**Java 集合框架**

早在Java 2中之前，Java就提供了特设类。比如：Dictionary, Vector, Stack, 和Properties这些类用来存储和操作对象组。

虽然这些类都非常有用，但是它们缺少一个核心的，统一的主题。由于这个原因，使用Vector类的方式和使用Properties类的方式有着很大不同。

集合框架被设计成要满足以下几个目标。

* 该框架必须是高性能的。基本集合（动态数组，链表，树，哈希表）的实现也必须是高效的。
* 该框架允许不同类型的集合，以类似的方式工作，具有高度的互操作性。
* 对一个集合的扩展和适应必须是简单的。

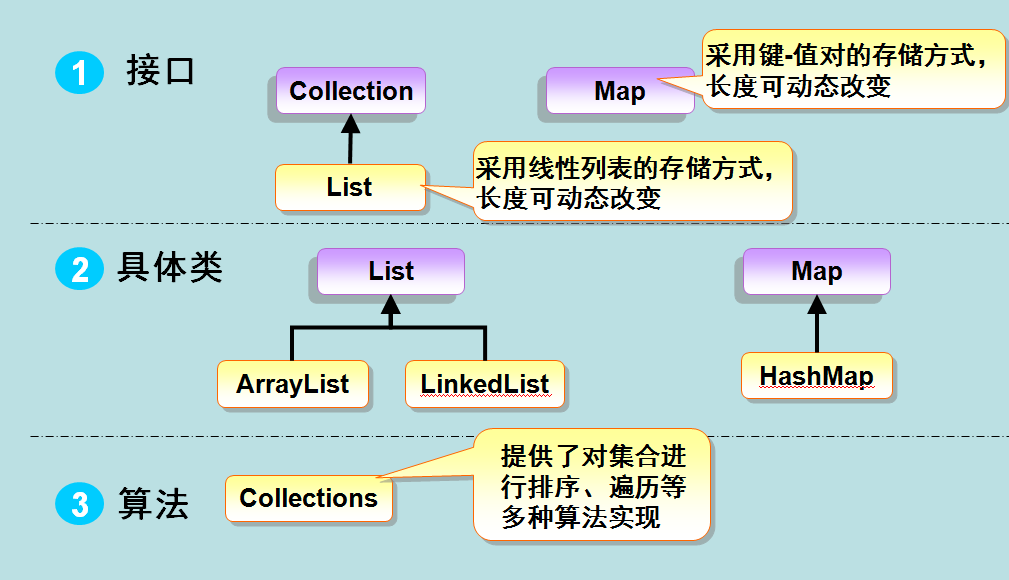
为此，整个集合框架就围绕一组标准接口而设计。你可以直接使用这些接口的标准实现，诸如： LinkedList, HashSet, 和 TreeSet等,除此之外你也可以通过这些接口实现自己的集合。

集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：

* **接口：**是代表集合的抽象数据类型。接口允许集合独立操纵其代表的细节。在面向对象的语言，接口通常形成一个层次。
* **实现（类）：**是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构。
* **算法：**是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现。

除了集合，该框架也定义了几个Map接口和类。Map里存储的是键/值对。尽管Map不是collections，但是它们完全整合在集合中。

**集合框架体系如图所示**



Java 集合框架提供了一套性能优良，使用方便的接口和类，java集合框架位于java.util包中， 所以当使用集合框架的时候需要进行导包。

**集合接口**

集合框架定义了一些接口。本节提供了每个接口的概述：

|  |  |
| --- | --- |
| **序号** | **接口描述** |
| 1 | Collection 接口  Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素, Java不提供直接继承自Collection的类，只提供继承于的子接口(如List和set)。 |
| 2 | List 接口  List接口是一个有序的 Collection，使用此接口能够精确的控制每个元素插入的位置，能够通过索引(元素在List中位置，类似于数组的下标)来访问List中的元素，第一个元素的索引为 0，而且允许有相同的元素。 |
| 3 | Set  Set 具有与 Collection 完全一样的接口，只是行为上不同，Set 不保存重复的元素。 |
| 4 | SortedSet 继承于Set保存有序的集合。 |
| 5 | Map 将唯一的键映射到值。 |
| 6 | Map.Entry 描述在一个Map中的一个元素（键/值对）。是一个Map的内部类。 |
| 7 | SortedMap 继承于Map，使Key保持在升序排列。 |
| 8 | Enumeration 这是一个传统的接口和定义的方法，通过它可以枚举（一次获得一个）对象集合中的元素。这个传统接口已被迭代器取代。 |

**Set和List的区别**

* 1. Set 接口实例存储的是无序的，不重复的数据。List 接口实例存储的是有序的，可以重复的元素。
* 2. Set检索效率低下，删除和插入效率高，插入和删除不会引起元素位置改变 **<实现类有HashSet,TreeSet>**。
* 3. List和数组类似，可以动态增长，根据实际存储的数据的长度自动增长List的长度。查找元素效率高，插入删除效率低，因为会引起其他元素位置改变 **<实现类有ArrayList,LinkedList,Vector>** 。

**集合实现类（集合类）**

Java提供了一套实现了Collection接口的标准集合类。其中一些是具体类，这些类可以直接拿来使用，而另外一些是抽象类，提供了接口的部分实现。

标准集合类汇总于下表：

|  |  |
| --- | --- |
| **序号** | **类描述** |
| 1 | **AbstractCollection** 实现了大部分的集合接口。 |
| 2 | **AbstractList** 继承于AbstractCollection 并且实现了大部分List接口。 |
| 3 | **AbstractSequentialList** 继承于 AbstractList ，提供了对数据元素的链式访问而不是随机访问。 |
| 4 | LinkedList  该类实现了List接口，允许有null（空）元素。主要用于创建链表数据结构，该类没有同步方法，如果多个线程同时访问一个List，则必须自己实现访问同步，解决方法就是在创建List时候构造一个同步的List。例如：  Listlist=Collections.synchronizedList(newLinkedList(...));  LinkedList 查找效率低。 |
| 5 | ArrayList  该类也是实现了List的接口，实现了可变大小的数组，随机访问和遍历元素时，提供更好的性能。该类也是非同步的,在多线程的情况下不要使用。ArrayList 增长当前长度的50%，插入删除效率低。 |
| 6 | **AbstractSet** 继承于AbstractCollection 并且实现了大部分Set接口。 |
| 7 | HashSet  该类实现了Set接口，不允许出现重复元素，不保证集合中元素的顺序，允许包含值为null的元素，但最多只能一个。 |
| 8 | LinkedHashSet 具有可预知迭代顺序的 Set 接口的哈希表和链接列表实现。 |
| 9 | TreeSet  该类实现了Set接口，可以实现排序等功能。 |
| 10 | **AbstractMap** 实现了大部分的Map接口。 |
| 11 | HashMap  HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。 该类实现了Map接口，根据键的HashCode值存储数据，具有很快的访问速度，最多允许一条记录的键为null，不支持线程同步。 |
| 12 | TreeMap  继承了AbstractMap，并且使用一颗树。 |
| 13 | WeakHashMap  继承AbstractMap类，使用弱密钥的哈希表。 |
| 14 | LinkedHashMap  继承于HashMap，使用元素的自然顺序对元素进行排序. |
| 15 | IdentityHashMap  继承AbstractMap类，比较文档时使用引用相等。 |

在前面的教程中已经讨论通过java.util包中定义的类，如下所示：

|  |  |
| --- | --- |
| **序号** | **类描述** |
| 1 | Vector  该类和ArrayList非常相似，但是该类是同步的，可以用在多线程的情况，该类允许设置默认的增长长度，默认扩容方式为原来的2倍。 |
| 2 | Stack  栈是Vector的一个子类，它实现了一个标准的后进先出的栈。 |
| 3 | Dictionary  Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似。 |
| 4 | Hashtable  Hashtable 是 Dictionary(字典) 类的子类，位于 java.util 包中。 |
| 5 | Properties  Properties 继承于 Hashtable，表示一个持久的属性集，属性列表中每个键及其对应值都是一个字符串。 |
| 6 | BitSet 一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。 |

一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。

**集合算法**

集合框架定义了几种算法，可用于集合和映射。这些算法被定义为集合类的静态方法。

在尝试比较不兼容的类型时，一些方法能够抛出 ClassCastException异常。当试图修改一个不可修改的集合时，抛出UnsupportedOperationException异常。

集合定义三个静态的变量：EMPTY\_SET，EMPTY\_LIST，EMPTY\_MAP的。这些变量都不可改变。

|  |  |
| --- | --- |
| **序号** | **算法描述** |
| 1 | Collection Algorithms 这里是一个列表中的所有算法实现。 |

**如何使用迭代器**

通常情况下，你会希望遍历一个集合中的元素。例如，显示集合中的每个元素。

一般遍历数组都是采用for循环或者增强for，这两个方法也可以用在集合框架，但是还有一种方法是采用迭代器遍历集合框架，它是一个对象，实现了Iterator 接口或ListIterator接口。

迭代器，使你能够通过循环来得到或删除集合的元素。ListIterator 继承了Iterator，以允许双向遍历列表和修改元素。

|  |  |
| --- | --- |
| **序号** | **迭代器方法描述** |
| 1 | 使用 Java Iterator 这里通过实例列出Iterator和listIterator接口提供的所有方法。 |

**遍历 ArrayList**

**实例**

import java.util.\*;

public class Test{

public static void main(String[] args) {

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

list.add("Hello");

list.add("World");

list.add("HAHAHAHA"); //第一种遍历方法使用foreach遍历List

for (String str : list) {

//也可以改写for(int i=0;i<list.size();i++)这种形式

System.out.println(str); } //第二种遍历，把链表变为数组相关的内容进行遍历

String[] strArray=new String[list.size()];

list.toArray(strArray);

for(int i=0;i<strArray.length;i++) //这里也可以改写为 foreach(String str:strArray)这种形式 {

System.out.println(strArray[i]);

} //第三种遍历 使用迭代器进行相关遍历

Iterator<String> ite=list.iterator();

while(ite.hasNext())//判断下一个元素之后有值 {

System.out.println(ite.next());

}

}

}

**解析：**

三种方法都是用来遍历ArrayList集合，第三种方法是采用迭代器的方法，该方法可以不用担心在遍历的过程中会超出集合的长度。

**遍历 Map**

**实例**

import java.util.\*;

public class Test{

public static void main(String[] args) {

Map<String, String> map = new HashMap<String, String>();

map.put("1", "value1");

map.put("2", "value2");

map.put("3", "value3"); //第一种：普遍使用，二次取值

System.out.println("通过Map.keySet遍历key和value：");

for (String key : map.keySet()) {

System.out.println("key= "+ key + " and value= " + map.get(key)); } //第二种

System.out.println("通过Map.entrySet使用iterator遍历key和value：");

Iterator<Map.Entry<String, String>> it = map.entrySet().iterator();

while (it.hasNext()) {

Map.Entry<String, String> entry = it.next();

System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());

} //第三种：推荐，尤其是容量大时

System.out.println("通过Map.entrySet遍历key和value");

for (Map.Entry<String, String> entry : map.entrySet()) {

System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());

} //第四种

System.out.println("通过Map.values()遍历所有的value，但不能遍历key");

for (String v : map.values()) {

System.out.println("value= " + v);

}

}

}

**如何使用比较器**

TreeSet和TreeMap的按照排序顺序来存储元素. 然而，这是通过比较器来精确定义按照什么样的排序顺序。

这个接口可以让我们以不同的方式来排序一个集合。

|  |  |
| --- | --- |
| **序号** | **比较器方法描述** |
| 1 | 使用 Java Comparator 这里通过实例列出Comparator接口提供的所有方法 |

**总结**

Java集合框架为程序员提供了预先包装的数据结构和算法来操纵他们。

集合是一个对象，可容纳其他对象的引用。集合接口声明对每一种类型的集合可以执行的操作。

集合框架的类和接口均在java.util包中。

任何对象加入集合类后，自动转变为Object类型，所以在取出的时候，需要进行强制类型转换。

**Java 泛型**

Java 泛型（generics）是 JDK 5 中引入的一个新特性, 泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。

泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

*假定我们有这样一个需求：写一个排序方法，能够对整型数组、字符串数组甚至其他任何类型的数组进行排序，该如何实现？*

*答案是可以使用****Java 泛型****。*

*使用 Java 泛型的概念，我们可以写一个泛型方法来对一个对象数组排序。然后，调用该泛型方法来对整型数组、浮点数数组、字符串数组等进行排序。*

**泛型方法**

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

下面是定义泛型方法的规则：

* 所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的<E>）。
* 每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。
* 类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。
* 泛型方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像int,double,char的等）。

**实例**

下面的例子演示了如何使用泛型方法打印不同字符串的元素：

**实例**

public class GenericMethodTest {

// 泛型方法 printArray

public static < E > void printArray( E[] inputArray ) {

// 输出数组元素

for ( E element : inputArray ){

System.out.printf( "%s ", element );

}

System.out.println();

}

public static void main( String args[] ) {

// 创建不同类型数组： Integer, Double 和 Character

Integer[] intArray = { 1, 2, 3, 4, 5 };

Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };

Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println( "整型数组元素为:" );

printArray( intArray ); // 传递一个整型数组

System.out.println( "\n双精度型数组元素为:" );

printArray( doubleArray ); // 传递一个双精度型数组

System.out.println( "\n字符型数组元素为:" );

printArray( charArray ); // 传递一个字符型数组

}

}

编译以上代码，运行结果如下所示：

整型数组元素为:

1 2 3 4 5

双精度型数组元素为:

1.1 2.2 3.3 4.4

字符型数组元素为:

H E L L O

有界的类型参数:

可能有时候，你会想限制那些被允许传递到一个类型参数的类型种类范围。例如，一个操作数字的方法可能只希望接受Number或者Number子类的实例。这就是有界类型参数的目的。

要声明一个有界的类型参数，首先列出类型参数的名称，后跟extends关键字，最后紧跟它的上界。

**实例**

下面的例子演示了"extends"如何使用在一般意义上的意思"extends"（类）或者"implements"（接口）。该例子中的泛型方法返回三个可比较对象的最大值。

**实例**

public class MaximumTest {

// 比较三个值并返回最大值

public static <T extends Comparable<T>> T maximum(T x, T y, T z) {

T max = x; // 假设x是初始最大值

if ( y.compareTo( max ) > 0 ){

max = y; //y 更大

}

if ( z.compareTo( max ) > 0 ){

max = z; // 现在 z 更大

}

return max; // 返回最大对象

}

public static void main( String args[] ) {

System.out.printf( "%d, %d 和 %d 中最大的数为 %d\n\n", 3, 4, 5, maximum( 3, 4, 5 ) );

System.out.printf( "%.1f, %.1f 和 %.1f 中最大的数为 %.1f\n\n", 6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

System.out.printf( "%s, %s 和 %s 中最大的数为 %s\n","pear", "apple", "orange", maximum( "pear", "apple", "orange" ) ); } }

编译以上代码，运行结果如下所示：

3, 4 和 5 中最大的数为 5

6.6, 8.8 和 7.7 中最大的数为 8.8

pear, apple 和 orange 中最大的数为 pear

**泛型类**

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。

和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

**实例**

如下实例演示了我们如何定义一个泛型类:

**实例**

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("菜鸟教程"));

System.out.printf("整型值为 :%d\n\n", integerBox.get());

System.out.printf("字符串为 :%s\n", stringBox.get());

}

}

编译以上代码，运行结果如下所示：

整型值为 :10

字符串为 :菜鸟教程

**类型通配符**

1、类型通配符一般是使用?代替具体的类型参数。例如 **List<?>** 在逻辑上是**List<String>,List<Integer>** 等所有List<具体类型实参>的父类。

**实例**

import java.util.\*;

public class GenericTest {

public static void main(String[] args) {

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

List<Integer> age = new ArrayList<Integer>();

List<Number> number = new ArrayList<Number>();

name.add("icon");

age.add(18);

number.add(314);

getData(name);

getData(age);

getData(number);

}

public static void getData(List<?> data) {

System.out.println("data :" + data.get(0));

}

}

输出结果为：

data :icon

data :18

data :314

**解析：** 因为getData()方法的参数是List类型的，所以name，age，number都可以作为这个方法的实参，这就是通配符的作用

2、类型通配符上限通过形如List来定义，如此定义就是通配符泛型值接受Number及其下层子类类型。

**实例**

import java.util.\*;

public class GenericTest {

public static void main(String[] args) {

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

List<Integer> age = new ArrayList<Integer>();

List<Number> number = new ArrayList<Number>();

name.add("icon");

age.add(18);

number.add(314); //getUperNumber(name);//1

getUperNumber(age);//2

getUperNumber(number);//3

}

public static void getData(List<?> data) {

System.out.println("data :" + data.get(0));

}

public static void getUperNumber(List<? extends Number> data) {

System.out.println("data :" + data.get(0));

}

}

输出结果：

data :18

data :314

**解析：**在(//1)处会出现错误，因为getUperNumber()方法中的参数已经限定了参数泛型上限为Number，所以泛型为String是不在这个范围之内，所以会报错

3、类型通配符下限通过形如 **List<? super Number>**来定义，表示类型只能接受Number及其三层父类类型，如Objec类型的实例。