**数据库连接池总结**

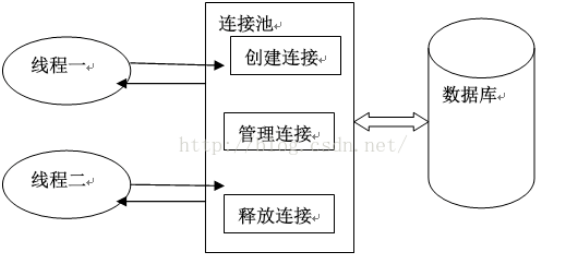
## 一、应用程序直接获取数据库连接的缺点

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**。

## 二、使用数据库连接池优化程序性能

**1 数据库连接是一种关键的有限的昂贵的资源**

这一点在多用户的网页应用程序中体现的尤为突出.对数据库连接的管理能显著影响到整个应用程序的伸缩性和健壮性,影响到程序的性能指标.数据库连接池正式针对这个问题提出来的.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。如下图所示：



**2 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。个人理解：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。

**3 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.

**4 数据库连接池的最小连接数和最大连接数的设置要考虑到以下几个因素:**

**@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。

**5 数据库连接池的运行机制**

（1） 程序初始化时创建连接池

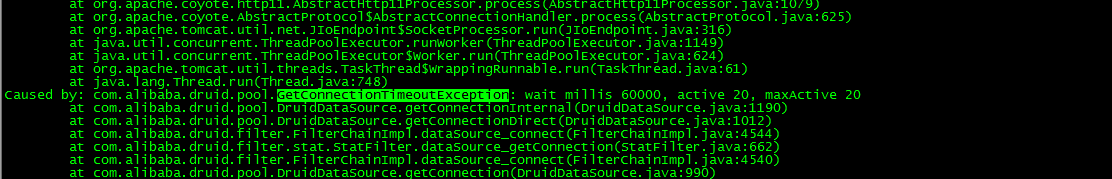
（2） 使用时向连接池申请可用连接

（3） 使用完毕，将连接返还给连接池

（4） 程序退出时，断开所有连接，并释放资源

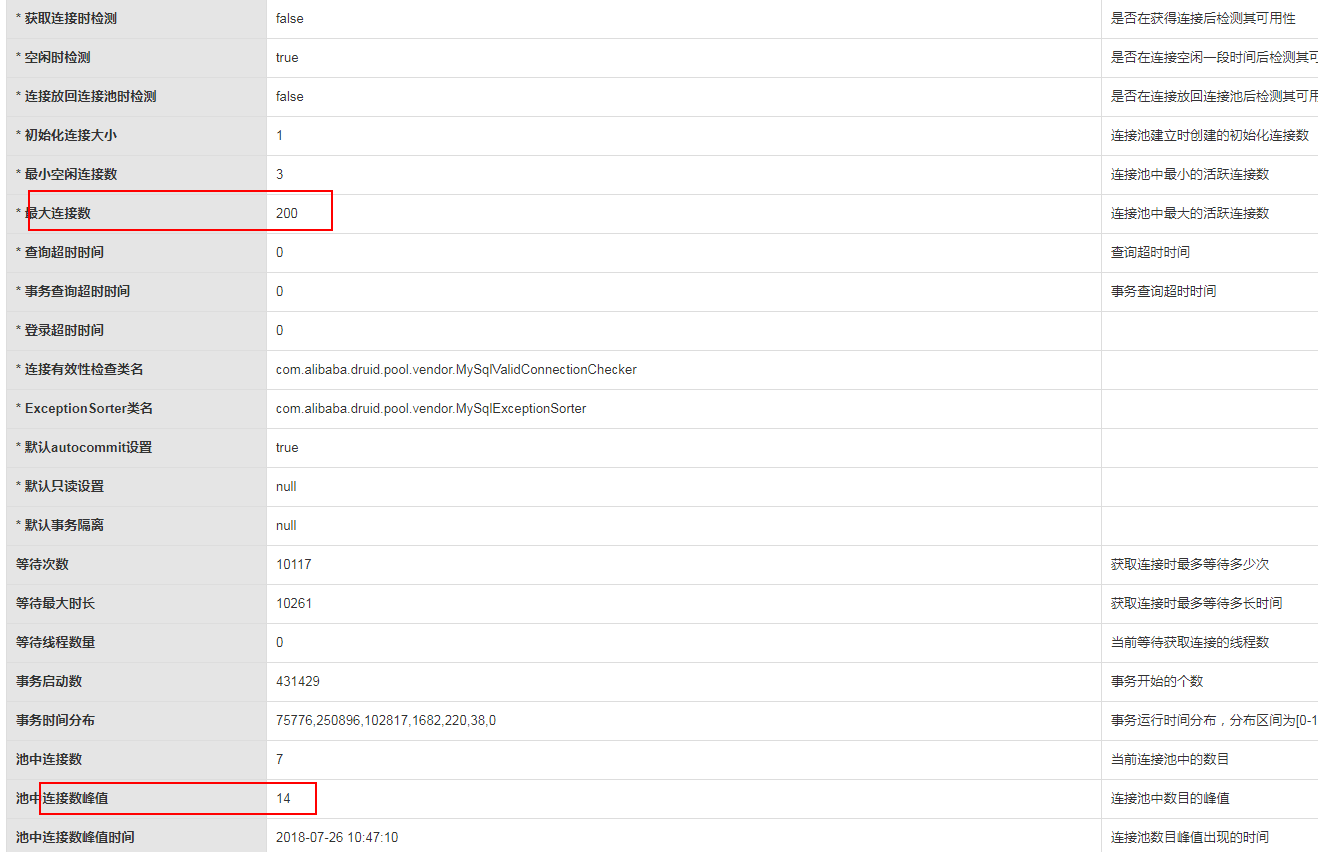
**6 异常分析**

**问题异常**

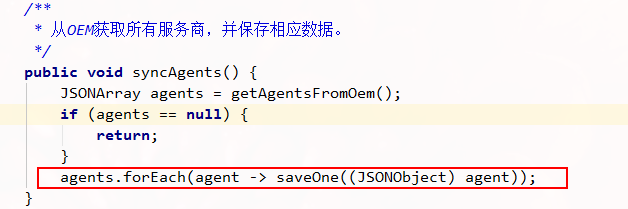


**监控**

<http://oc.yunnex.com/druid/datasource.html>



**问题原因：**

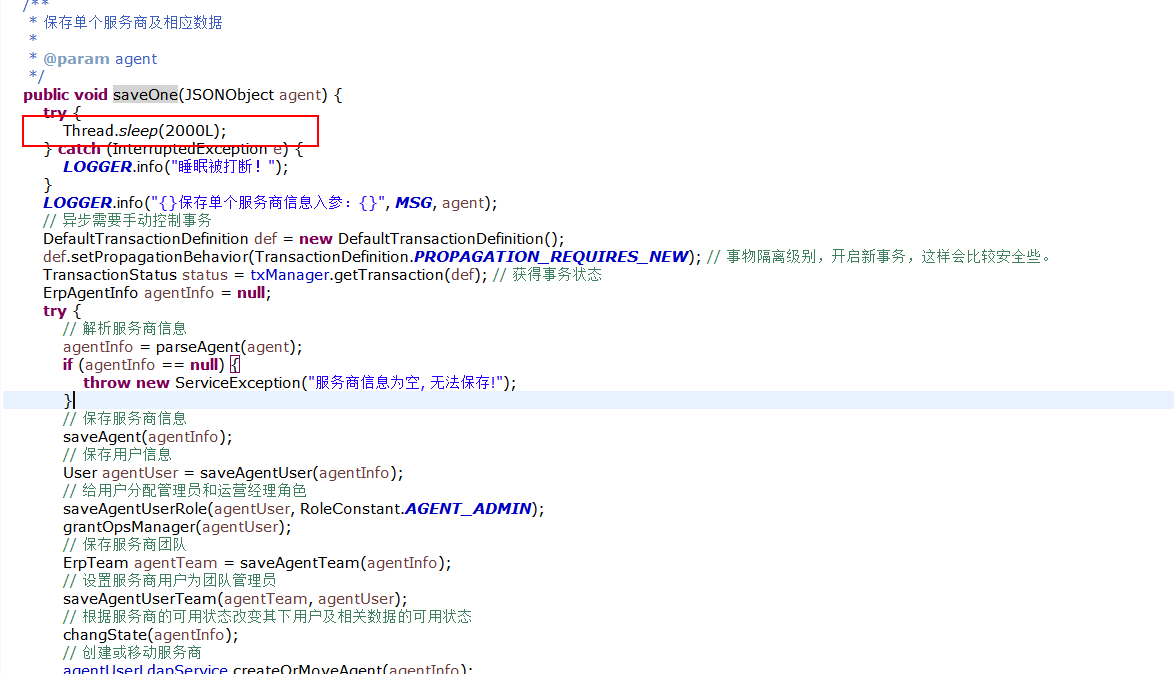


同步服务商数据，每次循环有20个数据库操作，由于是同步操作，导致1次循环下来所有的数据库连接还没有完全释放，以此类推，未释放的链接越开越多，最后冲破最大值，获取不到连接了

**解决措施：**

（1）加大数据库连接池到200

（2）线程睡眠



## 三、编写数据库连接池

**1 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：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。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。当用户使用完Connection，调用Connection.close()方法时，Collection对象应保证将自己返回到LinkedList中,而不要把conn还给数据库。Collection保证将自己返回到LinkedList中是此处编程的难点。

**2 数据库连接池核心代码**

使用动态代理技术构建连接池中的connection

1 proxyConn = (Connection) Proxy.newProxyInstance(this.getClass()

2 .getClassLoader(), conn.getClass().getInterfaces(),

3 new InvocationHandler() {

4 //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

5 public Object invoke(Object proxy, Method method,

6 Object[] args) throws Throwable {

7 if (method.getName().equals("close")) {

8 pool.addLast(conn);

9 return null;

10 }

11 return method.invoke(conn, args);

12 }

13 });

**3 数据库连接池编写范例：**

1 package me.gacl.demo;

3 import java.io.InputStream;

4 import java.io.PrintWriter;

5 import java.lang.reflect.InvocationHandler;

6 import java.lang.reflect.Method;

7 import java.lang.reflect.Proxy;

8 import java.sql.Connection;

9 import java.sql.DriverManager;

10 import java.sql.SQLException;

11 import java.util.LinkedList;

12 import java.util.Properties;

13 import javax.sql.DataSource;

15 /\*\*

16 \* @ClassName: JdbcPool

17 \* @Description:编写数据库连接池

18 \* @author: 孤傲苍狼

19 \* @date: 2014-9-30 下午11:07:23

20 \*

21 \*/

22 public class JdbcPool implements DataSource{

23

24 /\*\*

25 \* @Field: listConnections

26 \* 使用LinkedList集合来存放数据库链接，

27 \* 由于要频繁读写List集合，所以这里使用LinkedList存储数据库连接比较合适

28 \*/

29 private static LinkedList<Connection> listConnections = new LinkedList<Connection>();

30

31 static{

32 //在静态代码块中加载db.properties数据库配置文件

33 InputStream in = JdbcPool.class.getClassLoader().getResourceAsStream("db.properties");

34 Properties prop = new Properties();

35 try {

36 prop.load(in);

37 String driver = prop.getProperty("driver");

38 String url = prop.getProperty("url");

39 String username = prop.getProperty("username");

40 String password = prop.getProperty("password");

41 //数据库连接池的初始化连接数大小

42 int jdbcPoolInitSize =Integer.parseInt(prop.getProperty("jdbcPoolInitSize"));

43 //加载数据库驱动

44 Class.forName(driver);

45 for (int i = 0; i < jdbcPoolInitSize; i++) {

46 Connection conn = DriverManager.getConnection(url, username, password);

47 System.out.println("获取到了链接" + conn);

48 //将获取到的数据库连接加入到listConnections集合中，listConnections集合此时就是一个存放了数据库连接的连接池

49 listConnections.add(conn);

50 }

51

52 } catch (Exception e) {

53 throw new ExceptionInInitializerError(e);

54 }

55 }

56

57 @Override

58 public PrintWriter getLogWriter() throws SQLException {

59 // TODO Auto-generated method stub

60 return null;

61 }

62

63 @Override

64 public void setLogWriter(PrintWriter out) throws SQLException {

65 // TODO Auto-generated method stub

66

67 }

68

69 @Override

70 public void setLoginTimeout(int seconds) throws SQLException {

71 // TODO Auto-generated method stub

72

73 }

74

75 @Override

76 public int getLoginTimeout() throws SQLException {

77 // TODO Auto-generated method stub

78 return 0;

79 }

80

81 @Override

82 public <T> T unwrap(Class<T> iface) throws SQLException {

83 // TODO Auto-generated method stub

84 return null;

85 }

86

87 @Override

88 public boolean isWrapperFor(Class<?> iface) throws SQLException {

89 // TODO Auto-generated method stub

90 return false;

91 }

92

93 /\* 获取数据库连接

94 \* @see javax.sql.DataSource#getConnection()

95 \*/

96 @Override

97 public Connection getConnection() throws SQLException {

98 //如果数据库连接池中的连接对象的个数大于0

99 if (listConnections.size()>0) {

100 //从listConnections集合中获取一个数据库连接

101 final Connection conn = listConnections.removeFirst();

102 System.out.println("listConnections数据库连接池大小是" + listConnections.size());

103 //返回Connection对象的代理对象

104 return (Connection) Proxy.newProxyInstance(JdbcPool.class.getClassLoader(), conn.getClass().getInterfaces(), new InvocationHandler(){

105 @Override

106 public Object invoke(Object proxy, Method method, Object[] args)

107 throws Throwable {

108 if(!method.getName().equals("close")){

109 return method.invoke(conn, args);

110 }else{

111 //如果调用的是Connection对象的close方法，就把conn还给数据库连接池

112 listConnections.add(conn);

113 System.out.println(conn + "被还给listConnections数据库连接池了！！");

114 System.out.println("listConnections数据库连接池大小为" + listConnections.size());

115 return null;

116 }

117 }

118 });

119 }else {

120 throw new RuntimeException("对不起，数据库忙");

121 }

122 }

123

124 @Override

125 public Connection getConnection(String username, String password)

126 throws SQLException {

127 return null;

128 }

129 }

**4 db.properties配置文件如下：**

1 driver=com.mysql.jdbc.Driver

2 url=jdbc:mysql://localhost:3306/jdbcStudy

3 username=root

4 password=XDP

5

6 jdbcPoolInitSize=10

**5 写一个JdbcUtil测试数据库连接池**

1 package me.gacl.utils;

3 import java.sql.Connection;

4 import java.sql.ResultSet;

5 import java.sql.SQLException;

6 import java.sql.Statement;

7 import me.gacl.demo.JdbcPool;

9 public class JdbcUtil {

10

11 /\*\*

12 \* @Field: pool

13 \* 数据库连接池

14 \*/

15 private static JdbcPool pool = new JdbcPool();

16

17 /\*\*

18 \* @Method: getConnection

19 \* @Description: 从数据库连接池中获取数据库连接对象

20 \* @Anthor:孤傲苍狼

21 \* @return Connection数据库连接对象

22 \* @throws SQLException

23 \*/

24 public static Connection getConnection() throws SQLException{

25 return pool.getConnection();

26 }

27

28 /\*\*

29 \* @Method: release

30 \* @Description: 释放资源，

31 \* 释放的资源包括Connection数据库连接对象，负责执行SQL命令的Statement对象，存储查询结果的ResultSet对象

32 \* @Anthor:孤傲苍狼

33 \*

34 \* @param conn

35 \* @param st

36 \* @param rs

37 \*/

38 public static void release(Connection conn,Statement st,ResultSet rs){

39 if(rs!=null){

40 try{

41 //关闭存储查询结果的ResultSet对象

42 rs.close();

43 }catch (Exception e) {

44 e.printStackTrace();

45 }

46 rs = null;

47 }

48 if(st!=null){

49 try{

50 //关闭负责执行SQL命令的Statement对象

51 st.close();

52 }catch (Exception e) {

53 e.printStackTrace();

54 }

55 }

56

57 if(conn!=null){

58 try{

59 //关闭Connection数据库连接对象

60 conn.close();

61 }catch (Exception e) {

62 e.printStackTrace();

63 }

64 }

65 }

66 }

## 四、开源数据库连接池

现在很多WEB服务器(Weblogic, WebSphere, Tomcat)都提供了DataSoruce的实现，即连接池的实现。通常我们把DataSource的实现，按其英文含义称之为数据源，数据源中都包含了数据库连接池的实现。也有一些开源组织提供了数据源的独立实现：

**1、DBCP数据源**

　　DBCP 是 Apache 软件基金组织下的开源连接池实现，要使用DBCP数据源，需要应用程序应在系统中增加如下两个 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。

**2、在应用程序中加入dbcp连接池**

　　1.导入相关jar包commons-dbcp-1.2.2.jar、commons-pool.jar

　　2、在类目录下加入dbcp的配置文件：dbcpconfig.properties

1 #连接设置

2 driverClassName=com.mysql.jdbc.Driver

3 url=jdbc:mysql://localhost:3306/jdbcstudy

4 username=root

5 password=XDP

6

7 #<!-- 初始化连接 -->

8 initialSize=10

9

10 #最大连接数量

11 maxActive=50

12

13 #<!-- 最大空闲连接 -->

14 maxIdle=20

15

16 #<!-- 最小空闲连接 -->

17 minIdle=5

18

19 #<!-- 超时等待时间以毫秒为单位 6000毫秒/1000等于60秒 -->

20 maxWait=60000

21

22

23 #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;]

24 #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。

25 connectionProperties=useUnicode=true;characterEncoding=UTF8

26

27 #指定由连接池所创建的连接的自动提交（auto-commit）状态。

28 defaultAutoCommit=true

29

30 #driver default 指定由连接池所创建的连接的只读（read-only）状态。

31 #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）

32 defaultReadOnly=

33

34 #driver default 指定由连接池所创建的连接的事务级别（TransactionIsolation）。

35 #可用值为下列之一：（详情可见javadoc。）NONE,READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE

36 defaultTransactionIsolation=READ\_UNCOMMITTED

**3、在获取数据库连接的工具类(如jdbcUtils)的静态代码块中创建池**

1 package me.gacl.util;

2

3 import java.io.InputStream;

4 import java.sql.Connection;

5 import java.sql.ResultSet;

6 import java.sql.SQLException;

7 import java.sql.Statement;

8 import java.util.Properties;

9 import javax.sql.DataSource;

10 import org.apache.commons.dbcp.BasicDataSourceFactory;

11

12 /\*\*

13 \* @ClassName: JdbcUtils\_DBCP

14 \* @Description: 数据库连接工具类

15 \* @author: 孤傲苍狼

16 \* @date: 2014-10-4 下午6:04:36

17 \*

18 \*/

19 public class JdbcUtils\_DBCP {

20 /\*\*

21 \* 在java中，编写数据库连接池需实现java.sql.DataSource接口，每一种数据库连接池都是DataSource接口的实现

22 \* DBCP连接池就是java.sql.DataSource接口的一个具体实现

23 \*/

24 private static DataSource ds = null;

25 //在静态代码块中创建数据库连接池

26 static{

27 try{

28 //加载dbcpconfig.properties配置文件

29 InputStream in = JdbcUtils\_DBCP.class.getClassLoader().getResourceAsStream("dbcpconfig.properties");

30 Properties prop = new Properties();

31 prop.load(in);

32 //创建数据源

33 ds = BasicDataSourceFactory.createDataSource(prop);

34 }catch (Exception e) {

35 throw new ExceptionInInitializerError(e);

36 }

37 }

38

39 /\*\*

40 \* @Method: getConnection

41 \* @Description: 从数据源中获取数据库连接

42 \* @Anthor:孤傲苍狼

43 \* @return Connection

44 \* @throws SQLException

45 \*/

46 public static Connection getConnection() throws SQLException{

47 //从数据源中获取数据库连接

48 return ds.getConnection();

49 }

50

51 /\*\*

52 \* @Method: release

53 \* @Description: 释放资源，

54 \* 释放的资源包括Connection数据库连接对象，负责执行SQL命令的Statement对象，存储查询结果的ResultSet对象

55 \* @Anthor:孤傲苍狼

56 \*

57 \* @param conn

58 \* @param st

59 \* @param rs

60 \*/

61 public static void release(Connection conn,Statement st,ResultSet rs){

62 if(rs!=null){

63 try{

64 //关闭存储查询结果的ResultSet对象

65 rs.close();

66 }catch (Exception e) {

67 e.printStackTrace();

68 }

69 rs = null;

70 }

71 if(st!=null){

72 try{

73 //关闭负责执行SQL命令的Statement对象

74 st.close();

75 }catch (Exception e) {

76 e.printStackTrace();

77 }

78 }

79

80 if(conn!=null){

81 try{

82 //将Connection连接对象还给数据库连接池

83 conn.close();

84 }catch (Exception e) {

85 e.printStackTrace();

86 }

87 }

88 }

89 }

**4 测试DBCP数据源**

1 package me.gacl.test;

3 import java.sql.Connection;

4 import java.sql.PreparedStatement;

5 import java.sql.ResultSet;

6 import org.junit.Test;

7 import me.gacl.util.JdbcUtils\_DBCP;

8

9 public class DataSourceTest {

10

11 @Test

12 public void dbcpDataSourceTest() {

13 Connection conn = null;

14 PreparedStatement st = null;

15 ResultSet rs = null;

16 try{

17 //获取数据库连接

18 conn = JdbcUtils\_DBCP.getConnection();

19 String sql = "insert into test1(name) values(?)";

20 st = conn.prepareStatement(sql);

21 st.setString(1, "gacl");

22 st.executeUpdate();

23 //获取数据库自动生成的主键

24 rs = st.getGeneratedKeys();

25 if(rs.next()){

26 System.out.println(rs.getInt(1));

27 }

28 }catch (Exception e) {

29 e.printStackTrace();

30 }finally{

31 //释放资源

32 JdbcUtils\_DBCP.release(conn, st, rs);

33 }

34 }

35 }

**5、C3P0数据源**

C3P0是一个开源的JDBC连接池，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。目前使用它的开源项目有Hibernate，Spring等。C3P0数据源在项目开发中使用得比较多。

**c3p0与dbcp区别：**

dbcp没有自动回收空闲连接的功能

c3p0有自动回收空闲连接功能

**6、在应用程序中加入C3P0连接池**

（1）.导入相关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

（2）、在类目录下加入C3P0的配置文件：c3p0-config.xml

1 <?xml version="1.0" encoding="UTF-8"?>

2 <!--

3 c3p0-config.xml必须位于类路径下面

4 private static ComboPooledDataSource ds;

5 static{

6 try {

7 ds = new ComboPooledDataSource("MySQL");

8 } catch (Exception e) {

9 throw new ExceptionInInitializerError(e);

10 }

11 }

12 -->

13

14 <c3p0-config>

15 <!--

16 C3P0的缺省(默认)配置，

17 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

18 -->

19 <default-config>

20 <property name="driverClass">com.mysql.jdbc.Driver</property>

21 <property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbcstudy</property>

22 <property name="user">root</property>

23 <property name="password">XDP</property>

24

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

26 <property name="initialPoolSize">10</property>

27 <property name="minPoolSize">5</property>

28 <property name="maxPoolSize">20</property>

29 </default-config>

30

31 <!--

32 C3P0的命名配置，

33 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

34 -->

35 <named-config name="MySQL">

36 <property name="driverClass">com.mysql.jdbc.Driver</property>

37 <property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbcstudy</property>

38 <property name="user">root</property>

39 <property name="password">XDP</property>

40

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

42 <property name="initialPoolSize">10</property>

43 <property name="minPoolSize">5</property>

44 <property name="maxPoolSize">20</property>

45 </named-config>

46

47 </c3p0-config>

**7 、在获取数据库连接的工具类(如jdbcUtils)的静态代码块中创建池**

1 package me.gacl.util;

2

3 import java.sql.Connection;

4 import java.sql.ResultSet;

5 import java.sql.SQLException;

6 import java.sql.Statement;

7 import com.mchange.v2.c3p0.ComboPooledDataSource;

8

9 /\*\*

10 \* @ClassName: JdbcUtils\_C3P0

11 \* @Description: 数据库连接工具类

12 \* @author: 孤傲苍狼

13 \* @date: 2014-10-4 下午6:04:36

14 \*

15 \*/

16 public class JdbcUtils\_C3P0 {

17

18 private static ComboPooledDataSource ds = null;

19 //在静态代码块中创建数据库连接池

20 static{

21 try{

22 //通过代码创建C3P0数据库连接池

23 /\*ds = new ComboPooledDataSource();

24 ds.setDriverClass("com.mysql.jdbc.Driver");

25 ds.setJdbcUrl("jdbc:mysql://localhost:3306/jdbcstudy");

26 ds.setUser("root");

27 ds.setPassword("XDP");

28 ds.setInitialPoolSize(10);

29 ds.setMinPoolSize(5);

30 ds.setMaxPoolSize(20);\*/

31

32 //通过读取C3P0的xml配置文件创建数据源，C3P0的xml配置文件c3p0-config.xml必须放在src目录下

33 //ds = new ComboPooledDataSource();//使用C3P0的默认配置来创建数据源

34 ds = new ComboPooledDataSource("MySQL");//使用C3P0的命名配置来创建数据源

35

36 }catch (Exception e) {

37 throw new ExceptionInInitializerError(e);

38 }

39 }

40

41 /\*\*

42 \* @Method: getConnection

43 \* @Description: 从数据源中获取数据库连接

44 \* @Anthor:孤傲苍狼

45 \* @return Connection

46 \* @throws SQLException

47 \*/

48 public static Connection getConnection() throws SQLException{

49 //从数据源中获取数据库连接

50 return ds.getConnection();

51 }

52

53 /\*\*

54 \* @Method: release

55 \* @Description: 释放资源，

56 \* 释放的资源包括Connection数据库连接对象，负责执行SQL命令的Statement对象，存储查询结果的ResultSet对象

57 \* @Anthor:孤傲苍狼

58 \*

59 \* @param conn

60 \* @param st

61 \* @param rs

62 \*/

63 public static void release(Connection conn,Statement st,ResultSet rs){

64 if(rs!=null){

65 try{

66 //关闭存储查询结果的ResultSet对象

67 rs.close();

68 }catch (Exception e) {

69 e.printStackTrace();

70 }

71 rs = null;

72 }

73 if(st!=null){

74 try{

75 //关闭负责执行SQL命令的Statement对象

76 st.close();

77 }catch (Exception e) {

78 e.printStackTrace();

79 }

80 }

81

82 if(conn!=null){

83 try{

84 //将Connection连接对象还给数据库连接池

85 conn.close();

86 }catch (Exception e) {

87 e.printStackTrace();

88 }

89 }

90 }

91 }

**8 测试C3P0数据源**

1 package me.gacl.test;

2

3 import java.sql.Connection;

4 import java.sql.PreparedStatement;

5 import java.sql.ResultSet;

6 import org.junit.Test;

7 import me.gacl.util.JdbcUtils\_C3P0;

8 import me.gacl.util.JdbcUtils\_DBCP;

9

10 public class DataSourceTest {

11

12 @Test

13 public void c3p0DataSourceTest() {

14 Connection conn = null;

15 PreparedStatement st = null;

16 ResultSet rs = null;

17 try{

18 //获取数据库连接

19 conn = JdbcUtils\_C3P0.getConnection();

20 String sql = "insert into test1(name) values(?)";

21 st = conn.prepareStatement(sql);

22 st.setString(1, "gacl");

23 st.executeUpdate();

24 //获取数据库自动生成的主键

25 rs = st.getGeneratedKeys();

26 if(rs.next()){

27 System.out.println(rs.getInt(1));

28 }

29 }catch (Exception e) {

30 e.printStackTrace();

31 }finally{

32 //释放资源

33 JdbcUtils\_C3P0.release(conn, st, rs);

34 }

35 }

36 }

## 五、配置Tomcat数据源

在实际开发中，我们有时候还会使用服务器提供给我们的数据库连接池，比如我们希望Tomcat服务器在启动的时候可以帮我们创建一个数据库连接池，那么我们在应用程序中就不需要手动去创建数据库连接池，直接使用Tomcat服务器创建好的数据库连接池即可。要想让Tomcat服务器在启动的时候帮我们创建一个数据库连接池，那么需要简单配置一下Tomcat服务器。

**1、JNDI技术简介**

JNDI(Java Naming and Directory Interface)，Java命名和目录接口，它对应于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。Tomcat服务器创建的数据源是以JNDI资源的形式发布的，所以说在Tomat服务器中配置一个数据源实际上就是在配置一个JNDI资源，通过查看Tomcat文档，我们知道使用如下的方式配置tomcat服务器的数据源：

1 <Context>

2 <Resource name="jdbc/datasource" auth="Container"

3 type="javax.sql.DataSource" username="root" password="XDP"

4 driverClassName="com.mysql.jdbc.Driver"

5 url="jdbc:mysql://localhost:3306/jdbcstudy"

6 maxActive="8" maxIdle="4"/>

7 </Context>

**2 服务器创建好数据源之后，我们的应用程序又该怎么样得到这个数据源呢**

Tomcat服务器创建好数据源之后是以JNDI的形式绑定到一个JNDI容器中的，我们可以把JNDI想象成一个大大的容器，我们可以往这个容器中存放一些对象，一些资源，JNDI容器中存放的对象和资源都会有一个独一无二的名称，应用程序想从JNDI容器中获取资源时，只需要告诉JNDI容器要获取的资源的名称，JNDI根据名称去找到对应的资源后返回给应用程序。我们平时做javaEE开发时，服务器会为我们的应用程序创建很多资源，比如request对象，response对象，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。

**3 对于上面的name="jdbc/datasource"数据源资源，在应用程序中可以用如下的代码去获取**

1 Context initCtx = new InitialContext();

2 Context envCtx = (Context) initCtx.lookup("java:comp/env");

3 dataSource = (DataSource)envCtx.lookup("jdbc/datasource");

此种配置下，数据库的驱动jar文件需放置在tomcat的lib下

**4、配置Tomcat数据源**

（1）、在Web项目的WebRoot目录下的META-INF目录创建一个context.xml文件

（2）、在context.xml文件配置tomcat服务器的数据源

1 <Context>

2 <Resource

3 name="jdbc/datasource"

4 auth="Container"

5 type="javax.sql.DataSource"

6 username="root"

7 password="XDP"

8 driverClassName="com.mysql.jdbc.Driver"

9 url="jdbc:mysql://localhost:3306/jdbcstudy"

10 maxActive="8"

11 maxIdle="4"/>

12 </Context>

（3）、将数据库的驱动jar文件需放置在tomcat的lib下

（4）、在获取数据库连接的工具类(如jdbcUtils)的静态代码块中获取JNDI容器中的数据源

1 package me.gacl.util;

3 import java.sql.Connection;

4 import java.sql.ResultSet;

5 import java.sql.SQLException;

6 import java.sql.Statement;

7 import javax.naming.Context;

8 import javax.naming.InitialContext;

9 import javax.sql.DataSource;

11 /\*\*

12 \* @ClassName: JdbcUtils\_DBCP

13 \* @Description: 数据库连接工具类

14 \* @author: 孤傲苍狼

15 \* @date: 2014-10-4 下午6:04:36

16 \*

17 \*/

18 public class JdbcUtils\_JNDI {

19

20 private static DataSource ds = null;

21 //在静态代码块中创建数据库连接池

22 static{

23 try{

24 //初始化JNDI

25 Context initCtx = new InitialContext();

26 //得到JNDI容器

27 Context envCtx = (Context) initCtx.lookup("java:comp/env");

28 //从JNDI容器中检索name为jdbc/datasource的数据源

29 ds = (DataSource)envCtx.lookup("jdbc/datasource");

30 }catch (Exception e) {

31 throw new ExceptionInInitializerError(e);

32 }

33 }

34

35 /\*\*

36 \* @Method: getConnection

37 \* @Description: 从数据源中获取数据库连接

38 \* @Anthor:孤傲苍狼

39 \* @return Connection

40 \* @throws SQLException

41 \*/

42 public static Connection getConnection() throws SQLException{

43 //从数据源中获取数据库连接

44 return ds.getConnection();

45 }

46

47 /\*\*

48 \* @Method: release

49 \* @Description: 释放资源，

50 \* 释放的资源包括Connection数据库连接对象，负责执行SQL命令的Statement对象，存储查询结果的ResultSet对象

51 \* @Anthor:孤傲苍狼

52 \*

53 \* @param conn

54 \* @param st

55 \* @param rs

56 \*/

57 public static void release(Connection conn,Statement st,ResultSet rs){

58 if(rs!=null){

59 try{

60 //关闭存储查询结果的ResultSet对象

61 rs.close();

62 }catch (Exception e) {

63 e.printStackTrace();

64 }

65 rs = null;

66 }

67 if(st!=null){

68 try{

69 //关闭负责执行SQL命令的Statement对象

70 st.close();

71 }catch (Exception e) {

72 e.printStackTrace();

73 }

74 }

75

76 if(conn!=null){

77 try{

78 //将Connection连接对象还给数据库连接池

79 conn.close();

80 }catch (Exception e) {

81 e.printStackTrace();

82 }

83 }

84 }

85 }

**5 写一个Servlet测试JNDI数据源**

1 package me.gacl.test;

3 import java.io.IOException;

4 import java.sql.Connection;

5 import java.sql.PreparedStatement;

6 import java.sql.ResultSet;

7 import javax.servlet.ServletException;

8 import javax.servlet.http.HttpServlet;

9 import javax.servlet.http.HttpServletRequest;

10 import javax.servlet.http.HttpServletResponse;

11 import me.gacl.util.JdbcUtils\_JNDI;

13 public class JNDITest extends HttpServlet {

14

15 public void doGet(HttpServletRequest request, HttpServletResponse response)

16 throws ServletException, IOException {

17 Connection conn = null;

18 PreparedStatement st = null;

19 ResultSet rs = null;

20 try{

21 //获取数据库连接

22 conn = JdbcUtils\_JNDI.getConnection();

23 String sql = "insert into test1(name) values(?)";

24 st = conn.prepareStatement(sql);

25 st.setString(1, "gacl");

26 st.executeUpdate();

27 //获取数据库自动生成的主键

28 rs = st.getGeneratedKeys();

29 if(rs.next()){

30 System.out.println(rs.getInt(1));

31 }

32 }catch (Exception e) {

33 e.printStackTrace();

34 }finally{

35 //释放资源

36 JdbcUtils\_JNDI.release(conn, st, rs);

37 }

38 }

39

40 public void doPost(HttpServletRequest request, HttpServletResponse response)

41 throws ServletException, IOException {

42 doGet(request, response);

43 }

44

45 }

## 六、数据库连接池问题

**1 简单介绍一下笔者项目中MySQL的使用情况。**

项目中使用MySQL做数据存储，Redis做数据缓存。读写数据流程如下：读数据：先查询Redis中是否存在，存在直接取出数据，不存在从MySQL查询并保存包Redis中缓存起来。写数据：先写入Redis，然后异步提交到MySQL写入线程，实现数据落地。

**2 如何保证数据读写的“原子性”?如何解决脏数据?**

众所周知，当使用数据库跨表操作时，利用数据库引擎提供的事务和回滚机制，可以保证数据库跨表操作的原子性。一方面在项目中引入数据缓存层也是一种常见的减轻数据库压力的做法。另一方面引入了缓存增加了数据操作的复杂度。

**3 如何解决缓存中和数据库中数据不一致的问题?**

笔者以线上项目为例简单的说了说。在线上的代码，主要以缓存层数据为主，数据不统一的情况，会在下一次落地时覆盖数据库的数据，实现同步。假如是因为版本迭代数据库字段变更导致数据库落地错误时，由于Redis缓存的有效期是7天，在7天内，手动执行脚本将数据重新落地。

**4 由于面试官一再强调是自动修复脏数据。**

说实话笔者在保证Redis和MySQL两者数据操作的原子性这块。所以也没有什么好的办法。

由于面试官一再询问，笔者就稍微讲了讲一个思路。也许是否可以使用数据库的原子性，回滚的时候回退数据?当然事实证明一时的想法总是有漏洞的。数据回退涉及到数据回退前多次被修改的复杂情况。显得不是很合适。

**5 最后尴尬的是面试官的结论：(大意如下)**

在修改数据前，尽可能多的判定用户是否满足操作的条件，以便于提早发现问题并退出, 以避免数据操作。同时避免异步操作数据库，同步操作及早出现问题并退出。

## 七、性能测试对比

对现有的数据库连接池做调研对比，综合性能，可靠性，稳定性，扩展性等因素选出推荐出最优的数据库连接池，本文所有测试均是MySQL库，测试结论：

1：性能方面 hikariCP>druid>tomcat-jdbc>dbcp>c3p0 。hikariCP的高性能得益于最大限度的避免锁竞争。

2：druid功能最为全面，sql拦截等功能，统计数据较为全面，具有良好的扩展性。

3：综合性能，扩展性等方面，可考虑使用druid或者hikariCP连接池。

4：可开启prepareStatement缓存，对性能会有大概20%的提升。



**1：获取关闭连接性能测试**

（1）测试说明：

@初始连接和最小连接均为5，最大连接为20。在borrow和return均不心跳检测

@其中打开关闭次数为: 100w次

@测试用例和mysql在同一台机器上面，尽量避免io的影响

@使用mock和连接mysql在不同线程并发下的响应时间

（2）测试结果：

**@mock和mysql连接性能表现差不多，主要是由于初始化的时候建立了连接后期不再建立连接，和使用mock连接逻辑一致。**

**@性能表现：hikariCP>druid>tomcat-jdbc>dbcp>c3p0。**

**@hikariCP 的性能及其优异。hikariCP号称java平台最快的数据库连接池。**

**@hikariCP在并发较高的情况下，性能基本上没有下降。**

**@c3p0连接池的性能很差，不建议使用该数据库连接池。**

**2：查询一条语句性能测试**

（1）测试说明：

@初始连接和最小连接均为8，最大连接为8。在borrow和return均不心跳检测

@查询的次数为10w次，查询的语句为 1：打开连接 2：执行 ：select 1 3：关闭连接

@测试用例和mysql在同一台机器上面，尽量避免io的影响

（2）测试结果：

**@在并发比较少的情况下，每个连接池的响应时间差不多。是由于并发少，基本上没有资源竞争。**

**@在并发较高的情况下，随着并发的升高，hikariCP响应时间基本上没有变动。**

**@c3p0随着并发的提高，性能急剧下降**

## 八、数据库连接池优化配置(druid,dbcp,c3p0)

**1 考虑因素**

（1）当前连接DB的规模

（2）并发情况

（3）执行db的响应时间

**2 配置考虑**

（1）初始化连接：可考虑设置为3个连接 。对于db规模特别大的情况下可考虑设置为1个。避免启动时间过长；

（2）最小连接：可考虑该值的设置和初始化连接保持一致；

（3）最大连接：对于有较大DB规模，最大连接不要设置过大，避免本地维护的db太大。 如果对应到数据源的并发数过高，可考虑增大最大连接数。

（4）获取连接的超时时间：如果连接全部被占用，需要等待的时间。可以根据当前系统的响应时间判定，如果容忍度较高，可以大点。容忍度较低，设置小点。

（5）当获取连接和释放连接心跳检测：建议全部关闭，否则每个数据库访问指令会对数据库生产额外的两条心跳检测的指令，增加数据库的负载。连接有效性的检查改用后台空闲连接检查。

（6）连接有效性检测时间：该值需要结合数据库的wait\_timeout,interactive\_timeout值进行设置。假如数据库为120s，则心跳检测时间在120s以内越大越好。如果太小，心跳检测时间会比较频繁。建议设置为90s。

（7）最大空闲时间：如果连接超过该时间没有使用过，则会进行close掉。 该值不要太小，避免频繁的建立连接关闭连接。也不要太大，导致一直无法关闭。

（8）心跳检查的sql语句：尽量使用ping命令，ping的性能较查询语句高。大部分的数据库连接池不配置query语句，便会调用ping命令。

（9）prepareStatement缓存：可以根据自己的业务来判定是否开启。开启后对性能的影响依赖于具体业务和并发情况。可考虑暂时不开启。

（10）连接使用超时：业务拿到一个连接，如果超过指定的时间未归还，是否把该连接给给回收掉。超时时间等和具体的业务关联。暂时建议先不开启。

**3 配置推荐**

**（1）druid推荐配置：**

|  |  |  |
| --- | --- | --- |
| initialSize | 3 | 初始化配置 |
| minIdle | 3 | 最小连接数 |
| maxActive | 15 | 最大连接数 |
| maxWait | 5000 | 获取连接超时时间（单位：ms） |
| timeBetweenEvictionRunsMillis | 90000 | 连接有效性检测时间(单位:ms) |
| testOnBorrow | false | 获取连接检测 |
| testOnReturn | false | 归还连接检测 |
| minEvictableIdleTimeMillis | 1800000 | 最大空闲时间(单位ms) |
| testWhileIdle | true | 在获取连接后，确定是否要进行连接空间时间的检查 |

**配置说明：**

1：minEvictableIdleTimeMillis(最大空闲时间)：默认为30分钟，配置里面不进行设置。

2：testOnBorrow ,testOnReturn 默认为关闭，可以设置为不配置。

3：testWhileIdle(在获取连接后，确定是否要进行连接空闲时间的检查)。默认为true。配置里面不再进行设置。

**流程说明：**

1：在第一次调用connection的时候，才会进行 initialSize的初始化。

2：心跳检测时间线程，会休眠timeBetweenEvictionRunsMillis时间，然后只对(没有borrow的线程 减去 minIdle)的线程进行检查，如果空闲时间大于minEvictableIdleTimeMillis则进行close。

3：testWhileIdle必须设置为true，在获取到连接后，先检查testOnBorrow，然后再判定testwhileIdle，如果连接空闲时间大于timeBetweenEvictionRunsMillis，则会进行心跳检测。

4：不需要配置validationQuery，如果不配置的情况下会走ping命令，性能更高。

5：连接保存在数组里面，获取连接的时候，获取数组的最后一位。在timeBetweenEvictionRunsMillis时是从前往后进行检查连接的有效性。

**（2）dbcp推荐配置**

|  |  |  |
| --- | --- | --- |
| **参数** | **配置** | **说明** |
| initialSize | 3 | 初始化配置 |
| minIdle | 3 | 最小连接数 |
| maxIdle | 15 | 最大空闲连接 |
| maxTotal | 15 | 最大连接数 |
| maxWaitMillis | 5000 | 获取连接超时时间(单位ms) |
| timeBetweenEvictionRunsMillis | 90000 | 心跳检测时间(单位ms) |
| minEvictableIdleTimeMillis | 1800000 | 最大空闲时间(单位ms) |
| testOnBorrow | FALSE | 获取连接检测 |
| testOnReturn | FALSE | 归还连接检测 |
| numTestsPerEvictionRun | -1 | 空闲连接检查的个数 |
| testWhileIdle | TRUE | 是否开启对空闲连接的检查 |

**配置说明：**

1：关于maxidle和maxTotal尽量保持一致。

2：numTestsPerEvictionRun 设置为-1，代表对所有的连接均进行检查。默认值为3。-1代表对全部idle的连接检查有效性。 否则有可能造成部分连接的有效性未进行检查。

3：testWhileIdle 也必须为true，代表需要检查有效性。

4：minEvictableIdleTimeMillis默认值为30分钟，可以不用进行设置。

5：testOnReturn默认值为false，可以不用设置。但是testOnBorrow必须进行设置为false，默认值为true。

6：validationQuery不配置默认走ping命令

**流程说明：**

1：在第一次调用connection的时候，才会进行 initialSize的初始化。

2：不需要配置validationQuery，如果不配置的情况下会走ping命令，性能更高。

3：连接保存在LinkedBlockingDeque 中。来做并发的控制。

4：后端会有一个定时任务，间隔为timeBetweenEvictionRunsMillis，先确定需要对多少线程进行检测(numTestsPerEvictionRun控制)，然后判定是否超过minEvictableIdleTimeMillis，如果超过则close掉。没有超过，则判定testWhileIdle为true的话，进行心跳检查。如果检查失败则关闭连接。

5：在return连接的时候会判定maxIdle，如果当前空闲连接是否大于maxIdle，则会关闭掉连接。

**（3）c3p0 推荐配置**

|  |  |  |
| --- | --- | --- |
| 参数 | 推荐值 | 说明 |
| initialPoolSize | 3 | 初始化配置 |
| minPoolSize | 3 | 最小连接数 |
| maxPoolSize | 15 | 最大连接数 |
| acquireIncrement | 1 | 每次获取的个数 |
| checkoutTimeout | 5000 | 获取连接超时时间(单位ms) |
| idleConnectionTestPeriod | 90 | 心跳检测时间(单位 s) |
| maxIdleTime | 1800 | 最大空闲时间(单位 s) |
| testConnectionOnCheckout | FALSE | 获取连接检测 |
| testConnectionOnCheckin | FALSE | 归还连接检测 |
| numHelperThreads | 1 |  |

**配置说明：**

1：testConnectionOnCheckout和testConnectionOnCheckin默认为false，可不用配置

2：preferredTestQuery不用配置，默认走ping命令。

3：numHelperThreads 默认是开启3个线程。如果数据源较多，这里会存在较多线程。 这里设置为1，避免线程较多的情况。

**流程说明：**

1：在第一次调用connection的时候，才会进行 initialPoolSize的初始化。

2：在进行心跳检测的时候，会对所有的空闲连接进行心跳检测。如果发现总连接小于最小连接数，则会创建连接，保持最小的连接数。

**Druid总结**

## 一、Druid简介

1 Druid是一个为在大数据集之上做实时统计分析而设计的开源数据存储。**这个系统集合了一个面向列存储的层，一个分布式、shared-nothing的架构，和一个高级的索引结构，来达成在秒级以内对十亿行级别的表进行任意的探索分析。**

2 互联网技术的快速增长催生了各类大体量的数据，Hadoop很大的贡献在于帮助企业将他们那些低价值的事件流数据转化为高价值的聚合数据，这适用于各种应用但Hadoop擅长的是存储和获取大规模数据，但是它并不提供任何性能上的保证它能多快获取到数据。此外，虽然Hadoop是一个高可用的系统，但是在高并发负载下性能会下降Hadoop是一个很好的后端、批量处理和数据仓库系统。在一个需要高并发并且保证查询性能和数据可用性的并需要提供产品级别的保证的需求，Hadoop并不能满足，因此创建了Druid，一个开源的、分布式、列存储、实时分析的数据存储。在许多方面，Druid和其他OLAP系统有很多相似之处，交互式查询系统，内存数据库(MMDB)，众所周知的分布式数据存储。其中的分布式和查询模型都参考了当前的一些搜索引擎的基础架构。

## 二、Durid特性

**1 亚秒级查询：**druid提供了快速的聚合能力以及亚秒级的OLAP查询能力，多租户的设计，是面向用户分析应用的理想方式

**2 实时数据注入**：druid支持流数据的注入，并提供了数据的事件驱动，保证在实时和离线环境下事件的实效性和统一性

**3 可扩展的PB级存储：**druid集群可以很方便的扩容到PB的数据量，每秒百万级别的数据注入。即便在加大数据规模的情况下，也能保证时其效性

**4 多环境部署：**druid既可以运行在商业的硬件上，也可以运行在云上。它可以从多种数据系统中注入数据，包括hadoop，spark，kafka，storm和samza等

**5丰富的社区：**druid拥有丰富的社区，供大家学习

## 三、使用场景

第一：适用于清洗好的记录实时录入，但不需要更新操作  
第二：支持宽表，不用join的方式（换句话说就是一张单表）  
第三：可以总结出基础的统计指标，可以用一个字段表示  
第四：对时区和时间维度(year、month、week、day、hour等)要求高的（甚至到分钟级别）  
第五：实时性很重要  
第六：对数据质量的敏感度不高  
第七：用于定位效果分析和策略决策参考

**架构**

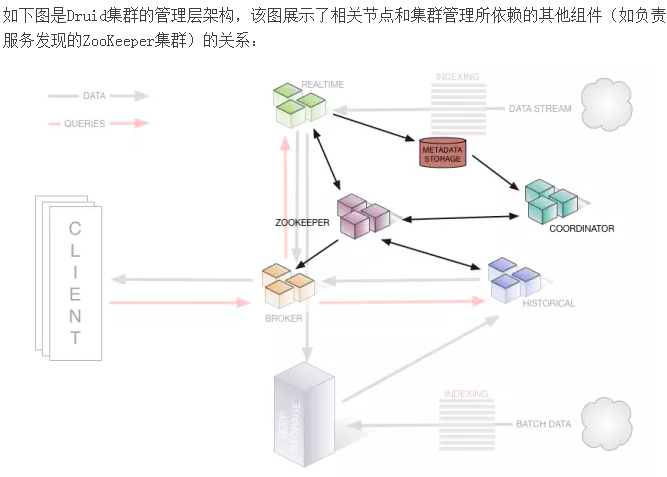
**Historical：**对非实时数据进行处理存储和查询；

**Realtime：**实时摄取数据、监听输入数据流

**Coordinator：**监控historical节点

**Broker：**接收来自外部客户端的查询，和将查询转发到Realtime和historical

**Indexer：**负责索引服务



## 四、RealNode

**1 实时节点封装了导入和查询事件数据的功能，经由这些节点导入的事件数据可以立刻被查询。**

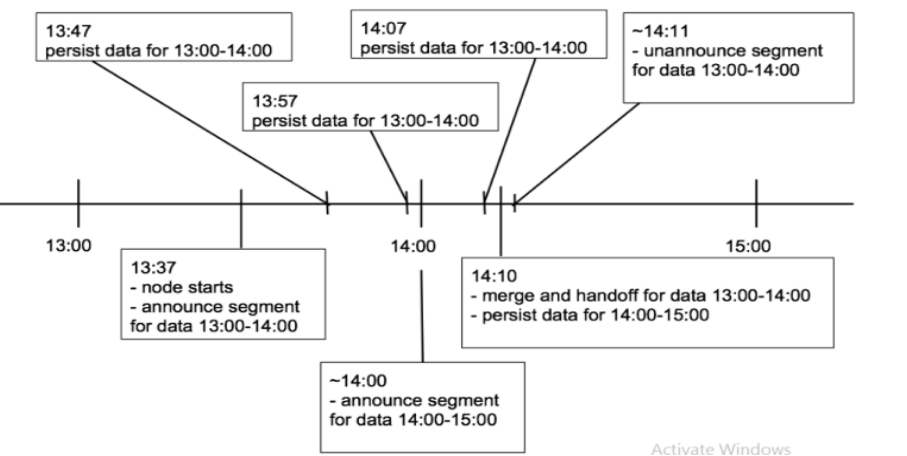
**2 实时节点只关心一小段时间内的事件数据，并定期把这段时间内收集的这批不可变事件数据导入到Druid集群里面另外一个专门负责处理不可变的批量数据的节点中去。**实时节点通过Zookeeper的协调和Druid集群的其他节点协调工作。实时节点通过Zookeeper来宣布他们的在线状态和他们提供的数据。实时节点为所有传入的事件数据维持一个内存中的索引缓存, 随着事件数据的传入，这些索引会逐步递增，并且这些索引是可以立即查询的，查询这些缓存于JVM的基于堆的缓存中的事件数据，Druid就表现得和行存储一样。

**3 为了避免堆溢出问题，实时节点会定期地、或者在达到设定的最大行限制的时候，把内存中的索引持久化到磁盘去。**这个持久化进程会把保存于内存缓存中的数据转换为基于列存储的格式，所有持久化的索引都是不可变的，并且实时节点会加载这些索引到off-heap内存中使得它们可以继续被查询。

**4 上图实时节点缓存事件数据到内存中的索引上，然后有规律的持久化到磁盘上。**在转移之前，持久化的索引会周期性地合并在一起。查询会同时命中内存中的和已持久化的索引

所有的实时节点都会周期性的启动后台的计划任务搜索本地的持久化索引，后台计划任务将这些持久化的索引合并到一起并生成一块不可变的数据，这些数据块包含了一段时间内的所有已经由实时节点导入的事件数据，我们称这些数据块为”Segment”。在传送阶段，实时节点将这些segment上传到一个永久持久化的备份存储中，通常是一个分布式文件系统，例如S3或者HDFS，Druid称之为”Deep Storage”。

**5 实时节点处理流程：导入、持久化、合并和传送这些阶段都是流动的，并且在这些处理阶段中不会有任何数据的丢失**，数据流图如下：



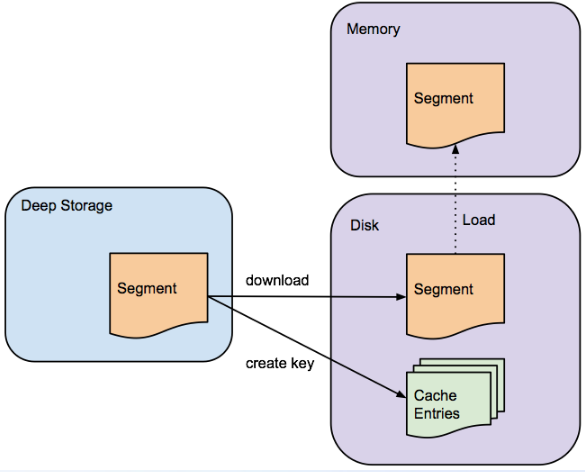
节点启动于13:47，并且只会接受当前小时和下一小时的事件数据。当事件数据开始导入后，节点会宣布它为13:00到14:00这个时间段的Segment数据提供服务每10分钟（这个时间间隔是可配置的），节点会将内存中的缓存数据刷到磁盘中进行持久化，在当前小时快结束的时候，节点会准备接收14:00到15:00的事件数据，一旦这个情况发生了，节点会准备好为下一个小时提供服务，并且会建立一个新的内存中的索引。

**6 随后，节点宣布它也为14:00到15:00这个时段提供一个segment服务。**节点并不是马上就合并13:00到14:00这个时段的持久化索引，而是会等待一个可配置的窗口时间，直到所有的13:00到14:00这个时间段的一些延迟数据的到来。这个窗口期的时间将事件数据因延迟而导致的数据丢失减低到最小。在窗口期结束时，节点会合并13:00到14:00这个时段的所有持久化的索引合并到一个独立的不可变的segment中，并将这个segment传送走，一旦这个segment在Druid集群中的其他地方加载了并可以查询了，实时节点会刷新它收集的13:00到14:00这个时段的数据的信息，并且宣布取消为这些数据提供服务。

## 五、HistoricalNode

**1 历史节点封装了加载和处理由实时节点创建的不可变数据块（segment）的功能。**在很多现实世界的工作流程中，大部分导入到Druid集群中的数据都是不可变的，因此，历史节点通常是Druid集群中的主要工作组件。

**2 历史节点遵循shared-nothing的架构，因此节点间没有单点问题。**节点间是相互独立的并且提供的服务也是简单的，它们只需要知道如何加载、删除和处理不可变的segment  (注：shared nothing architecture是一 种分布式计算架构，这种架构中不存在集中存储的状态，整个系统中没有资源竞争，这种架构具有非常强的扩张性，在web应用中广泛使用）类似于实时节点，历史节点在Zookeeper中通告它们的在线状态和为哪些数据提供服务。加载和删除segment的指令会通过Zookeeper来进行发布，指令会包含segment保存在deep storage的什么地方和怎么解压、处理这些segment的相关信息在历史节点从deep storage下载某一segment之前，它会先检查本地缓存信息中看segment是否已经存在于节点中，如果segment还不存在缓存中，历史节点会从deep storage中下载segment到本地一旦处理完成，这个segment就会在Zookeeper中进行通告。此时，这个segment就可以被查询了。历史节点的本地缓存也支持历史节点的快速更新和重启，在启动的时候，该节点会检查它的缓存，并为任何它找到的数据立刻进行服务的提供，如下图：

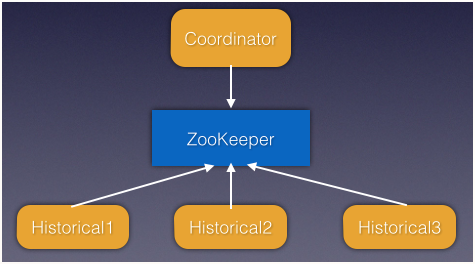


**历史节点从deep storage下载不可变的segment。segment在可以被查询之前必须要先加载到内存中**

**3 历史节点可以支持读一致性，因为它们只处理不可变的数据。**不可变的数据块同时支持一个简单的并行模型：历史节点可以以非阻塞的方式并发地去扫描和聚合不可变的数据块

Tiers: 历史节点可以分组到不同的tier中，哪些节点会被分到一个tier中是可配置的。Tier的目的是可以根据segment的重要程度来分配高或低的优先级来进行数据的分布。可以为不同的tier配置不同的性能和容错参数。例如，可以使用一批很多个核的CPU和大容量内存的节点来组成一个“热点数据”的tier，这个“热点数据”集群可以配置来用于下载更多经常被查询的数据。

**4 一个类似的”冷数据”集群可以使用一些性能要差一些的硬件来创建，**“冷数据”集群可以只包含一些不是经常访问的segment可用性: **历史节点依赖于Zookeeper来管理segment的加载和卸载。**



如果Zookeeper变得不可用的时候，历史节点就不再可以为新的数据提供服务和卸载过期的数据，因为是通过HTTP来为查询提供服务的。对于那些查询它当前已经在提供服务的数据，历史节点仍然可以进行响应。这意味着Zookeeper运行故障时不会影响那些已经存在于历史节点的数据的可用性。

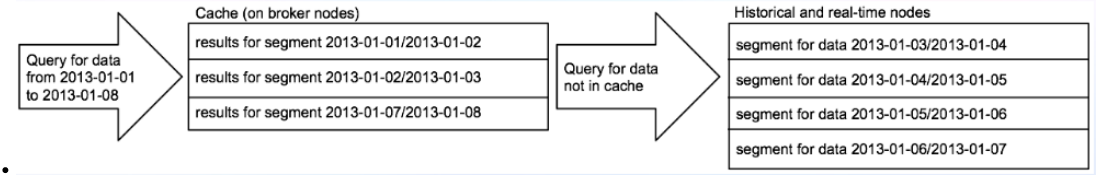
## 六、BrokerNode

**1 Broker节点扮演着历史节点和实时节点的查询路由的角色。**

**2 Broker节点知道发布于Zookeeper中的关于哪些segment是可查询的和这些segment**是保存在哪里的，Broker节点就可以将到来的查询请求路由到正确的历史节点或者是实时节点，Broker节点也会将历史节点和实时节点的局部结果进行合并，然后返回最终的合并后的结果给调用者

**3 缓存：Broker节点包含一个支持LRU失效策略的缓存。**这个缓存可以使用本地堆内存或者是一个外部的分布式 key/value 存储，例如Memcached。每次Broker节点接收到查询请求时，都会先将查询映射到一组segment中去。这一组确定的segment的结果可能已经存在于缓存中，而不需要重新计算。对于那些不存在于缓存的结果，Broker节点会将查询转发到正确的历史节点和实时节点中去，一旦历史节点返回结果，Broker节点会将这些结果缓存起来以供以后使用，这个过程如下图所示

**注意：实时数据永远不会被缓存，因此查询实时节点的数据的查询请求总是会被转发到实时节点上去。实时数据是不断变化的，因此缓存实时数据是不可靠的**



上图：结果会为每一个segment缓存。查询会合并缓存结果与历史节点和实时节点的计算结果

**4 缓存也可作为数据可用性的附加级别。**在所有历史节点都出现故障的情况下，对于那些命中已经在缓存中缓存了结果的查询，仍然是可以返回查询结果的。

**5 可用性：**在所有的Zookeeper都中断的情况下，数据仍然是可以查询的。如果Broker节点不可以和Zookeeper进行通信了，它会使用它最后一次得到的整个集群的视图来继续将查询请求转发到历史节点和实时节点，Broker节点假定集群的结构和Zookeeper中断前是一致的。在实践中，在我们诊断Zookeeper的故障的时候，这种可用性模型使得Druid集群可以继续提供查询服务，为我们争取了更多的时间。

**6 说明**：通常在ShareNothing的架构中,如果一个节点变得不可用了,会有一个服务将下线的这个节点的数据搬迁到其他节点，但是如果这个节点下线后又立即重启,而如果服务在一下线的时候就开始搬迁数据,是会产生跨集群的数据传输,实际上是没有必要的。因为分布式文件系统对同一份数据会有多个副本,搬迁数据实际上是为了满足副本数.而下线又重启的节点上的数据不会有什么丢失的，因此短期的副本不足并不会影响整体的数据健康状况.何况跨机器搬迁数据也需要一定的时间,何不如给定一段时间如果它真的死了,才开始搬迁。

## 七、CoordinatorNode

**1 主要负责数据的管理和在历史节点上的分布。**协调节点告诉历史节点加载新数据、卸载过期数据、复制数据、和为了负载均衡移动数据。Druid为了维持稳定的视图，使用一个多版本的并发控制交换协议来管理不可变的segment。如果任何不可变的segment包含的数据已经被新的segment完全淘汰了，则过期的segment会从集群中卸载掉。

**2 协调节点会经历一个leader选举的过程，来决定由一个独立的节点来执行协调功能，其余的协调节点则作为冗余备份节点。**协调节点会周期性的执行来确定集群的当前状态，它通过在运行的时候对比集群的预期状态和集群的实际状态来做决定。和所有的Druid节点一样，协调节点维持一个和Zookeeper的连接来获取当前集群的信息。

**3 协调节点也维持一个与MySQL数据库的连接，MySQL包含有更多的操作参数和配置信息。**其中一个存在于MySQL的关键信息就是历史节点可以提供服务的所有segment的一个清单，这个表可以由任何可以创建segment的服务进行更新，例如实时节点。

**4 MySQL数据库中还包含一个Rule表来控制集群中segment的是如何创建、销毁和复制**

Rules：Rules管理历史segment是如何在集群中加载和卸载的。

Rules指示segment应该如何分配到不同的历史节点tier中，每一个tier中应该保存多少份segment的副本。Rules还可能指示segment何时应该从集群中完全地卸载。Rules通常设定为一段时间，例如，一个用户可能使用Rules来将最近一个月的有价值的segment载入到一个“热点数据”的集群中，最近一年的有价值的数据载入到一个“冷数据”的集群中，而将更早时间前的数据都卸载掉。协调节点从MySQL数据库中的rule表加载一组rules。Rules可能被指定到一个特定的数据源，或者配置一组默认的rules。协调节点会循环所有可用segment并会匹配第一条适用于它的rule

**5 负载均衡**

在典型的生产环境中，查询通常命中数十甚至上百个segment，由于每个历史节点的资源是有限的，segment必须被分布到整个集群中，以确保集群的负载不会过于不平衡。要确定最佳的负载分布，需要对查询模式和速度有一定的了解。通常，查询会覆盖一个独立数据源中最近的一段邻近时间的一批segment。平均来说，查询更小的segment则更快这些查询模式提出以更高的比率对历史segment进行复制，把大的segment以时间相近的形式分散到多个不同的历史节点中，并且使存在于不同数据源的segment集中在一起为了使集群中segment达到最佳的分布和均衡，根据segment的数据源、新旧程度、和大小，开发了一个基于成本的优化程序

**6 副本/复制（Replication）**

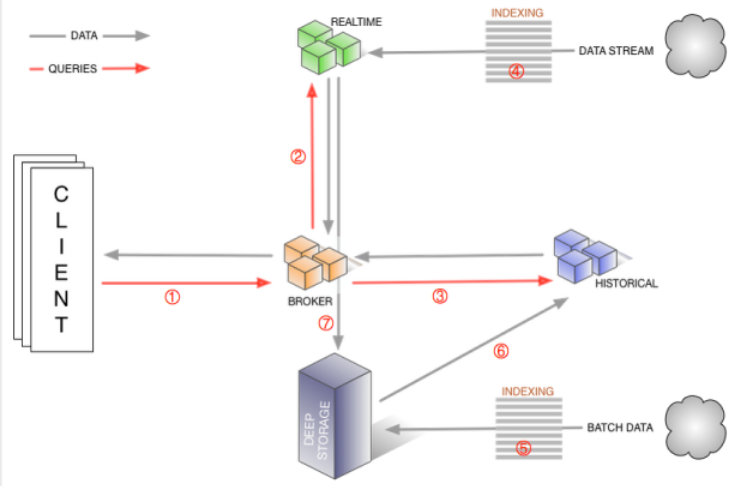
协调节点可能会告诉不同的历史节点加载同一个segment的副本。每一个历史节点tier中副本的数量是完全可配置。设置一个高级别容错性的集群可以设置一个比较高数量的副本数。segment的副本被视为和原始segment一样的，并使用相同的负载均衡算法通过复制segment，单一历史节点故障对于整个Druid集群来说是透明的，不会有任何影响。

**7 可用性**

协调节点有Zookeeper和MySQL这两个额外的依赖，协调节点依赖Zookeeper来确定集群中有哪些历史节点。如果Zookeeper变为不可用，协调节点将不可以再进行segment的分配、均衡和卸载指令的发送。不过，这些都不会影响数据的可用性。对于MySQL和Zookeeper响应失效的设计原则是一致的：如果协调节点一个额外的依赖响应失败了，集群会维持现状。Druid使用MySQL来存储操作管理信息和关于segment如何存在于集群中的segment元数据。如果MySQL下线了，这些信息就在协调节点中变得不可用，不过这不代表数据不可用如果协调节点不可以和MySQL进行通信，他们会停止分配新的segment和卸载过期的segment。在MySQL故障期间Broker节点、历史节点、实时节点都是仍然可以查询的。

## 八、架构介绍

**1 Druid 由上面介绍的角色组成的构架图**



**2 查询路径：红色箭头:**

①客户端向Broker发起请求,Broker会将请求路由到

②实时节点和

③历史节点

**3 Druid数据流转:黑色箭头：数据源包括实时流和批量数据.**

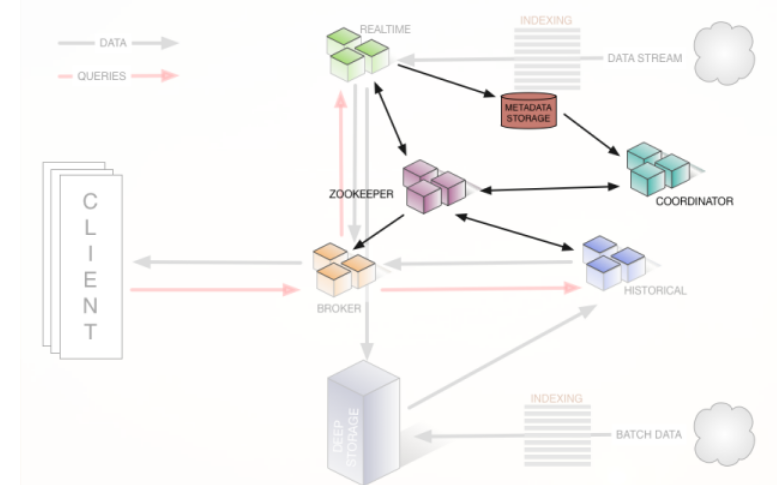
④实时流经过索引直接写到实时节点，

⑤批量数据通过IndexService存储到DeepStorage,

⑥再由历史节点加载.

⑦实时节点也可以将数据转存到DeepStorage

**4 Druid的集群依赖了ZooKeeper来维护数据拓扑. 每个组件都会与ZooKeeper交互**



实时节点在转存Segment到DeepStorage, 会写入自己转存了什么Segment。协调节点管理历史节点,它负责从ZooKeeper中获取要同步/下载的Segment,并指派任务给具体的历史节点去完成

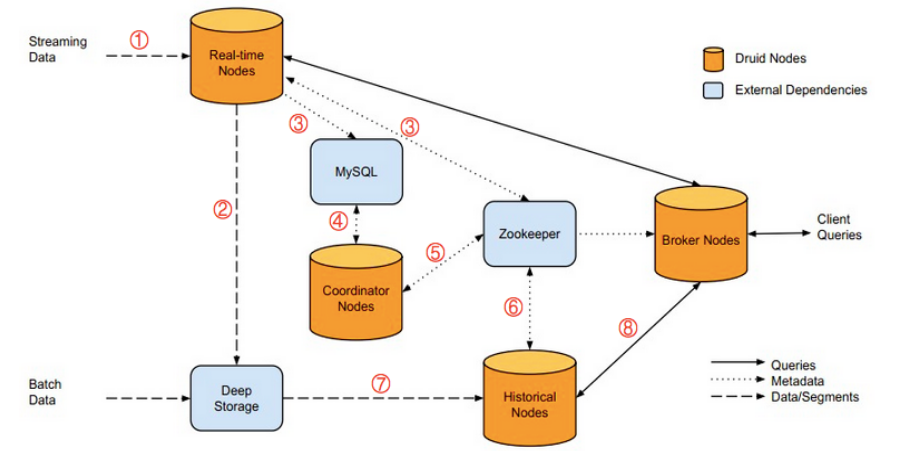
**5 历史节点从ZooKeeper中领取任务,任务完成后要将ZooKeeper条目删除表示完成了任务。**Broker节点根据ZooKeeper中的Segment所在的节点, 将查询请求路由到指定的节点

对于一个查询路由路径,Broker只会将请求分发到实时节点和历史节点, 因此元数据存储和DeepStorage都不会参与查询中(看做是后台的进程).

## 九、MetaData Storage 与 Zookeeper

**1 MetaStore和ZooKeeper中保存的信息是不一样的. ZooKeeper中保存的是Segment属于哪些节点.** 而MetaStore则是保存Segment的元数据信息。为了使得一个Segment存在于集群中,MetaStore存储的记录是关于Segment的自描述元数据: Segment的元数据,大小,所在的DeepStorage。元数据存储的数据会被协调节点用来知道集群中可用的数据应该有哪些(Segment可以通过实时节点转存或者批量数据直接写入).

**2 除了上面介绍的节点角色外，Druid还依赖于外部的三个组件：ZooKeeper, Metadata Storage, Deep Storage，**数据与查询流的交互图如下



① 实时数据写入到实时节点,会创建索引结构的Segment.

② 实时节点的Segment经过一段时间会转存到DeepStorage

③ 元数据写入MySQL; 实时节点转存的Segment会在ZooKeeper中新增一条记录

④ 协调节点从MySQL获取元数据,比如schema信息(维度列和指标列)

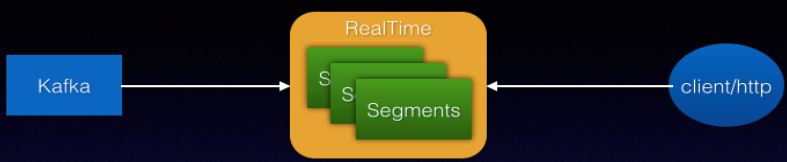
⑤ 协调节点监测ZK中有新分配/要删除的Segment,写入ZooKeeper信息:历史节点需要加载/删除Segment

⑥ 历史节点监测ZK, 从ZooKeeper中得到要执行任务的Segment

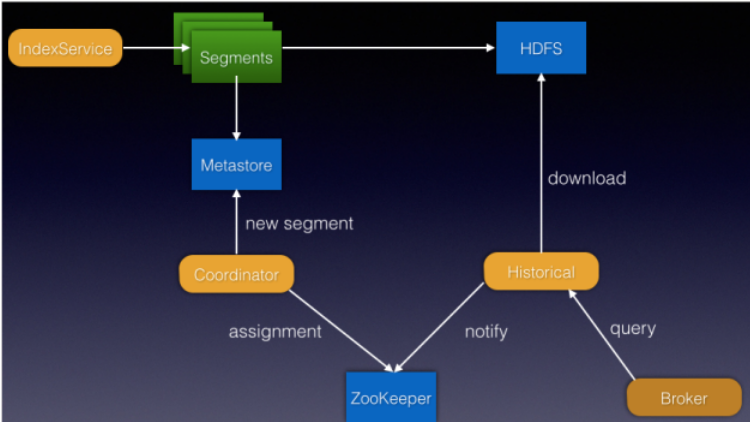
⑦ 历史节点从DeepStorage下载Segment并加载到内存/或者将已经保存的Segment删除掉

⑧ 历史节点的Segment可以用于Broker的查询路由

**3 由于各个节点和其他节点都是最小化解耦的, 所以下面两张图分别表示实时节点和批量数据的流程:**



**4 数据从Kafka导入到实时节点, 客户端直接查询实时节点的数据.**



**5 批量数据使用IndexService,接收Post请求的任务,直接产生Segment写到DeepStorage里.DeepStorage中的数据只会被历史节点使用.。**所以这里要启动的服务有: IndexService(overlord), Historical, Coordinator(协调节点通知历史节点下载Segment)

## 十、参数配置

|  |  |  |
| --- | --- | --- |
| **配置** | **缺省值** | **说明** |
| **name** |  | 配置这个属性的意义在于，如果存在多个数据源，监控的时候可以通过名字来区分开来。  如果没有配置，将会生成一个名字，格式是："DataSource-" + System.identityHashCode(this) |
| **jdbcUrl** |  | 连接数据库的url，不同数据库不一样。例如：  mysql : jdbc:mysql://10.20.153.104:3306/druid2  oracle : jdbc:oracle:thin:@10.20.149.85:1521:ocnauto |
| **username** |  | 连接数据库的用户名 |
| **password** |  | 连接数据库的密码。如果你不希望密码直接写在配置文件中，可以使用ConfigFilter。详细看这里：https://github.com/alibaba/druid/wiki/%E4%BD%BF%E7%94%A8ConfigFilter |
| **driverClassName** | 根据url自动识别 | 这一项可配可不配，如果不配置druid会根据url自动识别dbType，然后选择相应的driverClassName(建议配置下) |
| **initialSize** | 0 | 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时 |
| **maxActive** | 8 | 最大连接池数量 |
| **maxIdle** | 8 | 已经不再使用，配置了也没效果 |
| **minIdle** |  | 最小连接池数量 |
| **maxWait** |  | 获取连接时最大等待时间，单位毫秒。配置了maxWait之后，缺省启用公平锁，并发效率会有所下降，如果需要可以通过配置useUnfairLock属性为true使用非公平锁。 |
| **poolPreparedStatements** | false | 是否缓存preparedStatement，也就是PSCache。PSCache对支持游标的数据库性能提升巨大，比如说oracle。在mysql下建议关闭。 |
| **maxOpenPreparedStatements** | -1 | 要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements自动触发修改为true。在Druid中，不会存在Oracle下PSCache占用内存过多的问题，可以把这个数值配置大一些，比如说100 |
| **validationQuery** |  | 用来检测连接是否有效的sql，要求是一个查询语句。如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会其作用。 |
| **testOnBorrow** | true | 申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。 |
| **testOnReturn** | false | 归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能 |
| **testWhileIdle** | false | 建议配置为true，不影响性能，并且保证安全性。申请连接的时候检测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效。 |
| **timeBetweenEvictionRunsMillis** |  | 有两个含义：  1) Destroy线程会检测连接的间隔时间2) testWhileIdle的判断依据，详细看testWhileIdle属性的说明 |
| **numTestsPerEvictionRun** |  | 不再使用，一个DruidDataSource只支持一个EvictionRun |
| **minEvictableIdleTimeMillis** |  |  |
| **connectionInitSqls** |  | 物理连接初始化的时候执行的sql |
| **exceptionSorter** | 根据dbType自动识别 | 当数据库抛出一些不可恢复的异常时，抛弃连接 |
| **filters** |  | 属性类型是字符串，通过别名的方式配置扩展插件，常用的插件有：  监控统计用的filter:stat日志用的filter:log4j防御sql注入的filter:wall |
| **proxyFilters** |  | 类型是List<com.alibaba.druid.filter.Filter>，如果同时配置了filters和proxyFilters，是组合关系，并非替换关系 |

## 源码解析

1 定义好数据源

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"* init-method=*"init"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driver}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

<property name=*"password"* value=*"${jdbc.password}"*/> <property name=*"initialSize"* value=*"${jdbc.pool.init}"*/>

<property name=*"minIdle"* value=*"${jdbc.pool.minIdle}"*/>

<property name=*"maxActive"* value=*"${jdbc.pool.maxActive}"*/>

</bean>

2 把dataSource接口对象设置进去

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"*/>

<property name=*"typeAliasesPackage"* value=*"com.yunnex.ops.erp"*/>

<property name=*"typeAliasesSuperType"* value=*"com.yunnex.ops.erp.common.persistence.BaseEntity"*/>

<property name=*"mapperLocations"* value=*"classpath:/mappings/\*\*/\*Dao.xml"*/>

<property name=*"configLocation"* value=*"classpath:config/mybatis/mybatis-config.xml"*></property

</bean>

**C3P0总结**

## 一、三种实现方式

**1.自己动手写代码，实现数据源**

例如：在类路径下配置一个属性文件,config.properties，内容如下：

driverClass=xxx

jdbcUrl=xxx

user=xxx

password=xxx

...

然后代码中实现

Properties props = new Properties();

InputStream in = Thread.class.getResourceAsStream("config.properties");

props.load(in);

in.close();

ComboPooledDataSource cpds = new ComboPooledDataSource();

cpds.setDriverClass(props.getProperty("driverClass"));

cpds.setJdbcUrl(props.getProperty("jdbcUrl"));

cpds.setUser(props.getProperty("user"));

cpds.setPassword(props.getProperty("password"));

...

这里实现了一个数据源。

也可以这样配置，在类路径下配置一个xml文件，config.xml

<config>

<source name="source1">

<property name="user">root</property>

<property name="password">xxx</property>

<property name="url">xxx</property>

<property name="driverClass">xxx</property>

</source>

<source name="source2">

...

</source>

</config>

然后自己解析xml文件，这样可以实现多个数据源的配置

**2.配置默认的熟悉文件**

类路径下提供一个c3p0.properties文件(不能改名)

配置如下：

c3p0.driverClass=com.mysql.jdbc.Driver

c3p0.jdbcUrl=jdbc:mysql://localhost:3306/jdbc

c3p0.user=root

c3p0.password=java

...

上面只提供了最基本的配置项，其他配置项参照 [文档](http://www.mchange.com/projects/c3p0/index.html" \l "c3p0-config.xml" \t "http://haoran-10.iteye.com/blog/_blank)配置，记得是c3p0.后面加属性名就是了，最后初始化数据源的方式就是这样简单：

...

DataSource  ds = new ComboPooledDataSource();

return ds;

...

然后就可以使用数据源了，C3P0会对c3p0.properties进行自动解析的

**3.路径下提供一个c3p0-config.xml文件**

这种方式使用方式与第二种差不多，但是有更多的优点   
(1).更直观明显，很类似hibernate和spring的配置  
(2).可以为多个数据源服务，提供default-config和named-config两种配置方式

<c3p0-config>

  <default-config>

    <property name="user">root</property>

    <property name="password">java</property>

    <property name="driverClass">com.mysql.jdbc.Driver</property>

    <property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbc</property>

    <property name="initialPoolSize">10</property>

    <property name="maxIdleTime">30</property>

    <property name="maxPoolSize">100</property>

    <property name="minPoolSize">10</property>

  </default-config>

  <named-config name="mySource">

    <property name="user">root</property>

    <property name="password">java</property>

    <property name="driverClass">com.mysql.jdbc.Driver</property>

    <property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbc</property>

    <property name="initialPoolSize">10</property>

    <property name="maxIdleTime">30</property>

    <property name="maxPoolSize">100</property>

    <property name="minPoolSize">10</property>

  </named-config>

</c3p0-config>

...

DataSource  ds = new ComboPooledDataSource("mySource");

return ds;

...

这样就可以使用数据源了。

## 二、部分参数配置说明

**1.最常用配置**

**initialPoolSize:**连接池初始化时创建的连接数,default : 3（建议使用）

**minPoolSize:**连接池保持的最小连接数,default : 3（建议使用）

**maxPoolSize:**连接池中拥有的最大连接数，如果获得新连接时会使连接总数超过这个值则不会再获取新连接，而是等待其他连接释放，所以这个值有可能会设计地很大,default : 15（建议使用）acquireIncrement:连接池在无空闲连接可用时一次性创建的新数据库连接数,default : 3（建议使用）

**2.管理连接池的大小和连接的生存时间**

**maxConnectionAge:**配置连接的生存时间，超过这个时间的连接将由连接池自动断开丢弃掉。当然正在使用的连接不会马上断开，而是等待它close再断开。配置为0的时候则不会对连接的生存时间进行限制。default : 0 单位 s（不建议使用）

**maxIdleTime:**连接的最大空闲时间，如果超过这个时间，某个数据库连接还没有被使用，则会断开掉这个连接。如果为0，则永远不会断开连接,即回收此连接。default : 0 单位 s（建议使用）

**maxIdleTimeExcessConnections:**这个配置主要是为了快速减轻连接池的负载，比如连接池中连接数因为某次数据访问高峰导致创建了很多数据连接，但是后面的时间段需要的数据库连接数很少，需要快速释放，必须小于maxIdleTime。其实这个没必要配置，maxIdleTime已经配置了。default : 0 单位 s（不建议使用）

**3.配置连接测试：**

**automaticTestTable:**配置一个表名，连接池根据这个表名用自己的测试sql语句在这个空表上测试数据库连接,这个表只能由c3p0来使用，用户不能操作。default : null（不建议使用）

**preferredTestQuery:**与上面的automaticTestTable二者只能选一。自己实现一条SQL检测语句。default : null（建议使用）

**idleConnectionTestPeriod:**用来配置测试空闲连接的间隔时间。测试方式还是上面的两种之一，可以用来解决MySQL8小时断开连接的问题。因为它保证连接池会每隔一定时间对空闲连接进行一次测试，从而保证有效的空闲连接能每隔一定时间访问一次数据库，将于MySQL8小时无会话的状态打破。为0则不测试。default : 0(建议使用)

**testConnectionOnCheckin:**如果为true，则在close的时候测试连接的有效性。default : false（不建议使用）

**testConnectionOnCheckout:**性能消耗大。如果为true，在每次getConnection的时候都会测试，为了提高性能,尽量不要用。default : false（不建议使用）

**4.配置PreparedStatement缓存：**

**maxStatements:**连接池为数据源缓存的PreparedStatement的总数。由于PreparedStatement属于单个Connection,所以这个数量应该根据应用中平均连接数乘以每个连接的平均PreparedStatement来计算。同时maxStatementsPerConnection的配置无效。default : 0（不建议使用）

**maxStatementsPerConnection**:连接池为数据源单个Connection缓存的PreparedStatement数，这个配置比maxStatements更有意义，因为它缓存的服务对象是单个数据连接，如果设置的好，肯定是可以提高性能的。为0的时候不缓存。default : 0（看情况而论）

**5.重连相关配置**

**acquireRetryAttempts:**连接池在获得新连接失败时重试的次数，如果小于等于0则无限重试直至连接获得成功。default : 30（建议使用）

**acquireRetryDelay:**连接池在获得新连接时的间隔时间。default : 1000 单位ms（建议使用）

**breakAfterAcquireFailure:**如果为true，则当连接获取失败时自动关闭数据源，除非重新启动应用程序。所以一般不用。default : false（不建议使用）

**checkoutTimeout:**配置当连接池所有连接用完时应用程序getConnection的等待时间。为0则无限等待直至有其他连接释放或者创建新的连接，不为0则当时间到的时候如果仍没有获得连接，则会抛出SQLException。其实就是acquireRetryAttempts\*acquireRetryDelay。**default :** 0（与上面两个，有重复，选择其中两个都行）

**6.定制管理Connection的生命周期**

**connectionCustomizerClassName:**用来定制Connection的管理，比如在Connection acquire 的时候设定Connection的隔离级别，或者在Connection丢弃的时候进行资源关闭，

就可以通过继承一个AbstractConnectionCustomizer来实现相关方法，配置的时候使用全类名。有点类似监听器的作用。default : null（不建议使用）

**7.配置未提交的事务处理**

**autoCommitOnClose:**连接池在回收数据库连接时是否自动提交事务。如果为false，则会回滚未提交的事务，如果为true，则会自动提交事务。default : false（不建议使用）

**forceIgnoreUnresolvedTransactions：**这个配置强烈不建议为true。default : false（不建议使用）一般来说事务当然由自己关闭了，为什么要让连接池来处理这种不细心问题呢？

8.配置debug和回收Connection

**unreturnedConnectionTimeout:**为0的时候要求所有的Connection在应用程序中必须关闭。如果不为0，则强制在设定的时间到达后回收Connection，所以必须小心设置，保证在回收之前所有数据库操作都能够完成。这种限制减少Connection未关闭情况的不是很适用。建议手动关闭。default : 0 单位 s（不建议使用）

**debugUnreturnedConnectionStackTraces:**如果为true并且unreturnedConnectionTimeout设为大于0的值，当所有被getConnection出去的连接unreturnedConnectionTimeout时间到的时候，就会打印出堆栈信息。只能在debug模式下适用，因为打印堆栈信息会减慢getConnection的速度default : false（不建议使用）

**9.其他配置项：**因为有些配置项几乎没有自己配置的必要，使用默认值就好，所以没有再写出来。

## 代码示例

**1.c3p0.properties**

#驱动

c3p0.driverClass=com.mysql.jdbc.Driver

#地址

c3p0.jdbcUrl=jdbc:mysql://localhost:3306/jdbc

#用户名

c3p0.user=root

#密码

c3p0.password=lovejava

#-------------------------------

#连接池初始化时创建的连接数

c3p0.initialPoolSize=3

#连接池保持的最小连接数

c3p0.minPoolSize=3

#连接池在无空闲连接可用时一次性创建的新数据库连接数,default:3

c3p0.acquireIncrement=3

#连接池中拥有的最大连接数，如果获得新连接时会使连接总数超过这个值则不会再获取新连接，而是等待其他连接释放，所以这个值有可能会设计地很大,default : 15

c3p0.maxPoolSize=15

#连接的最大空闲时间，如果超过这个时间，某个数据库连接还没有被使用，则会断开掉这个连接,单位秒

c3p0.maxIdleTime=100

#连接池在获得新连接失败时重试的次数，如果小于等于0则无限重试直至连接获得成功

c3p0.acquireRetryAttempts=30

#连接池在获得新连接时的间隔时间

c3p0.acquireRetryDelay=1000

**2.ConnectionPool**

package com.study.pool;

import java.sql.Connection;

import java.sql.SQLException;

import javax.sql.DataSource;

import com.mchange.v2.c3p0.ComboPooledDataSource;

public class ConnectionPool {

    private DataSource ds;

    private static ConnectionPool pool;

    private ConnectionPool(){

        ds = new ComboPooledDataSource();

    }

    public static final ConnectionPool getInstance(){

        if(pool==null){

            try{

                pool = new ConnectionPool();

            }catch (Exception e) {

                e.printStackTrace();

            }

        }

        return pool;

    }

    public synchronized final Connection getConnection() {

        try {

            return ds.getConnection();

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return null;

    }

}

**3.PoolThread**

 package com.study.pool;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class PoolThread extends Thread {

    @Override

    public void run(){

        ConnectionPool pool = ConnectionPool.getInstance();

        Connection con = null;

        PreparedStatement stmt= null;

        ResultSet rs = null;

        try{

            con = pool.getConnection();

            stmt = con.prepareStatement("select sysdate as nowtime from dual");

            rs = stmt.executeQuery();

            while(rs.next()){

                System.out.println(Thread.currentThread().getId()+"---------------开始"+rs.getString("nowtime"));

            }

        } catch (Exception e) {

            e.printStackTrace();

        }finally{

            try {

                rs.close();

                stmt.close();

                con.close();

            } catch (SQLException e) {

                e.printStackTrace();

            }

        }

        System.out.println(Thread.currentThread().getId()+"--------结束");

    }

}

**4.PoolMain**

 package com.study.pool;

public class PoolMain {

    /\*\*

     \* 数据源缓冲池 实例练习

     \*/

    public static void main(String[] args) {

        System.out.println("缓冲池模拟开始");

        PoolThread[] threads = new PoolThread[50];

        for(int i=0;i<threads.length;i++){

            threads[i] = new PoolThread();

        }

        for(int i=0;i<threads.length;i++){

            threads[i].start();

        }

    }

}

## 四、c3p0原理解析

**1 c3p0的出现，是为了大大提高应用程序和数据库之间访问效率的。**说到c3p0，不得不说一下jdbc本身，c3p0愿意就是对数据库连接的管理，那么原有的概念还是得清晰：DriverManager、Connection、StateMent、ResultMent。

**2 jdbc**：java database connective这套API，不用多说，是一套用于连接各式dbms或连接桥接器的api，两个层级：上层供应用方调用api，下层，定义了各个dbms的spi的api（具体文档见：[这里](http://www.oracle.com/technetwork/java/javase/jdbc/index.html" \o "jdbc" \t "https://www.cnblogs.com/leeying/p/_blank)）。

**3 主要要提的是：**datasource、DriverManager，想到哪儿写到哪儿，datasource是更高级一点的api，原因在于相对对应用来说更透明。

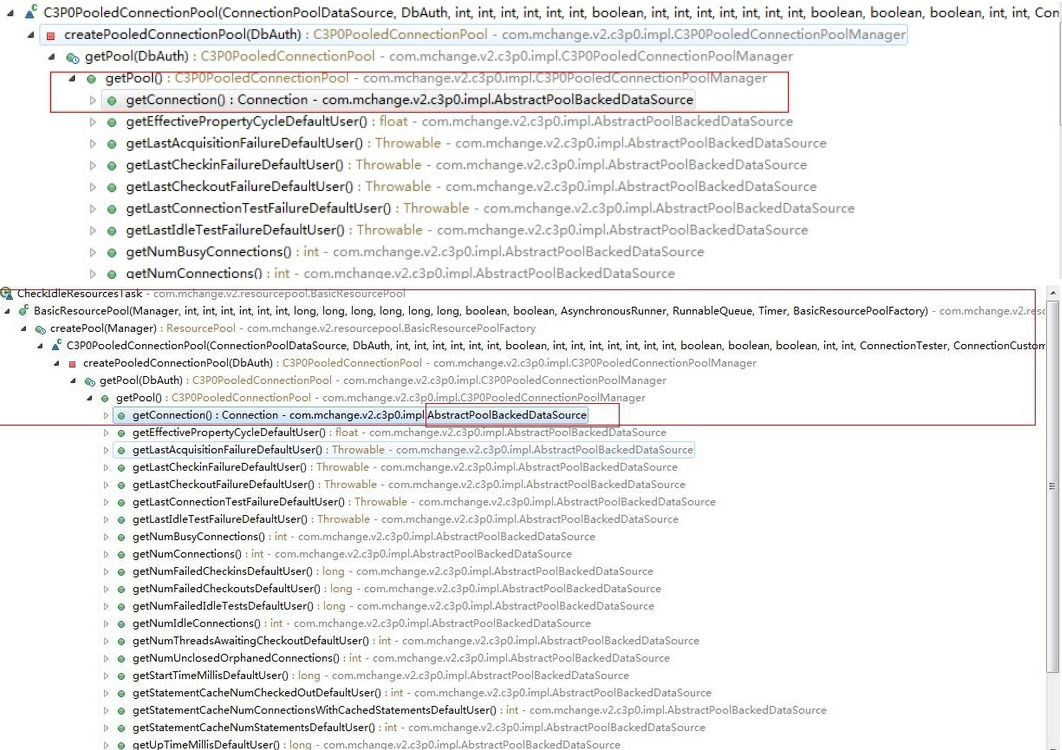
**4 Connection：**同dbms的逻辑链接，类似于session管理概念， SQL statements are executed and results are returned within the context of a connection。jdbc的概念就到这里，平时用得比较多。

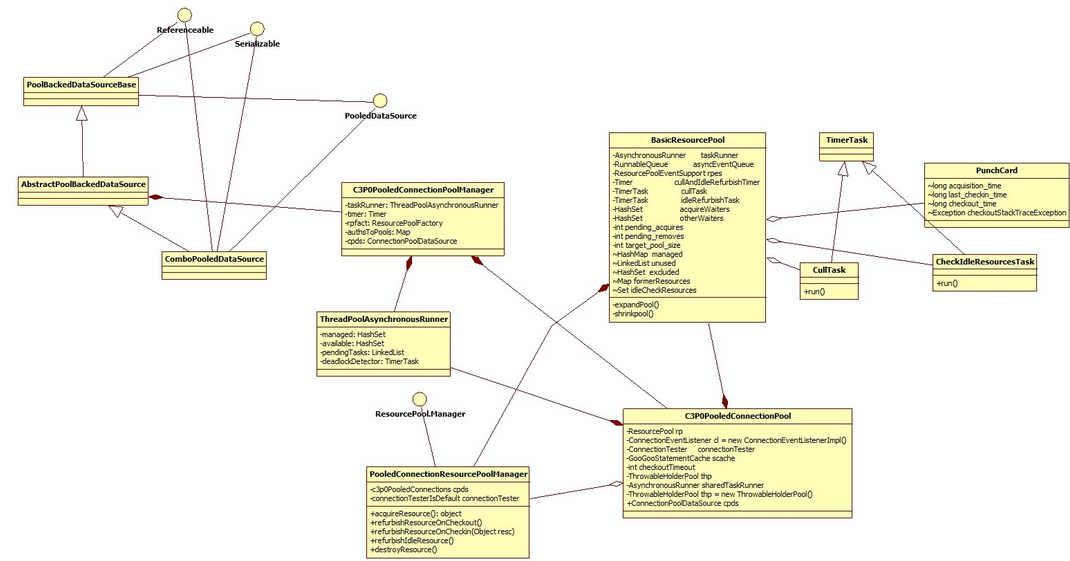
## 五、c3p0的bean配置

**1 配置一览**

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close"> 2 <property name="driverClass" value="${jdbc.driverClassName}" /> 3 <property name="jdbcUrl" value="${jdbc.url}" /> 4 <property name="user" value="${jdbc.username}" /> 5 <property name="password" value="${jdbc.password}" /> 6 <property name="checkoutTimeout" value="30000" /> 7 <property name="maxPoolSize" value="15" /> 8 <property name="idleConnectionTestPeriod" value="180" /> 9 <property name="maxIdleTime" value="180" />10 </bean>

**2 ComboPooledDataSource，该类结构如下**





**3 以上类图都不是很完全，不过大体能表达出类之间的原理**

（1）、bean：ComboPooledDataSource的父类：AbstractPoolBackedDataSource有一个poolmanager字段，存储着对pool管理器

（2）、获取ds.getConnection（）链接对象时，内部使用getPoolManger()获取C3p0ConnectionPooledManager（mgr）对象，该manager管理着pool对象：C3P0PooledConnectionPool对象，mgr.getPool().checkoutPooledConnection()

（3）、自此该connection已经被获取到了

（4）、让我们看看该connection的真实面目吧：ProxyConnection。

（5）、因此其实原理是：

**从pool里获取到的connection，是proxy包装的connection，而对connection的释放或者重用，是pool的管理责任：初始化池大小，维护池的大小（expand或shrink），管理unused、expired、checkout、checkin连接。真正底层的连接是jdbc自己的连接，而c3p0的管理部分，基本上使用的是synchronized关键字，使用timerTask定时器工作。**