# java 函数式编程

函数编程是将函数作为数据自由传递，结合泛型推导能力，从而提升代码表达能力。在函数式编程中将函数看成一等公民，使得处理力度从类变小为函数，从而可以更好满足系统对重用性和扩展性的支持。

# **函数式编程概念**

下面是Java Consumer接口使用示例，其是函数式接口包含抽象accept方法，定义如下：

*@FunctionalInterface*

*public interface Consumer<T> {*

*//抽象方法accept，只有输入而无输出*

*void accept(T t);*

*default Consumer<T> andThen(Consumer<? super T> after) {*

*Objects.requireNonNull(after);*

*return (T t) -> { accept(t); after.accept(t); };*

*}*

*}*

在Java 8之前，通过对象来传递，定义ConSumer对象，定义方式如下：

*Consumer c = new Consumer(){*

*@Override*

*public void accept(Object obj) {*

*System.out.println(obj);*

*}};*

而在Java 8中，针对函数式编程接口，定义如下：

*Consumer c2 = (obj) -> {*

*System.out.println(obj);*

*};*

这两种定义方式结果是相同的。函数式编程接口主要有三个核心概念：

1. 函数接口，Function
2. 流（Stream）
3. 聚合器（Collector）

# **函数式接口**

在Java8中新定义了一种接口类型，函数式接口，与其他接口的区别是：

* 函数式接口中只能有一个抽象方法
* 可以有从Object继承过来的抽象方法，所有类的最终父类都是Object
* 接口中唯一抽象方法的命名不重要，因为函数式接口对某一行为进行抽象，主要目的是支持Lambda表达式

函数接口的核心功能是对行为的抽象和进行数据转换，在JDK中最直接的支持就是Function包，定义了四个最基础的函数接口，如下所示：

* Supplier<T>，数据提供器，可以提供T类型对象，其内定义无参构造器，提供了Get方法，定义如下：

*@FunctionalInterface*

*public interface Supplier<T> {*

*//获取类型为T的值*

*T get();*

*}*

* Function<T,R>，数据转换器，接收T类型的对象，返回R类型对象，单参数单返回值的行为接口，提供apply,compose,andThen,identity方法

*@FunctionalInterface*

*public interface Function<T, R> {*

*R apply(T t);*

*//compose,andThen,identity，默认方法，不详述*

*}*

* Consumer<T>，数据消费者，接收T类型对象，无返回值，通常用于设置T对象的值，单参数无返回值的行为接口，提供accept,andThen方法

*@FunctionalInterface*

*public interface Consumer<T> {*

*void accept(T t);*

*default Consumer<T> andThen(Consumer<? super T> after) {*

*Objects.requireNonNull(after);*

*return (T t) -> { accept(t); after.accept(t); };*

*}}*

* Predicate<T>，条件测试器，接收T类型的对象，返回布尔值，通常用于传递条件函数，单参数布尔值的条件下接口，提供test和and-or\_negate方法

*@FunctionalInterface*

*public interface Predicate<T> {*

*boolean test(T t);*

*// and ,or,negate,isEqual*

*}*

## FunctionalInterface

Java 8为函数式接口引入了注解@FunctionalInterface，主要用于编译级错误检查，加上该注解，当写的接口不符合函数式接口定义时，编译器会报错，但是函数式接口不强制添加该注解，示例如下：

*@FunctionalInterface*

*public interface GreetingService {*

*void sayMessage(String message);*

*}*

测试方法执行如下：

*GreetingService gs = message -> System.out.println("Hello " + message);*

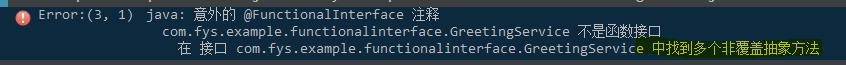
*gs.sayMessage("Greetting");*

执行成功。

在GreetingServer接口中添加第二个方法：

*@FunctionalInterface  
public interface GreetingService {  
 void sayMessage(String message);  
 int sayHello(String hello);  
}*

编译后，报错如下：



接口中包含了两个抽象方法，违反了函数式接口的定义。

但是函数式接口中可以包含默认方法、静态方法、Object中继承的public方法，

例如GettingService中添加默认方法：

*@FunctionalInterface*

*public interface GreetingService {*

*void sayMessage(String message);*

*default void sayHello() { //其定义default，上述的测试程序执行成功*

*System.out.println("Hello Default");*

*}*

*}*

## **Lambda表达式**

1）语法结构

Lambda表达式又称为闭包或者匿名方法，其语法结构和函数几乎是一样的，其由参数列表、箭头符号和函数体组成，格式如下所示：

*(parameters) -> expression*

*(parameters) ->{ statements; }*

函数体可以是表达式或者语句块，表达式的重要特征如下：

* 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，如果主体包含一个已经，就不需要使用大括号
* 可选的返回关键字，如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指明表达式返回一个数值

表达式的示例有以下5中形式：

*()-> 5， //不需要参数，返回值为5*

*X -> 2\* X， //接收一个参数，返回其2的倍值*

*(x,y) -> x - y //接收两个参数，并返回他们的差值*

*(int x, int y) -> x +y //接收2个int型整数，返回他们的和*

*(String s) -> System.out.println(s) //接收String对象，并在控制台打印，不返回任何值*

2）闭包

闭包是有状态的函数，其定义在另一个函数的作用域中，并且函数中引用了外部函数的局部变量（函数的执行环境依赖于创建时的一系列作用域的现象），那么这个函数就是闭包，下面看一个示例：

*static IntFunction<Integer> sumEx(int a) {*

*// 定义局部变量*

*final int ex = a \* 10;*

*// 自动类型推导，这是JAVA 8最牛的精华*

*return (value) -> { //这里返回一个闭包函数，其ex为创建时的局部变量*

*// 引用局部变量*

*return ex + value;*

*};*

*}*

## **回调函数**

简单解释：去商店买东西，刚好没有货，于是给店员留下联系电话。店里有货了，店员就打你的电话，然后你到店中取货。在这个例子中：

* 电话号码就叫回调函数
* 把电话留个店员就叫登记回调函数
* 店里后来有货了触发回调关联事件
* 店员给你打电话叫做调用回调函数
* 到店里取货叫做响应回调事件

回调函数就是一个通过函数指针调用的函数，如果把函数的指针作为参数传递给另一个函数，该指针用来调用所指向的函数是，这就是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外一方调用，用来对该事件或者条件进行响应。下面是一个示例：

1. 定义Callback接口

*public interface CallBack {*

*public void slove(); //回调函数*

*}*

2）定义药店行为

*public class DrugStore {*

*public void call(CallBack a) { //当货物时，通知买家拿货*

*System.out.println("Goods enougth, Store dial buyer to get goods");*

*a.slove();*

*}*

*}*

1. 定义买家买货行为

*public class Buyer implements CallBack {*

*DrugStore store = new DrugStore(); //去商店，并注册回调函数*

*@Override*

*public void slove() {*

*System.out.println("The problem is solved!");*

*}*

*public void askQuestion() {*

*System.out.println("Ask Store to get A Good, But lack"); //买货，但是缺货*

*new Thread(new Runnable() { //去做其他事情*

*@Override*

*public void run() {*

*System.out.println("Buyer want to do another thing!"); //执行完毕*

*}*

*}).start();*

*this.store.call(this); //回调User*

*}*

*public static void main(String[] args) {*

*Buyer a = new Buyer();*

*a.askQuestion();*

*}*

*}*

执行后，输出如下：

*Ask Store to get A Good, But lack*

*Goods enougth, Store dial buyer to get goods*

*The problem is solved!*

*Buyer want to do another thing!*

## **类型检查与推断**

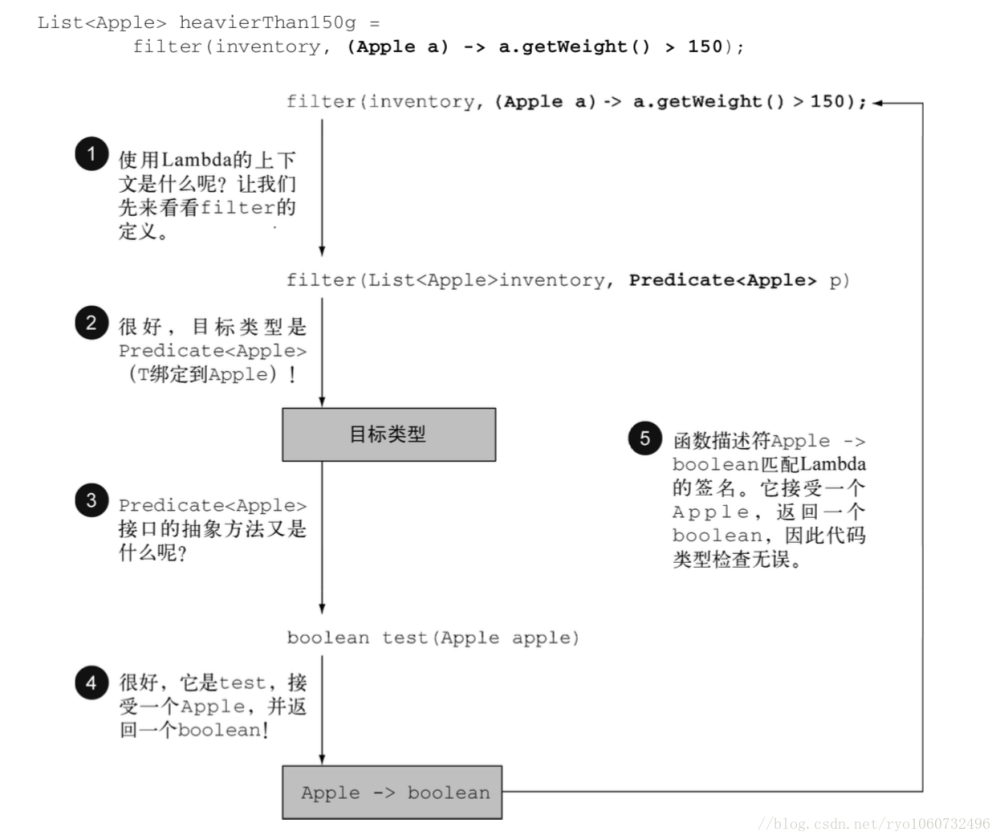
Java 8支持类型推断，而且仅对lambda表达式支持此功能，例如在某个数字范围内提取一个值时，编译器就知道该值的类型为int，如下：

*IntStream.rangeClosed(1, 5)*

*.forEach((number) -> System.out.println(number \* 2));*

1. 类型检查

Lambda表达式中类型推断违背了Java常规做法（指定每个变量和参数的类型），Lambda是基于上下文实现类型的推断：



这是一个类型检查过程

2）类型推断

Java编译器会从上下文推断出用什么函数式接口来配合Lambda表达式，意味着可以推断适合Lambda签名，因为函数描述符可以通过目标类型来得到。通过这种方式，编译器可以了解Lambda表达式的参数类型，在Lambda语法中省去标注参数类型，示例中：

*IntStream.forEach(number -> System.out.println(number \* 2)*

number省去了参数类型。

## **Lambda使用示例**

*//new HashMap<String, Integer>指定泛型类型*

*Map<String, Integer> wordcounts = new HashMap<String, Integer>();*

*//使用菱形操作符<>，new HashMap<>不用明确泛型类型，编译器就可以推断出。*

*Map<String, Integer> diamond = new HashMap<>();*

*//传递方法参数，根据形式参数做类型推断*

*targerTpyeMethod(new HashMap());*

*//lambda表达式类型推断,函数接口必须指明泛型类型，其他类型可以不用指明，可以进行推断。*

*Predicate<Integer> atLeast = x -> x > 5;*

*//The operator > is undefined for the argument type(s) Object, int*

*Predicate wrongLeast1 = x -> x > 5;*

*//Lambda expression's parameter x is expected to be of type Object*

http://www.runoob.com/java/java8-lambda-expressions.html

# **Stream API**

很多支持函数式编程的语言，基本上将map/reduce操作内置到语言的标准库中。Java 8 引入了全新的Stream API，其更像是具有Iterable的集合类，但行为和集合类有所不同，可以看做是集合对象功能的增加，专注于对集合对象进行各种便捷、高效的聚合操作或大批量数据操作，其使用示例如下：

*IntStream.range(0,10).forEach(*

*value -> System.out.println(value) //Lambda表达式*

*);*

Java 8 的Stream API充分利用Lambda表达式的特性，极大提高编程效率和程序可读性。IntStream.forEach方法的定义如下：

*void forEach(IntConsumer action);*

*@FunctionalInterface*

*public interface IntConsumer { //forEach方法中传入函数式接口*

*void accept(int value);*

*default IntConsumer andThen(IntConsumer after) {*

*Objects.requireNonNull(after);*

*return (int t) -> { accept(t); after.accept(t); };*

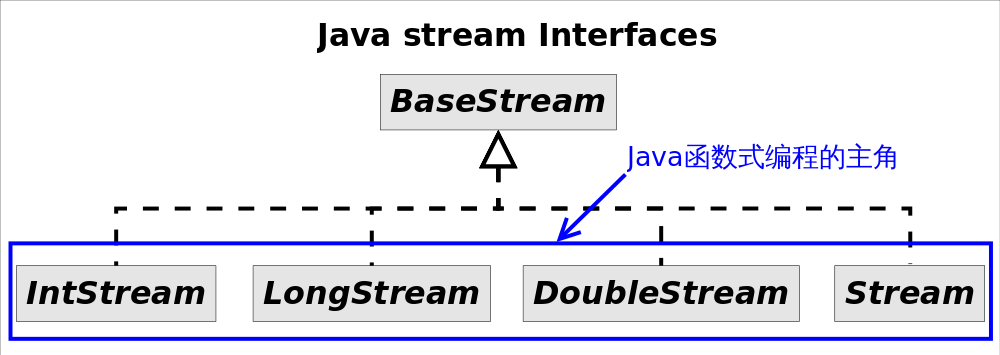
*}}*

## **Stream概念**

Stream不是集合元素，并不保存数据，其和有关算法和计算的，更像是高级版本的Iterator 和迭代器不同的：

* 可消费型，用户显式的遍历元素并对执行某些操作，比如”过滤掉长度大于10的字符串”，在使用时Stream数据单向遍历一次。
* 可以并行化操作，数据会被分成多个段，每一个都在不同的线程中处理，然后将结果一起输出。Stream的并行操作依赖于Java7中引入的Fork/Join框架（JSR166Y）来拆分任务和加快处理过程。
* 为函数式编程而生，数据源可以是无限的，其只是某种数据源的视图，可以是数组、Java容器或者I/O channel等，在Stream上执行任何修改都不会修改背后的数据源
* 惰性执行，Stream上的操作不会立即执行，只有等到用户真正需要结果的时候才会执行

常见的Stream接口继承关系如下图所示：

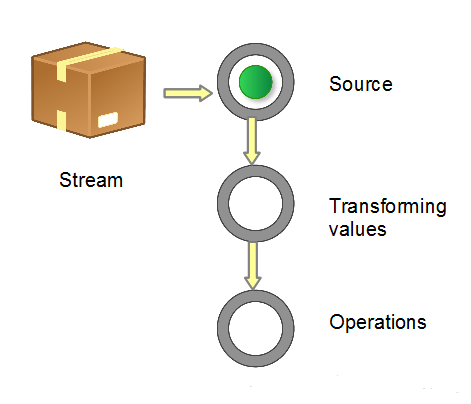


其分别对应三种类型int,long,double，Stream对应剩余类型的Stream视图。

https://www.ibm.com/developerworks/cn/java/j-lo-java8streamapi/

## **Stream的使用**

当使用一个流的时候，通常包括三个基本步骤，如下图：



* 获取一个数据源（source）
* 数据转换，Transforming values
* 执行操作，获取想要的结果

每次转换原有Stream对象不改变，返回新Stream对象，可以有多次转换，这允许对其操作像链条一样排列，形成一个管道。

*ArrayList<User> users = new ArrayList<>(); //定义用户列表，作为数据源*

*users.add(new User("2017001", false, 0));*

*users.add(new User("2017002", true, 36));*

*users.add(new User("2017010", false, 88));*

*users.add(new User("2017000", true, 10));*

*Stream<User> stream = users.stream(); //通过数组构造，转换成Stream*

*List<String> array = stream.filter(User::isVip)*

*.sorted(Comparator.comparing(User::getUserID))*

*.map(User::getUserID) //filter,sorted,map为执行操作*

*.collect(Collectors.toList()); //获取结果*

*array.stream().forEach(System.out::println);*

## **常用操作**

对Stream的操作分为两类，中间操作（Intermediate Operations）和结束操作（terminal Operations），二者的特点是：

1）Intermediate，中间操作

流后面可以跟随零个或者多个intermediate操作。其目的主要是打开流，作出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用。这类操作都是惰性化的（Lazy），也就是说仅调用这类方法，并没有真正开始流的遍历。诸如filter/sorted等中间操作会返回另一个流，这让多个操作可以连接起来。上例中：

*stream.filter(User::isVip)*

*.sorted(Comparator.comparing(User::getUserID))*

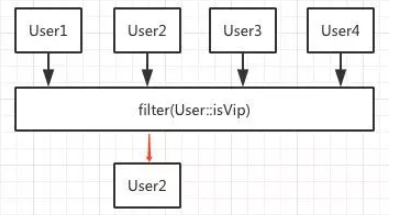
*.map(User::getUserID) //filter,sorted,map为中间操作*

Stream的filter接口如下：

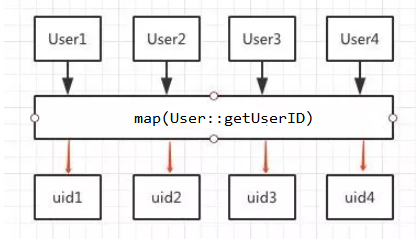
*Stream<T> filter(Predicate<? super T> predicate);*

常用方法如下所示：

* filter，需要传入实现Predicate函数式接口的对象，该接口的抽象方法接口一个参数并返回boolean值，为true则保留



* map/flatMap，函数需要传入实现Function函数式接口的对象，该方法接口抽象方法apply接收一个参数并返回值，可以理解为映射关系，原理如图所示：



map方法是一对一的映射，每输入数据也只会输出一个值。flatMap方法是一对多的映射，对每个元素映射出来的仍是Stream。

* limit/skip，用于限制获取多少个结果，与数据库limit作用类似
* sorted函数需要传入实现Comparator函数式接口的对象，该接口的抽象方法compare接收两个参数并返回整型值，作用是排序
* distinct，用于剔除重复
* 其他操作，findFirst,min,max及reduce等

2）Terminal，流只能有一个Terminal操作，当这个操作执行后，流就被使用光了，无法再被操作，所以这必定是流的最后一个操作。当Terminal操作执行，才会真正开始流的遍历，并且会生成一个结果（或者side affect）。示例：

*array.stream().forEach(System.out::println);*

把System.out.println传递给forEach，其返回void的终端操作

* + forEach，消费流中每个元素并对其应用Lambda，操作返回void
  + count，返回流中元素的个数，这一操作返回long
  + collect，把流规约成一个集合，比如List,Map甚至是Integer

# **Optional**

Java程序中最常见的异常是NullPointerException，为了处理null类型，从而使代码充斥着深度嵌套的null检查，代码的可读性会很差。

Java 8中引入新的类java.util.Optional<T>，其封装Optional值的类，其使用如下所示：

*public static String getName(User u) {*

*if (u == null)*

*return "Unknown";*

*return u.name;*

*}*

对上面的类进行改写，

*public static String getName(User u) {*

*return Optional.ofNullable(u)*

*.map(user->user.name)*

*.orElse("Unknown");*

*}*

可以把Optional对象看成是特殊的集合数据，其至少包含一个元素，对于Map操作，如果Optional包含一个值那么就把该值作为参数传递给map，对该值进行转换。如果Optional为null，则什么都不做。在系统中使用如下：

*Optional<String> maybeCommand = cliArguments.stream()*

*.filter(arg -> executionContext.launcher().getCommandNames().contains(arg))*

*.findFirst();*

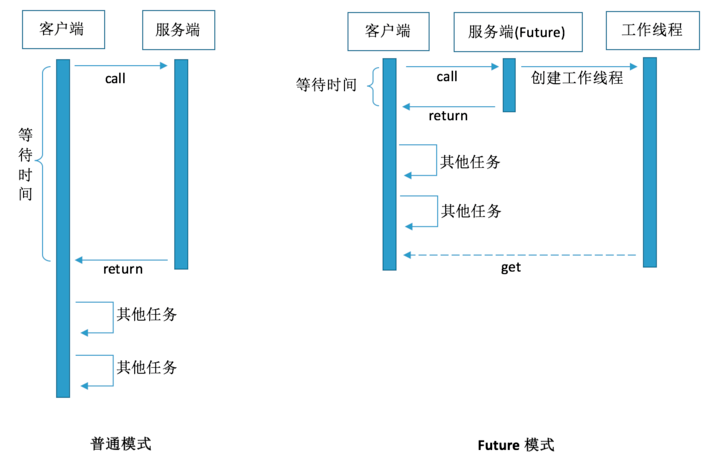
*if (! maybeCommand.isPresent()) {*

*ExecUtils.addArgument(cmd, "run");*

*}*

# **CompletableFuture**

在系统中为了避免等待远程服务或者对数据库的查询，而等到线程的执行，Java提供了Future接口来构建异步应用，如下：



其使用示例如下：

*ExecutorService executor = Executors.newCachedThreadPool();*

*Future<Double> future = executor.submit(new Callable<Double>() {*

*@Override*

*public Double call() throws Exception {*

*Thread.sleep(2000);*

*System.out.println("Callable ");*

*return 1.0;*

*}*

*});*

*Double result = future.get(3, TimeUnit.SECONDS);*

*System.out.println("result:"+result);*

在程序中通过ExecutorService调用Callable，通过Future#get方法获取执行结果，该方法会阻塞直到任务返回结果。程序输出如下：

*Callable*

*result:1.0*

## **Future分析**

在上例中，Future接口提供了方法来检测异步计算是否已经结束，其使用iDone方法，等待异步操作结束，以及获取计算的结果，其接口定义如下：

*public interface Future<V> {*

*boolean cancel(boolean mayInterruptIfRunning);*

*boolean isCancelled();*

*boolean isDone();*

*V get() throws InterruptedException, ExecutionException;*

*V get(long timeout, TimeUnit unit)*

*throws InterruptedException, ExecutionException, TimeoutException;*

*}*

ExecutorService返回的Future类为FutureTask，其返回调用如下：

*public <T> Future<T> ExecutorService#submit(Callable<T> task) {*

*if (task == null) throw new NullPointerException();*

*RunnableFuture<T> ftask = newTaskFor(task);*

*execute(ftask);*

*return ftask;*

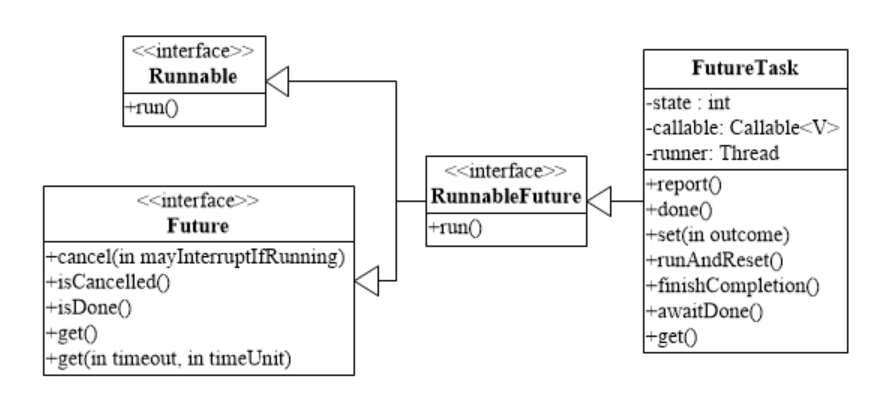
*}*

*protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {*

*return new FutureTask<T>(callable);*

*}*

类图如下所：



1）FutureTask的执行如下：

*public void run() {*

*try {*

*Callable<V> c = callable;*

*if (c != null && state == NEW) {*

*V result;*

*boolean ran;*

*try {*

*result = c.call(); //执行Callable*

*ran = true;*

*} .....*

*if (ran) //执行成功*

*set(result); //设置结果，及设置状态*

*}*

*}*

*}*

其中Result为请求返回值，其类型定义如下

*Future<Double> future = executor.submit(new Callable<Double>()*

返回值类型为Double。

2）Future.get

通过Future.get获取结果

*future.get(3, TimeUnit.SECONDS);*

其执行分析如下：

*public V get(long timeout, TimeUnit unit)*

*int s = state;*

*if (s <= COMPLETING &&*

*(s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING) //等待结束*

*throw new TimeoutException();*

*return report(s);*

*}*

调用get时会阻塞执行，其核心方法是awaitDone，执行如下：

*private int awaitDone(boolean timed, long nanos){*

*final long deadline = timed ? System.nanoTime() + nanos : 0L; //设置Deadline*

*WaitNode q = null;*

*boolean queued = false;*

*for (;;) { //循环查询结果*

*...*

*int s = state;*

*if (s > COMPLETING) { //获取状态，结束后则直接退出*

*if (q != null)*

*q.thread = null;*

*return s;*

*} ....*

*else if (timed) {*

*nanos = deadline - System.nanoTime(); //超时，则直接退出*

*if (nanos <= 0L) {*

*removeWaiter(q);*

*return state;*

*} }}*

## **CompletableFuture**

Future接口提供了方法来检测异步计算是否已经结束（isDone），等待异步操作结束，以及获取计算的结果，但是很难表述Future结果之间的依赖性，使用场景如下：

当长时间计算任务完成时，请将该计算的结果通知到另一个长时间运行的计算任务，这两个计算任务都完成后，将结果的结果通知到另一个长时间运行的计算任务，这两个计算任务都完成后，将计算结果与另一个查询操作结果合并。使用Future中提供的方法完成这样的操作又是另外一回事。其要实现的方式如下：

* 将两个异步计算合并为一个，这两个异步计算之间相互独立，同时第二个又依赖第一个结果
* 等待Future集合中所有任务都完成
* 仅等待Future集合中最快结束的任务完成，并返回它的结果
* 通过编程方式完成一个Future任务的执行
* 应对Future的完成事件（即当Future完成事件发生时会收到通知，并能使用Future计算的结果进行下一步操作，不只是简单的阻塞操作的结果）

Java 8提供新的CompletableFuture类（其实现Future接口）来实现上述需求。Stream和CompletableFuture的设计都实现了Lambda表达式以及流水线的思想，从这个角度看CompletableFuture和Future的关系就跟Stream和Collection的关系一样。

使用CompletableFuture需要异步API，如何将同步API改成异步API及响应式处理异步操作完成事件。

* 同步API，调用方在被调用方运行过程中等待，被调用方运行结束，调用方取得被调用方的返回值并继续运行。即使调用方和被调用方在不同线程中运行，调用还是需要等待被调用方结束运行，阻塞式调用
* 异步API，在调用后直接返回，或者至少在被调用方计算完成之前，将它剩余的计算任务交给另一个线程去做，该线程和调用方式异步的，这就是非阻塞式调用的由来。执行剩余计算任务的线程会将它的计算结果返回给调用方。返回的方式是通过回调函数，要么是由调用方再次执行一个等待，直到计算完成的方法调用。

下面基于CompletableFuture构建异步应用，创建名为“最佳价格查询器”的应用，其查询多个在线商店，依据给定的产品或者服务找出最低的价格。其使用如下：

*shops.stream()*

*.map(shop -> CompletableFuture.supplyAsync(() -> shop.getPrice(product), executor))*

*.map(future -> future.thenApply(Quote::parse))*

*.map(future -> future.thenCompose(*

*quote -> CompletableFuture.supplyAsync(() -> Discount.applyDiscount(quote),*

*executor)));*