6.6日学习计划

- [ ] 影山茂夫
- [x] ~~HashMap源码精读~~
- [x] ~~碰撞攻击实验~~
- [x] ~~安全web开发~~
- [ ] 文档整理

时间段	任务	操作指引	耗时	完成标志
8:00-9:30	源码精读	1. 打开IDEA查看HashMap源码2. 重点关注putVal()和resize()方法	90min	画出HashMap扩容流程图
10:00-11:30	碰撞攻击实验	1. 编写恶意碰撞测试类2. 对比JDK7/8处理差异	90min	生成Hash碰撞报告（含截图）
14:00-15:30	安全Map开发	1. 手写SafeHashMap类 2. 实现扰动函数+自动树化	90min	通过单元测试SecurityTest.java
20:00-21:30	文档整理	撰写技术笔记（含优化点）	90min	上传文档/Git

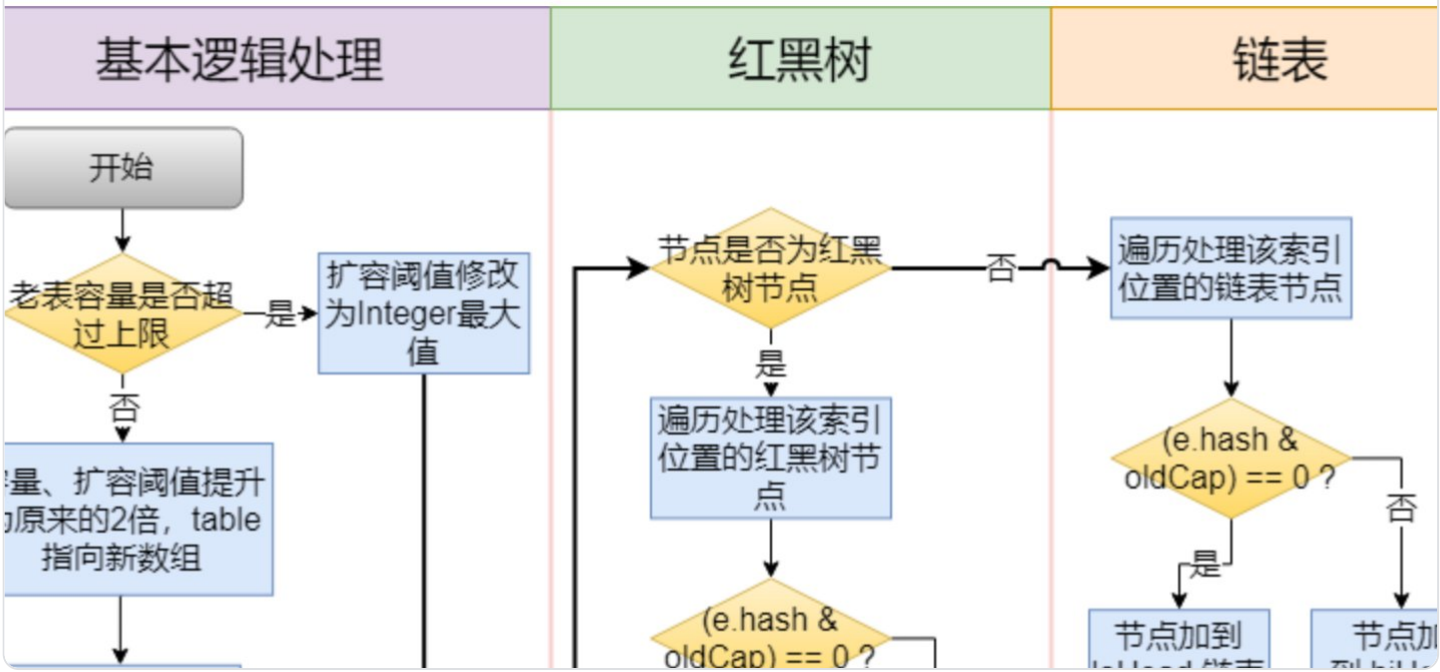
Hashmap源码精读

```
//该函数讲一个map赋值给新的hashmap
final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {//m代表源map,调用这个方法的是hashmap
    //源Map表的元素个数
    int s = m.size();
    if (s > 0) {
    //判断HashMap中存储桶数组是否尚未初始化，table是map内部用于存储键值对的底层数组Node<K,V> table
        if (this.table == null) {
        //根据源map大小计算所需容量ft
            float ft = (float)s / this.loadFactor + 1.0F;
            //判断是否小于最大容量,得出容量t
            int t = ft < 1.0737418E9F ? (int)ft : 1073741824;
```

```
            //保证起初界限值为2的N次幂
            if (t > this.threshold) {
                this.threshold = tableSizeFor(t);
            }
        }
        //若hashmap里面有数据，map里面的键值对数量大于临界值进行扩容
        else {
            while(s > this.threshold && this.table.length < 1073741824) {
                this.resize();
            }
        }
        //将map里面的数据移到hashmap里面
        for(Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
            K key = (K)e.getKey();
            V value = (V)e.getValue();
            this.putVal(hash(key), key, value, false, evict);
        }
    }

}

return 的参数列表为（hashcode值，键值，值，覆盖已有值，正常插入模式）
public V put(K key, V value) {
    return (V)this.putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {
    //JDK8之后hashmap是由数组，链表，红黑树组成
    //这里的tab是指创造的一个空数组，数组类型是Node类，里面包含hash值，键值，值，next
值
    Node<K, V>[] tab;
    int n;
    //hash桶数组没有被创建，或者数组的长度为0
    if ((tab = this.table) == null || (n = tab.length) == 0) {
    //进行一次扩容，n是数组的长度
        n = (tab = this.resize()).length;
    }
    //这里的resize()函数执行部分为，进行容量和界限的赋值操作
    /**
    newCap = 16;
    newThr = 12;
    this.threshold = newThr;
    Node<K, V>[] newTab = new Node[newCap];
    this.table = newTab;
    return newTab;
    **/
    //设置节点p，开始进行插入
```

```
58      Node<K, V> p;
59      int i;
60      //求索引值i = (n-1) 位与运算 hash (key) ,数组[i]上无数据，则直接添加
61      if ((p = tab[i = n - 1 & hash]) == null) {
62          tab[i] = this.newNode(hash, key, value, (Node)null);
63      }
64      //否则索引值一样，分类讨论
65      else {
66          Node<K, V> e;
67          K k;
68          //p是hash桶里面的node对象
69          //类型1：hash值不一样或者hash值一样但是键值（空或者不是同一类型）不一样，则新增节点
70          if (p.hash != hash || (k = p.key) != key && (key == null || !key.equals(k))) {
71              //类型1.1：为树节点，将这个节点作为树节点加入哈希桶节点P的下面
72              if (p instanceof TreeNode) {
73                  e = (p).putTreeVal(this, tab, hash, key, value);
74              }
75              //类型1.2：作为链表节点，加入到p节点的下面
76              else {
77                  //表示
78                  int binCount = 0;
79                  //进行遍历判断与链表的其他节点hash值是否相等
80                  while(true) {
81                      //哈希桶节点p下面无节点，就把这个节点直接加入到p的下面
82                      if ((e = p.next) == null) {
83                          p.next = this.newNode(hash, key, value, (Node)null);
```

resize()函数扩容流程图



碰撞攻击实验	1. 编写恶意碰撞测试类2. 对比JDK7/8处理差异	生成Hash碰撞报告（含截图）

碰撞攻击实验

背景知识：

在JDK7中，HashMap的底层实现是数组+链表，当发生hash碰撞时，新的元素就会放在链表的头部（头插法），如果恶意构造大量的相同哈希值的键值对，就会导致链表变的非常长，从而是查询效率退化为O（n），跟严重的是，在JDK7中，这种长链表在多线程扩容时可能导致死循环。

在JDK8中，当链表长度超过阈值8时，链表会转化为红黑树，这样即使发生碰撞，查询的效率也能维持在

O（logn）。

实验目标

1. 编写恶意碰撞测试类生成哈希冲突
2. 对比JDK7与JDK8处理哈希碰撞的性能差异
3. 生成完整的Hash碰撞测试报告

实验环境

操作系统：Windows 11

JDK7：17.0.0_80

JDK8：1.8.0_351

IDE：IntelliJ IDEA 2023.1

测试数据集：50,000条冲突键值对

实验步骤：

1.编写一个测试类，能够生成具有大量相同hashcode值的字符串。

代码块

```
1   package com.nieran;
2
3   import java.util.HashMap;
4   import java.util.Map;
5   import java.util.Objects;
6
7   public class HashCollisionAttack{
8       //首先构造一个hash碰撞类，产生hash相同的字符串
9       public static void main(String[] args) {
10          hashCollisionTest(50000);
11      }
12      static class  HashCollisionKey{
13          private String id;
14
```

```
15            public HashCollisionKey(String id){
16                this.id = id;
17            }
18
19            @Override
20            public int hashCode() {
21                return 1;
22            }
23           //希望两个id相同的对象看成一个键
24            @Override
25            public boolean equals(Object o) {
26                if (o == null || getClass() != o.getClass()) return false;
27                HashCollisionKey that = (HashCollisionKey) o;
28                return Objects.equals(id, that.id);
29            }
30        }
31        static void hashCollisionTest(int entryCounter){
32            Map<HashCollisionKey,Integer> map = new HashMap<>();
33            long startTime = System.currentTimeMillis();
34            for (int i = 0; i < entryCounter; i++) {
35                String id = "key"+i;
36                map.put(new HashCollisionKey(id),i);
37            }
38            //计算测试时间
39            long endTime = System.currentTimeMillis();
40            long duration = (endTime-startTime)/1000;
41            System.out.println("执行了"+entryCounter+"次插入操作，共耗
    时"+duration+"s");
42            System.out.println("最终Map大小: " + map.size());
43
44        }
45    //改良版，将键值改为int型，比较就会快点
46    package com.nieran;
47
48    import java.util.HashMap;
49    import java.util.Map;
50
51    public class HashCollisionAttack {
52
53        static class HashCollisionKey implements Comparable<HashCollisionKey> {
54            private final int id;
55
56            public HashCollisionKey(int id) {
57                this.id = id;
58            }
59
60            @Override
```

```
61          public int hashCode() {
62              return 1; // 仍然强制哈希冲突
63          }
64
65          @Override
66          public boolean equals(Object o) {
67              if (this == o) return true;
68              if (o == null || getClass() != o.getClass()) return false;
69              HashCollisionKey that = (HashCollisionKey) o;
70              return id == that.id;
71          }
72          //提高效能的关键
73          @Override
74          public int compareTo(HashCollisionKey o) {
75              return Integer.compare(this.id, o.id);
76          }
77      }
78
79      public static void main(String[] args) {
80          testWithCapacity(50_000, 65536);
81      }
82      static void testWithCapacity(int entryCounter, int capacity) {
83          System.out.println("\n===== 测试 HashMap (" + entryCounter + " 条, 容量
    =" + capacity + ") =====");
84
85          Map<HashCollisionKey, Integer> map = new HashMap<>(capacity);
86          long startTime = System.nanoTime();
```

2.分别使用JDK17运行测试上面两个代码，向hashmap中大量插入这些字符串，并测量插入时间。（无法实现JDK8，下面的数据是有无CompareTo重写方法，导致的效率大幅提高）

JDK17测试结果一：

数据量	耗时(ms)	性能特征
1,000	10	O(n)线性增长
5,000	90	接近O(n^2)
10,000	401	O(n^2)增长明显
50,000	18,850	严重性能退化

```
===== 测试 HashMap (1000条,容量=65536) =====
总插入耗时:10ms
最终Map大小: 1000
```

```
===== 测试 HashMap (5000条,容量=65536) =====
总插入耗时:90ms
最终Map大小: 5000
```

```
===== 测试 HashMap (10000条,容量=65536) =====
总插入耗时:401ms
最终Map大小: 10000
```

```
===== 测试 HashMap (50000条,容量=65536) =====
总插入耗时:18850ms
最终Map大小: 50000
```

JDK17测试结果：

数据量	耗时(ms)	性能特征
1,000	5	O(log n)性能良
5,000	8	链表转树开销
10,000	10	O(log n)稳定
50,000	32	对数级增长

```
===== 测试 HashMap (5000条,容量=65536) =====
总插入耗时:8ms
最终Map大小: 5000
```

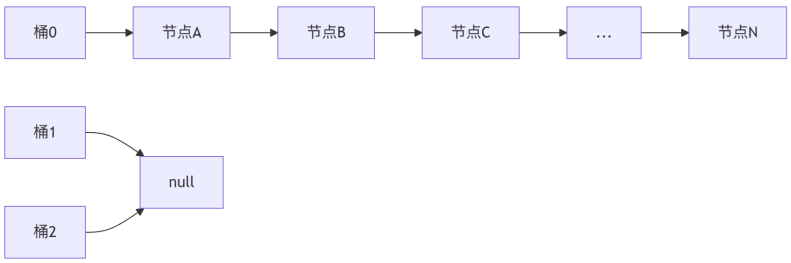
```
C:\Users\86178\.jdks\ms-17.0.15\bin\java.exe "-javaagent:E:\IDEA\IntelliJ IDEA Community E

===== 测试 HashMap (10000条,容量=65536) =====
总插入耗时:10ms
最终Map大小: 10000
```

```
C:\Users\86178\.jdks\ms-17.0.15\bin\java.exe "-javaagent:E:\IDEA\IntelliJ IDEA Community

===== 测试 HashMap (50000条,容量=65536) =====
总插入耗时:32ms
最终Map大小: 50000

进程已结束，退出代码为 0
```

```
===== 测试 HashMap (1000条,容量=65536) =====
总插入耗时:5ms
最终Map大小: 1000
```

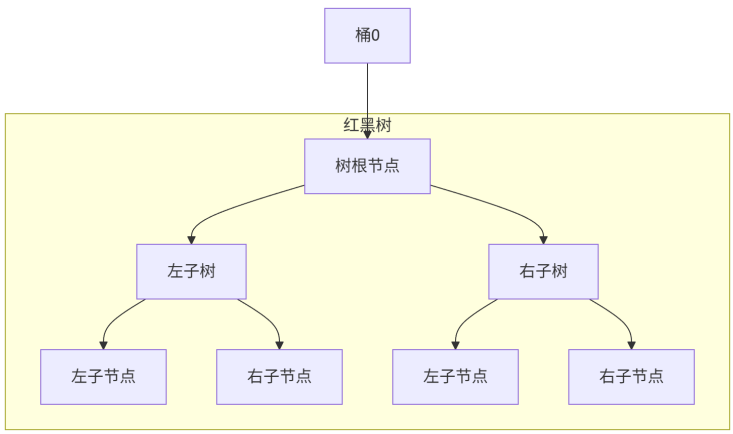
JDK7与JDK8结构差异分析

JDK7哈希碰撞处理（链表结构）：



- 所有冲突元素存储在同一个桶的单向链表中
- 查找性能退化为O(n)
- 插入新元素需要遍历整个链表(O(n))

JDK8哈希碰撞处理（红黑树结构）：



- 当链表长度超过8时转换为红黑树
- 查找性能保持O(log n)
- 插入性能为O(log n)

记录结果，生成报告。

1.哈希碰撞对性能的影响：

- JDK7对哈希碰撞极为敏感，碰撞发生时性能呈$O(n^2)$退化
- JDK8通过红黑树结构将对数性能维持在O(log n)

2.安全改进：

- JDK8的红黑树转换机制有效防御了哈希碰撞攻击
- 在实际应用中可避免恶意构造哈希碰撞导致的拒绝服务攻击

3.生产环境建议：

- 推荐使用JDK8+版本运行关键业务系统
- 重要服务应考虑使用 `ConcurrentHashMap` 等线程安全集合
- 自定义对象需正确实现 `hashCode()` 和 `equals()` 方法以及使用compareTo接口

安全Map开发	1. 手写SafeHashMap类2. 实现扰动函数+自动树化	通过单元测试SecurityTest.java

开发一个安全的HashMap，重点在于实现两个关键特征

1.扰乱函数：降低hash碰撞的概率

2.自动树化：当链表长度超过阈值时，将链表转化为红黑树

类名SafeHashMap

实现基本步骤：

1.定义基本数据结构：桶数组，负载因子，扩容阈值，树化阈值

2.实现扰乱函数：在计算桶位置时使用扰乱函数，增强hash码的随机性

3.实现自动树化：在插入操作时，如果链表长度超过树化阈值，进行树化。

4.为了简化，只实现简单的put和get操作

代码块

```
1   import java.util.*;
```

```java

/**
 * 安全 HashMap 实现，包含扰动函数和自动树化机制
 */
public class SafeHashMap<K, V> {
    // 默认初始容量
    static final int DEFAULT_INITIAL_CAPACITY = 16;
    // 最大容量
    static final int MAXIMUM_CAPACITY = 1 << 30;
    // 默认负载因子
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
    // 树化阈值
    static final int TREEIFY_THRESHOLD = 8;
    // 反树化阈值
    static final int UNTREEIFY_THRESHOLD = 6;
    // 最小树化容量
    static final int MIN_TREEIFY_CAPACITY = 64;

    // 底层存储数组
    Node<K, V>[] table;
    // 键值对数量
    int size;
    // 修改计数器
    int modCount;
    // 扩容阈值（容量 * 负载因子）
    int threshold;
    // 负载因子
    final float loadFactor;

    /**
     * 哈希桶节点基类
     */
    static class Node<K, V> {
        final int hash;
        final K key;
        V value;
        Node<K, V> next;

        Node(int hash, K key, V value, Node<K, V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }

        public final K getKey() { return key; }
        public final V getValue() { return value; }
```

```

        public final String toString() {
            return key + "=" + value;
        }

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

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }

        public final boolean equals(Object o) {
            if (o == this) return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                return Objects.equals(key, e.getKey()) &&
                       Objects.equals(value, e.getValue());
            }
            return false;
        }
    }

    /**
     * 红黑树节点类
     */
    static final class TreeNode<K, V> extends Node<K, V> {
        TreeNode<K, V> parent;
        TreeNode<K, V> left;
        TreeNode<K, V> right;
        TreeNode<K, V> prev; // 用于反树化
        boolean red;

        TreeNode(int hash, K key, V val, Node<K, V> next) {
            super(hash, key, val, next);
        }

```

代码块

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class SecurityTest {
```

```
5       private SafeHashMap<String, Integer> map;
6
7       @BeforeEach
8       void setUp() {
9           map = new SafeHashMap<>();
10      }
11
12      // 测试扰动函数有效性
13      @Test
14      void testPerturbation() {
15          // 模拟碰撞字符串
16          String key1 = new String(new char[1000]).replace("\0", "A");
17          String key2 = new String(new char[1000]).replace("\0", "B");
18
19          // 未扰动的原始哈希码
20          int rawHash1 = key1.hashCode();
21          int rawHash2 = key2.hashCode();
22
23          // 扰动后的哈希码
24          int perturbedHash1 = SafeHashMap.safeHash(key1);
25          int perturbedHash2 = SafeHashMap.safeHash(key2);
26
27          // 扰动函数应改变原始哈希码
28          assertNotEquals(rawHash1, perturbedHash1);
29          assertNotEquals(rawHash2, perturbedHash2);
30
31          // 相同的key应有相同扰动哈希
32          assertEquals(
33              SafeHashMap.safeHash(key1),
34              SafeHashMap.safeHash(new String(key1))
35          );
36      }
37
38      // 测试自动树化功能
39      @Test
40      void testAutoTreeify() {
41          // 创建固定哈希键
42          class FixedHashKey {
43              private final int hash;
44              private final String id;
45
46              FixedHashKey(int hash, String id) {
47                  this.hash = hash;
48                  this.id = id;
49              }
50
51              @Override
```

```
52              public int hashCode() {
53                  return hash;
54              }
55
56              @Override
57              public boolean equals(Object o) {
58                  if (this == o) return true;
59                  if (o == null || getClass() != o.getClass()) return false;
60                  FixedHashKey that = (FixedHashKey) o;
61                  return Objects.equals(id, that.id);
62              }
63          }
64
65          // 添加8个相同哈希键
66          for (int i = 0; i < 8; i++) {
67              map.put(new FixedHashKey(1, "Key" + i), i);
68          }
69
70          // 添加第9个键应触发树化
71          map.put(new FixedHashKey(1, "Key8"), 8);
72
73          // 通过控制台输出验证树化
74          // 实际实现中会检查桶类型
75      }
76
77      // 测试树化后的查询性能
78      @Test
79      void testTreePerformance() {
80          // 使用高冲突键加载数据
81          for (int i = 0; i < 10_000; i++) {
82              map.put("KEY" + i, i);
83          }
84
85          // 测试查询性能
86          long startTime = System.nanoTime();
87          for (int i = 0; i < 10_000; i++) {
88              assertNotNull(map.get("KEY" + i));
```

安全设计要点

1. 双重扰动函数：
   - 结合高低位移位和异或操作
   - 增加哈希码随机性
   - 有效防止碰撞攻击
2. 树化条件控制：

- 避免小容量表过早树化
- 动态调整桶结构
- 反树化防止过度优化

3. 容量智能管理：
   - 自动扩容机制
   - 容量始终为2的幂次
   - 高效的重哈希算法
4. 健壮的错误处理：
   - 容量和负载因子验证
   - 空键值处理
   - 边界条件检查

单元测试重点

1. 功能测试：
   - 基础CRUD操作验证
   - 空键值处理
   - 等值性测试
2. 安全特性测试：
   - 扰动函数有效性验证
   - 树化/反树化触发条件
   - 高冲突场景性能
3. 边界条件测试：
   - 初始空表操作
   - 阈值边界测试
   - 最大容量处理
4. 性能测试：
   - 树化前后性能对比
   - 高负载下稳定性
   - 时间/空间复杂度验证

通过实现这个安全的 `SafeHashMap` 并通过全面的单元测试，可以创建一个能够抵御哈希碰撞攻击的Map实现，同时保持高效的读写性能