



# Java8新特性

讲师：宋红康

新浪微博：尚硅谷-宋红康

# 主要内容

1. 接口的新特性
2. 注解的新特性
3. 集合的底层源码实现
4. 新日期时间的API
5. Optional类的使用
6. Lambda 表达式(Lambda Expressions)
7. Stream API



Java 9已于今年9月份发布，那么还有必要学习java 8 吗？

# Java 8新特性简介

Java 8 (又称为 jdk 1.8) 是 Java 语言开发的一个主要版本。Java 8 是oracle公司于2014年3月发布，可以看成是自Java 5 以来最具革命性的版本。Java 8 为Java语言、编译器、类库、开发工具与JVM带来了大量新特性。



# Java 8新特性简介

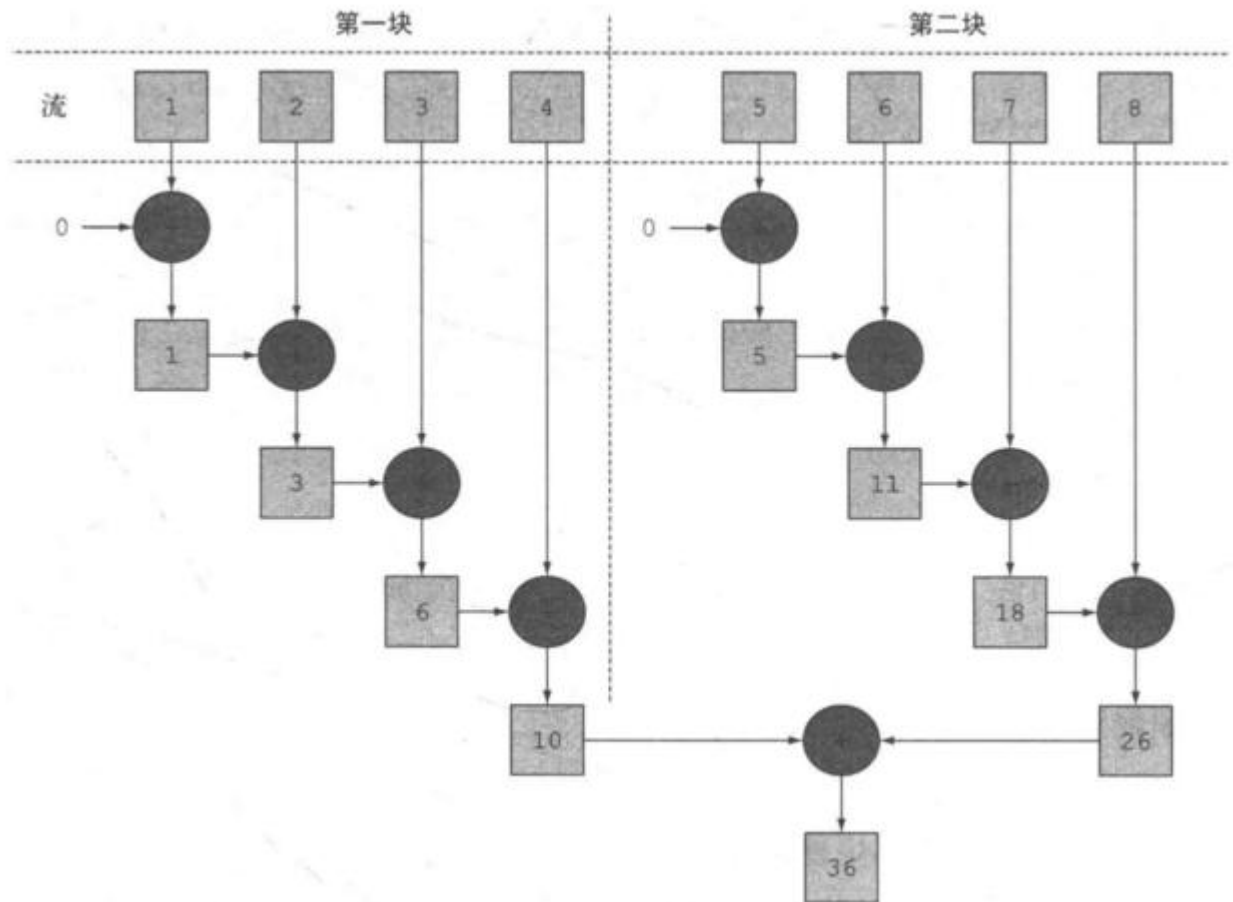
- 代码更少(增加了新的语法: **Lambda 表达式**)
- 强大的 **Stream API**
- 速度更快
- 最大化减少空指针异常: Optional
- Nashorn引擎, 允许在JVM上运行JS应用
- 便于并行

# 并行流与串行流

Java 8 中将并行进行了优化，我们可以很容易的对数据进行并行操作。Stream API 可以声明性地通过 `parallel()` 与 `sequential()` 在并行流与顺序流之间进行切换。

并行流就是把一个内容(数组或集合)分成多个数据块，并用不同的线程分别处理每个数据块的流。这样一来，你就可以自动把给定操作的工作负荷分配给多核处理器的所有内核，让他们都忙起来。整个过程无需程序员显示实现优化。

```
public static long parallelSum(long n){  
    return Stream.iterate(1L,i -> i +1) .limit(n) .parallel()  
        .reduce(0L,Long::sum);  
}
```



# 1 - 接口的新特性



## Java 8中关于接口的改进

Java 8中，你可以为接口添加**静态方法**和**默认方法**。从技术角度来说，这是完全合法的，只是它看起来违反了接口作为一个抽象定义的理念。

**静态方法：**使用 **static** 关键字修饰。可以通过接口直接调用静态方法，并执行其方法体。我们经常在相互一起使用的类中使用静态方法。你可以在标准库中找到像 `Collection/Collections` 或者 `Path/Paths` 这样成对的接口和类。

**默认方法：**默认方法使用 **default** 关键字修饰。可以通过实现类对象来调用。我们在已有的接口中提供新方法的同时，还保持了与旧版本代码的兼容性。

比如：java 8 API中对 `Collection`、`List`、`Comparator` 等接口提供了丰富的默认方法。

```
public interface AA {  
    double PI = 3.14;  
  
    public default void method() {  
        System.out.println("北京");  
    }  
  
    default String method1() {  
        return "上海";  
    }  
  
    public static void method2() {  
        System.out.println("hello lambda!");  
    }  
}
```

# 接口中的默认方法

## 接口默认方法的“类优先”原则

若一个接口中定义了一个默认方法，而另外一个父类或接口中又定义了一个同名的方法时

- 选择父类中的方法。如果一个父类提供了具体的实现，那么接口中具有相同名称和参数的默认方法会被忽略。
- 接口冲突。如果一个父接口提供一个默认方法，而另一个接口也提供了一个具有相同名称和参数列表的方法（不管方法是否是默认方法），那么实现类必须覆盖该方法来解决冲突

# 接口冲突的解决方式

```
interface MyFunc{  
    default String getName(){  
        return "Hello Java8!";  
    }  
}  
  
interface Named{  
    default String getName(){  
        return "Hello atguigu!";  
    }  
}  
  
class MyClass implements MyFunc, Named{  
    public String getName(){  
        return Named.super.getName();  
    }  
}
```



## 2 - 注解的新特性

# Java 8 中关于注解的修改

Java 8对注解处理提供了两点改进：**可重复的注解**及**可用于类型的注解**。此外，反射也得到了加强，在Java8中能够得到方法参数的名称。这会简化标注在方法参数上的注解。

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotations {
    MyAnnotation[] value();
}
```

```
@Repeatable(MyAnnotations.class)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ElementType.TYPE_PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    String value();
}
```

```
@MyAnnotation("Hello")
@MyAnnotation("World")
public void show(@MyAnnotation("abc") String str){
}
```

```
@Target({TYPE, FIELD, METHOD, PARAMETER, PACKAGE, CONSTRUCTOR,  
LOCAL_VARIABLE, TYPE_PARAMETER})  
@Retention(RetentionPolicy.RUNTIME)  
@Repeatable(MyAnnotations.class)  
public @interface MyAnnotation {  
    String[] value();  
}
```

应用场景：

```
public Person(@MyAnnotation(value="notnull")String name){  
    this.name = name;  
}
```

## 3 - 集合的底层源码实现



## ArrayList、LinkedList、Vector区别？

|-----List子接口：存储序的、可重复的数据 ---->"动态"数组

|-----ArrayList:作为List的主要实现类；线程不安全的，效率高；**底层使用数组实现**

(Collections中定义了synchronizedList(List list)将此ArrayList转化为线程安全的)

|-----LinkedList:对于频繁的插入、删除操作，我们建议使用此类，因为效率高；内存消耗较ArrayList大；**底层使用双向链表实现**

|-----Vector:List的古老实现类；线程安全的，效率低；**底层使用数组实现**

然后可以分析一下ArrayList和LinkedList的底层源码实现

## 补充：你不可不知的数据结构



## ArrayList 源码分析:

jdk7:

```
ArrayList list = new ArrayList();//初始化一个长度为10的Object[] elementData  
sysout(list.size());//返回存储的元素的个数:0  
list.add(123);  
list.add(345);
```

...

当添加第11个元素时，需要扩容，默认扩容为原来的1.5倍。还需要将原有数组中的数据复制到新的数组中。

删除操作：如果删除某一个数组位置的元素，需要其后面的元素依次前移。

```
remove(Object obj) / remove(int index)
```

jdk8:

```
ArrayList list = new ArrayList();//初始化一个长度为0的Object[] elementData  
sysout(list.size());//返回存储的元素的个数:0  
list.add(123);//此时才创建一个长度为10的Object[] elementData  
list.add(345);
```

...

当添加第11个元素时，需要扩容，默认扩容为原来的1.5倍。还需要将原有数组中的数据复制到新的数组中。

开发时的启示:

1. 建议使用：`ArrayList list = new ArrayList(int length);`
2. jdk8延迟了底层数组的创建：内存的使用率；对象的创建更快



## LinkedList 源码分析:

LinkedList: 底层使用双向链表存储添加的元素

```
void linkLast(E e) {  
    final Node<E> l = last;  
    final Node<E> newNode = new Node<>(l, e, null);  
    last = newNode;  
    if (l == null)  
        first = newNode;  
    else  
        l.next = newNode;  
    size++;  
    modCount++;  
}
```

内部类体现:

```
private static class Node<E> {  
    E item;  
    Node<E> next;  
    Node<E> prev;  
}
```



## HashMap和Hashtable的对比

HashMap:Map的主要实现类；线程不安全的，效率高；可以存储null的key和value  
(存储结构：jdk7.0 数组+链表； jdk8.0 数组+链表+红黑树)

Hashtable:Map的古老实现类；线程安全的，效率低；不可以存储null的key和value

## HashMap的底层实现原理

HashMap map = new HashMap();//底层创建了长度为16的Entry数组  
向HashMap中添加entry1(key, value), 需要首先计算entry1中key的哈希值(根据key所在类的hashCode()计算得到), 此哈希值经过处理以后, 得到在底层Entry[]数组中要存储的位置i.如果位置i上没有元素, 则entry1直接添加成功。如果位置i上已经存在entry2(或还有链表存在的entry3, entry4),则需要通过循环的方法, 依次比较entry1中key和其他的entry是否equals.如果返回值为true.则使用entry1的value去替换equals为true的entry的value.如果遍历一遍以后, 发现所有的equals返回都为false,则entry1仍可添加成功。entry1指向原有的entry元素。

默认情况下, 如果添加元素的长度  $\geq \text{DEFAULT\_INITIAL\_CAPACITY} * \text{DEFAULT\_LOAD\_FACTOR}$  (临界值threshold默认值为12)且新要添加的数组位置不为null的情况下, 就进行扩容。默认扩容为原有长度的2倍。将原有的数据复制到新的数组中。

## HashMap的底层实现原理

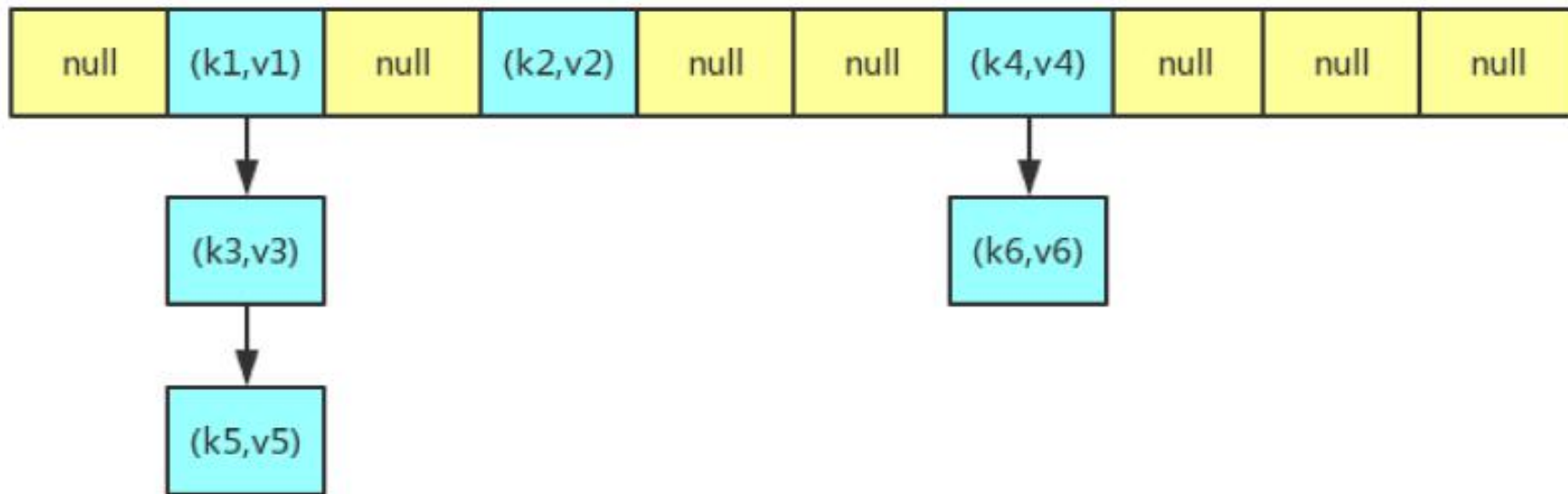
### jdk 8.0 :

1. `HashMap map = new HashMap();` //默认情况下，先不创建长度为16的数组。
2. 当首次调用 `map.put()` 时，再创建长度为16的数组
3. 当数组指定索引位置的链表长度  $> 8$  时，且 `map` 中的数组的长度  $> 64$  时，此索引位置上的所有 `entry` 使用红黑树进行存储。而 `jdk 7` 中没有红黑树结构
4. 新添加的元素如果与现有的元素以链表方式存储的时候：  
“七上八下”：`jdk7`：新添加的当链表头，`jdk8`：新添加的当链表尾

# HashMap的存储结构

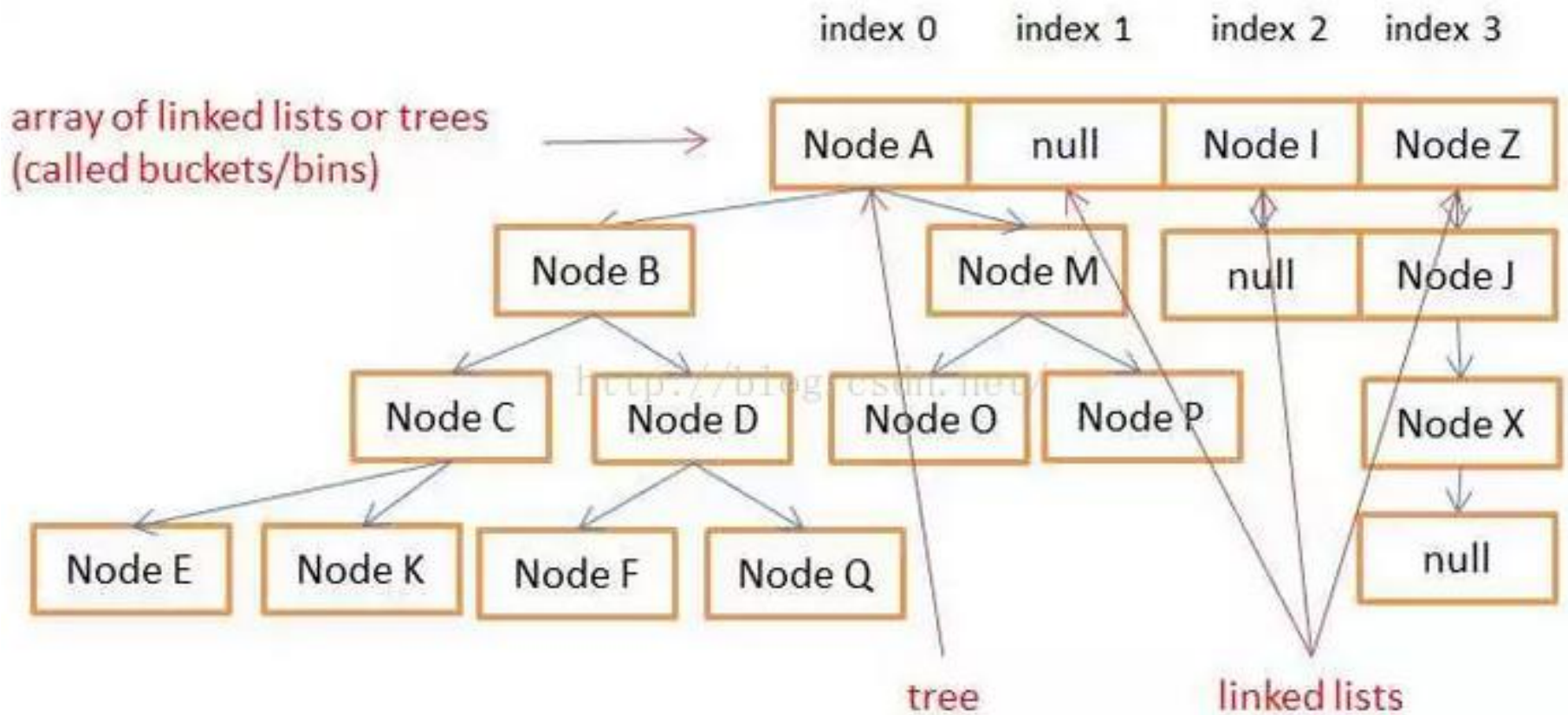
JDK 7及以前版本：HashMap是数组+链表结构(即为链地址法)

JDK 8版本发布以后：HashMap是数组+链表+红黑树实现。





# HashMap的存储结构



## 负载因子值的大小，对HashMap有什么影响

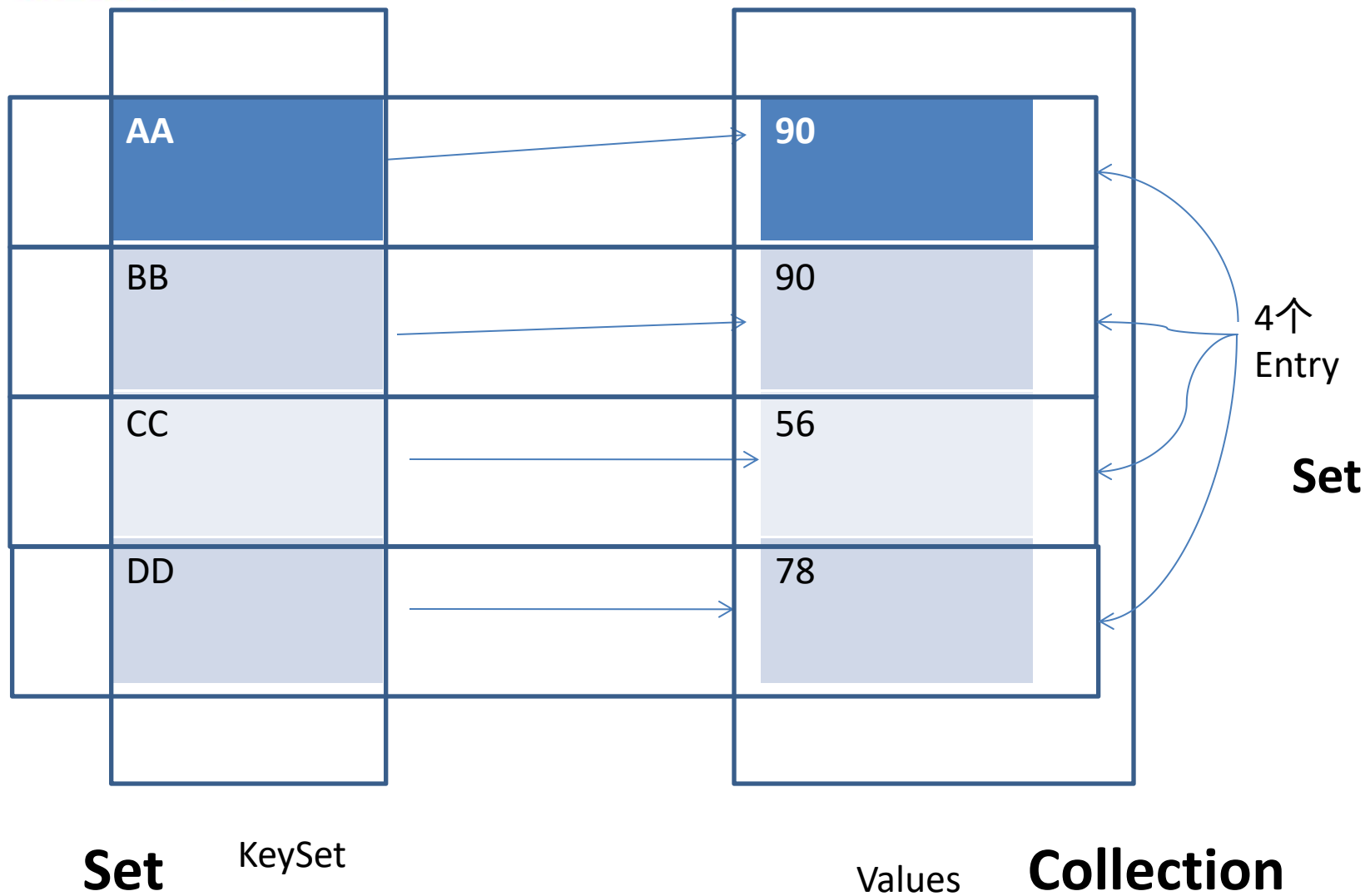
负载因子的大小决定了HashMap的数据密度，负载因子越大密度越大，发生碰撞的几率越高，数组中的链表越容易长，造成查询或插入时的比较次数增多，性能会下降。负载因子越小，就越容易触发扩容，数据密度也越小，意味着发生碰撞的几率越小，数组中的链表也就越短，查询和插入时比较的次数也越小，性能会更高。但是会浪费一定的内容空间。而且经常扩容也会影响性能，建议初始化预设大一点的空间。

## 拓展1: LinkedHashMap 源码分析

LinkedHashMap继承于HashMap，在HashMap底层结构的基础上额外添加了一对链表：

```
static class Entry<K,V> extends HashMap.Node<K,V> {  
    Entry<K,V> before, after;  
}
```

## 拓展2: HashSet / LinkedHashSet 等底层源码有变化吗？





```
public static void main(String[] args) {  
    List list = new ArrayList();  
    list.add(1);  
    list.add(2);  
    list.add(3);  
    updateList(list);  
    System.out.println(list);  
}
```

```
private static void updateList(List list) {  
    list.remove(2);  
}
```

```
HashSet set = new HashSet();  
Person p1 = new Person(1001,"AA");  
Person p2 = new Person(1002,"BB");  
set.add(p1);  
set.add(p2);  
p1.name = "CC";  
set.remove(p1);  
System.out.println(set);  
set.add(new Person(1001,"CC"));  
System.out.println(set);  
set.add(new Person(1001,"AA"));  
System.out.println(set);
```

其中，其中Person类中重写了hashCode()和equal()方法

## 4 - 新日期时间的API

# 新时间日期API

如果我们可以跟别人说：“我们在1502643933071见面，别晚了！”那么就再简单不过了。但是我们希望时间与昼夜和四季有关，于是事情就变复杂了。JDK 1.0中包含了一个java.util.Date类，但是它的大多数方法已经在JDK 1.1引入Calendar类之后被弃用了。而Calendar并不比Date好多少。它们面临的问题是：

可变性：像日期和时间这样的类应该是不可变的。

偏移性：Date中的年份是从1900开始的，而月份都从0开始。

格式化：格式化只对Date有用，Calendar则不行。

此外，它们也不是线程安全的；不能处理闰秒等。

总结：对日期和时间的操作一直是Java程序员最痛苦的地方之一。



# 新时间日期API

- 第三次引入的API是成功的，并且java 8中引入的java.time API 已经纠正了过去的缺陷，将来很长一段时间内它都会为我们服务。
- Java 8 吸收了 Joda-Time 的精华，以一个新的开始为 Java 创建优秀的 API。新的 java.time 中包含了所有关于本地日期（LocalDate）、本地时间（LocalTime）、本地日期时间（LocalDateTime）、时区（ZonedDateTime）和持续时间（Duration）的类。历史悠久的 Date 类新增了 toInstant() 方法，用于把 Date 转换成新的表示形式。这些新增的本地化时间日期 API 大大简化了日期时间和本地化的管理。

# 新时间日期API

`java.time` – 包含值对象的基础包

`java.time.chrono` – 提供对不同的日历系统的访问

`java.time.format` – 格式化和解析时间和日期

`java.time.temporal` – 包括底层框架和扩展特性

`java.time.zone` – 包含时区支持的类

说明：大多数开发者只会用到基础包和`format`包，也可能会用到`temporal`包。因此，尽管有68个新的公开类型，大多数开发者，大概将只会用到其中的三分之一。

## 4.1 LocalDate、LocalTime、LocalDateTime

- LocalDate、LocalTime、LocalDateTime 类是其中较重要的几个类，它们的实例是**不可变的对象**，分别表示使用 ISO-8601 日历系统的日期、时间、日期和时间。它们提供了简单的本地日期或时间，并不包含当前的时间信息，也不包含与时区相关的信息。

注：ISO-8601 日历系统是国际标准化组织制定的现代公民的日期和时间的表示法，也就是公历。

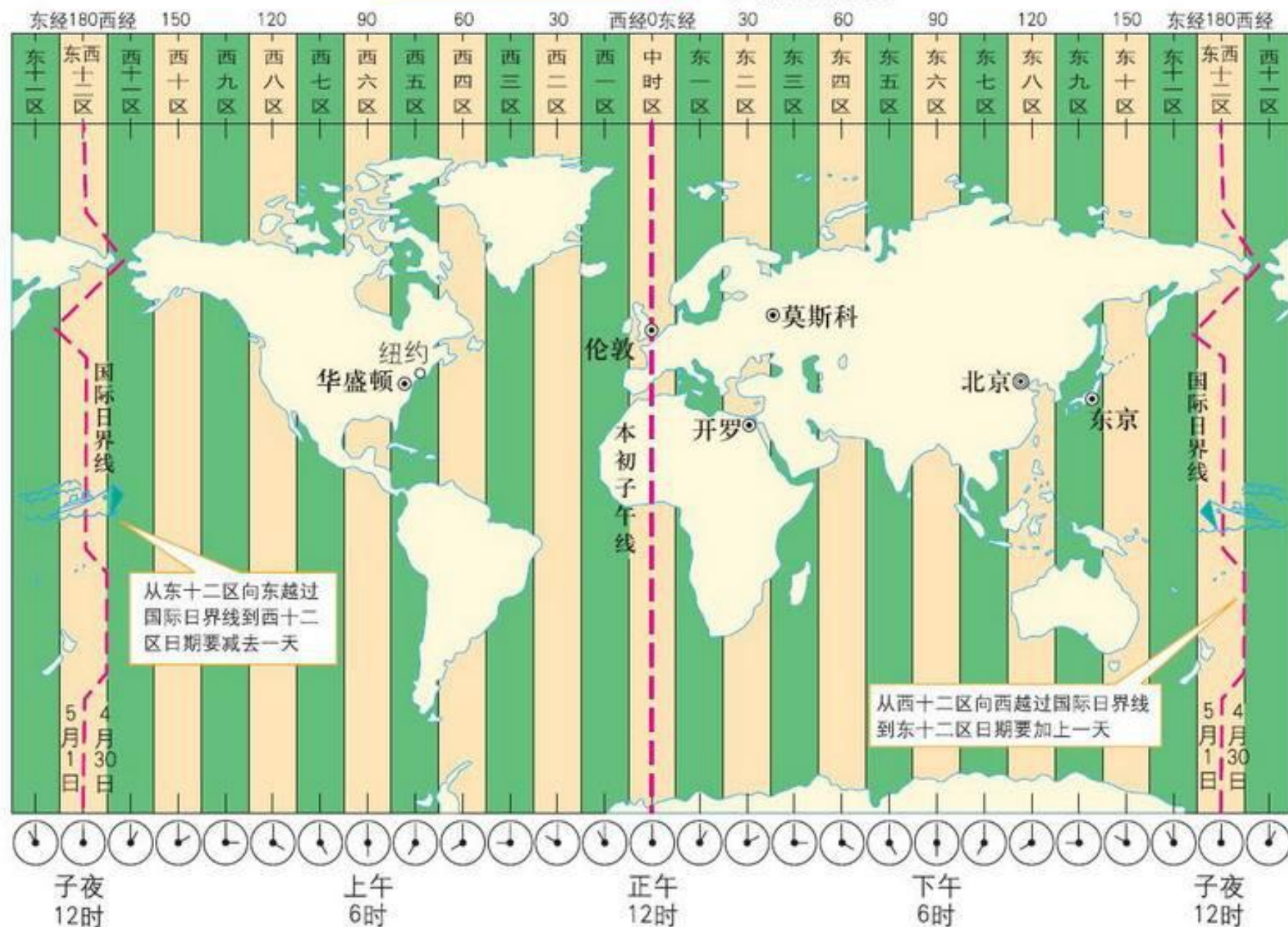
方法	描述
<b>now() / * now(Zoned zone)</b>	静态方法，根据当前时间创建对象/指定时区的对象
<b>of()</b>	静态方法，根据指定日期/时间创建对象
<b>getDayOfMonth()/getDayOfYear()</b>	获得月份天数(1-31) /获得年份天数(1-366)
<b>getDayOfWeek()</b>	获得星期几(返回一个 DayOfWeek 枚举值)
<b>getMonth()</b>	获得月份, 返回一个 Month 枚举值
<b>getMonthValue() / getYear()</b>	获得月份(1-12) /获得年份
<b>getHour()/getMinute()/getSecond()</b>	获得当前对象对应的小时、分钟、秒
<b>withDayOfMonth()/withDayOfYear()/ withMonth()/withYear()</b>	将月份天数、年份天数、月份、年份修改为指定的值并返回新的对象
<b>plusDays(), plusWeeks(), plusMonths(), plusYears(),plusHours()</b>	向当前对象添加几天、几周、几个月、几年、几小时
<b>minusMonths() / minusWeeks()/ minusDays()/minusYears()/minusHours()</b>	从当前对象减去几月、几周、几天、几年、几小时



## 4.2 Instant 时间点

- 在处理时间和日期的时候，我们通常会想到年,月,日,时,分,秒。然而，这只是时间的一个模型，是面向人类的。第二种通用模型是面向机器的，或者说是连续的。在此模型中，时间线中的一个点表示为一个很大的数，这有利于计算机处理。在UNIX中，这个数从1970年开始，以秒为的单位；同样的，在Java中，也是从1970年开始，但以毫秒为单位。
- java.time包通过值类型Instant提供机器视图，不提供处理人类意义上的时间单位。Instant表示时间线上的一点，而不需要任何上下文信息，例如，时区。概念上讲，它只是简单的表示自1970年1月1日0时0分0秒（UTC）开始的秒数。因为java.time包是基于纳秒计算的，所以Instant的精度可以达到纳秒级。 $(1\text{ ns} = 10^{-9}\text{ s})$  1秒 = 1000毫秒 =  $10^6$ 微秒 =  $10^9$ 纳秒

# 地球自转方向



方法	描述
<code>now()</code>	静态方法，返回默认UTC时区的Instant类的对象
<code>ofEpochMilli(long epochMilli)</code>	静态方法，返回在1970-01-01 00:00:00基础上加上指定毫秒数之后的Instant类的对象
<code>atOffset(ZoneOffset offset)</code>	结合即时的偏移来创建一个 OffsetDateTime
<code>toEpochMilli()</code>	返回1970-01-01 00:00:00到当前时间的毫秒数，即为时间戳

时间戳是指格林威治时间1970年01月01日00时00分00秒(北京时间1970年01月01日08时00分00秒)起至现在的总秒数。



## 4.3 格式化与解析日期或时间

`java.time.format.DateTimeFormatter` 类：该类提供了三种格式化方法：

- 预定义的标准格式。如： `ISO_LOCAL_DATE_TIME`; `ISO_LOCAL_DATE`
- 本地化相关的格式。如： `ofLocalizedDate(FormatStyle.FULL)`
- 自定义的格式。如： `ofPattern("yyyy-MM-dd hh:mm:ss E")`

方法	描述
<code>ofPattern(String pattern)</code>	静态方法，返回一个指定字符串格式的 <code>DateTimeFormatter</code>
<code>format(TemporalAccessor t)</code>	格式化一个日期、时间，返回字符串
<code>parse(CharSequence text)</code>	将指定格式的字符序列解析为一个日期、时间



## 5 – Optional类的使用

## 13.4 Optional 类

到目前为止，臭名昭著的空指针异常是导致Java应用程序失败的最常见原因。以前，为了解决空指针异常，Google公司著名的Guava项目引入了Optional类，Guava通过使用检查空值的方式来防止代码污染，它鼓励程序员写更干净的代码。受到Google Guava的启发，Optional类已经成为Java 8类库的一部分。

Optional实际上是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

Optional类的Javadoc描述如下：这是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

## 13.4 Optional 类

`Optional<T>` 类(`java.util.Optional`) 是一个容器类，代表一个值存在或不存在，原来用 `null` 表示一个值不存在，现在 `Optional` 可以更好的表达这个概念。并且可以避免空指针异常。

常用方法：

`Optional.empty()` : 创建一个空的 `Optional` 实例

`Optional.of(T t)` : 创建一个 `Optional` 实例

`Optional.ofNullable(T t)`: 若 `t` 不为 `null`, 创建 `Optional` 实例, 否则创建空实例

`isPresent()` : 判断是否包含值

`T get()`: 如果调用对象包含值，返回该值，否则抛异常

`orElse(T t)` : 如果调用对象包含值，返回该值，否则返回 `t`

`orElseGet(Supplier s)` : 如果调用对象包含值，返回该值，否则返回 `s` 获取的值

`map(Function f)`: 如果有值对其处理，并返回处理后的 `Optional`，否则返回 `Optional.empty()`

`flatMap(Function mapper)`: 与 `map` 类似，要求返回值必须是 `Optional`

## 6-1 Lambda表达式



# 为什么使用 Lambda 表达式

- Lambda 是一个**匿名函数**，我们可以把 Lambda 表达式理解为是**一段可以传递的代码**（将代码像数据一样进行传递）。使用它可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使 Java 的语言表达能力得到了提升。

# Lambda 表达式

- 从匿名类到 Lambda 的转换举例1

```
//匿名内部类
```

```
Runnable r1 = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Hello World!");  
    }  
};
```



```
//Lambda 表达式
```

```
Runnable r1 = () -> System.out.println("Hello Lambda!");
```

# Lambda 表达式

- 从匿名类到 Lambda 的转换举例2

//原来使用匿名内部类作为参数传递

```
TreeSet<String> ts = new TreeSet<>(new Comparator<String>() {  
    @Override  
    public int compare(String o1, String o2) {  
        return Integer.compare(o1.length(), o2.length());  
    }  
});
```



//Lambda 表达式作为参数传递

```
TreeSet<String> ts2 = new TreeSet<>(  
    (o1, o2) -> Integer.compare(o1.length(), o2.length())  
);
```

# Lambda 表达式语法

Lambda 表达式：在Java 8 语言中引入的一种新的语法元素和操作符。这个操作符为 “**->**”，该操作符被称为 **Lambda 操作符**或箭头操作符。它将 Lambda 分为两个部分：

**左侧：** 指定了 Lambda 表达式需要的参数列表

**右侧：** 指定了 **Lambda 体**，是抽象方法的实现逻辑，也即 Lambda 表达式要执行的功能。



# Lambda 表达式语法

语法格式一：无参，无返回值

```
Runnable r1 = () -> {System.out.println("Hello Lambda!");};
```

语法格式二：Lambda 需要一个参数，但是没有返回值。

```
Consumer<String> con = (String str) -> {System.out.println(str);};
```

语法格式三：数据类型可以省略，因为可由编译器推断得出，称为“类型推断”

```
Consumer<String> con = (str) -> {System.out.println(str);};
```

# Lambda 表达式语法

语法格式四：Lambda 若只需要一个参数时，参数的小括号可以省略

```
Consumer<String> con = str -> {System.out.println(str);};
```

语法格式五：Lambda 需要两个或以上的参数，多条执行语句，并且可以有返回值

```
Comparator<Integer> com = (x,y) -> {  
    System.out.println("实现函数式接口方法！");  
    return Integer.compare(x,y);  
};
```

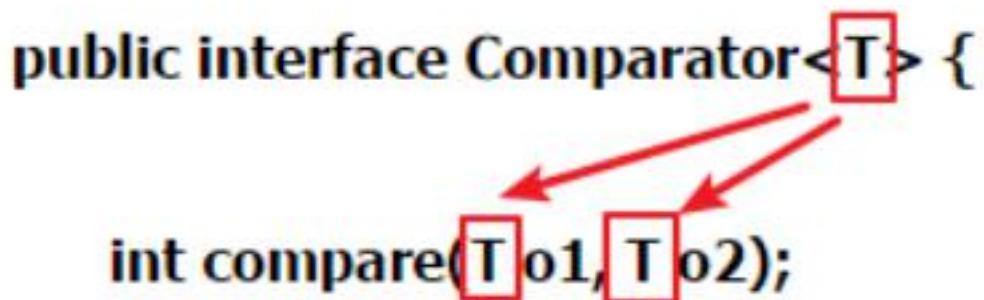
语法格式六：当 Lambda 体只有一条语句时，return 与大括号若有，都可以省略

```
Comparator<Integer> com = (x,y) -> Integer.compare(x, y);
```

# 类型推断

上述 Lambda 表达式中的参数类型都是由编译器推断得出的。Lambda 表达式中无需指定类型，程序依然可以编译，这是因为 javac 根据程序的上下文，在后台推断出了参数的类型。Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的。这就是所谓的“类型推断”。

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```



## 6-2 函数式接口



# 什么是函数式(Functional)接口

- 只包含一个抽象方法的接口，称为函数式接口。
- 你可以通过 Lambda 表达式来创建该接口的对象。（若 Lambda 表达式抛出一个受检异常(即：非运行时异常)，那么该异常需要在目标接口的抽象方法上进行声明）。
- 我们可以在一个接口上使用 **@FunctionalInterface** 注解，这样做可以检查它是否是一个函数式接口。同时 javadoc 也会包含一条声明，说明这个接口是一个函数式接口。
- 在java.util.function包下定义了java 8 的丰富的函数式接口

# 如何理解函数式接口

Java从诞生日起就一直倡导“一切皆对象”，在java里面面向对象(OOP)编程是一切。但是随着python、scala等语言的兴起和新技术的挑战，java不得不做出调整以便支持更加广泛的技术要求，也即java不但可以支持OOP还可以支持OOF（面向函数编程）

在函数式编程语言当中，函数被当做一等公民对待。在将函数作为一等公民的编程语言中，Lambda表达式的类型是函数。但是在Java8中，有所不同。在Java8中，Lambda表达式是对象，而不是函数，它们必须依附于一类特别的对象类型——函数式接口。

简单的说，在Java8中，Lambda表达式就是一个函数式接口的实例。这就是Lambda表达式和函数式接口的关系。也就是说，只要一个对象是函数式接口的实例，那么该对象就可以用Lambda表达式来表示。所以以前用匿名内部类表示的现在都可以用Lambda表达式来写。

## 函数式接口举例

```
@FunctionalInterface
```

```
public interface Runnable {
```

```
    /**
```

```
     * When an object implementing interface Runnable is
     * to create a thread, starting the thread causes the object's
     * run method to be called in that separately executi
     * thread.
```

```
     * <p>
```

```
     * The general contract of the method run is that it
     * take any action whatsoever.
```

```
     *
```

```
     * @see      java.lang.Thread#run()
```

```
     */
```

```
    public abstract void run();
```

```
}
```



## 自定义函数式接口

```
@FunctionalInterface↵  
public interface MyNumber{↵  
    public double getValue();↵  
}↵
```

函数式接口中使用泛型：

```
@FunctionalInterface↵  
public interface MyFunc<T>{↵  
    public T getValue(T t);↵  
}↵
```



## 作为参数传递 Lambda 表达式

```
public String toUpperString(MyFunc<String> mf, String str){  
    return mf.getValue(str);  
}
```

作为参数传递 Lambda 表达式:

```
String newStr = toUpperString(  
    (str) -> str.toUpperCase(), "abcdef");  
System.out.println(newStr);
```

作为参数传递 Lambda 表达式: 为了将 Lambda 表达式作为参数传递, 接收 Lambda 表达式的参数类型必须是与该 Lambda 表达式兼容的函数式接口的类型。

# Java 内置四大核心函数式接口

函数式接口	参数类型	返回类型	用途
<b>Consumer&lt;T&gt;</b> 消费型接口	T	void	对类型为T的对象应用操作，包含方法： <b>void accept(T t)</b>
<b>Supplier&lt;T&gt;</b> 供给型接口	无	T	返回类型为T的对象，包含方法： <b>T get()</b>
<b>Function&lt;T, R&gt;</b> 函数型接口	T	R	对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法： <b>R apply(T t)</b>
<b>Predicate&lt;T&gt;</b> 断定型接口	T	boolean	确定类型为T的对象是否满足某约束，并返回boolean 值。包含方法 <b>boolean test(T t)</b>

## 其他接口

函数式接口	参数类型	返回类型	用途
<code>BiFunction&lt;T, U, R&gt;</code>	<code>T, U</code>	<code>R</code>	对类型为 <code>T, U</code> 参数应用操作，返回 <code>R</code> 类型的结果。包含方法为 <code>R apply(T t, U u);</code>
<code>UnaryOperator&lt;T&gt;</code> ( <code>Function</code> 子接口)	<code>T</code>	<code>T</code>	对类型为 <code>T</code> 的对象进行一元运算，并返回 <code>T</code> 类型的结果。包含方法为 <code>T apply(T t);</code>
<code>BinaryOperator&lt;T&gt;</code> ( <code>BiFunction</code> 子接口)	<code>T, T</code>	<code>T</code>	对类型为 <code>T</code> 的对象进行二元运算，并返回 <code>T</code> 类型的结果。包含方法为 <code>T apply(T t1, T t2);</code>
<code>BiConsumer&lt;T, U&gt;</code>	<code>T, U</code>	<code>void</code>	对类型为 <code>T, U</code> 参数应用操作。包含方法为 <code>void accept(T t, U u)</code>
<code>BiPredicate&lt;T, U&gt;</code>	<code>T, U</code>	<code>boolean</code>	包含方法为 <code>boolean test(T t, U u)</code>
<code>ToIntFunction&lt;T&gt;</code> <code>ToLongFunction&lt;T&gt;</code> <code>ToDoubleFunction&lt;T&gt;</code>	<code>T</code>	<code>int</code> <code>long</code> <code>double</code>	分别计算 <code>int</code> 、 <code>long</code> 、 <code>double</code> 、值的函数
<code>IntFunction&lt;R&gt;</code> <code>LongFunction&lt;R&gt;</code> <code>DoubleFunction&lt;R&gt;</code>	<code>int</code> <code>long</code> <code>double</code>	<code>R</code>	参数分别为 <code>int</code> 、 <code>long</code> 、 <code>double</code> 类型的函数

## 6-3 方法引用与构造器引用



# 方法引用(Method References)

- 当要传递给Lambda体的操作，已经有实现的方法了，可以使用方法引用！
- 方法引用就是Lambda表达式，就是函数式接口的一个实例，通过方法的名字来指向一个方法，可以认为是Lambda表达式的一个语法糖。
- 要求：实现抽象方法的参数列表和返回值类型，必须与方法引用的方法的参数列表和返回值类型保持一致！
- 方法引用：使用操作符 “::” 将类(或对象) 与 方法名分隔开来。
- 如下三种主要使用情况：
  - 对象::实例方法名
  - 类::静态方法名
  - 类::实例方法名

## 方法引用

例如：

```
Consumer<String> con = (x) -> System.out.println(x);
```

等同于：

```
Consumer<String> con2 = System.out :: println;
```

例如：

```
Comparator<Integer> com = (x,y) -> Integer.compare(x, y);
```

等同于：

```
Comparator<Integer> com1 = Integer::compare;
```

```
int value = com.compare(12, 32);
```

## 方法引用

例如：

```
BiPredicate<String, String> bp = (x,y) -> x.equals(y);
```

等同于：

```
BiPredicate<String,String> bp1 = String::equals;  
boolean flag = bp1.test("hello", "hi");
```

注意：当函数式接口方法的第一个参数是需要引用方法的调用者，并且第二个参数是需要引用方法的参数(或无参数)时：**ClassName::methodName**



# 构造器引用

格式: **ClassName::new**

与函数式接口相结合, 自动与函数式接口中方法兼容。  
可以把构造器引用赋值给定义的方法, 要求构造器参数列表要与接口中抽象方法的参数列表一致! 且方法的返回值即为构造器对应类的对象。

例如:

```
Function<Integer, MyClass> fun = (n) -> new MyClass(n);
```

等同于:

```
Function<Integer, MyClass> fun = MyClass::new;
```



## 数组引用

格式: **type[] :: new**

例如:

```
Function<Integer, Integer[]> fun = (n) -> new Integer[n];
```

等同于:

```
Function<Integer, Integer[]> fun = Integer[]::new;
```

## 7 - 强大的Stream API

# Stream API说明

Java8中有两大最为重要的改变。第一个是 **Lambda 表达式**；另外一个则是 **Stream API**。

**Stream API ( java.util.stream)** 把真正的函数式编程风格引入到Java中。这是目前为止对Java类库最好的补充，因为Stream API可以极大提供Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

Stream 是 Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用

**Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。**

也可以使用 Stream API 来并行执行操作。简言之，Stream API 提供了一种高效且易于使用的处理数据的方式。

# 为什么要使用Stream API

实际开发中，项目中多数数据源都来自于Mysql，Oracle等。但现在数据源可以更多了，有MongoDB，Redis等，而这些NoSQL的数据就需要java层面去处理。



# 什么是 Stream

## Stream到底是什么呢？

是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。

“集合讲的是数据，Stream讲的是计算！”

### 注意：

- ①Stream 自己不会存储元素。
- ②Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。
- ③Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

# Stream 的操作三个步骤

## ● 1- 创建 Stream

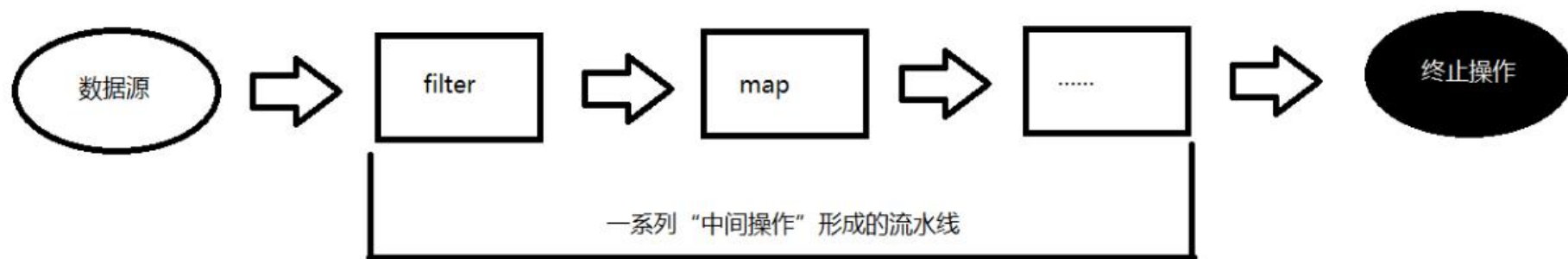
一个数据源（如：集合、数组），获取一个流

## ● 2- 中间操作

一个中间操作链，对数据源的数据进行处理

## ● 3- 终止操作(终端操作)

一旦执行终止操作，就执行中间操作链，并产生结果。之后，不会再被使用



## 创建 Stream方式一：通过集合

Java8 中的 Collection 接口被扩展，提供了两个获取流的方法：

- **default Stream<E> stream()** : 返回一个顺序流
- **default Stream<E> parallelStream()** : 返回一个并行流

## 创建 Stream 方式二：通过数组

Java8 中的 Arrays 的静态方法 `stream()` 可以获取数组流：

- **`static <T> Stream<T> stream(T[] array)`**: 返回一个流

重载形式，能够处理对应基本类型的数组：

- `public static IntStream stream(int[] array)`
- `public static LongStream stream(long[] array)`
- `public static DoubleStream stream(double[] array)`



## 创建 Stream方式三：通过Stream的of()

可以调用Stream类静态方法 `of()`, 通过显示值创建一个流。它可以接收任意数量的参数。

- `public static<T> Stream<T> of(T... values)` : 返回一个流

## 创建 Stream 方式四：创建无限流

可以使用静态方法 `Stream.iterate()` 和 `Stream.generate()`, 创建无限流。

- 迭代

```
public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
```

- 生成

```
public static<T> Stream<T> generate(Supplier<T> s)
```

# Stream 的中间操作

多个**中间操作**可以连接起来形成一个**流水线**，除非流水线上触发终止操作，否则**中间操作不会执行任何的处理！而在终止操作时一次性全部处理，称为“惰性求值”。**

## 1-筛选与切片

方 法	描 述
<b>filter(Predicate p)</b>	接收 Lambda ， 从流中排除某些元素
<b>distinct()</b>	筛选，通过流所生成元素的 hashCode() 和 equals() 去除重复元素
<b>limit(long maxSize)</b>	截断流，使其元素不超过给定数量
<b>skip(long n)</b>	跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 limit(n) 互补

# Stream 的中间操作

## 2-映射

方法	描述
<b>map(Function f)</b>	接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
<b>mapToDouble(ToDoubleFunction f)</b>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 DoubleStream。
<b>mapToInt(ToIntFunction f)</b>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 IntStream。
<b>mapToLong(ToLongFunction f)</b>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 LongStream。
<b>flatMap(Function f)</b>	接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流



# Stream 的中间操作

## 3-排序

方法	描述
<b>sorted()</b>	产生一个新流，其中按自然顺序排序
<b>sorted(Comparator com)</b>	产生一个新流，其中按比较器顺序排序

# Stream 的终止操作

- 终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List、Integer，甚至是 void。
- 流进行了终止操作后，不能再次使用。

## 1-匹配与查找

方法	描述
<b>allMatch(Predicate p)</b>	检查是否匹配所有元素
<b>anyMatch(Predicate p)</b>	检查是否至少匹配一个元素
<b>noneMatch(Predicate p)</b>	检查是否没有匹配所有元素
<b>findFirst()</b>	返回第一个元素
<b>findAny()</b>	返回当前流中的任意元素

## Stream 的终止操作

方法	描述
<code>count()</code>	返回流中元素总数
<code>max(Comparator c)</code>	返回流中最大值
<code>min(Comparator c)</code>	返回流中最小值
<code>forEach(Consumer c)</code>	<b>内部迭代</b> (使用 Collection 接口需要用户去做迭代, 称为 <b>外部迭代</b> 。相反, Stream API 使用内部迭代——它帮你把迭代做了)

# Stream 的终止操作

## 2-归约

方法	描述
<b>reduce(T iden, BinaryOperator b)</b>	可以将流中元素反复结合起来，得到一个值。返回 T
<b>reduce(BinaryOperator b)</b>	可以将流中元素反复结合起来，得到一个值。返回 Optional<T>

备注：map 和 reduce 的连接通常称为 map-reduce 模式，因 Google 用它来进行网络搜索而出名。



# Stream 的终止操作

## 3-收集

方 法	描 述
<code>collect(Collector c)</code>	将流转换为其他形式。接收一个 <code>Collector</code> 接口的实现，用于给 <code>Stream</code> 中元素做汇总的方法

`Collector` 接口中方法的实现决定了如何对流执行收集的操作(如收集到 `List`、`Set`、`Map`)。

另外，`Collectors` 实用类提供了很多静态方法，可以方便地创建常见收集器实例，具体方法与实例如下表：

方法	返回类型	作用
<b>toList</b>	List<T>	把流中元素收集到List
<code>List&lt;Employee&gt; emps= list.stream().collect(Collectors.toList());</code>		
<b>toSet</b>	Set<T>	把流中元素收集到Set
<code>Set&lt;Employee&gt; emps= list.stream().collect(Collectors.toSet());</code>		
<b>toCollection</b>	Collection<T>	把流中元素收集到创建的集合
<code>Collection&lt;Employee&gt; emps =list.stream().collect(Collectors.toCollection(ArrayList::new));</code>		
<b>counting</b>	Long	计算流中元素的个数
<code>long count = list.stream().collect(Collectors.counting());</code>		
<b>summingInt</b>	Integer	对流中元素的整数属性求和
<code>int total=list.stream().collect(Collectors.summingInt(Employee::getSalary));</code>		
<b>averagingInt</b>	Double	计算流中元素Integer属性的平均值
<code>double avg = list.stream().collect(Collectors.averagingInt(Employee::getSalary));</code>		
<b>summarizingInt</b>	IntSummaryStatistics	收集流中Integer属性的统计值。 如：平均值
<code>int SummaryStatisticsiss= list.stream().collect(Collectors.summarizingInt(Employee::getSalary));</code>		





