3

Puissance du QObject

La classe QObject est sûrement la classe la plus connue et la plus populaire de Qt. Elle est la classe parent de tous les objets Qt. Attention, toutes les classe Qt n'héritent pas de QObject. Mais quand un développeur crée une nouvelle classe dans un projet et indique qu'il s'agit d'un objet Qt, il indique probablement qu'elle hérite de QObject. Ce chapitre explique certaines propriétés inhérentes aux objets, comme l'arbre d'objets Qt qui permet d'indiquer à qui appartient quoi. Le système de signaux et slots est également une pierre angulaire de Qt, du fait de sa popularité. Il permet la communication puissante et multithread entre les objets Qt. Enfin, ce chapitre explique le fonctionnement en surface du programme `moc` qui est à la base du système métaobjets de Qt.

3.1. Transformer un objet en un QObject et ses implications

Pour créer des objets Qt, il suffit de dériver la classe QObject. L'application modulaire Qt6BookApp accompagnant ce livre définit une classe StatisticsController héritant de QObject.

```
class StatisticsController : public QObject
{
    Q_OBJECT
public:
    explicit StatisticsController(QObject *parent = nullptr);
};
```

Comme StatisticsController hérite de QObject, ses instances ont un nom qui est une propriété que l'on peut questionner avec la méthode `objectName`. Par défaut cette propriété est vide. Une bonne pratique consiste à donner un nom unique au QObject via la méthode `setObjectName`. Cela permet à l'objet d'être trouvé par son nom dans un arbre d'objets Qt. La propriété `nom` d'un QObject est une propriété bindable, ce qui lui permet d'être utilisée par le système récent de Qt 6 des QProper-

Cet extrait provient du livre Qt 6 en pratique
écrit par Mériadeg Perrinel - © 2025 Éditions D-BookeR

ty (voir chapitre Fondamentaux avec le module Qt Core). Le constructeur prend un QObject en paramètre.

Le paramètre `QObject *parent` du constructeur permet d'associer un autre QObject comme parent, ce qui ferait de lui le propriétaire du nouveau QObject construit, qui devient alors un des enfants du QObject parent. Il est aussi possible de ne rien faire de tel. Dans ce cas, savoir qui est le propriétaire du QObject est le problème de celui qui le crée, c'est-à-dire la méthode appelante. Le propriétaire d'un objet doit s'occuper de sa destruction, pendant ou à la fin de l'exécution de l'application. Si un QObject appartient à un autre QObject, alors la destruction du parent entraîne automatiquement la destruction de ses enfants. Ce système permet d'avoir une gestion de mémoire propre concernant les QObject.

***Note** › Supprimer un QObject implique la suppression de tous ses enfants. Donc si un pointeur sur un enfant est stocké dans un conteneur ailleurs, il ne sera plus valide et cela peut entraîner rapidement un crash de l'application.*

Les méthodes `findChild` et `findChildren` permettent de trouver des QObject enfants soit au premier niveau des enfants d'un QObject, soit à travers tous les niveaux avec le paramètre `Qt::FindChildOptions` qui peut prendre les valeurs `Qt::FindDirectChildrenOnly` ou `Qt::FindChildrenRecursively`.

***Note** › Le constructeur de copie d'un QObject est désactivé car il apporterait un certain nombre d'ambiguïtés dans la copie. L'unicité de son nom, les connexions associées au QObject, sa position dans la hiérarchie d'objet Qt sont des problèmes inhérents à sa copie.*

La macro `Q_OBJECT` est nécessaire si vous souhaitez ajouter des signaux, des slots ou bénéficier des fonctionnalités proposées par le système de métaobjets de Qt.

Personnellement, je pense qu'il est judicieux que les classes principales, dans une application Qt constituée de modules, soient des QObject pour avoir une gestion propre de la mémoire, c'est-à-dire, s'assurer que lorsque l'application se ferme, la destruction de toutes les classes ayant une durée de vie égale à celle de l'application se déroule implicitement ; chaque classe des modules se détruit automatiquement suite à la destruction du parent et peut se charger de la destruction de ses propres données dont il est le propriétaire. En plus de la gestion de mémoire, l'utilisation des signaux et slots est la base de la communication entre les modules de l'application, justifiant davantage l'utilisation des QObject pour ces classes principales. En revanche, des classes de données à durée de vie parfois limitée et dont les instances peuvent voyager à travers les différents modules de l'application n'ont pas vocation

Cet extrait provient du livre Qt 6 en pratique
écrit par Mériadeg Perrinel – © 2025 Éditions D-BookeR

à hériter de QObject - c'est le cas des conteneurs ou d'autres classes provenant de Qt lui-même comme la classe QString.

3.2. Signaux, slots et connexions

Le mécanisme des signaux et slots de Qt est un mécanisme que j'apprécie énormément et qui simplifie fortement le développement d'applications Qt. Je n'ai pas souvenir d'un projet Qt industriel n'utilisant pas son système de connexions. Il permet de créer très facilement des routes entre des objets Qt. Deux modules d'une application peuvent communiquer sans même se connaître grâce au système de connexions. Ce système est aussi compatible avec les applications multithread (voir chapitre Créer une application multithread).

L'application Qt6BookApp présentée dans ce livre utilise ce système pour la communication entre ses modules : statistique, réseau, monitoring, base de données et localisation.

La classe StatisticsController définit des signaux :

```
signals:
  void statisticsGenerated();
  void quartilesGenerated();
  void statisticsAdded();
  void quartilesComputed();
```

La classe MonitoringController définit ce slot :

```
public slots:
  void onStatisticsGenerated();
```

Ceci est possible car les classes StatisticsController et MonitoringController possèdent toutes deux la macro `Q_OBJECT` qui est nécessaire pour l'utilisation de signaux et de slots. Qt fournit un mécanisme permettant la connexion d'un signal à un slot.

```
connect(m_statisticsController.get(),
 &StatisticsController::statisticsGenerated, m_monitoringController.get(),
 &MonitoringController::onStatisticsGenerated);
```

Ce code est la façon moderne de créer une connexion. Sur cet exemple, il y a quatre paramètres. Les deux premiers indiquent l'instance de l'objet émetteur et un pointeur de signal. Autrement dit, qui envoie le signal et de quel signal il s'agit. Les

Cet extrait provient du livre Qt 6 en pratique
écrit par Mériadeg Perrinel - © 2025 Éditions D-BookeR