Le système de LOG

Besoin d'enregistrer dans un fichier ou afficher sur la sortie standard (std::cout) ou dans la fenêtre de LOG de l'interface graphique un/des événements particuliers de votre plugin, étape, action ? Vous êtes dans la bonne section !

Le log d'évènements trivial

Dans votre code il faut tout d'abord inclure (si ce n'est pas déjà fait) le fichier **ct_global/ct_context.h**. Puis simplement appeler la méthode **addMessage** de l'objet **PS_LOG**.

```
#include "ct_global/ct_context.h" // Contexte

// .....

PS_LOG->addMessage(LogInterface::info, LogInterface::step, tr("Mon message"));

// ou si vous voulez que le type soit détecté automatiquement
PS_LOG->addMessage(LogInterface::info, this, tr("Mon message"));

// .....
```

Le premier paramètre permet de connaitre le type de message, afin d'offrir une première fonction de filtrage :

- trace
- debug
- info
- warning
- error
- fatal

Le deuxième paramètre permet de connaitre l'élément qui envoie l'événement, afin d'offrir une deuxième fonction de filtrage :

- core : utilisé par ComputreeCore
- gui : utilisé par ComputreeGui
- plugin
- step
- result
- itemdrawable
- action
- reader
- exporter
- unknow

La méthode **addMessage** distribue le message à tous les **CT_AbstractLogListener** qui décident de traiter ou non le message, en fonction des éléments de filtrage ou autres.

Par défaut la fenêtre "log" de l'interface graphique affiche **tous les messages** envoyé à **PS_LOG**.

CT_AbstractLogListener

Mais qu'est-ce donc ?

Un **CT_AbstractLogListener** est une interface dont vous pouvez hériter si vous souhaitez traiter différemment les messages reçu par la méthode **addMessage**.

Dans **PluginShared** il existe deux **CT_AbstractLogListener** prédéfinis :

- CT_FileLogListener : permet d'enregistrer les logs dans un fichier dont vous définissez le chemin
- CT_BasicOStreamLogListener : permet d'afficher les logs sur une sortie standard (std::cout ou std::cerr ou etc...)

Ces deux classes héritent de **CT_AbstractLogListener** et permettent ainsi de filtrer les messages reçus.

LogInterface

La classe **LogInterface** permet d'installer plusieurs **CT_AbstractLogListener** et répartit les messages reçu lors de l'appel à **PS_LOG→addMessage...** à tous les listener.

Installer un CT_AbstractLogListener prioritaire

Si vous souhaitez recevoir les messages en priorité (dès que l'objet LogInterface reçoit le message) et que **vous traiter rapidement les messages** vous devez appeler la méthode **addPrioritaryLogListener** de l'objet **PS_LOG** en lui passant le listener.

```
#include "ct_global/ct_context.h" // Contexte

// constructeur du StepPluginManager
XXXX_StepPluginManager::XXXX_StepPluginManager() : CT_AbstractStepPlugin()
{

//m_myLogListener est un attribut de ma classe XXXX_StepPluginManager. Cet attribut est du type d'
une classe qui hérite de CT_AbstractLogListener.


// défini le filtre sur le type de message. Nous n'accepterons que les messages du type "debug" et
 "error"
    m_myLogListener.setSeverityAccepted(QVector<int
>() << LogInterface::debug << LogInterface::error);


// défini le filtre sur l'élément qui a envoyé le message. Nous n'accepterons que les message d'un
e étape ou d'une action
    m_myLogListener.setTypeAccepted(QVector<int>() << LogInterface::step << LogInterface::action);


// défini un dernier filtre afin de n'accepter que les message de mon plugin. Nous n'accepterons q
ue les messages dont la variable "filter" contiendra "xxxx"
    m_myLogListener.setFilter("xxxx");
}

// destructeur du StepPluginManager
XXXX_StepPluginManager::~XXXX_StepPluginManager()
{
    // enleve le listener de l'objet du type LogInterface

    // TRES IMPORTANT !! sous peine de planter l'application
    PS_LOG->removeLogListener(&m_myLogListener);
}

// méthode init du StepPluginManager
bool XXXX_StepPluginManager::init()
{
    // enregistrement du listener auprès de PS_LOG
    PS_LOG->addNormalLogListener(&m_myLogListener);

    // envoi d'un message pour signaler l'initialisation du plugin
    PS_LOG->addMessage(LogInterface::debug, LogInterface::plugin, QObject::tr("
Plugin_XXXX initialized"), "xxxx");


// envoi d'un message qui ne sera pas traité par notre propre listener puisque la variable "filter
" ne contiendra pas "xxxx". Par contre il sera vu dans la fenêtre de log de l'interface graphique.
    PS_LOG->addMessage(LogInterface::debug, LogInterface::plugin, QObject::tr("Plugin_XXXX test"
```

```
),);

    return CT_AbstractStepPlugin::init();
}

// .....
```

Un listener enregistré en tant que listener prioritaire doit absolument traiter les messages très rapidement sous peine de voir l'application ralentir.

Si l'application plante et que vous voulez être sûr d'avoir le message avant le plantage vous devez installer votre listener de cette façon.

Installer un CT_AbstractLogListener "patient"

Si votre listener met du temps à traiter les messages vous devez l'installer de la façon suivante :

```
#include "ct_global/ct_context.h" // Contexte

// constructeur du StepPluginManager
XXXX_StepPluginManager::XXXX_StepPluginManager() : CT_AbstractStepPlugin()
{

//m_myFileLogListener est un attribut de ma classe XXXX_StepPluginManager qui est du type CT_FileL
ogListener

    // défini le chemin vers le fichier
    m_myFileLogListener.setFilePath("./logXXXX.txt");


// défini le filtre sur le type de message. Nous acceptons tous les types de messages. Nous aurion
s très bien pu ne pas appeler cette méthode, le résultat aurait été le même.
    m_myFileLogListener.setSeverityAccepted(QVector<int>());


// défini le filtre sur l'élément qui a envoyé le message. Nous acceptons les messages de n'import
e qui. Nous aurions très bien pu ne pas appeler cette méthode, le résultat aurait été le même.
    m_myFileLogListener.setTypeAccepted(QVector<int>());


// défini un dernier filtre afin de n'accepter que les message de mon plugin. Nous n'accepterons q
ue les messages dont la variable "filter" contiendra "xxxx"
    m_myFileLogListener.setFilter("xxxx");
}

// destructeur du StepPluginManager
XXXX_StepPluginManager::~XXXX_StepPluginManager()
{
    // enleve le listener de l'objet du type LogInterface

    // TRES IMPORTANT !! sous peine de planter l'application
    PS_LOG->removeLogListener(&m_myLogListener);
}

// méthode init du StepPluginManager
bool XXXX_StepPluginManager::init()
{
    // enregistrement du listener auprès de PS_LOG
    PS_LOG->addNormalLogListener(&m_myLogListener);

    // envoi d'un message pour signaler l'initialisation du plugin
```

```
    PS_LOG->addMessage(LogInterface::debug, LogInterface::plugin, QObject::tr("
Plugin_XXXX initialized"), "xxxx");


// envoi d'un message qui ne sera pas traité par notre propre listener puisque la variable "filter
" ne contiendra pas "xxxx". Par contre il sera vu dans la fenêtre de log de l'interface graphique.
    PS_LOG->addMessage(LogInterface::debug, LogInterface::plugin, QObject::tr("Plugin_XXXX test"
),);

    return CT_AbstractStepPlugin::init();
}

// .....
```

Un autre thread traite la file des messages. On peut voir le principe comme celui d'un producteur (LogInterface) et de consommateurs (CT_AbstractLogListener).

Installez toujours un CT_FileLogListener en tant que listener patient car le temps d'écriture dans le fichier est long.

Créer votre propre Listener

Il vous suffit d'hériter de la classe CT_AbstractLogListener disponible dans "ct_log/abstract" et de redéfinir la méthode *addMessage".

```
#include "ct_log/abstract/ct_abstractloglistener.h"

class XXXX_MyLogListener : public QObject, public CT_AbstractLogListener
{
public:
    XXXX_MyLogListener ();

    void addMessage(const int &severity, const int &type, const QString &s, const
 QString &filter);

private slots:
    void traiterMessage(const QString &message);

signals:
    void newMessageReceived(const QString &message);
};
```

```
#include "xxxx_myloglistener.h"

XXXX_MyLogListener ::XXXX_MyLogListener () : QObject(), CT_AbstractLogListener()
{
      // je m'envoie le message par l'intermédiaire d'un slot pour être dans le thread du GUI
      connect(this, SIGNAL(newMessageReceived(QString)), this
, SLOT(traiterMessage(QString)), Qt::QueuedConnection);
}

void XXXX_MyLogListener ::addMessage(const int &severity, const int &type, const QString &s, const
 QString &filter)
{
    // si j'accepte ce message
    if(acceptMessage(severity, type, filter))
    {
```

```
            // je traite mon message.

// ATTENTION cette méthode est appelée par un thread donc si j'affiche ce message dans un élément de l'interface graphique je dois m'envoyer un signal avec le message, exemple :
            emit newMessageReceived(s);
        }
}

void XXXX_MyLogListener ::traiterMessage(const QString &s)
{
    // je peux traiter sereinement mon message puisque je suis dans le thread du GUI
}
```