数据库连接池

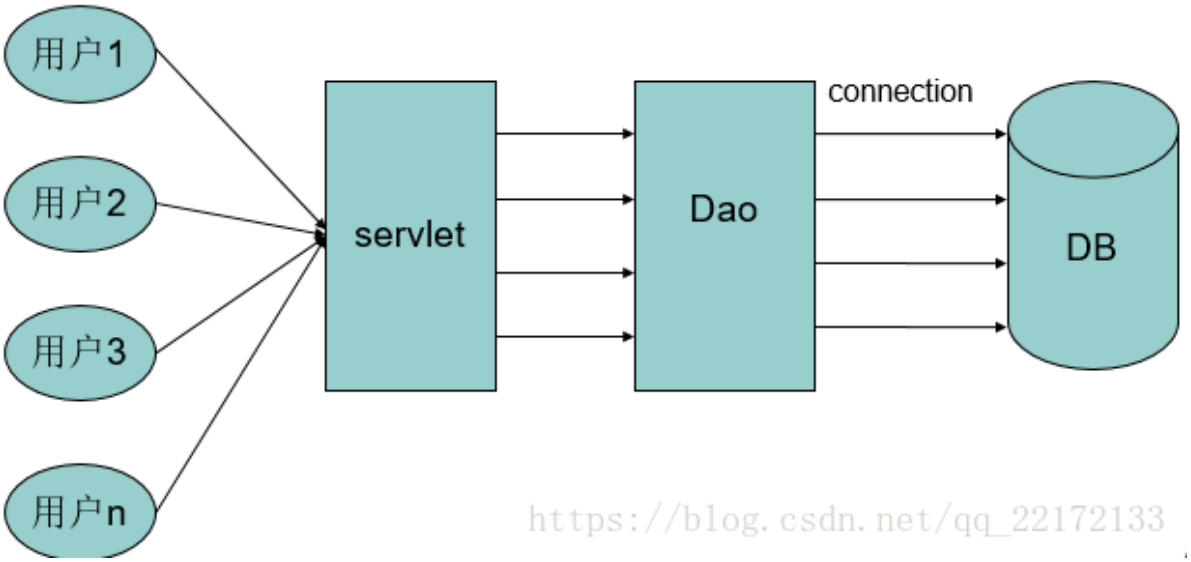
整个数据源最核心的三个东西：

一个是连接池，在这里体现为common-pool中的GenericObjectPool，**它负责缓存和管理连接，所有的配置策略都是由它管理。**

第二个是连接，这里的连接就是PoolableConnection，当然它是对底层连接进行了封装。

第三个则是连接池和连接的关系，在此表现为一对多的互相引用。

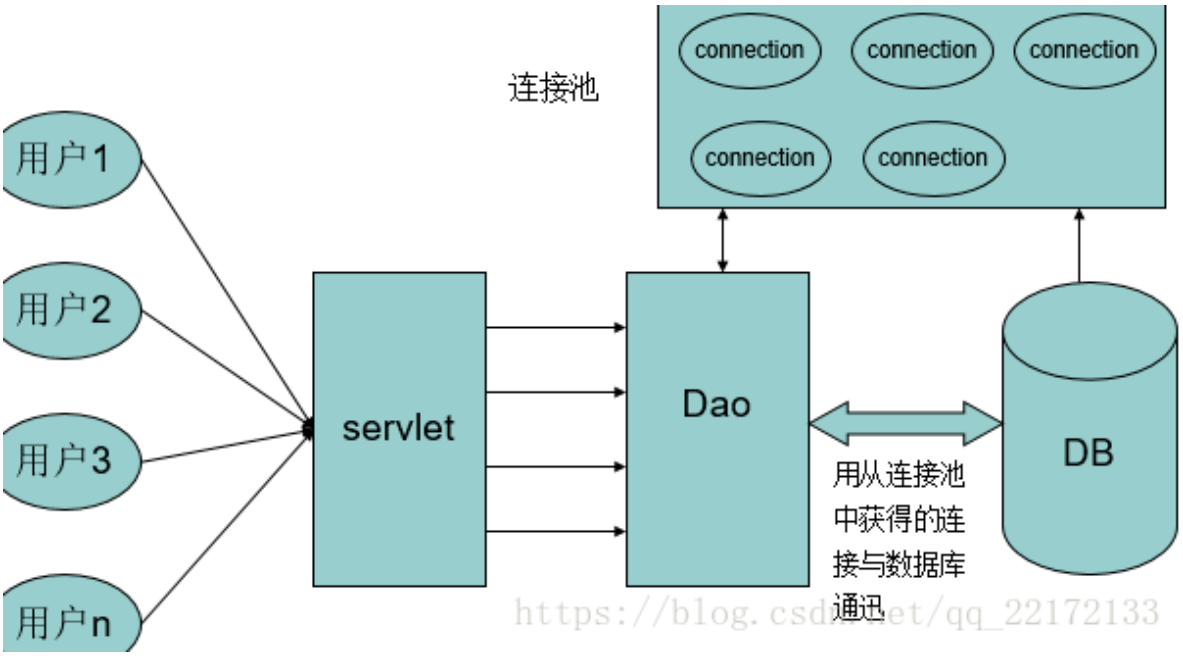
传统连接方法



缺点：

用户每次请求都需要向数据库获得链接，而数据库创建连接通常需要消耗相对较大的资源，创建时间也较长。假设网站一天10万访问量，数据库服务器就需要创建10万次连接，极大的浪费数据库的资源，并且极易造成数据库服务器内存溢出、拓机。

数据库连接池



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，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。

最小连接数：是连接池一直保持的数据库连接，所以如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费。

最大连接数：是连接池能申请的最大连接数，如果数据库连接请求超过次数，后面的数据库连接请求将被加入到等待队列中，这会影响以后的数据库操作

如果最小连接数与最大连接数相差很大：那么最先连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接. 不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，他将被放到连接池中等待重复使用或是空间超时后被释放。

编写连接池

编写连接池需实现java.sql.DataSource接口。DataSource接口中定义了两个重载的getConnection方法：

- Connection getConnection()

- Connection getConnection(String username, String password)

实现DataSource接口，并实现连接池功能的步骤：

1. 在DataSource构造函数中批量创建与数据库的连接，并把创建的连接加入LinkedList对象中。
2. 实现getConnection方法，让getConnection方法每次调用时，从LinkedList中取一个Connection返回给用户。
3. 当用户使用完Connection，调用Connection.close()方法时，Connection对象应保证将自己返回到LinkedList中,而不要把conn还给数据库。

核心代码：

```
proxyConn = (Connection) Proxy.newProxyInstance(this.getClass().getClassLoader(),
conn.getClass().getInterfaces(), new InvocationHandler() {
         //此处为内部类，当close方法被调用时将conn还回池中,其它方法直接执行
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable{
        if (method.getName().equals("close")) {
            pool.addLast(conn);
            return null;
        }
    return method.invoke(conn, args);
  }
});
```

开源数据库连接池

DBCP数据源

Apache 软件基金组织下的开源连接池实现，需要的 jar 文件：

- Commons-dbcp.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。

dbcpconfig.properties配置：

```
#连接设置
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/jdbcstudy
username=root
password=XDP
#<!-- 初始化连接 -->
initialSize=10
#最大连接数量
maxActive=50
#<!-- 最大空闲连接 -->
maxIdle=20
#<!-- 最小空闲连接 -->
minIdle=5
#<!-- 超时等待时间以毫秒为单位 6000毫秒/1000等于60秒 -->
maxWait=60000
#JDBC驱动建立连接时附带的连接属性属性的格式必须为这样：[属性名=property;]
#注意："user" 与 "password" 两个属性会被明确地传递，因此这里不需要包含他们。
connectionProperties=useUnicode=true;characterEncoding=UTF8
#指定由连接池所创建的连接的自动提交（auto-commit）状态。
defaultAutoCommit=true
#driver default 指定由连接池所创建的连接的只读（read-only）状态。
#如果没有设置该值，则“setReadOnly”方法将不被调用。（某些驱动并不支持只读模式，如：Informix）
defaultReadOnly=
#driver default 指定由连接池所创建的连接的事务级别（TransactionIsolation）。
```

#可用值为下列之一：（详情可见javadoc。）NONE,READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE

defaultTransactionIsolation=READ_UNCOMMITTED

在获取数据库连接的工具类(如jdbcUtils)的静态代码块中创建池：

```
public class JdbcUtils_DBCP {
	private static DataSource ds = null;
	static{
		try{
			InputStream in = 
JdbcUtils_DBCP.class.getClassLoader().getResourceAsStream("dbcpconfig.properties");
			Properties prop = new Properties();
			prop.load(in);
			ds = BasicDataSourceFactory.createDataSource(prop);
		}catch (Exception e) {
			throw new ExceptionInInitializerError(e);
		}
	}

	public static Connection getConnection() throws SQLException{
	 //获取连接
		return ds.getConnection();
	}

	public static void release(Connection conn,Statement st,ResultSet rs){
		//释放资源
		if(rs!=null){
			try{
				rs.close();
			}catch (Exception e) {
				e.printStackTrace();
			}
			rs = null;
		}
		if(st!=null){
			try{
```

```
                st.close();
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
        if(conn!=null){
            try{
                conn.close();
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

C3P0数据源

实现了数据源和JNDI绑定,使用它的开源项目有***Hibernate，Spring***等。相较于DBCP，C3P0有自动回收空闲连接功能。需要的jar包：c3p0-0.9.2-pre1.jar、mchange-commons-0.2.jar，如果操作的是Oracle数据库，那么还需要导入c3p0-oracle-thin-extras-0.9.2-pre1.jar。

在类目录下加入C3P0的配置文件：c3p0-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c3p0-config>
    <!--C3P0的缺省(默认)配置，
    如果在代码中"ComboPooledDataSource ds = new ComboPooledDataSource();"这样写就表示
    使用的是C3P0的缺省(默认)配置信息来创建数据源
    -->
        <default-config>
            <property name="driverClass">com.mysql.jdbc.Driver</property>
            <property
name="jdbcUrl">jdbc:mysql://localhost:3306/day16</property>
            <property name="user">root</property>
            <property name="password">root</property>

            <property name="acquireIncrement">5</property>
```

```
                <property name="initialPoolSize">10</property>
                <property name="minPoolSize">5</property>
                <property name="maxPoolSize">20</property>
        </default-config>
      <!--C3P0的命名配置，
    如果在代码中"ComboPooledDataSource ds = new ComboPooledDataSource("MySQL");"这样
    写就表示使用的是name是MySQL的配置信息来创建数据源
    -->
        <named-config name="mysql">
                <property name="driverClass">com.mysql.jdbc.Driver</property>
                <property
name="jdbcUrl">jdbc:mysql://localhost:3306/day16</property>
                <property name="user">root</property>
                <property name="password">root</property>

                <property name="acquireIncrement">5</property>
                <property name="initialPoolSize">10</property>
                <property name="minPoolSize">5</property>
                <property name="maxPoolSize">20</property>
        </named-config>

        <named-config name="oracle">
                <property name="driverClass">com.mysql.jdbc.Driver</property>
                <property
name="jdbcUrl">jdbc:mysql://localhost:3306/day16</property>
                <property name="user">root</property>
                <property name="password">root</property>

                <property name="acquireIncrement">5</property>
                <property name="initialPoolSize">10</property>
                <property name="minPoolSize">5</property>
                <property name="maxPoolSize">20</property>
        </named-config>
</c3p0-config>
```

创建数据源：

```
private static ComboPooledDataSource ds;
static{
    try {
        ds = new ComboPooledDataSource("MySQL");
    } catch (Exception e) {
        throw new ExceptionInInitializerError(e);
    }
}
```

配置Tomcat数据源

JNDI技术

Java命名和目录接口，对应于J2SE中的javax.naming包。这套API的主要作用在于：它可以把Java对象放在一个容器中（JNDI容器），并为容器中的java对象取一个名称，以后程序想获得Java对象，只需 通过名称检索即可。其核心API为Context，它代表JNDI容器，其lookup方法为检索容器中对应名称的对象。

配置tomcat服务器的数据源（在Web项目的WebRoot目录下的META-INF目录创建一个context.xml文件）：

```
<Context>
  <Resource name="jdbc/datasource" auth="Container"
            type="javax.sql.DataSource" username="root"
password="XDP"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/jdbcstudy"
            maxActive="8" maxIdle="4"/>
</Context>
```

服务器创建的这些资源有两种方式提供给我们的应用程序使用：

第一种是通过方法参数的形式传递进来，比如我们在Servlet中写的doPost和doGet方法中使用到的request对象和response对象就是服务器以参数的形式传递给我们的。

第二种就是JNDI的方式，服务器把创建好的资源绑定到JNDI容器中去，应用程序想要使用资源时，就直接从JNDI容器中获取相应的资源即可。

在应用程序中可以用代码去获取数据源：

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
dataSource = (DataSource)envCtx.lookup("jdbc/datasource");
```