文件监视器FileMonitor接口使用说明

此接口的信息同步采用发消息和注册及响应事件管理模式(消息和事件管理是通过自己开发的c++代码实现的，代码包括在Infra文件夹中)

FileMonitor线程调用系统接口inotify_add_watch实现文件的监视，
FileMonitor发现监视的文件夹有符合要求的文件文件出现，则通过IEventManager::Initialize()->notifyEvent 给事件管理线程发消息,事件管理线程通过onAppEvent接口(虚函数)调用(通知)文件处理线程进行处理（此处的文件处理线程需要用户自己实现，后面会提供样例)

使用方法举例

1.项目代码新引入源文件或库文件说明

- 引入Json的静态库文件libjson_linux-gcc-4.8.5_libmt.a(此库文件是通过开源代码jsoncpp-src-0.5.0编译而成,不同的编译环境名称略有不同)
- 引入自己开发的Infra库文件libInfra.a和Infra头文件 或者直接包括Infra的源码;
- 将Json的头文件夹 json 引入到项目（注意此处并不把json文件夹下的文件直接引入项目，因此程序中在包括头文件是加了json文件夹的路径的）

2.编写适合自己软件处理逻辑的文件消息接收处理逻辑

文件接收处理程序常用代码框架见后面的样例中的TestFileMonitor：

代码中的TestFileMonitor名在实际软件中需要修改, 需要自己实现的个性代码逻辑在 TestFileMonitor::onAppEvent 及 TestFileMonitor::TestFileHandler::handleMessage .

- onAppEvent是接收文件监视器通过事件管理线程发现的消息.
- handleMessage是在onAppEvent处理后接收到消息根据不同的文件事件类型及文件路径文件名等进行文件处理的逻辑。

上面举例代码方法实现的源码见后面部分的附录部分

3.在自己的项目主函数中按如下顺序调用接口

(1) 启动事件管理线程

```
IEventManager::Initialize()->start();
```

(2) 启动响应文件监视器的处理线程

```
TestFileMonitor::Instance()->start();
```

(3) 文件监视线程启动(添加监视路径,文件类型等)

```
FileMonitor::Instance()->start();
```

(4) 添加监视的文件路径和文件类型等设置

```
//可重复调用，完成同时对多个路径的监测
FileMonitor::Instance()-> addWatch(
        const char *pMonitorDir,          //(必填参数)监视路径
        unsigned int event,               //(必填参数)监视的文件事件(文件创建、修改等)
        const char *monitorFiles = NULL,  //(选填参数)关注的文件名(多个用|分隔)
        const char *monitorPrefix = NULL, //(选填参数)关注的文件名前缀(多个用|分隔)
        const char *monitorSuffix = NULL  //(选填参数)关注的文件名后缀(多个用|分隔)
       );
```

- 文件名及文件前缀、文件后缀有优先级关系，优先匹配文件全名;
- event决定什么样的文件动作才能监测到，当前版本支持的动作有如下几种类型的组合

```
EVENT_CREATE
EVENT_DELETE
EVENT_MOVE_TO
EVENT_MOVE_FROM
EVENT_MODIFY
EVENT_STOCK
EVENT_CLOSEWRITE
```

- 在centos7.8的系统中测试对监视文件夹的文件做如下操作及触发的event结果如下(在开发时请参考下面的测试结果及业务需要对event进行组合)

    1. mv 一个文件 到 监视目录(监视目录在mv之前不管是否存在mv的文件都只触发如下)

        触发: EVENT_MOVE_TO

    2. mv 监视目录下的文件 到 别的目录

触发: EVENT_MOVE_FROM

3. cp 一个文件 到 监视目录 (监视目录在cp之前不存在要cp的文件)

   先后依次触发: EVENT_CREATE 多个EVENT_MODIFY 和 EVENT_CLOSEWRITE

4. cp 一个文件 到 监视目录 (监视目录在cp之前存在要cp的文件)

   触发: 多个EVENT_MODIFY 和 EVENT_CLOSEWRITE

5. rm监视目录的文件

   触发: EVENT_DELETE

6. 修改监视目录的文件(根据不同的编辑器编辑原理，可能会触发不同的event组合，但一定在最后会触发EVENT_MODIFY、EVENT_CLOSEWRITE)

   echo "some content">> 监视日志的文件 会触发: EVENT_MODIFY 和 EVENT_CLOSEWRITE
   vim 监视日志的文件进行编辑修改 会先后触发: EVENT_MOVE_FROM 、 EVENT_CREATE 、EVENT_MODIFY、EVENT_CLOSEWRITE

7. 特殊event说明

   针对程序启动之前在监视目录就有文件如果需要通过监视器处理则需要添加"存量文件事件"
   EVENT_STOCK
   添加此事件后FileMonitor会将监视目录下符合要求的文件按文件修改时间从"老"到"新"的排序给文件处理线程发消息

注意:

在开发中需要斟酌注册什么事件类型，以防对同样的操收到重复的消息:
比如要监视某个文件夹是否出现某文件则只需要注册EVENT_MOVE_TO和EVENT_CLOSEWRITE就可以很好满足需求

附样例源码及说明

1.接收文件监视处理线程样例及说明

(1). 文件处理线程的逻辑与具体的业务和程序逻辑关联紧密,需要开发人员自己实现， 接收处理线程的框架如下样例所示，开发人员一般只需要变动样例的onAppEvent及TestFileHandler::handleMessage;

(2)onAppEvent是事件管理线程接收文件监视器发来的消息需要处理的接口函数，handleMessage是onAppEvent调用处理后发给TestFileMonitor收到消息根据不同的文件事件类型 及文件路径文件名等对文件进行处理的逻辑代码实现。

样例：TestFileMonitor.h:

```
/*************************************************************
 *
 * @file    TestFileMonitor.h
 *
 * @brief   TestFileMonitor头文件
 *
 * @author  fu.sky
 *
 * @date    2022-10-09_15:35:46
 *
 * @version V10.010.000
 *
 *************************************************************/

#ifndef _TESTFILEMONITOR_H_
```

```
#define _TESTFILEMONITOR_H_

#include "IEventManager.h"
#include "json/value.h"

class TestFileMonitor: public CThread, public IEventManager::IEventHandler
{

private:
    TestFileMonitor();
    ~TestFileMonitor();

public:
    static int isRun;
    static TestFileMonitor* Instance();
private:
    static TestFileMonitor* mInstance;

public:
    void onAppEvent(
            const char* event,
            IEventManager::EVENT_ACTION action,
            Json::Value& eventAttr
            );

    void start( const int &pre_val = -1 );
    void stop();

private:
```

```
    class TestFileHandler: public Handler
    {
        public:
            TestFileHandler( TestFileMonitor* testFileMo );
            ~TestFileHandler();
            typedef enum file_test_handler_msg
            {
                MSG_STOCK_FILE,
                MSG_CREAT_FILE,
                MSG_MODIF_FILE,
                MSG_DELET_FILE,
                MSG_MOVTO_FILE,
                MSG_MOVFM_FILE,
                MSG_CLOSER_FILE,
                MSG_NULL
            }FILE_TST_HANDLER_MSG;

            void handleMessage( Message& msg );
        private:
            TestFileMonitor* mFileTestMonitor;
    };

    Handler*    mHandler;
    Looper*     mLooper;

private:
    void threadHandler();

};
```

```
#endif//_TESTFILEMONITOR_H_
```

上面头文件中的TestFileHandler根据需求作适当变动即可满足大部分需求。

样例：TestFileMonitor.cpp:

```
/*************************************************************
 *
 * @file    TestFileMonitor.cpp
 *
 * @brief   TestFileMonitor源文件
 *
 * @author  fu.sky
 *
 * @date    2022-10-09_15:35:46
 *
 * @version V10.010.000
 *
 *************************************************************/

#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#include <libgen.h>
#include <string.h>
```

```
#include "TestFileMonitor.h"
#include "CustomOutLog.h"
#include "InfraBase.h"


static pthread_mutex_t  gsMutex = PTHREAD_MUTEX_INITIALIZER;

TestFileMonitor* TestFileMonitor::mInstance = NULL;
int TestFileMonitor::isRun = 0;


TestFileMonitor::TestFileHandler::TestFileHandler( TestFileMonitor* testFileM )
{
    mFileTestMonitor = testFileM;
}

TestFileMonitor::TestFileHandler::~TestFileHandler()
{

}

void TestFileMonitor::TestFileHandler::handleMessage( Message& msg )
{
    //c_write_log(_DEBUG,"evetAttr:[%s]",msg.mAttr.toStyledString().c_str() );

    std::string tpath  = msg.mAttr["Parameter"]["Dir"].asString();
    std::string tfname = msg.mAttr["Parameter"]["FileName"].asString();
    std::string tfop   = msg.mAttr["Parameter"]["Operation"].asString();
```

```
c_write_log(_DEBUG,"[%s][%s][%s]",
        tpath.c_str(), tfop.c_str(), tfname.c_str() );

switch( msg.mWhat )
{
    case MSG_STOCK_FILE:
        {
            c_write_log(_DEBUG,"MSG_STOCK_FILE");
        }
        break;
    case MSG_CREAT_FILE:
        {
            c_write_log(_DEBUG,"MSG_CREAT_FILE");
        }
        break;
    case MSG_MODIF_FILE:
        {
            c_write_log(_DEBUG,"MSG_MODIF_FILE");
        }
        break;
    case MSG_DELET_FILE:
        {
            c_write_log(_DEBUG,"MSG_DELET_FILE");
        }
        break;
    case MSG_MOVTO_FILE:
        {
            c_write_log(_DEBUG,"MSG_MOVTO_FILE");
        }
```

```
            break;
        case MSG_MOVFM_FILE:
            {
                c_write_log(_DEBUG,"MSG_MOVFM_FILE");
            }
            break;
        case MSG_CLOSER_FILE:
            {
                c_write_log(_DEBUG,"MSG_CLOSER_FILE");
            }
            break;
        default:
            c_write_log(_DEBUG,"MSG type not recognized");
            break;
    }

    return;
}


TestFileMonitor::TestFileMonitor()
{
    mHandler = NULL;
}


TestFileMonitor::~TestFileMonitor()
{
    if ( mHandler != NULL )
```

```
    {
        delete mHandler;
        mHandler = NULL;
    }
}


void TestFileMonitor::start( const int &pre_val )
{
    if ( pre_val != -1 )
    {
        while( 0 == pre_val )
        {
            PauseThreadSleep( 0, 10 );
        }
        c_write_log(_DEBUG,"Wait for the prepend to end!");
    }

    if ( mHandler == NULL )
    {
        mHandler = new TestFileHandler( this );
    }
    else
    {
        return;
    }

    if ( isAlive() )
    {
```

```
        return;
    }

    IEventManager::Initialize()->attachEventHandler(
            STORAGE_EVENT,
            (IEventManager::IEventHandler*) this,
            (IEventManager::HANDLER_FUNC) &TestFileMonitor::onAppEvent
            );

    startThread();

    c_write_log(_DEBUG,"TestFileMonitor::start() Done!");

    return;
}


void TestFileMonitor::stop()
{
    if ( !isAlive() || mHandler == NULL )
    {
        return;
    }

    //Looper 与 线程释放顺序不能颠倒
    stopThread();
    mLooper->decRef();
    return;
}
```

```
void TestFileMonitor::onAppEvent(
        const char* event,
        IEventManager::EVENT_ACTION action,
        Json::Value& eventAttr
        )
{
    if ( strcmp( event, STORAGE_EVENT) != 0 )
    {
        return;
    }

    Message msg;

    //打印json值eventAttr (格式化后输出)
    c_write_log(_DEBUG,"eventAttr=[%s]!",eventAttr.toStyledString().c_str() );

    std::string tOpName = eventAttr["Parameter"]["Operation"].asString();

    c_write_log(_DEBUG,"topName=[%s]!",tOpName.c_str() );


    if ( tOpName == STORAGE_FILE_STOCK )
    {
        msg.mWhat = TestFileHandler::MSG_STOCK_FILE;
    }
    else if ( tOpName == STORAGE_FILE_ADDED )
    {
```

```
        msg.mWhat = TestFileHandler::MSG_CREAT_FILE;
    }
    else if ( tOpName == STORAGE_FILE_MODIFY )
    {
        msg.mWhat = TestFileHandler::MSG_MODIF_FILE;
    }
    else if ( tOpName == STORAGE_FILE_DELETED )
    {
        msg.mWhat = TestFileHandler::MSG_DELET_FILE;
    }
    else if ( tOpName == STORAGE_FILE_MOVEDTO )
    {
        msg.mWhat = TestFileHandler::MSG_MOVTO_FILE;
    }
    else if ( tOpName == STORAGE_FILE_MOVEDFROM )
    {
        msg.mWhat = TestFileHandler::MSG_MOVFM_FILE;
    }
    else if ( tOpName == STORAGE_FILE_CLOSEWRITE )
    {
        msg.mWhat = TestFileHandler::MSG_CLOSER_FILE;
    }
    else
    {
        c_write_log(_DEBUG,"topName=[%s], not recognized!",tOpName.c_str() );
        msg.mWhat = TestFileHandler::MSG_NULL;
    }

    msg.mMetaStr = event;
```

```
    msg.mArg1    = action;
    msg.mAttr    = eventAttr;
    msg.setValid( true);
    msg.mTarget  = mHandler;
    msg.mTargetLooper = mLooper;

    mHandler->sendMessage( msg );

    return;
}

void TestFileMonitor::threadHandler()
{
    pid_t tid;
    tid = syscall(SYS_gettid);
    c_write_log(_INFO,"thread id[%d]", tid);


    Looper* me = Looper::getLooper();
    mLooper = me;
    mLooper->incRef();
    mLooper->prepare();

    isRun = 1;

    mLooper->Loop();
}

TestFileMonitor* TestFileMonitor::Instance()
```

```
{
    if ( mInstance == NULL )
    {
        pthread_mutex_lock( &gsMutex );
        if ( mInstance == NULL )
        {
            mInstance = new TestFileMonitor();
        }
        pthread_mutex_unlock( &gsMutex );
    }
    return mInstance;
}
```

(2). 对样例的几点说明

- 不要在 TestFileMonitor::onAppEvent做针对具体文件进行处理相关耗时的逻辑，耗时的逻辑放在TestFileMonitor::TestFileHandler::handleMessage中实现,onAppEvent主要是接收事件管理线程收到消息后针对TestFileMonitor消息进行本地化的处理，然后将消息发给handleMessage进行处理；
- onAppEvent中的参数: Json::Value& eventAttr 格式如下:

```
 {
    "ID" : 0,
    "KeyWord" : "StorageEvent",
    "Parameter" : {
      "Dir" : "/home/zfmd/tmp/t",       #文件路径
      "FileName" : "t.sh",              #文件名
```

```
    "Operation" : "MovedFrom",        #监测到的文件操作
    "capTime" : "2022-10-27_09:11:10"   #监测到文件操作的时间
}
```

2.调用FileMonitor的测试样例

```
#include <stdio.h>

#include "IEventManager.h"
#include "FileMonitor.h"
#include "TestFileMonitor.h"

int main()
{
    //启动事件管理线程
    IEventManager::Initialize()->start();

    //响应文件监视器的处理线程
    TestFileMonitor::Instance()->start( IEventManager::isRun );

    //启动文件监视器线程
    FileMonitor::Instance()->start( TestFileMonitor::isRun );

    //定义对监视目录中出现的文件,需要监视的动作
    //   即: 如果监视目录中的文件有如下动作时则文件监视器就给响应进程发消息
    unsigned int  event = 0;
        //event |= EVENT_CREATE;
```

```
    event |= EVENT_DELETE;
    event |= EVENT_MOVE_TO;
    event |= EVENT_MOVE_FROM;
    //event |= EVENT_MODIFY;
    //event |= EVENT_STOCK;
    event |= EVENT_CLOSEWRITE;

//添加对某个目录的监视
//  其中 文件名、文件前缀、文件后缀 是否选项
FileMonitor::Instance()->addWatch(
        "/home/zfmd/tmp/t",  /*监控的路径*/
        event,               /*关注事件的类型*/
        "t.sh",              /*关注的文件名(多个用|分隔)*/
        "busi|mytest",       /*关注的文件名前缀(多个用|分隔) */
        ".xml|.csv"          /*关注的文件名后缀(多个用|分隔)*/
        );

//对/home/fusky/tmp/t1目录的k.ini和t.sh文件进行监视
FileMonitor::Instance()->addWatch(
        "/home/zfmd/tmp/t1",
        (EVENT_MOVE_TO|EVENT_CLOSEWRITE),
        "k.ini|t.sh"
        );


while(1)
{
    PauseThreadSleep( 100, 0 );
}
```

```
    return 0;
}
```