Appunti 19-01

Il problema di partenza

Il sistema faceva troppe query al database. Praticamente ogni volta che si cercava un ospite, si interrogava il database anche se magari si era già cercata quella stessa persona poco prima. È come andare in biblioteca ogni volta per prendere lo stesso libro invece di tenerlo sulla scrivania mentre lo si usa. Questo causava rallentamenti e caricava inutilmente il server del database.[html]

Il profiling è stato implementato per capire dove il programma perdeva tempo. È stata creata una classe chiamata `ProfilingMonitor` che conta due metriche fondamentali: quante query vengono eseguite sul database e quante righe vengono caricate. Niente di complicato, solo due variabili statiche che si incrementano ogni volta che si tocca il database. Così si può verificare con dati concreti se il caching sta funzionando davvero oppure no.

Il concetto di caching

Il caching in sostanza significa tenere i dati in memoria invece di andare ogni volta a cercarli nel database. È come quando si prepara il caffè e si lascia lo zucchero sul tavolo invece di rimetterlo sempre nell'armadietto. Il problema è che la memoria RAM non è infinita, quindi bisogna decidere cosa tenere e cosa no, e quando rimuovere elementi per fare spazio.

Nel progetto sono state implementate due strategie diverse. Per le Room ha senso tenere tutto in memoria perché sono poche, circa una ventina. Per i Guest invece non si può fare lo stesso, sono centinaia e continuano ad aumentare. Quindi servivano due approcci completamente diversi: cache totale per le Room, cache parziale per i Guest.

La classe PartialCache generica

È stata creata una classe generica `PartialCache<X extends Entity>` riutilizzabile per qualsiasi entità del sistema. Quando si scrive `<X extends Entity>` si sta specificando che X può essere Entity o qualsiasi classe che ne eredita. Questo è vantaggioso

perché poi si possono chiamare metodi come `getId()` senza problemi, sapendo che esistono perché Entity li definisce.

La cache ha una dimensione massima configurabile, impostata a 1000 elementi per i Guest. Quando si riempie, viene rimosso l'elemento più vecchio. È una politica FIFO (First In First Out), come la coda alla posta. I primi elementi che entrano sono i primi che escono quando non c'è più spazio disponibile.

Un aspetto importante riguarda la prevenzione dei duplicati. Inizialmente non era stato considerato e la cache si riempiva con lo stesso ospite aggiunto multiple volte, sprecando memoria preziosa. È stato quindi aggiunto un controllo con il metodo `contains()` prima di aggiungere un elemento. Se l'elemento è già presente, non viene aggiunto di nuovo. Semplice ma molto efficace per mantenere la cache pulita.

La logica di ricerca con findById

Il metodo fondamentale è `findById()` nel repository dei Guest. La logica implementata funziona così: prima si cerca in cache, se l'elemento viene trovato lo si restituisce immediatamente senza toccare il database. Se non viene trovato in cache, solo allora si interroga il database, ma appena viene recuperato lo si aggiunge alla cache per renderlo disponibile per le prossime ricerche.

Una decisione importante riguarda i metodi che restituiscono liste di risultati, come la ricerca per cognome o per città. Quando si cercano gli ospiti di Milano e se ne trovano trenta nel database, sarebbe uno spreco caricarli e poi non metterli in cache. Quindi tutti i risultati vengono aggiunti alla cache. In questo modo se subito dopo qualcuno cerca uno di questi ospiti per ID, il dato è già disponibile in memoria e non serve nessuna query aggiuntiva.

Questa strategia si è rivelata particolarmente efficace. Nei test si è visto che cercando tutti gli ospiti di Milano, i trenta risultati finiscono tutti in cache. Poi cercando tre di questi per ID, non viene eseguita nemmeno una query al database. Il risparmio è notevole.

Gestione di Insert, Update e Delete con la cache

Queste operazioni richiedono attenzione particolare per mantenere la coerenza tra cache e database. Quando si inserisce un nuovo Guest nel database, viene anche

aggiunto alla cache immediatamente. Altrimenti se subito dopo qualcuno lo cerca per ID, non lo trova in cache e si deve fare una query inutile che il caching dovrebbe proprio evitare.

Per l'update la situazione è più complessa. Bisogna aggiornare il database ma anche sincronizzare la cache, che potrebbe contenere la vecchia versione dell'oggetto. La soluzione implementata prevede di rimuovere la vecchia versione dalla cache e aggiungere la nuova. Tuttavia c'è un'ottimizzazione: si controlla prima con `contains()` se la vecchia versione è effettivamente in cache. Se non c'è, non si perde tempo a cercare di rimuoverla, ma si aggiunge comunque la nuova versione così è pronta per accessi futuri.

Lo stesso ragionamento vale per la delete. Prima si controlla se il Guest è in cache con `contains()`, e solo in quel caso lo si rimuove. Si evitano così operazioni inutili su elementi non presenti.

Le Room con cache totale

Per le Room è stato adottato un approccio completamente diverso. Visto che sono poche e relativamente stabili, vengono caricate tutte in memoria nel costruttore del repository. Una query sola all'inizio, durante l'inizializzazione, e poi basta. Tutto il resto funziona completamente in memoria. Ogni volta che si cerca una stanza, non si tocca mai il database. Le performance sono eccellenti.

Naturalmente bisogna mantenere la cache sincronizzata quando si aggiunge, modifica o cancella una Room, ma la gestione è relativamente semplice. Si rimuove la vecchia versione e si aggiunge la nuova quando necessario. Il vantaggio di avere tutti i dati sempre in memoria supera ampiamente questo piccolo overhead.

I test di verifica

Sono stati implementati diversi test con JUnit per verificare che tutto funzionasse correttamente. Il primo test cerca lo stesso Guest sei volte consecutive e verifica che venga eseguita una sola query al database. Le altre cinque chiamate sono tutte cache hit, cioè trovano il dato già in memoria

Il secondo test è più interessante: cerca tutti gli ospiti di Milano (operazione che restituisce trenta risultati) (nel mio caso), poi cerca tre di questi ospiti specifici per

ID. Il risultato mostra che viene eseguita solo una query, quella iniziale per Milano. Le tre ricerche successive per ID sono tutte cache hit perché quegli ospiti sono stati aggiunti alla cache durante la prima ricerca. Questo conferma che la strategia di popolare la cache con i risultati delle ricerche multiple è corretta e vantaggiosa.

È stato implementato anche un test per verificare l'assenza di duplicati. Si cerca lo stesso Guest tre volte e si controlla che la dimensione della cache non cambi dopo la prima aggiunta. Senza il controllo con `contains()` nel metodo `addElement()`, ogni chiamata aggiungerebbe una copia duplicata. Con il controllo implementato, la cache rimane pulita e ottimizzata.

Confronto tra strategie di caching

Le due strategie adottate sono profondamente diverse e complementari. Per i Guest si usa una cache parziale perché il dataset è grande e potenzialmente illimitato. Potrebbero esserci migliaia di ospiti nel database. Con la cache parziale si tengono solo gli elementi più recenti, con un massimo configurato a mille elementi. Quando arriva l'elemento numero 1001, si rimuove automaticamente il più vecchio per fare spazio

Per le Room invece si usa una cache totale perché il dataset è piccolo e relativamente stabile. Le stanze di un B&B non cambiano continuamente e sono in numero limitato. Ha senso caricarle tutte una volta sola all'inizializzazione e tenerle sempre in memoria. Dopo il caricamento iniziale, tutte le operazioni di lettura hanno zero query al database e performance ottimali.

La scelta della strategia dipende sempre dal caso d'uso specifico. Non esiste una soluzione universale. Bisogna analizzare i dati: quanto sono grandi, quanto cambiano frequentemente, con quale pattern vengono acceduti, quanta memoria è disponibile.

Metriche e risultati misurati

I miglioramenti nelle performance sono stati quantificati con precisione usando il ProfilingMonitor. Se si cerca lo stesso ospite tre volte senza cache si eseguono tre query separate. Con la cache implementata se ne esegue una sola alla prima chiamata, le successive due sono cache hit. Risparmio del 66% sul numero di query.

In uno scenario più complesso, cercando gli ospiti di Milano e poi cercando tre specifici ospiti per ID, senza cache si eseguirebbero quattro query totali (una per la ricerca per città e tre per gli ID). Con la cache si esegue una sola query, quella iniziale, perché i tre ospiti cercati successivamente sono già in cache dalla prima ricerca. Risparmio del 75% sulle query.

Questi non sono numeri teorici ma dati reali misurati attraverso i contatori del ProfilingMonitor. Vedere i contatori che rimangono invariati quando si accede alla cache dimostra empiricamente l'efficacia del sistema implementato.

Concetti e principi applicati

L'implementazione ha richiesto l'applicazione di diversi principi della programmazione object-oriented. I generics permettono di creare una cache riutilizzabile per qualsiasi tipo di entità. L'incapsulamento nasconde la complessità della gestione della cache all'interno dei repository, così il resto del codice non deve preoccuparsene. Il polimorfismo permette di chiamare metodi come `getId()` che vengono risolti a runtime sul tipo specifico (Guest o Room).[mattepuffo]

Il principio di Single Responsibility è rispettato: la classe PartialCache si occupa solo di gestire la cache, ProfilingMonitor solo di monitorare le metriche, i repository solo della persistenza dei dati. Non ci sono responsabilità mescolate.

Il metodo `contains()` merita una menzione particolare perché sembra banale ma fa una differenza significativa. Senza questo controllo la cache si riempie di dati duplicati e sprecati. Con il controllo la cache rimane pulita, efficiente e predicibile.