c3p0

version 0.9.1.2

中文文档

Steve Waldman

PSJay 译

目录

快速开始

c3p0 被设计地非常容易使用。你只需要把[lib/c3p0-0.9.1.2.jar]这个 jar 文件放入有效的 CLASSPATH，然后像这样创建数据源：

```
import com.mchange.v2.c3p0.*;

...

ComboPooledDataSource cpds = new ComboPooledDataSource();
cpds.setDriverClass( "org.postgresql.Driver" ); //loads the jdbc driver
cpds.setJdbcUrl( "jdbc:postgresql://localhost/testdb" );
cpds.setUser("dbuser");
cpds.setPassword("dbpassword");
```

[**可选的**] 如果你需要开启 PreparedStatement 池，你必须设置 *maxStatements* 和/或 *maxStatementsPerConnection* 两个值（默认都为 0）。

```
cpds.setMaxStatements( 180 );
```

你可以使用这个数据源做你要做的工作，它底层的连接池将会使用默认的参数来创建。你可以根据你的喜好将这个数据源绑定一个 JNDI 命名空间或者直接的使用它。

当你完成工作的时候，你可以像这样将你所创建的数据源销毁：

```
DataSources.destroy( cpds );
```

就这么简单！接下来就要进行详细的介绍了。

c3p0 是什么？

c3p0 是一个容易使用的，将 jdbc3 规范和 jdbc2 中的扩展的功能扩充到传统的 JDBC 驱动，使其“企业就绪（enterprise-ready）”的类库。特别的，c3p0 提供了一系列有用的服务：

- 提供一些用于获取数据库连接的类，这些类兼容于传统的基于驱动管理器（DriverManager-based）的 JDBC 驱动和新的 *javax.sql.DataSource* 视图。
- 透明地池化管理数据库连接和 PreparedStatement，数据源包装了传统的驱动，也可以任意地使其脱离池的管理。

还有几点需要强调：

- c3p0 数据源是*可引用*（Referenceable）并且*可序列化*（Serializable）的，因此它很适合绑定大量的基于 JNDI 的命名服务。
- Statement 和 ResultSet 会随着被池管理的数据库连接或者 Statement 的失效而被清理。这是为了防止因用户使用延迟策略（用户常用的资源管理策略），仅仅清理数据库连接而造成的资源紧缺现象。
- c3p0 与定义在 JDBC 2 和 JDBC 3 中的方法都兼容(尽管这与类库作者的喜好有冲突)。数据源是用 JavaBean 风格写的，开放了所有的必须属性和大部分的可选属性（还有一些不标准的属性），提供了无参数的构造方法。所有 JDBC 中定义的内部接口均被实现了（*ConnectionPoolDataSource, PooledConnection, ConnectionEvent-generating Connections,* 等等）。你可以将 c3p0 的类和与其兼容的第三方实现一起使用（虽然并不是所有的 c3p0 特性都能够起作用）。

c3p0 希望能够提供一个优秀的 J2EE 企业级应用的数据源实现。希望您能给我们一些建议，或者帮助我们修复漏洞，等等！

必备环境

c3p0 需要 1.3.x 或者更高版本的 Java 运行环境，JDBC 2.x 或更高版本的 javax.sql 类库。c3p0 在 Java 1.4.x 和 Java 1.5.x 下工作得不错。

安装

将 *lib/c3p0-0.9.1.2.jar* 文件放到你的 CLASSPATH 下（或者你的类加载器能够加载到的其他任何地方）。就这么简单！

使用 c3p0

从普通用户的角度来看，c3p0 只是提供了标准的 jdbc2 数据源对象。当创建这些数据源对象时，用户可以控制与池有关的（pooling-related）、与命名有关的（naming-related）及其他一些属性（参见附录 A）。数据源对象被创建之后，所有的池管理操作对用户是完全透明的。

有三种方式创建 c3p0 数据源：1)直接实例化、配置一个 *ComboPooledDataSource* Bean；2)使用数据源工厂类；或者 3)创建你自己的具有池技术的数据源：直接实例化 *PoolBackedDataSource* 并设定它的 *ConectionPoolDataSource*。大部分用户可能会觉得第一种方式最方便。一旦实例化成功，c3p0 数据源就可以和与 JNDI 标准兼容的命名服务绑定。

不管你使用哪种方式，如果你不指定配置的话，c3p0 将会使用默认的参数来创建数据源。c3p0 内置了硬编码的配置，但是你可以通过创建一个 *c3p0.properties* 文件并将其放在与加载 c3p0 的 CLASSPATH（或加载器）相同的顶级目录中来覆盖配置信息。在 c3p0-0.9.1 之后，你也可以创建 *c3p0-config.xml* 文件来进行一些更高级的配置。请查看下面的配置章节。

实例化并配置 ComboPooledDataSource

也许创建 c3p0 池数据源的最直接方式就是实例化一个 *com.mchange.v2.c3p0.ComboPooledDataSource* 。这是一个 JavaBean 风格的，拥有一个无参数的 public 构造方法的类，但当你使用它之前，你应该确保你设定了 *jdbcUrl* 属性。你很可能也需要设定 user 和 password 属性。如果你还没有在外部预加载老式的 JDBC 驱动的话，你还应该设定 *driverClass* 属性。

```
ComboPooledDataSource cpds = new ComboPooledDataSource();
cpds.setDriverClass( "org.postgresql.Driver" ); //loads the jdbc driver
cpds.setJdbcUrl( "jdbc:postgresql://localhost/testdb" );
cpds.setUser("swaldman");
cpds.setPassword("test-password");

// the settings below are optional -- c3p0 can work with defaults
cpds.setMinPoolSize(5);
cpds.setAcquireIncrement(5);
cpds.setMaxPoolSize(20);

// The DataSource cpds is now a fully configured and usable pooled DataSource

...
```

任何一个 c3p0 DataSource 实例的最初状态都取决于你提供的配置，或者仅仅是硬编码的默认配置[查看属性配置]。c3p0-0.9.1 之后支持了混合的，命名的配置（multiple, named configurations）。如果你想使用命名配置，可以用一个命名配置名称作为参数来构造

```
ComboPooledDataSource cpds = new ComboPooledDataSource("intergalactoApp");
```

com.mchange.v2.c3p0.ComboPooledDataSource：

当然，你仍然可以像上面提到的那样，用编程的方法来覆盖任何配置信息。

使用数据源工厂类

另外，你可以使用静态工厂类 com.mchange.v2.c3p0.DataSources 来从传统的 JDBC 驱动里创建没有池的数据源，然后用这个数据源来构造一个池化了的数据源：

```
DataSource ds_unpooled = DataSources.unpooledDataSource("jdbc:postgresql://localhost/testdb",
                                                         "swaldman",
                                                         "test-password");
DataSource ds_pooled = DataSources.pooledDataSource( ds_unpooled );

// The DataSource ds_pooled is now a fully configured and usable pooled DataSource.
// The DataSource is using a default pool configuration, and Postgres' JDBC driver
// is presumed to have already been loaded via the jdbc.drivers system property or an
// explicit call to Class.forName("org.postgresql.Driver") elsewhere.

...
```

如果你使用这个数据源工厂类，并且你想编程式地覆盖默认的配置，你可以提供一个 map 来覆盖属性：

```
DataSource ds_unpooled = DataSources.unpooledDataSource("jdbc:postgresql://localhost/testdb",
                                                         "swaldman",
                                                         "test-password");

Map overrides = new HashMap();
overrides.put("maxStatements", "200");          //Stringified property values work
overrides.put("maxPoolSize", new Integer(50)); //"boxed primitives" also work

// create the PooledDataSource using the default configuration and our overrides
ds_pooled = DataSources.pooledDataSource( ds_unpooled, overrides );

// The DataSource ds_pooled is now a fully configured and usable pooled DataSource,
// with Statement caching enabled for a maximum of up to 200 statements and a maximum
// of 50 Connections.

...
```

如果你使用命名配置，你就可以指定一个命名来配置你的数据源：

```
// create the PooledDataSource using the a named configuration and specified overrides
ds_pooled = DataSources.pooledDataSource( ds_unpooled, "intergalactoAppConfig", overrides );
```

罕见情况：强制使用认证信息，忽略底层的非池化数据源的配置信息

你可以用 *DataSource.pooledDataSource(...)* 方法来包装任何数据源，通常这没有什么问题。数据源应该能够通过数据库连接的标准的 *user* 和 *password* 属性来自动判别用户名和密码。有些数据源实现没有提供这些属性。通常这也一点儿都不成问题。c3p0 如果不能找到默认的认证信息，用户也没有通过 *getConnection(user, password)* 方法来指定认证信息的话，c3p0 将通过获取默认连接来解决这个问题。

然而，有一些非常罕见的情况，比如非 c3p0 的非池化数据源提供了一个 *user* 属性，但是没有提供 *password* 属性。或者有时你想将数据源池化并覆盖内置的认证信息但又不想修改它的 *user* 和 *password* 属性。

c3p0 提供了 *overrideDefaultUser* 和 *overrideDefaultPassword* 两个配置属性。如果你通过编程式的或者通过其他任何 c3p0 的配置机制设定了这些属性，c3p0 *PooledDataSources* 将忽略底层数据源的用户名和密码，转而用这两个配置信息来替代。

查询池数据源的当前状态

c3p0 数据源依赖着一个池，它包括了 *ComboPooledDataSource* 的实现和 *DataSources.pooledDataSource(...)* 方法返回的对象以及 com.mchange.v2.c3p0.PooledDataSource 接口的所有实现，这使得有大量查询数据源连接池状态的方法可用。下面有一段查询数据源状态的示例代码：

```
// fetch a JNDI-bound DataSource
InitialContext ictx = new InitialContext();
DataSource ds = (DataSource) ictx.lookup( "java:comp/env/jdbc/myDataSource" );

// make sure it's a c3p0 PooledDataSource
if ( ds instanceof PooledDataSource)
{
   PooledDataSource pds = (PooledDataSource) ds;
   System.err.println("num_connections: "           + pds.getNumConnectionsDefaultUser());
   System.err.println("num_busy_connections:                    "                    +
pds.getNumBusyConnectionsDefaultUser());
   System.err.println("num_idle_connections: " + pds.getNumIdleConnectionsDefaultUser());
   System.err.println();
}
else
   System.err.println("Not a c3p0 PooledDataSource!");
```

这些状态查询方法有三种重载方式，比如：

- *public int getNumConnectionsDefaultUser()*
- *public int getNumConnections(String username, String password)*
- *public int getNumConnectionsAllUsers()*

c3p0 为不同认证信息的连接维护着不同的连接池。这些方法能够让你逐个地查询这些连接池的状态，或者统计出数据源所维护的所有连接池的状态。请注意连接池配置参数，比如 *maxPoolSize* 是对于每个认证信息来设定的！举个例子，比如你设定了 *maxPoolSize* 的值为 20，如果数据源管理的连接在两套用户名-密码之下[一套默认的，和另外一套通过调用 *getConnection(user, password)*方法建立的]，你应该要知道 *getNumConnectionsAllUsers()* 会返回 40。

大多数的应用只从数据源中获取默认认证信息的数据库连接，并且通常可以使用 getXXXDefaultUser() 来统计所有的连接。

与统计跟连接池有关的状态一样，你也可以获取每个数据源的线程池的状态信息。请参阅池化数据源的可用操作的完整列表。

使用 C3P0Registry 来获得数据源的引用

如果不方便或不可能通过 JNDI 或其他方法获得数据源的引用的话，你可以通过使用 *C3P0Registry* 类来找到所有可用的数据源。这个类通过三个静态方法来帮助你摆脱困境：

- *public static Set getPooledDataSources()*
- *public static Set pooledDataSourcesByName(String dataSourceName)*
- *public static PooledDataSource pooledDataSourceByName(String dataSourceName)*

第一个方法将会返回给你所有的可用的 c3p0 PooledDataSources。如果你确定你的应用只会创建一个 PooledDataSources，或者你可以通过配置信息（用“getters”检测）来区分数据源的话，第一个方法对你来说足矣。因为情况并不总是这样，c3p0 PooledDataSources 有一个特殊性属性，名字为 *dataSourceName*。你可以在构造数据源对象时直接设定这个属性，也可以像其他属性那样通过一个命名配置或默认配置来设置。不然的话，*dataSourceName* 属性的默认值是这样生成的：1)如果你使用了一个命名配置来构造数据源，*dataSourceName* 的值就是这个命名配置的名称；或者，2)如果你使用了默认的配置，*dataSourceName* 会是一个唯一（但是变幻莫测并且不可确保唯一性）的值。

无法保证 *dataSourceName* 的值是唯一的。比如，如果两个 c3p0 数据源共用同一个命名配置，并且你没有编程式地设定 *dataSourceName* 的值，那么这两个数据源的 *dataSourceName* 就会相同，即这个命名配置的名称。使用 *pooledDataSourcesByName(…)*方法可以得到特定的 *dataSourceName* 的所有数据源。如果你确保你的数据源的名称是唯一的（这通常是你想要的，如果你打算用 C3P0Registry 来找到你的数据源的话），你可以使用 *pooledDataSourceByName()*这个更方便的方法来直接得到你需要的数据源（或者当找不到对应的数据源时返回 null）。如果你使用 *pooledDataSourceByName(…)*，但是数据源名不唯一的话，返回哪个数据源对象是不确定的。

清理 c3p0 PooledDataSources

最简单的清理 c3p0 PooledDataSources 的方法就是调用 *DataSources* 类的 *destroy* 方法。只有 PooledDataSource 对象需要被清理，而 DataSources 不需要。对非池化或非 c3p0 的数据源调用 destory(…)方法也没有什么影响。

```
DataSource ds_pooled     = null;

try
{
   DataSource                                    ds_unpooled                                    =
DataSources.unpooledDataSource("jdbc:postgresql://localhost/testdb",
                                                                                  "swaldman",
                                                                                  "test-password");
   ds_pooled = DataSources.pooledDataSource( ds_unpooled );

   // do all kinds of stuff with that sweet pooled DataSource...
}
finally
{
   DataSources.destroy( ds_pooled );
}
```

另外，c3p0 的 *PooledDataSource* 接口有一个 *close()*方法，你可以调用它来关闭 *DataSource* 对象。所以你可以通过这个方法来使把 *DataSource* 对象转换成一个 *PooledDataSource* 对象然后关闭它。

```
static void cleanup(DataSource ds) throws SQLException
{
   // make sure it's a c3p0 PooledDataSource
   if ( ds instanceof PooledDataSource)
   {
      PooledDataSource pds = (PooledDataSource) ds;
      pds.close();
   }
   else
      System.err.println("Not a c3p0 PooledDataSource!");
}
```

未被引用的 *PooledDataSource* 实例不会在被用户关闭它之前通过垃圾回收器调用它的 *finalize()*而被关闭。就像我们知道的那样，自动销毁机制应该只能成为一种辅助方法，而不要用其来确保资源的清理。

进阶：创建你自己的 PoolBackedDataSource

对大多数程序员来说并没有太多理由这样做，但是你的确可以通过这种方法来一步步地创建一个 *PooledDataSource*：实例化并且配置一个非池化的 *DriverManagerDataSource*，实例化一个 *WrapperConnectionPoolDataSource*，并且将那个非池化的数据源设置为它的 *nestedDataSource* 属性，然后将它设定为一个 *PoolBackedDataSource* 对象的 *connectionPoolDataSource* 属性。

如果你的驱动提供了 *ConnectionPoolDataSource* 的实现，上面的这一系列步骤是你主要要关心的。除了使用 c3p0 的 *WrapperConeectionPoolDataSource* 之外，你还可以创建一个 *PollBackedDataSource*，然后设定它的 *connectionPoolDataSource* 属性。第三方的 *ConnectionPoolDataSource* 不支持 Statement 池，*ConnectionCustomizers*，和一些特定的 c3p0 属性（第三方的 *DataSource* 实现可以用来替代 c3p0 的 *DriverManagerDataSource*，这在功能上并没有什么明显的损失）。

进阶：原生连接（Raw Connetion）和 Statement 操作

JDBC 驱动在实现 Connection 和 Statement 上有时定义了一些与特定厂商的，不标准的 API。c3p0 将这些对象包装进了一个代理里，所以你不能把 c3p0 返回的 Connection 或 Statement 转换成特定厂商的实现。c3p0 没有提供任何能够直接访问原生连接和 Statement 的方法，这是因为 c3p0 需要一直按顺序地追踪 Statement 和结果集的创建，来防止资源紧缺和池讹误（pool corruption）现象。

c3p0 的确提供了一个能让你通过反射在底层的连接上来执行非标准方法的 API。使用方法是，首先将返回的 Connection 转换成一个 *C3P0ProxyConnection*。然后调用 *rawConnectionOperation* 方法，然后提供一个你想要执行的 *java.lang.reflect.Method* 对象作为这个非标准方法的参数。你提供的这个 Method 对象将会在第二个参数（null 或者静态方法）上执行，并且使用你提供的第三个参数来完成这个方法。对目标对象和这个方法的所有参数来讲，你都可以使用 *C3P0ProxyConnection.RAW_CONNECTION* 这个特殊的标记，它将会在 Method 执行前替代底层的特定厂商的 Connection。

C3P0ProxyStatement 也提供了类似的 API。

原生操作返回的所有 Statement（包括 Prepared 和 CallableStatement）和 ResultSet 都是被 c3p0 所管理的，所以在其父代理连接上调用 *close()* 方法的时候它们都会被正确地清理。用户必须注意清理那些与特定厂商相关的方法返回的那些非标准的资源。

这里有一个使用 Oracle 特定的 API 在原生连接上调用静态方法的例子：

```
C3P0ProxyConnection castCon = (C3P0ProxyConnection) c3p0DataSource.getConnection();
Method m = CLOB.class.getMethod("createTemporary", new Class[]{Connection.class, boolean.class, int.class});
Object[] args = new Object[] {C3P0ProxyConnection.RAW_CONNECTION, Boolean.valueOf( true ), new Integer( 10 )};
CLOB oracleCLOB = (CLOB) castCon.rawConnectionOperation(m, null, args);
```

注意:C3P0 现在支持了一些 Oracle 特定的方法了，见附录 F。

配置

简介

c3p0 没有过多的*必须*（required）配置信息，它非常灵活可控。大多数的配置信息都是 JavaBean 属性。下面就是 JavaBean 的惯例，如果有一个对象有一个类型为 *T* 的属性 *foo*,这个对象就会有类似这样的方法…

public T getFoo();
public void setFoo(T foo);

这样的方法是否成对存在，取决于这个属性是只可读的，只可写，还是可读写的。

有多种方法修改 c3p0 的属性：你可以直接在代码里通过关联一个特定的数据源来改变配置，你也可以在外部通过简单的 Java 属性文件（simple Java properties file），XML 配置文件或系统属性来配置 c3p0。通常来讲，配置文件都被命名为 *c3p0.properties* 或 *c3p0-config.xml*，并放置在应用的 CLASSPATH 的顶级路径中，但是 XML 配置文件可以被放在应用所在的文件系统的任何位置，你只需要改变 com.mchange.v2.c3p0.cfg.xml 这个系统属性即可（绝对路径）。

数据源通常在使用之前被配置，比如在构造方法中配置或者在构造方法返回之后马上进行配置。不过 c3p0 也支持使用过程中修改配置。

如果你通过调用工具类 *com.mchange.v2.c3p0.DataSources* 的工厂方法来获取数据源，并且不希望这个数据源使用默认的配置，那么你可以提供一个 Map 类型的参数作为配置信息[key 必须是小写字母开头的属性名，value 可以是字符串或者“被打包（boxed）”的 Java 原生类型，例如 Integer 和 Boolean]。

所有可改变的属性在附录 A 中都有详细的文档。下面我们来讨论那些最常用，最重要

的 c3p0 配置。

连接池的基本配置

c3p0 连接池可以通过下面几个基本参数来简单地配置：

- *acquireIncrement*
- *initialPoolSize*
- *maxPoolSize*
- *maxIdleTime*
- *minPoolSize*

initialPoolSize, *minPoolSize*, *maxPoolSize* 定义了由池管理的连接数量。请确保 *minPoolSize* <= *maxPoolSize*。不合理的 *initialPoolSize* 值将会被忽略，然后使用 *minPoolSize* 来替代。

在 *minPoolSize* 和 *maxPoolSize* 的范围之内，池中的连接数量与使用模式（usage patterns）有关。当用户请求一个连接，池中又没有可用连接，并且池中的连接数量还未达到 *maxPoolSize* 的时候，池中的连接数就会增长。因为获取连接非常慢，所以成批地增加连接数量通常都很有效，而不是强制要用户在需要新连接时从头开始激活并获取一个连接。*acquireIncrement* 属性决定了 c3p0 在没有可用连接时一次性获取的新连接数量。（不过 c3p0 绝不会因此而使连接数超过 *maxPoolSize* 值。）

当连接池测试一个连接并且发现它已失效（broken）（参考下面的配置连接测试），或者当一个连接因空闲期超过一段时间而过期，或太旧（too old）（参考管理池尺寸和连接寿命）的时候，连接池中的连接数量将会下降。

管理连接池尺寸与连接寿命

不同的应用在对性能、内存占用和可靠性等方面有不同的权衡需求。C3P0 提供了大量的选项来控制池内连接数的增长或减小的速度，也可以用一些选项来决定“旧”连接是否应该主动地被替换以维持应用的可靠性。

- *maxConeectionAge*
- *maxIdleTime*
- *maxIdleTimeExcessConeections*

默认情况下，连接池不会给连接设定过期时间。如果你为了保持连接“新鲜”，想要给连接设定过期时间的话，需要设定 *maxIdleTime* 和/或 *maxConnectionAge*。*maxIdleTime* 定义了连接因在多少秒内未被使用而被连接池剔除的时间。*maxConnectionAge* 决定了所有从数据库中获取的连接将在多少秒后被连接池剔除。

maxIdleTimeExcessConnections 用来最小化 c3p0 欠载（under load）时的连接数。默认情况下，在 c3p0 欠载时，c3p0 只会因连接测试失败或者连接过期而缩小池中的连接数。有一些用户需要在一些突然增大会连接数的操作之后快速地释放不必要的连接。你可以通过把 *maxIdleTimeExcessConnetions* 设定为一个比 *maxIdleTime* 小得多的值来达到这个目的。超过最小连接数的那些连接会在较小的一段空闲时间之后被连接池剔除。

对设置这类超时参数有一些普通的建议：悠着点！使用连接池的重点就是尽量只从数据库中获取连接一次，然后不断重复地使用它们。大多数的数据库的连接可以一次维持若干小时。没有必要每隔几秒钟或几分钟就剔除那些空闲的连接。把 *maxConnectionAge* 或 *maxIdleTime* 设置成 1800（30 分钟）都是有些激进的。对于大多数数据库来讲，几个小时或许更加合理。你可以用连接测试来确保可靠性，而不是一味地剔除它们（见配置连接测试）。通常只有 *maxIdleTimeExcessConnetions* 这个参数可以被设置成几分钟或更短的时间。

配置连接测试

c3p0 的连接测试可以用来最小化你的应用遇到失效或过时的连接的可能性，它可以用多种方式来进行配置。池中的连接可能会因为各种原因而变得不可用——有些 JDBC 驱动有意地对长连接设置了超时参数；后端的数据库或网络有时候会宕掉；有些连接仅仅是因为资源紧缺，驱动的漏洞或者其他原因而变得不可用。

c3p0 通过以下参数为用户提供了灵活的测试连接的方法：

- *automaticTestTable*
- *connectionTesterClassName*
- *idleConnectionTestPeriod*
- *preferredTestQuery*
- *testConnectionOnCheckin*
- *testConnectionOnChechout*

idleConnetionTestPeriod，*testConnectionOnCheckout* 和 *testConnectionOnCheckin* 决定了连接何时被测试。*automaticTestTable*，*connectionTesterClassName* 和 *preferredTestQuery* 决定了连接怎样被测试。

当配置连接测试的时候，首先应该考虑如何减少测试的开支。默认情况下，连接通过在与其关联的 *DatabaseMetaData* 对象上调用 *getTables()*方法来进行测试。这对所有数据库来讲都有效，因为这与数据库的视图（database schema）无关。然而，从经验上来讲，调用 *DatabaseMetaData.getTables()*方法相对于进行一个简单的数据库查询要慢多了。

提高连接测试速度的最方便的方法就是定义 *automaticTestTable* 属性。c3p0 将会使用你提供的名字创建一个空的表，然后通过一个简单的查询来测试数据库。你也可以通过设定 *preferredTestQuery* 参数来定义一个测试语句。不过你得当心点，设置 *preferredTestQuery* 将会导致*在初始化数据源之前*出现错误，如果你查询的目标表不存在的话。

高级用户可以实现一个 *ConnectionTester* 然后提供一个类的全限定名作为 *connectionTesterClassName* 属性来实现任何想要的连接测试。如果你想要使你的 *ConnectionTester* 能够支持 *preferredTestQuery* 和 *automaticTestTable* 属性，实现 *UnifiedConnectionTester* 接口即可，实现 *AbstractConnectionTester* 是最方便的。更多信息见 API 文档。

检测连接最可靠的时间就是从池中取出连接的时候（check-out）。但从客户端性能的角度来看，是这也是开销最大的。大多数应用将 *idleConnectionTestPeriod* 和 *testConnectionOnCheckIn* 结合起来用就已经非常可靠了。空闲测试和将连接放回池时（check-in）的测试都是异步执行的，这就是为什么它们有更好的性能的原因。

注意，对有些应用程序来说，拥有高性能远比避免偶然发生的数据库异常更重要。默认情况下，c3p0 不会做任何连接测试。设置一个非常长的 *idleConnectionTestPeriod* 值和避免 check-out 与 check-out 测试是很不错的，高性能的方法。

配置 Statement 池

c3p0 实现了符合 JDBC 规范的透明的 *PreparedStatement* 池。在一些情况下，Statement 池能够显著地提高应用程序的性能。但在另一些情况下，Statement 池的开销又会稍微的降低性能。statement 到底能不能改善性能或者能够改善多少性能还是取决于你的数据库对查询的解析，规划和优化（parsing, planning , and optimizing）。不同数据库（和 JDBC 驱动）之间在这个方面存在很大的差异。给你的应用程序在使用和不使用 statement 池的时候设定基准，然后比较它们来看看究竟 statement 能不能改善性能是个不错的点子。

你可以通过设置下面的配置参数来配置 statement 池：

- *maxStatement*
- *maxStatementPerConnection*

maxStatement 是 JDBC 规范的标准参数。*maxStatement* 定义了每个数据源会缓存的 *PreparedStatement* 的总数。池内的 Statement 总数在达到这个限制时会销毁那些最近最近最少使用的（least-recently-used）Statement。这听起来很简单，不过事实上有些奇怪，因为从概念上来讲，被缓存的 Statement 是属于单个的数据库连接的，它们并不是全局资源。为了弄清楚 *maxStatements* 的大小，你不应该简单地认为它就是池中 statement 的数量，你应该将你的应用程序中的最常用的 *PreparedStatement* 的数量乘以合理的连接数（比如在一个高负荷的应用中的 *maxPoolSize* 值）。

maxStatementsPerConnection 不是一个标准的配置参数，这可能会使你感觉有些不自然。它定义了连接池中每个连接最多能拥有（缓存）多少 Statement。为了避免过多的折腾，你可以把这个值设为稍大于你应用中的 *PreparedStatements* 数量的一个数字。

这两个值中的任何一个大于 0 的话，statement 池就会被开启。如果两个参数都大于 0，它们的限制都会被强制执行。如果只有一个参数大于 0，仅仅只有一个限制会被强制执行。

配置数据库故障的修复

c3p0 被设计成（并且默认开启了）可以从临时的数据库故障中恢复，比如数据库重启或者短暂地断开网络。你可以通过以下几个属性改变对 c3p0 在获取连接时遇到的错误的处理方式：

- *acquireRetryAttempts*
- *acquireRetryDelay*
- *breakAfterAcquireFailure*

当 c3p0 在尝试获取数据库连接失败时，会自动地重试 *acquireRetryAttempts* 次，每次间隔 *acquireRetryDelay*。如果依然失败，所有在等待这些连接的客户端将会收到一个异常，用来表示不能获取连接。请注意，如果不是所有的获取连接尝试都失败，客户端并不会收到异常，这可能在初始化尝试获取连接之后还需要一点儿时间。如果 *acquireRetryAttempts* 被设置为 0，c3p0 将会无限期地尝试获取一个新的连接，对 *getConnection()*的调用可能会无限阻塞下去，直到成功获取一个连接。

一旦所有的获取连接的尝试都失败，有两种可能的处理方式。默认情况下，c3p0 数据源会保持活性，然后对后续的请求作出回应。如果你将 *breakAfterAcquireFailure* 设置为 true 的话，数据源将会在所有尝试失败后马上停止工作，并且后续的请求也会马上失败。

请注意，如果数据库重启了，一个连接池也许还维持着那些以前获取的而现在变得不可用的连接。默认情况下，这些陈旧的连接不会被马上发现并且清理掉，当一个应用程序使用它们的时候，会马上得到一个异常。设定 *maxIdleTime* 或者 *maxConnectionAge* 可以帮助你加速替换掉这些不可用的连接。（见连接寿命的管理）如果你想完全避免应用程序因此遇到异常，你必须得指定一中连接测试策略用来在客户端使用不可用连接之前就清理掉它们。（见连接测试的配置）即使使用积极的连接测试（*testConnectionsOnCheckout* 设置为 true, 或者 *testConnectionsOnCheckin* 为 true 并且设置了一个小的 *idleConnectionTestPeriod* 值），你的应用程序在数据库重启的时候依然有可能遇到一个相关的异常，比如数据库在你已经将连接测试成功后才重启。

使用链接定制器（Connection Customizer）管理连接生命周期

应用程序在获取连接后经常会希望能够以某些标准的方法来重复地使用这些连接。例如，通过特定厂商的 APIs 或不标准的 SQL 扩展来设定字符编码或者日期时间等行为。有时候重写标准的连接中的默认属性值是很有用的，比如重写 *transactionIsolation*, *holdbility* 或者 *readOnly*。c3p0 提供了一个“钩子”接口，你实现它就有机会在刚刚从数据库获得连接之后，在连接被送至客户端之前，在连接返回连接池之前，在连接最终被连接池淘汰之前，修改或追踪这个连接。交给 *ConnectionCustomizer* 的连接是原生的，它所有的特定厂商的 API 都可以被访问。*ConnectionCustomizer* 类的更多信息见 API 文档。

安装 *ConnectionCustomizer* 的方法就是实现这个接口，让其对 c3p0 的类加载器课件，然后设置下面这个配置属性:

- *connectionCustomizerClassName*

ConnectionCustomizer 必须是不可变类，并且拥有一个没有参数的公开的构造方法。它不应该保存任何状态。(很少)有一些应用程序希望使用 *ConnectionCustomizer* 来追踪单个的数据源的行为，这些与生命周期有关的方法都能接受一个特定的数据源的“实体令牌（identityToken）”作为参数，每个 *PooledDataSource* 的实体令牌都是唯一的。

下面是一个简单的 *ConnectionCustomizer*。实现类没有必要重写 *ConnectionCustomizer* 接口中的所有的四个方法，只需要继承 *AbstractConnectionCustomizer* 类就行了。

```
import com.mchange.v2.c3p0.*;
import java.sql.Connection;

public class VerboseConnectionCustomizer
{
    public void onAcquire( Connection c, String pdsIdt )
    {
        System.err.println("Acquired " + c + " [" + pdsIdt + "]");

        // override the default transaction isolation of
        // newly acquired Connections
        c.setTransactionIsolation( Connection.REPEATABLE_READ );
    }

    public void onDestroy( Connection c, String pdsIdt )
    { System.err.println("Destroying " + c + " [" + pdsIdt + "]"); }

    public void onCheckOut( Connection c, String pdsIdt )
    { System.err.println("Checked out " + c + " [" + pdsIdt + "]"); }

    public void onCheckIn( Connection c, String pdsIdt )
    { System.err.println("Checking in " + c + " [" + pdsIdt + "]"); }
}
```

配置未结束事务（Unresolved Transaction）的处理方式

连接池中被检查过的连接不能有任何未结束的事务与其相关联。如果用户把一个连接的 *autoCommit* 属性设置成 false，并且 c3p0 不能保证在这个连接上没有后续的事务工作的话，

c3p0 就会在 check-in（当用户调用 close()方法）的时候调用 *rollback()*方法或者 *commit()*方法。JDBC 在有未结束事务的连接关闭时是应该回滚事务还是应该提交事务这个问题上保持了沉默（这简直不可原谅）。在默认情况下，c3p0 会当用户在有未结束事务的连接上调用 *close()* 方法时回滚事务。

你可以通过下面的配置参数更改这些相关行为：

- *autoCommitOnClose*
- *foreIgnoreUnresolvedTransactions*

如果你想要 c3p0 在连接返回连接池时（checkin）提交未结束的事务，只需要把 *autoCommitOnClose* 设置为 true。如果你希望自己管理这些未结束的事务的话（并且没有设置连接的 *autoCommit* 属性），你可以把 *forceIgnoreUnresolvedTransactions* 设置为 true。我们强烈地不鼓励设置 *forceIgnoreUnresolvedTransactions* 的值，因为如果客户端在连接关闭前即没有回滚也没有提交事务，并且也没有开启自动提交的话，一些奇怪的不可再生的行为（unreproduceable behavior）和数据库被锁住的现象会发生。

调试和解决有问题的客户端应用（Broken Client Applications）的配置

有时候客户端应用程序对关闭连接这种工作很马虎。于是最后池内连接数增长到了 *maxPoolSize*，然后就用尽了所有连接，这种结果是这些有问题得客户端造成的。

解决这个问题的正确的方法就是修复这些客户端程序。c3p0 可以帮助你找到那些偶尔不能正确地返回连接池的连接。在一些极少见并且不幸的情况下，即使你关闭了有漏洞的应用程序，你也修复不了它。c3p0 能够帮你解决这些问题。

下面的几个参数可以帮助你调试和解决有问题得客户端程序。

- *debugUnreturnedConnectionsStackTraces*
- *unreturnedConnectionTimout*

unreturnedConnectionTimeout 决定了从池中取出的连接能维持多少秒。如果这个值非零，那么从池中取出的那些在超过这个限制时间的连接还没有返回池得话，就会被立刻销毁，然后在池中被替代。很显然，你必须要保证这个参数的值能够让连接有时间完成那些应该做的工作。你可以用这个参数来解决那些关闭连接失败的不可靠的客户端程序。

完全修复漏洞要比仅仅使应用恢复正常工作要好得多。除了设置了 *unreturnedConnectionTimeout* 之外，如果你还把 *debugUnreturnedConnectionStackTraces* 设置为 true 的话，你将可以得到那些从池中取出的连接的轨迹栈（stack trace）。当有连接没有按时返回池的时候，轨迹栈将会被打印出来，来揭示哪里有连接没有按时返回。

数据源的其他配置

在附录 A 中查看以下几个配置参数的更多信息：

- *checkoutTimeout*
- *factoryClassLocation*
- *maxAdminisrativeTaskTime*
- *numHelperThreads*
- *usesTraditionalReflectiveProxies*

numHelperThreads 和 *maxAdministrativeTaskTime* 帮助你配置数据源线程池的行为。默认情况下，每个数据源仅有三个助手线程（helper threads）。如果性能看起来被高负荷工作拖慢，或者你通过 JMX 观察到或直接检测出了“附加任务（pending tasks）”数量超过了 0 的话，把 *numHelperThreads* 的值提高试试吧。*maxAdministrativeTaskTime* 可能对那些面临无限挂起的任务或者出现明显的死锁信息的用户有帮助。（更多信息见附录 A）

如果所有的连接都从池中取了出去，客户端不能立即得到连接的话，*checkoutTimeout* 限制了客户端会为得到一个连接等待多久。*usesTraditionalReflectiveProxies* 这个参数很少用到，它将允许你使用一种陈旧的，现在已经被取代的由 C3P0 生成的代理对象。（C3P0 以前使用反射和动态代理，而现在为了提升性能，使用了字节码生成，非反射的实现。）如果客户端没有在本地安装 c3p0，并且 c3p0 的数据源是以一种引用的形式从 JNDI 里得到的，*factoryClassLocation* 就能用来甄别 c3p0 的哪些类是可以被下载下来的。

通过 JMX 配置和管理 c3p0

如果在你的环境中有 JMX 类库和 JMX MbeanServer（它们在 JDK 1.5 以上版本已经被包括了进去），你可以通过 JMX 管理工具（比如 JDK 1.5 内置的 jconsole）来检测和配置 c3p0。你会发现 c3p0 在 *com.mchange.v2.c3p0* 下注册了很多 MBean，有一个 MBean 是整个库的汇总（叫做 C3P0Registry），每个你部署的 *PooledDataSource* 也对应着一个 MBean。你可以通过这个 *PooledDataSource* 的 MBean 来查看或者修改你的配置信息，追踪连接、Statement、线程池和其他的池与数据源的活动。（你可能需要查看 *PoolDataSource* 的 API 文档来获取它的可用的操作。）

如果你不需要 c3p0 在你的 JMX 环境下注册 MBean，你可以在系统属性或者 *c3p0.properties* 配置文件中这样来配置：

```
com.mchange.v2.c3p0.management.ManagementCoordinator=com.mchange.v2.c3p0.management.NullManagementCoordinator
```

日志配置

c3p0 使用了一种与 jakarta commons-logging 很相似的日志类库。日志信息可以传给流行的日志类库 log4j，JDK 1.4 中推荐的标准日志设备或者 System.err。差不多所有的配置都可以在你喜欢的那些高层的日志类库中完成。只有少数几个配置是 c3p0 的日志所特有的，并且使用默认配置就行了。和日志相关的配置参数可能在你的 c3p0.properties 文件里，或者在你 CLASSPATH 的顶级目录中的 mchange-log.properties 文件里，也有可能在系统属性里定义。(下面的日志配置参数可能不能在 c3p0-config.xml 中定义！)。见下面的文本框。

c3p0 的日志行为会被一些编译期选项（build-time options）影响。如果编译期选项 c3p0.debug 被设置成 false，所有低于 INFO 级别的日志信息将会被忽略。编译期选项 c3p0.trace 能够控制低于 INFO 级别的日志信息的报告颗粒细度。目前来讲，c3p0 的发行版本的二进制文件都是把 debug 设置成 true，把 trace 设置成最大值 10 来进行编译的。不过最终可能会在发行版中把 debug 设置为 false。[就目前来讲，日志级别检查（logging level-checks）对性能的影响是很小的，编译期间对这些信息的控制都相当灵活，你也能够让你的日志类库来控制哪些信息是要被记录的。]当 c3p0 启动的时候，那些 debug 和 trace 的编译期的值也会随着版本和编译时间被记录。

com.mchange.v2.log.Mlog

决定了 c3p0 的日志信息输出给哪个库。默认是 log4j，不然的话会使用 jdk1.4 的日志 API，如果它可用的话。如果 log4j 和 jdk1.4 的日志 API 都不可用，c3p0 只会将日志信息反馈给 System.err。如果你要直接控制用哪个日志类库，你就要设定以下几个属性中的一个：

- *com.mchange.v2.log.log4j.Log4jMLog*
- *com.mchange.v2.log.jdk14logging.Jdk14Mlog*
- *com.mchange.v2.log.FallbackMLog*

你也可以将这些属性设置为一个用逗号间隔的列表，来定义列表中的日志库的使用顺序。

com.mchange.v2.log.NameTransformer

默认情况下，c3p0 使用一种非常细粒度的日志策略，通常给每个 c3p0 类都分配一个日志记录器。因为各种原因，一些用户可能更喜欢更少，更加全局的日志记录器。你可以选择每个包一个日志记录器，只需要把 com.mchange.v2.log.NameTransformer 设置成 *com.mchange.v2.log.PackageNames* 就可以了。高级用户也可以定义自己实现 *com.mchange.v2.log.NameTransformer* 接口，然后把它的全限定名设置成 *com.mchange.v2.log.NameTransformer* 的值，这样来控制日志记录器的数量或名称和其他日志策略。

com.mchange.v2.log.FallbackMLog.DEFAULT_CUTOFF_LEVEL

不管是出于选择还是必要性的考虑，如果你使用 *System.err* 作为日志记录器的话，你可以用这个参数来决定日志记录的细节。下面的这些值（从 jdk1.4 中借鉴）都是可以接受的：

- *OFF*
- *SEVERE*
- *WARNING*
- *INFO*
- *CONFIG*
- *FINE*
- *FINER*
- *FINEST*
- *ALL*

这个值默认是 INFO。

性能

增强性能，是连接池、Statement 池以及 c3p0 类库的主要目的。对大多数应用程序来说，连接池会使得性能显著提高，特别是当你给每个客户端访问每次都重新获取连接的话。如果你让单个的共享的连接来为多个客户端服务以避免过多的连接获取工作的话，那么你在多线程的环境下可能会遇到性能或者事物管理问题；连接池将会能够让你有机会选择很少或没有开销的单客户端单连接模型（One Connection-per client model）。如果你正在编写企业级 Java Bean，你可能会想要只获取一次连接然后并不返回它，直到它被销毁或者过时。但是这样做太耗费资源了，因为这些 Bean 不必要的占用了连接网络和数据库资源。连接池允许 Bean 只在使用连接时才占有连接。

但是，c3p0 也有性能上的开销。为了实现当父资源返回池得时候自动清理未被关闭的 ResultSet 和 Statement，所有客户端可见的 *Connection*，*ResultSet*，*Statement* 都封装了那些底层的数据源或者“传统的”JDBC 驱动。因此，所有 JDBC 调用都会有一些额外的开销。

c3p0 在减小“封装”所带来的性能开销方面下了一些工夫。在我的环境中，由包装所带来的性能问题来自成百上千的数据库获取操作。所以，你应该从 c3p0 中得到的是性能的提升和高效的资源利用，除非你快速地轮载（succession）了很多很多 JDBC 调用。很显然，与结果集相关操作（比如要在其上遍历一个有上千条记录的表）带来的开销是可以忽略不计的。

已知缺陷

- 连接和 Statement 是基于单个认证的。所以，如果一个连接池后端的数据源从[*user*=alice, *password*=secret1]和[*user*=bob, *password*=secret2]中获取连接的话，就相当于有两个不同的池，所以在最坏的情况下，数据源可能要管理两套 *maxPoolSize* 这么多数量的连接。

  这个问题是数据源规范（它允许从多个用户认证中获取连接）造成的结果，并且同一个池中的连接在功能上必须是一致的。所以这个“问题”不会被改变或者修复。这里提出来只是为了让你弄清楚到底是怎么一回事。

- Statement 池的开销太大了。如果驱动不对 *PreparedStatement* 执行明显的预处理，Statement 池的开销要大于它节省的时间。因此他默认是关闭的。如果你的驱动的确预处理了 *PreparedStatement*，特别是通过 IPC 与 RDBMS 的话，你将可能会从中获得很明显的性能提升。（通过把 *maxStatements* 或 *maxStatementsPerConnection* 的配置属性设置成比 0 大的数）

反馈与支持

请向 swaldman@mchange.com 提供任何反馈！也可以免费加入 c3p0-users 邮件列表来提问。注册地址：http://sourceforge.net/projects/c3p0/

感谢您使用 c3p0!!!

译者注：

如果您觉得本文档有翻译不妥或错误的地方，请反馈给我。
邮箱： psjay.peng [at] gmail [dot] com
博客： http://www.psjay.com

感谢您阅读本文档！