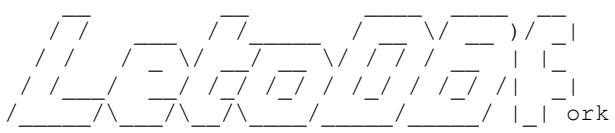

Bem-vindo ao LetoDBf

Conteúdo
---

0. tl; dr

Nos capítulos seguintes, com muitas palavras, é descrito o uso estendido do par:
<client> e <server>.
O <server> é uma construção executável com o Harbour rodando em uma rede,
e <client> se comunicando com o servidor é o seu projeto vinculado a esta biblioteca,
Então você obtém um R-eplaceable D-atabase D-river RDD "LETO", selecionado depois de conectado como driver DB padrão,

```
 se não for explicitamente fornecido em funções como DbUseArea (), para usar um arquivo
local do cliente com, por exemplo, "DBFNTX".
 Bancos de dados e ordens de índice são então usados pelo servidor, o cliente solicita
ao servidor sobre os registros
 e os armazena em cache cronometrado na RAM do cliente para acesso posterior.

 Para um teste inicial rápido, os usuários do Harbour precisam ler os capítulos / e
fazer:
 # 2.1 -> construção do executável do servidor // lib do cliente:
   hbmk2 letodb [svc] // hbmk2 rddletoaddon
 # adapt server dataroot- & log- path em bin / letodb.ini
   3. -> iniciar executável [ou serviço]
   verifique o servidor e trabalhe com um ou mais exemplos de teste
 # 5.1 -> construa seu projeto: hbmk2 yourapp [.prg | .hbp] letodb.hbc
   [não xHarbour] fornece uma cópia de "tests / rddleto.txt" renomeado como ".ini" junto
com
   seu executável - adapte-o IP | Nome DNS do servidor ou use "DETECT"
 # inicie seu aplicativo ...

# realmente encontrou uma sobra de BUG? - tente acertar com um trecho, relatório !!



      1. Estrutura do diretório

      bin / - arquivo executável do servidor
      incluir / - arquivos de cabeçalho de origem
      biblioteca lib / - rdd
      fonte/
          cliente / - fontes de lib RDD do cliente
          comum / - alguns arquivos de origem comuns para servidor e cliente
          servidor / - fontes de servidor
          3ª / - fonte de terceiros
          3rd / lz4 - biblioteca de compressão LZ4
      testes / - programas de teste, amostras
      útil/
          gerenciador / - utilitários de gerenciamento de servidor
          backup / - demonstração de backup de um servidor em execução
          uhura / - detecção automática de IP do servidor



      2. Construindo binários

* Este software foi desenvolvido para ser você mesmo criado a partir do código-fonte! *
 * Portanto, é recomendável verificar a versão atualizada das origens de download
fornecidas abaixo *

 Obtenha e construa o fantástico Harbour:
    O servidor letodb pode ser compilado apenas pelo compilador Harbour> = V3.0.
    É altamente recomendável baixar e construir o Harbour a partir da fonte 3.2 recente:
       git clone https://github.com/harbour/core.git
    Para isso, você precisa do C-Compiler usado para o Harbour em seu caminho de
pesquisa do sistema operacional.
 Ou use o pacote binário Harbour mais recente ('noturno'):
       https://sourceforge.net/projects/harbour-project/files/
       https://github.com/vszakats/harbour-core/releases
    Afinal, o caminho para o executável 'hbmk2' também é adicionado à lista de caminhos
de pesquisa do sistema operacional.
    Siga as instruções encontradas com o Harbour.

 Obtenha a fonte mais recente do LetoDBf
    com GIT:
       git clone https://github.com/elchs/LetoDBf.git
    ou como pacote embalado em:
       https://github.com/elchs/LetoDBf
       ZIP: https://github.com/elchs/LetoDBf/zipball/master
```

```
       TAR: https://github.com/elchs/LetoDBf/tarball/master
    e mude na janela de comando para o diretório raiz do pacote de download.


      2.1 construindo letodb com hbmk2, para todos os compiladores C

 Próprio servidor:
    letodb.hbp é um servidor configurado pronto para daemon Windows e Linux,
    letodbsvc.hbp é um servidor configurado pronto para uso como serviço do Windows.
    - Serviço do Windows hbmk2 letodbsvc
    - todos os outros sistemas operacionais hbmk2 letodb

 Biblioteca cliente:
    - todos os sistemas operacionais: hbmk2 rddleto
    O recomendado é integrar a biblioteca cliente LetoDbf em seu ambiente Harbour como
um 'complemento':
    - todos os sistemas operacionais: hbmk2 rddletoaddon

    Se o usuário Linux 'instalou' o Harbour, você precisa de direitos de root para
também instalar o LetoDBf como 'addon':
    - Linux: [sudo] hbmk2 rddletoaddon


 O executável do servidor resultante será encontrado no diretório "bin", a biblioteca
estará em "lib".
 No diretório "bin" também está o arquivo "letodb.ini" para configurar o servidor.

 Depois de construir com sucesso como 'addon', você pode compilar * em qualquer lugar *
seus aplicativos com:
    hbmk2 your_application letodb.hbc
 caso contrário, você deve apontar para o "letodb.hbc", exemplo de um subdiretório em
LetoDBf:
    hbmk2 your_application ../letodb.hbc

 ! * Se "letodb.hbc" não for usado, verifique a seção: 5.1.2 criando seu aplicativo *!
 como garantir manualmente o que o "letodb.hbc" automatiza.

 Para o primeiro propósito de teste, é recomendado deixar o executável do servidor
permanecer no "bin"
diretório do seu pacote LetoDBf. O seguinte irá apenas copiar o executável e letodb.ini
 para outro lugar, você pode conhecer um lugar melhor para eles e fazer isso manualmente.
 Para instalar o servidor LetoDBf nos caminhos de pesquisa do sistema operacional:
 - Linux com Harbour 'instalado':
                        sudo hbmk2 letodbaddon.hbp
 - todos os sistemas operacionais: hbmk2 letodbaddon.hbp
 Em seguida, o executável do servidor vai para o local onde está o diretório executável
do Harbour.
 No Windows, o letodb.ini também vai para o mesmo lugar, no Linux ele vai para: "/ etc",
 onde você precisa de direitos de administrador para alterar as opções de configuração.
! A instalação do LetoDBf precisa de comentar e ajustar o <LogPath> em letodb.ini!
 Use, por exemplo, o diretório temporário do seu sistema operacional, onde usuários
normais têm direitos de gravação, por exemplo: "/ tmp".

 Role para baixo, continue lendo na seção: 3. Executando e interrompendo o servidor


      2.2 Compilador Borland Win32 C ++
      2.3 Compilador MS Visual C antigo

 Se a maneira descrita acima de compilar com arquivos ".hbp" não funcionar (configuração
errada?, Sem hbmk2),
para BCC e MsVc mais antigo existe um make_b32.bat e um make_vc.bat. Analise, adapte a
pesquisa de sistema operacional
caminhos para apontar para o Harbour e seu executável C-compilador. Mais importante é
definir:
"HB_PATH" para apontar para a base! diretório do Harbour, por exemplo, "C: \ Harbour"
 Chamados sem argumento, eles criam a biblioteca cliente C-API, chamada com "full" como
primeiro argumento
```

eles constroem o servidor LetoDBf e a biblioteca do cliente Harbour.
Você saberá o que fazer por conta própria. Eu os uso apenas para testes de compilação esporádicos.

BCC55 e talvez também os mais novos têm um problema com a compilação da biblioteca de compressão LZ4, você vai
obtenha neste caso uma compressão ZLib mais lenta. Isso deve se encaixar na biblioteca do cliente e no servidor quando você
deseja usar compressão de tráfego de rede. É configurado por este "{! Bcc}" no início dos arquivos ".hbp".

2.4 Old Harbour 3.0

Basicamente, é possível compilar e usar o LetoDBf com o Harbour versão 3.0 anterior.
Para isso, você deve pesquisar nos arquivos HBP nomeados acima para a linha com: "# -cflag = -D__HARBOUR30 __ = 1",
e lá para remover o único caractere '#' no início da linha, o que significa que a linha foi comentada.
Esta deve ser a última solução, pois você perderá alguns novos recursos fantásticos do Harbour 3.2 e
em vez disso, consiga alguns bugs corrigidos.

Como o hbmk2 make tool v 3.0 não conhece a opção "-env:" nos arquivos HBP, você deve definir
estes como variáveis ??de ambiente. Portanto, para definir as variáveis ??de ambiente: __LZ4 = sim e __PMURHASH = sim
para obter os padrões para usar a compactação LZ4 e o algoritmo PMurHash. Isso é feito em seu terminal por
Windows com: SET ... = ... e no Linux com: export ... = ...
O Hbmk2 3.0 não faz isso '#include rddleto.ch' automático em seus projetos, mas pode ser feito por
usando a chave do Harbour: "/u+rddleto.ch".

2,5 x Harbour

SERVIDOR: o próprio servidor deve ser construído com Harbour, não pode ser feito com xHB.
O mesmo se aplica a utilitários como o monitor do console.

CLIENTE: a biblioteca cliente (RDD) pode ser construída com xHarbour, use a definição 'rddleto.lib.xbp' para
xBuilder. Para Windows (mas não para XCC), ele usará por padrão um segundo thread (sem HVM),
portanto, o executável deve ser vinculado a uma biblioteca contendo '_beginthreadex ()'.
cFlag define: LETO_NO_THREAD = 1 definido para xHB irá desabilitar isso e a necessidade de função de threading,
[Compilador C: observe que o xBuilder não armazena o compilador C usado - altere-o sob demanda.
XCC: não é possível compilar 'lz4.c' de terceiros, compilá-lo com PellesC> = 4.5 manualmente,
e substitua-o na lista de arquivos do xBuilder pelo 'lz4.obj' resultante:
pocc.exe -Fo "obj \ lz4.obj" -Ot -I "incluir" -I "fonte \ 3rd \ lz4 \ lib" -I% PATH_XHB% "\ incluir"
-I% PATH_POCC% "\ Incluir" -I% PATH_POCC% "\ include \ Win" "fonte \ 3rd \ lz4 \ lib \ lz4.c"
]

DEMO: uma única demonstração 'test_mem.exe.xbp' foi projetada e testada com PellesC (POCC) V8.0 [> = 6.0]
Para esta lib 'crtmt.lib' está na lista de links, outro compilador C pode substituir aquele "crtmt.lib" por um dos
sua distribuição (cw32mt.lib, libcmt.lib ..)
XCC e RDD lib com thread desabilitado precisam remover a biblioteca da lista.
Da mesma forma, você pode construir outros exemplos "test_ [func | filt | dbf | dbfe | var | arquivo]"

SEU APP:
como a demonstração acima: vincule uma biblioteca de tempo de execução MultiThread C,
#include "rddleto.ch" para cada '.prg' de um projeto xHB LetoDBf pela opção xHB:
"/u+rddleto.ch".
Um arquivo fonte do seu projeto, eu sugiro que com a função main () e Leto_Connect (),
deveria: SOLICITAR LETO
(*) Os nomes das páginas de código de xHB e a construção do servidor com o Harbour
podem ser diferentes, que precisam ser
configurar uma 'tabela de tradução de nomes' - ver LETO_ADDCDPTRANSLATE ()

2.6 Biblioteca C-API

Isso é para usar uma parte da biblioteca cliente de nível C puro * apenas * com um
compilador C.
Todos os métodos Harbour RDD (leto1.c) e todas as relações com as funções de nível PRG
(letomgmn.c) são
deixado de fora para este propósito. É construído com:
hbmk2 apileto.hbp
O libleto. [A | lib] resultante é encontrado posteriormente no diretório lib.
Exemplos curtos dados em: tests / c_api / *. C devem dar a um desenvolvedor C um pouco
experiente uma visão rápida.
Para construir os exemplos, use nesse diretório um bldc.bat [BCC] ou um bldc.sh [GCC].
Adapte aí os caminhos para o diretório bin do Harbour e do compilador C, chame-os com o
nome do arquivo sem
extensão como primeiro parâmetro, por exemplo;
bldc. [bat | sh] test_dbf
O resultado é executado com o primeiro parâmetro IP | nome DNS do servidor a ser
conectado, se não for fornecido 'localhost'
aka '127.0.0.1:2812' será usado.

Notas: a biblioteca C-API usa hb_xgrab [z] (), hb_xrealloc () e hb_xfree () da Harbours.
Essas memórias
alocações / liberações podem coexistir com outro sistema de memória, mas a memória
alocada com
um sistema deve ser liberado pelo mesmo. Portanto, uma lista de bibliotecas Harbour é
necessária para compilações estáticas,
causado por hvm. [a | lib] contendo as funções de memória, mas também links para outras
bibliotecas.
O link para a lib do Harbour dinâmico facilitaria muito isso, o exemplo é destacado nos
arquivos em lote.

É possível executar funções no servidor com LetoUdf (), mas para isso alguns
conhecimentos básicos
de tratamento de PHB_ITEMs é necessário para transmitir parâmetros de função para o
servidor e para lidar com os recebidos
resultado. Observe o segundo parâmetro de LetoUdf (, LETOTABLE * pTable, ..): se não for
NULL, este WA
será pré-selecionado no servidor antes que a função remota seja executada. [ver também
4.2.1 suporte UDF]

Nenhum Harbour HVM é iniciado, então tome cuidado ao chamar as funções internas do
Harbour que precisam disso,
por exemplo, funções de retransmissão na pilha de um HVM como hb_setGet .. (). Além
disso, o sistema de página de código não é
inicializado - isto é, com a intenção de manter o uso da lib a <raw> C-API possível.

3. Executando e parando o servidor

Antes de fazer isso, adapte o "DataPath" em letodb.ini, a configuração mais importante,
consulte 4.1
Se este caminho não existe ou é inválido, o servidor não iniciará!
Se você 'instalou' o servidor ou o usa como serviço, também comente e adapte o LogPath,
para onde os arquivos de log irão. Se LogPath não estiver definido, eles vão para o
diretório do executável do servidor.

Para ambos os diretórios, o usuário precisa de direitos de gravação concedidos pelo sistema operacional.

3.1 a maneira clássica para todos os sistemas operacionais

Inicie, no modo padrão __WIN_DAEMON__ ou ambos os modos __LINUX_DAEMON__ e __CONSOLE__
```
start / B letodb.exe (Windows)
./letodb (Linux)
```

Para desligar o servidor, execute o executável com o parâmetro 'stop':
```
letodb.exe stop (Windows)
./letodb stop (Linux)
```

Para recarregar o módulo "letoudf.hrb", contendo funções do lado do servidor UDF, ele deve estar no mesmo
diretório igual ao executável do servidor, use o parâmetro 'reload':
```
recarregar letodb.exe (Windows)
./letodb reload (Linux)
```

Os exemplos acima usarão o arquivo de configuração 'letodb.ini' padrão. Para usar um nome explicitamente
adicione estes dois parâmetros: <start-command> config myini.ini
ou use um terceiro parâmetro para parar / recarregar, por exemplo: ./letodb stop myini.ini

Linux: é necessária uma pausa de 1 a 2 minutos antes de reiniciar o servidor após um desligamento.
Para automatizar isso, use o script bash: 'leto.sh' no diretório "bin", ele iniciará o servidor
quando for novamente possível. É sobre o tempo que deve decorrer antes que o TCP / IP possa liberar um
fechou a conexão e reutilizou seus recursos. Isso é conhecido como estado TIME_WAIT.
Windows: acima do comando 'start / B' para evitar uma janela preta pode ser colocado em um arquivo de lote '.bat'.
Existem algumas ferramentas minúsculas, também conhecidas como executáveis ??para iniciar outro processo 'escondido'.
(exemplo incl. C-source encontrado em: http://www.commandline.co.uk/chp/).

-> Procure em letodbf.log, esse servidor está ativo - também mais erros relacionados ao servidor vão aqui.
No mesmo local, em 'letodbf_xx.log', pode aparecer um feedback de erro de uma conexão específica.
Ambos podem ser facilmente visualizados com: 8.1.1 Todos os consoles de SO

3.2 Executar como serviço Windows @

Para uso como "serviço do Windows", o executável do servidor deve ser compilado para esta tarefa, consulte 2.1
Para instalar o LetoDbf como serviço, o executável deve ser colocado em um diretório coberto pelo sistema operacional
caminhos de pesquisa do sistema a serem encontrados em qualquer lugar. Em seguida, execute letodb com o parâmetro 'install':
```
instalação de letodb.exe [letodb.ini]
```
O terceiro parâmetro é opcional para diferentes configurações a serem usadas para vários serviços LetoDBf, por exemplo
rodando em diferentes partições de disco.

Verifique em letodbf.log se o serviço foi instalado e iniciado com sucesso.
Para verificar o estado de um serviço do Windows, use o gerenciamento de GUI para serviços.
Alternativamente, na linha de comando pode ser usado para iniciar / parar o serviço:
```
net start LetoDBf_Service
net stop LetoDBf_Service
```

Para desinstalar o serviço novamente, execute letodb com o parâmetro 'uninstall':

```
      desinstalação de letodb.exe [letodb.ini]
Pode ser necessário reiniciar sua máquina Windows. Você pode verificar com antecedência
se o serviço é
 ainda listado no gerenciamento de GUI para serviços, caso contrário, não é necessário.



      4. Configuração do servidor

      4.1 letodb.ini

 No ambiente Windows, o arquivo de configuração letodb.ini deve ser colocado no
diretório do executável do servidor.
 No Linux, o servidor procura por ele no diretório "/ etc", se não for encontrado no
diretório de
 executável do servidor.
 Este arquivo é lido apenas uma vez ao iniciar o LetoDBf, após as alterações nele você
deve reiniciar o servidor
 para deixá-lo ficar ativo.

 As opções realmente importantes normalmente são: DataPath, LogPath, Share_Tables,
No_Save_WA.
 Se o servidor deve usar um conjunto de arquivos DBF existente, adapte 'Default_Driver'
se DBFNTX foi usado.

 Outros valores padrão no arquivo de configuração distribuído são escolhidos para
satisfazer as primeiras necessidades mais comuns,
 pode ser endurecido e otimizado ocasionalmente.
 Os novatos do LetoDBf agora gostam de continuar lendo a seção: 5. Como trabalhar com o
servidor LetoDBf.

      Atualmente existem os seguintes parâmetros (os valores padrão são designados).

      [A PRINCIPAL]
      IP = - Interface fornecida pelo endereço IP, onde o servidor Letodb escuta as
conexões.
                                    Vazio (padrão) é recomendado para uso da intranet,
então todas as interfaces
                                    disponíveis no computador, incluindo o dispositivo
virtual de 'loopback'.
                                    Se definido, apenas esta interface (fornecida pelo
endereço IP) é usada.
                                    O usuário Unix pode alternativamente usar nomes de
interface como: 'eth1'
                                    Se a Internet estiver disponível na máquina, leia
também a seção 4.5 Segurança
      Servidor = - endereço IP usado por ferramentas como o console de gerenciamento
para localizar o servidor.
                                    Pode ser, mas não deve ser, o mesmo usado para a
opção de configuração 'IP'
                                    e é apenas por conveniência.
      Porta = 2812 - número da porta do servidor, o padrão é 2812 [então 2813 usado para
o segundo soquete]
                                    Existem dois! portas usadas pelo servidor, este e o
número seguinte.
                                    ! Você já se conectou ao primeiro número de porta!
      DataPath = - PATH para um diretório base em um servidor com seus bancos de dados,
                                    pode incluir também uma letra de unidade para
sistemas Windows deficientes
      LogPath = - se o nível da opção de configuração <DEBUG> for maior que zero [0],
                                    PATH para um diretório (com acesso de gravação) para
todos os arquivos de log.
                                    O arquivo letodbf.log do servidor principal conterá
algumas informações das configurações
                                    na hora de início do servidor, além de informações
sobre novos clientes conectados e desconectados
      No_Save_WA = 1 - modo de servidor de manipulação interna de tabelas de banco de
dados
```

```
                                    1 cada dbUseArea () causará uma operação real de
abertura de arquivo pelo SO,
                                      idêntico ao que o cliente solicitou, então as
áreas de trabalho no servidor são as mesmas que
                                      no lado do cliente. [Número WA, alias, condições
de filtro, relações]
                                    0 cada mesa é aberta apenas uma vez, esta área de
trabalho 'trocada' entre o cliente
                                      solicitações de. portanto, apenas uma conexão
terá acesso à mesa por vez.
                                      Nenhuma relação ativa no servidor, os nomes de
alias no servidor são diferentes de
                                      o cliente.
                                   Recomende '1' se você planeja executar funções no
lado do servidor (UDF).
      Share_Tables = 0 - outras tabelas de acesso simultâneo de software usadas pelo
servidor,
                                   que muda o bloqueio de registro lógico ou físico -
na dependência:
                                   # No_Save_WA = 0
                                   0 servidor abre todas as tabelas em modo exclusivo,
o que leva a
                                      aumento de desempenho, por exemplo, bloqueios de
registro / arquivo não são aplicados pelo sistema operacional.
                                   1 tabelas são abertas no mesmo modo [compartilhado /
exclusivo] que o cliente
                                      aplicativos os abriram, o que permite que o
LetoDB funcione em coexistência com
                                      outros aplicativos [não usuários LetoDB]
simultâneos nas mesmas tabelas DBF.
                                   # No_Save_WA = 1
                                   1 registro físico / bloqueios de arquivo definidos
com o sistema operacional são visíveis para outros
                                   0 apenas bloqueio interno lógico, não respeite
outros bloqueios de registro
                                   * SAMBA: *
                                     trabalhar em conjunto com o serviço de arquivos
Samba precisa de tratamento / configuração muito especial
                                     e tem limites de possibilidades - ver capítulo 4.4
      Default_Driver = CDX - RDD padrão para abrir tabelas DBF no servidor, se não for
definido explicitamente em seu
                                   código fonte com leto_DbDriver ().
                                   Valores possíveis para config: CDX NTX
                                   Se o servidor estiver vinculado ao suporte de índice
rushmore,
                                   CDX obtém BMCDX e NTX se tornará BMNTX
                                   Se não for definido, o servidor por padrão usa CDX
[BMCDX]
      Cache_Records - O número padrão de registros a serem lidos no cache de leitura do
cliente,
                                   usado para pular etc sem nova solicitação ao
servidor.
                                   Os registros são válidos no cliente enquanto o tempo
limite do hotbuffer.
                                   O padrão é 10, o mínimo é 1 (desempenho lento), bons
valores são 10 - 50,
                                   teórico ! máximo de 65535. Adapte-se ao desempenho
em seu ambiente.
                                   Pode ser definido para tabelas e ocasiões
específicas com leto_SetSkipBuffer ().
      Lock_Scheme = 0 - Se> 0, o esquema de bloqueio estendido será usado pelo servidor.
                                   * Isso é necessário apenas se o tamanho do seu DBF
for maior que 1 GB. *
                                   Então DB_DBFLOCK_HB32 será usado para NTX / CDX;
                                   _ou_ se definido como 6, DB_DBFLOCK_CLIPPER2 para
NTX, HB32 para outro
                                   _ou_ se definido como 2, DB_DBFLOCK_COMIX para CDX,
HB32 para outro
```

```
                                        _ou_ se definido como 5, DB_DBFLOCK_HB64 para todos.
      Memo_Type = - DEIXE-O VAZIO, para obter o padrão para a opção escolhida
Default_Driver.
                                        Padrão: FPT para DBFCDX, DBT para DBFNTX, SMT para
outros.
      Memo_BSize = - para! Expert! usuários!, isso mudará o tamanho do bloco do memo
padrão
                                        para * NOVAS * tabelas de dados DBF criadas. Antes
de fazer isso, você precisa de uma lição sobre.
      Lower_Path = 0 - padrão 0: respeita maiúsculas e minúsculas nos nomes de arquivos
solicitados
                                        se 1, converte todos os caminhos e nomes de arquivos
em minúsculas;
                                        Isso é útil se todos os arquivos no disco estiverem
em letras minúsculas 'sugeridas',
                                        mas o nome no código-fonte é escrito em maiúsculas
ou mistas,
                                        que irá falhar para sistema de arquivos de
'armazenamento' com distinção entre maiúsculas e minúsculas,
      EnableFileFunc = 0 - se 1, uso de funções de arquivo (leto_file (), leto_ferase (),
 leto_frename () etc.
                                        é permitido. Caso contrário, essas funções não fazem
nada ou retornam .F.
      EnableAnyExt = 0 - se 1, * criação * de tabelas de dados e índices com qualquer
extensão, diferente de
                                        padrão (dbf, cdx, ntx) é permitido. Caso contrário,
eles seriam rejeitados.
      Pass_for_Login = 0 - Nível mais baixo de verificação de senha: após o login, todos
são permitidos a todos
                                        se for 1, a autenticação do usuário é necessária
para fazer o login no servidor;
      Pass_for_Manage = 0 - se 1, a autenticação do usuário é necessária para usar as
funções de gerenciamento,
                                        por exemplo, execute o console do monitor
[Leto_mggetinfo ()]
      Pass_for_Data = 0 - se 1, a autenticação do usuário é necessária para ter acesso
de gravação aos dados;
      Pass_File = leto_users - [path +] nome do arquivo do banco de dados do usuário
para autenticação.
                                        Se nenhum caminho for fornecido, ele aparecerá no
diretório do executável do servidor
      Server_User = O nome de usuário Unix / Linux de quem UID e GUI são obtidos para
__LINUX_DAEMON__.
                                        Ter precedência sobre as duas opções a seguir, se o
nome de usuário for fornecido e existir.
      Server_UID = 0 O ID do usuário e ID do grupo para o servidor Linux executar como
daemon.
      Server_GID = 0 Suas tabelas DBF receberão esses IDs, importantes para escolher os
direitos de acesso corretos.
                                        O padrão é vazio, então este será o U-ID e o G-ID
que iniciaram o servidor.
      Max_Vars_Number = 1000 - Número máximo de variáveis ??compartilhadas
      Max_Var_Size = 67108864 - Tamanho máximo na soma de todas as variáveis ??de texto
/ array, padrão 64 MB.
                                        Uma única variável de texto / matriz pode ser um
quarto disso (16 MB)
                                        Tenha muito cuidado ao aumentar impensadamente esse
valor para tamanhos muito maiores,
                                        pois o servidor precisará de pelo menos 4 vezes esse
valor como RAM.
                                        Teórico! máximo para um único! item é ~ 4 GB, então
seu servidor precisará
                                        para ter 64! GB !! RAM. [ Não testado ! :-)]
      Trigger = cFuncName - função de gatilho do lado do servidor para * cada * tabela.
! USE COM CUIDADO ESPECIAL!
                                        Se fornecida, esta função de gatilho é executada
para * todos * os WAs abertos para
```

```
                                    ações como anexar registro, atualizar, ...
                                    A função <cFuncName> deve ser conhecida no servidor
e pode ser um UDF carregado com HRB
                                    função. Ele recebe 4 parâmetros: nEvent, nArea,
nFieldPos, xTrigVa;
                                    onde nFieldPos e xTrigVal são preenchidos apenas
para eventos EVENT_PUT e EVENT_GET.
                                    (veja um exemplo de "Leto_Trigger" em: tests /
letoudf.prg)
      Tables_Max = 999 - Número de * MAXIMUM * tabelas DBF designadas manipuladas pelo
servidor,
                                    para o modo de servidor No_Save_WA == 0, são tabelas
DBF físicas,
                                    para o modo de servidor No_Save_WA == 1 são tabelas
DBF abertas por todos os usuários.
                                    Este número * não * pode ser aumentado durante o
tempo de execução do servidor.
                                    Valor teoricamente máximo: 999999, mínimo:> = 99
                                    Aumente o valor padrão grande o suficiente para suas
necessidades,
                                    Exemplo para No_Save_WA == 0: 2 * DBF existente
físico
                                    Exemplo para No_Save_WA == 1: Users_Max * DBF físico
existente
                                    (O limite máximo para uma única conexão de usuário é
cerca de ~ 60.000).
                                    ** O sistema operacional pode limitar os arquivos
abertos por 'usuário', neste caso o próprio servidor **
                                    [/etc/security/limits.conf; ... \ CurrentControlSet
\ services \ Tcpip \ Parameters]
      Users_Max = 99 - Número de * MAXIMUM * usuários designados. não defina muito baixo.
                                    Este número * não * pode ser aumentado durante o
tempo de execução do servidor.
                                    Valor teoricamente máximo: 65534. mínimo é> = 9
                                    Aumente o valor padrão grande o suficiente para suas
necessidades, por exemplo, duas vezes mais do que realmente
                                    usuários, mas não exagere.
      Depurar = 1 - nível de depuração, padrão: 1 -> apenas informações secundárias
sobre login / logout de
                                    conexões são gravadas em letodbf.log
                                    0 = nenhuma mensagem de depuração - observe que
erros reais são sempre registrados.
                                    Quanto maior o valor, mais informações são gravadas.
                                    Um valor> = 15 incluirá tráfego de comunicação
parcial para o servidor,
                                    com um valor> 20, o protocolo de comunicação
completo é registrado.
                                    !! USE COM CUIDADO !!, os arquivos de log podem
ficar muito rápidos MUITO GRANDES.
                                    Ele pode ser alterado 'em tempo real' para seções
críticas com novos
                                    RDDI_DEBUGLEVEL - ver 7.5
                                    SOMENTE aumente o valor em caso de problemas, para
rastrear o que aconteceu no servidor,
                                    e as ações do cliente. Cada conexão obterá um
arquivo de log próprio com
                                    ID de conexão como extensão de arquivo; novo criado
quando uma conexão é iniciada.
      HardCommit = 0 - Extensão Harbour para servidor, NÃO USE - Padrão: SET HARDCOMMIT
OFF [0]
                                    Pode ser uma experiência apenas para o servidor
INSTÁVEL defini-lo como <1>,
                                    o que significa que as alterações de dados são
forçadas tanto quanto possível, esvaziando os buffers de arquivo
                                    para 'write_through' o cache do sistema operacional
para armazenamento permanente - ao lado de
                                    diminuição de desempenho. Verifique os sistemas de
arquivos de journaling ou melhore o dedicado
```

```
                                        hardware de servidor como LetoDBf normalmente NÃO é
a causa para tal necessidade.
                                        A configuração 'SET HARDCOMMIT' no aplicativo do
lado * cliente é ignorada, pois qualquer pendente
                                        as alterações de dados são enviadas imediatamente
para o servidor com ação pular / desbloquear / fechar.
                                        Para atualizar o servidor com mudanças do lado do
cliente de registros bloqueados * sem * tal ação
                                        use a função DbCommit [All] () em locais especiais
onde for realmente necessário.
                                        (relacionado mais: capítulo 7.3 DBI_BUFREFRESHTIME /
DBI_AUTOREFRESH)
      ForceOpt = 0 - configuração _SET_FORCEOPT
      Allow_Udf = 0 - configuração de segurança, DEFAULT é! NÃO ! para permitir o uso de
                                        carregado UserDefinedFunction para execução remota
no servidor.
                                        Com valor 0, mesmo um Leto_UDFExist () negará para
responder.
                                        Defina como 1 para usar a funcionalidade UDF no
servidor LetoDB.
                                        0 irá desativar essa possibilidade.
      TimeOut = -1 - Tempo limite de conexão em segundos, -1 significa espera infinita.
                                        Este tempo limite determina quanto tempo uma
gravação na rede / espera pela área de trabalho solicitada
                                        irá esperar para ter sucesso, antes que o thread da
conexão desista.
                                        Se usado: Zombie_Check, este valor shell será menor
do que isso.
      Zombie_Check = 0 - Tempo em segundos, que um cliente deve ficar quieto (sem
atividade), antes
                                        uma consulta 'você está saudável' (ping) é enviada
do servidor, para verificar se não é um
                                        conexão morta / desconectada. ! O aplicativo deve
ser multi-thread vinculado ('-mt')!,
                                        caso contrário, essas verificações não podem ser
feitas.
                                        Como 3 vezes em um determinado intervalo, uma
verificação é feita, um zumbi pode ser 1/3 vez
                                        mais 'morto', por exemplo, 60 ==> max. 80 segundos
'mortos' antes de serem detectados.
                                        Essa conexão será encerrada, os arquivos abertos e
os bloqueios serão reiniciados.
                                        Se definido como 0 [padrão], essas verificações são
desativadas.

     ; BC_Services = letodb; - Servidor BroadCast 'Uhura' integrado, <off> padrão como
comentado:
                                        ativa o serviço de resposta BC BroadCast para a
lista de 'nomes de serviço',
                                        cada nome deve terminar com um ';' - um pedido irá
obter de volta o IP do servidor
     ; BC_Interface = eth2 - experimental / Linux: use apenas interface específica para
resposta
     ; BC_Port = 2812 - especificar uma porta diferente da padrão para interfaces UDP BC,
                                        por padrão, a mesma porta configurada para a porta
TCP
     ; IP_SPACE = - como um firewall: contém uma parte do endereço IP à esquerda que
deve ser
                                        encontrado no IP que deseja conectar ao servidor.
                                        Várias partes podem ser definidas, delimitadas por
';' (sem espaços extras)
                                        Exemplo para duas intranet NIC: IP_SPACE =
192.168.0.; 192.168.1 .;
     ; SMB_SERVER = 0 - defina como 1 para uso simultâneo com Samba e leia 4.4 Samba
     ; SMB_PATH = - em conjunto com SMB_SERVER = 1, para ambas as opções leia mais
                                        em 4.4 serviço de arquivo Samba
     ; SVC_NAME = - utilizável para o serviço do Windows para definir um nome de serviço
diferente para
```

```
                                        vários servidores em execução na mesma máquina, o
padrão é "LetoDBf_Service";
                                        comumente usado junto com diferentes nomes de
arquivos para configuração
     ; Crypt_Traffic = - se definido como '1' [default = 0], ele reforça a criptografia
do tráfego de rede,
                                        aka todas as conexões que não usam criptografia são
bloqueadas / desligadas.
                                        Um Leto_ToggleZip () usado manualmente não terá
efeito, também conhecido como compressão
                                        com criptografia não pode ser desativado.
     ; Backup_Info = ... Contém uma string com linhas separadas por vírgulas para ser
mostrada na 'caixa de backup'
                                        nos clientes, quando a função: Leto_LockLock () é
executada.
                                        String padrão: BACK-UP, WAITING, ESC-> GO, ESC-> QUIT
                                        mostra 3 linhas e as duas últimas são para o modo
reclamar / final
                                        Uma opção não comentada com string explicitamente
vazia é ativada
                                        Comportamento de 'estilo antigo' para Leto_LockLock
().
     ; Data_LogFile = para ativar um registro de alteração de dados no nome de arquivo
fornecido.
                                        Se nenhum caminho for fornecido ou o caminho começar
com "./", o local é "DataPath"
                                        ou em um determinado subdiretório para aquele, caso
contrário, um determinado caminho é tratado como absoluto
                                        caminho possível localizado em outro armazenamento.
Durante a inicialização do servidor, um caminho para
                                        o arquivo é verificado e criado se não existir.
```

4.2 Diferentes configurações / extensões de compilação de servidor

4.2.1 Suporte UDF

```
 Além de chamar o comando único do Harbour com leto_UDF ("cCommand" [, xParam]),
 você pode carregar suas próprias funções de nível PRG com um arquivo <HRB> também
durante a execução do servidor.
 Um exemplo muito básico é encontrado em: tests / letoudf.prg.
Como compilar um PRG para um HRB, consulte letoudf.hbp. Isso é chamado com: hbmk2
letoudf.
 Coloque o arquivo <HRB> resultante no mesmo diretório do executável do servidor.
 Após o comando "reload" ou junto com o servidor iniciar você tem uma entrada em
letodbf.log se eles
foram carregados com sucesso. Em caso de erro, você também encontrará um pequeno texto
que falhou.
 Veja mais em Leto_Udf () ...

 Para a execução de funções Harbour no lado do servidor, ou quando suas funções no
arquivo HRB
 precisam de comandos do Harbour (como "STR", "DTOC"), essas funções do Harbour devem
ser vinculadas durante a compilação
 no executável do servidor.
 Por padrão, os mais comuns usados já estão disponíveis, feitos por um REQUEST em source
/ server / server.prg.
 Se você precisar de alguma função central especial do Harbour, você pode adicionar seu
próprio REQUEST na fonte,
 ou habilite facilmente o conjunto de comandos Harbour completo ativando (remova '#') em
letodb.hbp.
 O conjunto completo da função Harbour Cl * pper tools contrib [CT] pode ser obtido no
mesmo lugar.

* No modo servidor No_Save_WA = 0: para todas as tabelas; e para todas as tabelas
HbMemIO em qualquer modo de servidor: *
 no lado do servidor, o nome ALIAS para uma área de trabalho é * diferente * daquele
usado em seu aplicativo.
```

O cliente ALIAS é traduzido * automaticamente * se fizer parte da string <cCommand>, por exemplo, em blocos de código.
Se você deseja acessar várias áreas de trabalho simultaneamente em seu UDF, e tal WA não é o ativo,
As funções de UDFs em HRB são necessárias. Aqui, então, para trabalhar com Leto_Select () para WA e WA especificados acima
Leto_Alias() para todas as outras tabelas para consultar o nome WA ALIAS no servidor.
O ALIAS no servidor para essas tabelas não é previsível porque criado dinmicamente,
(Para sua informação: o esquema é: "Exxxxxxx", onde x é um número de ocorrências sequenciais para criar um ALIAS global
para o servidor para todas as conexões - é reutilizado depois que uma área de trabalho é fechada pela última vez.)

4.2.2 Suporte a página de código

Este tópico está relacionado principalmente a arquivos de índice.
Cada conexão pode usar uma página de código diferente.
Cada conexão é, na verdade, limitada para usar a mesma página de código para todas as suas tabelas.
Você deve evitar abrir o * mesmo * DBF, também conhecido como seus arquivos de índice com páginas de código diferentes.
Esses limites podem ser estendidos, se possível, no futuro, se for realmente necessário.
É importante, com qual configuração de CP o índice foi criado e, posteriormente, os dados DBF são modificados.

Se suas perguntas agora são: o que é uma página de códigos? - como determino a página de códigos usada em meu código-fonte?
- qual é o comando para alterar no meu código-fonte a página de códigos? - e perguntas semelhantes ...
indicar que você está usando as configurações padrão. Então você está quase terminando esta seção.

No arquivo: source / include / letocdp.ch
você pode adaptar a lista de páginas de código disponíveis. Eles podem então ser habilitados / carregados para uma conexão de cliente.
! Depois de alterar o conteúdo desse arquivo de inclusão, você deve reconstruir o servidor!
Os nomes nesse arquivo são os mesmos que você usa em seu código-fonte, mas com o prefixo: HB_CODEPAGE_MyCodepage

Acima do shell é a maneira recomendada de adaptar a lista de páginas de código possíveis.
Como alternativa, você pode ativar * todas * as páginas de código conhecidas do Harbour fazendo comentários em:
letodb.hbp (ledodbaddon / letodbsvc) a linha com: "__HB_EXT_CDP__" [remova o '#' no início]

4.2.3 Suporte a índice de bitmap Rushmore

Antes que você acredite, esta será a mãe dos problemas de desempenho de filtragem, digamos que ela não é.
Melhor investir um pouco de tempo sobre como obter filtros * do lado do servidor * otimizados com a ajuda do Leto_Var * ()
sistema, consulte a seção 5.2 e procure a ideia Leto_VarGetCached () ...
Porque para que a matriz com números de registro seja definida como válida, ela deve ser ignorada uma vez no
banco de dados inteiro, em vez de sugerir primeiro onde isso é feito para tantos registros conforme o necessário.
Para atualizar este 'conjunto fixo', novamente uma execução completa é necessária, onde a sugestão acima é apenas atualizar a expressão.

O servidor e a biblioteca cliente podem ser construídos com suporte do driver BMDBFCDX / BMDBFNTX / BMDBFNSX.
Nesse caso, esses RDD serão usados por padrão, também conhecido como e, g. BMDBFCDX se "CDX" for encontrado em letodb.ini

Basicamente, eles suportam a mesma funcionalidade do CDX / NTX clássico, mas existem cinco funções para definir
filtros de bitmap, consulte a seção 7.11
Para construir um servidor para isso, você precisa descomentar no arquivo hbp / letodb.hbp, rddleto.hbp) uma macro:
"__BM", feito com a remoção do '#' no início das linhas com aquele "__BM".
Em seguida, recompile * ambos * os lados do LetoDBf, também conhecido como executável do servidor e biblioteca cliente.
Como eles são implementados como uma chamada UDF, você também deve definir em letodb.ini Allow_UDF = 1.

4.2.4 Compressão LZ4 / ZLib

O padrão é usar o algoritmo de compressão LZ4 em tempo real de alta velocidade, mas o compilador Stoneage BCC 5.5 não pode usá-lo!
Isso pode ser alterado para compressão ZLib clássica (mais lenta!), Comentando com definir um <#> na primeira posição
dos arquivos .hbp correspondentes, encontrados no início da linha com: "# -env: __ LZ4 = yes"
Eu recomendaria o LZ4 extremamente rápido.
Recompile ambos !! executável do servidor e biblioteca do cliente. Isso deve se encaixar,
um servidor com LZ4 não vai entender
um aplicativo cliente com ZLib.

Observação adicional: pelo menos o Stoneage old BCC 5.5 teve problemas com o uso de LZ4,
por isso não foi comentado para todos os BCC
versões. Se quiser experimentá-lo com uma versão BCC mais recente, você deve remover: "{! Bcc}" dos arquivos HBP.

4.2.5 opções especiais de construção

Alguns sinalizadores especiais podem ser passados com hbmk2, a forma universal é:
hbmk2 ... -cflag = -D ... = ...
hbmk2 ... -prgflag = -D ... = ...
Isso define um: #define para o valor, para ser ativo para fontes PRG- e / ou C-. Como não é permitido redefinir
um valor já definido, isso é possível apenas para alguns #defines usados no LetoDBf.
Exemplo:
hbmk2 letodb -cflag = -DLETO_SENDRECV_BUFFSIZE = 32767
será semelhante ao C-source: #define LETO_SENDRECV_BUFFSIZE 32768

Sinalizadores de origem C:
LETO_SENDRECV_BUFFSIZE
altere o tamanho padrão de 64 KB para buffers de envio / recebimento / criptografia para o executável do servidor.
Use a seguinte fórmula para valores diferentes: (x * 8192) - 1
Ele pode ser usado para poupar um pouco o uso de RAM do servidor, se necessário, valores baixos levarão à realocação frequente,
e maior que 64 KB raramente aumentam o desempenho. Com nível de depuração> 20, você encontra dicas sobre essa atividade.
Ele também pode ser usado para a biblioteca cliente com, por exemplo, hbmk2 rddleto, mas como seu aplicativo normalmente tem apenas
poucas conexões terá menos influência no uso de RAM.

LETO_USERPASS
Senha interna para o banco de dados de credenciais do usuário, consulte 4.3 Autenticação

LETO_MYPASSWORD
Senha codificada no executável do servidor * e * na biblioteca do cliente: devem ser iguais para ambos os lados!,
caso contrário, a conexão com o servidor falhará, consulte: 4.5 Segurança

Sinalizadores de fonte PRG:
LETO_FULLCMDSET_HB
LETO_FULLCMDSET_CT

```
 Esses sinalizadores ativam o Harbour completo e / ou o conjunto completo de CT [Cl *
pper Tools] contrib.
 Use-os apenas se realmente necessário e, antes de fazer isso, verifique se uma função
desejada ainda não está
 PEDIDO em 'source / server / server.prg' - melhor informar o desenvolvedor LetoDBf para
adicionar um importante ausente,
 e também pode ser feito pelo próprio usuário final.
```

4.3 Autenticação

```
 Para ligar o sistema de autenticação, também conhecido como login no servidor com nome
de usuário / senha necessários,
 você precisa definir um dos seguintes parâmetros letodb.ini para 1:
 Pass_for_Login, Pass_for_Manage, Pass_for_Data.
! De antemão, você precisa criar pelo menos um usuário com direitos de administrador,
porque quando a autenticação
 sistema está ativo, apenas usuários autenticados com direitos de administrador são
capazes de adicionar / alterar usuários
 e senhas.

 Para adicionar um primeiro usuário, você precisa executar LETO_USERADD () uma vez, por
exemplo:
      LETO_USERADD ("admin", "secret:", "YYY")
 onde "secret" é a senha e "YYY" é uma string de 3 letras Y_es e N_o,
que concede direitos para; 'admin' 'gerenciar' 'acesso de escrita'.

 Você também pode usar o programa de console em utils / manager / console.prg para
adicionar / excluir usuários.

 Para se conectar a um servidor com nome de usuário e senha quando a autenticação é
ativada,
 você deve usar a função LETO_CONNECT ().
 Em seguida, também o monitor do console precisa fazer o login com nome de usuário /
senha - consulte a seção 8.1 para parâmetros.

 As credenciais do usuário são lidas durante a inicialização do servidor, seu conteúdo é
mantido na memória,
 aka deletar este arquivo requer a reinicialização do servidor para redefinir (além de
revogar a autenticação necessária).
 O local padrão no diretório do servidor pode ser adaptado com a opção de configuração:
'Pass_file' com um caminho [absoluto].
 As credenciais do usuário são criptografadas com uma senha padrão gravada no executável,
 que pode ser personalizado para o seu ambiente, por exemplo:
     hbmk2 letodb -cflag = -DLETO_USERPASS = AbsoluteSureNobodyKnowMe
 Um novo executável com uma senha diferente torna as credenciais do usuário antigas
inválidas e precisa ser iniciado do scatch.
```

4.4 Serviço de arquivo Samba

```
 Se os arquivos servidos por um servidor Samba a seus clientes precisarem ser
compartilhados com um servidor LetoDBf,
 LetoDBf deve ser configurado de forma especial.

 Antecedentes técnicos: <exclusive> A sinalização do sistema de arquivos do SO para um
arquivo aberto não está definida, nem mesmo
 verificado pelo servidor Samba - tratado apenas internamente no Samba, não é visível
para o servidor Linux.
 Além disso, uma tabela aberta <exclusiva> feita em uma unidade de rede CIFS aparecerá
sempre como uma <shared>,
 * então, tal tabela <exclusiva> aberta por LetoDBf pode ser aberta compartilhada por um
aplicativo Samba *
 O servidor LetoDBf faz o seu melhor para garantir a integridade dos dados, respeita
tabelas exclusivas abertas pelo Samba,
 mas precisa de uma verificação adicional para tabelas <exclusive> abertas por ele.
```

```
 usado para tráfego de rede em tempo real sob demanda:
 onde é servido e onde o (s) servidor (es) LetoDBf serão executados simultaneamente nos
mesmos dados.
 (Normalmente você precisa do pacote: <cifs-utils> para poder montar um sistema de
arquivos de rede CIFS.
   Crie um ponto de montagem [/ mnt / samba] e monte o compartilhamento nele;
   isso pode ser feito, por exemplo, em / etc / fstab ou com um comando de montagem no
script de inicialização de inicialização:
      mount -t cifs -o guest, file_mode = 0666, dir_mode = 0777 // localhost /
share_name / mnt / samba
   a lista de opções separadas [-o] reflete as necessidades de configuração do servidor
Samba)

 ÚNICO servidor: você pode usar um único servidor LetoDBf normalmente e apontar para
letodb.ini
 DataPath = / mnt / samba, defina: Share_Tables = 1 e adicione a opção especial:
SMB_SERVER = 1.
 * Problema: o desempenho do servidor LetoDBf funcionando em uma unidade de rede é *
drástico * mais lento
 como trabalhando em um disco rígido real.

 Servidor DOUBLE: então a seguinte estratégia é usada:
 As tabelas <exclusivas> são abertas no servidor LetoDBf trabalhando na unidade de rede
As tabelas <shared> são abertas em ambos! servidor, onde todo o trabalho ativo acontece
no disco rígido.
 A tabela aberta no drive de rede é para informar o usuário do Samba e evitar que ele
abra <exclusivo>.
 * Isso tudo acontecerá de forma totalmente automática para o aplicativo cliente LETO
RDD, não há necessidade de se preocupar com *,
 mas a configuração é muito mais complexa:
   # crie uma cópia de letodb.ini (letosmb.ini) e adapte-a:
   - use uma porta diferente, por exemplo: Porta = 2814
   - DataPath para apontar para o ponto de montagem CIFS [por exemplo, / mnt / samba] ou
um subdiretório dele
   - LogPath pode ser o mesmo, mas defina: Debug = 0 para obter apenas erros
   - adicionar entrada: SMB_SERVER = 1
      isso usará a verificação 'smbstatus -L' para garantir a simultaneidade para a
tabela <exclusive>
   - Para diminuir os atrasos, possivelmente causados pela chamada 'smbstatus -L',
adicione também:
      SMB_PATH = / absoluto / caminho / de / compartilhar: / mnt / samba
      O lado esquerdo é o caminho absoluto do <share> no servidor, o lado direito do
<mount-point> de
      acima do comando de montagem ou da entrada / etc / fstab - sem espaços em branco
ao redor do delimitador ':'.
      * Construa * o executável 'elsof' em 'tests / c_lang / elsof.c' -> veja no topo
desse arquivo como fazer.
      Sem esta opção, 'smbstatus' é usado, e espera-se que ambos os executáveis estejam
em
      '/ usr / bin'. Em caso de problemas para executar, você receberá uma mensagem de
erro em letodbf.log.
   - * ambos os arquivos * .ini precisam da opção: Share_Tables = 1
   # iniciar ambos os servidores com a opção de especificar um nome de arquivo ini,
literalmente no exemplo:
      letodb config letodb
      letodb config letosmb
      (o desligamento é semelhante, por exemplo: letodb stop letodb; letodb stop letosmb)

   * Adaptação da biblioteca RDD
      # comentário externo (remova o '#') para LETO_SMBSERVER = 1 nos arquivos de make
hbmk2:
        rddleto [addon] .hbp
        Recrie a biblioteca RDD, para obter um comportamento diferente para trabalhar em
dois servidores
      # em seu aplicativo, você precisa se conectar a * ambos * o servidor com
Leto_Connect (),
```

```
        Conecte-se ao que tem letosmb.ini * antes * da conexão padrão comum,
        [para evitar uma necessidade posterior de voltar]
      # no diretório do seu aplicativo é necessário um arquivo de configuração
'rddleto.ini':
        copie tests / rddleto.txt como amostra para o local e defina nele:
        SMB_SERVER = IP | nome DNS
        SMB_PORT = 2814
        [a porta que você escolheu em letosmb.ini e o endereço ou nome de seus
servidores]


   Adaptação do aplicativo SAMBA,
     # build: hbmk2 tests / excltest.prg
       e copie o executável, por exemplo, no diretório raiz do <share>.
     # copie tests / rddleto.txt como excltest.ini no mesmo diretório;
       isto será específico para aquele exe como com o mesmo nome de arquivo
     # adaptar em excltest.ini:
       SMB_SERVER = its.IP.address (para poupar tempo de resolução de DNS para cada
chamada)
       SMB_PORT = 2814 (para configuração de servidor único, a porta padrão é 2812)
   Este executável deve ser chamado após qualquer tabela <shared> aberta com sucesso.
   Ele define o nível de erro do sistema operacional como resultado para 1, se a tabela
já estiver <exclusiva> usada,
   então, apenas <shared> aberto era inválido e a tabela deve ser fechada imediatamente.
   Veja no início em tests / excltest.prg para mais instruções sobre como usar.


      4.5 Segurança

   Por padrão, o LetoDBf escuta todas as interfaces com a opção de configuração vazia: IP
   Caso o servidor também possua interface conectada à internet, recomenda-se
   limite isso à interface (fornecida pelo endereço IP ou nome do dispositivo Unix) com
a intranet.
   Isso também excluirá o dispositivo de loopback local (lo).
   Se não quiser, se houver apenas uma interface para internet e intranet, ou se houver
vários
   interfaces para a intranet para balanceamento de carga, deixe a opção IP vazia e
defina
   opção: IP_SPACE para limitar os espaços de endereço IP permitidos.
   Esta string contém partes do lado esquerdo do intervalo de endereços IP permitido,
delimitado com ';'.

   Para estabelecer uma nova conexão com o servidor, uma senha codificada é usada.
   'Codificado' significa que está gravado no executável do servidor e na biblioteca do
cliente.
   Essas senhas devem ser as mesmas para o servidor e a biblioteca RDD, caso contrário,
a conexão falhará.
   Usando outra string para 'LETO_PASSWORD' em 'include / funcleto.h', você pode
personalizar
   seu servidor apenas para ser conectado por aplicativos vinculados à sua biblioteca
cliente.
   Para alterar o valor de LETO_PASSWORD * sem * alterar o código-fonte, use a chave
para definir o sinalizador:
      hbmk2 [letodb | rddleto] -cflag = -DLETO_MYPASSWORD = MyOwnSecurePassword
   onde <MyOwnSecurePassword> não pode conter ',' e evite aspas simples e duplas.
   [obter um erro de compilação indica que a string da senha é inválida]
   Como essas senhas são gravadas como "texto simples", é bom que não consistam em
palavras reais,
   dê uma olhada no padrão.

   O servidor LetoDbf pode ser configurado para aceitar somente conexões com nome de
usuário / senha.
   Veja aqui: 4.3 Autenticação

   Além disso, o LetoDBf oferece tráfego de rede criptografado blowfish no modo CBC.
   Isso é ativado sob demanda em conjunto com a compressão de rede, usando o cPassword>
   parâmetro em Leto_Togglezip (nLevel, cPassword).
```

A compactação (mais criptografia) pode ser ativada imediatamente após o estabelecimento de uma conexão.
NOVO: com a opção de servidor "CRYPT_TRAFFIC", a criptografia de tráfego de rede é exigida para ser usada por
cliente, é como acima Leto_Togglezip () desde o início usando uma senha aleatória.
Isso bloqueará qualquer conexão que não esteja usando criptografia.

5. Como trabalhar com o servidor LetoDBf

5.1 Conectando ao servidor

```
# .. sem mudança de fonte - NOVO [não xHarbour]
```

Ao fornecer um "rddleto.ini" (arquivo de amostra "rddleto.txt" no diretório "testes") com o IP do servidor | nome DNS,
[Servidor = ... nome simples ou com pré-liderança "//" e porta de servidor opcional ':'],
-> sua aplicação na inicialização, durante a inicialização do "LETO" RDD, tenta se conectar ao servidor,
e! sai! se isso falhou.
Use 'DETECT' em vez do IP do servidor | DNS para transmitir para o servidor com um Leto_Detect () no início do aplicativo,
que foram configurados em letodb.ini [BC_Service] para responder a essas solicitações.
Isso significa que nenhum trabalho de configuração deve ser feito, o servidor deve estar ativo e os clientes devem encontrá-lo.

```
# .. com a adição de algumas linhas de origem
```

A maneira 'tradicional' comum é adicionar uma sequência curta sobre "leto_Connect ()" à fonte existente,
exemplo, coloque-o na função main (). Dê uma olhada em 'tests / basic.prg'

```
IF leto_Connect ("//192.168.7.42:2812/") <0
   Alerta ("Não é possível conectar ao servidor ...")
   SAIR
FIM SE
```

Isso abre possibilidades como várias solicitações de conexão, feedback detalhado sobre falhas,
ou se seu aplicativo precisa se conectar a vários! Servidor LetoDBf, verifique os parâmetros detalhados de
leto_Connect () em: 7.1

Se você precisar fazer alterações de fonte específicas para LetoDBf, para manter sua fonte portátil para uso
sem LetoDBf, você pode querer encapsulá-los da mesma forma:

```
#ifdef RDDLETO_CH_
   ... especial para Leto
#fim se
```

5.1.2 construir seu aplicativo

Espero que você não tenha definido o segundo parâmetro de: DbUseArea (, cDriver, ...)
ou use a opção "VIA" do "comando USE": em seguida, remova cDriver para todos os WA a serem usados com LetoDBf.
Isso tornará seu aplicativo portátil também para outro RDD como LETO, já que o RDD padrão pode ser fácil
determinado no início do programa definindo: RDDSetDefault ("XXXX"), por exemplo, "DBFCDX".
Um login bem-sucedido no servidor define o RDD padrão como "LETO".

Mais fácil e altamente recomendado é construir seu aplicativo para LetoDBf usando "letodb.hbc" para hbmk2.
Você pode incluí-lo em seu projeto HBP adicionando-o em uma linha extra ou usá-lo na linha de comando:

```
      hbmk2 yourapp [.prg | .hbp] letodb.hbc

 Ele automatiza para você as seguintes etapas:
 a) adicionar a biblioteca LetoDbf e incluir caminhos para o processo de vinculação
 b) solicitação para vincular o driver LetoDBf RDD, caso contrário, deve ser definido
manualmente fora do procedimento principal ()
      SOLICITAR LETO
 c) inclui o arquivo de cabeçalho: "rddleto.ch", caso contrário, deve ser feito
manualmente em cada um de seus PRG:
      #include "rddleto.ch"
      ou
      isso pode ser feito usando a opção <-u +> para Harbour: -u + rddleto.ch
 d) o mesmo que para c) inclui o arquivo de cabeçalho: "leto_std.ch"
 e) defina a opção "-mt = yes" para vincular seu aplicativo à biblioteca de cliente
LetoDBf com suporte multi-thread.
    Além disso, uma idletask aditiva interna é ativada, se seu aplicativo for compilado
com '-mt'.
 e no final vincula as funções lib do cliente "rddleto" chamadas em seu código-fonte
para o executável.


       5.1.3 conectado com sucesso ao servidor

 Isso definirá o driver RDD padrão para "LETO" após a conexão semelhante feita com:
RddSetDefault ("LETO").
 Além disso, o servidor é informado sobre quatro configurações de SET: DELETED /
SOFTSEEK / AUTOPEN / AUTORDER.
 Com a conexão ao servidor, e posteriormente com a abertura ou criação de uma tabela,
informações sobre a página de código
e as configurações de formato de data são enviadas ao servidor. Isso é importante para
criar pedidos de índice contendo
 caracteres especiais nacionais ou chaves de índice contendo um valor de data.
 A última configuração de formato de data aplicada será a partir de então a configuração
ativa no servidor.

 Todos os nomes de arquivos e caminhos agora são relativos ao DataPath raiz em
letodb.ini.
 Pode ser parecido: (a este exemplo me refiro nas seguintes explicações)
      DataPath = [drive:] \ path \ to \ data_diretory

 Se nenhum DataPath for fornecido (! NÃO! Recomendado), será o diretório raiz com o
servidor
 executável.

 Exemplo: DbUseArea (.T. ,, "test \ customer.dbf") abrirá DBF em:
      [unidade:] \ caminho \ para \ data_diretory \ test \ cliente.dbf.

 Uma letra de unidade em seus nomes de arquivo será cortada e apenas um diretório ".. \"
acima do
o <DataPath> em letodb.ini é permitido. Isso significa: DbUseArea (.T. ,, ".. \
data_other \ customer.dbf")
 irá apontar para:
      [unidade:] \ caminho \ para \ dados_outros \ cliente.dbf.

 Se os seus nomes de arquivo contiverem pelo menos um separador de caminho '/' ou '\',
eles serão tratados como relativos a
<DataPath>. Todos os separadores de caminho em seus nomes de arquivo são convertidos
pelo servidor LetoDBf interno para o necessário
 um, não há necessidade de se preocupar porque "\" ou "/" é igual.

 Este caminho raiz é igual a uma configuração SET DEFAULT TO, então se seus nomes de
arquivo não contiverem * nenhum separador de caminho,
 todos os novos arquivos serão criados diretamente em <DataPath>.
 Os nomes de arquivo podem ter um líder dependente do sistema operacional '\' ou '/',
por exemplo: /mydbf.dbf, que forçará
 eles no diretório <DatPath>, apesar de um determinado "SET DEFAULT" para um
subdiretório.
```

Para um nome de arquivo em branco puro, a configuração "SET DEFAULT TO ...", quando definida * antes de * Leto_Connect () nomeará
um subdiretório aditivo para <DataPath>. Exemplo: "SET DEFAULT TO data" colocará todos os arquivos NOVOS sem
separador de caminho em:
[unidade:] \ caminho \ para \ data_diretory \ data
!! Permitido para "DEFAULT" é apenas um único caminho, NOT ENDING WITH ';' OU ":" !!
A configuração de exemplo de "SET PATH TO system; tmp" levará à busca de arquivos em:
[drive:] \ path \ to \ data_diretory \ system e [drive:] \ path \ to \ data_diretory \ tmp
Se você usar: DbUseArea (,, cFile, ..) e não estiver no diretório DEFAULT, mas em um diretório fornecido
por SET PATH, ele será aberto sem que você precise fazer mais nada.
Ambos DEFAULT e PATH reconhecem o <DataPath>, o que significa que um "SET DEFAULT TO to \ data_diretory \ data" irá
leva a um diretório DEFAULT: '[unidade:] \ path \ to \ data_diretory \ data' conforme fornecido como "SET DEFAULT TO data".
!! Novamente, ambas as configurações de "DEFAULT" e "PATH" se aplicam a nomes de arquivos sem nenhum separador de caminho '\' ou '/'.

Para verificar alguns primeiros exemplos, olhe para o diretório "testes". Construa todos de uma vez com: "buildall"
resp. "- / buildall.sh" para Linux. Adicione para execução como primeiro parâmetro o endereço IP do servidor,
se desconhecido, mas na mesma máquina em execução que o aplicativo, use a rede de loopback: 127.0.0.1
Em qualquer caso, você deve tentar o executável "test_file", já que existem funções "Leto_File () que não funcionam
múltiplo! quilômetros de discussão mística no grupo Harbour Google sobre sua magia interior :-) 8-) !!

A outra maneira, não recomendada por não ser portátil, é abrir uma tabela DBF com o
método 'em tempo real' adicionando o endereço IP [: porta] mais o caminho relativo para o comando 'USE' do Harbours,
exemplo:
USE "//192.168.5.22:2812/mydir/test"
irá abrir o arquivo em:
[unidade:] \ caminho \ para \ data_diretory \ mydir \ test.dbf.
Depois de fazer isso pela primeira vez, você também está conectado ao servidor e precisa para as próximas chamadas
no more IPaddress: port prefix more. Esta é uma dica para os usuários experientes do LetoDB pensarem.

5.2 Filtros e Relações

5.2.1 Filtros

O filtro é estabelecido geralmente: pelo comando SET FILTER TO ou chamando a função DbSetFilter ().
O parâmetro mais importante de DbSetFilter () é a expressão, o segundo parâmetro: DbSetFilter (bBlock, cExpression)
pois apenas isso pode ser transmitido ao servidor. (os codeblocks não podem ser trocados entre os aplicativos)
A expressão de filtro que pode ser executada no servidor é chamada de "otimizada".
Caso o filtro deva ser executado localmente no cliente, é denominado: "não otimizado".
Esse filtro é lento como todos
os registros devem ser recebidos do servidor e o cliente deve descartar todos os registros inválidos.
No caso de um filtro otimizado, apenas os registros válidos são enviados para o aplicativo cliente.

Para obter um filtro otimizado, é necessário que a expressão seja executável exclusivamente para o servidor.
Portanto, todas as funções nele contidas, como Str (), Upper (), DToS (), etc, devem ser conhecidas pelo servidor.

Se sua expressão contém uma função própria, ela pode ser carregada com um arquivo HRB a qualquer momento durante
um servidor em execução, consulte, portanto, 4.2.1 UDF supprt.
Se sua expressão contém uma variável conhecida apenas por seu aplicativo, o poderoso sistema Leto_Var * () entra em
açao. Com isso, você pode compartilhar o conteúdo de uma variável do lado do cliente com o servidor e vice-versa.

Assim, com a ajuda das funções leto_Var * () mais funções carregáveis UDF mais um rico conjunto de clássicos Harbour
comandos, é possível transformar qualquer filtro não otimizado em otimizado.
Eles são rápidos como um relâmpago e é um prazer trabalhar com eles. Para testar se um filtro é otimizado, basta verificar
para com a função LETO_ISFLTOPTIM (), se retornar TRUE.

As duas configurações a seguir influenciam o uso de filtros otimizados do lado do servidor:

SET (_SET_OPTIMIZE, .F.) Irá desabilitar qualquer avaliação de filtro no servidor, então cada filtro se tornará um
filtro não otimizado a ser executado pelo cliente [o padrão é .T. == permitir].
Apenas no modo de servidor: No_save_WA = 1 e depois configuração:

SET (_SET_FORCEOPT, .T.) Irá habilitar expressões de filtro no servidor com nomes ALIAS,
talvez de um relacionamento
área de trabalho. [o padrão é .F. == ALIAS não permitido].
O modo de servidor No_Save_WA = 1 é necessário com nomes ALIAS diferentes do WA ativo, porque no modo '0' LetoDBf
servidor usa técnicas de desanexação / solicitação WA e outras WA não estão disponíveis / desanexadas e, portanto, não há
relação ativa no servidor.
Com ForceOpt = TRUE, uma expressão de filtro DEVE ser válida no servidor, caso contrário, um RTE é lançado - NÃO será
avaliado alternativamente no cliente como sem ForceOptimize.
[NOVO] Se a expressão do filtro contiver um memvar PRIVADO / PÚBLICO, eles serão sincronizados entre o cliente e o servidor
com a ajuda do sistema LetoVar (). Um Leto_VarCreate () único com o valor do memvar é executado, o filtro
string expression é automaticamente modificada para usar um Leto_VarGet () em vez de memvar.
A cada movimento (GoTo [p | bottom], Skip, Seek) no WA, um possível valor memvar alterado é sincronizado automaticamente
com uma nova chamada Leto_VarSet (). Limpando o filtro, também feito com o fechamento da tabela, Leto_VarDel () todos
criou LetoVars.
Exemplo: 'SET FILTER TO table-> field> xMemvar' funcionará como sem LeotDBf, mas como filtro otimizado.
Veja também as funções Leto_VarExpr * () se quiser fazê-lo manualmente ou para outras ocasiões.

A otimização de desempenho para pode ser feita adaptando as configurações padrão para tamanho / tempo limite do buffer de salto:
Veja o capítulo 7.3 para DBI_BUFREFRESHTIME, DBI_AUTOREFRESH e 7.5 para LETO_SETSKIPBUFFER ()

5.2.2 Relações

No modo de servidor: No_Save_WA = 1 relações são ativas adicionais no lado do servidor. Isso não é possível para o servidor
mode: No_Save_WA = 0, aqui as relações são ativas apenas no lado do cliente.
Para transmitir a expressão de relação ao servidor, o segundo parâmetro de DbSetRelation (bBlock, cExpression) é
necessário, já que apenas uma string pode ser transferida para o servidor, sem bloco de código.
Usando o comando: "SET RELATION TO ... INTO ..." este segundo parâmetro é preenchido automaticamente através do

arquivo de cabeçalho: <std.ch>, que é incluído a qualquer momento para qualquer aplicativo Harbour. Este segundo parâmetro deve ser definido manualmente se a função DbSetFilter () for usada. Tenha cuidado para que bBlock e cExpression tenham o mesmo significado.
Se cExpression contiver um erro ou não for executável no lado do servidor devido a uma função conhecida apenas em
seu aplicativo (consulte 4.2.1 Suporte a UDF) ou variáveis de aplicativo cliente (consulte 7.9 Variáveis de servidor), você
obter um RTE com uma descrição do que falhou no servidor.
Definir uma 'relação cíclica', também conhecida como um conjunto de relações onde uma área filho relacionado se refere a uma área pai,
levar a um RTE informando você sobre. A 'relação cíclica' mais simples é uma relação que aponta para si mesma como uma área de trabalho.

5.3 Driver de banco de dados

Se em nenhum lugar for definido explicitamente, o driver de banco de dados padrão para o servidor é DBFCDX.
Este driver padrão no servidor pode ser alterado no letodb.ini com a configuração de Default_Driver.
O driver ativo para sua conexão você pode consultar e! SET! com:

```
leto_DbDriver ([<cNewDriver>], [<cMemoType>], [<nBlocksize>])
                                                    ==> aInfo
```

Isso retornará uma matriz tridimensional, na ordem dos parâmetros, então aInfo [1] é o driver usado ativo. Sem argumentos, as configurações ativas são retornadas.
Você pode alterar esse padrão usando "DBFNTX", "DBFCDX", "DBFFPT", "DBFNSX" ou "SIXDBF".

Cada driver pode deixar de lado seus padrões combinados com MemoType "DBT", "FPT" ou "SMT".
Mais para especialista! usuários, o tamanho do bloco usado para o memofield pode ser alterado:
o mínimo é de 32 bytes, o máximo de 64 KB == 65535 bytes e sempre deve ser um múltiplo de 32 bytes.
! Isso deve ser feito antes de criar um novo DBF, caso contrário, a configuração ativa com a criação é usada para
esse DBF.

!! Use a função leto_DbDriver () antes de abrir ou criar novos arquivos !!
Desta forma, você pode até mesmo misturar diferentes drivers para uma única conexão. Claro, você não pode misturar diferentes
drivers para o mesmo banco de dados, portanto, por exemplo, uma tabela DBFNTX deve ser usada para todas as conexões do tipo NTX.

5.4 Arquivos de dados especiais em RAM

O LetoDB pode criar (e compartilhar entre as conexões) tabelas de dados na RAM, não no disco rígido.
Essas tabelas chamadas HbMemIO são especialmente úteis para tabelas temporárias. Eles são limitados a
RAM disponível em seu servidor e válido apenas durante a execução de um servidor, tudo perdido após o desligamento do LetoDB.

Os nomes de arquivo para essas tabelas de dados especiais opcionais podem ter * opcional * um separador de caminho inicial
['/' ou '\'], seguido obrigatoriamente por um prefixo 'mem:'. Por exemplo:
"/mem:speeddata.dbf" será um nome de arquivo válido. Essas tabelas de dados funcionam como tabelas de dados 'reais',
portanto, eles são criados usando o driver de banco de dados ativo (consulte 5.3).
Quando os pedidos de índice são criados, o bagname também deve ter este prefixo: "mem:",
caso contrário, será
criado no disco rígido.
Você também pode combinar uma tabela de dados no disco rígido com um índice (temporário) na RAM.

Ou uma tabela de dados na RAM com índice no disco rígido, mas esta combinação não faz muito sentido ;-)
Esses arquivos HbMemIO têm, especialmente em discos rígidos mais lentos, algumas vantagens de desempenho.

!! Não se esqueça de descartar / excluir nenhum arquivo hbMemIO usado, caso contrário, o servidor pode esgotar rapidamente
de RAM disponível. !!

6. Gestão de variáveis

Letodb permite gerenciar variáveis, que são compartilhadas entre aplicativos, usuários conectados
para o servidor e com o próprio servidor. Todas as operações em variáveis são realizadas consecutivamente por
um thread, então as variáveis podem funcionar como semáforos.

Role para a seção: 7.9 Funções de variáveis de servidor e verifique um primeiro exemplo em tests / test_var.prg.

7. Lista de funções

7.1 Funções de gerenciamento de conexão

Abaixo está uma lista completa (pelo menos no momento em que escrevo) de funções,
disponível para uso em aplicativos cliente com RDD LETO vinculado.

Nessas funções, <cAddress> significa o endereço IP no formato: "// IP: porta /".
A parte ": port" agora é opcional, se não for fornecida, usará ": 2812" como padrão.

```
LETO_CONNECT ([cAddress], [cUserName], [cPassword],
              [nTimeOut], [nBufRefreshTime], [lZombieCheck])
                                                    ==> nConnection, -1 se falhou
```

Sem qualquer parâmetro fornecido, o ID de <nConnection> no servidor da conexão ativa é retornado,
caso contrário, o ID de conexão no cliente onde '0' == primeira conexão de um thread de aplicativo.
<cAddress> pode ser fornecido como endereço IP bruto "127.0.0.1" ou "//127.0.0.1/".
Se a porta for omitida (também conhecida como "//127.0.0.1:2812/"), o número da porta padrão: 2812 é escolhido.
Como alternativa, os nomes DNS podem ser usados em vez de um número IP, por exemplo: 'localhost'.
<nTimeOut> define como o log para uma resposta do aplicativo do servidor irá esperar, em 0,001 segundos.
Este valor de tempo limite é válido para cada solicitação ao servidor, não apenas para a conexão inicial.
O padrão é 120000 aka 2 minutos. '-1' significa espera infinita. Após esse período, o aplicativo irá
interromper com um erro se nenhuma resposta do servidor tiver sido enviada.
<nBufRefreshTime> define o intervalo de tempo em unidades de 0,01 segundos. Depois que esse tempo acabar,
o skipbuffer de registros será atualizado, 100 por padrão (100/100 == 1 seg).
Valor zero (0) significa infinito! caching, -1 irá desabilitar usando o buffer de salto. Esses dois extremos
os valores devem ser aplicados apenas em muito especial! ocasião e necessidade.
<lZombieCheck> = .F. desabilite o uso de um segundo soquete para comunicação mais rápida com o servidor,
e como a verificação de conexões inativas precisa disso, isso também o desabilitará para esta conexão.
O padrão para Harbour é .T., Para xHarbour em qualquer caso .F. - isso geralmente não deve ser alterado.

```
      LETO_CONNECT_ERR ([lAsText]) ==> nError [cError]

 Recupera o último erro ocorrido para a conexão ativa, não apenas após a conexão.
 Com <lAsText> TRUE dado um string com uma descrição.

      LETO_DISCONNECT ([cAddress]) ==> lDesconectar
 Desconectar a conexão atual, retorna booleano se uma conexão for desconectada.
 Com o parâmetro opcional <cAddress>, essa conexão é tentada se desconectar.

      LETO_DETECT ([cService], [nNrOfPossible], [nPort]) ==> cServerIP
 Esta função envia um broadcast para a rede local, para todas as interfaces disponíveis,
 (virtual) loopback de interfaces (lo) e outros sem endereço MAC são excluídos disso.
 O <cService> padrão é "letodb", e isso está correlacionado ao lado do servidor:
 veja no exemplo letodb.ini a opção de configuração: BC_Services = letodb;
 Esse servidor LetoDBf configurado então responderá a esta consulta com seu IP / Porta,
 onde <cServerIP> está no formato: "//9.9.9.9:2812/" para ser usado, ou seja, para
conectar
 para o servidor: Leto_Connect (Leto_Detect ()),
 Com <nNrOfPossible> pode ser selecionado entre vários servidores respondendo ao mesmo
nome de serviço.
A alternativa para o novo build-in no servidor era antes de um exe autônomo: 8.2. 'Uhura'

      LETO_SETCURRENTCONNECTION ([cAddress [, lBefore]]) ==> cAddress
 Retorna o endereço da nova conexão ativa <cAddress> após uma tentativa de alterar a
ativa.
 <cAddress> deve ser fornecido no formato usual, sem ou com o número da porta:
"//127.0.0.1:2812/".
 A string vazia "" é retornada se a conexão deseja-ativar não for encontrada ou se houver
 nenhuma conexão ativa antes.
 NOVO: o parâmetro opcional <lBefore> dado como verdadeiro (.T.) Retornará o endereço da
conexão ativa
 antes que uma mudança bem-sucedida fosse feita.
 Observe que é impossível mudar para <não> conexão enquanto pelo menos uma estiver
disponível.

      LETO_GETCURRENTCONNECTION () ==> cAddress
 Obsoleto, existindo posteriormente como função traduzida - Leto_SetCurrentConnection ()
sem qualquer parâmetro
 ou uma string vazia também retorna a conexão ativa.
 Esta função foi usada junto com Leto_SetCurrentConnection () para salvar / restaurar a
conexão ativa.

      LETO_GETSERVERVERSION ([lHarbourVersion]) ==> cVersion
Retorna a versão do servidor LetoDBf, com .T fornecido. parâmetro booleano a versão do
Harbour em
 tempo de compilação.

      LETO_GETLOCALIP ([lLocal]) ==> endereço IP do cliente [servidor]
 Retorna o endereço IP da primeira interface encontrada para a sub-rede, que pode ser a
errada,
 se você tiver várias NICs (como pontes físicas ou lógicas) para essa sub-rede.
 Com opcional <lLocal> == FALSE (.F.) Retorna o endereço IP do servidor conectado.

      LETO_ADDCDPTRANSLATE (cClientCdp, cServerCdp) ==> lAdded
 Para o usuário xHarbour com nomes de CP diferentes como os do Harbour no servidor.
 Depois de definido, ele não pode ser removido até o final do aplicativo cliente.
 Nenhuma verificação feita se o CP está disponível no cliente, nem no servidor.

      LETO_SET (nOption [, xNewSet]) ==> xActiveSetting
 Recomenda-se usar os comandos: SET xxx [TO] ou a função SET () ao invés, para manter seu
código fonte portátil. Uma tradução é feita em "rddleto.ch" por definir macros, para
informar o servidor sobre
mudanças de quatro configurações: SOFTSEEK, DELETED, AUTORDER, AUTOPEN. Todas as outras
solicitações / configurações são
 simplesmente encaminhado para a função SET ().
 Sem o <xNewSet> fornecido, você obtém a configuração ativa sem alterá-la.
```

```
      * LETO_SETLOCKTIMEOUT ([nTimeout]) *
 * obsoleto - é melhor usar RddInfo com uma constante conhecida no Harbour:
      RDDInfo (RDDI_LOCKRETRY ([nMilliSec])
 * A configuração é ignorada para o modo de servidor: No_Save_WA = 0, aqui sempre
padrão: '0' == uma tentativa *
 Retorne o antigo valor <nTimeout> e defina-o opcionalmente para a conexão ativa.
 O padrão é 0 == solicitação única imediata, as unidades estão em ms (1/1000 s).
 Um valor de tempo limite é aplicado no servidor como várias solicitações de bloqueio,
enquanto o intervalo de tempo
não é alcançado em caso de falha. Valores entre 50 e x000 devem ser úteis.

      LETO_RECONNECT ([cAddress], [cUserName], [cPassword],
                      [nTimeOut], [nBufRefreshTime], [lZombieCheck],
                      [nDelay], [nLockTimeout])
                                                          ==> nConnection, -1 se
falhou
Todos os parâmetros são opcionais! e o mesmo que LETO_CONNECT (), mais o 7° parâmetro
opcional <nDelay> em segundos da unidade.
 Isso fechará uma possível conexão ainda viva ou morta com o servidor e restabelecerá
uma nova conexão
para o mesmo servidor ou até mesmo para um servidor diferente se <cAddress> for
fornecido. Exceto <cUserName> e <cPassword> todos
 os parâmetros são preenchidos pela configuração da conexão antiga.
 O ambiente WA completo (tabelas, ordens de índice, filtros, escopo, relações, bloqueios
R | F) é restaurado
 para a nova conexão.
 Se for o mesmo servidor, <nDelay> será de 1,0 segundo para permitir que o servidor
feche as tabelas e remova
 bloqueios existentes antes de tentar estabelecer uma nova conexão, para um servidor
diferente, nenhum atraso é necessário.
 <nDelay> só deve precisar ser definido manualmente, se uma conexão for feita com o
mesmo servidor, mas mais
rede diferente. No caso de um erro LETO da conexão antiga, um atraso extra é aplicado
antes
 o soquete da rede está fechado (padrão: 1 s).
 <nLockTimeout> em ms pode ser usado como valor de tempo limite para restabelecer
bloqueios existentes anteriores,
 onde o valor padrão é 0 == * não * tempo limite extra para obter um bloqueio, uma
tentativa imediata uma vez.

      LETO_SETCONNECTLOOKUP ([lSet]) ==> lOld, configuração antes
 Isso ativa a pesquisa / pesquisa em outras conexões disponíveis para a tabela DBF
existente,
 se a tabela não for encontrada na conexão ativa.
 A configuração é específica para uma conexão, portanto, a conexão ativa deve ter
definido este sinalizador.
O padrão é falso [.F. ] para uma nova conexão.
 Se a tabela não for encontrada em nenhum lugar, espere feedback como de costume - já
que nenhuma pesquisa acontecerá no caso de um
 Erro de abertura compartilhada.

      LETO_SETPATH (cPath [, lDefault]) ==> cOldPath
 Ele define "SET PATH TO (cPath)" no servidor LetoDBf.
 Este (s) caminho (s) são relativos a "DataPath" na configuração 'letodb.ini' e são
pesquisados por tabelas DBF
 quando nomes de arquivo de tabela * simples * sem um componente de caminho são usados
para DbUseArea () / DbSetIndex ().
 Com o <lDefault> fornecido opcionalmente como verdadeiro (.T.), "SET DEFAULT TO
(cPath)" é definido no servidor,
 então, ele se tornará um subdiretório de "DatPath" onde * novas * tabelas são criadas.
 Também aqui para o nome de arquivo deve ser fornecido na forma * simples * - quando
contém um caminho, os nomes de arquivo são sempre
 tratado como relativo a 'DataPath'.
 Definir o caminho DEFAULT também define 'DataPath' de 'letodb.ini' como search PATH, se
ainda não estiver definido.
 Definir um PATH de pesquisa também adiciona o próprio 'DataPath' como último PATH de
pesquisa adicional.
```

<cOldPath> será apenas um único ";" se nenhuma conexão ativa ou então ocorreu um erro, caso contrário, é
o (s) caminho (s) relativo (s) anteriormente ativo (s) - útil para mudança temporária e redefinir para antes.
Observe que nenhuma verificação é feita se os caminhos já existem, então caminhos não existentes
levar a arquivos não encontrados ou até mesmo a um erro de criação.

7.2 Funções de transação

Uma transação é uma série de alterações de dados, cuja aplicação é garantida em uma sequência semelhante
'um bloco'. Na vida prática, apenas bloqueios de registro / arquivo são definidos no servidor (sem bloqueio DbAppend ()),
e todas as alterações de dados são armazenadas em buffer no lado do cliente. Se você DbSeek () / DbSkip () para um registro com alterado
dados, você verá no cliente suas alterações ainda não aplicadas no lado do servidor.
Ao confirmar a transação, uma única solicitação sobre todas as alterações é enviada ao servidor, que irá
comece a processar as alterações de dados primeiro após concluir o recebimento da solicitação e outras pré-verificações.
Até este ponto, é garantido que * tudo ou nada * de uma transação seja processada.
Se o servidor tiver problemas de hardware durante o processamento da sequência, como perda de energia, peças
de uma transação vai perder ...

Como efeito colateral, as transações são boas para Flock () ed ou não compartilhadas, tabelas abertas exclusivas, porque
para estes, leva a um! drástico! desempenho aprimorado para anexar 10.000 registros em uma fração de segundo.
Para RLock () uma sobrecarga tremenda insana é necessária, se realmente ** muitos ** registros precisam ser
mudado. Exemplo: para Rlock () o 1000° registro no servidor, ele deve pesquisar entre 999 bloqueios existentes.
Em suma, uma comparação foi feita 499.500 vezes para o milésimo registro.
Apenas 1000 mudanças de registro Rlock () do exemplo acima são feitas imediatamente, mas se houver 10
de milhares ...

!! Importante !! - durante uma transação ativa, também conhecida como após leto_BeginTransaction ():
NOVO: if <lUnlockAll> == .T. todos os * un * -locks serão ignorados em silêncio até o final da transação.
é expressamente proibido misturar RLock () e FLock () PARA A MESMA ÁREA DE TRABALHO
você não pode desbloquear nenhum registro ou tabela inteira (por exemplo, NÃO use DbUnlock ())
transações devem começar e terminar com uma área de trabalho LETO RDD ativa, e isso deve ser mais
do mesmo servidor LetoDBf.
você pode modificar o mesmo campo do mesmo registro várias vezes, e entre mudar para diferentes
registro, mas obterá como valor sempre a primeira modificação. Em outras palavras: você não pode somar uma soma.
Mais tarde, no servidor, essas mudanças são aplicadas como mudanças consecutivas, então a última mudança faz
o resultado.
Recno () será '0' após 'anexar' um registro em branco com, por exemplo, DbAppend ()
conforme o servidor faz o trabalho para campos de incremento automático, eles estarão vazios no lado do cliente por um
DbAppend () transacionado. Deixe-os inalterados, o servidor os preencherá mais tarde durante o acréscimo real.
a DbSeek () não encontrará novos registros anexados, isso também é verdadeiro para relações

LETO_BEGINTRANSACTION ([[lUnlockAll])

Por padrão, todos os bloqueios existentes (bloqueios R e bloqueios F) permanecem com o início da transação.
Com o parâmetro <lUnlockAll> dado como verdadeiro (.T.), Todos os bloqueios existentes são dissolvidos,
e é muito conveniente fazer isso para todas as áreas de trabalho LETO de uma vez, o que dará
você um novo começo (também conhecido como não deve pensar em continuar para esta ou aquela área de trabalho com Rlock ()
ou FLock ().
NOVO: ao desbloquear tudo com .T., Também seguindo * un * -locks são ignorados até o final da transação.
Isso é pensado para manter as alterações no código-fonte o menos possível, ou seja, basta adicionar duas linhas
com Begin / Commit -transaction para a fonte existente, sem a necessidade de remover desbloqueios existentes.

Nota: com RDDInfo (RDDI_LOCKRETRY [, nMilliSec (padrão 0)]) um tempo limite para obter um bloqueio pode ser definido.
Isso é muito eficaz contra tentativas repetidas de obter um bloqueio do lado do cliente,
como tudo acontece em
lado do servidor, nenhuma nova solicitação para tentar novamente o bloqueio deve ser enviada.

LETO_ROLLBACK ([lUnlockAll]) ==> nulo

Isso descartará as alterações de campo coletadas e os registros a serem anexados.
Por padrão, <lUnlockAll> é verdadeiro (.T.), Então todos os bloqueios (bloqueios R e bloqueios F) são removidos,

LETO_COMMITTRANSACTION ([lUnlockAll]) ==> l Êxito

padrão <lUnlockAll> == .T. irá desbloquear automaticamente * todos * os bloqueios (Rlock () e Flocks ()) após
a transação é processada no servidor.
Padrão (.T.) Recomendado para fazer, pois é mais rápido. Além disso, o cliente não terá nenhuma informação
sobre quais registros são anexados e R-bloqueados no servidor, ele deve consultar isso novamente.

! ATENÇÃO !
Se você deseja também * anexar * registros em uma transação, você precisa de uma mesa aberta compartilhada em
pelo menos Flock () * ou * um único Rlock () feito durante a transação para esta área de trabalho.
Se nenhum estiver bloqueado, por exemplo, uma transação contendo apenas registros para anexar,
O cliente LetoDBf tentará internamente definir um Flock ().
Isso falhará se qualquer outra conexão tiver um registro bloqueado [Rlock ()] para esta área de trabalho.
Em seguida, a transação falhará com um erro de tempo de execução.
Acima é necessário para garantir que, quando a sequência de transações for executada no servidor, ninguém possa entrar
ocasião para Flock () a tabela, que então bloquearia o DbAppend () desejado no servidor.
Confie neste automático, ou garanta explicitamente um Flock () ou pelo menos um Rlock () ativo para
cada área de trabalho com registros para anexar. Este registro Rlock () ed não deve conter dados alterados,
apenas o bloqueio conta.

LETO_INTRANSACTION () ==> lTransactionActive

Apenas retorna um booleano se estiver dentro ou fora do céu.

7.3 Funções adicionais para a área de trabalho atual

LETO_COMMIT ()

! Descontinuada !
Mas a funcionalidade ainda está lá, é o que um DbCommit () comum faz. Ambos os comandos podem ser
usado durante as transações.

LETO_RECLOCK ([nRecord], [nSecs]) ==> l Sucesso
LETO_RECUNLOCK ([nRecord]) ==> lWasLocked
LETO_TABLELOCK ([nSecs]) ==> l Êxito
LETO_TABLEUNLOCK () ==> lWasLocked

Pode ser usado no cliente como substituto para estes no lado do servidor em: 9. Lado do servidor.
No cliente, eles redirecionam para métodos RDD universais para qualquer WA, enquanto <nSecs> mudará temporariamente
o valor de tempo limite configurável com RDDI_LOCKRETRY e <lWasLocked> será FALSE se o registro for
não bloqueado com R. (<nSecs> significa segundos inteiros: nSecs == 0,7 == 700 milissegundos RDDI_LOCKRETRY)
Se <nRecord> não for fornecido, é o RecNo () ativo.

LETO_DBEVAL (<cbBlock>, [<cbFor>], [<cbWhile>], [nPróximo], [nRegistro], [lRest] ,;
[[@] <lResultArr>], [<lNeedLock>], [<lDescend>], [<lFicar>], [cJoins])
==> lSucesso | aResults | xValue
Esta função é uma substituição 'drop-in' para DBEval (), e "leto_std.ch" pré-processa todas as chamadas DBEval () para
use esta função.
! Os codeblocks: <cBlock>, <cFor>, <cWhile> devem ser fornecidos literalmente ["{||}"] "para serem executáveis no servidor,
que na verdade também precisa da opção do servidor: Allow_udf = 1.
Se o parâmetro for fornecido como bloco de código (não literal), o WA deve ser processado localmente no lado do cliente. Este também é o
caso se um FILTER não otimizado ou RELATION não otimizado estiverem ativos. [não otimizar == apenas no cliente ativo].

Isso funciona da mesma forma que o comando DbEval () do Harbours, mas é otimizado para agir provavelmente como um UDF diretamente no servidor,
portanto, apenas o resultado é transferido pela rede, nem todos os registros [a serem] processados localmente.
Expressões de codeblock podem, mas não obrigatórias, começar / terminar com caracteres '{' e '}' - se eles estiverem faltando,
na frente um '{||' e no final um '}' é adicionado. "Literalmente codeblocks" são uma extensão LETO para DBEval ().

* Capacidade estendida do LetoDBf: passe dois parâmetros para os blocos de código: "{| <n>, <x> | ...}"
<cbBlock>: recebe como <n> número de registro de processamento, por exemplo:
"{| nDone | IIF (nDone == 1, DoThis (), DoThat ()), DoEver ()}"
Para resumir, o segundo parâmetro <x> dá o valor de retorno do último <cbBlock> executado,
que é NIL para a primeira execução. A seguir dá uma impressão de uso:
"{| nFeito, xVal | IIF (nFeito == 1, xVal: = 1,), xVal + = 1}"
<cbFor> e <cbWhile>: aqui <n> é o número já válido! registros processados,
significa o mesmo <n> que para o cbBlock, mas uma etapa antes: este <n> aumenta após um <cbFor> válido;
O segundo parâmetro é o número do registro geral avaliado (incluindo o 'inválido'.
Um exemplo para imprimir o uso com <cbFor>: verifique no máximo 1000 registros, pare após 3 válidos encontrados
"{| nReadyDone, nEvaluated | nReadyDone <3 .AND. nEvaluated <= 1000}"

<nNext>, <nRecord> e <lRest> são usados do mesmo tipo que na função Harbour DbEval ().

LetoDBf garante que o valor <nNext> a ser iniciado no registro ativo / topo da tabela é influenciado por
<lRest>: dado explicitamente como FALSE nNext começa no topo da tabela, senão começa no registro ativo.

<lResultArr> definido como TRUE (.T.) retorna uma matriz <aResultados> dos resultados de <cbBlock>
para cada registro processado. (para ser mais preciso: valor de retorno da última função dentro de <cbBlock>)
Com o padrão <lResultArr> == FALSE (.F.), O resultado é o valor de retorno de * último * executado <cbBock>.
O valor <lResultArr> pode ser dado em uma variável por referência (@) para receber posteriormente o conjunto de resultados.

<lNeedLock> set TRUE (.T.) informa Leto_DbEvil (), que os registros devem ser bloqueados para a execução de <cbBlock>.
Se a tabela * não * for aberta como <exclusiva> ou <compartilhada com set Flock ()>, RDDI_AUTOLOCK precisa ser ativado.
para RLock interno () feito por Leto_DbEvil () para cada registro válido, possível com um tempo limite fornecido com
RDDI_LOCKRETRY como milissegundos.
O padrão não é a necessidade de bloqueio, apenas para visualizar os dados do registro sem modificá-los.

Com <lNeedLock> == .F., O bloqueio manual é possível usando as funções LOCK dentro de <cbBlock>
[R | F] lock, DbR [un] lock e DbUnlock são inválidos no servidor, em vez de Leto_Rec [Un] Lock () deve ser usado,
essas duas funções são válidas no cliente e no servidor. Exemplo:
"{|| IIF (Leto_RecLock (), ..tarefa-a-fazer .., NIL), IIF (Leto_RecUnlock (), 1, -1) * RecNo ()}"
retornará com <lResultArr> == .T. uma matriz com RecNo de registros processados, com falha ao bloquear são negativos.
O exemplo acima como CB (sem cotação "") é executado localmente no cliente.

<lDescend> definido como TRUE (.T.) começará na parte inferior e DbSkip () na parte superior - o padrão é de cima para baixo.

<lRest> no mundo Harbour: um determinado valor é substituído por uma determinada condição <cbFor> ou <cbWhile>,
então <lRest> se tornará verdadeiro (.T.), o que significa continuar com o registro ativo.
LetoDBf apresenta: RddInfo (RDDI_DBEVALCOMPAT, .F.) - então a compatibilidade com o Harbour é ignorada,
e apenas o dado <lRest> determina se deve começar no TOPO de WA ou no registro ativo, independente de
dadas as condições, o que oferece uma maior flexibilidade.

<lStay> o padrão é, T., permanecerá no último registro processado,
com um dado .F. ele volta para o registro que estava ativo quando Leto_DbEval () foi iniciado.
Este FALSE também ativa para tentar um Flock () inicial para o primeiro registro processado, quando bem sucedido
aumentará drasticamente o desempenho se * muitos * registros tiverem que ser processados. Se Flock () falhar,
cada registro será testado para ser editado com Rlock ().
Usar Rlocks para alguns registros x00 é bom, talvez até mesmo para alguns milhares, mas muitos mais 'não fazem graça'.

<cJoins> é uma lista de áreas de trabalho conectadas ao WA ativo. Isso se refere à 'teoria dos conjuntos' de SQL,
portanto, é necessário um conhecimento básico dos tipos SQL JOIN. Implementados são:
CRUZAR; INTERNO; ESQUERDA externa; Exterior DIREITO; FULL exterior; VAZIO (como um DIREITO para não encontrado) tipos.
A lista de WA associados é fornecida com a máscara: "JOIN type, ALIAS, [expression];"
onde é importante que cada junção termine com ";" e qualquer um, exceto a junção CROSS, precisa de uma expressão.

```
   Uma expressão que não retorna um booleano tentará usar uma ordem de índice apropriada,
 também conhecida como DbSeek () para
   o resultado, onde qualquer "a-> x == b-> y" precisa usar técnicas de pulo para
encontrar os registros combinados.
   Um determinado <lNeedLock> também bloqueará automaticamente os WAs associados
adicionais ao WA mestre ativo,
   mais <cbFor> e / ou <cbWhile> podem referir-se a campos de todos os WA associados.
   Os tipos de junção podem ser usados combinados, com os seguintes limites:
   # apenas um JOIN permitido por WA inscrito, também conhecido como não é possível
ingressar no mesmo WA com mais de um JOIN
      (se necessário, abra a mesa dupla com um ALIAS diferente)
   # RIGHT Joins não podem ser combinados com INNER ou LEFT
   Um exemplo impossível, infringir todas as regras: "INNER, WA_1, ...; RIGHT, WA_1,
WA_1-> field1 == WA_1-> field2;"

 Valor RETURN: FALSE (.F.) Para FAILURE (por exemplo, o bloqueio automático falhou ao
bloquear todos os registros)
   <xValue> valor de retorno do último <cbBock> executado (da última função dentro)
   <l Êxito> == .T. se <xValue> for NIL
   <aResultados> array de <xResult> quando <lResultArr> == .T.
   Ocorre um RTE, se um dos <cb *> codeblocks for inválido, sintático ou se uma variável
local estiver lá.

 Dica: dê uma olhada em "leto_std.ch" para mais alguns exemplos, usados em 'comandos de
processamento de dados' ...


      LETO_SUM (<cFieldNames> | <cExpr>, [cFilter], [xScopeTop], [xScopeBottom])
                                                                  ==> nSumma | aSumma
 O primeiro parâmetro de leto_sum é uma lista de campos ou expressões separados por
vírgulas,
 Se um único campo ou expressão for passado, um único valor numérico é retornado,
 caso contrário, uma matriz com as somas para vários nomes / expressões de campo
fornecidos.
 Opcional <cFilter> é uma condição de filtro a ser tomada em vez de um possível filtro
ativo [DbSetFilter ()],
 opcional xScope [Top | Bottom] são valores de escopo para um pedido de índice ativo,
 Exemplo:
    leto_sum ("NumField1, numField2, #", "'JOY' $ cField", cScopeTop, cScopeBottom)
 retorna uma matriz com valores de campos de soma NumField1 e NumField2.
 Se o símbolo "#" for passado como nome de campo, leto_sum retornará uma contagem de
registros avaliados, por exemplo:
    leto_sum ("Sum1, Sum2, Sum1 + Sum2, #", cFilter, cScopeTop, cScopeBottom) -> {nSum1,
nSum2, nSum3, nCount}

      LETO_GROUPBY (cGroup, cFields, [cFilter], [xScopeTop], [xScopeBottom])
                                                                  ==> aValues
 Esta função retorna uma matriz bidimensional no formato: {{xGroup1, nSumma1, nSumma2,
...}, ...}
 <cGrupo> e <cCampos> podem utilizar funções, também conhecidas como uma expressão em
vez de apenas um nome de campo simples.
 Matriz resultante: o primeiro elemento de cada submatriz é o valor do campo <cGrupo>,
 os outros são a soma numérica dos campos / expressões.
 Se o símbolo "#" for passado como nome do campo em cFields, uma contagem de registros
avaliados para o grupo será retornada.

      LETO_ISFLTOPTIM () ==> lFilterOptimized

 Para determinar se um filtro ativo na área de trabalho selecionada é otimizado [também
conhecido como executado apenas no lado do servidor]
ou não otimizado [servidor envia todos os registros ao cliente, que então deve decidir
por si mesmo para registros válidos. ]
 Veja 5.2 para mais informações.

      LETO_FTS ([cSearch, [lCaseInsensitive], [lNoMemo]) ==> lFound [| cRawData]
 Full-Text-Search: com a string <cSearch> fornecida, * todos * os campos de uma tabela
são pesquisados com distinção entre maiúsculas e minúsculas,
```

```
 incluindo campos de memorando possíveis (armazenados externamente).
Alterável por set logigal .T. para <lCaseInsensitive>, e dito para <lNoMemos> para
excluir campos de memorando.
 Se <cSearch> não for uma string válida, todos os dados do registro (sem memorando)
serão retornados para processamento externo.

      LETO_MEMOISEMPTY (cnField [, cnAlias]) ==> lEmpty (TRUE para não um memofield)

 Esta é uma função otimizada para teste muito rápido, se um memofield do registro atual
está vazio ou não.
 A verificação será feita apenas no lado do cliente, portanto, nenhum tráfego de rede
para o servidor ocorrerá.
 Da mesma forma que aconteceria com um teste como: EMPTY (FIELD-> memofield), para o
qual o servidor enviará o
 todo o conteúdo de um memofield para o cliente, antes que o cliente possa decidir se
está vazio ou não.
 <cnField> cFIELDNAME ou nFIELDPOS, <cnAlias> cALIAS ou nSELECT ou WA ativo se vazio.

      DbInfo (DBI_BUFREFRESHTIME [, nBufRefreshTime]) ==> nOldVal

 Isso retorna o valor de tempo limite para o skipbuffer válido para esta tabela, antes
de um opcional
 a nova configuração é aplicada com <nNova configuração>.
 O padrão é nenhum tempo limite específico para uma tabela, também conhecido como usar o
valor de tempo limite de conexão geral.
 Com opcional <nBufRefreshTime> pode ser aplicada uma nova configuração apenas culpada
para esta tabela específica.
 "-1" == skipbuffer desabilitado, "0" == infinito skipbuffer, nHotBuffer> 0 ==
nHotBuffer / 100 segundos.
 Acima está também o intervalo possível para <nNovaSetting> - mais um valor <-1 irá
desativar novamente um determinado
configuração para esta tabela. Para obter o valor de tempo limite global, consulte:
função Leto_Connect (), 5° parâmetro.

      DbInfo (DBI_AUTOREFRESH [, lNewSetting]) ==> lSet

Isso retorna e define o comportamento, quando o valor de tempo limite de hotbuffer é
decorrido. Tempo limite padrão
é um segundo e pode ser alterado com Leto_Connect (). Se a tabela tiver um tempo limite
específico definido, este
têm precedência sobre o valor global. Então, com o próximo acesso a um campo após o
tempo limite decorrido,
 um DbSkip (0) interno é executado para atualizar os dados do servidor, se os dados no
servidor estiverem acessíveis
 para outros, também conhecido como registro ou tabela não bloqueada, tabela
compartilhada aberta e não somente leitura.
! Observe que o tempo limite desabilitado (-1) levará a um DbSkip (0) para cada acesso
de campo.
 Portanto, se você deseja acessar vários campos, é melhor definir o valor mínimo de 1 ==
0,01 segundo como tabela
 tempo limite específico - caso contrário, ocorrerá muito tráfego de rede.

      DbInfo (DBI_CLEARBUFFER)
      RDDInfo (RDDI_CLEARBUFFER)

  Este comando limpa o buffer de salto e força a obter dados novos com um Dbskip (0).
  <RDDI_CLEARBUFFER> fará isso para todas as áreas de trabalho LETO de conexão ativa.

      leto_DbCreateTemp (cFile, aStruct [, cDriver, lKeepOpen, cAlias, xDelim, cCdp,
nConnection])
                                                          ==> l Sucesso

  Substitui e estende a função Harbour HB_DBCREATETEMP (), já que não pode ser usada
para o driver RDD "LETO".
  O arquivo de cabeçalho 'leto_std.ch' irá traduzir a função Harbour em chamadas
leto_DbCreateTemp (),
  onde o parâmetro <cArquivo> está vazio.
```

Se <cArquivo> estiver vazio [NIL], você obterá uma tabela nomeada temporária 'real' no diretório temporário do sistema operacional.
Um determinado <cFile> se refere a uma tabela no servidor <DataPath>, provavelmente feito com um DbCreate () comum.
Ambas as tabelas resultantes são 'temporárias', o que significa que são * automaticamente excluídas * quando a tabela é fechada.
Se explicitamente <cDriver> é fornecido e diferente de "LETO", a função encaminha para HB_DBCREATETEMP ()
para criar uma tabela local no cliente - se <cDriver> estiver vazio [NIL ou ""], "LETO" será usado.
O parâmetro <aStruct> é obrigatório, deve ser sempre fornecido,
<lKeepOpen> é um parâmetro 'cego', sempre definido como verdadeiro (.T.) - para ter os mesmos parâmetros que DbCreate ().
Isso significa ainda que tais tabelas são sempre tabelas abertas <exclusivas> - até que sejam excluídas.
Usar um <cAlias> é recomendado, mas se não houver um nome-alias universal é criado automaticamente:
no caso de nome de arquivo derivado, caso contrário, um ALIAS é criado como "TMPWAxxxxxx".

leto_DbTrans (cnDstArea, aFields, cbFor, cbWhile, nNext, nRecord, lRest))
leto_DbSort (cToFile, aFields, cbFor, cbWhile, nNext, nRecord, lRest, cRDD, nConn, cCDP)
leto_DbArrange (cnToArea, aStruct, cbFor, cbWhile, nNext, nRecord, lRest, aFields))
leto_DbCopy (cFile, aFields, cbFor, cbWhile, nNext, nRecID, lRest, cRDD, nConn, cCDP, xDelim))
leto_DbApp (cFile, aFields, cbFor, cbWhile, nNext, nRecord, lRest, cRDD, nConn, cCDP, xDelim))
leto_DbTotal (cFile, xKey, aFields, xFor, xWhile, nNext, nRec, lRest, cRDD, nConnection, cCodePage)
leto_DbUpdate (cnAlias, cbKey, lRandom, aAssign, aFields)
==> aqui

Essas funções fazem o mesmo que __Db * (), veja também suas descrições,
mas o grupo acima pode usar uma expressão <cFor> e <cWhile> literalmente fornecida, capaz de enviar ao servidor:
então, toda a movimentação de dados pode acontecer no servidor sem carga de rede.
Se <bFor> ou <bWhile> forem codeblocks, também conhecidos como {|| ...} versus "{|| ...}" com apóstolos, estes
não pode ser transferido para o servidor e muito ser avaliado (desempenho lento) no lado do cliente.
O cabeçalho: "leto_std.ch" traduz a versão COMMAND correspondente para variantes letoDBf (),
por exemplo, 'CLASSIFICAR PARA cArquivo TODOS PARA cFor' -> letoDbSort (cToFile, {{todos os campos}}, "cFor", ...).

leto_DBJOIN (cnAlias, [cFile, [aFields], cbFor, cRDD, nConnection, cCodePage [, [lTemp]])
==> l Sucesso

Esta função estende a função __dbJoin (): se <Fields> estiver vazio, todos os campos do WA ativo
são usados para a tabela resultante, caso contrário, contém uma lista de nomes de campos de ambos WA, ou:
CBs fornecidos literalmente na forma: "{|| ..}" - estes são reconhecidos por início + fim com colchetes '{' '}'.
Os nomes de campo podem conter um prefixo ALIAS-> do WA do escravo, campos sem ALIAS assumido do WA do mestre.
Um CB literalmente pode ser pré-liderado por um 'alias->' como fieldname, provavelmente: "FieldName -> {|| ...}".
<cArquivo> se refere a uma tabela clássica no servidor <DataPath>, ou
com <cArquivo> vazio para um sem nome, com <cArquivo> preenchido mais <lTemp> definido como TRUE (.t.), para um
tabela 'exclusiva' aberta temporária sem nome ou nomeada - excluída ao fechar a tabela de resultados.
É considerado para ser usado com segurança também com campos de incremento automático.
Diferente de __dbJoin (): não inicia o mestre, mas o resultado WA (no topo) é selecionado depois,
as posições de registro dos WAs do escravo e mestre são restauradas.

7.4 Funções adicionais de rdd

leto_DbDriver (["cNewDriver"], ["cMemoType"], [nBlocksize])
==> aInfo

retornar uma matriz com e na ordem dos três parâmetros, os novos valores são opcionais.
O novo valor para <cNewDriver> é então válido para a próxima tabela de dados aberta / criada.
Novos valores para <cMemoType> e <nBlockSize> somente entram em ação para novas tabelas DBF criadas.
Apenas alterar <cMemoType> sem fornecer <cNewDriver> também mudará <cNewDriver>.
cNewDriver: "DBFNTX", "DBFCDX", "DBFNSX", "SIXCDX", "DBFFPT" (sem índice!)
cMemoType: "DBT", "FPT", "SMT"
nBlockSize: o padrão para DBT é 512, para FPT 64 e SMT 32 bytes.
o mínimo é de 32 bytes, o máximo de 65535 == 64 KB; novos valores como múltiplos de 32 bytes.
Somente se o servidor e a biblioteca cliente estiverem vinculados ao suporte de índice de bitmap rushmore (4.2.3),
adicional possível para <cNewDriver>: "BMDBFCDX", "BMDBFNTX", "BMDBFNSX"

leto_CloseAll ([cConnString]) ==> nulo

De alguma forma obsoleto: fechando todas as áreas de trabalho para uma conexão especificada ou padrão.
Esta função só é interessante para usuários com várias conexões de servidor e faz o mesmo que:
DbCloseAll () com configuração e restauração da conexão atual com Leto_ [Set | Get] CurrentConnetion ()

7.5 Configurando o paramentador do cliente

LETO_SETSKIPBUFFER (nSkip) ==> nSet (estatística de buffer usando)

Este buffer é destinado à otimização de várias chamadas de salto.
Esta função define o tamanho dos registros em cache em um buffer de salto para a área de trabalho atual.
Por padrão, o tamanho do buffer de salto é o valor de configuração do servidor CACHE_RECORDS. O buffer de salto é bidirecional.
O buffer de salto é atualizado após BUFF_REFRESH_TIME (padrão: 1 seg)
O valor mínimo é 1.
Se o parâmetro <nSkip> estiver ausente, a função retornará a estatística do buffer (número de ocorrências do buffer)
com o tamanho do conjunto efetivo do valor numérico fornecido ou 0 se nenhuma área de trabalho foi selecionada.
Relacionado ao tempo limite do skipbuffer, consulte também 7.3: DbInfo (DBI_BUFREFRESHTIME).

RddInfo (RDDI_REFRESHCOUNT [, <lSet>]) ==> lOldSet

Por padrão, o sinalizador RDDI_REFRESHCOUNT é definido como verdadeiro.
Se este sinalizador estiver definido, a função: RecCount () recupera a quantidade de registros do servidor.
Se não for definido, com dados de registro, os valores transmitidos são usados e podem estar um pouco fora do tempo.
Se outros aplicativos estiverem acrescentando registros à tabela, a nova contagem de registros não será vista imediatamente.

RddInfo (RDDI_BUFKEYNO [, <lSet>]) ==> lOldSet

Use-o somente se precisar consultar com frequência OrdKeyNo (), por exemplo, durante a navegação para atualizar a barra de rolagem.
O valor padrão é .F., O que significa que a função: OrdKeyNo () enviará uma solicitação extra ao servidor.
Se <lSet> for definido como TRUE (.T.), Os valores para OrdKeyNo () são transmitidos com os dados de registro e sem extras

o pedido é enviado. Geralmente, este é um desempenho muito diminuindo conforme a contagem do tempo necessário no servidor, tão imediato
 desative-o (.F.) se não for mais necessário.

```
     RddInfo (RDDI_DEBUGLEVEL [, nNewLevel]) ==> nOldLevel
```

 Relata [e altera] o nível de depuração no servidor, responsável pela quantidade de feedback nos arquivos de log.
 Use com cuidado, os arquivos de log crescerão em um servidor ocupado em apenas alguns segundos MB passo a passo ...
 Para valores possíveis veja 4.1: letodb.ini ...
Com <nNewLevel> isso pode ser alterado em tempo real, nenhuma reinicialização do servidor é necessária. Isso então se aplica a todos
 conexões de servidor ativas e novas.

```
     RddInfo (RDDI_AUTOLOCK [, lActivated]) ==> lWasActive
```

 Por padrão desativado, é usado em Leto_DbEval () para registros Rlock () automáticos antes de alterar os dados do registro.
 Se não for alterado o tempo limite padrão: 0 com RddInfo (RDDI_LOCKRETRY [nMilliSec]), tal Rlock () será tentado apenas
uma vez com sucesso ou não. Com o tempo limite, o servidor tentará várias vezes durante o intervalo de tempo. Isso poupa
 solicitação repetida do cliente pela rede.

```
    RddInfo (RDDI_BUFREFRESHTIME [, nNewTimeout]) ==> nTimeout
```

 Isso é semelhante ao 5° parâmetro de Leto_Connect (), o valor de tempo limite global para o cache skipbuffer.
 Um <lNewTimeout> não fornecido, ele retorna o valor de tempo limite ativo, onde '-2' indica que nenhuma conexão está ativa.
 Um determinado <lNewTimeout> é válido a partir de então para * novas tabelas a serem abertas *.
 <lNewTimeout> unidade é ms (1/100 s), o intervalo é de '-1' == sem atualização, '0' == infinito e> 0.

```
    RddInfo (RDDI_CONNECT, {"LETO", cAddress, [cUserName], [cPassword],
                            [nTimeOut], [nBufRefreshTime], [lZombieCheck})
                                                    ==> nConnection, -1 se
falhou
```

Fusado para tráfego de rede em tempo real sob demanda:
 Exemplo de uso:

```
   #include "dbinfo.ch"
   ...
   SE ASCAN (RddList (), "LETO")> 0 / * verificar LETO disponível * /
      RddSetDefault ("LETO") / * definido como driver padrão * /
      nConnection: = RddInfo (RDDI_CONNECT, {"LETO", cAddress, ...})
   FIM SE
```

```
    RddInfo (RDDI_DISCONNECT ([cAddress]) ==> l Êxito
```

 Leto_Disconnect () semelhante, "LETO" deve ser o driver padrão em tempo de execução para fechar a conexão [ativa].

```
    RddInfo (RDDI_DBEVALCOMPAT ([, lNew]) ==> lOldSet
```

 Alterne o comportamento de compatibilidade para Leto_DbEval () relacionado ao seu argumento <lRest>.
O padrão (.T.) É se comportar da mesma forma que o Harbour, desabilitando apenas o parâmetro <lRest> para determinar. se Leto_DbEval ()
 começa o processamento em TOP ou no registro ativo.

```
    RddInfo (RDDI_DBEVALTIMEOUT ([, nMilliSec]) ==> nOldValue
```

Para consultar [ou alterar] o tempo limite de um Leto_DbEval () no servidor [padrão: 120000 ms == 120 s].
Após este intervalo de tempo, um Leto_DbEval () rodando otimizado no servidor (também conhecido como não executado localmente) irá parar
e retornar um erro. É para limitar uma solicitação malformada e fora de controle a uma execução não infinita no servidor.

LETO_SETSEEKBUFFER (nRecsInBuf) ==> 0
! DESCONTINUADA !

LETO_SETFASTAPPEND (lFastAppend) ==> .F.
! DESCONTINUADA !
por causa de problemas de design feios. Ele é deixado como uma função fictícia sem fazer nada.
Se essa funcionalidade anterior for realmente necessária, encapsule sua solicitação em uma transação.

7.6 Funções de arquivo

IMPORTANTE: todos os comandos de arquivo começam no servidor com o DataPath, então <cFileName> é um caminho relativo
para o caminho raiz definido em letodb.ini com DataPath. Apenas UM (1) único ".." é permitido.
Exceção: nomes de arquivos começando com "mem:" redirecionam para a RAM do servidor (HbNetIO FS virtual)

Por razões de portabilidade é recomendado o uso de nomes de arquivo simples, sem (ou vazio) prefixo de servidor
- após o estabelecimento inicial de uma conexão com Leto_Connect ().
Se todo o prefixo de conexão "//..:../" for omitido, a conexão atualmente ativa será usada;
se a conexão ainda não estiver disponível, ela será estabelecida por esse prefixo de conexão.
Depois que a conexão for estabelecida, você pode deixar de lado todo este IP: prefixo da porta ...

Alternativamente, no estilo ANTIGO, o parâmetro <cFileName> de todas as funções de arquivo * pode * conter um
string de conexão ao servidor letodb em um formato:
// endereço_IP: porta / [mem:] [\] nome_arquivo
onde IP_address também pode ser um nome DNS, como por exemplo "localhost".
Este estilo antigo pode ser uma alternativa se estiver trabalhando com vários servidores LetoDBf simultaneamente,
mas também pode ser feito dessa forma trocando a conexão ativa com o Leto_Connect () freqüentemente observado.

Dica experiente: no diretório de inclusão está o cabeçalho: "letofile.ch".
Aplicando-o como cabeçalho padrão adicional: "-u + letofile.ch" na linha de comando / .hbp para hbmk2,
todas as funções de arquivo local em sua fonte funcionam no lado do servidor. Para algumas funções funcionarem,
A opção de configuração 'Allow_UDF' deve ser definida para o servidor.
Qual grupo de funções trabalha então no servidor, e quais grupos continuam a atuar localmente no cliente,
pode ser configurado no cabeçalho letofile.ch '- por padrão, todos, exceto o grupo: "Fxxx ()" é
pré-processado para usar variantes Leto_F * ().

Leto_FError ([lAskServer]) ==> nError
Retorna um código de erro definido por (alguns, não para todos) funções de arquivo no cliente.
NOVO: com <lAskServer> opcional definido como TRUE (.T.), Uma consulta é enviada ao servidor.

Leto_File (cFileName [, @cFilePath]) ==> lFileExists

```
 Determine se o arquivo existe no servidor, analógico da função File ().
 <cFileName> é sempre relativo ao <DataPath> de 'letodb.ini'
 Se <cFileName> for um nome de arquivo * simples * sem elemento de caminho, e
<@cFilePath> fornecido por referência,
 mais um caminho de pesquisa é definido com Leto_SetPath (), <@cFilePath> contém [sub]
diretório mais o nome do arquivo
 onde <cFileName> foi encontrado.

      Leto_FCopy (cFilename, cFileNewName [, lMove]) ==> -1 se falhou
 Copie um arquivo no servidor com um novo nome no servidor.
 Com o conjunto opcional <lMove> == true (.T.), A fonte é excluída posteriormente.

      Leto_FErase (cFileName) ==> -1 se falhou
 Exclua um arquivo no servidor.

      Leto_FRename (cFileName, cFileNewName) ==> -1 se falhou
Renomeie um arquivo: <cFileName> ==> <cFileNewName>. <cFileNewName> deve ser sem
 string de conexão.

      Leto_MemoRead (cFileName) ==> cString
 Retorna o conteúdo do arquivo no servidor como string de caracteres, analógico de
 Função MemoRead ().
! FIXO ! Se o último caractere no arquivo for o caractere 26 == strg-z, ele será
removido.

      Leto_MemoWrite (cFileName, cBuf) ==> lSuccess
 Grava uma string de caracteres em <cBuf> em um arquivo no servidor, análogo a
 Função MemoWrit ().
! FIXO ! Observe que o caractere '26' == 'strg-z' é aplicado a <cBuf>.

      Leto_Directory ([cDir] [, cnAttr]) ==> aDirectory
 Retorna um conteúdo do diretório no servidor no mesmo formato da função Directory ().
 Sem um <cDir> fornecido, o diretório raiz do DataPath é usado.

      Leto_DirExist (cPath) ==> lDirExists
 Determine se existe um directory no servidor, analógico da função Leto_File (), mas
 para diretórios

      Leto_DirMake (cPath) ==> -1 se falhou
Cria um diretório no servidor. [renomeado, anteriormente: Leto_MakeDir]

      Leto_DirRemove (cPath) ==> -1 se falhou
 Exclui um diretório no servidor

      Leto_FileSize (cFileName) ==> -1 se falhou
 Retorna um comprimento de arquivo no servidor

      Leto_FileTime (cFile [, dDate] [, cTime] [, nReturnType]) ==> vários retornos
 Ele consulta _ou_ define uma data / hora / carimbo de data / hora para o arquivo
<cArquivo>.
 Sem qualquer um dos opcionais <dDate> e <cTime>, uma consulta é feita no servidor,
 então, dependendo de <nReturnType>, o valor retornado é:
 '0' = booleano para sucesso, '1' = carimbo de data / hora, '2' = dDate, '3' = cTime com
milissegundos,
 '4' = cTime encurtado de 8 caracteres, '5' = nJulianDay.
 Se for fornecido <dDate> e / ou <cTime> _ou_ um carimbo de data / hora para <dDate>,
então esta data e hora irá
ser definido para o arquivo. Se omitido <dDate> ou <cTime>, o valor ausente será obtido
do
 arquivo existente, também conhecido como <dDate>, mas ausente <cTime>, altera apenas o
filedate, mas não o filetime,
 e apenas <cTime> mudará o tempo do arquivo, mas não o arquivo.
 <dDate> e / ou <cTime> deve ser NIL ou preenchido com o tipo correto, <cTime> pode
conter MiliSegundos
 no formato: "hh: mm: ss.xxx".
 O valor de retorno para qualquer ação * definir * é um booleano (.T ./. F.) Para
sucesso ou falso se não houver conexão,
```

```
 ou o valtype desejado como um valor em branco no caso de falha ao consultá-lo.

      Leto_FileMD5 (cFileName [, lBinary] ==> cMD5sum
 Retorna a soma MD5 de um arquivo no servidor, string vazia se não encontrada ou
qualquer outro erro.
 Por padrão, como string hexadecimal de 32 caracteres, com <lBinary> == true (.T.) Como
valor binário de 16 bytes.

      Leto_FileAttr (cFile [, cnNewAttr] [, lAsNumeric]) ==> cnAttr
 Obter (sem cNewAttr fornecido) ou definir atributos de arquivo <cNewAttr>, onde o valor
retornado
 <cnAttr> são os atributos ativos (após uma alteração opcional com <cnNewAttr>)
 <cnNewAttr> pode conter em primeiro lugar um "-", para reverter o (s) seguinte (s)
atributo (s),
por exemplo; "-A" irá remover o atributo 'arquivo'; "-" irá remover todos os atributos.
 <lAsNumeric> opcional como verdadeiro (.T.) Retornará os atributos como valor numérico.
 Os atributos de arquivo são válidos apenas para FileSystem que os suporta!

      Leto_FileRead (cFileName, [nStart], [nLen], @cBuf) ==> -1 se falhou
 Leia o conteúdo do arquivo no servidor de deslocamento <nIniciar> e comprimento máximo
de <nLen>.
 Os padrões para <nIniciar> e <nLen> são 0, e nLen == 0 significa o arquivo inteiro.

      Leto_FileWrite (cFileName, [nStart], cBuf) ==> lSuccess
 Grava a cadeia de caracteres <cBuf> em um arquivo no servidor a partir do deslocamento
<nInício>.
 O padrão para <nIniciar> é 0, significa no início do arquivo.

      Leto_FCopyToSrv (cLocalFileName, sServerFileName [, nStepSize])
                                                          ==> l Sucesso
      Leto_FCopyFromSrv (cLocalFileName, sServerFileName [, nStepSize])
                                                          ==> l Sucesso
 Copie um arquivo de / para o cliente para / do servidor, onde:
 <cLocalFileName> é o nome do arquivo no lado do cliente,
 <sServerFileName> é o nome do arquivo no servidor que pode conter informações de
conexão "// IP: porta /".
 Opcional <nStepSize> determina o tamanho dos bytes a serem copiados com uma etapa,
padrão se
 não fornecido é 1 MB.
 <sServerFileName> pode conter apenas o prefixo: "mem:" para arquivos na RAM,
 <cLocalFileName> pode conter qualquer prefixo de redirecionador conhecido por Harbour.
 Um backup simples:
    aArr: = Leto_Directory ("*")
    AEval (aArr, {| aItem | Leto_FCopyFromSrv (aItem [1], aItem [1]})
 Copie de um servidor HbNetIO conectado um arquivo para LetoDBf localizado na RAM:
    Leto_FCopyToSrv ("net: hbnetio.txt", "mem: RAMfile.txt")

      Leto_ProcessRun (<cCmd>, NIL, [<@cStdOut>], [<@cStdErr>])
                                                          ==> nError
 * TENHA CUIDADO COM O COMANDO <cCmd> *, pois um processo não finalizado bloqueará a
conexão
 infinito e será necessário reiniciar o servidor.
 Muito semelhante em uso à função Harbour: hb_ProcessRun (), onde o uso de stdin é
 não implementado, e o comando <cCmd> é executado com o diretório de trabalho definido
como "DataPath"
 opção, o que significa que os arquivos no servidor podem ser referenciados usando um
caminho relativo.
 <cCmd> normalmente contém um prefixo não obrigatório, dependendo do sistema operacional
no servidor:
 # para Windows: cmd / C "comando para executar"
 # para Linux: / bin / bash -c "comando para executar"
 e encapsular a string de comando acima entre aspas simples <'>.
 O <nError> retornado é o ErrorLevel que um comando executado retorna ao sistema
operacional, onde '0'
 normalmente significa que nenhum erro ocorreu.
 Duas variáveis opcionais <@cStdOut> e <@cStdErr> fornecidas pela referência [@] conterão
 a saída da string do comando executado para o dispositivo stdout e stderr.
```

```
      Leto_FOpen (cFile [, nMode]) ==> nHandle
      Leto_FCreate (cFile [, nMode]) ==> nHandle
      Leto_FSeek (nHandle, nBytes [, nOffset]) ==> nPos
      Leto_FRead (nHandle, @cBuffer, nLen) ==> nRead
      Leto_FWrite (nHandle, cBuffer [, nLen]) ==> nWritten
      Leto_FClose (nHandle) ==> l Sucesso
      Leto_FEof (nHandle) ==> lEndOfFile

      Leto_FReadStr (nHandle, nLen) ==> cBytes
 Pára de ler em CHR (0)

      Leto_FReadLen (nHandle, nLen) ==> cBytes
Versão binária de Leto_FReadStr () incluindo qualquer! char para ler

 As funções acima fazem o mesmo e com os mesmos parâmetros que os pendentes Harbour sem
'Leto_'
prefixo, também conhecido como Leto_FOpen () == FOpen (), mas eles atuam em arquivos no
servidor. Nomes de arquivos <cArquivo>
 respeitar o caminho de dados do servidor, são relativos a ele.
 É garantido que todos os arquivos abertos / criados sejam fechados com o fim da conexão,
 e Leto_FClose () fechará apenas os arquivos abertos / criados com Leto_FOpen /
Leto_FCreate.



      7.7 Funções de gerenciamento

      LETO_MGGETINFO () ==> aInfo [17]
 Esta função retorna os parâmetros da conexão atual como uma matriz de 17 elementos
 de valores de tipo char:
 aInfo [1] - contagem de usuários ativos
 aInfo [2] - contagem máxima de usuários
 aInfo [3] - tabelas abertas
 aInfo [4] - máximo de tabelas abertas
 aInfo [5] - 0 [tempo de atividade do servidor movido para LETO_MGGETTIME ()
 aInfo [6] - contagem de operações
 aInfo [7] - bytes enviados
 aInfo [8] - bytes lidos
 aInfo [9] - índices abertos
 aInfo [10] - índices máximos abertos
 aInfo [11] - caminho de dados
 aInfo [12] - tempo de CPU (s) do servidor na soma para todas as conexões
 aInfo [13] - ulWait
 aInfo [14] - contagem de transações
 aInfo [15] - contagem de transações com sucesso
 aInfo [16] - 0 [memória atual usada] - movido para LETO_MGSYSINFO ()
 aInfo [17] - 0 [memória máxima usada] - movido para LETO_MGSYSINFO ()

      LETO_MGGETUSERS ([nTabela]) ==> aInfo [x, 5]
 A função retorna uma matriz bidimensional, cada linha contém informações sobre o
usuário:
 aInfo [i, 1] - número do usuário
 aInfo [i, 2] - endereço IP
 aInfo [i, 3] - nome líquido do cliente
 aInfo [i, 4] - nome do programa
 aInfo [i, 5] - tempo limite

      LETO_MGGETTABLES ([nUser]) ==> aInfo [x, 3]
 aInfo [i, 1] - número da mesa
 aInfo [i, 2] - nome do arquivo da tabela de dados (sem caminho de dados raiz do
servidor).
aInfo [i, 3] - 0 se solicitado para todas as conexões (nUser <0), senão cliente! número
da área de trabalho.
aInfo [i, 4] - vazio "" se solicitado para todos os usuários, caso contrário, nome ALIAS
do cliente! área de trabalho .
```

```
aInfo [i, 5] - compartilhado? 'T': 'F',
 aInfo [i, 6] - RDD usado
 aInfo [i, 7] - tipo de memorando: 1 = DBT, 2 = FPT, caso contrário SMT

      LETO_MGGETINDEX ([nUser], [nTabela] ==> aInfo [x]
 array com nomes de arquivos de arquivos de índice abertos, global para todas as
conexões (nUser <0),
 ou apenas para um determinado usuário e também apenas para uma tabela de dados
específica.
 Os números da tabela de dados devem ser recuperados antes com a função acima.

      LETO_MGSYSINFO () ==> aInfo [8]
 aInfo [1] - espaço livre em disco
 aInfo [2] - caminho de dados do servidor
 aInfo [3] - número de núcleos de CPU do servidor
 aInfo [4] - RAM disponível no servidor
 aInfo [5] - modo de servidor (1 == No_Save_WA = 1, 2 == Share_Tables = 1, 3 ==
Share_Tables = 0)
 os próximos quatro itens a seguir precisam de construção especial do Harbour com
estatísticas de memória (todos os demais: 0),
 e são apenas para fins de depuração - para constantes, consulte: hbmemory.ch
 aInfo [6] - HB_MEM_USEDMAX este e os três seguintes precisam de construção especial
Harbour
 aInfo [7] - HB_MEM_STACKITEMS com estatísticas de memória - para constantes consulte:
hbmemory.ch
 aInfo [8] - HB_MEM_STACK
 aInfo [9] - HB_MEM_STACK_TOP

      LETO_MGGETTIME () ==> aInfo [3]
 A função retorna a matriz {<dDate>, <nSeconds>, <nServerUp>}:
 aInfo [1] - servidor dDate;
 aInfo [2] - hora do servidor em nSegundos após a meia-noite.
 aInfo [3] - tempo UP do servidor em nSegundos
 Converta os primeiros valores em uma variável datetime (Harbour):
 hb_DTOT (aDateTime [1], aDateTime [2])

      LETO_MGID ([<lRefresh>]) ==> nConexão
 A função retorna o número de ID que esta conexão tem no servidor.
 Para isso não é necessária nenhuma consulta ao servidor, as informações estão
disponíveis no lado do cliente.
 <lRefresh> opcional envia uma consulta ao servidor, utilizável para testar se a conexão
está ativa e funcionando,
 como feito alternativamente com LETO_PING (), pois uma conexão de rede inválida
retornará '-1'.

      LETO_PING ([<cConnection>]) ==> ao vivo
A função retorna <lAlive> como verdadeiro [.T. ] se o servidor respondeu por um 'ping'
de envio.
 Sem <cConnection> opcional, a conexão ativa atual é usada.

      LETO_MGLOG ([<nConnectID>], [<nRows>], [<lErase>] ==> cLogContent
 A função recupera o conteúdo do arquivo de log [letodbf_xx.log] da conexão ativa.
 Se nenhum arquivo de log estiver disponível, uma string vazia "" é retornada.
 Dado opcional <nConnectID>> = 0 recuperar o arquivo de log da conexão com aquele ID,
 <nRows> pode ser usado para limitar a quantidade de últimas linhas da parte inferior
para conteúdo grande esperado,
 nenhum valor fornecido ou '0' recuperará todo o conteúdo.
Opcional <lErase> == true [.T. ] apagará o arquivo de log após recuperar o conteúdo.

      LETO_MGKILL (<nConnection> | <IPAddress>) ==> nConnectionClosed
 Mate o número do usuário fornecido por <nConnection> ou cIPAddress,
 retorna o número de conexão interrompida ou -1 em caso de não encontrada

      LETO_LOCKCONN (<lOnOff>) ==> l Êxito
 Após leto_lockconn (.T.), A solicitação de novas conexões são bloqueadas pelo servidor,
até
 leto_lockconn (.F.) é chamado ou o aplicativo / conexão termina, o que bloqueou o
servidor.
```

```
      LETO_LOCKLOCK ([<lOnOff>] [, <nSecs>] [, <nDelaySecs>])
                                                    ==> l Sucesso
 Esta função bloqueia (<lOnOff> = true) ou desbloqueia o servidor (<lOnOff> = false).
 Sem nenhum argumento, o status ativo é retornado.
O bloqueio do servidor existe enquanto a função é chamada com <lOnOff> == false [.F. ]
ou o
 o aplicativo / conexão termina o que bloqueou o servidor.
 Possui dois comportamentos diferentes, dependendo da opção "Backup_Info" na
configuração do servidor.
 Comportamento antigo, (com uma string vazia ou menor que 3 caracteres fornecidos para
'Backup_Info'):
   A função bloqueia o servidor contra novos bloqueios de arquivo / registro e espera o
intervalo de tempo <nSecs> que
   todos os bloqueios existentes para áreas de trabalho abertas COMPARTILHADAS são
desbloqueados.
   Caso não haja sucesso, o servidor é desbloqueado novamente, e <lSucesso> será falso
[.F.]
   ** Este bloqueio antigo não cobre tabelas abertas EXCLUSIVAS: se o servidor estiver
bloqueado, eles podem
   não anexar novos registros [DbAppend () -> false (.F.); NetErr () -> verdadeiro
(.T.)] E alterações de dados
   para tais registros não anexados são perdidos. Dados contrários podem ser alterados
para registros existentes.
   (Isso ocorre porque o bloqueio de estilo antigo trata dos bloqueios e as tabelas
EXCLUSIVAS não têm nenhum.)
 Novo comportamento [padrão]:
   Em primeiro lugar, o servidor é bloqueado contra novas conexões [semelhante a um
LETO_LOCKCONN (.T.)],
   então, todos os clientes ativos recebem uma nota informativa sobre o servidor que
deseja bloquear contra alterações de dados.
   Esta nota informativa é exibida em cada cliente e interromperá totalmente o
aplicativo.
   <nSecs> é o tempo limite, o servidor dá a todos os clientes para desbloquear os
bloqueios e fechar suas tabelas,
   e esse valor deve durar alguns segundos para que muitas conexões reajam a essa
solicitação.
   Opcional <nDelaySecs> dá aos clientes um intervalo de tempo para recusar o bloqueio
do servidor e deixá-los
   continuar seu trabalho de aplicação. Depois de <nDelaySecs>, o aplicativo cliente
finalmente bloqueará
   atividade adicional - e desbloqueia quaisquer bloqueios de arquivo / registro
existentes.
   Para um servidor LetoDBf rodando no sistema operacional Windows, também todas as
tabelas EXCLUSIVAS abertas são fechadas, para que
   eles podem ser acessados no servidor para um backup.
   Com um '0' explicitamente fornecido para <nDelaySecs>, também conhecido como o
aplicativo não tem tempo limite para reclamar,
   TUDO! possíveis (*) as tabelas que podem ser fechadas são fechadas, independentemente
do sistema operacional para o servidor. Isso é útil para
   tarefas administrativas no servidor, onde é necessário abrir uma mesa em modo
exclusivo. O administrador
   pode até reconstruir um índice, mas deve evitar qualquer ação de alteração de número
de registro [RecNo ()] como PACK,
   ZAP ou SORT - porque os aplicativos que aguardam para continuar esperam registros
exatamente nas mesmas posições.
   [(*) tabelas temporárias nunca podem ser fechadas sem perdê-las. ]
   Depois de finalmente bloquear com sucesso todos os clientes, o aplicativo cliente só
pode ser encerrado - ou tem que
   espere o servidor desbloquear novamente.
   Ao desbloquear o servidor, os clientes obtêm uma nota informativa, as tabelas
fechadas são reabertas e todos
   os bloqueios anteriores são restaurados. Quando todos os clientes estiverem prontos
com isso, o aplicativo obterá o
   nota informativa para continuar seu trabalho anterior exatamente no ponto em que foi
interrompido.
```

```
      LETO_TOGGLEZIP ([<nCompessLevel> [, <cPassword>]]) ==> nValueBefore
 Esta função sob demanda [desativa] ativa a compressão do tráfego de rede entre o
servidor e o cliente,
 onde nCompessLevel é:
 -1: == compactação desativada, tráfego de dados regularmente otimizado,
 [0 = nenhuma compactação, usado apenas pelo desenvolvedor para fins de depuração]
 1: nível de compressão LZ4 [ou zlib].
 Se você compilou o servidor e o cliente LetoDBf com compressão LZ4 de alta velocidade
em tempo real [padrão],
 então 1 é o valor recomendado para compressão.
Se LZ4 for desabilitado durante a construção de LeoDBf, a compactação zlib será usada
pelo Harbour. Também aqui '1' está
 valor recomendado, mas zLib conhece os valores de 1 a 9.
 Com nCompression> = 0, também uma cPassword pode ser fornecida para criptografia de
tráfego adicional,
 A conexão inicial ao servidor (por exemplo, LETO_CONNECT ()) é sempre feita no modo de
tráfego regular [-1].
 Sem qualquer parâmetro de função, a configuração de compressão ativa é retornada sem
alterá-la.

      LETO_LOGTOGGLE () ==> l Sucesso
      LETO_LOGREPLAY ([<cFile>], [<cExludeAction>], [<nTSFrom>], [<nTSTo>],
[<nExcludeConnect>])
                                                          ==> nResult
      LETO_LOGREQUEST ([<lLogToggle>], [<nOffSet>], [<lAuditOnOff>])
                                                          ==> cResult
 As funções são para gerenciar o registro de mudança de dados, ativado pela opção
letodb.ini: "Data_LogFile"
 # Leto_LogToggle () renomeia com segurança o log ativo com data e hora no nome do
arquivo,
 para reiniciar o login em um novo arquivo de log.
 # Leto_LogRequests () solicita até 500 KB de um registro de alteração de dados
existente,
 começando no final de uma solicitação anterior [com o padrão: '-1' para <nOffSet>].
 Um <nOffSet> explicitamente fornecido irá redefinir a posição do arquivo para reiniciar
a leitura no topo do log.
 Separado e independente do log de mudança de dados do servidor, um log para auditoria
pode ser recuperado usando o
3° parâmetro: <lAuditOnOff> == true [.T. ] inicia um registro de auditoria sobre
mudanças de dados, que é registrado em
 o log de conexão letodbf_xx.log, seu conteúdo é retornado interrompendo o log de
auditoria com false [.F.]
 # Leto_LogReplay () repete as ações de mudança de dados no arquivo <cArquivo> - se
nenhum nome de arquivo for fornecido,
 o arquivo de log do servidor configurado em "Data_LogFile" é usado como fonte.
 As opções opcionais <cExludeAction>, <nTSFrom>, <nTSTo> e <nExcludeConnect> podem ser
usadas para excluir
 alterações de dados específicas sendo reproduzidas - mais descrição a seguir ...


      7.8 Funções de gerenciamento de conta de usuário

      LETO_USERADD (cUserName, cPass [, cRights]) ==> l Êxito
      LETO_USERDELETE (cUserName) ==> l Êxito
  Nova função ;-), agora um usuário pode ser excluído
      LETO_USERPASSWD (cUserName, cPass) ==> lSuccess
      LETO_USERRIGHTS (cUserName, cRights) ==> lSucesso

      LETO_USERFLUSH () ==> l Êxito
  Grava as mudanças na memória em arquivo no armazenamento, será feito outra coisa
quando o servidor for encerrado.

      LETO_USERGETRIGHTS () ==> cRights
  Retorne os direitos do usuário ativo.

  Geralmente, o usuário que deseja alterar / adicionar / excluir propriedades do usuário
deve ter direitos de administrador no caso
```

em letodb.ini qualquer Pass_For_ * é definido como '1'.
cOs direitos são três letras: Y == permitir, N == negar, em sequência para:
admin == alterar / adicionar usuários; gerenciar == usuário do console; write == permite alterar dados em tabelas DBF.

Todo o arquivo de senha no armazenamento agora é criptografado, e também após o carregamento do conteúdo durante
servidor iniciar na memória, as senhas são mantidas criptografadas. Eles serão descriptografados apenas
'em tempo real' para verificar novamente a senha do usuário que solicita acesso.

7.9 Funções variáveis de servidor

Como uma variável PUBLIC é para um aplicativo, isso é algo parecido com uma variável pública para todos
as conexões / aplicativos / e o próprio servidor para trocar / acessar / modificar o conteúdo desse.
As variáveis podem ser 'agrupadas' em grupos de variáveis, como por exemplo um grupo de variáveis usado por
um aplicativo específico ou um grupo de variáveis apenas para o próprio servidor.

```
LETO_VARSET (cGroupName, cVarName, xValue [, nFlags [, @xRetValue]])
                                                    ==> l Sucesso
```

Esta função atribui o valor <xValue> à variável <cVarName> do grupo <cGroupName>.
<xValue> pode ser:
- booleano (.T., .F.)
- inteiro (sem decimais, pode ser incrementado e decrementado)
- decimal (inteiro com decimais, por exemplo, 4,321)
- limitado em tamanho!:
- data (formato de data [NOVO], será tratado internamente como AAAAMMDD
- string ([NEW] também string binária, significa conter qualquer caractere como, por exemplo, CHR (0))
- array ([NOVO] {...} com qualquer tipo de item acima, com tamanho limitado)

String / array são limitados por padrão a somar no máximo 64 MB, o máximo pode ser alterado em
letodb.ini com opção de configuração "Max_Var_Size". Uma única string / array limita-se a 1/4 do total
máximo (padrão 16 MB).
Isso é por razões de segurança, para que usuários malucos não consigam preencher toda a memória do servidor.
Só é permitido atribuir um novo valor do mesmo tipo a uma variável existente.
Nomes de grupos e Var- NÃO são cortados de espaços em branco, mas char: ';' é um caractere inválido.
SE <@xRetValue> for dado por referência (@), ele manterá o valor antigo antes do novo um foi definido. Isso só será feito para tipos booleanos e numéricos, não para string etc.

O parâmetro opcional <nFlags> define o modo / limitações de criação da variável.
Esses sinalizadores podem ser combinados agregando (+) as constantes:
LETO_VCREAT - [NOVO como padrão] cria variável se ela não existir;
não faz mal se a variável já existe, neste caso, sem mais efeito
LETO_VNOCREAT - [NOVO] defina isto se você não quiser a criação automática acima
LETO_VOWN - própria variável do usuário (excluída após a desconexão do usuário)
LETO_VDENYWR - negação de gravação para outros usuários
LETO_VDENYRD - negar leitura para outros usuários
LETO_VPREVIOUS - apenas um sinalizador válido para Leto_VarIncr () e Leto_VarDecr ():
retorna o valor antes do incremento / decremento da variável

```
LETO_VARGET (cGroupName, cVarName) ==> xValue
```

Valor de retorno da função da variável <cVarName> do grupo <cGroupName>

```
LETO_VARINCR (cGroupName, cVarName, [nFlags], [nVal]) ==> nValue
LETO_VARDECR (cGroupName, cVarName, [nFlags], [nVal]) ==> nValue
```

```
 Valor numérico de incremento / decremento da função da variável <cVarName> do grupo
<cGroupName>.
 [NOVO] Se <nVal> não for fornecido explicitamente, 1.0 é incrementado / decrementado,
 e todos os valores numéricos (inteiros ou decimais) agora são permitidos.
 Essas duas funções são boas para somar um valor, onde incrementar um valor negativo
está em vigor
 uma diminuição e vice-versa.
 Observe que, por exemplo, o resultado de uma divisão: 'x / y' é sempre um valor decimal.
 Um valor numérico, uma vez definido como inteiro, continuará a ser um número inteiro
sem decimais.

      LETO_VARDEL (cGroupName [, cVarName]) ==> lSuccess

 Sem <cVarName>, todos os membros variáveis de <cGroupName> e o próprio grupo são
excluídos.
 Com o <cVarName> fornecido, a variável <cVarName> é excluída do grupo <cGroupName>.

      LETO_VARGETLIST ([cGroupName [, nMaxLen]]) ==> aList

 Retornos de função
 # sem um parâmetro de string para <cGroupName>:
   matriz com nomes de grupo {<cGroupName1>, ...}
 # com <cGroupName> válido, sem o parâmetro <nMaxLen> ou <0:
   array com nomes de variáveis no grupo
 # com <cGroupName> válido e com <nMaxLen>> = 0:
   matriz bidimensional com variáveis: {{<cVarName>, <value>}, ...}
   <nMaxLen> == 0 significa valores de string ilimitados, caso contrário, estes são
limitados em tamanho
   e corte se for mais longo como <nMaxLen>.
   IF <nMaxLen>> 0, as variáveis da matriz têm um valor de string simbólico: "{...}"

      LETO_VARGETCACHED () ==> xValue [NIL]

 É uma forma otimizada de LETO_VARGET (...) e assim pode ser usado excelente em filtro
expressões. Retorna a * última alteração * LETO_VAR [SET | INCR | DECR] () variável *
valor *.
! Para ser usado com cuidado!
 Quando uma conexão específica definir uma variável, ela mudará o último valor
armazenado em cache.
 Se for usado em uma condição de filtro e o tipo de valor [string, numérico, ..] for
alterado,
 o filtro fica inválido!
 Se nenhuma variável foi definida antes por esta conexão, o valor NIL é retornado.
 Leto_VarGet [List] () não tem influência, então você pode consultar outras variáveis.
 Leto_VarDel () feito pelo * mesmo usuário * que configurou o cache da variável, irá
limpar o valor para NIL.
 O valor em cache é pessoal para um usuário / thread, então * outros * podem usar
simultaneamente a mesma variável,
 ou até mesmo excluí-lo, sem interferir no valor armazenado em cache pessoalmente.

 A ideia complicada é a seguinte: primeiro você cria no aplicativo cliente uma variável
com:
    'leto_VarSet ("MY_GROUP", "MY_VAR", xValue, LETO_VCREAT)'
 então você define uma condição de filtro para sua área de trabalho, por exemplo, para
um numérico:
    DbSetFilter (NIL, "field_var <leto_VarGetCached ()")
 Evite nomes 'ALIAS->' na condição de filtro para o modo: No_Save_WA = 0: eles são
inválidos.
 O único parâmetro que realmente importa é o valor do texto do filtro, o codeblock é
opcional.
 Nomes de funções solicitados SOMENTE na condição de filtro devem ser solicitados
explicitamente por:
 Instrução REQUEST a ser vinculada ao seu aplicativo.

Isso dá a condição do filtro no cliente muito! acesso rápido ao valor SEM perguntar ao
 servidor, também o servidor tem acesso muito rápido ao último valor armazenado em cache.
```

```
 A conexão que define o filtro pode alterar de forma interativa o conteúdo desta variável
 com um novo leto_Var [Set | Incr | Decr] () para atualizar a condição do filtro, pois
isso será definido
 um novo valor em cache.
 Dica: se houver milhões de registros e a condição do filtro reduzi-los a apenas alguns,
 pode aumentar o desempenho reduzindo a quantidade padrão de registros retornados com
 leto_setSkipBuffer () se não tantos [padrão: 10] registros com uma solicitação DbSkip
() necessária.


 Família de funções Leto_VarExpr * () lidando com memvars (PRIVATE / PUBLIC) em
EXPRession
cordas. A própria expressão <cExpressão> * não * deve ser uma expressão executável
válida,
 é basicamente o suficiente que um nome memvar apareça nele.
 Então, quando você quiser sincronizar variáveis com o servidor, faça:
    cExpr: = Leto_VarExprCreate ("Memvar1 [, MemvarX]", @aArr)
 Ligue repetidamente após cada ocasião de alteração do valor:
    Leto_VarExprSync (aArr)
 Limpe com:
    Leto_VarExprClear (aArr)

      LETO_VAREXPRTEST (cExpression) ==> lContainMemvar

 Retorna TRUE (.T.) Se <cExpressão> contiver memvars

      LETO_VAREXPRCREATE (cExpression [, @aLetoVar]) ==> cModifiedExpression

 Retorna uma <cExpressão> modificada, onde o nome de memvars são substituídos por uma
conexão
Leto_VarGet exclusivo (). Cada um desses Leto_Vars são criados durante a chamada com um
 Leto_VarCreate () contendo o valor do memvar, e tem o sinalizador LETO_VOWN,
 portanto, eles serão excluídos automaticamente com o fim do aplicativo.
 Se o segundo parâmetro opcional <@aLetoVar> for dado por referência (@), uma matriz 3-
dim com
 LetoVars para memvars relacionadas é atribuído à variável param, o que vai poupar um
extra
 chamada de Leto_VarExprVars ().

      LETO_VAREXPRVARS (cModifiedExpression [, lOnlyMemvar]) ==> aLetoVar

 Verifica o resultado de <cModifiedExpression> de Leto_VarExprCreate () para Leto_Vars e
retorna um
 Array 3-dim a ser usado com LetoVarExprSync () e Leto_VarExprClear ().
 O parâmetro opcional <lOnlyMemvar> tem o padrão TRUE (.T.): Então apenas LetoVars com um
 memvar relacionados são adicionados ao array de resultados.

      LETO_VAREXPRCLEAR (cModifiedExpression | aLetoVar [, lOnlyMemvar])
                                                              ==> lVarDeleted
 Verifica <cModifiedExpression> para LetoVars com o padrão TRUE (.T.), Que tem
um Memvar releatado. Para cada LetoVar encontrado, um Leto_VarDel () será executado.
 Em vez disso, a expressão de string pode ser o resultado <aLetoVar> de LetoVarExprVars
() usado.

      LETO_VAREXPRSYNC (aLetoVar [, fSyncLetoToMemvar]) ==> lVarSynced
 Para todos os LetoVars em <aLetoVar> um LetoVarSet () será executado, se o valor de
 memvar relacionado mudou entretanto (mas o tipo deve ser o mesmo).
 TRUE (.t.) É retornado apenas para o caso de uma sincronização ser executada, em todos
os outros casos:
 LetoVar igual a memvar, sem conexão ou vazio / inválido array aLetoVar FALSE (.F.).
 Alterar <fSyncLetoToMemvar> padrão TRUE (.T.) Para FALSE sincronizará da maneira
inversa:
 então, um valor LetoVar alterado, talvez no servidor por um UDF ou por outra conexão,
 ser aplicado ao memvar local.


      7.10 Chamando funções UDF no servidor
```

```
 Se uma função do lado do servidor modifica os dados da tabela, e especialmente quando o
ambiente de área de trabalho como
 ordem do índice ativo, condição do filtro, etc., é afetada pela ação, verifique também:
9. Funções do lado do servidor
 para limites e convenções para manter o cliente e o servidor em sincronia.

      LETO_UDF (cSeverFunc [, xParam1, ...]) ==> xResult
Esta função é chamada a partir do aplicativo cliente. A string <cServerFunc> pode
 opcional contém uma string de conexão do servidor, o mínimo é o nome da função udf:
 [// ip_address: port /] funcname
 Uma função <funcname> deve ser definida (SOLICITADA no momento do link) no servidor
LetoDBf.
 A função Udf pode retornar o resultado (qualquer tipo, exceto ponteiros) para o cliente.
 Exemplos de funções UDF 'carregáveis' estão em: tests / letoudf.prg

      LETO_UDF (cCodeBlock [, xParam1, ...]) ==> xResult
 Se <cCodeBlock> contém uma string de bloco de código válida, ela é avaliada no servidor.
 O comprimento máximo de <cCodeBlock> é 255 caracteres.
 Exemplo: leto_Udf ("{| x | UPPER (x)}", "to_upper")

      LETO_RPC (cSeverFunc, xParam1, ...) ==> NIL!
! Use com cuidado !, ele precisa de funções UDF bem projetadas.
 O nome da função é, na verdade, limitado a 20 caracteres.
 É um pouco semelhante a LETO_UDF, mas quando a função iniciar, ela NÃO terá áreas de
trabalho
 aberto: é como uma nova conexão, independente de quem a iniciou.
 Depois que tal trabalho for iniciado, nenhuma espera pelo resultado ocorrerá e apenas
um booleano (.T.)
é devolvido ao seu aplicativo se for iniciado com êxito. Você pode iniciar vários
trabalhos desta forma.
 Isso requer cuidado especial para não impedir que outras pessoas da rede acessem suas
áreas de trabalho,
 portanto, é aconselhável ser usado apenas no modo de servidor 3, também conhecido como
No_save_Wa = 1.

 Isso é extremamente interessante para tarefas em que você não precisa de feedback, por
exemplo, alguma 'limpeza'
 ou trabalhos de cálculo em segundo plano.
 Você pode escrever um resultado de algo em um DBF / TXT e recuperar o resultado dessa
forma.
 Ou você pode trocar informações com qualquer conexão com a ajuda da função LETO_VAR *
(), ou mesmo gerenciar
 a atividade de sua UDF com aquele sistema de variável de servidor - uma área deixada
para muitas idéias ...

 Mas essas funções UDF devem ser projetadas com * MUITO * cuidado, pois NÃO podem ser
interrompidas pelo
 conexão que os iniciou, então é melhor testá-los intensivamente de antemão com LETO_UDF.
 Normalmente, deve terminar por si só, como tal UDF 'sem cabeça' no servidor só pode ser
 interrompido pelo console de gerenciamento, se a UDF em execução for projetada para
verificar repetidamente:
leto_UDFmustQuit (). Se ele ignorar isso, é imparável até o desligamento do servidor ...

      LETO_UDFEXIST (cSeverFunc) ==> lExist
 leto_udfExist verifica a existência da função udf no servidor letodb.
 O parâmetro <cSeverFunc> é o mesmo que leto_udf ().

      LETO_PARSEREC (cRecBuf)
! obsoleto - removido!

      LETO_PARSERECORDS (cRecBuf)
! obsoleto - removido!
 veja novo LETO_DBEVAL () como alternativa poderosa, ou amostra de UDF_dbEval () em
tests / letoudf.prg
```

7.11 Funções para filtros de bitmap

Se letodb compilado com rdd BMDBFCDX / BMDBFNTX, então há suporte
as seguintes funções:

```
LBM_DbGetFilterArray () ==> aFilterRec
LBM_DbSetFilterArray (aFilterRec) ==> bFilterActive
LBM_DbSetFilterArrayAdd (aFilterRec) ==> bFilterActive
LBM_DbSetFilterArrayDel (aFilterRec) ==> bFilterActive
```

O objetivo e os parâmetros dessas funções são os mesmos que para o
funções BM _ * () correspondentes.
<bFilterActive> indica que há um filtro ativo no lado do servidor após a chamada.
Isso também pode ser verificado com LETO_ISFLTOPTIM () retornando TRUE.
Um filtro ativo pode ser limpo por um DbClearFilter () / SET FILTER TO comum.

```
LBM_DbSetFilter ([<xScope>], [<xScopeBottom>], [<cFilter>])
                                                    ==> nulo
```

Esta função define o filtro de bitmap pela ordem do índice atual e por condição,
definido nos parâmetros <xScope>, <xScopeBottom>, <cFilter>.
O registro atual após LBM_DbSetFilter () é o primeiro registro que satisfaz a condição
de filtro.

7.12 Funções Diversas

```
leto_Hash (cText) ==> nHashValue
```

Retorna um valor hash de 32 bits do <cText> fornecido, útil para uma pesquisa rápida
por uma string.
Dependendo de como o LetoDBF é compilado, o algoritmo de hashing MurMur3 [padrão] ou um
homebrew
algoritmo é usado. Pode haver colisões raras, significa que é possível que dois <cText>
resultando no mesmo <nHashValue>. Para ficar no lado seguro, sempre compare o texto
caractere por caractere,
depois que um valor hash válido for encontrado.

8. Utils

8.1 Utilitário de gerenciamento de servidor

Existem dois utilitários de gerenciamento, Windows GUI e todos os console do sistema
operacional.
As fontes estão no diretório utils / manage.

8.1.1 Todos os consoles do sistema operacional

Para todos os sistemas operacionais, inclusive Windows, existe um! NOVO ! utilitário de
console que pode ser encontrado em:
utils / manager / console.prg. Crie-o facilmente com apenas um:

```
    console hbmk2
```

O executável será encontrado posteriormente no diretório "bin".

O conjunto completo de parâmetros é: console [.exe] [IP [: porta]] [nome de usuário]
[senha]
O primeiro parâmetro pode ser definido em letodb.ini com a opção: <Server> para executá-
lo localmente no servidor.
Mas como este é um console remoto independente do sistema operacional, você pode
assistir o servidor de qualquer estação,
então você normalmente fornecerá o endereço IP como primeiro parâmetro.

! Observe a janela redimensionável, as navegações irão se esticar dinamicamente em um
determinado tamanho de tela.
As informações exibidas são atualizadas automaticamente, quanto mais tempo você não
digitar nenhuma tecla, maior será esta
o intervalo obterá, para interferir menos na atividade do servidor.

Se não for mais necessário, encerre-o e reinicie-o ocasionalmente. Terminará automático com desligamento
Servidor LetoDBf.
A navegação nas quatro navegações é feita com cliques do mouse / roda ou teclado (TAB, tecla de cursor)
Observe também este botão "Menu" no canto superior esquerdo da tela para outras ações, como eliminar conexões,
adicionar usuários ao sistema de autenticação. Para algumas ações, você deve rolar até o desejado
conexão, em seguida, para mudar para esse menu. Posicionado na conexão do console mostra um resumo em
as outras navegações, posicionadas em uma conexão específica, fornecem informações detalhadas sobre quais
WA é apenas selecionado e qual ordem de índice está ativa.

8.1.2 Console de gerenciamento de janela GUI

A ferramenta GUI * não mais mantida * é construída com a ajuda da biblioteca HwGUI, você precisa desse pacote
pronto compilado, então adapte no arquivo manage.hbp no topo o caminho para o diretório raiz do seu pacote HwGUI.
Em seguida, crie o executável com:

```
hbmk2 manage
```

O executável será encontrado posteriormente no diretório "bin".
Observe que esta ferramenta oferece muito menos recursos do que o console 'todos os sistemas operacionais'.

8.2 Uhura

Isso é para detecção automática do servidor, muito útil em redes com atribuição dinâmica Endereços IP.
Novo: disponível como opção 'embutida' no próprio servidor LetoDBf -> veja Leto_Detect (),
a seguir está a descrição da versão autônoma.

Construa o executável em utils / uhura / uhura.prg com a: hbmk2 uhura
O executável será encontrado posteriormente no diretório "bin".
Inicie na mesma máquina onde o servidor LetoDBf está rodando.
Linux! o usuário pode dar com o primeiro parâmetro um nome de interface como "eth1" na qual uhura irá escutar.
Para parar Uhura, chame: uhura [interface] STOP

Em seguida, procure em utils / uhura / detect.prg por exemplo de uso de função: detectServer ("letodb").
Ele envia uma solicitação de transmissão e retorna a resposta de Uhura com o endereço IP do
máquina com Uhura, também conhecido como endereço IP do servidor LetoDBf.
Tudo que você precisa para isso deve ser encontrado em detect.prg, você precisa vincular as funções auxiliares a
seu aplicativo, pois não integrei essa funcionalidade ao LetoDBf. [Pendência ?]

A chamada de: uhura help
exibirá informações sobre as interfaces de rede disponíveis nesta máquina.
Esta tela de ajuda também aparecerá em caso de problemas, por exemplo, ao tentar parar Uhura e ela
não está funcionando, ou em caso de nome de interface inválido ...

9. Funções do lado do servidor

!!! CHANGED: nUserStru obsoleto !!!
O primeiro parâmetro para essas funções UDF anteriormente era o tipo numérico, agora está obsoleto.
Como resultado, os lugares para seus parâmetros em suas funções definidas mais antigas sobem um lugar na ordem.

As funções REQUESTed podem ser executadas a partir do cliente pela função Leto_UDF (),
incluindo também funções definidas no arquivo letoudf.hrb, carregado com o início do servidor.

O aplicativo cliente não sabe o que a função UDF fez, então todas as mudanças no ambiente da área de trabalho
no servidor deve ser redefinido, porque o ambiente da área de trabalho do cliente e do servidor deve permanecer em sincronia.
Exemplo: o cliente define a ordem do índice <x> como ativo, na próxima etapa faz um DbSeek () - mas entre esses dois
solicita que uma UDF mudou no servidor o pedido ativo e não o reconfigurou -> falha.

* registro / arquivo bloqueado recentemente por UDF, mas não liberado: permanece definido até que todas as conexões fechem a tabela
* se o cliente tem algo bloqueado, mas UDF desbloqueia, pode levar mais tarde a um RTE: bloqueio ausente
O recomendado é usar as funções de bloqueio / desbloqueio explicadas abaixo, pois elas registram bloqueios * no lado do servidor *,
respeitando os diferentes modos de servidor - e desaparecem se esquecidos, quando esta conexão fecha a área de trabalho
* não permitido é deixar uma lista alterada de pedidos de índice - ou um pedido de índice ativo diferente
* não permitido é fechar qualquer mesa aberta - a nova mesa aberta deve ser fechada novamente, para que seja melhor usar
leto_DbUseArea (), pois respeita diferentes modos de servidor; leto_DbCloseArea () não fecha os não permitidos
* nunca mude qualquer SET (xxx) mais como <lDeleted> com as duas funções abaixo;
este e mais três SET-ting podem ser definidos * do lado do cliente *: _SET_SOFTSEEK, _SET_AUTOPEN, _SET_AUTORDER
* melhor ainda é restaurar o RECNO, feito pelas duas funções abaixo

Leto_SetEnv ([xTopScope, xScopeBottom, ncOrder, cFilter, lDeleted])
Leto_ClearEnv ()
Este par de funções de ajuda é para alterar de forma segura o ambiente da área de trabalho em uma função UDF,
e para restaurá-lo no final da função - eles se poupam para fazer manualmente por conta própria.
O cenário de uso mais fácil é chamar no início da UDF: Leto_SetEnv () e no final da função: Leto_ClearEnv ()
Os estados salvos antes da primeira mudança não serão substituídos por outra chamada, então
não faz mal chamar Leto_SetEnv () várias vezes para a mesma área de trabalho, e deve ser chamado para cada
área de trabalho com mudanças esperadas para:
<xTopScope> e <xScopeBottom> definem o escopo superior e / ou inferior para a ordem de índice ativa ou <ncOrder>,
onde é verificado que o tipo de <x ... Escopo> é válido para a ordem de índice desejada.
<ncOrder> fornecido sozinho (como numérico ou string) apenas altera a ordem do índice ativo
<cFilter> define uma expressão de filtro, string vazia ("") irá limpar um possível filtro ativo
<lDeleted> altera a configuração 'SET (_SET_DELETED)'.
NOVO: As funções podem ser usadas para várias áreas de trabalho, uma vez que armazenam e configuram / restauram para WA específico.
Leto_SetEnv () chamado sem nenhum argumento salva todas as configurações da área de trabalho ativa.
Nenhum argumento é mais necessário para leto_ClearEnv (), ele restaura todas as configurações salvas anteriormente
para toda a área de trabalho salva, então é suficiente chamar Leto_ClearEnv () uma vez no final da função UDF

leto_Alias(cClientAlias) ==> cServerAlias
Esta função é necessária principalmente para o modo 'No_Save_WA = 0', aqui para usar outras áreas de trabalho diferentes
do que aquele que estava ativo quando uma UDF foi inicialmente chamada.

```
 Esta função retorna o nome ALIAS usado no servidor para um determinado alias do lado do
cliente <cClientAlias>.
 O efeito colateral é que, se o nome ALIAS for válido, ele mudará a área de trabalho
selecionada ativa no servidor.
 O servidor ALIAS retornado pode então ser usado em operações RDD usuais.
 No modo servidor: 'No_Save_WA = 0', os nomes ALIAS no servidor e o nome ALIAS no
cliente são diferentes.
 Todas as áreas de trabalho usadas em uma UDF não estão disponíveis para outras conexões,
 desde que a UDF esteja funcionando.
 No modo servidor: 'No_Save_WA = 1', os nomes ALIAS no servidor e no cliente são iguais.
 Também neste modo, não há restrição exclusiva de uso da área de trabalho apenas para a
conexão UDF,
 e usar leto_Alias() se tornou de alguma forma obsoleto.

      leto_RecLock ([nRecord], [nSecs], [bAppend]) ==> lSuccess
  A função leto_Reclock () bloqueia o registro com o número <nRecord>, ou o registro
atual se nRecord for
  deixado vazio, para acesso de alteração de dados.
  O parâmetro <nSecs> está disponível apenas no modo de abertura de arquivo do servidor:
No_save_WA = 1.
  Com os <nSecs> opcionais fornecidos [pode ser decimal: 1,5], ele irá esperar pelo
sucesso se não
  sucesso imediato.
  Esses bloqueios são internos do servidor conhecidos para esta conexão, mas não são
visíveis para o aplicativo cliente.
  Se você anexar um registro em seu UDF, que é bloqueado após um anexo bem-sucedido,
você só precisa
  registre-o para sua conexão, feito pelo terceiro parâmetro definido como TRUE (.T.) ==
append lock.
  Esses bloqueios são removidos quando o aplicativo cliente fecha este WA.

      leto_RecUnLock ([nRecord]) ==> lWasLocked
  A função leto_RecUnlock desbloqueia um registro bloqueado com o número <nRecord>.
  Nenhum <nRecord> fornecido significa o registro ativo.
  Se o registro não estava bloqueado por R, FALSE (.F.) É retornado.

      leto_RecLockList (aRecNo) ==> lSuccess
  A função leto_ReclockList bloqueou registros com número na matriz <aRecNo>.
  Se algum registro não estiver bloqueado, todos os registros serão desbloqueados e a
função retornará
  .F. resultado.
  Esta função pode ser usada no servidor a partir do módulo letoodf.prg ou a partir de
  cliente por uma chamada leto_UDF ("leto_RecLockList", aRecNo).

      leto_TableLock ([nSecs]) ==> lSuccess
      leto_TableUnLock () ==> lWasLocked
  [obsoleto: não há mais parâmetros <nFlags> disponíveis]
  Isso bloqueará toda a tabela DBF, não apenas um único registro, para acesso de
alteração de dados.
  O parâmetro <nSecs> está disponível apenas no modo de abertura de arquivo do servidor:
No_save_WA = 1.
  Com os <nSecs> opcionais fornecidos [pode ser decimal: 1,5], ele irá esperar pelo
sucesso se não
  sucesso imediato.
  Se a tabela não estava bloqueada por F, FALSE (.F.) É retornado.

      leto_SelectArea (nAreaId) ==> lSuccess

      leto_Select ([ncClientAlias]) ==> nWorkareaID
 Seleciona a área de trabalho fornecida pelo ID da área de trabalho ou ALIAS (se válido)
 Para o modo de servidor "NO_SAVE_WA = 0" solicita a área de trabalho destacada, que
pode ser impedida por outra
 conexão realmente usando isso, como neste modo de servidor, um WA pode ser usado apenas
uma vez por todas as conexões.

      leto_AreaID ([cAlias]) ==> nAreaId
 Função retornar workarea-ID interno da área de trabalho atual / por ALIAS,
```

```
 0 se não for encontrado

 [As quatro funções a seguir foram renomeadas para LetoDBf - para serem as mesmas do
Harbour,
   mas com prefixo: 'leto_']

      leto_DbUseArea ([cDriver], cFileName, [cAlias, lShared, lReadOnly, cdp])
                                                              ==> nAreaId
 Parâmetros ordenam como DbUseArea (), EXCETO o primeiro omitido: a tabela está sempre
em uma nova área de trabalho,
 você NÃO pode pré-selecionar o Workarea-ID.
 Se não for fornecido, cAlias é criado a partir de nome do arquivo, cDriver é a última
configuração usada,
 lShared é o padrão de _SET_EXCLUSIVE do cliente LetoDBf !,
lReadonly é o padrão .F. e página de código são as configurações usadas no lado do
cliente.

      leto_DbCreate (cFilename, aStruct, [cRDD], [lKeepOpen], [cAlias], [cCodePage])
                                                              ==> l Sucesso
 Cria um novo banco de dados, alguns parâmetros são opcionais - cFilename pode ser
prefixado com 'mem:'.
 cRDD pode ser um dos seguintes: DBFNTX, DBFCDX, DBFFPT, SIXCDX, DBFNSX - se não for
fornecido o último RDD usado.
 O padrão lKeepOpen é fechar o arquivo após a criação.
 cAlias só é necessário se a nova tabela criada permanecer aberta.
 cCodepage será o último usado - melhor deixar sempre vazio, especialmente se você não
souber exatamente
 a causa pela qual você deseja defini-lo.

      leto_OrdListAdd (cBagName [, ncAreaID]) ==> lSuccess
 Recomenda-se usar o segundo parâmetro ncAreaID, melhor como cALIAS ou como número da
área de trabalho no
lado seguro. LetoDBf adicionará todas as chaves de índice em um arquivo de índice
[multitags] de uma vez.
 nAreaID é opcional, se não for fornecido, a área de trabalho real selecionada é usada.

      leto_OrdCreate ([ncAreaID], cBagName, cKey, cTagName,
                      lUnique, cFor, cWhile, lAll, nRecNo,
                      nPróximo, lRest, lDesc, lCustom, lAdditive)
                                                              ==> l Sucesso
 Recomenda-se usar o primeiro parâmetro ncAreaID, melhor como cALIAS ou como número da
área de trabalho.
 Mas o ID no primeiro lugar também pode ser omitido, então o pedido será criado para
esse WA.
 cBagName é em qualquer caso o segundo parâmetro, cKey a expressão de índice como string.
 Para os outros valores, consulte a documentação do Harbour para OrdCondSet ().

      leto_DbCloseArea ([ncAreaID]) ==> l Êxito
 Feche o ativo ou por numérico ou por string ALIAS dada área de trabalho, se WA foi
aberto pelas funções acima.

      LETO_DBEVAL (<cbBlock>, [<cbFor>], [<cbWhile>], [nPróximo], [nRegistro], [lRest] ,;
                   [<lResultArr>], [<lNeedLock>], [<lDescend>], <lStay>)
                                                              ==> lSucesso | aResults |
xValue

 * procure a explicação para parâmetros estendidos versus DBEval () no capítulo: 7.3
Funções adicionais *

Esta função UDF é do mesmo tipo que DbEval () de Harbour. Os três codeblocks podem ser
fornecidos
 em vez disso, como um codeBlock dentro de uma função UDF ou como parâmetros de string
quando chamado 'diretamente'
 de um cliente por leto_udf ("leto_DbEval", "cBlock", ...).
 Se nenhum cbBlock for fornecido (nem como CB nem como string vazia), você obtém o
conteúdo bruto de um todo
 registro no formato LetoDBf, senão o resultado do cbBlock avaliado para cada registro
convertido em string.
```

```
 Esta é uma funcionalidade muito difícil, portanto, deve ser usada com algum cuidado.
 cbFor significa uma condição para ser válida, senão o registro não é processado e
pulado [mas conta para nNext].
cbEnquanto significa uma expressão para parar quando resulta em falso (.F.). Se vazio, o
padrão é: '.NOT. EOF () '.
 Ambos cbFor e cbWhile devem retornar valores lógicos (booleanos).
 Se nNext for 0, todos os registros até EOF () [ou cbWhileareprocessados, caso contrário,
 a quantidade fornecida.
 lRest == verdadeiro (.T.) começa no registro real, lRest == falso (.F.) do primeiro,
primeiro registro.

 As funções leto_DbUseArea, leto_OrdListAdd, leto_DbCreate (), leto_OrdCreate,
leto_DbCloseArea,
 leto_RecLock, destina-se ao uso em funções UDF em vez de funções rdd:
 dbUseArea, OrdListAdd, OrdCreate, dbCloseArea, RLock, dbUnlock

      leto_UDFMustQuit () ==> lTrue
 Se o monitor do console sair de um thread UDF ou RPC no servidor, ele definirá uma
variável interna para este
fio. Esta função relata sobre isso, então termina o shell.
Exemplo: DO WHILE .NOT. leto_UDFMustQuit (); faça alguma coisa; ENDDO; RETORNA
 Especialmente útil para threads iniciados com Leto_RPC ().

      leto_WUsLog (cText)
 Grave <cText> no arquivo de log específico da conexão: letodbf_xx.log.
 ["WUsLog" diz: W-rite Us-er Log]

      leto_WrLog (cText)
 Grave <cText> no arquivo de log do servidor global: letodbf.log.

 As seguintes funções são as mesmas do cliente LetoDBf, agora também sem nUserStru.
 Nota: xValue será 'NIL' no caso de cGroupName / cVarName não encontrado.

      LETO_VARSET (cGroupName, cVarName, xValue [, nFlags) ==> xValue
      LETO_VARGET (cGroupName, cVarName) ==> xValue
      LETO_VARINCR (cGroupName, cVarName [nFlags], [nVal]) ==> nValue
      LETO_VARDECR (cGroupName, cVarName [nFlags], [nVal]) ==> nValue
      LETO_VARDEL (cGroupName, cVarName) ==> lSuccess
      LETO_VARGETLIST ([cGroupName, [lValue]]) ==> aList

 especial:
      LETO_VARGETCACHED () ==> xValue

! veja as explicações para o propósito na função do lado do cliente leto_VarGetCached ()!



      10. Abreviações e observações

 Talvez neste texto use abreviações:

    CP CodePage; um conjunto de caracteres que representam caracteres especiais
'normais' e nacionais.
          Às vezes, também abreviado com CDP.
    Formato de data DF; um padrão de como um valor "DATE" é apresentado, onde as letras:
          d == dia, m == mês, y == ano mais caracteres extras, por exemplo, "dd.mm/aaaa"
    Endereço de protocolo de Internet IP; Endereço IPv4
    Chamada de procedimento remoto RPC; execução de um procedimento não na máquina
cliente, mas na
          máquina onde o LetoDBf é executado.
    Erro de tempo de execução RTE; pode ocorrer se o desenvolvedor do aplicativo tentar
algo melhor
          verifiquei antes de fazer isso. Ou quando algo realmente inesperado aconteceu.
    Protocolo de controle de transmissão TCP; baseado em pacotes; o que o ET perdeu
    Função definida pelo usuário UDF; em oposição aos comandos Harbour definidos pelo
usuário.
          Normalmente contendo comandos Harbour.
```

```
    Área de trabalho WA; um ambiente lógico que representa uma tabela de banco de dados.
          Normalmente, eles têm um nome recuperável com ALIAS ()

 Aqui estão algumas observações instantâneas, que aguardam outro lugar para serem
explicadas.
   # Não é possível, sobrescrever um DBF usado, também conhecido como DbCreate () irá
falhar com RTE
     se já aberto por outro usuário / conexão de outro segmento.
   # Uma tabela de banco de dados pode ser usada apenas com a mesma página de código
para todas as outras conexões.
   # Nomes ALIAS () para tabelas HbMemIO (estes com prefixo "mem:") são sempre os mesmos
     como o nome do próprio DBF. Não é possível dar outro ALIAS, então você
     pode abrir tal mesa apenas uma vez por conexão.
   # Você não pode usar * em sua UDF * no modo No_Save_WA = 0 !! [a única exceção] Nomes
ALIAS
    "Exxxxx", onde "xxxxx" é um valor numérico, também conhecido como: "E123" :-)
   # O uso de pedidos de índice temporário [criado no caminho temporário do sistema
operacional do servidor] só é possível em
     modo: NO_Save_Wa = 1.
   # valor numérico máximo para um campo: "N", 20, 0 é: +/- 92233720368547758 07
     um a mais (ou menos) e o arredondamento ocorrerá com zeros à direita



      11 Resolução de Problemas

 + servidor não iniciará
   -> verifique <DataPath> em 'letodb.ini' - procure o log do servidor 'letodbf.log'
 + tabelas não podem ser acessadas
   * sistema de arquivos do sistema operacional que diferencia maiúsculas de minúsculas
e em sua fonte escrita provavelmente em maiúsculas e minúsculas
      -> renomeie todos os arquivos para minúsculas, use a opção de configuração:
<Lower_Path>
   * Direitos de acesso ao sistema operacional
      -> definir <Server_User> -ou- <Server_UID> [+ <Server_GID>]
 + o armazenamento do servidor está barulhento, desempenho muito ruim na atualização de
dados
   -> * não * use a opção <hardcommit> em letodb.ini

 + rede de largura de banda baixa disponível apenas, o que melhorar
   -> otimizar o desempenho
      Se linhas específicas de LetoDbf devem ser adicionadas ao código-fonte, é melhor
colocá-las entre:
      #ifdef LETO_DBF / * ou: RDDLETO_CH_, ambos definidos em "rddleto.ch" * /
         ...
      #endif / * para manter a fonte portátil para uso sem LetoDBf * /
      Se você precisar desativar as linhas de origem existentes para LetoDBf, use:
      #ifndef LETO_DBF / * lê: se n [ot] definido * /
         ...
      #fim se

      * remova DbCommit [All] (), ou pelo menos desative-os para LetoDBf
        Isso é necessário apenas para informar o servidor sobre dados atualizados para
outros usuários
        * sem * desbloquear registro e * sem * SKIP / GOTO para outro registro.
        Em todos os outros casos, isso causa tráfego de rede extra desnecessário.
      * tornar as expressões <filter> e <relation> avaliáveis no lado do servidor,
        [capítulo: 5.2 Filtros e relações]
        Importante é o uso do parâmetro * <cEpression> * nas funções:
        DbSetFilter ([<bCodeBlock>], <cEpression>) e
        DbSetRelation (<cnAlias>, <bCodeBlock>, cEpression>)
      * aumentar o tamanho de Leto_SetSkipbuffer () para ocasiões em que pular muito,
        [por exemplo, o clássico: "DO WHILE! EOF (), DbSkip (), ..."]
        É contraproducente! quando apenas alguns registros precisam ser ignorados,
        portanto, é melhor redefinir para o valor padrão [10 | 21] logo após as mudanças.
        Alterar o padrão do servidor é possível com a opção de configuração:
Cache_Records
```

```
      * aumentar o tempo limite de registros em cache [o padrão é 1 segundo] com
configuração
        DBI_BUFREFRESHTIME, ou com: LETO_CONNECT (,,,,, <nBufRefreshTime>)
        Não exagere !, tente 5 a 10 segundos [valores dados como x /! 100! s]
      * precisa determinar se um campo MEMO está 'vazio ou não vazio', sem a necessidade
de
        o próprio conteúdo (ou seja, para exibir em um navegador "MEMO" ou "memo" se
estiver vazio),
        use Leto_MemoIsEmpty (): não precisa de nenhum pedido extra para o servidor, o
cliente já sabe disso
      * tente ativar a compressão de tráfego de rede com Leto_ToggleZip ()
      * verifique se você pode usar TRANSACTIONs,
      * contrate-me! ;-)
 + dados de alteração frequente devem ser visualizados
   -> diminuir o tempo limite do cache definido com o cache DBI_BUFREFRESHTIME,
      ou o LETO_CONNECT (,,,,, <nBufRefreshTime>)
   -> talvez ative até mesmo DBI_AUTOREFRESH


-------------

      A. Alguns internos

      Algumas explicações, o que acontece 'por baixo do capô'.
 O servidor é desenvolvido Multi-Threaded, o que é "um pouco" comparável à execução de
várias instâncias
ou muitos programas em um computador. Isso usará todas as CPUs do seu servidor e
abrangerá a carga
 eles.
 Lá é executado um thread principal, responsável pela solicitação de conexão de entrada,
que iniciará a cada
conexão com um usuário um novo tópico. Toda a comunicação para uma conexão é feita neste
novo
fio. Durante a inicialização do servidor também são iniciados mais dois threads: um é
como o thread principal
 responsável por lidar com a seqüência de comunicação inicial para o segundo soquete (*)
para uma conexão.
Isso só acontecerá para aplicativos vinculados multithread definindo a opção hbmk2; '-mt'
 Se habilitado, um outro thread verifica periodicamente se há conexões mortas e irá
fechá-las nesse caso.
 Isso pode ser ajustado definindo o intervalo de tempo para verificação como zero [o
padrão é: Zombie_Check = 0]
 em letodb.ini.

 (*) Segundo socket: esta segunda porta para o servidor é usada também para a
verificação de 'conexão morta * mencionada.
 E usado para um grupo de solicitações do cliente ao servidor: essas solicitações têm em
comum, que você regularmente
receba apenas um 'ACK' == 'concluído com sucesso' do servidor. Este 'ACK' NÃO será
enviado, e apenas no caso
 de um erro, um 'NO-ACK' negativo será enviado por meio dessa segunda porta, então a
chamei de 'socket de erro'.
 usado para tráfego de rede em tempo real sob demanda:lizar registros no
 para o 'ACK' positivo, pode prosseguir imediatamente e também o tráfego de rede é
evitado.
 Mas o cliente não perderá tal 'NO-ACK' negativo: nesse caso, uma mensagem de erro de
tempo de execução
 pop up 'atrasado' com o conhecido sistema de tratamento de erros Harbour no lado do
cliente / usuário.


 Ou em minhas outras palavras:
 (https://groups.google.com/forum/#!topic/harbour-users/q_ZqmOB6Sns)
 No modelo clássico 'cliente-servidor', o cliente envia uma solicitação e recebe uma
resposta do servidor
- a qualquer momento. Cada uma dessas solicitações / respostas precisa de um 'pacote'
inteiro para ser enviado, até mesmo menos
```

```
 preenchido com apenas alguns bytes em vez de no máximo possível ~ 1500 bytes.
 A quantidade de pacotes por intervalo de tempo é limitada, portanto, apesar da
comunicação em velocidade máxima, você vê menos
 tráfego de rede.

 Existe um grupo de requisições, para as quais o servidor envia um 'ACK' aka: concluído!
 Isso atinge o limite do pacote e, além disso, o cliente tem que esperar por ele.
 Melhor exemplo: atualizar dados, também conhecido como: REPLACE .. WITH .., leva a
bilhões de patos ACK ACK ACK para aguardar
após cada solicitação de atualização. :-) Então eu queria poupar o ACK, mas * não perder
* o muito raro NÃO! -ACK.
 (exemplo no caso: quando o registro não está bloqueado)

 Solução: o cliente abre um segundo soquete para o servidor, no qual um segundo thread
está aguardando a entrada
 info (inverter o primeiro soquete, onde o servidor está esperando pela entrada)
 Assim o servidor não envia um ACK, mas apenas em caso de problema um NO! -ACK para este
segundo socket.
 O segundo encadeamento no cliente o recebe e prepara um objeto de erro 'global'
protegido por mutex.
 Isso é verificado pelo primeiro thread para ver se não está vazio, após cada
solicitação de envio (e ao entrar no modo inativo).
 No caso de um objeto de erro preenchido, o primeiro thread principal permite que o
sistema de erro em tempo de execução apareça com isso.
 Chamei isso de erro 'atrasado' :-)

 Soma: isso leva a um aumento significativo de desempenho, pois o cliente pode enviar
todas essas 'atualizações de dados'
 solicitações (e alguma outra solicitação como desbloqueio, etc), uma após a outra sem
demora para o ACK,
bom preencher a fila no servidor. Caso o cliente não tenha capacidade MultiThread,
nenhum segundo soquete
 é aberto, o servidor age 'clássico' para esta conexão, faz ACK ACK ACK :-)


 Solicitando e desanexando áreas de trabalho: muitos outros servidores, também HbNetIO,
abrirão uma tabela DBF novamente para cada
conexão após o comando do usuário: DbUseArea (). Desanexar uma área de trabalho aberta
significa que ela está no lado do servidor 'fechada'
 para essa conexão e colocados em um pool de 'áreas de trabalho separadas'.
 Se a conexão precisar da área de trabalho novamente para a próxima ação, ela a
solicitará fora do pool de
áreas de trabalho. Imediatamente após cada ação, o WA é desconectado novamente.
 Se outra conexão deseja abrir a mesma tabela DBF, em vez disso, solicita-a fora do pool
de WA desanexado,
 e o desanexa novamente após a conclusão da ação.
 Neste tipo, uma tabela DBF é aberta apenas uma vez pelo servidor, e por desanexação /
solicitação, em seguida, trocada
entre várias conexões. Isso tem uma vantagem de desempenho significativa para o modo
padrão 1 explicado abaixo.
 A desvantagem é: enquanto uma conexão / UDF precisar desse WA para ação, nenhuma outra
conexão poderá usá-lo.
 Além disso, é para uma conexão apenas uma única área de trabalho ativa: isso é
importante para UDF que precisa de mais
 de um WA.

 Modos de abertura de arquivo: LetoDBf conhece três (4) modos, escolha o apropriado para
suas necessidades.
 # 1 # Share_Tables = 0, No_Save_WA = 0
 O servidor usa exclusivamente os arquivos DBF, de modo que nenhum software de terceiros
(não usuário LetoDBf) pode acessar
Tabelas DBF abertas pelo servidor LetoDBf. Este é o modo de servidor mais rápido.
 As áreas de trabalho são trocadas entre as conexões usando a técnica de desconexão /
solicitação explicada acima.
 # 2 # Share_Tables = 1, No_Save_WA = 0
 O segundo modo é igual, exceto que os DBFs estão neste modo abertos de acordo com a
primeira solicitação de conexão.
```

Se o primeiro usuário de uma tabela DBF abri-la com o atributo compartilhado para DbUseArea (), outras conexões também podem
use-o. Se a primeira conexão abrir a tabela DBF exclusiva, nenhum outro poderá usar essa área de trabalho simultaneamente.
Abrir uma tabela DBF em modo compartilhado dá a usuários terceiros / não LetoDBf a possibilidade de trabalhar simultaneamente
com tabelas DBF abertas pelo servidor LetoDBf. Este modo é um pouco mais lento, mas quando o acesso de simultaneidade é necessário
o caminho para seguir.
3 # No_Save_WA = 1 [mais: Share_Tables = 1; padrão se não for definido explicitamente]
A opção Share_Tables indica que outro software como um servidor LetoDBf está acessando as tabelas DBF,
então o bloqueio físico do arquivo / registro acontecerá para coordenar o compartilhamento - caso contrário, os bloqueios são apenas internos / lógicos
e pode ser notado apenas por esse servidor LetoDBf. Espera-se que o bloqueio lógico forneça um desempenho ligeiramente melhor.
As áreas de trabalho NÃO são trocadas entre as conexões usando a técnica de desconexão / solicitação.
No lado do servidor, cada DBF é aberto exatamente com o mesmo ID de área de trabalho e nome ALIAS que no lado do cliente, e
para cada próxima conexão, mais uma vez - compartilhada ou exclusiva, como o cliente solicitou a tabela DBF.
Neste modo, todas as relações e filtros no lado do cliente estão ativos no lado do servidor.
Isso habilitará as expressões de índice ou filtro em FIELDs de uma área de trabalho relacionada.
Este modo é geralmente o modo mais lento, mas com ação de filtro rápida. Tem grandes vantagens para longa duração
UDF funciona e pode ter vantagens de desempenho para muitos usuários simultâneos acessando a mesma tabela,
como neste modo, uma conexão não deve esperar por uma área de trabalho separada de outro encadeamento.

Do ponto de vista de desempenho, o modo # 1 # é o mais rápido, # 3 # entre os dois, e o mais lento é o # 2 #.
Mas espere apenas alguns% de diferença de desempenho entre os modos, já que o fator mais limitante é a rede TCP / IP
em si: a contagem de pacotes de dados em um período de tempo é limitada, e cada solicitação ao servidor e resposta dele, é
um pacote de dados inteiro, muitas vezes preenchido com muito menos conteúdo do que um único pacote pode ter (~ 1500 bytes).

...

B. Links para terceiros etc.

MurMurHash versão 3, resultados de hash de 32 bits, domínio público, retirado de:
https://github.com/aappleby/smhasher
LetoDBF distribui apenas PMurHash. [C | h]

Algoritmo de compressão LZ4, licença BSD, compressão extremamente rápida e descompressão ainda mais rápida,
usado para tráfego de rede em tempo real sob demanda:
https://github.com/lz4/lz4
LetoDBF distribui apenas o diretório 'lib' com a licença BSD desse contrib.

C. Nota

! O mais importante: SEM GARANTIA minha sobre nada - decida você mesmo se o LetoDBf atende às suas necessidades!
Até agora VOCÊ é o único responsável por tudo o que acontece com e ao redor de LetoDBf em suas casas.

Para o caso de você desejar confiável * rápido e pessoal *! apoio de mim, devemos falar sobre 'doação'.

```
desejamos toda a diversão possível!
elch
```