北太天元 SDK 开发文档 v3.3

北太振寰（重庆）科技有限公司

最近一次更新：2023 年 5 月 9 日

1 介绍

北太天元给开发者提供了底层数据的访问接口，这使得开发者能够使用 C/C++ 等底层语言编写可与北太天元交互的可执行程序。使用开发者工具箱（SDK）可以使得开发工作变得更加灵活，用底层语言编写的程序在运行时也会有更高的效率。

北太天元提供的开发者工具有两种主要的使用场景：插件研发和 BEX 文件生成。在后续版本中还会引入更加通用的场景。

1.1 插件研发

开发者可以使用 C/C++ 等底层语言编写独立功能模块，并将该程序以插件的模式整合到北太天元中。一个插件主要提供如下的功能：

- 为软件增加功能函数或提供额外脚本；
- 为软件增加或托管自定义数据结构；
- 修改软件部分 UI 元素（待支持）；

在北太天元中使用插件中的函数基本流程为

```
% 载入名为 name 的插件，后面是可选参数
ierr = load_plugin('name', ...);
% 使用插件中的函数 fun1, fun2, ...
A = fun1(m, n);
B = fun2(A, x);
% 使用插件函数 init 创建自定义对象 C，并用作为函数 fun3 的参数调用
C = init(a, b, c);  % C 是插件定义的外部对象
X = fun3(C);

% ...

```

```
% 卸载插件，之后就不出现插件函数的调用了
% 和插件相关的不透明对象也被自动清理干净
ierr = unload_plugin('name');
```

1.2 BEX 文件生成

用户可以使用一种更简单的方式在北太天元中运行 C/C++ 函数，使用这个功能需要借助 BEX 文件。相关内容请阅读第 2.2 节。

2 使用方法

2.1 插件开发

2.1.1 插件目录结构

我们约定所有插件都存放在 `plugins` 中，该目录下每个目录中含有一个完整的插件。例如一个名为 `foo` 的插件，应该存放如下的文件：

- `plugins/foo`
  - `main.[dll|so|dylib]` 库的主要文件，必须包含（后缀和系统相关）。
  - `main.sig` 由北太天元团队提供的签名文件，可选。某些情况下北太天元只会载入经过验证的插件。
  - `main.conf` 库的配置文件，可选。若存在，北太天元在载入时会读取相关内容。
  - `scripts/` 目录，可选，包含插件提供的辅助脚本。若存在，北太天元在载入插件时会自动将该目录添加到脚本搜索路径，卸载时自动将该目录从搜索路径中移除。
  - （其它文件）

需要注意，若插件使用了第三方动态库，建议也将第三方动态库打包放到和 main 库相同的目录，在 Linux 等系统中还需要设置运行时库的路径。

2.1.2 插件的编译链接

以 Linux 为例，编译插件时，需要链接的库有

- `libbex.so`
- `libbaltam_core.so`

在这里推荐使用北太天元自带的 `bex` 编译器（v3.1+），开发者无需手动链接这些库，见第 2.3 节。

2.1.3 插件内容

我们规定插件的主程序需要包含以下内容。这里注意所有的函数都是使用 C 语言的命名方式，即声明时需要前面给定 `extern "C"`，见头文件 `bex.h`。

头文件

- 使用 C 语言时，需要包含 `bex/bex.h`
- 使用 C++ 语言时，需要包含 `bex/bex.hpp`

函数列表 必须实现，原型为

```
bexfun_info_t *bxPluginFunctions();
```

该函数返回一个 `bexfun_info_t` 类型结构体的指针，指向函数列表（是一个 C 数组）中的第一个元素。`bexfun_info_t` 的定义如下：

```
typedef struct {
    const char * name; // 函数名
    bexfun_t ptr;      // 实际的函数指针
    const char * help; // 帮助文字（可以为 NULL）
} bexfun_info_t;
```

`bexfun_t` 是插件提供函数的统一原型，详细信息见后文。一种可能的定义函数列表的方式为：

```
static bexfun_info_t flist [] = {
    {"fun1", fun1, help1},
    {"fun2", fun2, help2},
    {"fun3", fun3, NULL}, // 帮助位置写 NULL 表示该命令没有帮助
    {"", NULL, NULL}; // 表示列表结尾
}

bexfun_info_t *bxPluginFunction(){
    return flist;
}
```

我们约定：**列表的最后一个元素的函数指针必须为空指针，以便北太天元确定列表的结尾。**

目前北太天元在载入多个插件时，假定插件提供的函数名互不相同。如果载入某插件发现其它插件已经定义了同名函数，则 `load_plugin` 会失败并返回非零值。为了尽可能减少函数冲突，我们建议开发者在函数列表中使用命名空间，或者是定义重载函数，见第 2.1.6 节。

初始化外部库句柄 可选，原型为

```
int bxPluginInitLib(void *hdl);
```

- hdl libbex 的句柄，用于提供核心库信息以供插件内部使用；
- 返回值：一个 int 类型的整数，正常结束返回 0，当出现异常时返回非零值。

北太天元在调用 load_plugin(name, ...) 时会检查 bxPluginInitLib 是否有定义，若有定义则调用该初始化外部库句柄。当初始化句柄返回非 0 值时，load_plugin 也返回一个非零值，此时载入插件失败，北太天元会在命令行中打印错误信息。

初始化外部库句柄的主要使用场合是动态载入插件依赖的核心库，获取函数地址。该函数先于 bxPluginInit 执行。当插件没有采用传统的编译 – 链接方式生成时，一些 API 需要动态载入，这时候就需要开发者实现 bxPluginInitLib。当插件采用常规的编译 – 链接方式生成时，无需实现该函数。

初始化句柄 可选，原型为

```
int bxPluginInit(int nrhs, const bxArray *prhs[]);
```

- nrhs 输入的参数个数，实际是 load_plugin 函数的输入参数减 1；
- prhs 输入参数指针，这里 bxArray 是北太天元 SDK 中一个不透明的对象，用于存储北太天元管理的变量；
- 返回值：一个 int 类型的整数，正常结束返回 0，当出现异常时返回非零值。

北太天元在调用 load_plugin(name, ...) 时会在检查 bxPluginInit 是否有定义，若有定义则调用该初始化句柄。当初始化句柄返回非 0 值时，load_plugin 也返回一个非零值，此时载入插件失败，北太天元会在命令行中打印错误信息。

初始化句柄的主要使用场合是插件载入时的资源预分配，以及**用户自定义数据结构**的添加。北太天元在托管这些数据类型时，必须知道其相关的信息，详见第 2.1.4 节。该函数执行于 bxPluginInitLib 之后（如果有），这是因为添加自定义数据结构的 API 是在前者中载入的。

析构句柄 可选，原型为

```
int bxPluginFini();
```

- 该函数不需要参数。
- 返回值：一个 int 类型的整数，正常结束返回 0，当出现异常时返回非零值。

北太天元在调用 unload_plugin(name, ...) 时检查 bxPluginFini 是否有定义，若有定义则调用该析构句柄。当析构句柄返回非 0 值时，unload_plugin 也返回一个非零值，此时北太天元已经完成主要函数库的关闭以及内存释放，会在屏幕上打印警告信息。

析构句柄主要使用场合是释放初始化句柄中分配的内存。由于北太天元已经知道插件的信息，在卸载时会从内核中自动删除插件函数，自定义对象，无需开发者手动操作。

插件函数 插件提供的函数的类型为 bexfun_t，原型是

```
void (*bexfun_t)(int nlhs, bxArray *plhs[], int nrhs, const bxArray *prhs
    []);
```

- nlhs 输出参数个数，不包含 ans 变量（见注意事项）；
- plhs 输出参数指针数组，初始值均为 NULL；
- nrhs 输入参数个数；
- prhs 输入参数指针数组；
- 返回值：插件函数没有返回值。

假设插件中定义了 foo 这个函数，命令行中使用

```
[x, y] = foo(a, b, c);
```

则实际传递给插件的参数为

```
foo(2, [NULL, NULL], 3, [pa, pb, pc]);
```

有关如何和北太天元 SDK 接口交互请参考第 3 节。

注意事项

- bxPluginFunctions 返回的列表中包含的函数必须全部定义，否则调用 load_plugin 会失败，插件不会成功载入；
- 北太天元规定所有的插件函数名称必须互不相同，且不能与内置函数名称相同。北太天元在加载插件时会检查函数定义是否有重复，并在发生冲突时报错。如果想定义多个同名函数，需要使用插件重载函数功能，见第 2.1.6 节；
- 北太天元允许插件对部分内置算符进行重载，详见第 2.1.5 节；
- 北太天元约定插件函数正常退出时，plhs 数组中所有的指针均已经被赋值。若在调用结束后 plhs 数组中仍然有元素为空指针，北太天元认为插件函数执行出现错误，在命令行中打印相关错误信息；

- 当使用 C++ 时，不建议插件函数中直接抛出 std::exception 对象，由于 ABI 的问题，北太天元试图捕捉这些异常时可能会失败。建议使用 bxErrMsgTxt 函数来处理函数异常。
- 比较特殊的是 plhs[0] 这个值。当 nlhs = 0 时，plhs[0] 仍然可以正常使用并赋值（也可以不使用）。若 nlhs = 0 时对 plhs[0] 赋值，则赋值的结果会传递到 ans 变量中。如果若 nlhs = 0 时不对 plhs[0] 赋值，那么北太天元不会产生 ans 变量。换言之，nlhs 的含义是**用户实际给出的输出参数个数**。

使用命名空间 为了使得插件编写更灵活，开发者可以在定义函数列表时使用命名空间来区分来自本插件的函数和其它函数。为此在函数列表中函数名前添加 <namespace>:: 前缀即可。其中 <namespace> 可以取为任意名称。例如

```
static bexfun_info_t flist [] = {
    {"my_tools::fun1", fun1, help1},
    {"", NULL, NULL}; // 表示列表结尾
}
```

函数 fun1 定义在命名空间 my_tools 下。用户在软件中如果想使用命名空间中的函数，同样加入相应的命名空间前缀即可。

```
Y = my_tools::fun1(X);
```

需要注意：命名空间 builtin 为软件保留，用于放置内置函数，不能定义为插件的命名空间。

2.1.4 托管自定义数据类型

北太天元支持开发者定义 C/C++ 数据类型并托管到内核中。以下分两种情形：

C++ 插件中所有需要托管的 C++ 类都必须以 public 的方式继承基类 extern_obj_base，该基类位于命名空间 baltam 中。假设需要托管的类名为 derived，开发者需要实现**复制函数**和**析构函数**，原型如下

```
// 产生本对象的一个副本，返回副本的指针
extern_obj_base* derived::dup() const;
// 析构函数，释放一切开发者自行分配的内存
derived::~derived();
```

复制函数用于实现北太天元命令行中形如 B = A 的赋值语句，其中 A 是一个托管的不透明对象。析构函数用于不透明对象的空间释放。

实现 dup() 函数时，推荐的方式是先实现自定义数据类型的复制构造函数，然后在 dup() 中调用复制构造函数。确保在复制时也将基类 extern_obj_base 中的相关数据也复制，否则插件运行时可能会有意外的错误。例如：

```
derived::derived(const derived& other) : extern_obj_base(other){
    // 实际定义
}
extern_obj_base* derived::dup() const {
    return new derived(*this);
}
```

开发者可以选择实现**字符串转化函数**，原型如下

```
// 将本对象转化为字符串
std::string derived::to_string() const;
```

该函数在命令行显示对象内容时可能会有用。如果不实现，北太天元会显示空字符串。

开发者可以选择实现**类名函数**，原型如下

```
// 返回该类类名
std::string derived::classname() const;
```

该函数在命令行显示对象内容时可能会有用。如不实现，北太天元会显示默认值 extern object。

此外，自定义的类必须包含访问属性为 public 的一个静态变量 static int ID，并进行初始化。北太天元内核会用到这个变量，不需要开发者额外进行操作。

一个简单的例子为

```
using namespace baltam;
struct derived : public extern_obj_base {
    int a = 0;
    double x = 0;
    extern_obj_base *dup() const;
    ~derived();
    static int ID;
};

// 初始化静态变量
int derived::ID = 0;

// 定义复制
extern_obj_base *derived::dup() const{
```

```
    return new derived(*this);
}

// 定义析构
derived::~derived(){}
```

定义好自定义类型后，需要在 load_plugin 中调用类型添加的命令。

```
template <class T>
int bxAddCXXClass<T>(const std::string& name);
```

- T 模板参数，要添加的类名；
- name C++ 字符串，必须为插件的名称；
- 返回值：一个 int 类型的整数表示北太天元自动分配的类 ID，一般不会用到。添加失败时返回 -1。

例子：

```
int bxPluginInit(int, const bxArray *[]){
    bxAddCXXClass<derived>("test");
    return 0;
}
```

有关如何与自定义数据类型交互，请参考第 3 节。

C 结构体 插件中所有需要托管的 C 类型数据理论上可以定义为任意类型，但实际使用时大多使用结构体。假设需要托管的数据类型为 derived，开发者需要实现**复制函数**和**析构函数**，原型如下

```
// 产生本数据类型的一个副本，返回副本的指针
void* (*cstruct_copy_t)(const void *);
// 析构函数，释放一切开发者自行分配的内存
void (*cstruct_delete_t)(void*);
```

复制函数用于实现北太天元命令行中形如 B = A 的赋值语句，其中 A 是一个托管的不透明对象。析构函数用于不透明对象的空间释放。实际使用时，传入传出的数据类型实际为 derive *，但为了接口统一定义成了 void *。**注意：析构函数一定要包含释放右端指针参数本身的内容，因为北太天元无法得知该指针指向的地址是如何分配内存的。**

一个简单的例子为

```c
/**
 * @brief my_complex 自定义复数数组，
 * 它的实部和虚部是分开放置的。
 */
typedef struct {
    double *real;
    double *imag;
    baSize N;
} my_complex;

// 定义复制
void *my_complex_dup(const void *x){
    const my_complex *X = (const my_complex*)x;
    my_complex *ret = (my_complex*)malloc(sizeof(my_complex));
    ret->N = X->N;

    // 创建
    ret->real = (double*)malloc(ret->N * sizeof(double));
    ret->imag = (double*)malloc(ret->N * sizeof(double));

    // 复制
    memcpy(ret->real, X->real, X->N * sizeof(double));
    memcpy(ret->imag, X->imag, X->N * sizeof(double));

    return ret;
}

// 定义析构
void my_complex_destroy(void *x){
    my_complex *X = (my_complex*)x;
    free(X->real); X->real = NULL;
    free(X->imag); X->imag = NULL;
    // 这里的 free(X) 是必要的
    // 软件无法得知原始的 x 是如何分配内存的，
    // 因此交给开发者处理
    free(X);
}
```

定义好自定义数据类型后，需要在 load_plugin 中调用类型添加的命令。

```
int bxRegisterCStruct(
    const char *name,
    cstruct_copy_t cpy,
    cstruct_delete_t del);
```

- name C++ 字符串常量，必须为插件的名称；
- cpy 复制函数的指针（不可以为 NULL）；
- del 析构函数的指针（不可以为 NULL）；
- 返回值：一个 int 类型的整数表示北太天元自动分配的 CStruct ID，这个 ID 在后续的函数调用中要使用，因此需要记录下来（例如使用全局变量）。添加失败时返回 -1。

例子：

```
/**
 * @brief 初始化 sid，记录 cstruct 信息
 */
int my_sid = 0;

/**
 * @brief bxPluginInit 的定义（可选）
 *
 * bxPluginInit(args...) 会在调用 load_plugin(name, args...)
 * 时调用，目的是进行一些初始化工作。
 *
 * 这里的初始化工作是向内核添加 my_complex 这个结构体
 */

int bxPluginInit(int nrhs, const bxArray *prhs[]){
    my_sid = bxRegisterCStruct("my_complex", my_complex_dup,
    my_complex_destroy);
    return 0;
}
```

2.1.5 重载内置运算符

北太天元中允许被重载的算符见运算符对照表。插件通过注册的方式来完成对算符的重载。假设插件要重载双目加法算符 +，使得它可以支持插件自定义的外部对象 Foo，那么

代码只需要增加如下片段:

```
// 用于实现重载 + 的声明
// 函数名称任意，函数头仍然是 bexfun_t 类型
void my_add(int, bxArray*[], int, const bxArray*[]);

int bxPluginInit(int, const bxArray*[]){
    bxAddCXXClass<Foo>(PLUGIN_NAME);
    // 向内核注册重载的算符
    bxRegisterOperator(PLUGIN_NAME, "+", Foo::ID, my_add);
    return 0;
}

// 实际实现，这里从略
void my_add(int, bxArray*[], int, const bxArray*[]){}
```

通过在 bxPluginInit 中增加注册算符的语句来达到重载的效果，以上例子表示向内核注册一个双目加法的重载算符，它与类型 ID 是 Foo::ID 的对象绑定，实际由 my_add 函数实现。注意，注册重载算符的操作一定需要在添加类型之后，这是因为只有先添加了外部对象类型，Foo::ID 的值才有意义。

参数类型的 ID 必须为外部对象的 ID，bxRegisterOperator 的作用实际是为一个外部对象绑定了一个重载运算符。假设重载的运算符为 +，那么当北太天元执行 a + b 且 a 对象重载了加法时，它实际会调用 a 所注册的重载函数。

注意事项 从 BEX API v3.2 起，算符的重载规则和含义发生了变化，更细致的说明可以参考第 4.5 节。

2.1.6 定义重载函数

北太天元支持在插件中定义重载函数，即定义名称相同但执行操作不同的若干函数。特别地，插件可以定义和内置函数同名的函数，例如插件可以重载 disp 函数以显示插件自定义数据类型信息，但软件内置的 disp 仍然不受影响。通过使用重载机制，插件开发者可以更加灵活地控制北太天元调用函数的过程。插件重载函数功能在 BEX API v3.3 引入。

插件重载函数的函数原型和普通插件函数相同，如在 point 插件中的 show_point 函数

```
void show_point(int nlhs, bxArray *[], int nrhs, const bxArray *prhs[]){
    if (nlhs > 0 || nrhs != 1){
        bxReturnOverloadFailure();
        return;
    }
```

```
    Point *bv = bxGetExtObj<Point>(prhs[0]);
    if (bv == nullptr){
        bxReturnOverloadFailure();
        return;
    }
    bxPrintf(bv->to_string().c_str());
}
```

与普通插件函数不同，插件重载函数需要根据输入参数和输出参数判断该重载是否是可用的，如果判断不可用，需要使用 bxReturnOverloadFailure 函数立即结束本重载函数的运行，这样，北太天元就会尝试下一个可能的重载函数。在以上例子中，show_point 函数先对输入参数进行检查，且当输入参数仅有 1 个且为 Point 对象的时候才会正常执行，否则会提示本重载失效。

实现插件重载函数后，还需使用 bxOverloadFunction 将重载函数注册到北太天元，这样北太天元在调用函数时，会将该函数添加到待测试的重载函数列表中。该命令一般需要添加到 bxPluginInit 函数中：

```
int bxPluginInit(int, const bxArray*[]){
    // ...
    // 其它初始化

    // 调用 bxOverloadFunction, 重载函数 disp
    bxOverloadFunction(PLUGIN_NAME, "disp", show_point);
    return 0;
}
```

以上代码段表示将 show_point 注册为函数 disp 的一个可能重载，北太天元在运行 disp 函数时会检查所有的重载函数，并找到能成功运行的那一个，如果所有重载均失败，北太天元会执行内置函数 disp。在这里我们建议在添加重载函数的同时，也在插件函数列表中以名空间的形式添加该函数，例如 point::disp，这样就可以提供直接调用该重载函数（而不经过其它重载判断）的功能了。

显然，同一个函数名可能有来自不同插件的各类重载，还可能是普通的插件函数或脚本函数。北太天元执行函数的顺序为：

1. 同名 M 脚本或 M 函数，函数句柄；
2. 普通插件函数，即定义在 bxPluginFunctions 列表中的函数；
3. 插件重载函数；
4. 内置函数；

当高优先级类型的函数未找到时，北太天元会寻找下一优先级的函数。如果已经找到了相应函数，但执行出现错误，北太天元直接从执行中返回，**不会**到下一优先级函数中寻找。但对于重载的函数，在因为参数不满足重载要求而发生退出（调用了 `bxReturnOverloadFailure`）的条件下，北太天元会继续在其余重载函数或内置函数中寻找定义。若任意一个重载的函数正常退出，北太天元会判定重载函数调用成功并结束函数调用流程，若所有可能的重载均失败，则北太天元会调用内置函数。

2.1.7 例子

所有例子可在 `plugins` 中找到源码。目前包括：

- `vector`：包装 `std::vector`，演示插件的基本用法，定义插件函数，托管自定义数据类型；
- `point`：演示如何在插件中重载内置算符以及重载函数；

2.2 BEX 文件

2.2.1 用法

用户允许编译 BEX 文件（Baltamatica EXecutable），主要应用场合是可以在北太天元中调用使用 C/C++ 的函数。BEX 文件中的函数本质上和插件函数相同，但使用前无需手动进行载入。北太天元会自动载入处于 `path` 中以及当前工作目录中的 BEX 文件。

BEX 文件的后缀和系统相关，目前支持的有

- Windows 64 位：`.bexw64`
- Linux 64 位：`.bexa64`
- MacOS 64 位：`.bexmaci64`

不同系统的 BEX 文件不能混用。

编写 BEX 文件源码时，使用的头文件仍然为 `bex.h` 或 `bex.hpp`。但只需要实现一个函数入口，原型为

```
void bexFunction(int nlhs, bxArray *plhs[], int nrhs, const bxArray *prhs);
```

`bexFunction` 的原型与插件函数完全相同。

2.2.2 例子

下面是一个简单的例子，使用北太天元的 C 接口生成一个指定大小的矩阵。假设源文件名为 `create.c`。

```c
#include "bex/bex.h"

/**
* @brief A = create(m, n);
* 使用 C 语言接口创建一个 m x n 的 double 类型矩阵。
*/

void bexFunction(int nlhs, bxArray *plhs[], int nrhs, const bxArray *prhs
    []){
    if (nlhs != 1){
        bxErrMsgTxt("one output argument needed.");
        return;
    }

    if (nrhs != 2){
        bxErrMsgTxt("two input arguments are needed.");
        return;
    }

    baSize m = *bxGetDoubles(prhs[0]);
    baSize n = *bxGetDoubles(prhs[1]);

    plhs[0] = bxCreateDoubleMatrix(m, n, bxREAL);
    return;
}
```

编译命令（在北太天元命令行执行）：

```
bex "create.c"
```

即可在当前目录生成编译好的 BEX 文件。以 Linux 为例，生成的文件为 `create.bexa64`。随后可以像使用普通函数一样调用 `create` 函数（即函数名同文件名）：

```
A = create(3, 2);
```

2.2.3 注意事项

- 由于北太天元刷新的机制，新编译的 BEX 文件如果处于当前目录不会被立即载入，此时可以重新进入当前目录进行刷新。

- 考虑到执行效率，BEX 文件被北太天元载入后，会一直存在于系统内存中。在 BEX 载入的状态下，如果对文件重新编译那么北太天元仍然使用的是旧版文件（在 Windows 系统下甚至无法对正在打开的文件进行更新）。若需要暂时将 BEX 从内存中卸载，可以使用 `clear_bex <name>`。例如：

```
clear_bex create
```

下一次调用 `bex_create` 时，北太天元会自动从原位置重新加载 BEX 文件。

2.3 bex 编译器

`bex` 编译器从 BEX v3.1 引入，用于辅助进行 BEX 文件和插件的编译。其好处在于开发者只需要在系统中安装底层编译器，无需在编译时手动输入和北太天元相关的编译选项。目前 `bex` 仅支持 GCC，Linux 用户可直接在系统安装，Windows 用户可以选择安装 MinGW，例如 w64devkit，MacOS 用户可以选择配置 Xcode。使用前请确保 `gcc/g++` 等工具在系统的 PATH 上。

`bex` 编译器提供两种使用方式：命令行模式以及函数模式。在命令行模式下，可以像系统命令一样使用，例如：

```
bex create.c
```

即可编译 BEX 文件。若要编译插件请给定 `-plugin` 选项，如：

```
bex -plugin main.cpp
```

`bex` 会自动根据输入的文件识别应该调用哪一种语言的编译器。

在函数模式下，可以像普通的北太天元函数一样使用，例如：

```
bex "bex_create.c"
bex "-plugin" "main.cpp"
```

由于北太天元对函数调用的限制，每一个参数都需要使用双引号包含。

更详细的 `bex` 使用方法可以使用 `bex -h`（命令行模式）或者 `bex "-h"`（函数模式）进行查看。

3 API 文档

3.1 宏

BALTAM_PLUGIN_FCN	插件函数声明展开
BEX_API_VERSION_MAJOR	API 主版本号
BEX_API_VERSION_MINOR	API 次版本号
BALTAM_BEX_LEGACY_GETTER_RW	切换 Getter 函数默认行为

BALTAM_PLUGIN_FCN

BALTAM_PLUGIN_FCN(word) 会展开为

```
void word(int nlhs, bxArray *plhs[], int nrhs, const bxArray *prhs[])
```

从而方便定义接口函数。

BEX_API_VERSION_MAJOR

定义了当前 API 的主版本号。若当前 API 版本号为 v3.0，则该宏的值为 3。需要特别注意的是，若两个 API 主版本号不同，则至少引入了一个后向不兼容的修改。此时插件必须进行重新编译才能生效。

BEX_API_VERSION_MINOR

定义了当前 API 的次版本号。若当前版本号为 v3.0，则该宏的值为 0。次版本号增加时，可能引入了新的 API 或是 BUG 修复，原有的 API 不受影响，可以正常使用。

BALTAM_BEX_LEGACY_GETTER_RW

宏定义开关，控制了 bxGetXxxx 系列函数的默认行为。若启用该宏定义，则 bxGetXxxx 系列函数会被映射为 bxGetXxxxRW，即返回读写指针；否则会以只读模式获取。详细的讨论见第 4.3 节（Getter 函数接口使用问题）。

3.2 枚举类型

bxClassID	接口中北太天元内置数据类型 ID
bxComplexity	表示数组实数或复数的标记
bxOperatorID	内置算符 ID
bxFHandleType	表示函数句柄的类型

bxClassID

接口中北太天元类型 ID。

- bxUNKNOWN_CLASS 未知类型，所有 API 不能识别的类型；
- bxINT8_CLASS **数值类型**，有符号 8 位整数矩阵；
- bxINT16_CLASS **数值类型**，有符号 16 位整数矩阵；
- bxINT_CLASS bxINT32_CLASS **数值类型**，有符号 32 位整数矩阵；
- bxINT64_CLASS **数值类型**，有符号 64 位整数矩阵；
- bxUINT8_CLASS **数值类型**，无符号 8 位整数矩阵；
- bxUINT16_CLASS **数值类型**，无符号 16 位整数矩阵；
- bxUINT32_CLASS **数值类型**，无符号 32 位整数矩阵；
- bxUINT64_CLASS **数值类型**，无符号 64 位整数矩阵；

- bxSINGLE_CLASS **数值类型**，单精度浮点数矩阵（包含实数和复数，稠密和稀疏）；
- bxDOUBLE_CLASS **数值类型**，双精度浮点数矩阵（包含实数和复数，稠密和稀疏）；
- bxCHAR_CLASS 字符矩阵；
- bxLOGICAL_CLASS 逻辑类型矩阵；
- bxSTRUCT_CLASS 结构体；
- bxSTRING_CLASS 字符串数组；
- bxEXTERN_CLASS 外部对象；
- bxCELL_CLASS 元胞数组（cell）；

bxComplexity

是否为复数类型的标记。目前不支持复数整数。

- bxREAL 实数标记；
- bxCOMPLEX 复数标记；

bxOperatorID

算符类型 ID。用于表示算符。对照关系如下表。

算符	字符串表示	bxOperatorID	说明
+	"+"	bxADD_OP	加法，a + b
-	"-"	bxSUB_OP	减法，a - b
.*	".*"	bxTIMES_OP	数组乘法，a .* b
./	"./"	bxRDIV_OP	数组右除，a ./ b
.\	".\\"	bxLDIV_OP	数组左除，a .\b
*	"*"	bxMTIMES_OP	矩阵乘法，a * b
/	"/"	bxMRDIV_OP	矩阵右除（解线性方程组），b / A
\	"\\"	bxMLDIV_OP	矩阵左除（解线性方程组），A \ b
.^	".^"	bxPOW_OP	数组乘方，a .^ b
^	"^"	bxMPOW_OP	矩阵乘方，a ^ b
<	"<"	bxLT_OP	小于，a < b
>	">"	bxGT_OP	大于，a > b
<=	"<="	bxLE_OP	小于或等于，a <= b
>=	">="	bxGE_OP	大于或等于，a >= b
~=	"~="	bxNE_OP	不等于，a ~= b
==	"=="	bxEQ_OP	等于，a == b
&	"&"	bxAND_OP	逻辑与，a & b

|	"|"	bxOR_OP	逻辑或，a | b
[a, b]	"[,]"	bxHCAT_OP	水平并置，[a, b]
[a; b]	"[;]"	bxVCAT_OP	垂直并置，[a; b]
+	"u+"	bxUPLUS_OP	单目加，+a
-	"u-"	bxUMINUS_OP	单目减，-a
~	"~"	bxNOT_OP	非，~a
.'	".'"	bxTRANSP_OP	转置，a.'
'	"'"	bxCTRANSP_OP	共轭转置，a'
X(a)	"subsind"	bxSUBSIND_OP	转化为下标，X(a)
A(S)	"subsref"	bxSUBSREF_OP	取下标运算，A(S)
A(S) = B	"subsasgn"	bxSUBSASGN_OP	取下标赋值运算，A(S) = B

bxFHandleType

表示函数句柄的类型。

- bxFH_UNKNOWN 未知类型，或未能识别的类型；
- bxFH_ANONYMOUS 匿名函数，如 @(x) sin(x)；
- bxFH_VARIABLE 变量类型；
- bxFH_BUILTIN 内置函数，如 @zeros；
- bxFH_MFUNCTION M 脚本编写的函数；
- bxFH_SCRIPT M 脚本（非函数）。

3.3 数据类型

bxArray	北太天元内核数据的不透明类型
baSize	表示数组大小的整数类型
baIndex	表示数组指标的整数类型
baSparseIndex	表示稀疏矩阵 Ir、Jc 数组的整数类型

bxArray

bxArray 是用于表示软件内核数据的包装类型。约定软件中所有变量在 SDK 中均使用 bxArray* 来表示，开发者可认为每一个 bxArray* 类型指针都指向一个内核中的变量。

需要注意，所有 bxArray* 类型的指针都应该使用 SDK 的接口函数产生或获取。开发者不需要也不应该在程序中直接生成 bxArray 类型的对象，例如以下例子中的语句都是不合法的，会引发编译错误：

```
bxArray data; // 不要在栈空间中创建变量
bxArray *pdata = new bxArray(); // 不要手动生成一个 bxArray
```

但是创建 bxArray* 指针数组是允许的：

```
std::vector<bxArray*> arrays; // 合法
```

baSize

表示数组大小的整数类型，一般为 64 位有符号整数。

baIndex

表示指标的整数类型，一般为 64 位有符号整数。

baSparseIndex

稀疏矩阵中 Ir、Jc 数组中元素类型，与 baIndex 类型相同。

3.4 数组基本属性

用于获取数组的各项基本属性，或者判断数组的一些性质。

bxGetClassID	返回指定 bxArray 的类型 ID
bxTypeCStr	返回指定 bxArray 的类型名
bxCalcSingleSubscript	将下标形式转化为线性索引形式
bxGetNumberOfElements	获取数组的元素个数
bxGetNumberOfDimensions	获取数组的维数
bxGetDimensions	获取数组各个维度的大小
bxSetDimensions	设置数组各个维度的大小
bxGetM	获取数组的行数
bxSetM	设置数组的行数
bxGetN	获取数组的列数
bxSetN	设置数组的列数
bxResize	重设数组的大小（同时设置行列）
bxIsDouble	判断矩阵是否是双精度
bxIsSingle	判断矩阵是否是单精度
bxIsComplex	判断矩阵是否为复数类型

bxGetClassID

返回给定 bxArray 的类型 ID。

```
bxClassID bxGetClassID(
    const bxArray *ba);
```

- ba bxArray* 类型指针；

- 返回值：bxClassID 枚举类型。如果 bxArray* 对应的类型在 API 中不支持，则返回 bxUNKNOWN_CLASS。

bxTypeCStr

返回给定 bxArray 的类型名。

```
const char * bxTypeCStr(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：const char * 指针，表示类型对应的字符串常量。类型 ID 和类型名称对应如下表：

bxINT8_CLASS	int8
bxINT16_CLASS	int16
bxINT32_CLASS	int32
bxINT64_CLASS	int64
bxUINT8_CLASS	uint8
bxUINT16_CLASS	uint16
bxUINT32_CLASS	uint32
bxUINT64_CLASS	uint64
bxDOUBLE_CLASS	double
bxSINGLE_CLASS	single
bxCHAR_CLASS	char
bxLOGICAL_CLASS	logical
bxSTRUCT_CLASS	struct
bxSTRING_CLASS	string
bxEXTERN_CLASS	extern
bxCELL_CLASS	cell

- 注意：该函数不适合判断一个 bxArray* 对象的具体类型。推荐使用 bxGetClassID 进行判断。

bxCalcSingleSubscript

将下标形式转化为线性索引形式。即返回该位置的元素和数组首元素的距离。例如下标为 (2, 3)，bxCalcSingleSubscript 会计算 (2, 3) 位置的元素在内存上与 (1, 1) 位置元素的距离。

```
baIndex bxCalcSingleSubscript(
```

```
    const bxArray *ba,
    int ndim,
    baIndex *ind);
```

- ba bxArray* 类型指针，计算线性指标时所参考的数组和矩阵；
- ndim 数组的维数，也是 ind 参数的长度；
- ind 下标数组，长度至少为 ndim。指定要转换的下标。下标是从 1 开始的；
- 返回值：baIndex 类型的整数。表示 ind 对应的元素距数组首元素的距离。当发生错误时返回 -1，出错的场合包括：
  - ba 不是矩阵或元胞数组，取下标无意义；
  - ind 中存在分量超出数组维数；
- 计算公式：(i, j) 位置（从 1 开始）处的元素偏离首元素的距离为

$$\texttt{dist = (j - 1) * nrow + (i - 1)}$$

  其中 nrow 为矩阵或元胞数组的行数；
- 例子：

```
baIndex ind[] = {2, 3};
// 假设 ba 是一个 4 x 4 矩阵
baIndex i = bxCalcSingleSubscript(ba, 2, ind);
// 计算得 i = 9
```

bxGetNumberOfElements

返回给定 bxArray 的元素个数。等同于在软件中执行函数 numel()

```
baSize bxGetNumberOfElements(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：baSize 类型的整数（有符号）。表示元素个数。当 ba 为标量类型时（如函数句柄等），返回 1。

bxGetNumberOfDimensions

返回给定 bxArray 的维数。对于数组类型的变量，维数至少为 2。

```
baSize bxGetNumberOfDimensions(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：baSize 类型的整数（有符号）。表示维数。对数组类型的变量（数值数组，元胞数组，结构体等）返回值至少为 2，当 ba 为标量类型时（如函数句柄等），返回 1。

bxGetDimensions

返回给定 bxArray 各个维度的大小。该函数通常需要配合 bxGetNumberOfDimensions 共同使用。

```
const baSize * bxGetDimensions(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：const baSize * 指针，指向维数大小数组。数组的长度由 bxGetNumberOfDimensions 取得，禁止对返回的数组内容进行修改。若 ba 为标量类型时（如函数句柄等），返回 NULL。

bxSetDimensions

更改给定 bxArray 的各个维度大小。该函数和 bxSetM 和 bxSetN 类似，但可以用于高维数组。

```
void bxSetDimensions(
    bxArray *ba,
    const baSize *dims,
    baSize ndim);
```

- ba bxArray* 类型指针；
- dims 变更后各个维度的大小，如果里面有负的元素，则不进行任何操作；
- ndim 变更后的维数。数组 dims 的长度至少为 ndim。
- 变更后，同时会根据新的数组大小分配内存，并且会尽量保留原有部分的内容：每个维度大小增加时以 0 扩充（若是元胞数组则为空矩阵），减少时则进行截断。

bxGetM

返回给定 bxArray 的行数。若为高维数组，则返回的是第一个维度的大小。

```
baSize bxGetM(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：baSize 类型的整数（有符号）。如果 bxArray* 对应的实际变量不是数组类型，则返回 -1。

bxGetN

返回给定 bxArray 的列数。若为高维数组，则返回的是第二个维度的大小。

```
baSize bxGetN(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：baSize 类型的整数（有符号）。如果 bxArray* 对应的实际变量不是数组类型，则返回 -1。

bxSetM

更改指定 bxArray 的行数。当行数增加时以 0 填充，当行数减少时进行截断。原有的其它数据不变。若变量为高维数组，则仅更改第一个维度的大小，其它维度的大小不变。

```
void bxSetM(
    bxArray* ba,
    baSize m);
```

- ba bxArray* 类型指针；
- m 修改后矩阵的行数；
- 说明：当 ba 不是数组时，该函数无效果。

bxSetN

更改指定 bxArray 的列数。当列数增加时以 0 填充，当列数减少时进行截断。原有的其它数据不变。若变量为高维数组，则仅更改第二个维度的大小，其它维度的大小不变。

```
void bxSetN(
    bxArray* ba,
    baSize n;
```

- ba bxArray* 类型指针；

- n 修改后矩阵的列数；
- 说明：当 ba 不是数组时，该函数无效果。

bxResize

同时更改指定 bxArray 的行数和列数。当相应维数增加时以 0 填充，当减少时进行截断。原有的其它数据不变。若变量为高维数组，则仅更改前两个维度的大小，其它维度的大小不变。

```
void bxResize(
    bxArray* ba,
    baSize m,
    baSize n);
```

- ba bxArray* 类型指针；
- m 修改后矩阵的行数；
- n 修改后矩阵的列数；
- 说明：当 ba 不是数组时，该函数无效果。
- 例子：

```
/* 假设矩阵 A 为
    1 2 3
    4 5 6
    7 8 9      */
bxResize(A, 2, 4);
/* 修改后为
    1 2 3 0
    4 5 6 0    */
```

bxIsDouble

判断 bxArray 对象是否是双精度存储。

```
bool bxIsDouble(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象为任意类型的双精度矩阵（稠密、稀疏、实数、复数）时，返回 true，否则返回 false；

- 说明：该函数为兼容 MATLAB，不能用于判断一个变量更具体的类型。
  请使用 bxIsRealDouble 等函数来进行精确判断。

bxIsSingle

判断 bxArray 对象是否是单精度存储。

```
bool bxIsSingle(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象为任意类型的单精度矩阵（稠密、稀疏、实数、复数）时，返回 true，否则返回 false；
- 说明：该函数为兼容 MATLAB，不能用于判断一个变量更具体的类型。
  请使用 bxIsRealSingle 等函数来进行精确判断。

bxIsComplex

判断 bxArray 对象是否属于复数类型。

```
bool bxIsComplex(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象为任意类型的复数矩阵（稠密双精度，稠密单精度，稀疏双精度，稀疏单精度）时，返回 true，否则返回 false；
- 说明：该函数用于判断一个 bxArray 对象是否存储了虚部。虚部为 0 时，该函数依然返回 true。

3.5 创建、访问、修改数组数据

用于创建、访问、修改各种类型的数据。根据数组类型进一步分成若干子类。

3.5.1 稠密数值矩阵

针对稠密数值类型矩阵的 API，包括整数类型矩阵，实数/复数浮点类型矩阵。字符矩阵、逻辑矩阵、字符串矩阵等不在此范畴。

bxCreateDoubleMatrix	创建双精度类型矩阵
bxCreateNumericMatrix	创建给定数据类型的矩阵
bxCreateNumericArray	创建给定数据类型的高维数组
bxCreateInt8Scalar	创建 1×1 8 位有符号整数类型矩阵

bxCreateInt16Scalar	创建 1 × 1 16 位有符号整数类型矩阵
bxCreateInt32Scalar	创建 1 × 1 32 位有符号整数类型矩阵
bxCreateInt64Scalar	创建 1 × 1 64 位有符号整数类型矩阵
bxCreateUInt8Scalar	创建 1 × 1 8 位无符号整数类型矩阵
bxCreateUInt16Scalar	创建 1 × 1 16 位无符号整数类型矩阵
bxCreateUInt32Scalar	创建 1 × 1 32 位无符号整数类型矩阵
bxCreateUInt64Scalar	创建 1 × 1 64 位无符号整数类型矩阵
bxCreateDoubleScalar	创建 1 × 1 双精度实数类型矩阵
bxCreateSingleScalar	创建 1 × 1 单精度实数类型矩阵
bxCreateComplexDoubleScalar	创建 1 × 1 双精度复数类型矩阵
bxCreateComplexSingleScalar	创建 1 × 1 单精度复数类型矩阵
bxGetInt8s	返回 8 位有符号整数类型矩阵数据首地址
bxGetInt16s	返回 16 位有符号整数类型矩阵数据首地址
bxGetInt32s	返回 32 位有符号整数类型矩阵数据首地址
bxGetInt64s	返回 64 位有符号整数类型矩阵数据首地址
bxGetUInt8s	返回 8 位无符号整数类型矩阵数据首地址
bxGetUInt16s	返回 16 位无符号整数类型矩阵数据首地址
bxGetUInt32s	返回 32 位无符号整数类型矩阵数据首地址
bxGetUInt64s	返回 64 位无符号整数类型矩阵数据首地址
bxGetDoubles	返回双精度实数类型矩阵数据首地址
bxGetSingles	返回单精度实数类型矩阵数据首地址
bxGetComplexDoubles	返回双精度复数类型矩阵数据首地址
bxGetComplexSingles	返回单精度复数类型矩阵数据首地址
bxGetInt8sRO	返回 8 位有符号整数类型矩阵数据首地址（只读）
bxGetInt16sRO	返回 16 位有符号整数类型矩阵数据首地址（只读）
bxGetInt32sRO	返回 32 位有符号整数类型矩阵数据首地址（只读）
bxGetInt64sRO	返回 64 位有符号整数类型矩阵数据首地址（只读）
bxGetUInt8sRO	返回 8 位无符号整数类型矩阵数据首地址（只读）
bxGetUInt16sRO	返回 16 位无符号整数类型矩阵数据首地址（只读）
bxGetUInt32sRO	返回 32 位无符号整数类型矩阵数据首地址（只读）
bxGetUInt64sRO	返回 64 位无符号整数类型矩阵数据首地址（只读）
bxGetDoublesRO	返回双精度实数类型矩阵数据首地址（只读）
bxGetSinglesRO	返回单精度实数类型矩阵数据首地址（只读）
bxGetComplexDoublesRO	返回双精度复数类型矩阵数据首地址（只读）
bxGetComplexSinglesRO	返回单精度复数类型矩阵数据首地址（只读）
bxGetInt8sRW	返回 8 位有符号整数类型矩阵数据首地址（读写）

bxGetInt16sRW	返回 16 位有符号整数类型矩阵数据首地址（读写）
bxGetInt32sRW	返回 32 位有符号整数类型矩阵数据首地址（读写）
bxGetInt64sRW	返回 64 位有符号整数类型矩阵数据首地址（读写）
bxGetUInt8sRW	返回 8 位无符号整数类型矩阵数据首地址（读写）
bxGetUInt16sRW	返回 16 位无符号整数类型矩阵数据首地址（读写）
bxGetUInt32sRW	返回 32 位无符号整数类型矩阵数据首地址（读写）
bxGetUInt64sRW	返回 64 位无符号整数类型矩阵数据首地址（读写）
bxGetDoublesRW	返回双精度实数类型矩阵数据首地址（读写）
bxGetSinglesRW	返回单精度实数类型矩阵数据首地址（读写）
bxGetComplexDoublesRW	返回双精度复数类型矩阵数据首地址（读写）
bxGetComplexSinglesRW	返回单精度复数类型矩阵数据首地址（读写）
bxIsInt8	判断是否为 8 位有符号整数类型矩阵
bxIsInt16	判断是否为 16 位有符号整数类型矩阵
bxIsInt32	判断是否为 32 位有符号整数类型矩阵
bxIsInt64	判断是否为 64 位有符号整数类型矩阵
bxIsUInt8	判断是否为 8 位无符号整数类型矩阵
bxIsUInt16	判断是否为 16 位无符号整数类型矩阵
bxIsUInt32	判断是否为 32 位无符号整数类型矩阵
bxIsUInt64	判断是否为 64 位无符号整数类型矩阵
bxIsRealDouble	判断是否为双精度实数浮点类型矩阵
bxIsRealSingle	判断是否为单精度实数浮点类型矩阵
bxIsComplexDouble	判断是否为双精度复数浮点类型矩阵
bxIsComplexSingle	判断是否为单精度复数浮点类型矩阵

bxCreateDoubleMatrix

创建一个 $m \times n$ 双精度类型矩阵。

```
bxArray* bxCreateDoubleMatrix(
    baSize m,
    baSize n,
    bxComplexity comp);
```

- m,n 矩阵维数；
- comp 矩阵是否为复数的标记；
- 返回值：bxArray* 类型指针。

bxCreateNumericMatrix

创建一个 $m \times n$ 数值类型的矩阵。

```
bxArray* bxCreateNumericMatrix(
    baSize m,
    baSize n,
    bxClassID id,
    bxComplexity comp);
```

- m,n 矩阵维数；
- id 矩阵类型 ID，仅对数值类型有效；
- comp 矩阵是否为复数的标记（仅对浮点数有效）；
- 返回值：bxArray* 类型指针，若 id 取值无效则返回空指针。

bxCreateNumericArray

创建一个数值类型的高维数组。

```
bxArray* bxCreateNumericArray(
    baSize ndim,
    const baSize * dims,
    bxClassID id,
    bxComplexity comp);
```

- ndim 高维数组的维数，至少为 2；
- dims 每个维度的大小，长度至少为 ndim；
- id 类型 ID，仅对数值类型有效；
- comp 数组是否为复数的标记（仅对浮点数有效）；
- 返回值：bxArray* 类型指针，若 id 取值无效或创建失败则返回空指针。

bxCreateInt8Scalar
bxCreateInt16Scalar
bxCreateInt32Scalar
bxCreateInt64Scalar
bxCreateUInt8Scalar
bxCreateUInt16Scalar
bxCreateUInt32Scalar

bxCreateUInt64Scalar
bxCreateDoubleScalar
bxCreateSingleScalar
bxCreateComplexDoubleScalar
bxCreateComplexSingleScalar

创建 1×1 标量。

```
bxArray *bxCreateInt8Scalar(int8_t v);
bxArray *bxCreateInt16Scalar(int16_t v);
bxArray *bxCreateInt32Scalar(int32_t v);
bxArray *bxCreateInt64Scalar(int64_t v);
bxArray *bxCreateUInt8Scalar(uint8_t v);
bxArray *bxCreateUInt16Scalar(uint16_t v);
bxArray *bxCreateUInt32Scalar(uint32_t v);
bxArray *bxCreateUInt64Scalar(uint64_t v);
bxArray *bxCreateDoubleScalar(double v);
bxArray *bxCreateSingleScalar(float v);
bxArray *bxCreateComplexDoubleScalar(double v_real, double v_imag);
bxArray *bxCreateComplexSingleScalar(float v_real, float v_imag);
```

- `v` C/C++ 中基本数据类型的取值；
- `v_real`，`v_imag` 当创建复数标量时，分别表示实部、虚部的值；
- 返回值：`bxArray*` 类型指针，实际类型与创建的标量相对应。

bxGetInt8s
bxGetInt16s
bxGetInt32s
bxGetInt64s
bxGetUInt8s
bxGetUInt16s
bxGetUInt32s
bxGetUInt64s
bxGetDoubles
bxGetSingles
bxGetComplexDoubles
bxGetComplexSingles

返回指定类型数据的首地址。其中 `Xxxx` 表示内置数据类型（见函数原型）。

```
int8_t *bxGetInt32s(const bxArray* ba);
```

```
int16_t *bxGetInt64s(const bxArray* ba);
int32_t *bxGetInt32s(const bxArray* ba);
int64_t *bxGetInt64s(const bxArray* ba);
uint8_t *bxGetInt32s(const bxArray* ba);
uint16_t *bxGetInt64s(const bxArray* ba);
uint32_t *bxGetInt32s(const bxArray* ba);
uint64_t *bxGetInt64s(const bxArray* ba);
double *bxGetDoubles(const bxArray* ba);
float *bxGetSingles(const bxArray* ba);
void *bxGetComplexDoubles(const bxArray* ba);
void *bxGetComplexSingles(const bxArray* ba);
```

- `ba` `bxArray*` 类型指针；
- 返回值：对应类型的 C 语言指针。可以直接访问进行修改，会体现到对应的变量中。若 `ba` 的实际类型不是函数指定的类型，返回 `NULL`。当指定是复数时，返回的为 `void*`。使用者需要根据实际情形进行类型转化。北太天元复数存储方式为 `double[2]` 或 `float[2]`，和 C/C++ 标准库是二进制兼容的。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 `p` 是返回的指针，那么矩阵 $(2,3)$ 位置的元素（以 0 开始）位于 `p[3 * m + 2]`，`m` 是 `bxGetM` 返回的数（矩阵的行数）。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 `bxGetXxxxs` 并未考虑这样的机制。尽量使用 `bxGetXxxxsRO` 或 `bxGetXxxxsRW` 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetInt8sRO
bxGetInt16sRO
bxGetInt32sRO
bxGetInt64sRO
bxGetUInt8sRO
bxGetUInt16sRO
bxGetUInt32sRO
bxGetUInt64sRO
bxGetDoublesRO
bxGetSinglesRO
bxGetComplexDoublesRO
bxGetComplexSinglesRO

以只读模式返回指定类型数据的首地址。其中 `Xxxx` 表示内置数据类型（见函数原型）。

```
const int8_t *bxGetInt32sRO(const bxArray* ba);
const int16_t *bxGetInt64sRO(const bxArray* ba);
const int32_t *bxGetInt32sRO(const bxArray* ba);
const int64_t *bxGetInt64sRO(const bxArray* ba);
const uint8_t *bxGetInt32sRO(const bxArray* ba);
const uint16_t *bxGetInt64sRO(const bxArray* ba);
const uint32_t *bxGetInt32sRO(const bxArray* ba);
const uint64_t *bxGetInt64sRO(const bxArray* ba);
const double *bxGetDoublesRO(const bxArray* ba);
const float *bxGetSinglesRO(const bxArray* ba);
const void *bxGetComplexDoublesRO(const bxArray* ba);
const void *bxGetComplexSinglesRO(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：对应类型的 C 语言指针，不允许修改其中的内容。若 ba 的实际类型不是函数指定的类型，返回 NULL。当指定是复数时，返回的为 const void*。使用者需要根据实际情形进行类型转化。北太天元复数存储方式为 double[2] 或 float[2]，和 C/C++ 标准库是二进制兼容的。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 $(2,3)$ 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。

bxGetInt8sRW
bxGetInt16sRW
bxGetInt32sRW
bxGetInt64sRW
bxGetUInt8sRW
bxGetUInt16sRW
bxGetUInt32sRW
bxGetUInt64sRW
bxGetDoublesRW
bxGetSinglesRW
bxGetComplexDoublesRW
bxGetComplexSinglesRW

以读写模式返回指定类型数据的首地址。其中 Xxxx 表示内置数据类型（见函数原型）。

```
int8_t *bxGetInt32sRW(const bxArray* ba);
int16_t *bxGetInt64sRW(const bxArray* ba);
```

```
int32_t *bxGetInt32sRW(const bxArray* ba);
int64_t *bxGetInt64sRW(const bxArray* ba);
uint8_t *bxGetInt32sRW(const bxArray* ba);
uint16_t *bxGetInt64sRW(const bxArray* ba);
uint32_t *bxGetInt32sRW(const bxArray* ba);
uint64_t *bxGetInt64sRW(const bxArray* ba);
double *bxGetDoublesRW(const bxArray* ba);
float *bxGetSinglesRW(const bxArray* ba);
void *bxGetComplexDoublesRW(const bxArray* ba);
void *bxGetComplexSingles(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：对应类型的 C 语言指针。可以直接访问进行修改，会体现到对应的变量中。若 ba 的实际类型不是函数指定的类型，返回 NULL。当指定是复数时，返回的为 void*。使用者需要根据实际情形进行类型转化。北太天元复数存储方式为 double[2] 或 float[2]，和 C/C++ 标准库是二进制兼容的。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 $(2,3)$ 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。
- 该函数考虑了写入时复制，必要时会对 ba 指向的对象进行复制再返回，防止错误地修改了其它变量。

bxIsInt8
bxIsInt16
bxIsInt32
bxIsInt64
bxIsUInt8
bxIsUInt16
bxIsUInt32
bxIsUInt64
bxIsRealDouble
bxIsRealSingle
bxIsComplexDouble
bxIsComplexSingle

判断 bxArray 对象是否属于某特定类型。其中 Xxxx 表示内置数据类型（见函数原型）。

```
bool bxIsInt8(const bxArray* ba);
bool bxIsInt16(const bxArray* ba);
bool bxIsInt32(const bxArray* ba);
```

```
bool bxIsInt64(const bxArray* ba);
bool bxIsUInt8(const bxArray* ba);
bool bxIsUInt16(const bxArray* ba);
bool bxIsUInt32(const bxArray* ba);
bool bxIsUInt64(const bxArray* ba);
bool bxIsRealDouble(const bxArray* ba);
bool bxIsRealSingle(const bxArray* ba);
bool bxIsComplexDouble(const bxArray* ba);
bool bxIsComplexSingle(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象属于指定类型时，返回 true，否则返回 false；
- 以上函数在软件中分别对应的类型为：int8、int16、int32、int64、uint8、uint16、uint32、uint64、double，single，complex double，complex single。

3.5.2 字符矩阵

针对字符类型矩阵的 API，实际类型为 char，在软件中以单引号的形式赋值和使用。若需要针对字符串（双引号形式），请参阅节 3.5.6。

bxCreateCharMatrixFromStrings	使用 C 风格字符串数组创建多维字符矩阵
bxCreateCharArray	创建高维字符数组
bxCreateString	使用 C 风格字符串创建 1 × n 字符矩阵
bxGetChars	返回字符矩阵数据的首地址
bxGetCharsRO	返回字符矩阵数据的首地址（只读）
bxGetCharsRW	返回字符矩阵数据的首地址（读写）
bxIsChar	判断是否为字符矩阵

bxCreateCharMatrixFromStrings

使用 C 风格的字符串数组创建多维字符矩阵。

```
bxArray* bxCreateCharMatrixFromStrings(
    baSize n_str,
    const char ** p_str);
```

- n_str 字符串数组的大小（即使用多少个字符串创建矩阵）；
- p_str 字符串数组的首元素指针。p_str 的元素至少需要有 n_str 个，且每个元素必须以 '\0' 结尾；

- 返回值：bxArray* 类型指针，实际为 char mat 对象，大小为 n_str × maxlen，其中 maxlen 是输入字符串数组中最长的长度。其余字符串末尾以空字符补齐。
- 例子：

```
const char * p_str[] = {"hello", "world", "中国"};
bxArray* SS = bxCreateCharMatrixFromStrings(3, p_str);
/*
SS =
3 x 6 char matrix

    'hello '
    'world '
    '中国'
*/
```

bxCreateCharArray

创建高维字符数组，每个元素初始化为空字符 \0。

```
bxArray* bxCreateCharArray(
    baSize ndim,
    const baSize * dims);
```

- ndim 字符数组的维数，至少为 2；
- dims 每个维度的大小，长度至少为 ndim；
- 返回值：bxArray* 类型指针，实际为字符数组。若创建失败则返回空指针。

bxCreateString

创建一个字符类型矩阵。

```
bxArray* bxCreateString(
    const char *s);
```

- s 字符串常量；
- 返回值：bxArray* 类型指针（实际类型是 char mat）。

bxGetChars

返回字符矩阵数据的首地址。

```
char *bxGetChars(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：char * 类型的 C 语言指针。可以直接访问进行修改，会体现到对应的变量中。若 ba 的实际类型不是函数指定的类型，返回 NULL。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 (2,3) 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetChars 并未考虑这样的机制。尽量使用 bxGetCharsRO 或 bxGetCharsRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetCharsRO

以只读模式返回字符矩阵数据的首地址。

```
const char *bxGetCharsRO(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：const char * 类型的 C 语言指针，不允许修改其中的内容。若 ba 的实际类型不是函数指定的类型，返回 NULL。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 (2,3) 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。

bxGetCharsRW

以读写模式返回字符矩阵数据的首地址。

```
char *bxGetCharsRW(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：char * 类型的 C 语言指针，可以直接访问进行修改，会体现到对应的变量中。若 ba 的实际类型不是函数指定的类型，返回 NULL。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 (2,3) 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。
- 该函数考虑了写入时复制，必要时会对 ba 指向的对象进行复制再返回，防止错误地修改了其它变量。

bxIsChar

判断 bxArray 对象是否属于字符矩阵类型。

```
bool bxIsChar(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象属于字符矩阵类型 (char) 时，返回 true，否则返回 false。

3.5.3 逻辑矩阵

针对逻辑类型矩阵的 API，实际类型为 logical。

bxCreateLogicalMatrix	创建逻辑值类型的矩阵
bxCreateLogicalScalar	创建逻辑值类型的标量
bxCreateLogicalArray	创建逻辑类型高维数组
bxGetLogicals	返回逻辑类型矩阵数据的首地址
bxGetLogicalsRO	返回逻辑类型矩阵数据的首地址（只读）
bxGetLogicalsRW	返回逻辑类型矩阵数据的首地址（读写）
bxIsLogical	判断是否为逻辑类型矩阵

bxCreateLogicalMatrix

创建一个 $m \times n$ 逻辑值类型矩阵。

```
bxArray* bxCreateLogicalMatrix(
    baSize m,
    baSize n);
```

- m,n 矩阵维数；
- 返回值：bxArray* 类型指针。

bxCreateLogicalScalar

创建 1×1 逻辑类型标量。

```
bxArray *bxCreateLogicalScalar(bool v);
```

- v 标量的取值；
- 返回值：bxArray* 类型指针，实际类型为 logical。

bxCreateLogicalArray

创建高维逻辑数组，每个元素初始化为 false。

```
    bxArray* bxCreateLogicalArray(
    baSize ndim,
    const baSize * dims);
```

- ndim 逻辑数组的维数，至少为 2；

- dims 每个维度的大小，长度至少为 ndim；
- 返回值：bxArray* 类型指针，实际为逻辑数组。若创建失败则返回空指针。

bxGetLogicals

返回逻辑类型矩阵数据的首地址。

```
bool *bxGetLogicals(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：bool * 类型的 C 语言指针。可以直接访问进行修改，会体现到对应的变量中。若 ba 的实际类型不是函数指定的类型，返回 NULL。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 $(2,3)$ 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetLogicals 并未考虑这样的机制。尽量使用 bxGetLogicalsRO 或 bxGetLogicalsRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetLogicalsRO

以只读方式返回逻辑类型矩阵数据的首地址。

```
const bool *bxGetLogicalsRO(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：bool * 类型的 C 语言指针，不允许修改其中的内容。若 ba 的实际类型不是函数指定的类型，返回 NULL。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 $(2,3)$ 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。

bxGetLogicalsRW

以读写模式返回逻辑类型矩阵数据的首地址。

```
bool *bxGetLogicalsRW(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：bool * 类型的 C 语言指针，可以直接访问进行修改，会体现到对应的变量中。若 ba 的实际类型不是函数指定的类型，返回 NULL。
- 注意：对矩阵类型数据，在内存中是按列存储的。若 p 是返回的指针，那么矩阵 $(2,3)$ 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。

- 该函数考虑了写入时复制，必要时会对 ba 指向的对象进行复制再返回，防止错误地修改了其它变量。

bxIsLogical

判断 bxArray 对象是否属于逻辑矩阵类型。

```
bool bxIsLogical(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象属于逻辑矩阵（logical）类型时，返回 true，否则返回 false。

3.5.4 结构体

针对结构体类型变量的 API，实际类型为 struct。

bxCreateStructMatrix	创建结构体矩阵
bxCreateStructArray	创建高维结构体数组
bxGetNumberOfFields	获取字段总数量
bxGetFieldNumber	获取字段的编号
bxGetFieldNameByNumber	使用编号获取字段名称
bxGetField	按照名称获取字段
bxGetFieldRO	按照名称获取字段（只读）
bxGetFieldRW	按照名称获取字段（读写）
bxGetFieldByNumber	使用编号获取字段的值
bxGetFieldByNumberRO	使用编号获取字段的值（只读）
bxGetFieldByNumberRW	使用编号获取字段的值（读写）
bxSetField	按照名称设置字段
bxSetFieldByNumber	使用编号设置字段的值
bxAddField	添加一个字段
bxAddFieldAt	在指定位置添加一个字段
bxRenameField	将字段进行重命名
bxRemoveField	按照名称删除字段
bxIsStruct	判断是否为结构体矩阵
bxGetFieldNames	（C++）获取全部字段名称

bxCreateStructMatrix

创建一个结构体矩阵。

```
bxArray* bxCreateStructMatrix(
    baSize m,
    baSize n,
    int n_fields,
    const char **fieldnames);
```

- m,n 矩阵维数;
- n_fields 初始化时的字段个数;
- fieldnames 初始化时的字段，为字符串数组，长度不小于 n_fields。当 n_fields 为 0 时，fieldnames 参数可以为空指针;
- 返回值：bxArray* 类型指针（实际类型是 struct）;
- 说明：当给定 fieldnames 时，返回的结构体中相应字段的值被初始化为 double 类型的空矩阵。

bxCreateStructArray

创建一个高维结构体数组。

```
bxArray* bxCreateStructArray(
    baSize ndim,
    const baSize * dims,
    int n_fields,
    const char **fieldnames);
```

- ndim 结构体数组的维数，至少为 2;
- dims 各个维度的大小，长度至少为 ndim;
- n_fields 初始化时的字段个数;
- fieldnames 初始化时的字段，为字符串数组，长度不小于 n_fields。当 n_fields 为 0 时，fieldnames 参数可以为空指针;
- 返回值：bxArray* 类型指针（实际类型是 struct）;
- 说明：当给定 fieldnames 时，返回的结构体中相应字段的值被初始化为 double 类型的空矩阵。

bxGetField
bxGetFieldRO
bxGetFieldRW

获取一个结构体类型变量的某个字段的值。后缀为 RO 和 RW 分别对应只读模式和读写模式。

```
bxArray *bxGetField(
    const bxArray* ba,
    baIndex ind,
    const char *key);
const bxArray *bxGetFieldRO(
    const bxArray* ba,
    baIndex ind,
    const char *key);
bxArray *bxGetFieldRW(
    const bxArray* ba,
    baIndex ind,
    const char *key);
```

- ba bxArray* 类型指针；
- ind 所需要获取的结构体的所在分量位置；
- key 需要获取的字段名；
- 返回值：对于 bxGetField 和 bxGetFieldRW 函数，返回 bxArray* 指针，表示该字段的对象，之后对该对象内容的修改会直接体现到结构体中；对于 bxGetFieldRO 函数，返回 const bxArray* 指针，不允许进行修改。当 ba 不是结构体，或字段不存在时返回 NULL；
- 注意：不要使用 bxDestroyArray 函数来释放 bxGetField 的返回值，这通常会使得软件内部出现不可预见的错误。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetField 并未考虑这样的机制。尽量使用 bxGetFieldRO 或 bxGetFieldRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetFieldByNumber
bxGetFieldByNumberRO
bxGetFieldByNumberRW

使用字段编号获取结构体类型变量的某个字段的值。字段编号的范围为 0 至字段总数减 1，越早添加的字段编号越小。后缀为 RO 和 RW 分别对应只读模式和读写模式。

```
bxArray *bxGetFieldByNumber(
    const bxArray* ba,
    baIndex ind,
    int number);
```

```
const bxArray *bxGetFieldByNumberRO(
    const bxArray* ba,
    baIndex ind,
    int number);
bxArray *bxGetFieldByNumberRW(
    const bxArray* ba,
    baIndex ind,
    int number);
```

- ba bxArray* 类型指针；
- ind 所需要获取的结构体的所在分量位置；
- number 需要获取的字段编号；
- 返回值：对于 bxGetFieldByNumber 和 bxGetFieldByNumberRW 函数，返回 bxArray* 指针，表示该编号字段对应的对象，之后对该对象内容的修改会直接体现到结构体中；对于 bxGetFieldRO 函数，返回 const bxArray* 指针，不允许进行修改。当 ba 不是结构体，或字段编号超出范围时返回 NULL；
- 注意：不要使用 bxDestroyArray 函数来释放 bxGetFieldByNumber 的返回值，这通常会使得软件内部出现不可预见的错误。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetFieldByNumber 并未考虑这样的机制。尽量使用 bxGetFieldByNumberRO 或 bxGetFieldByNumberRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetNumberOfFields

获取一个结构体类型变量总的字段数量。

```
baSize bxGetNumberOfFields(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：一个整数，表示字段数量。如果 ba 不是结构体，则返回 -1。

bxGetFieldNumber

获取一个字段的编号。字段编号的范围为 0 至字段总数减 1，越早添加的字段编号越小。

```
int bxGetFieldNumber(
    const bxArray* ba,
    const char * fieldname);
```

- ba bxArray* 类型指针；
- fieldname 字段名；
- 返回值：一个整数，表示字段编号。如果字段不存在，则返回 -1。

bxGetFieldNameByNumber

按照字段编号获取字段名称。字段编号的范围为 0 至字段总数减 1，越早添加的字段编号越小。

```
const char * bxGetFieldNameByNumber(
    const bxArray* ba,
    int number);
```

- ba bxArray* 类型指针；
- number 字段编号，合法范围为 0 至字段总数减 1；
- 返回值：一个字符串，表示字段名称。如果编号越界，则返回 NULL。

bxGetFieldNames

C++ 函数。返回结构体对象的所有字段名称。该函数在对结构体字段进行遍历时非常有用。

```
std::vector<std::string> bxGetFieldNames(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：字符串数组（C++ 标准库）。其中包含了所有该结构体的字段名称。当结构体发生变化时，需要重新调用该函数获取更新后的字段名称。

bxSetField

修改一个结构体的某字段。**注意：** v3.0+ 起，要修改的字段必须在结构体中已经存在，该函数不能执行添加字段的操作。若需要添加字段，需要使用 bxAddField 或 bxAddFieldAt。

```
void bxSetField(
    bxArray* ba,
    baIndex ind,
    const char * key,
    bxArray* val);
```

- ba bxArray* 类型指针；

- ind 所需要修改的结构体的所在分量位置；
- key 需要修改的字段名；
- val 修改后的新值；
- 说明：
  - 当 ba 不是结构体类型，索引越界，**或字段不存在时**，该函数无任何效果；
  - 为了运行效率，bxSetField **不对** val 表示的 bxArray 对象进行复制，因此执行该函数过后再修改 val 依然会影响结构体内部的字段；
  - 在 API 中，同一个 bxArray 对象不可以被多个结构体共享，在同一结构体中也不可以被不同字段共享，也不可以同时出现在元胞数组和结构体中。以下的代码都是不合法的：

```
x = bxGetField(s1, 0, "x");
bxSetField(s2, 0, "x", x); // x 已经被 s1.x 占有，不能设置为 s2.x
bxSetField(s1, 0, "y", x); // x 已经被 s1.x 占有，不能设置为 s1.y
bxSetCell(c1, 0, x); // x 已经被 s1.x 占有，不能设置为 c1(1)

t = bxCreateDoubleMatrix(1, 1, bxREAL);
bxSetField(s3, 0, "t", t); // 设置 s3.t = t
bxSetField(s3, 0, "t1", t); // 非法，t 已经为 s3.t
```

    如有需要，应该先使用 bxDuplicateArray 进行复制；
  - 当 val 已经被设置为一个结构体的字段时，不要使用 bxDestroyArray 直接对 val 进行释放。当 val 所在结构体被销毁时，其内存会自动被系统回收。

bxSetFieldByNumber

根据字段编号，修改一个结构体的某字段。字段编号的范围为 0 至字段总数减 1，越早添加的字段编号越小。若需要添加字段，需要使用 bxAddField。

```
void bxSetFieldByNumber(
    bxArray* ba,
    baIndex ind,
    int number,,
    bxArray* val);
```

- ba bxArray* 类型指针；
- ind 所需要修改的结构体的所在分量位置；
- number 需要修改的字段编号；

- val 修改后的新值;
- 说明:
  - 当 ba 不是结构体类型，索引越界，或编号越界时，该函数无任何效果;
  - 其它的使用注意事项见 bxSetField

bxAddField

添加一个结构体字段。当执行 bxSetField 等函数时，请首先使用本函数创建字段对应的数据表。

```
void bxAddField(
    bxArray* ba,
    const char * fieldname);
```

- ba bxArray* 类型指针;
- fieldname 创建的字段名称。如果该字段名称已经存在则什么也不做，否则会生成和结构体维数相同的数据表。

bxAddFieldAt

在指定位置添加一个结构体字段，该位置之后的字段依次向后移动。当执行 bxSetField 等函数时，请首先使用本函数创建字段对应的数据表。

```
void bxAddFieldAt(
    bxArray* ba,
    baIndex number,
    const char * fieldname);
```

- ba bxArray* 类型指针;
- number 整数，表示要插入的位置，合法值为 0 至字段总数，该位置之后的字段将依次向后移动。例如原有字段为 ["a", "b", "c"]，在 number = 1 插入字段 x 后变为 ["a", "x", "b", "c"];
- fieldname 创建的字段名称。如果该字段名称已经存在则什么也不做，否则会生成和结构体维数相同的数据表。

bxRenameField

将字段进行重命名。

```
void bxRenameField(
    bxArray *ba,
    baIndex number,
    const char * new_name);
```

- ba bxArray* 类型指针；
- number 表示要重命名字段的位置，合法值为 0 至字段总数减 1；
- new_name 字段新的名字。

该函数根据 ba 对应结构体分以下情况进行处理：

- 若 number 超出合法范围，则什么也不做；
- 若修改前后的字段名称没有变化，则什么也不做；
- 若修改后的字段名恰好和别处字段同名，则直接将同名字段的值改为当前旧字段对应的值，结构体总字段数会减少 1 个。例如原有字段 ["a", "b", "c"]，现在要将 "a" 改名为 "c"，则修改后字段为 ["b", "c"]，且字段 "c" 的取值为原字段 "a" 的取值；
- 若修改后的字段名没有重名，则直接在当前位置做名称的修改，结构体总字段数和字段相对顺序均不变。

无论何种情况，使用 bxRenameField 不会使得 bxGetField 等结果的返回值失效，但要注意这些返回值仍然代表首次调用 bxGetField 时对应字段和分量的数据。例如，开发者先获取字段 "a" 的第 1 个分量的数据 p，又将字段 "a" 重命名为 "b"，则 p 现在表示字段 "p" 的第一个分量的数据。

bxRemoveField

删除一个结构体的某字段。

```
void bxRemoveField(
    bxArray *ba,
    const char * key);
```

- ba bxArray* 类型指针；
- key 需要删除的字段名；
- 说明：当 ba 不是结构体，或字段不存在时，该函数无效果；
- 注意：当一个字段被删除时，其对应的变量也会销毁。因此，如果先使用 bxGetField 获取一个字段之后使用 bxRemoveField 删除了该字段，则之前获取的 bxArray* 将会失效，不能继续访问。同样情景也发生在 bxSetField 执行过程中。

```
x = bxGetField(s, 0, "x");
bxRemoveField(s, "x"); // 此时 x 将失效，不能再使用

v = bxCreateDoubleMatrix(1, 1, bxREAL);
bxSetField(s, 0, "v", v);
bxRemoveField(s, "v"); // 此时 v 将失效，不可访问
```

bxIsStruct

判断 bxArray 对象是否属于结构体类型。

```
bool bxIsStruct(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象属于结构体（struct）类型时，返回 true，否则返回 false。

3.5.5 元胞数组

针对元胞数组的 API，实际类型为 cell。

bxCreateCellMatrix	创建元胞数组
bxCreateCellArray	创建高维元胞数组
bxGetCell	获取元胞数组中给定位置的元素
bxGetCellRO	获取元胞数组中给定位置的元素（只读）
bxGetCellRW	获取元胞数组中给定位置的元素（读写）
bxGetCellsV	（C++）获取元胞数组中所有元素
bxGetCellsVRO	（C++）获取元胞数组中所有元素（只读）
bxGetCellsVRW	（C++）获取元胞数组中所有元素（读写）
bxSetCell	设置元胞数组中给定位置元素的值
bxIsCell	判断是否为元胞数组

bxCreateCellMatrix

创建一个元胞数组。

```
bxArray* bxCreateCellMatrix(
    baSize m,
    baSize n);
```

- m,n 元胞数组维数；
- 返回值：bxArray* 类型指针（实际类型是 cell）；
- 说明：返回的元胞数组的每个元素的值被初始化为 double 类型的空矩阵。

bxCreateCellArray

创建一个高维元胞数组。

```
bxArray* bxCreateCellArray(
    baSize ndim,
    const baSize * dims);
```

- ndim 元胞数组的维数，至少为 2；
- dims 各个维度的大小，长度至少为 dims；
- 返回值：bxArray* 类型指针（实际类型是 cell）；
- 说明：返回的元胞数组的每个元素的值被初始化为 double 类型的空矩阵。

bxGetCell
bxGetCellRO
bxGetCellRW

获取一个元胞数组中特定位置的元素。后缀为 RO 和 RW 分别对应只读模式和读写模式。

```
bxArray *bxGetCell(
    const bxArray *ba,
    baIndex ind);
const bxArray *bxGetCellRO(
    const bxArray *ba,
    baIndex ind);
bxArray *bxGetCellRW(
    const bxArray *ba,
    baIndex ind);
```

- ba bxArray* 类型指针；
- ind 所需要获取的分量在元胞数组中的位置。一般通过 bxCalcSingleSubscript 来获取，也可手动计算；
- 返回值：对于 bxGetCell 和 bxGetCellRW 函数，返回 bxArray* 指针，表示元胞数组 ind 位置的对象，之后对该对象内容的修改会直接体现到结构体中；对于 bxGetCellRO 函数，返回 const bxArray* 指针，不允许进行修改。当 ba 不是元胞数组，或 ind 超出索引范围时返回 NULL；
- 说明：该函数用于访问元胞数组的一个分量，如要一次性获得元胞数组的全部分量，请使用 bxGetCellsV（C++ 函数）。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetCell 并未考虑这样的机制。尽量使用 bxGetCellRO 或 bxGetCellRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetCellsV
bxGetCellsVRO
bxGetCellsVRW

C++ 函数。以 std::vector 的形式返回一个元胞数组的所有元素。后缀为 RO 和 RW 分别对应只读模式和读写模式。

```
std::vector<bxArray*> bxGetCellsV(
    const bxArray* ba);
std::vector<const bxArray*> bxGetCellsVRO(
    const bxArray* ba);
std::vector<bxArray*> bxGetCellsVRW(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：对于 bxGetCellsV 和 bxGetCellsVRW 函数，返回 bxArray* 数组（C++ 标准库）；对于 bxGetCellsVRO 函数，返回 const bxArray* 数组（C++ 标准库），每个元素不允许修改。返回的 vector 其中包含了该元胞数组所有元素的指针。元素指针在内存中是按列存储的。若 v 是返回的 vector，那么元胞数组 $(2, 3)$ 位置的元素（以 0 开始）位于 v[3 * m + 2]，m 是 bxGetM 返回的数（矩阵的行数）。如果 ba 不是元胞数组，则返回一个空的 vector。
- 说明：该函数返回值（假设为 v）中包含的指针直接指向元胞数组对应位置的元素，在非 RO 模式下可以通过 bxGetXxxxs 等函数直接修改内容，元胞数组相应位置的元素内容也会发生改变。但 v 本身不是元胞数组所存储的数据，对 v 进行直接修改不会影响到元胞数组的内容，例如：

  ```
  v = bxGetCellsV(C); // 假设 C 是一个 bxArray*, 且为 cell
  v[0] = bxCreateDoubleMatrix(1, 1, bxREAL); // 无效果，因为直接修改了 v
  ```

  若要直接改变元胞数组本身的内容，请使用 bxSetCell。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetCellsV 并未考虑这样的机制。尽量使用 bxGetCellsVRO 或 bxGetCellsVRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxSetCell

设置一个元胞数组中某分量的值。

```
void bxSetCell(
    bxArray* ba,
    baIndex ind,
    bxArray* val);
```

- ba bxArray* 类型指针；

- ind 所需要获取的分量在元胞数组中的位置。一般通过 bxCalcSingleSubscript 来获取，也可手动计算；
- val 修改后的新值；
- 说明：
  - 当 ba 不是元胞数组，或 ind 超出索引范围，该函数无任何效果；
  - 修改后，原有位置的值会被丢弃，相应内存会被释放；
  - 为了运行效率，bxSetCell **不对** val 表示的 bxArray 对象进行复制，因此执行该函数过后再修改 val 依然会影响结构体内部的字段；
  - 在 API 中，同一个 bxArray 对象不可以被多个元胞数组共享，在同一元胞数组中也不可以被不同位置的分量共享，也不可以同时出现在元胞数组和结构体中。以下的代码都是不合法的：

```
x = bxGetCell(c1, 0);
bxSetCell(c2, 0, x); // x 已经被 c1(1) 占有，不能设置为 c2(1)
bxSetCell(c1, 1, x); // x 已经被 c1(1) 占有，不能设置为 c1(2)
bxSetField(s1, 0, "x", x); // x 已经被 c1(1) 占有，不能设置为 s1.x

t = bxCreateDoubleMatrix(1, 1, bxREAL);
bxSetCell(c3, 0, t); // 设置 c3(1) = t
bxSetCell(c3, 1, t); // 非法，t 已经为 c3(1)，不能设置为 c3(2)
```

    如有需要，应该先使用 bxDuplicateArray 进行复制；
  - 当 val 已经被设置为一个元胞数组元素时，不要使用 bxDestroyArray 直接对 val 进行释放。当 val 所在元胞数组被销毁时，其内存会自动被系统回收。
- 注意：当一个位置的元素因某种操作（改变元胞数组大小，同位置设置新值）被覆盖时，其对应的变量也会销毁。因此，如果先使用 bxGetCell 获取某位置的元素之后使用 bxSetCell 对同一位置的元素进行更改，则之前获取的 bxArray* 将会失效，不能继续访问。同样的情景也发生在 bxResize 等函数执行后。bxGetCellsV 函数的返回值同样也会受到 bxSetCell 的影响（仅限所修改的分量）。例如：

```
x = bxGetCell(c, 0);
y = bxCreateDoubleMatrix(1, 1, bxREAL);
bxSetCell(c, 0, y); // 对 c(1) 进行修改，此时 x 将失效，不可访问

v = bxGetCellsV(c);
z = bxCreateDoubleMatrix(1, 1, bxREAL);
bxSetCell(c, 1, z); // 对 c(2) 修改，此时 v[1] 将失效，不可访问
```

bxIsCell

判断 bxArray 对象是否属于元胞数组类型。

```
bool bxIsCell(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象属于元胞数组（cell）类型时，返回 true，否则返回 false。

3.5.6 字符串矩阵

针对字符串矩阵的 API，实际类型为 string，在软件中以双引号的形式赋值和使用。若需要针对字符矩阵（单引号形式），请参阅节 3.5.2。**如无特殊情况，不要使用本范畴中已过时的函数**。

bxCreateStringMatrix	创建字符串矩阵
bxCreateStringScalar	创建字符串标量
bxCreateStringArray	创建高维字符串数组
bxCreateStringMatrixFromStrings	使用 C 风格字符串数组创建字符串矩阵
bxGetString	返回字符串数组指定位置的数据首地址
bxGetStringPr	（C++）返回字符串数组首元素的地址
bxGetStringLength	返回字符串数组指定位置的字符串长度
bxSetString	使用 C 风格字符串设置字符串数组
bxIsString	判断是否为字符串矩阵
bxCreateStringObj	已过时
bxGetStringLen	已过时
bxGetStringDataPr	已过时
bxSetStringFromCStr	已过时

bxCreateStringScalar

创建 1×1 字符串矩阵。

```
bxArray *bxCreateStringScalar(const char *v);
```

- v C 风格字符串常量；
- 返回值：bxArray* 类型指针，实际类型是 string。

bxCreateStringMatrix

创建一个空字符串数组。

```
bxArray* bxCreateStringMatrix(
    baSize m,
    baSize n);
```

- m,n 字符串数组行列大小；
- 返回值：bxArray* 类型指针（实际类型是 string）；
- 说明：返回的字符串数组的每个元素的值被初始化空字符串。

bxCreateStringArray

创建一个高维字符串数组。

```
bxArray* bxCreateStringMatrix(
    baSize ndim,
    const baSize * dims);
```

- ndim 字符串数组维数，至少为 2；
- dims 各个维度的大小，长度至少为 ndim；
- 返回值：bxArray* 类型指针（实际类型是 string）；
- 说明：返回的字符串数组的每个元素的值被初始化空字符串。

bxCreateStringMatrixFromStrings

创建一个字符串数组并初始化为给定值。

```
bxArray* bxCreateStringMatrixFromStrings(
    baSize m,
    baSize n,
    const char **str);
```

- m,n 字符串数组维数；
- str C 风格字符串数组，大小至少为 m * n；
- 返回值：bxArray* 类型指针（实际类型是 string）；
- 说明：成功创建后，str 中的元素将按照**列优先**的顺序排布。

bxGetString

返回字符串数组指定位置的字符串的字符首地址（只读，实际为 C 风格的字符串）。

```
const char* bxGetString(
    const bxArray* ba,
    baIndex ind);
```

- ba bxArray* 类型指针；
- ind 所需要获取的分量在字符串数组中的位置。一般通过 bxCalcSingleSubscript 来获取，也可手动计算；
- 返回值：const char* 指针。只用作读取，不可以直接访问进行修改。若要对 string 进行修改，请使用 bxSetString 函数。若 ba 的实际类型不是 string，或为 ind 超出数组维度时，返回 NULL；

bxGetStringPr

C++ 函数。返回字符串数组首元素的地址。

```
std::string* bxGetStringPr(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：std::string* 指针，表示该字符串数组首元素的地址。可以直接访问或修改，会体现到对应的变量中。若 ba 的实际类型不是 string 时，返回 nullptr；
- 注意：对字符串数组类型数据，在内存中是按列存储的。若 p 是返回的指针，那么字符串数组 $(2,3)$ 位置的元素（以 0 开始）位于 p[3 * m + 2]，m 是 bxGetM 返回的数（数组的行数）。

bxGetStringLength

返回 string 数组给定位置的字符串的长度。仅对 string 类型的对象有效。

```
baSize bxGetStringLen(
    const bxArray* ba,
    baIndex ind);
```

- ba bxArray* 类型指针；
- ind 所需要获取的分量在字符串数组中的位置。一般通过 bxCalcSingleSubscript 来获取，也可手动计算；
- 返回值：baSize 类型的整数（有符号），表示首个字符串的长度。如果 bxArray* 对应的实际变量不是 string 类型，或是 ind 超出数组维数，则返回 -1；

bxSetString

将一个字符串数组指定位置的字符串的内容修改为给定的 C 风格字符串。请使用本函数进行字符串的修改。

```
void bxSetString(
    bxArray* ba,
    bxIndex ind,
    const char * str);
```

- ba bxArray* 类型指针；
- ind 所需要获取的分量在字符串数组中的位置。一般通过 bxCalcSingleSubscript 来获取，也可手动计算；
- str 修改后的字符串，必须以 '\0' 为结尾；
- 说明：需要使用本函数对 string 对象进行修改，而不是丢弃 bxGetString 的返回值的 const 属性后直接修改。

bxIsString

判断 bxArray 对象是否属于字符串矩阵类型。注：字符串类型为 string，和字符数数组（类型为 char）不同。

```
bool bxIsString(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象属于字符串矩阵（string）类型时，返回 true，否则返回 false。

bxCreateStringObj

已过时。创建一个 1 x 1 字符串矩阵。

```
bxArray* bxCreateStringObj(
    const char *s);
```

- s 字符串常量；
- 返回值：bxArray* 类型指针（实际类型是 string）；
- 该函数已过时。请使用 bxCreateStringScalar 创建 1 x 1 字符串矩阵。

bxGetStringLen

已过时。返回 string 数组首个字符串的长度。仅对 string 类型的对象有效。

```
baSize bxGetStringLen(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：baSize 类型的整数（有符号），表示首个字符串的长度。如果 bxArray* 对应的实际变量不是 string 类型，或是空的 string 数组，则返回 -1；
- 该函数已过时。建议使用 bxGetStringLength 获取给定位置的字符串长度。

bxGetStringDataPr

已过时。返回字符串数组首个字符串的字符首地址（只读，实际为 C 风格的字符串）。

```
const char* bxGetStringDataPr(
    const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：const char* 指针。只用作读取，不可以直接访问进行修改。若要对 string 进行修改，请使用 bxSetStringFromCStr 函数。若 ba 的实际类型不是 string，或为空字符串数组时，返回 NULL；
- 该函数已过时。请使用 bxGetString 来指定获取的字符串分量位置。

bxSetStringFromCStr

已过时。将一个字符串数组首元素的内容修改为给定的 C 风格字符串。请使用本函数进行字符串的修改。

```
void bxSetStringFromCStr(
    bxArray* ba,
    const char * str);
```

- ba bxArray* 类型指针；
- str 修改后的字符串，必须以 '\0' 为结尾；
- 说明：需要使用本函数对 string 对象进行修改，而不是丢弃 bxGetStringDataPr 的返回值的 const 属性后直接修改。
- 该函数已过时。请使用bxSetString 对指定位置的字符串进行修改。

3.5.7 稀疏矩阵

稀疏矩阵相关的 API。内核中支持的稀疏矩阵类型为单双精度浮点矩阵以及逻辑类型矩阵。

bxCreateSparse	创建双精度类型的稀疏矩阵
bxCreateSparseNumericMatrix	创建指定数据类型的稀疏矩阵
bxCreateSparseLogicalMatrix	创建逻辑类型的稀疏矩阵
bxIsSparse	判断变量是否为稀疏矩阵
bxIsSparseRealDouble	判断是否为稀疏双精度实数矩阵类型
bxIsSparseRealSingle	判断是否为稀疏单精度实数矩阵类型
bxIsSparseComplexDouble	判断是否为稀疏双精度复数矩阵类型
bxIsSparseComplexSingle	判断是否为稀疏单精度复数矩阵类型
bxIsSparseLogical	判断是否为稀疏逻辑矩阵类型
bxGetSparseDoubles	获取双精度实数稀疏矩阵数据指针
bxGetSparseSingles	获取单精度实数稀疏矩阵数据指针
bxGetSparseComplexDoubles	获取双精度复数稀疏矩阵数据指针
bxGetSparseComplexSingles	获取单精度复数稀疏矩阵数据指针
bxGetSparseLogicals	获取稀疏逻辑矩阵数据指针
bxGetIr	获取稀疏矩阵行指标数组
bxGetJc	获取稀疏矩阵列起始位置数组
bxGetSparseDoublesRO	获取双精度实数稀疏矩阵数据指针（只读）
bxGetSparseSinglesRO	获取单精度实数稀疏矩阵数据指针（只读）
bxGetSparseComplexDoublesRO	获取双精度复数稀疏矩阵数据指针（只读）
bxGetSparseComplexSinglesRO	获取单精度复数稀疏矩阵数据指针（只读）
bxGetSparseLogicalsRO	获取稀疏逻辑矩阵数据指针（只读）
bxGetIrRO	获取稀疏矩阵行指标数组（只读）
bxGetJcRO	获取稀疏矩阵列起始位置数组（只读）
bxGetSparseDoublesRW	获取双精度实数稀疏矩阵数据指针（读写）
bxGetSparseSinglesRW	获取单精度实数稀疏矩阵数据指针（读写）
bxGetSparseComplexDoublesRW	获取双精度复数稀疏矩阵数据指针（读写）
bxGetSparseComplexSinglesRW	获取单精度复数稀疏矩阵数据指针（读写）
bxGetSparseLogicalsRW	获取稀疏逻辑矩阵数据指针（读写）
bxGetIrRW	获取稀疏矩阵行指标数组（读写）
bxGetJcRW	获取稀疏矩阵列起始位置数组（读写）
bxGetNnz	获取实际非零元个数
bxGetNzmax	获取稀疏矩阵能存储的最大非零元个数
bxSetNzmax	设置稀疏矩阵能存储的最大非零元个数
bxSparseFinalize	更新稀疏矩阵内部信息，完成手动创建

内存格式

北太天元内部存储稀疏矩阵格式为 Compressed Sparse Column (CSC) 格式。该格式主要使用三个数组来对矩阵进行表示：ir、jc、pr 数组，如图 1 所示。

在 CSC 格式中各个数组的含义如下（其中 n 表示矩阵列数，nnz 表示矩阵存储的非零元个数，所有下标都以 0 为起始）：

- jc：baSparseIndex 数组，长度为 n + 1，jc[j] 表示第 j 列的第一个元素在数组 ir 和 pr 中的对应位置；
- ir：baSparseIndex 数组，长度为 nnz，里面按照列放置所有非零元的行坐标（和 pr 是对应的）；
- pr：可以为 double、float、bool 等类型数组，长度为 nnz，里面按照列放置所有非零元实际数值（和 ir 是对应的）。

需要注意，和直观理解不同，jc 数组**并不表示非零元的列坐标**，而是表示第 j 列的首元素在 ir 和 pr 中的位置。这种表示使得稀疏矩阵在内存中更加紧凑，也节省了一部分存储列指标的空间（一般情况下认为稀疏矩阵非零元数远大于 n，远小于 m * n）。

进一步地，稀疏矩阵先忽略矩阵中的 0，然后按照**列**将稀疏矩阵的每个非零元进行排布，ir 和 pr 数组的第 jc[j] 至 jc[j+1] 位置（不含末尾）就表示第 j 列所有非零元的位置和数值。利用这个结构容易对稀疏矩阵的非零元进行遍历：

```
for (baIndex j = 0; j < n; ++j){
    for (baIndex p = jc[j]; p < jc[j+1]; ++p){
        baIndex i = ir[p];
        double v = pr[p];

        // 此时 (i, j, v) 就是矩阵的实际非零元以及对应位置
    }
}
```

bxCreateSparse

创建双精度类型的稀疏矩阵，并初始化为全 0 矩阵。

```
bxArray* bxCreateSparse(
    baSize m,
    baSize n,
    baSize nzmax,
    bxComplexity cflag);
```

- m 矩阵行数；
- n 矩阵列数；

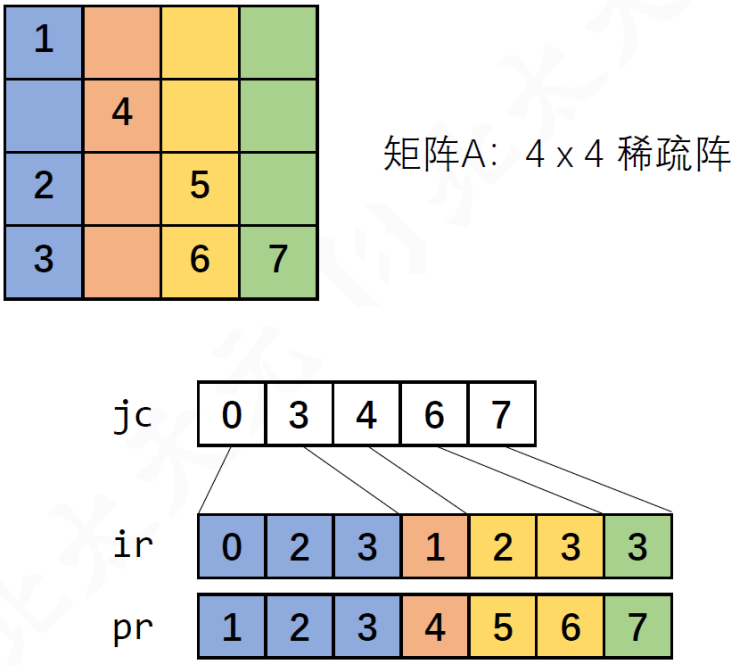

图 1: CSC 存储格式

- nzmax 稀疏矩阵最大非零元个数；
- cflag 矩阵是否为复数的标记；
- 返回值：bxArray* 类型指针，实际为双精度稀疏矩阵。

bxCreateSparseNumericMatrix

创建给定类型的稀疏矩阵，并初始化为全 0 矩阵。

```
bxArray* bxCreateSparseNumericMatrix(
    baSize m,
    baSize n,
    baSize nzmax,
    bxClassId id,
    bxComplexity cflag);
```

- m 矩阵行数；
- n 矩阵列数；
- nzmax 稀疏矩阵最大非零元个数；
- id 矩阵类型 ID，仅支持 bxDOUBLE_CLASS 或 bxSINGLE_CLASS；

- cflag 矩阵是否为复数的标记；
- 返回值：bxArray* 类型指针，实际为指定类型的稀疏矩阵。若 id 取值非法，返回 NULL。

bxCreateSparseLogicalMatrix

创建逻辑类型稀疏矩阵，并初始化为全 0 矩阵。

```
bxArray* bxCreateSparseLogicalMatrix(
    baSize m,
    baSize n,
    baSize nzmax);
```

- m 矩阵行数；
- n 矩阵列数；
- nzmax 稀疏矩阵最大非零元个数；
- 返回值：bxArray* 类型指针，实际为逻辑类型稀疏矩阵。

bxIsSparse

判断 bxArray 对象是否为稀疏矩阵。

```
bool bxIsSparse(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：当 ba 对应的对象为稀疏矩阵时，返回 true，否则返回 false。

bxIsSparseRealDouble
bxIsSparseRealSingle
bxIsSparseComplexDouble
bxIsSparseComplexSingle
bxIsSparseLogical

判断是否为指定类型的稀疏矩阵。

```
bool bxIsSparseRealDouble(const bxArray* ba);
bool bxIsSparseRealSingle(const bxArray* ba);
bool bxIsSparseComplexDouble(const bxArray* ba);
bool bxIsSparseComplexSingle(const bxArray* ba);
bool bxIsSparseLogical(const bxArray* ba);
```

- ba bxArray* 类型指针；

- 返回值：当对象属于相应类型的稀疏矩阵时，返回 `true`，否则返回 `false`；
- 说明：上面五个函数在软件中分别对应：`sparse double`、`sparse single`、`sparse complex double`，`sparse complex single`、`sparse logical`。

bxGetSparseDoubles
bxGetSparseSingles
bxGetSparseComplexDoubles
bxGetSparseComplexSingles
bxGetSparseLogicals

获取指定类型稀疏矩阵数据指针，实际为所有非零元素值按列优先依次排列。

```
double *bxGetSparseDoubles(const bxArray* ba);
float *bxGetSparseSingles(const bxArray* ba);
void *bxGetSparseComplexDoubles(const bxArray* ba);
void *bxGetSparseComplexSingles(const bxArray* ba);
bool *bxGetSparseLogicals(const bxArray* ba);
```

- `ba` `bxArray*` 类型指针；
- 返回值：对应类型的 C 语言指针。可以直接访问进行修改，会体现到对应的变量中。若 `ba` 的实际类型不是函数指定的类型，返回 `NULL`。当指定是复数时，返回的为 `void*`。使用者需要根据实际情形进行类型转化。北太天元复数存储方式为 `double[2]` 或 `float[2]`，和 C/C++ 标准库是二进制兼容的。
- 注意：返回数组的实际长度至少为 `bxGetNnz` 返回的数值，也可能会有一些额外的空间。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 `bxGetXxxxs` 并未考虑这样的机制。尽量使用 `bxGetXxxxsRO` 或 `bxGetXxxxsRW` 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetSparseDoublesRO
bxGetSparseSinglesRO
bxGetSparseComplexDoublesRO
bxGetSparseComplexSinglesRO
bxGetSparseLogicalsRO

以只读模式获取指定类型稀疏矩阵数据指针，实际为所有非零元素值按列优先依次排列。

```
const double *bxGetSparseDoublesRO(const bxArray* ba);
const float *bxGetSparseSinglesRO(const bxArray* ba);
const void *bxGetSparseComplexDoublesRO(const bxArray* ba);
```

```
const void *bxGetSparseComplexSinglesRO(const bxArray* ba);
const bool *bxGetSparseLogicalsRO(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：对应类型的 C 语言指针，不允许进行修改。若 ba 的实际类型不是函数指定的类型，返回 NULL。当指定是复数时，返回的为 const void*。使用者需要根据实际情形进行类型转化。北太天元复数存储方式为 double[2] 或 float[2]，和 C/C++ 标准库是二进制兼容的。
- 注意：返回数组的实际长度至少为 bxGetNnz 返回的数值，也可能会有一些额外的空间。

bxGetSparseDoublesRW
bxGetSparseSinglesRW
bxGetSparseComplexDoublesRW
bxGetSparseComplexSinglesRW
bxGetSparseLogicalsRW

获取指定类型稀疏矩阵数据指针，实际为所有非零元素值按列优先依次排列。

```
double *bxGetSparseDoublesRW(const bxArray* ba);
float *bxGetSparseSinglesRW(const bxArray* ba);
void *bxGetSparseComplexDoublesRW(const bxArray* ba);
void *bxGetSparseComplexSinglesRW(const bxArray* ba);
bool *bxGetSparseLogicalsRW(const bxArray* ba);
```

- ba bxArray* 类型指针；
- 返回值：对应类型的 C 语言指针。可以直接访问进行修改，会体现到对应的变量中。若 ba 的实际类型不是函数指定的类型，返回 NULL。当指定是复数时，返回的为 void*。使用者需要根据实际情形进行类型转化。北太天元复数存储方式为 double[2] 或 float[2]，和 C/C++ 标准库是二进制兼容的。
- 注意：返回数组的实际长度至少为 bxGetNnz 返回的数值，也可能会有一些额外的空间。
- 该函数考虑了写入时复制，必要时会对 ba 指向的对象进行复制再返回，防止错误地修改了其它变量。

bxGetIr
bxGetIrRO
bxGetIrRW

获取稀疏矩阵行指标数组（ir 数组）。后缀为 RO 和 RW 分别对应只读模式和读写模式。

```
bxSparseIndex* bxGetIr(const bxArray *ba);
const bxSparseIndex* bxGetIrRO(const bxArray *ba);
bxSparseIndex* bxGetIrRW(const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：对于 bxGetIr 和 bxGetIrRW 函数，返回 bxSparseIndex* 类型指针；对于 bxGetIrRO 函数，返回 const bxSparseIndex* 指针。所有情况下指针的内容实际为 CSC 格式的行指标数组（下标从 0 开始）。当 ba 实际类型不是稀疏矩阵时，返回 NULL。
- 说明：在非 RO 模式下可以直接对返回值数组进行直接修改，会体现到对应的稀疏矩阵变量中。返回数组的实际长度至少为 bxGetNnz 返回的数值，也可能会有一些额外的空间。
- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetIr 并未考虑这样的机制。尽量使用 bxGetIrRO 或 bxGetIrRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetJc
bxGetJcRO
bxGetJcRW

获取稀疏矩阵列指标数组（jc 数组）。后缀为 RO 和 RW 分别对应只读模式和读写模式。

```
bxSparseIndex* bxGetJc(const bxArray *ba);
const bxSparseIndex* bxGetJcRO(const bxArray *ba);
bxSparseIndex* bxGetJcRW(const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：对于 bxGetJc 和 bxGetJcRW 函数，返回 bxSparseIndex* 类型指针；对于 bxGetJcRO 函数，返回 const bxSparseIndex* 指针。所有情况下实际为 CSC 格式的列起始位置数组（下标从 0 开始）。**注：**该数组不表示列指标，请参考稀疏矩阵内存格式。当 ba 实际类型不是稀疏矩阵时，返回 NULL。
- 说明：可以直接对返回值数组进行直接修改，会体现到对应的稀疏矩阵变量中。返回数组的实际长度为 n + 1（n 为矩阵列数）。
- **注意**：当 Jc 数组内容发生改变时，需要调用 bxSparseFinalize 来完成稀疏矩阵修改，更新内部元数据。

- 建议：由于在 v3.0+ 版本引入了写入时复制的机制，旧版 bxGetJc 并未考虑这样的机制。尽量使用 bxGetJcRO 或 bxGetJcRW 系列函数获取数据地址。更多讨论见第 4.3 节。

bxGetNnz

获取稀疏矩阵实际非零元个数。

```
baSize bxGetNnz(const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：baSize 类型整数（有符号），表示稀疏矩阵实际非零元个数，它总是不超过 bxGetNzmax 返回的数值，一般情况下等价于 p[n] - p[0]，其中 p 是 bxGetJc 返回值，n 为矩阵列数。如果 bxArray* 对应的实际变量不是稀疏矩阵，则返回 -1；

bxGetNzmax

获取稀疏矩阵能存储的最大非零元个数。

```
baSize bxGetNzmax(const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：baSize 类型的整数（有符号），表示最大非零元个数。如果 bxArray* 对应的实际变量不是稀疏矩阵，则返回 -1；
- 说明：最大非零元个数一般情况下大于或等于矩阵实际的非零元个数。

bxSetNzmax

设置稀疏矩阵能存储的最大非零元个数。

```
void bxSetNzmax(
    bxArray *ba,
    baSize nzmax);
```

- ba bxArray* 类型指针；
- nzmax 新的最大非零元个数；
- 说明：如果 bxArray* 对应的实际变量不是稀疏矩阵，则该函数无效果。若 nzmax 参数小于矩阵实际的非零元个数，则 nzmax 被调整为实际非零元个数。

bxSparseFinalize

当稀疏矩阵 Jc 数组发生变化时，更新稀疏矩阵内部信息，完成手动修改。

```
void bxSparseFinalize(bxArray *ba);
```

- ba bxArray* 类型指针；
- 说明：当使用 bxGetJc 并直接对返回的数据进行修改时，稀疏矩阵的非零元结构已经发生改变。由于软件无法感知内存层面的直接修改，需要调用本函数完成内部信息的更新，否则稀疏矩阵将会处于一个无法使用的状态。比较常见的场合是手动构造 CSC 矩阵，或是第三方库计算后得到一个 CSC 矩阵用于传入软件内部。

3.5.8 函数句柄

函数句柄相关的 API。仅支持函数句柄的内容读取，暂不支持函数句柄的创建。

bxIsFunctionHandle	判断一个对象是否为函数句柄
bxGetFunctionHandleType	获取函数句柄的类型
bxGetFunctionHandleData	获取函数句柄的内容

bxIsFunctionHandle

判断 bxArray 对象是否属于函数句柄类型。

```
bool bxIsFunctionHandle(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：若 ba 的类型为函数句柄，返回 true。

bxGetFunctionHandleType

获取函数句柄的类型。

```
bxFHandleType bxGetFunctionHandleType(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：若 ba 的类型为函数句柄，返回实际句柄类型，为 bxFHandleType 类型枚举常量；否则返回 bxFH_UNKNOWN。

bxGetFunctionHandleData

获取函数句柄的具体内容，例如 f = @sin，则具体内容为 @sin 组成的字符串。

```
const char * bxGetFunctionHandleData(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：若 ba 的类型为函数句柄，返回字符串形式的函数句柄内容；否则返回 NULL。

3.6 复制和删除数组

bxArray 数据复制和释放相关 API。

bxDuplicateArray	复制 bxArray 对象
bxDuplicateArrayS	浅复制 bxArray 对象
bxDestroyArray	释放 bxArray 对象

bxDuplicateArray

复制一个 bxArray 对象。为深拷贝，复制得到的新 bxArray 所指对象和原对象有相同的内容。

```
bxArray* bxDuplicateArray(
    const bxArray *ba);
```

- ba 要复制的 bxArray* 类型指针；
- 返回值：bxArray* 类型指针。

bxDuplicateArrayS

复制一个 bxArray 对象。为浅拷贝，复制得到的新 bxArray 所指对象和原对象相同，是一个引用。**注意：**从 v3.0+ 起，该函数使用无任何限制（之前的版本不能用于生成 plhs[] 中的对象。

```
bxArray* bxDuplicateArrayS(
    const bxArray *ba);
```

- ba 要复制的 bxArray* 类型指针；
- 返回值：bxArray* 类型指针。

bxDestroyArray

释放一个 bxArray* 对应的内存。释放后，将不能通过该指针访问改对象。该函数不能使用的情形见如下函数的说明。**请勿**释放处于 prhs[]，plhs[] 数组中的对象、bxGetField、或 bxGetCell 返回的对象。

```
void bxDestroyArray(
    bxArray *ba);
```

- ba 要释放的 bxArray* 类型指针；

- 说明：以下情况**不能**使用本函数释放内存，否则会引起软件行为异常：
  - prhs 数组中的对象；
  - bxGetField 或 bxGetFieldByNumber 返回的指针；
  - bxGetCell 以及 bxGetCellsV 返回的指针；
  - plhs 数组中的对象，如果有必要释放，请保证释放后相应指针为空指针。

3.7 运行内置函数与命令

在插件函数或者 BEX 文件中可以直接调用北太天元命令。使用这个功能需要北太天元解释器处于运行状态。

bxEvalString	运行一个脚本命令
bxCallBaltamatica	使用字符串方式运行函数或脚本
bxCallBaltamaticaF	使用函数句柄的方式运行函数或脚本

bxEvalString

运行一个由字符串表示的北太天元命令。

```
int bxEvalString(const char *expr);
```

- expr 所需运行的表达式，例如 "3 + 5;"；
- 返回值：如果 expr 成功执行，返回 0，执行出现错误返回 1；
- 说明：这个函数无法获取返回值，但运行的表达式可能会更改工作区的状态。如果想获取表达式的返回值，需要使用 bxCallBaltamatica 或者 bxCallBaltamaticaF；
- 注意：用户需要自己保证 expr 内容的安全性，北太天元不会对运行的命令内容做检查。

bxCallBaltamatica

使用字符串方式运行函数或脚本，并获取返回值。

```
int bxCallBaltamatica(
    int nlhs,
    bxArray *plhs[],
    int nrhs,
    const bxArray *prhs[],
    const char *fun);
```

- nlhs 输出参数的个数；

- plhs 输出参数的指针数组，长度为 nlhs；
- nrhs 输入参数的个数；
- prhs 输入参数的指针数组，长度为 nrhs；
- fun 字符串，表示要运行的函数名或脚本名，例如 "sin" 表示运行内置的 sin 函数；
- 返回值：若成功运行，返回 0，若运行出现错误，返回 1；
- 说明：实际执行的内容和软件当前状态相关，例如当前目录，path 信息等，用户需自行保证函数的正确性。

bxCallBaltamaticaF

使用函数句柄运行函数或脚本，并获取返回值。

```
int bxCallBaltamaticaF(
    int nlhs,
    bxArray *plhs[],
    int nrhs,
    const bxArray *prhs[],
    const bxArray *fh);
```

- nlhs 输出参数的个数；
- plhs 输出参数的指针数组，长度为 nlhs；
- nrhs 输入参数的个数；
- prhs 输入参数的指针数组，长度为 nrhs；
- fh 函数句柄，通常是实际命令行传递进来的参数；
- 返回值：若成功运行，返回 0，若运行出现错误，返回 1；
- 说明：实际执行的内容和软件当前状态相关，例如当前目录，path 信息等，用户需自行保证函数的正确性。

3.8 数据转化

按照一定的规则将 bxArray 对象转化为 C++ 内置数据类型。或者是 bxArray 对象之间进行相互转化。

bxAsInt	转化为整数
bxAsCStr	转化为 C 风格字符串
bxAsString	（C++）转化为 STL string

bxAsInt

尝试将给定 bxArray 对象转化为内置整数。

```
baInt bxAsInt(
    const bxArray *ba,
    int * err);
```

- ba bxArray* 类型指针；
- err 整数指针。退出时 err 指向的变量表示 bxAsInt 转化是否成功。0 表示成功，1 表示失败。
- 返回值：baInt 类型的整数（有符号），表示 ba 对应的整数。如果 ba 不可转化为整数，则返回 0，err 指向变量的值也会被设置为 1。
- 可以被转化为整数的对象包括：
  - 大小为 1×1 的整数矩阵或逻辑值矩阵；
  - 大小为 1×1 的实数矩阵，且取值为整数；

  注意：复数类型的数据不认为可以转化为整数。

bxAsCStr

尝试将给定 bxArray 对象转化为 char 类型字符串。

```
int bxAsCStr(
    const bxArray *ba,
    char *buff,
    baSize size);
```

- ba bxArray* 类型指针；
- buff 存放输出字符串的数组，必须预先分配好内存空间；
- size 输出数组 buff 的大小；
- 返回值：int 类型整数，表示转化的错误码。0 表示正常退出，1 表示 buff 数组空间不足，此时转化发生截断，-1 表示 ba 不可被转化为 char 类型字符串。
- 在本函数成功执行后，buff 包含的字符串总是以 '\0' 结尾。如果 ba 对应的字符串长度超过 size - 1，则转化时会做截断。例如 size = 5 时，"hello world" 会被转化为 "hell\0"。
- 可以被转化的对象包括：

- 大小为 1×1 的 string 类型对象；
- 大小为 $1 \times n$ 的字符矩阵；
- 空字符矩阵（转化后结果为空字符串）；

bxAsString

C++ 函数。尝试将给定 bxArray 对象转化为 std::string 字符串对象。当转化失败时抛出异常。

```
std::string bxAsString(
    const bxArray *ba);
```

- ba bxArray* 类型指针；
- 返回值：转化后的字符串；
- 异常：当 ba 不可被转化为字符串时，会抛出 std::invalid_argument 类型的异常。可以被转化的对象见 bxAsCStr 函数说明。

3.9 算符重载

插件对内核操作符进行重载的函数，通常在插件载入时会调用。从 API v3.2 起，运算符重载的机制发生变化，具体见第 4.5 节的说明。

bxRegisterOperator	根据算符名注册重载算符
bxRegisterOperatorID	根据算符 ID 注册重载算符
bxRegisterUnaryOperator	（**已过时**）重载单目运算符
bxRegisterBinaryOperator	（**已过时**）重载双目运算符
bxRegisterTernaryOperator	（**已过时**）重载三目运算符

bxRegisterOperator

使用算符的名称向内核注册重载的算符，一般该函数会写在 bxPluginInit 内部。

```
int bxRegisterOperator(
    const char * plugin_name,
    const char * op_str,
    int t,
    bexfun_t fun_ptr);
```

- plugin_name 字符串常量，必须为插件的名称；
- op_str 算符对应的字符串，见 bxOperatorID 枚举类型；
- t 类型 ID，必须是 bxRegisterCStruct 或 bxAddCXXClass 返回的对象 ID；

- fun_ptr 完成重载功能的函数指针，和插件函数有相同的签名，即 bexfun_t 类型；
- 返回值：int 类型整数。含义如下：0 表示成功注册，-1 表示插件未被载入，-2 表示该算符已经和类型 t 进行了绑定，-3 表示类型 t 无效；
- 注意：从 API v3.2 起，为了减少干扰，不再允许注册对内置数据类型的重载算符。

bxRegisterOperatorID

使用算符 ID 向内核注册重载的算符，一般该函数会写在 bxPluginInit 内部。

```
int bxRegisterOperatorID(
    const char * plugin_name,
    bxOperatorID op,
    int t,
    bexfun_t fun_ptr);
```

- plugin_name 字符串常量，必须为插件的名称；
- op 算符 ID，见 bxOperatorID 枚举类型；
- t 类型 ID，必须是 bxRegisterCStruct 或 bxAddCXXClass 返回的对象 ID；
- fun_ptr 完成重载功能的函数指针，和插件函数有相同的签名，即 bexfun_t 类型；
- 返回值：int 类型整数。含义如下：0 表示成功注册，-1 表示插件未被载入，-2 表示该算符已经和类型 t 进行了绑定，-3 表示类型 t 无效；
- 注意：从 API v3.2 起，为了减少干扰，不再允许注册对内置数据类型的重载算符。

bxRegisterUnaryOperator
bxRegisterBinaryOperator
bxRegisterTernaryOperator

已过时。向内核中注册重载的单目 (unary)、双目 (binary)、或三目 (ternary) 算符。一般该函数会写在 bxPluginInit 内部。由于使用了新的重载机制，这三个函数已过时，但为了兼容性仍然保留在 API 中。建议用户使用 bxRegisterOperator 或 bxRegisterOperatorID 来进行注册。

```
int bxRegisterUnaryOperator(
    const char * plugin_name,
    const char * op_str,
    int t1,
    bexfun_t fun_ptr);

int bxRegisterBinaryOperator(
```

```
    const char * plugin_name,
    const_char * op_str,
    int t1,
    int t2,
    bexfun_t fun_ptr);

int bxRegisterTernaryOperator(
    const char * plugin_name,
    const_char * op_str,
    int t1,
    int t2,
    int t3,
    bexfun_t fun_ptr);
```

- plugin_name 字符串常量，必须为插件的名称；
- op_str 算符对应的字符串，见 bxOperatorID 枚举类型；
- t1，t2，t3 对象类型 ID，为 int 类型整数。
  可以是 bxRegisterCStruct 或 bxAddCXXClass 返回的对象 ID，在 v3.2+ 中，t2，t3 已经不起作用；
- fun_ptr 完成重载功能的函数指针，和插件函数有相同的签名，即 bexfun_t 类型；
- 返回值：int 类型整数。含义如下：0 表示成功注册，-1 表示插件未被载入，-2 表示该算符已经和类型 t1，t2，t3 进行了绑定，-3 表示类型 t1，t2，t3 无效；
- 注意：从 API v3.2 起，为了减少干扰，不再允许注册对内置数据类型的重载算符。因此 t1，t2，t3 都必须是外部对象的 ID；
- 由于注册机制调整，这三个函数现在都等价于调用 bxRegisterOperator(name，str，t1，fun)，即参数 t2，t3 无效。

3.10 函数重载

bxOverloadFunction	添加插件重载函数
bxReturnOverloadFailure	立即结束函数运行，并标记为重载失败
bxReturnOverloadFailureMsg	立即结束函数运行，标记为重载失败并显示失败信息

bxOverloadFunction

添加插件重载函数。一般在 bxPluginInit 中调用。

```
int bxOverloadFunction(
    const char * plugin_name,
    const char * name,
    bexfun_t ptr);
```

- plugin_name 插件名，注明这是哪个插件提供的重载函数；
- name 函数名，表示要重载的函数名；
- ptr 函数指针，表示该重载实际使用的 C/C++ 函数实现，函数原型和普通插件函数相同；
- 返回值：int 类型整数，若注册成功返回 0，否则返回非 0 值。

bxReturnOverloadFailure

立即结束函数运行，并标记为重载失败。

```
void bxReturnOverloadFailure();
```

- 该函数无任何输入和输出，当插件重载函数认为无法处理目前的输入参数，而又不能确定是执行错误时，需要调用本函数立即结束重载函数的执行，并标记为这是一个重载失败错误；
- 该函数和 bxErrMsgTxt 的区别是，bxErrMsgTxt 会立即终止函数执行，但北太天元会认为是执行错误，不会进行后续重载的判断；
- 若要显示额外信息，请使用 bxReturnOverloadFailureMsg。

bxReturnOverloadFailureMsg

立即结束函数运行，标记为重载失败，并显示失败信息。

```
void bxReturnOverloadFailure(const char * msg);
```

- msg 重载失败时显示的错误信息，可能会被北太天元后续使用；
- 当重载函数需要同时注册为普通函数时，使用函数可以传递重载错误信息。当其作为普通函数调用时，重载错误会当成一般错误进行处理。如果没有将重载函数注册为普通函数，一般不需要输出错误信息，直接使用 bxReturnOverloadFailure 即可；
- 该函数和 bxErrMsgTxt 的区别是，bxErrMsgTxt 会立即终止函数执行，但北太天元会认为是执行错误，不会进行后续重载的判断；
- 若不需要显示额外信息，请使用 bxReturnOverloadFailure。

3.11 输出与调试

用于在插件函数中直接向命令行输出字符串或调试信息的函数。

bxPrintf	向命令行格式化输出
bxErrMsgTxt	终止函数调用并打印错误信息

bxPrintf

向图形界面或命令行窗口格式化输出（和北太天元打开的模式有关）。

```
int bxPrintf(
    const char * format,
    ...);
```

- format 格式串；
- ... 变长参数，和 C/C++ 中的 printf 含义一致；
- 返回值：int 类型整数。表示实际输出的字符个数。

bxErrMsgTxt

终止当前函数调用并向图形界面或命令行窗口输出错误信息。

```
void bxErrMsgTxt(
    const char * str);
```

- str 表示错误信息的字符串；
- **该函数只能在插件函数或 BEX 函数内部调用**。调用后，插件函数或 BEX 函数会立即返回，不会执行代码之后的内容。

3.12 外部类

和外部类注册、托管、读取相关的 API。主要分为 C++ 对象和 C 结构体两种情形。

bxRegisterCStruct	注册 C 结构体
bxCreateCStruct	创建 C 结构体对象
bxGetCStruct	获取 C 结构体指针
bxAddCXXClass	（C++）添加 C++ 对象
bxCreateExtObj	（C++）创建并托管外部 C++ 对象
bxGetExtObj	（C++）获取外部 C++ 对象指针

bxRegisterCStruct

向内核添加一个 C 结构体类型。在调用所有外部 C 结构体操作前，必须先执行。一般该函数会写在 bxPluginInit 内部。

```
int bxRegisterCStruct(
    const char *name,
    cstruct_copy_t cpy,
    cstruct_delete_t del);
```

- name C 风格字符串常量，必须为插件的名称；
- cpy 复制函数的指针（不可以为 NULL）；
- del 析构函数的指针（不可以为 NULL）；
- 返回值：一个 int 类型的整数表示北太天元自动分配的 CStruct ID（后续会使用）。添加失败时返回 -1。

bxCreateCStruct

创建一个外部 C 结构体对象。

```
bxArray* bxCreateCStruct(
    int sid,
    void *data);
```

- sid 要创建的结构体 ID（由 bxRegisterCStruct 提供）；
- data void* 类型指针，北太天元会将 data 指向的内存直接绑定到北太天元中，不涉及复制操作，之后对 data 操作可以直接影响绑定后变量的状态。由开发者来保证 data 和 sid 是一致的；
- 返回值：bxArray* 类型的指针，表示创建好的北太天元对象。

bxGetCStruct

获取原始 C 结构体指针。

```
void* bxGetCStruct(
    int sid,
    const bxArray *ba);
```

- sid 要获取的结构体 ID（由 bxRegisterCStruct 提供）；
- ba bxArray* 类型指针；
- 返回值：void* 类型的指针，可以安全转化为 sid 对应的数据类型的指针。若 ba 不是合法的外部对象或 ID 不匹配，返回空指针。

bxAddCXXClass

C++ 函数。向内核添加一个 C++ 类。在调用所有外部类操作前，必须首先执行这个函数。一般该函数会写在 bxPluginInit 内部。

```
template <class T>
int bxAddCXXClass(
    const std::string& name);
```

- T 模板参数，要添加的 C++ 类名字；
- name 本插件的名字；
- 返回值：int 类型整数，表示类的 ID（通常用不到）。

bxCreateExtObj

C++ 函数。创建一个外部对象。

```
template <class T, typename... Args>
bxArray *bxCreateExtObj(
    Args&&... args);
```

- T 模板参数，要创建的外部对象的 C++ 类型，必须是 extern_obj_base 的子类。
- Args 模板变长参数，实际使用时 args 会作为 T 构造函数的参数传入。
- 返回值：bxArray* 类型指针，如果创建出现错误返回 NULL（通常可能是 T 不是由 extern_obj_base 派生的类型，或者没有添加到内核中）。

bxGetExtObj

C++ 函数。返回一个 bxArray 对象所对应的外部类的 C++ 指针。

```
template <class T>
T * bxGetExtObj(
    const bxArray* ba);
```

- T 模板参数，要获取的 C++ 类名字；
- ba bxArray* 类型指针；
- 返回值：T* 类型的 C++ 指针。若 ba 不是合法的外部对象，返回空指针。

3.13 内部查询

向前端或内核查询内部数据的函数，供高级开发者使用。

bxF2KQuery	向内核的查询协议函数
bxK2FQuery	向前端的查询协议函数

bxF2KQuery

向内核查询的函数。支持的协议见内部文档。

```
void * bxF2KQuery(
    const char * op,
    void * data);
```

- op 字符串常量，表示要做何种查询操作；
- data 操作对应的辅助数据；
- 返回值：void* 指针，表示查询结果。如果出错返回 NULL。个别协议正常退出也会返回空指针。

bxK2FQuery

向前端查询的函数。支持的协议见内部文档。

```
void * bxK2FQuery(
    const char * op,
    void * data);
```

- op 字符串常量，表示要做何种查询操作；
- data 操作对应的辅助数据；
- 返回值：void* 指针，表示查询结果。如果出错返回 NULL。个别协议正常退出也会返回空指针。

4 常见问题

4.1 编译器与二进制兼容性问题

使用 SDK 中的注册对象机制时，需要使用二进制兼容的编译器。在 Windows 系统下需要使用 MinGW 中的 GCC，而不能使用微软编译器（MSVC）或者是英特尔编译器。同样，需要使用版本兼容的 GCC 编译器。构建 SDK 使用的编译器版本如下：

- Linux：GCC 9.4.0

- Windows：GCC 11.2.0

编译插件时请尽量使用相同主版本的编译器，在一些情况下使用高版本的编译器编译的插件将不会工作。

4.2 BEX 函数入参的修改问题

在绝大多数情况下，不建议修改插件函数的入参，例如下面的代码可能会带来一些错误：

```
void my_fun(int nlhs, bxArray *plhs[], int nrhs, const bxArray *prhs[]){
    double * p = bxGetDoubles(prhs[0]);
    p[0] = 1; // 试图修改 p 的内容
}
```

由于写入时复制的机制，修改入参可能会意外修改了其它变量。如果想编写一个强制修改入参的函数，请确保入参没有被其它变量共享。

4.3 Getter 函数接口使用问题

由于 v3.0+ 引入了写入时复制机制，使用 bxGetXxxx 获取的数据指针有可能和其他变量是共享的。例如下面代码可能会意外修改其他的变量：

```
double *data = bxGetDoubles(ba);
data[0] = 1; // 将 ba 代表的变量第一个分量变为 1
```

这是由于 bxGetXxxx 没有对变量的数据共享情况做检查，直接返回了内部数据地址。因此在 BEX API v3.1+ 对 Getter 系列函数引入了额外两个变形：

- bxGetXxxxRO 以只读模式获取数据指针，返回的数据指针带有 const 修饰符，不允许修改；
- bxGetXxxxRW 以读写模式获取数据指针，如果数据被多个变量共享，那么先进行一次复制再返回复制之后的数据指针，对返回的指针指向的内容可以进行修改，且不会影响其他变量。

建议在情况允许时，尽量使用后缀为 RO 或 RW 的函数。旧版不带后缀的函数的默认行为同 RO 函数，但丢弃了返回值的 const 属性。因此，如果不涉及对返回值的修改，或是确定数据没有被共享（例如使用 bxCreateXxxx 生成），就可以继续使用旧版函数，否则应该使用后缀为 RW 的函数。

如果希望旧版函数的默认行为为 RW 函数，可以在引用头文件 bex.h 之前或在调用编译器时定义宏 BALTAM_BEX_LEGACY_GETTER_RW，此时 bxGetXxxx 会被映射为 bxGetXxxxRW。当不确定 bxGetXxxx 是否修改了数据时，定义该宏是一种确保现有代码不会出错的快速方

式，这是因为读写指针总是会在必要的情形下引入复制从而防止数据被错误修改。但对于代码中的只读情形，这种做法同样引入了不必要的复制行为，可能会降低代码效率。

附：提供两种 Getter 的函数列表，当使用表中的函数时，建议考虑只读/读写的情形。

bxGetInt8s	返回 8 位有符号整数类型矩阵数据首地址
bxGetInt16s	返回 16 位有符号整数类型矩阵数据首地址
bxGetInt32s	返回 32 位有符号整数类型矩阵数据首地址
bxGetInt64s	返回 64 位有符号整数类型矩阵数据首地址
bxGetUInt8s	返回 8 位无符号整数类型矩阵数据首地址
bxGetUInt16s	返回 16 位无符号整数类型矩阵数据首地址
bxGetUInt32s	返回 32 位无符号整数类型矩阵数据首地址
bxGetUInt64s	返回 64 位无符号整数类型矩阵数据首地址
bxGetDoubles	返回双精度实数类型矩阵数据首地址
bxGetSingles	返回单精度实数类型矩阵数据首地址
bxGetComplexDoubles	返回双精度复数类型矩阵数据首地址
bxGetComplexSingles	返回单精度复数类型矩阵数据首地址
bxGetChars	返回字符矩阵数据的首地址
bxGetLogicals	返回逻辑类型矩阵数据的首地址
bxGetField	按照名称获取字段
bxGetFieldByNumber	使用编号获取字段的值
bxGetCell	获取元胞数组中给定位置的元素
bxGetCellsV	（C++）获取元胞数组中所有元素
bxGetSparseDoubles	获取双精度实数稀疏矩阵数据指针
bxGetSparseSingles	获取单精度实数稀疏矩阵数据指针
bxGetSparseComplexDoubles	获取双精度复数稀疏矩阵数据指针
bxGetSparseComplexSingles	获取单精度复数稀疏矩阵数据指针
bxGetSparseLogicals	获取稀疏逻辑矩阵数据指针
bxGetIr	获取稀疏矩阵行指标数组
bxGetJc	获取稀疏矩阵列起始位置数组

4.4 静态变量使用问题

如果插件函数中含有静态变量，那么在调用过程中静态变量的值将会被保留。例如：

```
1 void foo(int nlhs, bxArray *plhs[], int nrhs, const bxArray *prhs[]){
2     static int i = 0;
3     bxPrintf("i = %d\n", i);
```

```
    ++i;
}
```

在插件被载入期间，使用北太天元命令行反复调用插件函数 `foo` 将会看到变量 `i` 在不同的调用中被保留了下来。静态变量在插件被卸载时才会被清除。同理，插件中其它位置的静态变量在插件被载入期间也会一直存在。

4.5 运算符重载机制变化问题

从 API v3.2 起，运算符重载机制变化为单一类型 ID 绑定机制，不再使用**所有**参数的类型 ID 作为重载函数的识别依据。因此，在注册算符时，无需指明这个算符具体需要多少个参数，每个参数的类型如何，仅仅需要指明这个算符以及相应的函数绑定到什么类型上就可以了。例如，执行 `bxRegisterOperator("point", "+", sid, add)` 表示将 `add` 函数作为算符 `+` 的实际实现和类 `sid` 进行绑定。这意味着执行 `a + b` 时，若 `a` 刚好为 `sid` 指定的类，北太天元会调用插件中的 `add` 函数进行加法操作。和旧机制相比，新机制下开发者不用列出所有 `sid` 可能的情况进行注册，而 `add` 函数的实现也可以更加灵活。

执行运算符时，北太天元会从左到右对参数进行测试，找到第一个支持该运算符重载的类并调用关联的重载函数。例如，执行 `[a, b, c, d]` 操作时，若 `b` 和 `d` 所在的类都重载了 `horzcat` 操作，那么实际会调用 `b` 所在类的重载函数进行执行。因此在使用运算符重载时要注意参数的顺序，避免调用到错误的重载函数。

需要注意，为了避免冲突，v3.2 起将不能重载内置类型的操作符，因此您不能将操作符绑定到内置类型的 ID 上。

5 更新日志

v3.3 (2023.5.9)

- 新增插件函数重载机制，见第 2.1.6 节；
- 增加了结构体相关的 API:
  `bxAddFieldAt`、`bxRenameField`
- 增加了插件函数重载相关 API:
  `bxOverloadFunction`、`bxReturnOverloadFailure`、
  `bxReturnOverloadFailureMsg`
- 文档术语改变：“Baltamatica” 统一修改为 “北太天元”。

v3.2 (2023.4.11)

- 变更了运算符重载的机制，见第 4.5 节；

- 增加了直接运行命令的函数：
  bxEvalString、bxCallBaltamatica、bxCallBaltamaticaF
- 新增了新机制下运算符重载函数：
  bxRegisterOperator、bxRegisterOperatorID
- 将如下函数标记为已过时：
  bxRegisterUnaryOperator、bxRegisterBinaryOperator、
  bxRegisterTernaryOperator
- 增加了 bxOperatorID 的两个新的枚举量：bxSUBSREF_OP 和 bxSUBSASGN_OP。

v3.1 (2023.3.24)

- 新增 bxGetXxxxRO 和 bxGetXxxxRW，用于支持写入时复制。请参考第 4.3 节；
- 新增 BEX 文件的使用说明。BEX 文件对标 MATLAB 的 MEX 文件，每一个文件包含一个 C/C++ 实现的函数，用于在北太天元中调用；
- 新增包装编译器 bex，用于辅助编译 BEX 文件或插件；
- 重写了文档第 1 节和第 2 节的部分内容；
- 修复了文档中的一些笔误。

v3.0 (2022.12.8)

- 从这个发布起，引入了 API 版本号的控制。引入了宏：
  BEX_API_VERSION_MAJOR、BEX_API_VERSION_MINOR；
- **特别注意：v3.0+ 与之前的版本不兼容，插件需要重新编译。**
- 新增了各种不同类型整数矩阵/数组相关 API，如 bxGetInt8s 等。正式支持 8 种不同类型的整数；
- 新增了高维数组的支持，如 bxCreateNumericArray、bxCreateCharArray 等。各类数值矩阵、字符矩阵、逻辑矩阵、字符串矩阵、结构体、元胞都已经支持高维；
- 对于结构体变量，增加了字段的编号，现在结构体的字段将以添加的顺序进行排列。相应地，增加了如下函数：
  bxGetFieldNameByNumber、bxGetFieldByNumber、
  bxGetFieldNumber、bxSetFieldByNumber；
- [**函数行为变更**] bxSetField 当字段不存在时，现在**不会**创建新字段。请在设置之前使用 bxAddField 创建字段；

- [**函数行为变更**] bxDuplicateArrayS 现在的使用无任何限制；
- 文档中的术语：“单循环索引”统一变更为“线性索引”；
- 文档的最后一节改为对常见使用问题的整理；

2022.8.29

- 文档术语相关修正；
- BEX 文件的功能尚未完成，先去掉相关文档；

2022.7.4

- 新增函数句柄相关读取函数：
  bxIsFunctionHandle、bxGetFunctionHandleType、bxGetFunctionHandleData；
- 新增函数句柄类型的枚举类型：bxFHandleType。

2022.6.2

- 新增稀疏逻辑矩阵相关函数：
  bxCreateSparseLogicalMatrix、bxIsSparseLogical、bxGetSparseLogicals；
- 新增有关稀疏矩阵内部存储格式的说明，修改了对 jc 数组的描述以防引起误解；
- 修复 BUG：bxGetSparseXxxxs 在类型不对时会正确返回空指针。

2022.5.20

- 新增内部查询函数：bxF2KQuery、bxK2FQuery。

2022.5.15

- 新增了插件函数可以使用命名空间的说明。见命名空间。

2022.5.14

- bxIsDouble、bxIsSingle、bxIsComplex 移动到基本属性下，并作更详细的说明；
- 新增更具体的类型判断函数：
  bxIsRealDouble、bxIsRealSingle
  bxIsComplexDouble、bxIsComplexSingle
  bxIsSparseRealDouble、bxIsSparseRealSingle
  bxIsSparseComplexDouble、bxIsSparseComplexSingle

2022.5.11

- 文档组织结构调整：现在 API 中的函数将按照用途排列，并在每个子类前增加子目录；
- 文档增加了索引，便于按照 API 名称查询；
- 修改了文档的单位和时间格式；
- 修复了文档中的笔误；
- 增加了稀疏矩阵的 API；
- 修改 `bxArray` 定义方式，现在 `bxArray` 是个不透明的类型，不能直接生成变量，必须通过接口函数获得。

2022.3.23

- 增加了 `bxTypeCStr` 的说明以及变更（不建议用来判断类型）。

2022.3.15

- 内核的 string 改为 string array，增加了相关 API：
  `bxCreateStringMatrix`（创建空字符串数组）
  `bxCreateStringMatrixFromStrings`（创建指定内容的字符串数组）
  `bxGetString`、`bxSetString`（获取与修改字符串数组的分量）
  `bxGetStringLength`（获取给定分量的字符串长度）
  `bxCreateStringScalar`（创建字符串标量）
- 将以下函数标记为过时：
  `bxGetStringDataPr`（被 `bxGetString` 替代）
  `bxCreateStringObj`（被 `bxCreateStringScalar` 替代）
  `bxSetStringFromCStr`（被 `bxSetString` 替代）
  `bxGetStringLen`（被 `bxGetStringLength` 替代）
- 以下函数行为发生改变：
  `bxGetStringPr`：现在返回的是字符串数组的首地址。

2022.2.11

- 增加了和元胞数组（cell）相关的 API：
  `bxCreateCellMatrix`（创建元胞数组）
  `bxIsCell`（判断是否为 cell）
  `bxGetCell`、`bxSetCell`（获取与修改分量）
  `bxGetCellsV`（获取全部元素至 vector）

- 增加了一个辅助函数：bxCalcSingleSubscript，用于产生 bxGetCell、bxSetCell 的指标参数；
- 更改大小的函数现在可以支持 cell；
- bxClassID 中增加了结构体和元胞数组的定义，bxGetClassID 也可以正确返回结构体和元胞数组的类型 ID，而不会返回 bxUNKNOWN_CLASS；
- 明确了一处下标的说明（zero-based 或 one-based）。

2022.1.12

- 增加了和结构体相关的 API：
  bxCreateStructMatrix（创建结构体）
  bxGetField、bxSetField（获取与修改字段）
  bxGetNumberOfFields（获取字段总数）、bxGetFieldNames（获取所有字段，C++）
  bxRemoveField（删除字段）
- 更改了 bxArray* 接口的相关约定，文档中增加了对 bxArray* 类型的解释；

2021.12.18

- bxGetStringDataPr 的返回值变为 const char *，即为只读属性，不允许修改；
- 增加了修改与创建的 API：
  bxSetM、bxSetN、bxResize
  bxCreateCharMatrixFromStrings（创建多维字符数组）
  bxSetStringFromCStr（更合理的修改字符串的函数）；

2021.11.27

- 增加了和 String 相关的 API：
  bxGetStringDataPr、bxGetStringLen、bxGetStringPr、bxCreateStringObj；
- bxGetM 和 bxGetN 在对象不是矩阵时返回值为 -1（之前描述为 1）；
- 修改插件函数异常的处理（不建议使用异常）；
- 修改插件的默认存储位置；

2021.11.18

- 增加了 API：bxErrMsgTxt；
- 例子 vector 的 bv_at 返回索引的拷贝，并加入了 bxErrMsgTxt 的使用说明；

- bxDuplicateArrayS 变更：不能生成 plhs[] 数组中的对象；
- 增加了可以设置自定义数据类型名的标准；
- 添加了 dup() 函数实现的细致说明；
- 因为不方便维护，删除文档中完整例子的代码，改为指明 SDK 中的路径；
- 编译插件时 GCC 版本的说明（见“技术细节”一节）

2021.10.10

- 增加了 API：bxAsCStr、bxAsString；

2021.10.4

- 增加了更多重载运算符；
- 将缺省重载相关的接口接收的输入变为 const char *，以 bxOperatorID 为输入的接口更名为 bxRegisterXxxxOperatorID；

2021.9.29

- 增加了重载内置算符的标准；
- 增加了和重载相关的三个 API；

2021.9.24

- 增加了若干 API：
  bxIsXxxx、bxAsInt、bxCreateXxxxScalar、bxPrintf、bxGetClassID；
- 增加了返回值 nlhs 设置标准，处理 ans 变量的情形；

索引