# Java技能点

## 对象克隆

在Java中，我们讨论两个Java对象是否相等，通常有两个层次：

1. 通过equals方法比较两个对象。只要两个对象的内容一致，就返回true；
2. 通过 == 号判断两个对象是否相等时，只有当两个对象存储的引用指向内存堆中的同一个位置，两个对象才是相等的。实际上，它们共享同一个对象。

当两个对象通过赋值符号 = 赋值时，表明这两个对象指向了内存中同一个地址，所以改变其中一个对象的内容，也就间接地改变了另一个对象的内容。比如，对象A指向内存中的2315位置，这时，创建一个对象Object B = A，此时内存中并没有新建一个和A相同的对象，而是直接让B也指向了内存中的2315位置。这时候，A和B实际上共享了同一个对象，无论是A还是B修改了对象内容，对应的B或A也会对应改变。

有时候，我们需要从一个已经存在的对象重新拷贝一份出来，并且不仅这两个对象内容相等，在内存中存在两个独立的存储地址，互不影响，这时，就需要用到 Java 中的克隆机制。

如果要实现Java对象的克隆，需要在定义Java类时实现Cloneable接口并重写clone方法。Cloneable接口中没有任何方法，只是一个标识接口。其作用是告诉JRE这些接口的实现类是否具有某个功能，如是否支持克隆、是否支持序列化等。而clone方法是Object类提供的，Object类时所有类的祖先。

**public interface** Cloneable {}

### 为什么要克隆

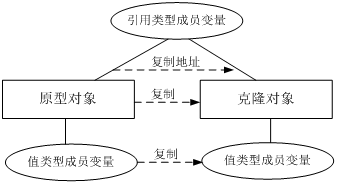
为什么需要克隆一个对象呢？这时因为在实际代码编程中，我们时常需要对一个已存在的对象进行操作，但我们又不想改变原来的对象。这时候克隆无疑是一个很好的选择。当然，我们也可以new一个对象，然后调用原对象的get方法，set进新建的对象，但毫无疑问，这种代码是繁琐且毫无美感的，并且完全不如clone方法克隆一个对象高效。

### 浅克隆与深克隆

#### 浅克隆

在浅克隆中，如果原型对象的成员变量是值类型，将复制一份给克隆对象；如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址。

简单来说，在浅克隆中，当对象被复制时只复制它本身和其中包含的值类型的成员变量，而引用类型的成员对象并没有复制。

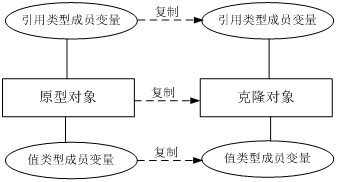


在Java语言中，通过覆盖Object类的clone()方法可以实现浅克隆。

#### 深克隆

在深克隆中，无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象，深克隆将原型对象的所有引用对象也复制一份给克隆对象。

简单来说，在深克隆中，除了对象本身被复制外，对象所包含的所有成员变量也将复制。



在Java语言中，如果需要实现深克隆，可以通过覆盖Object类的clone()方法实现，也可以通过序列化(Serialization)等方式来实现。

（如果引用类型里面还包含很多引用类型，或者内层引用类型的类里面又包含引用类型，使用clone方法就会很麻烦。这时我们可以用序列化的方式来实现对象的深克隆。）

序列化就是将对象写到流的过程，写到流中的对象是原有对象的一个拷贝，而原对象仍然存在于内存中。通过序列化实现的拷贝不仅可以复制对象本身，而且可以复制其引用的成员对象，因此通过序列化将对象写到一个流中，再从流里将其读出来，可以实现深克隆。需要注意的是能够实现序列化的对象其类必须实现Serializable接口，否则无法实现序列化操作。

### 对象克隆的实现

浅克隆：

当对象中没有引用变量时，直接使用浅克隆即可（int、long、char等基本类型对应的Java类Integer、Long、Character和String类型使用浅克隆即可，虽然它们也是对象）。

首先创建一个Java对象Student，它包含3个属性，id、name、age。

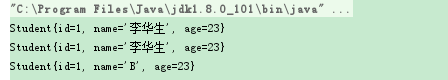
**public class** Student **implements** Cloneable {  
  
**private** Long **id**;  
**private** String **name**;  
**private** Long **age**;  
@Override  
**public** Object clone(){  
 Student student = **null**;  
 **try** {  
 student = (Student) **super**.clone();  
 } **catch** (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 **return** student;  
}

}

然后编写一个测试类，测试浅克隆和直接赋值：

@org.junit.Test  
**public void** testClone(){  
 Student A = **new** Student(1L, **"李华生"**, 23L);  
 Student B = A;  
 Student C = (Student) A.clone();  
 System.***out***.println(A.toString());  
 C.setName(**"C"**);  
 System.***out***.println(A.toString());  
 B.setName(**"B"**);  
 System.***out***.println(A.toString());  
}

测试类打印结果：



可以看出，赋值是对象共享，而克隆是获取一个具有相同内容的新对象。

深克隆：

1. 深克隆时，对象和它内部的引用对象类都需要实现Clone接口并重写clone方法；
2. 对象的clone方法中需要调用引用对象的clone方法，实现对引用对象的克隆；

新建一个Teacher类，有id和name属性，同样Teacher需要实现Cloneable接口，重写clone方法。

**public class** Teacher **implements** Cloneable {  
 **private** Long **id**;  
 **private** String **name**;  
  
 @Override  
 **public** Object clone() **throws** CloneNotSupportedException {  
 Teacher teacher = **null**;  
 teacher = (Teacher)**super**.clone();  
 **return** teacher;  
 }

}

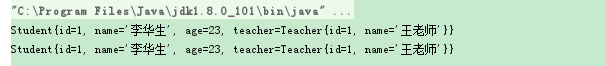
在Student中添加一个Teacher属性，并在Student类中调用Teacher类的clone方法：

@Override  
**public** Object clone(){  
 Student student = **null**;  
 **try** {  
 student = (Student) **super**.clone();  
 **if**(**teacher** != **null**){  
 student.**teacher** = (Teacher)**teacher**.clone();  
 }  
 } **catch** (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 **return** student;  
}

然后编写测试类进行测试：

@org.junit.Test  
**public void** testClone(){  
 Teacher teacher = **new** Teacher(1L, **"王老师"**);  
 Student A = **new** Student(1L, **"李华生"**, 23L, teacher);  
 Student B =(Student) A.clone();  
 System.***out***.println(A.toString());  
 B.getTeacher().setName(**"B"**);  
 System.***out***.println(A.toString());  
}

测试代码执行结果：



从结果发现，对象B修改了它的引用对象属性，但是A没有改变。说明A和B的引用对象不是同一个引用对象。如果在Student类的clone方法中不调用Teacher的clone方法，这里输出的对象A的Teacher的name属性值会变成B。

### 多层克隆

当要克隆的对象是多层的时候，即克隆对象中包含引用对象类型，引用对象中又包含引用对象，这种时候使用clone机制实现对象的克隆是非常麻烦的。这种时候，我们可以选择使用序列化的方式实现对象的深克隆。

上面的Student类中引入了Teacher类，已经是两层的克隆了。这里我们使用序列化方式实现对象克隆。

首先，让Student和Teacher类实现Serializable接口，并去掉内部的clone方法：

**public class** Student **implements** Serializable{

**private static final long *serialVersionUID*** = -7638855613264336693L;

…

}

**public class** Teacher **implements** Serializable{

**private static final long *serialVersionUID*** = -9188379133266222018L;

…

}

然后在Student类（外部类）中编写一个自定义的对象克隆方法：

**public** Student myClone() **throws** IOException, ClassNotFoundException {  
 Student student = **null**;  
 ByteArrayOutputStream baos = **new** ByteArrayOutputStream();  
 ObjectOutputStream oos = **new** ObjectOutputStream(baos);  
 oos.writeObject(**this**);  
 *// 将流序列化成对象* ByteArrayInputStream bais = **new** ByteArrayInputStream(baos.toByteArray());  
 ObjectInputStream ois = **new** ObjectInputStream(bais);  
 student = (Student)ois.readObject();  
 **return** student;  
}

测试类编写测试代码：

@org.junit.Test  
**public void** testClone() **throws** IOException, ClassNotFoundException {  
 Teacher teacher = **new** Teacher(1L, **"王老师"**);  
 Student A = **new** Student(1L, **"李华生"**, 23L, teacher);  
 Student B = A.myClone();  
 System.***out***.println(A.toString());  
 B.getTeacher().setName(**"B"**);  
 System.***out***.println(A.toString());  
}

执行结果：

C:\Users\hand\AppData\Local\Temp\1516095184(1).png

# 反射机制

<https://www.cnblogs.com/bojuetech/p/5896551.html>

<http://blog.csdn.net/nieweilin/article/details/5908165>

<https://www.cnblogs.com/hongxinlaoking/p/4684652.html>

<https://www.cnblogs.com/hxsyl/archive/2013/03/23/2977593.html>

forName：<http://blog.csdn.net/fengyuzhengfan/article/details/38086743>

## 反射定义

什么是反射机制

Java反射机制：就是在运行状态中，对于任意一个类，我们都能够知道这个类的属性和方法；对于任意一个对象，我们都能够调用它的属性和方法。这种动态的获取类信息以及动态的调用对象方法的功能称为Java语言的反射机制。

反射机制的应用

生成动态代理、面向切面编程、以及自定义注解的使用解析等。

## 反射机制的原理

## 反射基础元素

### 类加载器

### Class类

Java是面向对象编程，而几乎每个对象都是一个Class对象。

1. Class是用来描述类的，里面包含了很多获取类信息的方法，比如获取类中的方法（Method）、字段（Field）和构造器（Constructor）等。
2. 一个类可以有多个实例（可人为创建），但在JVM中，一个Class只有一个对应的Class实例，包含该Class定义的信息（只能由系统创建）。
3. 使用Class的newInstance方法获取类实例时，该Class必须提供一个无参的构造方法。

获取JVM中Class唯一实例的三种方式：

1. 直接通过类名 + ‘.class’的方式获取：

Class classInstance = PubPartyContact.**class**;

1. 通过Class的forName方法获取，需要传入全类名作为参数，并抛出ClassNotFoundException：

Class classInstance = Class.*forName*(**"om.core.order.dto.PubPartyContact"**);

1. 通过类实例的getClass方法获取：

PubPartyContact dto = **new** PubPartyContact();  
Class classInstance = dto.getClass();

。

### Constructor类

### Method类

Invoke方法

### Field类

### Annotation类

declaredFields方法获取所有声明属性，包括公有和私有

getFields方法不能获取类中的私有属性

JAVA 反射机制中，Field的getModifiers()方法返回int类型值表示该字段的修饰符。

其中，该修饰符是java.lang.reflect.Modifier的静态属性。

对应表如下：

PUBLIC: 1

PRIVATE: 2

PROTECTED: 4

STATIC: 8

FINAL: 16

SYNCHRONIZED: 32

VOLATILE: 64

TRANSIENT: 128

NATIVE: 256

INTERFACE: 512

ABSTRACT: 1024

STRICT: 2048

# 容器集合

在Java中，如果有一个类专门用来存放其他类的对象，那么这个类就叫作容器或者集合。

容器就类似与我们生活中的收纳盒，可以将各种物品按照类别分别放进不同的收纳盒，方便以后取用。集合也是这样，用于存放Java中各种类型的对象。并且这些对象也需要归类存放，比如一个集合只能存储某个类的不同实例，或只能存储实现同一接口的不同实现类的实例。其实是为了方便从集合中取出元素时，我们能够知道取出的元素属于哪个类，可以干嘛。

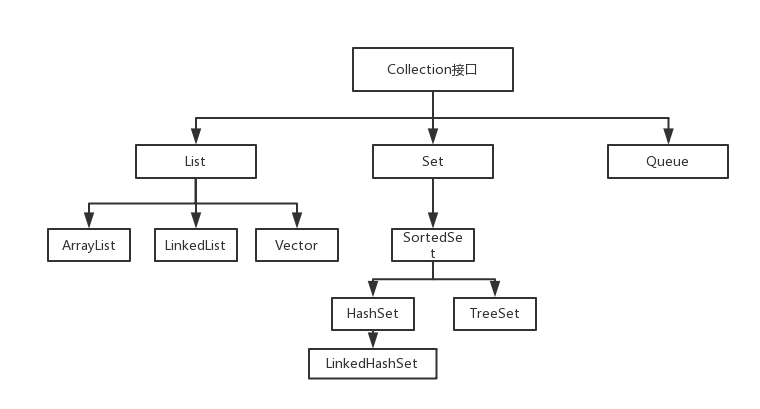
因此：集合就是将若干性质相同或相近的类组合在一起形成的一个整体，这个整体就是集合。

集合有两个基本的接口：Collection和Map。

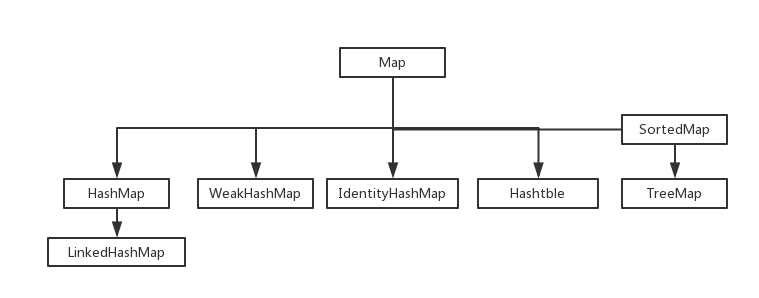
#### Fail-Fast机制

## 容器的结构

Collection包括List、Set和Queue。Map虽然没有Collection接口，但也属于集合。



Map接口：



## Iterator

Iterator是Java中提供的一个专门的迭代器，主要用于对集合进行遍历操作的。

如果我们自定义了一个序列类，则可以为这个序列类实现Iterator接口，然后就可以通过标准的Java迭代器对我们自定义的序列进行遍历了。

### Iterable接口

Java中还提供了一个Iterable接口。Iterable接口中只有一个抽象方法iterator，通过该方法，Iterable接口的任何实现类都可以返回一个标准的Iterator（迭代器）对象。

在Java中，我们常用的实现了Iterable接口的子接口有Collection, Deque, List, Queue, Set等。任何实现Iterable接口的实现类对象都可以成为for-each语句块的目标，也就是可以通过for-each遍历循环实现类的底层序列。

Iterator<T> iterator();

在Java 8中，为Iterable接口提供了一个默认的forEach方法，因此该接口的所有实现类现在可以直接调用forEach方法对底层序列进行遍历。

**default void** forEach(Consumer<? **super** T> action)

总结来看：Iterable接口主要有两个作用：一是为实现类提供一个标准的Java迭代器对象；二是提供了forEach方法遍历实现类的底层序列。

### Iterator

Iterator用来对集合进行遍历，但是在遍历期间不能通过集合自身对集合执行增删操作（可以通过Iterator对象的remove方法删除集合中的元素）。否则会抛出异常java.util.ConcurrentModificationException。

#### Iterator原理

迭代器是工作在一个独立的线程中，并且拥有一个mutex锁，就是说iterator在工作的时候，是不允许被迭代的对象被改变的。

Iterator被创建的时候回创建一个内存索引链表（单链表），这个索引表和原来的对象一一对应，当原来的集合数量改变的时候，这个索引表的内容没有同步改变，所以当索引指针往下移的时候找不到对应的对象，就会抛出异常。

Iterator是一个单序不可变，只能顺序读取，不能逆序操作的数据结构。

集合返回Iterator时，Iterator的指针指向集合中第一个元素的前面，当调用next方法时，指针开始向下移动，指向集合中的第一个元素。调用remove方法时，删除的就是指针指向的对象。

#### Iterator的方法

要使用Iterator的方法，首先我们需要获取Iterator对象。Java提供了一个Iterable接口，通过实现该接口，我们可以获取一个类的迭代器对象。

**boolean** hasNext();

判断集合中是否有下一个元素。如果有，则返回true，没有则返回false。

E next();

获取集合中的下一个元素。如果不存在，则抛出异常java.util.NoSuchElement。

**default void** remove()

移除迭代器指针指向的集合中的元素。如果不存在，则抛出异常。

**default void** forEachRemaining(Consumer<? **super** E> action)

遍历集合中剩余的元素。

### ListIterator

ListIterator是Iterator的子类，它实现了Iterator接口。除了拥有Iterator中的方法外，还有自己特有的方法。

**public interface** ListIterator<E> **extends** Iterator<E>

ListIterator也是用于对集合进行遍历。只是它的使用范围没有集合那么广，它只能在List和List的子类中使用。并且，在遍历期间，集合不能通过它自身增删集合中的元素，但是ListIterator提供了remove、set和add方法实现遍历时对集合的修改操作。

#### ListIterator的方法

Iterator是一个单向的链表，只能向一个方向遍历，但ListIterator是双向的。

**boolean** hasNext();

判断集合中是否存在下一个元素。

E next();

获取集合中的下一个元素。

**boolean** hasPrevious();

判断集合中是否存在上一个元素。以当前指针位置为准。

E previous();

返回集合中的上一个元素。

**int** nextIndex();

返回集合下一元素的索引。

**int** previousIndex();

返回集合上一元素的索引。

**void** remove();

移除集合中当前指针指向的元素。

**void** set(E e);

为集合中当前指针指向的元素设置为新元素（替换掉原来的）。

ListIterator iterator = dataList.listIterator();  
**while** (iterator.hasNext()){  
 **if**(iterator.next().equals(**"1"**)){  
 iterator.set(**"J"**);  
 }  
}

上述代码将集合中的元素1改为了J。

**void** add(E e);

向集合中新增一个元素。在当前指针指向的下一个位置新增。

### Iterator和ListIterator的区别

1. Iterator可以用于所有的集合，包括Map。而ListIterator只能用于List和List的实现类。
2. Iterator只有remove方法可以移除集合中的元素，而ListIterator除了可以通过remove方法删除集合元素外，还可以通过set方法修改集合元素，add方法新增集合元素。
3. Iterator只能顺序向后遍历，而ListIterator除了可以顺序遍历外，还可以逆序遍历。
4. ListIterator可以通过nextIndex和previousIndex方法定位当前的索引位置。

## Comparator

## 重写HashCode和equals方法

## Collection

Collection接口是Java容器的顶级接口之一，基本上所有的集合类都实现了该接口。

**public interface** Collection<E> **extends** Iterable<E>

由Collection源码可以看出，Collection是一个接口类，它继承了Java迭代接口Iterable。

Collection作为容器接口，提供了一些操作容器内元素的通用方法。主要是增删改查：

**boolean** add(E e);

用于向集合中添加一个元素。

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

将一个集合中的所有元素添加到另一个集合中。

**boolean** remove(Object o);

将集合中的某个指定元素移除。如果对象不存在，则返回false。

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

该方法用于从一个集合中移除另一个集合。比如集合A包含集合B，则可以用该方法将删除集合A中的B集合的部分。

注意：使用removeAll方法时，如果存储的是Java对象，必须重写Java对象的equals方法(一般重写equals方法时也会将hashCode方法重写)。因为在移除元素时，会比较元素对象是否相等。

**default boolean** removeIf(Predicate<? **super** E> filter)

可以删除集合中不满足指定条件的元素。Java 8中提供的新方法，可以传入一个函数作为方法参数，在函数中编写过滤条件。然后对集合进行遍历，将所有满足过滤条件的元素过滤掉。

Iterator<E> iterator();

返回集合的迭代器对象，通过返回的迭代器对象，可以获取集合的下一个元素或对集合进行遍历操作。

**void** clear();

用于清空集合中的元素。

**default** Stream<E> stream()

将集合以流的形式返回。然后通过操作流来操作原集合中的数据。Stream是Java 8中提供的新特性。

**default** Stream<E> parallelStream()

也是将集合以流的方式返回，不过返回的是一个并行流。处理速度更快，不过要注意线程问题。

Object[] toArray();

将集合以数组形式返回。

**boolean** contains(Object o);

判断集合是否包含制定元素。

### List

#### ArrayList

ArrayList是基于动态数组实现的。也就是它本质上是一个动态数组。

在开始创建ArrayList时，它会创建一个默认容量为10的空数组，然后每次向ArrayList中添加元素的时候，ArrayList都会校验数组的容量是否够用，如果不够，就创建一个新的数组，然后调用Arrays的copyOf方法将旧数组的元素复制到新数组中。、

ArrayList和Vector的底层实现都是数组，它们的区别如下：

1. ArrayList在内存不够时默认是扩展50% + 1个，Vector是默认扩展1倍。
2. Vector提供indexOf(obj, start)接口，ArrayList没有。
3. Vector属于线程安全级别的，但是大多数情况下不使用Vector，因为线程安全需要更大的系统开销。

注意：

1. ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低。
2. 在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。
3. ArrayList是线程不安全的，只能在单线程环境下使用。在多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类。

##### ArrayList实现

ArrayList定义：

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

由定义可知，ArrayList实现了List、RandomAccess、Cloneable和Serializable接口。

ArrayList实现了Serializable接口，因此它支持序列化，能够通过序列化传输，实现了RandomAccess接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了Cloneable接口，能被克隆。

###### 私有属性

ArrayList中声明了两个私有属性：

**transient** Object[] **elementData**;

**private int size**;

其中，elementData是一个Object类型的数组，用来存储ArrayList中的元素，而size则用来表示ArrayList中元素的数量。

transient关键字：

Java的serialization提供了一种持久化对象实例的机制。但当持久化对象时，可能有某个特殊的对象数据成员，我们不想用serialization机制来保存它。此时，为了在某个特定对象的一个域上关闭serialization，就可以在这个域前加上关键字transient。

也就是序列化时，对transient关键字标注的对象成员不起作用。

###### 构造方法

ArrayList提供了3种构造方法：

**public** ArrayList()

**public** ArrayList(**int** initialCapacity)

**public** ArrayList(Collection<? **extends** E> c)

第一种可以构造一个默认初始容量为10的空列表；第二种可以构造一个指定初始容量的空列表；第三种可以构造一个包含指定collection的元素的列表，这些元素按照该collection的迭代器返回它们的顺序排列的。

###### 添加元素

ArrayList是一个集合容器，主要的操作其实是对容器内元素的增删改查。

**public** E set(**int** index, E element)

set方法可以会用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。

**public boolean** add(E e)

**public void** add(**int** index, E element)

add方法有两种。第一种是直接将元素添加到ArrayList的末尾；第二种方式用于在指定位置添加一个元素，如果这个位置已经有了元素，则会将该位置和它后面的所有元素后移一位，同时ArrayList的size加1。

**public boolean** addAll(Collection<? **extends** E> c)

**public boolean** addAll(**int** index, Collection<? **extends** E> c)

addAll方法和add方法类似，不过是批量添加元素。第一种是将集合中的元素添加到ArrayList的末尾；第二种是将集合中的元素从ArrayList的指定位置开始插入，同样，如果原位置处有元素，则该元素和它后面的所有元素后移一位。

Add方法中先调用了ensureCapacity(size+1)方法，之后将元素的索引赋给elementData[size]，而后size自增。例如初次添加时，size为0，add将elementData[0]赋值为e，然后size设置为1（类似执行以下两条语句elementData[0]=e;size=1）。将元素的索引赋给elementData[size]不是会出现数组越界的情况吗？这里关键就在ensureCapacity(size+1)中了。

###### 获取元素

获取元素一般是根据索引获取元素，因为ArrayList本身是一个数组，所以其实根据索引获取元素就是元素根据下标取指定位置的元素。

**public** E get(**int** index)

E elementData(**int** index)

上面两种方法都是根据索引获取指定位置的元素。不同的是：get方法会先校验index是否合法，也就是数组越界；而elementData方法不会校验。所以，一般用get方法为好。

**public boolean** contains(Object o)

**public int** indexOf(Object o)

两种方法一般配合使用比较好。indexOf用于返回指定元素的下标，前提是元素存在。

**public int** lastIndexOf(Object o)

如果ArrayList中有多个相同的元素，使用lastIndexOf方法可以返回ArrayList中最后一个该元素索引。文件后缀名获取时使用的就是这个方法。

###### 移除元素

ArrayList移除元素实际上是先获取指定位置上的元素，然后调用System的arraycopy方法复制数组，arraycopy方法可以指定复制数组的区间，从而过滤掉要删除的元素。

**public** E remove(**int** index)

**public boolean** remove(Object o)

移除指定的元素，不同的是第一种方法会返回移除的那个元素。

**protected void** removeRange(**int** fromIndex, **int** toIndex)

移除ArrayList中某个区间内的元素。

**public boolean** removeAll(Collection<?> c)

从一个ArrayList中移除某个集合中的所有元素，如果有的话。

注意：移除操作会通过对象的equals方法比较两个集合中的元素是否相等，因此当是我们定义的JavaBean时，一般需要重写equals和hashCode方法。

**public void** clear()

清空ArrayList，将数组elementData中的元素置为null，同时设置size为0。

###### 遍历集合

ArrayList实现了List接口，而List接口提供了多种遍历方法：

**public void** forEach(Consumer<? **super** E> action)

forEach方法通过一个消费型接口Consumer会对集合中的每个元素进行遍历。类似于流Stream中的forEach方法。

**public** Iterator<E> iterator()

返回一个ArrayList的迭代器对象，用于遍历。

**public** ListIterator<E> listIterator()

List接口及子对象独有的遍历器，除了可以向后遍历，还可以向前遍历，另外可以获取到当前索引等。

**public** ListIterator<E> listIterator(**int** index)

从集合中的指定索引出返回一个迭代器，也就是返回一个只包含原集合部分元素的子集合的迭代器。

###### 其它方法

**public** List<E> subList(**int** fromIndex, **int** toIndex)

切割集合，类似与String的subString方法。

**public void** trimToSize()

将底层数组的容量调整为当前列表保存的实际元素的大小

**public void** sort(Comparator<? **super** E> c)

通过对象的equals方法对集合元素进行排序。

**public** Object clone()

复制一个集合对象。

**public** Object[] toArray()

调用Arrays.copyOf将返回一个数组，数组内容是size个elementData的元素，即拷贝elementData从0至size-1位置的元素到新数组并返回。

**public** <T> T[] toArray(T[] a)

如果传入数组的长度小于size，返回一个新的数组，大小为size，类型与传入数组相同。所传入数组长度与size相等，则将elementData复制到传入数组中并返回传入的数组。若传入数组长度大于size，除了复制elementData外，还将把返回数组的第size个元素置为空

###### 动态扩容

我们知道，每当ArrayList添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容通过一个公开的方法ensureCapacity(int minCapacity)来实现。在实际添加大量元素前，我也可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。

**public void** ensureCapacity(**int** minCapacity) {  
 **int** minExpand = (**elementData** != ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***)  
 **if** (minCapacity > minExpand) {  
 ensureExplicitCapacity(minCapacity);  
 }  
}

**private void** ensureExplicitCapacity(**int** minCapacity) {  
 **modCount**++;**if** (minCapacity - **elementData**.**length** > 0)  
 grow(minCapacity);  
}

**private void** grow(**int** minCapacity) {**int** oldCapacity = **elementData**.**length**;  
 **int** newCapacity = oldCapacity + (oldCapacity >> 1);  
 **if** (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;  
 **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  
 newCapacity = *hugeCapacity*(minCapacity);  
**elementData** = Arrays.*copyOf*(**elementData**, newCapacity);  
}

数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的1.5倍。这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。

**private void** ensureCapacityInternal(**int** minCapacity)

Object oldData[] = elementData;//为什么要用到oldData[]

乍一看来后面并没有用到关于oldData， 这句话显得多此一举！但是这是一个牵涉到内存管理的类， 所以要了解内部的问题。 而且为什么这一句还在if的内部，这跟elementData = Arrays.copyOf(elementData, newCapacity); 这句是有关系的，下面这句Arrays.copyOf的实现时新创建了newCapacity大小的内存，然后把老的elementData放入。好像也没有用到oldData，有什么问题呢。问题就在于旧的内存的引用是elementData， elementData指向了新的内存块，如果有一个局部变量oldData变量引用旧的内存块的话，在copy的过程中就会比较安全，因为这样证明这块老的内存依然有引用，分配内存的时候就不会被侵占掉，然后copy完成后这个局部变量的生命期也过去了，然后释放才是安全的。不然在copy的的时候万一新的内存或其他线程的分配内存侵占了这块老的内存，而copy还没有结束，这将是个严重的事情。

#### LinkedList

LinkedList的底层实现是一个双向链表结构，有一个头节点和一个尾节点，并且它的头节点中是不存储数据的。

1. LinkedList的节点上也可以和ArrayList一样存储null。所以在调用indexOf(Object o)和remove(Object o)方法时，源码中会遍历两遍，当o为null的时候，遍历LinkedList中是否存在null元素节点；当不为null的时候，调用对象的equals方法比较o和LinkedList中的元素。
2. LinkedList是非同步的。
3. LinkedList是一个双向队列，也可以被当做堆栈和队列使用。实现了List接口可以当做队列使用。双向队列支持双向遍历。比如index方法，就会先判断index和LinkedList的size的关系。如果index大于size的二分之一，则从最后向前遍历；如果小于，则从头向后遍历。

##### LinkedList实现

LinkedList定义：

**public class** LinkedList<E>  
 **extends** AbstractSequentialList<E>  
 **implements** List<E>, Deque<E>, Cloneable, java.io.Serializable

由上面定义可知，LinkedList继承了AbstractSequentialList抽象类，实现了List、Deque、Cloneable和Serializable接口。

1. Serializable接口主要用于序列化对象，以实现网络传输。
2. Cloneable接口中没有任何方法，但如果一个对象想要支持复制，则必须要实现Cloneable接口，类似于标识接口。实现Cloneable接口后，在类中重写Object类的clone方法，用于克隆对象。
3. Deque接口表示是一个双端队列，那么也意味着LinkedList是双端队列的一种实现，所以，基于双端队列的操作在LinkedList中全部有效。

为什么继承AbstractSequentialList抽象类?

AbstractSequentialList 实现了get(int index)、set(int index, E element)、add(int index, E element) 和 remove(int index)这些骨干性函数。降低了List接口的复杂度。这些接口都是随机访问List的，LinkedList是双向链表；既然它继承于AbstractSequentialList，就相当于已经实现了“get(int index)这些方法”。

此外，我们若想自定义一个列表，只需要扩展此类，并提供 listIterator() 和 size() 方法的实现即可。若要自定义一个不可修改的列表，则需要实现列表迭代器的 hasNext、next、hasPrevious、previous 和 index 方法。

###### 私有属性

LinkedList中定义了3个私有属性：

**transient int size** = 0;

**transient** Node<E> **first**;

**transient** Node<E> **last**;

size属性用来描述LinkedList中元素节点的数量，增加或移除元素节点时，会修改size的值；first和last分别表示LinkedList的头节点和尾节点，可以为null。

使用transient关键字修饰的属性在序列化的时候不会被序列化。

###### 内部类

**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;  
 }  
}

**static final class** LLSpliterator<E> **implements** Spliterator<E>

###### 构造方法

LinkedList中提供了两种构造方式，一种是使用默认构造方法创建一个空的LinkedList列表；另一种是传入一个集合，然后创建一个包含集合元素的LinkedList对象。

**public** LinkedList()

**public** LinkedList(Collection<? **extends** E> c)

第二种方式实际上先调用第一种无参构造方法创建一个空的LinkedList，然后调用addAll方法，将集合元素添加入LinkedList中。addAll方法有个有意思的地方是它不是直接对集合进行遍历，而是先将集合转换为数组，然后遍历数组，添加元素。这样做的原因是防止在addAll的过程中，集合的元素又发生了变化，尤其是多线程高并发时。

###### 添加元素

链表的组成单元是节点Node，它除了储存元素外，还存储了两个指向前一节点和后一节点的指针，通过指针可以访问它的相邻节点。添加和移除元素时就是通过修改节点中的前后指针指向的节点来实现的。

LinkedList提供了几个私有的link和unlink方法，用于修改节点的指针指向。而LinkedList的add、remove等方法本质上就是通过调用link和unlink方法实现的。

**private void** linkFirst(E e)

**void** linkLast(E e)

**void** linkBefore(E e, Node<E> succ)

linkFirst方法用于在链表头部添加一个新的元素；linkLast方法用于在链表尾部添加一个新的元素；linkBefore方法用于在链表的指定节点前添加一个元素，它会先根据传入的节点Node找到节点在LinkedList中对应的位置。这里我们分析下linkFirst方法即可：

**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**++;  
}

可以看到，在向链表头部添加新元素时：先把链表的原头元素first取出，然后新建一个节点，让这个节点的next指针指向原来的头节点。这样，新元素就成了LinkedList的头节点，接下来对first属性赋值并将链表中原头节点的prev指针指向新加的元素（如果原LinkedList中存在元素的话）。

**public void** addFirst(E e) linkFirst

**public void** addLast(E e) linkLast

**public boolean** add(E e) linkLast

**public void** add(**int** index, E element) index == size -> linkLast; else linkBefore

**public boolean** addAll(Collection<? **extends** E> c)

**public boolean** addAll(**int** index, Collection<? **extends** E> c)

LinkedList提供了上述添加元素的方法，除了addAll是直接操作节点的指针添加元素外，其他的都是调用linkLast、linkFirst和linkBefore方法。

**public boolean** offer(E e)

**public boolean** offerFirst(E e)

**public boolean** offerLast(E e)

offer系列方法内部直接调用了对应的add方法，只是返回值都是boolean型。

**public void** push(E e) addFirst

向链表的头部添加元素，push和pop方法的提供可以把链表当栈使用，类似与出栈、入栈。

**public** E set(**int** index, E element)

set方法会先找到index下标对应的节点，然后用形参中的元素代替节点中原来的元素，同时会讲旧的元素返回。

###### 获取元素

相比较添加元素的形式统一，获取元素的形式就多种多样了。

Node<E> node(**int** index)

node方法是获取index下标对应的元素，且为了提高效率，node方法会先对index和size的一半进行大小比较。如果index大于size的一半，就从后向前遍历；如果小于size的一半，就从头向后遍历。

**public** E getFirst() first

**public** E getLast() last

**public** E get(**int** index) node，返回节点的item

get方法内部调用的是node方法，然后返回对应节点存储的数据，也就是节点Node的item元素。

**public** E element() getFirst

返回链表中的第一个节点的item值。

**public** E peekFirst() **return** (f == **null**) ? **null** : f.**item**;

**public** E peekLast() **return** (l == **null**) ? **null** : l.**item**;

**public** E peek() **return** (f == **null**) ? **null** : f.**item**;

返回链表的第一个节点的item值。如果为空链表，就返回一个null。

**public** E pollFirst() **return** (f == **null**) ? **null** : unlinkFirst(f);

**public** E pollLast() **return** (l == **null**) ? **null** : unlinkLast(l);

**public** E poll() **return** (f == **null**) ? **null** : unlinkFirst(f);

和peek相似，只是会在返回的同时移除对应的节点。比如poll会在返回链表第一个节点的同时移除链表中的第一个节点。

**public int** indexOf(Object o)

**public int** lastIndexOf(Object o)

indexOf和lastIndexOf方法都会对链表遍历，且如果o为null的时候，找到链表中为null的节点下标返回。lastIndexOf方法会从后向前遍历。

###### 移除元素

LinkedList中提供了3中unlink方法，操作节点的指针，移除链表中的元素。

**private** E unlinkFirst(Node<E> f)

**private** E unlinkLast(Node<E> l)

E unlink(Node<E> x)

是link系列方法的反过程。

remove方法会移除链表中的节点，主要是通过unlinkFirst、unlinkLast和unlink方法实现的。

**public** E removeFirst() unlinkFirst

**public** E removeLast() unlinkLast

**public** E remove() removeFirst

**public** E remove(**int** index) **return** unlink(node(index));

**public boolean** removeFirstOccurrence(Object o) remove

**public boolean** removeLastOccurrence(Object o) 从后遍向前历

**public** E pop() removeFirst

移除链表中的第一个元素，和push方法一起使用时工作过程类似与出栈、入栈。

###### 遍历链表

**public** ListIterator<E> listIterator(**int** index)

**public** Spliterator<E> spliterator()

###### 其它方法

**public** Object[] toArray()

**public** <T> T[] toArray(T[] a)

**public void** clear()

#### Vector

Vector和ArrayList相似，底层也是通过一个Object数组实现的。只不过Vector是线程安全的，这必然影响了它的执行效率。

Vector是Java 1.0提供的，现在基本不再使用。之所以Java集合类仍旧存在Vector，主要是为了提供对之前版本的支持。

### Set

#### HashSet

HashSet是基于HashMap实现的。在看源码之前，一直以为HashSet是用HashMap存储数据，然后在此基础上进行的扩展。然后看完源码后发现，HashSet无论构造方法还是元素增删改查，其实都是HashMap中对Key的操作。

Set只利用了Map的键key部分，因为Map中键是不可以重复的，所以Set中存储的值也不能重复。又因为Map中存储的是一个键值对，所以Set在存储元素的时候，也必须存入键值对。但是Set只需要键Key，所以Set中提供了一个空的对象，当存储元素时，将Key和这个空的对象存入Map。所以Map中的所有值都是同一个空对象，而Set的值就是Map的keySet方法返回的值。

##### 实现

HashSet定义：

**public class** HashSet<E>  
 **extends** AbstractSet<E>  
 **implements** Set<E>, Cloneable, java.io.Serializable

和List、Map一样，Set中也定义了一个AbstractSet抽象类，方便用户自定义Set，继承该抽象类，然后实现自己的增删改元素即可。

Set接口中提供了元素的增删改方法，需要子类自己实现。

##### 私有属性

HashSet中提供了两个私有属性：

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

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

第一个是HashSet的基石，HashSet的底层数据结构就是一个HashMap；

第二个是一个空的对象，HashSet只使用了HashMap的key部分特性，但是HashMap中存放的都是一个个键值对，所以HashSet中提供了一个空的对象PRESENT。在HashSet中存储元素时，会将一个key和PRESENT存入Map，也就是HashSet对应的Map中所有的键对应的值都是一个空的对象PRESENT。

##### 构造方法

HashSet提供了多种5种构造方法：

**public** HashSet()

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

**public** HashSet(**int** initialCapacity, **float** loadFactor)

**public** HashSet(**int** initialCapacity)

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

在HashSet的构造方法中其实就是创建一个Map对象，传递的参数可以指定Map的初始容量，加载因子等属性。

**public** HashSet(**int** initialCapacity, **float** loadFactor) {  
 **map** = **new** HashMap<>(initialCapacity, loadFactor);  
}

最后一种方式中传递了一个boolean型参数，但是在源码中没有任何与之相关的逻辑，也就是它是多余的：

HashSet(**int** initialCapacity, **float** loadFactor, **boolean** dummy) {  
 **map** = **new** LinkedHashMap<>(initialCapacity, loadFactor);  
}

仅仅是一个标识作用，通过这种方式声明的是一个LinkedHashMap对象。LinkedHashSet继承了HashSet，这个构造方法就视为LinkedHashSet准备的，创建LinkedHashSet对象时会调用这个构造方法。

##### 操作元素

添加元素：

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

添加元素就是调用Map的add方法添加一个键值对(值是一个空的对象)：

移除元素：

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

移除元素是调用Map的remove方法，通过键Key移除对应的键值对，同时将键值对对应的值返回：

遍历Set：

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

调用Map的keySet对应的迭代器。

**public** Spliterator<E> spliterator() {  
 **return new** HashMap.KeySpliterator<E,Object>(**map**, 0, -1, 0, 0);  
}

返回HashMap的Spliterator类型迭代器，可以指定一个int类型的参数，然后从这个参数之后的元素开始遍历。

Clone方法：

**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);  
 }  
}

采用的是深复制。与深复制对应的是浅复制，浅复制只会复制对象本身。对象内部如果有成员变量不是基础类型（这里String看做基础类型），则该成员变量不会被复制。比如类A中有一个成员变量是类B。现在创建一个A的实例S，然后调用A实例的clone方法复制出一个实例T，这时候T和S的属性值都是相同的，但是T和S是不同的对象，修改T的属性不会影响到S。但因为采用的是浅复制，实例A中有一个成员变量是一个对象B，这时候B没有被复制，也就是T和S共享B，它们存储的引用都指向对象B，无论T还是S修改了B属性，二者都会改变。

#### LinkedHashSet

LinkedHashMap继承了HashMap，是HashMap的子类。同样，LinkedHashSet继承了HashSet。

LinkedHashSet在HashSet的基础上没有提供新的方法，它的内部只有四个构造方法和一个spliterator遍历方法。

构造方法：

**public** LinkedHashSet()

**public** LinkedHashSet(**int** initialCapacity)

**public** LinkedHashSet(**int** initialCapacity, **float** loadFactor)

**public** LinkedHashSet(Collection<? **extends** E> c)

LinkedHashSet提供了上面四种方法，和HashSet一一对应，HashSet中的第五种方法就是为LinkedHashSet准备的。

**public** LinkedHashSet() {  
 **super**(16, .75f, **true**);  
}

LinkedHashSet在创建对象时，会传递一个boolean型参数，表明底层选择的实现是LinkedHashMap。

遍历：

**public** Spliterator<E> spliterator() {  
 **return** Spliterators.*spliterator*(**this**, Spliterator.***DISTINCT*** | Spliterator.***ORDERED***);  
}

返回一个迭代器。

#### TreeSet

### Queue

## Map

### HashMap

在JDK 1.6和1.7中，HashMap采用了数组+链表的底层实现。HashMap本质上是一个数组，只是数组中存储的元素都是链表。在HashMap存储元素的时候，会先计算键值对中key的哈希值，然后用得到的哈希值对数组长度求余，得到的余数是几，就将新的键值对存放到数组中对应下标的链表中。比如，HashMap在初始化时默认创建一个容量为16的数组，然后当向HashMap中添加元素的时候，会通过hash(key)%len计算元素应该被存入数组中的哪个位置。如key的hash值为28，然后28%16=12，那么这个元素就会被存入数组中下标为12的链表中。

但是我们知道，链表这种数据结构，它的存储空间是离散的，占用的内存宽松，空间复杂度很小，但时间复杂度很大。特点就是插入和删除数据比较简单，但是查找效率很低。因此，当HashMap中某个链表中存入的键值对很多时，通过key值查找元素的效率就相当的低。为了改善这个问题，JDK 1.8中引入了红黑树。

在JDK 1.8中，HashMap采用了数组+链表+红黑树的实现方式。当某个链表的长度大于阀值8时，就将数组中这个链表转换为红黑树，这样可以在一定程度上提高查询效率。

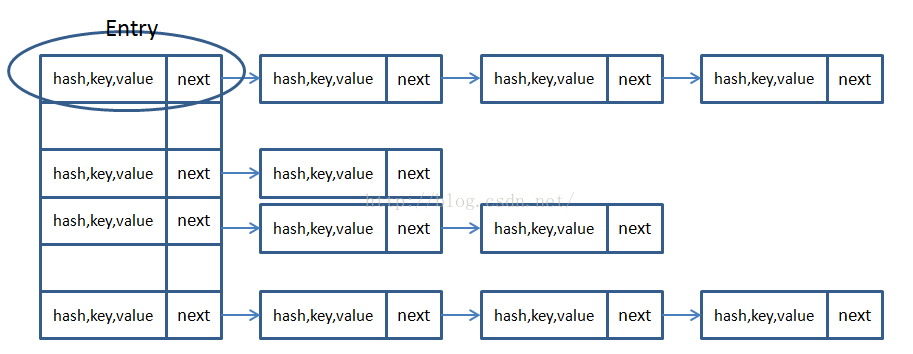
红黑树工作过程：链表中产生冲突的那些KEY对应的记录只是简单的追加到一个链表后面，这些记录只能通过遍历来进行查找。但是超过这个阈值后HashMap开始将列表升级成一个二叉树，使用哈希值作为树的分支变量，如果两个哈希值不等，但指向同一个数组下标的话，较大的那个会插入到右子树里。如果哈希值相等，HashMap希望key值最好是实现了Comparable接口的，这样它可以按照顺序来进行插入。这对HashMap的key来说并不是必须的，不过如果实现了当然最好。如果没有实现这个接口，在出现严重的哈希碰撞的时候，你就并别指望能获得性能提升了。

#### 哈希表

数组的存储空间是连续的，占用内存严重，空间复杂度很高，但是它的二分查找时间复杂度很小，只有O(1)。它的特点就是：添加和删除元素困难，但是查找效率很高。而链表刚好和数组相反。

既然这样，我们肯定会想得到一种二者的结合版，做出一种查找、删除和添加元素效率都不错的数据结构。这就是下面要提起的哈希表(Hash Table)了。既满足了数据的查找方便，同时不占用太多的内存空间，使用也十分方便。

哈希表有多种不同的实现方法，接下来介绍的是一种最常用的方法—— 拉链法，我们可以理解为“链表的数组”，它的结构图如下：



由图可以看出，哈希表最底层是一个数组，然后数组中的每个元素都是一个链表，链表中的存储的节点又是键值对结构Entry。Entry中有三个主要元素：键Key，值Value和指向下一节点的指针Next。

下面是链表节点Node的代码：

**static class** Node<K,V> **implements** Map.Entry<K,V> {  
 **final int hash**;  
 **final** K **key**;  
 V **value**;  
 Node<K,V> **next**;  
  
 Node(**int** hash, K key, V value, Node<K,V> next) {  
 **this**.**hash** = hash;  
 **this**.**key** = key;  
 **this**.**value** = value;  
 **this**.**next** = next;  
 }  
  
 **public final** K getKey() { **return key**; }  
 **public final** V getValue() { **return value**; }  
 **public final** String toString() { **return key** + **"="** + **value**; }  
  
 **public final int** hashCode() {  
 **return** Objects.*hashCode*(**key**) ^ Objects.*hashCode*(**value**);  
 }  
  
 **public final** V setValue(V newValue) {  
 V oldValue = **value**;  
 **value** = newValue;  
 **return** oldValue;  
 }  
 如果两个对象是否是同一个对象或者两个对象的键和值都相等，equals方法返回true  
 **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实现

HashMap定义：

**public class** HashMap<K,V> **extends** AbstractMap<K,V>  
 **implements** Map<K,V>, Cloneable, Serializable

由源码可以看出，HashMap继承了AbstractMap抽象类，实现了Map接口。至于Cloneable和Serializable接口，一个用于对象的克隆，一个对象序列化时使用。

**public abstract class** AbstractMap<K,V> **implements** Map<K,V>

Abstract抽象类也是Map的一个实现类。它为Map中操作元素的一些通用方法提供了实现。当我们需要自定义某种Map数据结构时，可以继承该抽象类，然后定义自己的特定需求即可。类似于List接口的实现类AbstractList。

#### HashMap私有属性

HashMap是一种组合的数据结构，应用场景也比ArrayList和LinkedList稍复杂，因此它的私有属性也较多：

哈希表，HashMap的链表数组

**transient** Node<K,V>[] **table**;

**transient** Set<Map.Entry<K,V>> **entrySet**;

**transient int size**;

被修改的次数，fast-fail机制

**transient int modCount**;

临界值 当实际大小(容量\*填充比)超过临界值时，会进行扩容

**int threshold**;

填充比，当哈希表中的实际容量/总容量达到填充比时，对HashMap扩容

**final float loadFactor**;

默认的初始容量

**static final int *DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4; *// aka 16*

最大容量

**static final int *MAXIMUM\_CAPACITY*** = 1 << 30;

默认的加载因子(填充比),

**static final float *DEFAULT\_LOAD\_FACTOR*** = 0.75f;

当哈希表中的某个链表长度超过阀值8时，将链表转换为红黑树结构

**static final int *TREEIFY\_THRESHOLD*** = 8;

在哈希表扩容时，如果发现链表长度小于 6，则会由树重新退化为链表。

**static final int *UNTREEIFY\_THRESHOLD*** = 6;

在转变成树之前，还会有一次判断，只有键值对数量大于 64 才会发生转换。这是为了避免在哈希表建立初期，多个键值对恰好被放入了同一个链表中而导致不必要的转化

**static final int *MIN\_TREEIFY\_CAPACITY*** = 64;

加载因子的作用：

加载因子，也就是填充比，默认为0.75，它是容器是否进行扩容的一个标准。当哈希表的实际容量/总容量大于加载因子时，就会对哈希表扩容。那么为什么Hash Table需要扩容呢？这是因为填充比越大，说明存储空间的利用率越高，如果不进行扩容的话，链表就会越来越长，这样查找的效率就很低（即使JDK 1.8采用了红黑树的结构）。扩容之后，将原来链表数组的每一个链表分成奇偶两个子链表分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，提高了查找效率。HashMap本来就是以空间换时间数据结构，所以填充比没必要太大，但是填充比太小又会导致空间浪费严重。因此，要根据实际场景判断。如果关注内存，填充比可以稍大；如果更关注查询效率，填充比可以适当小点。

#### HashMap构造方法

HashMap提供了四种构造方法，可以指定数组的初始容量，填充比及初始化元素：

**public** HashMap()

**public** HashMap(**int** initialCapacity)

**public** HashMap(**int** initialCapacity, **float** loadFactor)

**public** HashMap(Map<? **extends** K, ? **extends** V> m)

第一种会构造一个默认容量，默认填充比的HashMap；第二种可以指定初始容量；第三种会设置初始容量和填充比；第四种会调用putMapEntries方法将Map中的元素添加到HashMap中。

#### 添加元素

Put方法用于HashMap添加键值对。它内部调用的是putVal方法。

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

putVal的大概流程：

1. 判断当前哈希表是否为空，如果为空，就创建一个默认大小的哈希表；
2. 根据key的hash值，获取要存入的数组下标。如果是链表的第一个元素，新建一个节点加入链表；
3. 如果不是，根据第一个节点的对象类型，判断当前是红黑树还是链表，分别插入新节点。

**public void** putAll(Map<? **extends** K, ? **extends** V> m) {  
 putMapEntries(m, **true**);  
}

批量添加元素。putMapEntries方法在添加元素时，会先判断哈希表的容量是否超过最大值，是否需要扩容，然后再循环调用putVal方法添加键值对。

#### 获取元素

get方法是根据传入的key获取对应的value。由代码可以看出，它内部调用了HashMap的getNode方法。

**public** V get(Object key) {  
 Node<K,V> e;  
 **return** (e = getNode(*hash*(key), key)) == **null** ? **null** : e.**value**;  
}

getNode方法会先根据key的哈希值判断应该去数组中的哪个链表（或红黑树）中查找元素。找到指定位置后，再判断链表或红黑树的第一个元素是不是我们要找的元素。如果不是，根据第一个元素的对象类型判断是链表还是红黑树，然后再根据key找对应的节点，最后返回对应的节点。

#### 移除元素

移除元素也需要一系列判断，判断是从链表还是红黑树中移除节点。

**public** V remove(Object key)

移除key对应的键值对。

#### 动态扩容

构造hash表时，如果不指明初始大小，默认大小为16（即Node数组大小16），如果Node[]数组中的元素达到（填充比\*Node.length）重新调整HashMap大小 变为原来2倍大小,扩容很耗时，因此，如果能够大概预估HashMap中要存的元素数量时，可以在创建HashMap的时候指定一个近似值作为初始容量。

**final** Node<K,V>[] resize() {  
 Node<K,V>[] oldTab = **table**;  
 **int** oldCap = (oldTab == **null**) ? 0 : oldTab.**length**;  
 **int** oldThr = **threshold**;  
 **int** newCap, newThr = 0;  
 **if** (oldCap > 0) {  
 **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* newCap = ***DEFAULT\_INITIAL\_CAPACITY***;  
 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;  
 @SuppressWarnings({**"rawtypes"**,**"unchecked"**})  
 Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];  
 **table** = newTab;  
 **if** (oldTab != **null**) {  
 **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;  
}

一个动态扩容，那么复杂。

### LinkedHashMap

LinkedHashMap继承了HashMap抽象类，底层采用的是数组+双向链表的实现方式，而HashMap采用的是数组+单向链表的实现方式。

LinkedHashMap中提供的方法并不多，且大部分还是在HashMap的方法基础上扩展了一下。所以，LinkedHashMap只是对HashMap的功能进行了扩展。

其中主要的扩展包括：

1. LinkedHashMap中的链表节点Entry继承了HashMap中的节点Entry，同时提供了节点对应的前后节点的索引。所以LinkedHashMap中的链表是双向链表，拥有双向链表的特性。
2. LinkedHashMap中提供了一个accessOrder的属性，它是boolean型的。当accessOrder为false时，LinkedHashMap的迭代顺序就是元素的插入顺序；当accessOrder为true时，LinkedHashMap的迭代顺序为元素的访问顺序。也就是通过get方法访问元素时，会调整元素顺序，将元素放到节链表的末尾位置。

#### 实现方式

**public class** LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>

LinkedHashMap继承了HashMap抽象类，所以HashMap支持的操作，LinkedHashMap都支持。

#### 私有属性

指定迭代的方式，如果为false，按照元素插入顺序迭代；为true，按照元素访问顺序迭代；

**final boolean accessOrder**;

LinkedHashMap中的最新节点，一般是尾节点

**transient** LinkedHashMap.Entry<K,V> **tail**;

LinkedHashMap中的头节点

**transient** LinkedHashMap.Entry<K,V> **head**;

LinkedHashMap继承了HashMap，它在此基础进行了扩展，提供的新的特性和方法并不是很多。

#### 构造方法

LinkedHashMap提供了5种构造方法，但前面四种构造方法和HashMap提供的构造方法相同，只是指定了accessOrder属性为false，也就是按照元素插入顺序迭代。

**public** LinkedHashMap()

**public** LinkedHashMap(**int** initialCapacity)

**public** LinkedHashMap(**int** initialCapacity, **float** loadFactor)

**public** LinkedHashMap(Map<? **extends** K, ? **extends** V> m)

**public** LinkedHashMap(**int** initialCapacity,**float** loadFactor, **boolean** accessOrder)

最后一种构造方法除了指定初始容量和加载因子外，还可以设置LinkedHashMap的迭代方式。

#### 操作元素

LinkedHashMap连添加元素的put方法都没有提供，我们在通过put添加元素时直接使用的是HashMap的put方法。Map中key是不可以重复的，如果向Map中put两个相同key的元素，第二个元素会覆盖第一个。也就是如果put(1, “A”);put(1, “B”);那么map中只有一个元素 1, “B”。

**public** V get(Object key) {  
 Node<K,V> e;  
 **if** ((e = getNode(*hash*(key), key)) == **null**)  
 **return null**;  
 **if** (**accessOrder**)  
 afterNodeAccess(e);  
 **return** e.**value**;  
}

LinkedHashMap的get方法实现代码如上：它首先调用HashMap的getNode方法，试图找到key对应的节点。如果key对应的节点为null，也就是元素不存在，直接返回。然后如果accessOrder的值为true，也就是按访问顺序迭代，就调用afterNodeAccess方法，将key对应的节点挪到链表的末尾。

**public** V getOrDefault(Object key, V defaultValue)

getOrDefault方法和get方法一样的，只不过在通过key查找节点的时候，如果节点不存在，不会返回null，而是返回方法参数中提供的默认值而已。

**public boolean** containsValue(Object value)

对LinkedHashMap从头到尾遍历，如果存在节点值和参数value相等，则返回true。

**public** Set<K> keySet() {  
 Set<K> ks = **keySet**;  
 **if** (ks == **null**) {  
 ks = **new** LinkedKeySet();  
 **keySet** = ks;  
 }  
 **return** ks;  
}

返回LinkedHashMap中的keySet属性，如果keySet为空，也就是LinkedHashMap中没有元素，则新建一个空的LinkedKeySet返回，LinkedKeySet继承了AbstractSet，是Set的一个子类。

**public** Collection<V> values() {  
 Collection<V> vs = **values**;  
 **if** (vs == **null**) {  
 vs = **new** LinkedValues();  
 **values** = vs;  
 }  
 **return** vs;  
}

返回LinkedHashMap中的所有值。类似的，LinkedValues继承了AbstractCollection，是集合Collection的一个子类。

**public** Set<Map.Entry<K,V>> entrySet() {  
 Set<Map.Entry<K,V>> es;  
 **return** (es = **entrySet**) == **null** ? (**entrySet** = **new** LinkedEntrySet()) : es;  
}

返回LinkedHashMap中的所有节点。

**public void** forEach(BiConsumer<? **super** K, ? **super** V> action)

遍历LinkedHashMap，forEach中实现accept方法即可。Accept方法中有两个参数，分别是节点的key和value。

map.forEach(**new** BiConsumer <Integer, String>() {  
 @Override  
 **public void** accept(Integer integer, String s) {  
 System.***out***.println(integer + s);  
 }  
});

### TreeMap

### IdentityMap

### WeakedHashMap

### HashTable

HashTable和HashMap很相似，执行效率也差不多。但它基本也不再使用，和Vector一样，主要是为了提供对之前版本的支持。

# IO

# NIO(New IO)

Java NIO(New IO)是一个可以替代标准Java IO API的IO API（从Java 1.4开始)，Java NIO提供了与标准IO不同的IO工作方式。现在使用NIO的场景越来越多，很多网上的技术框架或多或少的使用NIO技术，譬如Tomcat，Jetty。因此，了解NIO技术是很有必要的。

NIO有三大核心部分，分别是通道Channel、缓存Buffer和选择器Selector。

通道和缓冲区：

标准的IO是基于字节流和字符流进行操作的，而NIO是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

非阻塞IO：

Java NIO可以让你非阻塞的使用IO，例如：当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

选择器：

Java NIO引入了选择器的概念，选择器用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。

## 概述

通道Channel、缓存Buffer和选择器Selector是NIO技术的核心API，虽然除此之外，NIO还有很多其他工具类和组件，如Pipe和FileLock。

通道Channel和缓存区Buffer：

基本上，所有的 IO 在NIO 中都从一个通道开始的。通道有点象流。 数据可以从通道读到缓存中，也可以从缓存写到通道中。

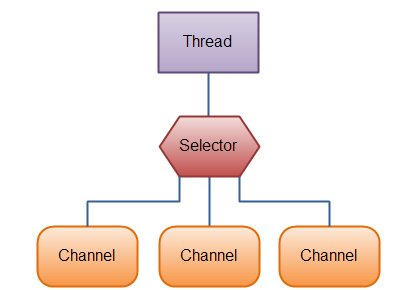
缓冲区Buffer

通道Channel

Channel和IO中的Stream(流)是差不多一个等级的。只不过Stream是单向的，譬如：InputStream, OutputStream.而Channel是双向的，既可以用来进行读操作，又可以用来进行写操作。

选择器Selector：

Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。

   
要使用Select，需要将通道Channel注册到Selector中。这样，Selector才能够和通道关联起来，实现对通道的监听。

## 通道(Channel)

NIO中的通道和IO中的流类似，但存在以下区别：

1. 流是单向的，而通道Channel是双向的，既可以读数据，又可以写数据；
2. 通道Channel读数据时，先将数据读到一个缓存区Buffer；写数据时，也是从Buffer中写入通道。
3. 通道可以异步的读写。

### 通道类型

NIO中的Channel的主要实现有：

1. FileChannel
2. DatagramChannel
3. SocketChannel
4. SocketChannel

这些通道涵盖了UDP 和 TCP 网络IO，以及文件IO。

### FileChannel

NIO中的FileChannel是一个连接到文件的通道。可以通过文件通道读写文件。FileChannel无法设置为非阻塞模式，它总是运行在阻塞模式下。

#### 打开FileChannel

在使用FileChannel之前，必须先打开它。但是，我们无法直接打开一个FileChannel，需要通过使用一个InputStream、OutputStream或RandomAccessFile来获取一个FileChannel实例。

通过InputStream/OutputStream方式：

FileInputStream fis = **new** FileInputStream(**"C:\\Users\\hand\\Desktop\\new.txt"**);  
FileChannel fileChannel = fis.getChannel();

FileOutputStream fos = **new** FileOutputStream(**"C:\\Users\\hand\\Desktop\\new.txt"**);  
FileChannel fileChannel = fos.getChannel();

通过RandomAccessFile方式：

RandomAccessFile aFile = **new** RandomAccessFile(**"data/nio-data.txt"**, **"rw"**);  
FileChannel inChannel = aFile.getChannel();

#### 从FileChannel读取数据

调用read()方法从FileChannel中读取数据。如：

ByteBuffer buf = ByteBuffer.*allocate*(48);  
**int** bytesRead = inChannel.read(buf);

调用FileChannel.read()方法。该方法将数据从FileChannel写入到Buffer中。read()方法返回的int值表示了有多少字节被读到了Buffer中。如果返回-1，表示到了文件末尾。

#### 向FileChannel写数据

使用FileChannel.write()方法向FileChannel写数据，该方法的参数是一个Buffer

String newData = **"New String to write to file..."** + System.*currentTimeMillis*();  
  
ByteBuffer buf = ByteBuffer.*allocate*(48);  
buf.clear();  
buf.put(newData.getBytes());  
  
buf.flip();  
  
**while**(buf.hasRemaining()) {  
 channel.write(buf);  
}

注意FileChannel.write()是在while循环中调用的。因为无法保证write()方法一次能向FileChannel写入多少字节，因此需要重复调用write()方法，直到Buffer中已经没有尚未写入通道的字节。

#### 关闭FileChannel

用完FileChannel后必须将其关闭。

channel.close();

#### 常用方法

##### read

read方法用于读取文件中的数据，并将数据写入到缓存中。read方法提供了多种实现方式：

**int** read(ByteBuffer dst)

将数据写入缓存，并返回总共写入了多少个元素。

**long** read(ByteBuffer[] dsts, **int** offset, **int** length)

将数据写入缓存数组，offset指定从数组中第几个缓存开始写入，length指定有几个缓存可以写入。在传递缓存数组作为参数时，必须前一个缓存区写满之后，才会写入下一个缓存区。

**long** read(ByteBuffer[] dsts)

将数据写入缓存数组。调用的是上面那个方法，offset设置为0，length为数组长度。

##### write

write方法用于从缓存中获取数据，写入通道中。和read方法一样，也提供了几种方式。

**int** write(ByteBuffer src)

**long** write(ByteBuffer[] srcs, **int** offset, **int** length)

**long** write(ByteBuffer[] srcs)

##### open

##### position

position方法用于获取缓存区中的当前position值或者设置缓存区的position属性。

**long** position()

FileChannel position(**long** newPosition)

##### truncate

可以用FileChannel.truncate()方法截取一个文件。截取文件时，文件中指定长度后面的部分将被删除。如：截取文件的前1024个字节。

channel.truncate(1024);

##### size

size()方法将返回该实例所关联文件的大小。

**long** fileSize = channel.size();

##### force

force()方法用于将通道里尚未写入磁盘的数据强制写到磁盘上。出于性能方面的考虑，使用通道读取数据时，操作系统会将数据先缓存在内存中，因此无法保证写入到FileChannel里的数据一定会即时写到磁盘上。要保证这一点，需要调用force()方法。   
  
force()方法有一个boolean类型的参数，指明是否同时将文件元数据（权限信息等）写到磁盘上。

channel.force(**true**);

##### transferFrom

用于将数据从源通道传输到FileChannel中，也就是将数据从一个通道写入一个FileChannel中。

##### transferTo

用于将数据从FileChannel传输到其他的channel中。

##### map

##### lock

##### tryLock

### SocketChannel

SocketChannel是一个连接到TCP网络套接字的通道。可以通过以下2种方式创建SocketChannel：

1. 打开一个SocketChannel并连接到互联网上的某台服务器
2. 当一个新连接到达ServerSocketChannel时，创建一个SocketChannel

打开SocketChannel

通过open方法打开一个SocketChannel，然后通过connect让它连接到某个网址的指定端口。

SocketChannel socketChannel = SocketChannel.open();  
socketChannel.connect(**new** InetSocketAddress(**"http://jenkov.com"**, 80));

当用完SocketChannel之后调用close()方法关闭SocketChannel，否则会造成资源浪费：

socketChannel.close();

读写数据

要从SocketChannel中读取数据，可以调用read方法，通道提供了多种形式的read方法。

ByteBuffer buf = ByteBuffer.*allocate*(48);  
**int** bytesRead = socketChannel.read(buf);

调用SocketChannel.read()。该方法将数据从SocketChannel 读到Buffer中。read()方法返回的int值表示读了多少字节进Buffer里。如果返回的是-1，表示已经读到了流的末尾（连接关闭了）。

写数据到SocketChannel用的是SocketChannel.write()方法，该方法以一个Buffer作为参数。

String newData = **"New String to write to file..."** + System.*currentTimeMillis*();  
  
ByteBuffer buf = ByteBuffer.*allocate*(48);  
buf.clear();  
buf.put(newData.getBytes());  
  
buf.flip();  
  
**while**(buf.hasRemaining()) {  
 channel.write(buf);  
}

注意SocketChannel.write()方法的调用是在一个while循环中的。Write()方法无法保证能写多少字节到SocketChannel。所以，我们需要重复调用write()直到Buffer没有要写的字节为止。

非阻塞模式

非阻塞模式与选择器搭配可以更好的工作，通过将一或多个SocketChannel注册到Selector，然后可以通过选择器哪个通道已经准备好了读取，写入等。

设置SocketChannel为非阻塞模式之后，就可以在异步模式下调用connect(), read() 和write()了。

Connect()

当SocketChannel在非阻塞模式下，调用connect()时，该方法可能在连接建立之前就返回了。为了确定是否已建立连接，可以调用finishConnect()的方法。

socketChannel.configureBlocking(false);  
socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));  
  
while(! socketChannel.finishConnect() ){  
//wait, or do something else...  
}

write()

在非阻塞模式下，write()方法可能在尚未写出任何内容时可能就返回了。所以需要在循环中调用write()。

Read()

非阻塞模式下,read()方法在尚未读取到任何数据时可能就返回了。所以需要关注它的int返回值，它会告诉你读取了多少字节。

### ServerSocketChannel

可以监听新进来的TCP连接，像Web服务器那样。对每一个新进来的连接都会创建一个SocketChannel。

ServerSocketChannel 是一个可以监听新进来的TCP连接的通道，就像标准IO中的ServerSocket一样。ServerSocketChannel类在 java.nio.channels包中。

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
  
serverSocketChannel.socket().bind(new InetSocketAddress(9999));  
  
while(true){  
SocketChannel socketChannel =  
serverSocketChannel.accept();  
  
//do something with socketChannel...  
}

打开、关闭

通过调用open() 方法可以打开ServerSocketChannel，调用close()方法关闭。

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.close();

监听连接：

通过 ServerSocketChannel.accept() 方法监听新进来的连接。当 accept()方法返回的时候，它返回一个包含新进来的连接的 SocketChannel。因此，accept()方法会一直阻塞到有新连接到达。

通常不会仅仅只监听一个连接，在while循环中调用 accept()方法

while(true){  
SocketChannel socketChannel =  
serverSocketChannel.accept();  
  
//do something with socketChannel...  
}

非阻塞模式

ServerSocketChannel可以设置成非阻塞模式。在非阻塞模式下，accept() 方法会立刻返回，此时，如果还没有新进来的连接，返回的将是null。 因此，需要检查返回的SocketChannel是否是null。

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
  
serverSocketChannel.socket().bind(new InetSocketAddress(9999));  
serverSocketChannel.configureBlocking(false);  
  
while(true){  
SocketChannel socketChannel =  
serverSocketChannel.accept();  
  
if(socketChannel != null){  
//do something with socketChannel...  
}  
}

### DatagramChannel

DatagramChannel是一个能收发UDP包的通道。因为UDP是无连接的网络协议，所以不能像其它通道那样读取和写入。它发送和接收的是数据包。

#### 打开DatagramChannel

DatagramChannel提供了一个open方法用于获取一个实例，并且可以指定通道监听的端口，由于UDP是无连接的，连接到特定地址并不会像TCP通道那样创建一个真正的连接。而是锁住DatagramChannel ，让其只能从特定地址收发数据

当UD连接该端口并发送数据时，通道才能获取数据包。

DatagramChannel channel = DatagramChannel.open();  
channel.socket().bind(**new** InetSocketAddress(9999));

这段代码中 DatagramChannel可以在UDP端口9999上接收数据包。

当连接指定端口后，也可以使用read()和write()方法，就像在用传统的通道一样。只是在数据传送方面没有任何保证。

**int** bytesRead = channel.read(buf);  
**int** bytesWritten = channel.write(but);

#### 收发数据

通过receive()方法从DatagramChannel接收数据，如：

ByteBuffer buf = ByteBuffer.*allocate*(48);  
buf.clear();  
channel.receive(buf);

 receive()方法会将接收到的数据包内容复制到指定的Buffer. 如果Buffer容不下收到的数据，多出的数据将被丢弃。

通过send()方法从DatagramChannel发送数据，如:

String newData = **"New String to write to file..."** + System.*currentTimeMillis*();  
  
ByteBuffer buf = ByteBuffer.*allocate*(48);  
buf.clear();  
buf.put(newData.getBytes());  
buf.flip();  
  
**int** bytesSent = channel.send(buf, **new** InetSocketAddress(**"jenkov.com"**, 80));

上面的代码用于发送一串字符到”jenkov.com”服务器的UDP端口80。 因为服务端并没有监控这个端口，所以什么也不会发生。也不会通知你发出的数据包是否已收到，因为UDP在数据传送方面没有任何保证。

### 通道之间数据的传输

在Java NIO中，如果两个通道中有一个是FileChannel，那你可以直接将数据从一个channel传输到另外一个channel。

transferFrom()方法用于将数据从源通道传输到FileChannel中，也就是将数据从一个通道写入一个FileChannel中。

RandomAccessFile fileFrom = **new** RandomAccessFile(**"C:\\Users\\hand\\Desktop\\文档.txt"**, **"rw"**);  
FileChannel inChannel = fileFrom.getChannel();  
RandomAccessFile fileTo = **new** RandomAccessFile(**"C:\\Users\\hand\\Desktop\\new.txt"**, **"rw"**);  
FileChannel toChannel = fileTo.getChannel();  
//toChannel调用transferFrom方法，也就是它的数据来源于inChannel  
**long** position = 0l;  
**long** count = inChannel.size();  
toChannel.transferFrom(inChannel, position, count);

方法的输入参数position表示从position处开始向目标文件写入数据，count表示最多传输的字节数。如果源通道的剩余空间小于 count 个字节，则所传输的字节数要小于请求的字节数。

此外要注意，在SoketChannel的实现中，SocketChannel只会传输此刻准备好的数据（可能不足count字节）。因此，SocketChannel可能不会将请求的所有数据(count个字节)全部传输到FileChannel中。

transferTo()方法将数据从FileChannel传输到其他的channel中。

RandomAccessFile fileFrom = **new** RandomAccessFile(**"C:\\Users\\hand\\Desktop\\new.txt"**, **"rw"**);  
FileChannel inChannel = fileFrom.getChannel();  
RandomAccessFile fileTo = **new** RandomAccessFile(**"C:\\Users\\hand\\Desktop\\新建文本文档.txt"**, **"rw"**);  
FileChannel toChannel = fileTo.getChannel();  
// inChannel中的数据会被写入到toChannel中。  
**long** position = 0l;  
**long** count = inChannel.size();  
inChannel.transferTo(position, count, toChannel);

## 缓存(Buffer)

Buffer用于和NIO通道进行交互。正如前面介绍的，数据是从通道读入缓冲区，从缓冲区写入到通道中的。

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成NIO Buffer对象，并提供了一组方法，用来方便的访问该块内存。

缓存区Buffer核心属性：

1. capacity capacity代表缓存区的容量大小，也就是可以存储多少个byte、long或者char类型的字符。不管是读模式还是写模式，capacity代表的含义不变。
2. position 将数据写入Buffer中时，position代表缓存区中当前存储元素数。类似与一个游标，当向缓存中写入一个数据时，position的值加1，直到为capacity-1，也就是缓存满了；当将Buffer从写模式切换到读模式，position会被重置为0。当从Buffer的position处读取数据时，position向前移动到下一个可读的位置。
3. limit 在写模式下，Buffer的limit表示你最多能往Buffer里写多少数据。 写模式下，limit等于Buffer的capacit；当切换Buffer到读模式时， limit表示你最多能读到多少数据，limit会被设置成写模式下的position值，也就是现在缓存中存储了多少数据，

Buffer读写数据步骤：

1. 将数据写入Buffer中
2. 调用flip()方法切换Buffer模式，从写模式切换到读或从读模式切换到写
3. 从Buffer中读取数据
4. 调用clear()或者compact()方法

当向buffer写入数据时，buffer会记录下写了多少数据。当要读取数据，需要通过flip()方法将Buffer从写模式切换到读模式。在读模式下，可以读取之前写入到buffer中的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用clear()或compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。未读的数据会被移到缓冲区的起始处，新写入的数据将被放到缓冲区未读数据的后面。

### 缓存类型

NIO里关键的Buffer实现：

1. ByteBuffer
2. CharBuffer
3. DoubleBuffer
4. FloatBuffer
5. IntBuffer
6. LongBuffer
7. ShortBuffer

这些Buffer覆盖了你能通过IO发送的基本数据类型：byte, short, int, long, float, double 和 char。除此之外，NIO还有个Mappedyteuffer，用于表示内存映射文件

### Buffer常用方法

allocate()

我们知道，基于通道的文件数据读写总是离不开缓存，而allocate方法就是用来分配缓存大小的，也就是指定缓存区的capacity属性。

ByteBuffer buf = ByteBuffer.*allocate*(48);  
CharBuffer cuf = CharBuffer.*allocate*(48);

Buffer有多种类型，它们继承了Buffer抽象类，也继承了它的allocate方法。ByteBuffer.allocate(48)创建一个容量为48字节的缓存区；CharBuffer.*allocate*(48)用于创建一个容量为48字符的缓存区。

flip()

flip方法将Buffer从写模式切换到读模式。调用flip()方法会将position设回0，并将limit设置成之前position的值。 换句话说，position现在用于标记读的位置，limit表示之前写进了多少个byte、char等 —— 现在能读取多少个byte、char等。

put()

put方法用于向缓存中写入数据，Buffer提供了多种形式的put方法。比如：

ByteBuffer put(ByteBuffer src)

将一个缓存中的数据写入到另一个缓存区。

ByteBuffer put(**byte**[] src)

将一个字节数组中的数据写入到一个缓存区中。

ByteBuffer put(**byte**[] src, **int** offset, **int** length)

将一个字节数组中下标从offset到offset+length的元素写入缓存区中。

ByteBuffer put(**int** index, **byte** b)

将一个字节写入到缓存区中的指定位置。

get()

get方法用于从Buffer中读取数据。每次读取一个元素。

**char c** = (**char**) buf.get();

rewind()

Buffer rewind()

Buffer.rewind()将position设回0，所以你可以重读Buffer中的所有数据。limit保持不变，仍然表示能从Buffer中读取多少个元素（byte、char等）。

clear()与compact()方法

一旦读完Buffer中的数据，需要让Buffer准备好再次被写入。可以通过clear()或compact()方法来完成。

Buffer clear()

如果调用的是clear()方法，position将被设回0，limit被设置成 capacity的值。换句话说，Buffer 被清空了。Buffer中的数据并未清除，只是这些标记告诉我们可以从哪里开始往Buffer里写数据。

如果Buffer中有一些未读的数据，调用clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

ByteBuffer compact()

如果Buffer中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用compact()方法是个不错的选择。

compact()方法将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读元素正后面。limit属性依然像clear()方法一样，设置成capacity。现在Buffer准备好写数据了，但是不会覆盖未读的数据。

mark()与reset()方法

通过调用Buffer.mark()方法，可以标记Buffer中的一个特定position。之后可以通过调用Buffer.reset()方法恢复到这个position。例如：

buf.mark();  
buf.reset();

mark方法中不能传参，它只会标记调用mark方法时position的值。

equals()与compareTo()方法

equals()与compareTo()方法都用来比较两个Buffer。但equals方法只比较两个Buffer中剩余部分是否相等；compareTo方法比较两个Buffer时，会对两个Buffer中的剩余的每一个元素进行比较，当遇到第一个不等的元素时，如果前一个小于后一个，返回-1，否则返回1.。如果所有元素都相等，但第一个Buffer比另一个先耗尽(第一个Buffer的元素个数比另一个少)。

### 示例

从文件中读写数据。

RandomAccessFile aFile = **new** RandomAccessFile(**"C:\\Users\\hand\\Desktop\\新建文本文档.txt"**, **"rw"**);  
FileChannel inChannel = aFile.getChannel();  
ByteBuffer buf = ByteBuffer.*allocate*(48);  
**int** bytesRead = inChannel.read(buf);  
**while** (bytesRead != -1) {  
 System.***out***.println(**"Read "** + bytesRead);  
 buf.flip();  
 **while**(buf.hasRemaining()){  
 System.***out***.print(**new** String(String.*valueOf*((**char**) buf.get()).getBytes(**"ISO-8859-1"**), **"utf-8"**) );  
 }  
 buf.clear();  
 bytesRead = inChannel.read(buf);  
}  
aFile.close();

## 选择器(Selector)

选择器(Selector)是NIO中能够检测多个NIO通道并且判断通道是否为读写事件做好准备的组件。这样，一个单独的线程可以管理多个channel，从而管理多个网络连接。、

理论上，可以只用一个线程处理所有的通道，以减少线程之间上下文切换的开销。但实际上，现在电脑都有多个内核，本身就支持多线程处理，使用单线程反而会浪费多内核配置，而且随着硬件的提升，线程间切换的开销越来越小，因此选择开启合适数量的线程提高代码效率非常重要。

### Selector的创建

通过调用Selector.open()方法创建一个Selector。

Selector selector = Selector.*open*();

### 向Selector注册通道

为了将Channel和Selector配合使用，必须将channel注册到selector上。可以通过SelectableChannel.register()方法来实现：

channel.configureBlocking(**false**);  
SelectionKey key = channel.register(selector,  
 Selectionkey.OP\_READ);

与Selector一起使用时，Channel必须处于非阻塞模式下。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

注意register()方法的第二个参数。这是一个“interest集合”，意思是在通过Selector监听Channel时对什么事件感兴趣。可以监听四种不同类型的事件及它们的常量表示方式：

1. Connect SelectionKey.OP\_CONNECT
2. Accept SelectionKey.OP\_ACCEPT
3. Read SelectionKey.OP\_READ
4. Write SelectionKey.OP\_WRITE

通道触发了一个事件意思是该事件已经就绪。所以，某个channel成功连接到另一个服务器称为“连接就绪”。一个server socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。

如果选择器对通道的不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来。

**int** interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;

### SelectionKey

当向Selector注册Channel时，register()方法会返回一个SelectionKey对象。这个对象包含了一些重要的属性：

1. interest集合
2. ready集合
3. Channel
4. Selector
5. 附加的对象（可选）

Interest集合就是选择器注册通道时，选择的感兴趣的事件集合。

**int** interestSet = selectionKey.interestOps();  
  
**boolean** isInterestedInAccept = (interestSet & SelectionKey.OP\_ACCEPT) == SelectionKey.OP\_ACCEPT；  
**boolean** isInterestedInConnect = interestSet & SelectionKey.OP\_CONNECT;  
**boolean** isInterestedInRead = interestSet & SelectionKey.OP\_READ;  
**boolean** isInterestedInWrite = interestSet & SelectionKey.OP\_WRITE;

用“位与”操作interest 集合和给定的SelectionKey常量，可以确定某个确定的事件是否在interest 集合中。

ready集合是通道已经准备就绪的操作的集合。在一次选择(Selection)之后，你会首先访问这个ready set。然后可以像检测interest集合那样的方法，来检测channel中什么事件或操作已经就绪。或通过下面几个方法：

**int** readySet = selectionKey.readyOps();  
  
selectionKey.isAcceptable();  
selectionKey.isConnectable();  
selectionKey.isReadable();  
selectionKey.isWritable();

从SelectionKey访问Channel和Selector很简单

Channel channel = selectionKey.channel();  
Selector selector = selectionKey.selector();

附加对象：

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。例如，可以附加与通道一起使用的Buffer，或是包含聚集数据的某个对象

selectionKey.attach(theObject);  
Object attachedObj = selectionKey.attachment();

还可以在用register()方法向Selector注册Channel的时候附加对象

SelectionKey key = channel.register(selector, SelectionKey.OP\_READ, theObject);

### 通过Selector选择通道

当向选择器Selector中注册过一个或者多个通道后，就可以使用选择器提供的select()方法了。Select方法会返回存在事件已经准备就绪的通道（注册通道时，会指定选择器对该通道感兴趣的事件，当这些被监听的通道关联的选择器感兴趣事件准备就绪时，该通道才会被select返回）。比如：将通道A注册到选择器1，其中关注的事件是读事件，那么当A的读操作准备好了后，调用select方法就会返回通道A。

Select()方法调用后，如果没有满足条件的通道，则会一直阻塞。直到有一个通道中的选择器关注的事件已就绪。

select(long timeout)和select()一样，只是会传递一个阻塞时间，如果到了超时时间，仍没有通道满足条件，则放弃等待。

selectNow()不会阻塞，不管什么通道就绪都立刻返回，如果没有满足条件的通道则返回0.

select()方法返回的int值表示有多少通道已经就绪。注意：它是自上次调用select()方法后又有多少通道变成就绪状态。比如：第一次调用select方法时，有一个通道准备就绪，返回1；之后又有一个通道准备就绪，此时再调用select方法，虽然有两个通道就绪了，但返回值仍旧是1

### SelectedKeys

调用select()方法，并且返回值大于0时，就可以通过调用selector的selectedKeys()方法，访问“已选择键集（selected key set）”中的就绪通道。

Set selectedKeys = selector.selectedKeys();

当向Selector注册Channel时，Channel.register()方法会返回一个SelectionKey 对象。这个对象代表了注册到该Selector的通道。可以通过SelectionKey的selectedKeySet()方法可以访问这些对象。

可以遍历这个已选择的键集合来访问就绪的通道。

Set selectedKeys = selector.selectedKeys();  
Iterator keyIterator = selectedKeys.iterator();  
**while**(keyIterator.hasNext()) {  
 SelectionKey key = keyIterator.next();  
 **if**(key.isAcceptable()) {  
 *// a connection was accepted by a ServerSocketChannel.* } **else if** (key.isConnectable()) {  
 *// a connection was established with a remote server.* } **else if** (key.isReadable()) {  
 *// a channel is ready for reading* } **else if** (key.isWritable()) {  
 *// a channel is ready for writing* }  
 keyIterator.<tuihighlight **class**=**"tuihighlight"**><a href=**"javascript:;"** style=**"display:inline;float:none;position:inherit;cursor:pointer;color:#7962D5;text-decoration:underline;"** onclick=**"return false;"**>remove</a></tuihighlight>();  
}

这个循环遍历已选择键集中的每个键，并检测各个键所对应的通道的就绪事件。   
  
注意每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除。下次该通道变成就绪时，Selector会再次将其放入已选择键集中。   
  
SelectionKey.channel()方法返回的通道需要转型成你要处理的类型，如ServerSocketChannel或SocketChannel等。

### 其他方法

wakeUp

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，也有办法让其从select()方法返回。只要让其它线程在第一个线程调用select()方法的那个对象上调用Selector.wakeup()方法即可。阻塞在select()方法上的线程会立马返回。

如果有其它线程调用了wakeup()方法，但当前没有线程阻塞在select()方法上，下个调用select()方法的线程会立即“醒来（wake up）”。

Close

用完Selector后调用其close()方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey实例无效。通道本身并不会关闭。

### 示例

Selector **selector** = Selector.*open*();   
channel.configureBlocking(**false**);  
 SelectionKey key = channel.register(selector, SelectionKey.OP\_READ);  
 **while**(**true**) {  
 **int** readyChannels = selector.select();  
 **if**(readyChannels == 0) **continue**;  
 Set selectedKeys = selector.selectedKeys();  
 Iterator keyIterator = selectedKeys.iterator();  
 **while**(keyIterator.hasNext()) {  
 SelectionKey key = keyIterator.next();  
 **if**(key.isAcceptable()) {  
 *// a connection was accepted by a ServerSocketChannel.* } **else if** (key.isConnectable()) {  
 *// a connection was established with a remote server.* } **else if** (key.isReadable()) {  
 *// a channel is ready for reading* } **else if** (key.isWritable()) {  
 *// a channel is ready for writing* }  
 keyIterator.<tuihighlight **class**=**"tuihighlight"**><a href=**"javascript:;"** style=**"display:inline;float:none;position:inherit;cursor:pointer;color:#7962D5;text-decoration:underline;"** onclick=**"return false;"**>remove</a></tuihighlight>();  
 }  
 }

## 分散(Scatter)/聚集(Gather)

Java NIO开始支持分散/聚集，聚集/分散操作，用于描述从Channel向缓存中写入数据或者将缓存中的数据读取到Channel的操作。

scatter / gather操作经常用于需要将传输的数据分开处理的场合，例如传输一个由消息头和消息体组成的消息，你可能会将消息体和消息头分散到不同的buffer中，这样你可以方便的处理消息头和消息体。

分散（scatter）指的是在读操作时将读取的数据写入多个buffer中。也就是将从Channel中读取的数据“分散（scatter）”到多个Buffer中。

ByteBuffer **header** = ByteBuffer.*allocate*(128);  
ByteBuffer **body** = ByteBuffer.*allocate*(1024);  
  
ByteBuffer[] **bufferArray** = { **header**, **body** };  
channel.read(bufferArray);

注意buffer首先被插入到数组，然后再将数组作为channel.read() 的输入参数。read()方法按照buffer在数组中的顺序将从channel中读取的数据写入到buffer，当一个buffer被写满后，channel紧接着向另一个buffer中写。

Scattering Reads在移动下一个buffer前，必须填满当前的buffer，这也意味着它不适用于动态消息(即消息大小不固定)。换句话说，如果存在消息头和消息体，消息头必须完成填充（例如 128byte），Scattering Reads才能正常工作。

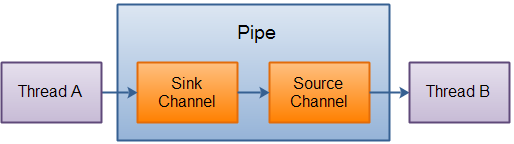
聚集（gather）是指在写操作时将多个buffer的数据写入同一个Channel。也就是将多个Buffer中的数据“聚集（gather）”后发送到Channel。

ByteBuffer **header** = ByteBuffer.*allocate*(128);  
ByteBuffer **body** = ByteBuffer.*allocate*(1024);  
  
ByteBuffer[] **bufferArray** = { **header**, **body** };  
channel.write(bufferArray);

buffers数组是write()方法的入参，write()方法会按照buffer在数组中的顺序，将数据写入到channel，注意只有position和limit之间的数据才会被写入。因此，如果一个buffer的容量为128byte，但是仅仅包含58byte的数据，那么这58byte的数据将被写入到channel中。因此与Scattering Reads相反，Gathering Writes能较好的处理动态消息。

## 管道(Pipe)

管道是NIO中两个线程之间的单向数据连接。Pipe中有一个Source通道，一个Sink通道，数据从Source通道流向Sink通道。



管道中提供了三个方法open()、source()、sink()和两个内部类SourceChannel和SinkChannel，其中open方法用于打开一个通道；source方法用于获取Source通道；sink方法用于获取Sink通道。

### 创建管道

通过Pipe.open()方法打开管道。

Pipe pipe = Pipe.*open*();

### 向管道写入数据

要向管道写数据，需要访问sink通道。

Pipe.SinkChannel sinkChannel = pipe.sink();

通过调用SinkChannel的write()方法，将数据写入SinkChannel

String newData = **"New String to write to file..."** + System.*currentTimeMillis*();  
ByteBuffer buf = ByteBuffer.*allocate*(48);  
buf.clear();  
buf.put(newData.getBytes());  
  
buf.flip();  
  
**while**(buf.hasRemaining()) {  
 sinkChannel.write(buf);  
}

### 从管道读取数据

从读取管道的数据，需要访问source通道。

Pipe.SourceChannel sourceChannel = pipe.source();

调用source通道的read()方法来读取数据。

ByteBuffer buf = ByteBuffer.*allocate*(48);  
**int** bytesRead = inChannel.read(buf);

## NIO与IO的区别

NIO是同步非阻塞的文件读取方式，IO是阻塞的文件读取方式，它们之间的主要区别是：

1. IO是面向流的，NIO是面向缓冲区的。IO每次从流中读取一个或多个字节，直至读取所有内容，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要移动从流中读取的数据，需要先将它缓存到一个缓冲区。而NIO的缓冲读取会先将数据读取到一个它稍后处理的缓冲区中，如果需要对数据进行处理，可以在缓冲区中移动数据，增加了文件数据处理的灵活性。但是，在处理数据时，需要检查该缓冲区中是否包含你需要处理的数据，以及在数据读入缓冲区时，需要注意新读入的数据是否会覆盖缓冲区里尚未处理的数据。
2. IO中的各种流是阻塞的。也就是当一个线程调用read或write方法读取/写入文件时，该线程将会被阻塞，直到文件数据被读写完成，而在此期间，该线程不能做其他事情。而NIO是非阻塞的，当一个线程从通道Channel中读取数据时，当它读取了部分数据而当前没有数据可用时，它不会保持线程阻塞，而是会去做其他事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。
3. NIO提供了选择器(Selector)的概念。一个线程可以实现对多个通道的监听进而管理多个通道。适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器。

NIO是非阻塞的，且可以使用一个线程管理多个管道，但付出的代价是解析数据比阻塞流(IO)解析数据复杂了很多。

在IO设计中，我们从InputStream或 Reader逐字节读取数据。假设你正在处理一基于行的文本数据流，例如：

Name: Anna  
Age: 25  
Email: anna@mailserver.com

使用IO中的readLine方法每次读取文本中的一行数据：

InputStream **input** = … ; *// get the InputStream from the client socket*BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(input));  
  
String nameLine = reader.readLine();  
String ageLine = reader.readLine();  
String emailLine = reader.readLine();

由于IO是阻塞的，所以当readLine方法返回时，我们就能确定文本行数据已经读取完。比如上面代码中当第一个readLine方法返回时，我们知道Name和它对应的值已经读写完成；当第二个readLine方法返回时，Age和它对应的值已经读写完成。

而NIO的实现则与之不同：

ByteBuffer **buffer** = ByteBuffer.allocate(48);  
  
**int bytesRead** = inChannel.read(**buffer**);

由于NIO是非阻塞的，因此当read方法返回时，你不知道它读取了多少数据，也就是你不知道缓存区里面的数据是否包含了你要处理的所有数据。例如，第一次read方法返回时只读取了半行数据“Name: A”，这时如果你想处理数据的话就显得很困难。

那么我们如何知道缓存区中是否包含我们要处理的数据呢？这种情况下，我们只能不停地查看缓存区。这就导致在我们知道缓存中数据可以处理之前，我们需要多次检查缓存区数据。严重影响了代码效率和质量。

ByteBuffer **buffer** = ByteBuffer.allocate(48);  
**int bytesRead** = inChannel.read(**buffer**);   
**while**(! bufferFull(bytesRead) ) {  
 bytesRead = inChannel.read(buffer);  
}

bufferFull方法会扫描缓存区，并根据缓存区是否已满返回真或假。

实际上的处理过程是：线程从管道中读取数据，然后管道中的数据存到缓存中，与此同时，线程还要不停的检查缓存中的数据。

总结：

1. 如果要管理成千上万个连接，并且这些连接每次只是发送少量的数据，例如聊天服务器。使用NIO的服务器是一个不错的选择。
2. 如果只有少量的连接，但使用非常高的带宽，一次发送大量的数据，典型的IO服务器也许更加契合。

# AIO

# 操作Excel

<http://www.cnblogs.com/huajiezh/p/5467821.html>

<http://so.csdn.net/so/search/s.do?q=excel&t=blog>

Java操作Excel文件有两种常用的方式：jxl和poi。

poi和jxl的区别：

网上有人做过测试，当数据量较小时（<30000），jxl的处理速度由于poi方式；当数据量较大时，poi由于jxl。但是jxl只能处理xls的文件，现已停止更新，而poi方式可以处理xls跟xlsx文件。

综上，使用Java从操作Excel时选择poi方式更好。

## POI介绍

POI是Apache软件基金会为Java操作Microsoft Office提供支持的开放API。它是一个开源项目。通过它，Java开发人员可以实现对Office文件的新建、读写等操作。

### POI包结构

1. HSSF 提供读写Microsoft Excel XLS格式档案的功能。
2. XSSF 提供读写Microsoft Excel OOXML XLSX格式档案的功能。
3. HWPF 提供读写Microsoft Word DOC格式档案的功能。
4. HSLF 提供读写Microsoft PowerPoint格式档案的功能。
5. HDGF 提供读Microsoft Visio格式档案的功能。
6. HPBF 提供读Microsoft Publisher格式档案的功能。
7. HSMF 提供读Microsoft Outlook格式档案的功能。

其中，操作Excel需要HSSF和XSSF模块。

## HSSF模块

HSSF模块主要用于操作.xls格式的文档。它里面包括了xls文档的各种结构：

1、HSSFWorkbook excel文档对象

2、HSSFSheet excel的sheet

3、HSSFRow excel的行

4、HSSFCell excel的单元格

5、HSSFFont excel字体

6、HSSFName 名称

7、HSSFDataFormat 日期格式

8、HSSFHeader sheet 头

9、HSSFFooter sheet 尾

10、HSSFCellStyle cell样式

11、HSSFDateUtil 日期

12、HSSFPrintSetup 打印

13、HSSFErrorConstants 错误信息表

### HSSFWorkbook

HSSFWorkbook用于创建一个Excel文本对象，同时能够设置Excel的一些文本级属性。比如设置Excel中的sheet属性，设置文档包含几个sheet，那个sheet为显示状态等。

HSSFWorkbook提供的方法：

1. 构造

#### 构造方法

HSSFWorkbook提供了多个构造方法用于创建一个Excel文档。这里介绍几种常用的构造方式。

使用构造方法创建一个HSSFWorkbook实例后，还需要通过HSSFWorkbook提供的write方法将创建的Excel文档输出到本地。此时，write方法接收一个输出流参数，可以在这里指定文档的生成位置。

1. 创建一个空的Excel文档

@Test  
**public void** testExcel() **throws** IOException{  
 HSSFWorkbook workbook = **new** HSSFWorkbook();  
 OutputStream os = **null**;  
 **try** {  
 os = **new** FileOutputStream(**"C:\\Users\\hand\\Desktop\\samle.xls"**);  
 workbook.write(os);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*结束\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **finally** {  
 os.close();  
 }  
}

1. POIFSFileSystem类是专门用来读取excel表格的，它可以把Excel文件作为数据流来进行传入传出。

# 注解

注解是JDK 1.5及以后版本引入的一个特性。与类、接口、枚举在同一层次。

注解也叫元数据，用于在代码的源文件、编译文件和运行时提供一些注释、说明。我们可以根据这些注释对代码进行相应的处理。

## 元注解

元注解的主要作用就是注解其他注解。

也就是在定义注解的时候，可以再使用元注解进行注解，指定自定义注解的作用对象等属性。

Java5提供了四个标准的元注解：

1、@Target

2、@Retention

3、@Documented

4、@Inherited

### @Target

@Target注解用于设置注解可修饰对象。也就是可以在哪个地方使用该注解。

@Target可以设置值：

1. @Target(ElementType.CONSTRUCTOR) 用于修饰构造器，也就是注解只能在构造器方法上声明
2. @Target(ElementType.METHOD) 用于修饰方法
3. @Target(ElementType.FIELD) 用于修饰字段、枚举常量
4. @Target(ElementType.PARAMETER) 用于修饰方法中参数
5. @Target(ElementType.PACKAGE) 用于修饰包
6. @Target(ElementType.LOCAL\_VARIABLE) 用于修饰局部变量
7. @Target(ElementType.TYPE) 用于修饰类、接口、注解、枚举
8. @Target(ElementType.ANNOTATION\_TYPE) 用于修饰注解7

@Documented  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***ANNOTATION\_TYPE***)  
**public** @**interface** Target {  
 *//可以设置多个值，格式为*{ElementType.***METHOD***, ElementType.***TYPE***}ElementType[] value();  
}

### @Retention

@Retention用于定义注解的保留策略，也就是注解存在于代码的哪个阶段，比如源码阶段、编译阶段、运行阶段。（注解的生命周期）

@Retention注解提供了三种保留策略：

1、@Retention(RetentionPolicy.SOURCE) 注解仅存在于源码中，在class字节码文件中不包含

2、@Retention(RetentionPolicy.CLASS) 默认的保留策略，注解会在class字节码文件中存在，但运行时无法获得

3、@Retention(RetentionPolicy.RUNTIME) 注解在源码和class字节码文件中都存在，在运行时可以通过反射获取到

通过上面介绍可知，三种保留策略的声明周期长度：SOURCE < CLASS < RUNTIME

前者能起作用的地方，后者也能起作用。一般如果我们需要在运行时解析注解，则必须用RUNTIME类型；如果要在编译时进行一些预处理操作，比如生成一些辅助代码（如 ButterKnife），就用 CLASS注解；如果只是做一些检查性的操作，比如 @Override 和 @SuppressWarnings，则可选用 SOURCE 注解。

### @Documented

@Document是一个没有成员的标记注解。被它修饰的注解可以被Javadoc等工具文档化，也就是生成API。

### @Inherited

@Inherited 元注解是一个标记注解。设置注解是否可继承。

一个注解如果使用了@Inherited修饰，则这个注解是可继承的。那么当这个注解修饰一个类时，这个类的子类也会自动继承该注解。

@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
@Documented  
@Inherited  
**public** @**interface** MyAnnocation {  
 String funName(); *// 如果没有设置默认值，则使用该注解时必须设置该成员值* String methodName() **default "METHOD"**;  
 String className() **default "CLASS"**;  
 **boolean** isSuit() **default true**;  
}

使用注解：

@MyAnnocation(funName = **"printVal"**)  
**public void** printVal(String val){  
 System.***out***.println(**"hello world"**+val);  
}

## 内置注解

JDK 自身提供的注解称为内置注解。、

JDK 5.0提供了三个常用注解：

1、@Deprecated 废弃，表示被修饰的资源已经或即将废弃，不建议使用这些资源

2、@Override 覆盖，主要用于子类继承或实现接口后，重写方法时使用

3、@SuppressWarnings 取消警告，忽略某些代码产生的特殊警告

### @Deprecated

是一个标注注解。

当我们想要告诉编译器一个方法或字段已经被弃用时，可以使用该注解修饰。这样当其他人引用该注解修饰的资源时会发出警告。

@Deprecated 注解一般和javadoc配合使用。在javadoc中告诉其他人这个弃用方法的替代。也就是哪个新的方法可以实现它的功能及如何使用。

@Documented  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(value={***CONSTRUCTOR***, ***FIELD***, ***LOCAL\_VARIABLE***, ***METHOD***, ***PACKAGE***, ***PARAMETER***, ***TYPE***})  
**public** @**interface** Deprecated {  
}

### @Override

是一个标注注解。

当我们重写父类方法时，可以使用该注解告诉编译器我们正在覆盖父类的一个方法。这样当父类的这个方法被删除或修改时，编译器就会提示错误信息。

@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***SOURCE***)  
**public** @**interface** Override {  
}

### @SuppressWarnings

这个注解仅仅是告知编译器，忽略它们产生了特殊警告，比如：在java泛型中使用原始类型。它的保持性策略(retention policy)是SOURCE，在编译器中将被丢弃。

@Target({***TYPE***, ***FIELD***, ***METHOD***, ***PARAMETER***, ***CONSTRUCTOR***, ***LOCAL\_VARIABLE***})  
@Retention(RetentionPolicy.***SOURCE***)  
**public** @**interface** SuppressWarnings {

String[] value();

}

## 自定义注解

创建自定义注解与编写接口比较相似，只是必须使用@interface关键字声明注解。

在定义注解时，不能继承其他的注解或接口。@interface用来声明一个注解，注解中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）。可以通过default来声明参数的默认值。

@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
@Documented  
@Inherited  
**public** @**interface** MyAnnocation {  
  
 String funName(); *// 如果没有设置默认值，则使用该注解时必须设置该成员值* String methodName() **default "METHOD"**;  
 String className() **default "CLASS"**;  
 **boolean** isSuit() **default true**;  
  
}

注意：

1. 自定义注解必须通过@interface声明
2. 自定义注解中对的每一个方法都类似于对象的一个参数，方法名是参数名，返回类型是参数值类型。
3. 自定义注解中的方法可以通过 default 关键字设置默认值。如果没有设置默认值，在使用该注解时，必须为注解的该方法设值。
4. 自定义注解中方法的返回值必须是基本类型：

4.1、所有基本数据类型（int,float,boolean,byte,double,char,long,short)

4.2、String类型

4.3、Class类型

4.4、enum类型

4.5、以上所有类型的数组

## 通过映射解析注解

解析注解：即通过反射获取类、函数或成员上的运行时注解信息，从而实现动态控制程序运行的逻辑。

注解解析首先针对的是运行时注解，通过元注解@Retention控制。其次，运行时注解是可以控制程序的执行逻辑的。比如Spring中的@RequestMapping注解，就可以根据url选择不同的方法处理请求。

Java中的Class.java文件中只提供了isAnnotation方法用于判断当前类是不是注解，但Class实现了AnnotatedElement接口，这个接口中提供了大量与注解相关的方法，这为通过反射解析运行时注解提供了基础。

**public final class** Class<T> **implements** java.io.Serializable,  
 GenericDeclaration,  
 Type,  
 AnnotatedElement {

**public boolean** isAnnotation()

}

### AnnotatedElement接口

AnnotatedElement表示正在运行的程序中的一个注释元素。当一个类被运行时注解修饰时，就可以通过该接口提供的方法解析运行时注解。

方法介绍：

#### isAnnotationPresent

isAnnotationPresent方法判断当前元素上是否存在指定类型的注释。存在则返回true，不存在则false。

**default boolean** isAnnotationPresent(Class<? **extends** Annotation> annotationClass)

#### getAnnotation

getAnnotation方法用于获取当前元素上是否含有指定类型的注释。如果存在，则返回该注释实例，不存在则返回null。

<T **extends** Annotation> T getAnnotation(Class<T> annotationClass);

#### getAnnotations

返回当前元素上的所有注释实例。

Annotation[] getAnnotations();

#### getDeclaredAnnotation

返回直接存在于此元素上的指定类型的注解（不包括父类的注解）

<T **extends** Annotation> T getDeclaredAnnotation(Class<T> annotationClass)

#### getDeclaredAnnotations

返回直接存在于此元素上的所有注解（不包括父类的注解），调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响，没有则返回长度为0的数组

Annotation[] getDeclaredAnnotations();

### 解析注解实例

通过反射解析注解，我们可以获取注解提供的辅助信息，然后根据获取的信息对注解修饰类执行不同的处理逻辑。

创建一个普通Java类，使用MyAnnotation自定义注解修饰。

@MyAnnocation(funName = **"this is not a fun"**, className = **"ScanHe.class"**)  
**public class** ScanHe {  
  
 @MyAnnocation(funName = **"printVal"**)  
 **public void** printVal(String val){  
 System.***out***.println(**"hello world"**+val);  
 }  
}

1. 获取ScanHe类上的所有注解，然后获取注解信息

Annotation[] annotations = ScanHe.**class**.getAnnotations();  
**for**(Annotation annotation:annotations){  
 **if**(annotation **instanceof** MyAnnocation){  
 System.***out***.println(((MyAnnocation) annotation).className());  
 System.***out***.println(((MyAnnocation) annotation).funName());  
 System.***out***.println(((MyAnnocation) annotation).isSuit());  
 }  
}

1. 判断ScanHe类有没有被MyAnnotation注解修饰，如果有，就获取注解并解析

**if**(ScanHe.**class**.isAnnotationPresent(MyAnnocation.**class**)){  
 MyAnnocation annocation = ScanHe.**class**.getAnnotation(MyAnnocation.**class**);  
 System.***out***.println(annocation.className());  
 System.***out***.println(annocation.funName());  
 System.***out***.println(annocation.isSuit());  
}

1. 获取ScanHe中的所有方法，并对方法上的注解解析

Method[] methods = ScanHe.**class**.getMethods();  
**for**(Method method:methods){  
 **if**(method.isAnnotationPresent(MyAnnocation.**class**)){  
 MyAnnocation annocation = method.getAnnotation(MyAnnocation.**class**);  
 System.***out***.println(annocation.className());  
 System.***out***.println(annocation.funName());  
 System.***out***.println(annocation.isSuit());  
 }  
}

# 多线程和高并发

http://blog.csdn.net/defonds/article/category/1363593

<http://www.iteye.com/magazines/131-Java-Concurrency>

<http://blog.csdn.net/u013256816/article/details/51325246>

# Java8新特性

## 接口默认/静态方法

在Java8之前，接口中只能定义抽象方法。即只有方法定义，不能有方法实现。一个实体类如果要实现一个接口，就必须实现接口中所有方法。

Java8提供了对接口的默认方法和静态方法的支持。在接口的默认方法和静态方法中都可以有方法实现。

接口的静态方法只能在接口中使用，也就是子类不能调用其实现接口的静态方法。

接口中默认方法，子类可以直接调用而不必实现。当然，如果有特殊需求，接口中的默认方法不能满足，也可以在子类中重写。

默认方法的应用场景：

1. 对旧的接口进行扩展。在旧的接口中新增一个方法，在Java8以前，需要在该接口的所有实现类中都添加新增方法，比较麻烦。此时可以将新增方法声明为默认方法。
2. 如果接口中提供一个比较通用的方法，其子类可以直接调用，没必要每个实现类都重写该方法。

**void** test();  
  
**default void** concatClassName(){  
 *printStaticMethod*();  
 System.***out***.println(**"this is a default method"**);  
}  
  
**static void** printStaticMethod(){  
 System.***out***.println(**"this is a static method"**);  
}

在实现类中可以直接调用：

@Override  
**public void** test() {  
 System.***out***.println(**"this is a common method"**);  
}  
  
**public static void** main(String[] args){  
 **new** DefaultMethod().test();  
 **new** DefaultMethod().concatClassName();  
}

## 函数式接口

函数式接口在本质上还是一个接口；特殊的一点是函数式接口是SAM类型的接口（Single Abstract Method），接口中只有一个抽象方法。使用@FunctionalInterface进行修饰。

因为函数式接口只有一个抽象方法，使得在以其作为参数的方法中，可以使用Lambda表达式代替。Lambda表达式会自动覆盖函数式接口的抽象方法（个人理解）

Java8内置的四大核心型函数式接口：

1. 函数型接口 Function<T, R>
2. 断言型接口 Predicate<T>
3. 消费型接口 Consumer<T>
4. 供给型接口 Supplier<T>

### 接口声明方式

接口的声明有两种方式，一种是匿名内部类形式；一种是Lambda表达式形式；

#### **匿名内部类方式**

Stream.*of*(**"abc"**,**"dec"**,**"hid"**,**"yhc"**).filter(**new** Predicate <String>() {  
 @Override  
 **public boolean** test(String s) {  
 **return** s.contains(**"c"**);  
 }  
}).forEach(System.***out***::println);

在方法中直接new一个函数式接口对象，实现其抽象方法，进行逻辑处理。

#### Lambda表达式方式

Stream.*of*(**"abc"**,**"dec"**,**"hid"**,**"yhc"**).filter(

s -> s.contains(**"c"**)).forEach(System.***out***::println);

Lambda表达式中的前面部分是方法的参数，后半部分是方法体。如果只有一条语句，可以不用{}；如果有多条语句，需要将方法体用{}括起来。

注意：

Lambda表达式会自动覆盖函数式接口的抽象方法，其抽象方法是什么返回类型，Lambda表达式的方法体重就需要返回什么类型。

比如上面代码，Predicate接口的抽象方法test()返回的是一个boolean型值，则使用Lambda表达式时，方法体也必须返回一个boolean型。如果Lambda表达式后面部分是一条语句，则同等于return s.contains(**"c"**)；如果多条语句，则在{}中要使用return主动返回一个boolean型。

Stream.*of*(**"abc"**,**"dec"**,**"hid"**,**"yhc"**).filter(s ->   
 {**boolean** flag = s.contains(**"c"**);  
 **return** flag;}  
).forEach(System.***out***::println);

### Function接口

Function接口中的唯一抽象方法是apply()。

R apply(T t);

apply()方法接收一个泛型参数，在方法体中进行逻辑处理后再返回一个泛型。它的参数和返回值类型可以相同，可以不同。

Function接口中的apply方法和Consumer接口中的accept接口相比，apply方法可以返回一个泛型。

**new** Function<OmAccounts, Object>() {  
 @Override  
 **public** Object apply(OmAccounts o) {  
 **return null**;  
 }  
}

实现Function接口时需要指定两个参数类型，其中第一个是apply方法参数类型，第二个是apply方法返回值类型。

### Predicate接口

Predicate接口中有五个方法，但只有一个test()方法是抽象方法，是需要实现的。其他四个都是接口默认方法。

**boolean** test(T t);

test()方法返回一个boolean型。所以在使用Predicate接口时，方法体中返回值必须是一个boolean型数据。

IntStream.*of*(1,3,5,6,8,3,9).filter(**new** IntPredicate() {  
 @Override  
 **public boolean** test(**int** value) {  
 **return** value < 5;  
 }  
}).forEach(System.***out***::println);

IntStream.*of*(1,3,5,6,8,3,9).filter(item -> item<5).forEach(System.***out***::println);

上面两种写法是一样的，这个方法是过滤出流中数字小于5的元素，并打印出来。

### Consumer接口

Consumer接口中有2个方法，其中一个抽象方法accept，另一个是默认方法。

Consumer接口是消费型接口，它接收一个参数，但没有返回值。类似于生活中的消费行为。

**void** accept(T t);

accept()方法接收一个泛型参数，在方法体中对参数进行处理，没有返回值。

网上的例子：

**class** Student{  
 String **firstName**;  
 String **lastName**;  
 Double **grade**;  
 Double **feeDiscount** = 0.0;  
 Double **baseFee** = 20000.0;  
  
 **public** Student(String firstName, String lastName, Double grade) {  
 **this**.**firstName** = firstName;  
 **this**.**lastName** = lastName;  
 **this**.**grade** = grade;  
 }  
  
 **public void** printFee(){  
 Double newFee = **baseFee** - ((**baseFee** \* **feeDiscount**) / 100);  
 System.***out***.println(**"The fee after discount: "** + newFee);  
 }  
}

**public static** Student updateStudentFee(Student student, Predicate<Student> predicate, Consumer<Student> consumer){  
 **if** ( predicate.test(student)){  
 *//Use the consumer to update the discount value.* consumer.accept(student);  
 }  
 **return** student;  
}

**public static void** main(String[] args){  
 Student student1 = **new** Student(**"Ashok"**,**"Kumar"**, 9.5);  
 student1 = *updateStudentFee*(student1,  
 student -> student.**grade** > 8.5,  
 student -> student.**feeDiscount** = 30.0);  
 student1.printFee();  
 Student student2 = **new** Student(**"Rajat"**,**"Verma"**, 8.0);  
 student2 = *updateStudentFee*(student2,  
 student -> student.**grade** >= 8,  
 student -> student.**feeDiscount** = 20.0);  
 student2.printFee();  
}

### Supplier接口

Supplier接口中的唯一抽象方法是get()方法。可以在get方法中定义一个数据生成逻辑，然后调用get方法，就会获得一个新的数据。Stream的generate方法的参数就是一个Supplier接口，返回一个无限长的流。

Supplier接口是供给型接口，它的get方法不需要接收参数，但会返回一个泛型，类似于生活中的生产行为。

T get();

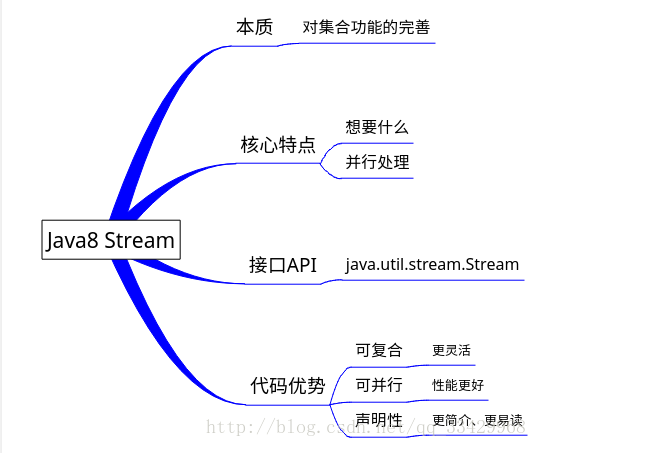
Stream.*generate*(**new** Supplier <Character>() {  
 **char c** = 65;  
 @Override  
 **public** Character get() {  
 **c** =(**char**)(**c**+1);  
 **return c**;  
 }  
}).limit(10).forEach(System.***out***::println);

## Stram流

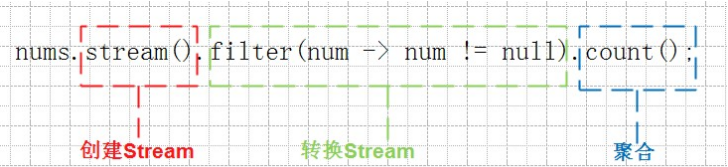
在Java中集合具有非常广泛的应用，当涉及到复杂的业务逻辑处理时，需要借助大量迭代器对象对集合进行多次遍历，才能完成操作。而Java8中Stream的流出现大大简化了Java代码对集合的操作过程。

### 流的定义

* 流是元素的集合。
* 它不是数据结构，不会保存数据，只是操作集合的过程中数据的一种存在形式。
* 在Java8之前，对集合的操作通过迭代对象Iterator完成。如果我们想要集合中的元素执行某个操作，比如选出集合中数字小于10的元素；需要通过迭代器Iterator对集合进行遍历，然后对遍历的每个元素执行函数调用。而对于Stream，如果我们想要对流中的元素执行某个函数，只需要把函数交给Stream即可，具体它是如何遍历元素，如何调用函数的，不需要我们关心。



### Stream语法



关于流的操作基本可以分为以下三个过程：

1. 创建流（Stream）对象，Stream生命开始的地方
2. 转换Stream，也就是流中元素的逻辑处理，每次转换原有Stream对象不改变，返回一个新的Stream对象
3. Stram聚合操作（Reduce和Collect方法）

### 获取Stream实例

最常用的创建Stream对象的两种方法：

1. 通过Stream接口的静态工厂方法（注意：Java8里接口可以带静态方法）；
2. 通过Collection接口的默认方法（默认方法：Default method，也是Java8中的一个新特性，就是接口中的一个带有实现的方法）–stream()，把一个Collection对象转换成Stream

#### Stream接口的静态工厂方法

**Stream.of()**方法接受两种方式的参数，一种是多个值；一种接受单一值；

Stream stream = Stream.*of*(**"abc"**,**"dfc"**,**"hig"**);  
Stream stream1 = Stream.*of*(**"abcde"**);

单一值中可以传递一个字符串，也可以传递一个数组。

**Stream.generator()**方法接受一个Supplier对象，生成一个无限长度的Stream，其元素的生成是通过给定的Supplier（这个接口可以看成一个对象的工厂，每次调用返回一个给定类型的对象）

Stream stream = Stream.*generate*(**new** Supplier <Object>() {  
 **int i** = 1; //自定义元素生成规则，也可以使用Math.random()生成随机数  
 @Override  
 **public** Object get() {  
 **i** = **i** + 2;  
 **return i**;  
 }  
});

Stream的generator方法会生成一个无限长度的Stream，不过这个无限Stream是懒加载的，也就是不调用的时候并不会主动生成元素。

这种无限长Stream一般和Stream的limit()方法配合使用。

Stream.*generate*(**new** Supplier <Object>() {  
 @Override  
 **public** Object get() {  
 **return** Math.*random*();  
 }  
});  
Stream.*generate*(() -> Math.*random*());  
Stream.*generate*(Math::*random*);//方法引用的语法只能在静态方法中使用，generator是静态方法

三条语句的作用都是一样的，只是使用了lambda表达式和方法引用的语法来简化代码。

**Stream.iterate()**方法也是生成无限长度的Stream，和generator不同的是，iterate方法接受两个参数，第一个参数作为种子值，第二个参数是一个函数接口UnaryOperator（可以用Lambda定义），其元素的生成规则是重复对给定的种子值(seed)调用UnaryOperator函数。Stream中的元素可以认为是：seed，f(seed),f(f(seed))无限循环。

Stream.*iterate*(1, num -> num+2).limit(10).forEach(System.***out***::println);

上面这个函数的逻辑就是生成一个包含10个元素的从1（种子值）开始，每次递加2的Stream流，然后输出流中的元素。

#### Collection子类获取Stream

Collection接口有一个stream方法，所以其所有子类都都可以获取对应的Stream对象。如List，Arrays等对象。

Arrays.*asList*(**"abc"**,**"hig"**,**"igc"**).stream();

### 转换Stream

#### filter

filter()方法，接收一个函数接口Predicate参数，对于Stream中包含的元素使用给定的过滤函数进行过滤操作，新生成的Stream只包含符合条件的元素

Predicate接口中有一个test方法，返回一个boolean值。所以，Lambda表达式中的逻辑处理部分必须返回的是boolean型。

Stream<String> stream = Arrays.*asList*(**"abc"**,**"hig"**,**"igc"**).stream();  
stream.filter(str -> str.contains(**"c"**)).forEach(System.***out***::println);

过滤掉流中不含有c的元素，然后将含有c的打印出来

#### map、flatMap

对于Stream中包含的元素使用给定的转换函数进行转换操作，新生成的Stream只包含转换生成的元素。

这个方法有三个对于原始类型的变种方法，分别是：mapToInt，mapToLong和mapToDouble。这三个方法也比较好理解，比如mapToInt就是把原始Stream转换成一个新的Stream，这个新生成的Stream中的元素都是int类型。之所以会有这样三个变种方法，可以免除自动装箱/拆箱的额外消耗；

同样，Stream提供了IntStream、LongStream、DoubleStream对象，避免自动装箱/拆箱的额外消耗；

Stream<String> stream = Arrays.*asList*(**"abc"**,**"hig"**,**"igc"**).stream();  
stream.map(str -> str.concat(**"\*\_\*"**)).forEach(System.***out***::println);

Stream的map方法执行的是流的转换逻辑，它接收的参数是一个函数，函数可以返回任何类型的值。

flatMap方法和map类似，不同的是其每个元素转换得到的是Stream对象，会把子Stream中的元素压缩到父集合中

#### peek

生成一个包含原Stream的所有元素的新Stream，同时会提供一个消费函数（Consumer实例），新Stream每个元素被消费的时候都会执行给定的消费函数；

Stream<T> peek(Consumer<? **super** T> action);

peek()方法接收一个Consumer实例，然后对流中的每个元素执行Consumer实例中的逻辑操作。

#### limit、skip

 limit()方法对一个Stream进行截断操作，获取其前N个元素，如果原Stream中包含的元素个数小于N，那就获取其所有的元素；

skip()方法也是对一个Stream进行截断操作，返回一个丢弃原Stream的前N个元素后剩下元素组成的新Stream，如果原Stream中包含的元素个数小于N，那么返回空Stream；

IntStream stream = IntStream.*iterate*(1, i -> i+1).limit(10);  
stream.limit(5);  
stream.skip(11);

#### 普通方法

forEach()方法用于遍历流中的元素。

distinct()方法用于对Stream中包含的元素进行去重操作（去重逻辑依赖元素的equals方法，如果是实体类对象，有时需要重新equals和hashCode方法），新生成的Stream中没有重复的元素；

IntStream stream = IntStream.*of*(1,2,4,5,2,5,6);  
stream.distinct();

Sort()方法用于对流中的元素进行排序

Stream stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.sorted();

findFirst()方法用于获取Stream流中的第一个元素，如果Stream为空，则返回一个Optional对象。

Stream<Integer> stream = Stream.*of*();  
Optional<Integer> result = stream.findFirst();

anyMatch、allMatch、noneMatch方法都接受一个Predicate函数接口作为参数，返回一个boolean型，判断是否匹配成功。anyMath方法是只要流中存在满足匹配条件的元素，就返回true；allMatch方法要求流中所有的元素都必须满足匹配条件，才返回true；noneMatch方法要求流中的所有元素都不满足匹配条件，返回true。

Stream stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.anyMatch(i -> (**int**)i > 5);  
stream.allMatch(i -> (**int**)i > 5);  
stream.noneMatch(i -> (**int**)i > 5);

### Stream汇聚

汇聚操作（也称为折叠）接受一个元素序列为输入，反复使用某个合并操作，把序列中的元素合并成一个汇总的结果。比如查找一个数字列表的总和或者最大值，或者把这些数字累积成一个List对象。

Stream接口有一些通用的汇聚操作，比如reduce()和collect()；也有一些特定用途的汇聚操作，比如sum(),max()和count()。

汇聚可以分成两部分来看：

1. 可变汇聚：通过反复修改可变对象，把输入的元素们累积到一个可变的容器中，比如Collection或者StringBuilder；
2. 其他汇聚：除了可变汇聚，剩余的都是其他汇聚。它是通过把前一次的汇聚结果当成下一次的入参，反复如此。比如reduce，count，allMatch；

#### collect

可变汇聚对应的只有一个方法：collect，正如其名字显示的，它可以把Stream中的要有元素收集到一个结果容器中（比如Collection）。先看一下最通用的collect方法的定义（还有其他override方法）：

<R> R collect(Supplier<R> supplier,  
 BiConsumer<R, ? super T> accumulator,  
 BiConsumer<R, R> combiner);

先来看看这三个参数的含义：Supplier supplier是一个工厂函数，用来生成一个新的容器；BiConsumer accumulator也是一个函数，用来把Stream中的元素添加到结果容器中；BiConsumer combiner还是一个函数，用来把中间状态的多个结果容器合并成为一个（并发的时候会用到）。

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.collect(() -> **new** ArrayList<Integer>(),  
 (list,item) -> list.add(item),  
 (list1, list2) -> list1.addAll(list2));

上面这段代码就是对一个元素是Integer类型的List，先过滤掉全部的null，然后把剩下的元素收集到一个新的List中。进一步看一下collect方法的三个参数，都是lambda形式的函数。

在上面代码中，collect()方法中的三个参数：

* 1、第一个函数生成一个新的ArrayList实例；
* 2、第二个函数接受两个参数，第一个是前面生成的ArrayList对象，二个是stream中包含的元素，函数体就是把stream中的元素加入ArrayList对象中。第二个函数被反复调用直到原stream的元素被消费完毕；
* 3、第三个函数也是接受两个参数，这两个都是ArrayList类型的，函数体就是把第二个ArrayList全部加入到第一个中；

collect()方法提供了多种接收参数的方式，上面这种参数接收过于麻烦，可以使用下面这种方式：

<R, A> R collect(Collector<? **super** T, A, R> collector);

在这个方法中，我们只需要传递一个Collector对象作为参数即可。另外，Collector一般不需要我们手动构造，Java8提供的Collectors工具类中提供了很多静态方法可以直接返回一个Collector对象，比如toList()、toSet()和toCollection()方法。

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.collect(Collectors.*toList*());

#### reduce

reduce()方法提供了三种实现方式，第一种是接收一个函数接口作为参数

Optional<T> reduce(BinaryOperator<T> accumulator);

这个函数接口接收两个参数

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
Optional<Integer> result = stream.reduce((sum, item) -> sum+=item);  
System.***out***.print(result.get());

第一次执行时，第一个参数是Stream中的第一个元素，第二个参数是Stream中的第二个元素；然后将函数的返回结果赋给第一个参数，Stream流中接下来的元素依次赋给函数中的第二个参数。

第一次运算时：sum = 1；item=2；

第二次运算时：sum = 3；item=4；

……

T reduce(T identity, BinaryOperator<T> accumulator);

这种方式中，reduce接收两个参数，其中第一个参数作为Stream为空的时候的返回值，第二个参数接收一个函数接口。

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
Integer result = stream.reduce(1, (sum, item) -> {System.***out***.println(sum + **" "**+item); sum+=item; **return** sum;});

函数仍旧接收两个参数，此时第一次执行时，第一个参数是reduce()中的第一个参数，第二个参数是Stream中的第一个元素；然后将函数的返回结果赋给第一个参数，Stream流中接下来的元素依次赋给函数中的第二个参数。

第一次运算时：sum = 1；item=1；

第二次运算时：sum = 2；item=2；

……

## parallelStream并行流

parallelStream其实就是一个并行执行的流.它通过默认的ForkJoinPool,可能提高你的多线程任务的速度。

<http://blog.csdn.net/u011001723/article/details/52794455>

## Lambda表达式

Lambda表达式更像是一种函数定义方式，是一段带有输入参数的可执行代码块。

与普通函数定义相比，使用Lambda表达式可以让代码更简洁；

函数一般将方法与一个函数名绑定，Lambda表达式不需要与标识符绑定；

### Lambda语法

Lambda表达式的一般语法，

(Type1 param1, Type2 param2, ..., TypeN paramN) -> {  
 statement1;  
 statement2;  
 ......  
 **return** statmentM;  
}

上述是Lambda的一般语法，在实际使用时部分代码还可以省略。

Stream.*of*(**"A"**,**"B"**,**"C"**).map((String s) -> {s = s.concat(**"\_"**); **return** s;}).forEach(System.***out***::println);

这段代码是对Stream中的每个元素都拼接一个\_符号，然后打印。

Lambda表达式省略情形：

1、参数类型省略–绝大多数情况，编译器都可以从上下文环境中推断出lambda表达式的参数类型；

2、当lambda表达式的参数个数只有一个，可以省略小括号；

3、当lambda表达式只包含一条语句时，可以省略大括号、return和语句结尾的分号；

4、在静态方法中可以使用方法引用；

Stream.*of*(**"A"**,**"B"**,**"C"**).map(s -> s.concat(**"\_"**)).forEach(System.***out***::println);

### 访问外部变量

接口中定义的变量只能是final类型，也就是不可变。

接口可以访问接口的外部变量，Lambda表达式也可以，只是Lambda只能访问外部的final修饰的变量。即不可变变量。这里的不变是指引用不变。即如果一个final变量指向了一个对象，那么它将一直指向这个对象，引用不变，但它指向的对象可变。

String[] arry = **new** String[]{**"A"**, **"B"**, **"C"**};  
**for**(Integer i:Arrays.*asList*(1,2,3)){  
 Arrays.*stream*(arry).map(s -> s.concat(i.toString())).forEach(System.***out***::println);  
}

在Lambda表达式中引用了外部变量i，此时i引用不变，引用的对象值改变了。

String[] arry = **new** String[]{**"A"**, **"B"**, **"C"**};  
**for**(Integer i=1; i<=3; i++ ){  
 Arrays.*stream*(arry).map(s -> s.concat(i.toString())).forEach(System.***out***::println);  
}

这种写法是错误的，此时i的引用改变了，不符合final变量定义。

### 方法引用

使用方法引用可以简化代码。但方法引用的使用有条件限制。**方法引用的唯一用途是支持Lambda的简写**

方法引用是用来直接访问类或者实例的已经存在的方法或者构造方法。方法引用提供了一种引用而不执行方法的方式，它需要由兼容的函数式接口构成的目标类型上下文。

当Lambda表达式中只是执行一个方法调用时，不用Lambda表达式，直接通过方法引用的形式可读性更高一些。方法引用是一种更简洁易懂的Lambda表达式。

1、引用静态方法   
ContainingClass::staticMethodName   
例子: String::valueOf，对应的Lambda：(s) -> String.valueOf(s)   
比较容易理解，和静态方法调用相比，只是把.换为::

2、引用特定对象的实例方法   
containingObject::instanceMethodName   
例子: x::toString，对应的Lambda：() -> this.toString()   
与引用静态方法相比，都换为实例的而已

3、引用构造函数   
ClassName::new   
例子: String::new，对应的Lambda：() -> new String()   
构造函数本质上是静态方法，只是方法名字比较特殊。

4、引用特定类型的任意对象的实例方法   
ContainingType::methodName   
例子: String::toString，对应的Lambda：(s) -> s.toString()   
太难以理解了。难以理解的东西，也难以维护。建议还是不要用该种方法引用。   
实例方法要通过对象来调用，方法引用对应Lambda，Lambda的第一个参数会成为调用实例方法的对象。

## Optional类

Optional是Java8提供的为了解决null安全问题的一个API。正确使用Optional可以使我们代码中很多繁琐、丑陋的设计变得十分优雅。

Optional是一个类，不是接口。

Optional是一个可以为null的容器对象，如果值在则isPresent()方法会返回true，调用get()方法会返回该对象。

### 获取Optional实例

Optional类中提供了三种方法用于获取Optional实例。

1、empty()方法

**private static final** Optional<?> ***EMPTY*** = **new** Optional<>();

**public static**<T> Optional<T> empty() {  
 @SuppressWarnings(**"unchecked"**)  
 Optional<T> t = (Optional<T>) ***EMPTY***;  
 **return** t;  
}

empty()方法返回的是一个新的Optional对象。但Optional中的值为空。

2、of()方法

**public static** <T> Optional<T> of(T value) {  
 **return new** Optional<>(value);  
}

of方法接收一个泛型对象作为参数，如果参数value为null，代码运行时会直接抛出空指针异常。因此，使用of方法获取Optional实例时，应该确保value不为null。

3、ofNullable()方法

**public static** <T> Optional<T> ofNullable(T value) {  
 **return** value == **null** ? *empty*() : *of*(value);  
}

ofNullable方法也是接收一个泛型对象作为参数，和of方法的区别是，如果参数value为null，会调用empty方法自动创建一个新的Optional对象

### Optional普通方法

isPresent()

isPresent方法用于判断Optional对象当中的值是否为null，是则返回false；否则返回true。

LambdaTest lambdaTest = **null**;  
System.***out***.println(Optional.*ofNullable*(lambdaTest).isPresent()); //false

get()

get方法用于获取Optional对象中的值，如果没值则抛出NoSuchElementException异常。

LambdaTest lambdaTest = **new** LambdaTest();  
Optional.*of*(lambdaTest).get();

ifParsent()

ifParsent方法接收一个消费型接口Consumer对象作为参数。

注意：

Lambda表达式方法参数是Optional对象中的值，不是Optional对象本身。

LambdaTest lambdaTest = **new** LambdaTest();  
Optional.*of*(lambdaTest).ifPresent(u -> System.***out***.println(u **instanceof** LambdaTest));

orElse()/orElseGet()/orElseThrow()

orElse方法，如果Optional实例有值则将其返回，否则返回orElse方法传入的参数

orElseGet与orElse方法类似，区别在于得到的默认值。orElse方法将传入的参数作为默认值，orElseGet方法可以接受[Supplier接口](http://blog.sanaulla.info/2013/04/02/supplier-interface-in-java-util-function-package-in-java-8/)的实现用来生成默认值。

在orElseThrow中我们可以传入一个lambda表达式或方法，如果值不存在来抛出异常。

在orElse和orElseGet方法中，传入的参数和Supplier接口生成的默认值都必须和Optional对象中的值是同一个类型。

LambdaTest lambdaTest = **new** LambdaTest();  
Optional.*of*(lambdaTest).orElse(**new** LambdaTest());  
Optional.*of*(lambdaTest).orElseGet(LambdaTest::**new**);  
Optional.*of*(lambdaTest).orElseThrow(NullPointerException::**new**);

### map()

map方法中接收一个函数型接口Function作为参数。在执行时，map会先判断Optional对象中的值是不是为空，如果值为空，调用empty方法新建一个空的Optional对象；如果不为空，则调用Function函数对Optional的值进行处理。

### flatMap()

flatMap方法也是接收一个函数型接口Function作为参数。它和map方法类似，也是用于对Optional对象进行处理，不同的是flatmap返回的是Optional对象，而map方法需要对返回值进行Optional封装。

### filter()

filter个方法通过传入限定条件对Optional实例的值进行过滤。然后返回满足条件的值。

## DateTime日期API

http://www.blogjava.net/4cai/archive/2014/04/29/413041.html