Java 基础提升篇：Java Lambda 表达式的常见应用场景

Lambda 表达式是 Java 8 引入的新特性，结合 forEach 方法可以更方便地实现遍历。此外，它还可代替 Runnable 类，大大简化了代码的编写。

下面介绍了一些常见的应用场景，在这些场景中适时地使用 Lambda 表达式要比通常的方式来得更加简洁和方便。

列表迭代

对一个列表的每一个元素进行操作，不使用 Lambda 表达式时如下：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
for (int element : numbers) {
    System.out.prinln(element);
}
```

使用 Lambda 表达式：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach(x -> System.out.println(x));
```

如果只需要调用单个函数对列表元素进行处理，那么可以使用更加简洁的方法引用代替 Lambda 表达式：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach(System.out::println);
```

事件监听

不使用 Lambda 表达式：

```
button.addActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent e) {
        //handle the event
    }
});
```

使用 Lambda 表达式，需要编写多条语句时用花括号包围起来：

```
button.addActionListener(e -> {
    //handle the event
});
```

Predicate 接口

java.util.function 包中的 Predicate 接口可以很方便地用于过滤。如果你需要对多个对象进行过滤并执行相同的处理逻辑，那么可以将这些相同的操作封装到 filter 方法中，由调用者提供过滤条件，以便重复使用。

不使用 Predicate 接口，对于每一个对象，都需要编写过滤条件和处理逻辑：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<String> words = Arrays.asList("a", "ab", "abc");
numbers.forEach(x -> {
    if (x % 2 == 0) {
        //process logic
    }
})
words.forEach(x -> {
    if (x.length() > 1) {
        //process logic
    }
})
```

使用 Predicate 接口，将相同的处理逻辑封装到 filter 方法中，重复调用：

```
public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    List<String> words = Arrays.asList("a", "ab", "abc");
    filter(numbers, x -> (int)x % 2 == 0);
    filter(words, x -> ((String)x).length() > 1);
}
public static void filter(List list, Predicate condition) {
    list.forEach(x -> {
        if (condition.test(x)) {
            //process logic
        }
    })
}
```

filter 方法也可写成：

```
public static void filter(List list, Predicate condition) {
```

```
    list.stream().filter(x -> condition.test(x)).forEach(x -> {
        //process logic
    })
}
```

Map 映射

使用 Stream 对象的 map 方法将原来的列表经由 Lambda 表达式映射为另一个列表，并通过 collect 方法转换回 List 类型：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> mapped = numbers.stream().map(x -> x *
2).collect(Collectors.toList());
mapped.forEach(System.out::println);
```

Reduce 聚合

reduce 操作，就是通过二元运算对所有元素进行聚合，最终得到一个结果。例如使用加法对列表进行聚合，就是将列表中所有元素累加，得到总和。

因此，我们可以为 reduce 提供一个接收两个参数的 Lambda 表达式，该表达式就相当于一个二元运算：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream().reduce((x, y) -> x + y).get();
System.out.println(sum);
```

代替 Runnable

以创建线程为例，使用 Runnable 类的代码如下：

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        //to do something
    }
};
Thread t = new Thread(r);
t.start();
```

使用 Lambda 表达式：

```
Runnable r = () -> {
```

```
    //to do something
};
Thread t = new Thread(r);
t.start();
```

或者使用更加紧凑的形式：

```
Thread t = new Thread(() -> {
    //to do something
});
t.start;
```