# Java容器

## 概览

容器主要包含两种：Collection和Map。Collection存储对象的集合，Map存储键值对的映射。

### Collection



#### Set

* TreeSet：基于红黑树实现，支持有序性操作，例如根据一个范围查找元素的操作。查找效率不如HashSet，HashSet查找时间复杂度O(1)，TreeSet查找时间复杂度O(logN)。
* HashSet：基于哈希表实现，支持快速查找，但是不支持有序性操作。并且失去了元素的插入顺序信息，也就是说Iterator遍历HashSet得到的结果是不确定的。
* LinkedHashSet：具有HashSet的查找效率，并且内部使用双向链表维护插入的顺序。

#### List

* ArrayList：基于动态数组实现，支持随机访问。
* Vector：和ArrayList类似，但是是线程安全的。
* LinkedList：基于双向链表实现，只能顺序访问，但是可以快速地在链表中插入和删除元素。不仅如此，LinkedList还可以作栈、队列和双向队列。

#### Queue

* LinkedList：可以用来实现双向队列；
* PriorityQueue：基于堆结构实现，可以用来实现优先队列。

### Map



* TreeMap：基于红黑树实现。
* HashMap：基于哈希表实现。
* HashTable：与HashMap类似，但是是线程安全的。这意味着同一时刻多个线程可以同时写入HashTable并且不会导致数据不一致，它是遗留类，不应该去使用它。现在可以使用ConcurrentHashMap来支持线程安全，ConcurrentHashMap效率更高，因为引入了分段锁。
* LinkedHashMap：使用双向链表维护元素的顺序，顺序为插入顺序或者最近最少使用顺序（LRU）。

## 容器中的设计模式

### 迭代器模式



Collection继承了Iterable接口，通过其中的Iterator()方法产生一个Iterator对象，通过对象迭代遍历Collection中的元素。

Java1.5之后可以使用foreach方法来遍历实现了Iterator接口的对象。

|  |
| --- |
| List<String> list = new ArrayList<>();  list.add("a");  list.add("b");  for (String item : list) {  System.out.println(item);  } |

### 适配器模式

Java.util.Arrays#asList()可以把数组类型转化为List类型。

|  |
| --- |
| @SafeVarargs  public static <T> List<T> asList(T... a) |

应该注意的是 asList() 的参数为泛型的变长参数，不能使用基本类型数组作为参数，只能使用相应的包装类型数组。例如：

|  |
| --- |
| Integer[] arr = {1, 2, 3};  List list = Arrays.asList(arr); |

也可以使用以下方式调用 asList()：

|  |
| --- |
| List list = Arrays.asList(1, 2, 3); |

## 源码分析

如果没有特殊说明，以下源码分析基于Java1.8。

### ArrayList

#### 概览

因为 ArrayList 是基于数组实现的，所以支持快速随机访问。RandomAccess接口标识着该类支持快速随机访问。

|  |
| --- |
| public class ArrayList<E> extends AbstractList<E>  implements List<E>, RandomAccess, Cloneable, java.io.Serializable |

数组的默认大小为 10。

|  |
| --- |
| private static final int DEFAULT\_CAPACITY = 10; |

#### 扩容

添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，如果不够时，需要使用 grow() 方法进行扩容，新容量的大小为 oldCapacity + (oldCapacity >> 1)，也就是旧容量的 1.5 倍。

扩容操作需要调用 Arrays.copyOf() 把原数组整个复制到新数组中，这个操作代价很高，因此最好在创建 ArrayList 对象时就指定大概的容量大小，减少扩容操作的次数。

|  |
| --- |
| public boolean add(E e) {  ensureCapacityInternal(size + 1); // Increments modCount!!  elementData[size++] = e;  return true;  }  private void ensureCapacityInternal(int minCapacity) {  if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {  minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);  }  ensureExplicitCapacity(minCapacity);  }  private void ensureExplicitCapacity(int minCapacity) {  modCount++;  // overflow-conscious code  if (minCapacity - elementData.length > 0)  grow(minCapacity);  }  private void grow(int minCapacity) {  // overflow-conscious code  int oldCapacity = elementData.length;  int newCapacity = oldCapacity + (oldCapacity >> 1);  if (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  if (newCapacity - MAX\_ARRAY\_SIZE > 0)  newCapacity = hugeCapacity(minCapacity);  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.copyOf(elementData, newCapacity);  } |

#### 删除元素

需要调用 System.arraycopy() 将 index+1 后面的元素都复制到 index 位置上，该操作的时间复杂度为 O(N)，可以看出 ArrayList 删除元素的代价是非常高的。

|  |
| --- |
| public E remove(int index) {  rangeCheck(index);  modCount++;  E oldValue = elementData(index);  int numMoved = size - index - 1;  if (numMoved > 0)  System.arraycopy(elementData, index+1, elementData, index, numMoved);  elementData[--size] = null; // clear to let GC do its work  return oldValue;  } |

#### Fail-Fast

modCount 用来记录 ArrayList 结构发生变化的次数。结构发生变化是指添加或者删除至少一个元素的所有操作，或者是调整内部数组的大小，仅仅只是设置元素的值不算结构发生变化。

在进行序列化或者迭代等操作时，需要比较操作前后 modCount 是否改变，如果改变了需要抛出 ConcurrentModificationException。

|  |
| --- |
| private void writeObject(java.io.ObjectOutputStream s)  throws java.io.IOException{  // Write out element count, and any hidden stuff  int expectedModCount = modCount;  s.defaultWriteObject();  // Write out size as capacity for behavioural compatibility with clone()  s.writeInt(size);  // Write out all elements in the proper order.  for (int i=0; i<size; i++) {  s.writeObject(elementData[i]);  }  if (modCount != expectedModCount) {  throw new ConcurrentModificationException();  }  } |

#### 序列化

ArrayList 基于数组实现，并且具有动态扩容特性，因此保存元素的数组不一定都会被使用，那么就没必要全部进行序列化。

保存元素的数组 elementData 使用 transient 修饰，该关键字声明数组默认不会被序列化。

|  |
| --- |
| transient Object[] elementData; // non-private to simplify nested class access |

ArrayList 实现了 writeObject() 和 readObject() 来控制只序列化数组中有元素填充那部分内容。

|  |
| --- |
| private void readObject(java.io.ObjectInputStream s)  throws java.io.IOException, ClassNotFoundException {  elementData = EMPTY\_ELEMENTDATA;  // Read in size, and any hidden stuff  s.defaultReadObject();  // Read in capacity  s.readInt(); // ignored  if (size > 0) {  // be like clone(), allocate array based upon size not capacity  ensureCapacityInternal(size);  Object[] a = elementData;  // Read in all elements in the proper order.  for (int i=0; i<size; i++) {  a[i] = s.readObject();  }  }  } |

|  |
| --- |
| private void writeObject(java.io.ObjectOutputStream s)  throws java.io.IOException{  // Write out element count, and any hidden stuff  int expectedModCount = modCount;  s.defaultWriteObject();  // Write out size as capacity for behavioural compatibility with clone()  s.writeInt(size);  // Write out all elements in the proper order.  for (int i=0; i<size; i++) {  s.writeObject(elementData[i]);  }  if (modCount != expectedModCount) {  throw new ConcurrentModificationException();  }  } |

序列化时需要使用 ObjectOutputStream 的 writeObject() 将对象转换为字节流并输出。而 writeObject() 方法在传入的对象存在 writeObject() 的时候会去反射调用该对象的 writeObject() 来实现序列化。反序列化使用的是 ObjectInputStream 的 readObject() 方法，原理类似。

### Vector

#### 同步

它的实现与 ArrayList 类似，但是使用了 synchronized 进行同步。

|  |
| --- |
| public synchronized boolean add(E e) {  modCount++;  ensureCapacityHelper(elementCount + 1);  elementData[elementCount++] = e;  return true;  }  public synchronized E get(int index) {  if (index >= elementCount)  throw new ArrayIndexOutOfBoundsException(index);  return elementData(index);  } |

##### 与ArrayList比较

* Vector是同步的，因此开销就会比ArrayList大，访问速度更慢，最好使用ArrayList而不是Vector，因为同步操作完全可以由程序员自己控制。
* Vector每次扩容请求其大小2倍的空间，而ArrayList为1.5倍。

##### 替代方案

可以使用 Collections.synchronizedList(); 得到一个线程安全的 ArrayList。

|  |
| --- |
| List<String> list = new ArrayList<>();  List<String> synList = Collections.synchronizedList(list); |

也可以使用 concurrent 并发包下的 CopyOnWriteArrayList 类。

|  |
| --- |
| List<String> list = new CopyOnWriteArrayList<>(); |

### CopyOnWriteArrayList

#### 读写分离

写操作在一个复制的数组上进行，读操作还是在原始数组中进行，读写分离，互不影响。

写操作需要加锁，防止并发写入时导致写入数据丢失。

写操作结束之后需要把原始数组指向新的复制数组。

|  |
| --- |
| public boolean add(E e) {  final ReentrantLock lock = this.lock;  lock.lock();  try {  Object[] elements = getArray();  int len = elements.length;  Object[] newElements = Arrays.copyOf(elements, len + 1);  newElements[len] = e;  setArray(newElements);  return true;  } finally {  lock.unlock();  }  }  final void setArray(Object[] a) {  array = a;  } |

|  |
| --- |
| @SuppressWarnings("unchecked")  private E get(Object[] a, int index) {  return (E) a[index];  } |

#### 适用场景

CopyOnWriteArrayList 在写操作的同时允许读操作，大大提高了读操作的性能，因此很适合读多写少的应用场景。

#### 缺陷

* 内存占用：在写操作时需要复制一个新的数组，使得内存占用为原来的两倍左右；
* 数据不一致：读操作不能读取实时性的数据，因为部分写操作的数据还未同步到读数组中。

CopyOnWriteArrayList 不适合内存敏感以及对实时性要求很高的场景。

### LinkedList

#### 概览

基于双向链表实现，使用Node存储节点信息。

|  |
| --- |
| private static class Node<E> {  E item;  Node<E> next;  Node<E> prev;  } |

每个链表存储了first和last指针：

|  |
| --- |
| transient Node<E> first;  transient Node<E> last; |



#### 与ArrayList的比较

* ArrayList基于动态数组实现，LinkedList基于双向链表实现；
* ArrayList支持随机访问，LinkedList不支持；
* LinkedList在任意位置添加或者删除元素更快。

### HashMap

为了便于理解，HashMap源码基于JDK1.7分析。

#### 存储结构

内部包含了一个Entry类型的数组table。

|  |
| --- |
| transient Entry[] table; |

Entry 存储着键值对。它包含了四个字段，从 next 字段我们可以看出 Entry 是一个链表。即数组中的每个位置被当成一个桶，一个桶存放一个链表。HashMap 使用拉链法来解决冲突，同一个链表中存放哈希值相同的 Entry。



|  |
| --- |
| static class Entry<K,V> implements Map.Entry<K,V> {  final K key;  V value;  Entry<K,V> next;  int hash;  Entry(int h, K k, V v, Entry<K,V> n) {  value = v;  next = n;  key = k;  hash = h;  }  public final K getKey() {  return key;  }  public final V getValue() {  return value;  }  public final V setValue(V newValue) {  V oldValue = value;  value = newValue;  return oldValue;  }  public final boolean equals(Object o) {  if (!(o instanceof Map.Entry))  return false;  Map.Entry e = (Map.Entry)o;  Object k1 = getKey();  Object k2 = e.getKey();  if (k1 == k2 || (k1 != null && k1.equals(k2))) {  Object v1 = getValue();  Object v2 = e.getValue();  if (v1 == v2 || (v1 != null && v1.equals(v2)))  return true;  }  return false;  }  public final int hashCode() {  return Objects.hashCode(getKey()) ^ Objects.hashCode(getValue());  }  public final String toString() {  return getKey() + "=" + getValue();  }  } |

#### 拉链法工作原理

|  |
| --- |
| HashMap<String, String> map = new HashMap<>();  map.put("K1", "V1");  map.put("K2", "V2");  map.put("K3", "V3"); |

* 新建一个HashMap，默认大小为16；
* 插入<K1,V1>键值对，先计算K1的hashCode为115，使用除留余数法得到所在桶的下标115%16=3；
* 插入 <K2,V2> 键值对，先计算 K2 的 hashCode 为 118，使用除留余数法得到所在的桶下标 118%16=6。
* 插入 <K3,V3> 键值对，先计算 K3 的 hashCode 为 118，使用除留余数法得到所在的桶下标 118%16=6，插在 <K2,V2> 前面。

应该注意到链表的插入是以头插法方式进行的，例如上面的 <K3,V3> 不是插在 <K2,V2> 后面，而是插入在链表头部。

查找需要分成两步进行：

* 计算键值所在的桶；
* 在链表上顺序查找，时间复杂度与链表长度成正比。



#### put操作

|  |
| --- |
| public V put(K key, V value) {  if (table == EMPTY\_TABLE) {  inflateTable(threshold);  }  // 键为 null 单独处理  if (key == null)  return putForNullKey(value);  int hash = hash(key);  // 确定桶下标  int i = indexFor(hash, table.length);  // 先找出是否已经存在键为 key 的键值对，如果存在的话就更新这个键值对的值为 value  for (Entry<K,V> e = table[i]; e != null; e = e.next) {  Object k;  if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  V oldValue = e.value;  e.value = value;  e.recordAccess(this);  return oldValue;  }  }  modCount++;  // 插入新键值对  addEntry(hash, key, value, i);  return null;  } |

HashMap 允许插入键为 null 的键值对。但是因为无法调用 null 的 hashCode() 方法，也就无法确定该键值对的桶下标，只能通过强制指定一个桶下标来存放。HashMap 使用第 0 个桶存放键为 null 的键值对。

|  |
| --- |
| private V putForNullKey(V value) {  for (Entry<K,V> e = table[0]; e != null; e = e.next) {  if (e.key == null) {  V oldValue = e.value;  e.value = value;  e.recordAccess(this);  return oldValue;  }  }  modCount++;  addEntry(0, null, value, 0);  return null;  } |

使用链表的头插法，也就是新的键值对插在链表的头部，而不是链表的尾部。

|  |
| --- |
| void addEntry(int hash, K key, V value, int bucketIndex) {  if ((size >= threshold) && (null != table[bucketIndex])) {  resize(2 \* table.length);  hash = (null != key) ? hash(key) : 0;  bucketIndex = indexFor(hash, table.length);  }  createEntry(hash, key, value, bucketIndex);  }  void createEntry(int hash, K key, V value, int bucketIndex) {  Entry<K,V> e = table[bucketIndex];  // 头插法，链表头部指向新的键值对  table[bucketIndex] = new Entry<>(hash, key, value, e);  size++;  } |

|  |
| --- |
| Entry(int h, K k, V v, Entry<K,V> n) {  value = v;  next = n;  key = k;  hash = h;  } |

#### 确定桶下标

很多操作都需要先确定一个键值对所在的桶下标。

|  |
| --- |
| int hash = hash(key);  int i = indexFor(hash, table.length); |

##### 计算hash值

|  |
| --- |
| final int hash(Object k) {  int h = hashSeed;  if (0 != h && k instanceof String) {  return sun.misc.Hashing.stringHash32((String) k);  }  h ^= k.hashCode();  // This function ensures that hashCodes that differ only by  // constant multiples at each bit position have a bounded  // number of collisions (approximately 8 at default load factor).  h ^= (h >>> 20) ^ (h >>> 12);  return h ^ (h >>> 7) ^ (h >>> 4);  } |

|  |
| --- |
| public final int hashCode() {  return Objects.hashCode(key) ^ Objects.hashCode(value);  } |

##### 取模

令 x = 1<<4，即 x 为 2 的 4 次方，它具有以下性质：

|  |
| --- |
| x : 00010000  x-1 : 00001111 |

令一个数 y 与 x-1 做与运算，可以去除 y 位级表示的第 4 位以上数：

|  |
| --- |
| y : 10110010  x-1 : 00001111  y&(x-1) : 00000010 |

这个性质和 y 对 x 取模效果是一样的：

|  |
| --- |
| y : 10110010  x : 00010000  y%x : 00000010 |

我们知道，位运算的代价比求模运算小的多，因此在进行这种计算时用位运算的话能带来更高的性能。

确定桶下标的最后一步是将 key 的 hash 值对桶个数取模：hash%capacity，如果能保证 capacity 为 2 的 n 次方，那么就可以将这个操作转换为位运算。

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

#### 扩容-基本原理

设 HashMap 的 table 长度为 M，需要存储的键值对数量为 N，如果哈希函数满足均匀性的要求，那么每条链表的长度大约为 N/M，因此平均查找次数的复杂度为 O(N/M)。

为了让查找的成本降低，应该尽可能使得 N/M 尽可能小，因此需要保证 M 尽可能大，也就是说 table 要尽可能大。HashMap 采用动态扩容来根据当前的 N 值来调整 M 值，使得空间效率和时间效率都能得到保证。

和扩容相关的参数主要有：capacity、size、threshold 和 loadFactor。

|  |  |
| --- | --- |
| 参数 | 含义 |
| capacity | table 的容量大小，默认为 16。需要注意的是 capacity 必须保证为 2 的 n 次方。 |
| size | 键值对的数量 |
| threshold | Size的临界值，当size大于Threshold值时就必须进行扩容操作 |
| loadFactor | 装载因子，table能够使用的比例，threshold = capacity \* loadFacrory |

|  |
| --- |
| static final int DEFAULT\_INITIAL\_CAPACITY = 16;  static final int MAXIMUM\_CAPACITY = 1 << 30;  static final float DEFAULT\_LOAD\_FACTOR = 0.75f;  transient Entry[] table;  transient int size;  int threshold;  final float loadFactor;  transient int modCount; |

从下面的添加元素代码中可以看出，当需要扩容时，令 capacity 为原来的两倍。

|  |
| --- |
| void addEntry(int hash, K key, V value, int bucketIndex) {  Entry<K,V> e = table[bucketIndex];  table[bucketIndex] = new Entry<>(hash, key, value, e);  if (size++ >= threshold)  resize(2 \* table.length);  } |

扩容使用 resize() 实现，需要注意的是，扩容操作同样需要把 oldTable 的所有键值对重新插入 newTable 中，因此这一步是很费时的。

|  |
| --- |
| void resize(int newCapacity) {  Entry[] oldTable = table;  int oldCapacity = oldTable.length;  if (oldCapacity == MAXIMUM\_CAPACITY) {  threshold = Integer.MAX\_VALUE;  return;  }  Entry[] newTable = new Entry[newCapacity];  transfer(newTable);  table = newTable;  threshold = (int)(newCapacity \* loadFactor);  }  void transfer(Entry[] newTable) {  Entry[] src = table;  int newCapacity = newTable.length;  for (int j = 0; j < src.length; j++) {  Entry<K,V> e = src[j];  if (e != null) {  src[j] = null;  do {  Entry<K,V> next = e.next;  int i = indexFor(e.hash, newCapacity);  e.next = newTable[i];  newTable[i] = e;  e = next;  } while (e != null);  }  }  } |

#### 扩容-重新计算桶下标

在进行扩容时，需要把键值对重新放到对应的桶上。HashMap 使用了一个特殊的机制，可以降低重新计算桶下标的操作。

假设原数组长度 capacity 为 16，扩容之后 new capacity 为 32：

|  |
| --- |
| capacity : 00010000  new capacity : 00100000 |

对于一个Key：

* 它的哈希值如果在第 5 位上为 0，那么取模得到的结果和之前一样；
* 如果为 1，那么得到的结果为原来的结果 +16。

#### 计算数组容量

HashMap 构造函数允许用户传入的容量不是 2 的 n 次方，因为它可以自动地将传入的容量转换为 2 的 n 次方。

先考虑如何求一个数的掩码，对于 10010000，它的掩码为 11111111，可以使用以下方法得到：

|  |
| --- |
| mask |= mask >> 1 11011000  mask |= mask >> 2 11111110  mask |= mask >> 4 11111111 |

mask+1 是大于原始数字的最小的 2 的 n 次方。

|  |
| --- |
| num 10010000  mask+1 100000000 |

以下是 HashMap 中计算数组容量的代码：

|  |
| --- |
| static final int tableSizeFor(int cap) {  int n = cap - 1;  n |= n >>> 1;  n |= n >>> 2;  n |= n >>> 4;  n |= n >>> 8;  n |= n >>> 16;  return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;  } |

#### 链表转红黑树

从 JDK 1.8 开始，一个桶存储的链表长度大于 8 时会将链表转换为红黑树。

#### 与HashTable的比较

* HashTable 使用 synchronized 来进行同步。
* HashMap 可以插入键为 null 的 Entry。
* HashMap 的迭代器是 fail-fast 迭代器。
* HashMap 不能保证随着时间的推移 Map 中的元素次序是不变的。

### ConcurrentHashMap

#### 存储结构

|  |
| --- |
| static final class HashEntry<K,V> {  final int hash;  final K key;  volatile V value;  volatile HashEntry<K,V> next;  } |

ConcurrentHashMap和HashMap实现上类似，最主要的差别是ConcurrentHashMap采用了分段锁（Segment），每个分段锁维护着几个桶（HashEntry），多个线程可以同时访问不同分段锁上的桶，从而使其并发度更高（并发度就是 Segment 的个数）。

Segment继承自ReentrantLock。

|  |
| --- |
| static final class Segment<K,V> extends ReentrantLock implements Serializable {  private static final long serialVersionUID = 2249069246763182397L;  static final int MAX\_SCAN\_RETRIES =  Runtime.getRuntime().availableProcessors() > 1 ? 64 : 1;  transient volatile HashEntry<K,V>[] table;  transient int count;  transient int modCount;  transient int threshold;  final float loadFactor;  } |

|  |
| --- |
| final Segment<K,V>[] segments; |

默认的并发级别为16，也就是说默认创建16个Segment。

|  |
| --- |
| static final int DEFAULT\_CONCURRENCY\_LEVEL = 16; |



#### Size操作

每个 Segment 维护了一个 count 变量来统计该 Segment 中的键值对个数。

|  |
| --- |
| /\*\*  \* The number of elements. Accessed only either within locks  \* or among other volatile reads that maintain visibility.  \*/  transient int count; |

在执行size操作时，需要遍历所有的Segment然后把count加起来。

ConcurrentHashMap 在执行 size 操作时先尝试不加锁，如果连续两次不加锁操作得到的结果一致，那么可以认为这个结果是正确的。

尝试次数使用 RETRIES\_BEFORE\_LOCK 定义，该值为 2，retries 初始值为 -1，因此尝试次数为 3。

如果尝试的次数超过 3 次，就需要对每个 Segment 加锁。

|  |
| --- |
| /\*\*  \* Number of unsynchronized retries in size and containsValue  \* methods before resorting to locking. This is used to avoid  \* unbounded retries if tables undergo continuous modification  \* which would make it impossible to obtain an accurate result.  \*/  static final int RETRIES\_BEFORE\_LOCK = 2;  public int size() {  // Try a few times to get accurate count. On failure due to  // continuous async changes in table, resort to locking.  final Segment<K,V>[] segments = this.segments;  int size;  boolean overflow; // true if size overflows 32 bits  long sum; // sum of modCounts  long last = 0L; // previous sum  int retries = -1; // first iteration isn't retry  try {  for (;;) {  // 超过尝试次数，则对每个 Segment 加锁  if (retries++ == RETRIES\_BEFORE\_LOCK) {  for (int j = 0; j < segments.length; ++j)  ensureSegment(j).lock(); // force creation  }  sum = 0L;  size = 0;  overflow = false;  for (int j = 0; j < segments.length; ++j) {  Segment<K,V> seg = segmentAt(segments, j);  if (seg != null) {  sum += seg.modCount;  int c = seg.count;  if (c < 0 || (size += c) < 0)  overflow = true;  }  }  // 连续两次得到的结果一致，则认为这个结果是正确的  if (sum == last)  break;  last = sum;  }  } finally {  if (retries > RETRIES\_BEFORE\_LOCK) {  for (int j = 0; j < segments.length; ++j)  segmentAt(segments, j).unlock();  }  }  return overflow ? Integer.MAX\_VALUE : size;  } |

#### JDK 1.8的改动

JDK 1.7 使用分段锁机制来实现并发更新操作，核心类为 Segment，它继承自重入锁 ReentrantLock，并发度与 Segment 数量相等。

JDK 1.8 使用了 CAS 操作来支持更高的并发度，在 CAS 操作失败时使用内置锁 synchronized。

并且 JDK 1.8 的实现也在链表过长时会转换为红黑树。

### LinkedHashMap

#### 存储结构

继承自 HashMap，因此具有和 HashMap 一样的快速查找特性。

|  |
| --- |
| public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V> |

内部维护了一个双向链表，用来维护插入顺序或者 LRU 顺序。

|  |
| --- |
| /\*\*  \* The head (eldest) of the doubly linked list.  \*/  transient LinkedHashMap.Entry<K,V> head;  /\*\*  \* The tail (youngest) of the doubly linked list.  \*/  transient LinkedHashMap.Entry<K,V> tail; |

accessOrder 决定了顺序，默认为 false，此时维护的是插入顺序。

|  |
| --- |
| final boolean accessOrder; |

LinkedHashMap 最重要的是以下用于维护顺序的函数，它们会在 put、get 等方法中调用。

|  |
| --- |
| void afterNodeAccess(Node<K,V> p) { }  void afterNodeInsertion(boolean evict) { } |

#### afterNodeAccess()

当一个节点被访问时，如果 accessOrder 为 true，则会将该节点移到链表尾部。也就是说指定为 LRU 顺序之后，在每次访问一个节点时，会将这个节点移到链表尾部，保证链表尾部是最近访问的节点，那么链表首部就是最近最久未使用的节点。

|  |
| --- |
| void afterNodeAccess(Node<K,V> e) { // move node to last  LinkedHashMap.Entry<K,V> last;  if (accessOrder && (last = tail) != e) {  LinkedHashMap.Entry<K,V> p =  (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;  p.after = null;  if (b == null)  head = a;  else  b.after = a;  if (a != null)  a.before = b;  else  last = b;  if (last == null)  head = p;  else {  p.before = last;  last.after = p;  }  tail = p;  ++modCount;  }  } |

#### afterNodeInsertion()

在 put 等操作之后执行，当 removeEldestEntry() 方法返回 true 时会移除最晚的节点，也就是链表首部节点 first。

evict 只有在构建 Map 的时候才为 false，在这里为 true。

|  |
| --- |
| void afterNodeInsertion(boolean evict) { // possibly remove eldest  LinkedHashMap.Entry<K,V> first;  if (evict && (first = head) != null && removeEldestEntry(first)) {  K key = first.key;  removeNode(hash(key), key, null, false, true);  }  } |

removeEldestEntry() 默认为 false，如果需要让它为 true，需要继承 LinkedHashMap 并且覆盖这个方法的实现，这在实现 LRU 的缓存中特别有用，通过移除最近最久未使用的节点，从而保证缓存空间足够，并且缓存的数据都是热点数据。

|  |
| --- |
| protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {  return false;  } |

#### LRU缓存

以下是使用 LinkedHashMap 实现的一个 LRU 缓存：

* 设定最大缓存空间 MAX\_ENTRIES 为 3；
* 使用 LinkedHashMap 的构造函数将 accessOrder 设置为 true，开启 LRU 顺序；
* 覆盖 removeEldestEntry() 方法实现，在节点多于 MAX\_ENTRIES 就会将最近最久未使用的数据移除。

|  |
| --- |
| class LRUCache<K, V> extends LinkedHashMap<K, V> {  private static final int MAX\_ENTRIES = 3;  protected boolean removeEldestEntry(Map.Entry eldest) {  return size() > MAX\_ENTRIES;  }  LRUCache() {  super(MAX\_ENTRIES, 0.75f, true);  }  } |

|  |
| --- |
| public static void main(String[] args) {  LRUCache<Integer, String> cache = new LRUCache<>();  cache.put(1, "a");  cache.put(2, "b");  cache.put(3, "c");  cache.get(1);  cache.put(4, "d");  System.out.println(cache.keySet());  }  [3, ,1, 4] |

### WeakHashMap

#### 存储结构

WeakHashMap 的 Entry 继承自 WeakReference，被 WeakReference 关联的对象在下一次垃圾回收时会被回收。

WeakHashMap 主要用来实现缓存，通过使用 WeakHashMap 来引用缓存对象，由 JVM 对这部分缓存进行回收。

|  |
| --- |
| private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> |

#### ConcurrentCache

Tomcat 中的 ConcurrentCache 使用了 WeakHashMap 来实现缓存功能。

ConcurrentCache 采取的是分代缓存：

* 经常使用的对象放入 eden 中，eden 使用 ConcurrentHashMap 实现，不用担心会被回收（伊甸园）；
* 不常用的对象放入 longterm，longterm 使用 WeakHashMap 实现，这些老对象会被垃圾收集器回收。
* 当调用 get() 方法时，会先从 eden 区获取，如果没有找到的话再到 longterm 获取，当从 longterm 获取到就把对象放入 eden 中，从而保证经常被访问的节点不容易被回收。
* 当调用 put() 方法时，如果 eden 的大小超过了 size，那么就将 eden 中的所有对象都放入 longterm 中，利用虚拟机回收掉一部分不经常使用的对象。

|  |
| --- |
| public final class ConcurrentCache<K, V> {  private final int size;  private final Map<K, V> eden;  private final Map<K, V> longterm;  public ConcurrentCache(int size) {  this.size = size;  this.eden = new ConcurrentHashMap<>(size);  this.longterm = new WeakHashMap<>(size);  }  public V get(K k) {  V v = this.eden.get(k);  if (v == null) {  v = this.longterm.get(k);  if (v != null)  this.eden.put(k, v);  }  return v;  }  public void put(K k, V v) {  if (this.eden.size() >= size) {  this.longterm.putAll(this.eden);  this.eden.clear();  }  this.eden.put(k, v);  }  } |

# Java虚拟机

## 运行时数据区域



### 程序计数器

记录正在执行的虚拟机字节码指令的地址（如果正在执行的是本地方法则为空）。

### Java虚拟机栈

每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。



可以通过 -Xss 这个虚拟机参数来指定每个线程的 Java 虚拟机栈内存大小：

|  |
| --- |
| java -Xss512M HackTheJava |

该区域可能抛出以下异常：

* 当线程请求的栈深度超过最大值，会抛出 StackOverflowError 异常；
* 栈进行动态扩展时如果无法申请到足够内存，会抛出 OutOfMemoryError 异常。

### 本地方法栈

本地方法栈与 Java 虚拟机栈类似，它们之间的区别只不过是本地方法栈为本地方法服务。

本地方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为基于本机硬件和操作系统的程序，对待这些方法需要特别处理。



### 堆

所有对象都在这里分配内存，是垃圾收集的主要区域（"GC 堆"）。

现代的垃圾收集器基本都是采用分代收集算法，其主要的思想是针对不同类型的对象采取不同的垃圾回收算法。可以将堆分成两块：

* 新生代（Young Generation）
* 老年代（Old Generation）

堆不需要连续内存，并且可以动态增加其内存，增加失败会抛出 OutOfMemoryError 异常。

可以通过 -Xms 和 -Xmx 这两个虚拟机参数来指定一个程序的堆内存大小，第一个参数设置初始值，第二个参数设置最大值。

|  |
| --- |
| java -Xms1M -Xmx2M HackTheJava  // 堆初始值大小为1MB，最大值为2MB |

### 方法区

用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

和堆一样不需要连续的内存，并且可以动态扩展，动态扩展失败一样会抛出 OutOfMemoryError 异常。

对这块区域进行垃圾回收的主要目标是对常量池的回收和对类的卸载，但是一般比较难实现。

HotSpot 虚拟机把它当成永久代来进行垃圾回收。但很难确定永久代的大小，因为它受到很多因素影响，并且每次 Full GC 之后永久代的大小都会改变，所以经常会抛出 OutOfMemoryError 异常。为了更容易管理方法区，从 JDK 1.8 开始，移除永久代，并把方法区移至元空间，它位于本地内存中，而不是虚拟机内存中。

### 运行时常量池

运行时常量池是方法区的一部分。

Class 文件中的常量池（编译器生成的字面量和符号引用）会在类加载后被放入这个区域。

除了在编译期生成的常量，还允许动态生成，例如 String 类的 intern()。

### 直接内存

在 JDK 1.4 中新引入了 NIO 类，它可以使用 Native 函数库直接分配堆外内存，然后通过 Java 堆里的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在堆内存和堆外内存来回拷贝数据。

## 垃圾收集

垃圾收集主要是针对堆和方法区进行。程序计数器、虚拟机栈和本地方法栈这三个区域属于线程私有的，只存在于线程的生命周期内，线程结束之后就会消失，因此不需要对这三个区域进行垃圾回收。

### 判断一个对象是否可被回收

#### 引用计数法

为对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引用计数为 0 的对象可被回收。

在两个对象出现循环引用的情况下，此时引用计数器永远不为 0，导致无法对它们进行回收。正是因为循环引用的存在，因此 Java 虚拟机不使用引用计数算法。

|  |
| --- |
| public class Test {  public Object instance = null;  public static void main(String[] args) {  Test a = new Test();  Test b = new Test();  a.instance = b;  b.instance = a;  a = null;  b = null;  doSomething();  }  } |

在上述代码中，a 与 b 引用的对象实例互相持有了对象的引用，因此当我们把对 a 对象与 b 对象的引用去除之后，由于两个对象还存在互相之间的引用，导致两个 Test 对象无法被回收。

#### 可达性分析算法

以 GC Roots 为起始点进行搜索，可达的对象都是存活的，不可达的对象可被回收。

Java 虚拟机使用该算法来判断对象是否可被回收，GC Roots 一般包含以下内容：

* 虚拟机栈中局部变量表中引用的对象
* 本地方法栈中 JNI 中引用的对象
* 方法区中类静态属性引用的对象
* 方法区中的常量引用的对象



#### 方法区的回收

因为方法区主要存放永久代对象，而永久代对象的回收率比新生代低很多，所以在方法区上进行回收性价比不高。

主要是对常量池的回收和对类的卸载。

为了避免内存溢出，在大量使用反射和动态代理的场景都需要虚拟机具备类卸载功能。

类的卸载条件很多，需要满足以下三个条件，并且满足了条件也不一定会被卸载：

* 该类所有的实例都已经被回收，此时堆中不存在该类的任何实例。
* 加载该类的 ClassLoader 已经被回收。
* 该类对应的 Class 对象没有在任何地方被引用，也就无法在任何地方通过反射访问该类方法。

#### finalize()

类似 C++ 的析构函数，用于关闭外部资源。但是 try-finally 等方式可以做得更好，并且该方法运行代价很高，不确定性大，无法保证各个对象的调用顺序，因此最好不要使用。

当一个对象可被回收时，如果需要执行该对象的 finalize() 方法，那么就有可能在该方法中让对象重新被引用，从而实现自救。自救只能进行一次，如果回收的对象之前调用了 finalize() 方法自救，后面回收时不会再调用该方法。

### 引用类型

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关。

Java 提供了四种强度不同的引用类型。

#### 强引用

被强引用关联的对象不会被回收。

使用new一个新对象的方式来创建强引用。

|  |
| --- |
| Object obj = new Object(); |

#### 软引用

被软引用关联的对象只有在内存不够的情况下才会被回收。

|  |
| --- |
| Object obj = new Object();  SoftReference<Object> sf = new SoftReference<Object>(obj);  obj = null; // 使对象只被软引用关联 |

#### 弱引用

被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。

使用 WeakReference 类来创建弱引用。

|  |
| --- |
| Object obj = new Object();  WeakReference<Object> wf = new WeakReference<Object>(obj);  obj = null; |

#### 虚引用

又称为幽灵引用或者幻影引用，一个对象是否有虚引用的存在，不会对其生存时间造成影响，也无法通过虚引用得到一个对象。

为一个对象设置虚引用的唯一目的是能在这个对象被回收时收到一个系统通知。

使用 PhantomReference 来创建虚引用。

|  |
| --- |
| Object obj = new Object();  PhantomReference<Object> pf = new PhantomReference<Object>(obj, null);  obj = null; |

### 垃圾收集算法

#### 标记-清除



标记要回收的对象，然后清除。

不足：

* 标记和清除过程效率都不高；
* 会产生大量不连续的内存碎片，导致无法给大对象分配内存。

#### 标记-整理



让所有的存活对象都向一端移动，然后直接清理掉端边界以外的内存。

优点：

* 不会产生内存碎片；

不足：

* 需要移动大量对象，效率低下。

#### 复制



将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理。

不足：

* 只使用一半的内存，内存使用率不高。

现在的商业虚拟机都采用这种收集算法回收新生代，但是并不是划分为大小相等的两块，而是一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。在回收时，将 Eden 和 Survivor 中还存活着的对象全部复制到另一块 Survivor 上，最后清理 Eden 和使用过的那一块 Survivor。

HotSpot 虚拟机的 Eden 和 Survivor 大小比例默认为 8:1，保证了内存的利用率达到 90%。如果每次回收有多于 10% 的对象存活，那么一块 Survivor 就不够用了，此时需要依赖于老年代进行空间分配担保，也就是借用老年代的空间存储放不下的对象。

#### 分代收集

现在的商业虚拟机采用分代收集算法，它根据对象存活周期将内存划分为几块，不同块采用适当的收集算法。

一般将堆分为新生代和老年代。

* 新生代使用：复制算法
* 老年代使用：标记 - 清除 或者 标记 - 整理 算法

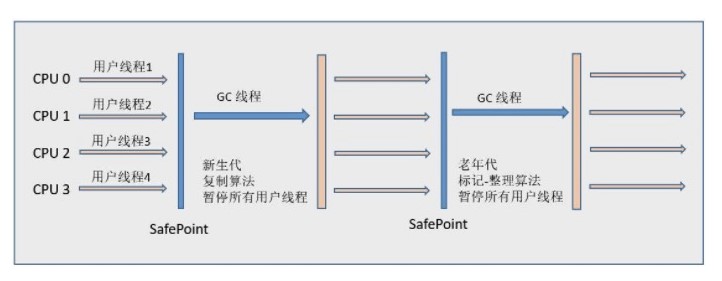
### 垃圾收集器



以上是 HotSpot 虚拟机中的 7 个垃圾收集器，连线表示垃圾收集器可以配合使用。

* 单线程与多线程：单线程指的是垃圾收集器只使用一个线程，而多线程使用多个线程；
* 串行与并行：串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序；并行指的是垃圾收集器和用户程序同时执行。除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。

#### Serial收集器



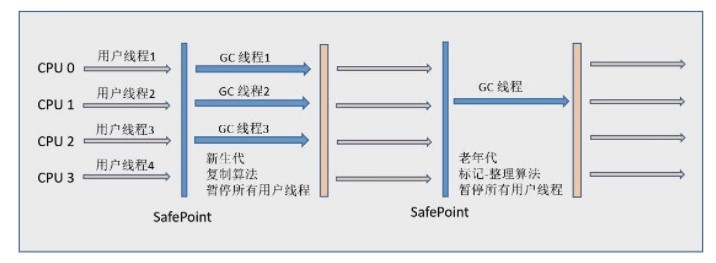
Serial 翻译为串行，也就是说它以串行的方式执行。

它是单线程的收集器，只会使用一个线程进行垃圾收集工作。

它的优点是简单高效，在单个 CPU 环境下，由于没有线程交互的开销，因此拥有最高的单线程收集效率。

它是 Client 场景下的默认新生代收集器，因为在该场景下内存一般来说不会很大。它收集一两百兆垃圾的停顿时间可以控制在一百多毫秒以内，只要不是太频繁，这点停顿时间是可以接受的。

#### ParNew收集器



它是 Serial 收集器的多线程版本。

它是 Server 场景下默认的新生代收集器，除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合使用。

#### Parallel Scavenge收集器

与 ParNew 一样是多线程收集器。

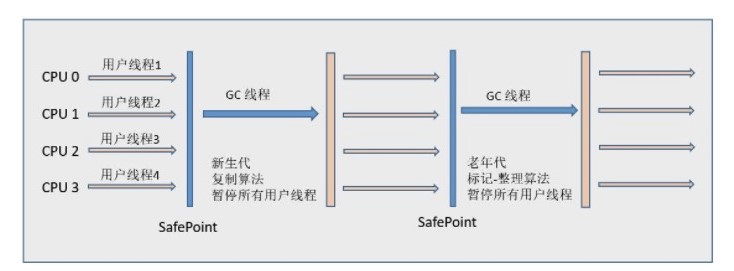
其它收集器目标是尽可能缩短垃圾收集时用户线程的停顿时间，而它的目标是达到一个可控制的吞吐量，因此它被称为“吞吐量优先”收集器。这里的吞吐量指 CPU 用于运行用户程序的时间占总时间的比值。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验。而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，适合在后台运算而不需要太多交互的任务。

缩短停顿时间是以牺牲吞吐量和新生代空间来换取的：新生代空间变小，垃圾回收变得频繁，导致吞吐量下降。

可以通过一个开关参数打开 GC 自适应的调节策略（GC Ergonomics），就不需要手工指定新生代的大小（-Xmn）、Eden 和 Survivor 区的比例、晋升老年代对象年龄等细节参数了。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。

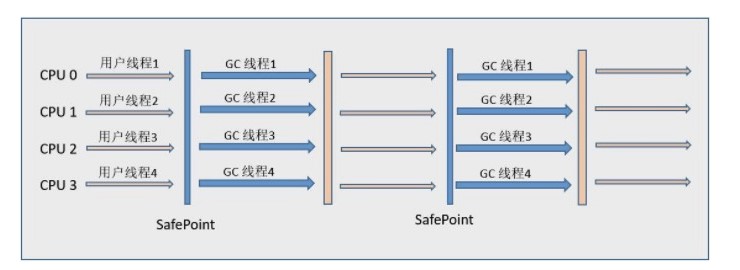
#### Serial Old收集器



是 Serial 收集器的老年代版本，也是给 Client 场景下的虚拟机使用。如果用在 Server 场景下，它有两大用途：

* 在 JDK 1.5 以及之前版本（Parallel Old 诞生以前）中与 Parallel Scavenge 收集器搭配使用。
* 作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

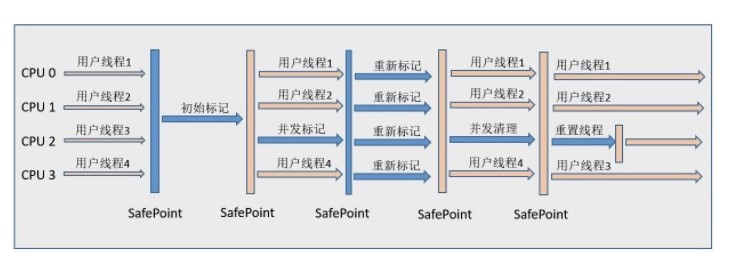
#### Parallel Old收集器



是 Parallel Scavenge 收集器的老年代版本。

在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

#### CMS收集器



CMS（Concurrent Mark Sweep），Mark Sweep 指的是标记 - 清除算法。

分为以下四个流程：

* 初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。
* 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
* 重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿。
* 并发清除：不需要停顿。

在整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，不需要进行停顿。

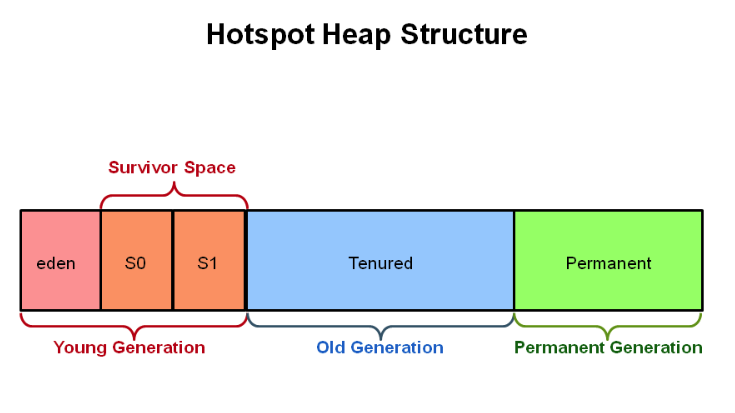
具有以下缺点：

* 吞吐量低：低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高。
* 无法处理浮动垃圾，可能出现 Concurrent Mode Failure。浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。
* 标记 - 清除算法导致的空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。

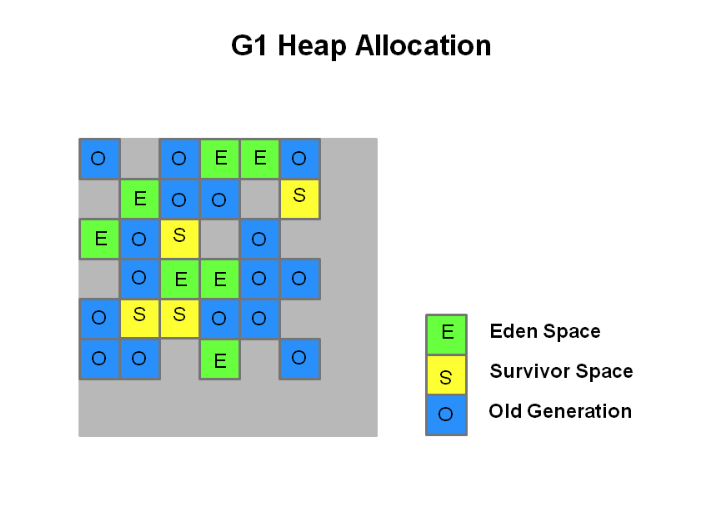
#### G1收集器

G1（Garbage-First），它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能。HotSpot 开发团队赋予它的使命是未来可以替换掉 CMS 收集器。

堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，而 G1 可以直接对新生代和老年代一起回收。

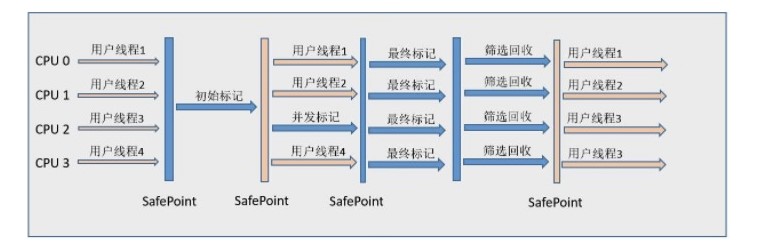


G1 把堆划分成多个大小相等的独立区域（Region），新生代和老年代不再物理隔离。



通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。这种划分方法带来了很大的灵活性，使得可预测的停顿时间模型成为可能。通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region。通过使用 Remembered Set，在做可达性分析的时候就可以避免全堆扫描。



如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：

* 初始标记
* 并发标记
* 最终标记：为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程的 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。这阶段需要停顿线程，但是可并行执行。
* 筛选回收：首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

具备如下特点：

* 空间整合：整体来看是基于“标记 - 整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。
* 可预测的停顿：能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在 GC 上的时间不得超过 N 毫秒。

## 内存分配与回收策略

### Minor GC和Full GC

* Minor GC：回收新生代，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。
* Full GC：回收老年代和新生代，老年代对象其存活时间长，因此 Full GC 很少执行，执行速度会比 Minor GC 慢很多。

### 内存分配策略

#### 对象优先在Eden分配

大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。

#### 大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。

经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

-XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 和 Survivor 之间的大量内存复制。

#### 长期存活的对象进入老年代

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。

-XX:MaxTenuringThreshold 用来定义年龄的阈值。

#### 动态对象年龄绑定

虚拟机并不是永远要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

#### 空间分配担保

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的。

如果不成立的话虚拟机会查看 HandlePromotionFailure 的值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 的值不允许冒险，那么就要进行一次 Full GC。

### Full GC触发的条件

对于 Minor GC，其触发条件非常简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件：

#### 调用System.gc()

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。

#### 老年代空间不足

老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。

为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。

#### 空间分配担保失败

使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC。具体内容请参考上面的第 5 小节。

#### JDK1.7及以前的永久代空间不足

在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。

当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。

为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

#### Concurrent Mode Failure

执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

## 类加载机制

类是在运行期间第一次使用时动态加载的，而不是一次性加载所有类。因为如果一次性加载，那么会占用很多的内存。

### 类的生命周期



包括以下7个阶段：

* **加载（Loading）**
* **验证（Verification）**
* **准备（Preparation）**
* **解析（Resolution）**
* **初始化（Initialization）**
* 使用（Using）
* 卸载（Unloading）、

### 类加载过程

包含了加载、验证、准备、解析和初始化这5个阶段。

#### 加载