## 1、基础

## 2、JDK8

JDK8优点：

1) 速度更快：①底层数据结构（HashMap：红黑树； ConcurrentHashMap：CAS+Synchronized，链表+红黑树）；②垃圾回收机制（）；③并行扩展；

2) 代码更少：Lambda 表达式；

3) 强大的 Stream API

4) 便于并行

5) 最大化减少空指针异常 Optional。

### 2.1 Lambda 表达式(☆☆☆☆☆)

#### 2.1.1 简介

Lambda 是一个 匿名函数，我们可以把 Lambda表达式理解为是 一段可以传递的代码（将代码像数据一样进行传递）。可以写出更紧凑、更简洁、更灵活的代码。

#### 2.2.2 表达式

Lambda 表达式的语法格式如下： (parameters) -> expression 或 (parameters) ->{ statements; }

其中，“->”被称为 Lambda 操作符或剪头操作符。它将 Lambda 分为两个部分：

**左侧：需要的所有参数；右侧：指定了 Lambda 体，即 Lambda 表达式要执行的功能。**

以下是lambda表达式的重要特征:

·  可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。

·  可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。

·  可选的大括号：如果主体包含了一个语句，就不需要使用大括号。

·  可选的返回关键字：如果主体包含了一个语句，就不需要 return。

·   Lambda 需要 函数式接口的支持。

·  lambda 表达式的局部变量可以不用声明为 final，但是不能被修改（即隐性的具有final 的语义）。

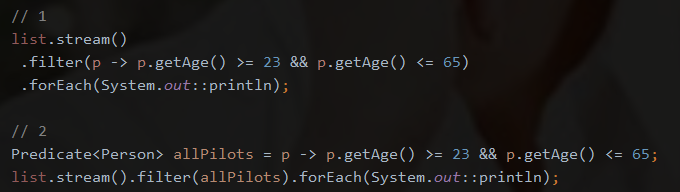
#### 2.2.3 例子

（1）Consumer<String> con = (a) -> System.out.println(a);

（2）Comparator<Person> c1 = (o1, o2) -> Integer.compare(o1.getAge(), o2.getAge());

（3）Collections.sort(list, (a,b)->a.compareTo(b));

（4）员工年龄在23-65的所有员工：



### 2.2 函数式接口

#### 2.2.1 什么是函数式接口

 只包含**一个**抽象方法的**接口**，称为 函数式接口。可以在任意函数式接口上使用 @FunctionalInterface 注解，这样就可以检查它是否是一个函数式接口，同时 javadoc 也会包含一条声明说明这是一个函数式接口。

 你可以通过 Lambda 表达式来创建该接口的对象。（若 Lambda表达式抛出一个受检异常，那么该异常需要在目标接口的抽象方法上进行声明）。

#### 2.2.2 自定义函数式接口

|  |
| --- |
| @FunctionalInterface  public interface LongFun<T, R> {  R getResult (T t, T o);  } |
| public long testLongFun(Long par1, Long par2, LongFun<Long, Long> myFun){  return myFun.getResult(par1, par2);  } |
| Long add = testLongFun(1000L, 2000L, (a, b) -> a + b);  Long mul = testLongFun(1000L, 2000L, (a, b) -> a \* b); |

#### 2.2.3 内置函数式接口java.util.function

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数式接口 | 参数类型 | 返回类型 | 用途 |  |
| Consumer<T> | T | void | void **accept**(T t); | 消费型接口 |
| Supplier<T> | 无 | T | T **get**(); | 供给型接口 |
| Function<T, R> | T | R | R **apply**(T t); | 函数型接口 |
| Predicate<T> | T | boolean | boolean **test**(T t); | 断言型接口 |

#### 2.2.4 其他函数式接口

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数式接口 | 参数类型 | 返回类型 | 用途 |  |
| BiFunction<T, U, R> | T, U | R | R apply(T t, U u); |  |
| UnaryOperator <T> | T | T | T apply(T t); | Function子接口 |
| BinaryOperator<T> | T, T | R | T apply(T t1, T t2); | BiFunction子接口 |
| BiConsumer<T, U> | T, U | void | void accept(T t, U u) |  |
| ToIntFunction<T>  ToLongFunction<T>  ToDoubleFunction<T> | T | int  long  double | int applyAsInt(T value);  long applyAsLong(T value);  double applyAsDouble(T value); |  |
| IntFunction<R>  LongFunction<R>  DoubleFunction<R> | int  long  double | R | R apply(int value);  R apply(long value);  R apply(double value); |  |

### 2.3 方法引用与构造器引用

#### 2.3.1 什么是方法引用

若Lambda体中的内容有实现的方法了，可以使用方法引用！可以理解为方法引用是 Lambda 表达式的另外一种表现形式。（实现抽象方法的参数列表，必须与方法引用方法的参数列表保持一致！）

#### 2.3.2 语法格式

方法引用：使用操作符 “ ::” 将方法名和对象或类的名字分隔开来。

如下三种主要使用情况 ：

 **① 对象 :: 实例方法**

** ② 类 :: 静态方法**

** ③ 类 :: 实例方法**：第一个参数是实例方法调用者，第二个参数是实例方法的参数

注意：Lambda体中调用方法的参数列表与返回值类型，要与函数式接口中抽象方法函数列表和返回值保持一致。例如：Comparator<Integer> c2 = Integer::compare; Comparator的抽象函数和Integer的compare方法参数返回值一致。

#### 2.3.3 实例

(1) 对象 :: 实例方法

**()->p.getAddress();** 🡪 Supplier<String> s = **p::getAddress;**

**(x)->x.equals(a);** 🡪 Predicate<String> p2 = **a::equals;**

(2) 类 :: 静态方法

**(x)-> System.out.println(x);** 🡪 Consumer<String> con1 = System.out::println;

**(x,y) -> Math.pow(x,y);** 🡪 BinaryOperator<Double> f2 = Math::pow;

**(x, y) -> Integer.compare(x, y);** 🡪 Comparator<Integer> c2 = Integer::compare;

(3) 类 :: 实例方法

**(x, y) -> x.equals(y);** 🡪 BiPredicate<String, String> bp2 = String::equals;

#### 2.3.4 构造器引用

(1) 格式：ClassName :: new

(2) 注意：需要调用的构造器的参数列表要与函数式接口中抽象方法的参数列表保持一致。

(3) 实例

①Supplier<People> s1 = People::new; // People 无参构造器

②Function<Integer, People> f1 = People::new; // People 需要 有 只传一个Integer类型的构造器

③BiFunction<String, Integer, People> bf = People::new; // People 全参构造器

#### 2.3.5 数组引用

(1) 格式：Type[] :: new

(2) 实例：Function<Integer, String[]> f2 = String[]::new;

### 2.4 Stream API(☆☆☆☆☆)

#### 2.4.1 什么是Stream

Stream 是 Java8 中处理集合的关键抽象概念。使用Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询，可以执行复杂的查找、过滤和映射数据等操作。也可以使用 Stream API 来并行执行操作。

集合讲的是数据，流讲的是计算！

**注意：**

①Stream 自己不会存储元素。

②Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。

③Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

#### 2.4.2 创建Stream

(1) Collection 获取流

① default Stream<E> **stream()** : 返回一个顺序流

 ② default Stream<E> **parallelStream()** : 返回一个并行流

(2) Arrays创建流：需要一个数组

① static <T> Stream<T> **stream(T[] array)**: 返回一个流

② public static IntStream stream(int[] array)

 ③ public static LongStream stream(long[] array)

 ④ public static DoubleStream stream(double[] array)

(3) Stream.of()

① public static<T> Stream<T> of(T... values) : 返回一个流

(4) Stream.iterate() 和Stream.generate(), 创建无限流

实例：

* Collection：Stream<String> sr = new ArrayList().stream();
* Arrays：Stream<String> sr2 = Arrays.stream(new String[10]);
* Stream：Stream<String> sr3 = Stream.of("aa","bb","cc");
* 迭代：Stream<Integer> stream4 = Stream.iterate(0, (x) -> x+2);
* 生成：Stream.generate(()->Math.random()).forEach(System.out::println);

### 2.4.2 Stream的中间操作

(1) 筛选与切片

① filter(Predicate p p)：接收 Lambda ，从流中排除某些元素。

② distinct()：筛选，通过流所生成元素的 hashCode() 和 equals() 去除重复元素。

③ limit(long maxSize)：截断流，使其元素不超过给定数量。

④ skip(long n)：跳过元素，返回一个扔掉前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。

(2) 映射

① map(Function f)：接收函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。

② flatMap(Function f)：将流中的每个值都换成另一个流，然后把所有流连接成一个流。

区别：map：Stream<Stream<Character>>；flatMap：Stream<Character>

(3) 排序

①sorted()：产生一个新流，其中按自然顺序排序。

②sorted(Comparator comp)：产生一个新流，其中按比较器顺序排序。

### 2.5 接口中的默认方法与静态方法

### 2.6 新时间日期 API

### 2.7 其他新特性

## 3、JDK9

## 4、MySQL

## 5、Druid

## 1 Spring Boot & Spring Cloud

### 1.1 Spring boot

#### 1.1.1什么是springboot

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

①创建独立的spring引用程序 main方法运行；②嵌入的Tomcat 无需部署war文件； ③简化maven配置；④自动配置spring添加对应功能starter自动化配置；

#### 1.1.2 springboot常用的starter有哪些

spring-boot-starter-web 嵌入tomcat和web开发需要servlet与jsp支持

spring-boot-starter-data-jpa 数据库支持

spring-boot-starter-data-redis redis数据库支持

spring-boot-starter-data-solr solr支持

mybatis-spring-boot-starter 第三方的mybatis集成starter

#### 1.1.3 springboot自动配置的原理

在spring程序main方法中 添加@SpringBootApplication或者@EnableAutoConfiguration

会自动去maven中读取每个starter中的spring.factories文件  该文件里配置了所有需要被创建spring容器中的bean。

#### 1.1.4 springboot读取配置文件的方式

 springboot默认读取配置文件为application.properties或者是application.yml

**热部署：**springboot如何添加【修改代码】自动重启功能：添加开发者工具集=====spring-boot-devtools

#### 1.1.5 springboot集成mybatis的过程

①添加mybatis的starter maven依赖

②在mybatis的接口中 添加@Mapper注解

③在application.yml配置数据源信息

#### 1.1.6 Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@**SpringBootApplication**，它也是 Spring Boot 的核心注解，

主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，

如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

#### ****1.1.7 Spring Boot 2.X 有什么新特性？****

**①配置变更：**大量的Servlet专属的server.\* properties被移到了server.servlet下，以前有几个Spring Boot starter是依靠Spring MVC和spring-boot-starter-web传递的。 为了对Spring WebFlux的支持，spring-boot-starter-mustache和spring-boot-starter-thymeleaf不再依赖spring-boot-starter-web。现在你要自己选择并添加spring-boot-starter-web或spring-boot-starter-webflux作为依赖。

**②JDK 版本升级：**Spring Boot 2.0 要求Java 版本必须8以上，支持 JDK 9， Java 6 和 7 不再支持。

**第三方类库升级：**Spring Boot 2.0 建立在 Spring Framework 5 之上。要求最低版本：Tomcat 8.5；Hibernate 5.2； Gradle 3.4，Thymeleaf 3。

**③响应式 Spring 编程支持**

④HTTP/2 支持

⑤配置属性绑定：在 Spring Boot 2.0 中，用于绑定Environment属性的机制@ConfigurationProperties已经完全彻底修改。我们借此机会收紧了松散绑定的规则，并修复了 Spring Boot 1.x 中的许多不一致之处。

⑥更多改进与加强…

#### 1.1.8开启 Spring Boot 特性

1）继承spring-boot-starter-parent项目

2）导入spring-boot-dependencies项目依赖

#### 1.1.9 Spring Boot日志框架

Spring Boot 支持 Java Util Logging, Log4j2, Lockback 作为日志框架，如果你使用 Starters 启动器，Spring Boot 将使用 Logback 作为默认日志框架。

### 1.2 Spring Cloud

#### 1.2.1 什么是微服务

以前的模式是：所有的代码在同一个工程中，部署在同一个服务器中，同一个项目的不同模块不同功能互相抢占资源；

微服务：将工程根据不同的业务规则拆分成微服务，微服务部署在不同的机器上，服务之间进行相互调用。

Java微服务的框架有 dubbo（只能用来做微服务），spring cloud（提供了服务的发现，断路器等）。

#### 1.2.1 Spring boot 和 Spring cloud 关系

Spring Cloud是一个基于Spring Boot实现的云应用开发工具；Spring boot专注于快速、方便集成的单个个体，Spring Cloud是关注全局的服务治理框架；spring boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring boot来实现。

#### 1.2.3 springcloud如何实现服务的注册和发现

服务在发布时 指定对应的服务名（服务名包括了IP地址和端口） 将服务注册到注册中心（eureka或者zookeeper）

这一过程是springcloud自动实现 只需要在main方法添加@EnableDisscoveryClient  同一个服务修改端口就可以启动多个实例

调用方法：传递服务名称通过注册中心获取所有的可用实例 通过负载均衡策略调用（ribbon和feign）对应的服务

#### 1.2.4 ribbon和feign区别

Ribbon添加maven依赖 spring-starter-ribbon 使用@RibbonClient(value="服务名称") 使用RestTemplate调用远程服务对应的方法

feign添加maven依赖 spring-starter-feign 服务提供方提供对外接口 调用方使用 在接口上使用@FeignClient("指定服务名")

Ribbon和Feign都是用于调用其他服务的，不过方式不同。

1.启动类使用的注解不同，Ribbon用的是@RibbonClient，Feign用的是@EnableFeignClients。

2.服务的指定位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。

        3.调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。

        Feign则是在Ribbon的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，

        不需要自己构建http请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。

springcloud断路器的作用

        当一个服务调用另一个服务由于网络原因或者自身原因出现问题时 调用者就会等待被调用者的响应 当更多的服务请求到这些资源时

                导致更多的请求等待 这样就会发生连锁效应（雪崩效应） 断路器就是解决这一问题

                断路器有完全打开状态

                        一定时间内 达到一定的次数无法调用 并且多次检测没有恢复的迹象 断路器完全打开，那么下次请求就不会请求到该服务

                半开

                        短时间内 有恢复迹象 断路器会将部分请求发给该服务 当能正常调用时 断路器关闭

                关闭

                        当服务一直处于正常状态 能正常调用 断路器关闭

什么是服务熔断？什么是服务降级？

熔断机制是应对雪崩效应的一种微服务链路保护机制。当某个微服务不可用或者响应时间太长时，会进行服务降级，进而熔断该节点微服务的调用，快速返回“错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现，Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内调用20次，如果失败，就会启动熔断机制。

服务降级，一般是从整体负荷考虑。就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值。这样做，虽然水平下降，但好歹可用，比直接挂掉强。

Eureka和zookeeper都可以提供服务注册与发现的功能，请说说两个的区别？

Zookeeper保证了CP（C：一致性，P：分区容错性），Eureka保证了AP（A：高可用）

1.当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的信息，但不能容忍直接down掉不可用。也就是说，服务注册功能对高可用性要求比较高，但zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新选leader。问题在于，选取leader时间过长，30 ~ 120s，且选取期间zk集群都不可用，这样就会导致选取期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够恢复，但是漫长的选取时间导致的注册长期不可用是不能容忍的。

2.Eureka保证了可用性，Eureka各个节点是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点仍然可以提供注册和查询服务。而Eureka的客户端向某个Eureka注册或发现时发生连接失败，则会自动切换到其他节点，只要有一台Eureka还在，就能保证注册服务可用，只是查到的信息可能不是最新的。除此之外，Eureka还有自我保护机制，如果在15分钟内超过85%的节点没有正常的心跳，那么Eureka就认为客户端与注册中心发生了网络故障，此时会出现以下几种情况：

①、Eureka不在从注册列表中移除因为长时间没有收到心跳而应该过期的服务。

②、Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其他节点上（即保证当前节点仍然可用）

③、当网络稳定时，当前实例新的注册信息会被同步到其他节点。

因此，Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像Zookeeper那样使整个微服务瘫痪。

---------------------

作者：分香卖履

来源：CSDN

原文：https://blog.csdn.net/qq\_38891512/article/details/82083389

版权声明：本文为博主原创文章，转载请附上博文链接！