

# Java + 您

---

# 全栈修养



技术与自我营销两手抓，做一个有追求的程序猿

**Leader.us@Mycat**

# 本节目录

- Lambda
- Java 8 Stream API
- Java高级并发框架

— : Lambda

# 函数式编程

**面向对象编程**是对数据进行抽象，而**函数式编程**是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因此这两种编程方式我们都得学。



这种代码更多地表达了业务逻辑，而不是从机制上如何实现。易读的代码也易于维护、更可靠、更不容易出错。



程序员能编写出更容易阅读的代码

```
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);  
double bill = costBeforeTax.stream().map((cost) -> cost + .12*cost)  
    .reduce((sum, cost) -> sum + cost)  
    .get();  
System.out.println("Total : " + bill);
```



# 编程函数式

如何对“行为”进行抽象和传递？

行为接口

```
public interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

行为实现与传递

```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        ui.dazzle(e.getModifiers());  
    }  
})
```

使用。。。



设计回调有难度，看懂别人的回调.....

## 函数式接口 ( Functional interface )

我们把这些只拥有一个方法的接口称为函数式接口，大多数回调接口都拥有这个特征：比如Runnable接口和Comparator接口。我们并不需要额外的工作来声明一个接口是函数式接口 **编译器会根据接口的结构自行判断**，不过API作者们可以通过@FunctionalInterface注解来显式指定一个接口是函数式接口。

你可以使用 下面语法实现Lambda:

**(params) -> expression**

**(params) -> statement**

**(params) -> { statements }**

(int even, int odd) -> even + odd

# 编程函数式

```
new Comparator<Student>() {  
    public int compare(Student stu1, Student stu2) {  
        return Integer.compare(stu1.age, stu2.age);  
    }  
};
```

```
class Student  
{  
    int age;  
}  
Arrays.sort(students, Comparator<Student> compa);
```

newbility

```
Student[] students = new Student[20];  
Arrays.sort(students, (Student stu1, Student stu2)->{return Integer.compare(stu1.age,stu2.age);});  
Arrays.sort(students, (Student stu1, Student stu2) ->Integer.compare(stu1.age, stu2.age));  
Arrays.sort(students, ( stu1, stu2) -> Integer.compare(stu1.age, stu2.age));
```



# 编程函数式

`delegate(string str){return str.Length;}` 匿名方法

`(string str) => {return str.Length;}` Lambda语句

`(string str) => str.Length` Lambda表达式

`(str) => str.Length` 让编译器推断参数类型

`str => str.Length` 去掉不必要的括号

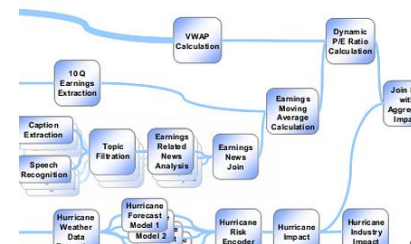


# 编程函数式

++、--都是单目运算符     a+b a-b a\*b都是双目运算符

判断、单目运算、双目运算、转换、遍历（生产者&消费者）

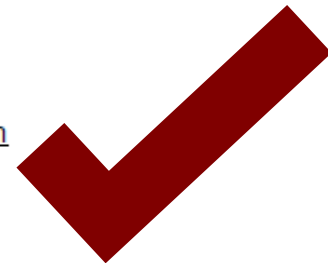
流式计算



Java8增加的函数式接口（Functional interface）java.util.function.\*

- Predicate<T>——接收T对象并返回boolean
- UnaryOperator<T>——接收T对象，返回T对象
- BiFunction<T, U, R>——接收两个对象(T, U)，返回R对象
- Function<T, R>——接收T对象，返回R对象
- Consumer<T>——接收T对象，不返回值
- Supplier<T>——提供T对象（例如工厂），不接收值

- [java.lang.Runnable](#)
- [java.util.concurrent.Callable](#)
- [java.security.PrivilegedAction](#)
- [java.util.Comparator](#)
- [java.io.FileFilter](#)
- [java.beans.PropertyChangeListener](#)



只为int、long和double提供了特化函数式接口：例如IntSupplier和LongBinaryOperator



# 编程函数式

## Java8增加的函数式接口 ( Functional interface )    java.util.function.\*

- 1 BiConsumer - java.util.function
- 1 BiFunction - java.util.function
- 1 BinaryOperator - java.util.function
- 1 BiPredicate - java.util.function
- 1 BooleanSupplier - java.util.function
- 1 Consumer - java.util.function
- 1 DoubleBinaryOperator - java.util.function
- 1 DoubleConsumer - java.util.function
- 1 DoubleFunction - java.util.function
- 1 DoublePredicate - java.util.function
- 1 DoubleSupplier - java.util.function
- 1 DoubleToIntFunction - java.util.function
- 1 DoubleToLongFunction - java.util.function
- 1 DoubleUnaryOperator - java.util.function
- 1 Function - java.util.function
- 1 IntBinaryOperator - java.util.function
- 1 IntConsumer - java.util.function
- 1 IntFunction - java.util.function
- 1 IntPredicate - java.util.function
- 1 IntSupplier - java.util.function
- 1 IntToDoubleFunction - java.util.function
- 1 IntToLongFunction - java.util.function
- 1 IntUnaryOperator - java.util.function

- 1 LongBinaryOperator - java.util.function
- 1 LongConsumer - java.util.function
- 1 LongFunction - java.util.function
- 1 LongPredicate - java.util.function
- 1 LongSupplier - java.util.function
- 1 LongToDoubleFunction - java.util.function
- 1 LongToIntFunction - java.util.function
- 1 LongUnaryOperator - java.util.function
- 1 ObjDoubleConsumer - java.util.function
- 1 ObjIntConsumer - java.util.function
- 1 ObjLongConsumer - java.util.function
- 1 Predicate - java.util.function
- 1 Supplier - java.util.function
- 1 ToDoubleBiFunction - java.util.function
- 1 ToDoubleFunction - java.util.function
- 1 ToIntBiFunction - java.util.function
- 1 ToIntFunction - java.util.function
- 1 ToLongBiFunction - java.util.function
- 1 ToLongFunction - java.util.function
- 1 UnaryOperator - java.util.function

```
package java.util.function;

import java.util.Objects;

/**
 * Represents a function that accepts one argument and produces a result.
 *
 * <p>This is a <a href="package-summary.html">functional interface</a>
 * whose functional method is {@link #apply(Object)}.
 *
 * @param <T> the type of the input to the function
 * @param <R> the type of the result of the function
 *
 * @since 1.8
 */
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
```

# 编程函数式

```
* @since 1.8
*/
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
}
```

Person.java

```
128 public void printWesternName() {
129
130     System.out.println("\nName: " + this.getGivenName() + " " + this.getSurName() + "\n" +
131         "Age: " + this.getAge() + " " + "Gender: " + this.getGender() + "\n" +
132         "Email: " + this.getEmail() + "\n" +
133         "Phone: " + this.getPhone() + "\n" +
134         "Address: " + this.getAddress());
135 }
136
137
138
139 public void printEasternName() {
140
141     System.out.println("\nName: " + this.getSurName() + " " + this.getGivenName() + "\n" +
142         "Age: " + this.getAge() + " " + "Gender: " + this.getGender() + "\n" +
143         "Email: " + this.getEmail() + "\n" +
144         "Phone: " + this.getPhone() + "\n" +
145         "Address: " + this.getAddress());
146 }
```

Person.java

```
123 public String printCustom(Function<Person, String> f) {
124     return f.apply(this);
125 }
126
```

```
// Print Western List
System.out.println("\n==Western List==");
for (Person person:list1) {
    System.out.println(
        person.printCustom(westernStyle)
    );
}
```

// Define Western and Eastern Lambdas

```
Function<Person, String> westernStyle = p -> {
    return "\nName: " + p.getGivenName() + " " + p.getSurName() + "\n" +
        "Age: " + p.getAge() + " " + "Gender: " + p.getGender() + "\n" +
        "Email: " + p.getEmail() + "\n" +
        "Phone: " + p.getPhone() + "\n" +
        "Address: " + p.getAddress();
};
```

```
Function<Person, String> easternStyle = p -> "\nName: " + p.getSurName() + " " +
    p.getGivenName() + "\n" + "Age: " + p.getAge() + " " +
    "Gender: " + p.getGender() + "\n" +
    "Email: " + p.getEmail() + "\n" +
    "Phone: " + p.getPhone() + "\n" +
    "Address: " + p.getAddress();
```

# 编程函数式

方法引用有很多种，它们的语法如下：

静态方法引用：ClassName::methodName

实例上的实例方法引用：instanceReference::methodName

超类上的实例方法引用：super::methodName

类型上的实例方法引用：ClassName::methodName

构造方法引用：Class::new

数组构造方法引用：TypeName[]::new

```
Consumer<Integer> b1 = System::exit;    // void exit(int status)
Consumer<String[]> b2 = Arrays::sort;    // void sort(Object[] a)
```

```
Set<String> knownNames = ...
Predicate<String> isKnown = knownNames::contains;
```

```
SocketImplFactory factory = MySocketImpl::new;
IntFunction<int[]> arrayMaker = int[]::new;
int[] array = arrayMaker.apply(10); // 创建数组 int[10]
```

## 二 : Java 8 Stream API

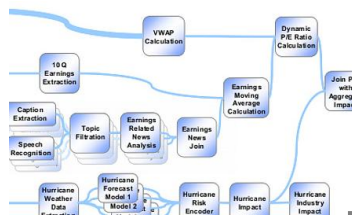
# 面向大数据计算的Stream



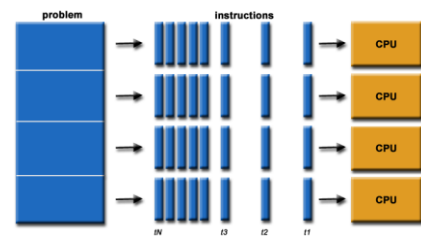
Java 8 中的 Stream 是对集合（Collection）对象功能的增强，它专注于对集合对象进行各种非常便利、高效的聚合操作（aggregate operation），或者大批量数据操作 (bulk data operation)。Stream API 借助于同样新出现的 Lambda 表达式，极大的提高编程效率和程序可读性。同时它提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势，使用 fork/join 并行方式来拆分任务和加速处理过程。通常编写并行代码很难而且容易出错, 但使用 Stream API 无需编写一行多线程的代码，就可以很方便地写出高性能的并发程序。所以说，Java 8 中首次出现的 java.util.stream 是一个函数式语言+多核时代综合影响的产物。



兰亩达



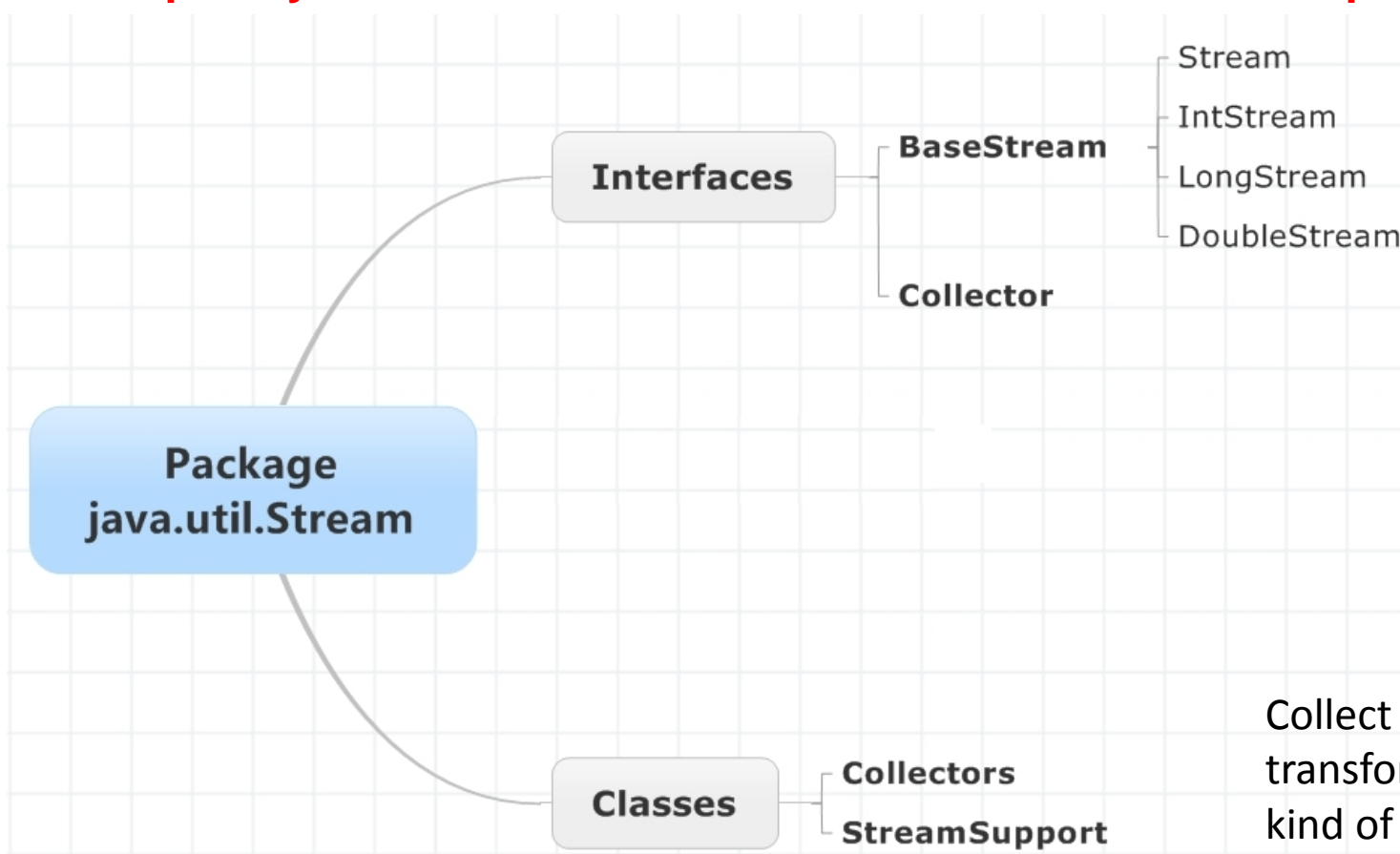
流式计算



并行多核计算

# Java Stream

A collection is an in-memory data structure, which holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection. In contrast, a stream is a conceptually fixed data structure in which elements are computed on demand.



## BaseStream

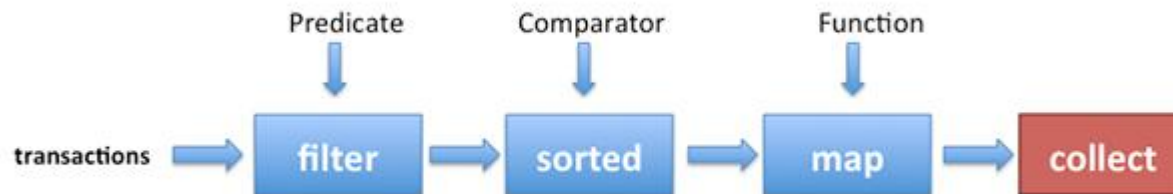
- `Iterator<T> iterator();`
- `Splitter<T> spliterator();`
- `boolean isParallel();`
- `S sequential();`
- `S parallel();`
- `S unordered();`
- `close();`

Collector 是针对 Reduce 操作的抽象接口

Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map.

# Java Stream

```
Stream<T>  
  filter(Predicate<? super T>) : Stream<T>  
  map(Function<? super T, ? extends R> <R> : Stream<R>  
  mapToInt(ToIntFunction<? super T>) : IntStream  
  mapToLong(ToLongFunction<? super T>) : LongStream  
  mapToDouble(ToDoubleFunction<? super T>) : DoubleStream  
  flatMap(Function<? super T, ? extends Stream<? extends R>> <R>  
  flatMapToInt(Function<? super T, ? extends IntStream>) : IntStream  
  flatMapToLong(Function<? super T, ? extends LongStream>) : LongStream  
  flatMapToDouble(Function<? super T, ? extends DoubleStream>) : DoubleStream  
  distinct() : Stream<T>  
  sorted() : Stream<T>  
  sorted(Comparator<? super T>) : Stream<T>  
  peek(Consumer<? super T>) : Stream<T>  
  limit(long) : Stream<T>  
  skip(long) : Stream<T>  
  forEach(Consumer<? super T>) : void  
  forEachOrdered(Consumer<? super T>) : void  
  toArray() : Object[]  
  toArray(IntFunction<A[]>) <A> : A[]  
  reduce(T, BinaryOperator<T>) : T  
  reduce(BinaryOperator<T>) : Optional<T>  
  reduce(U, BiFunction<U, ? super T, U>, BinaryOperator<U>) <U> : U  
  collect(Supplier<R>, BiConsumer<R, ? super T>, BiConsumer<R, R>) : R  
  collect(Collector<? super T, A, R>) <R, A> : R  
  min(Comparator<? super T>) : Optional<T>  
  max(Comparator<? super T>) : Optional<T>  
  count() : long  
  anyMatch(Predicate<? super T>) : boolean  
  allMatch(Predicate<? super T>) : boolean  
  noneMatch(Predicate<? super T>) : boolean  
  findFirst() : Optional<T>  
  findAny() : Optional<T>  
  builder() <T> : Builder<T>  
  empty() <T> : Stream<T>  
  of(T) <T> : Stream<T>  
  of(T...) <T> : Stream<T>  
  iterate(T, UnaryOperator<T>) <T> : Stream<T>
```



## filter方法族

Stream<T> filter(Predicate<? super T> predicate);

Stream<T> sorted(Comparator<? super T> comparator);

boolean anyMatch(Predicate<? super T> predicate);

## map方法族:

<R> Stream<R> map(Function<? super T, ? extends R> mapper);

## Reduce方法族:

Optional<T> min(Comparator<? super T> comparator);

Optional<T> max(Comparator<? super T> comparator);

long count();

OptionalDouble average(); (IntStream等)

int sum(); (IntStream等)

OptionalInt min(); (IntStream等)

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



# Java Stream

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");  
myList.stream().  
filter(s -> s.startsWith("c")).  
map(String::toUpperCase).  
sorted().  
forEach(System.out::println);
```

## java.util.Collection

```
 * @return a sequential {@code Stream} over the elements in this collection  
 * @since 1.8  
 */
```

```
default Stream<E> stream() {  
    return StreamSupport.stream(spliterator(), false);  
}
```

```
/**  
 * Abstract base class for an intermediate pipeline stage or pipeline source  
 * stage implementing whose elements are of type {@code U}.  
 *
```

```
 * @param <P_IN> type of elements in the upstream source  
 * @param <P_OUT> type of elements in produced by this stage  
 *
```

```
 * @since 1.8  
 */
```

```
abstract class ReferencePipeline<P_IN, P_OUT>  
    extends AbstractPipeline<P_IN, P_OUT, Stream<P_OUT>>  
    implements Stream<P_OUT> {
```

C1

C2

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]



# Java Stream

```
Stream.of("a1", "a2", "a3")  
    .findFirst()  
    .ifPresent(System.out::println); // a1
```

```
IntStream.range(1, 4)  
    .forEach(System.out::println);
```

```
Arrays.stream(new int[] {1, 2, 3})  
    .map(n -> 2 * n + 1)  
    .forEach(System.out::println);
```

```
Stream.of("a1", "a2", "a3")  
    .map(s -> s.substring(1))  
    .mapToInt(Integer::parseInt)  
    .max()  
    .ifPresent(System.out::println); // 3
```

```
public static <T> Stream<T> stream(T[] array) {  
    return stream(array, 0, array.length);  
}  
  
public static <T> Stream<T> of(T... values) {  
    return Arrays.stream(values);  
}  
  
/**  
 * Returns a sequential {@code Stream} constructed from the  
 * specified array as its source.  
 *  
 * @param <T> the type of the array elements  
 * @param array the array, assumed to be unmodified during use  
 * @param startInclusive the first index to cover, inclusive  
 * @param endExclusive index immediately past the last index to cover  
 * @return a {@code Stream} for the array range  
 * @throws ArrayIndexOutOfBoundsException if {@code startInclusive} is  
 *     negative, {@code endExclusive} is less than  
 *     {@code startInclusive}, or {@code endExclusive} is greater than  
 *     the array size  
 * @since 1.8  
 */  
public static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive) {  
    return StreamSupport.stream(spliterator(array, startInclusive, endExclusive), false);  
}
```


# Java Stream

The reduction operation combines all elements of the stream into a single result. Java 8 supports three different kind of reduce methods. The reduce method accepts a BinaryOperator accumulator function. That's actually a BiFunction where both operands share the same type

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);
```

```
Person result =
    persons
        .stream()
        .reduce(new Person("", 0), (p1, p2) -> {
            p1.age += p2.age;
            p1.name += p2.name;
            return p1;
        });
```

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) ->
        sum1 + sum2);
```



并行Stream时候的合并

# Java Stream



Streams can be executed in parallel to increase runtime performance on large amount of input elements. Parallel streams use a common ForkJoinPool available via the static `ForkJoinPool.commonPool()` method. The size of the underlying thread-pool uses up to five threads - depending on the amount of available physical CPU cores:

```
List<Person> persons = Arrays.asList(
    new Person("Max", 18),
    new Person("Peter", 23),
    new Person("Pamela", 23),
    new Person("David", 12));
persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s [%s]\n",
                sum, p, Thread.currentThread().getName());
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",
                sum1, sum2, Thread.currentThread().getName());
            return sum1 + sum2;
        });
```

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism()); // 3
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

# Java Stream

intermediate operations will only be executed when a terminal operation is present.

```
Stream.of("d2", "a2", "b1", "b3", "c")  
    .filter(s -> {  
        System.out.println("filter: " + s);  
        return true;  
    });
```

```
.count();
```

```
Supplier<Stream<String>> streamSupplier =  
    () -> Stream.of("d2", "a2", "b1", "b3", "c")  
        .filter(s -> s.startsWith("a"));
```

```
streamSupplier.get().anyMatch(s -> true); // ok  
streamSupplier.get().noneMatch(s -> true); // ok
```

```
Stream.of("d2", "a2", "b1", "b3", "c")  
    .map(s -> {  
        System.out.println("map: " + s);  
        return s.toUpperCase();  
    })  
    .anyMatch(s -> {  
        System.out.println("anyMatch: " + s);  
        return s.startsWith("A");  
    });
```

顺序的重要性

# Java Collector



Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a **List**, **Set** or **Map** or **Single Object**.

```
class Person {  
    String name;  
    int age;  
  
    Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

```
List<Person> persons =  
    Arrays.asList(  
        new Person("Max", 18),  
        new Person("Peter", 23),  
        new Person("Pamela", 23),  
        new Person("David", 12));
```

```
List<Person> filtered =  
    persons  
        .stream()  
        .filter(p -> p.name.startsWith("P"))  
        .collect(Collectors.toList());  
System.out.println(filtered); // [Peter, Pamela]
```

```
Map<Integer, List<Person>> personsByAge = persons  
    .stream()  
    .collect(Collectors.groupingBy(p -> p.age));  
personsByAge  
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));
```

```
Double averageAge = persons  
    .stream()  
    .collect(Collectors.averagingInt(p -> p.age));  
System.out.println(averageAge); // 19.0
```

# Java Collector

```
public interface Collector<T, A, R> {  
    Supplier<A> supplier(); //就是生成容器（A容器）  
    BiConsumer<A, T> accumulator(); //是往A容器添加元素T  
    BinaryOperator<A> combiner(); //合并容器（多个A容器合并）  
    Function<A, R> finisher(); //转换容器输出（可选从A容器改为R容器）  
    Set<Characteristics> characteristics(); （特性说明）  
}
```

specified by **four functions** that work together to **accumulate entries into a mutable result container**, and optionally perform a final transform on the result.

- \* creation of a **new result container** (supplier)
- \* incorporating a **new data element into a result container** (accumulator)
- \* **combining two result containers** into one (combiner)
- \* performing an optional **final transform on the container** (finisher)

# Java Collector



```
List<String> list2 = Arrays.asList("adf", "bcd", "abc", "hgr", "jyt", "edr", "biu");
```

```
String result = list2.stream().collect(StringBuilder::new,
```

```
(res, item) -> {  
    accumulator           res.append(' ').append(item);  
    },  
    supplier
```

```
(res1, res2) -> {
```

**combiner**

```
System.out.printf(Thread.currentThread().getName()+" res1=%s,res2=%s \r\n", res1,  
res2); res1.append(res2); }
```

```
).toString();
```

```
System.out.println("result " + result);
```

**list2.parallelStream()**

# 谢谢观看

Leader全栈Java报名群QQ 332702697