# 1、基础

## 1. 抽象类和接口的异同？

答：共同点：都不能实例化，以多态的方式去实现。不同点：单继承，多实现。

声明方式：抽象类是用class声明，接口是用 interface声明。

内部结构：抽象类有：可有可无抽象方法、属性、构造器、代码块、内部类。1.7中的接口只有常量和抽象方法；1.8中在此基础上还可以声明默认方法和静态方法，1.9中还可以将方法的访问权限修饰符声明为 private 。

## 2. String、StringBuffer、StringBuilder

答：JDK8及其以前，用 char[] 存储， JDK9 底层使用 char[] 存储。

String：不可变字符序列，private final char value[];

StringBuffer：可变字符序列，线程安全的，效率低。方法都有 synchronized 修饰。char[] value;

StringBuilder：可变字符序列，线程不安全的，效率高。JDK5才出现。char[] value;

## 3. HTTP/1.1和HTTP/2 区别

HTTP/1.1和HTTP/2的主要区别是如何在客户端和服务器之间构建和传输数据。

HTTP/1.1 依赖于请求/响应周期。

HTTP/2 允许服务器“push”数据：它可以发送比客户端请求更多的数据。

# 2、JDK8

## 2.1 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：Parallel

5) 最大化减少空指针异常 Optional。

6) JVM: JDK8的HotSpot中，把永久代从Java堆中移除了，并把类的元数据直接保存在本地内存区域。

## 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）。

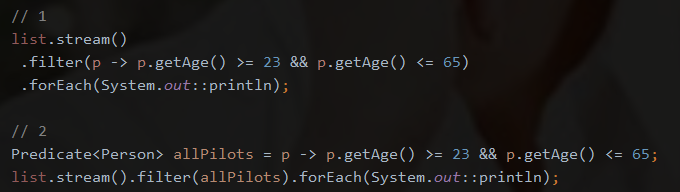
### 2.2.3 例子

（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);

#### 2.4.3 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。

#### 2.4.4 Stream的终止操作

(1) 查找与匹配

① boolean allMatch(Predicate p)：检查是否匹配所有元素

② boolean anyMatch( (Predicate p) )：检查是否至少匹配一个元素

③ boolean noneMatch(Predicate p)：检查是否没有匹配所有元素

④ Optional<T> findFirst()：返回第一个元素

⑤ Optional<T> findAny()：返回当前流中的任意元素

⑥ long count()：返回流中元素的总个数

⑦ Optional<T> max()：返回流中最大值

⑧ Optional<T> min()：返回流中最小值

⑨ void forEach (Consumer c c) )：内部迭代

例子：找工资最小的人和找工资最大的工资

Optional<Employee> op2 = emps.stream().min(Comparator.comparingDouble(Employee::getSalary));

Optional<Double> op = emps.stream().map(Employee::getSalary).max(Double::compare);

(2) 归约

① **T reduce(**T identity, BinaryOperator**)**：可以将流中元素反复结合起来，得到一个值。Identity为初始值。

② **Optional<T>** **reduce(**BinaryOperator**)**：可以将流中元素反复结合起来，得到一个值。

例子：Integer sum = list.stream().reduce(0, (x, y) -> x+y); //0 作为 x，流中第一个元素作为y

**Optional<Double>** salarySum = emps.stream().map(Employee::getSalary).reduce(Double::sum);

(3) 收集

① **collect(Collector c)**：将流转换为其他形式。接收一个 Collector接口的实现。

Collectors 实用类提供了很多静态方法。

|  |  |  |
| --- | --- | --- |
| 方法 | 返回类型 | 作用 |
| toList | List<T> | 把流中元素收集到List |
| toSet | Set<T> |  |
| toCollection | Collection<T> |  |
| toMap | Map |  |

例子 ① emps.stream().map(Employee::getName).collect(Collectors.toList());

② emps.stream().collect(Collectors.toCollection(ArrayList::new));

③ emps.stream().collect(Collectors.toMap(**Employee::getName, Function.identity(), (key1, key2)->key2**));

|  |  |  |
| --- | --- | --- |
| counting | Long | 计算流中元素的个数 |
| summingInt | Integer | 对流中元素的整数属性求和 |
| averagingInt | Double | 计算流中元素Integer属性的平均值 |
| summarizingInt | IntSummaryStatistics | 收集流中Integer属性的统计值。如：平均值 |
| maxBy | Optional<T> | 根据比较器选择最大值 |
| groupingBy | Map<K, List<T>> | 根据某属性值对流分组，属性为K，结果为V |
| partitioningBy | Map<Boolean, List<T>> | 根据true或false进行分区 |

例子 ① list.stream().collect(Collectors.groupingBy(Employee::getStatus));

#### 2.4.5 并行流

并行流 就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

Stream API 可以声明性地通过 parallel() 与sequential() 在并行流与顺序流之间进行切换。

并行流底层使用的是 Fork/Join框架。

Fork/Join 框架：就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），并将小任务加到线程队列（双端队列）中，再将一个个的小任务运算的结果进行 join 汇总。

优点：在一般的线程池中,如果一个线程正在执行的任务由于某些原因无法继续运行,那么该线程会处于等待状态。而在fork/join框架实现中,处理该子问题的线程会主动寻找其他尚未运行的子问题来执行。

|  |  |
| --- | --- |
| public class ForkJoinCalculate extends **RecursiveTask**<Long> {  private long start;  private long end;  private static final long THREHOLD = 10000; // 临界值  public ForkJoinCalculate(long start, long end) {  this.start = start;  this.end = end;  }  @Override  protected Long **compute**() {  long length = end - start;  if (length <= THREHOLD) { // 计算  long sum = 0;  for (long i = start; i <= end; i++) {  sum += i;  }  return sum;  }else { // 拆分  long middle = (start + end) / 2;  ForkJoinCalculate left = new ForkJoinCalculate(start, middle);  left.fork(); //拆分子任务，同时压入线程队列  ForkJoinCalculate right = new ForkJoinCalculate(middle+1, end);  right.fork();  return left.join() + right.join();  }  }  } | |
| // 测试自己写的 ForkJoin  public void test1(){  long start = System.currentTimeMillis();  ForkJoinPool pool = new ForkJoinPool();  ForkJoinTask<Long> task  = new ForkJoinCalculate(0L, 10000000000L);  long sum = pool.invoke(task);  System.out.println(sum);  long end = System.currentTimeMillis();  System.out.println("耗时为: " + (end - start));  } | // JDK8 并行流  public void test3(){  long start = System.currentTimeMillis();  Long sum  = **LongStream.rangeClosed**(0L, 10000000000L)  **.parallel()**  **.sum();**  System.out.println(sum);  long end = System.currentTimeMillis();  System.out.println("耗时为: " + (end - start));  } |

### 2.5 接口中的默认方法与静态方法

Java 8中允许接口中包含具有具体实现的方法，该方法称为“默认方法”，默认方法使用 default 关键字修饰。

ava8 中，接口中允许添加静态方法。调用方式相同。

注意：**接口中的默认方法**

若一个接口中定义了一个默认方法，而另外一个父类或接口中又定义了一个同名的方法时：

① 父类提供了具体的实现，那么选择父类中的方法，接口默认方法被忽略；

② 父接口提供一个默认方法

### 2.6 新时间日期 API

#### 2.6.1 LocalDate / LocalTime / LocalDateTime

LocalDate、LocalTime、LocalDateTime 类的实例是不可变的对象，分别表示日期、时间、日期和时间。

常用方法：

* LocalDateTime.**now**() / LocalDateTime.of(1989, 1,20, 21, 30, 55) 获取时间
* plusDays / plusWeeks / plusMonths / plusYears 加
* minusDays / minusWeeks / minusMonths / minusYears 减
* withDayOfMonth / withDayOfYear / withMonth / withYear 设置
* getDayOfMonth / getDayOfYear / getDayOfWeek / getMonth / getMonthValue / getYear
* isBefore / isAfter
* isLeapYear 是否是闰年

#### 2.6.2 Instant时间戳

Instant用于“时间戳”的运算。

常用方法：

Instant.now() / Instant.now().atOffset(ZoneOffset.ofHours(8));

toEpochSecond() 获取秒级时间戳

Instant.ofEpochSecond(1) / ofEpochMilli 从Unix 元年 开始 增加

#### 2.6.3 Duration / Period

Duration:用于计算两个“时间”间隔

Period:用于计算两个“日期”间隔

|  |
| --- |
| Duration.between(ins1,ins2).toMillis(); // 两个时间之间的毫秒差 toMinutes、toDays、toHours、getSeconds  Period.between(ld1,ld2).getYears(); // 两个日期之间的年差 getYears、getMonths、getDays |

#### 2.6.4 TemporalAdjuster

TemporalAdjuster : 时间校正器。

TemporalAdjusters : 该类通过静态方法提供了大量的常用 TemporalAdjuster 的实现。

实例：

① 获取下一个周日

LocalDateTime.now().with(TemporalAdjusters.next(DayOfWeek.SUNDAY));

② 获取下一个工作日

TemporalAdjuster ta = (l) -> {

LocalDateTime ldt3 = (LocalDateTime) l;

DayOfWeek dayOfWeek = ldt3.getDayOfWeek();

if (dayOfWeek.equals(DayOfWeek.FRIDAY)) {

return ldt3.plusDays(3);

} else if (dayOfWeek.equals(DayOfWeek.SATURDAY)) {

return ldt3.plusDays(2);

} else {

return ldt3.plusDays(1);

}

};

#### 2.6.5 DateTimeFormatter

java.time.format.DateTimeFormatter 类用于解析与格式化。

常用方法

① 自带的：DateTimeFormatter.ISO\_LOCAL\_DATE;

② 自定义格式：DateTimeFormatter.ofPattern("yyyy-MM-dd");

格式化：LocalDateTime.now().format(dtf); / dtf.format(LocalDateTime.now());

解析：LocalDateTime.parse(str, dft2);

#### 2.6.6 时区的处理

带时区的时间为分别为：ZonedDate、ZonedTime、ZonedDateTime

ZoneId：该类中包含了所有的时区信息，常用方法如下：

① getAvailableZoneIds() : 可以获取所有时区时区信息

② of(id) : 用指定的时区信息获取 ZoneId 对象

③ LocalDateTime.now(ZoneId.of("Europe/Monaco"));

④ LocalDateTime.now().atZone(ZoneId.of("Europe/Monaco"))

### 2.7 其他新特性

#### 2.7.1 Optional 类

Optional<T> 类(java.util.Optional) 是一个容器类，代表一个值存在或不存在，原来用 null 表示一个值不存在，现在 Optional 可以更好的表达这个概念。并且可以避免空指针异常。

**常用方法：**

* Optional.of(T t) : 创建一个 Optional 实例，若参数 t 为 null，报异常。
* Optional.empty() : 创建一个空的 Optional 实例。调用get方法，报异常。
* Optional.ofNullable(T t):若 t 不为 null,创建 Optional 实例；否则创建空实例，调用get方法，报异常。
* boolean isPresent() : 判断是否包含值。
* T orElse(T t) : 如果调用对象包含值，返回该值，否则返回 t。
* T orElseGet(Supplier s) :如果调用对象包含值，返回该值，否则返回 s 获取的值。
* Optional<U> map(Function<? super T, ? extends U> mapper): 如果有值对其处理，并返回处理后的Optional，否则返回 Optional.empty()。
* Optional<U> flatMap(Function<? super T, Optional<U>> mapper):与 map 类似，要求返回值必须是Optional。

实例：

|  |  |
| --- | --- |
| // old method  public String getGodnessName(Man man) {  if (man != null && man.getGodness() != null) {  return man.getGodness().getName();  } else { return "默认值";  }  } | // new method  public String getGodnessNameNew (Optional<NewMan> man) {  return name = man.orElse(new NewMan())  .getGodness()  .orElse(new Godness("默认值"))  .getName();  } |

#### 2.7.2重复注解与类型注解

# 3、JDK9

### 3.1 modularity System 模块系统(☆☆☆☆☆)

Java提供该功能的主要的动机在于，减少内存的开销，只需必要模块，简化开发和维护，改进安全性，提高性能。

Modularity提供了类似于OSGI框架的功能，模块之间存在相互的依赖关系，可以导出一个公共的API，并且隐藏实现的细节，本质上讲，模块(module)的概念，其实就是package外再裹一层。也就是说，用模块来管理各个package，通过声明某个package暴露，不声明默认就是隐藏。

#### 3.1.1 暴露

模块：由类和新的模块声明文件（module-info.java）组成。通过“exports”暴露“package”，让其它模块访问。

exports ：控制着哪些包可以被其它模块访问到。所有不被导出的包默认都被封装在模块里面。

|  |
| --- |
| module java9test { // module-info.java  exports com.atguigu.bean;  } |

#### 3.1.2 依赖

requires ：指明对其它模块的依赖。

|  |
| --- |
| module java9demo {  requires java9test;  requires java.logging;  requires junit;  } |

### 3.2 jShell 命令(☆☆☆☆☆)

交互式编程环境JShell。利用 jShell 在没有创建类的情况下直接声明变量，计算表达式，执行语句。即开发时可以

在命令行里直接运行 java 的代码，而无需创建 Java 文件，也可以从文件中加载语句或者将语句保存到文件中。

默认已经导入如下的所有包（使用命令“/imports”查看导入的包）：（包含 java.lang 包）



命令：

① “/imports”：查看导入的包；

② “/list”：列出当前 session 里所有有效的代码片段;

③ “/var”：查看当前 session下所有创建过的变量；

④ “/methods”：查看当前 session 下所有创建过的 方法。

⑤ “/open 路径”：从外部 文件加载源代码

⑥ 没有受检异常（编译时异常），因为jShell 在后台为我们隐藏了。

⑦ “/exit”：退出。

### 3.3 多版本兼容 JAR

多版本兼容 jar 功能能让你创建仅在特定版本的 Java 环境中运行库程序选择使用的 class 版本。

|  |
| --- |
| jar root  - A.class  - B.class  - META-INF  - versions  - 9  - A.class |

说明：在上述场景中， root.jar 可以在 Java 9 中使用, 不过 A类使用的 不 是 顶 层A. class, 而 是 处 在“META-INF/versions/9”下面的这个。因为较老版本的 Java 只会看到顶层的 A 类。

#### 3.3.1 测试

步骤一：提供必要的类

① multijar\src\main\java\com\fang：Generator.java

② multijar\src\main\java\com\fang：Application.java

③ multijar\src\main\java-9\com\fang：Generator.java

步骤二：打包

① javac -d build --release 8 src/main/java/com/atguigu/\*.java

② javac -d build9 --release 9 src/main/java-9/com/atguigu/\*.java

③ jar --create --main-class=Application --file multijar.jar -C build . --release 9 -C build9 .

步骤三：在 java 9 及之前版本的环境下进行测试即可：将打好的 jar 包作为 lib 添加到项目中。

### 3.4 语法改进：私有接口方法

Java7中的接口：抽象方法（public abstract）+常量（public static final）。

Java8中的接口：抽象方法+常量+默认方法+静态方法：Java 8 中规定接口中的方法除了抽象方法之外，还可以定义静态方法和默认的方法，此时的接口更像是一个抽象类。

Java9中接口：抽象方法+常量+默认方法+静态方法+私有方法+私有静态方法：在 Java 9 中，接口更加的灵活和强大，连方法的访问权限修饰符都可以声明为 private 的了

### 3.5 不可变集合工厂方法

|  |
| --- |
| // JDK8  List<String> list = Collections.unmodifiableList(Arrays.asList("Tom", "Jerry"));  list.forEach(System.out::println);  Set<String> set = Collections.unmodifiableSet(new HashSet<>(Arrays.asList("Tom", "Jerry")));  set.forEach(System.out::println);  Map<String, Integer> map = Collections.unmodifiableMap(new HashMap<String, Integer>(){  {  put("zhangsan",18);  put("lisi",19);  put("wangwu",20);  }  });  map.forEach((k,v)->System.out.println(k + ": " + v)); |
| // JDK9  List<String> list = List.of("a", "b", "c"); //接口静态方法  Set<String> set = Set.of("a", "b", "c");  Map<String, Integer> map1 = Map.of("Tom", 12, "Jerry", 21,"Lilei", 33, "HanMeimei", 18);  Map<String, Integer> map2 = Map.ofEntries(Map.entry("Tom", 89),Map.entry("Jim", 78),Map.entry("Tim", 98)); |

### 3.6 增强的 Stream API

在 Java 9 中，Stream API 变得更好，Stream 接口中添加了 4 个新的方法：dropWhile, takeWhile, ofNullable，还有个 iterate 方法的新重载方法。

① dropWhile(Predicate p)：当满足条件p就结束，返回前面的元素；

② takeWhile(Predicate p)：当满足条件p就开始，返回当前元素及其后面的元素；与dropWhile为补集；

③ ofNullable：Java 8：Strea.of(null)会报空指针异常。JDK9可以使用：Stream.ofNullable(null); [个数为：0]；

④ iterate(final T seed, Predicate p, final UnaryOperator<T> f)：当满足条件 P ，终止。

⑤ optional.stream(); 将optional转换为stream。

### 3.7垃圾收集机制

Java 9 移除了在 Java 8 中 被废弃的垃圾回收器配置组合，同时把G1设为默认的垃圾回收器实现。

jdk1.7 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）。

jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）。

jdk1.9 默认垃圾收集器G1。

Java7中已经将运行时常量池从永久代移除，在Java 堆（Heap）中开辟了一块区域存放运行时常量池。

Java8中已经彻底没有了永久代，将方法区直接放在一个与堆不相连的本地内存区域，这个区域被叫做元空间。

### 3.8 HTTP 2

JDK9之前提供HttpURLConnection API来实现Http访问功能，开发中一般都会选择Apache的Http Client，此次在Java 9的版本中引入了一个新的package:java.net.http，里面提供了对Http访问很好的支持，不仅支持Http1.1而且还支持HTTP2以及WebSocket。

### 3.9 其他

* 目录结构改变：JDK9的安装目录中少了 jre 目录；
* <>：Set<String> set = new HashSet<>(){}; //编译错误

Set<String> set = new HashSet<String>(){}; //编译正确

Set<String> set = new HashSet<>(){}; //JDK9 编译正确

* 关闭资源：JDK7：finally中显示关闭；JDK8：try(InputStreamReader reader = new InputStreamReader(System.in)) 可以自动关闭；JDK9：可以在 try 子句中使用:已经初始化过的资源，此时的资源是 final 的。

|  |  |
| --- | --- |
| // JDK 8  try(**InputStreamReader reader = new**  **InputStreamReader(System.in)**){  }catch (IOException e){ } | // JDK9  InputStreamReader reader = new InputStreamReader(System.in);  try(**reader**){ //是 final 的，多个使用“;”隔开  }catch (IOException e){ } |

* 在 JDK 8 中，标识符可以独立使用“\_”来命名，在JDK9 中不在可以单独使用。
* String 再也不用 char[] 来存储啦，改成了 byte[] 加上编码标记，节约了一些空间。StringBuffer与StringBuilder同时变化。
* HTML5风格的Java帮助文档：并添加了搜索框。

# 4、MySQL

# 五、Spring boot

## 1. 简介

### 1.1 什么是Spring Boot

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，简化了繁重的配置，提供了各种启动器，开发者能快速上手。

**优点**：①快速创建独立运行的Spring项目以及与主流框架集成；②使用嵌入式的Servlet容器，应用无需打成WAR包； ③大量的自动配置，简化开发，也可修改默认值；④自动配置spring添加对应功能starter自动化配置；⑤无需配置XML，无代码生成，开箱即用。

### 1.2 常用的starter有哪些

Spring Boot将所有的功能场景都抽取出来，做成一个个的starters（启动器），只需要在项目里面引入这些starter

相关场景的所有依赖都会导入进来。要用什么功能就导入什么场景的启动器。

① spring-boot-starter：核心starter, 包含自动化配置，日志和yaml；

② spring-boot-starter-aop：AOP and AspectJ

③ spring-boot-starter-web：嵌入tomcat和web开发需要servlet与jsp支持

④ spring-boot-starter-data-jpa / spring-boot-starter-jdbc：数据库支持

⑤ spring-boot-starter-data-redis：redis数据库支持

⑥ mybatis-spring-boot-starter ：第三方的mybatis集成starter

### 1.3 自动配置的原理

主程序类：@SpringBootApplication 来标注一个主程序类，运行这个类的main方法来启动SpringBoot应用；@SpringBootApplication 包含两个重要的注解：@SpringBootConfiguration @EnableAutoConfiguration；

* @SpringBootConfiguration：Spring Boot的配置类，表示这是一个Spring Boot的配置类。
* @SpringBootConfiguration里面包含 注解：@Configuration：配置类上来标注这个注解；
* @Configuration 里面包含 @Component，说明配置类也是容器中的一个组件；
* @EnableAutoConfiguration：开启自动配置功能；包含两个重要的注解：@ AutoConfigurationPackage @Import：
* @AutoConfigurationPackage：里面包含 @Import(Registrar.class) ，给容器中导入一个组件；
  + Registrar里面有一个方法：registerBeanDefinitions() 将主配置类（@SpringBootApplication标注的类）的所在包及下面所有子包里面的所有组件扫描到Spring容器；
* @Import({AutoConfigurationImportSelector.class})；导入哪些组件的选择器；
  + AutoConfigurationImportSelector里面有一个方法：selectImports，
  + selectImports方法调用getCandidateConfigurations方法，
  + 调用SpringFactoriesLoader.loadFactoryNames (EnableAutoConfiguration.class,classLoader)；
  + 将类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值以全类名的方式返回，
  + 作为自动配置类导入到容器中，会给容器中导入非常多的自动配置类（xxxAutoConfiguration）；
  + 就是给容器中导入这个场景需要的所有组件，并配置好这些组件；
  + 自动配置类就生效，帮我们进行自动配置工作；以前我们需要自己配置的东西，自动配置类都帮我们；
* J2EE的整体整合解决方案和自动配置都在spring-boot-autoconfigure-1.5.9.RELEASE.jar；

## 2. 配置

### 2.1 配置文件

* Spring Boot使用一个全局的配置文件：application.properties 或者application.yml
* 配置文件放在src/main/resources目录或者类路径/config下
* 全局配置文件的可以对一些默认配置值进行修改

### 2.2 YAML语法

基本语法：

* k:(空格)v：表示一对键值对（空格必须有）；
* 以**空格**的缩进来控制层级关系；
* 只要是左对齐的一列数据，都是同一个层级的
* 属性和值也是大小写敏感；

值的写法

* k: v
* 字符串：默认不用加上单引号或者双引号
* "" 双引号：name: "zhangsan \n lisi"：输出；zhangsan 换行 lisi
* '' 单引号： name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi
* Map：

|  |  |
| --- | --- |
| friends:  lastName: zhangsan | friends: {lastName: zhangsan} |

* List：

|  |  |
| --- | --- |
| pets:   ‐ cat | pets: [cat] |

### 2.3 配置文件值注入

#### 2.3.1 添加提示 pom

|  |
| --- |
| <!‐‐导入配置文件处理器，配置文件进行绑定就会有提示‐‐>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring‐boot‐configuration‐processor</artifactId>  <optional>true</optional>  </dependency> |

#### 2.3.2 实例

|  |  |
| --- | --- |
| person:  last-name: zhangsan  age: 18  boss: true  birth: 2000/01/20  maps:  k1: v1  k2: v2  lists:  - 桌子  - 椅子  dogs:  name: 小狗  age: 2 | @Data  @Component  @ConfigurationProperties(prefix = "person")  public class Person {  private String lastName;  private Integer age;  private Boolean boss;  private Date birth;  private Map<String, Object> maps;  private List<Object> lists;  private Dog dogs;  }  @Autowired  Person person; |

**注意：**

* @ConfigurationProperties：将本类中的所有属性和配置文件中相关的配置进行绑定；

prefix = "person"：配置文件中哪个下面的所有属性进行一一映射

* @Component：只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；

#### 2.3.3 @Value VS @ConfigurationProperties



|  |  |
| --- | --- |
| **// ConfigurationProperties**  @Component  @ConfigurationProperties(prefix = "person")  @Validated  public class Person {  @NotNull  private String lastName;  } | **// Value**  @Component  public class Person {  @Value("${person.last-name}") // 跟配置文件中名称一样  private String lastName;  @Value("#{11\*2}")  private Integer age;  } |

#### 2.3.4 PropertySource

@PropertySource：加载指定的配置文件；

|  |
| --- |
| @PropertySource(value = {"classpath:person.properties"})  @Component  @ConfigurationProperties(prefix = "person") |

#### 2.3.5 ImportResource 不建议

@ImportResource：导入Spring的配置文件，让配置文件里面的内容生效。

Spring Boot里面没有Spring的配置文件，我们自己编写的配置文件，也不能自动识别。想让Spring的配置文件生效，加载进来；需要在主程序类上 添加 注解 @ImportResource。

|  |
| --- |
| // 以前的方法，**不建议使用**  <bean id="helloService" class="com.fang..service.HelloService"></bean> // xml配置文件  @ImportResource(locations = "classpath:beans.xml") //主程序类  @SpringBootApplication  public class Application { |

#### 2.3.6 添加组件

使用@Configuration标识配置类，然后使用@Bean给容器中添加组件，例如：

|  |
| --- |
| /\*\*   \* @Configuration：指明当前类是一个配置类；就是来替代之前的Spring配置文件   \*  @Configuration  public class MyAppConfig {      //将方法的返回值添加到容器中；容器中这个组件默认的id就是方法名      @Bean      public HelloService helloService(){          return new HelloService();      }  } |

### 2.4 profile

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

#### 2.4.1 多环境配置文件

**第一种方式：使用多Profile文件**

Step1：不同的环境使用下面的命名方式： application-{profile}.properties/yml；

Step2：激活指定profile。

**第二种方式：yml支持多文档块方式**

Step1：在application.yml 中 用“‐‐‐”分成文档块；

Step2：激活指定profile。

|  |
| --- |
| server:    port: 8081  ‐‐‐  server:    port: 8083  spring:    profiles: dev  ‐‐‐  server:    port: 8084  spring:    profiles: prod  #指定属于哪个环境 |

#### 2.4.2激活指定profile

**第一种方式：**在配置文件application.yml中指定 spring.profiles.active=dev

**第二种方式：**运行时指定Program arguments参数：**--spring.profiles.active=dev**

或者 打包，运行 java -jar spring-boot-02-config-1.0.jar **--spring.profiles.active=dev**

**第三种方式：**运行时指定VM options参数：**-Dspring.profiles.active=dev**

优先级：第二种 > 第三种 > 第一种

### 2.5 配置文件加载位置

扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件：

① file:./config/ // 即 项目下，config与src同一层；

② file:./ // 即 项目下，与src同一层；

③ classpath:/config/ // 即 src/main/resources 下，config与 static同一层；

④ classpath:/ // 即 src/main/resources 下，与 static同一层；

优先级由高到底，高优先级的配置会覆盖低优先级的配置；

可以通过命令行参数：spring.config.location 指定配置文件的位置，优先级最高。

|  |
| --- |
| java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=E:/application.yml |

### 2.6 外部配置加载顺序

以下位置加载配置,优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会形成互补配置:

① **命令行参数**：java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc

② jar包外部的application-{profile}.properties配置文件

③ jar包内部的application-{profile}.properties配置文件

④ jar包外部的application.properties配置文件

⑤ jar包内部的application.properties配置文件

⑥ @Configuration注解类上的@PropertySource

⑦ 通过SpringApplication.setDefaultProperties指定的默认属性

### 2.7 自动化配置原理

SpringBoot启动的时候加载主配置类，即用注解@SpringBootApplication标记的类；SpringBootApplication 利用注解 @EnableAutoConfiguration 开启了自动配置功能；

* EnableAutoConfiguration 利用 @Import({AutoConfigurationImportSelector.class}) 给容器中导入一些组件；
* AutoConfigurationImportSelector 中有一个方法 selectImports，此方法调用 List<String> configurations = getCandidateConfigurations(); 获取候选的配置
  + 调用SpringFactoriesLoader.loadFactoryNames()：扫描所有jar包类路径下  META‐INF/spring.factories，将这些文件的内容包装成properties对象，从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中。【spring-boot-autoconfigure】
* 每一个自动配置类，自动配置。

#### 2.7.1 HttpEncodingAutoConfiguration

以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理：

|  |
| --- |
| // 表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件  **@Configuration**  // 启用HttpProperties的ConfigurationProperties功能  **@EnableConfigurationProperties**(HttpProperties.class)  // Spring底层@Conditional注解：根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会生效  **@ConditionalOnWebApplication**(type = ConditionalOnWebApplication.Type.SERVLET)  //判断当前项目有没有这个类CharacterEncodingFilter【SpringMVC中进行乱码解决的过滤器】  **@ConditionalOnClass**(CharacterEncodingFilter.class)   //判断配置文件中是否存在某个配置  spring.http.encoding.enabled；如果不存在，判断也是成立的  **@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing = true)**  **public class HttpEncodingAutoConfiguration {** |
| @Configuration**Properties**(prefix = " spring.http.encoding") //从配置文件中获取指定的值和bean的属性进行绑定  public class HttpProperties { // application.yml中可以配spring.http.encoding.属性 |

根据当前不同的条件判断，决定这个配置类是否生效？一旦这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的；

**精髓**：

1）、SpringBoot启动会加载大量的自动配置类

2）、我们看我们需要的功能有没有SpringBoot默认写好的自动配置类；

3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）

4）、给容器中自动配置类添加组件时，会从properties类中获取某些属性。可以在配置文件中指定这些属性的值；

**xxxxAutoConfigurartion**：自动配置类；给容器中添加组件

**xxxxProperties**：封装配置文件中相关属性；

#### 2.7.2 Conditional

**自动配置类必须在一定的条件下才能生效**；

我们可以通过启用 debug=true属性；来让控制台打印自动配置报告，这样就可以知道哪些自动配置类生效。

## 3. 日志

SpringBoot：底层是Spring框架，Spring框架默认是用JCL（Commons Logging）；

SpringBoot选用 SLF4j和logback；

### 3.1 在系统中使用SLF4j

|  |
| --- |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class HelloWorld {  public static void main(String[] args) {       Logger logger = LoggerFactory.getLogger(HelloWorld.class);       logger.info("Hello World");     }  } |

如何让系统中所有的日志都统一到slf4j；

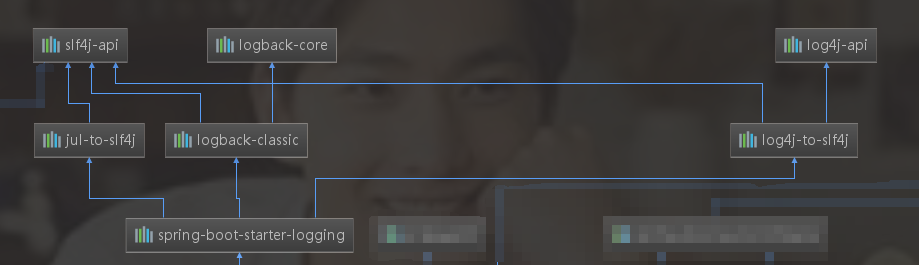
1、将系统中其他日志框架先排除出去；

2、用中间包来替换原有的日志框架；

3、我们导入slf4j其他的实现。

### 3.2 SpringBoot日志关系

spring-boot-starter 依赖 spring-boot-starter-logging



其中，logback-core：使用logback记录日志；

jul-to-slf4j 、log4j-to-slf4j ：将其他日志转为slf4j；slf4j-api导入了日志抽象层

总结：

1）、SpringBoot底层也是使用slf4j+logback的方式进行日志记录

2）、SpringBoot也把其他的日志都替换成了slf4j；

### 3.3 日志使用

#### 3.3.1 默认配置

spring-boot 下 org.springframework.boot.logging.logback配置文件：

|  |
| --- |
| <included>  <include resource="org/springframework/boot/logging/logback/defaults.xml" />  <property name="LOG\_FILE" value="${LOG\_FILE:-${LOG\_PATH:-${LOG\_TEMP:-${java.io.tmpdir:-/tmp}}}/spring.log}"/>  <include resource="org/springframework/boot/logging/logback/console-appender.xml" />  <include resource="org/springframework/boot/logging/logback/file-appender.xml" />  <root level="**INFO**">  <appender-ref ref="CONSOLE" />  <appender-ref ref="FILE" />  </root>  </included> |

日志的级别由低到高   trace<debug<info<warn<error

SpringBoot默认给我们使用的是info级别的。日志就只会在这个级别及以后的高级别生效。

#### 3.3.2 修改默认配置

|  |  |  |  |
| --- | --- | --- | --- |
| logging.file | logging.path | Example | Description |
| (none) | (none) |  | 只在控制台输出 |
| 指定文件名 | (none) | my.log | 输出日志到my.log文件 |
| (none) | 指定目录 | /var/log | 输出到指定目录的 spring.log 文件中 |

#### 3.3.3 指定位置

|  |  |
| --- | --- |
| Logging System | Customization |
| Logback | **logback-spring.xml** , logback-spring.groovy ,  logback.xml logback.groovy |
| Log4j2 | log4j2-spring.xml log4j2.xml |
| JDK (Java Util Logging) | logging.properties |

logback.xml：直接就被日志框架识别了；

logback-spring.xml：日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用SpringBoot

的高级Profile功能。例如： <**springProfile** name="dev"></**springProfile**>

#### 3.3.4 切换日志框架

在日志适配图中去掉：logback-classic 和 log4j‐over‐slf4j；添加slf4j‐log4j12依赖后，就可以使用log4j框架。

**第一种：切换为log4j**

|  |  |
| --- | --- |
| // 切换为 log4j  <dependency>    <groupId>org.springframework.boot</groupId>    <artifactId>**spring-boot-starter-web**</artifactId>    <exclusions>      <exclusion>        <artifactId>**logback-classic**</artifactId>        <groupId>ch.qos.logback</groupId>      </exclusion>      <exclusion>        <artifactId>**log4j-over-slf4j**</artifactId>        <groupId>org.slf4j</groupId>      </exclusion>    </exclusions>  </dependency>  <dependency>    <groupId>org.slf4j</groupId>    <artifactId>**slf4j-log4j12**</artifactId>  </dependency> | // 切换为 log4j2  <dependency>              <groupId>org.springframework.boot</groupId>              <artifactId>**spring-boot-starter-web**</artifactId>              <exclusions>                  <exclusion>                      <artifactId>**spring-boot-starter-logging**</artifactId>                      <groupId>org.springframework.boot</groupId>                  </exclusion>              </exclusions>          </dependency>  <dependency>    <groupId>org.springframework.boot</groupId>    <artifactId>**spring-boot-starter-log4j2**</artifactId>  </dependency> |

## 4. Web开发

Web开发的自动配置类为：**WebMvcAutoConfiguration**

### 4.1 获取静态资源

通过org.springframework.boot.autoconfigure.web.servlet.

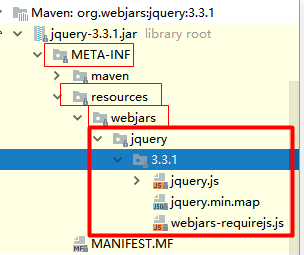
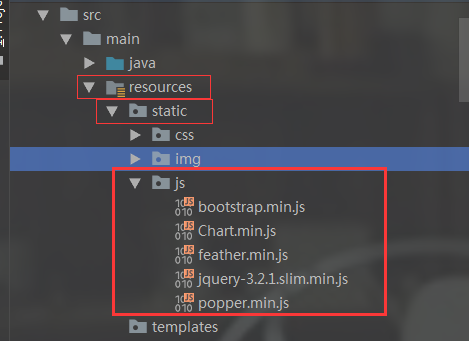
**WebMvcAutoConfiguration**.WebMvcAutoConfigurationAdapter# **addResourceHandlers方法：**

(1)、所有webjars方式导进来classpath:/META-INF/resources/webjars/的资源， 以“**/webjars/\*\***”访问；

|  |
| --- |
| <dependency> //【webjars：以jar包的方式引入静态资源；<http://www.webjars.org/>】  <groupId>org.webjars</groupId>  <artifactId>jquery</artifactId>  <version>3.3.1</version>  </dependency> |

(2)、以下路径的资源，都可以以 "/\*\*" 访问；

|  |
| --- |
| "classpath:/META‐INF/resources/",  "classpath:/resources/",  "classpath:/static/",  "classpath:/public/"  "/"：当前项目的根路径 |

访问地址：localhost:8080/webjars/jquery/3.3.1/jquery.js

访问地址：localhost:8080/js/Chart.min.js

## 1.1.4 springboot读取配置文件的方式

① spring‐boot‐starter‐parent 的父类 是 spring‐boot‐dependencies，其管理Spring Boot应用里面的所有依赖版本；

**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

#### 1.1.2 什么是微服务

微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。微服务的概念源于2014年3月Martin Fowler的文章中提出。

#### 1.1.3 SOA vs 微服务Microservice



#### 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！

1.基础