Summary of Common Java Concurrency Containers

Guide Java Java Concurrency About 3019 words About 10 minutes

Most of these containers provided by JDK are in `java.util.concurrent` the package.

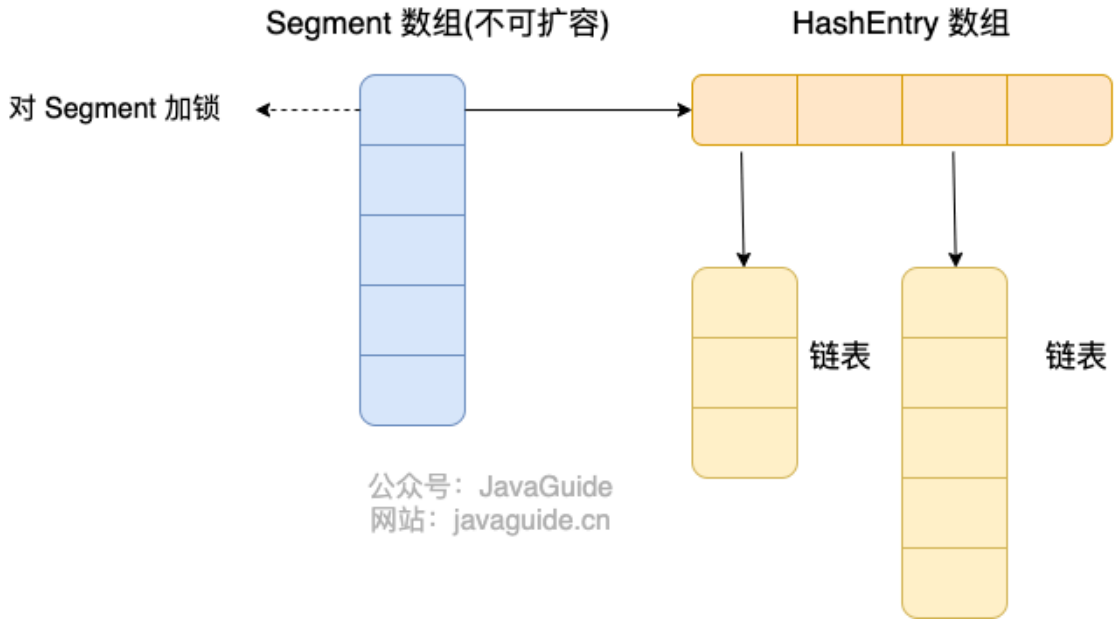
- **`ConcurrentHashMap`** : Thread-safe `HashMap`
- **`CopyOnWriteArrayList`** : Thread-safe `List` , performance is very good in situations where there are more reads than writes, far better than `Vector` .
- **`ConcurrentLinkedQueue`** : An efficient concurrent queue implemented using a linked list. `LinkedList` It can be considered a thread-safe, non-blocking queue.
- **`BlockingQueue`** : This is an interface that is implemented internally in the JDK through linked lists, arrays, and other methods. It represents a blocking queue and is very suitable for use as a channel for data sharing.
- **`ConcurrentSkipListMap`** : Implementation of skip list. This is a Map that uses the skip list data structure for fast search.

ConcurrentHashMap

As we know, `HashMap` is thread-unsafe. If used in concurrent scenarios, a common solution is to wrap with `Collections.synchronizedMap()` the method `HashMap` to make it thread-safe. However, this approach uses a global lock to synchronize concurrent access between different threads, which can lead to severe performance bottlenecks, especially in highly concurrent scenarios.

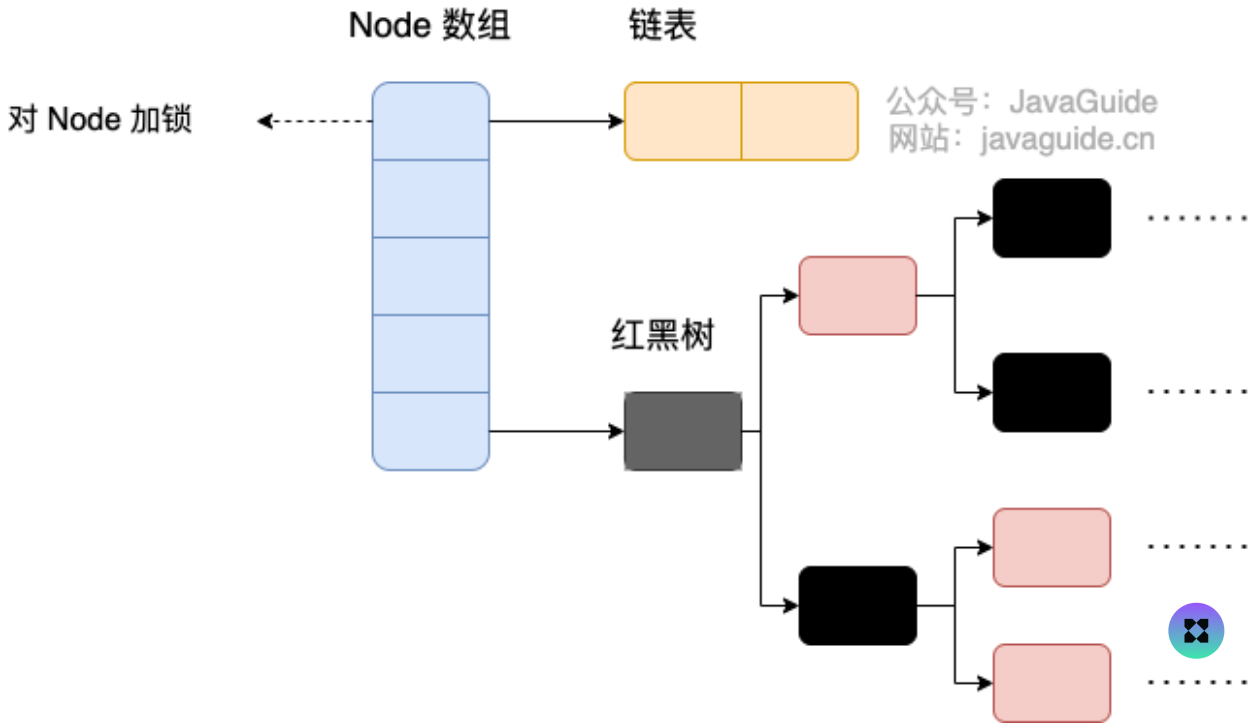
In order to solve this problem, `ConcurrentHashMap` came into being. As `HashMap` a thread-safe version of , it provides more efficient concurrent processing capabilities.

In JDK 1.7, `ConcurrentHashMap` the entire bucket array was segmented (`Segment` segmented locks). Each lock only locked a portion of the container data (see the diagram below). When multiple threads accessed data in different data segments in the container, there would be no lock contention, thereby improving the concurrent access rate.



In JDK 1.8, segmented locking `ConcurrentHashMap` was removed and concurrency safety was ensured. The data structure is similar to that of 1.8: array + linked list/red-black binary tree. Java 8 converts the linked list (with an addressing time complexity of O(N)) to a red-black tree (with an addressing time complexity of O(log(N)))) when the linked list length exceeds a certain threshold (8). `Segment` `Node + CAS + synchronized` `HashMap`

In Java 8, the lock granularity is finer, `synchronized` only locking the first node of the current linked list or red-black binary tree. In this way, as long as the hash does not conflict, no concurrency will occur, and the reading and writing of other nodes will not be affected, which greatly improves efficiency.



For `ConcurrentHashMap` a detailed introduction to , please see this article I wrote: `ConcurrentHashMap` Source Code Analysis .

CopyOnWriteArrayList

Prior to JDK 1.5, concurrency-safe `List` was the only choice `Vector` . However `Vector` , is an older, deprecated collection. `Vector` Implementations of add, delete, modify, and query methods were added `synchronized` . While this approach ensured synchronization, it effectively `Vector` locked the entire , forcing each method to acquire the lock before executing, resulting in very poor performance.

JDK1.5 introduced `Java.util.concurrent` the (JUC) package, which provides many thread-safe and concurrent performance containers, among which the only thread-safe `List` implementation is `CopyOnWriteArrayList` .

In most business scenarios, read operations often far outnumber write operations. Since read operations don't modify existing data, locking every read operation is a waste of resources. In contrast, we should allow multiple threads to access `List` internal data simultaneously; this is safe for read operations.

This approach `ReentrantReadWriteLock` is very similar to the design philosophy of read-write locks: reads and reads are mutually exclusive, reads and writes are mutually exclusive, and writes and writes are mutually exclusive (only reads and reads are mutually exclusive). `CopyOnWriteArrayList` This concept is further implemented. To maximize read performance, `CopyOnWriteArrayList` read operations in the lock are completely lock-free. Even more impressively, writes do not block reads; only writes and writes are mutually exclusive. This significantly improves read performance.

`CopyOnWriteArrayList` The core of thread safety lies in its use of the **Copy-On-Write** strategy, which `CopyOnWriteArrayList` can be seen from the name.

When the content needs to be modified (`add` , `set` , `remove` etc.) `CopyOnWriteArrayList` , the original array will not be modified directly. Instead, a copy of the underlying array will be created first, the copy array will be modified, and then the modified array will be assigned back after the modification. This ensures that the write operation will not affect the read operation.

For `CopyOnWriteArrayList` a detailed introduction to , please see this article I wrote: `CopyOnWriteArrayList` Source Code Analysis .

ConcurrentLinkedQueue

The thread-safe queues provided by Java `Queue` can be divided into **blocking queues** and **non-blocking queues** . A typical example of a blocking queue is `BlockingQueue` , and a typical example of a non-blocking queue is `ConcurrentLinkedQueue` . In practical applications, the choice of blocking queue or non-blocking queue should be based on actual needs. **Blocking queues can be implemented through locking, while non-blocking queues can be implemented through CAS operations.**

As the name suggests, `ConcurrentLinkedQueue` this queue uses a linked list as its data structure. `ConcurrentLinkedQueue` It's considered one of the best performing queues in highly concurrent environments. Its high performance is due to its complex internal implementation.

`ConcurrentLinkedQueue` We will not analyze the internal code. As long as we know that `ConcurrentLinkedQueue` the CAS non-blocking algorithm is mainly used to achieve thread safety.

`ConcurrentLinkedQueue` It is suitable for scenarios with relatively high performance requirements and where multiple threads are reading and writing queues simultaneously. That is, if the cost of locking the queue is high, it is suitable to use lock-free `ConcurrentLinkedQueue` instead.

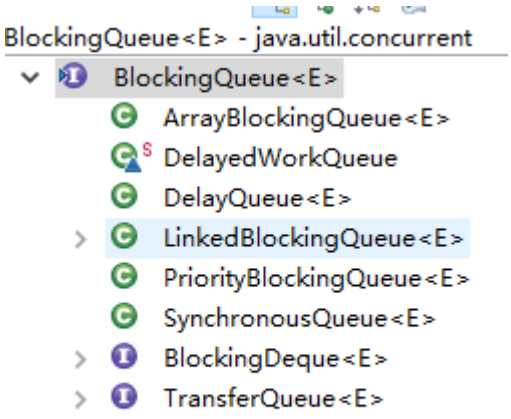
BlockingQueue

Introduction to BlockingQueue

We've already mentioned `ConcurrentLinkedQueue` as a high-performance non-blocking queue. Next, we'll discuss blocking queues `BlockingQueue` . Blocking queues (`BlockingQueue`) are widely used in producer-consumer problems because they `BlockingQueue` provide blocking insertion and removal methods. When the queue container is full, the producer thread is blocked until the queue is full; when the queue container is empty, the consumer thread is blocked until the queue is not empty.

`BlockingQueue` is an interface that inherits from `Queue` , so its implementation class also `Queue` be used as an implementation of , which in `Queue` turn inherits from `Collection` the interface. The following are `BlockingQueue` the relevant implementation

classes of :



The following mainly introduces three common `BlockingQueue` implementation classes: `ArrayBlockingQueue` , `LinkedBlockingQueue` , and `PriorityBlockingQueue` .

ArrayBlockingQueue

`ArrayBlockingQueue` It is `BlockingQueue` a bounded queue implementation class of the interface, and is implemented using an array at the bottom layer.

```java
public class ArrayBlockingQueue<E>
extends AbstractQueue<E>
implements BlockingQueue<E>, Serializable{}
```

`ArrayBlockingQueue` Once created, the capacity cannot be changed. Its concurrency control uses reentrant locks `ReentrantLock` , requiring both insert and read operations to acquire the lock. When the queue is full, attempting to put an element into the queue will cause the operation to block; attempting to retrieve an element from an empty queue will also block.

`ArrayBlockingQueue` By default, fairness in thread access to the queue is not guaranteed. Fairness means strict adherence to the absolute order in which threads wait, meaning that the thread waiting first receives access first `ArrayBlockingQueue` . Inequity, however, means that access `ArrayBlockingQueue` to queues does not strictly follow this order. It's possible that `ArrayBlockingQueue` a thread blocked for a long time may still be unable to access a queue when it becomes available `ArrayBlockingQueue` . Ensuring fairness typically reduces throughput. To achieve fairness `ArrayBlockingQueue` , use the following code:

```java
private static ArrayBlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<Integer>(10,true);
```

LinkedBlockingQueue

`LinkedBlockingQueue` The underlying blocking queue is implemented as **a singly linked list** . It can be used as either an unbounded or bounded queue, and it also meets the FIFO characteristics. `ArrayBlockingQueue` Compared with , it has higher throughput. To prevent `LinkedBlockingQueue` the capacity of from increasing rapidly and consuming a large amount of memory, `LinkedBlockingQueue` a size is usually specified when creating a . If not specified, the capacity is equal to `Integer.MAX_VALUE` .

Related construction methods:

```java
    /**
     *某种意义上的无界队列
     * Creates a {@code LinkedBlockingQueue} with a capacity of
     * {@link Integer#MAX_VALUE}.
     */
    public LinkedBlockingQueue() {
        this(Integer.MAX_VALUE);
    }

    /**
     *有界队列
     * Creates a {@code LinkedBlockingQueue} with the given (fixed) capacity.
     *
     * @param capacity the capacity of this queue
     * @throws IllegalArgumentException if {@code capacity} is not greater
     *         than zero
     */
    public LinkedBlockingQueue(int capacity) {
        if (capacity <= 0) throw new IllegalArgumentException();
        this.capacity = capacity;
        last = head = new Node<E>(null);
    }
```

PriorityBlockingQueue

`PriorityBlockingQueue` It is an unbounded blocking queue that supports priorities. By default, elements are sorted in natural order. You can also `compareTo()` specify the ordering rules by implementing methods in a custom class or by specifying the ordering rules via constructor parameters during initialization `Comparator` .

`PriorityBlockingQueue` Concurrency control uses reentrant locks `ReentrantLock` , and the queue is an unbounded queue (`ArrayBlockingQueue` it is a bounded queue, and you can also specify the maximum capacity of the queue `LinkedBlockingQueue` by passing it in the constructor , but you can only specify the initial queue size. When you insert elements later, **it will automatically expand if there is not enough space**). `capacity` `PriorityBlockingQueue`

Simply put, it's `PriorityQueue` a thread-safe version of . Null values cannot be inserted, and objects inserted into the queue must be of comparable size, otherwise `ClassCastException` an exception will be thrown. Its put method does not block insertions because it's an unbounded queue (the take method will block if the queue is empty).

Recommended article: "Interpreting Java Concurrent Queue BlockingQueue"

ConcurrentSkipListMap

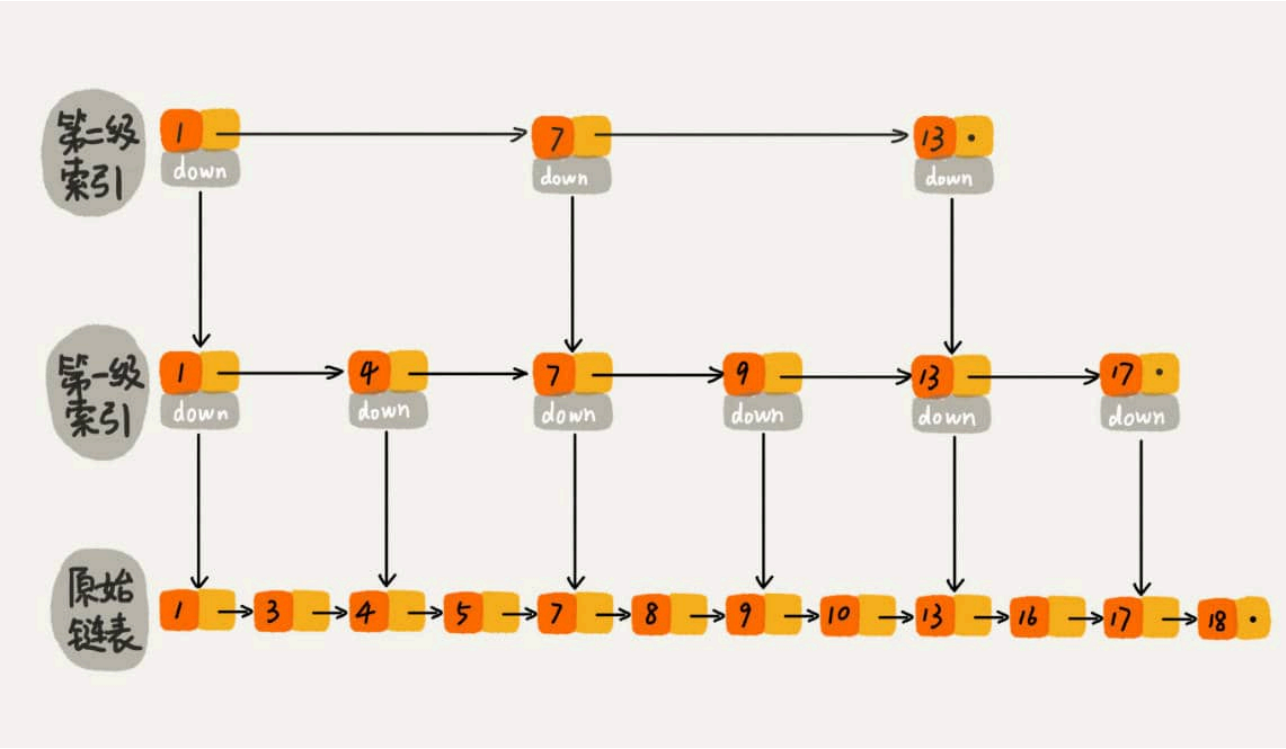
> The following content refers to the Geek Time column "The Beauty of Data Structures and Algorithms" and "Practical Java High Concurrency Programming".

To introduce it `ConcurrentSkipListMap` , let's first take a brief look at the jump table.

Even if a singly linked list is ordered, searching for data in it requires traversing the list from beginning to end, which is inherently inefficient. However, a skip list is a different approach. A skip list is a data structure that allows for fast searches, somewhat similar to a balanced tree. Both allow for fast element lookups. However, a key difference is that insertions and deletions in a balanced tree often require a global adjustment. Insertions and deletions in a skip list, on the other hand, require only local operations on the en' data structure. This offers the advantage that, in highly concurrent environments, a gl lock is required to ensure thread safety for the entire balanced tree. With a skip list, only

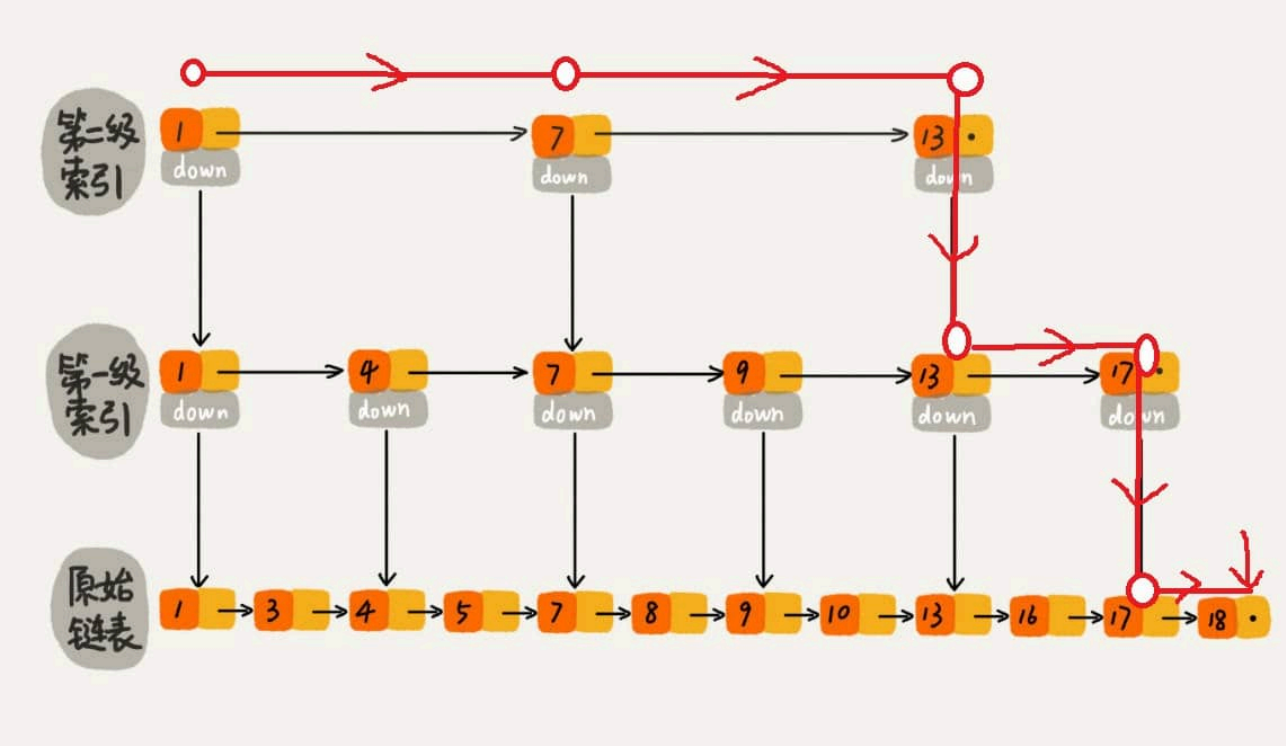
partial locks are required. This results in better performance in highly concurrent environments. Furthermore, the time complexity of a skip list is also **O(log n).** Therefore, among concurrent data structures, the JDK uses a skip list to implement a Map.

The essence of the jump list is to maintain multiple linked lists at the same time, and the linked lists are hierarchical.



The lowest level linked list maintains all the elements in the skip list, and each upper level linked list is a subset of the lower level.

All linked lists within a skip list are sorted. When searching, you start at the top-level linked list. If the element being searched is greater than the value in the current linked list, the search continues in the next linked list. This means that the search is performed in a jumpy manner. For example, in the figure above, we are searching for element 18 in the skip list.



When searching for 18, it used to require 18 traversals, but now it only requires 7. When the length of the linked list is relatively long, the improvement in search efficiency by building an index will be very obvious.

It is easy to see from the above that **the jump table is an algorithm that uses space to exchange time.**

Another difference between implementing a skip list `Map` and a hash algorithm `Map` is that a hash doesn't preserve the order of elements, while all elements in a skip list are sorted. Therefore, when traversing a skip list, you obtain an ordered result. Therefore, if your application requires order, a skip list is the best choice. The class that implements this data structure in the JDK is [`ConcurrentSkipListMap`].

refer to

- Practical Java High Concurrency Programming
- https://javadoop.com/post/java-concurrent-queue
- https://juejin.im/post/5aeebd02518825672f19c546

Recently Updated2024/12/29 12:24
Contributors: Snailclimb , Gene , IceDarron , SnailClimb , guide , Tan Jiuding , Guide , Mr.Hope