# Les threads Qt en projet

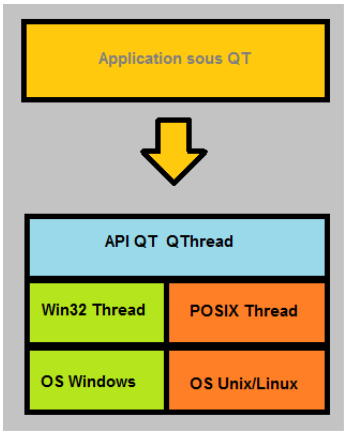


**L'objectif de ce document est de montrer la mise en œuvre des *threads* Qt en projet.**

➟ Le cours détaillé sur le "Multitâche sous Qt" : tvaira.free.fr/bts-sn/multitache/multitache-qt.pdf

## Les threads Qt

Qt fournit une classe `QThread` qui interface un *thread*. Elle permet de créer, stopper, faire exécuter par sa méthode `run()` (et d'autres opérations) un *thread*.




➟ Documentation Qt 4.8 : QThread

Pour les version de Qt inférieures à la 4.4, elle ne peut être instanciée, car la méthode `run()` est virtuelle pure (`QThread` était donc abstraite). Pour les versions plus récentes, la méthode `run()` est uniquement virtuelle et la classe peut donc être instanciée et être utilisée telle quelle.

Cette classe émet le signal `started()` lorsqu'un *thread* est lancé et `finished()` lorsque le *thread* est terminé.

La grande majorité des API graphiques des divers OS ne sont absolument pas *thread-safe*. Le traitement GUI dans différents *threads* peut être la source d'accès concurrents qui peuvent mener à des erreurs fatales de l'application. C'est pour cela que Qt oblige les traitements GUI dans le *thread* principal (celui exécutant le `main()` du programme). Il faut utiliser le système de connexion **signal/slot** pour manipuler l'affichage GUI à partir d'un *thread*.

Il y a plusieurs approches possibles dans l'utilisation de `QThread`, en voici deux :

- on dérive une classe de `QThread` et on implémente la fonction `run()` qui contiendra le code du *thread*. Seuls des objets n'héritant pas de `QObject` peuvent être utilisés. Si le *thread* doit utiliser des objets héritant de `QObject`, la boucle d'événements doit être exécutée (appel `exec()`). Il est possible de transférer les `QObject` que l'on veut utiliser vers ce *thread* (`moveToThread()`).
- on instancie directement un `QThread` et on assigne les objets héritant de `QObject` à ce *thread* en utilisant la fonction `moveToThread()`. Depuis Qt 4.4, `QThread` exécute une boucle d'événements par défaut.

# Un objet dans un thread

On instancie directement un `QThread` et on assigne un objet héritant de `QObject` à ce *thread* en utilisant la fonction `moveToThread()`.

On utilise ensuite les méthodes suivantes :

- `start()` : *slot* qui lance un *thread*, cette fonction peut prendre en paramètre la priorité donnée au *thread* ;
- `quit()` : *slot* qui stoppe la boucle d'événements du *thread* ;
- `wait()` : fonction bloquante qui attend la fin de l'exécution, il est possible de spécifier un *timeout*.

La classe `QThread` émet le *signal* `started()` lorsqu'un *thread* est lancé et `finished()` lorsque le *thread* est terminé.

```
#include <QApplication>
#include <QDebug>
#include "tthread.h"

int main(int argc, char **argv)
{
   QApplication a(argc, argv);

   qDebug() << "PID : " << (int) a.applicationPid();
   qDebug() << Q_FUNC_INFO << QApplication::instance()->thread()->currentThreadId() << a.
      thread();

   TThread t; // l'objet qui s'exécutera dans le thread
   QThread qThread; // le thread

   // attache l'objet au thread
   t.moveToThread(&qThread);

   // connexion signal/slot :

   //  - pour exécuter main() dans le thread au démarrage
   QObject::connect(&qThread, SIGNAL(started()), &t, SLOT(main()));

   //  - pour exécuter terminer() dans le thread à la fin
   QObject::connect(&qThread, SIGNAL(finished()), &t, SLOT(terminer()));

   // démarre le thread qui exécute main()
   qThread.start();

   //...

   // met fin à la boucle événementielle du thread qui exécutera ensuite terminer()
   qThread.quit();

   // attend la fin du thread
   qThread.wait();

   return 0;
}
```

On déclare notre objet qui sera exécuté dans le *thread* :

```
#ifndef TTHREAD_H
#define TTHREAD_H

#include <QtCore>

class TThread : public QObject
{
    Q_OBJECT

private:
    void traiter();

public:
    TThread();
    ~TThread();

public slots:
    void main(); // le corps principal du thread
    void terminer();

signals:

};

#endif // TTHREAD_H
```

On le définit :

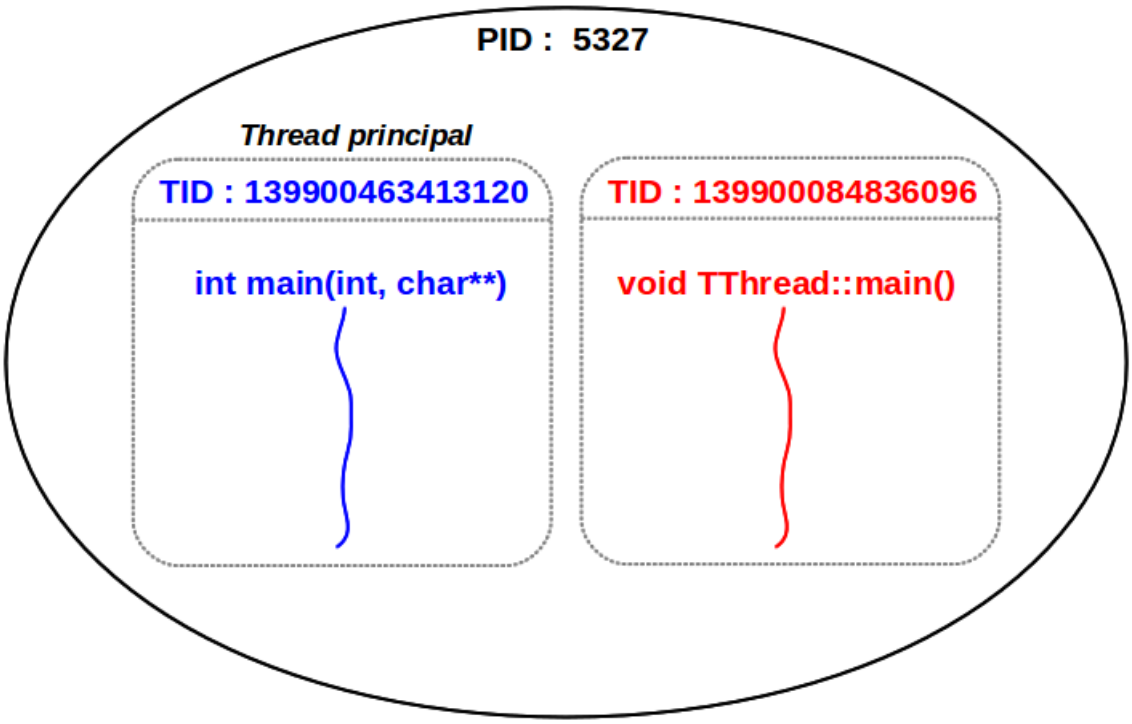
```
#include "tthread.h"

TThread::TThread()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}

TThread::~TThread()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}

void TThread::main()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;

    // par exemple :
    traiter();
}

// s'exécute dans le thread
void TThread::traiter()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}
```

```
// s'exécute dans le thread
void TThread::terminer()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}
```

On obtient :

```
PID : 5327
int main(int, char**) 139900463413120 QThread(0x21fef80)
TThread::TThread() 139900463413120 TThread(0x7fff607a88c0)
void TThread::main() 139900084836096 TThread(0x7fff607a88c0)
void TThread::traiter() 139900084836096 TThread(0x7fff607a88c0)
void TThread::terminer() 139900084836096 TThread(0x7fff607a88c0)
virtual TThread::~TThread() 139900463413120 TThread(0x7fff607a88c0)
```



# Les signaux et les slots entre threads

Qt offre le mécanisme *signal/slot* qui est utilisable entre les *threads*. Cela fournit une manière intéressante de **communiquer** (et donc passer des données) entre les *threads*.

Rappel : Si un *thread* interagit avec une GUI (*Graphical User Interface*), on doit alors utiliser le système de connexion *signal/slot*.

Contrairement aux slots, les signaux sont *thread safe* et peuvent donc être appelés par n'importe quel *thread*.

Par défaut, la connexion entre *threads* est **asynchrone**, car le slot sera exécuté dans le *thread* qui possède l'objet receveur. Pour cette raison, les paramètres du signal doivent pouvoir être copiés. Ce qui implique de **ne jamais utiliser un pointeur ou une référence non constante** dans les signatures des signaux/slots car rien ne permet de certifier que la mémoire sera encore valide lors de l'exécution du slot.

On reprend le programme précédent et on va faire communiquer par *signal/slot* deux *threads* :

```
#include <QApplication>
#include <QDebug>
#include "tthread.h"

int main(int argc, char **argv)
{
   QApplication a(argc, argv);

   qDebug() << "PID : " << (int) a.applicationPid();
   qDebug() << Q_FUNC_INFO << QApplication::instance()->thread()->currentThreadId() << a.
      thread();

   TThread t1; // un objet qui s'exécutera dans un thread
   TThread t2; // un objet qui s'exécutera dans un thread
   QThread qThread1; // un thread
   QThread qThread2; // un thread

   // attache les objets à leur thread
   t1.moveToThread(&qThread1);
   t2.moveToThread(&qThread2);

   // connexion signal/slot :
   //  - pour exécuter main() dans le thread au démarrage
   QObject::connect(&qThread1, SIGNAL(started()), &t1, SLOT(main()));
   QObject::connect(&qThread2, SIGNAL(started()), &t2, SLOT(main()));
   // - pour exécuter terminer() dans le thread à la fin
   QObject::connect(&qThread1, SIGNAL(finished()), &t1, SLOT(terminer()));
   QObject::connect(&qThread2, SIGNAL(finished()), &t2, SLOT(terminer()));
   // - communication inter thread
   QObject::connect(&t1, SIGNAL(nouvelleValeur(int)), &t2, SLOT(recupererValeur(int)));
   QObject::connect(&t2, SIGNAL(nouvelleValeur(int)), &t1, SLOT(recupererValeur(int)));

   // démarre les threads qui exécutent chacun leur main()
   qThread1.start();
   qThread2.start();
```

```
    // ...
    sleep(1);

    // met fin au thread qui exécutera terminer()
    qThread1.quit();
    qThread2.quit();

    // attend la fin du thread
    qThread1.wait();
    qThread2.wait();

    return 0;
}
```

Notre *thread* légèrement modifié :

```
#ifndef TTHREAD_H
#define TTHREAD_H

#include <QtCore>
#include "objet.h"

class TThread : public QObject
{
    Q_OBJECT

private:
    Objet monObjet;
    void traiter();

public:
    TThread();
    ~TThread();

public slots:
    void main(); // le corps principal du thread
    void terminer();
    void recupererValeur(int n); // un slot

signals:
    void nouvelleValeur(int n); // un signal
};

#endif // TTHREAD_H
```

Maintenant, notre *thread* émet un signal et possède un slot :

```
#include "tthread.h"

TThread::TThread()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}
```

```
TThread::~TThread()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}

// le corps principal du thread
void TThread::main()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;

    traiter();
}

// s'exécutera dans le thread
void TThread::traiter()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;

    qsrand(QThread::currentThreadId());

    int a = monObjet.calculer();

    emit nouvelleValeur(a);
}

// s'exécutera dans le thread
void TThread::terminer()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}

// s'exécutera dans le thread
void TThread::recupererValeur(int n)
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this << n;
}
```

Notre *thread* peut possèder des objets de type `QObject` et "travailler" avec :

```
#ifndef OBJET_H
#define OBJET_H

#include <QtCore>

class Objet : public QObject
{
    Q_OBJECT
private:

public:
    Objet();
    ~Objet();

    int calculer();
```

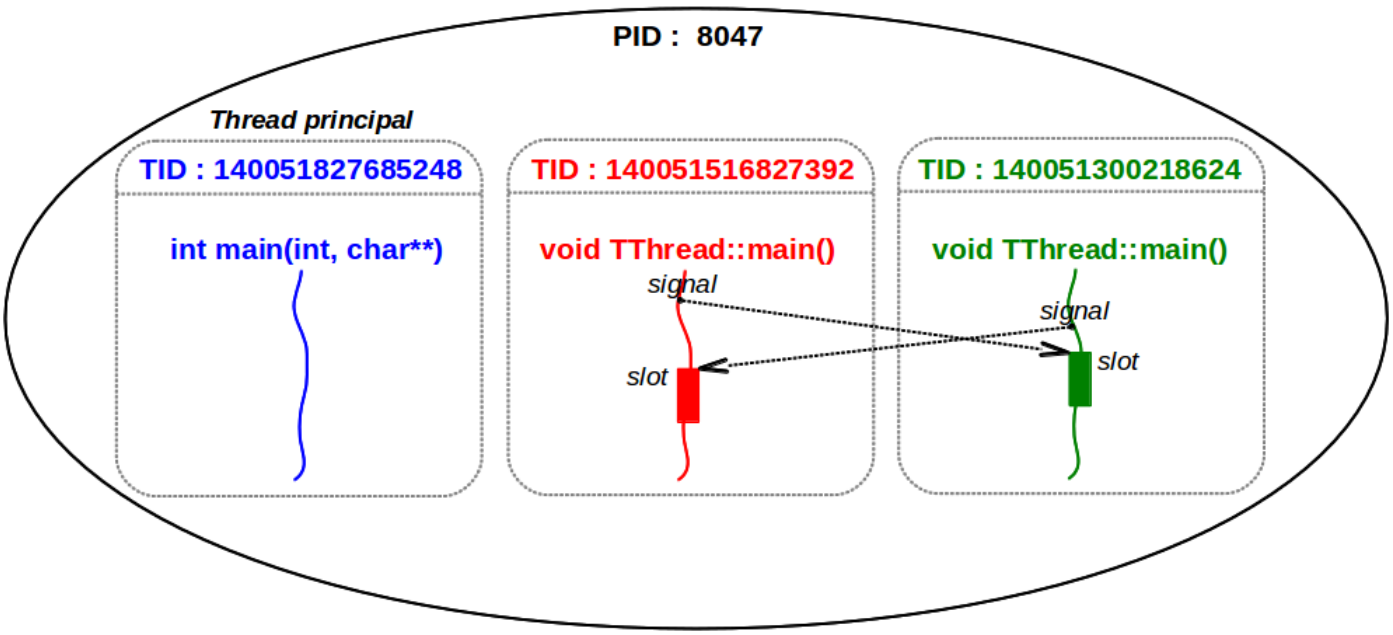
```
public slots:

signals:

};

#endif // OBJET_H
```

```
#include "objet.h"

Objet::Objet()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}

Objet::~Objet()
{
    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this;
}

int Objet::calculer()
{
    int a = (int)qrand() % 100;

    qDebug() << Q_FUNC_INFO << QThread::currentThreadId() << this << a;

    return a;
}
```

On obtient :

```
PID : 8047
int main(int, char**) 140051827685248 QThread(0xf17f80)
Objet::Objet() 140051827685248 Objet(0x7fff84fb71d0)
TThread::TThread() 140051827685248 TThread(0x7fff84fb71c0)
Objet::Objet() 140051827685248 Objet(0x7fff84fb71f0)
TThread::TThread() 140051827685248 TThread(0x7fff84fb71e0)
void TThread::main() 140051516827392 TThread(0x7fff84fb71c0)
void TThread::main() 140051300218624 TThread(0x7fff84fb71e0)
void TThread::traiter() 140051516827392 TThread(0x7fff84fb71c0)
void TThread::traiter() 140051300218624 TThread(0x7fff84fb71e0)
int Objet::calculer() 140051516827392 Objet(0x7fff84fb71d0) 72
int Objet::calculer() 140051300218624 Objet(0x7fff84fb71f0) 85
void TThread::recupererValeur(int) 140051300218624 TThread(0x7fff84fb71e0) 72
void TThread::recupererValeur(int) 140051516827392 TThread(0x7fff84fb71c0) 85
void TThread::terminer() 140051516827392 TThread(0x7fff84fb71c0)
void TThread::terminer() 140051300218624 TThread(0x7fff84fb71e0)
virtual TThread::~TThread() 140051827685248 TThread(0x7fff84fb71e0)
virtual Objet::~Objet() 140051827685248 Objet(0x7fff84fb71f0)
virtual TThread::~TThread() 140051827685248 TThread(0x7fff84fb71c0)
virtual Objet::~Objet() 140051827685248 Objet(0x7fff84fb71d0)
```




Il est aussi possible d'utiliser ses propres classes dans le mécanisme *signal/slot*. Pour cela, il faut l'enregistrer dans les métatypes par la méthode `qRegisterMetaType()`. Par exemple si vous voulez transmettre des `QVector` de `double` avec le mécanisme *signal/slot*, il faudra faire avant la connexion avec `connect()` :

```
qRegisterMetaType <QVector<double> >("QVector<double>");
```