REX DB 1.0

目录

- **概述**
  - 简介
  - 功能
- **开发环境**
- **全局配置文件**
  - 加载配置文件
  - 外部资源文件
  - 全局设置
  - 数据源
  - 监听
- **执行数据库操作**
  - 插入/更新/删除
  - 批量更新
  - 查询多行记录
  - 查询单行记录
  - 调用
  - 事物
- **扩展**
  - 监听
  - 方言
  - 日志
  - 动态字节码
- **接口列表**
  - 类Configuration
  - 接口Dialect
  - 接口DBListener
  - 类Ps
  - 类RMap
  - 类DefaultDefinition

用户手册

概述

简介

Rexdb是一款使用Java语言编写的，开放源代码的持久层框架。提供了查询、调用、（JTA）事务、数据源管理等功能。使用Rexdb时，不需要像JDBC一样编写繁琐的代码，也不需要编写映射文件，只要将SQL和Java对象等参数传递至框架接口，即可获取需要的结果。

Rexdb的官方网站地址为：http://db.rex-soft.org

功能

Rexdb具有如下功能：

- 数据库操作：查询、更新、批处理、调用、（JTA）事物等；
- ORM映射：支持数组、Map和任意Java对象；
- 数据源：内置连接池，支持第三方数据源和JNDI；
- 方言：自动分页，支持Oracle、DB2、SQL Server、Mysql、达梦等数据库；
- 高级功能：监听、国际化、异常管理等；

开发环境

Rexdb的官方网站提供了下载衔接，下载并解压后，可以得到编译好的jar包和全局配置文件的示例：

- **rexdb-1.0.0.jar**（或其它版本）
- **rexdb.xml**

rexdb-1.0.0.jar（或其它版本）是运行Rexdb必须的包，请确保它在开发/运行环境的`classpath`中。由于Rexdb直接调用JDBC的接口，所以您还需要在`classpath`中设置好待连接数据库的驱动。如果要使用Rexdb的扩展功能，还需在运行环境中增加其它jar包。具体请参考扩展。

rexdb.xml是Rexdb的全局配置文件，默认需要放置在开发/运行环境的`classpath`中。如果需要放置在其它位置，需要编写程序加载指定位置的文件。具体请参考全局配置文件-加载配置文件。

例如，在JavaEE应用中，**rexdb-1.0.0.jar**（或其它版本）应当放置在应用根目录下的“`/WEB-INF/lib`”文件夹中，**rexdb.xml**默认应当放置在“`/WEB-INF/classes`”中。

全局配置文件

Rexdb需要一个XML格式的全局配置文件**rexdb.xml**，用于配置运行选项、数据源、监听程序等。各节点的含义如下：

- `/configuration`：根节点；
- `/configuration/properties`：外部资源文件，可以在该文件中定义`key-value`对，并在其余配置中以“`#{key}`”的格式引用`value`；
- `/configuration/settings`：全局设置选项，用于配置Rexdb的全局选项，例如异常信息语言、日志、反射缓存等；
- `/configuration/dataSource`：数据源配置；
- `/configuration/listener`：监听程序配置，可以使用监听程序跟踪SQL执行、事物等事件。

例如，某应用中的**rexdb.xml**文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//rex-soft.org//REXDB DTD 1.0//EN" "http://www.rex-soft.org/dtd/rexdb-1-config.dtd">
<configuration>
    <properties path="rexdb-settings.properties" />
    <settings>
        <property name="lang" value="#{setting.lang}"/>
        <property name="nolog" value="true" />
        <property name="reflectCache" value="true" />
        <property name="dynamicClass" value="true" />
    </settings>
    <dataSource>
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/rexdb" />
        <property name="username" value="root" />
        <property name="password" value="12345678" />
    </dataSource>
    <dataSource id="oracleDs" jndi="java:/comp/env/oracleDb"/>
    <listener class="org.rex.db.listener.impl.SqlDebugListener"/>
</configuration>
```

配置文件中引用了一个外部资源文件**rexdb-settings.properties**，并设置了异常信息语言、禁用了日志输出等全局选项；配置了一个默认数据源和一个`oracleDs`数据源；启用了Rexdb内置的`SqlDebugListener`监听。各节点的详细含义和配置方法请参见下面的章节。

加载配置文件

rexdb.xml的默认存放路径是运行环境的`classpath`根目录。Rexdb会在类加载时自动读取该文件，并完成框架的初始化工作。如果您启用了日志，将在日志输出中看到类似如下内容（输出格式取决于您的日志配置）：

```
[INFO][2016-02-23 21:26:55] configuration.Configuration[main] - loading default configuration rexdb.xml.
... # detailed log messages.
[INFO][2016-02-23 21:26:59] configuration.Configuration[main] - default configuration rexdb.xml loaded.
```

当在默认路径中无法找到**rexdb.xml**文件时，会输出如下日志：

```
[INFO][2016-02-23 22:18:36] configuration.Configuration[main] - loading default configuration rexdb.xml.
[WARN][2016-02-23 22:18:36] configuration.Configuration[main] - could not load default configuration rexdb.xml from classpath, rex
```

在配置未被加载时调用Rexdb的接口，Rexdb会再次尝试从默认路径中加载配置，如果仍然无法加载，将会抛出异常信息。如果Rexdb的全局配置文件放置在其它位置，或者使用了其它的命名，可以使用类org.rex.db.configuration.Configuration类加载指定目录下的配置文件。该类有如下加载配置文件的接口：

返回值	接口	说明
`void`	loadDefaultConfiguration()	从`classpath`中加载名称为**rexdb.xml**的配置文件
`void`	loadConfigurationFromClasspath(String path)	从`classpath`中加载配置文件
`void`	loadConfigurationFromFileSystem(String path)	从文件系统中加载配置文件

例如，下面的代码加载了位于classpath中的rexdb-config.xml文件：

```
    Configuration.loadConfigurationFromClasspath("rexdb-config.xml");
```

需要注意到是，Rexdb在加载配置文件时具备容错机制，当某节点不符合配置要求，或无法根据配置完成初始化时，该节点将会被忽略，并继续加载下一个节点。所以，您通常需要留意日志的输出，检查是否有未被成功加载的配置。

外部资源文件

全局配置文件的`/configuration/properties`节点用于引用一个外部资源文件，在该文件中定义的`key-value`配置可以被其它节点以`#{key}`的格式引用。该节点有如下可选属性：

属性	必填	类型	说明
`path`	否	`String`	本地`classpath`中资源文件的相对路径，不能与`url`属性同时存在。
`url`	否	`String`	网络中资源文件的`URL`路径，不能与`path`属性同时存在。

例如，放置在`classpath`根目录的资源文件**rexdb-database-sample.properties**内容如下：

```
    driver=com.mysql.jdbc.Driver
    url=jdbc:mysql://localhost:3306/rexdb
    username=root
    password=12345678
```

rexdb.xml中的配置如下：

```
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE configuration PUBLIC "-//rex-soft.org//REXDB DTD 1.0//EN" "http://www.rex-soft.org/dtd/rexdb-1-config.dtd">
    <configuration>
        <properties path="rexdb-database-sample.propertie" />
        <dataSource>
            <property name="driverClassName" value="#{driver}" />
            <property name="url" value="#{url}" />
            <property name="username" value="#{username}" />
            <property name="password" value="#{password}" />
        </dataSource>
    </configuration>
```

Rexdb在初始化时会首先读取**rexdb-database-sample.properties**文件的内容。在解析**rexdb.xml**时，如果发现其内容符合`#{...}`的格式，则会替换为资源文件中配置的值。在上面的示例中，`dataSource`节点的属性`driverClassName`的值是`#{driver}`，则会被替换为资源文件中`driver`对应的值`com.mysql.jdbc.Driver`。

全局设置

全局配置文件的`/configuration/settings`节点用于设置Rexdb的运行参数，可用的配置选项有：

配置项	必填	类型	可选值	默认值	说明
lang	否	String	en, zh-cn	en	设置Rexdb异常信息的语言。要注意的是，在Linux系统中，中文异常信息在输出至控制台可能会出现乱码。
nolog	否	boolean	true, false	false	是否禁用所有日志输出，当设置为true时，Rexdb将不再输出任何日志。
validateSql	否	boolean	true, false	false	是否对SQL语句进行简单的校验。通常Rexdb只校验SQL中带有?标记的个数是否与的预编译参数个数相同。
checkWarnings	否	boolean	true, false	false	在执行SQL后，是否检查状态中的警告。当设置为true时，将执行检查，当发现警告信息时抛出异常。要注意的是，开启该选项可能会大幅降低Rexdb的性能。
queryTimeout	否	int	任意整数	-1	执行SQL的超时秒数，当小于或等于0时，不设置超时时间。当同时设置了事物超时时间时，Rexdb会自动选择一个较短的时间作为执行SQL的超时秒数。
transactionTimeout	否	int	任意整数	-1	执行事务的超时秒数，当小于或等于0时，不设置超时时间。要注意的是，Rexdb通过设置事物中每个SQL的执行时间来控制整体事物的时间，如果事物中有与SQL执行无关的操作，且在执行该操作时超时，事物超时时间将不起作用。
transactionIsolation	否	String	DEFAULT READ_UNCOMMITTED READ_COMMITTED REPEATABLE_READ SERIALIZABLE	DEFAULT	定义事物的隔离级别，仅在非JTA事物中时有效。各参数含义如下： - DEFAULT：使用数据库默认的事务隔离级别； - READ_UNCOMMITTED：一个事务可以看到其它事务未提交的数据 - READ_COMMITTED：一个事务修改的数据提交后才能被另外一个事务读取； - REPEATABLE_READ：同一事务的多个实例在并发读取数据时，会看到同样的数据行； - SERIALIZABLE：通过强制事务排序，使事物之间不可能相互冲突。
autoRollback	否	boolean	true, false	false	事务提交失败时是否自动回滚。
reflectCache	否	boolean	true, false	true	是否启用反射缓存。当启用时，Rexdb将会缓存类的参数、函数等信息。开启该选项可以大幅提升Rexdb的性能。
dynamicClass	否	boolean	true, false	true	是否启用动态字节码功能。当开启该选项时，Rexdb将使用javassist的生成中间类。启用该选项可以大幅提高Rexdb在查询Java对象时的性能。要注意的是，该选项需要配合jboss javassist包使用，Rexdb会在加载全局配置时检测javassist环境，当环境不可用时，该配置项会被自动切换为false。
dateAdjust	否	boolean	true, false	true	写入数据时，是否自动将日期类型的参数转换为java.sql.Timestamp类型。开启此选项可以有效避免日期、时间数据的丢失，以及因类型、格式不匹配而产生的异常。

batchTransaction	否	boolean	true, false	true	调用批量更新接口时，如果当前没有事物，是否自动开启。在某些数据库中，需要在在事物中执行批量更新，才能获得高效的性能。

例如，如果要设置Rexdb抛出异常时的语言为中文，并且禁用日志，可以使用如下配置：

```
    <settings>
        <property name="lang" value="en"/>
        <property name="nolog" value="false" />
    </settings>
```

要注意的是，如果设置项不被Rexdb支持，或者值的格式、值域不正确，则该设置会被忽略并使用默认值。

数据源

/configuration/dataSource节点用于配置数据源。该节点支持如下属性：

属性	必填	类型	说明
id	否	String	数据源编号。不设置时为Rexdb的默认数据源，配置文件中只允许出现一个默认数据源。
class	否	String	数据源实现类，不设置时使用Rexdb的内置数据源，不能与jndi属性一同出现。
jndi	否	String	上下文中的数据源JNDI，不能与class属性一同出现。
dialect	否	String	为该数据源指定的数据库方言，不设置时将由Rexdb根据元数据信息自动选择内置的方言，请参见[方言接口](#class-dialect)。

也可以通过property节点为数据源初始化参数。当不设置class和jndi属性时，Rexdb将使用内置的数据源org.rex.db.datasource.SimpleDataSource。该数据源支持如下初始化参数：

选项	必填	类型	可选值	默认值	说明
driverClassName	是	String	-	-	JDBC驱动类。
url	是	String	-	-	数据库连接URL。
username	是	String	-	-	数据库用户。
password	是	String	-	-	数据库密码。
initSize	否	int	大于0的整数	1	初始化连接池时创建的连接数。
minSize	否	int	大于0的整数	3	连接池保持的最小连接数。连接池将定期检查持有的连接数，达不到该数量时将开启新的空闲连接。
maxSize	否	int	大于0的整数	10	连接池的最大连接数。当程序所需连接超出此数量时，将置于等待状态，直到有新的空闲连接。
increment	否	int	大于0的整数	1	每次增长的连接数。当连接池的连接数量不足，需要开启新的连接时，将一次性增长该参数指定的连接数。
retries	否	int	大于0的整数	2	获取新的数据库连接失败后的重试次数。Rexdb不会判定失败原因，只要无法创建新的连接，即重试指定的次数。
retryInterval	否	int	大于0的整数	750	创建新的数据库连接失败后的重试间隔，单位为毫秒。即当获取一个新的数据库连接失败，直到下一次重试的等待时间.
					获取连接的超时时间，单位为毫秒。当程序从

`getConnectionTimeout`	否	`int`	大于0的整数	`5000`	Rexdb数据源中申请一个新的连接，且当前无空闲连接时，程序的等待时间。当超过改时间后，Rexdb会抛出一个超时的异常信息。
`inactiveTimeout`	否	`int`	大于0的整数	`600000`	数据库连接的最长空闲时间，单位为毫秒。当数据库连接的空闲时间超过该参数的值时，连接会被关闭。
`maxLifetime`	否	`int`	大于0的整数	`1800000`	数据库连接的最长时间，单位为毫秒。当数据库连接开启时间超过该参数的值时，连接会被关闭。
`testConnection`	否	`boolean`	`true`, `false`	`true`	开启新的数据库连接后，是否测试连接可用。当运行环境为JDK1.6及以上版本时，Rexdb将使用JDBC的测试接口执行测试；当JDK低于1.5时，如果未指定测试SQL，将调用方言接口获取测试SQL，如果能成功执行，则测试通过。
`testSql`	否	`String`	SQL语句	-	指定测试连接是否活跃的SQL语句。
`testTimeout`	否	`int`	大于0的整数	`500`	测试连接的超时时间。

类似于Rexdb内置的数据源，其它开源数据源（例如Apache DBCP、C3P0等），通常也支持设置多个初始化参数，具体请参考各自的用户手册。

例如，以下代码配置了3个数据源：

```
    <dataSource>
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:orcl" />
        <property name="username" value="test" />
        <property name="password" value="123456" />
    </dataSource>
    <dataSource id="mysqlDs" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/rexdb" />
        <property name="username" value="root" />
        <property name="password" value="12345678" /
    </dataSource>
    <dataSource id="oracleDs" jndi="java:/comp/env/oracleDb"/>
```

按照顺序分别是：

- 连接Oracle数据库的默认数据源，使用Rexdb自带的数据源和连接池；
- 连接Mysql数据库的数据源，编号为“mysqlDs”，使用了Apache DBCP数据源；
- 连接Oracle的数据源，编号为“oracleDs”，使用JNDI方式查找容器自带的数据源；

上面的示例中，默认数据源使用了Rexdb自带的数据源，如果希望将其配置为初始化连接数为3、每次增长3个连接、重试次数设置为3、不再测试连接活跃性时，可以调整为如下配置：

```
    <dataSource>
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:orcl" />
        <property name="username" value="test" />
        <property name="password" value="123456" />

        <property name="initSize" value="3"/>
        <property name="increment" value="3"/>
        <property name="retries" value="3"/>
        <property name="testConnection" value="false"/>
    </dataSource>
```

在配置好数据源后，在调用类`org.rex.DB`的方法执行SQL、处理事务时，可以指定数据源。例如，在执行查询时：

```
    DB.getMap("SELECT * FROM REX_TEST");            //使用默认数据源执行查询
    DB.getMap("mysqlDs", "SELECT * FROM REX_TEST"); //使用mysqlDs数据源执行查询
```

```
DB.getMap("oracleDs", "SELECT * FROM REX_TEST");//使用oracleDs数据源执行查询
```

监听

`/configuration/listener`节点用于设置监听程序。监听程序可以跟踪SQL执行、事物等事件，该节点支持如下属性：

属性	必填	类型	说明
`class`	否	`String`	监听程序实现类。

如果监听类定义了可以设置的属性，还可以通过设置`property`节点设置值。Rexdb内置了用于输出SQL和事物信息的监听类，分别是：

- `org.rex.db.listener.impl.SqlDebugListener`：使用日志接口输出SQL和事物信息。该监听类支持如下配置选项：

选项	必填	类型	可选值	默认值	说明
`level`	否	`String`	`debug`, `info`	`debug`	设置日志的输出级别。
`simple`	否	`boolean`	`true`, `false`	`false`	是否启用简易的日志输出。当设置为`true`时，仅在SQL或事物完成后输出简要的日志；设置为`false`时，在SQL和事物执行前后均会输出日志。

- `org.rex.db.listener.impl.SqlConsolePrinterListener`：将SQL和事物信息输出到`System.out`终端。该监听类支持如下配置选项：

选项	必填	类型	可选值	默认值	说明
`simple`	否	`boolean`	`true`, `false`	`false`	是否启用简易的日志输出，当设置为`true`时，仅在SQL或事物完成后输出日志；设置为`false`时，在SQL和事物执行前后均会输出日志。

例如，以下代码配置了一个监听：

```
<listener class="org.rex.db.listener.impl.SqlDebugListener">
    <property name="simple" value="true"/>
</listener>
```

上面的配置使用了Rexdb内置的`SqlDebugListener`监听类，并以DEBUG级别输出简要的日志信息。如果您需要自行定义监听程序，例如记录每个SQL的执行时间，可以编写程序实现监听接口，详情请查看扩展-监听。

需要注意的是，监听程序并非线程安全，且不运行于独立线程，在编程时需要注意线程安全和性能问题。

执行数据库操作

定义好全局配置文件后，就可以执行数据库操作了。Rexdb将数据库操作接口集中在类`org.rex.DB`中，且都是静态的，可以直接调用。根据SQL类型的不同，可以将接口分类如下：

- 插入/更新/删除操作：`DB.update(...)`系列接口；
- 批量更新：`DB.batchUpdate(...)`系列接口；
- 查询多行记录：`DB.getList(...)`和`DB.getMapList(...)`系列接口；
- 查询单行记录：`DB.get(...)`和`DB.getMap(...)`系列接口；
- 调用：`DB.call(...)`系列接口；
- 事物：`DB.beginTransaction(...)`, `DB.commit(...)`, `DB.rollback(...)`等接口
- 其它：`getDataSource(...)`, `getConnection(...)`等接口

如果您在开发时使用了Eclipse等IDE工具，可以方便的由工具提示出可用的接口列表，直接选择需要的接口使用即可。

插入/更新/删除

类`org.rex.DB`的下列接口负责执行数据库的插入/更新/删除操作，以及执行创建表、删除表等DDL SQL：

使用默认数据源

返回值	接口	说明
int	update(String sql)	执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。
int	update(String sql, Object[] parameterArray)	执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以?标记预编译参数，Object数组中的元素按照顺序与其对应。
int	update(String sql, Ps ps)	执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以?标记预编译参数，Ps对象内置的元素按照顺序与其对应。
int	update(String sql, Map	执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以#{key}的格式标记预编译参数，Map对象中键为key的值与其对应。当Map对象中没有键key时，将赋值为null。
int	update(String sql, Object parameterBean)	执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以#{key}的格式标记预编译参数，Rexdb将在Object对象中查找key对应的getter方法，通过该方法取值后作为相应的预编译参数。当Object对象中没有相应的getter方法时，将赋值为null。

使用指定的数据源

返回值	接口	说明
int	update(String dataSourceId, String sql)	在指定的数据源中执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。
int	update(String dataSourceId, String sql, Object[] parameterArray)	在指定的数据源中执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以?标记预编译参数，Object数组中的元素按照顺序与其对应。
int	update(String dataSourceId, String sql, Ps ps)	在指定的数据源中执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以?标记预编译参数，Ps对象内置的元素按照顺序与其对应。
int	update(String dataSourceId, String sql, Map	在指定的数据源中执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以#{key}的格式标记预编译参数，Map对象中键为key的值与其对应。当Map对象中没有键key时，将赋值为null。
int	update(String dataSourceId, String sql, Object parameterBean)	在指定的数据源中执行一个SQL语句，例如INSERT、UPDATE、DELETE或DDL语句。SQL语句需要以#{key}的格式标记预编译参数，Rexdb将在Object对象中查找key对应的getter方法，通过该方法取值后作为相应的预编译参数。当Object对象中没有相应的getter方法时，将赋值为null。

在执行带有预编译参数的SQL时，数组、org.rex.db.Ps、Map和Java对象都可以作为预编译参数。

当使用数组做参数时，SQL语句以?作为预编译参数标记，数组元素按照顺序与其对应。Rexdb还内置了类org.rex.db.Ps，提供了比数组更加丰富的操作接口，可以按照下标赋值，还可以声明输出参数等，详情请参见类org.rex.db.Ps。Ps对象中内置的元素同样按照顺序与SQL语句中的?标记对应。

Rexdb支持java.util.Map作为执行SQL的参数。此时，SQL语句中的预编译参数需要声明为#{key}的格式，Map中键为key的值将作为对应的预编译参数，当Map中没有键key时，预编译参数将被设置为null。

Rexdb还支持Java类作为预编译参数，与Map类似，SQL语句中的预编译参数需要声明为#{key}的格式，Rexdb将通过调用getter方法获取key的值，并将其作为预编译参数。当无法取值时，预编译参数将设置为null。需要注意的是，实体类还需要满足如下条件，才能被Rexdb正常调用：

- 类是可以访问的；

- 参数需要有标准的getter方法；
- 类具备无参的构造函数（启用动态字节码选项时需要调用）

以下是使用各种类型的参数执行SQL的示例：

以下代码直接执行了一个没有预编译参数的SQL：

```
DB.update("INSERT INTO REX_TEST(ID, NAME, CREATE_TIME) VALUES (1, 'Jim', now())"); //Mysql
```

当使用`数组`作为执行SQL的参数时，可以使用如下代码：

```
    String sql = "INSERT INTO REX_TEST(ID, NAME, CREATE_TIME) VALUES (?, ?, ?)";
    int i = DB.update(sql, new Object[]{1, "test", new Date()});
```

与数组类似，当使用`Ps`对象作为参数时：

```
String sql = "INSERT INTO REX_TEST(ID, NAME, CREATE_TIME) VALUES (?, ?, ?)";
int i = DB.update(sql, new Ps(1, "test", new Date()));
```

当使用`Map`对象做参数时，SQL语句中需要以`#{key}`的格式标记预编译参数，例如：

```
String sql = "INSERT INTO REX_TEST(ID, NAME, CREATE_TIME) VALUES (#{id}, #{name}, #{createTime})";
Map prameters = new HashMap();
prameters.put("id", 1);
prameters.put("name", "test");
prameters.put("createTime", new Date());

int i = DB.update(sql, prameters);
```

使用自定义的`Java对象`做参数时，首先需要编写一个成员变量能够与表的字段对应的类：

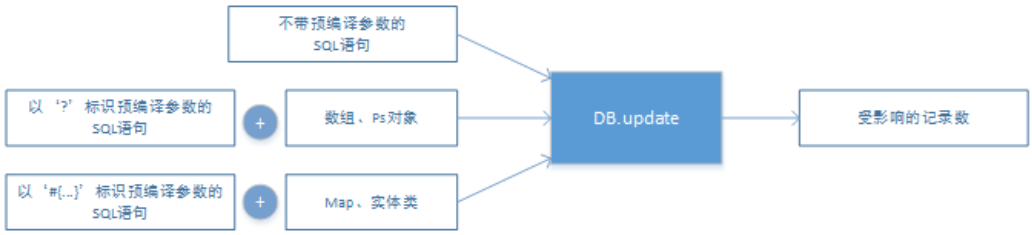
```
import java.util.Date;

public class RexTest {

    private int id;
    private String name;
    private Date createTime;

    public RexTest() {
    }

    public RexTest(int id, String name, Date createTime) {
        this.id = id;
        this.name = name;
        this.createTime = createTime;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getCreateTime() {
```

```
        return createTime;
    }

    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }
}
```

该类的实例可以作为执行SQL的参数：

```
String sql = "INSERT INTO REX_TEST(ID, NAME, CREATE_TIME) VALUES (#{id}, #{name}, #{createTime})";
RexTest rexTest = new RexTest(1, "test", new Date());

int i = DB.update(sql, rexTest);
```

下图展示了`DB.update(...)`接口中SQL语句和各种类型参数的组合方式：



批量更新

当插入多条记录时，使用批量更新接口可以获得更好的执行效率。类`org.rex.DB`中的批量更新接口如下：

> 使用默认数据源

返回值	接口	说明
`int[]`	batchUpdate(String[] sql)	将一批SQL提交至数据库执行，如果全部成功，则返回更新计数组成的数组。
`int[]`	batchUpdate(String sql, Object[][] parameterArrays)	将一组`java.lang.Object数组`作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`int[]`	batchUpdate(String sql, Ps[] parameters)	将一组`org.rex.db.Ps对象`作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`int[]`	batchUpdate(String sql, Map[] parameterMaps)	将一组`java.util.Map`作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。
`int[]`	batchUpdate(String sql, Object[] parameterBeans)	将一组`Object`对象作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句需要以`#{key}`的格式标记预编译参数，`Object`对象中的属性名称与其对应。
`int[]`	batchUpdate(String sql, List parameterList)	将一个`java.util.List`对象作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。`List`中的元素类型必须相同，Rexdb将根据类型确定SQL中预编译参数标记方式和取值方式。

> 使用指定的数据源

返回值	接口	说明
`int[]`	batchUpdate(String dataSourceId, String[] sql)	在指定数据源中执行一批SQL语句，如果全部成功，则返回更新计数组成的数组。

int[]	batchUpdate(String dataSourceId, String sql, Object[][] parameterArrays)	在指定数据源中将一组java.lang.Object数组作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句以?标记预编译参数，Object数组中的元素按照顺序与其对应。
int[]	batchUpdate(String dataSourceId, String sql, Ps[] parameters)	将一组org.rex.db.Ps对象作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句以?标记预编译参数，Ps对象内置的元素按照顺序与其对应。
int[]	batchUpdate(String dataSourceId, String sql, Map[] parameterMaps)	在指定数据源中将一组java.util.Map作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句需要以#{key}的格式标记预编译参数，Map对象中键为key的值与其对应。
int[]	batchUpdate(String dataSourceId, String sql, Object[] parameterBeans)	在指定数据源中将一组Object对象作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。SQL语句需要以#{key}的格式标记预编译参数，Object对象中的属性名称与其对应。
int[]	batchUpdate(String dataSourceId, String sql, List parameterList)	在指定数据源中将一个java.util.List对象作为参数提交至数据库执行，如果全部成功，则返回更新计数组成的数组。List中的元素类型必须相同，Rexdb将根据类型确定SQL中预编译参数标记方式和取值方式。

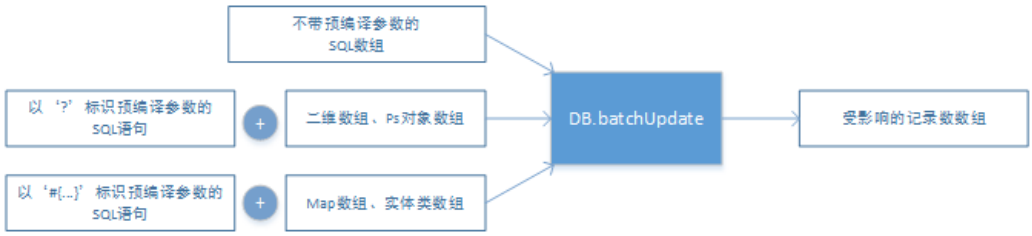
在使用批量更新接口时，需要预先准备好多个SQL或参数。当需要写入大量记录时，可以将考虑拆分成多份后多次调用批量更新接口，以减少内存占用。

以org.rex.db.Ps数组做参数为例，可以使用如下代码执行批量更新：

```
String sql = "INSERT INTO REX_TEST(ID, NAME, CREATE_TIME) VALUES (?, ?, ?)";
Ps[] pss = new Ps[10];
for (int i = 0; i < 10; i++)
    pss[i] = new Ps(i, "name", new Date());
DB.batchUpdate(sql, pss);
```

执行成功后，数据库将写入10条记录。

下图展示了DB.batchUpdate(...)系列接口的SQL语句和参数组合方式：



查询多行记录

类org.rex.DB中的getList(...)系列接口用于查询多条记录。返回值是一个java.util.ArrayList列表，列表中的元素为调用接口时指定类型的Java对象，每个元素对应一条数据库记录。如果没有找到符合条件的记录，将返回一个空的ArrayList。

如果没有编写结果集对应的Java对象，也可以使用getMapList(...)系列方法查询一个包含java.util.Map的列表。列表中的元素类型为org.rex.RMap，是java.util.HashMap的子类，该类的具体接口请查阅类org.rex.RMap。

Rexdb在进行O/R映射时，会读取结果集中元数据，并将标签（Label）名称转换为Java风格的命名（具体的转换规则为“分析小写的标签名称，将字符_后的首字母转换为大写后，再移除字符_”），再根据转换后的名称为Java对象或Map赋值。例如：

```
列名          ->  Map的key/Java对象的属性名称
ABC          ->  abc
ABC_DE       ->  abcDe
ABC_DE_F     ->  abcDeF
```

Rexdb内置了数据库方言，在查询指定偏移量和条目的记录时（以下称为分页查询），会根据方言自动封装相应的SQL语句，详情请见扩展-方言和接口org.rex.db.dialect.Dialect。

- 如果希望查询指定类型的`Java对象`，可以使用如下接口：

使用默认数据源

返回值	接口	说明
`List<T>`	get(String sql, Class<T> resultClass)	执行查询，获取元素类型为resultClass的Java对象列表。
`List<T>`	get(String sql, Object[] parameterArray, Class<T> resultClass)	执行查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<T>`	get(String sql, Ps parameters, Class<T> resultClass)	执行查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<T>`	get(String sql, Map<?, ?> parameters, Class<T> resultClass)	执行查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`List<T>`	get(String sql, Object parameters, Class<T> resultClass)	执行查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。
`List<T>`	getList(String sql, Class<T> resultClass, int offset, int rows)	执行分页查询，获取元素类型为resultClass的Java对象列表。
`List<T>`	getList(String sql, Object[] parameterArray, Class<T> resultClass, int offset, int rows)	执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<T>`	getList(String sql, Ps parameters, Class<T> resultClass, int offset, int rows)	执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<T>`	getList(String sql, Map<?, ?> parameters, Class<T> resultClass, int offset, int rows)	执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`List<T>`	getList(String sql, Object parameters, Class<T> resultClass, int offset, int rows)	执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。

使用指定的数据源

返回值	接口	说明
`List<T>`	get(String dataSourceId, String sql, Class<T> resultClass)	在指定数据源中执行查询，获取元素类型为resultClass的Java对象列表。
`List<T>`	get(String dataSourceId, String sql, Object[] parameterArray, Class<T> resultClass)	在指定数据源中执行查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<T>`	get(String dataSourceId, String sql, Ps parameters, Class<T> resultClass)	在指定数据源中执行查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<T>`	get(String dataSourceId, String sql, Map<?, ?> parameters, Class<T> resultClass)	在指定数据源中执行查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
		在指定数据源中执行查询，获取元素类型为resultClass的Java对象列表。

`List<T>`	get(String dataSourceId, String sql, Object parameters, Class<T> resultClass)	SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。
`List<T>`	getList(String dataSourceId, String sql, Class<T> resultClass, int offset, int rows)	在指定数据源中执行分页查询，获取元素类型为resultClass的Java对象列表。
`List<T>`	getList(String dataSourceId, String sql, Object[] parameterArray, Class<T> resultClass, int offset, int rows)	在指定数据源中执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<T>`	getList(String dataSourceId, String sql, Ps parameters, Class<T> resultClass, int offset, int rows)	在指定数据源中执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<T>`	getList(String dataSourceId, String sql, Map<?, ?> parameters, Class<T> resultClass, int offset, int rows)	在指定数据源中执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`List<T>`	getList(String dataSourceId, String sql, Object parameters, Class<T> resultClass, int offset, int rows)	在指定数据源中执行分页查询，获取元素类型为resultClass的Java对象列表。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。

- 如果希望查询出元素为`java.util.Map`的列表，可以使用下列接口：

> 使用默认数据源

返回值	接口	说明
`List<RMap>`	getMapList(String sql)	执行查询，获取元素为`Map`的列表。
`List<RMap>`	getMapList(String sql, Object[] parameterArray)	执行查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<RMap>`	getMapList(String sql, Ps parameters)	执行查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<RMap>`	getMapList(String sql, Map<?, ?> parameters)	执行查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`List<RMap>`	getMapList(String sql, Object parameters)	执行查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。
`List<RMap>`	getMapList(String sql, int offset, int rows)	执行分页查询，获取元素为`Map`的列表。
`List<RMap>`	getMapList(String sql, Object[] parameterArray, int offset, int rows)	执行分页查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<RMap>`	getMapList(String sql, Ps parameters, int offset, int rows)	执行分页查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<RMap>`	getMapList(String sql, Map<?, ?> parameters, int offset, int rows)	执行分页查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`List<RMap>`	getMapList(String sql, Object parameters, int offset, int rows)	执行分页查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的

		getter方法时，将赋值为`null`。

使用指定的数据源

返回值	接口	说明
`List<RMap>`	getMapList(String dataSourceId, String sql)	在指定数据源中执行查询，获取元素为`Map`的列表。
`List<RMap>`	getMapList(String dataSourceId, String sql, Object[] parameterArray)	在指定数据源中执行查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<RMap>`	getMapList(String dataSourceId, String sql, Ps parameters)	在指定数据源中执行查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<RMap>`	getMapList(String dataSourceId, String sql, Map<?, ?> parameters)	在指定数据源中执行查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`List<RMap>`	getMapList(String dataSourceId, String sql, Object parameters)	在指定数据源中执行查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。
`List<RMap>`	getMapList(String dataSourceId, String sql, int offset, int rows)	在指定数据源中执行分页查询，获取元素为`Map`的列表。
`List<RMap>`	getMapList(String dataSourceId, String sql, Object[] parameterArray, int offset, int rows)	在指定数据源中执行分页查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`List<RMap>`	getMapList(String dataSourceId, String sql, Ps parameters, int offset, int rows)	在指定数据源中执行分页查询，获取元素为`Map`的列表。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`List<RMap>`	getMapList(String dataSourceId, String sql, Map<?, ?> parameters, int offset, int rows)	在指定数据源中执行分页查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`List<RMap>`	getMapList(String dataSourceId, String sql, Object parameters, int offset, int rows)	在指定数据源中执行分页查询，获取元素为`Map`的列表。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。

例如，如下代码以多种方式查询了默认数据源中的表`REX_TEST`（在实际使用时，选择一种方式即可）。为便于演示，根据参数类型的不同进行了代码分组：

- 无预编译参数时

```
String sql = "SELECT * FROM REX_TEST";
List<RMap> list = DB.getMapList(sql);                    //查询包含Map对象的列表
List<RMap> list = DB.getMapList(sql, 0, 10);             //查询前10条记录，获取包含Map对象的列表
List<RexTest> list = DB.getList(sql, RexTest.class);     //查询包含RexTest的列表
List<RexTest> list = DB.getList(sql, RexTest.class, 0, 10); //查询前10条记录，包含RexTest的列表
```

- 以数组作参数时

```
String sqlo = "SELECT * FROM REX_TEST WHERE ID > ?";
Object[] obj = new Object[]{10};
List<RMap> list = DB.getMapList(sqlo, obj);                    //查询编号大于10，包含Map对象的列表
List<RMap> list = DB.getMapList(sqlo, obj, 0, 10);             //查询编号大于10的前10条记录，获取包含Map对象的列表
List<RexTest> list = DB.getList(sqlo, obj, RexTest.class);     //查询编号大于10的记录，包含RexTest的列表
List<RexTest> list = DB.getList(sqlo, obj, RexTest.class, 0, 10);//查询编号大于10的前10条记录，包含RexTest的列表
```

- 以Ps对象作参数时

```
String sqlp = "SELECT * FROM REX_TEST WHERE ID > ?";
Ps ps = new Ps(10);
List<RMap> list = DB.getMapList(sqlp, ps);                    //查询编号大于10，包含Map对象的列表
List<RMap> list = DB.getMapList(sqlp, ps, 0, 10);             //查询编号大于10的前10条记录，获取包含Map对象的列表
List<RexTest> list = DB.getList(sqlp, ps, RexTest.class);     //查询编号大于10的记录，包含RexTest的列表
List<RexTest> list = DB.getList(sqlp, ps, RexTest.class, 0, 10);//查询编号大于10的前10条记录，包含RexTest的列表
```

- 以Map对象作参数时

```
String sqlm = "SELECT * FROM REX_TEST WHERE ID > #{id}";
Map map = new HashMap();
ps.put("id", 10);
List<RMap> list = DB.getMapList(sqlm, map);                   //查询编号大于10，包含Map对象的列表
List<RMap> list = DB.getMapList(sqlm, map, 0, 10);            //查询编号大于10的前10条记录，获取包含Map对象的列表
List<RexTest> list = DB.getList(sqlm, map, RexTest.class);    //查询编号大于10的记录，包含RexTest的列表
List<RexTest> list = DB.getList(sqlm, map, RexTest.class, 0, 10);//查询编号大于10的前10条记录，包含RexTest的列表
```

- 以RexTest对象作参数时

```
String sqlj = "SELECT * FROM REX_TEST WHERE ID > #{id}";
RexTest rexTest = new RexTest();
rexTest.setId(10);
List<RMap> list = DB.getMapList(sqlj, rexTest);                   //查询编号大于10，包含Map对象的列表
List<RMap> list = DB.getMapList(sqlj, rexTest, 0, 10);            //查询编号大于10的前10条记录，获取包含Map对象的列表
List<RexTest> list = DB.getList(sqlj, rexTest, RexTest.class);    //查询编号大于10的记录，包含RexTest的列表
List<RexTest> list = DB.getList(sqlj, rexTest, RexTest.class, 0, 10);//查询编号大于10的前10条记录，包含RexTest的列表
```

`DB.getMapList(...)`接口的SQL语句和参数组合如下：

`DB.getList(...)`接口的SQL语句和参数组合如下：

查询单行记录

与查询多行记录类似，类`org.rex.DB`的`get(...)`和`getMap(...)`方法分别用于查询指定类型的`Java对象`和`Map`对象。要注意的是，如果未查询到记录，查询接口将返回`null`；如果查询出了多条记录，由于无法确定需要哪一条，因此会抛出异常。

- 如果希望查询出指定类型的`Java对象`，可以使用下面的接口：

> 使用默认数据源

返回值	接口	说明
`T`	get(String sql, Class<T> resultClass)	执行查询，获取指定类型的`Java对象`。
`T`	get(String sql, Object[] parameterArray, Class<T> resultClass)	执行查询，获取指定类型的`Java对象`。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。

T	get(String sql, Ps parameters, Class<T> resultClass)	执行查询，获取指定类型的Java对象。SQL语句需要以?标记预编译参数，Ps对象内置的元素按照顺序与其对应。
T	get(String sql, Map<?, ?> parameters, Class<T> resultClass)	执行查询，获取指定类型的Java对象。SQL语句需要以#{key}的格式标记预编译参数，Map对象中键为key的值与其对应。当Map对象中没有键key时，将赋值为null。
T	get(String sql, Object parameters, Class<T> resultClass)	执行查询，获取指定类型的Java对象。SQL语句需要以#{key}的格式标记预编译参数，Rexdb将在Object对象中查找key对应的getter方法，通过该方法取值后作为相应的预编译参数。当Object对象中没有相应的getter方法时，将赋值为null。

使用指定的数据源

返回值	接口	说明
T	get(String dataSourceId, String sql, Class<T> resultClass)	在指定数据源中执行查询，获取指定类型的Java对象。
T	get(String dataSourceId, String sql, Object[] parameterArray, Class<T> resultClass)	在指定数据源中执行查询，获取指定类型的Java对象。SQL语句需要以?标记预编译参数，Object数组中的元素按照顺序与其对应。
T	get(String dataSourceId, String sql, Ps parameters, Class<T> resultClass)	在指定数据源中执行查询，获取指定类型的Java对象。SQL语句需要以?标记预编译参数，Ps对象内置的元素按照顺序与其对应。
T	get(String dataSourceId, String sql, Map<?, ?> parameters, Class<T> resultClass)	在指定数据源中执行查询，获取指定类型的Java对象。SQL语句需要以#{key}的格式标记预编译参数，Map对象中键为key的值与其对应。当Map对象中没有键key时，将赋值为null。
T	get(String dataSourceId, String sql, Object parameters, Class<T> resultClass)	在指定数据源中执行查询，获取指定类型的Java对象。SQL语句需要以#{key}的格式标记预编译参数，Rexdb将在Object对象中查找key对应的getter方法，通过该方法取值后作为相应的预编译参数。当Object对象中没有相应的getter方法时，将赋值为null。

- 如果希望查询出Map类型的结果时象，可以使用下面的接口：

使用默认数据源

返回值	接口	说明
RMap	getMap(String sql)	执行查询，获取Map类型的结果。
RMap	getMap(String sql, Object[] parameterArray)	执行查询，获取Map类型的结果。SQL语句需要以?标记预编译参数，Object数组中的元素按照顺序与其对应。
RMap	getMap(String sql, Ps parameters)	执行查询，获取Map类型的结果。SQL语句需要以?标记预编译参数，Ps对象内置的元素按照顺序与其对应。
RMap	getMap(String sql, Map parameters)	执行查询，获取Map类型的结果。SQL语句需要以#{key}的格式标记预编译参数，Map对象中键为key的值与其对应。当Map对象中没有键key时，将赋值为null。
RMap	getMap(String sql, Object parameters)	执行查询，获取Map类型的结果。SQL语句需要以#{key}的格式标记预编译参数，Rexdb将在Object对象中查找key对应的getter方法，通过该方法取值后作为相应的预编译参数。当Object对象中没有相应的getter方法时，将赋值为null。

使用指定的数据源

返回值	接口	说明

`RMap`	getMap(String dataSourceId, String sql)	在指定数据源中执行查询，获取`Map`类型的结果。
`RMap`	getMap(String dataSourceId, String sql, Object[] parameterArray)	在指定数据源中执行查询，获取`Map`类型的结果。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`RMap`	getMap(String dataSourceId, String sql, Ps parameters)	在指定数据源中执行查询，获取`Map`类型的结果。SQL语句需要以`?`标记预编译参数，`Ps`对象内置的元素按照顺序与其对应。
`RMap`	getMap(String dataSourceId, String sql, Map parameters)	在指定数据源中执行查询，获取`Map`类型的结果。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`RMap`	getMap(String dataSourceId, String sql, Object parameters)	在指定数据源中执行查询，获取`Map`类型的结果。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。

例如，以下代码使用了多种方式查询了表`REX_TEST`中的一条记录：

- 无预编译参数时

```
String sql = "SELECT * FROM REX_TEST WHERE ID = 10";
RMap rMap = DB.getMap(sql);                    //查询Map对象
RexTest rexTest = DB.get(sql, RexTest.class);   //查询RexTest对象
```

- 以数组作参数时

```
String sqlo = "SELECT * FROM REX_TEST WHERE ID = ?";
Object[] obj = new Object[]{10};
RMap rMap = DB.getMap(sqlo, obj);                   //查询编号为10的Map对象
RexTest rexTest = DB.get(sqlo, obj, RexTest.class); //查询编号为10的RexTest对象
```

- 以Ps对象作参数时

```
String sqlp = "SELECT * FROM REX_TEST WHERE ID = ?";
Ps ps = new Ps(10);
RMap rMap = DB.getMap(sqlp, ps);                    //查询编号为10的Map对象
RexTest rexTest = DB.get(sqlp, ps, RexTest.class);  //查询编号为10的RexTest对象
```

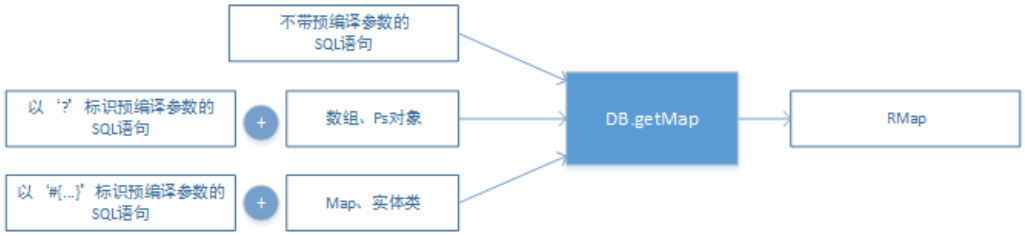
- 以Map对象作参数时

```
String sqlm = "SELECT * FROM REX_TEST WHERE ID = #{id}";
Map map = new HashMap();
ps.put("id", 10);
RMap rMap = DB.getMap(sqlm, map);                   //查询编号为10的Map对象
RexTest rexTest = DB.get(sqlm, map, RexTest.class); //查询编号为10的RexTest对象
```

- 以RexTest对象作参数时

```
String sqlj = "SELECT * FROM REX_TEST WHERE ID = #{id}";
RexTest rexTest = new RexTest();
rexTest.setId(10);
RMap rMap = DB.getMap(sqlj, rexTest);                   //查询编号为10的Map对象
RexTest rexTest = DB.get(sqlj, rexTest, RexTest.class); //查询编号为10的RexTest对象
```

`DB.getMap(...)`接口的SQL语句和参数组合如下：



`DB.get(...)`接口的SQL语句和参数组合如下：

调用

类`org.rex.DB`的`call(...)`系列方法用于执行存储过程和函数调用，接口列表如下：

> 使用默认数据源

返回值	接口	说明
`RMap`	call(String sql)	执行调用，并获取返回结果（如果有）。
`RMap`	call(String sql, Object[] parameterArray)	执行调用，并获取返回结果（如果有）。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`RMap`	call(String sql, Map parameterMap)	执行调用，并获取返回结果（如果有）。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`RMap`	call(String sql, Object parameterBean)	执行调用，并获取返回结果（如果有）。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。
`RMap`	call(String sql, Ps ps)	执行调用，并获取输出参数和返回结果（如果有）。SQL语句需要以`?`标记预编译参数、声明输出参数和输入输出参数，`Ps`对象内置的元素按照顺序与其对应。

> 使用指定的数据源

返回值	接口	说明
`RMap`	call(String dataSourceId, String sql)	在指定数据源中执行调用，并获取返回结果（如果有）。
`RMap`	call(String dataSourceId, String sql, Object[] parameterArray)	在指定数据源中执行调用，并获取返回结果（如果有）。SQL语句需要以`?`标记预编译参数，`Object数组`中的元素按照顺序与其对应。
`RMap`	call(String dataSourceId, String sql, Map parameterMap)	在指定数据源中执行调用，并获取返回结果（如果有）。SQL语句需要以`#{key}`的格式标记预编译参数，`Map`对象中键为`key`的值与其对应。当`Map`对象中没有键`key`时，将赋值为`null`。
`RMap`	call(String dataSourceId, String sql, Object parameterBean)	在指定数据源中执行调用，并获取返回结果（如果有）。SQL语句需要以`#{key}`的格式标记预编译参数，Rexdb将在`Object`对象中查找`key`对应的getter方法，通过该方法取值后作为相应的预编译参数。当`Object`对象中没有相应的getter方法时，将赋值为`null`。

`RMap`	call(String dataSourceId, String sql, Ps ps)	在指定数据源中执行调用，并获取输出参数和返回结果（如果有）。SQL语句需要以`?`标记预编译参数、声明输出参数和输入输出参数，`Ps`对象内置的元素按照顺序与其对应。

当调用有返回值时，Rexdb会自动遍历，并按照顺序存放在接口返回的`RMap`对象中，键分别为"`return_0`"、"`return_1`"等。当使用`org.rex.db.Ps`对象作为参数时，可以使用其`addReturnType(Class beanClass)`和`setReturnType(index, Class beanClass)`方法为每个返回值声明Java类型。如果声明了返回值类型`T`，`RMap`对象中的相应的结果将是`List<T>`；当不声明返回值类型时，每个返回结果都将是`List<RMap>`。

当调用有输出参数时，需要使用`org.rex.db.Ps`对象作为调用的参数，并使用其`addOut(...)`和`setOut(...)`系列方法声明输出参数，或者使用`addInOut(...)`和`setInOut(...)`系列方法声明输入输出参数。调用成功后，可以在返回的`RMap`对象中获取输出参数的值，键分别为"`out_0`"、"`out_1`"等。此外，`Ps`对象还支持对输出参数设置别名，在设置了别名后，返回的`RMap`对象中还可以按照别名取值。`org.rex.db.Ps`的接口详情请参见类org.rex.db.Ps。

例如，以下代码调用了存储过程`test_proc`，并获取了第1个返回值：

```
RMap result = DB.call("{call test_proc()}");
List<RMap> return1 = (List<RMap>)result.get("return_0");
```

例如，以下代码声明了1个`int`类型的输出参数，调用成功后可以在返回的`RMap`对象中以`out_0`的键取值：

```
Ps ps = new Ps();
ps.addOutInt();                                 //将第1个参数声明为int类型的输出参数
RMap result = DB.call("{call proc_out(?)}", ps);    //调用存储过程
int out = result.getInt("out_0");               //获取输出参数的值
```

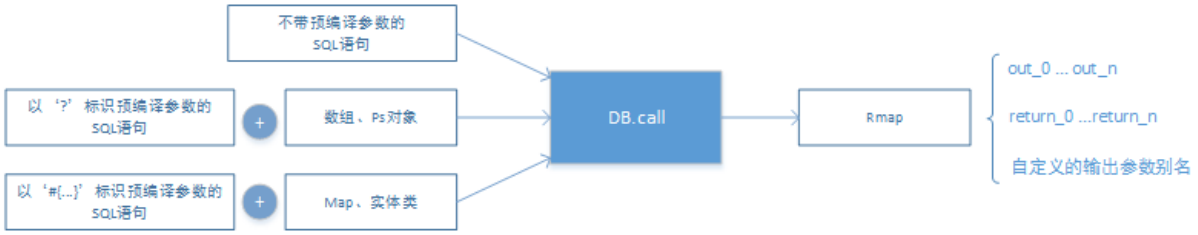
为取值方便，也可以在上面的代码中为输出参数设置一个别名，例如：

```
Ps ps = new Ps().addOutInt("age");              //将第1个参数声明为int类型的输出参数，并设置别名"age"
RMap result = DB.call("{call proc_out(?)}", ps);    //调用存储过程
int out = result.getInt("age");                 //使用别名获取输出参数的值
```

与输出参数类似，`Ps`对象还可以声明输入输出参数。例如：

```
Ps ps = new Ps().addInOut(1);                   //将第1个参数声明为输入输出参数
RMap result = DB.call("{call proc_inout(?)}", ps);  //调用存储过程
int out = result.getInt("out_0");               //获取输出参数的值
```

DB.call(...)接口的SQL语句和参数组合如下：



事物

Rexdb支持事物和标准的JTA事物，类`org.rex.DB`的事物接口有：

> 使用默认数据源的事物

返回值	接口	说明
`void`	beginTransaction()	开启事物。
`void`	beginTransaction(DefaultDefinition definition)	开启事物并设置事物参数。
`void`	commit()	提交事物。

`void`	rollback()	回滚事物
`java.sql.Connection`	getTransactionConnection()	获取事物所在的数据库连接。

> 使用指定的数据源的事物

返回值	接口	说明
`void`	beginTransaction(String dataSourceId)	在指定数据源中开启事物。
`void`	beginTransaction(String dataSourceId, DefaultDefinition definition)	在指定数据源中开启事物，并设置配置参数。
`void`	commit(String dataSourceId)	提交指定数据源的事物。
`void`	rollback(String dataSourceId)	回滚指定数据源事物
`java.sql.Connection`	getTransactionConnection(String dataSourceId)	获取指定数据源事物所在的连接。

> 分布式事物

返回值	接口	说明
`void`	beginJta()	开启JTA事物。
`void`	commitJta()	提交JTA事物。
`void`	rollbackJta()	回滚JTA事物。

在启用事物前，可以为事物设置超时时间、隔离级别等。首先实例化一个`org.rex.db.transaction.DefaultDefinition`对象，并在调用开启事物方法时将其作为参数，详情请见类DefaultDefinition。

需要注意的是，在使用Rexdb事物接口时，需要遵循`try...catch...`的写法，以防事物开启后未被提交或回滚。

例如，以下代码启用了事物：

```
DB.beginTransaction();
try{
    DB.update("DELETE FROM REX_TEST");
    DB.update("INSERT INTO REX_TEST(ID, NAME, CREATE_TIME) VALUES (?, ?, ?)", new Ps(1, "test", new Date()));
    DB.commit();
}catch(Exception e){//一般来说，应捕获Exception异常，以防程序抛出预期外的异常，导致事物未被回滚
    DB.rollback();
}
```

如果要设置事物的超时时间和隔离级别，可以使用如下代码：

```
DefaultDefinition definition = new DefaultDefinition();
definition.setTimeout(10);                                          //设置事物超时时间为10秒
definition.setIsolationLevel(DefaultDefinition.ISOLATION_READ_COMMITTED);   //设置事物的隔离级别为"READ_COMMITTED"
DB.beginTransaction(definition);
```

扩展

监听

可以通过配置监听程序，实现SQL、事物执行事件的捕获。Rexdb已经内置了以下监听类（详情请查看全局配置文件-监听）：

监听类	说明

`org.rex.db.listener.impl.SqlDebugListener`	使用日志包输出SQL和事物执行信息。
`org.rex.db.listener.impl.SqlConsolePrinterListener`	将SQL和事物执行信息输出到终端。

当需要实现一个新的监听时，首先编写监听程序，实现接口org.rex.db.listener.DBListener。例如，如果希望打印出执行时间超过10秒的所有SQL语句，则可以编写如下监听类：

```
package test;

import org.rex.db.listener.DBListener;
import org.rex.db.listener.SqlContext;
import org.rex.db.listener.TransactionContext;

public class CustomListener implements DBListener{
    public void onExecute(SqlContext context) {
    }

    public void afterExecute(SqlContext context, Object results) {
        long costs = System.currentTimeMillis() - context.getCreateTime().getTime();
        if(costs > 10000){                                  //当执行时间超过10秒时
            String[] sql = context.getSql();                //获取已经执行的SQL
            Object parameters = context.getParameters();    //获取预编译参数
            System.out.println("more than 10s: " + sql[0] + " : " + parameters);
        }
    }

    public void onTransaction(TransactionContext context) {
    }
    public void afterTransaction(TransactionContext context) {
    }
}
```

然后将将该监听类加入到全局配置文件即可：

```
<listener class="test.CustomListener" />
```

方言

Rexdb支持数据库方言功能，用于支持自动的分页查询等功能。已经内置的方言有：

数据库	方言实现类
DB2	`org.rex.db.dialect.impl.DB2Dialect`
Derby	`org.rex.db.dialect.impl.DerbyDialect`
DM	`org.rex.db.dialect.impl.DMDialect`
H2	`org.rex.db.dialect.impl.H2Dialect`
HSQL	`org.rex.db.dialect.impl.HSQLDialect`
MySQL	`org.rex.db.dialect.impl.MySQLDialect`
Oracle	`org.rex.db.dialect.impl.Oracle8iDialect` `org.rex.db.dialect.impl.Oracle9iDialect`
PostgreSQL	`org.rex.db.dialect.impl.PostgreSQLDialect`
SQLServer	`org.rex.db.dialect.impl.SQLServerDialect` `org.rex.db.dialect.impl.SQLServer2005Dialect`

Rexdb会根据数据库的类型和版本选择合适的方言。如果您使用的数据库不在列表中，可以编写一个实现接口org.rex.db.dialect.Dialect的方言类，并将其增加到全局配置文件-数据源中。

日志

Rexdb支持如下日志包：

顺序号	日志	官方网址
1	log4j-1.x	http://logging.apache.org/log4j
2	slf4j	http://www.slf4j.org/
3	log4j-2.x	http://logging.apache.org/log4j
4	JDK Logger	-

当Rexdb在加载类时，会按照上面的顺序检测日志支持环境，并使用首个可用的日志接口。如果希望禁用日志，可以在全局配置文件-全局设置中将`nolog`属性设置为`true`。

动态字节码

Rexdb支持jboss javassist（官方网址：http://jboss-javassist.github.io/javassist/）的动态字节码功能。当全局配置文件-全局设置中的`dynamicClass`属性为`true`，且检测到javassist环境可用时，Rexdb框架将会启动动态字节码功能。

启用动态字节码后，在查询指定类型的`Java对象`系列接口时将有大幅的性能提升，因此建议开启此扩展功能。

另外，我们注意到，在javassist的官方网站下载的编译包均是基于新版的JDK编译。因此，如果您的JDK运行环境较低，建议下载javassist的源代码，并使用低版本JDK重新编译。同时，我们在Rexdb的下载包中也内置了一个基于JDK1.5编译的新版javassist，您可以根据实际情况选用。

接口列表

类org.rex.db.configuration.Configuration

该类用于加载Rexdb的全局配置文件，有如下接口：

返回值	接口	说明
`void`	loadDefaultConfiguration()	从`classpath`中加载名为**rexdb.xml**的配置文件
`void`	loadConfigurationFromClasspath(String path)	从`classpath`中加载配置文件
`void`	loadConfigurationFromFileSystem(String path)	从文件系统中加载配置文件

Rexdb只允许加载一次配置文件。配置加载后，不能重新加载，也不能加载其它位置的配置。

需要注意的是，在类加载器加载`org.rex.db.configuration.Configuration`时，会自动调用`loadDefaultConfiguration()`方法加载默认配置文件**rexdb.xml**，当该文件不存在时，才能调用接口加载其它位置的配置。

接口org.rex.db.dialect.Dialect

该接口用于定义数据库的方言，而数据库方言类用于定义数据库个性化的语句，例如分页查询SQL、测试SQL等。Rexdb在执行分页查询、测试活跃连接等操作时调用数据库方言接口。接口定义如下：

返回值	接口	说明
`LimitHandler`	getLimitHandler(int rows)	获取一个分页查询SQL封装类，用于包装带有行数限制的查询。
`LimitHandler`	getLimitHandler(int offset, int rows)	获取一个分页查询SQL封装类，用于包装带有偏移数和行数限制的查询。
`String`	getTestSql()	获取测试SQL语句，通常用于测试数据库连接的有效性。
`String`	getName()	获取数据库名称。例如，oracle数据库方言将返回`ORACLE`。

其中，抽象类`org.rex.db.dialect.LimitHandler`用于封装分页查询语句。抽象接口接口定义如下：

返回值	接口	说明
`String`	wrapSql(String sql)	包装分页查询SQL，分页相关的预编译参数必须设置在其它预编译参数后。
`void`	afterSetParameters(PreparedStatement statement, int parameterCount)	设置分页相关的预编译参数，这个方法将在设置完其它预编译参数后调用。

接口org.rex.db.listener.DBListener

该接口用于定义一个数据库监听类，可以用于监听SQL执行、事物的启用、提交等事件。接口如下：

返回值	接口	说明
`void`	onExecute(SqlContext context)	SQL执行前调用该方法。
`void`	afterExecute(SqlContext context, Object results)	SQL执行后调用该方法。
`void`	onTransaction(TransactionContext context)	开始事物前调用该方法。
`void`	afterTransaction(TransactionContext context)	提交、回滚事物后调用该方法。

其中，类`org.rex.db.listener.SqlContext`中存放了与SQL执行相关的参数，常量和接口如下：

> 类`org.rex.db.listener.SqlContext`的常量

类型	常量名称	说明
`int`	SQL_QUERY	表示当前执行的是查询。
`int`	SQL_UPDATE	表示当前执行的是更新。
`int`	SQL_BATCH_UPDATE	表示当前执行的是批处理。
`int`	SQL_CALL	表示当前执行的是调用。

> 类`org.rex.db.listener.SqlContext`的方法

返回值	接口	说明
`String`	getContextId()	获取唯一的SQL上下文编号。这个参数可以用于区分属于一次执行的多个事件。
`Date`	getCreateTime()	获取当前上下文实例的创建时间。
`int`	getSqlType()	获取执行类型。返回值是常量`SQL_QUERY`、`SQL_UPDATE`、`SQL_BATCH_UPDATE`、`SQL_CALL`中的一种。
`boolean`	isBetweenTransaction()	当前数据库操作是否在事物中。
`DataSource`	getDataSource()	获取当前操作所在的数据源。
`String[]`	getSql()	获取SQL。当执行一条SQL语句时，返回的数组元素个数为1。
`Object`	getParameters()	获取执行SQL的参数（不包括分页查询的偏移、行数参数）。
`LimitHandler`	getLimitHandler()	获取当前查询的分页对象。如果当前执行的不是分页查询，返回`null`。

类`org.rex.db.listener.TransactionContext`中包含了与事物相关的参数，常量和方法如下：

> 类`org.rex.db.listener.TransactionContext`的常量

类型	常量名称	说明
`int`	TRANSACTION_BEGIN	当前事件为开始事务。
`int`	TRANSACTION_COMMIT	当前事件为提交事物。
`int`	TRANSACTION_ROLLBACK	当前事件为回滚事物。

> 类`org.rex.db.listener.TransactionContext`的方法

返回值	接口	说明
`String`	getContextId()	获取唯一的上下文编号。这个参数可以用于区分属于一次事物执行的多个事件。
`Date`	getCreateTime()	获取当前上下文实例的创建时间。
`Definition`	getDefinition()	获取事务设置。仅在启用事务时有值，提交、回滚事务时该方法将返回`null`。
`int`	getEvent()	获取事务事件类型。值为常量`TRANSACTION_BEGIN`、`TRANSACTION_COMMIT`、`TRANSACTION_ROLLBACK`中的一种。

类org.rex.db.Ps

类`org.rex.db.Ps`用于封装预编译参数，它可以设置执行SQL时的输入、输出和输入输出参数。与`数组`相比，提供了更多实用的接口，大致可分为如下几类：

- 类中定义的常量
- 构造函数
- 设置预编译输入参数
- 声明输出参数
- 声明输入输出参数
- 声明返回值映射类
- 其它方法

- 该类中定义的常量有：

类型	常量名称	说明
`String`	CALL_OUT_DEFAULT_PREFIX	`DB.call(...)`系列接口的返回值中，输出参数的前缀。
`String`	CALL_RETURN_DEFAULT_PREFIX	`DB.call(...)`系列接口的返回值中，返回值的前缀。

- 构造函数有：

构造函数	说明
`public Ps()`	初始化一个空的Ps对象。
`public Ps(Object... parameters)`	初始化一个Ps对象，并按照参数顺序设置预编译参数。

- 设置预编译输入参数

> 按顺序设置预编译参数

返回值	接口	说明
Ps	add(Object value)	增加一个预编译参数,SQL类型将根据value的Java类型自动匹配。
Ps	add(int index, Object value, int sqlType)	增加一个预编译参数，并指定SQL类型。
Ps	addNull()	增加一个值为null的预编译参数。
Ps	add(String value)	增加一个String类型的预编译参数。
Ps	add(boolean value)	增加一个boolean类型的预编译参数。
Ps	add(BigDecimal value)	增加一个BigDecimal类型的预编译参数。
Ps	add(int value)	增加一个int类型的预编译参数。
Ps	add(long value)	增加一个long类型的预编译参数。
Ps	add(double value)	增加一个double类型的预编译参数。
Ps	add(float value)	增加一个float类型的预编译参数。
Ps	add(Blob value)	增加一个Blob类型的预编译参数。
Ps	add(Clob value)	增加一个Clob类型的预编译参数。
Ps	dd(java.util.Date date)	增加一个java.util.Date类型的预编译参数。
Ps	add(java.sql.Date date)	增加一个java.sql.Date类型的预编译参数。
Ps	add(Time time)	增加一个Time类型的预编译参数。
Ps	add(Timestamp time)	增加一个Timestamp类型的预编译参数。

> 设置指定索引的预编译参数

返回值	接口	说明
Ps	set(int index, Object value)	在指定索引设置预编译参数。index起始于1，预编译参数的SQL类型将根据value的Java类型自动匹配。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, Object value, int sqlType)	在指定索引设置预编译参数，并指定SQL类型。index起始于1。index索引不能大于当前已经设置的预编译参数个数。
Ps	setNull(int index)	在指定索引设置一个值为null的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, String value)	在指定索引设置一个String类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, boolean value)	在指定索引设置一个boolean类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, BigDecimal value)	在指定索引设置一个BigDecimal类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, int value)	在指定索引设置一个int类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, long value)	在指定索引设置一个long类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, double value)	在指定索引设置一个double类型的预编译参数。index索引不能大于当前

Ps	set(int index, double value)	已经设置的预编译参数个数。
Ps	set(int index, float value)	在指定索引设置一个float类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, Blob value)	在指定索引设置一个Blob类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, Clob value)	在指定索引设置一个Clob类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, java.util.Date date)	在指定索引设置一个java.util.Date类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, java.sql.Date date)	在指定索引设置一个java.sql.Date类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, Time time)	在指定索引设置一个Time类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	set(int index, Timestamp time)	在指定索引设置一个Timestamp类型的预编译参数。index索引不能大于当前已经设置的预编译参数个数。

- 声明输出参数

> 按顺序声明输出参数

返回值	接口	说明
Ps	addOutResultSet(int sqlType)	声明一个结果集类型的输出参数，参数sqlType需要设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为Map对象。
Ps	addOutResultSet(int sqlType, Class resultClass)	声明一个结果集类型的输出参数，参数sqlType需要设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为指定类型的Java对象。
Ps	addOutString()	声明一个String类型的输出参数。
Ps	addOutBoolean()	声明一个boolean类型的输出参数。
Ps	addOutBigDecimal()	声明一个BigDecimal类型的输出参数。
Ps	addOutInt()	声明一个int类型的输出参数。
Ps	addOutLong()	声明一个long类型的输出参数。
Ps	addOutFloat()	声明一个float类型的输出参数。
Ps	addOutDouble()	声明一个double类型的输出参数。
Ps	addOutBlob()	声明一个Blob类型的输出参数。
Ps	addOutClob()	声明一个Clob类型的输出参数。
Ps	addOutDate()	声明一个Date类型的输出参数。
Ps	addOutTime()	声明一个Time类型的输出参数。
Ps	addOutTimestamp()	声明一个Timestamp类型的输出参数。

> 按顺序声明输出参数，并为参数设置别名

返回值	接口	说明
		声明一个结果集类型的输出参数，并为参数设置别名。参数sqlType需要

Ps	addOutResultSet(String paramName, int sqlType)	设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为Map对象
Ps	addOutResultSet(String paramName, int sqlType, Class resultClass)	声明一个结果集类型的输出参数，并为参数设置别名。参数sqlType需要设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为指定类型的Java对象。
Ps	addOutString(String paramName)	声明一个String类型的输出参数，并为参数设置别名。
Ps	addOutBoolean(String paramName)	声明一个boolean类型的输出参数，并为参数设置别名。
Ps	addOutBigDecimal(String paramName)	声明一个BigDecimal类型的输出参数，并为参数设置别名。
Ps	addOutInt(String paramName)	声明一个int类型的输出参数，并为参数设置别名。
Ps	addOutLong(String paramName)	声明一个long类型的输出参数，并为参数设置别名。
Ps	addOutFloat(String paramName)	声明一个float类型的输出参数，并为参数设置别名。
Ps	addOutDouble(String paramName)	声明一个double类型的输出参数，并为参数设置别名。
Ps	addOutBlob(String paramName)	声明一个Blob类型的输出参数，并为参数设置别名。
Ps	addOutClob(String paramName)	声明一个Clob类型的输出参数，并为参数设置别名。
Ps	addOutDate(String paramName)	声明一个Date类型的输出参数，并为参数设置别名。
Ps	addOutTime(String paramName)	声明一个Time类型的输出参数，并为参数设置别名。
Ps	addOutTimestamp(String paramName)	声明一个Timestamp类型的输出参数，并为参数设置别名。

> 声明指定索引的输出参数

返回值	接口	说明
Ps	setOutResultSet(int index, int sqlType)	在指定索引声明一个结果集类型的输出参数，参数sqlType需要设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为Map对象。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutResultSet(int index, int sqlType, Class resultClass)	在指定索引声明一个结果集类型的输出参数，参数sqlType需要设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为指定类型的Java对象。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutString(int index)	在指定索引声明一个String类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutBoolean(int index)	在指定索引声明一个boolean类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutBigDecimal(int index)	在指定索引声明一个BigDecimal类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutInt(int index)	在指定索引声明一个int类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutLong(int index)	在指定索引声明一个long类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutFloat(int index)	在指定索引声明一个float类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutDouble(int index)	在指定索引声明一个double类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。

Ps	setOutBlob(int index)	在指定索引声明一个Blob类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutClob(int index)	在指定索引声明一个Clob类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutDate(int index)	在指定索引声明一个Date类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutTime(int index)	在指定索引声明一个Time类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutTimestamp(int index)	在指定索引声明一个Timestamp类型的输出参数。index索引不能大于当前已经设置的预编译参数个数。

> 声明指定索引的输出参数，并为参数设置别名

返回值	接口	说明
Ps	setOutResultSet(int index, String paramName, int sqlType)	在指定索引声明一个结果集类型的输出参数，并为参数设置别名，参数sqlType需要设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为Map对象。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutResultSet(int index, String paramName, int sqlType, Class resultClass)	在指定索引声明一个结果集类型的输出参数，并为参数设置别名，参数sqlType需要设置为JDBC驱动中定义的结果集类型。执行调用后，结果集将被映射为指定类型的Java对象。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutString(int index, String paramName)	在指定索引声明一个String类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutBoolean(int index, String paramName)	在指定索引声明一个boolean类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutBigDecimal(int index, String paramName)	在指定索引声明一个BigDecimal类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutInt(int index, String paramName)	在指定索引声明一个int类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutLong(int index, String paramName)	在指定索引声明一个long类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutFloat(int index, String paramName)	在指定索引声明一个float类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutDouble(int index, String paramName)	在指定索引声明一个double类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutBlob(int index, String paramName)	在指定索引声明一个Blob类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutClob(int index, String paramName)	在指定索引声明一个Clob类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutDate(int index, String paramName)	在指定索引声明一个Date类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setOutTime(int index, String paramName)	在指定索引声明一个Time类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。

Ps	setOutTimestamp(int index, String paramName)	在指定索引声明一个Timestamp类型的输出参数，并为参数设置别名。index索引不能大于当前已经设置的预编译参数个数。

- 声明输入输出参数

> 按顺序声明输入输出参数

返回值	接口	说明
Ps	addInOut(Object value)	声明一个输入输出参数。预编译参数的SQL类型将根据value的Java类型自动匹配。
Ps	addInOut(Object value, int type)	声明一个输入输出参数，并指定SQL类型。
Ps	addInOutNull()	声明一个null类型的输入输出参数。
Ps	addInOut(String value)	声明一个String类型的输入输出参数。
Ps	addInOut(boolean value)	声明一个boolean类型的输入输出参数。
Ps	addInOut(BigDecimal value)	声明一个BigDecimal类型的输入输出参数。
Ps	addInOut(int value)	声明一个int类型的输入输出参数。
Ps	addInOut(long value)	声明一个long类型的输入输出参数。
Ps	addInOut(float value)	声明一个float类型的输入输出参数。
Ps	addInOut(double value)	声明一个double类型的输入输出参数。
Ps	addInOut(Blob value)	声明一个Blob类型的输入输出参数。
Ps	addInOut(Clob value)	声明一个Clob类型的输入输出参数。
Ps	addInOut(java.util.Date date)	声明一个java.util.Date类型的输入输出参数。
Ps	addInOut(java.sql.Date date)	声明一个java.sql.Date类型的输入输出参数。
Ps	addInOut(Time time)	声明一个Time类型的输入输出参数。
Ps	addInOut(Timestamp time)	声明一个Timestamp类型的输入输出参数。

> 按顺序声明输入输出参数，并设置别名

返回值	接口	说明
Ps	addInOut(String paramName, Object value)	声明一个输入输出参数，并设置别名。预编译参数的SQL类型将根据value的Java类型自动匹配。
Ps	addInOut(String paramName, Object value, int type)	声明一个输入输出参数，并设置别名和SQL类型。
Ps	addInOutNull(String paramName)	声明一个null类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, String value)	声明一个String类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, boolean value)	声明一个boolean类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, BigDecimal value)	声明一个BigDecimal类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, int value)	声明一个int类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, long value)	声明一个long类型的输入输出参数，并设置别名。

Ps	addInOut(String paramName, float value)	声明一个float类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, double value)	声明一个double类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, Blob value)	声明一个Blob类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, Clob value)	声明一个Clob类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, java.util.Date date)	声明一个java.util.Date类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, java.sql.Date date)	声明一个java.sql.Date类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, Time time)	声明一个Time类型的输入输出参数，并设置别名。
Ps	addInOut(String paramName, Timestamp time)	声明一个Timestamp类型的输入输出参数，并设置别名。

声明指定索引的输入输出参数

返回值	接口	说明
Ps	setInOut(int index, Object value)	在指定索引声明一个输入输出参数，预编译参数的SQL类型将根据value的Java类型自动匹配。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, Object value, int type)	在指定索引声明一个输入输出参数，并指定SQL类型。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOutNull(int index)	在指定索引声明一个null类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String value)	在指定索引声明一个String类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, boolean value)	在指定索引声明一个boolean类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, BigDecimal value)	在指定索引声明一个BigDecimal类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, int value)	在指定索引声明一个int类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, long value)	在指定索引声明一个long类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, float value)	在指定索引声明一个float类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, double value)	在指定索引声明一个double类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, Blob value)	在指定索引声明一个Blob类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, Clob value)	在指定索引声明一个Clob类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, java.util.Date date)	在指定索引声明一个java.util.Date类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
		在指定索引声明一个java.sql.Date类型的输入输出参数。index索引不

Ps	setInOut(int index, java.sql.Date date)	在指定索引声明一个java.sql.Date类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, Time time)	在指定索引声明一个Time类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, Timestamp time)	在指定索引声明一个Timestamp类型的输入输出参数。index索引不能大于当前已经设置的预编译参数个数。

> 声明指定索引的输入输出参数，并设置别名

返回值	接口	说明
Ps	setInOut(int index, String paramName, Object value)	在指定索引声明一个输入输出参数并设置别名，预编译参数的SQL类型将根据value的Java类型自动匹配。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, Object value, int type)	在指定索引声明一个输入输出参数，，并设置别名和SQL类型。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOutNull(int index, String paramName)	在指定索引声明一个null类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, String value)	在指定索引声明一个String类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, boolean value)	在指定索引声明一个boolean类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, BigDecimal value)	在指定索引声明一个BigDecimal类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, int value)	在指定索引声明一个int类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, long value)	在指定索引声明一个long类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, float value)	在指定索引声明一个float类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, double value)	在指定索引声明一个double类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, Blob value)	在指定索引声明一个Blob类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, Clob value)	在指定索引声明一个Clob类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, java.util.Date date)	在指定索引声明一个java.util.Date类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, java.sql.Date date)	在指定索引声明一个java.sql.Date类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, Time time)	在指定索引声明一个Time类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。
Ps	setInOut(int index, String paramName, Timestamp time)	在指定索引声明一个Timestamp类型的输入输出参数，并设置别名。index索引不能大于当前已经设置的预编译参数个数。

- 声明返回值映射类

返回值	接口	说明
`Ps`	addReturnType(Class<?> beanClass)	设置调用操作返回值的映射类。当调用的函数或存储过程有1个或多个返回值时，可以使用该方法按顺序设置一个返回值的映射类。
`Ps`	setReturnType(int index, Class<?> resultBeanClass)	按索引设置调用操作返回值的映射类。当调用的函数或存储过程有1个或多个返回值时，可以使用该方法设置一个返回值的映射类。索引`index`开始于1.

- 其它方法

返回值	接口	说明
`int`	getParameterSize()	获取已经设置的预编译参数个数。
`List<SqlParameter>`	getParameters()	获取已经设置或声明的预编译参数，包括输入、输出和输入输出参数。
`List<Class<?>>`	getReturnResultTypes()	获取已经设置的调用操作的返回值映射类。

类org.rex.RMap

类`org.rex.RMap`是`java.util.HashMap`的子类，额外提供了获取指定Java类型值的接口，更便于开发人员使用。
类`org.rex.DB`的`getMap(...)`、`getMapList(...)`等方法均返回该类型的对象。接口如下：

返回值	接口	说明
`String`	getString(String key, boolean emptyAsNull)	获取`String`类型的值。当`emptyAsNull`参数设置为`true`，且Map条目中的实际值为""时，会返回一个`null`值。
`String`	getString(String key)	获取`String`类型的值。当`Map`条目中原有数据类型不是`String`时，将进行格式转换；原条目为`数组`时，将返回类似于`[值1，值2，值3]`这样的格式。
`boolean`	getBoolean(String key)	获取`boolean`类型的值。
`int`	getInt(String key)	获取`int`类型的值。当`Map`条目中的值无法转换为`int`时，将抛出`NumberFormatException`异常。
`long`	getLong(String key)	获取`long`类型的值。当`Map`条目中的值无法转换为`long`时，将抛出`NumberFormatException`异常。
`float`	getFloat(String key)	获取`float`类型的值。当`Map`条目中的值无法转换为`float`时，将抛出`NumberFormatException`异常。
`double`	getDouble(String key)	获取`double`类型的值。当`Map`条目中的值无法转换为`double`时，将抛出`NumberFormatException`异常。
`BigDecimal`	getBigDecimal(String key)	获取`BigDecimal`类型的值。当`Map`条目中的值无法转换为`BigDecimal`时，将抛出`NumberFormatException`异常。
`java.util.Date`	getDate(String key)	获取`java.util.Date`类型的值。如果`Map`条目中的值不是`Date`类型或其子类时，将按照常见的日期格式进行格式化。如果无法匹配格式，将抛出异常。
`java.sql.Date`	getDateForSql(String key)	获取`java.sql.Date`类型的值。如果`Map`条目中的值不是`Date`类型或其子类时，将按照常见的日期格式进行格式化。如果无法匹配格式，将抛出异常。
		获取`java.sql.Time`类型的值。如果`Map`条目中的值不是`Date`类型或其

`Time`	getTime(String key)	子类时，将按照常见的日期格式进行格式化。如果无法匹配格式，将抛出异常。
`Timestamp`	getTimestamp(String key)	获取`java.sql.Timestamp`类型的值。如果`Map`条目中的值不是`Date`类型或其子类时，将按照常见的日期格式进行格式化。如果无法匹配格式，将抛出异常。
`String[]`	getStringArray(String key)	获取`String[]`类型的值。当`Map`条目中的值不是`String[]`类型时，将进行类型转换。
`Object`	set(K key, V value)	设置一个值，等同于`Map.put(K key, V value)`
`void`	setAll(Map m)	合并`Map`对象，等同于`Map.putAll(Map m)`
`void`	addDateFormat(String pattern, String format)	新增一个日期格式识别表达式。

在获取日期类型的值时，Rexdb可以自动识别日期格式的字符串，如果识别成功，将自动进行类型转换。支持的日期格式如下：

日期格式	示例
`yyyy-MM-dd HH-mm-ss`	`2016-02-02 02:02:02, 2016/2/2 2:2:2, 2016年2月2日 2时2分2秒`
`yyyy-MM-dd HH-mm`	`2016-02-02 02:02`
`yyyy-MM-dd HH`	`2016-02-02 02`
`yyyy-MM-dd`	`2016-02-02, 2016年02月02日`
`yyyy-MM`	`2016-02`
`yyyy`	`2016`
`yyyyMMddHHmmss`	`20160202020202`
`yyyyMMddHHmm`	`201602020202`
`yyyyMMddHH`	`2016020202`
`yyyyMMdd`	`20160202`
`yyyyMM`	`201602`
`HH:mm:ss`	`02:02:02`
`HH:mm`	`02:02`
`yy-MM-dd`	`02.02.02`
`dd-MM`	`02.02`
`dd-MM-yyyy`	`02.02.2016`
`java.text.DateFormat`支持的格式	`07/10/96 4:5 PM, PDT`

类org.rex.db.transaction.DefaultDefinition

类`org.rex.db.transaction.DefaultDefinition`用于设置事物选项。

- 该类声明了如下常量：

类型	常量	说明
`int`	`ISOLATION_DEFAULT`	默认事物的隔离级别。

int	ISOLATION_READ_UNCOMMITTED	指示防止发生脏读的常量；不可重复读和虚读有可能发生。
int	ISOLATION_READ_COMMITTED	指示可以发生脏读 (dirty read)、不可重复读和虚读 (phantom read) 的常量。
int	ISOLATION_REPEATABLE_READ	指示防止发生脏读和不可重复读的常量；虚读有可能发生。
int	ISOLATION_SERIALIZABLE	指示防止发生脏读、不可重复读和虚读的常量。
int	TIMEOUT_DEFAULT	默认的超时时间。

- 该类有如下接口：

返回值	接口	说明
int	getIsolationLevel()	获取事物的隔离级别。返回的值是类中以ISOLATION_开头的常量之一。
int	getTimeout()	获取事物超时时间。
int	isReadOnly()	是否是只读事务。
int	isAutoRollback()	提交失败时是否自动回滚。
void	setIsolationLevel(String isolationLevelName)	设置事物的隔离级别名称。参数isolationLevelName是类中以ISOLATION_开头的常量名称之一。
void	setIsolationLevel(int isolationLevel)	设置事物的隔离级别。参数isolationLevelName是类中以ISOLATION_开头的常量之一。
void	setTimeout(int timeout)	设置事物超时时间。
void	setReadOnly(boolean readOnly)	是否将事物设置为只读。
void	setAutoRollback(boolean autoRollback)	是否将事物设置为提交失败后自动回滚。