第 18 章　数据库编程

本章给出了 Qt 的 SQL 类功能的大致介绍，例子后端使用的是 sqlite。

需要学习结构化查询语言吗

学习 SQL 不要为寻找任何一本声称“标准” SQL(Structured Query Language)的书而苦恼。需要阅读的最好的手册就是所选数据库服务器软件的参考指南，因为每个服务器所支持的语法和数据类型都会略有不同。幸运的是，创建表和其他 DDL 查询是一个例外，通过 QSQL 应用程序编程接口(API)可以很轻松地实现 SQL 值和 `QVariant` 之间的映射。当使用了多个数据库后，就应该或者寻找一个抽象层(根据编写的好坏程度，可能最终会让阻隔层成为历史)，或者至少为共同特性编写一个 SQL(SQLite)。

Qt 提供了一个平台中立的与 JDBC 类似的数据库接口[①]。它需要为每一个可以连接的特定数据库提供驱动程序。为了构建面向特定数据库的驱动程序，开发数据库的头文件和库必须可用。可用 Qt 连接各种不同的 SQL 数据库，包括 Oracle, Postgre SQL 和 Sybase SQL 的数据库。下面的例子中已经在 Linux 上用 MySQL 和 SQLite 测试了代码。

如果要 QtSQL 开发新东西，建议使用 SQLite 的语法，因为它有如下特点。

1. 是开源的。
2. 是 Qt 自带的。
3. 不需要从源代码构建 Qt 中，不需要构建插件，也不需要设置独立服务器。
4. 每个数据库都会映射成磁盘上的一个单独的文件。
5. 支持 SQL 的子集，该子集在其他大多数系统上都是可用的。

SQLite 是一个进程内的、零配置的数据库类库。它并不单独作为服务器运行。建议在应用程序中使用 SQLite，因为它有较好的并发性、可靠性和高性能，还拥有更快的启动/关机时间和更小的内存等需求，因为所连接的并不是诸如 MySQL 这样的外部数据库进程。但是，SQLite 应足以应付简单的实验需求和 SQL 学习需要。

其他数据库驱动程序

欲了解更多信息，可以参阅 Qt 的 SQL 驱动程序[②]文档页面。

① Java Database Connectivity(Java 数据库连接，JDBC)API。

② 参见http://doc.qt.nokia.com/latest/sql-driver.html。

已经支持的驱动程序有哪些

欲要找出当前版本的 Qt 中哪种驱动程序是可用的，有下面几种途径。

1. 运行$QTDIR/demos/sqlbrowser，然后可以在初始的 Connection Settings 对话框的组合框中看到一个驱动程序列表，如图 18.1 所示。
2. 在代码中调用 QSqlDatabase::drivers()。

如果打算和 SQL 撇清关系并直接把对象映射到永久存储设备上，你可能会对 Code Synthesis ODB 感兴趣[①]，这是一个开源的、支持 Qt 类型的对象-关系型映射层。

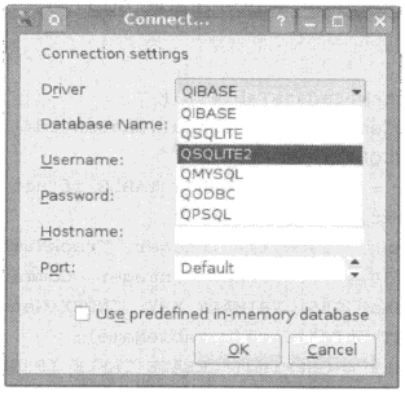

图 18.1　Sql Browser Connection Settings 对话框

18.1　QSqlDatabase：从 Qt 连接 SQL

QSqlDatabase 这个名称容易让人对这个类产生误解。它表示的并不是磁盘上的一个数据库，而是一个到数据库的连接。这个类更好的名称应该是 QSqlConnection。

与数据库服务器相连接需要这些信息：驱动程序类型、主机名、用户名、密码和数据库名称，如示例 18.1 所示。使用 SQLite 时，只需一个文件名，它会传递给 QSqlDatabase::setDatabaseName()。

用 static QSqlDatabase:: addDatabase()函数可以创建一个初始连接(即，QSqlDatabase 的一个实例)。可以给该实例一个可选的连接名称，也可以在随后用这个名称获得该连接。默认连接可以使用 QSqlDatabase::database()函数。

示例 18.1　src/sql/testprepare/testprepare.cpp

```
[ . . . . ]

void testprepare::testPrepare() {
```

① 参见http://www.codesynthesis.com/products/odb/。

```
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("localhost");
    db.setUserName("amarok");
    db.setPassword("amarok");
    db.setDatabaseName("amarok");
    QVERIFY(db.open());
```

18.1.1 数据定义语言语句：定义表

每个数据库都有一个表的集合。修改表定义的查询语句称为 DDL(Data Definition Language，DDL，数据定义语言)语句。表与结构数组非常相似，其中的每个数据成员对应数组中的一列，每个对象大致相当于一个记录，或者对应于表中的一行。要定义一个表，必须说明一个记录看起来像什么。这就意味着要定义每一列，可想像成字段、属性或数据成员。示例 18.2 定义了一个称为 MetaData 的 SQL 表。

示例 18.2 src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

bool MetaDataTable::createMetadataTable() {
    QSqlDatabase db = DbConnectionSettings::lastSaved();
    if (m_driver == "QMYSQL")
       m_createTableQStr = QString("CREATE TABLE if not exists %1 ("
         "TrackTitle  text, Artist text, "
         "AlbumTitle  text, TrackTime integer, TrackNumber integer, "
         "Genre varchar(30),  Preference integer, Comment  text, "
         "FileName  varchar(255) PRIMARY KEY, INDEX(Genre) ) "
         "DEFAULT CHARSET utf8").arg(m_tableName);
    else m_createTableQStr = QString("CREATE TABLE IF NOT EXISTS %1 ("
         "TrackTitle  text, Artist text, AlbumTitle  text, "
         "TrackTime integer, TrackNumber integer, Genre varchar(30), "
         "Preference integer, Comment  text, FileName  varchar(255) "
         "PRIMARY KEY)").arg(m_tableName);
    QSqlQuery q(m_createTableQStr);
    if (!q.isActive()) {
       qDebug() << "Create Table Fail: " << q.lastError().text()
                << q.lastQuery();
       return false;
    }
    db.commit();
    return true;
}
```

1 用 SQLite 3 测试通过。

根据所用 QSqlDriver 的不同，会用 SQL 创建不同的字符串。想要额外支持的数据库必须单独测试，因为从一台服务器到另一台不同的服务器其 SQL 语法可能会有所不同。例如，在 MySQL 中，起初使用 time 作为 TrackTime 的列类型，但随后将其改变成 integer[①]，以便可以在两种数据库中都可以使用相同的模式。

① 注意，要与 C++的类型名称区分开。

数据库连接打开后，就可以使用一个强大的称为 QSqlQuery 的类，其中有一个成员函数 exec()。

18.1.2　预处理语句：插入行

在使用 QSqlQuery 时，有两种执行 SQL 语句的方法：

1. QSqlQuery.exec(QString)
2. QSqlQuery.prepare(QString)

exec(QString)要慢一些，因为它需要服务器来解析每个 SQL 语句。预处理语句(prepared statement)更安全些，因为不需要对字符串进行转义。它们也比较快，尤其是当重复执行带不同参数的相同 SQL 语句时。SQL 驱动程序只需解析一次查询字符串。

示例 18.3 给出了预处理语句插入或更新行的用法。这里使用的是命名参数(named parameter)，但也有可能使用诸如 addBindValue 和形如":1"、":2"等的定位参数(positional parameter)。在 MySQL 中，诸如插入或更新一行这样的单一 SQL 操作语句与 SQLite 的语句略有不同。如此一来，就有两个不同的插入字符串。

示例 18.3　src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

MetaDataTable::MetaDataTable(QObject* parent)
    : QObject(parent), m_tableName("MetaData") {
    setObjectName(m_tableName);
    m_mdl = Abstract::MetaDataLoader::instance();
    m_driver = DbConnectionSettings::lastSaved().driverName();
    Q_ASSERT(createMetadataTable());
    QString preparedQuery = "INSERT into MetaData"                    1
         "(Artist, TrackTitle, AlbumTitle, TrackNumber, TrackTime, Genre,"
         "Preference, FileName, Comment) VALUES (:artist, :title, :album,"
         ":track, :time, :genre, :preference, :filename, :comment) "
         "ON DUPLICATE KEY UPDATE Preference=VALUES(Preference),"
         "Genre=VALUES(Genre), AlbumTitle=VALUES(AlbumTitle),"
         "TrackTitle=VALUES(TrackTitle), TrackNumber=VALUES(TrackNumber),"
         "Artist=VALUES(Artist), COMMENT=VALUES(Comment)";
    if (m_driver == "QSQLITE") {
        preparedQuery = "INSERT or REPLACE into MetaData"
            "(Artist, TrackTitle, AlbumTitle, TrackNumber, TrackTime, "
            "Genre, Preference, FileName, Comment)"
            "VALUES (:artist, :title, :album, :track, :time, :genre, "
            ":preference, :filename, :comment)";
    }
    bool prepSuccess = m_insertQuery.prepare(preparedQuery);
    if (!prepSuccess) {
        qDebug() << "Prepare fail: " << m_insertQuery.lastError().text()
                 << m_insertQuery.lastQuery();
        abort();
    }
```

1 用 MySQL 5 测试通过。

某些情况下，Qt 的 SQL 驱动程序可能不支持服务器端的预处理查询，但使用 Qt SQL，仍然可以使用客户的转义字符进行预处理查询，这是插入数据或处理用户提供的数据的最安全方式。预处理语句可以让你免受 SQL 注入攻击和其他可能出现的解析错误，且应该会比每次执行都需要解析的常规查询更快一些。

示例 18.4 给出了使用中的预处理语句。首先，对每一列的查询调用 `bindValue()`，然后再调用 `exec()`。

示例 18.4 src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

bool MetaDataTable::insert(const MetaDataValue &ft) {
    using namespace DbUtils;

    QSqlDatabase db = DbConnectionSettings::lastSaved();
    QSqlRecord record = db.record(m_tableName);
    if (record.isEmpty() && !createMetadataTable()) {
        qDebug() << "unable to create metadata: "
                 << db.lastError().text();
        return false;
    }

    m_insertQuery.bindValue(":artist", ft.artist());
    m_insertQuery.bindValue(":title", ft.trackTitle());
    m_insertQuery.bindValue(":album", ft.albumTitle());
    m_insertQuery.bindValue(":track", ft.trackNumber());
    QTime t = ft.trackTime();
    int secs = QTime().secsTo(t);
    m_insertQuery.bindValue(":time", secs);
    m_insertQuery.bindValue(":genre", ft.genre());
    m_insertQuery.bindValue(":filename", ft.fileName());
    int pref = ft.preference().intValue();
    m_insertQuery.bindValue(":preference", pref);
    m_insertQuery.bindValue(":comment", ft.comment());

    bool retval = m_insertQuery.exec();

    if (!retval) {
        qDebug() << m_insertQuery.lastError().text()
                 << m_insertQuery.lastQuery();
        abort();
    }
    emit inserted(ft);
    return retval;
}
```

正如所看到的，Qt SQL 针对不同的数据库引擎并不提供一种“一次编写，随处运行”的方式。尽管把列映射成属性是有可能的，但还是需要编写对象—关系型映射代码并必须在不同的服务器上进行测试。

18.2　查询和结果集

示例 18.4 中在 MetaData 表中插入了几行。这个应用程序中会试着将可能需要的对 SQL 表 MetaData 的全部 SQL 操作都封装在名称为 MetaDataTable 的类中。示例 18.5 中给出了一个简单查询，它返回一个 QStringList。

示例 18.5　src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

QStringList MetaDataTable::genres() const {
    QStringList sl;
    QSqlDatabase db = DbConnectionSettings::lastSaved();
    QSqlQuery q("SELECT DISTINCT Genre from MetaData");
    if (!q.isActive()) {
        qDebug() << "Query Failed: " << q.lastQuery()
                 << q.lastError().text();
    } else while (q.next()) {
        sl << q.value(0).toString();
    }
    return sl;
}
```

当需要返回一行数据时，API 中最好的表达方式是什么呢？推荐返回一个带 getter 方法和 setter 方法的对象，但该对象应该是一个堆型的 MetaDataObject 还是一个栈型的 MetaDataValue，尚值得讨论。如果返回的是指向这里所创建堆对象的指针，那么必须注意的是，到底是谁在拥有它们并负责在随后删除它们。示例 18.6 给出了另外一种方法，把 QObject 转换成其基类的值类型并返回这种类型值。

示例 18.6　src/libs/sqlmetadata/metadatatable.cpp

```
[ . . . . ]

MetaDataValue MetaDataTable::findRecord(QString fileName) {
    using namespace DbUtils;
    QFileInfo fi(fileName);
    MetaDataObject f;
    if (!fi.exists()) return f;                                    1
    QString abs = fi.absoluteFilePath();
    QSqlDatabase db = DbConnectionSettings::lastSaved();
    QString qs = QString("select * from %1 where FileName = \"%2\"")
                 .arg(m_tableName).arg(escape(abs));
    QSqlQuery findQuery(qs);
    if (!findQuery.isActive()) {
        qDebug() << "Query Failed: " << findQuery.lastQuery()
                 << findQuery.lastError().text();
        return f;
    }
    if (!findQuery.first()) return f;
    QSqlRecord rec = findQuery.record();
    for (int i=rec.count() -1; i >= 0; --i) {                      2
```

```
            QSqlField field = rec.field(i);
            QString key = field.name();
            QVariant value = field.value();
            if (key == "Preference") {
                int v = value.toInt();
                Preference p(v);
                f.setPreference(p);
            }
            else if (key == "TrackTime") {                          3
                QTime trackTime;
                trackTime = trackTime.addSecs(value.toInt());
                f.setTrackTime(trackTime);
            }
            else {
                f.setProperty(key, value);                          4
            }

        }
        return f;                                                   5
    }
```

1 QObject 按值返回？别忘了，MetaDataValue 才是这个特殊 QObject 的基类。

2 QObject 中的各个属性会映射成表中的列名/字段值。

3 SQLite 没有 time 类型，所以必须存储为 int。

4 对于其他列，使用 QObject 的 setProperty 函数。

5 从本地将要销毁的栈 QObject 创建一个值类型。

这个例子中创建了一个 MetaDataObject 栈用于设置属性，然后按值的方式返回它，所以会返回一个临时的 MetaDataValue。这可以用来说明派生对象是如何隐式转换成基类类型的[①]。

18.3 数据库模型

图 18.2 给出了用于连接到 QTableView 的一些具体模型类。

如果打算显示一个在调用示例 18.4 后创建的表，只需用 QSqlTableModel 的 5 行代码即可实现。

示例 18.7 src/libs/tests/testsqlmetadata/testsqlmetadata.cpp

```
[ . . . . ]

void TestSqlMetaData::showTable() {
    QSqlTableModel model;
    model.setTable("MetaData");
    model.select();
    QTableView *view = new QTableView;
    view->setModel(&model);
    view->setItemDelegate(new SimpleDelegate(view));
```

① 提供的复制构造函数不是私有的。

示例 18.7 中摘录的这个测试用例会扫描通过环境变量 TESTTRACKS 设置选择的目录，还会将找到的每个 MP3 文件的元数据添加到表模型中。该测试示例已包含在 dist 目录的源代码压缩包中。

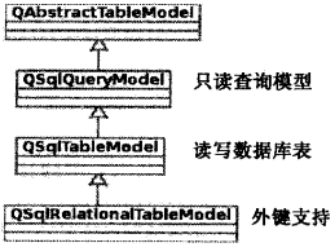

图 18.2　SQL 表模型

18.4　复习题

1. Qt 在所有平台上都包含的数据库是哪一个？
2. 如何确定你的 Qt 支持哪些数据库？
3. QSqlDatabase 是对什么的抽象：文件、连接、用户还是表？
4. DDL 查询和常规查询之间的区别是什么？
5. 为什么预处理查询要优于常规查询？
6. 类 QSqlDatabase 的名称最好应为什么？
7. 如果数据库驱动程序的 hasFeature(QSqlDriver:: PreparedQueries)返回 false，你还可以用预处理查询吗？
8. 非 DDL 查询可以修改表中的行吗？