PFS学习文档

---------------汪伟 2011年7月28日

# 什么是PFS

PFS是Packed File System的简称，是一种封装各种类型，然后以统一方式访问的文件系统。PFS(Packed File System)系统借用虚拟文件系统(VFS)的概念,将异构的文件系统实现抽象为统一的系统模型为上层使用.也可以叫做

UFS - Unified File System (统一文件系统)

GFS - Game File System (游戏文件系统)

GUFS–Game Unified File System(游戏统一文件系统)

# 为什么需要PFS

**要解决的问题：**

游戏的文件系统的需求模型如下：

1. 需要能够使用一致的接口访问如下类型的文件系统:
   1. 原始磁盘文件目录(操作系统文件系统,开发阶段(Windows文件系统比较常用))
   2. 包裹文件(合成,变换,压缩,加密(不常用))(安装/发布版的游戏资源文件)
   3. 网络文件(游戏文件更新)
2. 外围辅助工具开发和游戏运行遵从一致的资源访问规范
3. 与XML I/O无缝结合

**解决方案：**

为了解决上述问题，能让游戏和外围设备以统一的文件路径规范来访问不同的文件系统，于是使用PFS来定义统一的路径，以 ‘/’符做路径的根，同时也是路径结点分隔符, 内部统一转换为小写,称为 PFS路径。

# PFS的文件系统

为访问本地磁盘目录，包裹文件，网络文件这几种不同的文件系统，目前PFS一共有4种文件系统，来对应这些文件系统

**PFS中的文件系统基础架构**

PFS中一共有4种不同的文件系统：CNativeFS， CZipFS，CRunzipFS，CBriefFS，他们都继承自文件系统基类CPFSBase，这4种文件系统分别针对4种不同的设备：本地目录（Native），Zip包，PFS包（Runzip），网络文件（Brief）。

关系图如下：

对应本地目录（可读写）

对应网络文件（只读）

对应PFS格式（只读）

对应Zip格式（可读写）

CBriefFS

CRunzipFS

CZipFS

CNativeFS

CPFSBase

# 如何使用PFS

## 如何打开PFS系统文件

既然我们把资源文件都封装在了PFS数据包中，那么在游戏中或者其他外围工具中如何来使用PFS文件系统中的资源文件呢？ PFS通过一个Mount系统，使用统一的路径格式来访问PFS中的文件。首先看看PFS的Mount系统：

### Mount系统简介

我们通过定义一个Mount点和Mount设备的映射表，来做一个PFS路径到本地路径的映射，在使用时，通过PFS路径前缀“/”来识别Mount设备。

例如：

D:/Game/ /run

/run为Mount点，D:/Game即是Mount设备， Mount之后即可通过/run来访问D:/Game下的文件了，如:

/run/data.pfs 即等同于 D:/Game/data.pfs

还可以在Mount后的基础上进一步做Mount， 即以一个Mount点为Mount设备，再进行Mount，例如：

/run/cfg /cfg

以/cfg为Mount点，/run/cfg为Mount设备， 那么Mount之后/cfg即等同于D:/Game/cfg

### 如何使用Mount

PFS中Mount通过CEnv中提供的Mount接口来实现，例如：

CEnv::Mount( L"/run", L"d:\\temp", FST\_NATIVE, MT\_WRITABLE)；

CEnv::Mount( L"/run", L"d:\\temp \\data.zip", FST\_ZIP, MT\_WRITABLE)；

CEnv::Mount( L"/ui", L" d:\\temp \\data.pfs", FST\_RUNZIP, MT\_RUNTIME );

CEnv::Mount( L"/b", L"http://172.16.32.30/mhzx", FST\_BRIEF, MT\_READONLY | MT\_RUNTIME )；

其中参数FST为文件系统类型，共有4种，分别对应前面所讲的4种文件系统

**FST\_NATIVE**----------------- 本地文件系统

**FST\_ZIP**--------------------Zip格式包

**FST\_RUNZIP**-----------------pfs格式包

**FST\_BRIEF**------------------网络文件系统

最后一个参数为Mount方式

对于同一种设备，可有多种不同的Mount方式。

PFS中Mount方式有如下几种：

MT\_READONLY 只读

MT\_WRITABLE 可写

MT\_RUNTIME 运行时系统，只读，不加载metainfo

MT\_CACHEDATA 缓存文件数据（主要用于RUNZIP+RUNTIME组合下的特殊类型）

MT\_ZIPUPDATE update的数据为zip过的（主要用于pfs包更新用，与MT\_RUNTIME互斥），隐含MT\_WRITABLE。

MT\_OPTIONAL 该设备可有可无，没有时忽略此设备。(当前只用于MountManager)

MT\_TRUNCATE 将底层的文件系统在mount时清空，对于zip包系统，清空zip包文件，对于目录，清空目录下所有文件。隐含MT\_WRITABLE。

MT\_PREFERDIR 如果底层文件系统不存在，CDevice强制创建Directory系统，而非创建PFS包系统。隐含MT\_WRITABLE。

MT\_FSTYPEFLAG 最高位为1,显式设定的FS\_TYPE类型，如果为0，则自动识别。

MT\_FSTYPEMASK 当MT\_FSTYPEFLAG置1时，次高7位为显式设定的FS\_TYPE类型；MT\_FSTYPEFLAG为0时，不当做FS\_TYPE解释。

### 如何打开PFS系统中的文件

前面已经讲解了PFS的Mount系统，那么在Mount之后如何打开和使用PFS中的文件资源呢？PFS在CEnv中提供了上层的接口：

static CBaseFile\* OpenFile( const std::wstring& filename, FILE\_MODE fm, FILE\_ACCESS fa);

其中的FILE\_MODE为打开模式，FILE\_ACCESS为访问方式。

PFS中打开一个文件时可以提供不同的模式和访问方式的参数：

模式参数FileMode有3种：

FM\_EXCL //文件必须存在

FM\_CREAT //创建文件

FM\_TRUNC //清除文件内容

访问方式FileAccess也有3种：

FA\_RDONLY //只读 ReadOnly

FA\_WRONLY //只写 WriteOnly

FA\_RDWR //可读写 ReadAndWrite

要打开一个PFS文件，首先将文件所在的目录或包Mount到PFS系统中，然后就可以通过PFS在CEnv中提供的OpenFile接口来打开其中的文件了。

例如：

CEnv::Mount(L"/map", L"D:\\game\\map.pfs", FST\_RUNZIP, MT\_READONLY); //只读

CBaseFile\* basefile = CEnv::OpenFile(L"/map/map001.map.xml", FM\_EXCL, FA\_RDONLY);

对于本地文件，如果不做Mount，也可以直接使用本地路径，通过CNativeFile直接Open一个本地文件

例如：

CNativeFile nf;

nf.Open(L"d:\\device.pfs", FM\_EXCL, FA\_RDONLY);

或者使用CFile来Open一个文件，CFile支持本地路径和PFS路径

例如：

CFile file;

file.Open(L"d:\\test.pfs", FM\_EXCL, FA\_RDONLY);

或 file.Open(L"/run/test.pfs", FM\_EXCL, FA\_RDONLY);

在打开过程中，CFile会把PFS路径根据Mount的记录转化为本地路径。

# PFS数据包

PFS系统中，各种文件数据是封装在一个pfs包中的，那么pfs包是如何封装这些数据的呢？

## PFS数据包基本结构

PFS数据包由两部分组成：数据部分和文件尾部。

PFS包结构图：

尾部

数据部分

数据部分

即是PFS包的每个文件数据保存的地方，在PFS包的前面。

尾部

PFS包的尾部信息，包括前面所有文件的Central信息和一个标志结束的DirCentral。

尾部信息相当于PFS包的目录，通过尾部信息可以迅速找到PFS包中的文件。

那么在PFS包中文件是如何保存的呢？我们先要了解一下PFS包中的文件的存在形式。

**PFS中的文件的存在形式**

在PFS包中，每个文件可以分为3部分：Header ，Data ，Central 。

**Header部分**是头部描述信息，记录文件的基本信息，包含文件名称，压缩后的数据大小，原始数据大小，修改时间，压缩方式，CRC32校验码等一些基本信息

**Central部分**也是记录文件信息的，但Central包含了Header中的信息，同时Central记录了更多详细信息，包括该文件的版本，文件头偏移量（Offset），文件属性（为兼容ZIP而存在，PFS中未使用）。

**Data 部分**是文件本身的数据部分，写入PFS包中的文件数据需要先经过encode处理（例如ZIP压缩），根据不同的文件格式会有不同的encode处理， 当然，也可以设置不对原始数据进行encode处理，直接保存原始数据。

对于每个文件，这三个部分中Header和Data保存两个部分存储在PFS包的数据部分，而Central部分保存在PFS包的尾部。

### 文件在PFS包内的具体存储方式

我们已经知道，PFS包中一个文件是由Header + Central + Data三部分组成，那么这三部分如何存储在PFS包中呢？

在PFS中，Header部分记录了文件的一些必要信息，如文件名称，原始数据大小，压缩后的数据大小，压缩方式等，Central部分包含了Header部分的信息，但还记录了Header的偏移量（HeaderOffset），也就是Header在文件中的位置。我们已经知道，一个文件的Header和Data是保存在PFS包前面的数据部分，而Central是保存在后面的尾部，因为Central中记录了Header的位置，因此，尾部相当于PFS包的目录，根据文件名称，在尾部找到Central，然后根据Central中记录的Header位置，即可找到Header，再根据Header中记录信息，即可读取Data数据，这就是从PFS包中读文件的基本思想。那具体实现细节是怎样的呢？我们先来看看文件是怎样写入的。

当一个文件写入PFS包时，首先要写的是Header信息，也就是把Header里记录的数据按顺序序列化到 PFS包中，但在写入Header之前，我们需要先写入一个标识签名，以标识后面的内容是一个文件的Header，该签名是一个格式标识符，格式标识符有哪些呢？

PFS中有两类这种标识符

一类为zip格式：

SIGNATURE\_ZIP\_FILE\_HEADER = SIGNATURE\_FILE\_HEADER, // 别名

SIGNATURE\_ZIP\_FILE\_CENTRAL = SIGNATURE\_FILE\_CENTRAL,

SIGNATURE\_ZIP\_DIR\_CENTRAL = SIGNATURE\_DIR\_CENTRAL,

SIGNATURE\_FILE\_HEADER = 0x04034b50,

SIGNATURE\_FILE\_CENTRAL = 0x02014b50,

SIGNATURE\_DIR\_CENTRAL = 0x06054b50

一类pfs格式：

SIGNATURE\_PFS\_FILE\_HEADER = (DWORD)'HSFP', // PFSH

SIGNATURE\_PFS\_FILE\_CENTRAL = (DWORD)'FSFP', // PFSF

SIGNATURE\_PFS\_DIR\_CENTRAL = (DWORD)'DSFP', // PFSD

其中ZIP类标识与标准ZIP包格式相同，这样做的目的是为了与标准ZIP兼容，这样以Zip格式写的PFS包可以直接被标准Zip工具打开，而PFS标识则是我们PFS系统中自定义的，只能以PFS的方式打开。

在写入标识符后，就可以将Header信息序列化到包中了，Header中有以下内容：

解压的zip版本 unsigned short 2字节

设置标识 unsigned short 2字节

压缩方式 unsigned short 2字节

修改时间 unsigned short 2字节

修改日期 unsigned short 2字节

CRC32标识符 unsigned int 4字节

原始数据大小 unsigned int 4字节

压缩后的数据大小 unsigned int 4字节

文件名称长度 unsigned short 2字节

说明文字长度 unsigned short 2字节

在以上内容依次写入包中后，再写入名称，然后是说明文字，这样，Header部分就写完了。

之后需要将文件的数据经过encode处理后写入包中。 Encode的方式在Header中记录。数据经Encode处理完后即是压缩数据（compressed data），从Header后面继续写入，数据开始写入的地方称为DataOffset，其值为HeaderOffset的值加上Header SIGNATURE和Header所占的空间，即DataOffset = HeaderOffset + sizeof(SIGNATURE ) + sizeof（FileHeader），注意，该sizeof并非sizeof运算符，而是表示Header中序列化到包中的字节数，也就是26字节固定大小+文件名称+说明文字；

在数据部分写完后，接着写下一个文件，写的方式与前面相同。下一个文件的HeaderOffset从前一个文件的结束处开始，即下一个文件的HeaderOffset 值为上一个文件的DataOffset + DataSize.

在文件都写完后，PFS包的数据部分即完成了，其结构如下：

**PFS包中数据部分的文件存储结构示意图：**

DataOffset

HeaderOffset

File2

File3

File1

Next File Header

SIGNATURE

FileData

Header SIGNATURE

FileHeader

接下来开始写PFS包的尾部

PFS包的尾部信息，包括前面所有文件的Central和一个标志结束的DirCentral。

在前面写入每个文件时，会把文件名和相应的Central依次记录下来，在写入尾部信息时，再依次将每个文件的Central都依次写入其中

在写入Central之前，也必须先写一个Central的标识符 SIGNATURE\_ZIP\_FILE\_CENTRAL（或SIGNATURE\_PFS\_FILE\_CENTRAL），然后将Central中的数据写入。Central中包含前面Header中的信息，但还会多出以下几个信息：

版本号 unsigned short 2字节

DiskNumberStart unsigned short 2字节

内部文件属性 unsigned short 2字节

外部文件属性 unsigned int 4字节

Header位置 unsigned int 4字节

文件说明文长度 unsigned int 4字节

Central写入的过程与Header类似，在写入前面的固定部分长度后，再写入名称和说明文字，完成后就可以写下一个文件的Central。

全部的Central都写完后，最后再写一个Dir Central，作用是标识PFS的结束，同时里面也记录了Central部分开始的位置，这样我们可以快速找到Central部分的位置，Dir Central的写入过程与Header和Central一样，也需要先写一个标识符Dir SIGNATURE，然后再写入Dir Central的数据，不过一个PFS包只会有一个Dir Central。

完成后，尾部会有如下结构：

**PFS包的尾部结构示意图：**

File3

File2

File1

End

Dir SIGNATURE

CDirCentral

FileCentral

Central

SIGNATURE

至此，PFS包就写入完成了。

读的过程可以认为就是写的逆过程了，从后面开始，先根据Dir SIGNATURE找到Dir Central，从Dir Central得到Central部分的位置，然后根据Central的SIGNATURE 解析Central部分，得到所有文件的目录，然后就可以从Central中的Offset找到文件的Header，从Header的SIGNATURE开始，按Header写入的顺序依次读取，就可以得到文件的数据了。

## Packzip打包工具

目前我们使用的PFS数据包是由我们编写的一个PackZip工具来生成，其用法可参阅pfs工具简单使用手册.docx。

Packzip的打包就是根据以上所介绍的PFS包写的思想来做的，具体细节这里不再赘述，下面讲解一下大概的过程和需要特殊处理的地方。

Packzip在生成PFS数据包时，首先把需要打包的文件添加到列表，然后从列表中的第一个文件开始，依次按前面讲述的步骤写入所有文件。但打包的过程中可能会有子目录或子包出现，这些需要特殊处理。

当有子目录存在时，首先写入子目录的Header信息，然后递归遍历子目录下的所有文件和子目录，直到将所有子目录的文件都写入PFS包中。

如果有子包存在，需要将子包合并到当前包内，我们需要先看看Packzip生产的数据包中的几个特殊文件。

这几个文件称为meta信息文件，是根据当前游戏需求在PFS包中添加的，作用是记录PFS包的各种信息。

meta信息文件包括“.files.meta”, “.files2.meta”, “.setup.meta”, “.version.meta”四个文件。

**.files.meta**是一个二进制文件清单，记录PFS包中所有文件，包括所有子包内文件的信息，包括名称，大小，压缩方式，hash值，编解码器TSID等信息， 相当于PFS内文件的一个详细信息列表（参考windows详细信息查看方式），但不包括files.meta和files2.meta文件。

**.files2.meta**与files.meta一样，也是一个二进制文件清单，但它只记录所有当前目录下的文件信息，不包括子包的信息（子包中会有自己的file2.meta文件来记录自己的文件信息）。其记录的信息与.files.meta相同，但会多出两个值，即size2和hash2，这两个值是该文件经encode处理后的文件的size和hash值。

**.setup.meta**是一个xml文件，这是一个安装包描述文件，包含子包文件信息，在安装时根据此文件中的记录来判断是生产pfs包还是目录，每个子包信息中包含两个Hash值，其作用是用来判断包内文件是否有更新。

**.version.meta**是版本记录文件，记录了版本信息和更新方式及服务器的url等信息。

这几个文件的写入和其他文件一样，位置在其他文件后面，即在需要打包的文件都写完以后，才写入这几个文件。

在生成包之前需要先设定该PFS包的版本，可更新版本，更新方式和服务器url等信息。然后开始写入文件列表中的所有文件，完成后开始写meta信息文件时，首先根据子包记录生成setup.meta文件（在添加文件到列表时，可以设定是否是子包，如果是，那么就会记录在setup中），然后根据版本和url等设置信息生成version.meta文件

最后是生成.files.meta和.files2.meta两个文件，这两个文件中会包括setup.meta和version.meta，但不会包括本身。

.files.meta是根据pfs中的文件信息记录映射表（CMetaInfoMap）来生成的，在前面写入文件时，每写入一个文件就会将此文件的name和记录该文件信息的CMetaInfo添加到该map中。.files2.meta则是根据CMetaInfoMap的一个备份m\_bakAllMetaInfos来生成的，该备份会删除所有子包中的文件信息，因为子包中会有独立的file2.meta文件来记录。

合并子包的主要过程如下：

1. 写入子包的根目录信息
2. 读取子包setup.meta并记录到SetupMetaVector中
3. 读取子包的files.meta信息文件，得到子包的文件信息
4. 删除子包中的meta信息文件
5. 将子包的其他非meta文件写入当前包
6. 合并meta信息文件

合并后子包变成了当前包的一个子目录，子目录下不包含meta文件，包内文件记录在根目录下的files2.meta文件中，如果生成包时设置了某个目录是子包（Ispak属性），则合并后成为当前包的一个子包，与子目录不同，子包有自己的files2.meta文件来记录包内文件的信息，而不会记录在PFS包根目录下的files2.meta内。

最后，在PFS包结束时会将所有文件的Central信息序列化到包的最后，然后写入一个CDirCentral以表示结束。

# PFS的内部实现

## PFS中的文件系统

前面已经讲了PFS中的4种文件系统：

对应本地目录（可读写）

对应网络文件（只读）

对应PFS格式（只读）

对应Zip格式（可读写）

CBriefFS

CRunzipFS

CZipFS

CNativeFS

CPFSBase

基类CPFSBase中有以下数据成员：

CRefMountFSSet m\_rmfs;

CMetaInfoMap m\_MetaInfos;

bool m\_bMetaChanged;

std::wstring m\_mountpoint; // 由SetMountPoint()初始化

MOUNT\_TYPE m\_MountType; // 保存initialize()的参数mt

std::wstring m\_deviceName; // 保存initialize()的参数device

std::wstring m\_nativeDevicePath; // 由CalcNativeDevicePath()初始化

CPFSBase\* m\_BasePFS; // 本PFS的父PFS，如果本PFS为根PFS，那么此值为NULL

CPFSBase中与文件打开相关的接口：

virtual CPFSBase\* Clone() const = 0;

virtual void Delete() { delete this; }

virtual int Initialize( const std::wstring& device, MOUNT\_TYPE mt) = 0;

virtual FS\_TYPE GetFSType() const = 0;

virtual int OpenFile( const std::wstring& filename, FILE\_MODE fm, FILE\_ACCESS fa, CBaseFile\*& file) = 0;

virtual int RemoveFile( const std::wstring& filename) = 0;

virtual int CreateDirectory( const std::wstring& path, bool bFailIfExisting = false )

virtual int RemoveDirectory( const std::wstring& path, bool bFailIfNotEmpty = false )

virtual bool IsDirectoryExisting( const std::wstring& path ) = 0;

virtual bool IsFSBusy() const = 0;

virtual bool IsFileExisting( const std::wstring& filename) = 0;

virtual void OnFileClose(CBaseFile\* pFile) {};

在4种文件系统类中，会根据自身的文件系统特性实现基类中的这些接口。

## Mount如何实现

首先要了解一下PFS中的CPacketFileSystem类。

CPacketFileSystem是PFS中上层接口与底层实现的通道，上层接口通过调用CPacketFileSystem的接口来实现。

CPacketFileSystem有一个Map记录Mount的信息。

Mount过程是由CPacketFileSystem的Mount接口来实现。CPacketFileSystem有一个封装器 - CPacketFileSystemWrapper，该类有一个静态变量实例（保证CPacketFileSystem对象的生存期），每次Mount都是通过该实例的Mount函数来完成。在Mount时会根据文件系统类型（FS\_TYPE）参数生成相应的文件系统对象，创建文件对象时会将Mount的name及相应的CPFSBase指针记录在CPacketFileSystem的一个map（ m\_MountPoints）中，然后对该对象作初始化，初始化Initialize是CPFSBase的一个虚函数，在不同的文件系统继承类中有不同的实现，初始化会将Mount的参数记录在相应文件对象中的数据成员里。

基类CPFSBase中与Mount相关的数据：

std::wstring m\_mountpoint; // 由SetMountPoint()初始化

MOUNT\_TYPE m\_MountType; // 保存initialize()的参数mt

std::wstring m\_deviceName; // 保存initialize()的参数device

std::wstring m\_nativeDevicePath; // 由CalcNativeDevicePath()初始化

CPFSBase\* m\_BasePFS; // 本PFS的父PFS，如果本PFS为根PFS，那么此值为NULL

以一个Native的Mount为例：

CEnv::Mount( L"/run", L"d:\\temp", FST\_NATIVE, MT\_WRITABLE)；

Mount之后，生成一个新的CNativeFS对象，其m\_mountpoint为"/run"，

m\_MountType为MT\_WRITABLE， m\_deviceName为"d:\\temp"，

m\_nativeDevicePath记录本地路径，由CalcNativeDevicePath()反向递归计算得到，若m\_deviceName为本地路径，说明为第一层Mount，m\_nativeDevicePath的值即为m\_deviceName，本例中为"d:\\temp"，若为PFS路径（以“/”开头），则通过\_CalcNativeDevicePathForChildPFS计算上一层Mount，由上层Mount点找到Native路径，再与本层Mount的device结合即可得到全路径。

m\_BasePFS在本例中为NULL，若为PFS路径，则通过开始的路径去Mount点的map里查找相应的CPFSBase。

以一个PFS路径的Mount为例：

CEnv::Mount( L"/test", L"/run/test", FST\_NATIVE, MT\_WRITABLE)；

则新的CNativeFS对象中，m\_mountpoint = L"/test",

m\_MountType为MT\_WRITABLE， m\_deviceName为L"/run/test"，

由于device为PFS路径，则首先根据路径前面的“/run”去查找BasePFS，根据

BasePFS可以得到m\_nativeDevicePath为"d:\\temp",所以即可得到本PFS中全路径为"d:\\temp\\test".

CEnv还提供了PFS路径和本地路径相互转换的接口：

NativeFileNameToPFSFileName

PFSFileNameToNativeFileName

通过这两个接口可以实现PFS路径和本地路径的相互转换。

**Device**

Device是一类特殊的Mount，将某个目录或包文件Mount到一个CDevice对象上，之后可通过device内部路径直接对文件进行访问和操作，不再需要提供完全路径。一个Device相当于一个独立的系统，外部需要device里的文件时，通过device的接口，根据文件在device内的路径去读取，而不是通过device外的路径。

例如：

CDevice device;

device.Mount(L"d:\\test\\a.pfs", MT\_WRITABLE);

CDeviceFile df;

df.Open(&device, L"a.txt", FM\_CREAT, FA\_RDWR);

df.Close();

device.RemoveFile(L"a.txt");

Device也有多种不同类型：

CWindowsDirectoryDevice（FST\_NATIVE）

CPackageFileDevice（FST\_ZIP）

CRuntimePackageFileDevice（FST\_RUNZIP）

CMountDirectoryDevice（混合型）

在做Device Mount的时候，会创建一个Device对象，可能为以上任意一种，在DeviceFactory中，会根据Mount的路径来判断Device的类型：

如果是PFS路径，那Device可能为包类型，也可能为目录类型，因此，创建混合型Device，混合型能兼容目录和包类型。

如果是Native路径，并且目标文件存在，则创建CPackageFileDevice类型，

如果目标是存在的目录，则创建CWindowsDirectoryDevice类型，

如果Device目标不存在，则判断Mount方式是否优先创建目录系统（MT\_PREFERDIR方式），如果是，就创建CWindowsDirectoryDevice，如果不是，根据文件名是否带有后缀选择创建CWindowsDirectoryDevice目录还是创建CPackageFileDevice包。

需要注意的是混合型Device，在Mount的时候，默认情况下对应FST\_NATIVE类型，但如果Device目标存在，且Mount方式为只读，则转换为Runzip类型，若Mount方式为可写，则转化为Zip类型，若Device目标不存在，则为FST\_NATIVE类型。

在Device的Mount过程中，如果Mount的是PFS路径，且为只读方式，则把PFS路径转换成本地路径，因为将PFS路径转换为本地路径后，从本地路径创建的Device将与其他PFS无关，可以减少对象之间的关联性。对于只读的Mount类型可以这么做，但对可写的类型，则不需要这么做，因为PFS系统目前不支持共享写操作。

判断流程图如下：

Yes

IsPFSPath?

CMountDirectoryDevice

No

CPackageFileDevice

Yes

IsFileExisting?

No

IsDirectoryExisting?

CWindowsDirectoryDevice

Yes

No

Yes

IsMountPreferDir?

CWindowsDirectoryDevice

No

CPackageFileDevice

## PFS系统中文件的读写

打开一个PFS系统中的文件是通过PFS的OpenFile接口来打开。

OpenFile也是通过CPacketFileSystem来实现的，在打开一个指定的文件时，首先根据文件路径，从根路径开始逐级与CPacketFileSystem的m\_MountPoints（记录Mount的Map）中的key对比，查找相应的CPFSBase指针。

例如我们要打开的文件路径是"/run/map/map001.xml"，那么我们首先在m\_MountPoints中查找"/run"，得到Mount的文件系统的基类指针CPFSBase\*，然后通过该基类指针调用相应的继承类的OpenFile函数来打开文件。

对于不同的文件系统，OpenFile的方式是不一样的

下面看看打开不同类型文件的过程：

### Native文件的读写

如果我们要打开的是一个Native文件，那么在Mount的时候应该是采用Native文件系统方式，这样，我们通过基类指针打开CNativeFS中的OpenFile来读取文件。

在CNativeFS的OpenFile的实现中，是通过PFS中相应的文件类型的Open接口来打开的。

先看一看PFS中有哪些文件类型

**PFS的文件类型**

PFS系统中有多种文件类型，如：

CFile CNativeFile CImageFile CDataFile CInetFile CMemFile

各种文件类型针对不同的应用，他们都直接或间接继承自文件基类CBaseFile

PFS中的文件类型关系及应用：

CBaseFile

本地文件（可读写）

CNativeFile

支持PFS和Native路径的文件（可读写）

CFile

只支持读操作的文件

CImageFile

打开一个网络文件

CInetFile

CMemFile

可读写的内存文件

CDataFile

只支持写操作的内存文件

在基类CBaseFile中，提供了统一的文件数据操作接口：

virtual size\_t Read( void\* data, size\_t size) = 0;

virtual size\_t Write( const void\* data, size\_t size) = 0;

virtual fsize\_t GetSize() = 0;

virtual fsize\_t Seek( foff\_t pos, FILE\_SEEK\_METHOD fsm) = 0;

virtual fsize\_t GetPos() = 0;

virtual bool SetEnd() = 0;

virtual CImageData GetImage() = 0;

virtual bool ReadAll( void\* data, fsize\_t size ) = 0;

通过这些接口实现对文件数据的各种基本操作。

在继承类中，除实现以上的基类接口外，还会根据该文件类对应的文件类型提供一些操作接口，如打开本地文件的CFile，CNativeFile类中会提供Open， Close接口，

打开内存文件的CImageFile也会有Open接口，而内存操作文件CDataFile，

CMemFile由于不需要打开文件，只需要读写数据，于是没有提供Open接口，只实现了基类中的操作。

**Native文件的写**

在写Native文件时，首先在Mount时需要使用MT\_WRITABLE方式，如果是MT\_READONLY方式，会无法写入。 之后在通过CEnv的OpenFile接口打开文件时，需要将打开模式参数为FM\_CREAT模式，访问方式为FA\_WRONLY或FA\_RDWR。

例如：

CEnv::Mount(L"/root", L"d:\\game", FST\_NATIVE, MT\_WRITABLE)

CBaseFile\* basefile = CEnv::OpenFile(L"/root/files.txt", FM\_CREAT, FA\_RDWR);

如果不做Mount，直接通过CNativeFile 或CFile的Open来写，则需要设置Open的打开模式参数为FM\_CREAT模式，访问方式为FA\_WRONLY或FA\_RDWR。

例如：

CNativeFile nf;

nf.Open(L"d:\\game\\ files.txt ", FM\_CREAT, FA\_RDWR);

或

CFile file;

file.Open(L"d:\\ game\\ files.txt ", FM\_CREAT, FA\_RDWR);

### PFS包中的文件读写

要了解PFS包的读写首先要了解一下PFS的序列化和反序列化

**PFS中数据的序列化与反序列化**

数据的序列化是将内存数据按一定的顺序写入到数据流或文件中，反序列化则是按此顺序再将写入的数据读取到内存中，PFS中有2个流来做序列化：CMemoryStream和CFileStream。CMemoryStream是对内存数据进行序列化与反序列化操作， CFileStream是对文件数据进行序列化和反序列化操作。

CMemoryStream是操作内存数据CMemoryData，写数据时直接将数据写入CMemoryData内存中当前位置，如果CMemoryData当前分配的内存空间不足，则重新分配所需的内存空间，并做内存拷贝，读数据则是直接通过地址来读取数据。

CMemoryStream实现了对各种数据类型的序列化

例如要序列化一个整数，那就需要将整数的二进制形式，4个字节数据写入CMemoryData，其他数据类型方法类似，需要注意的是字符串，由于字符串长度不定，因此我们先写入一个记录其长度的整数，然后再写入字符串，在做反序列化的时候，先读取该长度值，然后再根据长度值去读取字符串，还需要注意编码问题，序列化和反序列化时应该采用相同的编码方式。

对于需要序列化的类，PFS中有实现序列化的接口：

class IMarshalable

{

public:

virtual CMemoryStream& marshal(CMemoryStream &os) const = 0;

virtual CMemoryStream& unmarshal(CMemoryStream &os) = 0;

};

每个要序列化和反序列化的类都必须实现该接口。

CMemoryStream实现对该接口的序列化和反序列化即可。

文件的序列化和反序列化与此类似。

实现文件序列化的接口：

class CFileMarshal

{

public:

CFileMarshal();

virtual ~CFileMarshal();

public:

virtual CFileStream& marshal( CFileStream& os) const = 0;

virtual const CFileStream& unmarshal( const CFileStream& os) = 0;

};

每个需要序列化和反序列化的PFS文件类也必须实现该接口

CFileStream实现各种数据的序列化和反序列化。

CFileStream的读写通过CFileIO提供的read和write读写到文件。

当需要把一块内存数据进行序列化的时候，通过CMarshalData来实现，该类的数据为CDataBlock，并实现了CFileMarshal接口。

CMemoryStream和CFileStream重载了<<和>>流操作符，相应的对应序列化和反序列化，同时将流本身作为返回值，以方便操作，使代码简单明确。

然后我们来了解一下PFS中的数据类型

**PFS中的数据类型**

**CMemoryData**

CMemoryData是内存数据，提供PFS中内存数据的基本操作的底层实现，其成员包括：

size\_t m\_cur; // 当前数据大小

size\_t m\_max; // 目前分配的最大空间

char\* m\_data; // 数据指针

volatile LONG m\_refcount; // 引用计数器

提供的操作包括：

static size\_t GetMaxSize( size\_t s) // 获得最大空间大小，私有函数

static size\_t IncreaseMaxSize( size\_t c) // 增加最大空间，每次增加一倍，私有函数

static CMemoryData\* Create( size\_t c = 0) // 创建一块指定size的内存数据

LONG Release() // 释放当前数据空间，引用计数置0

LONG AddRef() // 增加引用计数

LONG RefCount() const //获得当前引用数

bool Compare( const CMemoryData& dst) const // 数据对比

void\* GetData() const // 获得数据指针

size\_t GetSize() const // 获得当前数据size

void Resize( size\_t s) // 重新分配空间

void Reserve( size\_t s) // 空间不足时重新分配新的空间

CMemoryData\* Clone() const // 复制当前数据

void Clear( bool deep) // 清楚当前数据块，deep为true则删除数据，否则只是把当前大小置0

void Append( const void\* data, size\_t size) // 在当前数据后添加数据

**CDataBlock**

CDataBlock是内存块数据，对CMemoryData进行了封装，可读写。用以表示PFS中的内存数据。CDataBlock中的操作与CMemoryData相同，是通过CMemoryData来实现的。

**CImageData**

CImageData是抽象数据类型，只读，用以表示各种只读的数据类型，如从文件中读取的数据，该类对CAbstactImageImplement进行了封装

CAbstactImageImplement是抽象数据实现的一个基类，其派生类包括CRawImageData和COctetsImageData

CRawImageData表示原始文件数据，即只读文件中的数据，只读文件数据分为CFileImage和CPieceFileImage，CFileImage为读取一个文件的数据，CPieceFileImage则是根据位置和大小读取文件中一部分数据。他们有共同的基类CAbstactFileImage。

CAbstactFileImage表示抽象文件数据，记录了数据指针和大小，以及一个抽象文件句柄CFileImageHandle。

COctetsImageData是字节型只读数据，即只读的内存数据，可以将CDataBlock类型的内存数据转化为只读的ImageData类型，PFS中提供了转化接口：

PFS::CImageData DataBlockToImageData( const PFS::CDataBlock& data)

PFS::CImageData BaseFileToImageData( PFS::CBaseFile& file)

抽象数据各类之间的关系图如下：

CImageData

CAbstractImageImplementation

CRawImageData

COctetsImageData

CFileImageHandle

CAbstractFileImage

CDataBlock

CFileImage

CPieceFileImage

**读取PFS包中的文件**

要打开一个PFS包中的文件，首先要将该PFS包Mount到PFS系统中

例如：

CEnv::Mount(L"/map", L"D:\\game\\map.pfs", FST\_ZIP, MT\_ WRITABLE); //可读写

CEnv::Mount(L"/map", L"D:\\game\\map.pfs", FST\_RUNZIP, MT\_READONLY); //只读

对于一个PFS包，可以FST\_ZIP或FST\_RUNZIP 两种方式来Mount。在Mount时，先通过Native文件打开方式打开PFS包文件，然后按照之前讲解的PFS包的读法去解析包文件。

FST\_ZIP方式是解析成标准ZIP的Central，FST\_RUNZIP是解析成PFS的Central。 对应这两种方式，PFS分别提供了CFileReader和CRunFileReader两个类来实现，他们都继承自CFileReaderBase

CFileReaderBase中提供了SetFileIO接口，通过该接口可指定某个已经打开的PFS包文件，然后解析该PFS包文件。解析是在基类中实现的，因此不管用哪种方式Mount，都可以解析PFS包，但不同的方式生成的Central对象不一样。

当以FST\_ZIP方式Mount时，通过CFileReader的接口来解析，生成的是CFileCentral对象，里面包含标准zip的Central信息。

当以FST\_RUNZIP方式Mount时，通过CRunFileReader的接口来解析，生成的是CRunFileCentral对象，里面只包含标准zip的Central信息中的一部分，其余的PFS中没用到，因此不需要。在对Central部分作反序列化的时候，这些不需要的数据会被跳过。

之后通过Central信息即可找到文件在PFS包的位置及数据大小信息，然后通过相关类中的数据操作接口来读取文件数据。

读取zip和runzip系统文件的类：CRunZipBaseFile，CZipSystemReadFile，CFileCentralRead

这几个类的关系：

CBaseFile

CFileCentralRead

CZipSystemReadFile

CRunZipBaseFile

CRunZipBaseFile是runzip系统中用来读取文件的类，它继承自CZipSystemReadFile，内部通过CFileCentralRead来实现，CFileCentralRead即是根据Central中的信息，将zip或runzip系统中的文件读取到内存。

也可以使用CFileImageHandle，根据Central中的位置和大小记录，将zip系统中指定的部分读取后，当成一个抽象文件CImageFile。

**PFS包中写入文件**

要在PFS包中写入文件，必须采用FST\_ZIP方式Mount

PFS的写是由CFileWriter来实现的。

CFileWriter继承自CFileBase，提供的接口如下：

void SetZipLevel( int nLevel); // 设置Zip压缩级别，分为0-9级，0为不压缩

bool WriteNextCentral( const CFileCentral& fc); // 将指定的CFileCentral写入文件

bool WriteFileData( const PFS::CDataBlock& in, bool bFinish); // 将内存数据写入文件

bool WriteFileData( const void\* in, size\_t insize, bool bFinish);

size\_t WriteDeflateData( const void\* in, size\_t insize, bool bSeek);

virtual void CloseFile()

bool SetOffset( unsigned int uiOffset); // 设置文件偏移量

void SetCurrentCentral(CFileCentral& fc); // 设置当前正在操作的CFileCentral

CFileCentral\* AddFileCentral( const CFileCentral& fc); // 添加CFileCentral

bool SetIsZip( bool bZip) // 设置是否为zip格式，返回旧的值

size\_t DeflateBound( size\_t sourceLen) // 主要在CArrangeZipUnCompact 中使用，预先计算压缩后大小

const CFileCentral\* GetFileCentral( const std::wstring& name) const // 根据文件名得到对应的CFileCentral 指针

const CFileCentral\* GetCurCentral( ) const // 得到当前正在操作的CFileCentral

bool CFileWriter::WriteEnd() // 写入PFS包尾部

根据前面讲过的PFS包的写的过程，当要写入一个文件时，首先调用CFileWriter中的WriteNextCentral函数，注意，该函数实际不是写入文件的Central，而是写入Header（Header包含在Central中）。然后调用WriteFileData来写入文件数据（WriteFileData会调用WriteDeflateData，而WriteDeflateData中会对文件作Encode处理），这样，一个文件的数据就写完了，等要写的其他文件都写完后，最后调用WriteEnd，写入PFS包的尾部（Central会在WriteEnd中写入）。这样PFS包就写完了。

**打开读取网络文件**

对于网络文件，首先必须将网络地址Mount到PFS系统中，网络文件的Mount是采用FST\_BRIEF文件系统，只能为只读方式MT\_READONLY

例如：

CEnv::Mount( L"/b", L"http://172.16.32.30/mhzx", PFS::FST\_BRIEF, MT\_READONLY | MT\_RUNTIME )

对于url地址的Mount，必须为MT\_RUNTIME方式，否则会因为无法加载meta信息而导致Mount失败。

也可以不直接提供url地址，而提供客户端在本地的安装目录，在Mount时会读取本地目录中的version.meta文件（须保证.version.meta已经将URL设置为default类型），根据version.meta中记录的url来Mount

例如：

CEnv::Mount( L"/b", L"d:\\梦幻诛仙", PFS::FST\_BRIEF, MT\_READONLY)

这样也是可行的，Mount后的device会变成"d:\\梦幻诛仙"下version.meta文件内的default更新目录（不包含host地址），如version.meta中type为0的url地址为：

server type="0" url=<http://10.68.32.5/mhzx_ep03>

Mount后，BaseUrl即为"mhzx\_ep03"， host为10.68.32.5

Mount成功后，通过CEnv的OpenFile来打开文件，或者通过CFile的Open接口，方法与Native方式类似。

如果不做Mount，也可以直接通过CInetFile类来打开指定文件，如：

CInetFile file;

file.Open( L"http://172.16.32.30/mhzx/bin/gameclient.exe" )

在底层都是通过CInetIO的Open接口来打开的，CInetIO的Open有多个重载版本，可以根据文件的Http连接句柄，名称等相关参数，通过HttpOpenRequest创建一个Http请求句柄，然后通过HttpSendRequest发送该请求到服务器打开该文件。

也可以通过文件的url地址，使用InternetOpenUrl去打开文件。

## PFS包中的文件格式转换

在将文件数据写入PFS包中的时候，我们可能需要对数据进行压缩或作其他的处理，目的是为了节省空间，减少PFS数据包的大小。

那么，对于各种不同的文件格式，PFS是如何处理的呢？

PFS中文件的格式转换分为编码（encode）和解码（decode）两部分，encode将一种文件格式转换为另外一种文件格式，decode则与encode相反，将encode后的格式转换回原来的格式。encode和decode是通过CEncodeFile和CDecodeFile来进行的， PFS提供了CTranslateManager对Encoder和Decoder进行管理。

PFS中提供了几种encode的方式，每种方式有一个encoder和相应的decoder，例如，进行双向Zip压缩的ZipEncoder和ZipDecoder， 单向Zip压缩的ZpdEncoder和ZpdDecoder，专用于压缩xml文件的X2OEncoder和X2ODecoder，进行差别分析的BDFEncoder和BDFDecoder等。

我们看看常用的几种处理方式

**ZIP**

对于zip方式，在encode时使用ZipEncoder 来进行标准Zip压缩，而在decode时使用ZipDecoder进行标准Zip的解压缩

**ZPD**

Zpd方式是一种单向的zip压缩方式，即在encode时采用标准zip方法，但decode时不使用zip解压，而是直接使用压缩的数据，这样做的原因是为了节省时间和CPU资源，因为有些数据即使解压后，在使用时仍然要对数据进行压缩，为了避免这种无谓的运算，所以采用ZPD的方式

**X2O**

X2O是专门针对xml格式的方式，在encode时通过X2OEncoder将xml转换成二进制的xml文件，decode时与ZPD类似，也是直接copy数据，而不再进行转换。需要注意的是，X20的这种转换是有损转换，转换后的文件内容与原文件内容可能会不一样，但不一样的只是一些节点属性顺序，不会影响xml的使用。

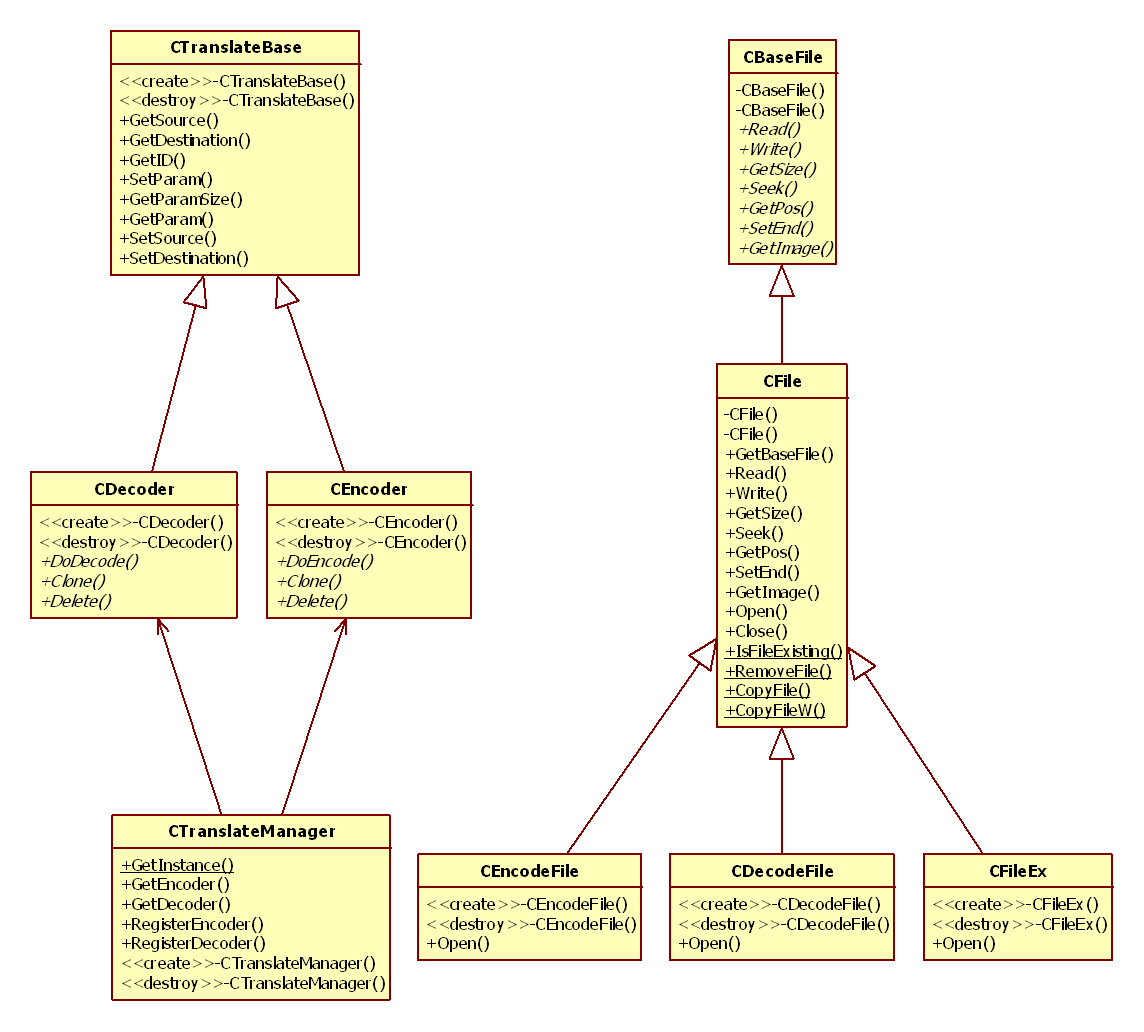
在文件写入PFS包时，我们需要指明文件的Encoder id，以标识该文件使用哪种Encoder和Decoder来转换。

在转换时，PFS有统一的转换器CFileTranslater，CFileTranslater可同时进行encode和decode处理，也可以只进行其中一种。

CFileTranslater对象中可指定源文件（转换前的文件）和目标文件（转换后的文件），同时可指定encoder 和 decoder，如果只需要做encode处理，则将decoder 置0， 同样，如果只需要做decode处理，则将encoder置0，可以同时有encoder和decoder，这时会先做decode处理，结果存储在一个临时文件CTempFile中，然后再对该临时文件做encode处理，结果输出到目标文件， 但encoder和decoder不可同时为0。

CFileTranslater中encoder和decoder都是基类CEncoder和CDecoder（CEncoder和CDecoder都继承自CTranslateBase类）的指针，在调用DoEncode()和DoDecode()时会调用相应的继承类中的虚函数来处理，对于Zip，BZF格式会调用相应的继承类（CZipEncoder，CZipDecoder和CBZFEncoder CBZFDecoder）的performencode和performdecode函数处理，而对于X20和Zpd文件，在encode时执行performencode进行编码处理，但在decode时不再进行解码处理，而是直接copy数据到目标文件。

PFS中文件转换相关类的关系如下：



文件格式转换接口：

bool EncodeFile( const std::wstring& srcFileName, const std::wstring& dstFileName );

bool EncodeFile( PFS::CBaseFile\* psrcFile, PFS::CBaseFile\* pdstFile, TSID encoderId );

bool DecodeFile( const std::wstring& srcFileName, const std::wstring& dstFileName );

bool DecodeFile( PFS::CBaseFile\* psrcFile, PFS::CBaseFile\* pdstFile, TSID decoderId );

bool TranslateFile( const std::wstring& srcFileName, const std::wstring& dstFileName );

bool TranslateFile( PFS::CBaseFile\* psrcFile, PFS::CBaseFile\* pdstFile,

TSID encoderId, TSID decoderId );

## PFS包文件的修改和整理

根据游戏的更新需求，我们可能需要对PFS包中的文件进行更新，包括添加，修改，删除操作，可能还需要对PFS包中的数据进行整理，那么，PFS中是如何实现的呢？

PFS的ArrangeZip中提供了修改整理PFS包的解决方案，其中有3个相关类：ArrangeZip，CArrangeZipCompact，CArrangeZipUnCompact

CArrangeZip为基类，提供包中的基本写操作，和相关数据如文件的Central的记录，.files.meta文件的修改等函数。

CArrangeZipCompact和CArrangeZipUnCompact是CArrangeZip的继承类，提供了修改和整理方法的具体实现，但分别对应两种不同的整理方案。

CArrangeZipCompact所采用的方案是一种紧凑整理方式，在对包中的数据修改完成后，包中可能会存在一定的垃圾数据，这样会浪费一定的空间，紧凑整理的方式就是针对这种空间浪费的，当包中存在垃圾空间时，做整理时会将文件移动， 将未修改的文件和修改的文件都重新写入，这样做即可不造成空间浪费，减小PFS包的大小，但缺点是处理起来较慢，当包比较大时，即使较小的改动，也会消耗很多时间。

CArrangeZipUnCompact则是采用非紧凑方式来整理，即未做修改的文件不移动，而修改的文件则插入到之前修改或删除文件后留下的冗余空间内，条件是要求插入的文件比冗余空间要小，在对CArrangeZipUnCompact进行初始化时，会解析包内的空闲块空间，并记录在一个vector中，再按大小作排序处理，这样，当有文件写入时，根据文件的大小就能快速找到合适的空闲块来写入该文件，如果未找到，则将该文件写入到最后一个文件的后面。

需要注意的是，对于这种非紧凑的方式，在写入文件后还需要同步更新文件的Central信息，因为该空间内的文件内容已发生变化，如不修改Central信息，那下次读取就会出错。

使用非紧凑的方式的好处在于速度比较快，特别是包比较大而修改的文件又比较少时，非常适合。

## Meta文件操作

前面已经介绍了PFS中的meta信息文件，我们知道files.meta和files2.meta是PFS包中的文件清单，其中记录的每个文件的信息在文件的校验，更新时，都需要用到，因此，对文件的操作，比如添加，修改，删除，也需要对meta文件进行同步更新。

在PFS系统中，文件的meta信息由CMetaInfo类记录，CMetaInfo结构如下：



其中记录了文件的Hash值，名称，大小，压缩方式，属性（包括TSID和CRC32）

CMetaInfo在文件的识别，格式转换，更新时都需要用到。

PFS系统中在MetaFileIO中提供了直接读写PFS包meta文件的接口：

CMetaFileRead中的读接口：

static bool DoRead( const CDataBlock& data, CMetaInfoMap& mm);

CMetaFileWrite中的写接口：

static bool DoWriteByVersion( CBaseFile& basefile, const CMetaInfoMap& mm, unsigned short usVersion);

通过这两个接口，可直接把meta文件中的meta信息读入到CMetaInfoMap中， 或将CMetaInfoMap中的CMetaInfo写入到meta文件中，其下层是通过CMetaRead来解析meta文件。需要注意，在写的时候存在版本差异，1.0版与1.1版的序列化方式不同（分别对应CMarshalMetaInfo和CMarshalMetaInfo\_1\_1），读的时候会根据meta文件的头部信息MetaFileHeader里记录的version来识别，写的时候需要设置，files.meta文件采用1.0方式，files2.meta文件采用1.1方式。

**1.0版与1.1版的差别**

1.1版与1.0版的差别在于1.1版多写入了2个值，size2和hash2,这两个数据表示什么意义，又为什么要在file2.meta里写入呢？ 这和游戏所使用的pfs的更新机制有关系，在客户端进行版本更新的时候，我们首先需要通过对比size和hash值来判断，文件是否有变化，如果这两个值中只要有一个不同，那就说明文件有变化，需要对该文件进行更新，把该文件添加到需要更新的文件列表，在更新文件时，由于服务器上的文件不是以pfs包存在的，而是解开状态，客户端的pfs包都对应服务器上的一个目录，例如服务器上的map.pfs，在服务器上相同路径对应一个map目录，但目录中的文件与客户端的pfs包内的文件是相同的，并且是在encode之后的状态，即有可能经过了压缩处理，出于压缩状态，其size与hash值很可能与原始文件的size和hash不一样，我们既然需要更新该文件，就需要将文件从服务器下载到本地，为了验证我们下载的文件在网络传输过程中没有被损坏，需要验证其size和hash值，为区别原始文件的size和hash，因此称为size2和hash2，也就是说，size和hash记录的是原始文件的size和hash值，如果这两个值有变化，说明该文件需要更新，而size2和hash2记录的则是该文件encode后的size和hash值，如果这两个值有变化，说明文件在下载过程中出现了损坏，需要重新下载。

## Hash值

PFS中通过Hash值来对文件和数据进行校验，对多种数据都可以通过Hash值来校验，如string，CDataBlock，文件等。

在PFS系统更新的过程中，也是通过Hash值的对比来判断是否有文件更新的

PFS包中的文件Hash值

Files.meta和files2.meta文件中都保存有文件的Hash值，其中files.meta中保存的是文件原始数据的hash值，通过对比这个hash值可以判断文件是否有改动，需要更新。Files2.meta中除记录有文件的hash值外，还记录有文件的hash2，hash2是文件经过translate处理后的hash值，其作用是判断文件是否与服务器上的相同，是否在传输过程中有损坏。

在PFS内部，文件的meta信息，即files.meta中记录的信息，都是保存在CMetaInfo类中，CMetaInfo中设置Hash值的相关接口：

const FILE\_DATA\_HASH & GetFileHash() const

void SetFileHash( const FILE\_DATA\_HASH & dataHash)

const FILE\_DATA\_HASH & GetFileHash2() const

void SetFileHash2( const FILE\_DATA\_HASH & dataHash2)

const FILE\_NAME\_HASH & GetFilePathHash() const

void SetFilePathHash( const FILE\_NAME\_HASH & pathHash)

CFileMetaInfo提供了为文件生成Hash值的接口：

MakeFileMetaInfo( CBaseFile& file, CMetaInfo& info) //生成Size和Hash

MakeFileMetaInfo2( CBaseFile& file, CMetaInfo& info)；//生成Size2和Hash2

这两个接口都由CMD5FileHashMethod的MakeFileHash实现。

此外，还有CFileNameHash实现文件名的Hash值。

## 日志记录

PFS中提供了log信息输出的基类：

class COutLog : public CUncopyableObject

{

public:

COutLog() {}

virtual ~COutLog() {}

public:

virtual void Print( const wchar\_t\* info, ...);

virtual void VPrint( const wchar\_t\* info, va\_list vl) = 0;

};

我们需要自行派生一个log记录类，如

class CPFSTestLog : public PFS::COutLog

并实现Print和VPrint接口

之后，通过PFS提供的接口设置使用我们的派生类来记录

PFS在CEnv中提供了接口：

static void SetOutLogInstance( COutLog\* log);

static COutLog\* GetOutLogInstance();

设置我们的实现的日志记录类实例，例如：

CPFSTestLog testLog;

PFS::CEnv::SetOutLogInstance(&testLog);

在PFS中，都是通过基类指针COutLog\*来调用Print记录log信息，如：

PFS::CEnv::GetOutLogInstance()->Print（message）;

对于错误日志信息，CEnv中也提供了获取错误信息的接口

static int GetLastError();

static void SetLastError(int nErrorCode);

需要注意的是，尽管PFS提供了SetLastError函数，但它应该仅用于PFS内部的实现代码中，而不要用于使用PFS的应用程序中。

# PFS系统配套更新机制

对于游戏版本的更新，PFS更新系统配置了多套更新方案，且灵活支持新方案的扩展。

当前使用的PFS更新方案可分为自动更新和手动更新，自动更新是通过游戏客户端中的相关更新程序根据相关配置信息自动从服务器下载更新文件进行更新，是一种在线更新方式，需要连接到网络。手动更新是用户手动从官网上下载更新补丁包，然后本地进行更新，是一种离线更新方式。

对于自动更新方式，又可以分为前台更新，后台更新，补丁包自动更新几种方式。

为了了解更新过程，我们先来了解相关配套工具：

补丁包制作相关工具：

1. 打包工具(PackZip.exe)

2. 差异包生成工具(makedifference.exe)

3. 自动更新Meta文件编辑工具(PatchMetaFile.exe)

4. FileMeta查看、编辑工具(EditMetaFile.exe)

5. pfs查看工具(WinPFS.exe - like WinRAR.exe, viewpfs.exe)

6. 其他辅助工具(VersionChecker.exe, ZipFileViewer.exe, viewxml.exe, pfsVerifier.exe, downloader.exe...)

客户端相关更新工具：

1. 安装工具 install.exe (安装包完整性校验，选择安装目录，执行安装)

2. 卸载工具uninstall.exe (完全删除游戏客户端目录，或只删除梦诛自身文件)

3. 游戏启动工具 launcher.exe (生成patcher相关文件的副本，根据命令行参数启动patcher)

4. 客户端更新工具patcher.exe (根据命令行参数和用户的操作执行相应的更新，启动游戏客户端，简单执行多开检测，显示版本信息和更新状态)

5. 修复工具repair.exe (修复客户端（排除自身），可只选部分pfs包，依赖于.setup.meta确定资源子包列表)

6. web数据下载工具getwebdata.exe (为了避免用户误用此程序，发布时，改名为getwebdata.dll, 以独立进程方式下载公告信息,推荐服务器列表等http资源，通过1M共享内存交换数据，需要2个命令行参数，第一个为url，第二个为共享内存对象名. 成功时，main返回0，共享内存的头4个字节为有效数据长度，后跟数据内容，失败时，main返回-1, -2, -3, -4等错误码)

当前默认的自动更新方式流程如下：

1. 使用packzip工具生成游戏完整安装包data.pfs
2. 使用makedifference工具，通过与前一版本的游戏安装包进行对比，生成一个差异包，即更新用的补丁包，补丁包有两种，分为客户端补丁包(.mzpch格式)和服务器补丁包(.mzsp格式)，其差别在于，包里的meta文件不同，客户端补丁包只包含一个记录补丁包内文件的files.meta文件和一个新的version.meta文件， 服务器补丁包除了包含以上两个文件外，还会包含files2.meta文件，其作用前面已介绍过，是用来验证服务器上的文件在下载到客户端后文件是否正确，由于服务器上的文件是以压缩方式存放，因此需要一个压缩后的文件的hash值来验证，也就是files2.meta里的hash2。
3. 将补丁包同步到更新服务器，需要将服务器补丁包或完整数据包data.pfs解压缩到更新服务器相应目录，服务器上的数据不会以pfs包形式存在，客户端上的资源包如map.pfs等，在服务器上对应一个同名目录。服务器上的包虽然解开存放，但里面的文件仍然是压缩形态，数据与pfs包内相同。
4. 客户端进行更新。

客户端更新时，首先需要通过version.meta文件中的版本对比来确定采用什么更新方式，以及确定更新服务器地址。

我们先看看version.meta文件的内容：

.version.meta 示例:

<data version="16777351" versionstring="1.0.135" lastversion="16777350" hash="3d9d854163f8f07a" release-date="1267270328" size="1323.627997" updatesize="19.996563" luv="16777362" >

<server type=“0” url=“http://patch2.mhzx.wanmei.com/mhzx”/>

<server type=“2” url=“http://patch4.mhzx.wanmei.com/mhzx2”/>

<server type=“3” url=“http://patch3.mhzx.wanmei.com/mhzx”/>

<server type=“5” url=“http://patch5.mhzx.wanmei.com/mhzx2”/>

<server type=“100” url=“http://patch100.mhzx.wanmei.com/mhzx3”/>

<server type=“103” url=“http://patch100.mhzx.wanmei.com/mhzx3”/> <updatelog><![CDATA[]]></updatelog>

</data>

其中的type表示更新类型，type目前有以下几类：

0 – 缺省方式，网通前台差异更新

1 – 完全更新，即重新安装

2 - 差异更新，后台方式

3 – 差异更新，电信前台

5 – 差异更新，电信后台

100 – 网通补丁包自动更新

103 – 电信补丁包自动更新

电信服务器是在网通服务器上自动加3，如0是网通前台更新，3表示电信前台，2表示网通后台更新，5表示电信后台更新

注意，这个表示方法目前已未使用，改为只提供一个更新地址，由服务器自动判断用户网络类型，但工具仍然提供了该功能。

url即是更新服务器的地址。

游戏启动时，launcher程序首先复制游戏安装目录下的Patcher目录到Patcher.1中，然后Launcher启动Patcher.1为目录下的Patcher.exe。

Patcher目录同时也是更新程序从服务器上下载的新Patcher的目录。

之后patcher.exe通过本地的版本号与服务器上luv（Luv即lowest update version，表示最低可更新版本号）对比，来确定更新方式，如果版本号不小于luv那么就采用默认的更新方式，否则采用补丁包自动更新方式。

**默认的在线差异更新方式**

Patcher程序从本地运行目录读取客户端的当前版本，获取更新服务器的url，从更新服务器读取新版本信息，确定是否有新版本。若有新版本存在，则对比setup.meta中子包的hash值, 得到需要更新的子包，然后对比本地子包内的 files.meta和服务器上相应目录下的files2.meta文件中各文件的hash值，得到需要更新的文件列表，将文件从服务器下载到本地临时目录中，执行Translate过程，将压缩格式的文件还原为非压缩格式，放置到运行目录中。

如果客户端版本过旧，需要执行完全更新。Launcher负责提供一个强制完全更新支持。

当本地版本过旧时，或差异更新出现异常时，可由用户执行完全更新。

完全更新模式等价于一般意义的在线安装。

用户只需下载一个很小在线安装程序，执行该程序由其完成安装程序的下载过程，并启动安装程序完成安装过程。

**补丁包自动更新**

该更新方案是为了提升大版本升级时的更新效率，当游戏客户端有大版本升级时，比如新的资料片发布，其手动更新包的大小会有上百兆甚至几百兆。此种情况下，即使从设计上我们能够实现以自动更新方式为游戏客户端升级，但是，对我们的更新服务器网络带宽的要求将是一个大的挑战。因此，我们可以通过更新程序本身给用户一种引导，建议其手动下载更新包完成客户端升级，或者，我们的更新程序自动切换到补丁包更新模式上来以减少网络下载量。

当本地版本号不小于luv即采用本方案更新，luv在打包时是可配置的，因此我们可以灵活控制更新采用什么方案。

version.meta中相应的更新信息在前面已介绍过。

首先根据version.meta中的url地址，从服务器下载补丁包描述文件，依据该文件决定需要下载的更新包，并记录更新包文件名和对应的MD5码。

补丁包描述文件的内容示例如下：

<?xml version="1.0" encoding="utf-16"?>

<data version="1.0">

<patches>

<patch id="1" name="1.0.100-1.0.101.mzpch"/>

<patch id="2" name="1.0.101-1.0.102.mzpch"/>

<patch id="3" name="1.0.102-1.0.104.mzpch"/>

<patch id="4" name="1.0.102-1.0.105.mzpch"/>

<patch id="5" name="1.0.103-1.0.105.mzpch"/>

<patch id="6" name="1.0.104-1.0.105.mzpch"/>

<patch id="7" name="1.0.104-1.0.107.mzpch"/>

<patch id="8" name="1.0.105-1.0.107.mzpch"/>

<patch id="9" name="1.0.106-1.0.107.mzpch"/>

</patches>

<updates version="1.0.107">

<update from="1.0.100" to="1.0.107" use="1,2,4,8"/>

<update from="1.0.101" to="1.0.107" use="2,4,8"/>

<update from="1.0.102" to="1.0.107" use="4,8"/>

</updates>

</data>

此文档可以用patchmetafile.exe工具编辑，各字段含义如下：

<data> - xml，文档根元素， 属性 version，文档格式版本，现为1.0

可扩展属性，暂不支持，所有其他属性均被当前版本忽略。

<patches> - 补丁包文件列表

<patch> - 补丁包描述项

属性id, 补丁包唯一id，其唯一性在本文档中保证即可，整数值

属性name,补丁包文件名，其唯一性需要在服务器上保证，且，需要保证文件的有效性。一旦文件从更新服务器上删除，本文档需要同时删除此项。

<updates> - 更新方案列表

属性version，用于标明当前补丁包更新方案所能更新的最新版本是多少，暂不使用。

<update> - 更新方案

属性from, 可用于更新的本地客户端版本

属性to, 本更新方案可以将本地客户端更新到的最新版本，一般应该是最新版本，但允许不是最新版本。

属性use，本方案需要使用的补丁包id序列，要保证id先后顺序。

例如，当我们本地版本为1.0.100，需要升级到1.0.107时，应该使用1,2,4,8这个补丁包。

下载补丁包描述文件后，我们即可知道需要下载哪些补丁包，然后从服务器下载这些补丁包，下载支持断点续传，因为一些相对比较大的子包的清单文件(.files2.meta)，比如map,monster等，在下载这些子包清单文件时往往因为超时等原因导致下载中断。 为此，我们需要记录下载成功的块信息到单独的文件中，以支持断点续传。

需要记录如下信息：

1.当前下载的进度(offset), 4字节。

2.完整文件的长度(total)，8字节。高四位暂时保留，应该总是为0。

3.完整文件的hash值，8字节。

根据这3个值我们即可实现文件的断点续传和下载后的校验，以保证文件的正确性。

如果缓存中有上次没有下完的文件，根据断点续传信息，从服务器下载尚未完成的文件内容。校验下载完毕的补丁包的MD5码，确定与初始化时获取的MD5码是否匹配。注意，这里的MD5码应该为128位的真实MD5码，而非将128位MD5折半计算的64位HASH值。

校验通过后，合并下载的补丁包。参考现有合并方案实现。

最后，清除临时文件，更新完成，删除下载的补丁包。合并被中断或者下载不完整时，不删除补丁包，考虑用于断点续传。