3
Getting Database Connections in UCP

The following sections are included in this chapter:

3.1 About Borrowing Connections from UCP

An application borrows connections using a pool-enabled data source. This section describes the following concepts about borrowing connections:

> **Note:**
>
> The instructions in this section use a pool-enabled data source to implicitly create and start a connection pool.

3.1.1 Overview of Borrowing Connections from UCP

The UCP API provides two pool-enabled data sources, one for borrowing regular connections and one for borrowing XA connections. These data sources provide access to UCP JDBC connection pool functionality, and include a set of `getConnection` methods that are used to borrow connections. The same pool features are included in both XA and non-XA UCP JDBC connection pools.

UCP JDBC connection pools maintain both available connections and borrowed connections. A connection is reused from the pool if an application requests to borrow a connection that matches an available connection. A new connection is created if no available connection in the pool matches the requested connection. The number of available connections and borrowed connections are subjected to pool properties such as pool size, timeout intervals, and validation rules.

3.1.1.1 Connection Creation Using Background Threads

Starting with Oracle Database Release 23ai, new connections are created using background threads instead of user threads.

A borrow request may trigger a new connection creation, when both the following conditions are met:

- When there is no connection available in the pool at the time of the request
- If there is enough room to grow the pool

The borrow request may be satisfied by either of the following, whichever event happens first:

- A brand new connection created by the background thread
- A connection that was just returned to the pool

If the connection borrow request cannot be satisfied within the `ConnectionWaitTimeout` (CWT) period, then a `UniveralConnectionPoolException` is thrown, with the UCP-29 error code.

This behavior is different from Oracle Database Release 19c or 21c in the following ways:

- If the CWT is equal to zero or a very small value, then a borrow request has a higher chance to throw an exception because there is not enough time to create a new JDBC connection. A borrow request with a zero CWT period can return a connection only if there is one immediately available in the pool.
- A UCP exception thrown by the connection request does not always include the JDBC exception as a cause. To troubleshoot such situations, where the driver cannot connect to the Database, you can implement the `ConnectionCreationInformation` callback.
- Unlike the previous releases, the CWT is not adjusted with the value of the `CONNECT_TIMEOUT` parameter.

The current default behavior is to use background threads for creating connections, instead of the user threads, which results in enhanced efficiency. If required, you can switch back to the old behavior in the following ways:

- Setting the new system property `oracle.ucp.createConnectionInBorrowThread` to `true`
- Using the `setCreateConnectionInBorrowThread(boolean)` method to set the `createConnectionInBorrowThread` flag to `true`

3.1.2 Using the Pool-Enabled Data Source

UCP provides a pool-enabled data source (`oracle.ucp.jdbc.PoolDataSource`) that is used to get connections to a database. The `oracle.ucp.jdbc.PoolDataSourceFactory` factory class provides a `getPoolDataSource()` method that creates the pool-enabled data source instance. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
```

The pool-enabled data source requires a connection factory class in order to get an actual physical connection. The connection factory is typically provided as part of a JDBC driver, and it can also be a data source itself. A UCP JDBC connection pool can use any JDBC driver to create physical connections that are then maintained by the pool. The `setConnectionFactoryClassName(String)` method is used to define the connection factory for the pool-enabled data source instance. The following example uses the `oracle.jdbc.pool.OracleDataSource` connection factory class included with the Oracle JDBC

driver. If you use a JDBC driver provided by a different vendor, then refer to the corresponding vendor documentation for an appropriate connection factory class.

```
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
```

In addition to the connection factory class, a pool-enabled data source requires the URL, user name, and password that are used to connect to a database. A pool-enabled data source instance includes methods to set each of these properties. The following example uses an Oracle JDBC Thin driver URL syntax. If you use a JDBC driver provided by a different vendor, then refer to the corresponding vendor documentation for the appropriate URL syntax.

```
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("user");
pds.setPassword("password");
```

> **See Also:**
>
> *Oracle Database JDBC Developer's Guide* for detailed Oracle URL syntax usage.

A pool-enabled data source provides the following `getConnection` methods:

- `getConnection()`: Returns a connection that is associated with the user name and the password that were used to connect to the database.
- `getConnection(String username, String password)`: Returns a connection that is associated with the specified user name and password.
- `getConnection(java.util.Properties labels)`: Returns a connection that matches a specified label.
- `getConnection(String username, String password, java.util.Properties labels)`: Returns a connection that is associated with a specified user name and password, and that matches a specified label.

An application uses the `getConnection` methods to borrow a connection handle from the pool that is of the type `java.sql.Connection`. If a connection handle is already in the pool that matches the requested connection (same URL, user name, and password), then it is returned to the application. Otherwise, a new connection is created and a new connection handle is returned to the application. The following examples demonstrate how to borrow a connection for Oracle Database and MySQL Database respectively:

Oracle Example

The following example demonstrates borrowing a connection using the JDBC Thin driver:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("<user>");
pds.setPassword("<password>");

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

MySQL Example

The following example demonstrates borrowing a connection using the Connector/J JDBC driver from MySQL:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("com.mysql.cj.jdbc.MysqlDataSource");
pds.setURL("jdbc:mysql://host:3306/dbname");
pds.setUser("<user>");
pds.setPassword("<password>");

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

3.1.3 Using the Pool-Enabled XA Data Source

UCP provides a pool-enabled XA data source (`oracle.ucp.jdbc.PoolXADataSource`) that is used to get XA connections that can be enlisted in a distributed transaction. UCP JDBC XA pools have the same features as non-XA UCP JDBC pools. The `oracle.ucp.jdbc.PoolDataSourceFactory` factory class provides a `getPoolXADataSource()` method that creates the pool-enabled XA data source instance. For example:

```
PoolXADataSource  pds = PoolDataSourceFactory.getPoolXADataSource();
```

A pool-enabled XA data source instance, like a non-XA data source instance, requires the connection factory, URL, user name, and password in order to get an actual physical connection. These properties are set in the same way as a non-XA data source instance (see above). However, an XA-specific connection factory class is required to get XA connections. The XA connection factory is typically provided as part of a JDBC driver and can be a data source itself. The following example uses Oracle's `oracle.jdbc.xa.client.OracleXADataSource` XA connection factory class included with the JDBC driver. If a different vendor's JDBC driver is used, refer to the vendor's documentation for an appropriate XA connection factory class.

```
pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("user");
pds.setPassword("password");
```

Lastly, a pool-enabled XA data source provides a set of `getXAConnection` methods that are used to borrow a connection handle from the pool that is of the type `javax.sql.XAConnection`. The `getXAConnection` methods are the same as the `getConnection` methods previously described. The following example demonstrates borrowing an XA connection.

```
PoolXADataSource  pds = PoolDataSourceFactory.getPoolXADataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("<user>");
pds.setPassword("<password>");

XAConnection conn = pds.getXAConnection();
```

Related Topics

- Labeling Connections in UCP

3.1.4 Setting Connection Properties

Oracle's connection factories support properties that configure connections with specific features. UCP pool-enabled data sources provide the `setConnectionProperties(Properties)` method, which is used to set properties on a given connection factory. The following example demonstrates setting connection properties for Oracle's JDBC driver. If you are using a JDBC

driver from a different vendor, then refer to the vendor-specific documentation to check whether setting properties in this manner is supported and what properties are available:

```
Properties connProps = new Properties();
connProps.put("fixedString", false);
connProps.put("remarksReporting", false);
connProps.put("restrictGetTables", false);
connProps.put("includeSynonyms", false);
connProps.put("defaultNChar", false);
connProps.put("AccumulateBatchResult", false);

pds.setConnectionProperties(connProps);
```

The UCP JDBC connection pool does not remove connections that are already created if `setConnectionProperties` is called after the pool is created and in use.

> **See Also:**
>
> *Oracle Database JDBC Java API Reference* for a detailed list of supported properties to configure the connection. For example, to set the auto-commit mode, you can use the `OracleConnection.CONNECTION_PROPERTY_AUTOCOMMIT` property.

3.1.5 Using JNDI to Borrow a Connection

A connection can be borrowed from a connection pool by performing a JNDI look up for a pool-enabled data source and then calling `getConnection()` on the returned object. The pool-enabled data source must first be bound to a JNDI context and a logical name. This assumes that an application includes a Service Provider Interface (SPI) implementation for a naming and directory service where object references can be registered and located.

The following example uses Sun's file system JNDI service provider, which can be downloaded from the JNDI software download page:

http://www.oracle.com/technetwork/java/index.html

The example demonstrates creating an initial context and then performing a lookup for a pool-enabled data source that is bound to the name `MyPooledDataSource`. The object returned is then used to borrow a connection from the connection pool.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:/tmp");

ctx = new InitialContext(env);

PoolDataSource jpds = (PoolDataSource)ctx.lookup(MyPooledDataSource);
Connection conn = jpds.getConnection();
```

In the example, `MyPoolDataSource` must be bound to the context. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("<user>");
```

```
pds.setPassword("<password>");

ctx.bind(MyPooledDataSource, pds);
```

3.1.6 About Connection Initialization Callback

The Connection Initialization Callback enables applications and frameworks to initialize connections retrieved from Universal Connection Pool. It is executed at every connection checkout from the pool, as well as at each successful reconnection during failover.

This section discusses initialization callbacks in the following sections:

- Overview of Connection Initialization Callback
- Creating an Initialization Callback
- Registering an Initialization Callback
- Removing or Unregistering an Initialization Callback

3.1.6.1 Overview of Connection Initialization Callback

If an application cannot use connection labeling because it cannot be changed, then the connection initialization callback is provided for such an application.

When registered, the initialization callback is executed every time a connection is borrowed from the pool and at each successful reconnection following a recoverable error. Using the same callback at both run time and replay ensures that exactly the same initialization, which was used when the original session was established, is reestablished at run time. If the callback invocation fails, then replay is disabled on that connection.

3.1.6.2 Creating an Initialization Callback

To create a UCP connection initialization callback, an application implements the `oracle.ucp.jdbc.ConnectionInitializationCallback` interface. This interface has the following method:

```
void initialize(java.sql.Connection connection) throws SQLException;
```

> **Note:**
>
> - One callback is created for every connection pool.
> - This callback is not used if a labeling callback is registered for the connection pool.

Example

The following example demonstrates how to create a simple initialization callback:

```
import oracle.ucp.jdbc.ConnectionInitializationCallback;
class MyConnectionInitializationCallback implements ConnectionInitializationCallback
{
  public MyConnectionInitializationCallback()
  {
    ...
  }
```

```
  public void initialize(java.sql.Connection connection) throws SQLException
  {
     // Reset the state for the connection, if necessary (like ALTER SESSION)
  }
}
```

3.1.6.3 Registering an Initialization Callback

UCP provides the `registerConnectionInitializationCallback` method in the `oracle.ucp.jdbc.PoolDataSource` interface for registering a connection initialization callback.

```
public void registerConnectionInitializationCallback (ConnectionInitializationCallback
cbk) throws SQLException;
```

One callback may be registered on each connection pool instance.

3.1.6.4 Removing or Unregistering an Initialization Callback

UCP provides the `unregisterConnectionInitializationCallback` method in the `oracle.ucp.jdbc.PoolDataSource` interface for unregistering a connection initialization callback.

```
public void unregisterConnectionInitializationCallback() throws SQLException;
```

> **See Also:**
>
> *Oracle Universal Connection Pool Java API Reference* for more information

3.2 Setting Connection Pool Properties for UCP

UCP JDBC connection pools are configured using connection pool properties. The properties have `get` and `set` methods that are available through a pool-enabled data source instance. The methods are a convenient way to programmatically configure a pool. If no pool properties are set, then a connection pool uses default property values.

The following example demonstrates configuring connection pool properties. The example sets the connection pool name and the maximum/minimum number of connections allowed in the pool.

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);pds.setMaxPoolSize(20);
```

UCP JDBC connection pool properties may be set in any order and can be dynamically changed at run time. For example, `setMaxPoolSize` could be changed at any time and the pool recognizes the new value and adapts accordingly.

Related Topics

- Optimizing Universal Connection Pool Behavior

3.3 Overview of Validating Connections in UCP

Connections can be validated using pool properties when the connection is borrowed, and also programmatically using the `ValidConnection` interface. Both approaches are detailed in this section. Invalid connections can affect application performance and availability.

3.3.1 Validating When Borrowing

A connection can be validated by executing a SQL statement on a connection when the connection is borrowed from the connection pool. Two connection pool properties are used in conjunction in order to enable connection validation:

- `setValidateConnectionOnBorrow(boolean)`: Specifies whether or not connections are validated when the connection is borrowed from the connection pool. The method enables validation for every connection that is borrowed from the pool. A value of `false` means no validation is performed. The default value is `false`.
- `setSQLForValidateConnection(String)`: Specifies the SQL statement that is executed on a connection when it is borrowed from the pool.

> **Note:**
>
> The `setSQLForValidateConnection` property is not recommended when using an Oracle JDBC driver. UCP performs an internal ping when using an Oracle JDBC driver. The mechanism is faster than executing an SQL statement and is overridden if this property is set. Instead, set the `setValidateConnectionOnBorrow` property to `true` and do not include the `setSQLForValidateConnection` property.

The following example demonstrates validating a connection when borrowing the connection from the pool. The example uses Connector/J JDBC driver from MySQL:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("com.mysql.jdbc.jdbc2.optional.
   MysqlDataSource");
pds.setURL("jdbc:mysql://host:3306/mysql");
pds.setUser("<user>");
pds.setPassword("<password>");

pds.setValidateConnectionOnBorrow(true);
pds.setSQLForValidateConnection("select * from mysql.user");

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

> **See Also:**
>
> Minimizing Connection Request Delay

3.3.2 Minimizing Connection Validation with setSecondsToTrustIdleConnection() Method

In UCP, when you set the value of the `setValidateConnectionOnBorrow(boolean)` method to `true`, then each connection is validated during the checkout. This validation may incur significant overhead in applications that checkout database connections frequently.

To minimize the impact of frequent connection validation, you can now set the `setSecondsToTrustIdleConnection(int)` method with an appropriate value to trust recently-used or recently-tested database connections. Setting this value skips the connection validation test and improves application performance significantly.

The following table describes the methods available in Oracle Database Release 23ai for using this feature:

Method	Description
`setSecondsToTrustIdleConnection(int secondsToTrustIdleConnection)`	Sets the time in seconds to trust a recently-used or recently-tested database connection and skip the validation test during connection checkout.
`getSecondsToTrustIdleConnection()`	Retrieves the value that was set using the `setSecondsToTrustIdleConnection(int)` method.

When you set the `setSecondsToTrustIdleConnection(int)` method to a positive value, then the connection validation is skipped, if the connection was used within the time specified in the `secondsToTrustIdleConnection(int)` method. The default value is 0 seconds, which means that the feature is disabled.

> **Note:**
>
> The `setSecondsToTrustIdleConnection(int)` method works only if the `setValidateConnectionOnBorrow(boolean)` method is set to `true`. If you set the `setSecondsToTrustIdleConnection(int)` method to a non-zero value, without setting the `setValidateConnectionOnBorrow(boolean)` method to `true`, then UCP throws the following exception:
>
> ```
> Invalid seconds to trust idle connection value or usage.
> ```

3.3.3 Checking If a Connection Is Valid

The `oracle.ucp.jdbc.ValidConnection` interface provides two methods: `isValid` and `setInvalid`. The `isValid` method returns whether or not a connection is usable and the `setInvalid` method is used to indicate that a connection should be removed from the pool instance.

The `isValid` method is used to check if a connection is still usable after an SQL exception has been thrown. This method can be used at any time to check if a borrowed connection is valid.

The method is particularly useful in combination with a retry mechanism, such as the Fast Connection Failover actions that are triggered after a down event of Oracle RAC.

Note:

- The `isValid` method checks with the pool instance and Oracle JDBC driver to determine whether a connection is still valid. The `isValid` method results in a round-trip to the database only if both the pool and the driver report that a connection is still valid. The round-trip is used to check for database failures that are not immediately discovered by the pool or the driver.
- Starting from Oracle Database Release 18c, there is a new variant of the `isValid` method that sends an empty packet to the database unlike the older version of the method that uses a ping-pong protocol and makes a full round-trip to the database.

See Also:

Oracle Database JDBC Developer's Guide

The `isValid` method is also helpful when used in conjunction with the connection timeout and connection harvesting features. These features may return a connection to the pool while a connection is still held by an application. In such cases, the `isValid` method returns `false`, allowing the application to get a new connection.

The following example demonstrates using the `isValid` method:

```
try {  conn = poolDataSouorce.getConnection  ...}catch (SQLException sqlexc)
{
   if (conn == null || !((ValidConnection) conn).isValid())

   // take the appropriate action

...
conn.close();
}
```

For XA applications, before calling the `isValid()` method, you must cast any `XAConnection` that is obtained from `PoolXADataSource` to a `ValidConnection`. If you cast a `Connection` that is obtained by calling the `XAConnection.getConnection()` method to `ValidConnecion`, then it may throw an exception.

Related Topics

- Using Oracle RAC Features

Related Topics

- Removing Connections from UCP

3.4 Returning Borrowed Connections to UCP

Borrowed connections that are no longer being used should be returned to the pool so that they can be available for the next connection request. The `close` method closes connections

and automatically returns them to the pool. The `close` method does not physically remove the connection from the pool.

Borrowed connections that are not closed will remain borrowed; subsequent requests for a connection result in a new connection being created if no connections are available. This behavior can cause many connections to be created and can affect system performance.

The following example demonstrates closing a connection and returning it to the pool:

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

//do some work with the connection.

conn.close();
conn=null;
```

3.5 Removing Connections from UCP

The `setInvalid` method of the `ValidConnection` interface indicates that a connection should be removed from the connection pool when it is closed. The method is typically used when a connection is no longer usable, such as after an exception or if the `isValid` method of the `ValidConnection` interface returns `false`. The method can also be used if an application deems the state on a connection to be bad. The following example demonstrates using the `setInvalid` method to close and remove a connection from the pool:

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

((ValidConnection) conn).setInvalid();
...

conn.close();
conn=null;
```

3.6 UCP Integration with Third-Party Products

Third-party products, such as middleware platforms or frameworks, can use UCP to provide connection pooling functionality for their applications and services. UCP integration includes the same connection pool features that are available to stand-alone applications and offers the same tight integration with the Oracle Database.

Two data source classes are available as integration points with UCP: `PoolDataSourceImpl` for non-XA connection pools and `PoolXADataSourceImpl` for XA connection pools. Both classes are located in the `oracle.ucp.jdbc` package. These classes are implementations of the `PoolDataSource` and `PoolXADataSource` interfaces, respectively, and contain default constructors.

> **See Also:**
>
> *Oracle Universal Connection Pool Java API Reference* for more information on the implementation classes.

These implementations explicitly create connection pool instances and can return connections. For example:

```
PoolXADataSource pds = new PoolXADataSourceImpl();

pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("user");
pds.setPassword("password");

XAConnection conn = pds.getXAConnection();
```

Third-party products can instantiate these data source implementation classes. In addition, the methods of these interfaces follow the JavaBean design pattern and can be used to set connection pool properties on the class using reflection. For example, a UCP data source that uses an Oracle JDBC connection factory and database might be defined as follows and loaded into a JNDI registry:

```
<data-sources>
   <data-source
      name="UCPDataSource"
      jndi-name="jdbc/UCP_DS"
      data-source-class="oracle.ucp.jdbc.PoolDataSourceImpl">
      <property name="ConnectionFactoryClassName"
                value="oracle.jdbc.pool.OracleDataSource"/>
      <property name="URL" value="jdbc:oracle:thin:@//localhost:1521:oracle"/>
      <property name="User" value"user"/>
      <property name="Password" value="password"/>
      <property name="ConnectionPoolName" value="MyPool"/>
      <property name="MinPoolSize" value="5"/>
      <property name="MaxPoolSize" value="50"/>
   </data-source>
</data-sources>
```

When using reflection, the name attribute matches (case sensitive) the name of the setter method used to set the property. An application could then use the data source as follows:

```
Connection connection = null;
try {
   InitialContext context = new InitialContext();
   DataSource ds = (DataSource) context.lookup( "jdbc/UCP_DS" );
   connection = ds.getConnection();
   ...
```