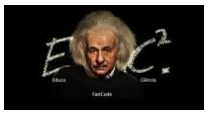


Conexão com Banco de Dados Relacional e Não Relacional em Java: Uma Análise Técnica nas Versões LTS 8, 11 e 17

A conectividade com bancos de dados é uma parte crítica no desenvolvimento de aplicações empresariais, especialmente em linguagens como Java, que oferecem suporte tanto para bancos de dados relacionais quanto não relacionais.

Neste contexto, as versões de suporte de longo prazo (LTS) de Java — 8, 11 e 17 — apresentaram diversas melhorias no gerenciamento de conexões, segurança e eficiência no acesso a dados.

A seguir, será realizada uma análise técnica das mudanças ocorridas em cada versão e seu impacto nas conexões com bancos de dados, com exemplos de códigos demonstrando boas práticas.

Java 8: Consolidação do JDBC e Integração com NoSQL via Bibliotecas Externas

Lançado em março de 2014, o **Java 8** foi um marco na evolução da linguagem, com a introdução de novas funcionalidades como **expressões lambda** e a API de **Streams**.

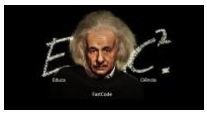
Entretanto, no que diz respeito à conexão com bancos de dados, a versão manteve o foco no modelo tradicional baseado em **JDBC (Java Database Connectivity)**, que já era amplamente utilizado para conectar a sistemas relacionais, como **MySQL**, **PostgreSQL** e **Oracle**.

Conexão com Banco Relacional Usando JDBC no Java 8:

O modelo JDBC no Java 8 é simples e direto. Um driver específico do banco de dados é registrado, e a conexão é estabelecida utilizando a classe DriverManager.

Embora este modelo seja amplamente utilizado, ele requer um controle manual de recursos, como o fechamento de conexões e o tratamento de exceções.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JdbcExample {
    public static void main(String[] args) {
```



```
        try {
            Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/meuBanco", "usuario", "senha");
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM tabela");

            while (rs.next()) {
                System.out.println("Coluna 1: " + rs.getString(1));
            }

            rs.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Integração com Bancos Não Relacionais (NoSQL)

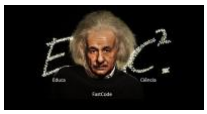
O Java 8 não inclui suporte nativo para bancos NoSQL, portanto, a integração com sistemas como **MongoDB** ou **Cassandra** depende de bibliotecas externas.

Um exemplo é o mongo-java-driver, que oferece uma interface programática para acessar dados no MongoDB.

```
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;

public class MongoDbExample {
    public static void main(String[] args) {
        MongoClient mongoClient = new MongoClient("localhost", 27017);
        MongoDatabase database = mongoClient.getDatabase("meuBanco");
        MongoCollection<Document> collection = database.getCollection("minhaColecao");

        for (Document doc : collection.find()) {
            System.out.println(doc.toJson());
        }

        mongoClient.close();
    }
}
```

Java 11: Otimizações de Memória e Conexões Assíncronas

Com o lançamento do **Java 11** em setembro de 2018, houve um foco mais forte em desempenho e simplificação de código, além da remoção de APIs desatualizadas.



A maior mudança para desenvolvedores de banco de dados foi a introdução de novas APIs para operações assíncronas e paralelas, facilitando o desenvolvimento de sistemas que requerem alta escalabilidade, principalmente em ambientes que utilizam bancos de dados distribuídos e de alta latência, como os bancos NoSQL.

Conexão JDBC no Java 11

Embora o JDBC tenha permanecido estável, a introdução de novos recursos como o **try-with-resources** simplificou o gerenciamento de conexões, tornando o código mais seguro e evitando vazamento de recursos.

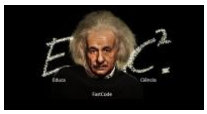
```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JdbcTryWithResources {
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost/meuBanco";
        String user = "usuario";
        String password = "senha";

        try (Connection conn = DriverManager.getConnection(url, user, password);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery("SELECT * FROM tabela")) {

            while (rs.next()) {
                System.out.println("Coluna 1: " + rs.getString(1));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Conexão Assíncrona com MongoDB no Java 11

O suporte a operações assíncronas foi aprimorado no Java 11, facilitando a integração com sistemas NoSQL em operações não bloqueantes.

No exemplo abaixo, o uso da API assíncrona do MongoDB permite que operações de leitura sejam realizadas de maneira eficiente.

```
import com.mongodb.async.client.MongoClient;
import com.mongodb.async.client.MongoClients;
import com.mongodb.async.client.MongoDatabase;
import org.bson.Document;

public class MongoDbAsyncExample {
    public static void main(String[] args) {
        MongoClient mongoClient = MongoClients.create("mongodb://localhost:27017");
        MongoDatabase database = mongoClient.getDatabase("meuBanco");
```



```
        database.getCollection("minhaColecao").find().forEach(doc -> {
            System.out.println(doc.toJson());
        }, (result, t) -> mongoClient.close());
    }
}
```

Java 17: Segurança Avançada e Melhorias no Gerenciamento de Memória

Lançado em setembro de 2021, o **Java 17** reforçou a segurança nas conexões com bancos de dados e trouxe melhorias no gerenciamento de memória, especialmente com a evolução do **Garbage Collector (G1)** e a introdução do **ZGC (Z Garbage Collector)**, tornando o uso de recursos mais eficiente em ambientes de alta concorrência.

Conexão Segura com Banco Relacional (JDBC) no Java 17

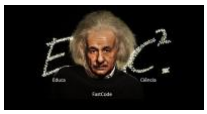
No Java 17, além das melhorias de desempenho, o foco está na segurança das conexões com bancos de dados relacionais.

O suporte nativo para conexões seguras via **SSL** foi aprimorado, garantindo que os dados sejam transmitidos de forma criptografada entre a aplicação e o banco de dados.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

public class SecureJdbcExample {
    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/meuBanco?useSSL=true", "usuario", "senha");
            Statement stmt = conn.createStatement();
            stmt.executeQuery("SELECT * FROM tabela");

            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Conexão com Bancos Não Relacionais Usando API HttpClient

O Java 17 também consolidou o uso do **HttpClient**, facilitando a integração com serviços externos, incluindo APIs de bancos de dados não relacionais baseados em HTTP, como **CouchDB** ou serviços baseados em RESTful.



```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class HttpClientExample {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create("http://localhost:5984/meuBanco/_all_docs"))
                .build();

        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        System.out.println(response.body());
    }
}
```

Comparação entre as Versões: Evolução Técnica

- **Java 8**: Consolidou o uso de JDBC para bancos relacionais e forneceu suporte básico para bancos NoSQL via bibliotecas externas. O código exige controle manual de recursos.
- **Java 11**: Melhorou o gerenciamento de recursos com o try-with-resources e introduziu a programação assíncrona, essencial para bancos de dados distribuídos e escaláveis.
- **Java 17**: Fortaleceu a segurança, melhorou o gerenciamento de memória com novos garbage collectors e trouxe uma API mais robusta para conexões seguras e integrações HTTP com bancos de dados não relacionais.

Conclusão

A evolução de Java nas versões LTS 8, 11 e 17 mostrou uma clara ênfase em melhorar a eficiência, a segurança e o desempenho no acesso a bancos de dados, tanto relacionais quanto não relacionais.

O uso de JDBC continuou a ser aprimorado, enquanto o suporte a arquiteturas modernas e distribuídas tornou-se mais fácil e seguro com o uso de APIs não bloqueantes e integrações nativas com bibliotecas NoSQL e serviços RESTful.

EducaCiência FastCode para a comunidade