_begin,
    deque_iterator_t it_end, size_t t_step);
```

描述:

将数据区间[it_begin, it_end)向 deque 容器开头移动。

参数:

pdeq_deque deque 容器。
it_begin 移动的数据区间的开头。
it_end 移动的数据区间的末尾。
t_step 移动的距离。

返回值:

移动之后空隙数据区间的开头。

注意:

如果 pdeq_deque == NULL, 那么函数是未定义的。deque 容器必须是已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end)必须是属于 pdeq_deque 的有效的数据区间, 否则函数的行为是未定义的。[it_begin, it_end)移动之后也必须是属于 pdeq_deque 的有效的数据区间, 否则函数的行为是未定义的。

第九章 slist 容器

slist 容器是单向链表，它是双向链表 list 的弱化，所以它比 list 的功能要弱，但是它占用的空间更小。与 list 同样在 slist 中插入和删除数据花费常数时间，同时不会引起迭代器失效，但是 slist 是单向的，所以在指定节点前面插入数据要花费线性时间，但是在指定节点后面插入数据是高效的。这就是 slist 与 list 的最大区别。

第一节 slist 容器的机制

slist 是单向链表结构，末尾指向 NULL。

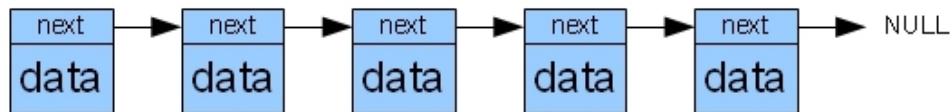


图 9.1 slist 的结构

这样的结构在一个指定的节点后面插入数据是很方便的，但是在指定节点的前面插入数据不是明智之选。此外在实际代码中为了保存这个链表还要引入一个类似 list 中开头节点的哑节点：

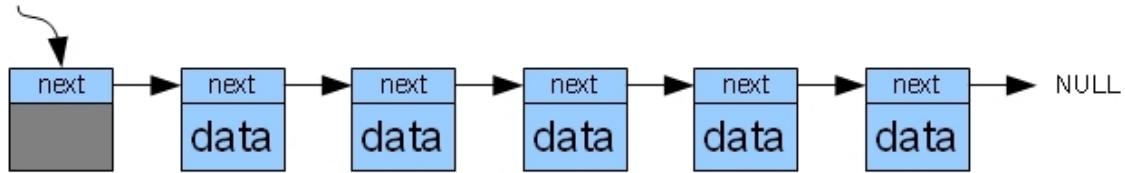


图 9.2 实际的 slist 结构

在 slist 中指定节点之后插入和删除数据要比在 list 中插入数据更方便，因为需要修改的指针更少。由于有了开头的哑元节点，在 slist 的开头插入和删除数据也花费常数时间。但是在 slist 的末尾插入数据就不方便了，要遍历整个链表找到最后一个节点，然后插入数据。

与 list 同样 slist 也有排序算法，但是由于 slist 是单向链表获得前一个节点的效率低下，而快速排序在每次交换数据的时候都要获得前一个节点，所以使用相对低效的插入排序代替。

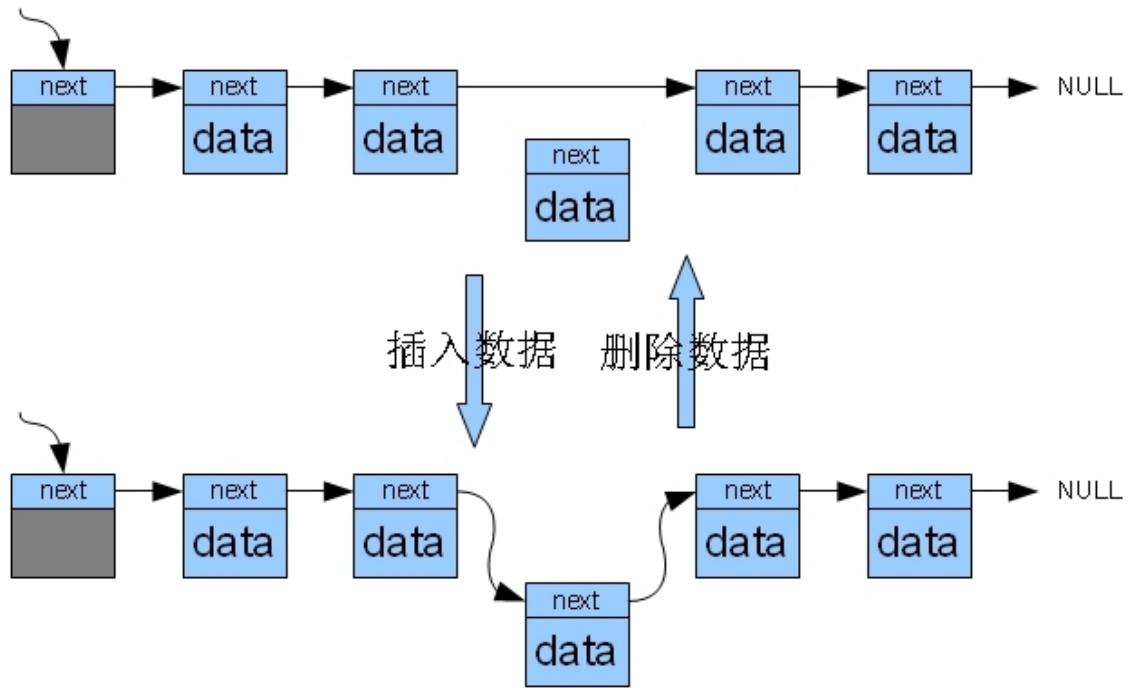


图 9.3 在 slist 中插入或者删除节点

第二节 slist 迭代器

由于 slist 是单向链表，所以它的迭代器也只能是单向迭代器。迭代器的结构也是很简单的，只需要记录每个节点的位置就可以了。

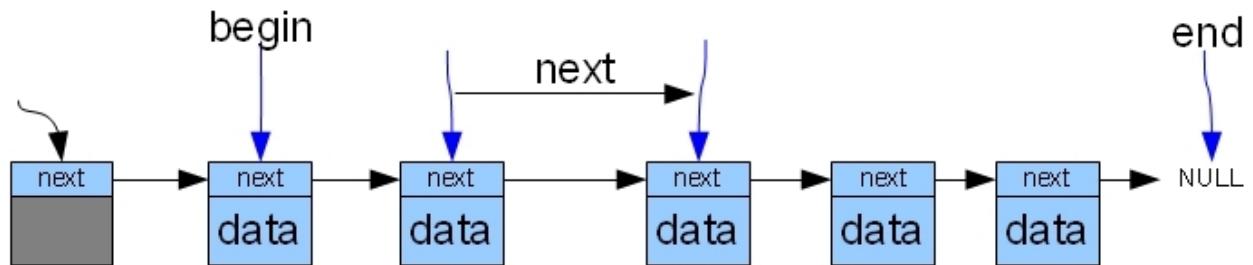


图 9.4 slist 迭代器

这样在迭代器中也不需要太多的结构来表示，所以 slist 的迭代器结构是：

```
typedef struct _tagiterator
{
    /* 容器内部结构的信息 */
    union
    {
        char*      _pc_corepos;    /* 链表容器中数据的位置 */
    }_t_pos;
```

```

void* _pt_container; /* 容器的位置 */
containertype_t _t_containertype; /* 容器的类型 */
iteratortype_t _t_iteratortype; /* 迭代器的类型 */
}iterator_t;

```

第三节 slist 的代码结构

slist 的代码结构分为两部分，一部分是 slist 的节点：

```

typedef struct _tagslistnode
{
    struct _tagslistnode* _pt_next;
    char _pc_data[1];
}slistnode_t;

```

第二部分是 slist 容器结构：

```

typedef struct _tagslist
{
    _typeinfo_t _t_typeinfo;
    _alloc_t _t_allocator;
    slistnode_t _t_head;
}slist_t;

```

实际 slist 结构就是：

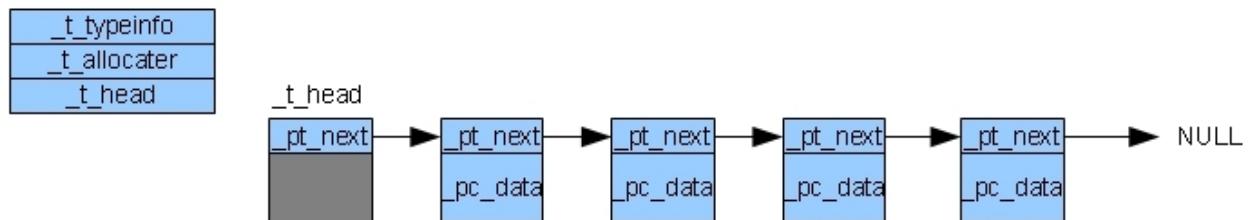


图 9.5 slist 的实际代码结构

第四节 外部接口

slist 容器要提供给用户外部接口：

create_slist	创建 slist 容器。
slist_init	初始化一个空的 slist 容器。
slist_init_n	使用 n 个默认数据初始化 slist 容器。

slist_init_elem	使用 n 个指定数据初始化 slist 容器。
slist_init_copy	使用一个既存的 slist 容器初始化另一个 slist 容器。
slist_init_copy_range	使用指定范围内的数据初始化一个 slist 容器。
slist_destroy	销毁创建的 slist 容器。
slist_size	获得 slist 容器中数据的数量。
slist_empty	测试 slist 容器是否为空。
slist_max_size	返回 slist 容器中保存数据的最大数量的可能值。
slist_equal	测试两个 slist 容器是否相等。
slist_not_equal	测试两个 slist 容器是否不等。
slist_less	测试第一个 slist 容器是否小于第二个 slist 容器。
slist_less_equal	测试第一个 slist 容器是否小于等于第二个 slist 容器。
slist_greater	测试第一个 slist 容器是否大于第二个 slist 容器。
slist_greater_equal	测试第一个 slist 容器是否大于等于第二个 slist 容器。
slist_assign	使用一个既存的 slist 容器为另一个 slist 容器赋值。
slist_assign_elem	使用指定的数据为 slist 容器赋值。
slist_assign_range	使用指定范围内的数据为 slist 容器赋值。
slist_swap	交换两个 slist 容器。
slist_front	访问 slist 容器中的第一个数据。
slist_begin	返回引用 slist 容器的第一个数据的迭代器。
slist_end	返回引用 slist 容器末尾的迭代器。
slist_previous	返回引用前一个节点的迭代器。
slist_insert	向 slist 容器中插入数据。
slist_insert_n	向 slist 容器中插入 n 个数据。
slist_insert_range	向 slist 容器中插入指定范围的数据。
slist_insert_after	向 slist 容器中指定位置的后面插入数据。
slist_insert_after_n	向 slist 容器中指定位置的后面插入 n 个数据。
slist_insert_after_range	向 slist 容器中指定位置的后面插入指定范围的数据。
slist_push_front	向 slist 容器的开头添加一个数据。
slist_pop_front	删除 slist 容器开头的一个数据。
slist_erase	删除 slist 容器中指定位置的一个数据。
slist_erase_range	删除 slist 容器中指定范围内的数据。
slist_erase_after	删除 slist 容器中指定位置的下一个数据。

slist_erase_after_range	删除 slist 容器中指定区间后面的数据。
slist_remove	删除 slist 容器中指定的数据。
slist_remove_if	删除 slist 容器中符合条件的数据。
slist_resize	重新设置 slist 容器中数据的数量。
slist_resize_elem	重新设置 slist 容器中数据的数量，多出的数据使用指定数据填充。
slist_clear	删除 slist 容器中所有的数据。
slist_unique	删除 slist 容器中相邻且相等的数据。
slist_unique_if	删除 slist 容器中相邻且符合条件的数据。
slist_splice	将既存 slist 容器中的数据转移到另一个 slist 容器的指定位置。
slist_splice_pos	将既存的 slist 容器中指定位置的数据转移到另一个 slist 容器的指定位置。
slist_splice_range	将既存 slist 容器中指定范围的数据转移到另一个 slist 容器中的指定位置。
slist_splice_after_pos	将既存的 slist 容器中指定位置的下一个数据转移到另一个 slist 容器中指定位置的后面。
slist_splice_after_range	将既存的 slist 容器中指定的下一个数据区间中的数据转移到另一个 slist 指定位置的后面。
slist_sort	将 slist 容器中的数据排序。
slist_sort_if	将 slist 容器中的数据安装指定的规则排序。
slist_merge	将两个 slist 容器合并。
slist_merge_if	将两个 slist 容器安装指定规则合并。
slist_reverse	将 slist 容器逆序。

函数原型

```
slist_t* create_slist(typename);
```

描述:

创建 slist 容器。

参数:

typename 类型描述。

返回值:

返回创建的 slist 容器的指针。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void slist_init(slist_t* pslist_slist);
```

描述:

初始化一个空的 slist 容器。

参数:

pslist_slist 指向 slist 容器的指针。

返回值:

无。

注意：

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 必须是通过 create_slist 创建的否则函数的行为是未定义的。

```
void slist_init_n(slist_t* pslist_slist, size_t t_count);
```

描述：

使用多个数据默认初始化 slist 容器。

参数：

pslist_slist 指向 slist 容器的指针。
t_count slist 容器中包含的数据个数。

返回值：

无。

注意：

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 必须是通过 create_slist 创建的否则函数的行为是未定义的。初始化后容器中包含的数据都是 slist 容器中保存的数据类型的默认值。

```
void slist_init_elem(slist_t* pslist_slist, size_t t_count, elem);
```

描述：

使用多个指定的数据初始化 slist 容器。

参数：

pslist_slist 指向 slist 容器的指针。
t_count slist 容器中包含的数据个数。
elem 初始化容器的数据。

返回值：

无。

注意：

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 必须是通过 create_slist 创建的否则函数的行为是未定义的。 elem 是与 slist 容器中保存的数据类型相同的数据， 否则函数的行为是未定义的。

```
void slist_init_copy(slist_t* pslist_dest, const slist_t* cpslist_src);
```

描述：

使用既存的 slist 容器初始化另一个 slist 容器。

参数：

pslist_dest 指向被初始化 slist 容器的指针。
cpslist_src 指向既存的 slist 容器的指针。

返回值：

无。

注意：

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 必须是通过 create_slist 创建的否则函数的行为是未定义的。既存的 slist 是有效的 slist 否则函数的行为是未定义的。既存的 slist 要与当前的 slist 保存的数据类型相同， 否则函数的行为是未定义的。初始化之后两个容器中的数据相等。

```
void slist_init_copy_range(
    slist_t* pslist_dest, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

使用指定的数据区间初始化另一个 slist 容器。

参数:

plis_dest	指向被初始化 slist 容器的指针。
it_begin	指定的数据区间的开始。
it_end	指定的数据区间的末尾。

返回值:

无。

注意:

如果 pslist_dest == NULL 则函数的行为是未定义的, slist 必须是通过 create_slist 创建的否则函数的行为是未定义的。[it_begin, it_end) 是有效数据区间, 否则函数的行为是未定义的。数据区间中保存的数据是与 slist 中保存的数据类型相同, 否则函数的行为是未定义的。

```
void slist_destroy(slist_t* pslist_slist);
```

描述:

销毁一个被创建出来的 slist 容器。

参数:

pslist_slist	指向 slist 容器的指针。
--------------	-----------------

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的, slist 必须是通过 create_slist 创建的否则函数的行为是未定义的。不管 slist 容器是否被初始化都需要调用这个函数销毁。

```
size_t slist_size(const slist_t* cpslist_slist);
```

描述:

返回 slist 容器中保存的数据的个数。

参数:

cpslist_slist	指向 slist 容器的指针。
---------------	-----------------

返回值:

slist 容器中保存的数据的个数。

注意:

如果 cpslist_slist == NULL 则函数的行为是未定义的, slist 必须已经初始化的, 否则函数的行为是未定义的。

```
bool_t slist_empty(const slist_t* cpslist_slist);
```

描述:

测试 slist 容器是否为空。

参数:

cpslist_slist	指向 slist 容器的指针。
---------------	-----------------

返回值:

如果 slist 容器为空返回 true, 否则返回 false。

注意:

如果 cpslist_slist == NULL 则函数的行为是未定义的, slist 必须已经初始化的, 否则函数的行为是未定义的。

```
size_t slist_max_size(const slist_t* cpslist_slist);
```

描述:

返回 slist 容器中能够保存的数据最大数量的可能值。

参数:

cpslist_slist 指向 slist 容器的指针。

返回值:

返回 slist 容器中能够保存的数据最大数量的可能值。

注意:

如果 cpslist_slist == NULL 则函数的行为是未定义的, slist 必须已经初始化的, 否则函数的行为是未定义的。这个返回值不是一个固定的值。

```
bool_t slist_equal(const slist_t* cpslist_first, const slist_t* cpslist_second);
```

描述:

测试两个 slist 容器是否相等。

参数:

cpslist_first 指向 slist 容器的指针。

cpslist_second 指向 slist 容器的指针。

返回值:

如果两个 slist 容器相等则返回 true, 否则返回 false。

注意:

如果 cpslist_first == NULL 或者 cpslist_second == NULL 则函数的行为是未定义的, 两个 slist 必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型不同认为这两个容器不等。两个 slist 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cpslist_first == cpslist_second 那么返回 true。

```
bool_t slist_not_equal(const slist_t* cpslist_first, const slist_t* cpslist_second);
```

描述:

测试两个 slist 容器是否不等。

参数:

cpslist_first 指向 slist 容器的指针。

cpslist_second 指向 slist 容器的指针。

返回值:

如果两个 slist 容器不等则返回 true, 否则返回 false。

注意:

如果 cpslist_first == NULL 或者 cpslist_second == NULL 则函数的行为是未定义的, 两个 slist 必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型不同认为两个 slist 容器不等。两个 slist 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cpslist_first == cpslist_second 那么返回 false。

```
bool_t slist_less(const slist_t* cpslist_first, const slist_t* cpslist_second);
```

描述:

测试第一个 slist 是否小于第二个 slist。

参数:

 cpslist_first 指向 slist 容器的指针。
 cpslist_second 指向 slist 容器的指针。

返回值:

如果第一个 slist 容器小于第二个 slist 容器则返回 true, 否则返回 false。

注意:

如果 cpslist_first == NULL 或者 cpslist_second == NULL 则函数的行为是未定义的, 两个 slist 必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 slist 中的数据小于第二个 slist 中对应的数据则返回 true, 如果大于则返回 false, 当两个 slist 中的对应数据相等, 第一个 slist 中的数据数量小于第二个 slist 中数据的数量返回 true, 否则返回 false。如果 cpslist_first == cpslist_second 那么返回 false。

```
bool_t slist_less_equal(  
  const slist_t* cpslist_first, const slist_t* cpslist_second);
```

描述:

测试第一个 slist 是否小于等于第二个 slist。

参数:

 cpslist_first 指向 slist 容器的指针。
 cpslist_second 指向 slist 容器的指针。

返回值:

如果第一个 slist 容器小于等于第二个 slist 容器则返回 true, 否则返回 false。

注意:

如果 cpslist_first == NULL 或者 cpslist_second == NULL 则函数的行为是未定义的, 两个 slist 必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型不同函数的行为是未定义的。如果 cpslist_first == cpslist_second 那么返回 true。

```
bool_t slist_greater(const slist_t* cpslist_first, const slist_t* cpslist_second);
```

描述:

测试第一个 slist 是否大于第二个 slist。

参数:

 cpslist_first 指向 slist 容器的指针。
 cpslist_second 指向 slist 容器的指针。

返回值:

如果第一个 slist 容器大于第二个 slist 容器则返回 true, 否则返回 false。

注意:

如果 cpslist_first == NULL 或者 cpslist_second == NULL 则函数的行为是未定义的, 两个 slist 必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 slist 中的数据大于第二个 slist 中对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 slist 中数据的数量大于第二个 slist 中数据的数量的时候返回 true 否则返回 false。如果 cpslist_first == cpslist_second 那么返回 false。

```
bool_t slist_greater_equal(  
    const slist_t* cpslist_first, const slist_t* cpslist_second);
```

描述:

测试第一个 slist 是否大于等于第二个 slist。

参数:

cpslist_first 指向 slist 容器的指针。
cpslist_second 指向 slist 容器的指针。

返回值:

如果第一个 slist 容器大于等于第二个 slist 容器则返回 true, 否则返回 false。

注意:

如果 cpslist_first == NULL 或者 cpslist_second == NULL 则函数的行为是未定义的, 两个 slist 必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型不同函数的行为是未定义的。如果 cpslist_first == cpslist_second 那么返回 true。

```
void slist_assign(slist_t* pslist_dest, const slist_t* cpslist_src);
```

描述:

使用一个 slist 为另一个 slist 赋值。

参数:

pslist_dest 指向被赋值的 slist 容器的指针。
cpslist_src 指向赋值的 slist 容器的指针。

返回值:

无。

注意:

如果 pslist_dest == NULL 或者 cpslist_src == NULL 则函数的行为是未定义的, 两个 slist 必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型不同函数的行为是未定义的。如果 slist_equal(pslist_dest, cpslist_src)那么函数不做任何动作。

```
void slist_assign_elem(slist_t* pslist_slist, size_t t_count, elem);
```

描述:

使用指定的数据为 slist 赋值。

参数:

pslist_slist 指向被赋值的 slist 容器的指针。
t_count 赋值的数据的个数。
elem 赋值的数据。

返回值:

无。

注意:

如果 pslist_slist== NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的 。 elem 是与 slist 中保存的数据类型相同的数据, 否则函数的行为是未定义的。

```
void slist_assign_range(
```

```
slist_t* pslist_slist, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

使用指定的数据区间为 slist 赋值。

参数:

pslist_slist	指向被赋值的 slist 容器的指针。
it_begin	指定的数据区间的开始。
it_end	指定的数据区间的末尾。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end) 是有效数据区间否则函数的行为是未定义的。[it_begin, it_end) 中保存的数据是与 slist 中保存的数据类型相同的数据, 否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 pslist_slist 则函数的行为是未定义的。

```
void slist_swap(slist_t* pslist_first, slist_t* pslist_second);
```

描述:

交换两个 slist 容器。

参数:

pslist_first	指向 slist 容器的指针。
pslist_second	指向 slist 容器的指针。

返回值:

无。

注意:

如果 pslist_first == NULL 或者 pslist_second == NULL 则函数的行为是未定义的, 两个 slist 容器必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型必须是一致的, 否则函数的行为是未定义的。如果 slist_equal(pslist_firt, pslist_second) 则函数不执行任何动作。

```
void* slist_front(const slist_t* cpslist_slist);
```

描述:

访问 slist 容器中的第一个数据

参数:

cpslist_slist	指向 slist 容器的指针。
---------------	-----------------

返回值:

指向 slist 容器中第一个数据的指针。

注意:

如果 cpslist_slist == NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的。如果 slist 为空则函数的行为是未定义的。

```
slist_iterator_t slist_begin(const slist_t* cpslist_slist);
```

描述:

获得引用 slist 容器中第一数据的迭代器。

参数:

`cpslist_slist` 指向 slist 容器的指针。

返回值:

返回引用 slist 容器中第一个数据的迭代器。

注意:

如果 `cpslist_slist == NULL` 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。

如果 slist 容器为空，则返回值与 `slist_end(cpslist_slist)` 相等。

```
slist_iterator_t slist_end(const slist_t* cpslist_slist);
```

描述:

获得引用 slist 容器末尾的迭代器。

参数:

`cpslist_slist` 指向 slist 容器的指针。

返回值:

返回引用 slist 容器末尾的迭代器。

注意:

如果 `cpslist_slist == NULL` 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。

```
slist_iterator_t slist_previous(
    const slist_t* cpslist_slist, slist_iterator_t it_pos);
```

描述:

获得引用当前节点的前一个节点的迭代器。

参数:

`cpslist_slist` 指向 slist 容器的指针。

`it_pos` slist 迭代器。

返回值:

引用当前节点的前一个节点的迭代器。

注意:

如果 `cpslist_slist == NULL` 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。

`it_pos` 必须是属于 slist 容器的有效的迭代器，否则函数的行为是未定义的。引用当前数据的前一个数据的迭代器也必须是有效的迭代器，否则函数的行为是未定义的。

```
slist_iterator_t slist_insert(slist_t* pslist_slist, slist_iterator_t it_pos, elem);
```

描述:

在 slist 容器的指定位置插入数据。

参数:

`pslist_slist` 指向 slist 容器的指针。

`it_pos` 插入数据的位置。

`elem` 被插入的数据。

返回值:

返回引用被插入的数据的迭代器。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。

`it_pos` 必须是属于 `pslist_slist` 容器的是有效迭代器，否则函数的行为是未定义的。`elem` 是与 `slist` 容器中保存的数据类型一致的数据，否则函数的行为是未定义的。

```
slist_iterator_t slist_insert_n(
    slist_t* pslist_slist, slist_iterator_t it_pos, size_t t_count, elem);
```

描述:

在 `slist` 容器的指定位置插入多个数据。

参数:

`pslist_slist` 指向 `slist` 容器的指针。
`it_pos` 插入数据的位置。
`t_count` 插入的数据的个数。
`elem` 被插入的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的，`slist` 容器必须已经初始化的，否则函数的行为是未定义的。
`it_pos` 必须是属于 `pslist_slist` 容器的是有效迭代器，否则函数的行为是未定义的。`elem` 是与 `slist` 容器中保存的数据类型一致的数据，否则函数的行为是未定义的。

```
void slist_insert_range(
    slist_t* pslist_slist, slist_iterator_t it_pos,
    slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

在 `slist` 容器的指定位置插入一个数据区间。

参数:

`pslist_slist` 指向 `slist` 容器的指针。
`it_pos` 插入数据的位置。
`it_begin` 插入的数据区间的开头。
`it_end` 插入的数据区间的末尾。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的，`slist` 容器必须已经初始化的，否则函数的行为是未定义的。
`it_pos` 必须是属于 `pslist_slist` 容器的是有效迭代器，否则函数的行为是未定义的。`[it_begin, it_end)` 必须是有效的数据区间，否则函数的行为是未定义的，`[it_begin, it_end)` 中保存的数据类型与 `slist` 中保存的数据类型一致，否则函数的行为是未定义的。如果 `[it_begin, it_end)` 属于 `pslist_slist` 则函数的行为是未定义的。

```
slist_iterator_t slist_insert_after(
    slist_t* pslist_slist, slist_iterator_t it_pos, elem);
```

描述:

在 `slist` 容器的指定位置后面插入数据。

参数:

pslist_slist 指向 slist 容器的指针。
it_pos 插入数据的位置。
elem 被插入的数据。

返回值：

返回引用被插入的数据的迭代器。

注意：

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 容器必须已经初始化的，否则函数的行为是未定义的。 it_pos 必须是属于 pslist_slist 容器的是有效迭代器，否则函数的行为是未定义的。 elem 是与 slist 容器中保存的数据类型一致的数据，否则函数的行为是未定义的。

```
list_iterator_t slist_insert_after_n(
    list_t* pslist_slist, list_iterator_t it_pos, size_t t_count, elem);
```

描述：

在 slist 容器的指定位置后面插入多个数据。

参数：

pslist_slist 指向 slist 容器的指针。
it_pos 插入数据的位置。
t_count 插入的数据的个数。
elem 被插入的数据。

返回值：

返回引用被插入的第一个数据的迭代器。

注意：

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 容器必须已经初始化的，否则函数的行为是未定义的。 it_pos 必须是属于 pt_slist 容器的是有效迭代器，否则函数的行为是未定义的。 elem 是与 slist 容器中保存的数据类型一致的数据，否则函数的行为是未定义的。

```
void slist_insert_after_range(
    list_t* pslist_slist, list_iterator_t it_pos,
    list_iterator_t it_begin, list_iterator_t it_end);
```

描述：

在 slist 容器的指定位置后面插入一个数据区间。

参数：

pslist_slist 指向 slist 容器的指针。
it_pos 插入数据的位置。
it_begin 插入的数据区间的开头。
it_end 插入的数据区间的末尾。

返回值：

无。

注意：

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 容器必须已经初始化的，否则函数的行为是未定义的。 it_pos 必须是属于 pslist_slist 容器的是有效迭代器，否则函数的行为是未定义的。 [it_begin, it_end) 必须是有效的数据区间，否则函数的行为是未定义的， [it_begin, it_end) 中保存的数据类型与 slist 中保存的数据类型一致，否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 pslist_slist 则函数的行为是未定义的。

```
void slist_push_front(slist_t* pslist_slist, elem);
```

描述:

向 slist 开头添加数据。

参数:

pslist_slist 指向 slist 容器的指针。

elem 添加的数据。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的。 elem 是与 slist 容器中保存的数据类型一致的数据, 否则函数的行为是未定义的。

```
void slist_pop_front(slist_t* pslist_slist);
```

描述:

删除 slist 容器中的第一个数据

参数:

pslist_slist 指向 slist 容器的指针。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的。如果 slist 容器为空则函数的行为是未定义的。

```
slist_iterator_t slist_erase(slist_t* pslist_slist, slist_iterator_t it_pos);
```

描述:

删除 slist 容器中指定位置的数据。

参数:

pslist_slist 指向 slist 容器的指针。

it_pos 被删除的数据的位置。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的。 it_pos 是属于 pslist_slist 容器的有效的迭代器, 否则函数的行为是未定义的。

```
slist_iterator_t slist_erase_range(  
  slist_t* pslist_slist, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

删除 slist 容器中指定数据区间的数据。

参数:

pslist_slist 指向 slist 容器的指针。

it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 容器必须已经初始化的， 否则函数的行为是未定义的。
[it_begin, it_end)是属于 pslist_slist 的有效数据区间， 否则函数的行为是未定义的。

```
slist_iterator_t slist_erase_after(slist_t* pslist_slist, slist_iterator_t it_pos);
```

描述:

删除 slist 容器中指定位置后面的数据。

参数:

pslist_slist 指向 slist 容器的指针。
it_pos 被删除的数据的位置。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 容器必须已经初始化的， 否则函数的行为是未定义的 。
it_pos 是属于 pslist_slist 容器的有效迭代器并且 it_pos 不等于 slist_end()， 否则函数的行为是未定义的。如果 it_pos 等于 slist_previous(slist_end())那么函数不执行任何操作并且返回 slist_end()。

```
slist_iterator_t slist_erase_after_range(  
    slist_t* pslist_slist, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

删除 slist 容器中指定数据区间下一个位置的数据。

参数:

pslist_slist 指向 slist 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

返回引用被删除的数据后面的数据的迭代器。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的， slist 容器必须已经初始化的， 否则函数的行为是未定义的。
[it_begin, it_end)是属于 pslist_slist 的有效数据区间并且 it_begin 不等于 slist_end()， 否则函数的行为是未定义的。如果 it_begin() == it_end()或者 it_begin() + 1 == it_end()则函数不执行任何操作。

```
void slist_remove(slist_t* pslist_slist, elem);
```

描述:

删除 slist 容器中指定的数据。

参数:

pslist_slist 指向 slist 容器的指针。
elem 要删除的数据。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的，`slist` 容器必须已经初始化的，否则函数的行为是未定义的。`elem` 是与 `slist` 中保存的数据类型一致的数据，否则函数的行为是未定义的。如果 `slist` 容器中包含 `elem` 则删除，如果不包含 `elem`，这个函数不做任何操作。

```
void slist_remove_if(slist_t* pslist_slist, unary_function_t ufun_op);
```

描述:

删除 `slist` 容器中符合条件的数据。

参数:

`pslist_slist` 指向 `slist` 容器的指针。

`ufun_op` 删除数据的条件。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的，`slist` 容器必须已经初始化的，否则函数的行为是未定义的。`ufun_op` 是一个一元谓词，如果 `ufun_op == NULL` 则使用默认的一元谓词。

```
void slist_resize(slist_t* pslist_slist, size_t t_resize);
```

描述:

重新设置 `slist` 中数据的数量。

参数:

`pslist_slist` 指向 `slist` 容器的指针。

`t_resize` `slist` 中数据的新的数量。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的，`slist` 容器必须已经初始化的，否则函数的行为是未定义的。如果重新设置的数据的数量超过了原有的数据的数量，那么添加的部分使用 `slist` 容器中保存的数据类型的默认值来填充。

```
void slist_resize_elem(slist_t* pslist_slist, size_t t_resize, elem);
```

描述:

重新设置 `slist` 中数据的数量，超出的部分使用 `elem` 来填充。

参数:

`pslist_slist` 指向 `slist` 容器的指针。

`t_resize` `slist` 中数据的新的数量。

`elem` 填充数据。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的，`slist` 容器必须已经初始化的，否则函数的行为是未定义的。如

果重新设置的数据的数量超过了原有的数据的数量，那么添加的部分 elem 来填充。elem 必须是与 slist 容器中保存的数据的类型一致的数据，否则函数的行为是未定义的。

```
void slist_clear(slist_t* pslist_slist);
```

描述:

删除 slist 中全部的数据。

参数:

pslist_slist 指向 slist 容器的指针。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。

```
void slist_unique(slist_t* pslist_slist);
```

描述:

使 slist 容器中的数据唯一。

参数:

pslist_slist 指向 slist 容器的指针。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。这个函数删除掉相邻且相等的数据的后面一个，但是不相邻的数据则不删除。

```
void slist_unique_if(slist_t* pslist_slist, binary_function_t bfun_op);
```

描述:

使 slist 容器中的符合条件的数据唯一。

参数:

pslist_slist 指向 slist 容器的指针。

bfun_op 使数据唯一的条件。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。这个函数删除掉相邻且符合条件的数据的后面一个，但是不相邻的数据则不删除。bfun_op 是二元谓词，如果 bfun_op == NULL 那么使用默认的二元谓词。

```
void slist_splice(
    slist_t* pslist_slist, slist_iterator_t it_pos, slist_t* pslist_src);
```

描述:

将既存 slist 中的全部数据转移到另一个 slist 的指定位置。

参数:

`pslist_slist` 指向 slist 容器的指针。
`it_pos` 插入数据的位置。
`pslist_src` 转移数据的 slist 容器。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `pslist_src == NULL` 则函数的行为是未定义的，两个 slist 容器必须已经初始化的，否则函数的行为是未定义的。`it_pos` 是属于 `pslist_slist` 的有效的迭代器，否则函数的行为是未定义的。`pslist_src` 中保存的数据是与 `pslist_slist` 中保存的数据类型一致，否则函数的行为是未定义的。当 `pslist_slist == pslist_src` 时，函数不执行任何动作。

```
void slist_splice_pos(
    slist_t* pslist_slist, slist_iterator_t it_pos,
    slist_t* pslist_src, slist_iterator_t it_src);
```

描述:

将既存 slist 中的指定位置的数据转移到另一个 slist 的指定位置。

参数:

`pslist_slist` 指向 slist 容器的指针。
`it_pos` 插入数据的位置。
`pslist_src` 转移数据的 slist 容器。
`it_src` 转移数据的位置。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `pslist_src == NULL` 则函数的行为是未定义的，两个 slist 容器必须已经初始化的，否则函数的行为是未定义的。`it_pos` 是属于 `pt_slist` 的有效的迭代器，且 `it_src` 是属于 `pt_slistsrc` 的有效迭代器，否则函数的行为是未定义的。`pslist_src` 中保存的数据是与 `pslist_slist` 中保存的数据类型一致，否则函数的行为是未定义的。当 `pslist_slist == pslist_src` 时，并且 `it_pos == it_src` 或者 `it_pos == iterator_next(it_src)`，函数不进行任何操作。

```
void slist_splice_range(
    slist_t* pslist_slist, slist_iterator_t it_pos,
    slist_t* pslist_src, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

将既存 slist 中的指定的数据区间转移到另一个 slist 的指定位置。

参数:

`pslist_slist` 指向 slist 容器的指针。
`it_pos` 插入数据的位置。
`pslist_src` 转移数据的 slist 容器。
`it_begin` 转移的数据区间的开头。
`it_end` 转移的数据区间的末尾。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `pslist_src == NULL` 则函数的行为是未定义的，两个 `slist` 容器必须已经初始化的，否则函数的行为是未定义的。`it_pos` 是属于 `pslist_slist` 的有效的迭代器，且 $[it_begin, it_end)$ 是属于 `pslist_src` 的有效数据区间，否则函数的行为是未定义的。`pslist_src` 中保存的数据是与 `pslist_slist` 中保存的数据类型一致，否则函数的行为是未定义的。`it_pos` 不能属于 $[it_begin, it_end)$ ，否则函数的行为是未定义的。当 `pslist_slist == pslist_src` 时，并且 `it_pos == it_begin` 或者 `it_pos == it_end` 函数不进行任何操作。

```
void slist_splice_after_pos(
    slist_t* pslist_slist, slist_iterator_t it_pos,
    slist_t* pslist_src, slist_iterator_t it_src);
```

描述:

将既存 `slist` 中的指定位置后面的数据转移到另一个 `slist` 的指定位置后面。

参数:

`pslist_slist` 指向 `slist` 容器的指针。
`it_pos` 插入数据的位置。
`pslist_src` 转移数据的 `slist` 容器。
`it_src` 转移数据的位置。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `pslist_src == NULL` 则函数的行为是未定义的，两个 `slist` 容器必须已经初始化的，否则函数的行为是未定义的。`it_pos+1` 是属于 `pslist_slist` 的有效的迭代器，且 `it_src+1` 是属于 `pslist_src` 的有效迭代器，否则函数的行为是未定义的。`pslist_src` 中保存的数据是与 `pslist_slist` 中保存的数据类型一致，否则函数的行为是未定义的。当 `pslist_slist == pslist_src` 时，并且 `it_pos == it_src` 或者 `it_pos == iterator_next(it_src)`，函数不进行任何操作。当 `it_src == slist_end()` 时，函数不执行任何操作。

```
void slist_splice_after_range(
    slist_t* pslist_slist, slist_iterator_t it_pos,
    slist_t* pslist_src, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

将既存 `slist` 中的指定的数据区间下面的数据转移到另一个 `slist` 的指定位置后面。

参数:

`pslist_slist` 指向 `slist` 容器的指针。
`it_pos` 插入数据的位置。
`pslist_src` 转移数据的 `slist` 容器。
`it_begin` 转移的数据区间的开头。
`it_end` 转移的数据区间的末尾。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `pslist_src == NULL` 则函数的行为是未定义的，两个 `slist` 容器必须已经初始化的，否则函数的行为是未定义的。`it_pos+1` 是属于 `pslist_slist` 的有效的迭代器，且 $[it_begin+1, it_end+1)$ 是属于 `pslist_src` 的有效数据区间，否则函数的行为是未定义的。`pslist_src` 中保存的数据是与 `pslist_slist` 中保存的数据类型一致，否则函数的行为是未定义的。`it_pos` 不能属于 $[it_begin, it_end)$ ，否则函数的行为是未定义的。当 `pslist_slist == pslist_src` 时，并且

it_pos == it_begin 或者 it_pos == it_end 函数不进行任何操作。

```
void slist_sort(slist_t* pslist_slist);
```

描述:

将 slist 中的数据按照由小到大的顺序排序。

参数:

pslist_slist 指向 slist 容器的指针。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的。

```
void slist_sort_if(slist_t* pt_slist, binary_function_t bfun_op);
```

描述:

将 slist 中的数据按照指定的规则排序。

参数:

pslist_slist 指向 slist 容器的指针。

bfun_op 排序规则。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的, slist 容器必须已经初始化的, 否则函数的行为是未定义的。 bfun_op 是一个表示比较的二元谓词。如果 bfun_op == NULL 使用 slist 中保存的数据类型默认的比较函数。

```
void slist_merge(slist_t* pslist_dest, slist_t* pslist_src);
```

描述:

将两个有序的 slist 合并。

参数:

pslist_dest 指向 slist 容器的指针。

pslist_src 指向 slist 容器的指针。

返回值:

无。

注意:

如果 pslist_dest == NULL 或者 pslist_src == NULL 则函数的行为是未定义的, 两个 slist 容器必须已经初始化的, 否则函数的行为是未定义的。两个 slist 容器中保存的数据类型必须是一致的, 否则函数的行为是未定义的。两个 slist 容器必须都是按照默认比较规则排序的, 否则函数的行为是未定义的。如果 pslist_dest == pslist_src 则函数不执行任何操作。

```
void slist_merge_if(
    slist_t* pslist_dest, slist_t* pslist_src, binary_function_t bfun_op);
```

描述:

将两个按照指定规则排序的 slist 合并。

参数:

pslist_dest	指向 slist 容器的指针。
pslist_src	指向 slist 容器的指针。
bfun_op	指定的排序规则。

返回值:

无。

注意:

如果 pslist_dest == NULL 或者 pslist_src == NULL 则函数的行为是未定义的，两个 slist 容器必须已经初始化的，否则函数的行为是未定义的。两个 slist 容器中保存的数据类型必须是一致的，否则函数的行为是未定义的。两个 slist 容器必须都是按照默认比较规则排序的，否则函数的行为是未定义的。如果 pslist_dest == pslist_src 则函数不执行任何操作。bfun_op 是表示排序规则的二元谓词，如果 bfun_op == NULL 则使用 slist 容器中保存的数据类型的默认比较函数。

```
void slist_reverse(slist_t* pslist_slist);
```

描述:

将 slist 中的数据逆序。

参数:

pslist_slist	指向 slist 容器的指针。
--------------	-----------------

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的，slist 容器必须已经初始化的，否则函数的行为是未定义的。

第五节 迭代器接口

slist 的迭代器是单向的迭代器，所以它没有随机访问迭代器和双向迭代器那么强大的功能，向前移动只能单步运行。这些接口只能由迭代器接口使用，用户不应该直接使用这些接口函数。

_create_slist_iterator	创建 slist 迭代器。
_slist_iterator_get_value	获得 slist 迭代器引用的数据。
_slist_iterator_set_value	设置 slist 迭代器引用的数据。
_slist_iterator_get_pointer	获得 slist 迭代器引用的数据的指针。
_slist_iterator_next	获得引用下一个数据的迭代器。
_slist_iterator_equal	测试两个迭代器是否相等。
_slist_iterator_distance	计算两个迭代器之间的距离。
_slist_iterator_before	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
slist_iterator_t _create_slist_iterator(void);
```

描述:

 创建一个 slist 迭代器。

参数:

 无。

返回值:

 slist 迭代器。

注意:

 返回的 slist 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _slist_iterator_get_value(slist_iterator_t it_iter, void* pv_value);
```

描述:

 获得迭代器引用的数据。

参数:

 it_iter slist 迭代器。

 pv_value 保存数据的缓冲区。

返回值:

 无。

注意:

 it_iter 是有效的 slist 迭代器，否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的，pv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
void _slist_iterator_set_value(slist_iterator_t it_iter, const void* cpv_value);
```

描述:

 设置迭代器引用的数据。

参数:

 it_iter slist 迭代器。

 cpv_value 保存数据的缓冲区。

返回值:

 无。

注意:

 it_iter 是有效的 slist 迭代器，否则函数的行为是未定义的。cpv_value == NULL 则函数的行为是未定义的，cpv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 cpv_value 中保存的数据被拷贝到迭代器引用的数据中。

```
const void* _slist_iterator_get_pointer(slist_iterator_t it_iter);
```

描述:

 获得迭代器引用的数据的指针。

参数:

 it_iter slist 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 slist 迭代器, 否则函数的行为是未定义的。

```
const slist_iterator_t & slist_iterator_next(slist_iterator_t t_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter slist 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 slist 迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 slist 有效的迭代器, 否则函数的行为是未定义的。

```
bool_t & slist_iterator_equal(
    const slist_iterator_t & it_first, const slist_iterator_t & it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first slist 迭代器。

it_second slist 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 slist 容器的有效的 slist 迭代器, 否则函数的行为是未定义的。两个 slist 迭代器相等是指两个迭代器引用相同的数据。

```
int & slist_iterator_distance(
    const slist_iterator_t & it_first, const slist_iterator_t & it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_first slist 迭代器。

it_second slist 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 slist 容器的有效的 slist 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为 0。

```
bool_t & slist_iterator_before(
```

```
slist_iterator_t it_first, slist_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first slist 迭代器。

it_second slist 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 slist 容器的有效的 slist 迭代器, 否则函数的行为是未定义的。

第六节 内部和辅助接口

内部接口是提供给实现外部接口使用的, 用户不能够直接使用内部接口。

_create_slist	创建一个 slist 容器。
_create_slist_auxiliary	创建 slist 容器的辅助函数。
_slist_init_elem	使用指定的数据初始化 slist 容器。
_slist_init_elem_varg	使用可变参数列表中的指定数据初始化 slist 容器。
_slist_destroy_auxiliary	slist 容器销毁辅助函数。
_slist_assign_elem	使用指定的数据为 slist 赋值。
_slist_assign_elem_varg	使用可变参数列表中的数据为 slist 赋值。
_slist_push_front	向 slist 开头添加一个指定数据。
_slist_push_front_varg	向 slist 开头添加一个可变参数列表中指定的数据。
_slist_resize_elem	重新设置 slist 中数据的个数, 使用指定的数据填充。
_slist_resize_elem_varg	重新设置 slist 中数据的个数, 使用可变参数列表中指定的数据。
_slist_remove	从 slist 中删除指定数据。
_slist_remove_varg	从 slist 中删除指定数据。
_slist_insert	向 slist 的指定位置插入指定的数据。
_slist_insert_n	向 slist 的指定位置插入多个指定的数据。
_slist_insert_after	向 slist 的指定位置后面插入指定的数据。
_slist_insert_after_n	向 slist 的指定位置后面插入多个指定的数据。
_slist_insert_after_n_varg	向 slist 的指定位置插后面入可变参数列表指定的数据。
_slist_init_elem_auxiliary	初始化数据节点的辅助函数。

```
slist_t* _create_slist(const char* s_typename);
```

描述:

创建一个 slist 容器。

参数:

s_typename slist 容器中保存的数据类型。

返回值:

创建成功返回指向容器的指针, 否则返回 NULL。

注意:

如果 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
bool_t _create_slist_auxiliary(slist_t* pslist_slist, const char* s_typename);
```

描述:

创建一个 slist 容器的辅助函数。

参数:

pslist_slist slist 容器

s_typename slist 容器中保存的数据类型。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 pslist_slist == NULL 或者 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _slist_init_elem(slist_t* pslist_slist, size_t t_count, ...);
```

描述:

使用用户指定的数据初始化 slist 容器。

参数:

pslist_slist 未初始化的 slist 容器。

t_count 数据个数。

... 用户指定的数据。

返回值:

无。

注意:

如果 pslist_slist == NULL 则函数的行为是未定义的。slist 容器必须是使用 create_slist 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
void _slist_init_elem_varg(
    slist_t* pslist_slist, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据初始化 slist 容器。

参数:

`pslist_slist` 未初始化的 slist 容器。
`t_count` 数据个数。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 则函数的行为是未定义的。slist 容器必须是使用 `create_slist` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_destroy_auxiliary(slist_t* pslist_slist);
```

描述:

销毁 slist 容器的辅助函数。

参数:

`pslist_slist` slist 容器。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `slist` 不是使用 `create_slist` 生成的则函数的行为是未定义的。

```
void _slist_assign_elem(slist_t* pslist_slist, size_t t_count, ...);
```

描述:

使用用户指定的数据为 slist 容器赋值。

参数:

`pslist_slist` slist 容器。
`t_count` 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `slist` 是未初始化的 slist 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_assign_elem_varg(
    slist_t* pslist_slist, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据为 slist 容器赋值。

参数:

`pslist_slist` slist 容器。
`t_count` 数据个数。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `slist` 是未初始化的 `slist` 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_push_front(slist_t* pslist_slist, ...);
```

描述:

向 `slist` 容器开头添加一个数据。

参数:

`pslist_slist` `slist` 容器。
... 用户指定的数据。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `slist` 是未初始化的 `slist` 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_push_front_varg(slist_t* pslist_slist, va_list val_elemlist);
```

描述:

向 `slist` 容器开头添加一个数据。

参数:

`pslist_slist` `slist` 容器。
`val_elemlist` 可变参数列表。

返回值:

无。

注意:

如果 `pslist_slist == NULL` 或者 `slist` 是未初始化的 `slist` 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_resize_elem(slist_t* pslist_slist, size_t t_resize, ...);
```

描述:

重新设置 `slist` 容器中的数据个数，不足的部分使用指定数据填充。

参数:

`pslist_slist` `slist` 容器。
`t_resize` 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用指定的数据填充。

```
void _slist_resize_elem_varg(
    slist_t* pslist_slist, size_t t_resize, va_list val_elemlist);
```

描述:

重新设置 slist 容器中的数据个数，不足的部分使用可变参数列表中的数据填充。

参数:

pslist_slist	slist 容器。
t_resize	数据个数。
val_elemlist	用户指定的数据的参数列表。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用可变参数列表中的数据填充。

```
void _slist_remove(slist_t* pslist_slist, ...);
```

描述:

从 slist 容器中删除指定的数据。

参数:

pslist_slist	slist 容器。
...	用户指定的数据。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。容器中所有等于指定数据的数据将会被删除，如果没有与指定数据相等的数据则不删除。

```
void _slist_remove_varg(slist_t* pslist_slist, va_list val_elemlist);
```

描述:

从 slist 容器中删除指定的数据。

参数:

pslist_slist	slist 容器。
val_elemlist	可变参数列表。

返回值:

无。

注意：

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。容器中所有等于指定数据的数据将会被删除，如果没有与指定数据相等的数据则不删除。

```
slist_iterator_t _slist_insert(
    slist_t* pslist_slist, slist_iterator_t it_pos, ...);
```

描述：

向 slist 容器的指定位置插入指定数据。

参数：

pslist_slist	slist 容器。
it_pos	插入数据的位置。
...	用户指定的数据。

返回值：

返回引用被插入的数据的迭代器。

注意：

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。it_pos 必须是属于 slist 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_insert_n(
    slist_t* pslist_slist, slist_iterator_t it_pos, size_t t_count, ...);
```

描述：

向 slist 容器的指定位置插入多个数据。

参数：

pslist_slist	slist 容器。
it_pos	插入数据的位置。
t_count	数据个数。
...	用户指定的数据。

返回值：

无。

注意：

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。it_pos 必须是属于 slist 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
slist_iterator_t _slist_insert_after(
    slist_t* pslist_slist, slist_iterator_t it_pos, ...);
```

描述：

向 slist 容器的指定位置后面插入指定数据。

参数：

pslist_slist slist 容器。
it_pos 插入数据的位置。
... 用户指定的数据。

返回值:

返回引用被插入的数据的迭代器。

注意:

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。it_pos 必须是属于 slist 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_insert_after_n(
    slist_t* pslist_slist, slist_iterator_t it_pos, size_t t_count, ...);
```

描述:

向 slist 容器的指定位置后面插入多个数据。

参数:

pslist_slist slist 容器。
it_pos 插入数据的位置。
t_count 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。it_pos 必须是属于 slist 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_insert_after_n_varg(
    slist_t* pslist_slist, slist_iterator_t it_pos, size_t t_count,
    va_list val_elemlist);
```

描述:

向 slist 容器的指定位置插入多个数据。

参数:

pslist_slist slist 容器。
it_pos 插入数据的位置。
t_count 数据个数。
val_elemlist 用户指定的数据可变参数列表。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 slist 是未初始化的 slist 则函数的行为是未定义的。it_pos 必须是属于 slist 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _slist_init_elem_auxiliary(slist_t* pslist_slist, void* pv_elem);
```

描述:

使用 slist 的数据类型对数据进行初始化。

参数:

pslist_slist	slist 容器。
pv_elem	数据。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 pv_elem == NULL 则函数的行为是未定义的。slist 必须是已经初始化或者是使用 create_slist 创建的 slist 容器，否则函数的行为是未定义的。

辅助接口是提供给实现迭代器接口，内部接口以及外部接口使用的，用户不能够直接使用辅助接口。

_slist_is_created	测试 slist 容器是否是由 create_slist 函数创建的。
_slist_is_inited	测试 slist 容器是否已经被初始化。
_slist_iterator_belong_to_slist	测试一个迭代器是否属于指定的 slist 容器。
_slist_same_type	测试两个 slist 保存的数据类型是否相同。
_slist_same_slist_iterator_type	测试 slist 容器中保存的数据类型和迭代器引用的数据类型是否相等。
_slist_get_varg_value_auxiliary	根据 slist 容器中数据的类型获得可变参数列表中的数据。
_slist_destroy_varg_value_auxiliary	销毁从可变参数列表中获得的数据。
_slist_init_node_auxiliary	根据 slist 容器中的数据类型初始化 slist 节点。
_slist_transfer	将数据区间内的数据转移到指定位置。
_slist_transfer_after	将数据区间内的数据转移到指定位置后面。

函数原型

```
bool_t _slist_is_created(const slist_t* cpslist_slist);
```

描述:

测试一个 slist 是否是使用 create_slist 创建的。

参数:

cpslist_slist	slist 容器。
---------------	-----------

返回值:

如果 slist 是使用 create_slist 创建的则返回 true，否则返回 false。

注意:

如果 cpslist_slist == NULL，函数的行为是未定义的。

```
bool_t _slist_is_inited(const slist_t* cpslist_slist);
```

描述:

测试一个 slist 是否已经初始化。

参数:

 cpslist_slist slist 容器。

返回值:

 如果 slist 已经初始化了, 返回 true, 否则返回 false。

注意:

 如果 cpslist_slist == NULL, 函数的行为是未定义的。

```
bool_t _slist_iterator_belong_to_slist(
    const slist_t* cpslist_slist, slist_iterator_t it_iter);
```

描述:

 判断一个迭代器引用的数据是否属于制定的容器。

参数:

 cpslist_slist slist 容器。
 it_iter slist 迭代器。

返回值:

 如果迭代器引用的数据属于 slist 容器的有效范围内, 返回 true, 否则返回 false。

注意:

 如果 cpslist_slist == NULL, 那么函数的行为是未定义的, 如果 it_iter 不是 slist 迭代器, 那么函数的行为是未定义的, 如果 it_iter 不属于容器 slist, 那么函数的行为是未定义的。

```
bool_t _slist_same_type(
    const slist_t* cpslist_first, const slist_t* cpslist_second);
```

描述:

 判断两个 slist 容器中保存的数据类型是否相同。

参数:

 cpslist_first 第一个 slist 容器。
 cpslist_second 第二个 slist 容器。

返回值:

 如果两个 slist 中保存的数据类型相同返回 true, 否则返回 false。

注意:

 如果 cpslist_first == NULL 或者 cpslist_second == NULL, 那么函数的行为是未定义的。两个 slist 容器必须是已经初始化或者使用 create_slist 创建的 slist 容器, 否则函数的行为是未定义的。如果 cpslist_first == cpslist_second 则返回 true。

```
bool_t _slist_same_slist_iterator_type(
    const slist_t* cpslist_slist, slist_iterator_t it_iter);
```

描述:

 判断 slist 容器中保存的数据类型与迭代器引用的数据类型是否相同。

参数:

 cpslist_slist slist 容器。
 it_iter slist 迭代器。

返回值:

 如果 slist 中保存的数据类型与 it_iter 引用的数据类型相同返回 true, 否则返回 false。

注意:

如果 cpslist_slist == NULL 或者 it_iter 不是 slist_iterator_t 类型, 那么函数的行为是未定义的。

```
void _slist_get_varg_value_auxiliary(
    slist_t* pslist_slist, va_list val_elemlist, _slistnode_t* pt_node);
```

描述:

从可变参数列表中获得与 slist 中的数据类型相同的数据。

参数:

pslist_slist slist 容器。
val_elemlist 可变参数列表。
pt_node 保存数据缓的冲区。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 pt_node == NULL, 那么函数的行为是未定义的。pslist_slist 必须是已经初始化或者是由 create_slist() 创建的 slist 容器, 否则函数的行为是未定义的。

```
void _slist_destroy_varg_value_auxiliary(
    slist_t* pslist_slist, _slistnode_t* pt_node);
```

描述:

销毁与 slist 中的数据类型相同的数据。

参数:

pslist_slist slist 容器。
pt_node 保存数据的缓冲区。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 pt_node == NULL, 那么函数的行为是未定义的。pslist_slist 必须是已经初始化或者是由 create_slist() 创建的 slist 容器, 否则函数的行为是未定义的。

```
void _slist_init_node_auxiliary(slist_t* pslist_slist, _slistnode_t* pt_node);
```

描述:

根据 slist 中的数据类型初始化 slist 节点。

参数:

pslist_slist slist 容器。
pt_node 保存数据的缓冲区。

返回值:

无。

注意:

如果 pslist_slist == NULL 或者 pt_node == NULL, 那么函数的行为是未定义的。pslist_slist 必须是已经初始化或者是由 create_slist() 创建的 slist 容器, 否则函数的行为是未定义的。

```
void _slist_transfer(
    slist_iterator_t it_pos, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

将数据区间[it_begin, it_end)内的数据转移到 it_pos。

参数:

- | | |
|----------|---------------|
| it_pos | 指定的插入数据的位置。 |
| it_begin | 转移的数据区间的开始位置。 |
| it_end | 转移的数据区间的末尾位置。 |

返回值:

无。

注意:

it_pos, it_begin, it_end 必须是有效的 slist 迭代器, 否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间, it_pos 与[it_begin, it_end)可以属于同一个 slist 容器, 但是 it_pos 属于(it_begin, it_end)的时候函数的行为是未定义的。it_pos == it_begin 或者 it_pos == it_end 或者 it_begin == it_end 的时候, 函数不做任何操作。

```
void _slist_transfer_after(
    slist_iterator_t it_pos, slist_iterator_t it_begin, slist_iterator_t it_end);
```

描述:

将数据区间[it_begin, it_end)内的数据转移到 it_pos+1。

参数:

- | | |
|----------|---------------|
| it_pos | 指定的插入数据的位置。 |
| it_begin | 转移的数据区间的开始位置。 |
| it_end | 转移的数据区间的末尾位置。 |

返回值:

无。

注意:

it_pos, it_begin, it_end 必须是有效的 slist 迭代器, 否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间, it_pos 与[it_begin, it_end)可以属于同一个 slist 容器, 但是 it_pos+1 属于(it_begin, it_end)的时候函数的行为是未定义的。it_pos+1 == it_begin 或者 it_pos+1 == it_end 或者 it_begin == it_end 的时候, 函数不做任何操作。

第十章 stack 容器适配器

stack 是容器适配器的一种，它是使用 vector, list 或者 deque 容器实现的。它的底层实现可以通过编译时进行配置。stack 没有迭代器操作。

第一节 stack 容器适配器的机制

stack 实现了 FILO，只能够在 stack 的顶部进行操作，不允许对 stack 中的数据进行遍历。

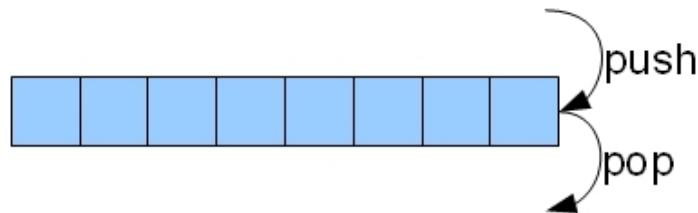


图 10.1 stack 的结构

第二节 stack 的代码结构

stack 的结构是如此的简单，所以它的代码结构也十分简单，并且都是由容器实现的：

```
typedef struct _tagstack
{
#if defined (CSTL_STACK_VECTOR_SEQUENCE)
    vector_t _t_sequence;
#elif defined (CSTL_STACK_LIST_SEQUENCE)
    list_t    _t_sequence;
#else
    deque_t   _t_sequence;
#endif
} stack_t;
```

第三节 外部接口

stack 要提供给用户外部接口：

create_stack	创建 stack 适配器。
stack_init	初始化一个空的 stack 适配器。
stack_init_copy	通过拷贝既存的 stack 适配器中的数据来初始化 stack 适配器。
stack_destroy	销毁 stack 适配器。
stack_size	返回 stack 适配器中数据的个数。
stack_empty	测试 stack 适配器是否为空。
stack_equal	测试两个 stack 适配器是否相等。
stack_not_equal	测试两个 stack 适配器是否不等。
stack_less	测试第一个 stack 是否小于第二个 stack 适配器。
stack_less_equal	测试第一个 stack 是否小于等于第二个 stack 适配器。
stack_greater	测试第一个 stack 是否大于第二个 stack 适配器。
stack_greater_equal	测试第一个 stack 是否大于等于第二个 stack 适配器。
stack_assign	使用一个既存的 stack 给 stack 适配器赋值。
stack_push	向 stack 中添加一个指定的数据。
stack_pop	删除 stack 顶部的一个数据。
stack_top	访问 stack 顶部的数据。

接口原型:

```
stack_t* create_stack(typename);
```

描述:

创建一个保存指定类型的 stack 适配器。

参数:

typename 保存的数据类型。

返回值:

指向 stack_t 适配器的指针。

注意:

这个函数的参数比较特殊, 它是描述适配器元素的数据类型的表达式, 关于类型机制以及类型描述语法请参考第四章。如果创建失败返回 NULL。

```
void stack_init(stack_t* psk_stack);
```

描述:

初始化一个空的 stack 适配器。

参数:

psk_stack 指向被初始化的 stack 适配器的指针。

返回值:

无。

注意:

被初始化的适配器一定是使用 `create_stack` 创建的适配器，否则函数的行为是未定义的。`psk_stack == NULL`，函数的行为是未定义的。使用 `stack_init` 初始化之后的 `stack` 适配器，`stack_size() == 0`。

```
void stack_init_copy(stack_t* psk_dest, const stack_t* cpsk_src);
```

描述：

使用一个既存的 `stack` 适配器来初始化 `stack` 适配器。

参数：

`psk_dest` 指向被初始化的适配器的指针。

`cpsk_src` 指向既存适配器的指针。

返回值：

无。

注意：

被初始化的适配器一定是使用 `create_stack` 创建的适配器，否则函数的行为是未定义的。`cpsk_src` 指向的是已经初始化的 `stack` 适配器，否则函数的行为是未定义的。`psk_dest == NULL` 或者 `cpsk_src == NULL`，函数的行为是未定义的。两个适配器保存的数据类型必须是一致的否则函数的行为是未定义的。初始化之后 `stack_size(psk_dest) == stack_size(cpsk_src)`。

```
void stack_destroy(stack_t* psk_stack);
```

描述：

销毁创建的 `stack` 适配器。

参数：

`psk_stack` 指向被销毁的适配器的指针。

返回值：

无。

注意：

被销毁的适配器一定已经初始化或者是使用 `create_stack` 创建的适配器，否则函数的行为是未定义的。销毁后的适配器不能够在用于其他的接口，否则接口的函数行为是未定义的。创建之后没有经过初始化的适配器也要进行销毁。

```
size_t stack_size(const stack_t* cpsk_stack);
```

描述：

返回 `stack` 适配器中数据的个数。

参数：

`cpsk_stack` 指向 `stack` 适配器的指针。

返回值：

适配器中数据的个数。

注意：

`cpsk_stack == NULL` 则函数的行为是未定义的，`cpsk_stack` 指向的适配器是经过初始化以后的 `stack` 适配器，否则函数的行为是未定义的。

```
bool_t stack_empty(const stack_t* cpsk_stack);
```

描述：

测试 `stack` 适配器是否为空。

参数:

cpsk_stack 指向 stack 适配器的指针。

返回值:

stack 适配器为空, 返回 true 否则返回 false。

注意:

cpsk_stack == NULL 则函数的行为是未定义的, cpsk_stack 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。

```
bool_t stack_equal(const stack_t* cpsk_first, const stack_t* cpsk_second);
```

描述:

测试两个适配器是否相等。

参数:

cpsk_first 指向 stack 适配器的指针。

cpsk_second 指向 stack 适配器的指针。

返回值:

如果两个适配器相等, 返回 true, 否则返回 false。

注意:

cpsk_first == NULL 或者 cpsk_second == NULL 则函数的行为是未定义的, cpsk_first 和 cpsk_second 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果两个 stack 适配器保存的数据类型不同, 那么这两个适配器不等。如果两个适配器中的数据个数相等并且对应数据相等则适配器相等, 如果 cpsk_first == cpsk_second 则返回 true。

```
bool_t stack_not_equal(const stack_t* cpsk_first, const stack_t* cpsk_second);
```

描述:

测试两个适配器是否不相等。

参数:

cpsk_first 指向 stack 适配器的指针。

cpsk_second 指向 stack 适配器的指针。

返回值:

如果两个适配器不相等, 返回 true, 否则返回 false。

注意:

cpsk_first == NULL 或者 cpsk_second == NULL 则函数的行为是未定义的, cpsk_first 和 cpsk_second 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果两个 stack 适配器保存的数据类型不同, 那么这两个适配器不等。如果两个适配器中的数据个数不等或者对应数据不等则适配器不等, 如果 cpsk_first == cpsk_second 则返回 false。

```
bool_t stack_less(const stack_t* cpsk_first, const stack_t* cpsk_second);
```

描述:

测试第一个 stack 适配器是否小于第二个 stack 适配器。

参数:

cpsk_first 指向 stack 适配器的指针。

cpsk_second 指向 stack 适配器的指针。

返回值:

如果第一个 stack 适配器小于第二个 stack 适配器, 返回 true, 否则返回 false。

注意:

cpsk_first == NULL 或者 cpsk_second == NULL 则函数的行为是未定义的, cpsk_first 和 cpsk_second 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果两个 stack 适配器保存的数据类型不同, 函数的行为是未定义的。如果第一个适配器中的数据小于第二个适配器中的对应数据则返回 true, 如果大于则返回 false, 如果对应数据都相等则, 如果第一个适配器中的数据个数小于第二个适配器中的数据个数则返回 true 否则返回 false, 如果 cpsk_first == cpsk_second 则返回 false。

```
bool_t stack_less_equal(const stack_t* cpsk_first, const stack_t* cpsk_second);
```

描述:

测试第一个 stack 适配器是否小于等于第二个 stack 适配器。

参数:

cpsk_first 指向 stack 适配器的指针。
cpsk_second 指向 stack 适配器的指针。

返回值:

如果第一个 stack 适配器小于等于第二个 stack 适配器, 返回 true, 否则返回 false。

注意:

cpsk_first == NULL 或者 cpsk_second == NULL 则函数的行为是未定义的, cpsk_first 和 cpsk_second 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果两个 stack 适配器保存的数据类型不同, 函数的行为是未定义的。如果第一个适配器中的数据小于第二个适配器中的对应数据则返回 true, 如果大于则返回 false, 如果对应数据都相等则, 如果第一个适配器中的数据个数小于等于第二个适配器中的数据个数则返回 true 否则返回 false, 如果 cpsk_first == cpsk_second 则返回 true。

```
bool_t stack_greater(const stack_t* cpsk_first, const stack_t* cpsk_second);
```

描述:

测试第一个 stack 适配器是否大于第二个 stack 适配器。

参数:

cpsk_first 指向 stack 适配器的指针。
cpsk_second 指向 stack 适配器的指针。

返回值:

如果第一个 stack 适配器大于第二个 stack 适配器, 返回 true, 否则返回 false。

注意:

cpsk_first == NULL 或者 cpsk_second == NULL 则函数的行为是未定义的, cpsk_first 和 cpsk_second 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果两个 stack 适配器保存的数据类型不同, 函数的行为是未定义的。如果第一个适配器中的数据大于第二个适配器中的对应数据则返回 true, 如果小于则返回 false, 如果对应数据都相等则, 如果第一个适配器中的数据个数大于第二个适配器中的数据个数则返回 true 否则返回 false, 如果 cpsk_first == cpsk_second 则返回 false。

```
bool_t stack_greater_equal(const stack_t* cpsk_first, const stack_t* cpsk_second);
```

描述:

测试第一个 stack 适配器是否大于等于第二个 stack 适配器。

参数:

- cpsk_first 指向 stack 适配器的指针。
- cpsk_second 指向 stack 适配器的指针。

返回值:

如果第一个 stack 适配器大于等于第二个 stack 适配器, 返回 true, 否则返回 false。

注意:

cpsk_first == NULL 或者 cpsk_second == NULL 则函数的行为是未定义的, cpsk_first 和 cpsk_second 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果两个 stack 适配器保存的数据类型不同, 函数的行为是未定义的。如果第一个适配器中的数据大于第二个适配器中的对应数据则返回 true, 如果小于则返回 false, 如果对应数据都相等则, 如果第一个适配器中的数据个数大于等于第二个适配器中的数据个数则返回 true 否则返回 false, 如果 cpsk_first == cpsk_second 则返回 true。

```
void stack_assign(stack_t* psk_dest, const stack_t* cpsk_src);
```

描述:

使用既存 stack 适配器为 stack 适配器赋值。

参数:

- psk_dest 指向目的 stack 适配器的指针。
- cpsk_src 指向源 stack 适配器的指针。

返回值:

无。

注意:

psk_dest == NULL 或者 cpsk_src == NULL 则函数的行为是未定义的, psk_dest 和 cpsk_src 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果两个 stack 适配器保存的数据类型不同, 函数的行为是未定义的。赋值以后两个适配器的 stack_size() 相等。如果 stack_equal(psk_dest, psk_src) 函数不执行任何操作。

```
void stack_push(stack_t* psk_stack, elem)
```

描述:

向 stack 适配器中添加一个数据。

参数:

- psk_stack 指向 stack 适配器的指针。
- elem 要添加的数据。

返回值:

无。

注意:

psk_stack == NULL 则函数的行为是未定义的, psk_stack 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。elem 是与 stack 适配器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
void stack_pop(stack_t* psk_stack);
```

描述:

删除 stack 适配器顶部的一个数据。

参数:

- psk_stack 指向 stack 适配器的指针。

返回值:

无。

注意:

psk_stack == NULL 则函数的行为是未定义的, psk_stack 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。如果 stack 为空, 则函数的行为是未定义的。

```
void* stack_top(const stack_t* cpsk_stack);
```

描述:

访问 stack 适配器顶部的数据。

参数:

cpsk_stack 指向 stack 适配器的指针。

返回值:

指向被访问的数据的指针。

注意:

cpsk_stack == NULL 则函数的行为是未定义的, cpsk_stack 指向的适配器是经过初始化以后的 stack 适配器, 否则函数的行为是未定义的。

第四节 内部接口

stack 内部接口是为了实现 stack 适配器以及外部接口提供的, 用户不应该直接使用这些接口。

_create_stack	创建一个 stack 适配器。
_create_stack_auxiliary	创建 stack 适配器的辅助函数。
_stack_destroy_auxiliary	销毁 stack 适配器的辅助函数。
_stack_push	向 stack 适配器添加一个数据。
_stack_push_varg	向 stack 适配器添加一个数据, 数据来自于可变参数列表。

```
stack_t* _create_stack(const char* s_typename);
```

描述:

创建一个 stack 适配器。

参数:

s_typename stack 适配器中保存的数据类型。

返回值:

创建成功返回指向适配器的指针, 否则返回 NULL。

注意:

如果 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
bool_t _create_stack_auxiliary(stack_t* psk_stack, const char* s_typename);
```

描述:

创建一个 stack 适配器的辅助函数。

参数:

psk_stack 没有创建的 stack 适配器。
s_typename stack 适配器中保存的数据类型。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 psk_stack == NULL 或者 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _stack_destroy_auxiliary(stack_t* psk_stack);
```

描述:

销毁 stack 适配器的辅助函数。

参数:

psk_stack stack 适配器。

返回值:

无。

注意:

如果 psk_stack == NULL 或者 stack 不是使用 create_stack 生成的则函数的行为是未定义的。

```
void _stack_push(stack_t* psk_stack, ...);
```

描述:

向 stack 适配器添加一个数据。

参数:

psk_stack stack 适配器。
... 用户指定的数据。

返回值:

无。

注意:

如果 psk_stack == NULL 或者 stack 是未初始化的 stack 则函数的行为是未定义的。用户指定的数据必须和适配器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
void _stack_push_varg(stack_t* psk_stack, va_list val_elemlist);
```

描述:

向 stack 适配器添加一个数据。

参数:

psk_stack stack 适配器。
val_elemlist 可变参数列表。

返回值:

无。

注意：

如果 `psk_stack == NULL` 或者 `stack` 是未初始化的 `stack` 则函数的行为是未定义的。用户指定的数据必须和适配器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

第十一章 queue 容器适配器

queue 容器适配器实现了 FIFO，它可以使用容器 list 和 deque 来实现，可以通过编译时进行配置。另外 queue 没设有迭代器。

第一节 queue 容器适配器的机制

queue 容器适配器实现了 FIFO，只能在 queue 末尾添加数据，在 queue 开头删除数据。不允许对 queue 中的数据进行便利。

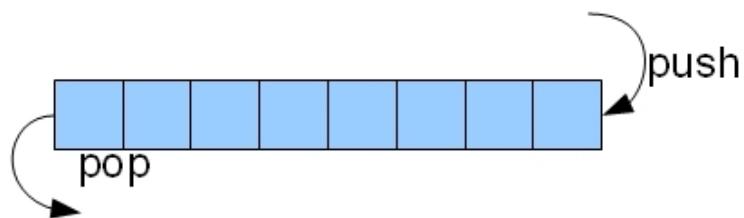


图 11.1 queue 的结构

第二节 queue 的代码结构

queue 的结构是如此的简单，所以它的代码结构也十分简单，并且都是由容器实现的：

```
typedef struct _tagqueue
{
#if defined (CSTL_QUEUE_LIST_SEQUENCE)
    list_t    _t_sequence;
#else
    deque_t   _t_sequence;
#endif
}queue_t;
```

第三节 外部接口

queue 要提供给用户外部接口：

create_queue	创建 queue 适配器。
queue_init	初始化一个空的 queue 适配器。
queue_init_copy	通过拷贝既存的 queue 适配器中的数据来初始化 queue 适配器。
queue_destroy	销毁 queue 适配器。
queue_size	返回 queue 适配器中数据的个数。
queue_empty	测试 queue 适配器是否为空。
queue_equal	测试两个 queue 适配器是否相等。
queue_not_equal	测试两个 queue 适配器是否不等。
queue_less	测试第一个 queue 是否小于第二个 queue 适配器。
queue_less_equal	测试第一个 queue 是否小于等于第二个 queue 适配器。
queue_greater	测试第一个 queue 是否大于第二个 queue 适配器。
queue_greater_equal	测试第一个 queue 是否大于等于第二个 queue 适配器。
queue_assign	使用一个既存的 queue 给 queue 适配器赋值。
queue_push	向 queue 中末尾添加一个指定的数据。
queue_pop	删除 queue 开头的一个数据。
queue_front	访问 queue 开头的数据。
queue_back	访问 queue 末尾的数据。

接口原型:

```
queue_t* create_queue(typename);
```

描述:

创建一个保存指定类型的 queue 适配器。

参数:

typename 保存的数据类型。

返回值:

指向 queue_t 适配器的指针。

注意:

这个函数的参数比较特殊, 它是描述适配器元素的数据类型的表达式, 关于类型机制以及类型描述语法请参考第四章。如果创建失败返回 NULL。

```
void queue_init(queue_t* pque_queue);
```

描述:

初始化一个空的 queue 适配器。

参数:

pque_queue 指向被初始化的 queue 适配器的指针。

返回值:

无。

注意:

被初始化的适配器一定是使用 `create_queue` 创建的适配器, 否则函数的行为是未定义的。`pque_queue == NULL`, 函数的行为是未定义的。使用 `queue_init` 初始化之后的 queue 适配器, `queue_size() == 0`。

```
void queue_init_copy(queue_t* pque_dest, const queue_t* cpque_src);
```

描述:

使用一个既存的 queue 适配器来初始化 queue 适配器。

参数:

<code>pque_dest</code>	指向被初始化的适配器的指针。
<code>cpque_src</code>	指向既存适配器的指针。

返回值:

无。

注意:

被初始化的适配器一定是使用 `create_queue` 创建的适配器, 否则函数的行为是未定义的。`cpque_src` 指向的是已经初始化的 queue 适配器, 否则函数的行为是未定义的。`pque_dest == NULL` 或者 `cpque_src == NULL`, 函数的行为是未定义的。两个适配器保存的数据类型必须是一致的否则函数的行为是未定义的。初始化之后 `queue_size(pque_dest) == queue_size(cpque_src)`。

```
void queue_destroy(queue_t* pque_queue);
```

描述:

销毁创建的 queue 适配器。

参数:

<code>pque_queue</code>	指向被销毁的适配器的指针。
-------------------------	---------------

返回值:

无。

注意:

被销毁的适配器一定已经初始化或者是使用 `create_queue` 创建的适配器, 否则函数的行为是未定义的。销毁后的适配器不能够在用于其他的接口, 否则接口的函数行为是未定义的。创建之后没有经过初始化的适配器也要进行销毁。

```
size_t queue_size(const queue_t* cpque_queue);
```

描述:

返回 queue 适配器中数据的个数。

参数:

<code>cpque_queue</code>	指向 queue 适配器的指针。
--------------------------	------------------

返回值:

适配器中数据的个数。

注意:

`cpque_queue == NULL` 则函数的行为是未定义的, `cpque_queue` 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。

```
bool_t queue_empty(const queue_t* cpque_queue);
```

描述:

测试 queue 适配器是否为空。

参数:

cpque_queue 指向 queue 适配器的指针。

返回值:

queue 适配器为空, 返回 true 否则返回 false。

注意:

cpque_queue == NULL 则函数的行为是未定义的, cpque_queue 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。

```
bool_t queue_equal(const queue_t* cpque_first, const queue_t* cpque_second);
```

描述:

测试两个适配器是否相等。

参数:

cpque_first 指向 queue 适配器的指针。

cpque_second 指向 queue 适配器的指针。

返回值:

如果两个适配器相等, 返回 true, 否则返回 false。

注意:

cpque_first == NULL 或者 cpque_second == NULL 则函数的行为是未定义的, cpque_first 和 cpque_second 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。如果两个 queue 适配器保存的数据类型不同, 那么这两个适配器不等。如果两个适配器中的数据个数相等并且对应数据相等则适配器相等, 如果 cpque_first == cpque_second 则返回 true。

```
bool_t queue_not_equal(const queue_t* cpque_first, const queue_t* cpque_second);
```

描述:

测试两个适配器是否不相等。

参数:

cpque_first 指向 queue 适配器的指针。

cpque_second 指向 queue 适配器的指针。

返回值:

如果两个适配器不相等, 返回 true, 否则返回 false。

注意:

cpque_first == NULL 或者 cpque_second == NULL 则函数的行为是未定义的, cpque_first 和 cpque_second 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。如果两个 queue 适配器保存的数据类型不同, 那么这两个适配器不等。如果两个适配器中的数据个数不等或者对应数据不等则适配器不等, 如果 cpque_first == cpque_second 则返回 false。

```
bool_t queue_less(const queue_t* cpque_first, const queue_t* cpque_second);
```

描述:

测试第一个 queue 适配器是否小于第二个 queue 适配器。

参数:

cpque_first 指向 queue 适配器的指针。

cpque_second 指向 queue 适配器的指针。

返回值:

如果第一个 queue 适配器小于第二个 queue 适配器，返回 true，否则返回 false。

注意:

cpque_first == NULL 或者 cpque_second == NULL 则函数的行为是未定义的，cpque_first 和 cpque_second 指向的适配器是经过初始化以后的 queue 适配器，否则函数的行为是未定义的。如果两个 queue 适配器保存的数据类型不同，函数的行为是未定义的。如果第一个适配器中的数据小于第二个适配器中的对应数据则返回 true，如果大于则返回 false，如果对应数据都相等则，如果第一个适配器中的数据个数小于第二个适配器中的数据个数则返回 true 否则返回 false，如果 cpque_first == cpque_second 则返回 false。

```
bool_t queue_less_equal(const queue_t* cpque_first, const queue_t* cpque_second);
```

描述:

测试第一个 queue 适配器是否小于等于第二个 queue 适配器。

参数:

cpque_first 指向 queue 适配器的指针。

cpque_second 指向 queue 适配器的指针。

返回值:

如果第一个 queue 适配器小于等于第二个 queue 适配器，返回 true，否则返回 false。

注意:

cpque_first == NULL 或者 cpque_second == NULL 则函数的行为是未定义的，cpque_first 和 cpque_second 指向的适配器是经过初始化以后的 queue 适配器，否则函数的行为是未定义的。如果两个 queue 适配器保存的数据类型不同，函数的行为是未定义的。如果第一个适配器中的数据小于第二个适配器中的对应数据则返回 true，如果大于则返回 false，如果对应数据都相等则，如果第一个适配器中的数据个数小于等于第二个适配器中的数据个数则返回 true 否则返回 false，如果 cpque_first == cpque_second 则返回 true。

```
bool_t queue_greater(const queue_t* cpque_first, const queue_t* cpque_second);
```

描述:

测试第一个 queue 适配器是否大于第二个 queue 适配器。

参数:

cpque_first 指向 queue 适配器的指针。

cpque_second 指向 queue 适配器的指针。

返回值:

如果第一个 queue 适配器大于第二个 queue 适配器，返回 true，否则返回 false。

注意:

cpque_first == NULL 或者 cpque_second == NULL 则函数的行为是未定义的，cpque_first 和 cpque_second 指向的适配器是经过初始化以后的 queue 适配器，否则函数的行为是未定义的。如果两个 queue 适配器保存的数据类型不同，函数的行为是未定义的。如果第一个适配器中的数据大于第二个适配器中的对应数据则返回 true，如果小于则返回 false，如果对应数据都相等则，如果第一个适配器中的数据个数大于第二个适配器中的数据个数则返回 true 否则返回 false，如果 cpque_first == cpque_second 则返回 false。

```
bool_t queue_greater_equal(const queue_t* cpque_first, const queue_t* cpque_second);
```

描述:

测试第一个 queue 适配器是否大于等于第二个 queue 适配器。

参数:

- cpque_first 指向 queue 适配器的指针。
- cpque_second 指向 queue 适配器的指针。

返回值:

如果第一个 queue 适配器大于等于第二个 queue 适配器, 返回 true, 否则返回 false。

注意:

cpque_first == NULL 或者 cpque_second == NULL 则函数的行为是未定义的, cpque_first 和 cpque_second 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。如果两个 queue 适配器保存的数据类型不同, 函数的行为是未定义的。如果第一个适配器中的数据大于第二个适配器中的对应数据则返回 true, 如果小于则返回 false, 如果对应数据都相等则, 如果第一个适配器中的数据个数大于等于第二个适配器中的数据个数则返回 true 否则返回 false, 如果 cpque_first == cpque_second 则返回 true。

```
void queue_assign(queue_t* pque_dest, const queue_t* cpque_src);
```

描述:

使用既存 queue 适配器为 queue 适配器赋值。

参数:

- pque_dest 指向目的 queue 适配器的指针。
- cpque_src 指向源 queue 适配器的指针。

返回值:

无。

注意:

pque_dest == NULL 或者 cpque_src == NULL 则函数的行为是未定义的, pque_dest 和 cpque_src 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。如果两个 queue 适配器保存的数据类型不同, 函数的行为是未定义的。赋值以后两个适配器的 queue_size() 相等。如果 queue_equal(pque_dest, pque_src) 函数不执行任何操作。

```
void queue_push(queue_t* pque_queue, elem)
```

描述:

向 queue 适配器末尾添加一个数据。

参数:

- pque_queue 指向 queue 适配器的指针。
- elem 要添加的数据。

返回值:

无。

注意:

pque_queue == NULL 则函数的行为是未定义的, pque_queue 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。elem 是与 queue 适配器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
void queue_pop(queue_t* pque_queue);
```

描述:

删除 queue 适配器开头的一个数据。

参数:

pque_queue 指向 queue 适配器的指针。

返回值:

无。

注意:

pque_queue == NULL 则函数的行为是未定义的, pque_queue 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。如果 queue 为空, 则函数的行为是未定义的。

```
void* queue_front(const queue_t* cpque_queue);
```

描述:

访问 queue 适配器开头的数据。

参数:

cpque_queue 指向 queue 适配器的指针。

返回值:

指向被访问的数据的指针。

注意:

cpque_queue == NULL 则函数的行为是未定义的, cpque_queue 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。

```
void* queue_back(const queue_t* cpque_queue);
```

描述:

访问 queue 适配器末尾的数据。

参数:

cpque_queue 指向 queue 适配器的指针。

返回值:

指向被访问的数据的指针。

注意:

cpque_queue == NULL 则函数的行为是未定义的, cpque_queue 指向的适配器是经过初始化以后的 queue 适配器, 否则函数的行为是未定义的。

第四节 内部接口

queue 内部接口是为了实现 queue 适配器以及外部接口提供的, 用户不应该直接使用这些接口。

_create_queue	创建一个 queue 适配器。
_create_queue_auxiliary	创建 queue 适配器的辅助函数。
_queue_destroy_auxiliary	销毁 queue 适配器的辅助函数。
_queue_push	向 queue 适配器末尾添加一个数据。
_queue_push_varg	向 queue 适配器末尾添加一个数据, 数据来自于可变参数列表。

```
queue_t* _create_queue(const char* s_typename);
```

描述:

创建一个 queue 适配器。

参数:

s_typename queue 适配器中保存的数据类型。

返回值:

创建成功返回指向适配器的指针, 否则返回 NULL。

注意:

如果 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
bool_t _create_queue_auxiliary(queue_t* pque_queue, const char* s_typename);
```

描述:

创建一个 queue 适配器的辅助函数。

参数:

pque_queue 没有创建的 queue 适配器。

s_typename queue 适配器中保存的数据类型。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 pque_queue == NULL 或者 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _queue_destroy_auxiliary(queue_t* pque_queue);
```

描述:

销毁 queue 适配器的辅助函数。

参数:

pque_queue queue 适配器。

返回值:

无。

注意:

如果 pque_queue == NULL 或者 queue 不是使用 create_queue 生成的则函数的行为是未定义的。

```
void _queue_push(queue_t* pque_queue, ...);
```

描述:

向 queue 适配器添加一个数据。

参数:

pque_queue queue 适配器。

... 用户指定的数据。

返回值:

无。

注意:

如果 `pque_queue == NULL` 或者 `queue` 是未初始化的 `queue` 则函数的行为是未定义的。用户指定的数据必须和适配器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _queue_push_varg(queue_t* pque_queue, va_list val_elemlist);
```

描述:

向 `queue` 适配器添加一个数据。

参数:

<code>pque_queue</code>	<code>queue</code> 适配器。
<code>val_elemlist</code>	可变参数列表。

返回值:

无。

注意:

如果 `pque_queue == NULL` 或者 `queue` 是未初始化的 `queue` 则函数的行为是未定义的。用户指定的数据必须和适配器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

第十二章 basic_string 容器

basic_string 实现了串的概念，在 basic_string 能够保存任何类型。主要的操作包括替换、查找等等。basic_string 作为 string_t 类型的底层实现，目前没有对外公开 basic_string 的接口。

第一节 basic_string 容器的机制

在底层使用 vector 作为 basic_string 的实现，所以 basic_string 的机制与 vector 的相同。

basic_string 将保存的数据作为串，那么当要向 basic_string 中添加一个数据串的时候，怎样表示数据串的结束？这是一个问题，我们借鉴了 STL 的做法，但是不完全使用 STL 的做法。

STL 的做法是使用数据 0 表示数据串的结束：

```
#include <string>
#include <vector>
#include <utility>
#include <iostream>

int main(int argc, char* argv[])
{
    std::basic_string<int> nstr;
    std::vector<int> nvec;

    nvec.push_back(9);
    nvec.push_back(4);
    nvec.push_back(0);
    nvec.push_back(5);
    nstr.append(&nvec[0]);
    std::cout << nstr.size() << std::endl;

    int v[5] = {2, 4, 5, 1, 5};
    nstr.clear();
    nstr.append(v);
    std::cout << nstr.size() << std::endl;

    std::basic_string<std::pair<int, int> > pstr;
    std::vector<std::pair<int, int> > pvec;
    pvec.push_back(std::make_pair(0, 5));
    pvec.push_back(std::make_pair(3, 0));
    pvec.push_back(std::make_pair(0, 0));
```

```

    pvec.push_back(std::make_pair(4, 4));

    pstr.append(&pvec[0]);
    std::cout << pstr.size() << std::endl;

    return 0;
}

```

这个程序的结果第一次输出的是 2，因为 nvec 第三个数据是 0，`basic_string` 认为数据串结束了。第二次输出的数应该大于等于 5，如果 v 后面内存中的数据是 0 的话那么结果是 5，如果不是那么 `basic_string` 会一直数到内存中的数据是 0 的时候停止(这就是内存越界了)，这时候的结果就无法知道是多少了。第三次输出的结果是 2，`basic_string` 遇到(0,0)的时候认为数据串结束了。

`libcstl` 不采用数据 0 而采用内存 0 来表示数据串结束，这样对于 C 内建类型来说与 STL 的效果是一样的，但是在数据结构和复杂类型的时候数据 0 不一定等于内存 0，这样就可以保存数据 0 的情况。而内存 0 对于复杂的数据结构来说一般是非法数据。

这样如果使用 `libcstl` 来编写上面的代码，前两种情况与 STL 是一致的，但是第三种情况是不同的 `libcstl` 可以保存(0,0)但是需要在数据串的末尾添加内存 0 的数据表示结束。

`basic_string` 的操作函数中后很多是带有`_cstr` 和`_subcstr` 这样的后缀的函数，这些函数使用的是数据串，对于 C 内建类型，这个串就是使用数据 0 表示的数组例如：

```

basic_string_t* pt_basic_string = create_basic_string(int);
int elems[] = {9, 23, 45, 0, 556};

```

这样可以将 `elems` 应用于带有`_cstr` 和`_subcstr` 的函数。如：

```
basic_string_init_cstr(pt_basic_string, elems);
```

这样函数初始化后 `basic_string_size(pt_basic_string) == 3;`

对于保存 `char` 类型的 `basic_string` 来说也适用于上面的做法，但是还有更简单的做法，直接使用字符串：

```

basic_string_t* pt_basic_string = create_basic_string(char);
basic_string_init_cstr(pt_basic_string, "abcdefg");

```

这样初始化之后 `basic_string_size(pt_basic_string) == 7;`

对于保存 `char*` 类型的 `basic_string` 来说，它的形式比较特殊，要使用保存 `char*` 类型的数组来表示，使用 `NUL` 表示数据串的结束如：

```

basic_string_t* pt_basic_string = create_basic_string(char*);
const char* elems[] = {"abc", "def", NULL, "hij"};
basic_string_init_cstr(pt_basic_string, elems);

```

初始化后 `basic_string_size(pt_basic_string) == 2;`

对于保存 `libcstl` 内建类型和用户自定义类型也要使用保存指针的数组来初始化如：

```

basic_string_t* pt_basic_string = create_basic_string(abc_t);
abc_t abc1, abc2, abc3;
abc_t* elems[] = {&abc1, &abc2, NULL, &abc3};
basic_string_init_cstr(pt_basic_string, elems);

```

这样初始化之后 `basic_string_size(pt_basic_string) == 2;`

在 `basic_string` 中有很多函数是使用下标表示位置而不是使用迭代器来表示位置的，在这种情况下，下标从 0 开始，超过 `basic_string_length-1` 的位置是无效位置，我们使用常量 `NPOS` 表示。

第二节 basic_string 迭代器

basic_string 迭代器与 vector 迭代器相同，但是它有自己的容器类型。

第三节 basic_string 的代码结构

basic_string 直接使用 vector 作为底层实现

```
typedef struct _tagbasicstring
{
    vector_t _t_vector;
}basic_string_t;
```

第四节 外部接口

basic_string 外部接口：

create_basic_string	创建 basic_string 容器。
basic_string_init	初始化一个空的 basic_string 容器。
basic_string_init_elem	使用指定的数据初始化 basic_string 容器。
basic_string_init_cstr	使用指定的数据串初始化 basic_string 容器。
basic_string_init_subcstr	使用指定的子数据串初始化 basic_string 容器。
basic_string_init_copy	使用指定的 basic_string 来初始化 basic_string 容器。
basic_string_init_copy_substring	使用指定的子 basic_string 来初始化 basic_string 容器。
basic_string_init_copy_range	使用指定的 basic_string 范围初始化 basic_string 容器。
basic_string_destroy	销毁一个 basic_string 容器。
basic_string_c_str	返回指向 basic_string 容器中保存数据的指针。
basic_string_data	返回指向 basic_string 容器中保存数据的指针。
basic_string_copy	将 basic_string 容器中保存的数据拷贝到指定的缓冲区中。
basic_string_size	返回 basic_string 容器中保存的数据的个数。
basic_string_length	返回 basic_string 容器中保存的数据的长度。
basic_string_empty	测试 basic_string 容器是否为空。

basic_string_max_size	返回 basic_string 容器能够保存的数据的数量。
basic_string_capacity	返回 basic_string 容器的容量。
basic_string_at	通过下标访问 basic_string 容器中的数据。
basic_string_equal	测试两个 basic_string 容器是否相等。
basic_string_not_equal	测试两个 basic_string 容器是否不等。
basic_string_less	测试第一个 basic_string 容器是否小于第二个 basic_string 容器。
basic_string_less_equal	测试第一个 basic_string 容器是否小于等于第二个 basic_string 容器。
basic_string_greater	测试第一个 basic_string 容器是否大于第二个 basic_string 容器。
basic_string_greater_equal	测试第一个 basic_string 容器是否大于等于第二个 basic_string 容器。
basic_string_equal_cstr	测试 basic_string 容器和数据串是否相等。
basic_string_not_equal_cstr	测试 basic_string 容器和数据串是否不等。
basic_string_less_cstr	测试 basic_string 容器是否小于数据串。
basic_string_less_equal_cstr	测试 basic_string 容器是否小于等于数据串。
basic_string_greater_cstr	测试 basic_string 容器是否大于数据串。
basic_string_greater_equal_cstr	测试 basic_string 容器是否大于等于数据串。
basic_string_compare	返回两个 basic_string 容器比较的结果。
basic_string_compare_substring_string	返回子 basic_string 容器与另一个 basic_string 容器比较的结果。
basic_string_compare_substring_substring	返回两个子 basic_string 容器的比较结果。
basic_string_compare_cstr	返回 basic_string 容器与数据串的比较结果。
basic_string_compare_substring_cstr	返回子 basic_string 容器与数据串的比较结果。
basic_string_compare_substring_subcstr	返回子 basic_string 容器与子数据串的比较结果。
basic_string_substr	返回指定的子 basic_string 容器。
basic_string_connect	将两个 basic_string 容器链接。
basic_string_connect_elem	将 basic_string 容器与指定的数据连接。
basic_string_connect_cstr	将 basic_string 容器与数据串链接。
basic_string_find	在 basic_string 容器中查找指定的 basic_string。
basic_string_find_cstr	在 basic_string 容器中查找指定的数据串。
basic_string_find_subcstr	在 basic_string 容器中查找指定的子数据串。
basic_string_find_elem	在 basic_string 容器中查找指定的数据。
basic_string_rfind	在 basic_string 容器中反向查找指定的 basic_string。
basic_string_rfind_cstr	在 basic_string 容器中反向查找数据串。
basic_string_rfind_subcstr	在 basic_string 容器中反向查找子数据串。

basic_string_rfind_elem	在 basic_string 容器中反向查找指定的数据。
basic_string_find_first_of	在 basic_string 容器中查找第一个在指定的 basic_string 容器出现的数据。
basic_string_find_first_of_cstr	在 basic_string 容器中查找第一个在数据串中出现的数据。
basic_string_find_first_of_subestr	在 basic_string 容器中查找第一个在子数据串中出现的数据。
basic_string_find_first_of_elem	在 basic_string 容器中查找第一个出现的指定数据。
basic_string_find_first_not_of	在 basic_string 容器中查找第一个不出现在指定 basic_string 容器中的数据。
basic_string_find_first_not_of_cstr	在 basic_string 容器中查找第一个不出现在数据串中的数据。
basic_string_find_first_not_of_subestr	在 basic_string 容器中查找第一个不出现在子数据串中的数据。
basic_string_find_first_not_of_elem	在 basic_string 容器中查找第一个不是指定数据的数据。
basic_string_find_last_of	在 basic_string 容器中查找最后一个出现在指定 basic_string 容器中的数据。
basic_string_find_last_of_cstr	在 basic_string 容器中查找最后一个出现在数据串中的数据。
basic_string_find_last_of_subestr	在 basic_string 容器中查找最后一个出现在子数据串中的数据。
basic_string_find_last_of_elem	在 basic_string 容器中查找最后一个出现的指定数据。
basic_string_find_last_not_of	在 basic_string 容器中查找最后一个不出现在指定 basic_string 容器中的数据。
basic_string_find_last_not_of_cstr	在 basic_string 容器中查找最后一个不出现在数据串中的数据。
basic_string_find_last_not_of_subestr	在 basic_string 容器中查找最后一个不出现在子数据串中的数据。
basic_string_find_last_not_of_elem	在 basic_string 容器中查找最后一个不是指定数据的数据。
basic_string_begin	返回指向 basic_string 容器开头的迭代器。
basic_string_end	返回指向 basic_string 容器末尾的迭代器。
basic_string_clear	清空 basic_string 容器。
basic_string_swap	交换两个 basic_string 容器的内容。
basic_string_reserve	重新设置 basic_string 容器的容量。
basic_string_assign	使用指定的 basic_string 容器为 basic_string 容器赋值。
basic_string_assign_substring	使用指定的子 basic_string 为 basic_string 容器赋值。
basic_string_assign_cstr	使用指定的数据串为 basic_string 容器赋值。
basic_string_assign_subestr	使用指定的子数据串为 basic_string 容器赋值。
basic_string_assign_range	使用指定的数据区间为 basic_string 容器赋值。
basic_string_assign_elem	使用指定的数据为 basic_string 容器赋值。
basic_string_append	向 basic_string 容器末尾添加指定的 basic_string 容器。
basic_string_append_substring	向 basic_string 容器末尾添加指定的子 basic_string。
basic_string_append_cstr	向 basic_string 容器末尾添加指定的数据串。
basic_string_append_subestr	向 basic_string 容器末尾添加指定的子数据串。

basic_string_append_range	向 basic_string 容器末尾添加指定的数据区间。
basic_string_append_elem	向 basic_string 容器末尾添加指定的数据。
basic_string_insert	向 basic_string 容器中插入指定数据。
basic_string_insert_n	向 basic_string 容器中插入多个指定数据。
basic_string_insert_string	向 basic_string 容器中插入指定的 basic_string 容器。
basic_string_insert_substring	向 basic_string 容器中插入指定的子 basic_string。
basic_string_insert_cstr	向 basic_string 容器中插入指定的数据串。
basic_string_insert_subcstr	向 basic_string 容器中插入指定的子数据串。
basic_string_insert_range	向 basic_string 容器中插入指定的数据区间。
basic_string_insert_elem	向 basic_string 容器中插入指定数据。
basic_string_push_back	向 basic_string 容器末尾添加指定数据。
basic_string_resize	重新设置 basic_string 容器中数据的个数。
basic_string_erase	删除 basic_string 容器中指定位置的数据。
basic_string_erase_range	删除 basic_string 容器中指定的数据区间。
basic_string_erase_substring	删除 basic_string 容器中的子数据串。
basic_string_replace	将 basic_string 容器中的子数据串替换成指定的 basic_string。
basic_string_replace_substring	将 basic_string 容器中的子数据串替换成指定的子 basic_string。
basic_string_replace_cstr	将 basic_string 容器中的子数据串替换成指定的数据串。
basic_string_replace_subcstr	将 basic_string 容器中的子数据串替换成指定的子数据串。
basic_string_replace_elem	将 basic_string 容器中的子数据串替换成指定的数据。
basic_string_range_replace	将 basic_string 容器中的数据区间替换成 basic_string。
basic_string_range_replace_substring	将 basic_string 容器中的数据区间替换成子 basic_string。
basic_string_range_replace_cstr	将 basic_string 容器中的数据区间替换成数据串。
basic_string_range_replace_subcstr	将 basic_string 容器中的数据区间替换成子数据区间。
basic_string_range_replace_elem	将 basic_string 容器中的数据区间替换成指定的数据。
basic_string_replace_range	将 basic_string 容器中的数据区间替换成指定的数据区间。

接口原型:

```
basic_string_t* create_basic_string(typename);
```

描述:

创建一个保存指定类型的 basic_string 容器。

参数:

typename 保存的数据类型。

返回值:

指向 basic_string_t 容器的指针。

注意：

这个函数的参数比较特殊，它是描述容器元素的数据类型的表达式，关于类型机制以及类型描述语法请参考第四章。如果创建失败返回 NULL。

```
void basic_string_init(basic_string_t* pt_basic_string);
```

描述：

初始化一个空的 basic_string 容器。

参数：

pt_basic_string 指向被初始化的 basic_string 容器的指针。

返回值：

无。

注意：

被初始化的容器一定是使用 create_basic_string 创建的容器，否则函数的行为是未定义的。pt_basic_string == NULL，函数的行为是未定义的。使用 basic_string_init 初始化之后的 basic_string 容器，basic_string_size() == 0, basic_string_capacity() == 0。

```
void basic_string_init_elem(basic_string_t* pt_basic_string, size_t t_count, elem);
```

描述：

使用用户指定的数据初始化 basic_string 容器。

参数：

pt_basic_string 指向被初始化的 basic_string 容器的指针。

t_count 初始化后包含的数据的个数。

elem 指定的数据。

返回值：

无。

注意：

如果 pt_basic_string == NULL 则函数的行为是未定义的。basic_string 容器必须是使用 create_basic_string 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。使用 basic_string_init_elem 初始化之后的 basic_string 容器，basic_string_size() == t_count，容量符合前面描述的算法，并且 t_count 是指定的数据 elem，elem 的限制请参考第四章。

```
void basic_string_init_cstr(
    basic_string_t* pt_basic_string, const void* cpv_value_string);
```

描述：

使用指定的数据串初始化 basic_string 容器。

参数：

pt_basic_string 指向被初始化的 basic_string 容器的指针。

cpv_value_string 指定的数据串。

返回值：

无。

注意：

如果 `pt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的。`basic_string` 容器必须是使用 `create_basic_string` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。数据串使用内存 0 表示结束，初始化之后 `basic_string_size()` 等于数据串的长度。

```
void basic_string_init_subcstr(
    basic_string_t* pt_basic_string, const void* cpv_value_string, size_t t_len);
```

描述:

使用指定的子数据串初始化 `basic_string` 容器。

参数:

<code>pt_basic_string</code>	指向被初始化的 <code>basic_string</code> 容器的指针。
<code>cpv_value_string</code>	指定的数据串。
<code>t_len</code>	子串的长度。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的。`basic_string` 容器必须是使用 `create_basic_string` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。如果 `t_len` 等于 `NPOS` 或者超过了数据串的长度则使用整个数据串进行初始化。

```
void basic_string_init_copy(basic_string_t* pt_dest, const basic_string_t* cpt_src);
```

描述:

使用一个既存的 `basic_string` 容器来初始化 `basic_string` 容器。

参数:

<code>pt_dest</code>	指向被初始化的容器的指针。
<code>cpt_src</code>	指向既存容器的指针。

返回值:

无。

注意:

被初始化的容器一定是使用 `create_basic_string` 创建的容器，否则函数的行为是未定义的。`cpt_src` 指向的是已经初始化的 `basic_string` 容器，否则函数的行为是未定义的。`pt_dest == NULL` 或者 `cpt_src == NULL`，函数的行为是未定义的。两个容器保存的数据类型必须是一致的否则函数的行为是未定义的。初始化之后 `basic_string_size(pt_dest) == basic_string_size(cpt_src)`；容量符合前面描述的算法。

```
void basic_string_init_copy_substring(
    basic_string_t* pt_dest, const basic_string_t* pt_src,
    size_t t_pos, size_t t_len);
```

描述:

使用指定的子 `basic_string` 来初始化 `basic_string` 容器。

参数:

<code>pt_dest</code>	指向被初始化的容器的指针。
<code>pt_src</code>	指向既存容器的指针。
<code>t_pos</code>	子字符串开始的位置。

t_len 子字符串长度。

返回值:

无。

注意:

被初始化的容器一定是使用 create_basic_string 创建的容器，否则函数的行为是未定义的。pt_src 指向的是已经初始化的 basic_string 容器，否则函数的行为是未定义的。pt_dest == NULL 或者 pt_src == NULL，函数的行为是未定义的。两个容器保存的数据类型必须是一致的否则函数的行为是未定义的。t_pos 必须是属于源字符串的有效位置，否则函数的行为是未定义的，如果 t_len == NPOS 或者 t_pos + t_len >= 元字符串的长度，那么使用从 t_pos 开始的所有子字符串。

```
void basic_string_init_copy_range(basic_string_t* pt_basic_string,
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end);
```

描述:

使用指定的 basic_string 范围初始化 basic_string 容器。

参数:

pt_basic_string 指向被初始化的容器的指针。

it_begin 指定的数据区间的开始。

it_end 制定的数据区间的末尾。

返回值:

无。

注意:

被初始化的容器一定是使用 create_basic_string 创建的容器，否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 basic_string 容器的有效数据区间，否则函数的行为是未定义的。pt_basic_string == NULL 函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。初始化之后 basic_string_size(pt_basic_string) == iterator_distance(it_begin, it_end); 容量符合前面描述的算法。

```
void basic_string_destroy(basic_string_t* pt_basic_string);
```

描述:

销毁创建的 basic_string 容器。

参数:

pt_basic_string 指向被销毁的容器的指针。

返回值:

无。

注意:

被销毁的容器一定已经初始化或者是使用 create_basic_string 创建的容器，否则函数的行为是未定义的。销毁后的容器不能够在用于其他的接口，否则接口的函数行为是未定义的。创建之后没有经过初始化的容器也要进行销毁。

```
const void* basic_string_c_str(const basic_string_t* cpt_basic_string);
```

描述:

返回指向 basic_string 容器中保存数据的指针。

参数:

cpt_basic_string basic_string 容器。

返回值:

返回指向 basic_string 中保存的数据串的指针。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 容器一定已经初始化的容器，否则函数的行为是未定义的。当 basic_string 为空时返回 NULL。

```
const void* basic_string_data(const basic_string_t* cpt_basic_string);
```

描述:

返回指向 basic_string 容器中保存数据的指针。

参数:

cpt_basic_string basic_string 容器。

返回值:

返回指向 basic_string 中保存的数据串的指针。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 容器一定已经初始化的容器，否则函数的行为是未定义的。当 basic_string 为空时函数的行为是未定义的。

```
size_t basic_string_copy(const basic_string_t* cpt_basic_string,
    void* pv_buffer, size_t t_len, size_t t_pos);
```

描述:

将 basic_string 容器中保存的数据拷贝到指定的缓冲区中。

参数:

cpt_basic_string basic_string 容器。

pv_buffer 目的缓冲区。

t_len 拷贝的数据串的长度。

t_pos 子串的位置。

返回值:

实际拷贝的子串的长度。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 容器一定已经初始化的容器，否则函数的行为是未定义的。如果 pv_buffer == NULL 那么函数的行为是未定义的。t_pos 是属于 basic_string 的有效的位置，否则函数的行为是未定义的。实际拷贝的长度是 min(t_len, basic_string_size() - t_pos)。

```
size_t basic_string_size(const basic_string_t* cpt_basic_string);
```

描述:

返回 basic_string 容器中保存的数据的个数。

参数:

cpt_basic_string basic_string 容器。

返回值:

basic_string 容器中包含的数据的个数。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 容器一定已经初始化的容器，否则函数的行为是未定义的。

```
size_t basic_string_length(const basic_string_t* cpt_basic_string);
```

描述:

 返回 basic_string 容器中保存的数据的长度。

参数:

 cpt_basic_string basic_string 容器。

返回值:

 basic_string 容器中包含的数据串的长度。

注意:

 如果 cpt_basic_string == NULL 或者 basic_string 容器一定已经初始化的容器, 否则函数的行为是未定义的。这个函数与 basic_string_size() 相同。

```
bool_t basic_string_empty(const basic_string_t* cpt_basic_string);
```

描述:

 测试 basic_string 容器是否为空。

参数:

 cpt_basic_string basic_string 容器。

返回值:

 如果 basic_string 为空则返回 true, 否则返回 false。

注意:

 如果 cpt_basic_string == NULL 或者 basic_string 容器一定已经初始化的容器, 否则函数的行为是未定义的。

```
size_t basic_string_max_size(const basic_string_t* cpt_basic_string);
```

描述:

 返回 basic_string 容器能够保存的数据的数量。

参数:

 cpt_basic_string basic_string 容器。

返回值:

 返回 basic_string 容器中能够保存数据的最大数量的可能值。

注意:

 cpt_basic_string == NULL 则函数的行为是未定义的, cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。这个最大数量是与系统相关的, 不是一个固定的值。

```
size_t basic_string_capacity(const basic_string_t* cpt_basic_string);
```

描述:

 返回 basic_string 容器的容量。

参数:

 cpt_basic_string basic_string 容器。

返回值:

 返回 basic_string 容器的容量。

注意:

 cpt_basic_string == NULL 则函数的行为是未定义的, cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器。

容器，否则函数的行为是未定义的。容量符合前面描述的算法。

```
void* basic_string_at(const basic_string_t* cpt_basic_string, size_t t_pos);
```

描述：

通过下标随机访问 basic_string 容器中的数据。

参数：

cpt_basic_string	指向 basic_string 容器的指针。
t_pos	要访问的数据在容器中的索引。

返回值：

指向被访问的数据的指针。

注意：

cpt_basic_string == NULL 则函数的行为是未定义的，cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果 t_pos 超过了 basic_string 容器中数据索引的范围，则函数的行为是未定义的。

```
bool_t basic_string_equal(  
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述：

测试两个容器是否相等。

参数：

cpt_first	指向 basic_string 容器的指针。
cpt_second	指向 basic_string 容器的指针。

返回值：

如果两个容器相等，返回 true，否则返回 false。

注意：

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的，cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同，那么这两个容器不等。如果两个容器中的数据个数相等并且对应数据相等则容器相等，如果 cpt_first == cpt_second 则返回 true。

```
bool_t basic_string_not_equal(  
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述：

测试两个容器是否不相等。

参数：

cpt_first	指向 basic_string 容器的指针。
cpt_second	指向 basic_string 容器的指针。

返回值：

如果两个容器不相等，返回 true，否则返回 false。

注意：

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的，cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同，那么这两个容器不等。如果两个容器中的数据个数不等或者对应数据不等则容器不等，如果 cpt_first == cpt_second 则返回

false。

```
bool_t basic_string_less(
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述:

测试第一个 basic_string 容器是否小于第二个 basic_string 容器。

参数:

cpt_first 指向 basic_string 容器的指针。
cpt_second 指向 basic_string 容器的指针。

返回值:

如果第一个 basic_string 容器小于第二个 basic_string 容器, 返回 true, 否则返回 false。

注意:

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同, 函数的行为是未定义的。如果第一个容器中的数据小于第二个容器中的对应数据则返回 true, 如果大于则返回 false, 如果对应数据都相等则, 如果第一个容器中的数据个数小于第二个容器中的数据个数则返回 true 否则返回 false, 如果 cpt_first == cpt_second 则返回 false。

```
bool_t basic_string_less_equal(
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述:

测试第一个 basic_string 容器是否小于等于第二个 basic_string 容器。

参数:

cpt_first 指向 basic_string 容器的指针。
cpt_second 指向 basic_string 容器的指针。

返回值:

如果第一个 basic_string 容器小于等于第二个 basic_string 容器, 返回 true, 否则返回 false。

注意:

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同, 函数的行为是未定义的。如果第一个容器中的数据小于第二个容器中的对应数据则返回 true, 如果大于则返回 false, 如果对应数据都相等则, 如果第一个容器中的数据个数小于等于第二个容器中的数据个数则返回 true 否则返回 false, 如果 cpt_first == cpt_second 则返回 true。

```
bool_t basic_string_greater(
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述:

测试第一个 basic_string 容器是否大于第二个 basic_string 容器。

参数:

cpt_first 指向 basic_string 容器的指针。
cpt_second 指向 basic_string 容器的指针。

返回值:

如果第一个 basic_string 容器大于第二个 basic_string 容器，返回 true，否则返回 false。

注意：

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的，cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同，函数的行为是未定义的。如果第一个容器中的数据大于第二个容器中的对应数据则返回 true，如果小于则返回 false，如果对应数据都相等则，如果第一个容器中的数据个数大于第二个容器中的数据个数则返回 true 否则返回 false，如果 cpt_first == cpt_second 则返回 false。

```
bool_t basic_string_greater_equal(
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述：

测试第一个 basic_string 容器是否大于等于第二个 basic_string 容器。

参数：

cpt_first 指向 basic_string 容器的指针。
cpt_second 指向 basic_string 容器的指针。

返回值：

如果第一个 basic_string 容器大于等于第二个 basic_string 容器，返回 true，否则返回 false。

注意：

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的，cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同，函数的行为是未定义的。如果第一个容器中的数据大于第二个容器中的对应数据则返回 true，如果小于则返回 false，如果对应数据都相等则，如果第一个容器中的数据个数大于等于第二个容器中的数据个数则返回 true 否则返回 false，如果 cpt_first == cpt_second 则返回 true。

```
bool_t basic_string_equal_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述：

测试 basic_string 容器和数据串是否相等。

参数：

cpt_basic_string basic_string 容器。
cpv_value_string 数据串。

返回值：

如果 basic_string 中的数据与 cpv_value_string 中的数据相等，返回 true，否则返回 false。

注意：

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果 basic_string 容器保存的数据与 cpv_value_string 中保存的数据类型不同，函数的行为是未定义的。如果 basic_string 容器中的数据与数据串中的对应数据都相等返回 true，否则返回 false。

```
bool_t basic_string_not_equal_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述：

测试 basic_string 容器和数据串是否不等。

参数:

cpt_basic_string basic_string 容器。
cpv_value_string 数据串。

返回值:

如果 basic_string 中的数据与 cpv_value_string 中的数据不等, 返回 true, 否则返回 false。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果 basic_string 容器保存的数据与 cpv_value_string 中保存的数据类型不同, 函数的行为是未定义的。

```
bool_t basic_string_less_cstr(  
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述:

测试 basic_string 容器是否小于数据串。

参数:

cpt_basic_string basic_string 容器。
cpv_value_string 数据串。

返回值:

如果 basic_string 容器中的数据小于 cpv_value_string, 返回 true, 否则返回 false。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果 basic_string 容器保存的数据与 cpv_value_string 中保存的数据类型不同, 函数的行为是未定义的。如果 basic_string 器中的数据小于数据串中的对应数据则返回 true, 如果大于则返回 false, 如果对应数据都相等则, 如果 basic_string 容器中的数据个数小于数据串中的数据个数则返回 true 否则返回 false。

```
bool_t basic_string_less_equal_cstr(  
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述:

测试 basic_string 容器是否小于等于数据串。

参数:

cpt_basic_string basic_string 容器。
cpv_value_string 数据串。

返回值:

如果 basic_string 容器中的数据小于等于 cpv_value_string, 返回 true, 否则返回 false。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果 basic_string 容器保存的数据与 cpv_value_string 中保存的数据类型不同, 函数的行为是未定义的。如果 basic_string 器中的数据小于数据串中的对应数据则返回 true, 如果大于则返回 false, 如果对应数据都相等则, 如果 basic_string 容器中的数据个数小于等于数据串中的数据个数则返回 true 否则返回 false。

```
bool_t basic_string_greater_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述:

测试 basic_string 容器是否大于数据串。

参数:

cpt_basic_string basic_string 容器。

cpv_value_string 数据串。

返回值:

如果 basic_string 容器中的数据大于 cpv_value_string, 返回 true, 否则返回 false。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果 basic_string 容器保存的数据与 cpv_value_string 中保存的数据类型不同, 函数的行为是未定义的。如果 basic_string 容器中的数据大于数据串中的对应数据则返回 true, 如果小于则返回 false, 如果对应数据都相等则, 如果 basic_string 容器中的数据个数大于数据串中的数据个数则返回 true 否则返回 false。

```
bool_t basic_string_greater_equal_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述:

测试 basic_string 容器是否大于等于数据串。

参数:

cpt_basic_string basic_string 容器。

cpv_value_string 数据串。

返回值:

如果 basic_string 容器中的数据大于等于 cpv_value_string, 返回 true, 否则返回 false。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果 basic_string 容器保存的数据与 cpv_value_string 中保存的数据类型不同, 函数的行为是未定义的。如果 basic_string 容器中的数据大于数据串中的对应数据则返回 true, 如果小于则返回 false, 如果对应数据都相等则, 如果 basic_string 容器中的数据个数大于等于数据串中的数据个数则返回 true 否则返回 false。

```
int basic_string_compare(
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述:

返回两个 basic_string 容器比较的结果。

参数:

cpt_first 第一个 basic_string 容器。

cpt_second 第二个 basic_string 容器。

返回值:

如果第一个 basic_string 容器中的数据大于第二个 basic_string 容器中的数据, 返回值大于 0, 小于, 返回值小于 0, 相等, 返回值等于 0。

注意:

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同, 函数的行为是未定义的。如果 cpt_first == cpt_second 则返回 0。

```
int basic_string_compare_substring_string(
    const basic_string_t* cpt_first, size_t t_pos, size_t t_len,
    const basic_string_t* cpt_second);
```

描述:

返回子 basic_string 容器与另一个 basic_string 容器比较的结果。

参数:

cpt_first 第一个 basic_string 容器。
t_pos 子 basic_string 的开始位置。
t_len 子 basic_string 的长度。
cpt_second 第二个 basic_string 容器。

返回值:

如果第一个 basic_string 容器的子串中的数据大于第二个 basic_string 容器中的数据, 返回值大于 0, 小于, 返回值小于 0, 相等, 返回值等于 0。

注意:

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同, 函数的行为是未定义的。t_pos 是属于第一个 basic_string 容器的有效的下标, 否则函数的行为是未定义的, t_len >= basic_string_size() - t_pos 时表示使用从 t_pos 开始的全部子串。

```
int basic_string_compare_substring_substring(
    const basic_string_t* cpt_first, size_t t_firstpos, size_t t_firstlen,
    const basic_string_t* cpt_second, size_t t_secondpos, size_t t_secondlen);
```

描述:

返回子 basic_string 容器与另一个 basic_string 容器比较的结果。

参数:

cpt_first 第一个 basic_string 容器。
t_firstpos 子 basic_string 的开始位置。
t_firstlen 子 basic_string 的长度。
cpt_second 第二个 basic_string 容器。
t_secondpos 子 basic_string 的开始位置。
t_secondlen 子 basic_string 的长度。

返回值:

如果第一个 basic_string 容器的子串中的数据大于第二个 basic_string 容器的子串中的数据, 返回值大于 0, 小于, 返回值小于 0, 相等, 返回值等于 0。

注意:

cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, cpt_first 和 cpt_second 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同, 函数的行为是未定义的。t_firstpos 是属于第一个 basic_string 容器的有效的下标, t_secondpos 是属于第二个 basic_string 容器的

有效下标，否则函数的行为是未定义的， $t_firstlen \geq basic_string_size() - t_firstpos$ 时表示使用从 $t_firstpos$ 开始的全部子串， $t_secondlen \geq basic_string_size() - t_secondpos$ 时表示使用从 $t_secondpos$ 开始的全部子串。

```
int basic_string_compare_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述：

返回 `basic_string` 容器与数据串的比较结果。

参数：

`cpt_basic_string` `basic_string` 容器。

`cpv_value_string` 数据串。

返回值：

如果 `basic_string` 容器中的数据大于数据串中的数据，返回值大于 0，小于，返回值小于 0，相等，返回值等于 0。

注意：

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器，否则函数的行为是未定义的。如果 `basic_string` 容器保存的数据与 `cpv_value_string` 中保存的数据类型不同，函数的行为是未定义的。

```
int basic_string_compare_substring_cstr(
    const basic_string_t* cpt_basic_string, size_t t_pos, size_t t_len,
    const void* cpv_value_string);
```

描述：

返回子 `basic_string` 容器与数据串的比较结果。

参数：

`cpt_basic_string` `basic_string` 容器。

`t_pos` 子 `basic_string` 的开始位置。

`t_len` 子 `basic_string` 的长度。

`cpv_value_string` 数据串。

返回值：

如果 `basic_string` 容器子串中的数据大于数据串中的数据，返回值大于 0，小于，返回值小于 0，相等，返回值等于 0。

注意：

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器，否则函数的行为是未定义的。如果 `basic_string` 容器保存的数据与 `cpv_value_string` 中保存的数据类型不同，函数的行为是未定义的。`t_pos` 是属于第一个 `basic_string` 容器的有效下标，否则函数的行为是未定义的， $t_len \geq basic_string_size() - t_pos$ 时表示使用从 `t_pos` 开始的全部子串。

```
int basic_string_compare_substring_subcstr(
    const basic_string_t* cpt_basic_string, size_t t_pos, size_t t_len,
    const void* cpv_value_string, size_t t_valuelen);
```

描述：

返回子 `basic_string` 容器与子数据串的比较结果。

参数：

cpt_basic_string basic_string 容器。
t_pos 子 basic_string 的开始位置。
t_len 子 basic_string 的长度。
cpv_value_string 数据串。
t_valuelen 子数据串长度。

返回值：

如果 basic_string 容器子串中的数据大于子数据串中的数据，返回值大于 0，小于，返回值小于 0，相等，返回值等于 0。

注意：

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果 basic_string 容器保存的数据与 cpv_value_string 中保存的数据类型不同，函数的行为是未定义的。t_pos 是属于第一个 basic_string 容器的有效的下标，否则函数的行为是未定义的，t_len >= basic_string_size() - t_pos 时表示使用从 t_pos 开始的全部子串。t_valuelen 大于等于数据串的长度则使用全部数据串。

```
basic_string_t* basic_string_substr(  
    const basic_string_t* cpt_basic_string, size_t t_pos, size_t t_len);
```

描述：

返回指定的子 basic_string 容器。

参数：

cpt_basic_string basic_string 容器。
t_pos 子 basic_string 的开始位置。
t_len 子 basic_string 的长度。

返回值：

返回指定的子 basic_string 容器。

注意：

cpt_basic_string == NULL 则函数的行为是未定义的，cpt_basic_string 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。t_pos 是属于 basic_string 容器的有效的下标，否则函数的行为是未定义的，t_len >= basic_string_size() - t_pos 时表示使用从 t_pos 开始的全部子串。返回的子串是已经初始化的 basic_string，用户在使用之后负责使用 basic_string_destroy() 释放资源。

```
void basic_string_connect(basic_string_t* pt_dest, const basic_string_t* cpt_src);
```

描述：

将两个 basic_string 容器链接。

参数：

pt_dest 目的 basic_string 容器。
cpt_src 源 basic_string 容器。

返回值：

无。

注意：

pt_dest == NULL 或者 cpt_src == NULL 则函数的行为是未定义的，pt_dest 和 cpt_src 必须是已经初始化的，否则函数的行为是未定义的。pt_dest 和 cpt_src 包含的数据类型必须是相同的，否则函数的行为是未定义的。

```
void basic_string_connect_elem(basic_string_t* pt_basic_string, elem);
```

描述:

将 basic_string 容器与指定的数据连接。

参数:

pt_basic_string	basic_string 容器。
elem	指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的，则函数的行为是未定义的。

```
void basic_string_connect_cstr(
    basic_string_t* pt_basic_string, const void* cpv_value_string);
```

描述:

将 basic_string 容器与指定的数据串连接。

参数:

pt_basic_string	目的 basic_string 容器。
cpv_value_string	源数据串。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，pt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。pt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的，否则函数的行为是未定义的。

```
size_t basic_string_find(
    const basic_string_t* cpt_basic_string, const basic_string_t* cpt_find,
    size_t t_pos);
```

描述:

在 basic_string 容器中查找指定的 basic_string。

参数:

cpt_basic_string	basic_string 容器。
cpt_find	要查找的 basic_string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpt_find == NULL 则函数的行为是未定义的，cpt_basic_string 和 cpt_find 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpt_find 包含的数据类型必须是相同的，否则函数的行为是未定义的。t_pos 是属于 cpt_basic_string 的有效位置，否则函数的行为是未定义的。如果 cpt_find 为空，则返回索引值。

```
size_t basic_string_find_cstr(
```

```
const basic_string_t* cpt_basic_string, const void* cpv_value_string,  
size_t t_pos);
```

描述:

在 basic_string 容器中查找指定的数据串。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，cpt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的，否则函数的行为是未定义的。t_pos 是属于 cpt_basic_string 的有效位置，否则函数的行为是未定义的。如果 cpv_value_string 为空，则返回索引值。

```
size_t basic_string_find_subcstr(  
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,  
    size_t t_pos, size_t t_len);
```

描述:

在 basic_string 容器中查找指定的子数据串。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。
t_len	数据串的长度。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，cpt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的，否则函数的行为是未定义的。t_pos 是属于 cpt_basic_string 的有效位置，否则函数的行为是未定义的。如果 cpv_value_string 为空或者 t_len == 0，则返回索引值。如果 t_len > 数据串的长度，则使用整个数据串。

```
size_t basic_string_find_elem(  
    const basic_string_t* cpt_basic_string, elem, size_t t_pos);
```

描述:

在 basic_string 容器中查找指定的数据。

参数:

cpt_basic_string	basic_string 容器。
elem	要查找的数据。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `elem` 的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `cpt_basic_string` 的有效位置，否则函数的行为是未定义的。

```
size_t basic_string_rfind(
    const basic_string_t* cpt_basic_string, const basic_string_t* cpt_find,
    size_t t_pos);
```

描述:

在 `basic_string` 容器中反向查找指定的 `basic_string`。

参数:

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>cpt_find</code>	要查找的 <code>basic_string</code> 容器。
<code>t_pos</code>	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 或者 `cpt_find == NULL` 则函数的行为是未定义的，`cpt_basic_string` 和 `cpt_find` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpt_find` 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 `t_pos >= basic_string_size(cpt_basic_string)` 则从 `basic_string` 的最后一个数据开始查找。如果 `cpt_find` 为空，则返回索引值。

```
size_t basic_string_rfind_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos);
```

描述:

在 `basic_string` 容器中反向查找数据串。

参数:

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>cpv_value_string</code>	要查找的数据串。
<code>t_pos</code>	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 `t_pos >= basic_string_size(cpt_basic_string)` 则从 `basic_string` 的最后一个数据开始查找。如果 `cpv_value_string` 为空，则返回索引值。

```
size_t basic_string_rfind_subcstr(
```

```
const basic_string_t* cpt_basic_string, const void* cpv_value_string,  
size_t t_pos, size_t t_len);
```

描述:

在 basic_string 容器中反向查找子数据串。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。
t_len	要查找的数据串的长度。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 `t_pos >= basic_string_size(cpt_basic_string)` 则从 `basic_string` 的最后一个数据开始查找。如果 `cpv_value_string` 为空或者 `t_len == 0`，则返回索引值。如果 `t_len >` 数据串的长度，则使用整个数据串。

```
size_t basic_string_rfind_elem(  
    const basic_string_t* cpt_basic_string, elem, size_t t_pos);
```

描述:

在 basic_string 容器中反向查找指定的数据。

参数:

cpt_basic_string	basic_string 容器。
elem	要查找的数据。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `elem` 的数据类型必须是相同的，否则函数的行为是未定义的。如果 `t_pos >= basic_string_size(cpt_basic_string)`，从最后一个字符开始查找。

```
size_t basic_string_find_first_of(  
    const basic_string_t* cpt_basic_string, const basic_string_t* cpt_find,  
    size_t t_pos);
```

描述:

在 basic_string 容器中查找第一个在指定的 basic_string 容器出现的数据。

参数:

cpt_basic_string	basic_string 容器。
cpt_find	要查找的 basic_string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符的索引，如果查找失败返回 NPOS。

注意：

`cpt_basic_string == NULL` 或者 `cpt_find == NULL` 则函数的行为是未定义的，`cpt_basic_string` 和 `cpt_find` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpt_find` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `cpt_basic_string` 的有效位置，否则函数的行为是未定义的。如果 `cpt_find` 为空，则返回索引值。

```
size_t basic_string_find_first_of_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos);
```

描述：

在 `basic_string` 容器中查找第一个在数据串中出现的数据。

参数：

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>cpv_value_string</code>	要查找的数据串。
<code>t_pos</code>	查找开始的位置。

返回值：

查找到的字符的索引，如果查找失败返回 NPOS。

注意：

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `cpt_basic_string` 的有效位置，否则函数的行为是未定义的。如果 `cpv_value_string` 为空，则返回索引值。

```
size_t basic_string_find_first_of_subcstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos, size_t t_len);
```

描述：

在 `basic_string` 容器中查找第一个在子数据串中出现的数据。

参数：

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>cpv_value_string</code>	要查找的数据串。
<code>t_pos</code>	查找开始的位置。
<code>t_len</code>	要查找的数据串的长度。

返回值：

查找到的字符的索引，如果查找失败返回 NPOS。

注意：

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `cpt_basic_string` 的有效位置，否则函数的行为是未定义的。如果 `cpv_value_string` 为空或者 `t_len == 0`，则返回索引值。如果 `t_len >` 数据串的长度，则使用整个数据串。

```
size_t basic_string_find_first_of_elem(
```

```
const basic_string_t* cpt_basic_string, elem, size_t t_pos);
```

描述:

在 basic_string 容器中查找第一个出现的指定数据。

参数:

cpt_basic_string	basic_string 容器。
elem	要查找的数据。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 则函数的行为是未定义的，cpt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 elem 的数据类型必须是相同的，否则函数的行为是未定义的。t_pos 是属于 cpt_basic_string 的有效位置，否则函数的行为是未定义的。

```
size_t basic_string_find_first_not_of(
    const basic_string_t* cpt_basic_string, const basic_string_t* cpt_find,
    size_t t_pos);
```

描述:

在 basic_string 容器中查找第一个不出现在指定 basic_string 容器中的数据。

参数:

cpt_basic_string	basic_string 容器。
cpt_find	要查找的 basic_string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpt_find == NULL 则函数的行为是未定义的，cpt_basic_string 和 cpt_find 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpt_find 包含的数据类型必须是相同的，否则函数的行为是未定义的。t_pos 是属于 cpt_basic_string 的有效位置，否则函数的行为是未定义的。如果 cpt_find 为空，则返回索引值。

```
size_t basic_string_find_first_not_of_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos);
```

描述:

在 basic_string 容器中查找第一个不出现在数据串中的数据。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。

返回值:

查找到的字符的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `cpt_basic_string` 的有效位置，否则函数的行为是未定义的。如果 `cpv_value_string` 为空，则返回索引值。

```
size_t basic_string_find_first_not_of_subcstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos, size_t t_len);
```

描述:

在 `basic_string` 容器中查找第一个不出现在子数据串中的数据。

参数:

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>cpv_value_string</code>	要查找的数据串。
<code>t_pos</code>	查找开始的位置。
<code>t_len</code>	要查找的数据串的长度。

返回值:

查找到的字符的索引，如果查找失败返回 `NPOS`。

注意:

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `cpt_basic_string` 的有效位置，否则函数的行为是未定义的。如果 `cpv_value_string` 为空或者 `t_len == 0`，则返回索引值。如果 `t_len >` 数据串的长度，则使用整个数据串。

```
size_t basic_string_find_first_not_of_elem(
    const basic_string_t* cpt_basic_string, elem, size_t t_pos);
```

描述:

在 `basic_string` 容器中查找第一个不是指定数据的数据。

参数:

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>elem</code>	要查找的数据。
<code>t_pos</code>	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 `NPOS`。

注意:

`cpt_basic_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `elem` 的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `cpt_basic_string` 的有效位置，否则函数的行为是未定义的。

```
size_t basic_string_find_last_of(
    const basic_string_t* cpt_basic_string, const basic_string_t* cpt_find,
    size_t t_pos);
```

描述:

在 basic_string 容器中查找最后一个出现在指定 basic_string 容器中的数据。

参数:

cpt_basic_string	basic_string 容器。
cpt_find	要查找的 basic_string 容器。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpt_find == NULL 则函数的行为是未定义的，cpt_basic_string 和 cpt_find 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpt_find 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 t_pos >= basic_string_size(cpt_basic_string) 则从 basic_string 的最后一个数据开始查找。如果 cpt_find 为空，则返回索引值。

```
size_t basic_string_find_last_of_cstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos);
```

描述:

在 basic_string 容器中查找最后一个出现在数据串中的数据。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，cpt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 t_pos >= basic_string_size(cpt_basic_string) 则从 basic_string 的最后一个数据开始查找。如果 cpv_value_string 为空，则返回索引值。

```
size_t basic_string_find_last_of_subcstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos, size_t t_len);
```

描述:

在 basic_string 容器中查找最后一个出现在子数据串中的数据。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。
t_len	要查找的数据串的长度。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 `t_pos >= basic_string_size(cpt_basic_string)` 则从 `basic_string` 的最后一个数据开始查找。如果 `cpv_value_string` 为空或者 `t_len == 0`，则返回索引值。如果 `t_len >` 数据串的长度，则使用整个数据串。

```
size_t basic_string_find_last_of_elem(  
    const basic_string_t* cpt_basic_string, elem, size_t t_pos);
```

描述:

在 `basic_string` 容器中查找最后一个出现的指定数据。

参数:

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>elem</code>	要查找的数据。
<code>t_pos</code>	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 则函数的行为是未定义的，`cpt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `elem` 的数据类型必须是相同的，否则函数的行为是未定义的。如果 `t_pos >= basic_string_size(cpt_basic_string)`，从最后一个字符开始查找。

```
size_t basic_string_find_last_not_of(  
    const basic_string_t* cpt_basic_string, const basic_string_t* cpt_find,  
    size_t t_pos);
```

描述:

在 `basic_string` 容器中查找最后一个不出现在指定 `basic_string` 容器中的数据

参数:

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>cpt_find</code>	要查找的 <code>basic_string</code> 容器。
<code>t_pos</code>	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

`cpt_basic_string == NULL` 或者 `cpt_find == NULL` 则函数的行为是未定义的，`cpt_basic_string` 和 `cpt_find` 必须是已经初始化的，否则函数的行为是未定义的。`cpt_basic_string` 和 `cpt_find` 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 `t_pos >= basic_string_size(cpt_basic_string)` 则从 `basic_string` 的最后一个数据开始查找。如果 `cpt_find` 为空，则返回索引值。

```
size_t basic_string_find_last_not_of_cstr(  
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,  
    size_t t_pos);
```

描述:

在 `basic_string` 容器中查找最后一个不出现在数据串中的数据。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，cpt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 t_pos >= basic_string_size(cpt_basic_string) 则从 basic_string 的最后一个数据开始查找。如果 cpv_value_string 为空，则返回索引值。

```
size_t basic_string_find_last_not_of_subcstr(
    const basic_string_t* cpt_basic_string, const void* cpv_value_string,
    size_t t_pos, size_t t_len);
```

描述:

在 basic_string 容器中查找最后一个不出现在子数据串中的数据。

参数:

cpt_basic_string	basic_string 容器。
cpv_value_string	要查找的数据串。
t_pos	查找开始的位置。
t_len	要查找的数据串的长度。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的，cpt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的，否则函数的行为是未定义的。如果 t_pos >= basic_string_size(cpt_basic_string) 则从 basic_string 的最后一个数据开始查找。如果 cpv_value_string 为空或者 t_len == 0，则返回索引值。如果 t_len > 数据串的长度，则使用整个数据串。

```
size_t basic_string_find_last_not_of_elem(
    const basic_string_t* cpt_basic_string, elem, size_t t_pos);
```

描述:

在 basic_string 容器中查找最后一个不是指定数据的数据。

参数:

cpt_basic_string	basic_string 容器。
elem	要查找的数据。
t_pos	查找开始的位置。

返回值:

查找到的数据串的索引，如果查找失败返回 NPOS。

注意:

cpt_basic_string == NULL 则函数的行为是未定义的，cpt_basic_string 必须是已经初始化的，否则函数的行为是未定义的。cpt_basic_string 和 elem 的数据类型必须是相同的，否则函数的行为是未定义的。如果 t_pos >=

`basic_string_size(cpt_basic_string)`, 从最后一个字符开始查找。

```
basic_string_iterator_t basic_string_begin(const basic_string_t* cpt_basic_string);
```

描述:

返回指向 `basic_string` 容器开头的迭代器。

参数:

`cpt_basic_string` `basic_string` 容器。

返回值:

返回指向 `basic_string` 容器开头的迭代器。

注意:

`cpt_basic_string == NULL` 则函数的行为是未定义的, `cpt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器, 否则函数的行为是未定义的。如果 `cpt_basic_string` 为空, 则返回值与 `basic_string_end(cpt_basic_string)` 相等。

```
basic_string_iterator_t basic_string_end(const basic_string_t* cpt_basic_string);
```

描述:

返回指向 `basic_string` 容器末尾的迭代器。

参数:

`cpt_basic_string` `basic_string` 容器。

返回值:

返回指向 `basic_string` 容器末尾的迭代器。

注意:

`cpt_basic_string == NULL` 则函数的行为是未定义的, `cpt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器, 否则函数的行为是未定义的。

```
void basic_string_clear(basic_string_t* pt_basic_string);
```

描述:

清空 `basic_string` 容器。

参数:

`pt_basic_string` `basic_string` 容器。

返回值:

无。

注意:

`pt_basic_string == NULL` 则函数的行为是未定义的, `pt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器, 否则函数的行为是未定义的。

```
void basic_string_swap(basic_string_t* pt_first, basic_string_t* pt_second);
```

描述:

交换两个 `basic_string` 容器的内容。

参数:

`pt_first` `basic_string` 容器。

`pt_second` `basic_string` 容器。

返回值:

无。

注意：

pt_first == NULL 或者 pt_second == NULL 则函数的行为是未定义的，pt_first 和 pt_second 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同，函数的行为是未定义的。如果 basic_string_equal(pt_first, pt_second) 这函数不执行任何动作。

```
void basic_string_reserve(basic_string_t* pt_basic_string, size_t t_reservesize);
```

描述：

重新设置 basic_string 容器的容量。

参数：

pt_basic_string basic_string 容器。
t_reservesize 修改后的容量。

返回值：

无。

注意：

pt_basic_string == NULL 则函数的行为是未定义的，pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果 t_reservesize > basic_string_capacity() 则函数执行后，basic_string 容器的容量扩充到 t_reservesize。否则函数执行后 basic_string 的容量不变。basic_string 的容量只能增大不能够减小。

```
void basic_string_assign(basic_string_t* pt_dest, const basic_string_t* cpt_src);
```

描述：

使用指定的 basic_string 容器为 basic_string 容器赋值。

参数：

pt_dest 目的 basic_string 容器。
cpt_src 源 basic_string 容器。

返回值：

无。

注意：

pt_dest == NULL 或者 cpt_src == NULL 则函数的行为是未定义的，pt_dest 和 cpt_src 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同，函数的行为是未定义的。赋值以后两个容器的 basic_string_size() 相等但是 basic_string_capacity() 不一定相等。如果 basic_string_equal(pt_dest, cpt_src) 函数不执行任何操作。

```
void basic_string_assign_substring(  
    basic_string_t* pt_dest, const basic_string_t* cpt_src,  
    size_t t_pos, size_t t_len);
```

描述：

使用指定的子 basic_string 为 basic_string 容器赋值。

参数：

pt_dest 目的 basic_string 容器。
cpt_src 源 basic_string 容器。
t_pos 子字符串开始的位置。

t_len 子字符串长度。

返回值:

无。

注意:

pt_dest == NULL 或者 cpt_src == NULL 则函数的行为是未定义的, pt_dest 和 cpt_src 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同, 函数的行为是未定义的。t_pos 必须是属于源字符串的有效位置, 否则函数的行为是未定义的, 如果 t_len == NPOS 或者 t_pos + t_len >= 元字符串的长度, 那么使用从 t_pos 开始的所有子字符串。

```
void basic_string_assign_cstr(
    basic_string_t* pt_basic_string, const void* cpv_value_string);
```

描述:

使用指定的数据串为 basic_string 容器赋值。

参数:

pt_basic_string basic_string 容器。

cpv_value_string 指定的数据串。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的。pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。

```
void basic_string_assign_subcstr(
    basic_string_t* pt_basic_string, const void* cpv_value_string, size_t t_len);
```

描述:

使用指定的子数据串为 basic_string 容器赋值。

参数:

pt_basic_string basic_string 容器。

cpv_value_string 指定的数据串。

t_len 子串的长度。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的。pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。如果 t_len 等于 NPOS 或者超过了数据串的长度则使用整个数据串进行初始化。

```
void basic_string_assign_range(
    basic_string_t* pt_basic_string,
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end)
```

描述:

使用指定的数据区间为 basic_string 容器赋值。

参数:

- pt_basic_string 目的 basic_string 容器。
- it_begin 指定范围的开始。
- it_end 指定范围的末尾。

返回值:

无。

注意:

pt_basic_string == NULL 则函数的行为是未定义的, pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围并且保存的数据类型与 basic_string 容器中保存的数据类型相同, 否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 pt_basic_string 则函数的行为是未定义的。

```
void basic_string_assign_elem(basic_string_t* pt_basic_string, size_t t_count, elem)
```

描述:

使用指定的数据为 basic_string 容器赋值。

参数:

- pvec_vector 目的 basic_string 容器。
- t_count 赋值的数据的个数。
- elem 赋值的数据。

返回值:

无。

注意:

pt_basic_string == NULL 则函数的行为是未定义的, pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。elem 是与 basic_string 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
void basic_string_append(basic_string_t* pt_dest, const basic_string_t* cpt_src);
```

描述:

向 basic_string 容器末尾添加指定的 basic_string 容器。

参数:

- pt_dest 目的 basic_string 容器。
- cpt_src 源 basic_string 容器。

返回值:

无。

注意:

pt_dest == NULL 或者 cpt_src == NULL 则函数的行为是未定义的, pt_dest 和 cpt_src 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同, 函数的行为是未定义的。

```
void basic_string_append_substring(  
    basic_string_t* pt_dest, const basic_string_t* cpt_src,  
    size_t t_pos, size_t t_len);
```

描述:

向 basic_string 容器末尾添加指定的子 basic_string。

参数:

pt_dest	目的 basic_string 容器。
cpt_src	源 basic_string 容器。
t_pos	子字符串开始的位置。
t_len	子字符串长度。

返回值:

无。

注意:

如果 pt_dest == NULL 或者 cpt_src == NULL 则函数的行为是未定义的。pt_dest 和 cpt_src 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。如果两个 basic_string 容器保存的数据类型不同，函数的行为是未定义的。t_pos 必须是属于源字符串的有效位置，否则函数的行为是未定义的，如果 t_len == NPOS 或者 t_pos + t_len >= 元字符串的长度，那么使用从 t_pos 开始的所有子字符串。

```
void basic_string_append_cstr(
    basic_string_t* pt_basic_string, const void* cpv_value_string);
```

描述:

向 basic_string 容器末尾添加指定的数据串。

参数:

pt_basic_string	basic_string 容器。
cpv_value_string	指定的数据串。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的。pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。

```
void basic_string_append_subcstr(
    basic_string_t* pt_basic_string, const void* cpv_value_string, size_t t_len);
```

描述:

向 basic_string 容器末尾添加指定的子数据串。

参数:

pt_basic_string	basic_string 容器。
cpv_value_string	指定的数据串。
t_len	子串的长度。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的。pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。如果 t_len 等于 NPOS 或者超过了数据串的长度则使用整个数据串进行初始化。

```
void basic_string_append_range(
    basic_string_t* pt_basic_string,
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end)
```

描述:

向 basic_string 容器末尾添加指定的数据区间。

参数:

pt_basic_string 目的 basic_string 容器。
it_begin 指定范围的开始。
it_end 指定范围的末尾。

返回值:

无。

注意:

pt_basic_string == NULL 则函数的行为是未定义的, pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围并且保存的数据类型与 basic_string 容器中保存的数据类型相同, 否则函数的行为是未定义的。如果[it_begin, it_end) 属于 pt_basic_string 则函数的行为是未定义的。

```
void basic_string_append_elem(basic_string_t* pt_basic_string, size_t t_count, elem)
```

描述:

向 basic_string 容器末尾添加指定的数据。

参数:

pt_basic_string 目的 basic_string 容器。
t_count 赋值的数据的个数。
elem 赋值的数据。

返回值:

无。

注意:

pt_basic_string == NULL 则函数的行为是未定义的, pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。elem 是与 basic_string 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。

```
basic_string_iterator_t basic_string_insert(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_pos, elem);
```

描述:

向 basic_string 容器中插入指定数据。

参数:

pt_basic_string basic_string 容器。
it_pos 插入数据的位置。
elem 用户指定的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`it_pos` 必须是属于 `basic_string` 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。

```
basic_string_iterator_t basic_string_insert_n(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_pos,
    size_t t_count, elem);
```

描述:

向 `basic_string` 容器的指定位置插入多个数据。

参数:

<code>pt_basic_string</code>	<code>basic_string</code> 容器。
<code>it_pos</code>	插入数据的位置。
<code>t_count</code>	数据个数。
<code>elem</code>	用户指定的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`it_pos` 必须是属于 `basic_string` 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。

```
void basic_string_insert_string(
    basic_string_t* pt_basic_string, size_t t_pos,
    const basic_string_t* cpt_insert);
```

描述:

向 `basic_string` 容器中插入指定的 `basic_string` 容器。

参数:

<code>cpt_basic_string</code>	<code>basic_string</code> 容器。
<code>t_pos</code>	插入的位置。
<code>cpt_insert</code>	要插入的 <code>basic_string</code> 容器。

返回值:

无。

注意:

`pt_basic_string == NULL` 或者 `cpt_insert == NULL` 则函数的行为是未定义的，`pt_basic_string` 和 `cpt_insert` 必须是已经初始化的，否则函数的行为是未定义的。`pt_basic_string` 和 `cpt_insert` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `pt_basic_string` 的有效位置，否则函数的行为是未定义的。如果 `pt_basic_string == cpt_insert`，则函数的行为是未定义的。

```
void basic_string_insert_substring(
    basic_string_t* pt_basic_string, size_t t_pos,
    const basic_string_t* cpt_insert, size_t t_startpos, t_len);
```

描述:

向 basic_string 容器中插入指定的子 basic_string。

参数:

cpt_basic_string	basic_string 容器。
t_pos	插入的位置。
cpt_insert	要插入的 basic_string 容器。
t_startpos	子 basic_string 的开始位置。
t_len	子 basic_string 的长度。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpt_insert == NULL 则函数的行为是未定义的, pt_basic_string 和 cpt_insert 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpt_insert 包含的数据类型必须是相同的, 否则函数的行为是未定义的。t_pos 是属于 pt_basic_string 的有效位置, 否则函数的行为是未定义的。t_startpos 是属于 cpt_insert 的有效位置, 否则函数的行为是未定义的。如果 t_len == NPOS 或者 t_len + t_startpos 大于等于 cpt_insert 的长度, 则使用从 t_startpos 开始的全部 cpt_insert 子串。如果 pt_basic_string == cpt_insert, 则函数的行为是未定义的。

```
void basic_string_insert_cstr(
    basic_string_t* pt_basic_string, size_t t_pos, const void* cpv_value_string);
```

描述:

向 basic_string 容器中插入指定的数据串。

参数:

cpt_basic_string	basic_string 容器。
t_pos	插入的位置。
cpv_value_string	要插入的数据串。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, pt_basic_string 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的, 否则函数的行为是未定义的。t_pos 是属于 pt_basic_string 的有效位置, 否则函数的行为是未定义的。

```
void basic_string_insert_subcstr(
    basic_string_t* pt_basic_string, size_t t_pos,
    const void* cpv_value_string, size_t t_len);
```

描述:

向 basic_string 容器中插入指定的子数据串。

参数:

pt_basic_string	basic_string 容器。
t_pos	插入的位置。
cpv_value_string	指定的数据串。
t_len	子串的长度。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的。`pt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器, 否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。`t_pos` 是属于 `pt_basic_string` 的有效位置, 否则函数的行为是未定义的。如果 `t_len` 等于 `NPOS` 或者超过了数据串的长度则使用整个数据串进行初始化。

```
void basic_string_insert_range(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_pos,
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end)
```

描述:

向 `basic_string` 容器中插入指定的数据区间。

参数:

`pt_basic_string` 目的 `basic_string` 容器。
`it_pos` 插入的位置。
`it_begin` 指定范围的开始。
`it_end` 指定范围的末尾。

返回值:

无。

注意:

`pt_basic_string == NULL` 则函数的行为是未定义的, `pt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器, 否则函数的行为是未定义的。`it_pos` 是属于 `pt_basic_string` 并且是有效的迭代器, 否则函数的行为是未定义的。`[it_begin, it_end)` 是有效的范围并且保存的数据类型与 `basic_string` 容器中保存的数据类型相同, 否则函数的行为是未定义的。如果`[it_begin, it_end)` 属于 `pt_basic_string` 则函数的行为是未定义的。

```
void basic_string_insert_elem(
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_count, elem)
```

描述:

向 `basic_string` 容器中插入指定数据。

参数:

`pt_basic_string` 目的 `basic_string` 容器。
`it_pos` 插入的位置。
`t_count` 插入的数据的个数。
`elem` 插入的数据。

返回值:

无。

注意:

`pt_basic_string == NULL` 则函数的行为是未定义的, `pt_basic_string` 指向的容器是经过初始化以后的 `basic_string` 容器, 否则函数的行为是未定义的。`elem` 是与 `basic_string` 容器中保存的数据类型兼容的数据, 否则函数的行为是未定义的。`it_pos` 是属于 `pt_basic_string` 并且是有效的迭代器, 否则函数的行为是未定义的。

```
void basic_string_push_back(basic_string_t* pt_basic_string, elem);
```

描述:

向 basic_string 容器末尾添加指定数据。

参数:

pt_basic_string basic_string 容器。
elem 指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的则函数的行为是未定义的。

```
void basic_string_resize(  
    basic_string_t* pt_basic_string, size_t t_resize, elem);
```

描述:

重新设置 basic_string 容器中数据的个数。

参数:

pt_basic_string basic_string 容器。
t_resize 数据个数。
elem 用户指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的 basic_string 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用指定的数据填充。

```
basic_string_iterator_t basic_string_erase(  
    basic_string_t* pt_basic_string, basic_string_iterator_t it_pos);
```

描述:

删除 basic_string 容器中指定位置的数据。

参数:

pt_basic_string basic_string 容器。
it_pos 插入的位置。

返回值:

指向被删除的数据后面数据的迭代器。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的 basic_string 则函数的行为是未定义的。it_pos 是属于 basic_string 的有效的迭代器，否则函数的行为是未定义的。

```
basic_string_iterator_t basic_string_erase_range(  
    basic_string_t* pt_basic_string,  
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end)
```

描述:

删除 basic_string 容器中指定的数据区间。

参数:

pt_basic_string basic_string 容器。
it_begin 指定范围的开始。
it_end 指定范围的末尾。

返回值:

指向被删除的数据后面数据的迭代器。

注意:

pt_basic_string == NULL 则函数的行为是未定义的, pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围, 否则函数的行为是未定义的。

```
void basic_string_erase_substring(  
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_len);
```

描述:

删除 basic_string 容器中的子数据串。

参数:

pt_basic_string basic_string 容器。
t_pos 删除的位置。
t_len 子串的长度。

返回值:

无。

注意:

如果 pt_basic_string == NULL 则函数的行为是未定义的。pt_basic_string 指向的容器是经过初始化以后的 basic_string 容器, 否则函数的行为是未定义的。t_pos 是属于 pt_basic_string 的有效位置, 否则函数的行为是未定义的。如果 t_len 等于 NPOS 或者超过了数据串的长度则删除从 t_pos 开始剩余全部的字串。

```
void basic_string_replace(  
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_len,  
    const basic_string_t* cpt_replace);
```

描述:

将 basic_string 容器中的子数据串替换成指定的 basic_string。

参数:

pt_basic_string basic_string 容器。
t_pos 替换的位置。
t_len 替换的长度。
cpt_replace 指定的 basic_string。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpt_replace == NULL 则函数的行为是未定义的, pt_basic_string 和 cpt_replace 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpt_replace 包含的数据类型必须是相同的, 否则函数的行为是未定义的。t_pos 是属于 pt_basic_string 的有效位置, 否则函数的行为是未定义的。如果 t_len == NPOS 或者 t_len + t_pos 大于等于 pt_basic_string 的长度, 则使用从 t_pos 开始的全部 pt_basic_string 子串。如果 pt_basic_string == cpt_replace, 则函数的行为是未定义的。

```
void basic_string_replace_substring(
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_len,
    const basic_string_t* cpt_replace, size_t t_position, size_t t_length);
```

描述:

将 basic_string 容器中的子数据串替换成指定的子 basic_string。

参数:

pt_basic_string	basic_string 容器。
t_pos	替换的位置。
t_len	替换的长度。
cpt_replace	指定的 basic_string。
t_position	拷贝的位置。
t_length	拷贝的长度。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpt_replace == NULL 则函数的行为是未定义的, pt_basic_string 和 cpt_replace 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpt_replace 包含的数据类型必须是相同的, 否则函数的行为是未定义的。t_pos 是属于 pt_basic_string 的有效位置, 否则函数的行为是未定义的。t_position 是属于 cpt_replace 的有效位置, 否则函数的行为是未定义的。如果 t_len == NPOS 或者 t_len + t_pos 大于等于 pt_basic_string 的长度, 则使用从 t_pos 开始的全部 pt_basic_string 子串。如果 t_length == NPOS 或者 t_length + t_position 大于等于 cpt_replace 的长度, 则使用从 t_position 开始的全部 cpt_replace 子串。如果 pt_basic_string == cpt_replace, 则函数的行为是未定义的。

```
void basic_string_replace_cstr(
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_len,
    const void* cpv_value_string);
```

描述:

将 basic_string 容器中的子数据串替换成指定的数据串。

参数:

pt_basic_string	basic_string 容器。
t_pos	替换的位置。
t_len	替换的长度。
cpv_value_string	指定的数据串。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, pt_basic_string 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的, 否则函数的行为是未定义的。t_pos 是属于 pt_basic_string 的有效位置, 否则函数的行为是未定义的。如果 t_len == NPOS 或者 t_len + t_pos 大于等于 pt_basic_string 的长度, 则使用从 t_pos 开始的全部 pt_basic_string 子串。

```
void basic_string_replace_subcstr(
```

```
basic_string_t* pt_basic_string, size_t t_pos, size_t t_len,
const void* cpv_value_string, size_t t_length);
```

描述:

将 basic_string 容器中的子数据串替换成指定的子数据串。

参数:

pt_basic_string	basic_string 容器。
t_pos	替换的位置。
t_len	替换的长度。
cpv_value_string	指定的数据串。
t_length	数据串的长度。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `cpv_value_string == NULL` 则函数的行为是未定义的，`pt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`pt_basic_string` 和 `cpv_value_string` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`t_pos` 是属于 `pt_basic_string` 的有效位置，否则函数的行为是未定义的。如果 `t_len == NPOS` 或者 `t_len + t_pos` 大于等于 `pt_basic_string` 的长度，则使用从 `t_pos` 开始的全部 `pt_basic_string` 子串。如果 `t_length == NPOS` 或者 `t_length` 大于等于 `cpv_value_string` 的长度，则使用全部 `cpv_value_string` 子串。

```
void basic_string_replace_elem(
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_len,
    size_t t_count, elem);
```

描述:

将 basic_string 容器中的子数据串替换成指定的数据。

参数:

pt_basic_string	basic_string 容器。
t_pos	替换的开始位置。
t_len	替换的长度。
t_count	数据个数。
elem	用户指定的数据。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`t_pos` 必须在 `basic_string` 有效范围内，否则函数的行为是未定义的，`t_len == NPOS` 或者 `t_pos + t_len >= basic_string_size()`，将 `t_pos` 开始到末尾的全部串替换。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。

```
void basic_string_range_replace(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_begin,
    basic_string_iterator_t it_end, const basic_string_t* cpt_replace);
```

描述:

将 basic_string 容器中的数据区间替换成 basic_string。

参数:

pt_basic_string basic_string 容器。
it_begin 指定范围的开始。
it_end 指定范围的末尾。
cpt_replace 指定的 basic_string。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpt_replace == NULL 则函数的行为是未定义的, pt_basic_string 和 cpt_replace 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpt_replace 包含的数据类型必须是相同的, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围, 否则函数的行为是未定义的。如果 pt_basic_string == cpt_replace, 则函数的行为是未定义的。

```
void basic_string_range_replace_substring(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_begin,
    basic_string_iterator_t it_end, const basic_string_t* cpt_replace,
    size_t t_position, size_t t_length);
```

描述:

将 basic_string 容器中的子数据串替换成指定的子 basic_string。

参数:

pt_basic_string basic_string 容器。
it_begin 指定范围的开始。
it_end 指定范围的末尾。
cpt_replace 指定的 basic_string。
t_position 拷贝的位置。
t_length 拷贝的长度。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpt_replace == NULL 则函数的行为是未定义的, pt_basic_string 和 cpt_replace 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpt_replace 包含的数据类型必须是相同的, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围, 否则函数的行为是未定义的。t_position 是属于 cpt_replace 的有效位置, 否则函数的行为是未定义的。如果 t_length == NPOS 或者 t_length + t_position 大于等于 cpt_replace 的长度, 则使用从 t_position 开始的全部 cpt_replace 子串。如果 pt_basic_string == cpt_replace, 则函数的行为是未定义的。

```
void basic_string_range_replace_cstr(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_begin,
    basic_string_iterator_t it_end, const void* cpv_value_string);
```

描述:

将 basic_string 容器中的数据区间替换成数据串。

参数:

pt_basic_string basic_string 容器。
it_begin 指定范围的开始。

it_end 指定范围的末尾。

cpv_value_string 指定的数据串。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, pt_basic_string 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围, 否则函数的行为是未定义的。

```
void basic_string_range_replace_subcstr(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_begin,
    basic_string_iterator_t it_end, const void* cpv_value_string, size_t t_length);
```

描述:

将 basic_string 容器中的数据区间替换成子数据区间。

参数:

pt_basic_string basic_string 容器。

it_begin 指定范围的开始。

it_end 指定范围的末尾。

cpv_value_string 指定的数据串。

t_length 数据串的长度。

返回值:

无。

注意:

pt_basic_string == NULL 或者 cpv_value_string == NULL 则函数的行为是未定义的, pt_basic_string 必须是已经初始化的, 否则函数的行为是未定义的。pt_basic_string 和 cpv_value_string 包含的数据类型必须是相同的, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围, 否则函数的行为是未定义的。如果 t_length == NPOS 或者 t_length 大于等于 cpv_value_string 的长度, 则使用全部 cpv_value_string 子串。

```
void basic_string_range_replace_elem(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_begin,
    basic_string_iterator_t it_end, size_t t_count, elem);
```

描述:

将 basic_string 容器中的数据区间替换成指定的数据。

参数:

pt_basic_string basic_string 容器。

it_begin 指定范围的开始。

it_end 指定范围的末尾。

t_count 数据个数。

elem 用户指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的 basic_string 则函数的行为是未定义的。[it_begin,

`it_end`)是有效的范围，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。

```
void basic_string_replace_range(
    basic_string_t* pt_basic_string,
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end,
    basic_string_iterator_t it_first, basic_string_iterator_t it_last);
```

描述:

将 `basic_string` 容器中的子数据串替换成指定的子 `basic_string`。

参数:

<code>pt_basic_string</code>	<code>basic_string</code> 容器。
<code>it_begin</code>	指定范围的开始。
<code>it_end</code>	指定范围的末尾。
<code>it_first</code>	指定范围的开始。
<code>it_last</code>	指定范围的末尾。

返回值:

无。

注意:

`pt_basic_string == NULL` 则函数的行为是未定义的，`pt_basic_string` 必须是已经初始化的，否则函数的行为是未定义的。`pt_basic_string` 和 `[it_first, it_last)` 包含的数据类型必须是相同的，否则函数的行为是未定义的。`[it_begin, it_end)` 是有效的范围，否则函数的行为是未定义的。如果 `[it_first, it_last)` 属于 `pt_basic_string`，则函数的行为是未定义的。

第五节 迭代器接口

迭代器接口主要是为实现迭代器操作提供的，这些接口有迭代器的外部接口使用，用户不应该直接使用这些接口。这些接口是与容器相关的，`basic_string` 的迭代器接口只能够处理 `basic_string` 容器的迭代器，对于其他容器的迭代器无能为力。

<code>_create_basic_string_iterator</code>	创建 <code>basic_string</code> 容器的迭代器。
<code>_basic_string_iterator_equal</code>	测试两个 <code>basic_string</code> 迭代器是否相等。
<code>_basic_string_iterator_less</code>	测试第一个 <code>basic_string</code> 迭代器是否小于第二个 <code>basic_string</code> 迭代器。
<code>_basic_string_iterator_before</code>	测试第一个 <code>basic_string</code> 迭代器是否在第二个 <code>basic_string</code> 迭代器前面。
<code>_basic_string_iterator_get_value</code>	获得迭代器引用的数据。
<code>_basic_string_iterator_set_value</code>	设置迭代器引用的数据。
<code>_basic_string_iterator_get_pointer</code>	获得指向迭代器引用的数据的指针。
<code>_basic_string_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_basic_string_iterator_prev</code>	获得引用上一个数据的迭代器。
<code>_basic_string_iterator_next_n</code>	获得引用向下第 n 个数据的迭代器。

<code>_basic_string_iterator_prev_n</code>	获得引用向上第 n 个数据的迭代器。
<code>_basic_string_iterator_at</code>	通过下标来访问迭代器引用的数据。
<code>_basic_string_iterator_minus</code>	获得两个迭代器之差。

`basic_string_iterator_t _create_basic_string_iterator(void);`

描述:

创建 basic_string 迭代器。

参数:

无。

返回值:

basic_string 迭代器。

注意:

获得的 basic_string 迭代器不是有效的迭代器，因为它并没有与任何 basic_string 容器关联。这个函数创建的 basic_string 迭代器是供给以后的 basic_string 函数使用的。

`bool_t _basic_string_iterator_equal(basic_string_iterator_t it_first, basic_string_iterator_t it_second);`

描述:

比较两个迭代器是否相等。

参数:

`it_first` 第一个 basic_string 迭代器。
`it_second` 第二个 basic_string 迭代器。

返回值:

如果两个迭代器相等则返回 true，否则返回 false。

注意:

两个迭代器必须属于同一个 basic_string 容器，否则函数的行为是未定义的。两个迭代器相等表示的是两个迭代器引用的是同一个容器中的同一个数据。

`bool_t _basic_string_iterator_less(basic_string_iterator_t it_first, basic_string_iterator_t it_second);`

描述:

测试第一个迭代器是否小于第二个迭代器。

参数:

`it_first` 第一个 basic_string 迭代器。
`it_second` 第二个 basic_string 迭代器。

返回值:

如果第一个迭代器小于第二个迭代器则返回 true，否则返回 false。

注意:

两个迭代器必须属于同一个 basic_string 容器，否则函数的行为是未定义的。第一个迭代器小于第二个迭代器表示的是第一个迭代器引用的数据比第二个迭代器引用的数据更靠近容器中的第一个数据。

```
bool_t _basic_string_iterator_before(
    basic_string_iterator_t it_first, basic_string_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个第二个迭代器前面。

参数:

it_first 第一个 basic_string 迭代器。
it_second 第二个 basic_string 迭代器。

返回值:

如果第一个迭代器在第二个迭代器前面则返回 true, 否则返回 false。

注意:

两个迭代器必须属于同一个 basic_string 容器, 否则函数的行为是未定义的。第一个迭代器在第二个迭代器前面表示的是第一个迭代器引用的数据比第二个迭代器引用的数据更靠近容器中的第一个数据。

```
void _basic_string_iterator_get_value(
    basic_string_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter basic_string 迭代器。
pv_value 保存数据的缓存区。

返回值:

无。

注意:

it_iter 必须是属于 basic_string 的有效迭代器, 否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的, pv_value 的缓冲区要能够保存 it_iter 引用的数据, 否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 中。

```
void _basic_string_iterator_set_value(
    basic_string_iterator_t it_iter, const void* cpv_value);
```

描述:

设置迭代器引用的数据。

参数:

it_iter basic_string 迭代器。
cpv_value 保存数据的缓存区。

返回值:

无。

注意:

it_iter 必须是属于 basic_string 的有效迭代器, 否则函数的行为是未定义的。cpv_value == NULL 则函数的行为是未定义的, cpv_value 的缓冲区中保存的数据要与 it_iter 引用的数据类型相同, 否则函数的行为是未定义的。函数执行后 cpv_value 中保存的数据被拷贝到 it_iter 引用的数据中。

```
const void* _basic_string_iterator_get_pointer(basic_string_iterator_t it_iter);
```

描述:

返回被迭代器应用的数据的指针。

参数:

it_iter basic_string 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 必须是属于 basic_string 的有效迭代器, 否则函数的行为是未定义的。

```
basic_string_iterator_t _basic_string_iterator_next(
    basic_string_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter basic_string 迭代器。

返回值:

引用下一个数据的迭代器。

注意:

it_iter 必须是属于 basic_string 的有效迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是属于 basic_string 的有效的迭代器, 否则函数行为是未定义的。

```
basic_string_iterator_t _basic_string_iterator_prev(
    basic_string_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter basic_string 迭代器。

返回值:

引用上一个数据的迭代器。

注意:

it_iter 必须是属于 basic_string 的有效迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是属于 basic_string 的有效的迭代器, 否则函数行为是未定义的。

```
basic_string_iterator_t _basic_string_iterator_next_n(
    basic_string_iterator_t it_iter, int n_step);
```

描述:

获得引用向下第 n 个数据的迭代器。

参数:

it_iter basic_string 迭代器。

n_step 前进的步数。

返回值:

引用向下第 n 个数据的迭代器。

注意:

it_iter 必须是属于 basic_string 的有效迭代器，否则函数的行为是未定义的。引用向下第 n 个数据的迭代器 也必须是属于 basic_string 的有效的迭代器，否则函数行为是未定义的。n_step > 0 表示迭代器向容器末尾移动 n 步，n_step < 0 表示迭代器向容器开头移动 n 步，n_step == 0 表示不移动。

```
basic_string_iterator_t _basic_string_iterator_prev_n(
    basic_string_iterator_t it_iter, int n_step);
```

描述:

获得引用向上第 n 个数据的迭代器。

参数:

it_iter basic_string 迭代器。
n_step 前进的步数。

返回值:

引用向上第 n 个数据的迭代器。

注意:

it_iter 必须是属于 basic_string 的有效迭代器，否则函数的行为是未定义的。引用向上第 n 个数据的迭代器 也必须是属于 basic_string 的有效的迭代器，否则函数行为是未定义的。n_step > 0 表示迭代器向容器开头移动 n 步，n_step < 0 表示迭代器向容器末尾移动 n 步，n_step == 0 表示不移动。

```
void* _basic_string_iterator_at(basic_string_iterator_t it_iter, int n_index);
```

描述:

使用下标随机访问迭代器引用的数据。

参数:

it_iter basic_string 迭代器。
n_index 访问的下标。

返回值:

获得随机访问的迭代器应用的数据的指针。

注意:

it_iter 必须是属于 basic_string 的有效迭代器，否则函数的行为是未定义的。随机访问的数据也必须在 basic_string 范围内，否则函数行为是未定义的。n_index > 0 表示访问从当前迭代器引用的数据起向容器末尾的第 n 个数据，n_index < 0 表示从当前迭代器引用的数据起向容器开头的第 n 个数据，n_index == 0 表示当前迭代器引用的数据。

```
int _basic_string_iterator_minus(
    basic_string_iterator_t it_first, basic_string_iterator_t it_second);
```

描述:

求两个迭代器之间的差。

参数:

it_first 第一个 basic_string 迭代器。
it_second 第二个 basic_string 迭代器。

返回值:

返回两个迭代器之间相差的距离。

注意:

`it_first` 和 `it_second` 必须是属于同一个 `basic_string` 的有效迭代器，否则函数的行为是未定义的。当 `it_first < it_second` 的时候返回值小于 0，`it_first == it_second` 时返回值等于 0，`it_first > it_second` 时返回值大于 0。

第六节 内部和辅助接口

`basic_string` 内部接口是为了实现 `basic_string` 容器以及外部接口提供的，用户不应该直接使用这些接口。

<code>_create_basic_string</code>	创建一个 <code>basic_string</code> 容器。
<code>_create_basic_string_auxiliary</code>	创建 <code>basic_string</code> 容器的辅助函数。
<code>_basic_string_init_elem</code>	使用用户指定的数据初始化 <code>basic_string</code> 容器。
<code>_basic_string_init_elem_varg</code>	使用可变参数列表中的数据初始化 <code>basic_string</code> 容器。
<code>_basic_string_destroy_auxiliary</code>	销毁 <code>basic_string</code> 容器的辅助函数。
<code>_basic_string_find_elem</code>	在 <code>basic_string</code> 容器中查找指定的数据。
<code>_basic_string_find_elem_varg</code>	在 <code>basic_string</code> 容器中查找可变参数列表中指定的数据。
<code>_basic_string_rfind_elem</code>	在 <code>basic_string</code> 容器中反向查找指定的数据。
<code>_basic_string_rfind_elem_varg</code>	在 <code>basic_string</code> 容器中反向查找可变参数列表中指定的数据。
<code>_basic_string_find_first_not_of_elem</code>	在 <code>basic_string</code> 容器中查找第一个非指定的数据。
<code>_basic_string_find_first_not_of_elem_varg</code>	在 <code>basic_string</code> 容器中查找第一个非可变参数列表指定的数据。
<code>_basic_string_find_last_not_of_elem</code>	在 <code>basic_string</code> 容器中查找最后一个非指定的数据。
<code>_basic_string_find_last_not_of_elem_varg</code>	在 <code>basic_string</code> 容器中查找最后一个非可变参数列表指定的数据。
<code>_basic_string_connect_elem</code>	将 <code>basic_string</code> 容器与指定的数据连接。
<code>_basic_string_connect_elem_varg</code>	将 <code>basic_string</code> 容器与可变参数列表指定的数据链接。
<code>_basic_string_assign_elem</code>	为 <code>basic_string</code> 容器赋值。
<code>_basic_string_assign_elem_varg</code>	为 <code>basic_string</code> 容器赋值，数据来自于可变参数列表。
<code>_basic_string_push_back</code>	向 <code>basic_string</code> 容器末尾添加一个数据。
<code>_basic_string_push_back_varg</code>	向 <code>basic_string</code> 容器末尾添加一个数据，数据来自于可变参数列表。
<code>_basic_string_pop_back</code>	删除 <code>basic_string</code> 容器的最后一个数据。
<code>_basic_string_resize</code>	重新设置 <code>basic_string</code> 中数据的个数，使用指定的数据填充。
<code>_basic_string_resize_varg</code>	重新设置 <code>basic_string</code> 中数据的个数，使用可变参数列表中的数据填充。
<code>_basic_string_append_elem</code>	向 <code>basic_string</code> 容器末尾添加指定的数据。
<code>_basic_string_append_elem_varg</code>	向 <code>basic_string</code> 容器末尾添加可变参数列表指定的数据。
<code>_basic_string_insert</code>	向 <code>basic_string</code> 容器插入指定数据。

<code>_basic_string_insert_n</code>	向 <code>basic_string</code> 容器的指定位置插入多个数据。
<code>_basic_string_insert_n_varg</code>	向 <code>basic_string</code> 容器的指定位置插入多个来自于可变参数的数据。
<code>_basic_string_insert_elem</code>	向 <code>basic_string</code> 容器插入指定数据。
<code>_basic_string_insert_elem_varg</code>	向 <code>basic_string</code> 容器的指定位置插入多个来自于可变参数的数据。
<code>_basic_string_range_replace_elem</code>	将 <code>basic_string</code> 容器中指定的数据区间使用指定数据替换。
<code>_basic_string_range_replace_elem_varg</code>	将 <code>basic_string</code> 容器中指定的数据区间使用可变参数列表指定的数据替换。
<code>_basic_string_replace_elem</code>	将 <code>basic_string</code> 容器中指定位置的数据使用指定数据替换。
<code>_basic_string_replace_elem_varg</code>	将 <code>basic_string</code> 容器中指定位置的数据使用可变参数列表指定的数据替换。
<code>_basic_string_init_elem_auxiliary</code>	初始化数据的辅助函数。

```
basic_string_t* _create_basic_string(const char* s_typename);
```

描述:

创建一个 `basic_string` 容器。

参数:

`s_typename` `basic_string` 容器中保存的数据类型。

返回值:

创建成功返回指向容器的指针, 否则返回 `NULL`。

注意:

如果 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型, `libcstl` 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
bool_t _create_basic_string_auxiliary(
    basic_string_t* pt_basic_string, const char* s_typename);
```

描述:

创建一个 `basic_string` 容器的辅助函数。

参数:

`pt_basic_string` 没有创建的 `basic_string` 容器。

`s_typename` `basic_string` 容器中保存的数据类型。

返回值:

创建成功返回 `true`, 否则返回 `false`。

注意:

如果 `pt_basic_string == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型, `libcstl` 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _basic_string_init_elem(basic_string_t* pt_basic_string, size_t t_count, ...);
```

描述:

使用用户指定的数据初始化 `basic_string` 容器。

参数:

`pt_basic_string` 未初始化的 `basic_string` 容器。

`t_count` 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 则函数的行为是未定义的。`basic_string` 容器必须是使用 `create_basic_string` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_init_elem_varg(
    basic_string_t* pt_basic_string, size_t t_count, va_list val_elemlist);
```

描述:

使用用户指定的数据初始化 `basic_string` 容器。

参数:

`pt_basic_string` 未初始化的 `basic_string` 容器。
`t_count` 数据个数。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 则函数的行为是未定义的。`basic_string` 容器必须是使用 `create_basic_string` 创建的否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_destroy_auxiliary(basic_string_t* pt_basic_string);
```

描述:

销毁 `basic_string` 容器的辅助函数。

参数:

`pt_basic_string` `basic_string` 容器。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 不是使用 `create_basic_string` 生成的则函数的行为是未定义的。

```
size_t _basic_string_find_elem(
    basic_string_t* pt_basic_string, size_t t_pos, ...);
```

描述:

在 `basic_string` 容器中查找指定的数据。

参数:

`cpt_basic_string` `basic_string` 容器。
`t_pos` 开始查找的位置。
... 指定的数据。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是未初始化的，则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，则返回 NPOS。

```
size_t _basic_string_find_elem_varg(
    basic_string_t* pt_basic_string, size_t t_pos, va_list val_elemlist);
```

描述:

在 basic_string 容器中查找可变参数列表指定的数据。

参数:

cpt_basic_string	basic_string 容器。
t_pos	开始查找的位置。
val_elemlist	可变参数列表。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是未初始化的，则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，则返回 NPOS。

```
size_t _basic_string_rfind_elem(
    basic_string_t* pt_basic_string, size_t t_pos, ...);
```

描述:

在 basic_string 容器中反向查找指定的数据。

参数:

cpt_basic_string	basic_string 容器。
t_pos	开始查找的位置。
...	指定的数据。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是未初始化的，则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，表示从 basic_string 末尾开始查找。

```
size_t _basic_string_rfind_elem_varg(
    basic_string_t* pt_basic_string, size_t t_pos, va_list val_elemlist);
```

描述:

在 basic_string 容器中反向查找可变参数列表指定的数据。

参数:

cpt_basic_string	basic_string 容器。
t_pos	开始查找的位置。
val_elemlist	可变参数列表。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是未初始化的，则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，表示从 basic_string 末尾开始查找。

```
size_t _basic_string_find_first_not_of_elem(
    basic_string_t* pt_basic_string, size_t t_pos, ...);
```

描述:

在 basic_string 容器中查找第一个非指定的数据。

参数:

cpt_basic_string	basic_string 容器。
t_pos	开始查找的位置。
...	指定的数据。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是未初始化的，则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，则返回 NPOS。

```
size_t _basic_string_find_first_not_of_elem_varg(
    basic_string_t* pt_basic_string, size_t t_pos, va_list val_elemlist);
```

描述:

在 basic_string 容器中查找第一个非可变参数列表指定的数据。

参数:

cpt_basic_string	basic_string 容器。
t_pos	开始查找的位置。
val_elemlist	可变参数列表。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是未初始化的，则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，则返回 NPOS。

```
size_t _basic_string_find_last_not_of_elem(
    basic_string_t* pt_basic_string, size_t t_pos, ...);
```

描述:

在 basic_string 容器中查找最后一个非指定的数据。

参数:

cpt_basic_string	basic_string 容器。
t_pos	开始查找的位置。
...	指定的数据。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是为初始化的则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，表示从 basic_string 末尾开始查找。

```
size_t _basic_string_find_last_not_of_elem_varg(
    basic_string_t* pt_basic_string, size_t t_pos, va_list val_elemlist);
```

描述:

在 basic_string 容器中查找最后一个非可变参数列表指定的数据。

参数:

cpt_basic_string	basic_string 容器。
t_pos	开始查找的位置。
val_elemlist	可变参数列表。

返回值:

如果查找成功返回字符串在 basic_string 中的位置，否则返回 NPOS。

注意:

如果 cpt_basic_string == NULL 或者 basic_string 是为初始化的则函数的行为是未定义的。如果 t_pos 不在 basic_string 范围内，表示从 basic_string 末尾开始查找。

```
void _basic_string_connect_elem(basic_string_t* pt_basic_string, ...);
```

描述:

将 basic_string 容器与指定的数据连接。

参数:

pt_basic_string	basic_string 容器。
...	指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是为初始化的则函数的行为是未定义的。

```
void _basic_string_connect_elem_varg(
    basic_string_t* pt_basic_string, va_list val_elemlist);
```

描述:

将 basic_string 容器与可变参数列表指定的数据链接。

参数:

pt_basic_string	basic_string 容器。
val_elemlist	可变参数列表。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是为初始化的则函数的行为是未定义的。

```
void _basic_string_assign_elem(
    basic_string_t* pt_basic_string, size_t t_count, ...);
```

描述:

为 basic_string 容器赋值。

参数:

pt_basic_string	basic_string 容器。
t_count	数据的个数。
...	指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的则函数的行为是未定义的。

```
void _basic_string_assign_elem_varg(
    basic_string_t* pt_basic_string, size_t t_count, va_list val_elemlist);
```

描述:

为 basic_string 容器赋值, 数据来自于可变参数列表。

参数:

pt_basic_string	basic_string 容器。
t_count	数据的个数。
val_elemlist	可变参数列表。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的则函数的行为是未定义的。

```
void _basic_string_push_back(basic_string_t* pt_basic_string, ...);
```

描述:

向 basic_string 容器末尾添加一个数据。

参数:

pt_basic_string	basic_string 容器。
...	指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的则函数的行为是未定义的。

```
void _basic_string_push_back_varg(
    basic_string_t* pt_basic_string, va_list val_elemlist);
```

描述:

向 basic_string 容器末尾添加一个数据, 数据来自于可变参数列表。

参数:

pt_basic_string basic_string 容器。
val_elemlist 可变参数列表。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的则函数的行为是未定义的。

```
void _basic_string_pop_back(basic_string_t* pt_basic_string);
```

描述:

删除 basic_string 容器的最后一个数据。

参数:

pt_basic_string basic_string 容器。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的则函数的行为是未定义的。 basic_string 为空则函数的行为是未定义的。

```
void _basic_string_resize(  
    basic_string_t* pt_basic_string, size_t t_resize, ...);
```

描述:

重新设置 basic_string 容器中的数据个数，不足的部分使用指定数据填充。

参数:

pt_basic_string basic_string 容器。
t_resize 数据个数。
... 用户指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的 basic_string 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。 t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用指定的数据填充。

```
void _basic_string_resize_varg(  
    basic_string_t* pt_basic_string, size_t t_resize, ...);
```

描述:

重新设置 basic_string 容器中的数据个数，不足的部分使用指定数据填充。

参数:

pt_basic_string basic_string 容器。
t_resize 数据个数。

`val_elemlist` 可变参数列表。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。`t_resize` 可以小于原来的数据数量，这是将末尾的数据删除，当 `t_resize` 大于原来数据的数量的时候，使用指定的数据填充。

```
basic_string_iterator_t _basic_string_insert(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_pos, ...);
```

描述:

向 `basic_string` 容器插入指定数据。

参数:

`pt_basic_string` `basic_string` 容器。
`it_pos` 插入数据的位置。
`...` 用户指定的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`it_pos` 必须是属于 `basic_string` 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
basic_string_iterator_t _basic_string_insert_n(
    basic_string_t* pt_basic_string, basic_string_iterator_t it_pos,
    size_t t_count, ...);
```

描述:

向 `basic_string` 容器的指定位置插入多个数据。

参数:

`pt_basic_string` `basic_string` 容器。
`it_pos` 插入数据的位置。
`t_count` 数据个数。
`...` 用户指定的数据。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`it_pos` 必须是属于 `basic_string` 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
basic_string_iterator_t _basic_string_insert_n_varg(
```

```
basic_string_t* pt_basic_string, basic_string_iterator_t it_pos,
size_t t_count, va_list val_elemlist);
```

描述:

向 basic_string 容器的指定位置插入多个来自于可变参数的数据。

参数:

pt_basic_string	basic_string 容器。
it_pos	插入数据的位置。
t_count	数据个数。
val_elemlist	用户指定的数据可变参数列表。

返回值:

返回引用被插入的第一个数据的迭代器。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的 basic_string 则函数的行为是未定义的。it_pos 必须是属于 basic_string 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_insert_elem(
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_count, ...);
```

描述:

向 basic_string 容器插入指定数据。

参数:

pt_basic_string	basic_string 容器。
t_pos	插入数据的位置。
t_count	数据个数。
...	用户指定的数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 basic_string 是未初始化的 basic_string 则函数的行为是未定义的。t_pos 必须是属于 basic_string 容器的有效位置，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_insert_elem_varg(
    basic_string_t* pt_basic_string, size_t t_pos,
    size_t t_count, va_list val_elemlist);
```

描述:

向 basic_string 容器的指定位置插入多个来自于可变参数的数据。

参数:

pt_basic_string	basic_string 容器。
t_pos	插入数据的位置。
t_count	数据个数。
val_elemlist	用户指定的数据可变参数列表。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`t_pos` 必须是属于 `basic_string` 容器的有效位置, 否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_range_replace_elem(
    basic_string_t* pt_basic_string,
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end,
    size_t t_count, ...);
```

描述:

将 `basic_string` 容器中指定的数据区间使用指定数据替换。

参数:

<code>pt_basic_string</code>	<code>basic_string</code> 容器。
<code>it_begin</code>	被替换的数据区间的开头。
<code>it_end</code>	被替换的数据区间的末尾。
<code>t_count</code>	数据个数。
...	用户指定的数据。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`[it_begin, it_end)` 必须是属于 `basic_string` 容器的有效数据区间, 否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表, 但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_range_replace_elem_varg(
    basic_string_t* pt_basic_string,
    basic_string_iterator_t it_begin, basic_string_iterator_t it_end,
    size_t t_count, va_list val_elemlist);
```

描述:

将 `basic_string` 容器中指定的数据区间使用指定数据替换。

参数:

<code>pt_basic_string</code>	<code>basic_string</code> 容器。
<code>it_begin</code>	被替换的数据区间的开头。
<code>it_end</code>	被替换的数据区间的末尾。
<code>t_count</code>	数据个数。
<code>val_elemlist</code>	用户指定的数据可变参数列表。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `basic_string` 是未初始化的 `basic_string` 则函数的行为是未定义的。`[it_begin,`

`it_end)`必须是属于`basic_string`容器的有效数据区间，否则函数的行为是未定义的。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_replace_elem(
    basic_string_t* pt_basic_string, size_t t_pos, size_t t_len, size_t t_count, ...);
```

描述:

将`basic_string`容器中指定位置的数据使用指定数据替换。

参数:

pt_basic_string	<code>basic_string</code> 容器。
t_pos	替换的开始位置。
t_len	替换的长度。
t_count	数据个数。
...	用户指定的数据。

返回值:

无。

注意:

如果`pt_basic_string == NULL`或者`basic_string`是未初始化的`basic_string`则函数的行为是未定义的。`t_pos`必须在`basic_string`有效范围内，否则函数的行为是未定义的，`t_len == NPOS`或者`t_pos + t_len >= basic_string_size()`，将`t_pos`开始到末尾的全部串替换。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_replace_elem_varg(
    basic_string_t* pt_basic_string,
    size_t t_pos, size_t t_len, size_t t_count, va_list val_elemlist);
```

描述:

将`basic_string`容器中指定的数据区间使用指定数据替换。

参数:

pt_basic_string	<code>basic_string</code> 容器。
t_pos	替换的开始位置。
t_len	替换的长度。
t_count	数据个数。
val_elemlist	用户指定的数据可变参数列表。

返回值:

无。

注意:

如果`pt_basic_string == NULL`或者`basic_string`是未初始化的`basic_string`则函数的行为是未定义的。`t_pos`必须在`basic_string`有效范围内，否则函数的行为是未定义的，`t_len == NPOS`或者`t_pos + t_len >= basic_string_size()`，将`t_pos`开始到末尾的全部串替换。用户指定的数据必须和容器中保存的数据类型是一致的否则函数的行为是未定义的。这个函数使用可变参数列表，但是只有第一个参数是有效的其他的参数无效。

```
void _basic_string_init_elem_auxiliary(
```

```
basic_string_t* pt_basic_string, void* pv_value);
```

描述:

使用 basic_string 的数据类型对数据进行初始化。

参数:

pt_basic_string basic_string 容器。
pv_value 数据。

返回值:

无。

注意:

如果 pt_basic_string == NULL 或者 pv_value == NULL 则函数的行为是未定义的。 basic_string 必须是已经初始化或者使用 create_basic_string 创建的 basic_string 容器, 否则函数的行为是未定义的。

basic_string 辅助接口是为了实现 basic_string 容器以及外部接口提供的, 用户不应该直接使用这些接口。

<u>_basic_string_same_type</u>	测试两个 basic_string 是否保存同样类型的数据。
<u>_basic_string_get_value_string_length</u>	获得数据串的长度。
<u>_basic_string_get_varg_value_auxiliary</u>	从可变参数列表中获得与 basic_string 中的数据类型相同的数据。
<u>_basic_string_destroy_varg_value_auxiliary</u>	销毁与 basic_string 中的数据类型相同的数据。

```
bool_t _basic_string_same_type(  
    const basic_string_t* cpt_first, const basic_string_t* cpt_second);
```

描述:

判断两个 basic_string 容器中保存的数据类型是否相同。

参数:

cpt_first 第一个 basic_string 容器。
cpt_second 第二个 basic_string 容器。

返回值:

如果两个 basic_string 中保存的数据类型相同返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL, 那么函数的行为是未定义的。两个 basic_string 容器必须是已经初始化或者使用 create_basic_string 创建的 basic_string 容器, 否则函数的行为是未定义的。如果 cpt_first == cpt_second 则返回 true。

```
size_t _basic_string_get_value_string_length(  
    const basic_string_t* cpt_basic_string, const void* cpv_value_string);
```

描述:

获得数据串的长度。

参数:

cpt_basic_string basic_string 容器。
cpv_value_string 数据串。

返回值:

返回数据串的长度。

注意:

如果 `cpt_basic_string == NULL` 或者 `cpv_value_string == NULL`, 那么函数的行为是未定义的。两个 `basic_string` 容器必须是已经初始化或者使用 `create_basic_string` 创建的 `basic_string` 容器, 否则函数的行为是未定义的。数据串是以内存 0 表示结束, 否则函数的行为是未定的。

```
void _basic_string_get_varg_value_auxiliary(
    basic_string_t* pt_basic_string, va_list val_elemlist, void* pv_varg);
```

描述:

从可变参数列表中获得与 `basic_string` 中的数据类型相同的数据。

参数:

<code>pt_basic_string</code>	<code>basic_string</code> 容器。
<code>val_elemlist</code>	可变参数列表。
<code>pv_varg</code>	保存数据缓的冲区。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `pv_varg == NULL`, 那么函数的行为是未定义的。`pt_basic_string` 必须是已经初始化或者是由 `create_basic_string()` 创建的 `basic_string` 容器, 否则函数的行为是未定义的。

```
void _basic_string_destroy_varg_value_auxiliary(
    basic_string_t* pt_basic_string, void* pv_varg);
```

描述:

销毁与 `basic_string` 中的数据类型相同的数据。

参数:

<code>pt_basic_string</code>	<code>basic_string</code> 容器。
<code>pv_varg</code>	保存数据的缓冲区。

返回值:

无。

注意:

如果 `pt_basic_string == NULL` 或者 `pv_varg == NULL`, 那么函数的行为是未定义的。`pt_basic_string` 必须是已经初始化或者是由 `create_basic_string()` 创建的 `basic_string` 容器, 否则函数的行为是未定义的。

第十三章 string 容器

string 容器是字符串的概念，它的底层使用 basic_string 实现，string 与其他容器不同，它内部保存的只是 char 类型而不保存其他类型的数据。

第一节 string 容器的机制

string 容器的机制与 basic_string 很相似，但是还是有不同的地方，string 内部要保存以\0 结尾的字符串。例如要将字符串”abc”保存到 string 中，实际在底层的 basic_string 中保存的是 abc\0，这样在对 string 进行操作的函数的内部实现中始终要注意\0 的存在。

在使用字符串时有些问题，当字符串中包含'\0'的时候会被截断，如” abc\0xxxxx”会被自动截断成” abc”，这与 STL 的行为不一致。

同时 string 还提供了一些 I/O 操作。

第二节 string 迭代器

直接使用 basic_string 的迭代器作为 string 的迭代器。

第三节 string 代码结构

string 直接使用 basic_string 作为底层实现

```
typedef basic_string_t string_t;
```

第四节 外部接口

string 外部接口：

create_string	创建 string 容器。
string_init	初始化一个空的 string 容器。
string_init_char	使用指定的字符初始化 string 容器。
string_init_cstr	使用指定的字符串初始化 string 容器。

string_init_subcstr	使用指定的子字符串初始化 string 容器。
string_init_copy	使用指定的 string 来初始化 string 容器。
string_init_copy_substring	使用指定的子 string 来初始化 string 容器。
string_init_copy_range	使用指定的 string 范围初始化 string 容器。
string_destroy	销毁一个 string 容器。
string_c_str	返回指向 string 容器中保存字符的指针。
string_data	返回指向 string 容器中保存字符的指针。
string_copy	将 string 容器中保存的字符拷贝到指定的缓冲区中。
string_size	返回 string 容器中保存的字符的个数。
string_length	返回 string 容器中保存的字符的长度。
string_empty	测试 string 容器是否为空。
string_max_size	返回 string 容器能够保存的字符的数量。
string_capacity	返回 string 容器的容量。
string_at	通过下标访问 string 容器中的字符。
string_equal	测试两个 string 容器是否相等。
string_not_equal	测试两个 string 容器是否不等。
string_less	测试第一个 string 容器是否小于第二个 string 容器。
string_less_equal	测试第一个 string 容器是否小于等于第二个 string 容器。
string_greater	测试第一个 string 容器是否大于第二个 string 容器。
string_greater_equal	测试第一个 string 容器是否大于等于第二个 string 容器。
string_equal_cstr	测试 string 容器和字符串是否相等。
string_not_equal_cstr	测试 string 容器和字符串是否不等。
string_less_cstr	测试 string 容器是否小于字符串。
string_less_equal_cstr	测试 string 容器是否小于等于字符串。
string_greater_cstr	测试 string 容器是否大于字符串。
string_greater_equal_cstr	测试 string 容器是否大于等于字符串。
string_compare	返回两个 string 容器比较的结果。
string_compare_substring_string	返回子 string 容器与另一个 string 容器比较的结果。
string_compare_substring_substring	返回两个子 string 容器的比较结果。
string_compare_cstr	返回 string 容器与字符串的比较结果。
string_compare_substring_cstr	返回子 string 容器与字符串的比较结果。
string_compare_substring_subcstr	返回子 string 容器与子字符串的比较结果。

string_substr	返回指定的子 string 容器。
string_connect	将两个 string 容器链接。
string_connect_char	将 string 容器与指定的字符连接。
string_connect_cstr	将 string 容器与字符串连接。
string_find	在 string 容器中查找指定的 string。
string_find_cstr	在 string 容器中查找指定的字符串。
string_find_substr	在 string 容器中查找指定的子字符串。
string_find_char	在 string 容器中查找指定的字符。
string_rfind	在 string 容器中反向查找指定的 string。
string_rfind_cstr	在 string 容器中反向查找字符串。
string_rfind_subcstr	在 string 容器中反向查找子字符串。
string_rfind_char	在 string 容器中反向查找指定的字符。
string_find_first_of	在 string 容器中查找第一个在指定的 string 容器出现的字符。
string_find_first_of_cstr	在 string 容器中查找第一个在字符串中出现的字符。
string_find_first_of_subestr	在 string 容器中查找第一个在子字符串中出现的字符。
string_find_first_of_char	在 string 容器中查找第一个出现的指定字符。
string_find_first_not_of	在 string 容器中查找第一个不出现在指定 string 容器中的字符。
string_find_first_not_of_cstr	在 string 容器中查找第一个不出现在字符串中的字符。
string_find_first_not_of_subcstr	在 string 容器中查找第一个不出现在子字符串中的字符。
string_find_first_not_of_char	在 string 容器中查找第一个不是指定字符的字符。
string_find_last_of	在 string 容器中查找最后一个出现在指定 string 容器中的字符。
string_find_last_of_cstr	在 string 容器中查找最后一个出现在字符串中的字符。
string_find_last_of_subcstr	在 string 容器中查找最后一个出现在子字符串中的字符。
string_find_last_of_char	在 string 容器中查找最后一个出现的指定字符。
string_find_last_not_of	在 string 容器中查找最后一个不出现在指定 string 容器中的字符。
string_find_last_not_of_cstr	在 string 容器中查找最后一个不出现在字符串中的字符。
string_find_last_not_of_subcstr	在 string 容器中查找最后一个不出现在子字符串中的字符。
string_find_last_not_of_char	在 string 容器中查找最后一个不是指定字符的字符。
string_begin	返回指向 string 容器开头的迭代器。
string_end	返回指向 string 容器末尾的迭代器。
string_clear	清空 string 容器。
string_swap	交换两个 string 容器的内容。

string_reserve	重新设置 string 容器的容量。
string_assign	使用指定的 string 容器为 string 容器赋值。
string_assign_substring	使用指定的子 string 为 string 容器赋值。
string_assign_cstr	使用指定的字符串为 string 容器赋值。
string_assign_subcstr	使用指定的子字符串为 string 容器赋值。
string_assign_range	使用指定的字符区间为 string 容器赋值。
string_assign_char	使用指定的字符为 string 容器赋值。
string_append	向 string 容器末尾添加指定的 string 容器。
string_append_substring	向 string 容器末尾添加指定的子 string。
string_append_cstr	向 string 容器末尾添加指定的字符串。
string_append_subcstr	向 string 容器末尾添加指定的子字符串。
string_append_range	向 string 容器末尾添加指定的字符区间。
string_append_char	向 string 容器末尾添加指定的字符。
string_insert	向 string 容器中插入指定字符。
string_insert_n	向 string 容器中插入多个指定字符。
string_insert_string	向 string 容器中插入指定的 string 容器。
string_insert_substring	向 string 容器中插入指定的子 string。
string_insert_cstr	向 string 容器中插入指定的字符串。
string_insert_subcstr	向 string 容器中插入指定的子字符串。
string_insert_range	向 string 容器中插入指定的字符区间。
string_insert_char	向 string 容器中插入指定字符。
string_push_back	向 string 容器末尾添加指定字符。
string_resize	重新设置 string 容器中字符的个数。
string_erase	删除 string 容器中指定位置的字符。
string_erase_range	删除 string 容器中指定的字符区间。
string_erase_substring	删除 string 容器中的子字符串。
string_replace	将 string 容器中的子字符串替换成指定的 string。
string_replace_substring	将 string 容器中的子字符串替换成指定的子 string。
string_replace_cstr	将 string 容器中的子字符串替换成指定的字符串。
string_replace_subcstr	将 string 容器中的子字符串替换成指定的子字符串。
string_replace_char	将 string 容器中的子字符串替换成指定的字符。
string_range_replace	将 string 容器中的字符区间替换成 string。

string_range_replace_substring	将 string 容器中的字符区间替换成子 string。
string_range_replace_cstr	将 string 容器中的字符区间替换成字符串。
string_range_replace_substr	将 string 容器中的字符区间替换成子字符区间。
string_range_replace_char	将 string 容器中的字符区间替换成指定的字符。
string_replace_range	将 string 容器中的字符区间替换成指定的字符区间。
string_output	将 string 容器中的字符串输出到指定的流中。
string_input	从指定的流中读取字符串到 string 容器。
string_getline	从指定的流中读取一行并保存到 string 容器中。
string_getline_delimiter	从指定的流中读取一行并保存到 string 容器中，用户指定行标识符。

接口原型:

```
string_t* create_string(void);
```

描述:

创建一个保存指定类型的 string 容器。

参数:

无。

返回值:

指向 string_t 容器的指针。创建失败返回 NULL。

注意:

创建的 string 容器中保存的数据是 char 类型。

```
void string_init(string_t* pstr_string);
```

描述:

初始化一个空的 string 容器。

参数:

pstr_string 指向被初始化的 string 容器的指针。

返回值:

无。

注意:

被初始化的容器一定是使用 create_string 创建的容器，否则函数的行为是未定义的。pstr_string == NULL，函数的行为是未定义的。使用 string_init 初始化之后的 string 容器，string_size() == 0。

```
void string_init_char(string_t* pstr_string, size_t t_count, c_char);
```

描述:

使用用户指定的字符初始化 string 容器。

参数:

pstr_string 指向被初始化的 string 容器的指针。

t_count 初始化后包含的字符的个数。

c_char 指定的字符。

返回值:

无。

注意:

如果 `pstr_string == NULL` 则函数的行为是未定义的。 `string` 容器必须是使用 `create_string` 创建的否则函数的行为是未定义的。 使用 `string_init_char` 初始化之后的 `string` 容器， `string_size() == t_count`。

```
void string_init_cstr(string_t* pstr_string, const char* s_cstr);
```

描述:

使用指定的字符串初始化 `string` 容器。

参数:

`pstr_string` 指向被初始化的 `string` 容器的指针。

`s_cstr` 指定的字符串。

返回值:

无。

注意:

如果 `pstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的。 `string` 容器必须是使用 `create_string` 创建的否则函数的行为是未定义的。 字符串使用内存\0 表示结束， 初始化之后 `string_size()` 等于实际字符串的长度。

```
void string_init_subcstr(string_t* pstr_string, const char* s_cstr, size_t t_len);
```

描述:

使用指定的子字符串初始化 `string` 容器。

参数:

`pstr_string` 指向被初始化的 `string` 容器的指针。

`s_cstr` 指定的字符串。

`t_len` 子串的长度。

返回值:

无。

注意:

如果 `pstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的。 `string` 容器必须是使用 `create_string` 创建的否则函数的行为是未定义的。 如果 `t_len` 等于 NPOS 或者超过了字符串的长度则使用整个字符串进行初始化。

```
void string_init_copy(string_t* pstr_dest, const string_t* cpstr_src);
```

描述:

使用一个既存的 `string` 容器来初始化 `string` 容器。

参数:

`pstr_dest` 指向被初始化的容器的指针。

`cpstr_src` 指向既存容器的指针。

返回值:

无。

注意:

被初始化的容器一定是使用 `create_string` 创建的容器， 否则函数的行为是未定义的。 `cpstr_src` 指向的是已经初始化的 `string` 容器， 否则函数的行为是未定义的。 `pstr_dest == NULL` 或者 `cpstr_src == NULL`， 函数的行为是未定义的。 初始

化之后 `string_size(pstr_dest) == string_size(cpstr_src);`。

```
void string_init_copy_substring(
    string_t* pstr_dest, const string_t* cpstr_src, size_t t_pos, size_t t_len);
```

描述:

使用指定的子 `string` 来初始化 `string` 容器。

参数:

`pstr_dest` 指向被初始化的容器的指针。
`cpstr_src` 指向既存容器的指针。
`t_pos` 子字符串开始的位置。
`t_len` 子字符串长度。

返回值:

无。

注意:

被初始化的容器一定是使用 `create_string` 创建的容器，否则函数的行为是未定义的。`cpstr_src` 指向的是已经初始化的 `string` 容器，否则函数的行为是未定义的。`pstr_dest == NULL` 或者 `cpstr_src == NULL`，函数的行为是未定义的。`t_pos` 必须是属于源字符串的有效位置，否则函数的行为是未定义的，如果 `t_len == NPOS` 或者 `t_pos + t_len >=` 元字符串的长度，那么使用从 `t_pos` 开始的所有子字符串。

```
void string_init_copy_range(string_t* pstr_string,
    string_iterator_t it_begin, string_iterator_t it_end);
```

描述:

使用指定的 `string` 范围初始化 `string` 容器。

参数:

`pstr_string` 指向被初始化的容器的指针。
`it_begin` 指定的数据区间的开始。
`it_end` 制定的数据区间的末尾。

返回值:

无。

注意:

被初始化的容器一定是使用 `create_string` 创建的容器，否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 `string` 容器的有效数据区间，否则函数的行为是未定义的。`pstr_string == NULL` 函数的行为是未定义的。初始化之后 `string_size(pstr_string) == iterator_distance(it_begin, it_end);`。

```
void string_destroy(string_t* pstr_string);
```

描述:

销毁创建的 `string` 容器。

参数:

`pstr_string` 指向被销毁的容器的指针。

返回值:

无。

注意:

被销毁的容器一定已经初始化或者是使用 `create_string` 创建的容器，否则函数的行为是未定义的。销毁后的容器不能够在用于其他的接口，否则接口的函数行为是未定义的。创建之后没有经过初始化的容器也要进行销毁。

```
const char* string_c_str(const string_t* cpstr_string);
```

描述:

返回指向 `string` 容器中保存字符的指针。

参数:

`cpstr_string` `string` 容器。

返回值:

返回指向 `string` 中保存的字符串的指针。

注意:

如果 `cpstr_string == NULL` 或者 `string` 容器一定已经初始化的容器，否则函数的行为是未定义的。当 `string` 为空时返回空串。

```
const char* string_data(const string_t* cpstr_string);
```

描述:

返回指向 `string` 容器中保存字符的指针。

参数:

`cpstr_string` `string` 容器。

返回值:

返回指向 `string` 中保存的字符串的指针。

注意:

如果 `cpstr_string == NULL` 或者 `string` 容器一定已经初始化的容器，否则函数的行为是未定义的。当 `string` 为空时函数的行为是未定义的。

```
size_t string_copy(  
    const string_t* cpstr_string, char* s_buffer, size_t t_len, size_t t_pos);
```

描述:

将 `string` 容器中保存的字符拷贝到指定的缓冲区中。

参数:

`cpstr_string` `string` 容器。

`s_buffer` 目的缓冲区。

`t_len` 拷贝的字符串的长度。

`t_pos` 子串的位置。

返回值:

实际拷贝的子串的长度。

注意:

如果 `cpstr_string == NULL` 或者 `string` 容器一定已经初始化的容器，否则函数的行为是未定义的。如果 `s_buffer == NULL` 那么函数的行为是未定义的。`t_pos` 是属于 `string` 的有效的位置，否则函数的行为是未定义的。实际拷贝的长度是 $\min(t_len, \text{string_size}() - t_pos)$ 。

```
size_t string_size(const string_t* cpstr_string);
```

描述:

返回 string 容器中保存的字符的个数。

参数:

cpstr_string string 容器。

返回值:

string 容器中包含的字符的个数。

注意:

如果 cpstr_string == NULL 或者 string 容器一定已经初始化的容器，否则函数的行为是未定义的。

```
size_t string_length(const string_t* cpstr_string);
```

描述:

返回 string 容器中保存的字符的长度。

参数:

cpstr_string string 容器。

返回值:

string 容器中包含的字符串的长度。

注意:

如果 cpstr_string == NULL 或者 string 容器一定已经初始化的容器，否则函数的行为是未定义的。这个函数与 string_size()相同。

```
bool_t string_empty(const string_t* cpstr_string);
```

描述:

测试 string 容器是否为空。

参数:

cpstr_string string 容器。

返回值:

如果 string 为空则返回 true，否则返回 false。

注意:

如果 cpstr_string == NULL 或者 string 容器一定已经初始化的容器，否则函数的行为是未定义的。

```
size_t string_max_size(const string_t* cpstr_string);
```

描述:

返回 string 容器能够保存的字符的数量。

参数:

cpstr_string string 容器。

返回值:

返回 string 容器中能够保存字符的最大数量的可能值。

注意:

cpstr_string == NULL 则函数的行为是未定义的，cpstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。这个最大数量是与系统相关的，不是一个固定的值。

```
size_t string_capacity(const string_t* cpstr_string);
```

描述:

返回 string 容器的容量。

参数:

cpstr_string string 容器。

返回值:

返回 string 容器的容量。

注意:

cpstr_string == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。容量符合前面描述的算法。

```
char* string_at(const string_t* cpstr_string, size_t t_pos);
```

描述:

通过下标随机访问 string 容器中的字符。

参数:

cpstr_string 指向 string 容器的指针。
t_pos 要访问的字符在容器中的索引。

返回值:

指向被访问的字符的指针。

注意:

cpstr_string == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果 t_pos 超过了 string 容器中字符索引的范围, 则函数的行为是未定义的。

```
bool_t string_equal(const string_t* cpstr_first, const string_t* cpstr_second);
```

描述:

测试两个容器是否相等。

参数:

cpstr_first 指向 string 容器的指针。
cpstr_second 指向 string 容器的指针。

返回值:

如果两个容器相等, 返回 true, 否则返回 false。

注意:

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的, cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果两个容器中的字符个数相等并且对应字符相等则容器相等, 如果 cpstr_first == cpstr_second 则返回 true。

```
bool_t string_not_equal(const string_t* cpstr_first, const string_t* cpstr_second);
```

描述:

测试两个容器是否不相等。

参数:

cpstr_first 指向 string 容器的指针。
cpstr_second 指向 string 容器的指针。

返回值:

如果两个容器不相等，返回 true，否则返回 false。

注意：

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的，cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果两个容器中的字符个数不等或者对应字符不等则容器不等，如果 cpstr_first == cpstr_second 则返回 false。

```
bool_t string_less(const string_t* cpstr_first, const string_t* cpstr_second);
```

描述：

测试第一个 string 容器是否小于第二个 string 容器。

参数：

cpstr_first	指向 string 容器的指针。
cpstr_second	指向 string 容器的指针。

返回值：

如果第一个 string 容器小于第二个 string 容器，返回 true，否则返回 false。

注意：

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的，cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果第一个容器中的字符小于第二个容器中的对应字符则返回 true，如果大于则返回 false，如果对应字符都相等则，如果第一个容器中的字符个数小于第二个容器中的字符个数则返回 true 否则返回 false，如果 cpstr_first == cpstr_second 则返回 false。

```
bool_t string_less_equal(const string_t* cpstr_first, const string_t* cpstr_second);
```

描述：

测试第一个 string 容器是否小于等于第二个 string 容器。

参数：

cpstr_first	指向 string 容器的指针。
cpstr_second	指向 string 容器的指针。

返回值：

如果第一个 string 容器小于等于第二个 string 容器，返回 true，否则返回 false。

注意：

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的，cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果第一个容器中的字符小于第二个容器中的对应字符则返回 true，如果大于则返回 false，如果对应字符都相等则，如果第一个容器中的字符个数小于等于第二个容器中的字符个数则返回 true 否则返回 false，如果 cpstr_first == cpstr_second 则返回 true。

```
bool_t string_greater(const string_t* cpstr_first, const string_t* cpstr_second);
```

描述：

测试第一个 string 容器是否大于第二个 string 容器。

参数：

cpstr_first	指向 string 容器的指针。
cpstr_second	指向 string 容器的指针。

返回值：

如果第一个 string 容器大于第二个 string 容器，返回 true，否则返回 false。

注意:

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的, cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果第一个容器中的字符大于第二个容器中的对应字符则返回 true, 如果小于则返回 false, 如果对应字符都相等则, 如果第一个容器中的字符个数大于第二个容器中的字符个数则返回 true 否则返回 false, 如果 cpstr_first == cpstr_second 则返回 false。

```
bool_t string_greater_equal(  
    const string_t* cpstr_first, const string_t* cpstr_second);
```

描述:

测试第一个 string 容器是否大于等于第二个 string 容器。

参数:

cpstr_first	指向 string 容器的指针。
cpstr_second	指向 string 容器的指针。

返回值:

如果第一个 string 容器大于等于第二个 string 容器, 返回 true, 否则返回 false。

注意:

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的, cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果第一个容器中的字符大于第二个容器中的对应字符则返回 true, 如果小于则返回 false, 如果对应字符都相等则, 如果第一个容器中的字符个数大于等于第二个容器中的字符个数则返回 true 否则返回 false, 如果 cpstr_first == cpstr_second 则返回 true。

```
bool_t string_equal_cstr(const string_t* cpstr_string, const char* s_cstr);
```

描述:

测试 string 容器和字符串是否相等。

参数:

cpstr_string	string 容器。
s_cstr	字符串。

返回值:

如果 string 中的字符与 s_cstr 中的字符相等, 返回 true, 否则返回 false。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果 string 容器中的字符与字符串中的对应字符都相等返回 true, 否则返回 false。

```
bool_t string_not_equal_cstr(const string_t* cpstr_string, const char* s_cstr);
```

描述:

测试 string 容器和字符串是否不等。

参数:

cpstr_string	string 容器。
s_cstr	字符串。

返回值:

如果 string 中的字符与 s_cstr 中的字符不等, 返回 true, 否则返回 false。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。

```
bool_t string_less_cstr(const string_t* cpstr_string, const char* s_cstr);
```

描述:

测试 string 容器是否小于字符串。

参数:

cpstr_string string 容器。
s_cstr 字符串。

返回值:

如果 string 容器中的字符小于 s_cstr, 返回 true, 否则返回 false。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果 string 器中的字符小于字符串中的对应字符则返回 true, 如果大于则返回 false, 如果对应字符都相等则, 如果 string 容器中的字符个数小于字符串中的字符个数则返回 true 否则返回 false。

```
bool_t string_less_equal_cstr(const string_t* cpstr_string, const char* s_cstr);
```

描述:

测试 string 容器是否小于等于字符串。

参数:

cpstr_string string 容器。
s_cstr 字符串。

返回值:

如果 string 容器中的字符小于等于 s_cstr, 返回 true, 否则返回 false。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果 string 器中的字符小于字符串中的对应字符则返回 true, 如果大于则返回 false, 如果对应字符都相等则, 如果 string 容器中的字符个数小于等于字符串中的字符个数则返回 true 否则返回 false。

```
bool_t string_greater_cstr(const string_t* cpstr_string, const char* s_cstr);
```

描述:

测试 string 容器是否大于字符串。

参数:

cpstr_string string 容器。
s_cstr 字符串。

返回值:

如果 string 容器中的字符大于 s_cstr, 返回 true, 否则返回 false。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果 string 器中的字符大于字符串中的对应字符则返回 true, 如果小于则返

回 false，如果对应字符都相等则，如果 string 容器中的字符个数大于字符串中的字符个数则返回 true 否则返回 false。

```
bool_t string_greater_equal_cstr(const string_t* cpstr_string, const char* s_cstr);
```

描述:

测试 string 容器是否大于等于字符串。

参数:

cpstr_string string 容器。
s_cstr 字符串。

返回值:

如果 string 容器中的字符大于等于 s_cstr，返回 true，否则返回 false。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果 string 器中的字符大于字符串中的对应字符则返回 true，如果小于则返回 false，如果对应字符都相等则，如果 string 容器中的字符个数大于等于字符串中的字符个数则返回 true 否则返回 false。

```
int string_compare(const string_t* cpstr_first, const string_t* cpstr_second);
```

描述:

返回两个 string 容器比较的结果。

参数:

cpstr_first 第一个 string 容器。
cpstr_second 第二个 string 容器。

返回值:

如果第一个 string 容器中的字符大于第二个 string 容器中的字符，返回值大于 0，小于，返回值小于 0，相等，返回值等于 0。

注意:

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的，cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果 cpstr_first == cpstr_second 则返回 0。

```
int string_compare_substring_string(  
    const string_t* cpstr_first, size_t t_pos, size_t t_len,  
    const string_t* cpstr_second);
```

描述:

返回子 string 容器与另一个 string 容器比较的结果。

参数:

cpstr_first 第一个 string 容器。
t_pos 子 string 的开始位置。
t_len 子 string 的长度。
cpstr_second 第二个 string 容器。

返回值:

如果第一个 string 容器的子串中的字符大于第二个 string 容器中的字符，返回值大于 0，小于，返回值小于 0，相等，返回值等于 0。

注意:

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的, cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。t_pos 是属于第一个 string 容器的有效的下标, 否则函数的行为是未定义的, t_len >= string_size() - t_pos 时表示使用从 t_pos 开始的全部子串。

```
int string_compare_substring_substring(
    const string_t* cpstr_first, size_t t_firstpos, size_t t_firstlen,
    const string_t* cpstr_second, size_t t_secondpos, size_t t_secondlen);
```

描述:

返回子 string 容器与另一个 string 容器比较的结果。

参数:

cpstr_first	第一个 string 容器。
t_firstpos	子 string 的开始位置。
t_firstlen	子 string 的长度。
cpstr_second	第二个 string 容器。
t_secondpos	子 string 的开始位置。
t_secondlen	子 string 的长度。

返回值:

如果第一个 string 容器的子串中的字符大于第二个 string 容器的子串中的字符, 返回值大于 0, 小于, 返回值小于 0, 相等, 返回值等于 0。

注意:

cpstr_first == NULL 或者 cpstr_second == NULL 则函数的行为是未定义的, cpstr_first 和 cpstr_second 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。t_firstpos 是属于第一个 string 容器的有效的下标, t_secondpos 是属于第二个 string 容器的有效下标, 否则函数的行为是未定义的, t_firstlen >= string_size() - t_firstpos 时表示使用从 t_firstpos 开始的全部子串, t_secondlen >= string_size() - t_secondpos 时表示使用从 t_secondpos 开始的全部子串。

```
int string_compare_cstr(const string_t* cpstr_string, const char* s_cstr);
```

描述:

返回 string 容器与字符串的比较结果。

参数:

cpstr_string	string 容器。
s_cstr	字符串。

返回值:

如果 string 容器中的字符大于字符串中的字符, 返回值大于 0, 小于, 返回值小于 0, 相等, 返回值等于 0。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的, cpstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。

```
int string_compare_substring_cstr(
    const string_t* cpstr_string, size_t t_pos, size_t t_len, const char* s_cstr);
```

描述:

返回子 string 容器与字符串的比较结果。

参数:

cpstr_string	string 容器。
t_pos	子 string 的开始位置。
t_len	子 string 的长度。
s_cstr	字符串。

返回值:

如果 string 容器子串中的字符大于字符串中的字符，返回值大于 0，小于，返回值小于 0，相等，返回值等于 0。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。t_pos 是属于第一个 string 容器的有效的下标，否则函数的行为是未定义的， $t_{len} \geq string_size() - t_{pos}$ 时表示使用从 t_pos 开始的全部子串。

```
int string_compare_substring_subcstr(
    const string_t* cpstr_string, size_t t_pos, size_t t_len,
    const char* s_cstr, size_t t_valuelen);
```

描述:

返回子 string 容器与子字符串的比较结果。

参数:

cpstr_string	string 容器。
t_pos	子 string 的开始位置。
t_len	子 string 的长度。
s_cstr	字符串。
t_valuelen	子字符串长度。

返回值:

如果 string 容器子串中的字符大于子字符串中的字符，返回值大于 0，小于，返回值小于 0，相等，返回值等于 0。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。t_pos 是属于第一个 string 容器的有效的下标，否则函数的行为是未定义的， $t_{len} \geq string_size() - t_{pos}$ 时表示使用从 t_pos 开始的全部子串。t_valuelen 大于等于字符串的长度则使用全部字符串。

```
string_t* string_substr(const string_t* cpt_string, size_t t_pos, size_t t_len);
```

描述:

返回指定的子 string 容器。

参数:

cpt_string	string 容器。
t_pos	子 string 的开始位置。
t_len	子 string 的长度。

返回值:

返回指定的子 string 容器。

注意:

cpt_string == NULL 则函数的行为是未定义的，cpt_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。t_pos 是属于 string 容器的有效的下标，否则函数的行为是未定义的， $t_{len} \geq string_size() - t_{pos}$

时表示使用从 `t_pos` 开始的全部子串。返回的子串是已经初始化的 `string`，用户在使用之后负责使用 `string_destroy()` 释放资源。

```
void string_connect(string_t* pstr_dest, const string_t* cpstr_src);
```

描述:

将两个 `string` 容器链接。

参数:

`pstr_dest` 目的 `string` 容器。
`cpstr_src` 源 `string` 容器。

返回值:

无。

注意:

`pstr_dest == NULL` 或者 `cpstr_src == NULL` 则函数的行为是未定义的，`pstr_dest` 和 `cpstr_src` 必须是已经初始化的，否则函数的行为是未定义的。

```
void string_connect_char(string_t* pstr_string, char c_char);
```

描述:

将 `string` 容器与指定的字符连接。

参数:

`pstr_string` `string` 容器。
`c_char` 指定的字符。

返回值:

无。

注意:

如果 `pstr_string == NULL` 或者 `string` 是未初始化的则函数的行为是未定义的。

```
void string_connect_cstr(string_t* pstr_string, const void* s_cstr);
```

描述:

将 `string` 容器与指定的字符串连接。

参数:

`pstr_string` 目的 `string` 容器。
`s_cstr` 源字符串。

返回值:

无。

注意:

`pstr_string == NULL` 或者 `cpstr_value_string == NULL` 则函数的行为是未定义的，`pstr_string` 必须是已经初始化的，否则函数的行为是未定义的。

```
size_t string_find(
    const string_t* cpstr_string, const string_t* cpstr_find, size_t t_pos);
```

描述:

在 `string` 容器中查找指定的 `string`。

参数:

cpstr_string	string 容器。
cpstr_find	要查找的 string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 cpstr_find == NULL 则函数的行为是未定义的，cpstr_string 和 cpstr_find 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 cpstr_string 的有效位置，否则函数的行为是未定义的。如果 cpstr_find 为空，则返回索引值。

```
size_t string_find_cstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos);
```

描述:

在 string 容器中查找指定的字符串。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 cpstr_string 的有效位置，否则函数的行为是未定义的。如果 s_cstr 为空，则返回索引值。

```
size_t string_find_subcstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos, size_t t_len);
```

描述:

在 string 容器中查找指定的子字符串。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。
t_len	字符串的长度。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 cpstr_string 的有效位置，否则函数的行为是未定义的。如果 s_cstr 为空或者 t_len == 0，则返回索引值。如果 t_len > 字符串的长度，则使用整个字符串。

```
size_t string_find_char(
```

```
const string_t* cpstr_string, char c_char, size_t t_pos);
```

描述:

在 string 容器中查找指定的字符。

参数:

cpstr_string	string 容器。
c_char	要查找的字符。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 cpstr_string 的有效位置，否则函数的行为是未定义的。

```
size_t string_rfind(  
    const string_t* cpstr_string, const string_t* cpstr_find, size_t t_pos);
```

描述:

在 string 容器中反向查找指定的 string。

参数:

cpstr_string	string 容器。
cpstr_find	要查找的 string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 cpstr_find == NULL 则函数的行为是未定义的，cpstr_string 和 cpstr_find 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string) 则从 string 的最后一个字符开始查找。如果 cpstr_find 为空，则返回索引值。

```
size_t string_rfind_cstr(  
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos);
```

描述:

在 string 容器中反向查找字符串。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string) 则从 string 的最后一个字符开始查找。如果 s_cstr 为空，则返回索引值。

```
size_t string_rfind_subcstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos, size_t t_len);
```

描述:

在 string 容器中反向查找子字符串。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。
t_len	要查找的字符串的长度。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string) 则从 string 的最后一个字符开始查找。如果 s_cstr 为空或者 t_len == 0，则返回索引值。如果 t_len > 字符串的长度，则使用整个字符串。

```
size_t string_rfind_char(
    const string_t* cpstr_string, char c_char, size_t t_pos);
```

描述:

在 string 容器中反向查找指定的字符。

参数:

cpstr_string	string 容器。
elem	要查找的字符。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string)，从最后一个字符开始查找。

```
size_t string_find_first_of(
    const string_t* cpstr_string, const string_t* cpstr_find, size_t t_pos);
```

描述:

在 string 容器中查找第一个在指定的 string 容器出现的字符。

参数:

cpstr_string	string 容器。
cpstr_find	要查找的 string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符的索引，如果查找失败返回 NPOS。

注意:

`cpstr_string == NULL` 或者 `cpstr_find == NULL` 则函数的行为是未定义的，`cpstr_string` 和 `cpstr_find` 必须是已经初始化的，否则函数的行为是未定义的。`t_pos` 是属于 `cpstr_string` 的有效位置，否则函数的行为是未定义的。如果 `cpstr_find` 为空，则返回索引值。

```
size_t string_find_first_of_cstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos);
```

描述:

在 `string` 容器中查找第一个在字符串中出现的字符。

参数:

<code>cpstr_string</code>	<code>string</code> 容器。
<code>s_cstr</code>	要查找的字符串。
<code>t_pos</code>	查找开始的位置。

返回值:

查找到的字符的索引，如果查找失败返回 `NPOS`。

注意:

`cpstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的，`cpstr_string` 必须是已经初始化的，否则函数的行为是未定义的。`t_pos` 是属于 `cpstr_string` 的有效位置，否则函数的行为是未定义的。如果 `s_cstr` 为空，则返回索引值。

```
size_t string_find_first_of_subcstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos, size_t t_len);
```

描述:

在 `string` 容器中查找第一个在子字符串中出现的字符。

参数:

<code>cpstr_string</code>	<code>string</code> 容器。
<code>s_cstr</code>	要查找的字符串。
<code>t_pos</code>	查找开始的位置。
<code>t_len</code>	要查找的字符串的长度。

返回值:

查找到的字符的索引，如果查找失败返回 `NPOS`。

注意:

`cpstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的，`cpstr_string` 必须是已经初始化的，否则函数的行为是未定义的。`t_pos` 是属于 `cpstr_string` 的有效位置，否则函数的行为是未定义的。如果 `s_cstr` 为空或者 `t_len == 0`，则返回索引值。如果 `t_len >` 字符串的长度，则使用整个字符串。

```
size_t string_find_first_of_char(
    const string_t* cpstr_string, char c_char, size_t t_pos);
```

描述:

在 `string` 容器中查找第一个出现的指定字符。

参数:

<code>cpstr_string</code>	<code>string</code> 容器。
<code>c_char</code>	要查找的字符。
<code>t_pos</code>	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 cpstr_string 的有效位置，否则函数的行为是未定义的。

```
size_t string_find_first_not_of(
    const string_t* cpstr_string, const string_t* cpstr_find, size_t t_pos);
```

描述:

在 string 容器中查找第一个不出现在指定 string 容器中的字符。

参数:

cpstr_string	string 容器。
cpstr_find	要查找的 string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 cpstr_find == NULL 则函数的行为是未定义的，cpstr_string 和 cpstr_find 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 cpstr_string 的有效位置，否则函数的行为是未定义的。如果 cpstr_find 为空，则返回索引值。

```
size_t string_find_first_not_of_cstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos);
```

描述:

在 string 容器中查找第一个不出现在字符串中的字符。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。

返回值:

查找到的字符的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 cpstr_string 的有效位置，否则函数的行为是未定义的。如果 s_cstr 为空，则返回索引值。

```
size_t string_find_first_not_of_subcstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos, size_t t_len);
```

描述:

在 string 容器中查找第一个不出现在子字符串中的字符。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。

`t_pos` 查找开始的位置。
`t_len` 要查找的字符串的长度。

返回值：
 查找到的字符的索引，如果查找失败返回 NPOS。

注意：
 `cpstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的，`cpstr_string` 必须是已经初始化的，否则函数的行为是未定义的。`t_pos` 是属于 `cpstr_string` 的有效位置，否则函数的行为是未定义的。如果 `s_cstr` 为空或者 `t_len == 0`，则返回索引值。如果 `t_len >` 字符串的长度，则使用整个字符串。

```
size_t string_find_first_not_of_char(
    const string_t* cpstr_string, char c_char, size_t t_pos);
```

描述：
 在 `string` 容器中查找第一个不是指定字符的字符。

参数：
 `cpstr_string` `string` 容器。
 `c_char` 要查找的字符。
 `t_pos` 查找开始的位置。

返回值：
 查找到的字符串的索引，如果查找失败返回 NPOS。

注意：
 `cpstr_string == NULL` 则函数的行为是未定义的，`cpstr_string` 必须是已经初始化的，否则函数的行为是未定义的。`t_pos` 是属于 `cpstr_basic_string` 的有效位置，否则函数的行为是未定义的。

```
size_t string_find_last_of(
    const string_t* cpstr_string, const string_t* cpstr_find, size_t t_pos);
```

描述：
 在 `string` 容器中查找最后一个出现在指定 `string` 容器中的字符。

参数：
 `cpstr_string` `string` 容器。
 `cpstr_find` 要查找的 `string` 容器。
 `t_pos` 查找开始的位置。

返回值：
 查找到的字符串的索引，如果查找失败返回 NPOS。

注意：
 `cpstr_string == NULL` 或者 `cpstr_find == NULL` 则函数的行为是未定义的，`cpstr_string` 和 `cpstr_find` 必须是已经初始化的，否则函数的行为是未定义的。如果 `t_pos >= string_size(cpstr_string)` 则从 `string` 的最后一个字符开始查找。如果 `cpstr_find` 为空，则返回索引值。

```
size_t string_find_last_of_cstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos);
```

描述：
 在 `string` 容器中查找最后一个出现在字符串中的字符。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string) 则从 string 的最后一个字符开始查找。如果 s_cstr 为空，则返回索引值。

```
size_t string_find_last_of_subcstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos, size_t t_len);
```

描述:

在 string 容器中查找最后一个出现在子字符串中的字符。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。
t_len	要查找的字符串的长度。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string) 则从 string 的最后一个字符开始查找。如果 s_cstr 为空或者 t_len == 0，则返回索引值。如果 t_len > 字符串的长度，则使用整个字符串。

```
size_t string_find_last_of_char(
    const string_t* cpstr_string, char c_char, size_t t_pos);
```

描述:

在 string 容器中查找最后一个出现的指定字符。

参数:

cpstr_string	string 容器。
c_char	要查找的字符。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string)，从最后一个字符开始查找。

```
size_t string_find_last_not_of(
```

```
const string_t* cpstr_string, const string_t* cpstr_find, size_t t_pos);
```

描述:

在 string 容器中查找最后一个不出现在指定 string 容器中的字符。

参数:

cpstr_string	string 容器。
cpstr_find	要查找的 string 容器。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 cpstr_find == NULL 则函数的行为是未定义的，cpstr_string 和 cpstr_find 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string) 则从 string 的最后一个字符开始查找。如果 cpstr_find 为空，则返回索引值。

```
size_t string_find_last_not_of_cstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos);
```

描述:

在 string 容器中查找最后一个不出现在字符串中的字符。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数的行为是未定义的。如果 t_pos >= string_size(cpstr_string) 则从 string 的最后一个字符开始查找。如果 s_cstr 为空，则返回索引值。

```
size_t string_find_last_not_of_subcstr(
    const string_t* cpstr_string, const char* s_cstr, size_t t_pos, size_t t_len);
```

描述:

在 string 容器中查找最后一个不出现在子字符串中的字符。

参数:

cpstr_string	string 容器。
s_cstr	要查找的字符串。
t_pos	查找开始的位置。
t_len	要查找的字符串的长度。

返回值:

查找到的字符串的索引，如果查找失败返回 NPOS。

注意:

cpstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的，cpstr_string 必须是已经初始化的，否则函数

的行为是未定义的。如果 `t_pos >= string_size(cpstr_string)` 则从 `string` 的最后一个字符开始查找。如果 `s_cstr` 为空或者 `t_len == 0`, 则返回索引值。如果 `t_len >` 字符串的长度, 则使用整个字符串。

```
size_t string_find_last_not_of_char(
    const string_t* cpstr_string, char c_char, size_t t_pos);
```

描述:

在 `string` 容器中查找最后一个不是指定字符的字符。

参数:

cpstr_string	string 容器。
c_char	要查找的字符。
t_pos	查找开始的位置。

返回值:

查找到的字符串的索引, 如果查找失败返回 NPOS。

注意:

`cpstr_string == NULL` 则函数的行为是未定义的, `cpstr_string` 必须是已经初始化的, 否则函数的行为是未定义的。如果 `t_pos >= string_size(cpstr_string)`, 从最后一个字符开始查找。

```
string_iterator_t string_begin(const string_t* cpstr_string);
```

描述:

返回指向 `string` 容器开头的迭代器。

参数:

cpstr_string	string 容器。
--------------	------------

返回值:

返回指向 `string` 容器开头的迭代器。

注意:

`cpstr_string == NULL` 则函数的行为是未定义的, `cpstr_string` 指向的容器是经过初始化以后的 `string` 容器, 否则函数的行为是未定义的。如果 `cpstr_string` 为空, 则返回值与 `string_end(cpstr_string)` 相等。

```
string_iterator_t string_end(const string_t* cpstr_string);
```

描述:

返回指向 `string` 容器末尾的迭代器。

参数:

cpstr_string	string 容器。
--------------	------------

返回值:

返回指向 `string` 容器末尾的迭代器。

注意:

`cpstr_string == NULL` 则函数的行为是未定义的, `cpstr_string` 指向的容器是经过初始化以后的 `string` 容器, 否则函数的行为是未定义的。

```
void string_clear(string_t* pstr_string);
```

描述:

清空 `string` 容器。

参数:

pstr_string string 容器。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的, pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。

```
void string_swap(string_t* pstr_first, string_t* pstr_second);
```

描述:

交换两个 string 容器的内容。

参数:

pstr_first string 容器。

pstr_second string 容器。

返回值:

无。

注意:

pstr_first == NULL 或者 pstr_second == NULL 则函数的行为是未定义的, pstr_first 和 pstr_second 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果 string_equal(pstr_first, pstr_second) 这函数不执行任何动作。

```
void string_reserve(string_t* pstr_string, size_t t_reservesize);
```

描述:

重新设置 string 容器的容量。

参数:

pstr_string string 容器。

t_reservesize 修改后的容量。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的, pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果 t_reservesize > string_capacity() 则函数执行后, string 容器的容量扩充到 t_reservesize。否则函数执行后 string 的容量不变。string 的容量只能增大不能够减小。

```
void string_assign(string_t* pstr_dest, const string_t* cpstr_src);
```

描述:

使用指定的 string 容器为 string 容器赋值。

参数:

pstr_dest 目的 string 容器。

cpstr_src 源 string 容器。

返回值:

无。

注意:

pstr_dest == NULL 或者 cpstr_src == NULL 则函数的行为是未定义的, pstr_dest 和 cpstr_src 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。赋值以后两个容器的 string_size() 相等但是 string_capacity() 不一定相等。如果 string_equal(pstr_dest, cpstr_src) 函数不执行任何操作。

```
void string_assign_substring(
    string_t* pstr_dest, const string_t* cpstr_src, size_t t_pos, size_t t_len);
```

描述:

使用指定的子 string 为 string 容器赋值。

参数:

pstr_dest	目的 string 容器。
cpstr_src	源 string 容器。
t_pos	子字符串开始的位置。
t_len	子字符串长度。

返回值:

无。

注意:

pstr_dest == NULL 或者 cpstr_src == NULL 则函数的行为是未定义的, pstr_dest 和 cpstr_src 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。t_pos 必须是属于源字符串的有效位置, 否则函数的行为是未定义的, 如果 t_len == NPOS 或者 t_pos + t_len >= 元字符串的长度, 那么使用从 t_pos 开始的所有子字符串。

```
void string_assign_cstr(string_t* pstr_string, const char* s_cstr);
```

描述:

使用指定的字符串为 string 容器赋值。

参数:

pstr_string	string 容器。
s_cstr	指定的字符串。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的。pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。

```
void string_assign_subcstr(string_t* pstr_string, const char* s_cstr, size_t t_len);
```

描述:

使用指定的子字符串为 string 容器赋值。

参数:

pstr_string	string 容器。
s_cstr	指定的字符串。
t_len	子串的长度。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的。pstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果 t_len 等于 NPOS 或者超过了字符串的长度则使用整个字符串进行初始化。

```
void string_assign_range(
    string_t* pstr_string, string_iterator_t it_begin, string_iterator_t it_end)
```

描述:

使用指定的字符区间为 string 容器赋值。

参数:

pstr_string	目的 string 容器。
it_begin	指定范围的开始。
it_end	指定范围的末尾。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的，pstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果 [it_begin, it_end) 属于 pstr_string 则函数的行为是未定义的。

```
void string_assign_char(string_t* pstr_string, size_t t_count, char c_char)
```

描述:

使用指定的字符为 string 容器赋值。

参数:

pvec_vector	目的 string 容器。
t_count	赋值的字符的个数。
c_char	赋值的字符。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的，pstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。

```
void string_append(string_t* pstr_dest, const string_t* cpstr_src);
```

描述:

向 string 容器末尾添加指定的 string 容器。

参数:

pstr_dest	目的 string 容器。
cpstr_src	源 string 容器。

返回值:

无。

注意:

pstr_dest == NULL 或者 cpstr_src == NULL 则函数的行为是未定义的，pstr_dest 和 cpstr_src 指向的容器是经过初始

化以后的 string 容器，否则函数的行为是未定义的。

```
void string_append_substring(
    string_t* pstr_dest, const string_t* cpstr_src, size_t t_pos, size_t t_len);
```

描述:

向 string 容器末尾添加指定的子 string。

参数:

pstr_dest	目的 string 容器。
cpstr_src	源 string 容器。
t_pos	子字符串开始的位置。
t_len	子字符串长度。

返回值:

无。

注意:

如果 pstr_dest == NULL 或者 cpstr_src == NULL 则函数的行为是未定义的，pstr_dest 和 cpstr_src 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。t_pos 必须是属于源字符串的有效位置，否则函数的行为是未定义的，如果 t_len == NPOS 或者 t_pos + t_len >= 元字符串的长度，那么使用从 t_pos 开始的所有子字符串。

```
void string_append_cstr(string_t* pstr_string, const char* s_cstr);
```

描述:

向 string 容器末尾添加指定的字符串。

参数:

pstr_string	string 容器。
s_cstr	指定的字符串。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的。pstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。

```
void string_append_subcstr(string_t* pstr_string, const char* s_cstr, size_t t_len);
```

描述:

向 string 容器末尾添加指定的子字符串。

参数:

pstr_string	string 容器。
s_cstr	指定的字符串。
t_len	子串的长度。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的。pstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。如果 t_len 等于 NPOS 或者超过了字符串的长度则使用整个字符串进行初

初始化。

```
void string_append_range(  
    string_t* pstr_string, string_iterator_t it_begin, string_iterator_t it_end)
```

描述:

向 string 容器末尾添加指定的字符区间。

参数:

pstr_string 目的 string 容器。
it_begin 指定范围的开始。
it_end 指定范围的末尾。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的, pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。如果[it_begin, it_end)属于 pstr_string 则函数的行为是未定义的。

```
void string_append_char(string_t* pstr_string, size_t t_count, char c_char)
```

描述:

向 string 容器末尾添加指定的字符。

参数:

pstr_string 目的 string 容器。
t_count 赋值的字符的个数。
elem 赋值的字符。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的, pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。

```
string_iterator_t string_insert(  
    string_t* pstr_string, string_iterator_t it_pos, char c_char);
```

描述:

向 string 容器中插入指定字符。

参数:

pstr_string string 容器。
it_pos 插入字符的位置。
c_char 用户指定的字符。

返回值:

返回引用被插入的第一个字符的迭代器。

注意:

如果 pstr_string == NULL 或者 string 是未初始化的 string 则函数的行为是未定义的。it_pos 必须是属于 string 容器的有效位置, 否则函数的行为是未定义的。

```
string_iterator_t string_insert_n(
    string_t* pstr_string, string_iterator_t it_pos, size_t t_count, char c_char);
```

描述:

向 string 容器的指定位置插入多个字符。

参数:

pstr_string	string 容器。
it_pos	插入字符的位置。
t_count	字符个数。
c_char	用户指定的字符。

返回值:

返回引用被插入的第一个字符的迭代器。

注意:

如果 pstr_string == NULL 或者 string 是未初始化的 string 则函数的行为是未定义的。it_pos 必须是属于 string 容器的有效位置，否则函数的行为是未定义的。

```
void string_insert_string(
    string_t* pstr_string, size_t t_pos, const string_t* cpstr_insert);
```

描述:

向 string 容器中插入指定的 string 容器。

参数:

cpstr_string	string 容器。
t_pos	插入的位置。
cpstr_insert	要插入的 string 容器。

返回值:

无。

注意:

pstr_string == NULL 或者 cpstr_insert == NULL 则函数的行为是未定义的，pstr_string 和 cpstr_insert 必须是已经初始化的，否则函数的行为是未定义的。t_pos 是属于 pstr_string 的有效位置，否则函数的行为是未定义的。如果 pstr_string == cpstr_insert，则函数的行为是未定义的。

```
void string_insert_substring(
    string_t* pstr_string, size_t t_pos,
    const string_t* cpstr_insert, size_t t_startpos, t_len);
```

描述:

向 string 容器中插入指定的子 string。

参数:

cpstr_string	string 容器。
t_pos	插入的位置。
cpstr_insert	要插入的 string 容器。
t_startpos	子 string 的开始位置。
t_len	子 string 的长度。

返回值:

无。

注意:

pstr_string == NULL 或者 cpstr_insert == NULL 则函数的行为是未定义的, pstr_string 和 cpstr_insert 必须是已经初始化的, 否则函数的行为是未定义的。t_pos 是属于 pstr_string 的有效位置, 否则函数的行为是未定义的。t_startpos 是属于 cpstr_insert 的有效位置, 否则函数的行为是未定义的。如果 t_len == NPOS 或者 t_len + t_startpos 大于等于 cpstr_insert 的长度, 则使用从 t_startpos 开始的全部 cpstr_insert 子串。如果 pstr_string == cpstr_insert, 则函数的行为是未定义的。

```
void string_insert_cstr(string_t* pstr_string, size_t t_pos, const char* s_cstr);
```

描述:

向 string 容器中插入指定的字符串。

参数:

cpstr_string	string 容器。
t_pos	插入的位置。
s_cstr	要插入的字符串。

返回值:

无。

注意:

pstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的, pstr_string 必须是已经初始化的, 否则函数的行为是未定义的。t_pos 是属于 pstr_string 的有效位置, 否则函数的行为是未定义的。

```
void string_insert_subcstr(
    string_t* pstr_string, size_t t_pos, const char* s_cstr, size_t t_len);
```

描述:

向 string 容器中插入指定的子字符串。

参数:

pstr_string	string 容器。
t_pos	插入的位置。
s_cstr	指定的字符串。
t_len	子串的长度。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 s_cstr == NULL 则函数的行为是未定义的。pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。t_pos 是属于 pstr_string 的有效位置, 否则函数的行为是未定义的。如果 t_len 等于 NPOS 或者超过了字符串的长度则使用整个字符串进行初始化。

```
void string_insert_range(
    string_t* pstr_string, string_iterator_t it_pos,
    string_iterator_t it_begin, string_iterator_t it_end)
```

描述:

向 string 容器中插入指定的字符区间。

参数:

pstr_string	目的 string 容器。
it_pos	插入的位置。
it_begin	指定范围的开始。
it_end	指定范围的末尾。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的, pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。it_pos 是属于 pstr_string 并且是有效的迭代器, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围, 否则函数的行为是未定义的。如果[it_begin, it_end) 属于 pstr_string 则函数的行为是未定义的。

```
void string_insert_char(  
    string_t* pstr_string, size_t t_pos, size_t t_count, char c_char)
```

描述:

向 string 容器中插入指定字符。

参数:

pstr_string	目的 string 容器。
it_pos	插入的位置。
t_count	插入的字符的个数。
c_char	插入的字符。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的, pstr_string 指向的容器是经过初始化以后的 string 容器, 否则函数的行为是未定义的。it_pos 是属于 pstr_string 并且是有效的迭代器, 否则函数的行为是未定义的。

```
void string_push_back(string_t* pstr_string, char c_char);
```

描述:

向 string 容器末尾添加指定数据。

参数:

pstr_string	string 容器。
c_char	指定的数据。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 string 是未初始化的则函数的行为是未定义的。

```
void string_resize(string_t* pt_string, size_t t_resize, char c_char);
```

描述:

重新设置 string 容器中数据的个数。

参数:

pt_string string 容器。
t_resize 数据个数。
c_char 用户指定的数据。

返回值:

无。

注意:

如果 pt_string == NULL 或者 string 是未初始化的 string 则函数的行为是未定义的。t_resize 可以小于原来的数据数量，这是将末尾的数据删除，当 t_resize 大于原来数据的数量的时候，使用指定的数据填充。

```
string_iterator_t string_erase(string_t* pstr_string, string_iterator_t it_pos);
```

描述:

删除 string 容器中指定位置的字符。

参数:

pstr_string string 容器。
it_pos 插入的位置。

返回值:

指向被删除的字符后面字符的迭代器。

注意:

如果 pstr_string == NULL 或者 string 是未初始化的 string 则函数的行为是未定义的。it_pos 是属于 string 的有效的迭代器，否则函数的行为是未定义的。

```
string_iterator_t string_erase_range(  
    string_t* pstr_string, string_iterator_t it_begin, string_iterator_t it_end)
```

描述:

删除 string 容器中指定的字符区间。

参数:

pstr_string string 容器。
it_begin 指定范围的开始。
it_end 指定范围的末尾。

返回值:

指向被删除的字符后面字符的迭代器。

注意:

pstr_string == NULL 则函数的行为是未定义的，pstr_string 指向的容器是经过初始化以后的 string 容器，否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围，否则函数的行为是未定义的。

```
void string_erase_substring(string_t* pstr_string, size_t t_pos, size_t t_len);
```

描述:

删除 string 容器中的子数据串。

参数:

pstr_string string 容器。
t_pos 删除的位置。

`t_len` 子串的长度。

返回值:

无。

注意:

如果 `pstr_string == NULL` 则函数的行为是未定义的。`pstr_string` 指向的容器是经过初始化以后的 `string` 容器, 否则函数的行为是未定义的。`t_pos` 是属于 `pstr_string` 的有效位置, 否则函数的行为是未定义的。如果 `t_len` 等于 `NPOS` 或者超过了字符串的长度则删除从 `t_pos` 开始剩余全部的字串。

```
void string_replace(
    string_t* pstr_string, size_t t_pos, size_t t_len,
    const string_t* cpstr_replace);
```

描述:

将 `string` 容器中的子字符串替换成指定的 `string`。

参数:

<code>pstr_string</code>	<code>string</code> 容器。
<code>t_pos</code>	替换的位置。
<code>t_len</code>	替换的长度。
<code>cpstr_replace</code>	指定的 <code>string</code> 。

返回值:

无。

注意:

`pstr_string == NULL` 或者 `cpstr_replace == NULL` 则函数的行为是未定义的, `pstr_string` 和 `cpstr_replace` 必须是已经初始化的, 否则函数的行为是未定义的。`t_pos` 是属于 `pstr_string` 的有效位置, 否则函数的行为是未定义的。如果 `t_len == NPOS` 或者 `t_len + t_pos` 大于等于 `pstr_string` 的长度, 则使用从 `t_pos` 开始的全部 `pstr_string` 子串。如果 `pstr_string == cpstr_replace`, 则函数的行为是未定义的。

```
void string_replace_substring(
    string_t* pstr_string, size_t t_pos, size_t t_len,
    const string_t* cpstr_replace, size_t t_position, size_t t_length);
```

描述:

将 `string` 容器中的子字符串替换成指定的子 `string`。

参数:

<code>pstr_string</code>	<code>string</code> 容器。
<code>t_pos</code>	替换的位置。
<code>t_len</code>	替换的长度。
<code>cpstr_replace</code>	指定的 <code>string</code> 。
<code>t_position</code>	拷贝的位置。
<code>t_length</code>	拷贝的长度。

返回值:

无。

注意:

`pstr_string == NULL` 或者 `cpstr_replace == NULL` 则函数的行为是未定义的, `pstr_string` 和 `cpstr_replace` 必须是已经初始化的, 否则函数的行为是未定义的。`t_pos` 是属于 `pstr_string` 的有效位置, 否则函数的行为是未定义的。`t_position` 是

属于 `cpstr_replace` 的有效位置，否则函数的行为是未定义的。如果 `t_len == NPOS` 或者 `t_len + t_pos` 大于等于 `pstr_string` 的长度，则使用从 `t_pos` 开始的全部 `pstr_string` 子串。如果 `t_length == NPOS` 或者 `t_length + t_position` 大于等于 `cpstr_replace` 的长度，则使用从 `t_position` 开始的全部 `cpstr_replace` 子串。如果 `pstr_string == cpstr_replace`，则函数的行为是未定义的。

```
void string_replace_cstr(
    string_t* pstr_string, size_t t_pos, size_t t_len, const char* s_cstr);
```

描述：

将 `string` 容器中的子字符串替换成指定的字符串。

参数：

<code>pstr_string</code>	<code>string</code> 容器。
<code>t_pos</code>	替换的位置。
<code>t_len</code>	替换的长度。
<code>s_cstr</code>	指定的字符串。

返回值：

无。

注意：

`pstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的，`pstr_string` 必须是已经初始化的，否则函数的行为是未定义的。`t_pos` 是属于 `pstr_string` 的有效位置，否则函数的行为是未定义的。如果 `t_len == NPOS` 或者 `t_len + t_pos` 大于等于 `pstr_string` 的长度，则使用从 `t_pos` 开始的全部 `pstr_string` 子串。

```
void string_replace_subcstr(
    string_t* pstr_string, size_t t_pos, size_t t_len,
    const char* s_cstr, size_t t_length);
```

描述：

将 `string` 容器中的子字符串替换成指定的子字符串。

参数：

<code>pstr_string</code>	<code>string</code> 容器。
<code>t_pos</code>	替换的位置。
<code>t_len</code>	替换的长度。
<code>s_cstr</code>	指定的字符串。
<code>t_length</code>	字符串的长度。

返回值：

无。

注意：

`pstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的，`pstr_string` 必须是已经初始化的，否则函数的行为是未定义的。`t_pos` 是属于 `pstr_string` 的有效位置，否则函数的行为是未定义的。如果 `t_len == NPOS` 或者 `t_len + t_pos` 大于等于 `pstr_string` 的长度，则使用从 `t_pos` 开始的全部 `pstr_string` 子串。如果 `t_length == NPOS` 或者 `t_length` 大于等于 `s_cstr` 的长度，则使用全部 `s_cstr` 子串。

```
void string_replace_char(
    string_t* pstr_string, size_t t_pos, size_t t_len, size_t t_count, char c_char);
```

描述:

将 string 容器中的子字符串替换成指定的字符。

参数:

pstr_string	string 容器。
t_pos	替换的开始位置。
t_len	替换的长度。
t_count	字符个数。
c_char	用户指定的字符。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 string 是未初始化的 string 则函数的行为是未定义的。t_pos 必须在 string 有效范围内, 否则函数的行为是未定义的, t_len == NPOS 或者 t_pos + t_len >= string_size(), 将 t_pos 开始到末尾的全部串替换。用户指定的字符必须和容器中保存的字符类型是一致的否则函数的行为是未定义的。

```
void string_range_replace(
    string_t* pstr_string, string_iterator_t it_begin, string_iterator_t it_end,
    const string_t* cpstr_replace);
```

描述:

将 string 容器中的字符区间替换成 string。

参数:

pstr_string	string 容器。
it_begin	指定范围的开始。
it_end	指定范围的末尾。
cpstr_replace	指定的 string。

返回值:

无。

注意:

pstr_string == NULL 或者 cpstr_replace == NULL 则函数的行为是未定义的, pstr_string 和 cpstr_replace 必须是已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end) 是有效的范围, 否则函数的行为是未定义的。如果 pstr_string == cpstr_replace, 则函数的行为是未定义的。

```
void string_range_replace_substring(
    string_t* pstr_string, string_iterator_t it_begin, string_iterator_t it_end,
    const string_t* cpstr_replace, size_t t_position, size_t t_length);
```

描述:

将 string 容器中的子字符串替换成指定的子 string。

参数:

pstr_string	string 容器。
it_begin	指定范围的开始。
it_end	指定范围的末尾。
cpstr_replace	指定的 string。
t_position	拷贝的位置。

`t_length` 拷贝的长度。

返回值:

无。

注意:

`pstr_string == NULL` 或者 `cpstr_replace == NULL` 则函数的行为是未定义的, `pstr_string` 和 `cpstr_replace` 必须是已经初始化的, 否则函数的行为是未定义的。`[it_begin, it_end)` 是有效的范围, 否则函数的行为是未定义的。`t_position` 是属于 `cpstr_replace` 的有效位置, 否则函数的行为是未定义的。如果 `t_length == NPOS` 或者 `t_length + t_position` 大于等于 `cpstr_replace` 的长度, 则使用从 `t_position` 开始的全部 `cpstr_replace` 子串。如果 `pstr_string == cpstr_replace`, 则函数的行为是未定义的。

```
void string_range_replace_cstr(
    string_t* pstr_string, string_iterator_t it_begin, string_iterator_t it_end,
    const char* s_cstr);
```

描述:

将 `string` 容器中的字符区间替换成字符串。

参数:

<code>pstr_string</code>	<code>string</code> 容器。
<code>it_begin</code>	指定范围的开始。
<code>it_end</code>	指定范围的末尾。
<code>s_cstr</code>	指定的字符串。

返回值:

无。

注意:

`pstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的, `pstr_string` 必须是已经初始化的, 否则函数的行为是未定义的。`pstr_string` 和 `s_cstr` 是有效的范围, 否则函数的行为是未定义的。

```
void string_range_replace_subcstr(
    string_t* pstr_string, string_iterator_t it_begin, string_iterator_t it_end,
    const char* s_cstr, size_t t_length);
```

描述:

将 `string` 容器中的字符区间替换成子字符串区间。

参数:

<code>pstr_string</code>	<code>string</code> 容器。
<code>it_begin</code>	指定范围的开始。
<code>it_end</code>	指定范围的末尾。
<code>s_cstr</code>	指定的字符串。
<code>t_length</code>	字符串的长度。

返回值:

无。

注意:

`pstr_string == NULL` 或者 `s_cstr == NULL` 则函数的行为是未定义的, `pstr_string` 必须是已经初始化的, 否则函数的行为是未定义的。`[it_begin, it_end)` 是有效的范围, 否则函数的行为是未定义的。如果 `t_length == NPOS` 或者 `t_length` 大于等于 `s_cstr` 的长度, 则使用全部 `s_cstr` 子串。

```
void string_range_replace_char(
    string_t* pstr_string, erator_t it_begin, string_iterator_t it_end,
    size_t t_count, char c_char);
```

描述:

将 string 容器中的字符区间替换成指定的字符。

参数:

pstr_string	string 容器。
it_begin	指定范围的开始。
it_end	指定范围的末尾。
t_count	字符个数。
c_char	用户指定的字符。

返回值:

无。

注意:

如果 pstr_string == NULL 或者 string 是未初始化的 string 则函数的行为是未定义的。[it_begin, it_end)是有效的范围，否则函数的行为是未定义的。用户指定的字符必须和容器中保存的字符类型是一致的否则函数的行为是未定义的。

```
void string_replace_range(
    string_t* pstr_string,
    string_iterator_t it_begin, string_iterator_t it_end,
    string_iterator_t it_first, string_iterator_t it_last);
```

描述:

将 string 容器中的子字符串替换成指定的子 string。

参数:

pstr_string	string 容器。
it_begin	指定范围的开始。
it_end	指定范围的末尾。
it_first	指定范围的开始。
it_last	指定范围的末尾。

返回值:

无。

注意:

pstr_string == NULL 则函数的行为是未定义的， pstr_string 必须是已经初始化的，否则函数的行为是未定义的。[it_begin, it_end)是有效的范围，否则函数的行为是未定义的。如果[it_first, it_last)属于 pstr_string，则函数的行为是未定义的。

```
void string_output(const string_t* cpstr_string, FILE* fp_stream);
```

描述:

将 string 容器中的字符串输出到指定的流中。

参数:

cpstr_string	string 容器。
--------------	------------

fp_stream 指定的流。

返回值:

无。

注意:

cpstr_string == NULL 或者 fp_stream == NULL 则函数的行为是未定义的, cpstr_string 必须是已经初始化的, 否则函数的行为是未定义的。

```
void string_input(string_t* pstr_string, FILE* fp_stream);
```

描述:

从指定的流中读取字符串到 string 容器。

参数:

pstr_string string 容器。

fp_stream 指定的流。

返回值:

无。

注意:

pstr_string == NULL 或者 fp_stream == NULL 则函数的行为是未定义的, pstr_string 必须是已经初始化的, 否则函数的行为是未定义的, fp_stream 是有效的输入流, 否则函数的行为是未定义的。

```
bool_t string_getline(string_t* pstr_string, FILE* fp_stream);
```

描述:

从指定的流中读取一行并保存到 string 容器中。

参数:

pstr_string string 容器。

fp_stream 指定的流。

返回值:

成功的从流中获得一行字符则返回 true, 否则返回 false。

注意:

pstr_string == NULL 或者 fp_stream == NULL 则函数的行为是未定义的, pstr_string 必须是已经初始化的, 否则函数的行为是未定义的。

```
bool_t string_getline_delimiter(
    string_t* pstr_string, FILE* fp_stream, char cDelimiter);
```

描述:

从指定的流中读取一行并保存到 string 容器中, 用户指定行标识符。

参数:

pstr_string string 容器。

fp_stream 指定的流。

cDelimiter 用户指定的行标识符。

返回值:

成功的从流中获得一行字符则返回 true, 否则返回 false。

注意:

`pstr_string == NULL` 或者 `fp_stream == NULL` 则函数的行为是未定义的，`pstr_string` 必须是已经初始化的，否则函数的行为是未定义的。

第五节 迭代器接口

由于直接使用 `basic_string` 迭代器作为实现，所以 `string` 并没有提供迭代器接口。

第六节 内部和辅助接口

`string` 也只是提供了简单的内部接口。

<code>_create_string_auxiliary</code>	创建 <code>string</code> 容器的辅助函数。
<code>_string_destroy_auxiliary</code>	销毁 <code>string</code> 容器的辅助函数。

```
bool_t _create_string_auxiliary(string_t* pstr_string);
```

描述:

创建一个 `string` 容器的辅助函数。

参数:

`pstr_string` 没有创建的 `string` 容器。

返回值:

创建成功返回 `true`, 否则返回 `false`。

注意:

如果 `pstr_string == NULL` 则函数的行为是未定义的。

```
void _string_destroy_auxiliary(string_t* pstr_string);
```

描述:

销毁 `string` 容器的辅助函数。

参数:

`pstr_string` `string` 容器。

返回值:

无。

注意:

如果 `pstr_string == NULL` 或者 `string` 不是使用 `create_string` 生成的则函数的行为是未定义的。

第十四章 avl tree

avl tree 是关联容器的底层的一种实现(另一种实现是红黑树)，它是一种平衡的二叉搜索树，但是在插入或者删除节点的时候可能破坏这种平衡，所以对 avl tree 进行插入或者删除后要进行调整使它恢复平衡。

第一节 avl tree 的机制

《数据结构与算法分析 --- C 语言描述》中详细的介绍了 avl tree。

二叉搜索树的特点是每一个非空节点都有一个左子节点和一个右子节点，这两个字节点可以为空。其中左字节点的值小于父节点，右字节点的值大于父节点。这样二叉搜索树中的值最小的节点位于最左边的节点，最大值位于树的最右边的节点：

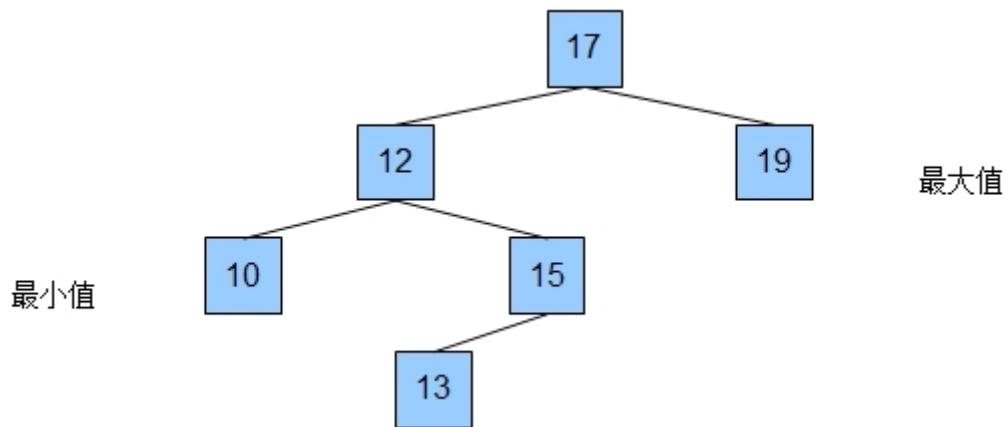


图 14.1 二叉搜索树

插入数据的时候：

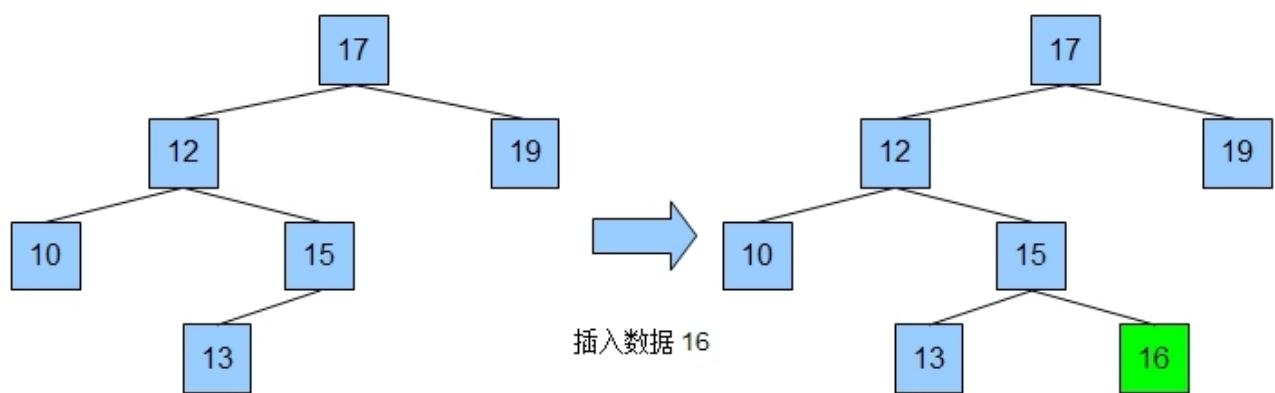


图 14.2 向二叉搜索树中插入数据

删除数据的时候比较复杂，当删除的节点是叶子节点时直接删除就可以了，如果包含一个子节点的情况也是比较

好处理的：

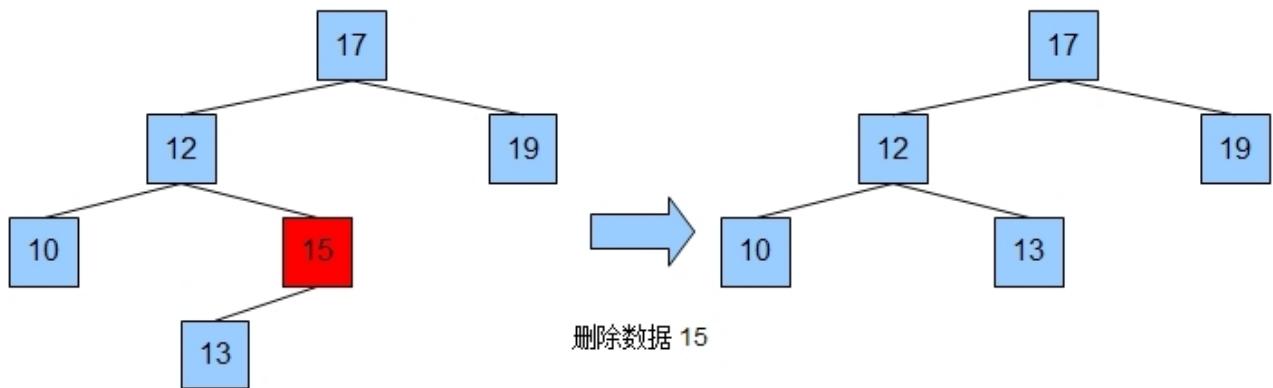


图 14.3 删除带有一个子节点的节点

删除带有两个子节点的节点要复杂一些，要将右子树中的值最小的节点替代当前要删除的节点：

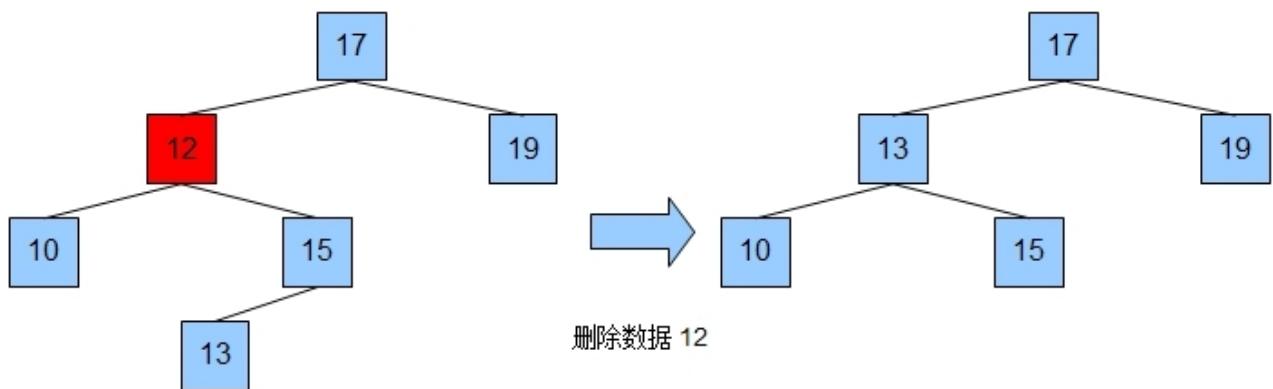


图 14.4 删除带有两个子节点的节点

上面的规则都是二叉搜索树的规则，但是往往由于插入的随机性可能造成下面这样的情况：

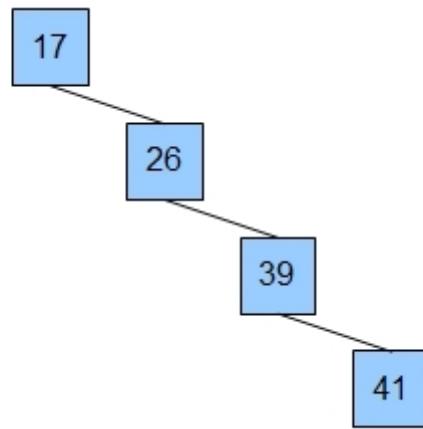


图 14.5 效率低下的二叉搜索树

为了让二叉搜索树能够保持较好的搜索效率就应该让它尽量保持平衡，也就是所谓的平衡二叉搜索树，avl tree 就是一种常见的平衡二叉搜索树。对于上面的这种情况 avl tree 会通过插入后调整来保持树的平衡。

为了让二叉搜索树具有最佳的搜索效率，应该让一个节点的左右子树保持相同的高度，这就是完全平衡二叉搜索树，但是受到输入条件的限制，往往不能够达到这样的效果，所以 avl tree 的平衡条件略微的宽松，它要求左右子树的高度差不大于 1。所以上面的例子就不是 avl tree。

当插入或者删除节点的时候都会破坏 avl tree 的平衡，avl tree 是通过旋转的方式来重新达到平衡的。当某一个节点失去平衡后，那么它的祖先节点都是不平衡的，所以要将一个节点从不平衡的位置调整为平衡，只需要调整最深层的不平衡节点。

当插入数据的时候会造成不平衡，这分为两种情况，当插入的数据位于外侧的时候：

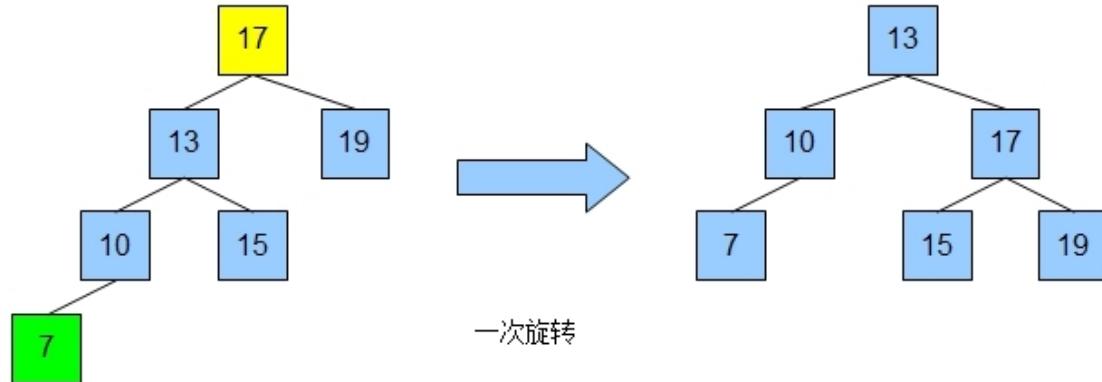


图 14.6 单旋转

当插入数据 7 之后，根节点 17 就不平衡了，左边高度是 3，右边高度是 1。这样的情况可以通过一次旋转来完成平衡。但是下面这样的情况通过一次旋转是不能够完成平衡的：

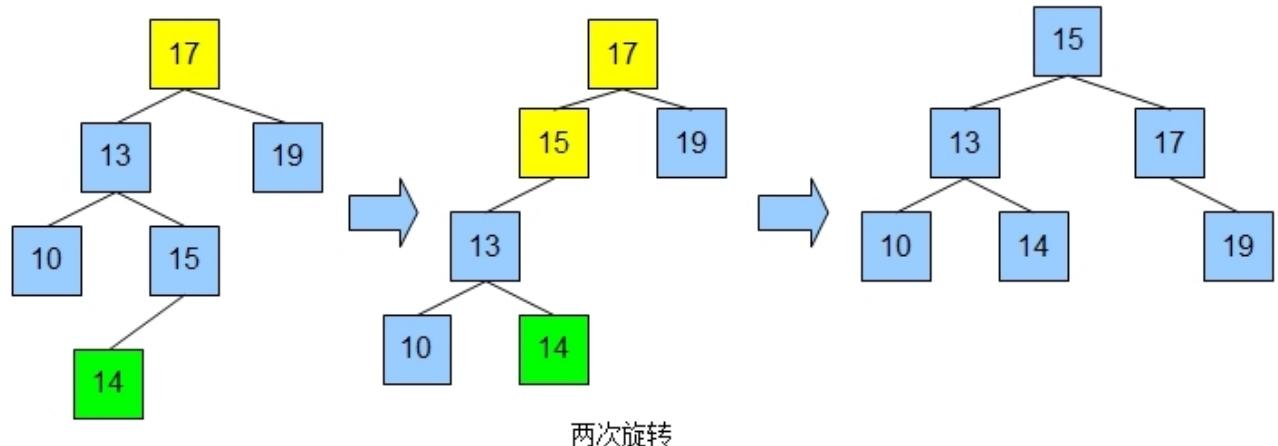


图 14.7 双旋转

总的来说调整平衡的两种情况，这两种情况都是针对左子树高于右子树的情况，当右子树高于左子树的情况是与下面的例子对称：

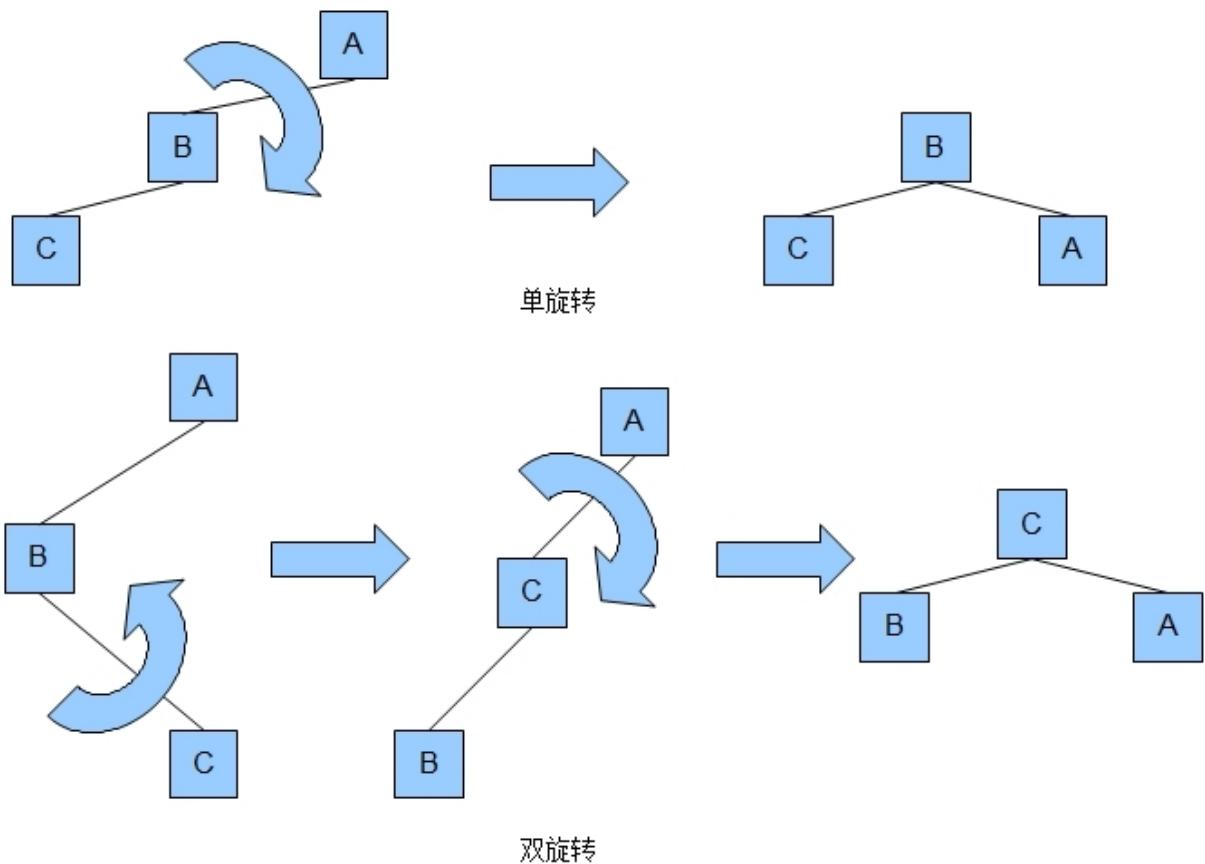


图 14.8 两种旋转

当删除一个节点的时候也会造成 avl tree 失去平衡，这时候就需要找到失去平衡的节点并且调整。调整的方法和上面的相同。

libcstl 中的 avl tree 的每个节点都是单独的结构，使用这个结构保存的指针可以找到父节点，左子树，右子树：

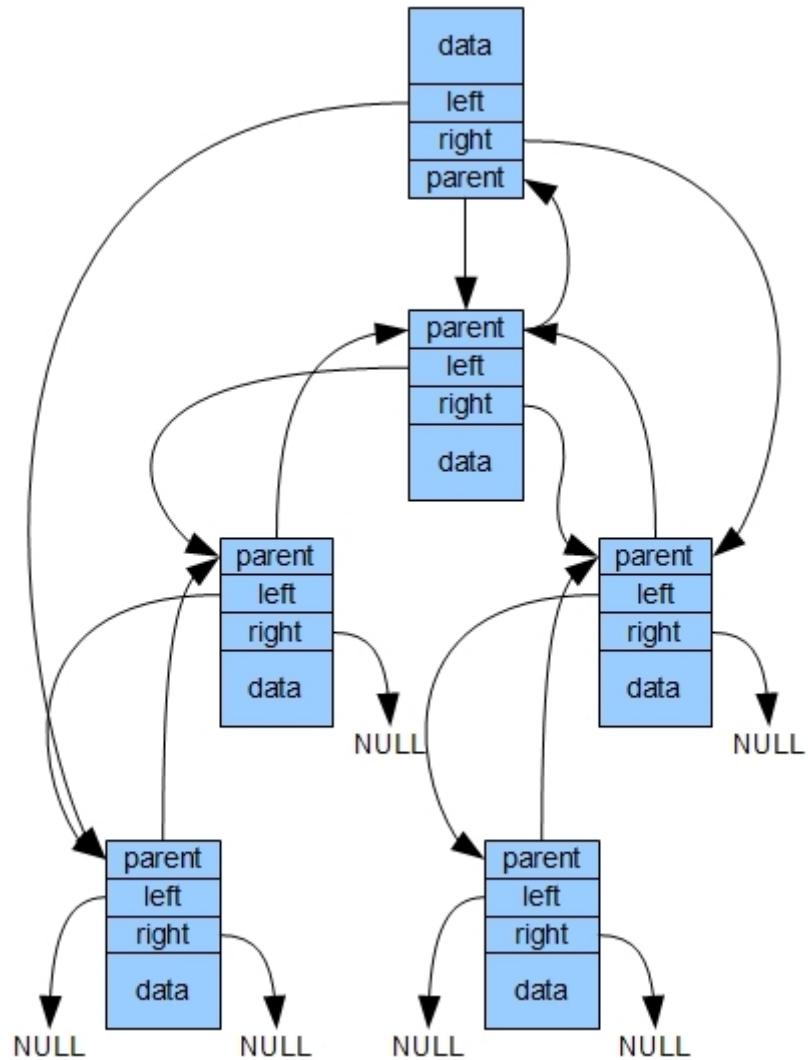


图 14.9 avl tree 节点结构

第二节 avl tree 迭代器

avl tree 的迭代器是双向迭代器，它是内部使用的迭代器，用来实现关联容器的迭代器。begin 是指向值最小的节点，end 指向 avl tree 的根。迭代器向前向后移动是根据数值的顺序移动。

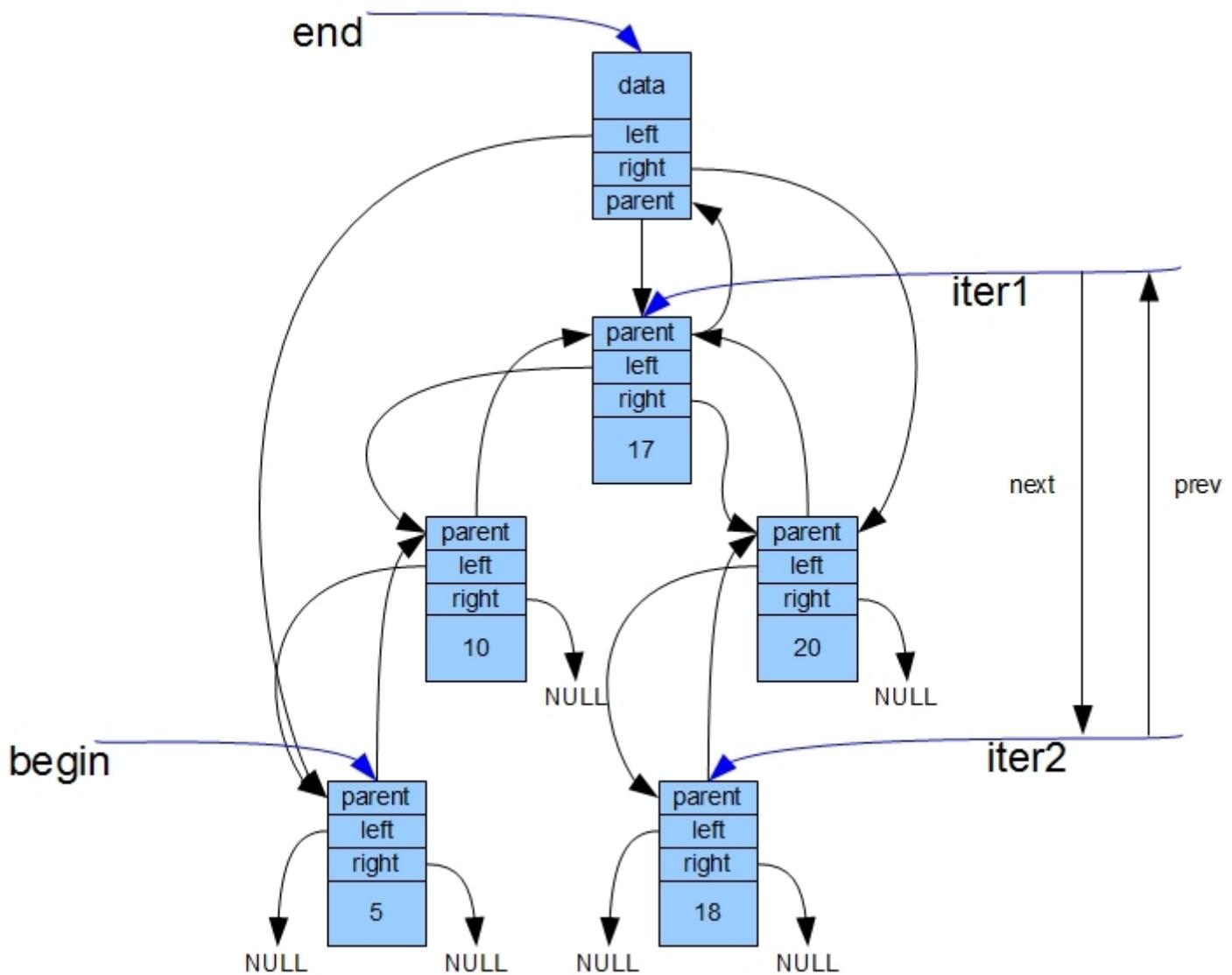


图 14.10 avl tree 迭代器

要使用特殊的结构表示 avl tree 的迭代器:

```

typedef struct _tagiterator
{
    /* 容器内部结构的信息 */
    union
    {
        char*      _pc_corepos;
        void*      _pt_tree;          /* 指向 avl tree */
        _t_pos;
        void*      _pt_container;    /* 容器的位置 */
        containertype_t _t_containertype; /* 容器的类型 */
        iteratortype_t _t_iteratortype; /* 迭代器的类型 */
    }iterator_t;
}

```

第三节 avl tree 代码结构

avl tree 节点中包含很多信息：

```
typedef struct _tagavlnode
{
    struct _tagavlnode* _pt_parent;
    struct _tagavlnode* _pt_left;
    struct _tagavlnode* _pt_right;
    unsigned int         _un_height;
    char                _pc_data[1];
}avlnode_t;
```

其中 _un_height 表示当前节点在书中的高度，叶子节点高度为 0，根的高度是左右子树高度的大值加 1。

下面的结构表示 avl tree：

```
typedef struct _tagavltree
{
    _typeinfo_t          _t_typeinfo;
    _alloc_t             _t_allocator;
    avlnode_t            _t_avlroot;
    size_t               _t_nodecount;
    binary_function_t   _t_compare;
}avl_tree_t;
```

结构中的 _t_avlroot 中 _pt_parent 指向实际 avl tree 的根，_pt_left 指向 avl tree 中值最小的数据，_pt_right 指向 avl tree 中值最大的数据，_un_height 为 0，_pc_data 数据未使用如图 14.10。

第四节 外部接口

avl tree 提供的外部接口是为了实现关联容器。

_create_avl_tree	创建 avl tree 容器。
_avl_tree_init	初始化 avl tree 容器。
_avl_tree_init_copy	使用存在的 avl tree 容器进行初始化。
_avl_tree_init_copy_range	使用指定的数据区间进行初始化。
_avl_tree_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
_avl_tree_destroy	销毁 avl tree 容器。
_avl_tree_assign	使用存在的 avl tree 容器赋值。

_avl_tree_size	返回 avl tree 容器中数据的数量。
_avl_tree_empty	判断 avl tree 容器是否为空。
_avl_tree_max_size	返回 avl tree 容器中能够保存数据的最大数量。
_avl_tree_begin	返回指向 avl tree 容器的第一个数据的迭代器。
_avl_tree_end	返回指向 avl tree 容器末尾的迭代器。
_avl_tree_key_comp	返回 avl tree 容器中的数据比较规则。
_avl_tree_find	在 avl tree 容器中查找指定的数据。
_avl_tree_clear	清空 avl tree 容器。
_avl_tree_count	返回 avl tree 容器中指定数据的数量。
_avl_tree_lower_bound	返回指向第一个大于等于指定数据的迭代器。
_avl_tree_upper_bound	返回指向第一个大于指定数据的迭代器。
_avl_tree_equal_range	返回包含指定数据的数据区间。
_avl_tree_equal	测试两个 avl tree 容器是否相等。
_avl_tree_not_equal	测试两个 avl tree 容器是否不等。
_avl_tree_less	测试第一个 avl tree 容器是否小于第二个 avl tree 容器。
_avl_tree_less_equal	测试第一个 avl tree 容器是否小于等于第二个 avl tree 容器。
_avl_tree_greater	测试第一个 avl tree 容器是否大于第二个 avl tree 容器。
_avl_tree_greater_equal	测试第一个 avl tree 容器是否大于等于第二个 avl tree 容器。
_avl_tree_swap	交换两个 avl tree 容器。
_avl_tree_insert_unique	向 avl tree 中插入数据，数据必须唯一。
_avl_tree_insert_equal	向 avl tree 中插入数据，数据可以重复。
_avl_tree_insert_unique_range	向 avl tree 中插入指定的数据区间，数据必须唯一。
_avl_tree_insert_equal_range	向 avl tree 中插入指定的数据区间，数据可以重复。
_avl_tree_erase_pos	删除 avl tree 中指定位置的数据。
_avl_tree_erase_range	删除 avl tree 中指定数据区间的数据。
_avl_tree_erase	删除 avl tree 中指定的数据。

函数原型

```
_avl_tree_t* _create_avl_tree(const char* s_typename);
```

描述:

创建一个 avl tree 迭代器。

参数:

s_typename 类型描述。

返回值:

成功返回指向 avl tree 容器的指针，否则返回 NULL。

注意：

s_typename != NULL，否则函数的行为是未定义的。

```
void _avl_tree_init(_avl_tree_t* pt_avl_tree, binary_function_t t_compare);
```

描述：

初始化 avl tree 迭代器。

参数：

pt_avl_tree	avl tree 容器。
t_compare	数据比较规则。

返回值：

无。

注意：

pt_avl_tree == NULL 则函数的行为是未定义的，pt_avl_tree 必须是使用_create_avl_tree() 创建的，否则函数的行为是未定义的，如果 t_compare == NULL 则使用类型默认的比较函数。

```
void _avl_tree_init_copy(_avl_tree_t* pt_dest, const _avl_tree_t* cpt_src);
```

描述：

使用已经存在的 avl tree 初始化 avl tree 迭代器。

参数：

pt_dest	目的 avl tree 容器。
cpt_src	源 avl tree 容器。

返回值：

无。

注意：

pt_dest == NULL 或者 cpt_src == NULL 则函数的行为是未定义的，pt_dest 必须是使用_create_avl_tree() 创建的，cpt_src 必须是已经初始化，否则函数的行为是未定义的。pt_dest 和 cpt_src 保存的数据类型必须相同，否则函数的行为是未定义。

```
void _avl_tree_init_copy_range(
    _avl_tree_t* pt_dest,
    _avl_tree_iterator_t it_begin, _avl_tree_iterator_t it_end);
```

描述：

使用指定的数据区间初始化 avl tree 迭代器。

参数：

pt_dest	目的 avl tree 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。

返回值：

无。

注意：

如果 pt_dest == NULL 则函数的行为是未定义的，pt_dest 必须是使用_create_avl_tree() 创建的否则函数的行为是未

定义的。`[it_begin, it_end)`属于一个已经初始化的 avl tree 容器的有效数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void _avl_tree_init_copy_range_ex(
    _avl_tree_t* pt_dest,
    _avl_tree_iterator_t it_begin, _avl_tree_iterator_t it_end,
    binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 avl tree 迭代器。

参数:

pt_dest	目的 avl tree 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。
t_compare	比较函数。

返回值:

无。

注意:

如果 `pt_dest == NULL` 则函数的行为是未定义的，`pt_dest` 必须是使用 `_create_avl_tree()` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 avl tree 容器的有效数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 `t_compare == NULL` 则使用类型默认的比较函数。

```
void _avl_tree_destroy(_avl_tree_t* pt_avl_tree);
```

描述:

销毁创建的 avl tree 容器。

参数:

pt_avl_tree	avl tree 容器。
-------------	--------------

返回值:

无。

注意:

如果 `pt_avl_tree == NULL` 则函数的行为是未定义的，`pt_avl_tree` 必须是初始化或者是使用 `_create_avl_tree()` 创建的，否则函数的行为是未定义的。

```
void _avl_tree_assign(_avl_tree_t* pt_dest, const _avl_tree_t* cpt_src);
```

描述:

使用一个 avl tree 为另一个 avl tree 赋值。

参数:

pt_dest	指向被赋值的 avl tree 容器的指针。
cpt_src	指向赋值的 avl tree 容器的指针。

返回值:

无。

注意:

如果 `pt_dest == NULL` 或者 `cpt_src == NULL` 则函数的行为是未定义的，两个 avl tree 必须已经初始化的，否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同函数的行为是未定义的。如果 `_avl_tree_equal(pt_dest, cpt_src)` 那么函数不做任何动作。

```
size_t _avl_tree_size(const _avl_tree_t* cpt_avl_tree);
```

描述:

获得 avl tree 容器中保存的数据的个数。

参数:

`cpt_avl_tree` avl tree 容器。

返回值:

avl tree 容器中保存的数据的个数。

注意:

如果 `cpt_avl_tree == NULL` 则函数的行为是未定义的，`cpt_avl_tree` 必须是已经初始化的 avl tree 容器，否则函数的行为是未定义的。

```
bool_t _avl_tree_empty(const _avl_tree_t* cpt_avl_tree);
```

描述:

测试 avl tree 容器是否为空。

参数:

`cpt_avl_tree` avl tree 容器。

返回值:

如果 avl tree 容器为空则返回 `true`，否则返回 `false`。

注意:

如果 `cpt_avl_tree == NULL` 则函数的行为是未定义的，`cpt_avl_tree` 必须是已经初始化的 avl tree 容器，否则函数的行为是未定义的。

```
size_t _avl_tree_max_size(const _avl_tree_t* cpt_avl_tree);
```

描述:

获得 avl tree 容器中数据的最大数量。

参数:

`cpt_avl_tree` avl tree 容器。

返回值:

avl tree 容器中保存的数据的个数的最大值。

注意:

如果 `cpt_avl_tree == NULL` 则函数的行为是未定义的，`cpt_avl_tree` 必须是已经初始化的 avl tree 容器，否则函数的行为是未定义的。

```
_avl_tree_iterator_t _avl_tree_begin(const _avl_tree_t* cpt_avl_tree);
```

描述:

获得引用 avl tree 容器中第一数据的迭代器。

参数:

`cpt_avl_tree` 指向 avl tree 容器的指针。

返回值:

返回引用 avl tree 容器中第一个数据的迭代器。

注意:

如果 cpt_avl_tree == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。如果 avl tree 容器为空, 则返回值与 _avl_tree_end(cpt_avl_tree) 相等。

```
_avl_tree_iterator_t _avl_tree_end(const _avl_tree_t* cpt_avl_tree);
```

描述:

获得引用 avl tree 容器末尾的迭代器。

参数:

cpt_avl_tree 指向 avl tree 容器的指针。

返回值:

返回引用 avl tree 容器末尾的迭代器。

注意:

如果 cpt_avl_tree == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t _avl_tree_key_comp(const _avl_tree_t* cpt_avl_tree);
```

描述:

返回 avl tree 容器中的数据比较规则。

参数:

cpt_avl_tree 指向 avl tree 容器的指针。

返回值:

返回 avl tree 容器中的数据比较规则。

注意:

如果 cpt_avl_tree == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。

```
_avl_tree_iterator_t _avl_tree_find(
    const _avl_tree_t* cpt_avl_tree, const void* cpv_value);
```

描述:

在 avl tree 容器中查找指定的数据。

参数:

cpt_avl_tree 指向 avl tree 容器的指针。

cpv_value 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 avl tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _avl_tree_clear(_avl_tree_t* pt_avl_tree);
```

描述:

删除 avl tree 容器中的所有数据。

参数:

pt_avl_tree 指向 avl tree 容器的指针。

返回值:

无。

注意:

如果 pt_avl_tree == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。

```
size_t _avl_tree_count(const _avl_tree_t* cpt_avl_tree, const void* cpv_value);
```

描述:

返回 avl tree 容器中指定数据的数量。

参数:

cpt_avl_tree 指向 avl tree 容器的指针。

cpv_value 要查找的指定数据。

返回值:

返回 avl tree 容器中指定数据的数量。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 avl tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
_avl_tree_iterator_t _avl_tree_lower_bound(
    const _avl_tree_t* cpt_avl_tree, const void* cpv_value);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cpt_avl_tree 指向 avl tree 容器的指针。

cpv_value 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 avl tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
_avl_tree_iterator_t _avl_tree_upper_bound(
    const _avl_tree_t* cpt_avl_tree, const void* cpv_value);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cpt_avl_tree 指向 avl tree 容器的指针。

cpv_value 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 avl tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t _avl_tree_equal_range(
    const _avl_tree_t* cpt_avl_tree, const void* cpv_value);
```

描述:

返回包含指定数据的数据区间。

参数:

cpt_avl_tree 指向 avl tree 容器的指针。
cpv_value 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 avl tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
bool_t _avl_tree_equal(
    const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述:

测试两个 avl tree 容器是否相等。

参数:

cpt_first 指向 avl tree 容器的指针。
cpt_second 指向 avl tree 容器的指针。

返回值:

如果两个 avl tree 容器相等则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 avl tree 必须已经初始化的, 否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同认为这两个容器不等。两个 avl tree 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cpt_first == cpt_second 那么返回 true。

```
bool_t _avl_tree_not_equal(
    const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述:

测试两个 avl tree 容器是否不等。

参数:

cpt_first 指向 avl tree 容器的指针。
cpt_second 指向 avl tree 容器的指针。

返回值:

如果两个 avl tree 容器不等则返回 true, 否则返回 false。

注意：

如果 `cpt_first == NULL` 或者 `cpt_second == NULL` 则函数的行为是未定义的，两个 avl tree 必须已经初始化的，否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同认为两个 avl tree 容器不等。两个 avl tree 容器不等是指容器中的对应的数据不相等，如果对应的数据都相等，那就要看容器中数据的数量是否不等。如果 `cpt_first == cpt_second` 那么返回 `false`。

```
bool_t _avl_tree_less(const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述：

测试第一个 avl tree 是否小于第二个 avl tree。

参数：

`cpt_first` 指向 avl tree 容器的指针。
`cpt_second` 指向 avl tree 容器的指针。

返回值：

如果第一个 avl tree 容器小于第二个 avl tree 容器则返回 `true`，否则返回 `false`。

注意：

如果 `cpt_first == NULL` 或者 `cpt_second == NULL` 则函数的行为是未定义的，两个 avl tree 必须已经初始化的，否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 avl tree 中的数据小于第二个 avl tree 中对应的数据则返回 `true`，如果大于则返回 `false`，当两个 avl tree 中的对应数据相等，第一个 avl tree 中的数据数量小于第二个 avl tree 中数据的数量返回 `true`，否则返回 `false`。如果 `cpt_first == cpt_second` 那么返回 `false`。

```
bool_t _avl_tree_less_equal(  
    const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述：

测试第一个 avl tree 是否小于等于第二个 avl tree。

参数：

`cpt_first` 指向 avl tree 容器的指针。
`cpt_second` 指向 avl tree 容器的指针。

返回值：

如果第一个 avl tree 容器小于等于第二个 avl tree 容器则返回 `true`，否则返回 `false`。

注意：

如果 `cpt_first == NULL` 或者 `cpt_second == NULL` 则函数的行为是未定义的，两个 avl tree 必须已经初始化的，否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同函数的行为是未定义的。如果 `cpt_first == cpt_second` 那么返回 `true`。

```
bool_t _avl_tree_greater(  
    const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述：

测试第一个 avl tree 是否大于第二个 avl tree。

参数：

`cpt_first` 指向 avl tree 容器的指针。
`cpt_second` 指向 avl tree 容器的指针。

返回值:

如果第一个 avl tree 容器大于第二个 avl tree 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 avl tree 必须已经初始化的, 否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同函数的行为是未定义的。如果第一 avl tree 中的数据大于第二个 avl tree 中的对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 avl tree 中数据的数量大于第二个 avl tree 中数据的数量的时候返回 true 否则返回 false。如果 cpdeqt_first == cpt_second 那么返回 false。

```
bool_t _avl_tree_greater_equal(
    const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述:

测试第一个 avl tree 是否大于等于第二个 avl tree。

参数:

cpt_first 指向 avl tree 容器的指针。
cpt_second 指向 avl tree 容器的指针。

返回值:

如果第一个 avl tree 容器大于等于第二个 avl tree 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 avl tree 必须已经初始化的, 否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同函数的行为是未定义的。如果 cpt_first == cpt_second 那么返回 true。

```
void _avl_tree_swap(_avl_tree_t* pt_first, _avl_tree_t* pt_second);
```

描述:

交换两个 avl tree 容器。

参数:

pt_first 指向 avl tree 容器的指针。
pt_second 指向 avl tree 容器的指针。

返回值:

无。

注意:

如果 pt_first == NULL 或者 pt_second == NULL 则函数的行为是未定义的, 两个 avl tree 必须已经初始化的, 否则函数的行为是未定义的。两个 avl tree 容器中保存的数据类型不同函数的行为是未定义的。如果 _avl_tree_equal(pt_first, pt_second), 函数不执行任何动作。

```
_avl_tree_iterator_t _avl_tree_insert_unique(
    _avl_tree_t* pt_avl_tree, const void* cpv_value);
```

描述:

向 avl tree 容器中插入唯一的数据。

参数:

pt_avl_tree 指向 avl tree 容器的指针。

cpv_value 要查找的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_avl_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的，avl tree 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 avl tree 容器中的数据类型相同，否则函数的行为是未定义的。

```
_avl_tree_iterator_t _avl_tree_insert_equal(
    _avl_tree_t* pt_avl_tree, const void* cpv_value);
```

描述:

向 avl tree 容器中插入数据，数据可以重复。

参数:

pt_avl_tree 指向 avl tree 容器的指针。
cpv_value 要查找的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_avl_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的，avl tree 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 avl tree 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _avl_tree_insert_unique_range(
    _avl_tree_t* pt_avl_tree,
    _avl_tree_iterator_t it_begin, _avl_tree_iterator_t it_end);
```

描述:

向 avl tree 中插入指定的数据区间，数据必须唯一。

参数:

pt_avl_tree 指向 avl tree 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pt_avl_tree == NULL 则函数的行为是未定义的，avl tree 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间并且与保存在 avl tree 容器中的数据类型相同，否则函数的行为是未定义的 [it_begin, it_end)必须不属于 pt_avl_tree，否则函数的行为是未定义的。

```
void _avl_tree_insert_equal_range(
    _avl_tree_t* pt_avl_tree,
    _avl_tree_iterator_t it_begin, _avl_tree_iterator_t it_end);
```

描述:

向 avl tree 中插入指定的数据区间。

参数:

pt_avl_tree 指向 avl tree 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pt_avl_tree == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间并且与保存在 avl tree 容器中的数据类型相同, 否则函数的行为是未定义的 [it_begin, it_end)必须不属于 pt_avl_tree, 否则函数的行为是未定义的。

```
void _avl_tree_erase_pos(_avl_tree_t* pt_avl_tree, _avl_tree_iterator_t it_pos);
```

描述:

删除 avl tree 容器中指定位置的数据。

参数:

pt_avl_tree 指向 avl tree 容器的指针。
it_pos 被删除的数据的位置。

返回值:

无。

注意:

如果 pt_avl_tree == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 是属于 pt_avl_tree 容器的有效的迭代器, 否则函数的行为是未定义的。

```
void _avl_tree_erase_range(_avl_tree_t* pt_avl_tree,  
    _avl_tree_iterator_t it_begin, _avl_tree_iterator_t it_end);
```

描述:

删除 avl tree 中指定数据区间的数据。

参数:

pt_avl_tree 指向 avl tree 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pt_avl_tree == NULL 则函数的行为是未定义的, avl tree 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end)必须属于 pt_avl_tree, 否则函数的行为是未定义的。

```
void _avl_tree_erase(_avl_tree_t* pt_avl_tree, const void* cpv_value);
```

描述:

删除 avl tree 中指定的数据。

参数:

pt_avl_tree 指向 avl tree 容器的指针。
cpv_value 要删除的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 `pt_avl_tree == NULL` 或者 `cpv_value == NULL` 则函数的行为是未定义的, `avl tree` 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 `avl tree` 容器中的数据类型相同, 否则函数的行为是未定义的。

第五节 迭代器接口

`avl tree` 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

<code>_create_avl_tree_iterator</code>	创建 <code>avl tree</code> 迭代器。
<code>_avl_tree_iterator_get_value</code>	获得 <code>avl tree</code> 迭代器引用的数据。
<code>_avl_tree_iterator_get_pointer</code>	获得 <code>avl tree</code> 迭代器引用的数据的指针。
<code>_avl_tree_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_avl_tree_iterator_prev</code>	获得引用前一个数据的迭代器。
<code>_avl_tree_iterator_equal</code>	测试两个迭代器是否相等。
<code>_avl_tree_iterator_distance</code>	计算两个迭代器之间的距离。
<code>_avl_tree_iterator_before</code>	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
_avl_tree_iterator_t _create_avl_tree_iterator(void);
```

描述:

创建一个 `avl tree` 迭代器。

参数:

无。

返回值:

`avl tree` 迭代器。

注意:

返回的 `avl tree` 迭代器并不是有效的迭代器, 它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作, 返回的迭代器供其他接口使用。

```
void _avl_tree_iterator_get_value(_avl_tree_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

`it_iter` `avl tree` 迭代器。

`pv_value` 保存数据的缓冲区。

返回值:

无。

注意:

`it_iter` 是有效的 avl tree 迭代器, 否则函数的行为是未定义的。`pv_value == NULL` 则函数的行为是未定义的, `pv_value` 是能够保存下 `it_iter` 引用的数据的缓冲区, 否则函数的行为是未定义的。函数执行后 `it_iter` 引用的数据被拷贝到 `pv_value` 指向的缓存区中。

```
const void* _avl_tree_iterator_get_pointer(_avl_tree_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

`it_iter` avl tree 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

`it_iter` 是有效的 avl tree 迭代器, 否则函数的行为是未定义的。

```
_avl_tree_iterator_t _avl_tree_iterator_next(_avl_tree_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

`it_iter` avl tree 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

`it_iter` 是有效的 avl tree 迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 avl tree 有效的迭代器, 否则函数的行为是未定义的。

```
_avl_tree_iterator_t _avl_tree_iterator_prev(_avl_tree_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

`it_iter` avl tree 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

`it_iter` 是有效的 avl tree 迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 avl tree 有效的迭代器, 否则函数的行为是未定义的。

```
bool_t _avl_tree_iterator_equal(
    _avl_tree_iterator_t it_first, _avl_tree_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first avl tree 迭代器。

it_second avl tree 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 avl tree 容器的有效的 avl tree 迭代器, 否则函数的行为是未定义的。两个 avl tree 迭代器相等是指两个迭代器引用相同的数据。

```
int _avl_tree_iterator_distance(
    _avl_tree_iterator_t it_first, _avl_tree_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_iterfirst avl tree 迭代器。

it_itersecond avl tree 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 avl tree 容器的有效的 avl tree 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为 0。

```
bool_t _avl_tree_iterator_before(
    _avl_tree_iterator_t it_first, _avl_tree_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first avl tree 迭代器。

it_second avl tree 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 avl tree 容器的有效的 avl tree 迭代器, 否则函数的行为是未定义的。

第六节 内部和辅助接口

avl tree 也只是提供了简单的内部接口。

<code>_create_avl_tree_auxiliary</code>	创建 avl tree 容器的辅助函数。
<code>_avl_tree_destroy_auxiliary</code>	销毁 avl tree 容器的辅助函数。

```
bool_t _create_avl_tree_auxiliary(_avl_tree_t* pt_avl_tree, const char* s_typename);
```

描述:

创建一个 avl tree 容器的辅助函数。

参数:

<code>pt_avl_tree</code>	没有创建的 avl tree 容器。
<code>s_typename</code>	avl tree 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pt_avl_tree == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型, `libcstl` 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _avl_tree_destroy_auxiliary(_avl_tree_t* pt_avl_tree);
```

描述:

销毁 avl tree 容器的辅助函数。

参数:

<code>pt_avl_tree</code>	avl tree 容器。
--------------------------	--------------

返回值:

无。

注意:

如果 `pt_avl_tree == NULL` 或者 avl tree 不是使用 `_create_avl_tree` 生成的则函数的行为是未定义的。

avl tree 提供了大量的辅助函数, 这些函数用于检测结构的有效性, 对树进行调整等等。

<code>_avl_tree_is_created</code>	测试 avl tree 容器是否是由 <code>_create_avl_tree</code> 函数创建的。
<code>_avl_tree_is_inited</code>	测试 avl tree 容器是否已经被初始化。
<code>_avl_tree_iterator_belong_to_avl_tree</code>	测试一个迭代器是否属于指定的 avl tree 容器。
<code>_avl_tree_same_avl_tree_iterator_type</code>	测试 avl tree 容器中保存的数据类型和迭代器引用的数据类型是否相等。
<code>_avl_tree_avlnode_belong_to_avl_tree</code>	测试一个 avl node 是否属于指定的 avl tree 容器。
<code>_avl_tree_same_type</code>	测试两个 avl tree 保存的数据类型是否相同。
<code>_avl_tree_same_type_ex</code>	测试两个 avl tree 保存的数据类型以及比较函数是否相同。
<code>_avl_tree_find_value</code>	在 avl tree 中查找指定的值所在的节点。
<code>_avl_tree_destroy_subtree</code>	销毁 sub avl tree。
<code>_avl_tree_left_signal_rotate</code>	左侧单旋转。
<code>_avl_tree_right_signal_rotate</code>	右侧单旋转。

<code>_avl_tree_left_double_rotate</code>	左侧双旋转。
<code>_avl_tree_right_double_rotate</code>	右侧双旋转。
<code>_avl_tree_insert_avlnode</code>	向 avl tree 中插保存指定值的 avl node。
<code>_avl_tree_get_height</code>	获得 avl tree 的高度。
<code>_avl_tree_get_min_avlnode</code>	获得 avl tree 中值最小的节点。
<code>_avl_tree_get_max_avlnode</code>	获得 avl tree 中值最大的节点。
<code>_avl_tree_rebalance</code>	重新调整 avl tree，使它恢复平衡。
<code>_avl_tree_init_elem_auxiliary</code>	初始化数据的辅助函数。
<code>_avl_tree_elem_compare_auxiliary</code>	比较数据的辅助函数。

函数原型

```
bool_t _avl_tree_is_created(const _avl_tree_t* cpt_avl_tree);
```

描述:

测试一个 avl_tree 是否是使用 create_avl_tree 创建的。

参数:

`cpt_avl_tree` avl_tree 容器。

返回值:

如果 avl_tree 是使用 create_avl_tree 创建的则返回 true，否则返回 false。

注意:

如果 `cpt_avl_tree == NULL`，函数的行为是未定义的。

```
bool_t _avl_tree_is_inited(const _avl_tree_t* cpt_avl_tree);
```

描述:

测试一个 avl_tree 是否已经初始化。

参数:

`cpt_avl_tree` avl_tree 容器。

返回值:

如果 avl_tree 已经初始化了，返回 true，否则返回 false。

注意:

如果 `cpt_avl_tree == NULL`，函数的行为是未定义的。

```
bool_t _avl_tree_iterator_belong_to_avl_tree(
    const _avl_tree_t* cpt_avl_tree, _avl_tree_iterator_t it_iter);
```

描述:

测试一个迭代器是否属于指定的 avl_tree 容器。

参数:

`cpt_avl_tree` avl tree 容器。

`it_iter` avl tree 迭代器。

返回值:

如果迭代器引用的数据属于 avl tree 容器的有效范围内，返回 true，否则返回 false。

注意：

如果 cpt_avl_tree == NULL，那么函数的行为是未定义的，cpt_avl_tree 必须是已经初始化的，否则函数的行为是未定义的。如果 it_iter 不是 avl tree 迭代器，那么函数的行为是未定义的。

```
bool_t _avl_tree_same_avl_tree_iterator_type(
    const _avl_tree_t* cpt_avl_tree, _avl_tree_iterator_t it_iter);
```

描述：

测试 avl tree 容器中保存的数据类型和迭代器引用的数据类型是否相等。

参数：

cpt_avl_tree avl tree 容器。
it_iter avl tree 迭代器。

返回值：

如果 avl tree 中保存的数据类型与 it_iter 引用的数据类型相同返回 true，否则返回 false。

注意：

如果 cpt_avl_tree == NULL 或者 it_iter 不是 _avl_tree_iterator_t 类型，那么函数的行为是未定义的。cpt_avl_tree 必须是使用 _create_avl_tree() 创建的，否则函数的行为是未定义的。

```
bool_t _avl_tree_avlnode_belong_to_avl_tree(
    const _avlnode_t* cpt_root, const _avlnode_t cpt_pos);
```

描述：

测试一个 avl node 是否属于指定的 avl tree 容器。

参数：

cpt_root avl tree 子树的根。
cpt_pos 要查找的 avl node。

返回值：

如果要查找的 avl node 在 avl tree 子树内，返回 true，否则返回 false。

注意：

如果 cpt_root == NULL 或者 cpt_pos == NULL 则返回 false。

```
bool_t _avl_tree_same_type(
    const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述：

判断两个 avl tree 容器中保存的数据类型是否相同。

参数：

cpt_first 第一个 avl tree 容器。
cpt_second 第二个 avl tree 容器。

返回值：

如果两个 avl tree 中保存的数据类型相同返回 true，否则返回 false。

注意：

如果 cpt_first == NULL 或者 cpt_second == NULL，那么函数的行为是未定义的。两个 avl tree 容器必须是已经初始化或者使用 _create_avl_tree 创建的 avl tree 容器，否则函数的行为是未定义的。如果 cpt_first == cpt_second 则返回 true。

```
bool_t _avl_tree_same_type_ex(
    const _avl_tree_t* cpt_first, const _avl_tree_t* cpt_second);
```

描述:

判断两个 avl tree 容器中保存的数据类型以及比较函数是否相同。

参数:

cpt_first 第一个 avl tree 容器。
cpt_second 第二个 avl tree 容器。

返回值:

如果两个 avl tree 中保存的数据类型以及比较函数相同返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL, 那么函数的行为是未定义的。两个 avl tree 容器必须是已经初始化或者使用_create_avl_tree 创建的 avl tree 容器, 否则函数的行为是未定义的。如果 cpt_first == cpt_second 则返回 true。

```
_avlnode_t* _avl_tree_find_value(
    const _avl_tree_t* cpt_avl_tree, const _avlnode_t* cpt_root,
    const void* cpv_value);
```

描述:

在 avl tree 中查找指定的值所在的节点。

参数:

cpt_val_tree avl tree 容器。
cpt_root 子树的根节点。
cpv_value 查找的数据。

返回值:

如果在 avl tree 子树中找到了相应的数据, 则返回保存该数据的 avlnode 指针, 否则返回 NULL。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_value == NULL, 那么函数的行为是未定义的。avl tree 容器必须是已经初始化的 avl tree 容器, 否则函数的行为是未定义的。

```
_avlnode_t* _avl_tree_destroy_subtree(
    _avl_tree_t* pt_avl_tree, _avlnode_t* pt_root);
```

描述:

销毁 sub avl tree。

参数:

pt_val_tree avl tree 容器。
pt_root 子树的根节点。

返回值:

返回 NULL。

注意:

如果 pt_avl_tree == NULL, 则函数的行为是未定义的。avl tree 容器必须是已经初始化或者使用_create_avl_tree 创建的 avl tree 容器, 否则函数的行为是未定义的。

```
_avlnode_t* _avl_tree_left_signal_rotate(_avlnode_t* pt_root);
```

描述:

左侧单旋转。

参数:

pt_root 子树的根节点。

返回值:

旋转之后新的子树根节点。

注意:

如果 pt_root == NULL 或者 pt_root->_pt_left == NULL 则函数的行为是未定义的。

```
_avlnode_t* _avl_tree_right_signal_rotate(_avlnode_t* pt_root);
```

描述:

右侧单旋转。

参数:

pt_root 子树的根节点。

返回值:

旋转之后新的子树根节点。

注意:

如果 pt_root == NULL 或者 pt_root->_pt_right == NULL 则函数的行为是未定义的。

```
_avlnode_t* _avl_tree_left_double_rotate(_avlnode_t* pt_root);
```

描述:

左侧双旋转。

参数:

pt_root 子树的根节点。

返回值:

旋转之后新的子树根节点。

注意:

如果 pt_root == NULL 则函数的行为是未定义的。

```
_avlnode_t* _avl_tree_right_double_rotate(_avlnode_t* pt_root);
```

描述:

右侧双旋转。

参数:

pt_root 子树的根节点。

返回值:

旋转之后新的子树根节点。

注意:

如果 pt_root == NULL 则函数的行为是未定义的。

```
_avl_tree_insert_result_t _avl_tree_insert_avlnode(
    const _avl_tree_t* cpt_avl_tree, _avlnode_t* pt_root, const void* cpv_value);
```

描述:

向 avl tree 中插保存指定值的 avl node。

参数:

cpt_val_tree	avl tree 容器。
cpt_root	子树的根节点。
cpv_value	查找的数据。

返回值:

返回插入节点后的新的根和新节点。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_value == NULL, 则函数的行为是未定义的。avl tree 容器必须是已经初始化, 否则函数的行为是未定义的。

```
int _avl_tree_get_height(const _avlnode_t* cpt_root);
```

描述:

获得 avl tree 的高度。

参数:

cpt_root	子树的根节点。
----------	---------

返回值:

返回节点的高度。

注意:

如果 cpt_root == NULL 则返回-1。

```
_avlnode_t* _avl_tree_get_min_avlnode(const _avlnode_t* cpt_root);
```

描述:

获得 avl tree 中值最小的节点。

参数:

cpt_root	子树的根节点。
----------	---------

返回值:

值最小的节点。

注意:

如果 cpt_root == NULL 则函数的行为是未定义的。

```
_avlnode_t* _avl_tree_get_max_avlnode(const _avlnode_t* cpt_root);
```

描述:

获得 avl tree 中值最大的节点。

参数:

cpt_root	子树的根节点。
----------	---------

返回值:

值最大的节点。

注意:

如果 cpt_root == NULL 则函数的行为是未定义的。

```
_avlnode_t* _avl_tree_rebalance(_avlnode_t* pt_root);
```

描述:

重新调整 avl tree，使它恢复平衡。

参数:

pt_root 子树的根节点。

返回值:

重新调整后的根节点。

注意:

无。

```
void _avl_tree_init_elem_auxiliary(_avl_tree_t* pt_avl_tree, _avlnode_t* pt_node);
```

描述:

初始化数据的辅助函数。

参数:

pt_avl_tree avl tree 容器。

pt_node 初始化节点。

返回值:

无。

注意:

如果 pt_avl_tree == NULL 或者 pt_node == NULL，那么函数的行为是未定义的。pt_avl_tree 必须是初始化的或者是使用_create_avl_tree()创建的，否则函数的行为是未定义的。

```
void _avl_tree_elem_compare_auxiliary(const _avl_tree_t* cpt_avl_tree,
    const void* cpv_first, const void* cpv_second, void* pv_output);
```

描述:

比较数据的辅助函数。

参数:

cpt_avl_tree avl tree 容器。

cpv_first 第一个数据。

cpv_second 第二个数据。

pv_output 比较结果。

返回值:

无。

注意:

如果 cpt_avl_tree == NULL 或者 cpv_first == NULL 或者 cpv_second == NULL 或者 pv_output == NULL，那么函数的行为是未定义的。cpt_avl_tree 必须是初始化的或者是使用_create_avl_tree()创建的，否则函数的行为是未定义的。

第十五章 红黑树

rb tree 也是一种平衡的二叉搜索树，与 avl tree 相比 rb tree 的在插入和删除上速度更快，但是在查找方面还是 avl tree 效率更高些。造成这种情况的原因主要是 avl tree 是高度平衡的，而 rb tree 是部分平衡的，它使用更少的旋转次数来获得更高的插入和删除效率，但是降低了查找效率。

第一节 红黑树的机制

rb tree 也是平衡二叉搜索树中的一种，所以它也要符合平衡二叉搜索树的规则。它与 avl tree 相比只是平衡条件不同，avl tree 使用高度差来作为平衡条件，但是 rb tree 使用节点的颜色来作为平衡条件。

rb tree 必须满足一下这些条件：

- 1) 树的节点要么是红色，要么是黑色。
- 2) 树的根是黑色的。
- 3) 一个空节点(NULL 节点)是黑色的。
- 4) 一个红色节点的子节点必须是黑色的。
- 5) 对于每个节点，从该节点到其子孙节点的所有路径上包含的黑色节点的数目是相同的。

《算法导论》中详细的讨论了 rb tree。

rb tree 主要是根据节点的颜色来保持平衡，当插入新的节点或者删除节点的时候，这个平衡条件可能就会被破坏，那么就需要特定的插入和删除节点的算法来保持插入和删除节点之后的树仍然是平衡的。

当插入一个新的节点的时候：

- 1) 如果当前插入的节点是根节点：

根据规则 2，这个根节点应该是黑色，并且插入后符合 rb tree 的平衡条件。

- 2) 当插入的节点不是根节点的时候，那么为了保持原有 rb tree 的性质就不能够添加新的黑节点，所以插入的节点应该是红色的，但是这样虽然满足了条件 5，但是会破坏其他的条件，这时候就需要调整：

情况 1) 当叔叔节点为红色的时候：

这时候祖父节点一定为黑色，要不然在没有插入之前就已经违反 rb tree 的规则了。这样，只要将祖父节点的黑色降到父节点这一层，然后将祖父节点认为是新插入的红色节点。

情况 2) 当叔叔节点为黑色的时候，并且插入的节点是父节点的左孩子：

这时候只需要将节点进行一次单旋转，这种情况可能发生在插入时，这是 C 没有右孩子，只有一个红色的左孩子 B，然后当 A 插入的时候，并且 A 比 B 小，所以就构成了这样的情况(C 没有右孩子，但是 NULL 节点被视为黑色节点)。当 C 有右孩子并且为黑色的时候，A 的下层肯定含有黑色的字节点，当前的情况是经过转换得到的一种中间状态。这种情况下左右子树的黑色节点是相同的，只是在左子树中包含了连续的红色节点，只要将红色节点隔开就可以解决了。

情况 3) 当叔叔节点为黑色的时候，并且插入的节点是父节点的右孩子：

情况 3 与情况 2 是相同的，只是插入的红色孩子是右孩子，并且通过一次的旋转不能够像情况 2 那样的解决问题。但是我们可以学习 avl tree 将情况 3 通过旋转转换成情况 2。

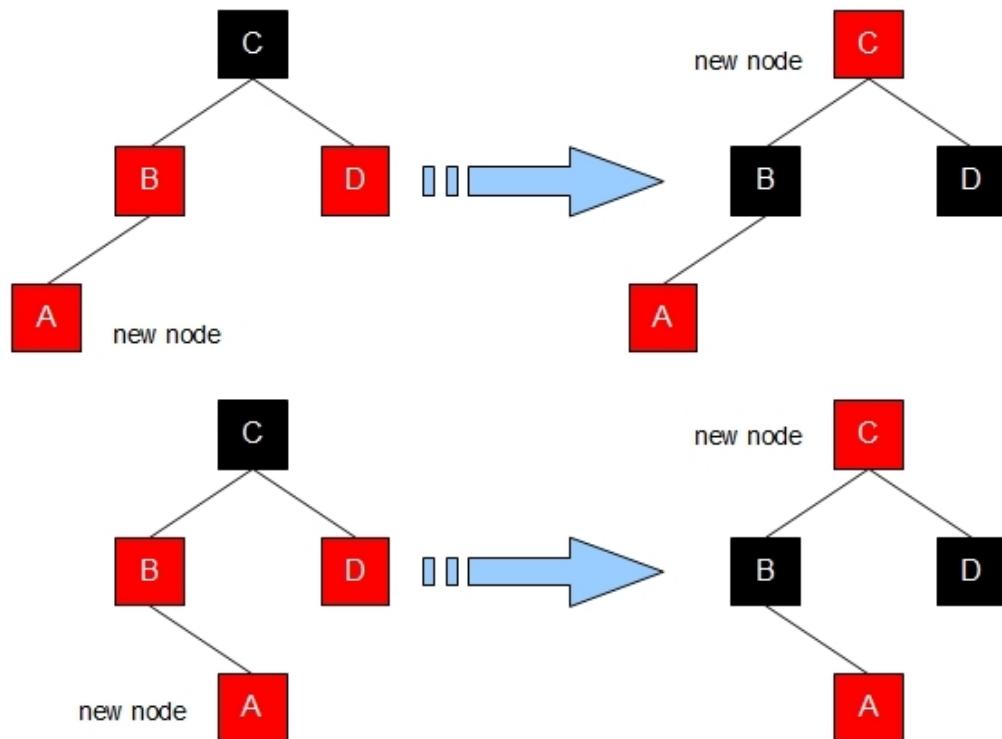


图 15.1 情况 1 叔叔节点为红色

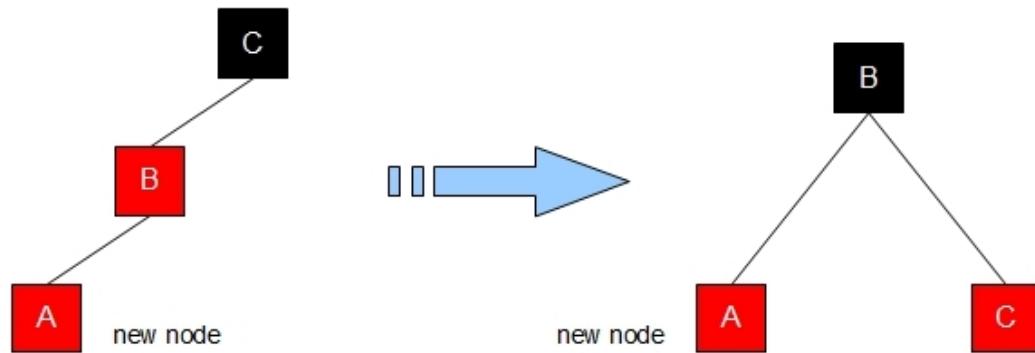


图 15.2 情况 2 叔叔节点为黑色的时候，并且插入的节点是父节点的左孩子

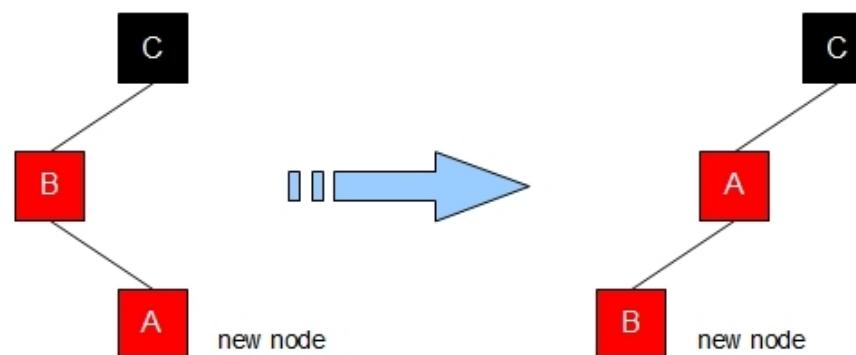


图 15.3 情况 3 叔叔节点为黑色的时候，并且插入的节点是父节点的右孩子

rb tree 的删除操作比插入操作来复杂的多，既要遵守二叉查找树的删除规则，又不能够破坏 rb tree 的规则。当删除 rb tree 中的节点的时候，首先按照二叉查找树中的删除规则删除掉指定的节点，并且使用右子树中的最小节点代替原来被删除的节点的位置，并且将这个节点的颜色变成原来被删除节点的颜色，这样在被删除的位置上就不会破坏 rb tree 的规则。再来看右子树的最小节点，实际上删除的相等于这个节点，这个节点的颜色决定了删除的结果。当这个节点的颜色是红色的时候，因为删除操作并没有改变黑色节点的数目，那么也不会影响 rb tree 的性质，所以直接删除就完成了操作，但是如果这个节点的颜色是黑色，那么就需要调整。

当一个黑色节点被删除之后，实际的黑色节点的数目就减少了，那么就相等于将这个节点的黑色推到了它的子节点上，如果子节点原来是黑色的，现在就变成了双重黑色，如果是红色的，现在就变成了红黑了。当子节点是红黑的时候，将原来子节点的红色变成黑色，就完成了调整，因为红色的数目不会影响 rb tree 的性质。当子节点是根节点的时候，调整也完成了，因为 rb tree 的根是多少层黑色都是没有关系的。

那么需要调整的情况就是当子节点是黑色并且不是根的情况：（下面列出的情况顺序与《算法导论》中出现的顺序不一致， x 表示具有双重黑色的节点，使用灰色节点表示颜色不重要的节点）

情况 4) x 的兄弟节点 w 是黑色的，且 w 的右子节点是红色的

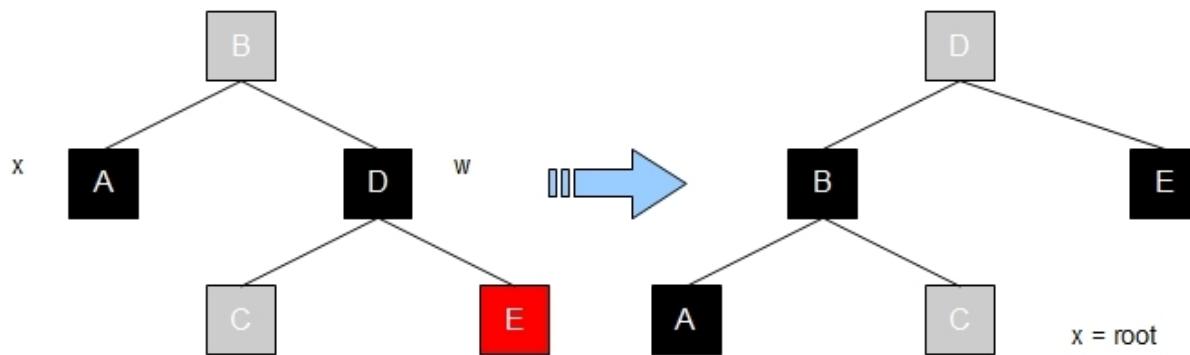


图 15.4 删节的第四种情况

经过一次左旋并且改变颜色，就可以去掉 x 上面的双重黑色。如图，左子树中 A 具有双重黑色，旋转并染色之后，左子树中 A 的一层黑色分给了 B ，右子树中用原来红色的 E 代替了原来黑色的 D ，这样左右子树的黑色就平衡了，将 $root$ 作为新的 x 从而结束调整。

除了情况 4 可以直接结束调整过程外，其他的情况都不能直接结束调整过程，但是情况 2 是将多出的一层黑色向树根方向推的过程。

情况 2) x 的兄弟 w 是黑色的，且 w 的两个子节点是黑色的

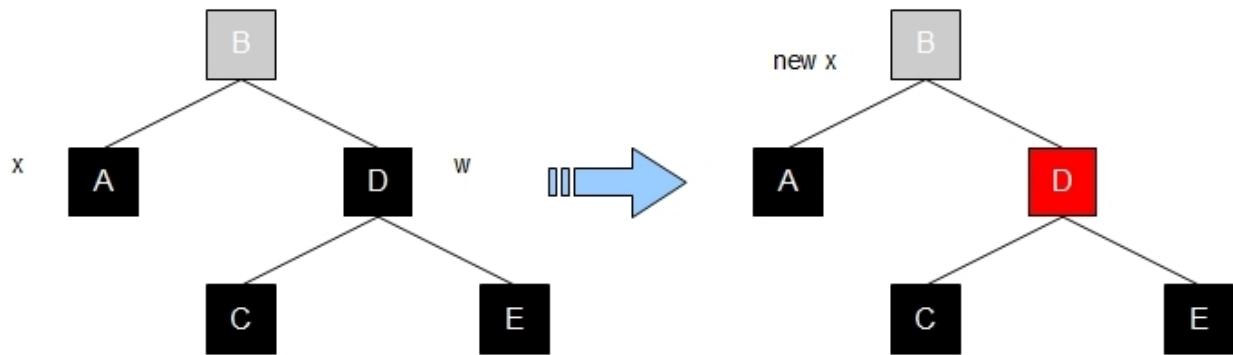


图 15.5 删节的第二种情况

这种情况不用选装，只是将 x 和兄弟节点 w 的黑色推到父节点中， x 依然保持黑色， w 失去黑色之后就变成了红色，这样新的 x 就是父节点。当父节点原来是红色的时候，或者父节点是根的时候，调整就借宿了。

情况 1 和情况 3 本身是不能够解决这个问题的，但是它们可以通过调整来转换成情况 2 或者情况 4。

情况 1) x 的兄弟节点 w 是红色的

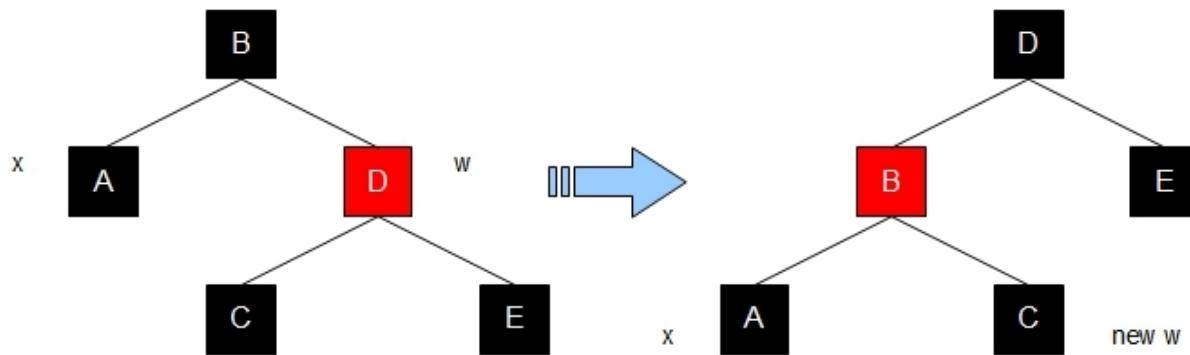


图 15.6 删除节点的第一种情况

经过旋转和染色之后，C 成为了新的 w，这样就将第一种情况转换成了其他的情况了。

情况 3) x 的兄弟节点 w 是黑色的，w 的左子节点是红色的，右子节点是黑色的

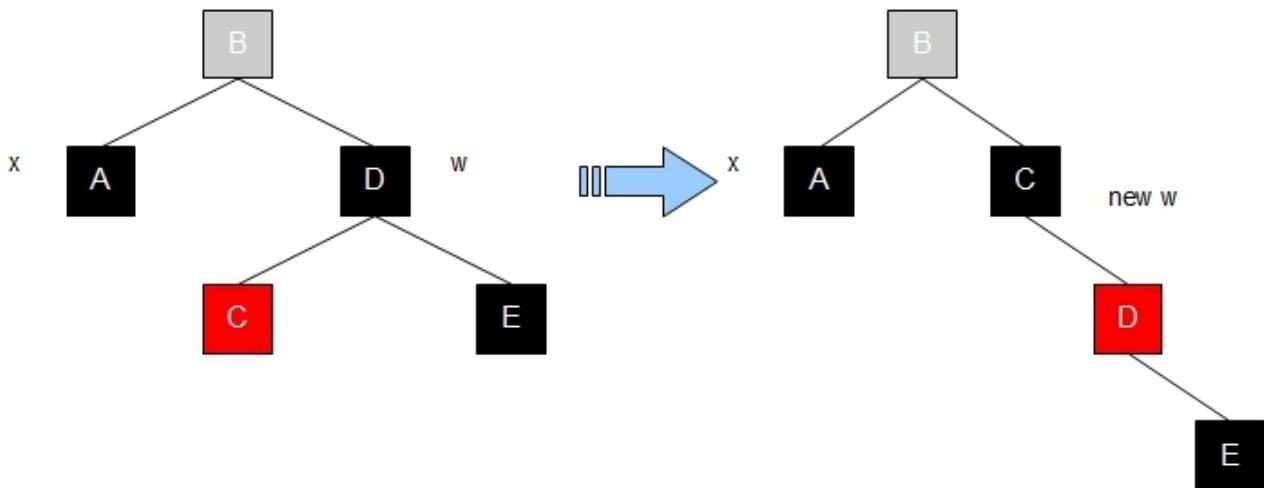


图 15.7 删除节点的第三种情况

经过旋转和染色之后，C 成为了新的 w，这样就将第三种情况转换成了其他的情况了。

第二节 红黑树迭代器

rb tree 的迭代器结构和 avl tree 的迭代器结构是一样的。代码结构也是完全相同的。

第三节 红黑树代码结构

rb tree 节点中包含很多信息：

```
typedef struct _tagrbnode
```

```
{
    struct _tagrbnode* _pt_parent;
    struct _tagrbnode* _pt_left;
    struct _tagrbnode* _pt_right;
    color_t           _t_color;
    char              _pc_data[1];
}avlnode_t;
```

其中 _t_color 表示当前节点的颜色。

```
typedef enum _tagcolor
{
    red, black
}color_t;
```

下面的结构表示 rb tree:

```
typedef struct _tagrbtree
{
    _typeinfo_t      _t_typeinfo;
    _alloc_t         _t_allocator;
    rbnodes_t        _t_rbroot;
    size_t           _t_nodecount;
    binary_function_t _t_compare;
}avl_tree_t;
```

结构中的 _t_rbroot 中 _pt_parent 指向实际 rb tree 的根， _pt_left 指向 rb tree 中值最小的数据， _pt_right 指向 rb tree 中值最大的数据。

第四节 外部接口

rb tree 提供的外部接口是为了实现关联容器。

_create_rb_tree	创建 rb tree 容器。
_rb_tree_init	初始化 rb tree 容器。
_rb_tree_init_copy	使用存在的 rb tree 容器进行初始化。
_rb_tree_init_copy_range	使用指定的数据区间进行初始化。
_rb_tree_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
_rb_tree_destroy	销毁 rb tree 容器。
_rb_tree_assign	使用存在的 rb tree 容器赋值。
_rb_tree_size	返回 rb tree 容器中数据的数量。
_rb_tree_empty	判断 rb tree 容器是否为空。

<code>_rb_tree_max_size</code>	返回 rb tree 容器中能够保存数据的最大数量。
<code>_rb_tree_begin</code>	返回指向 rb tree 容器的第一个数据的迭代器。
<code>_rb_tree_end</code>	返回指向 rb tree 容器末尾的迭代器。
<code>_rb_tree_key_comp</code>	返回 rb tree 容器中的数据比较规则。
<code>_rb_tree_find</code>	在 rb tree 容器中查找指定的数据。
<code>_rb_tree_clear</code>	清空 rb tree 容器。
<code>_rb_tree_count</code>	返回 rb tree 容器中指定数据的数量。
<code>_rb_tree_lower_bound</code>	返回指向第一个大于等于指定数据的迭代器。
<code>_rb_tree_upper_bound</code>	返回指向第一个大于指定数据的迭代器。
<code>_rb_tree_equal_range</code>	返回包含指定数据的数据区间。
<code>_rb_tree_equal</code>	测试两个 rb tree 容器是否相等。
<code>_rb_tree_not_equal</code>	测试两个 rb tree 容器是否不等。
<code>_rb_tree_less</code>	测试第一个 rb tree 容器是否小于第二个 rb tree 容器。
<code>_rb_tree_less_equal</code>	测试第一个 rb tree 容器是否小于等于第二个 rb tree 容器。
<code>_rb_tree_greater</code>	测试第一个 rb tree 容器是否大于第二个 rb tree 容器。
<code>_rb_tree_greater_equal</code>	测试第一个 rb tree 容器是否大于等于第二个 rb tree 容器。
<code>_rb_tree_swap</code>	交换两个 rb tree 容器。
<code>_rb_tree_insert_unique</code>	向 rb tree 中插入数据，数据必须唯一。
<code>_rb_tree_insert_equal</code>	向 rb tree 中插入数据，数据可以重复。
<code>_rb_tree_insert_unique_range</code>	向 rb tree 中插入指定的数据区间，数据必须唯一。
<code>_rb_tree_insert_equal_range</code>	向 rb tree 中插入指定的数据区间，数据可以重复。
<code>_rb_tree_erase_pos</code>	删除 rb tree 中指定位置的数据。
<code>_rb_tree_erase_range</code>	删除 rb tree 中指定数据区间的数据。
<code>_rb_tree_erase</code>	删除 rb tree 中指定的数据。

函数原型

```
_rb_tree_t* _create_rb_tree(const char* s_typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 rb tree 容器的指针，否则返回 NULL。

注意:

`s_typename` != NULL, 否则函数的行为是未定义的。

```
void _rb_tree_init(_rb_tree_t* pt_rb_tree, binary_function_t t_compare);
```

描述:

初始化 rb tree 迭代器。

参数:

pt_rb_tree	rb tree 容器。
t_compare	数据比较规则。

返回值:

无。

注意:

`pt_rb_tree` == NULL 则函数的行为是未定义的, `pt_rb_tree` 必须是使用`_create_rb_tree()`创建的, 否则函数的行为是未定义的, 如果 `t_compare` == NULL 则使用类型默认的比较函数。

```
void _rb_tree_init_copy(_rb_tree_t* pt_dest, const _rb_tree_t* cpt_src);
```

描述:

使用已经存在的 rb tree 初始化 rb tree 迭代器。

参数:

pt_dest	目的 rb tree 容器。
cpt_src	源 rb tree 容器。

返回值:

无。

注意:

`pt_dest` == NULL 或者 `cpt_src` == NULL 则函数的行为是未定义的, `pt_dest` 必须是使用`_create_rb_tree()`创建的, `cpt_src` 必须是已经初始化, 否则函数的行为是未定义的。`pt_dest` 和 `cpt_src` 保存的数据类型必须相同, 否则函数的行为是未定义。

```
void _rb_tree_init_copy_range(
    _rb_tree_t* pt_dest,
    _rb_tree_iterator_t it_begin, _rb_tree_iterator_t it_end);
```

描述:

使用指定的数据区间初始化 rb tree 迭代器。

参数:

pt_dest	目的 rb tree 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。

返回值:

无。

注意:

如果 `pt_dest` == NULL 则函数的行为是未定义的, `pt_dest` 必须是使用`_create_rb_tree()`创建的否则函数的行为是未定义的。`[it_begin, it_end)`属于一个已经初始化的 rb tree 容器的有效的数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void _rb_tree_init_copy_range_ex(
    _rb_tree_t* pt_dest,
    _rb_tree_iterator_t it_begin, _rb_tree_iterator_t it_end,
    binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 rb tree 迭代器。

参数:

pt_dest	目的 rb tree 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。
t_compare	比较函数。

返回值:

无。

注意:

如果 `pt_dest == NULL` 则函数的行为是未定义的, `pt_dest` 必须是使用 `_create_rb_tree()` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 rb tree 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 `t_compare == NULL` 则使用类型默认的比较函数。

```
void _rb_tree_destroy(_rb_tree_t* pt_rb_tree);
```

描述:

销毁创建的 rb tree 容器。

参数:

pt_rb_tree	rb tree 容器。
------------	-------------

返回值:

无。

注意:

如果 `pt_rb_tree == NULL` 则函数的行为是未定义的, `pt_rb_tree` 必须是初始化或者是使用 `_create_rb_tree()` 创建的, 否则函数的行为是未定义的。

```
void _rb_tree_assign(_rb_tree_t* pt_dest, const _rb_tree_t* cpt_src);
```

描述:

使用一个 rb tree 为另一个 rb tree 赋值。

参数:

pt_dest	指向被赋值的 rb tree 容器的指针。
cpt_src	指向赋值的 rb tree 容器的指针。

返回值:

无。

注意:

如果 `pt_dest == NULL` 或者 `cpt_src == NULL` 则函数的行为是未定义的, 两个 rb tree 必须已经初始化的, 否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同函数的行为是未定义的。如果 `_rb_tree_equal(pt_dest,`

cpt_src)那么函数不做任何动作。

```
size_t _rb_tree_size(const _rb_tree_t* cpt_rb_tree);
```

描述:

获得 rb tree 容器中保存的数据的个数。

参数:

cpt_rb_tree rb tree 容器。

返回值:

rb tree 容器中保存的数据的个数。

注意:

如果 cpt_rb_tree == NULL 则函数的行为是未定义的, cpt_rb_tree 必须是已经初始化的 rb tree 容器, 否则函数的行为是未定义的。

```
bool_t _rb_tree_empty(const _rb_tree_t* cpt_rb_tree);
```

描述:

测试 rb tree 容器是否为空。

参数:

cpt_rb_tree rb tree 容器。

返回值:

如果 rb tree 容器为空则返回 true, 否则返回 false。

注意:

如果 cpt_rb_tree == NULL 则函数的行为是未定义的, cpt_rb_tree 必须是已经初始化的 rb tree 容器, 否则函数的行为是未定义的。

```
size_t _rb_tree_max_size(const _rb_tree_t* cpt_rb_tree);
```

描述:

获得 rb tree 容器中数据的最大数量。

参数:

cpt_rb_tree rb tree 容器。

返回值:

rb tree 容器中保存的数据的个数的最大值。

注意:

如果 cpt_rb_tree == NULL 则函数的行为是未定义的, cpt_rb_tree 必须是已经初始化的 rb tree 容器, 否则函数的行为是未定义的。

```
_rb_tree_iterator_t _rb_tree_begin(const _rb_tree_t* cpt_rb_tree);
```

描述:

获得引用 rb tree 容器中第一数据的迭代器。

参数:

cpt_rb_tree 指向 rb tree 容器的指针。

返回值:

返回引用 rb tree 容器中第一个数据的迭代器。

注意:

如果 `cpt_rb_tree == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。如果 rb tree 容器为空, 则返回值与 `_rb_tree_end(cpt_rb_tree)` 相等。

```
_rb_tree_iterator_t _rb_tree_end(const _rb_tree_t* cpt_rb_tree);
```

描述:

获得引用 rb tree 容器末尾的迭代器。

参数:

`cpt_rb_tree` 指向 rb tree 容器的指针。

返回值:

返回引用 rb tree 容器末尾的迭代器。

注意:

如果 `cpt_rb_tree == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t _rb_tree_key_comp(const _rb_tree_t* cpt_rb_tree);
```

描述:

返回 rb tree 容器中的数据比较规则。

参数:

`cpt_rb_tree` 指向 rb tree 容器的指针。

返回值:

返回 rb tree 容器中的数据比较规则。

注意:

如果 `cpt_rb_tree == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。

```
_rb_tree_iterator_t _rb_tree_find(
    const _rb_tree_t* cpt_rb_tree, const void* cpv_value);
```

描述:

在 rb tree 容器中查找指定的数据。

参数:

`cpt_rb_tree` 指向 rb tree 容器的指针。

`cpv_value` 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 `cpt_rb_tree == NULL` 或者 `cpv_value == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 rb tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _rb_tree_clear(_rb_tree_t* pt_rb_tree);
```

描述:

删除 rb tree 容器中的所有数据。

参数:

`pt_rb_tree` 指向 rb tree 容器的指针。

返回值:

无。

注意:

如果 `pt_rb_tree == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。

```
size_t _rb_tree_count(const _rb_tree_t* cpt_rb_tree, const void* cpv_value);
```

描述:

返回 rb tree 容器中指定数据的数量。

参数:

`cpt_rb_tree` 指向 rb tree 容器的指针。

`cpv_value` 要查找的指定数据。

返回值:

返回 rb tree 容器中指定数据的数量。

注意:

如果 `cpt_rb_tree == NULL` 或者 `cpv_value == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 rb tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
_rb_tree_iterator_t _rb_tree_lower_bound(
    const _rb_tree_t* cpt_rb_tree, const void* cpv_value);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

`cpt_rb_tree` 指向 rb tree 容器的指针。

`cpv_value` 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cpt_rb_tree == NULL` 或者 `cpv_value == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 rb tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
_rb_tree_iterator_t _rb_tree_upper_bound(
    const _rb_tree_t* cpt_rb_tree, const void* cpv_value);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

`cpt_rb_tree` 指向 rb tree 容器的指针。

`cpv_value` 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cpt_rb_tree == NULL` 或者 `cpv_value == NULL` 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 rb tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t _rb_tree_equal_range(
    const _rb_tree_t* cpt_rb_tree, const void* cpv_value);
```

描述:

返回包含指定数据的数据区间。

参数:

cpt_rb_tree 指向 rb tree 容器的指针。
cpv_value 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_rb_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的，rb tree 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 rb tree 容器中的数据类型相同，否则函数的行为是未定义的。

```
bool_t _rb_tree_equal(
    const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

测试两个 rb tree 容器是否相等。

参数:

cpt_first 指向 rb tree 容器的指针。
cpt_second 指向 rb tree 容器的指针。

返回值:

如果两个 rb tree 容器相等则返回 true，否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的，两个 rb tree 必须已经初始化的，否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同认为这两个容器不等。两个 rb tree 容器相等是指容器中的数据对应相等，并且容器中数据的数量也相等。如果 cpt_first == cpt_second 那么返回 true。

```
bool_t _rb_tree_not_equal(
    const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

测试两个 rb tree 容器是否不等。

参数:

cpt_first 指向 rb tree 容器的指针。
cpt_second 指向 rb tree 容器的指针。

返回值:

如果两个 rb tree 容器不等则返回 true，否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的，两个 rb tree 必须已经初始化的，否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同认为两个 rb tree 容器不等。两个 rb tree 容器不等是指容器中的对应的数据不相等，如果对应的数据都相等，那就要看容器中数据的数量是否不等。如果 cpt_first == cpt_second 那么返回 false。

```
bool_t _rb_tree_less(const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

测试第一个 rb tree 是否小于第二个 rb tree。

参数:

cpt_first 指向 rb tree 容器的指针。
cpt_second 指向 rb tree 容器的指针。

返回值:

如果第一个 rb tree 容器小于第二个 rb tree 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 rb tree 必须已经初始化的, 否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 rb tree 中的数据小于第二个 rb tree 中对应的数据则返回 true, 如果大于则返回 false, 当两个 rb tree 中的对应数据相等, 第一个 rb tree 中的数据数量小于第二个 rb tree 中数据的数量返回 true, 否则返回 false。如果 cpt_first == cpt_second 那么返回 false。

```
bool_t _rb_tree_less_equal(  
    const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

测试第一个 rb tree 是否小于等于第二个 rb tree。

参数:

cpt_first 指向 rb tree 容器的指针。
cpt_second 指向 rb tree 容器的指针。

返回值:

如果第一个 rb tree 容器小于等于第二个 rb tree 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 rb tree 必须已经初始化的, 否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同函数的行为是未定义的。如果 cpt_first == cpt_second 那么返回 true。

```
bool_t _rb_tree_greater(  
    const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

测试第一个 rb tree 是否大于第二个 rb tree。

参数:

cpt_first 指向 rb tree 容器的指针。
cpt_second 指向 rb tree 容器的指针。

返回值:

如果第一个 rb tree 容器大于第二个 rb tree 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 rb tree 必须已经初始化的, 否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同函数的行为是未定义的。如果第一 rb tree 中的数据大于第二个 rb tree 中的对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 rb tree 中数据的

数量大于第二个 rb tree 中数据的数量的时候返回 true 否则返回 false。如果 cpt_deqt_first == cpt_second 那么返回 false。

```
bool_t _rb_tree_greater_equal(
    const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

测试第一个 rb tree 是否大于等于第二个 rb tree。

参数:

cpt_first 指向 rb tree 容器的指针。
cpt_second 指向 rb tree 容器的指针。

返回值:

如果第一个 rb tree 容器大于等于第二个 rb tree 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 rb tree 必须已经初始化的, 否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同函数的行为是未定义的。如果 cpt_first == cpt_second 那么返回 true。

```
void _rb_tree_swap(_rb_tree_t* pt_first, _rb_tree_t* pt_second);
```

描述:

交换两个 rb tree 容器。

参数:

pt_first 指向 rb tree 容器的指针。
pt_second 指向 rb tree 容器的指针。

返回值:

无。

注意:

如果 pt_first == NULL 或者 pt_second == NULL 则函数的行为是未定义的, 两个 rb tree 必须已经初始化的, 否则函数的行为是未定义的。两个 rb tree 容器中保存的数据类型不同函数的行为是未定义的。如果 _rb_tree_equal(pt_first, pt_second), 函数不执行任何动作。

```
_rb_tree_iterator_t _rb_tree_insert_unique(
    _rb_tree_t* pt_rb_tree, const void* cpv_value);
```

描述:

向 rb tree 容器中插入唯一的数据。

参数:

pt_rb_tree 指向 rb tree 容器的指针。
cpv_value 要插入的指定数据。

返回值:

指向插入的数据的迭代器, 如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_rb_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, rb tree 容器必须已经初始化的, 否则函数的行为是未定义的。插入的数据必须与保存在 rb tree 容器中的数据类型相同, 否则函数的行为是未定义的。

```
_rb_tree_iterator_t _rb_tree_insert_equal(
    _rb_tree_t* pt_rb_tree, const void* cpv_value);
```

描述:

向 rb tree 容器中插入数据，数据可以重复。

参数:

pt_rb_tree	指向 rb tree 容器的指针。
cpv_value	要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 `pt_rb_tree == NULL` 或者 `cpv_value == NULL` 则函数的行为是未定义的，rb tree 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 rb tree 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _rb_tree_insert_unique_range(
    _rb_tree_t* pt_rb_tree,
    _rb_tree_iterator_t it_begin, _rb_tree_iterator_t it_end);
```

描述:

向 rb tree 中插入指定的数据区间，数据必须唯一。

参数:

pt_rb_tree	指向 rb tree 容器的指针。
it_begin	指定的数据区间的开头。
it_end	指定的数据区间的末尾。

返回值:

无。

注意:

如果 `pt_rb_tree == NULL` 则函数的行为是未定义的，rb tree 容器必须已经初始化的，否则函数的行为是未定义的。`[it_begin, it_end)` 必须是有效的数据区间并且与保存在 rb tree 容器中的数据类型相同，否则函数的行为是未定义的。`[it_begin, it_end)` 必须不属于 `pt_rb_tree`，否则函数的行为是未定义的。

```
void _rb_tree_insert_equal_range(
    _rb_tree_t* pt_rb_tree,
    _rb_tree_iterator_t it_begin, _rb_tree_iterator_t it_end);
```

描述:

向 rb tree 中插入指定的数据区间。

参数:

pt_rb_tree	指向 rb tree 容器的指针。
it_begin	指定的数据区间的开头。
it_end	指定的数据区间的末尾。

返回值:

无。

注意:

如果 `pt_rb_tree == NULL` 则函数的行为是未定义的，rb tree 容器必须已经初始化的，否则函数的行为是未定义的。

[it_begin, it_end) 必须是有效的数据区间并且与保存在 rb tree 容器中的数据类型相同，否则函数的行为是未定义的。
[it_begin, it_end) 必须不属于 pt_rb_tree，否则函数的行为是未定义的。

```
void _rb_tree_erase_pos(_rb_tree_t* pt_rb_tree, _rb_tree_iterator_t it_pos);
```

描述：

删除 rb tree 容器中指定位置的数据。

参数：

pt_rb_tree 指向 rb tree 容器的指针。
it_pos 被删除的数据的位置。

返回值：

无。

注意：

如果 pt_rb_tree == NULL 则函数的行为是未定义的，rb tree 容器必须已经初始化的，否则函数的行为是未定义的。
it_pos 是属于 pt_rb_tree 容器的有效的迭代器，否则函数的行为是未定义的。

```
void _rb_tree_erase_range(_rb_tree_t* pt_rb_tree,  
    _rb_tree_iterator_t it_begin, _rb_tree_iterator_t it_end);
```

描述：

删除 rb tree 中指定数据区间的数据。

参数：

pt_rb_tree 指向 rb tree 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值：

无。

注意：

如果 pt_rb_tree == NULL 则函数的行为是未定义的，rb tree 容器必须已经初始化的，否则函数的行为是未定义的。
[it_begin, it_end) 必须属于 pt_rb_tree，否则函数的行为是未定义的。

```
void _rb_tree_erase(_rb_tree_t* pt_rb_tree, const void* cpv_value);
```

描述：

删除 rb tree 中指定的数据。

参数：

pt_rb_tree 指向 rb tree 容器的指针。
cpv_value 要删除的指定数据。

返回值：

被删除的数据的个数。

注意：

如果 pt_rb_tree == NULL 或者 cpv_value == NULL 则函数的行为是未定义的，rb tree 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 rb tree 容器中的数据类型相同，否则函数的行为是未定义的。

第五节 迭代器接口

rb tree 的迭代器是双向的迭代器，所以它没有随机访问迭代器那么强大的功能，向前和向后移动只能单步运行。这些接口只能由迭代器接口使用，用户不应该直接使用这些接口函数。

_create_rb_tree_iterator	创建 rb tree 迭代器。
_rb_tree_iterator_get_value	获得 rb tree 迭代器引用的数据。
_rb_tree_iterator_get_pointer	获得 rb tree 迭代器引用的数据的指针。
_rb_tree_iterator_next	获得引用下一个数据的迭代器。
_rb_tree_iterator_prev	获得引用前一个数据的迭代器。
_rb_tree_iterator_equal	测试两个迭代器是否相等。
_rb_tree_iterator_distance	计算两个迭代器之间的距离。
_rb_tree_iterator_before	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
_rb_tree_iterator_t _create_rb_tree_iterator(void);
```

描述：

创建一个 rb tree 迭代器。

参数：

无。

返回值：

rb tree 迭代器。

注意：

返回的 rb tree 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _rb_tree_iterator_get_value(_rb_tree_iterator_t it_iter, void* pv_value);
```

描述：

获得迭代器引用的数据。

参数：

it_iter rb tree 迭代器。

pv_value 保存数据的缓冲区。

返回值：

无。

注意：

it_iter 是有效的 rb tree 迭代器，否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的，pv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _rb_tree_iterator_get_pointer(_rb_tree_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter rb tree 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 rb tree 迭代器, 否则函数的行为是未定义的。

```
_rb_tree_iterator_t _rb_tree_iterator_next(_rb_tree_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter rb tree 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 rb tree 迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 rb tree 有效的迭代器, 否则函数的行为是未定义的。

```
_rb_tree_iterator_t _rb_tree_iterator_prev(_rb_tree_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter rb tree 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 rb tree 迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 rb tree 有效的迭代器, 否则函数的行为是未定义的。

```
bool_t _rb_tree_iterator_equal(  
    _rb_tree_iterator_t it_first, _rb_tree_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first rb tree 迭代器。

it_second rb tree 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 rb tree 容器的有效的 rb tree 迭代器，否则函数的行为是未定义的。两个 rb tree 迭代器相等是指两个迭代器引用相同的数据。

```
int _rb_tree_iterator_distance(
    _rb_tree_iterator_t it_first, _rb_tree_iterator_t it_second);
```

描述：

计算两个迭代器之间的距离。

参数：

it_iterfirst rb tree 迭代器。
it_itersecond rb tree 迭代器。

返回值：

返回两个迭代器之间的距离。

注意：

两个 iter 是属于同一个 rb tree 容器的有效的 rb tree 迭代器，否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0， it_first 引用的数据在 it_second 后面那么结果<0，如果两个迭代器相等那么结果为 0。

```
bool_t _rb_tree_iterator_before(
    _rb_tree_iterator_t it_first, _rb_tree_iterator_t it_second);
```

描述：

测试第一个迭代器是否在第二个迭代器的前面。

参数：

it_first rb tree 迭代器。
it_second rb tree 迭代器。

返回值：

如果第一个迭代器在第二个迭代器的前面则返回 true，否则返回 false。

注意：

两个 iter 是属于同一个 rb tree 容器的有效的 rb tree 迭代器，否则函数的行为是未定义的。

第六节 内部和辅助接口

rb tree 也只是提供了简单的内部接口。

_create_rb_tree_auxiliary	创建 rb tree 容器的辅助函数。
_rb_tree_destroy_auxiliary	销毁 rb tree 容器的辅助函数。

```
bool_t _create_rb_tree_auxiliary(_rb_tree_t* pt_rb_tree, const char* s_typename);
```

描述：

创建一个 rb tree 容器的辅助函数。

参数：

pt_rb_tree 没有创建的 rb tree 容器。

`s_typename` rb tree 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pt_rb_tree == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型 `libcstl` 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _rb_tree_destroy_auxiliary(_rb_tree_t* pt_rb_tree);
```

描述:

销毁 rb tree 容器的辅助函数。

参数:

`pt_rb_tree` rb tree 容器。

返回值:

无。

注意:

如果 `pt_rb_tree == NULL` 或者 rb tree 不是使用 `_create_rb_tree` 生成的则函数的行为是未定义的。

rb tree 提供了大量的辅助函数, 这些函数用于检测结构的有效性, 对树进行调整等等。

<code>_rb_tree_is_created</code>	测试 <code>rb_tree</code> 容器是否是由 <code>_create_rb_tree</code> 函数创建的。
<code>_rb_tree_is_inited</code>	测试 <code>rb_tree</code> 容器是否已经被初始化。
<code>_rb_tree_iterator_belong_to_rb_tree</code>	测试一个迭代器是否属于指定的 <code>rb_tree</code> 容器。
<code>_rb_tree_same_rb_tree_iterator_type</code>	测试 <code>rb_tree</code> 容器中保存的数据类型和迭代器引用的数据类型是否相等。
<code>_rb_tree_rbnodbelong_to_rb_tree</code>	测试一个 rb node 是否属于指定的 <code>rb_tree</code> 容器。
<code>_rb_tree_same_type</code>	测试两个 <code>rb_tree</code> 保存的数据类型是否相同。
<code>_rb_tree_same_type_ex</code>	测试两个 <code>rb_tree</code> 保存的数据类型以及比较函数是否相同。
<code>_rb_tree_find_value</code>	在 <code>rb tree</code> 中查找指定的值所在的节点。
<code>_rb_tree_destroy_subtree</code>	销毁 sub <code>rb tree</code> 。
<code>_rb_tree_clockwise_rotation</code>	顺时针旋转。
<code>_rb_tree_anticlockwise_rotation</code>	逆时针旋转。
<code>_rb_tree_insert_rbnod</code>	向 <code>rb tree</code> 中插保存指定值的 rb node。
<code>_rb_tree_get_color</code>	获得 <code>rb tree</code> 的节点颜色。
<code>_rb_tree_get_min_rbnod</code>	获得 <code>rb tree</code> 中值最小的节点。
<code>_rb_tree_get_max_rbnod</code>	获得 <code>rb tree</code> 中值最大的节点。
<code>_rb_tree_rebalance</code>	重新调整 <code>rb tree</code> , 使它恢复平衡。
<code>_rb_tree_fixup_deletion</code>	将删除节点之后的 <code>rb tree</code> 重新调整平衡。
<code>_rb_tree_init_elem_auxiliary</code>	初始化数据的辅助函数。

`_rb_tree_elem_compare_auxiliary` 比较数据的辅助函数。

函数原型

```
bool_t _rb_tree_is_created(const _rb_tree_t* cpt_rb_tree);
```

描述:

测试一个 rb_tree 是否是使用_create_rb_tree 创建的。

参数:

cpt_rb_tree rb_tree 容器。

返回值:

如果 rb_tree 是使用_create_rb_tree 创建的则返回 true, 否则返回 false。

注意:

如果 cpt_rb_tree == NULL, 函数的行为是未定义的。

```
bool_t _rb_tree_is_inited(const _rb_tree_t* cpt_rb_tree);
```

描述:

测试一个 rb_tree 是否已经初始化。

参数:

cpt_rb_tree rb_tree 容器。

返回值:

如果 rb_tree 已经初始化了, 返回 true, 否则返回 false。

注意:

如果 cpt_rb_tree == NULL, 函数的行为是未定义的。

```
bool_t _rb_tree_iterator_belong_to_rb_tree(
    const _rb_tree_t* cpt_rb_tree, _rb_tree_iterator_t it_iter);
```

描述:

测试一个迭代器是否属于指定的 rb_tree 容器。

参数:

cpt_rb_tree rb tree 容器。

it_iter rb tree 迭代器。

返回值:

如果迭代器引用的数据属于 rb tree 容器的有效范围内, 返回 true, 否则返回 false。

注意:

如果 cpt_rb_tree == NULL, 那么函数的行为是未定义的, cpt_rb_tree 必须是已经初始化的, 否则函数的行为是未定义的。如果 it_iter 不是 rb tree 迭代器, 那么函数的行为是未定义的。

```
bool_t _rb_tree_same_rb_tree_iterator_type(
    const _rb_tree_t* cpt_rb_tree, _rb_tree_iterator_t it_iter);
```

描述:

测试 rb_tree 容器中保存的数据类型和迭代器引用的数据类型是否相等。

参数:

cpt_rb_tree rb tree 容器。

it_iter rb tree 迭代器。

返回值:

如果 rb tree 中保存的数据类型与 it_iter 引用的数据类型相同返回 true, 否则返回 false。

注意:

如果 cpt_rb_tree == NULL 或者 it_iter 不是_rb_tree_iterator_t 类型, 那么函数的行为是未定义的。cpt_rb_tree 必须是使用_create_rb_tree()创建的, 否则函数的行为是未定义的。

```
bool_t _rb_tree_rbnоде_belong_to_rb_tree(
    const _rbnode_t* cpt_root, const _rbnode_t cpt_pos);
```

描述:

测试一个 rb node 是否属于指定的 rb tree 容器。

参数:

cpt_root rb tree 子树的根。

cpt_pos 要查找的 rb node。

返回值:

如果要查找的 rb node 在 rb tree 子树内, 返回 true, 否则返回 false。

注意:

如果 cpt_root == NULL 或者 cpt_pos == NULL 则返回 false。

```
bool_t _rb_tree_same_type(
    const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

判断两个 rb tree 容器中保存的数据类型是否相同。

参数:

cpt_first 第一个 rb tree 容器。

cpt_second 第二个 rb tree 容器。

返回值:

如果两个 rb tree 中保存的数据类型相同返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL, 那么函数的行为是未定义的。两个 rb tree 容器必须是已经初始化或者使用_create_rb_tree 创建的 rb tree 容器, 否则函数的行为是未定义的。如果 cpt_first == cpt_second 则返回 true。

```
bool_t _rb_tree_same_type_ex(
    const _rb_tree_t* cpt_first, const _rb_tree_t* cpt_second);
```

描述:

判断两个 rb tree 容器中保存的数据类型以及比较函数是否相同。

参数:

cpt_first 第一个 rb tree 容器。

cpt_second 第二个 rb tree 容器。

返回值:

如果两个 rb tree 中保存的数据类型以及比较函数相同返回 true, 否则返回 false。

注意:

如果 `cpt_first == NULL` 或者 `cpt_second == NULL`, 那么函数的行为是未定义的。两个 rb tree 容器必须是已经初始化或者使用 `_create_rb_tree` 创建的 rb tree 容器, 否则函数的行为是未定义的。如果 `cpt_first == cpt_second` 则返回 true。

```
_rbnode_t* _rb_tree_destroy_subtree(  
    _rb_tree_t* pt_rb_tree, _rbnode_t* pt_root);
```

描述:

销毁 sub rb tree。

参数:

<code>pt_val_tree</code>	rb tree 容器。
<code>pt_root</code>	子树的根节点。

返回值:

返回 NULL。

注意:

如果 `pt_rb_tree == NULL`, 则函数的行为是未定义的。rb tree 容器必须是已经初始化或者使用 `_create_rb_tree` 创建的 rb tree 容器, 否则函数的行为是未定义的。

```
_rbnode_t* _rb_tree_find_value(  
    const _rb_tree_t* cpt_rb_tree, const _rbnode_t* cpt_root,  
    const void* cpv_value);
```

描述:

在 rb tree 中查找指定的值所在的节点。

参数:

<code>cpt_val_tree</code>	rb tree 容器。
<code>cpt_root</code>	子树的根节点。
<code>cpv_value</code>	查找的数据。

返回值:

如果在 rb tree 子树中找到了相应的数据, 则返回保存该数据的 rbnodet 指针, 否则返回 NULL。

注意:

如果 `cpt_rb_tree == NULL` 或者 `cpv_value == NULL`, 那么函数的行为是未定义的。rb tree 容器必须是已经初始化的 rb tree 容器, 否则函数的行为是未定义的。

```
_rbnode_t* _rb_tree_clockwise_rotation(_rbnode_t* pt_root);
```

描述:

顺时针旋转。

参数:

<code>pt_root</code>	子树的根节点。
----------------------	---------

返回值:

旋转之后新的子树根节点。

注意:

如果 `pt_root == NULL` 或者 `pt_root->_pt_left == NULL` 则函数的行为是未定义的。

```
_rbnode_t* _rb_tree_anticlockwise_rotation(_rbnode_t* pt_root);
```

描述:

逆时针旋转。

参数:

pt_root 子树的根节点。

返回值:

旋转之后新的子树根节点。

注意:

如果 pt_root == NULL 或者 pt_root->_pt_right == NULL 则函数的行为是未定义的。

```
_rb_tree_insert_result_t _rb_tree_insert_rbnod(
    const _rb_tree_t* cpt_rb_tree, const void* cpv_value);
```

描述:

向 rb tree 中插保存指定值的 rb node。

参数:

cpt_rb_tree rb tree 容器。
cpv_value 查找的数据。

返回值:

返回插入节点后的新的根和新节点。

注意:

如果 cpt_rb_tree == NULL 或者 cpv_value == NULL, 则函数的行为是未定义的。rb tree 容器必须是已经初始化, 否则函数的行为是未定义的。

```
int _rb_tree_get_color(const _rbnode_t* cpt_root);
```

描述:

获得 rb tree 的颜色。

参数:

cpt_root 子树的根节点。

返回值:

返回节点的颜色。

注意:

如果 cpt_root == NULL 则返回 black。

```
_rbnode_t* _rb_tree_get_min_rbnod(const _rbnode_t* cpt_root);
```

描述:

获得 rb tree 中值最小的节点。

参数:

cpt_root 子树的根节点。

返回值:

值最小的节点。

注意:

如果 cpt_root == NULL 则函数的行为是未定义的。

```
_rbnode_t* _rb_tree_get_max_rbnodE(const _rbnode_t* cpt_root);
```

描述:

获得 rb tree 中值最大的节点。

参数:

cpt_root 子树的根节点。

返回值:

值最大的节点。

注意:

如果 cpt_root == NULL 则函数的行为是未定义的。

```
void _rb_tree_rebalance(_rb_tree_t* pt_rb_tree, _rbnode_t* pt_pos);
```

描述:

重新调整 rb tree, 使它恢复平衡。

参数:

_rb_tree rb tree 容器。

pt_pos 需要调整的位置。

返回值:

无。

注意:

如果 pt_rb_tree == NULL 或者 pt_pos == NULL, 则函数的行为是未定义的。rb tree 容器必须是已经初始化, 否则函数的行为是未定义的。pt_pos 不能够是 pt_rb_tree 的根, 否则函数的行为是未定义的。如果 pt_pos 不属于 pt_rb_tree 则函数的行为是未定义的。

```
void _rb_tree_fixup_deletion(
    _rb_tree_t* pt_rb_tree, _rbnode_t* pt_pos, _rbnode_t* pt_parent);
```

描述:

将删除节点之后的 rb tree 重新调整平衡。

参数:

_rb_tree rb tree 容器。

pt_pos 需要调整的位置。

pt_parent 调整位置的父节点。

返回值:

无。

注意:

如果 pt_rb_tree == NULL 或者 pt_parent == NULL, 则函数的行为是未定义的。rb tree 容器必须是已经初始化, 否则函数的行为是未定义的。pt_parent 必须是属于 pt_rb_tree 的节点并且 pt_parent 必须有一个子节点, 否则函数的行为是未定义的。

```
void _rb_tree_init_elem_auxiliary(_rb_tree_t* pt_rb_tree, _rbnode_t* pt_node);
```

描述:

初始化数据的辅助函数。

参数:

pt_rb_tree rb tree 容器。

pt_node 初始化节点。

返回值:

无。

注意:

如果 pt_rb_tree == NULL 或者 pt_node == NULL , 那么函数的行为是未定义的。pt_rb_tree 必须是初始化的或者是使用_create_rb_tree()创建的, 否则函数的行为是未定义的。

```
void _rb_tree_elem_compare_auxiliary(const _rb_tree_t* cpt_rb_tree,
                                     const void* cpv_first, const void* cpv_second, void* pv_output);
```

描述:

比较数据的辅助函数。

参数:

cpt_rb_tree rb tree 容器。

cpv_first 第一个数据。

cpv_second 第二个数据。

pv_output 比较结果。

返回值:

无。

注意:

如果 cpt_rb_tree == NULL 或者 cpv_first == NULL 或者 cpv_second == NULL 或者 pv_output == NULL , 那么函数的行为是未定义的。cpt_rb_tree 必须是初始化的或者是使用_create_rb_tree()创建的, 否则函数的行为是未定义的。

第十六章 set 和 multiset

第一节 set 和 multiset 的机制

Set 和 multiset 是集合的抽象，集合中的数据是有序的，可以实现数据的快速查找。set 和 multiset 的底层可以使用 avl tree 或者是 rb tree，set 和 multiset 是对于 avl tree 或者 rb tree 的封装，它们的区别就是 set 中不包含重复的元素，multiset 允许数据的重复。

第二节 set 和 multiset 的迭代器

Set 和 multiset 的迭代器就是 avl tree 或者 rb tree 的实现。

第三节 set 和 multiset 的代码结构

set 的代码结构如下

```
typedef struct _tagset
{
#ifdef CSTL_SET_AVL_TREE
    _avl_tree_t _t_tree;
#else
    _rb_tree_t _t_tree;
#endif
}set_t;
```

multiset 的代码结构如下

```
typedef struct _tagmultiset
{
#ifdef CSTL_MULTISSET_AVL_TREE
    _avl_tree_t _t_tree;
#else
    _rb_tree_t _t_tree;
#endif
}multiset_t;
```

可以通过配置来决定使用 avl tree 还是使用 rb tree 作为底层实现。

第四节 外部接口

set 提供的外部接口提供给用户使用。

create_set	创建 set 容器。
set_init	初始化 set 容器。
set_init_ex	使用自定义比较规则初始化 set 容器。
set_init_copy	使用存在的 set 容器进行初始化。
set_init_copy_range	使用指定的数据区间进行初始化。
set_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
set_destroy	销毁 set 容器。
set_assign	使用存在的 set 容器赋值。
set_size	返回 set 容器中数据的数量。
set_empty	判断 set 容器是否为空。
set_max_size	返回 set 容器中能够保存数据的最大数量。
set_begin	返回指向 set 容器的第一个数据的迭代器。
set_end	返回指向 set 容器末尾的迭代器。
set_key_comp	返回 set 容器中的键比较规则。
set_value_comp	返回 set 容器中的数据比较规则。
set_find	在 set 容器中查找指定的数据。
set_clear	清空 set 容器。
set_count	返回 set 容器中指定数据的数量。
set_lower_bound	返回指向第一个大于等于指定数据的迭代器。
set_upper_bound	返回指向第一个大于指定数据的迭代器。
set_equal_range	返回包含指定数据的数据区间。
set_equal	测试两个 set 容器是否相等。
set_not_equal	测试两个 set 容器是否不等。
set_less	测试第一个 set 容器是否小于第二个 set 容器。
set_less_equal	测试第一个 set 容器是否小于等于第二个 set 容器。
set_greater	测试第一个 set 容器是否大于第二个 set 容器。

set_greater_equal	测试第一个 set 容器是否大于等于第二个 set 容器。
set_swap	交换两个 set 容器。
set_insert	向 set 中插入数据，数据必须唯一。
set_insert_hint	向 set 中线索位置插入数据，数据必须唯一。
set_insert_range	向 set 中插入指定的数据区间，数据必须唯一。
set_erase_pos	删除 set 中指定位置的数据。
set_erase_range	删除 set 中指定数据区间的数据。
set_erase	删除 set 中指定的数据。

函数原型

```
set_t* create_set(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

typename 类型描述。

返回值:

成功返回指向 set 容器的指针，否则返回 NULL。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void set_init(set_t* pset_set);
```

描述:

初始化 set 迭代器。

参数:

pset_set set 容器。

返回值:

无。

注意:

pset_set == NULL 则函数的行为是未定义的，pset_set 必须是使用 create_set() 创建的，否则函数的行为是未定义的，这个函数使用类型默认的比较函数。

```
void set_init_copy(set_t* pset_dest, const set_t* cpset_src);
```

描述:

使用已经存在的 set 初始化 set 迭代器。

参数:

pset_dest 目的 set 容器。

cpset_src 源 set 容器。

返回值:

无。

注意:

pset_dest == NULL 或者 cpset_src == NULL 则函数的行为是未定义的, pset_dest 必须是使用 create_set() 创建的, cpset_src 必须是已经初始化, 否则函数的行为是未定义的。pset_dest 和 cpset_src 保存的数据类型必须相同, 否则函数的行为是未定义。

```
void set_init_copy_range(
    set_t* pset_dest, set_iterator_t it_begin, set_iterator_t it_end);
```

描述:

使用指定的数据区间初始化 rb tree 迭代器。

参数:

pset_dest	目的 set 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。

返回值:

无。

注意:

如果 pset_dest == NULL 则函数的行为是未定义的, pset_dest 必须是使用 create_set() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 set 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void set_init_copy_range_ex(
    set_t* pt_dest, set_iterator_t it_begin, set_iterator_t it_end,
    binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 set 迭代器。

参数:

pset_dest	目的 set 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。
t_compare	比较函数。

返回值:

无。

注意:

如果 pset_dest == NULL 则函数的行为是未定义的, pset_dest 必须是使用 create_set() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 set 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 t_compare == NULL 则使用类型默认的比较函数。

```
void set_destroy(set_t* pset_set);
```

描述:

销毁创建的 set 容器。

参数:

pset_set	set 容器。
----------	---------

返回值:

无。

注意:

如果 `pset_set == NULL` 则函数的行为是未定义的, `pset_set` 必须是初始化或者是使用 `create_set()` 创建的, 否则函数的行为是未定义的。

```
void set_assign(set_t* pset_dest, const set_t* cpset_src);
```

描述:

使用一个 set 为另一个 set 赋值。

参数:

`pset_dest` 指向被赋值的 set 容器的指针。
`cpset_src` 指向赋值的 set 容器的指针。

返回值:

无。

注意:

如果 `pset_dest == NULL` 或者 `cpset_src == NULL` 则函数的行为是未定义的, 两个 set 必须已经初始化的, 否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同函数的行为是未定义的。如果 `set_equal(pset_dest, cpset_src)` 那么函数不做任何动作。

```
size_t set_size(const set_t* cpset_set);
```

描述:

获得 set 容器中保存的数据的个数。

参数:

`cpset_set` set 容器。

返回值:

set 容器中保存的数据的个数。

注意:

如果 `cpset_set == NULL` 则函数的行为是未定义的, `cpset_set` 必须是已经初始化的 set 容器, 否则函数的行为是未定义的。

```
bool_t set_empty(const set_t* cpset_set);
```

描述:

测试 set 容器是否为空。

参数:

`cpset_set` set 容器。

返回值:

如果 set 容器为空则返回 `true`, 否则返回 `false`。

注意:

如果 `cpset_set == NULL` 则函数的行为是未定义的, `cpset_set` 必须是已经初始化的 set 容器, 否则函数的行为是未定义的。

```
size_t set_max_size(const set_t* cpset_set);
```

描述:

获得 set 容器中数据的最大数量。

参数:

cpset_set set 容器。

返回值:

set 容器中保存的数据的个数的最大值。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的, cpset_set 必须是已经初始化的 set 容器, 否则函数的行为是未定义的。

```
set_iterator_t set_begin(const set_t* cpset_set);
```

描述:

获得引用 set 容器中第一数据的迭代器。

参数:

cpset_set 指向 set 容器的指针。

返回值:

返回引用 set 容器中第一个数据的迭代器。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。如果 set 容器为空, 则返回值与 set_end(cpset_set)相等。

```
set_iterator_t set_end(const set_t* cpset_set);
```

描述:

获得引用 set 容器末尾的迭代器。

参数:

cpset_set 指向 set 容器的指针。

返回值:

返回引用 set 容器末尾的迭代器。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t set_key_comp(const set_t* cpset_set);
```

描述:

返回 set 容器中的键比较规则。

参数:

cpset_set 指向 set 容器的指针。

返回值:

返回 set 容器中的数据比较规则。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t set_value_comp(const set_t* cpset_set);
```

描述:

返回 set 容器中的数据比较规则。

参数:

cpset_set 指向 set 容器的指针。

返回值:

返回 set 容器中的数据比较规则。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。

```
set_iterator_t set_find(const set_t* cpset_set, elem);
```

描述:

在 set 容器中查找指定的数据。

参数:

cpset_set 指向 set 容器的指针。

elem 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void set_clear(set_t* pset_set);
```

描述:

删除 set 容器中的所有数据。

参数:

pset_set 指向 set 容器的指针。

返回值:

无。

注意:

如果 pset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。

```
size_t set_count(const set_t* cpset_set, elem);
```

描述:

返回 set 容器中指定数据的数量。

参数:

cpset_set 指向 set 容器的指针。

elem 要查找的指定数据。

返回值:

返回 set 容器中指定数据的数量。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t set_lower_bound(const set_t* cpset_set, elem);
```

描述:

 返回指向第一个大于等于指定数据的迭代器。

参数:

 cpset_set 指向 set 容器的指针。

 elem 要查找的指定数据。

返回值:

 返回指向第一个大于等于指定数据的迭代器。

注意:

 如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t set_upper_bound(const set_t* cpset_set, elem);
```

描述:

 返回指向第一个大于指定数据的迭代器。

参数:

 cpset_set 指向 set 容器的指针。

 elem 要查找的指定数据。

返回值:

 返回指向第一个大于指定数据的迭代器。

注意:

 如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t set_equal_range(const set_t* cpset_set, elem);
```

描述:

 返回包含指定数据的数据区间。

参数:

 cpset_set 指向 set 容器的指针。

 elem 要查找的指定数据。

返回值:

 返回包含指定数据的数据区间。

注意:

 如果 cpset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
bool_t set_equal(const set_t* cpset_first, const set_t* cpset_second);
```

描述:

 测试两个 set 容器是否相等。

参数:

 cpset_first 指向 set 容器的指针。

`cpset_second` 指向 set 容器的指针。

返回值:

如果两个 set 容器相等则返回 true, 否则返回 false。

注意:

如果 `cpset_first == NULL` 或者 `cpset_second == NULL` 则函数的行为是未定义的, 两个 set 必须已经初始化的, 否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同认为这两个容器不等。两个 set 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 `cpset_first == cpset_second` 那么返回 true。

```
bool_t set_not_equal(const set_t* cpset_first, const set_t* cpset_second);
```

描述:

测试两个 set 容器是否不等。

参数:

`cpset_first` 指向 set 容器的指针。

`cpset_second` 指向 set 容器的指针。

返回值:

如果两个 set 容器不等则返回 true, 否则返回 false。

注意:

如果 `cpset_first == NULL` 或者 `cpset_second == NULL` 则函数的行为是未定义的, 两个 set 必须已经初始化的, 否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同认为两个 set 容器不等。两个 set 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 `cpset_first == cpset_second` 那么返回 false。

```
bool_t set_less(const set_t* cpset_first, const set_t* cpset_second);
```

描述:

测试第一个 set 是否小于第二个 set。

参数:

`cpset_first` 指向 set 容器的指针。

`cpset_second` 指向 set 容器的指针。

返回值:

如果第一个 set 容器小于第二个 set 容器则返回 true, 否则返回 false。

注意:

如果 `cpset_first == NULL` 或者 `cpset_second == NULL` 则函数的行为是未定义的, 两个 set 必须已经初始化的, 否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 set 中的数据小于第二个 set 中对应的数据则返回 true, 如果大于则返回 false, 当两个 set 中的对应数据相等, 第一个 set 中的数据数量小于第二个 set 中数据的数量返回 true, 否则返回 false。如果 `cpset_first == cpset_second` 那么返回 false。

```
bool_t set_less_equal(const set_t* cpset_first, const set_t* cpset_second);
```

描述:

测试第一个 set 是否小于等于第二个 set。

参数:

`cpset_first` 指向 set 容器的指针。

`cpset_second` 指向 set 容器的指针。

返回值:

如果第一个 set 容器小于等于第二个 set 容器则返回 true, 否则返回 false。

注意:

如果 cpset_first == NULL 或者 cpset_second == NULL 则函数的行为是未定义的, 两个 set 必须已经初始化的, 否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同函数的行为是未定义的。如果 cpset_first == cpset_second 那么返回 true。

```
bool_t set_greater(const set_t* cpset_first, const set_t* cpset_second);
```

描述:

测试第一个 set 是否大于第二个 set。

参数:

cpset_first	指向 set 容器的指针。
cpset_second	指向 set 容器的指针。

返回值:

如果第一个 set 容器大于第二个 set 容器则返回 true, 否则返回 false。

注意:

如果 cpset_first == NULL 或者 cpset_second == NULL 则函数的行为是未定义的, 两个 set 必须已经初始化的, 否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同函数的行为是未定义的。如果第一 set 中的数据大于第二个 set 中的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 set 中数据的数量大于第二个 set 中数据的数量的时候返回 true 否则返回 false。如果 cpset_first == cpset_second 那么返回 false。

```
bool_t set_greater_equal(const set_t* cpset_first, const set_t* cpset_second);
```

描述:

测试第一个 set 是否大于等于第二个 set。

参数:

cpset_first	指向 set 容器的指针。
cpset_second	指向 set 容器的指针。

返回值:

如果第一个 set 容器大于等于第二个 set 容器则返回 true, 否则返回 false。

注意:

如果 cpset_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 set 必须已经初始化的, 否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同函数的行为是未定义的。如果 cpset_first == cpset_second 那么返回 true。

```
void set_swap(set_t* pset_first, set_t* pset_second);
```

描述:

交换两个 set 容器。

参数:

pset_first	指向 set 容器的指针。
pset_second	指向 set 容器的指针。

返回值:

无。

注意:

如果 pset_first == NULL 或者 pset_second == NULL 则函数的行为是未定义的，两个 set 必须已经初始化的，否则函数的行为是未定义的。两个 set 容器中保存的数据类型不同函数的行为是未定义的。如果 set_equal(pset_first, pset_second)，函数不执行任何动作。

```
set_iterator_t set_insert(set_t* pset_set, elem);
```

描述:

向 set 容器中插入唯一的数据。

参数:

pset_set 指向 set 容器的指针。

elem 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

```
set_iterator_t set_insert_hint(set_t* pset_set, set_iterator_t it_hint, elem);
```

描述:

向 set 容器中的线索位置插入唯一的数据。

参数:

pset_set 指向 set 容器的指针。

it_hint 用户提供的线索位置。

elem 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

```
void set_insert_range(
    set_t* pset_set, set_iterator_t it_begin, _set_iterator_t it_end);
```

描述:

向 set 中插入指定的数据区间，数据必须唯一。

参数:

pset_set 指向 set 容器的指针。

it_begin 指定的数据区间的开头。

it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。

[it_begin, it_end) 必须是有效的数据区间并且与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。
[it_begin, it_end) 必须不属于 pset_set，否则函数的行为是未定义的。

```
void set_erase_pos(set_t* pset_set, set_iterator_t it_pos);
```

描述：

删除 set 容器中指定位置的数据。

参数：

pset_set 指向 set 容器的指针。
it_pos 被删除的数据的位置。

返回值：

无。

注意：

如果 pset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。it_pos 是属于 pset_set 容器的有效的迭代器，否则函数的行为是未定义的。

```
void set_erase_range(  
    set_t* pset_set, set_iterator_t it_begin, set_iterator_t it_end);
```

描述：

删除 set 中指定数据区间的数据。

参数：

pset_set 指向 set 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值：

无。

注意：

如果 pset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。
[it_begin, it_end) 必须属于 pset_set，否则函数的行为是未定义的。

```
void set_erase(set_t* pset_set, elem);
```

描述：

删除 set 中指定的数据。

参数：

pset_set 指向 set 容器的指针。
elem 要删除的指定数据。

返回值：

被删除的数据的个数。

注意：

如果 pset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

multiset 提供的外部接口提供给用户使用。

create_multiset	创建 multiset 容器。
multiset_init	初始化 multiset 容器。
multiset_init_ex	使用自定义比较规则初始化 multiset 容器。
multiset_init_copy	使用存在的 multiset 容器进行初始化。
multiset_init_copy_range	使用指定的数据区间进行初始化。
multiset_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
multiset_destroy	销毁 multiset 容器。
multiset_assign	使用存在的 multiset 容器赋值。
multiset_size	返回 multiset 容器中数据的数量。
multiset_empty	判断 multiset 容器是否为空。
multiset_max_size	返回 multiset 容器中能够保存数据的最大数量。
multiset_begin	返回指向 multiset 容器的第一个数据的迭代器。
multiset_end	返回指向 multiset 容器末尾的迭代器。
multiset_key_comp	返回 multiset 容器中的键比较规则。
multiset_value_comp	返回 multiset 容器中的数据比较规则。
multiset_find	在 multiset 容器中查找指定的数据。
multiset_clear	清空 multiset 容器。
multiset_count	返回 multiset 容器中指定数据的数量。
multiset_lower_bound	返回指向第一个大于等于指定数据的迭代器。
multiset_upper_bound	返回指向第一个大于指定数据的迭代器。
multiset_equal_range	返回包含指定数据的数据区间。
multiset_equal	测试两个 multiset 容器是否相等。
multiset_not_equal	测试两个 multiset 容器是否不等。
multiset_less	测试第一个 multiset 容器是否小于第二个 multiset 容器。
multiset_less_equal	测试第一个 multiset 容器是否小于等于第二个 multiset 容器。
multiset_greater	测试第一个 multiset 容器是否大于第二个 multiset 容器。
multiset_greater_equal	测试第一个 multiset 容器是否大于等于第二个 multiset 容器。
multiset_swap	交换两个 multiset 容器。
multiset_insert	向 multiset 中插入数据。
multiset_insert_hint	向 multiset 中线索位置插入数据。
multiset_insert_range	向 multiset 中插入指定的数据区间。
multiset_erase_pos	删除 multiset 中指定位置的数据。

<code>multiset_erase_range</code>	删除 multiset 中指定数据区间的数据。
<code>multiset_erase</code>	删除 multiset 中指定的数据。

函数原型

```
multiset_t* create_multiset(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

`typename` 类型描述。

返回值:

成功返回指向 multiset 容器的指针，否则返回 NULL。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void multiset_init(multiset_t* pmset_mset);
```

描述:

初始化 multiset 迭代器。

参数:

`pmset_mset` multiset 容器。

返回值:

无。

注意:

`pmset_mset == NULL` 则函数的行为是未定义的，`pmset_mset` 必须是使用 `create_multiset()` 创建的，否则函数的行为是未定义的，这个函数使用类型默认的比较函数。

```
void multiset_init_copy(multiset_t* pmset_dest, const multiset_t* cpmset_src);
```

描述:

使用已经存在的 multiset 初始化 multiset 迭代器。

参数:

`pmset_dest` 目的 multiset 容器。

`cpmultisett_src` 源 multiset 容器。

返回值:

无。

注意:

`pmset_dest == NULL` 或者 `cpmultisett_src == NULL` 则函数的行为是未定义的，`pmset_dest` 必须是使用 `create_multiset()` 创建的，`cpmultisett_src` 必须是已经初始化，否则函数的行为是未定义的。`pmset_dest` 和 `cpmultisett_src` 保存的数据类型必须相同，否则函数的行为是未定义。

```
void multiset_init_copy_range(
    multiset_t* pmset_dest,
    multiset_iterator_t it_begin, multiset_iterator_t it_end);
```

描述:

使用指定的数据区间初始化 rb tree 迭代器。

参数:

pmset_dest	目的 multiset 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。

返回值:

无。

注意:

如果 pmset_dest == NULL 则函数的行为是未定义的, pmset_dest 必须是使用 create_multiset() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 multiset 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void multiset_init_copy_range_ex(
    multiset_t* pt_dest, multiset_iterator_t it_begin, multiset_iterator_t it_end,
    binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 multiset 迭代器。

参数:

pmset_dest	目的 multiset 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。
t_compare	比较函数。

返回值:

无。

注意:

如果 pmset_dest == NULL 则函数的行为是未定义的, pmset_dest 必须是使用 create_multiset() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 multiset 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 t_compare == NULL 则使用类型默认的比较函数。

```
void multiset_destroy(multiset_t* pmset_mset);
```

描述:

销毁创建的 multiset 容器。

参数:

pmset_mset	multiset 容器。
------------	--------------

返回值:

无。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的, pmset_mset 必须是初始化或者是使用 create_multiset() 创建的, 否则函数的行为是未定义的。

```
void multiset_assign(multiset_t* pmset_dest, const multiset_t* cpmset_src);
```

描述:

使用一个 multiset 为另一个 multiset 赋值。

参数:

pmset_dest 指向被赋值的 multiset 容器的指针。
cpmset_src 指向赋值的 multiset 容器的指针。

返回值:

无。

注意:

如果 pmset_dest == NULL 或者 cpmset_src == NULL 则函数的行为是未定义的，两个 multiset 必须已经初始化的，否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 multiset_equal(pmset_dest, cpmset_src) 那么函数不做任何动作。

```
size_t multiset_size(const multiset_t* cpmset_mset);
```

描述:

获得 multiset 容器中保存的数据的个数。

参数:

cpmset_mset multiset 容器。

返回值:

multiset 容器中保存的数据的个数。

注意:

如果 cpmset_mset == NULL 则函数的行为是未定义的，cpmset_mset 必须是已经初始化的 multiset 容器，否则函数的行为是未定义的。

```
bool_t multiset_empty(const multiset_t* cpmset_mset);
```

描述:

测试 multiset 容器是否为空。

参数:

cpmset_mset multiset 容器。

返回值:

如果 multiset 容器为空则返回 true，否则返回 false。

注意:

如果 cpmset_mset == NULL 则函数的行为是未定义的，cpmset_mset 必须是已经初始化的 multiset 容器，否则函数的行为是未定义的。

```
size_t multiset_max_size(const multiset_t* cpmset_mset);
```

描述:

获得 multiset 容器中数据的最大数量。

参数:

cpmset_mset multiset 容器。

返回值:

multiset 容器中保存的数据的个数的最大值。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的, `cpmset_mset` 必须是已经初始化的 multiset 容器, 否则函数的行为是未定义的。

```
multiset_iterator_t multiset_begin(const multiset_t* cpmset_mset);
```

描述:

获得引用 multiset 容器中第一数据的迭代器。

参数:

`cpmset_mset` 指向 multiset 容器的指针。

返回值:

返回引用 multiset 容器中第一个数据的迭代器。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。如果 multiset 容器为空, 则返回值与 `multiset_end(cpmset_mset)` 相等。

```
multiset_iterator_t multiset_end(const multiset_t* cpmset_mset);
```

描述:

获得引用 multiset 容器末尾的迭代器。

参数:

`cpmset_mset` 指向 multiset 容器的指针。

返回值:

返回引用 multiset 容器末尾的迭代器。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t multiset_key_comp(const multiset_t* cpmset_mset);
```

描述:

返回 multiset 容器中的键比较规则。

参数:

`cpmset_mset` 指向 multiset 容器的指针。

返回值:

返回 multiset 容器中的数据比较规则。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t multiset_value_comp(const multiset_t* cpmset_mset);
```

描述:

返回 multiset 容器中的数据比较规则。

参数:

`cpmset_mset` 指向 multiset 容器的指针。

返回值:

返回 multiset 容器中的数据比较规则。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。

```
multiset_iterator_t multiset_find(const multiset_t* cpmset_mset, elem);
```

描述:

在 multiset 容器中查找指定的数据。

参数:

`cpmset_mset` 指向 multiset 容器的指针。
`elem` 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void multiset_clear(multiset_t* pmset_mset);
```

描述:

删除 multiset 容器中的所有数据。

参数:

`pmset_mset` 指向 multiset 容器的指针。

返回值:

无。

注意:

如果 `pmset_mset == NULL` 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。

```
size_t multiset_count(const multiset_t* cpmset_mset, elem);
```

描述:

返回 multiset 容器中指定数据的数量。

参数:

`cpmset_mset` 指向 multiset 容器的指针。
`elem` 要查找的指定数据。

返回值:

返回 multiset 容器中指定数据的数量。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
multiset_iterator_t multiset_lower_bound(const multiset_t* cpmset_mset, elem);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cpmset_mset 指向 multiset 容器的指针。

elem 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cpmset_mset == NULL 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
multiset_iterator_t multiset_upper_bound(const multiset_t* cpmset_mset, elem);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cpmset_mset 指向 multiset 容器的指针。

elem 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 cpmset_mset == NULL 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t multiset_equal_range(const multiset_t* cpmset_mset, elem);
```

描述:

返回包含指定数据的数据区间。

参数:

cpmset_mset 指向 multiset 容器的指针。

elem 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_multiset == NULL 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
bool_t multiset_equal(const multiset_t* cpmset_first, const multiset_t* cpmset_second);
```

描述:

测试两个 multiset 容器是否相等。

参数:

cpmset_first 指向 multiset 容器的指针。

cpmset_second 指向 multiset 容器的指针。

返回值:

如果两个 multiset 容器相等则返回 true, 否则返回 false。

注意:

如果 cpmset_first == NULL 或者 cpmset_second == NULL 则函数的行为是未定义的, 两个 multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同认为这两个容器不等。两个 multiset 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cpmset_first == cpmset_second 那么返回 true。

```
bool_t multiset_not_equal(
    const multiset_t* cpmset_first, const multiset_t* cpmset_second);
```

描述:

测试两个 multiset 容器是否不等。

参数:

cpmset_first 指向 multiset 容器的指针。
cpmset_second 指向 multiset 容器的指针。

返回值:

如果两个 multiset 容器不等则返回 true, 否则返回 false。

注意:

如果 cpmset_first == NULL 或者 cpmset_second == NULL 则函数的行为是未定义的, 两个 multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同认为两个 multiset 容器不等。两个 multiset 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cpmset_first == cpmset_second 那么返回 false。

```
bool_t multiset_less(
    const multiset_t* cpmset_first, const multiset_t* cpmset_second);
```

描述:

测试第一个 multiset 是否小于第二个 multiset。

参数:

cpmset_first 指向 multiset 容器的指针。
cpmset_second 指向 multiset 容器的指针。

返回值:

如果第一个 multiset 容器小于第二个 multiset 容器则返回 true, 否则返回 false。

注意:

如果 cpmset_first == NULL 或者 cpmset_second == NULL 则函数的行为是未定义的, 两个 multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 multiset 中的数据小于第二个 multiset 中对应的数据则返回 true, 如果大于则返回 false, 当两个 multiset 中的对应数据相等, 第一个 multiset 中的数据数量小于第二个 multiset 中数据的数量返回 true, 否则返回 false。如果 cpmset_first == cpmset_second 那么返回 false。

```
bool_t multiset_less_equal(
    const multiset_t* cpmset_first, const multiset_t* cpmset_second);
```

描述:

测试第一个 multiset 是否小于等于第二个 multiset。

参数:

cpmset_first 指向 multiset 容器的指针。
cpmset_second 指向 multiset 容器的指针。

返回值:

如果第一个 multiset 容器小于等于第二个 multiset 容器则返回 true, 否则返回 false。

注意:

如果 cpmset_first == NULL 或者 cpmset_second == NULL 则函数的行为是未定义的, 两个 multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 cpmset_first == cpmset_second 那么返回 true。

```
bool_t multiset_greater(  
    const multiset_t* cpmset_first, const multiset_t* cpmset_second);
```

描述:

测试第一个 multiset 是否大于第二个 multiset。

参数:

cpmset_first 指向 multiset 容器的指针。
cpmset_second 指向 multiset 容器的指针。

返回值:

如果第一个 multiset 容器大于第二个 multiset 容器则返回 true, 否则返回 false。

注意:

如果 cpmset_first == NULL 或者 cpmset_second == NULL 则函数的行为是未定义的, 两个 multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 multiset 中的数据大于第二个 multiset 中的对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 multiset 中数据的数量大于第二个 multiset 中数据的数量的时候返回 true 否则返回 false。如果 cpmset_first == cpmset_second 那么返回 false。

```
bool_t multiset_greater_equal(  
    const multiset_t* cpmset_first, const multiset_t* cpmset_second);
```

描述:

测试第一个 multiset 是否大于等于第二个 multiset。

参数:

cpmset_first 指向 multiset 容器的指针。
cpmset_second 指向 multiset 容器的指针。

返回值:

如果第一个 multiset 容器大于等于第二个 multiset 容器则返回 true, 否则返回 false。

注意:

如果 cpmset_first == NULL 或者 cpmset_second == NULL 则函数的行为是未定义的, 两个 multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 cpmset_first == cpmset_second 那么返回 true。

```
void multiset_swap(multiset_t* pmset_first, multiset_t* pmset_second);
```

描述:

交换两个 multiset 容器。

参数:

- pmset_first 指向 multiset 容器的指针。
- pmset_second 指向 multiset 容器的指针。

返回值:

无。

注意:

如果 pmset_first == NULL 或者 pmset_second == NULL 则函数的行为是未定义的，两个 multiset 必须已经初始化的，否则函数的行为是未定义的。两个 multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 multiset_equal(pmset_first, pmset_second)，函数不执行任何动作。

```
multiset_iterator_t multiset_insert(multiset_t* pmset_mset, elem);
```

描述:

向 multiset 容器中插入数据。

参数:

- pmset_mset 指向 multiset 容器的指针。
- elem 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t multiset_insert_hint(  
    multiset_t* pmset_mset, multiset_iterator_t it_hint, elem);
```

描述:

向 multiset 容器中的线索位置插入数据。

参数:

- pmset_mset 指向 multiset 容器的指针。
- it_hint 用户提供的线索位置。
- elem 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_multiset == NULL 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
void multiset_insert_range(  
    multiset_t* pmset_mset,  
    multiset_iterator_t it_begin, _multiset_iterator_t it_end);
```

描述:

向 multiset 中插入指定的数据区间。

参数:

- | | |
|------------|--------------------|
| pmset_mset | 指向 multiset 容器的指针。 |
| it_begin | 指定的数据区间的开头。 |
| it_end | 指定的数据区间的末尾。 |

返回值:

无。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end) 必须是有效的数据区间并且与保存在 multiset 容器中的数据类型相同, 否则函数的行为是未定义的。[it_begin, it_end) 必须不属于 pmset_mset, 否则函数的行为是未定义的。

```
void multiset_erase_pos(multiset_t* pmset_mset, multiset_iterator_t it_pos);
```

描述:

删除 multiset 容器中指定位置的数据。

参数:

- | | |
|------------|--------------------|
| pmset_mset | 指向 multiset 容器的指针。 |
| it_pos | 被删除的数据的位置。 |

返回值:

无。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 是属于 pmset_mset 容器的有效的迭代器, 否则函数的行为是未定义的。

```
void multiset_erase_range(  
    multiset_t* pmset_mset,  
    multiset_iterator_t it_begin, multiset_iterator_t it_end);
```

描述:

删除 multiset 中指定数据区间的数据。

参数:

- | | |
|------------|--------------------|
| pmset_mset | 指向 multiset 容器的指针。 |
| it_begin | 指定的数据区间的开头。 |
| it_end | 指定的数据区间的末尾。 |

返回值:

无。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end) 必须属于 pmset_mset, 否则函数的行为是未定义的。

```
void multiset_erase(multiset_t* pmset_mset, elem);
```

描述:

删除 multiset 中指定的数据。

参数:

pmset_mset 指向 multiset 容器的指针。

elem 要删除的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的, multiset 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

第五节 迭代器接口

set 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

create_set_iterator	创建 set 迭代器。
_set_iterator_get_value	获得 set 迭代器引用的数据。
_set_iterator_get_pointer	获得 set 迭代器引用的数据的指针。
_set_iterator_next	获得引用下一个数据的迭代器。
_set_iterator_prev	获得引用前一个数据的迭代器。
_set_iterator_equal	测试两个迭代器是否相等。
_set_iterator_distance	计算两个迭代器之间的距离。
_set_iterator_before	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
set_iterator_t create_set_iterator(void);
```

描述:

创建一个 set 迭代器。

参数:

无。

返回值:

set 迭代器。

注意:

返回的 set 迭代器并不是有效的迭代器, 它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作, 返回的迭代器供其他接口使用。

```
void _set_iterator_get_value(set_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter set 迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意:

it_iter 是有效的 set 迭代器, 否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的, pv_value 是能够保存下 it_iter 引用的数据的缓冲区, 否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _set_iterator_get_pointer(set_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter set 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 set 迭代器, 否则函数的行为是未定义的。

```
set_iterator_t _set_iterator_next(set_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter set 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 set 迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 set 有效的迭代器, 否则函数的行为是未定义的。

```
set_iterator_t _set_iterator_prev(set_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter set 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 set 迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 set 有效的迭代器, 否则函数的行为是未定义的。

```
bool_t _set_iterator_equal(set_iterator_t it_first, set_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first set 迭代器。
it_second set 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 set 容器的有效的 set 迭代器, 否则函数的行为是未定义的。两个 set 迭代器相等是指两个迭代器引用相同的数据。

```
int _set_iterator_distance(set_iterator_t it_first, set_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_iterfirst set 迭代器。
it_itersecond set 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 set 容器的有效的 set 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为 0。

```
bool_t _set_iterator_before(set_iterator_t it_first, set_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first set 迭代器。
it_second set 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 set 容器的有效的 set 迭代器, 否则函数的行为是未定义的。

multiset 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

create_multiset_iterator	创建 multiset 迭代器。
--------------------------	------------------

| _multiset_iterator_get_value | 获得 multiset 迭代器引用的数据。 |
| _multiset_iterator_get_pointer | 获得 multiset 迭代器引用的数据的指针。 |

<code>_multiset_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_multiset_iterator_prev</code>	获得引用前一个数据的迭代器。
<code>_multiset_iterator_equal</code>	测试两个迭代器是否相等。
<code>_multiset_iterator_distance</code>	计算两个迭代器之间的距离。
<code>_multiset_iterator_before</code>	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
multiset_iterator_t create_multiset_iterator(void);
```

描述:

创建一个 multiset 迭代器。

参数:

无。

返回值:

multiset 迭代器。

注意:

返回的 multiset 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _multiset_iterator_get_value(multiset_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

`it_iter` multiset 迭代器。

`pv_value` 保存数据的缓冲区。

返回值:

无。

注意:

`it_iter` 是有效的 multiset 迭代器，否则函数的行为是未定义的。`pv_value == NULL` 则函数的行为是未定义的，`pv_value` 是能够保存下 `it_iter` 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 `it_iter` 引用的数据被拷贝到 `pv_value` 指向的缓存区中。

```
const void* _multiset_iterator_get_pointer(multiset_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

`it_iter` multiset 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

`it_iter` 是有效的 multiset 迭代器，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_iterator_next(multiset_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter multiset 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 multiset 迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 multiset 有效的迭代器，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_iterator_prev(multiset_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter multiset 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 multiset 迭代器，否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 multiset 有效的迭代器，否则函数的行为是未定义的。

```
bool_t _multiset_iterator_equal(
    multiset_iterator_t it_first, multiset_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first multiset 迭代器。

it_second multiset 迭代器。

返回值:

如果两个迭代器相等则返回 true，否则返回 false。

注意:

两个 iter 是属于同一个 multiset 容器的有效的 multiset 迭代器，否则函数的行为是未定义的。两个 multiset 迭代器相等是指两个迭代器引用相同的数据。

```
int _multiset_iterator_distance(
    multiset_iterator_t it_first, multiset_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_iterfirst multiset 迭代器。

it_itersecond multiset 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 multiset 容器的有效的 multiset 迭代器，否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0，it_first 引用的数据在 it_second 后面那么结果<0，如果两个迭代器相等那么结果为 0。

```
bool_t _multiset_iterator_before(  
    multiset_iterator_t it_first, multiset_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first multiset 迭代器。
it_second multiset 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true，否则返回 false。

注意:

两个 iter 是属于同一个 multiset 容器的有效的 multiset 迭代器，否则函数的行为是未定义的。

第六节 内部和辅助接口

set 提供的内部接口是为了给外部接口使用。

_create_set	创建 set 容器。
_create_set_auxiliary	创建 set 容器的辅助函数。
_set_destroy_auxiliary	销毁 set 容器的辅助函数。
_set_find	在 set 容器中查找指定的数据。
_set_find_varg	在 set 容器中查找指定的数据，数据来自于可变参数列表。
_set_count	返回 set 容器中指定数据的数量。
_set_count_varg	返回 set 容器中指定数据的数量，数据来自于可变参数列表。
_set_lower_bound	返回指向第一个大于等于指定数据的迭代器。
_set_lower_bound_varg	返回指向第一个大于等于指定数据的迭代器，数据来自于可变参数列表。
_set_upper_bound	返回指向第一个大于指定数据的迭代器。
_set_upper_bound_varg	返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。
_set_equal_range	返回包含指定数据的数据区间。
_set_equal_range_varg	返回包含指定数据的数据区间，数据来自于可变参数列表。

<code>_set_erase</code>	删除 set 中指定的数据。
<code>_set_erase_varg</code>	删除 set 中指定的数据，数据来自于可变参数列表。
<code>_set_insert</code>	向 set 中插入数据，数据必须唯一。
<code>_set_insert_varg</code>	向 set 中插入数据，数据必须唯一，数据来自于可变参数列表。
<code>_set_insert_hint</code>	向 set 中插入数据，数据必须唯一。
<code>_set_insert_hint_varg</code>	向 set 中插入数据，数据必须唯一，数据来自于可变参数列表。
<code>_set_init_elem_auxiliary</code>	初始化数据的辅助函数。

函数原型

```
set_t* _create_set(const char* s_typename);
```

描述:

创建一个 set 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 set 容器的指针，否则返回 NULL。

注意:

`s_typename` != NULL，否则函数的行为是未定义的。

```
bool_t _create_set_auxiliary(set_t* pset_set, const char* s_typename);
```

描述:

创建一个 set 容器的辅助函数。

参数:

`pset_set` 没有创建的 set 容器。

`s_typename` set 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pset_set == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型，libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
void _set_destroy_auxiliary(set_t* pset_set);
```

描述:

销毁 set 容器的辅助函数。

参数:

`pset_set` set 容器。

返回值:

无。

注意:

如果 `pset_set == NULL` 或者 set 不是使用 `_create_set` 生成的则函数的行为是未定义的。

```
set_iterator_t _set_find(const set_t* cpset_set, ...);
```

描述:

在 set 容器中查找指定的数据。

参数:

cpset_set 指向 set 容器的指针。
... 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _set_count(const set_t* cpset_set, ...);
```

描述:

在 set 容器中查找指定的数据，数据来自于参数列表。

参数:

cpset_set 指向 set 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _set_count_varg(const set_t* cpset_set, ...);
```

描述:

返回 set 容器中指定数据的数量。

参数:

cpset_set 指向 set 容器的指针。
... 要查找的指定数据。

返回值:

返回 set 容器中指定数据的数量。

注意:

如果 cpset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _set_count_varg(const set_t* cpset_set, ...);
```

描述:

返回 set 容器中指定数据的数量，数据来自于参数列表。

参数:

cpset_set 指向 set 容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回 set 容器中指定数据的数量。

注意:

如果 `cpset_set == NULL` 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t _set_lower_bound(const set_t* cpset_set, ...);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

`cpset_set` 指 set 向容器的指针。

`...` 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cpset_set == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t _set_lower_bound_varg(const set_t* cpset_set, va_list val_elemlist);
```

描述:

返回指向第一个大于等于指定数据的迭代器, 数据来自于可变参数列表。

参数:

`cpset_set` 指 set 向容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cpset_set == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t _set_upper_bound(const set_t* cpset_set, ...);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

`cpset_set` 指 set 向容器的指针。

`...` 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cpset_set == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t _set_upper_bound_varg(const set_t* cpset_set, va_list val_elemlist);
```

描述:

 返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。

参数:

 cpset_set 指 set 向容器的指针。

 val_elemlist 用户指定的数据的参数列表。

返回值:

 返回指向第一个大于指定数据的迭代器。

注意:

 如果 cpset_set == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _set_equal_range(const set_t* cpset_set, ...);
```

描述:

 返回包含指定数据的数据区间。

参数:

 cpset_set 指向容器的指针。

 ... 要查找的指定数据。

返回值:

 返回包含指定数据的数据区间。

注意:

 如果 cpset_set == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _set_equal_range_varg(const set_t* cpset_set, va_list val_elemlist);
```

描述:

 返回包含指定数据的数据区间。

参数:

 cpset_set 指向容器的指针。

 val_elemlist 用户指定的数据的参数列表。

返回值:

 返回包含指定数据的数据区间。

注意:

 如果 cpset_set == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
void _set_erase(set_t* pset_set, ...);
```

描述:

 删除 set 中指定的数据。

参数:

 pset_set 指向 set 容器的指针。

... 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 pset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _set_erase_varg(set_t* pset_set, va_list val_elemlist);
```

描述:

删除 set 中指定的数据, 数据来自于可变参数列表。

参数:

pset_set 指向 set 容器的指针。

val_elemlist 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 pset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t _set_insert(set_t* pset_set, ...);
```

描述:

向 set 容器中插入唯一的数据。

参数:

pset_set 指向 set 容器的指针。

... 要查找的指定数据。

返回值:

指向插入的数据的迭代器, 如果插入失败则返回指向末尾的迭代器。

注意:

如果 pset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。插入的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t _set_insert_varg(set_t* pset_set, va_list val_elemlist);
```

描述:

向 set 容器中插入唯一的数据, 数据来自于可变参数列表。

参数:

pset_set 指向 set 容器的指针。

val_elemlist 用户指定的数据的参数列表。

返回值:

指向插入的数据的迭代器, 如果插入失败则返回指向末尾的迭代器。

注意:

如果 pset_set == NULL 则函数的行为是未定义的, set 容器必须已经初始化的, 否则函数的行为是未定义的。插入的数据必须与保存在 set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
set_iterator_t _set_insert_hint(set_t* pset_set, set_iterator_t it_hint, ...);
```

描述:

向 set 容器中插入唯一的数据。

参数:

pset_set	指向 set 容器的指针。
it_hint	线索位置。
...	要查找的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

```
set_iterator_t _set_insert_hint_varg(
    set_t* pset_set, set_iterator_t it_hint, va_list val_elemlist);
```

描述:

向 set 容器中插入唯一的数据，数据来自于可变参数列表。

参数:

pset_set	指向 set 容器的指针。
it_hint	线索位置。
val_elemlist	用户指定的数据的参数列表。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pset_set == NULL 则函数的行为是未定义的，set 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 set 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _set_init_elem_auxiliary(set_t* pset_set, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

pset_set	set 容器。
pv_value	初始化的数据。

返回值:

无。

注意:

如果 pset_set == NULL 或者 pv_value == NULL，那么函数的行为是未定义的。pset_set 必须是初始化的或者是使用 _create_set() 创建的，否则函数的行为是未定义的。

set 提供了少量的辅助函数。

```
_set_get_varg_value_auxiliary
```

根据 set 容器中数据的类型获得可变参数列表中的数据。

<code>_set_destroy_varg_value_auxiliary</code>	销毁从可变参数列表中获得的数据。
------------------------------------------------	------------------

```
void _set_get_varg_value_auxiliary(
    set_t* pset_set, va_list val_elemlist, void* pv_varg);
```

描述:

从可变参数列表中获得与 set 中的数据类型相同的数据。

参数:

<code>pset_set</code>	set 容器。
<code>val_elemlist</code>	可变参数列表。
<code>pv_varg</code>	保存数据缓的冲区。

返回值:

无。

注意:

如果 `pset_set == NULL` 或者 `pv_varg == NULL`, 那么函数的行为是未定义的。`pset_set` 必须是已经初始化或者是由 `create_set()` 创建的 set 容器, 否则函数的行为是未定义的。

```
void _set_destroy_varg_value_auxiliary(set_t* pset_set, void* pv_varg);
```

描述:

销毁与 set 中的数据类型相同的数据。

参数:

<code>pset_set</code>	set 容器。
<code>pv_varg</code>	保存数据的缓冲区。

返回值:

无。

注意:

如果 `pset_set == NULL` 或者 `pv_varg == NULL`, 那么函数的行为是未定义的。`pset_set` 必须是已经初始化或者是由 `create_set()` 创建的 set 容器, 否则函数的行为是未定义的。

`multiset` 提供的内部接口是为了给外部几口使用。

<code>_create_multiset</code>	创建 multiset 容器。
<code>_create_multiset_auxiliary</code>	创建 multiset 容器的辅助函数。
<code>_multiset_destroy_auxiliary</code>	销毁 multiset 容器的辅助函数。
<code>_multiset_find</code>	在 multiset 容器中查找指定的数据。
<code>_multiset_find_varg</code>	在 multiset 容器中查找指定的数据, 数据来自于可变参数列表。
<code>_multiset_count</code>	返回 multiset 容器中指定数据的数量。
<code>_multiset_count_varg</code>	返回 multiset 容器中指定数据的数量, 数据来自于可变参数列表。
<code>_multiset_lower_bound</code>	返回指向第一个大于等于指定数据的迭代器。
<code>_multiset_lower_bound_varg</code>	返回指向第一个大于等于指定数据的迭代器, 数据来自于可变参数列表。
<code>_multiset_upper_bound</code>	返回指向第一个大于指定数据的迭代器。

<code>_multiset_upper_bound_varg</code>	返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。
<code>_multiset_equal_range</code>	返回包含指定数据的数据区间。
<code>_multiset_equal_range_varg</code>	返回包含指定数据的数据区间，数据来自于可变参数列表。
<code>_multiset_erase</code>	删除 multiset 中指定的数据。
<code>_multiset_erase_varg</code>	删除 multiset 中指定的数据，数据来自于可变参数列表。
<code>_multiset_insert</code>	向 multiset 中插入数据。
<code>_multiset_insert_varg</code>	向 multiset 中插入数据，数据来自于可变参数列表。
<code>_multiset_insert_hint</code>	向 multiset 中插入数据。
<code>_multiset_insert_hint_varg</code>	向 multiset 中插入数据，数据来自于可变参数列表。
<code>_multiset_init_elem_auxiliary</code>	初始化数据的辅助函数。

函数原型

```
multiset_t* _create_multiset(const char* s_typename);
```

描述:

创建一个 multiset 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 multiset 容器的指针，否则返回 NULL。

注意:

`s_typename != NULL`，否则函数的行为是未定义的。

```
bool_t _create_multiset_auxiliary(multiset_t* pmset_mset, const char* s_typename);
```

描述:

创建一个 multiset 容器的辅助函数。

参数:

`pmset_mset` 没有创建的 multiset 容器。

`s_typename` multiset 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pmset_mset == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型，libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
void _multiset_destroy_auxiliary(multiset_t* pmset_mset);
```

描述:

销毁 multiset 容器的辅助函数。

参数:

`pmset_mset` multiset 容器。

返回值:

无。

注意:

如果 `pmset_mset == NULL` 或者 `multiset` 不是使用 `_create_multiset` 生成的则函数的行为是未定义的。

```
multiset_iterator_t _multiset_find(const multiset_t* cpmset_mset, ...);
```

描述:

在 `multiset` 容器中查找指定的数据。

参数:

`cpmset_mset` 指向 `multiset` 容器的指针。

`...` 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的，`multiset` 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 `multiset` 容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_find_varg(
    const multiset_t* cpmset_mset, va_list val_elemlist);
```

描述:

在 `multiset` 容器中查找指定的数据，数据来自于参数列表。

参数:

`cpmset_mset` 指向 `multiset` 容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的，`multiset` 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 `multiset` 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _multiset_count(const multiset_t* cpmset_mset, ...);
```

描述:

返回 `multiset` 容器中指定数据的数量。

参数:

`cpmset_mset` 指向 `multiset` 容器的指针。

`...` 要查找的指定数据。

返回值:

返回 `multiset` 容器中指定数据的数量。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的，`multiset` 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 `multiset` 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _multiset_count_varg(const multiset_t* cpmset_mset, va_list val_elemlist);
```

描述:

返回 multiset 容器中指定数据的数量，数据来自于参数列表。

参数:

cpmset_mset 指向 multiset 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回 multiset 容器中指定数据的数量。

注意:

如果 cpmset_mset == NULL 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_lower_bound(const multiset_t* cpmset_mset, ...);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cpmset_mset 指 multiset 向容器的指针。
... 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cpmset_mset == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_lower_bound_varg(  
    const multiset_t* cpmset_mset, va_list val_elemlist);
```

描述:

返回指向第一个大于等于指定数据的迭代器，数据来自于可变参数列表。

参数:

cpmset_mset 指 multiset 向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cpmset_mset == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_upper_bound(const multiset_t* cpmset_mset, ...);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cpmset_mset 指 multiset 向容器的指针。

... 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_upper_bound_varg(
    const multiset_t* cpmset_mset, va_list val_elemlist);
```

描述:

返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。

参数:

`cpmset_mset` 指 `multiset` 向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _multiset_equal_range(const multiset_t* cpmset_mset, ...);
```

描述:

返回包含指定数据的数据区间。

参数:

`cpmset_mset` 指向容器的指针。
... 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _multiset_equal_range_varg(
    const multiset_t* cpmset_mset, va_list val_elemlist);
```

描述:

返回包含指定数据的数据区间。

参数:

`cpmset_mset` 指向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回包含指定数据的数据区间。

注意:

如果 `cpmset_mset == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
void _multiset_erase(multiset_t* pmset_mset, ...);
```

描述:

删除 multiset 中指定的数据。

参数:

`pmset_mset` 指向 multiset 容器的指针。
... 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 `pmset_mset == NULL` 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _multiset_erase_varg(multiset_t* pmset_mset, va_list val_elemlist);
```

描述:

删除 multiset 中指定的数据，数据来自于可变参数列表。

参数:

`pmset_mset` 指向 multiset 容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 `pmset_mset == NULL` 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_insert(multiset_t* pmset_mset, ...);
```

描述:

向 multiset 容器中插入唯一的数据。

参数:

`pmset_mset` 指向 multiset 容器的指针。
... 要查找的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 `pmset_mset == NULL` 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_insert_varg(  
    multiset_t* pmset_mset, va_list val_elemlist);
```

描述:

向 multiset 容器中插入唯一的数据，数据来自于可变参数列表。

参数:

- pmset_mset 指向 multiset 容器的指针。
- val_elemlist 用户指定的数据的参数列表。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_insert_hint(  
    multiset_t* pmset_mset, multiset_iterator_t it_hint, ...);
```

描述:

向 multiset 容器中插入唯一的数据。

参数:

- pmset_mset 指向 multiset 容器的指针。
- it_hint 线索位置。
- ... 要查找的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
multiset_iterator_t _multiset_insert_hint_varg(  
    multiset_t* pmset_mset, multiset_iterator_t it_hint, va_list val_elemlist);
```

描述:

向 multiset 容器中插入唯一的数据，数据来自于可变参数列表。

参数:

- pmset_mset 指向 multiset 容器的指针。
- it_hint 线索位置。
- val_elemlist 用户指定的数据的参数列表。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pmset_mset == NULL 则函数的行为是未定义的，multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _multiset_init_elem_auxiliary(multiset_t* pmset_mset, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

pmset_mset multiset 容器。

pv_value 初始化的数据。

返回值:

无。

注意:

如果 pmset_mset == NULL 或者 pv_value == NULL，那么函数的行为是未定义的。pmset_mset 必须是初始化的或者是使用 create_multiset() 创建的，否则函数的行为是未定义的。

multiset 提供了少量的辅助函数。

_multiset_get_varg_value_auxiliary	根据 multiset 容器中数据的类型获得可变参数列表中的数据。
------------------------------------	-----------------------------------

_multiset_destroy_varg_value_auxiliary	销毁从可变参数列表中获得的数据。
----------------------------------------	------------------

```
void _multiset_get_varg_value_auxiliary(  
    multiset_t* pmset_mset, va_list val_elemlist, void* pv_varg);
```

描述:

从可变参数列表中获得与 multiset 中的数据类型相同的数据。

参数:

pmset_multiset multiset 容器。

val_elemlist 可变参数列表。

pv_varg 保存数据缓的冲区。

返回值:

无。

注意:

如果 pmset_mset == NULL 或者 pv_varg == NULL，那么函数的行为是未定义的。pmset_mset 必须是已经初始化或者是由 create_multiset() 创建的 multiset 容器，否则函数的行为是未定义的。

```
void _multiset_destroy_varg_value_auxiliary(multiset_t* pmset_mset, void* pv_varg);
```

描述:

销毁与 multiset 中的数据类型相同的数据。

参数:

pmset_mset mset 容器。

pv_varg 保存数据的缓冲区。

返回值:

无。

注意:

如果 pset_mset == NULL 或者 pv_varg == NULL，那么函数的行为是未定义的。pmset_mset 必须是已经初始化或者是由 create_multiset() 创建的 multiset 容器，否则函数的行为是未定义的。

第十七章 pair

第一节 pair 的机制

Pair 是一个 key/value 的对，它主要被用来作为 map 的内部存储。

第二节 pair 的代码

pair 的代码结构如下

```
typedef struct _tagpair
{
    _typeinfo_t      _t_typeinfofirst;
    _typeinfo_t      _t_typeinfosecond;

    void*           _pv_first;
    void*           _pv_second;

    binary_function_t _t_mapkeycompare;
    binary_function_t _t_mapvaluecompare;
}pair_t;
```

_t_mapkeycompare, _t_mapvaluecompare 这两个成员主要是在 map 和 multimap 中使用。

第三节 外部接口

create_pair	创建 pair。
pair_init	初始化 pair。
pair_init_elem	使用指定的数据初始化 pair。
pair_init_copy	pair 的拷贝初始化函数。
pair_destroy	销毁 pair。
pair_make	使用指定的数据给 pair 赋值。
pair_assign	pair 的赋值函数。

pair_first	获得 pair 的第一个数据。
pair_second	获得 pair 的第二个数据。
pair_equal	判断两个 pair 时候相等。
pair_not_equal	判断两个 pair 是否不等。
pair_less	判断第一个 pair 是否小于第二个 pair。
pair_less_equal	判断第一个 pair 是否小于等于第二个 pair。
pair_greater	判断第一个 pair 是否大于第二个 pair。
pair_greater_equal	判断第一个 pair 是否大于等于第二个 pair。

函数原型

```
pair_t* create_pair(typename);
```

描述:

创建一个 pair。

参数:

typename 类型描述。

返回值:

成功返回指向 pair 容器的指针, 否则返回 NULL。

注意:

关于类型描述参考第四章, 创建失败返回 NULL。

```
void pair_init(pair_t* ppair_pair);
```

描述:

初始化 pair。

参数:

ppair_pair pair 容器。

返回值:

无。

注意:

ppair_pair == NULL 则函数的行为是未定义的, ppair_pair 必须是使用 create_pair() 创建的, 否则函数的行为是未定义的。

```
void pair_init_elem(pair_t* ppair_pair, first_elem, second_elem);
```

描述:

使用指定的数据初始化 pair。

参数:

ppair_pair 目的 pair 容器。

first_elem 第一个指定的数据。

second_elem 第二个指定的数据。

返回值:

无。

注意:

ppair_pair == NULL 则函数的行为是未定义的, ppair_pair 必须是使用 create_pair() 创建的, 否则函数的行为是未定义的。

```
void pair_init_copy(pair_t* ppair_dest, const pair_t* cppair_src);
```

描述:

使用已经存在的 pair 初始化 pair。

参数:

ppair_dest 目的 pair。

cppair_src 源 pair。

返回值:

无。

注意:

ppair_dest == NULL 或者 cppair_src == NULL 则函数的行为是未定义的, ppair_dest 必须是使用 create_pair() 创建的, cppair_src 必须是已经初始化, 否则函数的行为是未定义的。ppair_dest 和 cppair_src 保存的数据类型必须相同, 否则函数的行为是未定义。

```
void pair_destroy(pair_t* ppair_pair);
```

描述:

销毁创建的 pair。

参数:

ppair_pair pair。

返回值:

无。

注意:

如果 ppair_pair == NULL 则函数的行为是未定义的, ppair_pair 必须是初始化或者是使用 create_pair() 创建的, 否则函数的行为是未定义的。

```
void pair_make(pair_t* ppair_pair, first_elem, second_elem);
```

描述:

使用指定的数据为 pair 赋值。

参数:

ppair_pair 指向被赋值的 pair 的指针。

first_elem 第一个指定的数据。

second_elem 第二个指定的数据。

返回值:

无。

注意:

如果 ppair_pair == NULL 则函数的行为是未定义的, pair 必须已经初始化的, 否则函数的行为是未定义的。

```
void pair_assign(pair_t* ppair_dest, const pair_t* cppair_src);
```

描述:

使用一个 pair 为另一个 pair 赋值。

参数:

ppair_dest 指向被赋值的 pair 容器的指针。
cppair_src 指向赋值的 pair 容器的指针。

返回值:

无。

注意:

如果 `ppair_dest == NULL` 或者 `cppair_src == NULL` 则函数的行为是未定义的，两个 pair 必须已经初始化的，否则函数的行为是未定义的。两个 pair 容器中保存的数据类型不同函数的行为是未定义的。如果 `pair_equal(ppair_dest, cppair_src)` 那么函数不做任何动作。

```
void* pair_first(const pair_t* cppair_pair);
```

描述:

访问 pair 的第一个数据。

参数:

cppair_pair 指向 pair 的指针。

返回值:

pair 的第一个数据的指针。

注意:

如果 `cppair_pair == NULL` 则函数的行为是未定义的，pair 必须已经初始化的，否则函数的行为是未定义的。

```
void* pair_second(const pair_t* cppair_pair);
```

描述:

访问 pair 的第二个数据。

参数:

cppair_pair 指向 pair 的指针。

返回值:

pair 的第二个数据的指针。

注意:

如果 `cppair_pair == NULL` 则函数的行为是未定义的，pair 必须已经初始化的，否则函数的行为是未定义的。

```
bool_t pair_equal(const pair_t* cppair_first, const pair_t* cppair_second);
```

描述:

测试两个 pair 是否相等。

参数:

cppair_first 指向 pair 的指针。
cppair_second 指向 pair 的指针。

返回值:

如果两个 pair 相等则返回 `true`，否则返回 `false`。

注意:

如果 `cppair_first == NULL` 或者 `cppair_second == NULL` 则函数的行为是未定义的，两个 pair 必须已经初始化的，

否则函数的行为是未定义的。两个 pair 中保存的数据类型不同认为这两个不等。两个 pair 相等是指 pair 中的数据对应相等。如果 `cppair_first == cppair_second` 那么返回 true。

```
bool_t pair_not_equal(const pair_t* cppair_first, const pair_t* cppair_second);
```

描述:

测试两个 pair 是否不等。

参数:

<code>cppair_first</code>	指向 pair 的指针。
<code>cppair_second</code>	指向 pair 的指针。

返回值:

如果两个 pair 不等则返回 true, 否则返回 false。

注意:

如果 `cppair_first == NULL` 或者 `cppair_second == NULL` 则函数的行为是未定义的, 两个 pair 必须已经初始化的, 否则函数的行为是未定义的。两个 pair 中保存的数据类型不同认为两个 pair 不等。两个 pair 不等是指 pair 中的对应的数据不相等。如果 `cppair_first == cppair_second` 那么返回 false。

```
bool_t pair_less(const pair_t* cppair_first, const pair_t* cppair_second);
```

描述:

测试第一个 pair 是否小于第二个 pair。

参数:

<code>cppair_first</code>	指向 pair 的指针。
<code>cppair_second</code>	指向 pair 的指针。

返回值:

如果第一个 pair 小于第二个 pair 则返回 true, 否则返回 false。

注意:

如果 `cppair_first == NULL` 或者 `cppair_second == NULL` 则函数的行为是未定义的, 两个 pair 必须已经初始化的, 否则函数的行为是未定义的。两个 pair 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 pair 中的数据小于第二个 pair 中对应的数据则返回 true, 如果大于则返回 false。如果 `cppair_first == cppair_second` 那么返回 false。

```
bool_t pair_less_equal(const pair_t* cppair_first, const pair_t* cppair_second);
```

描述:

测试第一个 pair 是否小于等于第二个 pair。

参数:

<code>cppair_first</code>	指向 pair 的指针。
<code>cppair_second</code>	指向 pair 的指针。

返回值:

如果第一个 pair 小于等于第二个 pair 则返回 true, 否则返回 false。

注意:

如果 `cppair_first == NULL` 或者 `cppair_second == NULL` 则函数的行为是未定义的, 两个 pair 必须已经初始化的, 否则函数的行为是未定义的。两个 pair 中保存的数据类型不同函数的行为是未定义的。如果 `cppair_first == cppair_second` 那么返回 true。

```
bool_t pair_greater(const pair_t* cppair_first, const pair_t* cppair_second);
```

描述:

测试第一个 pair 是否大于第二个 pair。

参数:

cppair_first 指向 pair 的指针。
cppair_second 指向 pair 的指针。

返回值:

如果第一个 pair 大于第二个 pair 则返回 true, 否则返回 false。

注意:

如果 cppair_first == NULL 或者 cppair_second == NULL 则函数的行为是未定义的, 两个 pair 必须已经初始化的, 否则函数的行为是未定义的。两个 pair 中保存的数据类型不同函数的行为是未定义的。如果第一 pair 中的数据大于第二个 pair 中的对应的数据则返回 true, 小于则返回 false。如果 cppair_first == cppair_second 那么返回 false。

```
bool_t pair_greater_equal(const pair_t* cppair_first, const pair_t* cppair_second);
```

描述:

测试第一个 pair 是否大于等于第二个 pair。

参数:

cppair_first 指向 pair 的指针。
cppair_second 指向 pair 的指针。

返回值:

如果第一个 pair 大于等于第二个 pair 则返回 true, 否则返回 false。

注意:

如果 cppair_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 pair 必须已经初始化的, 否则函数的行为是未定义的。两个 pair 容器中保存的数据类型不同函数的行为是未定义的。如果 cppair_first == cppair_second 那么返回 true。

第四节 内部和辅助接口

pair 提供了很多内部接口。

_pair_is_created	判断 pair 是否是使用 create_pair 创建的。
_pair_is_initiated	判断 pair 是否已经初始化。
_create_pair	创建 pair。
_create_pair_auxiliary	创建 pair 的辅助函数。
_pair_make_first	给 pair 的第一个数据赋值。
_pair_make_second	给 pair 的第二个数据赋值。
_pair_destroy_auxiliary	销毁 pair 的辅助函数。

函数原型

```
bool_t _pair_is_created(const pair_t* cppair_pair);
```

描述:

测试一个 pair 是否是使用 create_pair 创建的。

参数:

cppair_pair pair 容器。

返回值:

如果 pair 是使用 create_pair 创建的则返回 true, 否则返回 false。

注意:

如果 cppair_pair == NULL, 函数的行为是未定义的。

```
bool_t _pair_is_inited(const pair_t* cppair_pair);
```

描述:

测试一个 pair 是否已经初始化。

参数:

cppair_pair pair 容器。

返回值:

如果 pair 已经初始化了, 返回 true, 否则返回 false。

注意:

如果 cppair_pair == NULL, 函数的行为是未定义的。

```
pair_t* _create_pair(const char* s_typename);
```

描述:

创建一个 pair 迭代器。

参数:

s_typename 类型描述。

返回值:

成功返回指向 pair 容器的指针, 否则返回 NULL。

注意:

s_typename != NULL, 否则函数的行为是未定义的。

```
bool_t _create_pair_auxiliary(pair_t* ppair_pair, const char* s_typename);
```

描述:

创建一个 pair 容器的辅助函数。

参数:

ppair_pair 没有创建的 pair 容器。

s_typename pair 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 ppair_pair == NULL 或者 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型 libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _pair_destroy_auxiliary(pair_t* ppair_pair);
```

描述:

销毁 pair 容器的辅助函数。

参数:

ppair_pair pair 容器。

返回值:

无。

注意:

如果 ppair_pair == NULL 或者 pair 不是使用_create_pair 生成的则函数的行为是未定义的。

```
void _pair_make_first(pair_t* ppair_pair, ...);
```

描述:

为 pair 的第一个数据赋值。

参数:

ppair_pair pair 容器。
... 用户指定的数据。

返回值:

无。

注意:

如果 ppair_pair == NULL 或者 pair 是为初始化的则函数的行为是未定义的。

```
void _pair_make_second(pair_t* ppair_pair, ...);
```

描述:

为 pair 的第二个数据赋值。

参数:

ppair_pair pair 容器。
... 用户指定的数据。

返回值:

无。

注意:

如果 ppair_pair == NULL 或者 pair 是为初始化的则函数的行为是未定义的。

pair 提供了少量的辅助函数。

_pair_same_type

判断两个 pair 类型是否相同。

函数原型

```
bool_t _pair_same_type(const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

ppair_first 第一个 pair 类型。
ppair_second 第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true，否则返回 false。

注意:

如果 ppair_first == NULL 或者 ppair_second == NULL，那么函数的行为是未定义的。

第十八章 map 和 multimap

第一节 map 和 multimap 的机制

map 和 multimap 是对于映射的抽象，它们保存的数据类型都是 key/value 的 pair_t 类型，在 map 中 key 必须唯一，multimap 中 key 可以不唯一，对 value 没有唯一性的要求。Map 和 multimap 都是使用 avl tree 或者 rb tree 实现的。

第二节 map 和 multimap 的迭代器

Map 和 multimap 的迭代器使用 avl tree 或者 rb tree 的迭代器实现。

第三节 map 和 multimap 的代码结构

map 的代码结构如下

```
typedef struct _tagmap
{
    pair_t          _t_pair;
    binary_function_t _t_keycompare; /* for init ex */
    binary_function_t _t_valuecompare;
#ifndef CSTL_MAP_AVL_TREE
    _avl_tree_t      _t_tree;
#else
    _rb_tree_t       _t_tree;
#endif
}map_t;
```

multimap 的代码结构如下

```
typedef struct _tagmultimap
{
    pair_t          _t_pair;
    binary_function_t _t_keycompare; /* for init ex */
    binary_function_t _t_valuecompare;
#ifndef CSTL_MULTIMAP_AVL_TREE
    _avl_tree_t _t_tree;
```

```

#else
    _rb_tree_t _t_tree;
#endif
}multimap_t;

```

可以通过配置来决定使用 avl tree 还是使用 rb tree 作为底层实现。

第四节 外部接口

map 提供的外部接口提供给用户使用。

create_map	创建 map 容器。
map_init	初始化 map 容器。
map_init_ex	使用自定义比较规则初始化 map 容器。
map_init_copy	使用存在的 map 容器进行初始化。
map_init_copy_range	使用指定的数据区间进行初始化。
map_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
map_destroy	销毁 map 容器。
map_assign	使用存在的 map 容器赋值。
map_size	返回 map 容器中数据的数量。
map_empty	判断 map 容器是否为空。
map_max_size	返回 map 容器中能够保存数据的最大数量。
map_begin	返回指向 map 容器的第一个数据的迭代器。
map_end	返回指向 map 容器末尾的迭代器。
map_key_comp	返回 map 容器中的键比较规则。
map_value_comp	返回 map 容器中的数据比较规则。
map_find	在 map 容器中查找指定的数据。
map_clear	清空 map 容器。
map_count	返回 map 容器中指定数据的数量。
map_lower_bound	返回指向第一个大于等于指定数据的迭代器。
map_upper_bound	返回指向第一个大于指定数据的迭代器。
map_equal_range	返回包含指定数据的数据区间。
map_equal	测试两个 map 容器是否相等。
map_not_equal	测试两个 map 容器是否不等。

map_less	测试第一个 map 容器是否小于第二个 map 容器。
map_less_equal	测试第一个 map 容器是否小于等于第二个 map 容器。
map_greater	测试第一个 map 容器是否大于第二个 map 容器。
map_greater_equal	测试第一个 map 容器是否大于等于第二个 map 容器。
map_swap	交换两个 map 容器。
map_insert	向 map 中插入数据，数据必须唯一。
map_insert_hint	向 map 中线索位置插入数据，数据必须唯一。
map_insert_range	向 map 中插入指定的数据区间，数据必须唯一。
map_erase_pos	删除 map 中指定位置的数据。
map_erase_range	删除 map 中指定数据区间的数据。
map_erase	删除 map 中指定的数据。
map_at	使用下标对 map 中的数据进行访问。

函数原型

```
map_t* create_map(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

typename 类型描述。

返回值:

成功返回指向 map 容器的指针，否则返回 NULL。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void map_init(map_t* pmap_map);
```

描述:

初始化 map 迭代器。

参数:

pmap_map map 容器。

返回值:

无。

注意:

pmap_map == NULL 则函数的行为是未定义的，pmap_map 必须是使用 create_map() 创建的，否则函数的行为是未定义的，这个函数使用类型默认的比较函数。

```
void map_init_copy(map_t* pmap_dest, const map_t* cpmap_src);
```

描述:

使用已经存在的 map 初始化 map 迭代器。

参数:

 pmap_dest 目的 map 容器。
 cpmapt_src 源 map 容器。

返回值:

 无。

注意:

 pmap_dest == NULL 或者 cpmapt_src == NULL 则函数的行为是未定义的, pmap_dest 必须是使用 create_map() 创建的, cpmapt_src 必须是已经初始化, 否则函数的行为是未定义的。pmap_dest 和 cpmapt_src 保存的数据类型必须相同, 否则函数的行为是未定义。

```
void map_init_copy_range(  
  map_t* pmap_dest, map_iterator_t it_begin, map_iterator_t it_end);
```

描述:

 使用指定的数据区间初始化 rb tree 迭代器。

参数:

 pmap_dest 目的 map 容器。
 it_begin 数据区间的开始。
 it_end 数据区间的末尾。

返回值:

 无。

注意:

 如果 pmap_dest == NULL 则函数的行为是未定义的, pmap_dest 必须是使用 create_map() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 map 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void map_init_copy_range_ex(  
  map_t* pt_dest, map_iterator_t it_begin, map_iterator_t it_end,  
  binary_function_t t_compare);
```

描述:

 使用指定的数据区间和比较函数初始化 map 迭代器。

参数:

 pmap_dest 目的 map 容器。
 it_begin 数据区间的开始。
 it_end 数据区间的末尾。
 t_compare 比较函数。

返回值:

 无。

注意:

 如果 pmap_dest == NULL 则函数的行为是未定义的, pmap_dest 必须是使用 create_map() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 map 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 t_compare == NULL 则使用类型默认的比较函数。

```
void map_destroy(map_t* pmap_map);
```

描述:

销毁创建的 map 容器。

参数:

pmap_map map 容器。

返回值:

无。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的, pmap_map 必须是初始化或者是使用 create_map() 创建的, 否则函数的行为是未定义的。

```
void map_assign(map_t* pmap_dest, const map_t* cpmap_src);
```

描述:

使用一个 map 为另一个 map 赋值。

参数:

pmap_dest 指向被赋值的 map 容器的指针。

cpmap_src 指向赋值的 map 容器的指针。

返回值:

无。

注意:

如果 pmap_dest == NULL 或者 cpmap_src == NULL 则函数的行为是未定义的, 两个 map 必须已经初始化的, 否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同函数的行为是未定义的。如果 map_equal(pmap_dest, cpmap_src)那么函数不做任何动作。

```
size_t map_size(const map_t* cpmap_map);
```

描述:

获得 map 容器中保存的数据的个数。

参数:

cpmap_map map 容器。

返回值:

map 容器中保存的数据的个数。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, cpmap_map 必须是已经初始化的 map 容器, 否则函数的行为是未定义的。

```
bool_t map_empty(const map_t* cpmap_map);
```

描述:

测试 map 容器是否为空。

参数:

cpmap_map map 容器。

返回值:

如果 map 容器为空则返回 true, 否则返回 false。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, `cpmap_map` 必须是已经初始化的 map 容器, 否则函数的行为是未定义的。

```
size_t map_max_size(const map_t* cpmap_map);
```

描述:

获得 map 容器中数据的最大数量。

参数:

`cpmap_map` map 容器。

返回值:

map 容器中保存的数据的个数的最大值。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, `cpmap_map` 必须是已经初始化的 map 容器, 否则函数的行为是未定义的。

```
map_iterator_t map_begin(const map_t* cpmap_map);
```

描述:

获得引用 map 容器中第一数据的迭代器。

参数:

`cpmap_map` 指向 map 容器的指针。

返回值:

返回引用 map 容器中第一个数据的迭代器。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。如果 map 容器为空, 则返回值与 `map_end(cpmap_map)` 相等。

```
map_iterator_t map_end(const map_t* cpmap_map);
```

描述:

获得引用 map 容器末尾的迭代器。

参数:

`cpmap_map` 指向 map 容器的指针。

返回值:

返回引用 map 容器末尾的迭代器。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t map_key_comp(const map_t* cpmap_map);
```

描述:

返回 map 容器中的键比较规则。

参数:

`cpmap_map` 指向 map 容器的指针。

返回值:

返回 map 容器中的数据比较规则。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t map_value_comp(const map_t* cpmap_map);
```

描述:

返回 map 容器中的数据比较规则。

参数:

cpmap_map 指向 map 容器的指针。

返回值:

返回 map 容器中的数据比较规则。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
map_iterator_t map_find(const map_t* cpmap_map, elem);
```

描述:

在 map 容器中查找指定的数据。

参数:

cpmap_map 指向 map 容器的指针。

elem 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。

要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void map_clear(map_t* pmap_map);
```

描述:

删除 map 容器中的所有数据。

参数:

pmap_map 指向 map 容器的指针。

返回值:

无。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
size_t map_count(const map_t* cpmap_map, elem);
```

描述:

返回 map 容器中指定数据的数量。

参数:

cpmap_map 指向 map 容器的指针。

elem 要查找的指定数据。

返回值:

返回 map 容器中指定数据的数量。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
map_iterator_t map_lower_bound(const map_t* cpmap_map, elem);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cpmap_map 指向 map 容器的指针。

elem 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
map_iterator_t map_upper_bound(const map_t* cpmap_map, elem);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cpmap_map 指向 map 容器的指针。

elem 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t map_equal_range(const map_t* cpmap_map, elem);
```

描述:

返回包含指定数据的数据区间。

参数:

cpmap_map 指向 map 容器的指针。

elem 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
bool_t map_equal(const map_t* cpmap_first, const map_t* cpmap_second);
```

描述:

测试两个 map 容器是否相等。

参数:

cpmap_first 指向 map 容器的指针。
cpmap_second 指向 map 容器的指针。

返回值:

如果两个 map 容器相等则返回 true, 否则返回 false。

注意:

如果 cpmap_first == NULL 或者 cpmap_second == NULL 则函数的行为是未定义的, 两个 map 必须已经初始化的, 否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同认为这两个容器不等。两个 map 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cpmap_first == cpmap_second 那么返回 true。

```
bool_t map_not_equal(const map_t* cpmap_first, const map_t* cpmap_second);
```

描述:

测试两个 map 容器是否不等。

参数:

cpmap_first 指向 map 容器的指针。
cpmap_second 指向 map 容器的指针。

返回值:

如果两个 map 容器不等则返回 true, 否则返回 false。

注意:

如果 cpmap_first == NULL 或者 cpmap_second == NULL 则函数的行为是未定义的, 两个 map 必须已经初始化的, 否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同认为两个 map 容器不等。两个 map 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cpmap_first == cpmap_second 那么返回 false。

```
bool_t map_less(const map_t* cpmap_first, const map_t* cpmap_second);
```

描述:

测试第一个 map 是否小于第二个 map。

参数:

cpmap_first 指向 map 容器的指针。
cpmap_second 指向 map 容器的指针。

返回值:

如果第一个 map 容器小于第二个 map 容器则返回 true, 否则返回 false。

注意:

如果 cpmap_first == NULL 或者 cpmap_second == NULL 则函数的行为是未定义的, 两个 map 必须已经初始化的, 否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 map 中的数据小于第二个 map 中对应的数据则返回 true, 如果大于则返回 false, 当两个 map 中的对应数据相等, 第一个 map 中的数据数量小于第二个 map 中数据的数量返回 true, 否则返回 false。如果 cpmap_first == cpmap_second 那么返回 false。

```
bool_t map_less_equal(const map_t* cpmap_first, const map_t* cpmap_second);
```

描述:

测试第一个 map 是否小于等于第二个 map。

参数:

cpmap_first 指向 map 容器的指针。
cpmap_second 指向 map 容器的指针。

返回值:

如果第一个 map 容器小于等于第二个 map 容器则返回 true, 否则返回 false。

注意:

如果 cpmap_first == NULL 或者 cpmap_second == NULL 则函数的行为是未定义的, 两个 map 必须已经初始化的, 否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同函数的行为是未定义的。如果 cpmap_first == cpmap_second 那么返回 true。

```
bool_t map_greater(const map_t* cpmap_first, const map_t* cpmap_second);
```

描述:

测试第一个 map 是否大于第二个 map。

参数:

cpmap_first 指向 map 容器的指针。
cpmap_second 指向 map 容器的指针。

返回值:

如果第一个 map 容器大于第二个 map 容器则返回 true, 否则返回 false。

注意:

如果 cpmap_first == NULL 或者 cpmap_second == NULL 则函数的行为是未定义的, 两个 map 必须已经初始化的, 否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同函数的行为是未定义的。如果第一 map 中的数据大于第二个 map 中的对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 map 中数据的数量大于第二个 map 中数据的数量的时候返回 true 否则返回 false。如果 cpmap_first == cpmap_second 那么返回 false。

```
bool_t map_greater_equal(const map_t* cpmap_first, const map_t* cpmap_second);
```

描述:

测试第一个 map 是否大于等于第二个 map。

参数:

cpmap_first 指向 map 容器的指针。
cpmap_second 指向 map 容器的指针。

返回值:

如果第一个 map 容器大于等于第二个 map 容器则返回 true, 否则返回 false。

注意:

如果 cpmap_first == NULL 或者 cpmap_second == NULL 则函数的行为是未定义的, 两个 map 必须已经初始化的, 否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同函数的行为是未定义的。如果 cpmap_first == cpmap_second 那么返回 true。

```
void map_swap(map_t* pmap_first, map_t* pmap_second);
```

描述:

交换两个 map 容器。

参数:

- | | |
|-------------|---------------|
| pmap_first | 指向 map 容器的指针。 |
| pmap_second | 指向 map 容器的指针。 |

返回值:

无。

注意:

如果 pmap_first == NULL 或者 pmap_second == NULL 则函数的行为是未定义的，两个 map 必须已经初始化的，否则函数的行为是未定义的。两个 map 容器中保存的数据类型不同函数的行为是未定义的。如果 map_equal(pmap_first, pmap_second)，函数不执行任何动作。

```
map_iterator_t map_insert(map_t* pmap_map, const pair_t* cppair_pair);
```

描述:

向 map 容器中插入唯一的数据。

参数:

- | | |
|-------------|---------------|
| pmap_map | 指向 map 容器的指针。 |
| cppair_pair | 要插入的指定数据。 |

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的，map 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 map 容器中的数据类型相同，否则函数的行为是未定义的。

```
map_iterator_t map_insert_hint(  
    map_t* pmap_map, map_iterator_t it_hint, const pair_t* cppair_pair);
```

描述:

向 map 容器中的线索位置插入唯一的数据。

参数:

- | | |
|-------------|---------------|
| pmap_map | 指向 map 容器的指针。 |
| it_hint | 用户提供的线索位置。 |
| cppair_pair | 要插入的指定数据。 |

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_map == NULL 则函数的行为是未定义的，map 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 map 容器中的数据类型相同，否则函数的行为是未定义的。

```
void map_insert_range(  
    map_t* pmap_map, map_iterator_t it_begin, map_iterator_t it_end);
```

描述:

向 map 中插入指定的数据区间，数据必须唯一。

参数:

- | | |
|----------|---------------|
| pmap_map | 指向 map 容器的指针。 |
|----------|---------------|

it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。
[it_begin, it_end) 必须是有效的数据区间并且与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。
[it_begin, it_end) 必须不属于 pmap_map, 否则函数的行为是未定义的。

```
void map_erase_pos(map_t* pmap_map, map_iterator_t it_pos);
```

描述:

删除 map 容器中指定位置的数据。

参数:

pmap_map 指向 map 容器的指针。
it_pos 被删除的数据的位置。

返回值:

无。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。
it_pos 是属于 pmap_map 容器的有效的迭代器, 否则函数的行为是未定义的。

```
void map_erase_range(  
    map_t* pmap_map, map_iterator_t it_begin, map_iterator_t it_end);
```

描述:

删除 map 中指定数据区间的数据。

参数:

pmap_map 指向 map 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。
[it_begin, it_end) 必须属于 pmap_map, 否则函数的行为是未定义的。

```
void map_erase(map_t* pmap_map, elem);
```

描述:

删除 map 中指定的数据。

参数:

pmap_map 指向 map 容器的指针。
elem 要删除的指定数据。

返回值:

被删除的数据的个数。

注意：

如果 `pmap_map == NULL` 则函数的行为是未定义的，`map` 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 `map` 容器中的数据类型相同，否则函数的行为是未定义的。

```
void* map_at(const map_t* cpmap_map, elem);
```

描述：

通过下标访问 `map` 中的数据。

参数：

`pmap_map` 指向容器的指针。

`elem` 要查找的指定数据。

返回值：

返回以指定数据为键的数据的指针。

注意：

如果 `pmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 `map` 中不包含以指定的数据为键的数据，则首先向 `map` 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

`multimap` 提供的外部接口提供给用户使用。

<code>create_multimap</code>	创建 <code>multimap</code> 容器。
<code>multimap_init</code>	初始化 <code>multimap</code> 容器。
<code>multimap_init_ex</code>	使用自定义比较规则初始化 <code>multimap</code> 容器。
<code>multimap_init_copy</code>	使用存在的 <code>multimap</code> 容器进行初始化。
<code>multimap_init_copy_range</code>	使用指定的数据区间进行初始化。
<code>multimap_init_copy_range_ex</code>	使用指定的数据区间和比较函数进行初始化。
<code>multimap_destroy</code>	销毁 <code>multimap</code> 容器。
<code>multimap_assign</code>	使用存在的 <code>multimap</code> 容器赋值。
<code>multimap_size</code>	返回 <code>multimap</code> 容器中数据的数量。
<code>multimap_empty</code>	判断 <code>multimap</code> 容器是否为空。
<code>multimap_max_size</code>	返回 <code>multimap</code> 容器中能够保存数据的最大数量。
<code>multimap_begin</code>	返回指向 <code>multimap</code> 容器的第一个数据的迭代器。
<code>multimap_end</code>	返回指向 <code>multimap</code> 容器末尾的迭代器。
<code>multimap_key_comp</code>	返回 <code>multimap</code> 容器中的键比较规则。
<code>multimap_value_comp</code>	返回 <code>multimap</code> 容器中的数据比较规则。
<code>multimap_find</code>	在 <code>multimap</code> 容器中查找指定的数据。
<code>multimap_clear</code>	清空 <code>multimap</code> 容器。

multimap_count	返回 multimap 容器中指定数据的数量。
multimap_lower_bound	返回指向第一个大于等于指定数据的迭代器。
multimap_upper_bound	返回指向第一个大于指定数据的迭代器。
multimap_equal_range	返回包含指定数据的数据区间。
multimap_equal	测试两个 multimap 容器是否相等。
multimap_not_equal	测试两个 multimap 容器是否不等。
multimap_less	测试第一个 multimap 容器是否小于第二个 multimap 容器。
multimap_less_equal	测试第一个 multimap 容器是否小于等于第二个 multimap 容器。
multimap_greater	测试第一个 multimap 容器是否大于第二个 multimap 容器。
multimap_greater_equal	测试第一个 multimap 容器是否大于等于第二个 multimap 容器。
multimap_swap	交换两个 multimap 容器。
multimap_insert	向 multimap 中插入数据，数据必须唯一。
multimap_insert_hint	向 multimap 中线索位置插入数据，数据必须唯一。
multimap_insert_range	向 multimap 中插入指定的数据区间，数据必须唯一。
multimap_erase_pos	删除 multimap 中指定位置的数据。
multimap_erase_range	删除 multimap 中指定数据区间的数据。
multimap_erase	删除 multimap 中指定的数据。

函数原型

```
multimap_t* create_multimap(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

typename 类型描述。

返回值:

成功返回指向 multimap 容器的指针，否则返回 NULL。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void multimap_init(multimap_t* pmmmap_map);
```

描述:

初始化 multimap 迭代器。

参数:

pmmmap_map multimap 容器。

返回值:

无。

注意:

`pmmap_map == NULL` 则函数的行为是未定义的，`pmmap_map` 必须是使用 `create_multimap()` 创建的，否则函数的行为是未定义的，这个函数使用类型默认的比较函数。

```
void multimap_init_copy(multimap_t* pmmap_dest, const multimap_t* cpmmmap_src);
```

描述：

使用已经存在的 multimap 初始化 multimap 迭代器。

参数：

`pmmap_dest` 目的 multimap 容器。
`cpmmmap_src` 源 multimap 容器。

返回值：

无。

注意：

`pmmap_dest == NULL` 或者 `cpmmmap_src == NULL` 则函数的行为是未定义的，`pmmap_dest` 必须是使用 `create_multimap()` 创建的，`cpmmmap_src` 必须是已经初始化，否则函数的行为是未定义的。`pmmap_dest` 和 `cpmmmap_src` 保存的数据类型必须相同，否则函数的行为是未定义。

```
void multimap_init_copy_range(multimap_t* pmmap_dest,
    multimap_iterator_t it_begin, multimap_iterator_t it_end);
```

描述：

使用指定的数据区间初始化 rb tree 迭代器。

参数：

`pmmap_dest` 目的 multimap 容器。
`it_begin` 数据区间的开始。
`it_end` 数据区间的末尾。

返回值：

无。

注意：

如果 `pmmap_dest == NULL` 则函数的行为是未定义的，`pmmap_dest` 必须是使用 `create_multimap()` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 multimap 容器的有效数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void multimap_init_copy_range_ex(
    multimap_t* pt_dest, multimap_iterator_t it_begin, multimap_iterator_t it_end,
    binary_function_t t_compare);
```

描述：

使用指定的数据区间和比较函数初始化 multimap 迭代器。

参数：

`pmmap_dest` 目的 multimap 容器。
`it_begin` 数据区间的开始。
`it_end` 数据区间的末尾。
`t_compare` 比较函数。

返回值：

无。

注意：

如果 `pmmap_dest == NULL` 则函数的行为是未定义的，`pmmap_dest` 必须是使用 `create_multimap()` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 `multimap` 容器的有效数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 `t_compare == NULL` 则使用类型默认的比较函数。

```
void multimap_destroy(multimap_t* pmmap_map);
```

描述：

销毁创建的 `multimap` 容器。

参数：

`pmmap_map` `multimap` 容器。

返回值：

无。

注意：

如果 `pmmap_map == NULL` 则函数的行为是未定义的，`pmmap_map` 必须是初始化或者是使用 `create_multimap()` 创建的，否则函数的行为是未定义的。

```
void multimap_assign(multimap_t* pmmap_dest, const multimap_t* cpmmmap_src);
```

描述：

使用一个 `multimap` 为另一个 `multimap` 赋值。

参数：

`pmmap_dest` 指向被赋值的 `multimap` 容器的指针。

`cpmmmap_src` 指向赋值的 `multimap` 容器的指针。

返回值：

无。

注意：

如果 `pmmap_dest == NULL` 或者 `cpmmmap_src == NULL` 则函数的行为是未定义的，两个 `multimap` 必须已经初始化的，否则函数的行为是未定义的。两个 `multimap` 容器中保存的数据类型不同函数的行为是未定义的。如果 `multimap_equal(pmmap_dest, cpmmmap_src)` 那么函数不做任何动作。

```
size_t multimap_size(const multimap_t* cpmmmap_map);
```

描述：

获得 `multimap` 容器中保存的数据的个数。

参数：

`cpmmmap_map` `multimap` 容器。

返回值：

`multimap` 容器中保存的数据的个数。

注意：

如果 `cpmmmap_map == NULL` 则函数的行为是未定义的，`cpmmmap_map` 必须是已经初始化的 `multimap` 容器，否则函数的行为是未定义的。

```
bool_t multimap_empty(const multimap_t* cpmmmap_map);
```

描述:

测试 multimap 容器是否为空。

参数:

cpmmmap_map multimap 容器。

返回值:

如果 multimap 容器为空则返回 true, 否则返回 false。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的, cpmmmap_map 必须是已经初始化的 multimap 容器, 否则函数的行为是未定义的。

```
size_t multimap_max_size(const multimap_t* cpmmmap_map);
```

描述:

获得 multimap 容器中数据的最大数量。

参数:

cpmmmap_map multimap 容器。

返回值:

multimap 容器中保存的数据的个数的最大值。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的, cpmmmap_map 必须是已经初始化的 multimap 容器, 否则函数的行为是未定义的。

```
multimap_iterator_t multimap_begin(const multimap_t* cpmmmap_map);
```

描述:

获得引用 multimap 容器中第一数据的迭代器。

参数:

cpmmmap_map 指向 multimap 容器的指针。

返回值:

返回引用 multimap 容器中第一个数据的迭代器。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。如果 multimap 容器为空, 则返回值与 multimap_end(cpmmmap_map)相等。

```
multimap_iterator_t multimap_end(const multimap_t* cpmmmap_map);
```

描述:

获得引用 multimap 容器末尾的迭代器。

参数:

cpmmmap_map 指向 multimap 容器的指针。

返回值:

返回引用 multimap 容器末尾的迭代器。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未

定义的。

```
binary_function_t multimap_key_comp(const multimap_t* cpmmmap_map);
```

描述:

返回 multimap 容器中的键比较规则。

参数:

cpmmmap_map 指向 multimap 容器的指针。

返回值:

返回 multimap 容器中的数据比较规则。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t multimap_value_comp(const multimap_t* cpmmmap_map);
```

描述:

返回 multimap 容器中的数据比较规则。

参数:

cpmmmap_map 指向 multimap 容器的指针。

返回值:

返回 multimap 容器中的数据比较规则。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。

```
multimap_iterator_t multimap_find(const multimap_t* cpmmmap_map, elem);
```

描述:

在 multimap 容器中查找指定的数据。

参数:

cpmmmap_map 指向 multimap 容器的指针。

elem 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void multimap_clear(multimap_t* pmmmap_map);
```

描述:

删除 multimap 容器中的所有数据。

参数:

pmmmap_map 指向 multimap 容器的指针。

返回值:

无。

注意：

如果 cpmmmap_map == NULL 则函数的行为是未定义的， multimap 容器必须已经初始化的，否则函数的行为是未定义的。

```
size_t multimap_count(const multimap_t* cpmmmap_map, elem);
```

描述：

返回 multimap 容器中指定数据的数量。

参数：

cpmmmap_map 指向 multimap 容器的指针。

elem 要查找的指定数据。

返回值：

返回 multimap 容器中指定数据的数量。

注意：

如果 cpmmmap_map == NULL 则函数的行为是未定义的， multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
multimap_iterator_t multimap_lower_bound(const multimap_t* cpmmmap_map, elem);
```

描述：

返回指向第一个大于等于指定数据的迭代器。

参数：

cpmmmap_map 指向 multimap 容器的指针。

elem 要查找的指定数据。

返回值：

返回指向第一个大于等于指定数据的迭代器。

注意：

如果 cpmmmap_map == NULL 则函数的行为是未定义的， multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
multimap_iterator_t multimap_upper_bound(const multimap_t* cpmmmap_map, elem);
```

描述：

返回指向第一个大于指定数据的迭代器。

参数：

cpmmmap_map 指向 multimap 容器的指针。

elem 要查找的指定数据。

返回值：

返回指向第一个大于指定数据的迭代器。

注意：

如果 cpmmmap_map == NULL 则函数的行为是未定义的， multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t multimap_equal_range(const multimap_t* cpmmmap_map, elem);
```

描述:

返回包含指定数据的数据区间。

参数:

cpmmmap_map 指向 multimap 容器的指针。

elem 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_multimap == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
bool_t multimap_equal(
    const multimap_t* cpmmmap_first, const multimap_t* cpmmmap_second);
```

描述:

测试两个 multimap 容器是否相等。

参数:

cpmmmap_first 指向 multimap 容器的指针。

cpmmmap_second 指向 multimap 容器的指针。

返回值:

如果两个 multimap 容器相等则返回 true，否则返回 false。

注意:

如果 cpmmmap_first == NULL 或者 cpmmmap_second == NULL 则函数的行为是未定义的，两个 multimap 必须已经初始化的，否则函数的行为是未定义的。两个 multimap 容器中保存的数据类型不同认为这两个容器不等。两个 multimap 容器相等是指容器中的数据对应相等，并且容器中数据的数量也相等。如果 cpmmmap_first == cpmmmap_second 那么返回 true。

```
bool_t multimap_not_equal(
    const multimap_t* cpmmmap_first, const multimap_t* cpmmmap_second);
```

描述:

测试两个 multimap 容器是否不等。

参数:

cpmmmap_first 指向 multimap 容器的指针。

cpmmmap_second 指向 multimap 容器的指针。

返回值:

如果两个 multimap 容器不等则返回 true，否则返回 false。

注意:

如果 cpmmmap_first == NULL 或者 cpmmmap_second == NULL 则函数的行为是未定义的，两个 multimap 必须已经初始化的，否则函数的行为是未定义的。两个 multimap 容器中保存的数据类型不同认为两个 multimap 容器不等。两个 multimap 容器不等是指容器中的对应的数据不相等，如果对应的数据都相等，那就要看容器中数据的数量是否不等。如果 cpmmmap_first == cpmmmap_second 那么返回 false。

```
bool_t multimap_less(
```

```
const multimap_t* cpmmmap_less(
```

描述:

测试第一个 multimap 是否小于第二个 multimap。

参数:

cpmmmap_first 指向 multimap 容器的指针。
cpmmmap_second 指向 multimap 容器的指针。

返回值:

如果第一个 multimap 容器小于第二个 multimap 容器则返回 true, 否则返回 false。

注意:

如果 cpmmmap_first == NULL 或者 cpmmmap_second == NULL 则函数的行为是未定义的, 两个 multimap 必须已经初始化的, 否则函数的行为是未定义的。两个 multimap 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 multimap 中的数据小于第二个 multimap 中对应的数据则返回 true, 如果大于则返回 false, 当两个 multimap 中的对应数据相等, 第一个 multimap 中的数据数量小于第二个 multimap 中数据的数量返回 true, 否则返回 false。如果 cpmmmap_first == cpmmmap_second 那么返回 false。

```
bool_t multimap_less_equal(
```



```
const multimap_t* cpmmmap_first, const multimap_t* cpmmmap_second);
```

描述:

测试第一个 multimap 是否小于等于第二个 multimap。

参数:

cpmmmap_first 指向 multimap 容器的指针。
cpmmmap_second 指向 multimap 容器的指针。

返回值:

如果第一个 multimap 容器小于等于第二个 multimap 容器则返回 true, 否则返回 false。

注意:

如果 cpmmmap_first == NULL 或者 cpmmmap_second == NULL 则函数的行为是未定义的, 两个 multimap 必须已经初始化的, 否则函数的行为是未定义的。两个 multimap 容器中保存的数据类型不同函数的行为是未定义的。如果 cpmmmap_first == cpmmmap_second 那么返回 true。

```
bool_t multimap_greater(
```



```
const multimap_t* cpmmmap_first, const multimap_t* cpmmmap_second);
```

描述:

测试第一个 multimap 是否大于第二个 multimap。

参数:

cpmmmap_first 指向 multimap 容器的指针。
cpmmmap_second 指向 multimap 容器的指针。

返回值:

如果第一个 multimap 容器大于第二个 multimap 容器则返回 true, 否则返回 false。

注意:

如果 cpmmmap_first == NULL 或者 cpmmmap_second == NULL 则函数的行为是未定义的, 两个 multimap 必须已经初始化的, 否则函数的行为是未定义的。两个 multimap 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 multimap 中的数据大于第二个 multimap 中对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如

果第一个 multimap 中数据的数量大于第二个 multimap 中数据的数量的时候返回 true 否则返回 false。如果 cpmmmap_first == cpmmmap_second 那么返回 false。

```
bool_t multimap_greater_equal(
    const multimap_t* cpmmmap_first, const multimap_t* cpmmmap_second);
```

描述:

测试第一个 multimap 是否大于等于第二个 multimap。

参数:

cpmmmap_first 指向 multimap 容器的指针。
cpmmmap_second 指向 multimap 容器的指针。

返回值:

如果第一个 multimap 容器大于等于第二个 multimap 容器则返回 true, 否则返回 false。

注意:

如果 cpmmmap_first == NULL 或者 cpmmmap_second == NULL 则函数的行为是未定义的，两个 multimap 必须已经初始化的，否则函数的行为是未定义的。两个 multimap 容器中保存的数据类型不同函数的行为是未定义的。如果 cpmmmap_first == cpmmmap_second 那么返回 true。

```
void multimap_swap(multimap_t* pmmap_first, multimap_t* pmmap_second);
```

描述:

交换两个 multimap 容器。

参数:

pmmap_first 指向 multimap 容器的指针。
pmmap_second 指向 multimap 容器的指针。

返回值:

无。

注意:

如果 pmmap_first == NULL 或者 pmmap_second == NULL 则函数的行为是未定义的，两个 multimap 必须已经初始化的，否则函数的行为是未定义的。两个 multimap 容器中保存的数据类型不同函数的行为是未定义的。如果 multimap_equal(pmmap_first, pmmap_second)，函数不执行任何动作。

```
multimap_iterator_t multimap_insert(
    multimap_t* pmmap_map, const pair_t* cppair_pair);
```

描述:

向 multimap 容器中插入唯一的数据。

参数:

pmmap_map 指向 multimap 容器的指针。
cppair_pair 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pmmap_map == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
multimap_iterator_t multimap_insert_hint(  
    multimap_t* pmmmap_map, multimap_iterator_t it_hint, const pair_t* cppair_pair);
```

描述:

向 multimap 容器中的线索位置插入唯一的数据。

参数:

pmmmap_map 指向 multimap 容器的指针。
it_hint 用户提供的线索位置。
cppair_pair 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_multimap == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
void multimap_insert_range(multimap_t* pmmmap_map,  
    multimap_iterator_t it_begin, multimap_iterator_t it_end);
```

描述:

向 multimap 中插入指定的数据区间，数据必须唯一。

参数:

pmmmap_map 指向 multimap 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pmmmap_map == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间并且与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。[it_begin, it_end)必须不属于 pmmmap_map，否则函数的行为是未定义的。

```
void multimap_erase_pos(multimap_t* pmmmap_map, multimap_iterator_t it_pos);
```

描述:

删除 multimap 容器中指定位置的数据。

参数:

pmmmap_map 指向 multimap 容器的指针。
it_pos 被删除的数据的位置。

返回值:

无。

注意:

如果 pmmmap_map == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。it_pos 是属于 pmmmap_map 容器的有效的迭代器，否则函数的行为是未定义的。

```
void multimap_erase_range(multimap_t* pmmmap_map,
    multimap_iterator_t it_begin, multimap_iterator_t it_end);
```

描述:

删除 multimap 中指定数据区间的数据。

参数:

pmmmap_map 指向 multimap 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pmmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end) 必须属于 pmmmap_map, 否则函数的行为是未定义的。

```
void multimap_erase(multimap_t* pmmmap_map, elem);
```

描述:

删除 multimap 中指定的数据。

参数:

pmmmap_map 指向 multimap 容器的指针。
elem 要删除的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 pmmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

第五节 迭代器接口

map 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

create_map_iterator	创建 map 迭代器。
_map_iterator_get_value	获得 map 迭代器引用的数据。
_map_iterator_get_pointer	获得 map 迭代器引用的数据的指针。
_map_iterator_next	获得引用下一个数据的迭代器。
_map_iterator_prev	获得引用前一个数据的迭代器。
_map_iterator_equal	测试两个迭代器是否相等。
_map_iterator_distance	计算两个迭代器之间的距离。

_map_iterator_before

测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
map_iterator_t create_map_iterator(void);
```

描述:

创建一个 map 迭代器。

参数:

无。

返回值:

map 迭代器。

注意:

返回的 map 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _map_iterator_get_value(map_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter map 迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意:

it_iter 是有效的 map 迭代器，否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的，pv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _map_iterator_get_pointer(map_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter map 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 map 迭代器，否则函数的行为是未定义的。

```
map_iterator_t _map_iterator_next(map_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter map 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 map 迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 map 有效的迭代器，否则函数的行为是未定义的。

```
map_iterator_t _map_iterator_prev(map_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter map 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 map 迭代器，否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 map 有效的迭代器，否则函数的行为是未定义的。

```
bool_t _map_iterator_equal(map_iterator_t it_first, map_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first map 迭代器。

it_second map 迭代器。

返回值:

如果两个迭代器相等则返回 true，否则返回 false。

注意:

两个 iter 是属于同一个 map 容器的有效的 map 迭代器，否则函数的行为是未定义的。两个 map 迭代器相等是指两个迭代器引用相同的数据。

```
int _map_iterator_distance(map_iterator_t it_first, map_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_first map 迭代器。

it_second map 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 map 容器的有效的 map 迭代器，否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0，it_first 引用的数据在 it_second 后面那么结果<0，如果两个迭代器相等那么结果为 0。

```
bool_t _map_iterator_before(map_iterator_t it_first, map_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first map 迭代器。

it_second map 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 map 容器的有效的 map 迭代器, 否则函数的行为是未定义的。

multimap 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

create_multimap_iterator	创建 multimap 迭代器。
_multimap_iterator_get_value	获得 multimap 迭代器引用的数据。
_multimap_iterator_get_pointer	获得 multimap 迭代器引用的数据的指针。
_multimap_iterator_next	获得引用下一个数据的迭代器。
_multimap_iterator_prev	获得引用前一个数据的迭代器。
_multimap_iterator_equal	测试两个迭代器是否相等。
_multimap_iterator_distance	计算两个迭代器之间的距离。
_multimap_iterator_before	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
multimap_iterator_t create_multimap_iterator(void);
```

描述:

创建一个 multimap 迭代器。

参数:

无。

返回值:

multimap 迭代器。

注意:

返回的 multimap 迭代器并不是有效的迭代器, 它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作, 返回的迭代器供其他接口使用。

```
void _multimap_iterator_get_value(multimap_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter multimap 迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意：

it_iter 是有效的 multimap 迭代器，否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的，pv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _multimap_iterator_get_pointer(multimap_iterator_t it_iter);
```

描述：

获得迭代器引用的数据的指针。

参数：

it_iter multimap 迭代器。

返回值：

指向迭代器引用的数据的指针。

注意：

it_iter 是有效的 multimap 迭代器，否则函数的行为是未定义的。

```
multimap_iterator_t _multimap_iterator_next(multimap_iterator_t it_iter);
```

描述：

获得引用下一个数据的迭代器。

参数：

it_iter multimap 迭代器。

返回值：

返回引用下一个数据的迭代器。

注意：

it_iter 是有效的 multimap 迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 multimap 有效的迭代器，否则函数的行为是未定义的。

```
multimap_iterator_t _multimap_iterator_prev(multimap_iterator_t it_iter);
```

描述：

获得引用上一个数据的迭代器。

参数：

it_iter multimap 迭代器。

返回值：

返回引用上一个数据的迭代器。

注意：

it_iter 是有效的 multimap 迭代器，否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 multimap 有效的迭代器，否则函数的行为是未定义的。

```
bool_t _multimap_iterator_equal(
    multimap_iterator_t it_first, multimap_iterator_t it_second);
```

描述：

测试两个迭代器是否相等。

参数:

it_first multimap 迭代器。
it_second multimap 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 multimap 容器的有效的 multimap 迭代器, 否则函数的行为是未定义的。两个 multimap 迭代器相等是指两个迭代器引用相同的数据。

```
int _multimap_iterator_distance(  
    multimap_iterator_t it_first, multimap_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_first multimap 迭代器。
it_second multimap 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 multimap 容器的有效的 multimap 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为 0。

```
bool_t _multimap_iterator_before(  
    multimap_iterator_t it_first, multimap_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first multimap 迭代器。
it_second multimap 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 multimap 容器的有效的 multimap 迭代器, 否则函数的行为是未定义的。

第六节 内部和辅助接口

map 提供的内部接口是为了给外部接口使用。

_create_map

创建 map 容器。

_create_map_auxiliary	创建 map 容器的辅助函数。
_map_destroy_auxiliary	销毁 map 容器的辅助函数。
_map_find	在 map 容器中查找指定的数据。
_map_find_varg	在 map 容器中查找指定的数据，数据来自于可变参数列表。
_map_count	返回 map 容器中指定数据的数量。
_map_count_varg	返回 map 容器中指定数据的数量，数据来自于可变参数列表。
_map_lower_bound	返回指向第一个大于等于指定数据的迭代器。
_map_lower_bound_varg	返回指向第一个大于等于指定数据的迭代器，数据来自于可变参数列表。
_map_upper_bound	返回指向第一个大于指定数据的迭代器。
_map_upper_bound_varg	返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。
_map_equal_range	返回包含指定数据的数据区间。
_map_equal_range_varg	返回包含指定数据的数据区间，数据来自于可变参数列表。
_map_at	使用下标对 map 中的数据进行访问。
_map_at_varg	使用下标对 map 中的数据进行访问，数据来自于可变参数列表。
_map_erase	删除 map 中指定的数据。
_map_erase_varg	删除 map 中指定的数据，数据来自于可变参数列表。
_map_init_elem_auxiliary	初始化数据的辅助函数。

函数原型

```
map_t* _create_map(const char* s_typename);
```

描述:

创建一个 map 迭代器。

参数:

s_typename 类型描述。

返回值:

成功返回指向 map 容器的指针，否则返回 NULL。

注意:

s_typename != NULL，否则函数的行为是未定义的。

```
bool_t _create_map_auxiliary(map_t* pmap_map, const char* s_typename);
```

描述:

创建一个 map 容器的辅助函数。

参数:

pmap_map 没有创建的 map 容器。

s_typename map 中保存的数据类型的名字。

返回值:

创建成功返回 true，否则返回 false。

注意:

如果 pmap_map == NULL 或者 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libestl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _map_destroy_auxiliary(map_t* pmap_map);
```

描述:

销毁 map 容器的辅助函数。

参数:

pmap_map map 容器。

返回值:

无。

注意:

如果 pmap_map == NULL 或者 map 不是使用_create_map 生成的则函数的行为是未定义的。

```
map_iterator_t _map_find(const map_t* cpmap_map, ...);
```

描述:

在 map 容器中查找指定的数据。

参数:

cpmap_map 指向 map 容器的指针。

... 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
map_iterator_t _map_find_varg(const map_t* cpmap_map, va_list val_elemlist);
```

描述:

在 map 容器中查找指定的数据, 数据来自于参数列表。

参数:

cpmap_map 指向 map 容器的指针。

val_elemlist 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
size_t _map_count(const map_t* cpmap_map, ...);
```

描述:

返回 map 容器中指定数据的数量。

参数:

`cpmap_map` 指向 map 容器的指针。
... 要查找的指定数据。

返回值:

返回 map 容器中指定数据的数量。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
size_t _map_count_varg(const map_t* cpmap_map, va_list val_elemlist);
```

描述:

返回 map 容器中指定数据的数量, 数据来自于参数列表。

参数:

`cpmap_map` 指向 map 容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回 map 容器中指定数据的数量。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
map_iterator_t _map_lower_bound(const map_t* cpmap_map, ...);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

`cpmap_map` 指 map 向容器的指针。
... 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
map_iterator_t _map_lower_bound_varg(const map_t* cpmap_map, va_list val_elemlist);
```

描述:

返回指向第一个大于等于指定数据的迭代器, 数据来自于可变参数列表。

参数:

`cpmap_map` 指 map 向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查

找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
map_iterator_t _map_upper_bound(const map_t* cpmap_map, ...);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cpmap_map 指 map 向容器的指针。
... 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
map_iterator_t _map_upper_bound_varg(const map_t* cpmap_map, va_list val_elemlist);
```

描述:

返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。

参数:

cpmap_map 指 map 向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _map_equal_range(const map_t* cpmap_map, ...);
```

描述:

返回包含指定数据的数据区间。

参数:

cpmap_map 指向容器的指针。
... 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _map_equal_range_varg(const map_t* cpmap_map, va_list val_elemlist);
```

描述:

返回包含指定数据的数据区间。

参数:

`cpmap_map` 指向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回包含指定数据的数据区间。

注意:

如果 `cpmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
void* _map_at(const map_t* cpmap_map, ...);
```

描述:

通过下标访问 map 中的数据。

参数:

`pmap_map` 指向容器的指针。
... 要查找的指定数据。

返回值:

返回以指定数据为键的数据的指针。

注意:

如果 `pmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 map 中不包含以指定的数据为键的数据，则首先向 map 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

```
void* _map_at_varg(const map_t* pmap_map, va_list val_elemlist);
```

描述:

通过下标访问 map 中的数据。

参数:

`pmap_map` 指向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回以指定数据为键的数据的指针。

注意:

如果 `pmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 map 中不包含以指定的数据为键的数据，则首先向 map 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

```
void _map_erase(map_t* pmap_map, ...);
```

描述:

删除 map 中指定的数据。

参数:

`pmap_map` 指向 map 容器的指针。
... 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _map_erase_varg(map_t* pmap_map, va_list val_elemlist);
```

描述:

删除 map 中指定的数据, 数据来自于可变参数列表。

参数:

pmap_map 指向 map 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 pmap_map == NULL 则函数的行为是未定义的, map 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _map_init_elem_auxiliary(map_t* pmap_map, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

pmap_map map 容器。
pv_value 初始化的数据。

返回值:

无。

注意:

如果 pmap_map == NULL 或者 pv_value == NULL , 那么函数的行为是未定义的。pmap_map 必须是初始化的或者是使用_create_map()创建的, 否则函数的行为是未定义的。

multimap 提供的内部接口是为了给外部接口使用。

_create_multimap	创建 multimap 容器。
_create_multimap_auxiliary	创建 multimap 容器的辅助函数。
_multimap_destroy_auxiliary	销毁 multimap 容器的辅助函数。
_multimap_find	在 multimap 容器中查找指定的数据。
_multimap_find_varg	在 multimap 容器中查找指定的数据, 数据来自于可变参数列表。
_multimap_count	返回 multimap 容器中指定数据的数量。
_multimap_count_varg	返回 multimap 容器中指定数据的数量, 数据来自于可变参数列表。
_multimap_lower_bound	返回指向第一个大于等于指定数据的迭代器。
_multimap_lower_bound_varg	返回指向第一个大于等于指定数据的迭代器, 数据来自于可变参数列表。
_multimap_upper_bound	返回指向第一个大于指定数据的迭代器。

<code>_multimap_upper_bound_varg</code>	返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。
<code>_multimap_equal_range</code>	返回包含指定数据的数据区间。
<code>_multimap_equal_range_varg</code>	返回包含指定数据的数据区间，数据来自于可变参数列表。
<code>_multimap_erase</code>	删除 multimap 中指定的数据。
<code>_multimap_erase_varg</code>	删除 multimap 中指定的数据，数据来自于可变参数列表。
<code>_multimap_init_elem_auxiliary</code>	初始化数据的辅助函数。

函数原型

```
multimap_t* _create_multimap(const char* s_typename);
```

描述:

创建一个 multimap 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 multimap 容器的指针，否则返回 NULL。

注意:

`s_typename != NULL`，否则函数的行为是未定义的。

```
bool_t _create_multimap_auxiliary(multimap_t* pmmmap_map, const char* s_typename);
```

描述:

创建一个 multimap 容器的辅助函数。

参数:

`pmmmap_map` 没有创建的 multimap 容器。

`s_typename` multimap 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pmmmap_map == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
void _multimap_destroy_auxiliary(multimap_t* pmmmap_map);
```

描述:

销毁 multimap 容器的辅助函数。

参数:

`pmmmap_map` multimap 容器。

返回值:

无。

注意:

如果 `pmmmap_map == NULL` 或者 multimap 不是使用 `_create_multimap` 生成的则函数的行为是未定义的。

```
multimap_iterator_t _multimap_find(const multimap_t* cpmmmap_map, ...);
```

描述:

在 multimap 容器中查找指定的数据。

参数:

cpmmmap_map 指向 multimap 容器的指针。
... 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
multimap_iterator_t _multimap_find_varg(  
    const multimap_t* cpmmmap_map, va_list val_elemlist);
```

描述:

在 multimap 容器中查找指定的数据，数据来自于参数列表。

参数:

cpmmmap_map 指向 multimap 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _multimap_count(const multimap_t* cpmmmap_map, ...);
```

描述:

返回 multimap 容器中指定数据的数量。

参数:

cpmmmap_map 指向 multimap 容器的指针。
... 要查找的指定数据。

返回值:

返回 multimap 容器中指定数据的数量。

注意:

如果 cpmmmap_map == NULL 则函数的行为是未定义的，multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _multimap_count_varg(const multimap_t* cpmmmap_map, va_list val_elemlist);
```

描述:

返回 multimap 容器中指定数据的数量，数据来自于参数列表。

参数:

cpmmmap_map 指向 multimap 容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回 multimap 容器中指定数据的数量。

注意:

如果 `cpmmmap_map == NULL` 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
multimap_iterator_t _multimap_lower_bound(const multimap_t* cpmmmap_map, ...);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

`cpmmmap_map` 指 multimap 向容器的指针。

`...` 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cpmmmap_map == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
multimap_iterator_t _multimap_lower_bound_varg(
    const multimap_t* cpmmmap_map, va_list val_elemlist);
```

描述:

返回指向第一个大于等于指定数据的迭代器, 数据来自于可变参数列表。

参数:

`cpmmmap_map` 指 multimap 向容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cpmmmap_map == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
multimap_iterator_t _multimap_upper_bound(const multimap_t* cpmmmap_map, ...);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

`cpmmmap_map` 指 multimap 向容器的指针。

`...` 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cpmmmap_map == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要

查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
multimap_iterator_t _multimap_upper_bound_varg(
    const multimap_t* cpmmmap_map, va_list val_elemlist);
```

描述：

返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。

参数：

cpmmmap_map 指 multimap 向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值：

返回指向第一个大于指定数据的迭代器。

注意：

如果 cpmmmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _multimap_equal_range(const multimap_t* cpmmmap_map, ...);
```

描述：

返回包含指定数据的数据区间。

参数：

cpmmmap_map 指向容器的指针。
... 要查找的指定数据。

返回值：

返回包含指定数据的数据区间。

注意：

如果 cpmmmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _multimap_equal_range_varg(
    const multimap_t* cpmmmap_map, va_list val_elemlist);
```

描述：

返回包含指定数据的数据区间。

参数：

cpmmmap_map 指向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值：

返回包含指定数据的数据区间。

注意：

如果 cpmmmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
void _multimap_erase(multimap_t* pmmmap_map, ...);
```

描述：

删除 multimap 中指定的数据。

参数:

pmmap_map 指向 multimap 容器的指针。
... 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 pmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _multimap_erase_varg(multimap_t* pmmap_map, va_list val_elemlist);
```

描述:

删除 multimap 中指定的数据, 数据来自于可变参数列表。

参数:

pmmap_map 指向 multimap 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 pmmap_map == NULL 则函数的行为是未定义的, multimap 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _multimap_init_elem_auxiliary(multimap_t* pmmap_map, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

pmmap_map multimap 容器。
pv_value 初始化的数据。

返回值:

无。

注意:

如果 pmmap_map == NULL 或者 pv_value == NULL , 那么函数的行为是未定义的。pmmap_map 必须是初始化的或者使用_create_multimap()创建的, 否则函数的行为是未定义的。

map 提供了少量的辅助函数。

_map_same_pair_type	判断两个 pair 类型是否相同。
_map_same_pair_type_ex	判断两个 pair 类型是否相同。
_map_value_compare	map 的 key 和 value 比较函数。

函数原型

```
bool_t _map_same_pair_type(const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

ppair_first 第一个 pair 类型。

ppair_second 第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true, 否则返回 false。

注意:

如果 ppair_first == NULL 或者 ppair_second == NULL, 那么函数的行为是未定义的。

```
bool_t _map_same_pair_type_ex(  
    const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

ppair_first 第一个 pair 类型。

ppair_second 第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true, 否则返回 false。

注意:

如果 ppair_first == NULL 或者 ppair_second == NULL, 那么函数的行为是未定义的。

```
void _map_value_compare(  
    const void* cpv_first, const void* cpv_second, void* pv_output);
```

描述:

map 值的比较函数。

参数:

cpv_first 第一个数据。

cpv_second 第二个数据。

pv_output 输出结构。

返回值:

无。

注意:

这个函数是 map 的数据比较函数, 这个函数主要提供给 avl tree 或者 rb tree 比较数据使用。如果 cpv_first == NULL 或者 cpv_second == NULL 或者 pv_output == NULL 那么这个函数的行为是未定义的。cpv_first 和 cpv_second 必须是相同的 pair 类型, 否则函数的行为是未定义的。

multimap 提供了少量的辅助函数。

_multimap_same_pair_type	判断两个 pair 类型是否相同。
--------------------------	-------------------

_multimap_same_pair_type_ex	判断两个 pair 类型是否相同。
-----------------------------	-------------------

_multimap_value_compare

multimap 的 key 和 value 比较函数。

函数原型

```
bool_t _multimap_same_pair_type(
    const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

ppair_first 第一个 pair 类型。
ppair_second 第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true, 否则返回 false。

注意:

如果 ppair_first == NULL 或者 ppair_second == NULL, 那么函数的行为是未定义的。

```
bool_t _multimap_same_pair_type_ex(
    const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

ppair_first 第一个 pair 类型。
ppair_second 第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true, 否则返回 false。

注意:

如果 ppair_first == NULL 或者 ppair_second == NULL, 那么函数的行为是未定义的。

```
void _multimap_value_compare(
    const void* cpv_first, const void* cpv_second, void* pv_output);
```

描述:

multimap 值的比较函数。

参数:

cpv_first 第一个数据。
cpv_second 第二个数据。
pv_output 输出结构。

返回值:

无。

注意:

这个函数是 multimap 的数据比较函数, 这个函数主要提供给 avl tree 或者 rb tree 比较数据使用。如果 cpv_first == NULL 或者 cpv_second == NULL 或者 pv_output == NULL 那么这个函数的行为是未定义的。cpv_first 和 cpv_second 必须是相同的 pair 类型, 否则函数的行为是未定义的。

第十九章 hash 表

hash 表线性表，通过 hash 函数将数据映射到 hash 表中固定的位置，通过选取好的 hash 函数和合适的 hash 表长度，可以将数据分别映射到 hash 表的不同的位置，这样可以实现几乎常数时间的插入，删除，查询等操作。

第一节 hash 表的机制

libcstl 中采用的 hash 表是一种链式的 hash 表，使用线性表作为桶，链式寻址来解决 hash 冲突：

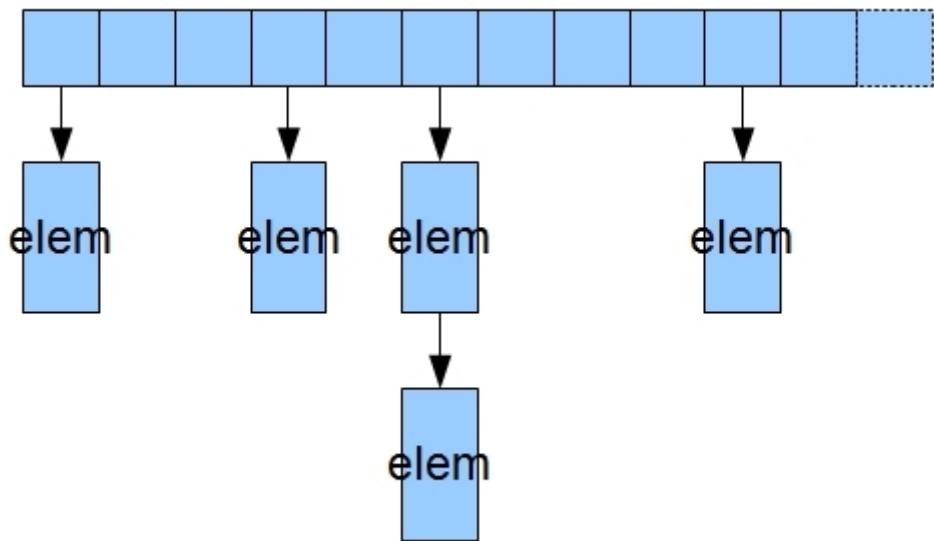


图 19.1 libcstl 采用的 hash 表

libcstl 的 hash 表桶的个数都是质数，采用质数个桶可以是 hash 表中的数据分布的更均匀。hash 表有一个质数列表，其中保存了 $50 \sim 2^{32}$ 范围内的质数，当用户指定 hash 表的大小的时候，在这个表中选择一个大于用户指定数据的最小指数作为 hash 表桶的个数。当用户指定的个数大于这个表中记录的个数的时候，那么就直接使用用户指定的个数。

hash 表是线性列表，我们使用 `vector_t` 类型来作为 hash 表的底层实现。在创建 `vector_t` 之前首先注册 `_hashnode_t*` 类型，这样桶中保存的就是指向 hash 节点的指针，这样就可以使用链表寻址来解决 hash 冲突。

hash 函数的好坏直接影响了数据在 hash 表中分布的情况。这个 hash 函数是允许用户指定的，libcstl 提供了默认的 hash 函数，这个 hash 函数只是简单的计算 hash 节点的内存值的和。

当 hash 表中的数据的个数大于 hash 表桶的个数的时候，我们认为这样在插入数据的时候产生冲突的几率极大，所以这个时候要扩充 hash 表。

第二节 hash 表迭代器

hash 表的实际结构以及迭代器的操作示意图:

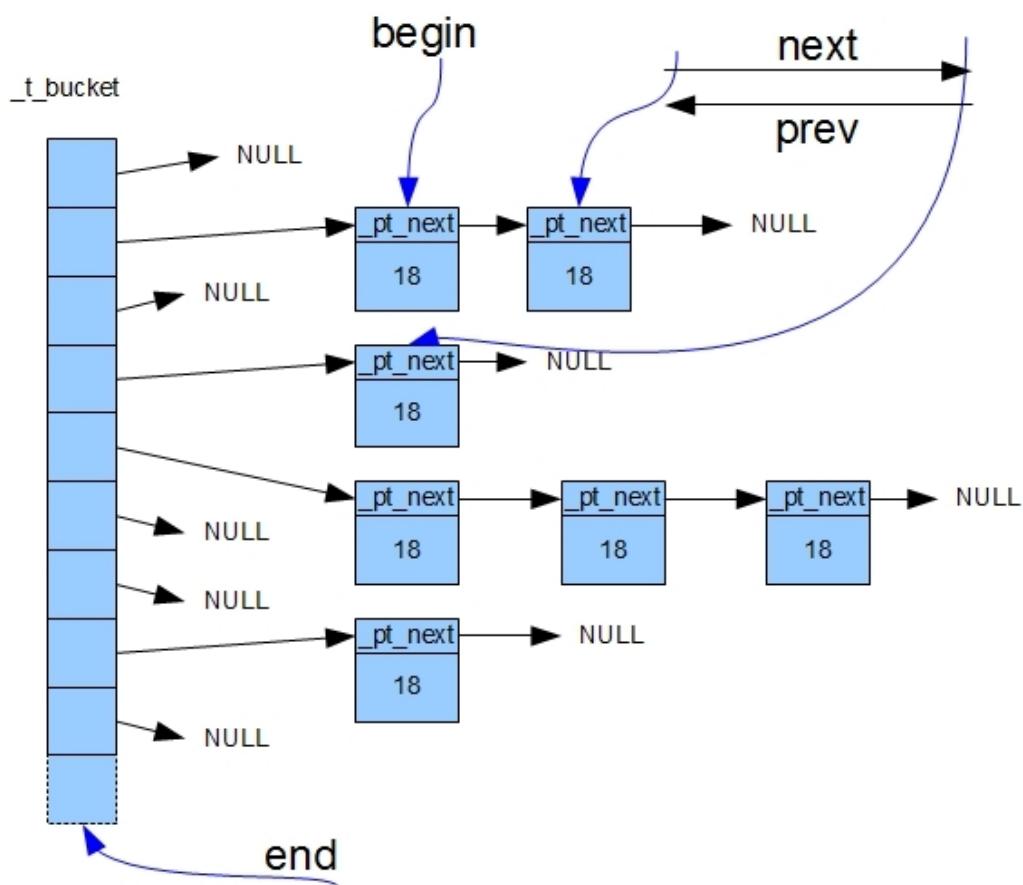


图 19.2 hash 表的结构和迭代器

要使用特殊的结构表示 hash 表的迭代器:

```
typedef struct __tagiterator
{
    /* 容器内部结构的信息 */
    struct
    {
        char*      _pc_corepos;
        char*      _pc_bucketpos; /* pointer to vector bucket position */
        void*      _pt_hashtable; /* point to hash node */
    }_t_pos;
    void*          _pt_container; /* 容器的位置 */
    containertype_t _t_containertype; /* 容器的类型 */
}
```

```
    iteratortype_t _t_iteratortype; /* 迭代器的类型 */
}iterator_t;
```

第三节 hash 表代码结构

hash 节点中包含很多信息：

```
typedef struct _taghashnode
{
    struct _taghashnode* _pt_next;
    char                 _pc_data[1];
}hashnode_t;
```

下面的结构表示 hash 表：

```
typedef struct _taghashtable
{
    _typeinfo_t      _t_typeinfo;
    _alloc_t         _t_allocator;
    vector_t         _t_bucket;
    size_t           _t_nodecount;
    unary_function_t _t_hash;
    binary_function_t _t_compare;
}hashtable_t;
```

第四节 外部接口

hashtable 提供的外部接口是为了实现关联容器。

_create_hashtable	创建 hashtable 容器。
_hashtable_init	初始化 hashtable 容器。
_hashtable_init_copy	使用存在的 hashtable 容器进行初始化。
_hashtable_init_copy_range	使用指定的数据区间进行初始化。
_hashtable_destroy	销毁 hashtable 容器。
_hashtable_assign	使用存在的 hashtable 容器赋值。
_hashtable_size	返回 hashtable 容器中数据的数量。

_hashtable_empty	判断 hashtable 容器是否为空。
_hashtable_max_size	返回 hashtable 容器中能够保存数据的最大数量。
_hashtable_bucket_count	获得 hashtable 中桶的数量。
_hashtable_begin	返回指向 hashtable 容器的第一个数据的迭代器。
_hashtable_end	返回指向 hashtable 容器末尾的迭代器。
_hashtable_hash	返回 hashtable 的 hash 函数。
_hashtable_key_comp	返回 hashtable 容器中的数据比较规则。
_hashtable_find	在 hashtable 容器中查找指定的数据。
_hashtable_clear	清空 hashtable 容器。
_hashtable_count	返回 hashtable 容器中指定数据的数量。
_hashtable_equal_range	返回包含指定数据的数据区间。
_hashtable_equal	测试两个 hashtable 容器是否相等。
_hashtable_not_equal	测试两个 hashtable 容器是否不等。
_hashtable_less	测试第一个 hashtable 容器是否小于第二个 hashtable 容器。
_hashtable_less_equal	测试第一个 hashtable 容器是否小于等于第二个 hashtable 容器。
_hashtable_greater	测试第一个 hashtable 容器是否大于第二个 hashtable 容器。
_hashtable_greater_equal	测试第一个 hashtable 容器是否大于等于第二个 hashtable 容器。
_hashtable_swap	交换两个 hashtable 容器。
_hashtable_insert_unique	向 hashtable 中插入数据，数据必须唯一。
_hashtable_insert_equal	向 hashtable 中插入数据，数据可以重复。
_hashtable_insert_unique_range	向 hashtable 中插入指定的数据区间，数据必须唯一。
_hashtable_insert_equal_range	向 hashtable 中插入指定的数据区间，数据可以重复。
_hashtable_erase_pos	删除 hashtable 中指定位置的数据。
_hashtable_erase_range	删除 hashtable 中指定数据区间的数据。
_hashtable_resize	重新设置 hashtable 的桶数。

函数原型

```
_hashtable_t* _create_hashtable(const char* s_typename);
```

描述:

创建一个 hashtable 迭代器。

参数:

s_typename 类型描述。

返回值:

成功返回指向 hashtable 容器的指针，否则返回 NULL。

注意:

s_typename != NULL，否则函数的行为是未定义的。

```
void _hashtable_init(
    _hashtable_t* pt_hashtable, size_t t_bucketcount,
    unary_function_t t_hash, binary_function_t t_compare);
```

描述:

初始化 hashtable 迭代器。

参数:

pt_hashtable	hashtable 容器。
t_bucketcount	hash 表中桶的个数。
t_hash	hash 函数。
t_compare	数据比较规则。

返回值:

无。

注意:

pt_hashtable == NULL 则函数的行为是未定义的，pt_hashtable 必须是使用_create_hashtable() 创建的，否则函数的行为是未定义的，如果 t_compare == NULL 则使用类型默认的比较函数。

```
void _hashtable_init_copy(_hashtable_t* pt_dest, const _hashtable_t* cpt_src);
```

描述:

使用已经存在的 hashtable 初始化 hashtable 迭代器。

参数:

pt_dest	目的 hashtable 容器。
cpt_src	源 hashtable 容器。

返回值:

无。

注意:

pt_dest == NULL 或者 cpt_src == NULL 则函数的行为是未定义的，pt_dest 必须是使用_create_hashtable() 创建的 cpt_src 必须是已经初始化，否则函数的行为是未定义的。pt_dest 和 cpt_src 保存的数据类型必须相同，否则函数的行为是未定义。

```
void _hashtable_init_copy_range(_hashtable_t* pt_dest,
    _hashtable_iterator_t it_begin, _hashtable_iterator_t it_end,
    size_t t_bucketcount, unary_function_t t_hash, binary_function_t t_compare);
```

描述:

使用指定的数据区间初始化 hashtable 迭代器。

参数:

pt_dest	目的 hashtable 容器。
it_begin	数据区间的开始。

it_end 数据区间的末尾。
t_bucketcount hash 表中桶的个数。
t_hash hash 函数。
t_compare 数据比较规则。

返回值:

无。

注意:

如果 `pt_dest == NULL` 则函数的行为是未定义的, `pt_dest` 必须是使用 `_create_hashtable()` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 hashtable 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void _hashtable_destroy(_hashtable_t* pt_hashtable);
```

描述:

销毁创建的 hashtable 容器。

参数:

pt_hashtable hashtable 容器。

返回值:

无。

注意:

如果 `pt_hashtable == NULL` 则函数的行为是未定义的, `pt_hashtable` 必须是初始化或者是使用 `_create_hashtable()` 创建的, 否则函数的行为是未定义的。

```
void _hashtable_assign(_hashtable_t* pt_dest, const _hashtable_t* cpt_src);
```

描述:

使用一个 hashtable 为另一个 hashtable 赋值。

参数:

pt_dest 指向被赋值的 hashtable 容器的指针。
cpt_src 指向赋值的 hashtable 容器的指针。

返回值:

无。

注意:

如果 `pt_dest == NULL` 或者 `cpt_src == NULL` 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同函数的行为是未定义的。如果 `_hashtable_equal(pt_dest, cpt_src)` 那么函数不做任何动作。

```
size_t _hashtable_size(const _hashtable_t* cpt_hashtable);
```

描述:

获得 hashtable 容器中保存的数据的个数。

参数:

cpt_hashtable hashtable 容器。

返回值:

hashtable 容器中保存的数据的个数。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, cpt_hashtable 必须是已经初始化的 hashtable 容器, 否则函数的行为是未定义的。

```
bool_t _hashtable_empty(const _hashtable_t* cpt_hashtable);
```

描述:

测试 hashtable 容器是否为空。

参数:

cpt_hashtable hashtable 容器。

返回值:

如果 hashtable 容器为空则返回 true, 否则返回 false。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, cpt_hashtable 必须是已经初始化的 hashtable 容器, 否则函数的行为是未定义的。

```
size_t _hashtable_max_size(const _hashtable_t* cpt_hashtable);
```

描述:

获得 hashtable 容器中数据的最大数量。

参数:

cpt_hashtable hashtable 容器。

返回值:

hashtable 容器中保存的数据的个数的最大值。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, cpt_hashtable 必须是已经初始化的 hashtable 容器, 否则函数的行为是未定义的。

```
size_t _hashtable_bucket_count(const _hashtable_t* cpt_hashtable);
```

描述:

获得 hashtable 容器中桶的数量。

参数:

cpt_hashtable hashtable 容器。

返回值:

hashtable 容器桶的数量。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, cpt_hashtable 必须是已经初始化的 hashtable 容器, 否则函数的行为是未定义的。

```
_hashtable_iterator_t _hashtable_begin(const _hashtable_t* cpt_hashtable);
```

描述:

获得引用 hashtable 容器中第一数据的迭代器。

参数:

cpt_hashtable 指向 hashtable 容器的指针。

返回值:

返回引用 hashtable 容器中第一个数据的迭代器。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的, 否则函数的行为是未定义的。如果 hashtable 容器为空, 则返回值与 _hashtable_end(cpt_hashtable) 相等。

```
_hashtable_iterator_t _hashtable_end(const _hashtable_t* cpt_hashtable);
```

描述:

获得引用 hashtable 容器末尾的迭代器。

参数:

cpt_hashtable 指向 hashtable 容器的指针。

返回值:

返回引用 hashtable 容器末尾的迭代器。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t _hashtable_hash(const _hashtable_t* cpt_hashtable);
```

描述:

返回 hashtable 容器的 hash 函数。

参数:

cpt_hashtable 指向 hashtable 容器的指针。

返回值:

返回 hashtable 容器的 hash 函数。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t _hashtable_key_comp(const _hashtable_t* cpt_hashtable);
```

描述:

返回 hashtable 容器中的数据比较规则。

参数:

cpt_hashtable 指向 hashtable 容器的指针。

返回值:

返回 hashtable 容器中的数据比较规则。

注意:

如果 cpt_hashtable == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的, 否则函数的行为是未定义的。

```
_hashtable_iterator_t _hashtable_find(
    const _hashtable_t* cpt_hashtable, const void* cpv_value);
```

描述:

在 hashtable 容器中查找指定的数据。

参数:

- cpt_hashtable 指向 hashtable 容器的指针。
- cpv_value 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cpt_hashtable == NULL 或者 cpv_value == NULL 则函数的行为是未定义的， hashtable 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hashtable 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _hashtable_clear(_hashtable_t* pt_hashtable);
```

描述:

删除 hashtable 容器中的所有数据。

参数:

- pt_hashtable 指向 hashtable 容器的指针。

返回值:

无。

注意:

如果 pt_hashtable == NULL 则函数的行为是未定义的， hashtable 容器必须已经初始化的，否则函数的行为是未定义的。

```
size_t _hashtable_count(const _hashtable_t* cpt_hashtable, const void* cpv_value);
```

描述:

返回 hashtable 容器中指定数据的数量。

参数:

- cpt_hashtable 指向 hashtable 容器的指针。
- cpv_value 要查找的指定数据。

返回值:

返回 hashtable 容器中指定数据的数量。

注意:

如果 cpt_hashtable == NULL 或者 cpv_value == NULL 则函数的行为是未定义的， hashtable 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hashtable 容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _hashtable_equal_range(
    const _hashtable_t* cpt_hashtable, const void* cpv_value);
```

描述:

返回包含指定数据的数据区间。

参数:

- cpt_hashtable 指向 hashtable 容器的指针。
- cpv_value 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_hashtable == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hashtable 容器中的数据类型相同, 否则函数的行为是未定义的。

```
bool_t _hashtable_equal(
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

测试两个 hashtable 容器是否相等。

参数:

cpt_first 指向 hashtable 容器的指针。
cpt_second 指向 hashtable 容器的指针。

返回值:

如果两个 hashtable 容器相等则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同认为这两个容器不等。两个 hashtable 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cpt_first == cpt_second 那么返回 true。

```
bool_t _hashtable_not_equal(
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

测试两个 hashtable 容器是否不等。

参数:

cpt_first 指向 hashtable 容器的指针。
cpt_second 指向 hashtable 容器的指针。

返回值:

如果两个 hashtable 容器不等则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同认为两个 hashtable 容器不等。两个 hashtable 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cpt_first == cpt_second 那么返回 false。

```
bool_t _hashtable_less(
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

测试第一个 hashtable 是否小于第二个 hashtable。

参数:

cpt_first 指向 hashtable 容器的指针。

cpt_second 指向 hashtable 容器的指针。

返回值:

如果第一个 hashtable 容器小于第二个 hashtable 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hashtable 中的数据小于第二个 hashtable 中对应的数据则返回 true, 如果大于则返回 false, 当两个 hashtable 中的对应数据相等, 第一个 hashtable 中的数据数量小于第二个 hashtable 中数据的数量返回 true, 否则返回 false。如果 cpt_first == cpt_second 那么返回 false。

```
bool_t _hashtable_less_equal(
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

测试第一个 hashtable 是否小于等于第二个 hashtable。

参数:

cpt_first 指向 hashtable 容器的指针。

cpt_second 指向 hashtable 容器的指针。

返回值:

如果第一个 hashtable 容器小于等于第二个 hashtable 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同函数的行为是未定义的。如果 cpt_first == cpt_second 那么返回 true。

```
bool_t _hashtable_greater(
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

测试第一个 hashtable 是否大于第二个 hashtable。

参数:

cpt_first 指向 hashtable 容器的指针。

cpt_second 指向 hashtable 容器的指针。

返回值:

如果第一个 hashtable 容器大于第二个 hashtable 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同函数的行为是未定义的。如果第一 hashtable 中的数据大于第二个 hashtable 中对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 hashtable 中数据的数量大于第二个 hashtable 中数据的数量的时候返回 true 否则返回 false。如果 cpdeqt_first == cpt_second 那么返回 false。

```
bool_t _hashtable_greater_equal(
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

测试第一个 hashtable 是否大于等于第二个 hashtable。

参数:

cpt_first 指向 hashtable 容器的指针。
cpt_second 指向 hashtable 容器的指针。

返回值:

如果第一个 hashtable 容器大于等于第二个 hashtable 容器则返回 true, 否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同函数的行为是未定义的。如果 cpt_first == cpt_second 那么返回 true。

```
void _hashtable_swap(_hashtable_t* pt_first, _hashtable_t* pt_second);
```

描述:

交换两个 hashtable 容器。

参数:

pt_first 指向 hashtable 容器的指针。
pt_second 指向 hashtable 容器的指针。

返回值:

无。

注意:

如果 pt_first == NULL 或者 pt_second == NULL 则函数的行为是未定义的, 两个 hashtable 必须已经初始化的, 否则函数的行为是未定义的。两个 hashtable 容器中保存的数据类型不同函数的行为是未定义的。如果 _hashtable_equal(pt_first, pt_second), 函数不执行任何动作。

```
_hashtable_iterator_t _hashtable_insert_unique(  
  _hashtable_t* pt_hashtable, const void* cpv_value);
```

描述:

向 hashtable 容器中插入唯一的数据。

参数:

pt_hashtable 指向 hashtable 容器的指针。
cpv_value 要插入的指定数据。

返回值:

指向插入的数据的迭代器, 如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_hashtable == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的否则函数的行为是未定义的。插入的数据必须与保存在 hashtable 容器中的数据类型相同, 否则函数的行为是未定义的。

```
_hashtable_iterator_t _hashtable_insert_equal(  
  _hashtable_t* pt_hashtable, const void* cpv_value);
```

描述:

向 hashtable 容器中插入数据, 数据可以重复。

参数:

- pt_hashtable 指向 hashtable 容器的指针。
- cpv_value 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_hashtable == NULL 或者 cpv_value == NULL 则函数的行为是未定义的，hashtable 容器必须已经初始化的否则函数的行为是未定义的。插入的数据必须与保存在 hashtable 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _hashtable_insert_unique_range(
    _hashtable_t* pt_hashtable,
    _hashtable_iterator_t it_begin, _hashtable_iterator_t it_end);
```

描述:

向 hashtable 中插入指定的数据区间，数据必须唯一。

参数:

- pt_hashtable 指向 hashtable 容器的指针。
- it_begin 指定的数据区间的开头。
- it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pt_hashtable == NULL 则函数的行为是未定义的，hashtable 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end) 必须是有效的数据区间并且与保存在 hashtable 容器中的数据类型相同，否则函数的行为是未定义的。[it_begin, it_end) 必须不属于 pt_hashtable，否则函数的行为是未定义的。

```
void _hashtable_insert_equal_range(
    _hashtable_t* pt_hashtable,
    _hashtable_iterator_t it_begin, _hashtable_iterator_t it_end);
```

描述:

向 hashtable 中插入指定的数据区间。

参数:

- pt_hashtable 指向 hashtable 容器的指针。
- it_begin 指定的数据区间的开头。
- it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pt_hashtable == NULL 则函数的行为是未定义的，hashtable 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end) 必须是有效的数据区间并且与保存在 hashtable 容器中的数据类型相同，否则函数的行为是未定义的。[it_begin, it_end) 必须不属于 pt_hashtable，否则函数的行为是未定义的。

```
void _hashtable_erase_pos(_hashtable_t* pt_hashtable, _hashtable_iterator_t it_pos);
```

描述:

删除 hashtable 容器中指定位置的数据。

参数:

pt_hashtable 指向 hashtable 容器的指针。

it_pos 被删除的数据的位置。

返回值:

无。

注意:

如果 pt_hashtable == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 是属于 pt_hashtable 容器的有效的迭代器, 否则函数的行为是未定义的。

```
void _hashtable_erase_range(_hashtable_t* pt_hashtable,
    _hashtable_iterator_t it_begin, _hashtable_iterator_t it_end);
```

描述:

删除 hashtable 中指定数据区间的数据。

参数:

pt_hashtable 指向 hashtable 容器的指针。

it_begin 指定的数据区间的开头。

it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 pt_hashtable == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end) 必须属于 pt_hashtable, 否则函数的行为是未定义的。

```
size_t _hashtable_erase(_hashtable_t* pt_hashtable, const void* cpv_value);
```

描述:

删除 hashtable 中指定的数据。

参数:

pt_hashtable 指向 hashtable 容器的指针。

cpv_value 要删除的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 pt_hashtable == NULL 或者 cpv_value == NULL 则函数的行为是未定义的, hashtable 容器必须已经初始化的否则函数的行为是未定义的。指定的数据必须与保存在 hashtable 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _hashtable_resize(_hashtable_t* pt_hashtable, size_t t_resize);
```

描述:

重新设置 hashtable 的桶数。

参数:

pt_hashtable 指向 hashtable 容器的指针。

`t_resize` 新的桶数

返回值:

无。

注意:

如果 `pt_hashtable == NULL` 则函数的行为是未定义的, `hashtable` 容器必须已经初始化的, 否则函数的行为是未定义的。

第五节 迭代器接口

`hashtable` 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

<code>_create_hashtable_iterator</code>	创建 <code>hashtable</code> 迭代器。
<code>_hashtable_iterator_get_value</code>	获得 <code>hashtable</code> 迭代器引用的数据。
<code>_hashtable_iterator_get_pointer</code>	获得 <code>hashtable</code> 迭代器引用的数据的指针。
<code>_hashtable_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_hashtable_iterator_prev</code>	获得引用前一个数据的迭代器。
<code>_hashtable_iterator_equal</code>	测试两个迭代器是否相等。
<code>_hashtable_iterator_distance</code>	计算两个迭代器之间的距离。
<code>_hashtable_iterator_before</code>	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
_hashtable_iterator_t _create_hashtable_iterator(void);
```

描述:

创建一个 `hashtable` 迭代器。

参数:

无。

返回值:

`hashtable` 迭代器。

注意:

返回的 `hashtable` 迭代器并不是有效的迭代器, 它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作, 返回的迭代器供其他接口使用。

```
void _hashtable_iterator_get_value(_hashtable_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

`it_iter` `hashtable` 迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意:

it_iter 是有效的 hashtable 迭代器, 否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的, pv_value 是能够保存下 it_iter 引用的数据的缓冲区, 否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _hashtable_iterator_get_pointer(_hashtable_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter hashtable 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 hashtable 迭代器, 否则函数的行为是未定义的。

```
_hashtable_iterator_t _hashtable_iterator_next(_hashtable_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter hashtable 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 hashtable 迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 hashtable 有效的迭代器, 否则函数的行为是未定义的。

```
_hashtable_iterator_t _hashtable_iterator_prev(_hashtable_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter hashtable 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 hashtable 迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 hashtable 有效的迭代器, 否则函数的行为是未定义的。

```
bool_t _hashtable_iterator_equal(
    _hashtable_iterator_t it_first, _hashtable_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first hashtable 迭代器。
it_second hashtable 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 hashtable 容器的有效的 hashtable 迭代器, 否则函数的行为是未定义的。两个 hashtable 迭代器相等是指两个迭代器引用相同的数据。

```
int _hashtable_iterator_distance(  
    _hashtable_iterator_t it_first, _hashtable_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_iterfirst hashtable 迭代器。
it_itersecond hashtable 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 hashtable 容器的有效的 hashtable 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为 0。

```
bool_t _hashtable_iterator_before(  
    _hashtable_iterator_t it_first, _hashtable_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first hashtable 迭代器。
it_second hashtable 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 hashtable 容器的有效的 hashtable 迭代器, 否则函数的行为是未定义的。

第六节 内部和辅助接口

hashtable 也只是提供了简单的内部接口。

<code>_create_hashtable_auxiliary</code>	创建 hashtable 容器的辅助函数。
<code>_hashtable_destroy_auxiliary</code>	销毁 hashtable 容器的辅助函数。

```
bool_t _create_hashtable_auxiliary(
    _hashtable_t* pt_hashtable, const char* s_typename);
```

描述:

创建一个 hashtable 容器的辅助函数。

参数:

<code>pt_hashtable</code>	没有创建的 hashtable 容器。
<code>s_typename</code>	hashtable 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `pt_hashtable == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。 `s_typename` 必须是 C 内建类型, `libcstl` 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _hashtable_destroy_auxiliary(_hashtable_t* pt_hashtable);
```

描述:

销毁 hashtable 容器的辅助函数。

参数:

<code>pt_hashtable</code>	hashtable 容器。
---------------------------	---------------

返回值:

无。

注意:

如果 `pt_hashtable == NULL` 或者 hashtable 不是使用 `_create_hashtable` 生成的则函数的行为是未定义的。

hashtable 提供了大量的辅助函数, 这些函数用于检测结构的有效性, 对 hashtable 进行调整等等。

<code>_hashtable_is_created</code>	测试 hashtable 容器是否是由 <code>_create_hashtable</code> 函数创建的。
<code>_hashtable_is_inited</code>	测试 hashtable 容器是否已经被初始化。
<code>_hashtable_iterator_belong_to_hashtable</code>	测试一个迭代器是否属于指定的 hashtable 容器。
<code>_hashtable_same_hashtable_iterator_type</code>	测试 hashtable 容器中保存的数据类型和迭代器引用的数据类型是否相等。
<code>_hashtable_same_hashtable_iterator_type_ex</code>	测试 hashtable 容器中保存的数据类型和迭代器引用的数据类型是否相等。
<code>_hashtable_same_type</code>	测试两个 hashtable 保存的数据类型是否相同。
<code>_hashtable_same_type_ex</code>	测试两个 hashtable 保存的数据类型以及比较函数是否相同。
<code>_hashtable_get_prime</code>	根据指定的数值来获取大于等于该值的最小的质数。
<code>_hashtable_default_hash</code>	默认的 hash 函数。
<code>_hashtable_init_elem_auxiliary</code>	初始化数据的辅助函数。
<code>_hashtable_hash_auxiliary</code>	hash 函数的辅助函数。

`_hashtable_elem_compare_auxiliary`

比较数据的辅助函数。

函数原型

```
bool_t _hashtable_is_created(const _hashtable_t* cpt_hashtable);
```

描述:

测试一个 hashtable 是否是使用 `_create_hashtable` 创建的。

参数:

`cpt_hashtable` hashtable 容器。

返回值:

如果 hashtable 是使用 `_create_hashtable` 创建的则返回 `true`, 否则返回 `false`。

注意:

如果 `cpt_hashtable == NULL`, 函数的行为是未定义的。

```
bool_t _hashtable_is_inited(const _hashtable_t* cpt_hashtable);
```

描述:

测试一个 hashtable 是否已经初始化。

参数:

`cpt_hashtable` hashtable 容器。

返回值:

如果 hashtable 已经初始化了, 返回 `true`, 否则返回 `false`。

注意:

如果 `cpt_hashtable == NULL`, 函数的行为是未定义的。

```
bool_t _hashtable_iterator_belong_to_hashtable(
    const _hashtable_t* cpt_hashtable, _hashtable_iterator_t it_iter);
```

描述:

测试一个迭代器是否属于指定的 hashtable 容器。

参数:

`cpt_hashtable` hashtable 容器。

`it_iter` hashtable 迭代器。

返回值:

如果迭代器引用的数据属于 hashtable 容器的有效范围内, 返回 `true`, 否则返回 `false`。

注意:

如果 `cpt_hashtable == NULL`, 那么函数的行为是未定义的, `cpt_hashtable` 必须是已经初始化的, 否则函数的行为是未定义的。如果 `it_iter` 不是 hashtable 迭代器, 那么函数的行为是未定义的。

```
bool_t _hashtable_same_hashtable_iterator_type(
    const _hashtable_t* cpt_hashtable, _hashtable_iterator_t it_iter);
```

描述:

测试 hashtable 容器中保存的数据类型和迭代器引用的数据类型是否相等。

参数:

`cpt_hashtable` hashtable 容器。

it_iter hashtable 迭代器。

返回值:

如果 hashtable 中保存的数据类型与 it_iter 引用的数据类型相同返回 true，否则返回 false。

注意:

如果 cpt_hashtable == NULL 或者 it_iter 不是 hashtable_iterator_t 类型，那么函数的行为是未定义的。cpt_hashtable 必须是使用 create_hashtable() 创建的，否则函数的行为是未定义的。

```
bool_t _hashtable_same_hashtable_iterator_type_ex(  
    const _hashtable_t* cpt_hashtable, _hashtable_iterator_t it_iter);
```

描述:

测试 hashtable 容器中保存的数据类型和迭代器引用的数据类型是否相等。

参数:

cpt_hashtable hashtable 容器。
it_iter hashtable 迭代器。

返回值:

如果 hashtable 中保存的数据类型与 it_iter 引用的数据类型相同返回 true，否则返回 false。

注意:

如果 cpt_hashtable == NULL 或者 it_iter 不是 hashtable_iterator_t 类型，那么函数的行为是未定义的。cpt_hashtable 必须是使用 create_hashtable() 创建的，否则函数的行为是未定义的。

```
bool_t _hashtable_same_type(  
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

判断两个 hashtable 容器中保存的数据类型是否相同。

参数:

cpt_first 第一个 hashtable 容器。
cpt_second 第二个 hashtable 容器。

返回值:

如果两个 hashtable 中保存的数据类型相同返回 true，否则返回 false。

注意:

如果 cpt_first == NULL 或者 cpt_second == NULL，那么函数的行为是未定义的。两个 hashtable 容器必须是已经初始化或者使用 create_hashtable 创建的 hashtable 容器，否则函数的行为是未定义的。如果 cpt_first == cpt_second 则返回 true。

```
bool_t _hashtable_same_type_ex(  
    const _hashtable_t* cpt_first, const _hashtable_t* cpt_second);
```

描述:

判断两个 hashtable 容器中保存的数据类型以及比较函数是否相同。

参数:

cpt_first 第一个 hashtable 容器。
cpt_second 第二个 hashtable 容器。

返回值:

如果两个 hashtable 中保存的数据类型以及比较函数相同返回 true，否则返回 false。

注意：

如果 cpt_first == NULL 或者 cpt_second == NULL，那么函数的行为是未定义的。两个 hashtable 容器必须是已经初始化或者使用_create_hashtable 创建的 hashtable 容器，否则函数的行为是未定义的。如果 cpt_first == cpt_second 则返回 true。

```
unsigned long _hashtable_get_prime(unsigned long ul_basenum);
```

描述：

根据指定的数值来获取大于等于该值的最小的质数。

参数：

ul_basenum 用户指定的基础数据。

返回值：

大于等于该值的最小的质数。

注意：

如果 ul_basenum 大于最大的质数，那么直接返回 ul_basenum。

```
void _hashtable_default_hash(const void* cpv_input, void* pv_output);
```

描述：

默认的 hash 函数。

参数：

cpv_first 输入参数。

pv_output 输出参数

返回值：

无。

注意：

如果 cpv_first == NULL 或者 pv_output == NULL 则函数的行为是未定义的。

```
void _hashtable_init_elem_auxiliary(
    _hashtable_t* pt_hashtable, _hashnode_t* pt_node);
```

描述：

初始化数据的辅助函数。

参数：

pt_hashtable hashtable 容器。

pt_node 初始化节点。

返回值：

无。

注意：

如果 pt_hashtable == NULL 或者 pt_node == NULL，那么函数的行为是未定义的。pt_hashtable 必须是初始化的或者是使用_create_hashtable()创建的，否则函数的行为是未定义的。

```
void _hashtable_hash_auxiliary(
    const _hashtable_t* cpt_hashtable, const void* cpv_input, void* pv_output);
```

描述:

hash 函数的辅助函数。

参数:

cpt_hashtable	hashtable 容器。
cpv_input	输入的参数。
pv_output	输出参数。

返回值:

无。

注意:

如果 cpt_hashtable == NULL, cpv_input == NULL 或者 pv_output , 那么函数的行为是未定义的。cpt_hashtable 必须是初始化的, 否则函数的行为是未定义的。

```
void _hashtable_elem_compare_auxiliary(const _hashtable_t* cpt_hashtable,
    const void* cpv_first, const void* cpv_second, void* pv_output);
```

描述:

比较数据的辅助函数。

参数:

cpt_hashtable	hashtable 容器。
cpv_first	第一个数据。
cpv_second	第二个数据。
pv_output	比较结果。

返回值:

无。

注意:

如果 cpt_hashtable == NULL 或者 cpv_first == NULL 或者 cpv_second == NULL 或者 pv_output == NULL , 那么函数的行为是未定义的。cpt_hashtable 必须是初始化的或者是使用_create_hashtable()创建的, 否则函数的行为是未定义的。

第二十章 hash_set 和 hash_multiset

第一节 hash_set 和 hash_multiset 的内部机制

hash_set 和 hash_multiset 是集合的抽象，集合中的数据是有序的，可以实现数据的快速查找。hash_set 和 hash_multiset 的底层使用 hashtable 实现，hash_set 和 hash_multiset 是对于 hashtable 的封装，它们的区别就是 hash_set 中不包含重复的元素，hash_multiset 允许数据的重复。

第二节 hash_set 和 hash_multiset 的迭代器

hash_set 和 hash_multiset 的迭代器就是使用 hashtable 的迭代器实现。

第三节 hash_set 和 hash_multiset 的代码结构

hash_set 的代码结构如下

```
typedef struct _taghashset
{
    _hashtable_t _t_hashtable;
}hash_set_t;
```

第四节 外部接口

hash_set 提供的外部接口提供给用户使用。

create_hash_set	创建 hash_set 容器。
hash_set_init	初始化 hash_set 容器。
hash_set_init_ex	使用自定义比较规则初始化 hash_set 容器。
hash_set_init_copy	使用存在的 hash_set 容器进行初始化。
hash_set_init_copy_range	使用指定的数据区间进行初始化。

hash_set_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
hash_set_destroy	销毁 hash_set 容器。
hash_set_assign	使用存在的 hash_set 容器赋值。
hash_set_size	返回 hash_set 容器中数据的数量。
hash_set_empty	判断 hash_set 容器是否为空。
hash_set_max_size	返回 hash_set 容器中能够保存数据的最大数量。
hash_set_begin	返回指向 hash_set 容器的第一个数据的迭代器。
hash_set_end	返回指向 hash_set 容器末尾的迭代器。
hash_set_key_comp	返回 hash_set 容器中的键比较规则。
hash_set_value_comp	返回 hash_set 容器中的数据比较规则。
hash_set_find	在 hash_set 容器中查找指定的数据。
hash_set_clear	清空 hash_set 容器。
hash_set_count	返回 hash_set 容器中指定数据的数量。
hash_set_equal_range	返回包含指定数据的数据区间。
hash_set_equal	测试两个 hash_set 容器是否相等。
hash_set_not_equal	测试两个 hash_set 容器是否不等。
hash_set_less	测试第一个 hash_set 容器是否小于第二个 hash_set 容器。
hash_set_less_equal	测试第一个 hash_set 容器是否小于等于第二个 hash_set 容器。
hash_set_greater	测试第一个 hash_set 容器是否大于第二个 hash_set 容器。
hash_set_greater_equal	测试第一个 hash_set 容器是否大于等于第二个 hash_set 容器。
hash_set_swap	交换两个 hash_set 容器。
hash_set_insert	向 hash_set 中插入数据，数据必须唯一。
hash_set_insert_range	向 hash_set 中插入指定的数据区间，数据必须唯一。
hash_set_erase_pos	删除 hash_set 中指定位置的数据。
hash_set_erase_range	删除 hash_set 中指定数据区间的数据。
hash_set_erase	删除 hash_set 中指定的数据。

函数原型

```
hash_set_t* create_hash_set(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

typename 类型描述。

返回值:

成功返回指向 hash_set 容器的指针，否则返回 NULL。

注意：

关于类型描述参考第四章，创建失败返回 NULL。

```
void hash_set_init(hash_set_t* phset_set);
```

描述：

初始化 hash_set 迭代器。

参数：

phset_set hash_set 容器。

返回值：

无。

注意：

phset_set == NULL 则函数的行为是未定义的，phset_set 必须是使用 create_hash_set() 创建的，否则函数的行为是未定义的，这个函数使用类型默认的比较函数。

```
void hash_set_init_copy(hash_set_t* phset_dest, const hash_set_t* cphset_src);
```

描述：

使用已经存在的 hash_set 初始化 hash_set 迭代器。

参数：

phset_dest 目的 hash_set 容器。
cphash_sett_src 源 hash_set 容器。

返回值：

无。

注意：

phset_dest == NULL 或者 cphash_sett_src == NULL 则函数的行为是未定义的，phset_dest 必须是使用 create_hash_set() 创建的，cphash_sett_src 必须是已经初始化，否则函数的行为是未定义的。phset_dest 和 cphash_sett_src 保存的数据类型必须相同，否则函数的行为是未定义。

```
void hash_set_init_copy_range(  
    hash_set_t* phset_dest,  
    hash_set_iterator_t it_begin, hash_set_iterator_t it_end);
```

描述：

使用指定的数据区间初始化 rb tree 迭代器。

参数：

phset_dest 目的 hash_set 容器。
it_begin 数据区间的开始。
it_end 数据区间的末尾。

返回值：

无。

注意：

如果 phset_dest == NULL 则函数的行为是未定义的，phset_dest 必须是使用 create_hash_set() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 hash_set 容器的有效数据区间，否则函数的行为是未定义的。

容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void hash_set_init_copy_range_ex(
    hash_set_t* pt_dest, hash_set_iterator_t it_begin, hash_set_iterator_t it_end,
    binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 hash_set 迭代器。

参数:

phset_dest	目的 hash_set 容器。
it_begin	数据区间的开始。
it_end	数据区间的末尾。
t_compare	比较函数。

返回值:

无。

注意:

如果 phset_dest == NULL 则函数的行为是未定义的, phset_dest 必须是使用 create_hash_set() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 hash_set 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 t_compare == NULL 则使用类型默认的比较函数。

```
void hash_set_destroy(hash_set_t* phset_set);
```

描述:

销毁创建的 hash_set 容器。

参数:

phset_set	hash_set 容器。
-----------	--------------

返回值:

无。

注意:

如果 phset_set == NULL 则函数的行为是未定义的, phset_set 必须是初始化或者是使用 create_hash_set() 创建的, 否则函数的行为是未定义的。

```
void hash_set_assign(hash_set_t* phset_dest, const hash_set_t* cphset_src);
```

描述:

使用一个 hash_set 为另一个 hash_set 赋值。

参数:

phset_dest	指向被赋值的 hash_set 容器的指针。
cphset_src	指向赋值的 hash_set 容器的指针。

返回值:

无。

注意:

如果 phset_dest == NULL 或者 cphset_src == NULL 则函数的行为是未定义的, 两个 hash_set 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_set 容器中保存的数据类型不同函数的行为是未定义的。如果

hash_set_equal(phset_dest, cphset_src)那么函数不做任何动作。

```
size_t hash_set_size(const hash_set_t* cphset_set);
```

描述:

获得 hash_set 容器中保存的数据的个数。

参数:

cphset_set hash_set 容器。

返回值:

hash_set 容器中保存的数据的个数。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的, cphset_set 必须是已经初始化的 hash_set 容器, 否则函数的行为是未定义的。

```
bool_t hash_set_empty(const hash_set_t* cphset_set);
```

描述:

测试 hash_set 容器是否为空。

参数:

cphset_set hash_set 容器。

返回值:

如果 hash_set 容器为空则返回 true, 否则返回 false。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的, cphset_set 必须是已经初始化的 hash_set 容器, 否则函数的行为是未定义的。

```
size_t hash_set_max_size(const hash_set_t* cphset_set);
```

描述:

获得 hash_set 容器中数据的最大数量。

参数:

cphset_set hash_set 容器。

返回值:

hash_set 容器中保存的数据的个数的最大值。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的, cphset_set 必须是已经初始化的 hash_set 容器, 否则函数的行为是未定义的。

```
hash_set_iterator_t hash_set_begin(const hash_set_t* cphset_set);
```

描述:

获得引用 hash_set 容器中第一数据的迭代器。

参数:

cphset_set 指向 hash_set 容器的指针。

返回值:

返回引用 hash_set 容器中第一个数据的迭代器。

注意:

如果 `cphset_set == NULL` 则函数的行为是未定义的, `hash_set` 容器必须已经初始化的, 否则函数的行为是未定义的。如果 `hash_set` 容器为空, 则返回值与 `hash_set_end(cphset_set)` 相等。

```
hash_set_iterator_t hash_set_end(const hash_set_t* cphset_set);
```

描述:

获得引用 `hash_set` 容器末尾的迭代器。

参数:

`cphset_set` 指向 `hash_set` 容器的指针。

返回值:

返回引用 `hash_set` 容器末尾的迭代器。

注意:

如果 `cphset_set == NULL` 则函数的行为是未定义的, `hash_set` 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t hash_set_key_comp(const hash_set_t* cphset_set);
```

描述:

返回 `hash_set` 容器中的键比较规则。

参数:

`cphset_set` 指向 `hash_set` 容器的指针。

返回值:

返回 `hash_set` 容器中的数据比较规则。

注意:

如果 `cphset_set == NULL` 则函数的行为是未定义的, `hash_set` 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t hash_set_value_comp(const hash_set_t* cphset_set);
```

描述:

返回 `hash_set` 容器中的数据比较规则。

参数:

`cphset_set` 指向 `hash_set` 容器的指针。

返回值:

返回 `hash_set` 容器中的数据比较规则。

注意:

如果 `cphset_set == NULL` 则函数的行为是未定义的, `hash_set` 容器必须已经初始化的, 否则函数的行为是未定义的。

```
hash_set_iterator_t hash_set_find(const hash_set_t* cphset_set, elem);
```

描述:

在 `hash_set` 容器中查找指定的数据。

参数:

`cphset_set` 指向 `hash_set` 容器的指针。

elem 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 `cphset_set == NULL` 则函数的行为是未定义的，`hash_set` 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 `hash_set` 容器中的数据类型相同，否则函数的行为是未定义的。

```
void hash_set_clear(hash_set_t* phset_set);
```

描述:

删除 `hash_set` 容器中的所有数据。

参数:

phset_set 指向 `hash_set` 容器的指针。

返回值:

无。

注意:

如果 `phset_set == NULL` 则函数的行为是未定义的，`hash_set` 容器必须已经初始化的，否则函数的行为是未定义的。

```
size_t hash_set_count(const hash_set_t* cphset_set, elem);
```

描述:

返回 `hash_set` 容器中指定数据的数量。

参数:

cphset_set 指向 `hash_set` 容器的指针。

elem 要查找的指定数据。

返回值:

返回 `hash_set` 容器中指定数据的数量。

注意:

如果 `cphset_set == NULL` 则函数的行为是未定义的，`hash_set` 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 `hash_set` 容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t hash_set_equal_range(const hash_set_t* cphset_set, elem);
```

描述:

返回包含指定数据的数据区间。

参数:

cphset_set 指向 `hash_set` 容器的指针。

elem 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 `cpt_hash_set == NULL` 则函数的行为是未定义的，`hash_set` 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 `hash_set` 容器中的数据类型相同，否则函数的行为是未定义的。

```
bool_t hash_set_equal(
```

```
const hash_set_t* cphset_equal(
```

描述:

测试两个 hash_set 容器是否相等。

参数:

cphset_first	指向 hash_set 容器的指针。
cphset_second	指向 hash_set 容器的指针。

返回值:

如果两个 hash_set 容器相等则返回 true, 否则返回 false。

注意:

如果 cphset_first == NULL 或者 cphset_second == NULL 则函数的行为是未定义的, 两个 hash_set 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_set 容器中保存的数据类型不同认为这两个容器不等。两个 hash_set 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cphset_first == cphset_second 那么返回 true。

```
bool_t hash_set_not_equal(
```



```
    const hash_set_t* cphset_first, const hash_set_t* cphset_second);
```

描述:

测试两个 hash_set 容器是否不等。

参数:

cphset_first	指向 hash_set 容器的指针。
cphset_second	指向 hash_set 容器的指针。

返回值:

如果两个 hash_set 容器不等则返回 true, 否则返回 false。

注意:

如果 cphset_first == NULL 或者 cphset_second == NULL 则函数的行为是未定义的, 两个 hash_set 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_set 容器中保存的数据类型不同认为两个 hash_set 容器不等。两个 hash_set 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cphset_first == cphset_second 那么返回 false。

```
bool_t hash_set_less(
```



```
    const hash_set_t* cphset_first, const hash_set_t* cphset_second);
```

描述:

测试第一个 hash_set 是否小于第二个 hash_set。

参数:

cphset_first	指向 hash_set 容器的指针。
cphset_second	指向 hash_set 容器的指针。

返回值:

如果第一个 hash_set 容器小于第二个 hash_set 容器则返回 true, 否则返回 false。

注意:

如果 cphset_first == NULL 或者 cphset_second == NULL 则函数的行为是未定义的, 两个 hash_set 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_set 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hash_set 中的数据小于第二个 hash_set 中对应的数据则返回 true, 如果大于则返回 false, 当两个 hash_set 中的对应数据相等, 第一个 hash_set 中的数据数量小于第二个 hash_set 中数据的数量返回 true, 否则返回 false。如果 cphset_first ==

cphset_second 那么返回 false。

```
bool_t hash_set_less_equal(
    const hash_set_t* cphset_first, const hash_set_t* cphset_second);
```

描述:

测试第一个 hash_set 是否小于等于第二个 hash_set。

参数:

cphset_first	指向 hash_set 容器的指针。
cphset_second	指向 hash_set 容器的指针。

返回值:

如果第一个 hash_set 容器小于等于第二个 hash_set 容器则返回 true, 否则返回 false。

注意:

如果 cphset_first == NULL 或者 cphset_second == NULL 则函数的行为是未定义的, 两个 hash_set 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_set 容器中保存的数据类型不同函数的行为是未定义的。如果 cphset_first == cphset_second 那么返回 true。

```
bool_t hash_set_greater(
    const hash_set_t* cphset_first, const hash_set_t* cphset_second);
```

描述:

测试第一个 hash_set 是否大于第二个 hash_set。

参数:

cphset_first	指向 hash_set 容器的指针。
cphset_second	指向 hash_set 容器的指针。

返回值:

如果第一个 hash_set 容器大于第二个 hash_set 容器则返回 true, 否则返回 false。

注意:

如果 cphset_first == NULL 或者 cphset_second == NULL 则函数的行为是未定义的, 两个 hash_set 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_set 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hash_set 中的数据大于第二个 hash_set 中的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 hash_set 中数据的数量大于第二个 hash_set 中数据的数量的时候返回 true 否则返回 false。如果 cphset_first == cphset_second 那么返回 false。

```
bool_t hash_set_greater_equal(
    const hash_set_t* cphset_first, const hash_set_t* cphset_second);
```

描述:

测试第一个 hash_set 是否大于等于第二个 hash_set。

参数:

cphset_first	指向 hash_set 容器的指针。
cphset_second	指向 hash_set 容器的指针。

返回值:

如果第一个 hash_set 容器大于等于第二个 hash_set 容器则返回 true, 否则返回 false。

注意:

如果 `cphset_first == NULL` 或者 `cpt_second == NULL` 则函数的行为是未定义的，两个 `hash_set` 必须已经初始化的，否则函数的行为是未定义的。两个 `hash_set` 容器中保存的数据类型不同函数的行为是未定义的。如果 `cphset_first == cphset_second` 那么返回 `true`。

```
void hash_set_swap(hash_set_t* phset_first, hash_set_t* phset_second);
```

描述:

交换两个 `hash_set` 容器。

参数:

`phset_first` 指向 `hash_set` 容器的指针。
`phset_second` 指向 `hash_set` 容器的指针。

返回值:

无。

注意:

如果 `phset_first == NULL` 或者 `phset_second == NULL` 则函数的行为是未定义的，两个 `hash_set` 必须已经初始化的，否则函数的行为是未定义的。两个 `hash_set` 容器中保存的数据类型不同函数的行为是未定义的。如果 `hash_set_equal(phset_first, phset_second)`，函数不执行任何动作。

```
hash_set_iterator_t hash_set_insert(hash_set_t* phset_set, elem);
```

描述:

向 `hash_set` 容器中插入唯一的数据。

参数:

`phset_set` 指向 `hash_set` 容器的指针。
`elem` 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 `phset_set == NULL` 则函数的行为是未定义的，`hash_set` 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 `hash_set` 容器中的数据类型相同，否则函数的行为是未定义的。

```
void hash_set_insert_range(  
    hash_set_t* phset_set,  
    hash_set_iterator_t it_begin, _hash_set_iterator_t it_end);
```

描述:

向 `hash_set` 中插入指定的数据区间，数据必须唯一。

参数:

`phset_set` 指向 `hash_set` 容器的指针。
`it_begin` 指定的数据区间的开头。
`it_end` 指定的数据区间的末尾。

返回值:

无。

注意:

如果 `phset_set == NULL` 则函数的行为是未定义的，`hash_set` 容器必须已经初始化的，否则函数的行为是未定义的。

[it_begin, it_end) 必须是有效的数据区间并且与保存在 hash_set 容器中的数据类型相同，否则函数的行为是未定义的。
[it_begin, it_end) 必须不属于 phset_set，否则函数的行为是未定义的。

```
void hash_set_erase_pos(hash_set_t* phset_set, hash_set_iterator_t it_pos);
```

描述：

删除 hash_set 容器中指定位置的数据。

参数：

phset_set 指向 hash_set 容器的指针。
it_pos 被删除的数据的位置。

返回值：

无。

注意：

如果 phset_set == NULL 则函数的行为是未定义的，hash_set 容器必须已经初始化的，否则函数的行为是未定义的。
it_pos 是属于 phset_set 容器的有效的迭代器，否则函数的行为是未定义的。

```
void hash_set_erase_range(  
    hash_set_t* phset_set,  
    hash_set_iterator_t it_begin, hash_set_iterator_t it_end);
```

描述：

删除 hash_set 中指定数据区间的数据。

参数：

phset_set 指向 hash_set 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值：

无。

注意：

如果 phset_set == NULL 则函数的行为是未定义的，hash_set 容器必须已经初始化的，否则函数的行为是未定义的。
[it_begin, it_end) 必须属于 phset_set，否则函数的行为是未定义的。

```
void hash_set_erase(hash_set_t* phset_set, elem);
```

描述：

删除 hash_set 中指定的数据。

参数：

phset_set 指向 hash_set 容器的指针。
elem 要删除的指定数据。

返回值：

被删除的数据的个数。

注意：

如果 phset_set == NULL 则函数的行为是未定义的，hash_set 容器必须已经初始化的，否则函数的行为是未定义的。
指定的数据必须与保存在 hash_set 容器中的数据类型相同，否则函数的行为是未定义的。

`hash_multiset` 提供的外部接口提供给用户使用。

<code>create_hash_multiset</code>	创建 <code>hash_multiset</code> 容器。
<code>hash_multiset_init</code>	初始化 <code>hash_multiset</code> 容器。
<code>hash_multiset_init_ex</code>	使用自定义比较规则初始化 <code>hash_multiset</code> 容器。
<code>hash_multiset_init_copy</code>	使用存在的 <code>hash_multiset</code> 容器进行初始化。
<code>hash_multiset_init_copy_range</code>	使用指定的数据区间进行初始化。
<code>hash_multiset_init_copy_range_ex</code>	使用指定的数据区间和比较函数进行初始化。
<code>hash_multiset_destroy</code>	销毁 <code>hash_multiset</code> 容器。
<code>hash_multiset_assign</code>	使用存在的 <code>hash_multiset</code> 容器赋值。
<code>hash_multiset_size</code>	返回 <code>hash_multiset</code> 容器中数据的数量。
<code>hash_multiset_empty</code>	判断 <code>hash_multiset</code> 容器是否为空。
<code>hash_multiset_max_size</code>	返回 <code>hash_multiset</code> 容器中能够保存数据的最大数量。
<code>hash_multiset_begin</code>	返回指向 <code>hash_multiset</code> 容器的第一个数据的迭代器。
<code>hash_multiset_end</code>	返回指向 <code>hash_multiset</code> 容器末尾的迭代器。
<code>hash_multiset_key_comp</code>	返回 <code>hash_multiset</code> 容器中的键比较规则。
<code>hash_multiset_value_comp</code>	返回 <code>hash_multiset</code> 容器中的数据比较规则。
<code>hash_multiset_find</code>	在 <code>hash_multiset</code> 容器中查找指定的数据。
<code>hash_multiset_clear</code>	清空 <code>hash_multiset</code> 容器。
<code>hash_multiset_count</code>	返回 <code>hash_multiset</code> 容器中指定数据的数量。
<code>hash_multiset_equal_range</code>	返回包含指定数据的数据区间。
<code>hash_multiset_equal</code>	测试两个 <code>hash_multiset</code> 容器是否相等。
<code>hash_multiset_not_equal</code>	测试两个 <code>hash_multiset</code> 容器是否不等。
<code>hash_multiset_less</code>	测试第一个 <code>hash_multiset</code> 容器是否小于第二个 <code>hash_multiset</code> 容器。
<code>hash_multiset_less_equal</code>	测试第一个 <code>hash_multiset</code> 容器是否小于等于第二个 <code>hash_multiset</code> 容器。
<code>hash_multiset_greater</code>	测试第一个 <code>hash_multiset</code> 容器是否大于第二个 <code>hash_multiset</code> 容器。
<code>hash_multiset_greater_equal</code>	测试第一个 <code>hash_multiset</code> 容器是否大于等于第二个 <code>hash_multiset</code> 容器。
<code>hash_multiset_swap</code>	交换两个 <code>hash_multiset</code> 容器。
<code>hash_multiset_insert</code>	向 <code>hash_multiset</code> 中插入数据，数据必须唯一。
<code>hash_multiset_insert_range</code>	向 <code>hash_multiset</code> 中插入指定的数据区间，数据必须唯一。
<code>hash_multiset_erase_pos</code>	删除 <code>hash_multiset</code> 中指定位置的数据。
<code>hash_multiset_erase_range</code>	删除 <code>hash_multiset</code> 中指定数据区间的数据。
<code>hash_multiset_erase</code>	删除 <code>hash_multiset</code> 中指定的数据。

函数原型

```
hash_multiset_t* create_hash_multiset(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

typename 类型描述。

返回值:

成功返回指向 hash_multiset 容器的指针, 否则返回 NULL。

注意:

关于类型描述参考第四章, 创建失败返回 NULL。

```
void hash_multiset_init(hash_multiset_t* phmset_set);
```

描述:

初始化 hash_multiset 迭代器。

参数:

phmset_set hash_multiset 容器。

返回值:

无。

注意:

phmset_set == NULL 则函数的行为是未定义的, phmset_set 必须是使用 create_hash_multiset() 创建的, 否则函数的行为是未定义的, 这个函数使用类型默认的比较函数。

```
void hash_multiset_init_copy(
    hash_multiset_t* phmset_dest, const hash_multiset_t* cphmset_src);
```

描述:

使用已经存在的 hash_multiset 初始化 hash_multiset 迭代器。

参数:

phmset_dest 目的 hash_multiset 容器。

cphash_multisett_src 源 hash_multiset 容器。

返回值:

无。

注意:

phmset_dest == NULL 或者 cphash_multisett_src == NULL 则函数的行为是未定义的, phmset_dest 必须是使用 create_hash_multiset() 创建的, cphmset_src 必须是已经初始化, 否则函数的行为是未定义的。phmset_dest 和 cphmset_src 保存的数据类型必须相同, 否则函数的行为是未定义。

```
void hash_multiset_init_copy_range(
    hash_multiset_t* phmset_dest,
    hash_multiset_iterator_t it_begin, hash_multiset_iterator_t it_end);
```

描述:

使用指定的数据区间初始化 rb tree 迭代器。

参数:

phmset_dest 目的 hash_multiset 容器。
it_begin 数据区间的开始。
it_end 数据区间的末尾。

返回值:

无。

注意:

如果 phmset_dest == NULL 则函数的行为是未定义的, phmset_dest 必须是使用 create_hash_multiset() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 hash_multiset 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void hash_multiset_init_copy_range_ex(  
    hash_multiset_t* pt_dest, hash_multiset_iterator_t it_begin,  
    hash_multiset_iterator_t it_end, binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 hash_multiset 迭代器。

参数:

phmset_dest 目的 hash_multiset 容器。
it_begin 数据区间的开始。
it_end 数据区间的末尾。
t_compare 比较函数。

返回值:

无。

注意:

如果 phmset_dest == NULL 则函数的行为是未定义的, phmset_dest 必须是使用 create_hash_multiset() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 hash_multiset 容器的有效数据区间, 否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 t_compare == NULL 则使用类型默认的比较函数。

```
void hash_multiset_destroy(hash_multiset_t* phmset_set);
```

描述:

销毁创建的 hash_multiset 容器。

参数:

phmset_set hash_multiset 容器。

返回值:

无。

注意:

如果 phmset_set == NULL 则函数的行为是未定义的, phmset_set 必须是初始化或者是使用 create_hash_multiset() 创建的, 否则函数的行为是未定义的。

```
void hash_multiset_assign(  
    hash_multiset_t* phmset_dest, const hash_multiset_t* cphmset_src);
```

描述:

使用一个 hash_multiset 为另一个 hash_multiset 赋值。

参数:

phmset_dest 指向被赋值的 hash_multiset 容器的指针。
cphmset_src 指向赋值的 hash_multiset 容器的指针。

返回值:

无。

注意:

如果 phmset_dest == NULL 或者 cphmset_src == NULL 则函数的行为是未定义的，两个 hash_multiset 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 hash_multiset_equal(phmset_dest, cphmset_src) 那么函数不做任何动作。

```
size_t hash_multiset_size(const hash_multiset_t* cphmset_set);
```

描述:

获得 hash_multiset 容器中保存的数据的个数。

参数:

cphmset_set hash_multiset 容器。

返回值:

hash_multiset 容器中保存的数据的个数。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的，cphmset_set 必须是已经初始化的 hash_multiset 容器，否则函数的行为是未定义的。

```
bool_t hash_multiset_empty(const hash_multiset_t* cphmset_set);
```

描述:

测试 hash_multiset 容器是否为空。

参数:

cphmset_set hash_multiset 容器。

返回值:

如果 hash_multiset 容器为空则返回 true，否则返回 false。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的，cphmset_set 必须是已经初始化的 hash_multiset 容器，否则函数的行为是未定义的。

```
size_t hash_multiset_max_size(const hash_multiset_t* cphmset_set);
```

描述:

获得 hash_multiset 容器中数据的最大数量。

参数:

cphmset_set hash_multiset 容器。

返回值:

hash_multiset 容器中保存的数据的个数的最大值。

注意:

如果 `cphmset_set == NULL` 则函数的行为是未定义的，`cphmset_set` 必须是已经初始化的 `hash_multiset` 容器，否则函数的行为是未定义的。

```
hash_multiset_iterator_t hash_multiset_begin(const hash_multiset_t* cphmset_set);
```

描述:

获得引用 `hash_multiset` 容器中第一数据的迭代器。

参数:

`cphmset_set` 指向 `hash_multiset` 容器的指针。

返回值:

返回引用 `hash_multiset` 容器中第一个数据的迭代器。

注意:

如果 `cphmset_set == NULL` 则函数的行为是未定义的，`hash_multiset` 容器必须已经初始化的，否则函数的行为是未定义的。如果 `hash_multiset` 容器为空，则返回值与 `hash_multiset_end(cphmset_set)` 相等。

```
hash_multiset_iterator_t hash_multiset_end(const hash_multiset_t* cphmset_set);
```

描述:

获得引用 `hash_multiset` 容器末尾的迭代器。

参数:

`cphmset_set` 指向 `hash_multiset` 容器的指针。

返回值:

返回引用 `hash_multiset` 容器末尾的迭代器。

注意:

如果 `cphmset_set == NULL` 则函数的行为是未定义的，`hash_multiset` 容器必须已经初始化的，否则函数的行为是未定义的。

```
binary_function_t hash_multiset_key_comp(const hash_multiset_t* cphmset_set);
```

描述:

返回 `hash_multiset` 容器中的键比较规则。

参数:

`cphmset_set` 指向 `hash_multiset` 容器的指针。

返回值:

返回 `hash_multiset` 容器中的数据比较规则。

注意:

如果 `cphmset_set == NULL` 则函数的行为是未定义的，`hash_multiset` 容器必须已经初始化的，否则函数的行为是未定义的。

```
binary_function_t hash_multiset_value_comp(const hash_multiset_t* cphmset_set);
```

描述:

返回 `hash_multiset` 容器中的数据比较规则。

参数:

`cphmset_set` 指向 `hash_multiset` 容器的指针。

返回值:

返回 hash_multiset 容器中的数据比较规则。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。

```
hash_multiset_iterator_t hash_multiset_find(  
    const hash_multiset_t* cphmset_set, elem);
```

描述:

在 hash_multiset 容器中查找指定的数据。

参数:

cphmset_set 指向 hash_multiset 容器的指针。
elem 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void hash_multiset_clear(hash_multiset_t* phmset_set);
```

描述:

删除 hash_multiset 容器中的所有数据。

参数:

phmset_set 指向 hash_multiset 容器的指针。

返回值:

无。

注意:

如果 phmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。

```
size_t hash_multiset_count(const hash_multiset_t* cphmset_set, elem);
```

描述:

返回 hash_multiset 容器中指定数据的数量。

参数:

cphmset_set 指向 hash_multiset 容器的指针。
elem 要查找的指定数据。

返回值:

返回 hash_multiset 容器中指定数据的数量。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t hash_multiset_equal_range(const hash_multiset_t* cphmset_set, elem);
```

描述:

返回包含指定数据的数据区间。

参数:

cphmset_set 指向 hash_multiset 容器的指针。
elem 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_hash_multiset == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
bool_t hash_multiset_equal(  
    const hash_multiset_t* cphmset_first, const hash_multiset_t* cphmset_second);
```

描述:

测试两个 hash_multiset 容器是否相等。

参数:

cphmset_first 指向 hash_multiset 容器的指针。
cphmset_second 指向 hash_multiset 容器的指针。

返回值:

如果两个 hash_multiset 容器相等则返回 true, 否则返回 false。

注意:

如果 cphmset_first == NULL 或者 cphmset_second == NULL 则函数的行为是未定义的, 两个 hash_multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同认为这两个容器不等。两个 hash_multiset 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cphmset_first == cphmset_second 那么返回 true。

```
bool_t hash_multiset_not_equal(  
    const hash_multiset_t* cphmset_first, const hash_multiset_t* cphmset_second);
```

描述:

测试两个 hash_multiset 容器是否不等。

参数:

cphmset_first 指向 hash_multiset 容器的指针。
cphmset_second 指向 hash_multiset 容器的指针。

返回值:

如果两个 hash_multiset 容器不等则返回 true, 否则返回 false。

注意:

如果 cphmset_first == NULL 或者 cphmset_second == NULL 则函数的行为是未定义的, 两个 hash_multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同认为两个 hash_multiset 容器不等。两个 hash_multiset 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cphmset_first == cphmset_second 那么返回 false。

```
bool_t hash_multiset_less(
```

```
const hash_multiset_t* cphmset_less_equal(
    const hash_multiset_t* cphmset_first, const hash_multiset_t* cphmset_second);
```

描述:

测试第一个 hash_multiset 是否小于第二个 hash_multiset。

参数:

cphmset_first 指向 hash_multiset 容器的指针。
cphmset_second 指向 hash_multiset 容器的指针。

返回值:

如果第一个 hash_multiset 容器小于第二个 hash_multiset 容器则返回 true, 否则返回 false。

注意:

如果 cphmset_first == NULL 或者 cphmset_second == NULL 则函数的行为是未定义的, 两个 hash_multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hash_multiset 中的数据小于第二个 hash_multiset 中对应的数据则返回 true, 如果大于则返回 false, 当两个 hash_multiset 中的对应数据相等, 第一个 hash_multiset 中的数据数量小于第二个 hash_multiset 中数据的数量返回 true, 否则返回 false。如果 cphmset_first == cphmset_second 那么返回 false。

```
bool_t hash_multiset_less_equal(
    const hash_multiset_t* cphmset_first, const hash_multiset_t* cphmset_second);
```

描述:

测试第一个 hash_multiset 是否小于等于第二个 hash_multiset。

参数:

cphmset_first 指向 hash_multiset 容器的指针。
cphmset_second 指向 hash_multiset 容器的指针。

返回值:

如果第一个 hash_multiset 容器小于等于第二个 hash_multiset 容器则返回 true, 否则返回 false。

注意:

如果 cphmset_first == NULL 或者 cphmset_second == NULL 则函数的行为是未定义的, 两个 hash_multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 cphmset_first == cphmset_second 那么返回 true.

```
bool_t hash_multiset_greater(
    const hash_multiset_t* cphmset_first, const hash_multiset_t* cphmset_second);
```

描述:

测试第一个 hash_multiset 是否大于第二个 hash_multiset。

参数:

cphmset_first 指向 hash_multiset 容器的指针。
cphmset_second 指向 hash_multiset 容器的指针。

返回值:

如果第一个 hash_multiset 容器大于第二个 hash_multiset 容器则返回 true, 否则返回 false。

注意:

如果 cphmset_first == NULL 或者 cphmset_second == NULL 则函数的行为是未定义的, 两个 hash_multiset 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hash_multiset 中的数据大于第二个 hash_multiset 中的对应的数据则返回 true, 小于则返回 false, 当对应的数据相

等的时候，如果第一个 hash_multiset 中数据的数量大于第二个 hash_multiset 中数据的数量的时候返回 true 否则返回 false。如果 cphmset_first == cphmset_second 那么返回 false。

```
bool_t hash_multiset_greater_equal(
    const hash_multiset_t* cphmset_first, const hash_multiset_t* cphmset_second);
```

描述:

测试第一个 hash_multiset 是否大于等于第二个 hash_multiset。

参数:

cphmset_first 指向 hash_multiset 容器的指针。
cphmset_second 指向 hash_multiset 容器的指针。

返回值:

如果第一个 hash_multiset 容器大于等于第二个 hash_multiset 容器则返回 true，否则返回 false。

注意:

如果 cphmset_first == NULL 或者 cphmset_second == NULL 则函数的行为是未定义的，两个 hash_multiset 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 cphmset_first == cphmset_second 那么返回 true。

```
void hash_multiset_swap(
    hash_multiset_t* phmset_first, hash_multiset_t* phmset_second);
```

描述:

交换两个 hash_multiset 容器。

参数:

phmset_first 指向 hash_multiset 容器的指针。
phmset_second 指向 hash_multiset 容器的指针。

返回值:

无。

注意:

如果 phmset_first == NULL 或者 phmset_second == NULL 则函数的行为是未定义的，两个 hash_multiset 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multiset 容器中保存的数据类型不同函数的行为是未定义的。如果 hash_multiset_equal(phmset_first, phmset_second)，函数不执行任何动作。

```
hash_multiset_iterator_t hash_multiset_insert(hash_multiset_t* phmset_set, elem);
```

描述:

向 hash_multiset 容器中插入唯一的数据。

参数:

phmset_set 指向 hash_multiset 容器的指针。
elem 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 phmset_set == NULL 则函数的行为是未定义的，hash_multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 hash_multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
void hash_multiset_insert_range(
    hash_multiset_t* phmset_set,
    hash_multiset_iterator_t it_begin, _hash_multiset_iterator_t it_end);
```

描述:

向 hash_multiset 中插入指定的数据区间，数据必须唯一。

参数:

phmset_set 指向 hash_multiset 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 phmset_set == NULL 则函数的行为是未定义的，hash_multiset 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end) 必须是有效的数据区间并且与保存在 hash_multiset 容器中的数据类型相同，否则函数的行为是未定义的。[it_begin, it_end) 必须不属于 phmset_set，否则函数的行为是未定义的。

```
void hash_multiset_erase_pos(
    hash_multiset_t* phmset_set, hash_multiset_iterator_t it_pos);
```

描述:

删除 hash_multiset 容器中指定位置的数据。

参数:

phmset_set 指向 hash_multiset 容器的指针。
it_pos 被删除的数据的位置。

返回值:

无。

注意:

如果 phmset_set == NULL 则函数的行为是未定义的，hash_multiset 容器必须已经初始化的，否则函数的行为是未定义的。it_pos 是属于 phmset_set 容器的有效的迭代器，否则函数的行为是未定义的。

```
void hash_multiset_erase_range(
    hash_multiset_t* phmset_set,
    hash_multiset_iterator_t it_begin, hash_multiset_iterator_t it_end);
```

描述:

删除 hash_multiset 中指定数据区间的数据。

参数:

phmset_set 指向 hash_multiset 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 phmset_set == NULL 则函数的行为是未定义的，hash_multiset 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end)必须属于 phmset_set，否则函数的行为是未定义的。

```
void hash_multiset_erase(hash_multiset_t* phmset_set, elem);
```

描述：

删除 hash_multiset 中指定的数据。

参数：

phmset_set 指向 hash_multiset 容器的指针。

elem 要删除的指定数据。

返回值：

被删除的数据的个数。

注意：

如果 phmset_set == NULL 则函数的行为是未定义的，hash_multiset 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 hash_multiset 容器中的数据类型相同，否则函数的行为是未定义的。

第五节 迭代器接口

hash_set 的迭代器是双向的迭代器，所以它没有随机访问迭代器那么强大的功能，向前和向后移动只能单步运行。这些接口只能由迭代器接口使用，用户不应该直接使用这些接口函数。

create_hash_set_iterator	创建 hash_set 迭代器。
_hash_set_iterator_get_value	获得 hash_set 迭代器引用的数据。
_hash_set_iterator_get_pointer	获得 hash_set 迭代器引用的数据的指针。
_hash_set_iterator_next	获得引用下一个数据的迭代器。
_hash_set_iterator_prev	获得引用前一个数据的迭代器。
_hash_set_iterator_equal	测试两个迭代器是否相等。
_hash_set_iterator_distance	计算两个迭代器之间的距离。
_hash_set_iterator_before	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
hash_set_iterator_t create_hash_set_iterator(void);
```

描述：

创建一个 hash_set 迭代器。

参数：

无。

返回值：

hash_set 迭代器。

注意：

返回的 hash_set 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _hash_set_iterator_get_value(hash_set_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter hash_set 迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意:

it_iter 是有效的 hash_set 迭代器，否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的，pv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _hash_set_iterator_get_pointer(hash_set_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter hash_set 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 hash_set 迭代器，否则函数的行为是未定义的。

```
hash_set_iterator_t _hash_set_iterator_next(hash_set_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter hash_set 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 hash_set 迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 hash_set 有效的迭代器，否则函数的行为是未定义的。

```
hash_set_iterator_t _hash_set_iterator_prev(hash_set_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter hash_set 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 hash_set 迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 hash_set 有效的迭代器, 否则函数的行为是未定义的。

```
bool_t _hash_set_iterator_equal(
    hash_set_iterator_t it_first, hash_set_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first hash_set 迭代器。
it_second hash_set 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 hash_set 容器的有效的 hash_set 迭代器, 否则函数的行为是未定义的。两个 hash_set 迭代器相等是指两个迭代器引用相同的数据。

```
int _hash_set_iterator_distance(
    hash_set_iterator_t it_first, hash_set_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_iterfirst hash_set 迭代器。
it_itersecond hash_set 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 hash_set 容器的有效的 hash_set 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为 0。

```
bool_t _hash_set_iterator_before(
    hash_set_iterator_t it_first, hash_set_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first hash_set 迭代器。
it_second hash_set 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 hash_set 容器的有效的 hash_set 迭代器，否则函数的行为是未定义的。

hash_multiset 的迭代器是双向的迭代器，所以它没有随机访问迭代器那么强大的功能，向前和向后移动只能单步运行。这些接口只能由迭代器接口使用，用户不应该直接使用这些接口函数。

create_hash_multiset_iterator	创建 hash_multiset 迭代器。
_hash_multiset_iterator_get_value	获得 hash_multiset 迭代器引用的数据。
_hash_multiset_iterator_get_pointer	获得 hash_multiset 迭代器引用的数据的指针。
_hash_multiset_iterator_next	获得引用下一个数据的迭代器。
_hash_multiset_iterator_prev	获得引用前一个数据的迭代器。
_hash_multiset_iterator_equal	测试两个迭代器是否相等。
_hash_multiset_iterator_distance	计算两个迭代器之间的距离。
_hash_multiset_iterator_before	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
hash_multiset_iterator_t create_hash_multiset_iterator(void);
```

描述：

创建一个 hash_multiset 迭代器。

参数：

无。

返回值：

hash_multiset 迭代器。

注意：

返回的 hash_multiset 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _hash_multiset_iterator_get_value(
    hash_multiset_iterator_t it_iter, void* pv_value);
```

描述：

获得迭代器引用的数据。

参数：

it_iter hash_multiset 迭代器。

pv_value 保存数据的缓冲区。

返回值：

无。

注意：

it_iter 是有效的 hash_multiset 迭代器，否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的，pv_value 是能够保存下 it_iter 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _hash_multiset_iterator_get_pointer(hash_multiset_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter hash_multiset 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 hash_multiset 迭代器，否则函数的行为是未定义的。

```
hash_multiset_iterator_t _hash_multiset_iterator_next(
    hash_multiset_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter hash_multiset 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 hash_multiset 迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 hash_multiset 有效的迭代器，否则函数的行为是未定义的。

```
hash_multiset_iterator_t _hash_multiset_iterator_prev(
    hash_multiset_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter hash_multiset 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 hash_multiset 迭代器，否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 hash_multiset 有效的迭代器，否则函数的行为是未定义的。

```
bool_t _hash_multiset_iterator_equal(
    hash_multiset_iterator_t it_first, hash_multiset_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first hash_multiset 迭代器。

it_second hash_multiset 迭代器。

返回值:

如果两个迭代器相等则返回 true，否则返回 false。

注意：

两个 iter 是属于同一个 hash_multiset 容器的有效的 hash_multiset 迭代器，否则函数的行为是未定义的。两个 hash_multiset 迭代器相等是指两个迭代器引用相同的数据。

```
int _hash_multiset_iterator_distance(
    hash_multiset_iterator_t it_first, hash_multiset_iterator_t it_second);
```

描述：

计算两个迭代器之间的距离。

参数：

it_iterfirst hash_multiset 迭代器。
it_itersecond hash_multiset 迭代器。

返回值：

返回两个迭代器之间的距离。

注意：

两个 iter 是属于同一个 hash_multiset 容器的有效的 hash_multiset 迭代器，否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0，it_first 引用的数据在 it_second 后面那么结果<0，如果两个迭代器相等那么结果为 0。

```
bool_t _hash_multiset_iterator_before(
    hash_multiset_iterator_t it_first, hash_multiset_iterator_t it_second);
```

描述：

测试第一个迭代器是否在第二个迭代器的前面。

参数：

it_first hash_multiset 迭代器。
it_second hash_multiset 迭代器。

返回值：

如果第一个迭代器在第二个迭代器的前面则返回 true，否则返回 false。

注意：

两个 iter 是属于同一个 hash_multiset 容器的有效的 hash_multiset 迭代器，否则函数的行为是未定义的。

第六节 内部和辅助接口

hash_set 提供的内部接口是为了给外部接口使用。

_create_hash_set	创建 hash_set 容器。
_create_hash_set_auxiliary	创建 hash_set 容器的辅助函数。
_hash_set_destroy_auxiliary	销毁 hash_set 容器的辅助函数。
_hash_set_find	在 hash_set 容器中查找指定的数据。

<code>_hash_set_find_varg</code>	在 hash_set 容器中查找指定的数据，数据来自于可变参数列表。
<code>_hash_set_count</code>	返回 hash_set 容器中指定数据的数量。
<code>_hash_set_count_varg</code>	返回 hash_set 容器中指定数据的数量，数据来自于可变参数列表。
<code>_hash_set_equal_range</code>	返回包含指定数据的数据区间。
<code>_hash_set_equal_range_varg</code>	返回包含指定数据的数据区间，数据来自于可变参数列表。
<code>_hash_set_erase</code>	删除 hash_set 中指定的数据。
<code>_hash_set_erase_varg</code>	删除 hash_set 中指定的数据，数据来自于可变参数列表。
<code>_hash_set_insert</code>	向 hash_set 中插入数据，数据必须唯一。
<code>_hash_set_insert_varg</code>	向 hash_set 中插入数据，数据必须唯一，数据来自于可变参数列表。
<code>_hash_set_init_elem_auxiliary</code>	初始化数据的辅助函数。

函数原型

```
hash_set_t* _create_hash_set(const char* s_typename);
```

描述:

创建一个 hash_set 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 hash_set 容器的指针，否则返回 NULL。

注意:

`s_typename != NULL`，否则函数的行为是未定义的。

```
bool_t _create_hash_set_auxiliary(hash_set_t* phset_set, const char* s_typename);
```

描述:

创建一个 hash_set 容器的辅助函数。

参数:

`phset_set` 没有创建的 hash_set 容器。

`s_typename` hash_set 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `phset_set == NULL` 或者 `s_typename == NULL` 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
void _hash_set_destroy_auxiliary(hash_set_t* phset_set);
```

描述:

销毁 hash_set 容器的辅助函数。

参数:

`phset_set` hash_set 容器。

返回值:

无。

注意:

如果 phset_set == NULL 或者 hash_set 不是使用_create_hash_set 生成的则函数的行为是未定义的。

```
hash_set_iterator_t _hash_set_find(const hash_set_t* cphset_set, ...);
```

描述:

在 hash_set 容器中查找指定的数据。

参数:

cphset_set 指向 hash_set 容器的指针。

... 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的，hash_set 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_set 容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_set_iterator_t _hash_set_find_varg(
    const hash_set_t* cphset_set, va_list val_elemlist);
```

描述:

在 hash_set 容器中查找指定的数据，数据来自于参数列表。

参数:

cphset_set 指向 hash_set 容器的指针。

val_elemlist 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的，hash_set 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_set 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _hash_set_count(const hash_set_t* cphset_set, ...);
```

描述:

返回 hash_set 容器中指定数据的数量。

参数:

cphset_set 指向 hash_set 容器的指针。

... 要查找的指定数据。

返回值:

返回 hash_set 容器中指定数据的数量。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的，hash_set 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_set 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _hash_set_count_varg(const hash_set_t* cphset_set, va_list val_elemlist);
```

描述:

返回 hash_set 容器中指定数据的数量，数据来自于参数列表。

参数:

cphset_set 指向 hash_set 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回 hash_set 容器中指定数据的数量。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的，hash_set 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_set 容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _hash_set_equal_range(const hash_set_t* cphset_set, ...);
```

描述:

返回包含指定数据的数据区间。

参数:

cphset_set 指向容器的指针。
... 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _hash_set_equal_range_varg(  
    const hash_set_t* cphset_set, va_list val_elemlist);
```

描述:

返回包含指定数据的数据区间。

参数:

cphset_set 指向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cphset_set == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
void _hash_set_erase(hash_set_t* phset_set, ...);
```

描述:

删除 hash_set 中指定的数据。

参数:

phset_set 指向 hash_set 容器的指针。

... 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 phset_set == NULL 则函数的行为是未定义的, hash_set 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 hash_set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _hash_set_erase_varg(hash_set_t* phset_set, va_list val_elemlist);
```

描述:

删除 hash_set 中指定的数据, 数据来自于可变参数列表。

参数:

phset_set 指向 hash_set 容器的指针。

val_elemlist 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 phset_set == NULL 则函数的行为是未定义的, hash_set 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 hash_set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_set_iterator_t _hash_set_insert(hash_set_t* phset_set, ...);
```

描述:

向 hash_set 容器中插入唯一的数据。

参数:

phset_set 指向 hash_set 容器的指针。

... 要查找的指定数据。

返回值:

指向插入的数据的迭代器, 如果插入失败则返回指向末尾的迭代器。

注意:

如果 phset_set == NULL 则函数的行为是未定义的, hash_set 容器必须已经初始化的, 否则函数的行为是未定义的。插入的数据必须与保存在 hash_set 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_set_iterator_t _hash_set_insert_varg(
    hash_set_t* phset_set, va_list val_elemlist);
```

描述:

向 hash_set 容器中插入唯一的数据, 数据来自于可变参数列表。

参数:

phset_set 指向 hash_set 容器的指针。

val_elemlist 用户指定的数据的参数列表。

返回值:

指向插入的数据的迭代器, 如果插入失败则返回指向末尾的迭代器。

注意:

如果 phset_set == NULL 则函数的行为是未定义的, hash_set 容器必须已经初始化的, 否则函数的行为是未定义的。

插入的数据必须与保存在 hash_set 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _hash_set_init_elem_auxiliary(hash_set_t* phset_set, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

phset_set hash_set 容器。
pv_value 初始化的数据。

返回值:

无。

注意:

如果 phset_set == NULL 或者 pv_value == NULL，那么函数的行为是未定义的。phset_set 必须是初始化的或者是使用 create_hash_set() 创建的，否则函数的行为是未定义的。

hash_set 提供了少量的辅助函数。

_hash_set_get_varg_value_auxiliary	根据 hash_set 容器中数据的类型获得可变参数列表中的数据。
_hash_set_destroy_varg_value_auxiliary	销毁从可变参数列表中获得的数据。

```
void _hash_set_get_varg_value_auxiliary(  
    hash_set_t* phset_set, va_list val_elemlist, void* pv_varg);
```

描述:

从可变参数列表中获得与 hash_set 中的数据类型相同的数据。

参数:

phset_set hash_set 容器。
val_elemlist 可变参数列表。
pv_varg 保存数据缓的缓冲区。

返回值:

无。

注意:

如果 phset_set == NULL 或者 pv_varg == NULL，那么函数的行为是未定义的。phset_set 必须是已经初始化或者是由 create_hash_set() 创建的 hash_set 容器，否则函数的行为是未定义的。

```
void _hash_set_destroy_varg_value_auxiliary(hash_set_t* phset_set, void* pv_varg);
```

描述:

销毁与 hash_set 中的数据类型相同的数据。

参数:

phset_set hash_set 容器。
pv_varg 保存数据的缓冲区。

返回值:

无。

注意:

如果 phset_set == NULL 或者 pv_varg == NULL，那么函数的行为是未定义的。phset_set 必须是已经初始化或者是由 create_hash_set() 创建的 hash_set 容器，否则函数的行为是未定义的。

由 `create_hash_set()` 创建的 `hash_set` 容器，否则函数的行为是未定义的。

`hash_multiset` 提供的内部接口是为了给外部接口使用。

<code>_create_hash_multiset</code>	创建 <code>hash_multiset</code> 容器。
<code>_create_hash_multiset_auxiliary</code>	创建 <code>hash_multiset</code> 容器的辅助函数。
<code>_hash_multiset_destroy_auxiliary</code>	销毁 <code>hash_multiset</code> 容器的辅助函数。
<code>_hash_multiset_find</code>	在 <code>hash_multiset</code> 容器中查找指定的数据。
<code>_hash_multiset_find_varg</code>	在 <code>hash_multiset</code> 容器中查找指定的数据，数据来自于可变参数列表。
<code>_hash_multiset_count</code>	返回 <code>hash_multiset</code> 容器中指定数据的数量。
<code>_hash_multiset_count_varg</code>	返回 <code>hash_multiset</code> 容器中指定数据的数量，数据来自于可变参数列表。
<code>_hash_multiset_equal_range</code>	返回包含指定数据的数据区间。
<code>_hash_multiset_equal_range_varg</code>	返回包含指定数据的数据区间，数据来自于可变参数列表。
<code>_hash_multiset_erase</code>	删除 <code>hash_multiset</code> 中指定的数据。
<code>_hash_multiset_erase_varg</code>	删除 <code>hash_multiset</code> 中指定的数据，数据来自于可变参数列表。
<code>_hash_multiset_insert</code>	向 <code>hash_multiset</code> 中插入数据，数据必须唯一。
<code>_hash_multiset_insert_varg</code>	向 <code>hash_multiset</code> 中插入数据，数据必须唯一，数据来自于可变参数列表。
<code>_hash_multiset_init_elem_auxiliary</code>	初始化数据的辅助函数。

函数原型

```
hash_multiset_t* _create_hash_multiset(const char* s_typename);
```

描述:

创建一个 `hash_multiset` 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 `hash_multiset` 容器的指针，否则返回 `NULL`。

注意:

`s_typename != NULL`，否则函数的行为是未定义的。

```
bool_t _create_hash_multiset_auxiliary(
    hash_multiset_t* phmset_set, const char* s_typename);
```

描述:

创建一个 `hash_multiset` 容器的辅助函数。

参数:

`phmset_set` 没有创建的 `hash_multiset` 容器。

`s_typename` `hash_multiset` 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 phmset_set == NULL 或者 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _hash_multiset_destroy_auxiliary(hash_multiset_t* phmset_set);
```

描述:

销毁 hash_multiset 容器的辅助函数。

参数:

phmset_set hash_multiset 容器。

返回值:

无。

注意:

如果 phmset_set == NULL 或者 hash_multiset 不是使用_create_hash_multiset 生成的则函数的行为是未定义的。

```
hash_multiset_iterator_t _hash_multiset_find(
    const hash_multiset_t* cphmset_set, ...);
```

描述:

在 hash_multiset 容器中查找指定的数据。

参数:

cphmset_set 指向 hash_multiset 容器的指针。
... 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_multiset_iterator_t _hash_multiset_find_varg(
    const hash_multiset_t* cphmset_set, va_list val_elemlist);
```

描述:

在 hash_multiset 容器中查找指定的数据, 数据来自于参数列表。

参数:

cphmset_set 指向 hash_multiset 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
size_t _hash_multiset_count(const hash_multiset_t* cphmset_set, ...);
```

描述:

返回 hash_multiset 容器中指定数据的数量。

参数:

cphmset_set 指向 hash_multiset 容器的指针。
... 要查找的指定数据。

返回值:

返回 hash_multiset 容器中指定数据的数量。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
size_t _hash_multiset_count_varg(  
    const hash_multiset_t* cphmset_set, va_list val_elemlist);
```

描述:

返回 hash_multiset 容器中指定数据的数量, 数据来自于参数列表。

参数:

cphmset_set 指向 hash_multiset 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回 hash_multiset 容器中指定数据的数量。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, hash_multiset 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multiset 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t _hash_multiset_equal_range(const hash_multiset_t* cphmset_set, ...);
```

描述:

返回包含指定数据的数据区间。

参数:

cphmset_set 指向容器的指针。
... 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cphmset_set == NULL 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t _hash_multiset_equal_range_varg(  
    const hash_multiset_t* cphmset_set, va_list val_elemlist);
```

描述:

返回包含指定数据的数据区间。

参数:

cphmset_set 指向容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回包含指定数据的数据区间。

注意:

如果 `phmset_set == NULL` 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _hash_multiset_erase(hash_multiset_t* phmset_set, ...);
```

描述:

删除 `hash_multiset` 中指定的数据。

参数:

`phmset_set` 指向 `hash_multiset` 容器的指针。

`...` 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 `phmset_set == NULL` 则函数的行为是未定义的, `hash_multiset` 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 `hash_multiset` 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _hash_multiset_erase_varg(hash_multiset_t* phmset_set, va_list val_elemlist);
```

描述:

删除 `hash_multiset` 中指定的数据, 数据来自于可变参数列表。

参数:

`phmset_set` 指向 `hash_multiset` 容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 `phmset_set == NULL` 则函数的行为是未定义的, `hash_multiset` 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 `hash_multiset` 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_multiset_iterator_t _hash_multiset_insert(hash_multiset_t* phmset_set, ...);
```

描述:

向 `hash_multiset` 容器中插入唯一的数据。

参数:

`phmset_set` 指向 `hash_multiset` 容器的指针。

`...` 要查找的指定数据。

返回值:

指向插入的数据的迭代器, 如果插入失败则返回指向末尾的迭代器。

注意:

如果 `phmset_set == NULL` 则函数的行为是未定义的, `hash_multiset` 容器必须已经初始化的, 否则函数的行为是未定义的。插入的数据必须与保存在 `hash_multiset` 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_multiset_iterator_t _hash_multiset_insert_varg(
    hash_multiset_t* phmset_set, va_list val_elemlist);
```

描述:

向 hash_multiset 容器中插入唯一的数据，数据来自于可变参数列表。

参数:

phmset_set 指向 hash_multiset 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 phmset_set == NULL 则函数的行为是未定义的，hash_multiset 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 hash_multiset 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _hash_multiset_init_elem_auxiliary(
    hash_multiset_t* phmset_set, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

phmset_set hash_multiset 容器。
pv_value 初始化的数据。

返回值:

无。

注意:

如果 phmset_set == NULL 或者 pv_value == NULL，那么函数的行为是未定义的。phmset_set 必须是初始化的或者是使用 create_hash_multiset() 创建的，否则函数的行为是未定义的。

hash_multiset 提供了少量的辅助函数。

_hash_multiset_get_varg_value_auxiliary	根据 hash_multiset 容器中数据的类型获得可变参数列表中的数据。
_hash_multiset_destroy_varg_value_auxiliary	销毁从可变参数列表中获得的数据。

```
void _hash_multiset_get_varg_value_auxiliary(
    hash_multiset_t* phmset_set, va_list val_elemlist, void* pv_varg);
```

描述:

从可变参数列表中获得与 hash_multiset 中的数据类型相同的数据。

参数:

phmset_set hash_multiset 容器。
val_elemlist 可变参数列表。
pv_varg 保存数据缓的冲区。

返回值:

无。

注意:

如果 phmset_set == NULL 或者 pv_varg == NULL, 那么函数的行为是未定义的。phmset_set 必须是已经初始化或者是由 create_hash_multiset() 创建的 hash_multiset 容器, 否则函数的行为是未定义的。

```
void _hash_multiset_destroy_value_auxiliary(
    hash_multiset_t* phmset_set, void* pv_varg);
```

描述:

销毁与 hash_multiset 中的数据类型相同的数据。

参数:

phmset_set hash_multiset 容器。

pv_varg 保存数据的缓冲区。

返回值:

无。

注意:

如果 phmset_set == NULL 或者 pv_varg == NULL, 那么函数的行为是未定义的。phmset_set 必须是已经初始化或者是由 create_hash_multiset() 创建的 hash_multiset 容器, 否则函数的行为是未定义的。

第二十一章 hash_map 和 hash_multimap

第一节 hash_map 和 hash_multimap 的内部机制

hash_map 和 hash_multimap 是对于映射的抽象，它们保存的数据类型都是 key/value 的 pair_t 类型，在 hash_map 中 key 必须唯一，hash_multimap 中 key 可以不唯一，对 value 没有唯一性的要求。hash_map 和 hash_multimap 都是使用 hashtable 实现的。

第二节 hash_map 和 hash_multimap 的迭代器

hash_map 和 hash_multimap 的迭代器使用 hashtable 的迭代器实现。

第三节 hash_map 和 hash_multimap 的代码结构

hash_map 的代码结构如下

```
typedef struct _taghashmap
{
    pair_t          _t_pair;
    binary_function_t _t_keycompare; /* for init ex */
    binary_function_t _t_valuecompare;
    _hashtable_t      _t_hashtable;
}hash_map_t;
```

hash_multimap 的代码结构如下

```
typedef struct _taghashmultimap
{
    pair_t          _t_pair;
    binary_function_t _t_keycompare; /* for init ex */
    binary_function_t _t_valuecompare;
    _hashtable_t      _t_hashtable;
}hash_multimap_t;
```

可以通过配置来决定使用 hashtable 作为底层实现。

第四节 外部接口

hash_map 提供的外部接口提供给用户使用。

create_hash_map	创建 hash_map 容器。
hash_map_init	初始化 hash_map 容器。
hash_map_init_ex	使用自定义比较规则初始化 hash_map 容器。
hash_map_init_copy	使用存在的 hash_map 容器进行初始化。
hash_map_init_copy_range	使用指定的数据区间进行初始化。
hash_map_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
hash_map_destroy	销毁 hash_map 容器。
hash_map_assign	使用存在的 hash_map 容器赋值。
hash_map_size	返回 hash_map 容器中数据的数量。
hash_map_empty	判断 hash_map 容器是否为空。
hash_map_max_size	返回 hash_map 容器中能够保存数据的最大数量。
hash_map_begin	返回指向 hash_map 容器的第一个数据的迭代器。
hash_map_end	返回指向 hash_map 容器末尾的迭代器。
hash_map_key_comp	返回 hash_map 容器中的键比较规则。
hash_map_value_comp	返回 hash_map 容器中的数据比较规则。
hash_map_find	在 hash_map 容器中查找指定的数据。
hash_map_clear	清空 hash_map 容器。
hash_map_count	返回 hash_map 容器中指定数据的数量。
hash_map_lower_bound	返回指向第一个大于等于指定数据的迭代器。
hash_map_upper_bound	返回指向第一个大于指定数据的迭代器。
hash_map_equal_range	返回包含指定数据的数据区间。
hash_map_equal	测试两个 hash_map 容器是否相等。
hash_map_not_equal	测试两个 hash_map 容器是否不等。
hash_map_less	测试第一个 hash_map 容器是否小于第二个 hash_map 容器。
hash_map_less_equal	测试第一个 hash_map 容器是否小于等于第二个 hash_map 容器。
hash_map_greater	测试第一个 hash_map 容器是否大于第二个 hash_map 容器。
hash_map_greater_equal	测试第一个 hash_map 容器是否大于等于第二个 hash_map 容器。
hash_map_swap	交换两个 hash_map 容器。

hash_map_insert	向 hash_map 中插入数据，数据必须唯一。
hash_map_insert_hint	向 hash_map 中线索位置插入数据，数据必须唯一。
hash_map_insert_range	向 hash_map 中插入指定的数据区间，数据必须唯一。
hash_map_erase_pos	删除 hash_map 中指定位置的数据。
hash_map_erase_range	删除 hash_map 中指定数据区间的数据。
hash_map_erase	删除 hash_map 中指定的数据。
hash_map_at	使用下标对 hash_map 中的数据进行访问。

函数原型

```
hash_map_t* create_hash_map(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

typename 类型描述。

返回值:

成功返回指向 hash_map 容器的指针，否则返回 NULL。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void hash_map_init(hash_map_t* phmap_map);
```

描述:

初始化 hash_map 迭代器。

参数:

phmap_map hash_map 容器。

返回值:

无。

注意:

phmap_map == NULL 则函数的行为是未定义的，phmap_map 必须是使用 create_hash_map() 创建的，否则函数的行为是未定义的，这个函数使用类型默认的比较函数。

```
void hash_map_init_copy(hash_map_t* phmap_dest, const hash_map_t* cphmap_src);
```

描述:

使用已经存在的 hash_map 初始化 hash_map 迭代器。

参数:

phmap_dest 目的 hash_map 容器。

cphash_mapt_src 源 hash_map 容器。

返回值:

无。

注意:

phmap_dest == NULL 或者 cphash_mapt_src == NULL 则函数的行为是未定义的，phmap_dest 必须是使用

create_hash_map()创建的，`cphmap_src` 必须是已经初始化，否则函数的行为是未定义的。`phmap_dest` 和 `cphmap_src` 保存的数据类型必须相同，否则函数的行为是未定义。

```
void hash_map_init_copy_range(hash_map_t* phmap_dest,
    hash_map_iterator_t it_begin, hash_map_iterator_t it_end);
```

描述:

使用指定的数据区间初始化 rb tree 迭代器。

参数:

`phmap_dest` 目的 `hash_map` 容器。

`it_begin` 数据区间的开始。

`it_end` 数据区间的末尾。

返回值:

无。

注意:

如果 `phmap_dest == NULL` 则函数的行为是未定义的，`phmap_dest` 必须是使用 `create_hash_map()` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 `hash_map` 容器的有效数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void hash_map_init_copy_range_ex(
    hash_map_t* pt_dest, hash_map_iterator_t it_begin,
    hash_map_iterator_t it_end, binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 `hash_map` 迭代器。

参数:

`phmap_dest` 目的 `hash_map` 容器。

`it_begin` 数据区间的开始。

`it_end` 数据区间的末尾。

`t_compare` 比较函数。

返回值:

无。

注意:

如果 `phmap_dest == NULL` 则函数的行为是未定义的，`phmap_dest` 必须是使用 `create_hash_map()` 创建的否则函数的行为是未定义的。`[it_begin, it_end)` 属于一个已经初始化的 `hash_map` 容器的有效数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 `t_compare == NULL` 则使用类型默认的比较函数。

```
void hash_map_destroy(hash_map_t* phmap_map);
```

描述:

销毁创建的 `hash_map` 容器。

参数:

`phmap_map` `hash_map` 容器。

返回值:

无。

注意：

如果 phmap_map == NULL 则函数的行为是未定义的，phmap_map 必须是初始化或者是使用 create_hash_map() 创建的，否则函数的行为是未定义的。

```
void hash_map_assign(hash_map_t* phmap_dest, const hash_map_t* cphmap_src);
```

描述：

使用一个 hash_map 为另一个 hash_map 赋值。

参数：

phmap_dest 指向被赋值的 hash_map 容器的指针。

cphmap_src 指向赋值的 hash_map 容器的指针。

返回值：

无。

注意：

如果 phmap_dest == NULL 或者 cphmap_src == NULL 则函数的行为是未定义的，两个 hash_map 必须已经初始化的，否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同函数的行为是未定义的。如果 hash_map_equal(phmap_dest, cphmap_src)那么函数不做任何动作。

```
size_t hash_map_size(const hash_map_t* cphmap_map);
```

描述：

获得 hash_map 容器中保存的数据的个数。

参数：

cphmap_map hash_map 容器。

返回值：

hash_map 容器中保存的数据的个数。

注意：

如果 cphmap_map == NULL 则函数的行为是未定义的，cphmap_map 必须是已经初始化的 hash_map 容器，否则函数的行为是未定义的。

```
bool_t hash_map_empty(const hash_map_t* cphmap_map);
```

描述：

测试 hash_map 容器是否为空。

参数：

cphmap_map hash_map 容器。

返回值：

如果 hash_map 容器为空则返回 true，否则返回 false。

注意：

如果 cphmap_map == NULL 则函数的行为是未定义的，cphmap_map 必须是已经初始化的 hash_map 容器，否则函数的行为是未定义的。

```
size_t hash_map_max_size(const hash_map_t* cphmap_map);
```

描述：

获得 hash_map 容器中数据的最大数量。

参数:

cphmap_map hash_map 容器。

返回值:

hash_map 容器中保存的数据的个数的最大值。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, cphmap_map 必须是已经初始化的 hash_map 容器, 否则函数的行为是未定义的。

```
hash_map_iterator_t hash_map_begin(const hash_map_t* cphmap_map);
```

描述:

获得引用 hash_map 容器中第一数据的迭代器。

参数:

cphmap_map 指向 hash_map 容器的指针。

返回值:

返回引用 hash_map 容器中第一个数据的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。如果 hash_map 容器为空, 则返回值与 hash_map_end(cphmap_map) 相等。

```
hash_map_iterator_t hash_map_end(const hash_map_t* cphmap_map);
```

描述:

获得引用 hash_map 容器末尾的迭代器。

参数:

cphmap_map 指向 hash_map 容器的指针。

返回值:

返回引用 hash_map 容器末尾的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t hash_map_key_comp(const hash_map_t* cphmap_map);
```

描述:

返回 hash_map 容器中的键比较规则。

参数:

cphmap_map 指向 hash_map 容器的指针。

返回值:

返回 hash_map 容器中的数据比较规则。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t hash_map_value_comp(const hash_map_t* cphmap_map);
```

描述:

返回 hash_map 容器中的数据比较规则。

参数:

cphmap_map 指向 hash_map 容器的指针。

返回值:

返回 hash_map 容器中的数据比较规则。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
hash_map_iterator_t hash_map_find(const hash_map_t* cphmap_map, elem);
```

描述:

在 hash_map 容器中查找指定的数据。

参数:

cphmap_map 指向 hash_map 容器的指针。

elem 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void hash_map_clear(hash_map_t* phmap_map);
```

描述:

删除 hash_map 容器中的所有数据。

参数:

phmap_map 指向 hash_map 容器的指针。

返回值:

无。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。

```
size_t hash_map_count(const hash_map_t* cphmap_map, elem);
```

描述:

返回 hash_map 容器中指定数据的数量。

参数:

cphmap_map 指向 hash_map 容器的指针。

elem 要查找的指定数据。

返回值:

返回 hash_map 容器中指定数据的数量。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_map_iterator_t hash_map_lower_bound(const hash_map_t* cphmap_map, elem);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cphmap_map 指向 hash_map 容器的指针。

elem 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_map_iterator_t hash_map_upper_bound(const hash_map_t* cphmap_map, elem);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cphmap_map 指向 hash_map 容器的指针。

elem 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
range_t hash_map_equal_range(const hash_map_t* cphmap_map, elem);
```

描述:

返回包含指定数据的数据区间。

参数:

cphmap_map 指向 hash_map 容器的指针。

elem 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_hash_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
bool_t hash_map_equal(
    const hash_map_t* cphmap_first, const hash_map_t* cphmap_second);
```

描述:

测试两个 hash_map 容器是否相等。

参数:

cphmap_first 指向 hash_map 容器的指针。
cphmap_second 指向 hash_map 容器的指针。

返回值:

如果两个 hash_map 容器相等则返回 true, 否则返回 false。

注意:

如果 cphmap_first == NULL 或者 cphmap_second == NULL 则函数的行为是未定义的, 两个 hash_map 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同认为这两个容器不等。两个 hash_map 容器相等是指容器中的数据对应相等, 并且容器中数据的数量也相等。如果 cphmap_first == cphmap_second 那么返回 true。

```
bool_t hash_map_not_equal(  
    const hash_map_t* cphmap_first, const hash_map_t* cphmap_second);
```

描述:

测试两个 hash_map 容器是否不等。

参数:

cphmap_first 指向 hash_map 容器的指针。
cphmap_second 指向 hash_map 容器的指针。

返回值:

如果两个 hash_map 容器不等则返回 true, 否则返回 false。

注意:

如果 cphmap_first == NULL 或者 cphmap_second == NULL 则函数的行为是未定义的, 两个 hash_map 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同认为两个 hash_map 容器不等。两个 hash_map 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cphmap_first == cphmap_second 那么返回 false。

```
bool_t hash_map_less(  
    const hash_map_t* cphmap_first, const hash_map_t* cphmap_second);
```

描述:

测试第一个 hash_map 是否小于第二个 hash_map。

参数:

cphmap_first 指向 hash_map 容器的指针。
cphmap_second 指向 hash_map 容器的指针。

返回值:

如果第一个 hash_map 容器小于第二个 hash_map 容器则返回 true, 否则返回 false。

注意:

如果 cphmap_first == NULL 或者 cphmap_second == NULL 则函数的行为是未定义的, 两个 hash_map 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hash_map 中的数据小于第二个 hash_map 中对应的数据则返回 true, 如果大于则返回 false, 当两个 hash_map 中的数据相等, 第一个 hash_map 中的数据数量小于第二个 hash_map 中数据的数量返回 true, 否则返回 false。如果 cphmap_first == cphmap_second 那么返回 false。

```
bool_t hash_map_less_equal(
    const hash_map_t* cphmap_first, const hash_map_t* cphmap_second);
```

描述:

测试第一个 hash_map 是否小于等于第二个 hash_map。

参数:

cphmap_first	指向 hash_map 容器的指针。
cphmap_second	指向 hash_map 容器的指针。

返回值:

如果第一个 hash_map 容器小于等于第二个 hash_map 容器则返回 true, 否则返回 false。

注意:

如果 cphmap_first == NULL 或者 cphmap_second == NULL 则函数的行为是未定义的, 两个 hash_map 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同函数的行为是未定义的。如果 cphmap_first == cphmap_second 那么返回 true。

```
bool_t hash_map_greater(
    const hash_map_t* cphmap_first, const hash_map_t* cphmap_second);
```

描述:

测试第一个 hash_map 是否大于第二个 hash_map。

参数:

cphmap_first	指向 hash_map 容器的指针。
cphmap_second	指向 hash_map 容器的指针。

返回值:

如果第一个 hash_map 容器大于第二个 hash_map 容器则返回 true, 否则返回 false。

注意:

如果 cphmap_first == NULL 或者 cphmap_second == NULL 则函数的行为是未定义的, 两个 hash_map 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hash_map 中的数据大于第二个 hash_map 中的对应的数据则返回 true, 小于则返回 false, 当对应的数据相等的时候, 如果第一个 hash_map 中数据的数量大于第二个 hash_map 中数据的数量的时候返回 true 否则返回 false。如果 cphmap_first == cphmap_second 那么返回 false。

```
bool_t hash_map_greater_equal(
    const hash_map_t* cphmap_first, const hash_map_t* cphmap_second);
```

描述:

测试第一个 hash_map 是否大于等于第二个 hash_map。

参数:

cphmap_first	指向 hash_map 容器的指针。
cphmap_second	指向 hash_map 容器的指针。

返回值:

如果第一个 hash_map 容器大于等于第二个 hash_map 容器则返回 true, 否则返回 false。

注意:

如果 cphmap_first == NULL 或者 cphmap_second == NULL 则函数的行为是未定义的, 两个 hash_map 必须已经初始化

的，否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同函数的行为是未定义的。如果 cphmap_first == cphmap_second 那么返回 true。

```
void hash_map_swap(hash_map_t* phmap_first, hash_map_t* phmap_second);
```

描述:

交换两个 hash_map 容器。

参数:

phmap_first 指向 hash_map 容器的指针。
phmap_second 指向 hash_map 容器的指针。

返回值:

无。

注意:

如果 phmap_first == NULL 或者 phmap_second == NULL 则函数的行为是未定义的，两个 hash_map 必须已经初始化的，否则函数的行为是未定义的。两个 hash_map 容器中保存的数据类型不同函数的行为是未定义的。如果 hash_map_equal(phmap_first, phmap_second)，函数不执行任何动作。

```
hash_map_iterator_t hash_map_insert(  
    hash_map_t* phmap_map, const pair_t* cppair_pair);
```

描述:

向 hash_map 容器中插入唯一的数据。

参数:

phmap_map 指向 hash_map 容器的指针。
cppair_pair 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_map_iterator_t hash_map_insert_hint(  
    hash_map_t* phmap_map, hash_map_iterator_t it_hint, const pair_t* cppair_pair);
```

描述:

向 hash_map 容器中的线索位置插入唯一的数据。

参数:

phmap_map 指向 hash_map 容器的指针。
it_hint 用户提供的线索位置。
cppair_pair 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_hash_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。

```
void hash_map_insert_range(
    hash_map_t* phmap_map,
    hash_map_iterator_t it_begin, hash_map_iterator_t it_end);
```

描述:

向 hash_map 中插入指定的数据区间，数据必须唯一。

参数:

phmap_map 指向 hash_map 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。[it_begin, it_end) 必须是有效的数据区间并且与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。[it_begin, it_end) 必须不属于 phmap_map，否则函数的行为是未定义的。

```
void hash_map_erase_pos(hash_map_t* phmap_map, hash_map_iterator_t it_pos);
```

描述:

删除 hash_map 容器中指定位置的数据。

参数:

phmap_map 指向 hash_map 容器的指针。
it_pos 被删除的数据的位置。

返回值:

无。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。it_pos 是属于 phmap_map 容器的有效的迭代器，否则函数的行为是未定义的。

```
void hash_map_erase_range(
    hash_map_t* phmap_map,
    hash_map_iterator_t it_begin, hash_map_iterator_t it_end);
```

描述:

删除 hash_map 中指定数据区间的数据。

参数:

phmap_map 指向 hash_map 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定

义的。[it_begin, it_end)必须属于 phmap_map，否则函数的行为是未定义的。

```
void hash_map_erase(hash_map_t* phmap_map, elem);
```

描述:

删除 hash_map 中指定的数据。

参数:

phmap_map 指向 hash_map 容器的指针。
elem 要删除的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。

```
void* hash_map_at(const hash_map_t* cphmap_map, elem);
```

描述:

通过下标访问 hash_map 中的数据。

参数:

phmap_map 指向容器的指针。
elem 要查找的指定数据。

返回值:

返回以指定数据为键的数据的指针。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 hash_map 中不包含以指定的数据为键的数据，则首先向 hash_map 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

hash_multimap 提供的外部接口提供给用户使用。

create_hash_multimap	创建 hash_multimap 容器。
hash_multimap_init	初始化 hash_multimap 容器。
hash_multimap_init_ex	使用自定义比较规则初始化 hash_multimap 容器。
hash_multimap_init_copy	使用存在的 hash_multimap 容器进行初始化。
hash_multimap_init_copy_range	使用指定的数据区间进行初始化。
hash_multimap_init_copy_range_ex	使用指定的数据区间和比较函数进行初始化。
hash_multimap_destroy	销毁 hash_multimap 容器。
hash_multimap_assign	使用存在的 hash_multimap 容器赋值。
hash_multimap_size	返回 hash_multimap 容器中数据的数量。
hash_multimap_empty	判断 hash_multimap 容器是否为空。

hash_multimap_max_size	返回 hash_multimap 容器中能够保存数据的最大数量。
hash_multimap_begin	返回指向 hash_multimap 容器的第一个数据的迭代器。
hash_multimap_end	返回指向 hash_multimap 容器末尾的迭代器。
hash_multimap_key_comp	返回 hash_multimap 容器中的键比较规则。
hash_multimap_value_comp	返回 hash_multimap 容器中的数据比较规则。
hash_multimap_find	在 hash_multimap 容器中查找指定的数据。
hash_multimap_clear	清空 hash_multimap 容器。
hash_multimap_count	返回 hash_multimap 容器中指定数据的数量。
hash_multimap_lower_bound	返回指向第一个大于等于指定数据的迭代器。
hash_multimap_upper_bound	返回指向第一个大于指定数据的迭代器。
hash_multimap_equal_range	返回包含指定数据的数据区间。
hash_multimap_equal	测试两个 hash_multimap 容器是否相等。
hash_multimap_not_equal	测试两个 hash_multimap 容器是否不等。
hash_multimap_less	测试第一个 hash_multimap 容器是否小于第二个 hash_multimap 容器。
hash_multimap_less_equal	测试第一个 hash_multimap 容器是否小于等于第二个 hash_multimap 容器。
hash_multimap_greater	测试第一个 hash_multimap 容器是否大于第二个 hash_multimap 容器。
hash_multimap_greater_equal	测试第一个 hash_multimap 容器是否大于等于第二个 hash_multimap 容器。
hash_multimap_swap	交换两个 hash_multimap 容器。
hash_multimap_insert	向 hash_multimap 中插入数据，数据必须唯一。
hash_multimap_insert_hint	向 hash_multimap 中线索位置插入数据，数据必须唯一。
hash_multimap_insert_range	向 hash_multimap 中插入指定的数据区间，数据必须唯一。
hash_multimap_erase_pos	删除 hash_multimap 中指定位置的数据。
hash_multimap_erase_range	删除 hash_multimap 中指定数据区间的数据。
hash_multimap_erase	删除 hash_multimap 中指定的数据。
hash_multimap_at	使用下标对 hash_multimap 中的数据进行访问。

函数原型

```
hash_multimap_t* create_hash_multimap(typename);
```

描述:

创建一个 rb tree 迭代器。

参数:

typename 类型描述。

返回值:

成功返回指向 hash_multimap 容器的指针，否则返回 NULL。

注意:

关于类型描述参考第四章，创建失败返回 NULL。

```
void hash_multimap_init(hash_multimap_t* phmmmap_map);
```

描述:

初始化 hash_multimap 迭代器。

参数:

phmmmap_map hash_multimap 容器。

返回值:

无。

注意:

phmmmap_map == NULL 则函数的行为是未定义的，phmmmap_map 必须是使用 create_hash_multimap() 创建的，否则函数的行为是未定义的，这个函数使用类型默认的比较函数。

```
void hash_multimap_init_copy(
    hash_multimap_t* phmmmap_dest, const hash_multimap_t* cphmmmap_src);
```

描述:

使用已经存在的 hash_multimap 初始化 hash_multimap 迭代器。

参数:

phmmmap_dest 目的 hash_multimap 容器。

cphash_multimap_src 源 hash_multimap 容器。

返回值:

无。

注意:

phmmmap_dest == NULL 或者 cphash_multimap_src == NULL 则函数的行为是未定义的，phmmmap_dest 必须是使用 create_hash_multimap() 创建的，cphmmmap_src 必须是已经初始化，否则函数的行为是未定义的。phmmmap_dest 和 cphmmmap_src 保存的数据类型必须相同，否则函数的行为是未定义。

```
void hash_multimap_init_copy_range(hash_multimap_t* phmmmap_dest,
    hash_multimap_iterator_t it_begin, hash_multimap_iterator_t it_end);
```

描述:

使用指定的数据区间初始化 rb tree 迭代器。

参数:

phmmmap_dest 目的 hash_multimap 容器。

it_begin 数据区间的开始。

it_end 数据区间的末尾。

返回值:

无。

注意:

如果 phmmmap_dest == NULL 则函数的行为是未定义的，phmmmap_dest 必须是使用 create_hash_multimap() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 hash_multimap 容器的有效的数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。

```
void hash_multimap_init_copy_range_ex(
    hash_multimap_t* pt_dest, hash_multimap_iterator_t it_begin,
    hash_multimap_iterator_t it_end, binary_function_t t_compare);
```

描述:

使用指定的数据区间和比较函数初始化 hash_multimap 迭代器。

参数:

phmmmap_dest 目的 hash_multimap 容器。
it_begin 数据区间的开始。
it_end 数据区间的末尾。
t_compare 比较函数。

返回值:

无。

注意:

如果 phmmmap_dest == NULL 则函数的行为是未定义的，phmmmap_dest 必须是使用 create_hash_multimap() 创建的否则函数的行为是未定义的。[it_begin, it_end) 属于一个已经初始化的 hash_multimap 容器的有效数据区间，否则函数的行为是未定义的。容器与数据区间中保存的数据类型必须是一致的否则函数的行为是未定义的。如果 t_compare == NULL 则使用类型默认的比较函数。

```
void hash_multimap_destroy(hash_multimap_t* phmmmap_map);
```

描述:

销毁创建的 hash_multimap 容器。

参数:

phmmmap_map hash_multimap 容器。

返回值:

无。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的，phmmmap_map 必须是初始化或者是使用 create_hash_multimap() 创建的，否则函数的行为是未定义的。

```
void hash_multimap_assign(
    hash_multimap_t* phmmmap_dest, const hash_multimap_t* cphmmmap_src);
```

描述:

使用一个 hash_multimap 为另一个 hash_multimap 赋值。

参数:

phmmmap_dest 指向被赋值的 hash_multimap 容器的指针。
cphmmmap_src 指向赋值的 hash_multimap 容器的指针。

返回值:

无。

注意:

如果 phmmmap_dest == NULL 或者 cphmmmap_src == NULL 则函数的行为是未定义的，两个 hash_multimap 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同函数的行为是未定义的。如

果 hash_multimap_equal(phmmmap_dest, cphmmmap_src)那么函数不做任何动作。

```
size_t hash_multimap_size(const hash_multimap_t* cphmmmap_map);
```

描述:

获得 hash_multimap 容器中保存的数据的个数。

参数:

cphmmmap_map hash_multimap 容器。

返回值:

hash_multimap 容器中保存的数据的个数。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, cphmmmap_map 必须是已经初始化的 hash_multimap 容器, 否则函数的行为是未定义的。

```
bool_t hash_multimap_empty(const hash_multimap_t* cphmmmap_map);
```

描述:

测试 hash_multimap 容器是否为空。

参数:

cphmmmap_map hash_multimap 容器。

返回值:

如果 hash_multimap 容器为空则返回 true, 否则返回 false。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, cphmmmap_map 必须是已经初始化的 hash_multimap 容器, 否则函数的行为是未定义的。

```
size_t hash_multimap_max_size(const hash_multimap_t* cphmmmap_map);
```

描述:

获得 hash_multimap 容器中数据的最大数量。

参数:

cphmmmap_map hash_multimap 容器。

返回值:

hash_multimap 容器中保存的数据的个数的最大值。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, cphmmmap_map 必须是已经初始化的 hash_multimap 容器, 否则函数的行为是未定义的。

```
hash_multimap_iterator_t hash_multimap_begin(const hash_multimap_t* cphmmmap_map);
```

描述:

获得引用 hash_multimap 容器中第一数据的迭代器。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。

返回值:

返回引用 hash_multimap 容器中第一个数据的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。如果 hash_multimap 容器为空, 则返回值与 hash_multimap_end(cphmmmap_map)相等。

```
hash_multimap_iterator_t hash_multimap_end(const hash_multimap_t* cphmmmap_map);
```

描述:

获得引用 hash_multimap 容器末尾的迭代器。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。

返回值:

返回引用 hash_multimap 容器末尾的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t hash_multimap_key_comp(const hash_multimap_t* cphmmmap_map);
```

描述:

返回 hash_multimap 容器中的键比较规则。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。

返回值:

返回 hash_multimap 容器中的数据比较规则。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。

```
binary_function_t hash_multimap_value_comp(const hash_multimap_t* cphmmmap_map);
```

描述:

返回 hash_multimap 容器中的数据比较规则。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。

返回值:

返回 hash_multimap 容器中的数据比较规则。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。

```
hash_multimap_iterator_t hash_multimap_find(
    const hash_multimap_t* cphmmmap_map, elem);
```

描述:

在 hash_multimap 容器中查找指定的数据。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。

elem 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
void hash_multimap_clear(hash_multimap_t* phmmmap_map);
```

描述:

删除 hash_multimap 容器中的所有数据。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。

返回值:

无。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为是未定义的。

```
size_t hash_multimap_count(const hash_multimap_t* cphmmmap_map, elem);
```

描述:

返回 hash_multimap 容器中指定数据的数量。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。

elem 要查找的指定数据。

返回值:

返回 hash_multimap 容器中指定数据的数量。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_multimap_iterator_t hash_multimap_lower_bound(
    const hash_multimap_t* cphmmmap_map, elem);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。

elem 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为

是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_multimap_iterator_t hash_multimap_upper_bound(
    const hash_multimap_t* cphmmmap_map, elem);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。
elem 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t hash_multimap_equal_range(const hash_multimap_t* cphmmmap_map, elem);
```

描述:

返回包含指定数据的数据区间。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。
elem 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cpt_hash_multimap == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
bool_t hash_multimap_equal(
    const hash_multimap_t* cphmmmap_first, const hash_multimap_t* cphmmmap_second);
```

描述:

测试两个 hash_multimap 容器是否相等。

参数:

cphmmmap_first 指向 hash_multimap 容器的指针。
cphmmmap_second 指向 hash_multimap 容器的指针。

返回值:

如果两个 hash_multimap 容器相等则返回 true，否则返回 false。

注意:

如果 cphmmmap_first == NULL 或者 cphmmmap_second == NULL 则函数的行为是未定义的，两个 hash_multimap 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同认为这两个容器不等。两个 hash_multimap 容器相等是指容器中的数据对应相等，并且容器中数据的数量也相等。如果 cphmmmap_first == cphmmmap_second 那么返回 true。

```
bool_t hash_multimap_not_equal(
    const hash_multimap_t* cphmmmap_first, const hash_multimap_t* cphmmmap_second);
```

描述:

测试两个 hash_multimap 容器是否不等。

参数:

cphmmmap_first	指向 hash_multimap 容器的指针。
cphmmmap_second	指向 hash_multimap 容器的指针。

返回值:

如果两个 hash_multimap 容器不等则返回 true, 否则返回 false。

注意:

如果 cphmmmap_first == NULL 或者 cphmmmap_second == NULL 则函数的行为是未定义的, 两个 hash_multimap 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同认为两个 hash_multimap 容器不等。两个 hash_multimap 容器不等是指容器中的对应的数据不相等, 如果对应的数据都相等, 那就要看容器中数据的数量是否不等。如果 cphmmmap_first == cphmmmap_second 那么返回 false。

```
bool_t hash_multimap_less(
    const hash_multimap_t* cphmmmap_first, const hash_multimap_t* cphmmmap_second);
```

描述:

测试第一个 hash_multimap 是否小于第二个 hash_multimap。

参数:

cphmmmap_first	指向 hash_multimap 容器的指针。
cphmmmap_second	指向 hash_multimap 容器的指针。

返回值:

如果第一个 hash_multimap 容器小于第二个 hash_multimap 容器则返回 true, 否则返回 false。

注意:

如果 cphmmmap_first == NULL 或者 cphmmmap_second == NULL 则函数的行为是未定义的, 两个 hash_multimap 必须已经初始化的, 否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同函数的行为是未定义的。如果第一个 hash_multimap 中的数据小于第二个 hash_multimap 中对应的数据则返回 true, 如果大于则返回 false, 当两个 hash_multimap 中的对应数据相等, 第一个 hash_multimap 中的数据数量小于第二个 hash_multimap 中数据的数量返回 true, 否则返回 false。如果 cphmmmap_first == cphmmmap_second 那么返回 false。

```
bool_t hash_multimap_less_equal(
    const hash_multimap_t* cphmmmap_first, const hash_multimap_t* cphmmmap_second);
```

描述:

测试第一个 hash_multimap 是否小于等于第二个 hash_multimap。

参数:

cphmmmap_first	指向 hash_multimap 容器的指针。
cphmmmap_second	指向 hash_multimap 容器的指针。

返回值:

如果第一个 hash_multimap 容器小于等于第二个 hash_multimap 容器则返回 true, 否则返回 false。

注意:

如果 cphmmmap_first == NULL 或者 cphmmmap_second == NULL 则函数的行为是未定义的, 两个 hash_multimap 必须

已经初始化的，否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同函数的行为是未定义的。如果 cphmmmap_first == cphmmmap_second 那么返回 true。

```
bool_t hash_multimap_greater(
    const hash_multimap_t* cphmmmap_first, const hash_multimap_t* cphmmmap_second);
```

描述:

测试第一个 hash_multimap 是否大于第二个 hash_multimap。

参数:

cphmmmap_first	指向 hash_multimap 容器的指针。
cphmmmap_second	指向 hash_multimap 容器的指针。

返回值:

如果第一个 hash_multimap 容器大于第二个 hash_multimap 容器则返回 true，否则返回 false。

注意:

如果 cphmmmap_first == NULL 或者 cphmmmap_second == NULL 则函数的行为是未定义的，两个 hash_multimap 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同函数的行为是未定义的。如果第一 hash_multimap 中的数据大于第二个 hash_multimap 中的对应的数据则返回 true，小于则返回 false，当对应的数据相等的时候，如果第一个 hash_multimap 中数据的数量大于第二个 hash_multimap 中数据的数量的时候返回 true 否则返回 false。如果 cphmmmap_first == cphmmmap_second 那么返回 false。

```
bool_t hash_multimap_greater_equal(
    const hash_multimap_t* cphmmmap_first, const hash_multimap_t* cphmmmap_second);
```

描述:

测试第一个 hash_multimap 是否大于等于第二个 hash_multimap。

参数:

cphmmmap_first	指向 hash_multimap 容器的指针。
cphmmmap_second	指向 hash_multimap 容器的指针。

返回值:

如果第一个 hash_multimap 容器大于等于第二个 hash_multimap 容器则返回 true，否则返回 false。

注意:

如果 cphmmmap_first == NULL 或者 cphmmmap_second == NULL 则函数的行为是未定义的，两个 hash_multimap 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同函数的行为是未定义的。如果 cphmmmap_first == cphmmmap_second 那么返回 true。

```
void hash_multimap_swap(
    hash_multimap_t* phmmmap_first, hash_multimap_t* phmmmap_second);
```

描述:

交换两个 hash_multimap 容器。

参数:

phmmmap_first	指向 hash_multimap 容器的指针。
phmmmap_second	指向 hash_multimap 容器的指针。

返回值:

无。

注意:

如果 phmmmap_first == NULL 或者 phmmmap_second == NULL 则函数的行为是未定义的，两个 hash_multimap 必须已经初始化的，否则函数的行为是未定义的。两个 hash_multimap 容器中保存的数据类型不同函数的行为是未定义的。如果 hash_multimap_equal(phmmmap_first, phmmmap_second)，函数不执行任何动作。

```
hash_multimap_iterator_t hash_multimap_insert(
    hash_multimap_t* phmmmap_map, const pair_t* cppair_pair);
```

描述:

向 hash_multimap 容器中插入唯一的数据。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
cppair_pair 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 hash_multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_multimap_iterator_t hash_multimap_insert_hint(
    hash_multimap_t* phmmmap_map,
    hash_multimap_iterator_t it_hint, const pair_t* cppair_pair);
```

描述:

向 hash_multimap 容器中的线索位置插入唯一的数据。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
it_hint 用户提供的线索位置。
cppair_pair 要插入的指定数据。

返回值:

指向插入的数据的迭代器，如果插入失败则返回指向末尾的迭代器。

注意:

如果 pt_hash_multimap == NULL 则函数的行为是未定义的，hash_multimap 容器必须已经初始化的，否则函数的行为是未定义的。插入的数据必须与保存在 hash_multimap 容器中的数据类型相同，否则函数的行为是未定义的。

```
void hash_multimap_insert_range(
    hash_multimap_t* phmmmap_map,
    hash_multimap_iterator_t it_begin, hash_multimap_iterator_t it_end);
```

描述:

向 hash_multimap 中插入指定的数据区间，数据必须唯一。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end)必须是有效的数据区间并且与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。[it_begin, it_end)必须不属于 phmmmap_map, 否则函数的行为是未定义的。

```
void hash_multimap_erase_pos(  
    hash_multimap_t* phmmmap_map, hash_multimap_iterator_t it_pos);
```

描述:

删除 hash_multimap 容器中指定位置的数据。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
it_pos 被删除的数据的位置。

返回值:

无。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。it_pos 是属于 phmmmap_map 容器的有效的迭代器, 否则函数的行为是未定义的。

```
void hash_multimap_erase_range(  
    hash_multimap_t* phmmmap_map,  
    hash_multimap_iterator_t it_begin, hash_multimap_iterator_t it_end);
```

描述:

删除 hash_multimap 中指定数据区间的数据。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
it_begin 指定的数据区间的开头。
it_end 指定的数据区间的末尾。

返回值:

无。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。[it_begin, it_end)必须属于 phmmmap_map, 否则函数的行为是未定义的。

```
void hash_multimap_erase(hash_multimap_t* phmmmap_map, elem);
```

描述:

删除 hash_multimap 中指定的数据。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
elem 要删除的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 ph mmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

第五节 迭代器接口

hash_map 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

create_hash_map_iterator	创建 hash_map 迭代器。
_hash_map_iterator_get_value	获得 hash_map 迭代器引用的数据。
_hash_map_iterator_get_pointer	获得 hash_map 迭代器引用的数据的指针。
_hash_map_iterator_next	获得引用下一个数据的迭代器。
_hash_map_iterator_prev	获得引用前一个数据的迭代器。
_hash_map_iterator_equal	测试两个迭代器是否相等。
_hash_map_iterator_distance	计算两个迭代器之间的距离。
_hash_map_iterator_before	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
hash_map_iterator_t create_hash_map_iterator(void);
```

描述:

创建一个 hash_map 迭代器。

参数:

无。

返回值:

hash_map 迭代器。

注意:

返回的 hash_map 迭代器并不是有效的迭代器, 它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作, 返回的迭代器供其他接口使用。

```
void _hash_map_iterator_get_value(hash_map_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

it_iter hash_map 迭代器。

pv_value 保存数据的缓冲区。

返回值:

无。

注意:

it_iter 是有效的 hash_map 迭代器, 否则函数的行为是未定义的。pv_value == NULL 则函数的行为是未定义的, pv_value 是能够保存下 it_iter 引用的数据的缓冲区, 否则函数的行为是未定义的。函数执行后 it_iter 引用的数据被拷贝到 pv_value 指向的缓存区中。

```
const void* _hash_map_iterator_get_pointer(hash_map_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

it_iter hash_map 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 hash_map 迭代器, 否则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_iterator_next(hash_map_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter hash_map 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 hash_map 迭代器, 否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 hash_map 有效的迭代器, 否则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_iterator_prev(hash_map_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter hash_map 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 hash_map 迭代器, 否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 hash_map 有效的迭代器, 否则函数的行为是未定义的。

```
bool_t _hash_map_iterator_equal(
    hash_map_iterator_t it_first, hash_map_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first hash_map 迭代器。
it_second hash_map 迭代器。

返回值:

如果两个迭代器相等则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 hash_map 容器的有效的 hash_map 迭代器, 否则函数的行为是未定义的。两个 hash_map 迭代器相等是指两个迭代器引用相同的数据。

```
int _hash_map_iterator_distance(  
    hash_map_iterator_t it_first, hash_map_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_first hash_map 迭代器。
it_second hash_map 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 hash_map 容器的有效的 hash_map 迭代器, 否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0, it_first 引用的数据在 it_second 后面那么结果<0, 如果两个迭代器相等那么结果为0。

```
bool_t _hash_map_iterator_before(  
    hash_map_iterator_t it_first, hash_map_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first hash_map 迭代器。
it_second hash_map 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true, 否则返回 false。

注意:

两个 iter 是属于同一个 hash_map 容器的有效的 hash_map 迭代器, 否则函数的行为是未定义的。

hash_multimap 的迭代器是双向的迭代器, 所以它没有随机访问迭代器那么强大的功能, 向前和向后移动只能单步运行。这些接口只能由迭代器接口使用, 用户不应该直接使用这些接口函数。

create_hash_multimap_iterator	创建 hash_multimap 迭代器。
-------------------------------	-----------------------

| _hash_multimap_iterator_get_value | 获得 hash_multimap 迭代器引用的数据。 |
| _hash_multimap_iterator_get_pointer | 获得 hash_multimap 迭代器引用的数据的指针。 |

<code>_hash_multimap_iterator_next</code>	获得引用下一个数据的迭代器。
<code>_hash_multimap_iterator_prev</code>	获得引用前一个数据的迭代器。
<code>_hash_multimap_iterator_equal</code>	测试两个迭代器是否相等。
<code>_hash_multimap_iterator_distance</code>	计算两个迭代器之间的距离。
<code>_hash_multimap_iterator_before</code>	测试第一个迭代器是否位于第二个迭代器前面。

函数原型

```
hash_multimap_iterator_t create_hash_multimap_iterator(void);
```

描述:

创建一个 hash_multimap 迭代器。

参数:

无。

返回值:

hash_multimap 迭代器。

注意:

返回的 hash_multimap 迭代器并不是有效的迭代器，它没有与任何迭代器关联。这个函数只是做了一些必要的初始化工作，返回的迭代器供其他接口使用。

```
void _hash_multimap_iterator_get_value(
    hash_multimap_iterator_t it_iter, void* pv_value);
```

描述:

获得迭代器引用的数据。

参数:

`it_iter` hash_multimap 迭代器。

`pv_value` 保存数据的缓冲区。

返回值:

无。

注意:

`it_iter` 是有效的 hash_multimap 迭代器，否则函数的行为是未定义的。`pv_value == NULL` 则函数的行为是未定义的，`pv_value` 是能够保存下 `it_iter` 引用的数据的缓冲区，否则函数的行为是未定义的。函数执行后 `it_iter` 引用的数据被拷贝到 `pv_value` 指向的缓存区中。

```
const void* _hash_multimap_iterator_get_pointer(hash_multimap_iterator_t it_iter);
```

描述:

获得迭代器引用的数据的指针。

参数:

`it_iter` hash_multimap 迭代器。

返回值:

指向迭代器引用的数据的指针。

注意:

it_iter 是有效的 hash_multimap 迭代器，否则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_iterator_next(
    hash_multimap_iterator_t it_iter);
```

描述:

获得引用下一个数据的迭代器。

参数:

it_iter hash_multimap 迭代器。

返回值:

返回引用下一个数据的迭代器。

注意:

it_iter 是有效的 hash_multimap 迭代器，否则函数的行为是未定义的。引用下一个数据的迭代器也必须是 hash_multimap 有效的迭代器，否则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_iterator_prev(
    hash_multimap_iterator_t it_iter);
```

描述:

获得引用上一个数据的迭代器。

参数:

it_iter hash_multimap 迭代器。

返回值:

返回引用上一个数据的迭代器。

注意:

it_iter 是有效的 hash_multimap 迭代器，否则函数的行为是未定义的。引用上一个数据的迭代器也必须是 hash_multimap 有效的迭代器，否则函数的行为是未定义的。

```
bool_t _hash_multimap_iterator_equal(
    hash_multimap_iterator_t it_first, hash_multimap_iterator_t it_second);
```

描述:

测试两个迭代器是否相等。

参数:

it_first hash_multimap 迭代器。

it_second hash_multimap 迭代器。

返回值:

如果两个迭代器相等则返回 true，否则返回 false。

注意:

两个 iter 是属于同一个 hash_multimap 容器的有效的 hash_multimap 迭代器，否则函数的行为是未定义的。两个 hash_multimap 迭代器相等是指两个迭代器引用相同的数据。

```
int _hash_multimap_iterator_distance(
    hash_multimap_iterator_t it_first, hash_multimap_iterator_t it_second);
```

描述:

计算两个迭代器之间的距离。

参数:

it_first hash_multimap 迭代器。
it_second hash_multimap 迭代器。

返回值:

返回两个迭代器之间的距离。

注意:

两个 iter 是属于同一个 hash_multimap 容器的有效的 hash_multimap 迭代器，否则函数的行为是未定义的。如果 it_first 引用的数据在 it_second 前面那么结果>0，it_first 引用的数据在 it_second 后面那么结果<0，如果两个迭代器相等那么结果为 0。

```
bool_t _hash_multimap_iterator_before(  
    hash_multimap_iterator_t it_first, hash_multimap_iterator_t it_second);
```

描述:

测试第一个迭代器是否在第二个迭代器的前面。

参数:

it_first hash_multimap 迭代器。
it_second hash_multimap 迭代器。

返回值:

如果第一个迭代器在第二个迭代器的前面则返回 true，否则返回 false。

注意:

两个 iter 是属于同一个 hash_multimap 容器的有效的 hash_multimap 迭代器，否则函数的行为是未定义的。

第六节 内部和辅助接口

hash_map 提供的内部接口是为了给外部接口使用。

_create_hash_map	创建 hash_map 容器。
_create_hash_map_auxiliary	创建 hash_map 容器的辅助函数。
_hash_map_destroy_auxiliary	销毁 hash_map 容器的辅助函数。
_hash_map_find	在 hash_map 容器中查找指定的数据。
_hash_map_find_varg	在 hash_map 容器中查找指定的数据，数据来自于可变参数列表。
_hash_map_count	返回 hash_map 容器中指定数据的数量。
_hash_map_count_varg	返回 hash_map 容器中指定数据的数量，数据来自于可变参数列表。
_hash_map_lower_bound	返回指向第一个大于等于指定数据的迭代器。
_hash_map_lower_bound_varg	返回指向第一个大于等于指定数据的迭代器，数据来自于可变参数列表。
_hash_map_upper_bound	返回指向第一个大于指定数据的迭代器。
_hash_map_upper_bound_varg	返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。

<code>_hash_map_equal_range</code>	返回包含指定数据的数据区间。
<code>_hash_map_equal_range_varg</code>	返回包含指定数据的数据区间，数据来自于可变参数列表。
<code>_hash_map_at</code>	使用下标对 <code>hash_map</code> 中的数据进行访问。
<code>_hash_map_at_varg</code>	使用下标对 <code>hash_map</code> 中的数据进行访问，数据来自于可变参数列表。
<code>_hash_map_erase</code>	删除 <code>hash_map</code> 中指定的数据。
<code>_hash_map_erase_varg</code>	删除 <code>hash_map</code> 中指定的数据，数据来自于可变参数列表。
<code>_hash_map_init_elem_auxiliary</code>	初始化数据的辅助函数。

函数原型

```
hash_map_t* _create_hash_map(const char* s_typename);
```

描述:

创建一个 `hash_map` 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 `hash_map` 容器的指针，否则返回 NULL。

注意:

`s_typename` != NULL，否则函数的行为是未定义的。

```
bool_t _create_hash_map_auxiliary(hash_map_t* phmap_map, const char* s_typename);
```

描述:

创建一个 `hash_map` 容器的辅助函数。

参数:

`phmap_map` 没有创建的 `hash_map` 容器。

`s_typename` `hash_map` 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 `phmap_map` == NULL 或者 `s_typename` == NULL 则函数的行为是未定义的。`s_typename` 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型，否则创建失败。

```
void _hash_map_destroy_auxiliary(hash_map_t* phmap_map);
```

描述:

销毁 `hash_map` 容器的辅助函数。

参数:

`phmap_map` `hash_map` 容器。

返回值:

无。

注意:

如果 `phmap_map` == NULL 或者 `hash_map` 不是使用 `_create_hash_map` 生成的则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_find(const hash_map_t* cphmap_map, ...);
```

描述:

在 hash_map 容器中查找指定的数据。

参数:

cphmap_map 指向 hash_map 容器的指针。
... 要查找的指定数据。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_find_varg(  
    const hash_map_t* cphmap_map, va_list val_elemlist);
```

描述:

在 hash_map 容器中查找指定的数据，数据来自于参数列表。

参数:

cphmap_map 指向 hash_map 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器，否则返回指向末尾的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _hash_map_count(const hash_map_t* cphmap_map, ...);
```

描述:

返回 hash_map 容器中指定数据的数量。

参数:

cphmap_map 指向 hash_map 容器的指针。
... 要查找的指定数据。

返回值:

返回 hash_map 容器中指定数据的数量。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。

```
size_t _hash_map_count_varg(const hash_map_t* cphmap_map, va_list val_elemlist);
```

描述:

返回 hash_map 容器中指定数据的数量，数据来自于参数列表。

参数:

cphmap_map 指向 hash_map 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回 hash_map 容器中指定数据的数量。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, hash_map 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_map 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_lower_bound(const hash_map_t* cphmap_map, ...);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cphmap_map 指 hash_map 向容器的指针。
... 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_lower_bound_varg(  
    const hash_map_t* cphmap_map, va_list val_elemlist);
```

描述:

返回指向第一个大于等于指定数据的迭代器, 数据来自于可变参数列表。

参数:

cphmap_map 指 hash_map 向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cphmap_map == NULL 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_upper_bound(const hash_map_t* cphmap_map, ...);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

cphmap_map 指 hash_map 向容器的指针。
... 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cphmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_map_iterator_t _hash_map_upper_bound_varg(
    const hash_map_t* cphmap_map, va_list val_elemlist);
```

描述:

返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。

参数:

`cphmap_map` 指 `hash_map` 向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cphmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _hash_map_equal_range(const hash_map_t* cphmap_map, ...);
```

描述:

返回包含指定数据的数据区间。

参数:

`cphmap_map` 指向容器的指针。
... 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 `cphmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _hash_map_equal_range_varg(
    const hash_map_t* cphmap_map, va_list val_elemlist);
```

描述:

返回包含指定数据的数据区间。

参数:

`cphmap_map` 指向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回包含指定数据的数据区间。

注意:

如果 `cphmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
void* _hash_map_at(const hash_map_t* cphmap_map, ...);
```

描述:

通过下标访问 hash_map 中的数据。

参数:

phmap_map 指向容器的指针。
... 要查找的指定数据。

返回值:

返回以指定数据为键的数据的指针。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 hash_map 中不包含以指定的数据为键的数据，则首先向 hash_map 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

```
void* _hash_map_at_varg(const hash_map_t* phmap_map, va_list val_elemlist);
```

描述:

通过下标访问 hash_map 中的数据。

参数:

phmap_map 指向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回以指定数据为键的数据的指针。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 hash_map 中不包含以指定的数据为键的数据，则首先向 hash_map 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

```
void _hash_map_erase(hash_map_t* phmap_map, ...);
```

描述:

删除 hash_map 中指定的数据。

参数:

phmap_map 指向 hash_map 容器的指针。
... 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 phmap_map == NULL 则函数的行为是未定义的，hash_map 容器必须已经初始化的，否则函数的行为是未定义的。指定的数据必须与保存在 hash_map 容器中的数据类型相同，否则函数的行为是未定义的。

```
void _hash_map_erase_varg(hash_map_t* phmap_map, va_list val_elemlist);
```

描述:

删除 hash_map 中指定的数据，数据来自于可变参数列表。

参数:

phmap_map 指向 hash_map 容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 `phmap_map == NULL` 则函数的行为是未定义的, `hash_map` 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 `hash_map` 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _hash_map_init_elem_auxiliary(hash_map_t* phmap_map, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

`phmap_map` `hash_map` 容器。

`pv_value` 初始化的数据。

返回值:

无。

注意:

如果 `phmap_map == NULL` 或者 `pv_value == NULL`, 那么函数的行为是未定义的。`phmap_map` 必须是初始化的或者是使用 `_create_hash_map()` 创建的, 否则函数的行为是未定义的。

`hash_map` 提供了少量的辅助函数。

<code>_hash_map_same_pair_type</code>	判断两个 pair 类型是否相同。
<code>_hash_map_same_pair_type_ex</code>	判断两个 pair 类型是否相同。
<code>_hash_map_value_compare</code>	<code>hash_map</code> 的 key 和 value 比较函数。
<code>_hash_map_default_hash</code>	<code>hash_map</code> 默认的 hash 函数。

函数原型

```
bool_t _hash_map_same_pair_type(
    const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

`ppair_first` 第一个 pair 类型。

`ppair_second` 第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 `true`, 否则返回 `false`。

注意:

如果 `ppair_first == NULL` 或者 `ppair_second == NULL`, 那么函数的行为是未定义的。

```
bool_t _hash_map_same_pair_type_ex(
    const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

ppair_first 第一个 pair 类型。
ppair_second 第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true, 否则返回 false。

注意:

如果 ppair_first == NULL 或者 ppair_second == NULL, 那么函数的行为是未定义的。

```
void _hash_map_value_compare(  
    const void* cpv_first, const void* cpv_second, void* pv_output);
```

描述:

hash_map 值的比较函数。

参数:

cpv_first 第一个数据。
cpv_second 第二个数据。
pv_output 输出结构。

返回值:

无。

注意:

这个函数是 hash_map 的数据比较函数, 这个函数主要提供给 hashtable 比较数据使用。如果 cpv_first == NULL 或者 cpv_second == NULL 或者 pv_outpu == NULL 那么这个函数的行为是未定义的。cpv_first 和 cpv_second 必须是相同的 pair 类型, 否则函数的行为是未定义的。

```
void _hash_map_default_hash(const void* cpv_input, void* pv_output);
```

描述:

hash_map 默认的 hash 函数。

参数:

cpv_input 第一个数据。
pv_output 输出结构。

返回值:

无。

注意:

这个函数是 hash_map 的默认的比较, 这个函数主要提供给 hashtable 计算 hash 使用。如果 cpv_input == NULL 或者 pv_outpu == NULL 那么这个函数的行为是未定义的。cpv_input 必须是 pair 类型, 否则函数的行为是未定义的。

hash_multimap 提供的内部接口是为了给外部接口使用。

_create_hash_multimap	创建 hash_multimap 容器。
_create_hash_multimap_auxiliary	创建 hash_multimap 容器的辅助函数。

<code>_hash_multimap_destroy_auxiliary</code>	销毁 hash_multimap 容器的辅助函数。
<code>_hash_multimap_find</code>	在 hash_multimap 容器中查找指定的数据。
<code>_hash_multimap_find_varg</code>	在 hash_multimap 容器中查找指定的数据，数据来自于可变参数列表。
<code>_hash_multimap_count</code>	返回 hash_multimap 容器中指定数据的数量。
<code>_hash_multimap_count_varg</code>	返回 hash_multimap 容器中指定数据的数量，数据来自于可变参数列表。
<code>_hash_multimap_lower_bound</code>	返回指向第一个大于等于指定数据的迭代器。
<code>_hash_multimap_lower_bound_varg</code>	返回指向第一个大于等于指定数据的迭代器，数据来自于可变参数列表。
<code>_hash_multimap_upper_bound</code>	返回指向第一个大于指定数据的迭代器。
<code>_hash_multimap_upper_bound_varg</code>	返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。
<code>_hash_multimap_equal_range</code>	返回包含指定数据的数据区间。
<code>_hash_multimap_equal_range_varg</code>	返回包含指定数据的数据区间，数据来自于可变参数列表。
<code>_hash_multimap_at</code>	使用下标对 hash_multimap 中的数据进行访问。
<code>_hash_multimap_at_varg</code>	使用下标对 hash_multimap 中的数据进行访问，数据来自于可变参数列表。
<code>_hash_multimap_erase</code>	删除 hash_multimap 中指定的数据。
<code>_hash_multimap_erase_varg</code>	删除 hash_multimap 中指定的数据，数据来自于可变参数列表。
<code>_hash_multimap_init_elem_auxiliary</code>	初始化数据的辅助函数。

函数原型

```
hash_multimap_t* _create_hash_multimap(const char* s_typename);
```

描述:

创建一个 hash_multimap 迭代器。

参数:

`s_typename` 类型描述。

返回值:

成功返回指向 hash_multimap 容器的指针，否则返回 NULL。

注意:

`s_typename != NULL`，否则函数的行为是未定义的。

```
bool_t _create_hash_multimap_auxiliary(
    hash_multimap_t* phmmmap_map, const char* s_typename);
```

描述:

创建一个 hash_multimap 容器的辅助函数。

参数:

`phmmmap_map` 没有创建的 hash_multimap 容器。

`s_typename` hash_multimap 中保存的数据类型的名字。

返回值:

创建成功返回 true, 否则返回 false。

注意:

如果 phmmmap_map == NULL 或者 s_typename == NULL 则函数的行为是未定义的。s_typename 必须是 C 内建类型, libcstl 内建类型以及已经注册的用户自定义类型, 否则创建失败。

```
void _hash_multimap_destroy_auxiliary(hash_multimap_t* phmmmap_map);
```

描述:

销毁 hash_multimap 容器的辅助函数。

参数:

phmmmap_map hash_multimap 容器。

返回值:

无。

注意:

如果 phmmmap_map == NULL 或者 hash_multimap 不是使用_create_hash_multimap 生成的则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_find(
    const hash_multimap_t* cphmmmap_map, ...);
```

描述:

在 hash_multimap 容器中查找指定的数据。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。
... 要查找的指定数据。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_find_varg(
    const hash_multimap_t* cphmmmap_map, va_list val_elemlist);
```

描述:

在 hash_multimap 容器中查找指定的数据, 数据来自于参数列表。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回查找到的数据的迭代器, 否则返回指向末尾的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
size_t _hash_multimap_count(const hash_multimap_t* cphmmmap_map, ...);
```

描述:

返回 hash_multimap 容器中指定数据的数量。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。
... 要查找的指定数据。

返回值:

返回 hash_multimap 容器中指定数据的数量。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
size_t _hash_multimap_count_varg(  
    const hash_multimap_t* cphmmmap_map, va_list val_elemlist);
```

描述:

返回 hash_multimap 容器中指定数据的数量, 数据来自于参数列表。

参数:

cphmmmap_map 指向 hash_multimap 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

返回 hash_multimap 容器中指定数据的数量。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_lower_bound(  
    const hash_multimap_t* cphmmmap_map, ...);
```

描述:

返回指向第一个大于等于指定数据的迭代器。

参数:

cphmmmap_map 指 hash_multimap 向容器的指针。
... 要查找的指定数据。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的, 容器必须已经初始化的, 否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同, 否则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_lower_bound_varg(  
    const hash_multimap_t* cphmmmap_map, va_list val_elemlist);
```

描述:

返回指向第一个大于等于指定数据的迭代器, 数据来自于可变参数列表。

参数:

cphmmmap_map 指 hash_multimap 向容器的指针。

`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回指向第一个大于等于指定数据的迭代器。

注意:

如果 `cphmmmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_upper_bound(
    const hash_multimap_t* cphmmmap_map, ...);
```

描述:

返回指向第一个大于指定数据的迭代器。

参数:

`cphmmmap_map` 指 `hash_multimap` 向容器的指针。
... 要查找的指定数据。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cphmmmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
hash_multimap_iterator_t _hash_multimap_upper_bound_varg(
    const hash_multimap_t* cphmmmap_map, va_list val_elemlist);
```

描述:

返回指向第一个大于指定数据的迭代器，数据来自于可变参数列表。

参数:

`cphmmmap_map` 指 `hash_multimap` 向容器的指针。
`val_elemlist` 用户指定的数据的参数列表。

返回值:

返回指向第一个大于指定数据的迭代器。

注意:

如果 `cphmmmap_map == NULL` 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _hash_multimap_equal_range(const hash_multimap_t* cphmmmap_map, ...);
```

描述:

返回包含指定数据的数据区间。

参数:

`cphmmmap_map` 指向容器的指针。
... 要查找的指定数据。

返回值:

返回包含指定数据的数据区间。

注意:

如果 cphmmmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
range_t _hash_multimap_equal_range_varg(
    const hash_multimap_t* cphmmmap_map, va_list val_elemlist);
```

描述：

返回包含指定数据的数据区间。

参数：

cphmmmap_map 指向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值：

返回包含指定数据的数据区间。

注意：

如果 cphmmmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。

```
void* _hash_multimap_at(const hash_multimap_t* cphmmmap_map, ...);
```

描述：

通过下标访问 hash_multimap 中的数据。

参数：

phmmmap_map 指向容器的指针。
... 要查找的指定数据。

返回值：

返回以指定数据为键的数据的指针。

注意：

如果 phmmmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 hash_multimap 中不包含以指定的数据为键的数据，则首先向 hash_multimap 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

```
void* _hash_multimap_at_varg(
    const hash_multimap_t* phmmmap_map, va_list val_elemlist);
```

描述：

通过下标访问 hash_multimap 中的数据。

参数：

phmmmap_map 指向容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值：

返回以指定数据为键的数据的指针。

注意：

如果 phmmmap_map == NULL 则函数的行为是未定义的，容器必须已经初始化的，否则函数的行为是未定义的。要查找的数据必须与保存在容器中的数据类型相同，否则函数的行为是未定义的。如果 hash_multimap 中不包含以指定的数据为键的数据，则首先向 hash_multimap 中插入以指定数据为键的 pair 数据，并且返回第二个数据的指针。

```
void _hash_multimap_erase(hash_multimap_t* phmmmap_map, ...);
```

描述:

删除 hash_multimap 中指定的数据。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
... 要查找的指定数据。

返回值:

被删除的数据的个数。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _hash_multimap_erase_varg(hash_multimap_t* phmmmap_map, va_list val_elemlist);
```

描述:

删除 hash_multimap 中指定的数据, 数据来自于可变参数列表。

参数:

phmmmap_map 指向 hash_multimap 容器的指针。
val_elemlist 用户指定的数据的参数列表。

返回值:

被删除的数据的个数。

注意:

如果 phmmmap_map == NULL 则函数的行为是未定义的, hash_multimap 容器必须已经初始化的, 否则函数的行为是未定义的。指定的数据必须与保存在 hash_multimap 容器中的数据类型相同, 否则函数的行为是未定义的。

```
void _hash_multimap_init_elem_auxiliary(  
    hash_multimap_t* phmmmap_map, void* pv_value);
```

描述:

初始化数据的辅助函数。

参数:

phmmmap_map hash_multimap 容器。
pv_value 初始化的数据。

返回值:

无。

注意:

如果 phmmmap_map == NULL 或者 pv_value == NULL , 那么函数的行为是未定义的。phmmmap_map 必须是初始化的或者是使用_create_hash_multimap()创建的, 否则函数的行为是未定义的。

hash_multimap 提供了少量的辅助函数。

_hash_multimap_same_pair_type	判断两个 pair 类型是否相同。
-------------------------------	-------------------

_hash_multimap_same_pair_type_ex	判断两个 pair 类型是否相同。
----------------------------------	-------------------

<code>_hash_multimap_value_compare</code>	hash_multimap 的 key 和 value 比较函数。
<code>_hash_multimap_default_hash</code>	hash_multimap 默认的 hash 函数。

函数原型

```
bool_t _hash_multimap_same_pair_type(
    const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

<code>ppair_first</code>	第一个 pair 类型。
<code>ppair_second</code>	第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true, 否则返回 false。

注意:

如果 `ppair_first == NULL` 或者 `ppair_second == NULL`, 那么函数的行为是未定义的。

```
bool_t _hash_multimap_same_pair_type_ex(
    const pair_t* ppair_first, const pair_t* ppair_second);
```

描述:

判断两个 pair 类型是否相同。

参数:

<code>ppair_first</code>	第一个 pair 类型。
<code>ppair_second</code>	第二个 pair 类型。

返回值:

如果两个 pair 类型相同返回 true, 否则返回 false。

注意:

如果 `ppair_first == NULL` 或者 `ppair_second == NULL`, 那么函数的行为是未定义的。

```
void _hash_multimap_value_compare(
    const void* cpv_first, const void* cpv_second, void* pv_output);
```

描述:

hash_multimap 值的比较函数。

参数:

<code>cpv_first</code>	第一个数据。
<code>cpv_second</code>	第二个数据。
<code>pv_output</code>	输出结构。

返回值:

无。

注意:

这个函数是 hash_multimap 的数据比较函数, 这个函数主要提供给 hashtable 比较数据使用。如果 `cpv_first == NULL` 或者 `cpv_second == NULL` 或者 `pv_output == NULL` 那么这个函数的行为是未定义的。`cpv_first` 和 `cpv_second` 必须是相同

的 pair 类型，否则函数的行为是未定义的。

```
void _hash_multimap_default_hash(const void* cpv_input, void* pv_output);
```

描述：

hash_multimap 默认的 hash 函数。

参数：

cpv_input	第一个数据。
pv_output	输出结构。

返回值：

无。

注意：

这个函数是 hash_multimap 的默认的比较，这个函数主要提供给 hashtable 计算 hash 使用。如果 cpv_input == NULL 或者 pv_outpu == NULL 那么这个函数的行为是未定义的。cpv_input 必须是 pair 类型，否则函数的行为是未定义的。

第二十二章 libcstl 2.1 更新

第一节 libcstl 2.1 新特性

- 容器迭代器的互操作性，一种容器可以使用另一种容器的迭代器，只要两种容器中保存的数据类型相同。
- 增加使用数组初始化容器的接口。
- 增加版本标识宏 libcstl 2.1.0 为 CSTL_VERSION 20100 CSTL_MAJOR_VERSION 2 CSTL_MINOR_VERSION 1 CSTL_REVISION_VERSION 0
- 增加 iterator_set_elem 设置成使用可变参数的函数。
- 添加对于 size_t 类型的支持。
- 提供默认的谓词 bfun_default_predicate 和 ufun_default_predicate 区别于默认的函数 bfun_default 和 ufun_default。

第二节 libcstl 2.1 改善

- 容器初始化后占用的内存过大。在 2.1 中间内存管理作为可选项使用。默认不使用内存管理。
- vector 容器在添加少量数据的时候重新分配的内存太小，导致后续的操作很快就用完了预留的内存。
- 增加容错的处理，减少外部接口的未定义情况，加强一致性有必要时要增加未定义的情况。
- 添加 VS2003 和 VS2010 的工程。
- 有内存泄漏的情况，每次 500~1k 左右。
- 改善目前已知的限制性，不能够对于 libcstl 内建类型进行重命名，对于使用范围的函数，当范围属于这个容器本身的时候，函数的行为是未定义的。
- 目前 string_t 类型与 STL 中的 string_t 类型行为不一致，主要是由于 string_t 类型的实现造成的，可能在 2.1 中更改这个实现。

第三节 libcstl 2.1 代码的优化

- 使用现有的算法实现部分容器接口的实现，这样减少代码的重复。
- 在同一容器中尽量使用同一算法来实现不同的接口。例如实现删除操作的有多个接口，可以提供一个删除操作的接口然后将它包装成多个接口。
- 调整代码结构，将实现代码从一个文件中拆分出来，减小 c 文件的大小。
- 检查代码中同时使用两个容器的指针的代码，例如 `vector_assign(pvec_dest, pvec_src);` 这样的代码中如果 `pvec_dest == pvec_src` 代码中是否做了处理，以及类似的当插入或者删除的数据个数为 0 或者区间为空的时候的情况。
- 在对于容器和数据区间之间进行操作的情况，尤其要考虑数据区间输入容器的时候的特殊处理。
- 修改各个容器的 `create_xxxx_iterator()` 接口，将这个接口改为迭代器内部接口 `_create_xxxx_iterator()`。
- 修改代码实现，是代码符合编码规范以及命名规则。
- 将有符号的 int 类型都改成有符号的 long 类型以适应 64 位环境。

- 修改根目录下的 Makefile.am，将 LICENSE 发布。或者改成 COPYING.LESSER(符合 GNU 标准的名字)。
- 使用 gcov 或者 gprof 对代码进行优化。使用 valgrind 和 vld 进行内存泄漏检查。使用 pclint 进行代码静态检查。
- 将代码中函数名统一，辅助函数也要以相应的容器或者算法为前缀。函数参数也要符合编码规范。
- 制定一个工程规范规范。
- 对于操作数据区间的接口函数，尤其要注意数据区间属于当前容器是的处理。如果 `list_insert_range(pt_list, t_pos, it_begin, it_end);` 当 `[it_begin, it_end)` 属于 `pt_list` 的时候的处理要注意。
- 对于接口中未定义的行为，debug 版本的库都要触发断言。
- 对于每一个容器类型都提供一个 `xxxx_is_created` 和 `xxxx_is_init` 函数用来检查容器是否是创建的以及容器是否已经初始化，这两个函数主要用于实现接口时断言使用。同时 `_alloc_t` 也要增加这个 `_alloc_is_init` 函数。
- 为容器的初始化函数增加检查是否容器是使用 `create` 函数创建的，为其他的函数检查容器是否是已经初始化的，`destroy` 函数可以销毁创建和初始化的容器。
- 将代码中的 `t_index` 全部使用 `i` 替换。
- 将代码中对于 `char*` 类型的处理隐藏到迭代器接口下面。在容器或者算法的实现中尽量不出现对于 `char*` 类型的处理。(这个可能要调整算法，因为对于 `char*` 类型的默认函数是直接对于 `string_t` 类型进行处理的，这与用户定义的函数的接口不同，主要是 `_type_less_cstr` 这个函数的接口的变化)。
- 容器内不实现尽量使用迭代器实现。
- 注意一些带有 `_if` 后缀的函数当函数为 `NULL` 的时候，使用的默认函数是 `fun_default_binary` 或者 `fun_default_unary` 还是类型注册是使用的与类型相关的小于比较函数。建议在使用 `fun_default_unary` 的地方使用将来增加的 `fun_default_unary_predicate` 在使用 `fun_default_binary` 的地方使用视情况而定，如果与小于语义相符，那么就使用与类型相关的小于比较函数，否则使用 `fun_default_binary_predicate`。(这个修改要等到将类型相关的代码重构之后再修改，主要是 `_type_less_cstr` 的问题)。
- 将 `_list_quick_sort` 修改成使用迭代器的版本。(有利于解决对于 `char*` 类型的代码重复实现的问题，这要等 `_type_less_cstr` 修改了以后再修改。)
- 标准并没有规定有符号的除法以及取余的商和余数的符号是什么，所以要注意代码中有符号的除法和取余的计算的地方。(最好的方法是分开写。)
- 使用容器的 `xxxx_init_n()` 函数实现 `xxxx_init_elem()` 函数，去掉两个函数之间的重复。DRY!!!!
- 对于使用 `iterator_distance` 的地方统一方法，在使用前先判断返回值的符号，然后转换成 `size_t` 类型。
- 将宏的名字定义的更规范如 `_GET_SLIST_TYPE_NAME` 改成 `_SLIST_GET_TYPE_NAME`。
- `_type_get_type` 中的 `bool_t` 类型要使用 `size_t` 来获得。
- 对于 `basic_string` 使用带有 `cstr` 后缀的函数时，传入的数组是使用数组中保存的是实例还是指针这是个问题。使用指针可以通过 `NULL` 来表示数组结束，同时在保存 `libcstl` 内建类型的时候还可以进行有效的初始化。现在的形式是使用实例数组的，要将这种方式改动的话，`_basic_string_get_value_string_length` 函数也要改动，主要是字符串确定方法的改动。
- `basic_string_copy` 现在只针对 `_TYPE_C_BUILTIN` 并且不包括 `char*` 类型进行了处理，这主要是由于没有考虑好 `buffer` 的形式，如 `basic_string` 保存 `char*` 类型的时候，`buffer` 应该是 `char* buffer[]` 这样的形式，`_TYPE_CSTL_BUILTIN` 是应该是 `vector_t buffer[]`，`_TYPE_USER_DEFINED` 就应该是 `abc_t buffer[]`。
- `basic_string_find` 系列函数没有对于 c 内建类型以外的类型进行测试，这里需要增加新的测试用例来测试其他类型。
- 检查 `basic_string` 的操作函数中两个函数都是 `basic_string_t*` 类型时，当第一个参数等以第二个参数时的语义。
- 检查 `basic_string` 的函数中 `_cstr` 和 `_subcstr` 以及 `_string` 和 `_substring` 这样关系的函数中前者是否使用了带有 `NPOS` 的后者实现。
- 对于 `basic_string` 的 `find`, `rfind`, `find_first_of`, `find_first_not_of`, `find_last_of`, `find_last_not_of` 这样的函数的 `basic_string` 为空或者要查找的串为空，或者两者为空的情况，以及位置，长度等参数的各种情况要清晰和统一。原则是 `t_pos` 位置是否有效，`t_len` 表示长度可是使用 `NPOS` 等大于字符串长度的值，当 `basic_string` 为空返

回 NPOS，当 basic_string 不为空，value string 为空的时候，返回有效的 t_pos，在 rfind 中当 t_pos 是大于字符串长度的值的时候，返回最有一个有效字符的位置。要和 SGI STL 的行为一致。

- 将 basic_string 结构中的 _t_vector 替换成 _vec_base。
- 据说 vector_t* pt_vec = create_vector(string_t); 在销毁的时候会有内存泄漏，看看到底是什么情况，在其他类型中也实验一下。
- 对于 basic_string_insert 系列函数，当 t_pos 超出 basic_string 有效范围的时候的处理情况要和 SGI STL 的行为一致。目前当 t_pos 为无效位置的时候出现未定义行为。
- 对于所有的容器，当操作函数对于两个容器指针进行操作的时候，以及对于一个指针一个数据区间进行操作的时候，一定要注意当两个指针相等，或者两个容器中的数据相等，或者数据区间属于第一个容器是函数的行为是否应该为未定义的，或者是需要特殊的处理。
- 对于代码中所有的局部变量都要进行初始化。
- 考虑 basic_string_t* pt_basic_string = create_basic_string(char*); 时 basic_string_c_str(pt_basic_string); 和 basic_string_data(pt_basic_string); 的返回值是什么情况。
- basic_string_insert_subcstr 与其他的 _subcstr 函数实现方式不一致，考虑是否要修改。
- 将 basic_string 函数中带有多个 t_pos, t_len 的函数参数名称修改成 t_pos0, t_len0, t_pos1, t_len1 等等。
- 缺少函数 create_string_iterator();
- 确定 xxxx_is_created() 和 xxxx_is_init() 是否应该提供给上层使用，如 vector 和 avl_tree 这样的结构就应该提供给上层使用。
- pair_assign 可能需要优化，在赋值之前判断 pair_equal 是需要优化掉的，因为在两个 pair 不等的情况下，执行了一次比较和一次拷贝。应该直接判断 pair_first == pair_second 就行了，其他的比较删除。
- 将 map 结构中的 _t_pair 改成 _pair_temp。_t_keycompare 改成 _bfun_keycompare, _t_valuecompare 改成 _bfun_valuecompare。
- 将 hashtable 中的 t_compare 改成 bfun_compare, t_hash 改成 ufun_hash。t_bucket 改成 vec_bucker。
- hashtable 代码优化，去掉重复地方。

第二十三章 libcstl 各版本特性比较

类型和功能	1.0	2.0	2.1	说明
deque_t	√	√		

list_t	✓	✓	
vector_t	✓	✓	
slist_t	✓	✓	
set_t	✓	✓	
multiset_t	✓	✓	
map_t	✓	✓	更新了默认的数据比较规则。
multimap_t	✓	✓	更新了默认的数据比较规则。
hash_set_t	✓	✓	更新了默认的哈希函数。
hash_multiset_t	✓	✓	更新了默认的哈希函数。
hash_map_t	✓	✓	更新了默认的数据比较规则和默认的哈希函数。
hash_multimap_t	✓	✓	更新了默认的数据比较规则和默认的哈希函数。
priority_queue_t	✓	✓	
queue_t	✓	✓	
stack_t	✓	✓	
多种 iterator_t	✓	✓	
range_t		✓	一种表示数据范围的类型。
数值算法	✓	✓	
通用算法	✓	✓	
针对于 C 语言内建类型的函数	✓	✓	
针对于 libcstl 内建类型的函数		✓	增加了针对容器以及工具类型的函数和谓词。
string_t	✓	✓	
pair_t	✓	✓	更新了默认的比较规则。
bool_t	✓	✓	
支持 C style 字符串		✓	增加了对于 C style 字符串的支持。
支持用户自定义类型	✓	✓	通过类型注册机制完善了对于用户自定义类型的支持。
类型注册		✓	增加了类型注册和复制的功能。
支持 Linux	✓	✓	使用 GNU Autotools 工具来管理软件的编译和安装。
支持 Windows		✓	增加了 VS2005 和 VS2008 的工程。

建议使用的前缀：

类型	前缀
vector_t	vec

list_t	list
slist_t	slist
deque_t	deq
set_t	set
multiset_t	mset
map_t	map
multimap_t	mmap
hash_set_t	hset
hash_multiset_t	hmset
hash_map_t	hmap
hash_multimap_t	hmmap
queue_t	que
priority_queue_t	pq
stack_t	sk
iterator_t	it
range_t	r
pair_t	pair
bool_t	b
string_t	str
_byte_t	by
char*	s
binary_function_t	bfun
unary_function_t	ufun