极客大学

极客大学 Java 进阶训练营
第 8 课
Java 并发编程（3）

KimmKing

Apache Dubbo/ShardingSphere PMC

个人介绍

Apache Dubbo/ShardingSphere PMC

前某集团高级技术总监/阿里架构师/某银行北京研发中心负责人

阿里云 MVP、腾讯 TVP、TGO 会员

10 多年研发管理和架构经验

熟悉海量并发低延迟交易系统的设计实现

目录

1. 常用线程安全类型*
2. 并发编程相关内容*
3. 并发编程经验总结*
4. 并发编程常见面试题
5. 第8课总结回顾与作业实践

1. 常用线程安全类型

JDK 基础数据类型与集合类

List：ArrayList、LinkedList、Vector、Stack

Set：LinkedSet、HashSet、TreeSet

Queue->Deque->LinkedList

Map：HashMap、LinkedHashMap、TreeMap

Dictionary->HashTable->Properties

线性数据结构都源于 Collection 接口，并且拥有迭代器

原生类型，数组类型，对象引用类型

ArrayList

基本特点：基于数组，便于按 index 访问，超过数组需要扩容，扩容成本较高

用途：大部分情况下操作一组数据都可以用 ArrayList

原理：使用数组模拟列表，默认大小10，扩容 x1.5，newCapacity = oldCapacity + (oldCapacity >> 1)

```
/**
 * The array buffer into which the elements of the ArrayList are stored.
 * The capacity of the ArrayList is the length of this array buffer. Any
 * empty ArrayList with elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
 * will be expanded to DEFAULT_CAPACITY when the first element is added.
 */
transient Object[] elementData; // non-private to simplify nested class access

/**
 * The size of the ArrayList (the number of elements it contains).
 *
 * @serial
 */
private int size;
```

安全问题：

1、写冲突：

- 两个写，相互操作冲突

2、读写冲突：

- 读，特别是 iterator 的时候，数据个数变了，拿到了非预期数据或者报错
- 产生 ConcurrentModificationException

LinkedList

基本特点：使用链表实现，无需扩容

用途：不知道容量，插入变动多的情况

原理：使用双向指针将所有节点连起来

安全问题：

1、写冲突：

- 两个写，相互操作冲突

2、读写冲突：

- 读，特别是 iterator 的时候，数据个数变了，拿到了非预期数据或者报错
- 产生 ConcurrentModificationException

```
transient int size = 0;

/**
 * Pointer to first node.
 * Invariant: (first == null && last == null) ||
 *            (first.prev == null && first.item != null)
 */
transient Node<E> first;

/**
 * Pointer to last node.
 * Invariant: (first == null && last == null) ||
 *            (last.next == null && last.item != null)
 */
transient Node<E> last;
```

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

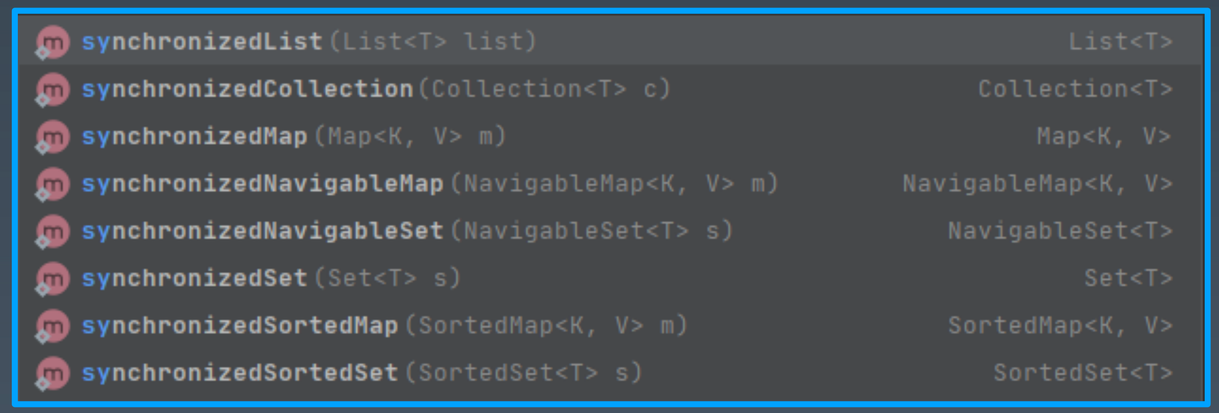
List线程安全的简单办法

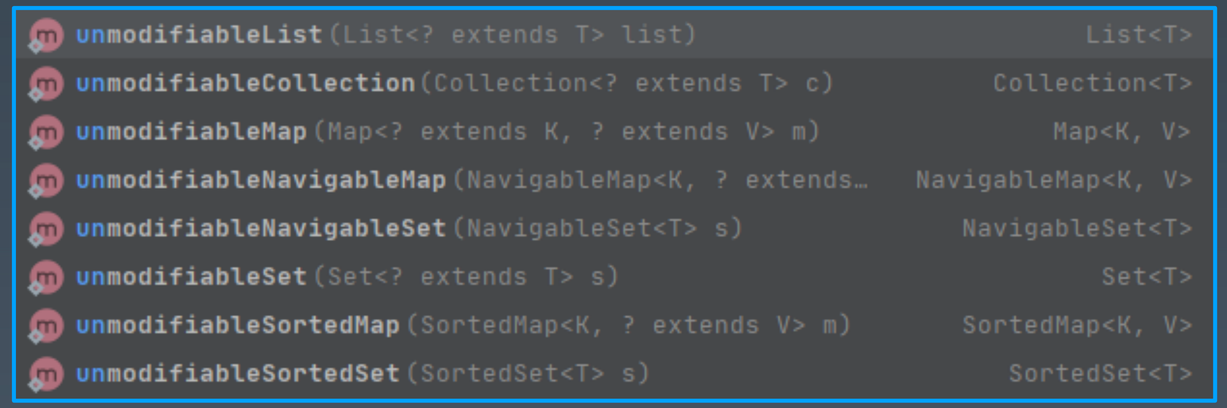
既然线程安全是写冲突和读写冲突导致的

最简单办法就是，读写都加锁。

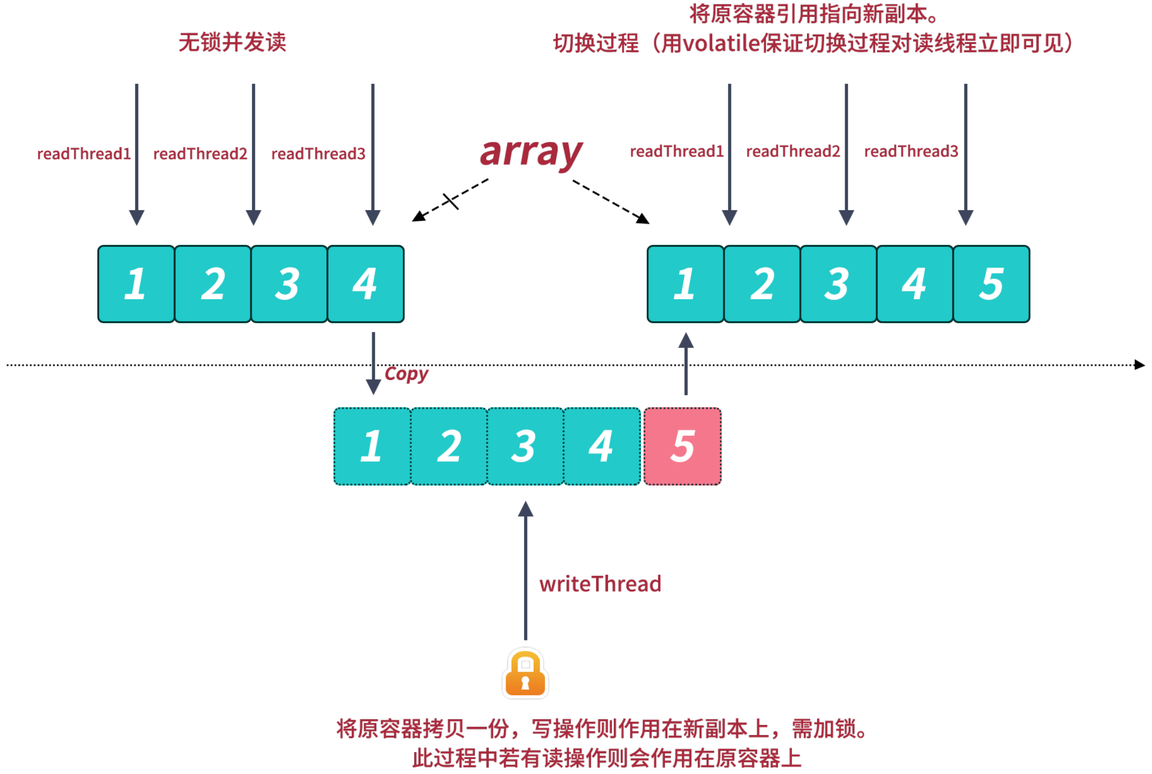
例如：

- 1.ArrayList 的方法都加上 synchronized -> Vector
- 2.Collections.synchronizedList，强制将 List 的操作加上同步
- 3.Arrays.asList，不允许添加删除，但是可以 set 替换元素
- 4.Collections.unmodifiableList，不允许修改内容，包括添加删除和 set





CopyOnWriteArrayList



核心改进原理：

1、写加锁，保证不会写混乱

2、写在一个 Copy 副本上，而不是原始数据上

（GC young 区用复制，old 区用本区内的移动）

读写分离

最终一致

CopyOnWriteArrayList

```
public      boolean
              // ReentrantLock加锁，保证线程安全
            final   ReentrantLock lock =  this  .lock;
            lock.lock();
            try   {
                    Object[] elements = getArray();
                      int   len =  elements . length;
                      // 拷贝原容器，长度为原容器长度加一
                    Object[] newElements - Arrays.copyOf(elements, len + 1 );
                      // 在新副本上执行添加操作
                    newElements[len] = e;
                      // 将原容器引用指向新副本
                    setArray(newElements);
                      return true ;
            } finally {
                      // 解锁
                    lock.unlock();
            }
```

1、插入元素时，在新副本操作，不影响旧引用，why?

CopyOnWriteArrayList

```
public   E remove （int index） {
          //加锁
        final ReentrantLock lock = this .lock;
        lock.lock();
        try {
              Object[] elements = getArray();
                int len = elements.length;
              E oldValue = get(elements, index);
                int numMoved = len - index - 1 ;
                if (numMoved == 0 )
                        //如果要删除的是列表末端数据，拷贝前len-1个数据到新副本上，再切换引用
                     setArray(Arrays.copyOf(elements, len - 1 ));
                else {
                        //否则，将除要删除元素之外的其他元素拷贝到新副本中，并切换引用
                    Object[] newElements = new Object[len - 1 ];
                     system.arraycopy(elements, 0, newElements, 0 , index);
                     System.arraycopy(elements, index + 1 , newElements, index,
                                                numMoved);
                     setArray(newElements);
            }
                return oldValue;
        } finally {
                  //解锁
                lock.unlock();
        }
    }
```

2、删除元素时

1）删除末尾元素，直接使用前N-1个元素创建一个新数组。

2）删除其他位置元素，创建新数组，将剩余元素复制到新数组。

CopyOnWriteArrayList

```
public E get ( int index) {
        return get(getArray(), index);
    }
```

直接读取即可，无需加锁

```
private E get(Object[] a, int index) {
        return (E), a[index];
    }
```

```
static final class COWIterator<E> implements ListIterator<E> {
    /** Snapshot of the array */
    private final Object[] snapshot;
    /** Index of element to be returned by subsequent call to next.  */
    private int cursor;

    private COWIterator(Object[] elements, int initialCursor) {
        cursor = initialCursor;
        snapshot = elements;
    }

    public boolean hasNext() { return cursor < snapshot.length; }

    public boolean hasPrevious() { return cursor > 0; }

    @SuppressWarnings("unchecked")
    public E next() {
        if (! hasNext())
            throw new NoSuchElementException();
        return (E) snapshot[cursor++];
    }

    @SuppressWarnings("unchecked")
    public E previous() {
        if (! hasPrevious())
            throw new NoSuchElementException();
        return (E) snapshot[--cursor];
    }

    public int nextIndex() { return cursor; }

    public int previousIndex() { return cursor-1; }
```

3、读取不需要加锁，why?

CopyOnWriteArrayList

```
static final class COWIterator<E> implements ListIterator<E> {
    /** Snapshot of the array */
    private final Object[] snapshot;
    /** Index of element to be returned by subsequent call to next.  */
    private int cursor;

    private COWIterator(Object[] elements, int initialCursor) {
        cursor = initialCursor;
        snapshot = elements;
    }

    public boolean hasNext() { return cursor < snapshot.length; }

    public boolean hasPrevious() { return cursor > 0; }

    @unchecked@
    public E next() {
        if (! hasNext())
            throw new NoSuchElementException();
        return (E) snapshot[cursor++];
    }

    @unchecked@
    public E previous() {
        if (! hasPrevious())
            throw new NoSuchElementException();
        return (E) snapshot[--cursor];
    }

    public int nextIndex() { return cursor; }

    public int previousIndex() { return cursor-1; }
```

4、使用迭代器的时候，

直接拿当前的数组对象做一个快照，此后的 List 元素变动，就跟这次迭代没关系了。

想想：淘宝商品 item 的快照。商品价格会变，每次下单都会生成一个当时商品信息的快照。

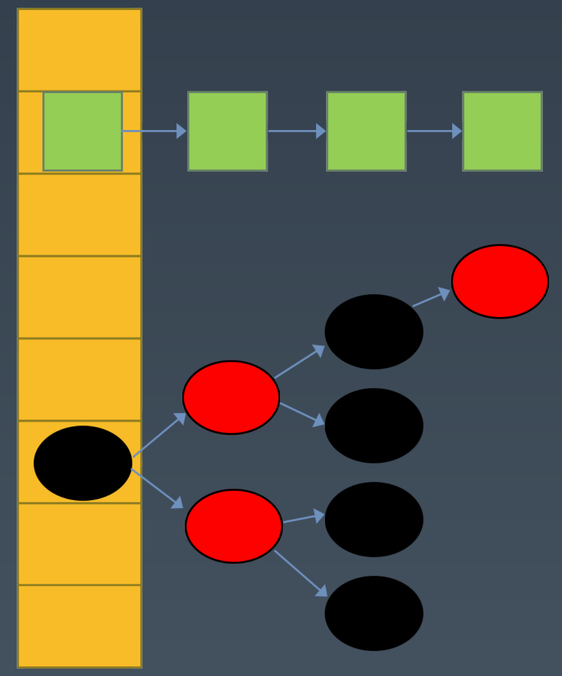
HashMap

基本特点：空间换时间，哈希冲突不大的情况下查找数据性能很高

用途：存放指定 key 的对象，缓存对象

原理：使用 hash 原理，存 k-v 数据，初始容量16，扩容x2，负载因子0.75

JDK8 以后，在链表长度到8 & 数组长度到64时，使用红黑树



安全问题：

1、写冲突

2、读写问题，可能会死循环

3、keys()无序问题

LinkedHashMap

基本特点：继承自 HashMap，对 Entry 集合添加了一个双向链表

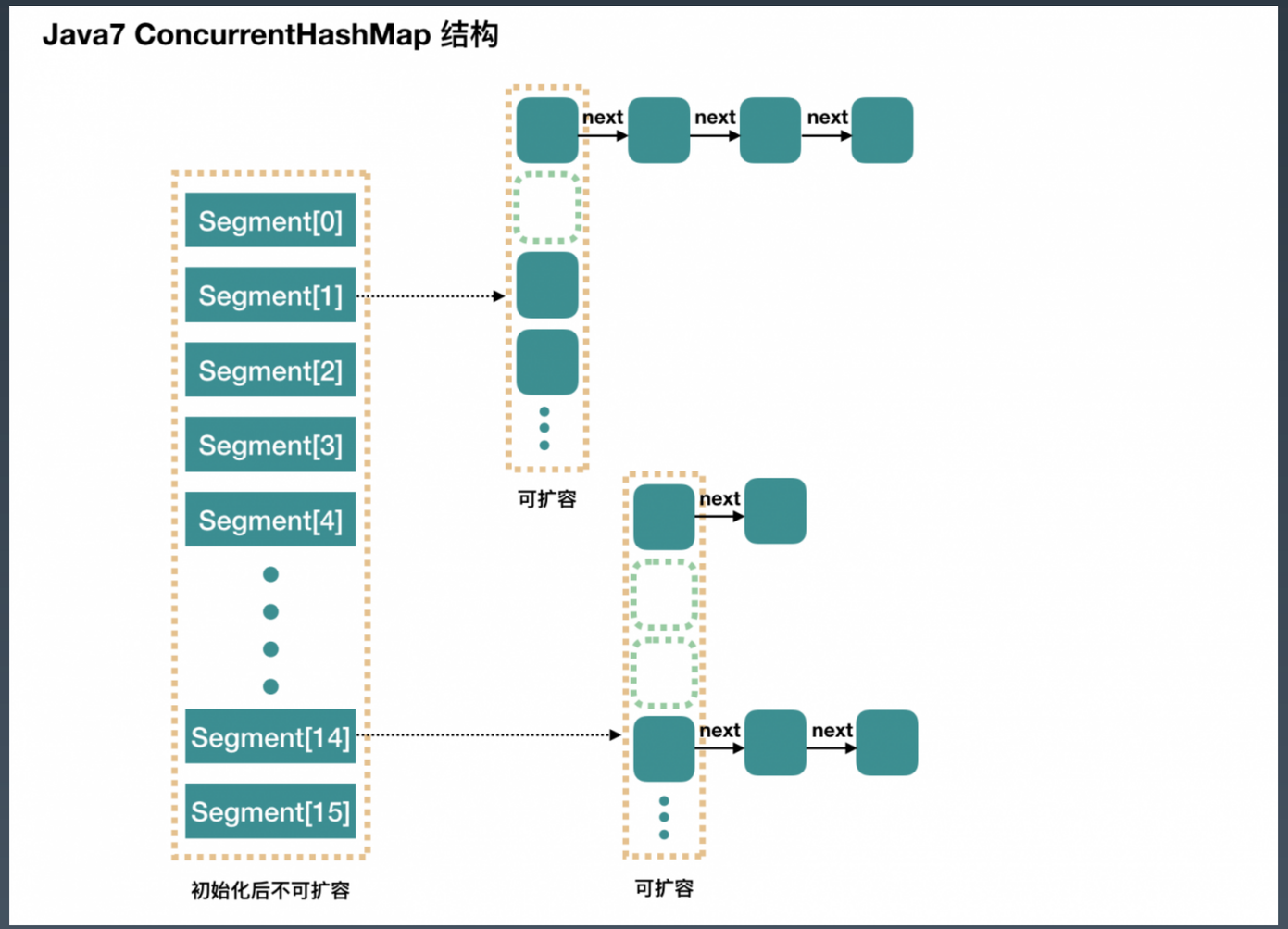
用途：保证有序，特别是 Java8 stream 操作的 toMap 时使用

原理：同 LinkedList，包括插入顺序和访问顺序

安全问题：

同 HashMap

ConcurrentHashMap-Java7 分段锁



分段锁

默认16个 Segment，降低锁粒度。

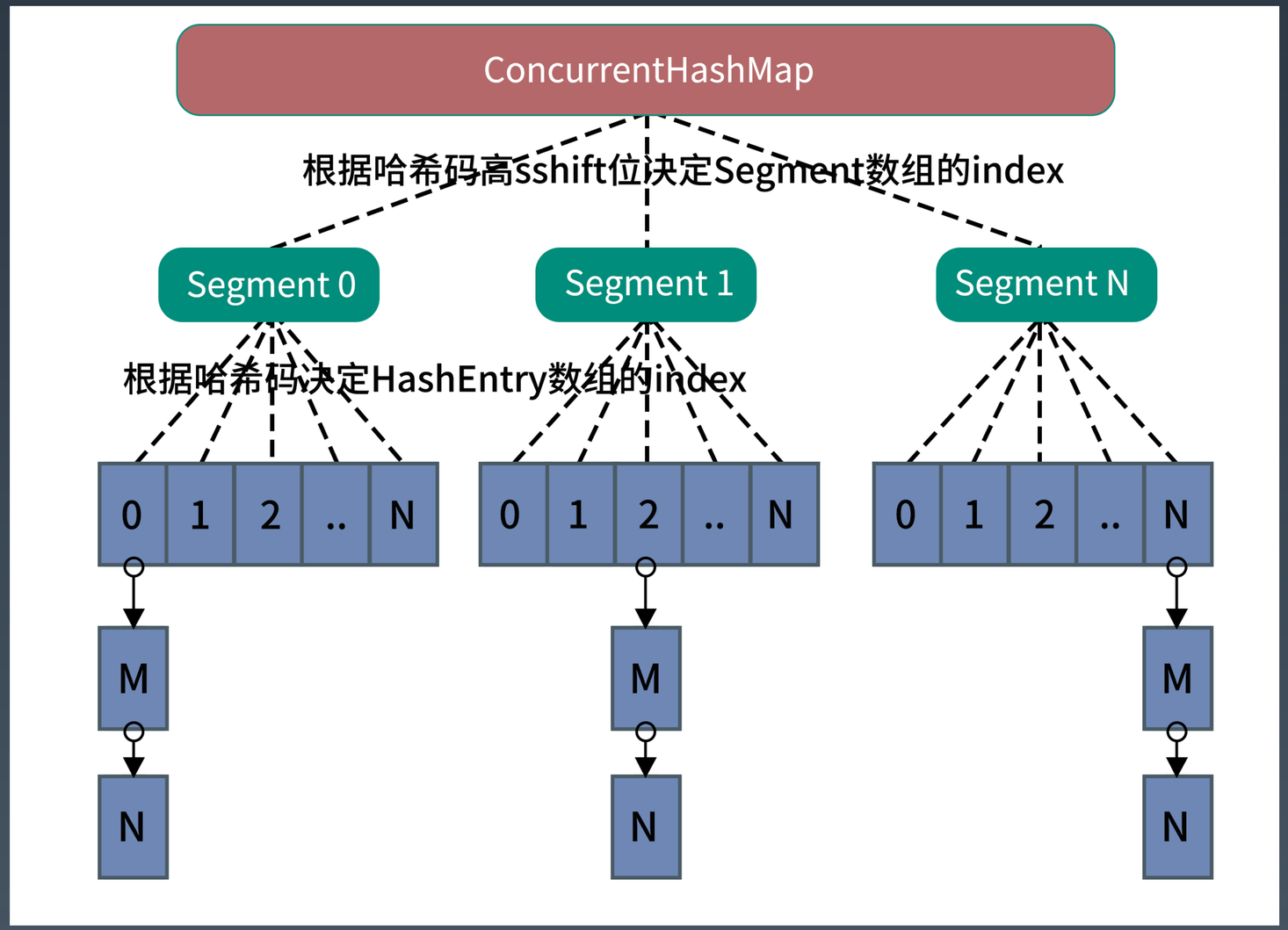
concurrentLevel = 16

想想：

Segment[] ~ 分库

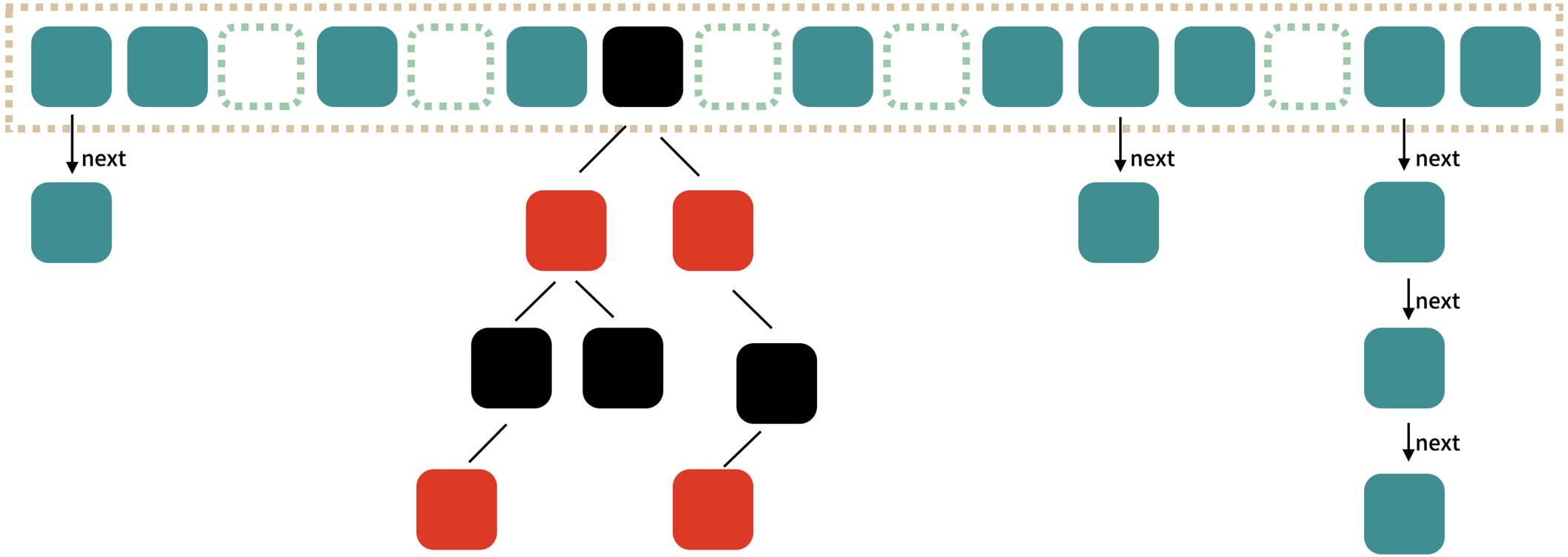
HashEntry[] ~ 分表

ConcurrentHashMap-Java7 分段锁

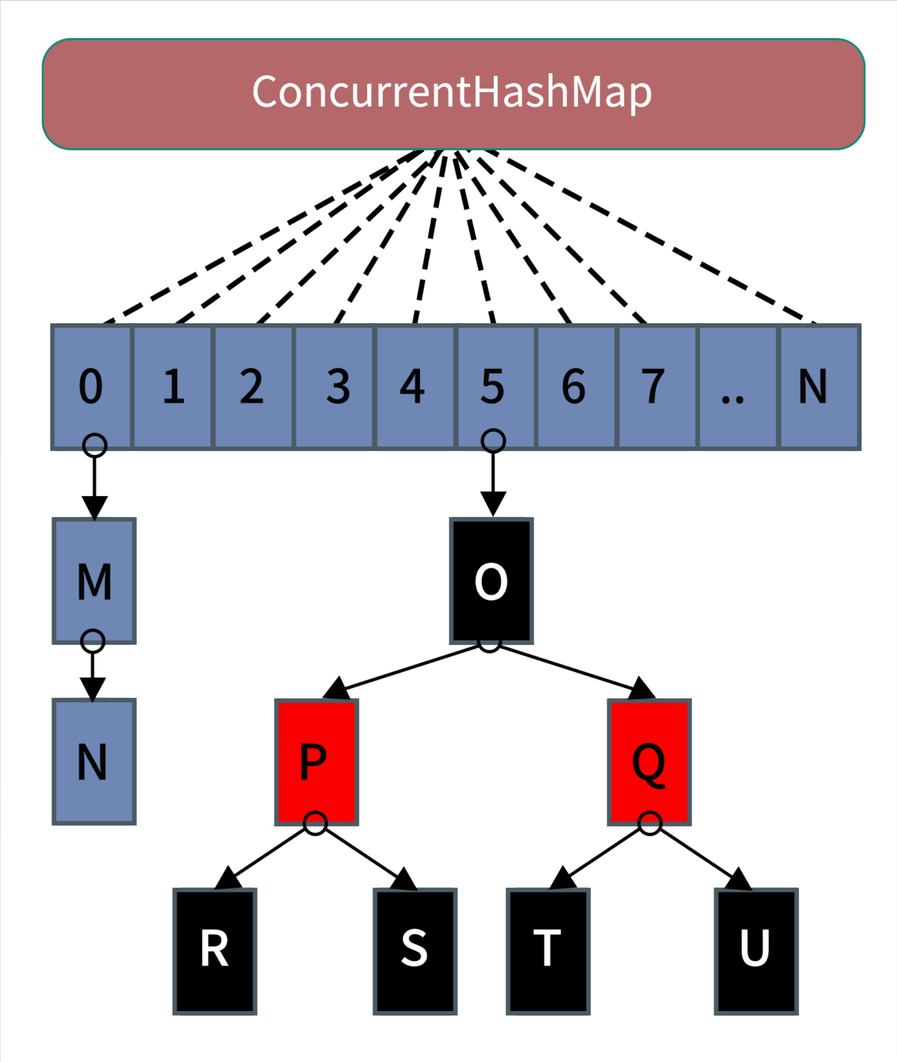


ConcurrentHashMap-Java8

Java8 ConcurrenHasMap 结构



ConcurrentHashMap-Java8



Java 7为实现并行访问，引入了 Segment 这一结构，实现了分段锁，理论上最大并发度与 Segment 个数相等。

Java 8为进一步提高并发性，摒弃了分段锁的方案，而是直接使用一个大的数组。

why?

并发集合类总结

ArrayList

并发读写不安全

LinkedList

使用副本机制改进

CopyOnWriteArrayList

HashMap

并发读写不安全

linkedHashMap

使用分段锁或 CAS

ConcurrentHashMap

2.并发编程相关内容

线程安全操作利器 - ThreadLocal

重要方法	说明
public ThreadLocal()	构造方法
protected T initialValue()	覆写-设置初始默认值
void set(T value)	设置本线程对应的值
void remove()	清理本线程对应的值
T get()	获取本线程对应的值

- 线程本地变量
- 场景: 每个线程一个副本
- 不改方法签名静默传参
- 及时进行清理

可以看做是 Context 模式，减少显式传递参数

四两拨千斤 - 并行 Stream

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    IntStream.range(1, 10000).forEach(i -> list.add(i));
    BlockingQueue<Long> blockingQueue = new LinkedBlockingQueue(10000);
    List<Long> longList = list.stream().parallel()
            .map(i -> i.longValue())
            .sorted()
            .collect(Collectors.toList());
    // 并行
    longList.stream().parallel().forEach(i -> {
        try {
            blockingQueue.put(i);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    System.out.println("blockingQueue" + blockingQueue.toString());
}
```

多线程执行，只需要加个 parallel 即可

伪并发问题

- 跟并发冲突问题类似的场景很多
- 比如浏览器端，表单的重复提交问题
-- 1、客户端控制（调用方），点击后按钮不可用，跳转到其他页
-- 2、服务器端控制（处理端），给每个表单生成一个编号，提交时判断重复

还有没有其他办法？

分布式下的锁和计数器

- 分布式环境下，多个机器的操作，超出了线程的协作机制，一定是并行的
- 例如某个任务只能由一个应用处理，部署了多个机器，怎么控制
- 例如针对用户的限流是每分钟60次计数，API 服务器有3台，用户可能随机访问到任何一台，怎么控制？（秒杀场景是不是很像？库存固定且有限。）

不要着急，分布式缓存会详细讲

3.并发编程经验总结

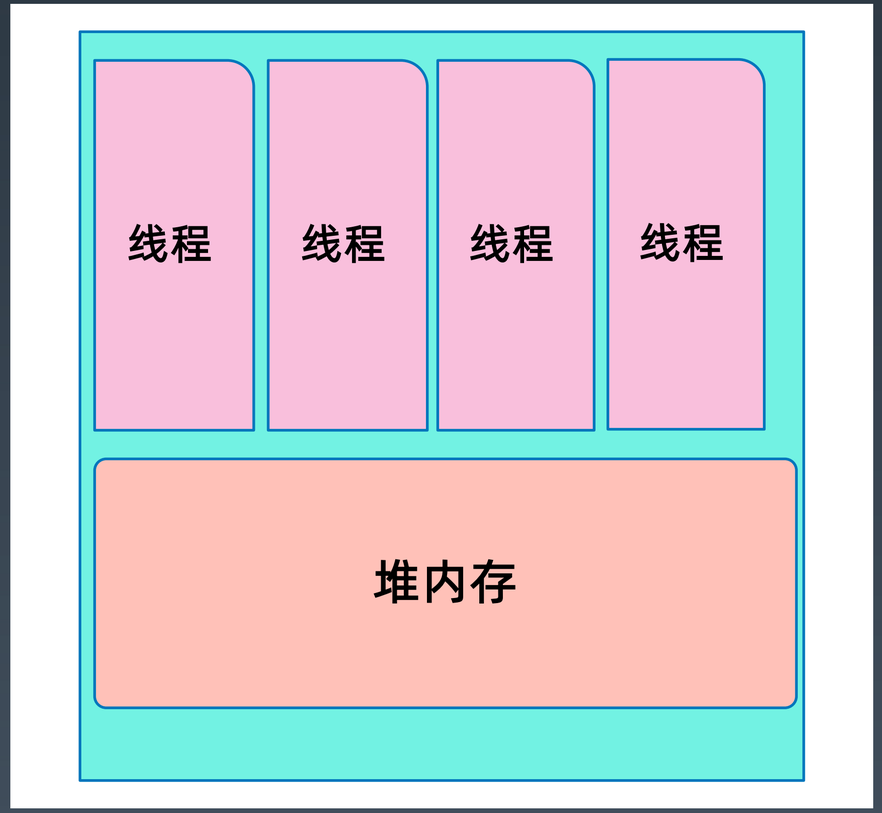
加锁需要考虑的问题

1. 粒度
2. 性能
3. 重入
4. 公平
5. 自旋锁（spinlock）
6. 场景: 脱离业务场景谈性能都是耍流氓

线程间协作与通信

1. 线程间共享：
- static/实例变量（堆内存）
- Lock
- synchronized

2. 线程间协作：
- Thread#join()
- Object#wait/notify/notifyAll
- Future/Callable
- CountdownLatch
- CyclicBarrier



可以思考：不同进程之间有哪些方式通信

4.并发编程常见面试题

第 8 节课总结回顾

常用线程安全类型

并发编程相关内容

并发编程经验总结

并发常见面试题(发给大家)

第7节课作业实践

1、（选做）列举常用的并发操作 API 和工具类，简单分析其使用场景和优缺点。

2、（选做）请思考：什么是并发？什么是高并发？实现高并发高可用系统需要考虑哪些因素，对于这些你是怎么理解的？

3、（选做）请思考：还有哪些跟并发类似/有关的场景和问题，有哪些可以借鉴的解决办法。

4、（必做）把多线程和并发相关知识梳理一遍，画一个脑图，截图上传到 GitHub 上。

可选工具：xmind，百度脑图，wps，MindManage，或其他。

THANKS!