Java 6 RowSet 使用完全剖析

本文内容包括:

- RowSet 简介
- 类继承结构
- 实验环境
- 使用 CachedRowSet
- 使用 WebRowSet
- 使用 FilteredRowSet
- 使用 JdbcRowSet
- 使用 JoinRowSet
- 结束语
- 下载
- 参考资料
- 作者简介
- 对本文的评价

级别: 中级

徐 睿智 (xuruizhi@cn.ibm.com), 软件开发工程师, IBM 中国软件开发中心
刘 威 (liuwei_cqu@hotmail.com), 实习生
许 彬 (terry.xubin@gmail.com), 实习生

2008 年 6 月 05 日

C# 提供了 DataSet，可以将数据源中的数据读取到内存中，进行离线操作，然后再同步到数据源。同样，在 Java 中也提供了类似的实现，即 RowSet。`javax.sql.rowset` 包下，定义了五个不同的 RowSet 接口，供不同的场合使用。本文将分别对这五个 RowSet 的使用场合以及详尽用法进行介绍，并且描述使用中可能出现的问题，以提醒读者在实际使用时绕开这些问题。

RowSet 简介

`javax.sql.rowset` 自 JDK 1.4 引入，从 JDK 5.0 开始提供了参考实现。它主要包括 CachedRowSet，WebRowSet，FilteredRowSet，JoinRowSet 和 JdbcRowSet。 除了 JdbcRowSet 依然保持着与数据源的连接之外，其余四个都是 Disconnected RowSet。

相比较 `java.sql.ResultSet` 而言，RowSet 的离线操作能够有效的利用计算机越来越充足的内存，减轻数据库服务器的负担，由于数据操作都是在内存中进行然后批量提交到数据源，灵活性和性能都有了很大的提高。RowSet 默认是一个可滚动，可更新，可序列化的结果集，而且它作为 JavaBeans，可以方便地在网络间传输，用于两端的数据同步。

回页首

类继承结构

RowSet 继承自 ResultSet，其他五个 RowSet 接口均继承自 RowSet。下图是它们的继承关系。

图 **1.** 继承结构图

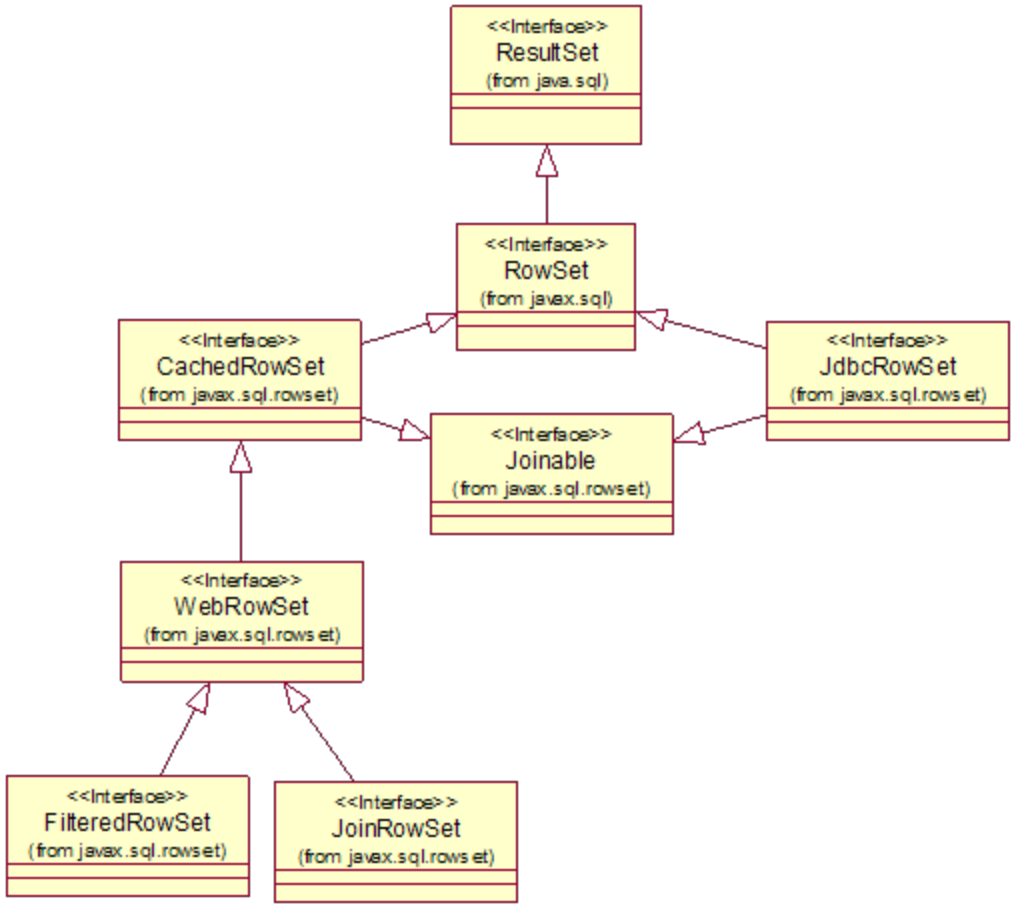

表 **1. RowSet** 接口说明

CachedRowSet	最常用的一种 RowSet。其他三种 RowSet（WebRowSet，FilteredRowSet，JoinRowSet）都是直接或间接继承于它并进行了扩展。它提供了对数据库的离线操作，可以将数据读取到内存中进行增删改查，再同步到数据源。可串行化，可作为 JavaBeans 在网络间传输。支持事件监听，分页等特性。
WebRowSet	继承自 CachedRowSet，并可以将 WebRowSet 写到 XML 文件中，也可以用符合规范的 XML 文件来填充 WebRowSet。
FilteredRowSet	通过设置 Predicate（在 `javax.sql.rowset` 包中），提供数据过滤的功能。可以根据不同的条件对 RowSet 中的数据进行筛选和过滤。
JoinRowSet	提供类似 SQL JOIN 的功能，将不同的 RowSet 中的数据组合起来。目前在 Java 6 中只支持内联（Inner Join）。
JdbcRowSet	对 ResultSet 的一个封装，使其能够作为 JavaBeans 被使用，是唯一一个保持数据库连接的 RowSet。

回页首

实验环境

本文示例的实验环境如下：

- Java 环境：Sun JDK 6.0
- 数据库：derby-10.3.1.4
- 数据库名：TESTDB
- 数据库用户名及密码：均使用 derby 默认用户名和密码。
- 表及测试数据：创建两个表：CUSTOMERS 和 ORDERS，并分别插入测试数据。
- 示例代码以附件形式提供 下载。

表 **2.** 表 **CUSTOMERS**

ID	NAME	REMARK
1	Tom	Tom is VIP
2	Jim	null

表 **3.** 表 **ORDERS**

ID	USER_ID	PRODUCT
1	1	Book
2	1	Computer
3	2	Phone

回页首

使用 **CachedRowSet**

填充 **CachedRowSet** 的两种方式

CachedRowSet 提供了两个用来获取数据的方法，一个是 execute()，另一个是 populate(ResultSet)。

使用 execute() 填充 CachedRowSet 时，需要设置数据库连接参数和查询命令 command，如下示例代码：

清单 **1.** 使用 **execute()**

```
cachedRS.setUrl(DBCreator.DERBY_URL);
cachedRS.setCommand(DBCreator.SQL_SELECT_CUSTOMERS);
// derby 默认用户名和密码都是 "APP"，也可以不设置。
cachedRS.setUsername("APP"); //$NON-NLS-1$
cachedRS.setPassword("APP"); //$NON-NLS-1$
cachedRS.execute();
```

cachedRS 根据设置的 url、username、password 三个参数去创建一个数据库连接，然后执行查询命令 command，用结果集填充 cachedRS，最后关闭数据库连接。execute() 还可以直接接受一

个已经打开的数据库连接，假设 conn 为一个已经打开的数据库连接，下段示例代码与上段代码结果一致：

清单 **2.** 使用 **execute(Connection)**

```
cachedRS.execute(conn);
cachedRS.setCommand(DBCreator.SQL_SELECT_CUSTOMERS);
cachedRS.execute();
```

填充 CachedRowSet 的第二个方法是使用 `populate(ResultSet)`。

清单 **3.** 使用 **populate(ResultSet)**

```
ResultSet rs = stmt.executeQuery(DBCreator.SQL_SELECT_CUSTOMERS);
cachedRS.populate(rs);
rs.close();
```

CachedRowSet 本身也是继承于 ResultSet，因此，也可以用一个有数据的 CachedRowSet 来填充另一个 CachedRowSet。

更新、删除、插入数据

更新数据。先把游标（cursor）移到要更新的行，根据每列的类型调用对应的 `updateXXX(index, updateValue)`，再调用 `updateRow()` 方法。此时，只是在内存中更新了该行，同步到数据库需要调用方法 `acceptChanges()` 或 `acceptChanges(Connection)`。如果 CachedRowSet 中保存着原数据库连接信息，则可以调用 `acceptChanges()`；否则，则应该传入可用的数据库连接或重新设置数据库连接参数。下段示例代码更新第一行的第二列。

清单 **4.** 更新

```
cachedRS.first();
cachedRS.updateString(2, "Hello"); //$NON-NLS-1$
cachedRS.updateRow();
cachedRS.acceptChanges();
```

删除数据。把游标移到要删除的行，调用 `deleteRow()`，再同步回数据库即可。

清单 **5.** 删除

```
cachedRS.last();
```

```
cachedRS.deleteRow();
cachedRS.acceptChanges();
```

在删除数据时，需要注意布尔值 showDeleted 这个属性的使用。CachedRowSet 提供了 `getShowDeleted()` 和 `setShowDeleted(boolean value)` 两个方法来读取和设置这个属性。showDeleted 是用来判断被标记为删除且尚未同步到数据库的行在 CachedRowSet 中是否可见。true 为可见，false 为不可见。默认值为 false。

插入数据。插入操作稍微比更新和删除复杂。先看下段示例代码。

清单 **6.** 新增

```
cachedRS.last();
cachedRS.moveToInsertRow();
cachedRS.updateInt(1, 3);
cachedRS.updateString(2, "Bob"); //$NON-NLS-1$
cachedRS.updateString(3, "A new user"); //$NON-NLS-1$
cachedRS.insertRow();
cachedRS.moveToCurrentRow();
cachedRS.acceptChanges();
```

新插入的行位于当前游标的下一行。本例中，先把游标移到最后一行，那么在新插入数据后，新插入的行就是最后一行了。在新插入行时，一定要先调用方法 `moveToInsertRow()`，然后调用 `updateXXX()` 设置各列值，再调用 `insertRow()`，最后再把游标移到当前行。注意一定要遵循这个步骤，否则将抛出异常。

冲突处理

当我们使用 CachedRowSet 更新数据库时，有可能因为内存中的数据过期而产生冲突。此时更新数据库的方法 `acceptChanges()` 会抛出 `SyncProviderException`，由此我们可以捕获产生冲突的原因并手动进行解决。

清单 **7.** 冲突

```
ResultSet rs = stmt.executeQuery(DBCreator.SQL_SELECT_CUSTOMERS);
CachedRowSet cachedRS = new CachedRowSetImpl();
cachedRS.populate(rs);
cachedRS.setUrl(DBCreator.DERBY_URL);

// 修改数据库中的数据
stmt.executeUpdate("UPDATE CUSTOMERS SET NAME = 'Terry' WHERE ID = 1");

// 在 CachedRowSet 中更新同一行
```

```
cachedRS.absolute(1);
cachedRS.updateString(3, "Tom is not VIP");
cachedRS.updateRow();

SyncResolver resolver = null;
try {
    cachedRS.acceptChanges();
} catch (SyncProviderException e) {
    resolver = e.getSyncResolver();
}

while (resolver.nextConflict()) {
    System.out.println(resolver.getStatus());
}
```

我们首先填充 cachedRS，然后在数据库中直接修改 ID 为 1 的行，将 NAME 字段设为 "Terry"，同时用 cachedRS 修改 ID 为 1 的行的 REMARK 字段，最后使用 acceptChanges 跟数据库进行同步。此时 cachedRS 中记录的原始值与数据库中的值不一致，从而产生冲突，抛出 SyncProviderException，数据也会更新失败。接下来我们通过 SyncProviderException 得到 SyncResolver 实例并遍历了产生的所有冲突。

SyncResolver 继承了 RowSet 接口，我们可以像使用一般 RowSet 一样操作它。SyncResolver 的实例拥有与正在同步的 RowSet 相同的行数和列数。使用 `nextConflict()` 和 `previousConflict()` 可以遍历所有产生的冲突，`getStatus()` 可以获得冲突的类型。在 SyncResolve 中定义了四种类型，分别是：DELETE_ROW_CONFLICT，INSERT_ROW_CONFLICT，NO_ROW_CONFLICT，UPDATE_ROW_CONFLICT。上例中产生的是 UPDATE_ROW_CONFLICT。

注：目前 Sun JDK 对 SyncResolver 的支持非常有限，只实现了 SyncResolver 接口中定义的方法，调用从 RowSet 接口继承的方法都会抛出 `UnsupportedOperationException` 异常；`getConflictValue()` 返回都是 null。

事件监听

一个监听器需要实现 RowSetListener 接口。RowSetListener 支持三种事件监听：cursor moved、row changed 和 rowSet changed。假定 Listener 实现了 RowSetListener 接口，看示例代码。

清单 **8.** 注册事件监听器

```
Listener listener = new Listener();
cachedRS.addRowSetListener(listener);
updateOnRowSet(cachedRS);
cachedRS.removeRowSetListener(listener);
```

`updateOnRowSet()` 所做的操作就是将游标移到第一行，更新，再同步回数据库。在这个方法中，依次触发了 Listener 的三个事件。下表列出了 CachedRowSet 中会触发监听器的所有方法。

表 **4. CachedRowSet** 中会触发监听器的方法

	cursor moved	row changed	rowSet changed
absolute()	√		
relative()	√		
next()	√		
previous()	√		
first()	√		
last()	√		
beforeFirst()	√		
afterLast()	√		
updateRow()		√	
deleteRow()		√	
insertRow()		√	
undoDelete()		√	
undoUpdate()		√	
undoInsert()		√	
populate()			√
acceptChanges()			√
acceptChanges(Connection)			√
execute()			√
execute(Connection)			√
nextPage()			√
previousPage()			√
restoreOriginal()			√
release()			√

事务

事务对于保证数据的一致性是非常重要的。CachedRowSet 专门提供了处理事务的接口，从而可以保证同步数据的原子性和一致性。CachedRowSet 默认是不使用事务的。

清单 **9.** 事务代码一

```
cachedRS.absolute(1);
// 第一列不能为 null，更新时将产生冲突
cachedRS.updateNull(1);
cachedRS.updateRow();

cachedRS.next();
cachedRS.updateString(2, "Terry");
cachedRS.updateRow();
try {
```

```
    cachedRS.acceptChanges(conn);
} catch (SyncProviderException e) {
    // expected
}

rs = stmt.executeQuery(DBCreator.SQL_SELECT_CUSTOMERS);
cachedRS = new CachedRowSetImpl();
cachedRS.populate(rs);
printRowSet(cachedRS);
```

我们更新了 cachedRS 的第一行和第二行，并将第一行第一列错误的设置为 null。由于 CachedRowSet 默认不使用事务，第一行没有更新，而第二行更新成功。我们也可以手动控制事务的作用范围。

清单 **10.** 事务代码二

```
cachedRS.absolute(1);
cachedRS.updateNull(1);
cachedRS.updateRow();

cachedRS.next();
cachedRS.updateString(2, "Terry");
cachedRS.updateRow();
conn.setAutoCommit(false);
try {
    cachedRS.acceptChanges(conn);
    cachedRS.commit();
} catch (SyncProviderException e) {
    // expected
    cachedRS.rollback();
}
conn.setAutoCommit(true);
rs = stmt.executeQuery(DBCreator.SQL_SELECT_CUSTOMERS);
cachedRS = new CachedRowSetImpl();
cachedRS.populate(rs);
printCachedRowSet(cachedRS);
```

与前面的例子不同的是，这里我们关闭了自动提交模式，并且在同步失败后回滚了事务，避免了数据不一致的情况。

需要注意的是，如果需要手动控制事务的范围，在调用 execute 或 acceptChanges 时必须传入 Connection，否则再调用 CachedRowSet 的 `commit()` 或 `rollback()` 会抛 `NullPointerException`。实际上 CachedRowSet 依然是通过在内部保存 Connection 的引用来实现事务操作的。

分页

由于 CachedRowSet 是将数据临时存储在内存中，因此对于许多 SQL 查询，会返回大量的数据。如果将整个结果集全部存储在内存中会占用大量的内存，有时甚至是不可行的。对此 CachedRowSet 提供了分批从 ResultSet 中获取数据的方式，这就是分页。应用程序可以简单的通过 setPageSize 设置一页中数据的最大行数。也就是说，如果页大小设置为 5，一次只会从数据源获取 5 条数据。下面的代码示范了如何进行简单分页操作。（分页部分代码默认 ORDERS 表中有 10 条数据）

清单 **11.** 分页代码一

```
ResultSet rs = stmt.executeQuery(DBCreator.SQL_SELECT_ORDERS);
CachedRowSet cachedRS = new CachedRowSetImpl();
// 设置页大小
cachedRS.setPageSize(4);
cachedRS.populate(rs, 1);

while (cachedRS.nextPage()) {
printRowSet(cachedRS);
}

while (cachedRS.previousPage()) {
   printRowSet(cachedRS);
}
```

可以看到只需要在 populate 之前使用 setPageSize 设置页的大小，就可以轻松实现分页了。每次调用 nextPage 或 previousPage 进行翻页后，行游标都会被自动移动到当前页第一行的前面，并且只能在当前页内移动。这样我们对每一页都可以像新的数据集一样进行遍历，非常方便。这里需要注意的是：

1. 用来填充 CachedRowSet 的 ResultSet 必须是可滚动的（Scrollable）。
2. populate 必须使用有两个参数的版本，否则无法进行分页。读者可以将 `cachedRS.populate(rs, 1);` 换成 `cachedRS.populate(rs);` 看看会有怎样的情况发生。
3. `ResultSet rs` 必须在遍历完毕后才能关闭，否则翻页遍历时会抛 `SQLException`。

我们注意到在使用分页遍历数据集时，`nextPage()` 是最先被调用的，也就是说页与行相似，最开始的页游标是指向第 0 页的，通过 `nextPage()` 方法移到第一页，这样就可以用非常简洁的代码遍历数据集。那么如果在第 0 页时（第一次调用 nextPage）之前使用 next 遍历当前页会是怎样的结果呢？

清单 **12.** 分页代码二

```
ResultSet rs = stmt.executeQuery(DBCreator.SQL_SELECT_ORDERS);
CachedRowSet cachedRS = new CachedRowSetImpl();
// 设置页大小
cachedRS.setPageSize(4);
```

```
cachedRS.populate(rs, 1);

printRowSet(cachedRS);

while (cachedRS.nextPage()) {
    printRowSet(cachedRS);
}
```

我们看到第一页被输出了两次。也就是说使用 next 始终是可以遍历第一页的，而每次 nextPage 后，行游标都会被重置到当前页的第一行数据之前。对于使用 execute 填充数据的方式，通过 setPageSize 同样可以实现分页。

setMaxRows 可以设置 CachedRowSet 可遍历的最大行数。下面是 setMaxRows 和 setPageSize 同时使用的例子。

清单 **13.** 分页代码三

```
ResultSet rs = stmt.executeQuery(DBCreator.SQL_SELECT_ORDERS);
CachedRowSet cachedRS = new CachedRowSetImpl();
cachedRS.setPageSize(4);
cachedRS.setMaxRows(7);
cachedRS.populate(rs, 1);

while (cachedRS.nextPage()) {
   printRowSet(cachedRS);
}

rs.close();
```

上面的例子中，分别将页大小设置为 4，最大行数设置为 7（设置页大小时，如果最大行数不等于 0，就必须小于等于最大行数），然后遍历打印所有行，输出如下。

清单 **14.** 清单 **13** 中的代码执行结果

```
The data in RowSet:
1 1 Book
2 1 Compute
3 2 Phone
4 2 Java
The data in RowSet:
5 2 Test
```

```
6 1 C++
7 2 Perl
```

我们注意到，虽然满足 Select 条件的数据有 10 条，但由于我们使用 setMaxRows 设置了允许的最大行数，所以最终我们只得到了 7 条数据。另外 setPageSize 只会改变下一次填充页时的大小，无法影响当前页的大小。我们可以在数据填充完毕后再改变页的大小。

清单 **15.** 分页代码四

```
ResultSet rs = stmt.executeQuery(DBCreator.SQL_SELECT_ORDERS);
CachedRowSet cachedRS = new CachedRowSetImpl();
cachedRS.setPageSize(4);
cachedRS.populate(rs, 1);
cachedRS.setPageSize(3);

while (cachedRS.nextPage()) {
   printRowSet(cachedRS);
}

rs.close();
```

我们在 populate 数据之前设置的页大小是 4，在 populate 数据之后又将页大小设置为 3，然后遍历输出结果。

清单 **16.** 清单 **15** 中的代码执行结果

```
The data in RowSet:
1 1 Book
2 1 Compute
3 2 Phone
4 2 Java
The data in RowSet:
5 2 Test
6 1 C++
7 2 Perl
The data in RowSet:
8 1 Ruby
9 1 Erlang
10 2 Python
```

我们发现除了第一页有 4 条数据外，其余页都只有 3 条数据。实际上 populate 中就已经对第一页数据进行了填充，并且使用之前设置的 4 作为页大小。所以 populate 之后设置的页大小就只能从第二页开始起作用了。setMaxSize 与此相似，也是在每次填充页时计算。

应注意的问题

在 JDK 5.0 中，当删除一行中某列值为 null 时，会抛出 `NullPointerException`。例如，表 `CUSTOMERS` 中第二行第三列的值为 null。假设 cachedRS 里面填充着表 `CUSTOMERS` 的数据，那么下段代码在 JDK 5.0 下运行时会抛出 `NullPointerException`。在 JDK 6.0 中，此问题已得到修正。

清单 **17.**

```
cachedRS.absolute(2);
cachedRS.deleteRow();
cachedRS.acceptChanges();
```

回页首

使用 **WebRowSet**

WebRowSet 继承于 CachedRowSet，因此用来填充 CachedRowSet 的方式同样适用于 WebRowSet。WebRowSet 也可以读取一个符合规范的 XML 文件，填充自己。假定 CUSTOMERS.xml 是一个符合规范的 XML 文件，里面存放的是表 CUSTOMERS 的数据。

清单 **18.** 读取 **XML** 文件

```
WebRowSet newWebRS = new WebRowSetImpl();
newWebRS.readXml(new FileReader("CUSTOMERS.xml"));
```

相比于 CachedRowSet，WebRowSet 就是添加了对 XML 文件读写的支持。它可以将自身数据输出到 XML 文件，也可以读取符合规范的 XML 文件，来填充自己。上段示例代码中已经演示了如何读取 XML 文件。如下示例代码则是生成一个 XML 文件。

清单 **19.** 生成 **XML** 文件

```
WebRowSet webRS = new WebRowSetImpl();
CachedRowSetDemo.fillRowSetWithExecute(webRS);
// 输出到XML文件
FileWriter fileWriter = new FileWriter("CUSTOMERS.xml");
```

```
webRS.writeXml(fileWriter);
```

`fillRowSetWithExecute(CachedRowSet)` 是一个静态方法，它的功能是用表 `CUSTOMERS` 来填充传入的 CachedRowSet。

应注意的问题

1. 按照规范，当一行被标记为更新时，在输出到 XML 文件时，应使用标签 `<modifyRow>`，但实际输出为 `<currentRow>`。
2. 按照规范，当一行中某列被更新时，在输出到 XML 文件时，应使用标签 `<updateValue>`，但实际输出为 `<updateRow>`。
3. 按照规范，读取 XML 文件时，如果某行标签为 `<deleteRow>`，在读到 WebRowSet 中时，该行应被标记为删除，实际读取后，状态丢失。

回页首

使用 **FilteredRowSet**

FilteredRowSet 继承自 WebRowSet。正如它的名字所示，FilteredRowSet 是一个带过滤功能的 RowSet。它的过滤规则在 Predicate 中定义。Predicate 也是 `javax.sql.rowset` 包下的接口，它定义了三个方法：`boolean evaluate(Object value, int column)`，`boolean evaluate(Object value, String columnName)`，`boolean evaluate(RowSet rs)`。 前两个方法主要是用来检查新插入行的值是否符合过滤规则，符合，返回 true；否则，返回 false。FilteredRowSet 在新插入行时，会用这个方法来检测。如果不符合，会抛出 `SQLException`。`boolean evaluate(RowSet rs)` 这个方法则是用来判断当前 RowSet 里面的所有数据，是否符合过滤规则。FilteredRowSet 在调用有关移动游标的方法时，会使用这个方法进行检测。只要符合过滤规则的数据才会显示出来。下面我们给出了一个非常简单的 Predicate 实现，它的过滤规则是每行第一列的值只有大于 1 的才是有效行。

清单 **20. Rang.java**

```
class Range implements Predicate {

    @Override
    public boolean evaluate(RowSet rs) {
        try {
            if (rs.getInt(1) > 1) {
                return true;
            }
        } catch (SQLException e) {
            // do nothing
        }
        return false;
    }

    @Override
    public boolean evaluate(Object value, int column) throws SQLException {
        return false;
```

```
    }

    @Override
    public boolean evaluate(Object value, String columnName)
            throws SQLException {
        return false;
    }
}
```

下面这段代码演示了使用上面定义的 class Range 前后的数据变化。

清单 **21.** 设置过滤器

```
FilteredRowSet filterRS = new FilteredRowSetImpl();
CachedRowSetDemo.fillRowSetWithExecute(filterRS);
System.out.println("/*******Before set filter***********/");
CachedRowSetDemo.printRowSet(filterRS);

System.out.println("\n/*******After set filter***********/");
Range range = new Range();
filterRS.setFilter(range);
CachedRowSetDemo.printRowSet(filterRS);
```

清单 **22.** 清单 **21** 中的代码执行结果

```
/*******Before set filter***********/
The data in RowSet:
1 Tom Tom is VIP.
2 Jim null

/*******After set filter***********/
The data in RowSet:
2 Jim null
```

可以看出，在设置了过滤器后，再次打印 FilteredRowSet 中的数据时，由于第一行第一列的值为 1，不符合过滤规则，因此，没有打印出来。上面实现的这个 Predicate 中，用来检测新插入行是否符合过滤规则的方法直接返回 false。这样，在新插入行时，无论插入什么值时，都将抛出 SQLException。

应注意的问题

1. 当设置了过滤器后，`FilteredRowSet.absolute(1)` 无论何时都返回 false。按照规范，如果第一行值不符合过滤规则，则移到第二行，依次类推，直到找到第一条符合过滤规则的结果行并返回 true，否则，返回 false。
2. 在 JDK 5.0 下，如果 FilteredRowSet 没有设置过滤器，那么在新插入一行时会抛出 `NullPointerException`。在 JDK 6.0 中，已解决该问题。

回页首

使用 JdbcRowSet

JdbcRowSet 是对 ResultSet 的一个简单的封装，让它可以作为一个 JavaBeans 组件来使用。填充 JdbcRowSet 只能通过一种方式，即 `execute()` 方法。之后可以通过类似于操作 ResultSet 的方法来操作 JdbcRowSet。

清单 **23. JdbcRowSet**

```
JdbcRowSet jrs = new JdbcRowSetImpl();
jrs.setCommand(DBCreator.SQL_SELECT_CUSTOMERS);
jrs.setUrl(DBCreator.DERBY_URL);
jrs.execute();
while (jrs.next()) {
    for(int i = 1; i <= jrs.getMetaData().getColumnCount(); i++) {
        System.out.print(jrs.getObject(i) + " "); //$NON-NLS-1$
    }
    System.out.println();
}
```

清单 **24.** 清单 **23** 中的代码执行结果

```
1 Tom Tom is VIP.
2 Jim null
```

下面一节里我们将会看到 JdbcRowSet 如何作为一个 RowSet 和其他的 RowSet 一起使用。

回页首

使用 JoinRowSet

支持的联合方式

JoinRowSet 接口中对五种不同的联合方式都定义了对应的常数和判断该实现是否支持的方法，如下表所示。

表 **5.** 五种联合方式

联合方式	对应的常数	判断是否支持的方法（返回布尔值）
内连接（INNER JOIN）	JoinRowSet.INNER_JOIN	supportsInnerJoin()
左外连接（LEFT OUTER JOIN）	JoinRowSet.LEFT_OUTER_JOIN	supportsLeftOuterJoin()
右外连接（RIGHT OUTER JOIN）	JoinRowSet.RIGHT_OUTER_JOIN	supportsRightOuterJoin()
全外连接（FULL OUTER JOIN）	JoinRowSet.FULL_JOIN	supportsFullJoin()
交叉连接（CROSS JOIN）	JoinRowSet.CROSS_JOIN	supportsCrossJoin()

同时还有两个方法，`getJoinType()` 返回当前的联合方式，`setJoinType(int)` 设置联合方式。值得注意的是，Java 5 和 Java 6 中都支持内连接 (INNER JOIN) 这一种联合方式。在 setJoinType 方法中传入除 Inner_Join 以外的任何一种联合方式都会抛出 `UnsupportedOperationException` 的异常。

另外一点需要注意的，虽然默认的联合方式就是内连接，但是在没有显示的调用 `setJoinType()` 之前调用 `getJoinType()` 会抛出 `ArrayIndexOutOfBoundsException` 的异常。 所以一般来讲，我们可以不需要调用这几个方法而直接认为 JoinRowSet 默认并且只允许的联合方式就是内连接(INNER JOIN)。

如何联合各种 **RowSet**

联合多个 RowSet 的方法其实就是往一个 JoinRowSet 里调用 add 方法添加其他 RowSet（这个 RowSet 可以是上面提到的五种 RowSet 中的任意一种，包括离线操作的 JdbcRowSet 和 JoinRowSet 本身）的过程。添加一个 RowSet 的同时也必须制定联合时匹配的列。JoinRowSet 接口中提供了以下几个 add 方法：

1. addRowSet(Joinable rowset)
2. addRowSet(RowSet[] rowset, int[] columnIdx)
3. addRowSet(RowSet[] rowset, String[] columnName)
4. addRowSet(RowSet rowset, int columnIdx)
5. addRowSet(RowSet rowset, String columnName)

下面是一个联合一个 JdbcRowSet 和一个 CachedRowSet 的简单例子。

清单 **25.** 使用 **JoinRowSet**

```
// 构造一个CachedRowSet并且填充CUSTOMERS表中的数据。
CachedRowSet cachedRS = new CachedRowSetImpl();
cachedRS.setUrl(DBCreator.DERBY_URL);
cachedRS.setCommand(DBCreator.SQL_SELECT_CUSTOMERS);
cachedRS.execute();

// 构造一个JdbcRowSet并且填充ORDERS表中的数据。
JdbcRowSet jdbcRS = new JdbcRowSetImpl();
```

```
jdbcRS.setUrl(DBCreator.DERBY_URL);
jdbcRS.setCommand(DBCreator.SQL_SELECT_ORDERS);
jdbcRS.execute();

// 把cachedRS添加进新构造的JoinRowSet中。
JoinRowSet joinRS = new JoinRowSetImpl();
joinRS.addRowSet(cachedRS, "ID"); //$NON-NLS-1$

// 下面这条被注释的语句会抛出ClassCastExcepion。
// joinRS.addRowSet(jdbcRS, "ID");

// 把jdbcRS添加进这个JoinRowSet中。
jdbcRS.setMatchColumn("ID"); //$NON-NLS-1$
joinRS.addRowSet(jdbcRS);

// 观察结果
printRowSet(joinRS);
```

清单 **26.** 清单 **25** 中的代码执行结果

```
The data in CachedRowSet:
2 Jim null 1 Compute
1 Tom Tom is VIP. 1 Book
```

虽然 JoinRowSet 提供了五个不同的 addRowSet 方法，但是对于某些 RowSet，并不是每个方法都能用的。比如当上面添加一个 JdbcRowSet 的时候，你只能调用 `addRowSet(Joinable)` 这个方法（JdbcRowSet 实现了 Joinable 接口），并且在调用这个方法之前一定要先设置配对的列名或者以 1 为基数的配对列的位置，如下所示。

清单 **27.** 添加 **JdbcRowSet**

```
// 正确的添加一个JdbcRowSet的方法。
jdbcRS.setMatchColumn("ID"); //$NON-NLS-1$
joinRS.addRowSet(jdbcRS);
```

输出结果表明此时 JoinRowSet 中的数据正是两个 RowSet 中的数据的内联结果。加进去的两个 RowSet 中的”ID”这一列，在 JoinRowSet 中合并成了一列。如果我们查看最后 JoinRowSet 的列名，会发现这一列的列名变成了”MergedCol”, 其它列名不变。这个时候的 JoinRowSet 可以继续添加其他的 RowSet，联合会在 MergedCol 这一列上进行。

清单 **28.** 查看 **JoinRowSet** 中列名

```
for (int i = 1; i <= joinRS.getMetaData().getColumnCount(); i++) {
         System.out.println(joinRS.getMetaData().getColumnName(i));
}
```

清单**29.** 清单 **28** 中的代码执行结果

```
MergedCol
NAME
REMARK
USER_ID
PRODUCT
```

JoinRowSet 作为 **CachedRowSet** 的使用

从前面的继承结构图可以看到，JoinRowSet 继续于 CachedRowSet。因此从理论上来看，JoinRowSet 也可以作为 CachedRowSet 来使用。但是其实 JoinRowSet 很多方法并不是简单的直接继承 CachedRowSet 中的方法，而是重写（Override）了这些方法。最典型的两个方法是 execute 和 populate。 前面已经提到过，这是两种填充一个 CachedRowSet 的方法。但是对于 JoinRowSet，这两个方法却不起作用，不应该被使用。

另外一个比较特殊的方法是 acceptChanges。当 JoinRowSet 中包含一个 CachedRowSet 的时候，这个方法可以使用，并且效果就相当于在里面的这个 CachedRowSet 里面调用。但是当 JoinRowSet 包含两个或多个 RowSet 的时候，这个方法就不起作用了。这就类似于数据库多表联结后形成的视图(View)，一般是不能够进行更新操作的。

应注意的问题

1. 虽然 JoinRowSet 提供了五个不同的 addRowSet 方法，但是并不是对于每个 RowSet 这五个方法都是可行的。这点前面已经提到过。
2. 程序不能依赖 JoinRowSet 中的数据的顺序。JoinRowSet 接口中并没有能够控制连接结果排序的方法。它只能保证最后连接结果的正确性，但不能保证顺序。
3. 当把一个 RowSet 加入到 JoinRowSet 中，这个作为 addRowSet 方法的参数的 RowSet 的指针位置可能发生变化。另外当这个 RowSet 是 JdbcRowSet 的时候，在通过 addRowSet 方法加入之前，如果该 JdbcRowSet 的指针位置发生变化的时候，也会影响联合的结果。请看下面这个例子（假设我们已经像前面一样构造好了一个 CachedRowSet 和一个 JdbcRowSet）：

   清单 **30**

   ```
   // 添加cachedRS.
   JoinRowSet joinRS = new JoinRowSetImpl();
   joinRS.addRowSet(cachedRS, "ID"); //$NON-NLS-1$

   // 在添加jdbcRS之前，改变指针的位置。
   jdbcRS.next();
   jdbcRS.setMatchColumn("ID"); //$NON-NLS-1$
   ```

```
joinRS.addRowSet(jdbcRS);

// 观察结果
printRowSet(joinRS);
```

清单 **31.** 清单 **30** 中的代码执行结果

```
The data in CachedRowSet:
2 Jim null 1 Compute
```

由于 jdbcRS 在加入之前，指针位置发生了变化，导致联结后的结果不一样了。实际上，由于 jdbcRS 是保持数据库连接的，所以一般只能够进行查看下一条数据的操作，而不能查看已经前面的数据，所以当把一个 JdbcRowSet 加入到 JoinRowSet 中时，我们只相当于对这个 JdbcRowSet 从当前位置开始的数据进行了联结操作，而忽略了处于当前指针位置前面的数据。

那么，对于 CachedRowSet，情况又是怎么样呢？

清单 **32**

```
// 添加jdbcRS.
JoinRowSet joinRS = new JoinRowSetImpl();
jdbcRS.setMatchColumn("ID"); //$NON-NLS-1$
joinRS.addRowSet(jdbcRS);

// 在添加cachedRS之前，改变指针的位置。
cachedRS.last();
joinRS.addRowSet(cachedRS, "ID"); //$NON-NLS-1$

// 结果. 此时JoinRowSet中的数据。
printRowSet(joinRS);
```

清单 **33.** 清单 **32** 中的代码执行结果

```
The data in CachedRowSet:
```

```
2 1 Compute Jim null
1 1 Book Tom Tom is VIP.
```

由此可见，即使 cachedRS 在加入之前，指针位置发生了变化，也不会影响联结的结果。这是因为 CachedRowSet 是离线的，可以前后滚动查看数据。

回页首

结束语

本文介绍了 javax.sql.rowset 包下五个 RowSet 接口的使用，并重点说明了在使用中可能出现的问题。合理利用 RowSet 提供的离线式数据处理功能可以达到事半功倍的效果。

回页首

下载

描述	名字	大小	下载方法
本文用到的 Java 示例代码	RowSetDemo.zip	21 KB	**HTTP**

关于下载方法的信息

参考资料

学习

- Java SE 6 文档：Java SE 6 的规范文档。
- JDBC 3.0 文档：JDBC 3.0 的规范文档。
- WebRowSet Schema：序列化 WebRowSet 为 XML 格式时需要的 Schema 文件。
- "使用 Java 5 RowSet 新特性访问 IBM DB2 数据库"（developerWorks 中国）：介绍了 Java 5 中 RowSet 的使用。
- "利用 DB2 UDB 来使用 WebRowSet 实现"（developerWorks 中国）：介绍了在 DB2 UDB 中如何使用 WebRowSet。
- 浏览 技术书店 获取关于这些主题和其他技术主题的书籍。
- developerWorks Java 技术专区：查找数百篇有关 Java 编程各方面的文章。

讨论

- 参与 developerWorks blogs 并加入 developerWorks 社区。

作者简介

徐睿智，目前就职于 IBM 中国开发中心 Harmony 开发团队，对于分布式系统和并行计算比较感兴趣。

刘威，就读于重庆大学软件学院。曾在 IBM 中国开发中心 Harmony 团队实习。对 J2EE Web 开发比较感兴趣。

许彬，就读于复旦大学软件学院。曾在 IBM 中国开发中心 Harmony 团队实习。对 Java 和 Web 技术有浓厚的兴趣。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？