



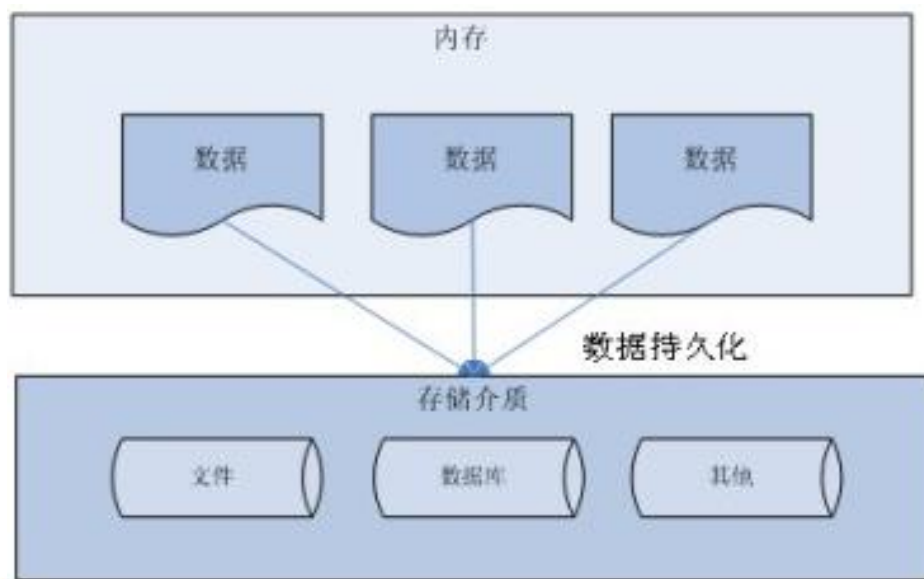
JDBC

讲师：佟刚

新浪微博：尚硅谷-佟刚

数据持久化

- 持久化(persistence): 把数据保存到可掉电式存储设备中以便之后使用。大多数情况下, 特别是企业级应用, 数据持久化意味着将内存中的数据保存到硬盘上加以”固化”, 而持久化的实现过程大多通过各种关系数据库来完成。
- 持久化的主要应用是将内存中的数据存储到关系型数据库中, 当然也可以存储在磁盘文件、XML数据文件中。

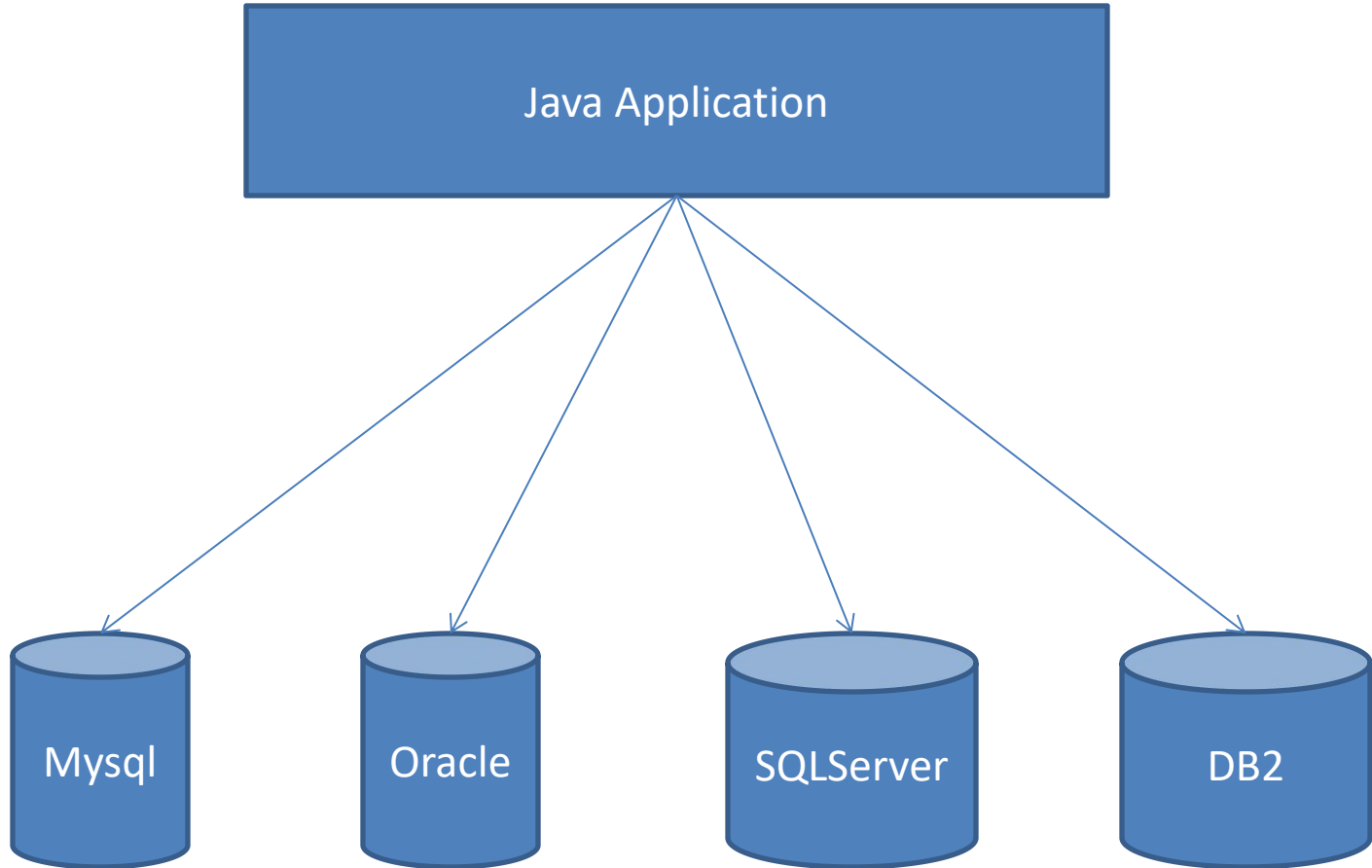


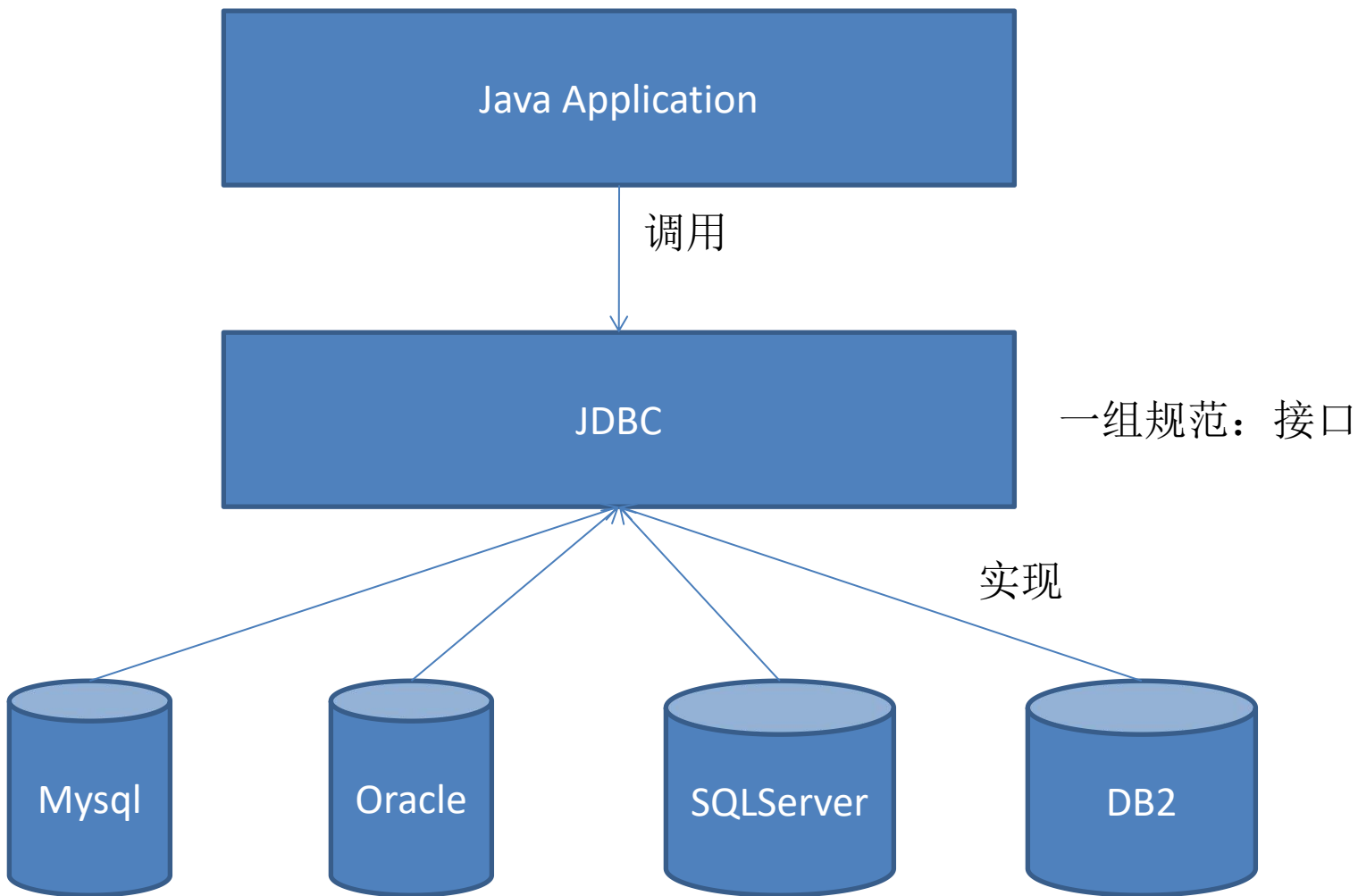
Java 中的数据存储服务技术

- 在Java中，数据库存取技术可分为如下几类：
 - **JDBC直接访问数据库**
 - JDO技术
 - 第三方O/R工具，如Hibernate, ibatis 等
- JDBC是java访问数据库的基石，JDO, Hibernate 等只是更好的封装了JDBC。

JDBC基础

- JDBC(Java Database Connectivity)是一个**独立于特定数据库管理系统、通用的SQL数据库存取和操作的公共接口**（一组API），定义了用来访问数据库的标准Java类库，使用这个类库可以以一种标准的方法、方便地访问数据库资源
- JDBC为访问不同的数据库提供了一种**统一的途径**，为开发者屏蔽了一些细节问题。
- JDBC的目标是使Java程序员使用JDBC可以连接任何**提供了JDBC驱动程序**的数据库系统，这样就使得程序员无需对特定的数据库系统的特点有过多的了解，从而大大简化和加快了开发过程。





Java Application

调用

JDBC

一组规范：接口

可行，但不建议，因为这意味着 Java 应用程序没有更好的可移植性

JDBCMysqlImpl

JDBCOracleImpl

JDBCSqlServerImpl

JDBCDB2Impl

JDBC驱动

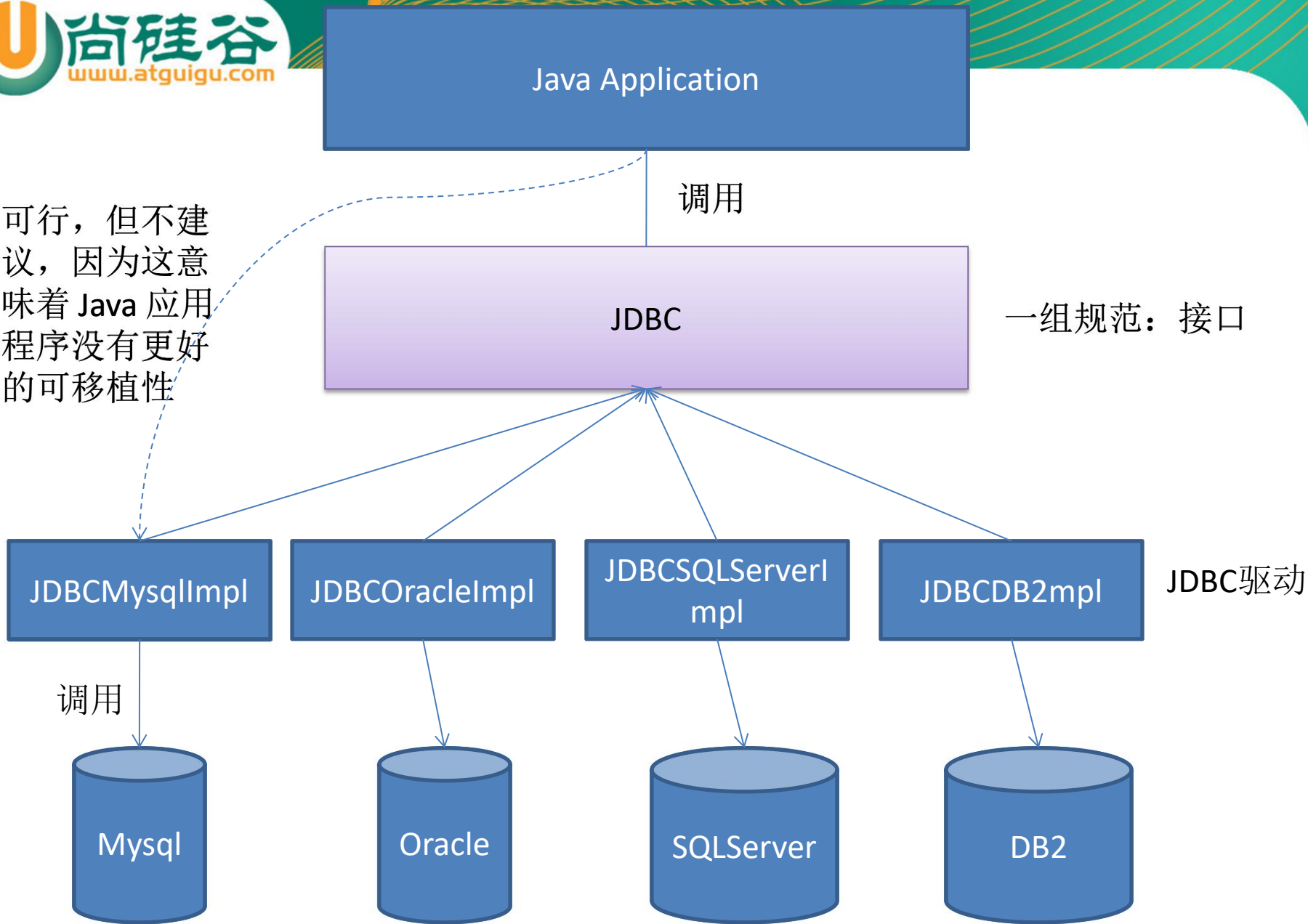
调用

Mysql

Oracle

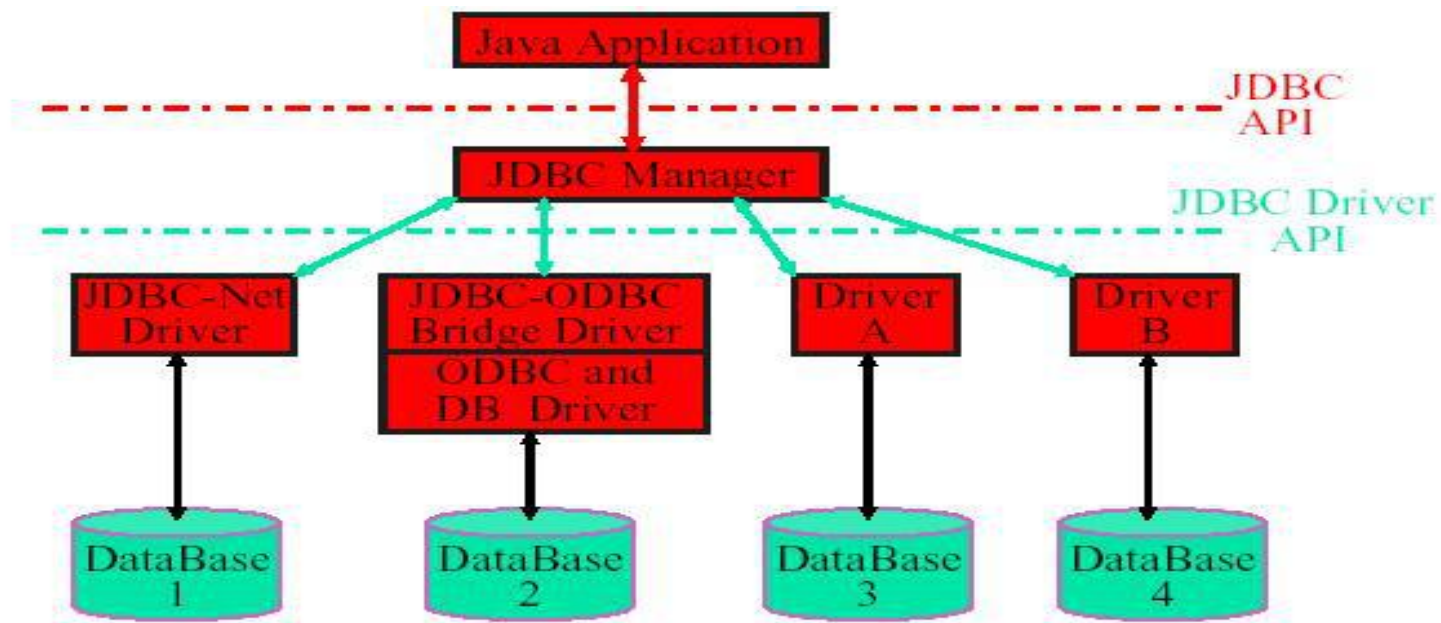
SQLServer

DB2



JDBC体系结构

- JDBC接口（API）包括两个层次：
 - 面向应用的**API**：Java API，抽象接口，供应用程序开发人员使用（连接数据库，执行SQL语句，获得结果）。
 - 面向数据库的**API**：Java Driver API，供开发商开发数据库驱动程序用。



JDBC驱动程序分类

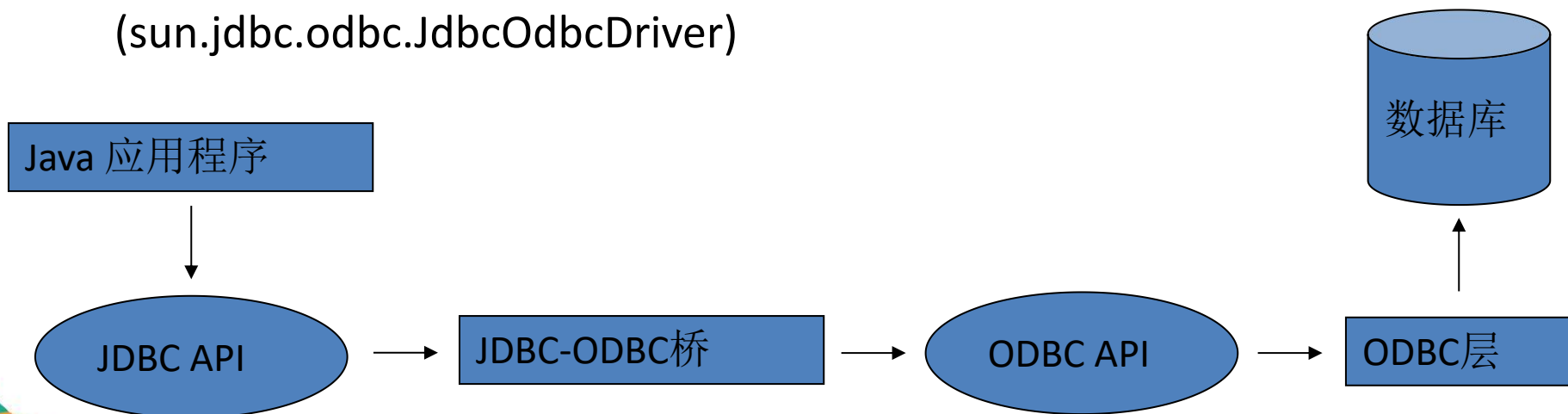
- JDBC驱动程序：各个数据库厂商根据JDBC的规范制作的 JDBC 实现类的类库
- JDBC驱动程序总共有四种类型：
 - 第一类：JDBC-ODBC桥。
 - 第二类：部分本地API部分Java的驱动程序。
 - 第三类：JDBC网络纯Java驱动程序。
 - **第四类：本地协议的纯 Java 驱动程序。**
 - 第三、四两类都是纯Java的驱动程序，因此，对于Java开发者来说，它们在性能、可移植性、功能等方面都有优势。

ODBC

- 早期对数据库的访问，都是调用数据库厂商提供的专有的API。为了在 **Windows 平台** 下提供统一的访问方式，微软推出了 ODBC (Open Database Connectivity, 开放式数据库连接)，并提供了 ODBC API，使用者在程序中只需要调用 ODBC API，由 ODBC 驱动程序将调用转换成为对特定的数据库的调用请求
- 一个基于ODBC的应用程序对数据库的操作不依赖任何DBMS (database manager system)，不直接与DBMS打交道，所有的数据库操作由对应的DBMS的**ODBC驱动程序**完成。也就是说，不论是FoxPro、Access，MySQL还是Oracle数据库，均可用ODBC API进行访问。由此可见，ODBC的最大优点是能以统一的方式处理所有的数据库。

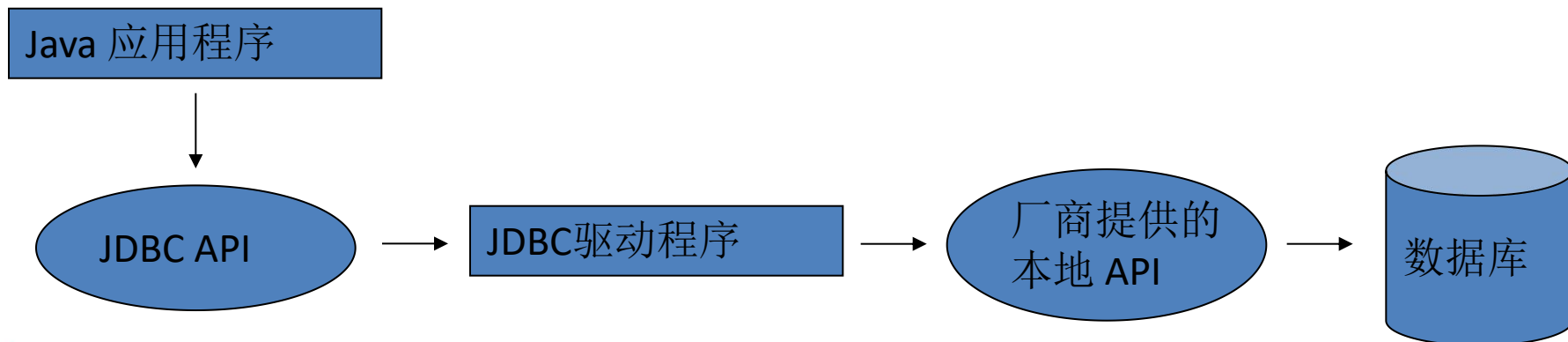
JDBC-ODBC桥

- JDBC-ODBC 桥本身也是一个驱动，利用这个驱动，可以使用 JDBC-API 通过 ODBC 去访问数据库。这种机制实际上是把标准的 JDBC 调用转换成相应的 ODBC 调用，并通过 ODBC 访问数据库
- 因为需要通过多层调用，所以利用 JDBC-ODBC 桥访问数据库的效率较低
- 在 JDK 中，提供了 JDBC-ODBC 桥的实现类 (sun.jdbc.odbc.JdbcOdbcDriver)



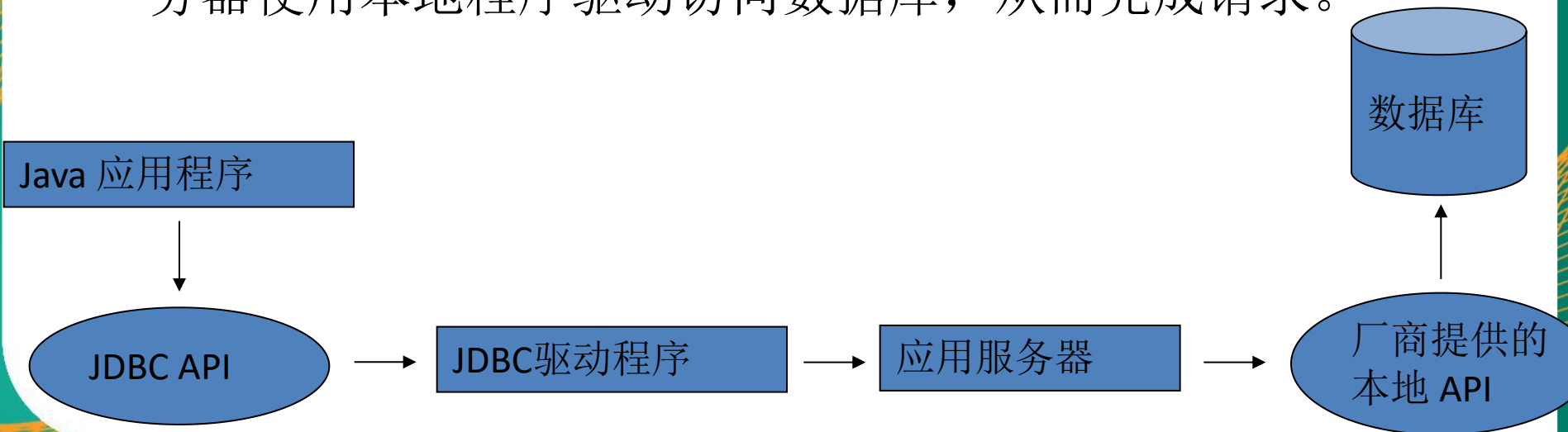
部分本地API部分Java的驱动程序

- 这种类型的 JDBC 驱动程序使用 Java 编写，它调用数据库厂商提供的本地 API
- 通过这种类型的 JDBC 驱动程序访问数据库减少了 ODBC 的调用环节，提高了数据库访问的效率
- 在这种方式下需要在客户的机器上安装本地 JDBC 驱动程序和特定厂商的本地 API



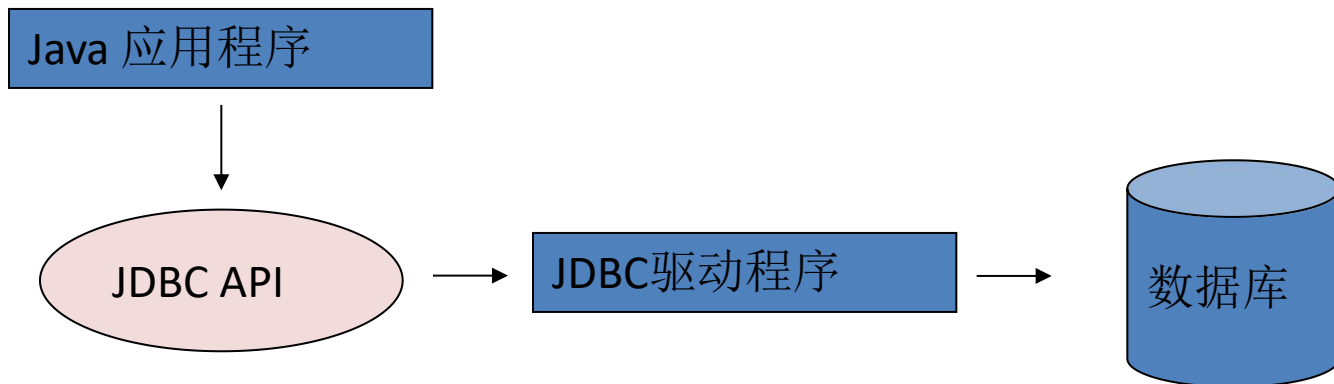
JDBC网络纯Java驱动程序

- 这种驱动利用中间件的应用服务器来访问数据库。应用服务器作为一个到多个数据库的网关，客户端通过它可以连接到不同的数据库服务器。
- 应用服务器通常有自己的网络协议，Java 用户程序通过 JDBC 驱动程序将 JDBC 调用发送给应用服务器，应用服务器使用本地程序驱动访问数据库，从而完成请求。



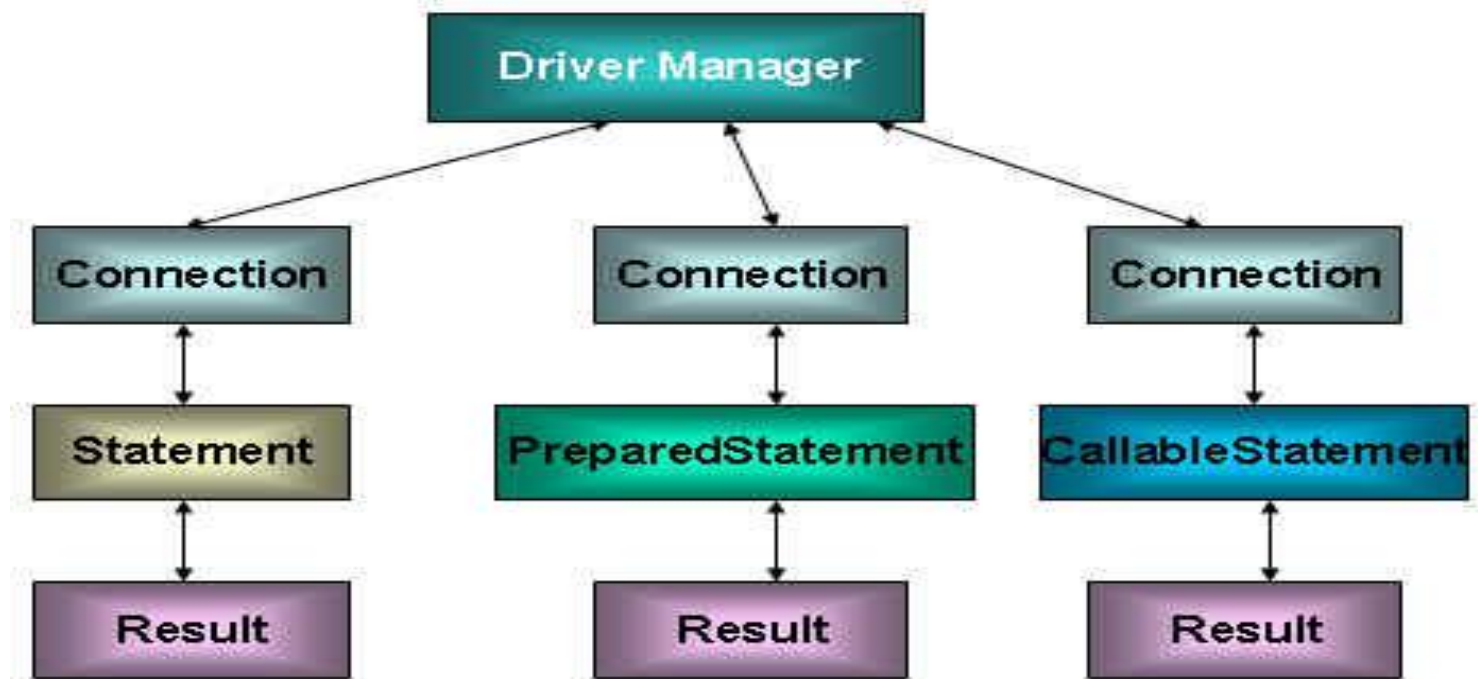
本地协议的纯 Java 驱动程序

- 多数数据库厂商已经支持允许客户程序通过网络直接与数据库通信的**网络协议**。
- 这种类型的驱动程序完全使用 Java 编写，通过与数据库建立的 **Socket** 连接，采用具体与厂商的网络协议把 JDBC 调用转换为直接连接的网络调用



JDBC API

- JDBC API 是一系列的接口，它使得应用程序能够进行数据库联接，执行SQL语句，并且得到返回结果。



Driver 接口

- `Java.sql.Driver` 接口是所有 JDBC 驱动程序需要实现的接口。这个接口是提供给数据库厂商使用的，不同数据库厂商提供不同的实现
- 在程序中不需要直接去访问实现了 `Driver` 接口的类，而是由驱动程序管理器类(`java.sql.DriverManager`)去调用这些 `Driver` 实现

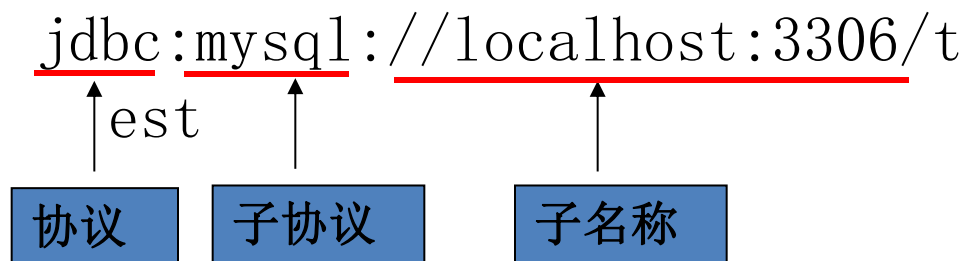
加载与注册 JDBC 驱动

- 加载 JDBC 驱动需调用 `Class` 类的静态方法 `forName()`，向其传递要加载的 JDBC 驱动的类型名
- `DriverManager` 类是驱动程序管理器类，负责管理驱动程序
- 通常不用显式调用 `DriverManager` 类的 `registerDriver()` 方法来注册驱动程序类的实例，因为 `Driver` 接口的驱动程序类都包含了静态代码块，在这个静态代码块中，会调用 `DriverManager.registerDriver()` 方法来注册自身的一个实例

建立连接

- 可以调用 `DriverManager` 类的 `getConnection()` 方法建立到数据库的连接
- JDBC URL 用于标识一个被注册的驱动程序，驱动程序管理器通过这个 URL 选择正确的驱动程序，从而建立到数据库的连接。
- JDBC URL 的标准由三部分组成，各部分间用冒号分隔。
 - `jdbc:<子协议>:<子名称>`
 - 协议：JDBC URL 中的协议总是 `jdbc`
 - 子协议：子协议用于标识一个数据库驱动程序
 - 子名称：一种标识数据库的方法。子名称可以依不同的子协议而变化，用子名称的目的是为了定位数据库提供足够的信息

几种常用数据库的JDBC URL



- 对于 Oracle 数据库连接，采用如下形式：
 - `jdbc:oracle:thin:@localhost:1521:sid`
- 对于 SQLServer 数据库连接，采用如下形式：
 - `jdbc:microsoft:sqlserver//localhost:1433; DatabaseName=sid`
- 对于 MYSQL 数据库连接，采用如下形式：
 - `jdbc:mysql://localhost:3306/sid`

访问数据库

- 数据库连接被用于向数据库服务器发送命令和 SQL 语句，在连接建立后，需要对数据库进行访问，执行 sql 语句
- 在 java.sql 包中有 3 个接口分别定义了对数据库的调用的不同方式：
 - Statement
 - PreparedStatement
 - CallableStatement

Statement

- 通过调用 Connection 对象的 createStatement 方法创建该对象
- 该对象用于执行静态的 SQL 语句，并且返回执行结果
- Statement 接口中定义了下列方法用于执行 SQL 语句：
 - ResultSet excuteQuery(String sql)
 - int excuteUpdate(String sql)


ResultSet

- 通过调用 Statement 对象的 `executeQuery()` 方法创建该对象
- ResultSet 对象以逻辑表格的形式封装了执行数据库操作的结果集，ResultSet 接口由数据库厂商实现
- ResultSet 对象维护了一个指向当前数据行的游标，初始的时候，游标在第一行之前，可以通过 ResultSet 对象的 `next()` 方法移动到下一行
- ResultSet 接口的常用方法：
 - `boolean next()`
 - `getString()`
 - ...

SELECT id, name, age, birth FROM customer_table

初始状态: 指向第一条记录的前面

next(): 若返回true,
就向下移动一行



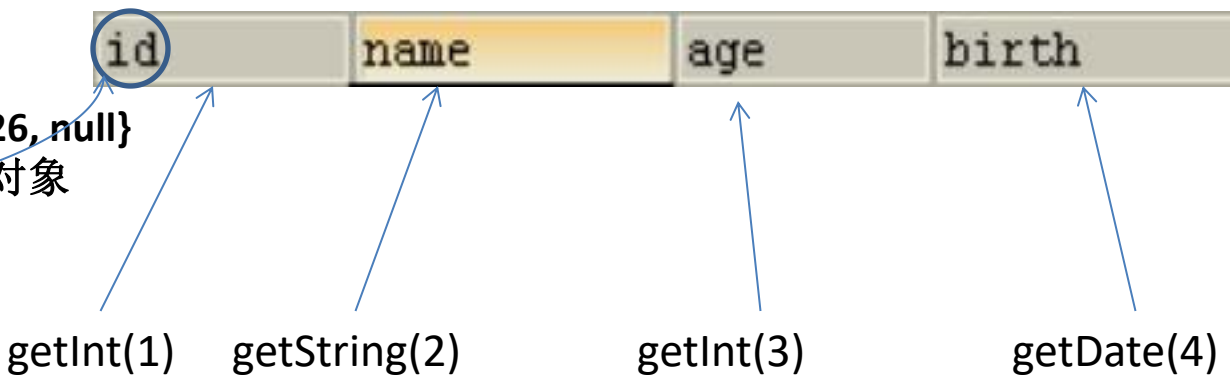
2	ChenXiang	25	1991-12-12
3	Arose	21	1990-12-13
4	Mike	26	(NULL)
5	Jinpeng	15	1990-11-11

- 1. 数组: new Object[]{4, "Mike", 26, null}
- 2. Customer: 一条记录对应一个对象

- 1. id
- 2. name
- 3. age
- 4. birth

id	name	age	birth
----	------	-----	-------

getInt(1) getString(2) getInt(3) getDate(4)



数据类型转换表

java类型	SQL类型
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
String	CHAR, VARCHAR, LONGVARCHAR
byte array	BINARY , VAR BINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

JDBC API 小结1

- java.sql.**DriverManager**用来装载驱动程序，获取数据库连接。
- java.sql.**Connection**完成对某一指定数据库的联接
- java.sql.**Statement**在一个给定的连接中作为SQL执行声明的容器，他包含了两个重要的子类型。
 - Java.sql.PreparedSatement 用于执行预编译的sql声明
 - Java.sql.CallableStatement用于执行数据库中存储过程的调用
- java.sql.**ResultSet**对于给定声明取得结果的途径

练习1

1. 创立数据库表 examstudent, 表结构如下:

字段名	说明	类型
FlowID	流水号	Int
Type	四级 / 六级	int
IDCard	身份证号码	Varchar(18)
ExamCard	准考证号码	Varchar(15)
StudentName	学生姓名	Varchar(20)
Location	区域	Varchar(20)
Grade	成绩	Int

练习1

2. 向数据库中添加如下数据

1	4	412824195263214584	200523164754000	张锋	郑州	85
2	4	222224195263214584	200523164754001	孙朋	大连	56
3	6	342824195263214584	200523164754002	刘明	沈阳	72
4	6	100824195263214584	200523164754003	赵虎	哈尔滨	95
5	4	454524195263214584	200523164754004	杨丽	北京	64
6	4	854524195263214584	200523164754005	王小红	太原	60

练习1

- 插入一个新的 student 信息

请输入考生的详细信息

Type:

IDCard:

ExamCard:

StudentName:

Location:

Grade:

信息录入成功!

练习1

3. 在 eclipse 中建立 java 程序：输入身份证号或准考证号可以查询到学生的基本信息。结果如下：

```
<terminated> TestGrade [Java Applica
```

```
请选择您要输入的类型：
```

```
a: 准考证号
```

```
b: 省份证号
```

```
c  
您的输入有误！请重新进入程序..
```

练习1

完成学生信息的删除功能

```
<terminated> TestGrade [Java Ap  
请输入学生的考号:  
3452435245245254  
查无此人! 请重新进入程序..
```

```
<terminated> TestGrade [Java  
请输入学生的考号:  
200523164754003  
删除成功!
```

SQL 注入攻击

- SQL 注入是利用某些系统没有对用户输入的数据进行充分的检查，而在用户输入数据中注入非法的 SQL 语句段或命令，从而利用系统的 SQL 引擎完成恶意行为的做法
- 对于 Java 而言，要防范 SQL 注入，只要用 PreparedStatement 取代 Statement 就可以了

PreparedStatement

- 可以通过调用 Connection 对象的 `prepareStatement()` 方法获取 PreparedStatement 对象
- PreparedStatement 接口是 Statement 的子接口，它表示一条预编译过的 SQL 语句
- PreparedStatement 对象所代表的 SQL 语句中的参数用问号 (?) 来表示，调用 PreparedStatement 对象的 `setXXX()` 方法来设置这些参数。setXXX() 方法有两个参数，第一个参数是要设置的 SQL 语句中的参数的索引(从 1 开始)，第二个是设置的 SQL 语句中的参数的值

PreparedStatement vs Statement

- 代码的可读性和可维护性.
- PreparedStatement 能最大可能提高性能:
 - DBServer会对预编译语句提供性能优化。因为预编译语句有可能被重复调用, 所以语句在被DBServer的编译器编译后的执行代码被缓存下来, 那么下次调用时只要是相同的预编译语句就不需要编译, 只要将参数直接传入编译过的语句执行代码中就会得到执行。
 - 在statement语句中,即使是相同操作但因为数据内容不一样,所以整个语句本身不能匹配,没有缓存语句的意义.事实是没有数据库会对普通语句编译后的执行代码缓存.这样每执行一次都要对传入的语句编译一次.
 - (语法检查, 语义检查, 翻译成二进制命令, 缓存)
- PreparedStatement 可以防止 SQL 注入

SQL

flowId
type
idCard
examCard
studentName
location
grade

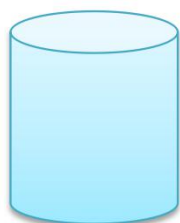
flowId	type	idCard	examCard	studentName	location	grade
5	6	21121199002045677	19901212	Wang	PanJIn	60

```
public class Student {  
  
    // 流水号  
    private int flowId;  
    // 考试的类型  
    private int type;  
    // 身份证号  
    private String idCard;  
    // 准考证号  
    private String examCard;  
    // 学生名  
    private String studentName;  
    // 学生地址  
    private String location;  
    // 考试分数。  
    private int grade;  
}
```

1. 先利用 SQL 进行查询，得到结果集
2. 利用反射创建实体类的对象：创建 Student 对象
3. 获取结果集的列的别名：idCard、studentName
4. 再获取结果集的每一列的值，结合 3 得到一个 Map，键：列的别名，值：列的值：{flowId:5, type:6, idCard: xxx}
5. 再利用反射为 2 的对应的属性赋值：属性即为 Map 的键，值即为 Map 的值

```
String sql = "SELECT flow_id flowId, type, id_card idCard, "
    + "exam_card examCard, student_name studentName, "
    + "location, grade " + "FROM examstudent WHERE flow_id = ?";
```

查询（要求：列的别名要和 Class 对应的类的属性名相同），得到 ResultSet 对象



得到
ResultSetM
etaData 对
象：可以
知道 SQL 语
句中查询了
哪些列，以
及列的别名
都是什么

flowId	type	idCard	examCard	studentName	location	grade
5	6	21121199002045677	19901212	Wang	PanJin	

反射：为
对应的属
性赋值

Class 对应的
类对象

Class 对象

反射：创
建对象

```
// 流水号
private int flowId;
// 考试的类型
private int type;
// 身份证号
private String idCard;
// 准考证号
private String examCard;
// 学生名
private String studentName;
// 学生地址
private String location;
// 考试分数。
private int grade;
```

flowId	type	idCard	examCard	...
--------	------	--------	----------	-----

使用 JDBC 驱动程序处理元数据

- Java 通过JDBC获得连接以后，得到一个Connection 对象，可以从这个对象获得**有关数据库管理系统的各种信息**，包括数据库中的各个表，表中的各个列，数据类型，触发器，存储过程等各方面的信息。根据这些信息，JDBC 可以访问一个实现事先并不了解的数据库。
- 获取这些信息的方法都是在**DatabaseMetaData**类的对象上实现的，而DataBaseMetaData对象是在Connection对象上获得的。

DatabaseMetaData类

- DatabaseMetaData 类中提供了许多方法用于获得数据源的各种信息，通过这些方法可以非常详细的了解数据库的信息：
 - `getURL()`：返回一个String类对象，代表数据库的URL。
 - `getUserName()`：返回连接当前数据库管理系统的用户名。
 - `isReadOnly()`：返回一个boolean值，指示数据库是否只允许读操作。
 - `getDatabaseProductName()`：返回数据库的产品名称。
 - `getDatabaseProductVersion()`：返回数据库的版本号。
 - `getDriverName()`：返回驱动驱动程序的名称。
 - `getDriverVersion()`：返回驱动程序的版本号。

ResultSetMetaData 类

- 可用于获取关于 ResultSet 对象中列的类型和属性信息的对象：
 - **getColumnName(int column)**: 获取指定列的名称
 - **getColumnCount()**: 返回当前 ResultSet 对象中的列数。
 - **getColumnTypeName(int column)**: 检索指定列的数据库特定的类型名称。
 - **getColumnDisplaySize(int column)**: 指示指定列的最大标准宽度，以字符为单位。
 - **isNullable(int column)**: 指示指定列中的值是否可以为 null。
 - **isAutoIncrement(int column)**: 指示是否自动为指定列进行编号，这样这些列仍然是只读的。

取得数据库自动生成的主键

- 示例:

```
Connection conn = JdbcUtil.getConnection();

String sql = "insert into user(name,password,email,birthday)
              values('abc','123','abc@sina.com','1978-08-08')";
PreparedStatement st = conn.
    prepareStatement(sql,Statement.RETURN_GENERATED_KEYS );

st.executeUpdate();
ResultSet rs = st.getGeneratedKeys(); //得到插入行的主键
if(rs.next())
    System.out.println(rs.getObject(1));
```

Oracle LOB

- LOB, 即Large Objects (大对象), 是用来存储大量的二进制和文本数据的一种数据类型 (一个LOB字段可存储可多达4GB的数据)。
- LOB 分为两种类型: 内部LOB和外部LOB。
 - 内部LOB将数据以字节流的形式存储在数据库的内部。因而, 内部LOB的许多操作都可以参与事务, 也可以像处理普通数据一样对其进行备份和恢复操作。
Oracle支持三种类型的内部LOB:
 - BLOB (二进制数据)
 - CLOB (单字节字符数据)
 - NCLOB (多字节字符数据)。
 - CLOB和NCLOB类型适用于存储超长的文本数据, BLOB字段适用于存储大量的二进制数据, 如图像、视频、音频, 文件等。
 - 目前只支持一种外部LOB类型, 即BFILE类型。在数据库内, 该类型仅存储数据在操作系统中的位置信息, 而数据的实体以外部文件的形式存在于操作系统的文件系统中。因而, 该类型所表示的数据是只读的, 不参与事务。该类型可帮助用户管理大量的由外部程序访问的文件。

MySQL BLOB 类型介绍

- MySQL中，BLOB是一个二进制大型对象，是一个可以存储大量数据的容器，它能容纳不同大小的数据。
- MySQL的四种BLOB类型(除了在存储的最大信息量上不同外，他们是等同的)

类型	大小(单位: 字节)
TinyBlob	最大 255
Blob	最大 65K
MediumBlob	最大 16M
LongBlob	最大 4G

- 实际使用中根据需要存入的数据大小定义不同的BLOB类型。
需要注意的是：如果存储的文件过大，数据库的性能会下降。

使用JDBC来写入Blob型数据到Oracle中

- **Oracle的Blob字段**比long字段的性能要好，可以用来保存如图片之类的二进制数据。
- **Oracle的BLOB字段由两部分组成：数据（值）和指向数据的指针（定位器）。**尽管值与表自身一起存储，但是一个BLOB列并不包含值，仅有它的定位指针。为了使用大对象，程序必须声明定位器类型的本地变量。
- 当Oracle内部LOB被创建时，**定位器被存放在列中，值被存放在LOB段中，LOB段是在数据库内部表的一部分。**
- 因为Blob自身有一个cursor，当写入Blob字段必须使用指针（定位器）对Blob进行操作，**因而在写入Blob之前，必须获得指针（定位器）才能进行写入**
- 如何获得Blob的指针（定位器）：需要先插入一个empty的blob，这将创建一个blob的指针，然后再把这个empty的blob的指针查询出来，这样通过两步操作，就获得了blob的指针，可以真正的写入blob数据了。

步骤

- 1、插入空blob
`insert into javatest(name,content) values(?,empty_blob());`
- 2、获得blob的cursor
`select content from javatest where name= ? for update;`

注意：须加**for update**，锁定该行，直至该行被修改完毕，保证不产生并发冲突。

- 3、利用 `io`，和获取到的**cursor**往数据库写数据流

数据库事务

- 在数据库中, 所谓事务是指**一组逻辑操作单元, 使数据从一种状态变换到另一种状态。**
- 为确保数据库中数据**的一致性**, 数据的操纵应当是离散的成组的逻辑单元: 当它全部完成时, 数据的一致性可以保持, 而当这个单元中的一部分操作失败, 整个事务应全部视为错误, 所有从起始点以后的操作应全部回退到开始状态。
- 事务的操作: 先定义开始一个事务, 然后对数据作修改操作, 这时如果**提交** (COMMIT), 这些修改就永久地保存下来, 如果**回退** (ROLLBACK), 数据库管理系统将放弃所作的所有修改而回到开始事务时的状态。

数据库事务

- 事务的ACID(acid)属性
 - 1. **原子性** (Atomicity)
原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
 - 2. **一致性** (Consistency)
事务必须使数据库从一个一致性状态变换到另外一个一致性状态。
 - 3. **隔离性** (Isolation)
事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
 - 4. **持久性** (Durability)
持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响

JDBC 事物处理

- 事务：指构成单个逻辑工作单元的操作集合
- 事务处理：保证所有事务都作为一个工作单元来执行，即使出现了故障，都不能改变这种执行方式。当在一个事务中执行多个操作时，要么所有的事务都被提交(commit)，要么整个事务回滚(rollback)到最初状态
- 当一个连接对象被创建时，默认情况下是自动提交事务：每次执行一个 SQL 语句时，如果执行成功，就会向数据库自动提交，而不能回滚
- 为了让多个 SQL 语句作为一个事务执行：
 - 调用 Connection 对象的 setAutoCommit(false); 以取消自动提交事务
 - 在所有的 SQL 语句都成功执行后，调用 commit(); 方法提交事务
 - 在出现异常时，调用 rollback(); 方法回滚事务
 - 若此时 Connection 没有被关闭, 则需要恢复其自动提交状态

数据库的隔离级别

- 对于同时运行的多个事务，当这些事务访问数据库中相同的数据时，如果没有采取必要的隔离机制，就会导致各种并发问题：
 - **脏读**：对于两个事物 T1, T2, T1 读取了已经被 T2 更新但还没有被提交的字段。之后，若 T2 回滚，T1 读取的内容就是临时且无效的。
 - **不可重复读**：对于两个事物 T1, T2, T1 读取了一个字段，然后 T2 更新了该字段。之后，T1 再次读取同一个字段，值就不同了。
 - **幻读**：对于两个事物 T1, T2, T1 从一个表中读取了一个字段，然后 T2 在该表中插入了一些新的行。之后，如果 T1 再次读取同一个表，就会多出几行。
- 数据库事务的隔离性：数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。
- 一个事务与其他事务隔离的程度称为隔离级别。数据库规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性越弱。

数据库的隔离级别

- 数据库提供的 4 种事务隔离级别：

隔离级别	描述
READ UNCOMMITTED (读未提交数据)	允许事务读取未被其他事物提交的变更,脏读,不可重复读和幻读的问题都会出现
READ COMMITED (读已提交数据)	只允许事务读取已经被其它事务提交的变更,可以避免脏读,但不可重复读和幻读问题仍然可能出现
REPEATABLE READ (可重复读)	确保事务可以多次从一个字段中读取相同的值,在这个事务持续期间,禁止其他事物对这个字段进行更新,可以避免脏读和不可重复读,但幻读的问题仍然存在.
SERIALIZABLE(串行化)	确保事务可以从一个表中读取相同的行,在这个事务持续期间,禁止其他事务对该表执行插入,更新和删除操作,所有并发问题都可以避免,但性能十分低下.

- Oracle 支持的 2 种事务隔离级别：**READ COMMITED**,
SERIALIZABLE. Oracle 默认的事务隔离级别为：READ
COMMITTED
- Mysql 支持 4 中事务隔离级别. Mysql 默认的事务隔离级
别为：REPEATABLE READ

在 MySQL 中设置隔离级别

- 每启动一个 mysql 程序，就会获得一个单独的数据库连接。每个数据库连接都有一个全局变量 @@tx_isolation，表示当前的事务隔离级别。MySQL 默认的隔离级别为 Repeatable Read
- 查看当前的隔离级别：SELECT @@tx_isolation;
- 设置当前 MySQL 连接的隔离级别：
 - set transaction isolation level read committed;
- 设置数据库系统的全局的隔离级别：
 - set global transaction isolation level read committed;

批量处理JDBC语句提高处理速度

- 当需要成批插入或者更新记录时。可以采用Java的批量**更新**机制，这一机制允许多条语句一次性提交给数据库批量处理。通常情况下比单独提交处理更有效率
- JDBC的批量处理语句包括下面两个方法：
 - `addBatch(String)`: 添加需要批量处理的SQL语句或是参数;
 - `executeBatch()`; 执行批量处理语句;
- 通常我们会遇到两种批量执行SQL语句的情况：
 - 多条SQL语句的批量处理;
 - 一个SQL语句的批量传参;

多条SQL语句的批量处理

```
...  
Statement st = conn.createStatement();  
st.addBatch(sql1);  
st.addBatch(sql2);  
...  
st.addBatch(sqln);  
st.executeBatch();  
...
```

一个SQL语句的批量传参

- 情景

```
...
PreparedStatement pst = conn.prepareStatement(sql);
for(int i=0;i<N;i++){
    pst.setInt(1,i);
    ....
    pst.executeQuery();
}
...
```

- 解决

```
...
PreparedStatement pst = conn.prepareStatement(sql);
for(int i=100;i<1101;i++){
    pst.setInt(1,i);
    ...
    pst.addBatch();
}
pst.executeBatch();
...
```


JDBC数据库连接池的必要性

- 在使用开发基于数据库的web程序时，**传统的模式**基本是按以下步骤：
 - 在主程序（如servlet、beans）中建立数据库连接。
 - 进行sql操作
 - 断开数据库连接。
- 这种模式开发，存在的问题：
 - 普通的JDBC数据库连接使用 DriverManager 来获取，每次向数据库建立连接的时候都要将 Connection 加载到内存中，再验证用户名和密码(得花费0.05s~1s的时间)。需要数据库连接的时候，就向数据库要求一个，执行完成后断开连接。这样的方式将会消耗大量的资源和时间。**数据库的连接资源并没有得到很好的重复利用**。若同时有几百人甚至几千人在线，频繁的进行数据库连接操作将占用很多的系统资源，严重的甚至会造成服务器的崩溃。
 - 对于每一次数据库连接，使用完后都得断开。否则，如果程序出现异常而未能关闭，将会导致数据库系统中的内存泄漏，最终将导致重启数据库。
 - 这种开发不能控制被创建的连接对象数，系统资源会被毫无顾及的分配出去，如连接过多，也可能导致内存泄漏，服务器崩溃。

数据库连接池（connection pool）

- 为解决传统开发中的数据库连接问题，可以采用数据库连接池技术。
- 数据库连接池的**基本思想**就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。
- **数据库连接池**负责分配、管理和释放数据库连接，它**允许应用程序重复使用一个现有的数据库连接，而不是重新建立一个**。
- 数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由**最小数据库连接数**来设定的。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的**最大数据库连接数量**限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。

conn1 **free**

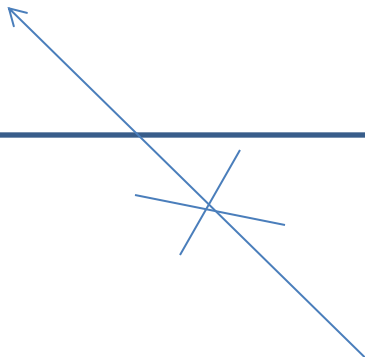
conn3 **free**

conn2 **free**

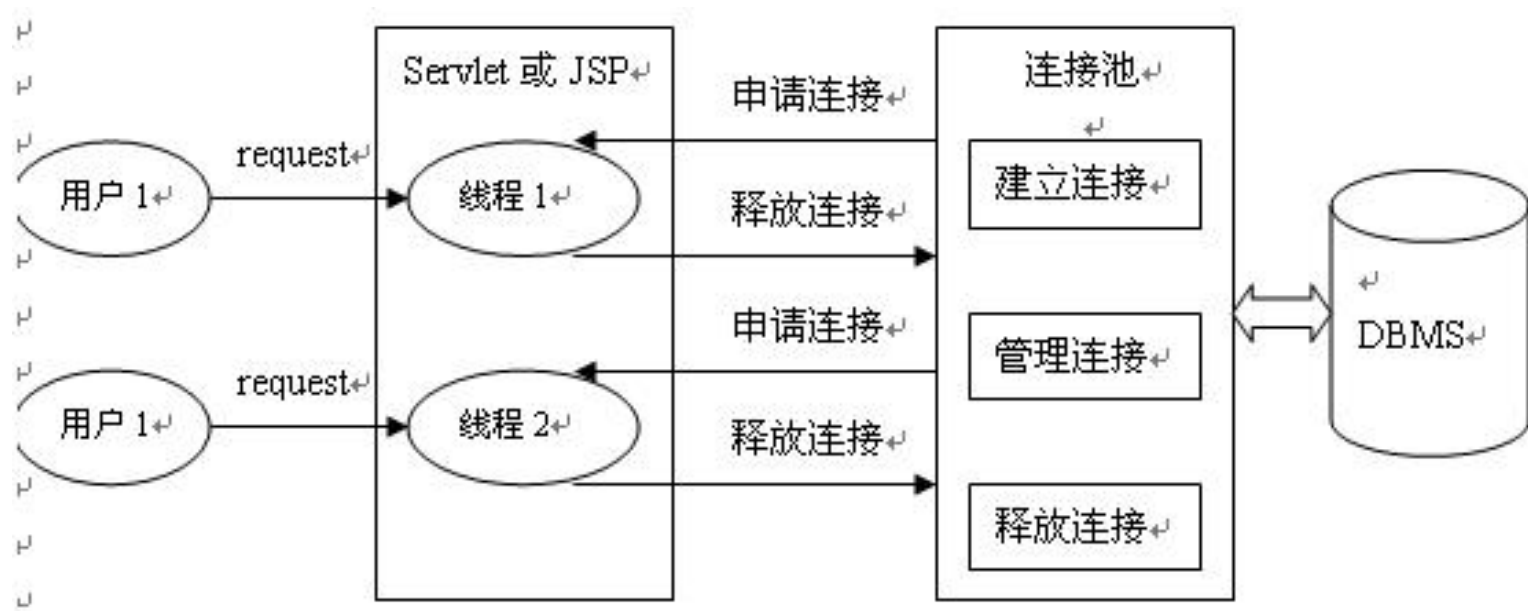
conn4 **free**

数据库连接池

Java Application



数据库连接池的工作原理



数据库连接池技术的优点

- 资源重用：
 - 由于数据库连接得以重用，避免了频繁创建，释放连接引起的大量性能开销。在减少系统消耗的基础上，另一方面也增加了系统运行环境的平稳性。
- 更快的系统反应速度
 - 数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于连接池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而减少了系统的响应时间
- 新的资源分配手段
 - 对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接池的配置，实现某一应用最大可用数据库连接数的限制，避免某一应用独占所有的数据库资源
- 统一的连接管理，避免数据库连接泄露
 - 在较为完善的数据库连接池实现中，可根据预先的占用超时设定，强制回收被占用连接，从而避免了常规数据库连接操作中可能出现的资源泄露

两种开源的数据库连接池

- JDBC 的数据库连接池使用 `javax.sql.DataSource` 来表示，`DataSource` 只是一个接口，该接口通常由服务器(Weblogic, WebSphere, Tomcat)提供实现，也有一些开源组织提供实现：
 - DBCP 数据库连接池
 - C3P0 数据库连接池
- `DataSource` 通常被称为数据源，它包含连接池和连接池管理两个部分，习惯上也经常把 `DataSource` 称为连接池

DBCP 数据源

- DBCP 是 Apache 软件基金组织下的开源连接池实现，该连接池依赖该组织下的另一个开源系统：Common-pool。如需使用该连接池实现，应在系统中增加如下两个 jar 文件：
 - Commons-dbc.jar: 连接池的实现
 - Commons-pool.jar: 连接池实现的依赖库
- Tomcat 的连接池正是采用该连接池来实现的。该数据库连接池既可以与应用服务器整合使用，也可由应用程序独立使用。

DBCP 数据源使用范例

- 数据源和数据库连接不同，数据源无需创建多个，它是产生数据库连接的工厂，因此整个应用只需要一个数据源即可。
- 当数据库访问结束后，程序还是像以前一样关闭数据库连接：`conn.close()`；但上面的代码并没有关闭数据库的物理连接，它仅仅把数据库连接释放，归还给了数据库连接池。


```
BasicDataSource ds = null;
//创建数据源对象
ds = new BasicDataSource();
//设置连接数据库的 驱动
ds.setDriverClassName("com.mysql.jdbc.Driver");
//设置连接数据库的 url
ds.setUrl("jdbc:mysql://localhost:3309/test");
//设置连接数据库的 用户名
ds.setUsername("root");
//设置连接数据库的 密码
ds.setPassword("1230");
//设置数据库连接池的 初始连接数
ds.setInitialSize(5);
//设置连接池最多可有多少个活动连接数
ds.setMaxActive(20);
//设置连接池中最少有 2 个空闲连接
ds.setMinIdle(2);

Connection conn = null;
try {
    conn = ds.getConnection();
    System.out.print(conn);
} catch (SQLException e) {
    e.printStackTrace();
}
```

C3P0 数据源

```
ComboPooledDataSource ds1 = null;
ds1 = new ComboPooledDataSource();
try {
    ds1.setDriverClass("com.mysql.jdbc.Driver");
    ds1.setJdbcUrl("jdbc:mysql://localhost:3309/test");
    ds1.setUser("root");
    ds1.setPassword("1230");
    ds1.setMaxPoolSize(40);
    ds1.setMinPoolSize(2);
    ds1.setInitialPoolSize(10);

    Connection conn1 = ds1.getConnection();
    System.out.println(conn1);

} catch (Exception e) {
    e.printStackTrace();
}
```

Apache—DBUtils 简介

- commons-dbutils 是 Apache 组织提供的一个开源 JDBC 工具类库，它是对 JDBC 的简单封装，学习成本极低，并且使用 dbutils 能极大简化 jdbc 编码的工作量，同时也不会影响程序的性能。
- API 介绍：
 - org.apache.commons.dbutils.QueryRunner
 - org.apache.commons.dbutils.ResultSetHandler
 - 工具类
 - org.apache.commons.dbutils.DbUtils、。


```
Object obj = queryRunner.query(connection, sql,  
    new MyResultSetHandler());
```

QueryRunner

```
query(Connection conn, String sql, ResultSetHandler rsh){
```

```
    stmt = this.prepareStatement(conn, sql);  
    this.fillStatement(stmt, params);  
    rs = this.wrap(stmt.executeQuery());
```

得到结果集对象

```
    result = rsh.handle(rs);
```

调用传入的 ResultSetHandler 对象的 handle 方法, 并且把前面得到的 ResultSet 对象作为参数传入

```
    return result; 把 result 作为结果返回
```

```
}
```

```
class MyResultSetHandler implements ResultSetHandler{
```

```
    @Override  
    public Object handle(ResultSet resultSet)  
        throws SQLException {  
        System.out.println("handle...");  
        return "atguigu";  
    }
```

```
}
```


DAO<T>

- batch(Connection, String, Object[]) : void
- getForValue(Connection, String, Object...) <E> : E
- getForList(Connection, String, Object...) : List<T>
- get(Connection, String, Object...) : T
- update(Connection, String, Object...) : void



JdbcDaoImpl<T>

- queryRunner : QueryRunner
- type : Class<T>



```
public JdbcDaoImpl() {  
    queryRunner = new QueryRunner();  
    type = ReflectionUtils.getSuperGenericType(getClass());  
}
```

```
public class CustomerDao  
    extends JdbcDaoImpl<Customer>{  
  
}
```

DbUtils类

- DbUtils：提供如关闭连接、装载JDBC驱动程序等常规工作的工具类，里面的所有方法都是静态的。主要方法如下：
 - `public static void close(...) throws java.sql.SQLException`： DbUtils类提供了三个重载的关闭方法。这些方法检查所提供的参数是不是NULL，如果不是的话，它们就关闭Connection、Statement和ResultSet。
 - `public static void closeQuietly(...)`：这一类方法不仅能在Connection、Statement和ResultSet为NULL情况下避免关闭，还能隐藏一些在程序中抛出的SQLException。
 - `public static void commitAndCloseQuietly(Connection conn)`：用来提交连接，然后关闭连接，并且在关闭连接时不抛出SQL异常。
 - `public static boolean loadDriver(java.lang.String driverClassName)`：这一方装载并注册JDBC驱动程序，如果成功就返回true。使用该方法，你不需要捕捉这个异常ClassNotFoundException。

QueryRunner类

- 该类简单化了SQL查询，它与ResultSetHandler组合在一起使用可以完成大部分的数据库操作，能够大大减少编码量。
- QueryRunner类提供了两个构造方法：
 - 默认的构造方法
 - 需要一个 `javax.sql.DataSource` 来作参数的构造方法。

QueryRunner类的主要方法

- `public Object query(Connection conn, String sql, Object[] params, ResultSetHandler rsh) throws SQLException`: 执行一个查询操作，在这个查询中，对象数组中的每个元素值被用来作为查询语句的置换参数。该方法会自行处理 `PreparedStatement` 和 `ResultSet` 的创建和关闭。
- `public Object query(String sql, Object[] params, ResultSetHandler rsh) throws SQLException`: 几乎与第一种方法一样；唯一的不同在于它不将数据库连接提供给方法，并且它是从提供给构造方法的数据源(`DataSource`) 或使用的 `setDataSource` 方法中重新获得 `Connection`。
- `public Object query(Connection conn, String sql, ResultSetHandler rsh) throws SQLException`: 执行一个不需要置换参数的查询操作。
- `public int update(Connection conn, String sql, Object[] params) throws SQLException`: 用来执行一个更新（插入、更新或删除）操作。
- `public int update(Connection conn, String sql) throws SQLException`: 用来执行一个不需要置换参数的更新操作。

***ResultSetHandler*接口**

- 该接口用于处理 `java.sql.ResultSet`，将数据按要求转换为另一种形式。
- `ResultSetHandler` 接口提供了一个单独的方法：`Object handle (java.sql.ResultSet rs)`。

ResultSetHandler 接口的实现类

- **ArrayHandler**: 把结果集中的第一行数据转成对象数组。
- **ArrayListHandler**: 把结果集中的每一行数据都转成一个数组，再存放到List中。
- **BeanHandler**: 将结果集中的第一行数据封装到一个对应的JavaBean实例中。
- **BeanListHandler**: 将结果集中的每一行数据都封装到一个对应的JavaBean实例中，存放到List里。

ResultSetHandler 接口的实现类

- ColumnListHandler: 将结果集中某一列的数据存放到List中。
- KeyedHandler(name): 将结果集中的每一行数据都封装到一个Map里，再把这些map再存到一个map里，其key为指定的key。
- MapHandler: 将结果集中的第一行数据封装到一个Map里，key是列名，value就是对应的值。
- MapListHandler: 将结果集中的每一行数据都封装到一个Map里，然后再存放到List

数据库的分页语句

- 在编写Web应用程序等系统时，会涉及到与数据库的交互，如果数据库中数据量很大的话，一次检索所有的记录，会占用系统很大的资源，因此常常采用分页语句：需要多少数据就只从数据库中取多少条记录。以下是Sql Server, Oracle和MySQL的分页语句(从数据库表中的第M条数据开始取N条记录):

SQL Server

- 从数据库表中的第M条记录开始取N条记录，利用**Top**关键字(如果Select语句中既有top，又有order by，则是从排序好的结果集中选择):

SELECT *

FROM (SELECT Top N *

FROM (SELECT Top (M + N - 1) * FROM 表名称 Order by 主键 desc) t1) t2

Order by 主键 asc

- 例如从表Sys_option(主键为sys_id)中从10条记录还是检索20条记录，语句如下：

SELECT *

FROM (SELECT TOP 20 *

FROM (SELECT TOP 29 * FROM Sys_option order by sys_id desc) t1) t2

Order by sys_id asc

Oracle数据库

- 从数据库表中第M条记录开始检索N条记录

```
SELECT *  
FROM (  
    SELECT ROWNUM rownum_, t1.*  
    FROM (  
        SELECT *  
        FROM table  
        ORDER BY table.id DESC  
    ) row_  
    WHERE rownum <= ?)  
WHERE rownum_ > ?
```

- 例如从表employees(主键为employee_id)中从11条记录还是检索20条记录，语句如下：

```
SELECT *  
FROM (  
    SELECT rownum r_, row_.*  
    FROM (  
        SELECT *  
        FROM employees  
        ORDER BY employee_id DESC  
    ) row_  
    WHERE rownum <= 20  
)  
WHERE r_ >= 11
```

MySQL数据库

- MySQL数据库最简单，是利用MySQL的LIMIT函数,LIMIT [offset,] rows从数据库表中M条记录开始检索N条记录的语句为：

SELECT [列名列表] FROM 表名称 LIMIT M,N

- 例如从表Sys_option(主键为sys_id)中从10条记录还是检索20条记录，语句如下：

select * from sys_option limit 10,20

