[Java8 Lambda表达式教程](http://blog.csdn.net/ioriogami/article/details/12782141)

**1. 什么是λ表达式**

λ表达式本质上是一个匿名方法。让我们来看下面这个例子：

    public int add(int x, int y) {  
        return x + y;  
    }

转成λ表达式后是这个样子：  
      
    (int x, int y) -> x + y;

参数类型也可以省略，**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/ioriogami/article/details/_blank)**编译器会根据上下文推断出来：

    (x, y) -> x + y; //返回两数之和  
   
或者

    (x, y) -> { return x + y; } //显式指明返回值

可见λ表达式有三部分组成：参数列表，箭头（->），以及一个表达式或语句块。

下面这个例子里的λ表达式没有参数，也没有返回值（相当于一个方法接受0个参数，返回void，其实就是Runnable里run方法的一个实现）：

    () -> { System.out.println("Hello Lambda!"); }

如果只有一个参数且可以被Java推断出类型，那么参数列表的括号也可以省略：

    c -> { return c.size(); }

**2. λ表达式的类型（它是Object吗？）**

λ表达式可以被当做是一个Object（注意措辞）。λ表达式的类型，叫做“目标类型（target type）”。λ表达式的目标类型是“函数接口（functional interface）”，这是Java8新引入的概念。它的定义是：一个接口，如果只有一个显式声明的抽象方法，那么它就是一个函数接口。一般用@FunctionalInterface标注出来（也可以不标）。举例如下：

    @FunctionalInterface  
    public interface Runnable { void run(); }  
      
    public interface Callable<V> { V call() throws Exception; }  
      
    public interface ActionListener { void actionPerformed(ActionEvent e); }  
      
    public interface Comparator<T> { int compare(T o1, T o2); boolean equals(Object obj); }

注意最后这个Comparator接口。它里面声明了两个方法，貌似不符合函数接口的定义，但它的确是函数接口。这是因为equals方法是Object的，所有的接口都会声明Object的public方法——虽然大多是隐式的。所以，Comparator显式的声明了equals不影响它依然是个函数接口。

你可以用一个λ表达式为一个函数接口赋值：  
   
    Runnable r1 = () -> {System.out.println("Hello Lambda!");};  
      
然后再赋值给一个Object：

    Object obj = r1;  
      
但却不能这样干：

    Object obj = () -> {System.out.println("Hello Lambda!");}; // ERROR! Object is not a functional interface!

必须显式的转型成一个函数接口才可以：

    Object o = (Runnable) () -> { System.out.println("hi"); }; // correct  
      
一个λ表达式只有在转型成一个函数接口后才能被当做Object使用。所以下面这句也不能编译：

    System.out.println( () -> {} ); //错误! 目标类型不明  
      
必须先转型:

    System.out.println( (Runnable)() -> {} ); // 正确

假设你自己写了一个函数接口，长的跟Runnable一模一样：

    @FunctionalInterface  
    public interface MyRunnable {  
        public void run();  
    }  
      
那么

    Runnable r1 =    () -> {System.out.println("Hello Lambda!");};  
    MyRunnable2 r2 = () -> {System.out.println("Hello Lambda!");};

都是正确的写法。这说明一个λ表达式可以有多个目标类型（函数接口），只要函数匹配成功即可。  
但需注意一个λ表达式必须至少有一个目标类型。

JDK预定义了很多函数接口以避免用户重复定义。最典型的是Function：

    @FunctionalInterface  
    public interface Function<T, R> {    
        R apply(T t);  
    }

这个接口代表一个函数，接受一个T类型的参数，并返回一个R类型的返回值。

另一个预定义函数接口叫做Consumer，跟Function的唯一不同是它没有返回值。

    @FunctionalInterface  
    public interface Consumer<T> {  
        void accept(T t);  
    }

还有一个Predicate，用来判断某项条件是否满足。经常用来进行筛滤操作：  
      
    @FunctionalInterface  
    public interface Predicate<T> {  
        boolean test(T t);  
    }  
      
综上所述，一个λ表达式其实就是定义了一个匿名方法，只不过这个方法必须符合至少一个函数接口。  
          
**3. λ表达式的使用**

**3.1 λ表达式用在何处**

λ表达式主要用于替换以前广泛使用的内部匿名类，各种回调，比如事件响应器、传入Thread类的Runnable等。看下面的例子：

    Thread oldSchool = new Thread( new Runnable () {  
        @Override  
        public void run() {  
            System.out.println("This is from an anonymous class.");  
        }  
    } );  
      
    Thread gaoDuanDaQiShangDangCi = new Thread( () -> {  
        System.out.println("This is from an anonymous method (lambda exp).");  
    } );

注意第二个线程里的λ表达式，你并不需要显式地把它转成一个Runnable，因为Java能根据上下文自动推断出来：一个Thread的构造函数接受一个Runnable参数，而传入的λ表达式正好符合其run()函数，所以Java编译器推断它为Runnable。

从形式上看，λ表达式只是为你节省了几行代码。但将λ表达式引入Java的动机并不仅仅为此。Java8有一个短期目标和一个长期目标。短期目标是：配合“集合类批处理操作”的内部迭代和并行处理（下面将要讲到）；长期目标是将Java向函数式编程语言这个方向引导（并不是要完全变成一门函数式编程语言，只是让它有更多的函数式编程语言的特性），也正是由于这个原因，**[Oracle](http://lib.csdn.net/base/oracle" \o "Oracle知识库" \t "http://blog.csdn.net/ioriogami/article/details/_blank)**并没有简单地使用内部类去实现λ表达式，而是使用了一种更动态、更灵活、易于将来扩展和改变的策略（invokedynamic）。

**3.2 λ表达式与集合类批处理操作（或者叫块操作）**

上文提到了集合类的批处理操作。这是Java8的另一个重要特性，它与λ表达式的配合使用乃是Java8的最主要特性。集合类的批处理操作API的目的是实现集合类的“内部迭代”，并期望充分利用现代多核CPU进行并行计算。  
Java8之前集合类的迭代（Iteration）都是外部的，即客户代码。而内部迭代意味着改由Java类库来进行迭代，而不是客户代码。例如：

    for(Object o: list) { // 外部迭代  
        System.out.println(o);  
    }

可以写成：

    list.forEach(o -> {System.out.println(o);}); //forEach函数实现内部迭代

集合类（包括List）现在都有一个forEach方法，对元素进行迭代（遍历），所以我们不需要再写for循环了。forEach方法接受一个函数接口Consumer做参数，所以可以使用λ表达式。

这种内部迭代方法广泛存在于各种语言，如C++的STL**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/ioriogami/article/details/_blank)**库、**[Python](http://lib.csdn.net/base/python" \o "Python知识库" \t "http://blog.csdn.net/ioriogami/article/details/_blank)**、ruby、**[Scala](http://lib.csdn.net/base/scala" \o "Scala知识库" \t "http://blog.csdn.net/ioriogami/article/details/_blank)**等。

Java8为集合类引入了另一个重要概念：流（stream）。一个流通常以一个集合类实例为其数据源，然后在其上定义各种操作。流的API设计使用了管道（pipelines）模式。对流的一次操作会返回另一个流。如同IO的API或者StringBuffer的append方法那样，从而多个不同的操作可以在一个语句里串起来。看下面的例子：

    List<Shape> shapes = ...  
    shapes.stream()  
      .filter(s -> s.getColor() == BLUE)  
      .forEach(s -> s.setColor(RED));

首先调用stream方法，以集合类对象shapes里面的元素为数据源，生成一个流。然后在这个流上调用filter方法，挑出蓝色的，返回另一个流。最后调用forEach方法将这些蓝色的物体喷成红色。（forEach方法不再返回流，而是一个终端方法，类似于StringBuffer在调用若干append之后的那个toString）

filter方法的参数是Predicate类型，forEach方法的参数是Consumer类型，它们都是函数接口，所以可以使用λ表达式。

还有一个方法叫parallelStream()，顾名思义它和stream()一样，只不过指明要并行处理，以期充分利用现代CPU的多核特性。

    shapes.parallelStream(); // 或shapes.stream().parallel()

来看更多的例子。下面是典型的**[大数据](http://lib.csdn.net/base/spark" \o "Apache Spark知识库" \t "http://blog.csdn.net/ioriogami/article/details/_blank)**处理方法，Filter-Map-Reduce：

    //给出一个String类型的数组，找出其中所有不重复的素数  
    public void distinctPrimary(String... numbers) {  
        List<String> l = Arrays.asList(numbers);  
        List<Integer> r = l.stream()  
                .map(e -> new Integer(e))  
                .filter(e -> Primes.isPrime(e))  
                .distinct()  
                .collect(Collectors.toList());  
        System.out.println("distinctPrimary result is: " + r);  
    }

第一步：传入一系列String（假设都是合法的数字），转成一个List，然后调用stream()方法生成流。

第二步：调用流的map方法把每个元素由String转成Integer，得到一个新的流。map方法接受一个Function类型的参数，上面介绍了，Function是个函数接口，所以这里用λ表达式。

第三步：调用流的filter方法，过滤那些不是素数的数字，并得到一个新流。filter方法接受一个Predicate类型的参数，上面介绍了，Predicate是个函数接口，所以这里用λ表达式。

第四步：调用流的distinct方法，去掉重复，并得到一个新流。这本质上是另一个filter操作。

第五步：用collect方法将最终结果收集到一个List里面去。collect方法接受一个Collector类型的参数，这个参数指明如何收集最终结果。在这个例子中，结果简单地收集到一个List中。我们也可以用Collectors.toMap(e->e, e->e)把结果收集到一个Map中，它的意思是：把结果收到一个Map，用这些素数自身既作为键又作为值。toMap方法接受两个Function类型的参数，分别用以生成键和值，Function是个函数接口，所以这里都用λ表达式。

你可能会觉得在这个例子里，List l被迭代了好多次，map，filter，distinct都分别是一次循环，效率会不好。实际并非如此。这些返回另一个Stream的方法都是“懒（lazy）”的，而最后返回最终结果的collect方法则是“急（eager）”的。在遇到eager方法之前，lazy的方法不会执行。

当遇到eager方法时，前面的lazy方法才会被依次执行。而且是管道贯通式执行。这意味着每一个元素依次通过这些管道。例如有个元素“3”，首先它被map成整数型3；然后通过filter，发现是素数，被保留下来；又通过distinct，如果已经有一个3了，那么就直接丢弃，如果还没有则保留。这样，3个操作其实只经过了一次循环。

除collect外其它的eager操作还有forEach，toArray，reduce等。

下面来看一下也许是最常用的收集器方法，groupingBy：

    //给出一个String类型的数组，找出其中各个素数，并统计其出现次数  
    public void primaryOccurrence(String... numbers) {  
        List<String> l = Arrays.asList(numbers);  
        Map<Integer, Integer> r = l.stream()  
            .map(e -> new Integer(e))  
            .filter(e -> Primes.isPrime(e))  
            .collect( Collectors.groupingBy(p->p, Collectors.summingInt(p->1)) );  
        System.out.println("primaryOccurrence result is: " + r);  
    }

注意这一行：

    Collectors.groupingBy(p->p, Collectors.summingInt(p->1))

它的意思是：把结果收集到一个Map中，用统计到的各个素数自身作为键，其出现次数作为值。

下面是一个reduce的例子：

    //给出一个String类型的数组，求其中所有不重复素数的和  
    public void distinctPrimarySum(String... numbers) {  
        List<String> l = Arrays.asList(numbers);  
        int sum = l.stream()  
            .map(e -> new Integer(e))  
            .filter(e -> Primes.isPrime(e))  
            .distinct()  
            .reduce(0, (x,y) -> x+y); // equivalent to .sum()  
        System.out.println("distinctPrimarySum result is: " + sum);  
    }

reduce方法用来产生单一的一个最终结果。  
流有很多预定义的reduce操作，如sum()，max()，min()等。

再举个现实世界里的栗子比如：

    // 统计年龄在25-35岁的男女人数、比例  
    public void boysAndGirls(List<Person> persons) {  
        Map<Integer, Integer> result = persons.parallelStream().filter(p -> p.getAge()>=25 && p.getAge()<=35).  
            collect(  
                Collectors.groupingBy(p->p.getSex(), Collectors.summingInt(p->1))  
        );  
        System.out.print("boysAndGirls result is " + result);  
        System.out.println(", ratio (male : female) is " + (float)result.get(Person.MALE)/result.get(Person.FEMALE));  
    }

**3.3 λ表达式的更多用法**

    // 嵌套的λ表达式  
    Callable<Runnable> c1 = () -> () -> { System.out.println("Nested lambda"); };  
    c1.call().run();

    // 用在条件表达式中  
    Callable<Integer> c2 = true ? (() -> 42) : (() -> 24);  
    System.out.println(c2.call());

    // 定义一个递归函数，注意[须用this限定](http://stackoverflow.com/questions/24509441/why-do-lambdas-in-java-8-disallow-forward-reference-to-member-variables-where-an" \t "http://blog.csdn.net/ioriogami/article/details/_blank)  
    protected UnaryOperator<Integer> factorial = i -> i == 0 ? 1 : i \* this.factorial.apply( i - 1 );  
    ...  
    System.out.println(factorial.apply(3));

在Java中，随声明随调用的方式是不行的，比如下面这样，声明了一个λ表达式(x, y) -> x + y，同时企图通过传入实参(2, 3)来调用它：

    int five = ( (x, y) -> x + y ) (2, 3); // ERROR! try to call a lambda in-place

这在C++中是可以的，但Java中不行。Java的λ表达式只能用作赋值、传参、返回值等。

**4. 其它相关概念**

**4.1 捕获（Capture）**

捕获的概念在于解决在λ表达式中我们可以使用哪些外部变量（即除了它自己的参数和内部定义的本地变量）的问题。

答案是：与内部类非常相似，但有不同点。不同点在于内部类总是持有一个其外部类对象的引用。而λ表达式呢，除非在它内部用到了其外部类（包围类）对象的方法或者成员，否则它就不持有这个对象的引用。

在Java8以前，如果要在内部类访问外部对象的一个本地变量，那么这个变量必须声明为final才行。在Java8中，这种限制被去掉了，代之以一个新的概念，“effectively final”。它的意思是你可以声明为final，也可以不声明final但是按照final来用，也就是一次赋值永不改变。换句话说，保证它加上final前缀后不会出编译错误。

在Java8中，内部类和λ表达式都可以访问effectively final的本地变量。λ表达式的例子如下：

    ...      
    int tmp1 = 1; //包围类的成员变量  
    static int tmp2 = 2; //包围类的静态成员变量  
    public void testCapture() {  
        int tmp3 = 3; //没有声明为final，但是effectively final的本地变量  
        final int tmp4 = 4; //声明为final的本地变量  
        int tmp5 = 5; //普通本地变量  
          
        Function<Integer, Integer> f1 = i -> i + tmp1;  
        Function<Integer, Integer> f2 = i -> i + tmp2;  
        Function<Integer, Integer> f3 = i -> i + tmp3;  
        Function<Integer, Integer> f4 = i -> i + tmp4;  
        Function<Integer, Integer> f5 = i -> {  
            tmp5  += i; // 编译错！对tmp5赋值导致它不是effectively final的  
            return tmp5;  
        };  
        ...  
        tmp5 = 9; // 编译错！对tmp5赋值导致它不是effectively final的  
    }  
    ...

Java要求本地变量final或者effectively final的原因是多线程并发问题。内部类、λ表达式都有可能在不同的线程中执行，允许多个线程同时修改一个本地变量不符合Java的设计理念。

**4.2 方法引用（Method reference）**

任何一个λ表达式都可以代表某个函数接口的唯一方法的匿名描述符。我们也可以使用某个类的某个具体方法来代表这个描述符，叫做方法引用。例如：

    Integer::parseInt //静态方法引用  
    System.out::print //实例方法引用  
    Person::new       //构造器引用

下面是一组例子，教你使用方法引用代替λ表达式：

    //c1 与 c2 是一样的（静态方法引用）  
    Comparator<Integer> c2 = (x, y) -> Integer.compare(x, y);  
    Comparator<Integer> c1 = Integer::compare;  
      
    //下面两句是一样的（实例方法引用1）  
    persons.forEach(e -> System.out.println(e));  
    persons.forEach(System.out::println);  
      
    //下面两句是一样的（实例方法引用2）  
    persons.forEach(person -> person.eat());  
    persons.forEach(Person::eat);  
      
    //下面两句是一样的（构造器引用）  
    strList.stream().map(s -> new Integer(s));  
    strList.stream().map(Integer::new);  
      
使用方法引用，你的程序会变得更短些。现在distinctPrimarySum方法可以改写如下：

    public void distinctPrimarySum(String... numbers) {  
        List<String> l = Arrays.asList(numbers);  
        int sum = l.stream().map(Integer::new).filter(Primes::isPrime).distinct().sum();  
        System.out.println("distinctPrimarySum result is: " + sum);  
    }  
      
还有一些其它的方法引用:

    super::toString //引用某个对象的父类方法  
    String[]::new //引用一个数组的构造器

**4.3 默认方法（Default method）**

Java8中，接口声明里可以有方法实现了，叫做默认方法。在此之前，接口里的方法全部是抽象方法。

    public interface MyInterf {  
      
        String m1();  
          
        default String m2() {  
            return "Hello default method!";  
        }  
          
    }  
      
这实际上混淆了接口和抽象类，但一个类仍然可以实现多个接口，而只能继承一个抽象类。

这么做的原因是：由于Collection库需要为批处理操作添加新的方法，如forEach()，stream()等，但是不能修改现有的Collection接口——如果那样做的话所有的实现类都要进行修改，包括很多客户自制的实现类。所以只好使用这种妥协的办法。

如此一来，我们就面临一种类似多继承的问题。如果类Sub继承了两个接口，Base1和Base2，而这两个接口恰好具有完全相同的两个默认方法，那么就会产生冲突。这时Sub类就必须通过重载来显式指明自己要使用哪一个接口的实现（或者提供自己的实现）：  
      
    public class Sub implements Base1, Base2 {  
      
        public void hello() {  
            Base1.super.hello(); //使用Base1的实现  
        }  
          
    }

除了默认方法，Java8的接口也可以有静态方法的实现：

    public interface MyInterf {  
      
        String m1();  
          
        default String m2() {  
            return "Hello default method!";  
        }  
          
        static String m3() {  
            return "Hello static method in Interface!";  
        }  
          
    }  
      
**4.4 生成器函数（Generator function）**

有时候一个流的数据源不一定是一个已存在的集合对象，也可能是个“生成器函数”。一个生成器函数会产生一系列元素，供给一个流。Stream.generate(Supplier<T> s)就是一个生成器函数。其中参数Supplier是一个函数接口，里面有唯一的抽象方法 <T> get()。

下面这个例子生成并打印5个随机数：

    Stream.generate(Math::random).limit(5).forEach(System.out::println);

注意这个limit(5)，如果没有这个调用，那么这条语句会永远地执行下去。也就是说这个生成器是无穷的。这种调用叫做终结操作，或者短路（short-circuiting）操作。