光伏2.0项目-软件自启动管理接口

此接口主要功能为：

(1) 通过定时器周期对被监控的程序进行：

程序启动(当程序退出运行了）
更新运行程序的运行状态
对停止程序第一次失败的进行多次重试

(2) 对外提供如下功能接口:

添加和删除被监控程序
查询被监控程序状态
打开或关闭被监控程序的自启动状态
运行或停止被监控的程序(只是发消息，非实时)

使用说明

1.项目代码新引入源文件或库文件

- 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）
- 引入本接口的源码文件 ELFcmd.h, ELFcmd.cpp, ELFmanager.h, ELFmanager.cpp

2.在项目中调用接口的顺序及接口使用说明

(1) 添加被监管的程序

```
// 入参vector的每一个元素的完全格式如下:
//
//  runname|cfgstat|stdlogflag|cfginter|runprefix|runpar
//
//  说明: runname是必须要有的,其他域如果没有可不带,多个域需要|分隔
//        缺少的域程序内容使用默认值
//        默认值为: runname|0|0|5||
//
//  其中和各部分的含义如下
//    runname            /*带绝对路径的程序名*/
//    cfgstat;           /*程序配置的状态*/
//    stdlogflag;        /*程序是否把输出到标准输出的日志保存到特殊日志文件*/
//    cfginter;          /*配置的程序再次重启间隔时间(单位秒)*/
//    runprefix;         /*程序运行需要的前缀:一般默认为空*/
//    runpar;            /*程序运行需要的参数:一般默认为空*/
int ELFM_addCfgInfo( std::vector<std::string> &cfgInfo );
```

其中程序配置的状态取值如下:

```
//程序配置的自启动状态
typedef enum
{
    CFG_STA_INIT = 0,   /*初始化状态*/
    CFG_STA_STARTERR,   /*自启动失败(重启到一定次数后还未成功)*/
    CFG_STA_AUTOS,      /*自启动监视状态*/
    CFG_STA_STOPS,      /*停止自启动监视状态*/
    CFG_STA_UNKNOWN

}cfg_status;
```

程序是否把输出到标准输出的取值如下:

```
//stdlogflag 标准输出日志是否保存
typedef enum
{
    std_log_save_off = 0,   /*不保存*/
    std_log_save_on         /*保存*/
}STD_LOG_FLAG;
```

说明:stdlogflag标识是为了调试某个程序需要查询标准输出而设定的配置,正常运行程序此标识推荐配置成0； 当输入标识配置为1时会在被监视程序所在同级目录下自动建立ttylog文件夹及"被监视程序名.log"日志文件,并将被监视程序运行中输出到标准输出的日志写入此文件.

使用举例如下:

```
    std::vector<std::string> cfgInfo;

    std::string tpstr = "/zfmd/pvfs20/gaolongshan/CDUS/CDUS|0|1|2||0";
    cfgInfo.push_back( tpstr );
    tpstr = "/zfmd/pvfs20/gaolongshan/01/STF/STF|0|1|2";
    cfgInfo.push_back( tpstr );
    tpstr = "/zfmd/pvfs20/gaolongshan/01/UTF/UTF";
    cfgInfo.push_back( tpstr );

    ELFM_addCfgInfo( cfgInfo );
```

(2) 启动主程序

```
/*************************************************************************
 *******           启动 或 停止 主程序(管理程序)接口 (其中包括定时器的启动)
 *******
 *************************************************************************/
//  timerFixSeconds 定时器的周期时间(单位秒)
//  starHold  自启动失败判断的阈值
//                  0:不判断,
//                  大于0:当自启动大于等于此次数时判断为失败
//  stopHold  停止失败判断的阈值
//                  0:不判断
//                 大于0:当停止程序次数大于等于此次数时判断为失败
//  pre_val   前一个线程程序运行标识(此参数一般默认即可)
unsigned char ELFM_start( int timerFixSeconds = 2,
```

```
                            int starHold = 0,
                            int stopHold = 0,
                            const int &pre_val = -1 );
unsigned char ELFM_stop();
```

使用举例:

```
    //定时器的周期时间为2秒
    //  如果某个被监视的程序启动了4次还未成功启动(有无进程判断是否成功),则
    //      停止启动，且置程序的运行状态为启动失败
    //  如果通过接口停止某个被监视的程序4次还未成功停止(有无进程判断是否成功),则
    //      停止重试程序，并将程序运行状态置为停止失败
    ELFM_start( 2, 4, 4 );
```

(3) 查询接口

查询接口有两个:

1. 被监视程序的配置状态及配置参数
2. 被监视程序的运行状态信息

具体的接口内容如下:

```
/*****************************************************************************
 *******              查询接口
 *******
 *****************************************************************************/
```

```
//获取程序的的自启动配置信息
//   fullExeName: 带绝对路径的程序名
//   return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_getPCInfo( const std::string &fullExeName, p_c_info &cinfo );


//获取程序的运行状态信息
//   fullExeName: 带绝对路径的程序名
//   return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_getPRInfo( const std::string &fullExeName, p_r_info &rinfo );
```

返回信息的数据结构如下:

```
//配置的程序自启动参数
typedef struct
{
    int cfgstat;            /*程序配置的自启动状态*/
    int stdlogflag;         /*程序是否把输出到标准输出的日志保存到特殊日志文件*/
    int cfginter;           /*配置的程序再次重启间隔时间(单位秒)*/
    std::string path;       /*程序路径*/
    std::string fname;      /*程序名*/
    std::string runprefix; /*程序运行需要的前缀:一般默认为空*/
    std::string runpar;     /*程序运行需要的参数:一般默认为空*/
}p_c_info;
```

```
//此处的结构是存储/proc/pid/status中信息
//   其中的信息展现形式为: human readable
//   例如:
//    进程状态也是带解读模式的如: S (sleeping)
//    其中内存信息是带单位的如: 184 kB
typedef struct
{
        char name[PID_STATU_LEN];    /*进程名*/
        char state[PID_STATU_LEN];   /*进程的状态*/
        int  ppid;                   /*进程的父进程*/
        char vmsize[PID_STATU_LEN]; /*进程现在正在占用的内存*/
        char vmpeak[PID_STATU_LEN]; /*当前进程运行过程中占用内存的峰值*/
        char vmrss[PID_STATU_LEN];   /*是程序现在使用的物理内存*/
        char vmhwm[PID_STATU_LEN];   /*是程序得到分配到物理内存的峰值*/
}p_s_info;


//被管理程序的运行的状态
typedef enum
{
    RUN_STA_INIT = 0,   /*初始化状态*/
    RUN_STA_STARING,    /*正在启动中状态*/
    RUN_STA_STARERR,    /*启动失败状态*/
    RUN_STA_RUNING,     /*运行状态*/
    RUN_STA_STOPING,    /*正在停止中状态*/
    RUN_STA_DELETING,   /*正在停止并删除运行信息状态中*/
    RUN_STA_STOPRERR,   /*停止失败状态*/
```

```
    RUN_STA_STOP,       /*停止状态*/

}run_status;


//被管理程序的运行信息
typedef struct
{
    int  runstat;                       /*程序运行状态*/
    char stardate[9];                   /*程序启动的日期:YYYYMMDD*/
    long timestamp;                     /*程序最近一次启动的时间戳*/
    int  startnum;                      /*程序在一天中重启的次数*/
    int  stopnum;                       /*对程序进行停止操作重试的次数*/
    int  pidnum;                        /*程序的进程数量*/
    int  pid[SAMEEXE_SPATH_PIDS];       /*程序pid值*/
    p_s_info pf[SAMEEXE_SPATH_PIDS]; /*pid对应的信息*/
}p_r_info;
```

(4) 打开或关闭被监视程序的自启动开关

- 打开被监视程序的自启动开关: 当被监视程序停止运行时，管理程序通过定时器周期查询发现没有进程时会自动启动被监视程序
- 关闭被监视程序的自启动开关: 当被监视程序停止运行时，将不会重启被监视程序
- 接口类型说明:

    1. 操作某一个具体的程序: 需要输入被监视程序的全路径名

例如：
STF运行在 /zfmd/pvfs20/gls/01/STF目录下
则参数为:/zfmd/pvfs20/gls/01/STF/STF

2. 操作某一类程序: 需要输入这一类程序路径公共的前缀

例如:
STF运行在/zfmd/pvfs20/gls/01/STF目录下
UTF运行在/zfmd/pvfs20/gls/01/UTF目录下
要操作上面/zfmd/pvfs20/gls/01下所有程序则参数为: /zfmd/pvfs20/gls/01

3. 对所有配置的程序进行操作: 输入参数为空

具体的接口如下所示：

```
/*************************************************************************
 *******打开程序自启动开关***
 *************************************************************************/

//  fullExeName: 带绝对路径的程序名
//  return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_openAutoStart( const std::string &fullExeName );

//  prefix 带绝对路径的程序名的前缀
//  return:
//          -1 没有找到配置信息
//          0  成功
```

```
int ELFM_openPrefixAutoStart( const std::string &prefix );

//打开所有配置的自启动
int ELFM_openAllAutoStart();



/*************************************************************************
 *******关闭程序自启动开关***
 *************************************************************************/

//  fullExeName: 带绝对路径的程序名
//  return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_closeAutoStart( const std::string &fullExeName );

//  prefix 带绝对路径的程序名的前缀
//  return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_closePrefixAutoStart( const std::string &prefix );

//关闭所有配置的自启动
int ELFM_closeAllAutoStart();
```

(5) 手动启动某个或某一类被监视程序

- 说明:

    1. 启动程序的同时打开程序的自启动开关(如果程序的配置状态不为自启动状态)
    2. 此类接口并不是实时对程序进行操作只是给管理程序发消息进行处理,被管理程序的具体状态需要调用查询进行获得
    3. 此类接口有两个常用场景:

        (1) 被监视程序配置的不是启动状态,可通过接口启动程序同时打开自启动设置;
        (2) 主进程对所有程序设置了启动失败次数限制且当前被监视的程序自启动次数达到上限 不再对被监视程序进行重启了。当被监视程序问题通过其他人工干预问题解决启动问题后， 可以调用此接口进行重启(如果重启成功则恢复程序的自启动状态)

- 接口类型说明:

    1. 操作某一个具体的程序: 需要输入被监视程序的全路径名

        例如：
        STF运行在 /zfmd/pvfs20/gls/01/STF目录下
        则参数为:/zfmd/pvfs20/gls/01/STF/STF

    2. 操作某一类程序: 需要输入这一类程序路径公共的前缀

        例如:
        STF运行在/zfmd/pvfs20/gls/01/STF目录下
        UTF运行在/zfmd/pvfs20/gls/01/UTF目录下
        要操作上面/zfmd/pvfs20/gls/01下所有程序则参数为: /zfmd/pvfs20/gls/01

    3. 对所有配置的程序进行操作: 输入参数为空

具体的接口如下所示：

```
//  fullExeName: 带绝对路径的程序名
//  return:
```

```
//          -1 没有找到配置信息
//          0  成功
int ELFM_startExe( const std::string &fullExeName );

//  prefix 带绝对路径的程序名的前缀
//  return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_startExeByPrefix( const std::string &prefix );

//发送启动消息给所有配置的程序
int ELFM_startExeAll();
```

(6) 手动停止某个或某一类被监视程序

- 说明:

    1. 停止程序的同时关闭程序的自启动开关
    2. 此类接口并不是实时对程序进行操作只是给管理程序发消息进行处理,被管理程序的具体状态需要调用查询进行获得
    3. 此类接口只有一个常用场景:

        (1) 因操作需要关闭或结束某个程序;

- 接口类型说明:

    1. 操作某一个具体的程序: 需要输入被监视程序的全路径名

        例如:
        STF运行在 /zfmd/pvfs20/gls/01/STF目录下

则参数为:/zfmd/pvfs20/gls/01/STF/STF

2. 操作某一类程序: 需要输入这一类程序路径公共的前缀

例如:
STF运行在/zfmd/pvfs20/gls/01/STF目录下
UTF运行在/zfmd/pvfs20/gls/01/UTF目录下
要操作上面/zfmd/pvfs20/gls/01下所有程序则参数为: /zfmd/pvfs20/gls/01

3. 对所有配置的程序进行操作: 输入参数为空

具体的接口如下所示:

```
//  fullExeName: 带绝对路径的程序名
//  return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_stopExe( const std::string &fullExeName );

//  prefix 带绝对路径的程序名的前缀
//  return:
//          -1 没有找到配置信息
//          0  成功
int ELFM_stopExeByPrefix( const std::string &prefix );

//发送停止消息给所有配置的程序
int ELFM_stopExeAll();
```

(7) 其他接口

```
/*************************************************************************
 *******              删除不再需要的被管理的程序
 *******
 *************************************************************************/
//  删除某个程序的配置和运行信息(同时结束此程序的运行,如果程序是运行状态)
//     此接口只是向ELFmanger主进程发消息进行处理，是否处理成功需要调用相关
//     查询接口进行查询;
//其中导致删除失败的一种常见原因是: 停止在运行的程序时停止不掉导致不能对
//     程序的信息进行删除
//
//   fullExeName: 带绝对路径的程序名
void ELFM_delExeInfo( const std::string &fullExeName );
```