20

Accès et gestion des bases de données locales

SQLite[1] est une base de données très appréciée car elle fournit une interface SQL tout en offrant une empreinte mémoire très réduite et une rapidité de traitement satisfaisante. En outre, elle appartient au domaine public et tout le monde peut donc l'utiliser. De nombreuses sociétés (Adobe, Apple, Google, Sun, Symbian) et plusieurs projets open-source (Mozilla, PHP, Python) fournissent désormais des produits intégrant SQLite.

SQLite étant intégré au moteur d'exécution d'Android, toute application peut créer des bases de données SQLite. Ce SGBD disposant d'une interface SQL, son utilisation est assez évidente pour quiconque a une expérience avec d'autres SGBDR. Cependant, son API native n'est pas JDBC, qui, d'ailleurs, serait trop lourd pour les terminaux limités en mémoire comme les téléphones. Par conséquent, les programmeurs Android doivent apprendre une nouvelle API mais, comme nous allons le voir, ce n'est pas très difficile.

Ce chapitre présente les bases de l'utilisation de SQLite dans le contexte du développement Android. Il ne prétend absolument pas être une présentation exhaustive de ce SGBDR : pour plus de renseignements et pour savoir comment l'utiliser dans d'autres

1. **http://www.sqlite.org**.

environnements qu'Android, nous vous conseillons l'ouvrage de Mike Owens, *The Definitive Guide to SQLite*[1] (Apress, 2006).

Les activités accédant généralement à une base de données *via* un fournisseur de contenu (*content provider)* ou un service, ce chapitre ne contient pas d'exemple complet : vous trouverez un exemple de fournisseur de contenu faisant appel à une base de données au Chapitre 28.

Présentation rapide de SQLite

SQLite, comme son nom l'indique, utilise un dialecte de SQL pour effectuer des requêtes (SELECT), des manipulations de données (INSERT, etc.) et des définitions de données (CREATE TABLE, etc.). À certains moments, il s'écarte du standard SQL-92, comme la plupart des autres SGBDR, d'ailleurs. La bonne nouvelle est que SQLite est si efficace en terme de mémoire que le moteur d'exécution d'Android peut l'inclure dans son intégralité : vous n'êtes donc pas obligé de vous contenter d'un sous-ensemble de ses fonctionnalités pour gagner de la place.

La plus grosse différence avec les autres SGBDR concerne principalement le typage des données. Tant que vous pouvez préciser les types des colonnes dans une instruction CREATE TABLE et tant que SQLite les utilise comme indication, tout va pour le mieux. Vous pouvez mettre les données que vous voulez dans les colonnes que vous souhaitez. Vous voulez placer une chaîne dans une colonne INTEGER ? Pas de problème ! Et *vice versa* ? Cela marche aussi ! C'est ce que SQLite appelle "typage manifeste" ; il est décrit de la façon suivante dans sa documentation[2] :

Avec le typage manifeste, le type d'une donnée est une propriété de la valeur elle-même, pas de la colonne dans laquelle la valeur est stockée. SQLite permet donc de stocker une valeur de n'importe quel type dans n'importe quelle colonne, quel que soit le type déclaré de cette colonne.

Certaines fonctionnalités standard de SQL ne sont pas reconnues par SQLite, notamment les contraintes FOREIGN KEY, les transactions imbriquées, RIGHT OUTER JOIN, FULL OUTER JOIN et certaines variantes de ALTER TABLE.

Ces remarques mises à part, vous disposez d'un SGBDR complet, avec des triggers, des transactions, etc. Les instructions SQL de base, comme SELECT, fonctionnent exactement comme vous êtes en droit de l'attendre. Si vous êtes habitué à travailler avec un gros SGBDR comme Oracle, vous pourriez considérer que SQLite est un "jouet", mais n'oubliez pas que ces deux systèmes ont été conçus pour résoudre des problèmes différents et que vous n'êtes pas près de voir une installation complète d'Oracle sur un téléphone.

1. **http://www.amazon.com/Definitive-Guide-SQLite/dp/1590596730**.
2. **http://www.sqlite.org/different.html**.

Commencer par le début

Android ne fournit aucune base de données de son propre chef. Si vous voulez utiliser SQLite, vous devez créer votre propre base, puis la remplir avec vos tables, vos index et vos données.

Pour créer et ouvrir une base de données, la meilleure solution consiste à créer une sous-classe de `SQLiteOpenHelper`. Cette classe enveloppe tout ce qui est nécessaire à la création et à la mise à jour d'une base, selon vos spécifications et les besoins de votre application. Cette sous-classe aura besoin de trois méthodes :

- Un constructeur qui appelle celui de sa classe parente et qui prend en paramètre le `Context` (une `Activity`), le nom de la base de données, une éventuelle fabrique de curseur (le plus souvent, ce paramètre vaudra `null`) et un entier représentant la version du schéma de la base.
- `onCreate()`, à laquelle vous passerez l'objet `SQLiteDatabase` que vous devrez remplir avec les tables et les données initiales que vous souhaitez.
- `onUpgrade()`, à laquelle vous passerez un objet `SQLiteDatabase` ainsi que l'ancien et le nouveau numéro de version. Pour convertir une base d'un ancien schéma à un nouveau, l'approche la plus simple consiste à supprimer les anciennes tables et à en créer de nouvelles. Le Chapitre 28 donnera tous les détails nécessaires.

Le reste de ce chapitre est consacré à la création et à la suppression des tables, à l'insertion des données, etc. Il présentera également un exemple de sous-classe de `SQLiteOpenHelper`.

Pour utiliser votre sous-classe, créez une instance et demandez-lui d'appeler `getReadableDatabase()` ou `getWriteableDatabase()` selon que vous vouliez ou non modifier son contenu :

```
db=(new DatabaseHelper(getContext())).getWritableDatabase();
return (db == null) ? false : true;
```

Cet appel renverra une instance de `SQLiteDatabase` qui vous servira ensuite à interroger ou à modifier la base de données.

Lorsque vous avez fini de travailler sur cette base (lorsque l'activité est fermée, par exemple), il suffit d'appeler la méthode `close()` de cette instance pour libérer votre connexion.

Mettre la table

Pour créer des tables et des index, vous devez appeler la méthode `execSQL()` de l'objet `SQLiteDatabase` en lui passant l'instruction du LDD (*langage de définition des données*) que vous voulez exécuter. En cas d'erreur, cette méthode renvoie `null`.

Vous pouvez, par exemple, utiliser le code suivant :

```
db.execSQL("CREATE TABLE
              constantes (_id INTEGER PRIMARY KEY AUTOINCREMENT,
                          titre TEXT, valeur REAL);");
```

Cet appel crée une table `constantes` avec une colonne de clé primaire `_id` qui est un entier incrémenté automatiquement (SQLite lui affectera une valeur pour vous lorsque vous insérerez les lignes). Cette table contient également deux colonnes de données : `titre` (un texte) et `valeur` (un nombre réel). SQLite créera automatiquement un index sur la colonne de clé primaire – si vous le souhaitez, vous pouvez en ajouter d'autres à l'aide d'instructions CREATE INDEX.

Le plus souvent, vous créerez les tables et les index dès la création de la base de données ou, éventuellement, lorsqu'elle devra être mise à jour suite à une nouvelle version de votre application. Si les schémas des tables ne changent pas, les tables et les index n'ont pas besoin d'être supprimés mais, si vous devez le faire, il suffit d'utiliser `execSQL()` afin d'exécuter les instructions DROP INDEX et DROP TABLE.

Ajouter des données

Lorsque l'on crée une base de données et une ou plusieurs tables, c'est généralement pour y placer des données. Pour ce faire, il existe principalement deux approches.

Vous pouvez encore utiliser `execSQL()`, comme vous l'avez fait pour créer les tables. Cette méthode permet en effet d'exécuter n'importe quelle instruction SQL qui ne renvoie pas de résultat, ce qui est le cas d'INSERT, UPDATE, DELETE, etc. Vous pourriez donc utiliser ce code :

```
db.execSQL("INSERT INTO widgets (name, inventory)"+
           "VALUES ('Sprocket', 5)");
```

Une autre solution consiste à utiliser `insert()`, `update()` et `delete()` sur l'objet `SQLiteDatabase`. Ces méthodes utilisent des objets `ContentValues` qui implémentent une interface ressemblant à `Map` mais avec des méthodes supplémentaires pour prendre en compte les types de SQLite : outre `get()`, qui permet de récupérer une valeur par sa clé, vous disposez également de `getAsInteger()`, `getAsString()`, etc.

La méthode `insert()` prend en paramètre le nom de la table, celui d'une colonne pour l'*astuce de la colonne nulle* et un objet `ContentValues` contenant les valeurs que vous voulez placer dans cette ligne. L'astuce de la colonne nulle est utilisée dans le cas où l'instance de `ContentValues` est vide – la colonne indiquée pour cette astuce recevra alors explicitement la valeur NULL dans l'instruction INSERT produite par `insert()`.

```
ContentValues cv=new ContentValues();
cv.put(Constantes.TITRE, "Gravity, Death Star I");
cv.put(Constantes.VALEUR, SensorManager.GRAVITY_DEATH_STAR_I);
db.insert("constantes", getNullColumnHack(), cv);
```

La méthode `update()` prend en paramètre le nom de la table, un objet `ContentValues` contenant les colonnes et leurs nouvelles valeurs et, éventuellement, une clause WHERE et une liste de paramètres qui remplaceront les marqueurs présents dans celle-ci. `update()` n'autorisant que des valeurs fixes pour mettre à jour les colonnes, vous devrez utiliser `execSQL()` si vous souhaitez affecter des résultats calculés.

La clause WHERE et la liste de paramètres fonctionnent comme les paramètres positionnels qui existent également dans d'autres API de SQL :

```
// remplacements est une instance de ContentValues
String[] params=new String[] {"snicklefritz"};
db.update("widgets", remplacements, "name=?", params);
```

La méthode `delete()` fonctionne comme `update()` et prend en paramètre le nom de la table et, éventuellement, une clause WHERE et une liste des paramètres positionnels pour cette clause.

Le retour de vos requêtes

Comme pour INSERT, UPDATE et DELETE, vous pouvez utiliser plusieurs approches pour récupérer les données d'une base SQLite avec SELECT :

- `rawQuery()` permet d'exécuter directement une instruction SELECT.
- `query()` permet de construire une requête à partir de ses différentes composantes.

Un sujet de confusion classique est la classe `SQLiteQueryBuilder` et le problème des curseurs et de leurs fabriques.

Requêtes brutes

La solution la plus simple, au moins du point de vue de l'API, consiste à utiliser `rawQuery()` en lui passant simplement la requête SELECT. Cette dernière peut contenir des paramètres positionnels qui seront remplacés par les éléments du tableau passé en second paramètre. Voici un exemple :

```
Cursor c=db.rawQuery("SELECT name FROM sqlite_master
                      WHERE type='table'
                      AND name='constantes'", null);
```

Ici, nous interrogeons une table système de SQLite (`sqlite_master`) pour savoir si la table `constantes` existe déjà. La valeur renvoyée est un `Cursor` qui dispose de méthodes permettant de parcourir le résultat (voir la section "Utilisation des curseurs").

Si vos requêtes sont bien intégrées à votre application, c'est une approche très simple. En revanche, elle se complique lorsqu'une requête comprend des parties dynamiques que les paramètres positionnels ne peuvent plus gérer. Si l'ensemble de colonnes que vous voulez

récupérer n'est pas connu au moment de la compilation, par exemple, concaténer les noms des colonnes pour former une liste délimitée par des virgules peut être ennuyeux – c'est là que `query()` entre en jeu.

Requêtes normales

La méthode `query()` prend en paramètre les parties d'une instruction SELECT afin de construire la requête. Ces différentes composantes apparaissent dans l'ordre suivant dans la liste des paramètres :

1. Le nom de la table interrogée.
2. La liste des colonnes à récupérer.
3. La clause WHERE, qui peut contenir des paramètres positionnels.
4. La liste des valeurs à substituer à ces paramètres positionnels.
5. Une éventuelle clause GROUP BY.
6. Une éventuelle clause ORDER BY.
7. Une éventuelle clause HAVING.

À part le nom de la table, ces paramètres peuvent valoir `null` lorsqu'ils ne sont pas nécessaires :

```
String[] colonnes={"ID", "inventory"};
String[] params={"snicklefritz"};
Cursor result=db.query("widgets", colonnes, "name=?",
                       params, null, null, null);
```

Utilisation des "builders"

Une autre possibilité consiste à utiliser `SQLiteQueryBuilder`, qui offre bien plus de possibilités pour construire les requêtes complexes, notamment celles qui impliquent d'unir les résultats de plusieurs sous-requêtes, par exemple. En outre, l'interface `SQLiteQueryBuilder` s'accorde parfaitement avec l'interface `ContentProvider` pour exécuter les requêtes. Un patron de conception classique pour l'implémentation de la méthode `query()` de votre fournisseur de contenu consiste donc à créer un objet `SQLiteQueryBuilder`, à lui fournir certaines valeurs par défaut, puis à lui faire construire (et, éventuellement, exécuter) la requête complète en combinant ces valeurs par défaut avec celles qui ont été passées au fournisseur de contenu lors de la demande de requête.

Voici, par exemple, un extrait de code d'un fournisseur de contenu utilisant `SQLiteQueryBuilder` :

```
@Override
public Cursor query(Uri url, String[] projection, String selection,
                    String[] selectionArgs, String sort) {
```

```
    SQLiteQueryBuilder qb=new SQLiteQueryBuilder();
    qb.setTables(getTableName());
    if (isCollectionUri(url)) {
      qb.setProjectionMap(getDefaultProjection());
    }
    else {
      qb.appendWhere(getIdColumnName()+"="+url.getPathSegments().get(1));
    }
    String orderBy;
    if (TextUtils.isEmpty(sort)) {
      orderBy=getDefaultSortOrder();
    } else {
      orderBy=sort;
    }
    Cursor c=qb.query(db, projection, selection, selectionArgs,
                      null, null, orderBy);
    c.setNotificationUri(getContext().getContentResolver(), url);
    return c;
  }
```

Les fournisseurs de contenu (*content provider*) seront expliqués en détail dans la cinquième partie de ce livre. Ici, nous pouvons nous contenter de remarquer que :

1. Nous construisons un objet `SQLiteQueryBuilder`.
2. Nous lui indiquons la table concernée par la requête avec `setTables(getTableName())`.
3. Soit nous lui indiquons l'ensemble de colonnes à renvoyer par défaut (avec `setProjectionMap()`), soit nous lui donnons une partie de clause WHERE afin d'identifier une ligne précise de la table à partir d'un identifiant extrait de l'URI fournie à l'appel de `query()` (avec `appendWhere()`).
4. Enfin, nous lui demandons d'exécuter la requête en mélangeant les valeurs de départ avec celles fournies à `query()` (`qb.query(db, projection, selection, selectionArgs, null, null, orderBy)`).

Au lieu de faire exécuter directement la requête par l'objet `SQLiteQueryBuilder`, nous aurions pu appeler `buildQuery()` pour la produire et renvoyer l'instruction SELECT dont nous avions besoin ; nous aurions alors pu l'exécuter nous-mêmes.

Utilisation des curseurs

Quelle que soit la façon dont vous exécutez la requête, vous obtiendrez un `Cursor` en retour. Il s'agit de la version Android/SQLite des curseurs de bases de données, un concept utilisé par de nombreux SGBDR. Avec ce curseur, vous pouvez effectuer les opérations suivantes :

- connaître le nombre de lignes du résultat grâce à `getCount()` ;
- parcourir les lignes du résultat avec `moveToFirst()`, `moveToNext()` et `isAfterLast()` ;

- connaître les noms des colonnes avec `getColumnNames()`, les convertir en numéros de colonnes grâce à `getColumnIndex()` et obtenir la valeur d'une colonne donnée de la ligne courante *via* des méthodes comme `getString()`, `getInt()`, etc. ;
- exécuter à nouveau la requête qui a créé le curseur, avec `requery()` ;
- libérer les ressources occupées par le curseur avec `close()`.

Voici, par exemple, comment parcourir les entrées de la table `widgets` rencontrée dans les extraits précédents :

```
Cursor result=
  db.rawQuery("SELECT ID, name, inventory FROM widgets");
result.moveToFirst();
while (!result.isAfterLast()) {
  int id=result.getInt(0);
  String name=result.getString(1);
  int inventory=result.getInt(2);
  // Faire quelque chose de ces valeurs...
  result.moveToNext();
}
result.close();
```

Créer ses propres curseurs

Dans certains cas, vous pouvez vouloir utiliser votre propre sous-classe de `Cursor` plutôt que l'implémentation de base fournie par Android. Dans ces situations, vous pouvez vous servir des méthodes `queryWithFactory()` et `rawQueryWithFactory()`, qui prennent toutes les deux en paramètre une instance de `SQLiteDatabase.CursorFactory`. Cette fabrique, comme l'on pourrait s'y attendre, est responsable de la création de nouveaux curseurs *via* son implémentation de `newCursor()`.

L'implémentation et l'utilisation de cette approche sont laissées en exercice au lecteur. En fait, vous ne devriez pas avoir besoin de créer vos propres classes de curseur au cours du développement d'une application Android classique.

Des données, des données, encore des données

Si vous avez l'habitude de développer avec d'autres SGBDR, vous avez sûrement aussi utilisé des outils permettant d'inspecter et de manipuler le contenu de la base de données et qui vont au-delà de l'API. Avec l'émulateur d'Android, vous avez également deux possibilités.

Premièrement, l'émulateur est censé fournir le programme `sqlite3`, accessible *via* la commande `adb shell`. Lorsque vous avez lancé cette commande, tapez simplement `sqlite3` suivi du chemin vers le fichier de votre base de données, qui est généralement de la forme :

```
/data/data/votre.paquetage.app/databases/nom_base
```

Ici, *votre.paquetage.app* est le paquetage Java de l'application (`com.commonsware.android`, par exemple) et *nom_base* est le nom de la base de données, tel qu'il est fourni à `createDatabase()`.

Le programme `sqlite3` fonctionne bien et, si vous avez l'habitude de manipuler vos tables à partir de la console, il vous sera très utile. Si vous préférez disposer d'une interface un peu plus conviviale, vous pouvez copier la base de données SQLite du terminal sur votre machine de développement, puis utiliser un client graphique pour SQLite. Cependant, n'oubliez pas que vous travaillez alors sur une copie de la base : si vous voulez répercuter les modifications sur le terminal, vous devrez retransférer cette base sur celui-ci.

Pour récupérer la base du terminal, utilisez la commande `adb pull` (ou son équivalent dans votre environnement de développement) en lui fournissant le chemin de la base sur le terminal et celui de la destination sur votre machine. Pour stocker une base de données modifiée sur le terminal, utilisez la commande `adb push` en lui indiquant le chemin de cette base sur votre machine et le chemin de destination sur le terminal.

L'extension SQLite Manager[1] pour Firefox est l'un des clients SQLite les plus accessibles (voir Figure 20.1), car elle est disponible sur toutes les plates-formes.

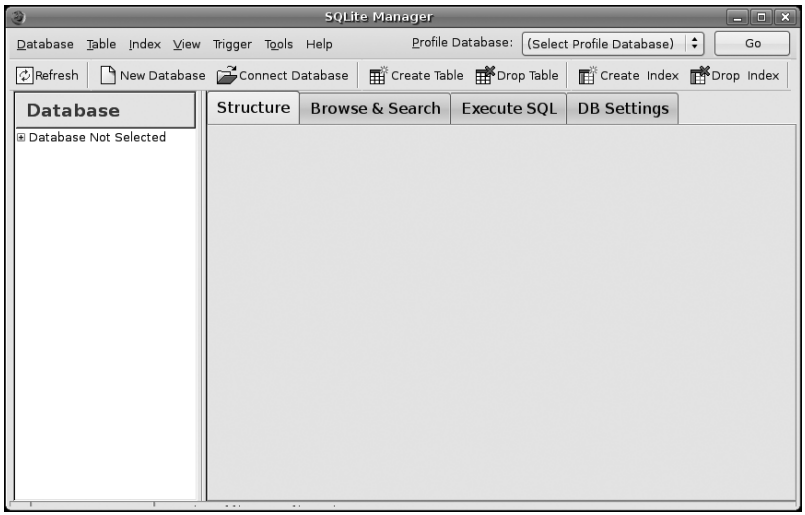

Figure 20.1
L'extension SQLite Manager de Firefox.

Vous trouverez également d'autres clients[2] sur le site web de SQLite[3].

1. **https://addons.mozilla.org/en-US/firefox/addon/5817**.
2. **http://www.sqlite.org/cvstrac/wiki?p=SqliteTools**.
3. **http://www.sqlite.org**.

21

Tirer le meilleur parti des bibliothèques Java

Java a autant de bibliothèques tierces que les autres langages de programmation modernes, si ce n'est plus. Quand nous parlons de "bibliothèques tierces", nous faisons référence ici aux innombrables JAR que vous pouvez inclure dans une application Java, quelle qu'elle soit : cela concerne tout ce que le SDK Java ne fournit pas lui-même.

Dans le cas d'Android, le cœur de la machine virtuelle Dalvik n'est pas exactement Java, et ce que fournit son SDK n'est pas la même chose qu'un SDK Java traditionnel. Ceci étant dit, de nombreuses bibliothèques Java fournissent les fonctionnalités dont ne dispose pas Android et peuvent donc vous être utiles.

Ce chapitre explique ce qu'il faut faire pour tirer parti de ces bibliothèques et décrit les limites de l'intégration du code tiers à une application Android.

Limites extérieures

Tout le code Java existant ne fonctionne évidemment pas avec Android. Un certain nombre de facteurs doivent être pris en compte :

- **API** pour la plate-forme. Est-ce que le code suppose que vous utilisez une JVM plus récente que celle sur laquelle repose Android ou suppose-t-il l'existence d'une API Java fournie avec J2SE mais qui n'existe pas dans Android, comme Swing ?
- **Taille.** Le code Java conçu pour être utilisé sur les machines de bureau ou les serveurs ne se soucie pas beaucoup de l'espace disque ni de la taille mémoire. Android, évidemment, manque des deux. L'utilisation de code tiers, notamment lorsqu'il est empaqueté sous forme de JAR, peut faire gonfler la taille de votre application..
- **Performances.** Est-ce que le code Java suppose un CPU beaucoup plus puissant que ceux que vous pouvez trouver sur la plupart des terminaux Android ? Ce n'est pas parce qu'un ordinateur de bureau peut l'exécuter sans problème qu'un téléphone mobile moyen pourra faire de même.
- **Interface.** Est-ce que le code Java suppose une interface en mode console, ou s'agit-il d'une API que vous pouvez envelopper dans votre propre interface ?

Une astuce pour régler quelques-uns de ces problèmes consiste à utiliser du code Java open-source et à modifier ce code pour l'adapter à Android. Si, par exemple, vous n'utilisez que 10 % d'une bibliothèque tierce, il est peut-être plus intéressant de recompiler ce sous-ensemble ou, au moins, d'ôter les classes inutilisées du JAR. La première approche est plus sûre dans la mesure où le compilateur vous garantit que vous ne supprimerez pas une partie essentielle du code, mais elle peut être assez délicate.

Ant et JAR

Vous avez deux possibilités pour intégrer du code tiers dans votre projet : utiliser du code source ou des JAR déjà compilés.

Si vous choisissez la première méthode, il suffit de copier le code source dans l'arborescence de votre projet (sous le répertoire `src/`) afin qu'il soit placé à côté de votre propre code, puis de laisser le compilateur faire son travail.

Si vous choisissez d'utiliser un JAR dont vous ne possédez peut-être pas les sources, vous devrez expliquer à votre chaîne de développement comment l'utiliser. Avec un IDE, il suffit de lui donner la référence du JAR. Si, en revanche, vous utilisez le script Ant `build.xml`, vous devez placer le fichier JAR dans le répertoire `libs/` créé par `activity-Creator`, où le processus de construction d'Ant ira le chercher.

Dans une édition précédente de ce livre, par exemple, nous présentions un projet MailBuzz qui, comme son nom l'indique, traitait du courrier électronique. Ce projet utilisait les API

JavaMail et avait besoin de deux JAR JavaMail : `mail-1.4.jar` et `activation-1.1.jar`. Avec ces deux fichiers dans le répertoire `libs/`, le `classpath` demandait à `javac` d'effectuer une édition de liens avec ces JAR afin que toutes les références à JavaMail dans le code de MailBuzz puissent être correctement résolues. Puis le contenu de ces JAR était énuméré avec les classes compilées de MailBuzz lors de la conversion en instructions Dalvik à l'aide de l'outil `dex`. Sans cette étape, le code se serait peut-être compilé, mais il n'aurait pas trouvé les classes JavaMail à l'exécution, ce qui aurait provoqué une exception.

Cependant, la machine virtuelle Dalvik et le compilateur fournis avec Android 0.9 et les SDK plus récents ne supportent plus certaines fonctionnalités du langage Java utilisées par JavaMail et, bien que le code source de JavaMail soit disponible, sa licence open-source (*Common Development and Distribution licence* – CDDL) pose… certains problèmes.

Suivre le script

À la différence des autres systèmes pour terminaux mobiles, Android n'impose aucune restriction sur ce qui peut s'exécuter tant que c'est du Java qui utilise la machine virtuelle Dalvik. Vous pouvez donc incorporer votre propre langage de script dans votre application, ce qui est expressément interdit sur d'autres terminaux.

BeanShell[1] est l'un de ces langages de script Java. Il offre une syntaxe compatible Java, avec un typage implicite, et ne nécessite pas de compilation.

Pour ajouter BeanShell, vous devez placer le fichier JAR de l'interpréteur dans votre répertoire `libs/`. Malheureusement, le JAR 2.0b4 disponible au téléchargement sur le site de BeanShell ne fonctionne pas tel quel avec Android 0.9 et les SDK plus récents, probablement à cause du compilateur utilisé pour le compiler. Il est donc préférable de récupérer son code source à l'aide de Subversion[2], d'exécuter `ant jarcore` pour le compiler et de copier le JAR ainsi obtenu (dans le répertoire `dist/` de BeanShell) dans le répertoire `libs/` de votre projet. Vous pouvez également utiliser le JAR BeanShell accompagnant les codes sources de ce livre (il se trouve dans le projet `Java/AndShell`). Ensuite, l'utilisation de BeanShell avec Android est identique à son utilisation dans un autre environnement Java :

1. On crée une instance de la classe `Interpreter` de BeanShell.
2. On configure les variables globales pour le script à l'aide d'`Interpreter#set()`.
3. On appelle `Interpreter#eval()` pour lancer le script et, éventuellement, obtenir le résultat de la dernière instruction.

1. **http://beanshell.org**.
2. **http://beanshell.org/developer.html**.

Voici par exemple le fichier de description XML du plus petit IDE BeanShell du monde :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<Button
    android:id="@+id/eval"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Go!"
    />
<EditText
    android:id="@+id/script"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:singleLine="false"
    android:gravity="top"
    />
</LinearLayout>
```

Voici l'implémentation de l'activité :

```
package com.commonsware.android.andshell;
import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import bsh.Interpreter;
public class MainActivity extends Activity {
  private Interpreter i=new Interpreter();

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    Button btn=(Button)findViewById(R.id.eval);
    final EditText script=(EditText)findViewById(R.id.script);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          i.set("context", MainActivity.this);
```

```
            i.eval(src);
          }
          catch (bsh.EvalError e) {
            AlertDialog.Builder builder=
                      new AlertDialog.Builder(MainActivity.this);

            builder
              .setTitle("Exception!")
              .setMessage(e.toString())
              .setPositiveButton("OK", null)
              .show();
          }
        }
      });
    }
}
```

Compilez ce projet (en incorporant le JAR de BeanShell comme on l'a mentionné plus haut), puis installez-le sur l'émulateur. Lorsque vous le lancerez, vous obtiendrez un IDE très simple, avec une grande zone de texte vous permettant de saisir votre script et un gros bouton `Go!` pour l'exécuter (voir Figure 21.1).

```
import android.widget.Toast;
Toast.makeText(context, "Hello, world!", 5000).show();
```

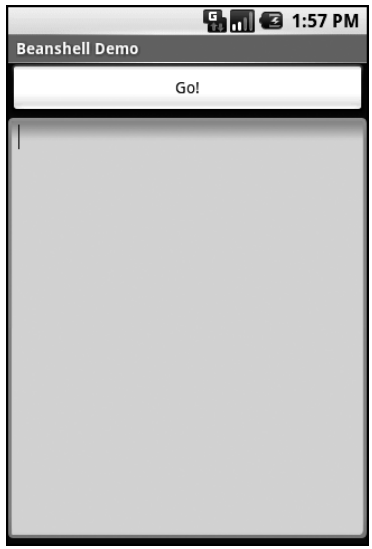

Figure 21.1
L'IDE AndShell.

Notez l'utilisation de `context` pour désigner l'activité lors de la création du toast. Cette variable a été configurée globalement par l'activité pour se désigner elle-même.

Vous pourriez l'appeler autrement : ce qui importe est que l'appel à `set()` et le code du script utilisent le même nom.

Lorsque vous cliquez sur le bouton `Go!`, vous obtenez le résultat de la Figure 21.2.

Figure 21.2
L'IDE AndShell exécutant un script BeanShell.

Ceci étant dit, il y a quelques précautions à prendre.

Premièrement, tous les langages de script ne fonctionneront pas. Ceux qui implémentent leur propre forme de compilation JIT (*just-in-time*) – en produisant le pseudo-code Java à la volée –, notamment, devront sûrement être modifiés pour produire du pseudo-code Dalvik à la place. Les langages plus simples, qui interprètent seulement les fichiers de script en appelant les API d'introspection de Java pour se ramener à des appels de classes compilées, fonctionneront probablement mieux. Même en ce cas, certaines fonctionnalités du langage peuvent ne pas être disponibles si elles reposent sur une caractéristique de l'API Java traditionnelle qui n'existe pas avec Dalvik – ce qui peut être le cas du BeanShell ou de certains JAR tiers avec les versions actuelles d'Android.

Deuxièmement, les langages de script sans JIT seront forcément plus lents que des applications Dalvik compilées ; cette lenteur peut déplaire aux utilisateurs et impliquer plus de consommation de batterie pour le même travail. Construire une application Android en BeanShell uniquement parce que vous trouvez qu'elle est plus facile à écrire peut donc rendre vos utilisateurs assez mécontents.

Troisièmement, les langages de script qui exposent toute l'API Java, comme BeanShell, peuvent réaliser tout ce qu'autorise le modèle de sécurité sous-jacent d'Android. Si votre application dispose de la permission READ_CONTACTS, par exemple, tous les scripts BeanShell qu'elle exécutera l'auront également.

Enfin, mais ce n'est pas le moins important, les JAR des interpréteurs ont tendance à être... gros. Celui du BeanShell utilisé ici fait 200 Ko, par exemple. Ce n'est pas ridicule si l'on considère ce qu'il est capable de faire, mais cela implique que les applications qui utilisent BeanShell seront bien plus longues à télécharger, qu'elles prendront plus de place sur le terminal, etc.

Tout fonctionne... enfin, presque

Tous les codes Java ne fonctionneront pas avec Android et Dalvik. Vous devez plus précisément tenir compte des paramètres suivants :

- Si le code Java suppose qu'il s'exécute avec Java SE, Java ME ou Java EE, il ne trouvera peut-être pas certaines API qu'il a l'habitude de trouver sur ces plates-formes mais qui ne sont pas disponibles avec Android. Certaines bibliothèques de tracé de diagrammes supposent, par exemple, la présence de primitives de tracé Swing ou AWT (*Abstract Window Toolkit*), qui sont généralement absentes d'Android.
- Le code Java peut dépendre d'un autre code Java qui, à son tour, peut avoir des problèmes pour s'exécuter sur Android. Vous pourriez vouloir utiliser un JAR qui repose, par exemple, sur une version de la classe `HTTPComponents` d'Apache plus ancienne (ou plus récente) que celle fournie avec Android.
- Le code Java peut utiliser des fonctionnalités du langage que le moteur Dalvik ne reconnaît pas.

Dans toutes ces situations, vous pouvez ne rencontrer aucun problème lors de la compilation de votre application avec un JAR compilé ; ces problèmes surviendront plutôt lors de l'exécution. C'est pour cette raison qu'il est préférable d'utiliser du code open-source avec Android à chaque fois que cela est possible : vous pourrez ainsi construire vous-même le code tiers en même temps que le vôtre et détecter plus tôt les difficultés à résoudre.

Relecture des scripts

Ce chapitre étant consacré à l'écriture des scripts avec Android, vous apprécierez sûrement de savoir qu'il existe d'autres possibilités que l'intégration directe de Beanshell dans votre projet.

Certains essais ont été réalisés avec d'autres langages reposant sur la JVM, notamment JRuby et Jython. Pour le moment, leur support d'Android est incomplet, mais cette intégration progresse.

En outre, ASE (*Android Scripting Environment)*, téléchargeable à partir d'Android Market, permet d'écrire des scripts en Python et Lua, et de les faire exécuter par BeanShell. Ces scripts ne sont pas des applications à part entière et, à l'heure où ce livre est écrit, elles ne sont pas vraiment redistribuables. De plus, ASE n'a pas été réellement

conçu pour étendre d'autres applications, même s'il peut être utilisé de cette façon. Cependant, si vous voulez programmer directement sur le terminal, il est sûrement la meilleure solution actuelle.