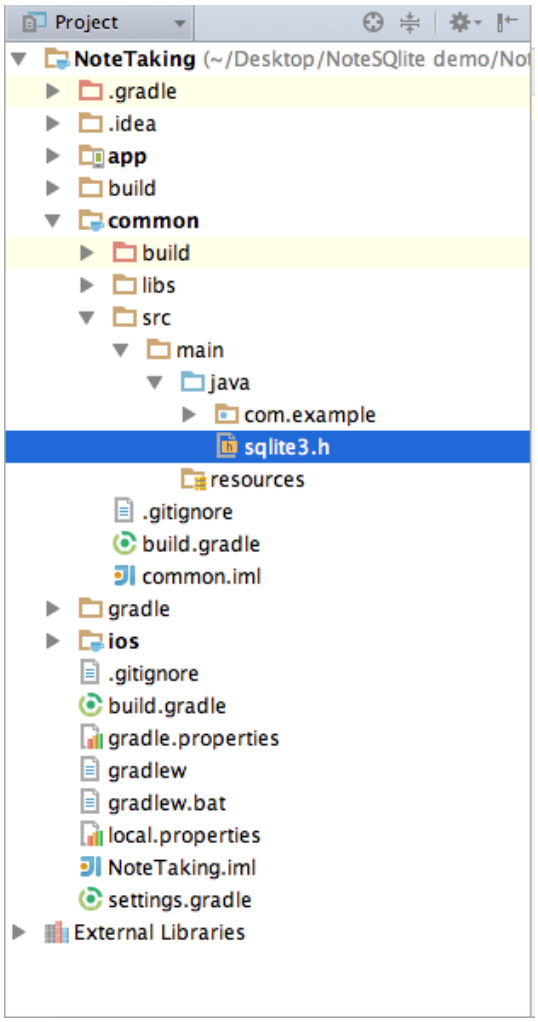
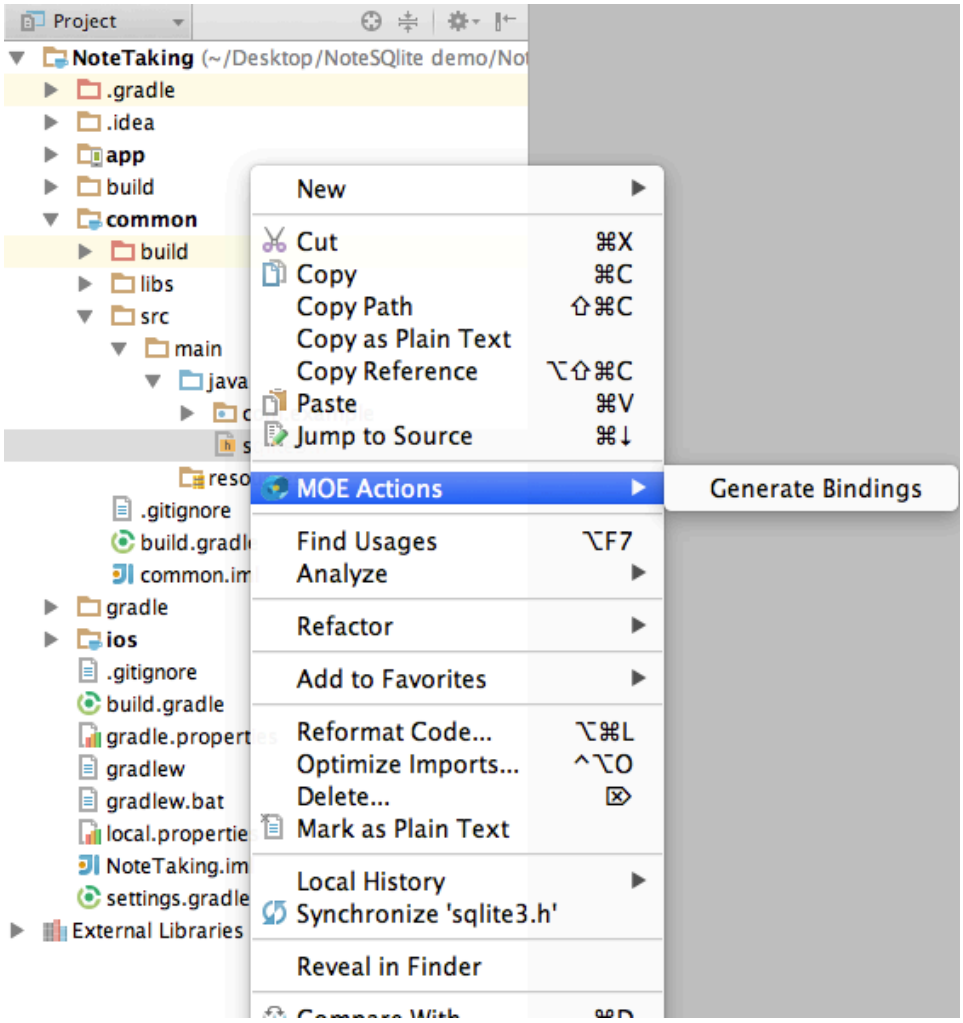
マルチ OS エンジンを使用した固定記憶域の操作 (テクノロジー・プレビュー) - パート 2

この記事は、インテル® デベロッパー・ゾーンに公開されている「Working with persistent storage using Multi-OS Engine (Technology Preview) - Part 2」の日本語参考訳です。

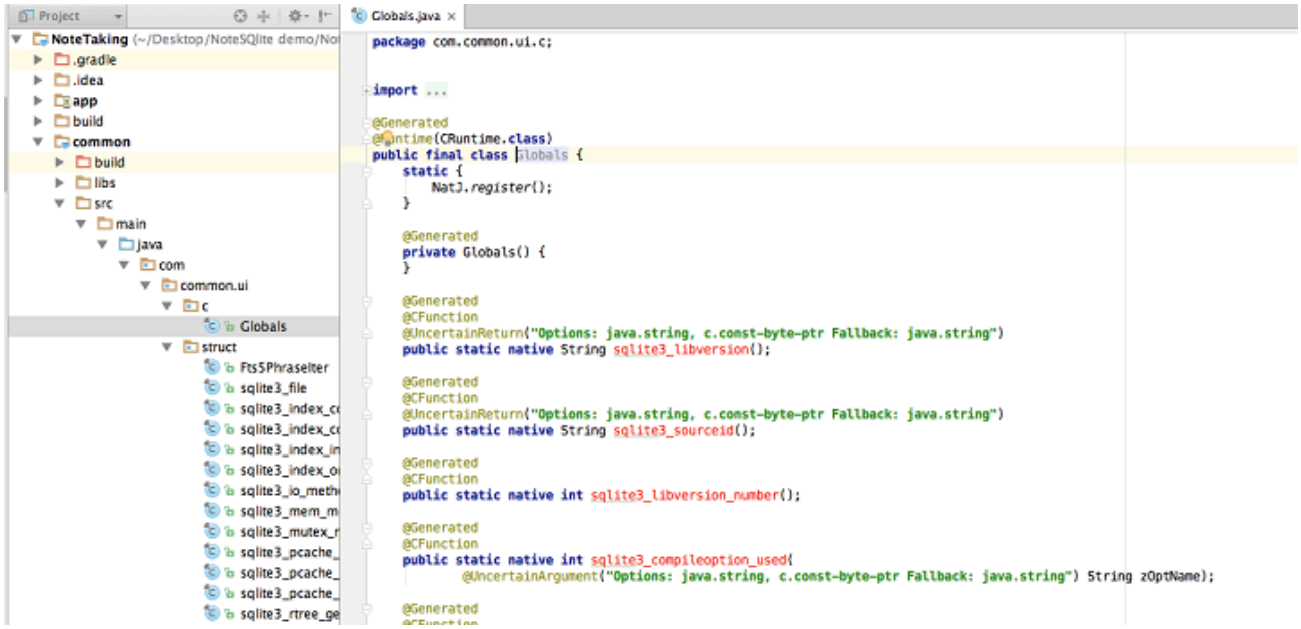
この記事は、固定記憶域の操作に関するチュートリアルのパート 2 です。ここでは、sqlite.h ファイルのバインディングを生成することで、iOS* および Android* で MOE を用いて SQLite ライブラリーを利用し、sqlite.h ファイルにアクセスする方法を示します。このチュートリアルのパート 1 はこちらからご覧になれます。ここでは、パート 1 で使用したアプリケーションを基に作業します。sqlite Web サイトから sqlite3.h をダウンロードして、共通ライブラリーのディレクトリーに配置します。



C ヘッダーファイルから Java* のバインディングを生成するには、[MOE Actions] > [Generate Bindings] メニューを選択します。



MOE により、sqlite に関連するすべての NatJ バインディングを含む新しいディレクトリーが作成されます。このバインディングを使用してデータベース・インスタンスを作成し、両プラットフォームに共通の CRUD 操作を実行します。



生成されたクラスファイルにアノテーション @Library("sqlite3") を追加します@Library アノテーションは、マークしたクラスが動作するためにロードする必要があるネイティブ・ライブラリーの名前を指定します。NatJ.register() が呼び出されると、NatJ は呼び出し元のクラスでこのアノテーションを検索し、NatJ.lookUpLibrary(...) で指定されたライブラリーのロードを試みます。

```
@Library("sqlite3")
@Runtime(CRuntime.class)
public final class Globals {
    static {
        NatJ.register();
    }
```

ネイティブ C ポインターにアクセスするため、MOE の NatJ バインディングも必要です。依存項目として、natj-api.jar ライブラリーを共通ライブラリー・フォルダーに追加します。この jar は github プロジェクトにあり、アプリケーションのプロジェクトに直接インポートすることができます。

次に、データベースのデータモデルとなる Note クラスを作成します。

```
public class Note implements Comparable<Note>{
    private Integer id;
    private String note;

    public void setId(int i){
        id =i;
    }

    public Integer getId(){
        return id;
    }

    public void setNote(String s){
        note = s;
    }
    public String getNote(){
        return note;
    }

    @Override
    public int compareTo(Note o) {
        return this.id.compareTo(o.id);
    }

}
```

データベースの作成・管理とノートの保存を行うための共通コードを作成します。

データベース・インスタンスのオープンとクローズには SQLiteDatabaseHelper.java を使用します。コンストラクターに、プラットフォーム固有のデータベース・ファイルのディレクトリー・パスが渡されます。onCreate メソッドは、ノートテーブルが存在しない場合、テーブルを新規作成します。

SQLiteDatabase.java クラスは、挿入、削除、更新、および選択クエリーを処理します。このクラスのメソッドは、ネイティブ sqlite 操作を抽象化します。例えば、以下は挿入クエリーの実装です。

```
@Override
    public void insert(String note) {
        StringBuilder sql = new StringBuilder();
        sql.append("INSERT");
        sql.append(" INTO ");
        sql.append(TABLE);
        sql.append(" (note) values (");
        sql.append("\"");
        sql.append(note);
        sql.append("\")");
        execSQL(sql.toString());
    }
```

execSQL メソッドは、SQL クエリーを実行する SQL ステートメントを準備します。

```
@Override
    public void execSQL(String statement) {
        SQLiteStatement stmt = new SQLiteStatement(statement, null);
        if (stmt.prepare(dbHandle)) {
            if (!stmt.exec()) {
                System.err.println("Error executing - " + stmt.getLastError());
            }
        } else {
            System.err.println("Error executing - " + stmt.getLastError());
            System.err.println("\tin: " + stmt.getStatement());
        }
    }
```

SQLiteStatement.java は、prepare()、exec()、query()、strep()、close() のような sqlite ライブラリーのネイティブルーチンを処理します。このクラスは、バインディングによって生成されたネイティブ C API 呼び出しを使用します。以下は、prepare() メソッドと exec() メソッドの実装例です。

```
public boolean prepare(VoidPtr dbHandle) {
    if (dbHandle == null) {
        throw new NullPointerException();
    }
    this.dbHandle = dbHandle;

    @SuppressWarnings("unchecked")
    Ptr<VoidPtr> stmtRef = (Ptr<VoidPtr>) PtrFactory.newPointerPtr(
        Void.class, 2, 1, true, false);
    int err = Globals.sqlite3_prepare_v2(dbHandle, statement, -1, stmtRef,
        null);
    if (err != 0) {
        lastError = Globals.sqlite3_errmsg(dbHandle);
        return false;
    }
    stmtHandle = stmtRef.get();
    int idx = 0;
    for (Object bind : bindArgs) {
        idx++;
        if (bind instanceof String) {
            err = Globals.sqlite3_bind_text(stmtHandle, idx, (String)bind, -1,
                new Globals.Function_sqlite3_bind_text(){
                    @Override
```

```
                    public void call_sqlite3_bind_text(VoidPtr arg0){}
                });
            } else if (bind instanceof Integer) {
                err = Globals.sqlite3_bind_int(stmtHandle, idx,  (Integer) bind);
            } else if (bind instanceof Long) {
                err = Globals.sqlite3_bind_int64(stmtHandle, idx,  (Long) bind);
            } else if (bind instanceof Double) {
                err = Globals.sqlite3_bind_double(stmtHandle, idx,  (Double) bind);
            } else if (bind == null) {
                err = Globals.sqlite3_bind_null(stmtHandle, idx);
            } else {
                lastError = "No implemented SQLite3 bind function found for " +
                    bind.getClass().getName();
                return false;
            }
            if (err != 0) {
                lastError = Globals.sqlite3_errmsg(dbHandle);
                return false;
            }
        }
        return true;
    }
    public boolean exec() {
        if (stmtHandle == null) {
            throw new RuntimeException("statement handle is closed");
        }
        //LOG.debug("Execing " + statement);
        int err = Globals.sqlite3_step(stmtHandle);
        if (err == 101 /* SQLITE_DONE */) {
            affectedCount = Globals.sqlite3_changes(dbHandle);
            lastInsertedID = Globals.sqlite3_last_insert_rowid(dbHandle);
        }
        close();
        if (err != 101 /* SQLITE_DONE */) {
            lastError = Globals.sqlite3_errmsg(dbHandle);
            return false;
        }
        return true;
    }
```

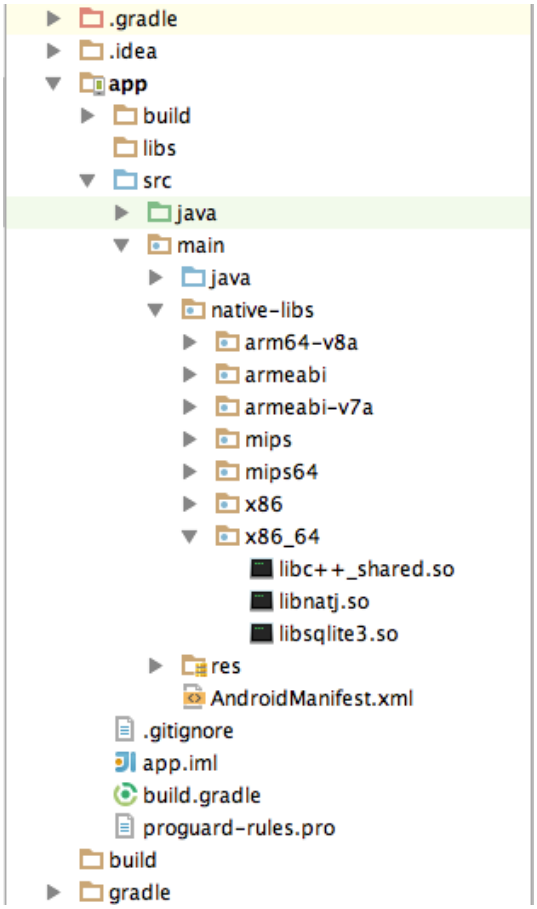
上記のコード例を基に、sqlite ステートメント・クラスのさまざまなメソッドを実装できます。

SQLiteCursor.java クラスは、ステートメント・クラスのデータ抽出 API を処理します。以下のメソッドは、実行したクエリーから整数値と文字列を取得します。

```
    @Override
        public String getString(int i) {
            if (stmt == null) {
                throw new RuntimeException("statement is closed");
            }
            return Globals.sqlite3_column_text(stmt.getStmtHandle(), i);
        }

    @Override
        public int getInt(int i) {
            if (stmt == null) {
                throw new RuntimeException("statement is closed");
            }
            return Globals.sqlite3_column_int(stmt.getStmtHandle(), i);
        }
```

Android* には専用の SQLite バージョンがあり、その実装はネイティブバージョンとわずかに異なるため、両プラットフォーム間でのコードの再利用が制限されます。そのため、ここでは NatJ バインディングを使用して、Android* 用にカスタムバージョンの sqlite を作成します。Android* が SQLite のネイティブ C 呼び出しを処理できるように、ネイティブ SQLite のダイナミック共有オブジェクト・ライブラリーを作成して、ランタイムにリンクします。sqlite.so のほかにも、libnatj.so と libc++_shared.so の 2 つのライブラリーをプロジェクトにリンクします。libnatj.so は、MOE の NatJ 呼び出しを使用するための NatJ ブリッジです。libc++_shared.so は、Android* 4.x をサポートするためのものです。Android* 5.x では不要です。



.so ネイティブ・ライブラリーを使用できるように、build.gradle ファイルを変更する必要があります。

最初に、sourceSets で jnilibs に src ディレクトリーを指定します。

```
sourceSets {
    main {
        manifest.srcFile 'src/main/AndroidManifest.xml'
        java.srcDir 'src'
        res.srcDir 'res'
        assets.srcDir 'assets'

        jniLibs.srcDir 'src/main/native-libs'
        jni.srcDirs = []    // ndk-build の自動呼び出しを無効にする
    }
}
```

次に、ネイティブ・ライブラリーの jar ファイルを作成します。

```
task nativeLibsToJar(type: Zip, description: 'create a jar archive of the native
libs') {
    destinationDir file("$buildDir/native-libs")
    baseName 'native-libs'
    extension 'jar'
    from fileTree(dir: 'src/main/native-libs', include: '**/*.so')
    into 'lib/'
}
```

そして、jar ファイルをビルドします。

```
tasks.withType(org.gradle.api.tasks.compile.JavaCompile) {
    compileTask -> compileTask.dependsOn nativeLibsToJar
}

clean.dependsOn 'cleanCopyNativeLibs'

tasks.withType(com.android.build.gradle.tasks.PackageApplication) {
    pkgTask ->
        pkgTask.jniFolders = new HashSet()
        pkgTask.jniFolders.add(new File(buildDir, 'native-libs'))
}
```

iOS* はネイティブバージョンの sqlite でビルドされているため、ライブラリーをリンクせずに直接使用できます。

次に、iOS* と Android* 向けに、データベース呼び出しのためのプラットフォーム固有のコードを作成します。

iOS* 固有コード

SQLiteDatabaseHelper 共通クラスを拡張する SQLiteDatabaseHelper クラスを作成します。iOS* 固有のドキュメント・ディレクトリーのパスを取得するため、getDocumentsPath() メソッドのみオーバーライドします。

```
@Override
    protected String getDocumentsPath() {
        NSArray paths = Foundation.NSSearchPathForDirectoriesInDomains(
                NSSearchPathDirectory.DocumentDirectory,
                NSSearchPathDomainMask.UserDomainMask, true);
        return (String) paths.firstObject();
    }
```

MasterViewController の viewDidLoad() メソッドで、コンストラクターにデータベースの filepath を渡して、データベースヘルパー・インスタンスを作成します。データベース・インスタンスを作成し、getWritableDatabase() メソッドでその参照を取得します。

```
    @Override
    @Selector("viewDidLoad")
    public void viewDidLoad() {
        // ビューをロード (通常は nib から) した後、追加のセットアップを行う
        navigationItem().setLeftBarButtonItem(editButtonItem());
        ISQLiteDatabaseHelper helper = new SQLiteDatabaseHelper(dbFileName);
        db = helper.getWritableDatabase();
}
```

追加ボタンにリスナーメソッドを追加し、データベースへの挿入クエリーを実行します。

```
@Selector("insertNewObject:")
    public void insertNewObject(Object sender){

        db.insert(defaultText);
        makeObjects();
        NSIndexPath indexPath = NSIndexPath.indexPathForRowInSection(0,0);
        tableView().insertRowsAtIndexPathsWithRowAnimation((NSArray)
NSArray.arrayWithObject(indexPath), UITableViewRowAnimation.Automatic);
        performSegueWithIdentifierSender("showDetail", this);
    }
```

DetailViewController で、更新 API を呼び出してデフォルトのテキストへの変更を更新します。テキストが空白の場合はノートを削除します。

```
@Selector("doSaveNote:")
    public void doSaveNote( Object sender){
        if (detailNote == null || db == null) {
            return;
        }
        if(!dataText.text().equals("")){
            detailNote.setNote(dataText.text());
            db.update(detailNote);
        }else{
            db.delete(detailNote.getId());
        }

    }
```

Android* 固有コード

iOS* と同様に、AndroidDatabaseHelper クラスを作成し、Android* の sqlite ファイルのディレクトリー・パスを渡すように getDocumentsPath() をオーバーライドします。

```
@Override
    protected String getDocumentsPath() {
        return Environment.getExternalStorageDirectory().getPath();
    }
```

iOS* と同様に、データベースヘルパー・インスタンスとデータベース・インスタンスを作成します。つまり、iOS* の viewDidLoad() に相当する MainActivity.java の onCreate() を作成します。

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        SQLiteDatabaseHelper dbHelper = new AndroidSQLiteDatabaseHelper(
            AndroidSQLiteDatabaseHelper.DB_NAME);
        db = dbHelper.getWritableDatabase();
...
}
```

追加ボタンにリスナーメソッドを追加し、データベースへの挿入を実行します。

```
public void insertNewObject() {
    db.insert(DEFAULT_TEXT);
    makeObjects();
}
```

ユーザー入力に応じて、EditorActivity.java は更新/削除クエリーを呼び出します。

```
private void finishEditing() {
    String newText = editor.getText().toString().trim();
    if (newText.equals("")){
        newText = DEFAULT_TEXT;
        db.delete(noteId);
    } else if (!newText.equals(note)){
        noteDetail.setNote(newText);
        db.update(noteDetail);
    }
    finish();
}
```

アプリケーションを実行すると、プラットフォームでデータベース・ファイルが作成されます。

このように、両プラットフォームで同じ sqlite 実装を利用して、コードのロジックを再利用できます。これにより、コーディングの労力を大幅に軽減でき、コードの保守が容易になります。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。