Módulo Table

O módulo **table** estende as funcionalidades padrão de manipulação de tabelas do Lua, fornecendo funções adicionais úteis para processamento de dados. Este módulo complementa as funções nativas da tabela `table` do Lua, adicionando operações comuns que não estão disponíveis na biblioteca padrão.

Características principais:

- Extensão da tabela `table` nativa do Lua
- Funções otimizadas para performance

Funções Disponíveis

1. `table.contains(tabela, valor)`

Verifica se um valor específico está presente em uma tabela Lua.

Parâmetros:

- **tabela** (tabela): A tabela a ser verificada (pode ser array ou tabela associativa)
- **valor** (qualquer tipo Lua): O valor a ser procurado na tabela

Retorno:

- **boolean**: `true` se o valor for encontrado na tabela, `false` caso contrário

Comportamento:

1. Percorre todos os pares chave-valor da tabela usando `pairs()`
2. Compara cada valor com o valor procurado usando igualdade estrita (`==`)
3. Retorna `true` na primeira ocorrência encontrada
4. Retorna `false` se percorrer toda a tabela sem encontrar o valor
5. Funciona com tabelas indexadas numericamente (arrays) e tabelas associativas

Exemplo de Uso:

```
-- Verificar se um valor existe em um array
```

```lua
local frutas = {"maçã", "banana", "laranja", "uva"}
local tem_banana = table.contains(frutas, "banana")
print("Tem banana?", tem_banana)  -- true

local tem_morango = table.contains(frutas, "morango")
print("Tem morango?", tem_morango)  -- false

-- Verificar em tabela associativa
local configuracoes = {
    timeout = 30,
    retries = 3,
    debug = false,
    hostname = "servidor.local"
}

local tem_timeout = table.contains(configuracoes, 30)
print("Tem valor 30?", tem_timeout)  -- true

local tem_true = table.contains(configuracoes, true)
print("Tem valor true?", tem_true)  -- false

-- Verificar tipos complexos
local usuarios = {
    {id = 1, nome = "Alice", ativo = true},
    {id = 2, nome = "Bob", ativo = false},
    {id = 3, nome = "Carol", ativo = true}
}

-- Para tipos complexos, precisa ser a mesma referência
local usuario_bob = usuarios[2]
local encontrou_bob = table.contains(usuarios, usuario_bob)
print("Encontrou Bob?", encontrou_bob)  -- true

-- Esta busca retornará false porque é um novo objeto
local encontrou_novo_bob = table.contains(usuarios, {id = 2, nome = "Bob", ativo = false})
print("Encontrou novo Bob?", encontrou_novo_bob)  -- false
```

2. `table.slice(tabela, primeiro, último)`

Extrai uma fatia (subarray) de uma tabela indexada numericamente (array).

Parâmetros:

- **tabela** (tabela): O array Lua do qual extrair a fatia
- **primeiro** (número, opcional): Índice inicial da fatia (padrão: 1)
- **último** (número, opcional): Índice final da fatia (padrão: comprimento da tabela)

Retorno:

- **tabela**: Novo array contendo os elementos da fatia especificada

Comportamento:

1. Cria uma nova tabela contendo os elementos do índice `primeiro` até `último` (inclusive)
2. Se `primeiro` for omitido ou `nil`, começa do primeiro elemento (índice 1)
3. Se `último` for omitido ou `nil`, vai até o último elemento da tabela
4. Se `primeiro` for maior que `último`, a função ainda itera mas os valores serão `nil` para índices inexistentes
5. Não realiza verificação de limites - índices fora do intervalo da tabela resultarão em valores `nil`
6. Preserva a ordem original dos elementos

Exemplo de Uso:

```
-- Extrair parte de um array
local numeros = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

-- Fatia do índice 3 ao 7
local fatia1 = table.slice(numeros, 3, 7)
-- fatia1 = {30, 40, 50, 60, 70}

-- Fatia do início até o índice 5
local fatia2 = table.slice(numeros, nil, 5)
-- fatia2 = {10, 20, 30, 40, 50}

-- Fatia do índice 8 até o final
local fatia3 = table.slice(numeros, 8)
-- fatia3 = {80, 90, 100}

-- Fatia com índices fora dos limites
local fatia4 = table.slice(numeros, -2, 15)
-- fatia4 = {nil, nil, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, nil, nil, nil, nil, nil} (não
```

```
trata limites)

-- Fatia com primeiro > último
local fatia5 = table.slice(numeros, 5, 3)
-- fatia5 = {50, 40, 30} (itera em ordem decrescente)

-- Processamento de dados em lotes
local function processar_em_lotes(dados, tamanho_lote)
    local lotes = {}
    local total = #dados

    for i = 1, total, tamanho_lote do
        local fim = math.min(i + tamanho_lote - 1, total)
        local lote = table.slice(dados, i, fim)
        table.insert(lotes, lote)
    end

    return lotes
end

local dados_grandes = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
local lotes = processar_em_lotes(dados_grandes, 5)
-- lotes = {{1,2,3,4,5}, {6,7,8,9,10}, {11,12,13,14,15}}
```

Revision #1
Created 3 February 2026 16:46:40 by Marc
Updated 3 February 2026 16:46:50 by Marc