Corso di Programmazione Concorrente

Lock-Splitting per Collezioni

Valter Crescenzi
http://crescenzi.inf.uniroma3.it

Sommario

- Sincronizzazione per Collezioni
  - operazioni aggregate e sincronizzate
  - visita tramite indicizzazione
  - client-side locking
  - snapshot
  - iteratori fail-fast
- Tecniche di *Lock-Splitting* per Collezioni
  - Es. su Mappe
  - Es. su Liste/Code
- Lock-Splitting in **`java.util.concurrent`**
  - **`ConcurrentMap`**
  - **`ConcurrentLinkedQueue`**
- Read/Write Lock

Sincronizzazione per Collezioni

- Premessa: prendiamo le liste di oggetti ad esempio di tutte le strutture dati per gestire collezioni
  - argomenti similari sono applicabili anche ad alberi, grafi, ed in generale altre strutture dati per la gestione di collezioni di oggetti
- La classe `ExpandableArray` che segue è un semplice esempio di classe completamente sincronizzata
  - ✔ una semplice generalizzazione degli array nativi per superare il vincolo di dover decidere la dimensione all'atto dell'istanziamento

ExpandableArray.java

```
class ExpandableArray {
  protected Object[] data;  // the elements
  protected int size = 0;   // the number of slots used

  public ExpandableArray(int cap) {
    data = new Object[cap];
  }

  public synchronized int size() { … }

  public synchronized Object get(int i)
    throws NoSuchElementException { … } //access by index

  public synchronized void add(Object x) { … }

  public synchronized void removeLast()
    throws NoSuchElementException { … }
}
```

✔ *Classe Completamente Sincronizzata*

Operazioni sugli Elementi nelle Collezioni

- Per discutere i problemi che la sincronizzazione solleva consideriamo due scenari molto comuni:
  - visita degli elementi di una collezione (>> *Iterator*)
  - esecuzione di operazioni specifiche sugli elementi ospitati

- Fintantoché si vogliono aggiungere nuovi metodi a livello di collezione senza compromettere la thread-safeness, basta marcarli come `synchronized`
  - ad es. `synchronized ExpandableArray.removeFirst()`
- Approccio non applicabile per metodi che modellano operazioni specifiche per il tipo degli oggetti ospitati:
  - ad es. *“somma tutti gli interi contenuti”, “trova il primo nome in ordine alfabetico”, “visualizza un widget grafico”, ...*

Operazioni Aggregate Sincronizzate

- Le operazioni sono specificate con l'interfaccia:

```
interface Procedure {
  void apply(Object obj);
}
```

- Quindi si prevede un metodo `applyToAll(Procedure p)`

```
class ExpandableArrayWithApply extends ExpandableArray {
  public ExpandableArrayWithApply(int cap) { super(cap); }
  synchronized void applyToAll(Procedure p) {
    for (int i = 0; i < size; ++i)
      p.apply(data[i]);
    }
  }
```

- Ad es. per stampare basta scrivere:

```
v.applyToAll(new Procedure() {
  public void apply(Object obj) {
    System.out.println(obj);
  }
});
```

In Java 8:

```
v.stream()
 .forEach(e -> System.out.println(e);
```

Considerazioni sulle Operazioni Aggregate Sincronizzate

- Tecnica semplice ed elegante, ma...
  - Richiede il possesso del lock della collezione per tutta la visita
  - Finisce per causare la completa serializzazione degli accessi
- Può risultare inaccettabile in presenza di
  - collezioni molto grandi
  - operazioni molto lunghe
  - numeri accessi concorrenti
- Alternative all'utilizzo di **`applyToAll()`** ...

Iterazione Tramite Indicizzazione

- Si consideri questa visita su `ExpandableArray`

```
for (int i=0; i<v.size(); i++) {
    System.out.println(v.get(i));
}
```

- Sebbene i metodi `size()` e `get()` siano marcati come `synchronized`, questa visita può essere oggetto di fenomeni di interferenza
  - la dimensione della collezione potrebbe cambiare dopo che finisca l'esecuzione di `size()` ma prima che lo stesso f.d.e. cominci l'esecuzione di `get()`
- Altre tecniche risolvono questo problema delegando la sincronizzazione agli utilizzatori della collezione e togliendola dalle responsabilità della classe contenitrice

Client-Side Locking (1)

- La responsabilità di ottenere l'utilizzo della collezione in mutua esclusione è a carico dell'utilizzatore
- Si consideri ancora una visita su `ExpandableArray`

```
for (int i=0; true; i++) {
   Object obj = null;
   synchronized (v) {
     if (i<v.size()) obj = v.get(i);
     else break;
   }
   System.out.println(obj);
}
```

Client-Side Locking (2)

- C'è comunque interferenza al livello della intera collezione
  - se la collezione supportasse operazioni che cambiano l’ordine degli elementi (ad es. `shuffle()`), lo stesso elemento potrebbe essere visitato più di una volta!

- L’unico modo per eliminare questo pericolo è effettuare un client-side locking che racchiuda tutta la visita, e non solo la singola iterazione
  - ma si torna al problema della *sincronizzazione completa*, ovvero la completa serializzazione degli accessi
  - Inoltre, dal punto di vista del client dell’API, questa soluzione è meno appetibile perché di più difficile utilizzo...

Snapshot (1)

- Un’alternativa utile in presenza di operazioni lunghe su collezioni piccole, oppure quando le modifiche sono rare: costruirsi un’istantanea della collezione da visitare

```
Object[] snapshot;
synchronized(v) {
  snapshot = new Object[v.size()];
  for(int i=0; i<snapshot.length; i++)
    snapshot[i]=v.get(i);
}
for(int i=0; snapshot.length; i++) {
  System.out.println(snapshot[i]);
}
```

Snapshot (2)

- A partire dalla versione 1.5 di java sono state introdotte due classi
  - **`java.util.CopyOnWriteArraySet`**
  - **`java.util.CopyOnWriteArrayList`**
- Creano una copia della collezione sottostante per ogni modifica
- Gli iteratori visitano una copia non modificabile della collezione, evitando ogni pericolo di interferenza e quindi di **`ConcurrentModificationException`**>>
- Conveniente quando le modifiche sono molto meno frequenti delle visite, riciclando gli snapshot
- Esempio di utilizzo: per mantenere un elenco di *listener* registrati presso di un componente grafico

Visita di Collezioni

- Per enumerare tutti gli elementi delle collezioni si usano gli iteratori; due metodi:
  - `boolean hasNext()`
  - `T next()`
- Ben noto Design Pattern *Iterator*: Un unico meccanismo di *enumerazione* degli elementi di una collezione con interfaccia indipendente dal tipo di collezione
  - creato da un apposito metodo *factory* della collezione. Es.
    - `Iterator<T> List.iterator()`
    - `Iterator<T> Set.iterator()`
- Ma la visita di una collezione, comporta l'accesso in lettura dei suoi elementi...
- Cosa succede se concorrentemente viene modificata?

ConcurrentModificationException

```
import java.util.*;
public class ConcurrentModificationExceptionTest {
  public static void main(String args[]) {
    List list = new ArrayList();
    Iterator it = list.iterator();
    list.add(new Object());
    it.next();
  }
}

$ java ConcurrentModificationExceptionTest
Exception in thread "main“
java.util.ConcurrentModificationException
  at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:448)
  at java.util.AbstractList$Itr.next(AbstractList.java:419)
  at ConcurrentModificationExceptionTest.main(ConcurrentModificationExceptionTest.java:9)
```

- Non si devono fare modifiche ad una collezione mentre la si sta visitando con un iteratore
- In caso contrario, l'iteratore risponderà, “alla prima occasione”, con una **`java.util.ConcurrentModificationException`**

Iteratori Ottimisti

- Le collezioni del *Java Collection Framework* che non sono state espressamente pensate per essere usate in un contesto concorrente, applicano una strategia *ottimistica*:
  - nessuna sincronizzazione
  - l'iteratore controlla ad ogni invocazione di metodo se la versione della collezione che sta visitando è cambiata rispetto a quella che lo ha creato e solleva una eccezione in caso affermativo
  - `java.util.ArrayList, LinkedList, HashSet, HashMap, TreeSet...`
- L'interferenza è rilevata con una tecnica basata sull'utilizzo di una *versione* della collezione
  - *versione*: numero incrementato ad ogni modifica della collezione
  - ogni iteratore conserva una copia della versione della collezione che era corrente *al momento della sua creazione*
  - prima di ogni operazione sull'iteratore, si controlla che la versione *al momento dell'operazione*, sia ancora allineata con quella alla creazione

Versioni ed Iteratori *fail-fast*

- Il *versionamento* della collezioni si utilizza per realizzare la semantica *fail-fast* tipica degli iteratori java

✔ in presenza di interferenza, meglio fallire il prima possibile per facilitare la diagnosi del problema che ritardarne gli effetti rendendoli ancora più difficili da correlare alla causa originale!

- Altra possibilità: effetture le modifiche attraverso l'iteratore stesso per renderlo “consapevole” delle modifiche alla collezione sottostante:
  - `Iterator.remove()`
    - rimuove l'ultimo elemento restituito dall'iteratore
    - allinea l’iteratore al numero di versione corrente

Utilizzo degli Iteratori *fail-fast*

- Gli iteratori *fail-fast* risultano semplici ed efficaci
- Ma servono *solo* per il debugging (leggere javadoc!)
  - Il numero di versione è rappresentato con precisione finita
  - `Integer.MAX_VALUE` cambiamenti sono "inosservabili"
  - *Non* bisogna basare la correttezza dei propri programmi sulle interferenze segnalate da un iteratore *fail-fast*
- Talvolta risultano eccessivamente conservativi
  - non distinguono le modifiche, né dove occorrono
- Alcune collezioni di `java.util.concurrent` offrono iteratori con semantiche più *lasche* e *scalabili* e quindi molto più adatte a contesti concorrenti

Collezioni e Sincronizzazione

- La *Tecnica delle Classi Completamente Sincronizzate* permette agevolmente di scrivere codice thread-safe
- Tuttavia, come nel caso dei monitor astratti, il grado di parallelismo può risultare compromesso
  - tutti i metodi sono eseguiti in mutua esclusione
- Esistono situazioni dove la perdita di parallelismo risulta inaccettabile, ovvero per collezioni:
  - di grandi dimensioni
  - accedute concorrentemente da molteplici thread
  - oggetto di lunghe operazioni
- Le collezioni, per diffusione ed importanza, meritano uno studio apposito per superare la *completa sincronizzazione*

Soluzioni
Non Completamente Sincronizzate

- Rinunciando alla sincronizzazione completa, è possibile creare implementazioni che consentano alcune seq. di interleaving altrimenti escluse
  - c'è interferenza! (a livello dell’intera collezione)
  - ma sarà “controllata”>>
- Approccio tanto più sensato quanto più risulta caratterizzabile (e prevedibile)
  - la tipologia delle operazioni di accesso (lettura/scrittura)
  - la frequenza delle operazioni

  esercitate dagli utilizzatori delle collezioni
- Vedi anche HWC2_THREAD (*lista_reader)*

Lock-Splitting per Collezioni

- Si passa da un unico lock che protegge l'intera collezione, a molteplici lock associati a *regioni* distinte in cui viene *suddivisa* la collezione
- Per accedere alla collezione è necessario acquisire i lock associati alle regioni degli accessi effettuati
- Risultano consentite s.e.a. in cui siano accedute concorrentemente *regioni* diverse della stessa collezione
- La “granularità” del locking su una collezione può essere decisa sulla base:
  - della natura degli accessi che vengono effettuati (lettura/scrittura) e dalla loro frequenza
  - compromesso tra *garanzie* desiderate e livello di parallelismo
  - esigenze dettate dallo specifico problema

Altro Esempio: Oltre le `Map` Completamente Sincronizzate

- In java 1.5 è stata introdotta **`java.util.concurrent.ConcurrentHashMap`** che affianca le precedenti versioni completamente sincronizzate dell'interfaccia **`java.util.`*`Map:`***
  - **`java.util.Hashtable`**
  - **`Collections.synchronizedMap(Map map)`**
- Queste implementazioni non sono *scalabili* all'aumentare del numero di thread che vi accedono
- I thread cominciano ad accodarsi per accedere serialmente alla mappa
  - accedono uno alla volta!

Lock-Splitting per **`java.util.Map`**

- **`java.util.concurrent.ConcurrentHashMap`** implementazione *non* completamente sincronizzata:
  - le operazioni di lettura non sono sincronizzate
  - gli iteratori non sollevano **`java.util.ConcurrentModificationException`**
  - si comportano “alla meglio”

- Lock a livello di *bucket*

  (elemento della tabella hash nel quale sono ospitati gli oggetti della *Map*; l'indice del bucket si ottiene applicando la funzione *hash* alle chiavi...)
  - costruttori per definire la “granularità” desiderata del locking
  - ✔ è suggerito l'uso di un numero di lock nell'ordine del numero di thread che vi accedono concorrentemente

Più Parallelismo, Meno Garanzie

- Origina fenomeni di interferenza (sull'intera collezione) *controllati*
  - Si può finire per leggere *porzioni* di uno stato della collezione che una volta ricomposte portano ad uno stato *complessivo* della collezione che non è mai esistito!
  - In molti problemi è perfettamente lecito e preferibile
    - es. del dizionario
  - In altri può risultare non accettabile
    - es. calcolo esatto per operazioni aggregate su collezioni
- N.B. Gli stati transienti visibili sono comunque ottenibili con una sequenza seriale di operazioni dell'interfaccia pubblica

Retrieval operations (including `get`) generally do not block, so may overlap with update operations (including `put` and `remove`). Retrievals reflect the results of the most recently *completed* update operations holding upon their onset. For aggregate operations such as `putAll` and `clear`, concurrent retrievals may reflect insertion or removal of only some entries. Similarly, Iterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration. They do *not* throw `ConcurrentModificationException`. However, iterators are designed to be used by only one thread at a time.

java.util.concurrent.ConcurrentMap

- Sono disponibili 4 operazioni *atomiche*
  - `V putIfAbsent(K key, V Value);`
  - `boolean remove(K key, V Value);`
  - `V replace(K key, V value);`
  - `boolean replace(K key, V oldValue, V newValue)`

  organizzate nell'interfaccia **`java.util.concurrent.`*`ConcurrentMap`***

  che non fanno parte di ***`java.util.Map`***

- N.B. Senza questi metodi atomici appositamente previsti, non sarebbe agevole lavorare con queste mappe: anche solo per eseguire un'operazione composita del tipo

```
If (!map.containsKey(k)) {
    map.put(k,v);
}
```

  si richiederebbe, di nuovo, una sincronizzazione esplicita a carico degli utilizzatori

Lock-Splitting per Liste e Code

- Consideriamo il caso di liste e code in presenza di una forte caratterizzazione delle operazioni attese
- Dettagliamo una tecnica di *lock-splitting* per un caso semplice ma di estrema utilità pratica

- Lista semplicemente collegata con politica FIFO
  - `put()`: inserimenti in coda
  - `poll()`: estrazioni dalla testa
  - si vogliono consentire estrazioni in testa ed inserimenti in coda concorrenti

- Le regioni sono la *testa* e la *coda* della lista

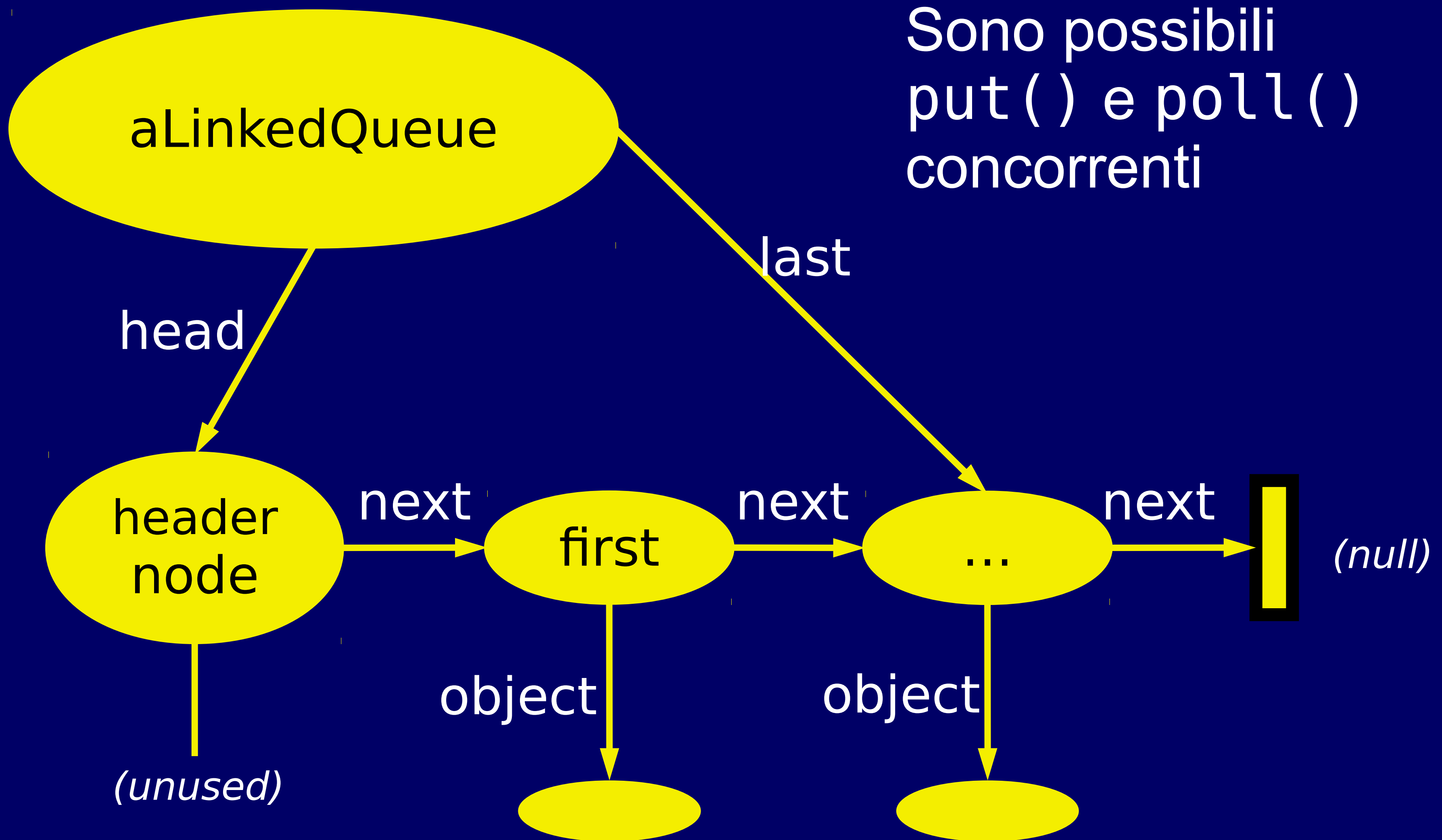
LinkedQueue.java (1)

```
class LinkedQueue {
  protected Node head = new Node(null);
  protected Node last = head;

  protected final Object pollLock = new Object();
  protected final Object putLock = new Object();

  public void put(Object x) {…}

  public Object poll() {…}

  static class Node {  // local node class for queue
    Object object;
    Node next = null;
    Node(Object x) { object = x; }
  }
}
```

LinkedQueue.java (2)

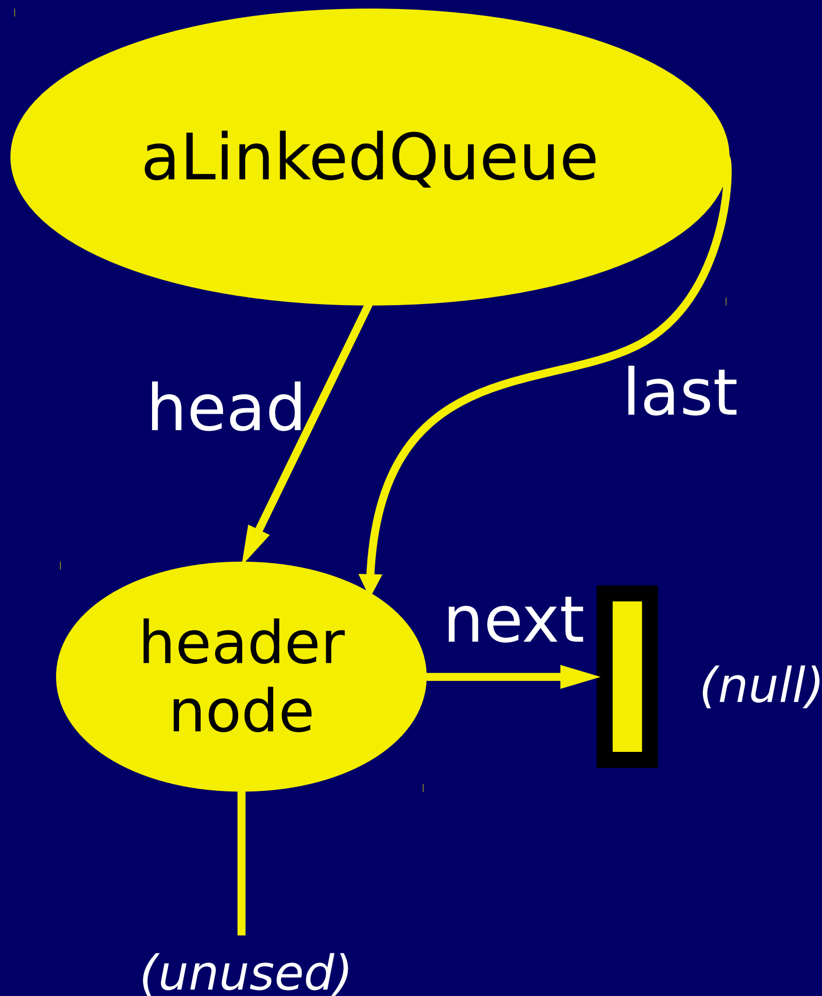
```
class LinkedQueue {…
  public void put(Object x) {
    Node node = new Node(x);
    synchronized (putLock) {     // insert at end of list
      synchronized (last) {
        last.next = node;        // extend list
        last = node;
      }
    }
  }
  public Object poll() {         // returns null if empty
    synchronized (pollLock) {
      synchronized (head) {
        Object x = null;
        Node first = head.next;  // get to first real node
        if (first != null) {
          x = first.object;
          first.object = null;   // forget old object
          head = first;          // first becomes new head
        }
        return x;
      }
    }}…}
```

Una Linked Queue non Vuota



Sono possibili `put()` e `poll()` concorrenti

Una Linked Queue Vuota



Non sono possibili una `put()` ed una `poll()` concorrenti

`LinkedQueue.java`: Conclusioni

- Grazie alla conoscenza della particolare semantica della coda è stato possibile raffinare la granularità della sincronizzazione
- `put()` e `poll()` possono essere eseguite concorrentemente perché interessano regioni diverse (rispett. la fine e l'inizio) della coda
- bisogna fare attenzione al caso limite di code vuote in cui inizio e fine coincidono
- **`java.util.concurrent.ConcurrentLinkedQueue`** è una implementazione thread-safe, di dimensione illimitata ed ispirata al codice appena visto
- tecniche simili sono alla base anche di ***`java.util.Deque`*** **>>**

Es. *Lock-Splitting*: Dizionario

- Tradizionale esempio di collezione di cui è possibile caratterizzare gli accessi
- Enorme e con numerosi accessi:
  - *frequenti* in lettura
  - *rari* in scrittura
- N.B. è necessario disciplinare *tutti* gli accessi
  (in quanto ci sono f.d.e. scrittori)

- Una soluzione completamente sincronizzata serializza gli accessi compromettendo le prestazioni
  - l'unico lock "blocca" tutto il dizionario
- *Lock-Splitting*: usiamo un lock per ogni "pagina"
  - i f.d.e. possono accedere concorrentemente a pagine distinte

Read/Write Lock (1)

- Strumento ideale per collezioni come quelle del *Dizionario*: grandi, letture frequenti, scritture rare
- Coppia di lock associati tra loro
  - uno, detto lock di lettura (read-lock), consente ai thread che lo possiedono di fare letture sulla risorsa protetta
  - l’altro, detto lock di scrittura (write-lock), consente ai thread che lo possiedono di fare scritture sulla risorsa protetta
- Utili per proteggere l’accesso a strutture condivise evitando fenomeni di interferenza senza compromettere il grado di parallelismo in presenza di
  - una maggioranza (per numero e durata) di operazioni di lettura
  - una minoranza (per numero e durata) di operazioni di scrittura

Read/Write Lock (2)

- *Molteplici* thread possono ottenere il read-lock concorrentemente
- *Un solo* thread alla volta può ottenere il write-lock
- Se un thread chiede il lock di lettura, può procedere *anche* se altri thread già lo possiedono ma solo se nessuno già possiede il lock in scrittura
- Se un thread chiede il lock in scrittura, può procedere solo quando ottiene l'accesso in mutua esclusione alla risorsa
  - tutti gli altri thread non possono disporre né del lock di scrittura né di quello in lettura

Read/Write Lock (3)

- Rispetto alla semplice mutua esclusione, il livello di parallelismo raggiungibile dipende fortemente dalla frequenza e dal tipo degli accessi
- Alcune delle scelte progettuali necessarie:
  - se sia scrittori che lettori sono in attesa passiva, chi preferire non appena la risorsa si libera da uno scrittore? e da un lettore?
  - il lock di scrittura è rientrante? e quello di lettura?
  - un lock di lettura può essere promosso a lock di scrittura? secondo quali politiche?
  - un lock di scrittura può essere retrocesso a lock di lettura? secondo quali politiche?
  - …

java.util.concurrent.locks.*ReadWriteLock*

- Fornisce due metodi factory per ottenere il lock di lettura e di scrittura di un medesimo Read/Write lock

java.util.concurrent.locks

Interface ReadWriteLock

All Known Implementing Classes:

ReentrantReadWriteLock

Method Summary	
Lock	**readLock**() Returns the lock used for reading.
Lock	**writeLock**() Returns the lock used for writing.

`java.util.concurrent.locks.` `ReentrantReadWriteLock` (1)

- Una possibile implementazione
  - una politica fair (opzionale) di gestione delle richieste
  - lock rientranti
  - locking interrompibile sia sul lock di lettura che di scrittura
  - lock in scrittura possono essere declassati a lock in lettura
  - i lock in scrittura supportano le variabili condizione

Constructor Summary
ReentrantReadWriteLock() Creates a new ReentrantReadWriteLock with default ordering properties.
ReentrantReadWriteLock(boolean fair) Creates a new ReentrantReadWriteLock with the given fairness policy.

java.util.concurrent.locks. ReentrantReadWriteLock (2)

Method Summary	
protected Thread	**getOwner()** Returns the thread that currently owns the write lock, or null if not owned.
protected Collection<Thread>	**getQueuedReaderThreads()** Returns a collection containing threads that may be waiting to acquire the read lock.
protected Collection<Thread>	**getQueuedWriterThreads()** Returns a collection containing threads that may be waiting to acquire the write lock.
protected Collection<Thread>	**getWaitingThreads(Condition** condition) Returns a collection containing those threads that may be waiting on the given condition associated with the write lock.
ReentrantReadWriteLock.ReadLock	**readLock()** Returns the lock used for reading.
ReentrantReadWriteLock.WriteLock	**writeLock()** Returns the lock used for writing.

Esempio: **ReadWriteLock** (1)

```
class RWDictionary {
   private final Map<String, Data> m =
        new TreeMap<String, Data>();
   private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();
   private final Lock r = rwl.readLock();
   private final Lock w = rwl.writeLock();
   public Data get(String key) {
     r.lock();
     try { return m.get(key); }
     finally { r.unlock(); }
   }
   public String[] allKeys() {
      r.lock();
      try { return m.keySet().toArray(); }
      finally { r.unlock(); }
   }
   …  // altri metodi a seguire
}
```

Esempio: **ReadWriteLock** (2)

```
class RWDictionary {
  …
  public Data put(String key, Data value) {
    w.lock();
    try { return m.put(key, value); }
    finally { w.unlock(); }
  }
  public void clear() {
    w.lock();
    try { m.clear(); }
    finally { w.unlock(); }
  }
}
```

Esercizi

Esercizio: creare una classe java che implementi le funzionalità di **`java.util.concurrent.locks.`*`ReadWriteLock`***. Tale classe rende possibile specificare che si vuole acquisire un lock in sola lettura oppure in sola scrittura. Mentre diversi thread possono acquisire concorrentemente il lock di sola lettura, solo un thread alla volta può acquisire il lock in scrittura. Eventuali altri thread richiedenti il lock (in lettura od in scrittura) dovrebbero essere sospesi in attesa passiva che il lock venga rilasciato.

Esercizio: analizzare le differenze tra i Read/Write Lock offerti dalla piattaforma java e quelli per thread descritti nello standard POSIX.