文件输入输出

Fortran: 需要包含头文件

mpiof.h

文件：和进程组相关联，所有进程必须同时对文件进行操作；

起始位置（displacement）：字节数；

基本单元类型（etype）：位移非负且单调上升；

文件单元类型（filetype）：在基本单元类型上派生出来；

文件视窗（view）：

位移（offset）：相对于起始位置，以基本单元类型为单位；

文件大小：文件的总字节数；

文件指针（file pointer）：独立文件指针，共享文件指针；

文件句柄（file handle）：

打开 MPI 文件

```
int MPI_File_open(MPI_Comm comm, char * filename,
                  int amode, MPI_Info info, MPI_File * fh);
```

comm：通信器，所有进程同时调用；
filename：文件名；
amode：打开方式
MPI_MODE_RDONLY, MPI_MODE_RDWR,
MPI_MODE_WRONLY,
MPI_MODE_CREATE, MPI_MODE_EXCL,
MPI_MODE_DELETE_ON_CLOSE, ...
fh：文件句柄

关闭 MPI 文件

```
int MPI_File_close(MPI_File * fh);
```

所有进程必须同时调用；

设定文件长度

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size);
```

文件大小将变成 size 个字节，但是存储设备上不一定已经分配了空间;

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size);
```

文件大小至少成为 size 个字节，并分配了存储空间;

这两个是聚合型函数。

设定文件视窗

```
int MPI_File_set_view(MPI_File fh,
                      MPI_Offset displacement,
                      MPI_Datatype etype,
                      MPI_Datatype file_type,
                      char * datarep,
                      MPI_Info info);
```

聚合型函数

datarep: native, internal, external32

同步方式	进程协同方式	
	非聚合	聚合式
阻塞型	MPI_File_xxxx_at	MPI_File_xxxx_at_all
非阻塞和分裂型	MPI_File_ixxxx_at	MPI_File_xxxx_at_all_begin MPI_File_xxxx_at_all_end
阻塞型	MPI_File_xxxx	MPI_File_xxxx_all
非阻塞和分裂型	MPI_File_ixxxx	MPI_File_xxxx_all_begin MPI_File_xxxx_all_end
阻塞型	MPI_File_xxxx_shared	MPI_File_xxxx_ordered
非阻塞和分裂型	MPI_File_ixxxx_shared	MPI_File_xxxx_ordered_begin MPI_File_xxxx_ordered_end

显式位移阻塞型读写

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset,
        void * buffer, int count, MPI_Datatype datatype,
        MPI_Status * status);

int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
        void * buffer, int count, MPI_Datatype datatype,
        MPI_Status * status);
```

如果文件用 MPI_FILE_SEQUENTIAL 方式打开，则不能使用这种方式读写。

独立指针阻塞型读写

```
int MPI_File_read(MPI_File fh, void * buffer,
        int count, MPI_Datatype datatype,
        MPI_Status * status);

int MPI_File_write(MPI_File fh, void * buffer,
        int count, MPI_Datatype datatype,
        MPI_Status * status);
```

共享指针阻塞型读写

```
int MPI_File_read_ordered(MPI_File fh, void * buffer,
        int count, MPI_Datatype datatype,
        MPI_Status * status);

int MPI_File_write_ordered(MPI_File fh, void * buffer,
        int count, MPI_Datatype datatype,
        MPI_Status * status);
```

移动文件指针

```
int MPI_File_seek(MPI_File fh,
                  MPI_Offset offset,
                  int whence);

int MPI_File_seek_shared(MPI_File fh,
                  MPI_Offset offset,
                  int whence);

whence: MPI_SEEK_SET, MPI_SEEK_CUR, MPI_SEEK_END
```

子数组数据类型

```
int MPI_Type_create_subarray(int ndims,
        int array_of_sizes[], int array_of_subsizes[],
        int array_of_starts[], int order,
        MPI_Datatype oldtype, MPI_Datatype * newtype);
```

ndims: 维数;
array_of_sizes: 整体数组大小;
array_of_subsizes: 本子数组大小;
array_of_starts: 起始位置;
order: 存储方式(Fortran or C)
old_type: 数组元素类型;
new_type: 新得到的子数组数据类型;

分布式数组

```
int MPI_Type_create_darray(int n_rank, int rank,
      int global_sizes[], int distributes[],
      int args[], int dim_processors[], int order,
      MPI_Datatype old_type, MPI_Datatype new_type);
```

n_rank: 进程总数;
rank: 本进程的秩;
global_sizes: 数组的总体大小;
distributes: 分布方式(MPI_DISTRIBUTION_BLOCK,
MPI_DISTRIBUTION_CYCLIC,
MPI_DISTRIBUTION_NONE)
args: MPI_DISTRIBUTE_DFLT_DARG
dim_processors: 每一维的进程个数;
order: 每个进程上的数组的存储方式(Fortran or C)
old_type: 数组元素的数据类型;
new_type: 新构造的分布式数组数据类型;

须参考HPF分布式数据结构

POSIX信号

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum,
                    sighandler_t handler);
```

POSIX 线程

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);

#include <sched.h>
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg);

#include <pthread.h>
int pthread_create(pthread_t * thread,
          pthread_attr_t * attr,
          void * (*start_routine)(void *),
          void * arg);
```

External Data Representation(XDR)

从文件指针创建：

```
void xdrstdio_create(xdrs, file, op)
        XDR *xdrs;
        FILE *file;
        enum xdr_op op;
```

xdr_op: XDR_ENCODE(输出), XDR_DECODE(输入)

销毁XDR对象：

```
void xdr_destroy(xdrs)
        XDR *xdrs;
```

```
#include <rpc/rpc.h>


... ...
FILE * fp =fopen("filename", "wb");
XDR xdrs;
xdrstdio_create(&xdrs, fp, XDR_ENCODE);
... ... /// 写出数据
xdr_destroy(&xdrs);
fclose(fp);
... ...
```