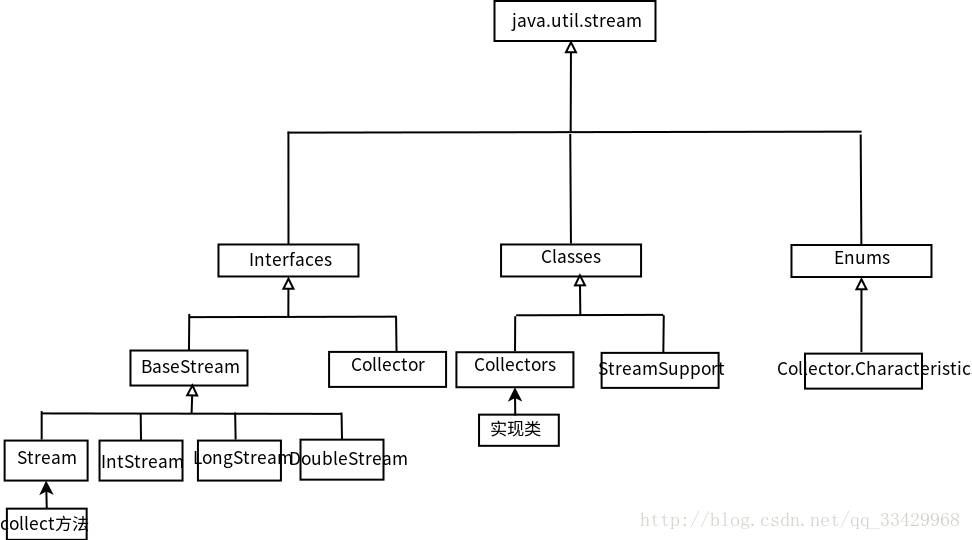
Java Stream API

在Java8中的集合支持stream方法，返回一个流（接口定义在java.util.stream.Stream中），流的定义为从支持数据处理操作的源生成的元素序列：

* 元素序列，和集合一样，流也是提供一个接口来访问特定元素类型的一组有序值。集合的主要目的是以特定的时间/空间复杂度存储和访问元素(如ArrayList和LinkedList)。但是Stream的目的在于表达计算，比如filter、sorted及map等
* 源，流会使用一个提供数据的源，如集合、数组或输入/输出资源。从有序集合生成流时会保留原有顺序。
* 数据处理操作，流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如filter、map、reduce、find、match及sort等。流操作可以顺序执行，也可以并行执行

java.util.stream包的结构图如下所示：



# **Stream的执行**

下面是Stream的执行示例：

*Dish.menu.stream()*

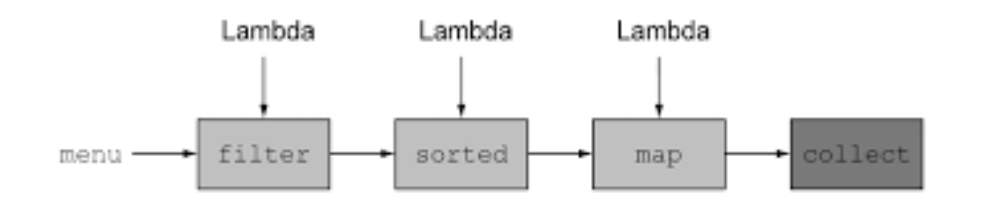
*.filter(d -> d.getCalories() < 400)*

*.sorted(Comparator.comparing(Dish::getCalories))*

*.map(Dish::getName)*

*.collect(toList())*

流执行序列图如下：



1. 创建Stream，最常用的创建方法：

* 通过Stream接口的静态工厂方法，示例如下：

*Stream.of(1,2,3,5)*

*Stream.generate(Math::random);*

*Stream.iterate(1, item -> item + 1)*

*.....*

* 通过Collection接口的默认stream方法，把一个Collection对象转换成Stream，

*default Stream<E> stream() {*

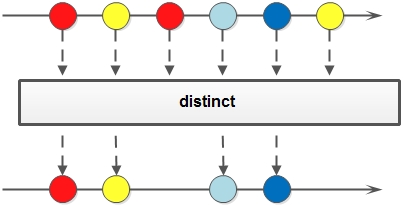
*return StreamSupport.stream(spliterator(), false);*

*}*

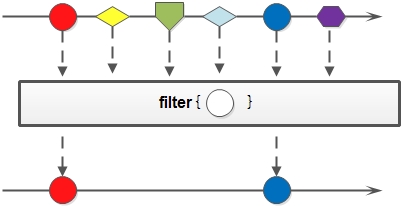
1. 转换Stream

把Steam通过某种行为转换成新的Stream，常用的转换方法：

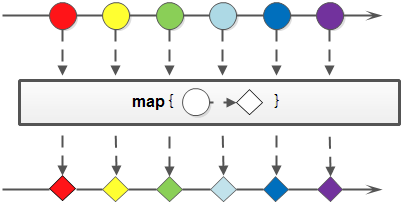
* distinct，对Stream中包含的元素进行去重操作，新生成的Stream中没有重复的元素



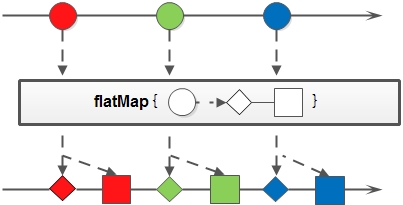
* filter，对Stream中包含的元素使用给定的过滤函数进行过滤操作



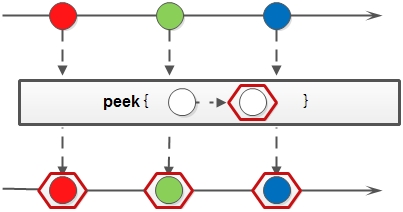
* map,对stream中包含的元素使用给定的转换函数进行转换操作



* flatmap，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中所有元素的新的Stream，同时会提供一个消费函数，新Stream每个元素被消费的时候都会执行给定的消费函数



* limit及skip等，进行截断操作及跳过前N个元素

1. 聚合操作（Reduce）

把Stream中元素组合起来，返回单个结果值，常用的方法有average、sum、min、max及count等都使用reduce方法实现，其使用方法如下：

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

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

*<U> U reduce(U identity,*

*BiFunction<U, ? super T, U> accumulator,*

*BinaryOperator<U> combiner);*

* identy，允许用户提供的一个循环计算初始值
* accumulator，计算的累加器
* combiner，将中间状态的多个结果整合到一起的方法

计算示例：

*List<Integer> numbers = Arrays.asList(3,4,5,1,2);*

*int sum = numbers.stream().reduce(0, (a,b) -> a+b);*

转换及聚合操作都是中间操作，在执行过程中并不产生结果（Lazy）

1. 终端操作

从流的流水线生成结果，其结果是任何不适流的值，比如List、Integer甚至void，使用示例如下：

*numbers.stream().forEach(System.out::println)*

* forEach，消费流中每个元素并对其应用Lambda，这一操作返回void
* count，返回流中的个数
* collect, 把流规约成一个集合，比如List、Map甚至是Integer

# **Stream收集器**

Stream中collect方法，该方法接收一个Collector实例，其是一个可变的汇聚操作，将输入元素累计到一个可变的结果容器中。它会在所有元素都处理完毕后，将累积的结果转换为一个最终的表示。Collector的使用示例：

*Map<Dish.Type,List<Dish>> dishedsByType*

*= Dish.menu.stream().collect(groupingBy(Dish::getType));*

Stream的collect定义如下：

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

在Collector中声明了4个函数，一起协调执行以将元素累计到可结果容器中：

*Supplier<A> supplier(): 创建新的结果容器*

*BiConsumer<A, T> accumulator(): 将元素添加到结果容器*

*BinaryOperator<A> combiner(): 将两个结果容器合并为一个结果容器*

*Function<A, R> finisher(): 对结果容器作相应的变换*

收集器主要提供三大功能：

* 将流元素规约和汇总为一个值
* 元素分组
* 元素分区

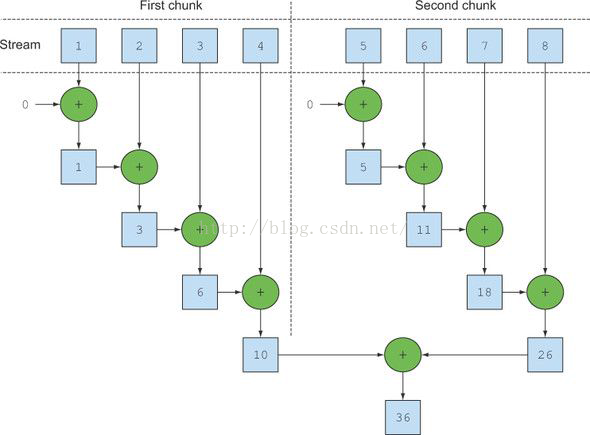
相对应的Collectors中的方法如下：

* maxBy，最大值
* minBy，最小值
* summarizingInt | summarizingLong | summarizingDouble，将对象映射为求和所需的int的统计函数，包括总和、平均值、最大值、最小值
* reducing
* groupingBy，将流中的元素分成不同的组
* partitioningBy，将流中的元素分区，分类函数

# **并行数据处理**

1）并行数据流

Stream接口通过对数据源调用parallelStream方法来把集合转换成并行流，并行流就是一个把内容分成多个数据块，并用不同的线程分别处理每个数块的流。然后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，如下图：



Collection的并行流构建方法：

*default Stream parallelStream() ： 返回并行流*

使用示例如下：

*public static long parallelSum(long n){*

*return Stream.iterate(1L, i -> i +1).*

*limit(n).*

*parallel(). //并行化流*

*reduce(0L,Long::sum);*

*}*

*list.parallelStream()// 获取并行流*

*.sorted()// 按自然排序，即按数字从小到大排序*

*.count();// count()是终止操作，有终止操作才会执行中间操作sorted()*

1. 分支/合并架构

目的是以递归的方式将可以并行的任务拆分成更小的任务，然后将每个子任务的结果合并起来生成整体结果，其是ExecutorService接口的一个实现，把子任务分配给ForkJoinPool线程池中的工作线程，其构造器如下：

*public ForkJoinPool(int parallelism)：创建一个包含parallelism个并行线程的ForkJoinPool*

*public ForkJoinPool() ：以Runtime.getRuntime().availableProcessors()的返回值作为parallelism来创建ForkJoinPool*

创建ForkJoinPool实例后，可以通过ForkJoinPool的submit(ForkJoinTask)或者invoke来执行指定任务，其中ForkJoinTask代表一个可以并行、合并的任务，有两个抽象子类：

* RecursiveTask，代表有返回值的任务
* RecursiveAction，代表没有返回值的任务

其定义如下:

*public class ForkJoinSumCalculator extends RecursiveTask<Long> {*

*public static final long THRESHOLD = 10\_100;*

*private final long[] numbers;*

*private final int start;*

*private final int end;*

*private ForkJoinSumCalculator(long[] numbers, int start,int end) {*

*this.numbers = numbers;*

*this.start = start;*

*this.end = end;*

*}*

*@Override*

*protected Long compute() {*

*int length = end - start;*

*if(length <= THRESHOLD) {return computeSequentially();}*

*ForkJoinSumCalculator leftTask = new ForkJoinSumCalculator(numbers, start, start+length/2);*

*leftTask.fork();*

*ForkJoinSumCalculator rightTask = new ForkJoinSumCalculator(numbers, start + length/2, end);*

*Long rightResult = rightTask.compute();*

*Long leftResult = leftTask.join();*

*return leftResult + rightResult;*

*}*

*.....*

*public static long forkJoinSum(long n) {*

*long[] numbers = LongStream.rangeClosed(1, n).toArray();*

*ForkJoinTask<Long> task = new ForkJoinSumCalculator(numbers);*

*return ParallelStreamsTest.FORK\_JOIN\_POOL.invoke(task); }}*

1. Spliterator

Spliterator是Java 8中加入的集合框架中的另一个接口，其为可分迭代器，和Iterator一样，Spliterator用于遍历数据源中的元素，其接口定义如下：

*public interface Spliterator<T> {*

*boolean tryAdvance(Consumer<? super T> action);*

*Spliterator<T> trySplit();*

*long estimateSize();*

*int characteristics();*

*}*

T是Spliterator遍历的元素的类型。