JDK8新特性

Java8概述

```
Java8又称JDK1.8，是Java语言开发的一个主要版本。Oracle公司于2014年3月18日发布Java 8。
    支持Lambda表达式
    函数式接口
    新的Stream API
    新的日期 API
    其他特性
```

Lambda表达式

lambda表达式可以理解为一种匿名函数的代替，lambda允许将函数作为一个方法的参数（函数作为方法参数传
递），将代码像数据一样传递，目的是简化代码的编写。

```
Lambda表达式：特殊的匿名内部类，语法更简洁。

Lambda表达式允许把函数作为一个方法的参数（函数作为方法参数传递），将代码像数据一样传递。

Lambda表达式基本语法：
    <函数式接口><变量名> = (参数1，参数2...)->{
        // 方法体
    }

Lambda引入了新的操作符：->（箭头操作符），->将表达式分成两部分
    .左侧：(参数1，参数2) 表示参数列表
    .右侧：{}内部是方法体

注意事项：
    1、形参列表的数据类型会自动推断；
    2、如果形参列表为空，只需保留()；
    3、如果形参只有1个，()可以省略，只需要参数的名称即可；
    4、如果执行语句只有1句，且无返回值，{}可以省略，若有返回值，则若想省去{}，则必须同时省略
return，且执行语句
    也 保证只有1句；
    5、lambda不会生成一个单独的内部类文件；
    6、lambda表达式若访问了局部变量，则局部变量必须是final的，若是局部变量没有加final关键
字，系统会自动添加，此后在修改该局部变量，会报错。
```

```
package comd.it.demo;

/**
 * Lambda表达式的使用
 *
```

```
 * @author DELL
 *
 */
public class LambdaRunableDemo {
    public static void main(String[] args) {
        // 匿名内部类
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("runnable1子线程执行了.....");
            }
        };

        new Thread(runnable).start();

        // Lambda表达式
        Runnable runnable2 = () -> {
            System.out.println("runnable2子线程执行了.....");
        };

        new Thread(runnable2).start();

        // Lambda表达式简化
        Runnable runnable3 = () -> System.out.println("runnable3子线程执行
了.....");
        new Thread(runnable3).start();

        // Lambda表达式作为参数进行传递
        new Thread(()->System.out.println("runnable4子线程执行
了.....")).start();
    }
}
```

```
package comd.it.demo;

import java.util.Comparator;
import java.util.TreeSet;

public class LambdaComparatorDemo {
    public static void main(String[] args) {
        // 匿名内部类
        Comparator<String> com1 = new Comparator<String>() {

            @Override
            public int compare(String o1, String o2) {
                return o1.length() - o2.length();
            }
        };

        TreeSet<String> treeSet = new TreeSet<String>(com1);

        // Lambda表达式
        Comparator<String> com2 =(String o1, String o2)  ->{
            return o1.length() - o2.length();
        };

        // Lambda表达式简化1
```

```
        Comparator<String> com3 =(o1,o2)->{
            return o1.length() - o2.length();
        };

        // Lambda表达式简化2
        Comparator<String> com4 = (o1,o2)->o1.length() - o2.length();

        // 将Lambda表达式作为参数进行传递
        TreeSet<String> treeSet1 = new TreeSet<String>((o1,o2)->o1.length() -
o2.length());

    }
}
```

函数式接口

lambda表达式需要函数式接口的支持；

所谓函数式接口，是指只有一个抽象方法；

如果一个接口只有一个抽象方法，则该接口称之为函数式接口，函数式接口可以使用lambda表达式，lambda表达式会被匹配到这个抽象方法上。

另外JDK8也提供了一个注解，帮助我们编译时检查语法是否符合函数式接口，即@FunctionInterface

匿名内部类

```
public interface Usb {
    void service();
}


public class TestUsb {
    public static void main(String[] args) {
        // 匿名内部类
        Usb mouse = new Usb() {

            @Override
            public void service() {
                System.out.println("鼠标开始工作了");
            }
        };

        run(mouse);
    }

    public static void run(Usb usb) {
        usb.service();
    }
}
```

lambda表达式

```
/**
 * 函数式接口
 * @author DELL
 *
 */
@FunctionalInterface
public interface Usb {
    void service();
}
```

```
public class TestUsb {
    public static void main(String[] args) {
        // 匿名内部类
        Usb mouse = new Usb() {

            @Override
            public void service() {
                System.out.println("鼠标开始工作了");
            }
        };

        run(mouse);


        // Lambda表达式
        Usb fan = ()->{
            System.out.println("U盘开始工作了");
        };
        run(fan);

        // Lambda表达式:只有一条语句时，{}可省略
        Usb shan = ()->System.out.println("电脑风扇开始工作了");
        run(shan);
    }



    public static void run(Usb usb) {
        usb.service();
    }
}
```

常见函数式接口

函数式接口	参数类型	返回类型	说明
Consumer<T> 消费型接口	T	void	void accept(T t);对类型为T的对象应用操作
Supplier<T> 供给型接口	无	T	T get(); 返回类型为T的对象
Function<T,R> 函数型接口	T	R	R apply(T t);对类型为T的对象应用操作，并返回类型为R类型的对象。
Predicate<T> 断言型接口	T	boolean	boolean test(T t);确定类型为T的对象是否满足条件，并返回boolean类型。

Consumer 消费型接口

```
package comd.it.demo;

import java.util.function.Consumer;

public class ConsumerDemo {

    public static void main(String[] args) {
        // 匿名内部类
        Consumer<Double> consumer = new Consumer<Double>() {
            @Override
            public void accept(Double t) {
                System.out.println("消费聚餐: "+t);
            }
        };

        happy(consumer, 1000);

        //  Lambda表达式
        Consumer<Double> consumer1 = t ->{
            System.out.println("消费聚餐: "+t);
        };

        happy(consumer1, 1200);

        // Lambda表达式:只有一条语句时，{}可省略
        Consumer<Double> consumer2 = t ->System.out.println("消费聚餐: "+t);
        happy(consumer2, 1300);


        // Lambda表达式作为参数传递
        happy(t ->System.out.println("消费聚餐: "+t), 1400);
    }

    // Consumer 消费型接口
    public static void happy(Consumer<Double> consumer,double money) {
        consumer.accept(money);
    }
}
```

Supplier 供给型接口

```
import java.util.Arrays;
import java.util.Random;
import java.util.function.Supplier;

public class SupplierDemo {
    public static void main(String[] args) {
        int[] arr = getNums(()->new Random().nextInt(100), 5);
        System.out.println(Arrays.toString(arr));
    }

    // Supplier
    public static int[] getNums(Supplier<Integer> supplier,int count) {
        int[] arr = new int[count];
```

```
        for (int i = 0; i < arr.length; i++) {
            arr[i] = supplier.get();
        }
        return arr;
    }
}
```

Function函数型接口

```
import java.util.function.Function;

public class FunctionDemo {
    public static void main(String[] args) {
        String result = handler(s->s.toUpperCase(), "abc");
        System.out.println(result);
    }

    // Function函数型接口
    public static String handler(Function<String, String> function,String str) {
        return function.apply(str);
    }
}
```

Predicate 断言型接口

```
import java.util.ArrayList;
import java.util.List;

public class PredicateDemo {

    public static void main(String[] args) {
        List<String> names = new ArrayList<String>();
        names.add("zhangsan");
        names.add("lisi");
        names.add("wangwu");
        names.add("zhaoliu");
        names.add("zhangwuji");
        List<String> result = filterNames(s->s.startsWith("zhang"), names);
        System.out.println(result.toString());
    }

    // 断言型接口
    public static List<String> filterNames(java.util.function.Predicate<String>
predicate, List<String> list) {
        List<String> resultList = new ArrayList<String>();
        for (String str : list) {
            if(predicate.test(str)) {
                resultList.add(str);
            }
        }

        return resultList;
    }
}
```

方法引用

方法引用是Lambda表达式的一种简写形式。如果Lambda表达式方法体中只是调用一个特定的已经存在的方法，则可以使用方法引用。

常见形式：
对象::实例方法
类::静态方法
类::实例方法
类::new

```
public class Employee {
    private String name;
    private double money;


    public Employee() {

    }


    public Employee(String name, double money) {
        this.name = name;
        this.money = money;
    }


    public String getName() {
        return name;
    }


    public void setName(String name) {
        this.name = name;
    }


    public double getMoney() {
        return money;
    }


    public void setMoney(double money) {
        this.money = money;
    }


    @Override
    public String toString() {
        return "Employee [name=" + name + ", money=" + money + "]";
    }
}
```

```
import java.util.Comparator;
import java.util.function.Consumer;
```

```
import java.util.function.Function;
import java.util.function.Supplier;

/**
 * 方法使用
 * 1、对象::实例方法
 * 2、类::静态方法
 * 3、类::实例方法
 * 4、类::new
 * @author DELL
 *
 */
public class Demo04 {
    public static void main(String[] args) {
        // 1、对象::实例方法
        Consumer<String> consumer = s->System.out.println(s);
        consumer.accept("hello");

        Consumer<String> consumer2 = System.out::println;
        consumer2.accept("world");

        // 2、类::静态方法
        Comparator<Integer> com = (o1,o2)->Integer.compare(o1, o2);

        // 使用的机会不是很高
        Comparator<Integer> com2 = Integer::compare;


        // 3、类::实例方法
        Function<Employee, String> function = e -> e.getName();
        System.out.println(function.apply(new Employee("小明", 10000)));

        Function<Employee, String> function2 = Employee::getName;
        System.out.println(function2.apply(new Employee("小刀", 100000)));

        // 4、类::new
        Supplier<Employee> supplier = ()->new Employee("liming",8000);
        Employee emp1 = supplier.get();
        System.out.println(emp1.toString());

        Supplier<Employee> supplier2 = Employee::new;
        Employee emp2 = supplier2.get();
        System.out.println(emp2.toString());

    }
}
```

流式编程-StreamAPI

什么是Stream

流（Stream）中保存对集合或数组数据的操作。和集合类似，但集合中保存的是数据。

Stream是Java8中处理数组、集合的抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过
滤和映射数据等操作。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。
一个Stream表面上与一个集合很类似，集合中保存的是数据，而流设置的是对数据的操作。

```
Stream的特点：
    1，Stream 自己不会存储元素。
    2，Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。
    3，Stream 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。
Stream遵循“做什么，而不是怎么去做”的原则。只需要描述需要做什么，而不用考虑程序是怎样实现的。
```

快速体验StreamAPI的特点

```
import java.util.ArrayList;
import java.util.List;

public class StreamDemo {
    public static void main(String[] args) {
        List<String> data = new ArrayList<>();
        data.add("hello");
        data.add("stream");
        data.add("good night~");

        // 传统做法
        long count = 0;
        for (String str : data) {
            if(str.length() > 3) {
                count++;
            }
        }
        System.out.println(count);

        // Stream结合Lambda表达式
        // 链式编程
        long count2 = data.stream().filter(s->s.length()>3).count();
        System.out.println(count2);
    }
}
```

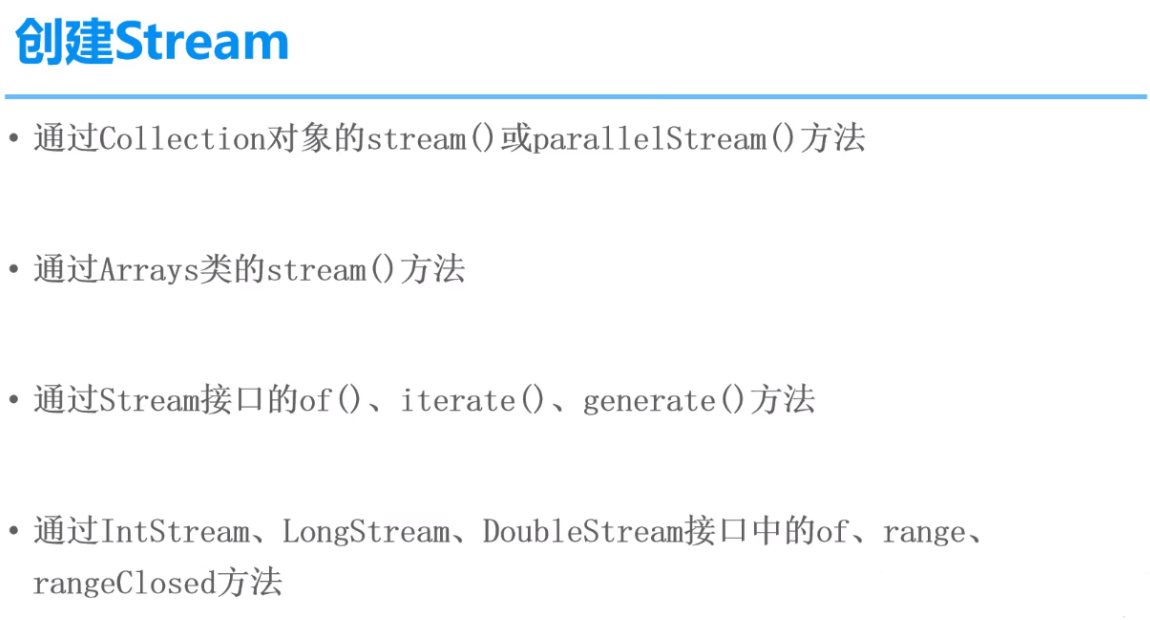
使用StreamAPI的步骤

Stream使用步骤

- 创建
  - 新建一个流
- 中间操作
  - 在一个或多个步骤中，将初始Stream转化到另一个Stream的中间操作
- 终止操作
  - 使用一个终止操作来产生一个结果。该操作会强制它之前的延迟操作立即执行。在这之后，该Stream就不能使用了。

1. 创建一个Stream。 （创建）
2. 在一个或多个步骤中，将初始Stream转化到另一个Stream的中间操作。 （中间操作）
3. 使用一个终止操作来产生一个结果。该操作会强制他之前的延迟操作立即执行。在这之后，该Stream就不会再被使用
了。（终止操作）

创建Stream



1、Collection对象中的stream()方法

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class StreamDemo {
    public static void main(String[] args) {

        // 1、Collection对象中的stream()方法
        List<String> data = new ArrayList<>();
        data.add("hello");
        data.add("stream");
        data.add("good night~");

        Stream<String> stream = data.stream();
        // 遍历
        //stream.forEach(s->System.out.println(s));
        // 方法引用
        stream.forEach(System.out::println);
    }
}
```

2、Collection对象中的parallelStream()方法

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class StreamDemo {
```

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

        // 1、Collection对象中的parallelStream()方法
        List<String> data = new ArrayList<>();
        data.add("hello");
        data.add("stream");
        data.add("good night~");

        Stream<String> stream = data.parallelStream();
        // 遍历
        //stream.forEach(s->System.out.println(s));
        // 方法引用
        stream.forEach(System.out::println);
    }
}
```

3、Arrays工具类的stream方法

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamDemo {
    public static void main(String[] args) {
        // 2、Arrays工具类的stream方法
        String[] arr = {"aaa","bbb","ccc"};
        Stream<String> stream = Arrays.stream(arr);
        stream.forEach(System.out::println);
    }
}
```

4、Stream接口中的of方法、iterate、generate

```
import java.util.Random;
import java.util.stream.Stream;

public class StreamDemo {
    public static void main(String[] args) {
        // 4、Stream接口中的of方法、iterate、generate
        Stream<Integer> stream = Stream.of(10,20,30,40,50);
        stream.forEach(System.out::println);
        System.out.println("------------------迭代流------------------");
        //迭代流
        Stream<Integer> iterate = Stream.iterate(0, x->x+2);
        iterate.limit(10).forEach(System.out::println);

        // 生成流
        System.out.println("------------------生成流------------------");
        Stream<Integer> generate= Stream.generate(()->new
Random().nextInt(100));
        generate.limit(10).forEach(System.out::println);
    }
}
```

5、IntStream、LongStream、DoubleStream接口中的of、range、rangeClosed方法

```
import java.util.stream.IntStream;
```

```
public class StreamDemo {
    public static void main(String[] args) {
        // 5、IntStream、LongStream、DoubleStream接口中的of、range、rangeClosed方法
        IntStream intStream = IntStream.of(100,200,300);
        intStream.forEach(System.out::println);


        IntStream range = IntStream.range(0, 50);
        range.forEach(System.out::println);

        IntStream rangeClosed = IntStream.rangeClosed(0, 50);
        rangeClosed.forEach(System.out::println);
    }
}
```

中间操作

```
中间操作
    filter、limit、skip、distinct、sorted
    map
    parallel
```

filter、limit、skip、distinct、sorted

```
public class Employee {
    private String name;
    private double money;


    public Employee() {

    }


    public Employee(String name, double money) {
        this.name = name;
        this.money = money;
    }


    public String getName() {
        return name;
    }


    public void setName(String name) {
        this.name = name;
    }


    public double getMoney() {
        return money;
    }
```

```
    public void setMoney(double money) {
        this.money = money;
    }


    @Override
    public String toString() {
        return "Employee [name=" + name + ", money=" + money + "]";
    }


    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        long temp;
        temp = Double.doubleToLongBits(money);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }

    // 重写hashCode和equals
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (Double.doubleToLongBits(money) !=
Double.doubleToLongBits(other.money))
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class StreamMiddleOperation {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee("小王", 15000));
        empList.add(new Employee("小张", 12000));
        empList.add(new Employee("小李", 11000));
        empList.add(new Employee("小孙", 13000));
        empList.add(new Employee("小赵", 14000));
        //empList.add(new Employee("小赵", 14000));
        // 中间操作
```

```
        // filter过滤
        System.out.println("---------filter过滤---------");
        empList.stream().filter(e->e.getMoney()>14000).forEach(System.out::println);

        // limit限制
        System.out.println("---------limit限制---------");
        empList.stream().limit(2).forEach(System.out::println);

        // skip 跳过
        System.out.println("----------skip 跳过--------");
        empList.stream().skip(2).forEach(System.out::println);

        // distinct 去重,必须重写hashCode和equals
        System.out.println("----------distinct 去重--------");
        empList.stream().distinct().forEach(System.out::println);

        // sorted 排序
        System.out.println("----------sorted 排序--------");
        empList.stream().sorted((e1,e2)->Double.compare(e1.getMoney(),e2.getMoney())).forEach(System.out::println);
    }
}
```

map、parallel

```
import java.util.ArrayList;
import java.util.List;

public class StreamMiddleOperation {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee("小王", 15000));
        empList.add(new Employee("小张", 12000));
        empList.add(new Employee("小李", 11000));
        empList.add(new Employee("小孙", 13000));
        empList.add(new Employee("小赵", 14000));

        // 中间操作
        // map

        System.out.println("----------map--------");
        empList.stream().map(e->e.getName()).forEach(System.out::println);

        // parallel 采用多线程
        empList.parallelStream().forEach(System.out::println);

    }
}
```

串行流

```
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class Demo7 {
```

```
    public static void main(String[] args) {
        // 串行流和并行流的区别
        List<String> list = new ArrayList<String>();
        for (int i = 0; i < 5000000; i++) {
            list.add(UUID.randomUUID().toString());
        }

        // 串行流
        long start = System.currentTimeMillis();
        long count = list.stream().sorted().count();
        System.out.println(count);
        long end = System.currentTimeMillis();

        System.out.println("用时："+(end - start));
    }

}
```

并行流

```
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class Demo7 {
    public static void main(String[] args) {
        // 串行流和并行流的区别
        List<String> list = new ArrayList<String>();
        for (int i = 0; i < 5000000; i++) {
            list.add(UUID.randomUUID().toString());
        }

        // 并行流
        long start = System.currentTimeMillis();
        long count = list.parallelStream().sorted().count();
        System.out.println(count);
        long end = System.currentTimeMillis();

        System.out.println("用时："+(end - start));
    }
}
```

终止操作

```
终止操作
    forEach、min、max、count
    reduce、collect
```

forEach

```
import java.util.ArrayList;
import java.util.List;

public class TerminateOperateion {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
```

```
        empList.add(new Employee("小王", 15000));
        empList.add(new Employee("小张", 12000));
        empList.add(new Employee("小李", 11000));
        empList.add(new Employee("小孙", 13000));
        empList.add(new Employee("小赵", 14000));

        // 终止操作 forEach
        //empList.stream().filter(e->e.getMoney() >
12000).forEach(System.out::println);
        empList.stream().filter(
                e->{
                    System.out.println("----过滤了----");
                    return e.getMoney() > 12000;
                    });
                // 注释forEach发现代码没执行
                //.forEach(System.out::println);

    }
}
```

min max count reduce规约 collect

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class TerminateOperateion {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee("小王", 15000));
        empList.add(new Employee("小张", 12000));
        empList.add(new Employee("小李", 11000));
        empList.add(new Employee("小孙", 13000));
        empList.add(new Employee("小赵", 14000));


        // min max count
        Optional<Employee> min = empList.stream().min((e1,e2)-
>Double.compare(e1.getMoney(), e2.getMoney()));
        System.out.println(min.get());

        Optional<Employee> max = empList.stream().max((e1,e2)-
>Double.compare(e1.getMoney(), e2.getMoney()));
        System.out.println(max.get());

        long count = empList.stream().count();
        System.out.println("员工总个数: "+count);

        // reduce规约
        // 计算所有员工的工资
        Optional<Double> sum = empList.stream().map(e-
>e.getMoney()).reduce((x,y)->x+y);
        System.out.println(sum.get());

        // collect 收集
```

```
        List<String> names = empList.stream().map(e-
>e.getName()).collect(Collectors.toList());
        for (String str : names) {
            System.out.println(str);
        }

    }
}
```

新时间API

之前时间API存在问题：线程安全问题、设计混乱

线程安全问题

```
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
/**
 * 线程安全问题
 * @author DELL
 *
 */
public class Demo8 {
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
        ExecutorService pool = Executors.newFixedThreadPool(10);
        Callable<Date> callable = new Callable<Date>() {
            @Override
            public Date call() throws Exception {
                return sdf.parse("20200425");
            }
        };
        List<Future> list = new ArrayList<Future>();

        for (int i = 0; i < 10; i++) {
            Future<Date> future = pool.submit(callable);
            list.add(future);
        }


        for (Future future : list) {
            try {
                System.out.println(future.get());
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (ExecutionException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
```

```
            }
        }

        pool.shutdown();
    }
}
```

```
java.util.concurrent.ExecutionException: java.lang.NumberFormatException: For input string: ""
        at java.util.concurrent.FutureTask.report(Unknown Source)
        at java.util.concurrent.FutureTask.get(Unknown Source)
        at comd.it.demo.Demo8.main(Demo8.java:37)
Caused by: java.lang.NumberFormatException: For input string: ""
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Long.parseLong(Unknown Source)
        at java.lang.Long.parseLong(Unknown Source)
        at java.text.DigitList.getLong(Unknown Source)
        at java.text.DecimalFormat.parse(Unknown Source)
        at java.text.SimpleDateFormat.subParse(Unknown Source)
        at java.text.SimpleDateFormat.parse(Unknown Source)
        at java.text.DateFormat.parse(Unknown Source)
```

解决办法1：加锁

```
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
/**
 * 线程安全问题
 * @author DELL
 *
 */
public class Demo8 {
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
        ExecutorService pool = Executors.newFixedThreadPool(10);
        Callable<Date> callable = new Callable<Date>() {
            @Override
            public Date call() throws Exception {
                synchronized (sdf) {
                    return sdf.parse("20200425");
                }
            }
        };
        List<Future> list = new ArrayList<Future>();

        for (int i = 0; i < 10; i++) {
            Future<Date> future = pool.submit(callable);
            list.add(future);
        }


        for (Future future : list) {
            try {
                System.out.println(future.get());
            } catch (InterruptedException e) {
```

```
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (ExecutionException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        pool.shutdown();
    }
}
```

解决办法2：新的时间类

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
/**
 * 线程安全问题解决方法 LocalDate
 * @author DELL
 *
 */
public class Demo8 {
    public static void main(String[] args) {
        //SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyyMMdd");
        ExecutorService pool = Executors.newFixedThreadPool(10);
        Callable<LocalDate> callable = new Callable<LocalDate>() {
            @Override
            public LocalDate call() throws Exception {
                    return LocalDate.parse("20200425",dtf);
            }
        };
        List<Future<LocalDate>> list = new ArrayList<Future<LocalDate>>();

        for (int i = 0; i < 10; i++) {
            Future<LocalDate> future = pool.submit(callable);
            list.add(future);
        }


        for (Future<LocalDate> future : list) {
            try {
                System.out.println(future.get());
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (ExecutionException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
```

```
        }

        pool.shutdown();
    }
}
```

本地化时间日期API

```
LocalDate
LocalTime
LocalDateTime
Instant:时间戳
ZoneId:时区
```

```
/**
 * LocalDateTim
 * @author DELL
 *
 */
public class Demo8 {
    public static void main(String[] args) {
      // 创建本地时间
      LocalDateTime localDateTime = LocalDateTime.now();
      //LocalDateTime localDateTime2 = LocalDateTime.of(year, month, dayOfMonth,
hour, minute)
      System.out.println(localDateTime);
      System.out.println(localDateTime.getYear());

      // 添加2天
      LocalDateTime plusDays = localDateTime.plusDays(2);
      System.out.println(plusDays);

      // 减少一个月
      LocalDateTime minusMinutes = localDateTime.minusMonths(1);
      System.out.println(minusMinutes);
    }
}
```

```
import java.time.Duration;
import java.time.Instant;
import java.time.ZoneId;
import java.util.Set;

/**
 * Instant:时间戳
 * ZoneId:时区
 * @author DELL
 *
 */
public class Demo9 {
    public static void main(String[] args) {
        // Instant:时间戳
        Instant instant = Instant.now();
        System.out.println(instant.toString());
        System.out.println(instant.toEpochMilli());
```

```
        System.out.println(System.currentTimeMillis());

        // 添加、减少时间
        Instant plusSeconds = instant.plusSeconds(10);

        System.out.println(Duration.between(instant, plusSeconds).toMillis());

        //ZoneId:时区
        Set<String> availableZoneIds = ZoneId.getAvailableZoneIds();
        for (String str : availableZoneIds) {
            // 遍历所有的时区
            System.out.println(str);
        }

        System.out.println("获取当前时区-->"+ZoneId.systemDefault().toString());
    }
}
```

Date Instant LocalDateTime的转换

```
import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Date;
/**
 * Instant:时间戳
 * ZoneId:时区
 * @author DELL
 *
 */
public class Demo9 {
    public static void main(String[] args) {
        // Date --> Instant -->LocalDateTime
        Date date = new Date();
        Instant instant = date.toInstant();
        System.out.println(instant);

        LocalDateTime localDateTime = LocalDateTime.ofInstant(instant,
ZoneId.systemDefault());
        System.out.println(localDateTime);

        // LocalDateTime --> Instant -->Date
        Instant instant2 =
localDateTime.atZone(ZoneId.systemDefault()).toInstant();
        System.out.println(instant2);
        Date from = Date.from(instant2);
        System.out.println(from);
    }
}
```

DateTimeFormatter:格式化类

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
/**
 * DateTimeFormatter的使用
```

```
 *
 */
public class Demo9 {
    public static void main(String[] args) {
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
        // 把时间格式化字符串
        String format = dtf.format(LocalDateTime.now());
        System.out.println(format);

        // 把字符串解析成时间
        LocalDateTime parse = LocalDateTime.parse("2020-08-19 18:48:13",dtf);
        System.out.println(parse);
    }
}
```