# Java核心内容

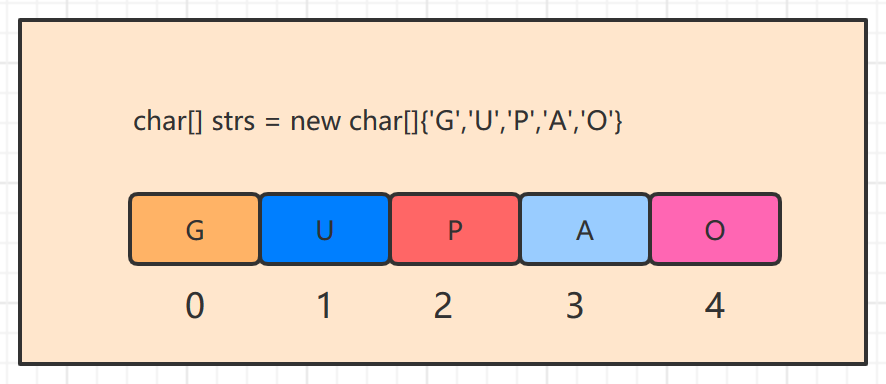
# 集合

Java中用来存储数据的结构有哪些呢？接下来我们分别给大家来介绍下Java中常用的用来存储数据的结构，并分别来介绍下各自的优缺点

## 常见数据结构

### 数组

如图，这就是我们给出的一个简单的数组



特点：

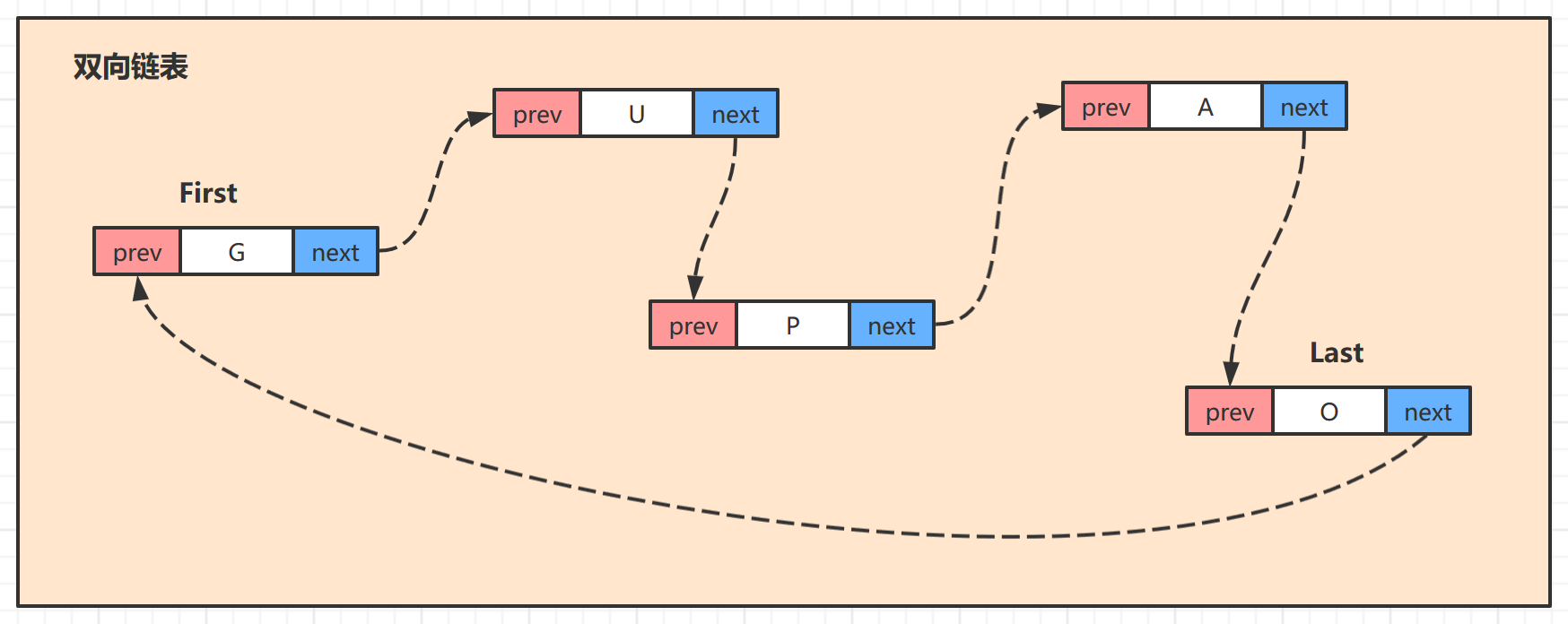
A.内存地址连续，

B.可以通过下标的成员访问,下标访问的性能高

C.**增删**操作带来更大的性能消耗(保证数据越界的问题,需动态扩容)

### 链表

此处我们主要来介绍下双向链表。



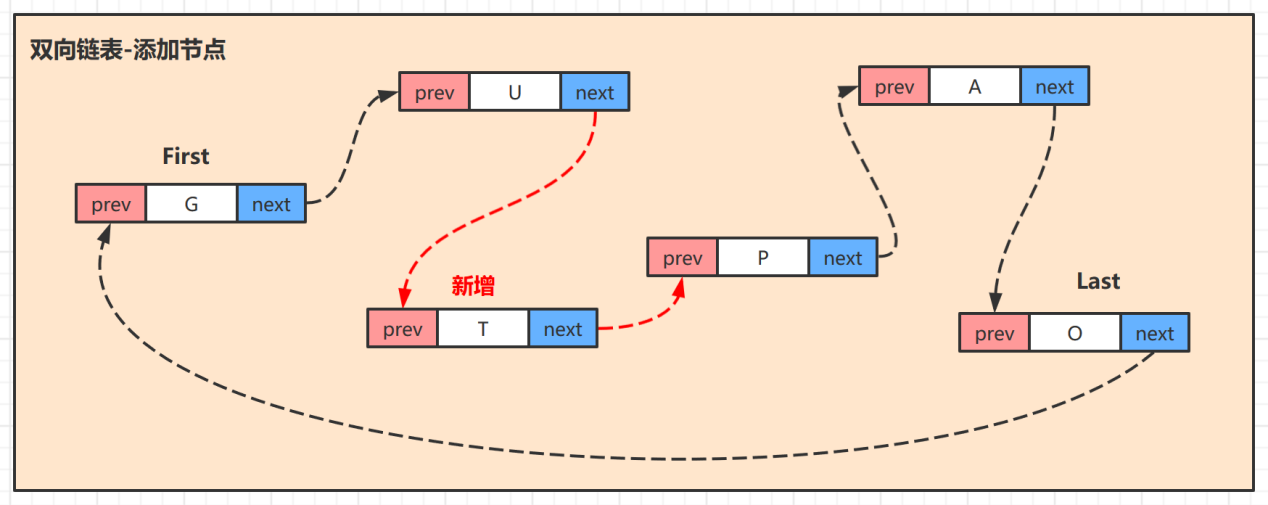
特点：

A.灵活的空间要求,存储空间不要求连续

B.不支持下标的访问.支持顺序的遍历搜索

C.针对增删操作找到对应的节点改变链表的头尾指向即可,无需移动元素存储位置.

添加节点



删除节点



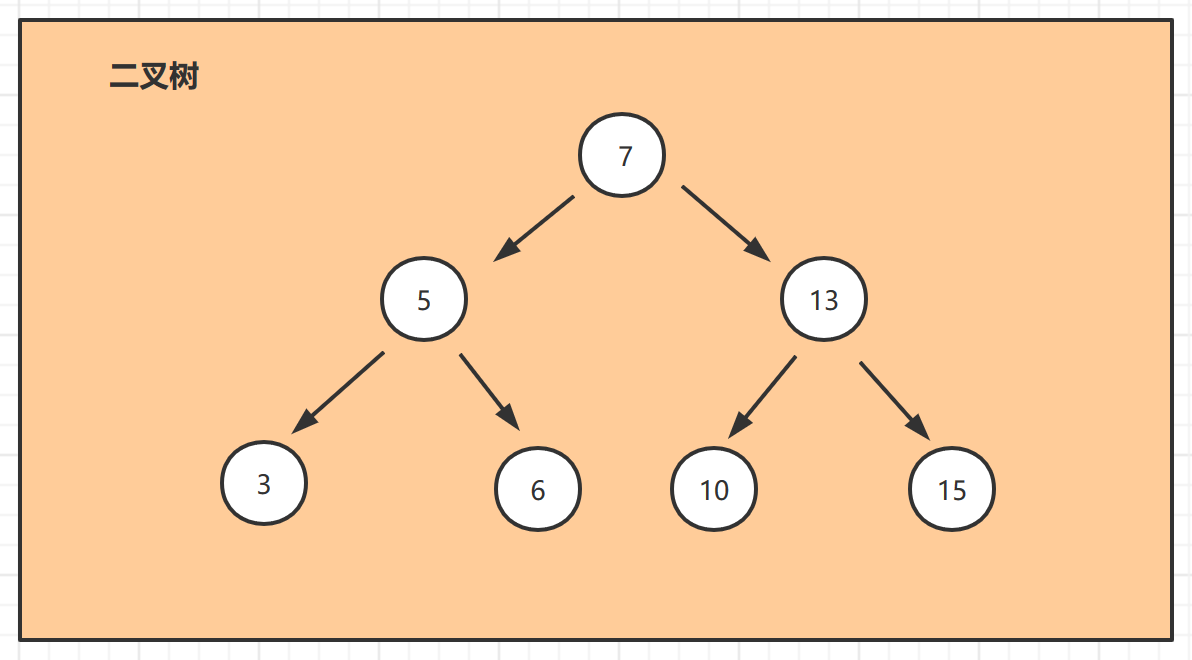
典型的Java集合类LinkedList

### 树

#### 3.1二叉树



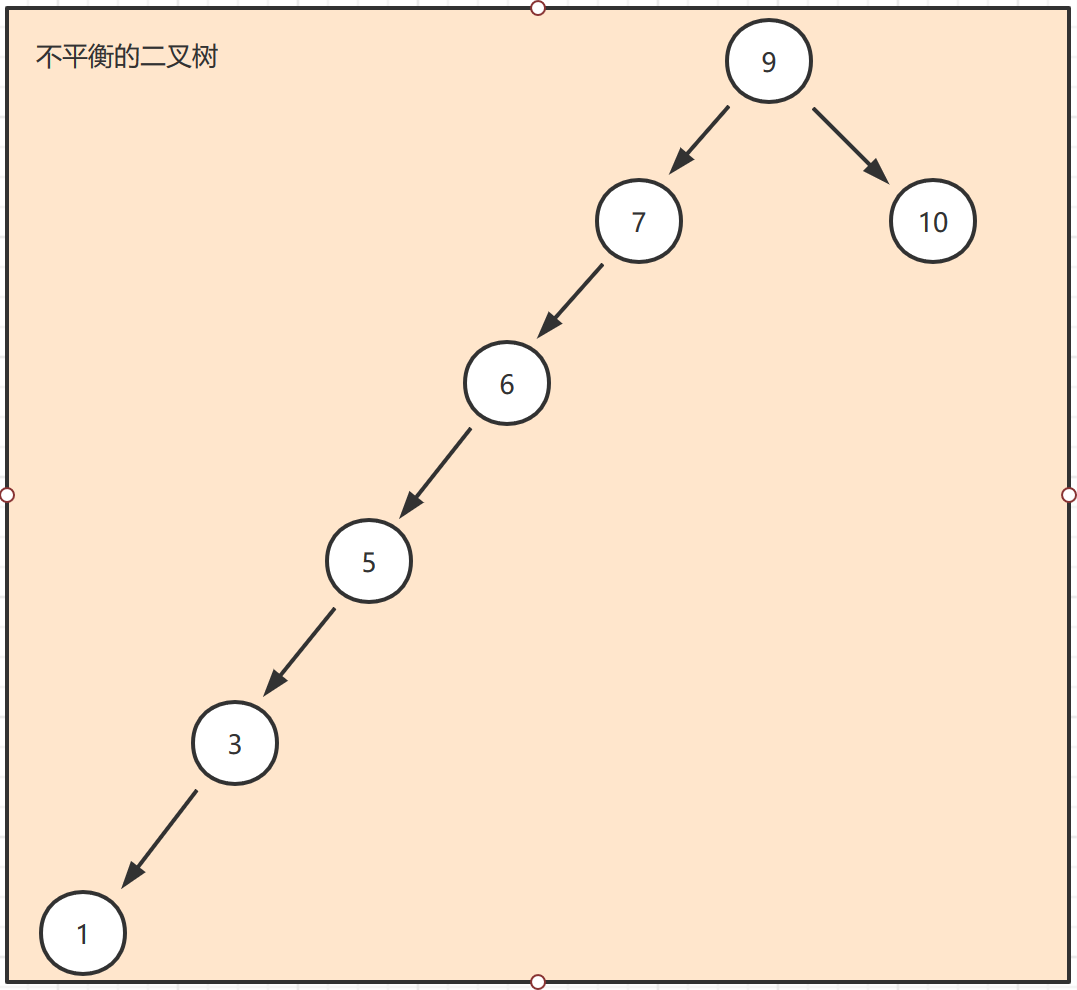
下图是一个简单的二叉树结构



二叉树具有如下的特点

1. 某节点的左子树节点值仅包含小于该节点值
2. 某节点的右子树节点值仅包含大于该节点值
3. 左右子树每个也必须是二叉查找树
4. 顺序排列

不平衡二叉树



查询效率不高，

面对这个问题我们可以参考在高中学习生物时学过一个关键字**去除顶端优势**，通过去除植物顶端优势，侧芽会迅速生长，慢慢变得强壮和平衡， **红黑树**其实就是**去除二叉查找树顶端优势的解决方案，从而达到树的平衡**

#### 3.2红黑树

红黑树，Red-Black Tree 「RBT」是一个**自平衡**(不是绝对的平衡)的二叉查找树(BST)，树上的每个节点都遵循下面的规则:

· 性质1：每个节点要么是黑色，要么是红色。

· 性质2：根节点是黑色。

· 性质3：每个叶子节点（NIL）是黑色。

· 性质4：每个红色结点的两个子结点一定都是黑色。

· **性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。**

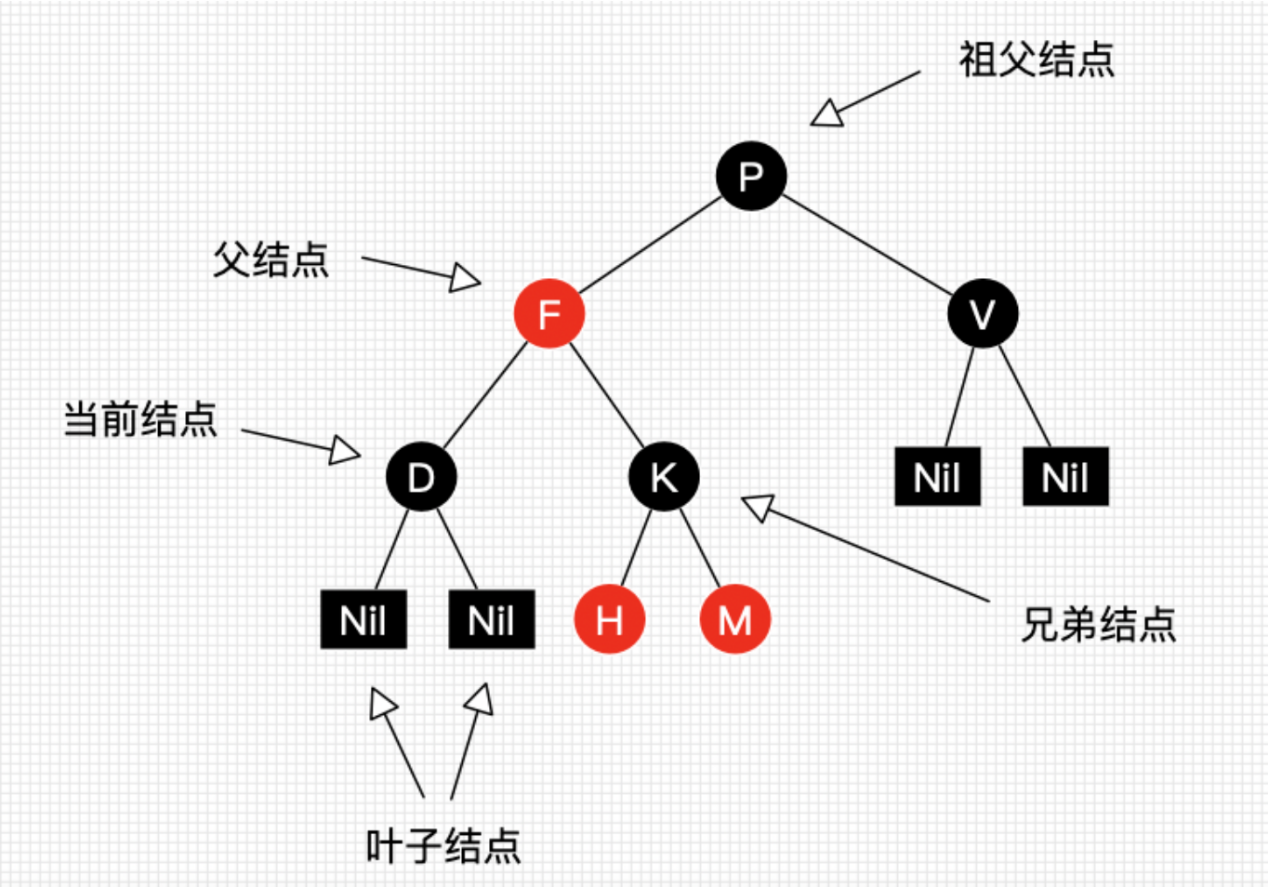
瞬间懵逼？了解一下印象就行，开始玩魔方都是要照着魔方公式一点点玩的，多玩几次就熟悉了。红黑树也一样，红黑树有两大操作:

recolor (重新标记黑色或红色)

rotation (旋转，这是树达到平衡的关键)

红黑树练习网站: <http://algoanim.ide.sk/index.php?page=showanim&id=63>

节点描述约定



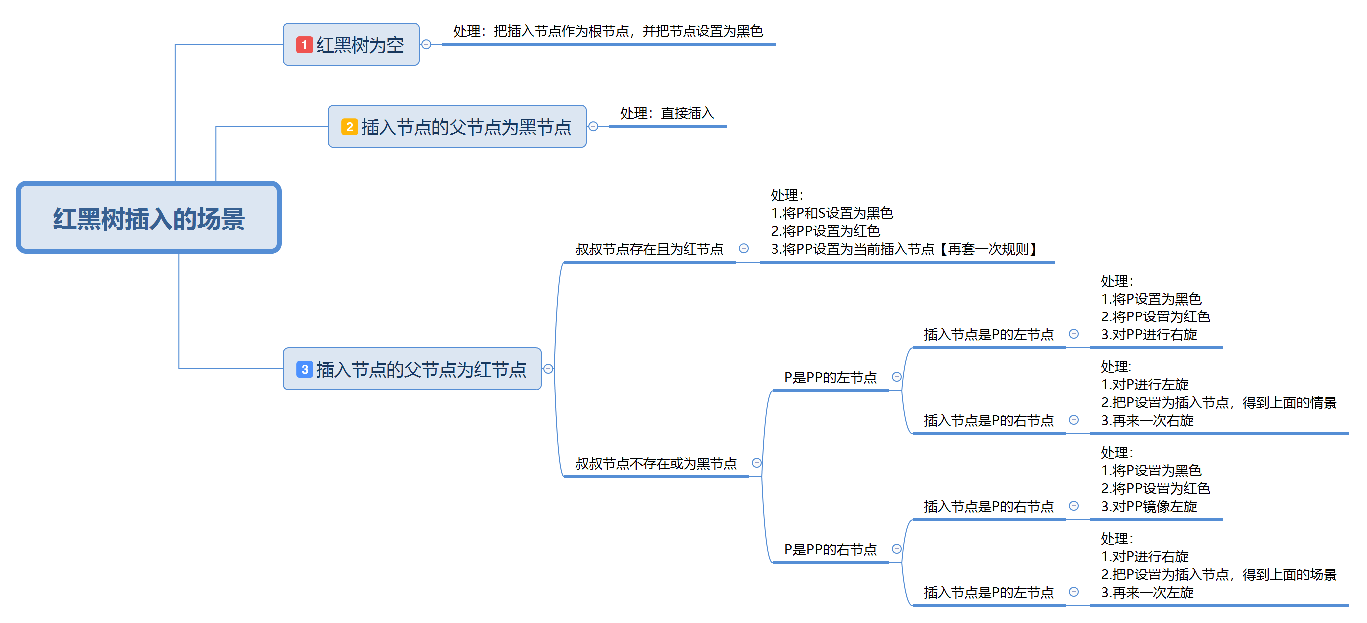
红黑树能自平衡，它靠的是什么？三种操作：左旋、右旋和变色

· **左旋**：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。

· **右旋**：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。

· **变色**：结点的颜色由红变黑或由黑变红。

插入处理的场景



插入情景1：红黑树为空树

最简单的一种情景，直接把插入结点作为根结点就行，但注意，根据红黑树性质2：根节点是黑色。还需要把插入结点设为黑色。

处理：把插入结点作为根结点，并把结点设置为黑色。

插入情景2：插入结点的父结点为黑结点

由于插入的结点是红色的，当插入结点的黑色时，并不会影响红黑树的平衡，直接插入即可，无需做自平衡。

处理：直接插入。

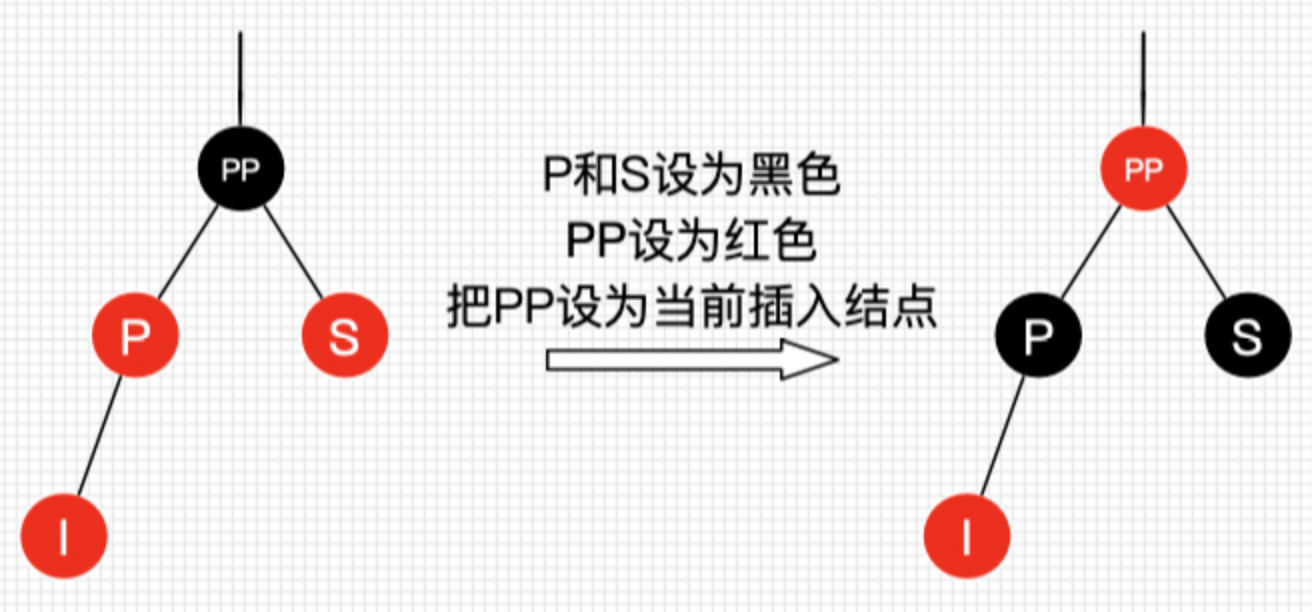
插入情景3：插入结点的父结点为红结点

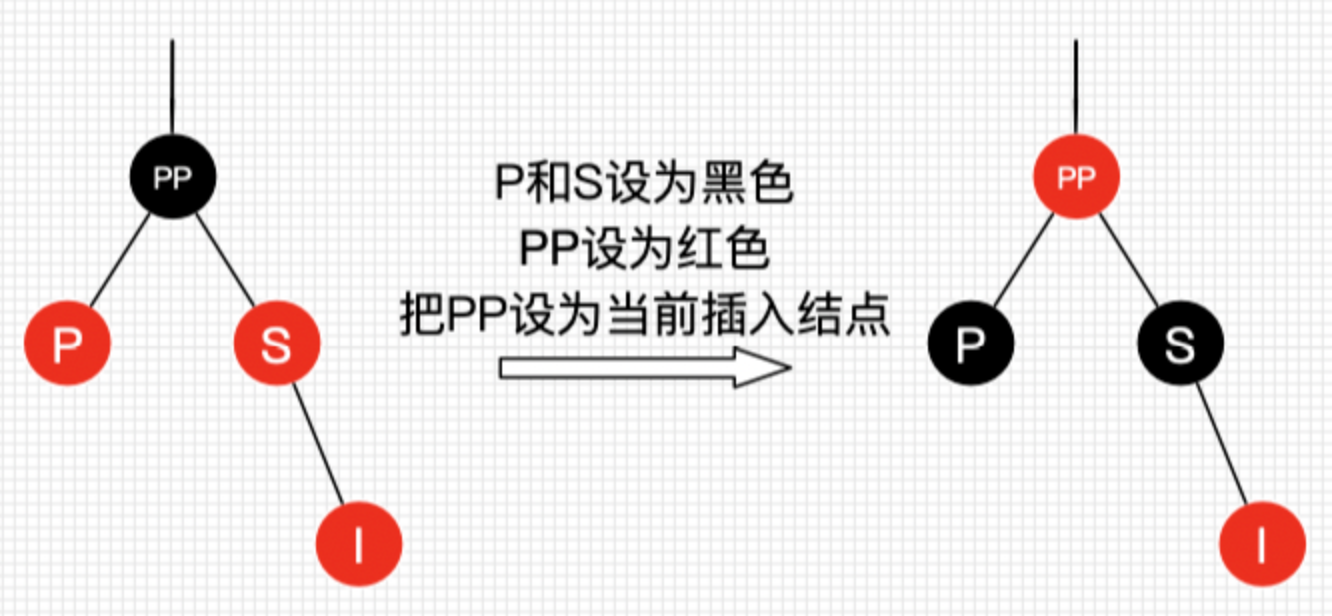
再次回想下红黑树的性质2：根结点是黑色。如果插入的父结点为红结点，那么该父结点不可能为根结点，所以插入结点总是存在祖父结点。这点很重要，因为后续的旋转操作肯定需要祖父结点的参与。

情景3又分为很多子情景，下面将进入重点部分

插入情景3.1：叔叔结点存在并且为红结点

从红黑树性质4可以，祖父结点肯定为黑结点，因为不可以同时存在两个相连的红结点。那么此时该插入子树的红黑层数的情况是：黑红红。显然最简单的处理方式是把其改为：红黑红。



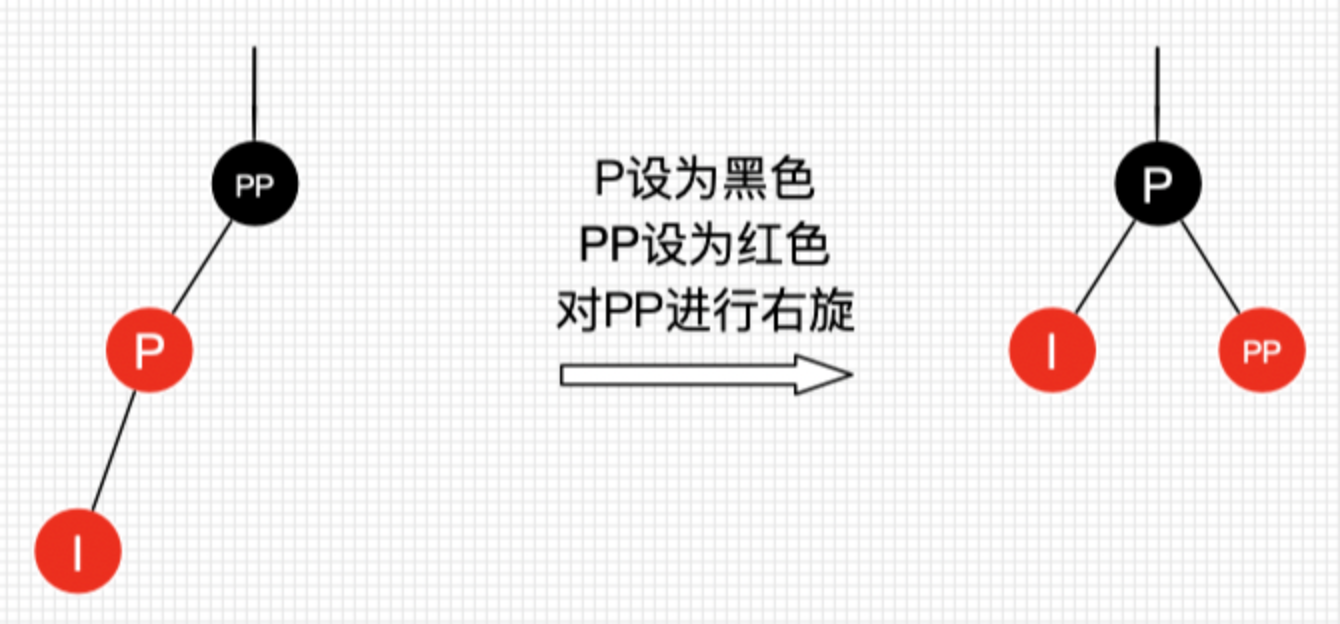


插入情景3.2：叔叔结点不存在或为黑结点，并且插入结点的父亲结点是祖父结点的左子结点

单纯从插入前来看，也即不算情景3.1自底向上处理时的情况，叔叔结点非红即为叶子结点(Nil)。因为如果叔叔结点为黑结点，而父结点为红结点，那么叔叔结点所在的子树的黑色结点就比父结点所在子树的多了，这不满足红黑树的性质5。后续情景同样如此，不再多做说明了。

前文说了，需要旋转操作时，肯定一边子树的结点多了或少了，需要租或借给另一边。插入显然是多的情况，那么把多的结点租给另一边子树就可以了。

插入情景3.2.1：插入结点是其父结点的左子结点

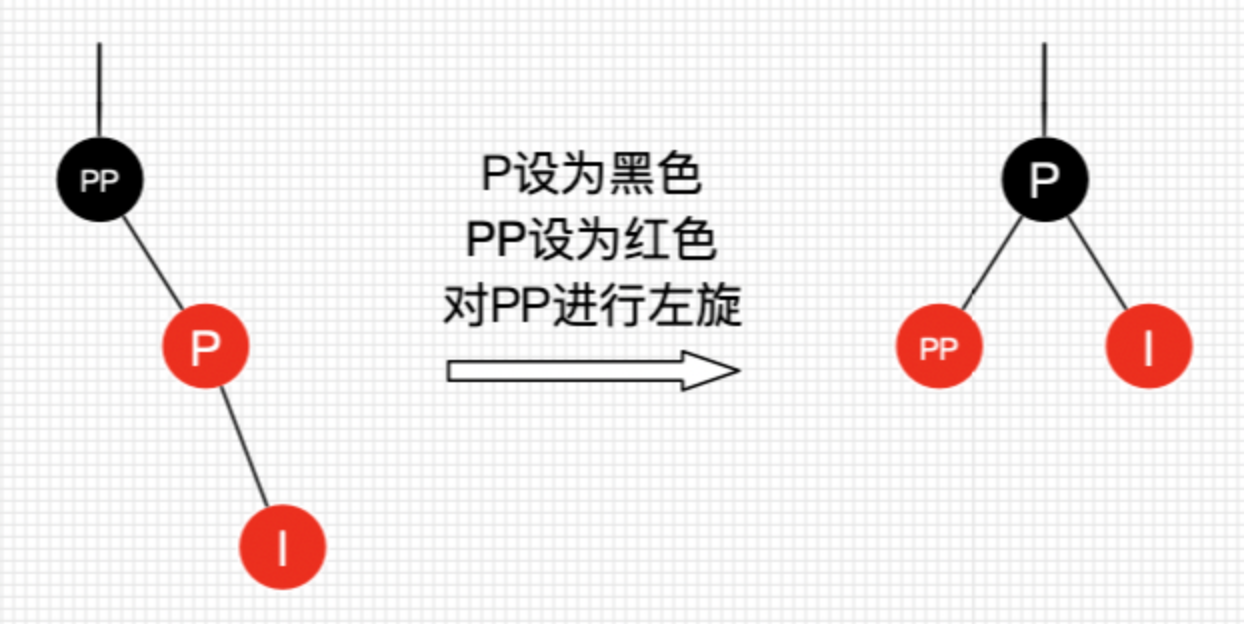


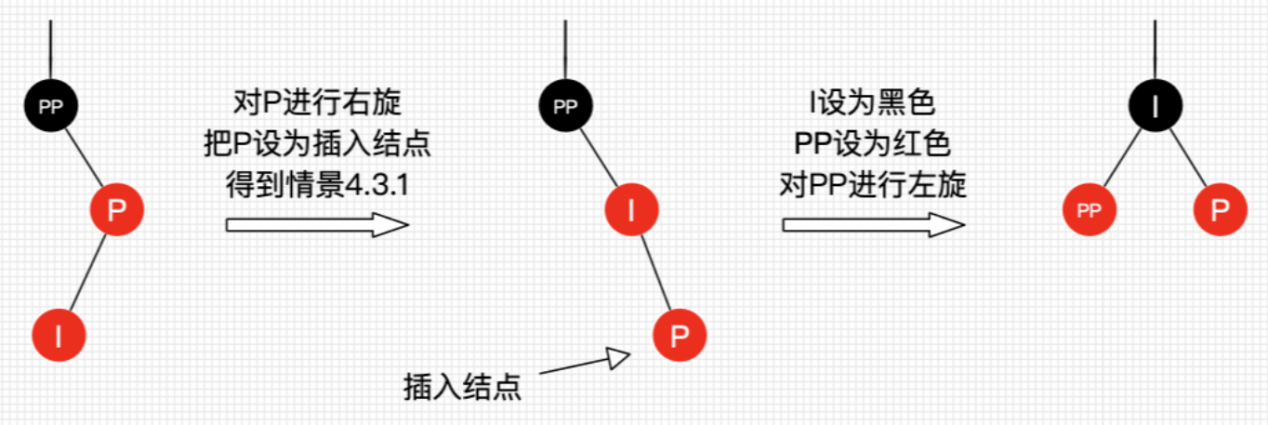
插入情景3.2.2：插入结点是其父结点的右子结点

这种情景显然可以转换为情景3.2.1



**插入情景3.3：叔叔结点不存在或为黑结点，并且插入结点的父亲结点是祖父结点的右子结点**  
该情景对应情景3.2，只是方向反转，不做过多说明了，直接看图

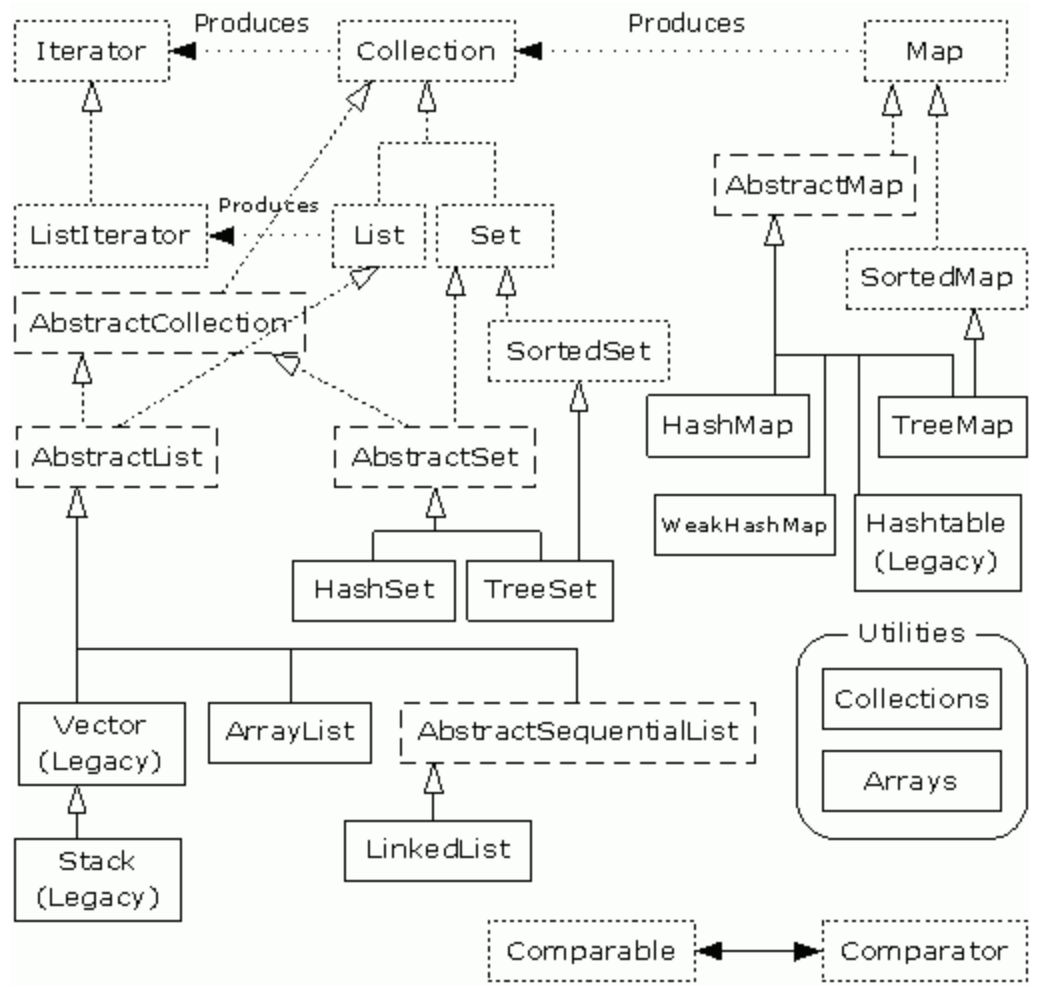




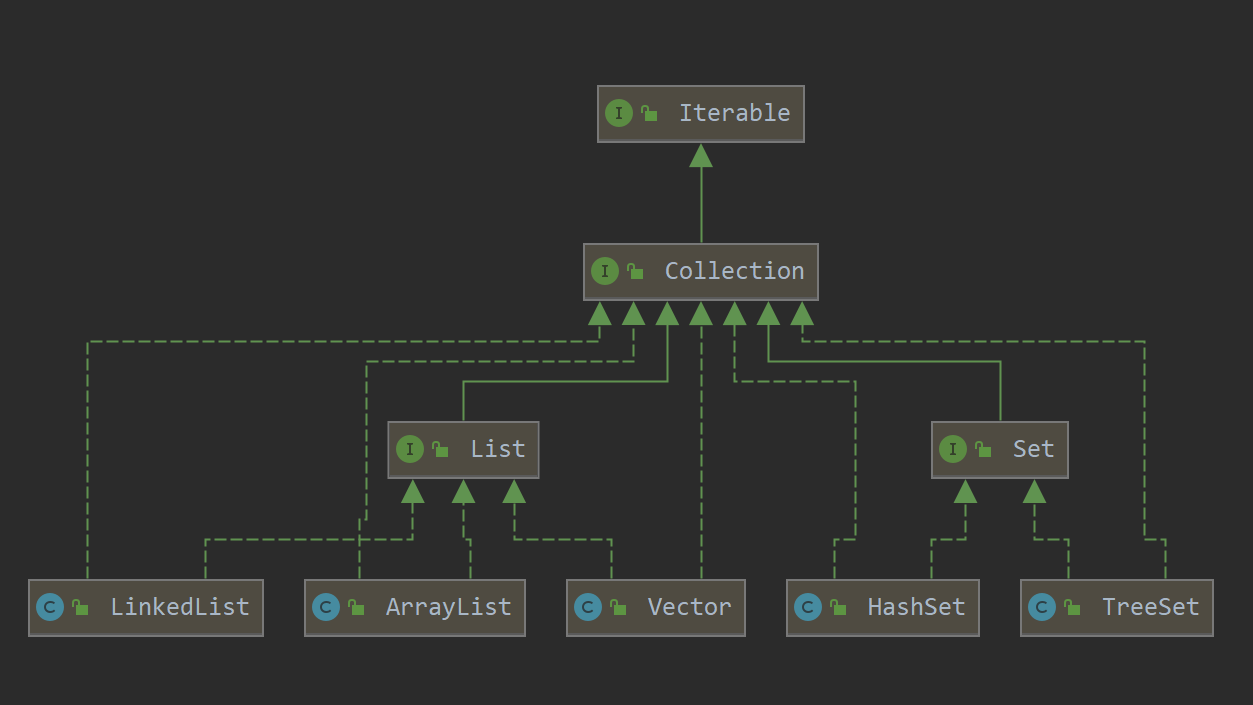
了解了常见的数据存储结构后我们就可以来看看Java中的集合有哪些具体的实现。

## 集合介绍

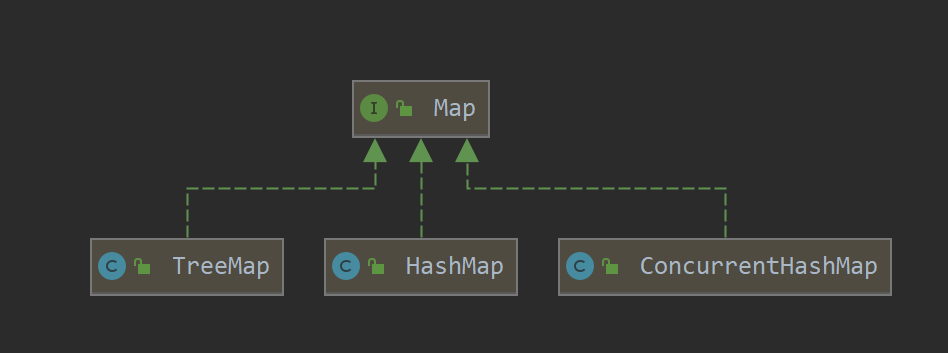
### 集合类图结构



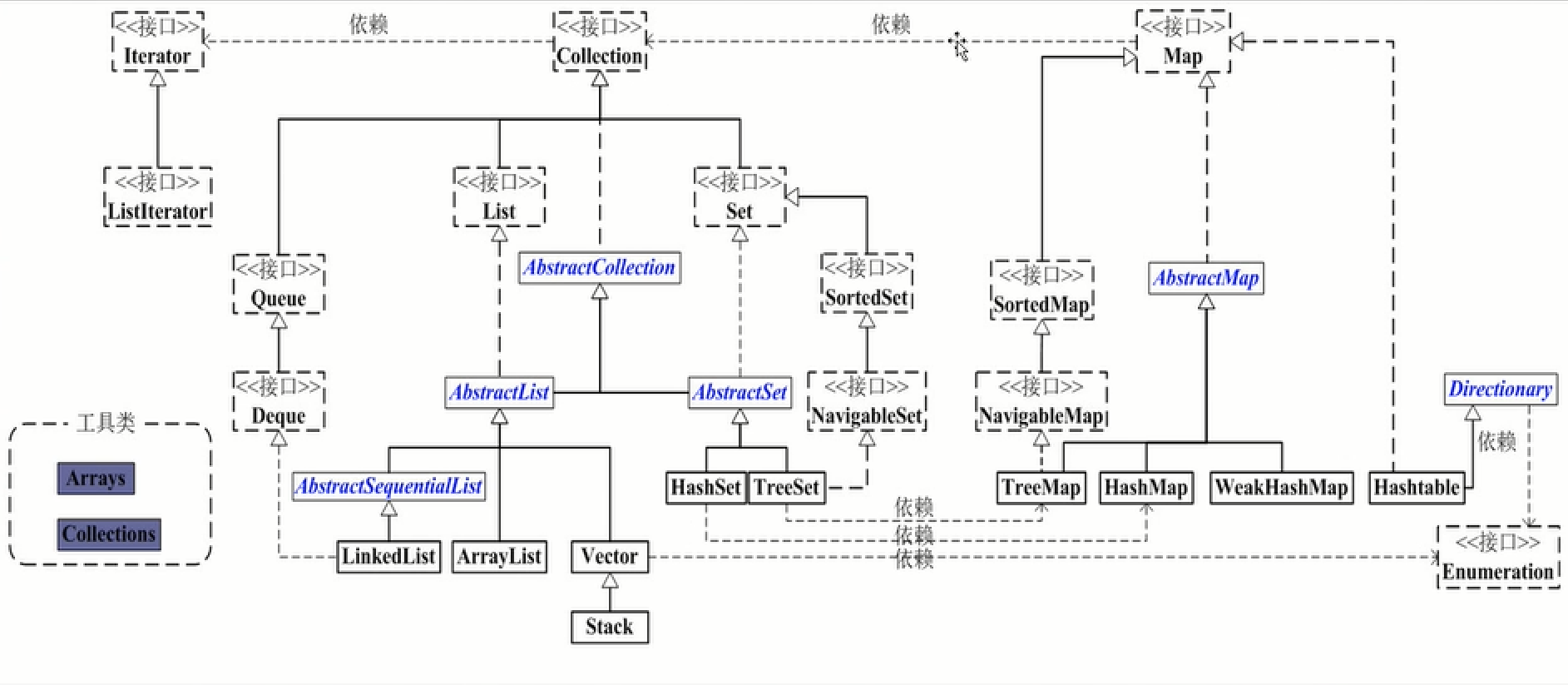
### Collection接口介绍



### Map接口介绍



### 4.两大派系相互之间的关系



HashSet本质上是一个HashMap

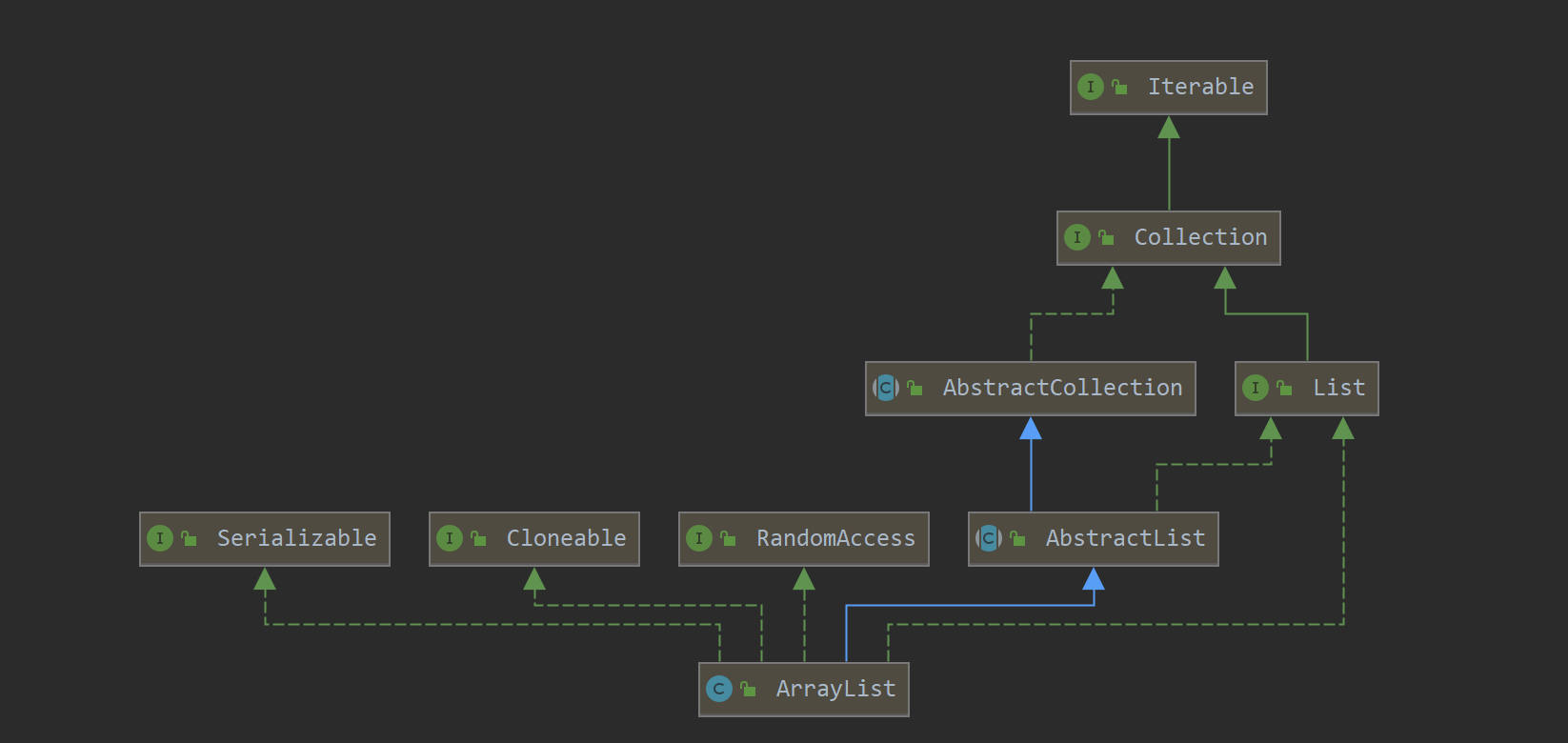
TreeSet本质上是一个NavigableMap

源码课论证

## Collection接口

### List接口

#### ArrayList源码分析



相关的接口抽象类的介绍

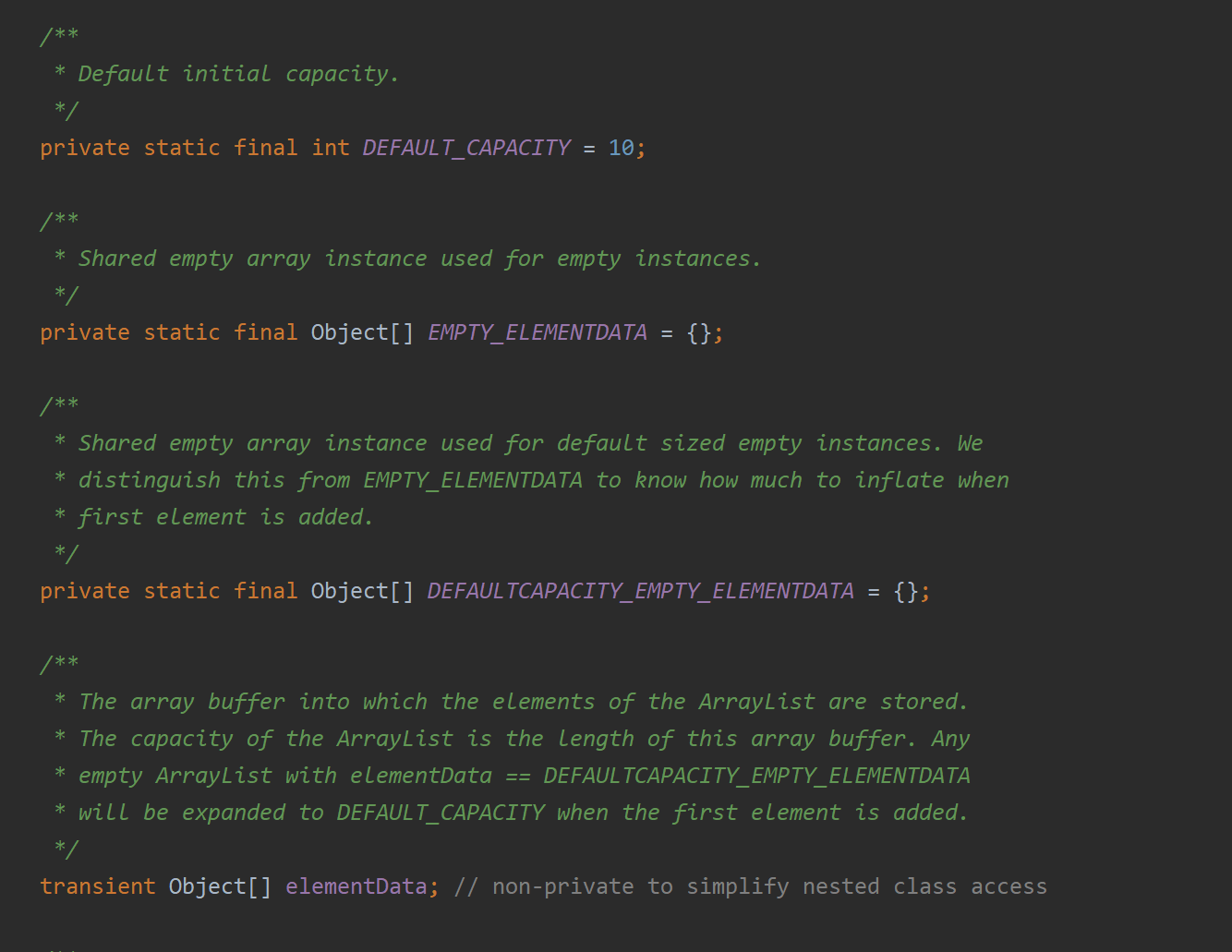
|  |  |
| --- | --- |
| 类名 | 说明 |
| AbstractCollection | 实现了Collection中大量的函数,除了特定的几个函数iterator()和size()之外的函数 |
| AbstractList | 该接口继承于AbstractCollection，并且实现List接口的抽象类。  它实现了List中除size()、get(int location)之外的函数。  AbstractList的主要作用：它实现了List接口中的大部分函数  和AbstractCollection相比，AbstractList抽象类中，实现了iterator()接口 |
| RandomAccess | RandmoAccess接口，即提供了随机访问功能。RandmoAccess是java中用来被List实现，为List提供快速访问功能的。在ArrayList中，我们即可以通过元素的序号快速获取元素对象；这就是快速随机访问 |
| List | 有序队列接口,提供了一些通过下标访问元素的函数  List是有序的队列，List中的每一个元素都有一个索引；第一个元素的索引值是0，往后的元素的索引值依次+1 |

ArrayList本质是动态数组。那么ArrayList是如何实现动态扩容的呢?

ArrayList的长度

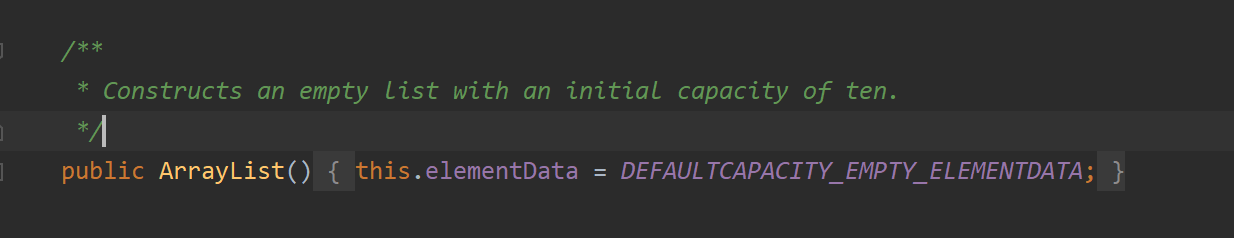


ArrayList中存储数据的数组对象



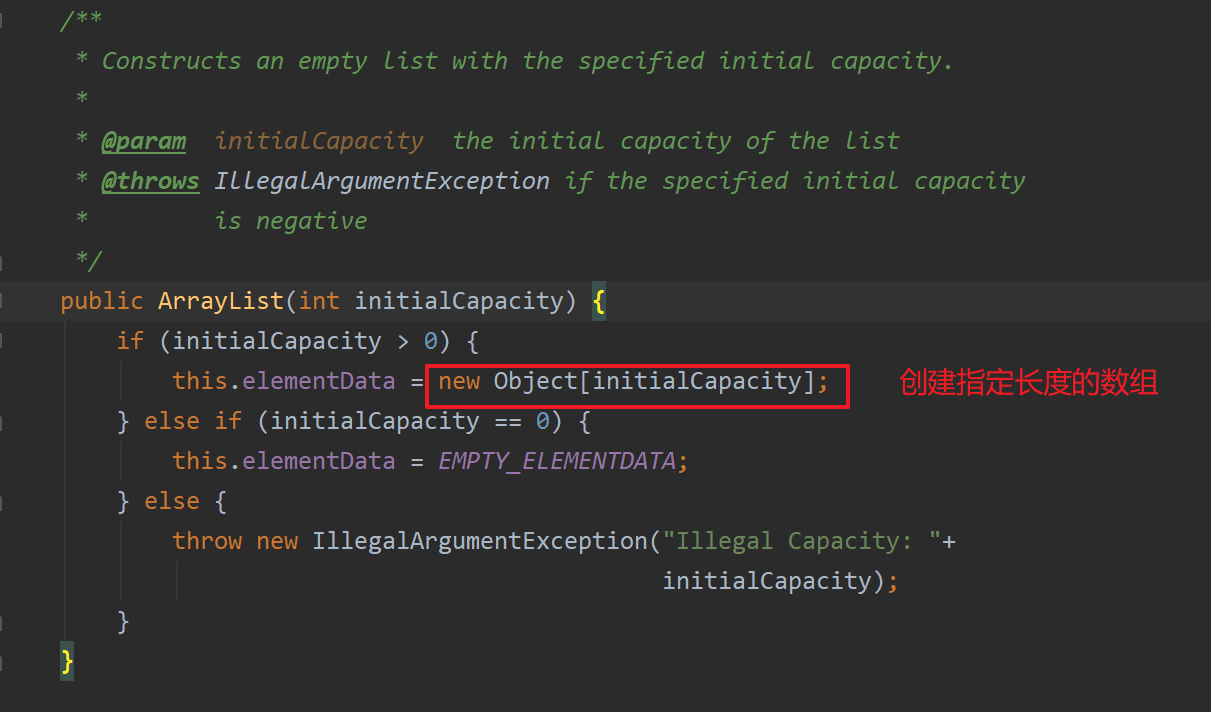
ArrayList的初始化

**无参构造器**



elementData=null;无参构造去并不会真实的创建数组，数组会在add方法中去创建，有助于性能的提升，懒加载的方式。

**有参构造器**



添加数据分析

add方法

|  |
| --- |
| */\*\*  \* 将元素添加到集合的尾部  \*  \** ***@param*** *e element to be appended to this list  \** ***@return*** *<tt>true</tt> (as specified by {****@link*** *Collection#add})  \*/* public boolean add(E e) {  // 确定容量，动态扩容  ensureCapacityInternal(size + 1); // Increments modCount!!  // 数组扩容后将数据添加到合适的位置 并size+1  elementData[size++] = e;  return true; } |

EnsureCapacityInternal

|  |
| --- |
| private void ensureCapacityInternal(int minCapacity) {  // 如果是第一次 true  if (elementData == *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*) {  // 第一次为10  minCapacity = Math.*max*(*DEFAULT\_CAPACITY*, minCapacity);  }  // 确保明确的容量，扩容  ensureExplicitCapacity(minCapacity); } |

ensureExplicitCapacity

|  |
| --- |
| private void ensureExplicitCapacity(int minCapacity) {  modCount++; // 记录修改的次数   // 第一次 10 - 0 > 0  if (minCapacity - elementData.length > 0)  // 扩容  grow(minCapacity); } |

grow:扩容的方法

|  |
| --- |
| private void grow(int minCapacity) {  // 记录原来的容量  int oldCapacity = elementData.length;  // 计算新的容量 新容量为 老容量的1.5倍 第一次为0  int newCapacity = oldCapacity + (oldCapacity >> 1);  // 0 - 10 < 0  if (newCapacity - minCapacity < 0)  // newCapacity = 10  newCapacity = minCapacity;  // 10 - 2亿 > 0 false  if (newCapacity - *MAX\_ARRAY\_SIZE* > 0)  newCapacity = *hugeCapacity*(minCapacity);  // 把原来数组中的内容拷贝到一个新建的指定容量为newCapacity的数组中，扩容  elementData = Arrays.*copyOf*(elementData, newCapacity); } |

get方法：获取集合中的元素

|  |
| --- |
| public E get(int index) {  // 检查下标是否合法  rangeCheck(index);  // 通过下标从数组中返回对应的元素  return elementData(index); } |

|  |
| --- |
| private void rangeCheck(int index) {  if (index >= size)  throw new IndexOutOfBoundsException(outOfBoundsMsg(index)); } |

set方法：修改集合中的元素

|  |
| --- |
| public E set(int index, E element) {  rangeCheck(index);   E oldValue = elementData(index);  elementData[index] = element;  return oldValue; } |

remove方法：移除指定的元素

|  |
| --- |
| public E remove(int index) {  rangeCheck(index);   modCount++;  E oldValue = elementData(index);  // 记录要移动的元素的个数  int numMoved = size - index - 1;  if (numMoved > 0)  // 源数组 开始下标 目标数组 开始下标 长度  System.*arraycopy*(elementData, index+1, elementData, index,  numMoved);  elementData[--size] = null; // clear to let GC do its work   return oldValue; } |

add(index,ele)方法：

|  |
| --- |
| public void add(int index, E element) {  rangeCheckForAdd(index);   ensureCapacityInternal(size + 1); // Increments modCount!!  System.*arraycopy*(elementData, index, elementData, index + 1,  size - index);  elementData[index] = element;  size++; } |

FailFast机制

快速失败机制,是java集合类应对并发访问在对集合进行迭代过程中,内部对象结构发生变化一种防护措施.这种错误检测的机制为这种有可能发生错误,通过抛出**java.util.ConcurrentModificationException**

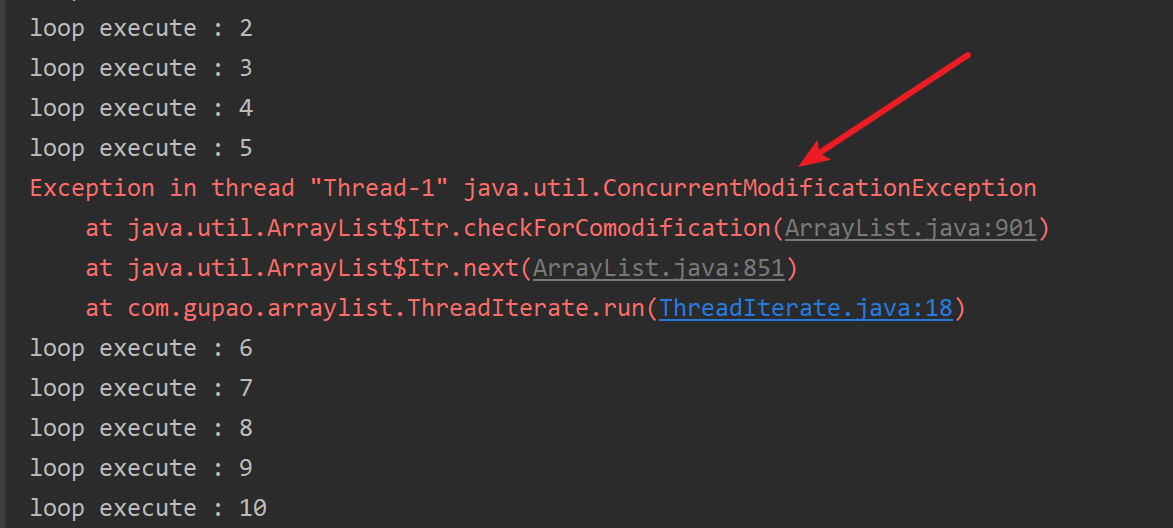
**效果：**

|  |
| --- |
| package com.gupao.arraylist;  import java.util.Iterator; import java.util.List;  public class ThreadIterate extends Thread {   private List list;   public ThreadIterate(List list){  this.list = list;  }   @Override  public void run() {  while(true){  for (Iterator iteratorTmp = list.iterator();iteratorTmp.hasNext();) {  iteratorTmp.next();  try {  Thread.*sleep*(5);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } } |

|  |
| --- |
| package com.gupao.arraylist;  import java.util.ArrayList; import java.util.List;  public class ThreadMain {   private static List *list* = new ArrayList();   public static void main(String[] args) {  new ThreadAdd(*list*).start();  new ThreadIterate(*list*).start();  } } |

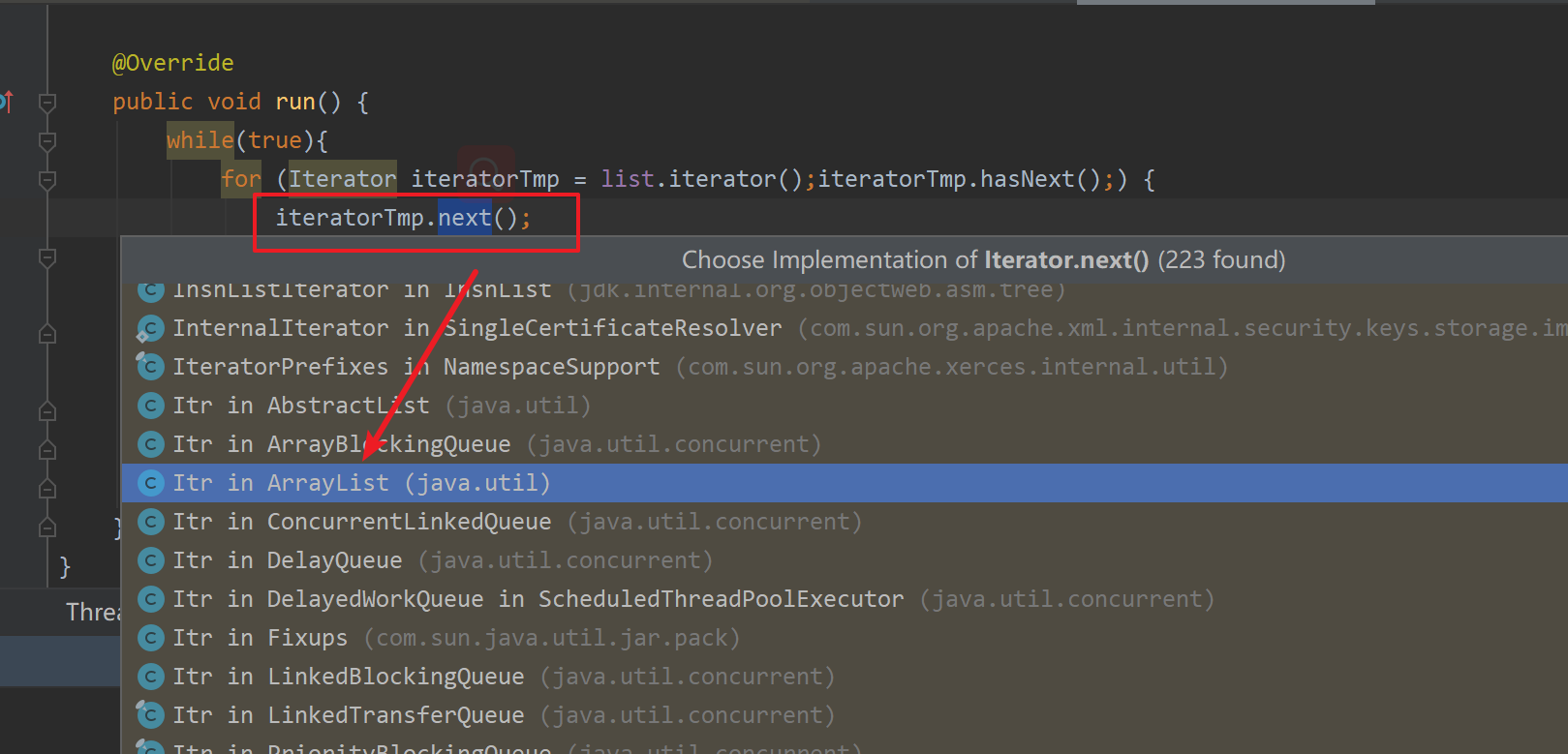
|  |
| --- |
| package com.gupao.arraylist;  import java.util.List;  public class ThreadAdd extends Thread{   private List list;   public ThreadAdd(List list){  this.list = list;  }   public void run(){  for (int i = 0; i < 100; i++) {  System.*out*.println("loop execute : " + i);  try {  Thread.*sleep*(5);  list.add(i);  }catch (Exception e){   }  }  } } |

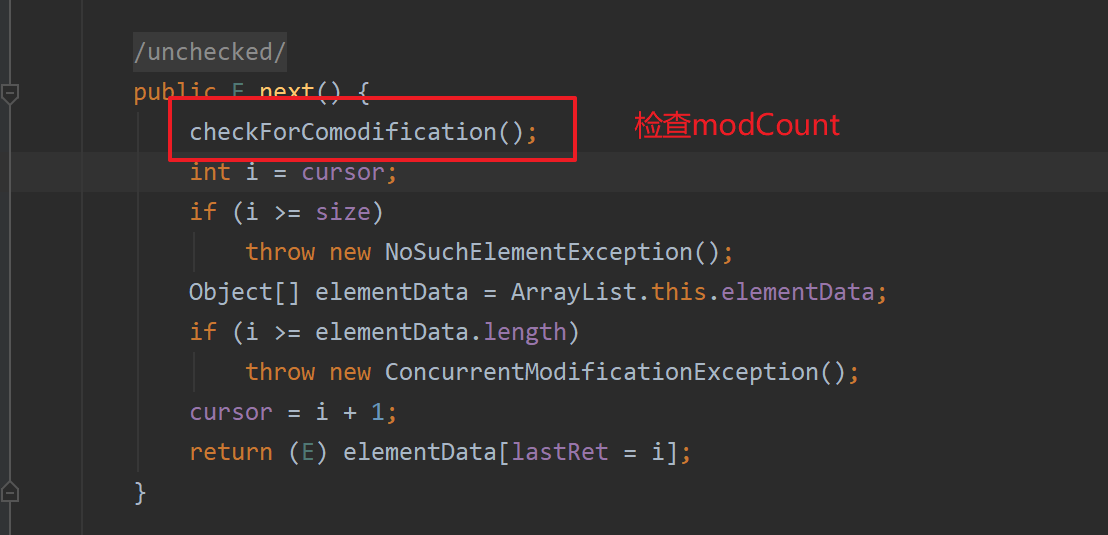
效果

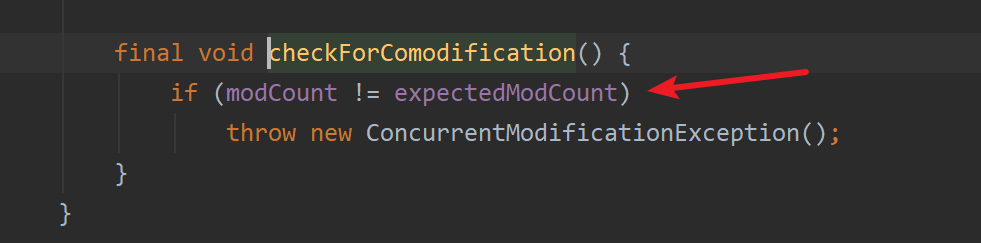


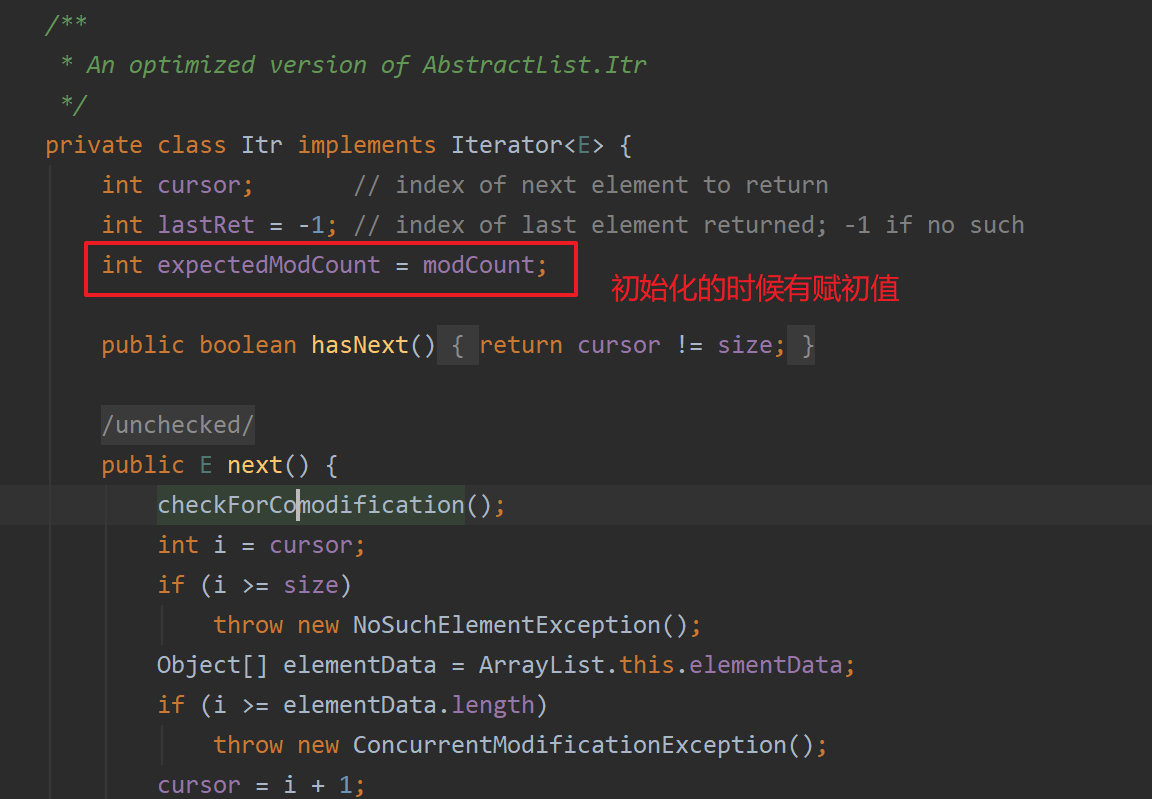
原因：



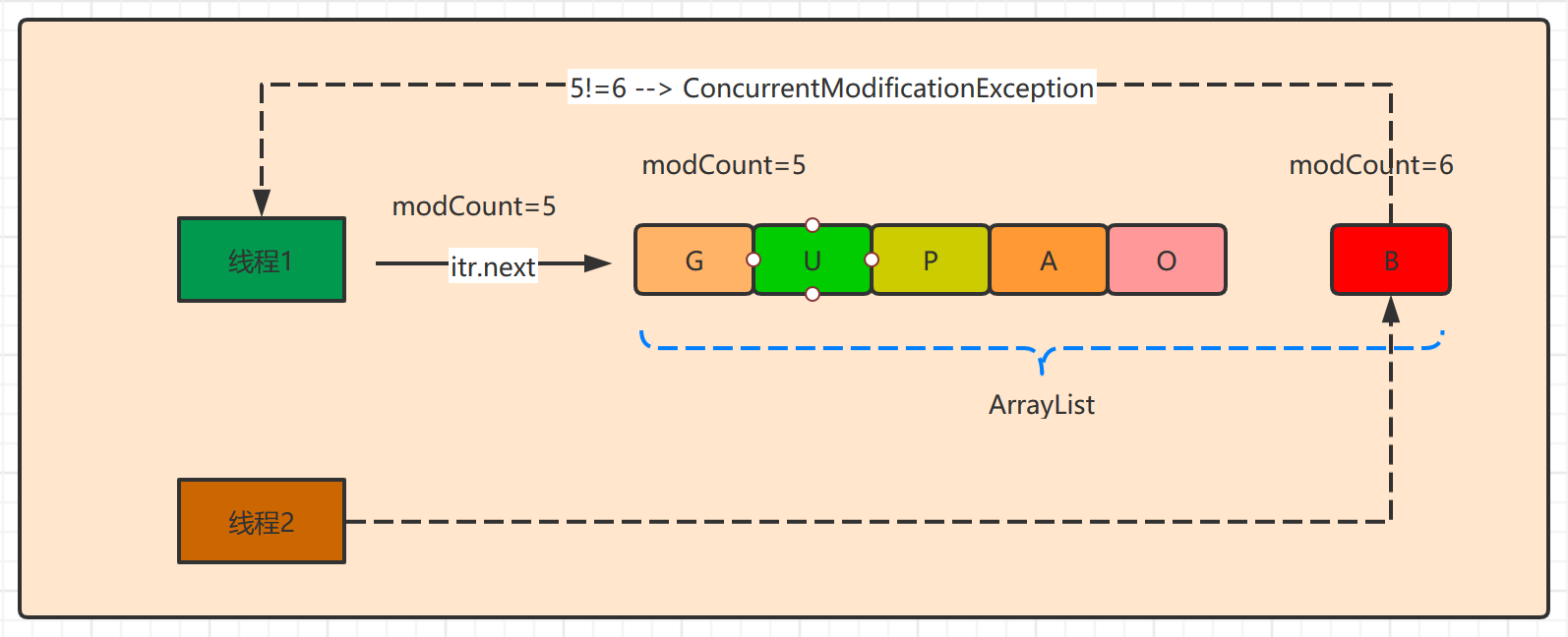






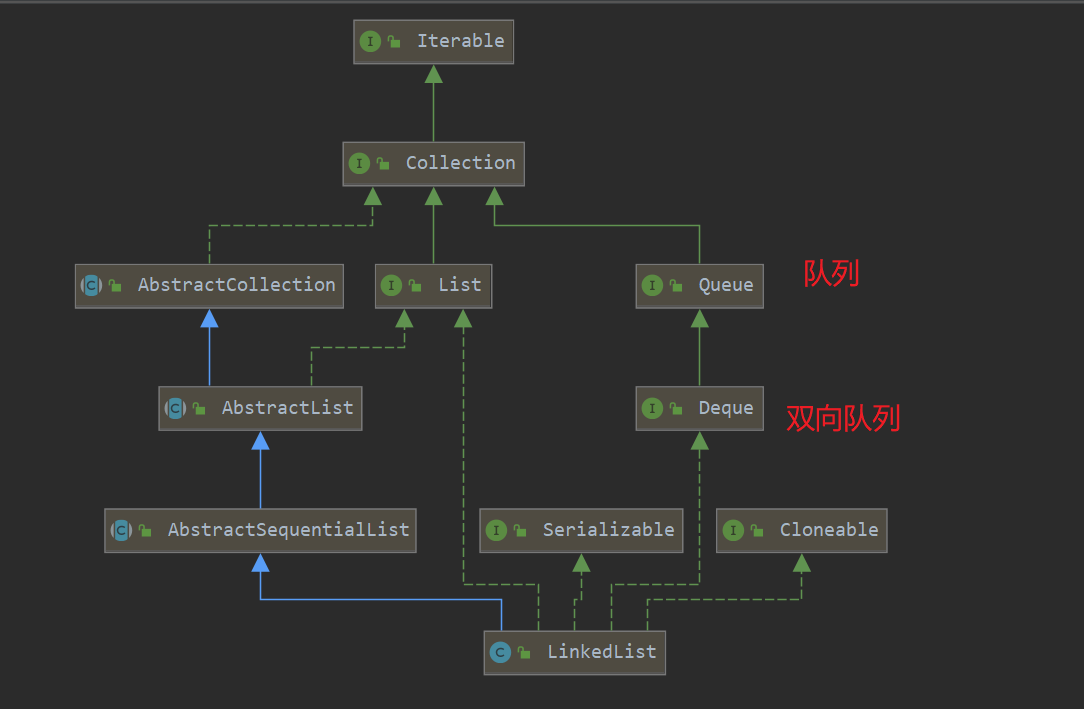


这是就ArrayList的failFast机制，也就是在多线程操作一个List数据的时候如果出现数据安全问题会直接抛异常。



#### LinkedList源码分析

接下来我们看看List接口中基于双向链表的实现LinkedList。首先来看看类图结构

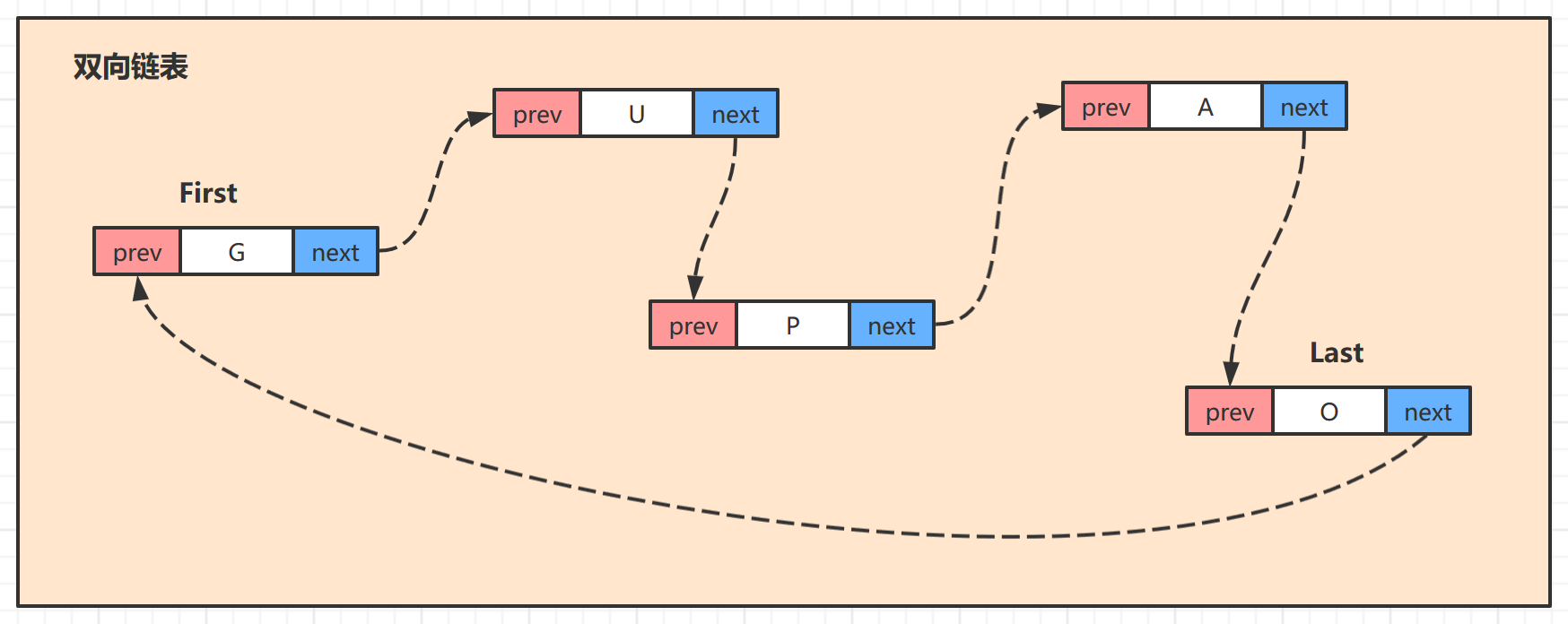


LinkedList是通过双向链表去实现的,他的数据结构具有双向链表结构的优缺点

既然是双向链表，那么它的顺序访问会非常高效，而随机访问效率比较低。

它包含一个非常重要的私有的静态内部类：**Node**。

|  |
| --- |
| private static class Node<E> {  E item; // 存储的元素  Node<E> next; // 下一个Node节点  Node<E> prev; // 前一个Node节点   Node(Node<E> prev, E element, Node<E> next) {  this.item = element;  this.next = next;  this.prev = prev;  } } |



重要的属性：

|  |
| --- |
| transient int size = 0; // 链表的长度  */\*\*  \* Pointer to first node.  \* Invariant: (first == null && last == null) ||  \* (first.prev == null && first.item != null)  \*/* transient Node<E> first; // 链表的首节点  */\*\*  \* Pointer to last node.  \* Invariant: (first == null && last == null) ||  \* (last.next == null && last.item != null)  \*/* transient Node<E> last; // 链表的尾节点 |

由于LinkedList实现了List的接口,所有必然具备List的特性.那我们先来看看List接口中定义的一个方法get(int index)和set(int index,E e);

get(index):

|  |
| --- |
| public E get(int index) {  checkElementIndex(index); // 检查下标是否合法  return node(index).item; } |

|  |
| --- |
| Node<E> node(int index) {  // assert isElementIndex(index);  // 判断index是否小于size的一般  if (index < (size >> 1)) {  Node<E> x = first;  // 从头开始遍历  for (int i = 0; i < index; i++)  x = x.next;  return x;  } else {  Node<E> x = last;  // 从尾部开始遍历  for (int i = size - 1; i > index; i--)  x = x.prev;  return x;  } } |

set(int index,E e)

|  |
| --- |
| public E set(int index, E element) {  checkElementIndex(index); // 检查下标是否越界  Node<E> x = node(index); // 根据下标获取对应的Node节点   E oldVal = x.item; // 记录原来的值  x.item = element; // 修改  return oldVal; // 返回原来的值 } |

|  |
| --- |
| public void addFirst(E e) {  linkFirst(e); } |

|  |
| --- |
| 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++; } |

#### Vector

Vector的底层与我们的ArrayList类似.都是以动态数组的方式进行对象的存储

Vector是线程同步操作安全的.

Vector的并发安全保证

我们去看他的源码发现很多对外的方法都是用Synchronized关键字进行修饰的.

所以通过vector进行操作性能并不高.所以慢慢被放弃.

|  |
| --- |
| public synchronized void addElement(E obj) {  modCount++;  ensureCapacityHelper(elementCount + 1);  elementData[elementCount++] = obj; } |

|  |
| --- |
| public synchronized boolean removeElement(Object obj) {  modCount++;  int i = indexOf(obj);  if (i >= 0) {  removeElementAt(i);  return true;  }  return false; } |

|  |
| --- |
| public synchronized E get(int index) {  if (index >= elementCount)  throw new ArrayIndexOutOfBoundsException(index);   return elementData(index); } |

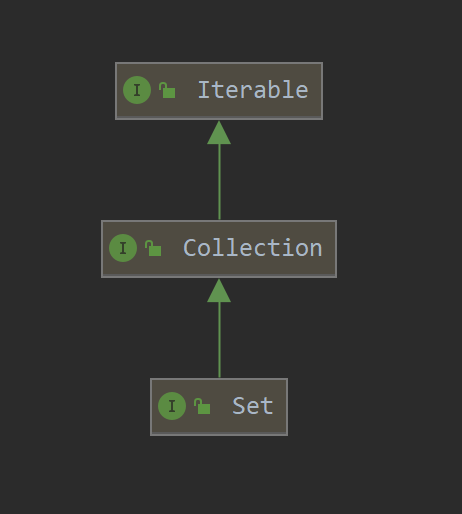
Collections工具类可以将List接口中线程不安全的工具类转换为线程安全的对象

|  |
| --- |
| public static <T> List<T> synchronizedList(List<T> list) {  return (list instanceof RandomAccess ?  new SynchronizedRandomAccessList<>(list) :  new SynchronizedList<>(list)); } |

### Set接口

一个不包含重复元素的 collection。更确切地讲，set 不包含满足 `e1.equals(e2)` 的元素对

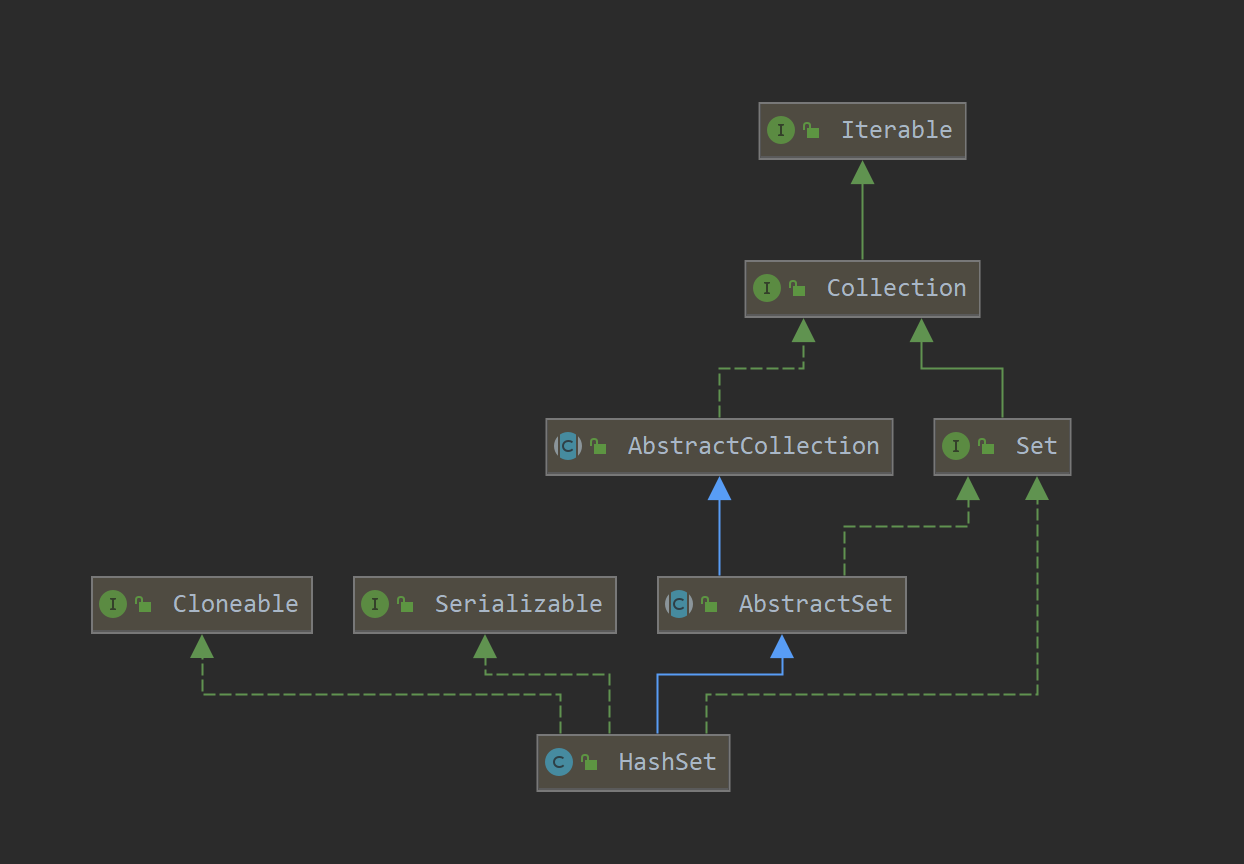
`e1` 和 `e2`，并且最多包含一个 null 元素



#### HashSet

##### 概述

此类实现 Set 接口，由哈希表（实际上是一个 HashMap 实例）支持。它不保证 set 的迭代顺序；特别是它不保证该顺序恒久不变。此类允许使用 null 元素。



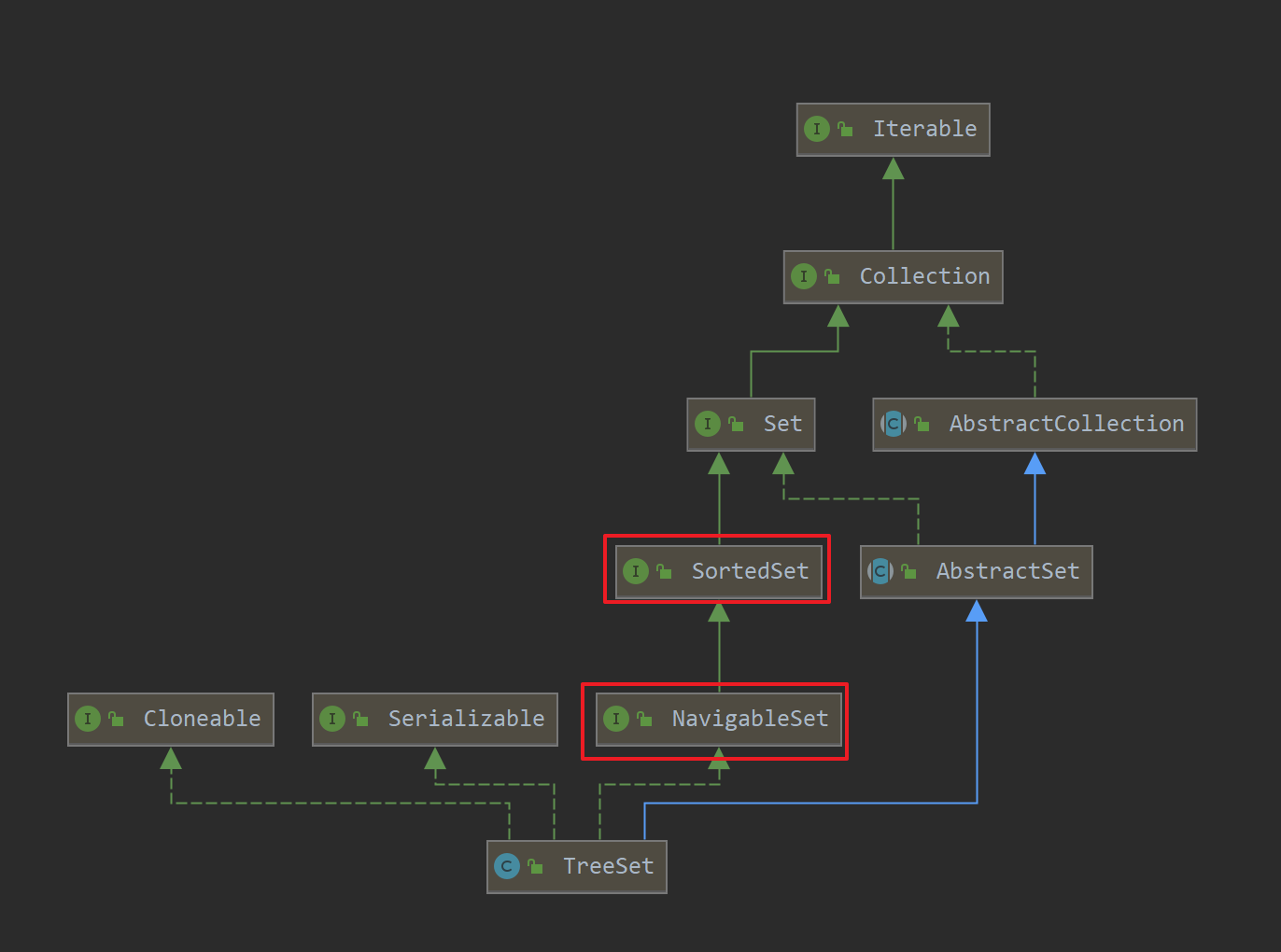
##### 特点

底层数据结构是 哈希表，HashSet的本质是一个"没有重复元素"的集合，它是通过**HashMap**实现的。HashSet中含有一个HashMap类型的成员变量map，在HashSet中操作函数，实际上都是通过map实现的。所以了解了HashMap就了解了HashSet。

#### TreeSet

##### 概述

基于TreeMap的 NavigableSet实现。使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的 Comparator进行排序，具体取决于使用的构造方法。



##### 特点

1. TreeSet 是一个有序的并且可排序的集合，它继承于AbstractSet抽象类，实现了NavigableSet<E>, Cloneable, java.io.Serializable接口。

2. TreeSet是基于TreeMap实现的。TreeSet中的元素支持2种排序方式：自然排序 或者 根据创建TreeSet 时提供的 Comparator 进行排序。这取决于使用的构造方法。同样的了解了TreeMap就了解了TreeSet。

## Map接口

Map集合的特点

1.能够存储唯一的列的数据(唯一，不可重复) Set

2.能够存储可以重复的数据(可重复) List

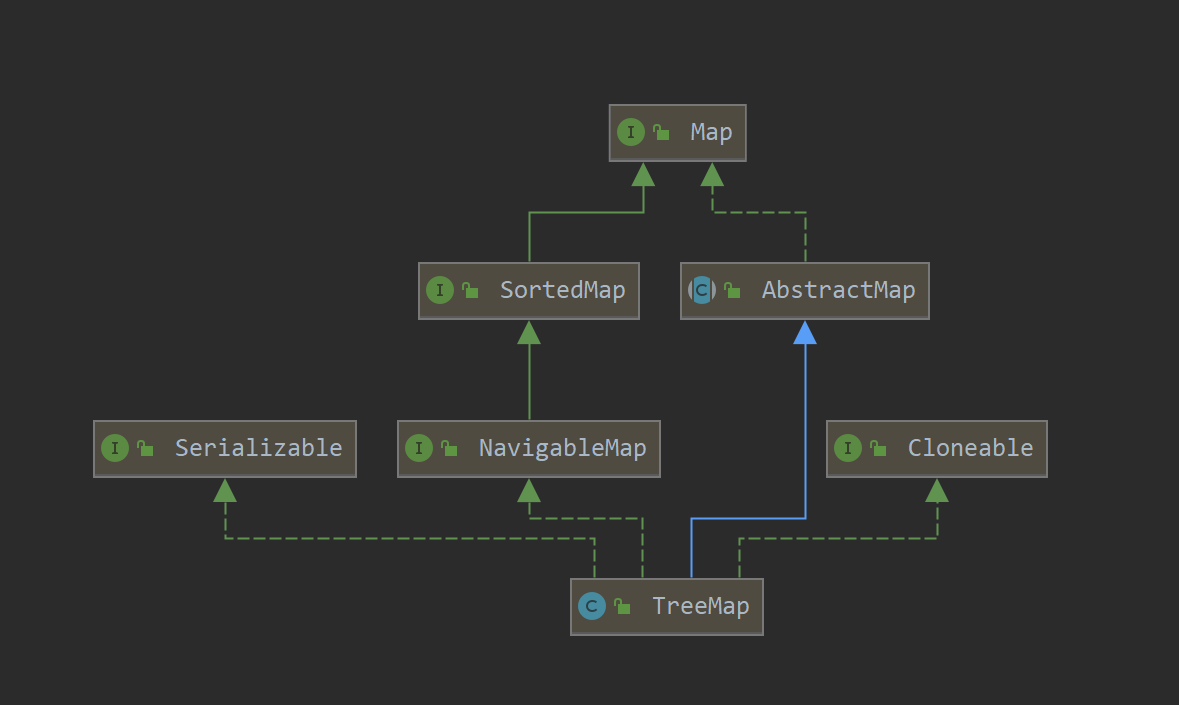
3.值的顺序取决于键的顺序

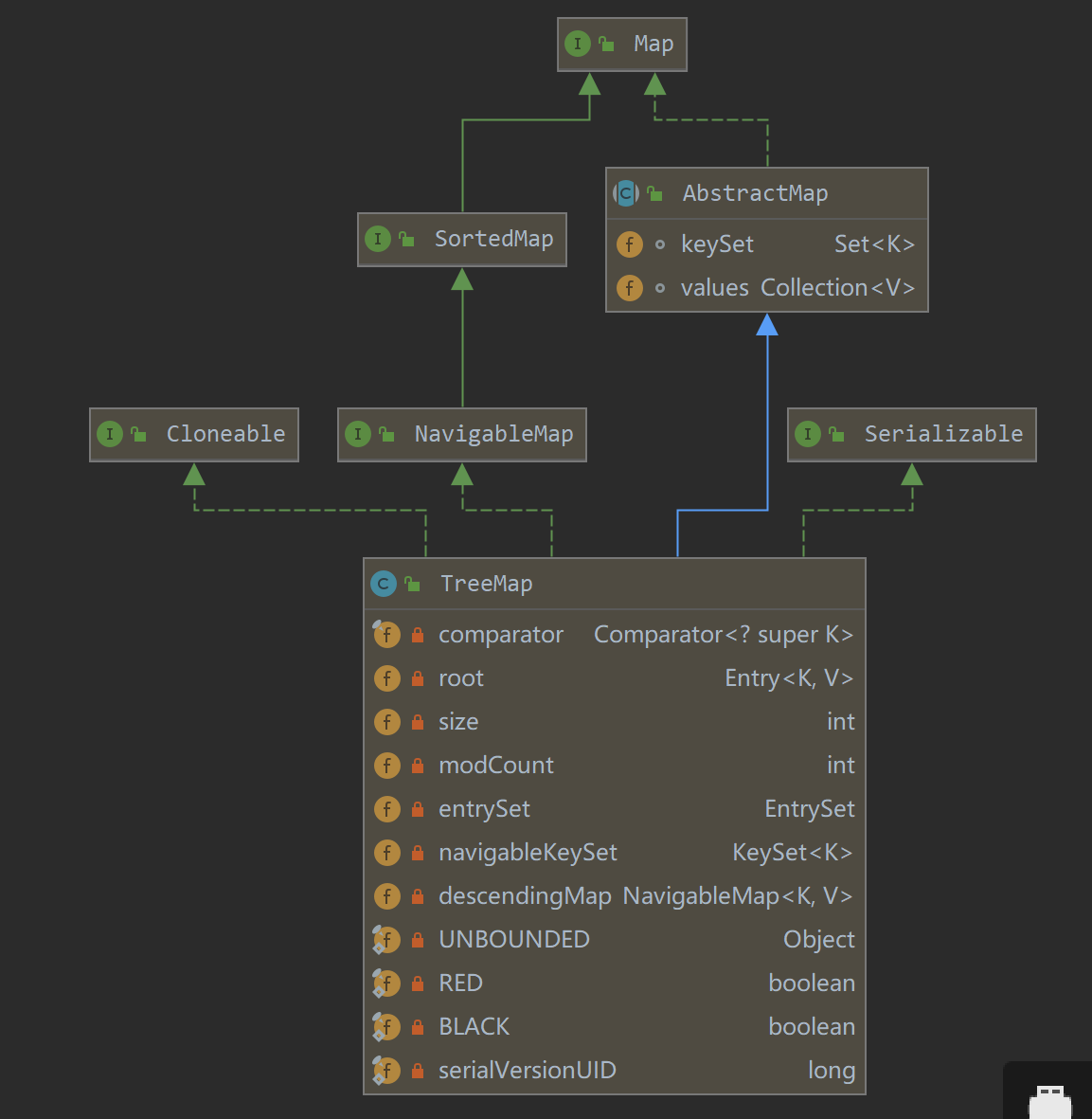
4.键和值都是可以存储null元素的

### TreeMap

TreeMap底层的实现原理是红黑树，所以我们要搞清楚TreeMap的底层原理，那么前提条件就是必须要搞清楚红黑树的原理

类图结构





|  |
| --- |
| private final Comparator<? super K> comparator; // 比较器  private transient Entry<K,V> root; // 根节点  */\*\*  \* The number of entries in the tree  \*/* private transient int size = 0; // map中元素的个数  */\*\*  \* The number of structural modifications to the tree.  \*/* private transient int modCount = 0; // 记录修改的次数 |

Entry内部类

|  |
| --- |
| static final class Entry<K,V> implements Map.Entry<K,V> {  K key; // key  V value; // vale  Entry<K,V> left; // 左子树  Entry<K,V> right; // 右子树  Entry<K,V> parent; // 父节点  boolean color = *BLACK*; // 颜色标志   */\*\*  \* Make a new cell with given key, value, and parent, and with  \* {****@code*** *null} child links, and BLACK color.  \*/* Entry(K key, V value, Entry<K,V> parent) {  this.key = key;  this.value = value;  this.parent = parent;  }   */\*\*  \* Returns the key.  \*  \** ***@return*** *the key  \*/* public K getKey() {  return key;  }   */\*\*  \* Returns the value associated with the key.  \*  \** ***@return*** *the value associated with the key  \*/* public V getValue() {  return value;  }   */\*\*  \* Replaces the value currently associated with the key with the given  \* value.  \*  \** ***@return*** *the value associated with the key before this method was  \* called  \*/* public V setValue(V value) {  V oldValue = this.value;  this.value = value;  return oldValue;  }   public boolean equals(Object o) {  if (!(o instanceof Map.Entry))  return false;  Map.Entry<?,?> e = (Map.Entry<?,?>)o;  // 重写了equals方法 必须是key和value都相等  return *valEquals*(key,e.getKey()) && *valEquals*(value,e.getValue());  }   public int hashCode() {  int keyHash = (key==null ? 0 : key.hashCode());  int valueHash = (value==null ? 0 : value.hashCode());  return keyHash ^ valueHash; // 异或  }   public String toString() {  return key + "=" + value;  } } |

以put方法为例来介绍TreeMap中红黑树的操作

· 性质1：每个节点要么是黑色，要么是红色。

· 性质2：根节点是黑色。

· 性质3：每个叶子节点（NIL）是黑色。

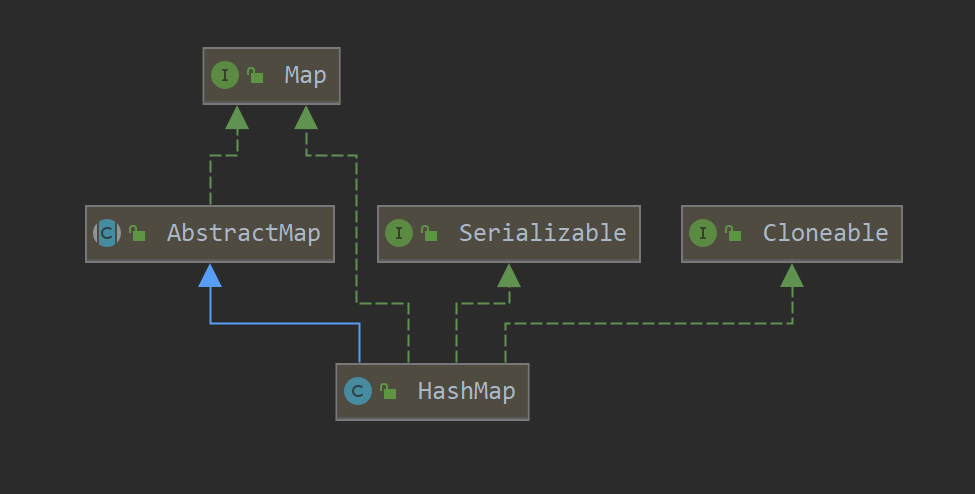
· 性质4：每个红色结点的两个子结点一定都是黑色。

· **性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。**

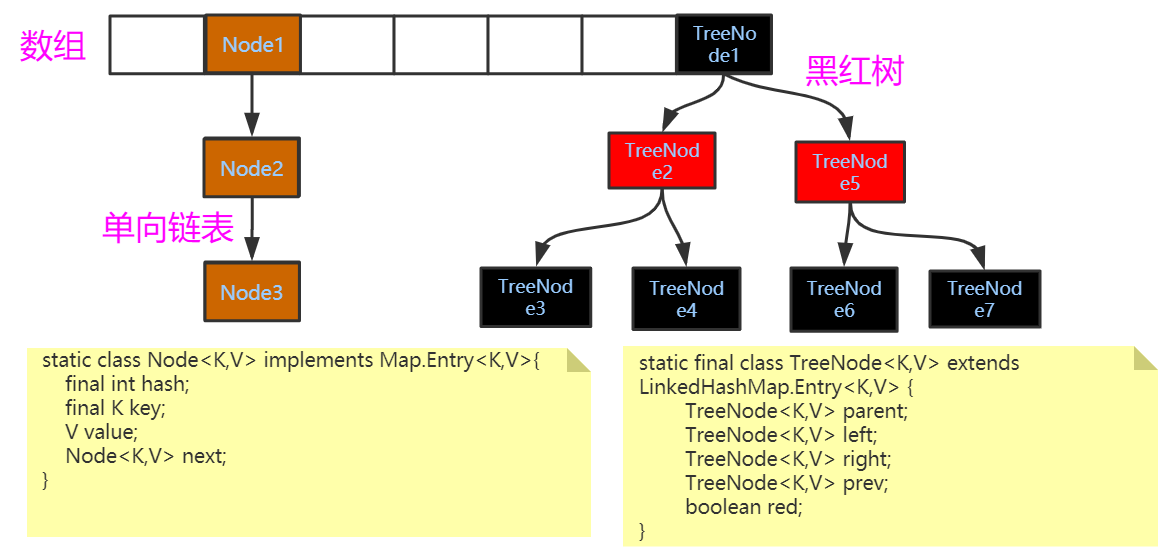
|  |
| --- |
| public V put(K key, V value) {  // 获取根节点 初始为Null  Entry<K,V> t = root;  if (t == null) {  // 检查key是否为null  compare(key, key); // type (and possibly null) check  // 对根节点初始化  root = new Entry<>(key, value, null);  size = 1;  modCount++;  return null; //返回null  }  int cmp;  Entry<K,V> parent;  // split comparator and comparable paths  Comparator<? super K> cpr = comparator;  if (cpr != null) { // 如果比较器不为null  do { // 循环  parent = t;  // 比较父节点和插入节点的值得大小  cmp = cpr.compare(key, t.key);  if (cmp < 0) // 插入节点比父节点小  t = t.left; // 把父节点左节点赋给t  else if (cmp > 0) // 插入的值比父节点大  t = t.right; // 把父节点右侧的节点赋给t  else // 如果想等，直接替换，并返回原来的值  return t.setValue(value);  } while (t != null);// 一直循环直到找到合适的插入位置  }  else {  if (key == null)  throw new NullPointerException();  // 比较器为空 就创建一个 通过ASCII码值进行比较  @SuppressWarnings("unchecked")  Comparable<? super K> k = (Comparable<? super K>) key;  do {  parent = t;  cmp = k.compareTo(t.key);  if (cmp < 0)  t = t.left;  else if (cmp > 0)  t = t.right;  else  return t.setValue(value);  } while (t != null);  }  // 将要添加的 key value 封装为一个Entry 对象  Entry<K,V> e = new Entry<>(key, value, parent);  if (cmp < 0)  parent.left = e; // 添加到父节点的左侧  else  parent.right = e; // 添加到父节点的右侧  fixAfterInsertion(e); // 红黑树的平衡  size++;  modCount++;  return null; } |

|  |
| --- |
| */\*\* From CLR \*/* private void fixAfterInsertion(Entry<K,V> x) {  // 创建的初始节点肯定是红色节点  x.color = *RED*;  // 循环 如果当前节点的父节点是红色 才有调整的必要，如果父节点是黑色不影响   while (x != null && x != root && x.parent.color == *RED*) {  // 判断父节点是否是 祖父节点的左节点  if (*parentOf*(x) == *leftOf*(*parentOf*(*parentOf*(x)))) {  // 获取父节点的兄弟节点  Entry<K,V> y = *rightOf*(*parentOf*(*parentOf*(x)));  // 父节点的兄弟节点是红色  if (*colorOf*(y) == *RED*) {  // 设置父节点为黑色  *setColor*(*parentOf*(x), *BLACK*);  *setColor*(y, *BLACK*); // 设置父节点的兄弟节点也为黑色  *setColor*(*parentOf*(*parentOf*(x)), *RED*); // 设置祖父节点为红色  // 把祖父节点赋给x 因为祖父节点是红色，当前新插入的节点 下一次循环向上再检查  x = *parentOf*(*parentOf*(x));  } else {  // 父节点的兄弟节点是黑色  // 如果插入节点是父节点的右侧节点  if (x == *rightOf*(*parentOf*(x))) {  // 插入节点指向父节点  x = *parentOf*(x);  // 以父节点为插入节点左旋  rotateLeft(x);  }  // 设置插入节点的父节点为黑色  *setColor*(*parentOf*(x), *BLACK*);  *setColor*(*parentOf*(*parentOf*(x)), *RED*); // 设置祖父节点为红色  rotateRight(*parentOf*(*parentOf*(x))); //以祖父节点为插入节点来做右旋  }  } else {// 判断父节点是否是 祖父节点的左节点 不是  // 获取父节点的兄弟节点  Entry<K,V> y = *leftOf*(*parentOf*(*parentOf*(x)));  if (*colorOf*(y) == *RED*) { // 兄弟节点为红色  // 变色即可  *setColor*(*parentOf*(x), *BLACK*);  *setColor*(y, *BLACK*);  *setColor*(*parentOf*(*parentOf*(x)), *RED*);  x = *parentOf*(*parentOf*(x));  } else {  // 右左 先右旋  if (x == *leftOf*(*parentOf*(x))) {  x = *parentOf*(x);  rotateRight(x);  }  *setColor*(*parentOf*(x), *BLACK*);  *setColor*(*parentOf*(*parentOf*(x)), *RED*);  rotateLeft(*parentOf*(*parentOf*(x)));  }  }  }  root.color = *BLACK*; // 根节点必须为黑色 } |

### HashMap



HashMap的底层数据结构



Jdk1.7及以前是采用数组+链表

Jdk1.8之后

采用数组+链表 或者 数组+红黑树方式进行元素的存储

存储在hashMap集合中的元素都将是一个Map.Entry的内部接口的实现

当数组的下标位是链表时,此时存储在该下标位置的内容将是Map.Entry的一个实现Node内部类对象

当数组的下标位是红黑树时,此时存储在该下标位置的内容将是Map.Entry的一个实现TreeNode内部类对象

比较重要的属性

|  |
| --- |
| */\*\*  \* The default initial capacity - MUST be a power of two.*  *数组的容量 初始是16 必须是2的幂次方  \*/* static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; // aka 16  */\*\*  \* The maximum capacity, used if a higher value is implicitly specified  \* by either of the constructors with arguments.  \* MUST be a power of two <= 1<<30. 最大容量  \*/* static final int *MAXIMUM\_CAPACITY* = 1 << 30;  */\*\*  \* The load factor used when none specified in constructor.*  *默认扩容的平衡因子  \*/* static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;  */\*\*  \* The bin count threshold for using a tree rather than list for a  \* bin. Bins are converted to trees when adding an element to a  \* bin with at least this many nodes. The value must be greater  \* than 2 and should be at least 8 to mesh with assumptions in  \* tree removal about conversion back to plain bins upon  \* shrinkage.*  *链表转红黑树的 临界值 当链表长度大于等于8时转换  \*/* static final int *TREEIFY\_THRESHOLD* = 8;  */\*\*  \* The bin count threshold for untreeifying a (split) bin during a  \* resize operation. Should be less than TREEIFY\_THRESHOLD, and at  \* most 6 to mesh with shrinkage detection under removal.*  *红黑树转链表的临界值 删除红黑树节点 当节点小于等于6时  \*/* static final int *UNTREEIFY\_THRESHOLD* = 6;  */\*\*  \* The smallest table capacity for which bins may be treeified.  \* (Otherwise the table is resized if too many nodes in a bin.)  \* Should be at least 4 \* TREEIFY\_THRESHOLD to avoid conflicts  \* between resizing and treeification thresholds.*  *链表转红黑树的另一个条件是 数组长度要大于64  \*/* static final int *MIN\_TREEIFY\_CAPACITY* = 64; |

|  |
| --- |
| /\* ---------------- Fields -------------- \*/  */\*\*  \* The table, initialized on first use, and resized as  \* necessary. When allocated, length is always a power of two.  \* (We also tolerate length zero in some operations to allow  \* bootstrapping mechanics that are currently not needed.)*  *数组结构  \*/* transient Node<K,V>[] table;  */\*\*  \* Holds cached entrySet(). Note that AbstractMap fields are used  \* for keySet() and values().  \*/* transient Set<Map.Entry<K,V>> entrySet;  */\*\*  \* The number of key-value mappings contained in this map.*  *Map集合中的元素个数  \*/* transient int size;  */\*\*  \* The number of times this HashMap has been structurally modified  \* Structural modifications are those that change the number of mappings in  \* the HashMap or otherwise modify its internal structure (e.g.,  \* rehash). This field is used to make iterators on Collection-views of  \* the HashMap fail-fast. (See ConcurrentModificationException).*  *修改的次数  \*/* transient int modCount;  */\*\*  \* The next size value at which to resize (capacity \* load factor).  \* 扩容的临界值  \** ***@serial*** *\*/* // (The javadoc description is true upon serialization. // Additionally, if the table array has not been allocated, this // field holds the initial array capacity, or zero signifying // DEFAULT\_INITIAL\_CAPACITY.) int threshold;  */\*\*  \** *The load factor for the hash table.*  *实际扩容因子  \*   \** ***@serial*** *\*/* final float loadFactor; |

put方法分析

|  |
| --- |
| public V put(K key, V value) {  return putVal(*hash*(key), key, value, false, true); } |

hash(key) 获取key对应的hash值

|  |
| --- |
| static final int hash(Object key) {  int h;  // key.hashCode() 长度32  return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16); } |

|  |
| --- |
| **为什么要右移16位**，大概是为了一下原因  首先，假设有一种情况，对象 A 的 hashCode 为 1000010001110001000001111000000，对象 B 的 hashCode 为 0111011100111000101000010100000。  如果数组长度是16，也就是 15 与运算这两个数， 你会发现结果都是0。这样的散列结果太让人失望了。很明显不是一个好的散列算法。  但是如果我们将 hashCode 值右移 16 位，也就是取 int 类型的一半，刚好将该二进制数对半切开。并且使用位异或运算（如果两个数对应的位置相反，则结果为1，反之为0），这样的话，就能避免我们上面的情况的发生。  总的来说，使用位移 16 位和 异或 就是防止这种极端情况。但是，该方法在一些极端情况下还是有问题，比如：10000000000000000000000000 和 1000000000100000000000000 这两个数，如果数组长度是16，那么即使右移16位，在异或，hash 值还是会重复。但是为了性能，对这种极端情况，JDK 的作者选择了性能。毕竟这是少数情况，为了这种情况去增加 hash 时间，性价比不高 |

第一次插入

|  |
| --- |
| final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  boolean evict) {  // 声明变量  Node<K,V>[] tab; Node<K,V> p; int n, i;  if ((tab = table) == null || (n = tab.length) == 0)  // 初始的情况下 进入resize方法查看  n = (tab = resize()).length;  // 确定新添加的key在数组中的位置 n = 16  if ((p = tab[i = (n - 1) & hash]) == null)  tab[i] = newNode(hash, key, value, null);  else {  Node<K,V> e; K k;  if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k))))  e = p;  else if (p instanceof TreeNode)  e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  else {  for (int binCount = 0; ; ++binCount) {  if ((e = p.next) == null) {  p.next = newNode(hash, key, value, null);  if (binCount >= *TREEIFY\_THRESHOLD* - 1) // -1 for 1st  treeifyBin(tab, hash);  break;  }  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k))))  break;  p = e;  }  }  if (e != null) { // existing mapping for key  V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null)  e.value = value;  afterNodeAccess(e);  return oldValue;  }  }  ++modCount;  if (++size > threshold)  resize();  afterNodeInsertion(evict);  return null; } |

resize方法 第一次执行时 创建了一个 Node[16] 扩容的临界值(12)

|  |
| --- |
| final Node<K,V>[] resize() {  // 记录table  Node<K,V>[] oldTab = table;  // 记录原来 数组的长度 初始为0  int oldCap = (oldTab == null) ? 0 : oldTab.length;  // 扩容临界值 默认值为0  int oldThr = threshold;  // 新的数组容量 和扩容临界值  int newCap, newThr = 0;  if (oldCap > 0) { // 初始不满足  if (oldCap >= *MAXIMUM\_CAPACITY*) {  threshold = Integer.*MAX\_VALUE*;  return oldTab;  }  else if ((newCap = oldCap << 1) < *MAXIMUM\_CAPACITY* &&  oldCap >= *DEFAULT\_INITIAL\_CAPACITY*)  newThr = oldThr << 1; // double threshold  }  else if (oldThr > 0) // initial capacity was placed in threshold  newCap = oldThr;  else { // zero initial threshold signifies using defaults  // 初始执行此处   newCap = *DEFAULT\_INITIAL\_CAPACITY*; // 16  // 16 \* 0.75 = 12 扩容的临界值 12  newThr = (int)(*DEFAULT\_LOAD\_FACTOR* \* *DEFAULT\_INITIAL\_CAPACITY*);  }  if (newThr == 0) {  float ft = (float)newCap \* loadFactor;  newThr = (newCap < *MAXIMUM\_CAPACITY* && ft < (float)*MAXIMUM\_CAPACITY* ?  (int)ft : Integer.*MAX\_VALUE*);  }  threshold = newThr;  // 实例化了一个容量为16的数组  @SuppressWarnings({"rawtypes","unchecked"})  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  table = newTab;  if (oldTab != null) { // 第一次不执行  for (int j = 0; j < oldCap; ++j) {  Node<K,V> e;  if ((e = oldTab[j]) != null) {  oldTab[j] = null;  if (e.next == null)  newTab[e.hash & (newCap - 1)] = e;  else if (e instanceof TreeNode)  ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  else { // preserve order  Node<K,V> loHead = null, loTail = null;  Node<K,V> hiHead = null, hiTail = null;  Node<K,V> next;  do {  next = e.next;  if ((e.hash & oldCap) == 0) {  if (loTail == null)  loHead = e;  else  loTail.next = e;  loTail = e;  }  else {  if (hiTail == null)  hiHead = e;  else  hiTail.next = e;  hiTail = e;  }  } while ((e = next) != null);  if (loTail != null) {  loTail.next = null;  newTab[j] = loHead;  }  if (hiTail != null) {  hiTail.next = null;  newTab[j + oldCap] = hiHead;  }  }  }  }  }  return newTab; } |

确定put进来的key在数组中的位置

|  |
| --- |
| if ((p = tab[i = (n - 1) & hash]) == null) |

|  |
| --- |
| 10101001110101001100101101  00000000000000000000011111  &  00000000000000000000001101 这是个0~15之间的数 |

HashMap的容量为什么是2的幂次方

|  |
| --- |
| hash 算法的目的是为了让hash值均匀的分布在桶中（数组），那么，如何做到呢？试想一下，如果不使用 2 的幂次方作为数组的长度会怎么样？  假设我们的数组长度是10，还是上面的公式：  1010 & 101010100101001001000 结果：1000 = 8  1010 & 101000101101001001001 结果：1000 = 8  1010 & 101010101101101001010 结果：1010 = 10  1010 & 101100100111001101100 结果：1000 = 8  这种散列结果，会导致这些不同的key值全部进入到相同的插槽中，形成链表，性能急剧下降。也就无法达成均匀的分布的目的  所以一定要保证 & 中的二进制位全为 1，才能最大限度的利用 hash 值，并更好的散列，只有全是1 ，才能有更多的散列结果。如果是 1010，有的散列结果是永远都不会出现的，比如 0111，0101，1111，1110…….，只要 & 之前的数有 0， 对应的 1 肯定就不会出现（因为只有都是1才会为1）。大大限制了散列的范围 |

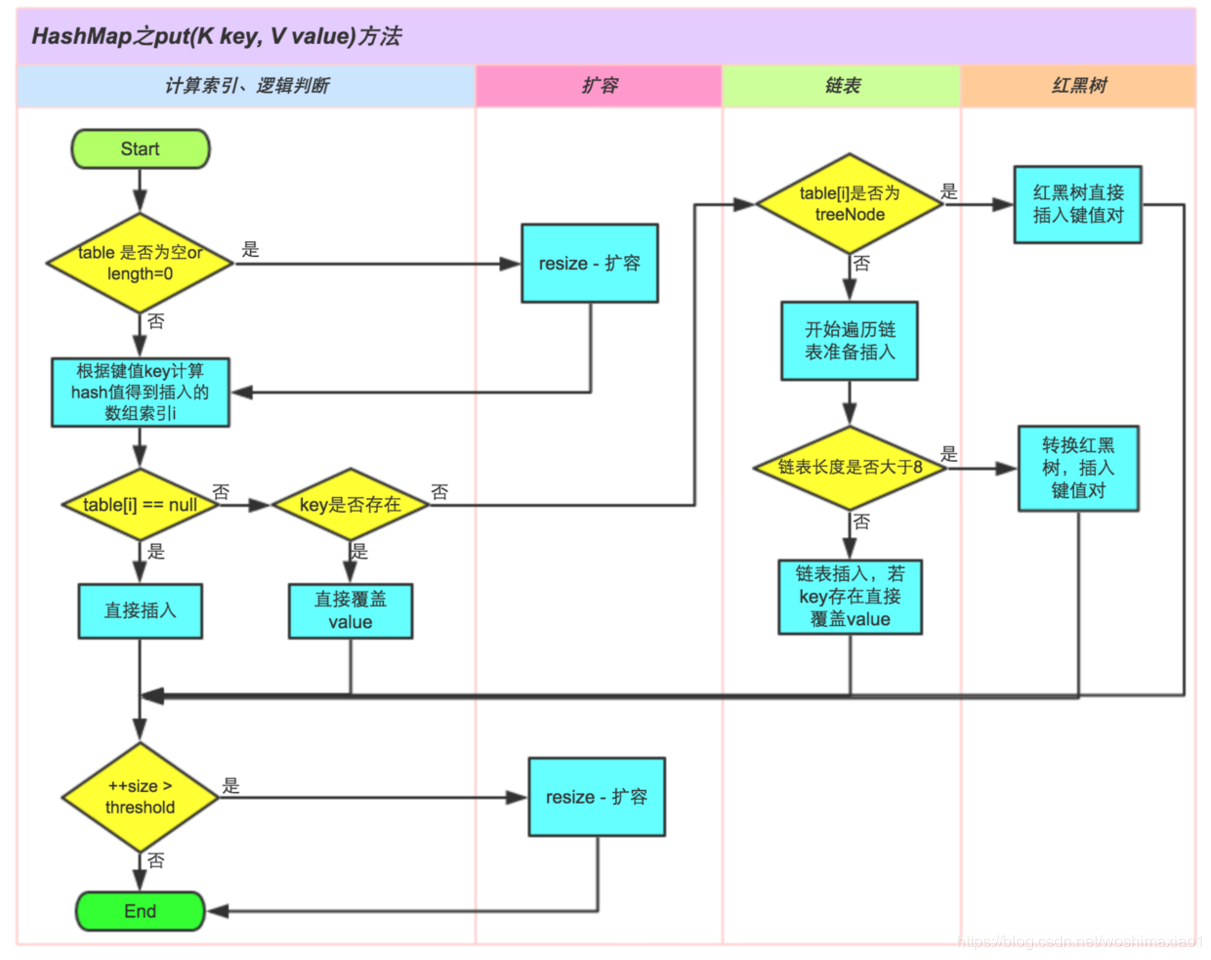
|  |
| --- |
| final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  boolean evict) {  // 声明变量  Node<K,V>[] tab; Node<K,V> p; int n, i;  if ((tab = table) == null || (n = tab.length) == 0)  // 初始的情况下 进入resize方法查看  n = (tab = resize()).length;  // 确定新添加的key在数组中的位置 n = 16  if ((p = tab[i = (n - 1) & hash]) == null)  // 如果数组的这个位置是null的就直接插入进去  tab[i] = newNode(hash, key, value, null);  else {  // 如果这个数组的位置不为null  Node<K,V> e; K k;  if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k))))  // 插入的信息和这个位置的数据是同一个key 那么久替换  e = p;  // 这个节点是 红黑树  else if (p instanceof TreeNode)  e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  else {  // 这个节点是普通链表 将数据添加到尾部  for (int binCount = 0; ; ++binCount) {  if ((e = p.next) == null) {  // 添加到链表的尾部  p.next = newNode(hash, key, value, null);  if (binCount >= *TREEIFY\_THRESHOLD* - 1) // -1 for 1st  // 链表转换为 红黑树  treeifyBin(tab, hash);  break;  }  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k))))  break;  p = e;  }  }  if (e != null) { // existing mapping for key  V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null)  e.value = value;  afterNodeAccess(e);  return oldValue;  }  }  // 修改次数加一  ++modCount;  if (++size > threshold) // 添加一条信息后数组长度如果大于扩容临界值的话就扩容  resize();  afterNodeInsertion(evict);  return null; } |

treeifyBin(tab, hash);

|  |
| --- |
| final void treeifyBin(Node<K,V>[] tab, int hash) {  int n, index; Node<K,V> e;  // 如果数组的长度没有达到64 那么就尝试扩容 并不会转换为红黑树  if (tab == null || (n = tab.length) < *MIN\_TREEIFY\_CAPACITY*)  resize();  else if ((e = tab[index = (n - 1) & hash]) != null) {  TreeNode<K,V> hd = null, tl = null;  do {  // 开始替换为红黑树  TreeNode<K,V> p = replacementTreeNode(e, null);  if (tl == null)  hd = p;  else {  p.prev = tl;  tl.next = p;  }  tl = p;  } while ((e = e.next) != null);  if ((tab[index] = hd) != null)  hd.treeify(tab);  } } |

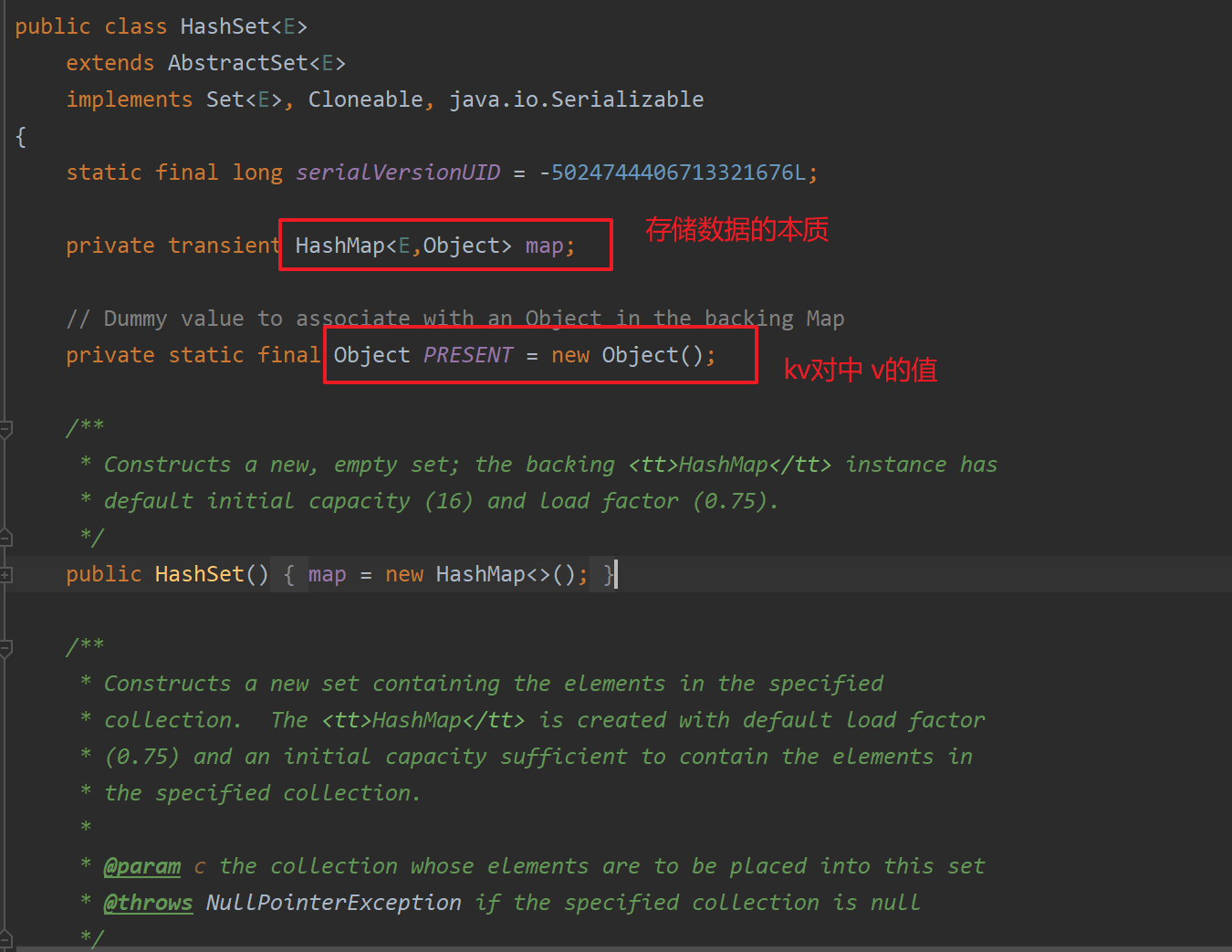
动态扩容

|  |
| --- |
| final Node<K,V>[] resize() {  // 记录table  Node<K,V>[] oldTab = table;  // 记录原来 数组的长度 16  int oldCap = (oldTab == null) ? 0 : oldTab.length;  // 扩容临界值12  int oldThr = threshold;  // 新的数组容量 和扩容临界值  int newCap, newThr = 0;  if (oldCap > 0) { // 16  if (oldCap >= *MAXIMUM\_CAPACITY*) {  threshold = Integer.*MAX\_VALUE*;  return oldTab;  }  // 原来的容量扩大一倍 扩容临界值也扩大一倍  else if ((newCap = oldCap << 1) < *MAXIMUM\_CAPACITY* &&  oldCap >= *DEFAULT\_INITIAL\_CAPACITY*)  newThr = oldThr << 1; // double threshold  }  else if (oldThr > 0) // initial capacity was placed in threshold  newCap = oldThr;  else { // zero initial threshold signifies using defaults  // 初始执行此处   newCap = *DEFAULT\_INITIAL\_CAPACITY*; // 16  // 16 \* 0.75 = 12 扩容的临界值 12  newThr = (int)(*DEFAULT\_LOAD\_FACTOR* \* *DEFAULT\_INITIAL\_CAPACITY*);  }  if (newThr == 0) {  float ft = (float)newCap \* loadFactor;  newThr = (newCap < *MAXIMUM\_CAPACITY* && ft < (float)*MAXIMUM\_CAPACITY* ?  (int)ft : Integer.*MAX\_VALUE*);  }  threshold = newThr;  // 实例化了一个容量为16的数组  @SuppressWarnings({"rawtypes","unchecked"})  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  table = newTab;  if (oldTab != null) { // 扩容执行 从原来的数组中将数据迁移到新的数组中  for (int j = 0; j < oldCap; ++j) {  // 循环数组  Node<K,V> e;  if ((e = oldTab[j]) != null) {  oldTab[j] = null;// 置空原来的节点  if (e.next == null) // 该节点没有子节点  // 直接在新数组中 找位置赋值  newTab[e.hash & (newCap - 1)] = e;  else if (e instanceof TreeNode)  // 红黑树 节点 替换  ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  else { // preserve order  // 双向链表   Node<K,V> loHead = null, loTail = null;  Node<K,V> hiHead = null, hiTail = null;  Node<K,V> next;  do {  next = e.next;  if ((e.hash & oldCap) == 0) {  if (loTail == null)  loHead = e;  else  loTail.next = e;  loTail = e;  }  else {  if (hiTail == null)  hiHead = e;  else  hiTail.next = e;  hiTail = e;  }  } while ((e = next) != null);  if (loTail != null) {  loTail.next = null;  newTab[j] = loHead;  }  if (hiTail != null) {  hiTail.next = null;  newTab[j + oldCap] = hiHead;  }  }  }  }  }  return newTab; } |

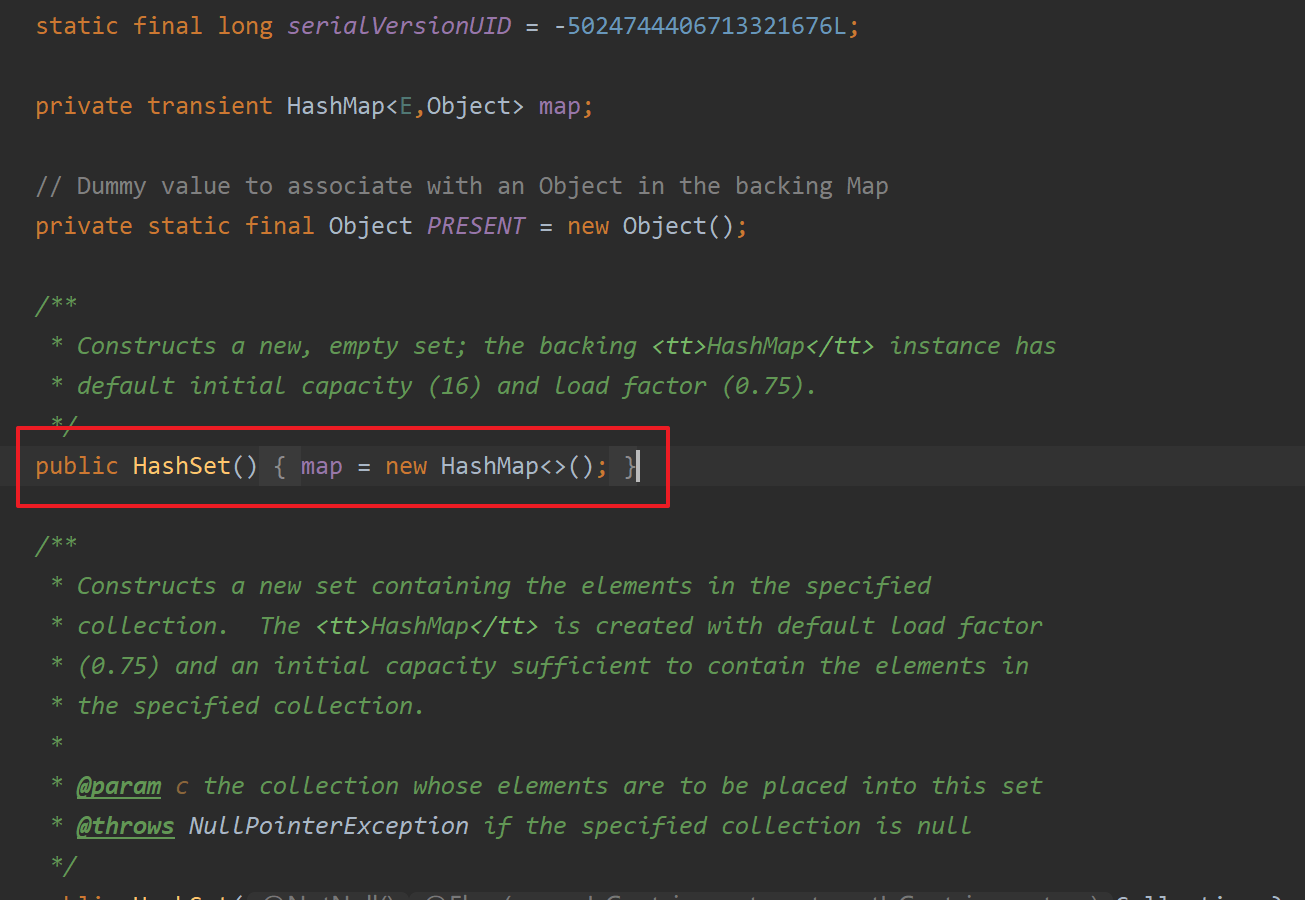


之后看HashSet和TreeSet的源码

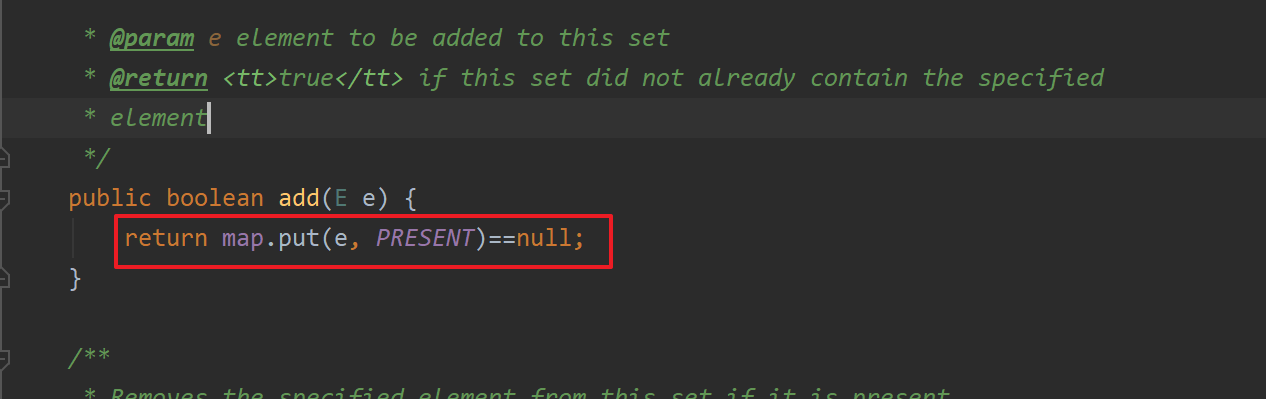
HashSet



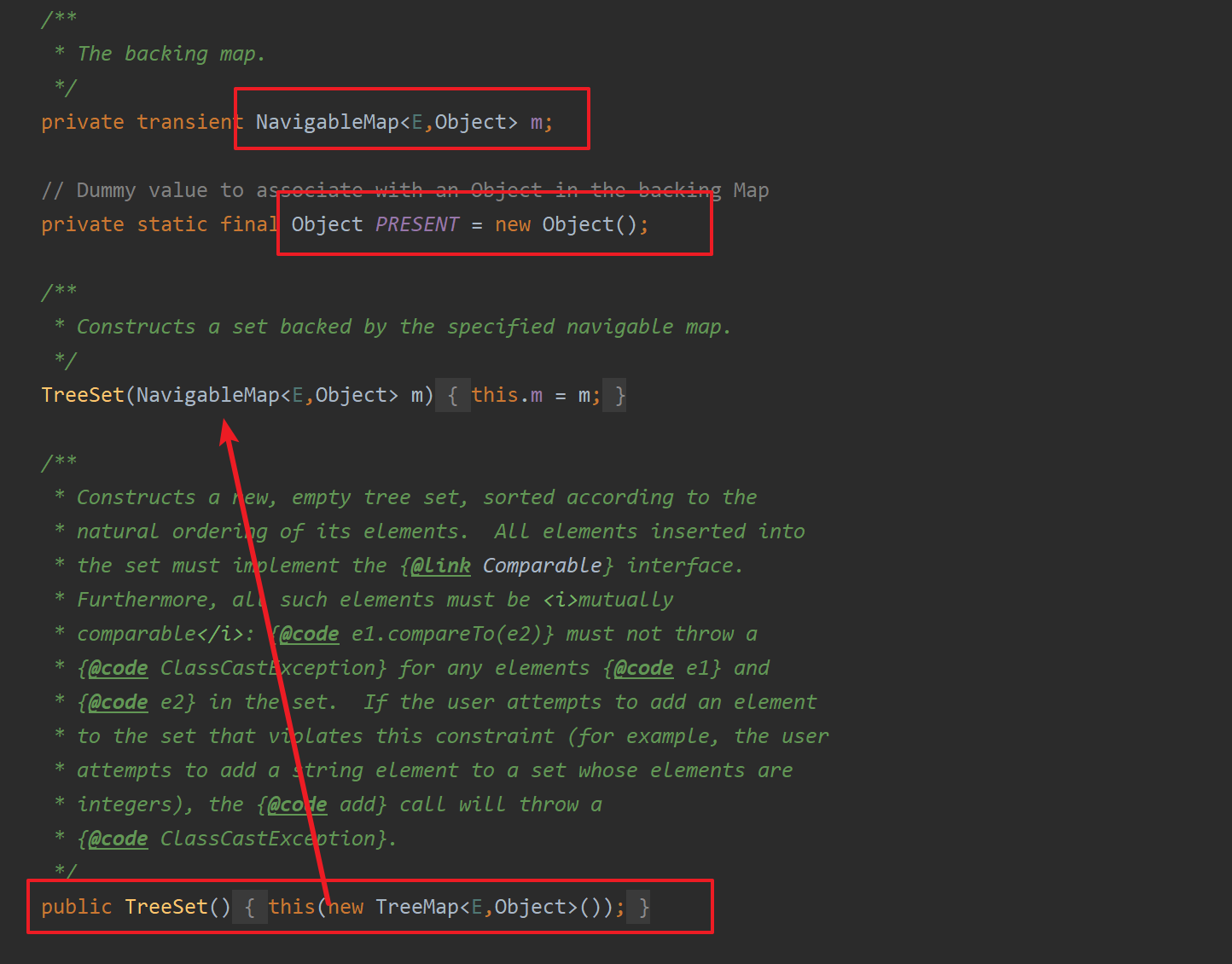
|  |
| --- |
| Set set = new HashSet();  --> 本质 new HashMap(); |



往Set中添加数据，本质上就是往Map集合中添加数据



TreeSet

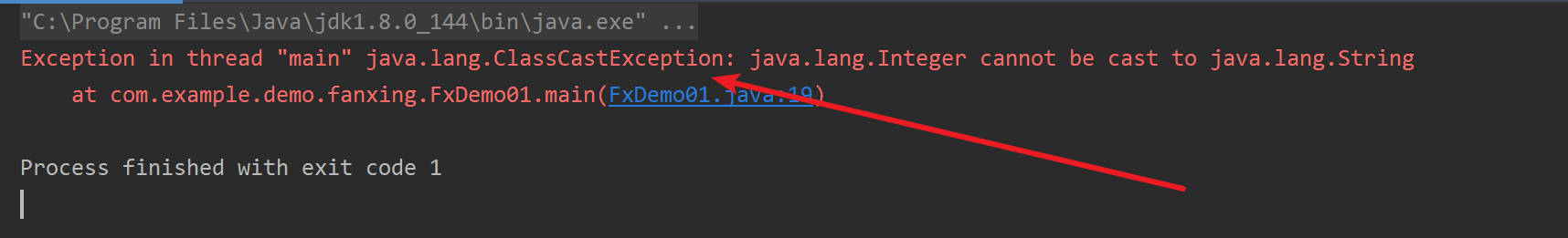


# 泛型

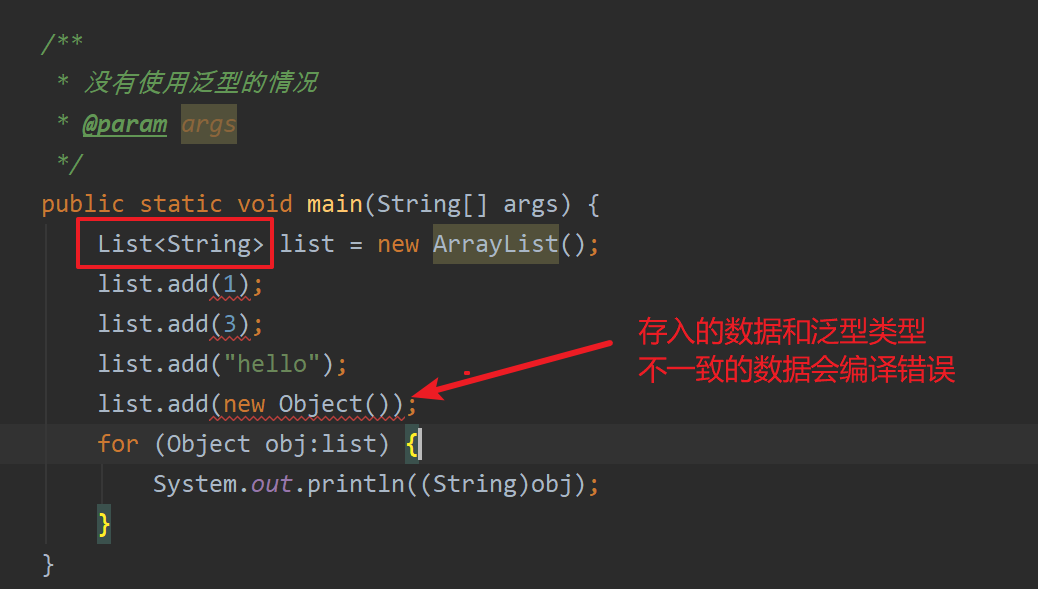
## 泛型的介绍

没有使用泛型的情况

|  |
| --- |
| */\*\*  \* 没有使用泛型的情况  \** ***@param*** *args  \*/* public static void main(String[] args) {  List list = new ArrayList();  list.add(1);  list.add(3);  list.add("hello");  list.add(new Object());  for (Object obj:list) {  System.*out*.println((String)obj);  } } |



通过上面的场景我们发现List中默认可以存放任意类型的数据，但是在具体操作的时候，如果数据类型不一致就会出现上面的类型转换异常的错误，为了避免这种情况我们可以在声明的时候来指定泛型【也就是list中能够存放的数据类型】



我们在申明ArrayList时,指定了存储类型为String 所以在编译阶段,我们存入其他类型的元素,就会报出错误.即我们的申明强制的指定了ArrayList集合中存储的元素只能是指定的String类型.这就是泛型使用的本质意义所在,指定参数化类型.

### 泛型的类型擦除

泛型只在编译阶段有效,泛型类型在逻辑上可看成是多个不同的类型，但是其实质都是同一个数据类型

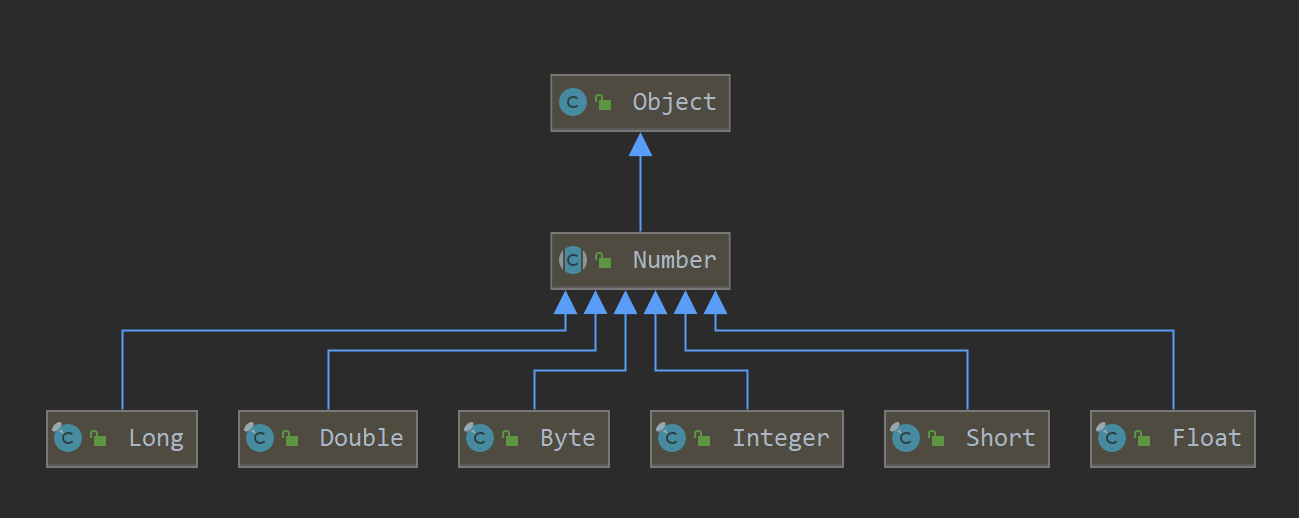
编译之后程序会采取去泛型化的措施，通过案例演示我们来给大家介绍

|  |
| --- |
| */\*\*  \* 泛型只在编译阶段有效  \* 在编译之后程序会采取去泛型化的措施  \* 正确检验泛型结果后，会将泛型相关的信息擦除  \** ***@param*** *args  \*/* public static void main(String[] args) {  List arrayList = new ArrayList();  List<String> stringList = new ArrayList<>();  System.*out*.println(arrayList.getClass());  System.*out*.println(stringList.getClass());  System.*out*.println(arrayList.getClass() == stringList.getClass()); } |

|  |
| --- |
| public static void main(String[] args) throws Exception{  List<String> stringList = new ArrayList<>();  stringList.add("zhangsan");  stringList.add("lisi");  stringList.add("gupao");  System.*out*.println("list集合的长度：" + stringList.size());  // 通过反射来添加元素  Class<? extends List> aClass = stringList.getClass();  Method method = aClass.getDeclaredMethod("add", Object.class);  method.invoke(stringList,666);  System.*out*.println("操作后的list长度:" + stringList.size()); } |

通过上面的例子可以证明，在编译之后程序会采取**去泛型化**的措施。也就是说Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。

## 泛型通配符的介绍



### 无边界通配符

? 通配符理解

前面的内容.我们知道ArrayList 是典型的泛型类.即可以限制存储的元素的元素类型.ArrayList<String>指定存入的元素时string. ArrayList<Integer>指定存储元素为Integer类型. 但是如果只有我们使用的时候才能明确知道存入类型时.我们可以使用ArrayList<?> 来表示

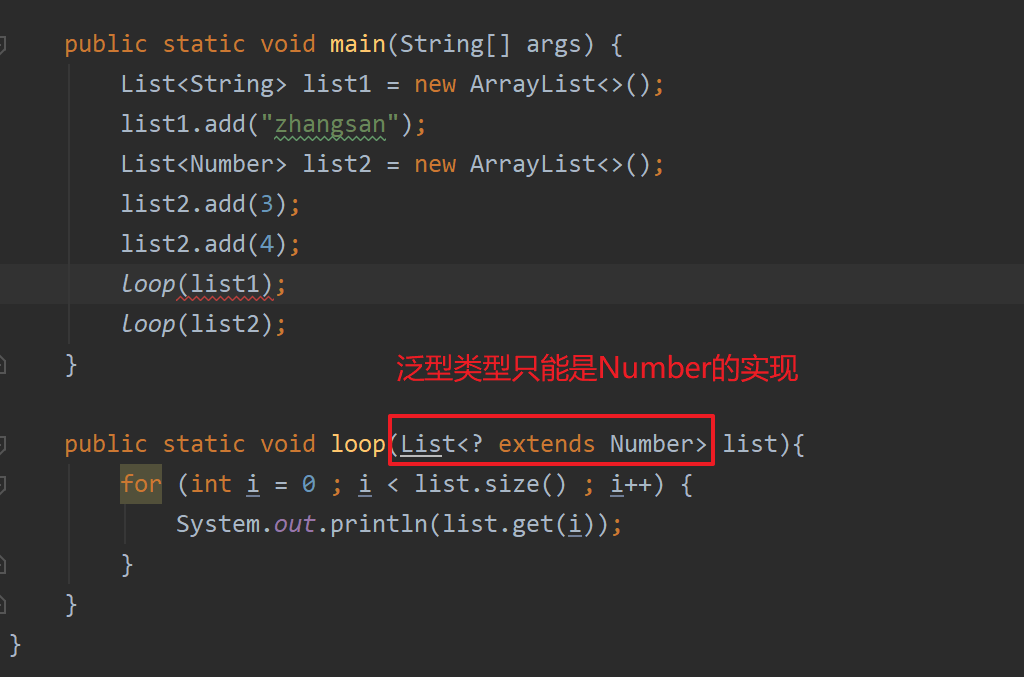
通用的类型.

|  |
| --- |
| public static void main(String[] args) {  List<String> list1 = new ArrayList<>();  list1.add("zhangsan");  List<Number> list2 = new ArrayList<>();  list2.add(3);  list2.add(4);  *loop*(list1);  *loop*(list2); }  public static void loop(List<?> list){  for (int i = 0 ; i < list.size() ; i++) {  System.*out*.println(list.get(i));  } } |

案例中的 **?** 代表着限定通用类型. 或者这么理解.代表可变的参数类型.

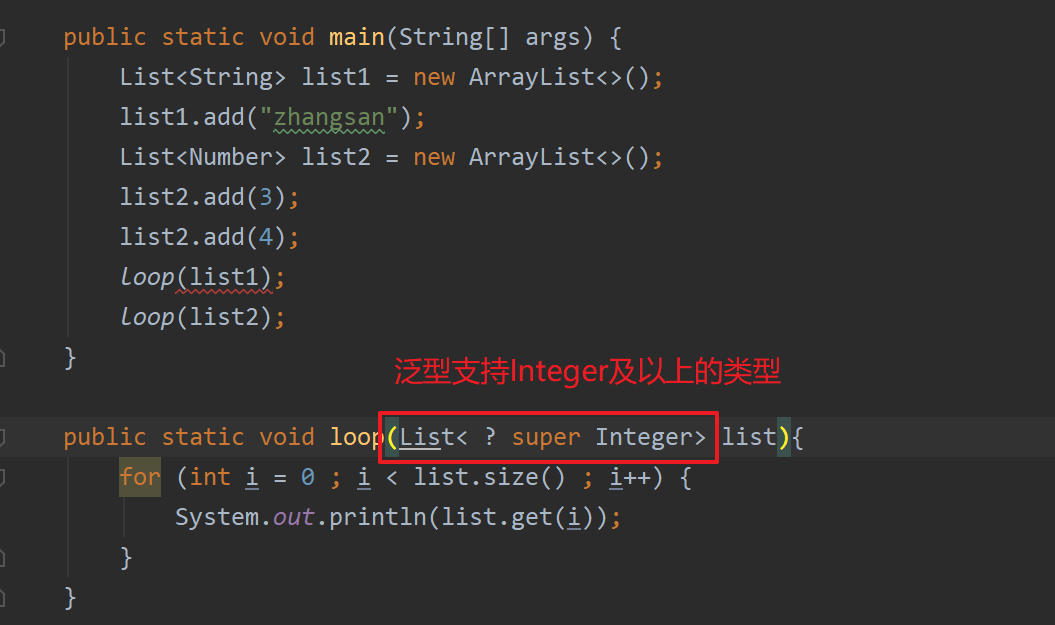
### 上界通配符

? extends Number 代表从Number往下的子类或者孙类对象都是可以的..



### 下界通配符

**? super Integer** 代表从Integer 到 Object 所有对象都是可以的.

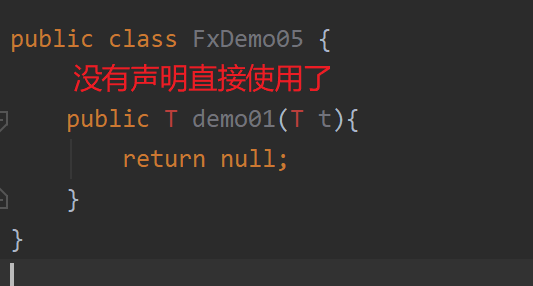


## 泛型的具体使用

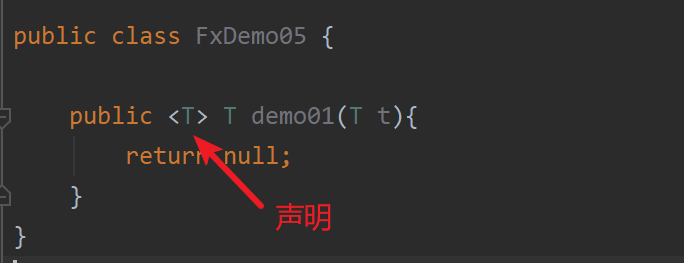
泛型跟我们的成员属性一样,需要先声明才能使用.泛型的声明采用 <> 进行声明.

申明一般约定采用单个大写字母表示.常用的有 K E T V 等等字符

错误的案例:



正确案例



### 泛型类

泛型类一般指泛型的定义与类名一起.在创建实体对象时,指定泛型的类型

普通Person类:

|  |
| --- |
| public class Person {   private String idCard;   public Person(String idCard) {  this.idCard = idCard;  } } |

实例化过程

|  |
| --- |
| public static void main(String[] args) {  Person p = new Person("431222xxxxxxxxxxxxxxxx"); } |

新加入需求,原本的身份证不仅只记录身份证,还需要记录地址 有效期 等等信息 这时我们需要创建一个实体IdentityCard进行信息的保存.此时得需求改变带来的代码代表使得我们的灵活度并不够.

我们可以使用泛型类进行解决..

|  |
| --- |
| public class IdCard {   private String name;   private Integer age;   private String address;   public String getName() {  return name;  }   public void setName(String name) {  this.name = name;  }   public Integer getAge() {  return age;  }   public void setAge(Integer age) {  this.age = age;  }   public String getAddress() {  return address;  }   public void setAddress(String address) {  this.address = address;  } } |

|  |
| --- |
| public class PersonNew <T> {   private T idCard;   public PersonNew(T idCard) {  this.idCard = idCard;  } } |

实例化过程

|  |
| --- |
| public static void main(String[] args) {  Person p = new Person("431222xxxxxxxxxxxxxxxx");   PersonNew pn1 = new PersonNew("431222xxxxxx.....");  IdCard idCard = new IdCard();  idCard.setAddress("湖南");  idCard.setAge(22);  idCard.setName("李四");  PersonNew pn2 = new PersonNew(idCard);  PersonNew pn3 = new PersonNew(18); } |

通过泛型类,我们可以提升我们程序固定逻辑的灵活度

### 泛型方法

方法的泛型有两种:

实体方法

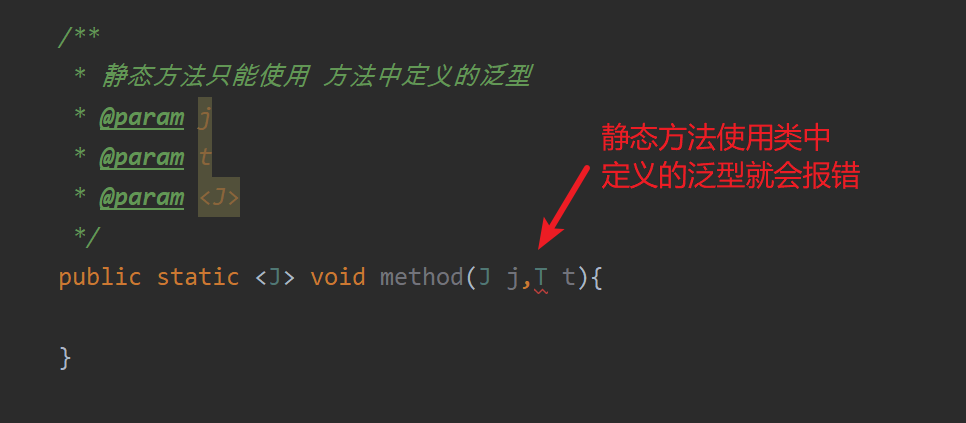
实体方法可以使用在类中定义的泛型或者方法中定义的泛型.

静态方法

不可以使用在类中定义的泛型,只能使用在静态方法上定义的泛型.

案例:

|  |
| --- |
| public class Demo07<T,K> {   */\*\*  \* 普通方法 使用类中定义的泛型  \** ***@param*** *t  \** ***@param*** *k  \** ***@return*** *\*/* public T method1(T t,K k){  return (T)null;  }   */\*\*  \* 普通方法使用 方法中定义的泛型  \** ***@param*** *y  \** ***@param*** <*S*>  *\** ***@param*** <*Y*>  *\** ***@return*** *\*/* public <S,Y> K method2(Y y){  return (K)null;  }   */\*\*  \* 静态方法只能使用 方法中定义的泛型  \** ***@param*** *j  \** ***@param*** <*J*>  *\*/* public static <J> void method(J j){   }   } |



### 泛型接口

指在接口的定义时进行泛型的申明.

接口是标准的指定者,指实现该接口的类必须实现其标准定义(即抽象方法).

所以在接口上进行泛型的申明,或者说使用泛型接口,可以让我们的程序代码更加简洁,更加多变.

案例:

计算接口的定义

|  |
| --- |
| public interface Cal {   int add(int a,int b);   int sub(int a,int b);   int mul(int a,int b);   int div(int a,int b); } |

接口的实现

|  |
| --- |
| public class Calculator implements Cal {  @Override  public int add(int a, int b) {  return 0;  }   @Override  public int sub(int a, int b) {  return 0;  }   @Override  public int mul(int a, int b) {  return 0;  }   @Override  public int div(int a, int b) {  return 0;  } } |

但是大家会发现他的程序灵活度会很差..当前结算接口的实现者,只能满足int类型数字计算..如果想满足其他数字类型的计算,需在接口中定义额外的方法.

此时,如果采用泛型接口即可完美解决问题.

计算泛型接口的定义:

|  |
| --- |
| public interface CalGeneric <T> {   T add(T t1,T t2);  T sub(T t1,T t2);  T mul(T t1,T t2);  T div(T t1,T t2); } |

计算泛型接口的实现者:

|  |
| --- |
| public class CalculatorDoubleGeneric implements CalGeneric<Double> {  @Override  public Double add(Double t1, Double t2) {  return null;  }   @Override  public Double sub(Double t1, Double t2) {  return null;  }   @Override  public Double mul(Double t1, Double t2) {  return null;  }   @Override  public Double div(Double t1, Double t2) {  return null;  } } |

如果我想实现float类型的计算需求. 我只需要实现CalGeneric<Float>即可..可以大大的提升代码的复用和改善程序的灵活度

# 反射

## Java反射定义

反射即反向探知，有点像考古学家根据发掘的物品来探知以前的事情



指在Java程序运行状态中，

1. 对于给定的一个类(Class)对象，可以获得这个类(Class)对象的所有属性和方法；
2. 对于给定的一个对象(new XXXClassName<? extends Object>)，都能够调用它的任意一个属性和方法.

这种动态获取类的内容以及动态调用对象的方法和获取属性的机制.就叫做JAVA的反射机制

如下案例

|  |
| --- |
| public static void main(String[] args) throws Exception {  // 获取类对象  Class clazz = Person.class;  // 创建对象  Person person = (Person) clazz.newInstance();  // 获取类的相关结构  System.*out*.println(clazz.getName()); // 类名  System.*out*.println(clazz.getPackage()); // 包名  System.*out*.println(clazz.getClassLoader()); // 获取类的classLoader  System.*out*.println(clazz.getSuperclass()); // 获取父类   // 获取方法  Method method = clazz.getDeclaredMethod("say");  // 等价于 Person p = new Person();  // p.say();  method.invoke(person); } |

### Java反射的优缺点

优势

增加程序的灵活性，避免将固有的逻辑程序写死到代码里

代码简洁,可读性强，可提高代码的复用率

缺点

相较直接调用在量大的情景下反射性能下降

内部暴露和安全隐患

|  |
| --- |
| public interface Office {   void toPDF(); } |

|  |
| --- |
| public class Word implements Office {  @Override  public void toPDF() {  System.*out*.println("Word 2 PDF ");  } } |

|  |
| --- |
| public class Excel implements Office {  @Override  public void toPDF() {  System.*out*.println(" Excel 2 PDF");  } } |

|  |
| --- |
| package com.example.demo.fashe1;  public class Main {  public static void main(String[] args) {  String key = "word";  }   */\*\*  \* 通过传入的Key，返回不同的对象  \** ***@param*** *key  \** ***@return*** *\*/* public static Office getInstanceByKey(String key){  if("word".equals(key)){  return new Word();  }   if("excel".equals(key)){  return new Excel();  }  return null;  }   */\*\*  \* 通过反射机制来动态创建  \** ***@param*** *key  \** ***@return*** *\*/* public static Office getInstanceReflectByKey(String key){  String packageName = "com.example.demo.fashe1";  Office office = null;  try {  Class clazz = Class.*forName*(packageName+"."+key);  office = (Office)clazz.newInstance();  }catch (Exception e){  e.printStackTrace();  }  return office;  } } |

### 反射到底慢在哪些地方

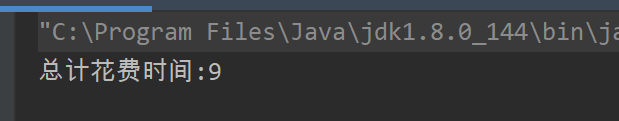
寻找类Class字节码的过程

安全管理机制的权限验证等等

若需要调用native方法调用时JNI接口的使用

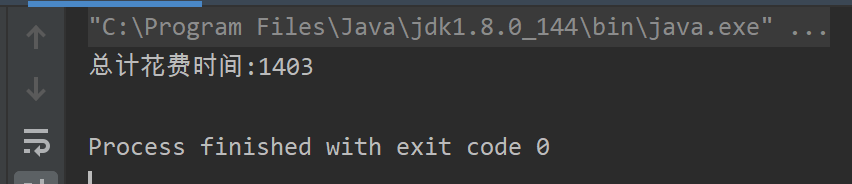
|  |
| --- |
| public static void main(String[] args) {  String key = "word";  //getInstanceByKey("word");  long startTime = System.*currentTimeMillis*();  for (int i = 0; i < 1000000 ; i++) {  *getInstanceByKey*(key);  }  long endTime = System.*currentTimeMillis*();  System.*out*.println("总计花费时间:" + (endTime - startTime)); } |

通过new的方式创建1000000个对象只需要花费9毫秒



如果通过反射的方式的话

|  |
| --- |
| public static void main(String[] args) {  String key = "Word";  //getInstanceByKey("word");  long startTime = System.*currentTimeMillis*();  for (int i = 0; i < 1000000 ; i++) {  //getInstanceByKey(key);  *getInstanceReflectByKey*(key);  }  long endTime = System.*currentTimeMillis*();  System.*out*.println("总计花费时间:" + (endTime - startTime)); } |

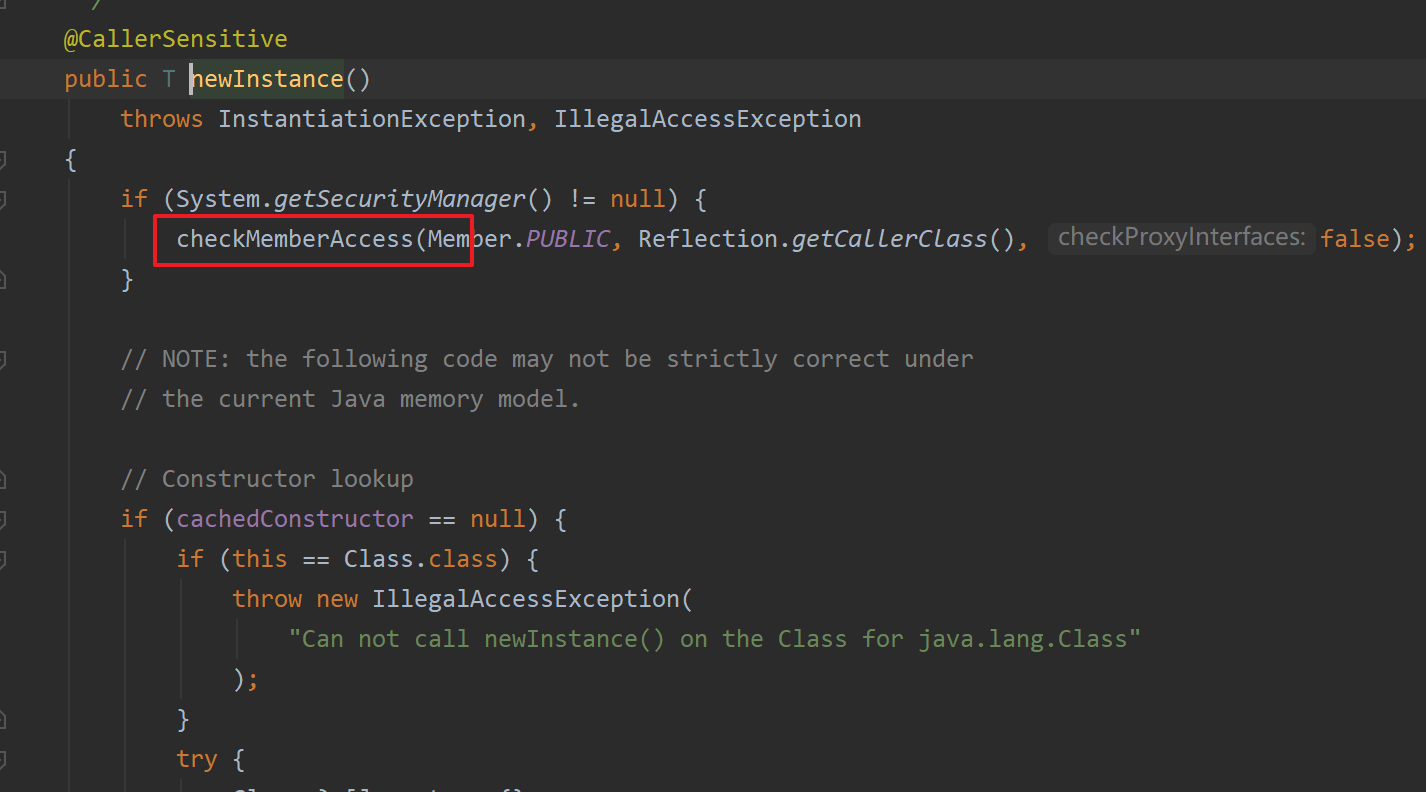


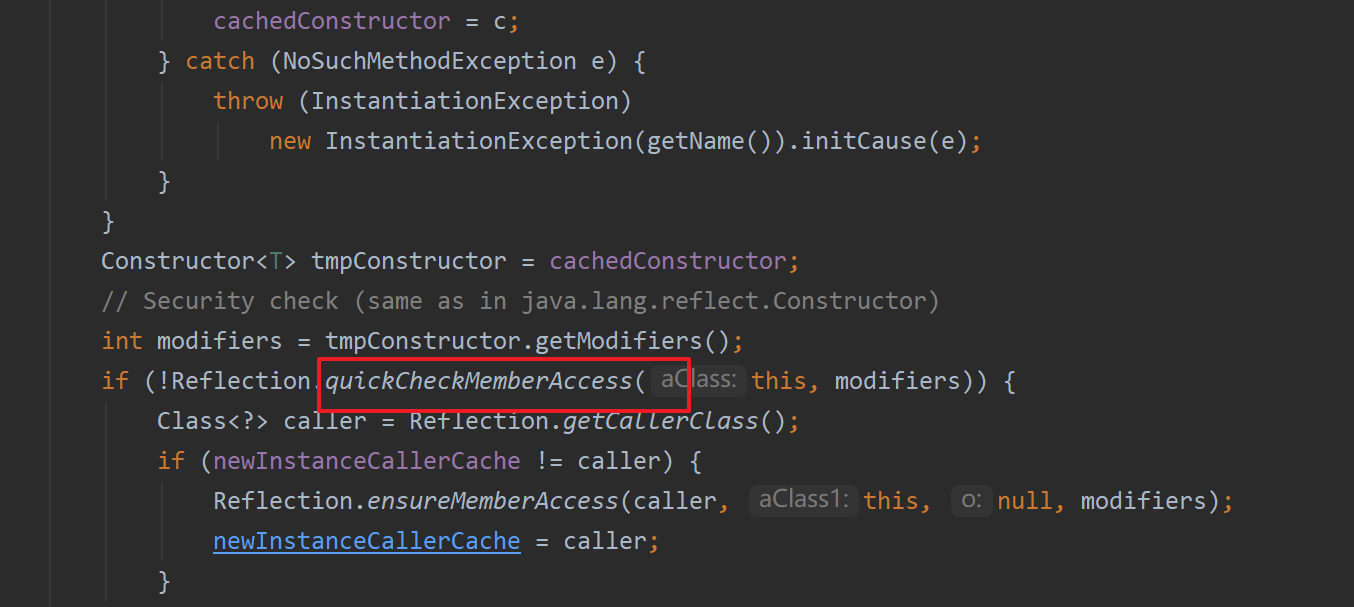
大家能发现差距还是非常大的~

源码层面

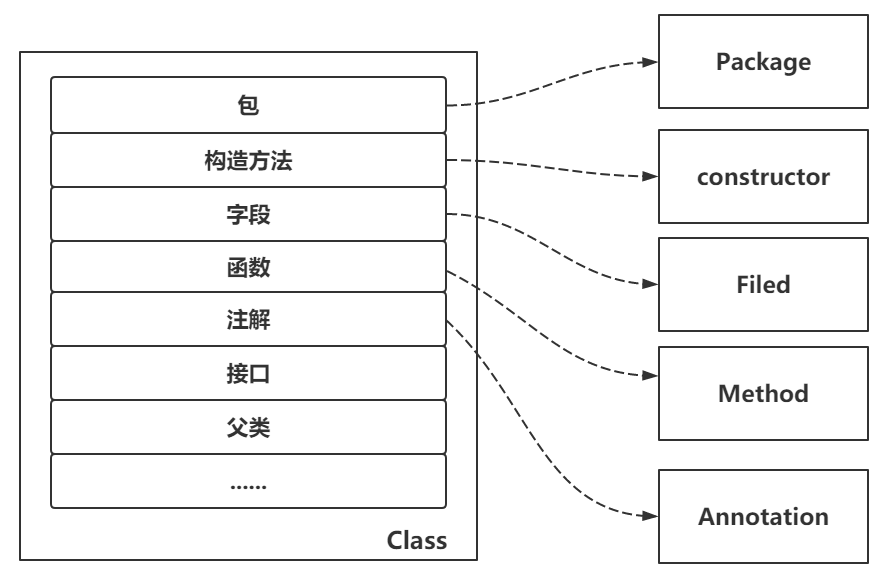
|  |
| --- |
| @CallerSensitive public static Class<?> forName(String className)  throws ClassNotFoundException {  Class<?> caller = Reflection.*getCallerClass*();  return *forName0*(className, true, ClassLoader.*getClassLoader*(caller), caller); } |

|  |
| --- |
| */\*\* Called after security check for system loader access checks have been made. \*/* private static native Class<?> forName0(String name, boolean initialize,  ClassLoader loader,  Class<?> caller)  throws ClassNotFoundException; |





## 深入Class内部



通过上面的内容,我们已经了解到我们创建的每一个自定义的Class实例都是基于他的模板类java.lang.Class类.

在大家每一个编写的类实例中,都会定义这个类的包名,类名,访问域,特征符,构造器,字段,函数,父类,接口等等内容.

这些内容在我们的Class类中都提供了对应的获取方法进行获取.

//clazz 代表对应的Class类实例

## 反射的基本操作

1. 获取类对象的四种方式

|  |
| --- |
| Class clazz = Person.class; Class clazz2 = new Person().getClass(); Class clazz3 = Class.*forName*("com.example.demo.fashe.Person"); Class clazz4 = Demo02.class.getClassLoader().loadClass("com.example.demo.fashe.Person"); |

1. 基本信息操作

|  |
| --- |
| // 获取类的相关结构 int modifier = clazz.getModifiers(); // 获取类修饰符 Package aPackage = clazz.getPackage(); // 获取类包名 String fullClassName = clazz.getName(); // 获取类的全路径名称 String simpleName = clazz.getSimpleName(); // 获取类的简单名称 ClassLoader classLoader = clazz.getClassLoader(); // 获取类加载器 Class[] interfaces = clazz.getInterfaces(); // 获取类实现的接口列表 Class superclass = clazz.getSuperclass(); // 获取类的父类 Annotation[] annotations = clazz.getAnnotations(); // 获取类的注解信息 |



## 类的属性操作

|  |
| --- |
| Person person = (Person) clazz.newInstance(); // 获取类中所有的共有字段 包含继承的字段 Field[] fields = clazz.getFields(); // 获取类中定义的字段 内部 Field[] declaredFields = clazz.getDeclaredFields(); // 获取指定名称的类中定义的字段 Field nameField = clazz.getDeclaredField("name"); // 获取字段的修饰符 int modifiers = nameField.getModifiers(); // 指定字段强制访问 nameField.setAccessible(true); // 修改字段你的值 nameField.set(person,"咕泡"); // 静态字段赋值 nameField.set(null,"静态字段赋值"); |

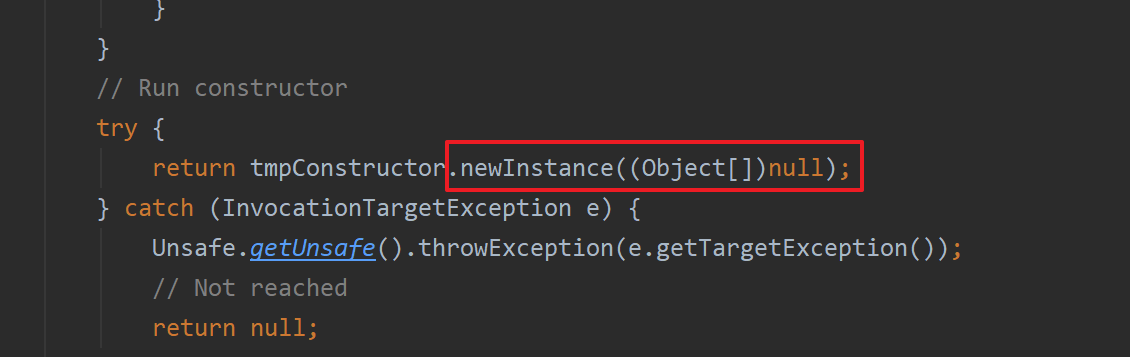
## 类的方法操作

|  |
| --- |
| // 获取类中的所有的共有的方法 继承 Method[] methods = clazz.getMethods(); // 获取类中的定义的方法 Method[] declaredMethods = clazz.getDeclaredMethods(); // 获取类中指定名称和参数的公有方法 Method say = clazz.getMethod("say", String.class); // 获取类中定义的指定名称和参数的方法 Method say1 = clazz.getDeclaredMethod("say"); // 获取方法的修饰符 int modifiers1 = say.getModifiers(); // 指定对象进行成员方法的调用 Object 咕泡666 = say.invoke(person, "咕泡666"); say.setAccessible(true);// 指定方法的强制执行 // 静态方法调用 say.invoke(null); |

## 构造器的操作

|  |
| --- |
| Constructor[] cons = clazz.getConstructors(); //获取类中所有的公有构造器 Constructor[] cons1 = clazz.getDeclaredConstructors(); //获取类中所有的构造器 Constructor conNoParam= clazz.getDeclaredConstructor(); //获取类中无参的构造器 Constructor con= clazz.getDeclaredConstructor(String.class,String.class); //获取类中有参构造 int modifers = con.getModifiers(); //获取构造器的修饰符 conNoParam.newInstance(); //构造器实例对象 con.setAccessible(true); //指定方法的强制访问 con.newInstance("abc","bbb"); //有参构造调用 Person.class.newInstance(); //class直接调用默认无参构造 |

newInstance();方法的本质



## 单例模式

定义一个简单的单例模式

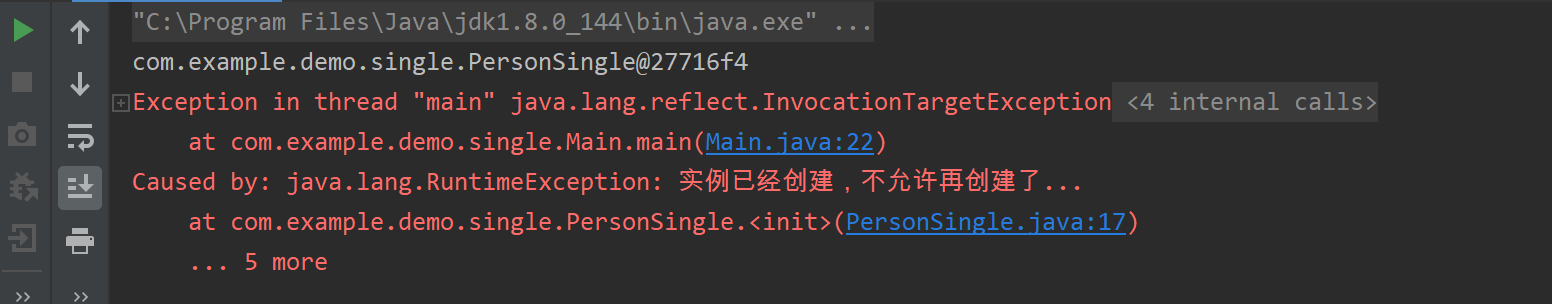
|  |
| --- |
| public class PersonSingle {   private static PersonSingle *instance*;   private PersonSingle(){   }   public static synchronized PersonSingle getInstance(){  if(*instance* == null){  *instance =* new PersonSingle();  }  return *instance*;  } } |

通过反射可以创建多个实例，从而破坏单例的设计

|  |
| --- |
| public static void main(String[] args) throws Exception{  PersonSingle s1 = PersonSingle.*getInstance*();  System.*out*.println(s1);  Class<? extends PersonSingle> aClass = s1.getClass();  Constructor<? extends PersonSingle> declaredConstructor = aClass.getDeclaredConstructor();  declaredConstructor.setAccessible(true);  Thread.*sleep*(1000);  // 通过反射 调用私有的构造器来创建对象，从而破坏单例设计  PersonSingle s2 = declaredConstructor.newInstance(null);  System.*out*.println(s2); } |

那么如何防止这种破坏呢，其实很简单我们只需要在私有构造中加个判断就可以了，如下

|  |
| --- |
| private PersonSingle(){  if(PersonSingle.*instance* != null){  throw new RuntimeException("实例已经创建，不允许再创建了...");  } } |



## 反射在Spring框架中的应用

IOC和DI

IOC(Inversion of Control)

控制反转,他是一种设计思想.并非实际的技术.最核心的思想就是将预先设计的对象实例创建的控制权交给程序(IOC容器)

IOC容器

一个管理所有控制反转过程中创建的对象的key-value容器结构(简单理解hashMap)

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">   <!-- 定义一个bean标签 -->  <bean class="com.example.demo.fashe.Person" id="person"/>  </beans> |

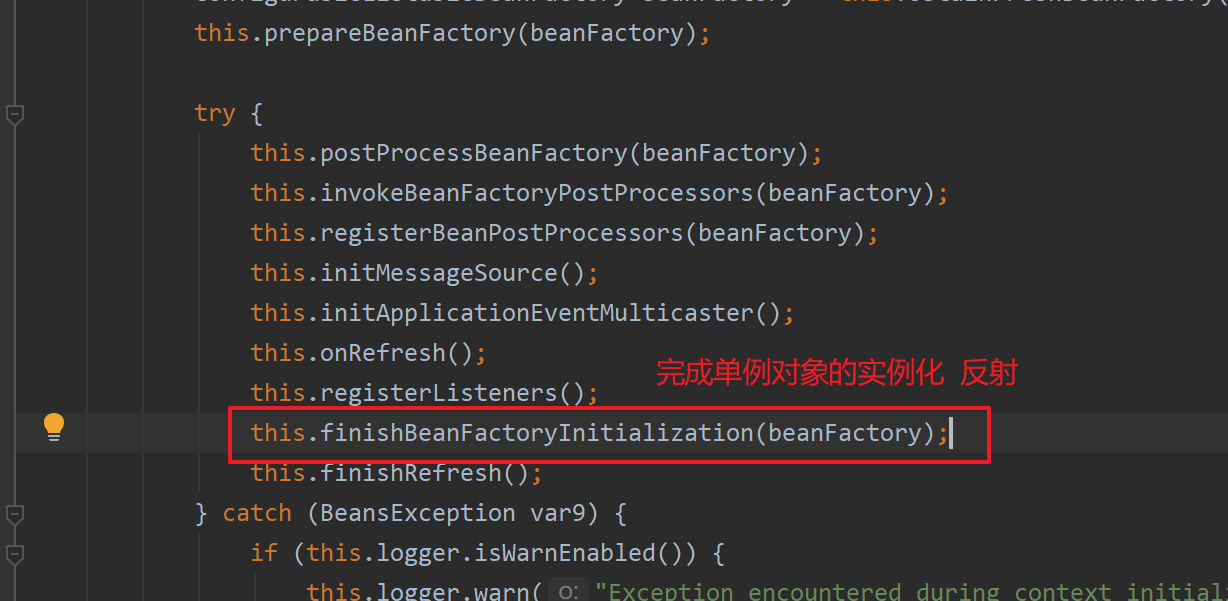
IOC是如何实例化Person对象并管理的呢？我们通过代码来看大家分析下

|  |
| --- |
| // 加载IOC容器 ApplicationContext ac = new ClassPathXmlApplicationContext("spring-ioc.xml"); |

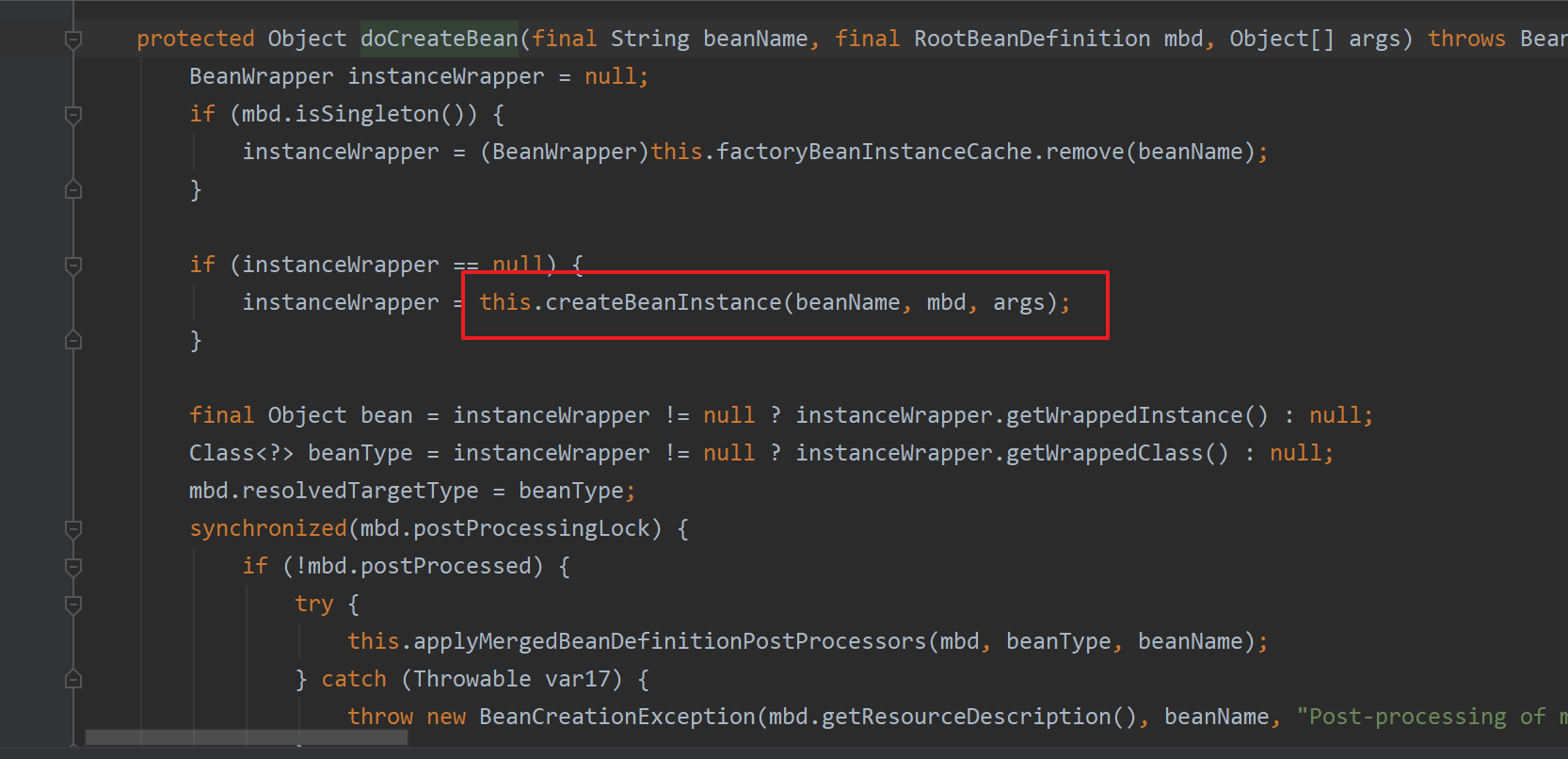
refresh方法

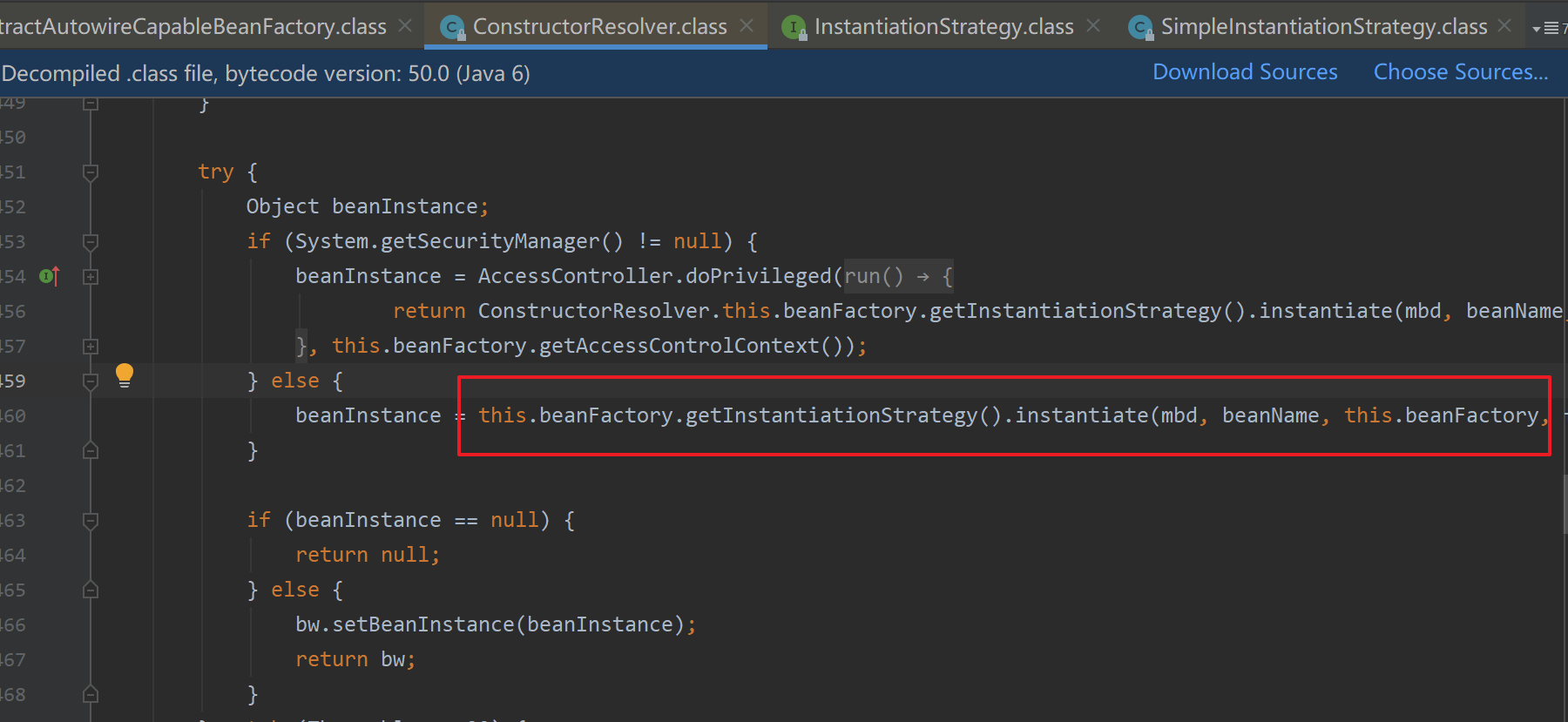
|  |
| --- |
| public void refresh() throws BeansException, IllegalStateException {  synchronized(this.startupShutdownMonitor) {  this.prepareRefresh();  // 创建一个全新的BeanFactory对象即IOC容器对象  ConfigurableListableBeanFactory beanFactory = this.obtainFreshBeanFactory();  this.prepareBeanFactory(beanFactory);   try {  this.postProcessBeanFactory(beanFactory);  this.invokeBeanFactoryPostProcessors(beanFactory);  this.registerBeanPostProcessors(beanFactory);  this.initMessageSource();  this.initApplicationEventMulticaster();  this.onRefresh();  this.registerListeners();  this.finishBeanFactoryInitialization(beanFactory);  this.finishRefresh();  } catch (BeansException var9) {  if (this.logger.isWarnEnabled()) {  this.logger.warn("Exception encountered during context initialization - cancelling refresh attempt: " + var9);  }   this.destroyBeans();  this.cancelRefresh(var9);  throw var9;  } finally {  this.resetCommonCaches();  }   } } |

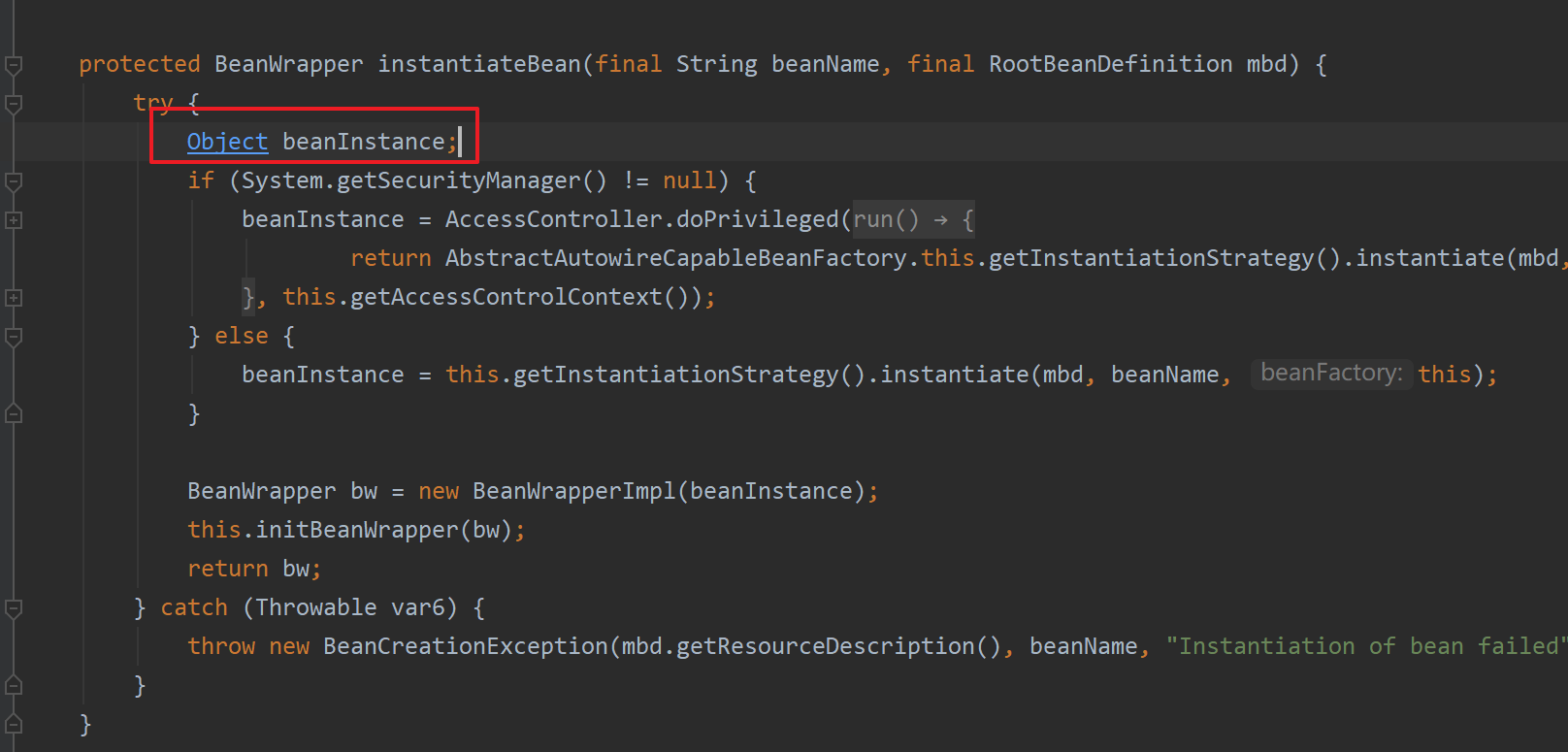
|  |
| --- |
| protected final void refreshBeanFactory() throws BeansException {  if (this.hasBeanFactory()) {  this.destroyBeans();  this.closeBeanFactory();  }   try {  // 创建IOC容器对象  DefaultListableBeanFactory beanFactory = this.createBeanFactory();  beanFactory.setSerializationId(this.getId());  this.customizeBeanFactory(beanFactory);  // 解析xml文件 把信息保存到了BeanDefinition对象中  this.loadBeanDefinitions(beanFactory);  synchronized(this.beanFactoryMonitor) {  this.beanFactory = beanFactory;  }  } catch (IOException var5) {  throw new ApplicationContextException("I/O error parsing bean definition source for " + this.getDisplayName(), var5);  } } |



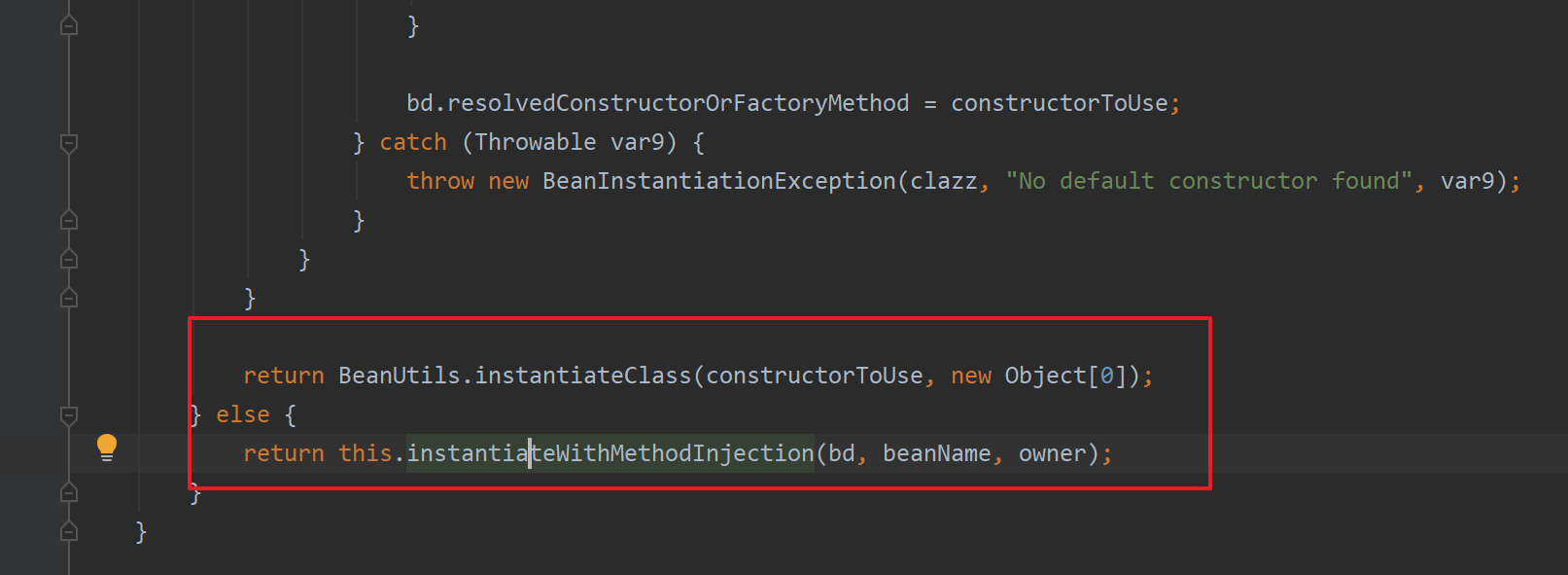
...











# 注解

## 概念

概念：说明程序的。给计算机看的

注释：用文字描述程序的。给程序员看的

定义：注解（Annotation），也叫元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明，注释。

**概念描述**：

JDK1.5之后的新特性

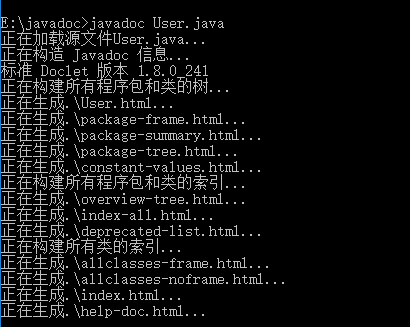
说明程序的

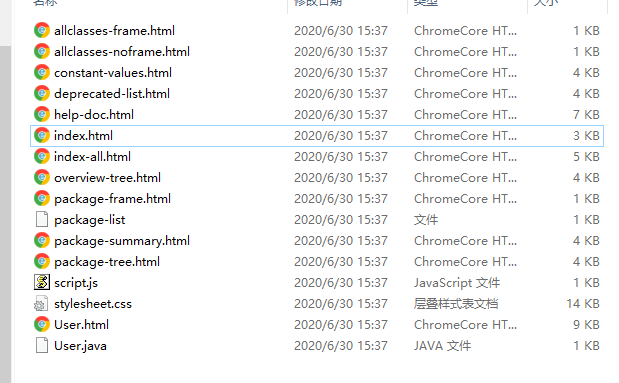
使用注解：@注解名称

**作用分类**：

1. 编写文档：通过代码里标识的注解生成文档【生成文档doc文档】

|  |
| --- |
| */\*\*  \* 注解JavaDoc描述  \*  \** ***@author*** *bobo  \** ***@version*** *1.0  \** ***@since*** *jdk1.5  \*/* **public class** User {   **private** Integer **id**;   **private** String **userName**;   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", userName='"** + **userName** + **'\''** +  **'}'**;  }   */\*\*  \* 两数相加  \** ***@param a*** *第一个整数  \** ***@param b*** *第二个整数  \** ***@return*** *返回结果  \*/* **public int** add(**int** a,**int** b){  **return** a + b;  } } |

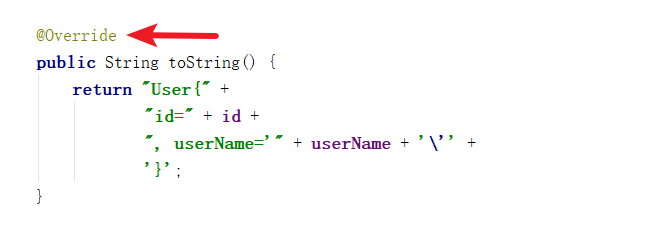






2.代码分析：通过代码里标识的注解对代码进行分析【使用反射】

3.编译检查：通过代码里标识的注解让编译器能够实现基本的编译检查【Override】



## JDK中预定义的注解

@Override：检测被该注解标注的方法是否是继承自父类(接口)的

@Deprecated：该注解标注的内容，表示已过时

@SuppressWarnings：压制警告

一般传递参数all @SuppressWarnings("all")

|  |
| --- |
| * all to suppress all warnings （抑制所有警告） * boxing to suppress warnings relative to boxing/unboxing operations（抑制装箱、拆箱操作时候的警告） * cast to suppress warnings relative to cast operations （抑制映射相关的警告） * dep-ann to suppress warnings relative to deprecated annotation（抑制启用注释的警告） * deprecation to suppress warnings relative to deprecation（抑制过期方法警告） * fallthrough to suppress warnings relative to missing breaks in switch statements（抑制确在switch中缺失breaks的警告） * finally to suppress warnings relative to finally block that don’t return （抑制finally模块没有返回的警告） * hiding to suppress warnings relative to locals that hide variable（） * incomplete-switch to suppress warnings relative to missing entries in a switch statement (enum case)(忽略没有完整的switch语句) * nls to suppress warnings relative to non-nls string literals(忽略非nls格式的字符) * null to suppress warnings relative to null analysis(忽略对null的操作) * rawtypes to suppress warnings relative to un-specific types when using generics on class params(使用generics时忽略没有指定相应的类型) * restriction to suppress warnings relative to usage of discouraged or forbidden references * serial to suppress warnings relative to missing serialVersionUID field for a serializable class(忽略在serializable类中没有声明serialVersionUID变量) * static-access to suppress warnings relative to incorrect static access（抑制不正确的静态访问方式警告) * synthetic-access to suppress warnings relative to unoptimized access from inner classes（抑制子类没有按最优方法访问内部类的警告） * unchecked to suppress warnings relative to unchecked operations（抑制没有进行类型检查操作的警告） * unqualified-field-access to suppress warnings relative to field access unqualified （抑制没有权限访问的域的警告） * unused to suppress warnings relative to unused code  （抑制没被使用过的代码的警告） |

## 自定义注解

**格式**：

元注解

public @interface 注解名称{

属性列表;

}

**本质**：注解本质上就是一个接口，该接口默认继承Annotation接口

|  |
| --- |
| public interface MyAnno extends java.lang.annotation.Annotation {} |

属性：接口中的抽象方法

要求：

1. 属性的返回值类型有下列取值

基本数据类型

String

枚举

注解

以上类型的数组

2. 定义了属性，在使用时需要给属性赋值

|  |
| --- |
| 1. 如果定义属性时，使用default关键字给属性默认初始化值，则使用注解时，可以不进行属性的赋值。  2. 如果只有一个属性需要赋值，并且属性的名称是value，则value可以省略，直接定义值即可。  3. 数组赋值时，值使用{}包裹。如果数组中只有一个值，则{}可以省略 |

元注解：用于描述注解的注解

|  |
| --- |
| @Target：描述注解能够作用的位置  ElementType取值：  TYPE：可以作用于类上  METHOD：可以作用于方法上  FIELD：可以作用于成员变量上  @Retention：描述注解被保留的阶段  @Retention(RetentionPolicy.RUNTIME)：当前被描述的注解，会保留到class字节码文件中，并被JVM读取到  @Documented：描述注解是否被抽取到api文档中  @Inherited：描述注解是否被子类继承 |

## 自定义注解案例实现

|  |
| --- |
| */\*\*  \* 自定义注解  \* 该注解表面要执行哪个类中的哪个方法  \*/* @Target(ElementType.***TYPE***) @Retention(RetentionPolicy.***RUNTIME***) **public** @**interface** InvokAnno {   String className();  String methodName(); } |

|  |
| --- |
| **public class** Student1 {  **public void** show(){  System.***out***.println(**"student1 show ...."**);  } } |

|  |
| --- |
| **package** com.gupao.edu.anno2;  **import** java.lang.reflect.Method;  @InvokAnno(className = **"com.gupao.edu.anno2.Student2"**,methodName = **"show"**) **public class** MyMain {   **public static void** main(String[] args) **throws** Exception {  *// 获取类对象* Class<MyMain> clazz = MyMain.**class**;  *// 获取类对象中的注解* InvokAnno an = clazz.getAnnotation(InvokAnno.**class**);  */\*\*  \* 注解本质是 接口 获取到的其实是接口的实现  \* public class MyInvokAnno implements InvokAnno{  \*  \* String className(){  \* return "com.gupao.edu.anno2.Student1";  \* }  \* String methodName(){  \* return "show";  \* }  \* }  \*/  // 获取注解中对应的属性* String className = an.className();  String methodName = an.methodName();  System.***out***.println(className);  System.***out***.println(methodName);   *// 通过反射的方式实现接口的功能* Class<?> aClass = Class.*forName*(className);  Method show = aClass.getDeclaredMethod(**"show"**);  *// 方法的执行* Object o = aClass.newInstance();  show.invoke(o); *// 执行对应的方法* } } |

效果

