1、lambda 表达式（→箭头操作符）

左侧：指定 Lambda 表达式需要的参数列表；
右侧：指定 Lambda 体，是抽象方法的实现逻辑，也即 Lambda 表达式要执行的功能。
1）无参，无返回值；

```
//使用Lambda 表达式
Runnable r2 = () -> {
    System.out.println("我爱北京故宫");
};
```

2）Lambda 需要一个参数（参数的小括号可以省略），但是没有返回值；
3）数据类型可以省略，可以有编译器推断；
4）当 Lambda 体只有一条语句时，return 与大括号若有，可以省略（要一起省略）。

2、函数式接口

只包含一个抽象方法的接口（可以包含其他非抽象方法）。可用@FunctionalInterface 注解。
学过的有：Runnabe 里的 run()方法；
Iterable 里的 iterate()方法；
Comparable 里的 compareTo()方法；
Comparator 里的 compare()方法。

3、四大核心函数式接口

1）消费型接口：void **accept**(T t)
有形参，但没有返回值；
2）供给型接口：T **get**()
无参，但有返回值；
3）函数型接口：R **apply**(T t)
既有参数，又有返回值；
4）判断型接口：boolean **test**(T t)
有参，返回值为 boolean 类型。

4、方法引用格式（::两个冒号）

1）情况 1：**对象 :: 实例**方法名；
2）情况 2：**类 :: 静态**方法名；
3）情况 3：**类 :: 实例**方法名；

【方法引用使用前提】

1）Lambda 体只有一句语句，并且是通过调用一个对象或类现有的方法来完成的；
2）情况 1：**函数式接口中的抽象方法 a** 在被**重写**时使用了**某一个对象的方法 b**。如果方法 a 的**形参列表**、**返回值类型**与方法 b 的形参列表、返回值类型**都相同**，则我们可以使用方法 b 实现对方法 a 的重写、替换。
3）情况 2：函数式接口中的抽象方法 a 在被重写时使用了**某一个类**的**静态方法 b**。如果方法 a 的形参列表、返回值类型与方法 b 的形参列表、返回值类型都相同，则我们可以使用方法 b 实现对方法 a 的重写、替换。
4）情况 3：函数式接口中的抽象方法 a 在被重写时使用了**某一个对象的方法 b**。如果方法 a 的返回值类型与方法 b 的**返回值类型相同**，同时方法 a 的**形参列表中有 n 个参数**，方法 b 的**形参列表有 n-1 个参数**，且方法 a 的**第 1 个参数**作为**方法 b 的调用者**，且方法 a 的后 n-1 参数与方法 b 的 **n-1 参数匹配**。

5、为什么要使用 Stream API？

实际开发中，项目中的数据大多数来自于关系数据库，但也有很多其他 **Nosql 的数据**，如 redis 里的数据，**需要在 Java 层面去处理**。

6、什么是 Stream？

Stream 是**数据渠道**，用于**操作数据源**（集合、数组等）**所生成的元素序列**。
Stream 和 Collection 集合的**区别**：Collection 是一种**静态的内存数据结构**，讲的是数据，而 Stream 是**有关计算的**，讲的是计算。前者是主要**面向内存，存储在内存中**，后者主要是**面向 CPU，通过 CPU 实现计算**。
【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**。即**执行终止操作**，才会**执行中间操作链**，并产生结果；
④ Stream 一旦**执行了终止操作**，就不能再调用其它中间操作或终止操作，**需要重新创建**。

7、stream 操作的三个步骤

1）创建 stream 流：从一个**数据源**获取流；
2）中间操作：每次**返回持有结果的新 stream**，可以是**操作链**，但在执**行终止操作前不执行**；
3）终止操作：**返回值类型**不再是 stream，并且执行完后**结束 stream**。

8、创建 stream 实例

1）集合中的实例方法：list.stream()；
2）数组中的静态方法：Arrays.stream(arr)；
3）Stream 类中的**静态方法 of()通过显示值**创建一个流：Stream.of(1,2,3,4,5)；

9、一系列中间操作

1）筛选和切片
1> Filter()：**过滤**不满足条件元素；
2> distinct()：通过流所生成元素的 hashCode()和 equals()**去除重复元素**；
3> limit()：截断流，流内元素**不超过给定数量**；
4> skip()：返回一个**跳过前 n 个元素**的流；（不足 n 则返回空流）
2）映射
map(函数表达式)：**该函数应用到每个元素上**，并将其映射为一个新的元素。
3）排序
sorted()：**自然顺序**排序；
sorted(Comparator com)：**比较器顺序**排序。

10、终止操作

1）匹配与查找
1>**allMatch**(Predicate p)：检查是否**匹配所有**元素；
2>**anyMatch**(Predicate p)：检查是否**至少匹配一个**元素；
3>**noneMatch**(Predicate p)：检查是否**没有匹配所有**元素；
4>**findFirst**()：**返回第一个**元素；
5>**findAny**()：返回当前流中的**任意元素**；
6>**count**()：返回流中**元素总数**；
7>**max**(Comparator c)：返回流中**最大值**；
8>**min**(Comparator c)：返回流中**最小值**；
9>**forEach**(Consumer c)：**内部迭代**，Stream API 使用内部迭代——它帮你把迭代做了。
2）归约
reduce()：可以将流中元素反复结合起来，得到一个值。
map 和 reduce 的连接称为 map-reduce 模式。
3）收集
collect()：给 Stream 中元素做汇总的方法。
Collector 接口中方法的实现决定了如何对流执行收集的操作。Collectors 实用类提供了很多静态方法：

```
toList← Collector<T, ?, List>← 把流中元素收集到 List←
List<Employee> emps= list.stream().collect(Collectors.toList());←
```

11、Java9 新增 stream 实例方法（ofNullable()）

Java 8 中 Stream **不能完全为 null**，否则会报**空指针异常**。而 **Java 9** 中的 **ofNullable 方法**允许我们**创建一个空 Stream 或带有空元素**。

```
//不报异常，允许通过
Stream<String> stringStream = Stream.of("AA", "BB", null);
System.out.println(stringStream.count());//3
```

12、Future 接口

Future 接口可以**为主线程开一个分支任务**，为主线程**处理耗时**和**费力**的复杂业务，然后 Future **获取**分支任务的**执行结果**或**任务状态**。
Future 接口（FutureTask 实现类）**定义了操作异步任务的方法**：获取异步任务的**执行结果** get()（可指定时间，超时未返回结果则抛异常），**取消任务**执行 cancel()，判断任务**是否被取消** isCancelled()，判断任务**执行是否完成** isDone()
【注意】
1）一旦调用 **get()方法求结果**，如果任务没有完成**会导致程序阻塞**，一般**建议放程序后面**。
2）一般业务中，会通过**循环去做**异步任务的**状态判断**，如果完成才会 get()取值，这样的**轮询会耗费 CPU**。
Future 只能通过阻塞或轮询获得结果，不友好。

13、CompletableFuture 类

CompletableFuture 类提供一种**观察者模式**机制，可以让任务执行**完成后通知监听的一方**。
CompletableFuture 同时实现了 **Future 接口**和 **CompletionStage 接口**。CompletionStage 接口描述了一个**异步计算的阶段**。很多计算可以**分成多个阶段或步骤**，此时可以通过它将所有步骤组合起来，**形成异步计算的流水线**。

14、创建 CompletableFuture 实例

1）通过 new 关键字，即当做 Future 使用；
2）四个静态工厂方法
【无返回值】
1> runAsync(Runnable r)
2> runAsync(Runnable r, Executor e)

```
CompletableFuture<Void> completableFuture = CompletableFuture.runAsync(() -> {
    System.out.println(Thread.currentThread().getName());
    //暂停几秒钟线程
    try { TimeUnit.SECONDS.sleep( timeout: 1); } catch (InterruptedException e) { e.printStackTrace(); }
});
System.out.println(completableFuture.get());
```

【有返回值】
3> supplyAsync(Supplier<U> s)
4> supplyAsync(Supplier<U> s, Executor e)

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    System.out.println(Thread.currentThread().getName());
    //暂停几秒钟线程
    try { TimeUnit.SECONDS.sleep( timeout: 1); } catch (InterruptedException e) { e.printStackTrace(); }
    return "hello supplyAsync";
});
System.out.println(completableFuture.get());
```

【注意】没有指定 Executor 的方法，**默认**使用 **ForkJoinPool()**作为线程池执行异步代码，该线程池**全局共享**，可能会**被其他任务占用**，导致**性能下降或者饥饿**。

15、CompletableFuture 类减少阻塞和轮询

他是 Future 的功能增强版，**减少阻塞和轮询**，可以**传入回调对象**，当异步任务**完成或发生异常时**，会**自动调用某个对象的回调方法**。主线程设置好回调后，不再关心一步任务的执行，异步任务之间可以顺序执行。

16、CompletableFuture 类获得结果的方法

get()阻塞等待；get(时间，单位)超时报异常；
join()和 get 区别：编译不用抛 Interrupte 异常；
getNow(valueIfAbsent):没计算完就返回该值；

17、主动触发计算

complete(value)：当分支未完成，则会打断(true)，立即返回括号中值；反之返回结果值。

18、处理异步计算的结果

【有返回值，可以访问异步结果】
thenApply()：**函数型接口**，s→{处理上一步的结果 s，**有返回值 return;**}
【没有返回值，可以访问异步结果】
thenAccept()：**消费型接口**；
【没有返回值，不可以访问异步结果】
thenRun()：方法参数是 **Runnable**；
whenComplete()：两个参数的消费型接口；

```
}).whenComplete((v,e) -> {
    if (e == null) {
        System.out.println("----计算结果：  "+v);
    }
}).exceptionally(e -> {
    e.printStackTrace();
    System.out.println(e.getMessage());
    return null;
```

19、处理异常结果

handle()：跟 thenApply()的区别在于，有异常的时候，可以往下多走一步，根据带的异常参数可以进一步处理。
20、选用计算速度快的
applyToEither()

```
CompletableFuture<String> result = playA.applyToEither(playB, f -> {
    return f + " is winer";
});
```

21、CompletableFuture 的组合

thenCompose()：将前一个任务的**返回结果作为下一个任务的参数**，它们之间存**在着先后顺序**。thenCombine()：会在**两个任务都执行完成后**，把两个任务的**结果合并**。两个任务并行。

22、并行运行多个 CompletableFuture（静态）

allOf()：并行运行多个 CompletableFuture，等**所有任务都运行完**成之后**再返回结果**；
anyOf()：返回**最先执行完**的**任务结果**。