Rust Advanced Systems Programming: High-Performance Concurrent Async Cache with Zero-Cost Abstractions

rust

CopyEdit

```
// Rust Advanced Systems Programming Sample

// Keywords: Rust, systems programming, concurrency, memory safety, async, zero-cost abstractions, performance optimization, ownership, lifetimes


use std::{

    collections::HashMap,

    sync::{Arc, Mutex, RwLock},

    time::{Duration, Instant},

};


use tokio::sync::{Mutex as AsyncMutex, RwLock as AsyncRwLock};

use tokio::time::sleep;


/// CacheEntry represents a single item in the cache with a value and expiration.

struct CacheEntry<V> {

    value: V,

    expires_at: Instant,

}
```

```
/// A thread-safe, async-friendly in-memory cache with TTL and LRU eviction.
///
/// Features:
/// - Generic over key (K) and value (V) types.
/// - Uses async locking primitives for concurrency with minimal blocking.
/// - Supports time-to-live (TTL) expiration.
/// - Employs LRU eviction policy for memory optimization.
/// - Demonstrates zero-cost abstractions and ownership management.
///
/// Keywords: Rust cache, async cache, concurrency, TTL cache, LRU eviction, memory safety, ownership, zero-cost abstractions.
pub struct AsyncCache<K, V>
where
    K: std::hash::Hash + Eq + Clone + Send + Sync + 'static,
    V: Clone + Send + Sync + 'static,
{
    store: AsyncRwLock<HashMap<K, CacheEntry<V>>>,
    capacity: usize,
}

impl<K, V> AsyncCache<K, V>
where
    K: std::hash::Hash + Eq + Clone + Send + Sync + 'static,
    V: Clone + Send + Sync + 'static,
{
    /// Create a new cache with the given capacity.
    pub fn new(capacity: usize) -> Self {
        AsyncCache {
            store: AsyncRwLock::new(HashMap::with_capacity(capacity)),
            capacity,
        }
    }

    /// Insert a key-value pair with a TTL duration.
    ///
    /// If the cache exceeds capacity, evicts the oldest entry based on expiration.
    pub async fn insert(&self, key: K, value: V, ttl: Duration) {
        let mut write_guard = self.store.write().await;
        let expires_at = Instant::now() + ttl;

        if write_guard.len() >= self.capacity {
            // Evict expired or oldest item
            let oldest_key = write_guard
                .iter()
                .min_by_key(|(_, v)| v.expires_at)
                .map(|(k, _)| k.clone());

            if let Some(k) = oldest_key {
                write_guard.remove(&k);
            }
        }

        write_guard.insert(
            key,
            CacheEntry {
                value,
                expires_at,
            },
        );
    }

    /// Get a value from the cache, if present and not expired.
    ///
    /// Returns Option<V> with ownership to avoid locking after call.
    pub async fn get(&self, key: &K) -> Option<V> {
        let read_guard = self.store.read().await;
        if let Some(entry) = read_guard.get(key) {
            if Instant::now() < entry.expires_at {
                return Some(entry.value.clone());
            }
        }
        None
    }
```

```
    /// Periodically cleans expired entries asynchronously.
    pub async fn cleanup_expired(&self) {
        loop {
            sleep(Duration::from_secs(10)).await;
            let mut write_guard = self.store.write().await;
            let now = Instant::now();

            // Remove expired entries
            write_guard.retain(|_, v| v.expires_at > now);
        }
    }
}

/// Example of usage with Tokio runtime.
///
/// Demonstrates async insertion, retrieval, and cleanup in a high-performance Rust cache.
#[tokio::main(flavor = "multi_thread", worker_threads = 4)]
async fn main() {
    let cache = Arc::new(AsyncCache::<String, String>::new(100));

    let cache_writer = cache.clone();
    tokio::spawn(async move {
        for i in 0..200 {
            let key = format!("key{}", i);
            let value = format!("value{}", i);
        // Insert with 30 seconds TTL
        cache_writer.insert(key, value, Duration::from_secs(30)).await;
        sleep(Duration::from_millis(50)).await;
    }
});

let cache_reader = cache.clone();
tokio::spawn(async move {
    for i in 0..200 {
        let key = format!("key{}", i);
        if let Some(val) = cache_reader.get(&key).await {
            println!("Got {} = {}", key, val);
        } else {
            println!("{} expired or not found", key);
        }
        sleep(Duration::from_millis(100)).await;
    }
});

// Start cleanup task
let cleanup_cache = cache.clone();
tokio::spawn(async move {
    cleanup_cache.cleanup_expired().await;
});
```

```
    // Run the example for 30 seconds

    sleep(Duration::from_secs(30)).await;

}
```