Map

ThreadLocal

1、一个数组，2x 容量；前后 分别存 key，value

2、标记 删除

3、依旧 有 扩容的概念；

为什么 threadLocal 不使用 自身的实例 做 key，而是 用自己的 弱引用对象 做key 呢？

--》为了 key-value 能够 自动 回收！

（1）如果，values 集合中的 key 是 自身，强引用；
那么，它是 没办法 检测到 是否有 其他 强引用 还在使用 这个对象的，
换句话说，其他的强引用 都没了，这个 对象 该被回收了，但是 values 这边 没办法 检查到，自行回收！

（2）而 弱引用 可以 --》
假设，外面 将 用到 该对象的 强引用 都置为 null了，
因为 这里只是 弱引用，所以 那个 threadLocal 实例对象 会被回收；
而我们 这里 在后面 各种 put 等 修改 values 的时候，
会自行 检测 弱引用 是否入队，
--》如果 发现 它 入队了，说明 threadLocal 实例 被回收了，
那么 这里 负责 将 对应的 value 对象 一并清空！

SparseArray：

唯一 没有 hash 冲突 的 类hash 数据结构

1、由于 key 都是 int，所以 可以快速的二分定位，找到 目标key的 合适 index，进行 数组的copy移动；

2、俩数组，一个key的，一个value的

3、标记 删除
会在gc() 时，整理数组，将 有效数据 (value != DELETED的 那些) 前移

4、因为 没有 拿 key 来 求hash 的过程，就没有 冲突 的概念，直接 使用了 key值 来快速的 二分定位，

新加一个key的时候，如果 有空闲的位置（比如 容量够，比如 之前 有 标记删除的）就 直接使用；

否则 就 扩容：
当前 存的有效数据 不到 4 的话，新容量 就是8；否则 就是 2x 扩容！

5、当然了，插入 合适位置的时候 key、value 俩数组，都要 进行 数组 copy。

6、这里面 扩容，申请数组时，用到了 反射 创建 指定类型 数组的方法：

T[ ] newArray = (T[ ]) Array.newInstance(array.getClass().getComponentType() , growSize(currentSize));

SparseXXXArray：

key 固定为 int 类型，是 不让 指定的！

主要是 指定了 value 的类型

ArrayMap

1、cache 复用机制 没太 搞清楚

俩数组：

2、一个 mHashes 数组，
存放 所有的key的 hash（有 hash的过程，就会有冲突），是 有序的，可以 二分查找 快速定位

3、一个 mArray 数组，2x 容量，存放 对应位置的 真实 key 和 value；

4、key 为 null的，hash值 就当做0 来找位置

5、put 操作时，可能 会进行扩容，
扩容的目标大小 分区间 是：4，8，原容量的 1.5倍

6、这里面，既然有 求 hash 操作，就意味着 有冲突，
--》寻找 key-value 在 mArray 数组中的 合适的 index时，

先用 key的 hash值 在 mHashes 数组 二分定位 到 合适的 起始位置，
若有这个 hash，index >=0 ，就在 mArray 查找 之前 是否 存了这个key，
从 hash 相同的那个位置 开始；

（1）若当前位置 就是，即 找 到了，直接 返回；
（2）否则，先 当前位置 往后找，从这个位置 开始，每次 +2，看 key 是否相同，直到 最后
（3）否则，再 当前位置 往前找，每次 -2；

TreeMap

1、用法 api

2、特性

1、map中 的 映射 entry
会按照 key 指定的 比较器 排序

2、没有指定 比较器 的 时候，就是 自然排序：
数字：从小到大
字母：按照 字母表 从前往后

遍历方式：出来的就是 有序的

1、
map.values(),
map.keySet(),
map.entrySet(),
map.forEach()

2、for 循环，或者 迭代器 都行

3、因为 要维持 key 的顺序，所以 性能差

依旧是 线程 不安全 的 数据结构！

没细看。。

实现原理：

--》 底层 是 红黑树

HashMap

1.7 版本

1、结构：

数组 + 链表（每个桶上 都是一个链表）

2、存在的问题

1、哈希冲突 较多的话，链表很长， 查询 速度慢

速度 由 O(1) 退化到 O(n)

2、并发 安全问题，不加锁，可能导致 死循环！

（1）这个版本，put 操作，新 Node 总是 插入 链表的头结点，
如果 某个桶的链表 确实在一开始 是空的话，当 多线程 并发 put 时，可能会 造成 循环链表，
线程 A 以为 没有后续结点，直接拼上 那个后续位置；
线程 B 也是如此；
因为 并发的 非原子性 操作，会导致 A next 指向 B，B next 指向 A；

（2）后面的 get 操作，一旦 读取到 这个 桶，next遍历 就 死循环 出不来了

3、并发性能 差

要想 安全，只能 对整个hash表 加锁

1.8 版本

1、结构：
数组 + 链表 / 红黑树

当桶上的元素 >= 8 时，树化

2、优化 的问题

1、链表 查询慢的问题 O(n)

树化后，提升到 O(logn)

2、并发 安全问题：

之前 链表的 头插法 导致 死循环；
现在 改为 尾插法，新结点 都插入 链表 的 尾部！

新结点的 next = null了，而不是 像之前一样 指向某个结点了！

3、依然 存在的问题

并发 性能问题，依旧 需要 整个hash表 加锁

HashTable：加锁的 HashMap

1、它 提供给 外界 使用的 public 方法，全都是 synchronized 修饰的，
并发 性能 低下，可以 粗略的 理解为，通过 synchronized 修饰的 HashMap

2、和 HashMap 不同的是，这里 不允许 null 的 key 和 value

ConcurrentHashMap

1.7 版本

1、结构
将 整个hash表 分成 多个 segment 片段（数组）；
其中 每个 segment 都是 一个 小型的 hash表
--》数据结构 组成 和 原 HashMap 一致

2、且 segment 继承自 ReentrantLock，自带 加锁 功能。

实现了 锁 细化，将 之前 对 整个表 加锁，细化为 对某个 segment 加锁，提高 并发性能；

而且 里面 只有 put 操作 的时候，才 用到了 锁，且支持 自旋；
get 则 不需要！

3、具体操作是：
先 hash 一遍 找到 segment，
再 hash 一遍 找到 segment 内部 的 桶；

1.8 版本

1、和 HashMap 一样 引入了 红黑树，提升 哈希冲突后的 查询性能

2、锁的粒度 进一步细化，去除了 segment 概念；

用 cas + synchronized 关键字，
--》锁的是 单个桶了（即 单个 链表 / 红黑树）

总体来说：

提升 并发性能 、优化的手段：就是 将 加锁的粒度 尽量细化！

由 整个 hash表 --》单个 segment 表（小型 hash表） --》单个桶（链表/红黑树）

待看 文章：

LinkedHashMap

1、继承自 hashMap，可以 通过 key 快速定位 目标节点 Node

2、又扩展了 Node，为 双向链表，维护了 访问顺序：

这样的话，当 访问到 旧的元素，就可以把 该元素 从当前位置 高效的 删除，提到 最前面

实现 LruCache：

非常简单，就是 基于 LinkeHashMap 实现的；

还有 DiskLruCache 都是

1、上面 已经 实现了 最后 被访问的节点，总在 最前面

2、那 我们 只要在 上面的基础上，
自己维护 一个 size++ 或者 size-- 的操作 就好了，

当++到 MaxSize的时候，就 依次 remove 链表 最后的元素，直到 符合 size 要求 即可。。