<https://blog.csdn.net/kingsyj/article/details/8747933>

# 1 引言

众所周知，C++语言提供了大名鼎鼎的标准模板库（STL）作为C++语言下的编程利器受到无数青睐，而C语言下并没有提供类似的工具，使得C语言的开发变得尤为困难和专业化。Nesty框架的NCollection容器为C语言提供了一整套标准的、丰富的模板工具，用以弥补C语言在结构化编程上的弱势。之前发表的一篇文章是关于如何在C语言下进行面向对象编程的，主要介绍了NOOC的诸多特性。然而，NOOC真正的作用是为NCollection建立基础。本文还是以一些简单的代码作为例子，介绍NCollection的使用方法，通过阅读本节的内容，你将了解为什么NCollection可以大大地加快C语言下开发的效率。

# 2 迭代模型

作者所指的迭代模型，是指对谋一同类数据进行遍历时所使用的方法，从设计模式的角度来看，迭代模式甚至包含了访问者模式。以Nesty的开发为例，NCollection整套框架的设计都是围绕其迭代模型展开的，因为迭代模型涉及到是否能为所有容器提供一套统一的访问标准。在Nesty的容器框架下，迭代模型通常包含了两种基本形式：（1）基于索引的；（2）基于位置（Position）的。基于索引的迭代模式比较常用和易于理解，我们通常利用索引来遍历数组；基于位置的迭代模式会更加抽象一些，其工作方式类似于数组索引，在Nesty中通过定义一个Position数据结构来维持元素在各自容器中的某个“位置”。为了方便对比，下例例举STL容器中的迭代器，以及Nesty容器定义的索引迭代模式及Position迭代模式的代码：

示例（1），索引迭代模式：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // incase MyList is a valid NList<NINT> instance
2. **for** (NINT Idx = 0; Idx < MyList.Len(); Idx++) {
3. // extract the element
4. NINT Val = MyList.GetIdx(Idx); // or MyList[Idx]
5. }

示例（2），Position迭代模式：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **for** (NPosition Pos = MyList.FirstPos(); Pos; Pos = MyList.Next(Pos)) {
2. // extract the element
3. NINT Val = MyList.GetPos(Pos); // or MyList(Pos)
4. }

示例（3），STL迭代器模式：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

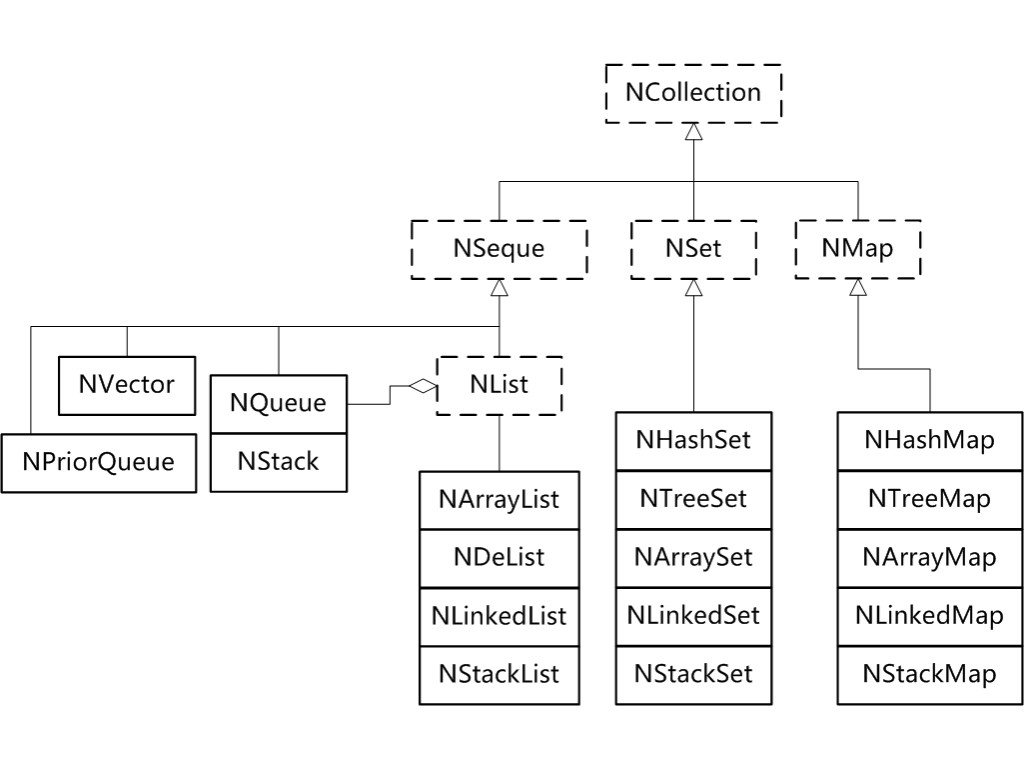
1. // incase my\_list is a valid std::list instance
2. **for** (std::list<**int**>::iterator it = my\_list.begin(); it != my\_list.end(); it++) {
3. // extract the element
4. **int** value = \*it;
5. }

通过观察上例，你会发觉Nesty容器所提供的Position迭代模式更加简洁，而相比之下STL容器迭代器是一个子包含的对象，每次使用都需要从list类型中解压，其语法是std::list<int>::iterator，而Position迭代模式通过调用容器的通用接口FirstPos返回一个位置信息，并不断通过Next及Prev来移动位置，并通过GetPos来从相应位置获取数据。因此，Position仅代表一个抽象的位置，好比索引（Index）代表的是一个具有具体偏移值的位置一样。

注意：为了方便阐述，上述代码使用了Nesty容器中的C++版本，C语言版本在编码结构上与其一致，但代码略有不同。

# 3 NCollection容器框架

NCollection是在NOOC的基础上开发的以面向对象为基础的一套容器库，提供约20种容器工具，其类型覆盖所有通用数据结构，包括向量（NVector），列表（NList），集合（NSet），关联表（NMap）等，以下是NCollection框架的全景UML图，其中虚线框代表的是抽象类，实线框代表的是实现类：



在上图的整个框架中，有五个最为重要的容器接口：

## NCollection

NCollection是框架中最顶层的基类，所有容器都派生自NCollection。对于外部而言，NCollection中所定义的操作是只读的，即只提供一个Push操作来单个插入元素；由于所有容器都有清空所有元素的操作，NCollection也提供了Empty操作用于批量删除元素。

## NSeque

单词Seque是Sequence的缩写，代表序列。序列能很好地模拟先进先出（FIFO）及先进后出（FILO）等行为的数据结构。NSeque派生自接口NCollection并提供了一个Pop操作用于一次从容器中弹出一个元素，Pop操作会根据Seque的类型表现出不同的行为。例如，NSeque接口指向的是一个NQueue，则Pop表现为从队列最前端弹出元素，如果NSeque指向的是一个NStack，则Pop表现为从栈最顶端弹出元素，以此类推。NSeque的实现类有NVector，NPriorQueue，NQueue，NStack。

## NList

NList是所有列表数据结构的抽象基类。列表具有队列或栈的属性，因此是从NSeque派生而来，但除此之外，列表通常还能以高效的速度（通常表现为常数级操作）从前/后、中间插入/删除元素。列表是最为常用的数据结构，尽管有时候向量（NVector）同样可以充当列表，但列表专门针对从内部插入/删除元素作了优化。对于NSeque的接口行为，NList及其所有实现类都表现为队列（FIFO）的属性。NList的实现类有NArrayList，NDeList（双端列表），NLinkedList，NStackList（栈表，行为和链表相同，但对内存碎片进行了优化）。

## NSet

NSet代表的是集合。如果仅从元素存储来看，集合非常类似于列表，都是用于存储单个元素，而集合却对元素有快速查找的能力。List的查找总是基于遍历的，其时间复杂度为O(N)，而集合根据规则的不同，能够提供比列表优越得多的查找速度，例如对于哈希而言，其平均查找的效率通常为常数，而对于红黑树而言，其平均查找时间通常为O(N Log N)。当然，集合在查找速度上的优化是要付出代价的，例如其遍历、插入/删除速度不及列表，而且也无法维持列表元素的先后顺序。NSet的实现类有NHashSet，NTreeSet，NArraySet，NLinkedSet，NStackSet。

## NMap

关联表（Map）是以键（Key）和值（Value）构成的二元关系集合。如同NSet一样，NMap对键拥有快速查找的能力。为了保证容器在组成上的统一，NMap继承了NCollection的接口，其接口表现为只对Key进行的操作。NMap的实现类包括：NHashMap，NTreeMap，NArrayMap，NLinkedMap，NStackMap。

# 4 在C语言中使用Nesty容器

C语言是一门面向过程的编程语言，然而，通过对设计方法的规范化，依然可以在C语言使用C++中类似封装，基于对象，甚至面向对象（NOOC）等的编程技术。对象包括了数据及方法的概念，因此在Nesty C的所有类型也按照类似的规则来定义。以NVector为例，NVector的数据是全大写的对象NVECTOR，而NVector的操作是一组以NVector单词开头的函数的集合，如NVectorNew，NVectorAdd，NVectorDel等，以下代码片段演示如何在C语言下进行容器编程：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // 创建NINT类型的容器对象
2. NVECTOR Vec = NVectorNew(NINT);
3. // 插入元素
4. NINT InsertValue = 3;
5. NVectorAdd(Vec, InsertValue);
6. // 删除元素
7. NINT DeleteValue = 3;
8. NVectorDel(Vec, DeleteValue);
9. // 遍历元素
10. NPOSITION Pos = NVectorFirstPos(Vec);
11. **for** (; Pos; Pos = NVectorNext(Vec, Pos)) {
12. NINT Val = NVectorGetPos(Vec, Pos, NINT);
13. }
14. // 不过对于向量，可以直接使用索引迭代模式
15. NINT Idx = 0;
16. **for** (; Idx < Vec->Len; Idx++) {
17. NINT Val = NvectorGetIdx(Vec, Pos, NINT);
18. }

# 5 C语言与泛型编程

泛型是从C++模板引入的概念，然而C语言下实现泛型可以利用无类型指针（void \*）。而类型数据从二进制的角度来探讨，不外只是一块有指定大小的内存块；由于任何类型的指针的都可以隐式转换为一个void \*的地址，因此利用一个数据大小值，以及一个指向该数据头部的无类型地址（void \*）即可以表达任何类型的泛型。

但是定义一个类型光知道类型的大小是不行的，类型必须具有其独特的行为，从类型数据的生存期来看，类型应该具备以下三个最为基本的行为（操作）：创建（Create），拷贝（Copy），销毁（Destroy）。以动态字符串为例，字符串的大小代表该字符串占用了多少个字符字节。在字符串创建时，需要将其初始化为指向一个有效字符串的地址，当需要拷贝字符串时，需要将字符串数据进行逐字符拷贝，当字符串不再被使用时，应该释放其占用的内存，并将字符串指针初始化为0。

因此，在Nesty的框架中，类型的大小，创建，拷贝，销毁这4个属性构成了该类型的特征签名，这一概念和C++类的构造函数，复制拷贝操作符，析构函数等是对应的。

# 6 泛型与Type Class

根据上一节的介绍，Nesty引入了Type Class的概念来支持C语言的泛型操作，在程序代码中由NTYPE\_CLASS数据结构给出相关定义，Type Class指定了某类型相关的特性及操作，以下便是NTYPE\_CLASS的定义：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** **struct** tagNTYPE\_CLASS
2. {
3. // type identifier
4. NINT                        TypeSize;
5. NPfnCreate                  FnCreate;
6. NPfnCopy                    FnCopy;
7. NPfnDestroy                 FnDestroy;
9. // type functions
10. NPfnMatch                   FnMatch;
11. NPfnCompare                 FnCompare;
12. NPfnHash                    FnHash;
13. NPfnPrint                   FnPrint;
15. // template functions
16. NPfnSwap                    FnSwap;
17. } NTYPE\_CLASS;

定义那些以NPfn\*开头的定义是函数指针的定义，因此结构中的数据FnCreate，FnCopy，FnDestroy等是函数指针。当需要为类型创建容器时，需要为容器提供该类型的NTYPE\_CLASS定义，以便告知容器根据这些操作来初始化/拷贝/删除元素。以下例的自定义数据为例，为了使我们的容器能够支持MYDATA，则需要为MYDATA的容器填充一个NTYPE\_CLASS数据，并传递给容器的创建函数。

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** tagMYDATA MYDATA;
2. **struct** tagMYDATA {
3. NINT    Value;
4. };
6. // define Create behavior
7. **void** CreateMyData(NVOID \* InThis, **const** NVOID \* InOther) {
8. MYDATA  \* This = (MYDATA  \*)InThis;
9. MYDATA  \* Other = (MYDATA  \*)Other;
10. **if** (Other) {
11. This->Value = Other->Value;
12. }
13. **else** {
14. This->Value = 0;
15. }
16. }
18. // define Copy behavior
19. **void** CopyMyData(NVOID \* InThis, **const** NVOID \* InOther) {
20. MYDATA  \* This = (MYDATA  \*)InThis;
21. MYDATA  \* Other = (MYDATA  \*)Other;
22. NASSERT(Other); // source MUST not NULL!!!
23. This->Value = Other->Value;
24. }
26. // define Destroy behavior
27. **void** DestroyMyData(NVOID \* InThis) {
28. MYDATA  \* This = (MYDATA  \*)InThis;
29. This->Value = 0;
30. }

为了方便阐述，以下先给出了NPfnCreate，NPfnCopy，及NPfnDestroy接口的定义：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** (\*NPfnCreate)(NVOID \* InThis, **const** NVOID \* InOther);
2. **typedef** (\*NPfnCopy)(NVOID \* InThis, **const** NVOID \* InOther);
3. **typedef** (\*NPfnDestroy)(NVOID \* InThis);

其中Create的接口能够对类型进行默认构造和拷贝构造（当InOther为NULL）。当上例给出了MYDATA的这些行为函数，则可以利用他们来初始化/拷贝/销毁该数据的实例：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. MYDATA MyData, OtherData;
2. // create MYDATA by default construct
3. CreateMyData(&MyData, NULL);
4. // after create, MyData.Value will have the default value of 0
5. NASSERT(MyData.Value == 0);
6. // create MYDATA by copy construct, incase OtherData.Value == 3
7. CreateMyData(&MyData, &OtherData);
8. // after create, MyData.Value will have the initialized value of 3
9. NASSERT(MyData.Value == 3);
11. // copy MYDATA, incase OtherData.Value == 5
12. CopyMyData(&MyData, &OtherData);
13. // after copy, MyData.Value will have the updated value of 5
14. NASSERT(MyData.Value == 5);
16. // destruct MYDATA
17. DestroyMyData(&MyData);
18. // after destroy, MyData.Value will have the cleanup value of 0
19. NASSERT(MyData.Value == 0);

如果仅仅是为了初始化MyData根本不需要调用给出的这些函数，但这里演示的是容器内部如何通过给定的这些操作来更新数据。另外，Type Class还定义了一些其他操作，如NPfnMatch用于比较两个数据是否相等，相当重载C++的==操作符，NPfnCompare用于数据做大小比较，相当C++中重载<操作符，NPfnHash返回该类型哈希值，NPfnPrint用于打印数据（主要为方便调试），NPfnSwap用于交换数据（在对容器排序时用到，一般情况下用户不需要提供NPfnSwap的定义，系统会自动创建）。以下给出了其余接口的定义及根据本例比较常用的实现：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** (\*NPfnCompare)(**const** NVOID \* InThis, **const** NVOID \* InOther);
2. **typedef** (\*NPfnMatch)(**const** NVOID \* InThis, **const** NVOID \* InOther);
3. **typedef** (\*NPfnHash)(**const** NVOID \* InThis);
4. **typedef** (\*NPfnPrint)(**const** NVOID \* InThis, NCHAR \* InBuf, NINT InLen);
5. **typedef** (\*NPfnSwap)(NVOID \* InThis, NVOID \* InOther);

根据本例的MYDATA，以下是最为常用的实现：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NBOOL MatchMyData(**const** NVOID \* InThis, **const** NVOID \* InOther) {
2. MYDATA \* This = (MYDATA \*)InThis;
3. MYDATA \* Other = (MYDATA \*)Other;
4. **return** (NBOOL)(This->Value == Other->Value);
5. }
7. NBOOL CompareMyData(**const** NVOID \* InThis, **const** NVOID \* InOther) {
8. MYDATA \* This = (MYDATA \*)InThis;
9. MYDATA \* Other = (MYDATA \*)Other;
10. **return** (NBOOL)(This->Value < Other->Value);
11. }
13. NUINT HashMyData(**const** NVOID \* InThis) {
14. MYDATA \* This = (MYDATA \*)InThis;
15. **return** (NUINT)This->Value;
16. }
18. NINT PrintMyData(**const** NVOID \* InThis, NCHAR \* InBuf, NINT InLen) {
19. MYDATA \* This = (MYDATA \*)InThis;
20. **return** NSnprintf(InBuf, InLen, \_t("%d"), This->Value);
21. }
23. **void** SwapMyData(NVOID \* InThis, NVOID \* InOther) {
24. MYDATA \* This = (MYDATA \*)InThis;
25. MYDATA \* Other = (MYDATA \*)Other;
26. MYDATA Tmp = \*This;
27. \*This = \*Other;
28. \*Other = Tmp;
29. }

为了结束本小节，将以上面定义的操作为例填充一个NTYPE\_CLASS数据结构，并创建相应的容器：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NTYPE\_CLASS TypeClass = { **sizeof**(MYDATA), CreateMyData, CopyMyData, DestroyMyData,
2. MatchMyData, CompareMyData, HashMyData, PrintMyData, SwapMyData };
3. NVECTOR Vec = NVectorNewCustom(&TypeClass, /\* more parameters has been omitted \*/);
4. NINT Idx = 0;
5. **for** (; Idx < 8; Idx++) {
6. MYDATA Tmp;
7. Tmp.Value = Idx;
8. NVectorAdd(Vec, Tmp);
9. }
10. // print elements
11. NVectorPrint(Vec, NSystemOut());
12. // Outputs:
13. // [8](0, 1, 2, 3, 4, 5, 6, 7)

看到这里你或许不禁要问，如果每使用一种类型都需要定义这么多类型操作函数，还需要填充NTYPE\_CLASS，在使用上岂不是相当麻烦？从表面上看，是的。然而，Nesty框架已经考虑到了这些问题，并且系统已经为大量常用类型预定义了Type Class，这些类型包括一些基本类型，如NINT, NUINT, NFLOAT, NDOUBLE等，大部分情况下，你不需要理会Type Class，只需要简单地使用这些预定义的类型；然而，对于像MYDATA这种用户自定义的数据结构，你还是需要提供一个Type Class，不过Nesty已经为你定制好了非常方便的工具，在最简单的情况下，你只需要两行代码即可以定义一个Type Class，之后会有单独的小节介绍如何使用这个功能。

# 7 创建容器

有了上一节的内容作为基础，本节将详细介绍容器的创建方法。由于所有的容器类型都是NOBJECT对象，其创建使用的是new规则，即从堆上分配一块内存并初始化，因此当结束使用时仍然需要调用NRELEASE接口释放对象：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NVECTOR Vec = NVectorNew(NINT);
2. // before exit
3. NRELEASE(Vec);

在上例中NVectorNew是一个宏定义，接受一个类型作为参数，该宏的实现会利用宏的黏合操作获取NINT预定义的TypeClass，其语法类似于：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NTYPE\_CLASS \* TypeClass = TypeClassNINT();
2. NVECTOR Vec = NVectorNewCustom(TypeClass, /\* more paramter has been omitted \*/);

正如上一节所述，Nesty已经为各种常用的类型预定义了TypeClass，因此当你需要使用这些类型来创建容器时，是不需要再手动填充NTYPE\_CLASS结构的，只需要像上例一样给容器创建函数提供类型定义（如NINT）。NCollection中的所有容器都提供了类似NVectorNew这样的接口，因此你完全不需要担心创建容器会过于麻烦。另外，所有系统预定义的TypeClass都位于ntype\_class.h头文件中，在此不再一一列举。因此，当你需要使用这些基本类型去创建容器时，应该优先考虑Nesty中定义的类型，例如你需要一个无符号整型的Vector，则应该使用NUINT去创建，而不应该应该使用unsigned int，并且像NVectorNew(unsigned int)这样的语法是非法的，如：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // create a unsigned int container
2. NVECTOR Vec = NVectorNew(NUINT);
3. // but, grammar like the following is invalid!
4. // NVectorNew(unsigned int);

对于Vector而言，NVectorNewCustom是最原始的接口，而NVectorNew及\_NVectorNew是为了方便使用而进行的封装，并且基本上能够满足需求；NCollection的所有容器都是基于这一规范设计的，因此当你创建容器时，尽量不要考虑NVectorNewCustom；但是，为了研究我们还是会细究NVectorNewCustom的各个参数，通过了解了NVectorNewCustom的接口，你将明白Nesty的容器到底是如何工作的；以下便是其定义：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NVECTOR  NVectorNewCustom(**const** NTYPE\_CLASS \* InTypeClass, NBOOL InAutoShrinkable, **const** NALLOC\_CLASS \* InAllocClass);

在参数中，InTypeClass是容器元素类型的TypeClass描述（参考上节），InAutoShrinkable稍后介绍，InAllocClass用于告诉容器，其内部元素所使用的内存是如何分配和销毁的，NALLOC\_CLASS的定义如下：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** **struct** tagNALLOC\_CLASS
2. {
3. NPfnMalloc              FnMalloc;
4. NPfnRealloc             FnRealloc;
5. NPfnFree                FnFree;
6. NPfnCalculateStackSize  FnCalculateStackSize;
7. } NALLOC\_CLASS;

下面是NPfnMalloc，NPfnRealloc，NPfnFree及NPfnCalculateStackSize的定义：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** NVOID \*     (\*NPfnMalloc)   (NSIZE\_T InSize);
2. **typedef** NVOID \*     (\*NPfnRealloc)  (NVOID \* InPtr, NSIZE\_T InSize);
3. **typedef** **void**        (\*NPfnFree) (NVOID \* InPtr);
4. **typedef** NINT        (\*NPfnCalculateStackSize)(NINT InStackNum, NINT InStackMax);

与TypeClass的规则类似，TypeClass用于描述类型数据是如何创建和销毁的，而AllocClass则描述容器内部的内存是如何被管理的；当容器内部需要为元素分配/释放内存时，都会调用FnMalloc/FnRealloc/FnFree所指向的函数；这三者最为常见的实现是，直接调用操作系统的内存分配策略，如下所示：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NVOID \* DefaultMalloc(NSIZE\_T InSize)                   { **return** malloc(InSize); }
2. NVOID \* DefaultAlloc(NVOID \* InPtr, NSIZE\_T InSize)     { **return** realloc(InPtr, InSize); }
3. **void** DefaultFree(NVOID \* InPtr)                         { free(InPtr); }

以上看来，为容器提供AllocClass看似是多余的，其实不然，当我们需要对容器的分配策略进行优化，例如，将容器的分配重定向到用户自定义的内存池，以便达到专一的目的时，NALLOC\_CLASS作为一个有用接口将给容器定制提供充分便利。

接下来，需要讨论的是容器的栈的属性。栈（Stack）顾名思义，代表的是一堆紧靠的东西的意思；Vector的实现，使用的是动态数组的策略，即Vector的内部会维护一个活动的内存堆栈，该堆栈总会预留一定数量的元素个数，当需要连续添加新元素时，会从预留的元素中分配，而不是重新分配一大块内存；只有当元素个数超出了预留的空间时，才会进行重新分配；相反，当容器元素个数变得很稀少，而预留的空间又太大时，为了不浪费内存，活动栈也会执行重新分配，并释放多余的元素空间。总之，容器内部的活动栈会在容器执行插入/删除操作时，根据当前元素的个数进行动态调整（扩张/收缩）。InAutoShrinkable参数默认为True，当为False时，容器在删除元素时将不会根据收缩内部的活动栈。这一属性对于某些静态容器相当有用，例如，假设有一个容器专门用于存储某些静态对象，这些对象一旦创建将不会被销毁，这时可以将容器的栈空间预设为经过统计得来的峰值，并进行一次分配，则可以省去了来回分配/释放内存的麻烦。

NPfnCalculateStackSize用于控制活动栈的预留策略，活动栈包括两个基本属性，当前元素个数（StackNum），及最大元素个数（StackMax）；一般情况下StackNum总是小于StackMax，当达到或者超过StackMax时，则表明栈空间不足，需要对栈重新分配，以容纳更多元素。这时候容器内部会调用FnCalculateStackSize所指向的策略函数，重新计算栈的上限值（StackMax）。Nesty容器所提供的默认策略是预留约25%的元素个数，假设当前容器的元素个数是100个，并且已经达到了上限，需要重新分配，则重新计算后的StackMax值为125（预留25%）；当然，用户随时可以修改该策略，例如总是预留约2倍的元素，可以通过修改FnCalculateStackSize所指向的策略来实现。

值得注意的是，只有那些具有活动栈属性的容器才会提供InAutoShrinkable标志及FnCalculateStackSize才会发挥作用，对于那些无法提供栈属性的容器（例如NLinkedList等），则上述的参数将被忽略掉。具有活动栈属性的容器有：NVector，NPriorQueue，NArrayList，NDeList，NStackList，NArraySet，NStackSet，NArrayMap，NStackMap。

# 8 添加/删除及访问元素

往容器中添加元素，可以通过Push和Add方法，Push与Pop构成堆栈相互对应的操作；然而，当容器不作为堆栈使用时，为了区分这些操作的意思，大部分容器都定义了相关的Add及Del操作用于添加/删除元素，实际上Push和Add的行为是一样的，只不过其代表的意义不同而已。下面以NArrayList为例，演示了如何往容器添加元素：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NARRAY\_LIST List = NArrayListNew(NINT);
2. NINT Val = 0;
3. NArrayListAdd(List, Val);
4. Val = 1;
5. NArrayListAdd(List, Val);
6. NArrayListPrint(List, NSystemOut());
7. // Outputs:
8. // [2](0, 1)

观察上面的例子，两次对NArrayListAdd的调用，都需要初始化一个临时变量Val，并将Val传递给NArrayListAdd，为什么需要这样做呢？原因是NArrayListAdd是一个宏定义，而实际产生作用的是带前下划线的函数定义\_NArrayListAdd，\_NArrayListAdd接受的数据实际上是一个无符号的类型指针，其接口的声明如下所示：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** **void** NELEMENT\_T;
2. NPOSITION \_NArrayListAdd(NARRAY\_LIST InObj, **const** NELEMENT\_T \* InElement);

在前面的小节曾探讨过C语言的泛型编程是通过无符号类型指针去泛化所有的类型，尽管之前在创建NArrayList的实例的时候，指明了使用NINT类型，但在添加元素的过程中，NArrayList实际上并不知道该类型，NArrayList唯一接受的是一个无符号的类型指针，其可能指向任何类型的数据，但该数据具体的行为是在创建NArrayList容器对象的时候通过传递进来的NTYPE\_CLASS数据结构来描述的。因此，在插入元素时，容器收到的是一个无符号类型数据的地址，并通过其指定的TypeClass的Create操作来初始化数据；删除元素时，容器依然会将存储在内部的元素当做一个无符号的类型对待，并且通过调用TypeClass的Destroy操作来销毁数据，以此类推。

Nesty泛型的概念是一个比较难以接受的东西，其工作原理完全不同于C++的模板；C++模板会在编译阶段根据实际类型“实例化”相关模板，并且不同类型都会产生一份代码的拷贝，而Nesty泛型所定义的操作永远只有唯一一份，并且通过无类型void及Type Class来泛化类型的实例。

当明白了Nesty泛型的工作原理后，你便了解为什么需要预先初始化一份临时数据并用于插入/删除，因为这个临时数据能够产生一个有效的无符号地址，并将该地址的数据作为插入元素的数据传递给容器内部的Type Class的Create操作。为了便于理解这一概念，下面提供了一个分解的操作：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NINT Val = 0;
2. \_NArrayListAdd(List, &Val);

因此，你无法将一个字面常量传递给NArrayListAdd，以下代码将会产生错误，提示字面量无法进行地址转换：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // invalid grammar, you MUST have a initialized varable, but NOT a literal
2. // NArrayListAdd(List, 0);

删除元素的操作与插入元素的原理是相同的，同样需要初始化一个临时数据：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NARRAY\_LIST List = NArrayListNew(NINT);
2. NINT Idx = 0;
3. **for** (; Idx < 8; Idx++) {
4. NArrayListAdd(List, Idx);
5. }
6. NArrayListPrint(List, NSystemOut());
7. // Outputs:
8. // [8](0, 1, 2, 3, 4, 5, 6, 7)
9. NINT ValToDel = 3;
10. NArrayListDel(List, ValToDel);
11. NArrayListPrint(List, NSystemOut());
12. // Outputs:
13. // [7](0, 1, 2, 4, 5, 6, 7)

对元素的访问也遵循同样的原理，获取元素数据的接口实际上返回的也是一个无符号类型的地址，用户需要通过强制类型转换将该地址转换为实际类型的地址，以\_NArrayListGetIdx为例，继续上面的示例代码：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // definition of \_NArrayListGetIdx
2. NELEMENT\_T \*    \_NArrayListGetIdx(NARRAY\_LIST InObj, NINT InIdx)
4. // incase List is a valide instance of NARRAY\_LIST of type NINT
5. NArrayListPrint(List, NSystemOut());
6. // Outputs:
7. // [8](0, 1, 2, 3, 4, 5, 6, 7)
8. NINT Val = \*(NINT \*)\_NArrayListGetIdx(List, 3);
9. NASSERT(Val == 3);

然而，带前置下划线的\_NArrayListGetIdx是一个不太好用的接口，因为每次调用都要执行类型转换，因此NArrayList提供了更为方便的工具，如下例所示：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NINT Val = NArrayListGetIdx(List, 3, NINT);
2. NASSERT(Val == 3);
3. NINT \* ValPtr = &NArrayListGetIdx(List, 6, NINT);
4. NASSERT(\*ValPtr == 6);

不过，你需要注意的是，类似于NArrayListGetIdx / Pos等的接口的最后一个参数必须与你创建容器时所填写的类型参数一致，否则将导致错误的转换。

# 9 基于接口的容器编程

从前面介绍NCollection框架的小节我们了解到，NCollection提供了几个通用的接口，这些接口和实现类是通过NOOC的继承关系实现的，意味着我们可以使用任一继承链上的接口的方法来操作实现类，下面的例子演示当我们需要使用NArrayList时，可以通过NList接口的方法，因为NArrayList是继承自NList的，这样可以增加很多灵活性；例如，我们可以在创建接口时将NArrayList更换为别的列表而程序的其他部分不受影响：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NLIST List = (NLIST)NArrayListNew(NINT);
2. NINT Idx = 0;
3. **for** (; Idx < 8; Idx++) {
4. NListAdd(List, Idx);
5. }
6. NListPrint(List, NSystemOut());
7. // Outputs:
8. // [8](0, 1, 2, 3, 4, 5, 6, 7)

由于NCollection是所有容器的接口，通过NCollection可以实现某些泛型的算法和功能，例如下列算法，将接受任何类型的容器，并将元素单例化并逐个拷贝到另一个容器中：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **void** CopyUniqueInt(NCOLLECTION InDst, **const** NCOLLECTION InSrc) {
2. NPOSITION Pos = NCollectionFirstPos(InSrc);
3. **for** (; Pos; Pos = NCollectionNext(InSrc, Pos)) {
4. NINT Val = NCollectionGetPos(InSrc, Pos, NINT);
5. **if** (NCollectionFindPos(InDst, Val) == 0) {
6. NCollectionPush(InDst, Val);
7. }
8. }
9. }

# 10 遍历元素

我们在第2小节的时候已经讨论过了迭代模型，Position模式作为Nesty容器的标准模式为所有类型的容器提供了一致的的迭代接口，其意义在于为实现泛型算法提供支持。如果你对Position模式并不适应，NList接口还提供了更为直观的Index模式，利用Index模式进行迭代类似于遍历数组元素，但对某些类型的容器，Index模式在效率上会大打折扣。

## Position/Index模式

为了更好理解Position模式和Index模式，下面的例子用Position模式分别以前向及后向迭代列表元素：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NLIST List = (NLIST)NLinkedListNew(NINT);
2. NPOSITION Pos = NULL;
3. NINT Idx = 0;
5. // push elements
6. **for** (; Idx < 8; Idx++) {
7. NListAdd(List, Idx);
8. }
10. // forward iteration
11. **for** (Pos = NListFirstPos(List); Pos; Pos = NListNext(List, Pos)) {
12. NINT Val = NListGetPos(List, Pos, NINT);
13. NPrintf(\_t("%d, "), Val);
14. }
15. // Outputs:
16. // 0, 1, 2, 3, 4, 5, 6, 7,
18. // backward iteration
19. **for** (Pos = NListLastPos(List); Pos; Pos = NListPrev(List, Pos)) {
20. NINT Val = NListGetPos(List, Pos, NINT);
21. NPrintf(\_t("%d, "), Val);
22. }
23. // Outputs:
24. // 7, 6, 5, 4, 3, 2, 1, 0,
26. // index iteration
27. **for** (Idx = 0; Idx < NListLen(List); Idx++) {
28. NINT Val = NListGetIdx(List, Idx, NINT);
29. NPrintf(\_t("%d, "), Val);
30. }
31. // Outputs:
32. // 0, 1, 2, 3, 4, 5, 6, 7,

如果是NVector，NArrayList及NDeList，使用Index模式将更加效率，原因在于这些容器都是基于动态数组的原理，即各个元素总是位于一块连续的内存块中；但是像NLinkedList及NStackList等其他一些容器，是基于链表的原理，即元素之间通过前向及后向指针进行链接的，各个元素在内存上的分布是分散的；但即便如此，链表中的各个元素还是维持了先后顺序，只不过在按Index迭代元素时，总是从列表的头部/尾部元素开始，并逐个元素移动到相应的索引位置上。因此，上例按索引迭代的代码，其实质远没有你表面上看到的这样简洁；然而庆幸的是，NLinkedList及NStackList得到了缓存的支持，如果仅仅是迭代元素，性能只是稍有损失；以上例中8个元素的链表为例，假设当前访问的索引位置是3，在第一次访问索引3时，链表依然要从头部开始逐个向前移动到第三个元素，并将索引3和对应节点缓存，假设下次访问索引4，由于4和已经缓存的索引3之间相差1，链表会直接从缓存的节点开始向前移动一个元素的位置，以此类推。当链表重新插入/删除元素时，缓存的索引将被清除；因此，当链表按照索引进行迭代时，真正造成性能瓶颈的时候，是在你迭代过程中删除/插入的时候。

## 迭代中删除元素

迭代容器是按Position逐个迭代，删除元素也是按Position逐个删除；需要注意的是，像NListDelPos这种类型的接口会返回删除后下一个元素的Position，其原型为：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NPOSITION NListDelPos(NLIST InObj, NPOSITION InPos);

这一返回值是专为迭代中删除元素而设计的，要解释其原理需要牵扯到不容器内部较为复杂的实现，你需要记住的是，当从正向迭代删除元素时，应该遵循以下的语法，如果观察比较细致的话，会发现C++的标准模板库的迭代器也使用了类似的原理：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NPOSITION Pos = NULL;
2. NListPrint(List, NSystemOut());
3. // Outputs:
4. // [8](0, 1, 2, 3, 4, 5, 6, 7)
6. **for** (Pos = NListFirstPos(List); Pos; ) {
7. NINT Val = NListGetPos(List, Pos, NINT);
8. **if** (Val > 1 && Val < 6) {
9. Pos = NListDelPos(List, Pos);
10. }
11. **else** {
12. Pos = NListNext(List, Pos);
13. }
14. }
15. NListPrint(List, NSystemOut());
16. // [4](0, 1, 6, 7)

你需要注意的是NListDelPos和NListNext的使用方法；如果是以后向的方式迭代删除元素，语法稍微简单一些，如下所示：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NPOSITION Pos = NULL;
2. NListPrint(List, NSystemOut());
3. // Outputs:
4. // [8](0, 1, 2, 3, 4, 5, 6, 7)
6. **for** (Pos = NListLastPos(List); Pos;) {
7. NINT Val = NListGetPos(List, Pos, NINT);
8. NPOSITION Prev = NListPrev(List, Pos);
9. **if** (Val > 1 && Val < 6) {
10. NListDelPos(List, Pos);
11. }
12. Pos = Prev;
13. }
15. NListPrint(List, NSystemOut());
16. // [4](0, 1, 6, 7)

NCollection容器通过引入Position的概念，抽象并统一了各个容器访问元素的规则，通过找到某一元素在容器中的Position并通过Next和Prev接口可以方便地前后移动移动元素，这种规则对于实现算法是非常有帮助的；然而，Position模式在迭代过程中删除元素时，规则复杂了些少，不过幸运的是，如果其他容器框架一样，Nesty容器一样提供了迭代器的接口，通过一下节的介绍，你将了解通过使用迭代器，可以大大简化遍历及删除元素的步骤。

# 11 使用迭代器

迭代器的接口是由对象NITERATOR提供的，下面通过几段简短的代码来演示其用法：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NLIST List = (NLIST)NLinkedListNew(NINT);
2. NITERATOR It = NULL;
3. NINT Vals[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
4. NListAddNum(List, Vals, NARRAY\_LEN(Vals));
6. It = NListIterator(List, NFALSE);
7. **while** (NIteratorHasNext(It)) {
8. NINT Val = NIteratorNext(It, NINT);
9. NPrintf(\_t("%d, "), Val);
10. **if** (Val > 1 && Val < 6) {
11. NIteratorRemove(It);
12. }
13. }
14. // Outputs:
15. // 0, 1, 2, 3, 4, 5, 6, 7
16. NRELEASE(It);
17. NListPrint(List, NSystemOut());
18. // Outputs:
19. // [4](0, 1, 6, 7)

当你调用NListIterator方法时，将会返回一个新创建的NITERATOR对象，此时的Iterator处于一个“无效位置”的状态，用户需要通过NIteratorHasNext及NIteratorNext操作将迭代移动到一个有效位置并获取数据。如果你曾经使用过Java则一定很清楚其迭代器接口的工作原理，这里所使用的迭代器与Java是一样的：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NLIST List = (NLIST)NLinkedListNew(NINT);
2. NITERATOR It = NListIterator(List, NFALSE);
3. NPOSITION Pos = NIteratorPosition(It);
4. NASSERT(Pos == NULL);
5. **if** (NIteratorHasNext(It)) {
6. NIteratorNext(It, NINT);
7. Pos = NIteratorPosition(It);
8. NASSERT(Pos != NULL);
9. }

由于NITERATOR是一个NOOC对象，在每次使用完后，都应该通过NRELEASE接口立即将其释放，否则迭代器将长期持有其宿主容器的一个引用计数。  
  
迭代器的方法NIteratorHasNext及NIteratorNext/\_NIteratorNext是配合使用的，只有通过NIteratorHasNext检测存在下一个元素的时候，才能调用NIteratorNext/\_NIteratorNext移动迭代器的位置，NIteratorNext/\_NIteratorNext在移动下一位置的同时返回当前的值，用户通过检测这个值判断是否要对当前元素执行删除操作，其语法如下所示：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NITERATOR It = NListIterator(List, NFALSE);
2. **while** (NIteratorHasNext(It)) {
3. NINT Val = NIteratorNext(It, NINT);
4. NBOOL ShouldRemove = /\* check current value \*/;
5. **if** (ShouldRemove) {
6. NIteratorRemove(It);
7. }
8. }
9. NRELEASE(It);

由于Remove操作总是发生在Next之后，因此你永远不需要像Position一样，关心Remove操作是否会影响下一个要遍历的元素；其代码的方式将更加简洁和易于理解。

迭代器同样可以用于键值二元对的关联表容器（NMap），当使用迭代器遍历Map时，Next操作总是返回值（Value）而不是键（Key），但是迭代器提供了另外的操作GetKey来获取当前遍历元素对的键；当迭代器作用于像列表（NList）或集合（NSet）这种一元容器时，Next操作和GetKey操作返回的都是相同的数值，其代码示例如下：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // for list or other single element containers
2. NList List = (NLIST)NArrayListNew(NINT);
4. NITERATOR Items = NListIterator(List, NFALSE);
5. **while** (NIteratorHasNext(Items)) {
6. NINT Val = NIteratorNext(Items, NINT);
7. NINT Key = NIteratorGetKey(Items, NINT);
8. NASSERT(Val == Key);
9. }
11. // for map the key-value pair element containers
12. NMAP Map = (NMAP)NHashMapNewMulti(NINT, NCHARS);
13. NINT Key = 0;
14. NCHAR \* Val = \_t("ABCD");
15. NMapAdd(Map, Key, Val); // push more 0 and "ABCD"
17. NITERATOR Pairs = NMapIterator(Map, NFALSE);
18. **while** (NIteratorHasNext(Pairs)) {
19. NCHAR \* Val = NIteratorNext(Pairs, NCHARS);
20. NINT Key = NIteratorGetKey(Pairs, NINT);
21. NASSERT(Key == 0);
22. NASSERT(NStrcmp(Val, \_t("ABCD") == 0);
23. }

# 12 字符串及容器

由于TypeClass接口所发挥的特殊作用，在Nesty容器接口中使用字符串元素显得十分方便，你只需要像创建其他类型一样使用NCHARS类型，如下所示：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NHASH\_MAP Map = NHashMap(NCHARS, NINT);
2. NCHAR \* Key = \_t("nesty");
3. NINT Val = 0;
4. NHashMapAdd(Map, Key, Val);
5. // Add more key and value ...

通过字符串去查找容器的语法也一样简单：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NCHAR \* Key = \_t("nesty");
2. NPOSITION Pos = NHashMapFindPos(Map, Key);
3. NINT Val = NHashMapGetValue(Map, Pos, NINT);

NCHARS符号其实只是NCHAR \*指针的另一个定义：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** **const** NCHAR \*   NCHARS;

在本节的第一个例子中，我们将临时变量Key作为一个参数传递给NHashMapAdd，但NHashMap容器并不会保存Key所指向的字符串的地址，而是通过NCHARS类型的Create操作为键值分配字符串大小的内存空间，并将Key字符串的内容拷贝到新分配的内存空间中；因此容器保存的是一个新分配的字符串对象，而不是Key所指向的静态字符串的内存地址；当容器通过Empty操作删除元素的时候，会再调用NCHARS的Destroy操作来释放从堆上分配的内存，因此无需主动去为字符串分配/释放内存。通过下面的例子说明容器中的键值与作为临时参数的Key指向的是不同的字符串地址：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NHASH\_MAP Map = NHashMap(NCHARS, NINT);
2. NCHAR \* Key = \_t("nesty");
3. NINT Val = 0;
4. NHashMapAdd(Map, Key, Val);
5. NPOSITION Pos = NHashMapFindPos(Map, Key);
6. **const** NCHAR \* KeyFromMap = NHashMapGetKey(Map, Pos, NCHARS);
7. NASSERT(Key != KeyFromMap);

如果你确实想自己去管理字符串内存，则不要使用NCHARS作为容器的创建类型，例如你可以使用NVOIDP，即让容器保存一个内存地址，或者使用更直接的NCHARP；虽然NCHARS跟NCHARP都是NCHAR \*的一个类型定义，但是其TypeClass的协议是不同的，使用NCHARS时容器会为你创建一个内存托管的字符串类型，但当使用NCHARP是，则容器会把该字符串视作一个纯粹的字符串指针，不会为你托管内存；下面是使用NCHARP创建容器时的代码，请注意其工作方式的不同：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NMAP\_MAP Map = NHashMapNew(NCHARP, NINT);
2. NCHAR \* Key = (NCHAR \*)NMalloc(**sizeof**(NCHAR \*) \* NStrlen(\_t("nesty")));
3. NINT Val = 0;
4. NStrcpy(Key, \_t("nesty"));
5. NHashMapAdd(Map, Key, Val);
6. // ...

由于容器使用的字符串的内存是通过你调用堆分配函数NMalloc进行分配，并将指针保存在容器的键元素中，因此当你清空容器元素之前，也需要从外部调用堆释放函数NFree来回收内存（容器本身不会为你释放内存）否则将引发泄漏：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NPOSITION Pos = NHashMapFirstPos(Map);
2. **for** (; Pos; Pos = NHashMapNext(Map, Pos)) {
3. **const** NCHAR \* Key = NHashMapGetKey(Map, Pos);
4. // release string memory
5. NFree(Key);
6. }
7. NHashMapEmpty(Map);
8. // ...

程序员手动地去管理内存是一件痛苦的事情，因此一般情况下，你只需要使用NCHARS类型来创建容器便足够了，NCHARS的TypeClass协议会为你管理字符串的内存。

最后一种创建字符串容器的方式是使用Nesty的NSTRING对象，NSTRING是Nesty框架提供的标准字符串接口，通过NSTRING可以方便的实现字符串连接，替换，查找等操作，但由于NSTRING是一个NOOC对象，其对象的创建和销毁遵循某些面向对象的规则，因此其过程是复杂且相对低效的；使用NSTRING来创建字符串容器将使程序简洁性和性能稍打折扣，一般情况下不推荐使用；不过如果你坚持使用，Nesty容器依然为NSTRING提供了相应的接口。另外，NSTRING容器使用的是NOOC对象的TypeClass规则，其步骤较为繁琐，后面会有单独的小节介绍如何创建NOOC对象容器：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NMAP\_MAP Map = NHashMapNew(NSTRING, NINT);
2. NSTRING Str = NStringNew(\_t("nesty"));
3. NINT Val = 0;
4. NHashMapAdd(Map, Str, Val);
5. NRELEASE(Str);

接下来示例如何在NSTRING容器中查找键值：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NSTRING Tmp = NStringNew(\_t("nesty"));
2. NPOSITION Pos = NHashMapFindPos(Map, Tmp);
3. // After done finding
4. NRELEASE(Tmp);

# 13 类型与Type Class

通过前面小节的介绍，Nesty已经预定义了部分常用数据类型的TypeClass，因此这些类型可以直接作为容器创建接口的参数，这些常用类型有：

**整型**：NBYTE，NUBYTE，NSHORT，NUSHORT，NINT，NUINT，NLONG，NULONG

**浮点型**：NFLOAT，NDOUBLE

**字符型**：NCHAR，NCHARP，NCHARS，NSTRING

及等等……

然而，对于某些用户自定义的类型（如第6节中的自定义类型MYDATA），为了能够让容器使用这些类型，程序员在定义数据的同时，也要定义相应的Type Class；Nesty的系统已经为你提供了非常方便的工具来达到目的。需要定义类型的TypeClass，需要两个步骤，首先在声明代码中（最好是与数据结构定义处于同一文件中）使用宏NTYPE\_CLASS\_DEC来声明TypeClass，然后在实现代码中提供相应的协议函数（如前面介绍的Create，Copy，Destroy的操作的函数），并用NTYPE\_CLASS\_IMP宏来绑定这些协议函数。

## 创建普通类型的TypeClass

当前所指的普通类型，是指由C的关键字struct或者typedef定义的类型，不包括NOOC对象类型，以MYDATA为例：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** tagMYDATA MYDATA;
2. **struct** tagMYDATA {
3. NINT    Value;
4. };
6. // declaration part
7. NTYPE\_CLASS\_DEC(MYDATA);
9. // implementation part
10. **void** CreateMyData(NVOID \* InThis, **const** NVOID \* InOther) {...}
11. **void** CopyMyData(NVOID \* InThis, **const** NVOID \* InOther) {...}
12. **void** DestroyMyData(NVOID \* InThis) { ... }
13. // ... and more actions, Match, Compare, Hash, Print
15. NTYPE\_CLASS\_IMP(MYDATA,
16. CreateMyData,
17. CopyMyData,
18. DestroyMyData,
19. MatchMyData,
20. CompareMyData,
21. HashMyData,
22. PrintMyData);
24. // After create the Type Class for MYDATA type,
25. // you can use 'MYDATA' as the parameter of containers
26. NLINKED\_LIST List = NLinkedListNew(MyData);
27. // ....

事实上提供TypeClass协议函数是一件非常烦人的事情，而NTYPE\_CLASS\_IMP另一个更加强大的功能是允许你提供缺省值，即传递NULL参数，这是NTYPE\_CLASS\_IMP将为你提供一个缺省实现，至于缺省实现的默认动作是什么将会稍后讨论。例如，如果你只想为MYDATA填写Create，Copy，和Destroy操作，则你可以按如下方式填写NTYPE\_CLASS\_IMP的参数：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NTYPE\_CLASS\_IMP(MYDATA,
2. CreateMyData,
3. CopyMyData,
4. DestroyMyData,
5. NULL,
6. NULL,
7. NULL,
8. NULL);

当然，你还可以为所有的TypeClass协议都提供缺省值，让系统为你提供默认实现：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NTYPE\_CLASS\_IMP(MYDATA,
2. NULL,
3. NULL,
4. NULL,
5. NULL,
6. NULL,
7. NULL,
8. NULL);

对于上述情况，你还可以使用另一个更加方便的宏，NTYPE\_CLASS\_IMP\_DEFAULT

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NTYPE\_CLASS\_IMP\_DEFAULT(MYDATA);

## TypeClass协议的缺省实现

当你使用宏NTYPE\_CLASS\_IMP并给相应的协议操作传递NULL参数时，Nesty将为这些操作提供缺省实现，下面列举出这些缺省实现的相关内容，继续以MYDATA为例：

1） **Create**会有条件地执行内存拷贝，当Create操作的InOther参数为空时，会通过NZeroMemory将内存清零，如：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **void** CreateDefaultMyData(NVOID \* InThis, **const** NVOID \* InOther) {
2. **if** (InOther) {
3. NMemcpy(InThis, InOther, **sizeof**(MYDATA));
4. }
5. **else** {
6. NZeroMemory(InThis, **sizeof**(MYDATA));
7. }
8. }

2） **Copy**执行简单的内存拷贝，如：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **void** CopyDefaultMyData(NVOID \* InThis, **const** NVOID \* InOther) {
2. NMemcpy(InThis, InOther, **sizeof**(MYDATA));
3. }

3） **Destroy**执行简单的内存清0，如：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **void** DestroyDefaultMyData(NVOID \* InThis, **const** NVOID \* InOther) {
2. NZeroMemory(InThis, **sizeof**(MYDATA));
3. }

4） **Match**，**Compare**和**Hash**等操作会执行相应的内存比较和哈希：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NBOOL MatchDefaultMyData(**const** NVOID \* InThis, **const** NVOID \* InOther) {
2. **return** (NBOOL)(**return** NMemcmp(InThis, InOther, **sizeof**(MYDATA)) == 0);
3. }
5. NBOOL CompareDefaultMyData(**const** NVOID \* InThis, **const** NVOID \* InOther) {
6. **return** (NBOOL)(**return** NMemcmp(InThis, InOther, **sizeof**(MYDATA)) < 0);
7. }
9. NUINT HashDefaultMyData(**const** NVOID \* InThis) {
10. **return** NMemhash(InThis, **sizeof**(MYDATA));
11. }

5） **Print**操作仅简单地打印This指针的地址：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NINT PrintDefaultMyData(**const** NVOID \* InThis, NCHAR \* InBuffer, NINT InLength)  {
2. **return** NSnprintf(InBuffer, InLength, \_t("%p"), InThis);
3. }

## NOOC对象类型Type Class

创建NOOC的对象类型的Type Class相当简单，只需要使用宏NTYPE\_CLASS\_IMP\_OBJECT，不需要实现及填写任何的协议操作，由于NOOC对象类型的NOBJECT接口包含了Clone，Comp，Equal，HashCode及ToString等操作，NTYPE\_CLASS\_IMP\_OBJECT实现会直接调用这些接口来实现TypeClass，因此NOOC对象是通过重载相应的接口来实现TypeClass的协议的；下面例子只简单演示如何为NOOC对象MYOBJ创建TypeClass：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NOBJECT\_PRED(MYOBJ);
2. NOBJECT\_DEC(MYOBJ, NOBJECT);
3. **struct** tagMYOBJ {
4. NOBJECT\_BASE(NOBJECT);
5. NINT Val;
6. }
8. // declaration part
9. NTYPE\_CLASS\_DEC(MYOBJ);
11. // implementation part
12. NOBJECT MyObjClone(**const** MYOBJ InObj) { ... }
13. NBOOL MyObjEqual(**const** MYOBJ InObj, **const** NOBJECT InOther) { ... }
14. NBOOL MyObjComp(**const** MYOBJ InObj, **const** NOBJECT InOther) { ... }
15. NUINT MyObjHashCode(**const** MYOBJ InObj) { ... }
16. NSTRING MyObjToString(**const** MYOBJ InObj) { ... }
18. NOBJECT\_IMP(MYOBJ, NOBJECT,
19. NCLONE\_BIND(MyObjClone)
20. NEQUAL\_BIND(MyObjEqual)
21. NHASHCODE\_BIND(MyObjHashCode)
22. NCOMP\_BIND(MyObjComp)
23. NTOSTRING\_BIND(MyObjToString)
24. );
26. NTYPE\_CLASS\_IMP\_OBJECT(MYOBJ);

NOOC对象的创建是一个相对复杂的过程，作者发表的另一篇文章《[C语言下的面向对象编程技术](http://blog.csdn.net/kingsyj/article/details/8744386)》提供了一部分参考，其连接为[点击打开链接](http://blog.csdn.net/kingsyj/article/details/8744386)。

# 14 容器与持有对象

NOOC对象是一个引用计数对象，其通过引用计数来实现实例之间的共享及垃圾回收等操作，每一个NOOC对象在通过NNEW创建时其原始计数都为，因此当使用完毕后，必须通过NRELEASE接口来释放计数，以便系统回收为其分配的动态内存。然而当该对象需要被共享（或者需要被其他对象所持有时），可以通过调用NACQUIRE接口来增加引用计数，以NSTRING对象为例：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NSTRING Str = NStringNew(\_t("hello Nesty"));
2. NASSERT(NCOUNTER(Str) == 1);
3. // Hold(Acquire) object reference
4. NSTRING NewRef = Str;
5. NACQUIRE(NewRef);
6. NASSERT(NCOUNTER(Str) == 2);
7. // After done working, you have to call equivalent numbers of NRELEASE to reclaim the object
8. NRELEASE(NewRef);
9. NRELEASE(Str);

通过引用计数可以很方便地实现对象间的共享，并节省内存；即当你需要在多个地方使用到同一对象时，只需要保有一个引用计数，并在退出时释放该计数，而不用为每个应用单独分配新的对象。

因此，容器对NOOC对象类型使用了同样的规则，当你以对象的方式来创建容器时，添加元素并不会导致容器为每个元素都克隆（clone）一个拷贝，而仅仅是持有该对象的引用，并且容器在删除元素时会自动将该引用释放。为此，本节将解释前面12（字符串与容器）小节中，当使用NSTRING对象创建容器时，为何必须在添加/查找元素后，将临时NSTRING对象释放：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NMAP\_MAP Map = NHashMapNew(NSTRING, NINT);
2. NSTRING Str = NStringNew(\_t("nesty"));
3. NINT Val = 0;
4. NHashMapAdd(Map, Str, Val);
5. NRELEASE(Str);

先回顾上面的这段代码，Str通过NStringNew来赋值时，该NSTRING对象的初始计数是1，当将该字符串对象传递给NHashMapAdd后，容器仅仅是保存了该对象地址的拷贝，并通过NACQUIRE持有该对象的一个计数；因此添加完毕后，Str对象的引用计数将变为2。而代码最后通过NRELEASE来将Str对象释放，其目的是使它的计数回复到1，即将该对象的持有权完全交给了HashMap，因为将来当你通过调用容器的Empty等操作来删除该元素时，容器将会自动帮你回收该对象。假如你在添加元素之后不释放Str对象的计数，即使将来HashMap删除了该元素，也仅仅使其计数变为1，但对象不会被释放，因此造成泄漏。

# 15 容器与元素分配

为了方便描述本节的内容，先回到之前MYDATA的定义，并在这里稍作修改：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. **typedef** **struct** tagMYDATA MYDATA;
2. **struct** tagMYDATA {
3. NBYTE   Data[32];
4. }

现在MYDATA变成了一个包含一块32字节大小的连续内存的数据结构，如果我们为其创建了TypeClass，然后创建容器并添加8个元素：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NVECTOR Vec = NVectorNew(MYDATA);
2. NINT Idx = 0;
3. **for** (; Idx < 8; Idx++) {
4. MYDATA Tmp;
5. NVectorAdd(Vec, Tmp);
6. }

则Vector会将这些元素的数据都保存在一块连续的内存中，且每个元素大小为sizeof(MYDATA) ，这与创建一个8个元素的静态数组的结构是类似的；容器会根据TypeClass中的TypeSize成员来分配并回收元素的内存。但有些时候，用户如果想单独管理各个元素的内存，则可以以指针方式来创建容器，如：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // NVOIDP is a typedef of void \*
2. NVECTOR Vec = NVectorNew(NVOIDP);
3. MYDATA \* Tmp = (MYDATA \*)NMalloc(**sizeof**(MYDATA));
4. NVectorAdd(Vec, Tmp);
5. // Add more elements...

当你以这种方式来创建容器时，容器管理的只是元素的指针，但不会为你管理该指针所指向的内存，因此当你删除元素时，还需要单独去回收每个元素所占用的内存：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NINT Idx = 0;
2. **for** (; Idx < Vec->Len; Idx++) {
3. NVOID \* Data = NVectorGetIdx(Vec, Idx, NVOIDP);
4. NFree(Data);
5. }
6. NVectorEmpty(Vec);

# 16 打印及调试容器

为了方便对容器进行调试，容器提供了一套规范的字符串化功能，但是字符串花需要TypeClass协议中Print操作的支持，即前提是某类型已经在其TypeClass中定义了Print操作。所有的容器都提供了相关的Print接口，如NArrayListPrint，NSetPrint等，这些Print接口都接受一个NSTREAM\_OUT的对象作为输出对象，NSystemOut()代表输出到系统的标准输出接口，通常为命令行控制台，但也可以输出到Buffer。以NINT类型的容器为例：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NARRAY\_LIST List = NArrayListNew(NINT);
2. // incase List has elements: 0, 1, 2, 3, 4, 5, 6, 7
3. NArrayListPrint(List, NSystemOut());

则通过调用NArrayListPrint，你会在命令行下看到以下输出：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. [8]( 0 1 2 3 4 5 6 7 )

方括号[ ]中的数值代表容器元素的个数，圆括号( )中的数值是各个元素的字符串化后的（打印）数值，以空格划分。元素的打印数值取决于你如何去编写相关类型的Print操作（详见13小节关于自定义类型TypeClass的介绍）。

另外，你还可以打印输出到一段Buffer，其做法是创建一个NSTRING\_BUFFER\_OUT的对象，NSTRING\_BUFFER\_OUT是NSTREAM\_OUT的实现，因此可以将该对传递给NArrayListPrint的InStream参数：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // String Buffer example:
2. NSTRING\_BUFFER Buffer = NStringBufferNew(4096);
3. NSTRING\_BUFFER\_OUT BufferOut = NStringBufferOutNew(Buffer);
4. NArrayListPrint(List, (NSTREAM\_OUT)BufferOut);
5. // Process buffer
6. NPrintf(Buffer->Chars);

对于某些比较复杂的数据结构，例如Set和Map，除了提供Print这个基于序列方式打印元素的方法外，还另外了一个功能更加强大的State方法，用于窥探数据结构内部的元素组织情况，以方便用户观察数据，并调整相应的键值函数。例如：

**[cpp]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NHASH\_SET Set = NHashSetNew(NINT);
2. NINT Idx = 0;
3. **for** (; Idx < 12; Idx++) {
4. NHashSetAdd(Set, Idx);
5. }
6. NHashSetState(Set, NSystemOut());
7. NRELEASE(Set);

通过State方法，你将能看到Set各元素的哈希分布状况：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. ----
2. State Map: Len = 12, HashSize = 8, Multiple = 0, Sorted = 0
3. ----
4. State Hash:
5. [0]( 0 8 )[2]
6. [1]( 1 9 )[2]
7. [2]( 2 10 )[2]
8. [3]( 3 11 )[2]
9. [4]( 4 )[1]
10. [5]( 5 )[1]
11. [6]( 6 )[1]
12. [7]( 7 )[1]
13. ----
14. Statistics:
15. Hash Chain = 8, Total Bucket = 12, Max Bucket = 2, Min Bucket = 1, Average Bucket = 1.500000

对于哈希表来说，这个功能是相当有用的；例如当你看到某部分哈希链上分布的元素十分密集，而其他哈希链的元素分布十分稀疏，则证明你当前使用的哈希函数很不均匀，在进行哈希插入时产生了大量的“碰撞”，导致效率低下；一旦发现这种情况，你应该重新考虑元素Hash操作中用到的算法， 或者考虑更换键的类型。

另外，对于二叉树数据结构，也同样可以利用State方法来窥探其各元素在树中的分配是否足够平衡：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

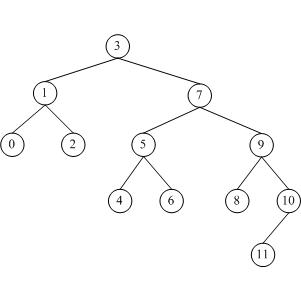
1. NTREE\_SET Set = NTreeSetNew(NINT);
2. NINT Idx = 0;
3. for (; Idx < 12; Idx++) {
4. NTreeSetAdd(Set, Idx);
5. }
6. NTreeSetState(Set, NSystemOut());
7. NRELEASE(Set);

其输出为：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. ----
2. State Map: Len = 12, Multiple = 0
3. ----
4. State Tree:
5. 3
6. .1
7. ..0
8. ..2
9. .7
10. ..5
11. ...4
12. ...6
13. ..9
14. ...8
15. ...10
16. ....11

在上面的输出中，点的数量代表的是当前叶节点的深度，该树的遍历采用的是先序遍历方法，当前打印格式是文本格式，你还要通过自顶向下的方法，将文本格式还原为树的视图格式；例如，根据上面的输出，还原出来的该树的视图为：



NTreeMap使用的是红黑树结构，由此可以观察到，红黑树的平衡仅仅是子树的局部平衡。

# 17 泛型算法

Nesty对泛型算法的支持采取了两种形式：（1）容器绑定的及（2）容器分离的。与容器绑定的算法，主要是考虑到数据结构的性质，例如对于一般的排序算法而言，像Vector，ArrayList等基于动态数组的数据结构，最高效的排序算法应当是快速排序，但是像LinkList等，其较为高效的排序则是归并排序；再举个例子，像列表旋转算法，基于数组的和基于链表的数据结构之间的实现的方法及效率也大为不同。NCollection容器集的大部分数据结构都提供了类似Sort的方法，如NVectorSort，NLinkedSetSort，NArrayMapSort等，只有少部分在算法上不允许排序的数据结构除外，例如NHashSet/Map，NTreeSet/Map等。另外，对于列表或向量来说，也提供了类似Rotate，Scroll及Reverse等等的算法接口，如NVectorRotate，NLinkedListScroll，NArrayListReverse等等，由于这些算法都与其容器的类型直接相关连，因此属于容器绑定的算法；尽管接口相同，但其内部实现会依据数据额结构的种类而有所/大为不同。下面以几个清晰的例子来掩饰如何使用这些算法。

对于排序而言，Sort方式将接受一个NPfnCompare的函数指针作为比较器，该函数的定义必须符合下面的格式（以NINT为例）：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // Compare two integer value with greater than comparation
2. NBOOL CompareNINT\_GT(const NVOID \* In1, const NVOID \* In2) {
3. return (NBOOL)(\*(const NINT \*)In1 > \*(const NINT \*)In2);
4. }

比较器的参数必须是void \*类型，因为之前已经就C语言泛型探讨过，void \* 可以作为一个通用接口来泛化所有类型，因此在函数实现部分，需要将void指针再强制转换为其实际类型，获取数据，比较并返回结果。一旦定义了比较器，则可以对列表进行常规排序：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NLINKED\_LIST List = NLinkedListNew(NINT);
2. NINT Idx = 0;
3. for (; Idx < 8; Idx++) {
4. NLinkedListAdd(List, Idx);
5. }
6. NLinkedListSort(List, CompareNINT\_GT);
7. NLinkedListPrint(List, NSystemOut());
8. // Outputs:
9. // [8](7, 6, 5, 4, 3, 2, 1, 0)

另外，你还可以对列表进行局部排序，例如将上面的例子修改为：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. // Sort elements between index 2 and index 2 + 4
2. NLinkedListSortNum(List, 2, 4, CompareNINT\_GT);
3. NLinkedListPrint(List, NSystemOut());
4. // Outputs:
5. // [8](0, 1, 5, 4, 3, 2, 6, 7)

由于Nesty容器集是基于接口设计的，例如NCollection，NList，NSet，NMap等这些都是容器的相关接口，接口所定义的操作对于所有实现类其行为是相同的，因此还可以针对各个接口层提供其他泛型算法，由于这些算法不与特定的容器绑定，因此属于容器分离的。容器分离算法主要是基于接口间某些共通的操作而实现的，例如，由于NCollection接口支持容器的Position迭代模式，基于这一模式可以实现很多有用的操作，例如Copy，Find等，下面举几个简单的例子。

拷贝，下面的例子在两个在结构上完全无关的容器间实现拷贝，因为他们都实现了NCollection的接口：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NHASH\_SET Set = NHashSetNew(NINT);
2. NVECTOR Vec = NVectorNew(NINT);
3. NCollections\_Copy((NCOLLECTION)Set, (const NCOLLECTION)Vec);

添加，下面的例子按指定次数重复地往序列中添加元素：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NLIST List = (NLIST)NArrayListNew(NINT);
2. NINT Val = 0;
3. NCollections\_PushFirst((NCOLLECTION)List, &Val, 20);

# 18 Nesty容器的局限性

在C++的模板泛型中，模板的实例化是通过编译器在编译时进行，并且会为每个模板参数的类型编译一个单独的类，因此模板类具有静态属性；但是，由于Nesty容器是通过void\*来对类型进行泛化的，然而任何类型的地址都可以自动转换为void\*，这将导致类型识别的问题；假设现在创建了下面两个容器：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NARRAY\_LIST ListOfNINT = NArrayListNew(NINT);
2. NARRAY\_LIST ListOfFLOAT = NArrayListNew(NFLOAT);

最荒谬的情况是，用户如果故意将一个NINT类型的参数，传递给一个NFLOAT类型的容器时，编译器根本无法识别这种错误：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NINT Val = 10;
2. NArrayListAdd(ListOfFLOAT, Val);

当然，int和float的长度的是相同的，因此在拷贝数据时顶多会引起数据错误，如果当连着的数据长度不一致时，极有可能引发崩溃，而这种错误只能通过检查代码才能找到，因此在使用时必须十分谨慎。

另外，由于NCollection容器集是基于NOOC的对象实现的，而对象实例其实是一个指针定义，而在C中指针是可以任意转换的，例如程序员可以恶作剧地将一个List容器对象强制转换为一个Set/Map然后再传递给Set/Map的方法；然而这不能完全责怪作者，因为C语言本身就是一门比较灵活的（或者说有点肆无忌惮的）编程语言；因此，在使用Nesty进行C语言开发的时候程序员一定要相当小心谨慎。

Nesty容器在性能上落后于C++的模板容器，其中主要原因是，TypeClass的操作都是基于函数指针实现的，因此在调用一个函数指针的函数时，执行的是代码跳转，而不像很多C++代码一样直接通过内联的方式来实施优化；但据作者测试比对来看，可以肯定的是，C与C++容器之间的差别仅仅是在代码内联上，在算法上不存在差异。

# 19 在C++中使用容器

对于大部分人来说（特别是那些习惯在面向对象的环境中开发的），C泛型及TypeClass是一个难以接受的东西，为此Nesty还针对C++模板进行了封装，但其接口和定义和C语言的容器是一模一样的，实际上C++的版本和C语言的版本都同属一套容器。下面是一些简单的例子：

**[plain]** [view plain](https://blog.csdn.net/kingsyj/article/details/8747933) [copy](https://blog.csdn.net/kingsyj/article/details/8747933)

1. NArrayList<NINT> List;
2. List.Add(3);
3. for (NINT Idx = 0; Idx < List.Len(); Idx++) {
4. NINT Val = List[Idx];
5. }
6. List.Sort<NGreater<NINT> >();

由于C++的容器仅仅是对C语言的代码进行了封装，并非重新开发，因此无法发挥C++语言的优势；在测试过程中，其效率相对低于STL；但是作者已经意识到了这个问题，目前正着力于对C++版本的容器进行全面重构和重新开发，力图在性能上能够赶上STL。如果有对此感兴趣的朋友，请多留意Nesty的进展。

# 20 结束语

NCollection容器集以其丰富的阵容构成了Nesty框架中相对独立的模块，其作用是为C语言提供一套规范的动态数据结构，以简化在C下从事算法开发的难度。NCollection容器涵盖的内容十分庞大，本文仅从实用角度筛选了部分比较常用的，且具有理论性质的内容加以讲解，为的是让对此感兴趣的朋友能够了解Nesty容器的设计思想及理念。相信从事过算法开发的朋友都知道，要想在C语言环境下开发动态数据结构是相对困难且专业的工作，大多数情况下，程序员需要单独去管理各个元素的内存，这不但增加了编程的难度，也容易引发错误。Nesty容器通过void\*类型及TypeClass协议泛化了所有类型的操作，因此能够达“到同一套编程接口对所有类型都适用”的目的。因此Nesty容器的代码可以高度地被重用，程序员只需要根据容器所持有类型的特性，实现相应的TypeClass操作。而Nesty容器基于NOOC对象模型实现了部分面向对象的功能，因此才能使容器基于接口编程；而更令人激动的是，通过在容器的接口层，引入Position迭代模式，统一了容器对元素实施遍历，插入，查找等的操作，为实现泛型算法打下良好的基础。容器接口的定义是精简的，有好的，通过阅读本文的代码，你将发现，在C语言下面使用Nesty容器将不比在C++中使用模板容器需要编写过多的代码。然而，Nesty容器依然有其局限性，由于C语言泛型不像C++模板能够在编译时实例化模板类，C泛型更多依赖对void\*及函数指针的操作，因此引发了类型识别及性能等问题；但是，只要从事开发的程序员正确使用，Nesty容器就C语言开发来说依然是相当高效且易用的。另外，本文仅仅是介绍少部分的功能，更多的功能需要感兴趣的朋友从代码及实例中去探讨，Nesty的工程有大量测试代码来供你学习和研究。如果你对此感兴趣并有所想法，请不妨留下你的宝贵意见，作者热切盼望得到你的反馈。Nesty是跨平台的，开源的软件，其下载站点为[点击打开链接](http://sourceforge.net/projects/nesty/)