Tutorial banco de dados sqlite3 prisma

TUTORIAL SQLITE3 PRISMA

Por Adalberto

Nota: Problemas no Win resolvidos, releia a parte **4,** exemplo já modificado.

Eu, pessoalmente recomendo muito o Linux, especialmente o Ubuntu. Pode acreditar, vai evitar tremendas dores de cabeça, inclusive se for correr um servidor nele.

Requisitos mínimos:

1. **instalação completa prisma-1.0 mais recente;**
2. **saber criar um arquivo fonte com prismacod ou outro editor e executá-lo com o interpretador prisma.**
3. **Caso não tenha o básico de entendimento, sugiro que visite o site www.linguagemprisma.br4.biz, baixe prisma 'tudo em um', siga os primeiros passos e leia o manual básico da linguagem prisma;**

Com este tutorial você aprenderá a:

0. Definição resumida de banco de dados;

1. ***mostrar a versão do sqlite3;***

2. ***abrir/criar banco de dados;***
3. ***criar tabelas e preencher com dados;***
4. ***ler e mostrar os dados;***
5. ***pegar o número atual de registros id de uma tabela;***
6. ***pegar o nome de todas as tabelas existentes em um bd;***
7. ***deletar e atualizar os dados de uma tabela;***

0 – Definição resumida de banco de dados;

Resumidamente, é um arquivo criado por um programa para armazenar dados de modo organizado, sendo possível, posteriormente, consultar ou modificar esses dados pelo programa;

1 – Mostrar a versão do sqlite3;

Crie um arquivo chamado sql_versao.prisma e copie o conteúdo abaixo:

```
Prisma
1  local sql = inclua'sqlite3';
2
3  imprima( sql.libversao() );
4
5  // saída--> 3.9.2
```

local sql = inclua’sqlite3′;No código fonte acima temos:

Esta linha serve para importar as funções da biblioteca sqlite3.pris para a variável ‘sql’;

imprima(sql.libversao());

Acima temos duas funções:

’imprima() – que serve para mostrar um texto ou número na tela de comandos do Linux ou Windows.

sql.libversao() – esta função retorna um texto (string) descrevendo a versão do sqlite3.

2 – abrir/criar banco de dados;

salve como: cria_bd.prisma

```
Prisma
```

```
local sql = inclua 'sqlite3';

ret , base = sql.abra( "teste.bd" );

se ret == sql.SQLITE_OK entao

imprima 'base criada com sucesso!\n';

senao

imprima( " ERRO: " , sql.mensagem_erro(base) );

fim

sql.feche(base);

leia();
```

local sql = inclua ‘sqlite3’;Detalhes:

No trecho acima incluímos as funções da biblioteca sqlite3 na tabela sql;

ret, base = sql.abra(‘teste.bd’);

Nesta linha abrimos o arquivo ‘teste.bd’ de banco de dados (único parâmetro é uma string descrevendo o nome do arquivo / caso não exista o arquivo, é criado um, caso exista, é aberto para leitura ou gravação) e são retornados dois valores: ret e base.

Entendendo os dois retornos:

***ret –** é um número que indica êxito (zero) ou erro (qualquer outro número);

***base** – é uma variável do banco de dados aberto. *(os nomes dos retornos ficam a critério do programador, aqui eu escolhi ret e base mas poderiam ser outros.)*

*(Obs. a variável tabela **sql** recebeu o retorno da função inclua’sqlite3'; é por isso que podemos usar os métodos após escrever **sql.**)*

Prisma

```
se ret == sql.SQLITE_OK entao

imprima 'base criada com sucesso!\n';

senao

imprima( " ERRO: " , sql.mensagem_erro(base) );

fim
```

Aqui pode ser lido como: se o retorno ret for igual a sql.SQLITE_OK então coloque na tela a frase de sucesso! Caso contrário (se ret não for igual a sql.SQLITE_OK) imprima a mensagem de erro.

**Nota – SQLITE_OK é uma variável reservada da biblioteca sqlite3 que tem como valor o número zero.*

A função **sql.mensagem_erro(base)** retorna uma string descrevendo o útimo erro contido em 'base' que é a variável do banco de dados aberto, lembra. Único parâmetro é a variável do banco de dados aberto.

sql.feche(base);

E por fim, não se esqueça de fechar o banco de dados aberto com a função acima.

3 – criar tabelas e preencher com dados;

Quando criamos um arquivo de dados o que pretendemos é gravar dados nele, e de forma organizada para pesquisarmos esses dados futuramente.

**Nota – Daqui em diante, ocasionalmente, poderei usar o termo bd para dizer 'banco de dados' ou o 'arquivo do banco de dados'!*

Para colocarmos dados no bd de modo organizados criamos tabelas, uma espécie de estrutura que lembra um prateleira com várias colunas e divisões horizontais(linhas), para cada coluna criada devemos colocar o tipo de dado correspondente. Para melhor visualizarmos, observe o exemplo abaixo:

tabela clientes:

índices	NOME	ENDEREÇO	TELEFONE
1	Paulo Martins	Rua das Graças	99 9999 99999
2	Edmara Silva	Rua dos Diamantes	66 6666 66666
3	Pedro Alcântara	Rua Eldorado	55 5555 55555

Veja que a tabela acima é como uma prateleira com três colunas cada uma para guardar um dado específico. A primeira são os índices da sequência, a segunda coluna somente nomes, a terceira o endereço e a quarta os números de telefones. Assim os dados ficam organizados, não podemos trocar, colocar, por exemplo, nomes na segunda ou terceira coluna ou vice-versa, pois os dados ficariam desorganizados.

É claro que num banco de dados as tabelas não serão visualmente como no exemplo acima, mas o modo de acessá-las será desta forma, em linhas e colunas. Por exemplo, poderíamos acessar 'Edmara Silva' direcionando a pesquisa para segunda coluna na linha 2; ou seu telefone em coluna – 4 e linha – 2;

**Nota – Dentro de um mesmo bd podemos criar inúmeras tabelas e dentro destas tabelas podemos colocar bem mais de um dado.*

Entendido um pouco o que são tabelas, vamos aos códigos:

Salve como tabela_insere_dados.prisma

Prisma

```
1  local sql = inclua'sqlite3'
2
3  ret , base = sql.abra('teste.bd');
4
5  se ret <> sql.SQLITE_OK entao
6
```

```
imprima( "Erro:" , sql.mensagem_erro(base) );

sql.feche(base);

sis.saia(); //deixa o programa em erro fatal.

fim

str_exec = [[

DROP TABLE IF EXISTS amigos;

CREATE TABLE amigos(Id INT, Nome TEXT, Idade INT);

INSERT INTO amigos VALUES(1, 'Marcos', 52);

INSERT INTO amigos VALUES(2, 'Maria', 19);

INSERT INTO amigos VALUES(3, 'Pricila', 9);

]];

ret , erro_msg = sql.exec(base, str_exec );

se ret <> sql.SQLITE_OK entao

imprima( "ERRO:", erro_msg);

sql.feche(base);

sis.saia();

senao

imprima("Tabela criada e dados inseridos com sucesso\n");

fim

sql.feche(base);

leia(); //para não fechar abruptamente no windows.
```

Vamos pular o que já foi explicado nos exemplos anteriores, ok!Detalhes:

```
str_exec = [[

DROP TABLE IF EXISTS amigos;

CREATE TABLE amigos(Id INT, Nome TEXT, Idade INT);

INSERT INTO amigos VALUES(1, 'Marcos', 52);

INSERT INTO amigos VALUES(2, 'Maria', 19);

INSERT INTO amigos VALUES(3, 'Pricila', 9);

]];
```

Este trecho acima é uma string de várias linhas em prisma contendo os comandos do sqlite3 (linguagem SQL); logo para usar o sqlite3 precisamos usar sua linguagem e compreendê-la. Aqui passarei o básico.

**Nota – É bom usar duplos colchetes de strings multilinhas ([[…]]) pois precisamos usar aspas internas.*

Vamos detalhar os comandos sqlite3 acima:

DROP TABLE IF EXISTS amigos;

Cuidado! Este trecho apaga totalmente a tabela amigos com seus dados se ela já existir.

CREATE TABLE amigos(Id INT, Nome TEXT, Idade INT);

O comando acima significa: **CRIE TABELA (CREATE TABLE) amigos** e entre parênteses são passados os nomes das colunas e o tipo de dados após os nomes:

Id (letra i maiuscula e d) é do tipo **INT** (inteiro – número sem casas decimais)

Nome é do tipo **TEXT** (‘string entre aspas simples’);

Idade é do tipo **INT** também, igual ao primeiro.

**Note que a linguagem sqlite3 segue uma sintaxe (regras) cada comando termina com ponto-e-vírgula.*

Observe que ao criar uma tabela, somos obrigados desde já a definir o número, tipo e nome das colunas podendo ser modificados posteriormente com funções do sqlite.

INSERT INTO amigos VALUES(1, ‘Marcos’, 52);

Aqui inserimos (INSERT) dentro (INTO) da tabela amigos os valores (VALUES): (1 , ‘Marcos’ , 52);

**note que o nome está entre apas simples pois é do tipo texto.*

**Note que entre parênteses os valores são postos na mesma ordem de criação das colunas da tabela.*

E os demais comandos fazem a mesma coisa, só trocando os valores:

INSERT INTO amigos VALUES(2, ‘Maria’, 19);

INSERT INTO amigos VALUES(3, ‘Pricila’, 9);

Após criar uma string de comandos sqlite3 podemos executá-la com a função abaixo:

ret , erro_msg = sql.exec(base, str_exec);

Esta função recebe 3 parâmetros:

1 – variável do banco de dados.

2 – a string de comandos sqlite3.

3 – foi omitido é opcional, trataremos adiante.

Com isso a string é executada, a tabela é criada (antes apagada se já existir) e os dados são inseridos.

E retorna 2 valores:

1 – **ret** é o número que descreve êxito caso seja zero (sql.SQLITE_OK) ou erro caso seja qualquer outro número.

2 – **erro_msg** é uma string que descreve o erro caso haja um.

Os outros comandos do exemplo são parecidos com o anterior, e já foi explicado!

Outro exemplo de inserção de dados no mesmo bd:

```
sql = inclua 'sqlite3'; //incluindo a biblioteca sqlite3

ret , base = sql.abra('teste.bd');

se ret <> sql.SQLITE_OK entao

imprima( "Erro:" , sql.mensagem_erro(base) );

sql.feche(base);

sis.saia();

fim

str_exec = [[

DROP TABLE IF EXISTS Carros;

CREATE TABLE Carros(Id INT, Nome TEXT, Preco INT);

INSERT INTO Carros VALUES(1, 'Audi', 52642);

INSERT INTO Carros VALUES(2, 'Mercedes', 57127);

INSERT INTO Carros VALUES(3, 'Skoda', 9000);

INSERT INTO Carros VALUES(4, 'Volvo', 29000);

INSERT INTO Carros VALUES(5, 'Bentley', 350000);

INSERT INTO Carros VALUES(6, 'Citroen', 21000);

INSERT INTO Carros VALUES(7, 'Hummer', 41400);

INSERT INTO Carros VALUES(8, 'Volkswagen', 21600);

]];

ret , erro_msg = sql.exec(base, str_exec );

imprima(erro_msg);
```

```
se ret <> sql.SQLITE_OK entao

imprima( "SQL error:", erro_msg);

sql.feche(base);

sis.saia();

senao

imprima("Tabela criada e dados inseridos com sucesso\n");

fim

sql.feche(base); //fechado a base de dados.

leia();
```

4 – ler e mostrar os dados;

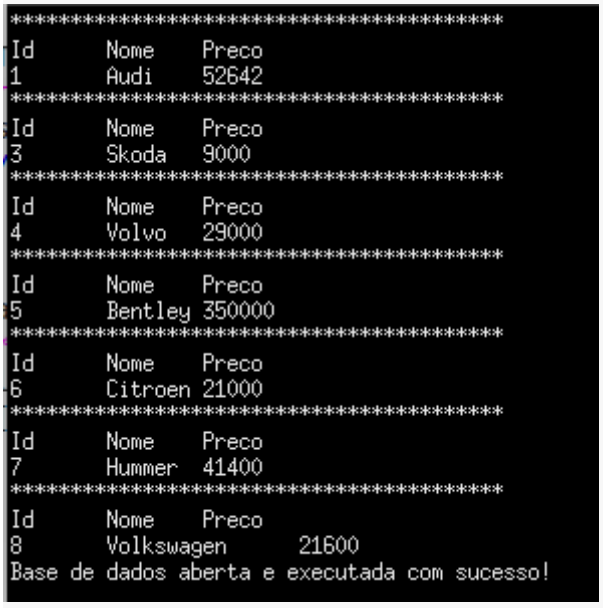
4.1 lendo uma tabela completa

Salve como ler_tabela.prisma

```
local sql = inclua'sqlite3'

ret , base = sql.abra("teste.bd");

se ret <> sql.SQLITE_OK entao //se ret for diferente de SQLiTE_OK

imprima( "Erro:" , sql.mensagem_erro(base) );

sql.feche(base);

sis.saia(); //deixa o programa em erro fatal.

fim

funcao sql_call ( valores , nome_coluna )

imprima( '****************************************');

imprima( nome_coluna[1] , nome_coluna[2] , nome_coluna[3] );

imprima( valores[1] , valores[2] , valores[3] );

fim

ret , erro_msg = sql.exec(base, "SELECT * FROM Carros" , "sql_call( %s , %s )" );

se ret <> sql.SQLITE_OK entao

imprima( "SQL error:", erro_msg );

sql.feche(base);

sis.saia();
```

```
senao

imprima("Base de dados aberta e executada com sucesso!\n");

fim

sql.feche(base); //fechado a base de dados.

leia();
```

Veja o resultado:



Detalhes do exemplo acima: (lembre-se, vamos pular o que já foi explicado nos exemplos anteriores)

```
funcao sql_call ( valores , nome_coluna )

imprima( '****************************************');

imprima( nome_coluna[1] , nome_coluna[2] , nome_coluna[3] );

imprima( valores[1] , valores[2] , valores[3] );

fim
```

Lembra do 3 parâmetro omitido no exemplo anterior? Certo, precisamos dele agora, é a função de retorno acima

para criar essa função não tem segredos, é uma função prisma normal, com o nome que preferir, os parâmetros principais são dois os valores e nomes das colunas.

O que acontece é que ao usar a função sql.exec(), esta percorrerá cada linha da tabela executando a função de retorno e passando os

valores e nomes de colunas de cada linha precorrida.

**Note que valores e nome_coluna são tabelas prisma contendo os dados da tabela sqlite3.No caso, como criamos três colunas na tabela Carros usamos 3 indices para acessar os dados de valores e nome_coluna.*

ret , erro_msg = sql.exec(base, “SELECT * FROM Carros” , “sql_call(%s , %s)”);

Esta é a função que executa a string sqlite3;

Algumas novidades aqui:

1 – passamos diretamente a string de comando sqlite3 dentro do parênteses. (tanto faz, por variável ou diretamente)

2 – usamos o terceiro parâmetro, omitido nos exemplos anteriores.

Vamos entender aqui o comando sqlite3.

“SELECT * FROM Carros” – pode ser lido como: selecione tudo (*) o que estiver (from) na tabela Carros.

“sql_call(%s , %s)” – para passar a função de retorno (call back) como parâmetro usamos uma string como se fôssemos utilizar a função executestring(); pois internamente, a biblioteca igsqlite3 usa essa função.

E os dois ‘%s’ dentro do parênteses? Eles são necessários pois são caracteres de formatação que serão substituídos pelos parâmetros valores e nome_colunas a cada linha da tabela. Não se preocupe apenas utilize-os sempre!

Como explicado anteriormente essa função retorna dois valores, o número e a mensagem de erro.

Quando essa função acima é executada, ela executa o comando sqlite3 “SELECT * FROM Carros” e faz um loop (repetição) passando cada linha dessa tabela selecionada até chegar a última, isso acontece pois usamos o * que significa todos as linhas da tabela.

E se em vez de quer apenas imprimir os dados eu quisesse manipulá-los. Muito simples crie uma tabela prisma e passe todos os dados para ela usando a função de retorno.

Veja um exemplo abaixo:

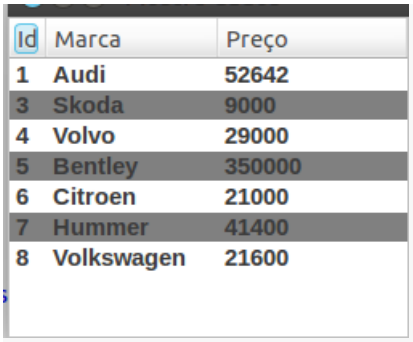
Prisma

```
/**

* Exemplo igsqlite3: obtendo e imprimindo os dados

*

* **//

local sql = inclua 'sqlite3'; //incluindo a biblioteca sqlite3

//abrindo uma base de dados chamada teste.base

ret , base = sql.abra("teste.bd");
```

```
14
15  se ret <> sql.SQLITE_OK entao //se ret for diferente de SQLiTE_OK
16
17  imprima( "Erro:" , sql.mensagem_erro(base) );
18
19  sql.feche(base);
20
21  sis.saia(); //deixa o programa em erro fatal.
22
23  fim
24
25  //funcao call back
26
27  dados = {};
28  dados[0] = nulo//para evitar erros no windows.
29
30  cont = 1;
31
32  funcao sql_call ( msg , valores , nome_coluna )
33
34  dados[cont] = {};
35
36  para i = 1 , #valores inicio
37
38  dados[cont][nome_coluna[i] ] = valores[i] ; //dados[1].coluna = valor
39
40  fim
41
42  cont = cont + 1;
43
44  fim
45
46  //a funcao exec executa a string de comando SQL = str_exec acima.
47
48  ret , erro_msg = sql.exec(base, "SELECT * FROM Carros" , "sql_call( nulo , %s , %s )" );
49
50  se ret <> sql.SQLITE_OK entao //se nao for igual a sucesso é erro.
51
52  imprima( "SQL error:", erro_msg );
53
54  sql.feche(base);
55
56  sis.saia();
57
58  senao
59
60  imprima("Base de dados aberta e executada com sucesso!\n");
61
62  fim
63
64  //imprimindo a tabela:
65
66  para i = 1 , #dados inicio
67
68  imprima('\n***********************************');
69
70  imprima( 'id: ' .. dados[i].Id .. ' | Marca: ' .. dados[i].Nome .. ' | Preço: ' .. dados[i].P
71
72  fim
73
74  sql.feche(base); //fechado a base de dados.
75
76  //criando uma janelinha com listagem mostrando os dados:
77
78  inclua'igbr'
79
80  jan = ig.janela('Mostre dados' , 250 , 200 );
81
82  ig.janela_def_posicao( jan , ig.janela_pos_centro);
```

```
83
84  ig.conecte(jan, ig.destruido , 'sis.saia(0)' ); //necessário este comando para evitar erros r
85
86  v = ig.caixavertical(falso , 0 );
87
88  ig.ad( jan , v);
89
90  listagem = ig.listagem_com_titulos ('Id', 'Marca' , 'Preço' );
91
92  ig.listagem_def_largura_coluna ( listagem , 2 , 100 );
93
94  ig.ad( v , listagem );
95
96  ig.componente_modifique_fonte ( listagem , "Arial", ig.negrito, 11); //define a fonte da list
97
98  ig.listagem_def_tipo_sombra( listagem , ig.sombra_riscado_dentro ); //define o estilo de bord
99
100 //tente ig.sombra_dentro, ig.sombra_fora, ig.sombra_riscado_fora, execute e veja o resultado.
101
102 para i = 1 , #dados inicio
103
104 ig.listagem_anexe ( listagem , dados[i].Id , dados[i].Nome , dados[i].Preco );
105
106 se i % 2 == 0 entao //se i for par
107
108 cor = ig.cor_analise( ig.cinza ); //definindo uma cor para letra
109
110 ig.listagem_def_cor_fundo( listagem , i , cor );
111
112 fim //se i % 2
113
114 fim
115
116 //call back do box de listagem
117
118 //funcao que sera conectada ao box de listagem:
119
120 funcao list_exec ( comp , lin , col , evento )
121
122 local id = ig.listagem_obt_texto(comp, lin, 1 );
123
124 local Marca = ig.listagem_obt_texto(comp, lin, 2 );
125
126 local Preco = ig.listagem_obt_texto(comp, lin, 3 );
127
128 texto2 = "id = " .. id .. '\nMarca: ' .. Marca .. '\nPreço: ' .. Preco;
129
130 ig.msg( janela , 'Listagem' , texto2 );
131
132 fim
133
134 ig.conecte_funcao( listagem , ig.clique_linha , list_exec );
135
136 //note que o s do %s é minusculo, senao causa um erro imperceptível para o debug.
137
138 ig.componente_mostre_todos(jan);
139
140 ig.fimprograma();
```

Saída:



Id	Marca	Preço
1	Audi	52642
3	Skoda	9000
4	Volvo	29000
5	Bentley	350000
6	Citroen	21000
7	Hummer	41400
8	Volkswagen	21600

Para o ambiente gráfico foi usado a biblioteca ‘igbr’. No site oficial prisma há vários exemplos na seção página de posts.

4.2 lendo uma linha específica da tabela:

Salve como ler_linha.prisma

```
local sql = inclua'sqlite3';

ret , base = sql.abra("teste.bd");

se ret <> sql.SQLITE_OK entao

imprima("Erro: ", sql.mensagem_erro(base));

sql.feche(base);

fim

str_exec = "SELECT Id, Nome , Preco FROM Carros WHERE Id = 3";

funcao sql_func(valor,coluna)

imprima('++++++++++++++++++++++');

imprima( coluna[1] , coluna[2] , coluna[3]);

imprima( valor[1] , valor[2] , valor[3] );

imprima('++++++++++++++++++++++');

fim

ret , msg = sql.exec(base , str_exec , 'sql_func(%s,%s)' );

sql.feche(base);

leia();
```

str_exec = “SELECT Id, Nome , Preco FROM Carros WHERE Id = 3”;Vamos direto para as novidades aqui, comando sqlite3 novo:

Pode ser entendido como: SELECIOME as colunas Id, Nome e Preco DA tabela Carros ONDE Id for igual a 3.

Ao ser executada essa string de comando sqlite3, ela procura pela linha onde a coluna Id for igual a 3 e retorna os valores para a função de retorno sql_func ();

```
funcao sql_func(valor,coluna)

```

```
imprima('++++++++++++++++++++++');

imprima( coluna[1] , coluna[2] , coluna[3]);

imprima( valor[1] , valor[2] , valor[3] );

imprima('++++++++++++++++++++++');

fim
```

Esta função acima é a função call back(retorno) que é executada automaticamente pela sql.exec() e recebe os valores nome da coluna da tabela sqlite3 e seus dados em valor.

ret , msg = sql.exec(base , str_exec , 'sql_func(%s,%s)');

E, por fim, executamos a string de comando sqlite3, relembre que o terceiro parâmetro é a string da função de retorno parênteses e os dois %s que será substituídos pelos valores e colunas da tabela sqlite3.

**Nota – Veja que a função sql_func tem dois parâmetros valor e coluna que são os %s passados para sql.exec. Esses dois valores são tabelas prisma e o número de elementos depende do número de colunas criadas na tabela sqlite3.*

Este exemplo acima funciona bem como consulta de dados. Caso queira pegar os dados, em vez de simplesmente imprimi-los, use uma tabela prisma para isso, assim:

Prisma

```
local dados = { [0] = nulo }; //evitar erros no windows.
local cont = 1;
funcao sql_func(valor,coluna)

dados[cont] = {}; //cada índice da tabela acima também será uma tabela

dados[cont][coluna[1] ] = valor[1]; //os campos serão os nomes das colunas, valor será os valo

dados[cont] [coluna[2] ] = valor[2];

dados[cont] [coluna[3] ] = valor[3];

cont = cont + 1; //caso a função se repita cont é incrementado para um novo índice da tabela c
fim
```

Não esqueça no final de usar o comando sis.saia(0) para evitar erro no Windows após fechar o programa ou se estiver usando modulo igbr use: ig.conecte(janela, ig.destruido , 'sis.saia(0)');

Pronto, agora podemos acessar os dados assim: dados.Id, dados.Nome , dados.Preco

Saída do exemplo acima:

4.3 Comandos parametrizados do sqlite3.

Comando sql.prepare_v2();

```
local sql = inclua'sqlite3';
ret , base = sql.abra("teste.bd");


  se ret <> sql.SQLITE_OK entao

        imprima("Erro: ", sql.mensagem_erro(base));
        sql.feche(base);
  fim //fim se ret <>

sql_cmd = "SELECT Id, Nome , Preco FROM Carros WHERE Id = ?"
    ret , res = sql.prepare_v2( base , sql_cmd );

    se ret == sql.SQLITE_OK entao
        sql.vincule_int( res , 1, 4); //vincula o valor 4 da tabela sql ao indice 1
   senao
        imprima( "Falha na execucao: " .. sql.mensagem_erro(base) );
       leia();
       sis.saia(0);
    fim

    pas = sql.passe(res);

    se (pas == sql.SQLITE_LIN) entao
        imprima ( sql.coluna_texto( res , 1) .. ' : ' .. sql.coluna_texto(res, 2) .. ' : ' ..
    fim

    sql.finalize(res);
    sql.close(base);

leia();
```

Detalhes:

“SELECT Id, Nome , Preco FROM Carros WHERE Id = ?”;

Essa é a string de comando sqlite. O ponto de interrogação será substituído por um valor posteriormente.

ret , res = sql.prepare_v2(base , sql_cmd);

Este comando compila (prepara) a string sqlite3. Parâmetros: 1- banco de dados aberto, 2- é a string sqlite3.

Dois retornos:

ret é um número que pode indicar êxito (zero) ou falha (qualquer outro número);

res é a string sqlite já compilada.

sql.vincule_int(res , 1, 4);

Esta função coloca o valor 4 no lugar do **“?”** dentro da string de comando sqlite.

Parâmetros: 1 – string sqlite compilada, 2 – índice sqlite, 3 – número que ficará no lugar do **“?”;**

pas = sql.passe(res);

A função sql.passe() analisa a string sqlite compilada e retorna uma linha se houver (pas); único parâmetro é a string compilada.

se (pas == sql.SQLITE_LIN) entao

A condição testa se pas (retorno de sql.passe()) é um alinha sqlite válida.

sql.coluna_texto(res , 1)

Esta função retorna o valor da coluna 1 da tabela Carros. (Id)

sql.coluna_texto(res , 2) retorna o valor da coluna 2; (Nome)

sql.coluna_texto(res , 3) retorna o valor da coluna 3; (Preco);

Obs. Este método, particularmente, tem um pequeno grau de complicação mas é mais rápido e seguro.

4.3 Leitura de dados sqlite3. (Não ocorre erro no Windows, comando mais confiável);

Lendo todos os dados da tabela:

Prisma

```
local sql = inclua'igsqlite3';
ret , base = sql.abra("teste.bd");


   se ret <> sql.SQLITE_OK entao

        imprima("Erro: ", sql.mensagem_erro(base));
        sql.feche(base);
  fim //fim se ret <>


    ret , res = sql.prepare_v2( base , "SELECT * FROM Carros;" );

   se ret <> sql.SQLITE_OK entao erro('Erro'); fim

 enquanto 1 inicio
    pas = sql.passe(res); //executa a string compilada res.

    se (pas == sql.SQLITE_LIN) entao
        imprima( sql.coluna_nome(res, 1)  , sql.coluna_nome(res, 2) , sql.coluna_nome(res, 3)
```

```
        imprima ( sql.coluna_texto( res , 1) .. ' : ' .. sql.coluna_texto(res, 2) .. ' : ' ..
    senao
        quebre;
    fim

 fim //enquanto

    sql.finalize(res); // **//
    sql.close(base);

leia();
```

"SELECT * FROM Carros;"

Esta é a cláusula sqlite, ela seleciona todos (*) os dados da (from) tabela Carros.

ret , res = sql.prepare_v2(base , *"SELECT * FROM Carros;"*);

Aqui compilamos a cláusula.

Prisma

```
enquanto 1 inicio
    pas = sql.passe(res); //executa a string compilada res.

    se (pas == sql.SQLITE_LIN) entao
        imprima( sql.coluna_nome(res, 1)  , sql.coluna_nome(res, 2) , sql.coluna_nome(res, 3)
        imprima ( sql.coluna_texto( res , 1) .. ' : ' .. sql.coluna_texto(res, 2) .. ' : ' ..
    senao
        quebre;
    fim

 fim //enquanto
```

**Nota – Baixe a versão mais atual de Prisma para a função sql.coluna_nome()*

Depois de compilar a string sqlite podemos usar a função **sql.passe().** A cada repetição dessa função uma linha da tabela é percorrida. O retorno é um código de erro que pode indicar uma linha válida (sql.SQLITE_LIN = 100) ou não. Caso seja uma linha válida é possível obter os nomes das colunas com a função **sql.coluna_nome()** e o valor das colunas com a função **sql.coluna_texto().**

Dica – use a função convnumero() caso queira converter de string para número.

5 – pegar o número atual de registros id de uma tabela;

Muitas vezes podemos querer saber qual foi o id da última linha inserida em uma tabela, para isto temos a função:

sql.ultimo_id(base);

Para essa função funcionar devemos usar um tipo de coluna chamado de chave de auto contagem ou auto incremento.

Nesse tipo de coluna não precisamos colocar seu valor 1, 2, 3, 4 … a contagem é feita automaticamente. Veja abaixo:

Salve como ultimo_id.prisma

```
sql = inclua 'sqlite3';

ret , base = sql.abra(sql.memoria);

se ret <> sql.SQLITE_OK entao

imprima( "Erro:" , sql.mensagem_erro(base) );

sql.feche(base);

sis.saia(); //deixa o programa em erro fatal.

fim

str_exec = [[

CREATE TABLE Amigos(Id INTEGER PRIMARY KEY, Nome TEXT, idade INT);

INSERT INTO Amigos(Nome, idade) VALUES ('Tom', 15);

INSERT INTO Amigos(Nome, idade) VALUES ('Rebecca', 34);

INSERT INTO Amigos(Nome, idade) VALUES ('Jim', 44);

INSERT INTO Amigos(Nome, idade) VALUES ('Roger', 23);

INSERT INTO Amigos(Nome, idade) VALUES ('Robert', 32);

]];

ret , erro_msg = sql.exec(base, str_exec );

se ret <> sql.SQLITE_OK entao

imprima( "SQL error:", erro_msg);

sql.feche(base);

sis.saia();

senao

imprima("Tabela criada e dados inseridos com sucesso\n");

fim

ult_id = sql.ultimo_id( base ); //esta funcao pega o ultimo id primary key inserido na tabela.

imprima( "Ultima chave id da tabela Amigos é:" , ult_id);

sql.feche(base); //fechado a base de dados.
```

ret , base = sql.abra(sql.memoria);

Novidades neste exemplo:

Ao usar o comando acima com o parâmetro sql.memoria um banco de dados será criado na memória, não haverá nenhum arquivo.

str_exec = [[

CREATE TABLE Amigos(Id INTEGER PRIMARY KEY, Nome VARCHAR(55), idade INT);

INSERT INTO Amigos(Nome, idade) VALUES ('Tom', 15);

INSERT INTO Amigos(Nome, idade) VALUES ('Rebecca', 34);

INSERT INTO Amigos(Nome, idade) VALUES ('Jim', 44);

INSERT INTO Amigos(Nome, idade) VALUES ('Roger', 23);

INSERT INTO Amigos(Nome, idade) VALUES ('Robert', 32);

]];

Acima é a string de comandos sqlite3. Podemos perceber nela a chave de auto contagem na criação da tabela:

CREATE TABLE Amigos(Id INTEGER PRIMARY KEY, Nome VARCHAR(55), idade INT);

eis a chave: **Id INTEGER PRIMARY KEY,** será sempre assim não a modifique.

A segunda coluna criada é o Nome que é do tipo VARCHAR(55) (uma string de no máximo 55 caracteres) e a terceira coluna é idade do tipo INT (número inteiro);

Veja que quando usamos uma chave (KEY) de auto contagem, devemos especificar o nome das colunas entre parênteses ao inserir os dados, deixando de fora a chave de auto contagem. Veja abaixo:

INSERT INTO Amigos(Nome, idade) VALUES ('Tom', 15);

E assim com os comandos restantes.

ult_id = sql.ultimo_id(base);

Aqui pegamos número da última chave (**INTEGER PRIMARY KEY**), o retorno é um número prisma.

O resultado será 5 pois criamos somente 5 linhas no exemplo, mas pense em um arquivo com milhares de dados, esta função é muito útil para inserir um novo dado sem correr o risco de errar a contagem.

6 – pegar o nome de todas as tabelas existentes em um bd;

Algumas vezes nós podemos querer descobrir que tabelas estão criadas em um bd, mas sem fazer ideia do nome delas. Para isso veja o exemplo abaixo:

Salve com o nome de nome_tabela.prisma

```
local sql = inclua 'igsqlite3'; //incluindo a biblioteca sqlite3

ret , base = sql.abra("Teste.bd");

se ret <> sql.SQLITE_OK entao

imprima( "Erro:" , sql.mensagem_erro(base) );

sql.feche(base);

sis.saia(); //deixa o programa em erro fatal.

fim

dados = {};
dados[0] = nulo; //atribua um valor qualquer para não dar erro no Win.
cont = 1;

funcao sql_call ( valores , nome_coluna )

dados[cont] = {[0]=nulo};

dados[cont] = { nome_coluna[1], valores[1] }

cont = cont + 1;

fim

ret , erro_msg = sql.exec(base, "SELECT name FROM sqlite_master WHERE type='table';" , "sql_cc

se ret <> sql.SQLITE_OK entao //se nao for igual a sucesso é erro.

imprima( "SQL error:", erro_msg );

sql.feche(base);

sis.saia();

senao

imprima("Base de dados aberta e executada com sucesso!\n");

fim

//imprimindo a tabela:

para i = 1 , #dados inicio

imprima('\n***********************************');

imprima( dados[i][1] , dados[i][2] );

fim

sql.feche(base); //fechado a base de dados.
```

Detalhes:

Prisma

```
funcao sql_call ( valores , nome_coluna )

dados[cont] = {[0] = nulo};

dados[cont] = { nome_coluna[1], valores[1] }

```

```
cont = cont + 1;

fim
```

Acima está a função de retorno (call back);

A novidade aqui é o comando sqlite3 que permite selecionar o nome de todas as tabelas do bd:

“SELECT name FROM sqlite_master WHERE type=’table’;”

Leia: SELECIONE name de sqlite_master ONDE o tipo for ‘tabela’;

sqlite_master é uma espécie de cabeçalho de informações do bd.

Este comando sqlite3 é passado como segundo parâmetro na função abaixo:

sql.exec(base, “SELECT name FROM sqlite_master WHERE type=’table’;” , “sql_call(%s , %s)”);

Que ao ser executado, executa também a função call back passando os dados selecionados no comando sqlite3.

7 – deletar e atualizar os dados de uma tabela;

7.1 Apagando uma linha da tabela:

Deletar uma linha é simples, veja o exemplo abaixo:

salve como deleta.prisma

Prisma

```
local sql = inclua'sqlite3'

local xstr = [[

DELETE from Carros where Id = 2;

]];

funcao exec() fim;

local ret , base = sql.abra('Teste.bd');

se ret <> sql.SQLITE_OK entao

poe('Erro ao abrir banco de dados');

sis.saia();

fim

local ret , erro_msg = sql.exec(base, xstr , "exec()" );

```

```
se ret <> sql.SQLITE_OK entao

poe('Erro: ' .. erro_msg );

sis.saia();

senao

poe'dado deletado com sucesso';

fim

leia();
```

DELETE from Carros where Id = 2;A novidade neste exemplo é o comando sqlite3 para deletar uma linha a partir de um id:

Leia: Apague de carros a linha onde o id for igual a 2;

Para apagar todos os dados da tabela use:

DELETE from Carros;

Para apagar completamente uma tabela:

DROP TABLE Carros;

7.2 Atualizando um dado da tabela:

Salve o programa como atualiza_bd.prisma

Prisma

```
local sql = inclua'sqlite3';

ret , bd = sql.abra('teste.bd');

se ret <> sql.SQLITE_OK entao

imprima( 'erro ao tentar abrir banco de dados, ENTER para sair');

leia();

sis.saia();

fim

sql_str = [[ UPDATE Carros set Nome = 'NADA_AQUI' where Id = 1;

SELECT * from Carros; ]];

funcao sql_func(v , col)

para i = 1 , #v inicio

imprima( col[i] , v[i] );

```

```
fim

fim

ret, msg = sql.exec(bd , sql_str , 'sql_func(%s,%s)' );

se ret == 0 entao imprima("sucesso!!!");

senao imprima("Erro" , msg );

fim

leia();
```

O comando sqlite3 para atualizar um dado é:**Detalhes:**

[[UPDATE Carros set Nome = 'NADA_AQUI' where Id = 1;]]

Leia-se ATUALIZE Carros defina Nome igual a 'NADA_AQUI' onde o Id for igual a 1;

basta executar essa string e a linha que tiver o id 1 na coluna Nome será trocado o valor por 'NADA AQUI';

Obs. para mudar o valor de outras linhas troque o Id = 1 por Id = 2; ou 3, 4, ou 5 etc.

Para mudar não só a coluna nome mas outras colunas também basta acrescentar na mesma string sqlite3 o nome das outras colunas, assim:

[[UPDATE Carros set Nome = 'NADA_AQUI' , Preco = 77777 where Id = 1;]]

Veja que ao acrescentar mais uma coluna devemos separá-la por virgula.

[[SELECT * from Carros;]]

Este outro comando na mesma string sqlite3 serve para selecionar a tabela Carros para imprimirmos o valor modificado.

O comando sqlite3 completo acima com mais uma coluna para modificar o valor fica:

Prisma

```
sql_str = [[

UPDATE Carros set Nome = 'NADA_AQUI' , Preco = 77777 where Id = 1;

SELECT * from Carros;

]];
```

bastando executar essa string:

ret, msg = sql.exec(bd , sql_str , 'sql_func(%s,%s)');

**Note que a string ficou atribuída na variável sql_str e passada como argumento na função sql.exec.*

Dicas finais:

***Veja que é simples executar um comando sqlite usando a função sql.exec();**

***Aqui explanei o básico, mas você pode facilmente aprender novos comandos sqlite e executar.**

***Você pode salvar os comandos sqlite em um texto e usar a função de leitura de arquivo para obter o conteúdo em uma string e executá-la.**

Qualquer dúvida:

Visite o site: linguagemprisma.br4.biz ou o fórum (link no site)

Até logo;

Att. Adalberto.

Fim

Você pode ajudar! Compartilhe:

Edit