# 一场 Chat 详解 Java 8 Lambda 表达式及 Stream

## 函数式编程简化你的代码提高代码可读性

在 Java 等面向对象编程语言中，方法属于“二等公民”其并不能通过参数进行传递，只能通过传递一个类实例来调用类中的方法，或者直接在方法中调用其可见范围的方法。这样的编程模式势必会造成大量模版代码的冗余、引入了过多类导致程序结构不清晰可读性较差。

模版代码冗余的例子：

```
public static void setValue(int index_db, String key, String value,
int seconds) {
    try (Jedis jedis = pool.getResource()) {
        jedis.select(index_db);
        jedis.setex(key, seconds, value);
    } catch (Exception e) {
        logger.error(e);
        e.printStackTrace();
    }
}

public static void setHSvalue(int index_db, String key, String value)
{
  try (Jedis jedis = pool.getResource()) {
        jedis.select(index_db);
        jedis.set(key, value);
    } catch (Exception e) {
        logger.error(e);
        e.printStackTrace();
    }
}
```



上面的代码是一段基于 Jedis 进行 Redis 操作的工具类，其中 `pool.getResource()` 是通过 Jedis 连接池获取 Jedis 对象的方法，如果你们有使用过 Redis 也无需担心，这里不需要了解每行代码的含义。我们可以看到， `try (Jedis jedis = pool.getResource())` 以及之后的异常捕获在这两个方法中都存在，而这两个方法仅仅是对 Redis 两个命令的封装， Redis 中的命令有 上百个命令，可以想象，如果将所有命令都进行封装后这个工具类会多么庞大，而这其中冗余的代码占据最大的篇幅。如果不编写这个工具类，或者当遇到需要的命令再对工具类进行补充，这固然会减少我们工具类的代码量，但是不编

写工具类则面临在业务代码中掺入 Jedis 对象获取以及异常捕获的相关代码，而这与我们的业务逻辑并无关联，会是的代码意图变得不那么明显，而随时补充的方案看起来很好却也面临维护上的困难，试想当多人开发时，对代码的统一以及避免冗余方法会是多么痛苦的一件事情，况且就算解决了沟通的问题，最后这个工具类同样会无比庞大及冗余居多。

其实我们仔细分析一下就会发现我们真实的操作其实仅仅是使用 jedis 来执行相应命令。如果我们能将相关处理的方法传入，就能解决代码冗余及易读性的问题，Java8 新增的 Lambda 表达式特性就能实现传入处理方法：

将我们的工具类简化一下：

```
public static <T> T exec(int index_db, Function<Jedis, T> executor) {
    T object = null;
    try (Jedis jedis = pool.getResource()) {
        jedis.select(index_db);
        object = executor.apply(jedis);
    } catch (Exception e) {
        logger.error(e);
        e.printStackTrace();
    }
    return object;
}
```



调用的代码：

```
RedisManager.exec(Constants.REDIS_DB.test, jedis -> {
        String key = "test";
        return jedis.sadd(key, String.valueOf(12));
});
```

这样我们既能减少样板代码，也无需创建过多的方法，而且 Jedis 相关操作的可读性更高，这样就比较优雅的解决了我们的问题。

接下来我们就讲解一下 Lambda 表达式究竟为何物。

# Lambda 表达式

通过前面的基本分析，我们已经了解使用 Lambda 表达式的好处，我们再总结一下：Lambda 可以看做是用一种实现匿名类的更简单、可读性更高的方式。

## 基本语法

（需要传递的接口抽象方法的参数列表）-> { 具体业务实现 }

Lambda 表达式对应的其实是一个 只有一个抽象方法 的接口（在 Java 中这种接口叫做功能接口）的匿名类。我们来看一些示例：

```
Runnable runnable1=()->System.out.println("Runnable 接口 Lambda 方式");
//完整写法： Runnable runnable1=()->{System.out.println("Runnable 接口
Lambda 方式")};
//当‘{}’中语句只有一行时可以省略 ‘{}’

Comparator<Integer> comparator = (int1,int2) -> int1 - int2;
//完整写法 Comparator<Integer> comparator = (Integer int1,Integer int2)
->{return int1 - int2;};
//（）中参数类型可以省略 当需要返回值且只有一个语句时可以省略 return 及‘{}’

Comparable<Integer> comparable = intVal -> 1;
//完整写法 Comparable<Integer> comparable = (Integer  intVal) ->{return
1;}
//当参数列表只有一个参数时，可以忽略‘（）’
```

我们可以看到，其实 Lambda 只关注功能接口中的方法参数列表，所以我们很多同方法参数列表的功能接口就可以相互替换，因此，Java8 在 java.util.function 包下为大家提供了很多可用的功能接口。



## 方法引用

方法引用结合 Lambda 可以引用已存在的方法，省略很多编码，而且可读性更强，它可以自动装配参数与返回值。

我们来通过一个示例来更好地理解一下我们上面提到的自动装配参数与返回值：

```
(String s) -> System.out.println(s)
//等价于 System.out::println
```

当有一个功能接口其抽象方法参数列表为 String s 时，我们使用方法引用，编译器会将 s 传入到 println 中。对比上面两种写法可以看到，代码简洁不少并且可读性也比较好。

## Stream

Stream 是 Java 8 提供的一系列对可迭代元素处理的优化方案，使用 Stream 可以大大减少代码量，提高代码的可读性并且使代码更易并行。

我们以英雄联盟中英雄的筛选为例讲解相关使用。

```
package com.coderknock;

import static com.coderknock.ATK_RANGE.NONE;

/**
 * <p>英雄</p>
 *
 * @author 三产
 * @version 1.0
 * @date 2017-12-11
 * @QQGroup 213732117
 * @website http://www.coderknock.com
 * @copyright Copyright 2017 拿客 coderknock.com  All rights reserved.
 * @since JDK 1.8
 */
public class Hero {
    private int id;
    private String name;
    private ATK_RANGE atk_range=NONE;
    private int hp;
    private int mp;
    private int physical_atk;//物理攻击值
    private int magic_atk;//魔法攻击值
    private int physical_defense;//物理防守值
    //...... 这里省略 getter setter
}
```



首先我们需要根据文件录入英雄信息：

```
安琪拉

3323

490

170

0

83

12.1%

程咬金

3437
```

```
0

161

0

125

17.2%

...省略其他数据
```

上面是我们从网上获取到的数据格式。大家可以在这里先思考下如何用 Java 代码将其转换为 Hero 类的集合。

Stream 解决方案：

```
List<String> heroList =
Files.lines(Paths.get("/home/sanchan/Documents/CoderknockProjects/Lambda/src/main/resources/英雄数据")) //读取数据，并将其转换为字符串集合
.filter(data -> !Strings.isNullOrEmpty(data))//筛选集合中不为空的元素
.collect(toList());//将筛选后的结果转换为集合
//处理数据
for (int i = 0; i <heroList.size() ; i+=8) {
    Hero hero = new Hero();
    int id = (i / 8) + 1;
    System.out.println(id);
    hero.setId(id);
    hero.setName(heroList.get(i));
    hero.setHp(Integer.parseInt(heroList.get(i+1)));
    hero.setMp(Integer.parseInt(heroList.get(i+2)));
    hero.setPhysical_atk(Integer.parseInt(heroList.get(i+3)));
    hero.setMagic_atk(Integer.parseInt(heroList.get(i+4)));
    hero.setPhysical_defense(Integer.parseInt(heroList.get(i+5)));
    heroes.add(hero);
}
```



是不是比你写出的代码少了很多呢？而且，这里还能通过 .parallel() 很方便的实现并行。

本文只是带领大家了解 Lambda 及 Stream 的基本使用方式，篇幅有限，更深入的内容还需要大家多多了解，之后的交流中大家也可以提相关问题或遗漏部分，我们一起探讨。