Введение в HikariCP

**1. Обзор**

В этой вводной статье мы узнаем о проекте пула соединений [HikariCP JDBC](https://brettwooldridge.github.io/HikariCP/). **Это очень легкая (примерно 130 КБ) и молниеносная структура пулов соединений JDBC**, разработанная [Brett Wooldridge](https://github.com/brettwooldridge)около 2012 года.

**2. Вступление**

Для сравнения производительности HikariCP с другими структурами пула соединений, такими как [*c3p0*](http://www.mchange.com/projects/c3p0/), [*https://commons.apache.org/proper/*](https://commons.apache.org/proper/)*, доступно несколько результатов тестирования производительности. commons-dbcp/[dbcp2]*, [*tomcat*](https://people.apache.org/~fhanik/jdbc-pool/jdbc-pool.html)и ​​http://www.vibur.org/[ *vibur*], Например, команда HikariCP опубликовала ниже тесты (оригинальные результаты доступны [here](https://github.com/brettwooldridge/HikariCP-benchmark)):

Фреймворк так быстр, потому что были применены следующие методы:

- Разработка на уровне байт-кода **-**немного экстремальная разработка на уровне байт-кода

(включая исходное кодирование на уровне сборки) Микро-оптимизации - \*\* хотя они едва поддаются измерению, они

- Объединенные оптимизации повышают общую производительность Интеллектуальное использование фреймворка Коллекций - \*\*

*- ArrayList <Statement>*был заменен пользовательским классом *FastList*, который исключает проверку диапазона и выполняет сканирование удаления от хвоста к голове

**3. Maven Dependency**

Давайте создадим пример приложения, чтобы подчеркнуть его использование. HikariCP поставляется с поддержкой всех основных версий JVM. Каждая версия требует своей зависимости; для Java 9 имеем:

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP-java9ea</artifactId>

<version>2.6.1</version>

</dependency>

И для Java 8:

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

<version>2.6.1</version>

</dependency>

Более старые версии JDK, такие как 6 и 7, также поддерживаются. Соответствующие версии можно найти по адресу [here](https://search.maven.org/classic/" \l "search%7Cgav%7C1%7Cg%3A%22com.zaxxer%22%20AND%20a%3A%22HikariCP-java7%22)и [here](https://search.maven.org/classic/" \l "search%7Cgav%7C1%7Cg%3A%22com.zaxxer%22%20AND%20a%3A%22HikariCP-java6%22).

Кроме того, мы можем проверить последние версии в <https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22com.zaxxer%22%20AND%20a%3A%22HikariCP%22>[Central Репозиторий Maven.

**4. Использование**

Давайте теперь создадим демонстрационное приложение. Обратите внимание, что нам нужно включить подходящую зависимость класса драйвера JDBC в *pom.xml*. Если зависимости не предоставлены, приложение сгенерирует исключение *ClassNotFoundException*.

**4.1. Создание *DataSource*** *(создание класса с методом для последующего использования* HikariCP-подключения*)*

Мы будем использовать *DataSource*от HikariCP для создания единственного экземпляра источника данных для нашего приложения:

**public class DataSource {**

**private static HikariConfig config = new HikariConfig();**

**private static HikariDataSource ds;**

**static {**

**config.setJdbcUrl( "jdbc\_\_url" );**

**config.setUsername( "database\_\_username" );**

**config.setPassword( "database\_\_password" );**

**config.addDataSourceProperty( "cachePrepStmts" , "true" );**

**config.addDataSourceProperty( "prepStmtCacheSize" , "250" );**

**config.addDataSourceProperty( "prepStmtCacheSqlLimit" , "2048" );**

**ds = new HikariDataSource( config );**

**}**

**private DataSource() {}**

**public static Connection getConnection() throws SQLException {**

**return ds.getConnection();**

**}**

**}**

обратите внимание: инициализация в блоке *static*.

[HikariConfig](https://github.com/openbouquet/HikariCP/blob/master/src/main/java/com/zaxxer/hikari/HikariConfig.java)- это класс конфигурации, используемый для инициализации источника данных. Он поставляется с четырьмя известными обязательными параметрами: *username*, *password*, *jdbcUrl*, *dataSourceClassName*.

Из *jdbcUrl*и *dataSourceClassName*любой из них должен использоваться одновременно. Однако при использовании этого свойства со старыми драйверами нам может потребоваться установить оба свойства.

В дополнение к этим свойствам, есть несколько других доступных свойств, которые могут не все быть предложены другими платформами объединения:

* *autoCommit*
* *время соединения вышло*
* *idleTimeout*
* *maxLifetime*
* *connectionTestQuery*
* *connectionInitSql*
* *validationTimeout*
* *maximumPoolSize*
* *poolName*
* *allowPoolSuspension*
* *readOnly*
* *transactionIsolation*
* *leakDetectionThreshold*

HikariCP выделяется из-за этих свойств базы данных. Он достаточно продвинут, чтобы самостоятельно обнаруживать утечки соединения!

Подробное описание этих свойств можно найти по адресу [here](https://github.com/brettwooldridge/HikariCP).

Мы также можем инициализировать *HikariConfig*с помощью файла свойств, размещенного в каталоге *resources*:

private static HikariConfig config = new HikariConfig("datasource.properties" );

Файл свойств должен выглядеть примерно так:

dataSourceClassName=//TBD

dataSource.user=//TBD//other properties name should start with dataSource as shown above

Мы также можем использовать конфигурацию на основе *\_java.util.Properties - \_*:

Properties props = new Properties();

props.setProperty( "dataSourceClassName" ,//TBD );

props.setProperty( "dataSource.user" ,//TBD );//setter for other required properties

private static HikariConfig config = new HikariConfig( props );

В качестве альтернативы, мы можем напрямую инициализировать источник данных:

ds.setJdbcUrl(//TBD );

ds.setUsername(//TBD );

ds.setPassword(//TBD );

**4.2. Использование источника данных** (создание таблиц)

Теперь, когда мы определили источник данных, мы можем использовать его для получения соединения из настроенного пула соединений и выполнения действий, связанных с JDBC.

Предположим, у нас есть две таблицы с именами ***dept***и ***emp***для имитации случая использования сотрудника отдела. Мы напишем класс для извлечения этих данных из базы данных, используя HikariCP.

Ниже приведен список операторов SQL, необходимых для создания таблиц:

//создание БД

create table dept(

deptno numeric,

dname varchar(14),

loc varchar(13),

constraint pk\_\_dept primary key ( deptno )

);

create table emp(

empno numeric,

ename varchar(10),

job varchar(9),

mgr numeric,

hiredate date,

sal numeric,

comm numeric,

deptno numeric,

constraint pk\_\_emp primary key ( empno ),

constraint fk\_\_deptno foreign key ( deptno ) references dept ( deptno )

);

//заполнение БД

insert into dept values( 10, 'ACCOUNTING', 'NEW YORK' );

insert into dept values( 20, 'RESEARCH', 'DALLAS' );

insert into dept values( 30, 'SALES', 'CHICAGO' );

insert into dept values( 40, 'OPERATIONS', 'BOSTON' );

insert into emp values( 7839, 'KING', 'PRESIDENT', null, to\_\_date( '17-11-1981' , 'dd-mm-yyyy' ), 7698, null, 10);

insert into emp values( 7698, 'BLAKE', 'MANAGER', 7839, to\_\_date( '1-5-1981' , 'dd-mm-yyyy' ),

7782, null, 20);

insert into emp values( 7782, 'CLARK', 'MANAGER', 7839, to\_\_date( '9-6-1981' , 'dd-mm-yyyy' ),

7566, null, 30);

insert into emp values( 7566, 'JONES', 'MANAGER', 7839, to\_\_date( '2-4-1981' , 'dd-mm-yyyy' ),

7839, null, 40);

Обратите внимание, что если мы используем какую-либо базу данных в памяти, такую ​​как H2, нам нужно автоматически загрузить сценарий базы данных перед запуском фактического кода для извлечения данных. К счастью, H2 поставляется с параметром *INIT*, который может загружать скрипт базы данных из classpath во время выполнения. URL JDBC должен выглядеть следующим образом:

jdbc:h2:mem:test;DB\_\_CLOSE\_\_DELAY=-1;INIT=runscript from 'classpath:/db.sql'

**4.3 Использование созданного HikariCP подключения**

Нам нужно создать метод для извлечения этих данных из базы данных:

**public static List<Employee> fetchData() throws SQLException {**

**String SQL\_\_QUERY = "select \*\* from emp";**

**List<Employee> employees = null;**

**try (Connection con = DataSource.getConnection(); //где DataSource.getConnection() это**

**//класс и метод, использующие HikariCP-**

**//подключение, созданное в п.4.1**

**PreparedStatement pst = con.prepareStatement( SQL\_\_QUERY );**

**ResultSet rs = pst.executeQuery();)**

**{**

**employees = new ArrayList<>();**

**Employee employee;**

**while ( rs.next() ) {**

**employee = new Employee();**

**employee.setEmpNo( rs.getInt( "empno" ) );**

**employee.setEname( rs.getString( "ename" ) );**

**employee.setJob( rs.getString( "job" ) );**

**employee.setMgr( rs.getInt( "mgr" ) );**

**employee.setHiredate( rs.getDate( "hiredate" ) );**

**employee.setSal( rs.getInt( "sal" ) );**

**employee.setComm( rs.getInt( "comm" ) );**

**employee.setDeptno( rs.getInt( "deptno" ) );**

**employees.add( employee );**

**}**

**}**

**return employees;**

**}**

Теперь нам нужно создать метод JUnit для его тестирования. Так как мы знаем количество строк в таблице *emp*, мы можем ожидать, что размер возвращаемого списка должен быть равен количеству строк:

**@Test**

**public void givenConnection\_\_thenFetchDbData() throws SQLException {**

**HikariCPDemo.fetchData();**

**assertEquals( 4, employees.size() );**

**}**