1. Collection相关概念
2. 定义

Java容器里**只能放对象**，对于基本类型（int, long, float, double等），需要将其包装成对象类型后（Integer, Long, Float, Double等）才能放到容器里。很多时候拆包装和解包装能够自动完成。

1. 泛型（Generics）
2. 定义

Java容器能够容纳任何类型的对象，这一点表面上是通过泛型机制完成，Java泛型不是什么神奇的东西，只是编译器为我们提供的一个“语法糖”，泛型本身并不需要Java虚拟机的支持，只需要在编译阶段做一下简单的字符串替换即可。实质上Java的单继承机制才是保证这一特性的根本，因为所有的对象都是Object的子类，容器里只要能够存放Object对象就行了。

事实上，所有容器的内部存放的都是Object对象，泛型机制只是简化了编程，由编译器自动帮我们完成了强制类型转换而已。

1. 泛型原理（重要！！）★
2. 内存管理
3. 容器中的内存管理实质

跟C++复杂的内存管理机制不同，Java GC自动包揽了一切，Java程序并不需要处理令人头疼的内存问题，因此JCF并不像C++ STL那样需要专门的空间适配器（alloctor）。

另外，由于Java里对象都在堆上，且对象只能通过引用（reference，跟C++中的引用不是同一个概念，可以理解成经过包装后的指针）访问，容器里放的其实是对象的引用而不是对象本身，也就不存在C++容器的复制拷贝问题。

1. 容器相关接口和实现
2. 14种容器接口



1. 通用实现表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | Implementations | | | | |
| Hash Table  (哈希表) | Resizable Array  (变长数组) | Balanced Tree | Linked List  (链表) | Hash Table + Linked List |
| Interfaces | Set | HashSet |  | TreeSet |  | LinkedHashSet |
| List |  | ArrayList |  | LinkedList |  |
| Deque |  | ArrayDeque |  | LinkedList |  |
| Map | HashMap |  | TreeMap |  | LinkedHashMap |

1. 迭代器（Iterator）

5.1 迭代器概念

1. 定义：

迭代器（iterator），是一个用来选取容器中的元素，并把它呈现给迭代器用户的对象。作为一个类，它提供了某种抽象层次，可以用来把容器的细节从访问容器的代码中分离出来。容器通过迭代器被抽象为仅仅是一个序列（sequence），迭代器允许你遍历这个序列而不用担心底层的结构。

1. 二要素：
2. 迭代器必定从属于某个容器，其作用是用来遍历所属容器中的元素。
3. 迭代器是在容器的数据视图之上进行迭代，因此**不能在迭代过程中修改容器中的数据，否则会抛出异常**！除非使用迭代器的专用方法对数据进行修改
4. 注意事项：
5. Java的迭代器只在Collection中有，而Map没有迭代器，它有不同的迭代方法。
6. 就是用统一的方法来迭代不同类型的集合！可能由于不同集合的内部数据结构不尽相同，如果要自己纯手工迭代的话相互之间会有很大的差别，而迭代器的作用就是**统一的方法对不同的集合进行迭代，而在迭代器底层隐藏不同集合之间的差异，从而为迭代提供最大的方便**；
7. Iterator接口和Iterable区别

实现Iterator接口称为迭代器类，位于包java.util。实现Iterable接口的类，可以通过方法得到一个迭代器对象，位于java.lang，方法为：iterator()

* 1. Iterator接口常用方法

1. boolean hasNext();

是否还有下一个元素（肯定有一个位置指针维护当前迭代的位置），如果有元素返回true。

1. Object next();

返回迭代中的下一个元素

1. void remove();

从容器中删除当前元素（即上一个next代表的那个元素），直接会改变容器中的数据！

1. 获取iterator对象

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

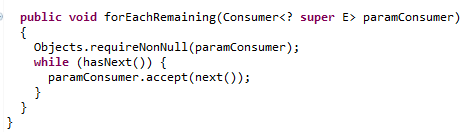
Iterator<String> iterator = list.iterator();

1. 注意：

实际上，Iterator迭代的“集合”是真正集合的视图，**视图和真实数据之间是一一映射的关系**。如果此时使用非迭代器方法对真实数据进行修改就会导致真实数据和映像之间不一致，因此会抛出异常。而迭代器的修改方法可以保证这种映射的一致性，即迭代器先对视图进行修改，然后将视图的修改更新到真实数据。但是反向就是无效的，因为映像自己是知道关联的是哪个真实数据，但是真实数据本身不知道有哪些映像和我关联的。即真实数据永远是被动的，而映像是主动的！

* 1. 遍历方法forEachRemaining()

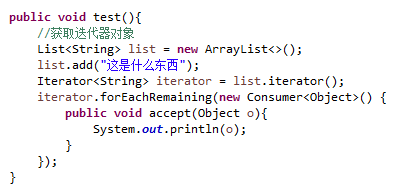
1. 源码：



* 里面是一个函数闭包Consumer action，用来定义对当前迭代变量进行何种操作。

**Java函数闭包的概念！！**

1. 实例代码：



里面的函数paramConsumer是接受每次迭代的变量，相当于代码"Type var = it.next()“中的var，同样也是数据视图，**不要在paramConsumer中用尝试修改集合中的元素，否则会报错**。