# 连接池

<http://blog.csdn.net/wenwen091100304/article/details/48035003>

<https://www.cnblogs.com/wym789/p/6374440.html>

数据库连接池（Connection pooling）是程序启动时建立足够的数据库连接，并将这些连接组成一个连接池，由**程序动态地对池中的连接进行申请，使用，释放**。**复用连接。通过建立一个数据库连接池以及一套连接使用管理策略，使得一个数据库连接可以得到高效、安全的复用，避免了数据库连接频繁建立、关闭的开销。**

创建数据库连接是一个很耗时的操作，也容易对数据库造成安全隐患。所以，在程序初始化的时候，集中创建多个数据库连接，并把他们集中管理，供程序使用，可以保证**较快的数据库读写速度，还更加安全可靠**。

对于一个简单的数据库应用，由于对于数据库的访问不是很频繁。这时可以简单地在需要访问数据库时，就新创建一个连接，用完后就关闭它，这样做也不会带来什么明显的性能上的开销。但是对于一个复杂的数据库应用，情况就完全不同了。**频繁的建立、关闭连接，会极大的降低系统的性能**，因为对于连接的使用成了系统性能的瓶颈。

## 优势

1. **资源重用**

由于**数据库连接得到重用**，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，另一方面也增进了系统运行环境的平稳性（减少内存碎片以及数据库临时进程/线程的数量）。

1. **更快的系统响应速度**

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，从而缩减了系统整体响应时间。

1. **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的配置，实现数据库连接池技术，几年前也许还是个新鲜话题，对于目前的业务系统而言，如果设计中还没有考虑到连接池的应用，那么…….快在设计文档中加上这部分的内容吧。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数据库资源。

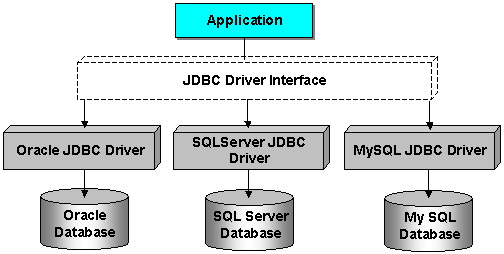
1. **统一的连接管理，避免数据库连接泄漏**

在较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用连接。从而避免了常规数据库连接操作中可能出现的资源泄漏。

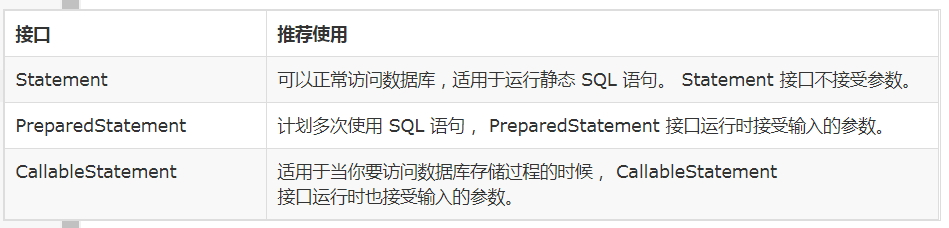
## JDBC

JDBC是一个**规范**，遵循JDBC接口规范，各个数据库厂家各自实现自己的驱动程序(Driver)。

JDBC提供了完备的数据库操作方法接口。但考虑到规范的适用性，JDBC只提供了最直接的数据库操作规范。



在完成数据操作后，还一定要关闭所有涉及到的数据库资源。



### Statement

使用Connection类的createStatement方法创建一个Statement对象。

创建之后可使用Statement的execute，executeUpdate（insert、update和delete…）和executeQuery（会返回一个结果集ResultSet）方法执行**静态sql语句**。

Statement 是 Java 执行数据库操作的一个重要接口，用于在已经建立数据库连接的基础上，向数据库发送要执行的SQL语句。Statement对象，用于执行不带参数的简单SQL语句。用于执行静态 SQL 语句并返回它所生成结果的对象。

在默认情况下，同一时间每个 Statement 对象在只能打开一个 ResultSet 对象。

查询结束之后调用Statement的close方法关闭Statement。

### PreparedStatement（继承自Statement）

PreparedStatement 接口扩展了 Statement 接口，它让你用一个常用的 Statement 对象增加几个高级功能。

用？占据参数的位置，执行SQL语句时需要传递实际参数以代替？。

使用结束时也应该使用close方法关闭连接。

### CallableStatement（继承自PreparedStatement）

正如一个 Connection 对象可以创建 Statement 对象和 PreparedStatement 对象，它也可以创建被用来执行调用数据库存储过程的 CallableStatement 对象。用于执行对数据库已存在的存储过程的调用。

## JDBC连接池

在标准JDBC对应用的接口中，并没有提供资源的管理方法。所以，JDBC为第三方应用服务器（Application Server）提供了一个由数据库厂家实现的管理标准接口：连接缓冲(connection pooling)。引入了连接池( Connection Pool )的概念 ，也就是以**缓冲池**的机制管理数据库的资源。

JDBC最常用的资源有三类:

**Connection**: 数据库连接。

**Statement**: 会话声明。

**ResultSet**: 结果集游标。

数据库连接池的实现及原理

对Connection的管理，就是对数据库资源的管理。在关闭Connection前，需要关闭所有相关的Statement和ResultSet。因此，**连接池(Connection Pool)所起到的作用，不仅仅简单地管理Connection，还涉及到 Statement和ResultSet。**

## 数据库连接池的任务

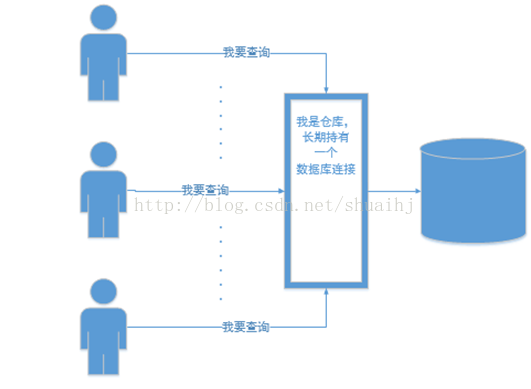
1. 连接池的第一个任务是**限制每个应用或系统可以拥有的最大资源**。也就是**确定连接池的大小**(PoolSize)。
2. 在连接池的大小(PoolSize)范围内，**最大限度地使用（复用）资源，缩短数据库访问的使用周期**。
3. 对资源的申请、释放、回收、共享和同步，这些管理是复杂精密的。所以，数据库连接池的另一个功能就是，封装这些操作，**为应用提供简单的，甚至是不改变应用风格的调用接口。**

## 原理

<http://blog.csdn.net/shuaihj/article/details/14223015>

### 分析

建立数据库连接是一个费时的活动，每次都得花费0.05s～1s的时间，而且系统还要分配内存资源。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，最终将不得不重启数据库。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，最终将不得不重启数据库。



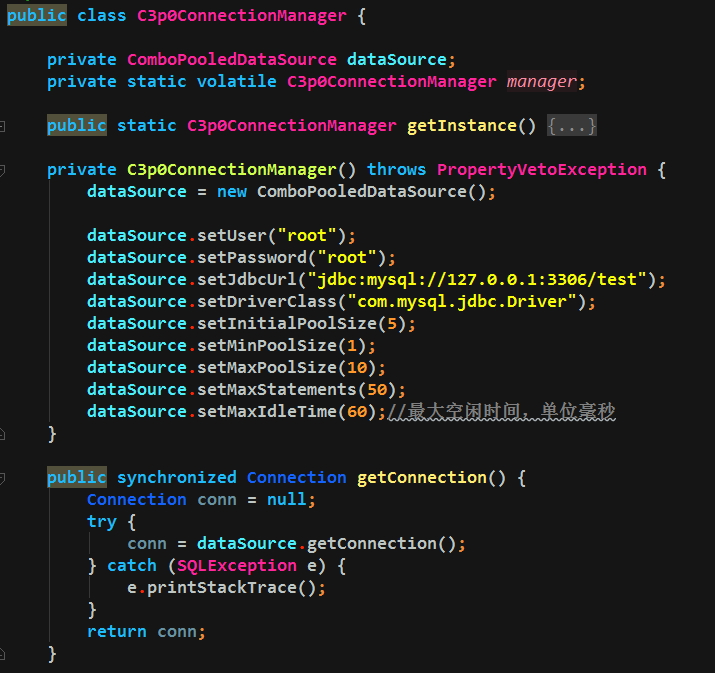
### 实现

利用**资源池**（resource pool）设计模式。该模式正是为了解决资源的频繁分配﹑释放所造成的问题。为解决上述问题，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。我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。更为重要的是我们可以通过连接池的管理机制监视数据库的连接的数量﹑使用情况，为系统开发﹑测试及性能调整提供依据。**

## C3P0

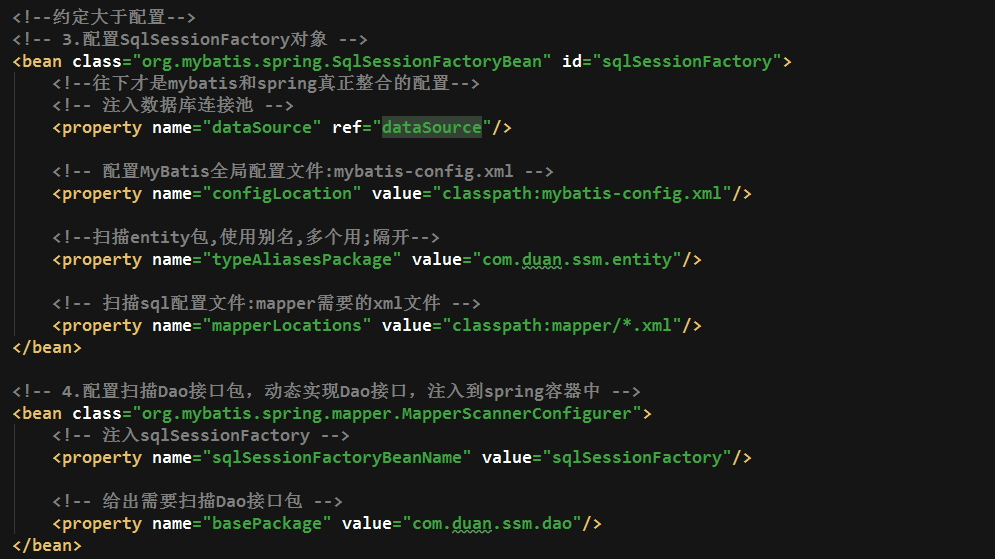
C3p0连接池特性：

编码简单易用、复用数据库连接、管理数据库连接。



### 用Spring进行配置





## Apache DBCP

## Alibaba-Druid

# 数据库事务4大特性

原子性、（前后）一致性、（并发）分离（隔离）性、持久性

## 1 原子性

事务的原子性指的是，事务中包含的程序作为数据库的逻辑工作单位，它所做的对数据修改操作**要么全部执行，要么完全不执行。这种特性称为原子性**。

事务的原子性要求，如果把一个事务可看作是一个程序，它要么完整的被执行，要么完全不执行。就是说事务的操纵序列或者完全应用到数据库或者完全不影响数据库。这种特性称为原子性。

假如用户在一个事务内完成了对数据库的更新，这时所有的更新对外部世界必须是可见的，或者完全没有更新。前者称事务已提交，后者称事务撤消（或流产）。DBMS必须确保由成功提交的事务完成的所有操纵在数据库内有完全的反映，而失败的事务对数据库完全没有影响。

## 2 （前后）一致性

事务的一致性指的是在**一个事务执行之前和执行之后数据库都必须处于一致性状态**。这种特性称为事务的一致性。假如数据库的状态满足所有的完整性约束，就说该数据库是一致的。

一致性处理数据库中对所有语义约束的保护。假如数据库的状态满足所有的完整性约束，就说该数据库是一致的。例如，当数据库处于一致性状态S1时，对数据库执行一个事务，在事务执行期间假定数据库的状态是不一致的，当事务执行结束时，数据库处在一致性状态S2。

## 3 （并发）分离（隔离）性

分离性指**并发的事务是相互隔离的。即一个事务内部的操作及正在操作的数据必须封锁起来，不被其它企图进行修改的事务看到**。

分离性是DBMS针对**并发事务间的冲突**提供的安全保证。DBMS可以通过**加锁**在并发执行的事务间提供不同级别的分离。假如并发交叉执行的事务没有任何控制，操纵相同的共享对象的多个并发事务的执行可能引起异常情况。

DBMS可以在并发执行的事务间提供不同级别的分离。分离的级别和并发事务的吞吐量之间存在反比关系（隔离级别越高，吞吐量越低）。较多事务的可分离性可能会带来较高的冲突和较多的事务流产。流产的事务要消耗资源，这些资源必须要重新被访问。因此，确保高分离级别的DBMS需要更多的开销。

## 4 持久性

持久性意味着**当系统或介质发生故障时，确保已提交事务的更新不能丢失**。即一旦一个事务提交，DBMS保证它对数据库中数据的改变应该是永久性的，耐得住任何系统故障。持久性通过**数据库备份**和**恢复**来保证。

持久性意味着当系统或介质发生故障时，确保已提交事务的更新不能丢失。即对已提交事务的更新能恢复。一旦一个事务被提交，DBMS必须保证提供适当的冗余，使其耐得住系统的故障。所以，持久性主要在于DBMS的恢复性能。

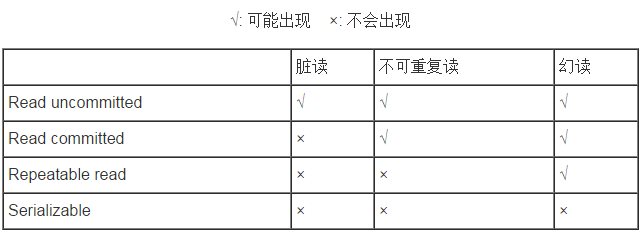
# 数据库事务4/7

<https://www.cnblogs.com/WJ-163/p/6023054.html>

## 四种隔离级别

数据库事务的隔离级别有4个，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决**脏读**、**不可重复读**、**幻读**这几类问题。

1. **脏读（读到脏数据）**：读取到了未提交的数据
2. **不可重复读（重复读时数据不一致）**：意味着我们在同一个事务中执行完全相同的select语句时可能看到不一样的结果
3. **幻读（同一范围数据行变动）**：当用户读取某一范围的**数据行**时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行



### ISOLATION\_READ\_UNCOMMITTED（会读取到未提交内容）

这是事务最低的隔离级别，它**充许另外一个事务可以看到这个事务未提交的数据**。这种隔离级别会产生脏读，不可重复读和幻像读。

本隔离级别很少用于实际应用，而且它的性能也不比其他级别好多少。

### ISOLATION\_READ\_COMMITTED（只能读取提交后的内容）

**保证一个事务修改的数据提交后才能被另外一个事务读取**。另外一个事务不能读取该事务未提交的数据。

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。

满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。

这种隔离级别出现的问题是：不可重复读。导致这种情况的原因可能有：

1. 有一个交叉的事务有新的commit，导致了数据的改变;
2. 一个数据库被多个实例操作时,同一事务的其他实例在该实例处理期间可能会有新的commit

### ISOLATION\_REPEATABLE\_READ（可以重读）

这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了**保证一个事务不能读取另一个事务未提交的数据外**，同时保证了重复读时获取到的数据是一致的。

这是**MySQL的默认事务隔离级别**，它确保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。

此级别可能出现的问题：幻读。InnoDB和Falcon存储引擎通过**多版本并发控制**解决了该问题

### ISOLATION\_SERIALIZABLE（可串行化）

这是花费**最高代价**但是**最可靠**的事务隔离级别。事务被处理为**顺序执行**。除了防止脏读，不可重复读外，还避免了幻像读。

它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之,它是在每个读的数据行上加上**共享锁**。

在这个级别，可能**导致大量的超时现象和锁竞争。**

## 七种传播行为

### PROPAGATION\_REQUIRED

**如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务**，该设置是最常用的设置。

### PROPAGATION\_SUPPORTS

支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就**以非事务执行**。

### PROPAGATION\_MANDATORY

支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就**抛出异常**。

### PROPAGATION\_REQUIRES\_NEW

创建新事务，无论当前存不存在事务，**都创建新事务**。

### PROPAGATION\_NOT\_SUPPORTED

以**非事务方式执行**操作，如果当前存在事务，就**把当前事务挂起**。

### PROPAGATION\_NEVER

以**非事务方式执行**，如果当前存在事务，则**抛出异常**。

### PROPAGATION\_NESTED

如果当前存在事务，则在**嵌套事务内执行**。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

# 数据库三大范式

为了建立冗余较小、结构合理的数据库，设计数据库时必须遵循一定的规则。在关系型数据库中这种规则就称为范式。范式是符合某一种设计要求的总结。要想设计一个结构合理的关系型数据库，必须满足一定的范式。

## 一 确保每列保持原子性

第一范式是**最基本的范式**。如果数据库表中的**所有字段值都是不可分解的原子值**，就说明该数据库表满足了第一范式。

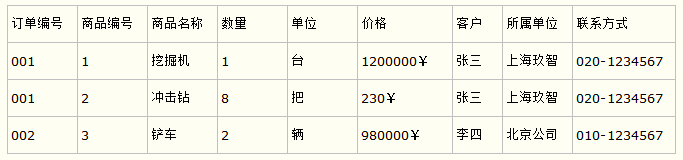
第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性，本来直接将“地址”属性设计成一个数据库表的字段就行。但是如果系统经常会访问“地址”属性中的“城市”部分，那么就非要将“地址”这个属性重新拆分为省份、城市、详细地址等多个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式，如下表所示。



## 二 保表中的每列都和主键相关

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说**在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中**。

比如要设计一个订单信息表，因为订单中可能会有多种商品，所以要将订单编号和商品编号作为数据库表的联合主键，如下表所示。



这样就产生一个问题：这个表中是以订单编号和商品编号作为联合主键。这样在该表中商品名称、单位、商品价格等信息不与该表的主键相关，而仅仅是与商品编号相关。所以在这里违反了第二范式的设计原则。

而如果把这个订单信息表进行拆分，把商品信息分离到另一个表中，把订单项目表也分离到另一个表中，就非常完美了。如下所示。



这样设计，在很大程度上减小了数据库的冗余。如果要获取订单的商品信息，使用商品编号到商品信息表中查询即可。

## 三 保每列都和主键列直接相关,而不是间接相关

第三范式需要确保**数据表中的每一列数据都和主键直接相关，而不能间接相关。**

比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。如下面这两个表所示的设计就是一个满足第三范式的数据库表。

这样在查询订单信息的时候，就可以使用客户编号来引用客户信息表中的记录，也不必在订单信息表中多次输入客户信息的内容，减小了数据冗余。

