

# 2019 一线互联网 Java 面试题解析大全

## 1. HashMap

(1) 问：HashMap 有用过吗？您能给我说说他的主要用途吗？

答：有用过，我在平常工作中经常会用到 HashMap 这种数据结构，HashMap 是基于 Map 接口实现的一种键-值对<key,value>的存储结构，允许 null 值，同时非有序，非同步(即线程不安全)。HashMap 的底层实现是数组 + 链表 + 红黑树（JDK1.8 增加了红黑树部分）。它存储和查找数据时，是根据键 key 的 hashCode 的值计算出具体存储位置。HashMap 最多只允许一条记录的键 key 为 null，HashMap 增删改查等常规操作都有不错的执行效率，是 ArrayList 和 LinkedList 等数据结构的一种折中实现。

示例代码：

```
// 创建一个 HashMap，如果没有指定初始大小，默认底层 hash 表数组的大小为 16
HashMap<String, String> hashMap = new HashMap<String, String>();
// 往容器里面添加元素
hashMap.put("小明", "好帅");
hashMap.put("老王", "坑爹货");
hashMap.put("老铁", "没毛病");
hashMap.put("掘金", "好地方");
hashMap.put("王五", "别搞事");
// 获取 key 为小明的元素 好帅
String element = hashMap.get("小明");
// value : 好帅
System.out.println(element);
// 移除 key 为王五的元素
String removeElement = hashMap.remove("王五");
// value : 别搞事
System.out.println(removeElement);
// 修改 key 为小明的元素的值 value 为 其实有点丑
hashMap.replace("小明", "其实有点丑");
// {老铁=没毛病, 小明=其实有点丑, 老王=坑爹货, 掘金=好地方}
System.out.println(hashMap);
// 通过 put 方法也可以达到修改对应元素的值的效果
hashMap.put("小明", "其实还可以啦, 开玩笑的");
```

```
// {老铁=没毛病, 小明=其实还可以啦, 开玩笑的, 老王=坑爹货, 掘金=好地方}
```

```
System.out.println(hashMap);  
// 判断 key 为老王的元素是否存在(捉奸老王)  
boolean isExist = hashMap.containsKey("老王");  
// true , 老王竟然来搞事  
System.out.println(isExist);  
// 判断是否有 value = "坑爹货" 的人  
boolean isHasSomeOne = hashMap.containsValue("坑爹货");  
// true 老王是坑爹货  
System.out.println(isHasSomeOne);  
// 查看这个容器里面还有几个家伙 value : 4  
System.out.println(hashMap.size());
```

- HashMap 的底层实现是数组 + 链表 + 红黑树 (JDK1.8 增加了红黑树部分), 核心组成元素有:

int size; 用于记录 HashMap 实际存储元素的个数;

float loadFactor; 负载因子 (默认是 0.75, 此属性后面详细解释)。

int threshold; 下一次扩容时的阈值, 达到阈值便会触发扩容机制 `resize` (阈值 `threshold = 容器容量 capacity * 负载因子 load factor`)。也就是说, 在容器定义好容量之后, 负载因子越大, 所能容纳的键值对元素个数就越多。

Node<K,V>[] table; 底层数组, 充当哈希表的作用, 用于存储对应 hash 位置的元素 Node<K,V>, 此数组长度总是 2 的 N 次幂。(具体原因后面详细解释)

示例代码:

```
public class HashMap<K,V> extends AbstractMap<K,V>  
    implements Map<K,V>, Cloneable, Serializable {  
    . . . . .  
  
    /* ----- Fields ----- */  
  
    /**  
     * 哈希表, 在第一次使用到时进行初始化, 重置大小是必要的操作,  
     * 当分配容量时, 长度总是 2 的 N 次幂。  
     */  
    transient Node<K,V>[] table;  
  
    /**
```

```

        * 实际存储的 key - value 键值对 个数
        */
        transient int size;

        /**
         * 下一次扩容时的阈值
         * (阈值 threshold = 容器容量 capacity * 负载因子 load factor).
         * @serial
         */
        int threshold;

        /**
         * 哈希表的负载因子
         *
         * @serial
         */
        final float loadFactor;

        . . . . .}

```

- 其中 `Node<K,V>[] table`; 哈希表存储的核心元素是 `Node<K,V>`, `Node<K,V>` 包含:

`final int hash`; 元素的哈希值, 决定元素存储在 `Node<K,V>[] table`; 哈希表中的位置。由 `final` 修饰可知, 当 `hash` 的值确定后, 就不能再修改。

`final K key`; 键, 由 `final` 修饰可知, 当 `key` 的值确定后, 就不能再修改。

`V value`; 值

`Node<K,V> next`; 记录下一个元素结点(单链表结构, 用于解决 `hash` 冲突)

示例代码:

```

/**
 * 定义 HashMap 存储元素结点的底层实现
 */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;//元素的哈希值 由 final 修饰可知, 当 hash 的值确定
    后, 就不能再修改
    final K key;// 键, 由 final 修饰可知, 当 key 的值确定后, 就不能再
    修改

```

V value; // 值  
Node<K,V> next; // 记录下一个元素结点(单链表结构, 用于解决 hash  
冲突)

```
/**
 * Node 结点构造方法
 */
Node(int hash, K key, V value, Node<K,V> next) {
    this.hash = hash;//元素的哈希值
    this.key = key;// 键
    this.value = value; // 值
    this.next = next;// 记录下一个元素结点
}

public final K getKey()      { return key; }
public final V getValue()    { return value; }
public final String toString() { return key + "=" + value; }

/**
 * 为 Node 重写 hashCode 方法, 值为: key 的 hashCode 异或 value
    的 hashCode
 * 运算作用就是将 2 个 hashCode 的二进制中, 同一位置相同的值为 0,
    不同的为 1。
 */
public final int hashCode() {
    return Objects.hashCode(key) ^ Objects.hashCode(value);
}

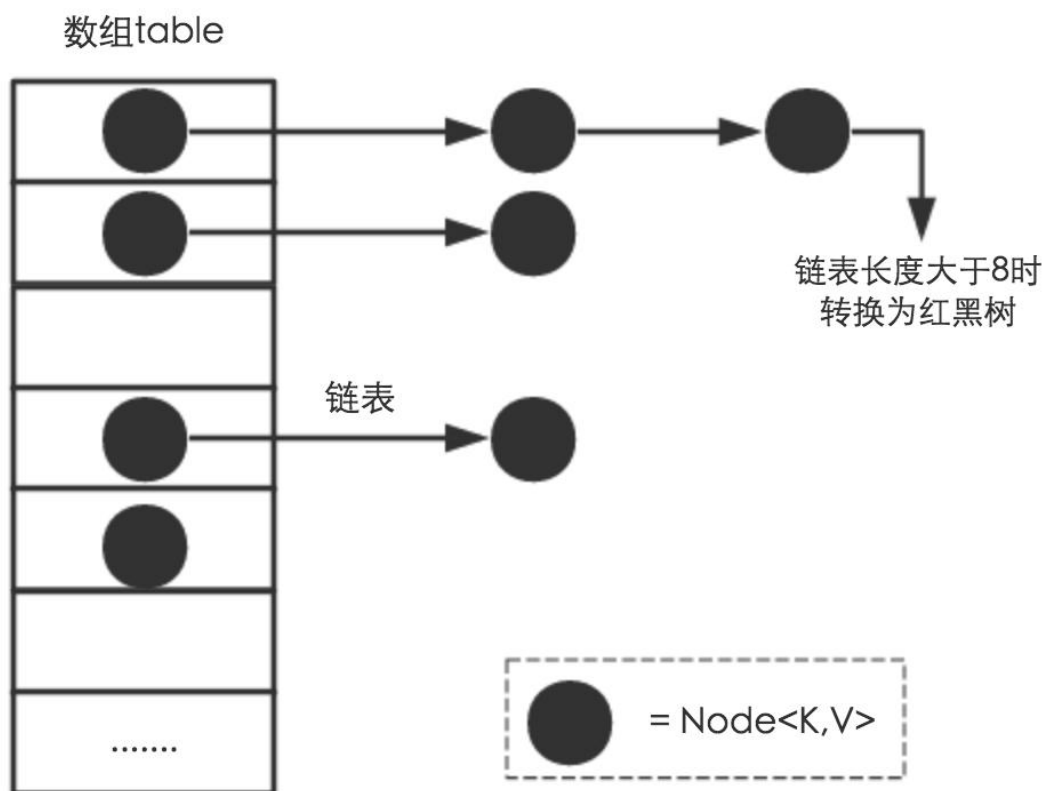
/**
 * 修改某一元素的值
 */
public final V setValue(V newValue) {
    V oldValue = value;
    value = newValue;
    return oldValue;
}

/**
 * 为 Node 重写 equals 方法
 */
public final boolean equals(Object o) {
    if (o == this)
        return true;
```

```

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

```



hashMap 内存结构图 - 图片来自于《美团点评技术团队文章》

2.问：您能说说 HashMap 常用操作的底层实现原理吗？如存储 `put(K key, V value)`，查找 `get(Object key)`，删除 `remove(Object key)`，修改 `replace(K key, V value)`等操作。

答：调用 `put(K key, V value)` 操作添加 key-value 键值对时，进行了如下操作：

判断哈希表 `Node<K,V>[] table` 是否为空或者 `null`，是则执行 `resize()` 方法进行扩容。

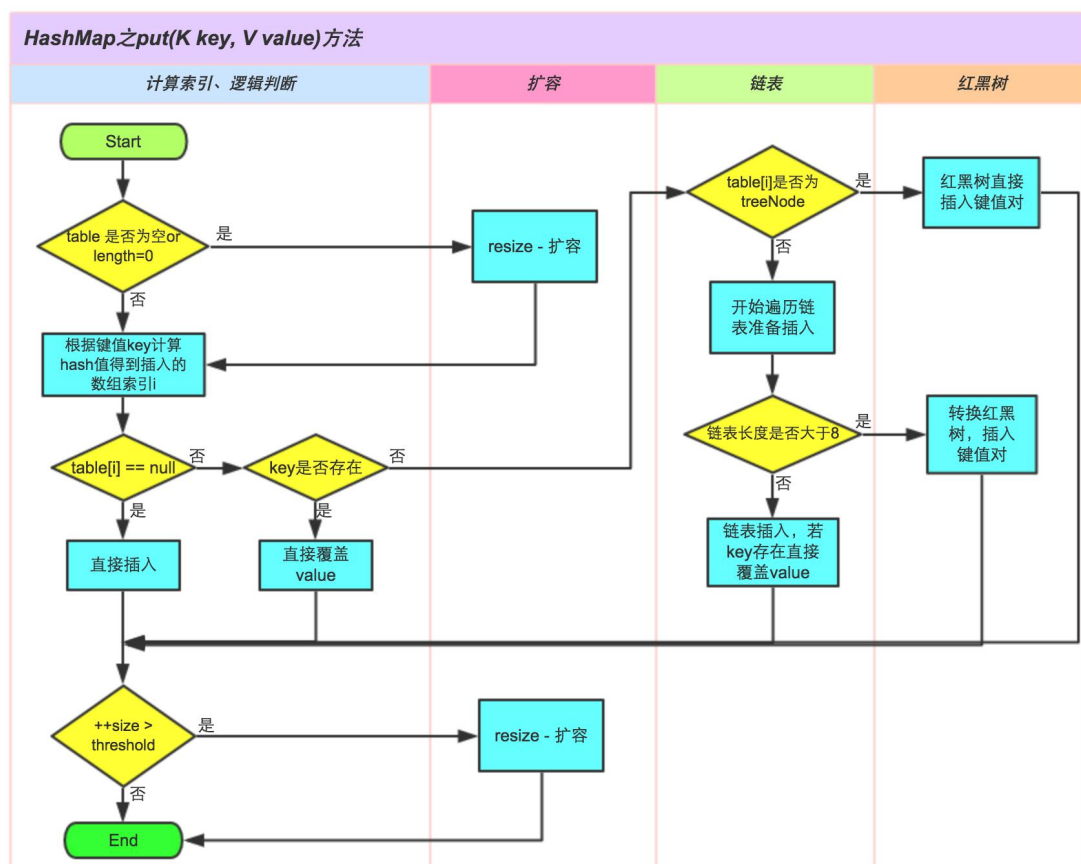
根据插入的键值 `key` 的 `hash` 值，通过  $(n - 1) \& \text{hash}$  当前元素的 `hash` 值  $\& \text{hash}$  表长度 - 1（实际就是 `hash` 值  $\% \text{hash}$  表长度）计算出存储位置 `table[i]`。如果存储位置没有元素存放，则将新增结点存储在此位置 `table[i]`。

如果存储位置已经有键值对元素存在，则判断该位置元素的 hash 值和 key 值是否和当前操作元素一致，一致则证明是修改 value 操作，覆盖 value 即可。

当前存储位置即有元素，又不和当前操作元素一致，则证明此位置 table[i] 已经发生了 hash 冲突，则通过判断头结点是否是 treeNode，如果是 treeNode 则证明此位置的结构是红黑树，已红黑树的方式新增结点。

如果不是红黑树，则证明是单链表，将新增结点插入至链表的最后位置，随后判断当前链表长度是否 大于等于 8，是则将当前存储位置的链表转化为红黑树。遍历过程中如果发现 key 已经存在，则直接覆盖 value。

插入成功后，判断当前存储键值对的数量 大于 阈值 threshold 是则扩容。



hashMap put 方法执行流程图- 图片来自于《美团点评技术团队文章》

示例代码：

```
/**
 * 添加 key-value 键值对
 *
 * @param key 键
```

```

    * @param value 值
    * @return 如果原本存在此 key, 则返回旧的 value 值, 如果是新增的 key-
    *         value, 则返回 null
    */
    public V put(K key, V value) {
        //实际调用 putVal 方法进行添加 key-value 键值对操作
        return putVal(hash(key), key, value, false, true);
    }

    /**
     * 根据 key 键的 hashCode 通过 “扰动函数” 生成对应的 hash 值
     * 经过此操作后, 使每一个 key 对应的 hash 值生成的更均匀,
     * 减少元素之间的碰撞几率 (后面详细说明)
     */
    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }

    /**
     * 添加 key-value 键值对的实际调用方法 (重点)
     *
     * @param hash key 键的 hash 值
     * @param key 键
     * @param value 值
     * @param onlyIfAbsent 此值如果是 true, 则如果此 key 已存在 value,
    则不执
     * 行修改操作
     * @param evict 此值如果是 false, 哈希表是在初始化模式
     * @return 返回原本的旧值, 如果是新增, 则返回 null
     */
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        // 用于记录当前的 hash 表
        Node<K, V>[] tab;
        // 用于记录当前的链表结点
        Node<K, V> p;
        // n 用于记录 hash 表的长度, i 用于记录当前操作索引 index
        int n, i;
        // 当前 hash 表为空
        if ((tab = table) == null || (n = tab.length) == 0)
            // 初始化 hash 表, 并把初始化后的 hash 表长度值赋值给 n
            n = (tab = resize()).length;
    }

```

```

// 1) 通过 (n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1
// 2) 确定当前元素的存储位置, 此运算等价于 当前元素的 hash 值 %
hash 表的长度
// 3) 计算出的存储位置没有元素存在
if ((p = tab[i = (n - 1) & hash]) == null)
    // 4) 则新建一个 Node 结点, 在该位置存储此元素
    tab[i] = newNode(hash, key, value, null);
else { // 当前存储位置已经有元素存在了(不考虑是修改的情况的话,
就代表发生 hash 冲突了)
    // 用于存放新增结点
    Node<K, V> e;
    // 用于临时存在某个 key 值
    K k;
    // 1) 如果当前位置已存在元素的 hash 值和新增元素的 hash 值相等
    // 2) 并且 key 也相等, 则证明是同一个 key 元素, 想执行修改 value
操作
    if (p.hash == hash &&
        ((k = p.key) == key || (key != null && key.equals(k))))
        e = p; // 将当前结点引用赋值给 e
    else if (p instanceof TreeNode) // 如果当前结点是树结点
        // 则证明当前位置的链表已变成红黑树结构, 则已红黑树结点
结构新增元素
        e = ((TreeNode<K, V>)p).putTreeVal(this, tab, hash, key,
value);
    else { // 排除上述情况, 则证明已发生 hash 冲突, 并 hash 冲突位
置现时的结构是单链表结构
        for (int binCount = 0; ; ++binCount) {
            // 遍历单链表, 将新元素结点放置此链表的最后一位
            if ((e = p.next) == null) {
                // 将新元素结点放在此链表的最后一位
                p.next = newNode(hash, key, value, null);
                // 新增结点后, 当前结点数量是否大于等于 阈值 8
                if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                    // 大于等于 8 则将链表转换成红黑树
                    treeifyBin(tab, hash);
                break;
            }
            // 如果链表中已经存在对应的 key, 则覆盖 value
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                break;
            p = e;
        }
    }
}

```



```

        if (e != null) { // 已存在对应 key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null) //如果允许修改，
            则修改 value 为新值
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    // 当前存储键值对的数量 大于 阈值 是则扩容
    if (++size > threshold)
        // 重置 hash 大小,将旧 hash 表的数据逐一复制到新的 hash 表中(后面详细讲解)
        resize();
    afterNodeInsertion(evict);
    // 返回 null, 则证明是新增操作, 而不是修改操作
    return null;
}

```

- 调用 `get(Object key)` 操作根据键 `key` 查找对应的 `key-value` 键值对时, 进行了如下操作:

先调用 `hash(key)` 方法计算出 `key` 的 `hash` 值

根据查找的键值 `key` 的 `hash` 值, 通过  $(n - 1) \& \text{hash}$  当前元素的 `hash` 值  $\& \text{hash 表长度} - 1$  (实际就是 `hash 值 % hash 表长度`) 计算出存储位置 `table[i]`, 判断存储位置是否有元素存在。

如果存储位置有元素存放, 则首先比较头结点元素, 如果头结点的 `key` 的 `hash` 值 和 要获取的 `key` 的 `hash` 值相等, 并且 头结点的 `key` 本身 和要获取的 `key` 相等, 则返回该位置的头结点。

如果存储位置没有元素存放, 则返回 `null`。

如果存储位置有元素存放, 但是头结点元素不是要查找的元素, 则需要遍历该位置进行查找。

先判断头结点是否是 `treeNode`, 如果是 `treeNode` 则证明此位置的结构是红黑树, 以红色树的方式遍历查找该结点, 没有则返回 `null`。

如果不是红黑树, 则证明是单链表。遍历单链表, 逐一比较链表结点, 链表结点的 `key` 的 `hash` 值 和 要获取的 `key` 的 `hash` 值相等, 并且 链表结点的 `key` 本身 和要获取的 `key` 相等, 则返回该结点, 遍历结束仍未找到对应 `key` 的结点, 则返回 `null`。

示例代码：

```
/**
 * 返回指定 key 所映射的 value 值
 * 或者 返回 null 如果容器里不存在对应的 key
 *
 * 更确切地讲，如果此映射包含一个满足 (key==null ?
k==null :key.equals(k))
 * 的从 k 键到 v 值的映射关系，
 * 则此方法返回 v；否则返回 null。（最多只能有一个这样的映射关系。）
 *
 * 返回 null 值并不一定 表明该映射不包含该键的映射关系；
 * 也可能该映射将该键显示地映射为 null。可使用 containsKey 操作来区
分这两种情况。
 *
 * @see #put(Object, Object)
 */
public V get(Object key) {
    Node<K,V> e;
    // 1. 先调用 hash(key)方法计算出 key 的 hash 值
    // 2. 随后调用 getNode 方法获取对应 key 所映射的 value 值
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

/**
 * 获取哈希表结点的方法实现
 *
 * @param hash key 键的 hash 值
 * @param key 键
 * @return 返回对应的结点，如果结点不存在，则返回 null
 */
final Node<K,V> getNode(int hash, Object key) {
    // 用于记录当前的 hash 表
    Node<K,V>[] tab;
    // first 用于记录对应 hash 位置的第一个结点，e 充当工作结点的作用
    Node<K,V> first, e;
    // n 用于记录 hash 表的长度
    int n;
    // 用于临时存放 Key
    K k;
    // 通过 (n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1
    // 判断当前元素的存储位置是否有元素存在
```

```

        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) { //元素存在的情况
            // 如果头结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等
            // 并且 头结点的 key 本身 和要获取的 key 相等
            if (first.hash == hash && // always check first node 总是检
查头结点
                ((k = first.key) == key || (key != null && key.equals(k))))
            // 返回该位置的头结点
            return first;
            if ((e = first.next) != null) { // 头结点不相等
            if (first instanceof TreeNode) // 如果当前结点是树结点
            // 则证明当前位置的链表已变成红黑树结构
            // 通过红黑树结点的方式获取对应 key 结点
            return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do { // 当前位置不是红黑树，则证明是单链表
            // 遍历单链表，逐一比较链表结点
            // 链表结点的 key 的 hash 值 和 要获取的 key 的 hash 值相
等
            // 并且 链表结点的 key 本身 和要获取的 key 相等
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
            // 找到对应的结点则返回
            return e;
            } while ((e = e.next) != null);
            }
        }
        // 通过上述查找均无找到，则返回 null
        return null;
    }
}

```

- 调用 `remove(Object key)` 操作根据键 `key` 删除对应的 `key-value` 键值对时，进行了如下操作：

先调用 `hash(key)` 方法计算出 `key` 的 `hash` 值

根据查找的键值 `key` 的 `hash` 值，通过 `(n - 1) & hash` 当前元素的 `hash` 值 `& hash` 表长度 - 1（实际就是 `hash` 值 `% hash` 表长度）计算出存储位置 `table[i]`，判断存储位置是否有元素存在。

如果存储位置有元素存放，则首先比较头结点元素，如果头结点的 `key` 的 `hash` 值 和 要获取的 `key` 的 `hash` 值相等，并且 头结点的 `key` 本身 和要获取的 `key` 相等，则该位置的头结点即为要删除的结点，记录此结点至变量 `node` 中。

如果存储位置没有元素存放，则没有找到对应要删除的结点，则返回 `null`。

如果存储位置有元素存放，但是头结点元素不是要删除的元素，则需要遍历该位置进行查找。

先判断头结点是否是 `TreeNode`，如果是 `TreeNode` 则证明此位置的结构是红黑树，以红色树的方式遍历查找并删除该结点，没有则返回 `null`。

如果不是红黑树，则证明是单链表。遍历单链表，逐一比较链表结点，链表结点的 `key` 的 `hash` 值和要获取的 `key` 的 `hash` 值相等，并且链表结点的 `key` 本身和要获取的 `key` 相等，则此为要删除的结点，记录此结点至变量 `node` 中，遍历结束仍未找到对应 `key` 的结点，则返回 `null`。

如果找到要删除的结点 `node`，则判断是否需要比较 `value` 是否一致，如果 `value` 值一致或者不需要比较 `value` 值，则执行删除结点操作，删除操作根据不同的情况与结构进行不同的处理。

如果当前结点是树结点，则证明当前位置的链表已变成红黑树结构，通过红黑树结点的方式删除对应结点。

如果不是红黑树，则证明是单链表。如果要删除的是头结点，则当前存储位置 `table[i]` 的头结点指向删除结点的下一个结点。

如果要删除的结点不是头结点，则将要删除的结点的后继结点 `node.next` 赋值给要删除结点的前驱结点的 `next` 域，即 `p.next = node.next;`。

1. `HashMap` 当前存储键值对的数量 - 1，并返回删除结点。

示例代码：

```
/**
 * 从此映射中移除指定键的映射关系（如果存在）。
 *
 * @param key 其映射关系要从映射中移除的键
 * @return 与 key 关联的旧值；如果 key 没有任何映射关系，则返回
null。
 *      （返回 null 还可能表示该映射之前将 null 与 key 关联。）
 */
public V remove(Object key) {
    Node<K, V> e;
    // 1. 先调用 hash(key) 方法计算出 key 的 hash 值
    // 2. 随后调用 removeNode 方法删除对应 key 所映射的结点
```

```

        return (e = removeNode(hash(key), key, null, false, true)) ==
null ?
        null : e.value;
    }

/**
 * 删除哈希表结点的方法实现
 *
 * @param hash 键的 hash 值
 * @param key 键
 * @param value 用于比较的 value 值，当 matchValue 是 true 时才有效，
否则忽略
 * @param matchValue 如果是 true 只有当 value 相等时才会移除
 * @param movable 如果是 false 当执行移除操作时，不删除其他结点
 * @return 返回删除结点 node，不存在则返回 null
 */
final Node<K,V> removeNode(int hash, Object key, Object value,
                           boolean matchValue, boolean movable) {
    // 用于记录当前的 hash 表
    Node<K,V>[] tab;
    // 用于记录当前的链表结点
    Node<K,V> p;
    // n 用于记录 hash 表的长度，index 用于记录当前操作索引 index
    int n, index;
    // 通过 (n - 1) & hash 当前元素的 hash 值 & hash 表长度 - 1
    // 判断当前元素的存储位置是否有元素存在
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (p = tab[index = (n - 1) & hash]) != null) { // 元素存在的情
况
        // node 用于记录找到的结点，e 为工作结点
        Node<K,V> node = null, e;
        K k; V v;
        // 如果头结点的 key 的 hash 值 和 要获取的 key 的 hash 值相等
        // 并且 头结点的 key 本身 和要获取的 key 相等
        // 则证明此头结点就是要删除的结点
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            // 记录要删除的结点的引用地址至 node 中
            node = p;
        else if ((e = p.next) != null) { // 头结点不相等
            if (p instanceof TreeNode) // 如果当前结点是树结点
                // 则证明当前位置的链表已变成红黑树结构
                // 通过红黑树结点的方式获取对应 key 结点

```

```

        // 记录要删除的结点的引用地址至 node 中
        node = ((TreeNode<K, V>)p).getTreeNode(hash, key);
    else { // 当前位置不是红黑树，则证明是单链表
        do {
            // 遍历单链表，逐一比较链表结点
            // 链表结点的 key 的 hash 值 和 要获取的 key 的 hash
            // 值相等

            // 并且 链表结点的 key 本身 和要获取的 key 相等
            if (e.hash == hash &&
                ((k = e.key) == key ||
                 (key != null && key.equals(k)))) {
                // 找到则记录要删除的结点的引用地址至 node 中，
                // 中断遍历

                node = e;
                break;
            }
            p = e;
        } while ((e = e.next) != null);
    }
    // 如果找到要删除的结点，则判断是否需要比较 value 是否一致
    if (node != null && (!matchValue || (v = node.value) == value
        ||
        (value != null && value.equals(v)))) {
        // value 值一致或者不需要比较 value 值，则执行删除结点操作
        if (node instanceof TreeNode) // 如果当前结点是树结点
            // 则证明当前位置的链表已变成红黑树结构
            // 通过红黑树结点的方式删除对应结点
            ((TreeNode<K, V>)node).removeTreeNode(this, tab,
movable);
        else if (node == p) // node 和 p 相等，则证明删除的是头结
        点
            // 当前存储位置的头结点指向删除结点的下一个结点
            tab[index] = node.next;
        else // 删除的不是头结点
            // p 是删除结点 node 的前驱结点，p 的 next 改为记录要删
            除结点 node 的后继结点
            p.next = node.next;
        ++modCount;
        // 当前存储键值对的数量 - 1
        --size;
        afterNodeRemoval(node);
        // 返回删除结点
        return node;
    }

```

```

    }
}
// 不存在要删除的结点，则返回 null
return null;
}

```

- 调用 `replace(K key, V value)` 操作根据键 `key` 查找对应的 `key-value` 键值对，随后替换对应的值 `value`，进行了如下操作：

先调用 `hash(key)` 方法计算出 `key` 的 `hash` 值

随后调用 `getNode` 方法获取对应 `key` 所映射的 `value` 值。

记录元素旧值，将新值赋值给元素，返回元素旧值，如果没有找到元素，则返回 `null`。

示例代码：

```

/**
 * 替换指定 key 所映射的 value 值
 *
 * @param key 对应要替换 value 值元素的 key 键
 * @param value 要替换对应元素的新 value 值
 * @return 返回原本的旧值，如果没有找到 key 对应的元素，则返回 null
 * @since 1.8 JDK1.8 新增方法
 */
public V replace(K key, V value) {
    Node<K, V> e;
    // 1. 先调用 hash(key) 方法计算出 key 的 hash 值
    // 2. 随后调用 getNode 方法获取对应 key 所映射的 value 值
    if ((e = getNode(hash(key), key)) != null) { // 如果找到对应的元
        // 元素旧值
        V oldValue = e.value;
        // 将新值赋值给元素
        e.value = value;
        afterNodeAccess(e);
        // 返回元素旧值
        return oldValue;
    }
    // 没有找到元素，则返回 null
    return null;
}

```

(3.) 问 1: 您上面说，存放一个元素时，先计算它的 `hash` 值确定它的存储位置，然后再

把这个元素放到对应的位置上，那万一这个位置上面已经有元素存在呢，新增的这个元素怎么办？

问 2: hash 冲突（或者叫 hash 碰撞）是什么？为什么会出现这种现象，如何解决 hash 冲突？

答: hash 冲突： 当我们调用 `put(K key, V value)` 操作添加 key-value 键值对，这个 key-value 键值对存放在的位置是通过扰动函数 `(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16)` 计算键 key 的 hash 值。随后将 这个 hash 值 % 模上 哈希表 `Node<K,V>[] table` 的长度 得到具体的存放位置。所以 `put(K key, V value)` 多个元素，是有可能计算出相同的存放位置。此现象就是 hash 冲突或者叫 hash 碰撞。

例子如下：

元素 A 的 hash 值 为 9, 元素 B 的 hash 值 为 17。哈希表 `Node<K,V>[] table` 的长度为 8。则元素 A 的存放位置为  $9 \% 8 = 1$ ，元素 B 的存放位置为  $17 \% 8 = 1$ 。两个元素的存放位置均为 `table[1]`，发生了 hash 冲突。

hash 冲突的避免：既然会发生 hash 冲突，我们就应该想办法避免此现象的发生, 解决这个问题最关键就是如果生成元素的 hash 值。Java 是使用“扰动函数”生成元素的 hash 值。

示例代码：

```
/**
 * JDK 7 的 hash 方法
 */
final int hash(int h) {

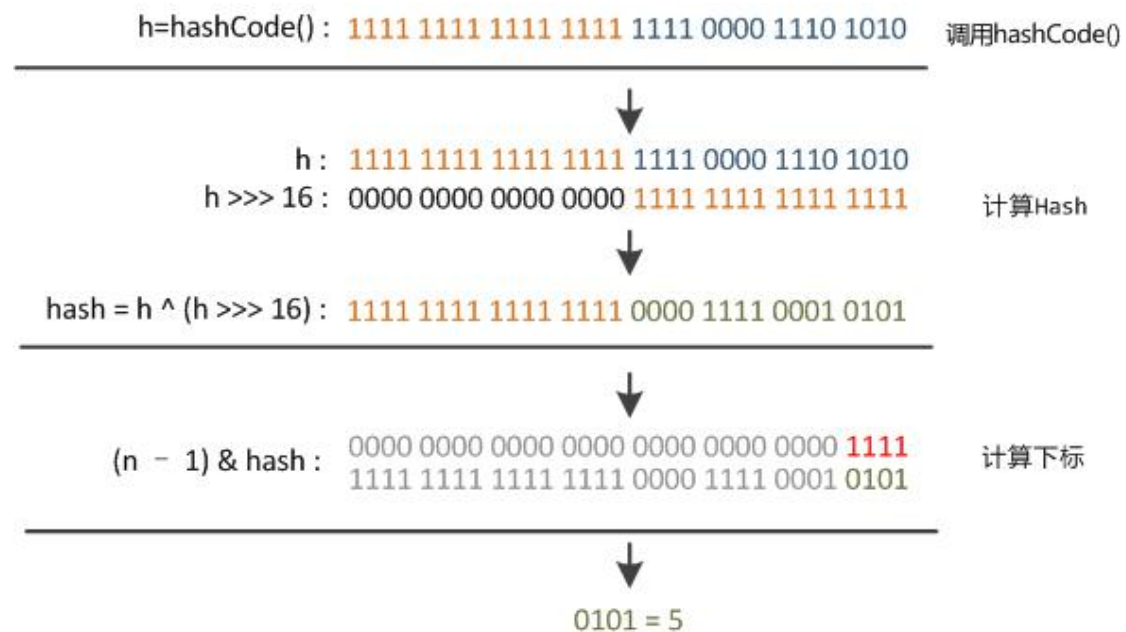
    h ^= k.hashCode();

    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

/**
 * JDK 8 的 hash 方法
 */
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```



Java7 做了 4 次 16 位右位移异或混合，Java 8 中这步已经简化了，只做一次 16 位右位移异或混合，而不是四次，但原理是不变的。例子如下：



扰动函数执行例子 - 图片来自于《知乎》

右位移 16 位，正好是 32bit 的一半，自己的高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下来。

上述扰动函数的解释参考自：[JDK 源码中 HashMap 的 hash 方法原理是什么？](#)

- hash 冲突解决：解决 hash 冲突的方法有很多，常见的有：开发定址法，再散列法，链地址法，公共溢出区法（详细说明请查看我的文章 [JAVA 基础-自问自答学 hashCode 和 equals](#)）。HashMap 是使用链地址法解决 hash 冲突的，当有冲突元素放进来时，会将此元素插入至此位置链表的最后一位，形成单链表。但是由于是单链表的缘故，每当通过  $\text{hash} \% \text{length}$  找到该位置的元素时，均需要从头遍历链表，通过逐一比较 hash 值，找到对应元素。如果此位置元素过多，造成链表过长，遍历时间会大大增加，最坏情况下的时间复杂度为  $O(N)$ ，造成查找效率过低。所以当存在位置的链表长度 大于等于 8 时，HashMap 会将链表 转变为 红黑树，红黑树最坏情况下的时间复杂度为  $O(\log n)$ 。以此提高查找效率。

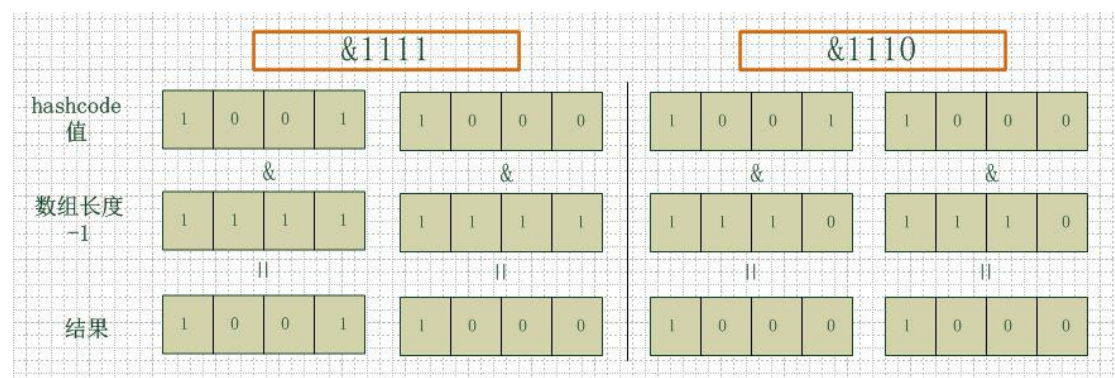
4.问：HashMap 的容量为什么一定要是 2 的 n 次方？

答：因为调用 `put(K key, V value)` 操作添加 key-value 键值对时，具体确定此元素的位置是通过  $\text{hash 值} \% \text{模上 哈希表 Node<K,V>[] table 的长度}$   $\text{hash} \% \text{length}$  计算的。但是"模"运算的消耗相对较大，通过位运算  $h \& (\text{length}-1)$  也

可以得到取模后的存放位置，而位运算的运行效率高，但只有 `length` 的长度是 2 的  $n$  次方时， $h \& (length-1)$  才等价于  $h \% length$ 。

而且当数组长度为 2 的  $n$  次幂的时候，不同的 `key` 算出的 `index` 相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

例子：



hash & (length-1) 运算过程. jpg

上图中，左边两组的数组长度是 16（2 的 4 次方），右边两组的数组长度是 15。两组的 `hash` 值均为 8 和 9。

当数组长度是 15 时，当它们和 1110 进行 & 与运算（相同为 1，不同为 0）时，计算的结果都是 1000，所以它们都会存放在相同的位置 `table[8]` 中，这样就发生了 `hash` 冲突，那么查询时就要遍历链表，逐一比较 `hash` 值，降低了查询的效率。

同时，我们可以发现，当数组长度为 15 的时候，`hash` 值均会与 14 (1110) 进行 & 与运算，那么最后一位永远是 0，而 0001, 0011, 0101, 1001, 1011, 0111, 1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率。

- 所以，`HashMap` 的容量是 2 的  $n$  次方，有利于提高计算元素存放位置时的效率，也降低了 `hash` 冲突的几率。因此，我们使用 `HashMap` 存储大量数据的时候，最好先预先指定容器的大小为 2 的  $n$  次方，即使我们不指定为 2 的  $n$  次方，`HashMap` 也会把容器的大小设置成最接近设置数的 2 的  $n$  次方，如，设置 `HashMap` 的大小为 7，则 `HashMap` 会将容器大小设置成最接近 7 的一个 2 的  $n$  次方数，此值为 8。

上述回答参考自：[深入理解 HashMap](#)

示例代码：

```
/**
 * 返回一个比指定数 cap 大的，并且大小是 2 的 n 次方的数
 * Returns a power of two size for the given target capacity.
 */
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;
}
```

(5.) 问：HashMap 的负载因子是什么，有什么作用？

答：负载因子表示哈希表空间的使用程度（或者说是哈希表空间的利用率）。

例子如下：

底层哈希表 `Node<K,V>[] table` 的容量大小 `capacity` 为 16，负载因子 `load factor` 为 0.75，则当存储的元素个数 `size = capacity 16 * load factor 0.75` 等于 12 时，则会触发 HashMap 的扩容机制，调用 `resize()` 方法进行扩容。

当负载因子越大，则 HashMap 的装载程度就越高。也就是能容纳更多的元素，元素多了，发生 hash 碰撞的几率就会加大，从而链表就会拉长，此时的查询效率就会降低。

当负载因子越小，则链表中的数据量就越稀疏，此时会对空间造成浪费，但是此时查询效率高。

我们可以在创建 HashMap 时根据实际需要适当地调整 `load factor` 的值；如果程序比较关心空间开销、内存比较紧张，可以适当地增加负载因子；如果程序比较关心时间开销，内存比较宽裕则可以适当的减少负载因子。通常情况下，默认负载因子 (0.75) 在时间和空间成本上寻求一种折衷，程序员无需改变负载因子的值。

因此，如果我们在初始化 HashMap 时，就预估知道需要装载 key-value 键值对的容量 `size`，我们可以通过 `size / load factor` 计算出我们需要初始化的容量大小 `initialCapacity`，这样就可以避免 HashMap 因为存放的元素达到阈值 `threshold` 而频繁调用 `resize()` 方法进行扩容。从而保证了较好的性能。

(6.) 问：您能说说 `HashMap` 和 `HashTable` 的区别吗？

答：`HashMap` 和 `HashTable` 有如下区别：

1) 容器整体结构：

`HashMap` 的 `key` 和 `value` 都允许为 `null`，`HashMap` 遇到 `key` 为 `null` 的时候，调用 `putForNullKey` 方法进行处理，而对 `value` 没有处理。

`Hashtable` 的 `key` 和 `value` 都不允许为 `null`。`Hashtable` 遇到 `null`，直接返回 `NullPointerException`。

2) 容量设定与扩容机制：

`HashMap` 默认初始化容量为 16，并且容器容量一定是 2 的  $n$  次方，扩容时，是以原容量 2 倍 的方式进行扩容。

`Hashtable` 默认初始化容量为 11，扩容时，是以原容量 2 倍 再加 1 的方式进行扩容。即 `int newCapacity = (oldCapacity << 1) + 1;`。

3) 散列分布方式（计算存储位置）：

`HashMap` 是先将 `key` 键的 `hashCode` 经过扰动函数扰动后得到 `hash` 值，然后再利用 `hash & (length - 1)` 的方式代替取模，得到元素的存储位置。

`Hashtable` 则是除留余数法进行计算存储位置的（因为其默认容量也不是 2 的  $n$  次方。所以也无法用位运算替代模运算），`int index = (hash & 0x7FFFFFFF) % tab.length;`。

由于 `HashMap` 的容器容量一定是 2 的  $n$  次方，所以能使用 `hash & (length - 1)` 的方式代替取模的方式计算元素的位置提高运算效率，但 `Hashtable` 的容器容量不一定是 2 的  $n$  次方，所以不能使用此运算方式代替。

4) 线程安全（最重要）：

`HashMap` 不是线程安全，如果想线程安全，可以通过调用 `synchronizedMap (Map<K,V> m)` 使其线程安全。但是使用时的运行效率会下降，所以建议使用 `ConcurrentHashMap` 容器以此达到线程安全。

`Hashtable` 则是线程安全的，每个操作方法前都有 `synchronized` 修饰使其同步，但运行效率也不高，所以还是建议使用 `ConcurrentHashMap` 容器以此达到线程安全。

因此，`Hashtable` 是一个遗留容器，如果我们不需要线程同步，则建议使用 `HashMap`，如果需要线程同步，则建议使用 `ConcurrentHashMap`。

此处不再对 `Hashtable` 的源码进行逐一分析了，如果想深入了解的同学，可以参考此文章

### [Hashtable 源码剖析](#)

(7.) 问：您说 `HashMap` 不是线程安全的，那如果多线程下，它是如何处理的？并且什么情况下会发生线程不安全的情况？

答：`HashMap` 不是线程安全的，如果多个线程同时对同一个 `HashMap` 更改数据的话，会导致数据不一致或者数据污染。如果出现线程不安全的操作时，`HashMap` 会尽可能的抛出 `ConcurrentModificationException` 防止数据异常，当我们在对一个 `HashMap` 进行遍历时，在遍历期间，我们是不能对 `HashMap` 进行添加，删除等更改数据的操作的，否则也会抛出 `ConcurrentModificationException` 异常，此为 **fail-fast**(快速失败)机制。从源码上分析，我们在 `put`, `remove` 等更改 `HashMap` 数据时，都会导致 `modCount` 的改变，当 `expectedModCount != modCount` 时，则抛出 `ConcurrentModificationException`。如果想要线程安全，可以考虑使用 `ConcurrentHashMap`。

而且，在多线程下操作 `HashMap`，由于存在扩容机制，当 `HashMap` 调用 `resize()` 进行自动扩容时，可能会导致死循环的发生。

由于时间关系，我暂不带着大家一起去分析 `resize()` 方法导致死循环发生的现象造成原因了，迟点有空我会再补充上去，请见谅，大家可以参考如下文章：

### [Java 8 系列之重新认识 HashMap](#)

#### [谈谈 HashMap 线程不安全的体现](#)

(8.) 问：我们在使用 `HashMap` 时，选取什么对象作为 **key** 键比较好，为什么？

答：可变对象：指创建后自身状态能改变的对象。换句话说，可变对象是该对象在创建后它的哈希值可能被改变。

我们在使用 `HashMap` 时，最好选择不可变对象作为 **key**。例如 `String`，`Integer` 等不可变类型作为 **key** 是非常明智的。

如果 **key** 对象是可变的，那么 **key** 的哈希值就可能改变。在 `HashMap` 中可变对象作为 **Key** 会造成数据丢失。因为我们再进行 `hash & (length - 1)` 取模运算计算位置查找对应元素时，位置可能已经发生改变，导致数据丢失。

详细例子说明请参考：[危险！在 HashMap 中将可变对象用作 Key](#)

总结

HashMap 是基于 Map 接口实现的一种键-值对<key,value>的存储结构，允许 null 值，同时非有序，非同步(即线程不安全)。HashMap 的底层实现是数组 + 链表 + 红黑树（JDK1.8 增加了红黑树部分）。

HashMap 定位元素位置是通过键 key 经过扰动函数扰动后得到 hash 值，然后再通过  $hash \& (length - 1)$  代替取模的方式进行元素定位的。

HashMap 是使用链地址法解决 hash 冲突的，当有冲突元素放进来时，会将此元素插入至此位置链表的最后一位，形成单链表。当存在位置的链表长度 大于等于 8 时，HashMap 会将链表 转变为 红黑树，以此提高查找效率。

HashMap 的容量是 2 的 n 次方，有利于提高计算元素存放位置时的效率，也降低了 hash 冲突的几率。因此，我们使用 HashMap 存储大量数据的时候，最好先预先指定容器的大小为 2 的 n 次方，即使我们不指定为 2 的 n 次方，HashMap 也会把容器的大小设置成最接近设置数的 2 的 n 次方，如，设置 HashMap 的大小为 7，则 HashMap 会将容器大小设置成最接近 7 的一个 2 的 n 次方数，此值为 8。

HashMap 的负载因子表示哈希表空间的使用程度（或者说是哈希表空间的利用率）。当负载因子越大，则 HashMap 的装载程度就越高。也就是能容纳更多的元素，元素多了，发生 hash 碰撞的几率就会加大，从而链表就会拉长，此时的查询效率就会降低。当负载因子越小，则链表中的数据量就越稀疏，此时会对空间造成浪费，但是此时查询效率高。

HashMap 不是线程安全的，Hashtable 则是线程安全的。但 Hashtable 是一个遗留容器，如果我们不需要线程同步，则建议使用 HashMap，如果需要线程同步，则建议使用 ConcurrentHashMap。

在多线程下操作 HashMap，由于存在扩容机制，当 HashMap 调用 `resize()` 进行自动扩容时，可能会导致死循环的发生。

我们在使用 HashMap 时，最好选择不可变对象作为 key。例如 String，Integer 等不可变类型作为 key 是非常明智的。

## 2. ArrayList

### 定义

快速了解 ArrayList 究竟是什么的一个好方法就是看 JDK 源码中对 ArrayList 类的注释，大致翻译如下：

/\*\*

- \* 实现了 List 的接口的可调整大小的数组。实现了所有可选列表操作，并且允许所有类型的元素，

- \* 包括 null。除了实现了 List 接口，这个类还提供了去动态改变内部用于存储集合元素的数组尺寸

- \* 的方法。（这个类与 Vector 类大致相同，除了 ArrayList 是非线程安全外。）size, isEmpty,

- \* get, set, iterator, 和 listIterator 方法均为常数时间复杂度。add 方法的摊还时间复杂度为

- \* 常数级别，这意味着，添加 n 个元素需要的时间为  $O(n)$ 。所有其他方法的时间复杂度都是线性级别的。

- \* 常数因子要比 LinkedList 低。

- \* 每个 ArrayList 实例都有一个 capacity。capacity 是用于存储 ArrayList 的元素的内部数组的大小。

- \* 它通常至少和 ArrayList 的大小一样大。当元素被添加到 ArrayList 时，它的 capacity 会自动增长。

- \* 在向一个 ArrayList 中添加大量元素前，可以使用 ensureCapacity 方法来增加 ArrayList 的容量。

- \* 使用这个方法来一次性地使 ArrayList 内部数组的尺寸增长到我们需要的大小提升性能。需要注意的

- \* 是，这个 ArrayList 实现是未经同步的。若在多线程环境下并发访问一个 ArrayList 实例，并且至少

- \* 一个线程对其作了结构型修改，那么必须在外部做同步。（结构性修改指的是任何添加或删除了一个或

- \* 多个元素的操作，以及显式改变内部数组尺寸的操作。set 操作不是结构性修改）在外部做同步通常通

- \* 过在一些自然地封装了 ArrayList 的对象上做同步来实现。如果不存在这样的对象，ArrayList 应

- \* 使用 Collections.synchronizedList 方法来包装。最好在创建时就这么做，以防止对 ArrayList

- \* 无意的未同步访问。（List list = Collections.synchronizedList(new ArrayList(...));）

- \* ArrayList 类的 iterator() 方法以及 listIterator() 方法返回的迭代器是 fail-fast 的：

- \* 在 iterator 被创建后的任何时候，若对 list 进行了结构性修改（以任何除了通过迭代器自己的

- \* remove 方法或 add 方法的方式），迭代器会抛出一个 ConcurrentModificationException 异常。

- \* 因此，在遇到并发修改时，迭代器马上抛出异常，而不是冒着以后可能在不确定的时间发生不确定行为

- \* 的风险继续。需要注意的是，迭代器的 fail-fast 行为是不能得到保证的，因为通常来说在未同步并发

- \* 修改面前无法做任何保证。fail-fast 迭代器会尽力抛出 ConcurrentModificationException 异常。

\* 因此，编写正确性依赖于这个异常的程序是不对的：fail-fast 行为应该仅仅在检测 bugs 时被使用。

\* ArrayList 类是 Java 集合框架中的一员。  
\*/

根据源码中的注释，我们了解了 **ArrayList** 用来组织一系列同类型的数据对象，支持对数据对象的顺序迭代与随机访问。我们还了解了 **ArrayList** 所支持的操作以及各项操作的时间复杂度。接下来我们来看看这个类实现了哪些接口。

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

我们可以看到，它实现了 4 个接口：**List**、**RandomAccess**、**Cloneable**、**Serializable**。官方文档对 **List** 接口的说明如下：**List** 是一个有序的集合类型（也被称作序列）。使用 **List** 接口可以精确控制每个元素被插入的位置，并且可以通过元素在列表中的索引来访问它。列表允许重复的元素，并且在允许 **null** 元素的情况下也允许多个 **null** 元素。

**List** 接口定义了以下方法：

```
ListIterator<E> listIterator(); void add(int i, E element); E remove(int i); E get(int i); E set(int i, E element); int indexOf(Object element);
```

我们可以看到，**add**、**get** 等方法都是我们在使用 **ArrayList** 时经常用到的。在 **ArrayList** 的源码注释中提到了，**ArrayList** 使用 **Object** 数组来存储集合元素。我们来一起看下它的源码中定义的如下几个字段：

```
/** * 默认初始 capacity. */
private static final int DEFAULT_CAPACITY = 10; /** * 供空的 ArrayList 实例使用的空的数组实例 */
private static final Object[] EMPTY_ELEMENTDATA = {}; /** * 供默认大小的空的 ArrayList 实例使用的空的数组实例。
    * 我们把它和 EMPTY_ELEMENTDATA 区分开来，一边指导当地一个元素被添加时把内部数组尺寸设为
    * 多少
    */
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {}; /**
    * 存放 ArrayList 中的元素的内部数组。
    * ArrayList 的 capacity 就是这个内部数组的大小。
    * 任何 elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA 的空 ArrayList 在第一个元素
    * 被添加进来时，其 capacity 都会被扩大至 DEFAULT_CAPACITY
    */
transient Object[] elementData; // non-private to simplify nested class access /** * ArrayList 所包含的元素数 */
private int size;
```



通过以上字段，我们验证了 `ArrayList` 内部确实使用一个 `Object` 数组来存储集合元素。

那么接下来我们看一下 `ArrayList` 都有哪些构造器，从而了解 `ArrayList` 的构造过程。

## ArrayList 的构造器

首先我们来看一下我们平时经常使用的 `ArrayList` 的无参构造器的源码：

```
/** * Constructs an empty list with an initial capacity of ten. */public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;}
```

我们可以看到，无参构造器仅仅是把 `ArrayList` 实例的 `elementData` 字段赋值为 `DEFAULTCAPACITY_EMPTY_ELEMENTDATA`。

接下来，我们再来看一下 `ArrayList` 的其他构造器：

```
/** * Constructs an empty list with the specified initial capacity.
 * * @param initialCapacity the initial capacity of the list
 * * @throws IllegalArgumentException if the specified initial
 * capacity
 * is negative
 */
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
            initialCapacity);
    }
}

/** * Constructs a list containing the elements of the specified
 * collection, in the order they are returned by the collection's *
 * iterator.
 * * @param c the collection whose elements are to be placed into this
 * list
 * * @throws NullPointerException if the specified collection is null
 */
public ArrayList(Collection<? extends E> c) {
    elementData = c.toArray();
    if ((size = elementData.length) != 0) {
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
```

```

        elementData = Arrays.copyOf(elementData, size, Object[].class);
    } else {
        // replace with empty array.
        this.elementData = EMPTY_ELEMENTDATA;
    }
}

```

通过源码我们可以看到，第一个构造器指定了 `ArrayList` 的初始 `capacity`，然后根据这个初始 `capacity` 创建一个相应大小的 `Object` 数组。若 `initialCapacity` 为 0，则将 `elementData` 赋值为 `EMPTY_ELEMENTDATA`；若 `initialCapacity` 为负数，则抛出一个 `IllegalArgumentException` 异常。

第二个构造器则指定一个 `Collection` 对象作为参数，从而构造一个含有指定集合对象元素的 `ArrayList` 对象。这个构造器首先把 `elementData` 实例域赋值为集合对象转为的数组，而后再判断传入的集合对象是否不含有任何元素，若是的话，则将 `elementData` 赋值为 `EMPTY_ELEMENTDATA`；若传入的集合对象至少包含一个元素，则进一步判断 `c.toArray` 方法是否正确返回了 `Object` 数组，若不是的话，则需要用 `Arrays.copyOf` 方法把 `elementData` 的元素类型改变为 `Object`。

现在，我们又了解了 `ArrayList` 实例的构建过程，那么接下来我们来通过 `ArrayList` 的 `get`、`set` 等方法的源码来进一步了解它的实现原理。

## add 方法源码分析

```

/** * Appends the specified element to the end of this list.
 * * @param e element to be appended to this list
 * * @return <tt>true</tt> (as specified by {@link Collection#add})
 */public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;}

```

我们可以看到，在 `add` 方法内部，首先调用了 `ensureCapacityInternal(size+1)`，这句的作用有两个：

- 保证当前 `ArrayList` 实例的 `capacity` 足够大；
- 增加 `modCount`，`modCount` 的作用是判断在迭代时是否对 `ArrayList` 进行了结构性修改。

然后通过将内部数组下一个索引处的元素设置为给定参数来完成了向 `ArrayList` 中添加元素，返回 `true` 表示添加成功。

## get 方法源码分析

```

/** * Returns the element at the specified position in this list.
    * * @param index index of the element to return
    * @return the element at the specified position in this list
    * @throws IndexOutOfBoundsException {@inheritDoc}
*/public E get(int index) {
    rangeCheck(index);
    return elementData(index);}

```

首先调用了 `rangeCheck` 方法来检查我们传入的 `index` 是否在合法范围内，然后调用了 `elementData` 方法，这个方法的源码如下：

```

E elementData(int index) {
    return (E) elementData[index];}

```

## set 方法源码分析

```

/** * Replaces the element at the specified position in this list with
    * the specified element.
    * * @param index index of the element to replace
    * @param element element to be stored at the specified position
    * @return the element previously at the specified position
    * @throws IndexOutOfBoundsException {@inheritDoc}
*/
public E set(int index, E element) {
    rangeCheck(index);
    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;}

```

我们可以看到，`set` 方法的实现也很简单，首先检查给定的索引是否在合法范围内，若在，则先把该索引处原来的元素存储在 `oldValue` 中，然后把新元素放到该索引处并返回 `oldValue` 即可。

# 3. LinkedList

## 定义

`LinkedList` 类源码中的注释如下：

```

/** * 实现了 List 接口的双向链表。实现了所有可选列表操作，并且可以存储
    所有类型的元素，包括 null。

```

- \* 对 LinkedList 指定索引处的访问需要顺序遍历整个链表，直到到达指定元素。
- \* 注意 LinkedList 是非同步的。若多线程并发访问 LinkedList 对象，并且至少一个线程对其做
- \* 结构性修改，则必须在外面对它进行同步。这通常通过在一些自然封装了 LinkedList 的对象上
- \* 同步来实现。若不存在这样的对象，这个 list 应使用 Collections.synchronizedList 来包装。
- \* 这最好在创建时完成，以避免意外的非同步访问。
- \* LinkedList 类的 iterator() 方法以及 listIterator() 方法返回的迭代器是 fail-fast 的：
- \* 在 iterator 被创建后的任何时候，若对 list 进行了结构性修改（以任何除了通过迭代器自己的
- \* remove 方法或 add 方法的方式），迭代器会抛出一个 ConcurrentModificationException 异常。
- \* 因此，在遇到并发修改时，迭代器马上抛出异常，而不是冒着以后可能在不确定的时间发生不确定行为
- \* 的风险继续。需要注意的是，迭代器的 fail-fast 行为是不能得到保证的，因为通常来说在未同步并发
- \* 修改面前无法做任何保证。fail-fast 迭代器会尽力抛出 ConcurrentModificationException 异常。
- \* 因此，编写正确性依赖于这个异常的程序是不对的：fail-fast 行为应该仅仅在检测 bugs 时被使用。
- \* LinkedList 类是 Java 集合框架中的一员。
- \*/

LinkedList 是对链表这种数据结构的实现（对链表还不太熟悉的小伙伴可以参考[深入理解数据结构之链表](#)），当我们需要一种支持高效删除/添加元素的数据结构时，可以考虑使用链表。

总的来说，链表具有以下两个优点：

- 插入及删除操作的时间复杂度为  $O(1)$
- 可以动态改变大小

链表主要的缺点是：由于其链式存储的特性，链表不具备良好的空间局部性，也就是说，链表是一种缓存不友好的数据结构。

## 支持的操作

LinkedList 主要支持以下操作：

```
void addFirst(E element); void addLast(E element); E getFirst(); E
getLast(); E removeFirst(); E removeLast(); boolean add(E e) //把元素 e
添加到链表末尾 void add(int index, E element) //在指定索引处添加元素
```

以上操作除了 `add(int index, E element)` 外, 时间复杂度均为  $O(1)$ , 而 `add(int index, E element)` 的时间复杂度为  $O(N)$ 。

## Node 类

在 `LinkedList` 类中我们能看到以下几个字段:

```
transient int size = 0; /** * 指向头结点 */ transient Node<E> first; /**
 * 指向尾结点 */ transient Node<E> last;
```

我们看到, `LinkedList` 只保存了头尾节点的引用作为其实例域, 接下来我们看一下 `LinkedList` 的内部类 `Node` 的源码如下:

```
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;
    }
}
```

每个 `Node` 对象的 `next` 域指向它的下一个结点, `prev` 域指向它的上一个结点, `item` 为本结点所存储的数据对象。

## addFirst 源码分析

```
/** * Inserts the specified element at the beginning of this list.
 * * @param e the element to add
 */ public void addFirst(E e) {
    linkFirst(e);
}
```

实际干活的是 `linkFirst`, 它的源码如下:

```
/** * Links e as first element. */ private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
}
```

```

size++;
modCount++;}

```

首先把头结点引用存于变量 **f** 中，而后创建一个新结点，这个新结点的数据为我们传入的参数 **e**，**prev** 指针为 **null**，**next** 指针为 **f**。然后把头结点指针指向新创建的结点 **newNode**。而后判断 **f** 是否为 **null**，若为 **null**，说明之前链表中没有结点，所以 **last** 也指向 **newNode**；若 **f** 不为 **null**，则把 **f** 的 **prev** 指针设为 **newNode**。最后还需要把 **size** 和 **modCount** 都加一，**modCount** 的作用与在 **ArrayList** 中的相同。

## getFirst 方法源码分析

```

/** * Returns the first element in this list.
    * * @return the first element in this list
    * @throws NoSuchElementException if this list is empty
*/public E getFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return f.item;}

```

这个方法的实现很简单，主需要直接返回 **first** 的 **item** 域（当 **first** 不为 **null** 时），若 **first** 为 **null**，则抛出 **NoSuchElementException** 异常。

## removeFirst 方法源码分析

```

/** * Removes and returns the first element from this list.
    * * @return the first element from this list
    * @throws NoSuchElementException if this list is empty
*/public E removeFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return unlinkFirst(f);}

```

**unlinkFirst** 方法的源码如下：

```

/** * Unlinks non-null first node f. */private E unlinkFirst(Node<E> f)
{
    // assert f == first && f != null;
    final E element = f.item;
    final Node<E> next = f.next;
    f.item = null;

```

```

f.next = null; // help GC
first = next;
if (next == null)
    last = null;
else
    next.prev = null;
size--;
modCount++;
return element;}

```

## add(int index, E e)方法源码分析

```

/** * Inserts the specified element at the specified position in this list.
 * Shifts the element currently at that position (if any) and any
 * subsequent elements to the right (adds one to their indices).
 * * @param index index at which the specified element is to be
inserted
 * @param element element to be inserted
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public void add(int index, E element) {
    checkPositionIndex(index);
    if (index == size)
        linkLast(element);
    else
        linkBefore(element, node(index));}

```

这个方法中，首先调用 `checkPositionIndex` 方法检查给定 `index` 是否在合法范围内。然后若 `index` 等于 `size`，这说明要在链表尾插入元素，直接调用 `linkLast` 方法，这个方法的实现与之前介绍的 `linkFirst` 类似；若 `index` 小于 `size`，则调用 `linkBefore` 方法，在 `index` 处的 `Node` 前插入一个新 `Node`（`node(index)`会返回 `index` 处的 `Node`）。`linkBefore` 方法的源码如下：

```

/** * Inserts element e before non-null Node succ. */void linkBefore(E
e, Node<E> succ) {
    // assert succ != null;
    final Node<E> pred = succ.prev;
    final Node<E> newNode = new Node<>(pred, e, succ);
    succ.prev = newNode;
    if (pred == null)
        first = newNode;
    else
        pred.next = newNode;
    size++;
}

```

```
modCount++;}
```

我们可以看到，在知道要在哪个结点前插入一个新结点时，插入操作是很容易的，时间复杂度也只有  $O(1)$ 。下面我们来看一下 `node` 方法是如何获取指定索引处的 `Node` 的：

```
/** * Returns the (non-null) Node at the specified element index. */
Node<E> node(int index) {
    // assert isElementIndex(index);
    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

首先判断 `index` 位于链表的前半部分还是后半部分，若是前半部分，则从头结点开始遍历，否则从尾结点开始遍历，这样可以提升效率。我们可以看到，这个方法的时间复杂度为  $O(N)$ 。

`HashSet` 是 `Set` 的一种实现方式，底层主要使用 `HashMap` 来确保元素不重复。

## 4. Hashset 源码分析

### 属性

```
// 内部使用 HashMap
```

```
private transient HashMap<E, Object> map;
```

```
// 虚拟对象，用来作为 value 放到 map 中
```

```
private static final Object PRESENT = new Object();
```

### 构造方法



```

public HashSet() {

    map = new HashMap<>();

}

public HashSet(Collection<? extends E> c) {

    map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));

    addAll(c);

}

public HashSet(int initialCapacity, float loadFactor) {

    map = new HashMap<>(initialCapacity, loadFactor);

}

public HashSet(int initialCapacity) {

    map = new HashMap<>(initialCapacity);

}

// 非 public, 主要是给 LinkedHashSet 使用的

HashSet(int initialCapacity, float loadFactor, boolean dummy) {

    map = new LinkedHashMap<>(initialCapacity, loadFactor);

}

```

构造方法都是调用 `HashMap` 对应的构造方法。

最后一个构造方法有点特殊，它不是 `public` 的，意味着它只能被同一个包或者子类调用，这是 `LinkedHashSet` 专属的方法。

## 添加元素

直接调用 `HashMap` 的 `put()` 方法，把元素本身作为 `key`，把 `PRESENT` 作为 `value`，也就是这个 `map` 中所有的 `value` 都是一样的。

```
public boolean add(E e) {  
  
    return map.put(e, PRESENT) != null;  
  
}
```

## 删除元素

直接调用 `HashMap` 的 `remove()` 方法，注意 `map` 的 `remove` 返回是删除元素的 `value`，而 `Set` 的 `remove` 返回的是 `boolean` 类型。

这里要检查一下，如果是 `null` 的话说明没有该元素，如果不是 `null` 肯定等于 `PRESENT`。

```
public boolean remove(Object o) {  
  
    return map.remove(o) == PRESENT;  
  
}
```

## 查询元素

`Set` 没有 `get()` 方法哦，因为 `get` 似乎没有意义，不像 `List` 那样可以按 `index` 获取元素。

这里只要一个检查元素是否存在的方法 `contains()`，直接调用 `map` 的 `containsKey()` 方法。

```
public boolean contains(Object o) {  
  
    return map.containsKey(o);  
  
}
```

## 遍历元素

直接调用 map 的 keySet 的迭代器。

```
public Iterator<E> iterator() {  
  
    return map.keySet().iterator();  
}
```

## 全部源码

```
package java.util;  
  
import java.io.InvalidObjectException;  
  
import sun.misc.SharedSecrets;  
  
public class HashSet<E>  
  
    extends AbstractSet<E>  
  
    implements Set<E>, Cloneable, java.io.Serializable  
  
{  
  
    static final long serialVersionUID = -5024744406713321676L;  
  
    // 内部元素存储在 HashMap 中  
  
    private transient HashMap<E, Object> map;  
  
    // 虚拟元素，用来存到 map 元素的 value 中的，没有实际意义  
  
    private static final Object PRESENT = new Object();  
  
    // 空构造方法  
  
    public HashSet() {  
  
        map = new HashMap<>();  
  
    }
```

```
// 把另一个集合的元素全都添加到当前 Set 中
```

```
// 注意，这里初始化 map 的时候是计算了它的初始容量的
```

```
public HashSet(Collection<? extends E> c) {
```

```
    map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
```

```
    addAll(c);
```

```
}
```

```
// 指定初始容量和装载因子
```

```
public HashSet(int initialCapacity, float loadFactor) {
```

```
    map = new HashMap<>(initialCapacity, loadFactor);
```

```
}
```

```
// 只指定初始容量
```

```
public HashSet(int initialCapacity) {
```

```
    map = new HashMap<>(initialCapacity);
```

```
}
```

```
// LinkedHashMap 专用的方法
```

```
// dummy 是没有实际意义的，只是为了跟上上面那个操作持方法签名不同而已
```

```
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
```

```
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
```

```
}
```

```
// 迭代器
```

```
public Iterator<E> iterator() {
```

```
    return map.keySet().iterator();
```

```
}
```

```
// 元素个数
```

```
public int size() {
```

```
    return map.size();
```

```
}
```

```
// 检查是否为空
```

```
public boolean isEmpty() {
```

```
    return map.isEmpty();
```

```
}
```

```
// 检查是否包含某个元素
```

```
public boolean contains(Object o) {
```

```
    return map.containsKey(o);
```

```
}
```

```
// 添加元素
```

```
public boolean add(E e) {
```

```
    return map.put(e, PRESENT)!=null;
```

```
}
```

```
// 删除元素
```

```
public boolean remove(Object o) {
```

```
·
```

```
·
```

```
    return map.remove(o)==PRESENT;
```

```
·
```

```
·
```

```
}
```

```
.
.

.
.
    // 清空所有元素
.
.
    public void clear() {
.
.
        map.clear();
.
.
    }
.
.

.
.
    // 克隆方法
.
.
    @SuppressWarnings("unchecked")
.
.
    public Object clone() {
.
.
        try {
.
.
            HashSet<E> newSet = (HashSet<E>) super.clone();
.
.
            newSet.map = (HashMap<E, Object>) map.clone();
.
.
            return newSet;
.
.
        } catch (CloneNotSupportedException e) {
.
.

```

```

        throw new InternalError(e);
    }
}

// 序列化写出方法

private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // 写出非 static 非 transient 属性

    s.defaultWriteObject();

    // 写出 map 的容量和装载因子

    s.writeInt(map.capacity());

    s.writeFloat(map.loadFactor());

    // 写出元素个数

```

```
.
    s.writeInt(map.size());
.
.
.
.
.
    // 遍历写出所有元素
.
.
    for (E e : map.keySet())
.
.
        s.writeObject(e);
.
.
    }
.
.
.
.
.
    // 序列化读入方法
.
.
    private void readObject(java.io.ObjectInputStream s)
.
.
        throws java.io.IOException, ClassNotFoundException {
.
.
        // 读入非 static 非 transient 属性
.
.
        s.defaultReadObject();
.
.
.
.
        // 读入容量，并检查不能小于 0
.
.
        int capacity = s.readInt();
```



```
.
.
    if (capacity < 0) {
.
.
        throw new InvalidObjectException("Illegal capacity: " +
.
.
            capacity);
.
.
    }
.
.
.
.
    // 读入装载因子，并检查不能小于等于 0 或者是 NaN(Not a Number)
.
.
    // java.lang.Float.NaN = 0.0f / 0.0f;
.
.
    float loadFactor = s.readFloat();
.
.
    if (loadFactor <= 0 || Float.isNaN(loadFactor)) {
.
.
        throw new InvalidObjectException("Illegal load factor: " +
.
.
            loadFactor);
.
.
    }
.
.
.
.
    // 读入元素个数并检查不能小于 0
.
.
.
```

```
        int size = s.readInt();
    .
    .
    .
        if (size < 0) {
    .
    .
        throw new InvalidObjectException("Illegal size: " +
    .
    .
        size);
    .
    .
    .
    }

    .
    .
    .
    // 根据元素个数重新设置容量
    .
    .
    .
    // 这是为了保证 map 有足够的容量容纳所有元素，防止无意义的扩容
    .
    .
    .
    capacity = (int) Math.min(size * Math.min(1 / loadFactor, 4.0f),
    .
    .
    .
    HashMap.MAXIMUM_CAPACITY);
    .
    .
    .
    .
    .
    // 再次检查某些东西，不重要的代码忽视掉
    .
    .
    .
    SharedSecrets.getJavaOISAccess()
    .
    .
    .
    .checkArray(s, Map.Entry[].class,
    HashMap.tableSizeFor(capacity));
    .
    .
    .
    .
    .
    // 创建 map，检查是不是 LinkedHashMap 类型
```

```

.
.
.
    map = (((HashSet<?>)this) instanceof LinkedHashSet ?
.
.
        new LinkedHashMap<E, Object>(capacity, loadFactor) :
.
.
        new HashMap<E, Object>(capacity, loadFactor));
.
.
.
.
.
    // 读入所有元素，并放入 map 中
.
.
    for (int i=0; i<size; i++) {
.
.
        @SuppressWarnings("unchecked")
.
.
        E e = (E) s.readObject();
.
.
        map.put(e, PRESENT);
.
.
    }
.
.
    }
.
.
    // 可分割的迭代器，主要用于多线程并行迭代处理时使用
.
    public Spliterator<E> spliterator() {
.
        return new HashMap.KeySpliterator<E, Object>(map, 0, -1, 0, 0);
.
    }
.
}

```

## 总结

- (1) HashSet 内部使用 HashMap 的 key 存储元素，以此来保证元素不重复；
- (2) HashSet 是无序的，因为 HashMap 的 key 是无序的；
- (3) HashSet 中允许有一个 null 元素，因为 HashMap 允许 key 为 null；
- (4) HashSet 是非线程安全的；
- (5) HashSet 是没有 get()方法的；

## 5. 内存模型

### 内存模型产生背景

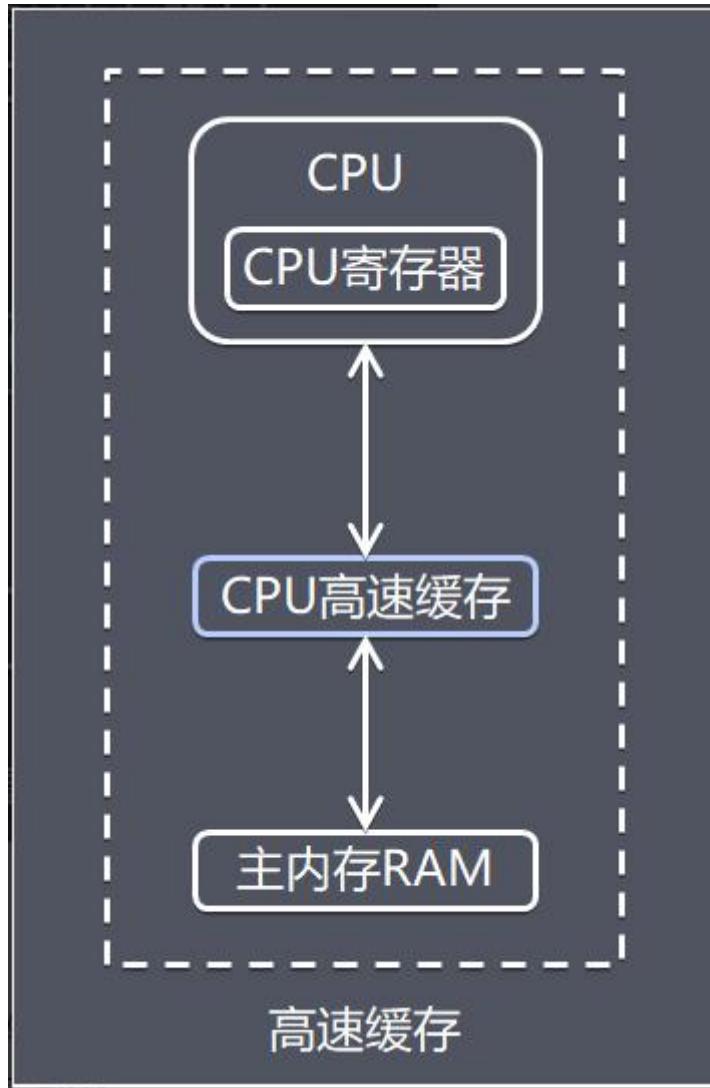
在介绍 Java 内存模型之前，我们先了解一下物理计算机中的并发问题，理解这些问题可以搞清楚内存模型产生的背景。物理机遇到的并发问题与虚拟机中的情况有不少相似之处，物理机的解决方案对虚拟机的实现有相当的参考意义。

### 物理机的并发问题

- 硬件的效率问题

计算机处理器处理绝大多数运行任务都不可能只靠处理器“计算”就能完成，处理器至少需要与**内存交互**，如读取运算数据、存储运算结果，这个 I/O 操作很难消除(无法仅靠寄存器完成所有运算任务)。

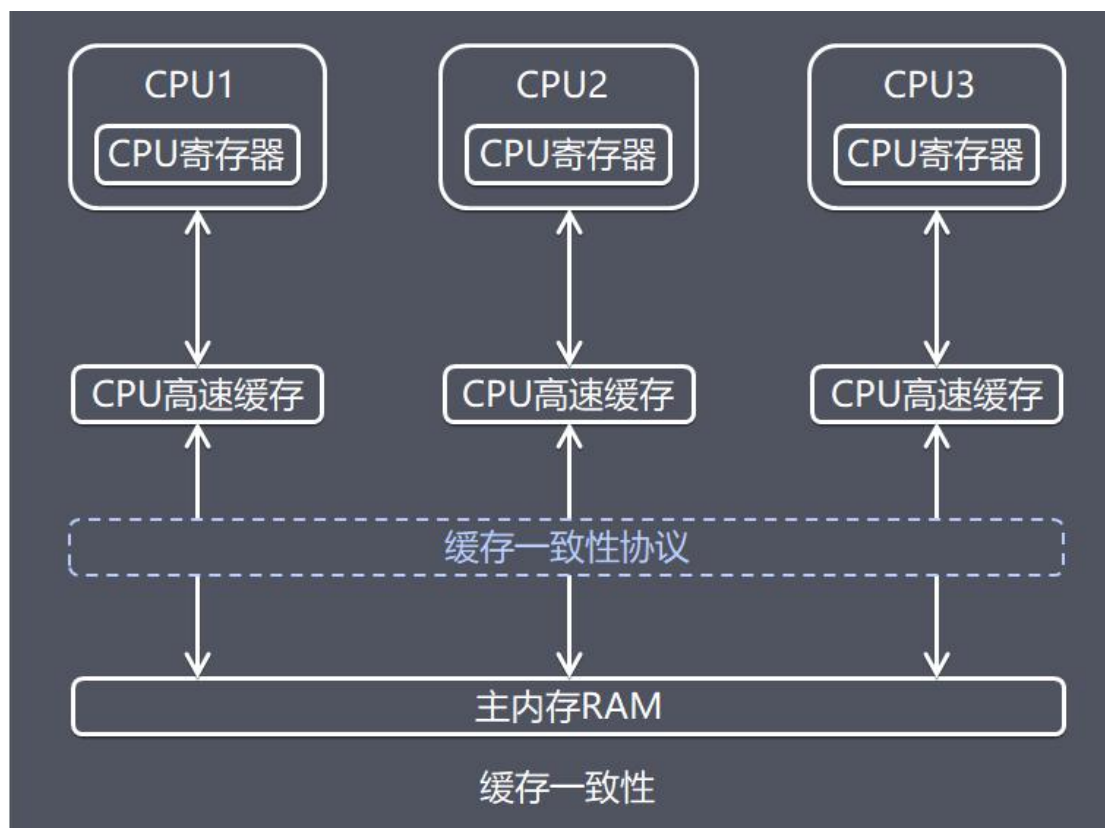
由于计算机的存储设备与处理器的运算速度有几个数量级的差距，为了避免处理器等待缓慢的内存读写操作完成，现代计算机系统通过加入一层读写速度尽可能接近处理器运算速度的高速缓存。缓存作为内存和处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速运行，当运算结束后再从缓存同步回内存之中。



- 缓存一致性问题

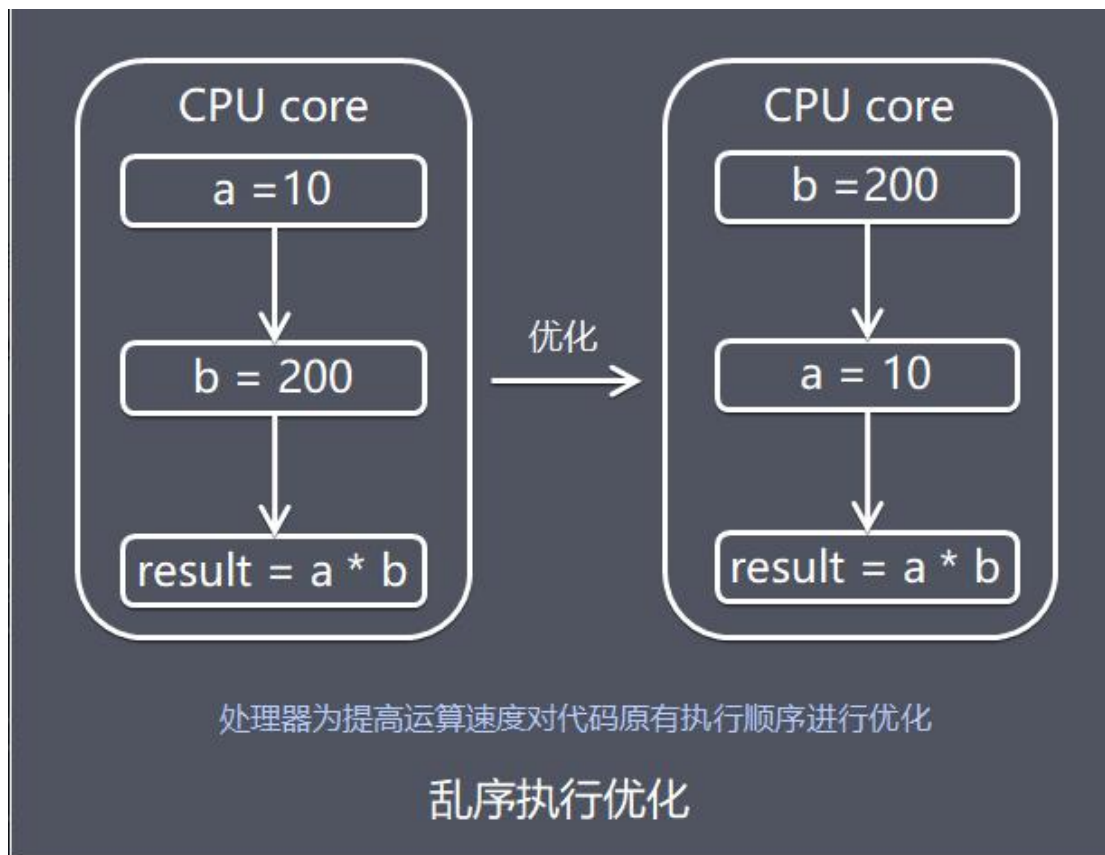
基于高速缓存的存储系统交互很好地解决了处理器与内存速度的矛盾，但是也为计算机系统带来更高的复杂度，因为引入了一个新问题：**缓存一致性**。

在多个处理器的系统中(或者单处理器多核的系统)，每个处理器(每个核)都有自己的高速缓存，而它们有共享同一主内存(Main Memory)。当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致。为此，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议进行操作，来维护缓存的一致性。



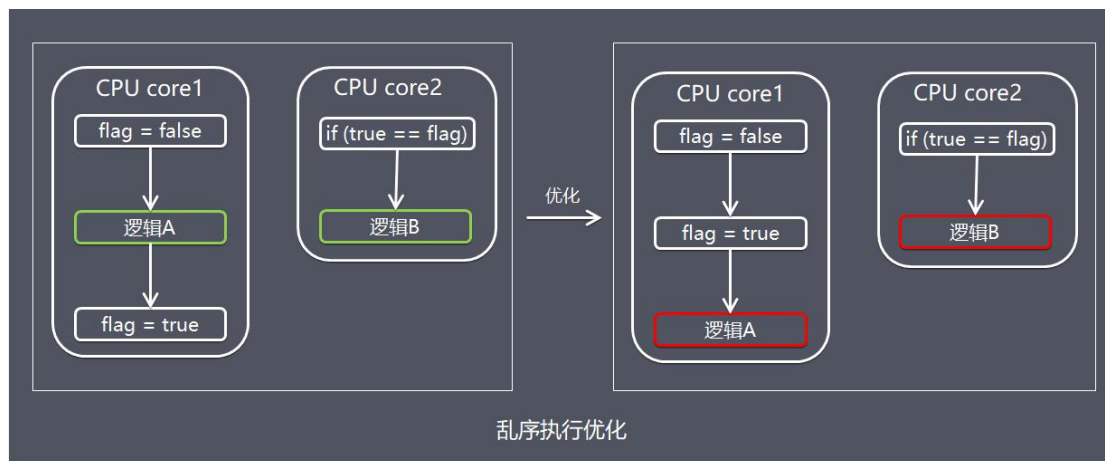
- 代码乱序执行优化问题

为了使得处理器内部的运算单元尽量被充分利用，提高运算效率，处理器可能会对输入的代码进行乱序执行，处理器会在计算之后将乱序执行的结果重组，**乱序优化可以保证在单线程下该执行结果与顺序执行的结果是一致的**，但不保证程序中各个语句计算的先后顺序与输入代码中的顺序一致。



乱序执行技术是处理器为提高运算速度而做出违背代码原有顺序的优化。在单核时代，处理器保证做出的优化不会导致执行结果远离预期目标，但在多核环境下却并非如此。

多核环境下， 如果存在一个核的计算任务依赖另一个核 计的算任务的中间结果，而且对相关数据读写没做任何防护措施，那么其顺序性并不能靠代码的先后顺序来保证，处理器最终得出的结果和我们逻辑得到的结果可能会大不相同。



以上图为例进行说明：CPU 的 core2 中的逻辑 B 依赖 core1 中的逻辑 A 先执行

- 正常情况下，逻辑 A 执行完之后再执行逻辑 B。
- 在处理器乱序执行优化情况下，有可能导致 flag 提前被设置为 true，导致逻辑 B 先于逻辑 A 执行。

## 2 Java 内存模型的组成分析

### 内存模型概念

为了解决上面提到的系列问题，内存模型被总结提出，我们可以把内存模型理解为在特定操作协议下，对特定的内存或高速缓存进行读写访问的过程抽象。

不同架构的物理计算机可以有不一样的内存模型，Java 虚拟机也有自己的内存模型。Java 虚拟机规范中试图定义一种 Java 内存模型（Java Memory Model，简称 JMM）来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果，不必因为不同平台上的物理机的内存模型的差异，对各平台定制化开发程序。

更具体一点说，Java 内存模型提出目标在于，定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量(Variables)与 Java 编程中所说的变量有所区别，它包括了实例字段、静态字段和构成数值对象的元素，但不包括局部变量与方法参数，因为后者是线程私有的。(如果局部变量是一个 reference 类型，它引用的对象在 Java 堆中可被各个线程共享，但是 reference 本身在 Java 栈的局部变量表中，它是线程私有的)。

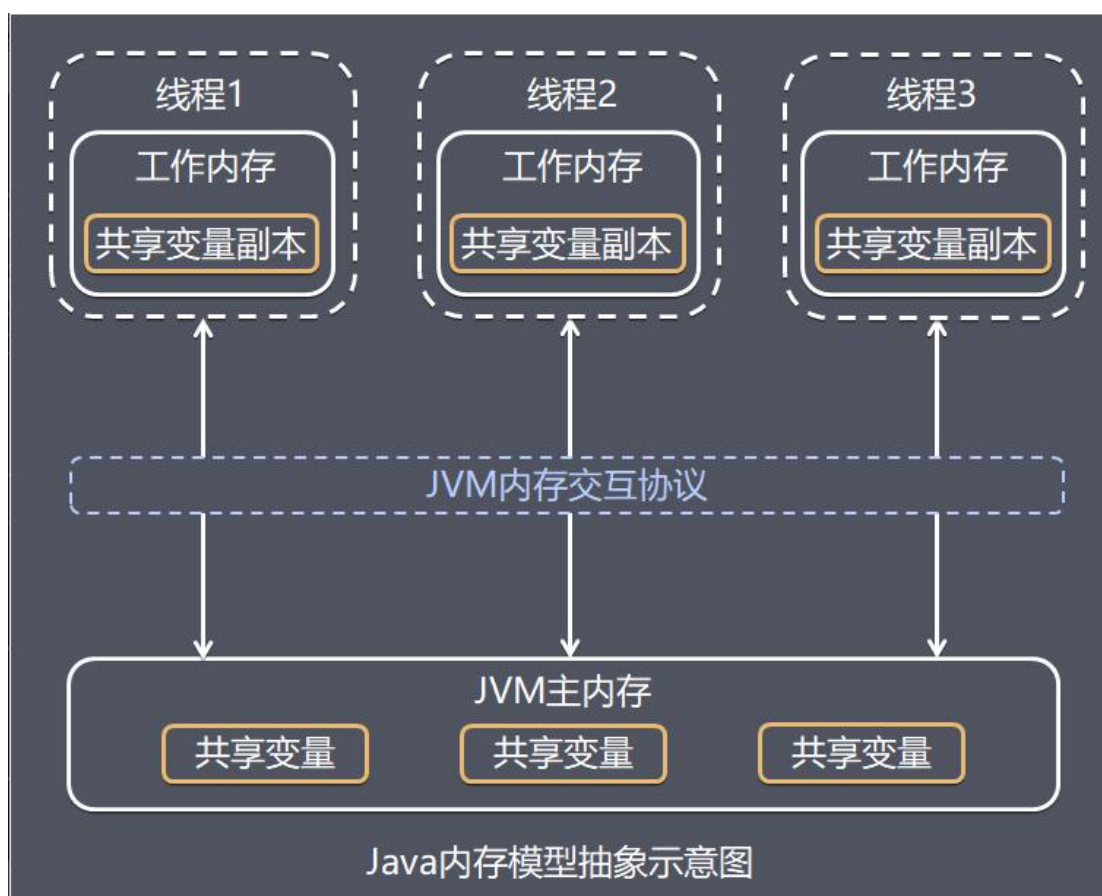
## Java 内存模型的组成



主内存 Java 内存模型规定了所有变量都存储在主内存(Main Memory)中（此处的主内存与介绍物理硬件的主内存名字一样，两者可以互相类比，但此处仅是虚拟机内存的一部分）。

工作内存 每条线程都有自己的工作内存(Working Memory，又称本地内存，可与前面介绍的处理器高速缓存类比)，线程的工作内存中保存了该线程使用到的变量的主内存中的共享变量的副本拷贝。工作内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他硬件和编译器优化。

Java 内存模型抽象示意图如下



## JVM 内存操作的并发问题

结合前面介绍的物理机的处理器处理内存的问题，可以类比总结出 JVM 内存操作的问题，下面介绍的 Java 内存模型的执行处理将围绕解决这 2 个问题展开：

**1 工作内存数据一致性** 各个线程操作数据时会保存使用到的主内存中的共享变量副本，当多个线程的运算任务都涉及同一个共享变量时，将导致各自的共享变量副本不一致，如果真的发生这种情况，数据同步回主内

存以谁的副本数据为准？Java 内存模型主要通过一系列的数据同步协议、规则来保证数据的一致性，后面再详细介绍。

**2 指令重排序优化** Java 中重排序通常是编译器或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。重排序分为两类：**编译期重排序和运行期重排序**，分别对应编译时和运行时环境。同样的，指令重排序不是随意重排序，它需要满足以下两个条件：

- 1 在单线程环境下不能改变程序运行的结果 即时编译器（和处理器）需要保证程序能够遵守 `as-if-serial` 属性。通俗地说，就是在单线程情况下，要给程序一个顺序执行的假象。即经过重排序的执行结果要与顺序执行的结果保持一致。
- 2 存在数据依赖关系的不允许重排序

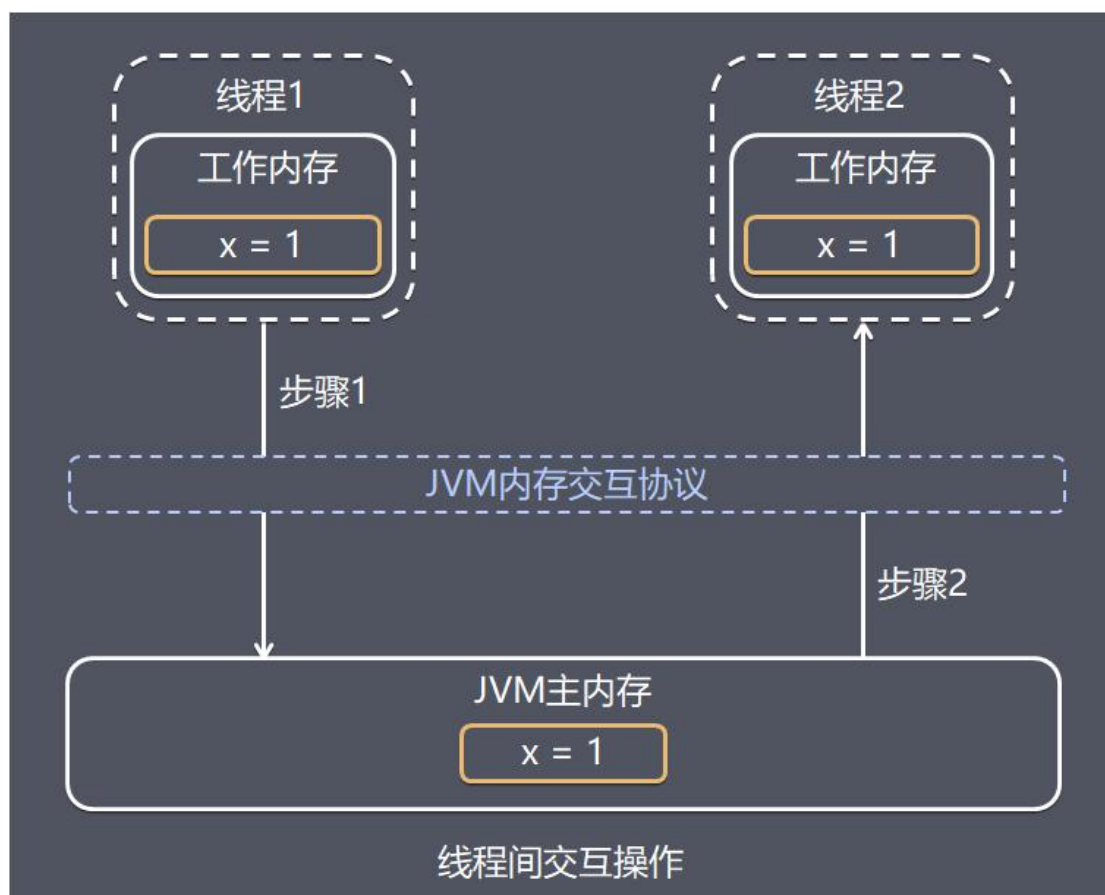
多线程环境下，如果线程处理逻辑之间存在依赖关系，有可能因为指令重排序导致运行结果与预期不同，后面再展开 Java 内存模型如何解决这种情况。

### 3 Java 内存间的交互操作

在理解 Java 内存模型的系列协议、特殊规则之前，我们先理解 Java 中内存间的交互操作。

#### 交互操作流程

为了更好理解内存的交互操作，以线程通信为例，我们看看具体如何进行线程间值的同步：



线程 1 和线程 2 都有主内存中共享变量 x 的副本，初始时，这 3 个内存中 x 的值都为 0。线程 1 中更新 x 的值为 1 之后同步到线程 2 主要涉及 2 个步骤：

- 1 线程 1 把线程工作内存中更新过的 x 的值刷新到主内存中
- 2 线程 2 到主内存中读取线程 1 之前已更新过的 x 变量

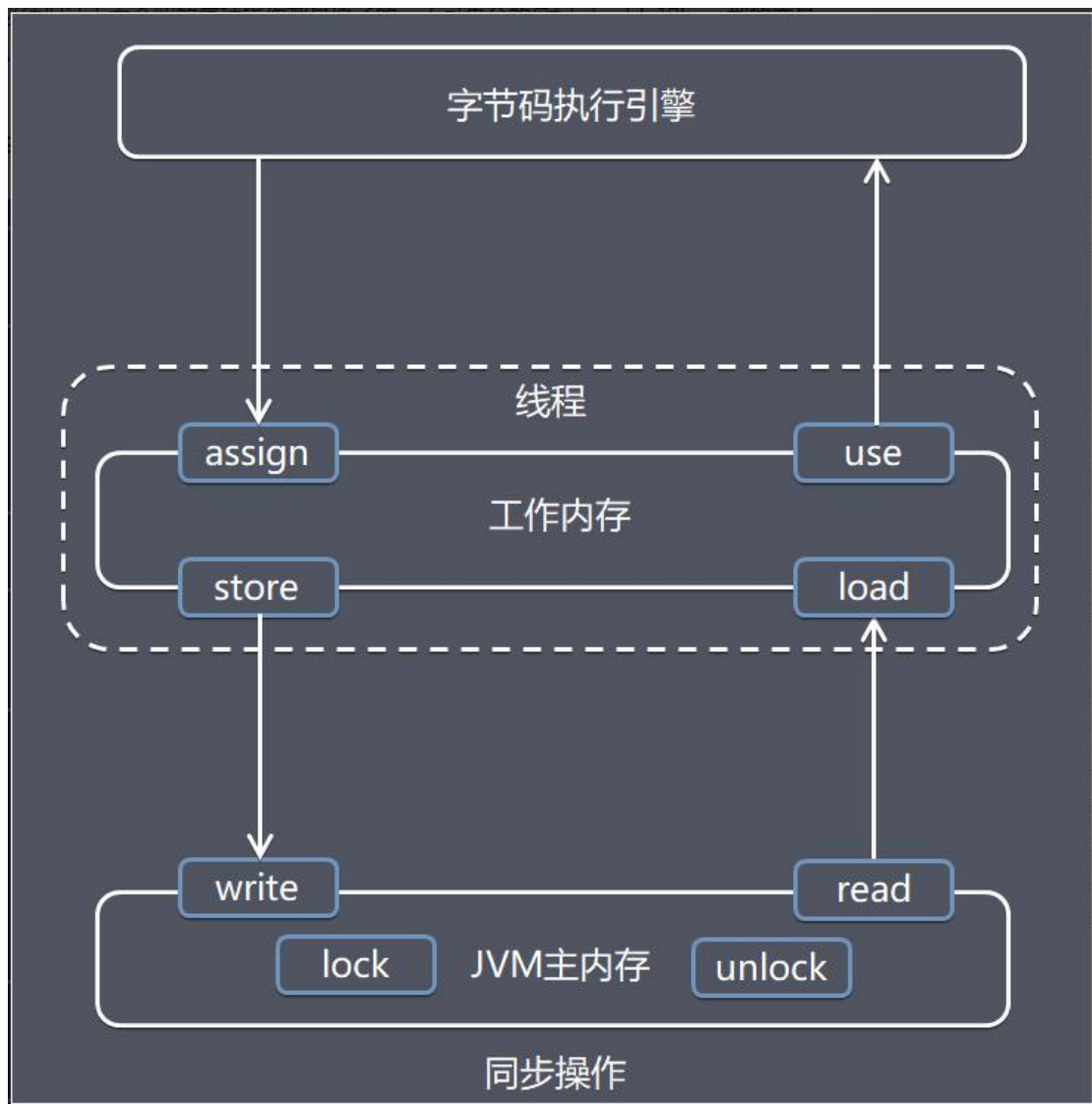
从整体上看，这 2 个步骤是线程 1 在向线程 2 发消息，这个通信过程必须经过主内存。线程对变量的所有操作（读取，赋值）都必须在**工作内存**中进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成，实现各个线程提供共享变量的可见性。

## 内存交互的基本操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java 内存模型中定义了下面介绍 8 种操作来完成。

虚拟机实现时必须保证下面介绍的每种操作都是原子的，不可再分的(对于 double 和 long 型的变量来说，load、store、read、和 write 操作在某些平台上允许有例外，后面会介绍)。

## 8 种基本操作



- **lock (锁定)** 作用于**主内存**的变量，它把一个变量标识为一条线程独占的状态。
- **unlock (解锁)** 作用于**主内存**的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- **read (读取)** 作用于**主内存**的变量，它把一个变量的值从主内存**传输**到线程的工作内存中，以便随后的 **load** 动作使用。
- **load (载入)** 作用于**工作内存**的变量，它把 **read** 操作从主内存中得到的变量值放入工作内存的变量副本中。
- **use (使用)** 作用于**工作内存**的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值得字节码指令时就会执行这个操作。
- **assign (赋值)** 作用于**工作内存**的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- **store (存储)** 作用于**工作内存**的变量，它把工作内存中一个变量的值传送到主内存中，以便随后 **write** 操作使用。

- **write (写入)** 作用于**主内存**的变量，它把 **store** 操作从工作内存中得到的变量的值放入主内存的变量中。

## 4 Java 内存模型运行规则

### 4.1 内存交互基本操作的 3 个特性

在介绍内存的交互的具体的 8 种基本操作之前，有必要先介绍一下操作的 3 个特性，Java 内存模型是围绕着在并发过程中如何处理这 3 个特性来建立的，这里先给出定义和基本实现的简单介绍，后面会逐步展开分析。

**原子性(Atomicity)** 即一个操作或者多个操作 **要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行**。即使在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程所干扰。

**可见性(Visibility)** 是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。正如上面“交互操作流程”中所说明的一样，JMM 是通过在线程 1 变量工作内存修改后将新值同步回主内存，线程 2 在变量读取前从主内存刷新变量值，这种**依赖主内存作为传递媒介**的方式来实现可见性。

**有序性(Ordering)** 有序性规则表现在以下两种场景：线程内和线程间

- **线程内** 从某个线程的角度看方法的执行，指令会按照一种叫“串行”(as-if-serial)的方式执行，此种方式已经应用于顺序编程语言。
- **线程间** 这个线程“观察”到其他线程并发地执行非同步的代码时，由于指令重排序优化，任何代码都有可能交叉执行。唯一起作用的约束是：对于同步方法，同步块(synchronized 关键字修饰)以及 volatile 字段的操作仍维持相对有序。

Java 内存模型的一系列运行规则看起来有点繁琐，但总结起来，是**围绕原子性、可见性、有序性特征建立**。归根究底，是为实现共享变量的在多个线程的工作内存的**数据一致性**，多线程并发，指令重排序优化的环境中程序能如预期运行。

### 4.2 happens-before 关系

介绍系列规则之前，首先了解一下 happens-before 关系：用于描述下 2 个操作的内存可见性：**如果操作 A happens-before 操作 B，那么 A 的结果对 B 可见**。

happens-before 关系的分析需要分为**单线程和多线程**的情况：

**单线程下的 happens-before** 字节码的先后顺序天然包含 happens-before 关系：因为单线程内共享一份工作内存，不存在数据一致性的问题。在

程序控制流路径中靠前的字节码 **happens-before** 靠后的字节码，即靠前的字节码执行完之后操作结果对靠后的字节码可见。然而，这并不意味着前者一定在后者之前执行。实际上，如果后者不依赖前者的运行结果，那么它们可能会被重排序。

**多线程下的 happens-before** 多线程由于每个线程有共享变量的副本，如果没有对共享变量做同步处理，线程 1 更新执行操作 A 共享变量的值之后，线程 2 开始执行操作 B，此时操作 A 产生的结果对操作 B 不一定可见。

为了方便程序开发，Java 内存模型实现了下述支持 **happens-before** 关系的操作：

- 程序次序规则 一个线程内，按照代码顺序，书写在前面的操作 **happens-before** 书写在后面的操作。
- 锁定规则 一个 `unlock` 操作 **happens-before** 后面对同一个锁的 `lock` 操作。
- `volatile` 变量规则 对一个变量的写操作 **happens-before** 后面对这个变量的读操作。
- 传递规则 如果操作 A **happens-before** 操作 B，而操作 B 又 **happens-before** 操作 C，则可以得出操作 A **happens-before** 操作 C。
- 线程启动规则 `Thread` 对象的 `start()` 方法 **happens-before** 此线程的每个一个动作。
- 线程中断规则 对线程 `interrupt()` 方法的调用 **happens-before** 被中断线程的代码检测到中断事件的发生。
- 线程终结规则 线程中所有的操作都 **happens-before** 线程的终止检测，我们可以通过 `Thread.join()` 方法结束、`Thread.isAlive()` 的返回值手段检测到线程已经终止执行。
- 对象终结规则 一个对象的初始化完成 **happens-before** 他的 `finalize()` 方法的开始

### 4.3 内存屏障

Java 中如何保证底层操作的有序性和可见性？可以通过内存屏障。

内存屏障是被插入两个 CPU 指令之间的一种指令，用来禁止处理器指令发生重排序（像屏障一样），从而保障**有序性**的。另外，为了达到屏障的效果，它也会使处理器写入、读取值之前，将主内存的值写入高速缓存，清空无效队列，从而保障**可见性**。

举个例子：

```
Store1;
Store2;
Load1;
StoreLoad; //内存屏障
Store3;
Load2;
Load3;
```

对于上面的一组 CPU 指令 (Store 表示写入指令, Load 表示读取指令), StoreLoad 屏障之前的 Store 指令无法与 StoreLoad 屏障之后的 Load 指令进行交换位置, 即**重排序**。但是 StoreLoad 屏障之前和之后的指令是可以互换位置的, 即 Store1 可以和 Store2 互换, Load2 可以和 Load3 互换。

常见有 4 种屏障

- **LoadLoad 屏障:** 对于这样的语句 Load1; LoadLoad; Load2, 在 Load2 及后续读取操作要读取的数据被访问前, 保证 Load1 要读取的数据被读取完毕。
- **StoreStore 屏障:** 对于这样的语句 Store1; StoreStore; Store2, 在 Store2 及后续写入操作执行前, 保证 Store1 的写入操作对其它处理器可见。
- **LoadStore 屏障:** 对于这样的语句 Load1; LoadStore; Store2, 在 Store2 及后续写入操作被执行前, 保证 Load1 要读取的数据被读取完毕。
- **StoreLoad 屏障:** 对于这样的语句 Store1; StoreLoad; Load2, 在 Load2 及后续所有读取操作执行前, 保证 Store1 的写入对所有处理器可见。它的开销是四种屏障中最大的 (冲刷写缓冲器, 清空无效化队列)。在大多数处理器的实现中, 这个屏障是个万能屏障, 兼具其它三种内存屏障的功能。

Java 中对内存屏障的使用在一般的代码中不太容易见到, 常见的有 volatile 和 synchronized 关键字修饰的代码块(后面再展开介绍), 还可以通过 Unsafe 这个类来使用内存屏障。

## 4.4 8 种操作同步的规则

JMM 在执行前面介绍 8 种基本操作时, 为了保证内存间数据一致性, JMM 中规定需要满足以下规则:

- **规则 1:** 如果要把一个变量从主内存中复制到工作内存, 就需要按顺序的执行 read 和 load 操作, 如果把变量从工作内存中同步回主内存中, 就要按顺序的执行 store 和 write 操作。但 Java 内存模型只要求上述操作必须按顺序执行, 而没有保证必须是连续执行。
- **规则 2:** 不允许 read 和 load、store 和 write 操作之一单独出现。
- **规则 3:** 不允许一个线程丢弃它的最近 assign 的操作, 即变量在工作内存中改变了之后必须同步到主内存中。
- **规则 4:** 不允许一个线程无原因的 (没有发生过任何 assign 操作) 把数据从工作内存同步回主内存中。
- **规则 5:** 一个新的变量只能在主内存中诞生, 不允许在工作内存中直接使用一个未被初始化 (load 或 assign) 的变量。即就是对一个变量实施 use 和 store 操作之前, 必须先执行过了 load 或 assign 操作。
- **规则 6:** 一个变量在同一个时刻只允许一条线程对其进行 lock 操作, 但 lock 操作可以被同一条线程重复执行多次, 多次执行 lock 后, 只有执行相同次数的 unlock 操作, 变量才会被解锁。所以 lock 和 unlock 必须成对出现。
- **规则 7:** 如果对一个变量执行 lock 操作, 将会清空工作内存中此变量的值, 在执行引擎使用这个变量前需要重新执行 load 或 assign 操作初始化变量的值。

- 规则 8: 如果一个变量事先没有被 lock 操作锁定, 则不允许对它执行 unlock 操作; 也不允许去 unlock 一个被其他线程锁定的变量。
- 规则 9: 对一个变量执行 unlock 操作之前, 必须先把此变量同步到主内存中 (执行 store 和 write 操作)

看起来这些规则有些繁琐, 其实也不难理解:

- 规则 1、规则 2 工作内存中的共享变量作为主内存的副本, 主内存变量的值同步到工作内存需要 read 和 load 一起使用, 工作内存中的变量的值同步回主内存需要 store 和 write 一起使用, 这 2 组操作各自都是一个固定的有序搭配, 不允许单独出现。
- 规则 3、规则 4 由于工作内存中的共享变量是主内存的副本, 为保证数据一致性, 当工作内存中的变量被字节码引擎重新赋值, 必须同步回主内存。如果工作内存的变量没有被更新, 不允许无原因同步回主内存。
- 规则 5 由于工作内存中的共享变量是主内存的副本, 必须从主内存诞生。
- 规则 6、7、8、9 为了并发情况下安全使用变量, 线程可以基于 lock 操作独占主内存中的变量, 其他线程不允许使用或 unlock 该变量, 直到变量被线程 unlock。

## 4.5 volatile 型变量的特殊规则

volatile 的中文意思是不稳定的, 易变的, 用 volatile 修饰变量是为了保证变量的可见性。

### volatile 的语义

volatile 主要有下面 2 种语义

#### 语义 1 保证可见性

保证了不同线程对该变量操作的内存可见性。

这里保证可见性是不等同于 volatile 变量并发操作的安全性, 保证可见性具体一点解释:

线程写 volatile 变量的过程:

- 1 改变线程工作内存中 volatile 变量副本的值
- 2 将改变后的副本的值从工作内存刷新到主内存

线程读 volatile 变量的过程:

- 1 从主内存中读取 volatile 变量的最新值到线程的工作内存中
- 2 从工作内存中读取 volatile 变量的副本



但是如果多个线程同时把更新后的变量值同时刷新回主内存,可能导致得到的值不是预期结果:

举个例子: 定义 `volatile int count = 0`, 2 个线程同时执行 `count++` 操作, 每个线程都执行 500 次, 最终结果小于 1000, 原因是每个线程执行 `count++` 需要以下 3 个步骤:

- 步骤 1 线程从主内存读取最新的 `count` 的值
- 步骤 2 执行引擎把 `count` 值加 1, 并赋值给线程工作内存
- 步骤 3 线程工作内存把 `count` 值保存到主内存 有可能某一时刻 2 个线程在步骤 1 读取到的值都是 100, 执行完步骤 2 得到的值都是 101, 最后刷新了 2 次 101 保存到主内存。

## 语义 2 禁止进行指令重排序

具体一点解释, 禁止重排序的规则如下:

- 当程序执行到 `volatile` 变量的读操作或者写操作时, 在其前面的操作的更改肯定全部已经进行, 且结果已经对后面的操作可见; 在其后面的操作肯定还没有进行;
- 在进行指令优化时, 不能将在对 `volatile` 变量访问的语句放在其后面执行, 也不能把 `volatile` 变量后面的语句放到其前面执行。

普通的变量仅仅会保证该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果, 而不能保证赋值操作的顺序与程序代码中的执行顺序一致。

举个例子:

```
volatile boolean initialized = false;

// 下面代码线程 A 中执行
// 读取配置信息, 当读取完成后将 initialized 设置为 true 以通知其他线程配置可用
doSomethingReadConfig();
initialized = true;

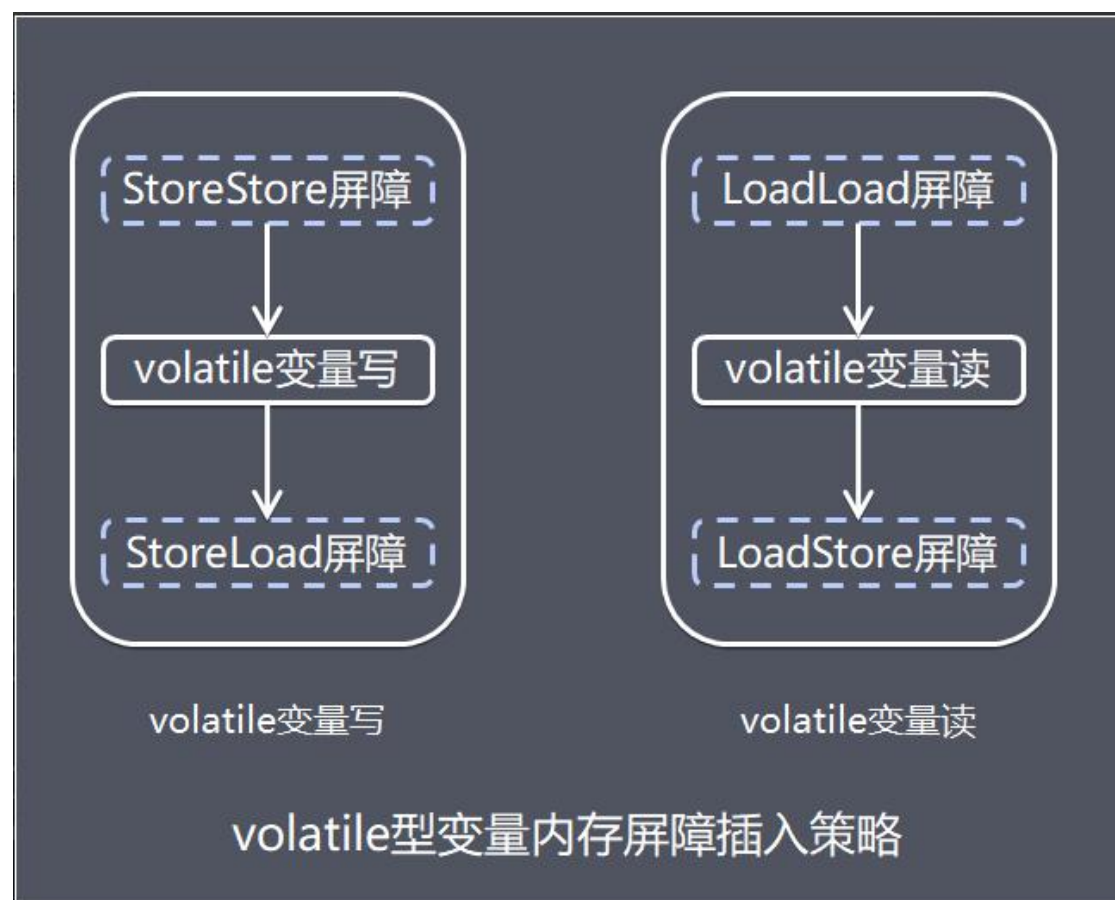
// 下面代码线程 B 中执行
// 等待 initialized 为 true, 代表线程 A 已经把配置信息初始化完成 while
(!initialized) {
    sleep();
}
// 使用线程 A 初始化好的配置信息 doSomethingWithConfig(); 复制代码
```

上面代码中如果定义 `initialized` 变量时没有使用 `volatile` 修饰, 就有可能由于指令重排序的优化, 导致线程 A 中最后一句代码 `"initialized = true"` 在 `"doSomethingReadConfig()"` 之前被执行, 这样会导致线程 B 中使用配置信息的代

码就可能出现错误，而 `volatile` 关键字就禁止重排序的语义可以避免此类情况发生。

## volatile 型变量实现原理

具体实现方式是在编译期生成字节码时，会在指令序列中增加内存屏障来保证，下面是基于保守策略的 JMM 内存屏障插入策略：



在每个 `volatile` 写操作的前面插入一个 **StoreStore** 屏障。该屏障除了保证了屏障之前的写操作和该屏障之后的写操作不能重排序，还会保证了 `volatile` 写操作之前，任何的读写操作都会先于 `volatile` 被提交。

在每个 `volatile` 写操作的后面插入一个 **StoreLoad** 屏障。该屏障除了使 `volatile` 写操作不会与之后的读操作重排序外，还会刷新处理器缓存，使 `volatile` 变量的写更新对其他线程可见。

在每个 `volatile` 读操作的后面插入一个 **LoadLoad** 屏障。该屏障除了使 `volatile` 读操作不会与之前的写操作发生重排序外，还会刷新处理器缓存，使 `volatile` 变量读取的为最新值。

在每个 `volatile` 读操作的后面插入一个 `LoadStore` 屏障。该屏障除了禁止了 `volatile` 读操作与其之后的任何写操作进行重排序，还会刷新处理器缓存，使其他线程 `volatile` 变量的写更新对 `volatile` 读操作的线程可见。

## volatile 型变量使用场景

总结起来，就是“一次写入，到处读取”，某一线程负责更新变量，其他线程只读取变量(不更新变量)，并根据变量的新值执行相应逻辑。例如状态标志位更新，观察者模型变量值发布。

## 4.6 final 型变量的特殊规则

我们知道，`final` 成员变量必须在声明的时候初始化或者在构造器中初始化，否则就会报编译错误。`final` 关键字的可见性是指：被 `final` 修饰的字段在声明时或者构造器中，一旦初始化完成，那么在其他线程无须同步就能正确看见 `final` 字段的值。这是因为一旦初始化完成，`final` 变量的值立刻回写到主内存。

## 4.7 synchronized 的特殊规则

通过 `synchronized` 关键字包住的代码区域，对数据的读写进行控制：

- 读数据 当线程进入到该区域读取变量信息时，对数据的读取也不能从工作内存读取，只能从内存中读取，保证读到的是最新的值。
- 写数据 在同步区内对变量的写入操作，在离开同步区时就将当前线程内的数据刷新到内存中，保证更新的数据对其他线程的可见性。

## 4.8 long 和 double 型变量的特殊规则

Java 内存模型要求 `lock`、`unlock`、`read`、`load`、`assign`、`use`、`store`、`write` 这 8 种操作都具有原子性，但是对于 64 位的数据类型(`long` 和 `double`)，在模型中特别定义相对宽松的规定：允许虚拟机将没有被 `volatile` 修饰的 64 位数据的读写操作分为 2 次 32 位的操作来进行。也就是说虚拟机可选择不保证 64 位数据类型的 `load`、`store`、`read` 和 `write` 这 4 个操作的原子性。由于这种非原子性，有可能导致其他线程读到同步未完成的“32 位的半个变量”的值。

不过实际开发中，Java 内存模型强烈建议虚拟机把 64 位数据的读写实现为具有原子性，目前各种平台下的商用虚拟机都选择把 64 位数据的读写操作作为原子操作来对待，因此我们在编写代码时一般不需要把用到的 `long` 和 `double` 变量专门声明为 `volatile`。

## 5 总结

由于 Java 内存模型涉及系列规则，网上的文章大部分就是对这些规则进行解析，但是很多没有解释为什么需要这些规则，这些规则的作用，其实这是不利于初学者学习的，容易绕进去这些繁琐规则不知所以然，下面谈谈我的一点学习知识的个人体会：

**学习知识的过程不是等同于只是理解知识和记忆知识，而是要对知识解决的问题的输入和输出建立连接**，知识的本质是解决问题，所以在学习之前要理解问题，理解这个问题要的输出和输出，而知识就是输入到输出的一个关系映射。知识的学习要结合大量的例子来**理解这个映射关系，然后压缩知识**，华罗庚说过：“把一本书读厚，然后再读薄”，解释的就是这个道理，先结合大量的例子理解知识，然后再压缩知识。

以学习 Java 内存模型为例：

- 理解问题，明确输入输出 首先理解 Java 内存模型是什么，有什么用，解决什么问题
- 理解内存模型系列协议 结合大量例子理解这些协议规则
- 压缩知识 大量规则其实就是通过数据同步协议，保证内存副本之间的数据一致性，同时防止重排序对程序的影响。

## 6. 垃圾回收算法（JVM）

JVM 类加载机制、垃圾回收算法对比、Java 虚拟机结构

当你讲到分代回收算法的时候，不免会被追问到新生对象是怎么从年轻代到老年代的，以及可以作为 root 结点的对象有哪些两个问题。

- 1、谈谈对 JVM 的理解？
- 2、JVM 内存区域，开线程影响哪块区域内存？
- 3、对 Dalvik、ART 虚拟机有什么了解？对比？

ART 的机制与 Dalvik 不同。

在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器（just in time ，JIT）转换为机器码，这会拖慢应用的运行效率。

而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，极大的提高了程序的运行效率，同时减少了手机的耗电量，使其成为真正的本地应用。这个过程叫做预编译（AOT,Ahead-Of-Time）。这样的话，应用的启动(首次)和执行都会变得更加快速。

优点：

系统性能的显著提升。

应用启动更快、运行更快、体验更流畅、触感反馈更及时。

更长的电池续航能力。

支持更低的硬件。

缺点：

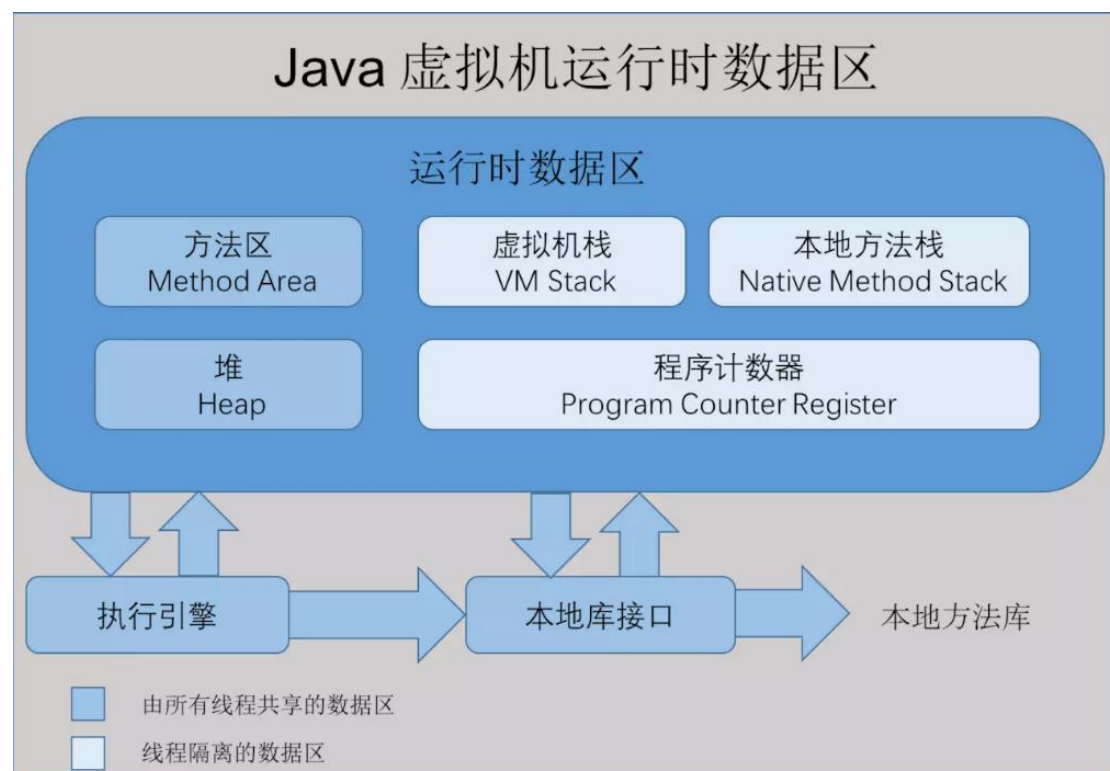
机器码占用的存储空间更大，字节码变为机器码之后，可能会增加 10%-20%（不过在应用中，可执行的代码常常只是一部分。比如最新的 Google+ APK 是 28.3 MB，但是代码只有 6.9 MB。）

应用的安装时间会变长。

## 7、垃圾回收机制和调用 `System.gc()` 的区别？

### 1. Java 内存区域与内存溢出异常

#### 1.1 运行时数据区域



##### 1.1.1 程序计数器

内存空间小，线程私有。字节码解释器工作就是通过改变这个计数器的值来选取下一条需要执行指令的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖计数器完成

如果线程正在执行一个 **Java** 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 **Native** 方法，这个计数器的值则为 (**Undefined**)。此内存区域是唯一一个在 **Java** 虚拟机规范中没有规定任何 **OutOfMemoryError** 情况的区域。

### 1.1.2 Java 虚拟机栈

线程私有，生命周期和线程一致。描述的是 **Java** 方法执行的内存模型：每个方法在执行时都会创建一个栈帧(**Stack Frame**)用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行结束，就对应着一个栈帧从虚拟机栈中入栈到出栈的过程。

**局部变量表**：存放了编译期可知的各种基本类型(**boolean**、**byte**、**char**、**short**、**int**、**float**、**long**、**double**)、对象引用(**reference** 类型)和 **returnAddress** 类型(指向了一条字节码指令的地址)

**StackOverflowError**：线程请求的栈深度大于虚拟机所允许的深度。

**OutOfMemoryError**：如果虚拟机栈可以动态扩展，而扩展时无法申请到足够的内存。

### 1.1.3 本地方法栈

区别于 **Java** 虚拟机栈的是，**Java** 虚拟机栈为虚拟机执行 **Java** 方法(也就是字节码)服务，而本地方法栈则为虚拟机使用到的 **Native** 方法服务。也会有 **StackOverflowError** 和 **OutOfMemoryError** 异常。

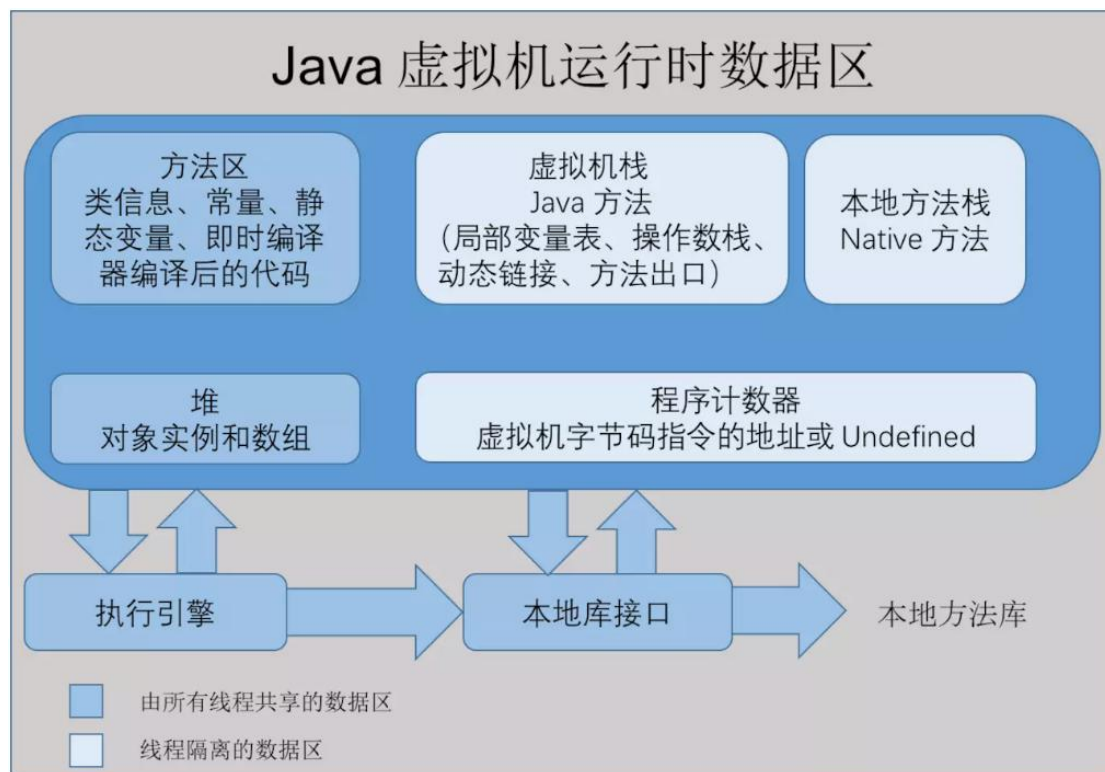
### 1.1.4 Java 堆

对于绝大多数应用来说，这块区域是 **JVM** 所管理的内存中最大的一块。线程共享，主要是存放对象实例和数组。内部会划分出多个线程私有的分配缓冲区(**Thread Local Allocation Buffer**, **TLAB**)。可以位于物理上不连续的空间，但是逻辑上要连续。

**OutOfMemoryError**：如果堆中没有内存完成实例分配，并且堆也无法再扩展时，抛出该异常。

### 1.1.5 方法区

属于共享内存区域，存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。



### 1.1.6 运行时常量池

属于方法区一部分，用于存放编译期生成的各种字面量和符号引用。编译器和运行期(`String` 的 `intern()`)都可以将常量放入池中。内存有限，无法申请时抛出 `OutOfMemoryError`。

### 1.1.7 直接内存

非虚拟机运行时数据区的部分

在 JDK 1.4 中新加入 `NIO (New Input/Output)` 类，引入了一种基于通道(`Channel`)和缓存(`Buffer`)的 I/O 方式，它可以使用 `Native` 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 `DirectByteBuffer` 对象作为这块内存的引用进行操作。可以避免在 Java 堆和 `Native` 堆中来回的数据耗时操作。

**OutOfMemoryError**：会受到本机内存限制，如果内存区域总和大于物理内存限制从而导致动态扩展时出现该异常。

## 1.2 HotSpot 虚拟机对象探秘

主要介绍数据是如何创建、如何布局以及如何访问的。

### 1.2.1 对象的创建

创建过程比较复杂，建议看书了解，这里提供个人的总结。

遇到 `new` 指令时，首先检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，执行相应的

类加载。

类加载检查通过之后，为新对象分配内存(内存大小在类加载完成后便可确认)。在堆的空闲内存中划分一块区域(‘指针碰撞-内存规整’或‘空闲列表-内存交错’的分配方式)。

前面讲的每个线程在堆中都会有私有的分配缓冲区(TLAB)，这样可以很大程度避免在并发情况下频繁创建对象造成的线程不安全。

内存空间分配完成后会初始化为 0(不包括对象头)，接下来就是填充对象头，把对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息存入对象头。

执行 new 指令后执行 init 方法后才算一份真正可用的对象创建完成。

### 1.2.2 对象的内存布局

在 HotSpot 虚拟机中，分为 3 块区域：对象头(Header)、实例数据(Instance Data)和对齐填充(Padding)

对象头(Header)：包含两部分，第一部分用于存储对象自身的运行时数据，如哈希码、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等，32 位虚拟机占 32 bit，64 位虚拟机占 64 bit。官方称为 ‘Mark Word’。第二部分是类型指针，即对象指向它的类的元数据指针，虚拟机通过这个指针确定这个对象是哪个类的实例。另外，如果是 Java 数组，对象头中还必须有一块用于记录数组长度的数据，因为普通对象可以通过 Java 对象元数据确定大小，而数组对象不可以。

实例数据(Instance Data)：程序代码中所定义的各种类型的字段内容(包含父类继承下来的和子类中定义的)。

对齐填充(Padding)：不是必然需要，主要是占位，保证对象大小是某个字节的整数倍。

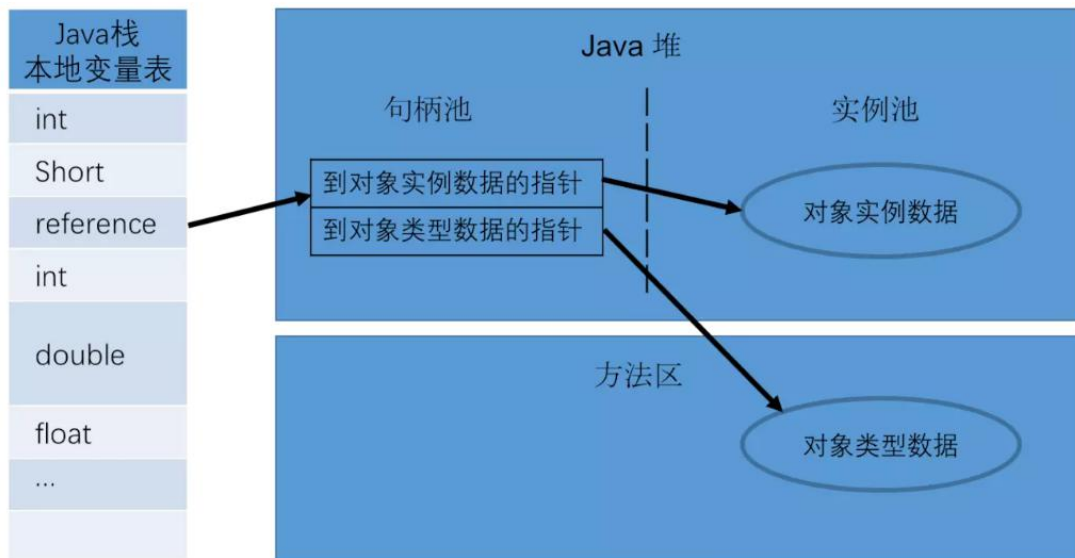
### 1.2.3 对象的访问定位

使用对象时，通过栈上的 reference 数据来操作堆上的具体对象。

通过句柄访问

Java 堆中会分配一块内存作为句柄池。reference 存储的是句柄地址。详情见图。

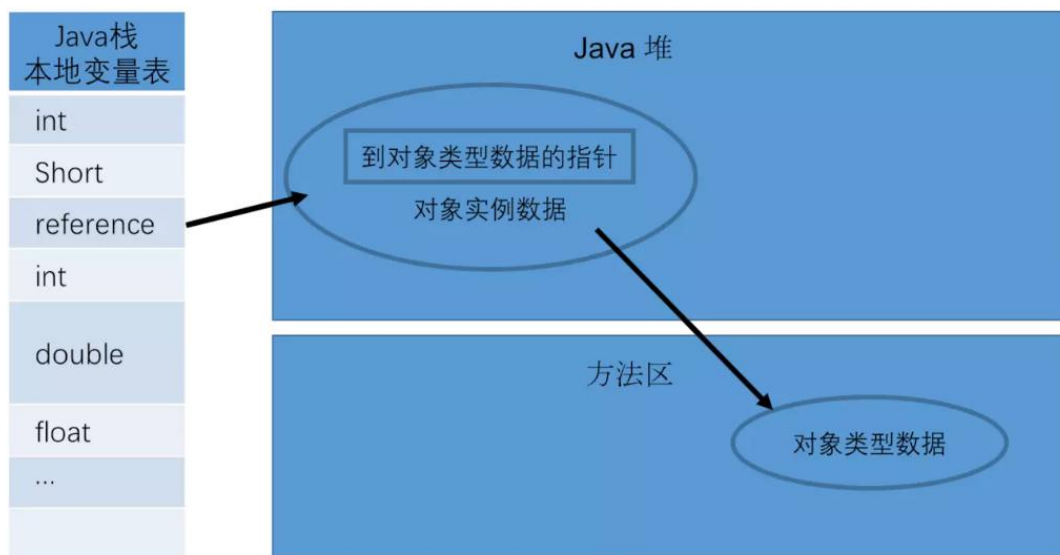




## 通过句柄访问对象

使用直接指针访问

reference 中直接存储对象地址



## 通过直接指针访问对象

比较：使用句柄的最大好处是 `reference` 中存储的是稳定的句柄地址，在对象移动(GC)是只改变实例数据指针地址，`reference` 自身不需要修改。直接指针访问的最大好处是速度快，节省了一次指针定位的时间开销。如果是对象频繁 GC 那么句柄方法好，如果是对象频繁访问则直接指针访问好。

### 1.3 实战

// 待填

## 2. 垃圾回收器与内存分配策略

### 2.1 概述

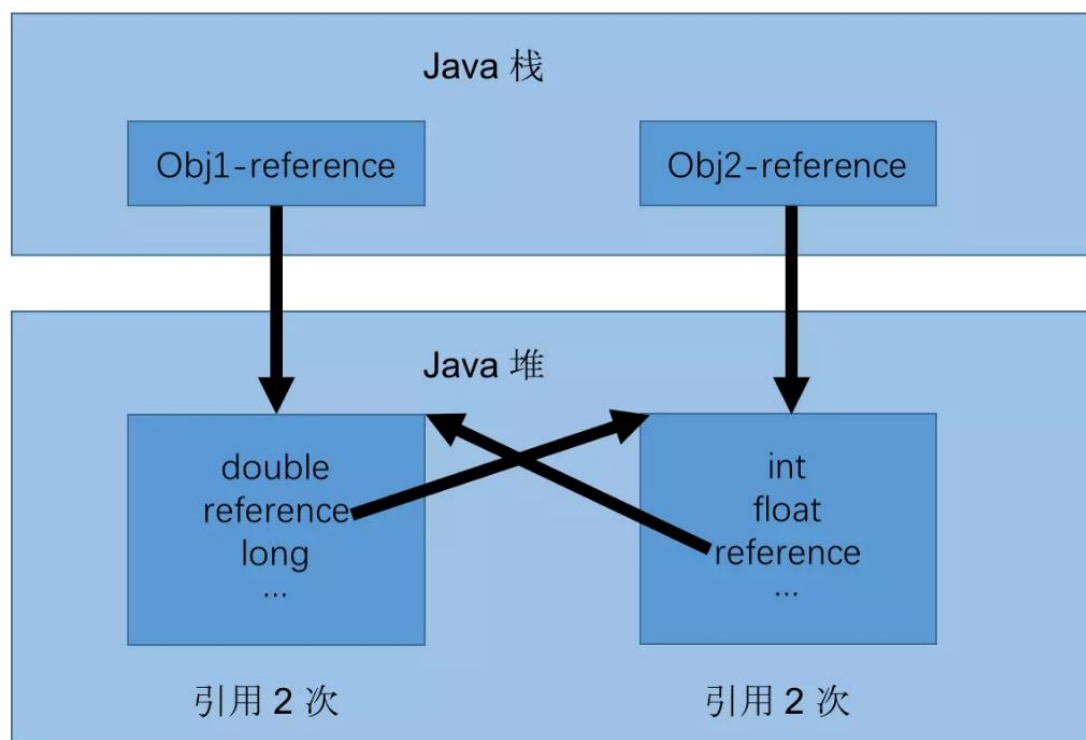
程序计数器、虚拟机栈、本地方法栈 3 个区域随线程生灭(因为是线程私有), 栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。而 Java 堆和方法区则不一样, 一个接口中的多个实现类需要的内存可能不一样, 一个方法中的多个分支需要的内存也可能不一样, 我们只有在程序处于运行期才知道那些对象会创建, 这部分内存的分配和回收都是动态的, 垃圾回收期所关注的就是这部分内存。

### 2.2 对象已死吗?

在进行内存回收之前要做的事情就是判断那些对象是‘死’的, 哪些是‘活’的。

#### 2.2.1 引用计数法

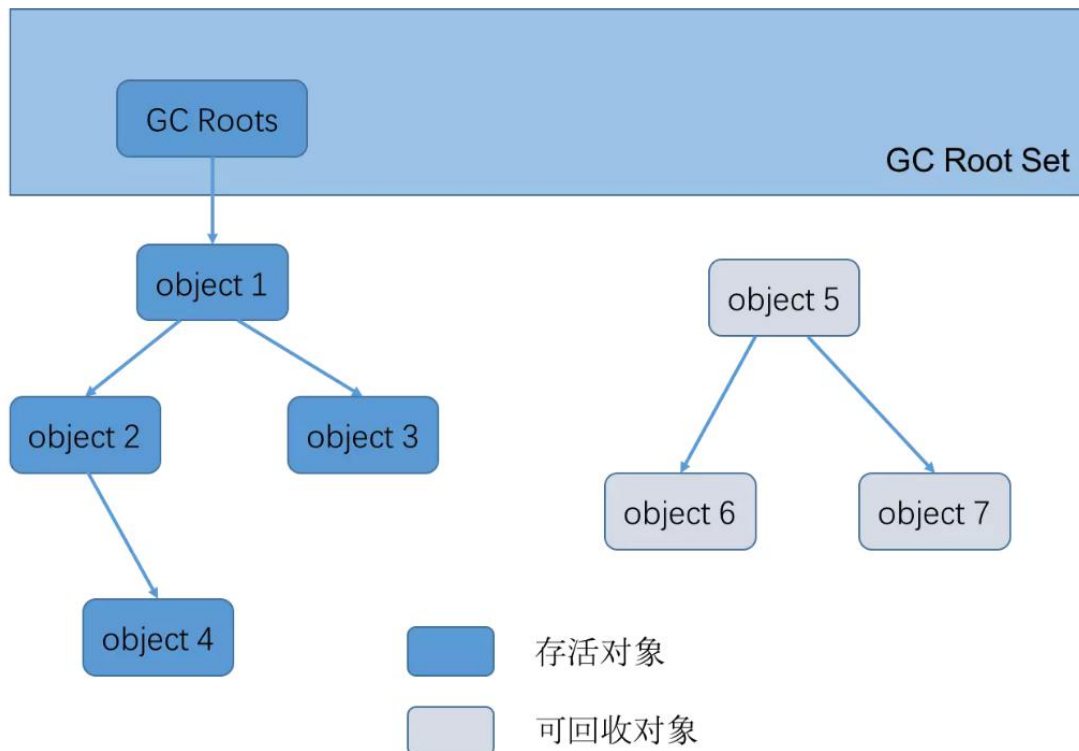
给对象添加一个引用计数器。但是难以解决循环引用问题。



从图中可以看出, 如果不下小心直接把 Obj1-reference 和 Obj2-reference 置 null。则在 Java 堆当中的两块内存依然保持着互相引用无法回收。

#### 2.2.2 可达性分析法

通过一系列的 ‘GC Roots’ 的对象作为起始点, 从这些节点出发所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连的时候说明对象不可用。



可作为 GC Roots 的对象：

- 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中 JNI(即一般说的 Native 方法) 引用的对象

### 2.2.3 再谈引用

前面的两种方式判断存活时都与‘引用’有关。但是 JDK 1.2 之后，引用概念进行了扩充，下面具体介绍。

下面四种引用强度一次逐渐减弱

强引用

类似于 `Object obj = new Object();` 创建的，只要强引用在就不回收。

软引用

`SoftReference` 类实现软引用。在系统要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行二次回收。

弱引用

`WeakReference` 类实现弱引用。对象只能生存到下一次垃圾收集之前。在垃圾收集器工作时，

无论内存是否足够都会回收掉只被弱引用关联的对象。

## 虚引用

`PhantomReference` 类实现虚引用。无法通过虚引用获取一个对象的实例，为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

### 2.2.4 生存还是死亡

即使在可达性分析算法中不可达的对象，也并非是“facebook”的，这时候它们暂时出于“缓刑”阶段，一个对象的真正死亡至少要经历两次标记过程：如果对象在进行中可达性分析后没有与 `GC Roots` 相连接的引用链，那他将会被第一次标记并且进行一次筛选，筛选条件是此对象是否有必要执行 `finalize()` 方法。当对象没有覆盖 `finalize()` 方法，或者 `finalize()` 方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行 `finalize()` 方法，那么这个对象竟会放置在一个叫做 `F-Queue` 的队列中，并在稍后由一个由虚拟机自动建立的、低优先级的 `Finalizer` 线程去执行它。这里所谓的“执行”是指虚拟机会出发这个方法，并不承诺或等待他运行结束。`finalize()` 方法是对象逃脱死亡命运的最后一次机会，稍后 `GC` 将对 `F-Queue` 中的对象进行第二次小规模标记，如果对象要在 `finalize()` 中成功拯救自己 —— 只要重新与引用链上的任何一个对象简历关联即可。

`finalize()` 方法只会被系统自动调用一次。

### 2.2.5 回收方法区

在堆中，尤其是在新生代中，一次垃圾回收一般可以回收 70% ~ 95% 的空间，而永久代的垃圾收集效率远低于此。

永久代垃圾回收主要两部分内容：废弃的常量和无用的类。

判断废弃常量：一般是判断没有该常量的引用。

判断无用的类：要以下三个条件都满足

该类所有的实例都已经回收，也就是 `Java` 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 `java.lang.Class` 对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法

## 2.3 垃圾回收算法

仅提供思路

### 2.3.1 标记 —— 清除算法

直接标记清除就可。

两个不足：

效率不高

空间会产生大量碎片

### 2.3.2 复制算法

把空间分成两块，每次只对其中一块进行 GC。当这块内存使用完时，就将还存活的对象复制到另一块上面。

解决前一种方法的不足，但是会造成空间利用率低下。因为大多数新生代对象都不会熬过第一次 GC。所以没必要 1:1 划分空间。可以分一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 空间和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另一块 Survivor 上，最后清理 Eden 和 Survivor 空间。大小比例一般是 8:1:1，每次浪费 10% 的 Survivor 空间。但是这里有一个问题就是如果存活的大于 10% 怎么办？这里采用一种分配担保策略：多出来的对象直接进入老年代。

### 2.3.3 标记-整理算法

不同于针对新生代的复制算法，针对老年代的特点，创建该算法。主要是把存活对象移到内存的一端。

### 2.3.4 分代回收

根据存活对象划分几块内存区，一般是分为新生代和老年代。然后根据各个年代的特点制定相应的回收算法。

新生代

每次垃圾回收都有大量对象死去，只有少量存活，选用复制算法比较合理。

老年代

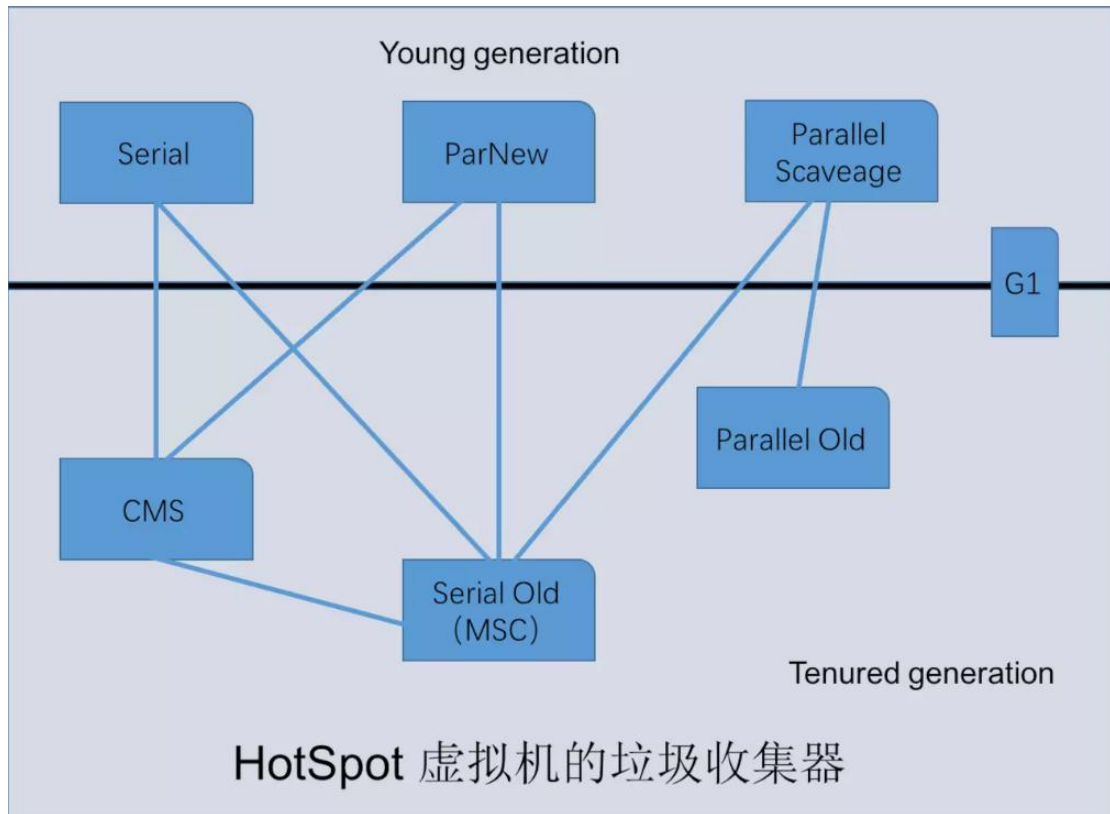
老年代中对象存活率较高、没有额外的空间分配对它进行担保。所以必须使用 标记 —— 清除 或者 标记 —— 整理 算法回收。

## 2.4 HotSpot 的算法实现

// 待填

## 2.5 垃圾回收器

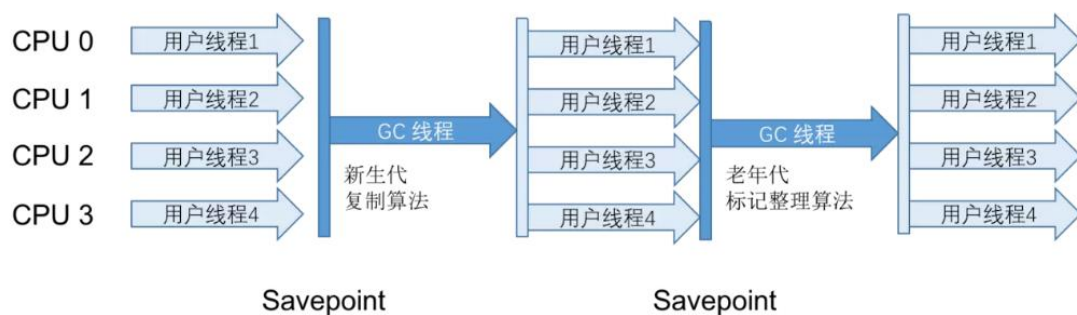
收集算法是内存回收的理论，而垃圾回收器是内存回收的实践。



说明：如果两个收集器之间存在连线说明他们之间可以搭配使用。

### 2.5.1 Serial 收集器

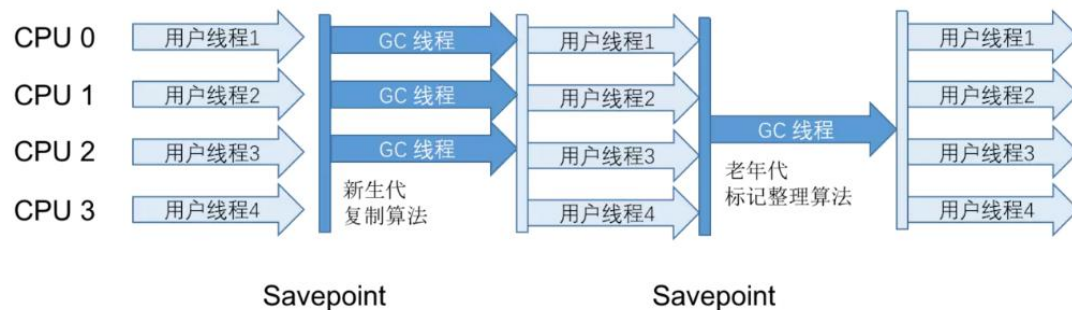
这是一个单线程收集器。意味着它只会使用一个 CPU 或一条收集线程去完成收集工作 ,并且在进行垃圾回收时必须暂停其它所有的工作线程直到收集结束。



Serial / Serial Old 收集器运行示意图

## 2.5.2 ParNew 收集器

可以认为是 Serial 收集器的多线程版本。



ParNew / Serial Old 收集器运行示意图

并行: Parallel

指多条垃圾收集线程并行工作, 此时用户线程处于等待状态

并发: Concurrent

指用户线程和垃圾回收线程同时执行(不一定是并行, 有可能是交叉执行), 用户进程在运行, 而垃圾回收线程在另一个 CPU 上运行。

## 2.5.3 Parallel Scavenge 收集器

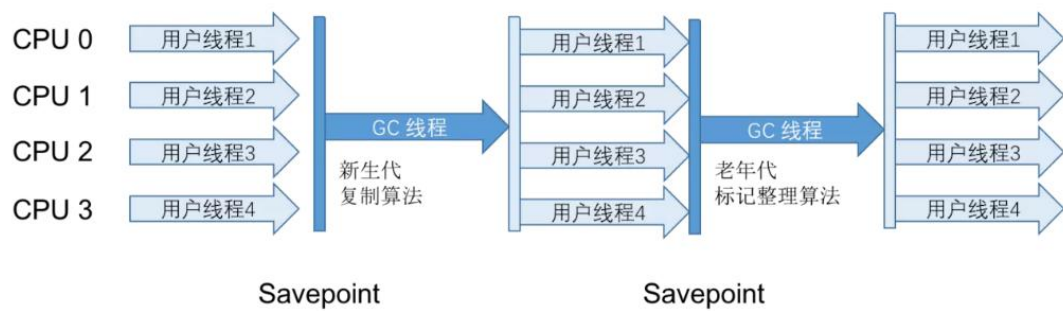
这是一个新生代收集器, 也是使用复制算法实现, 同时也是并行的多线程收集器。

CMS 等收集器的关注点是尽可能地缩短垃圾收集时用户线程所停顿的时间, 而 Parallel Scavenge 收集器的目的是达到一个可控制的吞吐量( $\text{Throughput} = \text{运行用户代码时间} / (\text{运行用户代码时间} + \text{垃圾收集时间})$ )。

作为一个吞吐量优先的收集器, 虚拟机会根据当前系统的运行情况收集性能监控信息, 动态调整停顿时间。这就是 GC 的自适应调整策略(GC Ergonomics)。

## 2.5.4 Serial Old 收集器

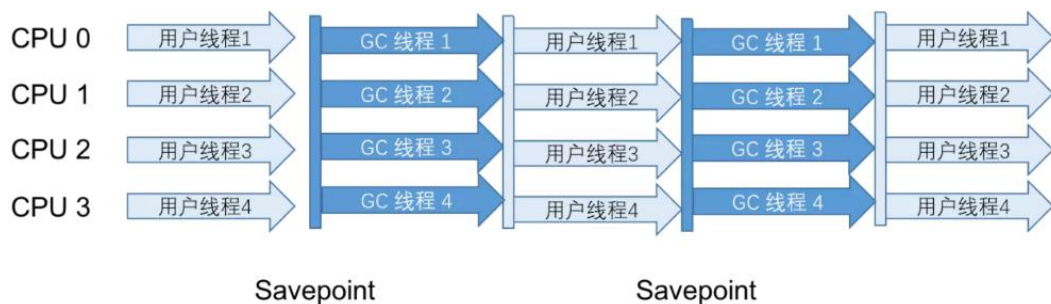
收集器的老年代版本, 单线程, 使用 标记 —— 整理。



Serial / Serial Old 收集器运行示意图

### 2.5.5 Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本。多线程，使用 标记 — 整理



Parallel Scavenge / Parallel Old 收集器运行示意图

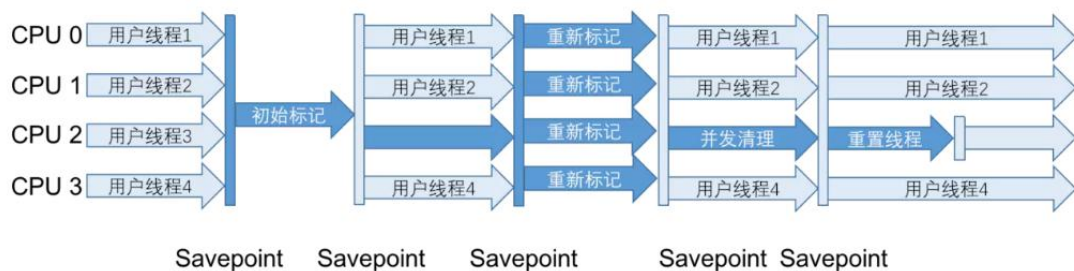
### 2.5.6 CMS 收集器

CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为目标的收集器。基于 标记 — 清除 算法实现。

运作步骤:

- 初始标记(CMS initial mark): 标记 GC Roots 能直接关联到的对象
- 并发标记(CMS concurrent mark): 进行 GC Roots Tracing
- 重新标记(CMS remark): 修正并发标记期间的变动部分
- 并发清除(CMS concurrent sweep)





Concurrent Mark Sweep 收集器运行示意图

缺点：对 CPU 资源敏感、无法收集浮动垃圾、标记 —— 清除 算法带来的空间碎片

### 2.5.7 G1 收集器

面向服务端的垃圾回收器。

优点：并行与并发、分代收集、空间整合、可预测停顿。

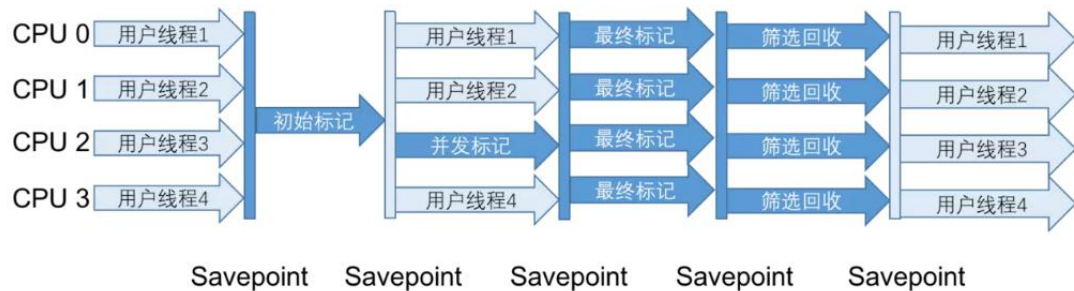
运作步骤：

初始标记(Initial Marking)

并发标记(Concurrent Marking)

最终标记(Final Marking)

筛选回收(Live Data Counting and Evacuation)



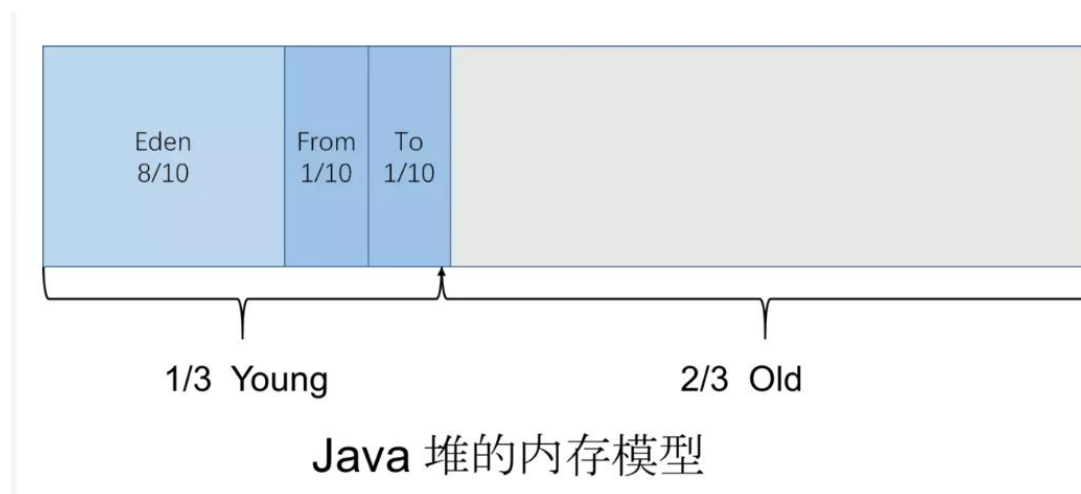
G1 收集器运行示意图

## 2.6 内存分配与回收策略

### 2.6.1 对象优先在 Eden 分配

对象主要分配在新生代的 Eden 区上，如果启动了本地线程分配缓冲区，将线程优先在 (TLAB) 上分配。少数情况会直接分配在老年代中。

一般来说 Java 堆的内存模型如下图所示：



新生代 GC (Minor GC)

发生在新生代的垃圾回收动作，频繁，速度快。

老年代 GC (Major GC / Full GC)

发生在老年代的垃圾回收动作，出现了 Major GC 经常会伴随至少一次 Minor GC(非绝对)。Major GC 的速度一般会比 Minor GC 慢十倍以上。

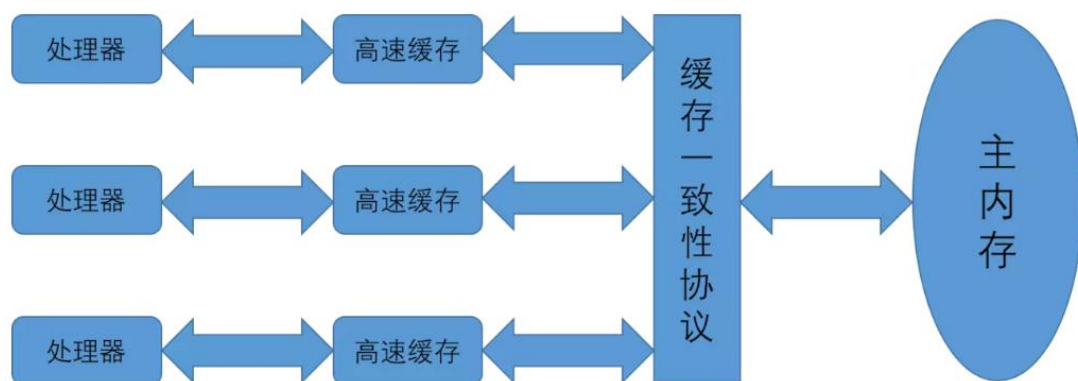
## 2.6.2 大对象直接进入老年代

### 2.6.3 长期存活的对象将进入老年代

### 2.6.4 动态对象年龄判定

### 2.6.5 空间分配担保

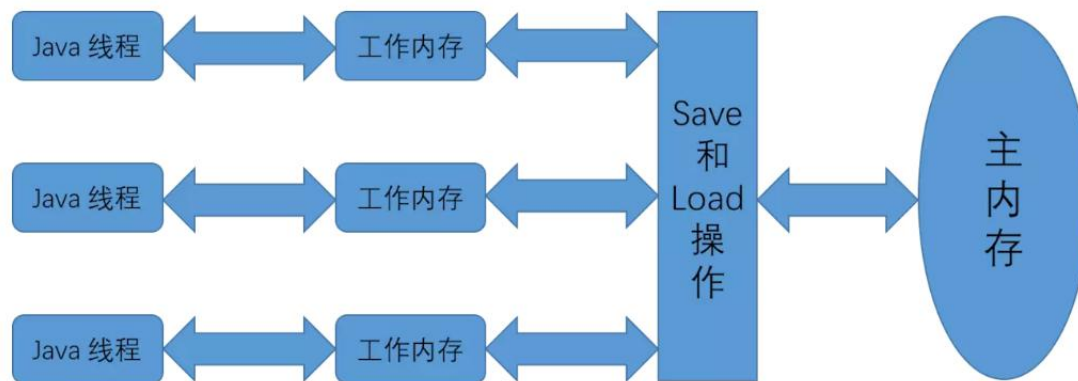
## 3. Java 内存模型与线程



处理器、高速缓存、主内存间的交互关系

## 3.1 Java 内存模型

屏蔽掉各种硬件和操作系统的内存访问差异。



## 线程、工作内存、主内存三者的交互关系

### 3.1.1 主内存和工作内存之间的交互

操作	作用对象	解释
lock	主内存	把一个变量标识为一条线程独占的状态
unlock	主内存	把一个处于锁定状态的变量释放出来，释放后才可被其他线程锁定
read	主内存	把一个变量的值从主内存传输到线程工作内存中，以便 load 操作使用
load	工作内存	把 read 操作从主内存中得到的变量值放入工作内存中
use	工作内存	把工作内存中一个变量的值传递给执行引擎， 每当虚拟机遇到一个需要用到变量值的字节码指令时将会执行这个操作
assign	工作内存	把一个从执行引擎接收到的值赋给工作内存的变量， 每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作
store	工作内存	把工作内存中的一个变量的值传送到主内存中，以便 write 操作
write	工作内存	把 store 操作从工作内存中得到的变量的值放入主内存的变量中

### 3.1.2 对于 volatile 型变量的特殊规则

关键字 `volatile` 是 Java 虚拟机提供的最轻量级的同步机制。

一个变量被定义为 `volatile` 的特性：

保证此变量对所有线程的可见性。但是操作并非原子操作，并发情况下不安全。

如果不符合 运算结果并不依赖变量当前值，或者能够确保只有单一的线程修改变量的值 和 变量不需要与其他的状态变量共同参与不变约束 就要通过加锁(使用 `synchronize` 或 `java.util.concurrent` 中的原子类)来保证原子性。

禁止指令重排序优化。

通过插入内存屏障保证一致性。

### 3.1.3 对于 long 和 double 型变量的特殊规则

Java 要求对于主内存和工作内存之间的八个操作都是原子性的，但是对于 64 位的数据类

型，有一条宽松的规定：允许虚拟机将没有被 `volatile` 修饰的 64 位数据的读写操作划分为两次 32 位的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 `load`、`store`、`read` 和 `write` 这 4 个操作的原子性。这就是 `long` 和 `double` 的非原子性协定。

### 3.1.4 原子性、可见性与有序性

回顾下并发下应该注意操作的那些特性是什么，同时加深理解。

#### 原子性(Atomicity)

由 Java 内存模型来直接保证的原子性变量操作包括 `read`、`load`、`assign`、`use`、`store` 和 `write`。大致可以认为基本数据类型的操作是原子性的。同时 `lock` 和 `unlock` 可以保证更大范围操作的原子性。而 `synchronize` 同步块操作的原子性是用更高层次的字节码指令 `monitorenter` 和 `monitorexit` 来隐式操作的。

#### 可见性(Visibility)

是指当一个线程修改了共享变量的值，其他线程也能够立即得知这个通知。主要操作细节就是修改值后将值同步至主内存(`volatile` 值使用前都会从主内存刷新)，除了 `volatile` 还有 `synchronize` 和 `final` 可以保证可见性。同步块的可见性是由“对一个变量执行 `unlock` 操作之前，必须先把此变量同步会主内存中(`store`、`write` 操作)”这条规则获得。而 `final` 可见性是指：被 `final` 修饰的字段在构造器中一旦完成，并且构造器没有把“`this`”的引用传递出去(`this` 引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象)，那在其他线程中就能看见 `final` 字段的值。

#### 有序性(Ordering)

如果在被线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句指“线程内表现为串行的语义”，后半句是指“指令重排”现象和“工作内存与主内存同步延迟”现象。Java 语言通过 `volatile` 和 `synchronize` 两个关键字来保证线程之间操作的有序性。`volatile` 自身就禁止指令重排，而 `synchronize` 则是由“一个变量在同一时刻指允许一条线程对其进行 `lock` 操作”这条规则获得，这条规则决定了持有同一个锁的两个同步块只能串行的进入。

### 3.1.5 先行发生原则

也就是 `happens-before` 原则。这个原则是判断数据是否存在竞争、线程是否安全的主要依据。先行发生是 Java 内存模型中定义的两项操作之间的偏序关系。

天然的先行发生关系

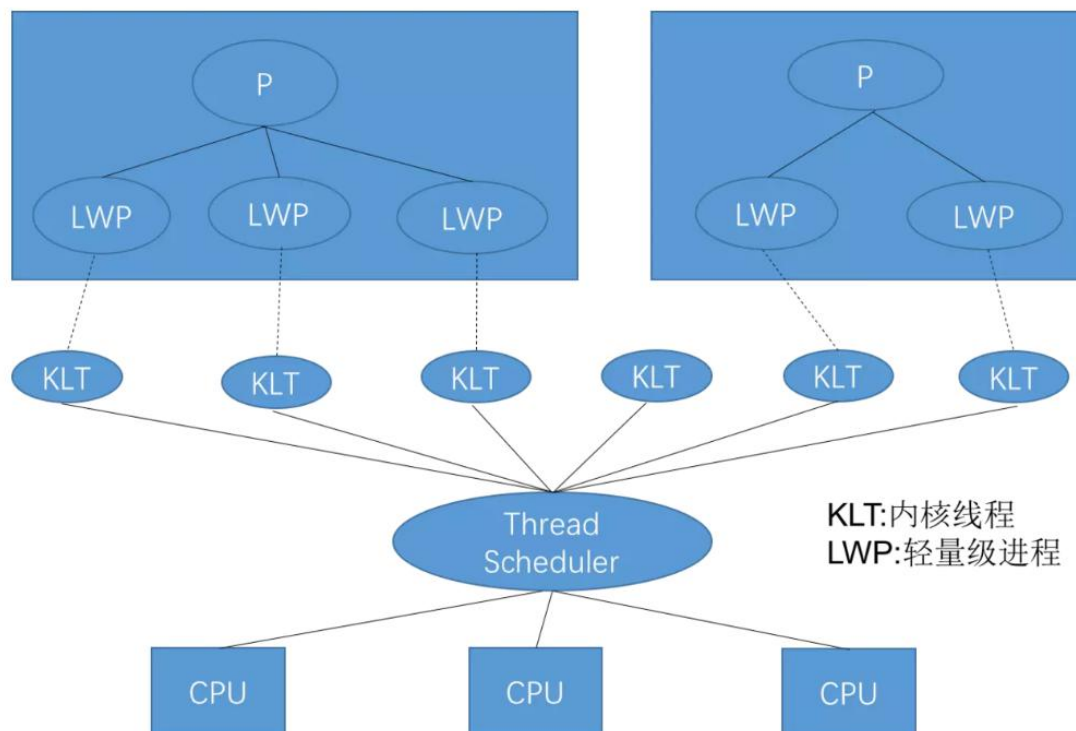
规则	解释
程序次序规则	在一个线程内，代码按照书写的控制流顺序执行
管程锁定规则	一个 unlock 操作先行发生于后面对同一个锁的 lock 操作
volatile 变量规则	volatile 变量的写操作先行发生于后面对这个变量的读操作
线程启动规则	Thread 对象的 start() 方法先行发生于此线程的每一个动作
线程终止规则	线程中所有的操作都先行发生于对此线程的终止检测 (通过 Thread.join() 方法结束、Thread.isAlive() 的返回值检测)
线程中断规则	对线程 interrupt() 方法调用优先发生于被中断线程的代码检测到中断事件的发生 (通过 Thread.interrupted() 方法检测)
对象终结规则	一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize() 方法的开始
传递性	如果操作 A 先于 操作 B 发生，操作 B 先于 操作 C 发生，那么操作 A 先于 操作 C

## 3.2 Java 与线程

### 3.2.1 线程的实现

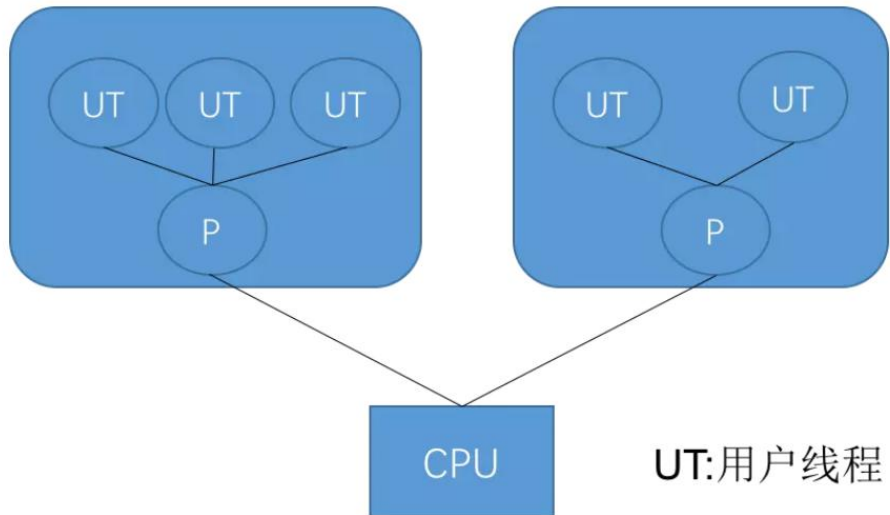
#### 使用内核线程实现

直接由操作系统内核支持的线程，这种线程由内核完成切换。程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口 —— 轻量级进程(LWP)，轻量级进程就是我们通常意义上所讲的线程，每个轻量级进程都有一个内核级线程支持。



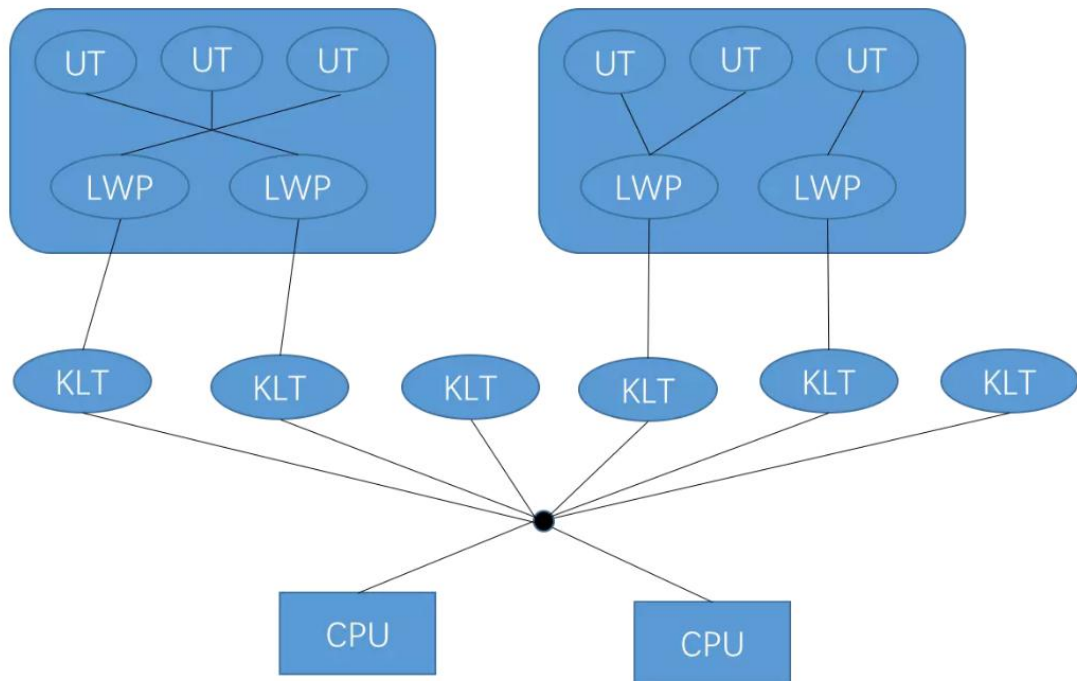
#### 使用用户线程实现

广义上来说，只要不是内核线程就可以认为是用户线程，因此可以认为轻量级进程也属于用户线程。狭义上说是完全建立在用户空间的线程库上的并且内核系统不可感知的。



使用用户线程夹加轻量级进程混合实现

[直接看图](#)



## Java 线程实现

平台不同实现方式不同，可以认为是一条 Java 线程映射到一条轻量级进程。

### 3.2.2 Java 线程调度

#### 协同式线程调度

线程执行时间由线程自身控制，实现简单，切换线程自己可知，所以基本没有线程同步问题。坏处是执行时间不可控，容易阻塞。

#### 抢占式线程调度

每个线程由系统来分配执行时间。

### 3.2.3 状态转换

五种状态：

#### 新建(new)

创建后尚未启动的线程。

#### 运行(Runnable)

Runnable 包括了操作系统线程状态中的 Running 和 Ready，也就是出于此状态的线程有可能正在执行，也有可能正在等待 CPU 为他分配时间。



### 无限期待(Waiting)

出于这种状态的线程不会被 CPU 分配时间，它们要等其他线程显示的唤醒。

以下方法会然线程进入无限期待状态：

- 1.没有设置 Timeout 参数的 Object.wait() 方法。
- 2.没有设置 Timeout 参数的 Thread.join() 方法。
- 3.LockSupport.park() 方法。

### 限期等待(Timed Waiting)

处于这种状态的线程也不会分配时间，不过无需等待配其他线程显示地唤醒，在一定时间后他们会由系统自动唤醒。

以下方法会让线程进入限期等待状态：

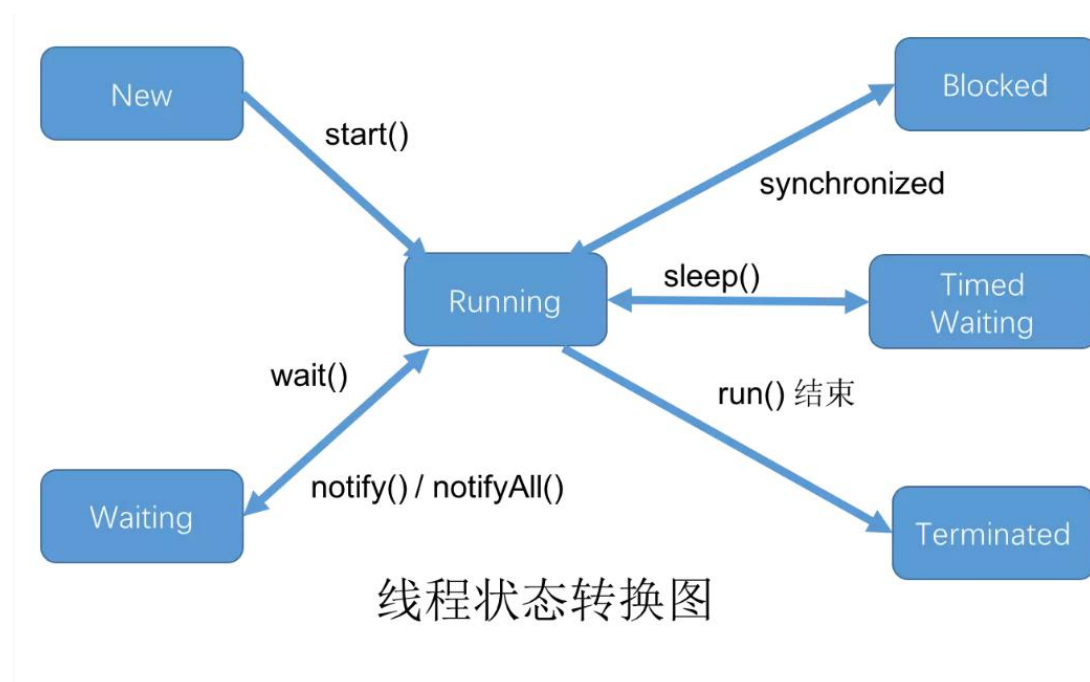
- 1.Thread.sleep() 方法。
- 2.设置了 Timeout 参数的 Object.wait() 方法。
- 3.设置了 Timeout 参数的 Thread.join() 方法。
- 4.LockSupport.parkNanos() 方法。
- 5.LockSupport.parkUntil() 方法。

### 阻塞(Blocked)

线程被阻塞了，“阻塞状态”和“等待状态”的区别是：“阻塞状态”在等待着获取一个排他锁，这个时间将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

### 结束(Terminated)

已终止线程的线程状态。



## 4. 线程安全与锁优化

// 待填

## 5. 类文件结构

// 待填

有点懒了。。。先贴几个网址吧。

### 1. Official: The class File Format

### 2. 亦山: 《Java 虚拟机原理图解》 1.1、class 文件基本组织结构

## 6. 虚拟机类加载机制

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、装换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型。

在 Java 语言中，类型的加载、连接和初始化过程都是在程序运行期间完成的。

### 6.1 类加载时机

类的生命周期(7 个阶段)



其中加载、验证、准备、初始化和卸载这五个阶段的顺序是确定的。解析阶段可以在初始化之后再开始(运行时绑定或动态绑定或晚期绑定)。

以下五种情况必须对类进行初始化(而加载、验证、准备自然需要在此之前完成):

遇到 `new`、`getstatic`、`putstatic` 或 `invokestatic` 这 4 条字节码指令时没初始化触发初始化。  
使用场景: 使用 `new` 关键字实例化对象、读取一个类的静态字段(被 `final` 修饰、已在编译期把结果放入常量池的静态字段除外)、调用一个类的静态方法。

使用 `java.lang.reflect` 包的方法对类进行反射调用的时候。

当初始化一个类的时候，如果发现其父类还没有进行初始化，则需先触发其父类的初始化。

当虚拟机启动时，用户需指定一个要加载的主类(包含 `main()` 方法的那个类)，虚拟机会先初始化这个主类。

当使用 JDK 1.7 的动态语言支持时，如果一个 `java.lang.invoke.MethodHandle` 实例最后的解析结果 `REF_getStatic`、`REF_putStatic`、`REF_invokeStatic` 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需先触发其初始化。

前面的五种方式是对一个类的主动引用，除此之外，所有引用类的方法都不会触发初始化，佳作被动引用。举几个例子~

```
public class SuperClass {
    static {
        System.out.println("SuperClass init!");
    }
    public static int value = 1127;
}

public class SubClass extends SuperClass {
    static {
        System.out.println("SubClass init!");
    }
}

public class ConstClass {
    static {
        System.out.println("ConstClass init!");
    }
    public static final String HELLOWORLD = "hello world!"
}

public class NotInitialization {
    public static void main(String[] args) {
        System.out.println(SubClass.value);
        /**
         *   output : SuperClass init!
         *
         *   通过子类引用父类的静态对象不会导致子类的初始化
         *   只有直接定义这个字段的类才会被初始化
         */

        SuperClass[] sca = new SuperClass[10];
        /**
         *   output :
         *
         *   通过数组定义来引用类不会触发此类的初始化
         *   虚拟机在运行时动态创建了一个数组类
         */
    }
}
```

```

        */

        System.out.println(ConstClass.HELLOWORLD);
    /**
     *   output :
     *
     *   常量在编译阶段会存入调用类的常量池当中，本质上并没有直接引用到定义类
    常量的类，
     *   因此不会触发定义常量的类的初始化。
     *   “hello world” 在编译期常量传播优化时已经存储到 NotInitialization 常量池中
    了。
     */
    }
}

```

## 6.2 类的加载过程

### 6.2.1 加载

通过一个类的全限定名来获取定义次类的二进制流(ZIP 包、网络、运算生成、JSP 生成、数据库读取)。

将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为方法去这个类的各种数据的访问入口。

数组类的特殊性：数组类本身不通过类加载器创建，它是由 Java 虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型最终是要靠类加载器去创建的，数组创建过程如下：

如果数组的组件类型是引用类型，那就递归采用类加载加载。

如果数组的组件类型不是引用类型，Java 虚拟机会把数组标记为引导类加载器关联。

数组类的可见性与他的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为 `public`。

内存中实例的 `java.lang.Class` 对象存在方法区中。作为程序访问方法区中这些类型数据的外部接口。

加载阶段与连接阶段的部分内容是交叉进行的，但是开始时间保持先后顺序。

### 6.2.2 验证

是连接的第一步，确保 `Class` 文件的字节流中包含的信息符合当前虚拟机要求。

文件格式验证

是否以魔数 `0xCAFEBABE` 开头

主、次版本号是否在当前虚拟机处理范围之内

常量池的常量是否有不被支持常量的类型（检查常量 `tag` 标志）

指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量

`CONSTANT_Utf8_info` 型的常量中是否有不符合 UTF8 编码的数据

Class 文件中各个部分集文件本身是否有被删除的附加的其他信息

.....

只有通过这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面 3 个验证阶段全部是基于方法区的存储结构进行的，不再直接操作字节流。

## 元数据验证

这个类是否有父类（除 `java.lang.Object` 之外）

这个类的父类是否继承了不允许被继承的类（`final` 修饰的类）

如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法

类中的字段、方法是否与父类产生矛盾（覆盖父类 `final` 字段、出现不符合规范的重载）

这一阶段主要是对类的元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息。

## 字节码验证

保证任意时刻操作数栈的数据类型与指令代码序列都配合工作（不会出现按照 `long` 类型读一个 `int` 型数据）

保证跳转指令不会跳转到方法体以外的字节码指令上

保证方法体中的类型转换是有效的（子类对象赋值给父类数据类型是安全的，反过来不合法的）

.....

这是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。这个阶段对类的方法体进行校验分析，保证校验类的方法在运行时不会做出危害虚拟机安全的事件。

## 符号引用验证

符号引用中通过字符创描述的全限定名是否能找到对应的类

在指定类中是否存在符方法的字段描述符以及简单名称所描述的方法和字段

符号引用中的类、字段、方法的访问性（`private`、`protected`、`public`、`default`）是否可被当前类访问

.....

最后一个阶段的校验发生在迅疾将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验，还有以上提及的内容。

符号引用的目的是确保解析动作能正常执行，如果无法通过符号引用验证将抛出一个 `java.lang.IncompatibleClass.ChangeError` 异常的子类。如 `java.lang.IllegalAccessError`、`java.lang.NoSuchFieldError`、`java.lang.NoSuchMethodError` 等。

### 6.2.3 准备

这个阶段正式为类分配内存并设置类变量初始值，内存存在方法去中分配(含 `static` 修饰的变量不含实例变量)。

```
public static int value = 1127;
```

这句代码在初始值设置之后为 `0`，因为这时候尚未开始执行任何 `Java` 方法。而把 `value` 赋值为 `1127` 的 `putstatic` 指令是程序被编译后，存放于 `clinit()` 方法中，所以初始化阶段才会对 `value` 进行赋值。

基本数据类型的零值

数据类型	零值	数据类型	零值
int	0	boolean	false
long	0L	float	0.0f
short	(short) 0	double	0.0d
char	'\u0000'	reference	null

特殊情况：如果类字段的字段属性表中存在 `ConstantValue` 属性，在准备阶段虚拟机就会根据 `ConstantValue` 的设置将 `value` 赋值为 `1127`。

6.2.4 解析

这个阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用

符号引用以一组符号来描述所引用的目标，符号可以使任何形式的字面量。

直接引用

直接引用可以使直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用和迅速的内存布局实现有关

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类符号引用进行，分别对应于常量池的 7 中常量类型。

6.2.5 初始化

前面过程都是以虚拟机主导，而初始化阶段开始执行类中的 `Java` 代码。

6.3 类加载器

通过一个类的全限定名来获取描述此类的二进制字节流。

6.3.1 双亲委派模型

从 `Java` 虚拟机角度讲，只存在两种类加载器：一种是启动类加载器（`C++` 实现，是虚拟机的一部分）；另一种是其他所有类的加载器（`Java` 实现，独立于虚拟机外部且全继承自 `java.lang.ClassLoader`）

启动类加载器

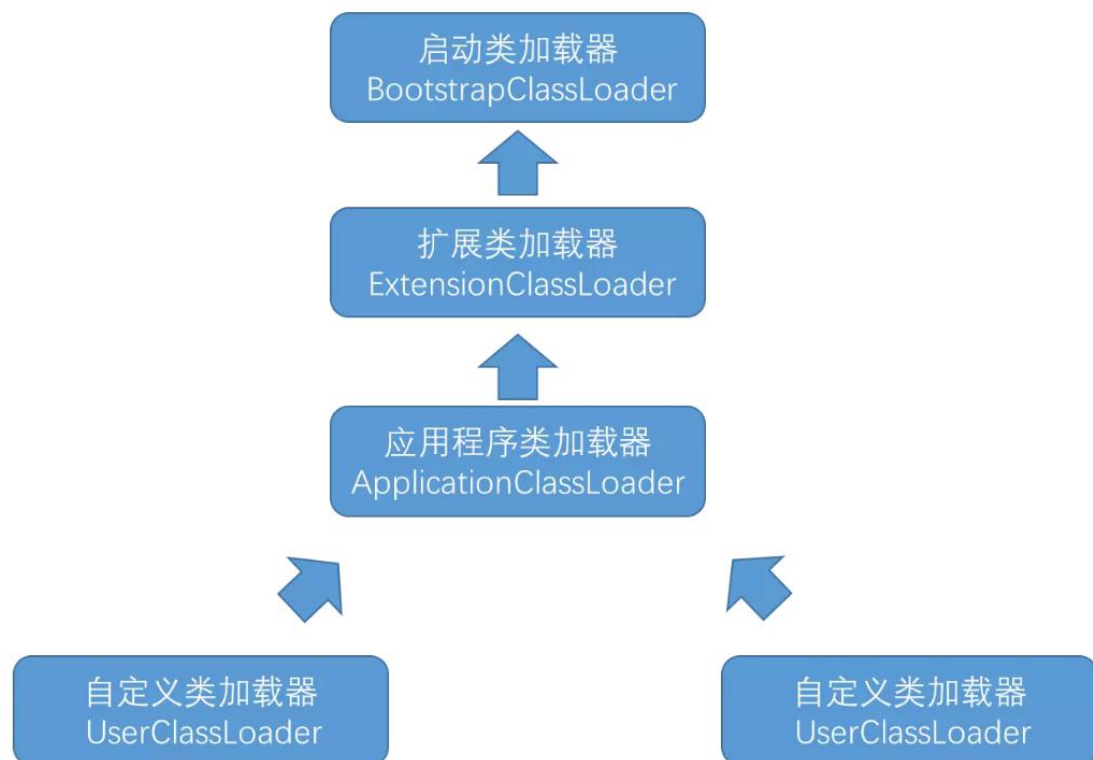
加载 lib 下或被 -Xbootclasspath 路径下的类

扩展类加载器

加载 lib/ext 或者被 java.ext.dirs 系统变量所指定的路径下的类

引用程序类加载器

ClassLoader 负责，加载用户路径上所指定的类库。



除顶层启动类加载器之外，其他都有自己的父类加载器。

工作过程：如果一个类加载器收到一个类加载的请求，它首先不会自己加载，而是把这个请求委派给父类加载器。只有父类无法完成时子类才会尝试加载。

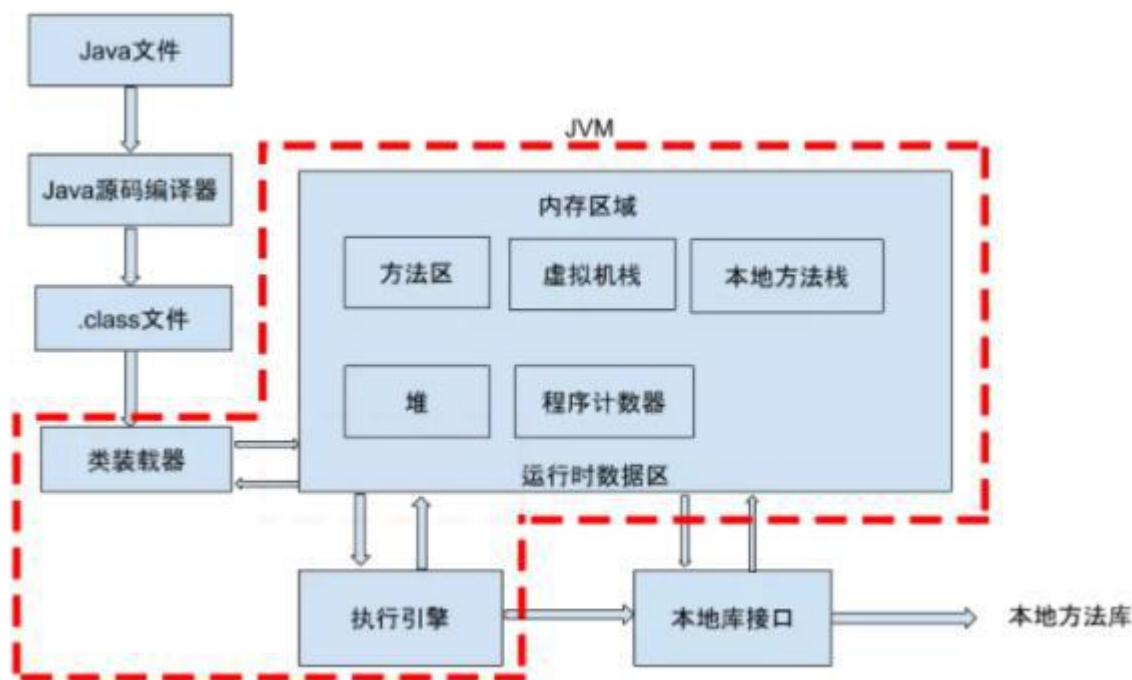
### 6.3.2 破坏双亲委派模型

keyword：线程上下文加载器(Thread Context ClassLoader)

## 8. 类加载过程

### 一、什么是类的加载

在介绍类的加载机制之前，先来看看，类的加载机制在整个 java 程序运行期间处于一个什么环节，下面使用一张图来表示：



从上图可以看，java 文件通过编译器变成了.class 文件，接下来类加载器又将这些.class 文件加载到 JVM 中。其中**类装载器**的作用其实就是类的加载。今天我们要讨论的就是这个环节。有了这个印象之后我们再来看**类的加载**的概念：

其实可以一句话来解释：类的加载指的是将类的.class 文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 `java.lang.Class` 对象，用来封装类在方法区内的数据结构。

到现在为止，我们基本上对类加载机制处于整个程序运行的环节位置，还有类加载机制的概念有了基本的印象。在类加载.class 文件之前，还有两个问题需要我们去弄清楚：

### 1、在什么时候才会启动类加载器？

其实，类加载器并不需要等到某个类被“首次主动使用”时再加载它，JVM 规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class 文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才



报告错误（**LinkageError** 错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。

## 2、从哪个地方去加载.class 文件

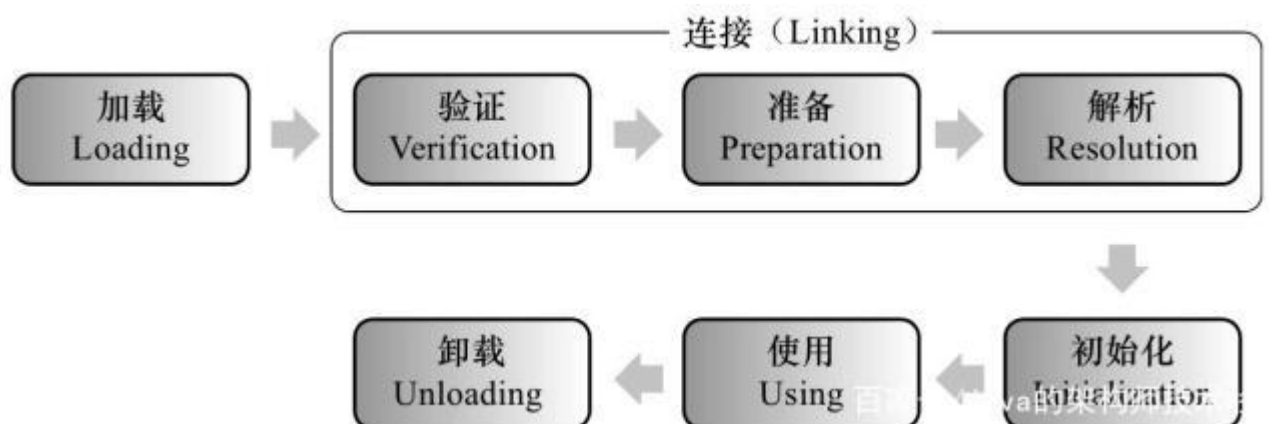
在这里进行一个简单的分类。例举了 5 个来源

- （1）本地磁盘
- （2）网上加载.class 文件（Applet）
- （3）从数据库中
- （4）压缩文件中（ZAR, jar 等）
- （5）从其他文件生成的（JSP 应用）

有了这个认识之后，下面就开始讲讲，类加载机制了。首先看的就是类加载机制的过程。

## 二、类加载的过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。它们的顺序如下图所示：



其中**类加载的过程**包括了**加载、验证、准备、解析、初始化**五个阶段。

在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始。另外注意这里的几个阶段是按顺序**开始**，而不是按顺序**进行**或**完成**，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。

下面就一个一个去分析一下这几个过程。

## 1、加载

“加载”是“类加载机制”的第一个过程，在加载阶段，虚拟机主要完成三件事：

- (1) 通过一个类的全限定名来获取其定义的二进制字节流
- (2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
- (3) 在堆中生成一个代表这个类的 **Class** 对象，作为方法区中这些数据的访问入口。

相对于类加载的其他阶段而言，加载阶段是可控性最强的阶段，因为程序员可以使用系统的类加载器加载，还可以使用自己的类加载器加载。我们在最后一部分会详细介绍这个类加载器。在这里我们只需要知道类加载器的作用就是上面虚拟机需要完成的三件事，仅此而已就好了。

## 2、验证

验证的主要作用就是确保被加载的类的正确性。也是连接阶段的第一步。说白了也就是我们加载好的 **.class** 文件不能对我们的虚拟机有危害，所以先检测验证一下。他主要是完成四个阶段的验证：

- (1) 文件格式的验证：验证 **.class** 文件字节流是否符合 **class** 文件的格式的规范，并且能够被当前版本的虚拟机处理。这里面主要对魔数、主版本号、常量池等等的校验（魔数、主版本号都是 **.class** 文件里面包含的数据信息、在这里可以不用理解）。

(2) 元数据验证：主要是对字节码描述的信息进行语义分析，以保证其描述的信息符合 **java** 语言规范的要求，比如说验证这个类是不是有父类，类中的字段方法是不是和父类冲突等等。

(3) 字节码验证：这是整个验证过程最复杂的阶段，主要是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在元数据验证阶段对数据类型做出验证后，这个阶段主要对类的方法做出分析，保证类的方法在运行时不会做出威胁虚拟机安全的事。

(4) 符号引用验证：它是验证的最后一个阶段，发生在虚拟机将符号引用转化为直接引用的时候。主要是对类自身以外的信息进行校验。目的是确保解析动作能够完成。

对整个类加载机制而言，验证阶段是一个很重要但是非必需的阶段，如果我们的代码能够确保没有问题，那么我们就没有必要去验证，毕竟验证需要花费一定的时间。当然我们可以使用 **-Xverify:none** 来关闭大部分的验证。

### 3、准备

**准备阶段主要为类变量分配内存并设置初始值。**这些内存都在方法区分配。

在这个阶段我们只需要注意两点就好了，也就是类变量和初始值两个关键词：

(1) 类变量 (**static**) 会分配内存，但是实例变量不会，实例变量主要随着对象的实例化一块分配到 **java** 堆中，

(2) 这里的初始值指的是数据类型默认值，而不是代码中被显示赋予的值。比如

```
public static int value = 1; //在这里准备阶段过后的 value 值为 0，而不是 1。赋值为 1 的动作在初始化阶段。
```

当然还有其他的默认值。

数据类型	默认值	数据类型	默认值
int	0	boolean	false
long	0L	float	0.0f
short	0(short)	double	0.0d

注意，在上面 **value** 是被 **static** 所修饰的准备阶段之后是 **0**，但是如果同时被 **final** 和 **static** 修饰准备阶段之后就是 **1** 了。我们可以理解为 **static final** 在编译器就将结果放入调用它的类的常量池中了。

## 4、解析

解析阶段主要是虚拟机将常量池中的符号引用转化为直接引用的过程。什么是符号应用和直接引用呢？

符号引用：以一组符号来描述所引用的目标，可以是任何形式的字面量，只要是能无歧义的定位到目标就好，就好比在班级中，老师可以用张三来代表你，也可以用你的学号来代表你，但无论任何方式这些都只是一个代号（符号），这个代号指向你（符号引用）直接引用：直接引用是可以指向目标的指针、相对偏移量或者是一个能直接或间接定位到目标的句柄。和虚拟机实现的内存有关，不同的虚拟机直接引用一般不同。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类符号引用进行。

## 5、初始化

这是类加载机制的最后一步，在这个阶段，**java** 程序代码才开始真正执行。我们知道，在准备阶段已经为类变量赋过一次值。在初始化阶段，程序员可以根据自己的需求来赋值了。一句话描述这个阶段就是执行类构造器 `<clinit>()` 方法的过程。

在初始化阶段，主要为类的静态变量赋予正确的初始值，JVM 负责对类进行初始化，主要对类变量进行初始化。在 Java 中对类变量进行初始值设定有两种方式：

①声明类变量是指定初始值

②使用静态代码块为类变量指定初始值

### JVM 初始化步骤

- 1、假如这个类还没有被加载和连接，则程序先加载并连接该类
- 2、假如该类的直接父类还没有被初始化，则先初始化其直接父类
- 3、假如类中有初始化语句，则系统依次执行这些初始化语句

类初始化时机：只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：

创建类的实例，也就是 **new** 的方式访问某个类或接口的静态变量，或者对该静态变量赋值调用类的静态方法反射（如 **Class.forName("com.shengsiyuan.Test")**）初始化某个类的子类，则其父类也会被初始化 Java 虚拟机启动时被标明为启动类的类（**JavaTest**），直接使用 **java.exe** 命令来运行某个主类好了，到目前为止就是类加载机制的整个过程，但是还有一个重要的概念，那就是类加载器。在加载阶段其实我们提到过类加载器，说是在后面详细说，在这就好好地介绍一下类加载器。

## 三、类加载器

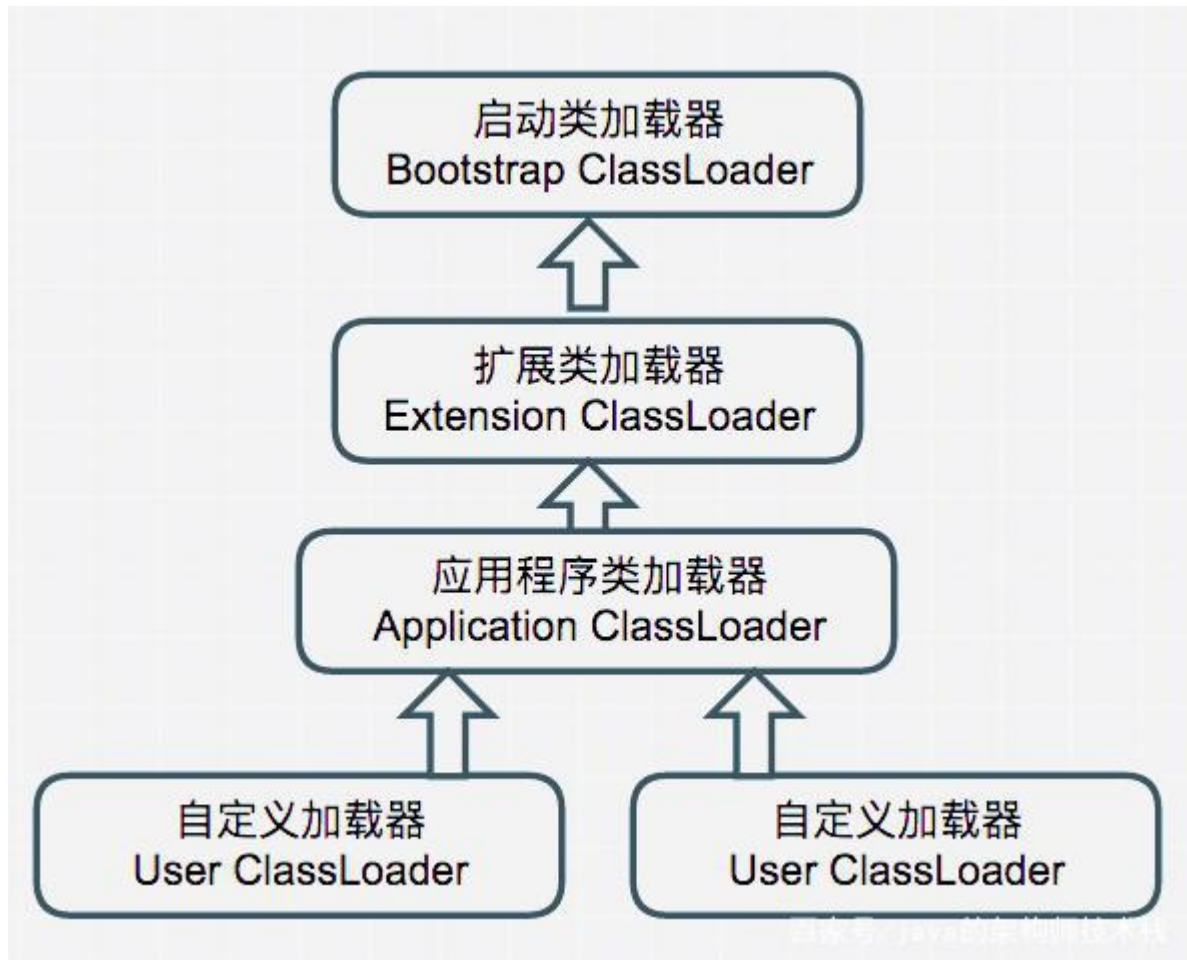
虚拟机设计团队把加载动作放到 JVM 外部实现，以便让应用程序决定如何获取所需的类。

### 1、Java 语言系统自带三个类加载器：

**Bootstrap ClassLoader**：最顶层的加载类，主要加载核心类库，也就是我们环境变量下面%JRE\_HOME%\lib 下的 rt.jar、resources.jar、charsets.jar 和 class 等。另外需要注意的是可以通过启动 jvm 时指定 -Xbootclasspath 和路径来改变 Bootstrap ClassLoader 的加载目录。比如 java -Xbootclasspath/a:path 被指定的文件追加到默认的 bootstrap 路径中。我们可以打开我的电脑，在上面的目录下查看，看看这些 jar 包是不是存在于这个目录。**Extention ClassLoader**：扩展的类加载器，加载目录%JRE\_HOME%\lib\ext 目录下的 jar 包和 class 文件。还可以加载 -D java.ext.dirs 选项指定的目录。**Appclass Loader**：也称为 **SystemAppClass**。加载当前应用的 classpath 的所有类。我们看到 java 为我们提供了三个类加载器，应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，我们还可以加入自定义的类加载器。这三种类加载器的加载顺序是什么呢？

## **Bootstrap ClassLoader > Extention ClassLoader > Appclass Loader**

一张图来看一下他们的层次关系



代码验证一下：

```
2* * @ClassName: ClassLoaderTest
7 package com.dff;
8
9 public class ClassLoaderTest {
10     public static void main(String[] args) {
11         ClassLoader loader = Thread.currentThread().getContextClassLoader();
12         1-> System.out.println(loader);
13         2-> System.out.println(loader.getParent());
14         3-> System.out.println(loader.getParent().getParent());
15     }
16 }
```

Problems JavaDoc Declaration Console

<terminated> ClassLoaderTest [Java Application] C:\Program Files\Java\jdk1.8.0\_171\bin\javaw.exe (2019年6月14日 下午5:05:59)

sun.misc.Launcher\$AppClassLoader@2a139a55 <-1

sun.misc.Launcher\$ExtClassLoader@7852e922 <-2

null <-3

从上面的结果可以看出，并没有获取到 `ExtClassLoader` 的父 `Loader`，原因是 `Bootstrap Loader`（引导类加载器）是用 C 语言实现的，找不到一个确定的返回父 `Loader` 的方式，于是就返回 `null`。

## 2、类加载的三种方式

认识了这三种类加载器，接下来我们看看类加载的三种方式。

（1）通过命令行启动应用时由 JVM 初始化加载含有 `main()` 方法的主类。

（2）通过 `Class.forName()` 方法动态加载，会默认执行初始化块（`static{}`），但是 `Class.forName(name, initialize, loader)` 中的 `initialize` 可指定是否要执行初始化块。

（3）通过 `ClassLoader.loadClass()` 方法动态加载，不会执行初始化块。

下面代码来演示一下

首先我们定义一个 `FDD` 类

```
public class FDD {static { System.out.println("我是静态代码块。。。。"); }}
```

然后我们看一下如何去加载

```
package com.fdd.test;public class FDDloaderTest { public static void  
main(String[] args) throws ClassNotFoundException { ClassLoader  
loader =  
HelloWorld.class.getClassLoader(); System.out.println(loader);// 一、  
使用 ClassLoader.loadClass() 来加载类，不会执行初始化块  
loader.loadClass("Fdd"); //二、 使用 Class.forName()来加载类，默认  
会执行初始化块 Class.forName("Fdd"); //三、使用 Class.forName()来  
加载类，指定 ClassLoader，初始化时不执行静态  
块 Class.forName("Fdd", false, loader); } }
```



上面是同不同的方式去加载类，结果是不一样的。

### 3、双亲委派原则

他的工作流程是： 当一个类加载器收到类加载任务，会先交给其父类加载器去完成，因此最终加载任务都会传递到顶层的启动类加载器，只有当父类加载器无法完成加载任务时，才会尝试执行加载任务。这个理解起来就简单了，比如说，另外一个人给小费，自己不会先去直接拿来塞自己钱包，我们先把钱给领导，领导再给领导，一直到公司老板，老板不想要了，再一级一级往下分。老板要是这个钱，下面的领导和自己就一分钱没有了。（例子不好，理解就好）

采用双亲委派的一个好处是比如加载位于 `rt.jar` 包中的类 `java.lang.Object`，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个 `Object` 对象。双亲委派原则归纳一下就是：

可以避免重复加载，父类已经加载了，子类就不需要再次加载更加安全，很好的解决了各个类加载器的基础类的统一问题，如果不使用该种方式，那么用户可以随意定义类加载器来加载核心 `api`，会带来相关隐患。

### 4、自定义类加载器

在这一部分第一小节中，我们提到了 `java` 系统为我们提供的三种类加载器，还给出了他们的层次关系图，最下面就是自定义类加载器，那么我们如何自己定义类加载器呢？这主要有两种方式

（1）遵守双亲委派模型：继承 `ClassLoader`，重写 `findClass()` 方法。

（2）破坏双亲委派模型：继承 `ClassLoader`, 重写 `loadClass()` 方法。 通常我们推荐采用第一种方法自定义类加载器，最大程度上的遵守双亲委派模型。

我们看一下实现步骤

（1）创建一个类继承 `ClassLoader` 抽象类

（2）重写 `findClass()` 方法

(3) 在 findClass()方法中调用 defineClass()

代码实现一下：

```
public class MyClassLoader extends ClassLoader {private String
libPath; public DiskClassLoader(String path) { libPath =
path; }@Override protected Class<?> findClass(String name) throws
ClassNotFoundException { String fileName =
getFileName(name); File file = new File(libPath,fileName); try
{ FileInputStream is = new
FileInputStream(file); ByteArrayOutputStream bos = new
ByteArrayOutputStream(); int len = 0; try { while ((len = is.read()) != -1)
{ bos.write(len); } } catch (IOException e) { e.printStackTrace(); } byte[]
data = bos.toByteArray(); is.close(); bos.close(); return
defineClass(name,data,0,data.length); } catch (IOException e)
{e.printStackTrace(); } return super.findClass(name); } //获取要加载
的 class 文件名 private String getFileName(String name) { int index =
name.lastIndexOf('.'); if(index == -1){ return
name+".class"; }else{return name.substring(index+1)+".class"; } }}
```

接下来我们就可以自己去加载类了，使用方法大体就是两行

```
MyClassLoader diskLoader = new MyClassLoader("D:\\lib");//加载
class 文件，注意是 com.fdd.TestClass c =
diskLoader.loadClass("com.fdd.Test");
```

## 9. 反射

定义

[JAVA 反射机制](#)是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 **java** 语言的反射机制。

用途

在日常的第三方应用开发过程中，经常会遇到某个类的某个成员变量、方法或是属性是私有的或是只对系统应用开放，这时候就可以利用 **Java** 的反射机制通过反射来获取所需的私有成员或是方法。当然，也不是所有的都适合反射，之前就遇到一个案例，通过反射得到的结果与预期不符。阅读源码发现，经过层层调用后在最终返回结果的地方对应用的权限进行了校验，对于没有权限的应用返回值是没有意义的缺省值，否则返回实际值起到保护用户的隐私目的。

## 反射机制的相关类

与 **Java** 反射相关的类如下：

类名	用途
Class类	代表类的实体，在运行的Java应用程序中表示类和接口
Field类	代表类的成员变量（成员变量也称为类的属性）
Method类	代表类的方法
Constructor类	代表类的构造方法

### Class 类

**Class** 代表类的实体，在运行的 **Java** 应用程序中表示类和接口。在这个类中提供了很多有用的方法，这里对他们简单的分类介绍。

- 获得类相关的方法

方法	用途
asSubclass(Class<U> clazz)	把传递的类的对象转换成代表其子类的对象
Cast	把对象转换成代表类或是接口的对象
getClassLoader()	获得类的加载器
getClasses()	返回一个数组，数组中包含该类中所有公共类和接口类的对象
getDeclaredClasses()	返回一个数组，数组中包含该类中所有类和接口类的对象
forName(String className)	根据类名返回类的对象
getName()	获得类的完整路径名字
newInstance()	创建类的实例
getPackage()	获得类的包
getSimpleName()	获得类的名字
getSuperclass()	获得当前类继承的父类的名字
getInterfaces()	获得当前类实现的类或是接口

- 获得类中属性相关的方法

方法	用途
getField(String name)	获得某个公有的属性对象
getFields()	获得所有公有的属性对象
getDeclaredField(String name)	获得某个属性对象
getDeclaredFields()	获得所有属性对象

- 获得类中注解相关的方法

方法	用途
getAnnotation(Class<A> annotationClass)	返回该类中与参数类型匹配的公有注解对象
getAnnotations()	返回该类所有的公有注解对象
getDeclaredAnnotation(Class<A> annotationClass)	返回该类中与参数类型匹配的所有注解对象
getDeclaredAnnotations()	返回该类所有的注解对象

- 获得类中构造器相关的方法

方法	用途
<code>getConstructor(Class...&lt;?&gt; parameterTypes)</code>	获得该类中与参数类型匹配的公有构造方法
<code>getConstructors()</code>	获得该类的所有公有构造方法
<code>getDeclaredConstructor(Class...&lt;?&gt; parameterTypes)</code>	获得该类中与参数类型匹配的构造方法
<code>getDeclaredConstructors()</code>	获得该类所有构造方法

- 获得类中方法相关的方法

方法	用途
<code>getMethod(String name, Class...&lt;?&gt; parameterTypes)</code>	获得该类某个公有的方法
<code>getMethods()</code>	获得该类所有公有的方法
<code>getDeclaredMethod(String name, Class...&lt;?&gt; parameterTypes)</code>	获得该类某个方法
<code>getDeclaredMethods()</code>	获得该类所有方法

- 类中其他重要的方法

方法	用途
<code>isAnnotation()</code>	如果是注解类型则返回true
<code>isAnnotationPresent(Class&lt;? extends Annotation&gt; annotationClass)</code>	如果是指定类型注解类型则返回true
<code>isAnonymousClass()</code>	如果是匿名类则返回true
<code>isArray()</code>	如果是一个数组类则返回true
<code>isEnum()</code>	如果是枚举类则返回true
<code>isInstance(Object obj)</code>	如果obj是该类的实例则返回true
<code>isInterface()</code>	如果是接口类则返回true
<code>isLocalClass()</code>	如果是局部类则返回true
<code>isMemberClass()</code>	如果是内部类则返回true

## Field 类

Field 代表类的成员变量（成员变量也称为类的属性）。

方法	用途
equals(Object obj)	属性与obj相等则返回true
get(Object obj)	获得obj中对应的属性值
set(Object obj, Object value)	设置obj中对应属性值

## Method 类

Method 代表类的方法。

方法	用途
invoke(Object obj, Object... args)	传递object对象及参数调用该对象对应的方法

## Constructor 类

Constructor 代表类的构造方法。

方法	用途
newInstance(Object... initargs)	根据传递的参数创建类的对象

## 示例

为了演示反射的使用，首先构造一个与书籍相关的 model——Book.java，然后通过反射方法示例创建对象、反射私有构造方法、反射私有属性、反射私有方法，最后给出两个比较复杂的反射示例——获得当前 ZenMode 和关机 Shutdown。

- 被反射类 **Book.java**

```
public class Book{
    private final static String TAG = "BookTag";

    private String name;
```

```

private String author;

@Override
public String toString() {
    return "Book{" +
        "name='" + name + '\'' +
        ", author='" + author + '\'' +
        '}';
}

public Book() {
}

private Book(String name, String author) {
    this.name = name;
    this.author = author;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}

private String declaredMethod(int index) {
    String string = null;
    switch (index) {
        case 0:
            string = "I am declaredMethod 1 !";
            break;
        case 1:
            string = "I am declaredMethod 2 !";
            break;
        default:
    }
}

```

```

        string = "I am declaredMethod 1 !";
    }

    return string;
}}

```

- 反射逻辑封装在 **ReflectClass.java**

```

public class ReflectClass {
    private final static String TAG = "peter.log.ReflectClass";

    // 创建对象
    public static void reflectNewInstance() {
        try {
            Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
            Object objectBook = classBook.newInstance();
            Book book = (Book) objectBook;
            book.setName("Android 进阶之光");
            book.setAuthor("刘望舒");
            Log.d(TAG, "reflectNewInstance book = " + book.toString());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 反射私有的构造方法
    public static void reflectPrivateConstructor() {
        try {
            Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
            Constructor<?> declaredConstructorBook =
classBook.getDeclaredConstructor(String.class, String.class);
            declaredConstructorBook.setAccessible(true);
            Object objectBook =
declaredConstructorBook.newInstance("Android 开发艺术探索", "任玉刚");
            Book book = (Book) objectBook;
            Log.d(TAG, "reflectPrivateConstructor book = " +
book.toString());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```



```

// 反射私有属性
public static void reflectPrivateField() {
    try {
        Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
        Object objectBook = classBook.newInstance();
        Field fieldTag = classBook.getDeclaredField("TAG");
        fieldTag.setAccessible(true);
        String tag = (String) fieldTag.get(objectBook);
        Log.d(TAG, "reflectPrivateField tag = " + tag);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

// 反射私有方法
public static void reflectPrivateMethod() {
    try {
        Class<?> classBook =
Class.forName("com.android.peter.reflectdemo.Book");
        Method methodBook =
classBook.getDeclaredMethod("declaredMethod", int.class);
        methodBook.setAccessible(true);
        Object objectBook = classBook.newInstance();
        String string = (String) methodBook.invoke(objectBook, 0);

        Log.d(TAG, "reflectPrivateMethod string = " + string);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

// 获得系统 Zenmode 值
public static int getZenMode() {
    int zenMode = -1;
    try {
        Class<?> cServiceManager =
Class.forName("android.os.ServiceManager");
        Method mGetService = cServiceManager.getMethod("getService",
String.class);
        Object oNotificationManagerService = mGetService.invoke(null,
Context.NOTIFICATION_SERVICE);
        Class<?> cINotificationManagerStub =
Class.forName("android.app.INotificationManager$Stub");

```

```

        Method mAsInterface =
cINotificationManagerStub.getMethod("asInterface", IBinder.class);
        Object oINotificationManager =
mAsInterface.invoke(null, oNotificationManagerService);
        Method mGetZenMode =
cINotificationManagerStub.getMethod("getZenMode");
        zenMode = (int) mGetZenMode.invoke(oINotificationManager);
    } catch (Exception ex) {
        ex.printStackTrace();
    }

    return zenMode;
}

// 关闭手机
public static void shutDown() {
    try {
        Class<?> cServiceManager =
Class.forName("android.os.ServiceManager");
        Method mGetService =
cServiceManager.getMethod("getService", String.class);
        Object oPowerManagerService =
mGetService.invoke(null, Context.POWER_SERVICE);
        Class<?> cIPowerManagerStub =
Class.forName("android.os.IPowerManager$Stub");
        Method mShutdown =
cIPowerManagerStub.getMethod("shutdown", boolean.class, String.class, bo
olean.class);
        Method mAsInterface =
cIPowerManagerStub.getMethod("asInterface", IBinder.class);
        Object oIPowerManager =
mAsInterface.invoke(null, oPowerManagerService);
        mShutdown.invoke(oIPowerManager, true, null, true);

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public static void shutdownOrReboot(final boolean shutdown, final
boolean confirm) {
    try {
        Class<?> ServiceManager =
Class.forName("android.os.ServiceManager");

```

```

        // 获得 ServiceManager 的 getService 方法
        Method getService = ServiceManager.getMethod("getService",
java.lang.String.class);
        // 调用 getService 获取 RemoteService
        Object oRemoteService = getService.invoke(null,
Context.POWER_SERVICE);
        // 获得 IPowerManager.Stub 类
        Class<?> cStub =
Class.forName("android.os.IPowerManager$Stub");
        // 获得 asInterface 方法
        Method asInterface = cStub.getMethod("asInterface",
android.os.IBinder.class);
        // 调用 asInterface 方法获取 IPowerManager 对象
        Object oIPowerManager = asInterface.invoke(null,
oRemoteService);
        if (shutdown) {
            // 获得 shutdown() 方法
            Method shutdownMethod =
oIPowerManager.getClass().getMethod(
                "shutdown", boolean.class, String.class,
boolean.class);
            // 调用 shutdown() 方法
            shutdownMethod.invoke(oIPowerManager, confirm, null,
false);
        } else {
            // 获得 reboot() 方法
            Method rebootMethod =
oIPowerManager.getClass().getMethod("reboot",
                boolean.class, String.class, boolean.class);
            // 调用 reboot() 方法
            rebootMethod.invoke(oIPowerManager, confirm, null,
false);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}}

```

- 调用相应反射逻辑方法

```

try {
    // 创建对象
    ReflectClass.reflectNewInstance();
}

```

```

// 反射私有的构造方法
ReflectClass.reflectPrivateConstructor();

// 反射私有属性
ReflectClass.reflectPrivateField();

// 反射私有方法
ReflectClass.reflectPrivateMethod();
} catch (Exception ex) {
    ex.printStackTrace();
}

Log.d(TAG, " zenmode = " + ReflectClass.getZenMode());

```

Log 输出结果如下:

```

08-27 15:11:37.999 11987-11987/com.android.peter.reflectdemo
D/peter.log.ReflectClass: reflectNewInstance book = Book{name=' Android
进阶之光', author=' 刘望舒' }08-27 15:11:38.000
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectClass:
reflectPrivateConstructor book = Book{name=' Android 开发艺术探索',
author=' 任玉刚' }08-27 15:11:38.000
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectClass:
reflectPrivateField tag = BookTag08-27 15:11:38.000
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectClass:
reflectPrivateMethod String = I am declaredMethod 1 !08-27 15:11:38.004
11987-11987/com.android.peter.reflectdemo D/peter.log.ReflectDemo:
zenmode = 0

```

## 总结

本文列举了反射机制使用过程中常用的、重要的一些类及其方法，更多信息和用法需要进一步的阅读 Google 提供的相关文档和示例。

在阅读 Class 类文档时发现一个特点，以通过反射获得 Method 对象为例，一般会提供四种方法，getMethod(parameterTypes)、getMethods()、getDeclaredMethod(parameterTypes)和 getDeclaredMethods()。getMethod(parameterTypes)用来获取某个公有的方法的对象，getMethods()获得该类所有公有的方法，getDeclaredMethod(parameterTypes)获得该类某个方法，getDeclaredMethods()获得该类所有方法。带有 **Declared** 修饰的方法可以反射到私有的方法，没有 **Declared** 修饰的只能用来反射公有的方法。其他的 Annotation、Field、Constructor 也是如此。

在 `ReflectClass` 类中还提供了两种反射 `PowerManager.shutdown()` 的方法，在调用的时候会输出如下 log，提示没有相关权限。之前在项目中尝试反射其他方法的时候还遇到过有权限和没权限返回的值不一样的情况。如果源码中明确进行了权限验证，而你的应用又无法获得这个权限的话，建议就不要浪费时间反射了。

```
W/System.err: java.lang.reflect.InvocationTargetException
W/System.err:     at java.lang.reflect.Method.invoke(Native Method)
W/System.err:     at .ReflectClass.shutdown(ReflectClass.java:104)
W/System.err:     at .MainActivity$1.onClick(MainActivity.java:25)
W/System.err:     at android.view.View.performClick(View.java:6259)
W/System.err:     at
android.view.View$PerformClick.run(View.java:24732)
W/System.err:     at
android.os.Handler.handleCallback(Handler.java:789)
W/System.err:     at
android.os.Handler.dispatchMessage(Handler.java:98)
W/System.err:     at android.os.Looper.loop(Looper.java:164)
W/System.err:     at
android.app.ActivityThread.main(ActivityThread.java:6592)
W/System.err:     at java.lang.reflect.Method.invoke(Native Method)
W/System.err:     at
com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:240)
W/System.err:     at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:769)
W/System.err: Caused by: java.lang.SecurityException: Neither user
10224 nor current process has android.permission.REBOOT.
W/System.err:     at
android.os.Parcel.readException(Parcel.java:1942)
W/System.err:     at
android.os.Parcel.readException(Parcel.java:1888)
W/System.err:     at
android.os.IPowerManager$Stub$Proxy.shutdown(IPowerManager.java:787)
W/System.err: ... 12 more
```

## 10. 多线程和线程池

(1) 无论 `synchronized` 关键字加在方法上还是对象上，他取得的锁都是对象，而不是把一段代码或函数当作锁，而且同步方法很可能还会被其他线程的

对象访问。

( 2 ) 每个对象只有一个锁 ( lock ) 和之相关联。

( 3 ) 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

说下 java 中的线程创建方式，线程池的工作原理。

java 中有三种创建线程的方式，或者说四种

- 1.继承 Thread 类实现多线程
- 2.实现 Runnable 接口
- 3.实现 Callable 接口
- 4.通过线程池

线程池的工作原理：线程池可以减少创建和销毁线程的次数，从而减少系统资源的消耗，当一个任务提交到线程池时

- a. 首先判断核心线程池中的线程是否已经满了，如果没满，则创建一个核心线程执行任务，否则进入下一步
- b. 判断工作队列是否已满，没有满则加入工作队列，否则执行下一步
- c. 判断线程数是否达到了最大值，如果不是，则创建非核心线程执行任务，否则执行饱和策略，默认抛出异常

## 11. HTTP、HTTPS、TCP/IP、Socket 通信、三次握手四次挥手过程

- 1.ARP 协议:在 IP 以太网中，当一个上层协议要发包时，有了该节点的 IP 地址，ARP 就能提供该节点的 MAC 地址。
- 2.HTTP HTTPS 的区别:
  - 1.HTTPS 使用 TLS(SSL)进行加密
  - 2.HTTPS 缺省工作在 TCP 协议 443 端口
  - 3.它的工作流程一般如以下方式:

- 1.完成 TCP 三次同步握手
  - 2.客户端验证服务器数字证书，通过，进入步骤 3
  - 3.DH 算法协商对称加密算法的密钥、hash 算法的密钥
  - 4.SSL 安全加密隧道协商完成
  - 5.网页以加密的方式传输，用协商的对称加密算法和密钥加密，保证数据机密性；用协商的 hash 算法进行数据完整性保护，保证数据不被篡改
- 3.http 请求包结构，http 返回码的分类，400 和 500 的区别
- 1.包结构：
    - 1.请求：请求行、头部、数据
    - 2.返回：状态行、头部、数据
  - 2.http 返回码分类：1 到 5 分别是，消息、成功、重定向、客户端错误、服务端错误
- 4.Tcp
- 1.可靠连接，三次握手，四次挥手
    - 1.三次握手：防止了服务器端的一直等待而浪费资源，例如只是两次握手，如果 s 确认之后 c 就掉线了，那么 s 就会浪费资源
      - 1.syn-c = x，表示这消息是 x 序号
      - 2.ack-s = x + 1，表示 syn-c 这个消息接收成功。syn-s = y，表示这消息是 y 序号。
      - 3.ack-c = y + 1，表示 syn-s 这条消息接收成功
  - 2.四次挥手：TCP 是全双工模式
    - 1.fin-c = x，表示现在需要关闭 c 到 s 了。ack-c = y,表示上一条 s 的消息已经接收完毕
    - 2.ack-s = x + 1，表示需要关闭的 fin-c 消息已经接收到了，同意关闭
    - 3.fin-s = y + 1，表示 s 已经准备好关闭了，就等 c 的最后一条命令
    - 4.ack-c = y + 1，表示 c 已经关闭，让 s 也关闭
  - 3.滑动窗口，停止等待、后退 N、选择重传
  - 4.拥塞控制，慢启动、拥塞避免、加速递减、快重传快恢复

## TCP 协议与 UDP 协议的区别

**TCP ( Transmission Control Protocol , 传输控制协议 )** 是面向连接的协议，也就是说，在收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次“对话”才能建立起来，其中的过程非常复杂，只简单的描述下这三次对话的简单过程：主机 A 向主机 B 发出连接请求数据包：“我想给你发数据，可以吗？”，这是第一次对话；主机 B 向主机 A 发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包：“可以，你什么时候发？”，这是第二次对话；主机 A 再发出一个数据包确认主机 B 的要求同步：“我现在就发，你接着吧！”，这是第三次对话。三次“对话”的目的是使数据包的发送和接收同步，经过三次“对话”之后，主机 A 才向主机 B 正式发送数据。详细点说就是：

### **TCP 三次握手过程**

- 1 主机 A 通过向主机 B 发送一个含有同步序列号的标志位的数据段给主机 B ,向主机 B 请求建立连接,通过这个数据段,主机 A 告诉主机 B 两件事:我想要和你通信;你可以用哪个序列号作为起始数据段来回应我.
- 2 主机 B 收到主机 A 的请求后,用一个带有确认应答(ACK)和同步序列号(SYN)标志位的数据段响应主机 A,也告诉主机 A 两件事:我已经收到你的请求了,你可以传输数据了;你要用哪个序列号作为起始数据段来回应我



3 主机 A 收到这个数据段后,再发送一个确认应答,确认已收到主机 B 的数据段:"我已收到回复,我现在要开始传输实际数据了"

这样 3 次握手就完成了,主机 A 和主机 B 就可以传输数据了.

### **3 次握手的特点**

没有应用层的数据

SYN 这个标志位只有在 TCP 建产连接时才会被置 1

握手完成后 SYN 标志位被置 0

### **TCP 建立连接要进行 3 次握手,而断开连接要进行 4 次**

1 当主机 A 完成数据传输后,将控制位 FIN 置 1,提出停止 TCP 连接请求

2 主机 B 收到 FIN 后对其作出响应,确认这一方向上的 TCP 连接将关闭,将 ACK 置 1

3 由 B 端再提出反方向的关闭请求,将 FIN 置 1

4 主机 A 对主机 B 的请求进行确认,将 ACK 置 1,双方向的关闭结束.

由 TCP 的三次握手和四次断开可以看出,TCP 使用面向连接的通信方式,大大提高了数据通信的可靠性,使发送数据端和接收端在数据正式传输前就有了交互,为数据正式传输打下了可靠的基础

## 名词解释

**ACK** TCP 报头的控制位之一,对数据进行确认.确认由目的端发出,用它来告诉发送端这个序列号之前的数据段

都收到了.比如,确认号为 X,则表示前 X-1 个数据段都收到了,只有当 ACK=1 时,确认号才有效,当 ACK=0 时,确认号无效,这时会要求重传数据,保证数据的完整性.

**SYN** 同步序列号,TCP 建立连接时将这个位置 1

**FIN** 发送端完成发送任务位,当 TCP 完成数据传输需要断开时,提出断开连接的一方将这位置 1

## UDP ( User Data Protocol , 用户数据报协议 )

( 1 ) UDP 是一个非连接的协议 , 传输数据之前源端和终端不建立连接 , 当它想传送时就简单地去抓取来自应用程序的数据 , 并尽可能快地把它扔到网络上。在发送端 , UDP 传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制 ; 在接收端 , UDP 把每个消息段放在队列中 , 应用程序每次从队列中读一个消息段。

( 2 ) 由于传输数据不建立连接 , 因此也就不需要维护连接状态 , 包括收发状态等 , 因此一台服务机可同时向多个客户机传输相同的消息。

( 3 ) UDP 信息包的标题很短 , 只有 8 个字节 , 相对于 TCP 的 20 个字节信息包的额外开销很小。

( 4 ) 吞吐量不受拥挤控制算法的调节 , 只受应用软件生成数据的速率、传输带宽、源端和终端主机性能的限制。

( 5 ) UDP 使用**尽最大努力交付** , 即不保证可靠交付 , 因此主机不需要维持复杂的链接状态表 ( 这里面有许多参数 ) 。

( 6 ) UDP 是**面向报文**的。发送方的 UDP 对应用程序交下来的报文 , 在添加首部后就向下交付给 IP 层。既不拆分 , 也不合并 , 而是保留这些报文的边界 , 因此 , 应用程序需要选择合适的报文大小。

### **小结 TCP 与 UDP 的区别 :**

1. 基于连接与无连接 ;
2. 对系统资源的要求 ( TCP 较多 , UDP 少 ) ;
3. UDP 程序结构较简单 ;
4. 流模式与数据报模式 ;
5. TCP 保证数据正确性 , UDP 可能丢包 , TCP 保证数据顺序 , UDP 不保证。

以下内容来自百度百科：

第一次握手：建立连接时，客户端发送 syn 包（ $\text{syn}=j$ ）到服务器，并进入 SYN\_SENT 状态，等待服务器确认；SYN：同步序列编号（*Synchronize Sequence Numbers*）。

第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ $\text{ack}=j+1$ ），同时自己也发送一个 SYN 包（ $\text{syn}=k$ ），即 SYN+ACK 包，此时服务器进入 SYN\_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK（ $\text{ack}=k+1$ ），此包发送完毕，客户端和服务器进入 ESTABLISHED（TCP 连接成功）状态，完成三次握手。

完成三次握手，客户端与服务器开始传送数据，在上述过程中，还有一些重要的概念：

## 未连接队列

在三次握手协议中，服务器维护一个未连接队列，该队列为每个客户端的 SYN 包（ $\text{syn}=j$ ）开设一个条目，该条目表明服务器已收到 SYN 包，并向客户发出确认，正在等待客户的确认包。这些条目所标识的连接在服务器处于 SYN\_RECV 状态，当服务器收到客户的确认包时，删除该条目，服务器进入 ESTABLISHED 状态。

## 关闭 TCP 连接：改进的三次握手

对于一个已经建立的连接，TCP 使用改进的三次握手来释放连接（使用一个带有 FIN 附加标记的报文段）。TCP 关闭连接的步骤如下：

第一步，当主机 A 的应用程序通知 TCP 数据已经发送完毕时，TCP 向主机 B 发送一个带有 FIN 附加标记的报文段（FIN 表示英文 finish）。

第二步，主机 B 收到这个 FIN 报文段之后，并不立即用 FIN 报文段回复主机 A，而是先向主机 A 发送一个确认序号 ACK，同时通知自己相应的应用程序：对方要求关闭连接（先发送 ACK 的目的是为了防止在这段时间内，对方重传 FIN 报文段）。

第三步，主机 B 的应用程序告诉 TCP：我要彻底的关闭连接，TCP 向主机 A 送一个 FIN 报文段。

第四步，主机 A 收到这个 FIN 报文段后，向主机 B 发送一个 ACK 表示连接彻底释放。

## 11. 设计模式（六大基本原则、项目中常用的设计模式、手写单例等）

## 1.单一职责原则:不要存在多于一个导致类变更的原因。

通俗的说:即一个类只负责一项职责。

## 2.里氏替换原则:所有引用基类的地方必须能透明地使用其子类的对象。

通俗的说:当使用继承时。类 B 继承类 A 时，除添加新的方法完成新增功能 P2 外，尽量不要重写父类 A 的方法，也尽量不要重载父类 A 的方法。如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。子类可以扩展父类的功能，但不能改变父类原有的功能。

## 3.依赖倒置原则:高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

通俗的说:在 java 中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的，是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。依赖倒置原则的核心思想是面向接口编程。

## 4.接口隔离原则:客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

通俗的说:建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

## 5.迪米特法则:一个对象应该对其他对象保持最少的了解

通俗的说:尽量降低类与类之间的耦合。

## 6.开闭原则:一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

**通俗的说:用抽象构建框架,用实现扩展细节。**因为抽象灵活性好,适应性广,只要抽象的合理,可以基本保持软件架构的稳定。而软件中易变的细节,我们用从抽象派生的实现类来进行扩展,当软件需要发生变化时,我们只需要根据需求重新派生一个实现类来扩展就可以了。

## 1、你所知道的设计模式有哪些

Java 中一般认为有 23 种设计模式,我们不需要所有的都会,但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了,当然能掌握的越多越好。

总体来说设计模式分为三大类:

创建型模式,共五种:工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式,共七种:适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式,共十一种:策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

## 2、单例设计模式

最好理解的一种设计模式,分为懒汉式和饿汉式。

◆ 饿汉式:

```
public class Singleton {  
    // 直接创建对象  
    public static Singleton instance = new Singleton();  
  
    // 私有化构造函数  
    private Singleton() {  
    }  
  
    // 返回对象实例  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

◆ 懒汉式:

```

public class Singleton {
    // 声明变量
    private static volatile Singleton singleton2 = null;

    // 私有构造函数
    private Singleton2() {
    }

    // 提供对外方法
    public static Singleton2 getInstance() {
        if (singleton2 == null) {
            synchronized (Singleton2.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}

```

### 3、工厂设计模式

工厂模式分为工厂方法模式和抽象工厂模式。

#### ◆ 工厂方法模式

工厂方法模式分为三种：普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

#### ◆ 普通工厂模式



```

public interface Sender {
    public void Send();
}
public class MailSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is mail sender!");
    }
}
public class SmsSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
public class SendFactory {
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}

```

#### ◆ 多个工厂方法模式

该模式是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

```

public class SendFactory {
    public Sender produceMail(){
        return new MailSender();
    }

    public Sender produceSms(){
        return new SmsSender();
    }
}

public class FactoryTest {
    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produceMail();
        sender.send();
    }
}

```

◆ 静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```

public class SendFactory {
    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}

public class FactoryTest {
    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.send();
    }
}

```

◆ 抽象工厂模式

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要

拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

```
public interface Provider {
    public Sender produce();
}

-----

public interface Sender {
    public void send();
}

-----

public class MailSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is mail sender!");
    }
}

-----

public class SmsSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is sms sender!");
    }
}

-----

public class SendSmsFactory implements Provider {

    @Override
    public Sender produce() {
        return new SmsSender();
    }
}
```

```

public class SendMailFactory implements Provider {

    @Override
    public Sender produce() {
        return new MailSender();
    }
}

-----

public class Test {
    public static void main(String[] args) {
        Provider provider = new SendMailFactory();
        Sender sender = provider.produce();
        sender.send();
    }
}

```

#### 4、建造者模式 (Builder)

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的 **Test** 结合起来得到的。

```

public class Builder {
    private List<Sender> list = new ArrayList<Sender>();

    public void produceMailSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new MailSender());
        }
    }

    public void produceSmsSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new SmsSender());
        }
    }
}

```

```
public class TestBuilder {
    public static void main(String[] args) {
        Builder builder = new Builder();
        builder.produceMailSender(10);
    }
}
```

## 5、适配器设计模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

### 类的适配器模式

```
public class Source {
    public void method1() {
        System.out.println("this is original method!");
    }
}

-----

public interface Targetable {
    /* 与原类中的方法相同 */
    public void method1();
    /* 新类的方法 */
    public void method2();
}

public class Adapter extends Source implements Targetable {
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
}

public class AdapterTest {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
    }
}
```

### ◆ 对象的适配器模式

基本思路和类的适配器模式相同，只是将 **Adapter** 类作修改，这次不继承 **Source** 类，而是持有 **Source** 类的实例，以达到解决兼容性的问题。

```
public class Wrapper implements Targetable {
    private Source source;

    public Wrapper(Source source) {
        super();
        this.source = source;
    }

    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }

    @Override
    public void method1() {
        source.method1();
    }
}

-----

public class AdapterTest {

    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

### ◆ 接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

## 6、装饰模式 (Decorator)

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

```
public interface Sourceable {  
    public void method();  
}  
-----  
public class Source implements Sourceable {  
    @Override  
    public void method() {  
        System.out.println("the original method!");  
    }  
}  
-----  
public class Decorator implements Sourceable {  
    private Sourceable source;  
    public Decorator(Sourceable source) {  
        super();  
        this.source = source;  
    }  
  
    @Override  
    public void method() {  
        System.out.println("before decorator!");  
        source.method();  
        System.out.println("after decorator!");  
    }  
}  
-----  
public class DecoratorTest {  
    public static void main(String[] args) {  
        Sourceable source = new Source();  
        Sourceable obj = new Decorator(source);  
        obj.method();  
    }  
}
```

## 7、策略模式 (strategy)

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类

提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

```
public interface ICalculator {
    public int calculate(String exp);
}

-----

public class Minus extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "-");
        return arrayInt[0] - arrayInt[1];
    }
}

-----

public class Plus extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp, "\\+");
        return arrayInt[0] + arrayInt[1];
    }
}

-----

public class AbstractCalculator {
    public int[] split(String exp, String opt) {
        String array[] = exp.split(opt);
        int arrayInt[] = new int[2];
        arrayInt[0] = Integer.parseInt(array[0]);
        arrayInt[1] = Integer.parseInt(array[1]);
        return arrayInt;
    }
}
```



```

public class StrategyTest {
    public static void main(String[] args) {
        String exp = "2+8";
        ICalculator cal = new Plus();
        int result = cal.calculate(exp);
        System.out.println(result);
    }
}

```

## 8、观察者模式 (Observer)

观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

```

public interface Observer {
    public void update();
}

public class Observer1 implements Observer {
    @Override
    public void update() {
        System.out.println("observer1 has received!");
    }
}

public class Observer2 implements Observer {
    @Override
    public void update() {
        System.out.println("observer2 has received!");
    }
}

public interface Subject {
    /*增加观察者*/
    public void add(Observer observer);

    /*删除观察者*/
    public void del(Observer observer);
}

```

```

        /*通知所有的观察者*/
        public void notifyObservers();

        /*自身的操作*/
        public void operation();
    }

    public abstract class AbstractSubject implements Subject {

        private Vector<Observer> vector = new Vector<Observer>();

        @Override
        public void add(Observer observer) {
            vector.add(observer);
        }

        @Override
        public void del(Observer observer) {
            vector.remove(observer);
        }

        @Override
        public void notifyObservers() {
            Enumeration<Observer> enumo = vector.elements();
            while (enumo.hasMoreElements()) {
                enumo.nextElement().update();
            }
        }
    }

    public class MySubject extends AbstractSubject {

        @Override
        public void operation() {
            System.out.println("update self!");
            notifyObservers();
        }
    }

    public class ObserverTest {
        public static void main(String[] args) {
            Subject sub = new MySubject();
            sub.add(new Observer1());
        }
    }

```

```
sub.add(new Observer2());  
sub.operation();  
}  
}
```

## 12. 断点续传

原理解析

在开发当中，“断点续传”这种功能很实用和常见，听上去也是比较有“逼格”的感觉。所以通常我们都有兴趣去研究研究这种功能是如何实现的？

以 Java 来说，网络上也能找到不少关于实现类似功能的资料。但是呢，大多数都是举个 Demo 然后贴出源码，真正对其实现原理有详细的说明很少。

于是我们在最初接触的时候，很可能就是直接 Ctrl + C/V 代码，然后捣鼓捣鼓，然而最终也能把效果弄出来。但初学时这样做其实很显然是有好有坏的。

好处在于，源码很多，解释很少；如果我们肯下功夫，针对于别人贴出的代码里那些自己不明白的东西去查资料，去钻研。最终多半会收获颇丰。

坏处也很明显：作为初学者，面对一大堆的源码，感觉好多东西都很陌生，就很容易望而生畏。即使最终大致了解了用法，但也不一定明白实现原理。

我们今天就一起从最基本的角度切入，来看看所谓的“断点续传”这个东西是不是真的如此“高逼格”。

其实在接触一件新的“事物”的时候，将它拟化成一些我们本身比较熟悉的事物，来参照和对比着学习。通常会事半功倍。

如果我们刚接触“断点续传”这个概念，肯定很难说清楚个一二三。那么，“玩游戏”我们肯定不会陌生。

OK，那就假设我们现在有一款“通关制的 RPG 游戏”。想想我们在玩这类游戏时通常会怎么做？

很明显，第一天我们浴血奋战，大杀四方，假设终于来到了第四关。虽然激战正酣，但一看墙上的时钟，已经凌晨 12 点，该睡觉了。

这个时候就很尴尬了，为了能够在下一次玩的时候，顺利接轨上我们本次游戏的进度，我们应该怎么办呢？

很简单，我们不关掉游戏，直接去睡觉，第二天再接着玩呗。这样是可以，但似乎总觉得哪里让人不爽。

那么，这个时候，如果这个游戏有一个功能叫做“存档”，就很关键了。我们直接选择存档，输入存档名“第四关”，然后就可以关闭游戏了。

等到下次进行游戏时，我们直接找到“第四关”这个存档，然后进行读档，就可以接着进行游戏了。

这个时候，所谓的“断点续传”就很好理解了。我们顺着我们之前“玩游戏”的思路来理一下：

假设，现在有一个文件需要我们进行下载，当我们下载了一部分的时候，出现情况了，比如：电脑死机、没电、网络中断等等。

其实这就好比之前玩游戏玩着玩着，突然 12 点需要去睡觉休息了是一个道理。OK，那么这个时候的情况是：

如果游戏不能存档，那么则意味着我们下次游戏的时候，这次已经通过的 4 关的进度将会丢失，无法接档。

对应的，如果“下载”的行为无法记录本次下载的一个进度。那么，当我们再次下载这个文件也就只能从头来过。

话到这里，其实我们已经发现了，对于我们以上所说的行为，关键就在于一个字“续”！而我们要实现让一种断开的行为“续”起来的目的，关键就在于要有“介质”能够记录和读取行为出现“中断”的这个节点的信息。

转化到编程世界

实际上这就是“断点续传”最最基础的原理，用大白话说就是：我们要在下载行为出现中断的时候，记录下中断的位置信息，然后在下次行为中读取。

有了这个位置信息之后，想想我们该怎么做。是的，很简单，在新的下载行为开始的时候，直接从记录的这个位置开始下载内容，而不再从头开始。

好吧，我们用大白话掰扯了这么久的原理，开始觉得无聊了。那么我们现在最后总结一下，然后就来看看我们应该怎么把原理转换到编程世界中去。

当“上传(下载)的行为”出现中断，我们需要记录本次上传(下载)的位置(position)。

当“续”这一行为开始，我们直接跳转到 position 处继续上传(下载)的行为。

显然问题的关键就在于所谓的“position”，以我们举的“通关游戏来说”，可以用“第几关”来作为这个 position 的单位。

那么转换到所谓的“断点续传”，我们该使用什么来衡量“position”呢？很显然，回归二进制，因为这里的本质无非就是文件的读写。

那么剩下的工作就很简单了，先是记录 position，这似乎都没什么值得说的，因为只是数据的持久化而已(内存，文件，数据库)，我们有很多方式。

另一个关键在于当“续传”的行为开始，我们需要需要从上次记录的 position 位置开始读写操作，所以我们需要一个类似于“指针”功能的东西。

我们当然也可以自己想办法去实现这样一个“指针”，但高兴地是，Java 已经为我们提供了这样的一个类，那就是 RandomAccessFile。

这个类的功能从名字就很直观的体现了，能够随机的去访问文件。我们看一下 API 文档中对该类的说明：

此类的实例支持对随机访问文件的读取和写入。随机访问文件的行为类似存储在文件系统中

的一个大型 `byte` 数组。

如果随机访问文件以读取/写入模式创建，则输出操作也可用；输出操作从文件指针开始写入字节，并随着对字节的写入而前移此文件指针。

写入隐含数组的当前末尾之后的输出操作导致该数组扩展。该文件指针可以通过 `getFilePointer` 方法读取，并通过 `seek` 方法设置。

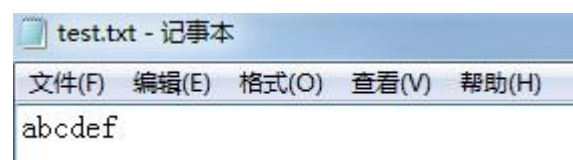
看完 API 说明，我们笑了，是的，这不正是我们要的吗？那好吧，我们磨刀磨了这么久了，还不快去砍砍柴吗？

### 实例演示

既然是针对于文件的“断点续传”，那么很明显，我们先搞一个文件出来。也许音频文件，图像文件什么的看上去会更上档次一点。

但我们已经说了，在计算机大兄弟眼中，它们最终都将回归“二进制”。所以我们这里就创建一个简单的“txt”文件，因为 txt 更利于理解。

我们在 D 盘的根目录下创建一个名为“test.txt”的文件，文件内容很简单，如图所示：



没错，我们输入的内容就是简单的 6 个英语字母。然后我们右键→属性：

位置: D:\  
大小: 6 字节 (6 字节)

我们看到，文件现在的大小是 6 个字节。这也就是为什么我们说，所有的东西到最后还是离不开“二进制”。

是的，我们都明白，因为我们输入了 6 个英文字母，而 1 个英文字母将占据的存储空间是 1 个字节(即 8 个比特位)。

目前为止，我们看到的都很无聊，因为这基本等于废话，稍微有计算机常识的人都知道这些知识。别着急，我们继续。

在 Java 中对一个文件进行读写操作很简单。假设现在的需求如果是“把 D 盘的这个文件写入到 E 盘”，那么我们会提起键盘，啪啪啪啪，搞定！

但其实所谓的文件的“上传(下载)”不是也没什么不同吗？区别就仅仅在于行为由“仅仅在本机之间”转变成了“本机与服务器之间”的文件读写。

这时我们会说，“别逼逼了，这些谁都知道，‘断点续传’呢？”，其实到了这里也已经很简单了，我们再次明确，断点续传要做的无非就是：

前一次读写行为如果出现中断，请记录下此次读写完成的文件内容的位置信息；当“续传开始”则直接将指针移到此处，开始继续读写操作。

反复的强调原理，实际上是因为只要弄明白了原理，剩下的就只是招式而已了。这就就像武侠小说里的“九九归一”大法一样，最高境界就是回归本源。任何复杂的事物，只要明白其原理，我们就能将其剥离，还原为一个个简单的事物。同理，一系列简单的事物，经过逻辑组合，就形成了复杂的事物。

下面，我们马上就将回归混沌，以最基本的形式模拟一次“断点续传”。在这里我们连服务器的代码都不去写了，直接通过一个本地测试类搞定。

我们要实现的效果很简单：将在 D 盘的”test.txt”文件写入到 E 盘当中，但中途我们会模拟一次”中断”行为，然后在重新继续上传，最终完成整个过程。

也就是说，我们这里将会把“D 盘”视作一台电脑，并且直接将”E 盘”视作一台服务器。那么这样我们甚至都不再与 http 协议扯上半毛钱关系了，（当然实际开发我们肯定是还是得与它扯上关系的 ^<^），从而只关心最基本的文件读写的”断”和”续”的原理是怎么样

为了通过对比加深理解，我们先来写一段正常的代码，即正常读写，不发生中断：

```
public class Test {

    public static void main(String[] args) {
        // 源文件与目标文件
        File sourceFile = new File("D:/", "test.txt");
        File targetFile = new File("E:/", "test.txt");
        // 输入输出流
        FileInputStream fis = null;
        FileOutputStream fos = null;
        // 数据缓冲区
        byte[] buf = new byte[1];

        try {
            fis = new FileInputStream(sourceFile);
            fos = new FileOutputStream(targetFile);
            // 数据读写
            while (fis.read(buf) != -1) {
                System.out.println("write data...");
                fos.write(buf);
            }
        } catch (FileNotFoundException e) {
            System.out.println("指定文件不存在");
        } catch (IOException e) {
            // TODO: handle exception
        } finally {
            try {
                // 关闭输入输出流
                if (fis != null)
```

```

        fis.close();

        if (fos != null)
            fos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

该段代码运行，我们就会发现在 E 盘中已经成功拷贝了一份“test.txt”。这段代码很简单，唯一稍微说一下就是：

我们看到我们将 buf，即缓冲区 设置的大小是 1，这其实就代表我们每次 read，是读取一个字节的数(即 1 个英文字母)。

现在，我们就来模拟这个读写中断的行为，我们将之前的代码完善如下：

```

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.RandomAccessFile;

public class Test {

    private static int position = -1;

    public static void main(String[] args) {
        // 源文件与目标文件
        File sourceFile = new File("D:/", "test.txt");
        File targetFile = new File("E:/", "test.txt");
        // 输入输出流
        FileInputStream fis = null;
        FileOutputStream fos = null;
        // 数据缓冲区
        byte[] buf = new byte[1];

        try {
            fis = new FileInputStream(sourceFile);
            fos = new FileOutputStream(targetFile);
            // 数据读写
            while (fis.read(buf) != -1) {

```

```

        fos.write(buf);
        // 当已经上传了 3 字节的文件内容时，网络中断了，抛出异常
        if (targetFile.length() == 3) {
            position = 3;
            throw new FileAccessException();
        }
    }
} catch (FileAccessException e) {
    keepGoing(sourceFile, targetFile, position);
} catch (FileNotFoundException e) {
    System.out.println("指定文件不存在");
} catch (IOException e) {
    // TODO: handle exception
} finally {
    try {
        // 关闭输入输出流
        if (fis != null)
            fis.close();

        if (fos != null)
            fos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

```

private static void keepGoing(File source, File target, int position) {
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    try {
        RandomAccessFile readFile = new RandomAccessFile(source, "rw");
        RandomAccessFile writeFile = new RandomAccessFile(target, "rw");
        readFile.seek(position);
        writeFile.seek(position);

        // 数据缓冲区
        byte[] buf = new byte[1];
    }
}

```



```

        // 数据读写
        while (readFile.read(buf) != -1) {
            writeFile.write(buf);
        }
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}
}

```

```

class FileAccessException extends Exception {

}

```

总结一下，我们在这次改动当中都做了什么工作：

首先，我们定义了一个变量 **position**，记录在发生中断的时候，已完成读写的位置。（这是为了方便，实际来说肯定应该讲这个值存到文件或者数据库等进行持久化）

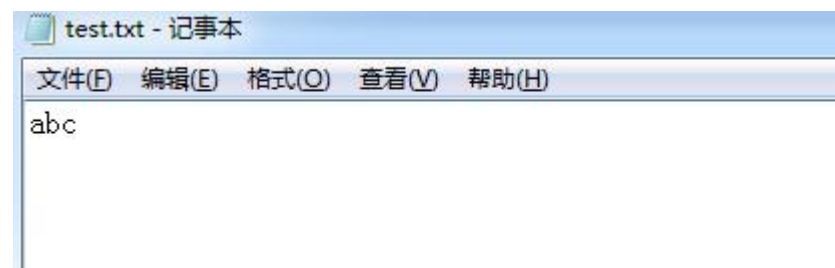
然后在文件读写的 **while** 循环中，我们去模拟一个中断行为的发生。这里是当 **targetFile** 的文件长度为 3 个字节则模拟抛出一个我们自定义的异常。（我们可以想象为实际下载中，已经上传(下载)了”x”个字节的内容，这个时候网络中断了，那么我们就在网络中断抛出的异常中将”x”记录下来）。

剩下的就如果我们之前说的一样，在“续传”行为开始后，通过 **RandomAccessFile** 类来包装我们的文件，然后通过 **seek** 将指针指定到之前发生中断的位置进行读写就搞定了。

（实际的文件下载上传，我们当然需要将保存的中断值上传给服务器，这个方式通常为 **httpConnection.setRequestProperty(“RANGE”, “bytes=x”);**）

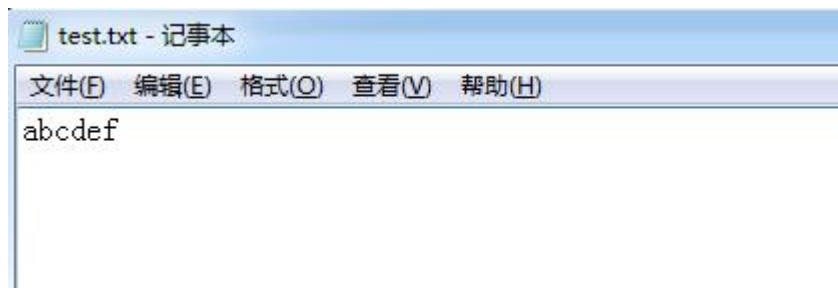
在我们这段代码，开启”续传”行为，即 **keepGoing** 方法中：我们起头让线程休眠 10 秒钟，这正是为了让我们运行程序看到效果。

现在我们运行程序，那么文件就会开启“由 D 盘上传到 E 盘的过程”，我们首先点开 E 盘，会发现的确多了一个 **test.txt** 文件，打开它发现内容如下：



没错，这个时候我们发现内容只有“abc”。这是在我们预料以内的，因为我们的程序模拟在文件上传了 3 个字节的时候发生了中断。

Ok，我们静静的等待 10 秒钟过去，然后再点开该文件，看看是否能够成功：



通过截图我们发现内容的确已经变成了“abc”，由此也就完成了续传。

## 13. Java 四大引用

从 JDK1.2 版本开始，把对象的引用分为四种级别，从而使程序能更加灵活的控制对象的生命周期。这四种级别由

高到低依次为：强引用、软引用、弱引用和虚引用。

### 强引用(StrongReference)

我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可

少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 `OutOfMemoryError` 错误，使程序

异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

### 软引用(SoftReference)

如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器

没有回收它，该对象就可以被程序使用。

### 弱引用 (WeakReference)

在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够

与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱

引用的对象。弱引用可以和一个引用队列(`ReferenceQueue`)联合使用，如果弱引用所引用的

对象

被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

虚引用(PhantomReference)

如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。虚引用主要用来跟

踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列(ReferenceQueue)联合

使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加

入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要

被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必

要的行动。

下面看两个 Demo

```
public class Demo1 {  
  
    public static void main(String[] args) {  
  
        //这就是一个强引用  
        String str="hello";  
        //现在我们由上面的强引用创建一个软引用,所以现在 str 有两个引用指向它  
        SoftReference<String> soft=new SoftReference<String>(str);  
        str=null;  
        //可以使用 get()得到引用指向的对象  
        System.out.println(soft.get());//输出 hello  
  
    }  
}
```

```
public class Demo2 {
```

```

public static void main(String[] args) {

    //这就是一个强引用
    String str="hello";
    ReferenceQueue<? super String> q=new ReferenceQueue<String>();
    //现在我们由上面的强引用创建一个虚引用,所以现在 str 有两个引用指向它
    PhantomReference<String> p=new PhantomReference<String>(str, q);
    //可以使用 get()得到引用指向的对象
    System.out.println(q.poll());//输出 null

}
}

```

下面再看一个，首先创建一个 **Store** 类，内部定义一个很大的数组，目的是创建对象时，会得到更多的内存，以提高回收的可能性！

```

public class Store {

    public static final int SIZE = 10000;
    private double[] arr = new double[SIZE];
    private String id;

    public Store() {

    }

    public Store(String id) {
        super();
        this.id = id;
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println(id + "被回收了");
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}

```

```

    }

    @Override
    public String toString() {
        return id;
    }
}

```

依次创建软引用，弱引用，虚引用个 10 个！

```

public class Demo3 {

    public static ReferenceQueue<Store> queue = new ReferenceQueue<Store>();

    public static void checkQueue()
    {
        if(queue!=null)
        {
            @SuppressWarnings("unchecked")
            Reference<Store> ref =(Reference<Store>)queue.poll();
            if(ref!=null)
                System.out.println(ref+"....."+ref.get());
        }
    }

    public static void main(String[] args) {

        HashSet<SoftReference<Store>> hs1 = new HashSet<SoftReference<Store>>();
        HashSet<WeakReference<Store>> hs2 = new HashSet<WeakReference<Store>>();

        //创建 10 个软引用
        for(int i=1;i<=10;i++)
        {
            SoftReference<Store> soft = new SoftReference<Store>(new
Store("soft"+i),queue);
            System.out.println("create soft"+soft.get());
            hs1.add(soft);
        }
        System.gc();
        checkQueue();

        //创建 10 个弱引用
        for(int i=1;i<=10;i++)

```

```

        {
            WeakReference<Store> weak = new WeakReference<Store>(new
Store("weak"+i),queue);
            System.out.println("create weak"+weak.get());
            hs2.add(weak);
        }

        System.gc();
        checkQueue();
        //创建 10 个虚引用
        HashSet<PhantomReference<Store>> hs3 = new
HashSet<PhantomReference<Store>>();
        for(int i=1;i<=10;i++)
        {
            PhantomReference<Store> phantom = new PhantomReference<Store>(new
Store("phantom"+i),queue);
            System.out.println("create phantom "+phantom.get());
            hs3.add(phantom);
        }
        System.gc();
        checkQueue();
    }
}

```

程序执行结果：

```

create softsoft1
create softsoft2
create softsoft3
create softsoft4
create softsoft5
create softsoft6
create softsoft7
create softsoft8
create softsoft9
create softsoft10
create weakweak1
create weakweak2
create weakweak3
create weakweak4
create weakweak5
create weakweak6
create weakweak7
create weakweak8
create weakweak9
create weakweak10
create phantom null
create phantom null
create phantom null
create phantom null
create phantom null
create phantom null
create phantom null
create phantom null
create phantom null
create phantom null
java.lang.ref.WeakReference@96cf11.....null
weak10被回收了
phantom10被回收了
phantom9被回收了
phantom8被回收了
phantom7被回收了

```

---

可以看到虚引用和弱引用被回收掉。。。

## 14. Java 的泛型

### 一. 泛型概念的提出（为什么需要泛型）？

首先，我们看下下面这段简短的代码：

```

public class GenericTest {

    public static void main(String[] args) {

        List list = new ArrayList();

        list.add("qqyumidi");

        list.add("corn");

        list.add(100);

        for (int i = 0; i < list.size(); i++) {

            String name = (String) list.get(i); // 1

            System.out.println("name:" + name);

        }

    }

}

```

定义了一个 **List** 类型的集合，先向其中加入了两个字符串类型的值，随后加入一个 **Integer** 类型的值。这是完全允许的，因为此时 **list** 默认的类型为 **Object** 类型。在之后的循环中，由于忘记了之前在 **list** 中也加入了 **Integer** 类型的值或其他编码原因，很容易出现类似于 **//1** 中的错误。因为编译阶段正常，而运行时会出现“**java.lang.ClassCastException**”异常。因此，导致此类错误编码过程中不易发现。

在如上的编码过程中，我们发现主要存在两个问题：

1. 当我们将一个对象放入集合中，集合不会记住此对象的类型，当再次从集合中取出此对象时，改对象的编译类型变成了 **Object** 类型，但其运行时类型任然为其本身类型。
2. 因此，**//1** 处取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现“**java.lang.ClassCastException**”异常。

那么有没有什么办法可以使集合能够记住集合内元素各类型，且能够达到只要编译时不出现问题，运行时就不会出现“**java.lang.ClassCastException**”异常呢？答案就是使用泛型。

## 二.什么是泛型？

泛型，即“参数化类型”。一提到参数，最熟悉的的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

看着好像有点复杂，首先我们看下上面那个例子采用泛型的写法。



```

public class GenericTest {

    public static void main(String[] args) {

        /*

        List list = new ArrayList();

        list.add("qqyumidi");

        list.add("corn");

        list.add(100);

        */

        List<String> list = new ArrayList<String>();

        list.add("qqyumidi");

        list.add("corn");

        //list.add(100);    // 1 提示编译错误

        for (int i = 0; i < list.size(); i++) {

            String name = list.get(i); // 2

            System.out.println("name:" + name);

        }

    }

}

```

采用泛型写法后，在//1处想加入一个 **Integer** 类型的对象时会出现编译错误，通过 **List<String>**，直接限定了 **list** 集合中只能含有 **String** 类型的元素，从而在//2处无须进行强制类型转换，因为此时，集合能够记住元素的类型信息，编译器已经能够确认它是 **String** 类型了。

结合上面的泛型定义，我们知道在 **List<String>** 中，**String** 是类型实参，也就是说，相应的 **List** 接口中肯定含有类型形参。且 **get()** 方法的返回结果也直接是此形参类型（也就是对应的传入的类型实参）。下面就来看看 **List** 接口的具体定义：

```

public interface List<E> extends Collection<E> {

    int size();
}

```

```
boolean isEmpty();
```

```
boolean contains(Object o);
```

```
Iterator<E> iterator();
```

```
Object[] toArray();
```

```
<T> T[] toArray(T[] a);
```

```
boolean add(E e);
```

```
boolean remove(Object o);
```

```
boolean containsAll(Collection<?> c);
```

```
boolean addAll(Collection<? extends E> c);
```

```
boolean addAll(int index, Collection<? extends E> c);
```

```
boolean removeAll(Collection<?> c);
```

```
boolean retainAll(Collection<?> c);
```

```
void clear();
```

```
boolean equals(Object o);
```

```
int hashCode();
```

```
E get(int index);
```

```
E set(int index, E element);
```

```
void add(int index, E element);
```

```
E remove(int index);
```

```
int indexOf(Object o);
```

```
int lastIndexOf(Object o);
```

```
ListIterator<E> listIterator();
```

```
ListIterator<E> listIterator(int index);
```

```
List<E> subList(int fromIndex, int toIndex);
```

```
}
```

我们可以看到，在 **List** 接口中采用泛型化定义之后，<E>中的 **E** 表示类型形参，可以接收具体的类型实参，并且此接口定义中，凡是出现 **E** 的地方均表示相同的接受自外部的类型实参。

自然的，**ArrayList** 作为 **List** 接口的实现类，其定义形式是：

```
public class ArrayList<E> extends AbstractList<E>

    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    public boolean add(E e) {

        ensureCapacityInternal(size + 1); // Increments modCount!!

        elementData[size++] = e;

        return true;

    }
```

```

    public E get(int index) {

        rangeCheck(index);

        checkForComodification();

        return ArrayList.this.elementData(offset + index);

    }

    //...省略掉其他具体的定义过程

}

```

由此，我们从源代码角度明白了为什么//1 处加入 **Integer** 类型对象编译错误，且//2 处 **get()**到的类型直接就是 **String** 类型了。

### 三.自定义泛型接口、泛型类和泛型方法

从上面的内容中，大家已经明白了泛型的具体运作过程。也知道了接口、类和方法也都可以使用泛型去定义，以及相应的使用。是的，在具体使用时，可以分为泛型接口、泛型类和泛型方法。

自定义泛型接口、泛型类和泛型方法与上述 **Java** 源码中的 **List**、**ArrayList** 类似。如下，我们看一个最简单的泛型类和方法定义：

```

public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");

        System.out.println("name:" + name.getData());

    }

}

class Box<T> {

    private T data;

```

```

    public Box() {

    }

    public Box(T data) {

        this.data = data;

    }

    public T getData() {

        return data;

    }

}

```

在泛型接口、泛型类和泛型方法的定义过程中，我们常见的如 **T**、**E**、**K**、**V** 等形式的参数常用于表示泛型形参，由于接收来自外部使用时传入的类型实参。**那么对于不同传入的类型实参，生成的相应对象实例的类型是不是一样的呢？**

```

public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");

        Box<Integer> age = new Box<Integer>(712);

        System.out.println("name class:" + name.getClass()); //
com.qgyumidi.Box

        System.out.println("age class:" + age.getClass()); //
com.qgyumidi.Box

        System.out.println(name.getClass() == age.getClass()); // true

    }
}

```

```
}
```

由此，我们发现，在使用泛型类时，虽然传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是原来的最基本的类型（本实例中为 **Box**），当然，在逻辑上我们可以理解成多个不同的泛型类型。

究其原因，在于 **Java** 中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出，也就是说，成功编译后的 **class** 文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

对此总结成一句话：**泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。**

#### 四.类型通配符

接着上面的结论，我们知道，**Box<Number>**和**Box<Integer>**实际上都是 **Box** 类型，现在需要继续探讨一个问题，那么在逻辑上，类似于 **Box<Number>**和**Box<Integer>**是否可以看成具有父子关系的泛型类型呢？

为了弄清这个问题，我们继续看下下面这个例子：

```
public class GenericTest {  
  
    public static void main(String[] args) {  
  
        Box<Number> name = new Box<Number>(99);  
  
        Box<Integer> age = new Box<Integer>(712);  
  
        getData(name);  
  
        //The method getData(Box<Number>) in the type GenericTest is  
        //not applicable for the arguments (Box<Integer>)  
  
        getData(age);    // 1  
  
    }  
  
    public static void getData(Box<Number> data){
```

```

        System.out.println("data :" + data.getData());
    }

}

```

我们发现，在代码//1 处出现了错误提示信息：The method `getData(Box<Number>)` in the type `GenericTest` is not applicable for the arguments `(Box<Integer>)`。显然，通过提示信息，我们知道 `Box<Number>` 在逻辑上不能视为 `Box<Integer>` 的父类。那么，原因何在呢？

```

public class GenericTest {

    public static void main(String[] args) {

        Box<Integer> a = new Box<Integer>(712);

        Box<Number> b = a; // 1

        Box<Float> f = new Box<Float>(3.14f);

        b.setData(f); // 2

    }

    public static void getData(Box<Number> data) {

        System.out.println("data :" + data.getData());

    }

}

```

```

class Box<T> {

    private T data;

    public Box() {

```

```

    }

    public Box(T data) {

        setData(data);

    }

    public T getData() {

        return data;

    }

    public void setData(T data) {

        this.data = data;

    }

}

```

这个例子中，显然//1 和//2 处肯定会出现错误提示的。在此我们可以使用反证法来进行说明。

假设 **Box<Number>** 在逻辑上可以视为 **Box<Integer>** 的父类，那么//1 和//2 处将不会有错误提示了，那么问题就出来了，通过 **getData()** 方法取出数据时到底是什么类型呢？**Integer?** **Float?** 还是 **Number?** 且由于在编程过程中的顺序不可控性，导致在必要的时候必须要进行类型判断，且进行强制类型转换。显然，这与泛型的理念矛盾，因此，**在逻辑上 Box<Number> 不能视为 Box<Integer> 的父类。**

好，那我们回过头来继续看“类型通配符”中的第一个例子，我们知道其具体的错误提示的深层次原因了。那么如何解决呢？总部能再定义一个新的函数吧。这和 **Java** 中的多态理念显然是违背的，因此，我们需要一个在逻辑上可以用来表示同时是 **Box<Integer>** 和 **Box<Number>** 的父类的一个引用类型，由此，类型通配符应运而生。

类型通配符一般是使用 **?** 代替具体的类型实参。注意了，此处是类型实参，而不是类型形参！且 **Box<?>** 在逻辑上是 **Box<Integer>**、**Box<Number>**... 等所有 **Box<具体类型实参>** 的父类。由此，我们依然可以定义泛型方法，来完成此类需求。

```

public class GenericTest {

    public static void main(String[] args) {

```



```

        Box<String> name = new Box<String>("corn");

        Box<Integer> age = new Box<Integer>(712);

        Box<Number> number = new Box<Number>(314);

        getData(name);

        getData(age);

        getData(number);

    }

    public static void getData(Box<?> data) {

        System.out.println("data :" + data.getData());

    }

}

```

有时候，我们还可能听到[类型通配符上限](#)和[类型通配符下限](#)。具体有是怎么样的呢？

在上面的例子中，如果需要定义一个功能类似于 `getData()` 的方法，但对类型实参又有进一步的限制：只能是 **Number** 类及其子类。此时，需要用到类型通配符上限。

```

public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");

        Box<Integer> age = new Box<Integer>(712);

        Box<Number> number = new Box<Number>(314);

        getData(name);

        getData(age);

        getData(number);

        //getUpperNumberData(name); // 1
    }
}

```

```

        getUpperNumberData(age);    // 2

        getUpperNumberData(number); // 3

    }

    public static void getData(Box<?> data) {

        System.out.println("data :" + data.getData());

    }

    public static void getUpperNumberData(Box<? extends Number> data){

        System.out.println("data :" + data.getData());

    }

}

```

此时，显然，在代码//1 处调用将出现错误提示，而//2 //3 处调用正常。

## 15. final、finally、finalize 的区别

### 1. final

在 java 中，final 可以用来修饰类，方法和变量（成员变量或局部变量）。下面将对其详细介绍。

#### 1.1 修饰类

当用 final 修饰类的时，表明该类不能被其他类所继承。当我们需要让一个类永远不被继承，此时就可以用 final 修饰，但要注意：

**final 类中所有的成员方法都会隐式的定义为 final 方法。**

#### 1.2 修饰方法


使用 final 方法的原因主要有两个：

(1) 把方法锁定，以防止继承类对其进行更改。


(2) 效率，在早期的 java 版本中，会将 final 方法转为内嵌调用。但若方法过于庞大，可能在性能上不会有太大提升。因此在最近版本中，不需要 final 方法进行这些优化了。

final 方法意味着“最后的、最终的”含义，即此方法不能被重写。

注意：若父类中 **final** 方法的访问权限为 **private**，将导致子类中不能直接继承该方法，因此，此时可以在子类中定义相同方法名的函数，此时不会与重写 **final** 的矛盾，而是在子类中重新地定义了新方法。



```
class A{  
  
    private final void getName(){  
  
    }  
}  
  
public class B extends A{  
  
    public void getName(){  
  
    }  
}  
  
public static void main(String[] args){  
  
    System.out.println("OK");  
}  
}
```



## 1.3 修饰变量

**final** 成员变量表示常量，只能被赋值一次，赋值后其值不再改变。类似于 C++ 中的 **const**。

当 **final** 修饰一个基本数据类型时，表示该基本数据类型的值一旦在初始化后便不能发生变化；如果 **final** 修饰一个引用类型时，则在对其初始化之后便不能再让其指向其他对象了，但该引用所指向的对象的内容是可以发生变化的。本质上是一回事，因为引用的值是一个地址，**final** 要求值，即地址的值不发生变化。

**final** 修饰一个成员变量（属性），必须要显示初始化。这里有两种初始化方式，一种是在变量声明的时候初始化；第二种方法是在声明变量的时候不赋初值，但是要在这个变量所在的类的所有的构造函数中对这个变量赋初值。

当函数的参数类型声明为 **final** 时，说明该参数是只读型的。即你可以读取使用该参数，但是无法改变该参数的值。

```

public class Main{
    public static void main(String[] args) {
        //String str=new String("ABC");
        final String str1="Hello KT";
        //final String str2=str;
        System.out.println(str1);
        //System.out.println(str2);
        str1="KT";
        //str="CBA";
        System.out.println(str1);
        //System.out.println(str2);
    }
}

```

```

ktao@ktao-PC: ~/Desktop/C++Test$ javac test1.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=gasp
test1.java:1: 错误: 类Main是公共的, 应在名为 Main.java 的文件中声明
public class Main{
      ^
test1.java:8: 错误: 无法为最终变量str1分配值
        str1="KT";
            ^
2 个错误

```

在 java 中, String 被设计成 final 类, 那为什么平时使用时, String 的值可以被改变呢?

字符串常量池是 java 堆内存中一个特殊的存储区域, 当我们建立一个 String 对象时, 假设常量池不存在该字符串, 则创建一个, 若存在则直接引用已经存在的字符串。当我们对 String 对象值改变的时候, 例如 `String a="A"; a="B"`。a 是 String 对象的一个引用 (我们这里所说的 String 对象其实是指字符串常量), 当 `a="B"` 执行时, 并不是原本 String 对象("A")发生改变, 而是创建一个新的对象("B"), 令 a 引用它。

## 2. finally

finally 作为异常处理的一部分, 它只能用在 try/catch 语句中, 并且附带一个语句块, 表示这段语句最终一定会被执行 (不管有没有抛出异常), 经常被用在需要释放资源的情况下。 (x) (这句话其实存在一定的问题)

很多人都认为 finally 语句块一定会执行, 但真的是这样么? 答案是否定的, 例如下面这个例子:

```

public class Test{
    public static int test(){
        int i=1;
        //if(i==1){
        //    return 0;
        //}
        System.out.println("The previous statement of try block");
        i=i/0;
        try{
            System.out.println("try block");
            return i;
        }
        finally{
            System.out.println("finally block");
        }
    }
    public static void main(String[] args) {
        System.out.println("return value of test(): "+test());
    }
}

```

```

>_ ktao +
ktao@ktao-PC:~$ java Test
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=gasp
The previous statement of try block
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Test.test(Test.java:8)
    at Test.main(Test.java:18)
ktao@ktao-PC:~$

```

当我们去掉注释的三行语句，执行结果为：

```

ktao@ktao-PC:~$ java Test
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=gasp
return value of test(): 0

```

为什么在以上两种情况下都没有执行 **finally** 语句呢，说明什么问题？

只有与 **finally** 对应的 **try** 语句块得到执行的情况下，**finally** 语句块才会执行。以上两种情况在执行 **try** 语句块之前已经返回或抛出异常，所以 **try** 对应的 **finally** 语句并没有执行。

但是，在某些情况下，即使 **try** 语句执行了，**finally** 语句也不一定执行。例如以下情况：

```

public class Test{
    public static int test(){
        int i=1;
        //if(i==1){
        //    return 0;
        //}
        try{
            System.out.println("try block");
            System.exit(0);
            return i;
        }
        finally{
            System.out.println("finally block");
        }
    }
    public static void main(String[] args) {
        System.out.println("return value of test(): "+test());
    }
}

```

```

>_ ktao +
ktao@ktao-PC:~$ java Test
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=gasp
try block
ktao@ktao-PC:~$ █

```

`finally` 语句块还是没有执行，为什么呢？因为我们在 `try` 语句块中执行了 `System.exit(0)` 语句，终止了 Java 虚拟机的运行。那有人说了，在一般的 Java 应用中基本上是不会调用这个 `System.exit(0)` 方法的。OK！没有问题，我们不调用 `System.exit(0)` 这个方法，那么 `finally` 语句块就一定会执行吗？

再一次让大家失望了，答案还是否定的。当一个线程在执行 `try` 语句块或者 `catch` 语句块时被打断（interrupted）或者被终止（killed），与其相对应的 `finally` 语句块可能不会执行。还有更极端的情况，就是在线程运行 `try` 语句块或者 `catch` 语句块时，突然死机或者断电，`finally` 语句块肯定不会执行了。可能有人认为死机、断电这些理由有些强词夺理，没有关系，我们只是为了说明这个问题。

## 易错点

在 `try-catch-finally` 语句中执行 `return` 语句。我们看如下代码：

```

public class Test {
    public static void main(String[] args) {
        System.out.println(test(null) + "," + test("0") + "," + test("4,4,4"));
    }

    public static int test(String str) {
        try {
            return str.charAt(0) - '0';
        } catch (NullPointerException e1) {
            return 1;
        } catch (StringIndexOutOfBoundsException e2) {
            return 2;
        } catch (Exception e3) {
            return 3;
        } finally {
            return 4;
        }
    }
}

```

答案：4,4,4 。 为什么呢？

首先 `finally` 语句在代码中一定会执行，从运行结果来看，每次 `return` 的结果都是 4（即 `finally` 语句），仿佛其他 `return` 语句被屏蔽掉了。

事实也确实如此，因为 `finally` 用法特殊，所以会撤销之前的 `return` 语句，继续执行最后的 `finally` 块中的代码。

### 3. finalize

`finalize()` 是在 `java.lang.Object` 里定义的，也就是说每一个对象都有这么个方法。这个方法在 `gc` 启动，该对象被回收的时候被调用。其实 `gc` 可以回收大部分的对象（凡是 `new` 出来的对象，`gc` 都能搞定，一般情况下我们又不会用 `new` 以外的方式去创建对象），所以一般是不需要程序员去实现 `finalize` 的。特殊情况下，需要程序员实现 `finalize`，当对象被回收的时候释放一些资源，比如：一个 `socket` 链接，在对象初始化时创建，整个生命周期内有效，那么就需要实现 `finalize`，关闭这个链接。

使用 `finalize` 还需要注意一个事，调用 `super.finalize()`;



一个对象的 `finalize()` 方法只会被调用一次，而且 `finalize()` 被调用不意味着 `gc` 会立即回收该对象，所以有可能调用 `finalize()` 后，该对象又不需要被回收了，然后到了真正要被回收的时候，因为前面调用过一次，所以不会调用 `finalize()`，产生问题。所以，推荐不要使用 `finalize()` 方法，它跟析构函数不一样。

## 16. 接口、抽象类的区别

### 抽象类

抽象类是用来捕捉子类的通用特性的。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里子类的模板。以 JDK 中的 `GenericServlet` 为例：

1	
2	<code>public abstract class GenericServlet implements Servlet, ServletConfig, Serializable {</code>
3	<code>    // abstract method</code>
4	<code>    abstract void service(ServletRequest req, ServletResponse res);</code>
5	
6	<code>    void init() {</code>
7	<code>        // Its implementation</code>
8	<code>    }     // other method related to Servlet</code>
9	<code>}</code>

当 `HttpServlet` 类继承 `GenericServlet` 时，它提供了 `service` 方法的实现：

1	<code>public class HttpServlet extends GenericServlet {</code>
2	<code>    void service(ServletRequest req, ServletResponse res) {</code>



3	<code>        // implementation</code>
4	<code>    }</code>
5	<code>    protected void doGet(HttpServletRequest req, HttpServletResponse resp) {</code>
6	<code>        // Implementation</code>
7	<code>    }</code>
8	<code>    protected void doPost(HttpServletRequest req, HttpServletResponse resp) {</code>
9	<code>        // Implementation</code>
10	<code>    }</code>
11	<code>    // some other methods related to HttpServlet</code>
12	<code>}</code>
13	
14	
15	

## 接口

接口是抽象方法的集合。如果一个类实现了某个接口，那么它就继承了这个接口的抽象方法。这就像契约模式，如果实现了这个接口，那么就必须确保使用这些方法。接口只是一种形式，接口自身不能做任何事情。以 `Externalizable` 接口为例：

1	<code>public interface Externalizable extends Serializable {</code>
2	

3	void writeExternal(ObjectOutput out) throws IOException;
4	
5	void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
6	}

当你实现这个接口时，你就需要实现上面的两个方法：

1	
2	public class Employee implements Externalizable {
3	
4	int employeeId;
5	String employeeName;
6	
7	@Override
8	public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
9	employeeId = in.readInt();
10	employeeName = (String) in.readObject();
11	}
12	@Override
13	public void writeExternal(ObjectOutput out) throws IOException {
14	out.writeInt(employeeId);
15	out.writeObject(employeeName);
16	}

17

18

19

## 抽象类和接口的对比

参数	抽象类	接口
默认的方法实现	它可以有默认的方法实现	接口完全是抽象的。它根本不存在方法的实现
实现	子类使用 <code>extends</code> 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。	子类使用关键字 <code>implements</code> 来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
与正常Java类的区别	除了你不能实例化抽象类之外，它和普通Java类没有任何区别	接口是完全不同的类型
访问修饰符	抽象方法可以有 <code>public</code> 、 <code>protected</code> 和 <code>default</code> 这些修饰符	接口方法默认修饰符是 <code>public</code> 。你不可以使用其它修饰符。
main方法	抽象方法可以有main方法并且我们可以运行它	接口没有main方法，因此我们不能运行它。（java8以后接口可以有 <code>default</code> 和 <code>static</code> 方法，所以可以运行main方法）
多继承	抽象方法可以继承一个类和实现多个接口	接口只可以继承一个或多个其它接口
速度	它比接口速度要快	接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
添加新方法	如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。	如果你往接口中添加方法，那么你必须改变实现该接口的类。

## 什么时候使用抽象类和接口

如果你拥有一些方法并且想让它们中的一些有默认实现 ,那么使用抽象类吧。

如果你想实现多重继承 ,那么你必须使用接口。由于 **Java 不支持多继承** ,子类不能够继承多个类 ,但可以实现多个接口。因此你就可以使用接口来解决它。

如果基本功能在不断改变 ,那么就需要使用抽象类。如果不断改变基本功能并且使用接口 ,那么就需要改变所有实现了该接口的类。

### 什么时候使用抽象类？

抽象类让你可以定义一些默认行为并促使子类提供任意特殊化行为。

例如 :Spring 的依赖注入就使得代码实现了集合框架中的接口原则和抽象实现。

## Java8 中的默认方法和静态方法

Oracle 已经开始尝试向接口中引入默认方法和静态方法，以此来减少抽象类和接口之间的差异。现在，我们可以为接口提供默认实现的方法了并且不用强制子类来实现它。这类内容我将在下篇博客进行阐述。

## 从 java 容器类的设计讨论抽象类和接口的应用

除了前面提到的一个问题 :多态到底是用抽象类还是接口实现？我还看到有人评论说：现在都是提倡面向接口编程，使用抽象类的都被称为上世纪的老码农了。哈哈。看到这个说法我也是苦笑不得。不过，面向接口编程的确是一个趋势，java 8 已经支持接口实现默认方法和静态方法了，抽象类和接口之间的差异越来越小。闲话少说，我们开始讨论抽象类和接口的应用。

full\_container\_taxonomy

上图是 java 容器类的类继承关系。我们以容器类中 ArrayList 为例子来讨论抽象类和接口的应用。

## ArrayList 类继承关系

ArrayList

上图是 ArrayList 的类继承关系。可以看到，ArrayList 的继承关系中既使用了抽象类，也使用了接口。

- 最顶层的接口是 Iterable，表示这是可迭代的类型。所有容器类都是可迭代的，这是一个极高的抽象。
- 第二层的接口是 Collection，这是单一元素容器的接口。集合，列表都属于此类。
- 第三层的接口是 List，这是所有列表的接口。

通过三个接口，我们可以找到容器类的三个抽象特性，实现这些接口就意味着拥有这些接口的特性。

- AbstractCollection 实现了 Collection 中的部分方法。
- ArrayList 实现了 AbstractCollection 和 List 中的部分方法。

上面的抽象类提供了一些方法的默认实现，给具体类提供了复用代码。

## 纯抽象类实现

如果我们像一个老码农一样，用抽象类来实现上面的接口会有怎样的效果？那么，类图可能变成这样。

AbstractList

抽象类在这里存在着一个很大的问题，它不能多继承，在抽象的层次上没有接口高，也没有接口灵活。例如说：

```
List + AbstractCollection -> AbstractList
```

•

•

```
Set + AbstractCollection -> AbstractSet
```

•

单纯用抽象类无法实现像接口一样灵活的扩展。

## 纯接口实现

如果我们像一个新码农一样，用纯接口来实现呢？

```
InterfaceList
```

这样写理论上没有问题，实际写代码的时候问题就来了。所有的接口都要提供实现，于是你不得不在各个实现类中重复代码。

## 总结

经过上面的讨论，我们得出两个结论：

- 抽象类和接口并不能互相替代。
- 抽象类和接口各有不可替代的作用。

从容器类的类关系图中可以看到，接口主要是用来抽象类型的共性，例如说，容器的可迭代特性。抽象类主要是给具体实现类提供重用的代码，例如说，List 的一些默认方法。

## 抽象类和接口的使用时机

那么，什么时候该用抽象类，什么时候该用接口呢？

要解决上面的问题，我们先从弄清楚抽象类和接口之间的关系。首先，我们都知道类对事物的抽象，定义了事物的属性和行为。而抽象类是不完全的类，具有抽象方法。接口则比类的抽象层次更高。所以，我们可以这样理解它们之间的关系：  
**类是对事物的抽象，抽象类是对类的抽象，接口是对抽象类的抽象。**

从这个角度来看 java 容器类，你会发现，它的设计正体现了这种关系。不是吗？

从 Iterable 接口，到 AbstractList 抽象类，再到 ArrayList 类。

现在回答前面的问题：在设计类的时候，首先考虑用接口抽象出类的特性，当你发现某些方法可以复用的时候，可以使用抽象类来复用代码。简单说，**接口用于抽象事物的特性，抽象类用于代码复用。**

当然，不是所有类的设计都要从接口到抽象类，再到类。程序设计本就没有绝对的范式可以遵循。上面的说法只是提供一个角度来理解抽象类和接口的关系，每个人都会有自己的理解，有人认为两者一点关系都没有，这也有道理。总之，模式和语法是死的，人是活的。