**集合**

# 泛型

## 作用：

泛型如要是为了指定操作数据的类型，比如：

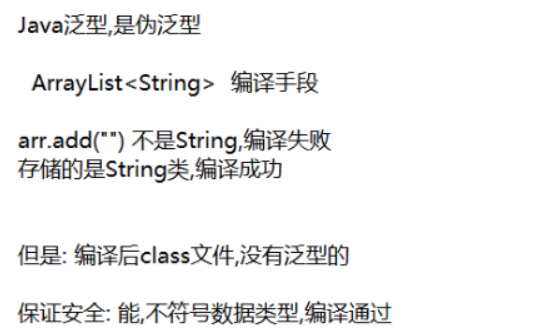
指定存入集合中存入数据的类型，解决集合中数据取出时强转错误的安全问题（ClassCastException）。当指定类型后，如果存入非指定类型数据时，会直接在编译期报错。

当然除了集合外，还能再框架设计中有更大的作用，比如泛型类、泛型接口、泛型方法等。能够提高框架的复用性。

## 伪泛型：

java中的泛型使伪泛型，泛型只在编译期出现，在运行期，会摸出类型标记。

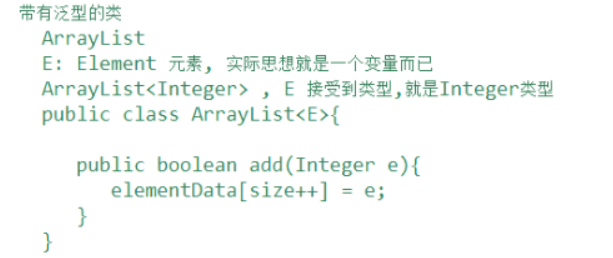
伪泛型安全性保证：存入非指定类型，无法通过编译。





## 泛型使用

### 泛型类



### 泛型方法

### 泛型接口

### 泛型的好处



### 泛型通配符

使用通配符和不适用泛型效果相同。

如下，?通配符可以匹配任务数据类型，但是就无法确定其数据类型了，所以在it.next()就不能使用强转了。



### 泛型的限定



# List

有序的单列集合。

## ArrayList

### 底层结构

底层是数组结构。

## Vector

是线程安全的数据。

跟ArrayList一样，底层是数组结构。只不过它的各操作都被**synchronized修饰了。**

## LinkedList

底层是双向链表。

|  |
| --- |
| **package** collection;  **public** **class** MyLinkedList<T> {  **private** **int** size = 0;  **private** Node<T> node;  **private** Node<T> first;  // 此处的last并不再链中，而是最后一次插入的那个值的代表  **private** Node<T> last;      **public** **void** add(T item) {  Node<T> oldLast = last;  Node<T> newNode = **new** Node<T>(oldLast, item, **null**);  // 先把最后一个操作的值设置好  last = newNode;  // 判断原始最后一个是否为空  **if**(oldLast == **null**) {  // 如果为空则，说明还没有操作过，则设置first为newNode  first = newNode;  }**else** {  // 如果oldLast不为空，说明上次已经插入过值，而且这个值排在链尾,直接将newNode 排在oldLast 后即可  oldLast.next = newNode;  }  size++;  }    **public** T get(**int** index) {  // 检查索引是否越界  **if**(index<0 || index>=size) {  System.***out***.println("下标不合法。。。");  **return** **null**;  }  // 如果下标合法  Node<T> node = first;  **for**(**int** i = 0;i<index;i++) {  node = node.next;  }  **return** node.item;  }      **private** **class** Node<T>{  **private** Node<T> previous;  **private** T item;  **private** Node<T> next;  **public** Node(Node<T> previous, T item, Node<T> next) {  **super**();  **this**.previous = previous;  **this**.item = item;  **this**.next = next;  }    }  **public** **static** **void** main(String[] args) {  MyLinkedList<Integer> list = **new** MyLinkedList<Integer>();  **for** (**int** i = 0; i < 10; i++) {  list.add(i);  }  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(list.get(i));  }  }    } |

# Set

无序单列集合，不能有重复值。

## HashSet

底层是HashMap，只不过只用到了key

## TreeSet

TreeSet的本质是一个"有序的，并且没有重复元素"的集合，它是通过[TreeMap](http://www.cnblogs.com/skywang12345/p/3310928.html)实现的。TreeSet中含有一个"NavigableMap类型的成员变量"m，而m实际上是"TreeMap的实例"

# Map

无序的双列集合。key - value

## HashMap

### 底层结构

数组+单向链表（红黑树）

|  |
| --- |
| **package** collection;  **public** **class** MyHashMap<K,V> {    **static** **class** Node<K,V>{  **private** K k;  **private** V v;  **private** Node<K,V> next;  **private** **int** hash;    **public** Node() {  }  **public** Node(K k, V v, Node<K, V> next, **int** hash) {  **super**();  **this**.k = k;  **this**.v = v;  **this**.next = next;  **this**.hash = hash;  }  }    **private** Node<K,V>[] nodes = (Node<K,V>[])**new** Node[16];  Object[] o = **new** Object[1];  **private** **int** capacity = 16;    **public** **void** put(K k,V v) {    putValue(k,v,**null**,getHash(k));  }    /\*\*  \* 存值  \* **@param** k  \* **@param** v  \* **@param** object  \* **@param** hash  \*/  **private** **void** putValue(K k, V v, Object object, **int** hash) {  // 给hash取模 0-capacity  **int** index = hash%capacity;  Node<K,V> newNode = **new** Node<K,V>(k, v, **null**, hash);  Node<K,V> node = nodes[index];  // 判断该根节点是否已经有值  **if**(node == **null**) {  // 如果为空，则直接设置尾新值  nodes[index] = newNode;  }**else** {  // 如果不为空，说明已经有值了，则将其放在链尾  // 循环遍历该链表，直到最后一个  **while**(node.next != **null**) {  node = node.next;  }  // 设置最后一个链表的下一个为新值  node.next = newNode;  }    }    **public** V get(K k) {  // 获取hash值  **int** hash = getHash(k);  // 取模  **int** index = hash%capacity;  // 获取数组中该位置的值  Node<K,V> node = nodes[index];  // 判断该node的key是否等于k  **if**(node.k == k) {  // 相等则返回其值  **return** node.v;  }  // 不等则遍历判断  **while**(node.next != **null**) {  node = node.next;  **if**(node.k == k) {  // 相等则返回其值  **return** node.v;  }  }  // 如果到此，说明没有该值  System.***out***.println("无匹配值。。。。");  **return** **null**;  }    // 获取key的hash值  **public** **int** getHash(K k) {  **return** k.hashCode();  }          **public** **static** **void** main(String[] args) {  MyHashMap<String, String> map = **new** MyHashMap<String, String>();  map.put("hello", "world");  System.***out***.println(map.get("hello"));      }  } |

## HashTable

### 底层结构

线程安全，每个操作都被**synchronized修饰。**

和HashMap类似，底层是数组+双向链表。只不过他没有链表和红黑树的转换。

## TreeMap

底层是红黑树。

