REFERÊNCIA DE COMANDOS PARA SCRIPTS

Módulo Cache

O módulo **cache** fornece um sistema de cache distribuído para scripts Lua. O cache suporta operações de leitura/escrita com expiração automática (TTL) e mecanismos de sincronização para operações concorrentes.

O sistema de cache é especialmente útil para:

- Armazenar resultados de operações caras (como requisições HTTP, consultas de banco de dados)
- Evitar chamadas redundantes a APIs externas
- Sincronizar múltiplos workers que tentam acessar o mesmo recurso simultaneamente
- Compartilhar dados entre diferentes scripts Lua em execução

Funções Disponíveis

`cache.put(key, value, ttl)`

Armazena um valor no cache com um tempo de vida específico.

Parâmetros:

- `key` (string): Chave única para identificar o valor no cache
- `value` (qualquer tipo Lua): Valor a ser armazenado (pode ser string, número, tabela, booleano, etc.)
- `ttl` (number): Time To Live em segundos - quanto tempo o valor permanecerá no cache

Valor de retorno:

- `nil` em caso de sucesso
- Lança erro em caso de falha

Exemplo:

```
-- Armazenar um resultado de API por 5 minutos (300 segundos)
local resultado_api = {status = "ativo", usuarios = 150}
cache.put("status_sistema", resultado_api, 300)
```

```
-- Armazenar uma string simples por 1 hora
cache.put("ultima_atualizacao", "2026-01-15T10:30:00Z", 3600)

-- Armazenar um número
cache.put("contador_requisicoes", 42, 60)
```

`cache.get(key)`

Recupera um valor do cache.

Parâmetros:

- `key` (string): Chave do valor a ser recuperado

Valor de retorno:

- O valor armazenado se existir e não estiver expirado
- `nil` se a chave não existir ou o valor tiver expirado

Exemplo:

```
-- Recuperar um valor do cache
local status = cache.get("status_sistema")

if status then
    print("Status do sistema:", status.status)
    print("Usuários ativos:", status.usuarios)
else
    print("Cache expirado ou não encontrado")
    -- Fazer uma nova requisição para obter os dados
end

-- Verificar se um valor existe
local ultima_atualizacao = cache.get("ultima_atualizacao")
if ultima_atualizacao then
    print("Última atualização:", ultima_atualizacao)
end
```

`cache.mark_pending(key)`

Marca uma chave como "pendente". Esta função é usada para sincronizar múltiplos workers que tentam calcular o mesmo valor simultaneamente.

Parâmetros:

- `key` (string): Chave a ser marcada como pendente

Valor de retorno:

- `nil` em caso de sucesso
- Lança erro em caso de falha

Comportamento:

- Quando uma chave é marcada como pendente, qualquer chamada subsequente a `cache.get()` para essa chave irá bloquear até que um valor seja armazenado com `cache.put()`
- O estado "pendente" expira automaticamente após 5 minutos (300 segundos)
- Útil para evitar que múltiplos workers recalculem o mesmo valor caro simultaneamente

Exemplo:

```
-- Padrão típico para evitar cache stampede
local function obter_dados_caros(chave)
    -- Primeiro tenta obter do cache
    local dados = cache.get(chave)

    if dados then
        return dados
    end

    -- Se não encontrou, marca como pendente
    local sucesso, erro = pcall(cache.mark_pending, chave)

    if not sucesso then
        -- Outro worker já marcou como pendente, espera pelo resultado
        dados = cache.get(chave)  -- Esta chamada irá bloquear até o valor estar
disponível
        if dados then
            return dados
        end
    end
```

```
    -- Este worker é responsável por calcular o valor
    -- ... cálculo caro aqui ...
    local resultado = calcular_dados_caros()

    -- Armazena no cache para outros workers
    cache.put(chave, resultado, 300)

    return resultado
end
```

`cache.delete(key)`

Remove uma chave do cache imediatamente.

Parâmetros:

- `key` (string): Chave a ser removida

Valor de retorno:

- `nil` em caso de sucesso
- Lança erro em caso de falha

Exemplo:

```
-- Remover um valor específico
cache.delete("dados_expirados")

-- Limpar cache relacionado a um usuário
cache.delete("usuario_123_perfil")
cache.delete("usuario_123_preferencias")
cache.delete("usuario_123_historico")
```

Informações Adicionais

Sincronização de Workers

O mecanismo de "pendente" permite que múltiplos workers sincronizem o cálculo de valores caros:

1. Primeiro worker marca a chave como pendente e calcula o valor
2. Outros workers que tentam acessar a mesma chave aguardam o resultado
3. Quando o primeiro worker termina, armazena o valor e todos os workers recebem

Tipos de Dados Suportados

O cache pode armazenar qualquer tipo de valor Lua suportado pelo sistema de serialização do Monsta, incluindo:

- Strings
- Números
- Booleanos
- Tabelas
- Valores nulos

Considerações de Performance

1. **Acesso rápido**: Operações de cache são muito mais rápidas que recalcular valores ou fazer requisições externas
2. **Memória**: O cache é mantido em memória, então valores muito grandes podem impactar a performance
3. **Concorrência**: O sistema é thread-safe e suporta acesso concorrente de múltiplos workers
4. **Network**: O cache é local ao processo, não há overhead de rede

Limitações

1. **Volátil**: Dados são perdidos se o processo for reiniciado
2. **Memória limitada**: Não use para armazenar grandes volumes de dados
3. **Distribuição**: Este é um cache local, não distribuído entre múltiplos servidores

Melhores Práticas

1. **TTL adequado**: Use TTLs apropriados para o tipo de dado (curto para dados dinâmicos, longo para dados estáticos)
2. **Chaves descritivas**: Use nomes de chave que descrevam claramente o conteúdo
3. **Namespace**: Use prefixos para organizar chaves (ex: `api_`, `user_`, `config_`)
4. **Fallback**: Sempre tenha um fallback caso o cache esteja vazio ou expirado

5. **Invalidação**: Use `delete()` para invalidar cache quando dados mudarem

Módulo DNS

O módulo **DNS** fornece funções para testar a performance e disponibilidade de servidores DNS. Este módulo é útil para monitoramento de infraestrutura de DNS, diagnóstico de problemas de resolução de nomes e medição de latência em consultas DNS.

Funções Disponíveis

1. `dns.ping(server, domain)`

Realiza uma consulta DNS a um servidor específico e mede o tempo de resposta (RTT - Round Trip Time).

Parâmetros:

- **server** (string): Endereço do servidor DNS a testar (pode ser IP ou nome de domínio)
- **domain** (string): Domínio a ser consultado no servidor DNS

Retorno:

- **número**: Tempo de resposta em milissegundos (latência DNS)

Comportamento:

1. Resolve o endereço do servidor DNS (se for um nome de domínio)
2. Conecta ao servidor DNS na porta 53 (UDP)
3. Realiza uma consulta do tipo A para o domínio especificado
4. Mede o tempo entre o envio da consulta e o recebimento da resposta
5. Retorna o tempo em milissegundos

Exemplo de Uso:

```
-- Testar latência do DNS do Google
local latencia = dns.ping("8.8.8.8", "google.com")
-- latencia = 25 (exemplo: 25 milissegundos)

-- Testar DNS público do Cloudflare
```

```
local latencia_cf = dns.ping("1.1.1.1", "github.com")
-- latencia_cf = 30 (exemplo)

-- Testar servidor DNS por nome
local latencia_local = dns.ping("dns.local", "servidor.producao")
-- Testa o servidor DNS interno "dns.local"

-- Comparar múltiplos servidores DNS
local servidores_dns = {
    {nome = "Google DNS", endereco = "8.8.8.8"},
    {nome = "Cloudflare", endereco = "1.1.1.1"},
    {nome = "Quad9", endereco = "9.9.9.9"},
    {nome = "DNS Local", endereco = "192.168.1.1"}
}

for _, dns in ipairs(servidores_dns) do
    local latencia = dns.ping(dns.endereco, "exemplo.com")
    print(dns.nome .. ": " .. latencia .. "ms")
end
```

Informações Adicionais

Resolução Automática do Servidor DNS

A função `dns.ping` resolve automaticamente o nome do servidor DNS se fornecido como domínio:

```
-- Funciona com IP
dns.ping("8.8.8.8", "google.com")

-- Funciona com nome de domínio (será resolvido primeiro)
dns.ping("dns.google", "exemplo.com")
dns.ping("one.one.one.one", "exemplo.com")  -- Cloudflare DNS
```

Consulta do Tipo A

A função sempre realiza consultas do tipo A (IPv4).

Protocolo UDP

As consultas são realizadas via UDP na porta 53, que é o protocolo padrão para consultas DNS.

Exemplo de uso

Teste de Propagação DNS

```
-- Verificar se um domínio está resolvendo corretamente em diferentes servidores
local function testar_propagacao_dns(dominio, ip_esperado)
    local servidores = {
        "8.8.8.8",      -- Google
        "1.1.1.1",      -- Cloudflare
        "9.9.9.9",      -- Quad9
        "208.67.222.222", -- OpenDNS
        "64.6.64.6",    -- Verisign
    }

    local resultados = {}
    for _, server in ipairs(servidores) do
        local ok, latencia = pcall(dns.ping, server, dominio)

        resultados[server] = {
            latencia = ok and latencia or nil,
            acessivel = ok,
            erro = not ok and latencia or nil
        }
    end

    return resultados
end
```

Funções Globais

Esta seção descreve as funções globais disponíveis no ambiente Lua do Monsta. Estas são funções que não pertencem a módulos específicos, mas estão disponíveis diretamente no escopo global do Lua. Existem funcionalidades para controle de fluxo, manipulação de tempo entre outras.

Funções Disponíveis

`sleep(seconds)`

Descrição: Pausa a execução do script por um número especificado de segundos.

Parâmetros:

- `seconds` (número): Número de segundos para pausar a execução

Retorno: `nil` (não retorna valor)

Exemplo:

```
-- Pausar por 5 segundos
sleep(5)
print("Execução retomada após 5 segundos")
```

`signal(signal_name)`

Descrição: Sinaliza o término da execução do script com um nome de sinal específico.

Parâmetros:

- `signal_name` (string): Nome do sinal a ser emitido

Retorno: Aborta a execução do script com nome do sinal

Características:

- Chama a função `_execution_done` se disponível no ambiente global
- Sempre lança um erro, interrompendo a execução normal

- Útil para controle de fluxo e sinalização de estados

Nota: Atualmente, esta função tem um único caso de uso específico: sinalizar com o nome `"RepeatPrevValue"`. Quando um script emite este sinal, o sistema interpreta que a coleta atual deve repetir o último valor válido da métrica, em vez de gerar um novo ponto de dados. Isso é útil em situações onde a fonte de dados está temporariamente indisponível ou a coleta falhou, mas não se deseja interromper a série temporal.

Exemplo de uso específico:

```
-- Tentar coletar um valor
local value, err = collect_metric()
if err then
    -- Em caso de falha, repetir o valor anterior
    signal("RepeatPrevValue")
end
```

`with_timeout(timeout_ms, func, ...)`

Descrição: Executa uma função com um limite de tempo (timeout).

Parâmetros:

- `timeout_ms` (número): Tempo limite em milissegundos
- `func` (função): Função Lua a ser executada
- `...` (opcional): Argumentos para passar para a função

Retorno: Retorna o resultado da função executada

Exemplo:

```
-- Executar uma função com timeout de 2 segundos
local result = with_timeout(2000, function()
    -- Operação que pode demorar
    return some_long_running_operation()
end)

-- Executar com argumentos
local data = with_timeout(1000, http.get, "https://api.example.com/data")

-- Tratamento de timeout
local success, result = pcall(function()
```

```
    return with_timeout(500, function()
        -- Operação que deve completar rapidamente
        return critical_operation()
    end)
end)

if not success then
    print("Operação excedeu o timeout de 500ms")
end
```

Características:

- Lança um erro se o timeout for excedido
- Preserva os argumentos passados para a função
- Útil para operações de rede ou I/O que podem travar

now()

Descrição: Retorna o número de segundos não bissextos desde 1 de janeiro de 1970 00:00:00 UTC (também conhecido como "timestamp UNIX").

Parâmetros: Nenhum

Retorno: Número representando segundos desde a época Unix

Exemplo:

```
-- Obter timestamp atual
local current_time = now()
print("Timestamp atual:", current_time)

-- Calcular duração de operação
local start_time = now()
-- Executar alguma operação
local end_time = now()
local duration = end_time - start_time
print("Operação levou", duration, "segundos")
```

Características:

- Mesma função disponível em `time.now()`
- Útil para medição de performance

`print(...)`

Descrição: Função de impressão que formata múltiplos argumentos.

Parâmetros:

- `...` (múltiplos valores): Valores a serem impressos

Retorno: `nil` (não retorna valor)

Exemplo:

```
print("Valor:", 42, "Status:", true, "Lista:", {1, 2, 3})
```

Características:

- Separa múltiplos argumentos com tabulação
- Útil para debugging

`diff(lhs, rhs)`

Descrição: Calcula a diferença entre dois valores numéricos, com tratamento especial para contadores que podem sofrer rollover.

Parâmetros:

- `lhs` (número): Valor atual (left-hand side)
- `rhs` (número): Valor anterior (right-hand side)

Retorno: Número representando a diferença entre os valores

Comportamento:

- Calcula `lhs - rhs`
- Se o resultado for negativo, emite o sinal `"RepeatPrevValue"`
- Usado internamente por `snmp.diff` e `wmi.diff` para cálculo de diferenças em contadores
- Útil para métricas de contador que podem sofrer rollover (como contadores de 32 ou 64 bits)

Exemplo:

```
-- Calcular diferença entre leituras de contador
local current_value = 4294967290  -- Valor atual (próximo do rollover de 32 bits)
local previous_value = 4294967280  -- Valor anterior
```

```
local difference = diff(current_value, previous_value)
-- difference = 10 (4294967290 - 4294967280)

-- Caso com rollover (valor diminuiu)
local current_with_rollover = 10  -- Após rollover
local previous_before_rollover = 4294967295  -- Antes do rollover

local rollover_diff = diff(current_with_rollover, previous_before_rollover)
-- Emite sinal "RepeatPrevValue" pois 10 - 4294967295 é negativo
```

Variáveis Globais

EXEC_IDENT

Descrição: Identificador único da execução atual do script.

Tipo: String

Exemplo:

```
-- Usar o identificador em logs
print("Execução ID:", EXEC_IDENT)

-- Incluir em dados de monitoramento
local metrics = {
    ident = EXEC_IDENT,
    timestamp = now(),
    value = collected_data
}

-- Usar como chave para armazenamento
store.put("results_" .. EXEC_IDENT, processing_result)
```

Características:

- Definida automaticamente pelo ambiente
- Única para cada execução de script
- Útil para rastreamento e correlação de logs

Limitações

1. `sleep` **Não é Preciso**: Devido à natureza assíncrona do sistema, `sleep` pode ter variações pequenas.
2. `signal` **Interrompe Execução**: Uma vez chamada, a execução normal é interrompida.

Exemplo

```
-- Monitorar serviço com backoff em caso de falha
local function monitor_with_backoff(service_url, max_attempts)
    local attempt = 1
    local backoff = 1  -- segundos

    while attempt <= max_attempts do
        log.info("Tentativa", attempt, "de", max_attempts)

        local success, status = pcall(function()
            return with_timeout(5000, function()
                return check_service(service_url)
            end)
        end)

        if success and status == "healthy" then
            log.info("Serviço saudável")
            return true
        end

        -- Incrementar backoff exponencial
        sleep(backoff)
        backoff = math.min(backoff * 2, 60)  -- Máximo 60 segundos
        attempt = attempt + 1
    end

    -- Todas as tentativas falharam
    return false
end
```

```
-- Executar monitoramento
monitor_with_backoff("https://api.example.com", 5)
```

Módulo HTTP

Este módulo fornece funções para realizar requisições HTTP a partir de scripts Lua. As funções retornam uma tabela com os resultados da requisição.

Funções Disponíveis

`http.request(method, url, body, headers, options)`

Função genérica para realizar requisições HTTP com qualquer método.

Parâmetros:

- `method` (string): Método HTTP (GET, POST, PUT, DELETE, PATCH, HEAD)
- `url` (string): URL do endpoint
- `body` (string, opcional): Corpo da requisição
- `headers` (tabela, opcional): Cabeçalhos HTTP como pares chave-valor
- `options` (tabela, opcional): Opções adicionais da requisição

Opções disponíveis:

- `verify` (boolean): Verificar certificados SSL (padrão: true)
- `charset` (string): Charset para decodificação da resposta (padrão: "utf-8")

Aviso de Segurança: Desabilitar a verificação SSL (`verify = false`) expõe a conexão a ataques "man-in-the-middle" e deve ser usado **apenas** em ambientes de desenvolvimento ou teste controlados. Nunca utilize esta opção em produção com servidores reais ou com dados sensíveis.

Valor de retorno: Retorna uma tabela com:

- `status` (number): Código de status HTTP
- `body` (string): Corpo da resposta

Exemplo:

```
local resultado = http.request("POST", "https://api.exemplo.com/dados",
    '{"nome": "teste", "valor": 123}',
```

```
    {["Content-Type"] = "application/json"},
    {verify = true, charset = "utf-8"}
)

print("Status:", resultado.status)
print("Resposta:", resultado.body)
```

http.get(url, body, headers, options)

Realiza uma requisição HTTP GET.

Parâmetros:

- `url` (string): URL do endpoint
- `body` (string, opcional): Corpo da requisição (raro para GET, mas suportado)
- `headers` (tabela, opcional): Cabeçalhos HTTP
- `options` (tabela, opcional): Opções adicionais

Exemplo:

```
local resultado = http.get("https://api.exemplo.com/usuarios/1",
    nil,
    {["Authorization"] = "Bearer token123"},
    {verify = true}
)

if resultado.status == 200 then
    print("Dados recebidos:", resultado.body)
else
    print("Erro:", resultado.status)
end
```

http.post(url, body, headers, options)

Realiza uma requisição HTTP POST.

Parâmetros:

- `url` (string): URL do endpoint
- `body` (string): Corpo da requisição (geralmente JSON ou dados de formulário)
- `headers` (tabela, opcional): Cabeçalhos HTTP
- `options` (tabela, opcional): Opções adicionais

Exemplo:

```
local dados_json = '{"nome": "Novo Usuário", "email": "usuario@exemplo.com"}'

local resultado = http.post("https://api.exemplo.com/usuarios",
    dados_json,
    {
        ["Content-Type"] = "application/json",
        ["Authorization"] = "Bearer token123"
    }
)

if resultado.status == 201 then
    print("Usuário criado com sucesso!")
    print("Resposta:", resultado.body)
else
    print("Falha ao criar usuário. Status:", resultado.status)
end
```

`http.put(url, body, headers, options)`

Realiza uma requisição HTTP PUT para atualizar recursos.

Parâmetros:

- `url` (string): URL do endpoint
- `body` (string): Corpo da requisição com dados atualizados
- `headers` (tabela, opcional): Cabeçalhos HTTP
- `options` (tabela, opcional): Opções adicionais

Exemplo:

```
local dados_atualizados = '{"nome": "Usuário Atualizado", "ativo": true}'

local resultado = http.put("https://api.exemplo.com/usuarios/1",
    dados_atualizados,
    {
        ["Content-Type"] = "application/json",
        ["Authorization"] = "Bearer token123"
    }
)
```

```
if resultado.status == 200 then
    print("Usuário atualizado com sucesso!")
else
    print("Falha na atualização. Status:", resultado.status)
end
```

http.delete(url, body, headers, options)

Realiza uma requisição HTTP DELETE para remover recursos.

Parâmetros:

- `url` (string): URL do endpoint
- `body` (string, opcional): Corpo da requisição (opcional para DELETE)
- `headers` (tabela, opcional): Cabeçalhos HTTP
- `options` (tabela, opcional): Opções adicionais

Exemplo:

```
local resultado = http.delete("https://api.exemplo.com/usuarios/1",
    nil,
    {["Authorization"] = "Bearer token123"}
)

if resultado.status == 204 then
    print("Usuário removido com sucesso!")
else
    print("Falha ao remover usuário. Status:", resultado.status)
end
```

http.head(url, body, headers, options)

Realiza uma requisição HTTP HEAD para obter apenas cabeçalhos.

Parâmetros:

- `url` (string): URL do endpoint
- `body` (string, opcional): Corpo da requisição
- `headers` (tabela, opcional): Cabeçalhos HTTP
- `options` (tabela, opcional): Opções adicionais

Exemplo:

```
local resultado = http.head("https://api.exemplo.com/usuarios/1",
    nil,
    {["Authorization"] = "Bearer token123"}
)

print("Status da verificação:", resultado.status)
-- A resposta HEAD geralmente não tem corpo
```

Informações Adicionais

URL Automática

Se a URL não contiver um esquema (como `http://` ou `https://`), o sistema automaticamente adiciona `http://` como prefixo.

```
-- Estas duas chamadas são equivalentes:
local r1 = http.get("api.exemplo.com/dados")
local r2 = http.get("http://api.exemplo.com/dados")
```

Verificação SSL

Por padrão, a verificação de certificados SSL está habilitada. Para desabilitar (útil em ambientes de desenvolvimento/teste):

Atenção: Desabilitar a verificação SSL (`verify = false`) expõe a conexão a ataques "man-in-the-middle" e deve ser usado **apenas** em ambientes de desenvolvimento ou teste controlados. Nunca utilize esta opção em produção com servidores reais ou com dados sensíveis.

```
local resultado = get("https://servidor-local.com",
    nil,
    nil,
    {verify = false}
)
```

Charsets

É possível especificar um charset diferente para decodificar a resposta:

```
local resultado = get("https://api.exemplo.com/dados",
    nil,
    nil,
    {charset = "iso-8859-1"}
)
```

Exemplo de Uso

```
-- Exemplo de monitoramento de API
function verificar_saude_api()
    local resultado = http.get("https://api.exemplo.com/health",
        nil,
        {["User-Agent"] = "MonstaAgent/1.0"}
    )

    if resultado.status == 200 then
        local dados = resultado.body
        -- Processar resposta JSON se necessário
        print("API está saudável")
        return true
    else
        print("API com problemas. Status:", resultado.status)
        return false
    end
end
```

Notas Importantes

1. **Timeout**: O timeout padrão é o timeout global configurado para scripts Lua no sistema.
2. **Performance**: Use conexões persistentes quando fizer múltiplas requisições para o mesmo servidor.
3. **Segurança**: Nunca desabilite a verificação SSL em produção sem necessidade.

Módulo JSON

O módulo **json** fornece funções para converter entre dados Lua e formato JSON (JavaScript Object Notation). É útil para comunicação com APIs web, armazenamento de configurações, serialização de dados e interoperabilidade com outros sistemas.

Funções Disponíveis

1. `json.encode(value)`

Codifica um valor Lua em uma string JSON.

Parâmetros:

- `value` (qualquer tipo): Valor a ser codificado para JSON.

Retorno:

- `string`: Representação JSON do valor

Exceções:

- Lança erro se o valor contiver referências circulares
- Lança erro se a tabela tiver chaves de tipos mistos
- Lança erro se não for possível serializar algum tipo de dado

Exemplos:

```
-- Codificar valores básicos
local json_null = json.encode(nil)           -- "null"
local json_bool = json.encode(true)          -- "true"
local json_num = json.encode(42.5)           -- "42.5"
local json_str = json.encode("Hello\nWorld") -- "\"Hello\\nWorld\""

-- Codificar array (tabela com índices numéricos sequenciais)
local array = {"apple", "banana", "orange"}
local json_array = json.encode(array)
```

```
-- Resultado: "[\"apple\",\"banana\",\"orange\"]"

-- Codificar objeto (tabela com chaves de string)
local person = {
    name = "João Silva",
    age = 30,
    active = true,
    tags = {"developer", "backend"},
    address = {
        street = "Rua das Flores, 123",
        city = "São Paulo"
    }
}
local json_person = json.encode(person)
-- Resultado: {"name":"João
Silva","age":30,"active":true,"tags":["developer","backend"],"address":{"street":"Rua das
Flores, 123","city":"São Paulo"}}

-- Codificar dados de monitoramento
local metrics = {
    timestamp = os.time(),
    hostname = "server-01",
    cpu_usage = 45.7,
    memory_mb = 2048,
    services = {"nginx", "postgresql", "redis"},
    status = "healthy"
}
local json_metrics = json.encode(metrics)

-- Codificar lista de eventos
local events = {
    {
        id = 1,
        type = "login",
        user = "admin",
        timestamp = "2024-01-15T10:30:00Z"
    },
    {
        id = 2,
        type = "logout",
```

```
        user = "user1",
        timestamp = "2024-01-15T11:45:00Z"
    }
}
local json_events = json.encode(events)
```

2. `json.decode(string)`

Decodifica uma string JSON em um valor Lua de forma assíncrona.

Parâmetros:

- `string` (string): String JSON a ser decodificada

Retorno:

- `any`: Objeto decodificado como um valor Lua (mais frequentemente uma tabela):

Exceções:

- Lança erro se a string não for JSON válido
- Lança erro se houver profundidade excessiva de aninhamento
- Lança erro se números forem muito grandes ou muito pequenos

Exemplos:

```
-- Decodificar valores básicos
local null_val = json.decode("null")           -- nil
local bool_val = json.decode("true")           -- true
local num_val = json.decode("42.5")            -- 42.5
local str_val = json.decode("\"Hello\"")       -- "Hello"

-- Decodificar array JSON
local json_array = "[\"apple\", \"banana\", \"orange\"]"
local array = json.decode(json_array)
-- Resultado: {"apple", "banana", "orange"}
print(array[1])  -- "apple"
print(array[2])  -- "banana"
print(#array)    -- 3

-- Decodificar objeto JSON
local json_person = [[
```

```
{
    "name": "Maria Santos",
    "age": 28,
    "active": true,
    "skills": ["Python", "Lua", "JavaScript"],
    "metadata": {
        "department": "Engineering",
        "level": "Senior"
    }
}
]]
local person = json.decode(json_person)
-- Resultado: tabela com chaves name, age, active, skills, metadata
print(person.name)                    -- "Maria Santos"
print(person.age)                     -- 28
print(person.skills[1])               -- "Python"
print(person.metadata.department)     -- "Engineering"

-- Decodificar resposta de API
local api_response = [[
{
    "status": "success",
    "data": {
        "users": [
            {"id": 1, "name": "Alice", "email": "alice@example.com"},
            {"id": 2, "name": "Bob", "email": "bob@example.com"}
        ],
        "total": 2,
        "page": 1
    },
    "timestamp": "2024-01-15T12:00:00Z"
}
]]
local response = json.decode(api_response)
if response.status == "success" then
    for _, user in ipairs(response.data.users) do
        print("Usuário: " .. user.name .. " (" .. user.email .. ")")
    end
    print("Total: " .. response.data.total)
end
```

```
-- Decodificar configurações
local config_json = [[
{
    "server": {
        "host": "0.0.0.0",
        "port": 8080,
        "timeout": 30
    },
    "database": {
        "host": "localhost",
        "name": "monagent",
        "pool_size": 10
    },
    "logging": {
        "level": "info",
        "file": "/var/log/monagent.log"
    }
}
]]
local config = json.decode(config_json)
local server_host = config.server.host          -- "0.0.0.0"
local db_pool = config.database.pool_size       -- 10
local log_level = config.logging.level          -- "info"
```

Módulo Log

O módulo **log** fornece funções de logging para scripts Lua, permitindo registro de mensagens com diferentes níveis de severidade. Este módulo é útil para debugging, monitoramento e auditoria de scripts em produção.

Características principais:

- 5 níveis de log: debug, info, warn, error
- Identificação por contexto de execução
- Formatação automática de valores Lua
- Suporte a múltiplos argumentos com separação por tabulação na saída

Funções Disponíveis

1. `log.set_ident(identificador)`

Define um identificador para os logs gerados pelo script atual.

Parâmetros:

- **identificador** (string): Identificador único para o contexto de execução

Retorno:

- **nil**: A função não retorna valor

Comportamento:

- O identificador é armazenado internamente e persiste durante toda a execução do script
- Todas as mensagens de log subsequentes incluirão este identificador
- Útil para distinguir logs de diferentes scripts ou instâncias

Exemplo de Uso:

```
-- Definir identificador para um script específico
log.set_ident("monitoramento-cpu")
```

```
-- Agora todos os logs incluirão "[monitoramento-cpu]"
log.info("Iniciando monitoramento")
-- Saída: [lua] [monitoramento-cpu] Iniciando monitoramento

-- Em outro script ou contexto
log.set_ident("backup-automatico")
log.info("Iniciando backup")
-- Saída: [lua] [backup-automatico] Iniciando backup
```

2. `log.debug(...)`

Registra mensagens de nível DEBUG para informações detalhadas de debugging.

Parâmetros:

- **...** (múltiplos valores): Valores a serem registrados, separados por tabulação

Retorno:

- **nil**: A função não retorna valor

Exemplo de Uso:

```
-- Log de valores de variáveis para debugging
local temperatura = 45.6
local uso_memoria = 78.3
log.debug("Variáveis de sistema:", "Temp:", temperatura, "Mem:", uso_memoria)
-- Saída: [lua] [ident] Variáveis de sistema: Temp: 45.6 Mem: 78.3

-- Debug de fluxo de execução
log.debug("Entrando na função processar_dados")
log.debug("Parâmetros recebidos:", parametros)
log.debug("Configuração atual:", config)

-- Debug de estruturas complexas
local dados = {
    usuario = "admin",
    acao = "login",
    timestamp = os.time()
}
```

```
log.debug("Dados da requisição:", dados)
```

3. `log.info(...)`

Registra mensagens de nível INFO para informações gerais sobre a execução.

Parâmetros:

- **...** (múltiplos valores): Valores a serem registrados, separados por tabulação

Retorno:

- **nil**: A função não retorna valor

4. `log.warn(...)`

Registra mensagens de nível WARN para situações que requerem atenção mas não são erros.

Parâmetros:

- **...** (múltiplos valores): Valores a serem registrados, separados por tabulação

Retorno:

- **nil**: A função não retorna valor

5. `log.error(...)`

Registra mensagens de nível ERROR para situações de erro que requerem intervenção.

Parâmetros:

- **...** (múltiplos valores): Valores a serem registrados, separados por tabulação

Retorno:

- **nil**: A função não retorna valor

Módulo Ping

O módulo **ping** fornece funções para testar conectividade de rede através de ICMP, TCP e DNS. Este módulo é útil para monitoramento de disponibilidade, diagnóstico de problemas de rede e medição de latência em ambientes de produção.

Características principais:

- Suporte a ICMP ping (IPv4 e IPv6)
- Testes de porta TCP
- Configuração flexível de parâmetros
- Resolução DNS automática
- Timeout configurável

Funções Disponíveis

1. `ping.up([host], [count], [data_size], [timeout], [interval_ms])`

Verifica se um host está acessível através de ping ICMP.

Parâmetros:

- **host** (opcional, string): Endereço do host ou nome de domínio (ex: "google.com", "192.168.1.1")
- **count** (opcional, número, padrão: `params.icmpUpNumPackets` ou 3): Número de tentativas de ping
- **data_size** (opcional, número, padrão: 24): Tamanho do pacote de dados em bytes
- **timeout** (opcional, número, padrão: `params.icmpUpTimeout` ou 2): Timeout em segundos para cada tentativa
- **interval_ms** (opcional, número, padrão: `params.icmpUpInterval` ou 100): Intervalo entre tentativas em milissegundos
- Se `host` não for fornecido, usa `params.address`
- Se `count` não for fornecido, usa `params.icmpUpNumPackets`
- Se `timeout` não for fornecido, usa `params.icmpUpTimeout`
- Se `interval_ms` não for fornecido, usa `params.icmpUpInterval`

Retorno:

- **boolean**: `true` se pelo menos um ping foi bem-sucedido, `false` caso contrário

Exemplo de Uso:

```
-- Verificar se um host está acessível
local acessivel = ping.up("google.com")
-- acessivel = true (se responder ao ping)

-- Verificar com parâmetros personalizados
local resultado = ping.up("servidor.local", 5, 32, 5, 200)
-- 5 tentativas, pacotes de 32 bytes, timeout de 5s, intervalo de 200ms

-- Testar endereço IPv6
local ipv6_ok = ping.up("2001:4860:4860::8888")
-- Testa conectividade IPv6 com Google DNS

-- Verificar múltiplos hosts
local hosts = {"router.local", "192.168.1.1", "8.8.8.8"}
for _, host in ipairs(hosts) do
    if ping.up(host) then
        print(host .. " está online")
    else
        print(host .. " está offline")
    end
end
```

2. `ping.send([host], [count], [data_size], [timeout], [interval_ms])`

Mede a latência de ping ICMP para um host.

Parâmetros:

- **host** (opcional,string): Endereço do host ou nome de domínio
- **count** (opcional, número, padrão: `params.icmpTimeNumPackets` ou 3): Número de tentativas de ping
- **data_size** (opcional, número, padrão: 24): Tamanho do pacote de dados em bytes
- **timeout** (opcional, número, padrão: `params.icmpTimeTimeout` ou 2): Timeout em segundos para cada tentativa

- **interval_ms** (opcional, número, padrão: `params.icmpTimeInterval` ou 100): Intervalo entre tentativas em milissegundos
- Se `host` não for fornecido, usa `params.address`
- Se `count` não for fornecido, usa `params.icmpTimeNumPackets`
- Se `timeout` não for fornecido, usa `params.icmpTimeTimeout`
- Se `interval_ms` não for fornecido, usa `params.icmpTimeInterval`

Retorno:

- **tuple**: `(latencia, erro)` onde:
  - **latencia** (número ou nil): Latência média em milissegundos, ou `nil` se falhar
  - **erro** (string ou nil): Mensagem de erro, ou `nil` se bem-sucedido

Exemplo de Uso:

```
-- Medir latência básica
local latencia, erro = ping.send("google.com")
-- latencia = 25.3 (exemplo), erro = nil

-- Medir com parâmetros personalizados
local latencia, erro = ping.send("servidor.remoto", 10, 64, 5, 500)
-- 10 tentativas, pacotes de 64 bytes, timeout 5s, intervalo 500ms

-- Tratar resultado
local latencia, erro = ping.send("host.inacessivel")
if latencia then
    print("Latência: " .. latencia .. "ms")
else
    print("Erro: " .. erro)
    -- erro = "ping failed" ou mensagem específica
end

-- Comparar latência entre múltiplos hosts
local hosts = {
    "dns.google",
    "cloudflare-dns.com",
    "quad9.net"
}

for _, host in ipairs(hosts) do
    local latencia, erro = ping.send(host, 5)
```

```
    if latencia then
        print(host .. ": " .. string.format("%.2f", latencia) .. "ms")
    end
end
```

3. `ping.port(host, port, [count], [timeout])`

Testa conectividade a uma porta TCP específica.

Parâmetros:

- **host** (string): Endereço do host ou nome de domínio
- **port** (número): Número da porta TCP a testar
- **count** (opcional, número, padrão: 3): Número de tentativas de conexão
- **timeout** (opcional, número, padrão: 2): Timeout em segundos para cada tentativa

Retorno:

- **tuple**: `(latencia, erro)` onde:
  - **latencia** (número ou nil): Latência média de conexão em milissegundos, ou `nil` se falhar
  - **erro** (string ou nil): Mensagem de erro, ou `nil` se bem-sucedido

Exemplo de Uso:

```
-- Testar porta HTTP padrão
local latencia, erro = ping.port("google.com", 80)
-- latencia = 45.2 (exemplo), erro = nil

-- Testar porta HTTPS
local latencia, erro = ping.port("google.com", 443)
if latencia then
    print("HTTPS acessível com latência: " .. latencia .. "ms")
else
    print("HTTPS inacessível: " .. erro)
end

-- Testar múltiplas portas
local portas = {22, 80, 443, 3306, 5432}
```

```
for _, porta in ipairs(portas) do
    local latencia, erro = ping.port("servidor.local", porta, 2, 3)
    if latencia then
        print("Porta " .. porta .. " aberta (" .. latencia .. "ms)")
    else
        print("Porta " .. porta .. " fechada ou inacessível")
    end
end

-- Verificar serviço específico
local function verificar_servico(host, porta, nome_servico)
    local latencia, erro = ping.port(host, porta)
    if latencia then
        log.info(nome_servico .. " OK (" .. latencia .. "ms)")
        return true
    else
        log.error(nome_servico .. " FALHA: " .. erro)
        return false
    end
end
```

Informações Adicionais

Resolução DNS Automática

Todas as funções realizam resolução DNS automaticamente:

```
-- Funciona com nomes de domínio
ping.up("google.com")
ping.send("api.github.com")
ping.port("servidor.local", 443)

-- Funciona com endereços IP
ping.up("8.8.8.8")
ping.send("192.168.1.1")
ping.port("10.0.0.5", 22)
```

Suporte a IPv4 e IPv6

O módulo detecta automaticamente o tipo de endereço:

```
-- IPv4
ping.up("8.8.8.8")

-- IPv6
ping.up("2001:4860:4860::8888")

-- Nome de domínio (resolvido para IPv4 ou IPv6)
ping.up("google.com")
```

Timeout Configurável

Cada função permite configurar timeout individual:

```
-- Timeout curto para redes locais
ping.up("router.local", 3, 24, 1)  -- 1 segundo

-- Timeout longo para redes com alta latência
ping.up("servidor.remoto", 3, 24, 10)  -- 10 segundos

-- Timeout específico por porta
ping.port("database.remoto", 5432, 3, 5)  -- 5 segundos
```

Módulo Process

O módulo **process** fornece uma função para execução de comandos do sistema operacional a partir de scripts Lua. Esta função é útil para automação de tarefas administrativas, coleta de informações do sistema e integração com ferramentas externas.

- Execução de comandos com usuário não privilegiado
- Captura de stdout e stderr separadamente
- Retorno estruturado com sucesso/erro
- Suporte a argumentos variáveis

Nota de segurança: Todos os comandos são executados com o usuário `monstasb` para garantir segurança e controle de permissões.

Funções Disponíveis

1. `process.exec(command, [arg1], [arg2], ...)`

Executa um comando do sistema operacional com os argumentos especificados.

Parâmetros:

- **command** (string): Nome do comando/programa a ser executado
- **...** (opcional, strings): Argumentos adicionais para o comando

Retorno:

- **tuple**: `(out, err)` onde:
  - **out** (string ou nil): Saída padrão do comando se bem-sucedido, ou `nil` se falhar
  - **err** (string ou nil): Saída de erro do comando se falhar, ou `nil` se bem-sucedido

Comportamento:

1. O comando é executado com o usuário `monstasb`
2. Se o comando retornar código de saída 0 (sucesso):
   - `out` contém a saída do comando
   - `err` é `nil`

3. Se o comando falhar (código ≠ 0):
   - `out` é nil
   - `err` contém a saída de erro (stderr) do comando
4. Se houver erro na execução (comando não encontrado, etc.):
   - `out` é `nil`
   - `err` contém a mensagem de erro

Exemplo de Uso:

```
-- Executar comando simples
local saida, erro = process.exec("ls", "-la", "/tmp")
if saida then
    print("Conteúdo de /tmp:")
    print(saida)
else
    print("Erro:", erro)
end

-- Executar comando com múltiplos argumentos
local saida, erro = process.exec("df", "-h")
if saida then
    -- Processar saída do df
    local linhas = string.split(saida, "\n")
    for _, linha in ipairs(linhas) do
        print("Linha:", linha)
    end
end

-- Verificar se um serviço está rodando
local saida, erro = process.exec("systemctl", "is-active", "nginx")
if saida then
    local status = string.trim(saida)
    if status == "active" then
        print("Nginx está ativo")
    else
        print("Nginx não está ativo:", status)
    end
else
    print("Erro ao verificar Nginx:", erro)
end
```

```
-- Coletar informações do sistema
local comandos = {
    {"uname", "-a"},
    {"uptime"},
    {"free", "-h"},
    {"df", "-h", "/"}
}

for _, cmd_args in ipairs(comandos) do
    local comando = table.remove(cmd_args, 1)
    local saida, erro = process.exec(comando, table.unpack(cmd_args))
    if saida then
        print("=== " .. comando .. " ===")
        print(saida)
    end
end
```

Informações Adicionais

Execução com Usuário Específico

Todos os comandos são executados com o usuário `monstasb`:

```
-- Segurança: comandos não são executados como root
process.exec("whoami")  -- Retornará "monstasb", não "root"
```

Suporte a Argumentos Variáveis

Aceita qualquer número de argumentos:

```
-- 1 argumento
process.exec("ls")

-- 2 argumentos
process.exec("ls", "-la")
```

```
-- Múltiplos argumentos
process.exec("find", ".", "-name", "*.log", "-type", "f", "-mtime", "+7")
```

Módulo Registry

O módulo **registry** fornece um sistema de armazenamento de dados persistente e compartilhado entre execuções de scripts Lua. Este módulo é útil para manter estado entre diferentes execuções, compartilhar dados entre scripts e implementar mecanismos de cache e configuração persistente.

- Armazenamento chave-valor persistente
- Timestamp automático para cada entrada
- Thread-safe (seguro para uso concorrente)
- Dados compartilhados entre todos os scripts Lua
- Persistência entre reinícios do agente

Funções Disponíveis

1. `registry.put(chave, valor)`

Armazena um valor no registry com a chave especificada.

Parâmetros:

- **chave** (string): Identificador único para o valor
- **valor** (qualquer tipo Lua): Valor a ser armazenado (string, número, booleano, tabela, nil)

Retorno:

- **nil**: A função não retorna valor

Comportamento:

1. Armazena o valor associado à chave
2. Registra automaticamente o timestamp atual (UTC)
3. Sobrescreve qualquer valor existente com a mesma chave
4. Aceita qualquer tipo de dado Lua suportado pelo sistema de valores do Monagent

Exemplo de Uso:

```
-- Armazenar valores básicos
registry.put("ultima_execucao", os.time())
```

```
registry.put("hostname", "servidor-producao")
registry.put("ativo", true)
registry.put("versao", 2.5)

-- Armazenar tabelas complexas
local config = {
    timeout = 30,
    retries = 3,
    servidores = {"srv1", "srv2", "srv3"},
    limites = {cpu = 80, memoria = 90, disco = 95}
}
registry.put("configuracao_monitoramento", config)

-- Armazenar resultados de processamento
local metricas = {
    cpu_usage = 45.6,
    memory_usage = 78.3,
    disk_usage = 62.1,
    timestamp = os.time()
}
registry.put("metricas_recentes", metricas)

-- Sobrescrever valor existente
registry.put("contador", 1)
-- ... mais tarde ...
local valor, timestamp = registry.get("contador")
registry.put("contador", (valor or 0) + 1)
```

2. `registry.get(chave)`

Recupera um valor do registry pela chave especificada.

Parâmetros:

- **chave** (string): Chave do valor a ser recuperado

Retorno:

- **tuple**: `(valor, timestamp)` onde:
  - **valor** (qualquer tipo ou nil): Valor armazenado, ou `nil` se a chave não existir

- **timestamp** (número ou nil): Timestamp Unix (segundos) da última atualização, ou `nil` se a chave não existir

Comportamento:

1. Retorna o valor e timestamp se a chave existir
2. Retorna `(nil, nil)` se a chave não existir
3. Preserva o tipo original do valor armazenado
4. O timestamp é em segundos desde a epoch Unix (UTC)

Exemplo de Uso:

```
-- Recuperar valor simples
local valor, timestamp = registry.get("config_timeout")
if valor then
    print("Timeout configurado:", valor, "desde", os.date("%H:%M:%S", timestamp))
else
    print("Configuração não encontrada")
end

-- Recuperar tabela
local config, ts = registry.get("configuracao_sistema")
if config then
    print("Configuração carregada:")
    for chave, valor in pairs(config) do
        print("  ", chave, "=", valor)
    end
    print("Última atualização:", os.date("%Y-%m-%d %H:%M:%S", ts))
end

-- Verificar existência antes de usar
local dados, timestamp = registry.get("dados_processamento")
if dados then
    -- Processar dados existentes
    processar_dados(dados)
else
    -- Inicializar novos dados
    dados = inicializar_dados()
    registry.put("dados_processamento", dados)
end
```

```
-- Usar valor padrão se não existir
local limite, _ = registry.get("limite_cpu")
limite = limite or 80  -- Valor padrão 80% se não configurado
```

3. `registry.delete(chave)`

Remove uma entrada do registry pela chave especificada.

Parâmetros:

- **chave** (string): Chave da entrada a ser removida

Retorno:

- **nil**: A função não retorna valor

Comportamento:

1. Remove completamente a entrada do registry
2. Não faz nada se a chave não existir
3. Libera a memória associada à chave
4. Operação atômica e thread-safe

Exemplo de Uso:

```
-- Remover entrada específica
registry.delete("cache_temporario")
print("Cache temporário removido")

-- Limpar todas as entradas de um prefixo
local function limpar_prefixo(prefixo)
    -- Nota: O registry não suporta listagem direta
    -- Em um caso real, você precisaria manter uma lista de chaves
    log.info("Limpando entradas com prefixo:", prefixo)

    -- Exemplo com chaves conhecidas
    local chaves_conhecidas = {
        prefixo .. "_cache_dns",
        prefixo .. "_cache_http",
        prefixo .. "_metricas",
        prefixo .. "_config"
```

```
    }

    for _, chave in ipairs(chaves_conhecidas) do
        registry.delete(chave)
        log.debug("Removido:", chave)
    end
end

-- Limpar entradas expiradas
local function limpar_expirados(ttl_segundos)
    local agora = os.time()
    -- Nota: Novamente, sem listagem direta, precisamos de estratégia
    log.info("Limpeza de expirados não suportada diretamente")
    log.info("Use timestamps nas chaves ou mantenha lista de chaves")
end

-- Remover após uso
local function processar_e_limpar(chave)
    local dados, timestamp = registry.get(chave)

    if dados then
        -- Processar dados
        local resultado = processar(dados)

        -- Remover após processamento
        registry.delete(chave)
        log.info("Dados processados e removidos:", chave)

        return resultado
    else
        log.warn("Chave não encontrada:", chave)
        return nil
    end
end
```

Módulo SNMP

O módulo **snmp** fornece uma interface completa para monitoramento e gerenciamento de dispositivos de rede através do protocolo SNMP (Simple Network Management Protocol). Este módulo é útil para coleta de métricas de roteadores, switches, servidores, impressoras e qualquer dispositivo que suporte SNMP.

- Suporte a SNMP v1, v2c e v3
- Operações GET, GET BULK e WALK
- Autenticação e criptografia SNMP v3
- Configuração flexível de timeout e retentativas
- Retorno seguro com tratamento de erros

Protocolos suportados:

- **SNMP v1**: Protocolo básico com comunidade pública/privada
- **SNMP v2c**: Melhorias de performance com operações BULK
- **SNMP v3**: Segurança avançada com autenticação e criptografia

Configuração SNMP

Todas as funções SNMP requerem um objeto de configuração que define os parâmetros de conexão. A configuração é uma tabela Lua com os seguintes campos:

Campos de Configuração Básicos:

Campo	Tipo	Padrão	Descrição
`address`	string	**obrigatório**	Endereço IP ou hostname do dispositivo
`snmpVersion`	número	1	Versão SNMP (1=v1, 2=v2c, 3=v3)
`snmpPort`	número	161	Porta SNMP
`snmpCommunity`	string	nil	Comunidade SNMP (v1/v2c)
`snmpTimeout`	número	5	Timeout em segundos
`snmpRetryCount`	número	3	Número de retentativas

Campo	Tipo	Padrão	Descrição
`snmpMaxBulkItems`	número	nil	Máximo de itens por operação BULK
`snmpExponentialBackoff`	booleano	false	Habilitar backoff exponencial

Campos para SNMP v3:

Campo	Tipo	Descrição
`snmpSecurityLevel`	string	Nível de segurança: "NoAuthNoPriv", "AuthNoPriv", "AuthPriv"
`snmpAuthProtocol`	string	Protocolo de autenticação: "MD5", "SHA1"
`snmpAuthUser`	string	Usuário de autenticação
`snmpAuthPassword`	string	Senha de autenticação
`snmpPrivProtocol`	string	Protocolo de criptografia: "DES", "AES"
`snmpPrivPassword`	string	Senha de criptografia

Configuração via Tabela `params`:

O ambiente Lua inclui uma tabela predefinida chamada `params` que contém os detalhes do dispositivo atual. Esta tabela pode ser usada diretamente como configuração SNMP, pois já possui os campos necessários no formato esperado.

Exemplo de uso direto:

```
-- Usar a tabela device diretamente como configuração
local sys_descr = snmp.getex(device, "1.3.6.1.2.1.1.1.0")
print("Descrição do dispositivo:", sys_descr)

-- Versão que não lança erro
local valor, erro = snmp.get_safe(device, "1.3.6.1.2.1.1.3.0")
if not erro then
    print("Uptime do dispositivo:", valor)
end
```

Combinando com configurações adicionais:

```
-- Criar configuração baseada em device com ajustes
local config = {
    address = device.address,
    snmpVersion = device.snmpVersion,
    snmpCommunity = device.snmpCommunity,
    snmpTimeout = device.snmpTimeout or 5,  -- Usar padrão se não definido
    snmpRetryCount = 2,  -- Sobrescrever valor padrão
    snmpMaxBulkItems = 50  -- Adicionar configuração extra
}

-- Usar para operação BULK
local oids = {"1.3.6.1.2.1.1.1.0", "1.3.6.1.2.1.1.3.0"}
local resultados = snmp.get_bulk(config, oids)
```

Exemplos de Configuração:

```
-- Configuração básica SNMP v2c
local config_v2c = {
    address = "192.168.1.1",
    snmpVersion = 2,
    snmpCommunity = "public",
    snmpTimeout = 3,
    snmpRetryCount = 2
}

-- Configuração SNMP v3 com autenticação e criptografia
local config_v3 = {
    address = "10.0.0.254",
    snmpVersion = 3,
    snmpSecurityLevel = "AuthPriv",
    snmpAuthProtocol = "SHA1",
    snmpAuthUser = "monitor",
    snmpAuthPassword = "senha123",
    snmpPrivProtocol = "AES",
    snmpPrivPassword = "chave456",
    snmpTimeout = 5
}
```

```
-- Configuração para dispositivo com porta não padrão
local config_custom_port = {
    address = "switch.piso1.local",
    snmpVersion = 2,
    snmpCommunity = "internal",
    snmpPort = 8161,  -- Porta customizada
    snmpTimeout = 10  -- Timeout maior para rede lenta
}
```

Funções Disponíveis

1. `snmp.getex(config, oid)`

Realiza uma consulta SNMP GET para um OID específico.

Parâmetros:

- **config** (tabela): Configuração SNMP (ver seção acima)
- **oid** (string): OID a ser consultado (formato numérico ou nomeado)

Retorno:

- **valor**: Valor retornado pelo dispositivo SNMP (número, string, etc.)

Erros:

- Lança erro se o OID não existir ou houver falha na comunicação

Exemplo de Uso:

```
-- Consultar sysDescr (descrição do sistema)
local config = {
    address = "192.168.1.1",
    snmpVersion = 2,
    snmpCommunity = "public"
}

local sys_descr = snmp.getex(config, "1.3.6.1.2.1.1.1.0")
-- ou usando OID nomeado
```

```
local sys_descr = snmp.getex(config, ".1.3.6.1.2.1.1.1.0")

print("Descrição do sistema:", sys_descr)
-- Exemplo de saída: "Cisco IOS Software, C3750 Software (C3750-IPSERVICESK9-M), Version
12.2(55)SE10, RELEASE SOFTWARE (fc2)"

-- Consultar uptime do sistema
local sys_uptime = snmp.getex(config, "1.3.6.1.2.1.1.3.0")
print("Uptime:", sys_uptime, "centésimos de segundo")

-- Consultar nome do host
local sys_name = snmp.getex(config, "1.3.6.1.2.1.1.5.0")
print("Nome do host:", sys_name)

-- Consultar localização
local sys_location = snmp.getex(config, "1.3.6.1.2.1.1.6.0")
print("Localização:", sys_location)
```

2. `snmp.get_safe(config, oid)`

Versão segura de `getex` que não lança exceções, retornando erro como segundo valor.

Parâmetros:

- **config** (tabela): Configuração SNMP
- **oid** (string): OID a ser consultado

Retorno:

- **tuple**: `(valor, erro)` onde:
  - **valor** (qualquer tipo ou nil): Valor retornado se bem-sucedido
  - **erro** (string ou nil): Mensagem de erro se falhar, nil se bem-sucedido

Exemplo de Uso:

```
local config = {
    address = "192.168.1.1",
    snmpVersion = 2,
    snmpCommunity = "public"
}
```

```
-- Consulta segura que não quebra o script em caso de erro
local valor, erro = snmp.get_safe(config, "1.3.6.1.2.1.1.1.0")

if erro then
    log.error("Falha na consulta SNMP:", erro)
    -- Tomar ação alternativa
else
    print("Valor obtido:", valor)
end

-- Consultar múltiplos OIDs com tratamento de erro individual
local oids = {
    "1.3.6.1.2.1.1.1.0",  -- sysDescr
    "1.3.6.1.2.1.1.3.0",  -- sysUpTime
    "1.3.6.1.2.1.1.5.0",  -- sysName
    "1.3.6.1.2.1.1.6.0"   -- sysLocation
}

local resultados = {}
for _, oid in ipairs(oids) do
    local valor, erro = snmp.get_safe(config, oid)
    if erro then
        log.warn("Falha no OID", oid, ":", erro)
        resultados[oid] = {erro = erro}
    else
        resultados[oid] = {valor = valor}
    end
end
```

3. `snmp.get_bulk(config, oids)`

Realiza operação SNMP GET BULK para múltiplos OIDs de uma vez (SNMP v2c/v3).

Parâmetros:

- **config** (tabela): Configuração SNMP (deve ser v2 ou v3)
- **oids** (array de strings): Lista de OIDs para consulta

Retorno:

- **tabela**: Mapa OID → valor para todos os OIDs consultados

Comportamento:

- Mais eficiente que múltiplas chamadas `getex` para muitos OIDs
- Suportado apenas em SNMP v2c e v3
- Usa `snmpMaxBulkItems` da configuração para limitar tamanho

Exemplo de Uso:

```
local config = {
    address = "192.168.1.1",
    snmpVersion = 2,  -- Deve ser v2 ou v3 para GET BULK
    snmpCommunity = "public",
    snmpMaxBulkItems = 50  -- Limitar a 50 OIDs por operação
}

-- Consultar múltiplas informações do sistema de uma vez
local oids = {
    "1.3.6.1.2.1.1.1.0",  -- sysDescr
    "1.3.6.1.2.1.1.3.0",  -- sysUpTime
    "1.3.6.1.2.1.1.5.0",  -- sysName
    "1.3.6.1.2.1.1.6.0",  -- sysLocation
    "1.3.6.1.2.1.1.7.0"   -- sysServices
}

local resultados = snmp.get_bulk(config, oids)

for oid, valor in pairs(resultados) do
    print("OID:", oid, "=", valor)
end

-- Consultar informações de múltiplas interfaces
local function obter_info_interfaces(config, indices)
    local oids = {}
    for _, idx in ipairs(indices) do
        table.insert(oids, "1.3.6.1.2.1.2.2.1.2." .. idx)   -- ifDescr
        table.insert(oids, "1.3.6.1.2.1.2.2.1.3." .. idx)   -- ifType
        table.insert(oids, "1.3.6.1.2.1.2.2.1.5." .. idx)
    end
    return snmp.get_bulk(config, oids)
```

```
end
```

4. `snmp.get(oid)`

Versão simplificada de `snmp.getex` que usa automaticamente a configuração do dispositivo atual (`params`).

Parâmetros:

- **oid** (string): OID a ser consultado

Retorno:

- **valor**: Valor retornado pelo dispositivo SNMP

Comportamento:

- Usa `params` como configuração
- Lança erro se o OID não existir ou houver falha na comunicação

Exemplo de Uso:

```
-- Consulta simplificada usando a configuração do dispositivo atual
local sys_descr = snmp.get("1.3.6.1.2.1.1.1.0")
print("Descrição do sistema:", sys_descr)

-- Consultar múltiplos OIDs
local uptime = snmp.get("1.3.6.1.2.1.1.3.0")
local hostname = snmp.get("1.3.6.1.2.1.1.5.0")

print("Uptime:", uptime, "Hostname:", hostname)
```

5. `snmp.walk(oid, cache_ttl, enforce_ordering)`

Realiza uma operação SNMP WALK com suporte a cache e ordenação.

Parâmetros:

- **oid** (string): OID base para o walk

- **cache_ttl** (número, opcional): Tempo de vida do cache em segundos
- **enforce_ordering** (booleano, opcional): Forçar ordenação dos resultados (ordem natural)

Comportamento do cache:

- Usa cache quando `cache_ttl` é especificado

Exemplo de Uso:

```
-- Walk com cache de 30 segundos
local interfaces = snmp.walk("1.3.6.1.2.1.2.2.1.2", 30)
for oid, ifname in pairs(interfaces) do
    print("Interface", oid, ":", ifname)
end

-- Walk com ordenação forçada (sem cache)
local ordered_interfaces = snmp.walk("1.3.6.1.2.1.2.2.1.2", nil, true)
print("Total de interfaces:", #ordered_interfaces)
```

Comportamento de Ordenação:

O parâmetro `enforce_ordering` controla como os resultados são estruturados:

- **Quando `false` (padrão)**: Os resultados são retornados como uma tabela Lua onde cada OID é uma chave que mapeia para seu valor. Esta estrutura é eficiente para acesso aleatório, mas **perde a ordenação natural** dos OIDs, pois as tabelas Lua não preservam a ordem de inserção das chaves.
- **Quando `true`**: Os resultados são retornados como uma **lista de pares** (tabela de tabelas), onde cada elemento é uma tabela contendo 2 elementos. Esta estrutura preserva a ordem natural dos OIDs conforme retornados pelo dispositivo SNMP.

Exemplo de diferença:

```
-- Com enforce_ordering = false (padrão)
local resultado_tabela = snmp.walk("1.3.6.1.2.1.2.2.1.2", nil, false)
-- Estrutura: { ["1.3.6.1.2.1.2.2.1.2.1"] = "eth0", ["1.3.6.1.2.1.2.2.1.2.2"] = "eth1" }
-- A ordem das chaves não é garantida

-- Com enforce_ordering = true
local resultado_lista = snmp.walk("1.3.6.1.2.1.2.2.1.2", nil, true)
-- Estrutura: { {"1.3.6.1.2.1.2.2.1.2.1", "eth0"},
--              {"1.3.6.1.2.1.2.2.1.2.2", "eth1"} }
-- A ordem dos elementos é preservada
```

Quando usar cada modo:

- Use `enforce_ordering = false` quando você só precisa acessar valores por OID específico e a ordem não importa.
- Use `enforce_ordering = true` quando você precisa processar os resultados na mesma ordem em que foram retornados pelo dispositivo, como para:
  - Gerar relatórios ordenados
  - Processar sequências de índices consecutivos
  - Manter correspondência com outras listas ordenadas

Retorno:

- **tabela**: Mapa OID → valor para todos os OIDs encontrados

6. `snmp.walkex(device, oid, cache_ttl)`

Função estendida de walk com sistema de cache avançado e prevenção de execuções concorrentes.

Parâmetros:

- **device** (tabela): Configuração do dispositivo
- **oid** (string): OID base para o walk
- **cache_ttl** (número, opcional): Tempo de vida do cache em segundos

Retorno:

- **tabela**: Mapa OID → valor para todos os OIDs encontrados

Comportamento:

- Implementa cache global compartilhado entre execuções
- Previne execuções concorrentes do mesmo walk
- Usa `registry` para coordenar execuções simultâneas

Exemplo de Uso:

```
-- Walk estendido com cache de 60 segundos
local device_config = {
    address = "192.168.1.1",
    snmpVersion = 2,
    snmpCommunity = "public"
}
```

```
local sys_oids = snmp.walkex(device_config, "1.3.6.1.2.1.1", 60)
for oid, value in pairs(sys_oids) do
    print("OID:", oid, "Valor:", value)
end
```

7. `snmp.count(oid)`

Conta o número de itens retornados por um walk.

Parâmetros:

- **oid** (string): OID base para contar

Retorno:

- **número**: Quantidade de itens encontrados

Exemplo de Uso:

```
-- Contar número de interfaces
local num_interfaces = snmp.count("1.3.6.1.2.1.2.2.1.2")
print("Número de interfaces:", num_interfaces)

-- Contar número de processos
local num_processes = snmp.count("1.3.6.1.2.1.25.4.2.1.2")
print("Número de processos:", num_processes)
```

8. `snmp.diff(typ, lhs, rhs)`

Calcula a diferença entre dois valores, tratando rollover de contadores.

Parâmetros:

- **typ** (número): Tipo do contador (32 ou 64 bits)
- **lhs** (número): Valor atual
- **rhs** (número): Valor anterior

Retorno:

- **número**: Diferença entre os valores

Comportamento:

- Trata rollover de contadores de 32 e 64 bits
- Sinaliza `RepeatPrevValue` se a diferença for negativa

Exemplo de Uso:

```
-- Calcular diferença para contador de 32 bits
local current_bytes = snmp.get("1.3.6.1.2.1.2.2.1.10.1")  -- ifInOctets.1
local prev_bytes = prev("1.3.6.1.2.1.2.2.1.10.1")
local bytes_diff = snmp.diff(32, current_bytes, prev_bytes)

print("Bytes recebidos desde última leitura:", bytes_diff)
```

9. `inst(oid)`

Resolve dinamicamente OIDs de instâncias baseado no nome da instância.

Parâmetros:

- **oid** (string): OID base (sem índice de instância)

Retorno:

- **string**: OID completo com índice de instância

Comportamento:

- Usa `params.InstanceName` e `params.snmpOIDDesc` para resolução
- Suporta cache de instâncias
- Lança erro se a instância não for encontrada

Exemplo de Uso:

```
-- Resolver OID para instância específica
-- params.InstanceName = "eth0"
-- params.snmpOIDDesc = "1.3.6.1.2.1.2.2.1.2"  -- ifDescr

local if_in_octets_oid = inst("1.3.6.1.2.1.2.2.1.10")  -- ifInOctets
print("OID resolvido:", if_in_octets_oid)
-- Saída: "1.3.6.1.2.1.2.2.1.10.1" (se eth0 for índice 1)
```

```
-- Consultar usando OID resolvido
local bytes_in = snmp.get(if_in_octets_oid)
print("Bytes recebidos na interface eth0:", bytes_in)
```

10. `prev(oid)`

Obtém o valor anterior de um OID armazenado.

Parâmetros:

- **oid** (string): OID para obter valor anterior

Retorno:

- **qualquer tipo**: Valor anterior armazenado, ou 0 se não existir

Comportamento:

- Busca valor em `store.get("snmp.value." .. oid)`
- Retorna 0 se não encontrar valor armazenado

Exemplo de Uso:

```
-- Obter valor anterior para cálculo de taxa
local current_value = snmp.get("1.3.6.1.2.1.2.2.1.16.1")  -- ifOutOctets.1
local previous_value = prev("1.3.6.1.2.1.2.2.1.16.1")

local bytes_out_diff = current_value - previous_value
print("Bytes enviados desde última leitura:", bytes_out_diff)
```

11. `lapsed(oid)`

Obtém o tempo decorrido desde a última leitura de um OID.

Parâmetros:

- **oid** (string): OID para verificar tempo decorrido

Retorno:

- **número**: Tempo em segundos desde última leitura, ou 1 se não houver registro

Exemplo de Uso:

```
-- Calcular taxa por segundo
local current_counter = snmp.get("1.3.6.1.2.1.2.2.1.10.1")  -- ifInOctets.1
local previous_counter = prev("1.3.6.1.2.1.2.2.1.10.1")
local time_elapsed = lapsed("1.3.6.1.2.1.2.2.1.10.1")

local bytes_per_second = (current_counter - previous_counter) / time_elapsed
print("Taxa de recebimento:", bytes_per_second, "bytes/segundo")
```

Exemplos Completos

Monitoramento de Interface de Rede:

```
-- Resolver OID da interface eth0
local if_index_oid = inst("1.3.6.1.2.1.2.2.1.1")  -- ifIndex
local if_descr_oid = inst("1.3.6.1.2.1.2.2.1.2")  -- ifDescr

-- Obter informações da interface
local interface_index = snmp.get(if_index_oid)
local interface_name = snmp.get(if_descr_oid)

print("Monitorando interface:", interface_name, "(índice", interface_index, ")")

-- Coletar estatísticas
local in_octets = snmp.get(inst("1.3.6.1.2.1.2.2.1.10"))  -- ifInOctets
local out_octets = snmp.get(inst("1.3.6.1.2.1.2.2.1.16")) -- ifOutOctets
local in_errors = snmp.get(inst("1.3.6.1.2.1.2.2.1.14"))  -- ifInErrors
local out_errors = snmp.get(inst("1.3.6.1.2.1.2.2.1.20")) -- ifOutErrors

-- Calcular diferenças desde última leitura
local time_elapsed = lapsed(inst("1.3.6.1.2.1.2.2.1.10"))
local prev_in = prev(inst("1.3.6.1.2.1.2.2.1.10"))
local prev_out = prev(inst("1.3.6.1.2.1.2.2.1.16"))
```

```
local in_rate = (in_octets - prev_in) / time_elapsed
local out_rate = (out_octets - prev_out) / time_elapsed

print("Taxa de entrada:", in_rate, "bytes/seg")
print("Taxa de saída:", out_rate, "bytes/seg")
print("Erros de entrada:", in_errors)
print("Erros de saída:", out_errors)
```

Inventário de Interfaces com Walk:

```
-- Listar todas as interfaces com walk
local interfaces = snmp.walk("1.3.6.1.2.1.2.2.1.2", 300)  -- ifDescr com cache de 5 minutos

print("=== Inventário de Interfaces ===")
for oid, ifname in pairs(interfaces) do
    -- Extrair índice da interface do OID
    local index = string.match(oid, "(%d+)$")

    -- Obter tipo e status da interface
    local iftype = snmp.get("1.3.6.1.2.1.2.2.1.3." .. index)   -- ifType
    local ifstatus = snmp.get("1.3.6.1.2.1.2.2.1.8." .. index)  -- ifOperStatus

    local status_text = "DOWN"
    if ifstatus == 1 then status_text = "UP" end

    print(string.format("Interface %s: %s (Tipo: %d, Status: %s)",
                        index, ifname, iftype, status_text))
end

print("Total de interfaces:", snmp.count("1.3.6.1.2.1.2.2.1.2"))
```

Monitoramento de Uso de CPU com Múltiplas Instâncias:

```
-- Usar walk para obter todas as CPUs
local cpu_oids = snmp.walk("1.3.6.1.2.1.25.3.3.1.2", 30)  -- hrProcessorLoad
```

```
local total_load = 0
local cpu_count = 0

for oid, load in pairs(cpu_oids) do
    cpu_count = cpu_count + 1
    total_load = total_load + load

    local cpu_index = string.match(oid, "(%d+)$")
    print(string.format("CPU %d: %d%%", cpu_index, load))
end

if cpu_count > 0 then
    local avg_load = total_load / cpu_count
    print(string.format("Média de uso de CPU: %.1f%%", avg_load))
end
```

Módulo SSH

O módulo **ssh** fornece funções para executar comandos remotos em servidores através do protocolo Secure Shell (SSH). Oferece duas abordagens principais: execução única de comandos e conexões persistentes para múltiplos comandos. O módulo suporta autenticação por senha e inclui resolução DNS automática.

Funções Disponíveis

1. `ssh.exec(opts)`

Executa um comando remoto via SSH de forma assíncrona e retorna a saída do comando.

Parâmetros:

- `opts` (tabela): Opções de conexão e execução contendo:
  - `host` (string): Endereço do servidor (hostname ou IP)
  - `username` (string): Nome de usuário para autenticação
  - `password` (string): Senha para autenticação
  - `command` (string): Comando a ser executado no servidor remoto
  - `port` (número opcional, padrão: 22): Porta SSH
  - `timeout` (número opcional, padrão: 60): Timeout em segundos para a conexão

Retorno:

- `string`: Saída padrão (stdout) do comando executado

Exceções:

- Lança erro se algum parâmetro obrigatório estiver faltando
- Lança erro se a conexão SSH falhar
- Lança erro se o comando falhar no servidor remoto
- Lança erro "connection timeout" se exceder o tempo limite

Exemplos:

```
-- Execução básica de comando
local opts = {
```

```
    host = "servidor.exemplo.com",
    username = "admin",
    password = "senha123",
    command = "uname -a",
    port = 22,
    timeout = 30
}

local success, output = pcall(function()
    return ssh.exec(opts)
end)

if success then
    print("Saída do comando: " .. output)
else
    print("Erro SSH: " .. output)
end

-- Verificar uso de disco
local disk_opts = {
    host = "192.168.1.100",
    username = "monitor",
    password = "monitor_pass",
    command = "df -h /"
}

local disk_usage = ssh.exec(disk_opts)

-- Verificar serviços em execução
local service_opts = {
    host = "app-server",
    username = "root",
    password = "root_password",
    command = "systemctl list-units --type=service --state=running"
}

local services = ssh.exec(service_opts)
```

2. `ssh.connect(opts)`

Estabelece uma conexão SSH persistente e retorna um objeto cliente que pode executar múltiplos comandos.

Nota: Esta função usa valores padrão da tabela `params` quando os parâmetros não são fornecidos.

Parâmetros:

- `opts` (tabela): Opções de conexão contendo:
  - `host` (string): Endereço do servidor (hostname ou IP)
  - `username` (string): Nome de usuário para autenticação
  - `password` (string): Senha para autenticação
  - `port` (número opcional, padrão: 22): Porta SSH
  - `timeout` (número opcional, padrão: `params.sshTimeout` ou 60): Timeout em segundos para a conexão

Retorno:

- `SshClient`: Objeto cliente SSH com método `exec()` para executar comandos

Exceções:

- Lança erro se algum parâmetro obrigatório estiver faltando
- Lança erro se a conexão SSH falhar
- Lança erro "connection timeout" se exceder o tempo limite

Exemplos:

```
-- Criar conexão persistente
local connect_opts = {
    host = "banco-de-dados.exemplo.com",
    username = "dba",
    password = "dba_password",
    port = 2222,
    timeout = 45
}

local success, client = pcall(function()
    return ssh.connect(connect_opts)
end)

if not success then
    print("Falha na conexão SSH: " .. client)
    return
end
```

```
-- Executar múltiplos comandos na mesma conexão
local cmd1_output = client:exec("pg_isready")
local cmd2_output = client:exec("psql -c 'SELECT version();'")
local cmd3_output = client:exec("df -h /var/lib/postgresql")

print("PostgreSQL status: " .. cmd1_output)
print("Versão PostgreSQL: " .. cmd2_output)
print("Espaço em disco: " .. cmd3_output)
```

3. `SshClient:exec(command)`

Método do objeto cliente retornado por `ssh.connect()` para executar comandos na conexão estabelecida.

Parâmetros:

- `command` (string): Comando a ser executado no servidor remoto

Retorno:

- `string`: Saída padrão (stdout) do comando executado

Exceções:

- Lança erro se a conexão subjacente falhar
- Lança erro se o comando falhar no servidor remoto

Exemplos:

```
-- Uso com múltiplos comandos relacionados
-- Exemplo usando valores explícitos
local client = ssh.connect({
    host = "monitor-server",
    username = "metrics",
    password = "metrics_pass"
})

-- Exemplo usando valores do params configurado no dispositivo
-- params.sshUsername = "admin"
-- params.sshPassword = "admin123"
```

```
-- params.address = "servidor.local"
-- params.sshPort = 22
-- params.sshTimeout = 30

local client_simplificado = ssh.connect({})
-- Equivalente a: ssh.connect({
--     host = "servidor.local",
--     username = "admin",
--     password = "admin123",
--     port = 22,
--     timeout = 30
-- })

-- Exemplo misto (alguns valores explícitos, outros do params)
local client_misto = ssh.connect({
    host = "backup-server",
    port = 2222
})
-- Usa: host = "backup-server", port = 2222
-- Usa do params: username = "admin", password = "admin123", timeout = 30

-- Coletar métricas do sistema
local uptime = client:exec("uptime")
local memory = client:exec("free -m")
local cpu = client:exec("top -bn1 | grep 'Cpu(s)'")
local disk = client:exec("iostat -x 1 1")

-- Processar resultados
print("Uptime: " .. uptime)
print("Memória: " .. memory)
print("CPU: " .. cpu)
print("Disk I/O: " .. disk)
```

Informações Adicionais

1. **Resolução DNS Automática**

- Hostnames são automaticamente resolvidos para endereços IP
- Usa o módulo DNS interno do Monsta
- Suporta tanto IPv4 quanto IPv6

2. **Conexões Persistentes**

- Conexões SSH podem ser reutilizadas para múltiplos comandos
- Reduz overhead de autenticação para comandos sequenciais
- Melhora performance em operações em lote

3. **Timeout Configurável**

- Timeout padrão de 60 segundos para conexões
- Configurável por conexão via parâmetro `timeout`
- Prevenção contra conexões travadas

Limitações

1. **Autenticação Apenas por Senha**

- Não suporta autenticação por chave pública (SSH key)

2. **Performance**

- Conexões SSH têm overhead significativo
- Não recomendado para comandos muito frequentes (use agentes locais)

Módulo Store

O módulo **store** fornece um sistema de armazenamento persistente de dados que utiliza namespacing automático. Cada valor armazenado é automaticamente associado ao identificador de execução (EXEC_IDENT) da métrica ou script, criando um namespace isolado para cada um. Isso significa que, por padrão, diferentes métricas não compartilham dados, mas múltiplas execuções da mesma métrica podem acessar e modificar os valores que ela armazenou anteriormente. O módulo é útil para armazenar estados, configurações ou históricos que precisam persistir entre reinícios do agente.

- Armazenamento chave-valor persistente em disco
- Namespacing automático baseado no identificador da métrica/script
- Timestamp automático para cada entrada
- Thread-safe (seguro para uso concorrente)
- Persistência garantida entre reinícios do agente

Diferença entre Store, Registry e Cache

Store vs Registry:

- **Store**: Persiste em disco, sobrevive a reinícios do agente mas é global para todas as métricas/scripts
- **Registry**: Armazenamento apenas em memória, perdido no reinício

Store vs Cache:

- **Store**: Persistente, sem políticas de expiração automática
- **Cache**: Otimizado para acesso rápido, com políticas de expiração

Namespacing:

- **Funções** `put`/`get`/`delete`: Usam namespacing automático (chave + ident)
- **Funções** `*_global`: Ignoram namespacing (chave pura)

Funções Disponíveis

1. `store.put(chave, valor)`

Armazena um valor no store com namespacing automático.

Parâmetros:

- **chave** (string): Identificador para o valor (será prefixado com o ident)
- **valor** (qualquer tipo Lua): Valor a ser armazenado (string, número, booleano, tabela, nil)

Retorno:

- **nil**: A função não retorna valor

Comportamento:

1. Prefixa a chave com o identificador de execução atual (se disponível)
2. Armazena o valor associado à chave prefixada
3. Registra automaticamente o timestamp atual (UTC)
4. Sobrescreve qualquer valor existente com a mesma chave
5. Persiste automaticamente em disco

Exemplo de Uso:

```
-- Armazenar dados específicos do script atual
store.put("ultima_execucao", os.time())
store.put("contador_falhas", 0)
store.put("config", {
    timeout = 30,
    intervalo = 60,
    alertas = true
})

-- Dados serão armazenados com prefixo do ident
-- Se EXEC_IDENT = "monitor-cpu", a chave se torna "monitor-cpu:ultima_execucao"
```

2. `store.get(chave)`

Recupera um valor do store com namespacing automático.

Parâmetros:

- **chave** (string): Chave do valor a ser recuperado (será prefixada com o ident)

Retorno:

- **tuple**: `(valor, timestamp)` onde:
  - **valor** (qualquer tipo ou nil): Valor armazenado, ou `nil` se a chave não existir
  - **timestamp** (número ou nil): Timestamp Unix (segundos) da última atualização, ou `nil` se a chave não existir

Comportamento:

1. Prefixa a chave com o identificador de execução atual
2. Retorna o valor e timestamp se a chave existir
3. Retorna `(nil, nil)` se a chave não existir
4. Preserva o tipo original do valor armazenado
5. O timestamp é em segundos desde a epoch Unix (UTC)

Exemplo de Uso:

```
-- Recuperar dados do script atual
local ultima_exec, timestamp = store.get("ultima_execucao")
if ultima_exec then
    local diferenca = os.time() - timestamp
    print("Última execução há", diferenca, "segundos")
end

-- Recuperar configuração
local config, ts = store.get("configuracao")
if config then
    print("Configuração carregada (atualizada em", os.date("%H:%M:%S", ts), ")")
    for chave, valor in pairs(config) do
        print("  ", chave, "=", valor)
    end
end

-- Verificar existência com valor padrão
local limite, _ = store.get("limite_temperatura")
limite = limite or 70  -- Valor padrão se não configurado
```

3. `store.delete(chave)`

Remove uma entrada do store com namespacing automático.

Parâmetros:

- **chave** (string): Chave da entrada a ser removida (será prefixada com o ident)

Retorno:

- **nil**: A função não retorna valor

Comportamento:

1. Prefixa a chave com o identificador de execução atual
2. Remove completamente a entrada do store
3. Não faz nada se a chave não existir
4. Libera o espaço em memória e disco
5. Operação atômica e thread-safe

Exemplo de Uso:

```
-- Remover dados específicos do script
store.delete("cache_temporario")
store.delete("dados_processados")

-- Limpar todos os dados do script atual
local function limpar_dados_script()
    -- Nota: Não há listagem direta de chaves
    -- É necessário conhecer as chaves usadas
    local chaves_conhecidas = {
        "configuracao",
        "cache",
        "estado",
        "historico",
        "lock_backup"
    }

    for _, chave in ipairs(chaves_conhecidas) do
        store.delete(chave)
        log.debug("Removido:", chave)
```

```
    end
end

-- Remover após processamento
local function processar_e_limpar(chave)
    local dados, timestamp = store.get(chave)

    if dados then
        -- Processar dados
        local resultado = processar_dados(dados)

        -- Remover após processamento
        store.delete(chave)
        log.info("Dados processados e removidos:", chave)

        return resultado
    else
        log.warn("Chave não encontrada:", chave)
        return nil
    end
end
```

4. `store.put_global(chave, valor)`

Armazena um valor no store SEM namespacing (chave global).

Parâmetros:

- **chave** (string): Identificador global para o valor (não é prefixado)
- **valor** (qualquer tipo Lua): Valor a ser armazenado

Retorno:

- **nil**: A função não retorna valor

Comportamento:

1. Armazena o valor com a chave exata fornecida
2. Não aplica prefixo do identificador de execução
3. Compartilhado entre todos os scripts (global)
4. Persiste em disco

5. Sobrescreve qualquer valor existente com a mesma chave

Exemplo de Uso:

```
-- Armazenar dados globais compartilhados
store.put_global("versao_agente", "2.5.1")
store.put_global("ultima_atualizacao", os.time())
store.put_global("config_global", {
    timezone = "America/Sao_Paulo",
    log_level = "info",
    retencao_logs = 30  -- dias
})

-- Dados serão acessíveis por todos os scripts
-- Chave exata: "versao_agente" (sem prefixo)
```

5. `store.get_global(chave)`

Recupera um valor do store SEM namespacing (chave global).

Parâmetros:

- **chave** (string): Chave global do valor a ser recuperado (não é prefixada)

Retorno:

- **tuple**: `(valor, timestamp)` onde:
  - **valor** (qualquer tipo ou nil): Valor armazenado, ou `nil` se a chave não existir
  - **timestamp** (número ou nil): Timestamp Unix (segundos) da última atualização, ou `nil` se a chave não existir

Comportamento:

1. Busca o valor usando a chave exata fornecida (sem prefixo)
2. Retorna o valor e timestamp se a chave existir
3. Retorna `(nil, nil)` se a chave não existir
4. Preserva o tipo original do valor armazenado
5. O timestamp é em segundos desde a epoch Unix (UTC)
6. Acessa dados compartilhados por todos os scripts

Exemplo de Uso:

```
-- Recuperar dados globais compartilhados
local versao, ts_versao = store.get_global("versao_agente")
if versao then
    print("Versão do agente:", versao, "(atualizada em", os.date("%Y-%m-%d %H:%M:%S",
ts_versao), ")")
end

-- Recuperar configuração global
local config_global, ts_config = store.get_global("config_global")
if config_global then
    print("Configuração global carregada:")
    for chave, valor in pairs(config_global) do
        print("  ", chave, "=", valor)
    end
end

-- Verificar e usar valor padrão para configuração global
local timezone, _ = store.get_global("timezone")
timezone = timezone or "UTC"  -- Valor padrão se não configurado
print("Timezone configurado:", timezone)

-- Verificar existência de flag global
local manutencao, ts_manutencao = store.get_global("modo_manutencao")
if manutencao then
    log.warn("Sistema em modo de manutenção desde", os.date("%H:%M:%S", ts_manutencao))
    -- Pular execuções não críticas
    return
end
```

Módulo String

O módulo **string** fornece funções utilitárias para manipulação de strings em Lua, complementando as funções nativas da linguagem.

Importante: Este módulo estende a tabela `string` padrão do Lua, portanto as funções são acessadas através da tabela global `string` (ex: `string.split()`).

Funções Disponíveis

1. `string.split(s, sep)`

Divide uma string em substrings com base em um separador especificado.

Parâmetros:

- **s** (string): A string original que será dividida
- **sep** (string): O separador usado para dividir a string

Retorno:

- **table**: Um array (tabela indexada numericamente) contendo todas as substrings resultantes da divisão

Comportamento:

- Se o separador for uma string vazia (`""`), a função retorna um array com cada caractere individual
- Se o separador não for encontrado na string, retorna um array contendo apenas a string original
- A divisão é feita em todas as ocorrências do separador

Exemplo de Uso:

```
-- Dividir uma string por vírgula
local resultado = string.split("maçã,banana,laranja,uva", ",")
-- resultado = {"maçã", "banana", "laranja", "uva"}
```

```
-- Dividir por espaço
local palavras = string.split("Olá mundo do Lua", " ")
-- palavras = {"Olá", "mundo", "do", "Lua"}

-- Dividir por nova linha (processamento de logs)
local linhas = string.split("2024-01-15 ERROR: Falha na conexão\n2024-01-15 INFO:
Reconectando", "\n")
-- linhas = {"2024-01-15 ERROR: Falha na conexão", "2024-01-15 INFO: Reconectando"}

-- Separador vazio (dividir em caracteres)
local chars = string.split("teste", "")
-- chars = {"t", "e", "s", "t", "e"}

-- Separador não encontrado
local unico = string.split("texto_sem_separador", "|")
-- unico = {"texto_sem_separador"}
```

2. `string.trim(s)`

Remove espaços em branco (whitespace) do início e do final de uma string.

Parâmetros:

- **s** (string): A string que será limpa

Retorno:

- **string**: A string original sem espaços em branco no início e no final

Comportamento:

- Remove espaços (` `), tabs (`\t`), novas linhas (`\n`), retornos de carro (`\r`)
- Não remove espaços no meio da string
- Retorna string vazia se a entrada for apenas espaços em branco

Exemplo de Uso:

```
-- Remover espaços extras
local limpo = string.trim("  texto com espaços  ")
-- limpo = "texto com espaços"
```

```
-- Limpar entrada de usuário
local entrada = "\t\n valor digitado \r\n"
local processado = string.trim(entrada)
-- processado = "valor digitado"

-- Processar configurações
local config_line = "   timeout = 30   "
local chave_valor = string.trim(config_line)
-- chave_valor = "timeout = 30"

-- String apenas com espaços
local vazio = string.trim("   \t\n   ")
-- vazio = ""
```

3. `string.starts(String, Start)`

Verifica se uma string começa com um prefixo específico.

Parâmetros:

- **String** (string): A string a ser verificada
- **Start** (string): O prefixo a ser procurado no início da string

Retorno:

- **boolean**: `true` se a string começar com o prefixo especificado, `false` caso contrário

Comportamento:

- A comparação é case-sensitive (diferencia maiúsculas de minúsculas)
- Retorna `true` se o prefixo for uma string vazia (`""`)
- Funciona com strings multibyte (UTF-8)

Exemplo de Uso:

```
-- Verificar se uma string começa com "http"
local url = "https://example.com"
local is_http = string.starts(url, "http")
-- is_http = true
```

```
-- Verificar prefixo em caminhos de arquivo
local path = "/var/log/app.log"
local is_absolute = string.starts(path, "/")
-- is_absolute = true

-- Verificar em processamento de logs
local log_line = "ERROR: Database connection failed"
local is_error = string.starts(log_line, "ERROR:")
-- is_error = true

-- Prefixo vazio sempre retorna true
local always_true = string.starts("qualquer string", "")
-- always_true = true

-- Case-sensitive
local case_check = string.starts("Hello World", "hello")
-- case_check = false (diferencia maiúsculas/minúsculas)
```

4. `string.ends(string, end)`

Verifica se uma string termina com um sufixo específico.

Parâmetros:

- **string** (string): A string a ser verificada
- **end** (string): O sufixo a ser procurado no final da string

Retorno:

- **boolean**: `true` se a string terminar com o sufixo especificado, `false` caso contrário

Comportamento:

- A comparação é case-sensitive (diferencia maiúsculas de minúsculas)
- Retorna `true` se o sufixo for uma string vazia (`""`)
- Funciona com strings multibyte (UTF-8)

Exemplo de Uso:

```
-- Verificar extensão de arquivo
local filename = "document.pdf"
```

```lua
local is_pdf = string.ends(filename, ".pdf")
-- is_pdf = true

-- Verificar sufixo em URLs
local url = "https://api.example.com/v1/data.json"
local is_json = string.ends(url, ".json")
-- is_json = true

-- Verificar terminação em strings de log
local log_entry = "Process completed successfully."
local is_success = string.ends(log_entry, "successfully.")
-- is_success = true

-- Sufixo vazio sempre retorna true
local always_true = string.ends("qualquer string", "")
-- always_true = true

-- Case-sensitive
local case_check = string.ends("Hello World", "world")
-- case_check = false (diferencia maiúsculas/minúsculas)
```

Vantagens do Módulo String do Monsta:

1. `string.split()` - Não existe nativamente no Lua, precisa ser implementada manualmente
2. `string.trim()` - Mais performática que implementações em Lua puro
3. **Consistência** - Mesma interface para todas as funções

Funções Complementares:

Use em conjunto com funções nativas do Lua:

- `string.find()` - Para buscas complexas
- `string.gsub()` - Para substituições
- `string.match()` - Para extração com padrões
- `string.gmatch()` - Para iteração sobre padrões

Exemplos Completos

Exemplo 1: Processamento de Log de Sistema

```
-- Simular leitura de log do sistema
local log_data = [[
Jan 15 10:30:45 servidor kernel: [12345.67890] CPU temperature: 65.5C
Jan 15 10:31:15 servidor sshd[1234]: Accepted password for user from 192.168.1.100
Jan 15 10:32:00 servidor crond[5678]: (root) CMD (/usr/bin/backup.sh)
]]

-- Processar cada linha do log
local function analisar_logs(logs)
    local eventos = {}

    for linha in string.gmatch(logs, "[^\n]+") do
        local linha_limpa = string.trim(linha)
        if linha_limpa ~= "" then
            -- Dividir por espaços (formato syslog)
            local partes = string.split(linha_limpa, " ")

            if #partes >= 5 then
                local evento = {
                    data = partes[1] .. " " .. partes[2] .. " " ..
partes[3],
                    host = partes[4],
                    servico = partes[5],
                    mensagem = table.concat(partes, " ", 6)
                }
                table.insert(eventos, evento)
            end
        end
    end

    return eventos
end

local eventos = analisar_logs(log_data)
```

Exemplo 2: Parser de Configuração Simples

```
-- Parser para arquivos de configuração no formato chave=valor
local function parse_config(conteudo)
    local config = {}

    for linha in string.gmatch(conteudo, "[^\n]+") do
        local linha_limpa = string.trim(linha)

        -- Ignorar linhas vazias e comentários
        if linha_limpa ~= "" and not string.starts(linha_limpa, "#") then
            -- Dividir por "="
            local partes = string.split(linha_limpa, "=")

            if #partes == 2 then
                local chave = string.trim(partes[1])
                local valor = string.trim(partes[2])

                -- Remover aspas se existirem
                if string.starts(valor, "\"") and string.ends(valor, "\"")
then
                    valor = string.sub(valor, 2, -2)
                end

                config[chave] = valor
            end
        end
    end

    return config
end

-- Exemplo de uso
local config_text = [[
# Configurações do monitor
hostname = "servidor-prod"
port = 8080
```

```
timeout = 30
debug = false
]]

local config = parse_config(config_text)
-- config.hostname = "servidor-prod", config.port = "8080", etc.
```

Módulo System

O módulo **system** fornece funções para obter informações sobre o sistema operacional, rede e ambiente de execução do agente. Este módulo é útil para scripts que precisam adaptar seu comportamento com base no ambiente, coletar informações de inventário ou realizar diagnósticos do sistema.

Características principais:

- Informações detalhadas do sistema operacional
- Nome do host e informações de rede
- Dados do agente (UUID, versão)
- Informações de kernel e distribuição
- Interface unificada para diferentes sistemas operacionais

Funções Disponíveis

1. `system.env()`

Retorna informações completas sobre o ambiente de execução do agente.

Parâmetros:

- Nenhum

Retorno:

- **tabela**: Estrutura com os seguintes campos:
  - `os_type` (string): Tipo do sistema operacional (ex: "linux", "windows", "macos")
  - `os_name` (string): Nome do sistema operacional (ex: "Ubuntu", "Windows 10", "macOS")
  - `os_version` (string): Versão do sistema operacional
  - `kernel_version` (string): Versão do kernel
  - `agent` (tabela ou nil): Informações do agente Monagent (se disponível):
    - `uuid` (string): UUID único do agente
    - `version` (string): Versão do Monagent

Exemplo de Uso:

```
-- Obter todas as informações do ambiente
local env_info = system.env()

print("Tipo do SO:", env_info.os_type)
print("Nome do SO:", env_info.os_name)
print("Versão do SO:", env_info.os_version)
print("Versão do Kernel:", env_info.kernel_version)

if env_info.agent then
    print("UUID do Agente:", env_info.agent.uuid)
    print("Versão do Monagent:", env_info.agent.version)
else
    print("Informações do agente não disponíveis")
end

-- Adaptar comportamento baseado no SO
local env = system.env()

if env.os_type == "linux" then
    -- Comandos específicos para Linux
    local saida, erro = process.exec("ps", "aux")
elseif env.os_type == "windows" then
    -- Comandos específicos para Windows
    local saida, erro = process.exec("tasklist")
elseif env.os_type == "macos" then
    -- Comandos específicos para macOS
    local saida, erro = process.exec("ps", "-ef")
end
```

2. `system.hostname()`

Retorna o nome do host do sistema.

Parâmetros:

- Nenhum

Retorno:

- **string**: Nome do host do sistema, ou "unknown" se não puder ser determinado

Exemplo de Uso:

```
-- Obter nome do host
local nome_host = system.hostname()
print("Nome do host:", nome_host)

-- Usar em identificadores únicos
local identificador = nome_host .. "_" .. os.date("%Y%m%d_%H%M%S")
print("Identificador único:", identificador)

-- Verificar se é um host específico
if nome_host == "servidor-producao" then
    log.info("Executando em servidor de produção")
    config.modo = "producao"
elseif string.find(nome_host:lower(), "test") then
    log.info("Executando em ambiente de teste")
    config.modo = "teste"
else
    log.info("Executando em host desconhecido:", nome_host)
    config.modo = "desenvolvimento"
end

-- Criar tags para métricas
local tags_metricas = {
    host = nome_host,
    ambiente = config.modo,
    timestamp = os.time()
}
```

3. `system.networks()`

Retorna uma lista de endereços IP de rede não-loopback do sistema.

Parâmetros:

- Nenhum

Retorno:

- **array de strings**: Lista de endereços IP das interfaces de rede (excluindo loopback)

Comportamento:

- Exclui endereços de loopback (127.0.0.1, ::1, etc.)
- Inclui endereços IPv4 e IPv6
- Atualiza a lista de interfaces antes de retornar
- Retorna apenas endereços IP, não nomes de interface

Exemplo de Uso:

```
-- Obter todos os endereços IP do sistema
local enderecos_ip = system.networks()

print("Endereços IP disponíveis:")
for i, ip in ipairs(enderecos_ip) do
    print("  " .. i .. ". " .. ip)
end

-- Identificar endereços IPv4 vs IPv6
local function classificar_enderecos()
    local enderecos = system.networks()
    local classificacao = {
        ipv4 = {},
        ipv6 = {},
        outros = {}
    }

    for _, ip in ipairs(enderecos) do
        if string.find(ip, ":") then
            -- IPv6
            table.insert(classificacao.ipv6, ip)
        elseif string.match(ip, "^%d+%.%d+%.%d+%.%d+$") then
            -- IPv4
            table.insert(classificacao.ipv4, ip)
        else
            -- Outro formato
            table.insert(classificacao.outros, ip)
        end
    end

    return classificacao
end
```

```
local ips = classificar_enderecos()
print("IPv4:", #ips.ipv4, "endereços")
print("IPv6:", #ips.ipv6, "endereços")

-- Encontrar endereço preferido para comunicação
local function encontrar_endereco_preferido()
    local enderecos = system.networks()

    -- Prioridade: IPv4 privado > IPv4 público > IPv6
    for _, ip in ipairs(enderecos) do
        -- Verificar se é IPv4 privado
        if string.match(ip, "^10%.") or
           string.match(ip, "^172%.%d?%d?%d?%.") or
           string.match(ip, "^192%.168%.") then
```

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}}
```

Módulo TCP

Este módulo fornece funcionalidades para conexões TCP e TCP+TLS a partir de scripts Lua. Oferece suporte para comunicação com sockets brutos, leitura/escrita de strings, JSON e pacotes JSON com tamanho prefixado.

O módulo **tcp** permite:

- Conexões TCP simples com timeout configurável
- Conexões TLS
- Leitura e escrita de strings simples
- Leitura e escrita de JSON
- Leitura e escrita de pacotes JSON com tamanho prefixado (para comunicação binária segura)
- Resolução DNS automática para conexões TLS

Funções Disponíveis

`tcp.connect(host, port, [timeout_secs])`

Estabelece uma conexão TCP simples com um servidor remoto.

Parâmetros:

- `host` (string): Endereço do host ou IP do servidor
- `port` (number): Porta TCP para conexão
- `timeout_secs` (number, opcional): Timeout em segundos (padrão: 5)

Valor de retorno:

- Objeto `Connection` que pode ser usado para leitura/escrita
- Lança erro em caso de falha ou timeout

Exemplo:

```
-- Conectar a um servidor na porta 8080 com timeout de 10 segundos
local conn = tcp.connect("servidor.exemplo.com", 8080, 10)
```

```
-- Conectar com timeout padrão (5 segundos)
local conn2 = tcp.connect("192.168.1.100", 3000)
```

`tcp.connect_tls(host, port, [timeout_secs])`

Estabelece uma conexão TLS (SSL) com um servidor remoto.

Parâmetros:

- `host` (string): Endereço do host do servidor
- `port` (number): Porta TLS para conexão
- `timeout_secs` (number, opcional): Timeout em segundos (padrão: 5)

Valor de retorno:

- Objeto `TlsConnection` que pode ser usado para leitura/escrita
- Lança erro em caso de falha ou timeout

Exemplo:

```
-- Conectar via TLS na porta 443
local conn = tcp.connect_tls("api.exemplo.com", 443)

-- Conectar com timeout personalizado
local conn2 = tcp.connect_tls("servidor-seguro.com", 8443, 15)
```

`tcp.send(data)`

Envia dados através da última conexão TCP aberta com a função `connect()`. Esta função é mantida para compatibilidade com versões anteriores. Para maior clareza e controle, prefira a forma `local conn = connect("host", porta)` e use os métodos do objeto de conexão retornado (ex: `conn:write_json_packet()`).

Parâmetros:

- `data` (string): Dados a serem enviados

Valor de retorno:

- `nil` em caso de sucesso
- Lança erro se não houver conexão ativa

Nota: Esta função usa a última conexão TCP aberta com a função `connect()`. É útil para scripts simples que mantêm uma única conexão.

Exemplo:

```
-- Primeiro estabelece uma conexão
local conn = tcp.connect("servidor.exemplo.com", 8080)


-- Envia dados
send("GET / HTTP/1.1\r\nHost: servidor.exemplo.com\r\n\r\n")
```

`tcp.recv()`

Recebe dados da última conexão TCP aberta com a função `connect()` e armazenada no registro Lua. Esta função é mantida para compatibilidade com versões anteriores. Para maior clareza e controle, prefira a forma `local conn = connect("host", porta)` e use os métodos do objeto de conexão retornado (ex: `conn:read_str()` ou `conn:read_json_packet()`).

Parâmetros:

- Nenhum

Valor de retorno:

- `string` com os dados recebidos
- Lança erro se não houver conexão ativa

Nota:

- Esta função usa a última conexão TCP aberta com a função `connect()`. É útil para scripts simples que mantêm uma única conexão.
- Lê até 8192 bytes por chamada. Para leitura completa, pode ser necessário chamar múltiplas vezes.

Exemplo:

```
-- Recebe resposta
local resposta = tcp.recv()
print("Resposta recebida:", resposta)
```

Métodos dos Objetos Connection

Os objetos retornados por `connect()` e `connect_tls()` possuem os seguintes métodos:

`conn:read_str()`

Lê uma string completa do socket até EOF.

Valor de retorno:

- `string` com os dados lidos
- Lança erro em caso de falha de leitura

Exemplo:

```
local conn = tcp.connect("servidor.exemplo.com", 8080)
local dados = conn:read_str()
print("Dados recebidos:", dados)
```

`conn:read_json()`

Lê uma string do socket e a interpreta como JSON.

Valor de retorno:

- Valor Lua decodificado do JSON
- Lança erro se os dados não forem JSON válido

Exemplo:

```
local conn = tcp.connect("api.exemplo.com", 3000)
local dados_json = conn:read_json()

-- Acessar dados decodificados
print("Status:", dados_json.status)
print("Mensagem:", dados_json.message)
```

`conn:read_json_packet()`

Lê um pacote JSON com tamanho prefixado (formato binário).

Formato do pacote:

1. 4 bytes (uint32): Tamanho dos dados JSON

2. N bytes: Dados JSON serializados

Limite de tamanho: 512KB (512.000 bytes)

Valor de retorno:

- Valor Lua decodificado do JSON
- Lança erro se o pacote for muito grande ou JSON inválido

Exemplo:

```
local conn = tcp.connect("servidor-binario.com", 9000)
local pacote = conn:read_json_packet()
print("Pacote recebido:", pacote)
```

`conn:write_json_packet(packet)`

Escreve um pacote JSON com tamanho prefixado (formato binário).

Parâmetros:

- `packet` (qualquer valor Lua): Dados a serem serializados como JSON

Valor de retorno:

- `nil` em caso de sucesso
- Lança erro em caso de falha de serialização ou escrita

Exemplo:

```
local conn = connect("servidor-binario.com", 9000)

-- Enviar um pacote JSON
local dados = {
    comando = "atualizar",
    id = 123,
    valores = {10, 20, 30}
}
conn:write_json_packet(dados)
```

Informações Adicionais

Timeout Configurável

Todas as funções de conexão aceitam timeout personalizado. Se não especificado, usa 5 segundos como padrão.

Resolução DNS Automática

A função `connect_tls()` resolve automaticamente o nome do host para um endereço IP antes de estabelecer a conexão.

Tamanho Máximo de Pacote

Pacotes JSON com tamanho prefixado têm um limite de 512KB para prevenir ataques de negação de serviço.

Exemplos de Uso

Comunicação HTTP Simples

```
function fazer_requisicao_http(host, porta, caminho)
    local conn = connect(host, porta, 10)

    -- Enviar requisição HTTP
    local requisicao = string.format(
        "GET %s HTTP/1.1\r\nHost: %s\r\nConnection: close\r\n\r\n",
        caminho, host
    )

    -- Usar o método send (conexão está no registro)
    send(requisicao)

    -- Ler resposta
    local resposta = recv()

    -- Extrair corpo da resposta (simplificado)
    local corpo = resposta:match("\r\n\r\n(.+)$")
```

```
    return corpo
end
```

Cliente de API JSON

```
function consultar_api_json(endpoint, dados)
    local conn = connect_tls("api.exemplo.com", 443)

    -- Enviar dados como pacote JSON
    conn:write_json_packet({
        endpoint = endpoint,
        dados = dados,
        timestamp = os.time()
    })

    -- Aguardar resposta
    local resposta = conn:read_json_packet()

    return resposta
end

-- Exemplo de uso
local resultado = consultar_api_json("/usuarios", {id = 123})
if resultado.success then
    print("Usuário:", resultado.usuario.nome)
end
```

Monitoramento de Serviço TCP

```
function verificar_servico_tcp(host, porta)
    local inicio = os.time()
    local sucesso, conn = pcall(connect, host, porta, 5)

    if sucesso then
        local tempo_resposta = os.time() - inicio
```

```
        -- Testar comunicação básica
        conn:write_json_packet({ping = true})
        local resposta = conn:read_json_packet()

        if resposta and resposta.pong then
            return {
                status = "online",
                tempo_resposta = tempo_resposta,
                versao = resposta.versao
            }
        end
    end

    return {
        status = "offline",
        erro = conn  -- conn contém a mensagem de erro quando pcall falha
    }
end
```

Comunicação Bidirecional

```
function chat_client(host, porta)
    local conn = connect(host, porta)

    -- Thread para receber mensagens
    local function receber_mensagens()
        while true do
            local mensagem = conn:read_json_packet()
            print("Recebido:", mensagem.texto)
        end
    end

    -- Thread para enviar mensagens (simplificado)
    local function enviar_mensagens()
        while true do
            io.write("Digite mensagem: ")
            local texto = io.read()
            if texto == "sair" then break end
```

```
            conn:write_json_packet({
                tipo = "mensagem",
                texto = texto,
                timestamp = os.time()
            })
        end
    end

    -- Em um ambiente real, isso seria feito com corrotinas
    -- Esta é uma simplificação para demonstração
end
```

Módulo Time

O módulo **time** fornece funções para manipulação de tempo e datas. Este módulo é útil para scripts que precisam medir intervalos, aguardar por períodos específicos, parsear datas e implementar timeouts em operações assíncronas.

Características principais:

- Timestamps Unix em segundos
- Parse de datas no formato RFC 3339
- Sleep assíncrono não-bloqueante
- Timeouts para operações assíncronas

Funções Disponíveis

1. `time.now()`

Retorna o timestamp Unix atual em segundos.

Parâmetros:

- Nenhum

Retorno:

- **número**: Timestamp Unix atual (segundos desde 1 de Janeiro de 1970, 00:00:00 UTC)

Comportamento:

- Retorna o tempo atual em UTC
- Precisão de segundos (não milissegundos)
- Mesmo valor retornado por `os.time()` em Lua padrão
- Disponível como função global e no módulo `time`

Exemplo de Uso:

```
-- Usando o módulo time
local timestamp_modulo = time.now()
```

```
print("Timestamp módulo:", timestamp_modulo)

-- Usando a função global (mesma função)
local timestamp_global = now()
print("Timestamp global:", timestamp_global)

-- Ambos retornam o mesmo valor
print("São iguais?", timestamp_global == timestamp_modulo)  -- true

-- Calcular duração de operação
local inicio = time.now()
-- ... operação demorada ...
local fim = time.now()
local duracao = fim - inicio
print("Operação levou", duracao, "segundos")

-- Registrar eventos com timestamp
local evento = {
    tipo = "login",
    usuario = "admin",
    timestamp = now(),
    origem = system.hostname()
}

-- Agendar execução futura
local proxima_execucao = time.now() + 3600  -- 1 hora a partir de agora
print("Próxima execução em:", os.date("%H:%M:%S", proxima_execucao))
```

2. `time.sleep(segundos)`

Aguarda assincronamente por um número específico de segundos.

Parâmetros:

- **segundos** (número): Número de segundos para aguardar

Retorno:

- **nil**: A função não retorna valor (após a espera)

Comportamento:

- Precisão dependente do scheduler do sistema

Exemplo de Uso:

```
-- Aguardar 5 segundos
await(time.sleep(5))
print("5 segundos se passaram")

-- Implementar polling com intervalo
local function poll_com_intervalo(url, intervalo_segundos, max_tentativas)
    for tentativa = 1, max_tentativas do
        log.info("Tentativa", tentativa, "de", max_tentativas)

        -- Fazer requisição
        local resposta, status = http.get(url)

        if status == 200 then
            log.info("Sucesso na tentativa", tentativa)
            return resposta
        end

        -- Aguardar antes da próxima tentativa
        if tentativa < max_tentativas then
            log.info("Aguardando", intervalo_segundos, "segundos...")
            time.sleep(intervalo_segundos)
        end
    end

    log.error("Todas as tentativas falharam")
    return nil
end

-- Uso
local dados = poll_com_intervalo("https://api.exemplo.com/status", 10, 6)

-- Controle de execução com backoff exponencial
local function executar_com_backoff(funcao, max_tentativas)
    local tentativa = 1
```

```
    while tentativa <= max_tentativas do
        local ok, resultado = pcall(funcao)

        if ok then
            return resultado
        end

        -- Backoff exponencial: 2^(tentativa-1) segundos
        local wait_time = math.pow(2, tentativa - 1)
        log.warn("Tentativa", tentativa, "falhou. Aguardando", wait_time, "segundos")

        if tentativa < max_tentativas then
            time.sleep(wait_time)
        end

        tentativa = tentativa + 1
    end

    error("Todas as tentativas falharam")
end
```

3. `time.parse(string_data)`

Parseia uma string de data/hora no formato RFC 3339 e retorna um objeto de data.

Parâmetros:

- **string_data** (string): Data/hora no formato RFC 3339 (ex: "2024-01-15T10:30:00Z")

Retorno:

- **objeto LuaDateTime**: Objeto com método `seconds()` que retorna timestamp Unix

Formato RFC 3339:

- `YYYY-MM-DDTHH:MM:SSZ` (UTC)
- `YYYY-MM-DDTHH:MM:SS+HH:MM` (com offset de fuso horário)
- Exemplos: "2024-01-15T10:30:00Z", "2024-01-15T07:30:00-03:00"

Exemplo de Uso:

```lua
-- Parsear data UTC
local data_utc = time.parse("2024-01-15T10:30:00Z")
local timestamp_utc = data_utc:seconds()
print("Timestamp UTC:", timestamp_utc)  -- 1705314600

-- Parsear data com offset de fuso
local data_local = time.parse("2024-01-15T07:30:00-03:00")
local timestamp_local = data_local:seconds()
print("Timestamp local:", timestamp_local)  -- 1705314600 (mesmo momento)

-- Converter para formato legível
local function formatar_data_rfc3339(timestamp)
    return os.date("%Y-%m-%dT%H:%M:%SZ", timestamp)
end

-- Calcular diferença entre datas
local data1 = time.parse("2024-01-15T10:30:00Z")
local data2 = time.parse("2024-01-15T11:45:00Z")
local diferenca = data2:seconds() - data1:seconds()
print("Diferença:", diferenca, "segundos (", diferenca/60, "minutos)")

-- Validar e parsear data de entrada
local function parsear_data_segura(data_string)
    local ok, data = pcall(time.parse, data_string)
    if ok then
        return data:seconds()
    else
        log.error("Data inválida:", data_string, "-", data)
        return nil
    end
end

-- Uso
local timestamp = parsear_data_segura("2024-01-15T10:30:00Z")
if timestamp then
    print("Data válida:", os.date("%d/%m/%Y %H:%M:%S", timestamp))
end
```

Módulo UUID

O módulo **uuid** fornece funções para geração de identificadores únicos universais (UUIDs) seguindo o padrão RFC 4122. Este módulo é útil para scripts que precisam criar identificadores únicos para rastreamento de transações, identificação de recursos, e casos que requerem unicidade garantida.

Características principais:

- Geração de UUID versão 4 (aleatório)
- Formato padrão RFC 4122 (8-4-4-4-12)
- Garantia de unicidade estatística
- Thread-safe e seguro para uso concorrente

Formato UUID v4:

- `xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx`
- Onde `x` é qualquer dígito hexadecimal
- `4` indica versão 4 (aleatório)
- `y` é 8, 9, A ou B (indicando variante)

Funções Disponíveis

1. `uuid.uuid4()`

Gera um UUID versão 4 (aleatório) no formato padrão RFC 4122.

Parâmetros:

- Nenhum

Retorno:

- **string**: UUID no formato `xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx`

Comportamento:

1. Gera 128 bits aleatórios usando gerador criptograficamente seguro
2. Define os bits de versão para 4 (0100)

3. Define os bits de variante para RFC 4122 (10)
4. Formata como string no padrão 8-4-4-4-12
5. Garante unicidade estatística (probabilidade de colisão extremamente baixa)

Exemplo de Uso:

```
-- Gerar um UUID simples
local id = uuid.uuid4()
print("UUID gerado:", id)
-- Exemplo: "f47ac10b-58cc-4372-a567-0e02b2c3d479"

-- Gerar múltiplos UUIDs
local ids = {}
for i = 1, 5 do
    ids[i] = uuid.uuid4()
    print("UUID", i, ":", ids[i])
end

-- Usar como identificador de transação
local transacao_id = uuid.uuid4()
local log_entry = {
    id = transacao_id,
    tipo = "pagamento",
    valor = 150.75,
    timestamp = now(),
    status = "processando"
}
print("Transação ID:", transacao_id)

-- Criar nomes de arquivo únicos
local nome_arquivo = "backup_" .. uuid.uuid4() .. ".tar.gz"
print("Arquivo de backup:", nome_arquivo)

-- Gerar token de sessão
local sessao_token = "sess_" .. string.sub(uuid.uuid4(), 1, 8)
print("Token de sessão:", sessao_token)
```

Informações Adicionais

Unicidade Garantida:

```
-- Probabilidade de colisão extremamente baixa
-- 2^128 possibilidades ≈ 3.4 × 10^38 UUIDs únicos
-- Mesmo gerando 1 bilhão de UUIDs por segundo,
-- levaria ~100 anos para ter 50% de chance de colisão

local function testar_unicidade(iteracoes)
    local uuids = {}
    local colisoes = 0

    for i = 1, iteracoes do
        local id = uuid.uuid4()

        if uuids[id] then
            colisoes = colisoes + 1
            log.error("COLISÃO DETECTADA!", "Iteração:", i, "ID:", id)
        else
            uuids[id] = true
        end
    end

    return {
        iteracoes = iteracoes,
        colisoes = colisoes,
        taxa_colisao = (colisoes / iteracoes) * 100
    }
end

-- Em testes práticos, colisões são virtualmente inexistentes
```

Formato Padrão:

```
-- UUID sempre no formato RFC 4122
local id = uuid.uuid4()
-- Formato: xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx
-- Exemplo: "f47ac10b-58cc-4372-a567-0e02b2c3d479"
```

```
-- Validar formato
local function validar_uuid(uuid_str)
    local padrao = "^%x%x%x%x%x%x%x%x-%x%x%x%x-4%x%x%x-[89ab]%x%x%x-
%x%x%x%x%x%x%x%x%x%x%x%x$"
    return string.match(uuid_str:lower(), padrao) ~= nil
end

print("UUID válido?", validar_uuid(id))  -- true
print("UUID válido?", validar_uuid("invalid"))  -- false
```

Exemplos de Uso

1. Nomes de Arquivos Temporários:

```
-- Gerar nomes de arquivos únicos para processamento
local function criar_arquivo_temporario(extensao, dados)
    local nome_arquivo = "tmp_" .. uuid.uuid4() .. "." .. (extensao or "tmp")
    local caminho = "/tmp/" .. nome_arquivo

    -- Escrever dados
    local arquivo = io.open(caminho, "w")
    if arquivo then
        arquivo:write(dados)
        arquivo:close()

        -- Registrar para limpeza
        local arquivos_temp, _ = store.get("arquivos_temporarios") or {}
        table.insert(arquivos_temp, {
            caminho = caminho,
            criado_em = now(),
            expira_em = now() + 3600  -- 1 hora
        })
        store.put("arquivos_temporarios", arquivos_temp)

        return caminho
    else
```

```
        return nil, "Não foi possível criar arquivo"
    end
end

-- Limpar arquivos temporários antigos
local function limpar_arquivos_temporarios()
    local arquivos_temp, _ = store.get("arquivos_temporarios") or {}
    local removidos = 0
    local agora = now()

    for i = #arquivos_temp, 1, -1 do
        local arquivo = arquivos_temp[i]

        if agora > arquivo.expira_em then
            -- Tentar remover arquivo
            local ok, _ = pcall(os.remove, arquivo.caminho)
            if ok then
                table.remove(arquivos_temp, i)
                removidos = removidos + 1
                log.debug("Arquivo temporário removido", "Caminho:",
arquivo.caminho)
            end
        end
    end

    store.put("arquivos_temporarios", arquivos_temp)
    return removidos
end
```

Módulo WMI

O módulo **wmi** fornece funções para interagir com o Windows Management Instrumentation (WMI), permitindo consultar informações do sistema, hardware, software e configurações em máquinas Windows remotas ou locais. Este módulo suporta tanto a execução cross-platform (usando o utilitário `wmic`) quanto a execução nativa no Windows (usando a API COM).

Funções Disponíveis

1. `wmi.buildwql(instance_id, table, fields)`

Constrói uma consulta WQL (WMI Query Language) a partir de parâmetros simplificados.

Parâmetros:

- `instance_id` (string opcional): Identificador de instância no formato "Tabela|Campo|Valor"
- `table` (string opcional): Nome da tabela WMI (usado quando não há instance_id)
- `fields` (tabela): Lista de campos a serem selecionados

Retorno:

- `string`: Consulta WQL formatada

Exceções:

- Lança erro se nenhum dos parâmetros `instance_id` ou `table` for fornecido
- Lança erro se `instance_id` estiver em formato inválido

Exemplos:

```
-- Consulta simples a uma tabela
local wql1 = wmi.buildwql(nil, "Win32_OperatingSystem", {"Caption", "Version", "BuildNumber"})
-- Resultado: "select Caption,Version,BuildNumber from Win32_OperatingSystem"

-- Consulta a uma instância específica
local wql2 = wmi.buildwql("Win32_Process|Name|explorer.exe", nil, {"ProcessId",
```

```
"WorkingSetSize"})
-- Resultado: "select ProcessId,WorkingSetSize from Win32_Process where Name =
\"explorer.exe\""

-- Consulta com múltiplos campos
local wql3 = wmi.buildwql(nil, "Win32_LogicalDisk", {"DeviceID", "Size", "FreeSpace",
"FileSystem"})
-- Resultado: "select DeviceID,Size,FreeSpace,FileSystem from Win32_LogicalDisk"
```

2. `wmi.exec(config, wql, namespace)`

Executa uma consulta WMI usando o utilitário `wmic`

Parâmetros:

- `config` (tabela): Configuração de conexão contendo:
  - `net.address` (string): Endereço IP ou hostname do alvo
  - `wmi.username` (string): Nome de usuário para autenticação
  - `wmi.password` (string): Senha para autenticação
  - `wmi.timeout` (número opcional, padrão: 10): Timeout em segundos
- `wql` (string): Consulta WQL a ser executada
- `namespace` (string opcional): Namespace WMI (padrão: "root\cimv2")

Retorno:

- `tabela`: Array de resultados, onde cada elemento é uma tabela com pares campo-valor

Nota: Em vez de criar manualmente uma tabela de configuração, você pode usar a tabela global `params` que já contém todos os campos necessários (`net.address`, `wmi.username`, `wmi.password`, etc.). Esta tabela é automaticamente disponibilizada pelo sistema quando o script é executado no contexto de um dispositivo gerenciado.

Exemplo usando `params`:

```
-- A tabela 'device' já contém as credenciais e endereço do dispositivo alvo
local success, results = pcall(function()
    return wmi.exec(params, wql, "root\\cimv2")
end)
```

Exemplo prático:

```
-- Consulta simplificada usando a tabela device
local wql = "select Caption,Version from Win32_OperatingSystem"
```

```
local results = wmi.exec(device, wql)

-- Para consultas locais no próprio dispositivo
if device["net.address"] == "127.0.0.1" or device["net.address"] == "localhost" then
    -- Pode-se usar exec_native para melhor performance
    if wmi.exec_native then
        results = wmi.exec_native(device, wql)
    end
end
```

Exceções:

- Lança erro se a conexão falhar
- Lança erro "Timeout" se a consulta exceder o tempo limite
- Lança erro se o utilitário `wmic` retornar código de erro

Exemplos:

```
-- Configuração básica
local config = {
    ["net.address"] = "192.168.1.100",
    ["wmi.username"] = "Administrator",
    ["wmi.password"] = "senha123",
    ["wmi.timeout"] = 15
}

-- Consulta informações do sistema operacional
local wql = "select Caption,Version,BuildNumber,OSArchitecture from Win32_OperatingSystem"
local success, results = pcall(function()
    return wmi.exec(config, wql, "root\\cimv2")
end)

if success then
    for _, row in ipairs(results) do
        print("Sistema: " .. row.Caption)
        print("Versão: " .. row.Version)
        print("Build: " .. row.BuildNumber)
        print("Arquitetura: " .. row.OSArchitecture)
    end
else
    print("Erro na consulta WMI: " .. results)
```

```
end

-- Consulta processos em execução
local process_wql = "select Name,ProcessId,WorkingSetSize,CommandLine from Win32_Process"
local process_results = wmi.exec(config, process_wql)

-- Consulta discos lógicos
local disk_wql = [[
select DeviceID,Size,FreeSpace,FileSystem
from Win32_LogicalDisk
where DriveType = 3
]]
local disk_results = wmi.exec(config, disk_wql, nil)  -- namespace padrão
```

3. `wmi.exec_native(config, wql, namespace)` (Apenas no agente Windows)

Executa uma consulta WMI usando a API nativa do Windows (COM). Esta função está disponível apenas em sistemas Windows e oferece melhor performance e integração.

Parâmetros:

- `config` (tabela): Configuração de conexão (ignorada na execução local)
- `wql` (string): Consulta WQL a ser executada
- `namespace` (string opcional): Namespace WMI

Retorno:

- `tabela`: Array de resultados, onde cada elemento é uma tabela com pares campo-valor

Exceções:

- Lança erro se a API COM falhar
- Lança erro se a consulta for inválida

Exemplos:

```
-- Apenas funciona em Windows
if wmi.exec_native then
    -- Consulta local (config é ignorado)
```

```
    local config = nil  -- Deixar vazio para consultas do agente
    local wql = "select Name,Manufacturer,Model from Win32_ComputerSystem"

    local success, results = pcall(function()
        return wmi.exec_native(config, wql)
    end)

    if success and #results > 0 then
        local computer = results[1]
        print("Computador: " .. computer.Name)
        print("Fabricante: " .. computer.Manufacturer)
        print("Modelo: " .. computer.Model)
    end
end
```

Informações Adicionais

1. **Suporte Cross-Platform**

- A função `exec` usa o utilitário `wmic` que funciona em sistemas Linux
- Permite consultar máquinas Windows remotamente
- Esta funcionalidade é legada, pois versões mais recentes do Windows não permitem conexões WMI remotas.

2. **Execução Nativa no Windows**

- A função `exec_native` oferece melhor performance em sistemas Windows
- Não requer autenticação para consultas locais
- Usa a API COM do Windows diretamente

3. **Timeout Configurável**

- Timeout padrão de 10 segundos
- Configurável via parâmetro `wmi.timeout` na configuração

Melhores Práticas

1. **Otimização de Consultas**

```
-- RUIM: Seleciona todas as colunas
local bad_wql = "select * from Win32_Process"

-- BOM: Seleciona apenas colunas necessárias
local good_wql = "select Name,ProcessId,WorkingSetSize from Win32_Process"

-- MELHOR: Adiciona filtros para reduzir resultados
local best_wql = [[
select Name,ProcessId,WorkingSetSize
from Win32_Process
where WorkingSetSize > 10485760  -- > 10MB
]]
```

4. `wmi.exec(config, wql, namespace, replace_backslash)`

Versão estendida da função `exec` com suporte a timeout e opção para substituir barras invertidas.

Parâmetros:

- **config** (tabela): Configuração de conexão
- **wql** (string): Consulta WQL a ser executada
- **namespace** (string, opcional): Namespace WMI (usa `wmi._namespace` ou `params.wmiNamespace` se não especificado)
- **replace_backslash** (booleano, opcional): Se `true`, substitui `\` por `\\` na WQL (padrão: `true`)

Retorno:

- **tabela**: Resultados da consulta WMI

Comportamento:

- Suporta execução via probe WMI quando `params["wmi.type"] == 0`

- Para localhost (`127.0.0.1`), usa execução nativa

Exemplo de Uso:

```
-- Executar com timeout configurado
-- params.wmiTimeout = 10 (10 segundos)

local config = {
    address = "192.168.1.100",
    username = "administrator",
    password = "senha123"
}

local wql = "select Name, ProcessId, WorkingSetSize from Win32_Process"
local results = wmi.exec(config, wql, "root\\cimv2")

for _, process in ipairs(results) do
    print(string.format("Processo: %s (PID: %d, Memória: %d bytes)",
                        process.Name, process.ProcessId, process.WorkingSetSize))
end
```

5. `wmi.query(wmiobj, ...)`

Consulta simplificada a uma tabela WMI sem instância específica.

Parâmetros:

- **wmiobj** (string): Nome da tabela WMI
- **...** (strings): Campos a serem selecionados (pode incluir `namespace=...`)

Retorno:

- **qualquer tipo**: Valor único se apenas um campo for selecionado, tabela se múltiplos campos

Comportamento:

- Constrói WQL automaticamente com `wmi.buildwql`
- Armazena resultado internamente para uso com `prev` e `lapsed`
- Suporta especificação de namespace via `namespace=` no início dos campos

Exemplo de Uso:

```
-- Consultar informações do sistema operacional
local os_name = wmi.query("Win32_OperatingSystem", "Caption")
print("Sistema Operacional:", os_name)

-- Consultar múltiplos campos
local disk_info = wmi.query("Win32_LogicalDisk", "DeviceID", "Size", "FreeSpace")
for _, disk in ipairs(disk_info) do
    local used_percent = 100 - (disk.FreeSpace / disk.Size * 100)
    print(string.format("Disco %s: %.1f%% usado", disk.DeviceID, used_percent))
end

-- Consultar com namespace específico
local cluster_info = wmi.query("MSCluster_Cluster", "namespace=root\\MSCluster", "Name",
"State")
```

6. `wmi.queryinst(...)`

Consulta WMI para instância específica definida em `params.InstanceId`.

Parâmetros:

- **...** (strings): Campos a serem selecionados (pode incluir `namespace=...`)

Retorno:

- **qualquer tipo**: Valor único se apenas um campo for selecionado, tabela se múltiplos campos

Comportamento:

- Usa `params.InstanceId` para construir consulta de instância
- Lança erro se instância não for encontrada
- Armazena resultado em `_mem_store` para uso com `prev` e `lapsed`

Exemplo de Uso:

```
-- params.InstanceId = "Win32_Process|Name|explorer.exe"

-- Consultar informações do processo explorer.exe
local pid = wmi.queryinst("ProcessId")
local memory = wmi.queryinst("WorkingSetSize")
```

```
print(string.format("Explorer.exe - PID: %d, Memória: %d bytes", pid, memory))

-- Consultar múltiplos campos
local process_info = wmi.queryinst("ProcessId", "WorkingSetSize", "ThreadCount", "Priority")
```

7. `wmi.prev(wmiobj, ...)`

Obtém o valor anterior de uma consulta WMI.

Parâmetros:

- **wmiobj** (string): Nome da tabela WMI
- **...** (strings): Campos da consulta original

Retorno:

- **qualquer tipo**: Valor anterior armazenado

Comportamento:

- Reconstrói a WQL original
- Busca valor em `store.get("wmi.value." .. wql)`
- Retorna valor único se consulta original retornou único valor

Exemplo de Uso:

```
-- Obter valor atual
local current_memory = wmi.query("Win32_OperatingSystem", "TotalVisibleMemorySize")

-- Obter valor anterior
local previous_memory = wmi.prev("Win32_OperatingSystem", "TotalVisibleMemorySize")

-- Calcular diferença
local memory_diff = current_memory - previous_memory
print("Variação de memória:", memory_diff, "bytes")
```

8. `wmi.previnst(...)`

Obtém o valor anterior de uma consulta WMI de instância.

Parâmetros:

- **...** (strings): Campos da consulta original

Retorno:

- **qualquer tipo**: Valor anterior armazenado

Exemplo de Uso:

```
-- params.InstanceId = "Win32_Process|Name|svchost.exe"

-- Obter uso atual de CPU
local current_cpu = wmi.queryinst("PercentProcessorTime")

-- Obter uso anterior
local previous_cpu = wmi.previnst("PercentProcessorTime")

-- Calcular variação
local cpu_change = current_cpu - previous_cpu
print("Variação no uso de CPU:", cpu_change, "%")
```

9. `wmi.lapsed(wmiobj, ...)`

Obtém o tempo decorrido desde a última consulta WMI.

Parâmetros:

- **wmiobj** (string): Nome da tabela WMI
- **...** (strings): Campos da consulta original

Retorno:

- **número**: Tempo em segundos desde última consulta

Exemplo de Uso:

```
-- Calcular taxa de transferência de disco
local current_reads = wmi.query("Win32_PerfRawData_PerfDisk_LogicalDisk",
"DiskReadBytesPerSec")
local previous_reads = wmi.prev("Win32_PerfRawData_PerfDisk_LogicalDisk",
```

```
"DiskReadBytesPerSec")
local time_elapsed = wmi.lapsed("Win32_PerfRawData_PerfDisk_LogicalDisk",
"DiskReadBytesPerSec")

local read_rate = (current_reads - previous_reads) / time_elapsed
print("Taxa de leitura de disco:", read_rate, "bytes/segundo")
```

10. `wmi.lapsedinst(...)`

Obtém o tempo decorrido desde a última consulta WMI de instância.

Parâmetros:

- **...** (strings): Campos da consulta original

Retorno:

- **número**: Tempo em segundos desde última consulta

Exemplo de Uso:

```
-- params.InstanceId = "Win32_PerfRawData_PerfDisk_LogicalDisk|Name|C:"

-- Calcular taxa de escrita para disco C:
local current_writes = wmi.queryinst("DiskWriteBytesPerSec")
local previous_writes = wmi.previnst("DiskWriteBytesPerSec")
local time_elapsed = wmi.lapsedinst("DiskWriteBytesPerSec")

local write_rate = (current_writes - previous_writes) / time_elapsed
print("Taxa de escrita no disco C:", write_rate, "bytes/segundo")
```

11. `wmi.diff(typ, lhs, rhs)`

Calcula a diferença entre dois valores WMI, tratando rollover de contadores.

Parâmetros:

- **typ** (número): Tipo do contador (32 ou 64 bits)
- **lhs** (número): Valor atual
- **rhs** (número): Valor anterior

Retorno:

- **número**: Diferença entre os valores

Comportamento:

- Usa a mesma implementação que `snmp.diff`
- Trata rollover de contadores de 32 e 64 bits
- Sinaliza `RepeatPrevValue` se a diferença for negativa

Exemplo de Uso:

```
-- Calcular diferença para contador de 64 bits
local current_bytes = wmi.queryinst("DiskReadBytesPerSec")
local prev_bytes = wmi.previnst("DiskReadBytesPerSec")
local bytes_diff = wmi.diff(64, current_bytes, prev_bytes)

print("Bytes lidos desde última leitura:", bytes_diff)
```

Exemplos Completos

Monitoramento de Processo Específico:

```
-- Configurar instância para monitorar processo específico
-- params.InstanceId = "Win32_PerfRawData_PerfProc_Process|Name|chrome.exe"

-- Obter métricas atuais
local cpu_usage = wmi.queryinst("PercentProcessorTime")
local memory_usage = wmi.queryinst("WorkingSetPrivate")
local thread_count = wmi.queryinst("ThreadCount")

-- Calcular variações
local prev_cpu = wmi.previnst("PercentProcessorTime")
local prev_memory = wmi.previnst("WorkingSetPrivate")
local time_elapsed = wmi.lapsedinst("PercentProcessorTime")

local cpu_delta = wmi.diff(32, cpu_usage, prev_cpu)
local memory_delta = memory_usage - prev_memory
```

```
print(string.format("Chrome.exe - CPU: %d%%, Memória: %d bytes, Threads:
%d",
                    cpu_delta / time_elapsed, memory_delta, thread_count))
```

Inventário de Hardware com Cache:

```
-- Função para obter informações de hardware com cache
local function get_hardware_info()
    local cache_key = "hardware_info_" .. params.device.address
    local cached = cache.get(cache_key)

    if cached then
        return cached
    end

    -- Coletar informações diversas
    local hardware_info = {
        os = wmi.query("Win32_OperatingSystem", "Caption", "Version", "BuildNumber"),
        cpu = wmi.query("Win32_Processor", "Name", "NumberOfCores", "MaxClockSpeed"),
        memory = wmi.query("Win32_ComputerSystem", "TotalPhysicalMemory"),
        disks = wmi.query("Win32_LogicalDisk", "DeviceID", "Size", "FreeSpace",
"FileSystem")
    }

    -- Armazenar em cache por 1 hora
    cache.put(cache_key, hardware_info, 3600)

    return hardware_info
end

-- Usar informações em cache
local info = get_hardware_info()
print("Sistema:", info.os.Caption, info.os.Version)
print("Processador:", info.cpu.Name, "(" .. info.cpu.NumberOfCores .. " núcleos)")
print("Memória total:", info.memory / (1024*1024*1024), "GB")
```

Módulo WS

O módulo **ws** fornece funcionalidades para comunicação bidirecional em tempo real através do protocolo WebSocket. Este módulo é útil para scripts que precisam se conectar a serviços que utilizam WebSockets.

Características principais:

- Conexão WebSocket assíncrona
- Envio e recebimento de mensagens de texto
- Suporte a URLs seguras (wss://) e não seguras (ws://)
- Timeout implícito baseado no sistema

Protocolo WebSocket:

- Protocolo de comunicação full-duplex sobre uma única conexão TCP
- Ideal para aplicações em tempo real
- Suportado pela maioria dos navegadores e servidores modernos
- Menor overhead comparado a HTTP polling

Funções Disponíveis

1. `ws.send_recv(url, dados)`

Estabelece uma conexão WebSocket, envia uma mensagem e aguarda uma resposta.

Parâmetros:

- **url** (string): URL do servidor WebSocket (ex: "ws://exemplo.com/socket", "wss://exemplo.com/ws")
- **dados** (string): Mensagem de texto a ser enviada para o servidor

Retorno:

- **string**: Resposta de texto recebida do servidor WebSocket

Comportamento:

1. Estabelece conexão WebSocket com o servidor especificado

2. Envia a mensagem de texto fornecida
3. Aguarda uma resposta do servidor
4. Retorna a primeira mensagem de texto recebida
5. Fecha a conexão após receber a resposta
6. Lança erro se receber mensagem binária ou se não houver resposta

Erros Comuns:

- `"ws binary messages not supported"` - Servidor enviou mensagem binária (não suportada)
- `"no response from web socket"` - Servidor não respondeu
- Erros de conexão (servidor offline, URL inválida, etc.)

Exemplo de Uso:

```
-- Conexão básica com servidor WebSocket
local url = "ws://echo.websocket.org"
local mensagem = "Olá, WebSocket!"
local resposta = ws.send_recv(url, mensagem)
print("Resposta do servidor:", resposta)
-- Saída: "Olá, WebSocket!" (servidor echo)

-- Conexão segura (wss://)
local url_segura = "wss://servidor.producao.com/ws"
local dados = json.encode({acao = "ping", timestamp = now()})
local resposta = ws.send_recv(url_segura, dados)
print("Resposta segura:", resposta)

-- Com tratamento de erro
local function enviar_com_tratamento(url, dados)
    local ok, resposta = pcall(ws.send_recv, url, dados)
    if ok then
        return resposta
    else
        log.error("Erro no WebSocket:", resposta)
        return nil
    end
end

-- Testar múltiplos servidores
local servidores = {
    "ws://servidor1.com/ws",
```

```
    "ws://servidor2.com/socket",
    "ws://backup.servidor.com/ws"
}

for _, servidor in ipairs(servidores) do
    local resposta = enviar_com_tratamento(servidor, "ping")
    if resposta then
        print("Servidor", servidor, "respondendo")
        break
    end
end
```

Informações Adicionais

Conexão Segura (wss://):

```
-- O módulo suporta tanto ws:// quanto wss://
local conexoes = {
    "ws://localhost:8080/ws",      -- Não seguro (HTTP)
    "wss://servidor.com/ws",       -- Seguro (HTTPS)
    "ws://192.168.1.100:3000/ws",  -- Local network
}

for _, url in ipairs(conexoes) do
    local ok, resposta = pcall(ws.send_recv, url, "ping")
    if ok then
        print("Conexão bem-sucedida:", url)
    else
        print("Falha na conexão:", url, "-", resposta)
    end
end
```

Mensagens de Texto Apenas:

```
-- O módulo só suporta mensagens de texto
-- Mensagens binárias retornam erro
```

```
local function enviar_dados_seguros(url, dados)
    -- Converter dados para JSON (texto)
    local dados_json = json.encode(dados)

    local resposta = ws.send_recv(url, dados_json)

    -- Parsear resposta JSON
    return json.decode(resposta)
end

-- Exemplo com dados complexos
local dados_complexos = {
    usuarios = {
        {id = 1, nome = "Alice", ativo = true},
        {id = 2, nome = "Bob", ativo = false}
    },
    metricas = {
        cpu = 45.6,
        memoria = 78.3,
        timestamp = now()
    }
}

local resposta = enviar_dados_seguros("wss://api.empresa.com/ws", dados_complexos)
```