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

Revision #2
Created 3 February 2026 16:38:43 by Marc
Updated 3 February 2026 16:41:15 by Marc