

Курсовая работа

“Разработка расширения языка Java для работы с реляционными базами данных”

Выполнил Никитин Павел Антонович
Кафедра системного программирования СПбГУ

Руководитель: Гуров Вадим Сергеевич
(кандидат наук, программист ООО "ИнтелиДжей Лабс")
9 мая 2009

1 Введение

Для разработчиков программного обеспечения могут быть полезны всевозможные проблемно-ориентированные расширения языков программирования общего назначения. Например, разработчики приложений для банковских нужд оценят встроенную в язык поддержку работы с денежными единицами. Но традиционные текстовые языки обязаны обладать однозначной грамматикой, что делает их трудно расширяемыми. Целью данной работы было именно расширение языка общего назначения (Java) и добавление в него конструкций для работы с реляционными базами данных. Для решения проблемы возможной неоднозначности грамматики полученного языка был выбран языковой инструментарий JetBrains MPS (Meta Programming System) – средство для создания языков и интегрированная среда для разработки на них. MPS решает эту проблему, работая непосредственно с абстрактным синтаксическим деревом программы, для редактирования которого используется текстово-подобный проекционный редактор. Однако, языково-ориентированное программирование в таком виде, как оно существует сегодня – достаточно молодая парадигма. Например, как другой заметный его представитель –

Intentional Software, так и MPS выходят из beta только в этом году.

2 О работе с реляционными БД

Тем не менее, потребность удобной работы с базами данных возникла давно. Поэтому существует масса ad-hoc решений для различных языков программирования. Самый известный из них – так называемый Embedded SQL. При таком подходе программа должна быть обработана специальным препроцессором, прежде чем она будет откомпилирована компилятором базового языка программирования. Препроцессор распознаёт вызовы запросов внутри языковых предложений и преобразует их в библиотечные вызовы, также распознаются специальным образом оформленные ссылки на переменные базового языка внутри SQL-предложений и некоторые другие менее фундаментальные конструкции – т. е. происходит трансляция. Данный подход доступен для языков C, Ada, Cobol, Fortran.

Похожий подход предлагает и стандарт SQLJ для языка Java. В этом стандарте в качестве конечной библиотеки так или иначе используется JDBC, входящая в стандартную поставку от Sun начиная с JDK версии 1.1, но чаще всего она обернута в проприетарный код, как, например, в реализации SQLJ от Oracle. Пример кода на SQLJ:

```
int i, j;  
i = 1;  
#sql { SELECT field INTO :OUT j  
        FROM table  
        WHERE id = :IN i };  
System.out.println(j);
```

О JDBC мы поговорим подробнее, так как именно она является основой для многих решений, работающих с базами, например Hibernate. В принципе, для несложных приложений вполне применимо использование JDBC напрямую, без каких-либо обёрток, поэтому наряду с Embedded SQL будем рассматривать и решения, не использующие препроцессор.

Все эти решения объединяет одно – слабо выраженная структура.

Тем не менее, они реально используются для разработки программного обеспечения, и в процессе общения с ними разработчик может столкнуться с вытекающими отсюда трудностями. Во-первых, это нарушение статического синтаксиса запроса, например – опечатка в ключевом слове SELECT. Класс решений с препроцессором позволит найти такую ошибку на стадии трансляции, что в принципе, приемливо. Однако строки, явно попадающие в параметры функций JDBC не обрабатываются препроцессором и, следовательно, ошибка будет замечена только во время выполнения и приведёт к исключению SQLException, что неприемлемо даже при наличии покрытия тестами части кода, содержащей ошибку, так как отвлекает разработчика от решения актуальных задач. Во-вторых, это проблема вложенности запросов. В JDBC конструкция с вложением языков вида $Java_1[SQL_1[Java_2[SQL_2]]]$ может привести к трудно воспроизводимому нарушению синтаксиса формируемого запроса $SQL_1 \cup SQL_2$, если на уровне вложенности $Java_2$ используется нетривиальное ветвление. SQLJ же вовсе исключает возможность такой вложенности.

Ещё один интересный подход – создание, в отличие от SQLJ, не внешнего проблемно-ориентированного языка (со своим расширенным синтаксисом и другой грамматикой), а внутреннего, то есть использование средств базового языка (Java) для придания коду на нём подобия специального языка (для работы с реляционными БД в нашем случае). Данный подход реализуют такие библиотеки, как, например, jequel, squill, quaere. Пример кода на jequel:

```
final ARTICLE ARTICLE2 = ARTICLE.as("article2");
final Field OID = ARTICLE_C.OID.as("article_c_oid");

SqlString sql = select(ARTICLE2.OID, ARTICLE_C.OID)
                  .from(ARTICLE2, ARTICLE_C)
                  .where(ARTICLE2.OID.eq(OID));
```

Особняком стоит недоступное для Java-мира расширение языка C# LINQ to SQL, но его использование предполагает знание своего SQL-подобного синтаксиса помимо знания LINQ и SQL.

3 Значимость IDE

Конечно, наличия ошибок в коде можно избежать при достаточной дисциплине со стороны разработчика. Но для существенного повышения производительности труда используются интегрированные среды разработки с структурированной подсказкой вводимого кода, подсветкой ошибок, анализом потока управления, рефакторингом и прочими полезными функциями. Однако популярные среды разработки на Java, такие, как Eclipse и IntelliJ IDEA не поддерживают Embedded SQL ни самостоятельно, ни в качестве плагинов. Есть некая среда JDeveloper от Oracle, но её SQLJ привязан к их базе данных и их проприетарный Java-библиотеке. Но, даже если бы плагины для SQLJ для популярных сред были разработаны, совместимость поддерживаемых ими расширений с другими была бы под большим вопросом.

4 Мой подход

Моей целью было разрешение всех вышеназванных недостатков – с использованием среды JetBrains MPS, которая предоставляет готовую инфраструктуру для всех современных IDE-функций (см. выше). Кроме того, язык должен был выглядеть интуитивно и быть полностью интегрированным в Java – позволять использование любых java-выражений внутри SQL и писать SQL-запросы в любом месте, где допустимо java-выражение, с соблюдением типов, удобно итерироваться по результатам запросов на выборку, задавать схему базы данных. Для создания (внешнего проблемно-ориентированного) языка необходимо:

- Создать иерархию концептов (возможных вершин AST – Abstract Syntax Tree) для SQL в MPS так, чтобы она была интегрирована в соответствующую иерархию в Java (называемый в MPS `baseLanguage`)
- Создать текстово-подобные редакторы для них
- Создать дополнительную иерархию концептов (таких, как кортежи) для приведения данных SQL к `baseLanguage` и наоборот
- Создать комплементарную дополнительной иерархии библиотеку (`sql.runtime`)

- Создать генератор новых конструкций в Java, JDBC и вызовы библиотеки `sql.runtime`
- Создать статическую систему типов для проверки их на лету в IDE (не на этапе генерации, как в Embedded SQL и SQLJ)

5 Реализация

За основу для AST был взят стандарт SQL'92 и документация Oracle, которая преподносит грамматику в SQL в удобной форме и с развёрнутыми комментариями. Встала проблема обширности стандарта. Первоначально возникла идея создания инструмента для разбора грамматики и создания по ней AST в формате MPS, но первые шаги в этом направлении и общение с разработчиками, использующими MPS привело к тому, что от данной идеи отказались. Также отказались и от полного покрытия стандарта – это слишком однообразная работа, чтобы выполнять её последовательно в рамках обучения работе с MPS, особенно если учесть то, что все остальные подзадачи требуют завершения создания AST. Из опыта разработчиков следовало, что гораздо более продуктивным решением будет минимальная реализация некоего подмножества AST, необходимого для работы, с оглядкой на его дальнейшее расширение, чтобы по мере использования добавлять новые концепты, понадобятся во время решения конкретной задачи. Оказалось, что хорошая документация языка помогает довольно быстро справиться с этой задачей. Уже на этом этапе, после добавления концепта для схемы и редакторов, пользователь языка может вводить хорошо структурированные предложения на SQL, не нарушая его синтаксиса. На приведённом ниже примере видно, что редакторы в MPS не текстовые, а текстово-подобные, поэтому в редакторах можно использовать многострочные скобки для более чёткого выделения структуры.

На этом примере все вхождения строки `accounts`, кроме первого являются ссылками на объявление соответствующей таблицы, поэтому без всяких рефакторингов переименование вхождения в `CREATE TABLE accounts` влечёт изменения текста во всех ссылках.

Дополнительная иерархия концептов состоит из:

```

Schema
[driver: "com.mysql.jdbc.Driver"
url: "jdbc:mysql://localhost:3306/server"
login: "Morj"
password: "mrj"
]
DROP TABLE stats CASCADE...
DROP TABLE admins CASCADE...
DROP TABLE accounts CASCADE...
CREATE TABLE accounts [id INTEGER DEFAULT... PRIMARY KEY <constraint_state>
, login VARCHAR ( 256 ) DEFAULT... NOT NULL <constraint_state>
, pw_hash INTEGER DEFAULT... <constraint>
, UNIQUE [login]
ON COMMIT...
<physical_properties> <table_properties>
]
CREATE TABLE admins [id INTEGER DEFAULT... PRIMARY KEY <constraint_state>
, FOREIGN KEY [id] REFERENCES accounts [id]
ON COMMIT...
<physical_properties> <table_properties>
]
CREATE TABLE stats [id INTEGER DEFAULT... NOT NULL <constraint_state>
, action VARCHAR ( 256 ) DEFAULT... NOT NULL <constraint_state>
, timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL <constraint_state>
, FOREIGN KEY [id] REFERENCES accounts [id]
, PRIMARY KEY [id, action, timestamp]
ON COMMIT...
<physical_properties> <table_properties>
]
INSERT <hint>
INTO accounts [id, login, pw_hash]
VALUES [1, "user1", 1], [2, "user2", 2]
<returning_clause>

```

Рис. 1: Пример кода схемы базы данных

1. Типа для кортежей наподобие generics в java
2. Типа для параметров кортежей – ссылок на колонки в таблицах и встроенные типы
3. Типа для экземпляров кортежей – для модификации (добавления строк, например) коллекций кортежей полученных в результате запросов на выборку или подготовки их в качестве параметров вставки

Этой иерархии достаточно для произвольных манипуляций с данными, возможно формирование коллекции кортежей определённого типа как с нуля, так и из базы, комбинируя эти методы прозрачным образом – без

resultSet'ов JDBC, которые уже невозможно передать в другой запрос после создания.

```
public int login(String login, int pw_hash) throws SQLException {
    TableRows < accounts . id > t = SELECT <hint> <modifier> [accounts . id]
                                   FROM accounts
                                   WHERE [accounts . login = login] AND [accounts . pw_hash = pw_hash]
                                   <hierarchical_query_clause>
                                   <group_by_clause>
                                   HAVING...
                                   <...(<subquery>)>
                                   <order_by_clause>

    System.out.println("login: " + login);
    if (t.isEmpty()) {
        System.out.println("no such user or incorrect password!");
        return 0;
    } else {
        int user_id = t.first.id;
        int timestamps_count = INSERT <hint>
                                INTO stats [id, action]
                                VALUES [user_id, "login"]
                                <returning_clause>

        System.out.println(login + " logged in, id: " + user_id + ", timestamps added: " + timestamps_count);
        return user_id;
    }
}
```

Рис. 2: Пример работы с результатом выборки

```
public void logout(int user_id) throws SQLException {
    TableRows < stats . id, stats . action > test = new TableRows < stats . id, stats . action >;
    for (char n = 'A'; n <= 'C'; n++) {
        test.add( user_id, "logout " + n );
    }
    System.out.println("logout: " + user_id + ", timestamps added: " + INSERT <hint>
                                INTO stats [id, action]
                                VALUES test
                                <returning_clause>
    );
}
```

Рис. 3: Пример подготовки коллекции для INSERT

Ключевым компонентом иерархии библиотеки sql.runtime является класс TableRow, который при создании из выборки принимает при создании resultSet и собирает метаинформацию об именах параметров кортежей, содержит аксессорные методы для доступа к параметрам из сгенерированного кода. При подготовке запроса без выборки TableRow принимает генерированную метаинформацию и соответствующие данные. Коллекции – списки этих TableRow. Для вставки таких коллекций в SQL существует класс TableRowExtractor, который получает в генерированном

коде метаинформацию и раскрывает во время исполнения коллекцию с конкретными данными в текст часть SQL-запроса.

Пример генерированного кода для приведённого выше метода `logout`:

```
public void logout(int user_id) throws SQLException {
    List<TableRow> test = ListOperations.<TableRow>
        createList();
    for(char n = 'A' ; n <= 'C' ; n++ ) {
        ListSequence.fromList(test).addElement(
            new TableRow(new String[]{ "id", "action" },
                new Object[]{ user_id, "logout_" + n } )
        );
    }
    System.out.println("logout:_" + user_id +
        ",_timestamps_added:_" +
        ConnectionManager.update("INSERT INTO stats (" +
            "id, action" + ") " + "_VALUES" +
            TableRowExtractor.getPresentation(test,
                new String[]{ "id", "action" })))
    );
}
```

Класс `ConnectionManager` оборачивает вызовы JDBC к БД. Видно, что в данном примере в INSERT будет передана полученная во время исполнения следующая коллекция кортежей:

```
(( user_id , 'logout_A' ),
 ( user_id , 'logout_B' ),
 ( user_id , 'logout_C' ))
```

Генератор в MPS состоит из наборов макросов, применяемых к вершинам AST, пока возможно. Реализация генератора моего языка преобразует его концепты в конкатенации строк, которые формируются с помощью `sql.runtime` и передаются `ConnectionManager` для исполнения, а также работу с классом `TableRow` (см. примеры выше).

Система типов в MPS представляет собой набор уравнений, описывающий отношения типов вершин AST в зависимости от их взаимного расположения и является легко расширяемой вследствие своей декларативности. Моя система типов описывает соотношения между базовыми

типами SQL и Java, типизацию кортежей и их коллекций, причём, в отличие от generics, кортеж, множество типов которого содержит множество типов второго, является его подтипом. Используются средства MPS, позволяющие выдавать понятные сообщения об ошибках.

```
public TableRows < accounts . login, stats . timestamp, stats . action > allStats() throws SQLException {
    return SELECT <hint> <modifier> [accounts . login, stats . action];
        FROM stats <join type> JOIN
            accounts ON st
Typesystem error: TableRows <accounts.login, stats.timestamp, stats.action> is expected
WHERE...
<hierarchical_query_clause>
<group_by_clause>
HAVING...
<...(<subquery>)>
ORDER BY stats . timestamp
}
```

Рис. 4: Пример кода с ошибкой

6 Заключение

В результате был получен расширяемый язык для работы с реляционными базами данных с полноценной IDE-инфраструктурой на базе JetBrains MPS, с интеграцией в Java, с широкой возможностью расширения, генерируемый в распространённый стандарт JDBC. Использование этого языка потенциально эффективнее существующих для решений встроенного SQL в Java. При этом затраченные ресурсы на порядок меньше, чем при создании IDE или плагина для SQLJ с нуля, а результат более гибок, чем плагины к IDE существующим. Таким образом, MPS представляется отличным средством для создания внешних проблемно-ориентированных языков, а использование таких языков – хорошим способом продуктивного создания программ, работающих с базами данных. Мощь этих языков возрастает благодаря возможности комбинирования. В частности, созданный язык автоматически становится совместим с языком для работы с коллекциями, входящим в стандартную поставку MPS, что позволяет писать более компактный код.

Дальнейшие возможности улучшения:

- Добавление типизации для встроенных в SQL функций
- Оптимизация формирования запросов во время исполнения
- Проверка типов для вложенных SQL-запросов

```

public static void main(String[] args) throws SQLException {
    CONNECT Schema {
        Logic logic = new Logic();
        logic.allStats().forEach({~stat => System.out.println("on: " +
stat.timestamp + " user " + stat.login + " did " + stat.action); });
    }
}

```

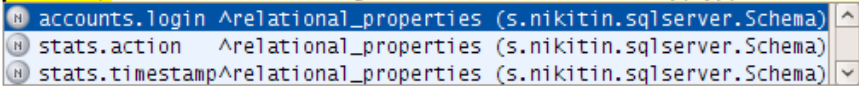


Рис. 5: MPS autocomplete, MPSSQL (мой язык), MPS collections language