数据结构与集合

集合框架图

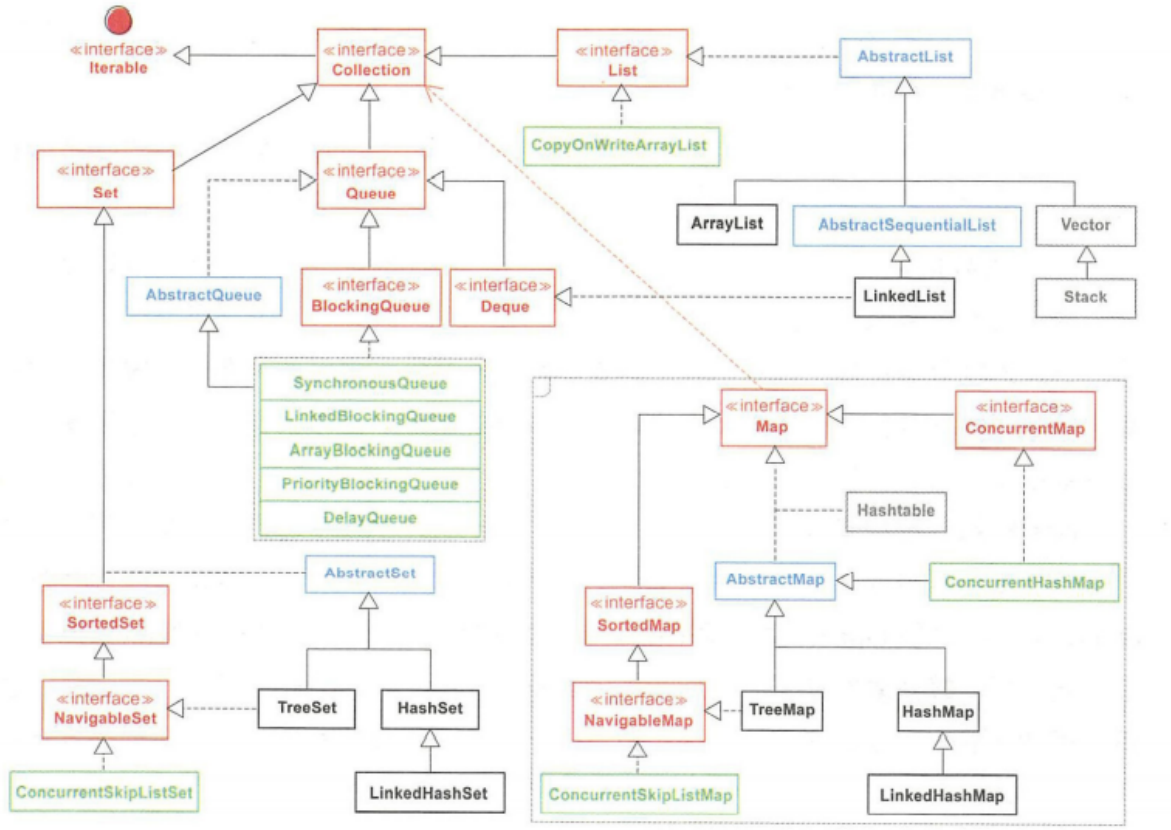


图 6–1　Java 集合框架图

框架图中主要为两类：

1. **按照单个元素存储的Collections**：Set，Queue， List 实现了Collections接口
2. **按照Key-Value 存储的Map**

- List集合：

  ArrayList是容量可以改变的非线程安全集合，集合内部使用数组进行存储，扩容时会创建更大的数组空间。支持对元素的快速随机访问，插入与删除通常很慢，因为要移动其他元素。

  LinkedList是双向链表，插入与删除速度很快，随机访问很慢。

- Queue集合：

  FIFO特性和阻塞操作

- Map集合：

  以Key-Value键值对作为存储元素实现的哈希结构。HashMap线程不安全。

- Set集合：

  不允许出现重复元素的集合类型。最常用的是HashSet、TreeSet和LinkedHashSet。HashSet使用HashMap实现，只是value固定为一个静态对象，使用Key保证集合元素的唯一性，但不保证顺序。

  TreeSet使用TreeMap实现，底层为树结构。LinkedHashSet继承自HashSet，内部使用链表维护了元素插入顺序。

集合初始化

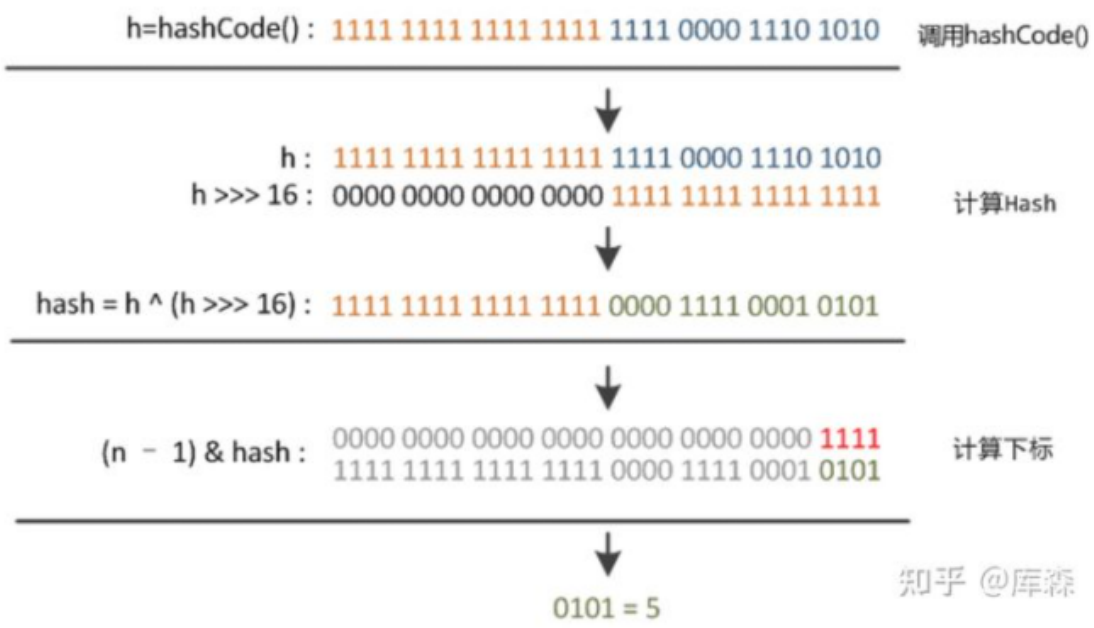
ArrayList：

- 使用无参构造时，**默认大小为10**，也就是说在第一次add的时候分配10的容量。后续的每次扩容都会创建新数组再复制。如果需要将很多元素放入ArrayList中，使用默认构造方法会造成性能损耗甚至OOM。

Hash Map：

- Capacity和Load Factor。Capacity决定存储容量大小，**默认为16**；Load Factor决定填充比例，一般使用默认0.75。基于这两个数的乘积，HashMap内部用threshold变量表示HashMap中能放入元素的个数。**HashMap的容量并不会在new的时候分配，而是在第一次put的时候完成创建的。**
- 为了提高运算速度，设定HashMap容量大小为2^n，这样的方式使计算落槽位置更快。如果初始化HashMap的时候制定了initialCapacity，则会先计算出比initialCapacity大的2的幂存入threshold，在第一次put时会按照这个2的幂初始化数组大小，此后每次扩容都是增加两倍。
- Hash算法：

  取key的 hashCode 值、根据 hashcode 计算出hash值、通过取模计算下标

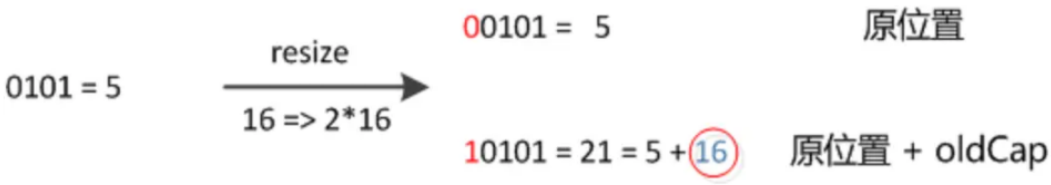


- **为什么扩容是2？（重点）**
  - HashMap源码的put方法会调用indexFor(int h, int length)方法：

    ```
    static int indexFor(int h, int length) {
        return h & (length - 1);
    }
    ```

    此方法用于确定entry在Hash表数组的位置。

    如果length不为2的幂，比如15。那么length-1的2进制就会变成1110。在h为随机数的情况下，和1110做&操作。尾数永远为0。那么0001、1001、1101等尾数为1的位置就永远不可能被entry占用。这样会造成浪费，不随机等问题。 length-1 二进制中为1的位数越多，那么分布就平均。（散列算法）
  - Resize的时候不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”



- **为什么Load Factor是0.75?**

  是使用泊松分布计算得到。负载因子是0.75的时候，空间利用率比较高，而且避免了相当多的Hash冲突，使得底层的链表或者是红黑树的高度比较低，提升了空间效率。

- **为什么链表长度到8时转化为红黑树？为什么减为6时退化回链表?**

  理想情况下随机hashCode算法下所有bin中节点的分布频率会遵循**泊松分布**，我们可以看到，一个bin中链表长度达到8个元素的概率为0.00000006，几乎是不可能事件。

  退化是6是为了避免链表和红黑树之间频繁的转换。如果阈值是7的话，删除一个元素红黑树就必须退化为链表，增加一个元素就必须树化，来回不断的转换结构无疑会降低性能，所以阈值才不设置的那么临界。

  转化为红黑树还有一个条件是table容量大于等于64，否则只会扩容。

数组与集合

数组一旦分配内存后无法扩容。

Arrays.asList()把数组转换成集合时，不能使用其修改集合相关的方法（add，remove，clear会报异常）。如果使用set修改元素值，原有数组相应位置的值也会被修改，因为后台的数据仍是原有数组，asList返回的对象是一个Arrays的内部类（和ArrayList同名），没有实现集合个数相关的修改方法。

集合与泛型

- <? extends T>可以赋值任何T及T子类的集合，上界为T，取出来的类型带有泛型限制，向上强制转型成T。null可以表示任何类型，所以除null外，任何元素都不得添加进<? extends T>集合内。
- <? super T>可以赋值任何T及T的父类集合，下界为T。投票选举类似于<? super T>的操作。选举代表时，你只能往里投选票，取数据时，根本不知道是谁投的票，相当于泛型丢失
- extends的场景是put功能受限，而super的场景是get功能受限
- <? super Fruit>的逻辑是：泛型类型可以是Object，可以是Food(假设food是fruit的父类)，但是不论是Object还是Food，显然Apple都是他们的子类，因此可以set进去。<? extends Fruit>表示泛型类型可以使Apple，Peach，Banana，所以你无法set任何东西进去，因为你根本不能确定实际类型是什么，你只能保证取出来的类型至少是Fruit类型。

元素的比较

1. Comparable和Comparator

   前者是自己和自己比，后者是第三方比较器。小于的情况返回-1，等于的情况返回0，大于的情况返回1。

2. hashCode和equals

   先比较hashCode，若相同再比较equals，若不同直接判定Objects不同，跳过equals。

   任何时候覆写equals，都必须同时覆写hashCode。

   - hashCode是根据对象的地址进行相关计算得到int类型数值的。如果不覆写hashCode，总是会得到一个与对象地址相关的唯一值，比如想实现HashSet存储不重复的元素就需要覆写。

fail-fast机制

对集合遍历操作时的错误检测机制，在遍历中途出现意料之外的修改时，通过unchecked异常暴力地反馈出来。这种机制经常出现在多线程环境下，当前线程会维护一个计数比较器，即expectedModCount，记录已经修改的次数。进入遍历前会把实时修改次数modCount赋值给expectedModCount，如果两个数据不相等就抛出异常。

Map类集合

KV是否能设置为null：

Map 集合类	Key	Value	Super	JDK	说　明
Hashtable	不允许为 null	不允许为 null	Dictionary	1.0	线程安全（过时）
ConcurrentHashMap	不允许为 null	不允许为 null	AbstractMap	1.5	锁分段技术或 CAS（JDK8 及以上）
TreeMap	不允许为 null	允许为 null	AbstractMap	1.2	线程不安全（有序）
HashMap	允许为 null	允许为 null	AbstractMap	1.2	线程不安全（resize 死链问题）

红黑树

与AVL树类似，都是在进行插入和删除元素时，通过特定的旋转来保持自身平衡。与AVL树相比，红黑树并不追求所有递归子树的高度差不超过1，而是保证从根节点到叶子节点的最长路径不超过最短路径的2倍，所以他的最坏运行时间也是O(logn)

五个约束条件：

1. 节点只能是红色或黑色
2. 根节点必须是黑色
3. 所有NIL节点都是黑色。NIL，即叶节点下挂的两个虚节点。
4. 一条路径上不能出现相邻的两个红色节点。
5. 在任何递归子树内，根节点到叶子节点的所有路径上包含相同数目的黑色节点。

总结： 有红必有黑，红红不相连。上述约束条件保证了红黑树新增、删除、查找的最坏时间复杂度均为O(logn)。如果一个树的左节点或右节点不存在，则均认定为黑色。红黑树的任何旋转在**3次**之内均可完成。

插入节点前，需要明确三个条件：

1. 需要调整的新节点总是红色的
2. 如果插入新节点的父节点是黑色的，无需调整。因为依然能符合红黑树的5个约束条件。
3. 如果插入新节点的父节点是红色的，因为红色不能相邻，所以进入循环判断，或重新着色，或左右旋转，最终达到五个约束条件。

HashMap

表 6-2　哈希类集合的三个基本存储概念

名　称	说　明
table	存储所有节点数据的数组
slot	哈希槽。即 table[i] 这个位置
bucket	哈希桶。table[i] 上所有元素形成的表或数的集合

- **数据丢失**

  HashMap resize过程中调用transfer()方法，遍历table并将元素复制到新的table中。当前线程迁移过程中，其他线程新增的元素有可能落在了已经遍历过的哈希槽上；遍历完成后，table数组引用指向newTable，这时新增元素就会丢失，被垃圾回收。

  - 并发赋值时被覆盖
  - 已遍历区间新增元素会丢失
  - “新表”被覆盖（多个线程同时resize）
  - 迁移丢失。在迁移过程中，有并发时，next被提前置成null
- **死链**

  旧数组元素迁移到新数组时，依旧采用头插入法，这样将会导致新链表元素的逆序排序。多线程情况下，如果线程1的entry和next指向了链表中节点然后被挂起，线程2此时完成了resize，线程1的entry和next此时顺序是反过来的。线程1的resize继续跑下去会形成环形链表。**产生问题的根源是Entry的next被并发修改**

ConcurrentHashMap

JDK7版本：

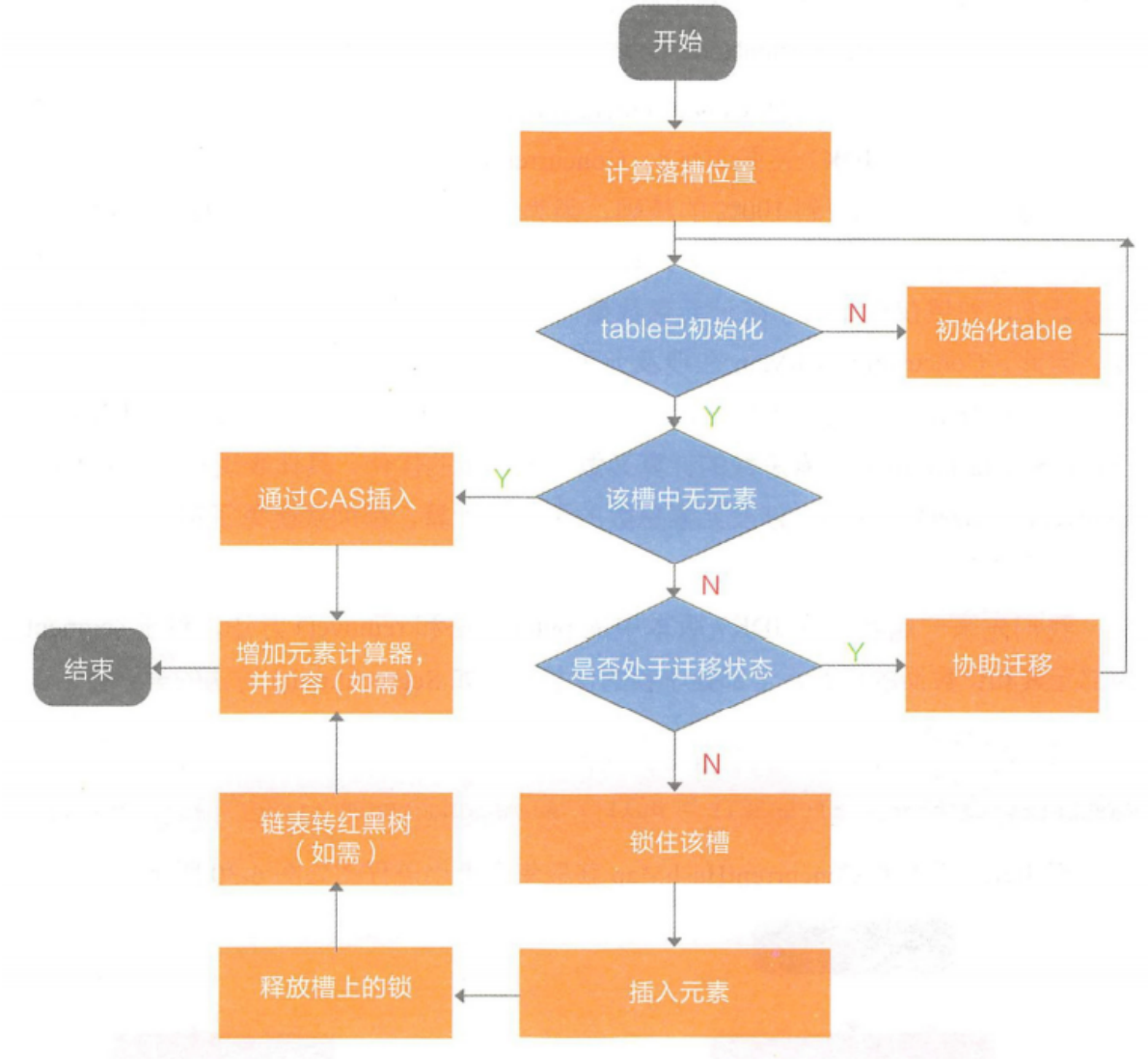
分段锁，由内部类Segment实现，用来管理辖区内各个HashEntry。ConcurrentMap被Segment分成了很多小区，小区通过加锁的方式，保证每个Segment内都不发生冲突。

JDK8及之后：

1. 取消分段锁机制，进一步降低冲突概率
2. 引入红黑树结构。同一个哈希槽上的元素个数超过一定阈值后，单向链表改为红黑树结构。
3. 使用了更加优化的方式统计集合内的元素数量。使用了CAS和多种优化以提高并发能力。

链表转化为红黑树：使用同步块锁住当前槽的首元素，防止其他进程对当前槽进行增删改操作，转化完成后利用CAS替换原有链表。红黑树转链表也是同理。

Concurrent HashMap元素插入流程：



Node的两个子类：

1. ForwardingNode

   在table扩容时使用，内部记录了扩容后的table（nextTable）。当table需要扩容时，遍历当前table中的每一个槽，如果不为null，则需要把其中所有元素根据hash值放入扩容后的nextTable中，而原table的槽内会放置一个ForwardingNode节点。此节点会把find()请求转发到扩容后的nextTable上。而执行put()方法的线程如果碰到此节点，也会协助进行迁移。

2. ReservationNode

   在computeIfAbsent()及相关方法中作为一个预留节点使用。computeIfAbsent()方法会先判断相应的Key值是否存在，如果不存在，则调用由用户实现的自定义方法来生成value值，组成KV键值对，随后插入此哈希集合中。在并发场景下，在从得知Key不存在到插入哈希集合的时间间隔内，为了防止哈希槽被其他线程抢占，当前线程会使用一个ReservationNode节点放到槽中并加锁，保证线程的安全性。

ConcurrentHashMap1.7和1.8的区别：

在jdk1.7是采用Segment+HashEntry的方式实现的，lock加在segment上， 1.7的size计算是先采用不加锁的方式，连续计算元素的个数，最多计算3次：

（1）如果前后两次计算结果相同，则说明计算出来的元素个数是准确的。

（2）如果前后两次计算结果不同，则给每个Segment进行加锁，再计算一次元素的个数。

1.8中放弃了Segment臃肿的设计，取而代之的是采用Node+CAS+synchronized来保证并发安全进行实现，1.8中使用一个volatile类型的变量baseCount记录元素的个数，当插入或删除数据时，会通过addCount()方法更新baseCount，通过累加baseCount和CounterCell数组中的数量，即可得到元素的总个数。

- 数据结构：取消了 Segment 分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。
- 保证线程安全机制：JDK1.7 采用 Segment 的分段锁机制实现线程安全，其中 Segment 继承自 ReentrantLock 。JDK1.8 采用`CAS+synchronized`保证线程安全。
- 锁的粒度：JDK1.7 是对需要进行数据操作的 Segment 加锁，JDK1.8 调整为对每个数组元素加锁（Node）。
- 链表转化为红黑树：定位节点的 hash 算法简化会带来弊端，hash 冲突加剧，因此在链表节点数量大于 8（且数据总量大于等于 64）时，会将链表转化为红黑树进行存储。
- 查询时间复杂度：从 JDK1.7的遍历链表O(n)， JDK1.8 变成遍历红黑树O(logN)。